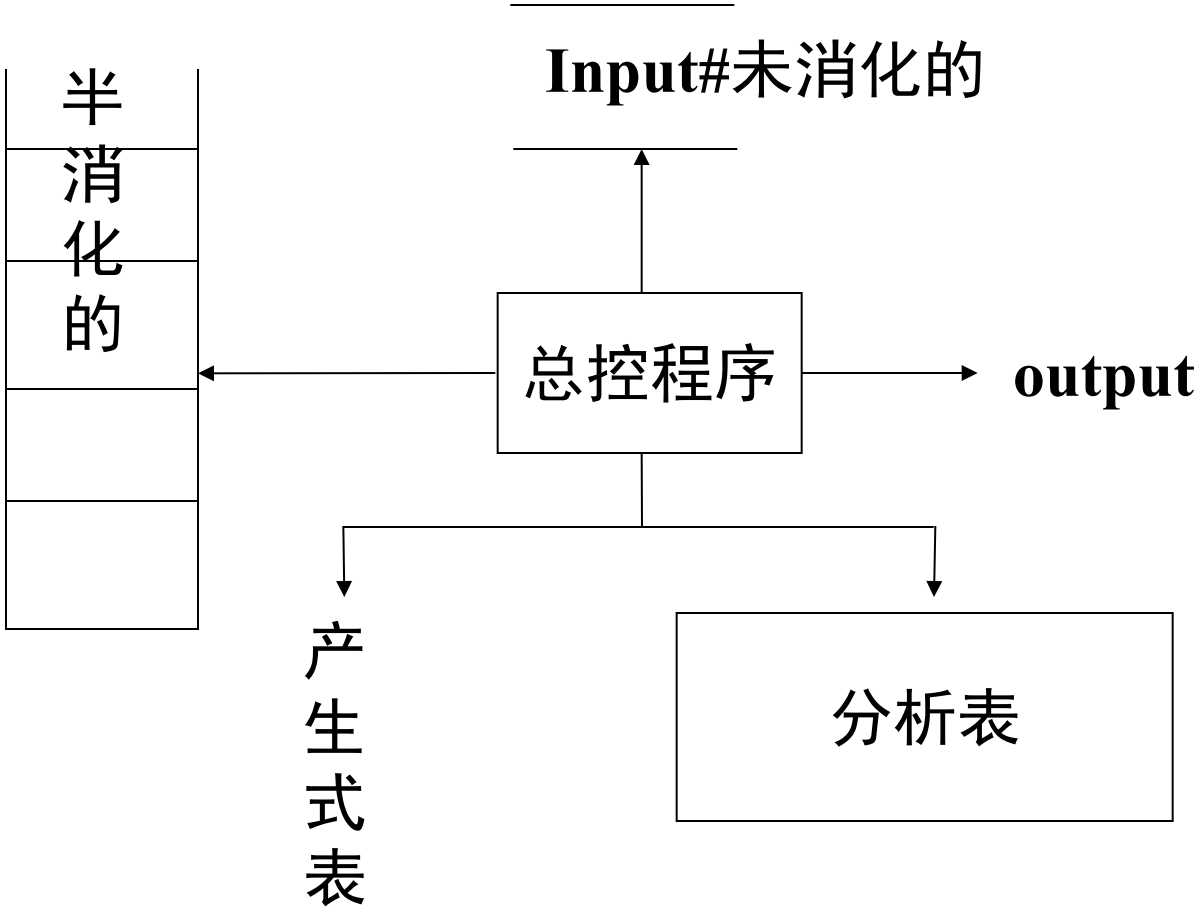
第6章　LR分析

自下而上分析算法：能力强、构造复杂

最常用和最有效的模型: 移进-归约

将输入分成两部分：**未消化的**和**半消化的**



S –> E　　E –> T | E + T　　T –> int | (E)

***Reduce*: 如能找到一产生式 A –> w 且栈中的内容是 qw (q 可能为空)，则可以将其归约为 qA，即倒过来用这个产生式。**

如上例，若栈中内容是 (int，我们使用产生式 T–> int 并把栈中内容归约为 (T。

***Shift*: 如不能执行一个归约且在未消化的输入中还有 token，就把它从输入移到栈中。**

如上例，假定栈中内容是 (，输入中还有 int+int)#。不能对(执行一个归约，因为它不和任何产生式的右端匹配。所以把输入的第一个符号移到栈中，于是栈中内容是 (int，而余留的输入是 +int)#。

Reduce的一个特殊情况：栈中的全部内容w归约为开始符号S（即施用 S –> w），且没有余留输入了，意味着已成功分析了整个输入串。

移进归约分析中还会出现一种情况，就是**出错**，比如当前的token不能构成一个合法句子的一部分，例如上面的文法，试分析 int+)时就会发生错误。

移进-归约模型分析(int + int)的过程

	STACK	REMAINING INPUT	PARSER ACTION
1		(int + int)#	Shift
2	(	int + int)#	Shift
3	(int	+ int)#	Reduce: T –> int
4	(T	+ int)#	Reduce: E –> T
5	(E	+ int)#	Shift
6	(E +	int)#	Shift
7	(E + int	)#	Reduce: T –> int
8	(E + T	)#	Reduce: E –> E + T
9	(E	)#	Shift
10	(E)	#	Reduce: T –> (E)
11	T	#	Reduce: E –> T
12	E	#	Reduce: S –> E
13	S	#	

S –> E　　E –> T | E + T　　T –> int | (E)

(E + T　　)#　　Reduce:E –> E + T

为什么不用 E –> T？

(E　　)#

若使用了E –> T，在栈中形成的 (E+E 不是规范句型的**活前缀**(viable prefixes)

(E+E 不能和任何产生式的右端匹配， (E+E) 不是规范句型

活前缀是规范句型的前缀，但不超过句柄

移进归约分析的栈中出现的内容加上剩余输入构成规范句型

规范推导 规范句型 规范归约

最右推导：在推导的任何一步$\alpha \Rightarrow \beta$，其中α、β是句型，都是对α中的最右非终结符进行替换

最右推导被称为**规范推导**

由规范推导所得的句型称为**规范句型**

G[S]： $S \rightarrow E \quad E \rightarrow E+T|T \quad T \rightarrow (E)|int$

$S \Rightarrow E \Rightarrow T \Rightarrow (E) \Rightarrow (E+T) \Rightarrow (E+int)$

$\Rightarrow (T+int) \Rightarrow (int+int)$

规范归约

假定α是G的一个句子，称序列$\alpha_n, \alpha_{n-1}, \ldots, \alpha_0$是$\alpha$的一个规范归约，如果该序列满足：

（1） $\alpha_n = \alpha$

（2） α_0为文法的开始符号

（3）对任何j, $0<j<=n$, α_{j-1}是从α_j经把句柄替换为相应产生式的左部而得到的

文法要求

shift-reduce or reduce-reduce 冲突（conflicts）

分析程序不能决定是*shift* 还是 *reduce*，**或者**

分析程序归约时有多个产生式可选

例子 (dangling *else*)：

S –> if E then S | if E then S else S

如输入为 if E then if E then S else S，分析某一时刻：

栈的内容: if E then if E then S，而下一 token是else，

那么，归约还是移进？

一种shift-reduce实现技术

LR 分析

LR的含义

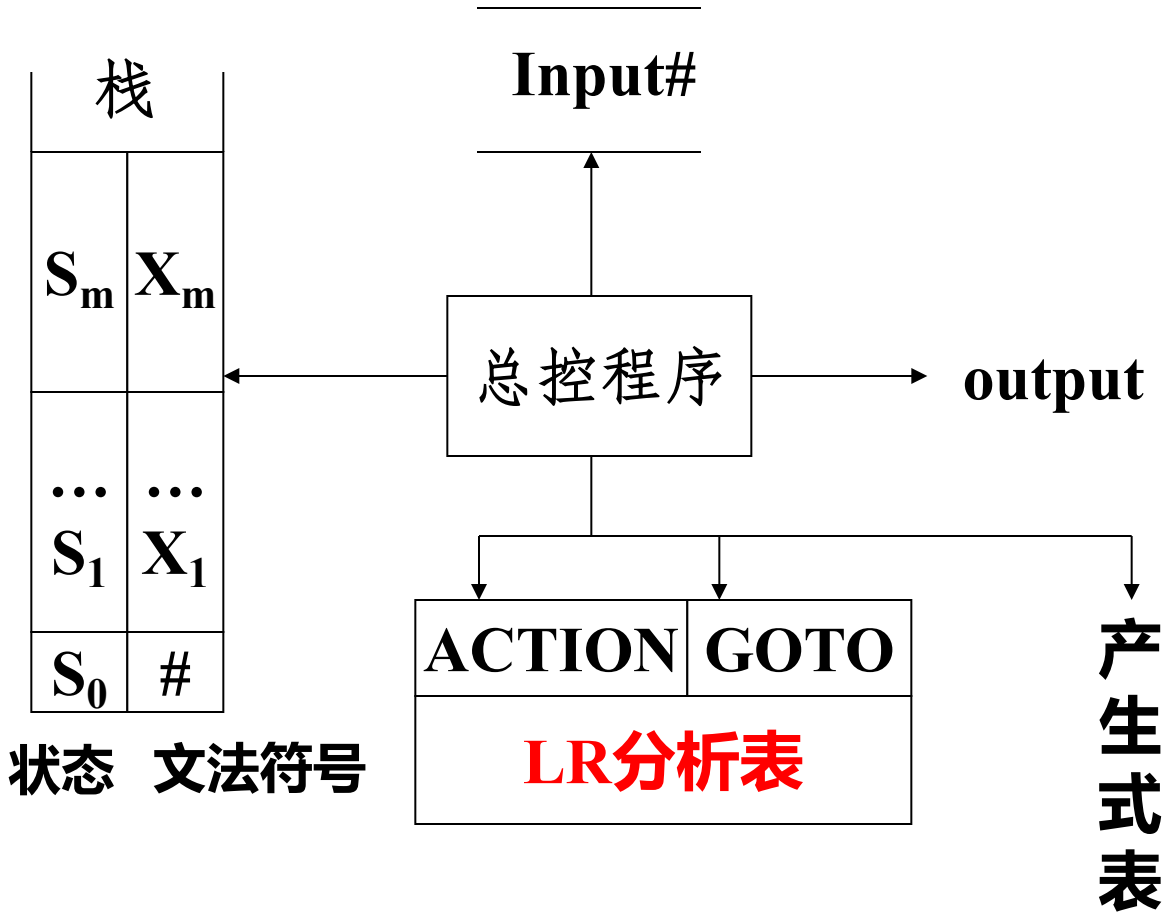
L

R 最右推导

分析器模型和分析算法

LR分析特征（分析表）

LR分析器模型


Input#
栈
S_m X_m
… …
S_1 X_1
S_0 #
状态 文法符号
总控程序
output
ACTION GOTO
LR分析表
产生式表

LR分析表

	ACTION						GOTO		
	a	c	e	b	d	#	S	A	B
0	S2						1		
1						acc			
2				S4				3	
3		S5		S6					
4	r2	r2	r2	r2	r2	r2			
5					S8				7
6	r3	r3	r3	r3	r3	r3			
7			S9						
8	r4	r4	r4	r4	r4	r4			
9	r1	r1	r1	r1	r1	r1			

LR分析算法

置ip指向输入串w的第一个符号
令S为栈顶状态
a是ip指向的符号
repeat begin
if ACTION[S,a]=S_j
then begin PUSH j,a(进栈)
ip 前进(指向下一输入符号)
end
else if ACTION[S,a]=r_j (第j条产生式为$A\rightarrow\beta$)

LR分析算法

then begin

pop |β| 项

令当前栈顶状态为S’

push GOTO[S’, A]和A(进栈)

end

else if ACTION[s,a]=acc

then return (成功)

else error

end

例6.1：

G[S]: S →aAcBe[1]

A →b[2]

A →Ab[3]

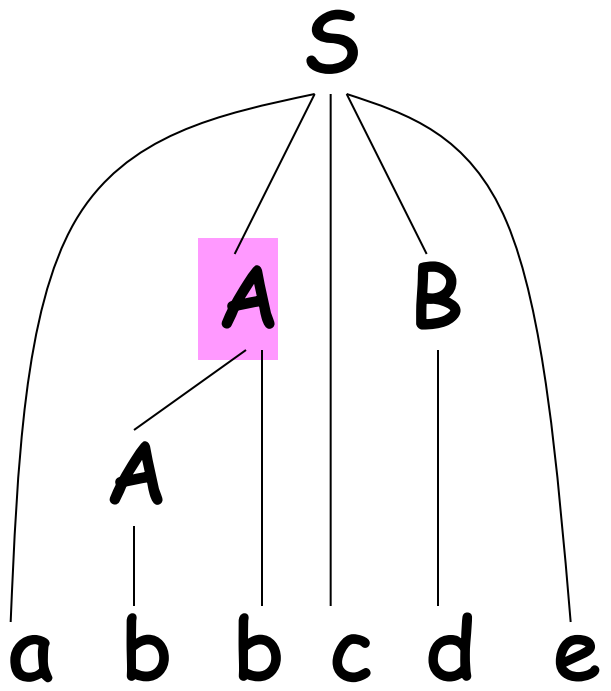
B →d[4]

w=abbcde#

Step	States	Syms	The rest of input	Action	Goto
1	0	#	abbcde#	s2	
2	02	#a	bbcde#	s4	
3	024	#ab	bcde#	r2	goto(2,A)
4	023	#aA	bcde#	s6	
5	0236	#aAb	cde#	r3	
6	023	#aA	cde#	s5	
7	0235	#aAc	de#	s8	
8	02358	#aAcd	e#	r4	
9	02357	#aAcB	e#	s9	
10	023579	#aAcBe	#	r1	
11	01	#S		acc	

文法G[S]：

(1) S → aAcBe
(2) A → b
(3) A → Ab
(4) B → d



步骤	符号栈	输入符号串	动作
1)	#	abbcde#	移进
2)	#a	bbcde#	移进
3)	#ab	bcde#	归约(A→b)
4)	#aA	bcde#	移进
5)	#aAb	cde#	归约(A→Ab)
6)	#aA	cde#	移进
7)	#aAc	de#	移进
8)	# aAcd	e#	归约(B→d)
9)	#aAcB	e#	移进
10)	#aAcBe	#	归约
11)	#S	#	接受

对输入串abbcde#的移进-归约分析过程

符号串abbcde是否是G[S]的句子

$$S \Rightarrow aAcBe \Rightarrow aAcde \Rightarrow aAbcde \Rightarrow abbcde$$

步骤	符号栈	输入符号串	动作	状态栈	ACTION	GOTO
1)	#	abbcde#	移进	0	S_2	
2)	#a	bbcde#	移进	02	S_4	
3)	#ab	bcde#	归约(A→b)	024	r_2	3
4)	#aA	bcde#	移进	023	S_6	
5)	#aAb	cde#	归约(A→Ab)	0236	r_3	3
6)	#aA	cde#	移进	023	S_5	
7)	#aAc	de#	移进	0235	S_8	
8)	#aAcd	e#	归约(B→d)	02358	r_4	7
9)	#aAcB	e#	移进	02357	S_9	
10)	#aAcBe	#	归约(S→aAcBe)	023579	r_1	1
11)	#S	#	接受	01	acc	

对输入串abbcde#的LR分析过程

文法G[S]：
(1) S → aAcBe
(2) A → b
(3) A → Ab
(4) B → d

s_i:移进，将状态i和输入符进栈

r_i:归约，用第i个产生式归约，同时状态栈与符号栈退出相应个符号，并把GOTO表相应状态和第i个产生式的左部非终结符入栈。

	ACTION						GOTO		
	a	c	e	b	d	#	S	A	B
0	s_2						1		
1						acc			
2				s_4				3	
3		s_5		s_6					
4	r_2	r_2	r_2	r_2	r_2	r_2			
5					s_8				7
6	r_3	r_3	r_3	r_3	r_3	r_3			
7			s_9						
8	r_4	r_4	r_4	r_4	r_4	r_4			
9	r_1	r_1	r_1	r_1	r_1	r_1			

LR文法

对于一个cfg文法，如果能够构造一张 LR 分析表，使得它的每一个表项（entry）均是唯一的（si, rj, acc, 空白），则称该 cfg 为LR 文法.

LR分析

特征：

- 规范的
- 符号栈中的符号串是规范句型的前缀，且其最右符号不超过句柄的末端（活前缀）
- 分析决策依据—栈顶状态和现行输入符号、识别活前缀的 DFA

四种技术：

- LR(0)　SLR(1)　LR(1)　LALR(1)

LR(0) 分析

LR(0)文法

- 能力最弱，理论上最重要
- 存在识别活前缀的FA
- 如何构造识别活前缀的DFA

  （LR(0)项目集规范族的构造）
- LR(0)分析表的构造

拓广文法

为使文法的初始符号不出现在任何产生式的右部，需对文法G[S]进行拓广：在原文法G中增加S'→S产生式。

文法G[S]：
(1) S → aAcBe
(2) A → b
(3) A → Ab
(4) B → d

文法G[S']：
(0) S'→S
(1) S → aAcBe
(2) A → b
(3) A → Ab
(4) B → d

最右推导过程：

S' ⇒ S[0]
⇒ aAcBe[1][0]
⇒ aAcd[4]e[1][0]
⇒ aAb[3]cd[4]e[1][0]
⇒ ab[2]b[3]cd[4]e[1][0]

归约时在栈中的句型的前缀	归约前可在栈中出现的前缀(不含句柄)
ab[2]	a,ab
aAb[3]	a,aA,aAb
aAcd[4]	a,aA,aAc,aAcd
aAcBe[1]	a,aA,aAc,aAcB,aAcBe
S[0]	S

活前缀

给定文法G=(Vn,Vt, P, S), 若有规范推导:

$S' \Rightarrow_R^* \alpha A w \Rightarrow_R \alpha\beta w$, γ是αβ的前缀,

则称γ是文法G的活前缀.

例如：

a,ab

a,aA,aAb

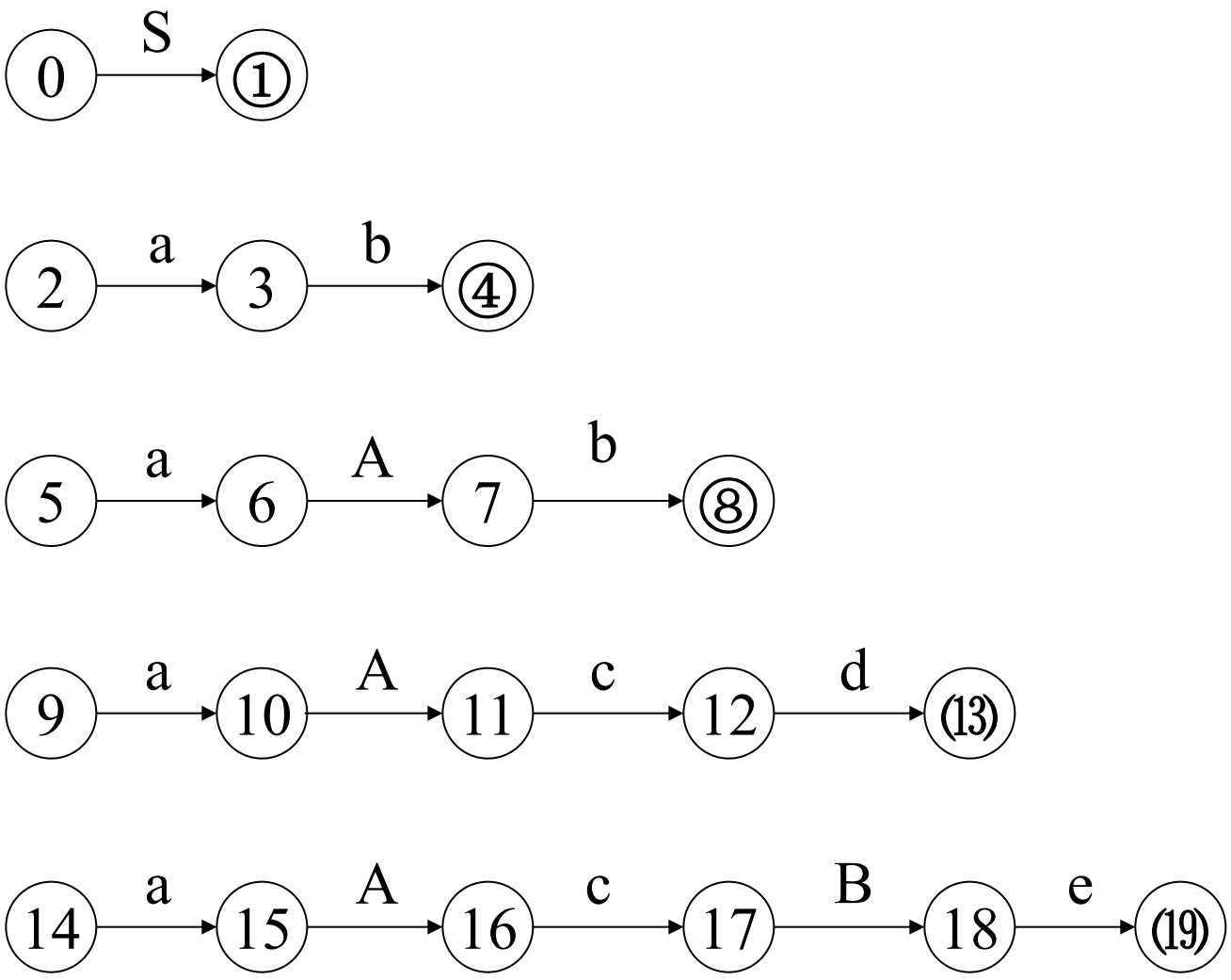
a,aA,aAc,aAcd

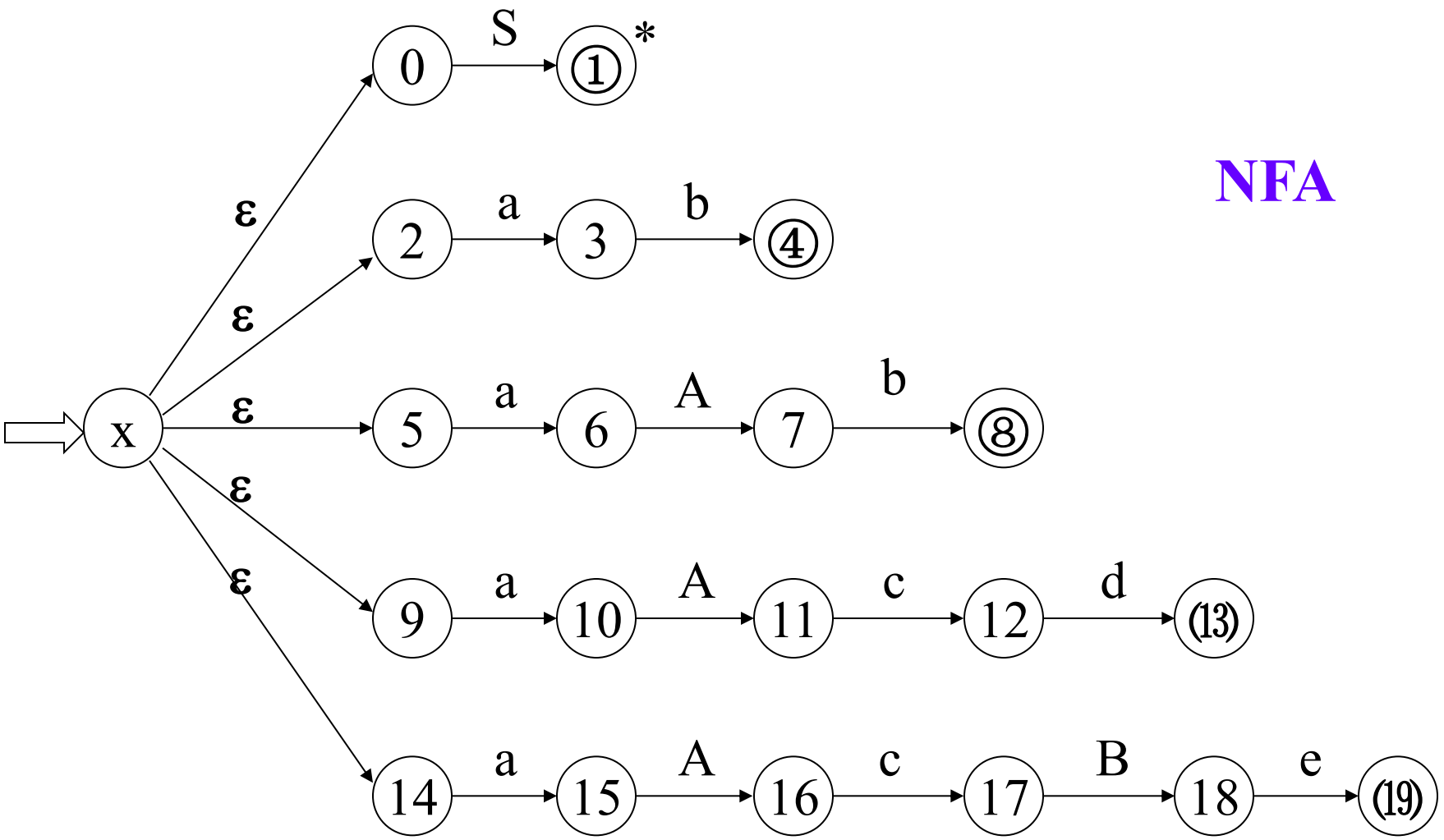
a,aA,aAc,aAcB,aAcBe

S

$S \Rightarrow aAcBe \Rightarrow aAcde \Rightarrow aAbcde \Rightarrow abbcde$

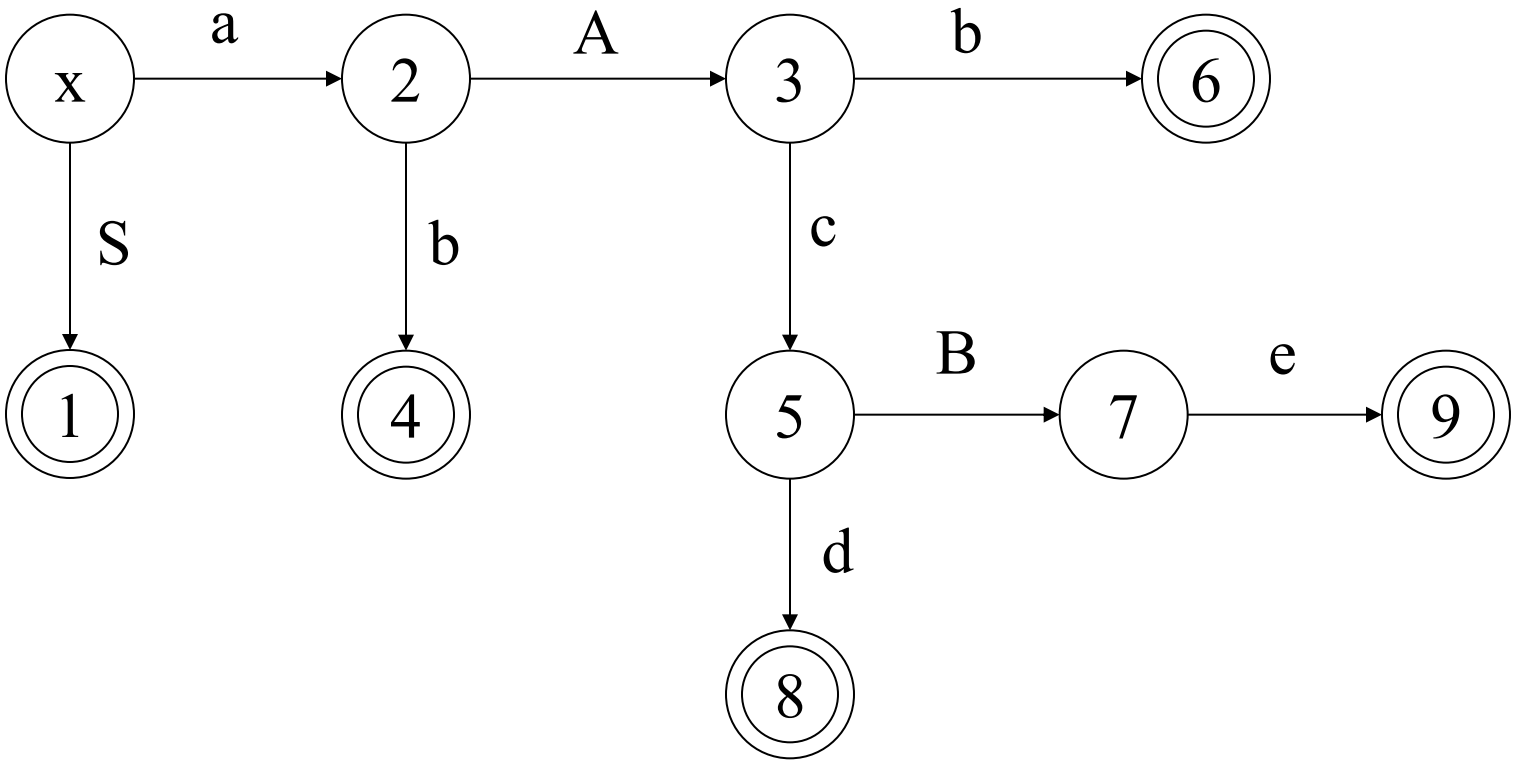
识别活前缀的NFA




NFA
x
ε
ε
ε
ε
ε
0
S
①*
2
a
3
b
④
5
a
6
A
7
b
⑧
9
a
10
A
11
c
12
d
(13)
14
a
15
A
16
c
17
B
18
e
(19)

$S' \Rightarrow S \Rightarrow aAcBe \Rightarrow aAcde \Rightarrow aAbcde \Rightarrow^{i} ab^{i}bcde$

DFA



活前缀及可归前缀的计算

定义(非终结符的左文)

$LC(A)=\{\beta \mid S' \overset{*}{\underset{R}{\Rightarrow}} \beta A\omega, \beta \in V^*, \omega \in V_t^*\}$,

对拓广文法的开始符号S'：

$$LC(S')=\{\varepsilon\}$$

若$B \to \gamma A\delta$**，则：**$LC(A) \supseteq LC(B).\{\gamma\}$

G[S]: (0) S'→S (1) S →a A c B e

(2)A →b (3) A →Ab (4)B →d

每个非终结符的左文方程组

LC(S')={ε}

LC(S)=LC(S').{ε}

LC(A)=LC(S).{a}∪ LC(A){ε}

LC(B)=LC(S).{aAc}

化简为：

[S']= ε

[S]=[S']

[A]=[S]a+[A]

[B]=[S]aAc

用代入法求解

[S']= ε

[S]= ε

[A]=a+[A]

[B]=aAc

令 ∑={[S'], [S], [A], [B], a, A, c}

则方程两边都是∑上的正规式

而[A]=a+[A] 即为 [A]=a | [A] 由正规式所表示的正规集

得：[A]=a

G[S]: (0) S'→S (1) S →a A c B e

(2)A →b (3) A →Ab (4)B →d

定义(产生式的LR(0)左文)

$LR(0)C(A\to\alpha)=\{\gamma\mid \gamma=\beta\alpha$且$S'\overset{*}{\underset{R}{\Rightarrow}}\beta A\omega\underset{R}{\Rightarrow}\beta\alpha\omega, \omega\in V_t^*\}$

推论：LR(0)C(A →α)=LC(A).{α}

则有：

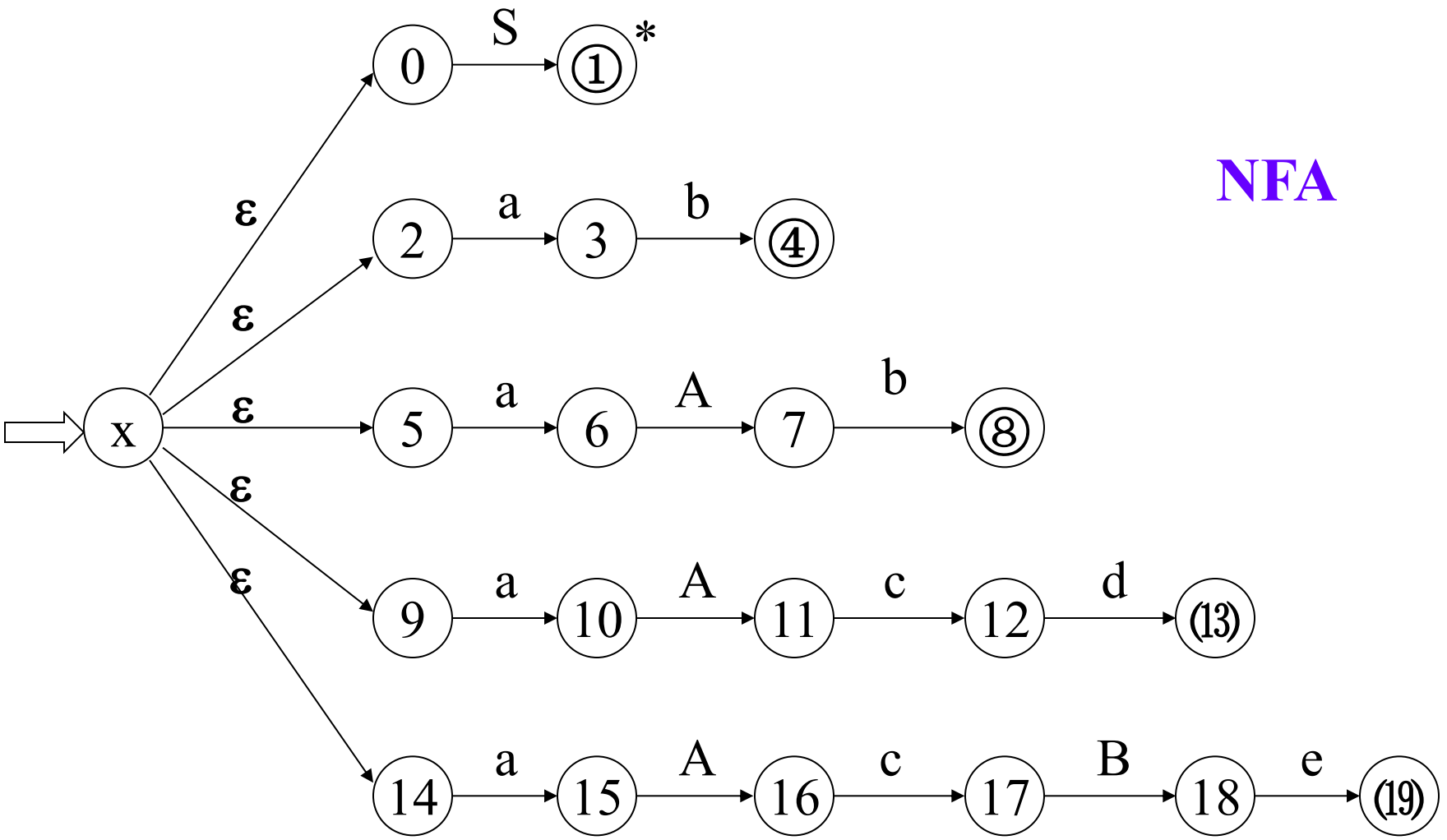
LR(0)C(S'→S)=S

LR(0)C(S→aAcBe)=aAcBe

LR(0)C(A→b)=ab

LR(0)C(A→Ab)=aAb

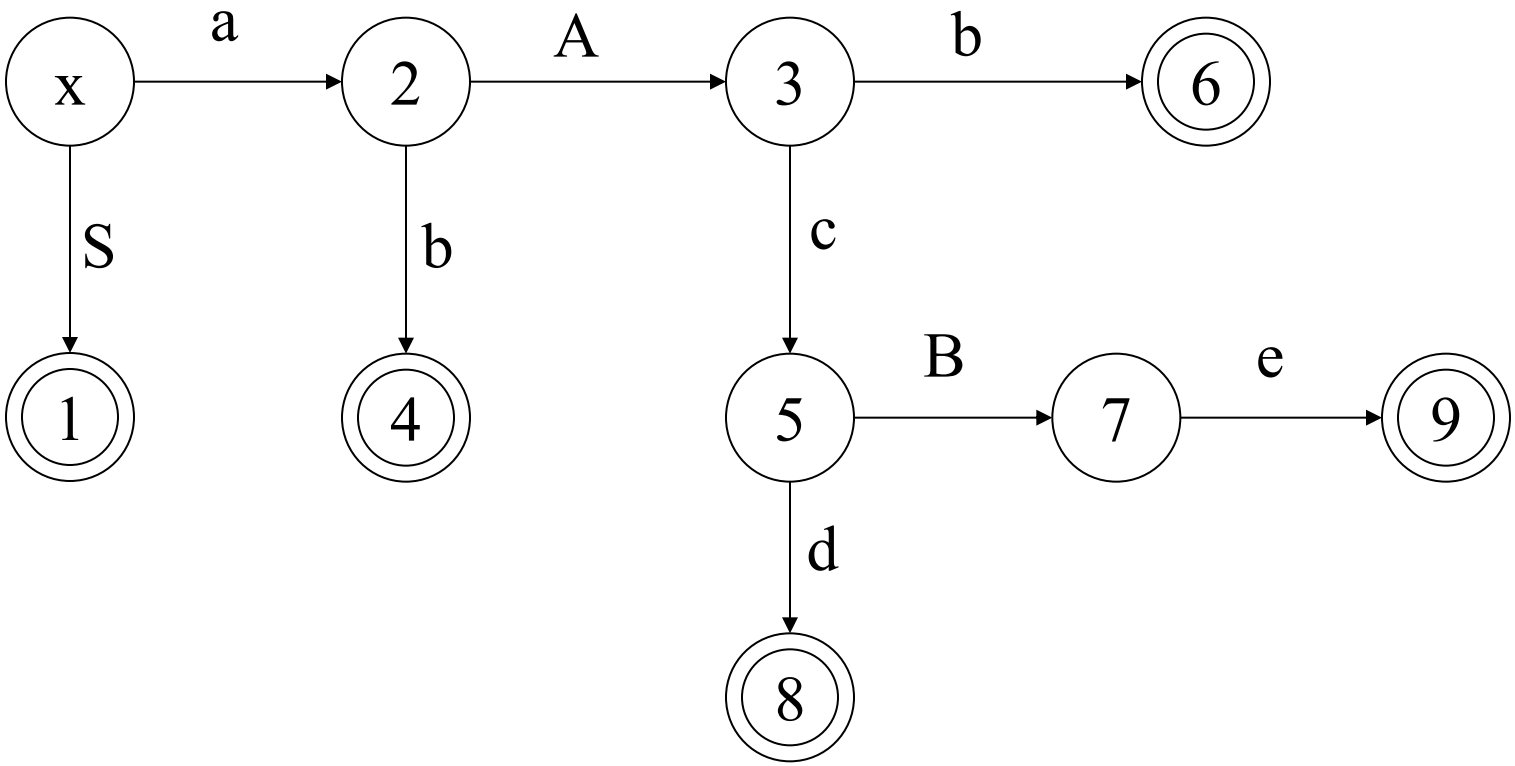
LR(0)C(B→d)=aAcd ($\Sigma=Vn\cup Vt$)上的正规式



NFA

Forward(DFA)

DFA



LR(0)项目

构造LR(0)项目

LR(0)项目或配置（*item* or *configuration*）

---在右端某一位置有圆点的文法G的产生式

给定**A→ xyz**

A→.xyz

A→x.yz

A→xy.z

A→xyz.

比如**S→aAd**

S→.aAd S→a.Ad S→aA.d S→aAd.

活前缀、句柄 、LR(0)项目

为刻划分析过程中文法G的每一个产生式的右部符号已有多大一部分被识别（出现在栈顶）的情况，采用标有圆点的产生式来指示位置：

$A \rightarrow \beta.$　刻画产生式$A \rightarrow \beta$的右部β已出现在栈顶

$A \rightarrow \beta_1.\beta_2$ 刻画$A \rightarrow \beta_1\beta_2$的右部子串$\beta_1$已出现在栈顶，期待从输入串中看到$\beta_2$推出的符号

$A \rightarrow .\beta$　刻画没有句柄的任何符号在栈顶，此时期望$A \rightarrow \beta$的右部所推出的符号串

对于$A \rightarrow \varepsilon$的LR(0)项目，只有$A \rightarrow .$

LR(0)项目与含句柄活前缀αβ的β对应

给定文法G=(Vn,Vt, P, S), 若有规范推导:

$S' \Rightarrow_R^* \alpha A w \Rightarrow_R \alpha\beta w$, **γ是αβ的前缀,**

则称γ是文法G的活前缀.

1. 活前缀已含有句柄的全部符号，表明产生式A→β的右部β已出现在栈顶（**A→β.**）
2. 活前缀只含句柄的一部分符号，表明A→$\beta_1\beta_2$的右部子串β_1已出现在栈顶，期待从输入串中看到β_2推出的符号（**A→β_1.β_2**）
3. 活前缀不含有句柄的任何符号，此时期望A→β的右部所推出的符号串（**A→.β**）

含句柄活前缀αβ的前缀α在LR(0)项目中的（隐含）对应部分

考虑右句型：$S' \Rightarrow^* \alpha Aw \Rightarrow \alpha\beta w$

若有项目k:

$A \rightarrow .\beta$， $A \neq S'$，

则一定存在项目i：

$X \rightarrow \gamma . A\delta$，

那么，

$\alpha = \alpha'\gamma$

例如：$S \rightarrow aAc.Be$ $B \rightarrow .d$，那么，

$\alpha = \alpha'aAc$，$\gamma = aAc$

由LR(0)项目构造识别活前缀的NFA

G[S]: (0) S'→S (1) S →a A c B e

(2) A →b (3) A →Ab (4)B →d

文法的项目为：

1. S'→.S 2. S'→S.

3. S →.aAcBe 4. S →a.AcBe 5. S →aA.cBe

6. S →aAc.Be 7. S →aAcB.e 8. S →aAcBe.

9. A →.b 10. A →b.

11. A →.Ab 12. A →A.b 13. A →Ab.

14. B →.d 15. B →d.

项目就是状态！

项目(状态)之间的转换

转换方法如下：

若有项目i： $X \rightarrow X_1 X_2 \ldots X_{i-1} \cdot X_i \ldots X_n$

项目j： $X \rightarrow X_1 X_2 \ldots X_{i-1} X_i \cdot X_{i+1} \ldots X_n$

则从状态i到状态j连一条标记为X_i的箭弧

若有项目i ： $X \rightarrow \gamma \cdot A\, \delta$

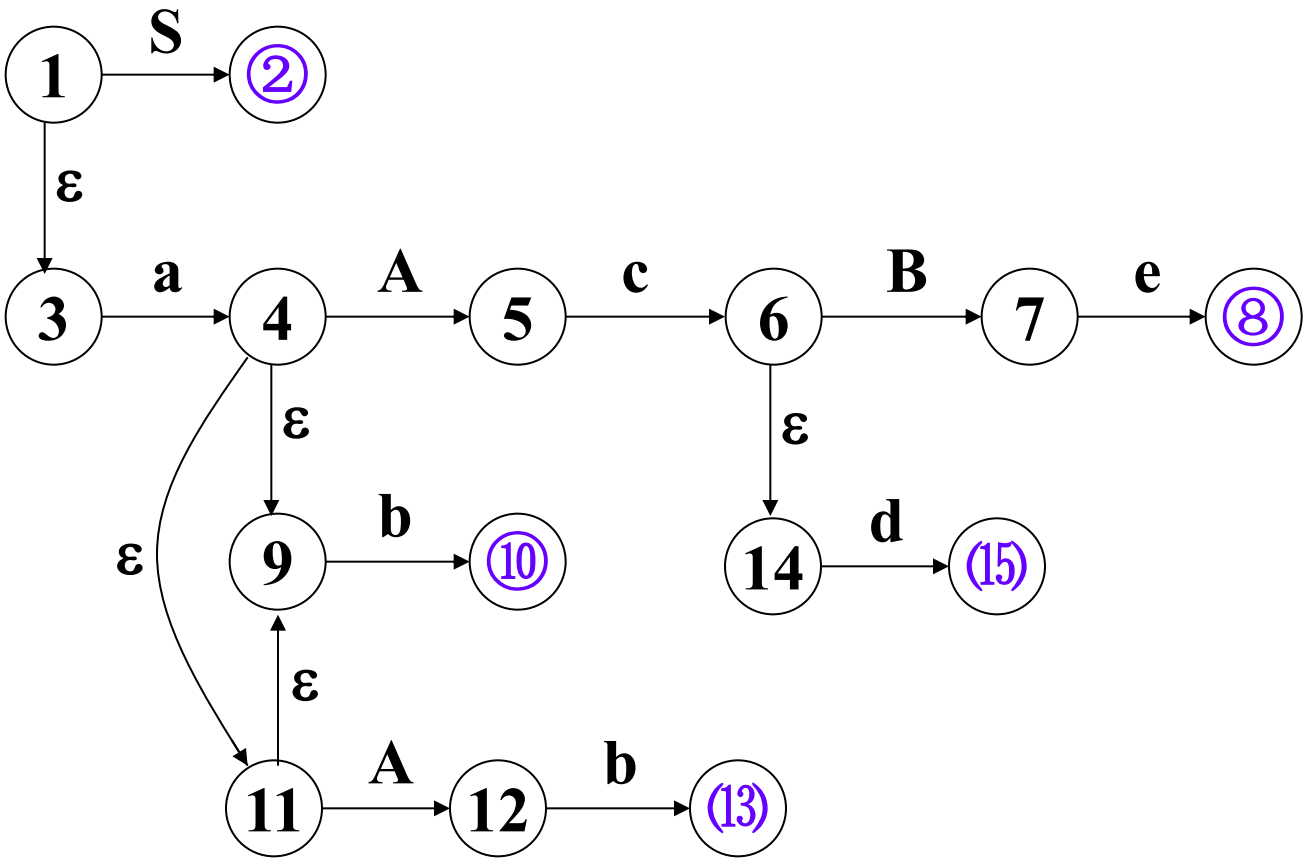
项目k： $A \rightarrow \cdot \beta$

则从状态i画一条标记为ε的箭弧到状态k

点在最右边的项目为句柄识别态，即NFA的终态

NFA?!

再将NFA转换成DFA？



Backward(NFA)

Forward(DFA)

LR(0)项目集的规范族

LR(0) 项目集的闭包CLOSURE

若当前处于$A \rightarrow \alpha \cdot B\beta$ 刻画的情况，则期望移进First(B)的某些符号；

如有产生式 $B \rightarrow \gamma$，那么$B \rightarrow \cdot \gamma$ 这个项目便是刻画期望移进 First(B)中某个符号的情况；

所以，

$A \rightarrow \alpha \cdot B\beta$

$B \rightarrow \cdot \gamma$

这两个项目对应移进-归约分析中的同一个状态，构成一个**配置集（项目集）**；

也就是，对每个配置集，**分析表**中将有一个状态与之对应

LR(0)项目集闭包的构造

LR(0)项目集的闭包CLOSURE

```
function  CLOSURE (I); /* I 是项目集*/
{  J:= I;
   repeat
      for  J 中每个项目A →α .B β 和产生式，  B→ γ ，
      若 B→ .γ 不在J中 do
         将 B→ .γ 加到J中
   until  再也没有项目可以再加到J中
   return  J
};
```

转换函数GO (I, X)

GO 函数

GO (I, X) == CLOSURE(J) ;

其中，I: 项目集，X: 文法符号，

J={任何形如 $A\rightarrow\alpha\ X.\ \beta$ 的项目$|A\rightarrow\alpha\ .X\beta\in I$}

LR(0)项目集规范族的构造

计算LR(0)项目集的规范族

C={ I_0 , I_1 , … , I_n }

```
procedure Itemsets(G');
begin
    C := { CLOSURE ({S'→.S}) }
    repeat
        for C中每一项目集I和每一文法符号x do
            if GO(I, x) 非空且不属于C
            then 把 GO(I, x) 放入C中
        endfor
    until C不再增大
end;
```

LR(0)项目集的规范族构成识别一个文法的活前缀的DFA的状态的全体

文法G:

(0) S'→E (1) E→aA (2) E→bB

(3) A→cA (4) A→d (5) B→cB

(6) B→d

LR(0) 项目集规范族(识别G的活前缀的DFA):

I_0: S'→.E E →.aA E →.bB

I_1: S'→E.

I_2: E →a.A A →.cA A →.d

I_3: E→b.B
B→.cB
B→.d

I_4: A→c.A
A→.cA
A→.d

I_5: B→c.B
B→.cB
B→.d

I_6: E→aA.

I_7: E→bB.

I_8: A→cA.

I_9: B→cB.

I_{10}: A→d.

I_{11}: B→d.

LR(0)分析表的构造

假定C={I_0, I_1, …, I_n}，令每个项目集I_k的下标k为分析器的一个状态，则G'的LR(0)分析表含有状态0，1，…，n。

令含有项目S'→.S的I_k的下标k为初态。

ACTION和GOTO可按如下方法构造：

若项目A→α.aβ属于I_k且GO (I_k, a)= I_j, a为任意终结符，则置ACTION[k, a]为“把状态j和符号a移进栈”，简记为“sj”；

若项目A→α.属于I_k, 则对任何终结符a, 置ACTION[k, a]为“用产生式A→α进行归约”，简记为“rj”；其中假定A→α为文法G'的第j个产生式；

若项目S'→S.属于I_k, 则置ACTION[k, #]为“接受”，简记为“acc”；

若GO (I_k, A)= I_j, A为非终结符，则置GOTO(k, A)=j；

分析表中凡不能用规则1至4填入信息的空白表项均置上“出错标志”。

按上述算法构造的含有ACTION和GOTO两部分的分析表，如果每个表项不含多重定义，则称它为文法G的一张LR(0)分析表。

具有LR(0)分析表的文法G称为一个LR(0)文法。

LR(0)文法是无二义的。

文法G:(0) S'→E (1) E→aA (2) E→bB

(3) A→cA (4) A→d (5) B→cB (6) B→d

	ACTION						GOTO		
	a	c	b	d	#		E	A	B
0	S2		S3				1		
1					acc				
2		S4		S10				6	
3		S5		S11					7
4		S4		S10				8	
5		S5		S11					9
6	r1	r1	r1	r1	r1				
7	r2	r2	r2	r2	r2				
8	r3	r3	r3	r3	r3				
9	r5	r5	r5	r5	r5				
10	r4	r4	r4	r4	r4				
11	r6	r6	r6	r6	r6				

LR(0)项目的分类

根据圆点所在的位置和圆点后是终结符还是非终结符或为空把项目分为以下几种：

移进项目，形如 $A \rightarrow \alpha \cdot a\beta$　　a 是终结符，$\alpha, \beta \in V^*$ 下同

待约项目，形如 $A \rightarrow \alpha \cdot B\beta$

归约项目，形如 $A \rightarrow \alpha \cdot$

接受项目，形如 $S' \rightarrow S \cdot$

$A \rightarrow \varepsilon$ 的LR(0)项目只有$A \rightarrow \cdot$，是归约项目

例7.1 G[S]为:

S → a A c B e

A → b

A → Ab

B → d

1) 构造识别活前缀的DFA;

2) 构造它的LR(0)分析表;

3) 分别给出对输入符号串abbcde和abbce的LR(0)分析步骤。

G[S]拓广为:

(0) S′→ S

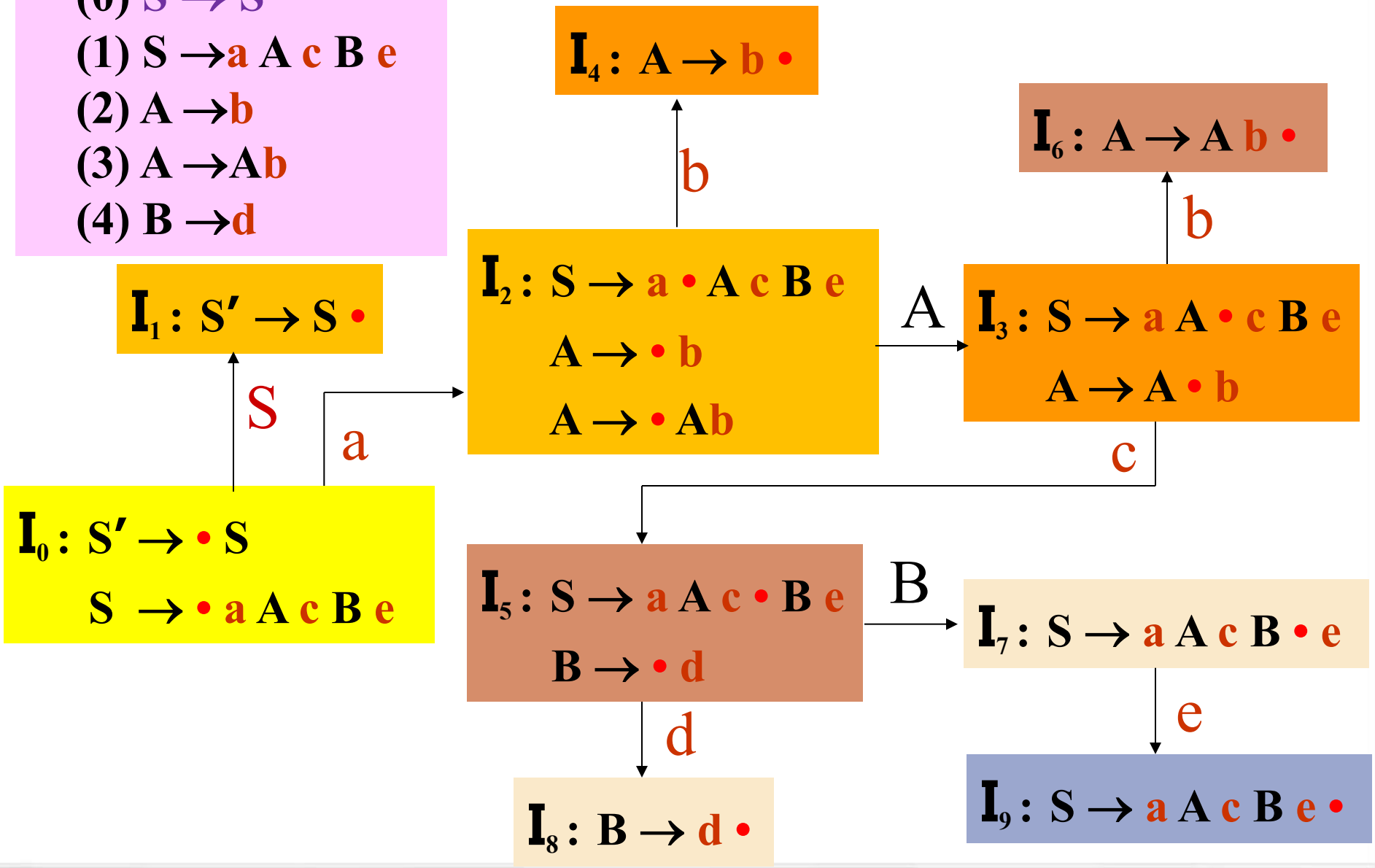
(1) S →a A c B e

(2) A →b

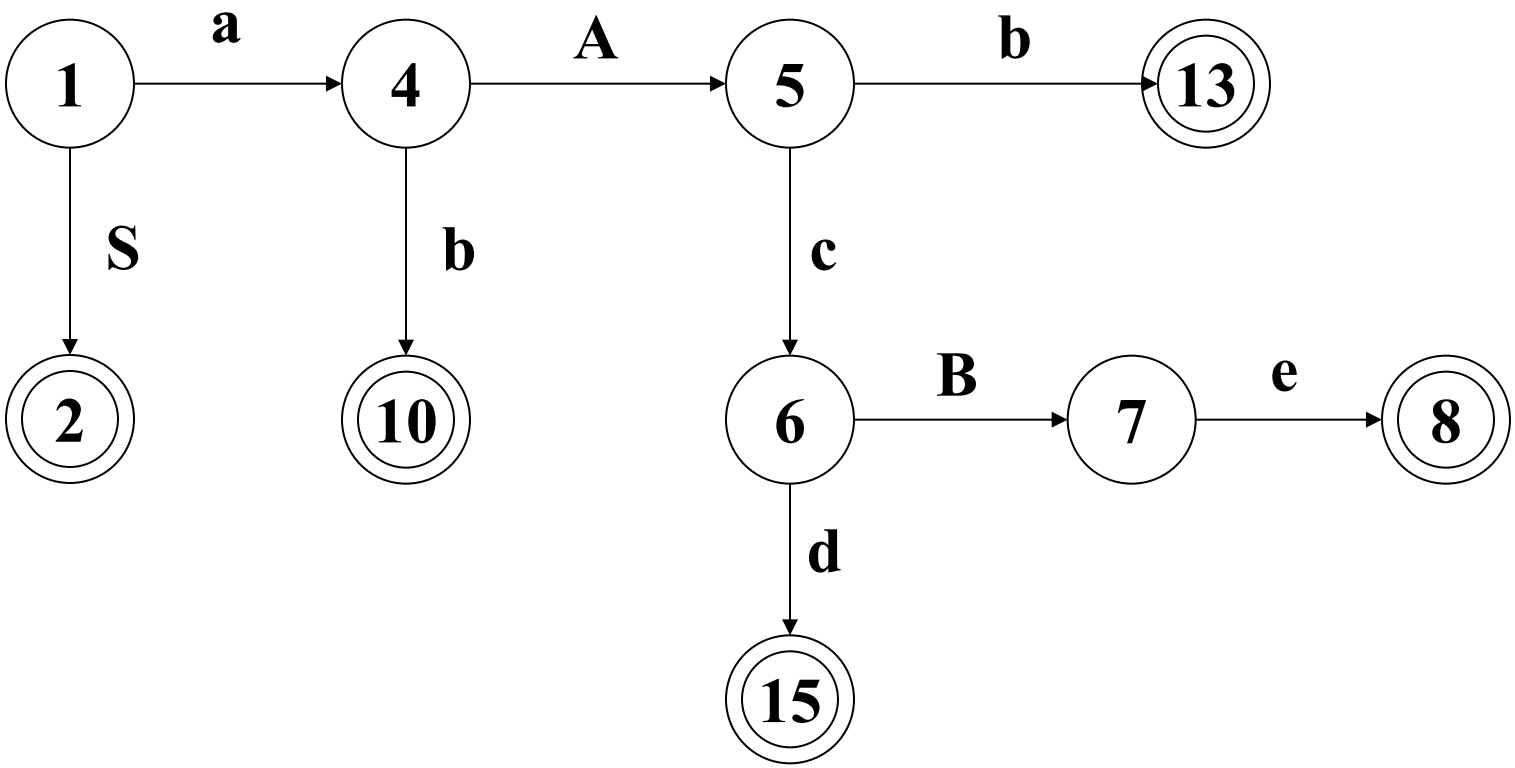
(3) A →Ab

(4) B →d

G[L]= ab^+cde



DFA



Backward(NFA)

例7.1 G[S]的LR(0)分析表

	ACTION						GOTO		
	a	c	e	b	d	#	S	A	B
0	S_2						1		
1						acc			
2				S_4				3	
3		S_5		S_6					
4	r_2	r_2	r_2	r_2	r_2	r_2			
5					S_8				7
6	r_3	r_3	r_3	r_3	r_3	r_3			
7			S_9						
8	r_4	r_4	r_4	r_4	r_4	r_4			
9	r_1	r_1	r_1	r_1	r_1	r_1			

对输入串abbcde#的分析过程

Step	states.	Syms.	The rest of input	action	goto
1	0	#	abbcde#	s2	
2	02	#a	bbcde#	s4	
3	024	#ab	bcde#	r2	3
4	023	#aA	bcde#	s6	
5	0236	#aAb	cde#	r3	3
6	023	#aA	cde#	s5	
7	0235	#aAc	de#	s8	
8	02358	#aAcd	e#	r4	7
9	02357	#aAcB	e#	s9	
10	023579	#aAcBe	#	r1	1
11	01	#S	#	acc	

对输入串abbce#的分析过程

Step	States	Syms.	The rest of input	action	goto
1	0	#	abbce#	s2	
2	02	#a	bbce#	s4	
3	024	#ab	bce#	r2	3
4	023	#aA	bce#	s6	
5	0236	#aAb	ce#	r3	3
6	023	#aA	ce#	s5	
7	0235	#aAc	e#	出错	

说明abbce#不是例7.1的文法G[S]的句子

The end of part 1