



第8章

静态语义分析和 中间代码生成

8.1 符号表

■ 符号表的作用

- 收集符号属性
- 上下文语义的合法性检查的依据
- 作为目标代码生成阶段地址分配的依据

■ 符号的常见属性

- ❑ 符号的名字
- ❑ 符号的类别
- ❑ 符号的类型
- ❑ 符号的存储类别
- ❑ 符号变量的存储分配信息
- ❑ 符号的作用域及可视性
- ❑ 符号的其它属性
 - 数组内情向量
 - 记录结构型的成员信息
 - 函数及过程的形参

```
int k;  
func(int a, int b)  
{static int m, i=2;  
    i+=m+1;  
    m=i+a+b;  
    return m;  
}  
main()  
{int k=4, m=1, p;  
    p=func(k, m);  
    printf("%d, ", p);  
    p=func(k, m);  
    printf("%d\n", p);  
}
```

■ 符号表的实现

□ 针对符号表的常见操作

- 创建符号表
- 插入表项
- 查询表项
- 修改表项
- 删除表项
- 释放符号表空间

8.2 静态语义分析

■ 静态语义分析的主要任务

□ 控制流检查

控制流语句必须使控制转移到合法的地方

□ 唯一性检查

很多场合要求对象只能被定义一次

□ 名字的上下文相关性检查

某些名字的多次出现之间应该满足一定的上下文相关性

□ 类型检查

检查每个操作是否遵守语言类型系统的定义

8.2.2.2语法制导的类型检查

表达式文法 $E \rightarrow T + T \mid T \text{ or } T$
 $T \rightarrow n \mid b$

```
E → T1 + T2    { if (T1.type==int && T2.type==int)
                      E.type=int ;
                      else error ; }
E → T1 or T2 {if (T1.type==bool && T2.type==bool)
                  E.type =bool ;
                  else error ; }
T → n            { T.type= int; }
T → b            { T.type= bool; }
```

n + n#

4	n	--
o	#	--

2	T	int
o	#	--

5 + --

2	T	int
o	#	--

4	n	---
5	+	---
2	T	int
o	#	--

6	T	int
5	+	---
2	T	int
o	#	--

1	E	int
o	#	--

$E \rightarrow T1 + T2$ { if ($T1.type == int$ &&
 $T2.type == int$)
 $E.type = int$;
 else error ; }

$E \rightarrow T1 \text{ or } T2$ { if ($T1.type == bool$ &&
 $T2.type == bool$)
 $E.type = bool$;
 else error ; }

$T \rightarrow n$ { $T.type = int$; }
 $T \rightarrow b$ { $T.type = bool$; }

LR(0)分析表

状态	action					GOTO	
	+	o	n	b	#	E	T
0			S4	S3		1	2
1					acc		
2	s5	s7					
3	r4	r4	r4	r4	r4		
4	r3	r3	r3	r3	r3		
5			s4	s3			6
6	r1	r1	r1	r1	r1		
7			s4	s3			8
8	r2	r2	r2	r2	r2		

n +b

4	n	--
o	#	--

2	T	int
o	#	--

5	+	---
2	T	int
o	#	--

3	b	---c
5	+	---
2	T	int
o	#	--

6	T	bool
5	+	---
2	T	int
o	#	--

1	E	error
o	#	--

$E \rightarrow T1 + T2$ { if (T1.type==int &&
T2.type==int)

E.type=int ;

else error ;}

$E \rightarrow T1 \text{ or } T2$ { if (T1.type==bool &&
T2.type==bool)

E.type=bool ;

else error ; }

$T \rightarrow n$ { T.type=int ; }

$T \rightarrow b$ { T.type=bool ; }

LR(0)分析表

状态	action					GOTO	
	+	o	n	b	#	E	T
0			S4	S3		1	2
1					acc		
2	s5	s7					
3	r4	r4	r4	r4	r4		
4	r3	r3	r3	r3	r3		
5			s4	s3			6
6	r1	r1	r1	r1	r1		
7			s4	s3			8
8	r2	r2	r2	r2	r2		

8.3 中间代码生成

■ 中间代码

□ 源程序的不同表示形式

□ 作用

- 源语言和目标语言之间的桥梁，避开二者之间较大的语义跨度，使编译程序的逻辑结构更加简单明确。
- 利于编译程序的重定向
- 利于进行与目标机无关的优化

8.3.1常见的中间表示形式

- **AST** (**Abstract syntax tree**, 抽象语法树)
- **TAC** (**Three-address code**, 三地址码, 四元式)
- **P-code** (特别用于 **Pascal** 语言实现)
- **Bytecode** (**Java** 编译器的输出, **Java** 虚拟机的输入)
- **SSA** (**Static single assignment form**, 静态单赋值形式)

中间代码生成-TAC

- 赋值语句 $x := y \text{ op } z$ (op 代表二元算术/逻辑运算)
- 赋值语句 $x := \text{op } y$ (op 代表一元运算)
- 复写语句 $x := y$ (y 的值赋值给 x)
- 无条件跳转语句 $\text{goto } L$ (无条件跳转至标号 L)
- 条件跳转语句 $\text{if } x \text{ rop } y \text{ goto } L$ (rop 代表关系运算)
- 标号语句 $L:$ (定义标号 L)
- 过程调用语句序列 $\text{param } x_1 \dots \text{param } x_n \text{ call } p, n$
- 过程返回语句 $\text{return } y$ (y 可选, 存放返回值)
- 下标赋值语句 $x := y[i]$ 和 $x[i] := y$ (前者表示将地址 y 起第 i 个存储单元的值赋值给 x , 后者类似)
- 指针赋值语句 $x := *y$ 和 $*x := y$

■ 四元式（三地址码）

格式1: (op,arg1,arg2,result)

格式2: result:= arg1 op arg2

例: a:=b*c+b*d

1) (*,b,c,t1)

2) (*,b,d,t2)

3) (+,t1,t2,t3)

4) (:=,t3,-,a)

1) t1:=b*c

2) t2:=b*d

3) t3:=t1+t2

4) a:=t3

(jump,-,-,L)

(jrop,B,C,L)

goto L

if B rop C goto L

(1) jump - - (3)

(2) j< b c (5)

(1) Goto (3)

(2) if b<c goto(5)

赋值语句及算术表达式的三地址码

$X := A + B * (C + D)$

(1) $T1 := C + D$

(2) $T2 := B * T1$

(3) $T3 := A + T2$

(4) $X := T3$

赋值语句及算数表达式的语法制导翻译

— 语义属性

$\underline{id}.place$: \underline{id} 对应的存储位置

$E.place$: 用来存放 E 的值的存储位置

$E.code$: E 求值的 TAC 语句序列

$S.code$: 对应于 S 的 TAC 语句序列

— 语义函数/过程

gen : 生成一条 TAC 语句

$newtemp$: 在符号表中新建一个从未使用过的名字,
并返回该名字的存储位置

\parallel 是 TAC 语句序列之间的链接运算

赋值语句及算术表达式的语法制导翻译

— 翻译模式

$S \rightarrow \underline{id} := E \quad \{ S.code := E.code \parallel gen(\underline{id}.place := E.place) \}$

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

$E \rightarrow \underline{int} \quad \{ E.place := newtemp; E.code := gen(E.place := \underline{int}.val) \}$

$E \rightarrow \underline{real} \quad \{ E.place := newtemp; E.code := gen(E.place := \underline{real}.val) \}$

$E \rightarrow E_1 + E_2 \quad \{ E.place := newtemp; E.code := E_1.code \parallel E_2.code \parallel \\ gen(E.place := E_1.place '+' E_2.place) \}$

$E \rightarrow E_1 * E_2 \quad \{ E.place := newtemp; E.code := E_1.code \parallel E_2.code \parallel \\ gen(E.place := E_1.place '*' E_2.place) \}$

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

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

说明语句的语法制导翻译

— 语义属性

id.name : *id* 的词法名字（符号表中的名字）

T.type : 类型属性 （综合属性）

T.width, *V.width* : 数据宽度（字节数）

L.offset : 列表中第一个变量的偏移地址

L.type : 变量列表被声明的类型 （继承属性）

L.num : 变量列表中变量的个数

— 语义函数/过程

enter (*id.name*, *t*, *o*) : 将符号表中 *id.name* 所对应表项的 *type* 域置为 *t*, *offset* 域置为 *o*

说明语句的语法制导翻译

— 翻译模式

$$\begin{aligned} V \rightarrow V_1 ; T & \quad \{ L.type := T.type; L.offset := V_1.width ; L.width := T.width \} \\ L & \quad \{ V.type := \text{make_product_3}(V_1.type, T.type, L.num); \\ & \quad V.width := V_1.width + L.num \times T.width \} \\ V \rightarrow \varepsilon & \quad \{ V.type := <>; V.width := 0 \} \\ T \rightarrow \text{boolean} & \quad \{ T.type := \text{bool} ; T.width := 1 \} \\ T \rightarrow \text{integer} & \quad \{ T.type := \text{int} ; T.width := 4 \} \\ T \rightarrow \text{real} & \quad \{ T.type := \text{real} ; T.width := 8 \} \end{aligned}$$

数组说明和数组元素引用的语法制导翻译

— 数组说明

参考前页的翻译模式，可了解（一维）数组说明的翻译思想。至于符号表中一般情况下是如何组织数组说明信息的，随后将会讨论。

.....

$$T \rightarrow \text{array } [\underline{\text{num}}] \text{ of } T_1 \quad \{ T.type := \text{array}(1.. \underline{\text{num}}.lexval, T_1.type); \\ T.width := \underline{\text{num}}.val \times T_1.width \}$$

.....

$$S \rightarrow E_1[E_2] := E_3 \quad \{ S.code := E_2.code \parallel E_3.code \parallel \\ \text{gen } (E_1.place '[' E_2.place ']' ':=' E_3.place) \}$$
$$E \rightarrow E_1[E_2] \quad \{ E.place := \text{newtemp}; \\ E.code := E_2.code \parallel \\ \text{gen } (E.place ':=' E_1.place '[' E_2.place ']') \}$$

布尔表达式的语法制导翻译

– 直接对布尔表达式求值

nextstat 返回输出代码序列
中下一条 TAC 语句的下标

$E \rightarrow E_1 \vee E_2$	$\{ E.place := newtemp; E.code := E_1.code \parallel E_2.code$ $\parallel gen(E.place := E_1.place \text{ 'or' } E_2.place) \}$
$E \rightarrow E_1 \wedge E_2$	$\{ E.place := newtemp; E.code := E_1.code \parallel E_2.code$ $\parallel gen(E.place := E_1.place \text{ 'and' } E_2.place) \}$
$E \rightarrow \neg E_1$	$\{ E.place := newtemp; E.code := E_1.code \parallel$ $gen(E.place := \text{'not' } E_1.place) \}$
$E \rightarrow (E_1)$	$\{ E.place := E_1.place; E.code := E_1.code \}$
$E \rightarrow \underline{id_1} \text{ rop } \underline{id_2}$	$\{ E.place := newtemp; E.code := gen(\text{'if' } \underline{id_1}.place$ $\underline{rop.op} \underline{id_2}.place \text{ 'goto' } nextstat+3) \parallel$ $gen(E.place := \text{'0'}) \parallel gen(\text{'goto' } nextstat+2)$ $\parallel gen(E.place := \text{'1'}) \}$
$E \rightarrow \text{true}$	$\{ E.place := newtemp; E.code := gen(E.place := \text{'1'}) \}$
$E \rightarrow \text{false}$	$\{ E.place := newtemp; E.code := gen(E.place := \text{'0'}) \}$

布尔表达式的三地址码

$a < b \vee c < d \wedge e < f$

直接对布尔表达式求值： 通过控制流体现布尔表达式的语义：

(1) if $a < b$ goto 4

(2) $T1 := 0$

(3) goto 5

(4) $T1 := 1$

(5) if $c < d$ goto 8

(6) $T2 := 0$

(7) goto 9

(8) $T2 := 1$

(9) if $e < f$ goto 12

(10) $T3 := 0$

(11) goto 13

(12) $T3 := 1$

(13) $T4 := T2 \text{ and } T3$

(14) $T5 := T1 \text{ or } T4$

(1) if $a < b$ goto E.true

(2) goto 3

(3) if $c < d$ goto 5

(4) goto E.false

(5) if $e < f$ goto E.true

(6) goto E.false

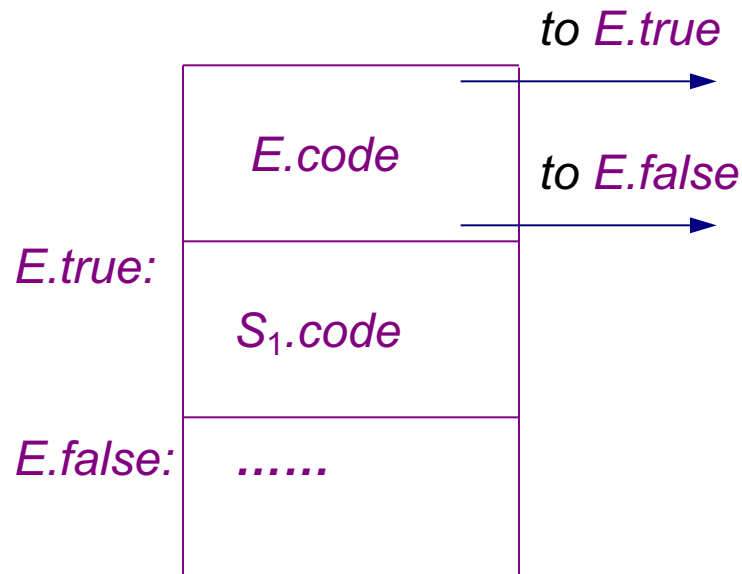
■ 把条件转移的布尔表达式翻译成仅含条件真转和无条件转的四元式

**if $a \text{ rop } b$ goto E.true
goto E.false**

控制语句的三地址码

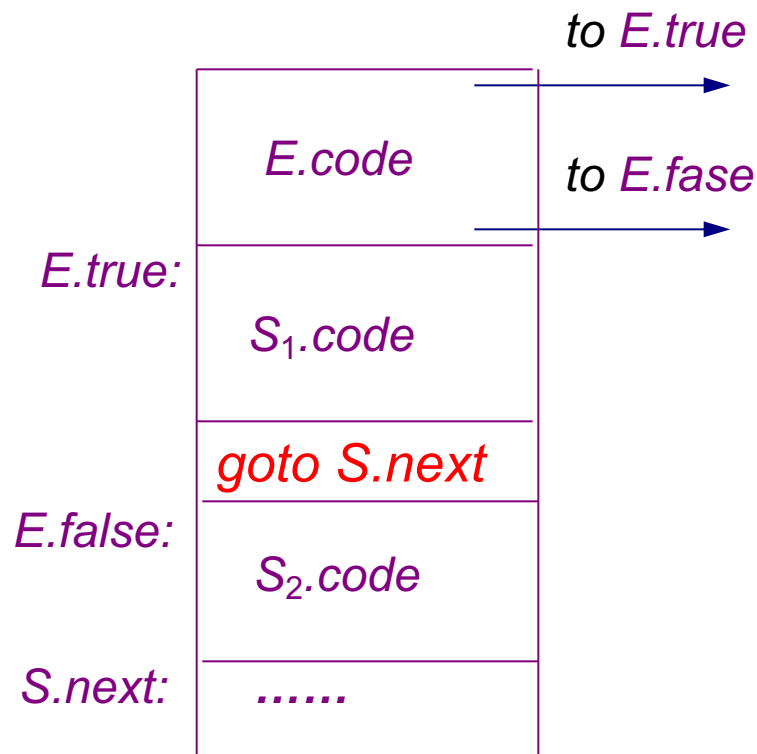
■ if-then 语句

$S \rightarrow \text{if } E \text{ then } S_1$



■ if-then-else 语句

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



■ 例

if **a<b** \vee **c<d** \wedge **e<f** **then**

x:=a

else

y:=b;

(1) if a<b goto 7

(2) go to 3

(3) if c<d goto 5

(4) goto 9

(5) if e<f goto 7

(6) goto 9

(7) x:=a

(8) goto 10

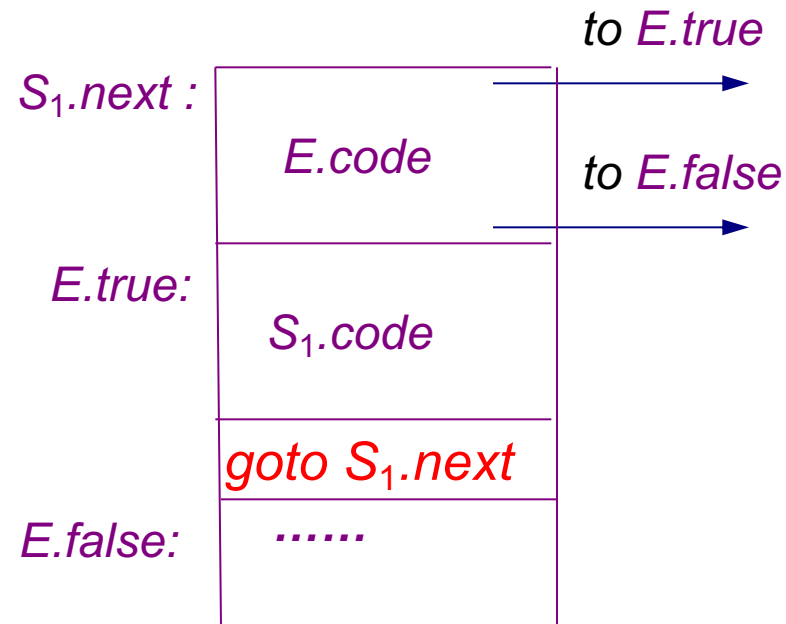
(9) y:=b

(10)

循环语句的三地址码

■ while 语句

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



■ 例

while $a < b \vee c < d \wedge e < f$ **do** $a := c + e$;

(1) if $a < b$ goto **T**
(2) go to 3
(3) if $c < d$ goto 5
(4) goto **F**
(5) if $e < f$ goto **T**
(6) goto **F**
(7) $T1 := c + e$
(8) $a := T1$
(9) **goto 1**
(10)

例:

1 while $A < B$ do

if $C < D$ or $B < D$ then

$X := Y + Z;$

2. if $c < 5$ then

while $x > y$ $z := x + 1;$

else $x := y;$

生成三地址码的S-翻译模式

■ 拉链与回填

□ 语义属性

E.truelist：“真链”，由一系列表示跳转语句的地址组成，这些跳转语句的目标标号是布尔表达式 **E** 为“真”的标号。

E. falselist：“假链”，由一系列表示跳转语句的地址组成，这些跳转语句的目标标号是布尔表达式 **E** 为假的标号。

S. nextlist：“next 链”，由一系列表示跳转语句的地址组成，这些跳转语句的目标标号是在执行序列中紧跟在 **S** 之后的下条**TAC**语句的标号。

□ 语义函数/过程

makelist(i) : 创建只有一个结点 **i** 的表, 对应存放目标**TAC** 语句数组的一个下标。

merge(p1,p2) : 连接两个链表 **p1** 和 **p2** , 返回结果链表。

backpatch(p,i) : 将链表 **p** 中每个元素所指向的跳转语句的标号置为 **i**。

nextstm : 下一条**TAC** 语句的地址

emit (...) : 输出一条**TAC** 语句, 并使 **nextstm** 加1

■ 赋值语句及算术表达式的翻译模式

□ 语义属性

id.place : id 对应的存储位置

E.place : 用来存放 E 的值的存储位置

□ 语义函数/过程

newtemp : 在符号表中新建一个从未使用过的名字，并返回该名字的存储位置。

(1) $S \rightarrow id := E$

(2) $E \rightarrow id$

(3) $E \rightarrow int$

(2) $E \rightarrow E + E$

(3) $E \rightarrow E * E$

(4) $E \rightarrow - E$

(5) $E \rightarrow (E)$

□ 翻译模式

$S \rightarrow id := E \{ \text{emit}(id.place := E.place) \}$

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

$E \rightarrow int \{ E.place := newtemp; \text{emit}(E.place := int.val) \}$

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

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

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

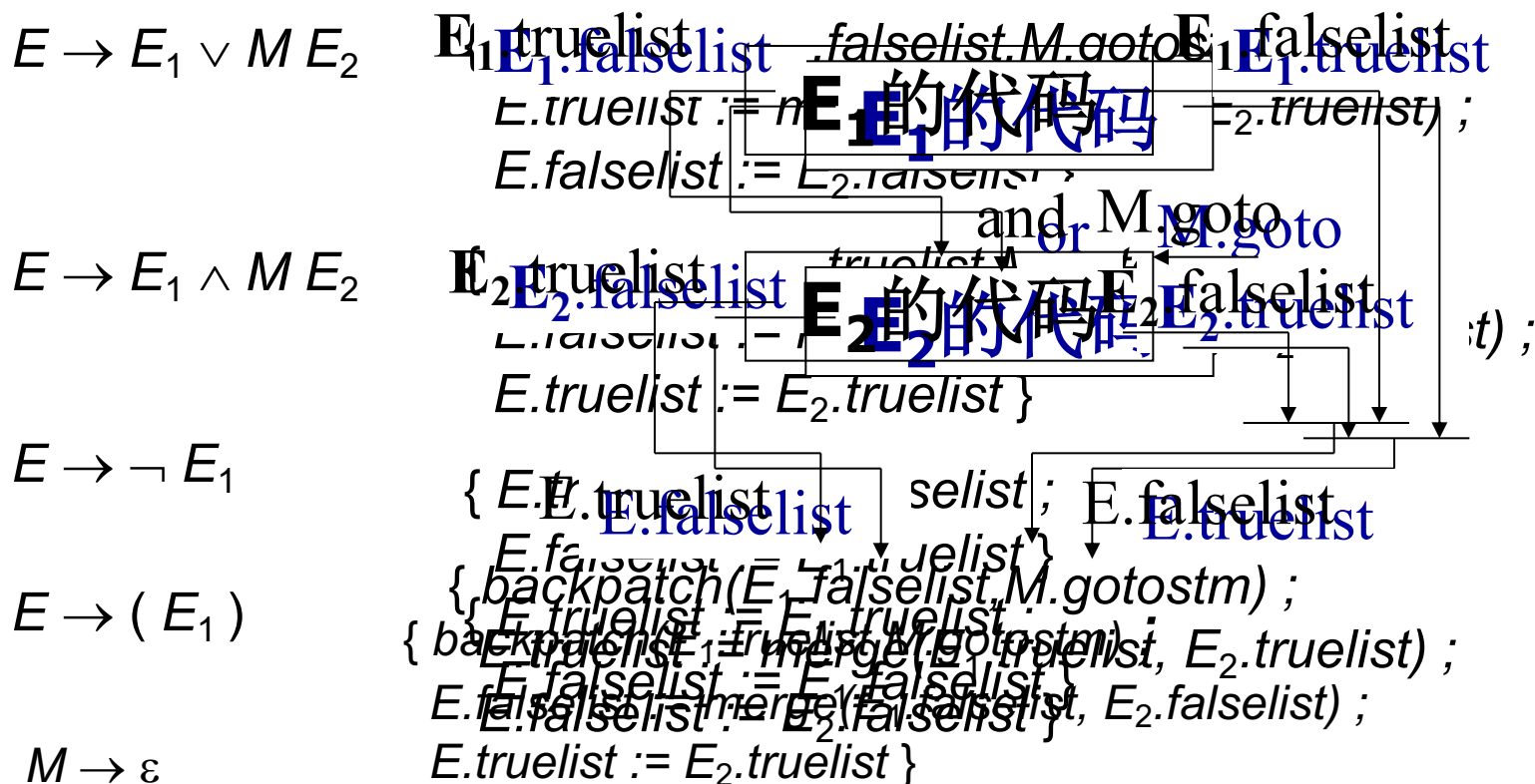
$E \rightarrow (E_1) \{ E.place := E_1.place \}$

■ 布尔表达式的翻译模式

□ 语义属性

M.gotostm: 处理到**M**时下一条待生成语句的标号

□ 翻译模式



$E \rightarrow \underline{id}_1 \text{ rop } \underline{id}_2$	{ $E.truelist := makelist (nextstm);$ $E.falselist := makelist (nextstm+1);$ $emit (\text{'if' } \underline{id}_1.place \text{ rop.op } \underline{id}_2.place \text{ 'goto _' });$ $emit (\text{'goto _' })$ }
$E \rightarrow true$	{ $E.truelist := makelist (nextstm);$ $emit (\text{'goto _' })$ }
$E \rightarrow false$	{ $E.falselist := makelist (nextstm);$ $emit (\text{'goto _' })$ }
$M \rightarrow \varepsilon$	{ $M.gotostm := nextstm$ }

例： $a < b \vee c < d \wedge e < f$ 的注释分析树

■ 条件语句的翻译模式

$S \rightarrow \text{if } E \text{ then } M \ S_1$
 { *backpatch*(*E.true*list, *M.goto*stm) ;
 *S.next*list := *merge*(*E.false*list, *S*₁.*next*list) }

$S \rightarrow \text{if } E \text{ then } M_1 \ S_1 \ N \text{ else } M_2 \ S_2$
 { *backpatch*(*E.true*list, *M*₁.*goto*stm) ;
 backpatch(*E.false*list, *M*₂.*goto*stm) ;
 *S.next*list := *merge*(*S*₁.*next*list,
 merge(*N.next*list, *S*₂.*next*list)) }

$M \rightarrow \epsilon$
 { *M.goto*stm := *next*stm }

$N \rightarrow \epsilon$
 { *N.next*list := *makelist*(*next*stm); *emit*('goto _') }

■ 循环、复合的翻译模式

$$S \rightarrow \text{while } M_1 \text{ E do } M_2 S_1$$
$$\{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{gotostm}) ;$$
$$\text{backpatch}(E.\text{truelist}, M_2.\text{gotostm}) ;$$
$$S.\text{nextlist} := E.\text{falselist};$$
$$\text{emit}(\text{'goto'}, M_1.\text{gotostm}) \}$$
$$S \rightarrow S_1 ; M S_2$$
$$\{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{gotostm}) ;$$
$$S.\text{nextlist} := S_2.\text{nextlist} \}$$
$$M \rightarrow \varepsilon$$
$$\{ M.\text{gotostm} := \text{nextstm} \}$$

练习:

```
1.while a > b and c < d do
    if a > 10 or c < 20 then
        begin s := a + c; a := a - 5; end
    else
        begin s := b + d; b := b + 3; end
```

```
2.while a > 10 do
    if b = 100 then
        while a < 20 do
            a := a + b - 1
```

作业：

```
while A < C and B < 10 do  
  if A == B or true then  
    C := C + A;  
  else  
    D := A + B;
```