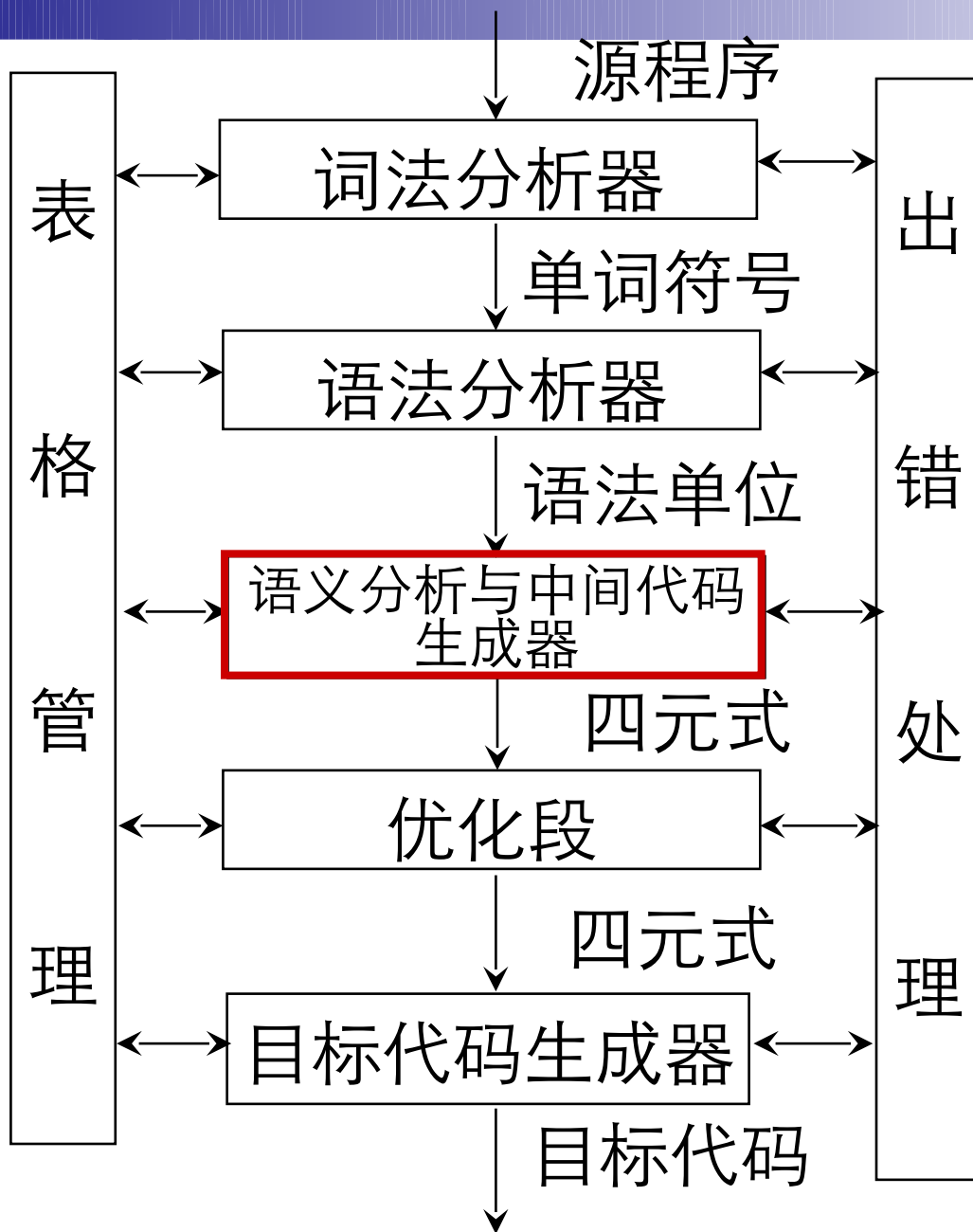




编译原理

第七章 语义分析和中间代码产生

编译程序总框

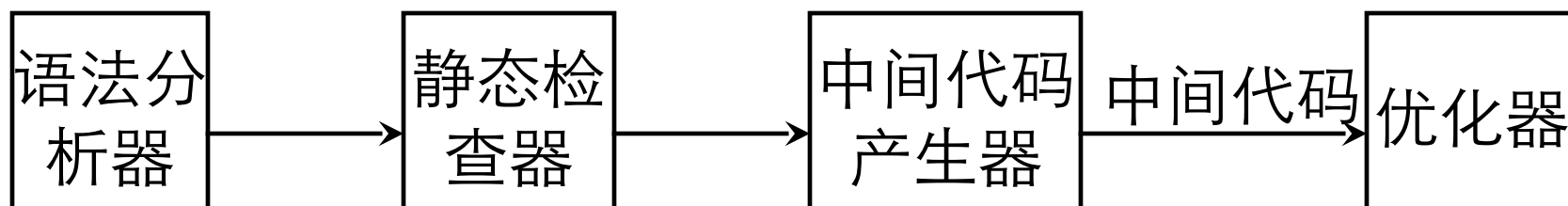


第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

第七章 语义分析和中间代码产生

- 静态语义检查
 - 类型检查
 - 控制流检查
 - 一致性检查
 - 相关名字检查
 - 名字的作用域分析

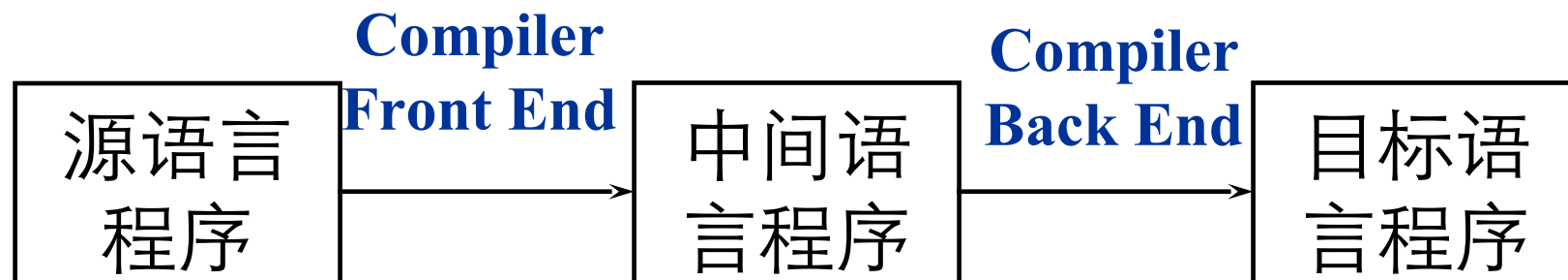


■ 中间语言

- 独立于机器
- 复杂性介于源语言和目标语言之间

■ 引入中间语言的优点

- 便于进行与机器无关的代码优化工作
- 易于移植
- 使编译程序的结构在逻辑上更为简单明确



7.1 中间语言

- 常用的中间语言

- 后缀式，逆波兰表示
- 图表示： DAG、抽象语法树
- 三地址代码
 - 三元式
 - 四元式
 - 间接三元式

7.1.1 后缀式

- **后缀式**表示法：Lukasiewicz 发明的一种表示表达式的方法，又称**逆波兰**表示法。
- 一个表达式 E 的后缀形式可以如下定义
 - 如果 E 是一个变量或常量，则 E 的后缀式是 E 自身。
 - 如果 E 是 $E_1 \text{ op } E_2$ 形式的表达式，其中 op 是任何二元操作符，则 E 的后缀式为 $E_1' E_2' \text{ op}$ ，其中 E_1' 和 E_2' 分别为 E_1 和 E_2 的后缀式。
 - 如果 E 是 (E_1) 形式的表达式，则 E_1 的后缀式就是 E 的后缀式。

后缀式

■ 逆波兰表示法不用括号

- 只要知道每个算符的目数，对于后缀式，不论从哪一端进行扫描，都能对它进行唯一分解。

■ 后缀式的计算

- 用一个栈实现
- 自左至右扫描后缀式，每碰到运算量就把它推进栈。每碰到 k 目运算符就把它作用于栈顶的 k 个项，并用运算结果代替这 k 个项。

将表达式翻译成后缀式的语义规则

产生式

$E \rightarrow E^{(1)} \text{op} E^{(2)}$

$E \rightarrow (E^{(1)})$

$E \rightarrow \text{id}$

语义规则

$E.\text{code} := E^{(1)}.\text{code} \mid \mid E^{(2)}.\text{code} \mid \mid$

op

$E.\text{code} := E^{(1)}.\text{code}$

$E.\text{code} := \text{id}$

- $E.\text{code}$ 表示 E 后缀形式
- op 表示任意二元操作符
- “ $\mid \mid$ ” 表示后缀形式的连接

$E \rightarrow E^{(1)} \text{op } E^{(2)}$	$E.\text{code} := E^{(1)}.\text{code} \parallel E^{(2)}.\text{code} \parallel \text{op}$
$E \rightarrow (E^{(1)})$	$E.\text{code} := E^{(1)}.\text{code}$
$E \rightarrow \text{id}$	$E.\text{code} := \text{id}$

- 数组 POST 存放后缀式：k 为下标，初值为 1

- 上述语义规则可实现为：

产生式

程序段

$E \rightarrow E^{(1)} \text{op } E^{(2)} \{ \text{POST}[k] := \text{op}; k := k + 1 \}$

$E \rightarrow (E^{(1)}) \{ \}$

$E \rightarrow i \{ \text{POST}[k] := i; k := k + 1 \}$

- 例：输入串 $a+b+c$ 的分析和翻译

POST :

a^1	b^2	$+^3$	c^4	$+^5$...
-------	-------	-------	-------	-------	-----

7.1.2 图表示法

- 图表示法

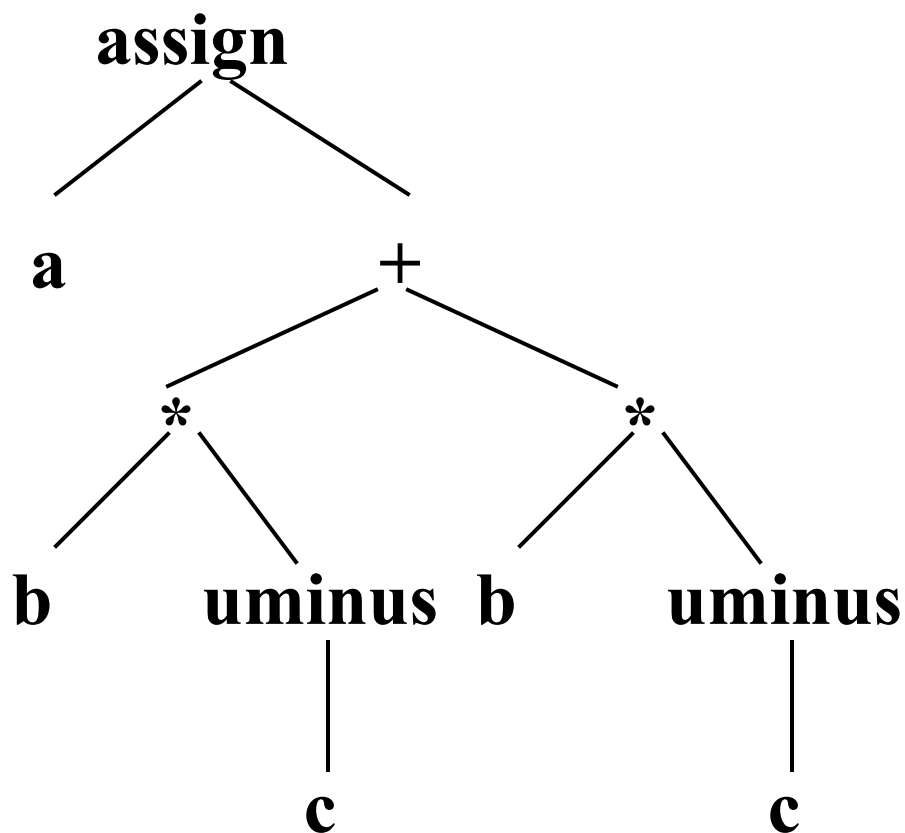
- DAG

- 抽象语法树

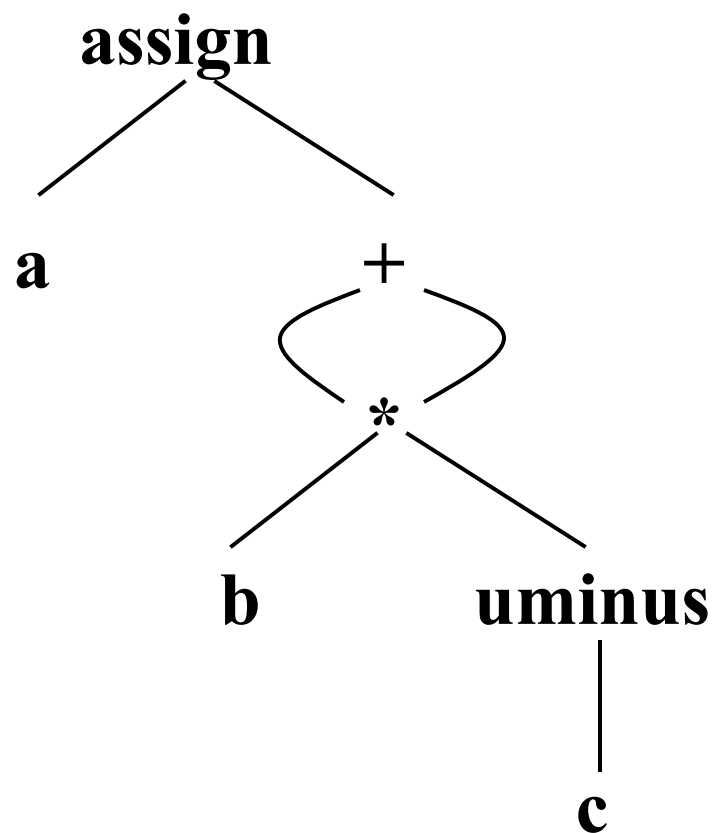
无循环有向图 (DAG)

- 无循环有向图 (Directed Acyclic Graph , 简称 DAG)
 - 对表达式中的每个子表达式, DAG 中都有一个结点
 - 一个内部结点代表一个操作符, 它的孩子代表操作数
 - 在一个 DAG 中代表公共子表达式的结点具有多个父结点

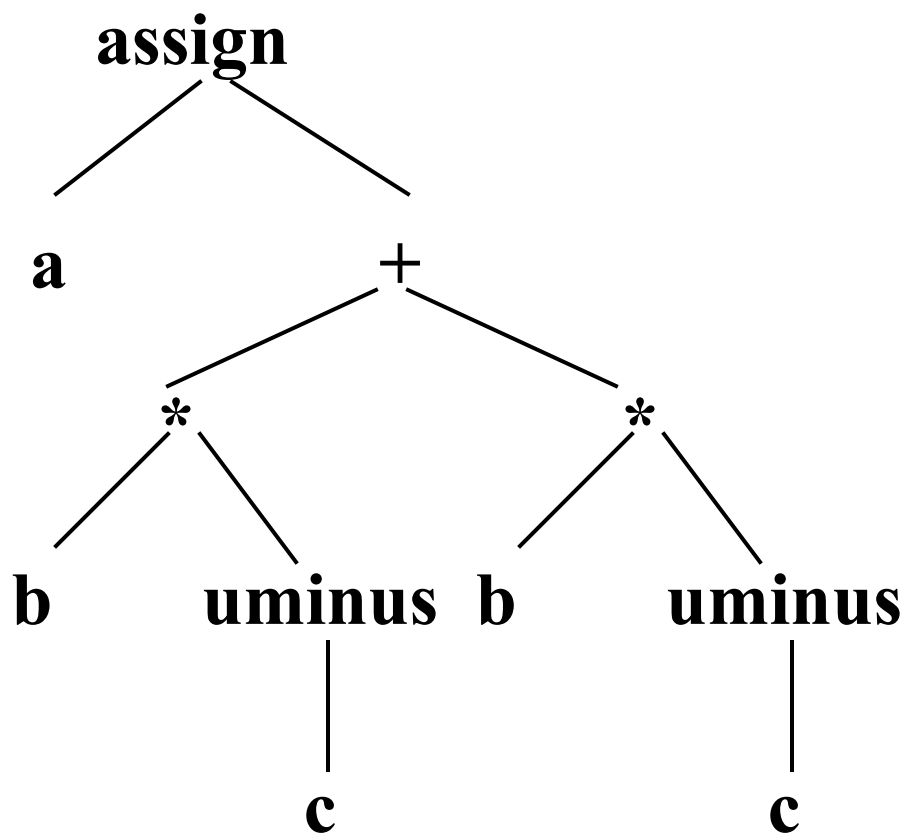
$a := b * (-c) + b * (-c)$ 的图表示法



抽象语法树



DAG



抽象语法树

抽象语法树对应的代码：

$T_1 := -c$

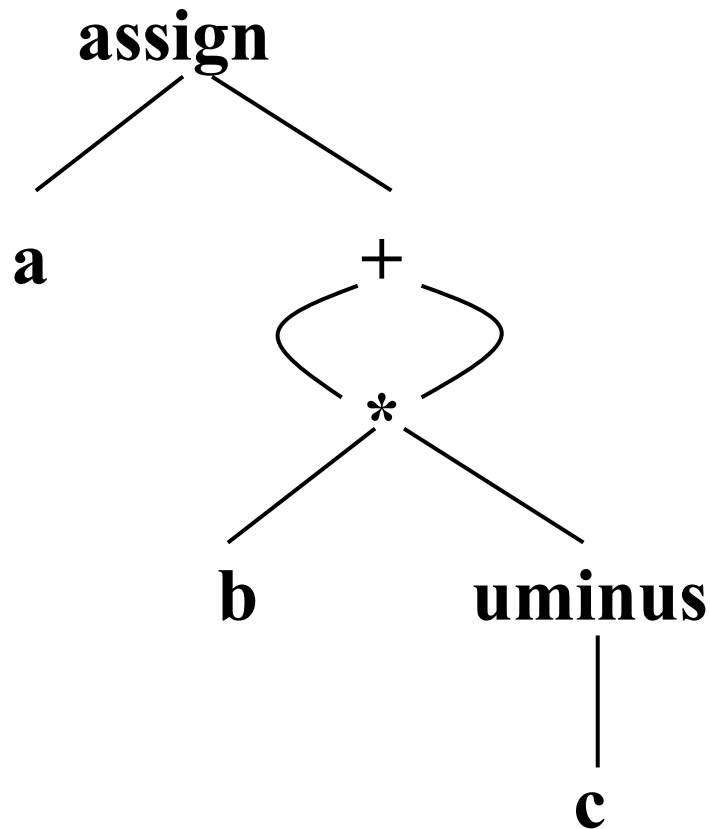
$T_2 := b * T_1$

$T_3 := -c$

$T_4 := b * T_3$

$T_5 := T_2 + T_4$

$a := T_5$



DAG

抽象语法树对应的代码:

$T_1 := -c$

$T_2 := b * T_1$

$T_3 := -c$

$T_4 := b * T_3$

$T_5 := T_2 + T_4$

$a := T_5$

DAG 对应的代码:

$T_1 := -c$

$T_2 := b * T_1$

$T_5 := T_2 + T_2$

$a := T_5$

产生赋值语句抽象语法树的属性文法

产生式

语义规则

$S \rightarrow id := E$	$S.nptr := mknnode('assign',$ $\quad mkleaf(id, id.place), E.nptr)$
$E \rightarrow E_1 + E_2$	$E.nptr := mknnode('+', E_1.nptr, E_2.nptr)$
$E \rightarrow E_1 * E_2$	$E.nptr := mknnode('*', E_1.nptr, E_2.nptr)$
$E \rightarrow -E_1$	$E.nptr := mknnode('uminus', E_1.nptr)$
$E \rightarrow (E_1)$	$E.nptr := E_1.nptr$
$E \rightarrow id$	$E.nptr := mkleaf(id, id.place)$

7.1.3 三地址代码

- 三地址代码

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

- 三地址代码可以看成是抽象语法树或 DAG 的一种线性表示

$a := b * (-c) + b * (-c)$ 的图表示法

DAG 对应的三地址代码:

$$T_1 := -c$$
$$T_2 := b * T_1$$
$$T_5 := T_2 + T_2$$
$$a := T_5$$

抽象语法树对应的三地址代码:

$$T_1 := -c$$
$$T_2 := b * T_1$$
$$T_3 := -c$$
$$T_4 := b * T_3$$
$$T_5 := T_2 + T_4$$
$$a := T_5$$

三地址语句的种类

- $x := y \text{ op } z$
- $x := \text{op } y$
- $x := y$
- `goto L`
- `if x relop y goto L` 或 `if a goto L`
- 传参、转子: `param x`、`call p,n`
- 返回语句: `return y`
- 索引赋值: $x := y[i]$ 、 $x[i] := y$
- 地址和指针赋值: $x := \&y$ 、 $x := *y$ 、 $*x := y$

三地址语句

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

■ 四元式

- 一个带有四个域的记录结构，这四个域分别称为 op, arg1, arg2 及 result

	<u>op</u>	<u>arg1</u>	<u>arg2</u>	<u>result</u>
(0)	uminus	c		T_1
(1)	*	b	T_1	T_2
(2)	uminus	c		T_3
(3)	*	b	T_3	T_4
(4)	+	T_2	T_4	T_5
(5)	:=	T_5		a

三地址语句

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

■ 三元式

- 三个域： op 、 arg1 和 arg2
- 引用临时变量（中间结果）：通过计算该值的语句的位置

	<u>op</u>	<u>arg1</u>	<u>arg2</u>
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

三地址语句

■ $x[i] := y$

	op	arg1	arg2
(0)	[] =	x	i
(1)	assign	(0)	y

■ $x := y[i]$

	op	arg1	arg2
(0)	= []	y	i
(1)	assign	x	(0)

三地址语句

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

■ 三元式

- 三个域： op 、 arg1 和 arg2
- 引用临时变量（中间结果）：通过计算该值的语句的位置

	<u>op</u>	<u>arg1</u>	<u>arg2</u>
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

三地址语句

- 间接三元式

- 三元式表 + 间接码表

- 间接码表

- 一张指示器表，按运算的先后次序列出有关三元式在三元式表中的位置

- 优点

- 方便优化，节省空间

- 例如，语句 $a:=b*(-c)+b*(-c)$ 的间接三元式表示如下表所示

三元式表				
<u>间接代码</u>		<u>op</u>	<u>arg1</u>	<u>arg2</u>
(0)	(0)	uminus	c	
(1)	(1)	*	b	(0)
(2)	(2)	+	(1)	(1)
(3)	(3)	assign	a	(2)

■ 例如，语句

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

$Y := D \uparrow (A + B)$

的间接三元式表示如下表所示

间接代码

三元式表

		OP	ARG1	ARG2
(1)				
(2)	(1)	+	A	B
(3)	(2)	*	(1)	C
(1)	(3)	:=	X	(2)
(4)	(4)	\uparrow	D	(1)
(5)	(5)	:=	Y	(4)

小结

- 常用的中间语言

- 后缀式，逆波兰表示
- 图表示： DAG、抽象语法树
- 三地址代码
 - 三元式
 - 四元式
 - 间接三元式

作业

- P217-1 , 3

例题 7.1.1 将下列语句翻译为逆波兰表示（后缀式），三元式和间接三元式序列和四元式表示：

$$a:=(b+c)*e+(b+c)/f$$

解题思路：

把中缀式转换后缀式的简单方法：按中缀式中各运算符的优先规则，从最先执行的部分开始写，一层层套。如对 $a \leq b+c \wedge a > d \vee a+b \neq e$ ，先把 $b+c$ 写为 $bc+$ ，然后把 $a \leq$ 套上去，成为 $abc+ \leq$ ；再把 $a > d$ 表示为 $ad >$ ，然后把 \wedge 套上去，成为 $abc+ \leq ad > \wedge$ ，依此类推。

四元式的由 4 个部分组成：算符 op 、第一和第二运算量 $arg1$ 和 $arg2$ ，以及运算结果 $result$ 。运算量和运算结果有时指用户自定义的变量，有时指编译程序引进的临时变量。如果 op 是一个算术或逻辑算符，则 $result$ 总是一个新引进的临时变量，用于存放运算结果。

三元式只需三个域： op 、 $arg1$ 和 $arg2$ 。与四元式相比，三元式避免了临时变量的填入，而是通过计算这个临时变量值的语句的位置来引用这个临时变量。我们很容易把一个算术表达式或一个赋值句表示为四元式序列或三元式序列。

间接三元式是指用一张间接码表辅以三元式表的办法来表示中间代码。间接码表按运算的先后顺序列出有关三元式在三元表中的位置，对于相同的三元式无需重复出现在三元表中。

解答

逆波兰表示为： $bc+e*bc+f/+:=$

三元式序列为：

- (1) $(+, b, c)$
- (2) $(*, (1), e)$
- (3) $(+, b, c)$
- (4) $(/, (3), f)$
- (5) $(+, (2), (4))$
- (6) $(:=, a, (5))$

间接三元式表示为：

三元式表	间接码表
(1) $(+, b, c)$	(1)
(2) $(*, \textcircled{1}, e)$	(2)
(3) $(/, \textcircled{3}, f)$	(1)
(4) $(+, \textcircled{2}, \textcircled{3})$	(3)
(5) $(:=, a, \textcircled{4})$	(4)

四元式表示为:

- (1) (+, b, c, T1)
- (2) (*, T1, e, T2)
- (3) (+, b, c, T3)
- (4) (/ , T3, f, T4)
- (5) (+, T2, T4, T5)
- (6) (:=, T5, -, a)

例题 7.1.2 利用回填技术把语句

```
while a>0 or b>0 do
```

```
    if c>0 and d<0 then x:=y+1;
```

翻译为三地址代码。

解题思路:

把表达式或赋值语句翻译为三地址代码是容易理解的, 如 $x:=y*z+1$ 翻译为:

```
T1:=y*z
```

```
T2:=T1+1
```

```
x:=T2
```

while 语句和 if 语句的翻译涉及到布尔表达式, 我们一并讨论。产生布尔表达式三地址代码的语义规则如表 6.3 所示。按表 6.3 的定义, 每个形如 $A \text{ relop } B$ 的表达式(其中 relop 为任一关系运算符)将翻译为如下两条转移指令:

```
if A relop B goto ...
```

```
goto ...
```

因此, 假定表达式的待确定的真假出口已分别为 Ltrue 和 Lfalse, 则 $a>0 \text{ or } b>0$ 将被翻译为

```
if a>0 goto Ltrue
```

```
goto L1
```

```
L1: if b>0 goto Ltrue
```

```
goto Lfalse
```

而 $c > 0$ and $d < 0$ 将被翻译为

```
if c>0 goto L3
```

```
goto Lfalse
```

```
L3: if d<0 goto Ltrue
```

```
goto Lfalse
```

有关 if 和 while 语句的属性文法如表 6.4 所示。

应用表 6.3 和表 6.4 不难生成含 if 和 while 的语句的三地址代码

解答：

所求三地址代码为：

```
L0: if a>0 goto L2
```

```
goto L1
```

```
L1: if b>0 goto L2
```

```
goto Lnext
```

```
L2: if c>0 goto L3
```

```
goto L0
```

```
L3: if d<0 goto L4
```

```
goto L0
```

```
L4: T1:=y + 1
```

```
x:= T1
```

```
goto L0
```

```
Lnext:
```

例题 7.1.3 把语句

```
while x>y do
```

if $x > 0$ then $x := x - 1$

else $y := y + 1$;

翻译为四元式序列。

解题思路：

因为三地址语句可看成中间代码的一种抽象形式, 而四元式是三地址代码语句的具体实现。因此, 上题介绍的语义规则及翻译方法可用于产生四元式。

我们也可通过适合语法制导翻译的语义子程序(或称翻译模式)来理解和翻译四元式。

If 语句和 While 语句的翻译模式如图 6.8 所示。根据此翻译模式, 可以把含 if 和 while 的语句翻译为四元式序列。

解答：

(1) $(j, x, y, 3)$

(2) $(j, -, -, 11)$

(3) $(j, x, 0, 5)$

(4) $(j, -, -, 8)$

(5) $(-, x, 1, T1)$

(6) $(:=, T1, -, x)$

(7) $(j, -, -, 1)$

(8) $(+, y, 1, T2)$

(9) $(:=, T2, -, y)$

(10) $(j, -, -, 1)$

一个表达式 E 的**后缀形式**可以如下定义：

- 1) 如果 E 是一个变量或常量，则 E 的后缀式是 E 自身。
- 2) 如果 E 是 $E_1 \text{ op } E_2$ 形式的表达式，其中 op 是任何二元操作符，则 E 的后缀式为 $E_1' E_2' \text{ op}$ ，其中 E_1' 和 E_2' 分别为 E_1 和 E_2 的后缀式。
- 3) 如果 E 是 (E_1) 形式的表达式，则 E_1 的后缀式就是 E 的后缀式。

无循环有向图 (Directed Acyclic Graph, 简称 DAG), 对表达式中的每个子表达式, DAG 中都有一个结点, 一个内部结点代表一个操作符, 它的孩子代表操作数, 在一个 DAG 中代表公共子表达式的结点具有多个父结点。

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

7.3 赋值语句的翻译

7.3.1 简单算术表达式及赋值语句

■ $id:=E$

- 对表达式 E 求值并置于变量 T 中值
- $id.place:=T$

从赋值语句生成三地址代码的 S- 属性文法

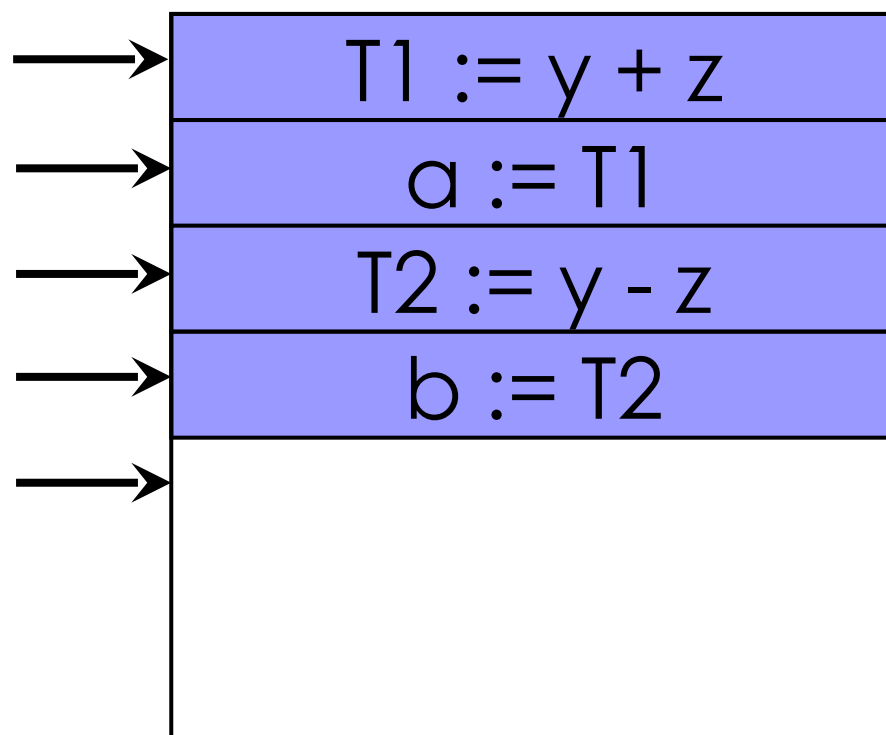
- 非终结符号 S 有综合属性 **S.code**，它代表赋值语句 S 的三地址代码
- 非终结符号 E 有如下两个属性
 - **E.place** 表示存放 E 值的名字
 - **E.code** 表示对 E 求值的三地址代码
 - 函数 **newtemp** 的功能是返回一个不同的临时变量名
- 计算思维的典型方法 -- 递归
 - 问题的解决又依赖于类似问题的解决，只不过后者的复杂程度或规模较原来的问题更小
 - 一旦将问题的复杂程度和规模化简到足够小时，问题的解法其实非常简单

为赋值语句生成三地址代码的 S- 属性文法定义

产生式	语义规则
$S \rightarrow id := E$	$S.code := E.code \parallel \text{gen}(id.place \text{ ':=' } E.place)$
$E \rightarrow E_1 + E_2$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel E_2.code \parallel$ $\text{gen}(E.place \text{ ':=' } E_1.place \text{ '+' } E_2.place)$
$E \rightarrow E_1 * E_2$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel E_2.code \parallel$ $\text{gen}(E.place \text{ ':=' } E_1.place \text{ '*' } E_2.place)$
$E \rightarrow -E_1$	$E.place := \text{newtemp};$ $E.code := E_1.code \parallel$ $\text{gen}(E.place \text{ ':=' 'uminus' } E_1.place)$
$E \rightarrow (E_1)$	$E.place := E_1.place;$ $E.code := E_1.code$
$E \rightarrow id$	$E.place := id.place;$

产生赋值语句三地址代码的翻译模式

- 过程 `emit` 将三地址代码送到输出文件中



$S \rightarrow id := E$ $S.code := E.code \parallel gen(id.place := E.place)$

$E \rightarrow E_1 + E_2$ $E.place := newtemp;$

$E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place + E_2.place)$

$E \rightarrow E_1 * E_2$ $E.place := newtemp;$

~~$E.code := E_1.code \parallel E_2.code \parallel gen(E.place := E_1.place * E_2.place)$~~

产生赋值语句三地址代码的翻译模式

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

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

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

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

产生赋值语句三地址代码的翻译模式

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

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

$a := uminus b$

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

7.3.2 数组元素的引用

- 数组元素地址的计算

- $X := A[i_1, i_2, \dots, i_k] + Y$

- $A[i_1, i_2, \dots, i_k] := X + Y$

数组元素地址计算

■ 设 A 为 n 维数组，按行存放，每个元素宽度为 w

□ low_i 为第 i 维的下界

□ up_i 为第 i 维的上界

□ n_i 为第 i 维可取值的个数 ($n_i = up_i - low_i + 1$)

□ $base$ 为 A 的第一个元素相对地址

■ 元素 $A[i_1, i_2, \dots, i_k]$ 相对地址公式

可变部分

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

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

不变部分

■ $C = ((\dots ((low_1 n_2 + low_2) n_3 + low_3) \dots) n_k + low_k) \times w$

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

$$\text{base} - ((\cdots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \cdots) n_k + \text{low}_k) \times w$$

- id 出现的地方也允许下面产生式中的 L 出现

$$L \rightarrow \text{id} [\text{Elist}] \mid \text{id}$$

$$\text{Elist} \rightarrow \text{Elist}, E \mid E$$

为了便于处理，文法改写为

$$L \rightarrow \text{Elist}] \mid \text{id}$$

$$\text{Elist} \rightarrow \text{Elist}, E \mid \text{id} [E$$

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

$\text{base} - ((\dots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \dots) n_k + \text{low}_k) \times w$

■ 引入下列语义变量或语义过程

- Elist.ndim: 下标个数计数器
- Elist.place: 表示临时变量，用来临时存放已形成的 Elist 中的下标表达式计算出来的值
- Elist.array: 记录数组名
- limit(array, j) : 函数过程，它给出数组 array 的第 j 维的长度

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

$$\text{base} - ((\cdots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \cdots) n_k + \text{low}_k) \times w$$


■ 每个代表变量的非终结符 L 有两项语义值

□ L.place

- 若 L 为简单变量 i, 指变量 i 的符号表入口
- 若 L 为下标变量, 指存放不变部分的临时变量的整数码

□ L.offset

- 若 L 为简单变量, null ,
- 若 L 为下标变量, 指存放可变部分的临时变量的整数码



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

(2) $E \rightarrow E + E$

(3) $E \rightarrow (E)$

(4) $E \rightarrow L$

(5) $L \rightarrow \text{Elist }]$

(6) $L \rightarrow \text{id}$

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

(8) $\text{Elist} \rightarrow \text{id } [E$

带数组元素引用的赋值语句翻译模式

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

{ if L.offset=null then /*L 是简单变量 */

emit(L.place ':=' E.place)

else emit(L.place '[' L.offset ']' ':=' E.place)}

(2) $E \rightarrow E_1 + E_2$

{ E.place:=newtemp;

emit(E.place ':=' E₁.place '+' E₂.place)}

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

(4) $E \rightarrow L$

```
{ if L.offset=null then
    E.place:=L.place
else begin
    E.place:=newtemp;
    emit(E.place ':=' L.place '[' L.offset ']' )
end
}
```


$A[i_1, i_2, \dots, i_k]$

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

$\text{base} - ((\dots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \dots) n_k + \text{low}_k) \times w$

(8) $\text{Elist} \rightarrow \text{id} [E$

$\{ \text{Elist.place} := E.\text{place};$

$\text{Elist.ndim} := 1;$

$\text{Elist.array} := \text{id.place} \}$

$A[i_1, i_2, \dots, i_k]$

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

$\text{base} - ((\dots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \dots) n_k + \text{low}_k) \times w$

(7) $\text{Elist} \rightarrow \text{Elist}_1, E$

{ $t := \text{newtemp};$

$m := \text{Elist}_1.\text{ndim} + 1;$

$\text{emit}(t := \text{Elist}_1.\text{place} * \text{limit}(\text{Elist}_1.\text{array}, m));$

$\text{emit}(t := t + E.\text{place});$

$\text{Elist.place} := t;$

$\text{Elist.ndim} := m$

$\text{Elist.array} := \text{Elist}_1.\text{array};$

}

$A[i_1, i_2, \dots, i_k]$

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

$\text{base} - ((\dots ((\text{low}_1 n_2 + \text{low}_2) n_3 + \text{low}_3) \dots) n_k + \text{low}_k) \times w$

(5) $L \rightarrow \text{Elist}$]

{ L.place := newtemp;

emit(L.place := Elist.array ' - ' C);

L.offset := newtemp;

emit(L.offset := w '*' Elist.place) }

(6) $L \rightarrow \text{id}$ { L.place := id.place;
L.offset := null }

类型转换

- 用 $E.type$ 表示非终结符 E 的类型属性
- 对应产生式 $E \rightarrow E_1 \text{ op } E_2$ 的语义动作中关于 $E.type$ 的语义规则可定义为：
 { if $E_1.type = \text{integer}$ and $E_2.type = \text{integer}$
 $E.type := \text{integer}$
 else $E.type := \text{real}$ }
- 算符区分为整型算符 int op 和实型算符 real
 op ,


■ $x := y + i*j$

其中 x 、 y 为实型； i 、 j 为整型。这个赋值句产生的三地址代码为：

$$T_1 := i \text{ int* } j$$
$$T_3 := \text{inttoreal } T_1$$
$$T_2 := y \text{ real+ } T_3$$
$$x := T_2$$

关于产生式 $E \rightarrow E_1 + E_2$ 的语义动作

```
{ E.place:=newtemp;  
  if  $E_1.type=integer$  and  $E_2.type=integer$  then  
    begin  
      emit (E.place ':='  $E_1.place$  'int+'  $E_2.place$ );  
      E.type:=integer  
    end  
  else if  $E_1.type=real$  and  $E_2.type=real$  then begin  
    emit (E.place ':='  $E_1.place$  'real+'  $E_2.place$ );  
    E.type:=real  
  end
```



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

小结

- 赋值语句的翻译
 - 简单算术表达式及赋值语句
 - 数组元素的引用
 - 产生有关类型转换的指令

作业

- P218-4 , 5
- 注: P218-5 , 设:
 - A 、 B : 10×20
 - C 、 D : 20
 - 宽度 $w = 4$
 - 下标从 1 开始

1.利用回填技术把语句

```
while a>0 or b>0 do
  if c>0 and d<0 then x:=y+1;
```

翻译为三地址代码。

(北京邮电大学 1998 年硕士生入学考试试题)

解题思路：

把表达式或赋值语句翻译为三地址代码是容易理解的, 如 $x:=y*z+1$ 翻译为:

```
T1:=y*z
T2:=T1+1
x:=T2
```

while 语句和 if 语句的翻译涉及到布尔表达式, 我们一并讨论。产生布尔表达式三地址代码的语义规则如表 5.4 所示。按表 5.4 的定义, 每个形如 $A \text{ relop } B$ 的表达式(其中 relop 为任一关系运算符)将翻译为如下两条转移指令:

```
if A relop B goto ---
goto ---
```

因此, 假定表达式的待确定的真假出口已分别为 Ltrue 和 Lfalse, 则 $a>0 \text{ or } b>0$ 将被翻译为

```
if a>0 goto Ltrue
goto L1
L1: if b>0 goto Ltrue
goto Lfalse
```

而 $c>0 \text{ and } d<0$ 将被翻译为

```
if c>0 goto L3
goto Lfalse
L3: if d<0 goto Ltrue
goto Lfalse
```

表 5.4 产生布尔表达式三地址代码的语义规则

产生式	语义规则
$E \rightarrow E_1 \text{ or } E_2$	$E_1.true := E.true;$ $E_1.false := newlabel;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code gen(E_1.false \text{ ':' }) E_2.code$
$E \rightarrow E_1 \text{ and } E_2$	$E_1.true := newlabel;$ $E_1.false := E.false;$ $E_2.true := E.true;$ $E_2.false := E.false;$ $E.code := E_1.code gen(E_1.true \text{ ':' }) E_2.code$
$E \rightarrow \text{not } E_1$	$E_1.true := E.false;$ $E_1.false := E.true;$

	E.code:=E ₁ .code
E→(E ₁)	E ₁ .true:=E.true; E ₁ .false:=E.false; E.code:=E ₁ .code
E→id ₁ relop id ₂	E.code:=gen('if ' id ₁ .place relop.op id ₂ .place 'goto' E.true) gen('goto' E.false)
E→true	E.code:=gen('goto' E.true)
E→false	E.code:=gen('goto' E.false)

if 和 while 语句的属性文法如表 5.5 所示。应用表 5.4 和表 5.5 不难生成含 if 和 while 的语句的三地址代码

表 5.5 if 和 while 语句的属性文法

产生式	语义规则
S→if E then S ₁	E.true:=newlabel; E.false:=S.next; S ₁ .next:=S.next S.code:=E.code gen(E.true ':') S ₁ .code
S→if E then S ₁ else S ₂	E.true:=newlabel; E.false:=newlabel; S ₁ .next:=S.next S ₂ .next:=S.next; S.code:=E.code gen(E.true ':') S ₁ .code gen('goto' S.next) gen(E.false ':') S ₂ .code
S→while E do S ₁	S.begin:=newlabel; E.true:=newlabel; E.false:=S.next; S ₁ .next:=S.begin; S.code:=gen(S.begin ':') E.code gen(S.true ':') S ₁ .code gen('goto' S.begin)

解答：

三地址代码为：

L0: if a>0 goto L2

goto L1

L1: if b>0 goto L2

```
    goto Lnext
L2: if c>0 goto L3
    goto L0
L3: if d<0 goto L4
    goto L0
L4: T1:=y + 1
    x:= T1
    goto L0
Lnext:
```



编译原理

第七章 语义分析和中间代码产生

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

7.4 布尔表达式的翻译

- 布尔表达式的两个基本作用

- 用于逻辑演算，计算逻辑值
- 用于控制语句的条件式

- 产生布尔表达式的文法

$$E \rightarrow E \text{ or } E \mid E \text{ and } E \mid \text{not } E \mid (E) \mid i \text{ rop } i \mid i$$

计算布尔表达式的两种方法

- 如同计算算术表达式一样，一步步算

$1 \text{ or } (\text{not } 0 \text{ and } 0) \text{ or } 0$
 $= 1 \text{ or } (1 \text{ and } 0) \text{ or } 0$
 $= 1 \text{ or } 0 \text{ or } 0$
 $= 1 \text{ or } 0$
 $= 1$

- 采用某种优化措施

- ☐ 把 $A \text{ or } B$ 解释成
B

if A then true else

- ☐ 把 $A \text{ and } B$ 解释成

if A then B else false

- ☐ 把 $\text{not } A$ 解释成

if A then false else true

两种不同的翻译方法

- 第一种翻译法

A or B and C=D 翻译成

(1) (=, C, D, T_1)

(2) (and, B, T_1 , T_2)

(3) (or, A, T_2 , T_3)

- 第二种翻译法适合于作为条件表达式的布尔表达式使用

7.4.1 数值表示法

- a or b and not c 翻译成

$T_1 := \text{not } c$

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

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

- $a < b$ 的关系表达式可等价地写成
if $a < b$ then 1 else 0 , 翻译成

100: if $a < b$ goto 103

101: $T := 0$

102: goto 104

103: $T := 1$

104:

关于布尔表达式的数值表示法的翻译模式

- 过程 `emit` 将三地址代码送到输出文件中
- `nextstat` 给出输出序列中下一条三地址语句的地址索引
- 每产生一条三地址语句后，过程 `emit` 便把 `nextstat` 加 1

关于布尔表达式的数值表示法的翻译模式

$$E \rightarrow E_1 \text{ or } E_2 \quad \{E.\text{place} := \text{newtemp}; \\ \text{emit}(E.\text{place} \text{ ':=' } E_1.\text{place} \text{ 'or' } E_2.\text{place})\}$$
$$E \rightarrow E_1 \text{ and } E_2 \quad \{E.\text{place} := \text{newtemp};$$
$$\quad \text{emit}(E.\text{place} \text{ ':=' } E_1.\text{place} \text{ 'and' } E_2.\text{place})\}$$

```
E → not E1      {E.place := newtemp;  
                    emit(E.place ':= ' 'not' E1.place)}
```

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

关于布尔表达式的数值表示法的翻译模式

a<b 翻译成
100: if a<b goto 103
101: T:=0
102: goto 104
103: T:=1
104:

$E \rightarrow id_1 \text{ relop } id_2 \quad \{ E.place := newtemp;$
 $emit('if' \ id_1.place \ relop.op \ id_2.place$
 $'goto' \ nextstat+3);$
 $emit(E.place \ ':=' \ '0');$
 $emit('goto' \ nextstat+2);$
 $emit(E.place \ ':=' \ '1') \}$

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

布尔表达式 $a < b$ or $c < d$ and $e < f$ 的翻译结果

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

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

```
E → id
{ E.place:=id.place }
```

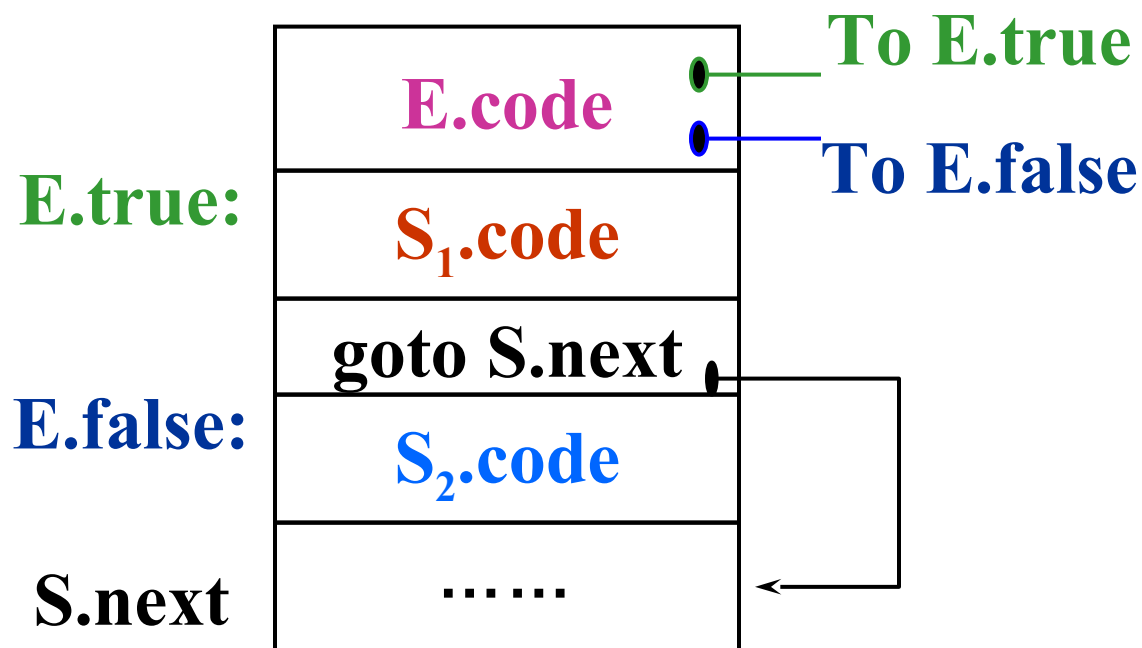
```
E → E1 or E2
{ E.place:=newtemp;
  emit(E.place ':=' E1.place 'or' E2.place) }
```

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

7.4.2 作为条件控制的布尔式翻译

- 条件语句 `if E then S1 else S2`

赋予 `E` 两种出口：一真一假



- 例：把语句：if $a > c$ or $b < d$ then S_1 else S_2
翻译成如下的一串三地址代码

	if $a > c$ goto L2	“真”出口
	goto L1	
L1:	if $b < d$ goto L2	“真”出口
	goto L3	“假”出口
L2:	(关于 S_1 的三地址代码序列)	
	goto Lnext	
L3:	(关于 S_2 的三地址代码序列)	

Lnext:

产生布尔表达式三地址代码的语义规则

- 每次调用函数 `newlabel` 后都返回一个新的符号标号
- 对于一个布尔表达式 E ，引用两个标号
 - $E.true$ 是 E 为‘真’时控制流转向的标号
 - $E.false$ 是 E 为‘假’时控制流转向的标号

产生布尔表达式三地址代码的语义规则

产生式

$E \rightarrow E_1 \text{ or } E_2$

语义规则

$E_1.\text{true} := E.\text{true};$

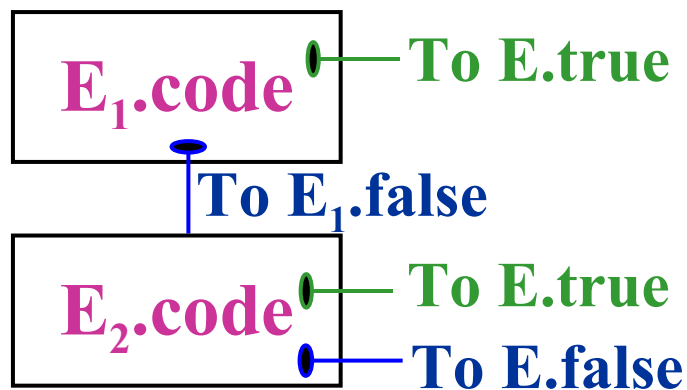
$E_1.\text{false} := \text{newlabel};$

$E_2.\text{true} := E.\text{true};$

$E_2.\text{false} := E.\text{false};$

$E.\text{code} := E_1.\text{code} \parallel$

$\text{gen}(E_1.\text{false} ':') \parallel E_2.\text{code}$



产生布尔表达式三地址代码的语义规则

产生式

$E \rightarrow E_1 \text{ and } E_2$

语义规则

$E_1.\text{true} := \text{newlabel};$

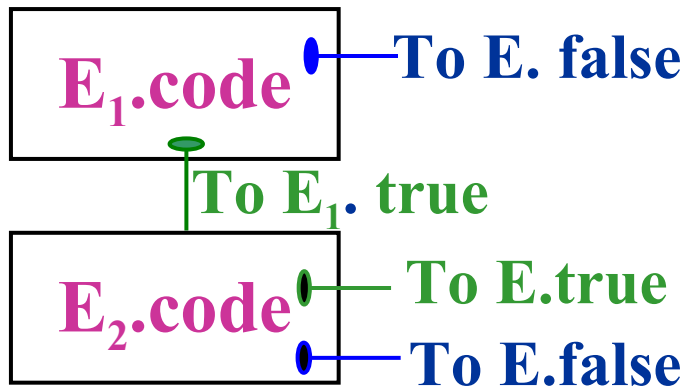
$E_1.\text{false} := E.\text{false};$

$E_2.\text{true} := E.\text{true};$

$E_2.\text{false} := E.\text{false};$

$E.\text{code} := E_1.\text{code} \parallel$

$\text{gen}(E_1.\text{true} ':') \parallel E_2.\text{code}$



产生布尔表达式三地址代码的语义规则

产生式

$E \rightarrow \text{not } E_1$

$E \rightarrow (E_1)$

语义规则

$E_1.\text{true} := E.\text{false};$
 $E_1.\text{false} := E.\text{true};$
 $E.\text{code} := E_1.\text{code}$

$E_1.\text{true} := E.\text{true};$
 $E_1.\text{false} := E.\text{false};$
 $E.\text{code} := E_1.\text{code}$

产生布尔表达式三地址代码的语义规则

产生式

语义规则

$E \rightarrow id_1 \text{ relop } id_2$	$E.code := \text{gen}(\text{'if ' } id_1.place$ $\text{relop.op } id_2.place \text{'goto' } E.true) \parallel$ $\text{gen('goto' } E.false)$
--	--

$E \rightarrow \text{true}$	$E.code := \text{gen('goto' } E.true)$
-----------------------------	--

$E \rightarrow \text{false}$	$E.code := \text{gen('goto' } E.false)$
------------------------------	---

考虑如下表达式:

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

产生式

$E \rightarrow id_1 \text{ relop } id_2$

语义规则

$E.code := \text{gen}(\text{'if ' } id_1.place$
 $\text{relop.op } id_2.place \text{'goto' } E.true) \parallel$
 $\text{gen('goto' } E.false)$

$E \rightarrow E_1 \text{ or } E_2$

$E_1.true := E.true;$
 $E_1.false := \text{newlabel};$
 $E_2.true := E.true;$
 $E_2.false := E.false;$
 $E.code := E_1.code \parallel \text{gen}(E_1.false \text{' : '}) \parallel E_2.code$

$E \rightarrow E_1 \text{ and } E_2$

$E_1.true := \text{newlabel};$
 $E_1.false := E.false;$
 $E_2.true := E.true;$
 $E_2.false := E.false;$

语义规则

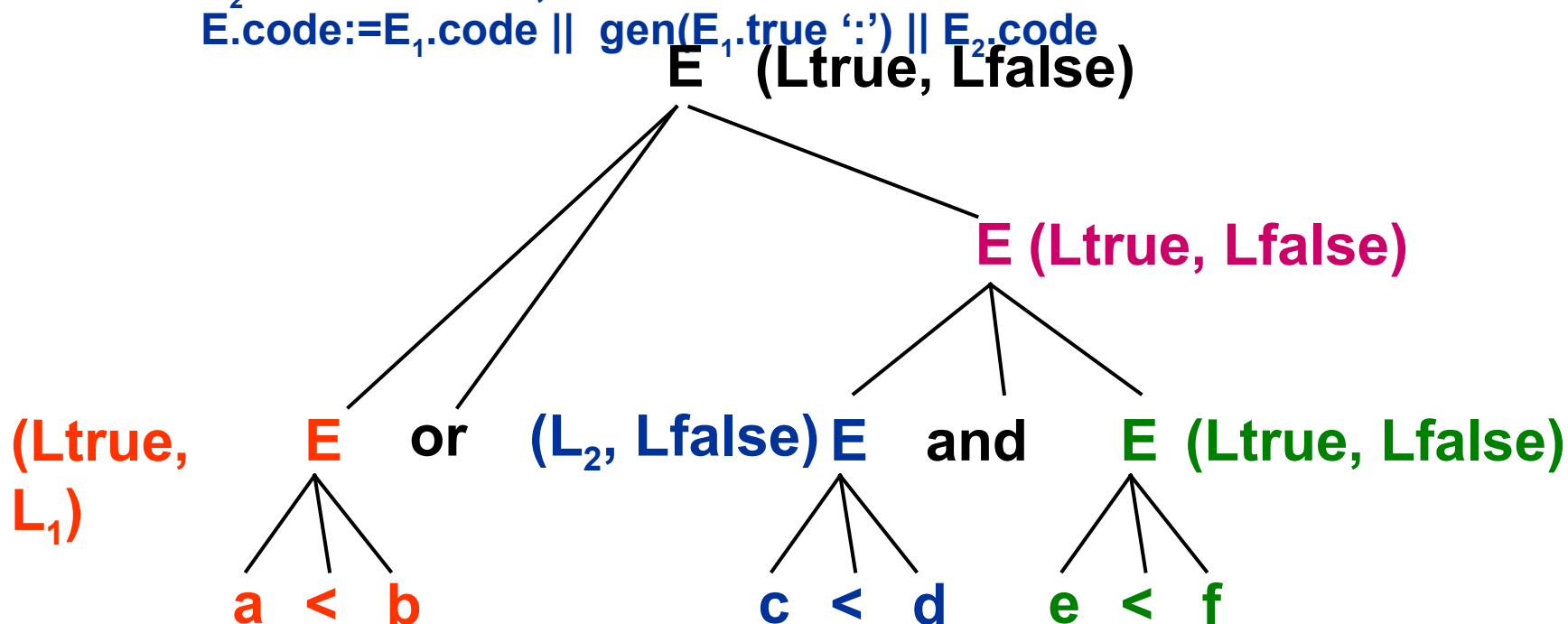
产生式

$E \rightarrow E_1 \text{ or } E_2$

$E_1.\text{true} := E.\text{true};$
 $E_1.\text{false} := \text{newlabel};$
 $E_2.\text{true} := E.\text{true};$
 $E_2.\text{false} := E.\text{false};$
 $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{false} ':') \parallel E_2.\text{code}$

$E \rightarrow E_1 \text{ and } E_2$

$E_1.\text{true} := \text{newlabel};$
 $E_1.\text{false} := E.\text{false};$
 $E_2.\text{true} := E.\text{true};$
 $E_2.\text{false} := E.\text{false};$
 $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{true} ':') \parallel E_2.\text{code}$



产生式

语义规则

$E \rightarrow id_1 \text{ relop } id_2$

$E.code := \text{gen}(\text{'if ' } id_1.place \text{ relop.op } id_2.place \text{ 'goto' } E.true) \parallel \text{gen}(\text{'goto' } E.false)$

$E \rightarrow E_1 \text{ or } E_2$

$E_1.true := E.true;$
 $E_1.false := \text{newlabel};$
 $E_2.true := E.true;$
 $E_2.false := E.false;$
 $E.code := E_1.code \parallel \text{gen}(E_1.false ':') \parallel E_2.code$

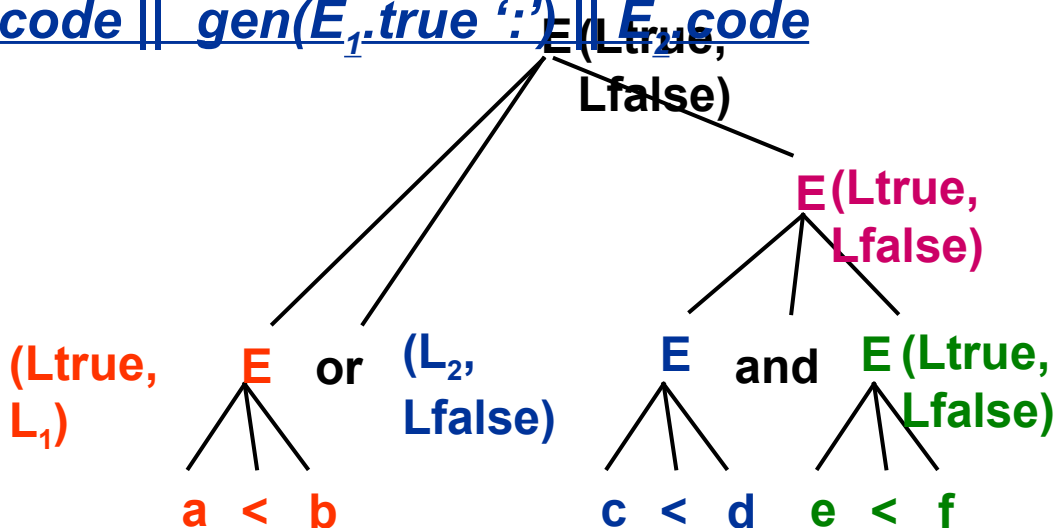
$E \rightarrow E_1 \text{ and } E_2$

$E_1.true := \text{newlabel};$
 $E_1.false := E.false;$
 $E_2.true := E.true;$
 $E_2.false := E.false;$
 $E.code := E_1.code \parallel \text{gen}(E_1.true ':') \parallel E_2.code$

if a < b goto Ltrue
 goto L₁

L₁: if c < d goto L₂
 goto Lfalse

L₂: if e < f goto Ltrue
 goto Lfalse



小结

- 布尔表达式的翻译
 - 数值表示法
 - 作为条件控制的布尔式翻译



编译原理

第七章 语义分析和中间代码产生

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

布尔表达式的翻译

- 两（多）遍扫描

- 为给定的输入串构造一棵语法树
 - 遍历语法树，进行语义规则中规定的翻译

考虑如下表达式:

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

产生式

$E \rightarrow id_1 \text{ relop } id_2$

语义规则

$E.code := \text{gen}(\text{'if ' } id_1.place$
 $\text{relop.op } id_2.place \text{'goto' } E.true) \parallel$
 $\text{gen('goto' } E.false)$

$E \rightarrow E_1 \text{ or } E_2$

$E_1.true := E.true;$
 $E_1.false := \text{newlabel};$
 $E_2.true := E.true;$
 $E_2.false := E.false;$
 $E.code := E_1.code \parallel \text{gen}(E_1.false \text{' : '}) \parallel E_2.code$

$E \rightarrow E_1 \text{ and } E_2$

$E_1.true := \text{newlabel};$
 $E_1.false := E.false;$
 $E_2.true := E.true;$
 $E_2.false := E.false;$

语义规则

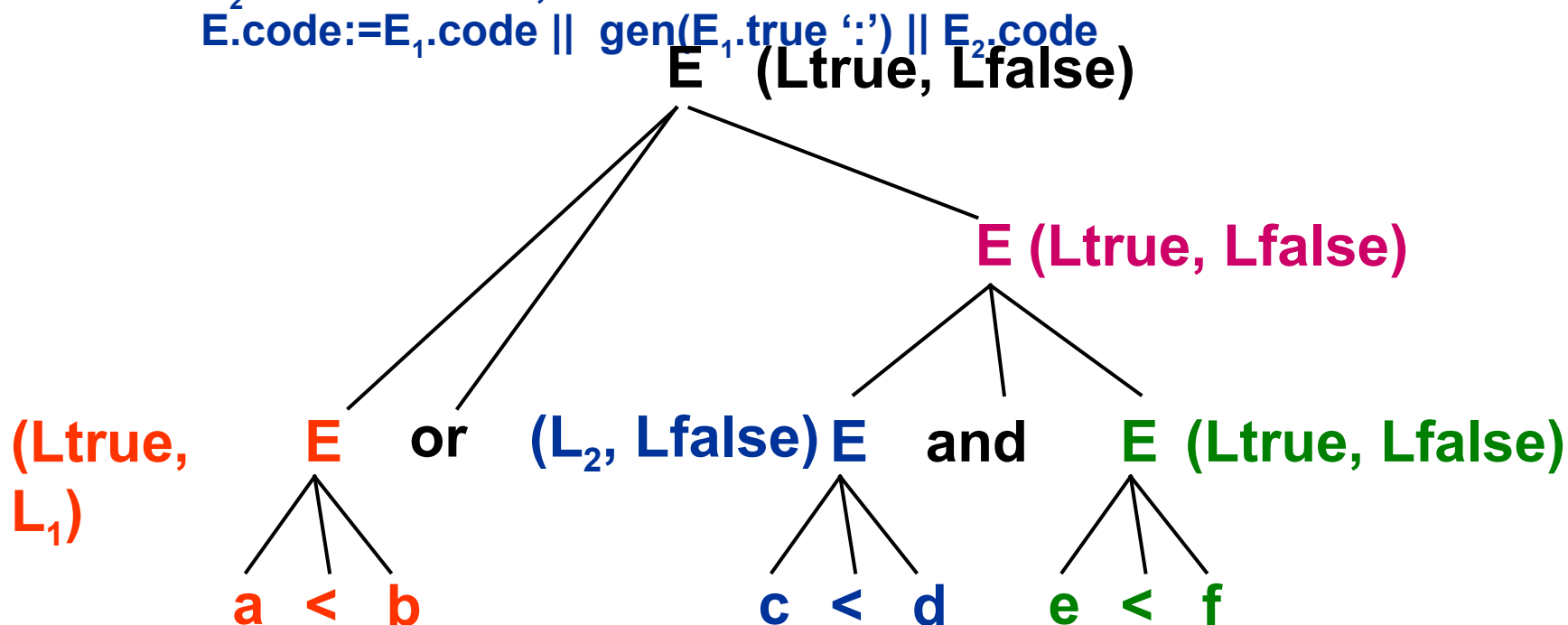
产生式

$E \rightarrow E_1 \text{ or } E_2$

$E_1.\text{true} := E.\text{true};$
 $E_1.\text{false} := \text{newlabel};$
 $E_2.\text{true} := E.\text{true};$
 $E_2.\text{false} := E.\text{false};$
 $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{false} ':') \parallel E_2.\text{code}$

$E \rightarrow E_1 \text{ and } E_2$

$E_1.\text{true} := \text{newlabel};$
 $E_1.\text{false} := E.\text{false};$
 $E_2.\text{true} := E.\text{true};$
 $E_2.\text{false} := E.\text{false};$
 $E.\text{code} := E_1.\text{code} \parallel \text{gen}(E_1.\text{true} ':') \parallel E_2.\text{code}$



产生式

语义规则

$E \rightarrow id_1 \text{ relop } id_2$

$E.code := \text{gen}(\text{'if ' } id_1.place$
 $\text{relop.op } id_2.place \text{'goto' } E.true) \parallel$
 $\text{gen('goto' } E.false)$

$E \rightarrow E_1 \text{ or } E_2$

$E_1.true := E.true;$
 $E_1.false := \text{newlabel};$
 $E_2.true := E.true;$
 $E_2.false := E.false;$
 $E.code := E_1.code \parallel \text{gen}(E_1.false \text{' : '}) \parallel E_2.code$

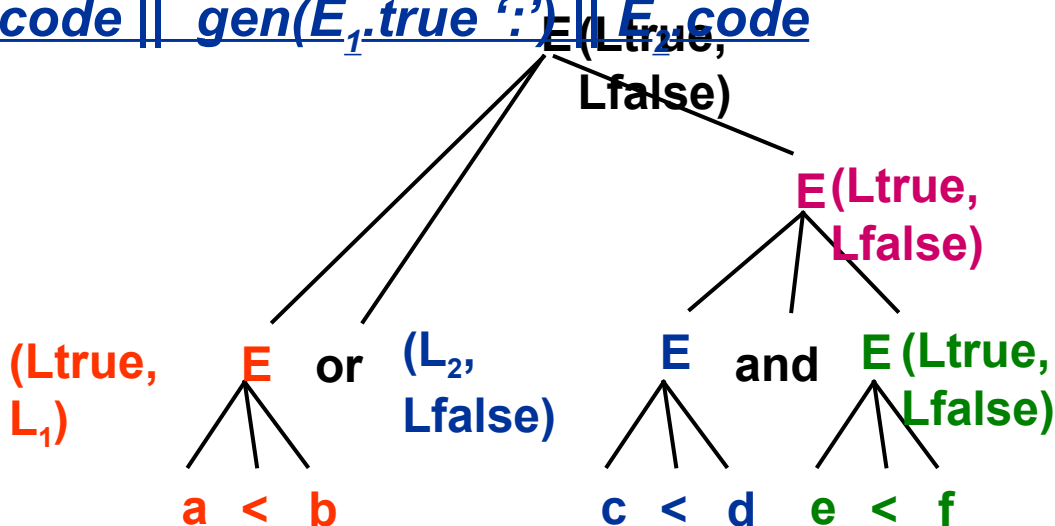
$E \rightarrow E_1 \text{ and } E_2$

$E_1.true := \text{newlabel};$
 $E_1.false := E.false;$
 $E_2.true := E.true;$
 $E_2.false := E.false;$
 $E.code := E_1.code \parallel \text{gen}(E_1.true \text{' : '}) \parallel E_2.code$

if a < b goto Ltrue
 goto L₁

L₁: if c < d goto L₂
 goto Lfalse

L₂: if e < f goto Ltrue
 goto Lfalse



布尔表达式的翻译

- 两（多）遍扫描

- 为给定的输入串构造一棵语法树
 - 遍历语法树，进行语义规则中规定的翻译

- 一遍扫描

一遍扫描实现布尔表达式的翻译

- 采用四元式形式
- 把四元式存入一个数组中，数组下标就代表四元式的标号
- 约定

四元式 (jnz, a, -, p) 表示 if a goto p

四元式 (jrop, x, y, p) 表示 if x rop y goto p

四元式 (j, -, -, p) 表示 goto p

一遍扫描实现布尔表达式的翻译

- 过程 emit 将四元式代码送到输出文件中

→	100	(j<, a, b, 104)
→	101	(j, -, -, 102)
→	102	(j<, c, d, 104)
→	103	(j, -, -, 110)
→	104	...
		...
→	110	...

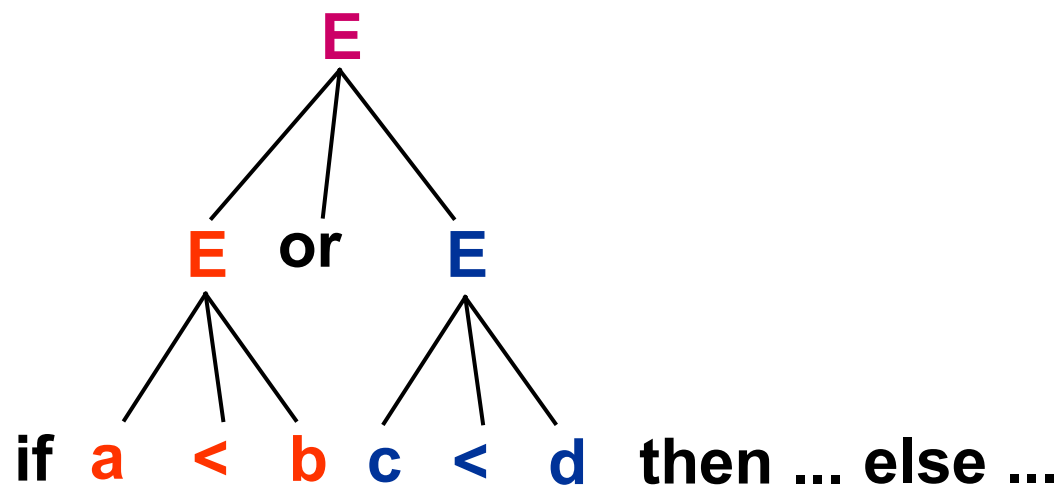
$a < b$ or $c < d$

■ 最大的困难？

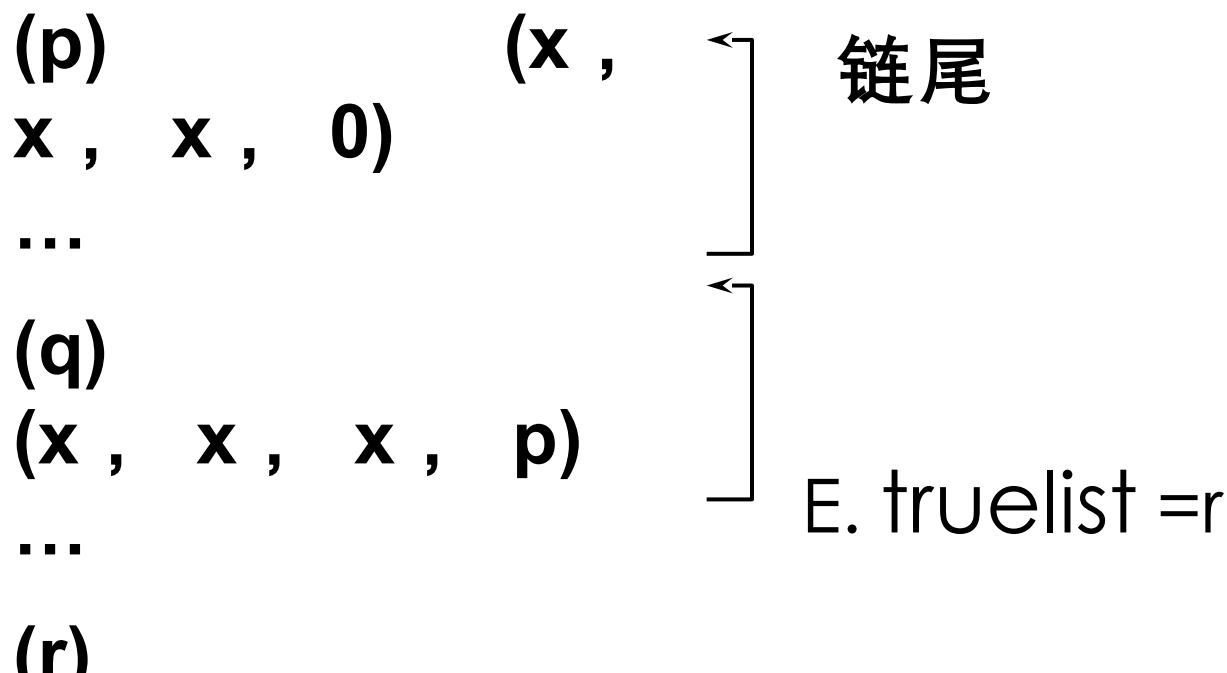
- 产生跳转四元式时，它的转移地址无法立即知道
- 需要以后扫描到特定位置时才能回过头来确定

■ 把这个未完成的四元式地址作为 E 的语义值保存，待机“回填”

100	(j<, a, b, 104)
101	(j, -, -, 102)
102	(j<, c, d, 104)
103	(j, -, -, 110)
104	...
	...
110	...



- 为非终结符 E 赋予两个综合属性 $E.truelist$ 和 $E.falselist$ 。它们分别记录布尔表达式 E 所应的四元式中需回填“真”、“假”出口的四元式的标号所构成的链表
- 例如：假定 E 的四元式中需要回填“真”出口的 p ， q ， r 三个四元式，则 $E.truelist$ 为下列链：



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

- 变量 **nextquad** ，它指向下一条将要产生但尚未形成的四元式的地址（标号）。nextquad 的初值为 1 ，每当执行一次 emit 之后，nextquad 将自动增 1 。
- 函数 **makelist(i)** ，它将创建一个仅含 i 的新链表，其中 i 是四元式数组的一个下标（标号）；函数返回指向这个链的指针。
- 函数 **merge(p_1, p_2)** ，把以 p_1 和 p_2 为链首的两条链合并为一，作为函数值，回送合并后的链首。
- 过程 **backpatch(p, t)** ，其功能是完成“回填”，把 p 所链接的每个四元式的第四区段都填为 t 。 13

过程 `backpatch(p, t)`，其功能是完成
“回填”，把 `p` 所链接的每个四元式的第
四区段都填为 `t`。

(r) (`t` ,
`x` , `x` , `0`)

... `t`

(q)
(`x` , `x` , `x` , | `t`

...

(p)
(`x` , `x` , `x` , `q`)

布尔表达式的文法

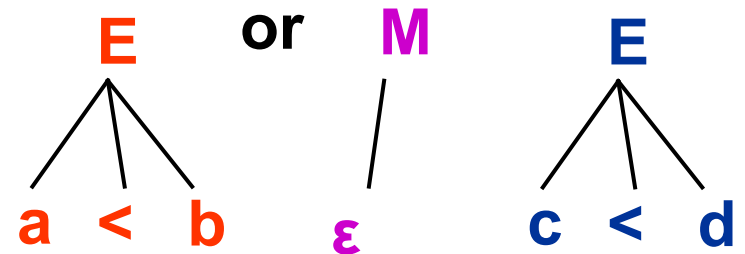
- (1) $E \rightarrow E_1 \text{ or } M E_2$
- (2) $\quad \quad \quad | E_1 \text{ and } M E_2$
- (3) $\quad \quad \quad | \text{ not } E_1$
- (4) $\quad \quad \quad | (E_1)$
- (5) $\quad \quad \quad | \text{ id}_1 \text{ relop id}_2$
- (6) $\quad \quad \quad | \text{ id}$
- (7) $M \rightarrow \varepsilon$

布尔表达式的翻译模式

(7) $M \rightarrow \varepsilon$

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

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



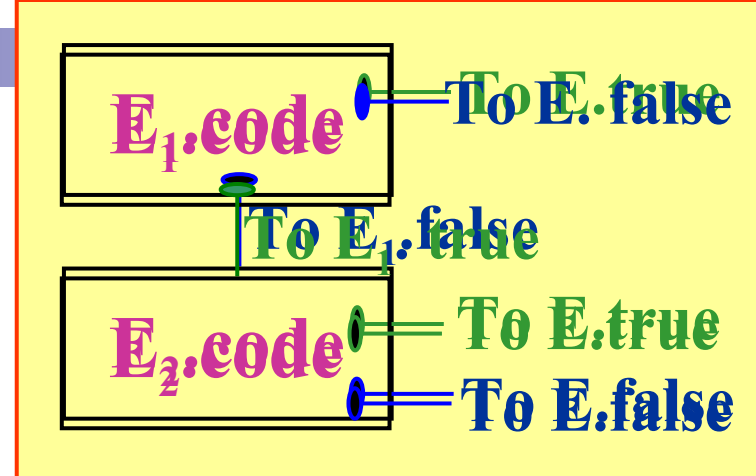
布尔表达式的翻译模式

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

```
{ backpatch( $E_1$ .falselist, M.quad);  
   $E$ .truelist:=merge( $E_1$ .truelist,  $E_2$ .truelist);  
   $E$ .falselist:= $E_2$ .falselist }
```

(2) $E \rightarrow E_1 \text{ and } M E_2$

```
{ backpatch( $E_1$ .truelist, M.quad);  
   $E$ .truelist:= $E_2$ .truelist;  
   $E$ .falselist:=merge( $E_1$ .falselist,  $E_2$ .falselist) }
```



布尔表达式的翻译模式

(3) $E \rightarrow \text{not } E_1$

```
{ E.truelist:=E1.falselist;  
  E.falselist:=E1.truelist}
```

(4) $E \rightarrow (E_1)$

```
{ E.truelist:=E1.truelist;  
  E.falselist:=E1. falselist}
```

布尔表达式的翻译模式

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

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

(6) $E \rightarrow id$

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

布尔表达式的翻译模式

- 作为整个布尔表达式的“真” “假” 出口
(转移目标) 仍待回填 .

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

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

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

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

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

```
{ backpatch(E1.falselist, M.quad);  
  E.truelist:=merge(E1.truelist, E2.truelist);  
  E.falselist:=E2.falselist }
```

(2) $E \rightarrow E_1 \text{ and } M E_2$

```
{ backpatch(E1.truelist, M.quad);  
  E.truelist:=E2.truelist;
```

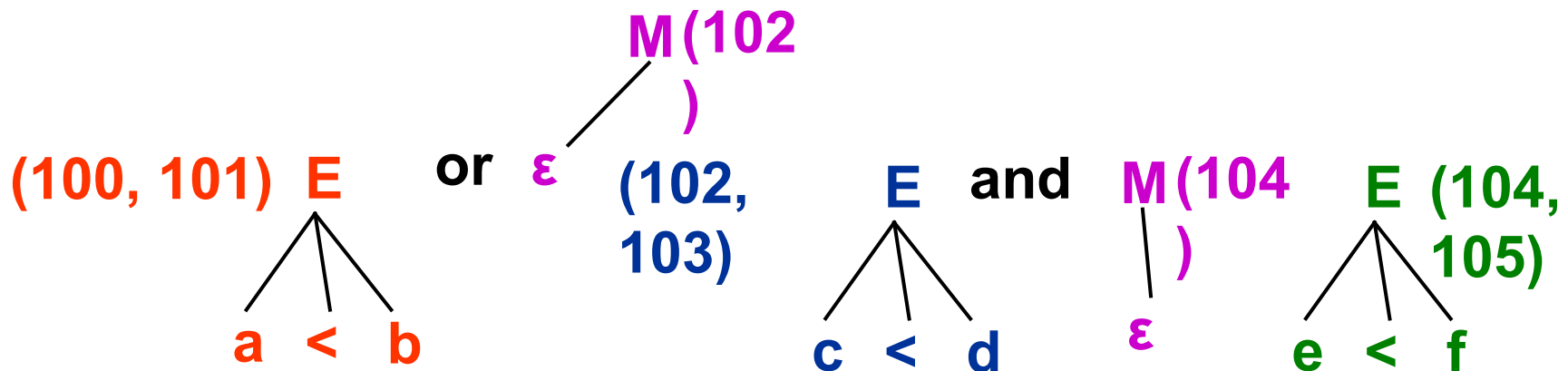
```
E.falselist:=merge(E1.falselist E2.falselist) }
```

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

```

(5)  $E \rightarrow id_1 \text{ relop } id_2$ 
    {  $E.\text{truelist} := \text{makelist}(\text{nextquad});$ 
       $E.\text{falselist} := \text{makelist}(\text{nextquad}+1);$ 
      emit('j' relop.op ', '  $id_1.\text{place}$  ', '  $id_2.\text{place}$  ', ' 0');
      emit('j, - , - , 0') }
(7)  $M \rightarrow \epsilon$  {  $M.\text{quad} := \text{nextquad}$  }
  
```

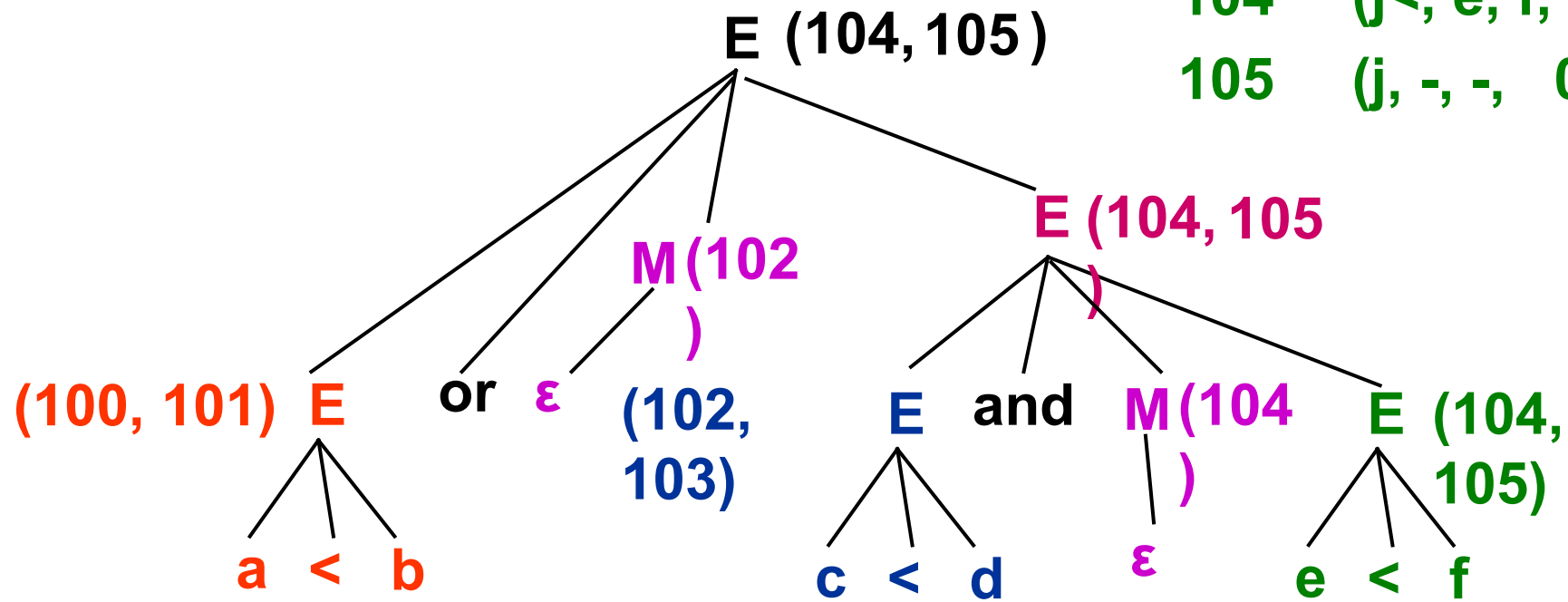
100	(j<, a, b, 0)
101	(j, -, -, 0)
102	(j<, c, d, 0)
103	(j, -, -, 0)
104	(j<, e, f, 0)
105	(j, -, -, 0)



(1) $E \rightarrow E_1 \text{ or } M E_2$
 { backpatch(E_1 .falseList, M.quad);
 E .trueList := merge(E_1 .trueList, E_2 .trueList);
 E .falseList := E_2 .falseList }

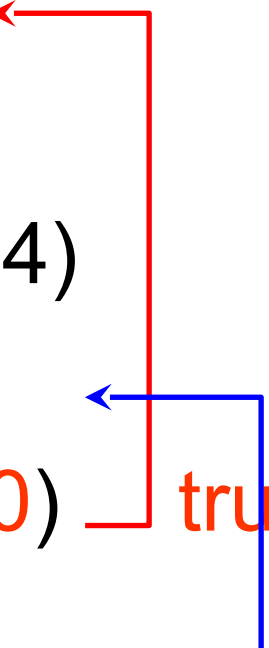
(2) $E \rightarrow E_1 \text{ and } M E_2$
 { backpatch(E_1 .trueList, M.quad);
 E .trueList := E_2 .trueList;
 E .falseList := merge(E_1 .falseList, E_2 .falseList) }

100	(j<, a, b, 0	102
)		
101	(j, -, -, 0	104
102	(j<, c, d,	
0)		100
103	(j, -, -, 103)	
104	(j<, e, f, 0)	
105	(j, -, -, 0)	



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

100 (j<, a, b, 0)
101 (j, -, -, 102)
102 (j<, c, d, 104)
103 (j, -, -, 0)
104 (j<, e, f, 100) truelist
105 (j, -, -, 103) false list



小结

- 布尔表达式的翻译
 - 数值表示法
 - 作为条件控制的布尔式翻译
 - 一遍扫描的翻译模式

作业

- P218-6

1. 请将表达式 $-(a+b)*(c+d)-(a+b+c)$ 分别表示成三元式、间接三元式和四元式序列。

解答：

四元式序列为：

- (1) (+, a, b, T1)
- (2) (@, T1, -, T2)
- (3) (+, c, d, T3)
- (4) (*, T2, T3, T4)
- (5) (+, a, b, T5)
- (6) (+, T5, c, T6)
- (7) (-, T4, T6, T7)

三元式序列为：

- (1) (+, a, b)
- (2) (@, (1), -)
- (3) (+, c, d)
- (4) (*, (2), (3))
- (5) (+, a, b)
- (6) (+, (5), c)
- (7) (-, (4), (6))

间接三元式为：

三元式表为：

- (1) (+, a, b)
- (2) (@, (1), -)
- (3) (+, c, d)
- (4) (*, (2), (3))
- (5) (+, (1), c)
- (6) (-, (4), (5))

间接码表：

- (1)
- (2)
- (3)
- (4)
- (1)
- (5)

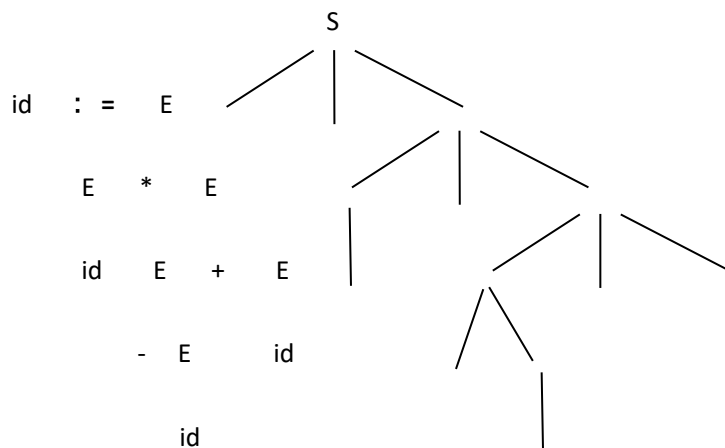
(6)

2. 按 7.3 节所说的办法，写出下面赋值句

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

的自下而上语法制导翻译过程。给出所产生的三地址代码。

解答：语法树为



分析过程：

步骤 地址代码	符号栈	输入符号串	动作	语义值	产生式	三
1	#	A: =B* (-C+D) #	预备			
2	#id	: =B* (-C+D) #	移进	A		
3	#id:=	B* (-C+D) #	移进	A_		
4	#id:=id	* (-C+D) #	移进	A_B		
5	#id:=E	* (-C+D) #	规约	A_B	$E \rightarrow id$	
6	#id:=E*	(-C+D) #	移进	A_B_		
7	#id:=E*(-C+D) #	移进	A_B__		
8	#id:=E*(-	C+D) #	移进	A_B___		
9	#id:=E*(-id	+D) #	移进	A_B___C		
10	#id:=E*(-E	+D) #	规约	A_B___C	$E \rightarrow id$	
11	#id:=E*(E	+D) #	规约	A_B__T1	$E \rightarrow -E$ $T1 := -C$	
12	#id:=E*(E+	D)#	移进	A_B__T1_		
13	#id:=E*(E+id)#	移进	A_B__T1_D		
14	#id:=E*(E+E)#	规约	A_B__T1_	$E \rightarrow id$	
15	#id:=E*(E)#	规约	A_B__T2	$E \rightarrow E+E$ $T2 := T1+D$	
16	#id:=E*(E)	#	移进	A_B__T2_		
17	#id:=E*E	#	规约	A_B_T2		
18	#id:=E	#	规约	A_T3	$E \rightarrow E*E$ $T3 := B*T2$	
19	#S	#	规约	T3	$S \rightarrow id:=E$ $A:=T3$	

3. 按 7.4.2 节的办法，写出布尔式 $A \text{ or } (B \text{ and not } (C \text{ or } D))$ 的四元式序列。

解答：

```

100 (jnz, A, __, 0) E。 T
101 (j, __, __, 102)
    102 (jnz, B, __, 104)
    103 (j, __, __, 0) E。 F
    104 (jnz, C, __, 103) E。 F
    105 (j, __, __, 106)
    106 (jnz, D, __, 104) E。 F
    107 (j, __, __, 100) E。 T

```

真出口为：E.truelist={100, 107} 假出口为：E.falselist={103, 104, 106}



编译原理

第七章 语义分析和中间代码产生

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

7.5 控制语句的翻译

- 考虑下列产生式所定义的语句

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

$\quad | \text{if } E \text{ then } S_1 \text{ else } S_2$

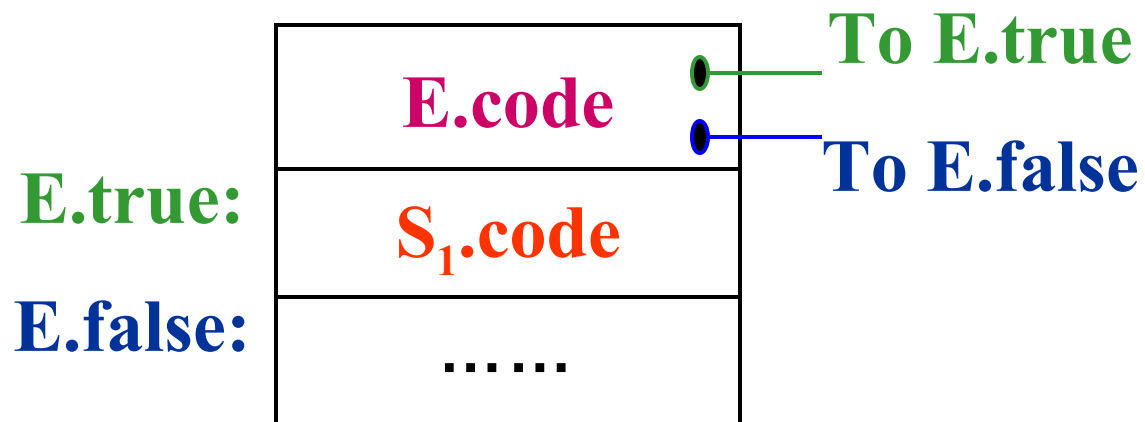
$\quad | \text{while } E \text{ do } S_1$

其中 E 为布尔表达式。

利用属性文法定义语义
设计一遍扫描的翻译模式

■ if-then 语句

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



if-then 语句的属性文法

产生式

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

语义规则

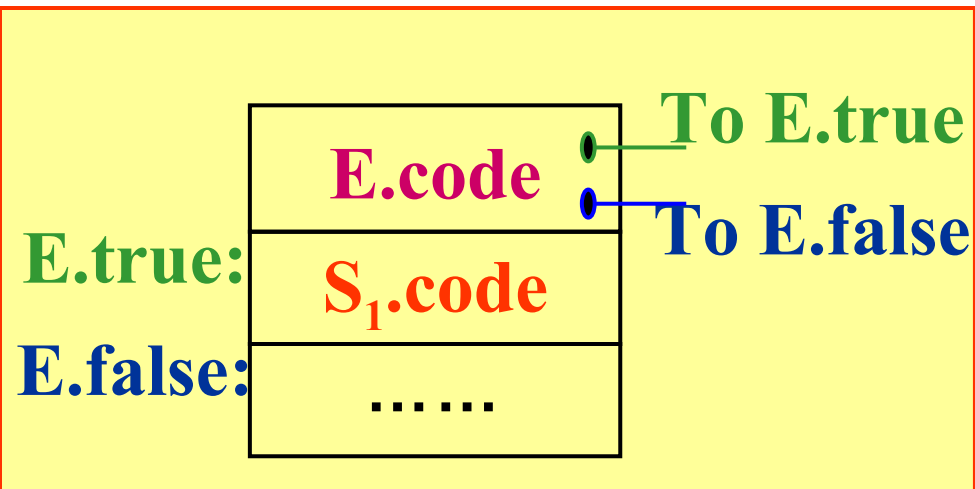
$E.\text{true} := \text{newlabel};$

$E.\text{false} := S.\text{next};$

$S_1.\text{next} := S.\text{next}$

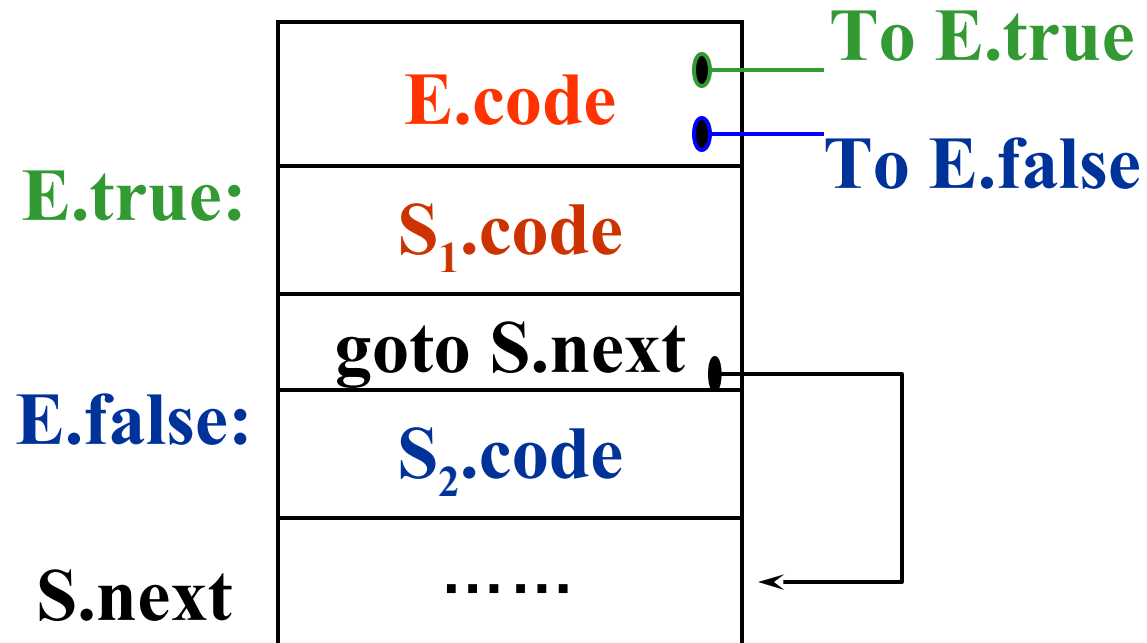
$S.\text{code} := E.\text{code} \parallel$

$\text{gen}(E.\text{true} ':') \parallel S_1.\text{code}$



■ if-then-else 语句

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



if-then-else 语句的属性文法

产生式

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

语义规则

$E.\text{true} := \text{newlabel};$

$E.\text{false} := \text{newlabel};$

$S_1.\text{next} := S.\text{next}$

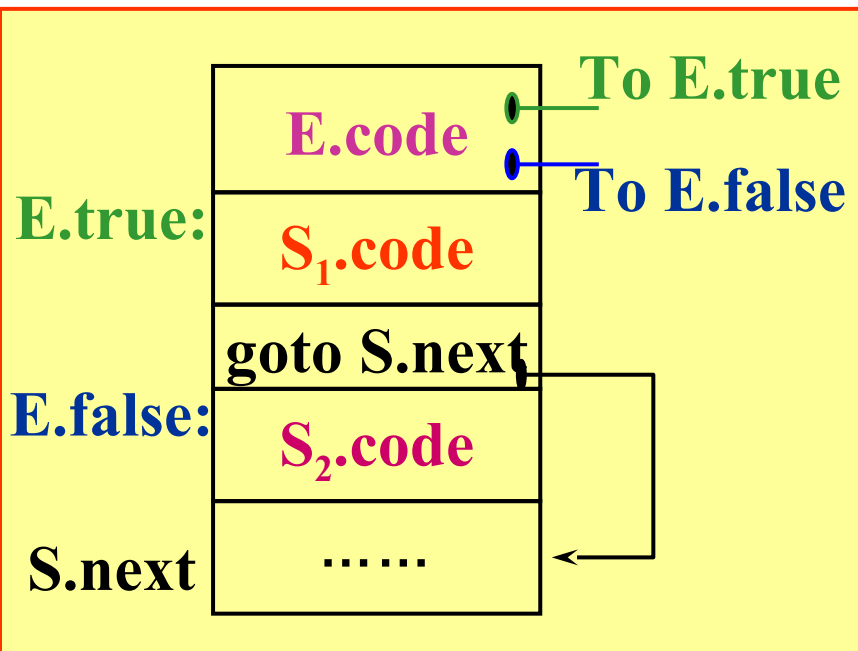
$S_2.\text{next} := S.\text{next};$

$S.\text{code} := E.\text{code} \parallel$

$\text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \parallel$

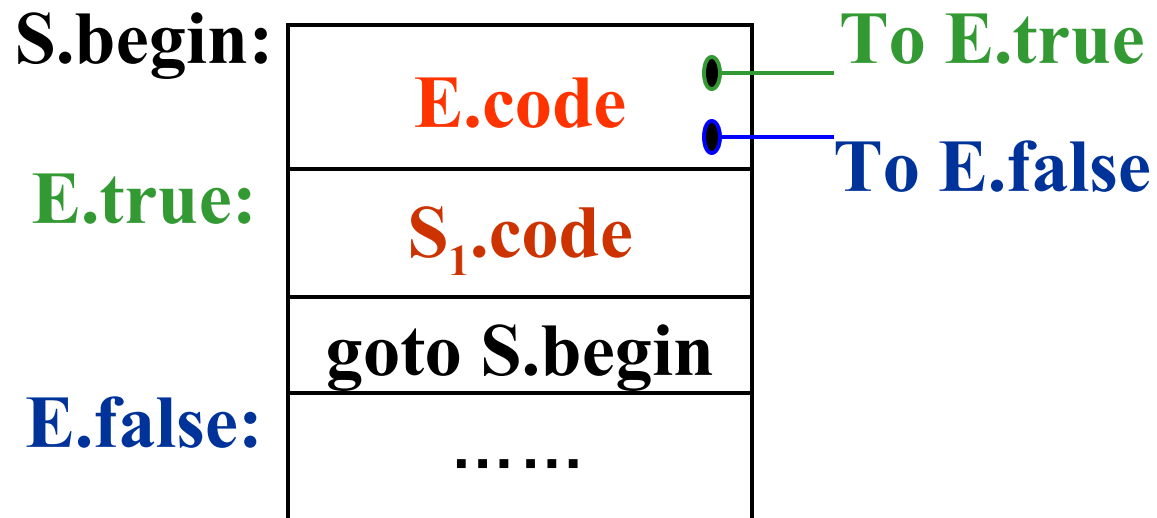
$\text{gen}(\text{'goto' } S.\text{next}) \parallel$

$\text{gen}(E.\text{false} ':') \parallel S_2.\text{code}$



■ while-do 语句

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



while-do 语句的属性文法

产生式

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

语义规则

$S.\text{begin} := \text{newlabel};$

$E.\text{true} := \text{newlabel};$

$E.\text{false} := S.\text{next};$

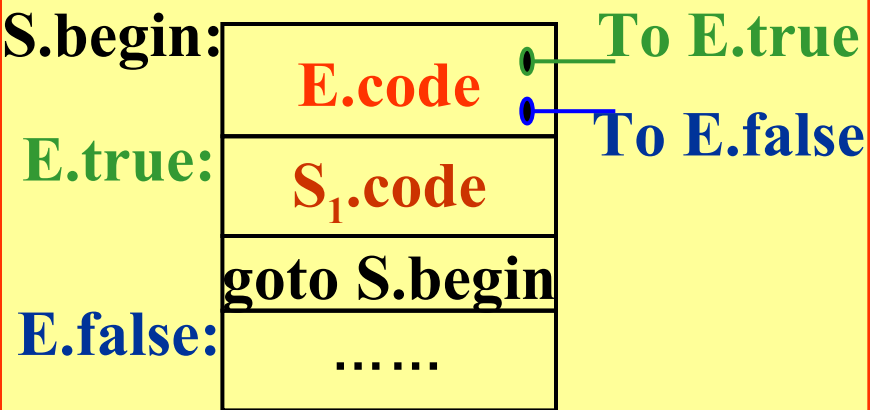
$S_1.\text{next} := S.\text{begin};$

$S.\text{code} := \text{gen}(S.\text{begin} ':') \parallel$

$E.\text{code} \parallel$

$\text{gen}(E.\text{true} ':') \parallel S_1.\text{code} \parallel$

$\text{gen}(\text{'goto' } S.\text{begin})$



产生式

语义规则

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

E.true:=newlabel;

E.false:=S.next;

S_1 .next:=S.next

S.code:=E.code || gen(E.true ':') || S_1 .code

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

E.true:=newlabel;

E.false:=newlabel;

S_1 .next:=S.next

S_2 .next:=S.next;

S.code:=E.code || gen(E.true ':') || S_1 .code ||

gen('goto' S.next) || gen(E.false ':') || S_2 .code

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

S.begin:=newlabel;

E.true:=newlabel;

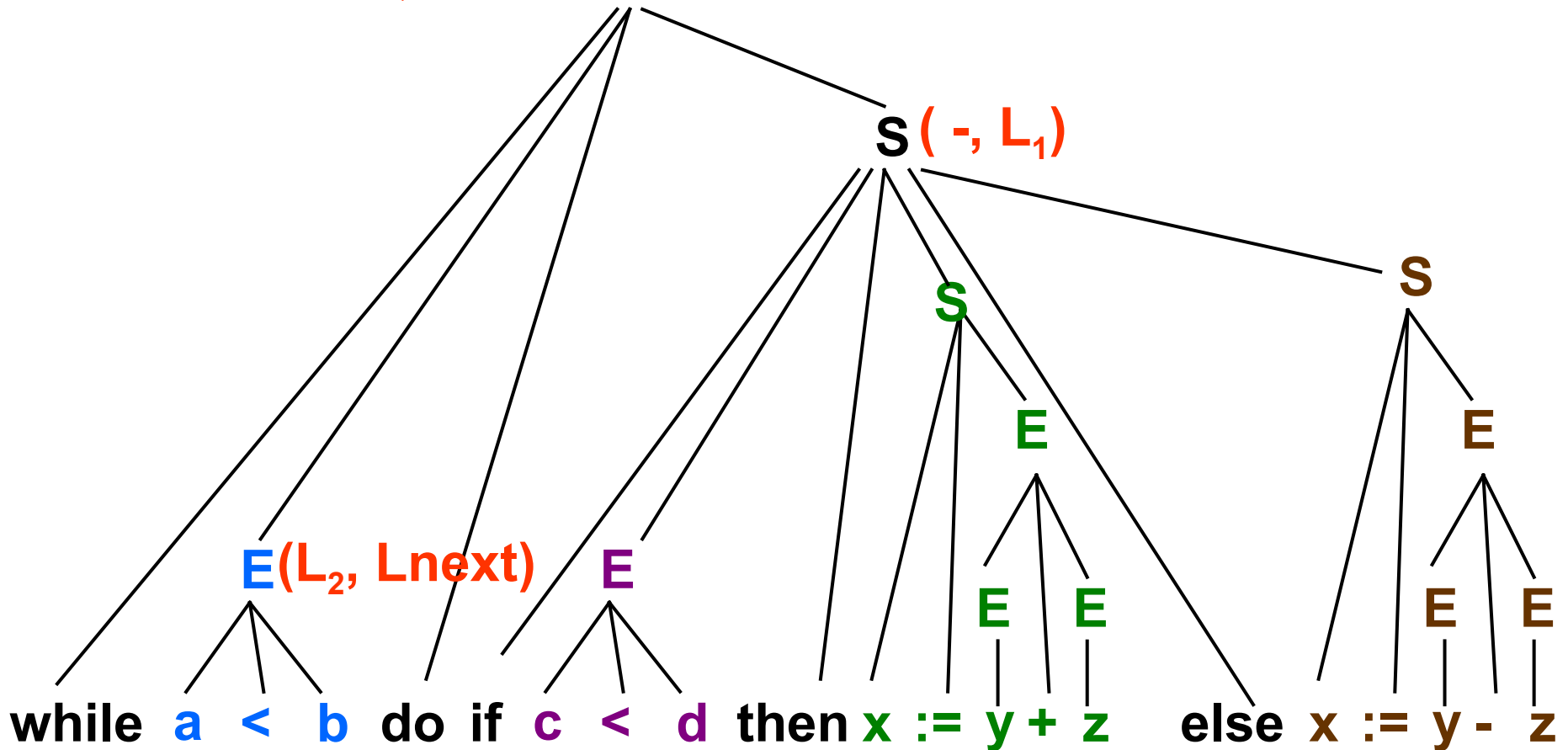
E.false:=S.next;

S_1 .next:=S.begin;

S.code:=gen(S.begin ':') || E.code || gen(E.true ':') ||

$S \rightarrow \text{while } E \text{ do } S_1$
 $S.\text{begin} := \text{newlabel};$
 $E.\text{true} := \text{newlabel};$
 $E.\text{false} := S.\text{next};$
 $S_1.\text{next} := S.\text{begin};$
 $S.\text{code} := \text{gen}(S.\text{begin} ':') \parallel E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel$

(L_1, L_{next})
 $S.\text{code} \parallel \text{gen}(\text{'goto' } S.\text{begin})$



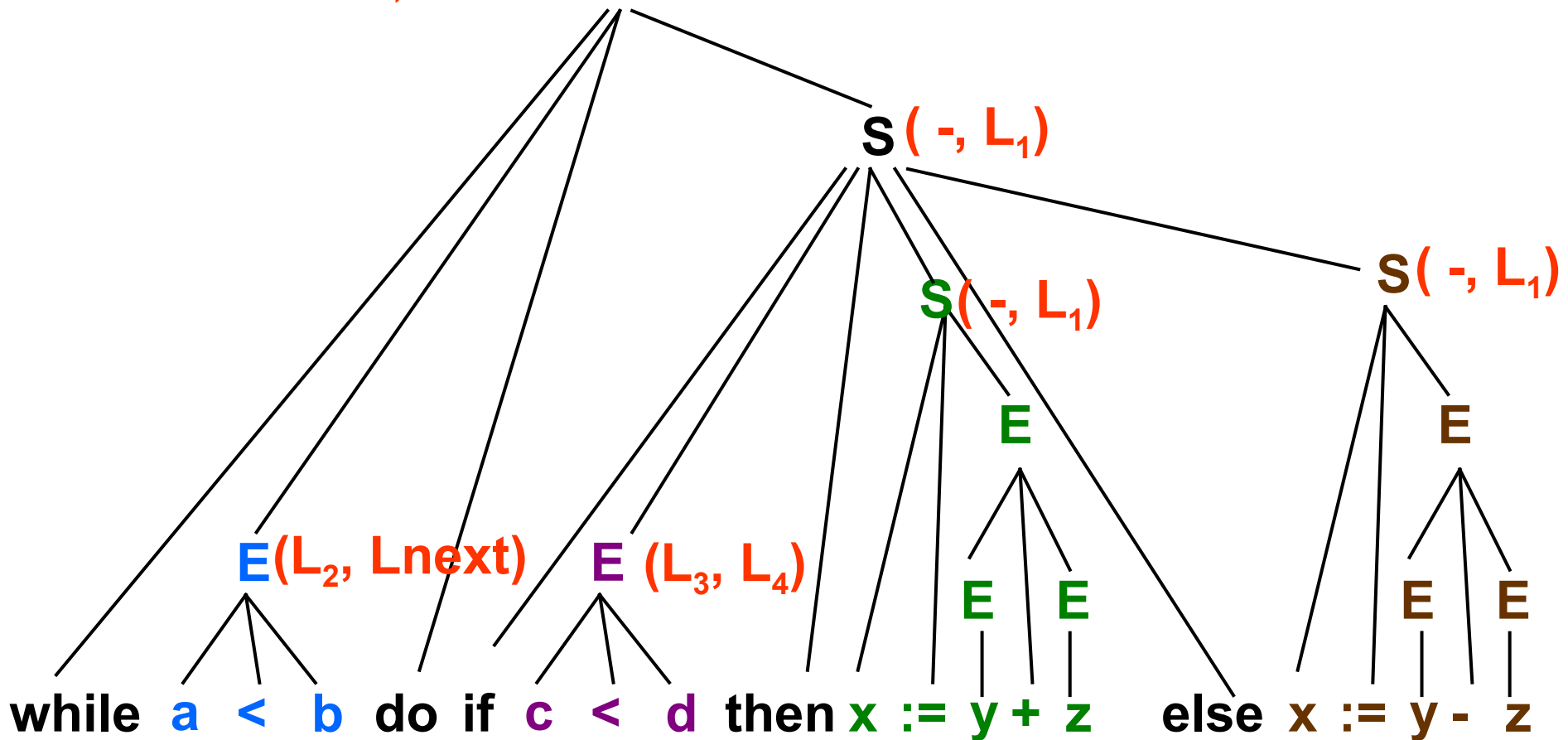
$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$

```

E.true:=newlabel;
E.false:=newlabel;
S1.next:=S.next
S2.next:=S.next;
S.code:=E.code || gen(E.true ':') ||
S1.code || gen('goto' S.next) ||
gen(E.false ':') || S2.code

```

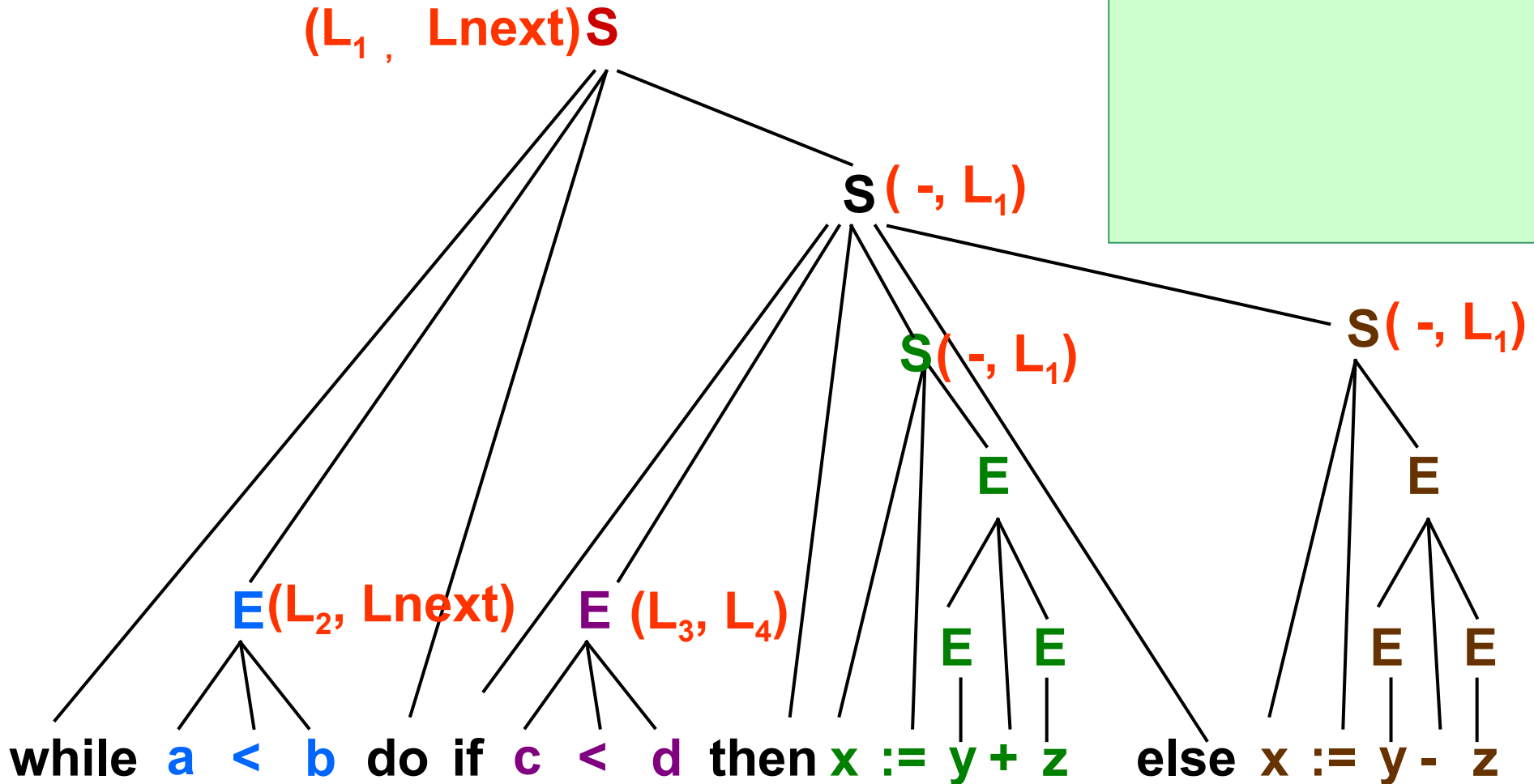
$(L_1, L_{next})S$



语义规则

```
E.code:=gen('if ' id1.place
    relop.op id2.place 'goto' E.true)
    || gen('goto' E.false)
```

```
if a<b goto L2
goto Lnext
if c<d goto L3
goto L4
```



```

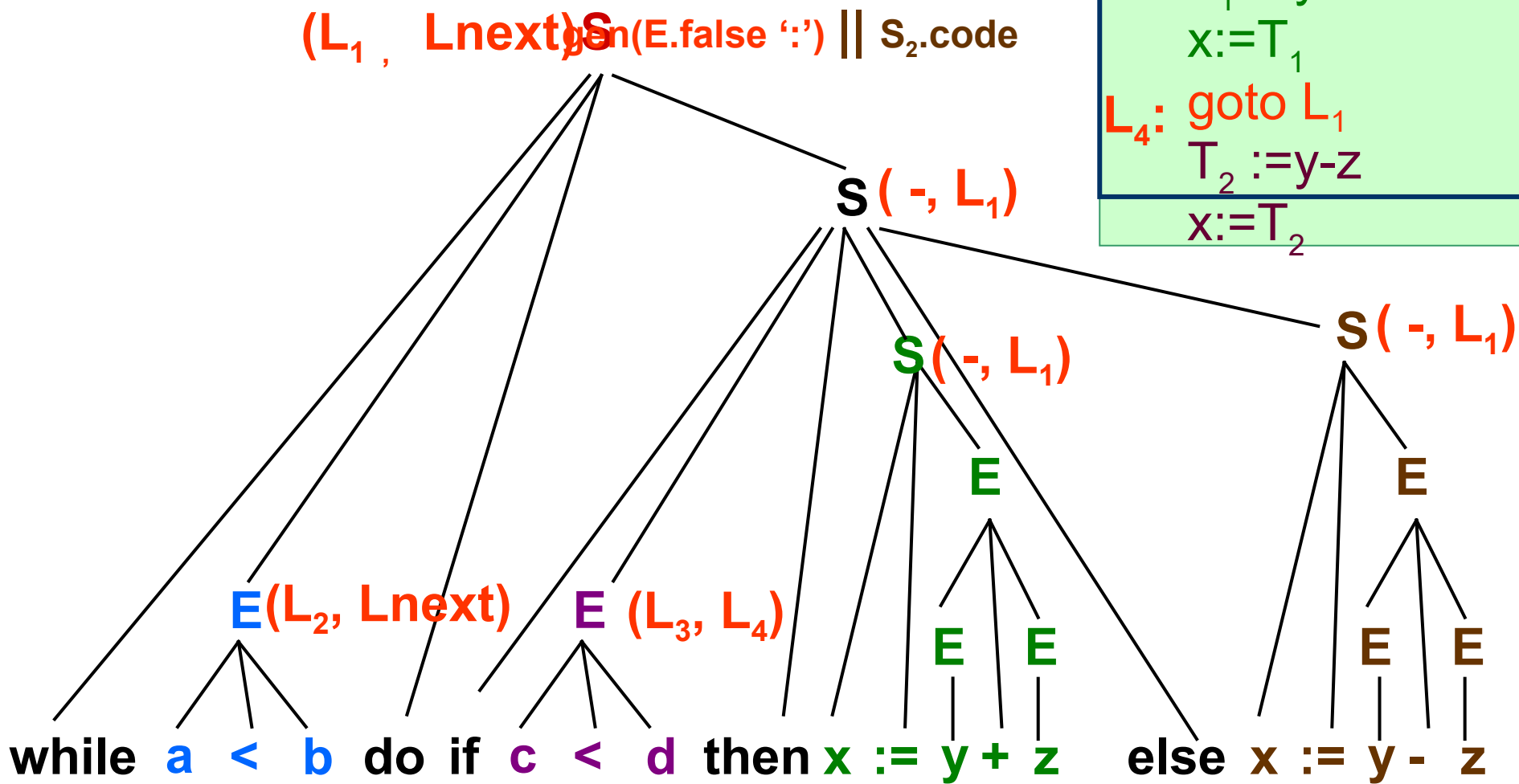
if a<b goto L2
goto Lnext
if c<d goto L3
goto L4
T1 := y+z
x := T1

```

$$\begin{array}{l} T_2 := y - z \\ x := T_2 \end{array}$$


$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$
 $E.\text{true} := \text{newlabel};$
 $E.\text{false} := \text{newlabel};$
 $S_1.\text{next} := S.\text{next}$
 $S_2.\text{next} := S.\text{next};$
 $S.\text{code} := E.\text{code} \parallel \text{gen}(E.\text{true} ':') \parallel$
 $S_1.\text{code} \parallel \text{gen}(\text{'goto' } S.\text{next}) \parallel$

if $a < b$ goto L_2
 goto L_{next}
 if $c < d$ goto L_3
 goto L_4
 $L_3:$ $T_1 := y + z$
 $x := T_1$
 $L_4:$ goto L_1
 $T_2 := y - z$
 $x := T_2$



$x := T_2$

考虑如下语句：

```
while a<b do  
    if c<d then x:=y+z  
    else x:=y-z
```

■ 生成下列代码：

L_1 : if a<b goto L_2

goto Lnext

L_2 : if c<d goto L_3

goto L_4

L_3 : $T_1:=y+z$

$x:=T_1$

goto L_1

L_4 : $T_2:=y-z$

$x:=T_2$

goto L_1

一遍扫描翻译控制流语句

- 考虑下列产生式所定义的语句：

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

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

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

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

(5) | A

(6) $L \rightarrow L; S$

(7) | S

- S 表示语句， L 表示语句表，
 A 为赋值语句， E 为一个布尔表达式

■ if 语句的翻译

相关产生式

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

改写后产生式

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

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

$$M \rightarrow \varepsilon$$

$$N \rightarrow \varepsilon$$

翻译模式：

1. $S \rightarrow \text{if } E \text{ then } M \ S_1$

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

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

{ backpatch(E.truelist, $M_1.\text{quad}$);
 backpatch(E.falselist, $M_2.\text{quad}$);
 $S.\text{nextlist} := \text{merge}(S_1.\text{nextlist}, N.\text{nextlist}, S_2.\text{nextlist})$ }

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

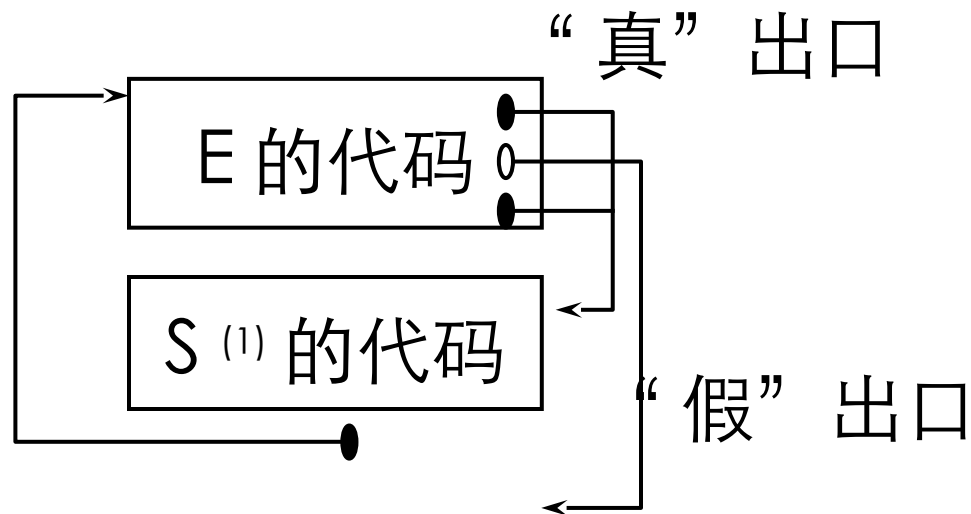
4. $N \rightarrow \varepsilon$ { $N.\text{nextlist} := \text{makelist}(\text{nextquad});$
 emit('j, - , - , - ') }

■ while 语句的翻译

相关产生式

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

■ 翻译为：



为了便于 "回填", 改写产生式为：

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

$M \rightarrow \varepsilon$

■ 翻译模式：

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

{

 backpatch(E.truelist, M_2 .quad);

 backpatch(S_1 .nextlist, M_1 .quad);

S .nextlist := E.falselist;

 emit('j, - , - ,' M_1 .quad) }

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

■ 产生式

$$L \rightarrow L; S$$

改写为：

$$L \rightarrow L_1; M S$$

$$M \rightarrow \varepsilon$$

■ 翻译模式：

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

■ 其它几个语句的翻译

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

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

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

示例：翻译语句

- 将下面的语句翻译为四元式
 while (a<b) do
 if (c<d) then x:=y+z;

P195

```
S → if E then M S1
{ backpatch(E.truelist, M.quad)
  S.nextlist := merge(E.falselist,
    S1.nextlist) }
M → ε { M.quad := nextquad }
S → A { S.nextlist := makelist( ) }
```

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

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

P195

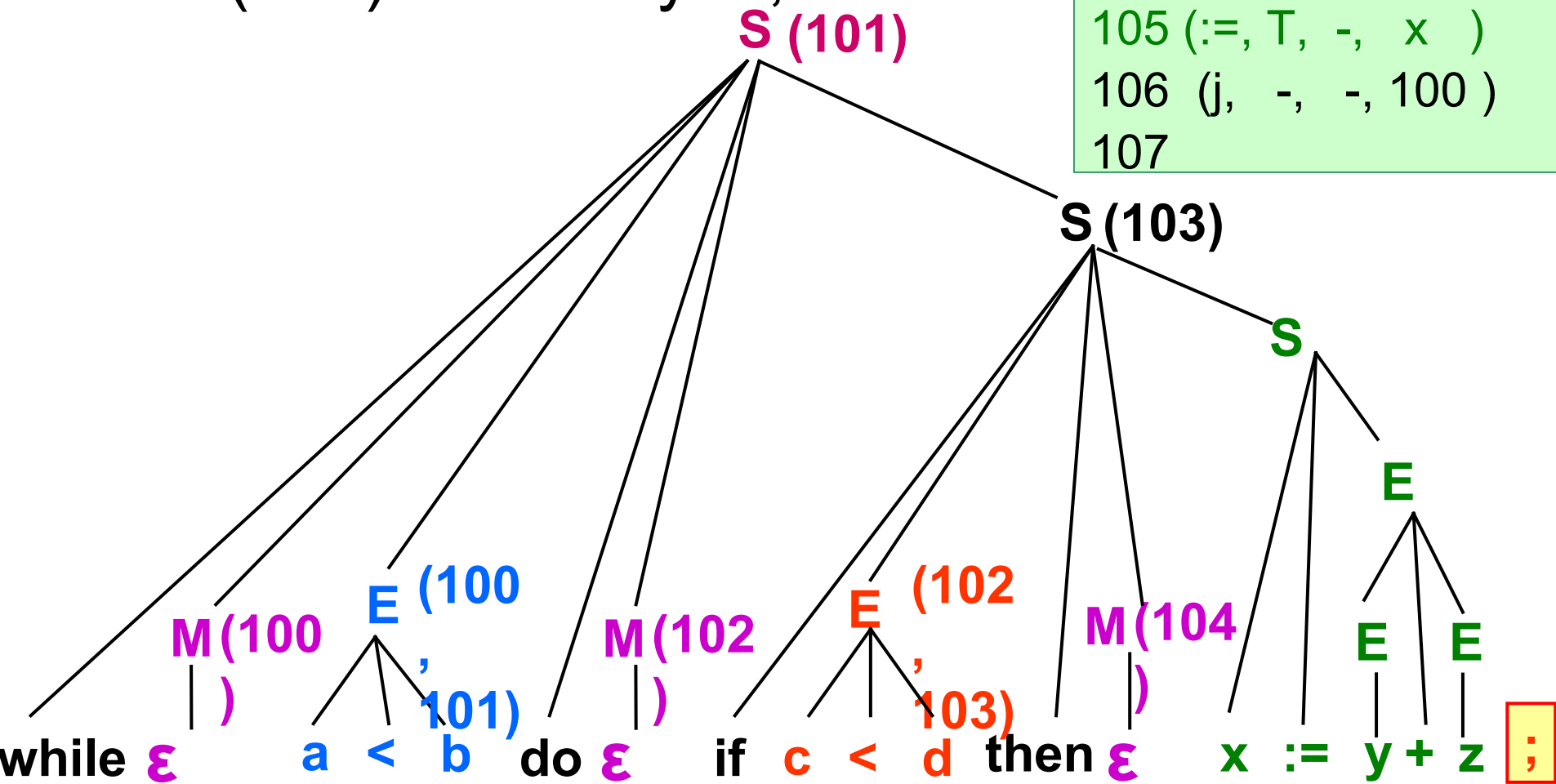
```
S → while M1 E do M2 S1
{ backpatch(E.truelist, M2.quad);
  backpatch(S1.nextlist, M1.quad);
  S.nextlist := E.falselist
  emit('j, - , - , ' M1.quad) }
M → ε { M.quad := nextquad }
```

P179

```
S → id := E      { p := lookup(id.name);
                  if p ≠ nil then emit(p ':=' E.place)
                  else error }
E → E1 + E2    { E.place := newtemp;
                  emit(E.place ':=' E1.place '+' E2.place) }
```

翻译语句

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



翻译语句

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

100 (j<, a, b, 102)

101 (j, -, -, 107)

102 (j<, c, d, 104)

103 (j, -, -, 100)

104 (+, y, z, T)

105 (:=, T, -, x)

106 (j, -, -, 100)

107

小结

- 控制语句的翻译

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

| $\text{if } E \text{ then } S_1 \text{ else } S_2$

| $\text{while } E \text{ do } S_1$

- 一遍扫描的翻译模式

作业

- P218 - 7 , 12



编译原理

第七章 语义分析和中间代码产生

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

控制语句的翻译

■ 控制语句

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

- | $\text{if } E \text{ then } S_1 \text{ else } S_2$

- | $\text{while } E \text{ do } S_1$

- 标号与 goto 语句

- CASE 语句

7.5.2 标号与 goto 语句

- 标号定义形式

L: S ;

- 标号引用

goto L ;

- 示例:

- 向后转移:

```
L1:  .....  
      .....  
      goto L1;
```

- 向前转移:

```
      goto L1;  
      .....  
L1:  .....
```

符号表信息

名字	类型	...	定义否	地址
...

- 向后转移

L1:
.....

goto L1;

符号表信息

名字	类型	...	定义否	地址
...
L1	标号		已	p

■ 向后转移

L1:
.....

goto L1;

(p) (..., ..., ..., ...)

符号表信息

名字	类型	...	定义否	地址
...
L1	标号		已	p

■ 向后转移

L1:
.....

goto L1;

(p) (..., ..., ..., ...)

(n) (j, -, -, p)

符号表信息

■ 向前转移
 goto L1;

L1:

名字	类型	...	定义否	地址
...

符号表信息

■ 向前转移
goto L1;
.....
L1:

名字	类型	...	定义否	地址
...
L1	标号		未	p

→ (p) (j, -, -, 0)

符号表信息

名字	类型	...	定义否	地址
...
L1	标号		未	q

向前转移

goto L1;

.....

goto L1;

.....

L1:

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

...

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

符号表信息

名字	类型	...	定义否	地址
...
L1	标号		未	r

向前转移

goto L1;

.....

goto L1;

.....

goto L1;

.....

L1:

(p) (j, -, -, 0) ←

...

(q) (j, -, -, p) ←

...

(r) (j, -, -, q) ←

符号表信息

名字	类型	...	定义否	地址
...
L1	标号		已	k

向前转移

goto L1;

.....

goto L1;

.....

goto L1;

.....

L1:

(p) (j, -, -, k) ←

...

(q) (j, -, -, k) ←

...

(r) (j, -, -, k) ←

...

(k) ...

产生式 $S' \rightarrow \text{goto } L$ 的语义动作：

```
{  查找符号表;  
    IF L 在符号表中且 " 定义否 " 栏为 " 已 "  
        THEN GEN(J, -, -, P)  
    ELSE IF L 不在符号表中  
        THEN BEGIN  
            把 L 填入表中;  
            置 " 定义否 " 为 " 未 ", " 地址 " 栏为 nextquad ;  
            GEN(J, -, -, 0)  
        END  
    ELSE BEGIN  
        Q:=L 的地址栏中的编号;  
        置地址栏编号为 nextquad ;  
        GEN(J, -, -, Q)  
    END  
}
```

■ 带标号语句的产生式：

$S \rightarrow \text{label } S$

$\text{label} \rightarrow i:$

■ $\text{label} \rightarrow i:$ 对应的语义动作：

1. 若 i 所指的标识符 (假定为 L) 不在符号表中, 则把它填入, 置 " 类型 " 为 " 标号 ", 定义否为 " 已 ", " 地址 " 为 `nextquad` ;
2. 若 L 已在符号表中但 " 类型 " 不为标号或 " 定义否 " 为 " 已 ", 则报告出错;
3. 若 L 已在符号表中, 则把标号 " 未 " 改为 " 已 ", 然后, 把地址栏中的链头 (记为 q) 取出, 同时把 `nextquad` 填在其中, 最后, 执行 `BACKPATCH(q, nextquad)` 。

7.5.3 CASE 语句的翻译

■ 语句结构

case E of

$C_1: S_1;$

$C_2: S_2;$

...

$C_{n-1}: S_{n-1};$

otherwise: S_n

end

■ 翻译法 (一):

$T := E$

L_1 : if $T \neq C_1$ goto L_2
 S_1 的代码
 goto next

L_2 : if $T \neq C_2$ goto L_3
 S_2 的代码
 goto next

L_3 :

...

L_{n-1} : if $T \neq C_{n-1}$ goto L_n
 S_{n-1} 的代码
 goto next

L_n : S_n 的代码

next:

case E of

$C_1: S_1;$

$C_2: S_2;$

...

$C_{n-1}: S_{n-1};$

otherwise: S_n

end

◆ 改进

C_1	S_1 的地址
C_2	S_2 的地址
\vdots	\vdots
E	S_n 的地址

■ 翻译法 (二):
 计算 E 并放入 T 中
 goto test
 L₁: 关于 S₁ 的中间码
 goto next
 ...
 L_{n-1}: 关于 S_{n-1} 的中间码
 goto next
 L_n: 关于 S_n 的中间码
 goto next
 test: if T=C₁ goto L₁
 if T=C₂ goto L₂
 ...
 if T=C_{n-1} goto L_{n-1}
 goto L_n
 next:

L ₁	S ₁ □ □ □ □ □ □ □ □
L ₂	S ₂ □ □ □ □ □ □ □ □
⋮	⋮
L _{n-1}	S _{n-1} □ □ □ □ □ □ □ □
L _n	S _n □ □ □ □ □ □ □ □

(case, C₁, P₁)
 (case, C₂, P₂)
 ...
 (case, C_{n-1}, P_{n-1})
 (case, T, P_n)
 (label, NEXT, -, -)

case E of
 C₁: S₁;
 C₂: S₂;
 ...
 C_{n-1}: S_{n-1};
 otherwise: S_n
 end

C ₁	P ₁
C ₂	P ₂
⋮	⋮
C _{n-1}	P _{n-1}

P_i 是 L_i 在
 符号表中的
 位置

第七章 语义分析和中间代码产生

- 中间语言
- 赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理

7.6 过程调用的处理

- 过程调用主要完成两项工作
 - 传递参数
 - 转子
- 传地址
 - 把实在参数的地址传递给相应的形式参数
 - 调用段预先把实在参数的地址传递到被调用段可以拿到的地方
 - 程序控制转入被调用段之后，被调用段首先把实在参数的地址抄进自己相应的形式单元中
 - 过程体对形式参数的引用域赋值被处理成对形式单元的间接访问

过程调用的翻译

- 翻译方法：把实参的地址逐一放在转子指令的前面。

例如， `CALL S(A, X+Y)` 翻译为：

计算 `X+Y` 置于 `T` 中的代码

`par A` `/* 第一个参数的地址 */`

`par T` `/* 第二个参数的地址 */`

`call S` `/* 转子 */`

过程调用的翻译

■ 过程调用文法：

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

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

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

■ 参数的地址存放在一个队列中

■ 最后对队列中的每一项生成一条 par 语句

CALL S(A, X+Y) 翻译为
计算 X+Y 置于 T 中的代码
par A /* 第一个参数的地址 */
par T /* 第二个参数的地址 */
call S /* 转子 */

过程调用的翻译

CALL S(A , X+Y) 翻译为
计算 X+Y 置于 T 中的代码

```
par A    /* 第一个参数的地址 */  
par T    /* 第二个参数的地址 */  
call S   /* 转子 */
```

■ 翻译模式

3. Elist \rightarrow E

{ 初始化 queue 仅包含 E.place }

2. Elist \rightarrow Elist, E

{ 将 E.place 加入到 queue 的队尾 }

1. S \rightarrow call id (Elist)

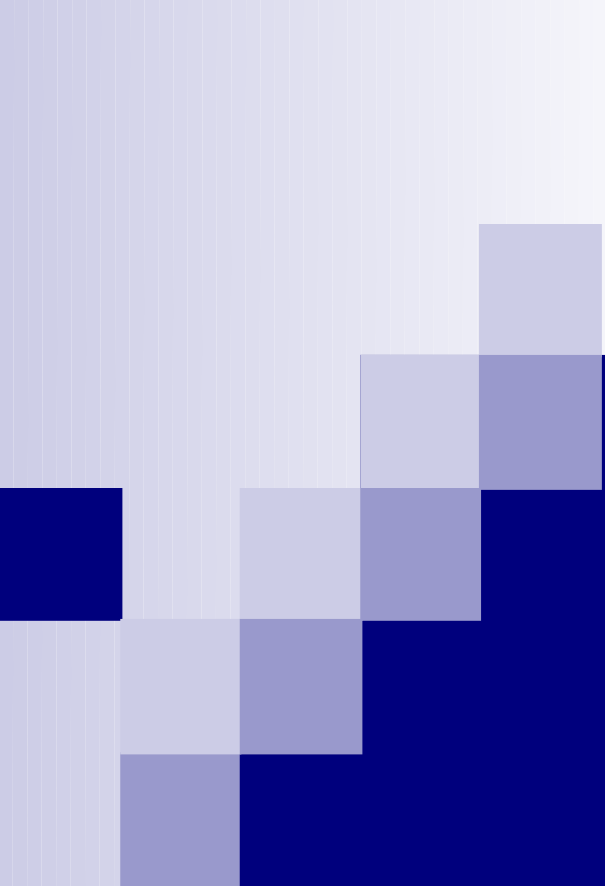
```
{ for 队列 queue 中的每一项 p do  
    emit('param' p);  
    emit('call' id.place) }
```

小结

- 标号与 goto 语句
- CASE 语句的翻译
- 过程调用的处理

本章小结

- 中间语言
- 表达式和赋值语句的翻译
- 布尔表达式的翻译
- 控制语句的翻译
- 过程调用的处理



编译原理

习题课 (1)

第二章 高级语言及其语法描述

- 程序语言的定义
- 高级语言的一般特性
- 程序语言的语法描述

上下文无关文法

- 一个上下文无关文法 G 是一个四元式 $G=(V_T, V_N, S, P)$ ，其中
 - V_T ：终结符集合（非空）
 - V_N ：非终结符集合（非空），且 $V_T \cap V_N = \emptyset$
 - S ：文法的开始符号， $S \in V_N$
 - P ：产生式集合（有限），每个产生式形式为
 - $P \rightarrow \alpha$ ， $P \in V_N$ ， $\alpha \in (V_T \cup V_N)^*$
 - 开始符 S 至少必须在某个产生式的左部出现一次

上下文无关文法

- 定义：称 $\alpha A\beta$ **直接推出** $\alpha\gamma\beta$ ，即

$$\alpha A\beta \Rightarrow \alpha\gamma\beta$$

仅当 $A \rightarrow \gamma$ 是一个产生式，

且 $\alpha, \beta \in (V_T \cup V_N)^*$ 。

- 如果 $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$ ，则我们称这个序列是从 α_1 到 α_n 的一个**推导**。若存在一个从 α_1 到 α_n 的推导，则称 α_1 可以**推导出** α_n

上下文无关文法

- 定义：假定 G 是一个文法， S 是它的开始符号。如果 $S \xRightarrow{*} \alpha$ 则 α 称是一个句型。
- 仅含终结符号的句型是一个句子。
- 文法 G 所产生的句子的全体是一个语言，将它记为 $L(G)$ 。

$$L(G) = \{\alpha \mid S \xRightarrow{+} \alpha, \alpha \in V_T^*\}$$

如何写出产生特定语言的文法

- 分析语言的特点
- 分解成层次结构
- 根据结构写出文法

P36-7 写一个文法，使其语言是奇数集，且每个奇数不以 0 开头。

非 0 开头数字串	奇数数字
-----------	------

■ $G(S)$:

$S \rightarrow O \mid A O$

$O \rightarrow 1 \mid 3 \mid 5 \mid 7 \mid 9$

$N \rightarrow 0 \mid 2 \mid 4 \mid 6 \mid 8$

$D \rightarrow 0 \mid N$

$A \rightarrow A D \mid N$

P36-11. 给出下面语言的相应文法

■ $L_1 = \{a^n b^n c^i \mid n \geq 1 \ \square \ i \geq 0\}$

■ $\square \square \square \ G(S):$

$$S \rightarrow A C$$

$$A \rightarrow a A b \mid ab$$

$$C \rightarrow c C \mid \varepsilon$$

■ $L_4 = \{1^n 0^m 1^m 0^n \mid n \square m \geq 0\}$

■ $\square \square \square \ G(S):$

$$S \rightarrow 1 S 0 \mid B \mid \varepsilon$$

$$B \rightarrow 0 B 1 \mid \varepsilon$$

语法树与二义性 (ambiguity)

- 定义：如果一个文法存在某个句子对应两颗不同的语法树，则说这个**文法是二义的**
- 语言的二义性：一个**语言是二义性的**，如果对它不存在无二义性的文法
- 二义性问题是不可判定问题，即不存在一个算法，它能在有限步骤内，确切地判定一个文法是否是二义的
- 可以找到一组无二义文法的充分条件

P36-9. 证明下面的文法是二义的:

$$S \rightarrow iSeS \mid iS \mid i$$

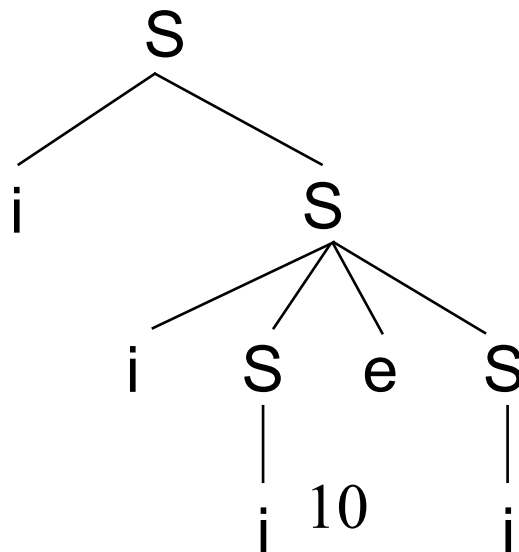
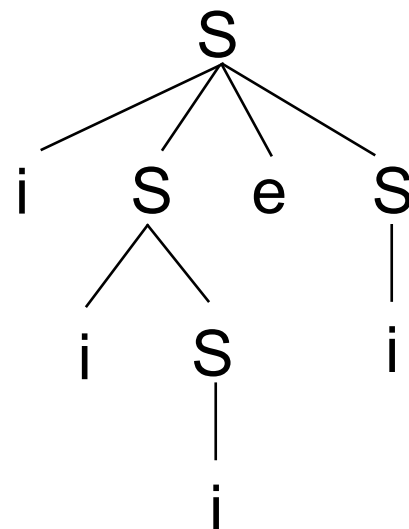
- 思路: 找出一个句子有两棵语法树 (两个最左推导、两个最右推导)

- 前提: 分析文法的特点, 找出歧义产生的源头

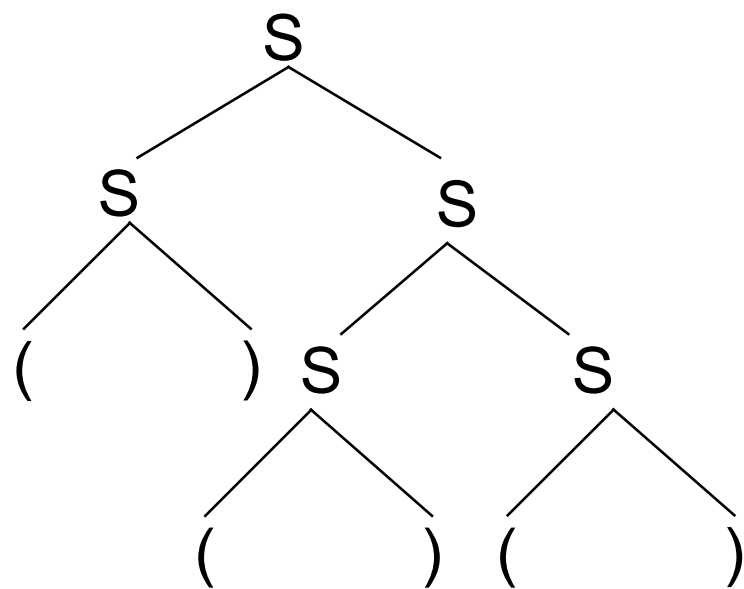
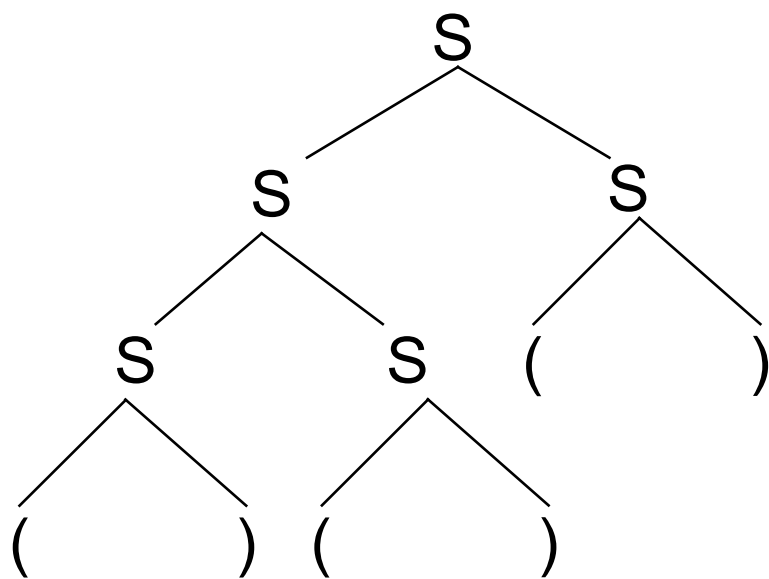
□ 考虑句型 **iiSeS**

- 解答:

句子 **iiiiei** 有两个语法树



P36-10. 把下面文法改写为无二义的：
 $S \rightarrow SS \mid (S) \mid ()$



■ 考虑句型 SSS

■ $G(S)$:

$S \rightarrow ST \mid T$

$T \rightarrow (S) \mid ()$

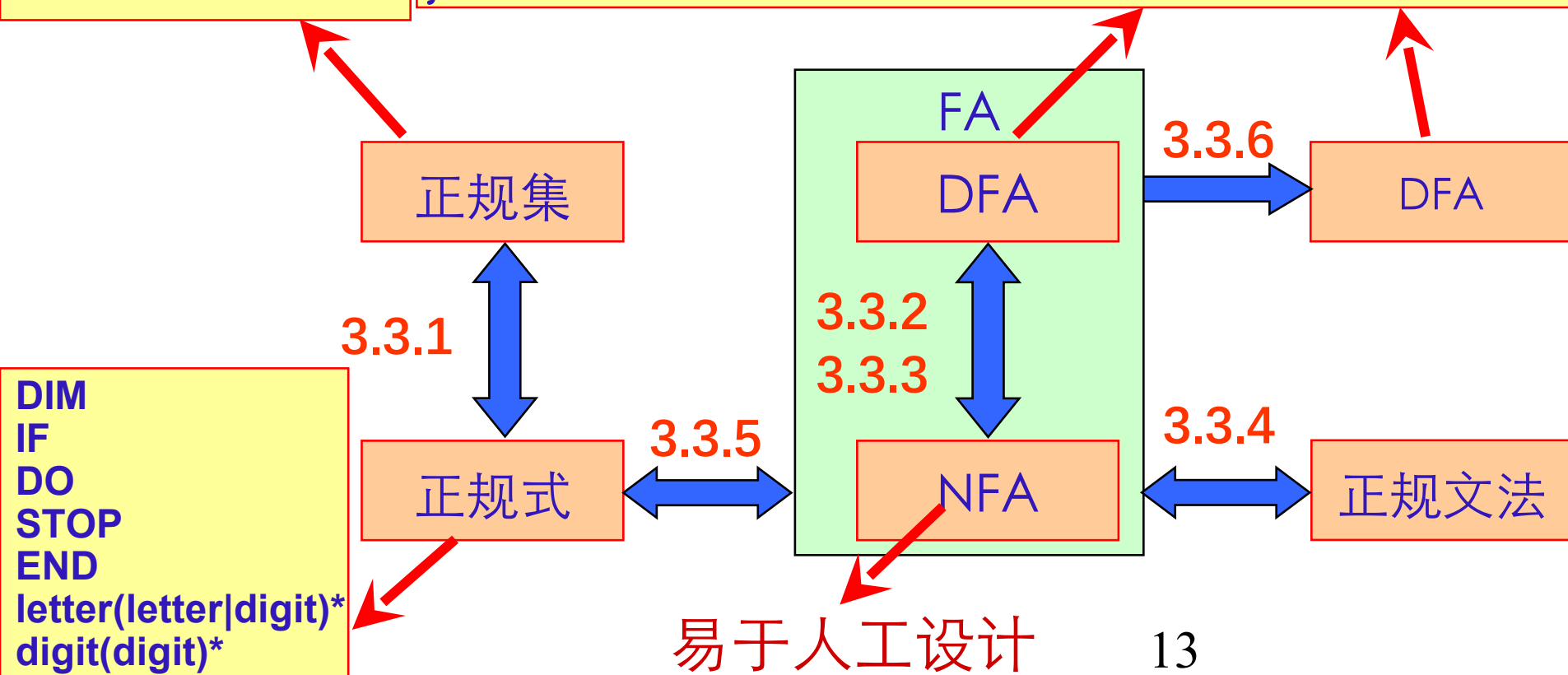
第三章 词法分析

- 对于词法分析器的要求
- 词法分析器的设计
- 正规表达式与有限自动机
- 词法分析器的自动产生 --LEX

关系图

DIM,IF, DO,STOP,END
number, name, age
125, 2169
...

```
curState = 初态
GetChar();
while( stateTrans[curState][ch] 有定义 ){
    // 存在后继状态, 读入、拼接
    Concat();
    // 转换入下一状态, 读入下一字符
    curState= stateTrans[curState][ch];
    if cur_state 是终态 then 返回 strToken 中的单
    GetChar( );
}
```



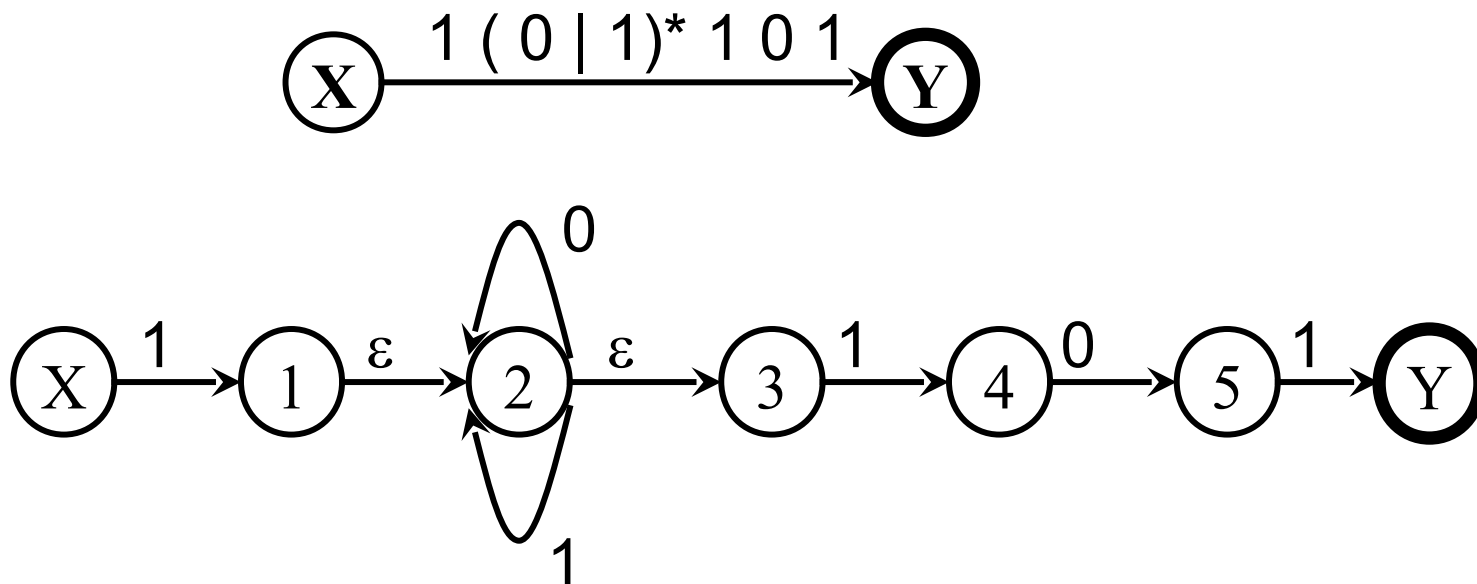
要点

- 几个转换算法
 - 正规式 \Leftrightarrow NFA
 - NFA \Rightarrow DFA
 - DFA 化简算法

P64-7. 构造下列正规式相应的 DFA

$1(0 \mid 1)^*101$

■ 思路：正规式 \Rightarrow NFA \Rightarrow DFA



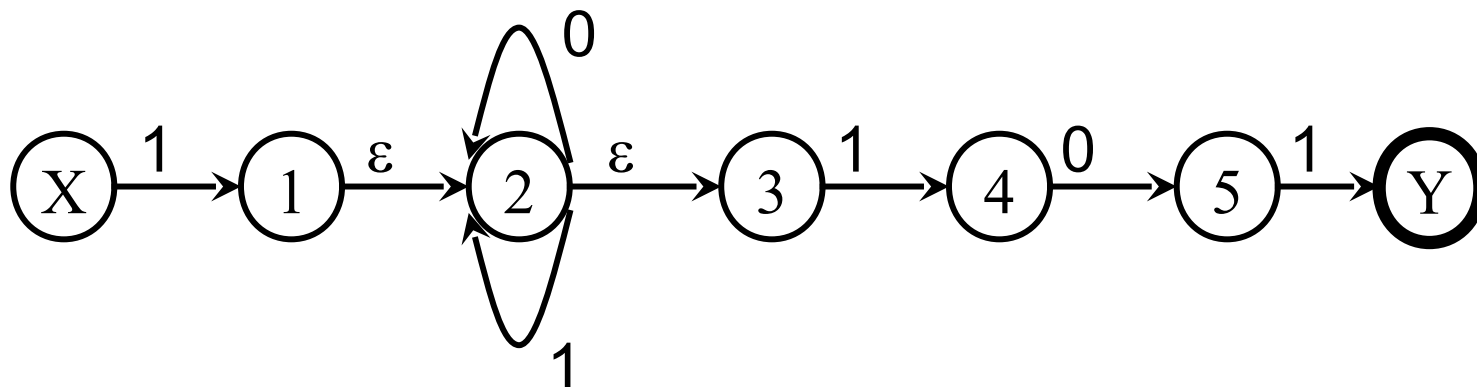
确定化的过程

- 不失一般性，设字母表只包含两个 a 和 b ，我们构造一张表：

I	I_a	I_b
$\epsilon\text{-Closure}(\{X\})$	$\{\dots\}$	$\{\dots\}$
$\{\dots\}$	$\{\dots\}$	$\{\dots\}$
$\{\dots\}$	$\{\dots\}$	$\{\dots\}$

- 首先，置第 1 行第 1 列为 $\epsilon\text{-closure}(\{X\})$ 求出这一列的 I_a ， I_b ；
- 然后，检查这两个 I_a ， I_b ，看它们是否已在表中的第一列中出现，把未曾出现的填入后面的空行的第 1 列上，求出每行第 2，3 列上的集合 ...
- 重复上述过程，直到所有第 2，3 列子集全部出现在第一列为止

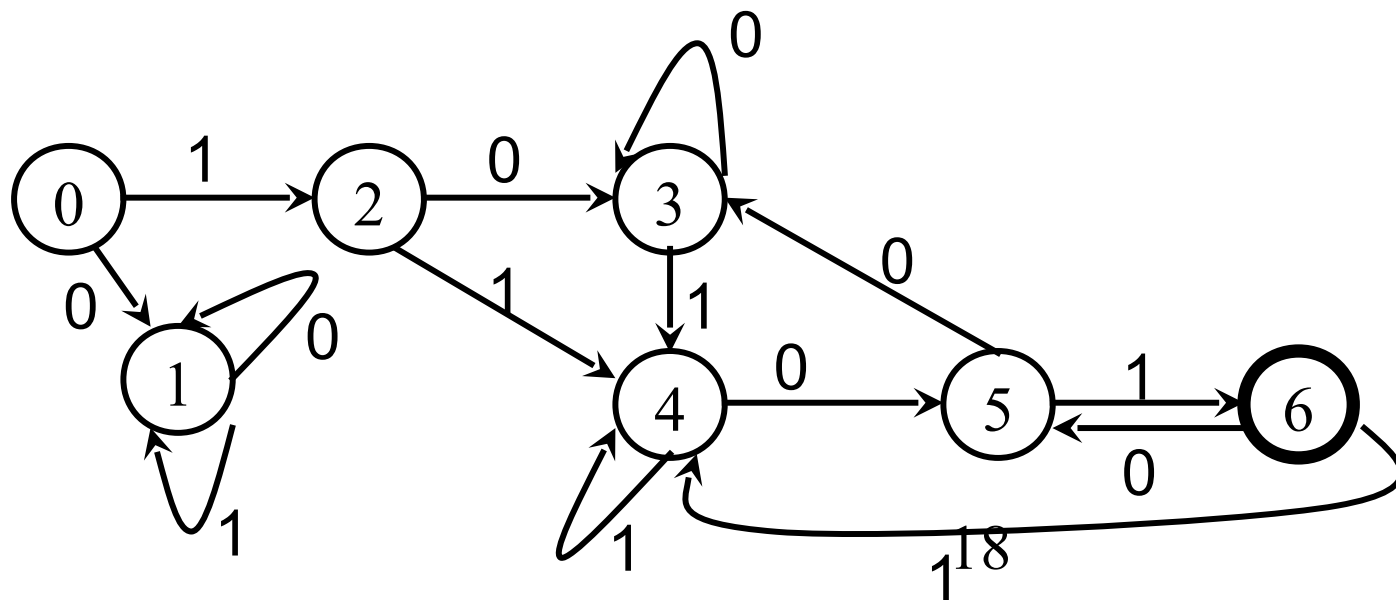
确定化



	0	1
{X}	ϕ	{1,2,3}
ϕ	ϕ	ϕ
{1,2,3}	{2,3}	{2,3,4}
{2,3}	{2,3}	{2,3,4}
{2,3,4}	{2,3,5}	{2,3,4}
{2,3,5}	{2,3}	{2,3,4,Y}
{2,3,4,Y}	{2,3,5}	{2,3,4,}

确定化

	0	1
$\{X\}$	ϕ	$\{1,2,3\}$
ϕ	ϕ	ϕ
$\{1,2,3\}$	$\{2,3\}$	$\{2,3,4\}$
$\{2,3\}$	$\{2,3\}$	$\{2,3,4\}$
$\{2,3,4\}$	$\{2,3,5\}$	$\{2,3,4\}$
$\{2,3,5\}$	$\{2,3\}$	$\{2,3,4,Y\}$
$\{2,3,4,Y\}$	$\{2,3,5\}$	$\{2,3,4,\}$



最小化：对状态集进行划分

- 首先，把 S 划分为终态和非终态两个子集，形成基本划分 Π 。
- 假定到某个时候， Π 已含 m 个子集，记为 $\Pi = \{I^{(1)}, I^{(2)}, \dots, I^{(m)}\}$ ，检查 Π 中的每个子集看是否能进一步划分：
 - 对某个 $I^{(i)}$ ，令 $I^{(i)} = \{s_1, s_2, \dots, s_k\}$ ，若存在一个输入字符 a 使得 $I_a^{(i)}$ 不会包含在现行 Π 的某个子集 $I^{(j)}$ 中，则至少应把 $I^{(i)}$ 分为两个部分。

最小化

$\{0,1,2,3,4,5\}, \{6\}$

$\{0,1,2,3,4,5\}_0 = \{1,3,5\}$ $\{0,1,2,3,4,5\}_1 = \{1,2,4,6\}$

$\{0,1,2,3,4\}, \{5\}, \{6\}$

$\{0,1,2,3,4\}_0 = \{1,3,5\}$

$\{0,1,2,3\}, \{4\}, \{5\}, \{6\}$

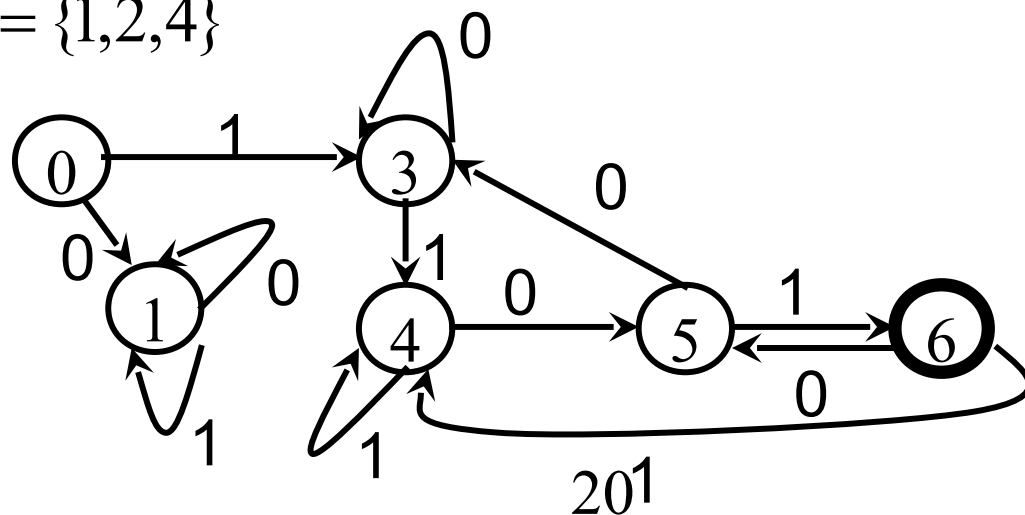
$\{0,1,2,3\}_0 = \{1,3\}$ $\{0,1,2,3\}_1 = \{1,2,4\}$

$\{0,1\}, \{2,3\}, \{4\}, \{5\}, \{6\}$

$\{0,1\}_0 = \{1\}$ $\{0,1\}_1 = \{1,2\}$

$\{2,3\}_0 = \{3\}$ $\{2,3\}_1 = \{4\}$

$\{0\}, \{1\}, \{2,3\}, \{4\}, \{5\}, \{6\}$

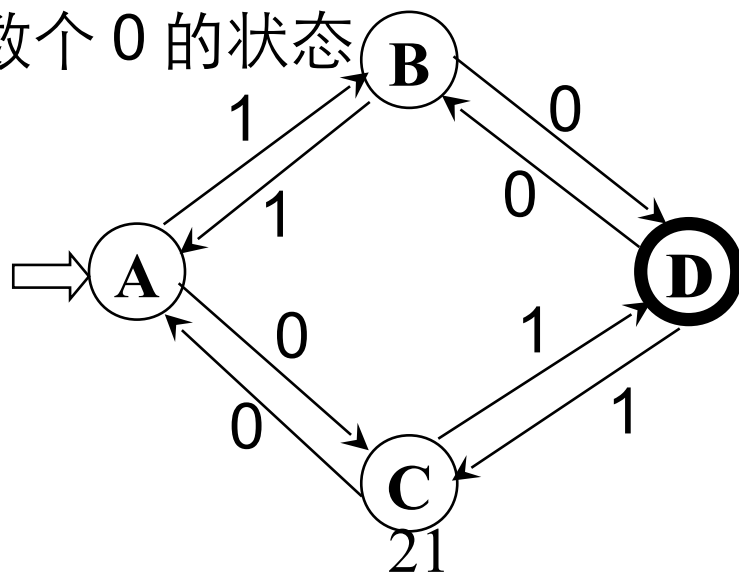


给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

- A：识别了偶数个 1 和偶数个 0 的状态
- B：识别了奇数个 1 和偶数个 0 的状态
- C：识别了偶数个 1 和奇数个 0 的状态
- D：识别了奇数个 1 和奇数个 0 的状态

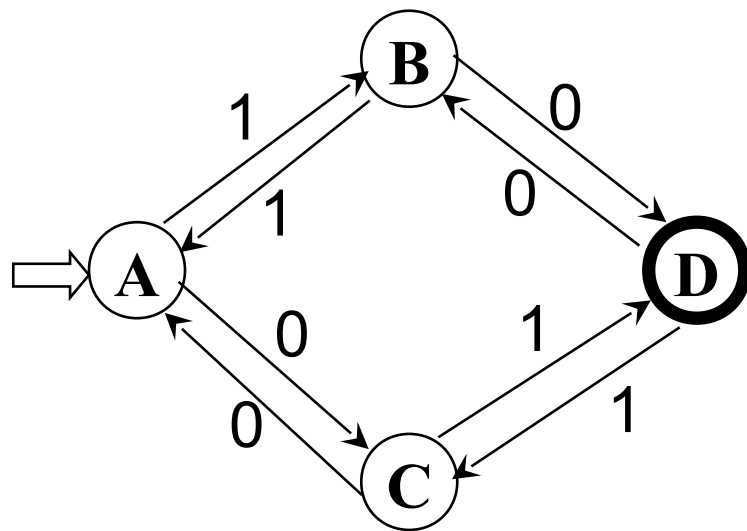
■ 将 NFA 转换成正规式



给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

- 先设计 NFA

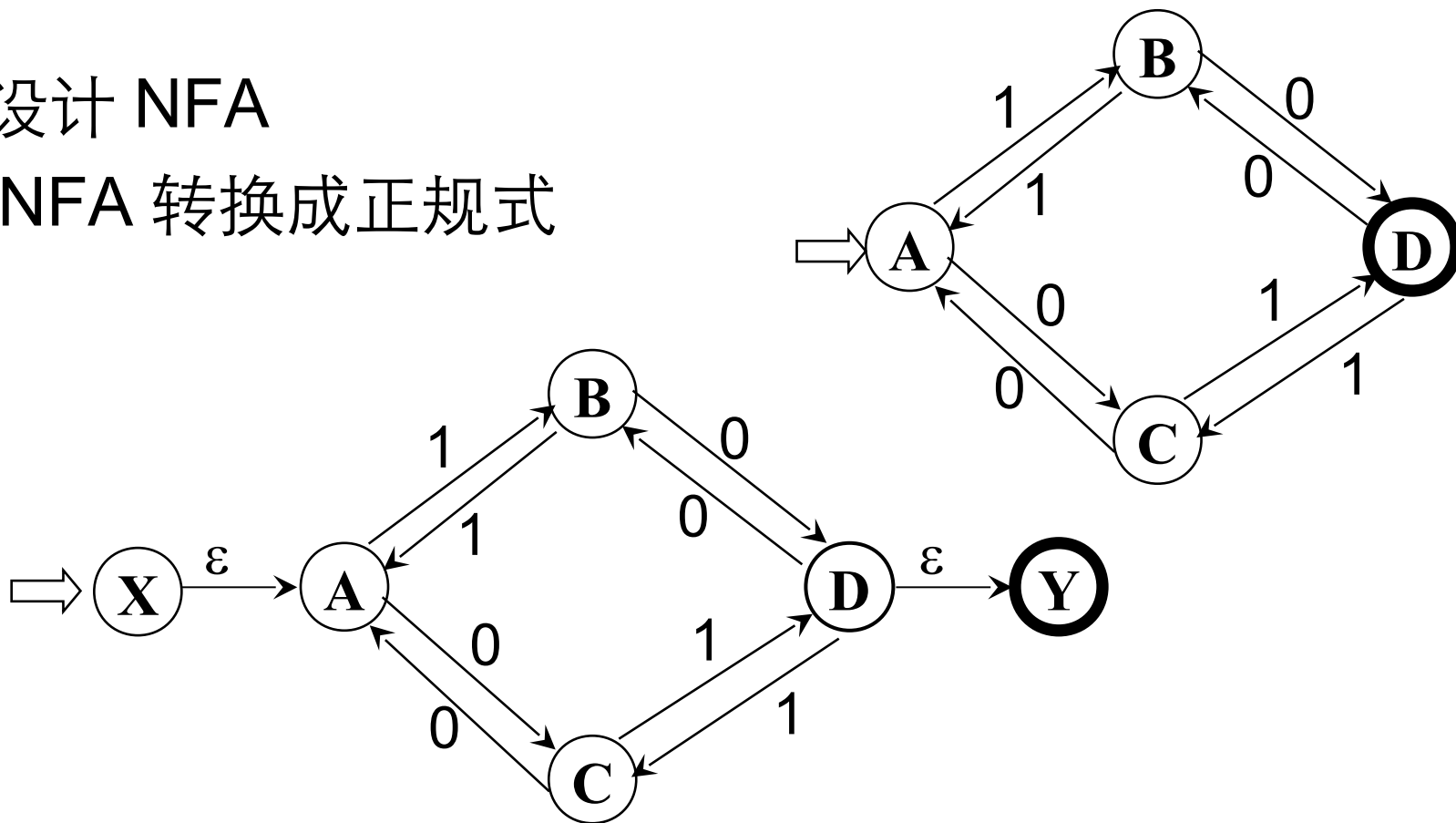
- 将 NFA 转换成正规式



给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

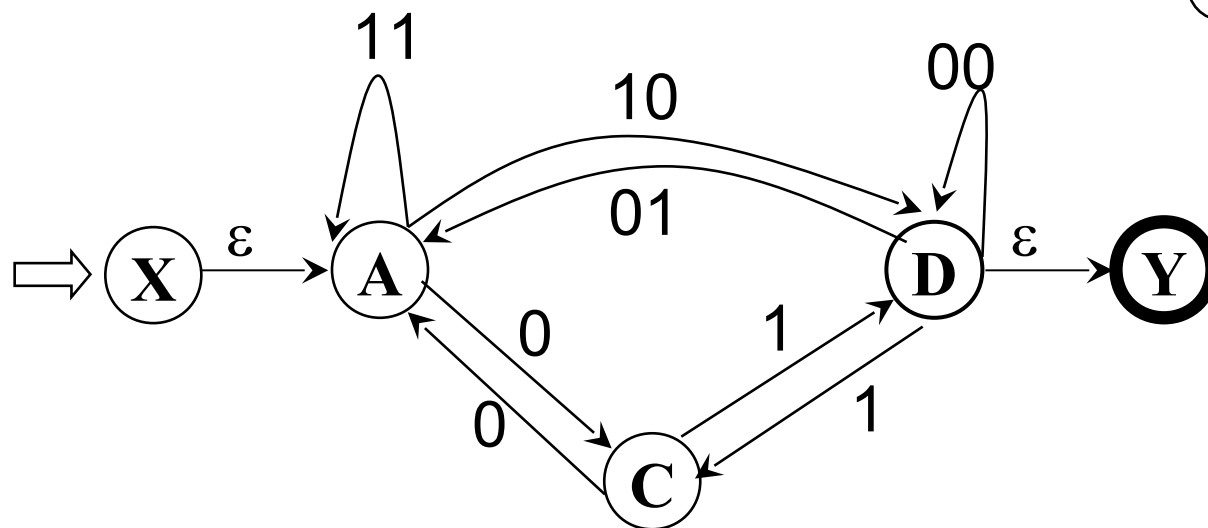
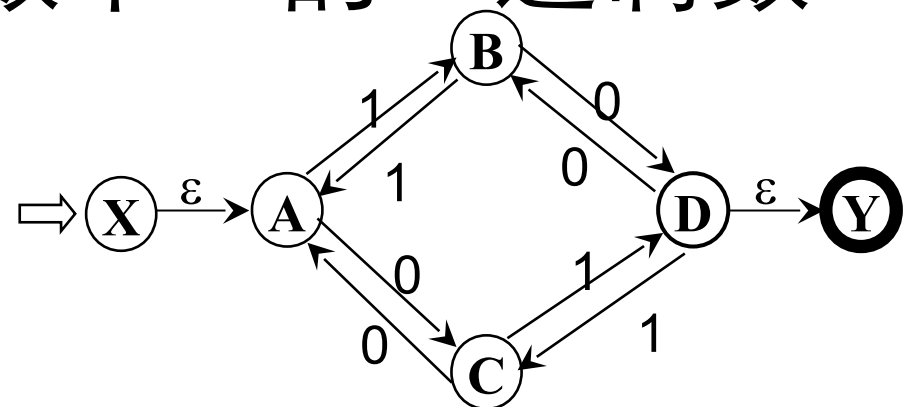
■ 将 NFA 转换成正规式



给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

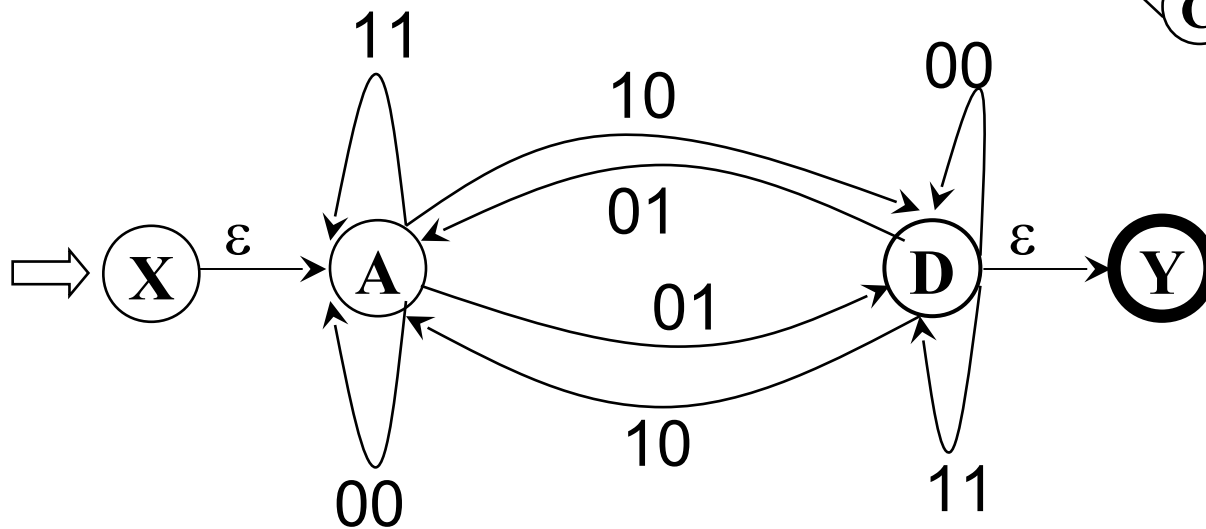
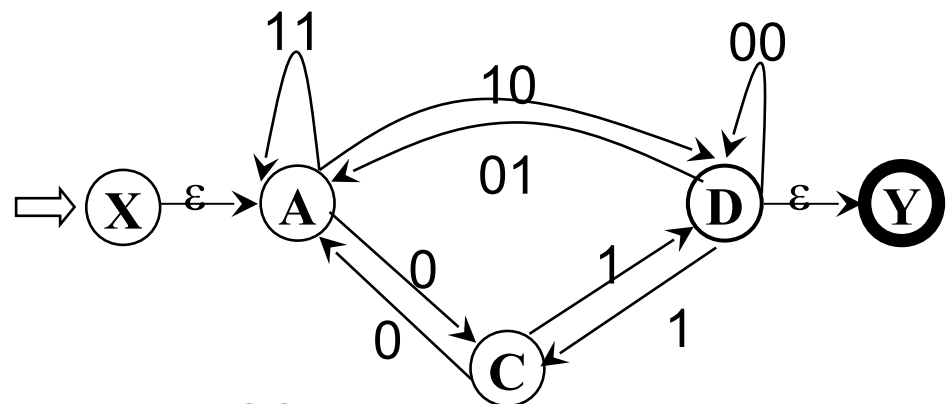
■ 将 NFA 转换成正规式



给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

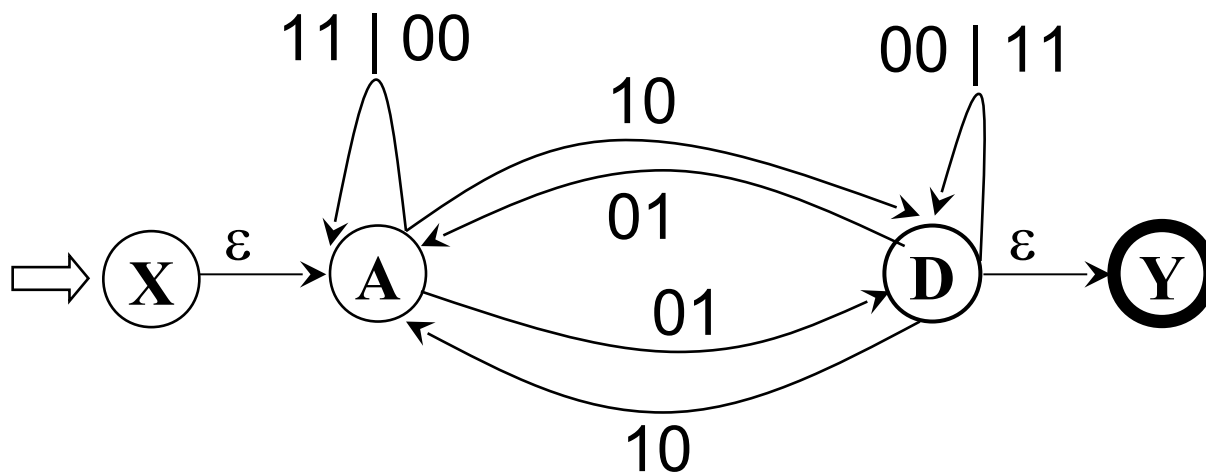
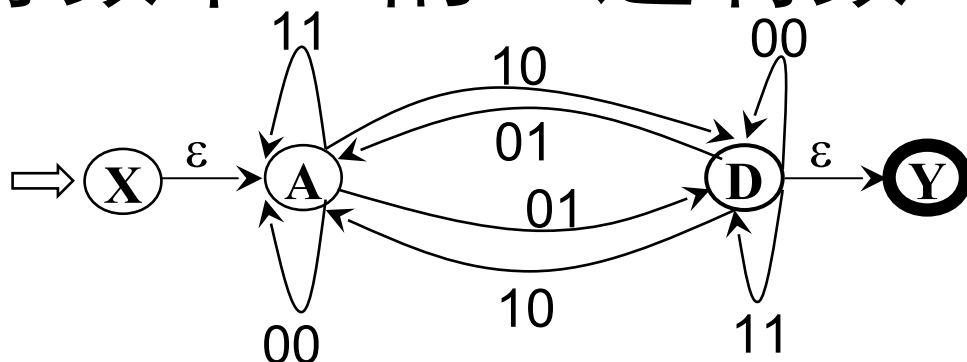
■ 将 NFA 转换成正规式



给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

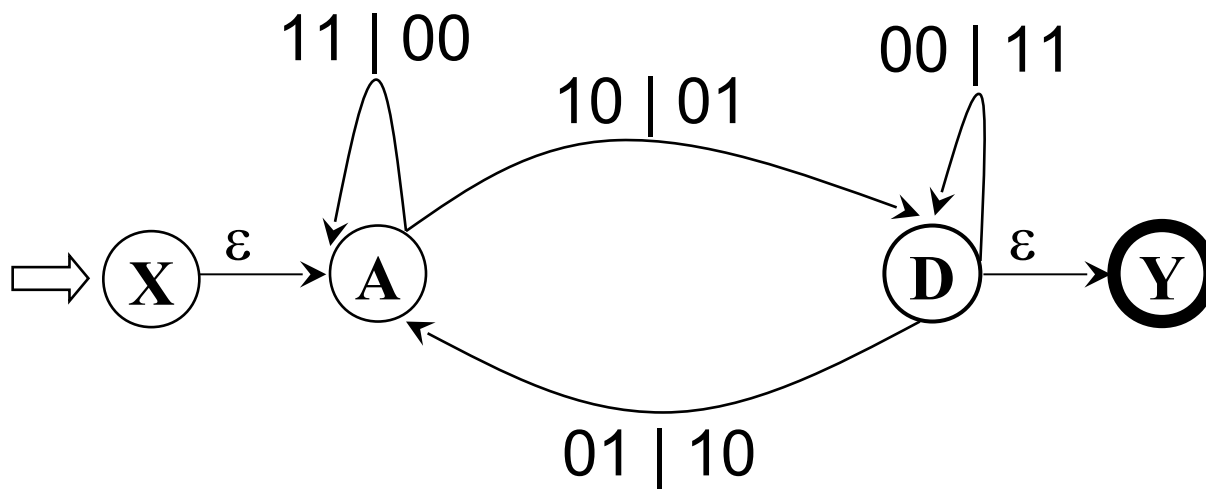
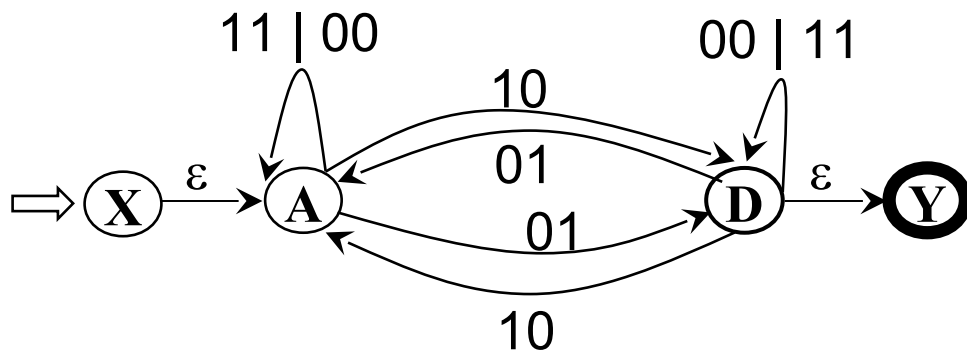
■ 将 NFA 转换成正规式



给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

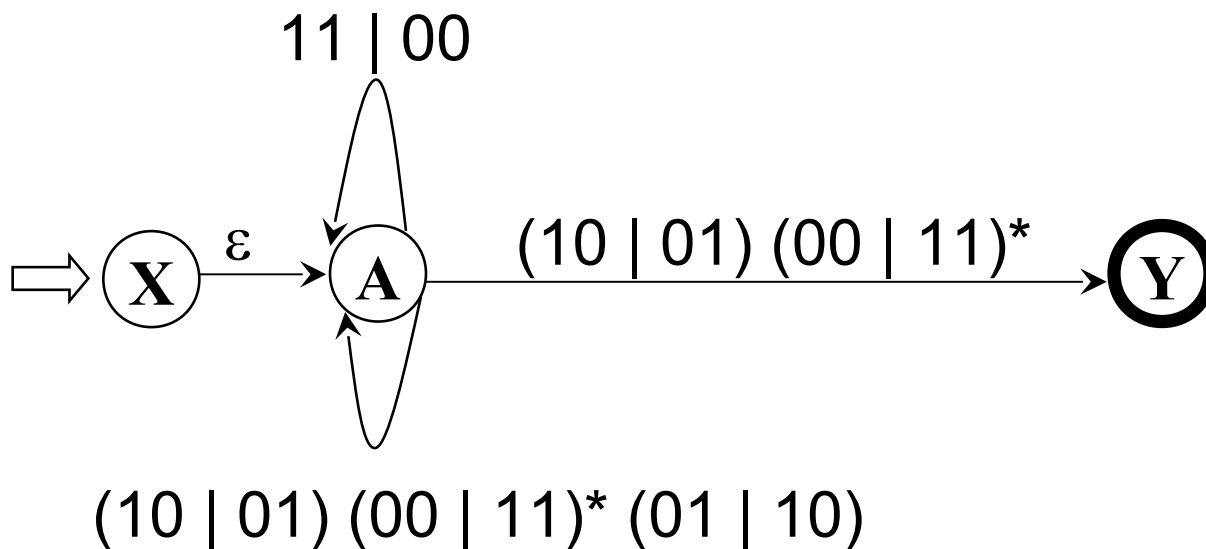
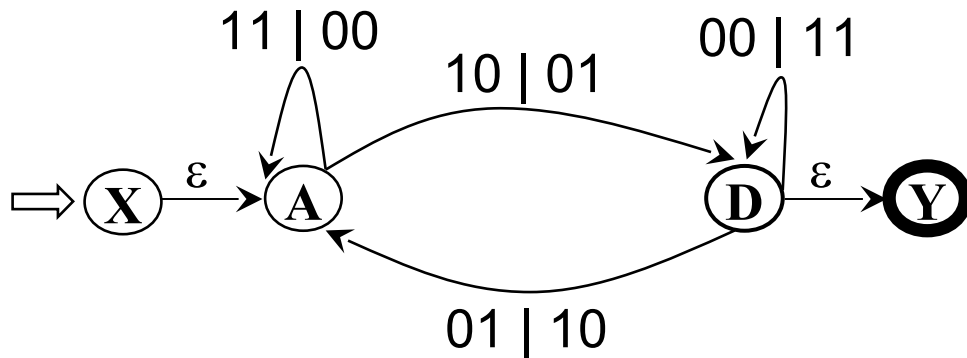
■ 将 NFA 转换成正规式 \Rightarrow



给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

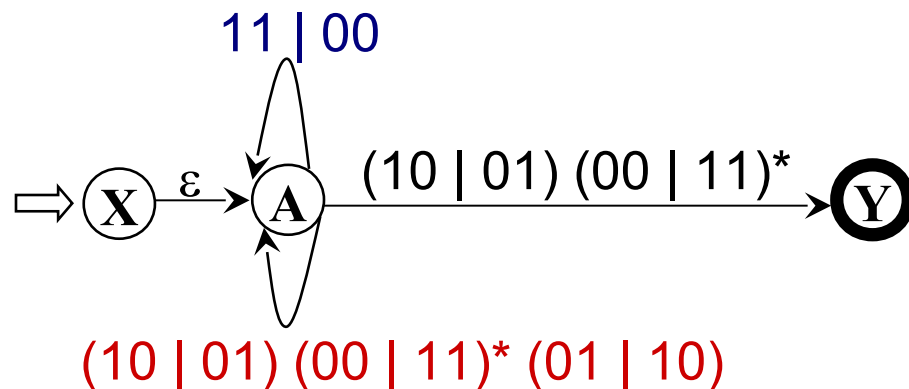
■ 将 NFA 转换成正规式 \Rightarrow



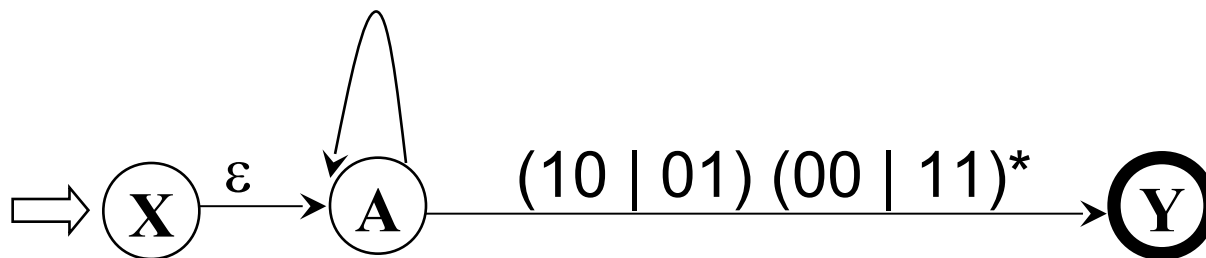
给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

■ 将 NFA 转换成正规式



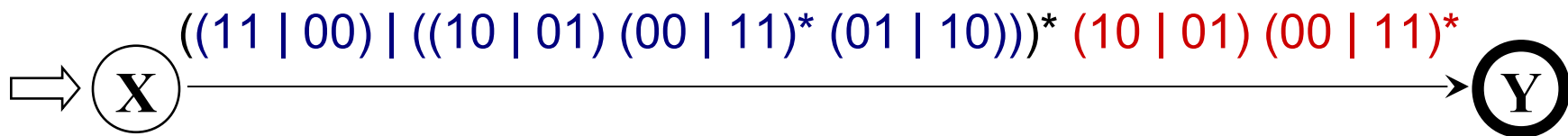
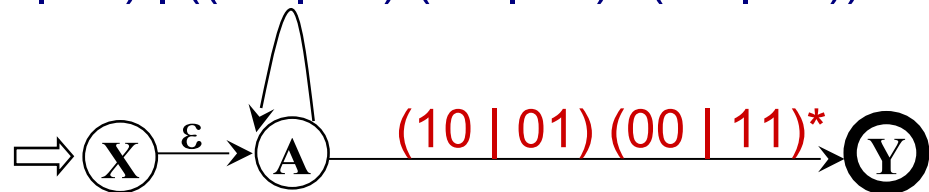
$(11 \mid 00) \mid ((10 \mid 01) (00 \mid 11)^* (01 \mid 10))$



给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

■ 将 NFA 转换成正规式 $(11 \mid 00) \mid ((10 \mid 01)(00 \mid 11)^*(01 \mid 10))$

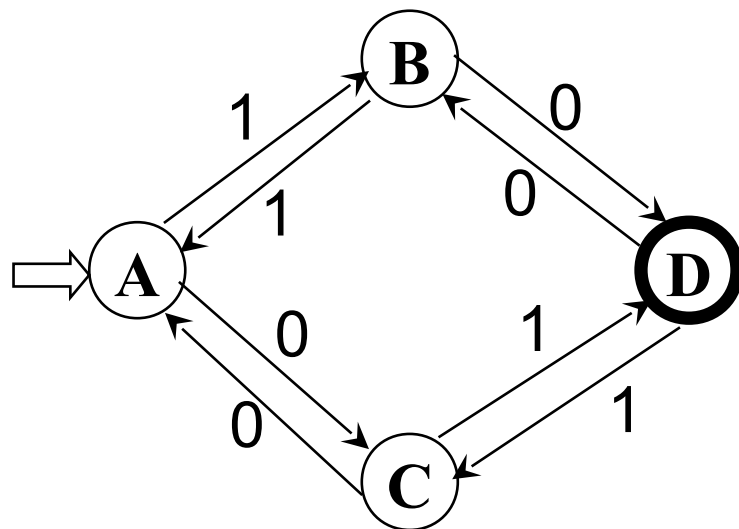


给出下面正规表达式：
包含奇数个 1 和奇数个 0 的二进制数串

■ 先设计 NFA

■ 将 NFA 转换成正规式

$((11 \mid 00) \mid ((10 \mid 01)(00 \mid 11)^*(01 \mid 10)))^*(10 \mid 01)(00 \mid 11)^*$



P65-14. 构造一个 DFA，它接受 $\Sigma = \{0,1\}$ 上所有满足如下条件的字符串：每个 1 都有 0 直接跟在右边。

■ 思路

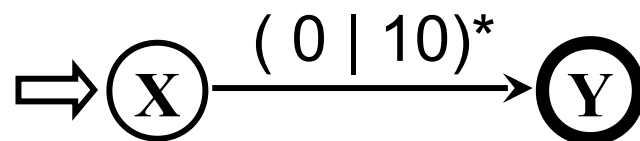
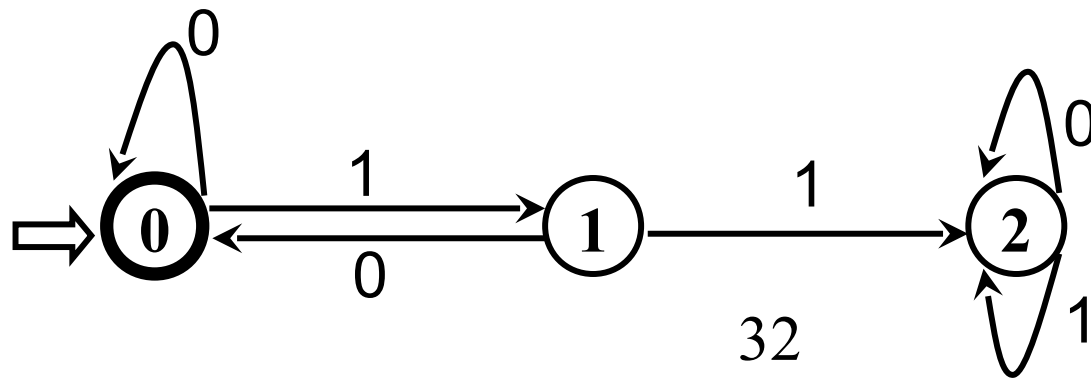
□ 分析语言特点

- 01000101000010

□ 写出正规式

- $(0 | 10)^*$

□ 正规式 \Rightarrow NFA \Rightarrow DFA



小结

- 文法与语言
- 正规式 vs. NFA vs. DFA



编译原理

习题课 (2)

第四章 语法分析—自上而下分析

- 语法分析器的功能
- 自上而下分析面临的问题
- LL(1) 分析法
- 递归下降分析程序构造
- 预测分析程序

语法分析的方法

■ 自上而下分析法 (Top-down)

□ 基本思想

- 它从文法的开始符号出发，反复使用各种产生式，寻找"匹配"的**推导**

□ **递归下降分析法**

- 对每一语法变量（非终结符）构造一个相应的子程序，每个子程序识别一定的语法单位
- 通过子程序间的相互调用实现对输入串的认识

□ **预测分析程序**

- 非递归实现
- 直观、简单

P81-1. 考虑下面文法 $G_1(S)$:

$$S \rightarrow a \mid \wedge \mid (T)$$
$$T \rightarrow T, S \mid S$$

(1) 消去 G_1 的左递归。然后，对每个非终结符，写出不带回溯的递归子程序。

(2) 经改写后的文法是否是 LL (1) 的？给出它的预测分析表。

■ 思路

- ☐ 消除左递归
- ☐ 提取左公共因子
- ☐ 计算非终结符的 FIRST 集合和 FOLLOW 集合
- ☐ 检查 LL(1) 条件
- ☐ 构造预测分析表或递归子程序

$G_1(S) :$

$S \rightarrow a \mid \wedge \mid (T)$

$T \rightarrow T, S \mid S$

■ 消除左递归：按照 T, S 的顺序消除左递归

■ $G'_1(S) :$

$S \rightarrow a \mid \wedge \mid (T)$

$T \rightarrow S T'$

$T' \rightarrow , S T' \mid \varepsilon$

■ 无左公共因子

$$\begin{aligned}
 G'_1(S) : \quad & FIRST(\alpha) = \{a \mid \alpha \Rightarrow^* a..., a \in V_T\} \\
 S \rightarrow a \mid \wedge \mid (T) \\
 T \rightarrow S T' \quad & FOLLOW(A) = \{a \mid S \Rightarrow^* ...Aa..., a \in V_T\} \\
 T' \rightarrow , S T' \mid \varepsilon
 \end{aligned}$$

■ 计算非终结符的 FIRST 和 FOLLOW 集合

- $FIRST(S) = \{a, \wedge, (\}$
- $FIRST(T) = \{a, \wedge, (\}$
- $FIRST(T') = \{ , , \varepsilon \}$
- $FOLLOW(S) = \{), , , \# \}$
- $FOLLOW(T) = \{) \}$
- $FOLLOW(T') = \{) \}$

■ 检查 LL(1) 条件

构造不带回溯的自上而下分析的文法条件

1. 文法不含左递归
2. 对于文法中每一个非终结符 A 的各个产生式的候选首符集两两不相交。即，若

$$A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$$

则 $\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi \quad (i \neq j)$

3. 对文法中的每个非终结符 A ，若它存在某个候选首符集包含 ε ，则

$$\text{FIRST}(\alpha_i) \cap \text{FOLLOW}(A) = \phi$$

$$i=1, 2, \dots, n$$

如果一个文法 G 满足以上条件，则称该文法 G 为 **LL(1) 文法**。

$G'_1(S) :$

$S \rightarrow a \mid \wedge \mid (T)$

$T \rightarrow S T'$

$T' \rightarrow , S T' \mid \varepsilon$

■ 计算非终结符的 FIRST 和 FOLLOW 集合

□ $\text{FIRST}(S) = \{a, \wedge, (\}$

□ $\text{FIRST}(T) = \{a, \wedge, (\}$

□ $\text{FIRST}(T') = \{ , , \varepsilon \}$

□ $\text{FOLLOW}(S) = \{), , , \# \}$

□ $\text{FOLLOW}(T) = \{) \}$

□ $\text{FOLLOW}(T') = \{) \}$

■ 检查 LL(1) 条件

□ 满足

分析表 $M[A, a]$ 的构造

- 在每个非终结符 A 及其任意候选 α 都构造出 $FIRST(\alpha)$ 和 $FOLLOW(A)$ 的基础上
 - 构造 G 的分析表 $M[A, a]$ ，确定每个产生式 $A \rightarrow \alpha$ 在表中的位置
1. 对文法 G 的每个产生式 $A \rightarrow \alpha$ 执行第 2 步和第 3 步；
 2. 对每个终结符 $a \in FIRST(\alpha)$ ，把 $A \rightarrow \alpha$ 加至 $M[A, a]$ 中；
 3. 若 $\varepsilon \in FIRST(\alpha)$ ，则对任何 $b \in FOLLOW(A)$ 把 $A \rightarrow \alpha$ 加至 $M[A, b]$ 中。
 4. 把所有无定义的 $M[A, a]$ 标上“出错标志”。

$G'_1(S) :$

$S \rightarrow a \mid \wedge \mid (T)$

$T \rightarrow S T'$

$T' \rightarrow , S T' \mid \varepsilon$

$FIRST(S) = \{a, \wedge, (\}$

$FIRST(T) = \{a, \wedge, (\}$

$FIRST(T') = \{ , , \varepsilon \}$

$FOLLOW(S) = \{), , , \# \}$

$FOLLOW(T) = \{) \}$

$FOLLOW(T') = \{) \}$

■ 构造预测分析表

	a	\wedge	()	,	#
S	$S \rightarrow a$	$S \rightarrow \wedge$	$S \rightarrow (T)$			
T	$T \rightarrow S T'$	$T \rightarrow S T'$	$T \rightarrow S T'$			
T'				$T' \rightarrow \varepsilon$	$T' \rightarrow , S T'$	

$G'_1(S) :$

$S \rightarrow a \mid \wedge \mid (T)$

$T \rightarrow S T'$

$T' \rightarrow , S T' \mid \varepsilon$

■ 构造递归子程序

```
procedure T';  
begin  
    if sym=',' then begin  
        advance;  
        S;T'  
    end  
end;
```

```
procedure S;  
begin  
    if sym='a' or sym='^'  
    then advance  
    else if sym='('  
        then begin  
            advance;T;  
            if sym=')' then advance;  
            else error;  
        end  
    else error  
end;
```

```
procedure T;  
begin  
    S;T'  
end;
```

第五章 语法分析——自下而上分析

- 自下而上分析的基本问题
- 算符优先分析算法
- LR 分析法

语法分析的方法

■ 自下而上分析法 (Bottom-up)

□ 基本思想

- 从输入串开始，逐步进行**归约**，直到文法的开始符号
- **归约**：根据文法的产生式规则，把产生式的右部替换成左部符号
- 从树末端开始，构造语法树

□ **算符优先分析法**

- 按照算符的优先关系和结合性质进行语法分析
- 适合分析表达式

□ **LR 分析法**

- 规范归约

短语、直接短语、句柄和素短语

- 定义：令 G 是一个文法， S 是文法的开始符号，假定 $\alpha\beta\delta$ 是文法 G 的一个句型，如果有

$$S \Rightarrow \alpha A \delta \text{ 且 } A \Rightarrow^+ \beta$$

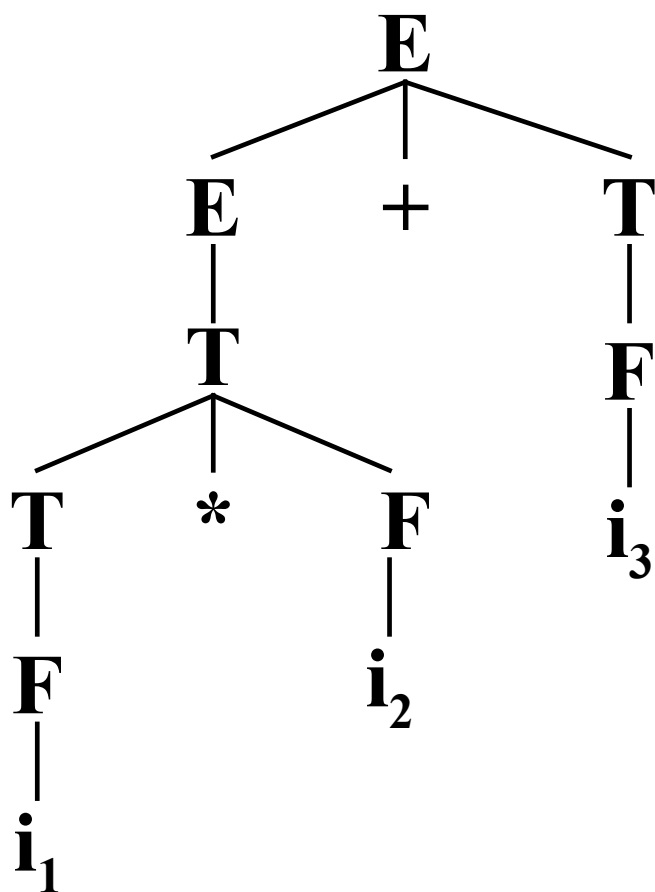
则 β 称是句型 $\alpha\beta\delta$ 相对于非终结符 A 的**短语**。

特别是，如果有 $A \Rightarrow \beta$ ，则称 β 是句型 $\alpha\beta\delta$ 相对于规则 $A \rightarrow \beta$ 的**直接短语**。一个句型的最左直接短语称为该句型的**句柄**。

一个文法 G 的句型的**素短语**是指这样一个短语，它至少含有一个终结符，并且，除它自身之外不再含任何更小的素短语

最左素短语是指处于句型最左边的那个素短语

短语、直接短语和句柄



■ 在一个句型对应的语法树中

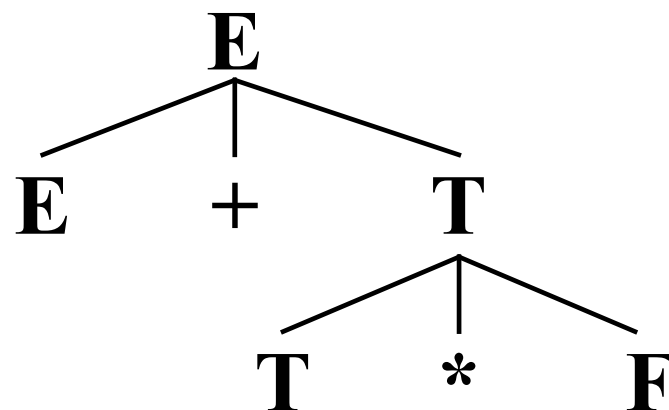
- 以某非终结符为根的两代以上的子树的所有末端结点从左到右排列就是相对于该非终结符的一个短语
- 如果子树只有两代，则该短语就是直接短语

P133-1. 令文法 G_1 为:

$$E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid i$$

证明 $E+T*F$ 是它的一个句型，指出这个句型的所有短语，直接短语和句柄。

- 短语： $E+T*F$, $T*F$
- 直接短语： $T*F$
- 句柄： $T*F$



构造集合 FIRSTVT(P) 的算法

$$FIRSTVT(P) = \{a \mid P \overset{+}{\Rightarrow} a \cdots, \text{ 或 } P \overset{+}{\Rightarrow} Qa \cdots, a \in V_T \text{ 而 } Q \in V_N\}$$

- 反复使用下面两条规则构造集合 FIRSTVT(P)
 1. 若有产生式 $P \rightarrow a \dots$ 或 $P \rightarrow Qa \dots$ ，则 $a \in FIRSTVT(P)$
 2. 若 $a \in FIRSTVT(Q)$ ，且有产生式 $P \rightarrow Q \dots$ ，
将对推导的遍历转换成对产生式的反复遍历

构造集合 LASTVT(P) 的算法

$$LASTVT(P) = \{a \mid P \overset{+}{\Rightarrow} \cdots a, \text{ 或 } P \overset{+}{\Rightarrow} \cdots aQ, a \in V_T \text{ 而 } Q \in V_N\}$$

■ 反复使用下面两条规则构造集合 LASTVT(P)

1. 若有产生式 $P \rightarrow \dots a$ 或 $P \rightarrow \dots aQ$ ，则 $a \in LASTVT(P)$ ；
2. 若 $a \in LASTVT(Q)$ ，且有产生式 $P \rightarrow \dots Q$ ，则 $a \in LASTVT(P)$ 。

构造优先关系表算法

- 通过检查 G 的每个产生式的每个候选式，可找出所有满足 $a \prec b$ 的终结符对
- 通过检查每个产生式的候选式确定满足关系 \prec 和 \succ 的所有终结符对

□ 假定有个产生式的一个候选形为

$\dots aP\dots$

那么，对任何 $b \in \text{FIRSTVT}(P)$ ，有 $a \prec b$

□ 假定有个产生式的一个候选形为

$\dots Pb\dots$

那么，对任何 $a \in \text{LASTVT}(P)$ ，有 $a \succ b$

P133-3.

(1) 计算 $G_2(S)$:

$$S \rightarrow a \mid \wedge \mid (T)$$
$$T \rightarrow T, S \mid S$$

的 FIRSTVT 和 LASTVT 。

(2) 计算 G_2 的优先关系。 G_2 是一个算符优先文法吗？

(3) 计算 G_2 的优先函数。

■ 思路 给出输入串 $(a, (a, a))$ 的算符优先分析过程。

- 计算非终结符的 FIRSTVT 和 LASTVT
- 计算终结符之间的优先关系
- 检查算符优先文法的条件
- 构造算符优先函数

$G_2(S)$:

$S \rightarrow a \mid \wedge \mid (T)$

$T \rightarrow T, S \mid S$

- $FIRSTVT(S) = \{ a, \wedge, (\}$
- $FIRSTVT(T) = \{ ,, a, \wedge, (\}$
- $LASTVT(S) = \{ a, \wedge,) \}$
- $LASTVT(T) = \{ ,, a, \wedge,) \}$

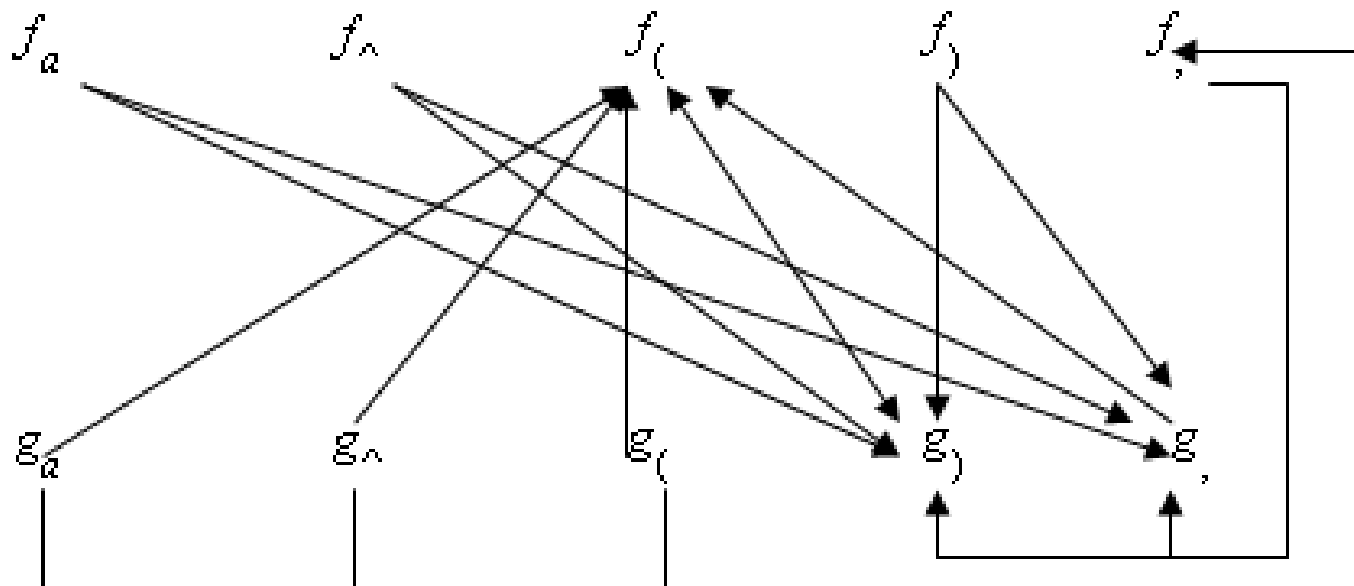
	a	\wedge	()	,
a					
\wedge					
(
)					
,					

该文法是算符文法，并且是算符优先文法

根据优先表构造优先函数

1. **画图**：对于每个终结符 a ，令其对应两个符号 f_a 和 g_a ，画一以所有符号和为结点的方向图。如果 $a < b$ ，则从 f_a 画一条弧至 g_b ，如果 $a > b$ ，则画一条弧从 g_b 至 f_a 。
2. **数数**：对每个结点都赋予一个数，此数等于从该结点出发所能到达的结点（包括出发点自身）。赋给 f_a 的数作为 $f(a)$ ，赋给 g_a 的数作为 $g(a)$ 。
3. **验证**：检查所构造出来的函数 f 和 g 是否与原来的关系矛盾。若没有矛盾，则 f 和 g 就是要求的优先函数，若有矛盾，则不存在优先函数。

优先函数



	a	\wedge	()	,
f	4	4	2	4	4
g	5	5	5	2	3

算符优先分析

- 算符优先文法句型（括在两个 # 之间）的一般形式：
：

$$\#N_1a_1N_2a_2\ldots N_na_nN_{n+1}\#$$

其中， a_i 是终结符， N_i 是可有可无的非终结符。

- 定理：一个算符优先文法 G 的任何句型的最左素短语是满足如下条件的最左子串 $N_ja_j\ldots N_ia_iN_{i+1}$ ，

$$\begin{array}{c} a_{j-1} \blacklozenge a_j \\ a_j \blacklozenge a_{j+1}, \quad \dots, \quad a_{i-1} \blacklozenge a_i \\ a_i \square a_{i+1} \end{array}$$

栈	输入字符串
#	(a, (a,a))#
#(a, (a,a)) #
#(a	, (a,a)) #
#(S	, (a,a)) #
#(S,	(a,a))#
#(S,(a,a))#
#(S,(a	,a))#
#(S,(S	,a))#
#(S,(S,	a))#
#(S,(S,a))#
#(S,(S,S))#
#(S,(T))#
#(S,(T))#
#(S, S)#
#(T)#
#(T)	#
# S	#

success

动作
 预备
 进
 进
 进
 归
 进
 进
 进
 进
 进
 归
 进
 归
 归
 进
 归
 归

	a	^	()	,
a					
^					
(
)					
,					

P134-5. 考虑文法

$$S \rightarrow AS \mid b$$
$$A \rightarrow SA \mid a$$

- (1) 列出这个文法的所有 LR(0) 项目。
- (2) 构造这个文法的 LR(0) 项目集规范族及识别活前缀的 DFA。
- (3) 这个文法是 SLR 的吗？若是，构造出它的 SLR 分析表。

P134-5. 考虑文法

$$S \rightarrow AS \mid b$$
$$A \rightarrow SA \mid a$$

(1) 列出这个文法的所有 LR (0) 项目。

■ 对文法进行拓广： $S' \rightarrow S$

■ LR (0) 项目：

0. $S' \rightarrow .S$

1. $S' \rightarrow S.$

2. $S \rightarrow .AS$

3. $S \rightarrow A.S$

4. $S \rightarrow AS.$

5. $S \rightarrow .b$

6. $S \rightarrow b.$

7. $A \rightarrow .SA$

8. $A \rightarrow S.A$

9. $A \rightarrow SA.$

10. $A \rightarrow .a$

11. $A \rightarrow a.$

P134-5. 考虑文法

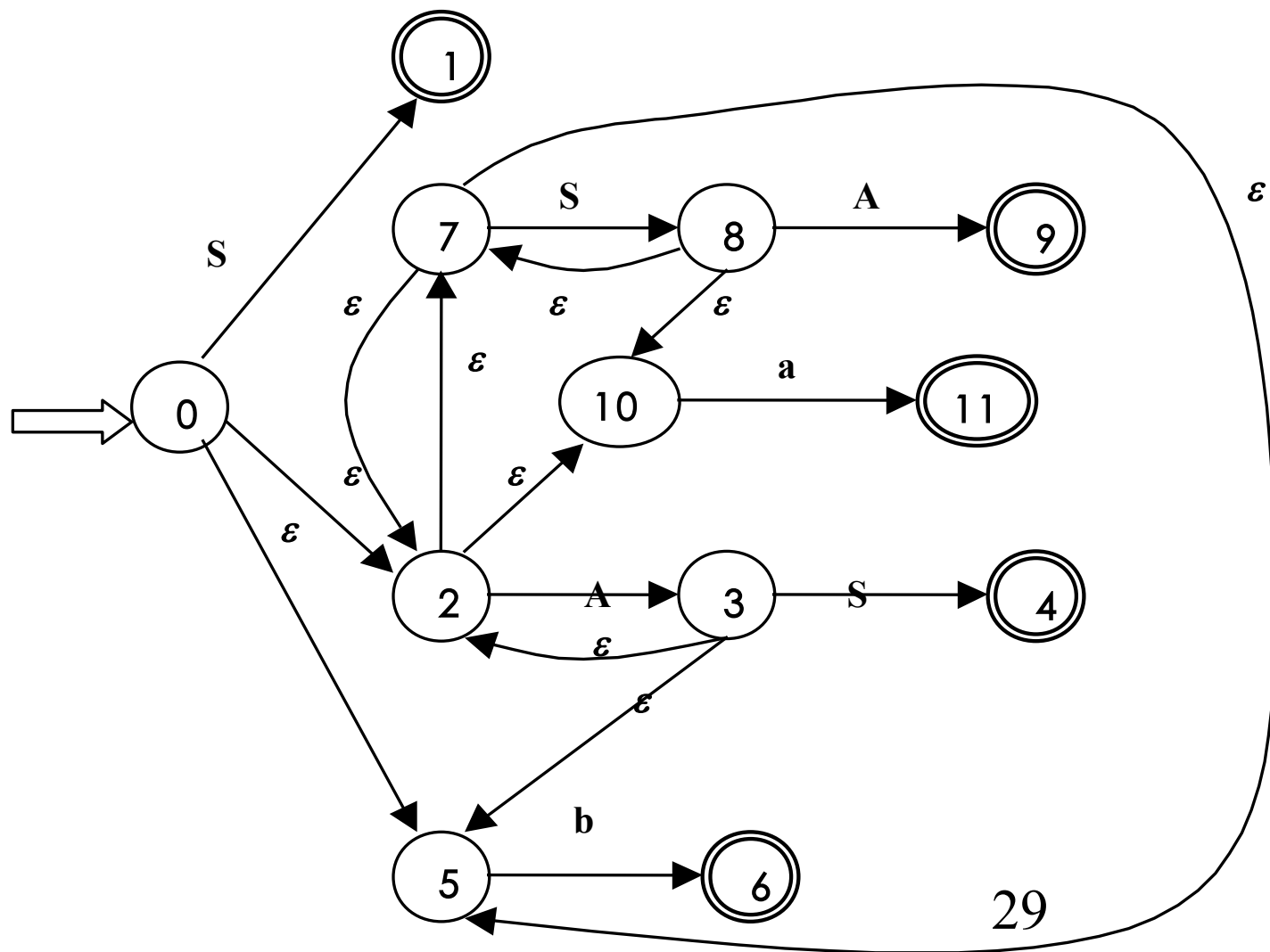
$$S \rightarrow AS \mid b$$
$$A \rightarrow SA \mid a$$

- (1) 列出这个文法的所有 LR (0) 项目。
- (2) 构造这个文法的 LR (0) 项目集规范族及识别活前缀的 DFA 。

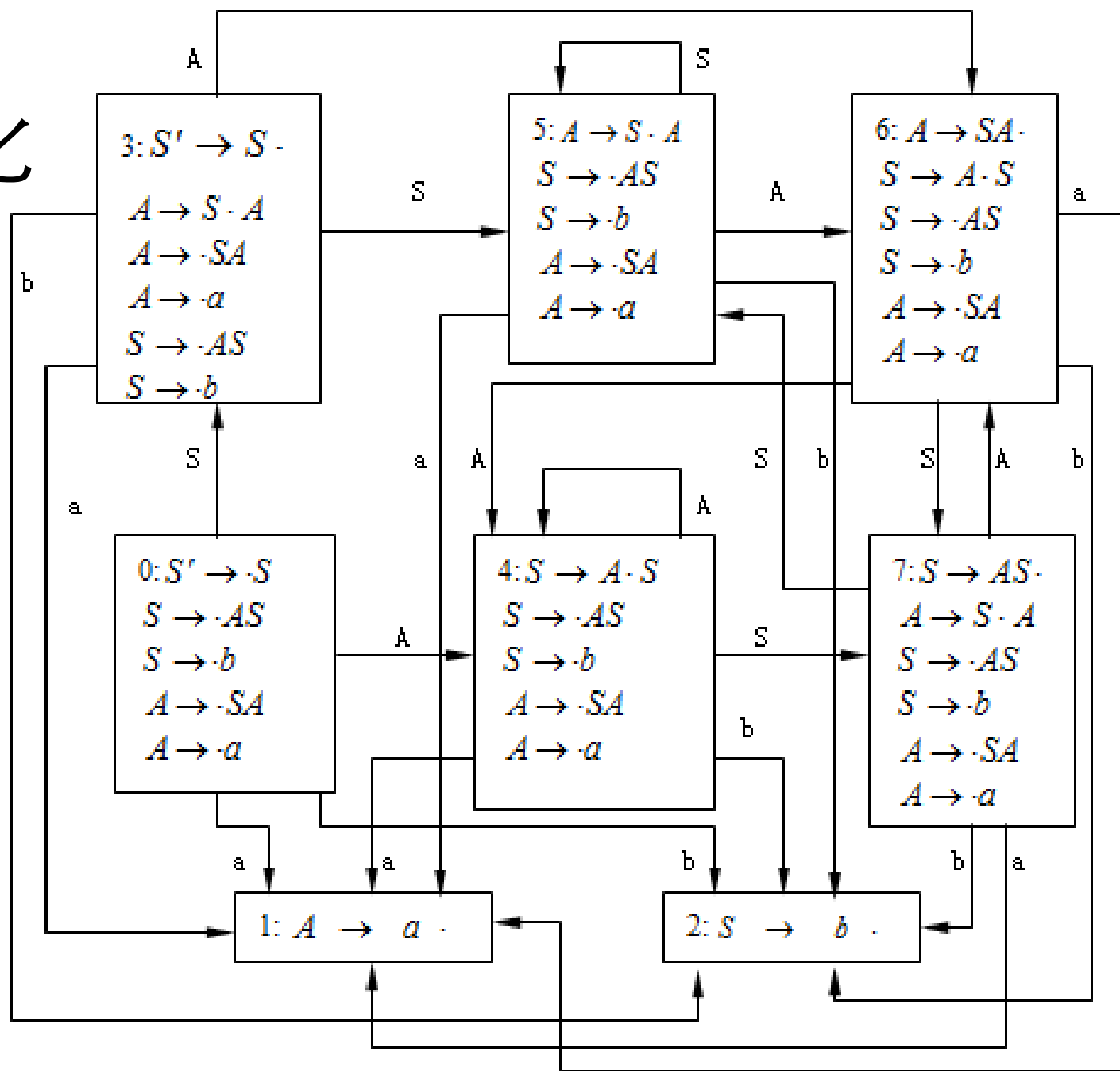
■ 构造 LR(0) 项目规范族，两种方法：

- 利用有限自动机来构造
- 利用函数 CLOSURE 和 GO 来构造

识别活前缀的 DFA



确定化



利用函数 CLOSURE 和 GO 来构造项目集规范族

$$I_0 = \{ S' \rightarrow .S, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .SA, A \rightarrow .a \}$$

$$GO(I_0, a) = \{ A \rightarrow a. \} = I_1$$

$$GO(I_0, b) = \{ S \rightarrow b. \} = I_2$$

$$GO(I_0, S) = \{ S' \rightarrow S., A \rightarrow S.A, A \rightarrow .SA, A \rightarrow .a, S \rightarrow .AS, S \rightarrow .b \} = I_3$$

$$GO(I_0, A) = \{ S \rightarrow A.S, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .AS, A \rightarrow .a \} = I_4$$

$$GO(I_3, a) = \{ A \rightarrow a. \} = I_1$$

$$GO(I_3, b) = \{ A \rightarrow b. \} = I_2$$

$$GO(I_3, S) = \{ A \rightarrow S.A, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .SA, A \rightarrow .a \} = I_5$$

$$GO(I_3, A) = \{ A \rightarrow SA., A \rightarrow S.A, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .SA, A \rightarrow .a \} = I_6$$

$$\text{GO}(I_4, a) = \{ A \rightarrow a. \} = I_1$$

$$\text{GO}(I_4, b) = \{ S \rightarrow b. \} = I_2$$

$$\text{GO}(I_4, S) = \{ S \rightarrow AS., A \rightarrow S.A, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .AS, A \rightarrow .a \} = I_7$$

$$\text{GO}(I_4, A) = \{ S \rightarrow A.S, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .AS, A \rightarrow .a \} = I_4$$

$$\text{GO}(I_5, a) = \{ A \rightarrow a. \} = I_1$$

$$\text{GO}(I_5, b) = \{ A \rightarrow b. \} = I_2$$

$$\text{GO}(I_5, S) = \{ A \rightarrow S.A, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .SA, A \rightarrow .a \} = I_5$$

$$\text{GO}(I_5, A) = \{ A \rightarrow SA., A \rightarrow S.A, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .SA, A \rightarrow .a \} = I_6$$

$$\text{GO}(I_6, a) = \{ A \rightarrow a. \} = I_1$$

$$\text{GO}(I_6, b) = \{ S \rightarrow b. \} = I_2$$

$$\text{GO}(I_6, S) = \{ S \rightarrow AS., A \rightarrow S.A, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .AS, A \rightarrow .a \} = I_7$$

$$\text{GO}(I_6, A) = \{ S \rightarrow A.S, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .AS, A \rightarrow .a \} = I_4$$

$$\text{GO}(I_7, a) = \{ A \rightarrow a. \} = I_1$$

$$\text{GO}(I_7, b) = \{ A \rightarrow b. \} = I_2$$

$$\text{GO}(I_7, S) = \{ A \rightarrow S.A, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .SA, A \rightarrow .a \} = I_5$$

$$\text{GO}(I_7, A) = \{ A \rightarrow SA., A \rightarrow S.A, S \rightarrow .AS, S \rightarrow .b, A \rightarrow .SA, A \rightarrow .a \} = I_6$$

项目集规范族为 $C = \{I_0, I_1, I_2, I_3, I_4, I_5, I_6, I_7\}$

P134-5. 考虑文法

$$S \rightarrow AS \mid b$$
$$A \rightarrow SA \mid a$$

- (1) 列出这个文法的所有 LR (0) 项目。
- (2) 构造这个文法的 LR (0) 项目集规范族及识别活前缀的 DFA。
- (3) 这个文法是 SLR 的吗？若是，构造出它的 SLR 分析表。

SLR 文法判断

- 状态 3 , 6 , 7 有移进归约冲突
- $I_3 = \{ S' \rightarrow S. , A \rightarrow S.A , A \rightarrow .SA , A \rightarrow .a , S \rightarrow .AS , S \rightarrow .b \}$
 - $FOLLOW(S') = \{ \# \}$ 不包含 a,b ; 冲突可以消解
- $I_6 = \{ A \rightarrow SA. , A \rightarrow S.A , S \rightarrow .AS , S \rightarrow .b , A \rightarrow .SA , A \rightarrow .a \}$
 - $FOLLOW(A) \cap \{ a, b \} = \{ a, b \} \neq \Phi$, 冲突无法消解
- $I_7 = \{ S \rightarrow AS. , A \rightarrow S.A , S \rightarrow .AS , S \rightarrow .b , A \rightarrow .AS , A \rightarrow .a \}$
 - $FOLLOW(S) = \{ \#, a, b \}$, 冲突无法消解
- 所以不是 SLR 文法。
- 这个文法也不是 LR(1) 文法。

小结

- 自上而下分析
 - LL(1) 分析条件
 - 递归下降分析程序
 - 预测分析程序
- 自下而上分析
 - 可归约串：句柄、最左素短语
 - 算符优先分析
 - LR 分析

1.对于循环语句:

for $i:=E^{(1)}$ step $E^{(2)}$ until $E^{(3)}$ do $S^{(1)}$;

其语义是(假定 $E^{(2)}$ 的值总是正的):

```
i:= $E^{(1)}$ ;
INCR:= $E^{(2)}$ ;
LIMIT:= $E^{(3)}$ ;
goto OVER;
AGAIN: i:=i+INCR;
OVER:  if  $i \leq$  LIMIT then
    begin
         $S^{(1)}$ ;
        goto AGAIN
    end
```

由于 $E^{(2)}$ 和 $E^{(3)}$ 都只计算一次,因此,改语句的文法为

$F \rightarrow \text{for } i:=E^{(1)} \text{ step } E^{(2)} \text{ until } E^{(3)}$

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

请写出相应的语义子程序。

(国防科技大学 1996 年硕士生入学考试试题)

解答:

$F \rightarrow \text{for } i:=E^{(1)} \text{ step } E^{(2)} \text{ until } E^{(3)}$

```
{
     $E^{(1)}$ 
    GEN(:=, .PLACE, -, ENTRY(i));
    F.PLACE:=ENTRY(i);
    INCR:=NEWTEMP;
     $E^{(2)}$ 
    GEN(:=, .PLACE, -, INCR);
    LIMIT=NEWTEMP;
    GEN(:=,  $E^{(3)}$ .PLACE, -, LIMIT);
    q:=NXQ;
    GEN(j, -, -, q+2)
    GEN(+, F.PLACE, INCR, F.PALCE)
    F.QUAD:=q+2;    //again
    GEN(j $\leq$ , F.PLACE, LIMIT, q+4);
    F.CHAIN := NXQ;
    GEN(j, -, -, 0)
}
```

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

```
{
```

S(1)

```
BACKPATCH(.CHAIN, NXQ);  
GEN(j, -, -, F.QUAD);  
S.CHAIN:= F.CHAIN;  
}
```

2. 给定语句

repeat S until E

- (1) 写出适合语法制导翻译的产生式;
- (2) 写出每个产生式对应的语义动作。

(国防科技大学 1997 年硕士生入学考试试题)

解答:

(1)

$$\begin{aligned} R &\rightarrow repeat \\ U &\rightarrow R \ S \ until \\ S &\rightarrow U \ E \end{aligned}$$

(2)语义动作:

$$R \rightarrow repeat$$

```
{ R.QUAD:=NXQ }
```

$U \rightarrow R \ S \ \text{until}$

```
{ U.QUAD:=R.QUAD;  
  BACKPATCH(S.CHAIN, NXQ) }
```

$S \rightarrow U \ E$

```
{ BACKPATCH(E.FC, U.QUAD);  
  S.CHAIN:=E.TC }
```

例题 7.4.1 设某语言的 for 语句的形式:

$$S \rightarrow \text{for } i := E^{(1)} \text{ to } E^{(2)} \text{ do } S^{(1)}$$

其语义解释为

$E^{(1)}$
 $i :=$;

$E^{(2)}$
LIMIT := ;

again: if $i \leq$ LIMIT then

Begin

$S^{(1)}$;

$i := i + 1$;

goto again

End;

(1). 写出适合语法制导翻译的产生式;

(2). 写出每个产生式对应的语义动作。

解题思路:

与上题相比, 本题的语义解释已勾画出把该 FOR 语句翻译为中间代码后的框架结构。语法制导翻译过程中, 当扫描到关键字 do 时, 需要做一定的语义工作, 如生成对 $i \leq$ LIMIT 判断的代码并记下该代码的地址等等。因此, 在 do 处把 $S \rightarrow \text{for } i := E^{(1)} \text{ to } E^{(2)} \text{ do } S^{(1)}$ 划分为两个产生式:

$$F \rightarrow \text{for } i := E^{(1)} \text{ to } E^{(2)} \text{ do}$$
$$S \rightarrow F S^{(1)}$$

解答:

(1) 适合语法制导翻译的产生式:

$$F \rightarrow \text{for } i := E^{(1)} \text{ to } E^{(2)} \text{ do}$$
$$S \rightarrow F S^{(1)}$$

(2) 每个产生式对应的语义动作

$F \rightarrow \text{for } i := E^{(1)} \text{ to } E^{(2)} \text{ do}$

```
{  
    GEN(:=, .PLACE, -, ENTRY(i));  $E^{(1)}$   
    F.PLACE:=ENTRY(i);  
    LIMIT=NEWTEMP;  
    GEN(:=, .PLACE, -, LIMIT);  $E^{(2)}$   
    q:=NXQ;  
    F.QUAD:=q;    //again  
    GEN(j≤, F.PLACE, LIMIT, q+2);  
    F.CHAIN := q+1;  
    GEN(j, -, -, 0)  
}
```

$S \rightarrow F S^{(1)}$

```
{  
    BACKPATCH(.CHAIN, NXQ);  $S^{(1)}$   
    GEN(+, F.PLACE, 1, F.PALCE)  
    GEN(j, -, -, F.QUAD);  
    S.CHAIN:= F.CHAIN;  
}
```