



# 语义分析与中间代码生成

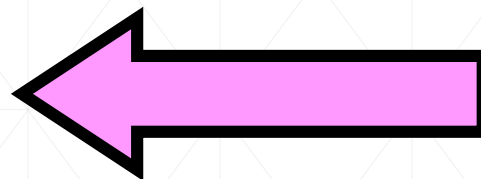


# 第 6 章 语义分析与中间代码生成



6.1 语法制导翻译

6.2 符号表



6.3 类型检查

6.4 中间语言

6.5 中间代码生成



## ■ 源程序流基本组成单位

固定单词(关键字、运算符、界限符...):

表示语句性质，反映语句结构；

用户定义的单词—标识符:

表示程序中各种实体；

如，变量名；常量名；标号名；过程名；函数名；文件名，数组名 ...

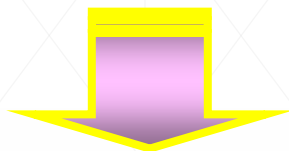


**一般变量:** name , type , addr , level ...

**数组:** 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



## ■ 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。



例如，设有C程序片段

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

词法分析  
语法分析  
语义分析

**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

.....

类型检查;

数组越界检查;

回填addr; ...



## ■ 常见标识符类的主要属性与作用

1. 标识符名。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。
4. 作用域、可视性。作用：动态活动环境支持，提供量所在层次；

# 第 6 章 语义分析与中间代码生成



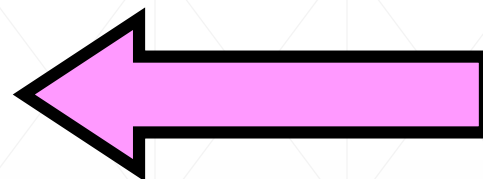
6.1 语法制导翻译

6.2 符号表

6.3 类型检查

6.4 中间语言

6.5 中间代码生成







语义处理的一个功能:

检查不能用上下文无关描述的上下文有关的结构是否正确

类型检查的要素

类型描述

—类型表达式

关于类型的处理

—类型说明语句中的类型处理

—表达式中的类型处理



## 类型表达式

1. 基本类型, `int`, `char`, `float`, ...

2. 构造类型

数组

记录

函数

指针



## 关于类型的处理

1. 类型说明语句中的类型处理  
将类型表达式信息填入符号表

2. 表达式或语句中的类型处理  
表达式:

基本类型: 获取类型

构造类型: 检查, 没错误获取类型

语句:

检查是否匹配,

不匹配是否可以类型转换,

可以转换的实施转换

后序说明语句  
的处理中讲



## 表达式中的类型处理

### 1. 简单表达式

$E \rightarrow \text{字符常数}$        $E.type = \text{char}$

$E \rightarrow \text{整型常数}$        $E.type = \text{int}$

$E \rightarrow i$        $E.type = \text{lookup}(i.entry)$

$E \rightarrow E[E]$        $\text{if}(E_2.type == \text{int} \ \& \ E_1.type == \text{array}(s, t))$   
                           $\text{then } E.type = t,$   
                           $\text{else } type\_error$



# 语句中的类型处理

## 2.一般语句

产生式	语义动作
$S \rightarrow i = E$	<b>if</b> ( <i>lookup</i> ( <i>i.entry</i> )== <i>E.type</i> ) <b>then</b> <i>S.type</i> =void, <b>else</b> <b>type_error</b>
$S \rightarrow \text{if } E \text{ then } S$	<b>if</b> ( <i>E.type</i> ==boolean) <b>then</b> <i>S.type</i> =void, <b>else</b> <b>type_error</b>

# 第 6 章 语义分析与中间代码生成



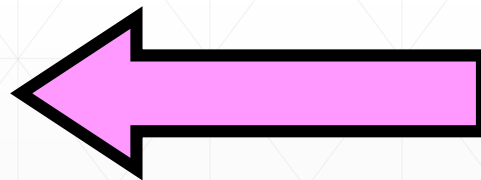
6.1 语法制导翻译

6.2 符号表

6.3 类型检查

6.4 中间语言

6.5 中间代码生成





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考虑的要素:

- 从源语言到目标语言的翻译跨度
- 目标机的指令集特点。

中间语言 { 树(AST)  
逆波兰式  
N元式(三元式、间接三元式四元式)



## 一.抽象语法树(Abstract\_SyntaxTree,AST)

语法结构的一种简介的树形表示形式，  
只包括该结构尚需转换或归约的信息。

简单算术表达式文法 $G(E)$ :

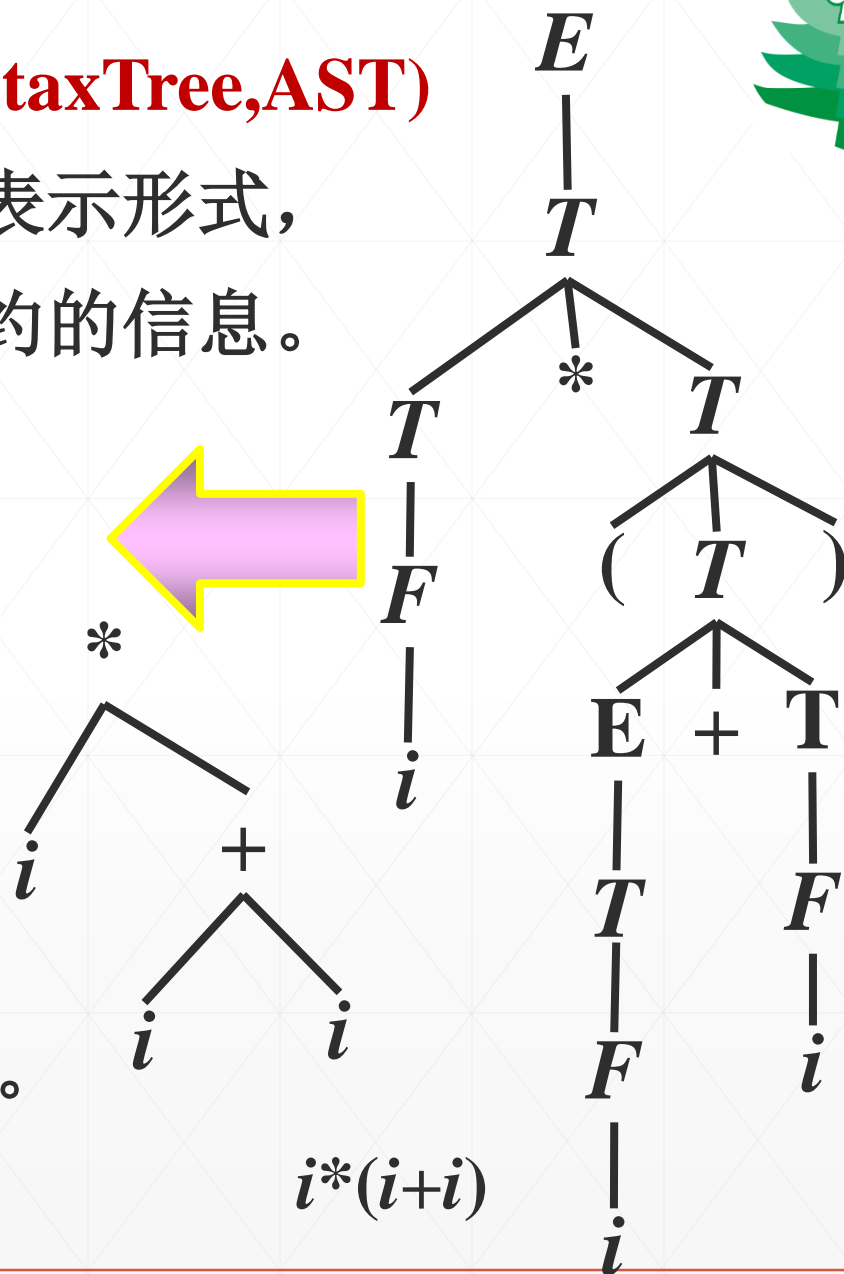
$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid i$$

语法分析树庞大的原因:

每步推导都在语法树有反映。

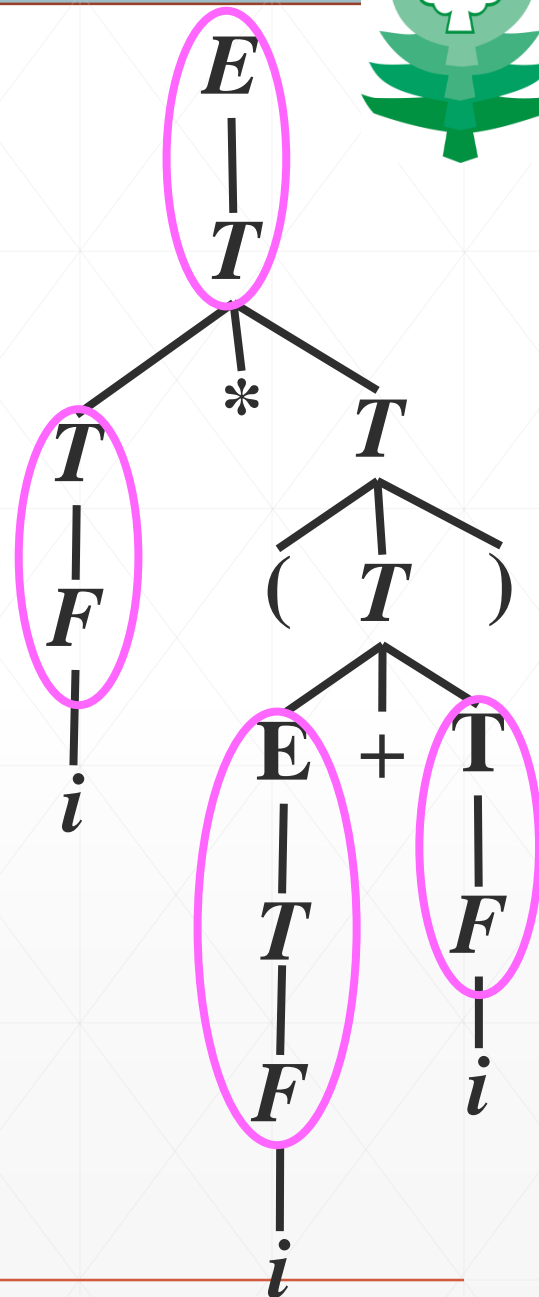
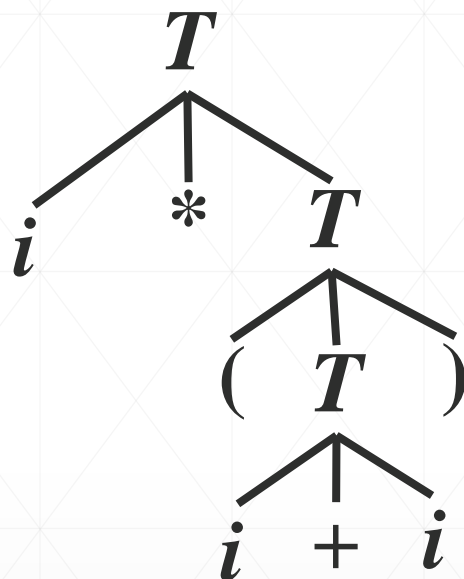






## 从语法分析树构造AST的方法

1. 去掉候选式为单一非终结符号(单非产生式)的相关子树，并上提相关分支上的终结符号结点。特别，若根节点开始就是单非产生式，则去掉根节点及其连线。

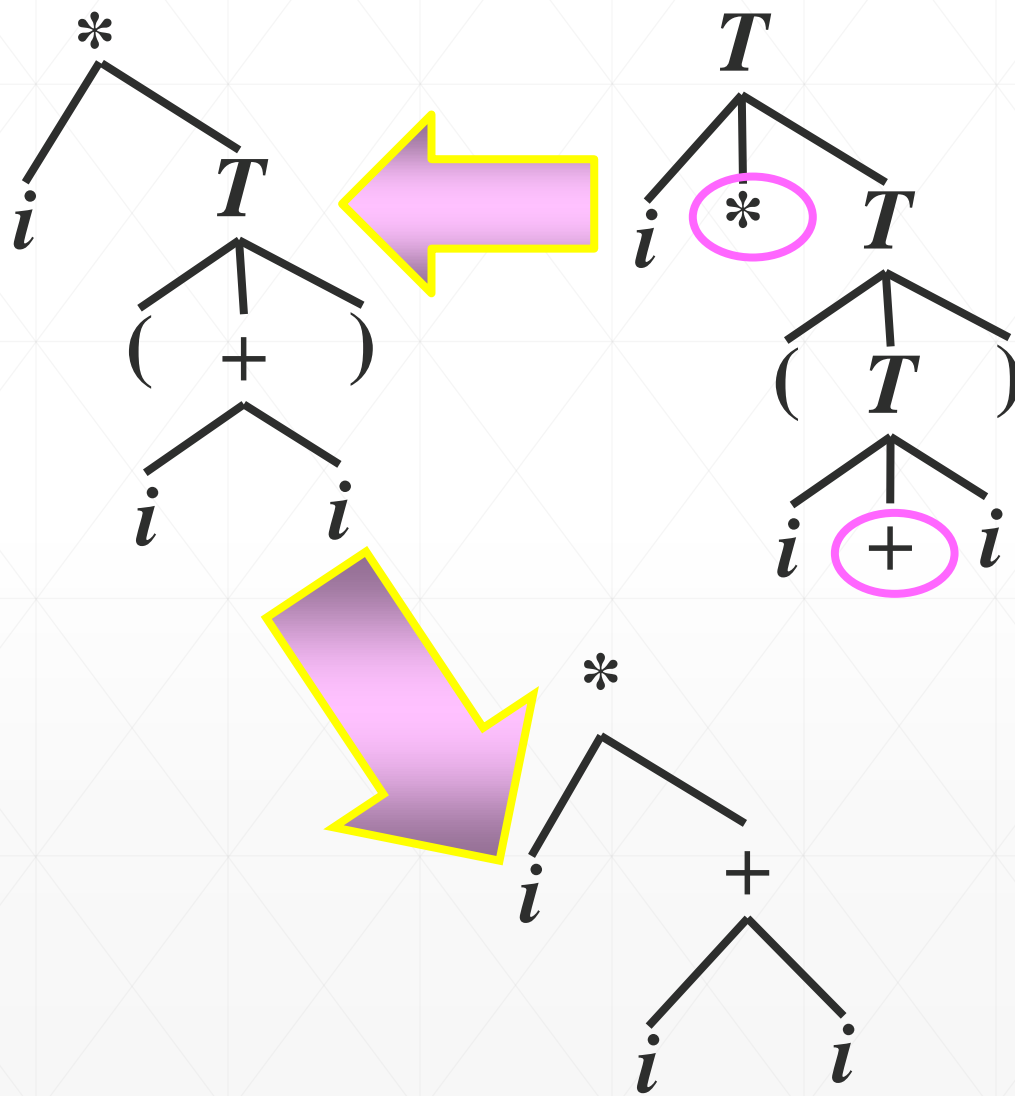




## 从语法分析构造AST的方法

2. 包含 **运算** 的子树，上提运算符并让它代替其父节点。

3. 去掉括号，并上提运算符





## 综述

AST的中间结点是运算符，  
叶子结点为简单变量或常数

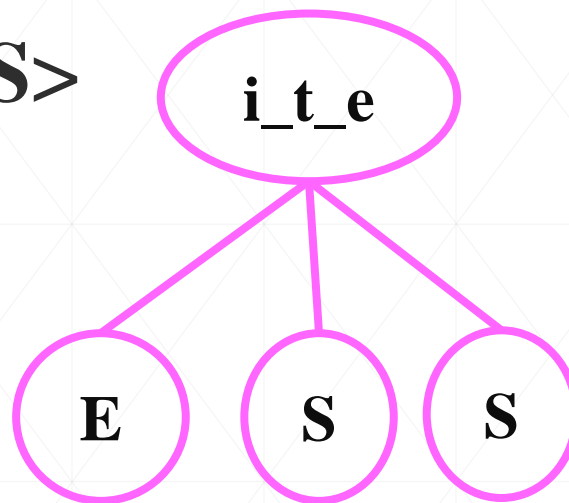
运算符号可以是任何单目、双目或 $n$ 目运算符

拓广：

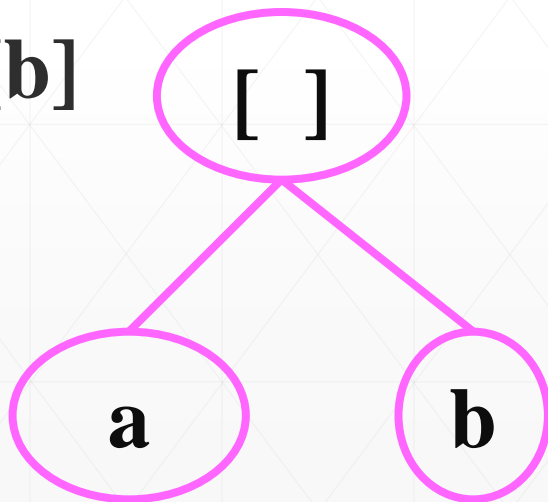
其他结构的表达式，只要约定了运算符、运算对象，就确定了AST表示的形式



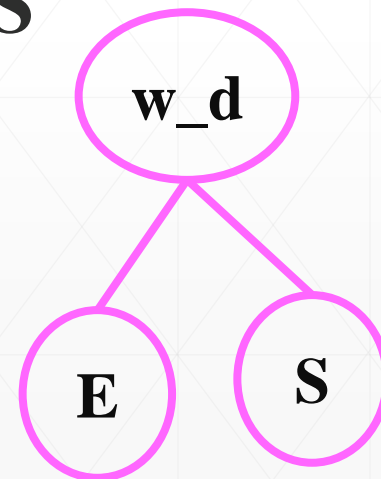
**if<E>then<S>else<S>**



**a[b]**

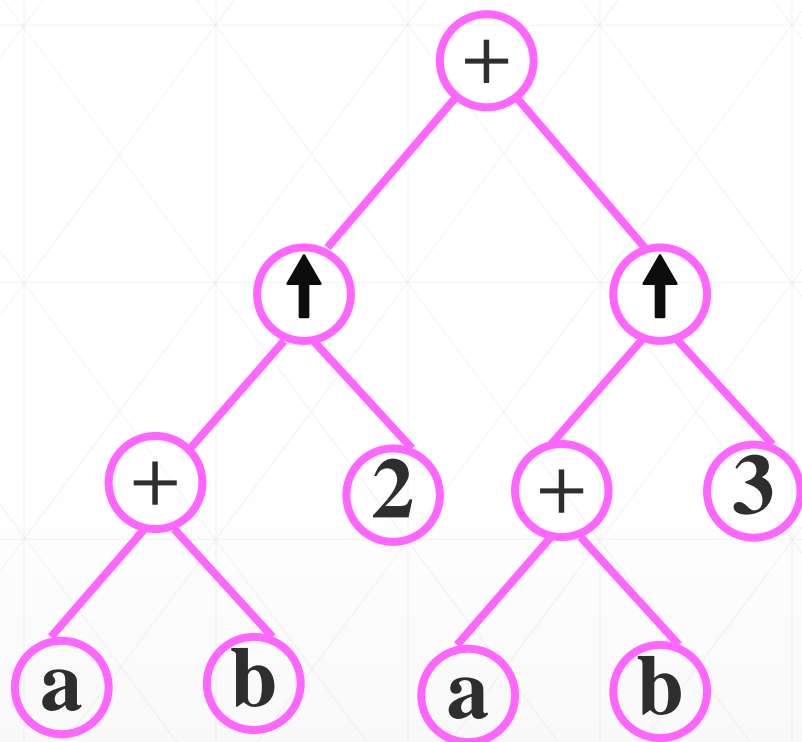


**while<E>S**





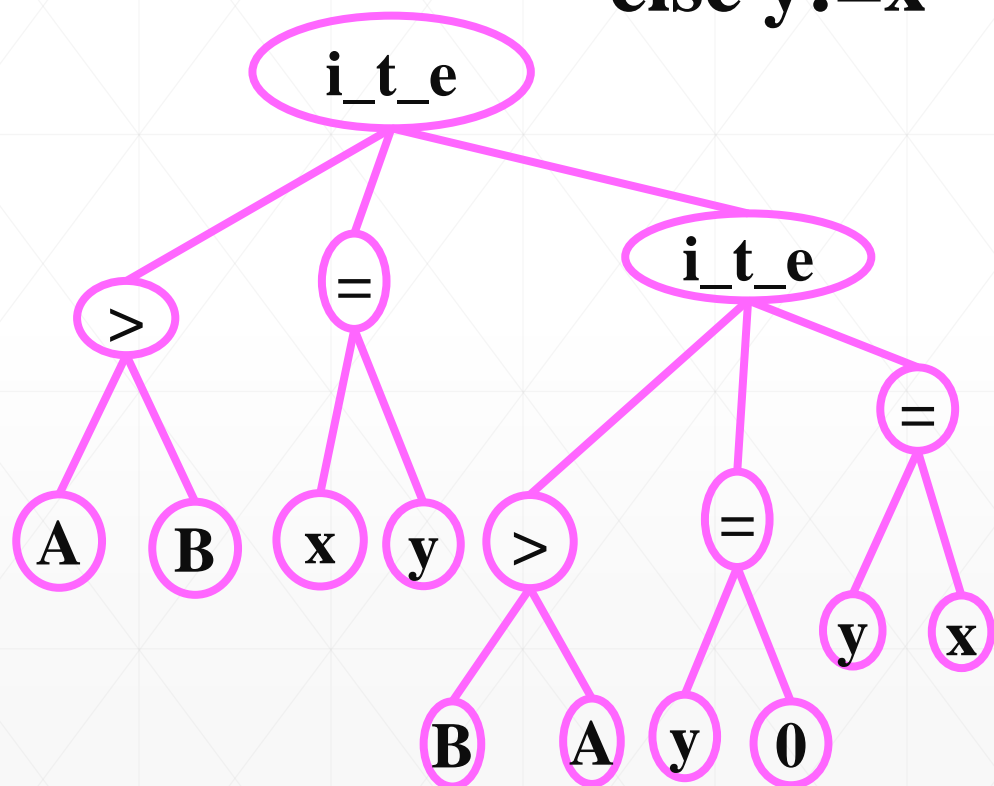
$$(a+b)^2 + (a+b)^3$$



if A>B then x=y

else if B>A then y=0

else y:=x





## 二. 逆波兰式(后缀式)

波兰逻辑学家卢卡西维奇发明

形式:

$$e_1 e_2 \dots e_n \theta \quad (n \geq 1)$$



运算对象      运算符

为了使中间代码更接近机器语言，在逆波兰式和多元式中，我们把结构化的程序块都转换为goto运算。



例:

$a \times b \Rightarrow ab\times$

$-a \Rightarrow a@$  (@表示单目-)

$a+b \times (c+d) \times (e+f) \Rightarrow a\ b\ c\ d\ +\times e\ f\ +\times +$

$\text{GOTO label} \Rightarrow label\ J$

**J:** 单目操作符，产生一个到运算对象的转移。

$\text{IF } e\ \text{GOTO label} \Rightarrow e\ label\ J_T$

**J<sub>T</sub>:** 二目操作符，如果第一运算对象的值为假，则产生一个到第二运算对象(标号)的转移。

**J<sub>F</sub>:** 二目操作符，如果第一运算对象的值为假，则产生一个到第二运算对象(标号)的转移。

**Label1**

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

**Label2**

(1)<expr> (4) J<sub>F</sub>

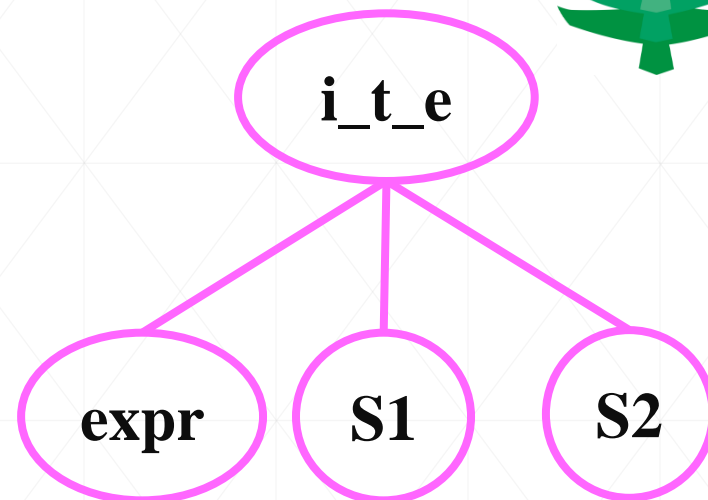
(2)<s1>

(3)(5) J

(4)<s2>

(5)

编号







**例：**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;  
}
```

逆波兰式代码：

(1) i j 0 ==

(2) k 100 =

(3) k i j + > (7) J<sub>F</sub>

(4) k k 1 - =

(5) i i 1 + =

(6) (8) J

(7) k i 2\*j2\*- =

(8) (2) J

编号



### 三. N元式      N个域的记录结构

$(D_1, D_2, D_3, \dots, D_n)$

OP域

操作对象域

#### ■ 常见N元式

三元式、间接三元式  $\longleftrightarrow$  双地址机指令形式

四元式  $\longleftrightarrow$  三地址机指令形式



## 三元式形式:

$$(OP, ARG_1, ARG_2)$$

其中:

**OP**: 操作符;

**ARG1**和**ARG2**: 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:

间接码表 + 三元式表



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)	=	X	(2)



**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)	J <sub>F</sub>	(1)	(5)
(3)	=	Z	X
(4)	J		(7)
(5)	+	Y	1
(6)	=	Z	(5)
(7)			



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

NO	OP	ARG1	ARG2
①	<u>+</u>	<u>a</u>	<u>b</u>
②	↑	①	2
③	<u>+</u>	<u>a</u>	<u>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



三元式表

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



## 四元式形式定义:

( OP , ARG<sub>1</sub> , ARG<sub>2</sub> , Result )

$(a+b)^2 + (a+b)^3$ 的四元式:

(0)	+	a,	b,	T <sub>1</sub>
(1)	↑	T <sub>1</sub> ,	2,	T <sub>2</sub>
(2)	+	a,	b,	T <sub>3</sub>
(3)	↑	T <sub>3</sub> ,	3,	T <sub>4</sub>
(4)	+	T <sub>2</sub> ,	T <sub>4</sub> ,	T <sub>5</sub>

\*\* 注: T<sub>i</sub>是临时变量。





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

的四元式可以表示为

NO.	OP	ARG1	ARG2	Result
(1)	>	X	Y	t1
(2)	J <sub>F</sub>	t1		(5)
(3)	=	X		Z
(4)	J			(7)
(5)	+	Y	1	t2
(6)	=	t2		Z
(7)				



跟语法分析的结果最相似的表示形式为树；  
逆波兰式适于栈式存储的计算机（堆栈机）；  
四元式便于优化；  
间接三元式优化时的时空效率类似于四元式。



语法分析树

去除非终结符号

AST

线性化：  
后序遍历

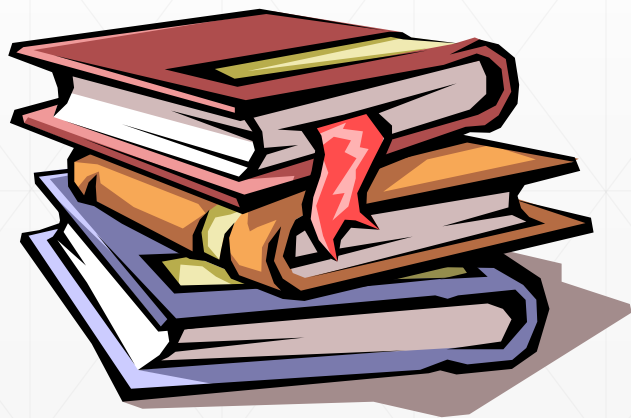
逆波兰式

“单步”  
表示

三元式

增加结果量

四元式



# 第 6 章 语义分析与中间代码生成



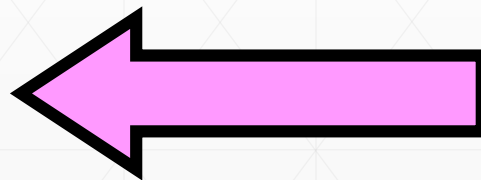
6.1 语法制导翻译

6.2 符号表

6.3 类型检查

6.4 中间语言

6.5 中间代码生成





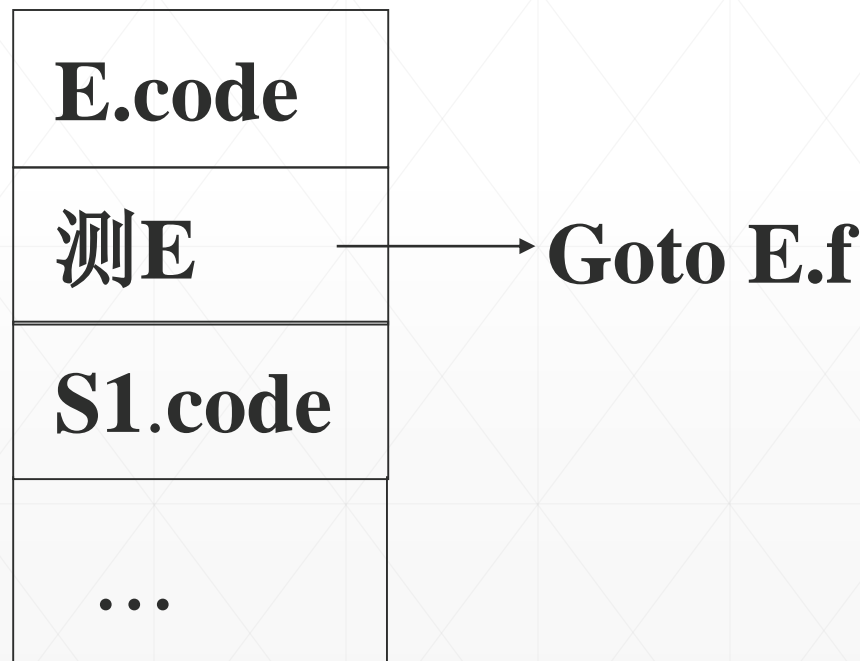
## ■ 语句翻译设计要点

1. 根据语义确定语句的目标结构；

源语句  中间及目标代码的布局

**If E then  $S_1$**

**E.f:**





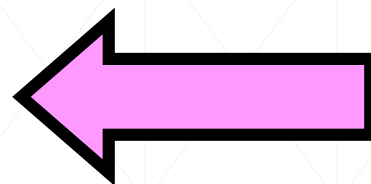
## ■ 语句翻译设计要点

1. 根据语义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，构造合适的SDT或属性翻译文法；
4. 涉及的实现技术



## 6.6 语句翻译与中间代码生成

6.6.1 说明类语句的翻译



6.6.2 赋值语句与表达式翻译

6.6.3 控制流类语句翻译

6.6.4 数组说明与数组元素引用的翻译

6.6.5 过程、函数说明和调用的翻译



## ■ 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， 常量说明， 变量说明， 类型说明，  
对象说明， 标号说明 .....

## ■ 说明类语句的处理

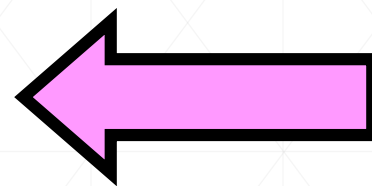
相关说明的属性信息填入符号表，提供语义检查和存储分配的依据。





## 6.6.1 说明类语句的翻译

一. 常量定义语句的翻译



二. 简单说明类语句

三. 复合类型说明语句



## 一. 常量定义语句的翻译

### 常量说明语句的文法

$\langle C\_D \rangle \rightarrow \text{CONST } \langle C\_L \rangle;$

$\langle C\_L \rangle \rightarrow \langle C\_L \rangle; \langle CD \rangle | \langle CD \rangle$

$\langle CD \rangle \rightarrow \text{id=num}$

### 常量说明语句的语义处理:

1. 等号右边的常量是第一次出现, 则将其填入常量表且回送常量表序号,
2. 将等号左边的标识符在符号表中登记新记录, 记录信息包括: 常量标志, 类型, 常量表序号



## 一. 常量定义语句的翻译

### 常量说明语句的SDT

**L属性**

$\langle C\_D \rangle \rightarrow \text{CONST} \quad \{ \langle C\_L \rangle.\text{att} = \text{cons} \}$

$\langle C\_L \rangle;$

$\langle C\_L \rangle \rightarrow \quad \{ \langle C\_L \rangle_1.\text{att} = \langle C\_L \rangle.\text{att} \}$

$\langle C\_L \rangle;$

$\{ \langle CD \rangle.\text{att} = \langle C\_L \rangle.\text{att}; \}$

$\langle CD \rangle$

$\langle C\_L \rangle \rightarrow$

$\{ \langle CD \rangle.\text{att} = \langle C\_L \rangle.\text{att}; \}$

$\langle CD \rangle$

$\langle CD \rangle \rightarrow \text{id} = \text{num}$

$\{ \text{look\_con\_table}(\text{num.lexval});$

$\text{enter}(\text{id}, \langle CD \rangle.\text{att}, \text{num.type}, \text{num.addr}); \}$



## 一. 常量定义语句的翻译

### 常量说明语句的SDT

$\langle C\_D \rangle \rightarrow \text{CONST id=num};$

$\{\text{look\_con\_table(num.lexval);}$

$\text{enter(id, cons,num.type,num.addr);}$

$\langle C\_D \rangle.\text{att}=\text{constant}; \}$

$\langle C\_D \rangle \rightarrow \langle C\_D \rangle \text{ id=num};$

$\{\text{look\_con\_table(num.lexval);}$

$\text{enter(id, } \langle C\_D_1 \rangle.\text{att,num.type,num.addr);}$

$\langle C\_D \rangle.\text{att}= \langle C\_D_1 \rangle.\text{att}\}$

S属性





## 举例说明:

形式: **CONST** 标识符=常量; ...

**CONST** pi=3.1416; true=1;

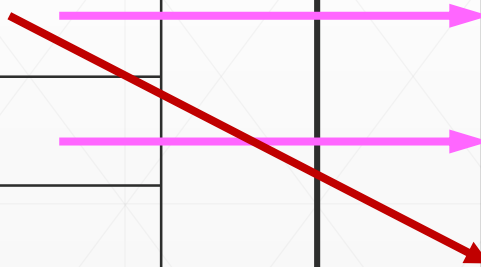
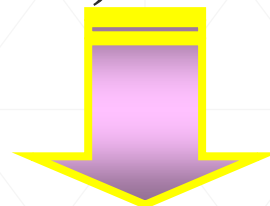
**CONST** pi=3.1415926;

**namelist**

**constlist**

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

value
3.1416
1
3.1415926



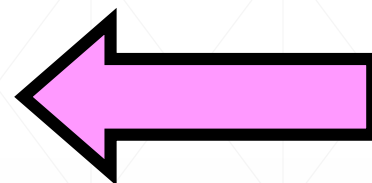


## 6.6.1 说明类语句的翻译

一. 常量定义语句的翻译

二. 简单说明类语句

三. 复合类型说明语句





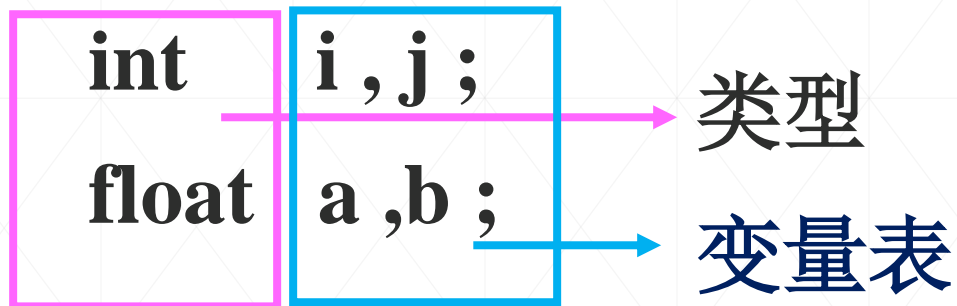
## ■ 简单说明语句的SDT

```
D→D,  id  {enter(id,v,D1.att,offset);  
offset=offset+D1.width;D.att=D1.att;D.width=D1.width;}
```

## 处理变量说明前设置offset变量为0



## 举例说明:



### namelist

name	kind	type	offset	
i	V	I	0	...
j	V	I	4	
a	V	R	8	
b	V	R	16	
...				



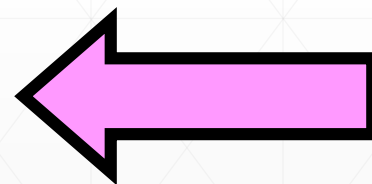


## 6.6.1 说明类语句的翻译

一. 常量定义语句的翻译

二. 简单说明类语句

三. 复合类型说明语句





### 三. 复合类型说明语句

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

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)



## namelist

name	kind	...	addr
k1	struct		
...			

例如,  
**struct k1{  
  int a;  
  float b;  
  int c[10];  
  char d;  
}**

## 结构成员分表

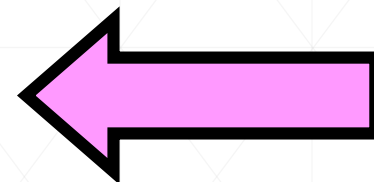
name	kind	type	LEN	OFFSET
a	V	I	4	0
b	V	R	8	4
c	array	I	40	12
d	V	C	1	52
...				



## 6.6 语句翻译与中间代码生成

6.6.1 说明类语句的翻译

6.6.2 赋值语句与表达式翻译



6.6.3 控制流类语句翻译

6.6.4 数组说明与数组元素引用的翻译

6.6.5 过程、函数说明和调用的翻译



## ■ 赋值语句形式定义

$$A \rightarrow V = E$$

## ■ 赋值语句目标结构



## ■ 赋值语句语义处理

- (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;



## ■ 赋值语句的四元式翻译SDT

$A \rightarrow i = E$

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

其中:

**GEN(OP, ARG1, ARG2, RESULT):** 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。



## ■ 算术表达式形式定义

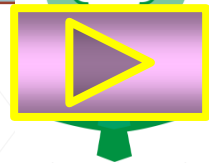
$$E \rightarrow E \text{ op } E \mid \text{op } E \mid \text{id}$$

其中：

**OP:** 为运算符；

**E, id:** 运算对象。





## ■ 算术表达式语句的四元式翻译SDT

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

$\{E. \text{ PLACE} = \text{NEWTEMP};$

$\text{GEN}(\text{OP}, E_1. \text{ PLACE}, E_2. \text{ PLACE}, E. \text{ PLACE})\}$

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

$\{E. \text{ PLACE} = \text{NEWTEMP};$

$\text{GEN}(\text{OP}, E_1. \text{ PLACE}, \_, E. \text{ PLACE})\}$

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

$\{E. \text{ PLACE} = \text{id}\}$



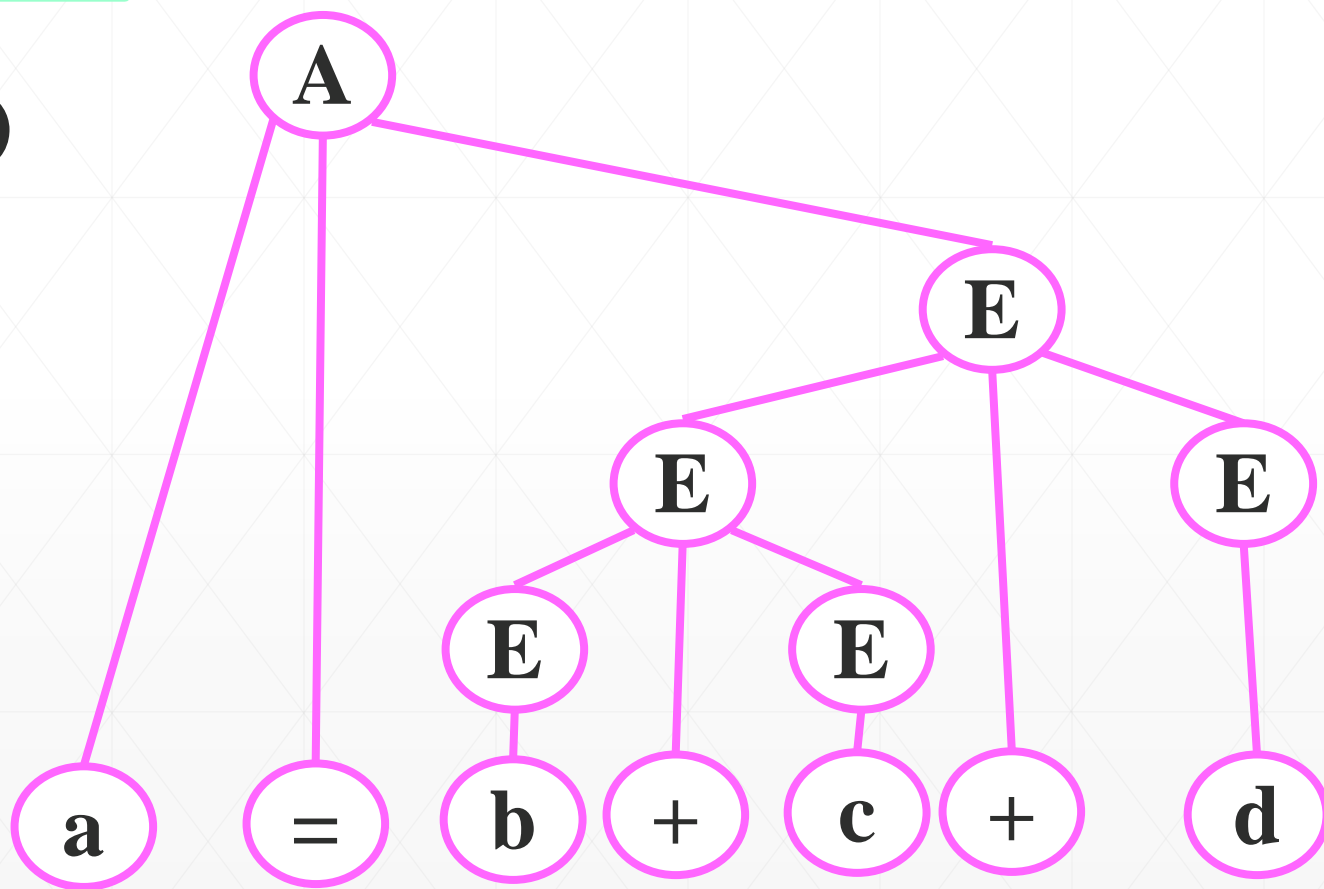
## ■ 赋值语句与表达式翻译举例:

写出C语言语句 **a=b+c+d** 翻译后的四元式代码:

(+ b c t1)

(+ t1 d t2)

(= t2 -- a)





## ■ 逻辑表达式形式定义

**$E \rightarrow E \text{ and } E \mid E \text{ or } E \mid \text{not } E \mid E \text{ rop } E \mid \text{true} \mid \text{false}$**

其中：

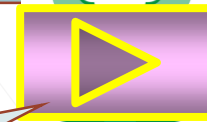
**and:** 代表并且运算

**or:** 代表或者运算

**not:** 代表非运算

**rop:** 代表关系运算比如  $\leq$ 、 $<$ 、 $=$ 、 $\geq$ 、 $>$ 、 $\neq$  等

## ■ 逻辑表达式语句的四元式翻译描述



简单翻译

- (1)  $E \rightarrow E \text{ and } E$   
 $\{ E. PLACE = NEWTEMP;$   
 $GEN(\text{and}, E_1. PLACE, E_2. PLACE, E.PLACE) \}$
- (2)  $E \rightarrow E \text{ or } E$   
 $\{ E. PLACE = NEWTEMP;$   
 $GEN(\text{or}, E_1. PLACE, E_2. PLACE, E.PLACE) \}$
- (3)  $E \rightarrow \text{not } E$   
 $\{ E. PLACE = NEWTEMP;$   
 $GEN(\text{not}, E_1. PLACE, \_, E.PLACE) \}$
- (4)  $E \rightarrow E \text{ rop } E$   
 $\{ E. PLACE = NEWTEMP;$   
 $GEN(\text{rop}, E_1. PLACE, E_2. PLACE, E.PLACE) \}$
- (5)  $E \rightarrow \text{true}$                      $\{ GEN(=, 1, -, E.PLACE) \}$
- (6)  $E \rightarrow \text{false}$                      $\{ GEN(=, 0, -, E.PLACE) \}$

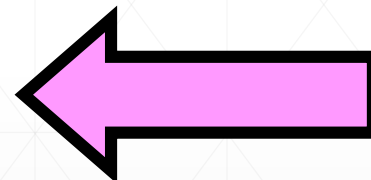


## 6.6 语句翻译与中间代码生成

6.6.1 说明类语句的翻译

6.6.2 赋值语句与表达式翻译

6.6.3 控制流类语句翻译



6.6.4 数组说明与数组元素引用的翻译

6.6.5 过程、函数说明和调用的翻译



■ **控制流语句**: 改变程序执行顺序, 引起程序执行发生跳转 (向前或向后)。

- 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**。



## ■ 跳转目标标记

语句标号 { 显式：位于源语句之前；（如，**L1:** S;）  
隐式：含于源语句之中但未标识

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

跳出if 继续

e<sub>r</sub>=F  
跳转目标

循环变量重赋值

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

循环体  
终止判别

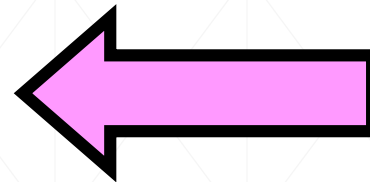
循环体  
开始

跳出循  
环体继续



## 6.6.3 控制流类语句翻译

一. 语句标号与拉链返填技术



二. 条件语句的翻译

三. 循环语句的翻译

四. 多分支语句的翻译

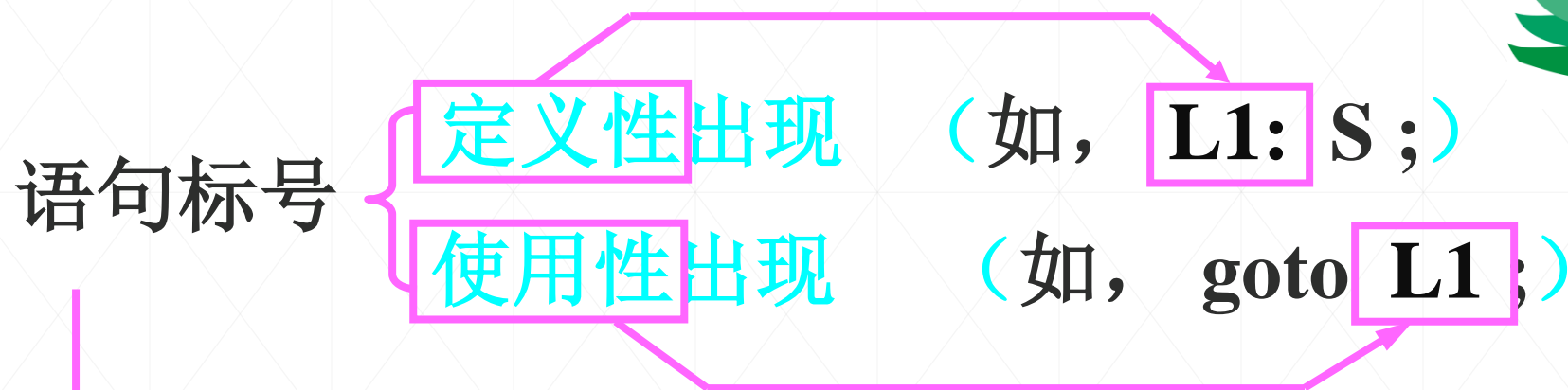




## ■ 语句标号处理

控制流类语句处理面对의公共问题和实现技术

- 语句标号处理 {
  - 先定义后引用
  - 先引用后定义



语句标号表 (LT)

标号名	定义否(flag)	addr
-----	-----------	------

The diagram shows a table with three columns: '标号名' (Label Name), '定义否(flag)' (Defined? flag), and 'addr' (Address). A pink arrow points from the 'addr' column down to the explanatory text below. A black arrow points from the '定义否(flag)' column down to the legend.

1表示定义性出现

0表示使用性出现

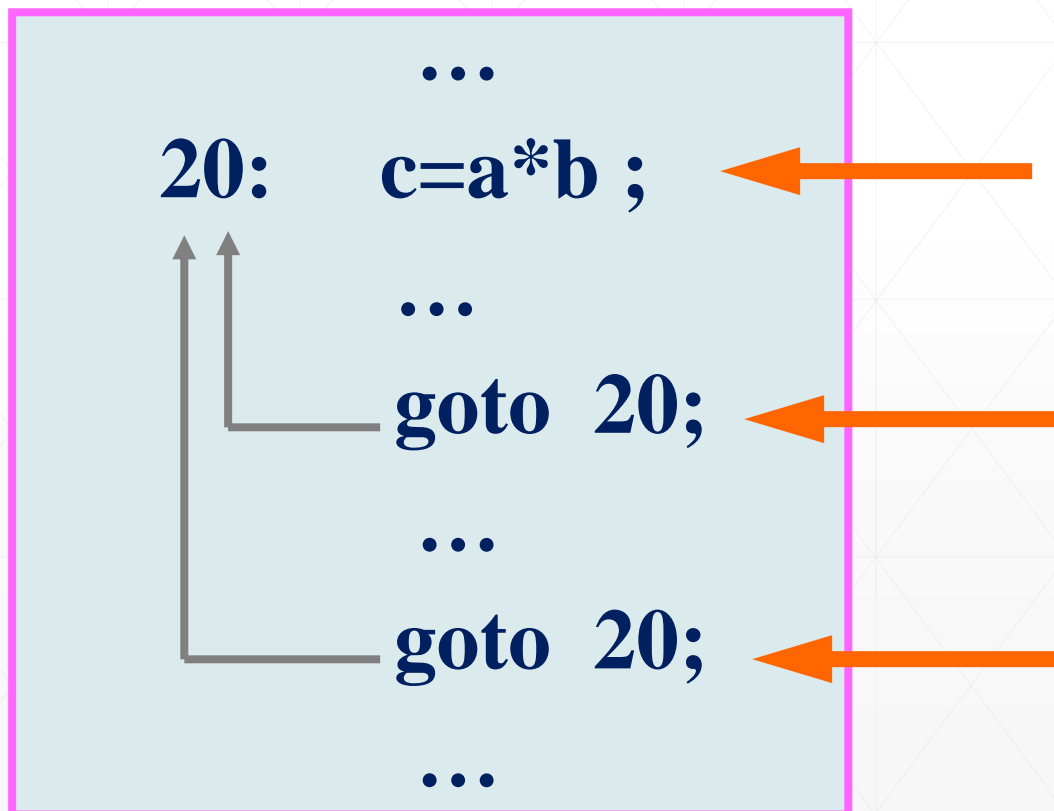
标号所标识的源语句的中间代码序列的  
第一条代码的存放地址或编号



## ■ 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，



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

Code区

q-1	...
q	*, a, b, c
	...
k	j, , , q
	...
n	j, , , q
	...

LT

20	1	q
----	---	---

20: c=a\*b;

goto 20;

goto 20;

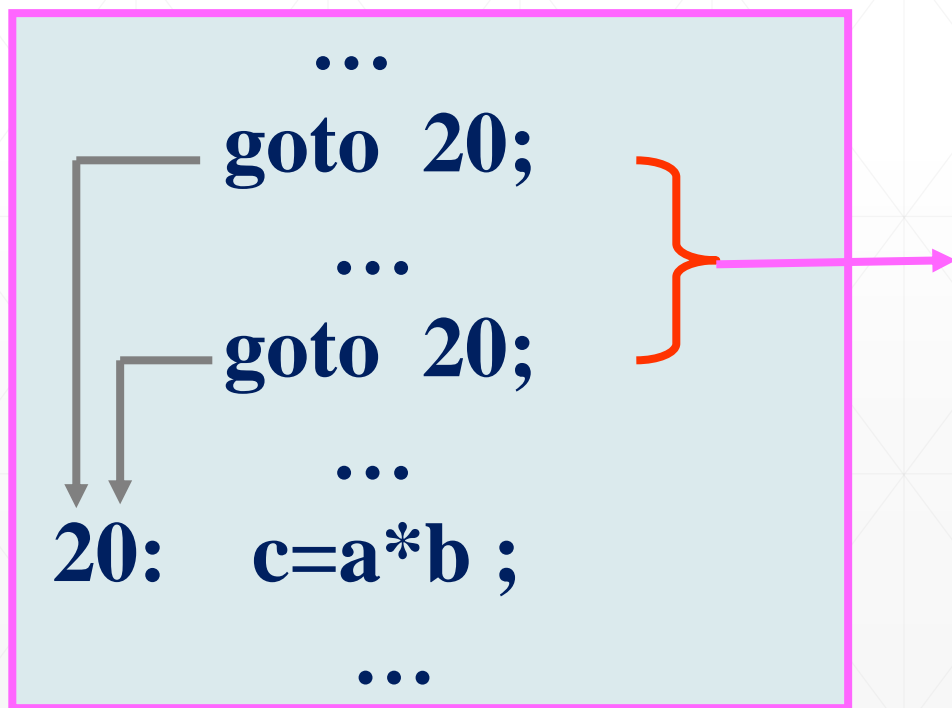


## ■ 拉链-返填技术

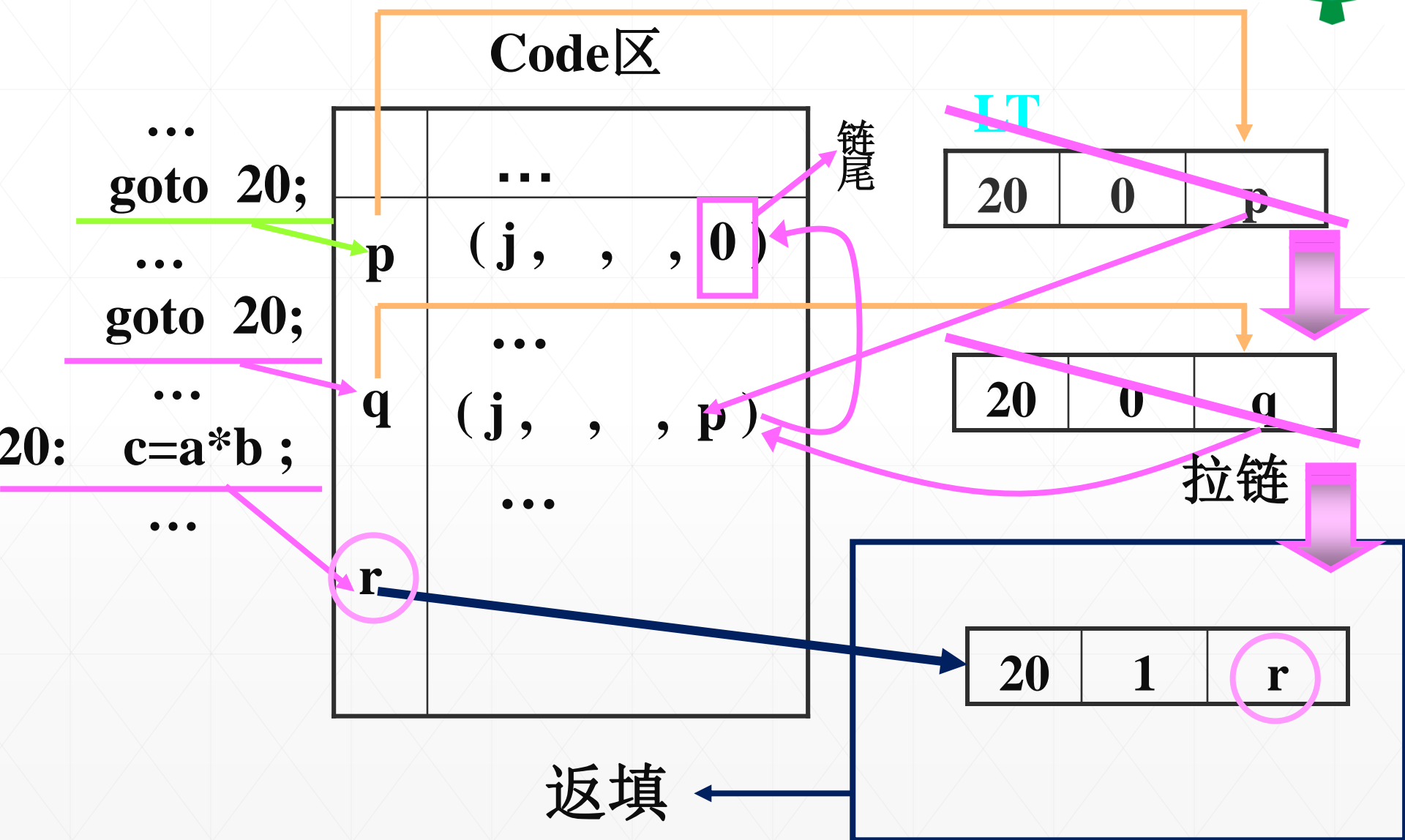
处理标号先引用后定义的情况

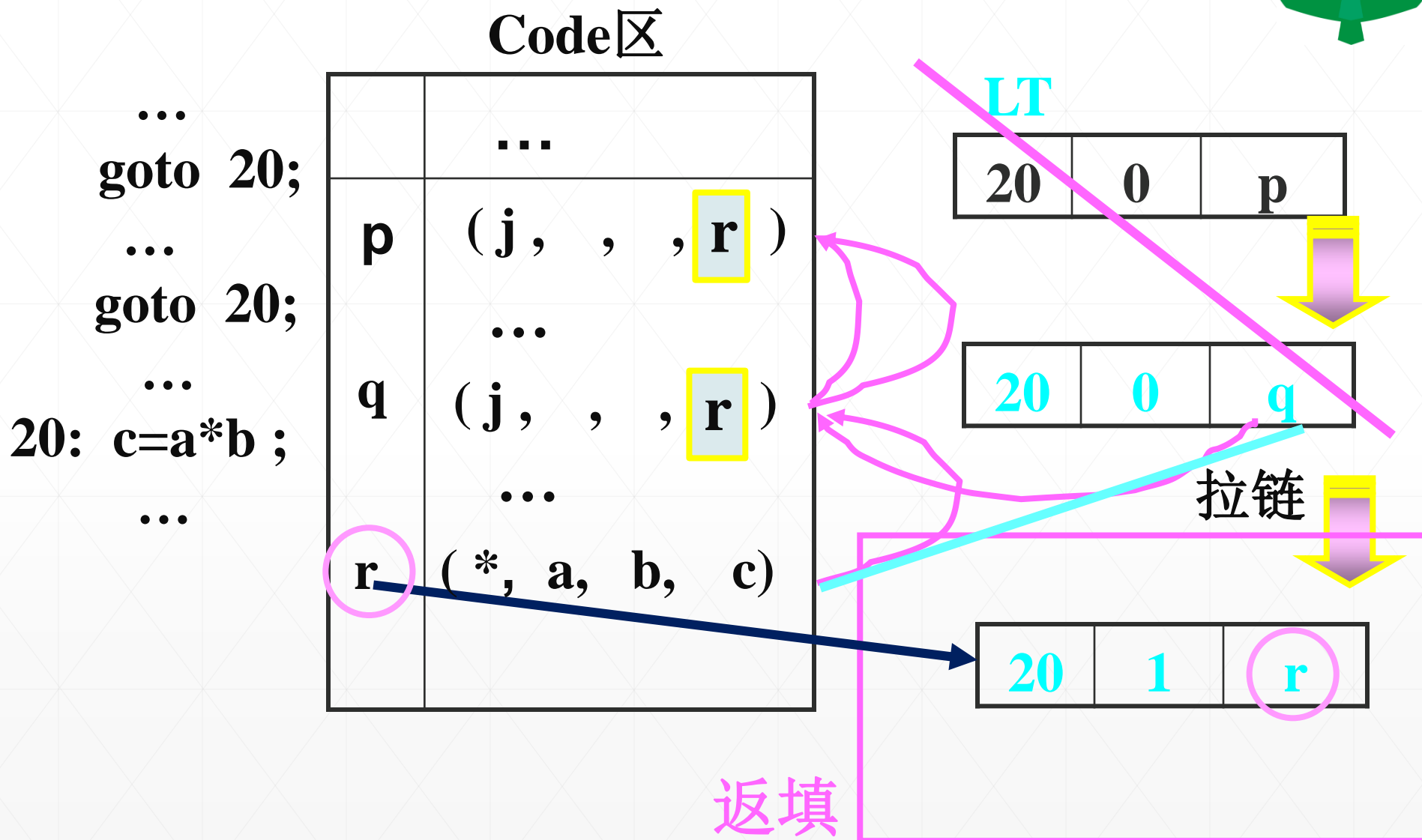
适用于一遍扫描的编译器

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。



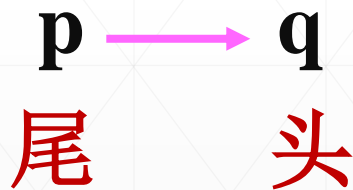




## 注意:

1. 链尾标志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:



3. 返填次序:



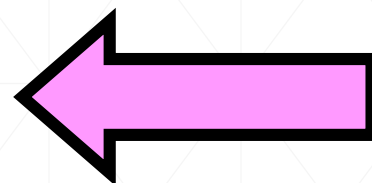




## 6.6.3 控制流类语句翻译

一. 语句标号与拉链返填技术

二. 条件语句的翻译



三. 循环语句的翻译

四. 多分支语句的翻译



## 二. 条件语句的翻译

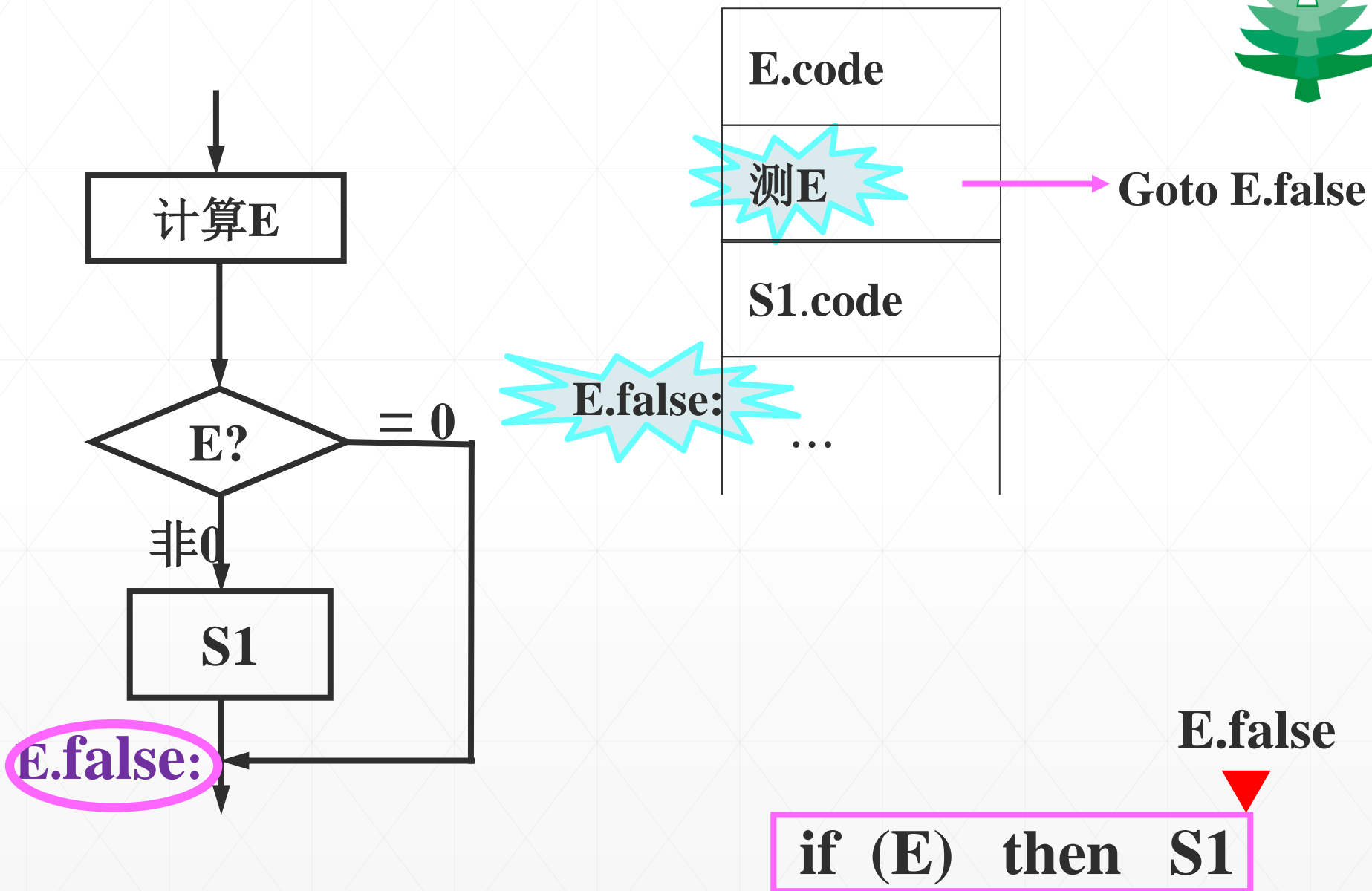
条件语句文法:

$$\begin{aligned} S \rightarrow & \text{if } (E) \text{ then } S1 \text{ ①} \mid \\ & \text{if } (E) \text{ } S1 \text{ else } S2 \text{ ②} \mid \\ & \text{while } (E) \text{ } S1 \text{ ③} \end{aligned}$$

其中:

**E:** 条件表达式;

**S1, S2:** 语句;





## ■ 条件语句①的SDT

$S \rightarrow I \text{ then } S$

回填

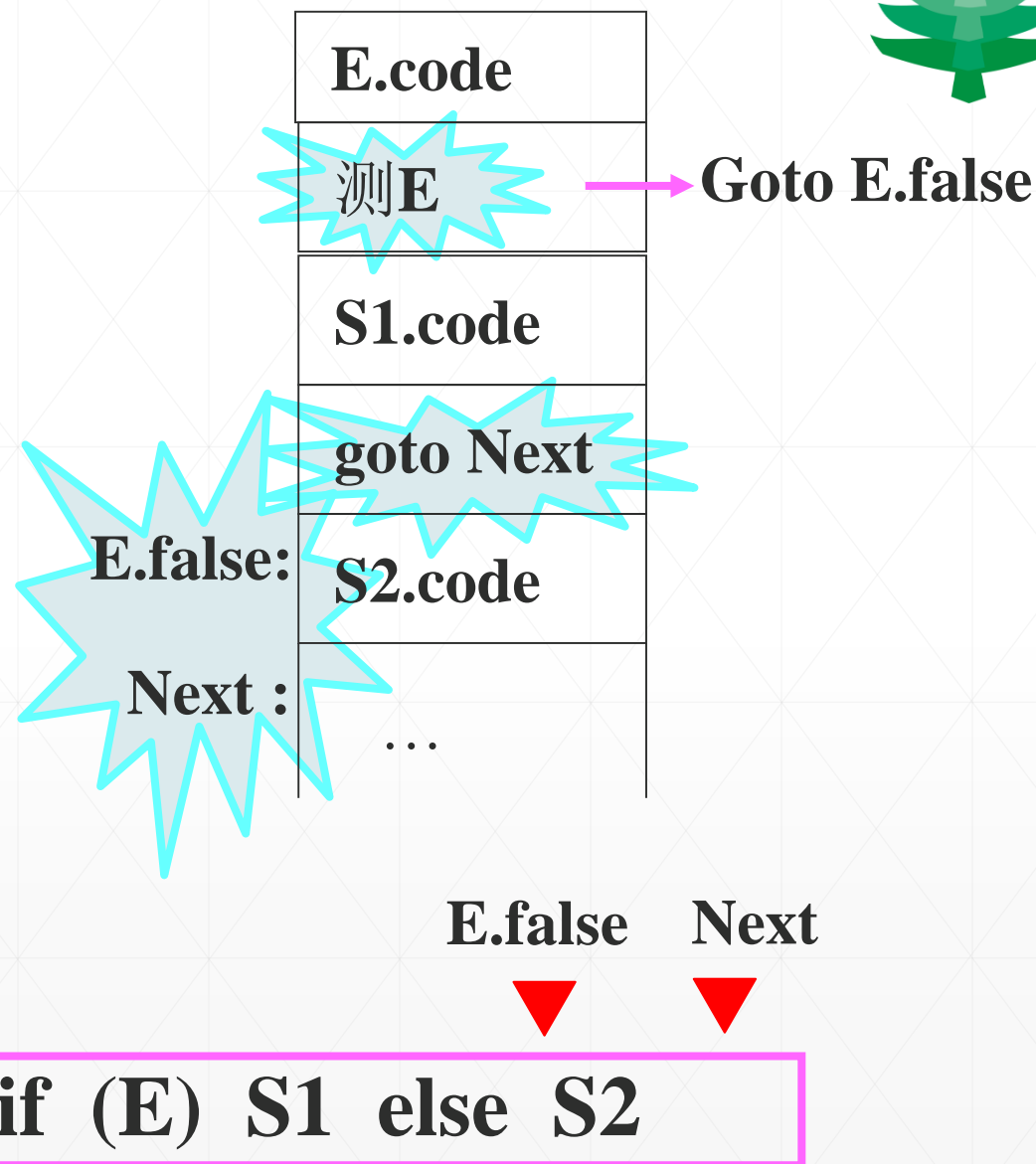
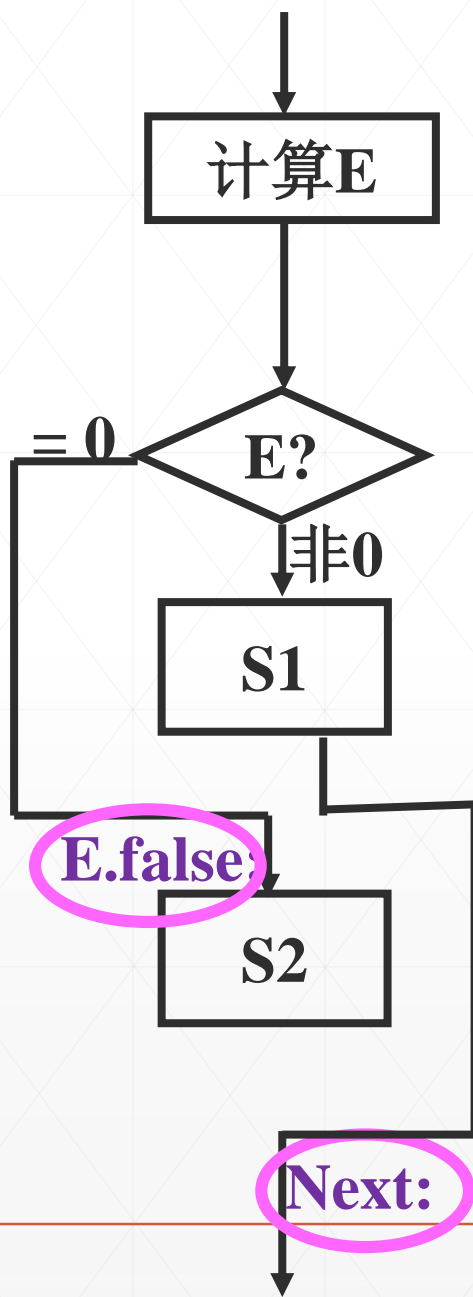
$\{\text{Label}(\text{E.false})\}$

$I \rightarrow \text{if } (E)$

$\{\text{GEN}(jF, E.\text{place}, \_, E.\text{false})\}$

E.false

if (E) then S1





## ■ 条件语句②的属性翻译文法

$S \rightarrow T \text{ else } S$

$\{\text{Label(Next)}\}$

回填

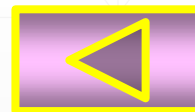
$T \rightarrow I \text{ then } S$

$\{\text{GEN}(J, \_, \_, \text{Next}); \text{Label}(\text{E.false})\}$

回填

$I \rightarrow \text{if } (E)$

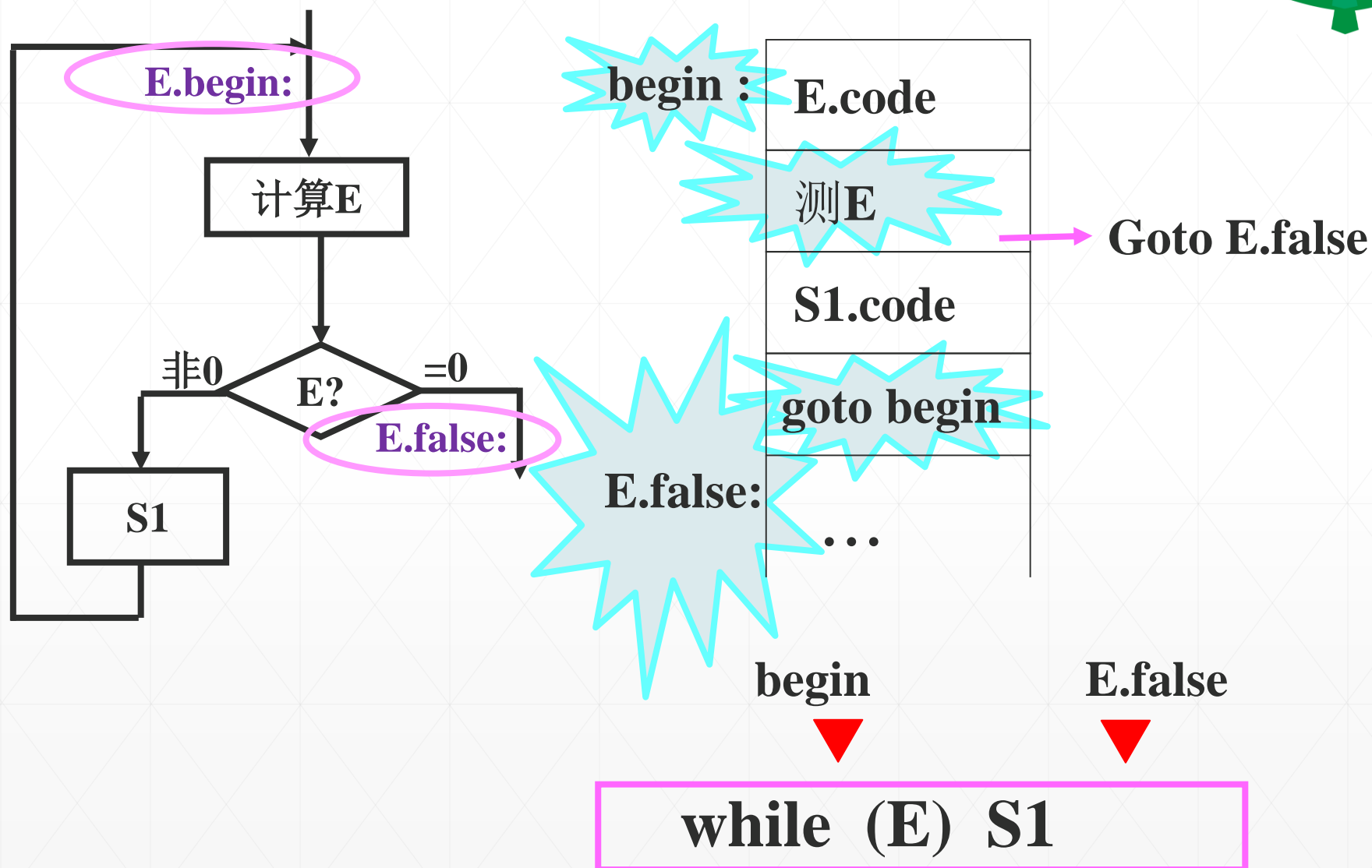
$\{\text{GEN}(jF, \text{E.place}, \_, \text{E.false})\}$



$\text{E.false}$        $\text{Next}$



$\text{if } (E) \text{ S1 else S2}$





## ■ 条件语句③的属性翻译文法

$S \rightarrow CS$

$\{\text{GEN}(j, \_, \_, \text{begin}); \text{Label}(E.\text{false})\}$

$C \rightarrow W(E)$

$\{\text{GEN}(jF, E.\text{palce}, \_, E.\text{false})\}$

$W \rightarrow \text{while}$

$\{\text{Label}(\text{begin})\}$

回填

begin

E.false

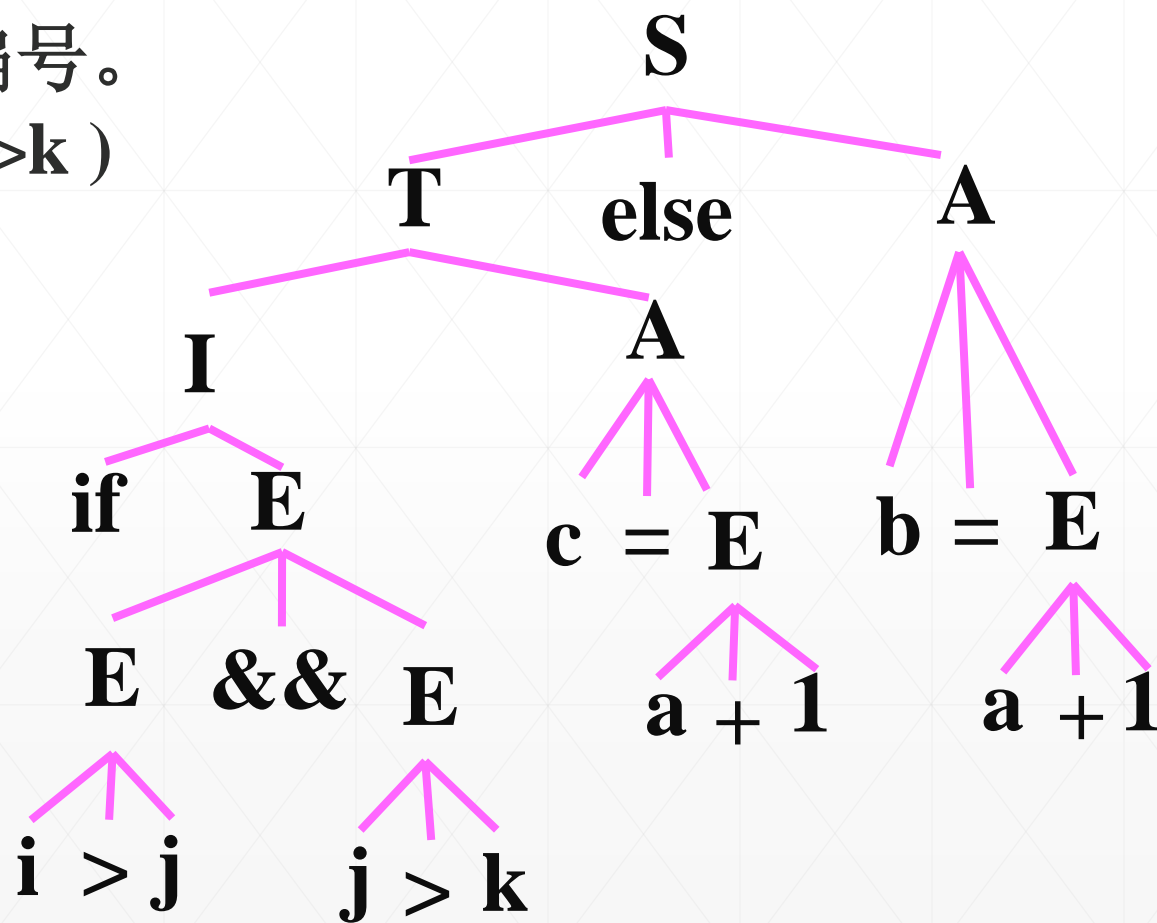
while (E) S1



**例：**模拟一遍编译的技术将下面的C语言代码翻译为四元式形式的中间代码,并给出最后的标号表。添加的标号根据标号出现的顺序依次命名为BH1, BH2, BH3, ...,标号地址使用四元式编号。

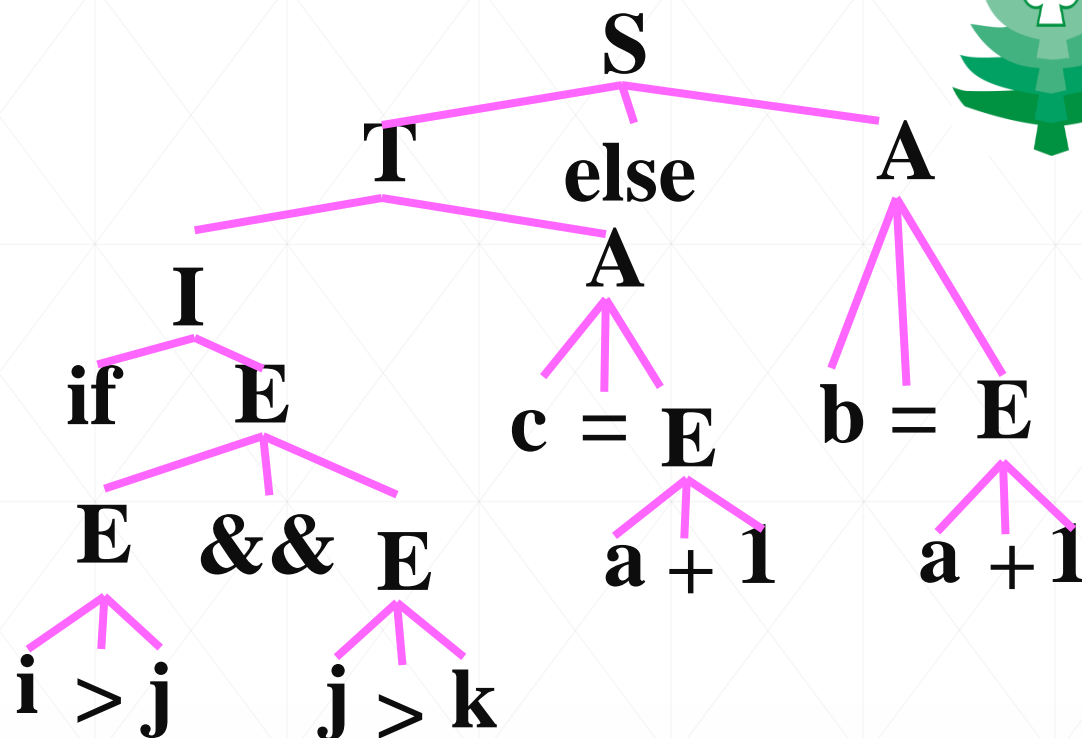
```
if (i>j && j>k )  
    c=a+1;  
else  
    b=a+1;
```

该代码段对应的  
语法分析树



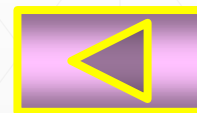


1:(> i j t1)  
 2:(> j k t2)  
 3:(&& t1 t2 t3)  
 4:(J<sub>F</sub> t3 \_ 8 )  
 5:(+ a 1 t4)  
 6:(= t4 \_ c)  
 7:(J \_ \_ 10 )  
 8:(+ a 1 t4)  
 9:(= t4 \_ b)  
 10:.....



标号表

N	D	A
BH1	1	8
BH2	1	10



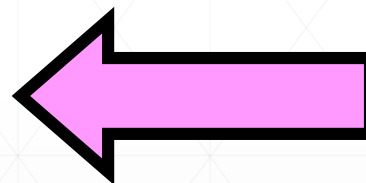


## 6.6.3 控制流类语句翻译

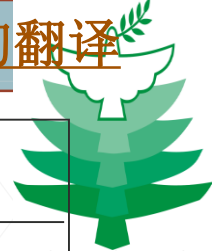
一. 语句标号与拉链返填技术

二. 条件语句的翻译

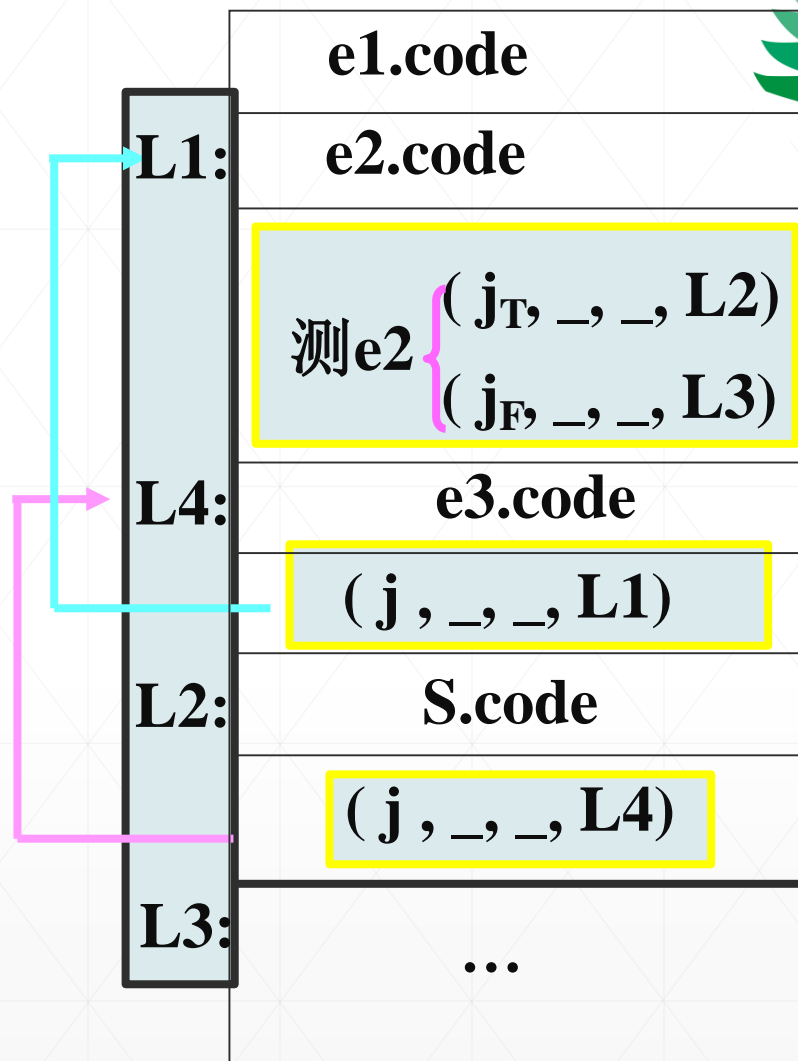
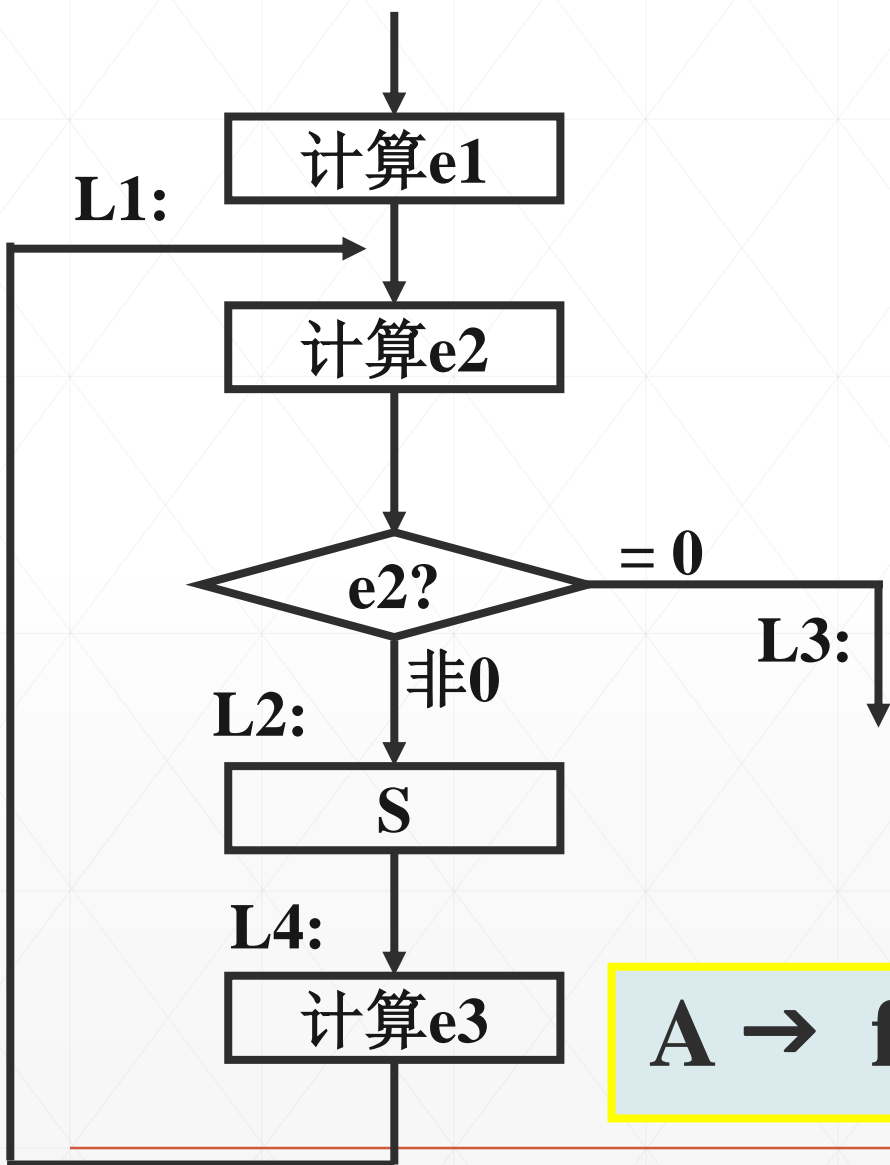
三. 循环语句的翻译



四. 多分支语句的翻译



### 三. 循环语句的翻译



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



# 循环语句的翻译的语义子程序

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

$\{ \text{GEN}(j, \_, \_, L4); \text{label}(L3) \}$

$A \rightarrow \varepsilon$

$\{ \text{Label}(L1) \}$

$B \rightarrow \varepsilon$

$\{ \text{GEN}(JT, \_, \_, L2); \text{GEN}(JF, \_, \_, L3);$   
 $\text{label}(L4) \}$

$C \rightarrow \varepsilon$

$\{ \text{GEN}(j, \_, \_, L1); \text{label}(L2) \}$

回填

回填

L1:

L4:

L2:

L3:

e1.code

e2.code

测e2  $\left\{ \begin{array}{l} (j_T, \_, \_, L2) \\ (j_F, \_, \_, L3) \end{array} \right.$

e3.code

$(j, \_, \_, L1)$

S.code

$(j, \_, \_, L4)$

...

for ( e1 ; e2 ; e3 ) S

● ● ●

 $S_2;$ 

• • •

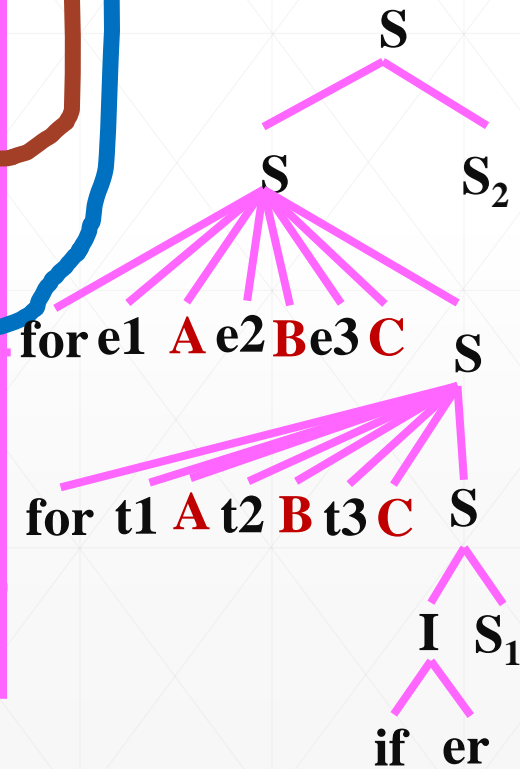
目标代码结构:

```
graph TD; S1[S] --- S2[S]; S1 --- S3[;]; S1 --- S4[S2]; S2 --- S5["for ( e1 ; Ae2; Be3C )"]; S2 --- S6[S]; S6 --- S7["for( t1 ; At2; B t3 C )"]; S6 --- S8[S]; S8 --- S9["if"]; S8 --- S10[S1]; S10 --- S11[er];
```

e1.code
L1: e2.code
测e2值 (j <sub>T</sub> , , , L2) (j <sub>F</sub> , , , L3)
L4: e3.code
(j, , , L1)
L2: t1.code
L5: t2.code
测t2值 (j <sub>T</sub> , , , L6) (j <sub>F</sub> , , , L7)

L8: t3.code
(j, , , L5)
L6: e <sub>r</sub> .code
测e <sub>r</sub> 值 (j <sub>F</sub> , , , L9) S1.code
L9: (j, , , L8)
L7: (j, , , L4)
L3: S2.code

for ( e1; e2; e3; )  
 for ( t1; t2; t3; )  
 if ( e<sub>r</sub> ) S<sub>1</sub>;  
 S<sub>2</sub>;





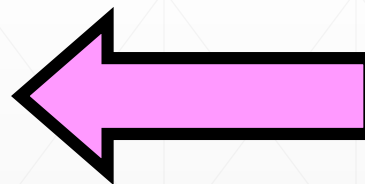
## 6.6.3 控制流类语句翻译

一. 语句标号与拉链返填技术

二. 条件语句的翻译

三. 循环语句的翻译

四. 多分支语句的翻译







## 四. 多分支语句的翻译

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

**$\{ \text{ case } c_1 : S_1 \text{ break ;}$**

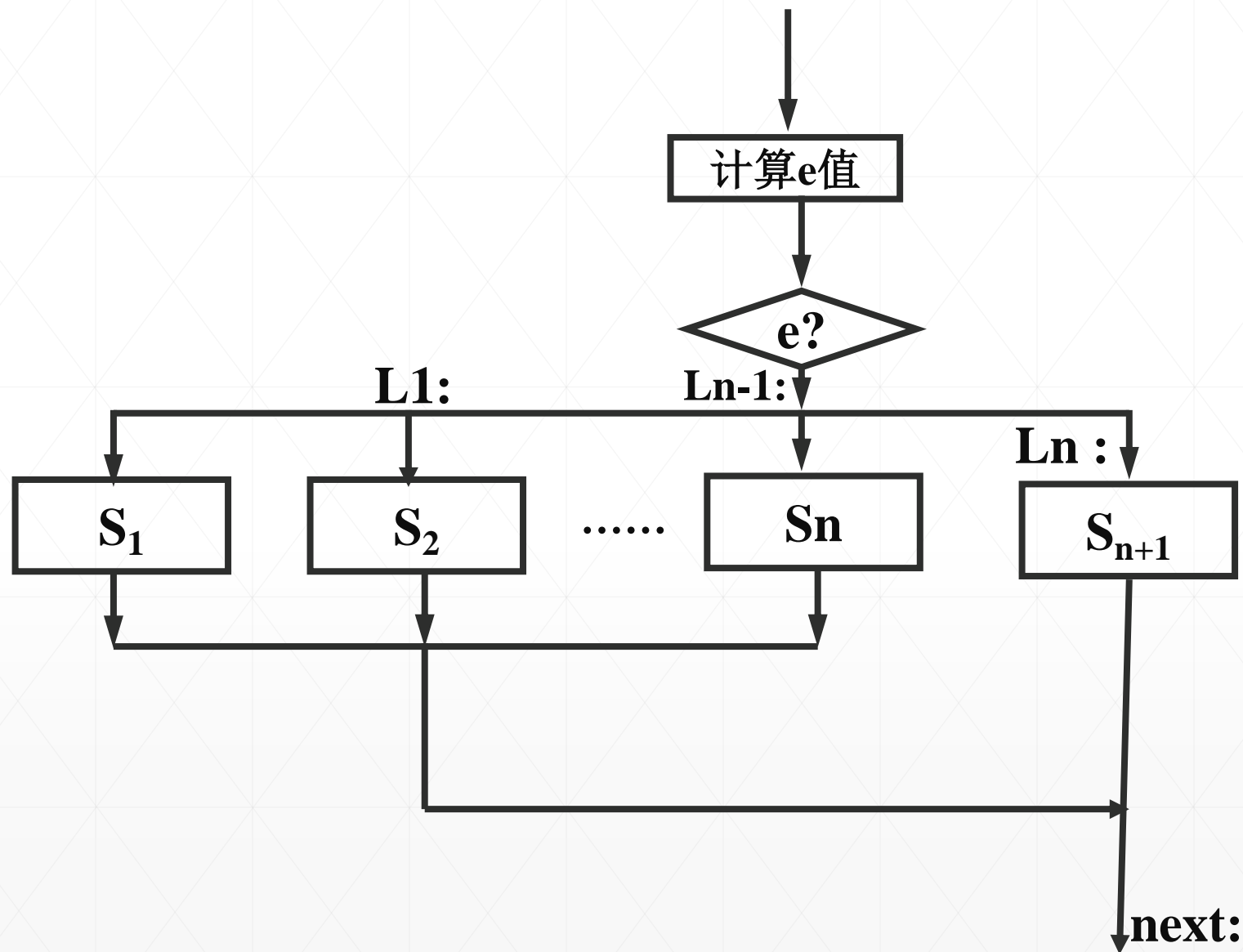
**$\text{ case } c_2 : S_2 \text{ break ;}$**

**.....**

**$\text{ case } c_n : S_n \text{ break ;}$**

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

**$\}$**





## 四. 多分支语句的翻译

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

**$\{ \text{ case } c_1 : S_1 \text{ break ;}$**

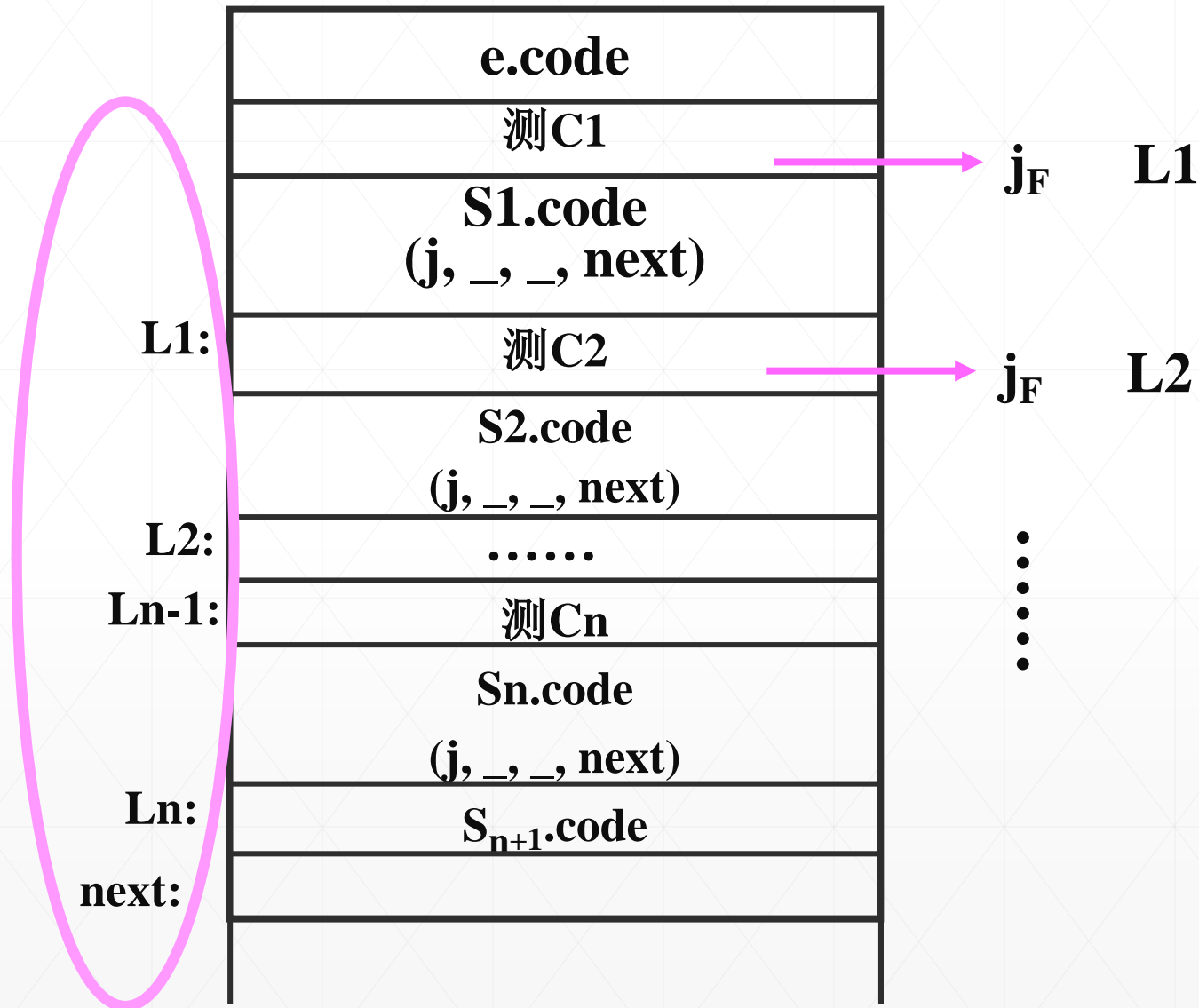
**▼  $\text{case } c_2 : S_2 \text{ break ;}$**

**.....**

**▼  $\text{case } c_n : S_n \text{ break ;}$**

**▼  $\text{default : } S_{n+1}$**

**}▼**





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

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

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

**.....**

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

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

**}**



	<b>e.code</b> <b>(j, _, _, test)</b>	
<b>L1:</b>	<b>S1.code</b> <b>(j, _, _, next)</b>	
<b>L2:</b>	<b>S2.code</b> <b>(j, _, _, next)</b>	
	<b>.....</b>	
<b>Ln:</b>	<b>Sn.code</b> <b>(j, _, _, next)</b>	
<b>Ln+1:</b>	<b>Sn+1.code</b> <b>(j, _, _, next)</b>	
<b>test:</b>	<b>If e=c1 goto L1</b> <b>If e=c2 goto L2</b> <b>.....</b> <b>If e=cn goto Ln</b> <b>goto Ln+1</b>	<b>测试表</b>
<b>next:</b>		



## 6.6 语句翻译与中间代码生成

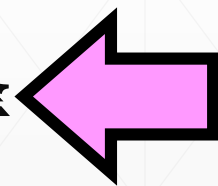
6.6.1 说明类语句的翻译

6.6.2 赋值语句与表达式翻译

6.6.3 控制流类语句翻译

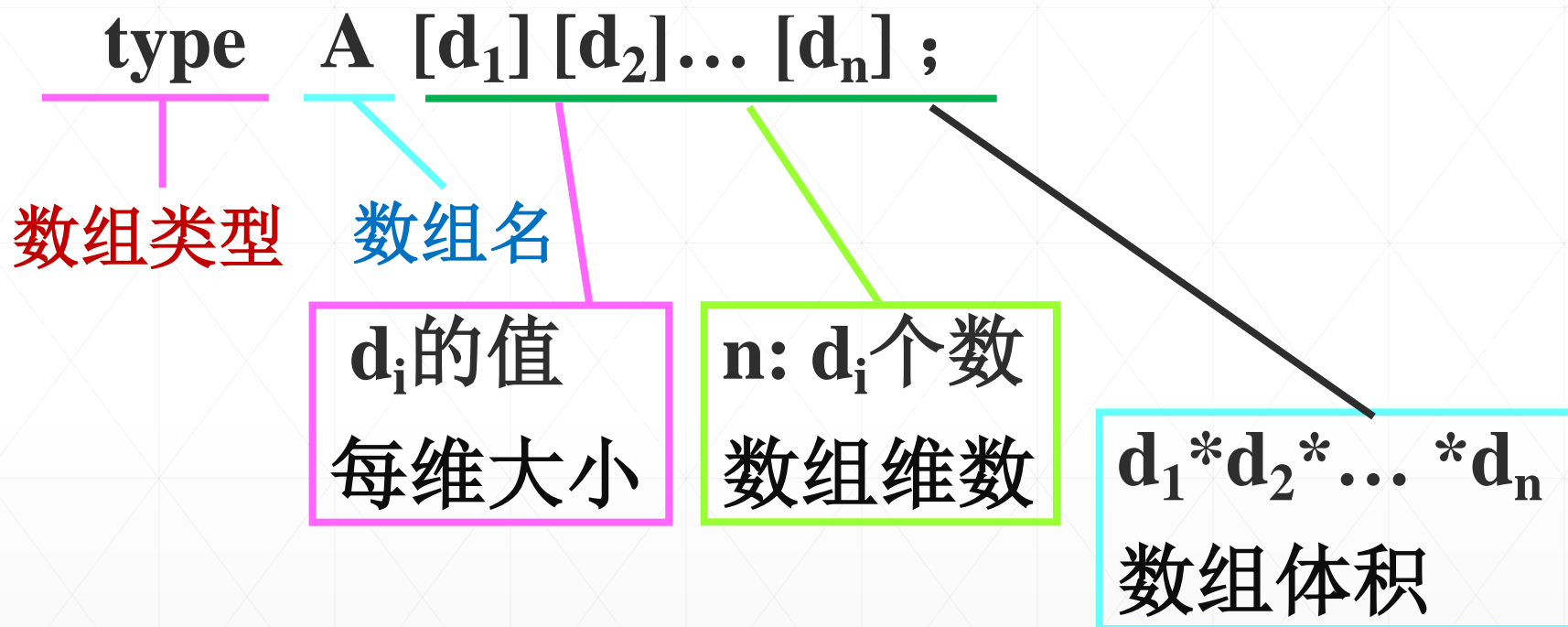
6.6.4 数组说明与数组元素引用的翻译

6.6.5 过程、函数说明和调用的翻译





## 一. 数组说明



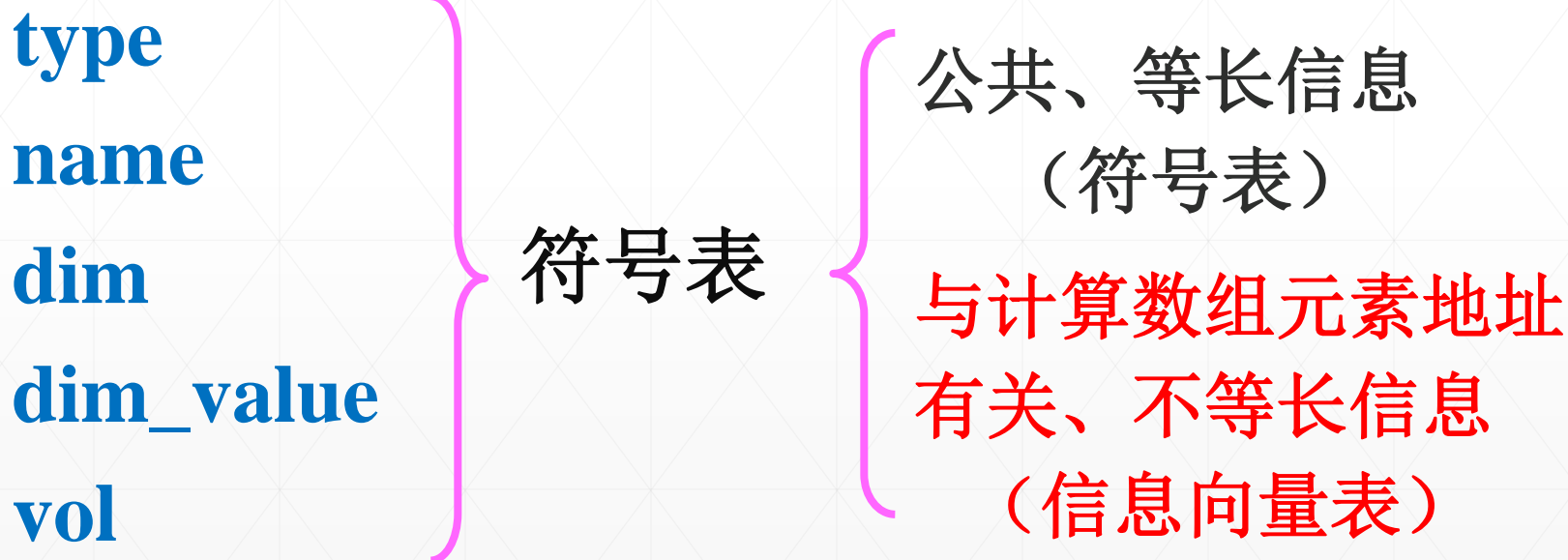
**type** **A** [ **$d_1 .. d_1'$** ] [ **$d_2 .. d_2'$** ]... [ **$d_n .. d_n'$** ] ;





## 数组说明的语义处理

填表，登记数组的属性信息。





例: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	
$C_a$	
$a_0$	b
1	
4	
4*1	
$C_b$	
$b_0$	

信息向量表

维数
第一维大小
第二维大小
...
第n维大小
体积 $=d_1*d_2*...*d_n$
计算数元地址不变部分C
数组第一个元素地址a
...

若为上下界需计算:

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

计算时间:

尽量编译时算

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。**

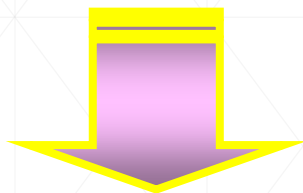
### ➤ 数组元素引用的语义处理

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。



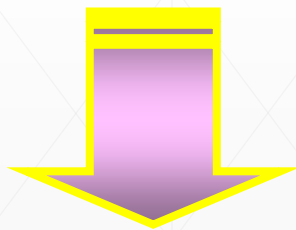
数组元素地址计算编译时能否完成？



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

$a_0$

$a[0][0]$
$a[0][1]$
$a[1][0]$
$a[1][1]$

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

按列

$a_0$

$a[0][0]$
$a[1][0]$
$a[0][1]$
$a[1][1]$

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



## ■ 数组元素地址计算

据存储方式，通过下述规则计算数组元素位置

$$(\text{顺序号}) * \text{width} + a_0$$

■ 对一维数组 `int a[n];` // 默认下标下界=0

则  $a[i]_{\text{addr}} = a_0 + (i) * \text{width}$

( 若数组下标下限=1, 则  $a[i] = a_0 + (i-1) * \text{width}$  )



■ 对二维数组 `int a[n][m];`

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

若数组下标下限=1, 则

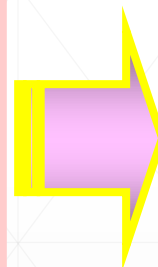
$a[i][j]_{\text{add}} = a_0 + ((i-1) * m + (j-1)) * \text{width}$



■ 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

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



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

若数组下标下限=1

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

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

+ ...

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

变换

V

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

$$- (d_2 d_3 \dots d_n + d_3 d_4 \dots d_n + \dots + d_n + 1) * \text{width}$$

C

不变

返回



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

$$V$$

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

$$C$$

$$= CUN + V$$

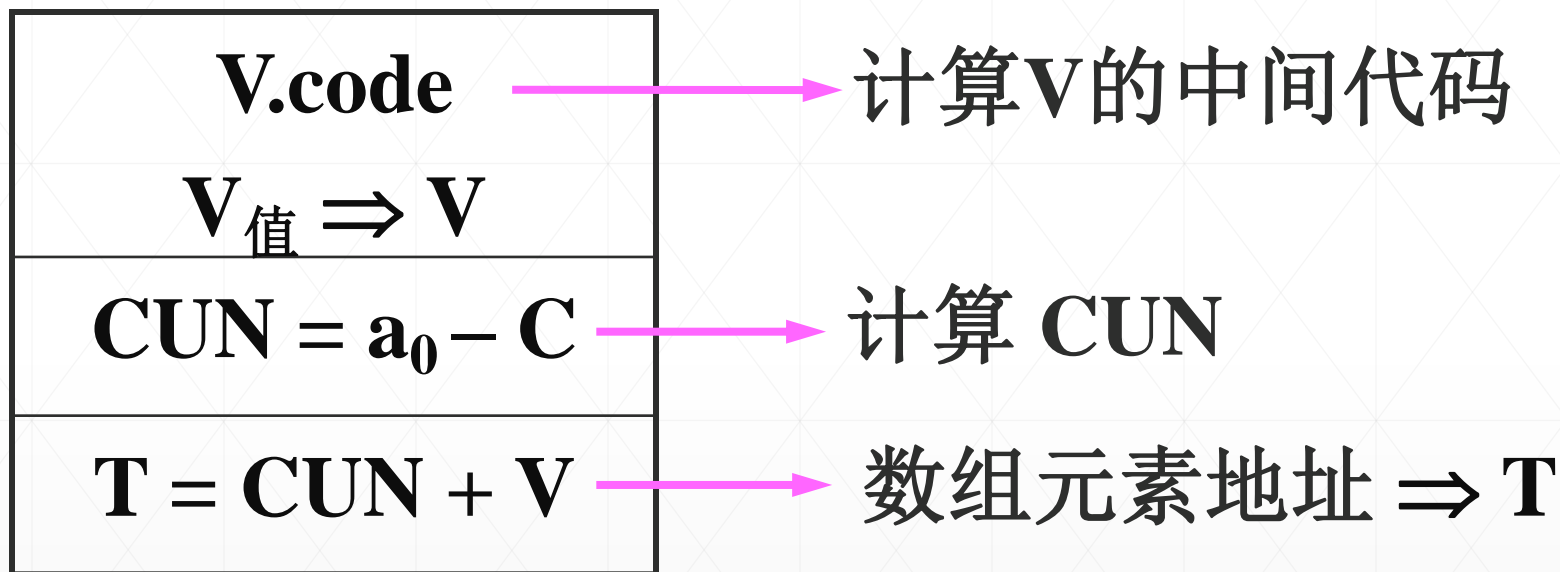
$$= a_0 - \underbrace{C}_{\text{constpart}} + \underbrace{V}_{\text{varpart}}$$

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 设有说明

...  
int a[10][20]

...  
s = a[x][y];

...  
a[x][y] = t;

namelist

name	kind	type	addr
a	array	<b>I</b>	
...			

...  
a[x][y]<sub>addr</sub>•code

①(\*, x ,20,t1)

②(+, t1,y,t2)

③(\*,4,t2 ,t3)

④(-, a<sub>0</sub>, 84,t4)

⑤(=[],t4[t3],-,t5)

⑥(=,t5,-,s)

...  
[ ]=,t,-, t9[t8])

V.code

V<sub>值</sub> ⇒ V

CUN = a<sub>0</sub> - C

T = CUN + V

2
10
20
10*20=200
C=(20+1)*4=84
a <sub>0</sub>
...

内情向量表



## ■ 动态数组？

`type A [d1] [d2]... [dn] ;` (  $d_i$  是变量 )  
( 每维大小、体积等信息程序运行时确定 )

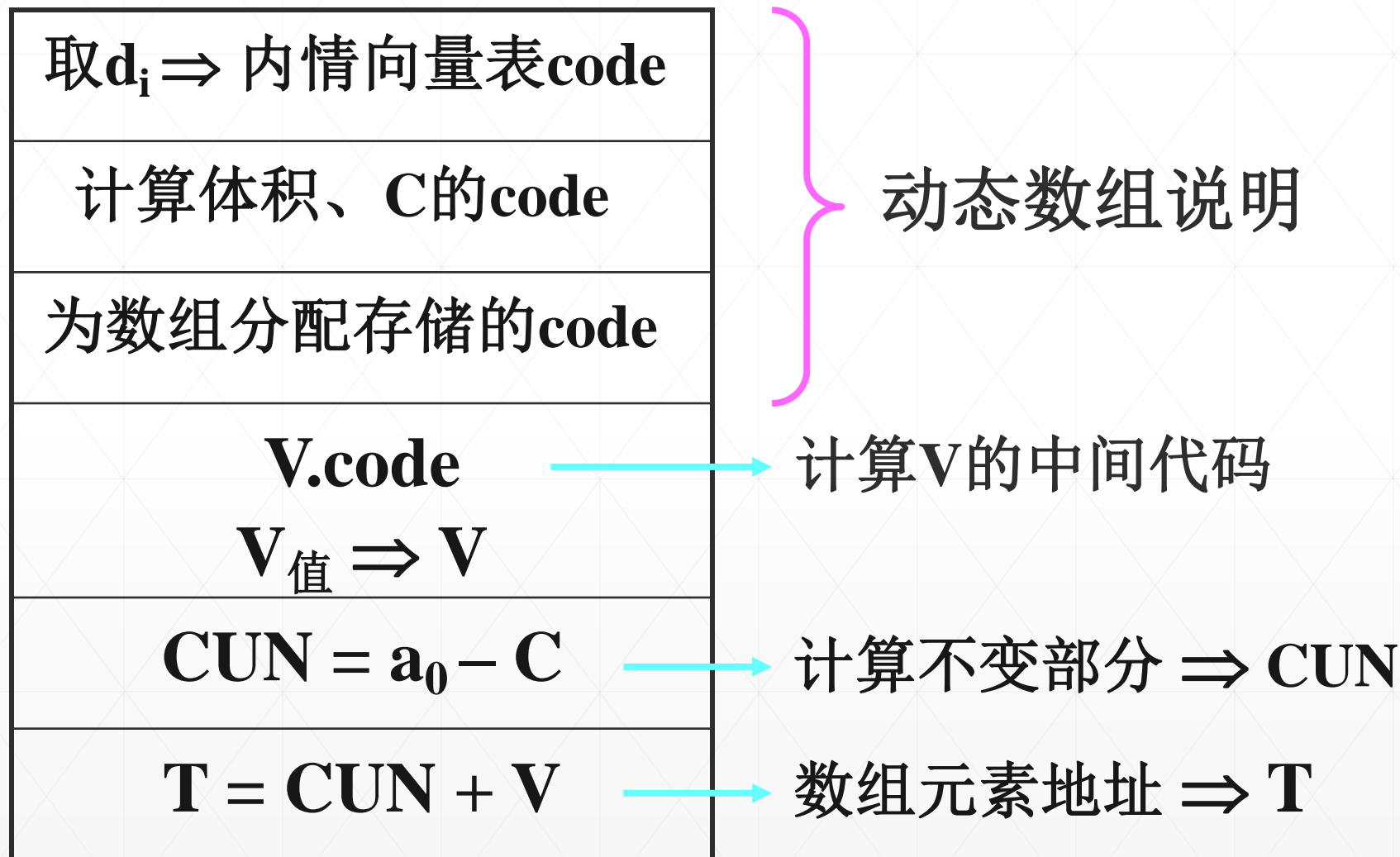
## ■ 动态数组处理

内情向量表在程序运行时建立，编译时仅分配内情向量表所占存储空间(据维数)。在运行时， $d_i$  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





## 6.6 语句翻译与中间代码生成

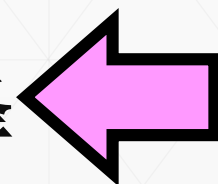
6.6.1 说明类语句的翻译

6.6.2 赋值语句与表达式翻译

6.6.3 控制流类语句翻译

6.6.4 数组说明与数组元素引用的翻译

6.6.5 过程、函数说明和调用的翻译





## 函数翻译 { 处理说明（定义） 处理调用

- 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由关键字(**PROCEDURE**, **FUNCTION**)引导, 有些语言则直接定义。





## 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.



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);



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；

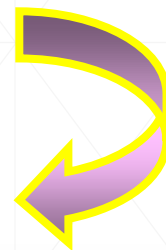


## 例6.5 设有C函数

```
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**



## 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 给被调过程或函数分配活动记录所需的存储空间；
- ③ 传送实参并执行代码；
- 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。



## ■ 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）;





# 函数调用举例（点击观看视频）

godbolt.org

COMPILER EXPLORER

Use [conan](#) or [vcpkg](#) to manage your C & C++ library dependencies

Sponsors Share Other Policies

C source #1

```
1 // Type your code here, or load an example.
2 int square(int num) {
3     if(num != 0)
4         return num * num;
5     else
6         return 0;
7 }
8
9 int main(){
10     int sum = 0;
11
12     for(int i = 0; i < 100; i++){
13         sum += square(i);
14     }
15     return sum;
16 }
```

MIPS64 gcc 5.4 (Editor #1, Compiler #1) C

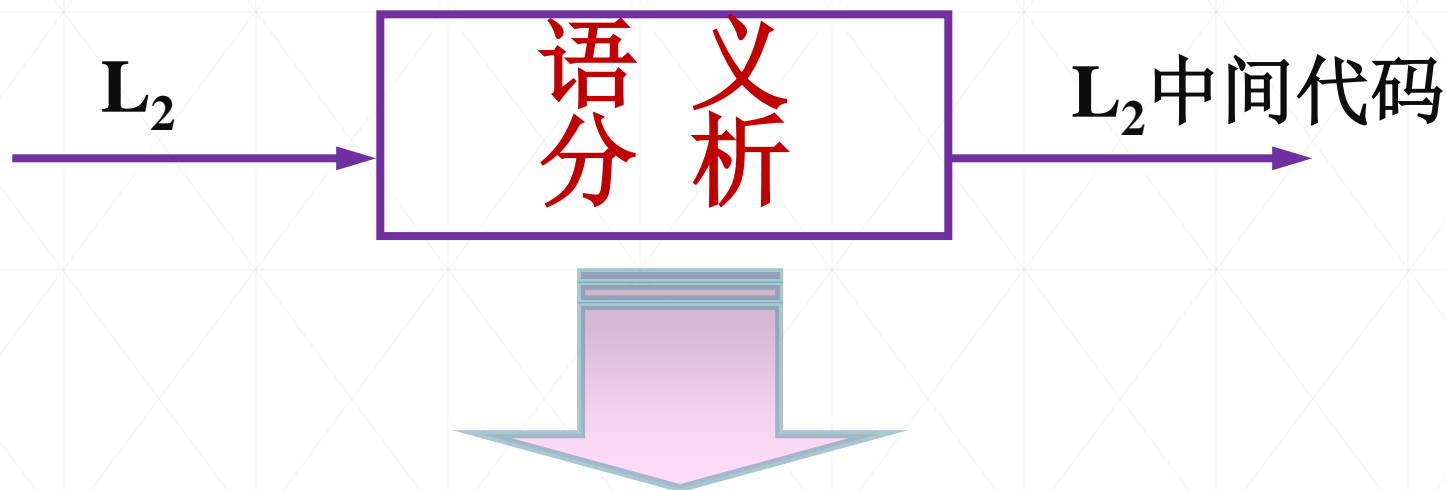
MIPS64 gcc 5.4

Compiler options...

Output... Filter... Libraries + Add new... Add tool...

```
50 move $3,$2
51 lw $2,0($fp)
52 addu $2,$2,$3
53 sw $2,0($fp)
54 lw $2,4($fp)
55 addiu $2,$2,1
56 sw $2,4($fp)
57 .L5:
58 lw $2,4($fp)
59 slt $2,$2,100
60 bne $2,$0,.L6
61 nop
62
63 lw $2,0($fp)
64 move $sp,$fp
65 ld $31,40($sp)
66 ld $fp,32($sp)
67 ld $28,24($sp)
68 daddiu $sp,$sp,48
69 j $31
70 nop
```

# 语义分析与中间代码生成总结



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性；  
语义映射的准确性；

# 语义分析与中间代码生成

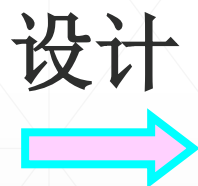


说明性语句

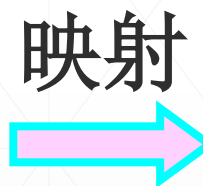


登录编译信息、为可执行语句提供语义检查和语句翻译的信息依据。

可执行语句

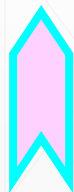


目标代码结构



中间代码

动态语义



语义动作



# 第六章 结束

