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

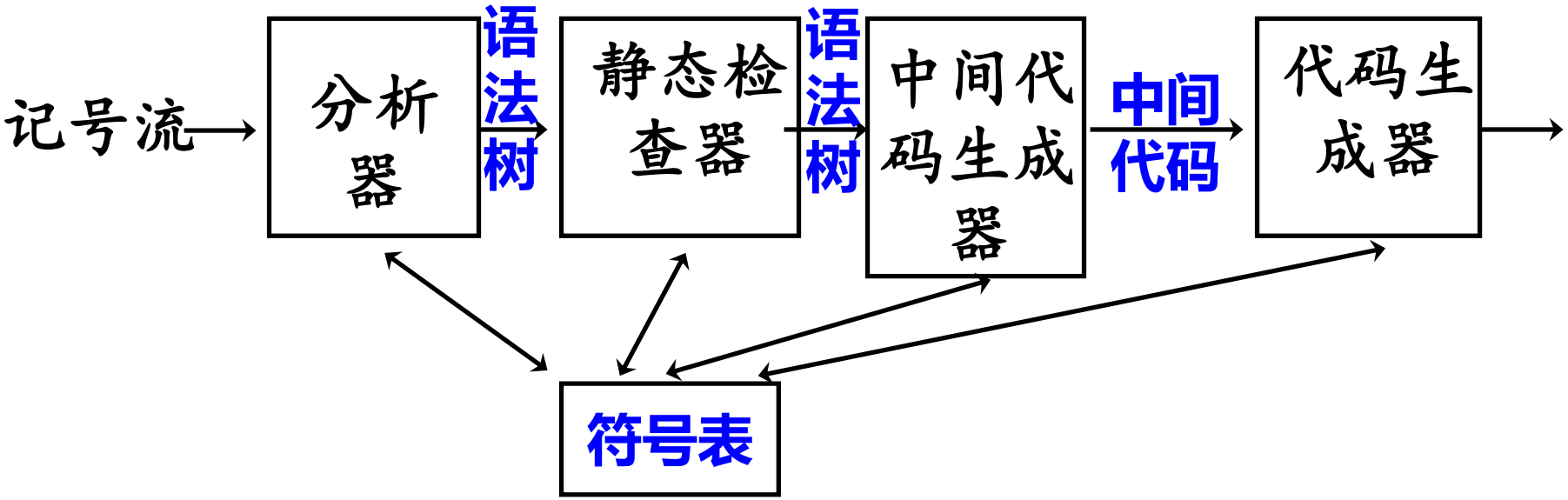
《编译原理与技术》
中间代码生成 II

计算机科学与技术学院
李 诚
26/11/2018

通知

- 实验课每周增加一次：
  - 周二晚上7-9:30
  - 电三楼408-410 小教室
- 期中考试查卷子
  - 时间安排：周三晚上19：00 – 21：30
  - 地点：东校区高性能计算中心402

主要内容



- **中间语言(Intermediate Representation)**
  - 后缀表达式、图表示、三地址码、静态单赋值
- **中间代码生成**
  - 声明语句(更新符号表)
  - 表达式、赋值语句(产生临时变量、查询符号表)
  - 布尔表达式、控制流语句(标号/回填、短路计算)

❑知识要点

- ❖分配临时变量，存储表达式计算的中间结果
- ❖数组元素的地址计算
- ❖类型转换

赋值语句的翻译

❑主要任务

- ❖复杂的表达式 => 多条计算指令组成的序列
- ❖分配临时变量保存中间结果
- ❖id：查符号表获得其存储的场所
- ❖数组元素：元素地址计算
  - ➢符号表中保存数组的基址和用于地址计算的常量表达式的值
  - ➢数组元素在中间代码指令中表示为“基址[偏移]”
- ❖可以进行一些语义检查
  - ➢类型检查、变量未定义/重复定义/未初始化
- ❖类型转换：因为目标机器的运算指令是区分类型的

赋值语句的中间代码生成

❑赋值语句文法

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

❑语义动作用到的函数

- ❖获取id的地址和存放E结果的场所
  - ➢*lookup(id.lexeme);*如果不存在，返回*nil*
- ❖产生临时变量
  - ➢*newTemp();*
- ❖输出翻译后的指令
  - ➢*Emit(addr, op, arg1, arg2)*：三地址码

❑属性：*E.place* 符号表条目的地址

赋值语句的中间代码生成

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

if *p* != *nil* then

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

else *error* }

$E \rightarrow E_1 + E_2$

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

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

赋值语句的中间代码生成

$E \rightarrow -E_1$ {*E.place = newTemp();*

emit (E.place, ‘=’, ‘uminus’, E_1*.place)* }

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

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

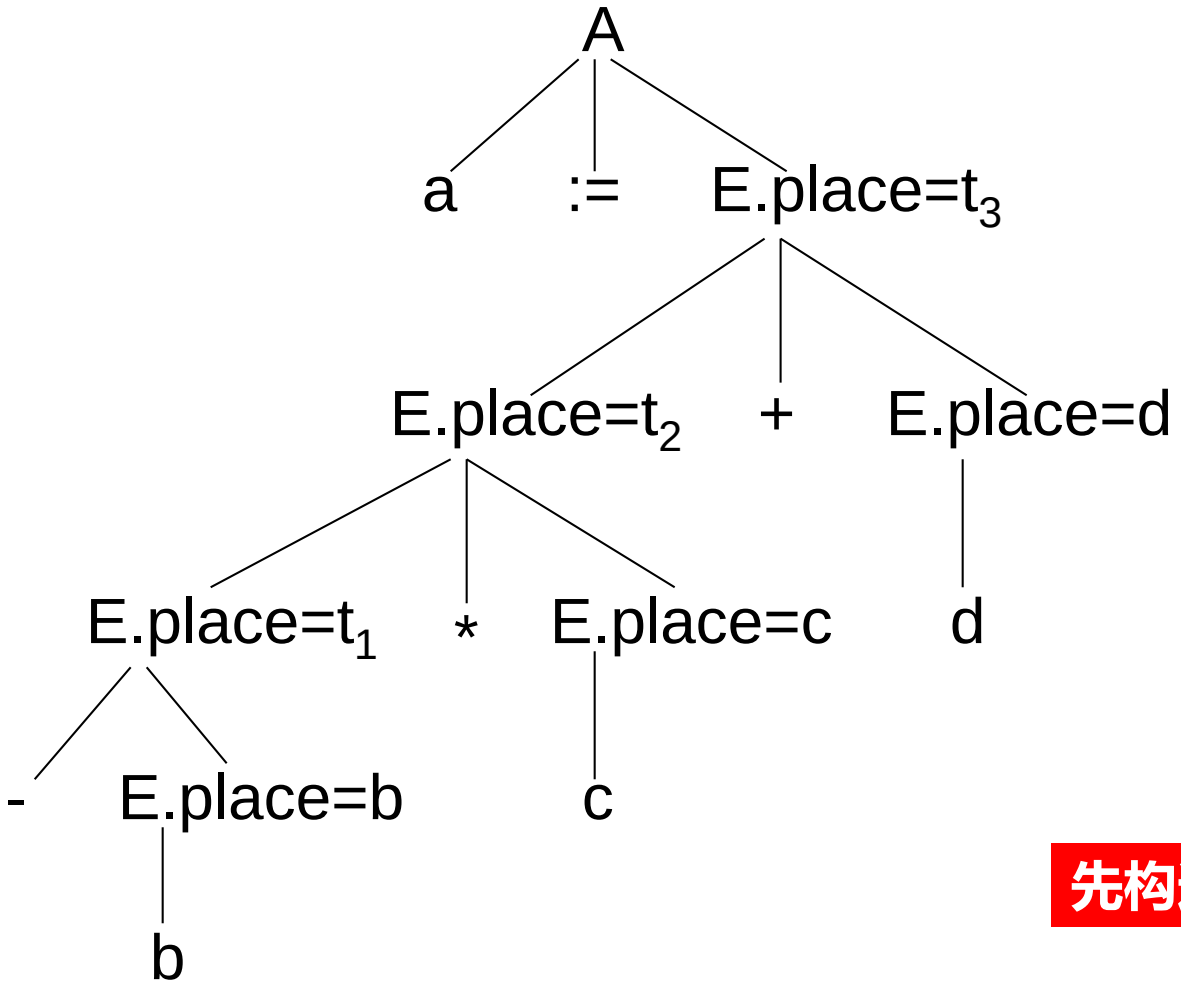
if *p* != *nil* then

E.place = p

else *error* }

举例：赋值语句翻译

❑**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

先构造语法树

数组元素的翻译

❑数组类型的声明

e.g. Pascal的数组声明，

A : array[$low_1..high_1$,…,$low_n..high_n$] of integer ;

数组元素：A[i , j, k,…] 或 A[i][j][k]…

（下界）$low_1 \le i \le high_1$（上界） , …

e.g. C的数组声明，

int A [100][100][100];

数组元素：A[i][30][40] $0 \le i \le (100-1)$

数组元素的翻译

- 翻译的主要任务
  - 输出(Emit)地址计算的指令
  - “基址[偏移]”相关的中间指令：t =b[o]，b[o]=t

数组元素的地址计算

- 一维数组A的第*i*个元素的地址计算

$$base + (i - low) \times w$$

base: 整个数组的基地址

low: 下标的下界

w: 每个数组元素的宽度

数组元素的地址计算

- 一维数组A的第*i*个元素的地址计算

$$base + (i - low) \times w$$

base: 整个数组的基地址

low: 下标的下界

w: 每个数组元素的宽度

可以变换成

$$i \times w + (base - low \times w)$$

$low \times w$**是常量，编译时计算，减少了运行时计算**

数组元素的地址计算

❑二维数组

A: array[1..2, 1..3] of T

❖列为主

A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

❖ 行为主

A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

数组元素的地址计算

❑二维数组

A: array[1..2, 1..3] of T

i_2 ↓

	A[1,1]	A[1,2]	…
i_1 →	A[2,1]	A[2,2]	…
	…	…	…

❖列为主

A[1, 1], A[2, 1], A[1, 2], A[2, 2], A[1, 3], A[2, 3]

❖行为主

A[1, 1], A[1, 2], A[1, 3], A[2, 1], A[2, 2], A[2, 3]

$$base + ((i_1 - low_1) \times n_2 + (i_2 - low_2)) \times w$$

（A[i_1, i_2]的地址，其中$n_2 = high_2 - low_2 + 1$）

变换成 $((i_1 \times n_2) + i_2) \times w +$

$$(base - ((low_1 \times n_2) + low_2) \times w)$$

数组元素的地址计算

- 多维数组下标变量A[i_1, i_2, ..., i_k]的地址表达式
  - 以行为主

$$((\ldots((i_1 \times n_2 + i_2) \times n_3 + i_3) \ldots) \times n_k + i_k) \times w$$

$$+ base - ((\ldots ((low_1 \times n_2 + low_2) \times n_3 + low_3) \ldots) \times n_k + low_k) \times w$$

数组元素地址计算翻译方案

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

- 下标变量访问的产生式

$S \rightarrow L := E$　　$L \rightarrow \mathrm{id}\ [\ Elist\ ] \mid \mathrm{id}$

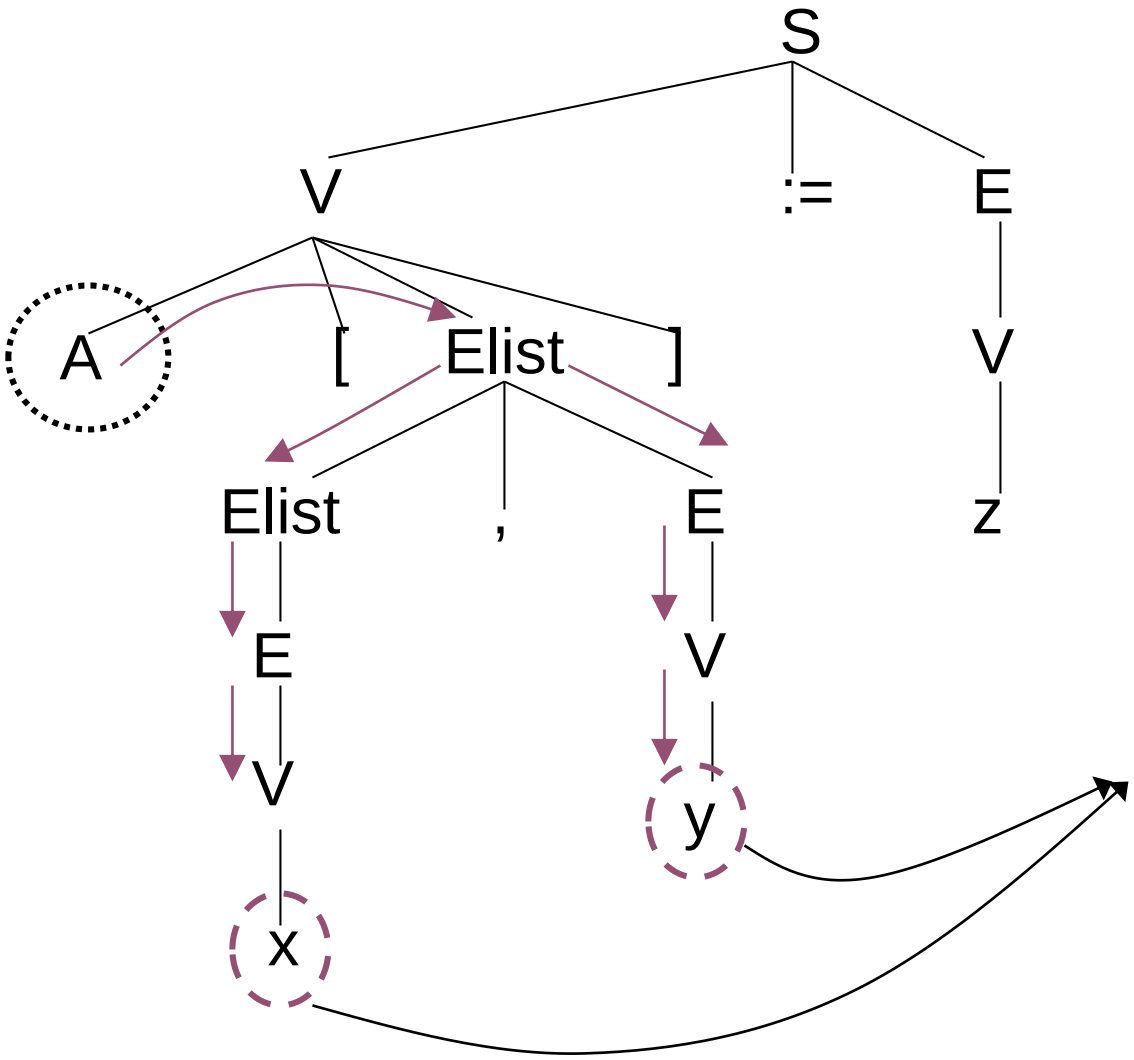
$Elist \rightarrow Elist, E \mid E$　　$E \rightarrow \mathrm{L} \mid \ldots$

- 采用语法制导的翻译方案时存在的问题

$Elist \rightarrow Elist, E \mid E$

由Elist的结构只能得到各维的下标值，但无法获得数组的信息（如各维的长度）

A[x,y] := z的分析树



当分析到下标（表达式）x和y时，要计算地址中的“可变部分”。这时需要知晓数组A的有关的属性，如n_m，类型宽度w等，而这些信息存于在结点A处。若想使用必须定义有关继承属性来传递之。但在移进—归约分析不适合继承属性的计算！

❑所有产生式

$S \rightarrow L := E$

$E \rightarrow E + E$

$E \rightarrow (E)$

$E \rightarrow L$

$L \rightarrow Elist\ ]$

$L \rightarrow \mathrm{id}$

$Elist \rightarrow Elist, E$

$Elist \rightarrow \mathrm{id}\ [\ E$

修改文法，使数组名id成为Elist的子结点（类似于前面的类型声明），从而避免继承属性的出现

相关符号属性定义：

L.place，L.offset ：

- 若L是简单变量，L.place为其“值”的存放场所，而L.offset为空（null）；
- 当L表示数组元素时，L.place是其地址的“常量值”部分；而此时L.offset为数组元素地址中可变部分的“值”存放场所，数组元素的表示为：L.place [L.offset]

相关符号属性定义：

Elist.place：“可变部分”的值，即下标计算的值

Elist.array：数组名条目的指针

Elist.ndim：当前处理的维数

limit(array, j)：第j维的大小

width(array)：数组元素的宽度

invariant(array)：静态可计算的值

数组元素的翻译

- 翻译时重点关注三个表达式：
  - $Elist \rightarrow \text{id} [E$ ： 计算第1维
  - $Elist \rightarrow Elist_1, E$ ： 传递信息
  - $L \rightarrow Elist]$ ： 计算最终结果

数组元素的翻译

$S \rightarrow L := E$ {if *L.offset* == null then /* *L*是简单变量 */

emit (*L.place*, '= ', *E.place*)

else

/*取数组元素的左值*/

emit (*L.place* , '[', *L.offset*, ']', '=', *E.place*) }

数组元素的翻译

$Elist \rightarrow \text{id} [E$ {*Elist.place* = *E.place*;
/*第一维下标*/
Elist.ndim = 1;
Elist.array = id.*place* }

数组元素的翻译

$Elist \rightarrow Elist_1, E$

{

$t = newTemp();$

/*维度增加1*/

$m = Elist_1.ndim + 1;$

/* 第m维的大小*/

$n_m = limit(Elist_1.array, m);$

/*计算公式7.6 $e_{m-1} * n_m$ */

$emit\ (t, '=', Elist_1.place, '*', n_m);$

/*计算公式7.6 $e_m = e_{m-1} * n_m + i_m$*/

$emit\ (t, '=', t, '+', E.place);$

$Elist.array = Elist_1.array;$

$Elist.place = t;$

$Elist.ndim = m$

}

数组元素的翻译

$L \rightarrow Elist$] { *L.place* = *newTemp*();

/*获取数组元素地址的常量值*/

emit (*L.place*, '=', *base*(*Elist.array*), '–',
invariant (*Elist.array*));

L.offset = *newTemp*();

/*获取数组元素地址的可变部分*/

emit (*L.offset*, '=', *Elist.place*, '*',
width(*Elist.array*))}

数组元素的翻译

$L \rightarrow$ id {*L.place* = id.*place*; *L.offset* = null }

$E \rightarrow L$ { if *L.offset* == null then /* *L*是简单变量 */

E.place = *L.place*

else begin *E.place* = *newTemp*();

emit (*E.place*, '=', *L.place*, '[', *L.offset*, ']') end }

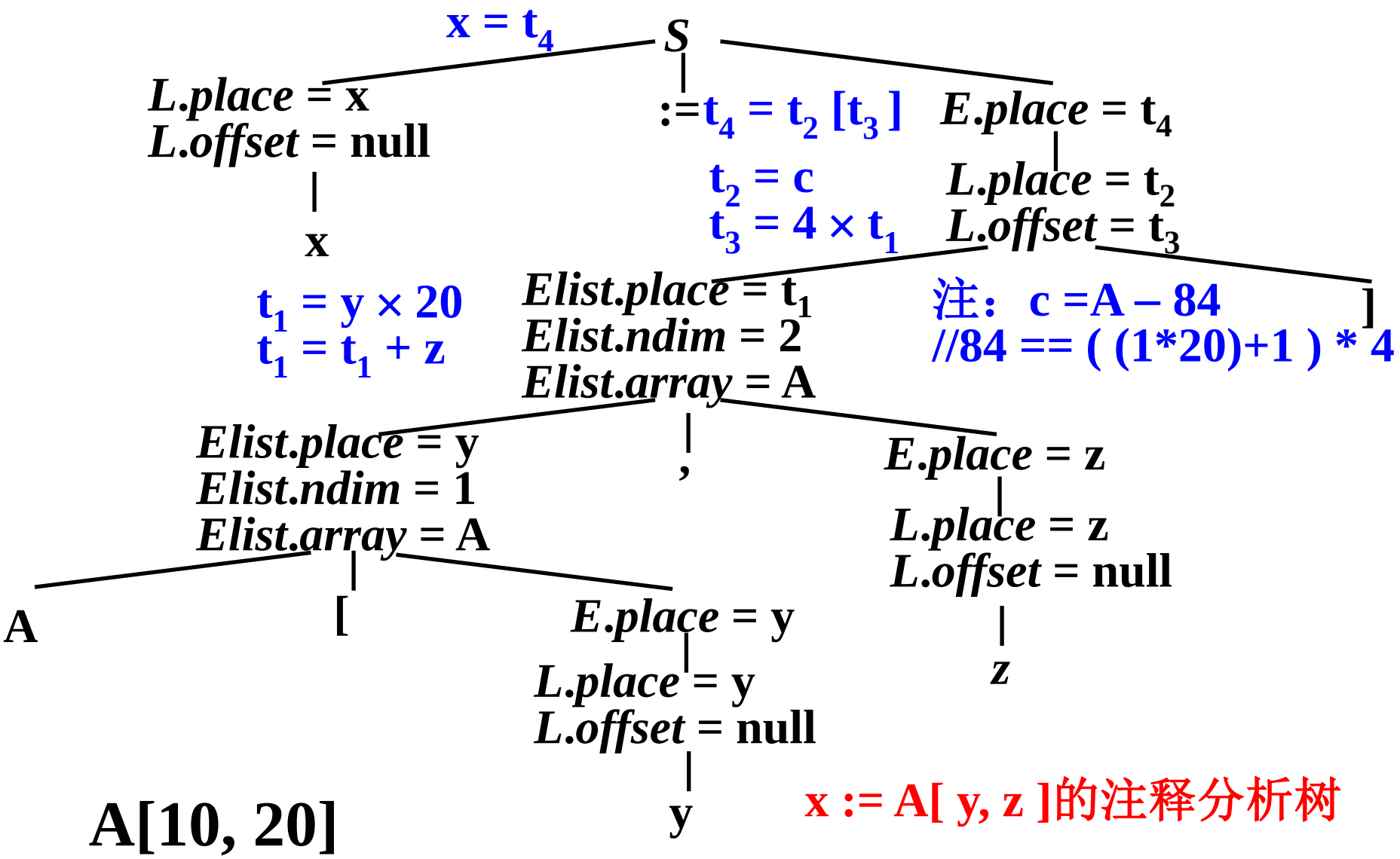
$E \rightarrow E_1 + E_2$ {*E.place* = *newTemp*();

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

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

其他翻译同前

举例： x := A[y, z]



x := A[y, z]的注释分析树

举例：A[i, j] := B[i, j] * k

- 数组A：A[1..10, 1..20] of integer;

  数组B：B[1..10, 1..20] of integer;

  w ：4 （integer）

- TAC如下：

  (1) t_1 := i * 20

  (2) t_1 := t_1 + j

  (3) t_2 := A - 84 // 84 == ((1*20)+1) * 4

  (4) t_3 := t_1 * 4 // 以上A[i, j]的（左值）翻译

举例：A[i, j] := B[i, j] * k

TAC如下（续）：

(5) t_4 := i * 20

(6) t_4 := t_4 + j

(7) t_5 := B - 84

(8) t_6 := t_4 * 4

(9) t_7 := t_5[t_6]

//以上计算B[i,j]的右值

TAC如下（续）：

(10) t_8 := t_7 * k

//以上整个右值表达

//式计算完毕

(11) t_2[t_3] := t_8

// 完成数组元素的赋值

类型转换

❑例 x = y + i * j
（x和y的类型是real，i和j的类型是integer）

中间代码

t_1 = i **int×** j
t_2 = **inttoreal** t_1
t_3 = y **real+** t_2
x = t_3

int× 和 real+ 不是类型转换，而是算符

目标机器的运算指令是区分整型和浮点型的
高级语言中的重载算符=>中间语言中的多种具体算符

类型转换的处理

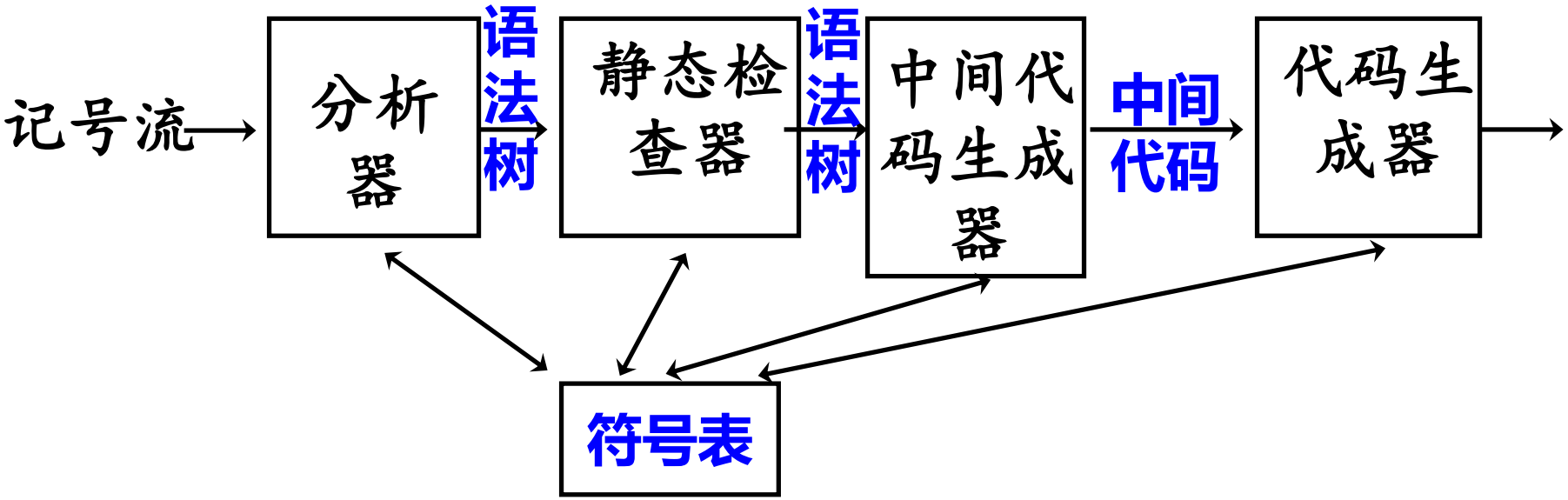
- 以$E \rightarrow E_1 + E_2$为例说明
  - 判断E1 和E2的类型，看是否要进行类型转换;若需要，则分配存放转换结果的临时变量并输出类型转换指令

```
{ E.place = newTemp();
if (E1.type == integer && E2.type == integer) then begin
  emit (E.place, '=', E1.place, 'int+', E2.place);
  E.type = integer
end
else if (E1.type == integer && E2.type == real) then
  begin
  u = newTemp(); emit (u, '=', 'inttoreal', E1.place);
  emit (E.place, '=', u, 'real+', E2.place); E.type = real;
end
  . . .}
```

通知

- Lab 4 is released today.
- Only three groups of students attended the tutorial on Tuesday evening. So sad!

主要内容



- **中间语言(Intermediate Representation)**
  - 后缀表达式、图表示、三地址码、静态单赋值
- **中间代码生成**
  - 声明语句(更新符号表)
  - 表达式、赋值语句(产生临时变量、查询符号表)
  - 布尔表达式、控制流语句(标号/回填、短路计算)

中间代码生成的主要任务

❑主要任务

- ❖布尔表达式的计算：完全计算、短路计算
- ❖控制流语句
  - ➢分支结构(if、switch)、循环结构、过程/函数的调用
- ❖各子结构的布局+无条件或有条件转移指令
- ❖跳转目标的两种处理方法
  - ➢标号技术：新建标号，跳转到标号
  - ➢回填技术：先构造待回填的指令链表，待跳转目标确定时再回填链表中各指令缺失的目标信息

布尔表达式

- 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 定义成 if B_1 then true else B_2
- ❖ B_1 and B_2 定义成 if B_1 then B_2 else false
- ❖ not A 定义成 if A then false else true

布尔表达式的“短路”计算

- 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:
```

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

| switch E

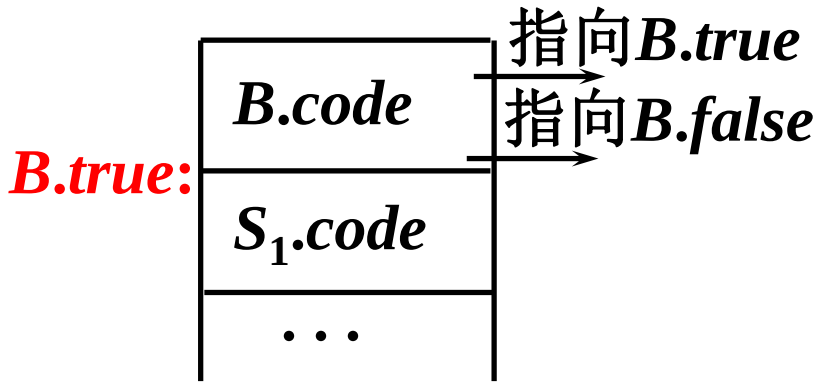
| call id ($Elist$)

if 语句的中间代码生成

❑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

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



(a) if-then

if 语句的中间代码生成

❑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

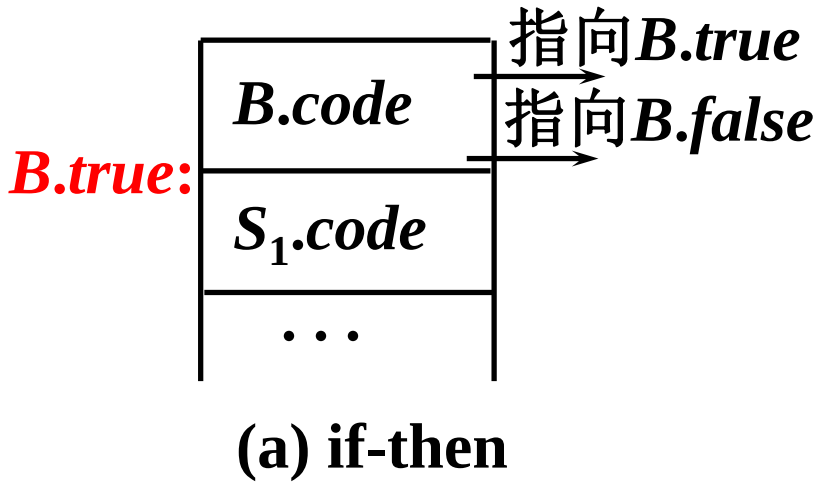
$S \rightarrow$ if B then S_1

{*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 语句的中间代码生成

❑问题与对策

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

令，
号

- 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

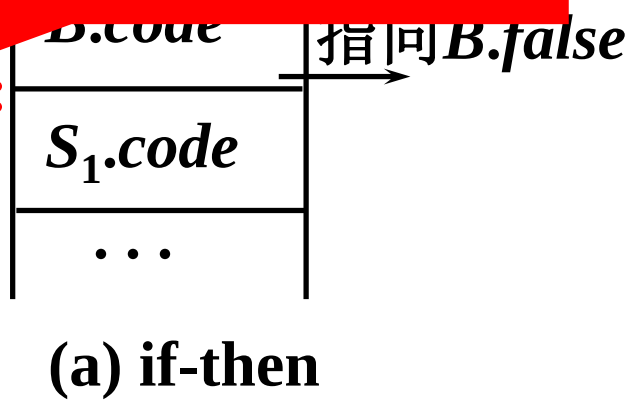
$S \rightarrow$ if B then S_1

{*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 语句的中间代码生成

❑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

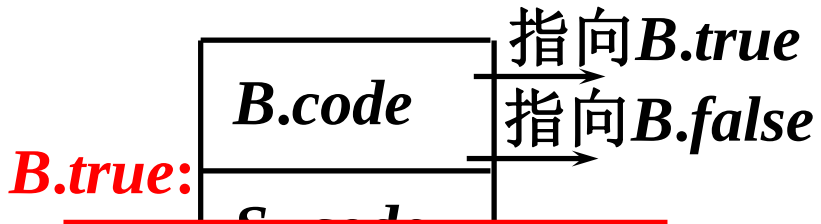
$S \rightarrow$ if B then S_1

{*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* }



把B.true这个标号附加到S_1的第一条三地址指令上

if 语句的中间代码生成

❑考虑else

$S \rightarrow$ if B then S_1 else S_2

{*B.true* = *newLabel*();

B.false = *newLabel*();

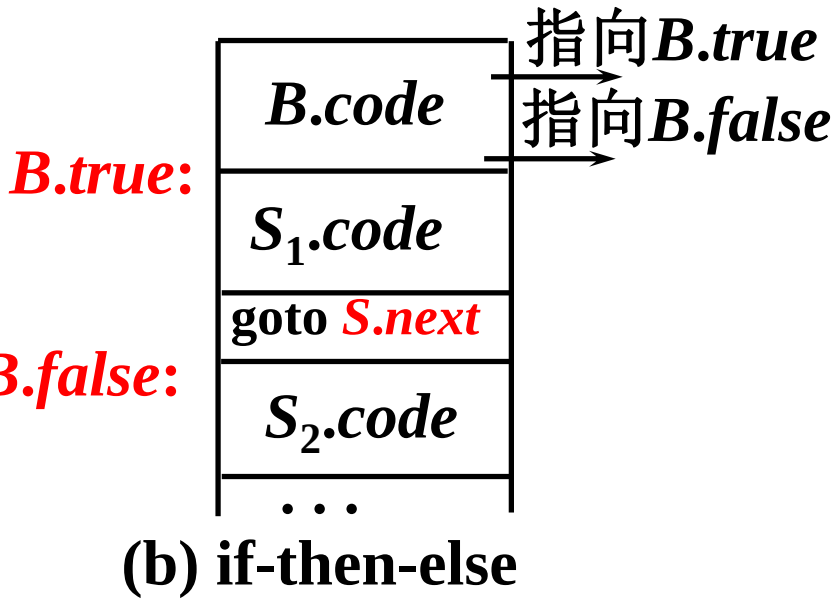
S_1.*next* = *S.next*;

S_2.*next* = *S.next*;

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

gen('goto', *S.next*) || *gen*(*B.false*, ':') ||

S_2.*code*}

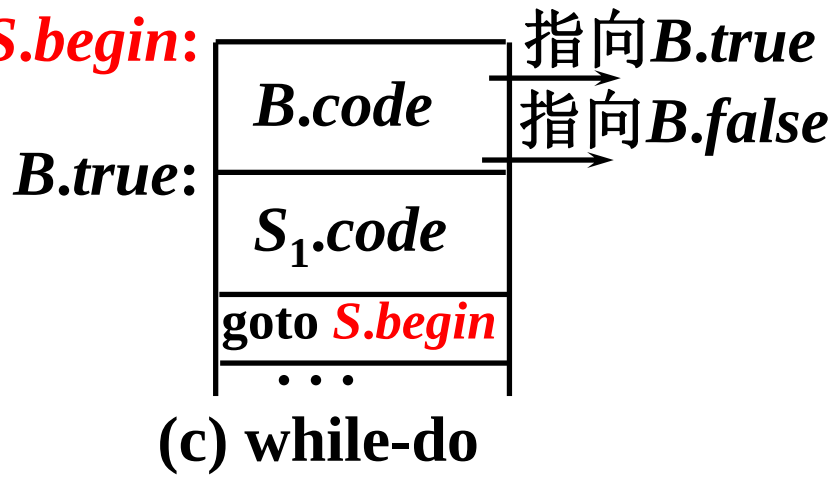


(b) if-then-else

while语句的中间代码生成

- 引入开始标号*S.begin*,作为循环的跳转目标

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



while语句的中间代码生成

- 引入开始标号*S.begin*,作为循环的跳转目标

$S \rightarrow$ while B do S_1

{*S.begin* = *newLabel*();

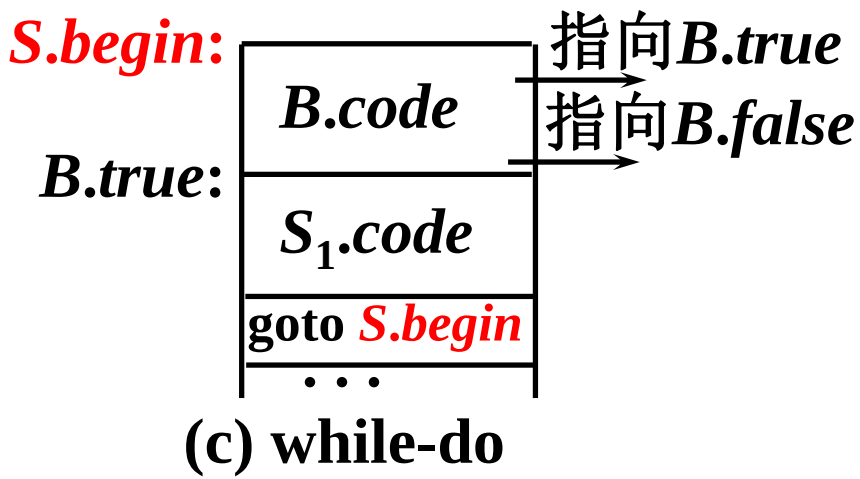
B.true = *newLabel*();

B.false = *S.next*;

S_1.*next* = *S.begin*;

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

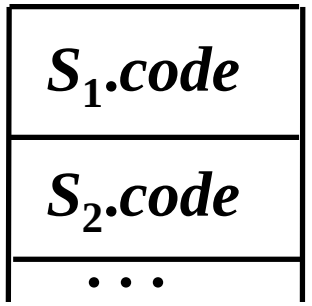
gen(*B.true*, ':') || S_1.*code* || *gen*('goto', *S.begin*) }



顺序结构的翻译

- 为每一语句S_1引入其后的下一条语句的标号

$S_1.next$



(d) $S_1; S_2$

$S \rightarrow S_1; S_2$

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

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

布尔表达式的控制流翻译

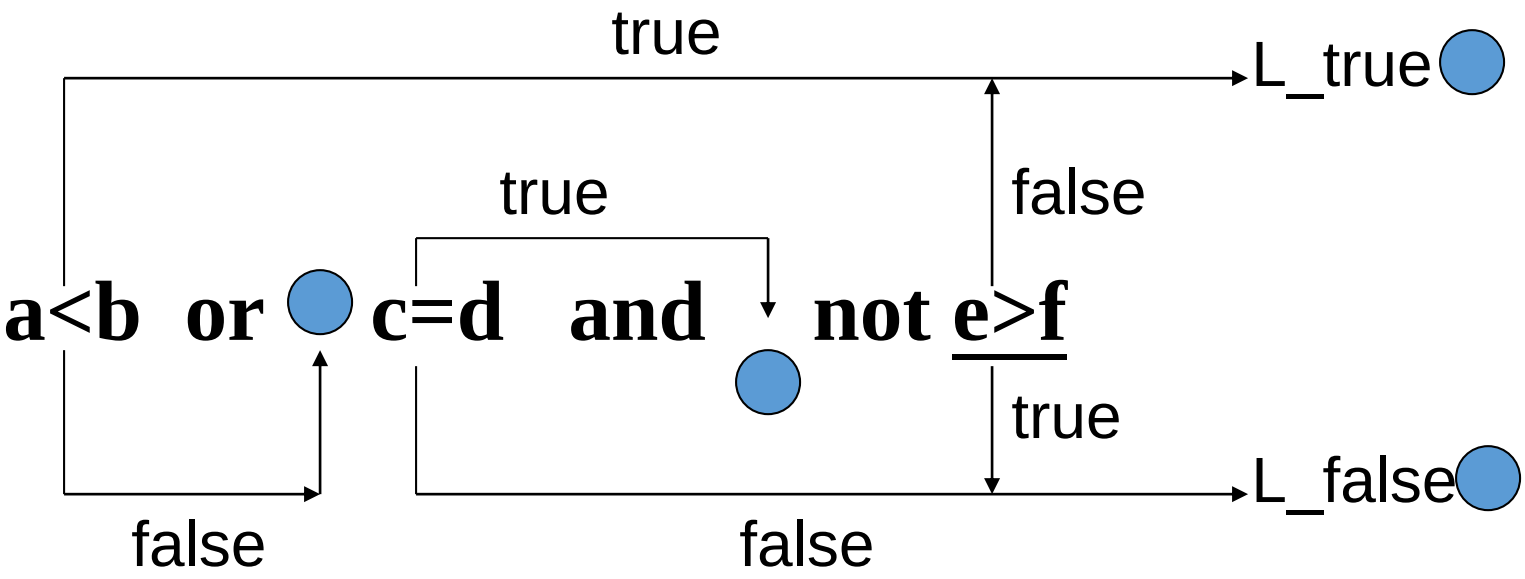
- 在控制流翻译中，并未对B.code进行展开，现在考虑B.code的三地址代码翻译
- 如果B是$a < b$的形式，

那么翻译生成的三地址码是：

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

goto $B.false$

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

❑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

$B \rightarrow B_1$ or B_2

{$B_1.true = B.true$;

$B_1.false = newLabel()$;

$B_2.true = B.true$;

$B_2.false = B.false$;

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

布尔表达式的控制流翻译

$B \rightarrow \text{not } B_1$

$\{B_1.true = B.false;$

$B_1.false = B.true;$

$B.code = B_1.code \}$

布尔表达式的控制流翻译

$B \rightarrow B_1$ and B_2

{$B_1.true = newLabel()$;

$B_1.false = B.false$;

$B_2.true = B.true$;

$B_2.false = B.false$;

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

布尔表达式的控制流翻译

$B \rightarrow (B_1)$

$\{B_1.true = B.true;$

$B_1.false = B.false;$

$B.code = B_1.code\ \}$

$B \rightarrow E_1$ relop E_2

{*B.code* = E_1.*code* || E_2.*code* ||

gen('if', E_1.*place*, relop.*op*, E_2.*place*,

'goto', *B.true*) ||

gen('goto', *B.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

$B \rightarrow$ true

{*B.code* = *gen*('goto', *B.true*)}

$B \rightarrow$ false

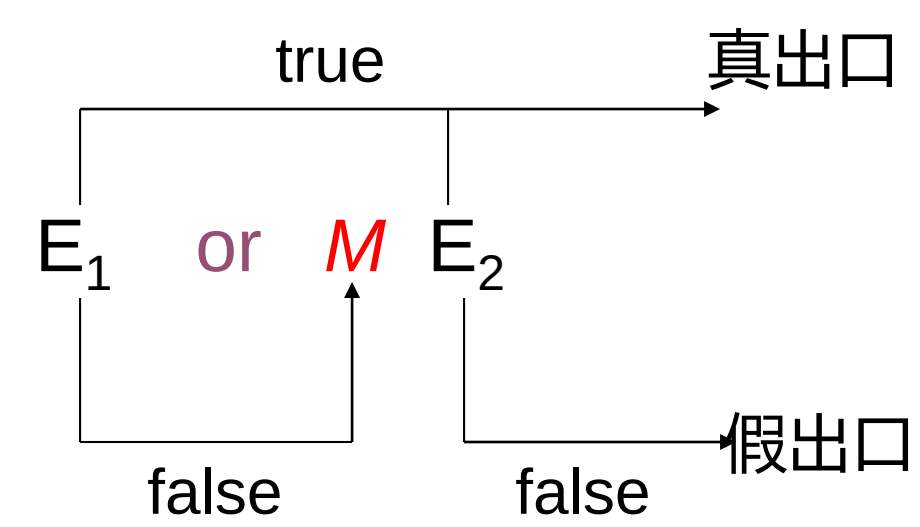
{*B.code* = *gen*('goto', *B.false*)}

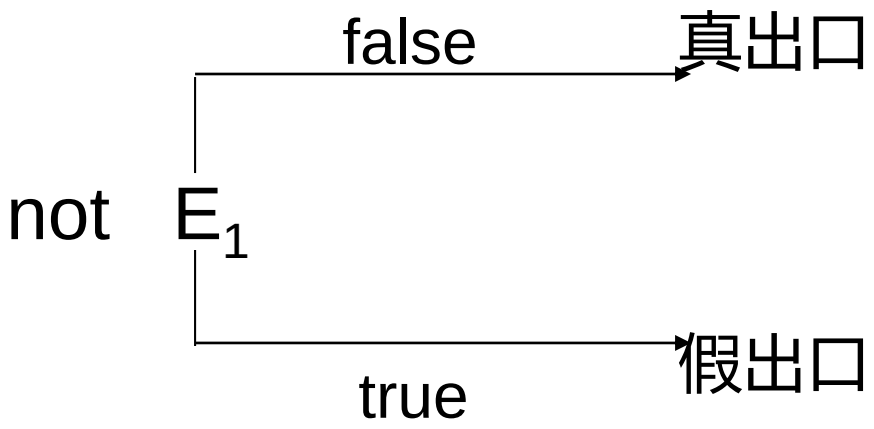
- 关键问题：将跳转指令与目标匹配起来
- B.true, B.false都是继承属性
- 需要两趟分析来计算
  - 1 pass：生成语法树
  - 2 pass：深度优先遍历树，计算属性值
    - 将标号和具体地址绑定起来
- 能否一趟完成?

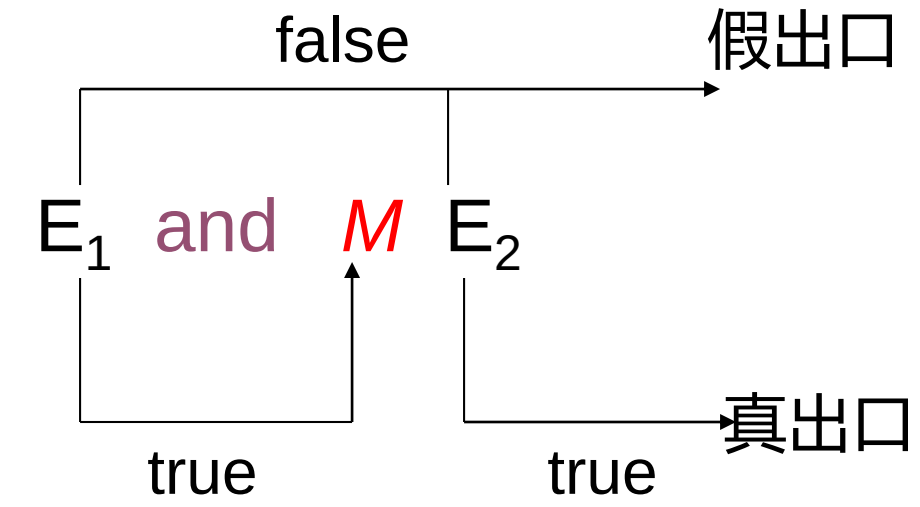
回填(backpatching)技术

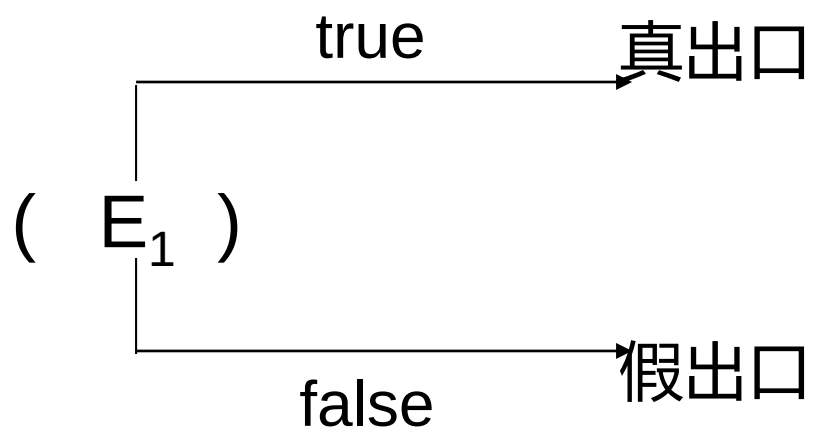
- 问题：
  - 布尔表达式短路计算翻译中，产生了转移目标不明确的条件或无条件代码；
- 解决方案：
  - 当生成跳转指令时，暂时不指定目标地址
  - 当有关目标地址确定后，再填回到翻译代码中
- 具体实现：
  - 将有相同转移目标的转移代码的编号串起来形成链；可以方便回填目标地址。
  - 该list变成了综合属性

短路计算

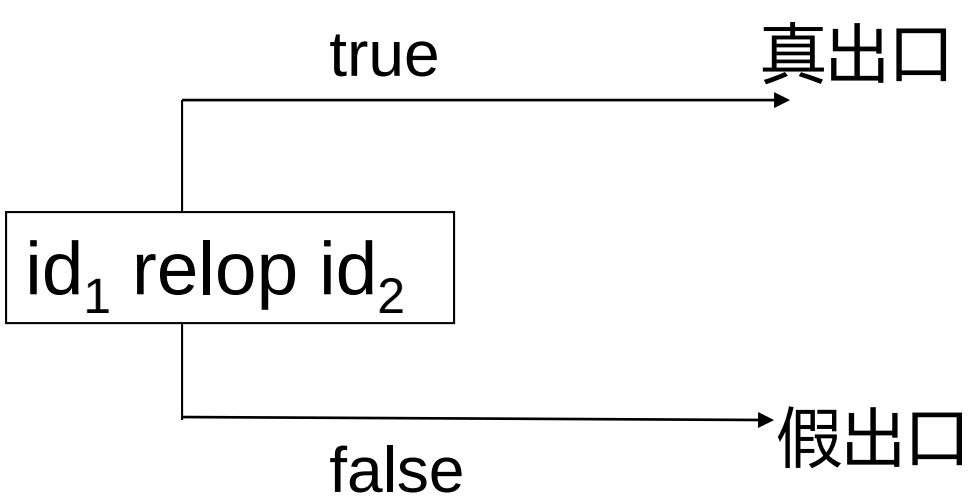

true
真出口
E1 or M E2
假出口
false
false


false
真出口
not E1
假出口
true


false
假出口
E1 and M E2
真出口
true
true

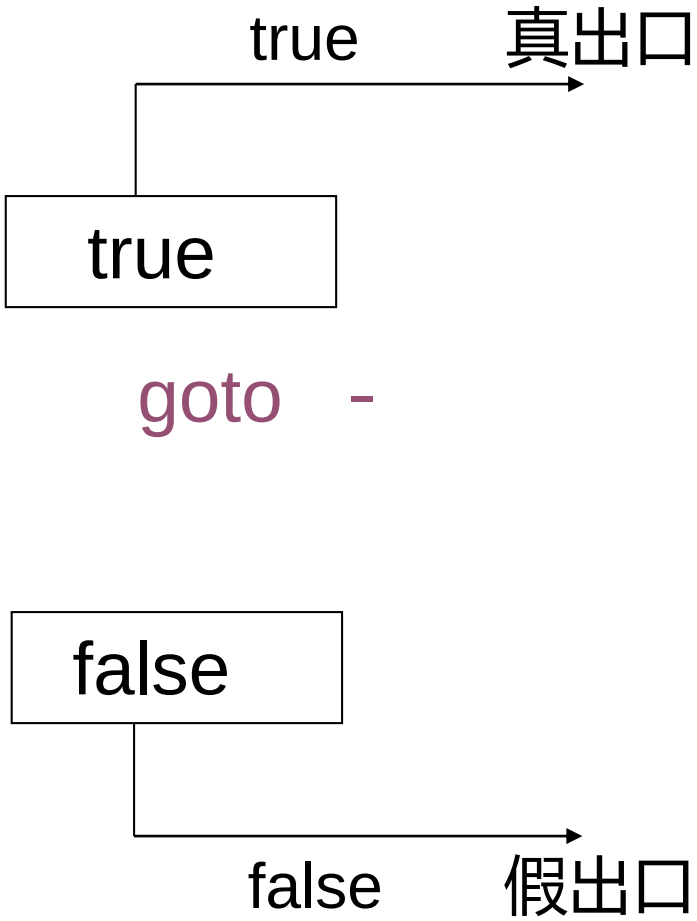

true
真出口
(E1)
假出口
false

短路计算



if id_1 relop id_2 goto -

goto -



相关符号属性及语义函数

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

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

相关符号属性及语义函数

/*将目标地址target-label填回instruction-list中每条语句*/

backpatch(instruction-list, target-label)

/*合并链$list_1$和$list_2$（它们包含的语句转移目标相同）*/

merge(instruction-$list_1$, instruction-$list_2$)

/*建立含语句编号为instruction的链或空链*/

makelist(instruction) , makelist()

/*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*/

$M \rightarrow \varepsilon$ { 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; }

$M \rightarrow \varepsilon$ { 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; }

$M \rightarrow \varepsilon$ { 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

$B \rightarrow$ 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

$B \rightarrow E_1$ 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→ 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);

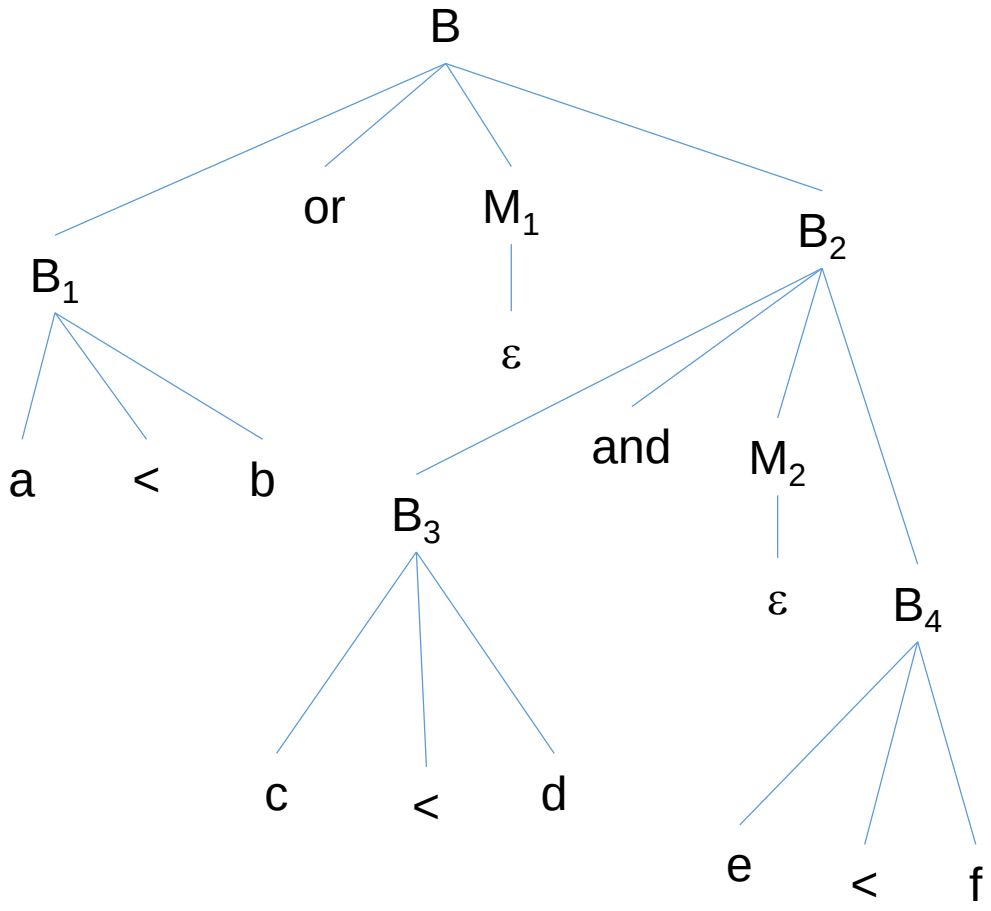
/*为假时，执行无条件跳转指令，但是目标为空。当目标明确后，回填nextinstr所对应的跳转指令*/

gen(“goto” -);}

布尔表达式的翻译

a<b or c<d and e<f

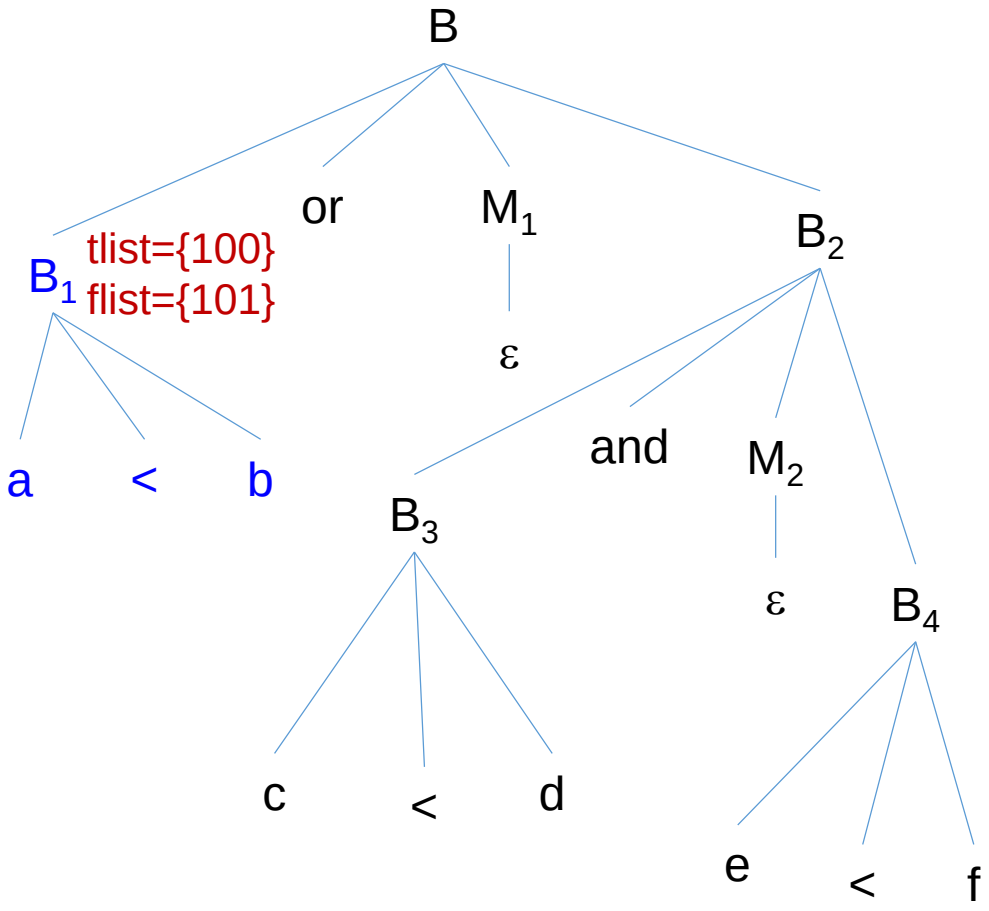
假设nextinstr = 100



布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

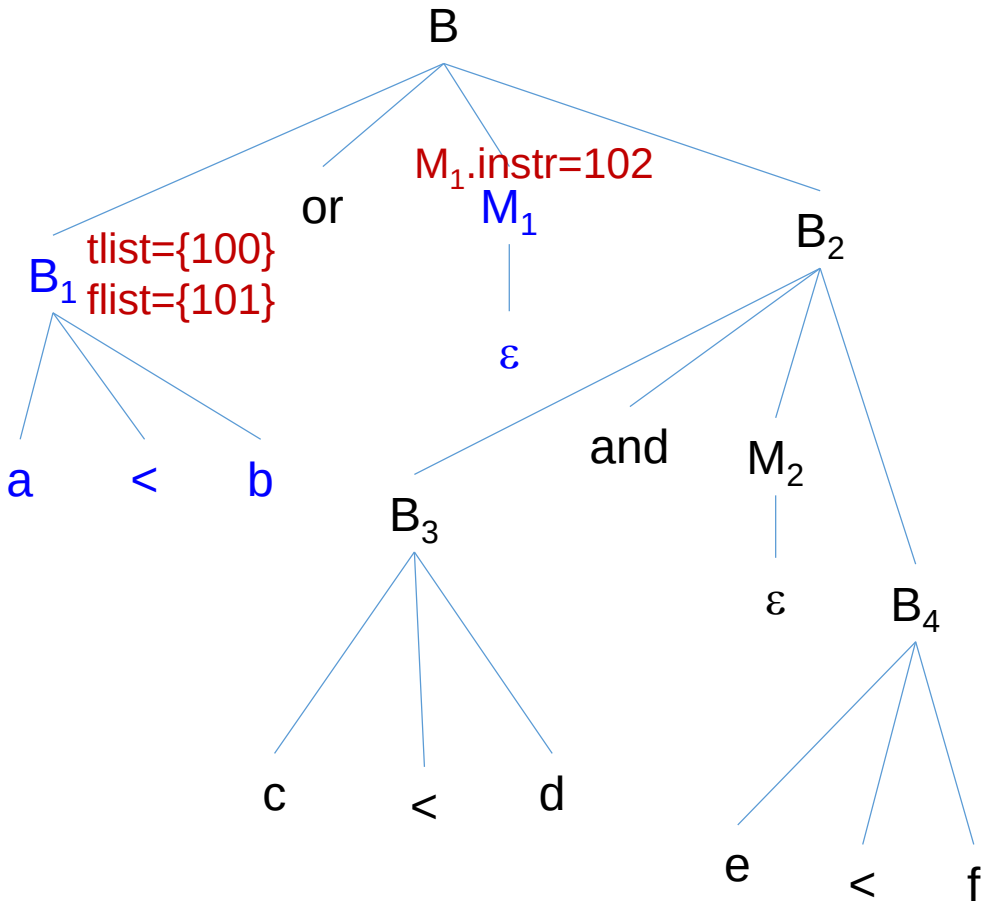


(100) if a<b goto -
(101) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

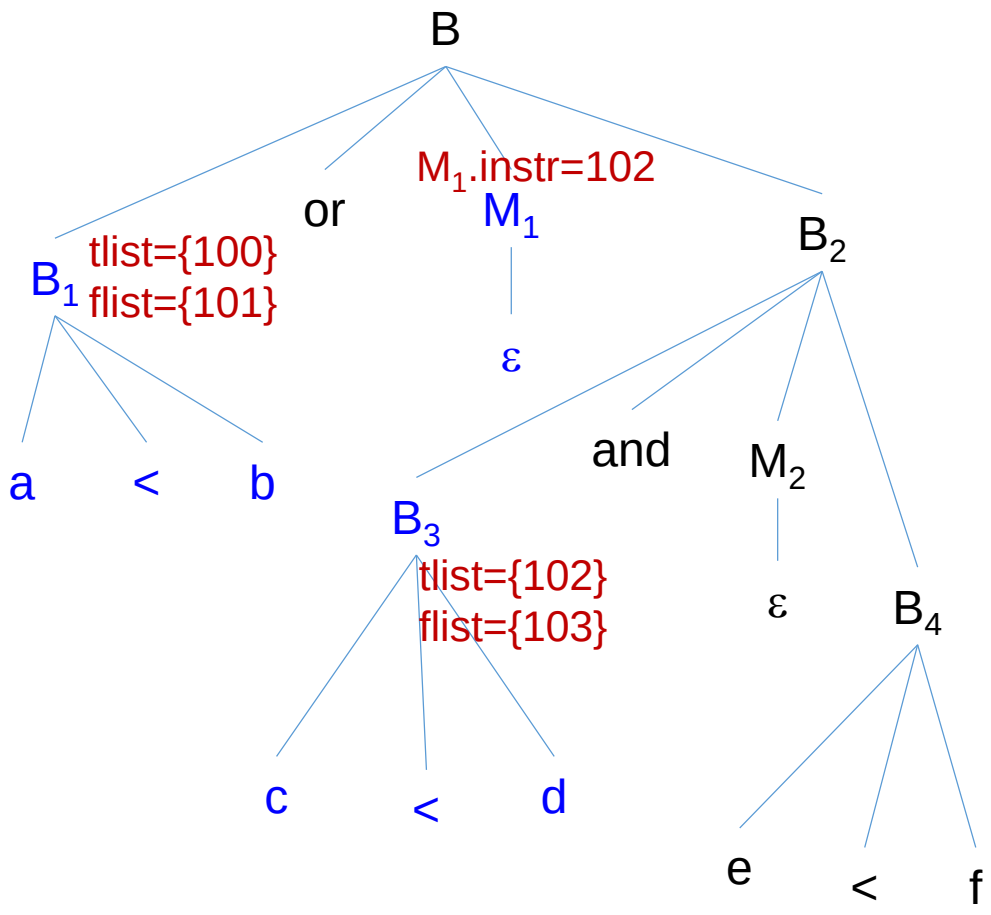


(100) if a<b goto -
(101) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

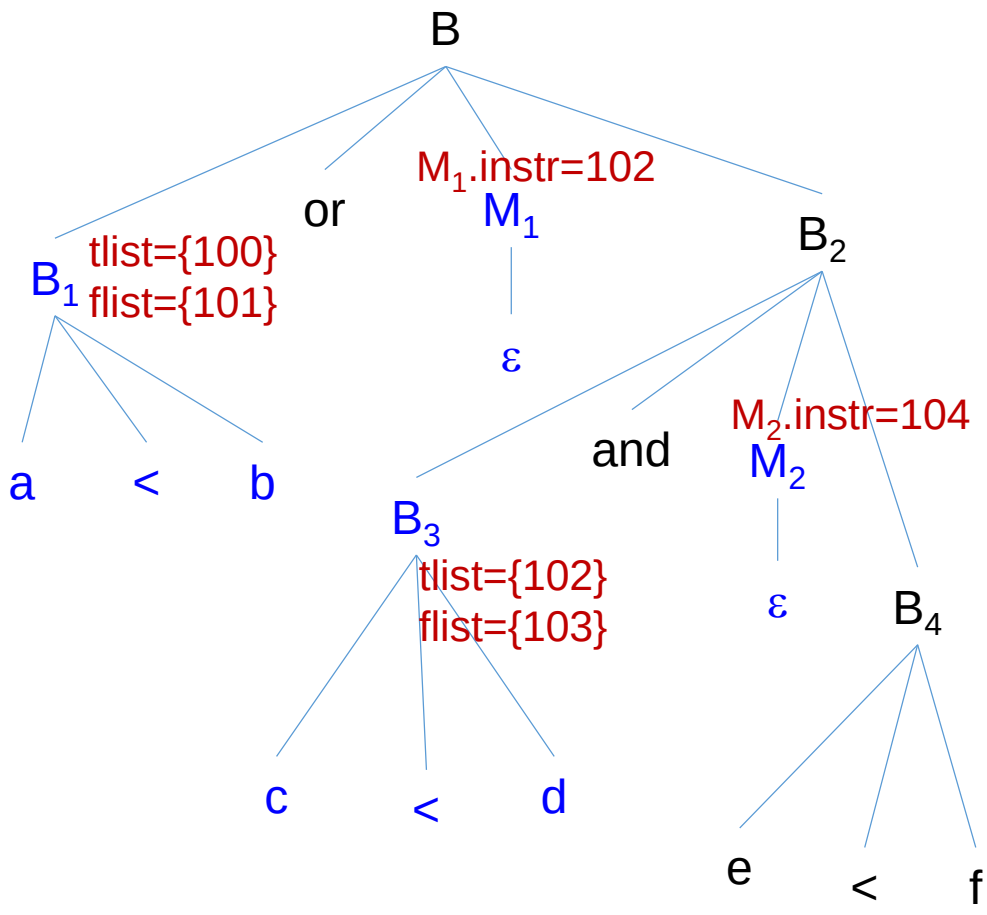


(100) if a<b goto -
(101) goto -
(102) if c<d goto -
(103) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

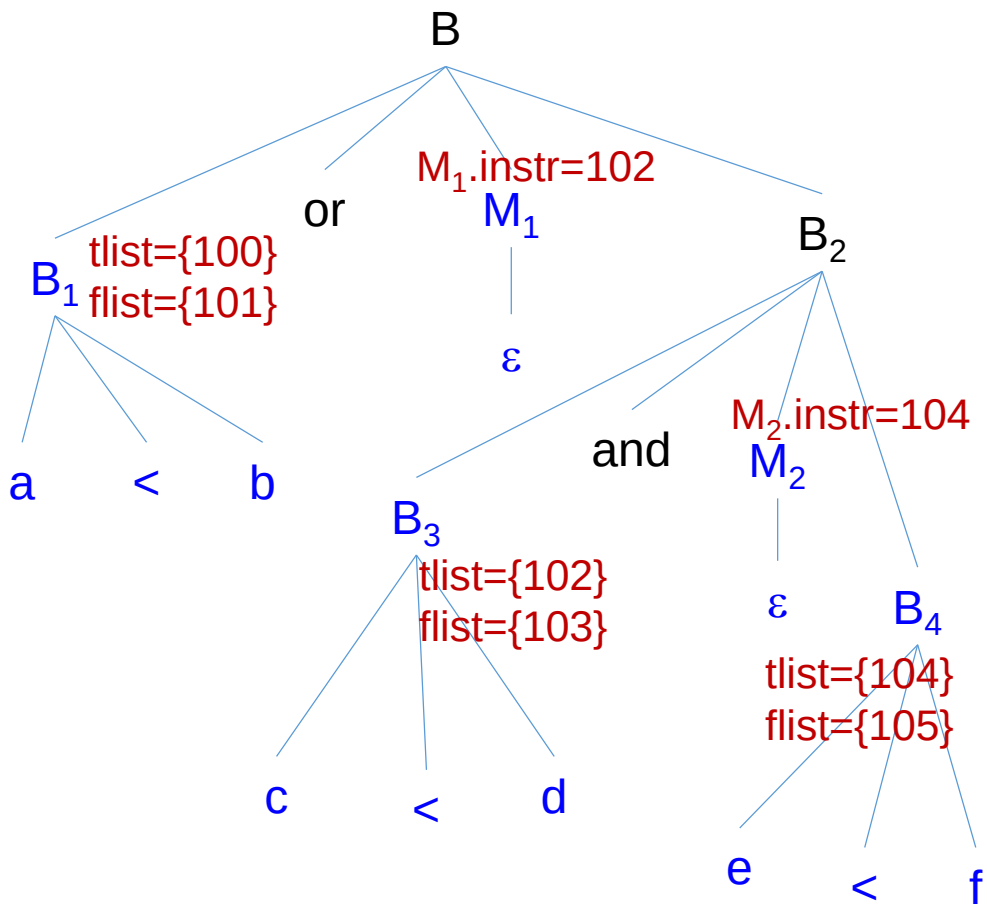


(100) if a<b goto -
(101) goto -
(102) if c<d goto -
(103) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

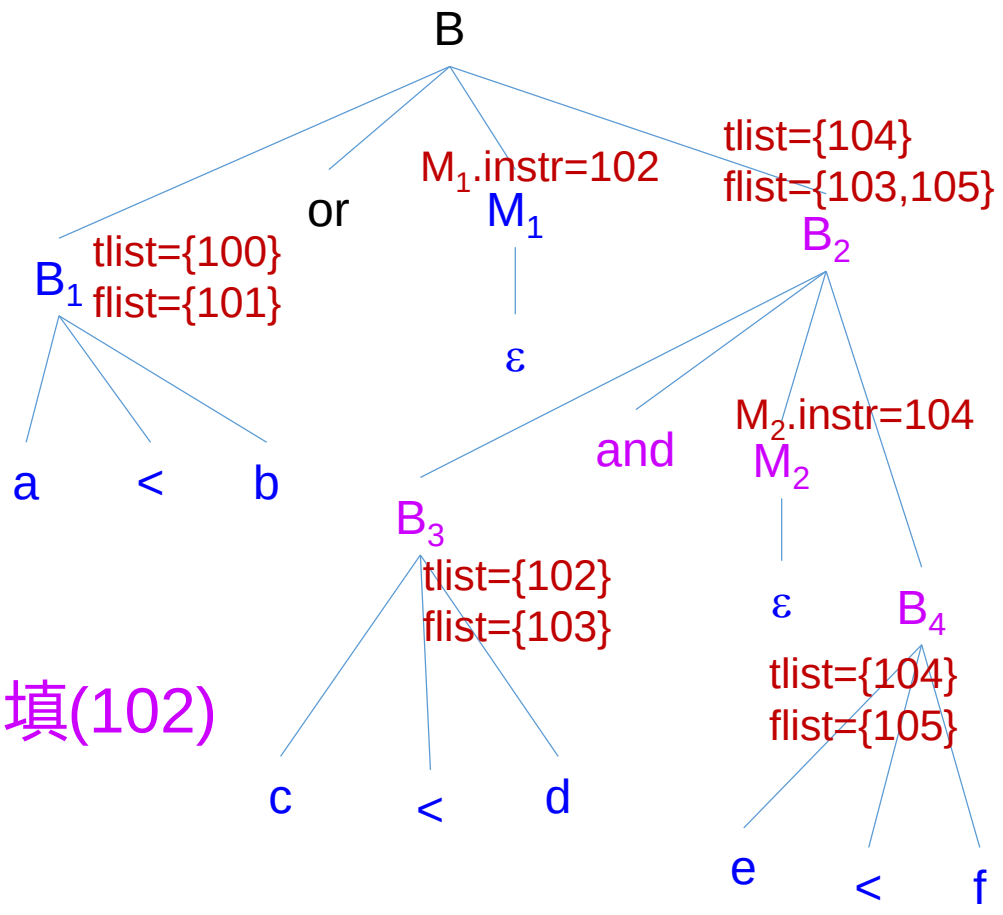


(100) if a<b goto -
(101) goto -
(102) if c<d goto -
(103) goto -
(104) if e<f goto -
(105) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

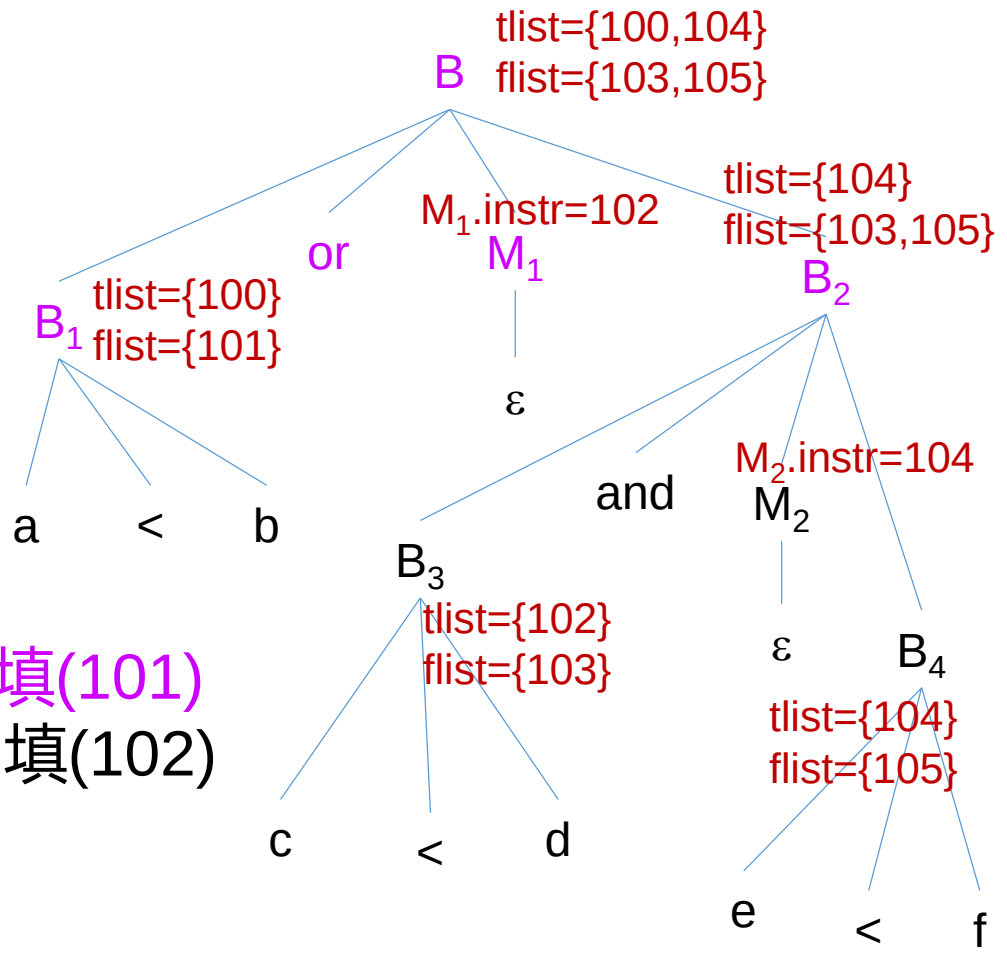


(100) if a<b goto -
(101) goto -
(102) if c<d goto 104//用104回填(102)
(103) goto -
(104) if e<f goto -
(105) goto -

布尔表达式的翻译

a<b or c<d and e<f

假设nextinstr = 100

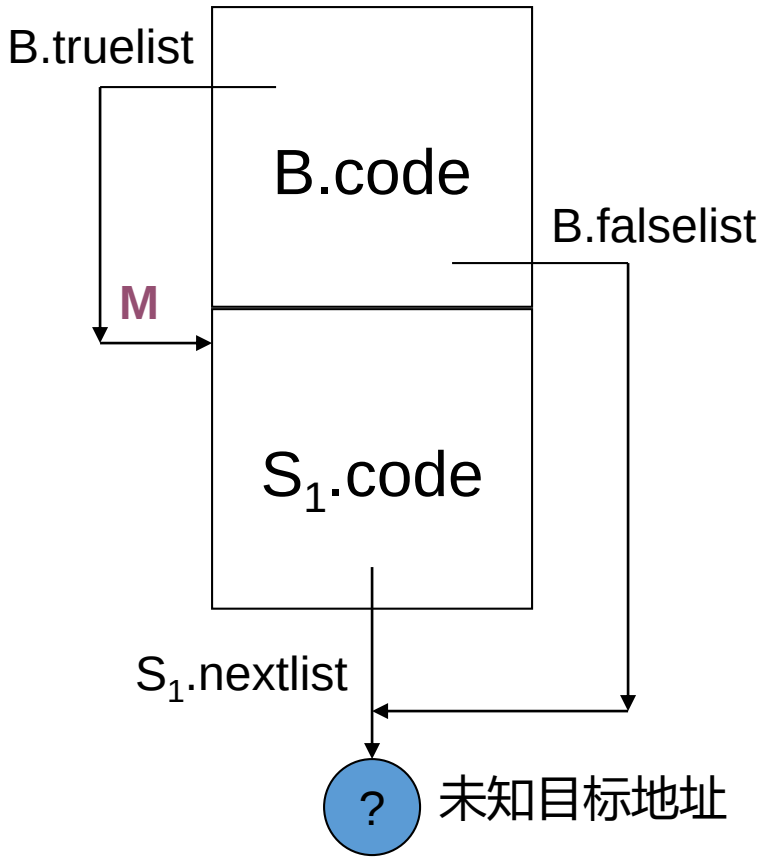


(100) if a<b goto -
(101) goto 102 //用102回填(101)
(102) if c<d goto 104//用104回填(102)
(103) goto -
(104) if e<f goto -
(105) goto -

其他部分的回填要依赖与其他语句的翻译

条件语句的翻译（1）

if B then S_1 的代码结构



箭头线表示控制流方向；

B.truelist和B.falselist 意义同前；

S.nextlist - 语句S的代码中所有跳转到未知目标地址的转移代码（如果有的话）的编号链。**该未知目标地址是指语义上语句S执行结束后应执行的下一代码的位置。**

条件语句的翻译（1）

S→ if B then M S_1

{

backpatch(B.truelist, M.instr);

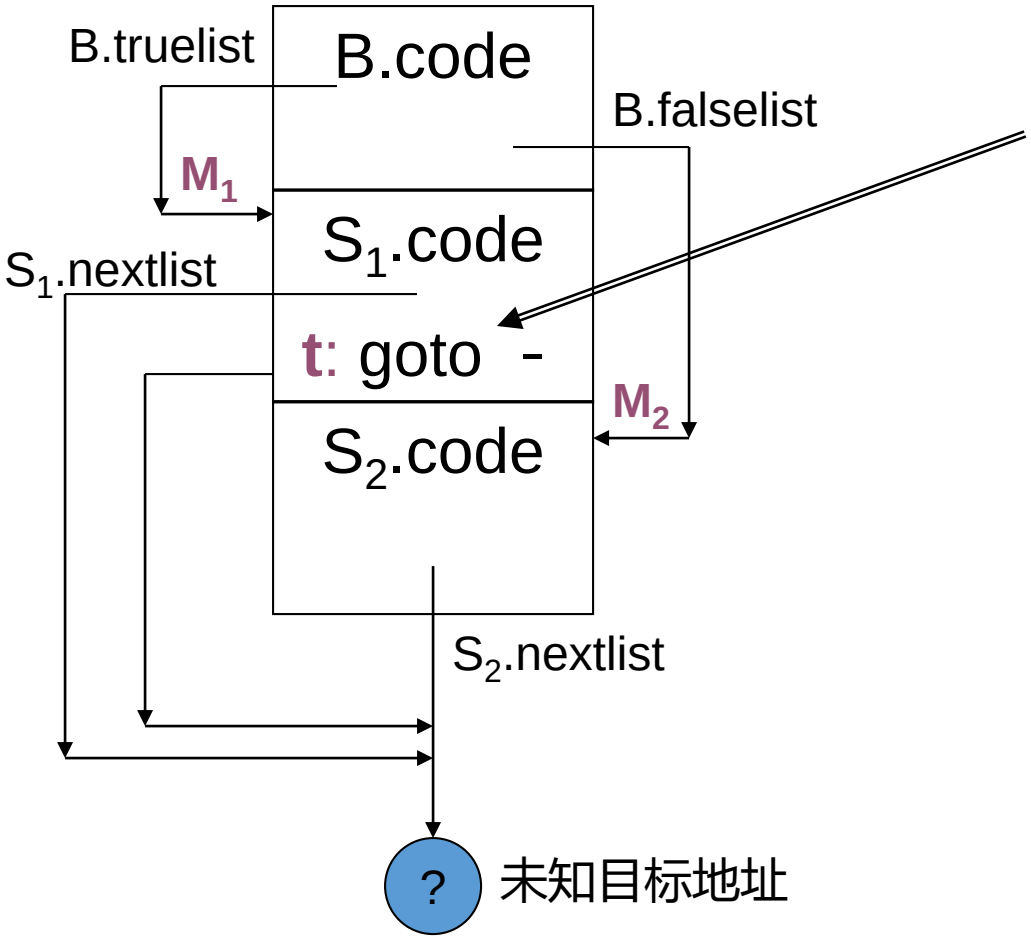
S.nextlist = merge(B.falselist, S_1.nextlist)

}

M→ε　　{ M.instr = nextinstr}

条件语句的翻译（2）

if B then S_1 else S2的代码结构



在代码标号t处强制产生无条件转移代码，转移目标待回填。

引入非终结符标记N

条件语句的翻译（2）

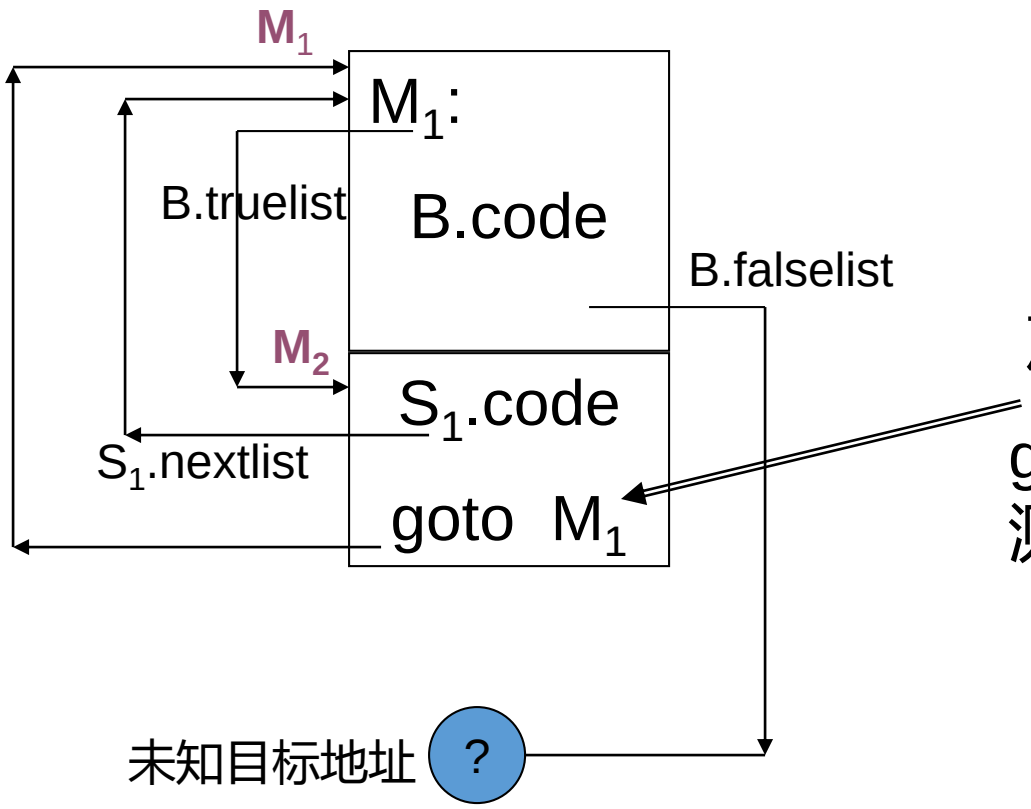
$S \rightarrow$ if B then M_1 S_1 *N* else M_2 S_2

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

```
N→ε { N.nextlist = makelist(nextinstr); //标号t
        gen( "goto"  - );
      }
```

循环语句的翻译

while B do S_1 的代码结构



产生无条件转移语句

goto M_1（跳转至循环条件测试代码开始处）

循环语句的翻译

$S \rightarrow$ 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

}

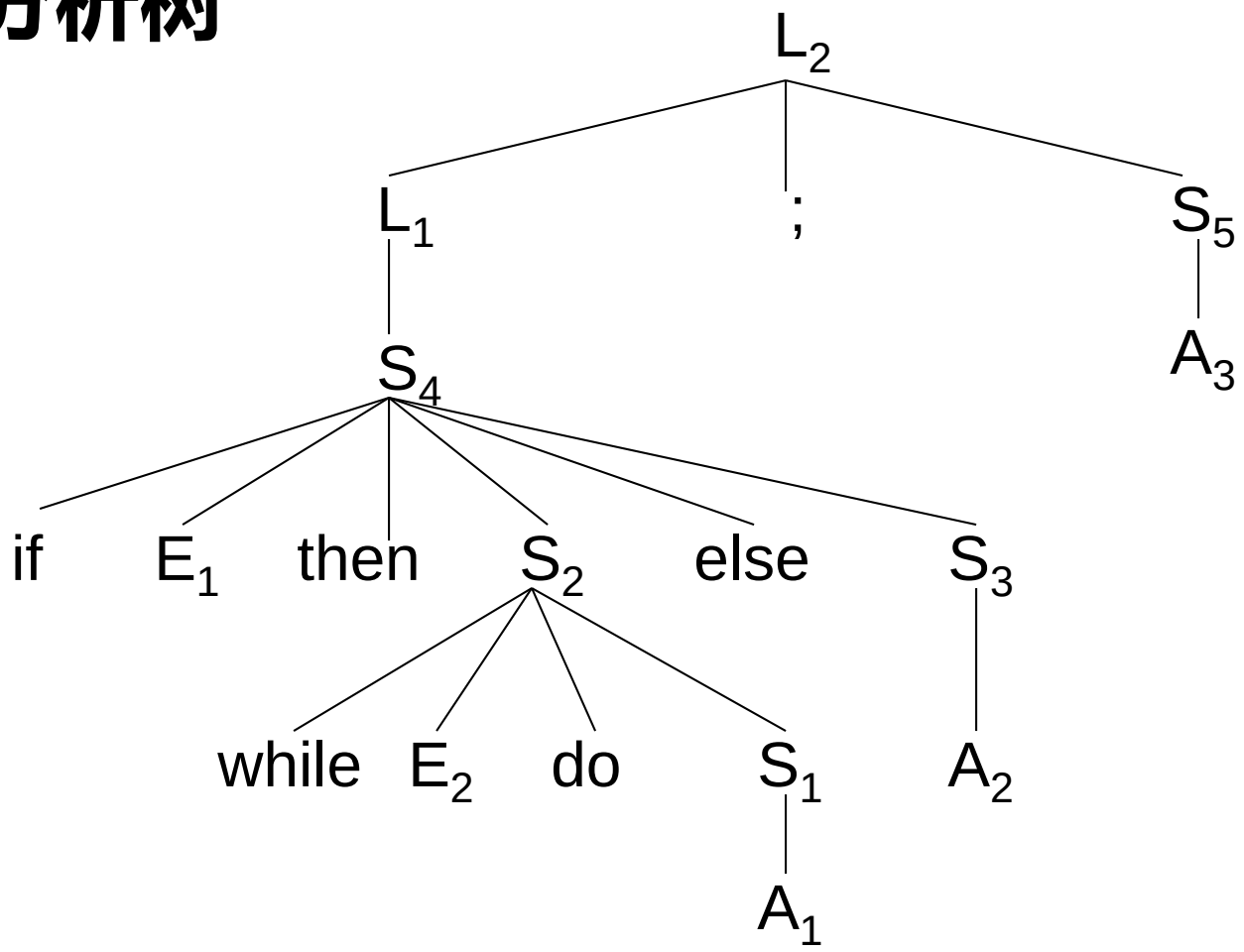
控制流语句的翻译

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

❑分析树



控制流语句的翻译

一、翻译 E_1：(a<b or c<d and e<f)

(100) if a<b goto 106

(101) goto 102　　//用102回填(101)

(102) if c<d goto 104　　//用104回填(102)

(103) goto 111

(104) if e<f goto 106

(105) goto 111

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

二、翻译 S_2： while E_2 do S_1

(106) if a>c goto 108 //用108回填(106)

(107) goto 112

(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

(110) goto 112 // 由N→ε生成

(111) d := d + 1 // $S_3 \rightarrow A_2$ S_3.nextlist={}

控制流语句的翻译

三、分析完S_4

- 用106回填(100)和(104)；用111回填(103)和(105)
- S_4.nextlist: { 107, 110 }

四、分析完L_1

- L_1.nextlist: { 107, 110 }

五、分析S_5

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

六、分析完L_2

- 用112回填(107)和(110)
- L_2.nextlist: {}

控制流语句的翻译

(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

开关语句的翻译

switch E

begin

case V_1: S_1

case V_2: S_2

. . .

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

default: S_n

end

开关语句的翻译

- 分支数较少时

```
        t = E的代码              | 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:    . . .
        . . .
```

开关语句的翻译

- 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的代码          | Ln: Sn的代码
        goto test            |     goto next
L1:     S1的代码             |test: if t == V1 goto L1
        goto next            |     if t == V2 goto L2
L2:     S2的代码             |     . . .
        goto next            |     if t == Vn-1 goto Ln-1
        . . .                |     goto Ln
Ln-1:   Sn-1的代码           | next:
        goto next
```

开关语句的翻译

- 中间代码增加一种case语句，便于代码生成器对它进行特别处理

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

一个生成：

- 用二分查找确定该执行的分支
- 直接找到该执行的分支

的例子见第244页习题8.8

过程调用的翻译

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

$Elist \rightarrow Elist, E$

$Elist \rightarrow E$

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

$E_1.place = E_1$的代码

$E_2.place = E_2$的代码

. . .

$E_n.place = E_n$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

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

{为长度为*n*的队列中的每个*E.place*，

emit(‘param’, *E.place*);

emit(‘call’, id.*place*, *n*) }

Elist $\rightarrow$ *Elist*, *E*

{把*E.place*放入队列末尾}

Elist $\rightarrow$ *E*

{将队列初始化，并让它仅含*E.place*}

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

Complexity has and will maintain a strong fascination for many people. It is true that we live in a complex world and strive to solve inherently complex problems, which often do require complex mechanisms. However, this should not diminish our desire for elegant solutions, which convince by their clarity and effectiveness. Simple, elegant solutions are more effective, but they are harder to find than complex ones, and they require more time, which we too often believe to be unaffordable. ——Niklaus Wirth (Turing award 1984)