



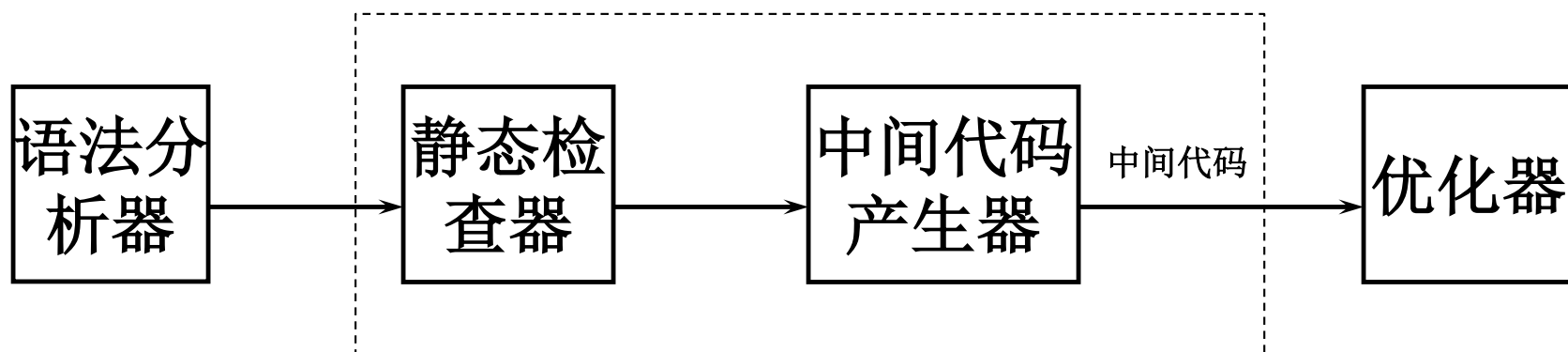
第七章 语义分析和 中间代码产生

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

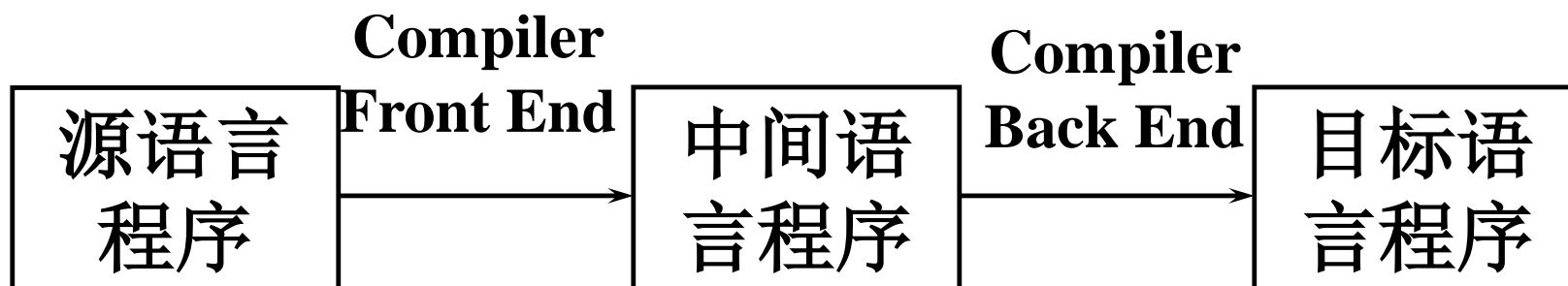


■ 静态语义检查

- ☐ 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（复杂性介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 $E_1 \text{ op } E_2$ 形式的表达式，其中op是任何二元操作符，则E的**后缀式**为 $E_1' E_2' \text{ op}$ ，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
$E \rightarrow E^{(1)} \text{ op } E^{(2)}$	$E.\text{code} := E^{(1)}.\text{code} \parallel E^{(2)}.\text{code} \parallel \text{op}$
$E \rightarrow (E^{(1)})$	$E.\text{code} := E^{(1)}.\text{code}$
$E \rightarrow \text{id}$	$E.\text{code} := \text{id}$

- $E.\text{code}$ 表示E后缀形式
- op 表示任意二元操作符
- “ \parallel ” 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) $(A \text{ and } B) \text{ or } (\text{not } C \text{ or } D)$

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

(4) $(A \text{ or } B) \text{ and } (C \text{ or not } D \text{ and } E)$

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

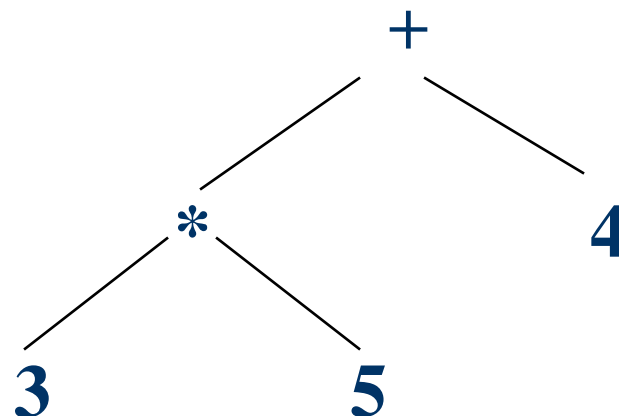
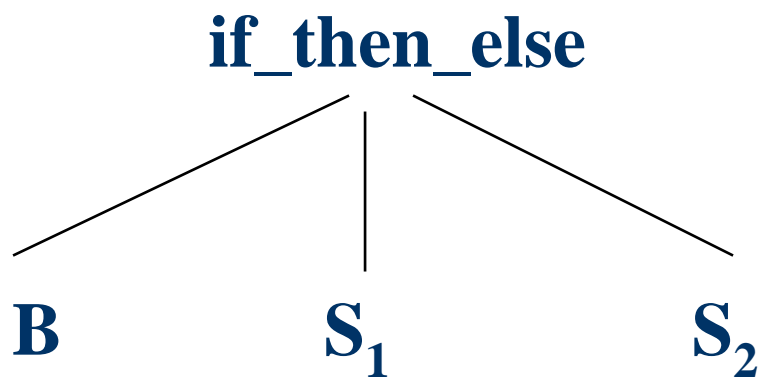
(6) $b:=-c*a+-c*a$

2 图表示法

- 抽象语法树
- DAG

抽象语法树

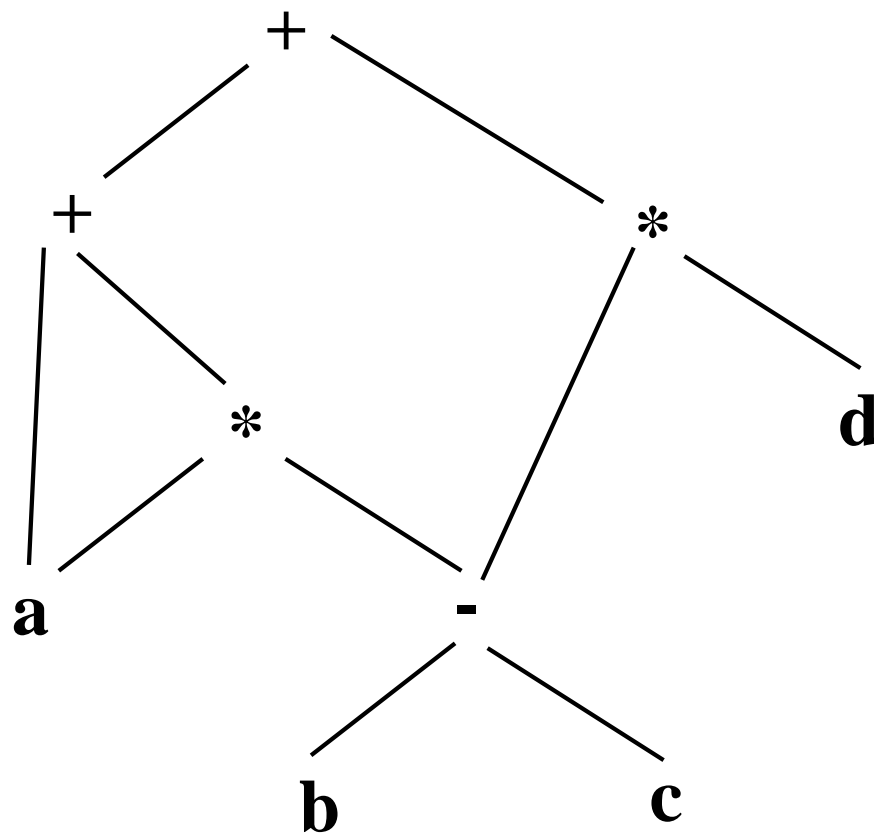
- 在语法树中去掉那些对翻译不必要的信息，从而获得更有效的源程序中间表示。这种经变换后的语法树称之为**抽象语法树** (Abstract Syntax Tree)
 - 操作符和关键字都不作为叶子结点，而作为内部节点
- $S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$ □ $3 * 5 + 4$



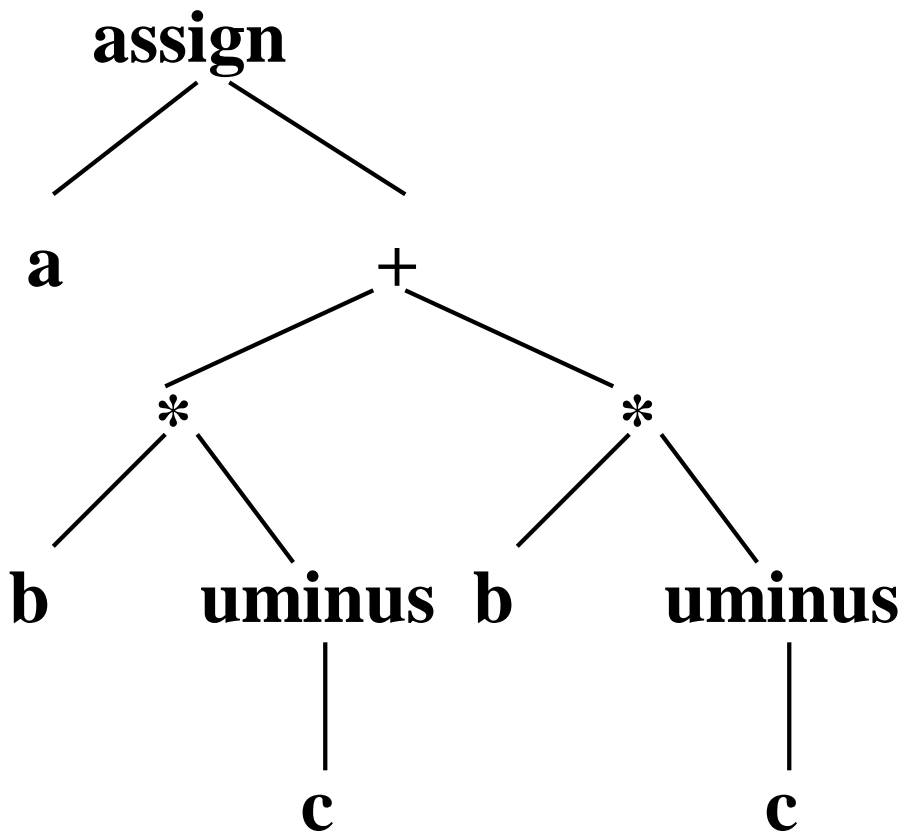
DAG

- 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

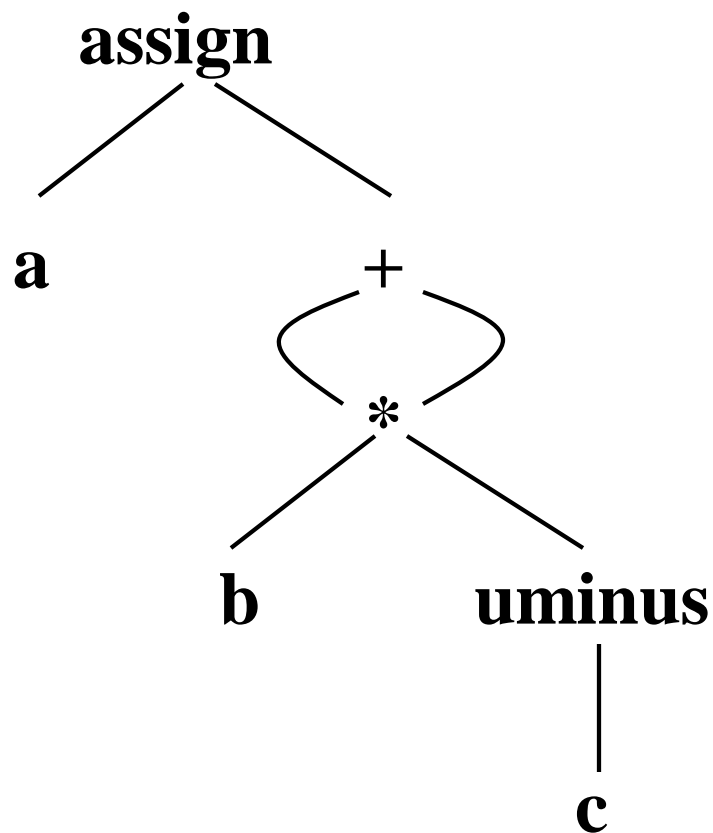
$a+a*(b-c)+(b-c)*d$ 的图表示法



$a := b * (-c) + b * (-c)$ 的图表示法



抽象语法树



DAG

3 三地址代码

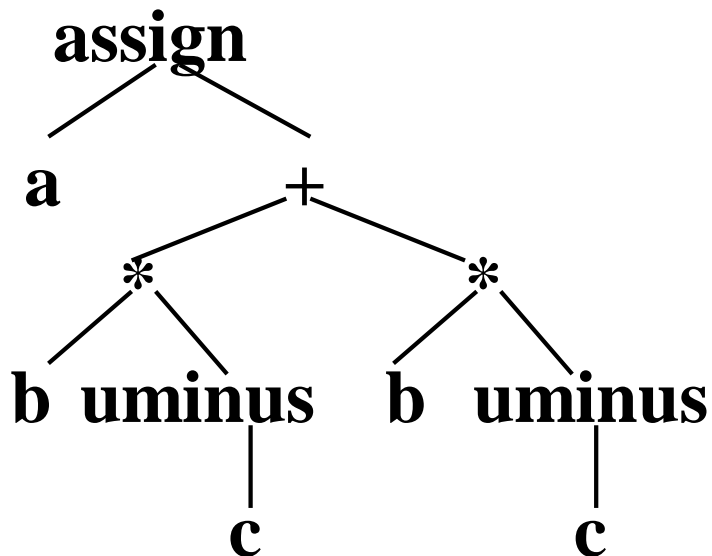
- 三地址代码

$x := y \text{ op } z$

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

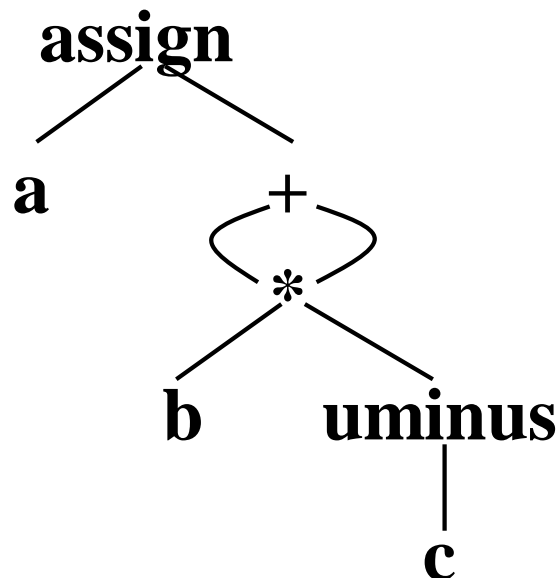
$a := b * (-c) + b * (-c)$ 的图表示法



抽象语法树

抽象语法树对应的代码:

```
T1 := -c  
T2 := b * T1  
T3 := -c  
T4 := b * T3  
T5 := T2 + T4  
a := T5
```



DAG

DAG对应的代码:

```
T1 := -c  
T2 := b * T1  
T5 := T2 + T2  
a := T5
```


三地址语句的种类

本书中所使用的三地址语句的种类

- $x := y \text{ op } z$
- $x := \text{op } y$
- $x := y$
- `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)

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

$-(a+b)*(c+d)-(a+b+c)$ 表达式的三地址代码:

$T_1 := a+b$

$T_2 := -T_1$

$T_3 := c+d$

$T_4 := T_2 * T_3$

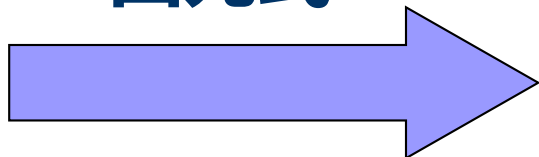
$T_5 := a+b$

$T_6 := T_5 + c$

$T_7 := T_4 - T_6$

	op	arg1	arg2	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

四元式



$-(a+b)*(c+d)-(a+b+c)$ 表达式的三地址代码:

$T_1 := a+b$

$T_2 := -T_1$

$T_3 := c+d$

$T_4 := T_2 * T_3$

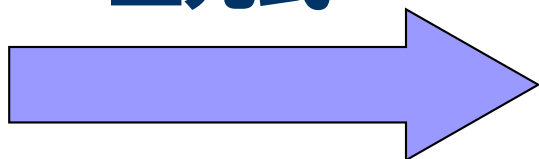
$T_5 := a+b$

$T_6 := T_5 + c$

$T_7 := T_4 - T_6$

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

三元式



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

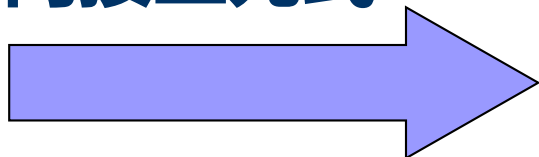
(4)

(1)

(5)

(6)

间接三元式



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 → E1+E2 E.place:=newtemp;
               E.code:=E1.code || E2.code || gen(E.place ':=' E1.place '+' E2.place)
E → E1*E2 E.place:=newtemp;
               E.code:=E1.code || E2.code || gen(E.place ':=' E1.place '*' E2.place)
```

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

```

```
E → E1+E2  { E.place:=newtemp;
                emit(E.place ':=' E1.place '+' E2.place)}

```

```
E → E1*E2  { E.place:=newtemp;
                emit(E.place ':=' E1.place '*' E2.place)}

```

$E \rightarrow -E_1$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel \text{gen}(E.place \text{ ' := ' } \text{ 'uminus' } E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place;$ $E.code = \text{' '}$

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

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

$E \rightarrow id$ { $p := \text{lookup}(id.name);$
 if $p \neq \text{nil}$ 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 $E \rightarrow E_1 \text{ op } E_2$ 的语义动作中关于E.type的语义规则可定义为：
 { if $E_1.\text{type}=\text{integer}$ and $E_2.\text{type}=\text{integer}$
 E.type:=integer
 else E.type:=real }
- 进行类型转换的三地址代码
 $x := \text{inttoreal } y$
- 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：

```
T1 := i int* j
T3 := inttoreal T1
T2 := y real+ T3
x := T2
```

关于产生式 $E \rightarrow E_1 + E_2$ 的语义动作

```
{ E.place:=newtemp;  
  if E1.type=integer and  
    E2.type=integer then begin  
    emit (E.place ':=' E  
      1.place 'int+' E2.place);  
    E.type:=integer  
  end  
  else if E1.type=real and  
    E2.type=real then begin  
    emit (E.place ':=' E  
      1.place 'real+' E2.place);  
    E.type:=real  
  end  
end
```

```
  else if E1.type=integer and E2.type=real  
    then begin  
      u:=newtemp;  
      emit (u ':=' 'inttoreal' E  
        1.place);  
      emit (E.place ':=' u 'real+' E  
        2.place);  
      E.type:=real  
    end  
  else if E1.type=real and E2.type=integer  
    then begin  
      u:=newtemp;  
      emit (u ':=' 'inttoreal' E  
        2.place);  
      emit (E.place ':=' E1.place  
        'real+' u);  
      E.type:=real  
    end  
  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。

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

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

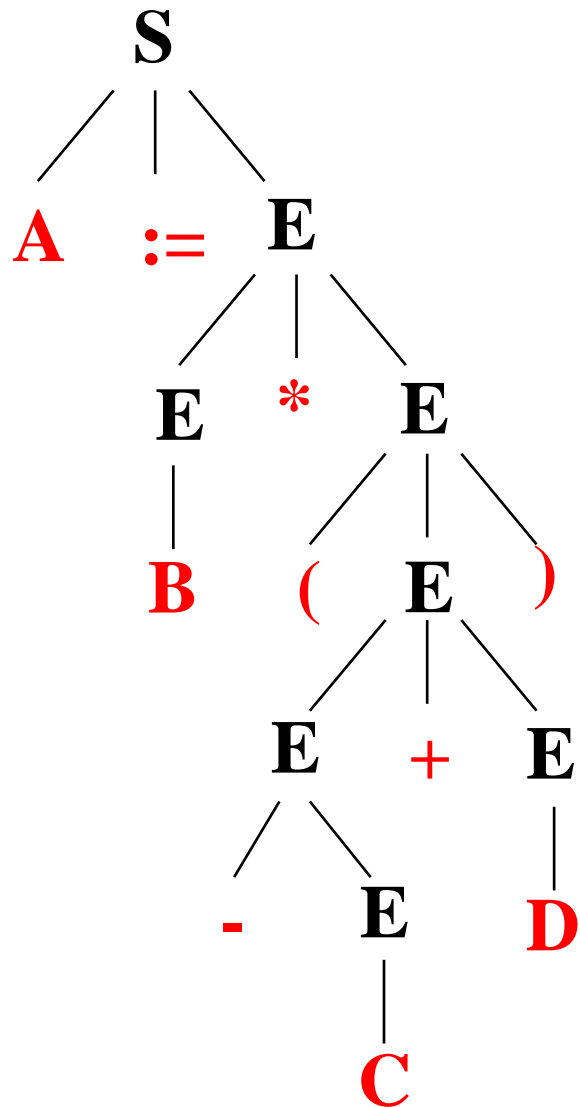
$E \rightarrow E_1 * E_2$ { $E.place := \text{newtemp};$
 emit($E.place := E_1.place * E_2.place$)}

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

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

$E \rightarrow id$ { $p := \text{lookup}(id.name);$
 if $p \neq \text{nil}$ 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：**
 - $E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \neg E \mid (E) \mid \text{id rop id} \mid \text{id}$
- **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

$T_1 := \text{not } c$

$T_2 := b \text{ and } T_1$

$T_3 := a \text{ or } T_2$

■ $a < b$ 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

$E \rightarrow E_1 \text{ or } E_2$	$\{E.place := \text{newtemp};$ $\text{emit}(E.place \text{ ' := ' } E_1.place \text{ ' or ' }$ $E_2.place)\}$
$E \rightarrow E_1 \text{ and } E_2$	$\{E.place := \text{newtemp};$ $\text{emit}(E.place \text{ ' := ' } E_1.place \text{ ' and ' }$ $E_2.place)\}$
$E \rightarrow \text{not } E_1$	$\{E.place := \text{newtemp};$ $\text{emit}(E.place \text{ ' := ' ' not ' } E_1.place)\}$
$E \rightarrow (E_1)$	$\{E.place := E_1.place\}$

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→id1 relop id2 { E.place:=newtemp;
                      emit( 'if' id1.place relop. op
                           id2. place 'goto' nextstat+3);
                      emit(E.place ':=' '0' );
                      emit( 'goto' nextstat+2);
                      emit(E.place ':=' '1' ) }
```

```
E→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:  T1:=0
102:  goto 104
103:  T1:=1
104:  if c<d goto 107
105:  T2:=0
106:  goto 108
107:  T2:=1
108:  if e<f goto 111
109:  T3:=0
110:  goto 112
111:  T3:=1
112:  T4:=T2 and T3
113:  T5:=T1 or T4
```

```
E → id1 relop id2
{ E.place:=newtemp;
  emit( 'if' id1.place relop. op id2.place
        'goto' nextstat+3);
  emit(E.place ':=' '0' );
  emit( 'goto' nextstat+2);
  emit(E.place ':=' '1') }
```

```
E → id
{ E.place:=id.place }
```

```
E → E1 or E2
{ E.place:=newtemp;
  emit(E.place ':=' E1.place 'or'
        E2.place) }
```

```
E → E1 and E2
{ E.place:=newtemp;
  emit(E.place ':=' E1.place 'and'
        E2.place) }
```

随堂练习

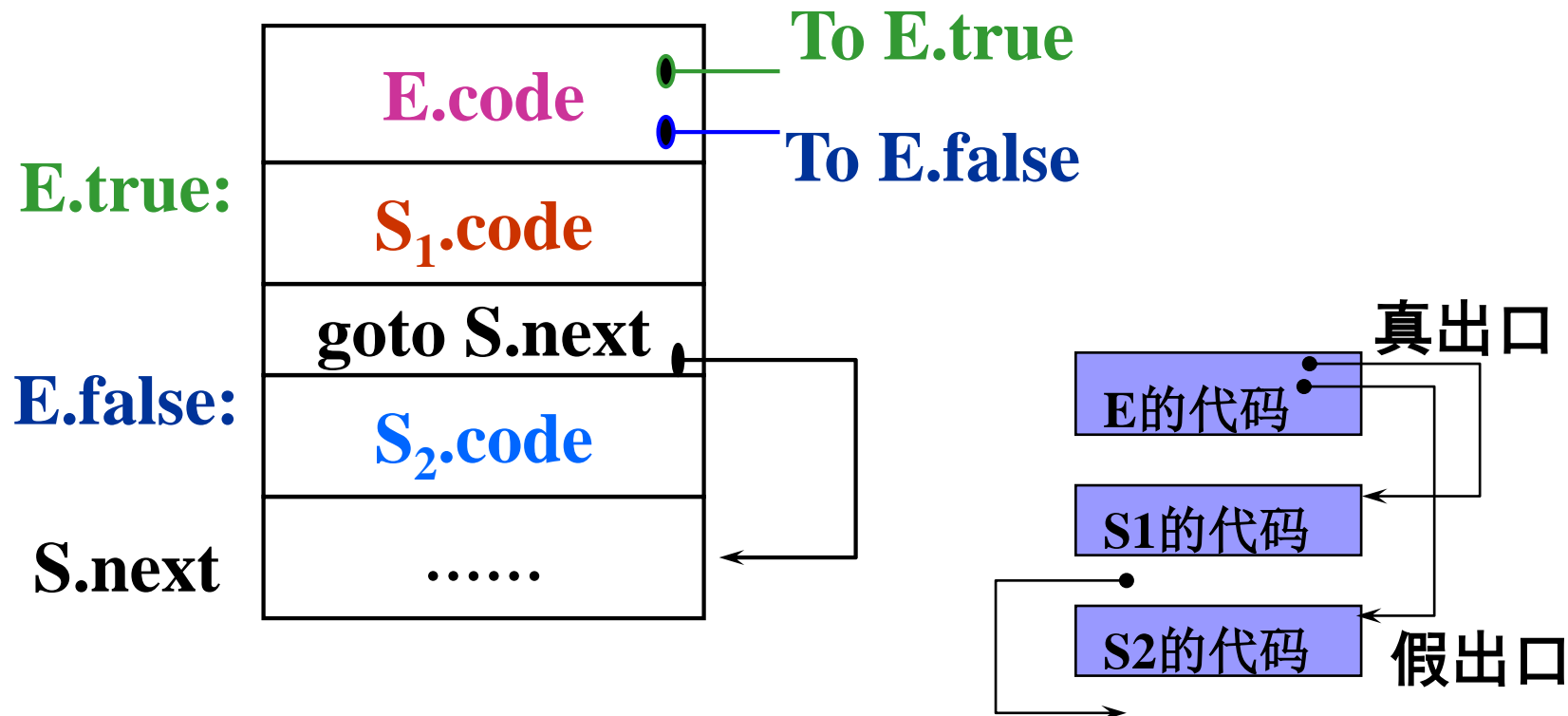
■ 求布尔表达式的翻译结果

□ $a > b$ and $c > d$

作为条件控制的布尔式翻译

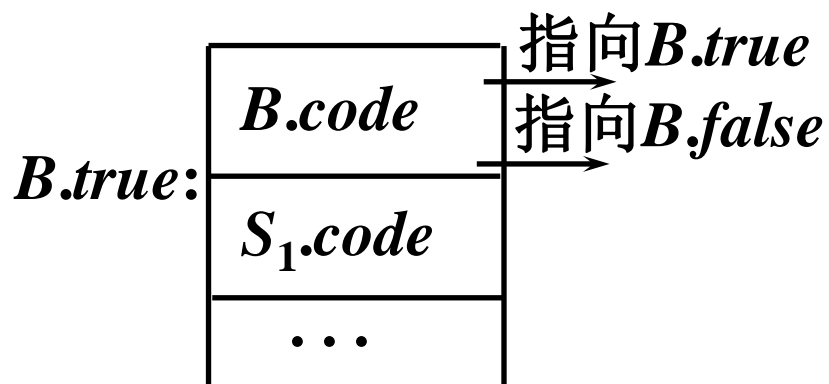
■ 条件语句 if E then S_1 else S_2

赋予 E 两种出口:一真一假

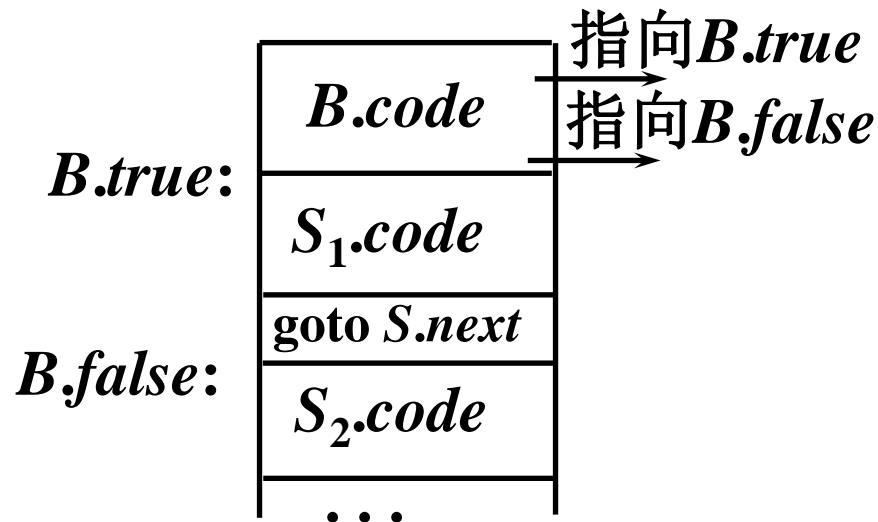


布尔表达式和控制流语句

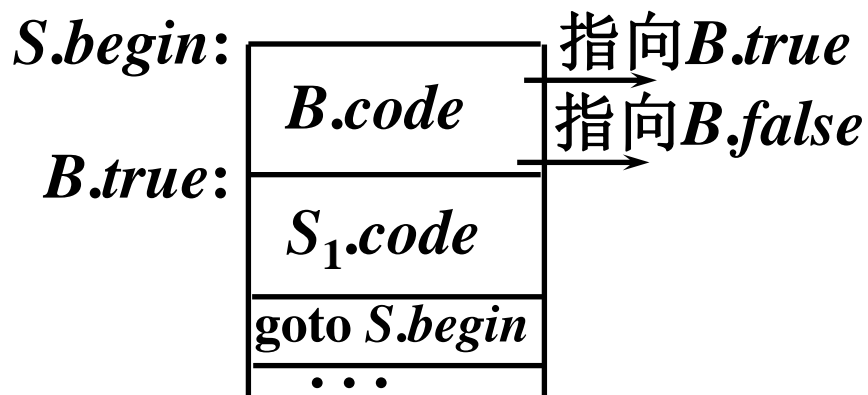
控制流语句的翻译



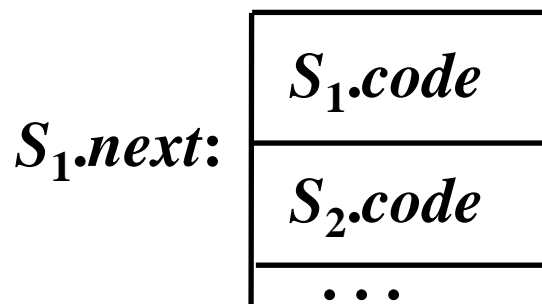
(a) if-then



(b) if-then-else



(c) while-do

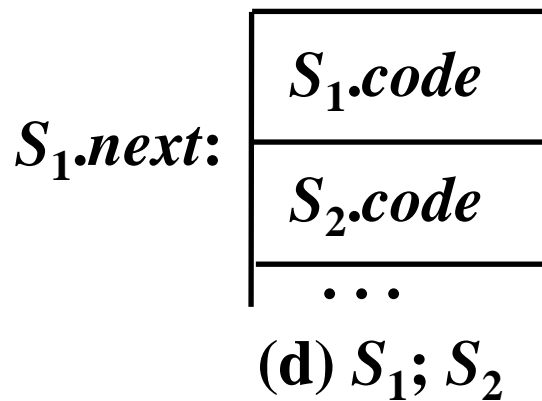
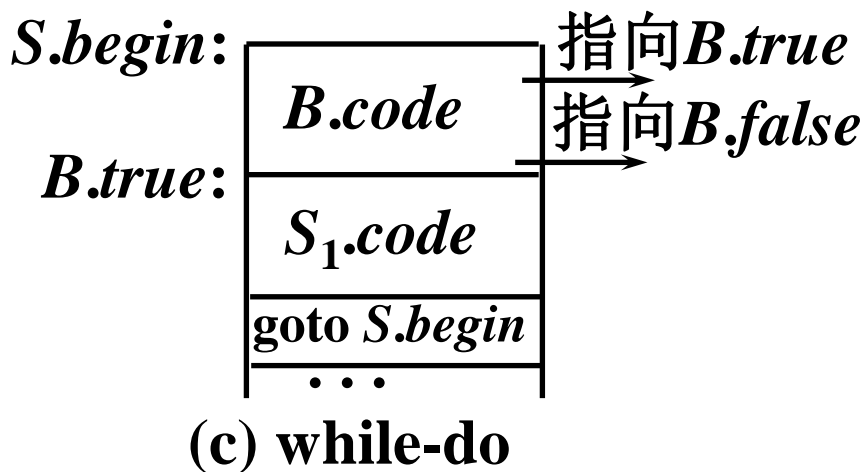
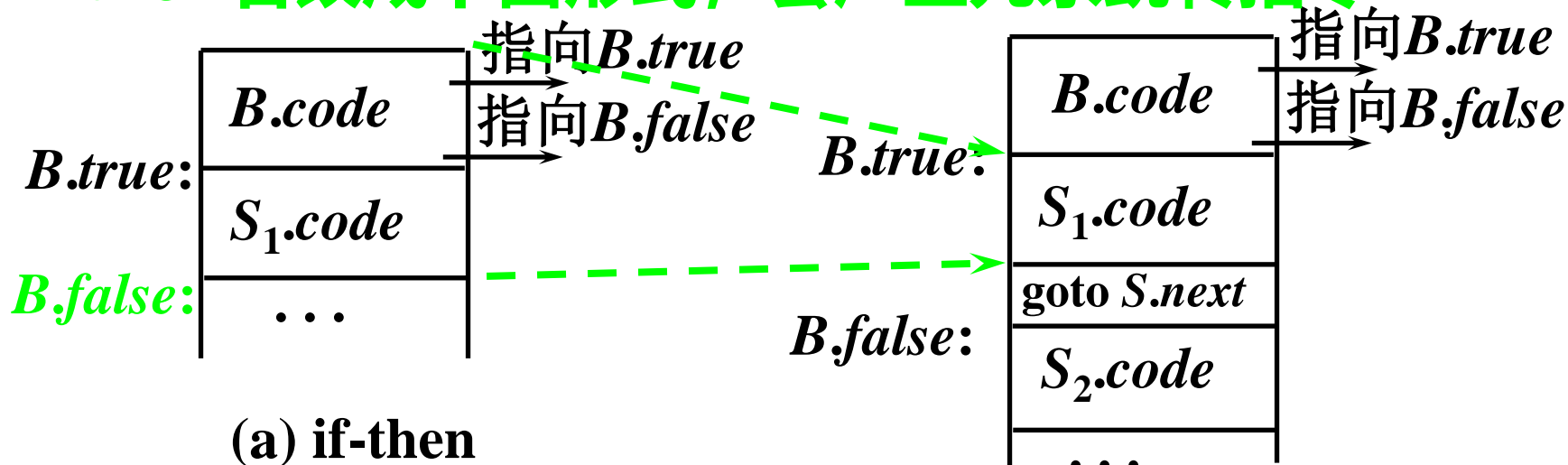


(d) *S₁; S₂*

布尔表达式和控制流语句

控制流语句的翻译

If-then若改成下面形式, 会产生冗余跳转指令



布尔表达式的翻译

■ 两遍扫描

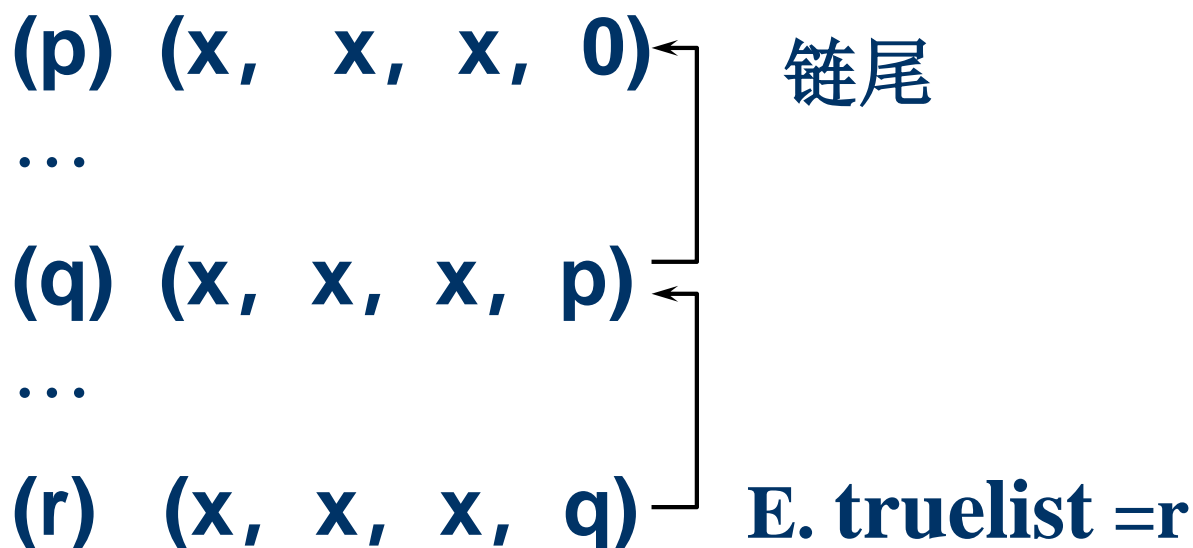
- 为给定的输入串构造一棵语法树；
- 对语法树进行深度优先遍历，进行语义规则中规定的翻译。

■ 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:



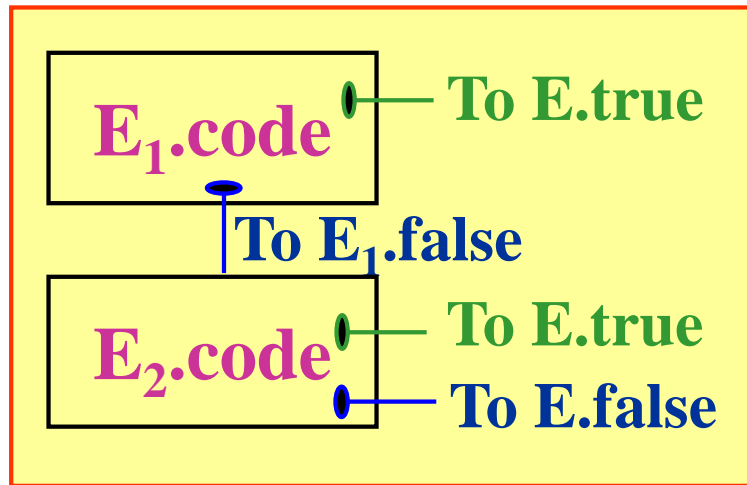
■ 为了处理E.truelist和E.falselist，引入下列语义变量和过程：

- 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

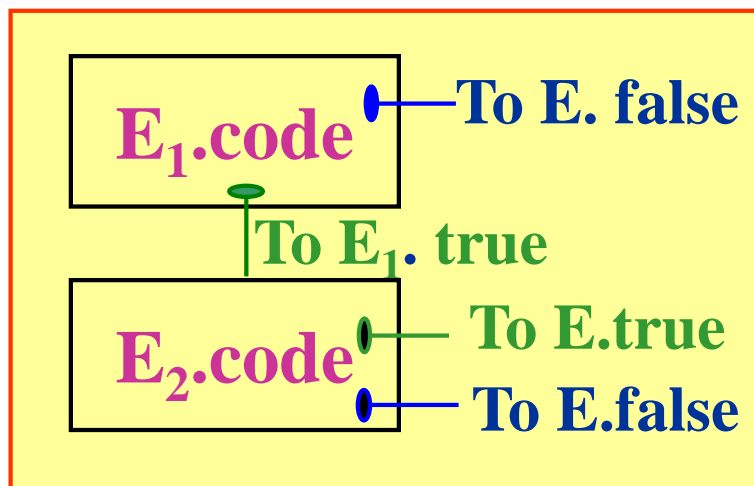
- (1) $E \rightarrow E_1 \text{ or } M E_2$
- (2) $\quad \quad \quad | E_1 \text{ and } M E_2$
- (3) $\quad \quad \quad | \text{ not } E_1$
- (4) $\quad \quad \quad | (E_1)$
- (5) $\quad \quad \quad | \text{id}_1 \text{ relop id}_2$
- (6) $\quad \quad \quad | \text{id}$
- (7) $M \rightarrow \varepsilon$

布尔表达式的翻译模式



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

```
{ 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) $E \rightarrow \text{not } E_1$

{ $E.\text{truelist} := E_1.\text{falselist};$
 $E.\text{falselist} := E_1.\text{truelist}$ }

(4) $E \rightarrow (E_1)$

{ $E.\text{truelist} := E_1.\text{truelist};$
 $E.\text{falselist} := E_1.\text{falselist}$ }

布尔表达式的翻译模式

(5) $E \rightarrow id_1 \text{ relop } id_2$

```
{ E.truelist:=makelist(nextquad);  
  E.falselist:=makelist(nextquad+1);  
  emit( 'j', relop.op, id1.place, id2.place, 0 );  
  emit( 'j, -, -, 0' ) }
```

(6) $E \rightarrow id$

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

(7) $M \rightarrow \varepsilon$

```
{ M.quad:=nextquad }
```

随堂练习

- 写出布尔式 $A \text{ or } B$ 的四元式序列
- 写出布尔式 $A \text{ or } (B \text{ and not } (C \text{ or } D))$ 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。

100 (jnz, A, -, 0)

101 (j, -, -, ~~0~~02)

102 (jnz, B, -, ~~0~~04)

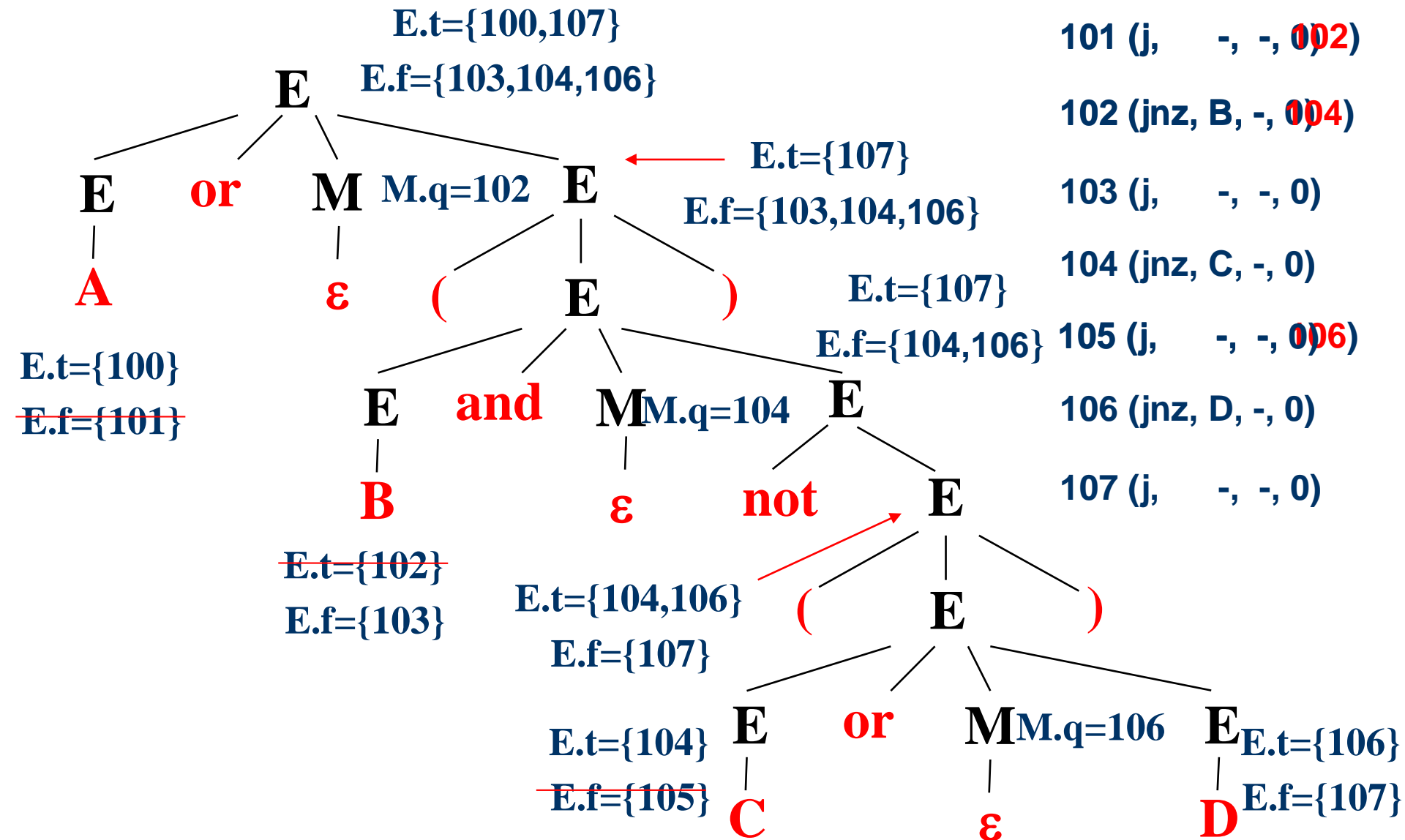
103 (j, -, -, 0)

104 (jnz, C, -, 0)

105 (j, -, -, ~~0~~06)

106 (jnz, D, -, 0)

107 (j, -, -, 0)



$a < b$ or $c < d$ and $e < f$

100 (j<, a, b, 0)

101 (j, -, -, 102)

102 (j<, c, d, 104)

103 (j, -, -, 0)

104 (j<, e, f, 100) truelist

105 (j, -, -, 103) falselist 5

作为整个布尔表达式的"
真""假"出口(转移目标)
仍待回填

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) $S \rightarrow \text{if } E \text{ then } S$

(2) $\quad \quad | \text{if } E \text{ then } S \text{ else } S$

(3) $\quad \quad | \text{while } E \text{ do } S$

(4) $\quad \quad | \text{begin } L \text{ end}$

(5) $\quad \quad | A$

(6) $L \rightarrow L; S$

(7) $\quad \quad | S$

- 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

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

改写后产生式

$$S \rightarrow \text{if } E \text{ then } M S_1 \\ S \rightarrow \text{if } E \text{ then } M_1 S_1 N \text{ else } M_2 S_2 \\ M \rightarrow \varepsilon \\ N \rightarrow \varepsilon$$

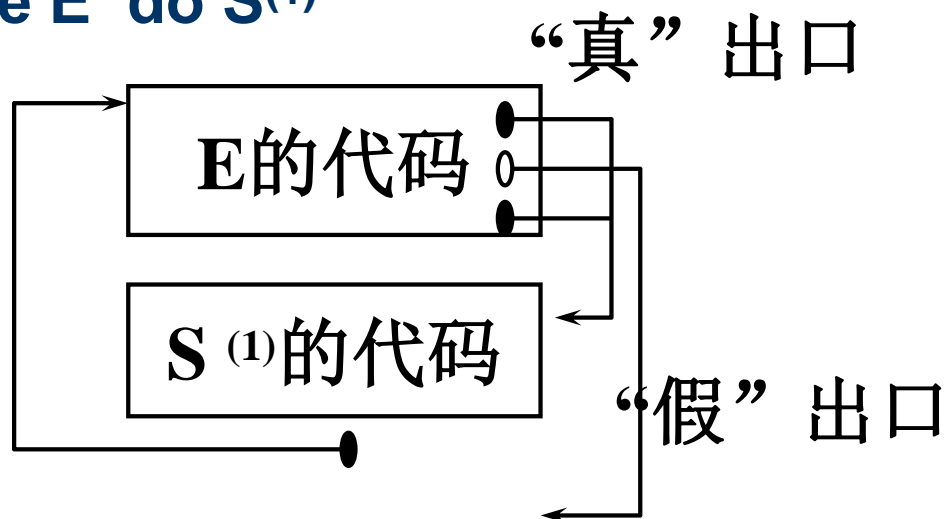
翻译模式:

1. $S \rightarrow \text{if } E \text{ then } M \ S_1$
{ backpatch(E.truelist, M.quad);
S.nextlist:=merge(E.falselist, S₁.nextlist) }
2. $S \rightarrow \text{if } E \text{ then } M_1 \ S_1 \ N \ \text{else } M_2 \ S_2$
{ backpatch(E.truelist, M₁.quad);
backpatch(E.falselist, M₂.quad);
S.nextlist:=merge(S₁.nextlist, N.nextlist, S₂.nextlist) }
3. $M \rightarrow \varepsilon$ { M.quad:=nextquad }
4. $N \rightarrow \varepsilon$ { N.nextlist:=makelist(nextquad);
emit('j, - , - ,0') }

while 语句的翻译

相关产生式

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



为了便于“回填”，改写产生式为：

$S \rightarrow \text{while } M_1 \ E \ \text{do } M_2 \ S_1$

$M \rightarrow \epsilon$

翻译模式:

1. $S \rightarrow \text{while } M_1 \text{ E do } M_2 S_1$
 {backpatch($S_1.\text{nextlist}$, $M_1.\text{quad}$);
 backpatch($E.\text{truelist}$, $M_2.\text{quad}$);
 $S.\text{nextlist} := E.\text{falselist}$
 emit('j, - , - ,' $M_1.\text{quad}$) }
2. $M \rightarrow \varepsilon$ { $M.\text{quad} := \text{nextquad}$ }

语句 $L \rightarrow L; S$ 的翻译

产生式

$$L \rightarrow L; S$$

改写为:

$$L \rightarrow L_1; M S$$

$$M \rightarrow \varepsilon$$

翻译模式:

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

其它几个语句的翻译

$S \rightarrow \text{begin } L \text{ end}$
 { $S.\text{nextlist} := L.\text{nextlist}$ }

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

$L \rightarrow S$
 { $L.\text{nextlist} := S.\text{nextlist}$ }

翻译语句

**while (a<b) do
if (c<d) then x:=y+z;**

P195

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

$\text{nextquad});$

$\text{id}_2.\text{place } ', '0');$

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

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

$S \rightarrow \text{id}_1 = E \ \{ p := \text{lookup}(\text{id}_1.\text{name});$
 $\text{if } p \neq \text{nil} \text{ then}$
 $\text{emit}(p, ':=$
 $\text{else error} \}$

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

P195

$S \rightarrow \text{while } M_1 \ E \ \text{do } M_2 \ S_1$
 $\{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{quad});$
 $\text{backpatch}(E.\text{truelist}, M_2.\text{quad});$
 $S.\text{nextlist} := E.\text{falselist}$
 $\text{emit}('j, - , - , ' M_1.\text{quad}) \}$

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

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

随堂作业

■ P218, 7

While $A < C$ and $B < D$ do

 If $A = 1$ then $C := C + 1$ else

 while $A \leq D$ 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

(p) (j, -, -, 0)

...

(q) (j, -, -, p)

...

(r) (j, -, -, q)

产生式 $S' \rightarrow \text{goto } L$ 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 $\text{GEN}(J, -, -, P)$ # 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;

$\text{GEN}(J, -, -, 0)$

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;

$\text{GEN}(J, -, -, q)$

END

}

■ 带标号语句的产生式:

$S_L \rightarrow \text{label } S$ $\text{label} \rightarrow i:$

■ $\text{label} \rightarrow i:$ 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

向后转移:

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

向前转移:

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

CASE语句的翻译

■ 语句结构

```
case E of
  C1: S1;
  C2: S2;
  ...
  Cn-1: Sn-1;
  otherwise: Sn
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
```

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:

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

(2) $\text{Elist} \rightarrow \text{Elist}, E$

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

■ 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

3. $Elist \rightarrow E$

{ 初始化queue仅包含E.place }

2. $Elist \rightarrow Elist, E$

{ 将E.place加入到queue的队尾 }

1. $S \rightarrow \text{call id} (Elist)$

{ for 队列queue中的每一项p do
 emit('param' p);
emit('call' id.place) }

课堂作业

- **CALL S(A, X+Y)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

धन्यवाद

Hindi

תודה רבה

Hebrew

Tack så mycket

Swedish

Obrigado

Brazilian
Portuguese

Dankon

Esperanto

Thank You !

谢谢

Chinese

ありがとうございます

Japanese

Trugarez

Breton

Danke

German

Tak

Danish

Grazie

Italian

நன்றி

Tamil

děkuji

Czech

ขอบคุณ

Thai

go raibh maith agat

Gaelic