中国科学技术大学
University of Science and Technology of China

USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
中间代码生成 I

计算机科学与技术学院

李 诚

2021-10-25

中间代码生成

- 类型检查
- 声明语句的翻译
- 记录或结构体的翻译
- 数组寻址的翻译
- 表达式的翻译
- 控制流语句的翻译
- switch语句的翻译
- 过程或函数的翻译

涉及到类型表达式、静态相对地址分配组织，因此，单独设计一个章节来讲

表达式的中间代码生成

❑表达式文法

$$S \rightarrow \text{id} := E \qquad E \rightarrow E_1 + E_2 \mid -E_1 \mid (E_1) \mid \text{id}$$

❑语法制导翻译方案

- ❖属性：*E.place* 存放结果的地址
- ❖语义动作：从符号表中获取id的地址
  - ➢*lookup(id.lexeme);*如果不存在，返回*nil*
- ❖语义动作：产生临时变量
  - ➢*newTemp();* 保存中间结果
- ❖语义动作：输出翻译后的三地址指令
  - ➢*gen(addr, op, arg1, arg2)*
  - ➢该动作将地址和运算符及临时变量拼接为字符串

表达式的中间代码翻译方案

中国科学技术大学
University of Science and Technology of China

$S \rightarrow$ id := E {*p* = *lookup*(id.*lexeme*);

if *p* != *nil* then

gen (*p*, '=', *E.place*)

else *error* }

$E \rightarrow E_1 + E_2$

{*E.place* = *newTemp*();

gen (*E.place*, '=', E_1.*place*, '+', E_2.*place*) }

表达式的中间代码翻译方案

中国科学技术大学
University of Science and Technology of China

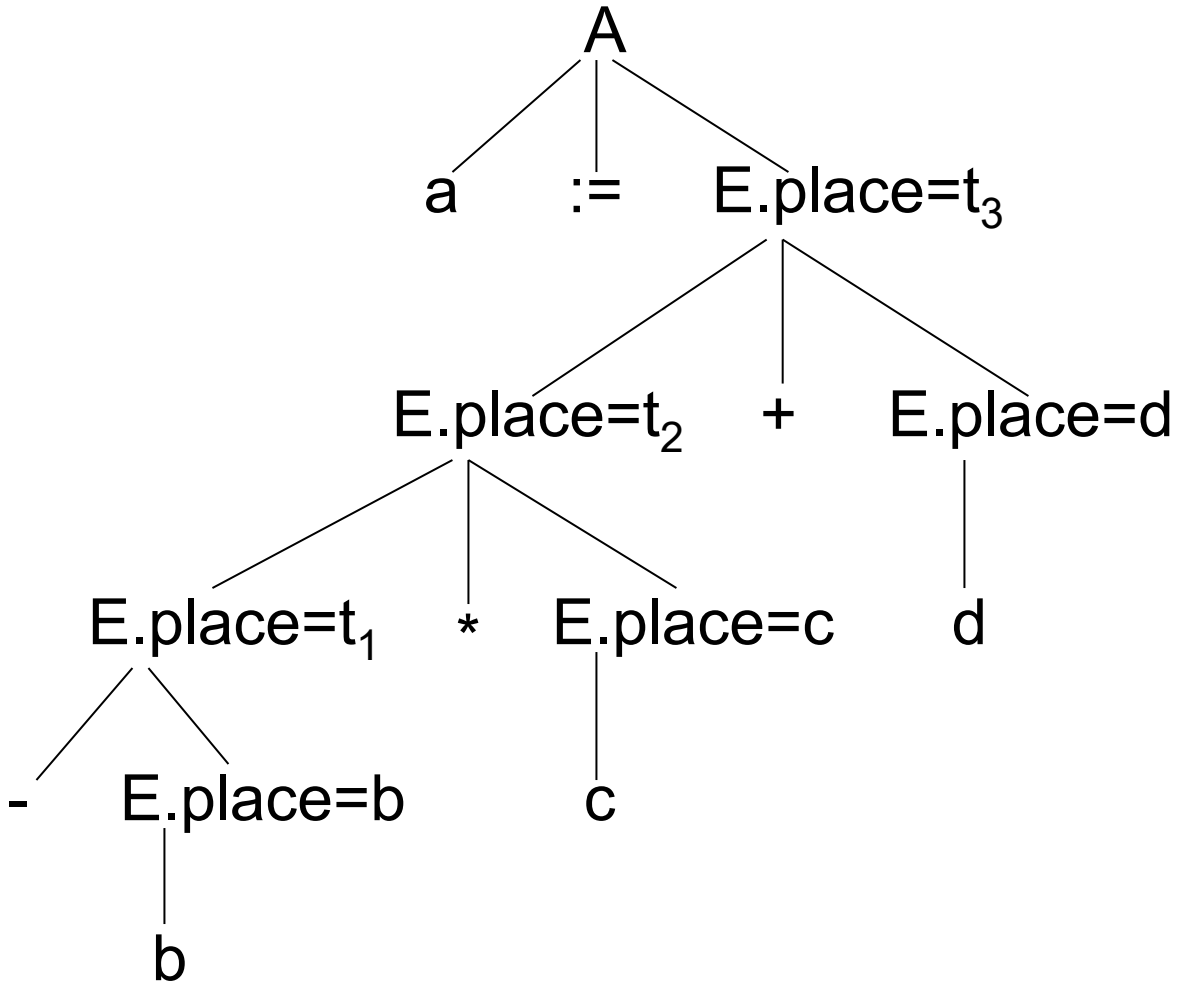
$E \rightarrow -E_1$ {*E.place = newTemp*();
gen (*E.place*, '=', 'uminus', E_1.*place*) }

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

$E \rightarrow$ id {*p* = *lookup*(id.*lexeme*);
if *p* != *nil* then
E.place = *p*
else *error* }

举例：表达式翻译

❑a := -b*c+d的翻译



TAC:

1) $t_1 := - b$

2) $t_2 := t_1 * c$

3) $t_3 := t_2 + d$

4) $a := t_3$

中间代码生成

- 类型检查
- 声明语句的翻译
- 记录或结构体的翻译
- 数组寻址的翻译
- 表达式的翻译
- **控制流语句的翻译**
- **switch语句的翻译**
- **过程或函数的翻译**

涉及到类型表达式以及内存组织形式，因此，单独设计一个章节来讲

控制流语句的翻译

$S \rightarrow$ if B then S_1

| if B then S_1 else S_2

| while B do S_1

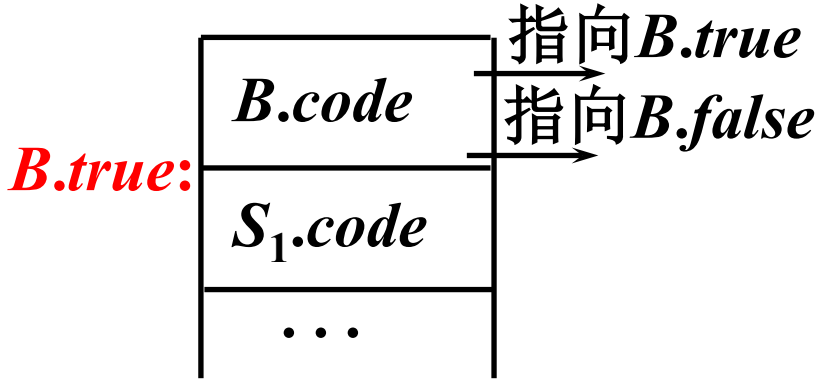
| S_1; S_2

if 语句中间代码生成的SDD

❑问题与对策

- ❖需要知道B为真或假时的跳转目标
- ❖B、S_1、S_2分别会输出多少条指令是不确定的
- ❖引入标号：先确定标号，在目标确定时输出标号指令，可调用newLabel() 产生新标号，每条语句有next 标号

$S \rightarrow$ if B then S_1



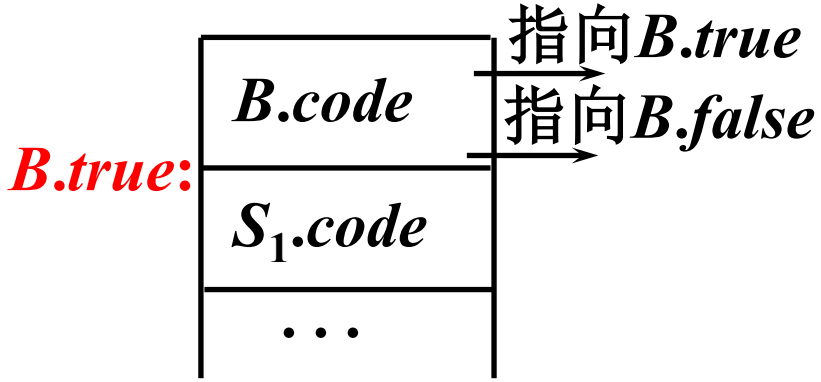
(a) if-then

if 语句中间代码生成的SDD

❑问题与对策

- ❖需要知道B为真或假时的跳转目标
- ❖B、S_1、S_2分别会输出多少条指令是不确定的
- ❖引入标号：先确定标号，在目标确定时输出标号指令，可调用newLabel() 产生新标号，每条语句有next 标号

$S \rightarrow$ if B then S_1



(a) if-then

语义规则：

B.true = *newLabel*();

B.false = *S.next*;//继承属性

$S_1.next = S.next$;

$S.code = B.code \,||\, gen(B.true, ':') \,||\, S_1.code$

if 语句中间代码生成的SDD

❑问题与对策

- ❖需要知道B为真或假时的
- ❖B、S_1、S_2分别会输出多
- ❖引入标号：先确定标号，
  调用newLabel() 产生新标

可

- 标号指向S内部的三地址代码时需要调用newLabel
- 标号指向S外部的三地址代码时从S继承

$S \rightarrow$ if B then S_1

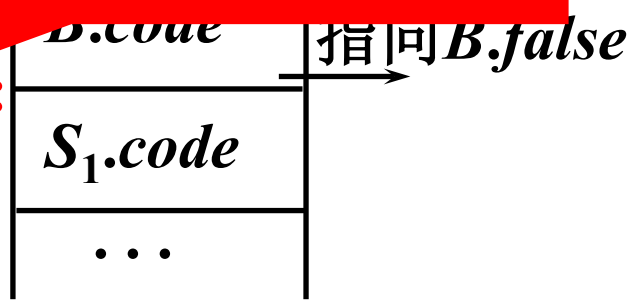
语义规则：

$B.true = newLabel();$

$B.false = S.next;$//继承属性

$S_1.next = S.next;$

$S.code = B.code \,||\, gen(B.true, ':') \,||\, S_1.code$



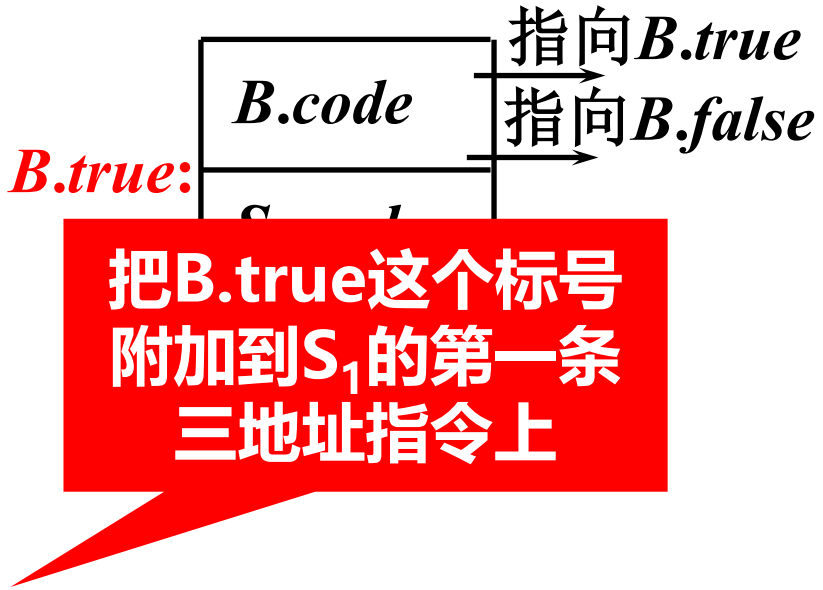
(a) if-then

if 语句中间代码生成的SDD

❑问题与对策

- ❖需要知道B为真或假时的跳转目标
- ❖B、S_1、S_2分别会输出多少条指令是不确定的
- ❖引入标号：先确定标号，在目标确定时输出标号指令，可调用newLabel() 产生新标号，每条语句有next 标号

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



语义规则：

B.true = *newLabel*();

B.false = *S.next*;//**继承属性**

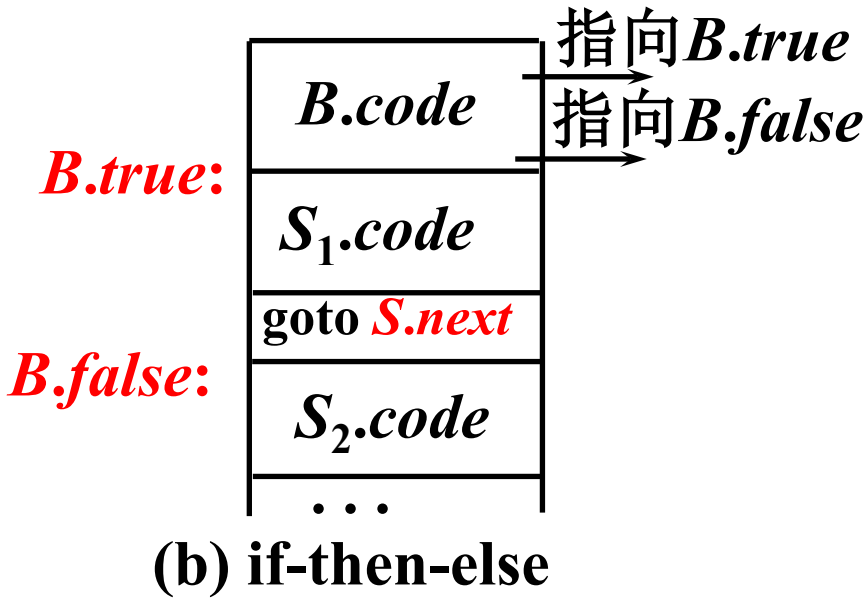
$S_1.next = S.next$;

$S.code = B.code \parallel gen(B.true, \text{':'}) \parallel S_1.code$

if 语句中间代码生成的SDD

- 考虑带有else的语句

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



(b) if-then-else

if 语句中间代码生成的SDD

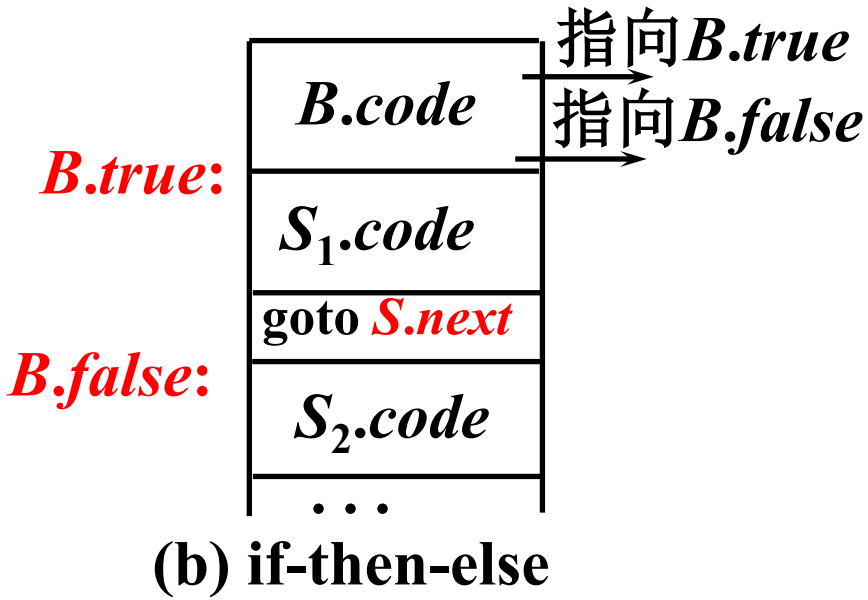
- 考虑带有else的语句

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

语义规则：

$B.true = newLabel();$

$B.false = newLabel();$



(b) if-then-else

if 语句中间代码生成的SDD

❑考虑带有else的语句

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

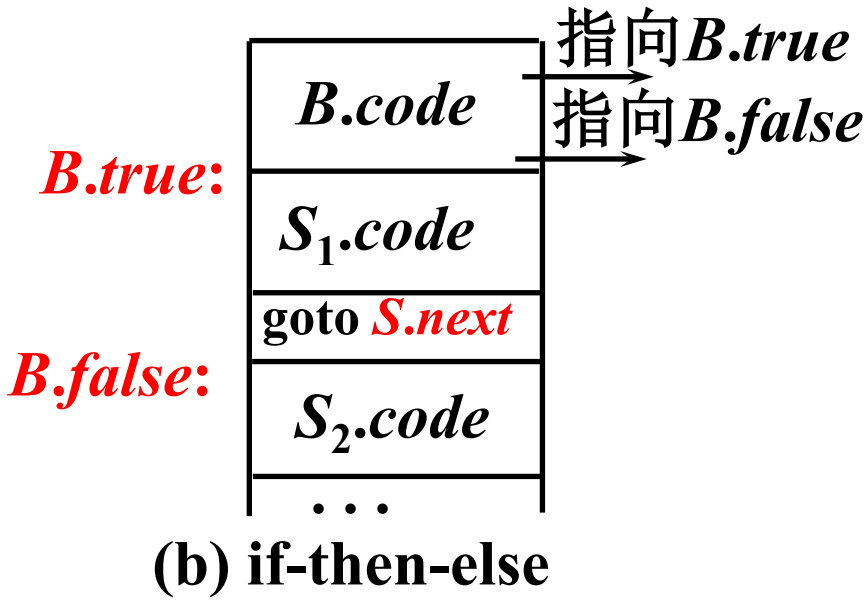
语义规则：

$B.true = newLabel();$

$B.false = newLabel();$

$S_1.next = S.next;$

$S_2.next = S.next;$

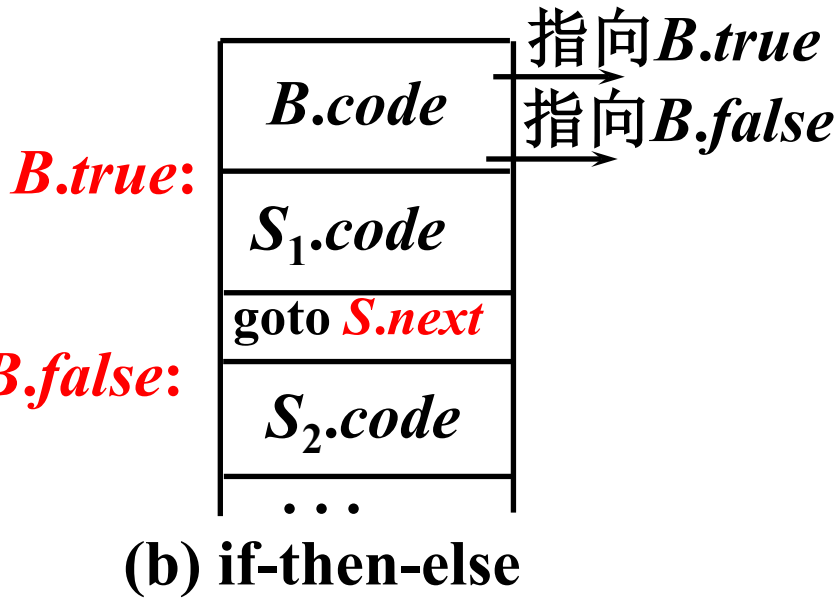


(b) if-then-else

if 语句中间代码生成的SDD

❑考虑带有else的语句

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



(b) if-then-else

语义规则：

$B.true = newLabel();$

$B.false = newLabel();$

$S_1.next = S.next;$

$S_2.next = S.next;$

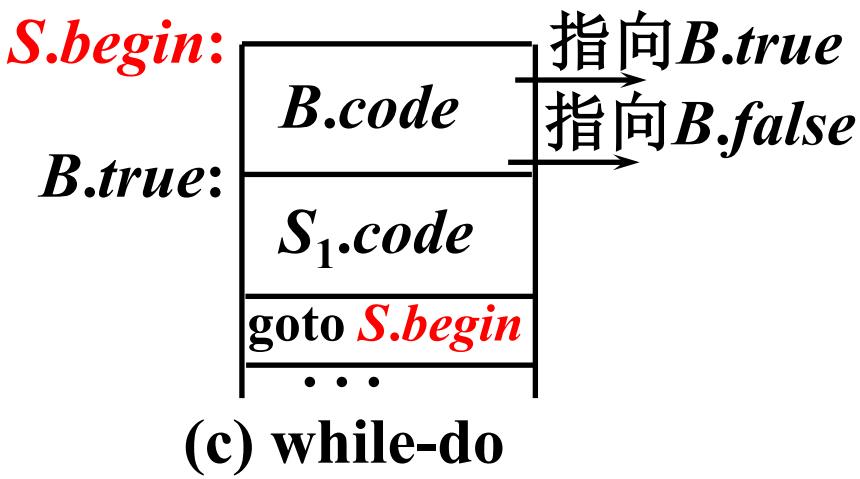
$S.code = B.code \,||\, gen(B.true, \text{':'}) \,||\, S_1.code \,||$

$gen(\text{'goto'}, S.next) \,||\, gen(B.false, \text{':'}) \,||\, S_2.code$

while语句中间代码生成SDD

- 引入开始标号*S.begin*,作为循环的跳转目标

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



while语句中间代码生成SDD

❑引入开始标号*S.begin*,作为循环的跳转目标

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

语义规则:

S.begin = *newLabel*();

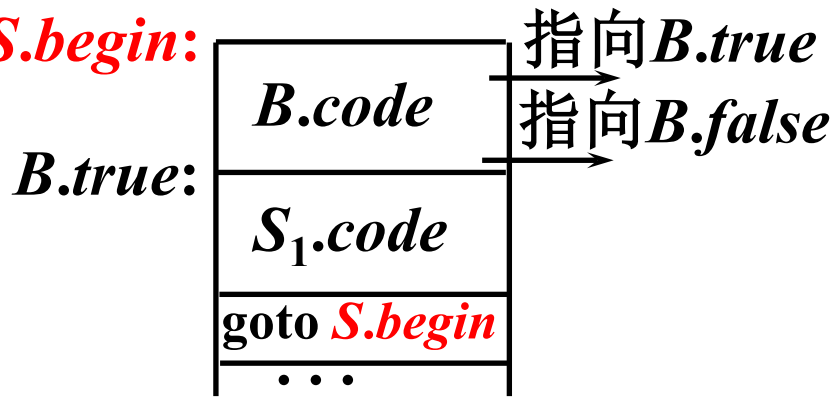
B.true = *newLabel*();

B.false = *S.next*;

$S_1.next = S.begin$;

S.code = *gen*(*S.begin*, ':') || *B.code* ||

gen(*B.true*, ':') || $S_1.code$ || *gen*('goto', *S.begin*)



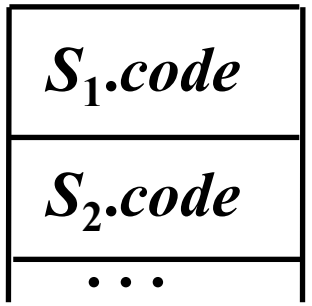
(c) while-do

顺序结构中间代码生成SDD

- 为每一语句S_1引入其后的下一条语句的标号

$S_1.next$

$S \rightarrow S_1; S_2$



(d) $S_1; S_2$

语义规则：

$S_1.next = newLabel(); S_2.next = S.next;$

$S.code = S_1.code \,||\, gen(S_1.next, ':') \,||\, S_2.code$

布尔表达式的控制流翻译

- 在if-else以及while语句翻译中，并未对B.code进行展开，现在考虑B.code的三地址代码翻译
- 如果B是$a < b$的形式，

那么翻译生成的三地址码是：

if $a < b$ goto $B.true$

goto $B.false$

我们把这种翻译称作为“布尔表达式的控制流翻译”

- 布尔表达式有两个基本目的
  - 计算逻辑值
    - 例如：作为赋值语句的右值
  - 在控制流语句中用作条件表达式
    - 例如：*if(B) then S*
- 本节所用的布尔表达式文法

$$B \rightarrow B \text{ or } B \mid B \text{ and } B \mid \text{not } B \mid (B) \mid E \text{ relop } E \mid \text{true} \mid \text{false}$$

布尔表达式

- 布尔表达式有两个基本目的
  - 计算逻辑值
  - 在控制流语句中用作条件表达式
- 本节所用的布尔表达式文法

$$B \rightarrow B \text{ or } B \mid B \text{ and } B \mid \text{not } B \mid (B) \mid E \text{ relop } E \mid \text{true} \mid \text{false}$$

  - 布尔运算符 or 、and 和 not（优先级、结合性）
  - 关系运算符 relop：＜、≤、＝、≠、＞和≥
  - 布尔常量：true和false

布尔表达式

❑布尔表达式的完全计算

- ❖值的表示数值化
- ❖其计算类似于算术表达式的计算

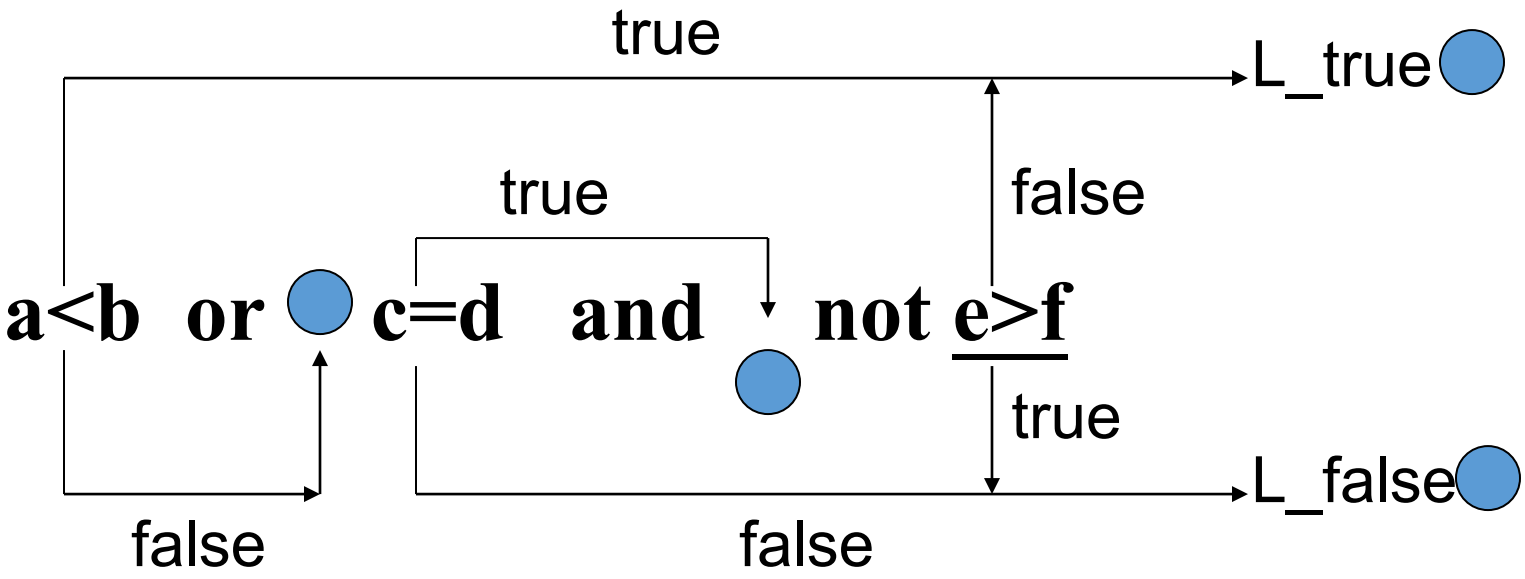
true and false or (2 > 1)的计算为

➔false or (2>1)➔false or true➔true

❑布尔表达式的“短路”计算

- ❖ B_1 or B_2　　B_1 为真即为真
- ❖ B_1 and B_2　　B_1 为假即为假

布尔表达式的短路计算



L_true-真出口：整个布尔表达式为**真**时，控制流应转移到的目标语句（代码）；反之为**假**时则转到 L_false-假出口。

- 表示转移到的目标语句在有关布尔表达式翻译时尚未确定。

布尔表达式的控制流翻译

- 用控制流来实现计算
  - 布尔运算符and, or, not不出现在翻译后的代码中
  - 用程序中的位置来表示值
- 例：if(x<3 or x>5 and x!=y) x =10;的翻译

```
        if x<3 goto L2
        goto L3
L3:     if x>5 goto L4
        goto L1
L4:     if x!=y goto L2
        goto L1
L2:     x = 10
L1:
```

布尔表达式的控制流翻译

❑例　表达式

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

的三地址码是:

if $a < b$ goto L_{true}

goto L_1

L_1: if $c < d$ goto L_2

goto L_{false}

L_2: if $e < f$ goto L_{true}

goto L_{false}

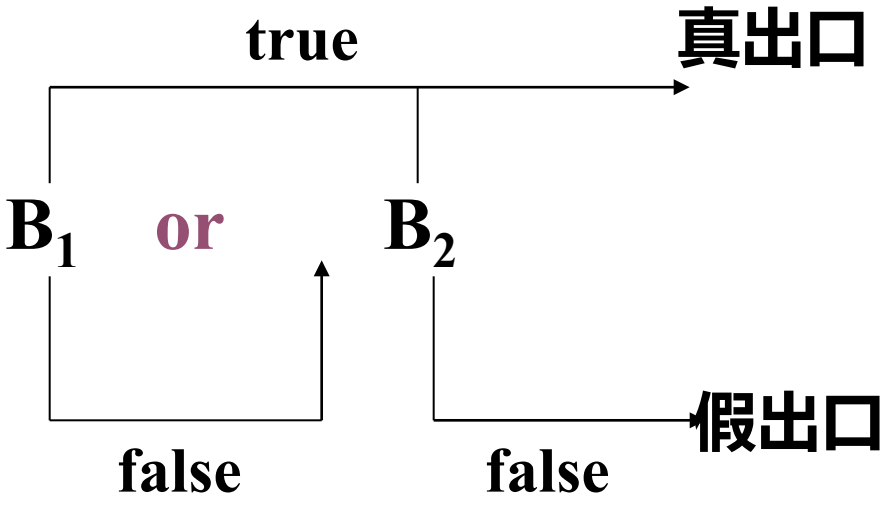
布尔表达式的控制流翻译

中国科学技术大学
University of Science and Technology of China

$B \rightarrow B_1$ or B_2

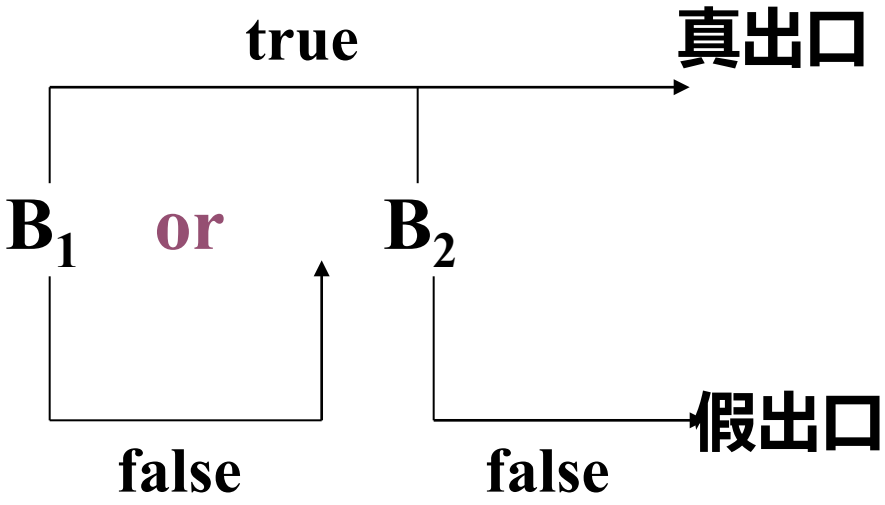
布尔表达式的控制流翻译

$B \rightarrow B_1$ or B_2



布尔表达式控制流翻译SDD

$B \rightarrow B_1$ or B_2



语义规则：

$B_1.true = B.true;$

$B_1.false = newLabel();$

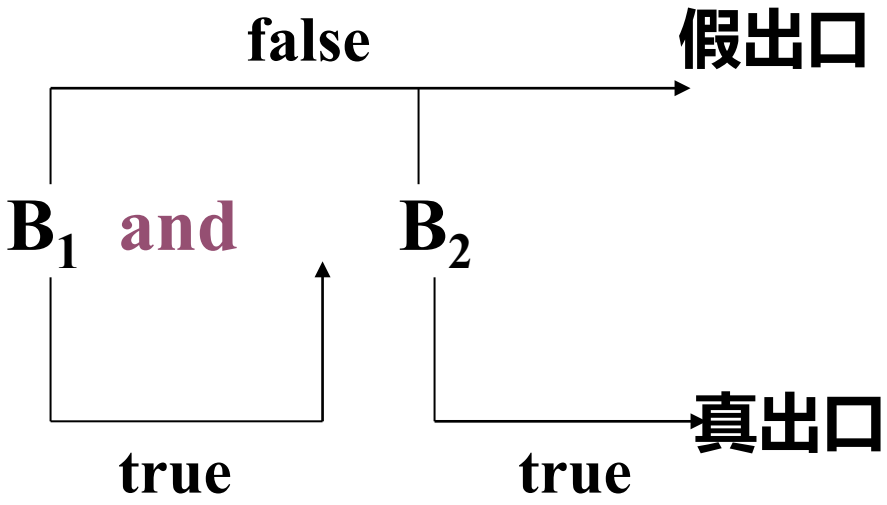
$B_2.true = B.true;$

$B_2.false = B.false;$

$B.code = B_1.code \,||\, gen(B_1.false, ':') \,||\, B_2.code$

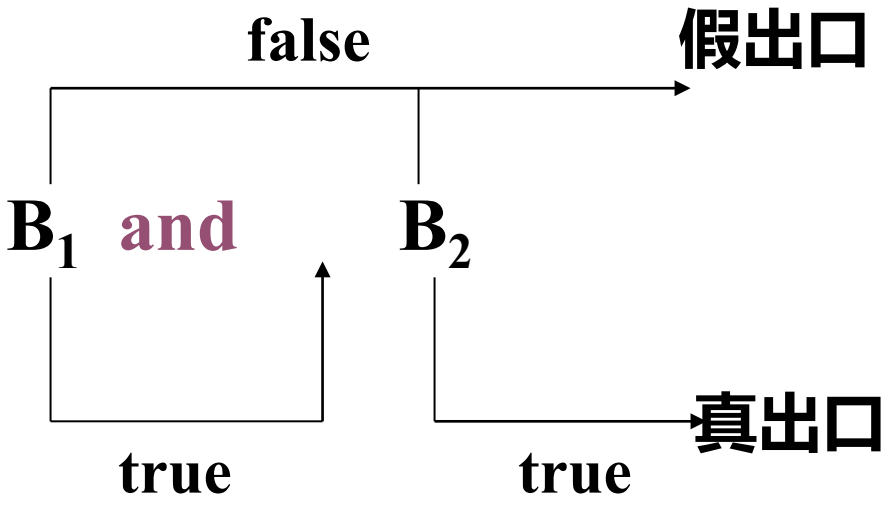
布尔表达式的控制流翻译

$B \rightarrow B_1$ and B_2



布尔表达式控制流翻译SDD

$B \rightarrow B_1$ and B_2



语义规则：

$B_1.true = newLabel();$

$B_1.false = B.false;$

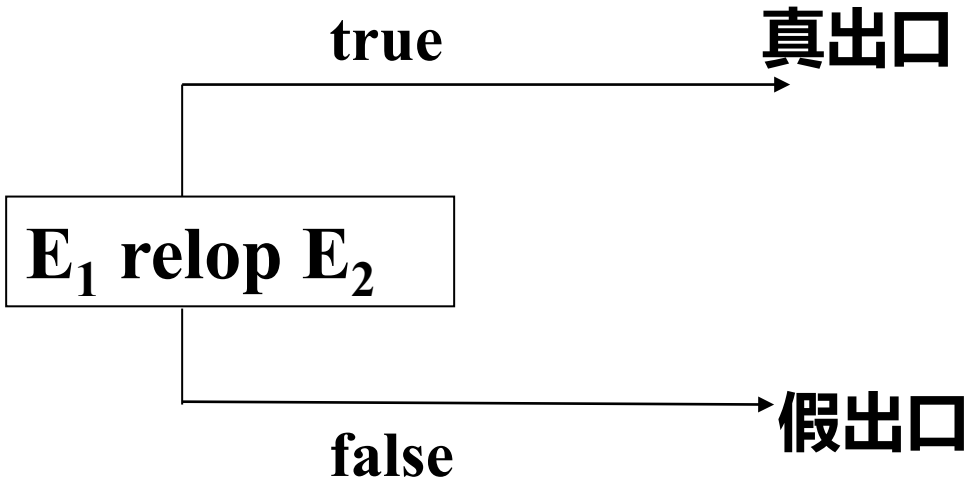
$B_2.true = B.true;$

$B_2.false = B.false;$

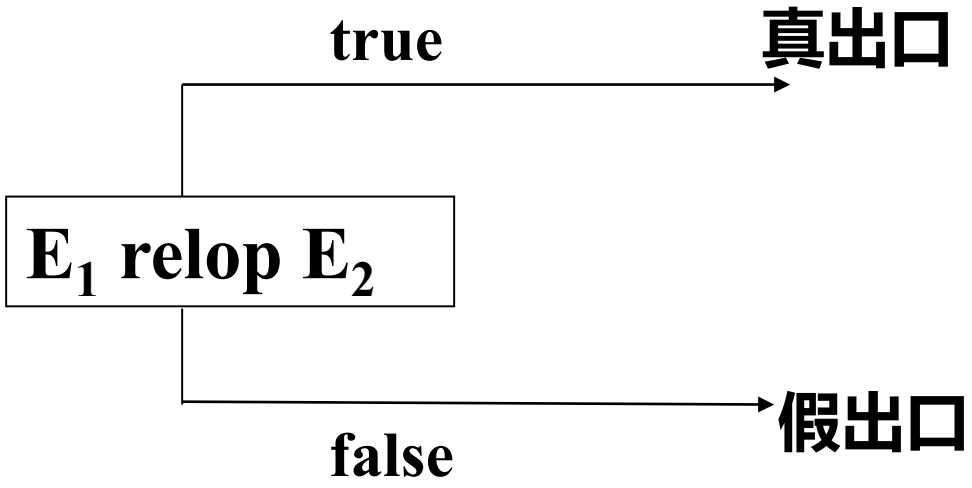
$B.code = B_1.code \,||\, gen(B_1.true, \text{':'}) \,||\, B_2.code$

布尔表达式的控制流翻译

$B \rightarrow E_1$ relop E_2



$B \rightarrow E_1$ relop E_2



语义规则：

$B.code = E_1.code \,||\, E_2.code \,||$

$gen(\text{‘if’}, E_1.place, \text{relop}.op, E_2.place,$

$\text{‘goto’}, B.true) \,||$

$gen(\text{‘goto’}, B.false)$

$B \rightarrow (B_1)$

语义规则：

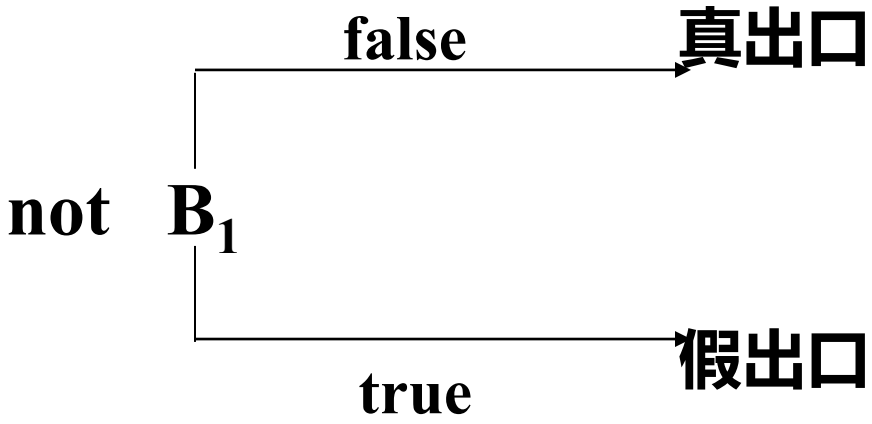
$B_1.true = B.true;$

$B_1.false = B.false;$

$B.code = B_1.code$

布尔表达式的控制流翻译

$B \rightarrow \text{not } B_1$



布尔表达式控制流翻译SDD

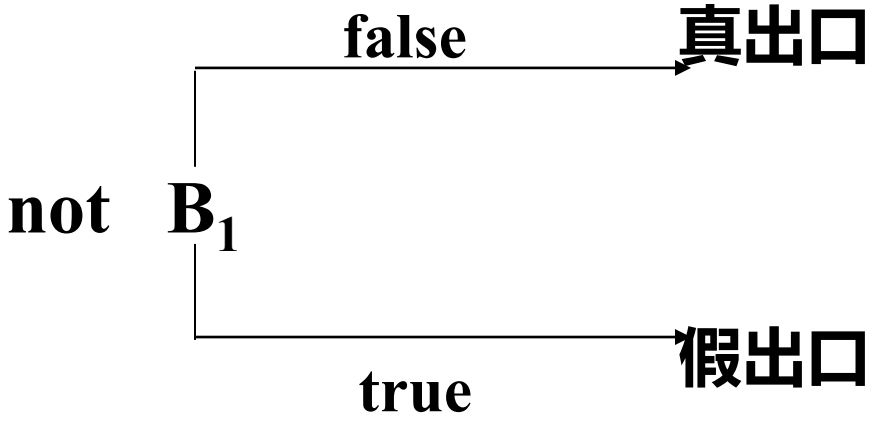
$B \rightarrow \text{not } B_1$

语义规则：

$B_1.true = B.false;$

$B_1.false = B.true;$

$B.code = B_1.code$



$B \rightarrow \text{true}$

语义规则：

$B.code = gen(\text{‘goto’}, B.true)$

$B \rightarrow \text{false}$

语义规则：

$B.code = gen(\text{‘goto’}, B.false)$

翻译难点

- 关键问题：将跳转指令与目标匹配起来
- B.true, B.false都是继承属性
- 需要两趟分析来计算
  - 1 pass：生成语法树
  - 2 pass：深度优先遍历树，计算属性值
    - 将标号和具体地址绑定起来
- 能否一趟完成?
  - 标号回填技术

回填(backpatching)技术

- 问题：
  - 布尔表达式短路计算翻译中，产生了转移目标不明确的条件或无条件代码；

回填(backpatching)技术

- 问题：
  - 布尔表达式短路计算翻译中，产生了转移目标不明确的条件或无条件代码；
- 解决方案：
  - 当生成跳转指令时，暂时不指定目标地址
  - 当有关目标地址确定后，再填回到翻译代码中

回填(backpatching)技术

- 问题：
  - 布尔表达式短路计算翻译中，产生了转移目标不明确的条件或无条件代码；
- 解决方案：
  - 当生成跳转指令时，暂时不指定目标地址
  - 当有关目标地址确定后，再填回到翻译代码中
- 具体实现：
  - 将有相同转移目标的转移代码的编号串起来形成链，可以方便回填目标地址。
  - 该list变成了综合属性，可以与LR结合
  - 注：后面的翻译均是与LR结合的语法制导翻译方案

相关符号属性

- 对布尔表达式而言，有两个综合属性：
  - B.truelist：代码中所有转向真出口的代码指令链；
  - B.falselist：所有转向假出口的代码指令链；
  - 在生成B的代码时，跳转指令goto是不完整的，目标标号尚未填写，用truelist和falselist来管理

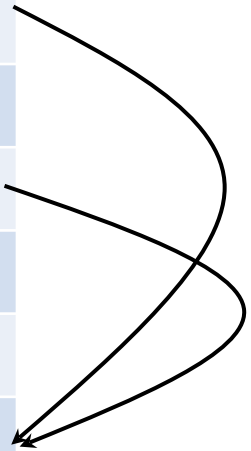
- 将生成的指令放入一个指令数组，指令的标号即为数组下标

标号	指令数组
100	…
101	goto -
102	…
103	goto -
104	
105	
106	

- 假设100-103号指令都属于布尔表达式B

❑将生成的指令放入一个指令数组，指令的标号即为数组下标

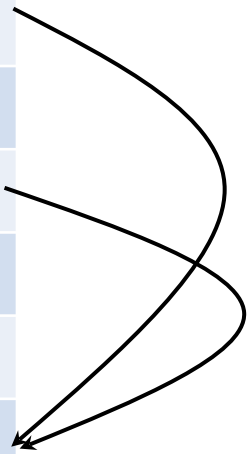
标号	指令数组
100	…
101	goto -
102	…
103	goto -
104	
105	
106	



- 假设100-103号指令都属于布尔表达式B
- 101和103号指令都指向B真出口

❑将生成的指令放入一个指令数组，指令的标号即为数组下标

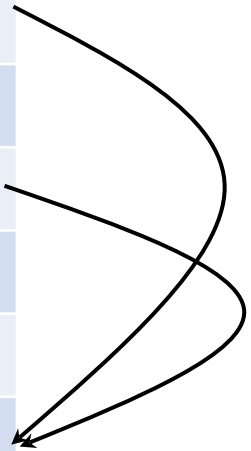
标号	指令数组
100	…
101	goto -
102	…
103	goto -
104	
105	
106	



- 假设100-103号指令都属于布尔表达式B
- 101和103号指令都指向B真出口
- B真出口是106，但还未生成
- B.truelist = {101, 103}

❑将生成的指令放入一个指令数组，指令的标号即为数组下标

标号	指令数组
100	…
101	goto **106**
102	…
103	goto **106**
104	…
105	…
106	…



- 假设100-103号指令都属于布尔表达式B
- 101和103号指令都指向B真出口
- B真出口是106，但还未生成
- B.truelist = {101, 103}
- 回填时，将101和103补齐

实现：数据结构及语义函数

中国科学技术大学
University of Science and Technology of China

makelist(i)

- 创建含标号为i的指令的链表
- i不是目标指令，而是源指令，也就是那一些不完整的goto指令

实现：数据结构及语义函数

中国科学技术大学
University of Science and Technology of China

backpatch(instruction-list, target-label)

- 将目标地址target-label填回instruction-list中每条指令
- 也就是将goto – 指令中不明确的目标补齐

实现：数据结构及语义函数

中国科学技术大学
University of Science and Technology of China

merge(instruction-list$_1$, instruction-list$_2$)

- 合并链list$_1$和list$_2$
- 要求list$_1$和list$_2$中每条指令都会跳转到同一条指令

短路计算及回填的翻译方案

$B \rightarrow \text{not } B_1$ { B.truelist = B_1.falselist;

B.falselist = B_1.truelist; }

$B \rightarrow (B_1)$ { B.truelist = B_1.truelist;

B.falselist = B_1.falselist; }

短路计算及回填的翻译方案

B→ true {

gen(“goto” -); }

短路计算及回填的翻译方案

B→ true {

B.truelist = makelist( );

/*为真时，执行无条件跳转指令，但是目标为空。当目标明确后，回填 所对应的跳转指令*/

gen(“goto” -); }

我们用变量nextinstr保存了紧跟着的下一条指令的序号

短路计算及回填的翻译方案

B→ true {

B.truelist = makelist(nextinstr);

/*为真时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -); }

短路计算及回填的翻译方案

B→ true {

B.truelist = makelist(nextinstr);

/*为真时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -); }

B→ false {

B.falselist = makelist(nextinstr);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -);}

短路计算及回填的翻译方案

$B \rightarrow E_1$ relop E_2 {

gen(“if” E_1.place relop.op E_2.place “goto” -);

gen(“goto” -); }

短路计算及回填的翻译方案

$B \rightarrow E_1$ relop E_2 {

B.truelist = makelist( );

B.falselist = makelist( );

/*为真时，执行条件跳转指令，但是目标为空。当目标明确后，回填 所对应的跳转指令*/

gen(“if” E_1.place relop.op E_2.place “goto” -);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填 所对应的跳转指令*/

gen(“goto” -); }

短路计算及回填的翻译方案

$B \rightarrow E_1$ relop E_2 {

B.truelist = makelist(nextinstr);

B.falselist = makelist(nextinstr+1);

/*为真时，执行条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

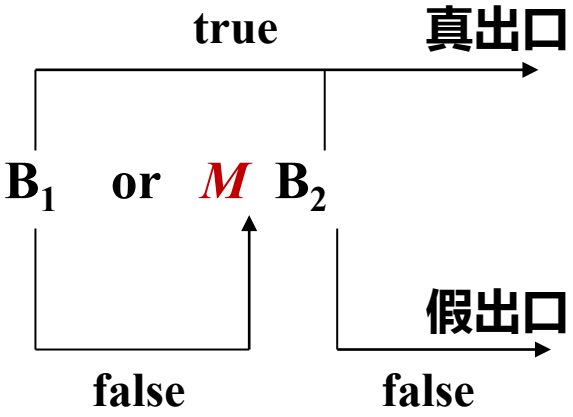
gen(“if” E_1.place relop.op E_2.place “goto” -);

/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr+1所对应的跳转指令*/

gen(“goto” -); }

短路计算及回填的翻译方案

$B \rightarrow B_1$ or M B_2



/*获取下一三地址代码（语句）的编号（作为转移目标来回填），在自底向上的语法分析中传递信息；
在分析B_2之前做，因此可以保存B_2开始的第一条指令的地址*/

M→ε { M.instr = nextinstr}

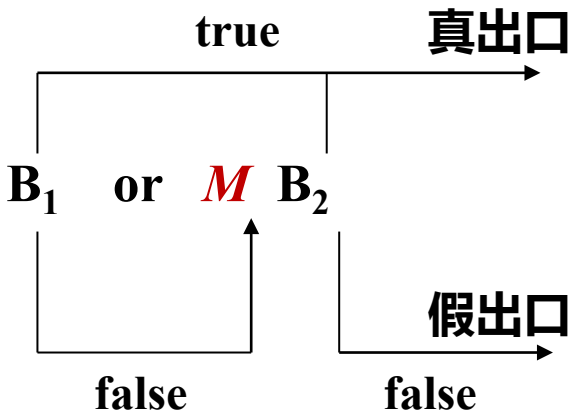
短路计算及回填的翻译方案

$B \rightarrow B_1$ or M B_2

{ backpatch(B_1.falselist, M.instr);

B.truelist = merge(B_1.truelist, B_2.truelist);

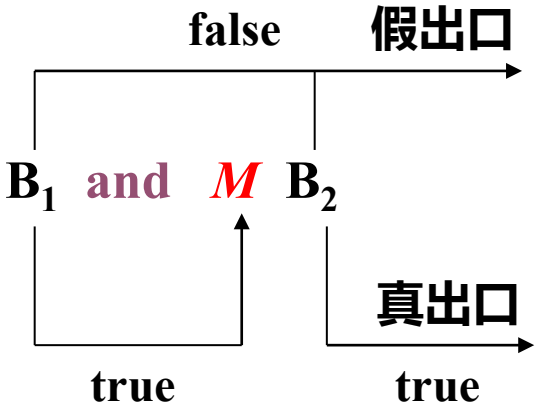
B.falselist = B_2.falselist; }



/*获取下一三地址代码（语句）的编号（作为转移目标来回填），在自底向上的语法分析中传递信息；
在分析B_2之前做，因此可以保存B_2开始的第一条指令的地址*/

$M \rightarrow \varepsilon$ { M.instr = nextinstr}

$B \rightarrow B_1$ and M B_2



$M \rightarrow \varepsilon$ { M.instr = nextinstr}//在分析B_2之前做，因此可以保存B_2开始的第一条指令的地址

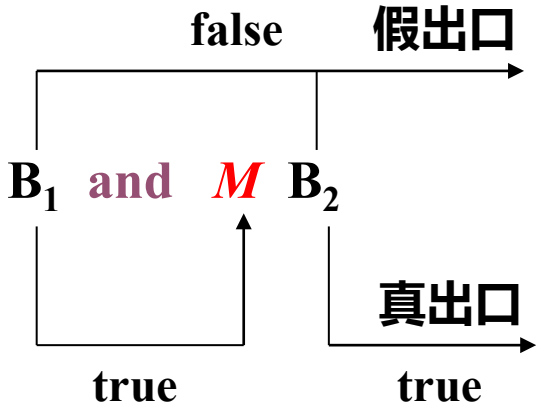
短路计算及回填的翻译方案

$B \rightarrow B_1$ and M B_2

{ backpatch(B_1.truelist, M.instr);

B.falselist = merge(B_1.falselist,B_2.falselist);

B.truelist = B_2.truelist; }

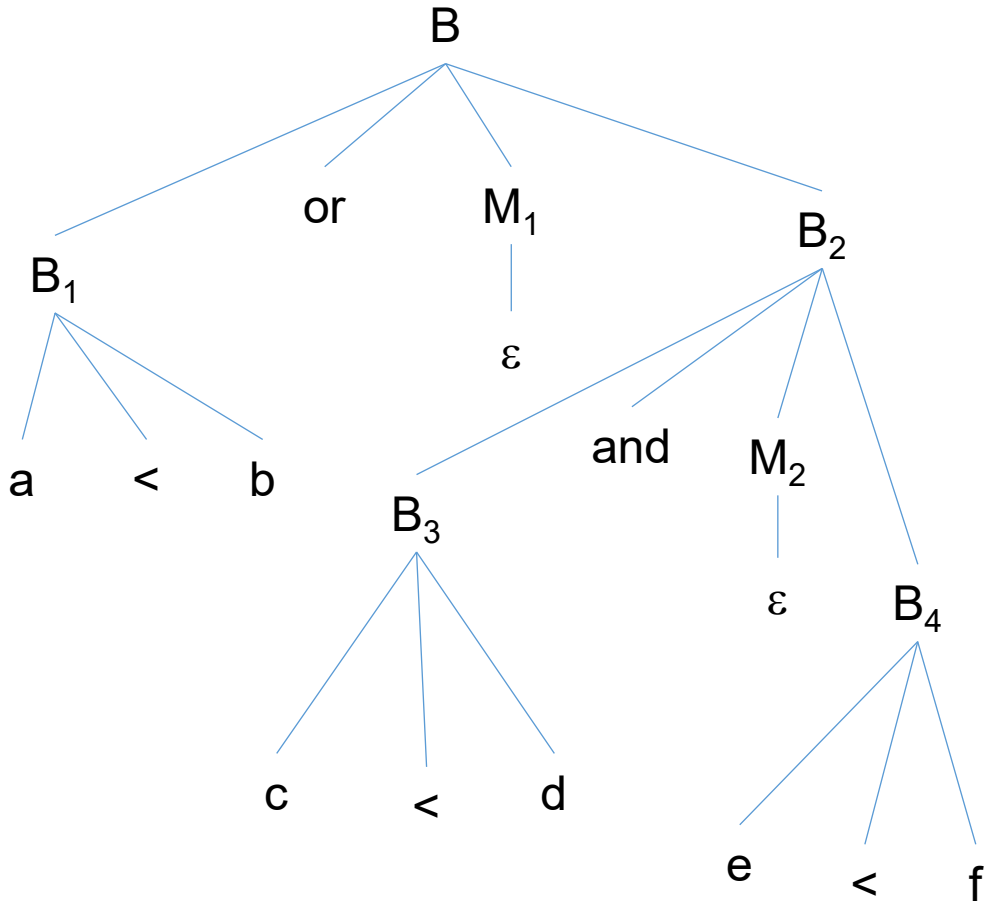


M→ε { M.instr = nextinstr}//在分析B_2之前做，因此可以保存B_2开始的第一条指令的地址

布尔表达式的翻译

a<b or c<d and e<f

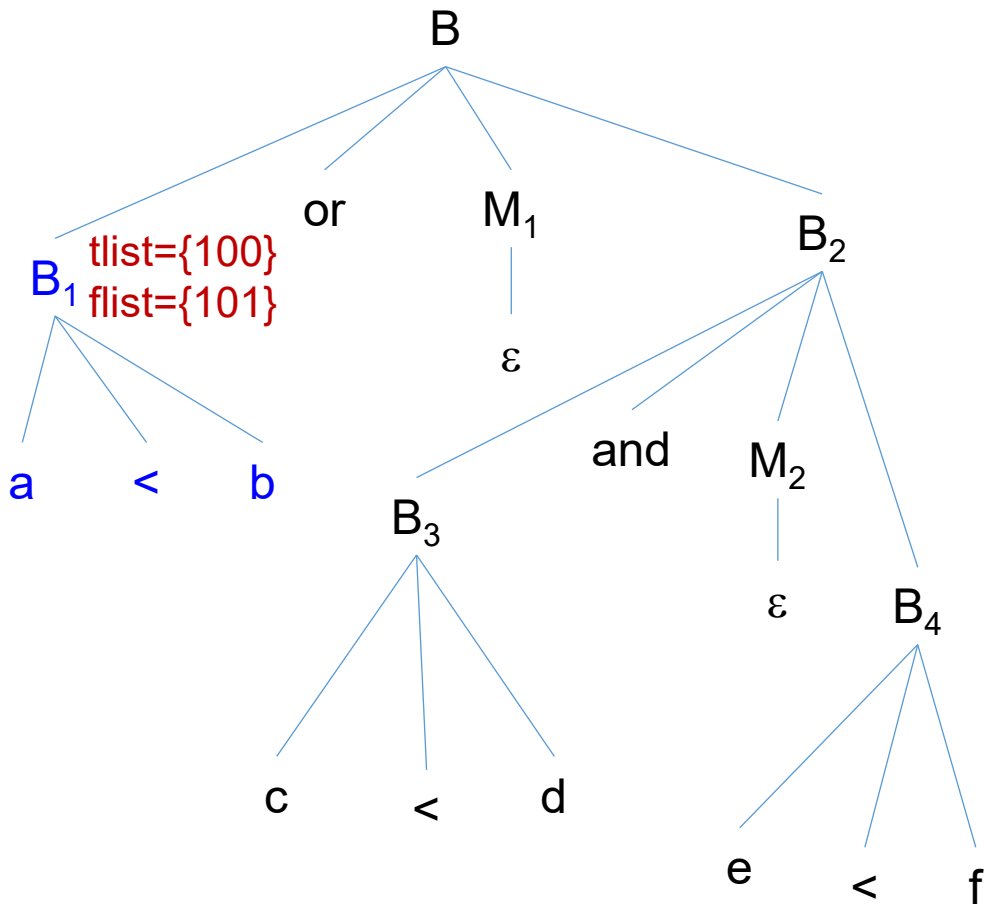
假设nextinstr = 100



布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

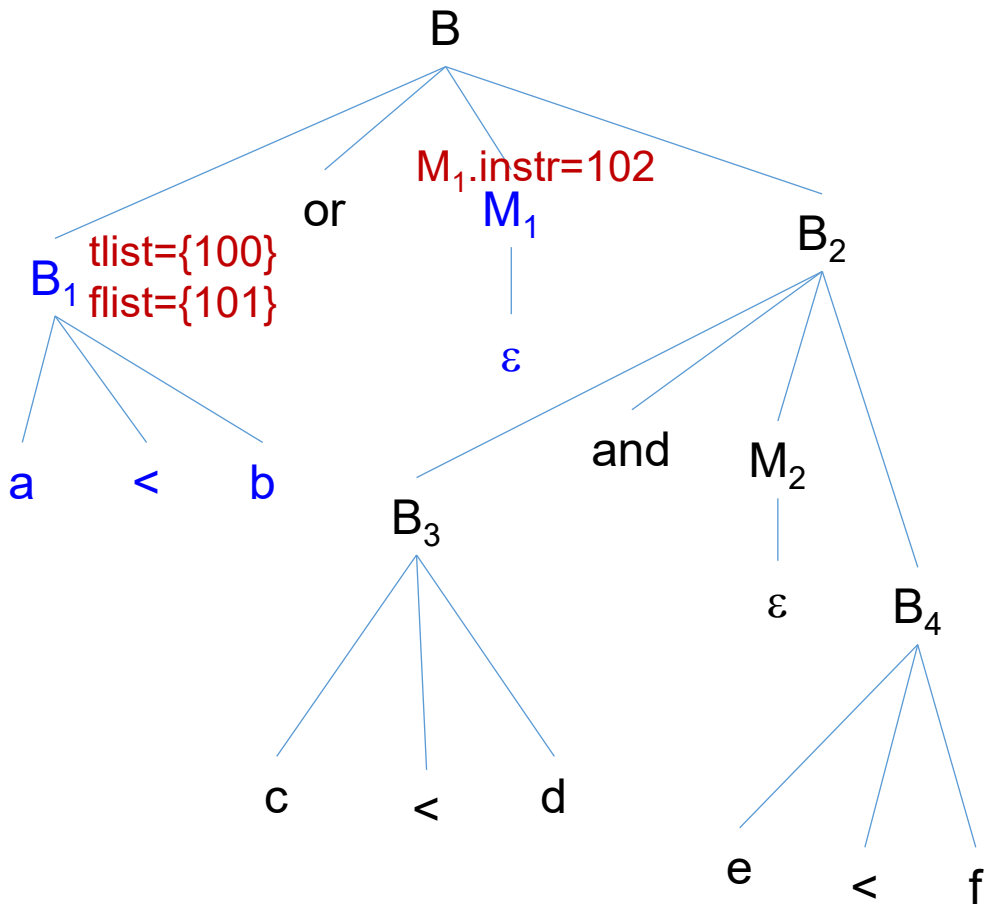


(100) if a<b goto -
(101) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

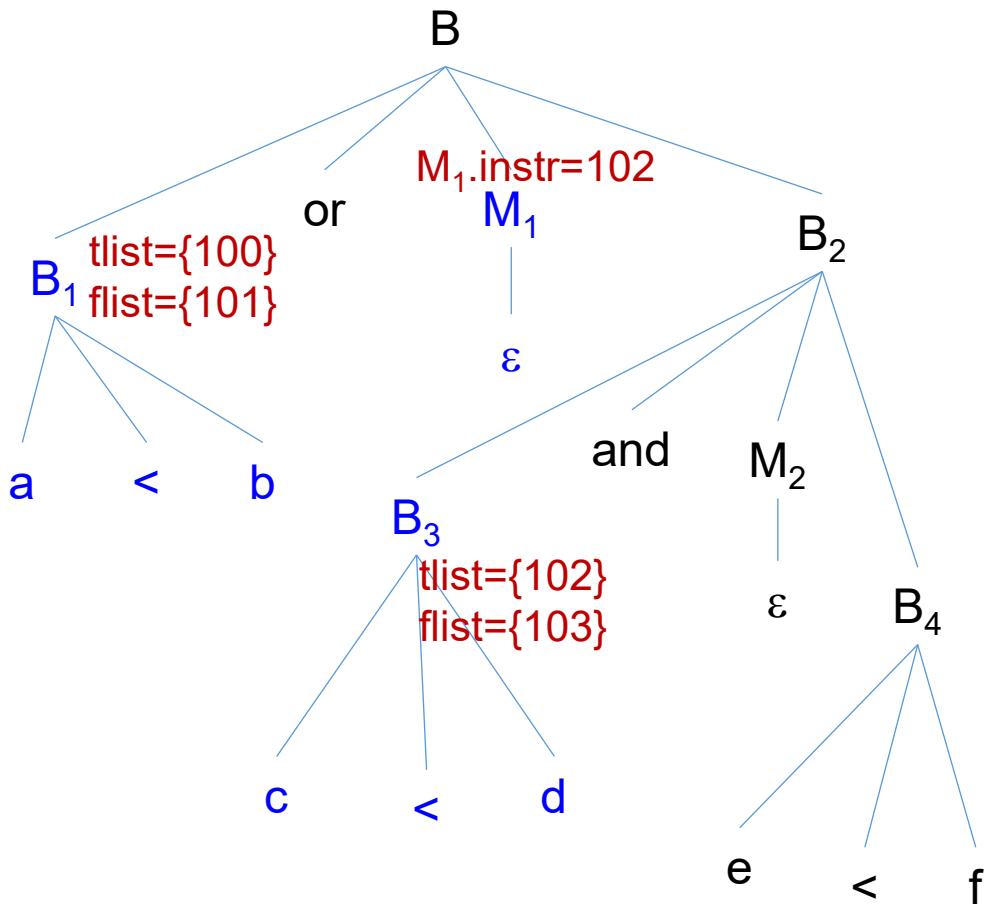


(100) if a<b goto -
(101) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

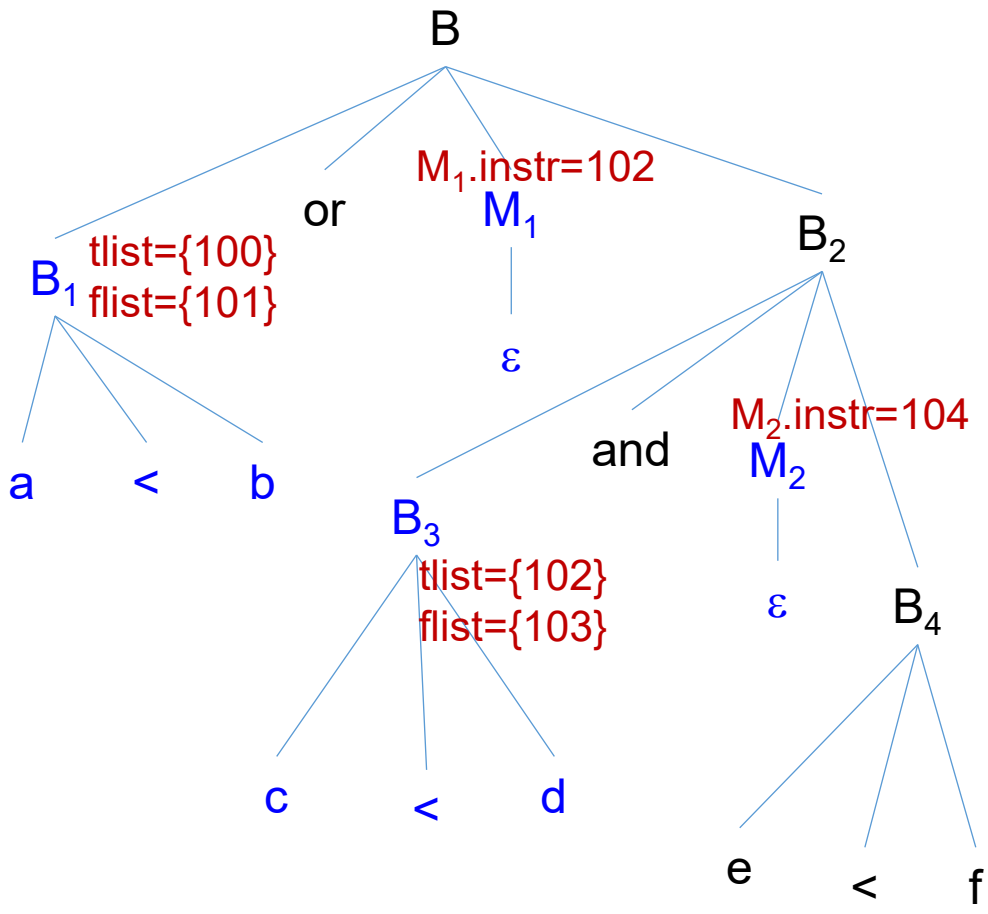


(100) if a<b goto -
(101) goto -
(102) if c<d goto -
(103) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

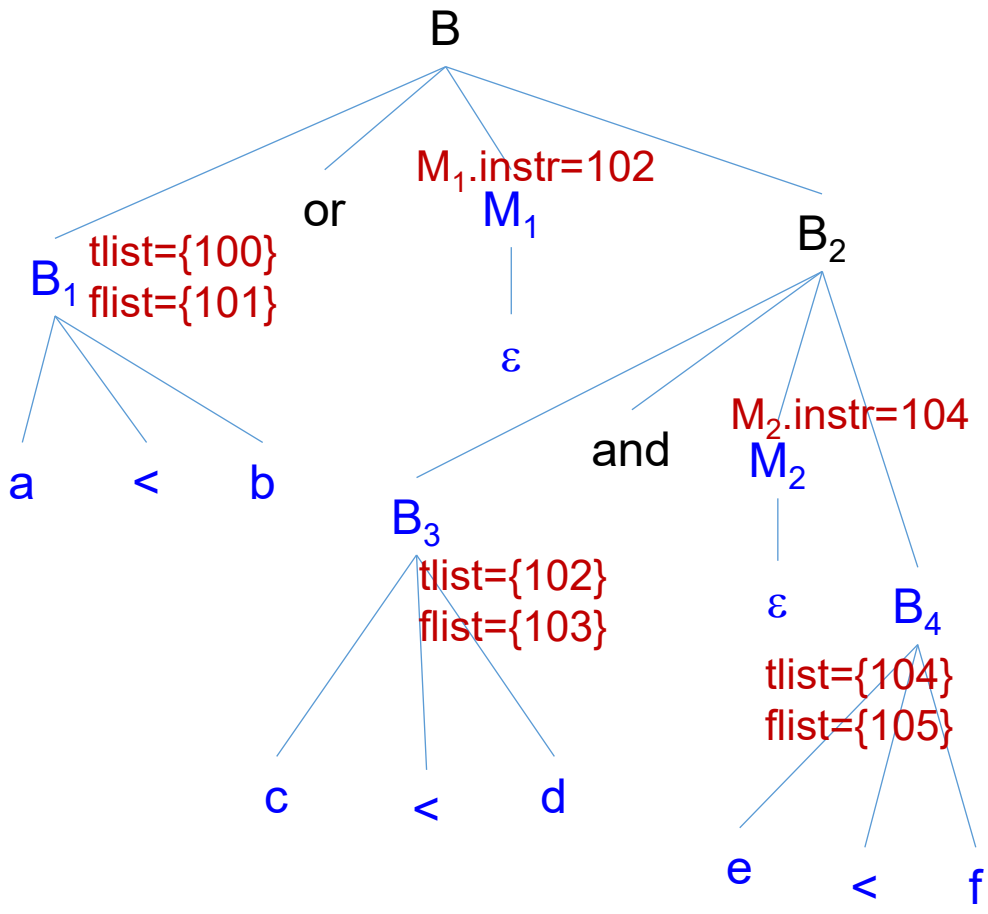


(100) if a<b goto -
(101) goto -
(102) if c<d goto -
(103) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

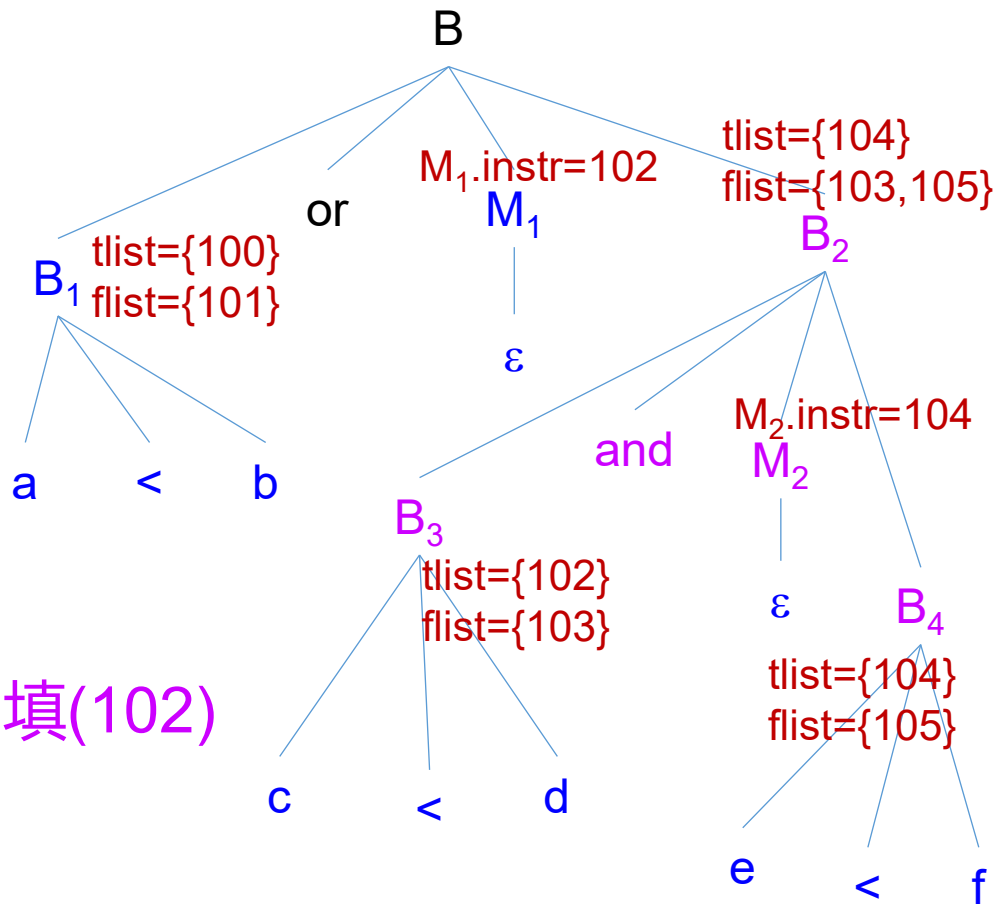


(100) if a<b goto -
(101) goto -
(102) if c<d goto -
(103) goto -
(104) if e<f goto -
(105) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

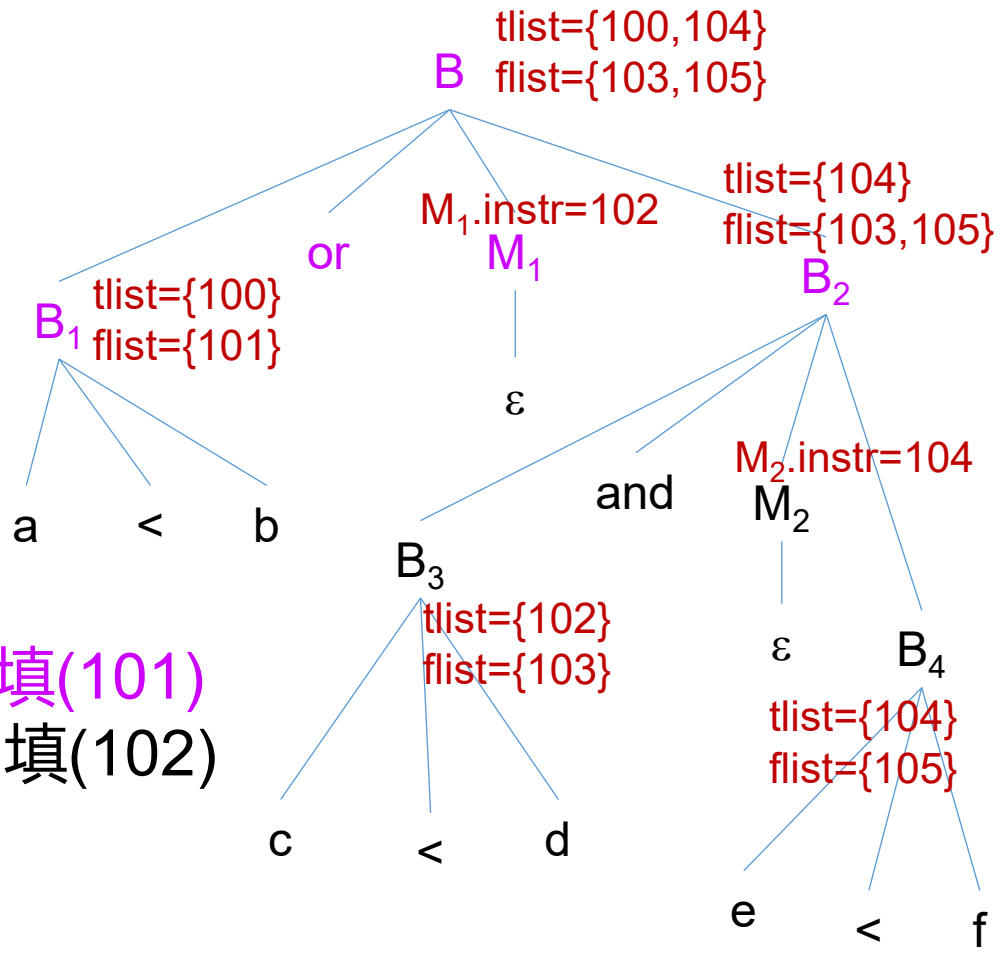


(100) if a<b goto -
(101) goto -
(102) if c<d goto 104//用104回填(102)
(103) goto -
(104) if e<f goto -
(105) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100



(100) if a<b goto -
(101) goto 102 //用102回填(101)
(102) if c<d goto 104//用104回填(102)
(103) goto -
(104) if e<f goto -
(105) goto -

其他部分的回填要依赖与其他语句的翻译

❑考虑布尔表达式

a>b or true and not c < f

- ❖ 参考本ppt中布尔表达式短路计算、标号回填等翻译技术，生成对应的三地址代码。
- ❖ 假设nextinstr = 200
- ❖ 除了三地址代码外，画出LR分析方法对应的注释分析树（如slide 69），标注出属性和属性值。
- ❖ 结合LR分析方法指出回填的具体细节
  - ➢ 在使用哪一个产生式归约时候进行的回填
  - ➢ 用哪一个标号，回填了哪一个不完整的goto指令

相关符号属性

- 对布尔表达式而言，有两个综合属性：
  - B.truelist：代码中所有转向真出口的代码指令链；
  - B.falselist：所有转向假出口的代码指令链；
  - 在生成B的代码时，跳转指令goto是不完整的，目标标号尚未填写，用truelist和falselist来管理
- 对一般语句而言，有一个综合属性：
  - S.nextlist：代码中所有跳转到紧跟S的代码之后的指令

例如：$S \rightarrow \{L\}$ //程序块

$S \rightarrow A$ //赋值语句

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

控制流语句文法汇总

- 用S和L分别表示一条语句和语句列表

  $S \rightarrow$ if B then S_1 // (B) 的括号此处省略

  $S \rightarrow$ if B then S_1 else S_2

  while B do S_1

  $S \rightarrow$ A //赋值语句

  $S \rightarrow$ {L} //大括号的作用是把内部的多个语句绑在一起，当成一个语句。

  $L \rightarrow L;S \mid S$ //语句列表，L和S之间用;分割

控制流语句文法-例

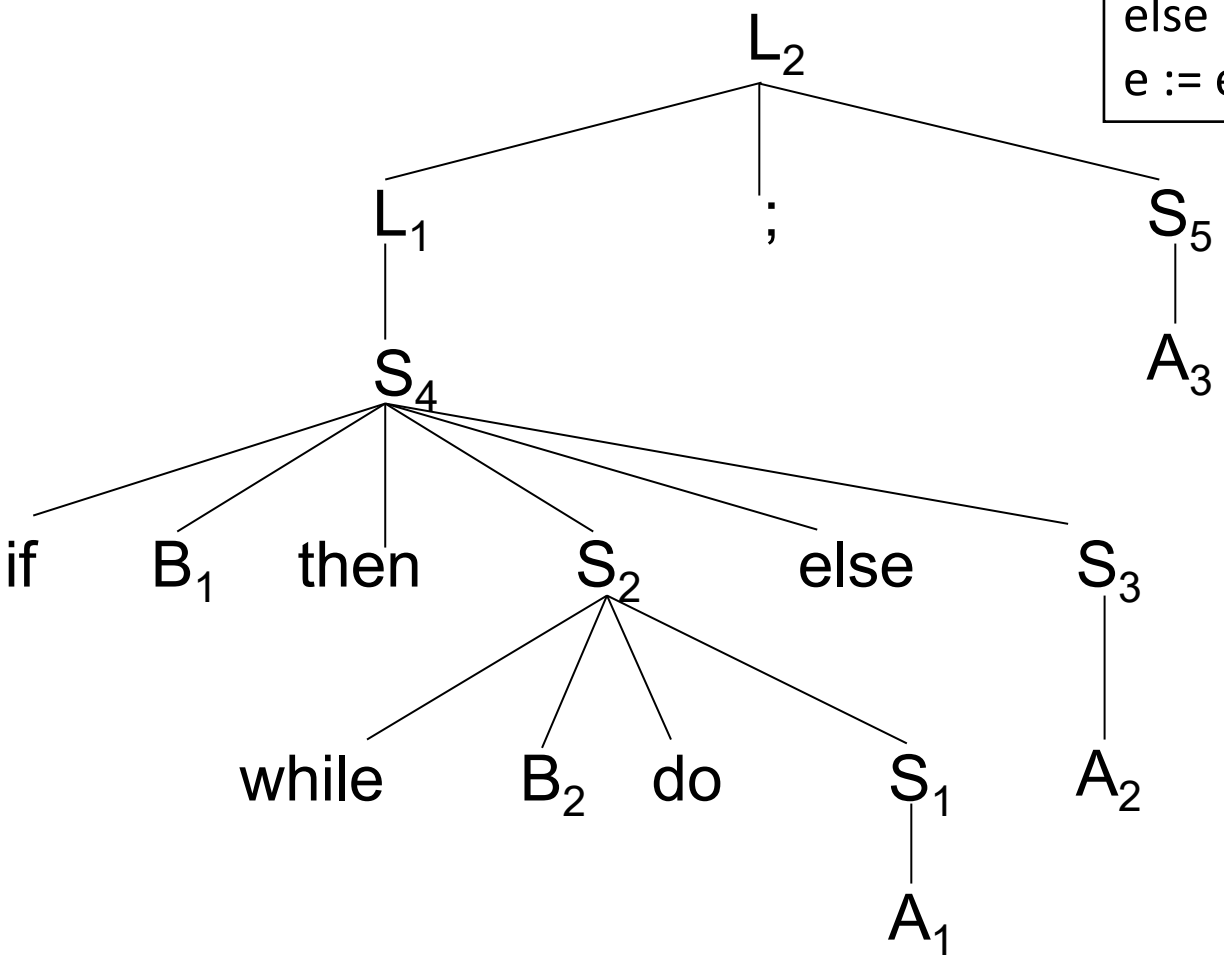
考虑以下语句序列：

```
if ( a<b or c<d and e<f )  then
  while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```

控制流语句文法-例

❑分析树

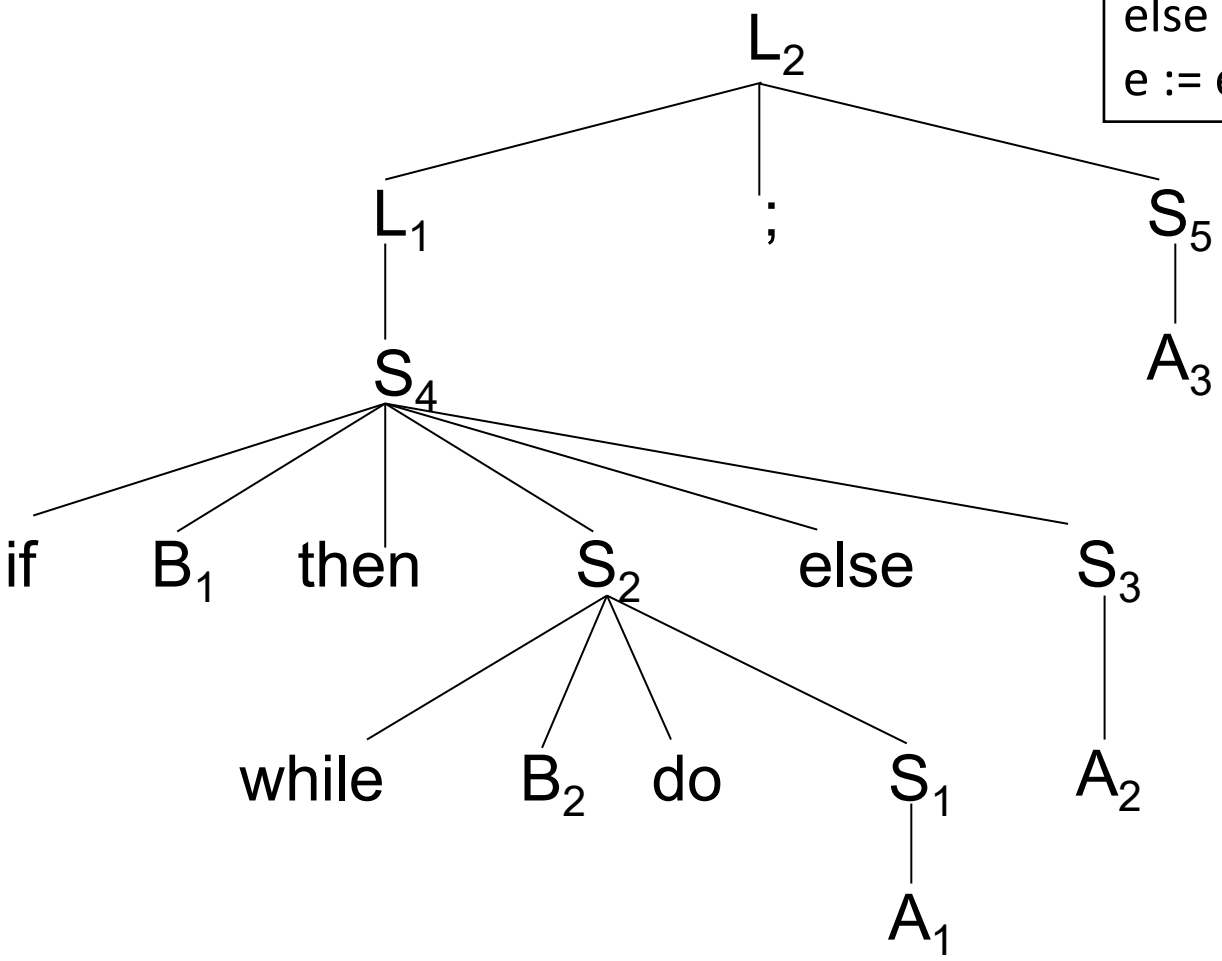
```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



控制流语句文法-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



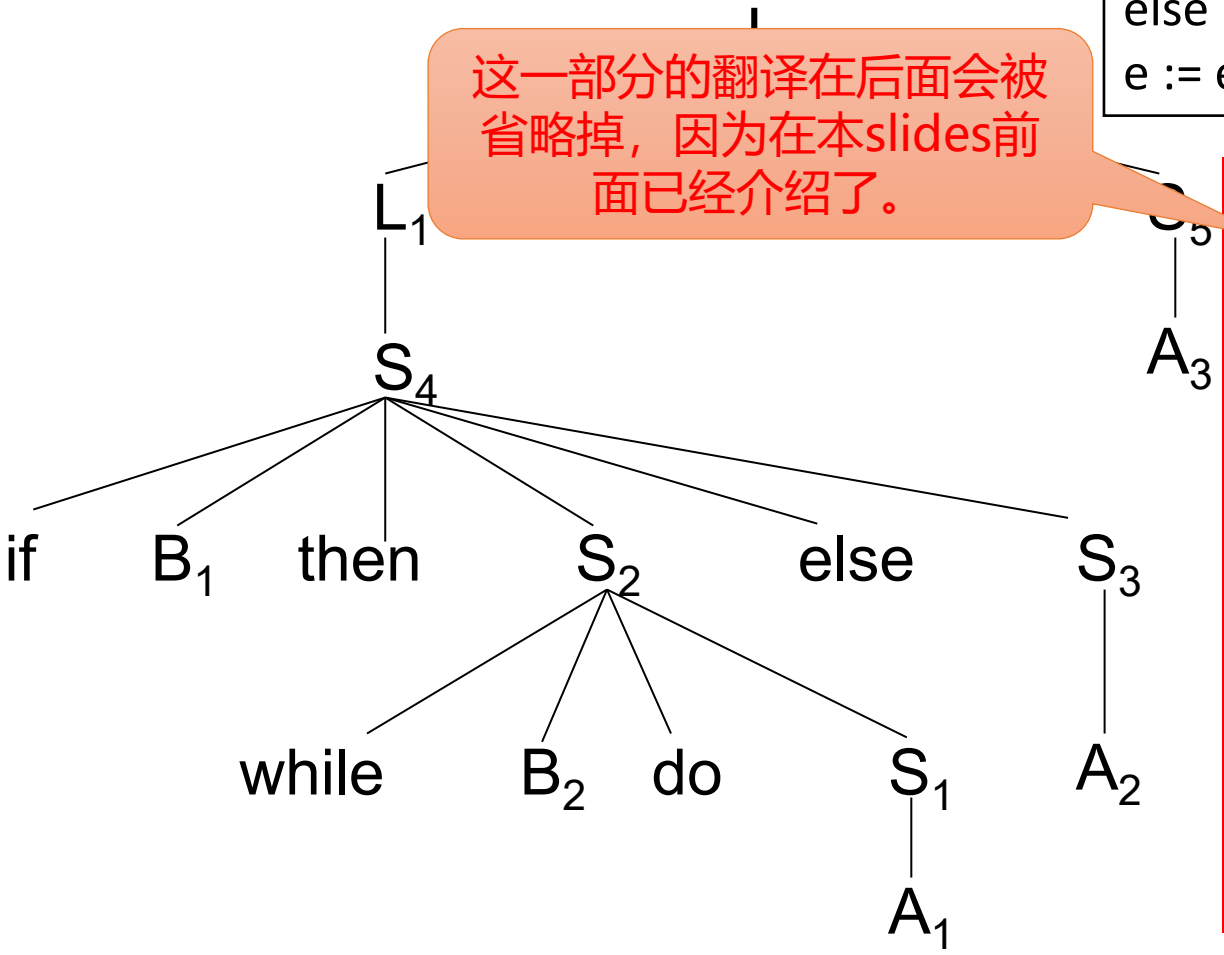
当前的任务：

- **为每一条语句或表达式生成对应的三地址代码**
- **为B_1和B_2创建分别指向真出口和假出口的 truelist 和 falselist**
- **为五个S语句和两个L语句列表创建指向下一指令的 nextlist**
- **在特定的归约环节，将跳转目标指令标号回填对应的列表中的待填指令**
- **按需生成无条件转移指令**

控制流语句文法-例

❑分析树

```
if ( a<b or c<d and e<f )  then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```

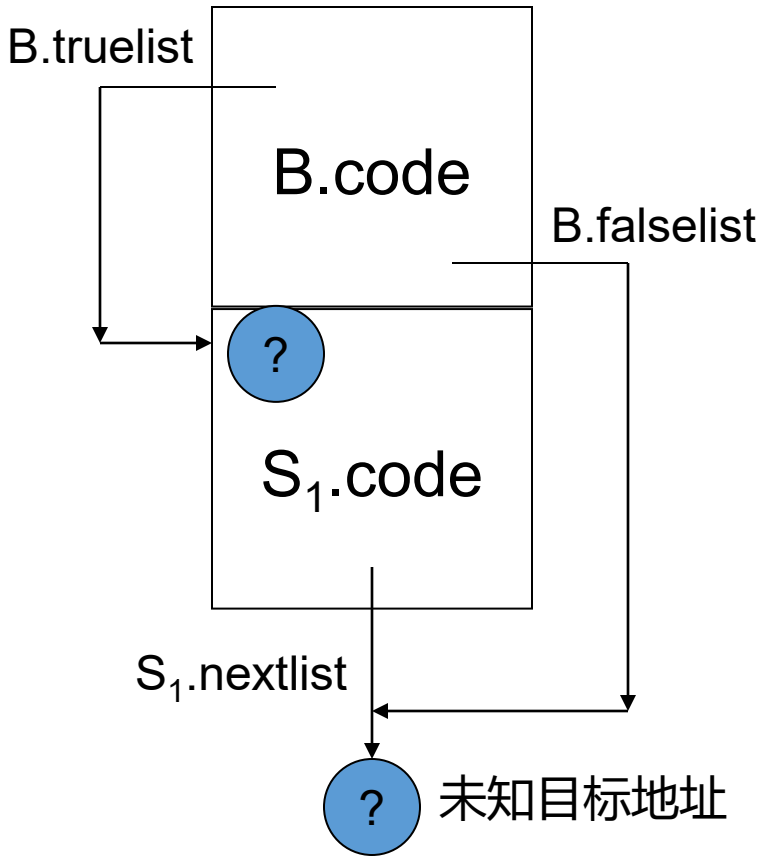


当前的任务：

- ~~为每一条语句或表达式生成对应的三地址代码~~
- 为B_1和B_2创建分别指向真出口和假出口的 truelist 和 falselist
- 为五个S语句和两个L语句列表创建指向下一指令的 nextlist
- 在特定的归约环节，将跳转目标指令标号回填对应的列表中的待填指令
- 按需生成无条件转移指令

条件语句的翻译方案 (1)

if B then S_1 的代码结构

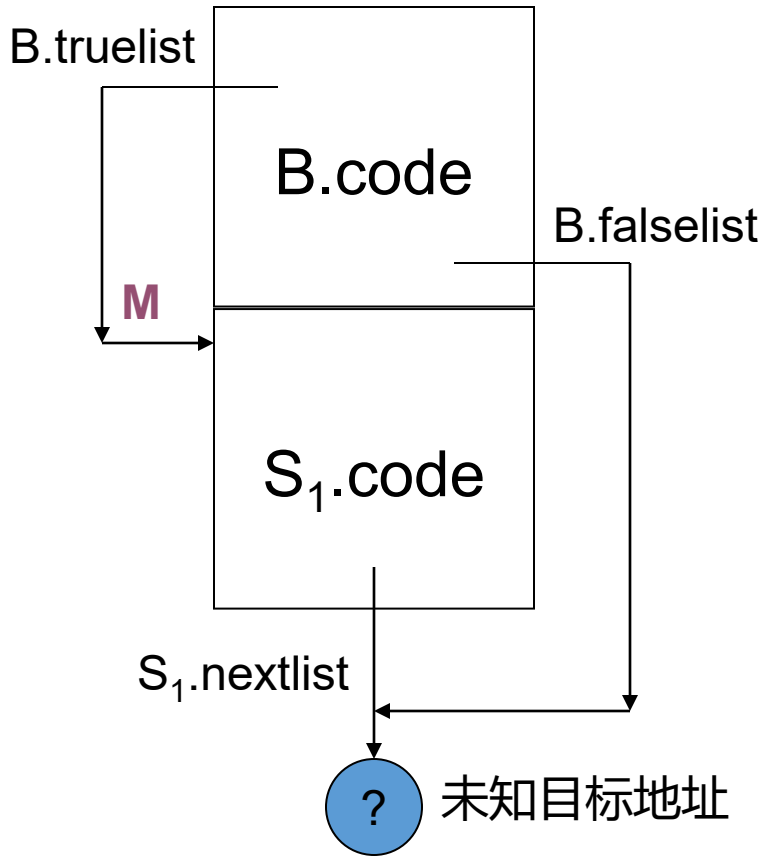


❑四个跳转目标未知

- ❖B的值为真跳转到S_1的开始
- ❖B的值为假/S_1结束/S结束应该跳转到同一个指令

条件语句的翻译方案 (1)

if B then S_1 的代码结构



❑四个跳转目标未知

- ❖B的值为真跳转到S_1的开始
- ❖B的值为假/S_1结束/S结束应该跳转到同一个指令

❑解决方案

- ❖引入M标记，记录B.code之后的下一条新的指令标号，方便回填B.truelist
- ❖将B.falselist、S_1.nextlist合并赋给S.nextlist

条件语句的翻译方案 (1)

$S \rightarrow$ if B then M S_1

{

backpatch(B.truelist, M.instr);

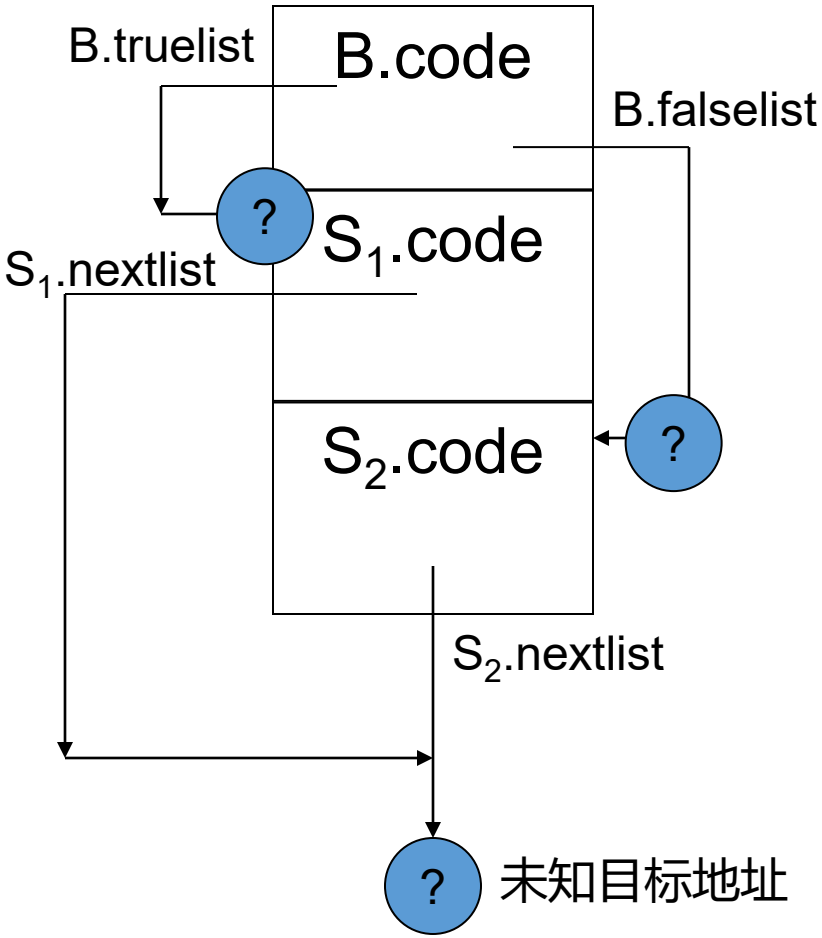
S.nextlist = merge(B.falselist, S_1.nextlist)

}

$M \rightarrow \varepsilon$ { M.instr = nextinstr}

条件语句的翻译方案 (2)

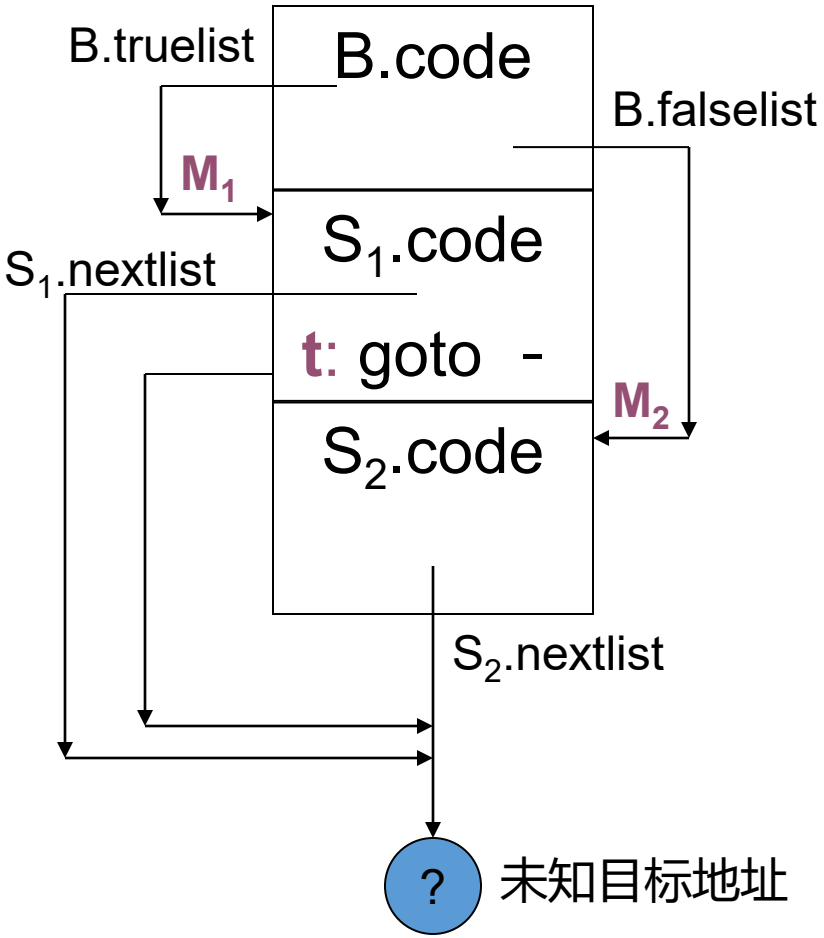
if B then S_1 else S2的结构



- 五个跳转目标未知
  - B的值为真跳转到S_1的开始
  - B的值为假跳转到S_2的开始
  - S_1/ S_2 /S结束应该跳转到同一个指令
- S_1结束要越过S_2

条件语句的翻译方案 (2)

if B then S_1 else S2的结构



- 五个跳转目标未知
  - B的值为真跳转到S_1的开始
  - B的值为假跳转到S_2的开始
  - S_1/ S_2 /S结束应该跳转到同一个指令
- S_1结束要越过S_2
- 解决方案
  - 引入M_1和M_2标记
  - 引入N标记，在S_1后插入无条件跳转指令
  - 将S_1.nextlist、{t}和S_2.nextlist合并赋给S.nextlist

条件语句的翻译方案 (2)

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

{

}

$N \rightarrow \varepsilon$ { N.nextlist = makelist(nextinstr); //标号t

gen(“goto” -);

}

条件语句的翻译方案 (2)

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

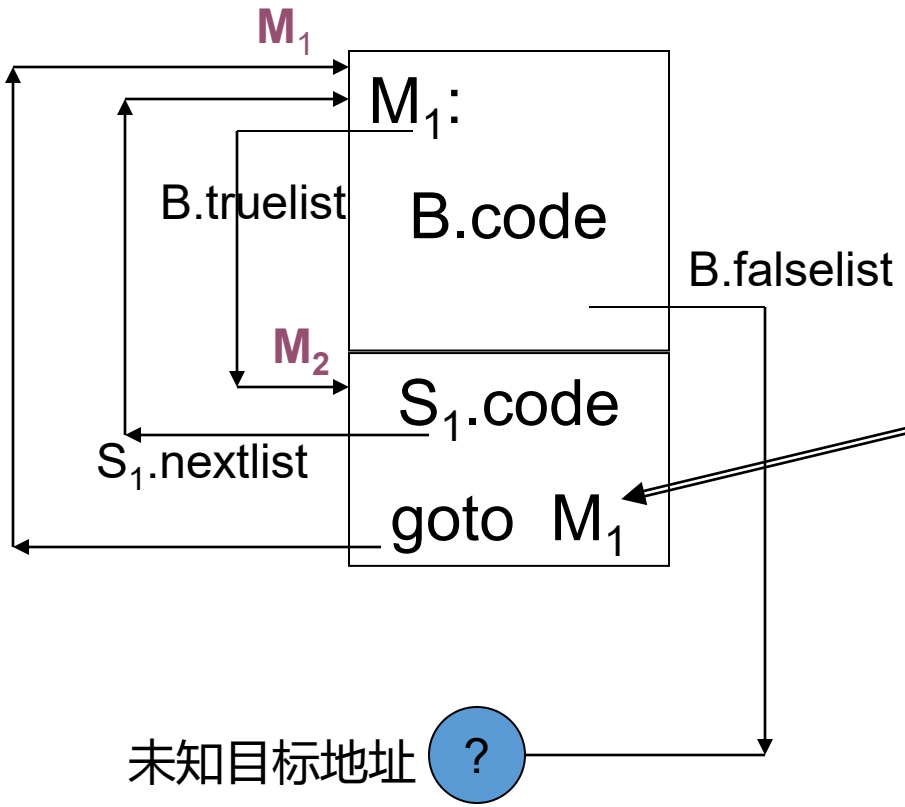
```
{   backpatch( B.truelist,  M1.instr );
    backpatch( B.falselist, M2.instr );
    temp = merge(S1.nextlist, N.nextlist);
    S.nextlist = merge(temp, S2.nextlist) ;
}
```

$N \rightarrow \varepsilon$ { N.nextlist = makelist(nextinstr); //标号t

```
        gen( "goto"  - );
    }
```

循环语句的翻译方案

while B do S_1 的代码结构

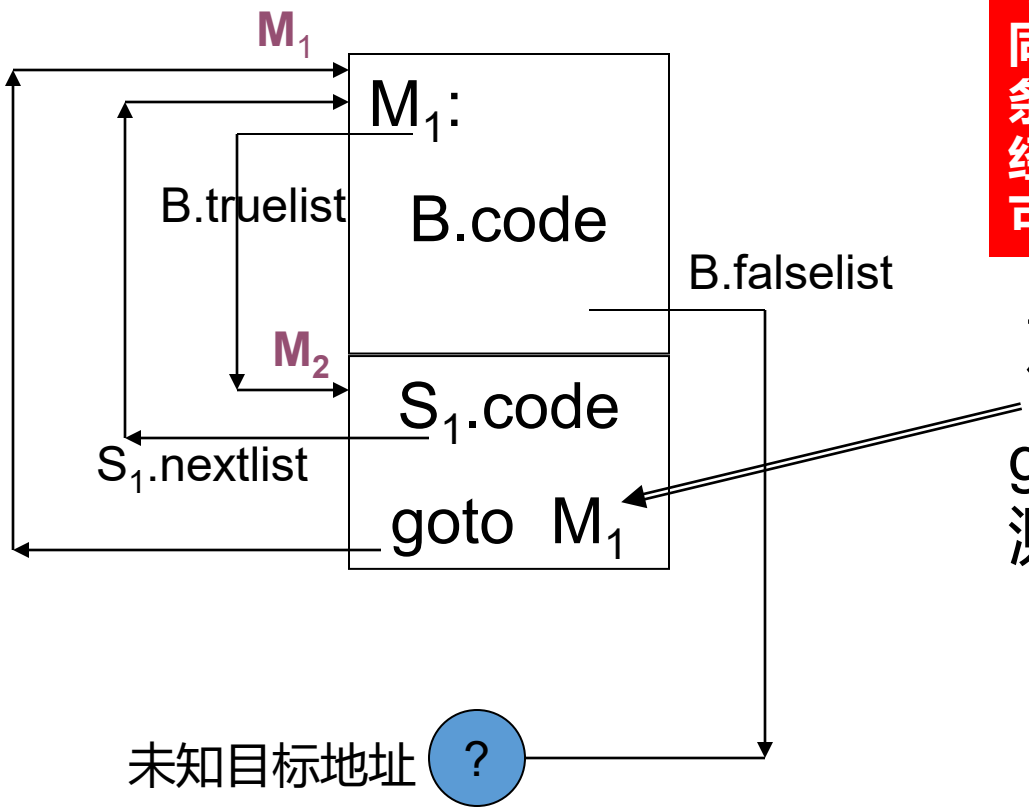


产生无条件跳转指令

goto M_1（跳转至循环条件测试代码开始处）

循环语句的翻译方案

while B do S_1 的代码结构



与 if B then M_1 S_1 N else M_2 S_2不同，此处不用引入N来生成一条无条件跳转指令，因为，S_1分析后已经看到句柄，因此，该指令的生成可以放到归约里。

产生无条件跳转指令

goto M_1（跳转至循环条件测试代码开始处）

循环语句的翻译方案

$S \rightarrow$ while M_1 B do M_2 S_1

{ backpatch(B.truelist, M_2.instr);

backpatch(S_1.nextlist, M_1.instr);

S.nextlist = B.falselist;

gen("goto" M_1.instr);//已知

}

$M \rightarrow \varepsilon$ { M.instr = nextinstr}

简单语句的翻译方案

$S \rightarrow A$ {S.nextlist = {};}

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

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

语句列表的翻译方案

$L \rightarrow L_1;S$

当分析完L_1时，由于不知道S是否会出现，因此需要引入M标记符，记录S的第一条指令；

当S分析完后，进行L的归约，此时，将M记录的指令标号回填L_1.nextlist, 并将S的nextlist赋给L的nextlist

语句列表的翻译方案

$L \rightarrow L_1;S$

当分析完L_1时，由于不知道S是否会出现，因此需要引入M标记符，记录S的第一条指令；

当S分析完后，进行L的归约，此时，将M记录的指令标号回填L_1.nextlist, 并将S的nextlist赋给L的nextlist

$L \rightarrow L_1;M\ S$ {

backpatch(L_1.nextlist, M.instr);

L.nextlist = S.nextlist;

}

$M \rightarrow \varepsilon$ { M.instr = nextinstr}

控制流语句的翻译-例

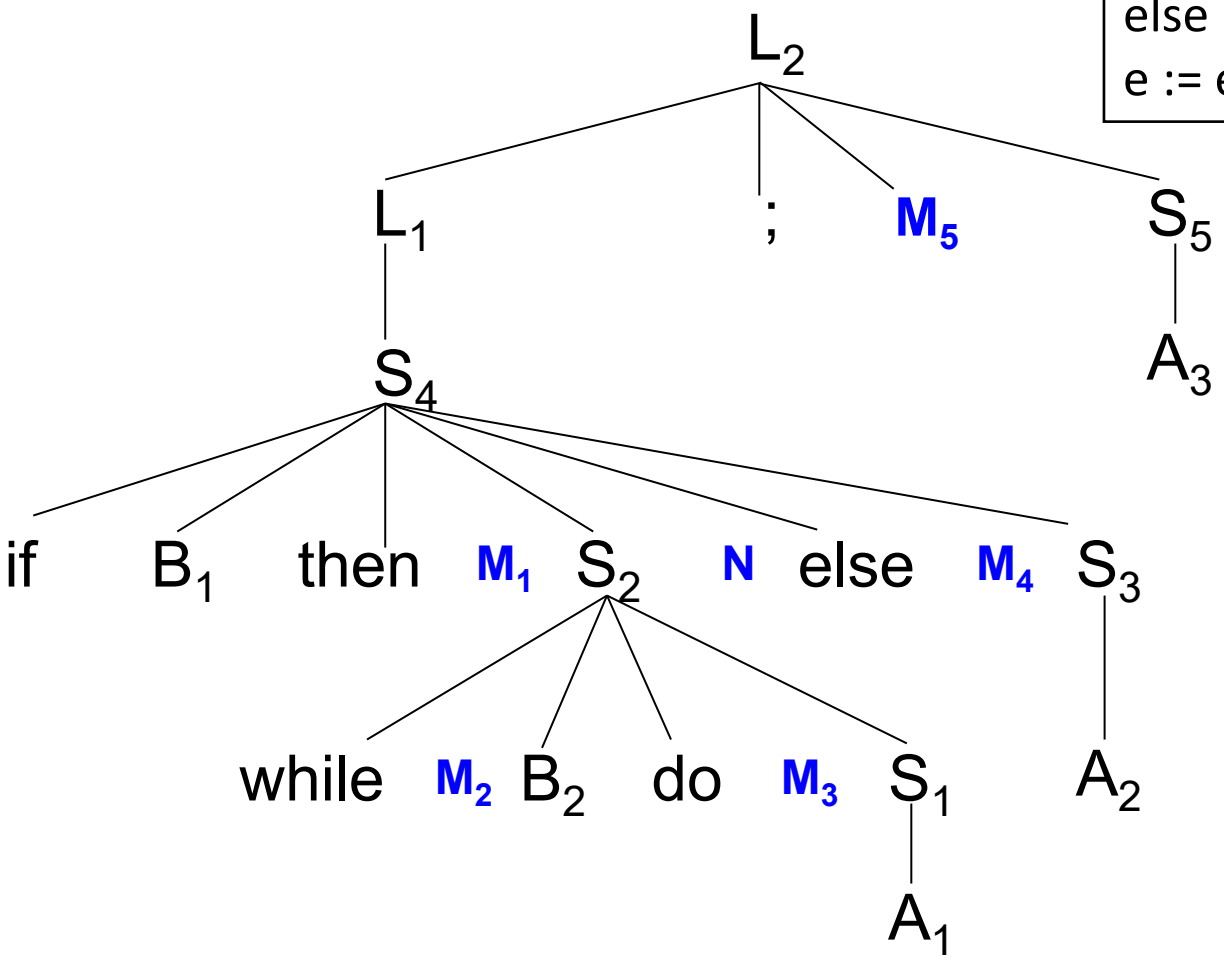
翻译以下语句序列：

```
if ( a<b or c<d and e<f )  then
  while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```

控制流语句的翻译-例

❑分析树

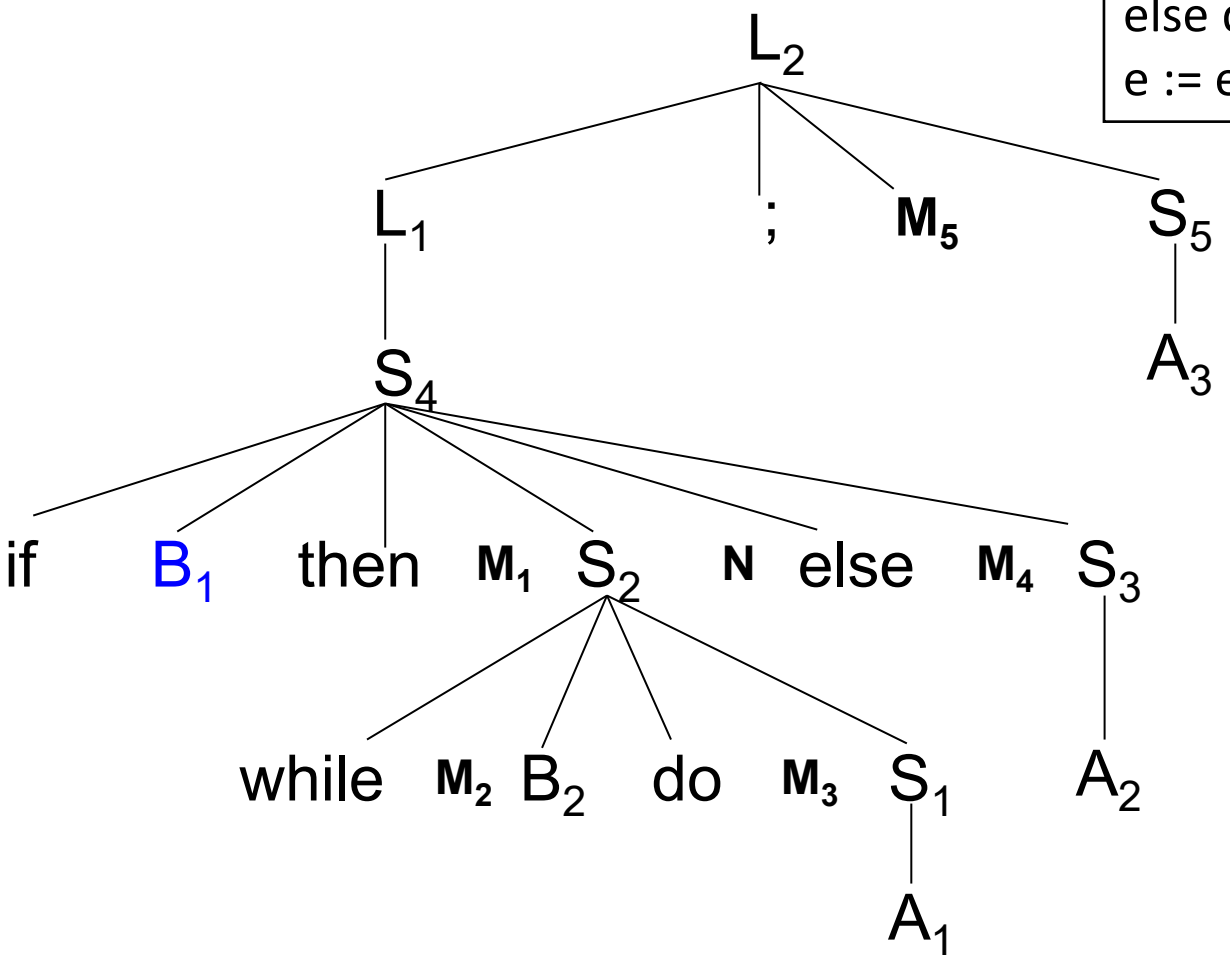
```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

一、翻译 B_1：(a<b or c<d and e<f)

(100) if a<b goto -

(101) goto 102

(102) if c<d goto 104

(103) goto -

(104) if e<f goto -

(105) goto -

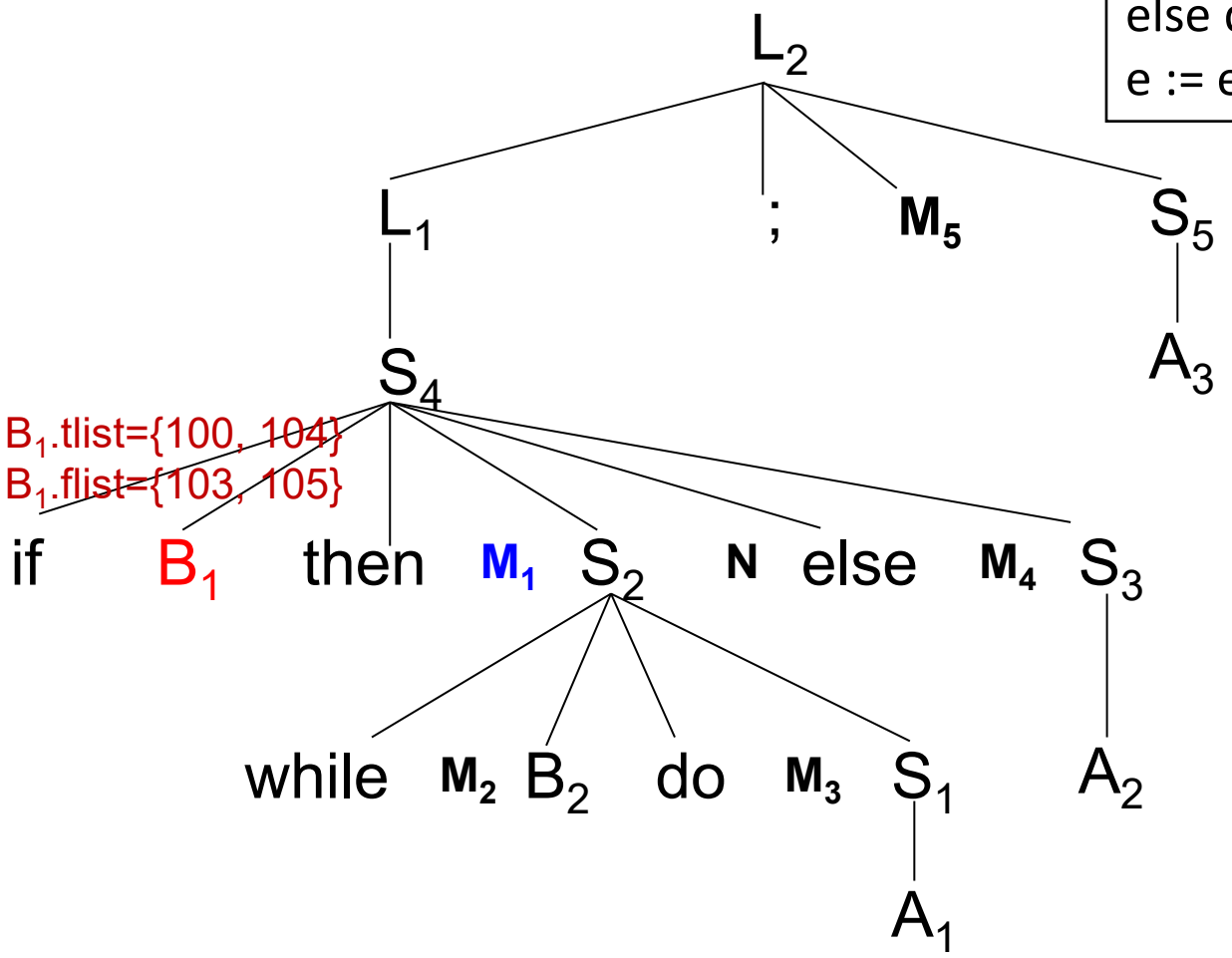
truelist: { 100, 104 } falselist: { 103, 105 }

在69页ppt已经进行了翻译，此处直接使用结果

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

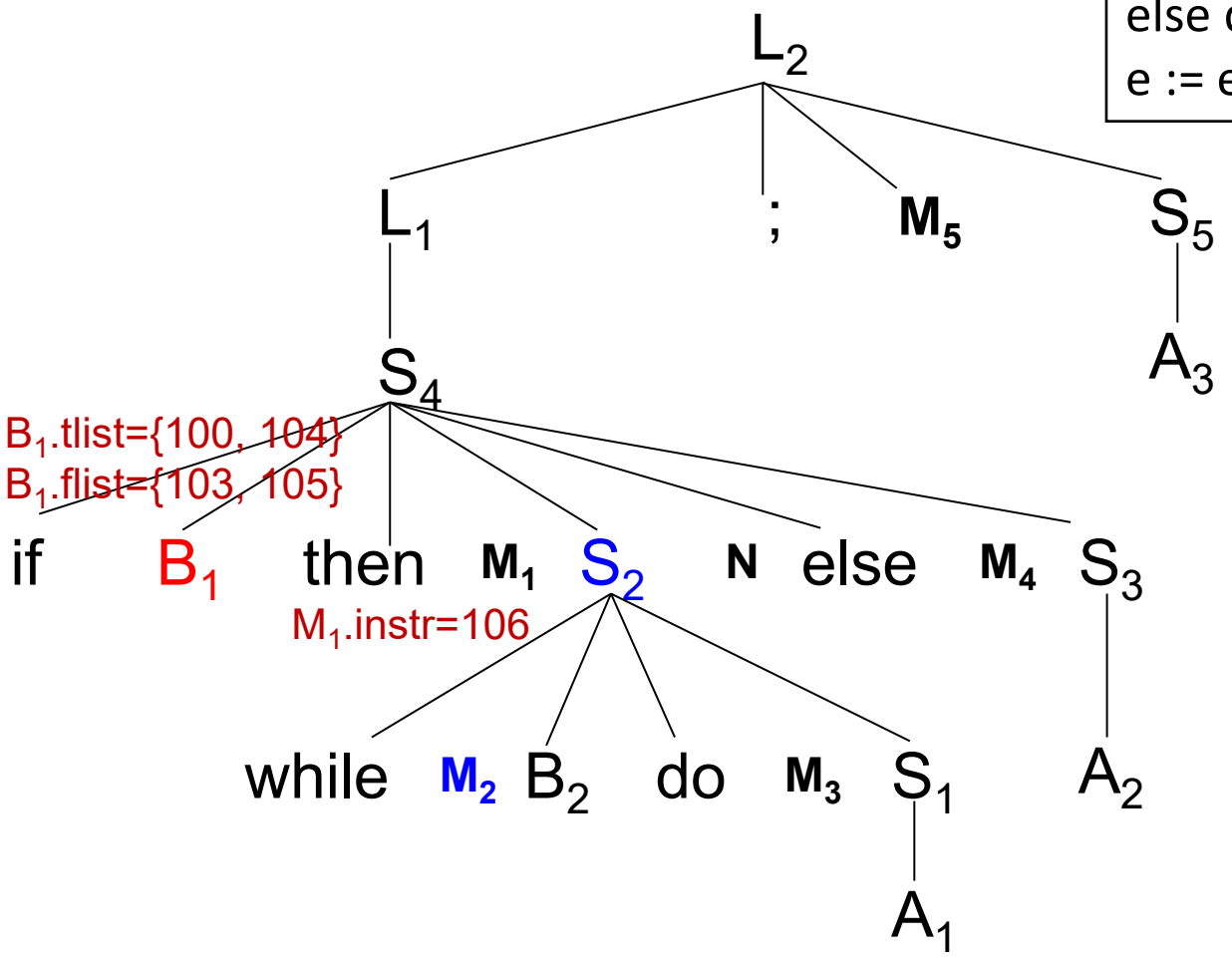
二、翻译 M_1： $M_1 \rightarrow \varepsilon$

记录下一个指令标号106，当if-then-else归约时，用106回填B_1的truelist { 100, 104 }

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

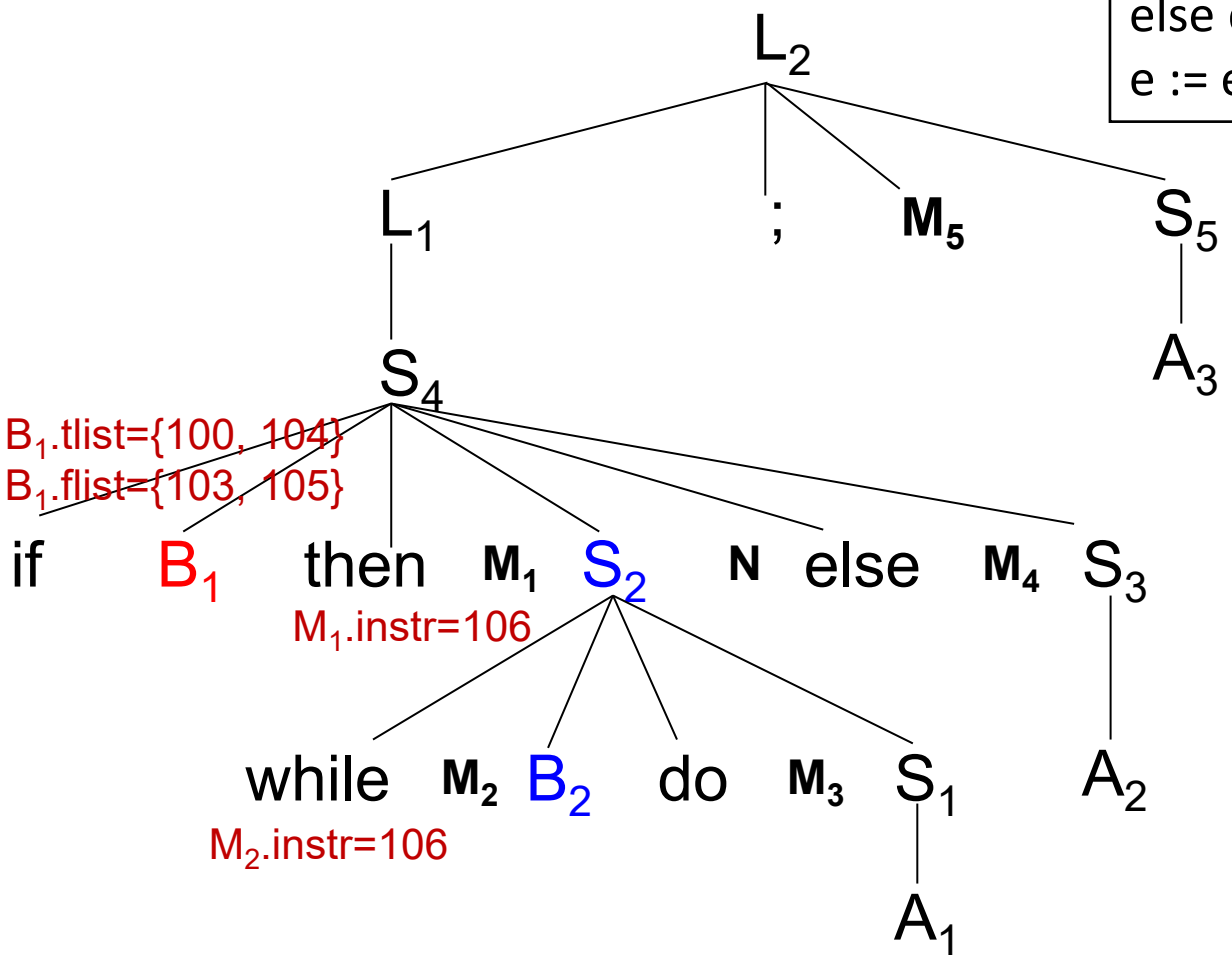
三、翻译 M_2： $M_2 \rightarrow \varepsilon$

记录下一个指令标号106，当while(B_2) S_1归约时，用106回填S_1的nextlist，并生成无条件跳转指令goto M_2.instr

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

四、翻译S_2中B_2： while B_2 do S_1

(106) if a>c goto -

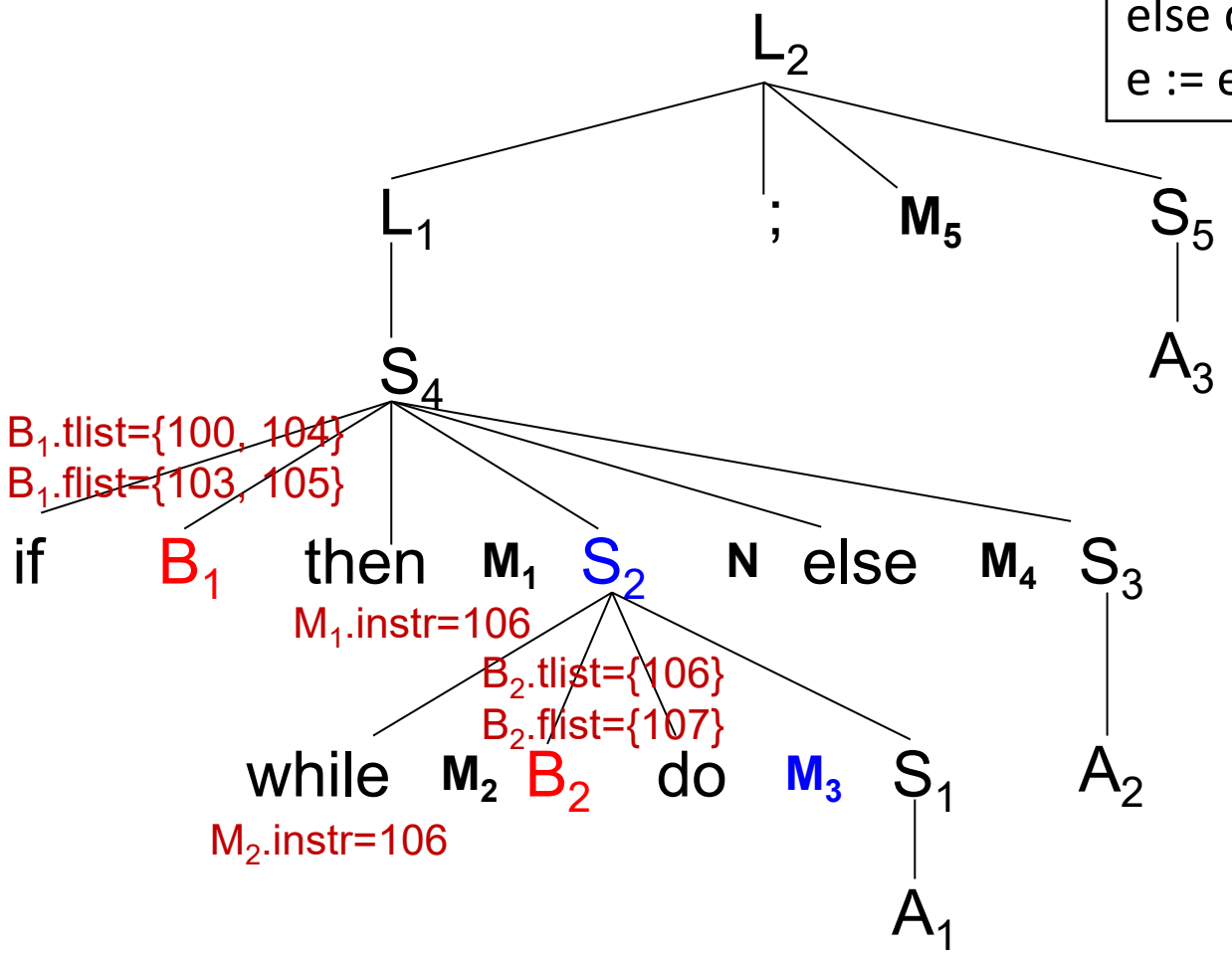
(107) goto -

truelist: { 106 } falselist: { 107 }//待回填

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

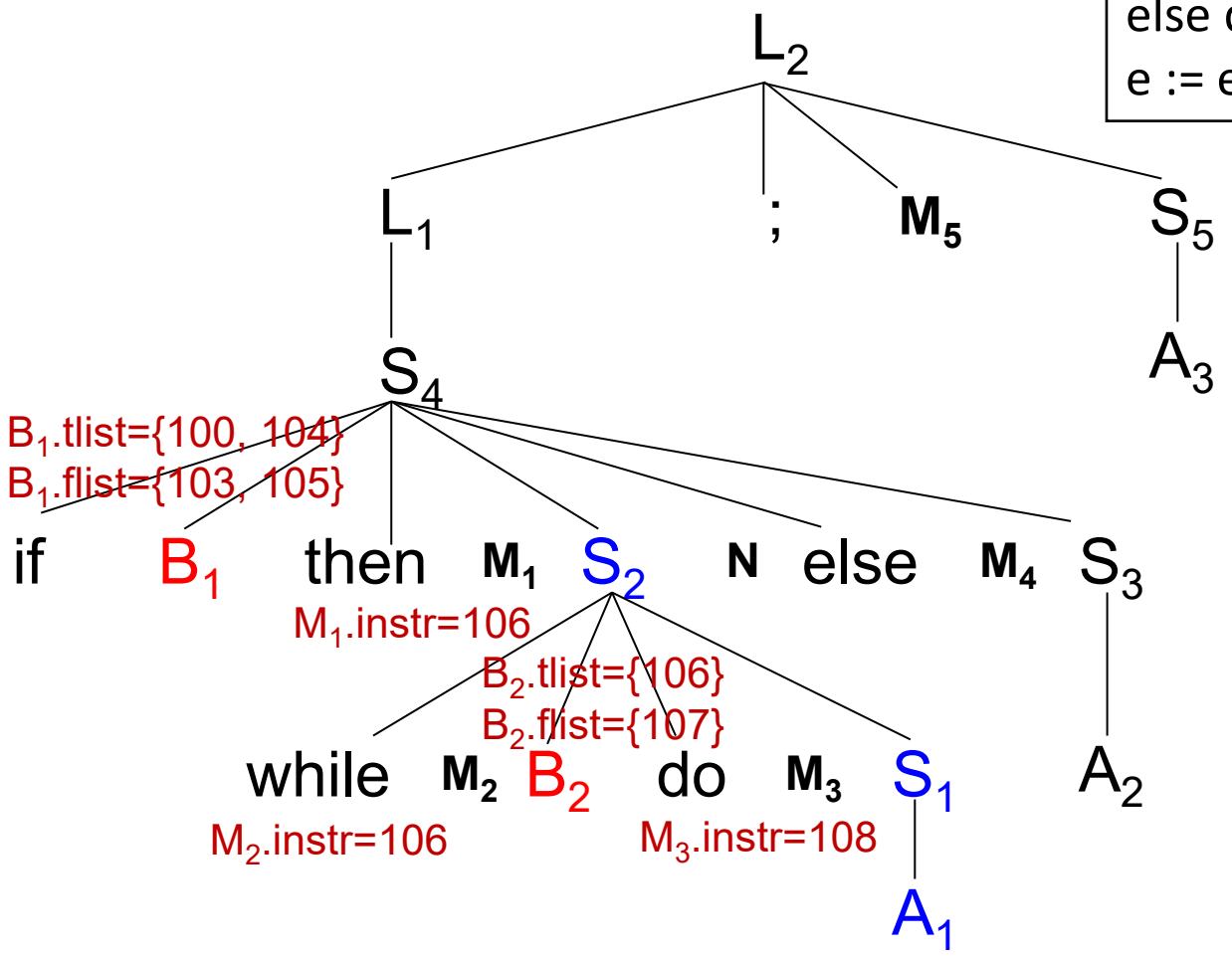
五、翻译 M_3： $M_3 \rightarrow \varepsilon$

记录下一个指令标号108，当while(B_2) S_1归约时，用108回填B_2的truelist

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

六、翻译 S_1： while B_2 do S_1

(106) if a>c goto -

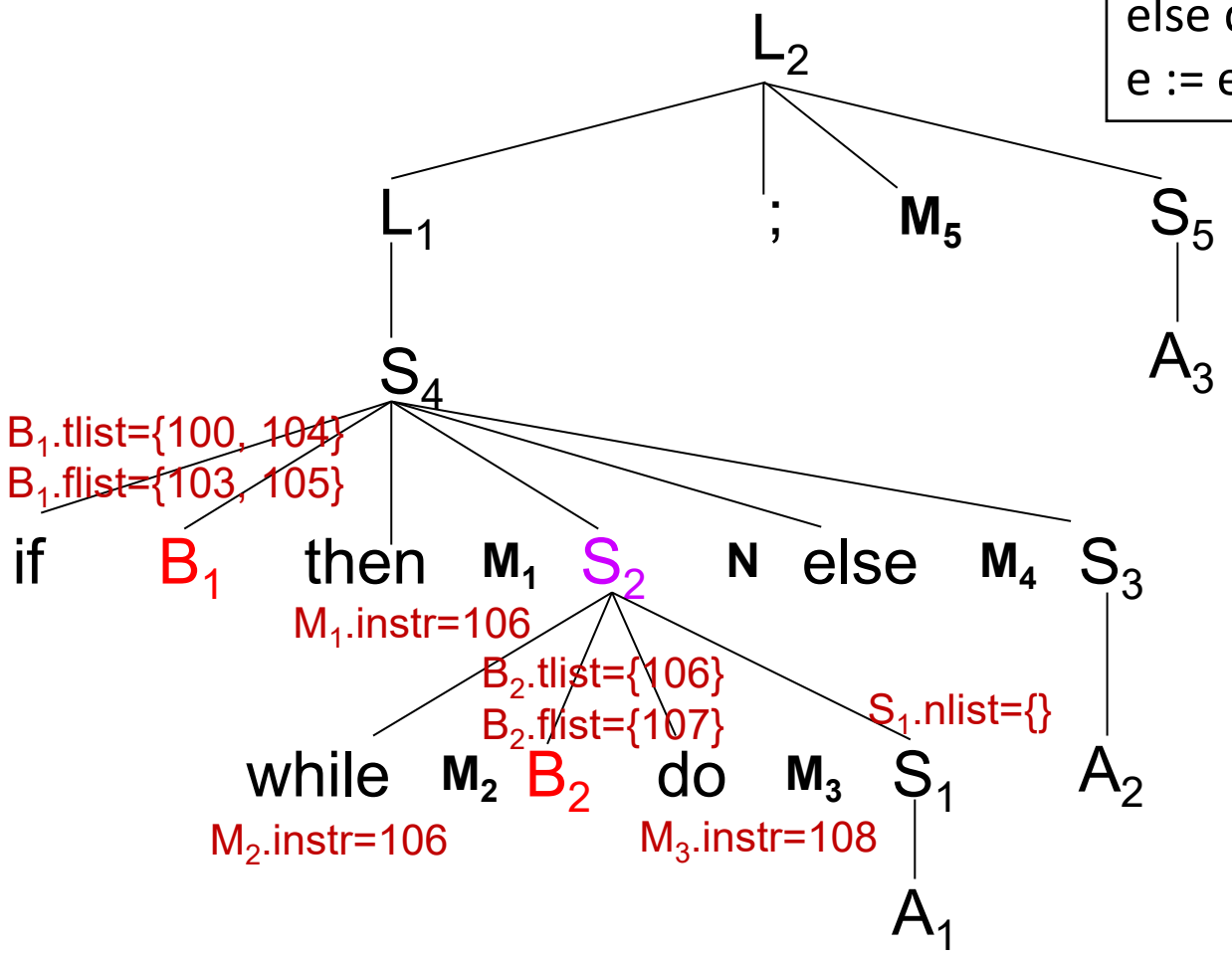
(107) goto -

(108) c := c + 1 // $S_1 \rightarrow A_1$ S_1.nextlist={}

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

控制流语句的翻译-例

七、归约 S_2： while B_2 do S_1

(106) if a>c goto -

(107) goto -

(108) c := c + 1 // $S_1 \rightarrow A_1$ S_1.nextlist={}

控制流语句的翻译-例

七、归约 S_2： while B_2 do S_1

(106) if a>c goto 108 //用108回填106

(107) goto -

(108) c := c + 1 // $S_1 \rightarrow A_1$ S_1.nextlist={}

(109) goto 106 // 转至循环入口(106)

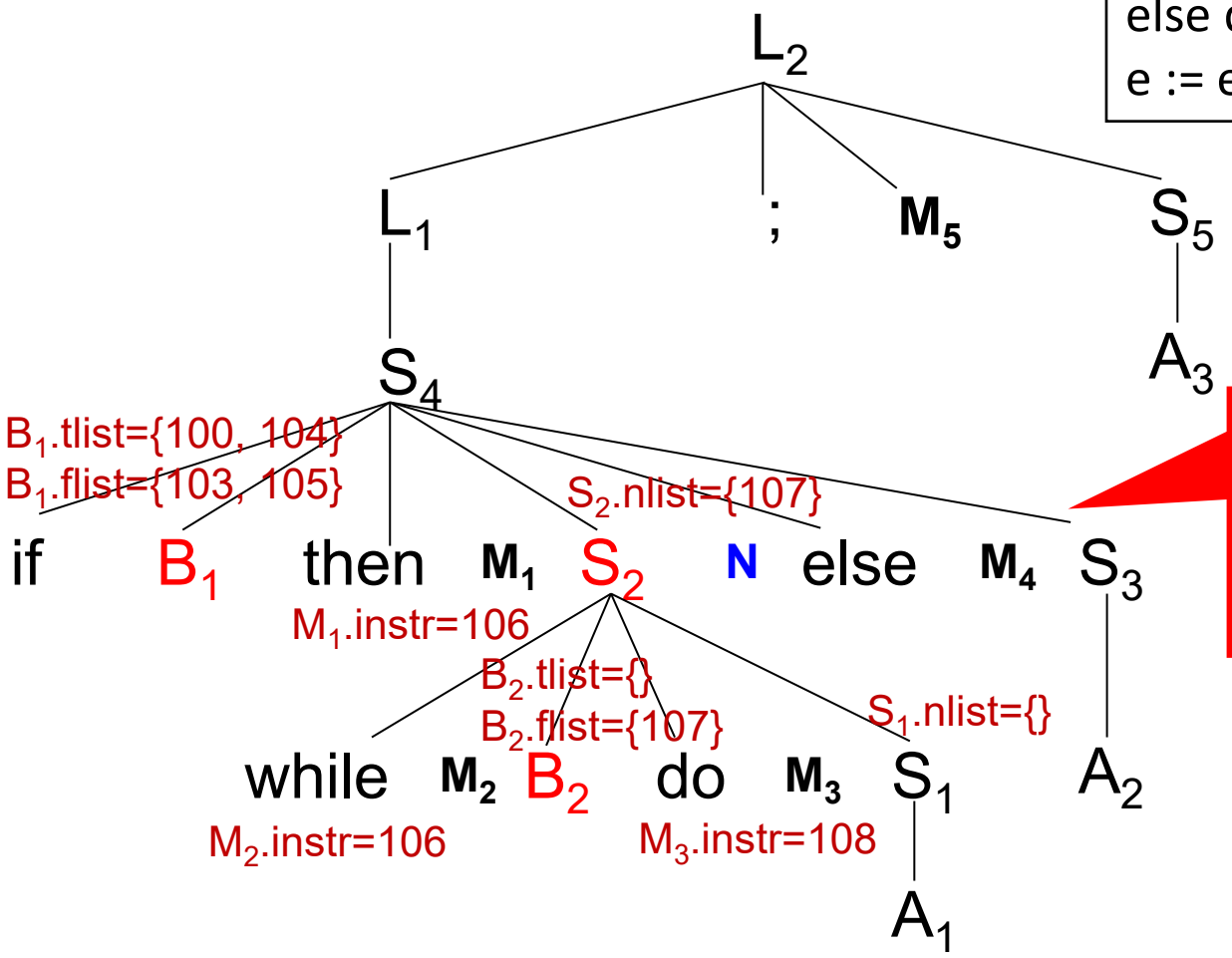
S_2.nextlist: { 107 } //转至循环外部

此处需要用106回填S_1.nextlist，但该list为空

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



虽然while语句归约到S_2，但是S_2的下一跳指令和B_2的假出口未定待填，它们指向相同。

蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

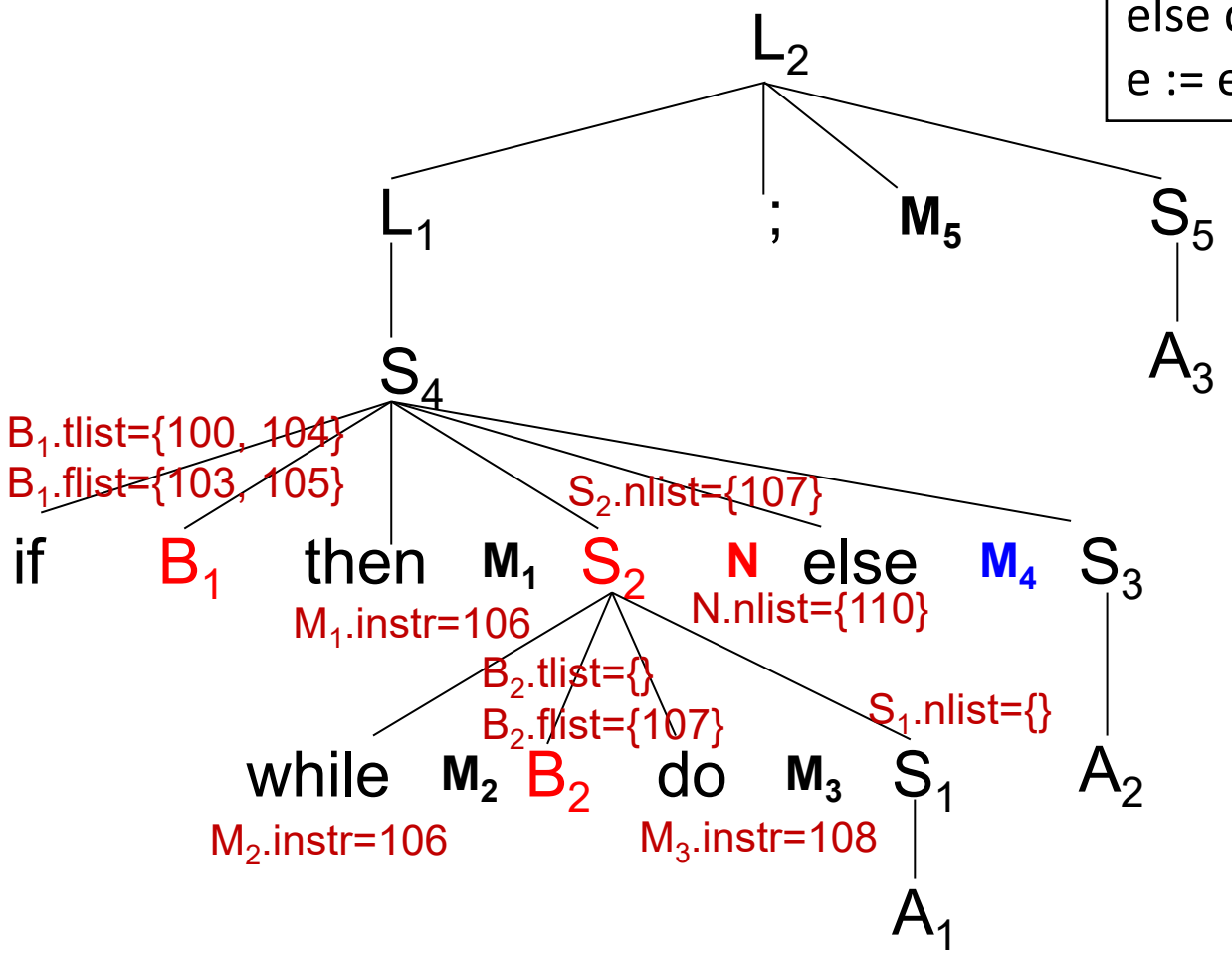
八、翻译 N： $N \rightarrow \varepsilon$

(110) goto - // N.nextlist = {110}

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

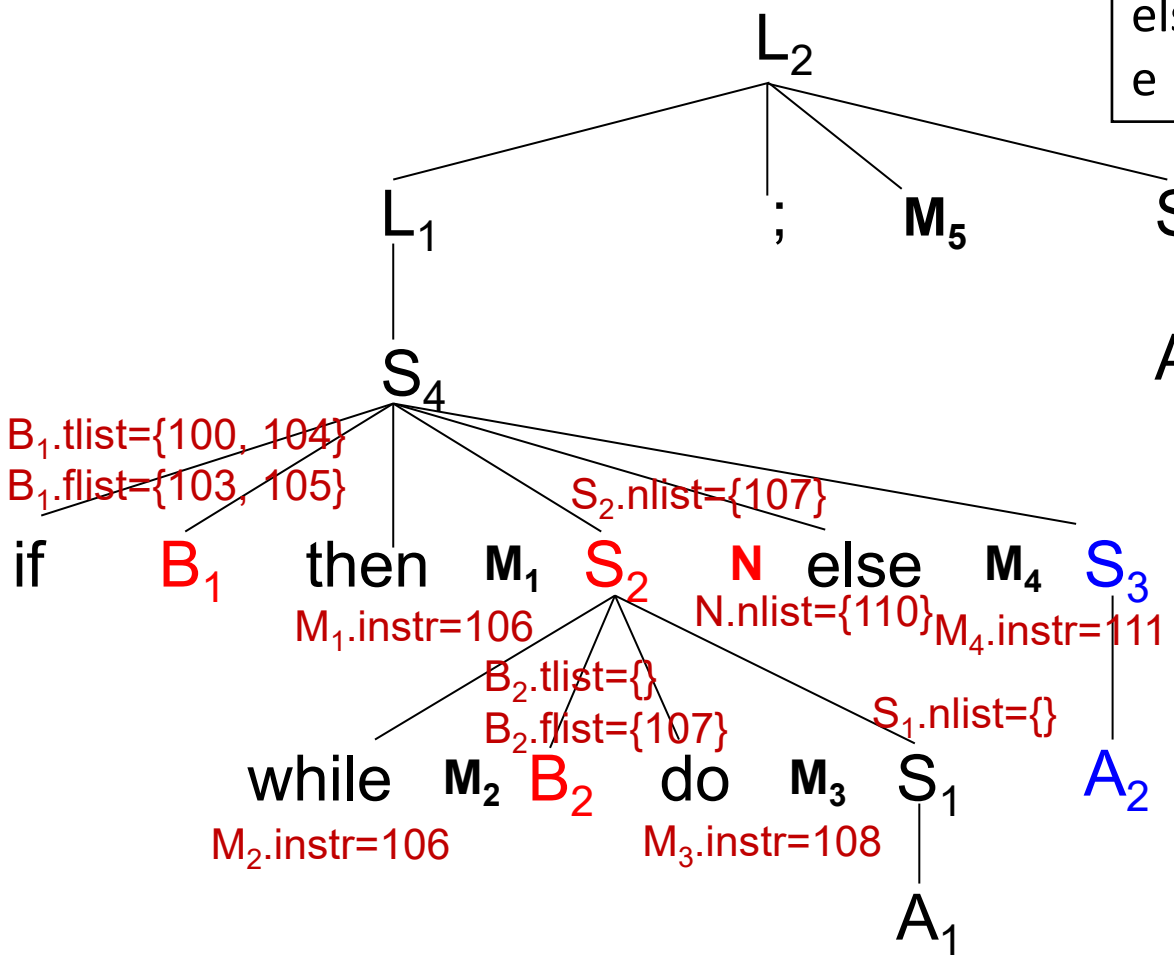
九、翻译 M_4： $M_4 \rightarrow \varepsilon$

记录下一个指令标号111，当if-then-else归约时，用111回填B_1的falselist{103, 105}

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

控制流语句的翻译-例

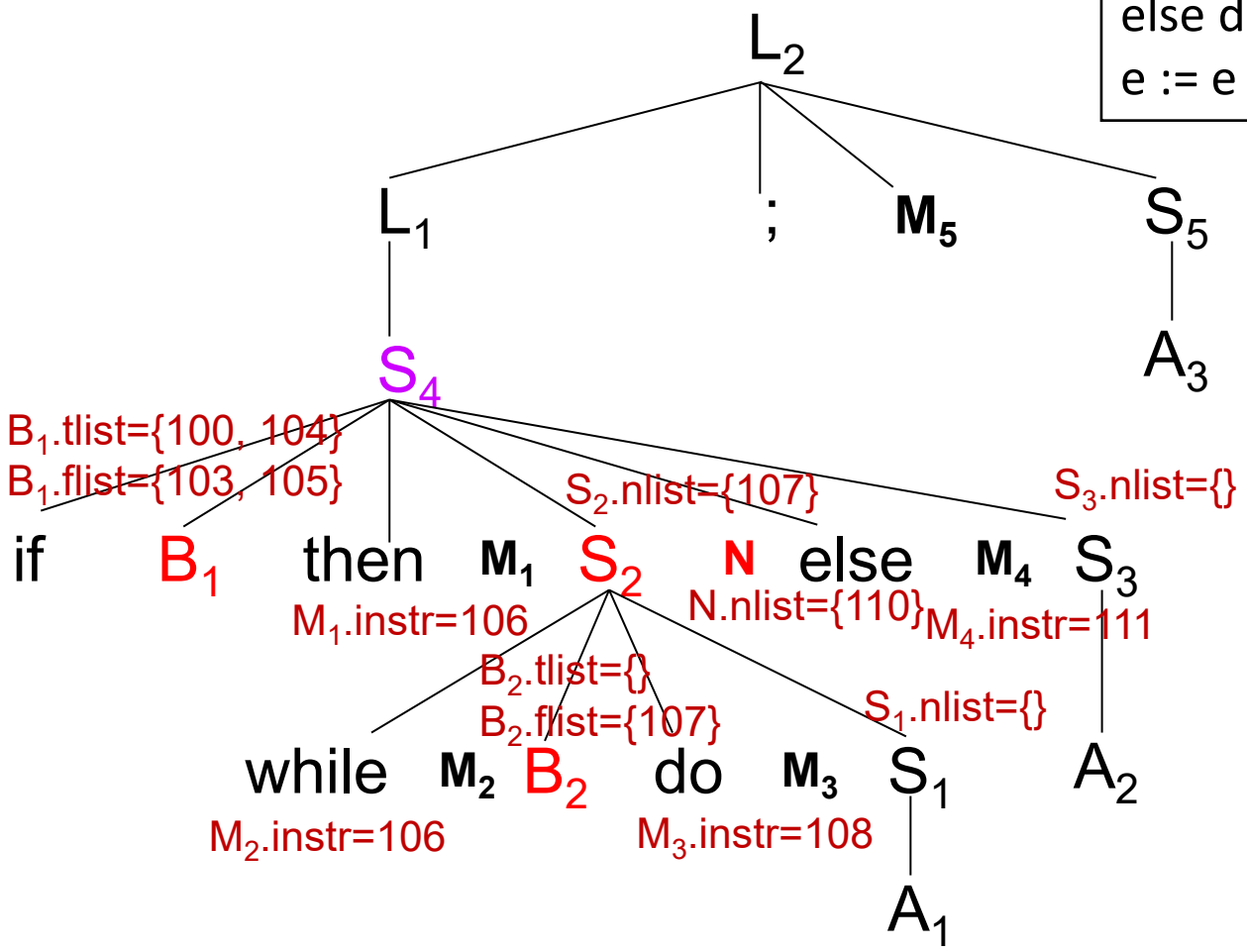
十、翻译 S_3： $S_3 \rightarrow A_2$

(111) d := d + 1 // S_3.nextlist={}

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

控制流语句的翻译-例

中国科学技术大学
University of Science and Technology of China

十一、按if-then-else归约到S_4，进行如下操作

- 用M_1.instr即106回填布尔表达式B_1的truelist{100,104}

控制流语句的翻译-例

(100) if a<b goto 106

(101) goto 102

(102) if c<d goto 104

(103) goto -

(104) if e<f goto 106

(105) goto -

truelist: { 100, 104 } falselist: { 103, 105 }

控制流语句的翻译-例

十一、按if-then-else归约到S_4，进行如下操作

- 用M_1.instr即106回填布尔表达式B_1的truelist{100,104}
- 用M_4.instr即111回填布尔表达式B_1的falselist{103,105}

控制流语句的翻译-例

(100) if a<b goto 106

(101) goto 102

(102) if c<d goto 104

(103) goto 111

(104) if e<f goto 106

(105) goto 111

truelist: { 100, 104 } falselist: { 103, 105 }

控制流语句的翻译-例

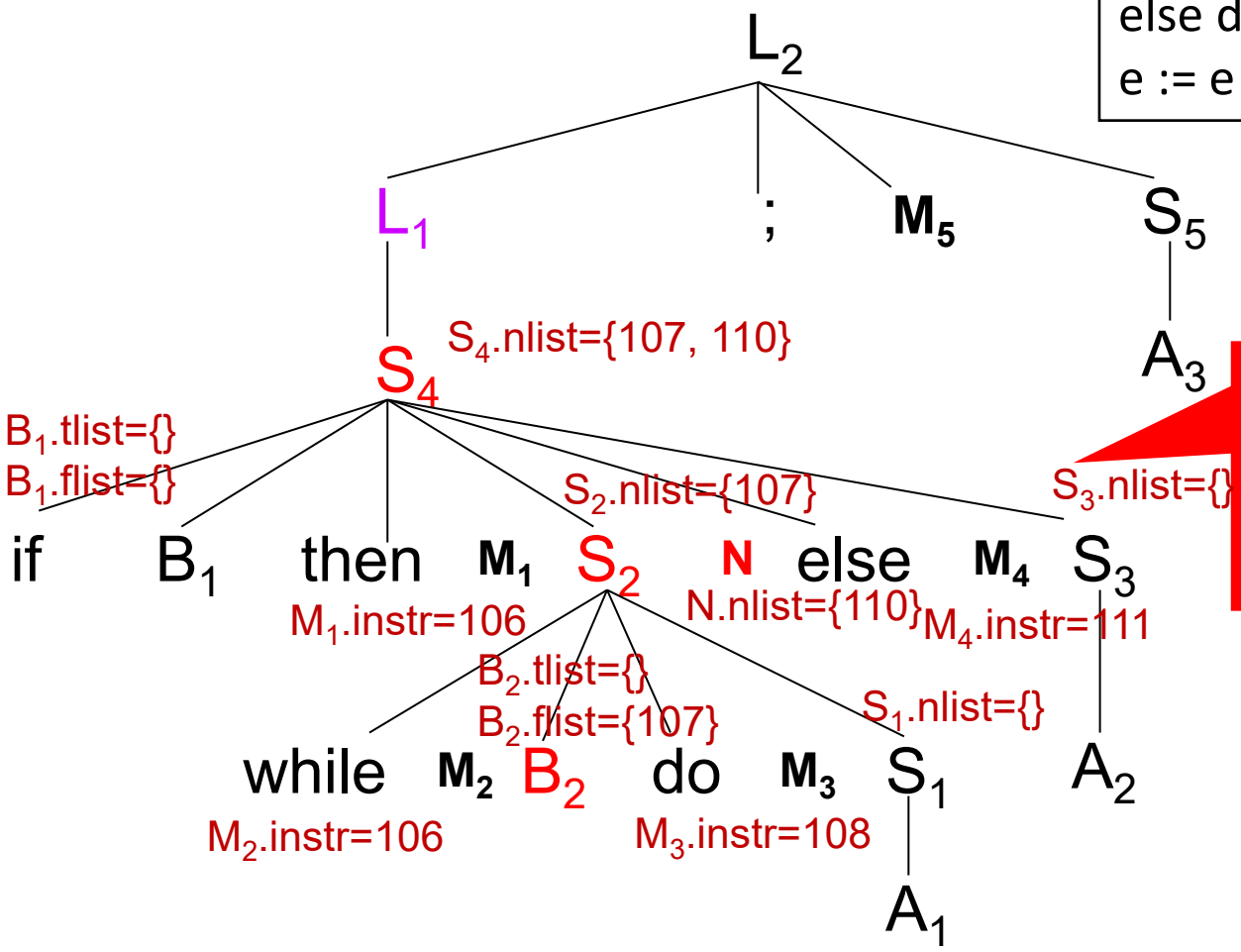
十一、按if-then-else归约到S_4，进行如下操作

- 用M_1.instr即106回填布尔表达式B_1的truelist{100,104}
- 用M_4.instr即111回填布尔表达式B_1的falselist{103,105}
- S_4.nextlist等于S_2、N和S_3的nextlist的并集，即为{107, 110}

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



S_4的下一跳指令、S_2的下一跳指令和B_2的假出口未定待填，它们指向相同。

蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

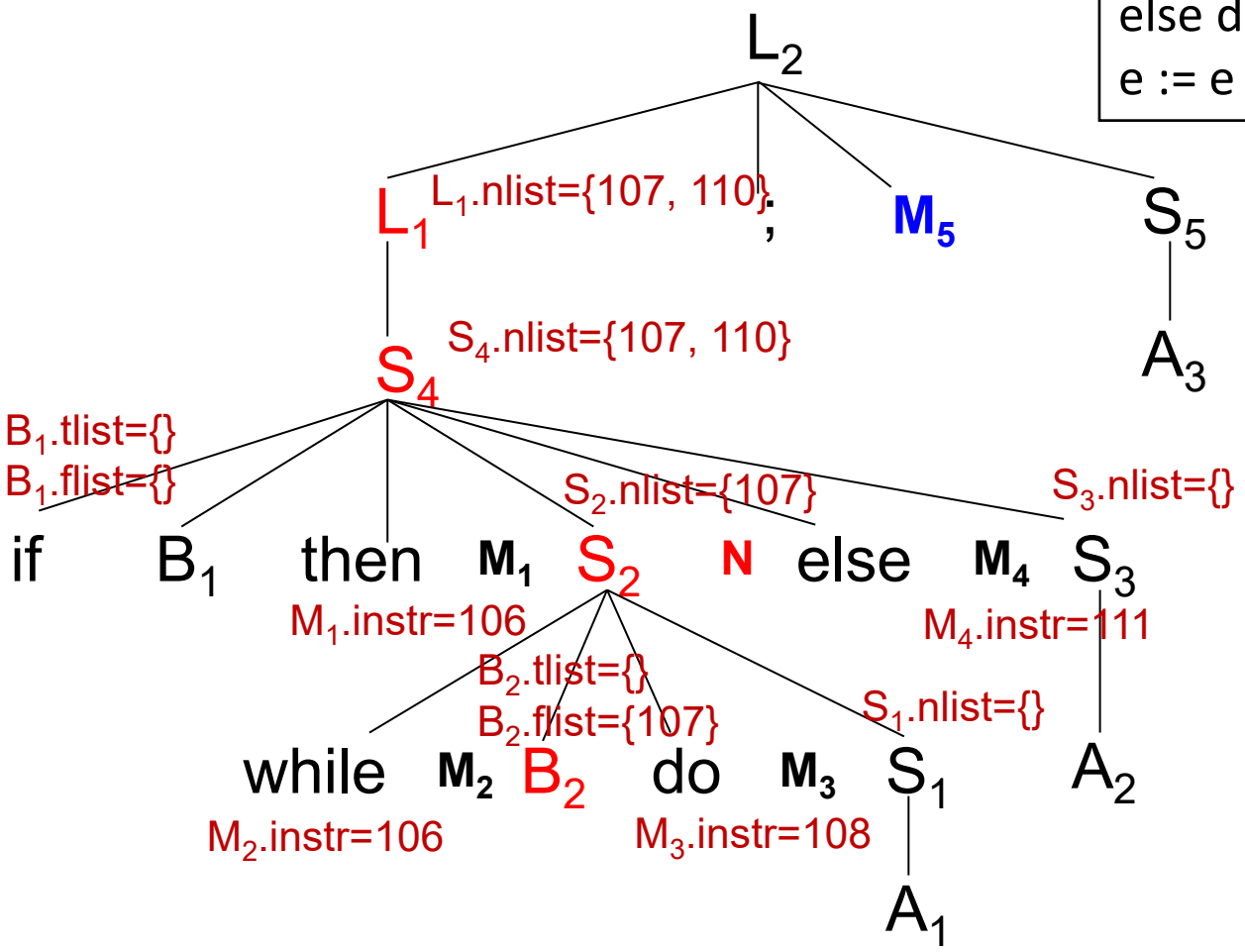
十二、归约L_1，将S_4.nextlist直接赋给L_1.nextlist

- L_1.nextlist: { 107, 110 }

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

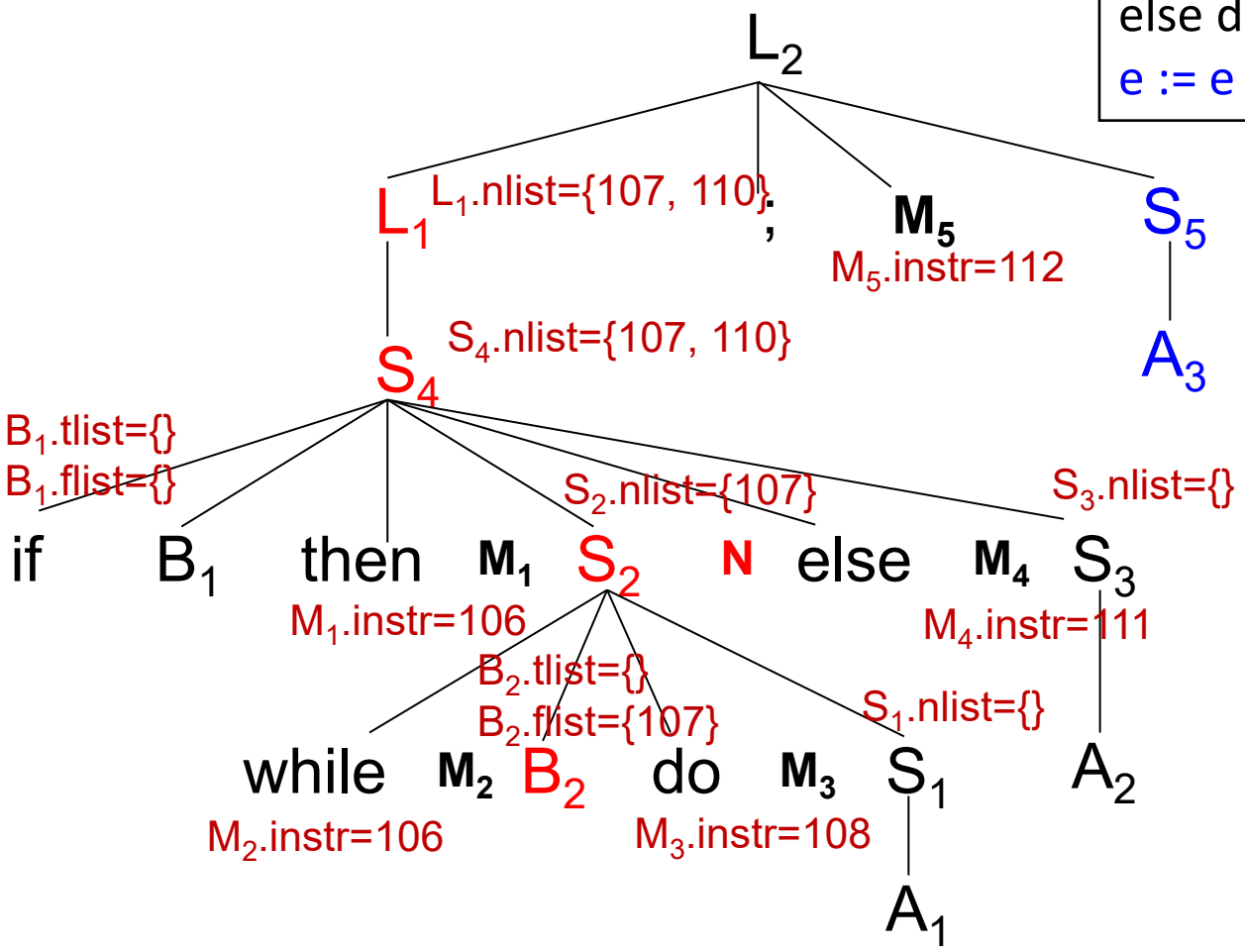
十三、翻译M_5： $M_5 \rightarrow \varepsilon$

记录下一个指令标号112，当L_1;S归约时，用112回填L_1的nextlist{107, 110}

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填

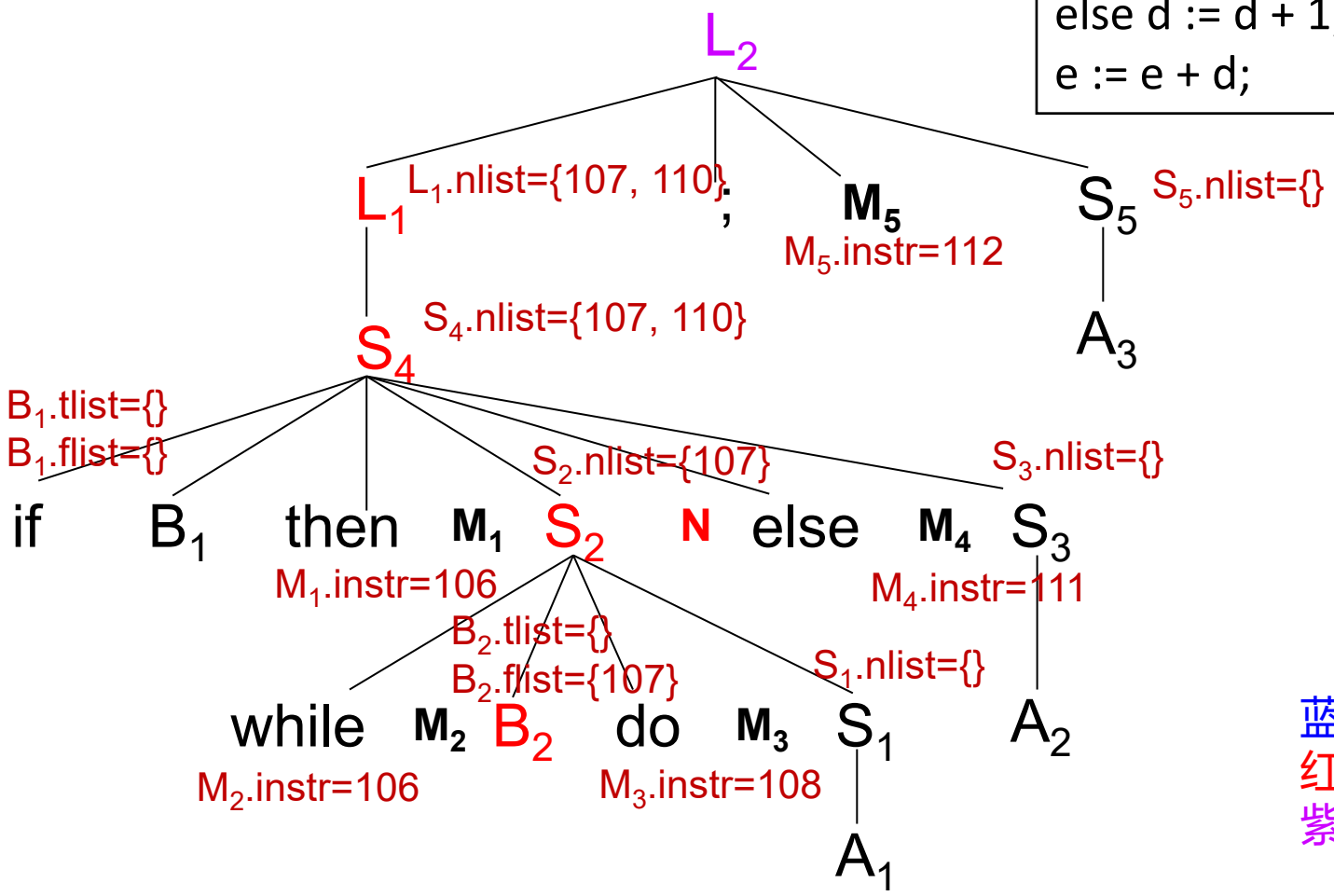
十四、翻译S_5

(112) e := e + d // $S_5 \rightarrow A_3$ S_5.nextlist={}

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

十五、归约L_2

- 用M_5.instr即112回填L_1的nextlist{107，110}
- L_2.nextlist = S_5.nextlist，所以为空

控制流语句的翻译-例

(100) if a<b goto 106

(101) goto 102

(102) if c<d goto 104

(103) goto 111

(104) if e<f goto 106

(105) goto 111

(106) if a>c goto 108

(107) goto -

(108) c := c + 1

(109) goto 106

(110) goto -

(111) d := d + 1

(112) e := e + d

控制流语句的翻译-例-终

(100) if a<b goto 106

(101) goto 102

(102) if c<d goto 104

(103) goto 111

(104) if e<f goto 106

(105) goto 111

(106) if a>c goto 108

(107) goto 112

(108) c := c + 1

(109) goto 106

(110) goto 112

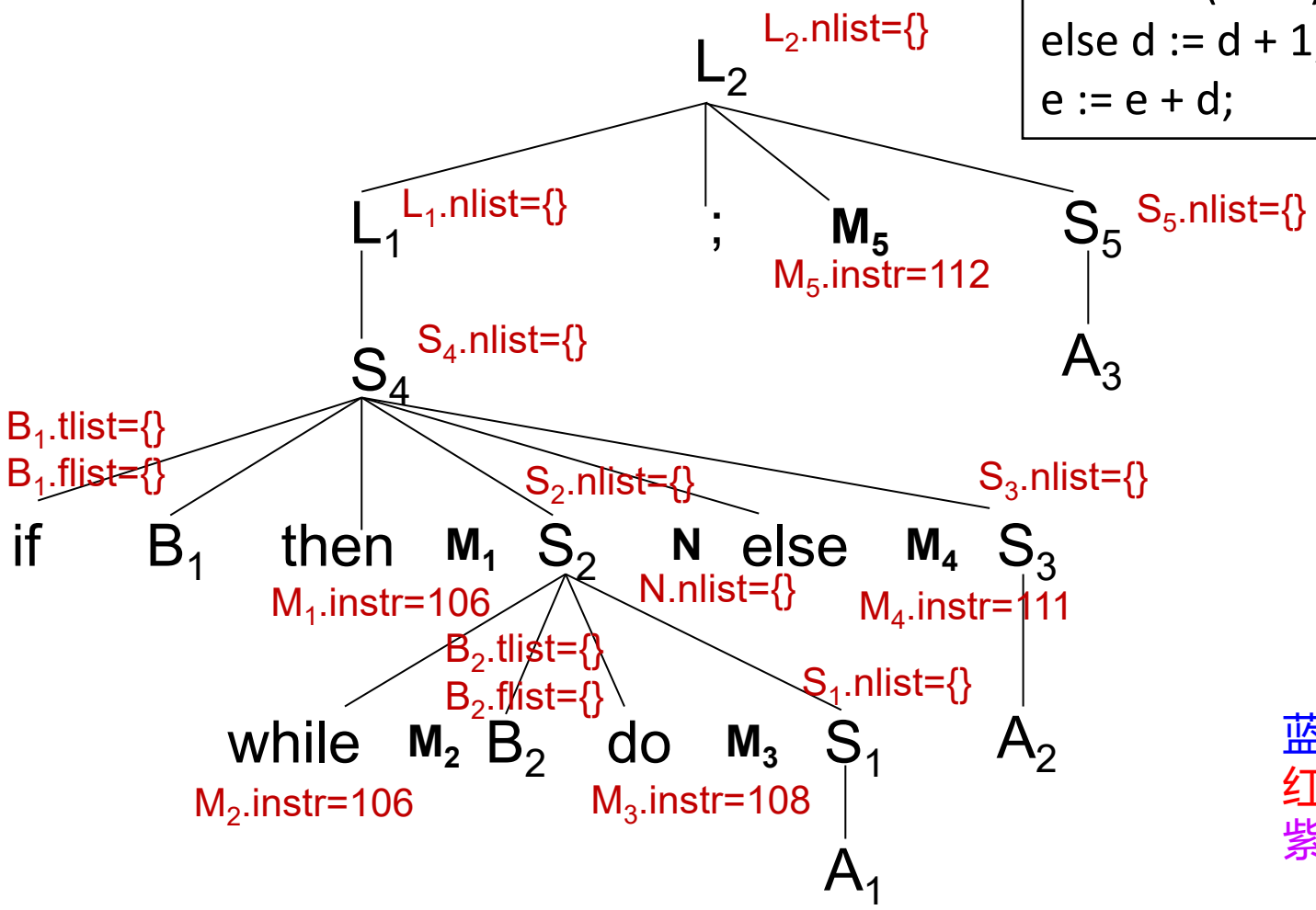
(111) d := d + 1

(112) e := e + d

控制流语句的翻译-例

❑分析树

```
if ( a<b or c<d and e<f ) then
   while ( a>c ) do c := c +1
else d := d + 1;
e := e + d;
```



蓝色表示即将翻译
红色表示需要回填
紫色表示即将归约

中间代码生成

- 类型检查
- 声明语句的翻译
- 记录或结构体的翻译
- 数组寻址的翻译
- 表达式的翻译
- 控制流语句的翻译
- switch语句的翻译
- 过程或函数的翻译

涉及到类型表达式以及内存组织形式，因此，单独设计一个章节来讲

switch语句的文法

switch (E){ //E是一个选择表达式

case V_1: S_1// V是常量，S是语句

case V_2: S_2

. . .

case V_{n-1}: S_{n-1}

default: S_n

}

switch语句的文法

switch (E){ //E是一个选择表达式

case V_1: S_1// V是常量，S是语句

case V_2: S_2

. . .

case V_{n-1}: S_{n-1}

default: S_n

}

三地址代码形态：
- 计算E的值的代码
- S_j语句的代码
- 匹配E的值与V_j，并执行对应S_j的逻辑

switch语句的翻译

❑分支数较少时

```
        E的代码
        t = E.place                 | L_{n-2}: if t != V_{n-1} goto L_{n-1}
        if t != V_1 goto L_1        |          S_{n-1}的代码
        S_1的代码                   |          goto next
        goto next                   | L_{n-1}: S_n的代码
L_1:    if t != V_2 goto L_2        | next:
        S_2的代码
        goto next
L_2:    . . .
        . . .
```

switch语句的翻译

- 分支较多时，将分支测试代码集中在一起，便于生成较好的分支测试代码

```
        t = E.place       | L_n: S_n的代码
        goto test         |      goto next
L_1:    S_1的代码         |test: if t == V_1 goto L_1
        goto next         |      if t == V_2 goto L_2
L_2:    S_2的代码         |      . . .
        goto next         |      if t == V_{n-1} goto L_{n-1}
        . . .             |      goto L_n
L_{n-1}: S_{n-1}的代码    | next:
        goto next
```

switch语句的翻译

- 中间代码表示增加一种case语句，与之前的翻译等价，便于代码生成器对它进行特别处理

```
test:  case t V1    L1
       case t V2    L2
       . . .
       case t Vn-1  Ln-1
       case t t     Ln
next:
```

中间代码生成

- 类型检查
- 声明语句的翻译
- 记录或结构体的翻译
- 数组寻址的翻译
- 表达式的翻译
- 控制流语句的翻译
- switch语句的翻译
- **过程或函数的翻译**

涉及到类型表达式以及内存组织形式，因此，单独设计一个章节来讲

过程调用的翻译

$S \rightarrow$ call id (*Elist*)

$Elist \rightarrow Elist, E$

$Elist \rightarrow E$

此处先忽略函数及过程的定义，会在后面的课程中讲解。

过程调用的翻译

- 过程调用id(E_1, E_2, …, E_n)的中间代码结构

E_1的代码
E_2的代码
. . .
E_n的代码
param E_1.*place*
param E_2.*place*
. . .
param E_n.*place*
call id.*place*, *n*

过程调用的翻译方案

为*Elist*设计一个综合属性paramlist，该列表记录函数调用的所有参数，且参数排列顺序与传参顺序一致

$Elist \rightarrow E$

{*Elist.paramlist = createEmptyList();*

Elist.push_back(E.place);}

过程调用的翻译方案

为*Elist*设计一个综合属性paramlist，该列表记录函数调用的所有参数，且参数排列顺序与传参顺序一致

$Elist \rightarrow E$

{*Elist.paramlist = createEmptyList();*

Elist.paramlist.push_back(E.place);}

$Elist \rightarrow Elist_1, E$

{*Elist.paramlist = Elist*$_1$*.paramlist;*

Elist.paramlist.push_back(E.place);}

过程调用的翻译方案

$S \rightarrow$ call id (*Elist*)

{for E_i in *Elist.paramlist*:

gen('param', E_i.*place*);

gen('call', id.*place*, *Elist.paramlist.size()*) ;}

本章小结

- 三地址代码格式
- 表达式翻译
- 控制流语句翻译
- 布尔表达式短路计算和翻译
- 标号回填技术

中国科学技术大学
University of Science and Technology of China

USTC, CHINA
ADSLAB
先进数据系统实验室

《编译原理与技术》
中间代码生成 I

Done!