同济大学
TONGJI UNIVERSITY

编译原理

第五章　语法分析——自下而上分析

丁志军

dingzj@tongji.edu.cn

概述

- 自下而上分析法就是从输入串开始，逐步进行“归约”，直至归约到文法的开始符号。
- 从语法树的末端，步步向上“归约”，直到根结。

自上而下分析法	开始符号S $\overset{*}{\Rightarrow}$ 输入串α （推导）
自下而上分析法	输入串α $\overset{*}{\Rightarrow}$ 开始符号S （归约）

内容线索

1. 自下而上分析基本问题
2. 算符优先分析方法
3. 规范归约
4. LR分析方法

示例

给定文法 G：

(1) S→aAcBe

(2) A→b

(3) A→Ab

(4) B→d

输入串 abbcde是否为句子?

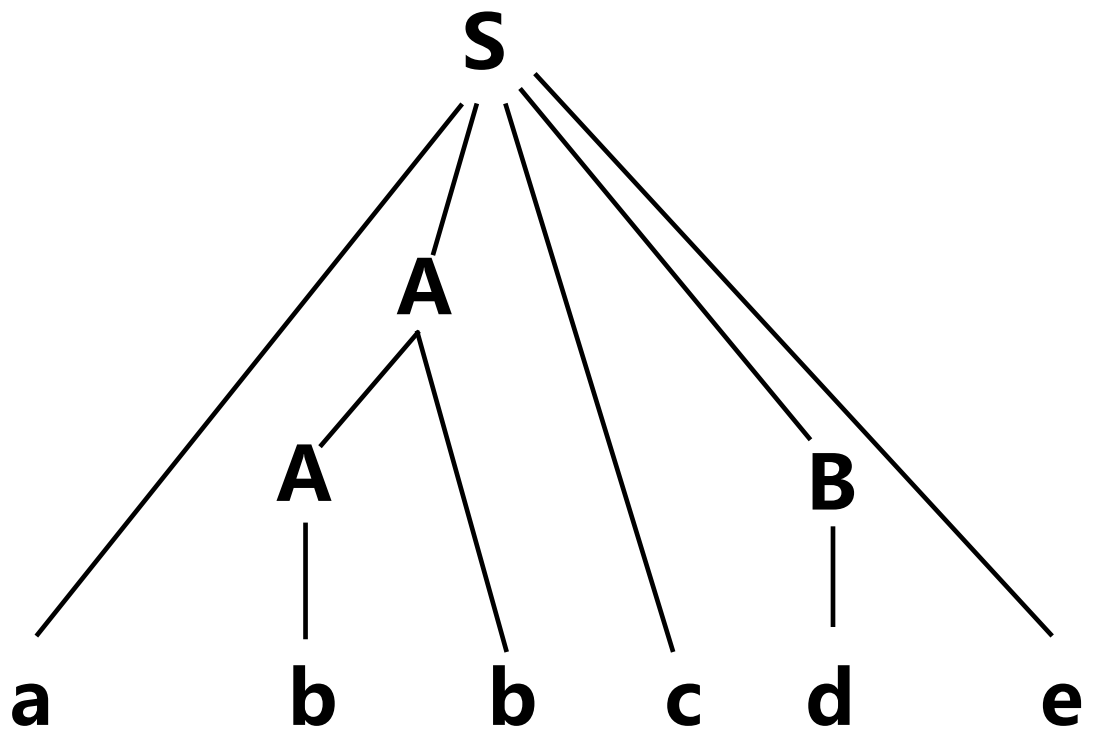
归约过程如下（分析树）：

S ⇒ aAcBe

⇒ aAcde

⇒ aAbcde

⇒ abbcde



归约

- 移进-归约法

  使用一个符号栈，把输入符号逐一移进栈，当栈顶形成某个产生式右部时，则将栈顶的这一部分替换（归约）为该产生式的左部符号。

示例

给定文法 G：

(1) S→aAcBe

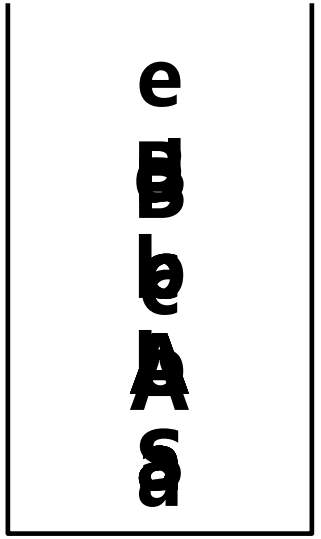
(2) A→b

(3) A→Ab

(4) B→d

输入串 abbcde是否为句子?

归约过程如下：



示例

给定文法 G：

(1) S→aAcBe

(2) A→b

(3) A→Ab

(4) B→d

输入串 abbcde是否为句子?

归约过程如下：

步骤：	1.	2.	3.	4.	5.	6.	7.	8.	9.	10.
动作：	进	进	归	进	归	进	进	归	进	归
	a	b	(2)	b	(3)	c	d	(4)	e	(1)

				b		c	d	B	e	
	b	A	A	A	A	A	c	c	B	S
a	a	a	a	a	a	a	A	A	c	
							a	a	A	
									a	

符号栈的使用

- 实现移进-归约分析的一个方便途径是用一个**栈**和一个**输入缓冲区**，用#表示栈底和输入的结束

	栈	输入串
初始	#	w#
最终	#S	#

示例

G: E→E+E | E*E | (E) | i 给出 i1*i2+i3 的移进归约过程

步骤	栈	输入串	动作
0	#	$i_1*i_2+i_3$#	预备
1	#i_1	$*i_2+i_3$#	移进
2	#E	$*i_2+i_3$#	归约E→i
3	#E*	i_2+i_3#	移进
4	#E*i_2	$+i_3$#	移进
5	#E*E	$+i_3$#	归约E→i
6	#E	$+i_3$#	归约E→E*E
7	#E+	i_3#	移进
8	#E+i_3	#	移进
9	#E+E	#	归约E→i
10	#E	#	归约E→E+E
11	#E	#	接受

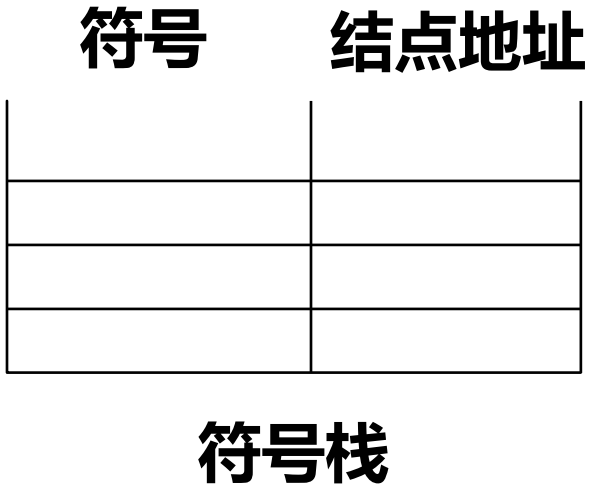
语法分析的操作

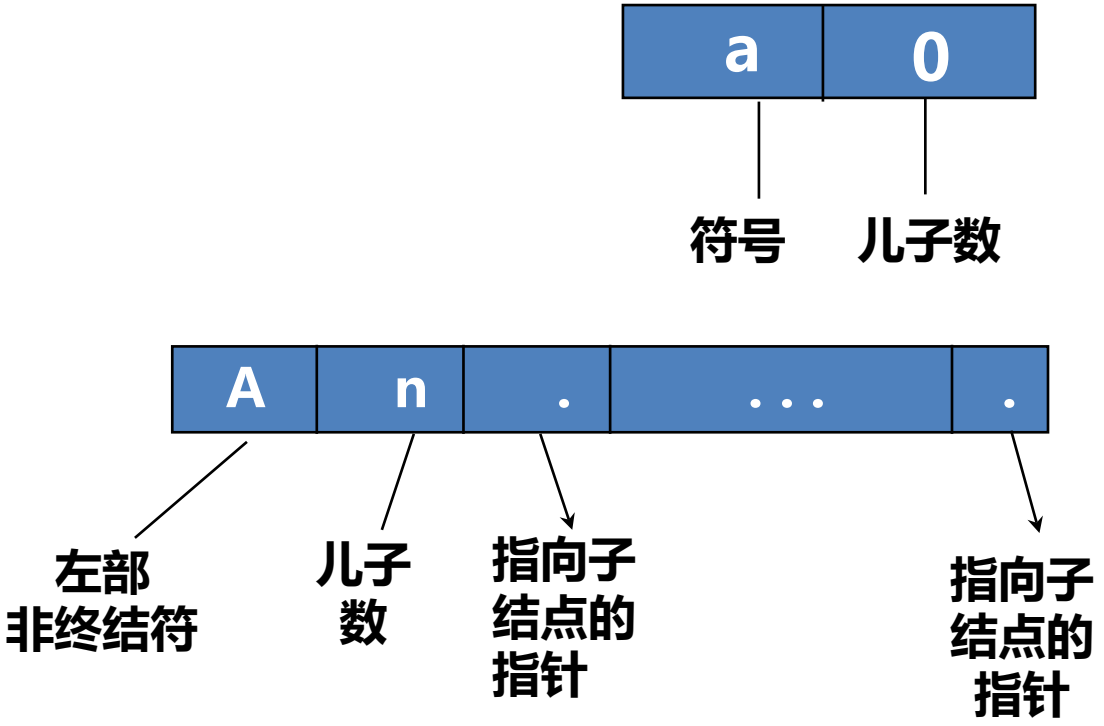
- **移进**
  - 下一输入符号移进栈顶,读头后移;
- **归约**
  - 检查栈顶若干个符号能否进行归约，若能，就以产生式左部替代该符号串，同时输出产生式编号;
- **接收**
  - 移进 - 归约的结局是栈内只剩下栈底符号和文法开始符号，读头也指向语句的结束符;
- **出错**
  - 发现了一个语法错,调用出错处理程序

注: 可归约的串在栈顶,不会在内部

语法树的表示——穿线表

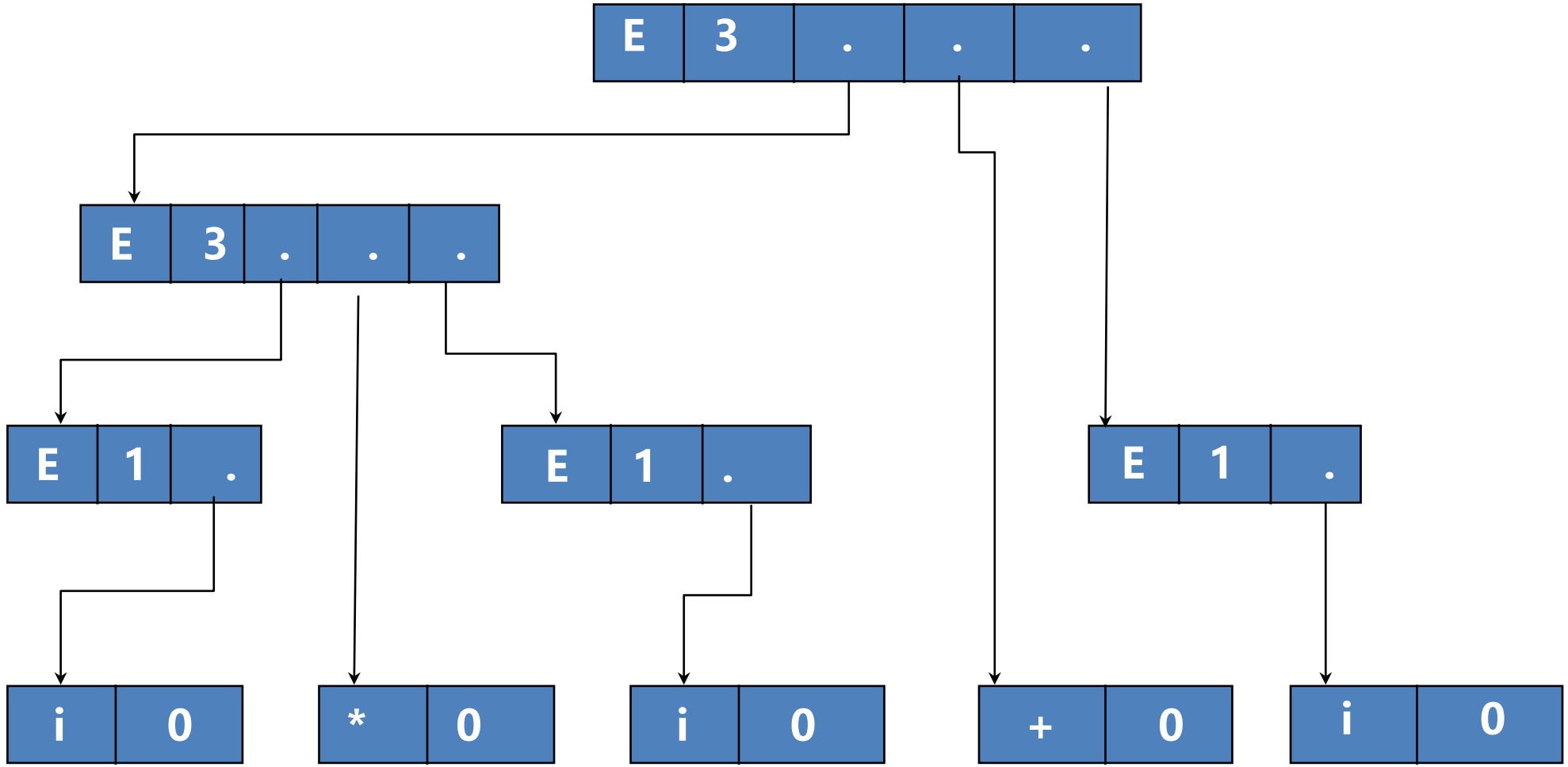
- 方法: 在移进-归约过程中自下而上构造句子的语法树
  - 移进符号a时，构造表示端末结a的数据结构，其地址与 a 同时进栈
  - 用 $A \rightarrow X_1X_2 \ldots X_n$ 归约时，构造新结A的数据结构，其地址与A同时进栈
  - 接受时，语法树已经构成，S 及根地址在栈中





示例

i * i + i 的语法树



给定文法 G：

(1) S→aA

(2) C→a|ab

(3) D→ab

(4) A→b

(5) B→b

输入串 ab是否为句子?

自下而上分析的基本问题

$$X \rightarrow \alpha b \beta$$

$$A \rightarrow \alpha$$

$$B \rightarrow \alpha$$

- 如何找出或确定可规约串?
- 对找出的可规约串替换为哪一个非终结符号?

内容线索

√. 自下而上分析基本问题

2. 算符优先分析方法

3. 规范归约

4. LR分析方法

算符优先分析方法

- 算符优先分析法是自下而上进行句型归约的一种分析方法。
- 定义**终结符**（算符）的优先关系，按终结符 （算符）的优先关系控制自下而上语法分析过程（寻找“**可归约串**”和进行归约）。
- 分析速度快，适于表达式的语法分析。

优先关系

- 任何两个可能**相继出现**的终结符a和b（它们之间可能插有一个非终结符）的**优先关系**：

$a \lessdot b$	a的优先级**低于**b
$a \doteq b$	a的优先级**等于**b
$a \gtrdot b$	a的优先级**高于**b

注：这三种关系不同于数学中的<,=,>关系。

算符文法

- 一个文法，如果它的任一产生式右部都**不含两个相继**（并列）的**非终结符**，即不含如下形式的产生式右部：

$$\ldots QR \ldots \quad , \quad Q, R \in V_N$$

则称该文法为**算符文法**。

算符优先关系

- 设 G为算符文法且不含ε- 产生式， a, b∈ V_T, 算符间的优先关系定义为：

a ≐ b	当且仅当G含有产生式 P →… ab… 或 P →… aQb…
a ⋖ b	当且仅当G含有产生式 P →… aR … 且 $R \overset{+}{\Rightarrow} b\ldots$ 或 $R \overset{+}{\Rightarrow} Qb\ldots$
a ⋗ b	当且仅当G含有产生式 P →… Rb… 且 $R \overset{+}{\Rightarrow} \ldots a$ 或$R \overset{+}{\Rightarrow} \ldots aQ$

算符优先文法

- 如果一个算符文法G中的任何终结符对(a, b)**至多**满足下述关系之一

$a \doteq b$　　$a \lessdot b$　　$a \gtrdot b$

则称 G 为**算符优先文法**。

示例

给定文法G：E→E+E|E*E|(E)|i　其中：V_T={+, *, i, (,)} 。

G是算符文法

G是算符优先文法吗?

考察终结符对(+, *)

(1) 因为E→E+E，且E ⇒ E*E，所以

$$+ \lessdot *$$

(2) 因为E→E*E，且E ⇒ E+E，所以

$$+ \gtrdot *$$

G不是算符优先文法

示例

文法G: (1) E→E+T | T　　(2) T→T*F | F

(3) F→P↑F | P　　(4) P→(E) | i

算符优先关系为：

由(4): P→(E)　　∴ (≐)

由(1)(2): E→E+T, T ⇒ T*F　　∴ + ⋖ *

由(2) (3): T→T*F, F ⇒ P↑F　　∴ * ⋖ ↑

由(1): E→E+T, E ⇒ E+T　　∴ + ⋗ +

由(3): F→P↑F, F ⇒ P↑F　　∴ ↑ ⋖ ↑

由(4): P→(E), E ⇒ E+T　　∴ (⋖ +, + ⋗)

. . .

∴ G为算符优先文法

（ #看作终结符号，作为句子括号）

优先关系表的构造

- 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 ≐ b的终结符对。

a ≐ b 当且仅当G含有产生式 P →… ab… 或 P →… aQb…

- 确定满足关系 <· 和 ·> 的所有终结符对：

a <· b 当且仅当G含有产生式 P →… aR … 且 $R \overset{+}{\Rightarrow} b\ldots$ 或 $R \overset{+}{\Rightarrow} Qb\ldots$

a ·> b 当且仅当G含有产生式 P →… Rb… 且 $R \overset{+}{\Rightarrow} \ldots a$ 或 $R \overset{+}{\Rightarrow} \ldots aQ$

FIRSTVT(P) 和 LASTVT(P)

设 $P \in V_N$，定义：

FIRSTVT (P) =

$\{ a \mid P \overset{+}{\Rightarrow} a \ldots$ 或 $P \overset{+}{\Rightarrow} Qa \ldots , a \in V_T , Q \in V_N \}$

LASTVT (P) =

$\{ a \mid P \overset{+}{\Rightarrow} \ldots a$ 或 $P \overset{+}{\Rightarrow} \ldots aQ , a \in V_T , Q \in V_N \}$

FIRSTVT(P) 和 LASTVT(P)构造

- **FIRSTVT (P) 构造**
  - **规则1: 若 P → a … 或 P → Qa … , 则 a ∈FIRSTVT(P);**
  - **规则2: 若 a ∈FIRSTVT(Q) , 且 P → Q … , 则 a ∈FIRSTVT(P)。**

- **LASTVT (P) 构造**
  - **规则1: 若 P→ … a 或 P→ … aQ , 则 a ∈ LASTVT(P);**
  - **规则2: 若 a ∈ LASTVT(Q) , 且 P→ … Q , 则 a ∈ LASTVT(P)。**

FIRSTVT(P) 的构造——数据结构

- 二维布尔矩阵 F[P,a] 和符号栈STACK

$$\text{布尔矩阵}F[P,a] = \begin{cases} .T. & a \in FIRSTVT(P) \\ .F. & a \notin FIRSTVT(P) \end{cases}$$

栈 STACK：存放使FIRSTVT 为真的符号对 (P, a).

FIRSTVT(P)的构造——算法

- 把所有初值为真的数组元素 F[P, a] 的符号对 (P, a) 全都放在STACK之中。
- 如果栈STACK不空，就将栈顶逐出，记此项为(Q, a)。对于每个形如P→Q…　的产生式，若F[P, a]为假，则变其值为真，且将(P, a)推进STACK栈。
- 上述过程必须一直重复，直至栈STACK拆空为止。

示例

G: S→ a | ^ | (T)

T→T,S | S

求 FIRSTVT(S), FIRSTVT(T)

$$M=\begin{array}{c} \\ S \\ \\ T \end{array}\begin{array}{c} \begin{array}{ccccc} a & \wedge & (&) & , \end{array} \\ \begin{pmatrix} T & T & T & F & F \\ \\ T & T & T & F & T \end{pmatrix} \end{array}$$

T	,
S	(
S	^
S	a

FIRSTVT(S) ={ a, ^, (}

FIRSTVT(T) ={ a, ^, (, ,}

类似地可得: LASTVT(S) ={a, ^,) }

LASTVT(T) ={a, ^,), , }

FIRSTVT主程序:

```
BEGIN
   FOR  每个非终结符P和终结符a  DO
            F[P,a] := FALSE;
   FOR  每个形如P→a… 或P→Qa … 的产生式  DO
            INSERT (P, a);
   WHILE  STACK非空  DO
   BEGIN
            把STACK 的顶项 (Q, a) 弹出;
            FOR  每条形如P→Q … 的产生式  DO
                 INSERT (P, a);
    END OF WHILE;
END
```

```
PROCEDURE   INSERT(P,a);
     IF  NOT F[P,a]  THEN
     BEGIN
       F[P,a]:= true;
       把 (P, a)下推进STACK栈
     END;
```

优先关系表的构造

- 有了这两个集合之后，就可以通过检查每个产生式的候选式确定满足关系 $\lessdot$ 和 $\gtrdot$ 的所有终结符对。
  - 假定有个产生式的一个候选形为

    ...aP...

    那么，对任何 b∈FIRSTVT(P)，有 a $\lessdot$ b。

  - 假定有个产生式的一个候选形为

    ...Pb...

    那么，对任何 a∈LASTVT(P)，有 a $\gtrdot$ b。

构造优先关系表算法

FOR 每条产生式$P \to X_1X_2...X_n$ DO

FOR i:=1 TO n-1 DO

BEGIN

IF X_i和X_{i+1}均为终结符 THEN 置$X_i = X_{i+1}$

IF $i \le n-2$且X_i和X_{i+2}都为终结符

但X_{i+1}为非终结符 THEN 置$X_i = X_{i+2}$;

IF X_i为终结符而X_{i+1}为非终结符 THEN

FOR FIRSTVT(X_{i+1})中的每个a DO

置 $X_i \lessdot a$;

IF X_i为非终结符而X_{i+1}为终结符 THEN

FOR LASTVT(X_i)中的每个a DO

置 $a \gtrdot X_{i+1}$

END

示例

G: S→a | ^ | (T)　　T→T,S | S

FIRSTVT(S) = { a, ^, (}　　FIRSTVT(T) = { a, ^, (, , }

LASTVT(S) = { a, ^,) }　　LASTVT(T) = { a, ^,), , }

优先关系	a	^	(	)	,
a					
^					
(					
)					
,					

约定任何终结符号有：a>#，#<a

短语

- **短语**

令G是一个文法，S是文法的开始符号，若αβδ是文法G的一个句型，如果有

$$S \overset{*}{\Rightarrow} \alpha A \delta \quad 且\ A \overset{+}{\Rightarrow} \beta$$

则称β是句型αβδ相对于非终结符A的短语。

示例

设文法G（S）：

(1) S → aAcBe

(2) A → b

(3) A → Ab

(4) B → d

给出句型aAbcde的短语。

由 S ⇒ aAcBe ⇒ aAcde ⇒ aAbcde

S ⇒ aAcBe ⇒ aAbcBe ⇒ aAbcde

短语：d，Ab, aAbcde

句型语法树和句型的短语

- **短语**
  - 句型语法树中**每棵子树**（某个结点连同它的所有子孙组成的树）的**所有叶子结点从左到右排列起来**形成一个相对于子树根的短语。

示例

设文法G（S）：　(1) S → aAcBe

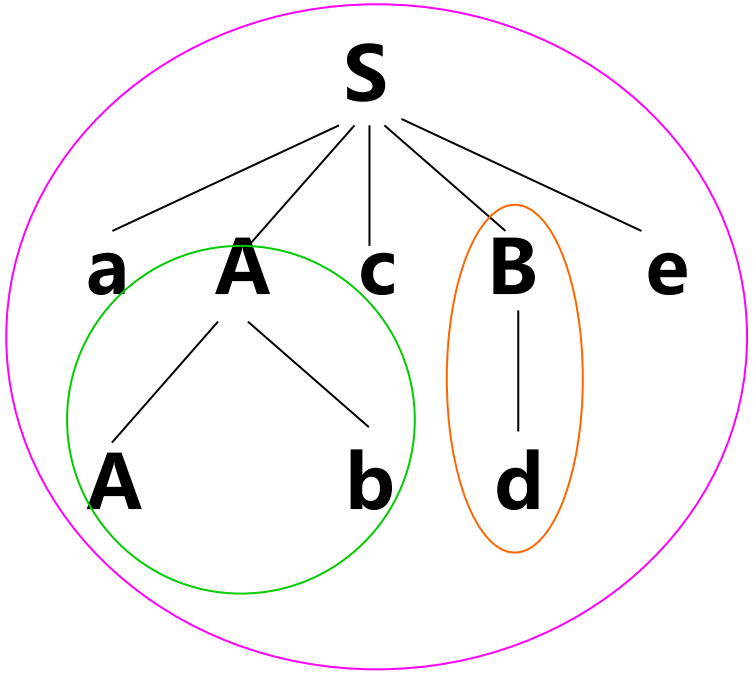
(2) A → b

(3) A → Ab

(4) B → d

给出句型aAbcde的短语。

句型aAbcde的语法树为：



短语：d，Ab, aAbcde

最左素短语

- **短语**

  令G是一个文法，S是文法的开始符号，若αβδ是文法G的一个句型，如果有

  $S \overset{*}{\Rightarrow} \alpha A \delta$ 且 $A \overset{+}{\Rightarrow} \beta$

  则称β是句型αβδ相对于非终结符A的短语。

- **素短语**

  - 是一个短语，它至少含有一个终结符且除它自身之外不含有任何更小的素短语。

- **最左素短语**

  - 处于句型最左边的那个素短语。

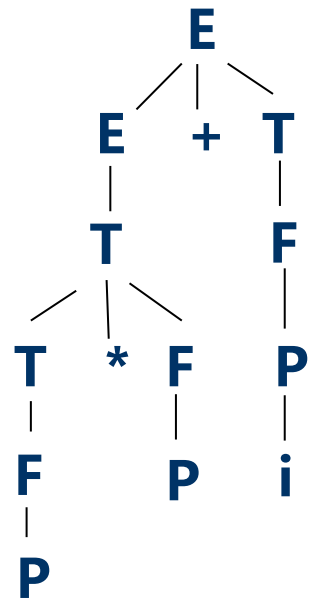
示例

对文法G:

(1) E→E+T | T
(2) T→T*F | F
(3) F→P↑F | P
(4) P→(E) | i

求句型P*P+i的短语、素短语、最左素短语

解: 句型的语法树为:



句型的短语:P, P*P, i, P*P+i

素短语:P*P, i

最左素短语:P*P

示例

对文法G:

(1) E→E+T | T

(2) T→T*F | F

(3) F→P↑F | P

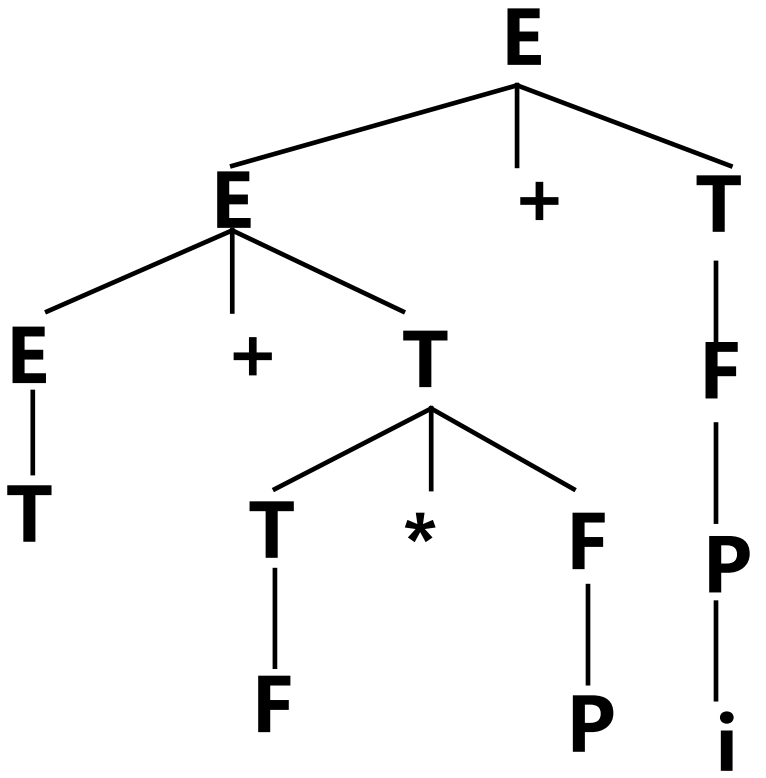
(4) P→(E) | i

句型： T+F*P+i

短语： T, F, P, i, F*P,
T+F*P, T+F*P+i

素短语： F*P , i

最左素短语： F*P



算符优先文法的最左素短语

- **算符优先文法句型**(括在两个 # 之间)的一般形式为:

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

  其中: $a_i \in V_T$, $N_i \in V_N$ (可有可无)

- 一个算符优先文法G的任何句型的**最左素短语**是满足下列条件的最左子串 $N_ja_j \ldots N_ia_i N_{i+1}$

  $a_{j-1} \lessdot a_j$

  $a_j \doteq a_{j+1} \doteq \ldots \doteq a_{i-1} \doteq a_i$

  $a_i \gtrdot a_{i+1}$

例. 句型#P*P+i#中，# $\lessdot$ * , * $\gtrdot$ +，所以P*P是最左素短语

最左素短语

- 一个算符优先文法G的任何句型的**最左素短语**是满足下列条件的最左子串 $N_j a_j \ldots N_i a_i N_{i+1}$

$$a_{j-1} \lessdot a_j$$

$$a_j \doteq a_{j+1} \doteq \ldots \doteq a_{i-1} \doteq a_i$$

$$a_i \gtrdot a_{i+1}$$

$$\#\, N_1 a_1 \ldots a_{j-1} \overbrace{N_j a_j \ldots N_i a_i N_{i+1}}^{R} a_{i+1} \ldots N_n a_n N_{n+1} \,\#$$

$$< \qquad = \qquad >$$

示例 对文法G: (1) E→E+T | T (2) T→T*F | F (3) F→P↑F | P (4) P→(E) | i

FIRSTVT

$$M=\begin{array}{c} \\ E \\ T \\ F \\ P \end{array}\begin{array}{c} \begin{array}{cccccc} + & * & \uparrow & (&) & i \end{array} \\ \begin{pmatrix} T & T & T & T & F & T \\ F & T & T & T & F & T \\ F & F & T & T & F & T \\ F & F & F & T & F & T \end{pmatrix} \end{array}$$

LASTVT

$$M=\begin{array}{c} \\ E \\ T \\ F \\ P \end{array}\begin{array}{c} \begin{array}{cccccc} + & * & \uparrow & (&) & i \end{array} \\ \begin{pmatrix} T & T & T & F & T & T \\ F & T & T & F & T & T \\ F & F & T & F & T & T \\ F & F & F & F & T & T \end{pmatrix} \end{array}$$

	+	*	↑	(	)	i
+	>	<	<	<	>	<
*	>	>	<	<	>	<
↑	>	>	<	<	>	<
(	<	<	<	<	=	<
)	>	>	>		>	
i	>	>	>		>	

示例

对文法G: (1) E→E+T | T (2) T→T*F | F

(3) F→P↑F | P (4) P→(E) | i

句型：T+F*P+i

最左素短语：F*P

\+ < * > +

	+	*	↑	(	)	i
+	>	<	<	<	>	<
*	>	>	<	<	>	<
↑	>	>	<	<	>	<
(	<	<	<	<	=	<
)	>	>	>		>	
i	>	>	>		>	

算符优先分析算法

1. 将输入串依此逐个存入符号栈S中，直到符号栈顶元素S_k与下一个待输入的符号a有优先关系$S_k > a$为止；

2. 至此，最左素短语尾符号S_k已在符号栈S的栈顶，由此往前在栈中找最左素短语的头符号S_{j+1}，直到找到第一个 < 为止；

3. 已找到最左素短语$S_{j+1}\ldots S_k$，将其归约为某个非终结符N及做相应的语义处理。

主控程序：　设 k 为符号栈S 的指针

```
1        k =1;  S[k]:=  “  # ” ;
2        REPEAT
3           把下一个输入字符读进a中;
4         IF  S[k]∈V_T  THEN  j:= k  ELSE  j:= k -1;
5           WHILE  S[ j ] > a   DO
6           BEGIN
7              REPEAT
8                    Q:= S[ j];
9                    IF  S[ j -1]∈V_T   THEN  j:= j -1   ELSE  j:= j -2
10             UNTIL   S[ j] < Q;
11             把S[ j +1] …S[k] 归约为某个N;
12             k := j+1;       S[k]:= N
13          END OF  WHILE;
14          IF  S[ j] < a  OR  S[ j] = a  THEN
15              BEGIN k := k+1;  S[k]:= a  END
16          ELSE  ERROR
17       UNTIL  a =  “#”
```

自左至右，终结符对终结符，非终结符对非终结符，而且**对应的终结符相同**。

$$N \rightarrow \underset{S[j+1]}{X_1} \quad \underset{S[j+2]}{X_2} \quad \cdots \quad \underset{S[k]}{X_{k-j}}$$

x

示例

对文法G: (1) E→E+T | T (2) T→T*F | F

(3) F→P↑F | P (4) P→(E) | i

句子：i*(i+i)

	+	*	↑	(	)	i
+	>	<	<	<	>	<
*	>	>	<	<	>	<
↑	>	>	<	<	>	<
(	<	<	<	<	=	<
)	>	>	>		>	
i	>	>	>		>	

对文法G，符号串i*(i+i)的分析过程如下：

符号栈	关系	输入串	最左素短语
#	<.	i* (i+i) #	
#i	>.	* (i+i) #	i
#N	<.	* (i+i) #	
#N*	<.	(i+i) #	
#N*(	<.	i+i) #	
#N*(i	>.	+i) #	i
#N*(N	<.	+ i) #	
#N*(N+	<.	i) #	
#N*(N+i	>.	) #	i
#N*(N+N	>.	) #	N+N
#N*(N	=.	) #	
#N*(N)	>.	#	(N)
#N*N	>.	#	N*N
#N	=.	#	
#N#		成功	

说明

- 约定任何终结符号有：a>#，#<a
- 在算法的工作过程中，若出现j减1后的值小于等于0时，则意味着输入串有错。在正确的情况下，算法工作完毕时，符号栈S应呈现：

  # N #

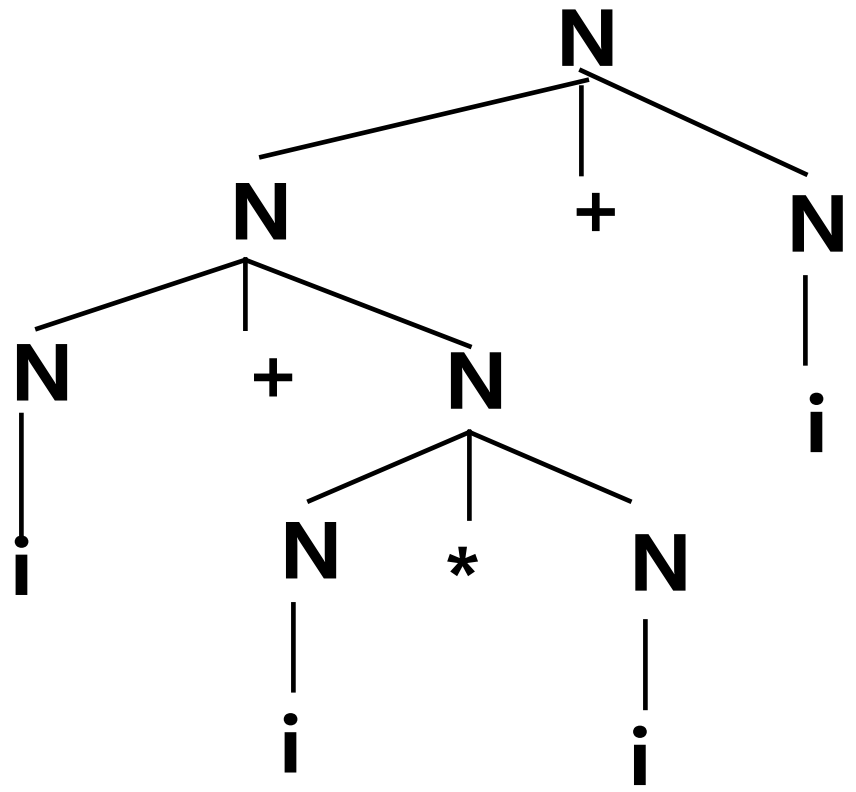
- 由于非终结符对归约没有影响，因此，非终结符可以不进符号栈S。

示例

对文法G: (1) E→E+T | T (2) T→T*F | F

(3) F→P↑F | P (4) P→(E) | i

给出句子i+i*i+i的算符优先分析的语法树



算符优先分析归约速度快，

但容易误判

内容线索

√. 自下而上分析基本问题

√. 算符优先分析方法

3. 规范归约

4. LR分析方法

短语

- 令G是一个文法，S是文法的开始符号，若αβδ是文法G的一个句型，

  如果有 $S \overset{*}{\Rightarrow} \alpha A \delta$ 且 $A \overset{+}{\Rightarrow} \beta$

  则称β是句型αβδ相对于非终结符A的**短语**。

  - 特别地，若 $A \Rightarrow \beta$，则称 β 是句型 αβδ 关于产生式A→β的**直接短语**。

- 一个句型的最左直接短语称为**句柄**。

示例

设文法G（S）：

(1) S → aAcBe

(2) A → b

(3) A → Ab

(4) B → d

给出句型aAbcde的短语、直接短语、句柄。

由 S ⇒ aAcBe ⇒ aAcde ⇒ aAbcde

S ⇒ aAcBe ⇒ aAbcBe ⇒ aAbcde

短语：d，Ab, aAbcde

直接短语：d，Ab

句柄： Ab

句型语法树和句型的短语、直接短语、句柄

- 短语：句型语法树中**每棵子树**（某个结点连同它的所有子孙组成的树）的**所有叶子结点从左到右排列起来**形成一个相对于子树根的短语。
- 直接短语：只有**父子两代的子树**形成的短语。
- 句柄：语法树中**最左**那棵**只有父子两代**的子树形成的短语。

示例

G（E）：E→E+T|E-T|T

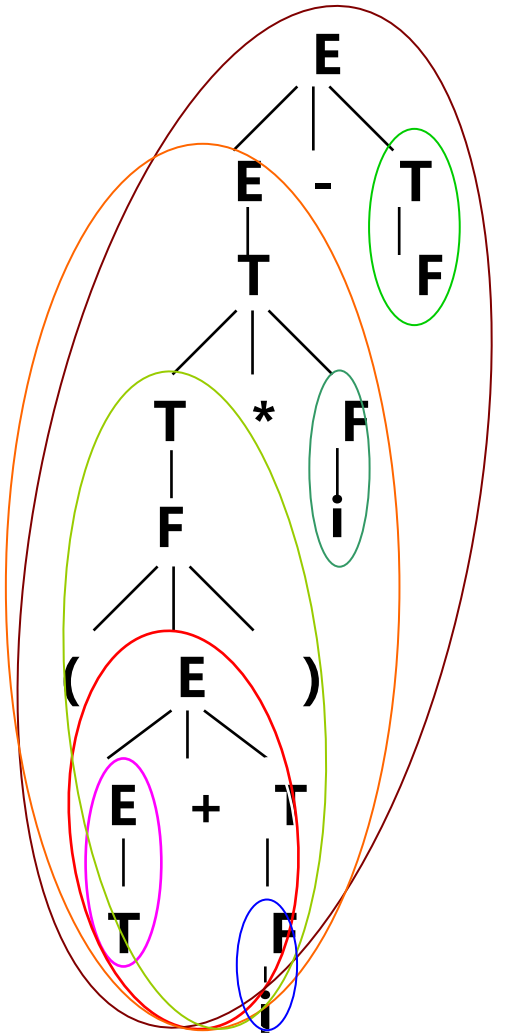
T→T*F|T/F|F

F→ i |(E)

试找出句型 (T+ i) * i-F 的所有短语、直接短语和句柄

解：

短语	直接短语	句柄
(T+ i) * i -F	T	T
(T+ i) * i	i	
(T+ i)	i	
T+ i	F	
T		
i (左)		
i (右)		
F		



示例

对文法G: (1) E→E+T | T　　(2) T→T*F | F

(3) F→P↑F | P　　(4) P→(E) | i

句型：T+F*P+i

短语：　T，F，P，i，F*P，

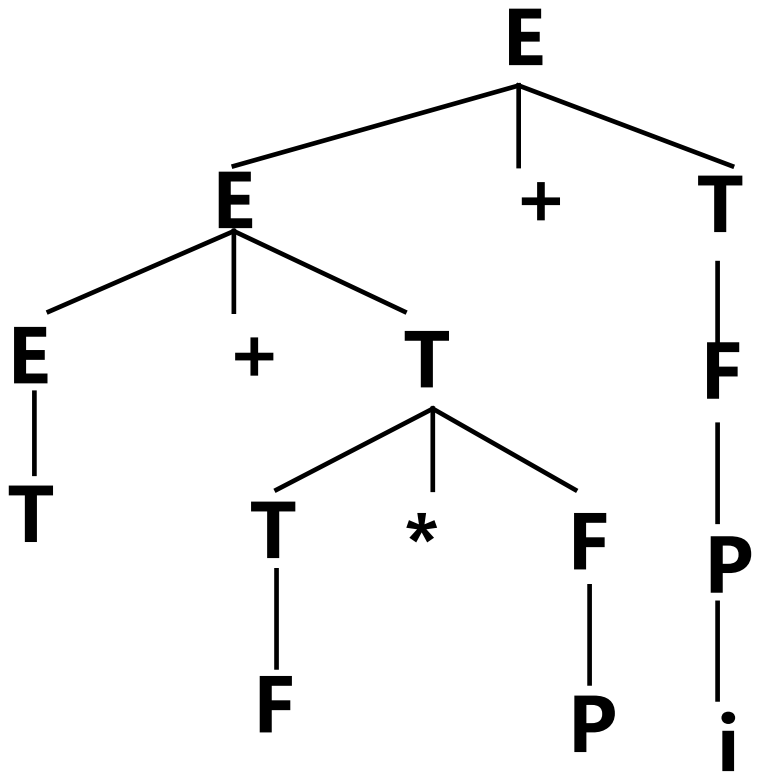
T+F*P，T+F*P+i

直接短语：　T，F，P，i

句柄：T

素短语：F*P，i

最左素短语：　F*P



示例

给定文法G：E→E+E|E*E|(E)|i

给出句型E+E*E的句柄

解.（1）E ⇒ E+E ⇒ E+E*E

E*E是句柄

（2）E ⇒ E*E ⇒ E+E*E

E+E是句柄

注: 二义性文法的句柄可能不唯一

规范归约

设α是文法G的一个句子，若序列α_n, α_{n-1}, …, α_0，满足：

（1）$\alpha_n = \alpha$;

（2）$\alpha_0 = S$;

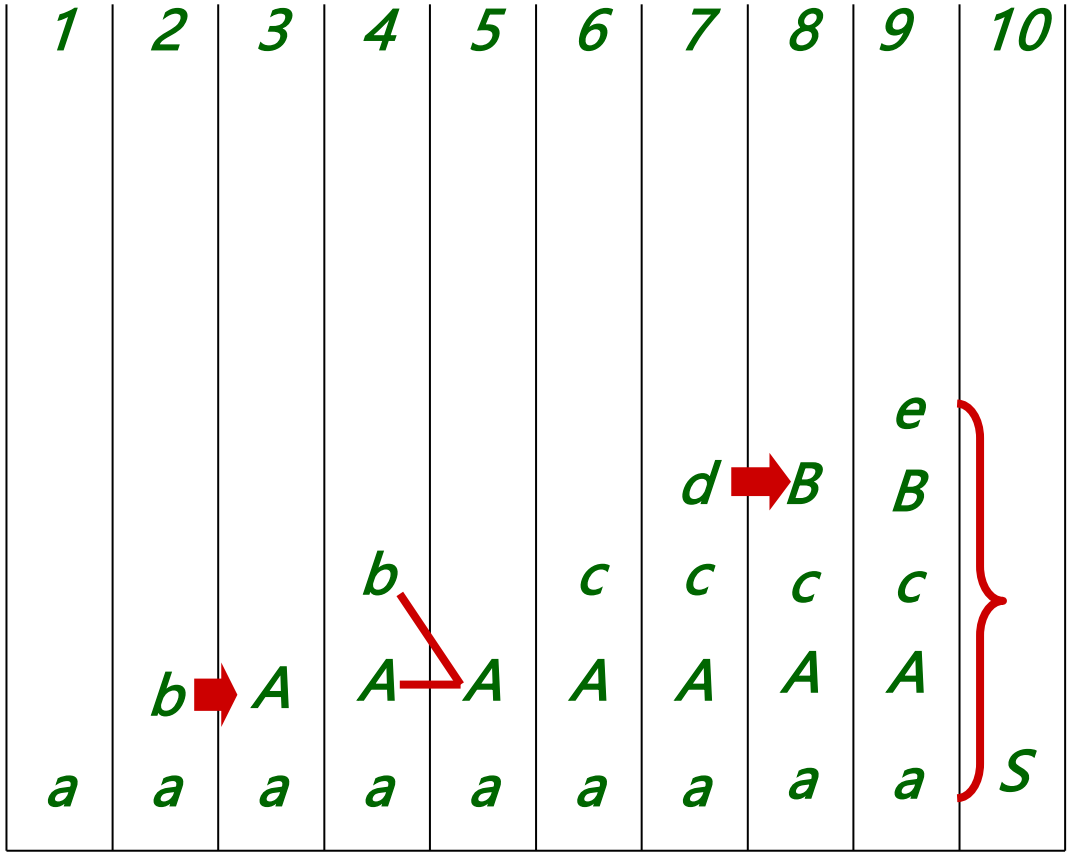
（3）对任意i，$0 < i \leq n$，α_{i-1}是从α_i将**句柄**替换成相应产生式左部符号而得到的

则称该序列是一个**规范归约**。

规范归约是关于α的一个最右推导的逆过程

最右推导： S => aAcBe => aAcde =>aAbcde => a b b c d e

栈

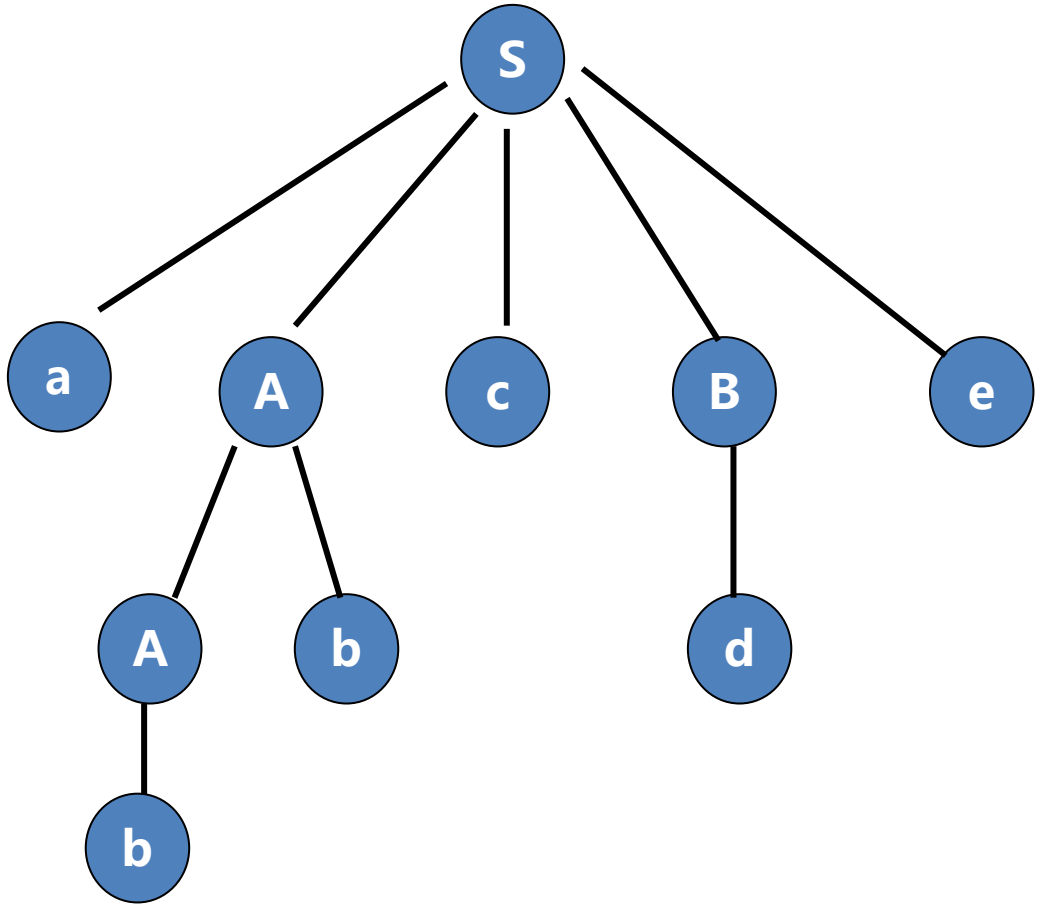


S->aAcBe A->Ab

A->b B->d

输入串：abbcde

最左归约： *a b b c d e => aAbcde => aAcde => aAcBe => S*



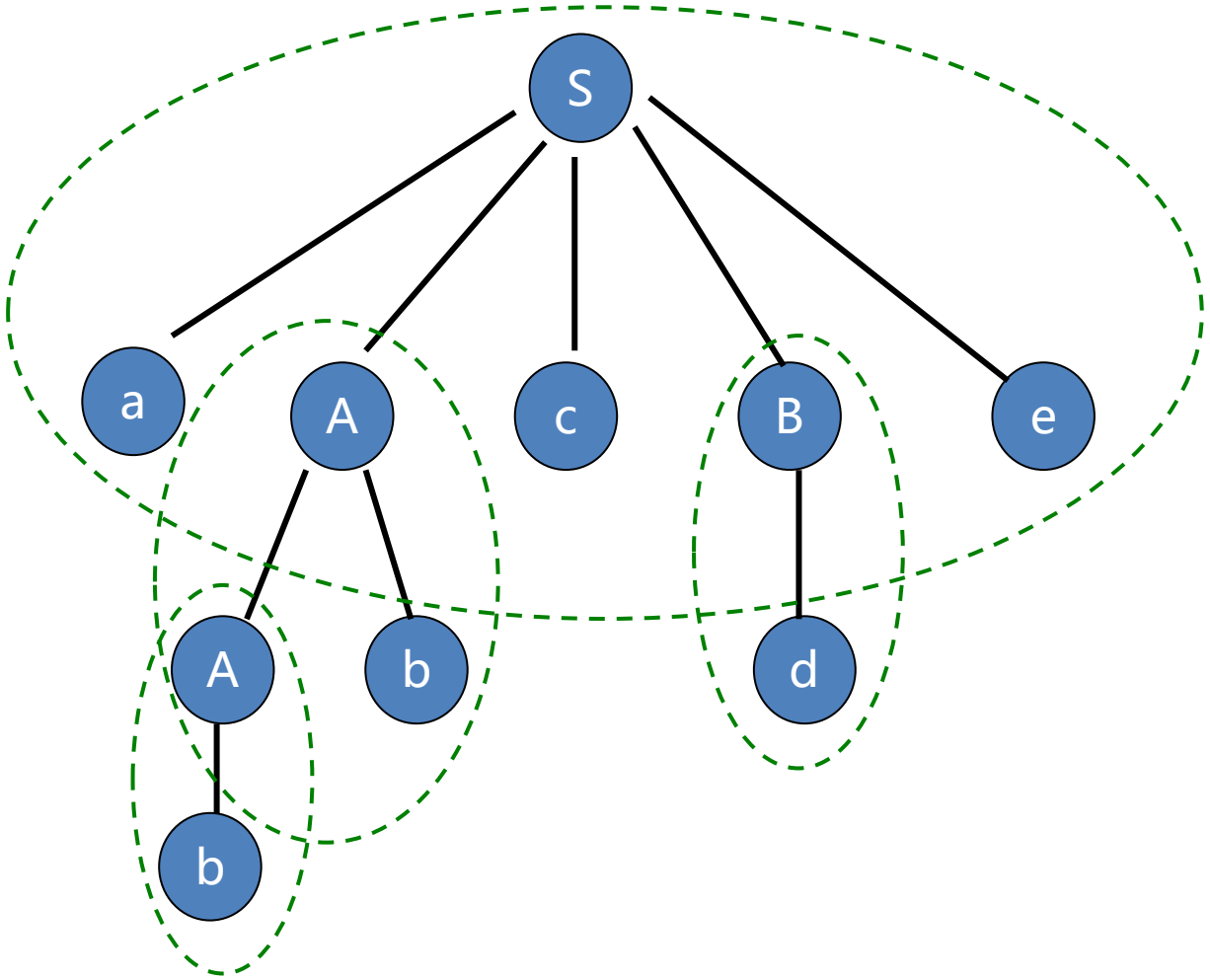
分 析 树

S->aAcBe **A->Ab**

A->b **B->d**

输入串：abbcde

句子abbcde 的规范归约过程如下: —— “剪枝”



文法G（S）：

S→aAcBe
A→b
A→Ab
B→ d

规范归约	归约规则
abbcde	A→b
aAbcde	A→Ab
aAcde	B→d
aAcBe	S→aAcBe
S	

示例

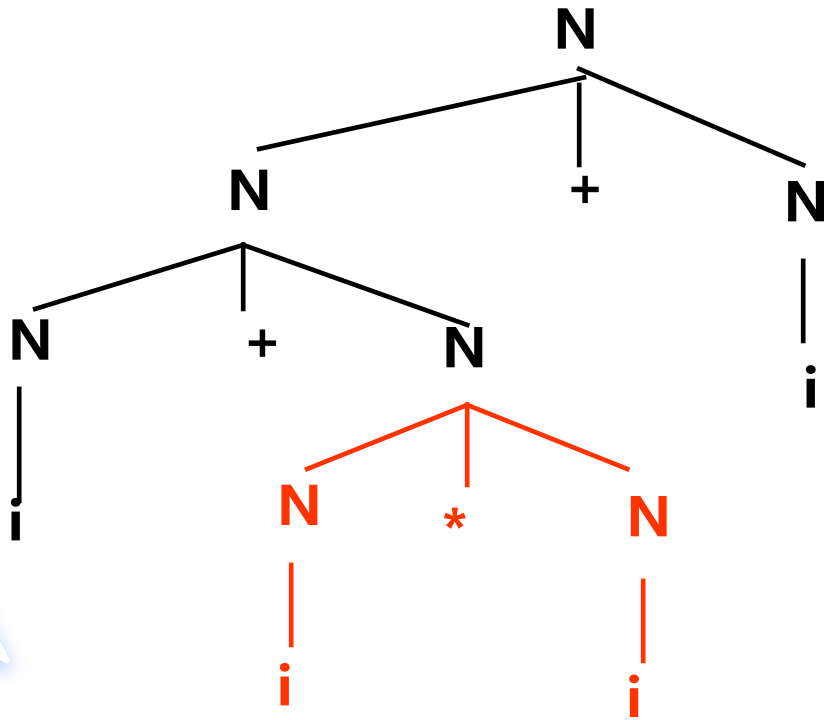
对文法G: (1) E→E+T | T (2) T→T*F | F

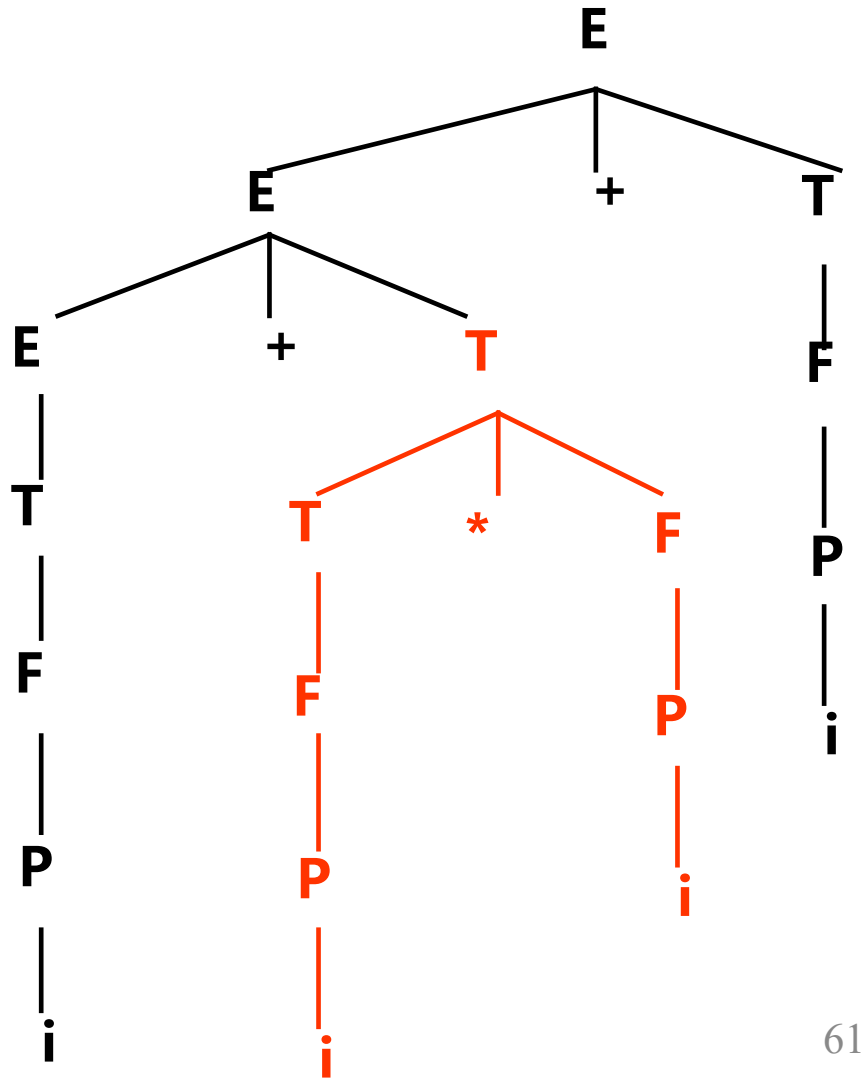
(3) F→P↑F | P (4) P→(E) | i

分别给出句子i+i*i+i的

算符优先分析和规范归约分析的语法树

算符优先分析相比规范归约,其归约速度快，但容易误判。





规范归约的基本问题

- 如何找出或确定可归约串——**句柄**?
- 对找出的可归约串——**句柄**替换为哪一个非终结符号?

内容线索

√. 自下而上分析基本问题

√. 算符优先分析方法

√. 规范归约

4. LR分析方法

LR分析法

- 在自下而上的语法分析中，算符优先分析算法只适用于算符优先文法，还有很大一类上下文无关文法可以用LR分析法分析。
- LR分析法中的L表示从左向右扫描输入串， R表示构造最右推导的逆。LR分析法是严格的规范归约。
- 不足：LR分析法手工构造分析程序工作量相当大。
  - YACC是一个语法分析程序的自动生成器。

LR分析法

- LR分析法：1965年 由Knuth提出

产生分析表

LR分析器工作

人物简介

- Don E. Knuth高德纳(1938-)
  - 编译程序： LR(k)
  - 属性文法
  - 算法: KMP算法
  - 数字化排版TeX
- 1974年图灵奖获得者
- 《计算机程序设计艺术》作者

LR分析法

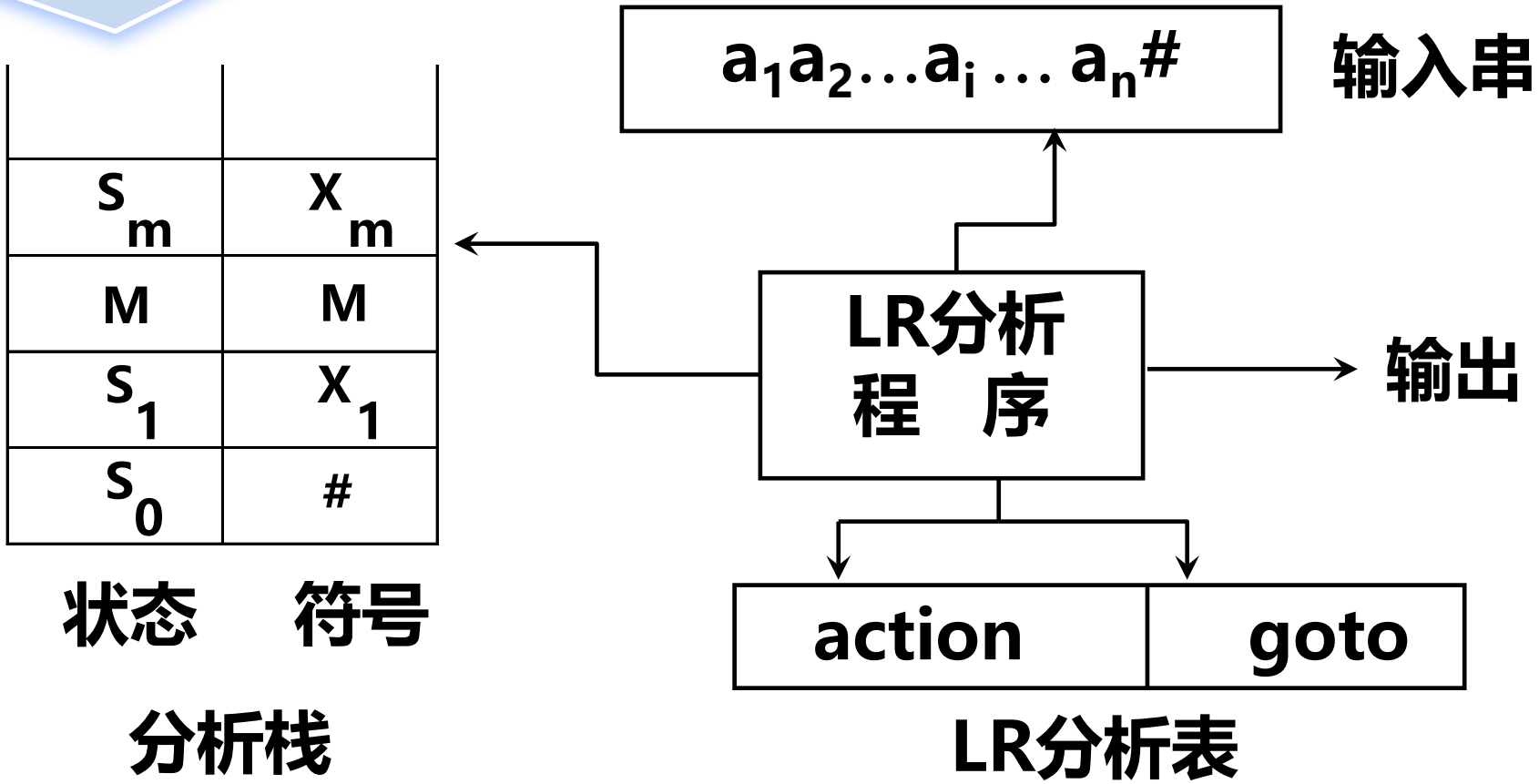
- 总控程序: 所有的LR分析器相同
- 分析表: 是自动生成语法分析器的关键
  - LR (0) 表: 基础、有局限性
  - SLR表: 简单LR表，实用
  - 规范LR表: 能力强、代价大
  - LALR表：向前LR表，介于SLR和规范LR之间

LR分析法原理

- 在移进 - 归约过程中寻找句柄
  - 历史: 在分析栈中已移进和归约的符号串
  - 展望: 根据当前使用的产生式推测未来可能遇到的输入符号
  - 现实: 当前输入符号

LR分析器模型

把“历史”及“展望”综合抽象成状态；
由栈顶的状态和现行的输入符号唯一确定每一步工作



活前缀

- 前缀: 一个字的任意首部。例:字abc的前缀有

  ε,a,ab,或abc.

- **活前缀**: 规范句型的一个前缀，前缀的尾符号最多包含到句型的句柄，即这种前缀不含句柄之后的任何符号（可归前缀）。
  - 对于规范句型$\alpha\beta\delta$，β为句柄，如果$\alpha\beta=u_1u_2...u_r$，则符号串$u_1u_2...u_i(1\le i\le r)$是$\alpha\beta\delta$的活前缀。($\delta$必为终结符串)

活前缀

- 在LR分析工作过程的任何时候，栈里的文法符号（自栈底向上）应该构成活前缀。
- 对于一个文法G，可以构造一个识别G的所有活前缀有限自动机，并以此构造LR分析表。

LR(0)项目

- 文法G 的产生式右部加一个圆点(·)，称为G的一个LR(0)项目。
  它指明了在分析过程的某时刻看到产生式的多大部分。

例. G(E):　E→ aA

A→ bA|a

则G的LR（0）项目有:

E → ·aA　　E→ a·A　　E→ aA·

A → ·bA　　A→ b·A　　A→ bA·

A → ·a　　A →a·

拓广文法

- 假定文法G是一个以S为开始符号的文法，我们构造一个G′
  - 包含整个G;
  - 引进了一个不出现在G中的非终结符S′ （ G′ 的开始符号）；
  - 增加一个新产生式S′ →S。
- 称G′ 是G的**拓广文法**。
- 拓广文法会有一个仅含项目S′ →S·的状态，这就是唯一的“接受”态。

示例

文法G(S′)

S′→E

E→aA|bB

A→cA|d

B→cB|d

该文法的项目有：

1. S′→•E
2. S′→E•
3. E→•aA
4. E→a•A
5. E→aA•
6. A→•cA
7. A→c•A
8. A→cA•
9. A→•d
10. A→d•
11. E→•bB
12. E→b•B
13. E→bB•
14. B→•cB
15. B→c•B
16. B→cB•
17. B→•d
18. B→d•

方法一：识别活前缀的NFA方法

- 构造识别文法所有活前缀的NFA

  项目1为NFA的唯一初态，任何状态（项目）均认为是 NFA的终态（活前缀识别态）

  1. 若状态i为$X \to X_1 \cdots X_{i-1} \cdot X_i \cdots X_n$，

     状态j为$X \to X_1 \cdots X_{i-1}X_i \cdot X_{i+1} \cdots X_n$，

     则从状态i画一条标志为X_i的有向边到状态j;

  2. 若状态i为$X \to \alpha \cdot A\beta$，A为非终结符，

     则从状态i画一条ε边到所有状态$A \to \cdot \gamma$。

- 把识别文法所有活前缀的NFA确定化。

1. S′→•E　2. S′→E•

3. E→•aA　4. E→a•A

5. E→aA•　6. A→•cA

7. A→c•A　8. A→cA•

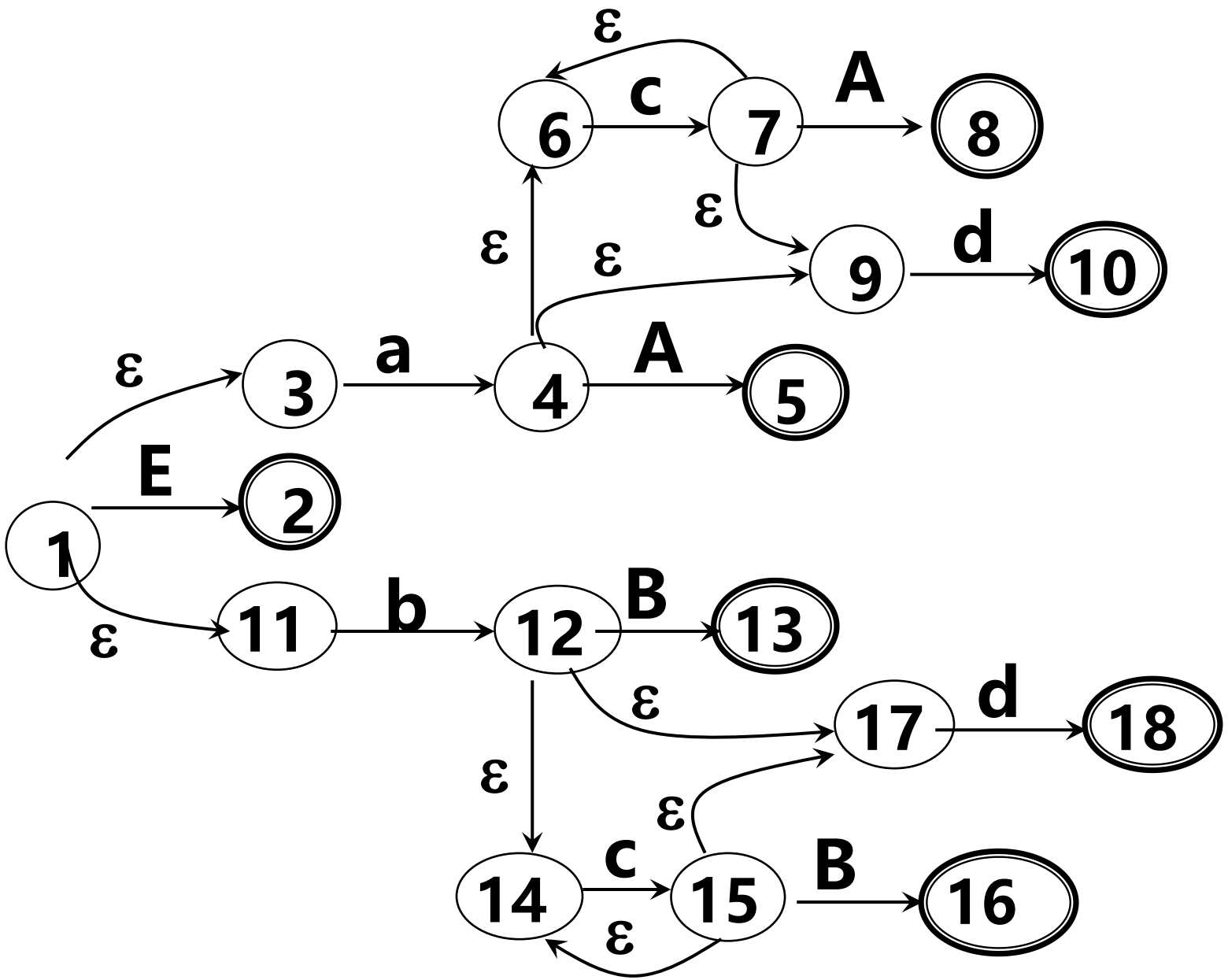
9. A→•d　10. A→d•

11. E→•bB　12. E→b•B

13. E→bB•　14. B→•cB

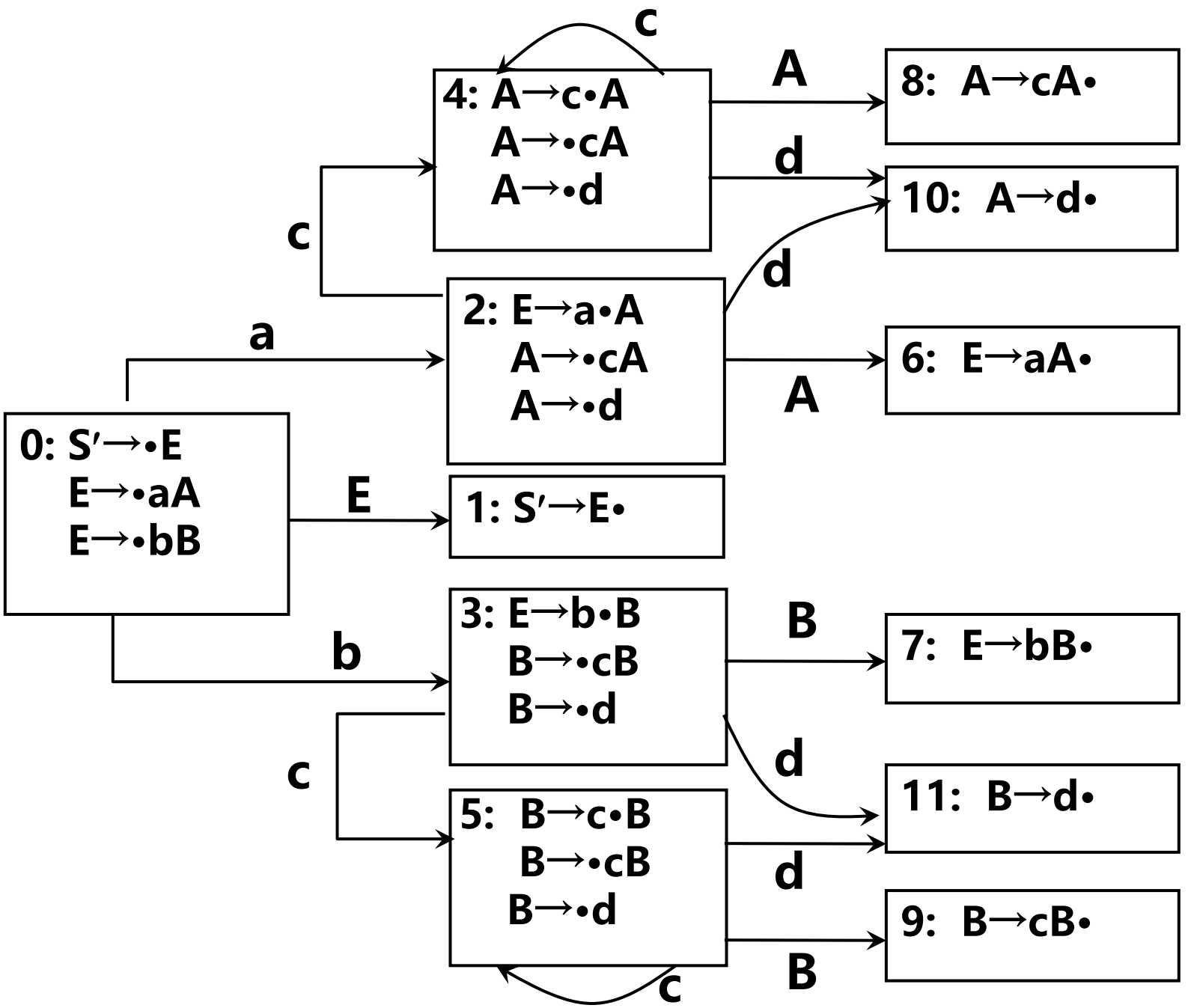
15. B→c•B　16. B→cB•

17. B→•d　18. B→d•



识别活前缀的NFA

识别活前缀的DFA


0: S′→•E
E→•aA
E→•bB
1: S′→E•
2: E→a•A
A→•cA
A→•d
3: E→b•B
B→•cB
B→•d
4: A→c•A
A→•cA
A→•d
5: B→c•B
B→•cB
B→•d
6: E→aA•
7: E→bB•
8: A→cA•
9: B→cB•
10: A→d•
11: B→d•
a
E
b
c
A
d
c
d
A
B
d
d
B
c

有效项目

- 项目 $A\rightarrow \beta_1 \cdot \beta_2$ 对活前缀 $\alpha\beta_1$ 是**有效**的，其条件是存在规范推导

$$S' \overset{*}{\underset{R}{\Rightarrow}} \alpha A\omega \underset{R}{\Rightarrow} \alpha\beta_1\beta_2\omega$$

- 活前缀$X_1X_2 \ldots X_m$的有效项目集从识别活前缀的DFA的初态出发，读出$X_1X_2 \ldots X_m$后到达的那个项目集(状态)。
- 在任何时候，分析栈中的活前缀$X_1X_2 \ldots X_m$的有效项目集正是栈顶状态S_m所代表的那个集合。

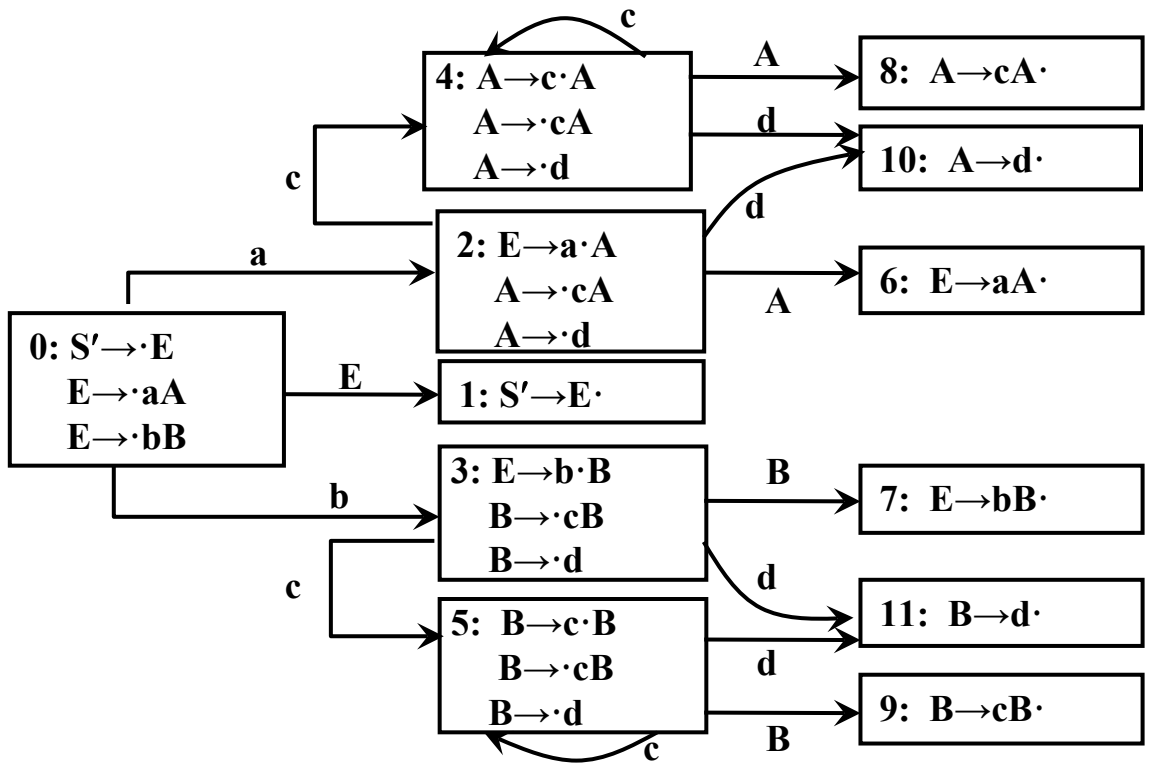
- G(S′)

  S′→E

  E→aA|bB

  A→cA|d

  B→cB|d



- 考虑活前缀：bc：

  项目：B → c.B　　B → . cB　　B → . d

  S′ ⇒ E ⇒ bB ⇒ bcB

  S′ ⇒ E ⇒ bB ⇒ bcB ⇒bccB

  S′ ⇒ E ⇒ bB ⇒ bcB ⇒ bcd

项目集I的闭包CLOSURE(I)

- 假定I是文法G′的任一项目集，定义和构造I的闭包CLOSURE(I)如下：

  1．I 的任何项目都属于CLOSURE(I)；

  2．若A→α•Bβ属于CLOSURE(I)，那么，对任何关于B的产生式B→γ，项目B→•γ也属于CLOSURE(I)；

  3．重复执行上述两步骤直至CLOSURE(I) 不再增大为止。

方法二： LR(0)项目集规范族

- 构成识别一个文法活前缀的DFA的项目集（状态）的全体称为文法的**LR(0)项目集规范族**。
  - $A \to \alpha \bullet$称为"归约项目"
  - 归约项目 $S' \to \alpha \bullet$ 称为"接受项目"
  - $A \to \alpha \bullet a\beta$ $(a \in V_T)$ 称为"移进项目"
  - $A \to \alpha \bullet B\beta$ $(B \in V_N)$ 称为"待约项目".

状态转换函数GO(I，X)

- GO是一个状态转换函数。I是一个项目集，X是一个文法符号。函数值GO(I，X)定义为：

$$GO(I，X) = CLOSURE(J)$$

其中

J = {任何形如$A\rightarrow\alpha X\bullet\beta$的项目| $A\rightarrow\alpha\bullet X\beta$属于I}。

直观上说，若I是对某个活前缀 γ有效的项目集，那么，GO(I，X)便是对 γX 有效的项目集。

示例

文法G(S′)：S′→E

E→aA|bB

A→cA|d

B→cB|d

设I={S' → •E}

则CLOSURE(I)={S' → •E, E→ •aA, E→ •bB }

设I_0={S′→•E, E→•aA, E→•bB}

GO(I_0, E)= closure(J)=closure({S' →E•})

= {S' →E•} = I_1

GO(I_0, a)= closure(J)=closure({E→a•A})

={ E→a•A, A→•cA, A→•d})=I_2

GO(I_0, b)= closure(J)=closure ({E →b.B})

={E →b•B, B→•cB, B→•d}= I_3

LR(0)项目集规范族构造算法

```
PROCEDURE  ITEMSETS(G′);
BEGIN
    C:={CLOSURE({S′→•S})};
    REPEAT
        FOR  C中每个项目集I和G′的每个符号X  DO
            IF  GO(I，X)非空且不属于C   THEN
                把GO(I，X)放入C族中;
    UNTIL C     不再增大
END
```

- 转换函数GO把项目集连接成一个DFA转换图

示例

文法G(S′)：　S′→E

E→aA|bB

A→cA|d

B→cB|d

C=CLOSURE({S′→•E})={{S′→•E, E→. aA, E→. bB }}=I_0

GO(I_0, E)= closure(J)=closure({S′ →E•})

= {S′ →E•} = I_1

GO(I_0, a)= closure(J)=closure({E→a•A})

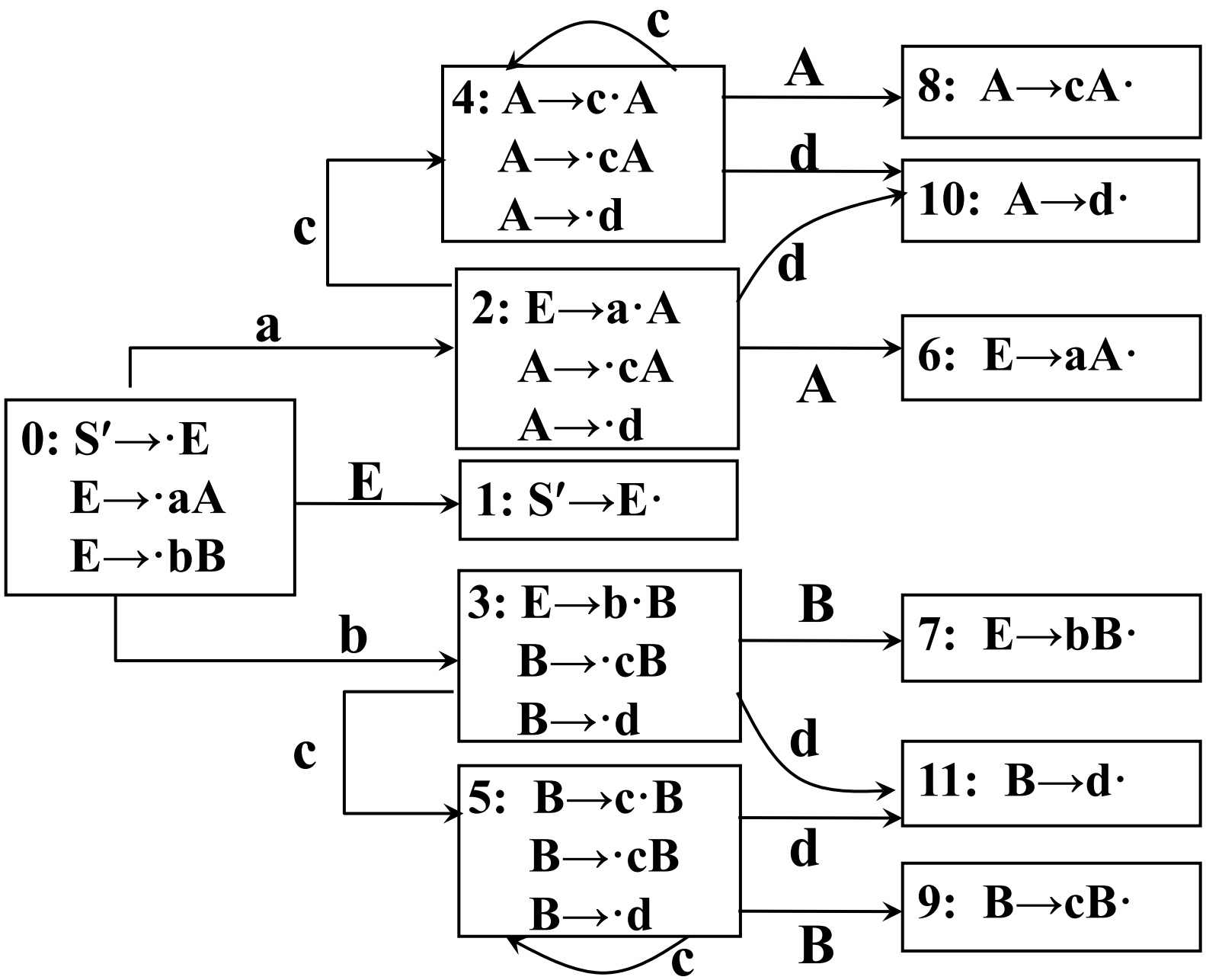
={ E→a•A, A→•cA, A→•d})=I_2

GO(I_0, b)= closure(J)=closure ({E →b.B})

={E →b•B, B→•cB, B→•d}= I_3

. . . . . .

可得12个项目集, 即为上面的DFA


0: S′→·E
E→·aA
E→·bB
1: S′→E·
2: E→a·A
A→·cA
A→·d
3: E→b·B
B→·cB
B→·d
4: A→c·A
A→·cA
A→·d
5: B→c·B
B→·cB
B→·d
6: E→aA·
7: E→bB·
8: A→cA·
9: B→cB·
10: A→d·
11: B→d·
a
b
c
E
A
d
B

分析表

- LR分析器的核心是一张分析表
  - ACTION[s, a]：当状态s面临输入符号a时，应采取什么动作.
  - GOTO[s, X]：状态s面对文法符号X时，下一状态是什么
    - GOTO[s, X]定义了一个以文法符号为字母表的DFA

Action[s, a]

- 每一项ACTION[s, a]所规定的四种动作:

1. **移进** 把(s, a)的下一状态s′和输入符号a推进栈，下一输入符号变成现行输入符号.
2. **归约** 指用某产生式A→β进行归约．假若β的长度为r， 归约动作是:去除栈顶r个项，使状态s_{m-r}变成栈顶状态，然后把(s_{m-r}, A)的下一状态s′ =GOTO[s_{m-r}, A]和文法符号A推进栈.
3. **接受** 宣布分析成功，停止分析器工作.
4. **报错**

构造LR(0)分析表的算法

- 令每个项目集I_k的下标k作为分析器的状态
- 包含项目$S'\rightarrow\bullet S$的集合I_k的下标k为分析器的初态。

分析表的ACTION和GOTO子表构造方法：

1. 若项目A→α•aβ属于I_k且GO(I_k, a) = I_j，a为终结符，则置ACTION[k,a] 为“sj”。
2. 若项目A→α•属于I_k，那么，对任何终结符a(或结束符#)，置ACTION[k,a]为 “rj”(假定产生式A→α是文法G′的第j个产生式)。
3. 若项目S′→S•属于I_k，则置ACTION[k,#]为 “acc”。
4. 若GO(I_k,A) = I_j，A为非终结符，则置GOTO[k,A]=j。
5. 分析表中凡不能用规则1至4填入信息的空白格均置上“报错标志”。

文法G(S′)

(0) S′→E

(1) E→aA

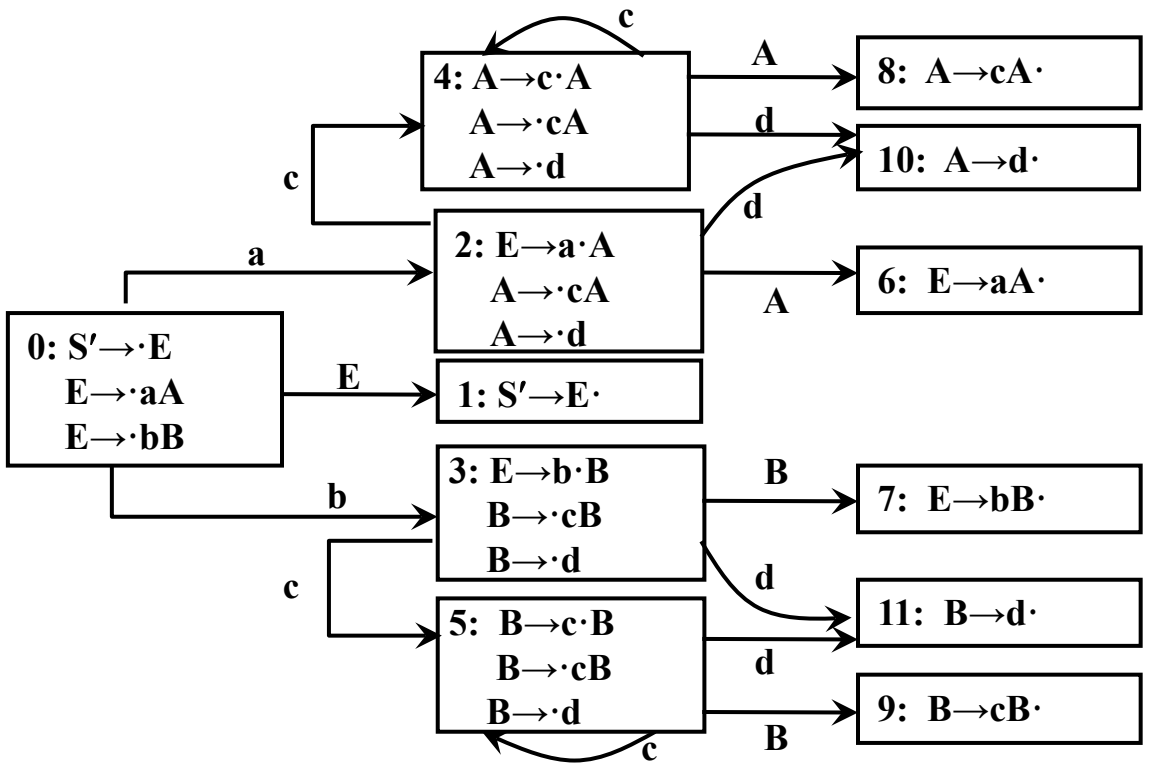
(2) E→bB

(3) A→cA

(4) A→d

(5) B→cB

(6) B→d



	ACTION					GOTO		
状态	a	b	c	d	#	E	A	B
0								
1								
2								
3								
4								
5								9
6								
7								
8								
9								
10								
11								

分析过程

三元式（栈内状态序列，移进归约串，输入串）的变化：

开　始：（　S_0，　　#，　　$a_1a_2 \ldots a_n$# ）

某一步：（$S_0S_1 \ldots S_m$，　　$\#X_1X_2 \ldots X_m$，　　$a_i\, a_{i+1} \ldots a_n$ #）

下一步：　ACTION [S_m, a_i]

若 ACTION [S_m, a_i] 为 “移进” 且GOTO [S_m, a_i] = S

则三元式为

（$S_0S_1 \ldots S_m$ **S**，　　$\#X_1X_2 \ldots X_m$**a_i** ，　　$a_{i+1} \ldots a_n$# ）

若 ACTION [S_m, a_i] 为 “ 归约” { A→β},

且｜β｜ = r ,β= $X_{m-r+1} \ldots X_m$, GOTO [S_{m-r}, A] =S,

则三元式为：

（$S_0S_1 \ldots S_{m-r}$ **S**，　　$\#X_1X_2 \ldots X_{m-r}$ **A**，　$a_i\, a_{i+1} \ldots a_n$ # ）

若ACTION [S_m, a_i] 为 “ 接受” 则结束

若ACTION [S_m, a_i] 为 “ 报错” 则进行出错处理

示例

按上表对acccd进行分析

步骤	状态	符号	输入串
1	0	#	acccd#
2	02	#a	cccd#
3	024	#ac	ccd#
4	0244	#acc	cd#
5	02444	#accc	d#
6	0244410	#acccd	#
7	024448	#acccA	#
8	02448	#accA	#
9	0248	#acA	#
10	026	#aA	#
11	01	#E	#

LR文法

- LR文法: 对于一个文法，如果能够构造一张LR分析表，使得它的每个入口均是唯一确定，则该文法称为LR文法。
  - 在进行自下而上分析时， 一旦栈顶形成句柄，即可归约。
- LR(k)文法：对于一个文法，如果每步至多向前检查 k个输入符号，就能用LR分析器进行分析。则这个文法就称为LR(k)文法。
  - 大多数程序语言，符合LR(1)文法
- LR(0) 文法：k ＝0，即只要根据当前符号和历史信息进行分析，而无需展望。

LR（0）文法

- 假若一个文法G的拓广文法G′ 的活前缀识别自动机中的每个状态（项目集）**不存在**下述情况：

  （1） 既含移进项目又含归约项目；

  （2）含有多个归约项目

  则称G是一个**LR(0)文法**。

SLR(1)分析表的构造

- 只有当一个文法G是LR(0)文法，才能构造出LR(0)分析表；
- 由于大多数适用的程序设计语言的文法不能满足LR(0) 文法的条件，因此本节将介绍对于LR(0)规范族中冲突的项目集（状态）用向前查看一个符号的办法进行处理，以解决冲突。

SLR(1)分析表的构造

- 假定一个LR(0)规范族中含有如下的项目集（状态）I:

$$I=\{\ X\rightarrow \alpha\cdot b\beta,$$

$$A\rightarrow \alpha\cdot\ ,$$

$$B\rightarrow \alpha\cdot\ \}$$

- 该项目集中含有移进-归约冲突和归约-归约冲突。

如何解决这种冲突?

- LR（0）在归约时不向前看输入符号；

- 在LR（0）基础上，如果存在移进-归约冲突或归约-归约冲突，则LR（k）方法通过向前看k个输入符号来解决冲突（利用上下文信息来消除当前的歧义）

- 因为只对有冲突的状态才向前查看一个符号，以确定做哪种动作，因而称这种分析方法为**简单的LR(1)分析法**，用**SLR(1)**表示

- 对于归约项目A→ α·，B → α· 分别求Follow(A)和Follow(B)，如果满足如下条件
  - FOLLOW(A)∩FOLLOW(B) = ϕ
  - FOLLOW(A)∩{b} = ϕ
  - FOLLOW(B)∩{b} = ϕ
- 那么，当在状态I时面临某输入符号为a时，则构造分析表时用以下方法即可解决冲突动作。

  (1) 若a=b，则移进。

  (2) 若a∈Follow(A)，则用A → α 进行归约。

  (3) 若a∈Follow(B)，则用B → α 进行归约。

  (4) 此外，报错。

SLR(1)文法

假若一个文法G的拓广文法G′的活前缀识别自动机中对于形如I={ X→ α • bβ, A → α• , B → α• }含有移进-归约冲突和归约-归约冲突的状态（项目集）满足下述情况：

- FOLLOW(A)∩FOLLOW(B) = ϕ
- FOLLOW(A)∩{b} = ϕ
- FOLLOW(B)∩{b} = ϕ

- 则称G是一个SLR(1)文法。

SLR(1)分析表的构造

- 例如文法：

(0) S′ →E

(1) E→E+T

(2) E→T

(3) T→T*F

(4) T→F

(5) F→(E)

(6) F→i

状态描述序列如下：

状态	项目集	后继符号	后继状态
I_0	{ S′ → • E E→ • E+T E→ • T T → • T*F T → • F F→ • (E) F→ • i }	E E T T F (i	S_1 S_1 S_2 S_2 S_3 S_4 S_5
I_1	{ S′ →E • E→E • +T }	#S′ →E +	S_{12} S_6
I_2	{E→ T • T → T • *F }	#E→T • *	S_{12} S_7
I_3	{ T →F • }	#T→F	S_{12}

状态	项目集	后继符号	后继状态
I_4	{F→ (• E) E→• E+T E→ • T T →• T *F T →• F F→• (E) F→• i }	E E T T F (i	S_8 S_8 S_2 S_2 S_3 S_4 S_5
I_5	{F→ i • }	#F→ i	S_{12}
I_6	{E→E + •T T→ • T*F T → • F F→• (E) F→• i }	T T F (i	S_9 S_9 S_3 S_4 S_5
I_7	{T→T* • F F→ • (E) F→ • i }	F (i	S_{10} S_4 S_5

状态	项目集	后继符号	后继状态
I_8	{F→(E •) E→E • +T }	) +	S_{11} S_6
I_9	{E→E+T • T→T • *F }	#E→E+T *	S_{12} S_7
I_{10}	{T→T*F • }	#T→T*F	S_{12}
I_{11}	{F→(E) • }	#F→(E)	S_{12}
I_{12}	{ }		

由上图可见，I1、I2和I9的项目集均不相容，其有移进项目和归约项目并存，构造LR(0)分析表如下：

状态	ACTION						GOTO		
	i	+	*	(	)	#	E	T	F
S_0	S_5			S_4			1	2	3
S_1		S_6				acc			
S_2	r_2	r_2	$r_2\ S_7$	r_2	r_2	r_2			
S_3	r_4	r_4	r_4	r_4	r_4	r_4			
S_4	S_5			S_4			8	2	3
S_5	r_6	r_6	r_6	r_6	r_6	r_6			
S_6	S_5			S_4				9	3
S_7	S_5			S_4					10
S_8		S_6			S_{11}				
S_9	r_1	r_1	$r_1\ S_7$	r_1	r_1	r_1			
S_{10}	r_3	r_3	r_3	r_3	r_3	r_3			
S_{11}	r_5	r_5	r_5	r_5	r_5	r_5			

SLR（1）方法

- 从上表也可见在S_1，S_2，S_9中存在移进-归约冲突 。这个表达式不是LR(0)文法，也就不能构造LR(0)分析表，现在分别考查这三个项目（状态）中的冲突是否能用SLR(1)方法解决。
- 对于S1：{S’ →E •，E→E • + T }
  - 由于Follow(S’)={＃}，而S’ →E•是唯一的接受项目，所以当且仅当遇到句子的结束符“＃”号时才被接受。又因{＃}∩{＋}=ϕ，因此S_1中的冲突可解决。

SLR（1）方法

- 对于S_2：S_2={ E →T • , T→T • *F }
  - 计算Follow(E) = { #, +,) }
  - 所以Follow(E)∩{*}=ϕ
  - 因此面临输入符为‘+’，‘)’或‘#’号时，则用产生式E→T进行归约。
  - 当面临输入符为‘*’号时，则移进, 其它情况则报错。

SLR（1）方法

- 对于S_9：S_9={ E →E+T •，T→T • *F }
  - 计算Follow(E) = { #, +,) }
  - 所以Follow(E)∩{*}=ϕ
  - 因此面临输入符为‘+’，‘)’或‘#’号时，则用产生式E→E+T进行归约。
  - 当面临输入符为‘*’号时，则移进。 其它情况则报错。

总结（1）

- 由以上考查，该文法在S_1，S_2，S_9三个项目集(状态)中存在的移进-归约冲突都可以用SLR(１)方法解决，因此该文法是SLR(１)文法。我们可构造其相应的SLR(１)分析表。
- SLR(1)分析表的构造与LR(0)分析表的构造类似，仅在含有冲突的项目集中分别进行处理。

总结（2）

- 进一步分析我们可以发现如下事实：如在状态S_3中，只有一个归约项目T→F·，按照SLR(1)方法，在该项目中没有冲突，所以保持原来LR(0)的处理方法，不论当前面临的输入符号是什么都将用产生式T→F进行归约。
- 但很显然T的Follow集合不含‘(’符号，如果当前面临输入符是‘(’，也进行归约是错误的。

总结（3）

- 因此可以对所有归约项目都采取SLR(1)的思想，即对所有非终结符都求出其Follow集合，这样凡是归约项目仅对面临输入符号包含在该归约项目左部非终结符的Follow集合中，才采取用该产生式归约的动作。
- 对于这样构造的SLR(1)分析表，称它为改进的SLR(1)分析表。

改进的SLR(1)分析表的构造方法

(1) 对于$A\rightarrow\alpha\cdot X\beta$，$GO[S_i, X]\in S_j$，若

$X\in V_T$，则置$actoin[S_i, X]=S_j$

$X\in V_N$，则置$goto[S_i, X]=j$

(2) 对于归约项目$A\rightarrow\alpha\cdot\in S_i$，若$A\rightarrow\alpha$为文法的第j个产生式，则对任何输入符号a，若$a\in Follow(A)$，则置$action[S_i, a]=r_j$

(3) 若$S\rightarrow\alpha\cdot\in S_i$，则置$action[S_i, \#]=acc$

(4) 其它情况均置出错。

状态	ACTION						GOTO		
	i	+	*	(	)	#	E	T	F
S_0	S_5			S_4			1	2	3
S_1		S_6				acc			
S_2		r_2	S_7		r_2	r_2			
S_3		r_4	r_4		r_4	r_4			
S_4	S_5			S_4			8	2	3
S_5		r_6	r_6		r_6	r_6			
S_6	S_5			S_4				9	3
S_7	S_5			S_4					10
S_8		S_6			S_{11}				
S_9		r_1	S_7		r_1	r_1			
S_{10}		r_3	r_3		r_3	r_3			
S_{11}		r_5	r_5		r_5	r_5			

SLR文法的局限性

- 非SLR文法示例：

  (0) S′→S

  (1) S→L=R

  (2) S→R

  (3) L→*R

  (4) L→i

  (5) R→L

这个文法的LR(0)项目集规范族为：

I_0：S′→•S
S→•L=R
S→•R
L→•*R
L→•i
R→•L

I_1：S′→S•

I_2：S→L•=R
R→L•

I_3：S→R•

I_4：L→*•R
R→•L
L→•*R
L→•i

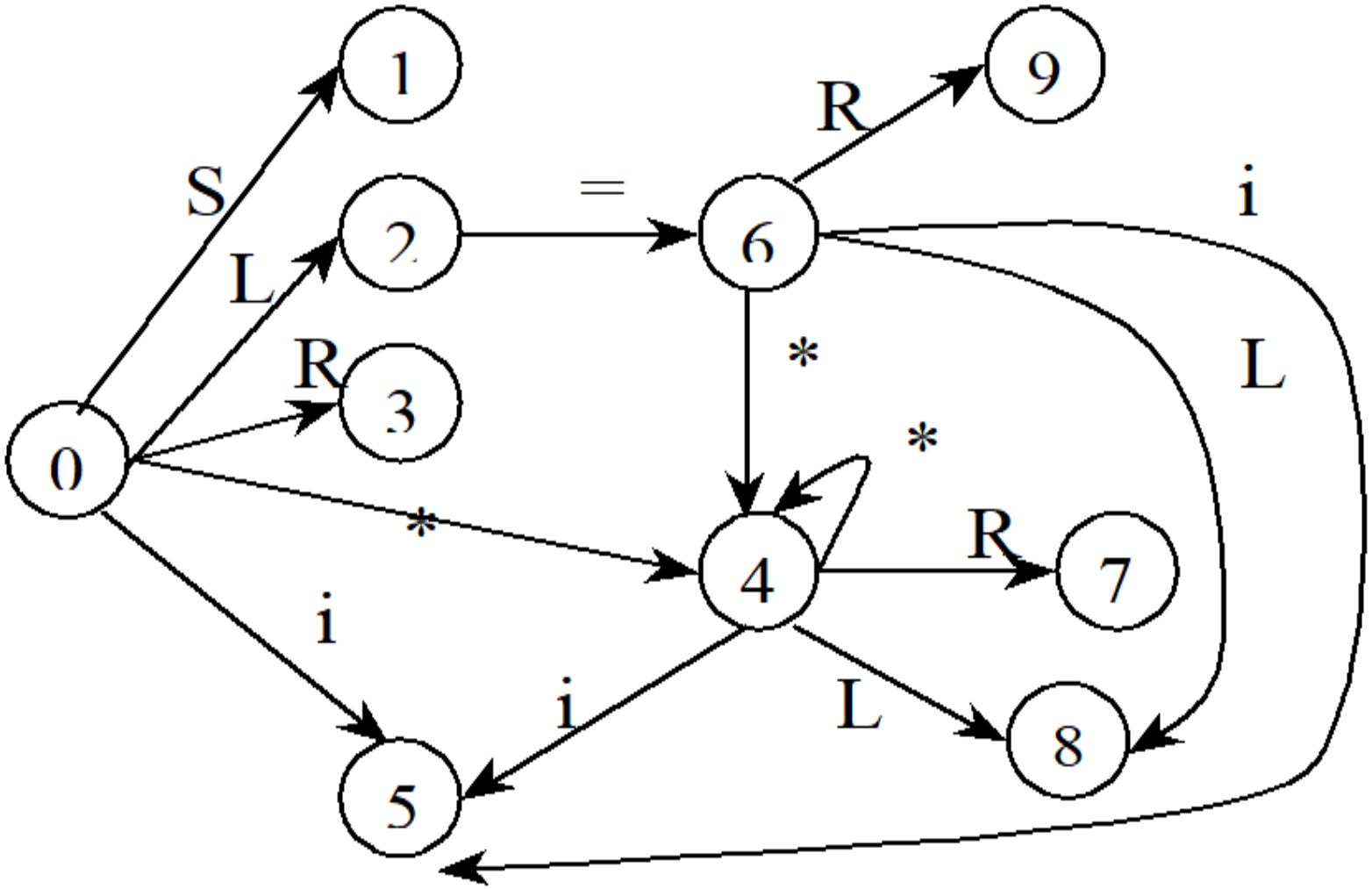
I_5：L→i•

I_6：S→L=•R
R→•L
L→•*R
L→•i

I_7：L→*R•

I_8：R→L•

I_9：S→L=R•



活前缀识别器

SLR文法的局限性

(0) S′→S

(1) S→L=R

(2) S→R

(3) L→*R

(4) L→i

(5) R→L

I_2: S→L•=R
R→L•

- I_2含有“移进－归约”冲突。
- FOLLOW(R) = {#, =}，移进项目和规约项目都是=，SLR(1)无法解决

计算FOLLOW集合所得到的超前符号集合可能大于实际能出现的超前符号集。

SLR文法的局限性

(0) S′→S

(1) S→L=R

(2) S→R

(3) L→*R

(4) L→i

(5) R→L

I_2: S→L•=R
R→L•

实际上 “R=”为前缀的规范句型不会出现

而是含有“*R=”为前缀的规范句型

FOLLOW集合不能提供精确的展望信息

- 在SLR方法中,如果项目集I_i含项目$A\rightarrow\alpha\bullet$而且下一输入符号$a\in FOLLOW(A)$,则状态i面临a时,可选用“用$A\rightarrow\alpha$归约”动作。
- 但在有些情况下,当状态i显现于栈顶时,栈里的**活前缀**未必允许把α归约为A，因为可能根本就不存在一个形如“βAa”的规范句型。因此，在这种情况下，用“ $A\rightarrow\alpha$ ”归约不一定合适。

规范LR分析冲突解决方法

- 重新定义项目，使得每个项目都附带有k个终结符。每个项目的一般形式是$[A\rightarrow\alpha\bullet\beta, a_1a_2...a_k]$，这样的一个项目称为一个**LR(k)项目**。项目中的 $a_1a_2...a_k$ 称为它的**向前搜索符串**(或**展望串**)。
- **向前搜索符串仅对归约项目**$[A\rightarrow\alpha\bullet, a_1a_2...a_k]$**有意义**。对于任何**移进或待约项目**$[A\rightarrow\alpha\bullet\beta, a_1a_2...a_k]$, $\beta\neq\varepsilon$，搜索符串 $a_1a_2...a_k$ 没有作用。
- 归约项目$[A\rightarrow\alpha\bullet, a_1a_2...a_k]$意味着：当它所属的状态呈现在栈顶且后续的k个输入符号为 $a_1a_2...a_k$ 时，才可以把栈顶上的α归约为A。

- 一般情况，向前搜索(展望)一个符号就多半可以确定“移进”或“归约”。
- 形式上我们说一个LR(1)项目[A→α•β, a]对于活前缀γ是**有效的**，如果存在规范推导

$$S \overset{*}{\Rightarrow} \delta A\omega \Rightarrow \delta\alpha\beta\omega$$

其中，1）$\gamma = \delta\alpha$；2）a是ω的第一个符号，或者a为#而ω为ε。

- 为构造有效的LR(1)项目集族，需要两个函数CLOSURE和GO。

- **项目集I 的闭包CLOSURE(I)构造方法：**

1. I的任何项目都属于CLOSURE(I)。

2. 若项目[A→α•Bβ, *a*]属于CLOSURE(I)，B→ξ 是一个产生式，那么，对于FIRST(β*a*) 中的每个终结符*b*，如果[B→•ξ, *b*]原来不在CLOSURE(I)中，则把它加进去。

3. 重复执行步骤2，直至CLOSURE(I)不再增大为止。

- 令I是一个项目集，X是一个文法符号，函数GO(I，X)定义为：

$$GO(I，X) = CLOSURE(J)$$

其中

$$J = \{\text{任何形如}[A\rightarrow\alpha X\cdot\beta, a]\text{的项目} \mid [A\rightarrow\alpha\cdot X\beta, a]\in I\}$$

- **文法G′的LR(1)项目集族C的构造算法：**

```
BEGIN
  C:={CLOSURE({[S′→•S，#]})};
  REPEAT
    FOR  C中每个项目集I和G′的每个符号X   DO
      IF GO(I，X)非空且不属于C，THEN
                把GO(I，X)加入C中
  UNTIL       C不再增大
END
```

- 构造LR(1)分析表的算法。
  - 令每个I_k的下标k为分析表的状态，令含有[S′→•S, #]的I_k的k为分析器的初态。

- 动作ACTION和状态转换GOTO构造如下：

1. 若项目[A→α•aβ, b]属于I_k且GO(I_k, a) = I_j， a为终结符，则置ACTION[k, a]为“ “sj”

2. 若项目[A→α•，a]属于I_k，则置ACTION[k, a]为 “rj”；其中假定A→α为文法G′的第j个产生式。

3. 若项目[S′→S•, #]属于I_k，则置ACTION[k, #]为 “acc”。

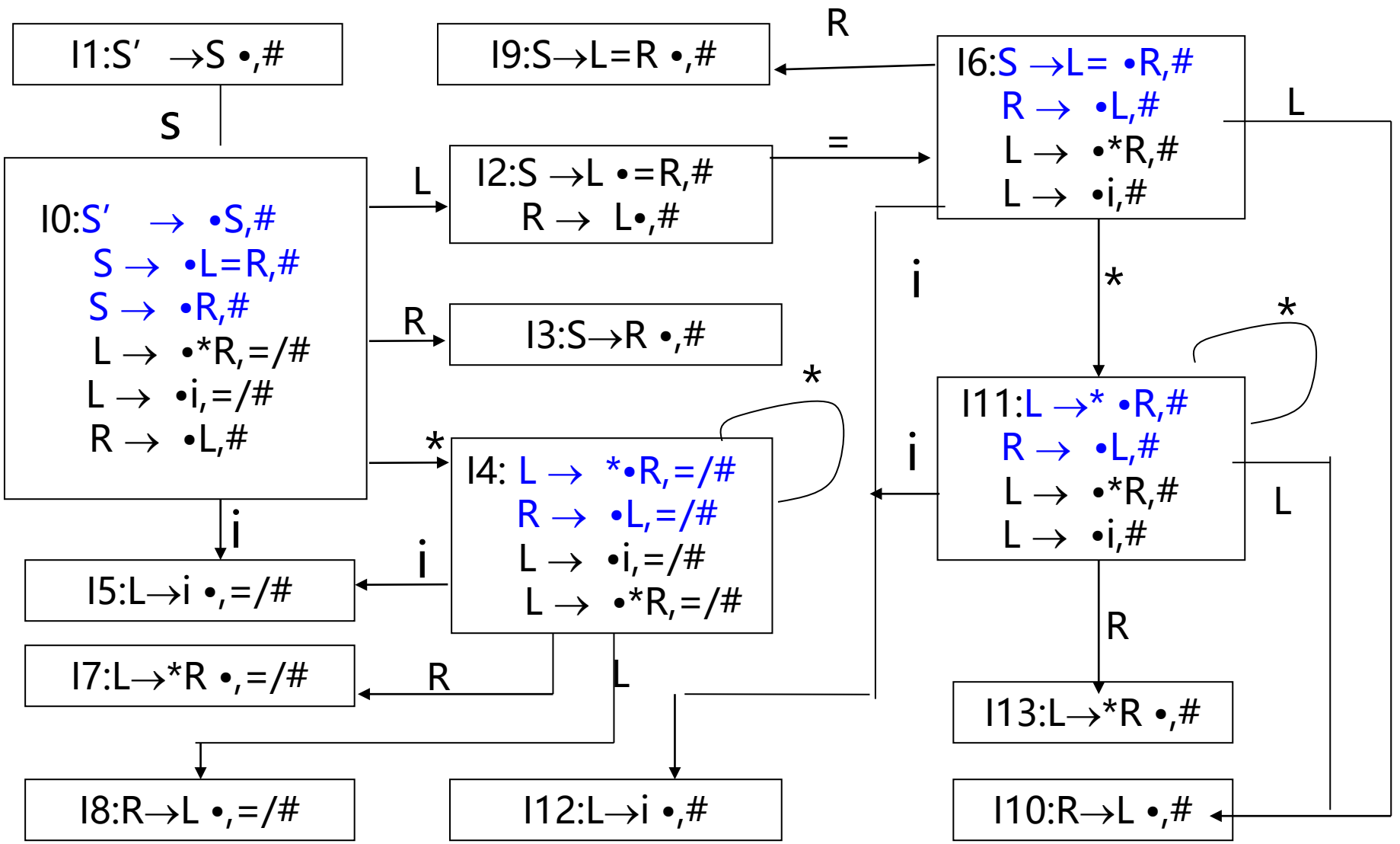
4. 若GO(I_k，A) = I_j，则置GOTO[k, A]=j。

5. 分析表中凡不能用规则1至4填入信息的空白栏均填上“出错标志”。

- 按上述算法构造的分析表，若不存在多重定义的入口(即，动作冲突)的情形，则称它是文法G的一张**规范的LR(1)分析表**。
- 使用这种分析表的分析器叫做一个**规范的LR分析器**。
- 具有规范的LR(1)分析表的文法称为一个**LR(1)文法**。
- LR(1)状态比SLR多，

$$LR(0) \subset SLR \subset LR(1) \subset 无二义文法$$

前例的 LR(1)项目集及转换函数



- **拓广文法G(S′)**

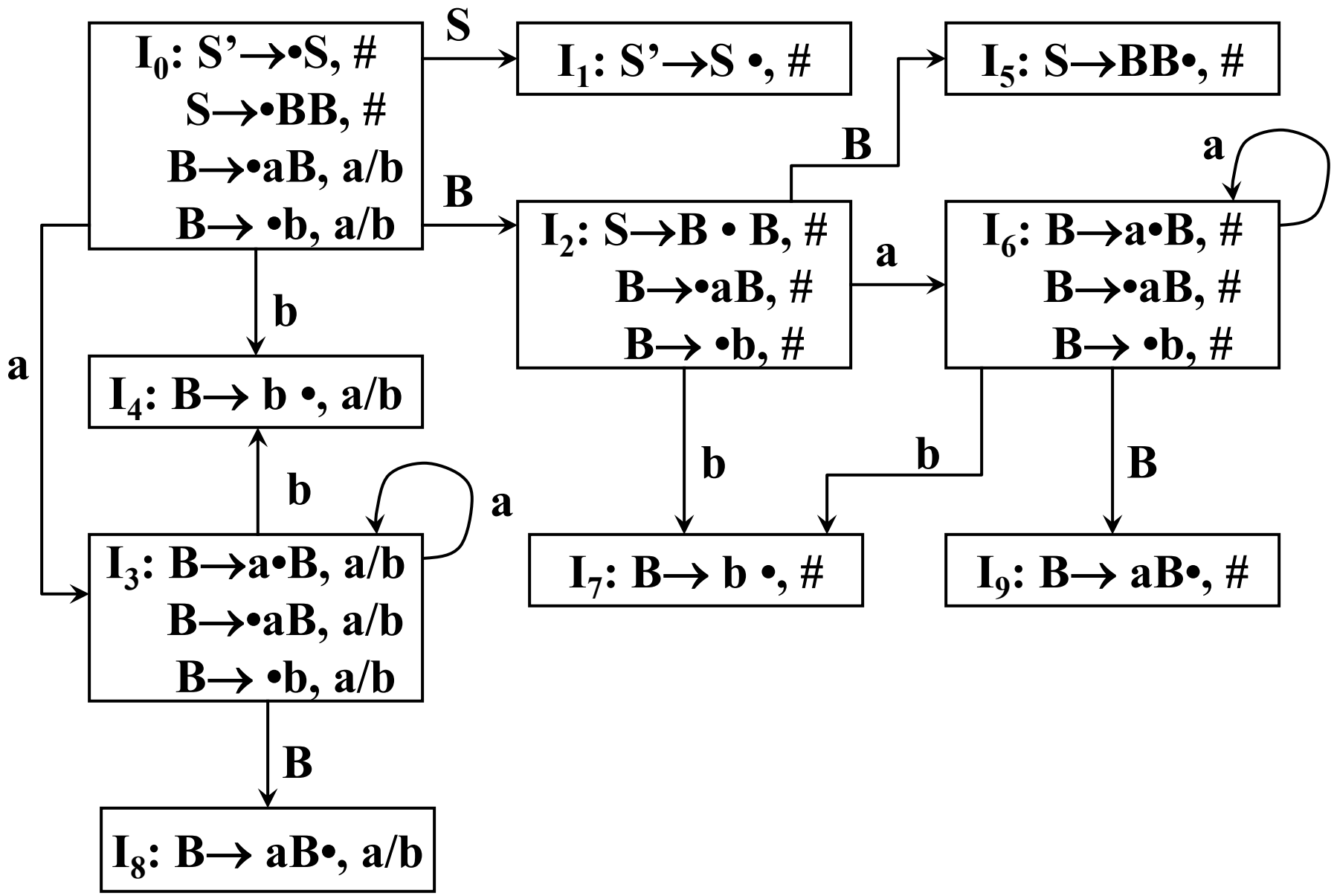
  (0) S′→S

  (1) S→BB

  (2) B→aB

  (3) B→b

LR(1)的项目集C和函数GO



LR(1)分析表为:

状态	ACTION			GOTO	
	a	b	#	*S*	B
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

- 按上表对aabab进行分析

步骤	状态	符号	输入串	
0	0	#	aabab#	
1	03	#*a*	abab#	
2	033	#aa	bab#	
3	0334	#aab	ab#	
4	0338	#aaB	ab#	
5	038	#aB	ab#	
6	02	#B	ab#	
7	026	#Ba	b#	
8	0267	#Bab	#	
9	0269	#BaB	#	
10	025	#BB	#	
11	01	#*S*	#	acc

- 按上表对abab进行分析

步骤	状态	符号	输入串	
0	0	#	abab#	
1	03	#*a*	bab#	
2	034	#ab	ab#	
3	038	#aB	ab#	
4	02	#B	ab#	
5	026	#Ba	b#	
6	0267	#Bab	#	
7	0269	#BaB	#	
8	025	#BB	#	
9	01	#*S*	#	acc

作业题

- P133
  - 1
  - 2　　(2)
  - 3　　(1)　(2)　(4)
  - 5　　(1)　(2)　(3)