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—>T+T| T or T
T—>n | b

$E \rightarrow T^1 + T^2$ { if (T^1.type==int && T^2.type==int)
E.type=int ;
else error ; }

$E \rightarrow T^1$ or T^2 {if (T^1.type==bool && T^2.type==bool)
E.type =bool ;
else error ; }

T → n { T.type= int; }

T → b { T.type= bool;}

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	#	--

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

5	+	--
2	T	int
o	#	--

1	E	int
o	#	--

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

LR(0)分析表

	action					GOTO	
状态	+	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→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 ; }

LR(0)分析表

	action					GOTO	
状态	+	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 op z* （*op* 代表二元算术/逻辑运算）
- 赋值语句 *x := 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 *goto L* （无条件跳转至标号 *L*）
- 条件跳转语句*if x rop y 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 *param* x_1 *… param* x_n *call p,n*
- 过程返回语句 *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)　　goto L

(jrop,B,C,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

  id.*place* : 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

  || 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

$S \rightarrow \underline{\text{id}} := E$ { *S.code* := *E.code* || *gen*(id *.place* ':=' *E.place*) }

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

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

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

$E \rightarrow E_1 + E_2$ { *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*) }

$E \rightarrow -E_1$ { *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* }

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* : 变量列表被申明的类型 （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

$V \rightarrow V_1$; T { $L.type := T.type$; $L.offset := V_1.width$; $L.width := T.width$ }

L { $V.type := make\_product\_3\ (V_1.type, T.type, L.num)$;

$V.width := V_1.width + L.num \times T.width$ }

$V \rightarrow \varepsilon$ { $V.type$:= <>; $V.width := 0$ }

$T \rightarrow$ boolean { $T.type := bool$; $T.width := 1$ }

$T \rightarrow$ integer { $T.type := int$; $T.width := 4$ }

$T \rightarrow$ real { $T.type := real$; $T.width := 8$ }

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.

……

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

……

$S \rightarrow E_1[E_2] := E_3$ { *S.code* := E_2 *.code* || E_3 *.code* ||
gen (E_1 *.place* '[' E_2 *.place* ']' ':=' E_3 *.place*) }

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

布尔表达式的的语法制导翻译

– 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

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

布尔表达式的三地址码

a<b ∨ c<d ∧ e<f

直接对布尔表达式求值：

(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 and T3
(14) T5 := T1 or T4

通过控制流体现布尔表达式的语义：

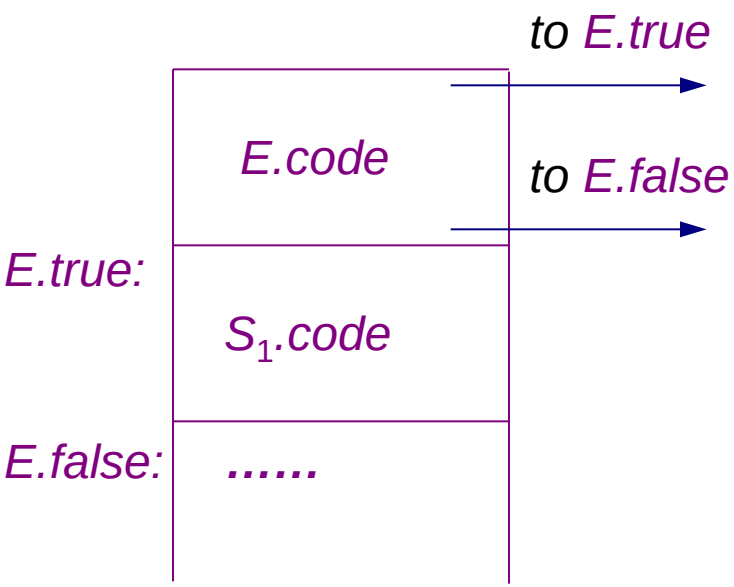
(1) if a<b goto E.true
(2) goto 3
(3) if c<d goto 5
(4) goto E.false
(5) if e<f goto E.true
(6) goto E.false

- 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

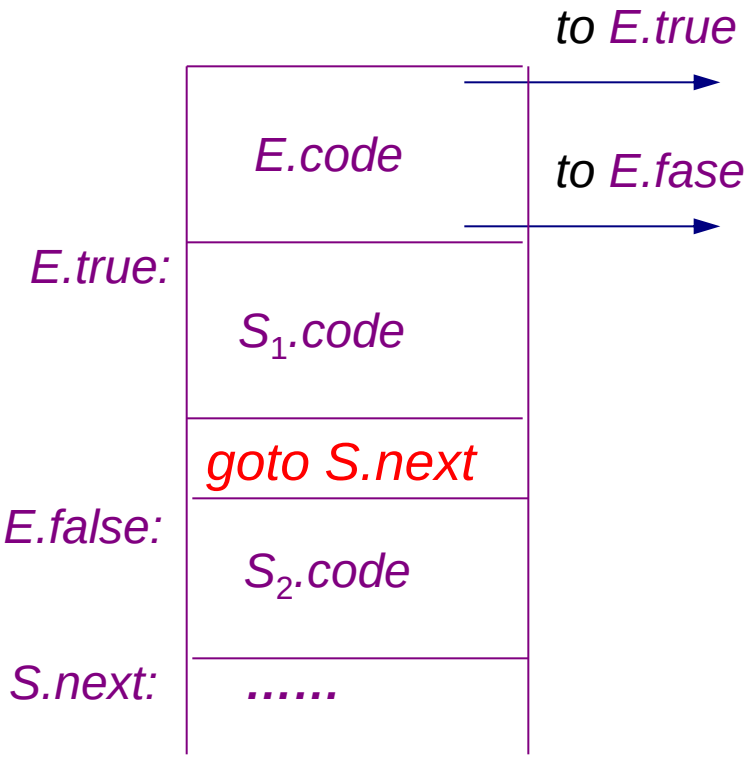
if a rop b goto E.true
goto E.false

控制语句的三地址码

- **if-then 语句**
  $S \rightarrow$ if E then S_1



- **if-then-else 语句**
  $S \rightarrow$ if E then S_1 else S_2



- 例

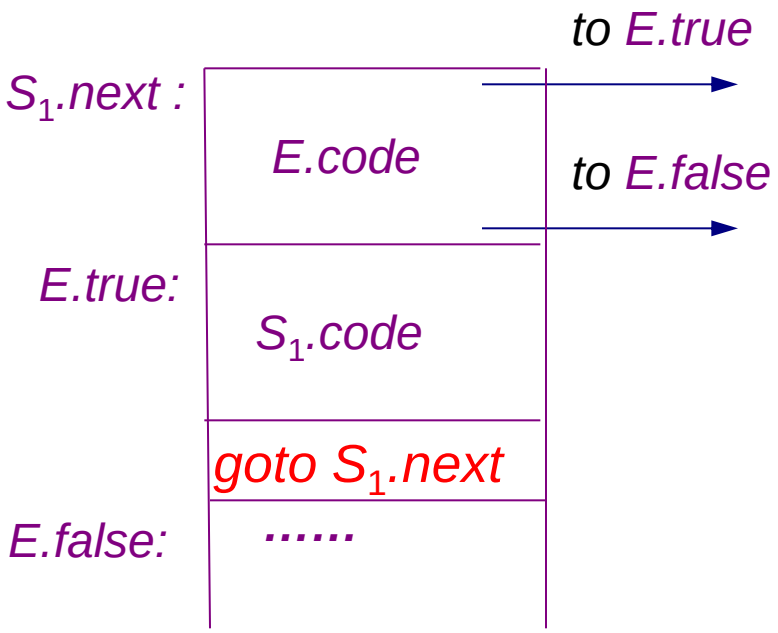
```
if a<b ∨ c<d ∧ e<f then
    x:=a
else
    y:=b;
```

(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**

  $S \rightarrow$ while E do S_1



- 例

  while a<b ∨ c<d ∧ 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 →id := E
(2) E→id
(3) E→int
(2) E→E+E
(3) E→E*E
(4) E→ - E
(5) E→(E)

- 翻译模式

S → id := E { emit (id .place ‘:=’ E.place)}

E → id { E.place := id .place }

E → int { E.place := newtemp; emit (E.place ‘:=’ int .val) }

E → E_1 + E_2 { E.place := newtemp;
emit (E.place ‘:=’ E_1.place ‘+’ E_2.place) }

E → E_1 * E_2 { E.place := newtemp;
emit (E.place ‘:=’ E_1.place ‘*’ E_2.place) }

E → -E_1 { E.place := newtemp;
emit (E.place ‘:=’ ‘uminus’ E_1.place) }

E → (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

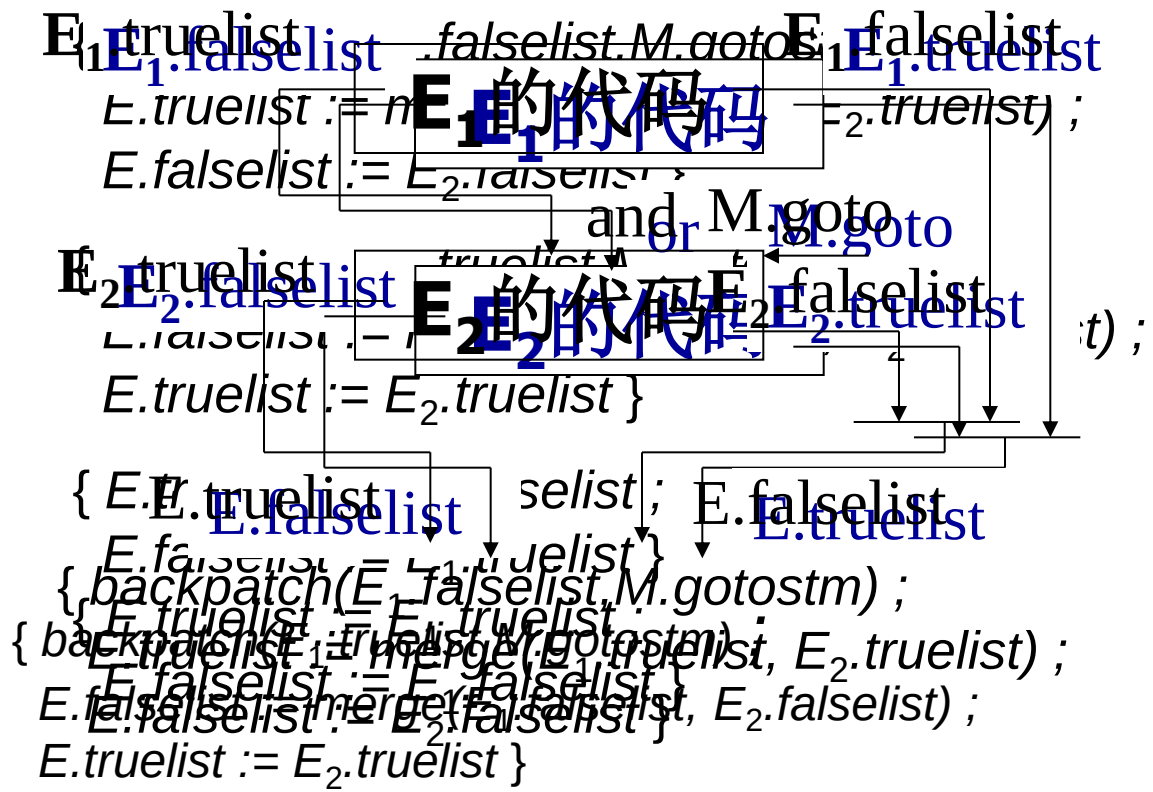
$E \to E_1 \vee M\ E_2$ { *E.truelist := merge(E*$_1$*.truelist, E*$_2$*.truelist) ;*
backpatch(E$_1$*.falselist, M.gotostm) ;*
E.falselist := E$_2$*.falselist* }

$E \to E_1 \wedge M\ E_2$ { *backpatch(E*$_1$*.truelist, M.gotostm) ;*
E.falselist := merge(E$_1$*.falselist, E*$_2$*.falselist) ;*
E.truelist := E$_2$*.truelist* }

$E \to \neg\ E_1$ { *E.truelist := E*$_1$*.falselist ;*
E.falselist := E$_1$*.truelist* }

$E \to (\ E_1\ )$ { *E.truelist := E*$_1$*.truelist ;*
E.falselist := E$_1$*.falselist* }

$M \to \varepsilon$ { *M.gotostm := nextstm* }



$E \rightarrow \underline{id}_1$ rop $\underline{id}_2$ { *E.truelist := makelist (nextstm);*
E.falselist := makelist (nextstm+1);
emit (‘if‘ $\underline{id}_1$*.place* rop.op $\underline{id}_2$*.place* ‘goto _’ *)*;
emit (‘goto _’*)* }

$E \rightarrow$ true { *E.truelist := makelist (nextstm);*
emit (‘goto _’*)* }

$E \rightarrow$ false { *E.falselist := makelist (nextstm);*
emit (‘goto _’*)* }

$M \rightarrow \varepsilon$ { *M.gotostm := nextstm* }

例： a<b ∨ c<d ∧ e<f的注释分析树

条件语句的翻译模式

$S \rightarrow$ if E then $M\ S_1$

{ *backpatch(E.truelist,M.gotostm) ;*
S.nextlist := merge(E.falselist, S_1.nextlist) }

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

{ *backpatch(E.truelist, M_1.gotostm) ;*
backpatch(E.falselist, M_2.gotostm) ;
S.nextlist := merge(S_1.nextlist,
merge(N.nextlist, S_2.nextlist)) }

$M \rightarrow \varepsilon$

{ *M.gotostm := nextstm* }

$N \rightarrow \varepsilon$

{ *N.nextlist := makelist(nextstm); emit('goto _')* }

循环、复合的翻译模式

$S \rightarrow$ while M_1 E do M_2 S_1

{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
backpatch(*E.truelist*, M_2.*gotostm*) ;
S.nextlist := *E.falselist*;
emit('goto', M_1.*gotostm*)}

$S \rightarrow S_1$; M S_2

{ *backpatch*(S_1.*nextlist*, M_1.*gotostm*) ;
S.nextlist := S_2.*nextlist* }

$M \rightarrow \varepsilon$

{ *M.gotostm* := *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<D do
    if A=1 or B=1 then
        C:=C+1
    else
        A:=A+2;
```

在采用拉链与代码回填技术的表达式和语句的 S-翻译模式中，增加如下的 S-翻译模式片段：

E → E1 @ P E2 {
backpatch (E1.falselist, P.gotostm);
E.truelist := E2.falselist;
E.falselist := merge (E1.truelist, E2.truelist) }

P → ε { P.gotostm := nextstm }

利用扩充后的 S-翻译模式，对如下源语句进行语法制导翻译：

```
while A > B @ S < C do
    if C < D then A = A – B
    else S = S + 1
```