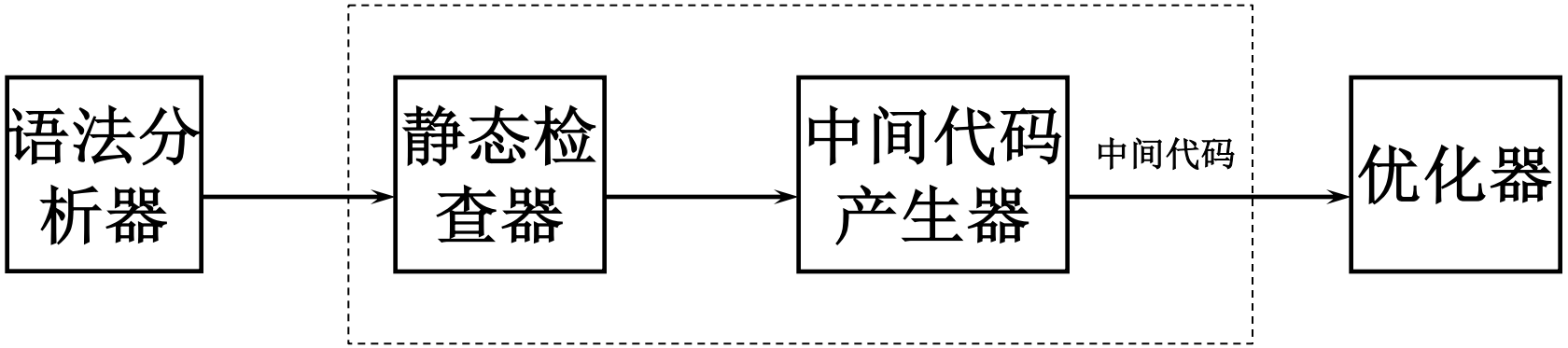
第七章 语义分析和中间代码产生

授课人：高珍

内容线索

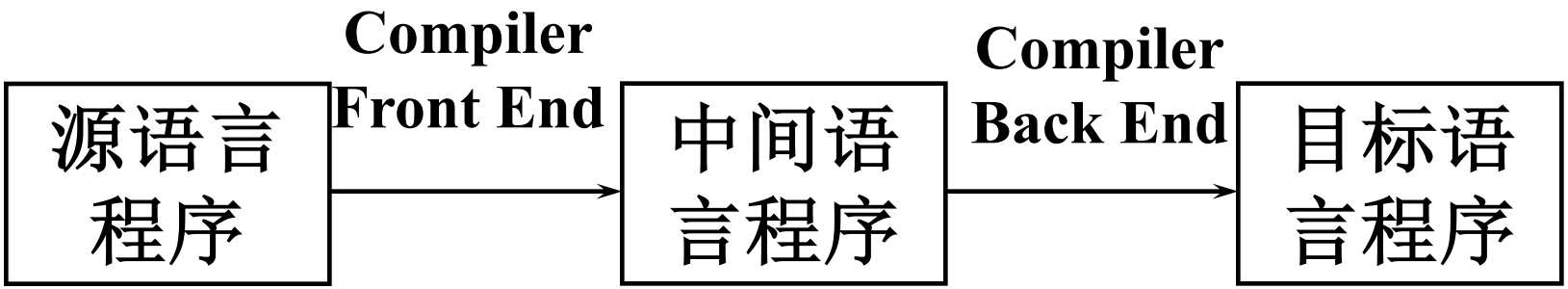
- **中间语言**
- **说明语句**
- **赋值语句的翻译**
- **布尔表达式的翻译**
- **控制语句的翻译**
- **过程调用的处理**

语义分析与中间代码产生



- **静态语义检查**
  - **类型检查**
  - **控制流检查**
  - **一致性检查**
  - **相关名字检查**
  - **名字的作用域分析**

- 中间语言（复杂性界于源语言和目标语言之间）的好处：
  - 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



中间语言

- 常用的中间语言
  - 后缀式
    - 逆波兰表示
  - 图表示
    - DAG
    - 抽象语法树
  - 三地址代码
    - 三元式
    - 四元式
    - 间接三元式

1 后缀式

- **后缀式**表示法：波兰逻辑学家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：
  1. 如果E是一个变量或常量，则E的后缀式是E自身。
  2. 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。
  3. 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。

- 逆波兰表示法不用括号。只要知道每个算符的目数，对于后缀式，不论从哪一端进行扫描，都能对它进行唯一分解。
- 后缀式的计算
  - 用一个栈实现。
  - 一般的计算过程是：自左至右扫描后缀式，每碰到运算量就把它推进栈。每碰到k目运算符就把它作用于栈顶的k个项，并用运算结果代替这k个项。

把表达式翻译成后缀式的语义规则描述

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

随堂练习

- **给出下面表达式的逆波兰表示（后缀式）**

(1) a+b*(c+d/e)

(2) (A and B) or(not C or D)

(3) -a+b*(-c+d)

(4) (A or B) and (C or not D and E)

(5) a+a*(b-c)+(b-c)*d

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

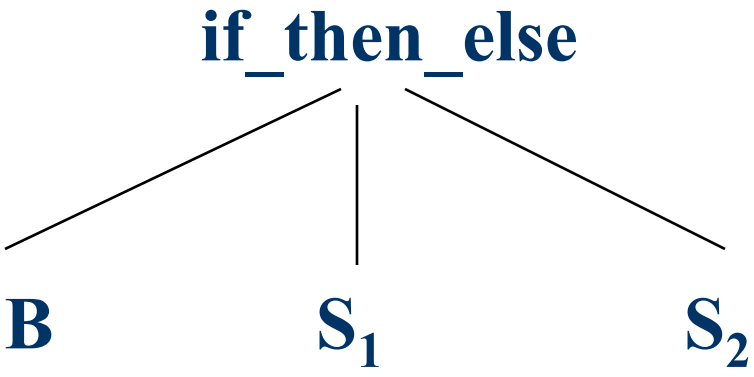
2 图表示法

- 抽象语法树
- DAG

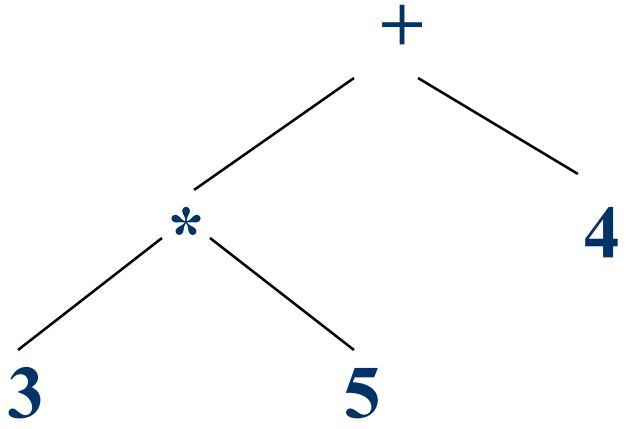
抽象语法树

- 在语法树中去掉那些对翻译不必要的信息，从而获得更有效的源程序中间表示。这种经变换后的语法树称之为**抽象语法树**(Abstract Syntax Tree)
- 操作符和关键字都不作为叶子结点，而作为内部节点

- $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$



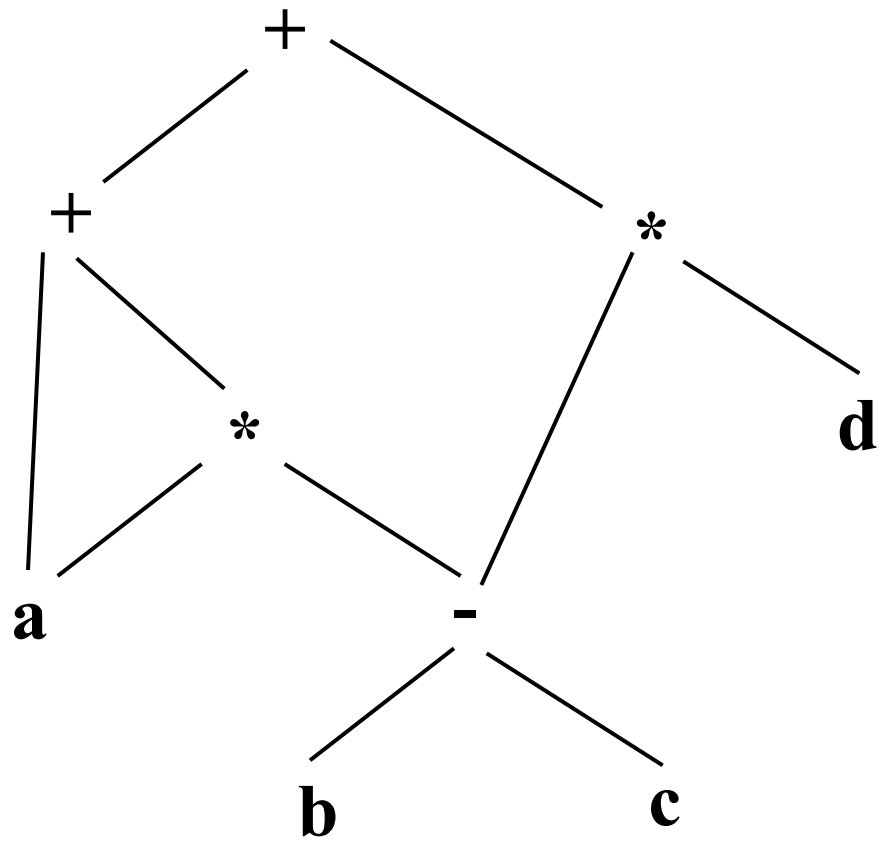
- 3*5+4



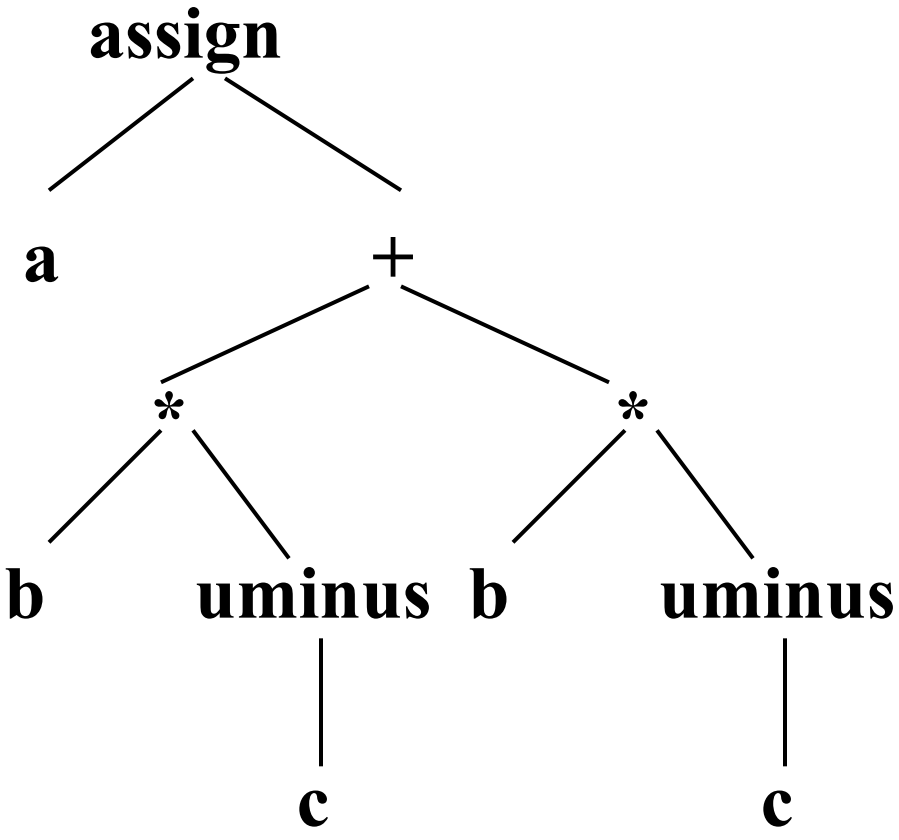
DAG

- 有向无循环图(Directed Acyclic Graph，简称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

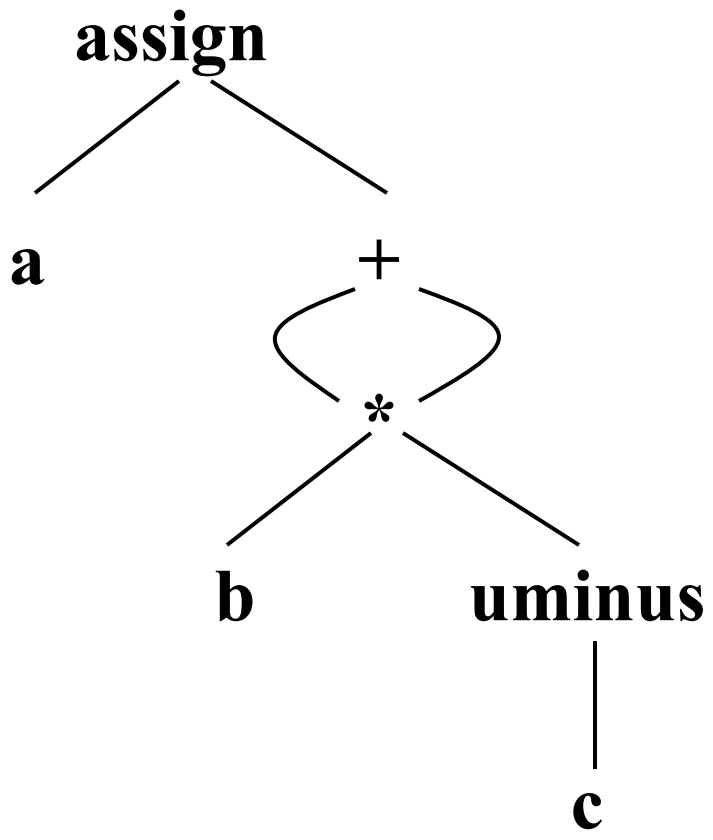
a+a*(b-c)+(b-c)*d的图表示法



a:=b*(-c)+b*(-c)的图表示法



抽象语法树



DAG

3 三地址代码

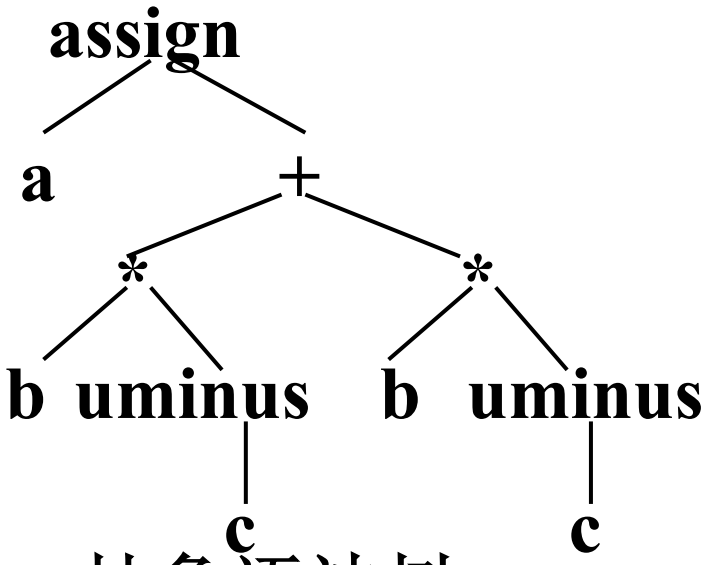
- 三地址代码

  x:=y op z

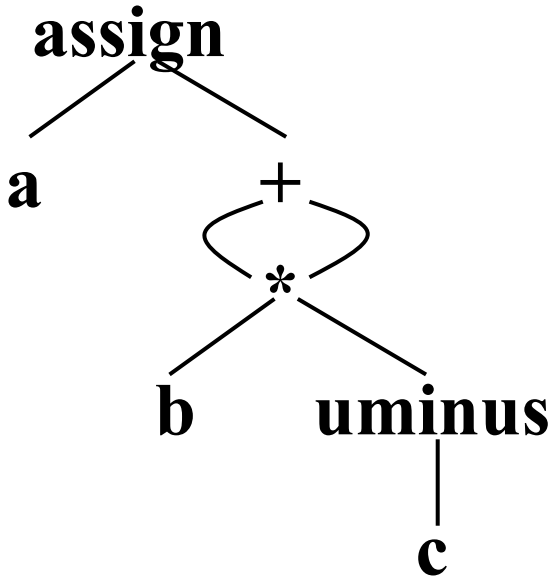
  那么x+y*z如何表示?

- 三地址代码可以看成是抽象语法树或DAG的一种线性表示

a:=b*(-c)+b*(-c)的图表示法



抽象语法树



DAG

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对应的代码：

T_1:=-c
T_2:=b*T_1
T_5:=T_2+T_2
a:=T_5

*

三地址语句的种类

本书中所使用的三地址语句的种类

- x:=y op z
- x:=op y
- x:=y
- goto L
- if x relop y goto L或if a goto L
- param x和call p,n，以及返回语句return y
- x:=y[i]及x[i]:=y的索引赋值
- x:=&y, x:=*y和*x:=y的地址和指针赋值

三地址语句

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

- 四元式
  - 一个带有四个域的记录结构，这四个域分别称为op, arg1, arg2及result

	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

  - 四元式之间的联系通过临时变量实现。
  - 单目运算只用arg1域，转移语句将目标标号放入result域。
  - arg1,arg2,result通常为指针，指向有关名字的符号表入口，且临时变量填入符号表。

三地址语句

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

- 三元式
  - 通过计算临时变量值的语句的位置来引用这个临时变量
  - 三个域：op、arg1和arg2

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

-(a+b)*(c+d)-(a+b+c) 表达式的三地址代码：

T_1: =a+b

T_2: =-T_1

T_3: =c+d

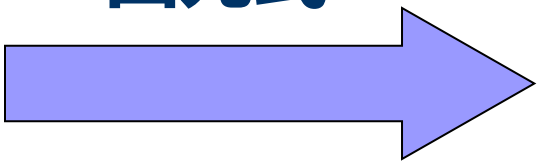
T_4: =T_2* T_3

T_5: =a+b

T_6: = T_5+c

T_7: = T_4-T_6

四元式



	op	arg1	arg2	result
(1)	+	a	b	T_1
(2)	uminus	T_1		T_2
(3)	+	c	d	T_3
(4)	*	T_2	T_3	T_4
(5)	+	a	b	T_5
(6)	+	T_5	c	T_6
(7)	-	T_4	T_6	T_7

-(a+b)*(c+d)-(a+b+c) 表达式的三地址代码：

T_1: =a+b

T_2: =-T_1

T_3: =c+d

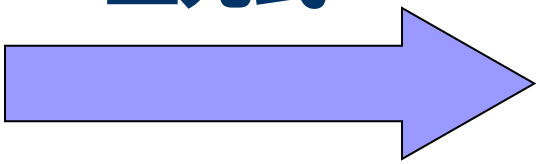
T_4: =T_2* T_3

T_5: =a+b

T_6: = T_5+c

T_7: = T_4-T_6

三元式



	op	arg1	arg2
(1)	+	a	b
(2)	uminus	(1)	
(3)	+	c	d
(4)	*	(2)	(3)
(5)	+	a	b
(6)	+	(5)	c
(7)	-	(4)	(6)

-(a+b)*(c+d)-(a+b+c) 表达式的三地址代码：

T_1: =a+b

T_2: =-T_1

T_3: =c+d

T_4: =T_2* T_3

T_5: =a+b

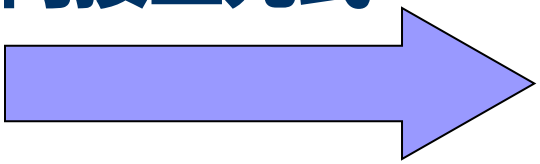
T_6: = T_5+c

T_7: = T_4-T_6

	op	arg1	arg2
(1)	+	a	b
(2)	uminus	(1)	
(3)	+	c	d
(4)	*	(2)	(3)
(5)	+	(1)	c
(6)	-	(4)	(5)

间接代码
(1)
(2)
(3)
(4)
(1)
(5)
(6)

间接三元式



三地址语句

- 间接三元式
  - 为了便于代码优化，用三元式表+间接码表表示中间代码
    - 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。
  - 优点: 方便优化，节省空间

四元式、三元式和间接三元式比较

- 三元式中使用了指向三元式的指针，优化时修改较难。
- 间接三元式优化只需要更改间接码表，并节省三元式表存储空间。
- 修改四元式表也较容易，只是临时变量要填入符号表，占据一定存储空间。

内容线索

- ✓ 中间语言
- ■ **说明语句的翻译**
- ■ 赋值语句的翻译
- ■ 布尔表达式的翻译
- ■ 控制语句的翻译
- ■ 过程调用的处理

内容线索

- ✓ 中间语言
- ✓ 说明语句的翻译
- ■ 赋值语句的翻译
- ■ 布尔表达式的翻译
- ■ 控制语句的翻译
- ■ 过程调用的处理

赋值语句的翻译

- 简单算术表达式及赋值语句
  - 简单算术表达式及赋值语句翻译为三地址代码的翻译模式
    - 属性id.name 表示id所代表的名字本身
    - 过程lookup(id.name)检查是否在符号表中存在相应此名字的入口。如果有，则返回一个指向该表项的指针，否则，返回nil表示没有找到
    - 过程emit将生成的三地址语句发送到输出文件中

产生赋值语句三地址代码的翻译模式

S→id:=E　S.code:=E.code || gen(id.place ':=' E.place)
$E \rightarrow E_1+E_2$　E.place:=newtemp;
E.code:=E_1.code || E_2.code ||gen(E.place ':=' E_1.place '+' E_2.place)
$E \rightarrow E_1*E_2$　E.place:=newtemp;
E.code:=E_1.code || E_2.code || gen(E.place ':=' E_1.place '*' E_2.place)

S→id:=E　{ p:=lookup(id.name);
if p≠nil then
emit(p ':=' E.place)
else error }

$E \rightarrow E_1+E_2$　{ E.place:=newtemp;
emit(E.place ':=' E_1.place '+' E_2.place)}

$E \rightarrow E_1*E_2$　{ E.place:=newtemp;
emit(E.place ':=' E_1.place '*' E_2.place)}

$E \rightarrow -E_1$	E.place:=newtemp; E.code:=E_1.code || gen(E.place ':=' 'uminus' E_1.place)
$E \rightarrow (E_1)$	E.place:=E_1.place; E.code:=E_1.code
$E \rightarrow id$	E.place:=id.place; E.code=' '

$E \rightarrow -E_1$ { E.place:=newtemp;
emit(E.place':='‘uminus’E_1.place)}

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

$E \rightarrow id$ { p:=lookup(id.name);
if p≠nil then
E.place:=p
else error }

类型转换

- 用E.type表示非终结符E的类型属性
- 对应产生式$E \rightarrow E_1$ op E_2的语义动作中关于E.type的语义规则可定义为：

  { if E_1.type=integer and E_2.type=integer
  E.type:=integer
  else E.type:=real }
- 进行类型转换的三地址代码

  x:= inttoreal y
- 算符区分为整型算符int op和实型算符real op，

例. x:=y + i*j

其中x、y为实型；i、j为整型。这个赋值句产生的三地址代码为：

T_1:=i int* j

T_3:=inttoreal T_1

T_2:=y real+ T_3

x:=T_2

关于产生式$E \rightarrow E_1 + E_2$的语义动作

{ E.place:=newtemp;
if E_1.type=integer and
E_2.type=integer then begin
emit (E.place ':=' E_1.place
'int+' E_2.place);
E.type:=integer
end
else if E_1.type=real and
E_2.type=real then begin
emit (E.place ':=' E_1.place
'real+' E_2.place);
E.type:=real
end
else if E_1.type=integer and E_2.type=real
then begin
u:=newtemp;
emit (u ':=' 'inttoreal' E_1.place);
emit (E.place ':=' u 'real+' E_2.palce);
E.type:=real
end
else if E_1.type=real and E_2.type=integer
then begin
u:=newtemp;
emit (u ':=' 'inttoreal' E_2.place);
emit (E.place ':=' E_1.place 'real+' u);
E.type:=real
end
else E.type:=type_error}

随堂练习：写出下面赋值句A:=B*(-C+D)的自下而上语法制导翻译过程。给出所产生的三地址代码。

S→id:=E { p:=lookup(id.name);
if p≠nil then
emit(p ':=' E.place)
else error }

$E \rightarrow E_1+E_2$ { E.place:=newtemp;
emit(E.place ':=' E_1.place '+' E_2.place)}

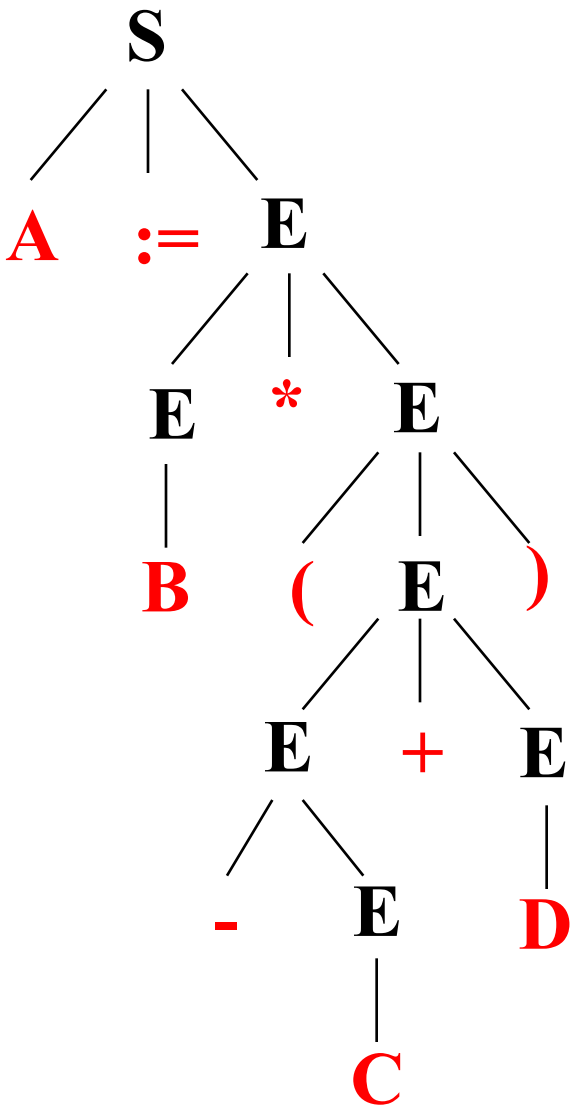
$E \rightarrow E_1*E_2$ { E.place:=newtemp;
emit(E.place ':=' E_1.place '*' E_2.place)}

$E \rightarrow -E_1$ { E.place:=newtemp;
emit(E.place':='‘uminus’E_1.place)}

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

E→id { p:=lookup(id.name);
if p≠nil then
E.place:=p
else error }

随堂练习：写出下面赋值句A:=B*(-C+D)的自下而上语法制导翻译过程。给出所产生的三地址代码。



内容线索

- ✓ 中间语言
- ✓ 说明语句的翻译
- ✓ 赋值语句的翻译
- ■ 布尔表达式的翻译
- ■ 控制语句的翻译
- ■ 过程调用的处理

布尔表达式的翻译

- **布尔表达式：用布尔运算符把布尔量、关系表达式联结起来的式子。**
  - 布尔运算符：and, or, not ；
  - 关系运算符 < ,≤, = , ≠， > ,≥
- **布尔表达式的两个基本作用:**
  - 用于逻辑演算,计算逻辑值;
  - 用于控制语句的条件式.
- **产生布尔表达式的文法:**
  - E→E or E | E andE | ¬ E | (E) | id rop id | id
- **运算符优先级：布尔运算由高到低:not and or，同级左结合**
  关系运算符同级，且高于布尔运算符

- 计算布尔表达式通常采用两种方法:

(1) 如同计算算术表达式一样,一步步算

1 or (not 0 and 0) or 0

=1 or (1 and 0) or 0

=1 or 0 or 0

=1 or 0

=1

(2) 采用优化措施(短路计算)

把A or B解释成　　if A then true else B

把A and B解释成　　if A then B else false

把¬ A解释成　　if A then false else true

对应的，有两种布尔表达式翻译方法

数值表示法

- a or b and not c 翻译成

  T_1:=not c

  T_2:=b and T_1

  T_3:=a or T_2

- a<b的关系表达式可等价地写成

  if a<b then 1 else 0， 翻译成

```
100:    if a<b goto 103
101:    T:=0
102:    goto 104
103:    T:=1
104:
```

数值表示法的翻译模式

- 过程emit将三地址代码送到输出文件中
- nextstat：给出输出序列中下一条三地址语句的地址索引
- 每产生一条三地址语句后，过程emit便把nextstat加1

数值表示法的翻译模式

$E \rightarrow E_1$ or E_2 {E.place:=newtemp;
emit(E.place ‘:=’ E_1.place ‘or’ E_2.place)}

$E \rightarrow E_1$ and E_2 {E.place:=newtemp;
emit(E.place ‘:=’ E_1.place ‘and’ E_2.place)}

$E \rightarrow$ not E_1 {E.place:=newtemp;
emit(E.place ‘:=’ ‘not’ E_1.place)}

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

数值表示法的翻译模式

a<b 翻译成

```
100:  if a<b goto 103
101:  T:=0
102:  goto 104
103:  T:=1
104:
```

$E \rightarrow id_1$ relop id_2 { E.place:=newtemp;
emit('if' id_1.place relop. op
id_2. place 'goto' nextstat+3);
emit(E.place ':=' '0');
emit('goto' nextstat+2);
emit(E.place':=' '1') }

$E \rightarrow id$ { E.place:=id.place }

布尔表达式a<b or c<d and e<f的翻译结果

100: if a<b goto 103
101: T_1:=0
102: goto 104
103: T_1:=1
104: if c<d goto 107
105: T_2:=0
106: goto 108
107: T_2:=1
108: if e<f goto 111
109: T_3:=0
110: goto 112
111: T_3:=1
112: T_4:=T_2 and T_3
113: T_5:=T_1 or T_4

$E \rightarrow id_1$ relop id_2
{ E.place:=newtemp;
emit('if' id_1.place relop. op id_2. place
'goto' nextstat+3);
emit(E.place ':=' '0');
emit('goto' nextstat+2);
emit(E.place':=' '1') }

$E \rightarrow id$
{ E.place:=id.place }

$E \rightarrow E_1$ or E_2
{ E.place:=newtemp;
emit(E.place ':=' E_1.place 'or' E_2.place)}

$E \rightarrow E_1$ and E_2
{ E.place:=newtemp;
emit(E.place ':=' E_1.place 'and' E_2.place)}

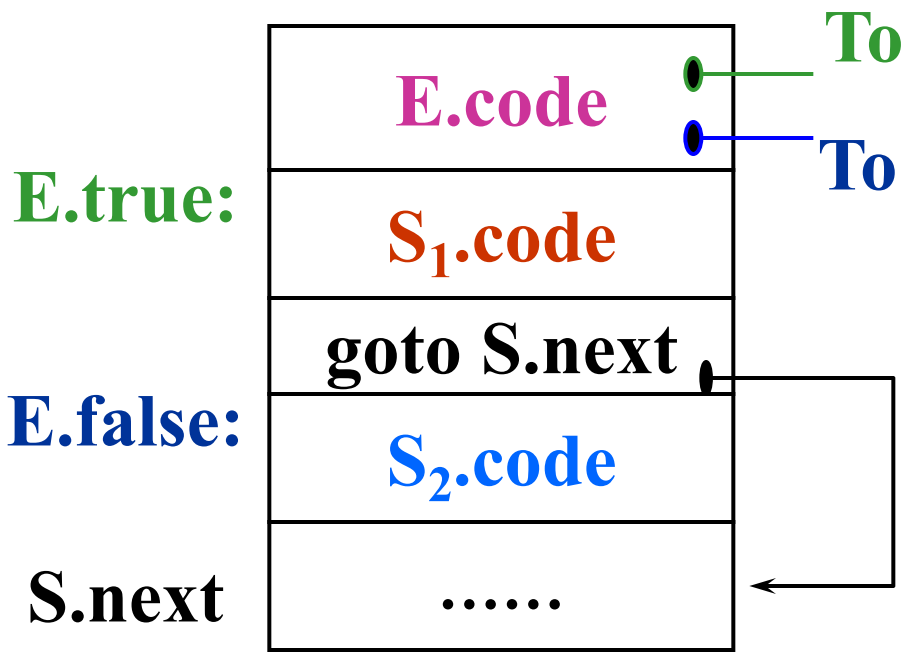
*

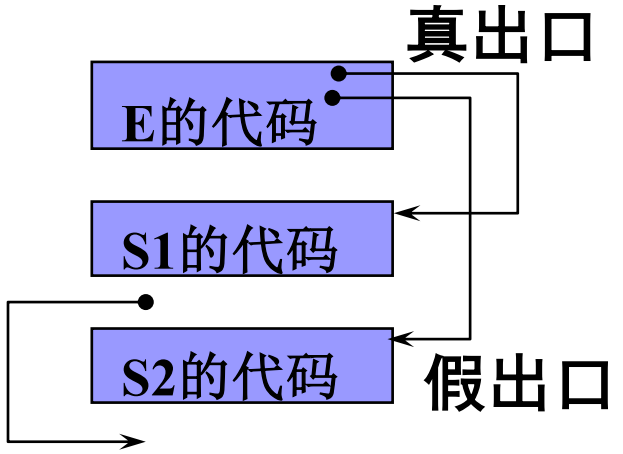
随堂练习

- **求布尔表达式的翻译结果**
  - **a>b and c>d**

作为条件控制的布尔式翻译

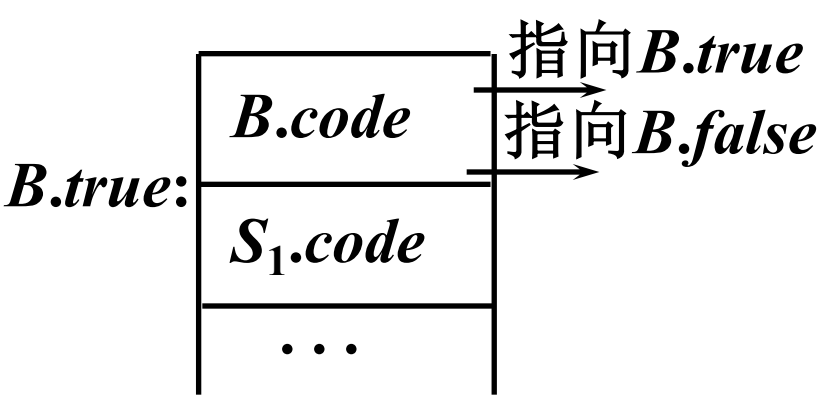
- 条件语句 if E then S_1 else S_2
  赋予 E 两种出口:一真一假



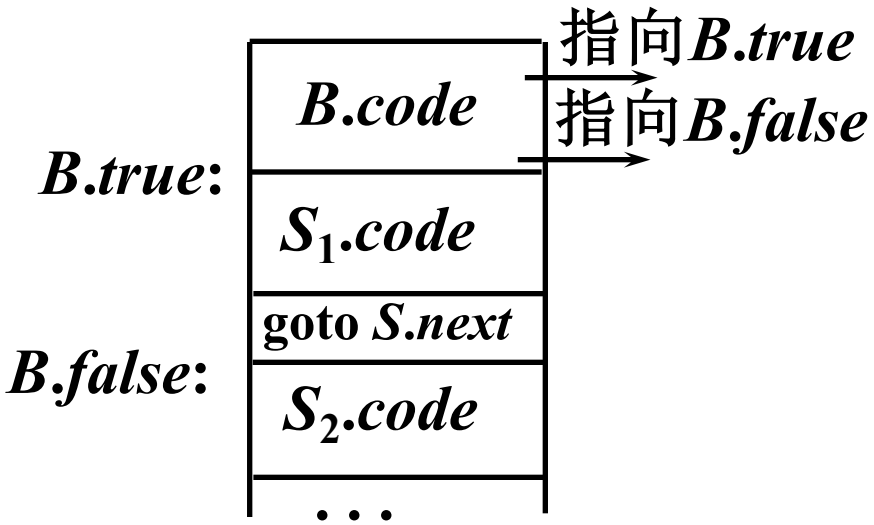


布尔表达式和控制流语句

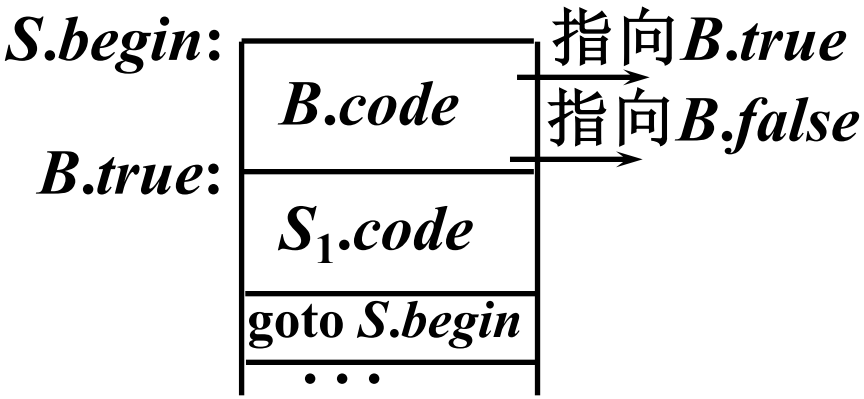
控制流语句的翻译



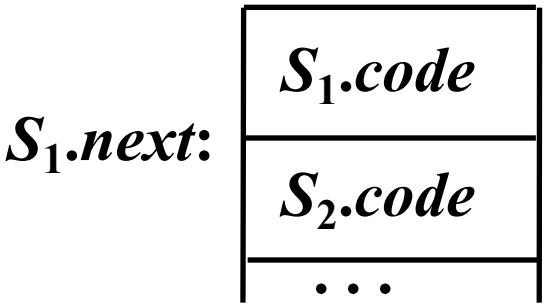
(a) if-then



(b) if-then-else



(c) while-do



(d) S_1; S_2

布尔表达式的翻译

- 两遍扫描
  - 为给定的输入串构造一棵语法树；
  - 对语法树进行深度优先遍历，进行语义规则中规定的翻译。
- 一遍扫描

一遍扫描实现布尔表达式的翻译

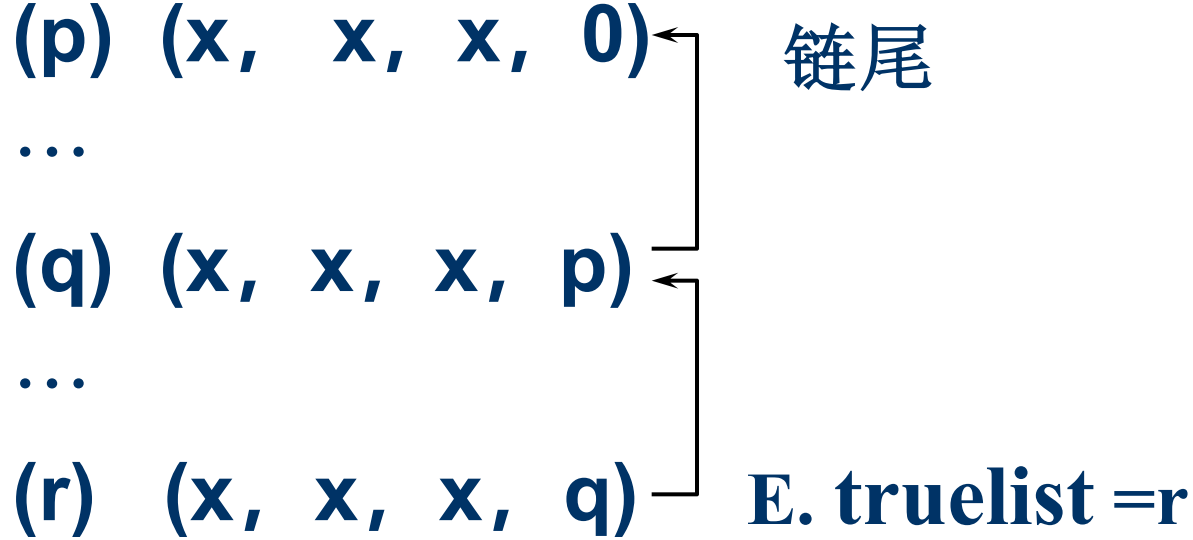
- 采用四元式形式
- 把四元式存入一个数组中，数组下标就代表四元式的标号
- 约定

  四元式(jnz, a, -, p)　表示　if a goto p

  四元式(jrop, x, y, p)表示　if x rop y goto p

  四元式(j, -, -, p)　　表示　goto p
- 有时,四元式转移地址无法立即知道,我们只好把这个未完成的四元式地址作为E的语义值保存,待机"回填"。

- 为非终结符E赋予两个综合属性E.truelist和E.falselist。它们分别记录布尔表达式E所应的四元式中需回填“真”、“假”出口的四元式的标号所构成的链表
- 例如:假定E的四元式中需要回填"真"出口的p，q，r三个四元式，则E.truelist为下列链:



- 为了处理E.truelist和E.falselist ，引入下列语义变量和过程:
  - 变量nextquad，它指向下一条将要产生但尚未形成的四元式的地址(标号)。nextquad的初值为1，每当执行一次emit之后，nextquad将自动增1。
  - 函数makelist(i)，它将创建一个仅含i的新链表，其中i是四元式数组的一个下标(标号)；函数返回指向这个链的指针。
  - 函数merge(p_1,p_2)，把以p_1和p_2为链首的两条链合并为一，作为函数值，回送合并后的链首。
  - 过程backpatch(p, t)，其功能是完成“回填”，把p所链接的每个四元式的第四区段都填为t。

布尔表达式的文法

(1) $E \rightarrow E_1$ or M E_2

(2) | E_1 and M E_2

(3) | not E_1

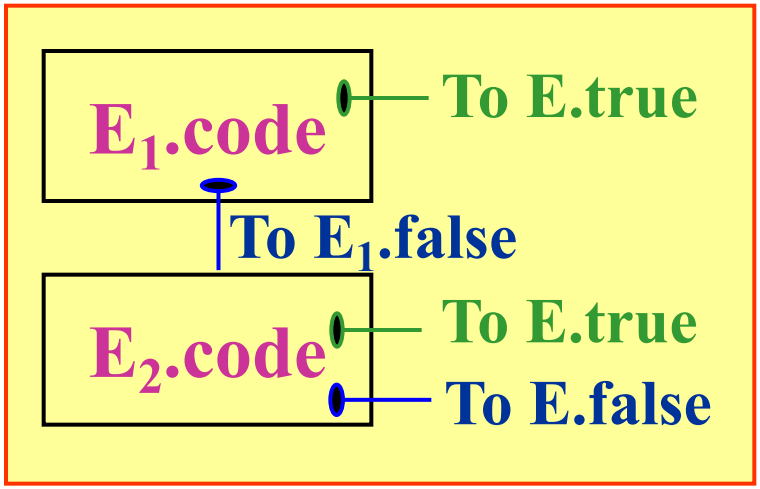
(4) | (E_1)

(5) | id_1 relop id_2

(6) | id

(7) $M \rightarrow \varepsilon$

布尔表达式的翻译模式



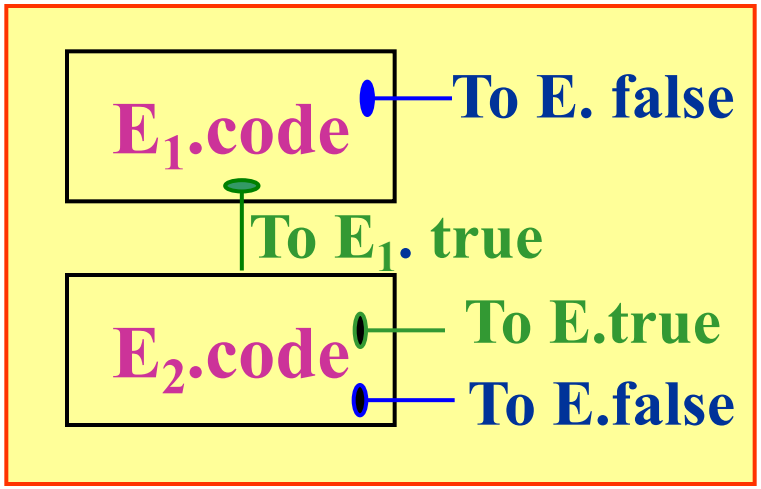
(1) $E \rightarrow E_1$ or M E_2

{ backpatch(E_1.falselist, M.quad);

E.truelist:=merge(E_1.truelist, E_2.truelist);

E.falselist:=E_2.falselist }

布尔表达式的翻译模式



(2) $E \rightarrow E_1$ and M E_2

{ backpatch(E_1.truelist, M.quad);

E.truelist:=E_2.truelist;

E.falselist:=merge(E_1.falselist,E_2.falselist) }

布尔表达式的翻译模式

(3) $E \rightarrow \text{not } E_1$

{ E.truelist:=E_1.falselist;

E.falselist:=E_1.truelist}

(4) $E \rightarrow (E_1)$

{ E.truelist:=E_1.truelist;

E.falselist:=E_1. falselist}

布尔表达式的翻译模式

(5) $E \rightarrow id_1$ relop id_2

{ E.truelist:=makelist(nextquad);
E.falselist:=makelist(nextquad+1);
emit('j' relop.op ',' id $_1$.place ',' id $_2$.place',''0');
emit('j, - , - , 0') }

(6) E→id

{ E.truelist:=makelist(nextquad);
E.falselist:=makelist(nextquad+1);
emit('jnz' ',' id .place ',' ' - ,' 0');
emit(' j, -, -, 0') }

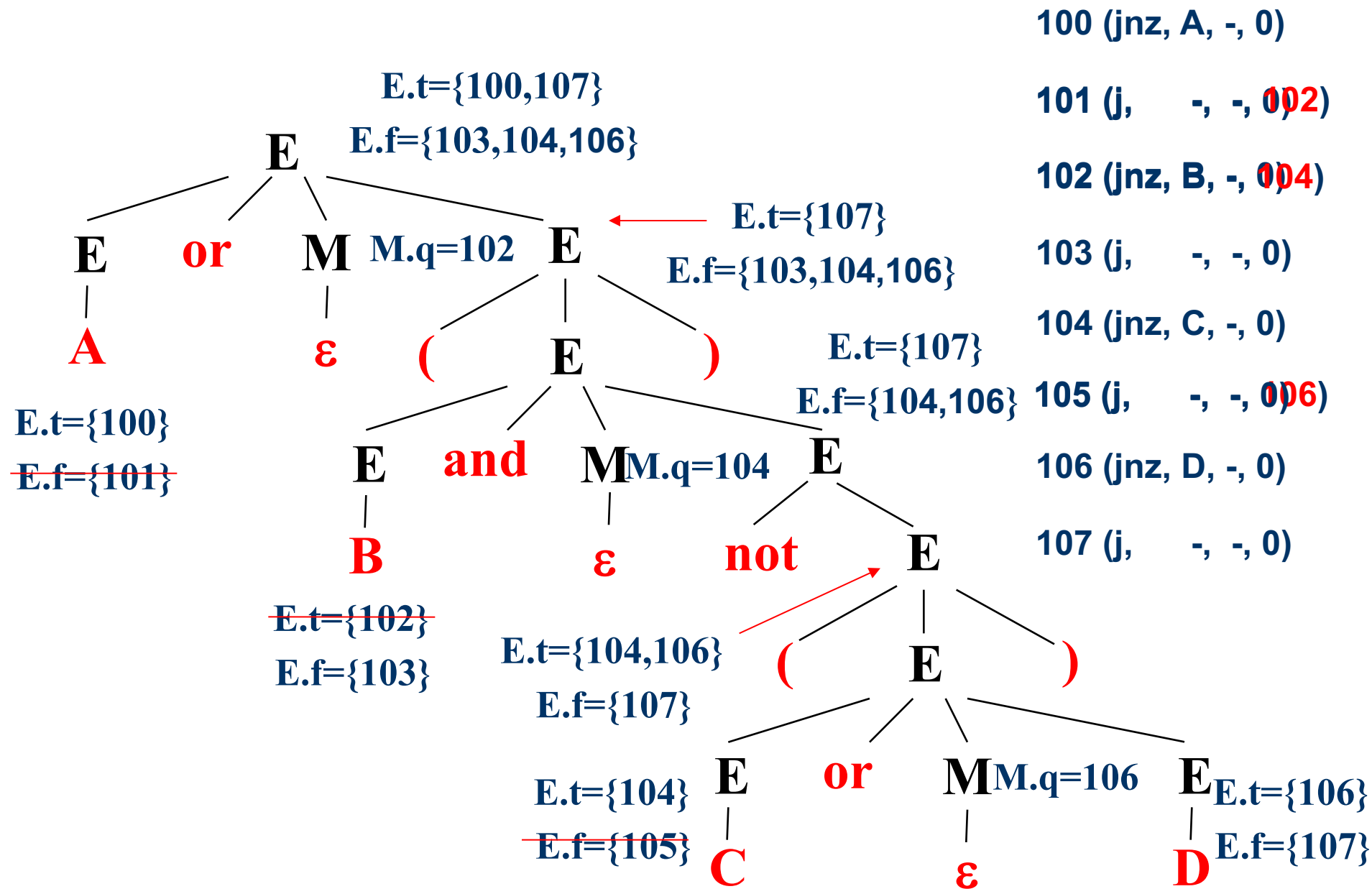
(7) M→ε

{ M.quad:=nextquad }

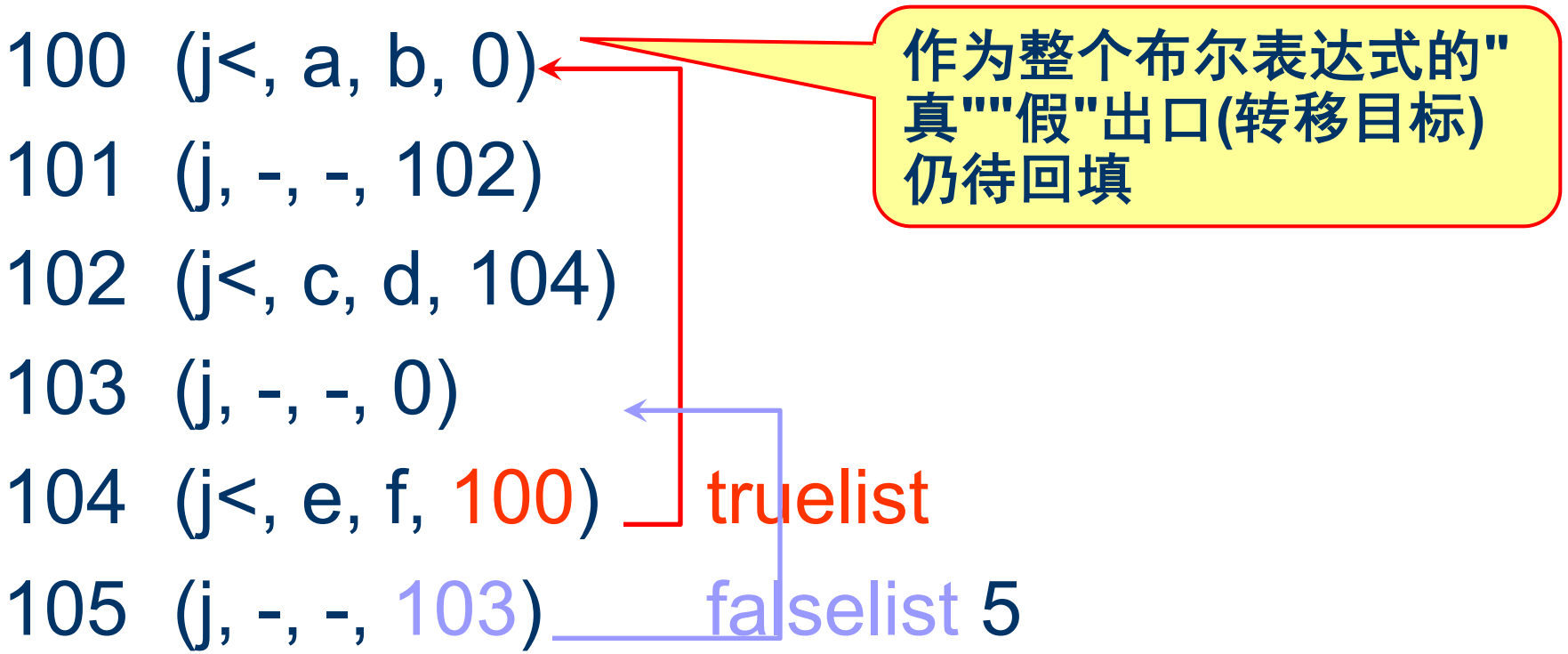
随堂练习

- 写出布尔式A or B的四元式序列
- 写出布尔式A or (B and not (C or D)) 的四元式序列

写出布尔式A or (B and not (C or D)) 的四元式序列。



a<b or c<d and e<f



内容线索

- ✓ 中间语言
- ✓ 说明语句的翻译
- ✓ 赋值语句的翻译
- ✓ 布尔表达式的翻译
- ■ 控制语句的翻译
- ■ 过程调用的处理

一遍扫描翻译控制流语句

- 考虑下列产生式所定义的语句:

  (1) S→if E then S

  (2) | if E then S else S

  (3) | while E do S

  (4) | begin L end

  (5) | A

  (6) L→L;S

  (7) | S

- S表示语句， L表示语句表，
  A为赋值语句， E为一个布尔表达式

if 语句的翻译

相关产生式

$S \rightarrow$ if E then $S^{(1)}$

| if E then $S^{(1)}$ else $S^{(2)}$

改写后产生式

$S \rightarrow$ if E then M S_1

$S \rightarrow$ if E then M_1 S_1 N else M_2 S_2

$M \rightarrow \varepsilon$

$N \rightarrow \varepsilon$

翻译模式:

1. $S \rightarrow$ if E then M S_1
 { backpatch(E.truelist, M.quad);
 S.nextlist:=merge(E.falselist, S_1.nextlist) }

2. $S \rightarrow$ if E then M_1 S_1 N else M_2 S_2
 { backpatch(E.truelist, M_1.quad);
 backpatch(E.falselist, M_2.quad);
 S.nextlist:=merge(S_1.nextlist, N.nextlist, S_2.nextlist) }

3. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

4. $N \rightarrow \varepsilon$ { N.nextlist:=makelist(nextquad);
 emit('j, - , - ,0') }

举例

1. $S \rightarrow$ if E then M S_1
 { backpatch(E.truelist, M.quad);
 S.nextlist:=merge(E.falselist, S_1.nextlist) }

2. $S \rightarrow$ if E then M_1 S_1 N else M_2 S_2
 { backpatch(E.truelist, M_1.quad);
 backpatch(E.falselist, M_2.quad);
 S.nextlist:=merge(S_1.nextlist, N.nextlist, S_2.nextlist) }

3. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

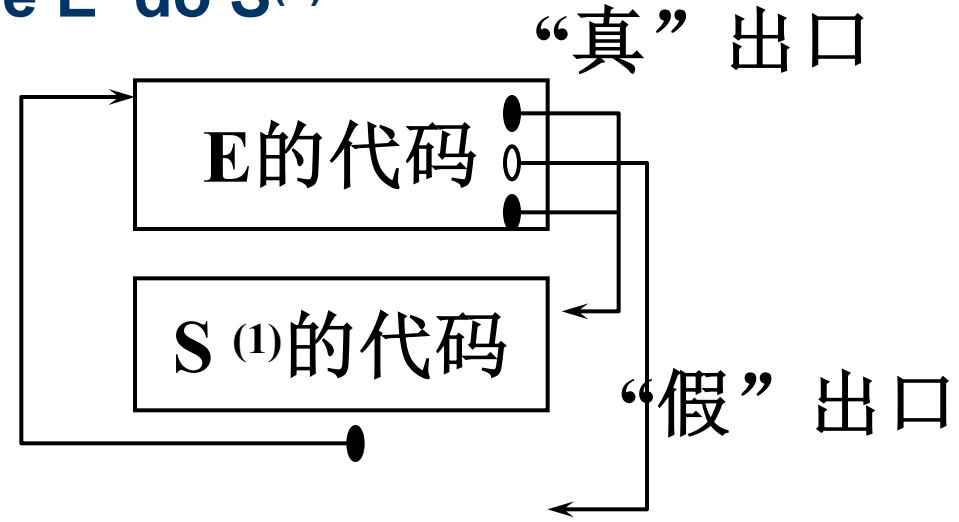
4. $N \rightarrow \varepsilon$ { N.nextlist:=makelist(nextquad);
 emit(‘j, - , - ,0’) }

if a then b:=1 else b:=2 翻译其中间代码

while 语句的翻译

相关产生式

$S \rightarrow$ while E do $S^{(1)}$



为了便于"回填"，改写产生式为:

$S \rightarrow$ while M_1 E do M_2 S_1

$M \rightarrow \varepsilon$

翻译模式:

1. $S \rightarrow$ while M_1 E do M_2 S_1

{backpatch(S_1.nextlist, M_1.quad);

backpatch(S_1.nextlist, nextquad);

backpatch(E.truelist, M_2.quad);

S.nextlist:=E.falselist

emit('j, - , - ,' M_1.quad) }

哪个更好？

2. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

3. $S \rightarrow A$ { S.nextlist:=makelist( ) }

举例

1. $S \rightarrow$ while M_1 E do M_2 S_1

{backpatch(S_1.nextlist, M_1.quad);

backpatch(E.truelist, M_2.quad);

S.nextlist:=E.falselist

emit(‘j, - , - ,’ M_1.quad) }

2. $M \rightarrow \varepsilon$ { M.quad:=nextquad }

while a>b do a:=a-1 翻译其中间代码

语句L→L;S的翻译

产生式

$L \rightarrow L;S \mid S$

改写为:

$L \rightarrow L_1; M\ S \mid S$

$M \rightarrow \varepsilon$

翻译模式:

1. $L \rightarrow L_1; M\ S$ { backpatch(L_1.nextlist, M.quad); L.nextlist:=S.nextlist }
2. $M \rightarrow \varepsilon$ { M.quad:=nextquad }
3. $L \rightarrow S$ { L.nextlist:=S.nextlist }

其它几个语句的翻译

S→begin L end
{ S.nextlist:=L.nextlist }

S→A
{ S.nextlist:=makelist( ) }

举例

1. $L \rightarrow L_1$; M S	{ backpatch(L_1.nextlist, M.quad); L.nextlist:=S.nextlist }
2. $M \rightarrow \varepsilon$	{ M.quad:=nextquad }
3. $L \rightarrow S$	{ L.nextlist:=S.nextlist }

a:=a+1;

b:=b+1

翻译其中间代码

翻译语句

while (a<b) do

if (c<d) then x:=y+z;

P190

(5) $E \rightarrow id_1$ relop id_2 { E.truelist:=makelist(nextquad);

E.falselist:=makelist(nextquad+1);

emit('j' relop.op ',' id_1.place ',' id_2.place',' '0');

emit('j, - , - , 0') }

P179

S→A **{ S.nextlist:=makelist( ) }**

A→id:=E { p:=lookup(id.name);

if p≠nil then

emit(p ':=' E.place)

else error }

$E \rightarrow E_1+E_2$ { E.place:=newtemp;

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

*

翻译语句

while (a<b) do

if (c<d) then x:=y+z;

P195

S→if E then M S_1

{ backpatch(E.truelist, M.quad);

S.nextlist:=merge(E.falselist, S_1.nextlist) }

M→ε { M.quad:=nextquad }

S→A { S.nextlist:=makelist( ) }

P195

S→while M_1 E do M_2 S_1

{ backpatch(S_1.nextlist, M_1.quad);

backpatch(E.truelist, M_2.quad);

S.nextlist:=E.falselist

emit('j, - , - ,' M_1.quad) }

M→ε { M.quad:=nextquad }

*

翻译语句

while (a<b) do

if (c<d) then x:=y+z;

100 (j<, a, b, 102)
101 (j, -, -, 107)
102 (j<, c, d, 104)
103 (j, -, -, 100)
104 (+, y, z, T)
105 (:=, T, -, x)
106 (j, -, -, 100)
107

Canvas作业

- 作业6-中间代码生成
  - P218(7)

```
While A<C and B<D do
        If A=1 then C:=C+1 else
        while A<=D do A:=A+2;
```

标号与goto语句

- 标号定义形式

L: S;

当这种语句被处理之后，标号L称为 *定义了* 的。即在符号表中，标号L的“地址”栏将登记上语句S的第一个四元式的地址。

- 标号引用

goto L;

向后转移：

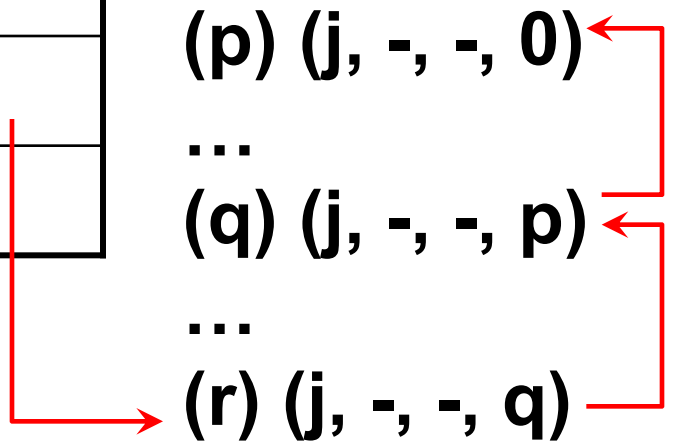
```
L1:   ……
      ……
      goto L1;
```

向前转移：

```
      goto L1;
      ……
L1:   ……
```

符号表信息

名字	类型	…	定义否	地址
…	…	…	…	…
L	标号		未	r



产生式S′→goto L的语义动作:

```
{  查找符号表;
   IF  L在符号表中且"定义否"栏为"已"
     THEN GEN(J，-，-，P) #P为地址栏上的编号
     ELSE IF  L不在符号表中
          THEN BEGIN
                    把L填入表中;
                    置"定义否"为"未"，"地址"栏为Nextquad;
                    GEN(J，-，-，0)
          END
     ELSE IF  L在符号表中且“定义否”栏为“未"
          ELSE BEGIN
                    q:=L的地址栏中的编号;
                    置地址栏编号为Nextquad;
                    GEN(J，-，-，q)
          END
}
```

- 带标号语句的产生式:

  $S_L \rightarrow$ label S　　label $\rightarrow$ i:

- label $\rightarrow$ i: 对应的语义动作:

1. 若i所指的标识符(假定为L)不在符号表中，则把它填入，置"类型"为"标号"，定义否为"已"，"地址"为nextquad；
2. 若L已在符号表中但"类型"不为标号或"定义否"为"已"，则报告出错；
3. 若L已在符号表中，则把标号"未"改为"已"，然后，把地址栏中的链头(记为q)取出，同时把nextquad填在其中，最后，执行BACKPATCH(q，nextquad)。

课堂作业Canvas

向后转移:

```
L1:    a:=1
goto L1;
```

向前转移:

```
goto L1;
L1: a:=1
```

请写出其中间代码以及符号表的变化

CASE语句的翻译

- 语句结构

case E of

C_1: S_1;

C_2: S_2;

…

C_{n-1}: S_{n-1};

otherwise: S_n

end

E是一个表达式，称为选择子。E通常是一个整型表达式或字符型变量。

- 翻译法(一):

T:=E

L_1: if $T \neq C_1$ goto L_2

S_1的代码

goto next

L_2: if $T \neq C_2$ goto L_3

S_2的代码

goto next

L_3:

…

L_{n-1}: if $T \neq C_{n-1}$ goto L_n

S_{n-1}的代码

goto next

L_n: S_n的代码

next:

- 翻译法(二):

计算E并放入T中

goto test

L_1: 关于S_1的中间码

goto next

…

L_{n-1}: 关于S_{n-1}的中间码

goto next

L_n: 关于S_n的中间码

goto next

test: if $T=C_1$ goto L_1

if $T=C_2$ goto L_2

…

if $T=C_{n-1}$ goto L_{n-1}

goto L_n

next: 易于生成多向转移的目标指令

课堂作业

```
case  a+1  of
      1:    b:=2;
      2:    b:=4;
      3:    b:=9;
      otherwise:    b:=0
end
```

- 翻译法(二):

```
        计算E并放入T中
        goto test
L1:     关于S1的中间码
        goto next
…
Ln-1:   关于Sn-1的中间码
        goto next
Ln:     关于Sn的中间码
        goto next
test:   if T=C1 goto L1
        if T=C2 goto L2
        …
        if T=Cn-1 goto Ln-1
        goto Ln
next:
```

内容线索

- ✓ 中间语言
- ✓ 说明语句的翻译
- ✓ 赋值语句的翻译
- ✓ 布尔表达式的翻译
- ✓ 控制语句的翻译
- ■ 过程调用的处理

过程调用的处理

- 过程调用主要对应两种事:
  - 传递参数
  - 转子（过程）
- 传地址:把实在参数的地址传递给相应的形式参数
  - 把实在参数的地址抄进对应的形式单元中;
  - 过程体对形式参数的引用与赋值被处理成对形式单元的间接访问。

  备注：只讨论“传地址”方式

过程调用的文法

- **过程调用文法:** 三地址代码：param、call语句

  (1) S → call id (Elist)

  (2) Elist → Elist, E

  (3) Elist → E

- **参数的地址存放在一个队列中**
- **最后对队列中的每一项生成一条param语句**

过程调用的翻译

- 翻译方法：把实参的地址逐一放在转子指令的前面.

例如， CALL S(A，X+Y) 翻译为:

中间代码：计算X+Y，置于T中

```
param A     /*第一个参数的地址*/
param T     /*第二个参数的地址*/
call S      /*转子*/
```

翻译模式

1. S→call id (Elist)

   { for 队列queue中的每一项p do

   emit(‘param’ p);

   emit(‘call’ id.place) }

2. Elist→Elist, E

   { 将E.place加入到queue的队尾 }

3. Elist→E

   { 初始化queue仅包含E.place }

课堂作业

- CALL f(a，b)的翻译结果?

内容线索

- ✓ 中间语言
- ✓ 说明语句的翻译
- ✓ 赋值语句的翻译
- ✓ 布尔表达式的翻译
- ✓ 控制语句的翻译
- ✓ 过程调用的处理

Dank u
Dutch

Merci
French

Спасибо
Russian

Gracias
Spanish

شكراً
Arabic

감사합니다
Korean

Tack så mycket
Swedish

धन्यवाद
Hindi

תודה רבה
Hebrew

Obrigado
Brazilian Portuguese

谢谢
Chinese

Thank You !

Dankon
Esperanto

ありがとうございます
Japanese

Trugarez
Breton

Tak
Danish

Danke
German

Grazie
Italian

நன்றி
Tamil

děkuji
Czech

ขอบคุณ
Thai

go raibh maith agat
Gaelic

*