第6章 LR分析

第五章我们学过自底向上分析方法是一种移进-归约过程，当分析的栈顶符号串形成句柄时就采取归约动作，因而自底向上分析法的关键问题是在分析过程中如何确定句柄。

LR分析法与第六章介绍的运算符优先函数一样，LR方法也是通过求句柄逐步归约进行语法分析。在运算符优先函数中，句柄是通过运算符的优先关系而求得，LR方法中句柄是通过求可归前缀而求得。

LR分析概述

- LR（k）分析是根据当前分析栈中的符号串和向右顺序查看输入串的k(k≥0)个符号就可以唯一确定分析的动作是移进还是归约以及用哪个产生式归约。
- 从左到右扫描(L)自底向上进行归约(R)(是规范归约）

主要内容

一、LR分析概述（基本构造原理与方法）

二、LR(0)分析

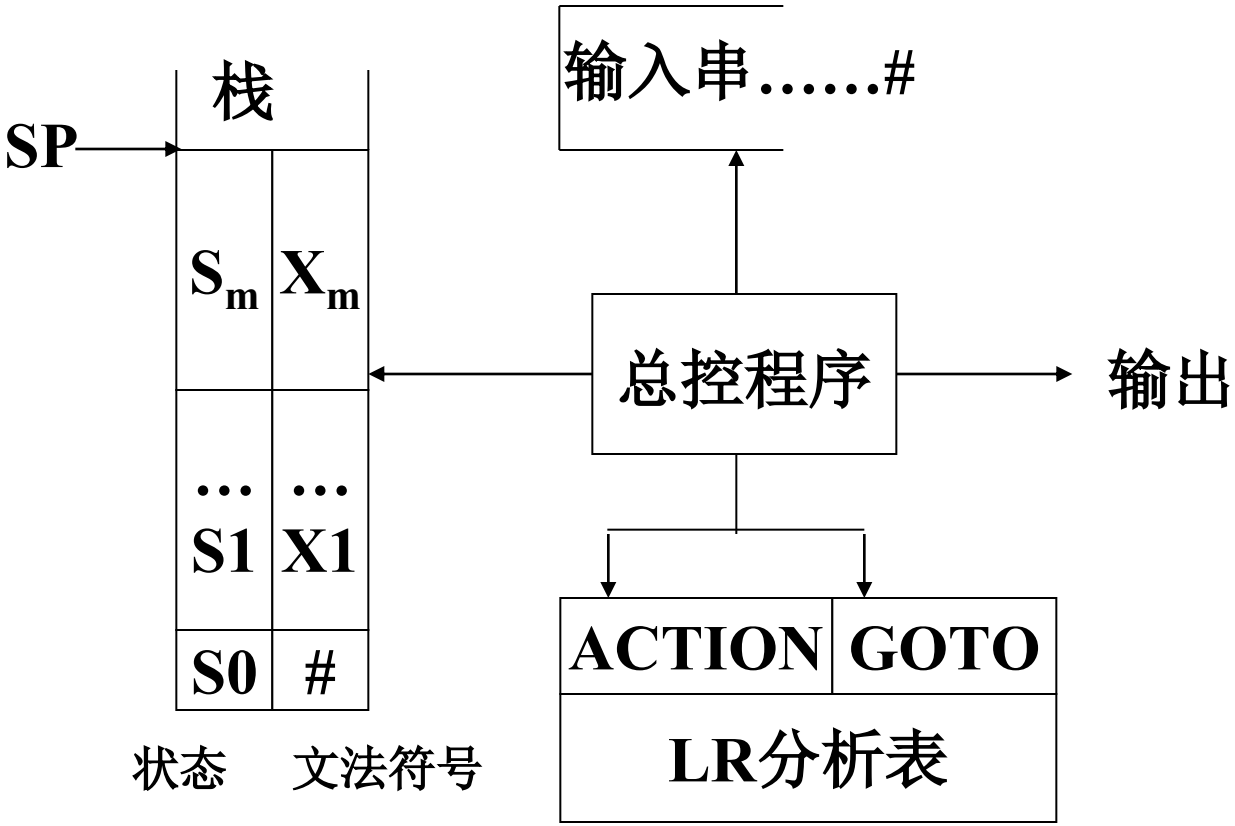
三、SLR(1)分析

四、LR(1)分析

五、LALR (1)分析

六、二义性文法在LR分析中的应用

LR分析器模型



LR分析方法的逻辑结构

逻辑上说，一个LR分析器由3个部分组成：

(1) 总控程序，也可以称为驱动程序。对所有的LR分析器总控程序都是相同的。

(2) 分析表或分析函数，不同的文法分析表将不同，同一个文法采用的LR分析器不同时，分析表也不同，分析表又可分为动作表（ACTION）和状态转换（GOTO）表两个部分，它们都可用二维数组表示。

(3) 分析栈，包括文法符号栈和相应的状态栈，它们均是先进后出栈。

分析器的动作就是由栈顶状态和当前输入符号所决定。

总控程序根据不同的分析表来决定其下一步的处理动作，分析表是根据具体的文法按某种规则构造出来的。**LR方法**是根据具体文法的分析表对输入串进行分析处理。

LR分析过程的思想是在总控程序的控制下，从左到右扫描输入符号串，根据分析栈中的状态和文法及当前输入符号，按分析表完成相应的分析工作。

分析表的组成：

(1) 分析动作表Action

状态 \ 符号	a_1	a_2	…	a_t
S_0	action[S_0, a_1]	action[S_0, a_2]	…	action[S_0, a_t]
S_1	action[S_1, a_1]	action[S_1, a_2]	…	action[S_1, a_t]
…	…	…	…	…
S_n	action[S_n, a_1]	action[S_n, a_2]	…	action[S_n, a_t]

表中action[S_i,a_j]为二维数组，指出当前栈顶为状态S_i，输入符号为a_j是所执行的动作。其动作有四种可能，分别为移进(S)、归约(r)、接受(acc)、出错(error)。

(2) 状态转换表goto

状态 \ 符号	X_1	X_2	…	x_t
S_0	goto[S_0 , x_1]	goto[S_0 , x_2]	…	goto[S_0 , x_t]
S_1	goto[S_1 ,x_1]	goto[S_1 , x_2]	…	goto[S_1 , x_t]
…	…	…	…	…
S_n	goto[S_n , x_1]	goto[S_n , x_2]	…	goto[S_n , x_t]

表中goto[S_i,x_j]指出栈顶状态为S_i，文法符号为x_j时应转到的下一状态。

LR分析过程：

(1)LR分析步骤：

(a) 将初始状态0和句子的左界符#分别进分析栈。

(b) 根据栈顶状态和当前输入符号查动作表，进 行如下的工作。

※ 移进：当前输入符号进符号栈，根据状态转换表新的状态进状态栈，继续扫描，从而下一输入符号变成当前输入符号。

※ 归约：(1)按某个产生式进行归约，若产生式的右端长度为n，则符号栈顶和状态顶n个元素同时相应退栈。(2)归约后的文法符号进符号栈，(3)查状态转换表，新的状态进状态栈。

※ 接受：分析成功，终止分析。

※ 出错：报告出错信息。

(2) 具体分析过程：

举例说明具体分析过程：

设文法为G[S] （假定已存在LR分析表）

(1) S →aAcBe

(2) A →b

(3) A →Ab

(4) B →d

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。

- 自底向上分析法的关键问题是在分析过程中如何确定句柄。
- LR方法中的句柄是通过求可归前缀而求得。

活前缀与可归前缀

例：文法G[S]为：

S →aAcBe

A →b

A →Ab

B →d

为产生式加序号变为：

S →aAcBe[1]

A →b[2]

A →Ab[3]

B →d[4]

对于输入串abbcde句子的最右推导（规范推导）如下：

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

对它的逆过程最左归约(规范归约)为：

ab[2]b[3]cd[4]e[1]

⇐aAb[3]cd[4]e[1]

⇐aAcd[4]e[1]

⇐aAcBe[1]

⇐S

为产生式加序号变为：

S →aAcBe[1]

A →b[2]

A →Ab[3]

B →d[4]

用哪个产生式继续归约仅取决于当前句型的前部。
我们把每次采取归约动作前的符号串部分称为可归前缀。
LR分析的关键就是识别何时到达可归前缀。

在规范句型中形成可归前缀之前包括可归前缀的所有前缀称为活前缀。活前缀为一个或若干规范句型的前缀。在规范归约过程中的任何时刻只要已分析过的部分即在符号栈中的符号串均为规范句型的活前缀，则表明输入串的已被分析过的部分是某规范句型的一个正确部分。

例如：有下面规范句型的前缀：

ε，a，ab

ε，a，aA，aAb

ε，a，aA，aAc，aAcd

ε，a，aA，aAc，aAcB，aAcBe

左部均为活前缀，其中，红色部分为可归前缀

活前缀的定义

- G=(Vn,Vt,P,S),若有$S' \overset{*}{\underset{R}{\Rightarrow}} \alpha A\omega \underset{R}{\Rightarrow} \alpha\beta\omega$, $A \to \beta$ （β是句柄）

γ是αβ的前缀,则称是文法G的活前缀。其中S'是对原文法扩充(S'→S)增加的非终结符。

- αβ是含句柄的活前缀，并且句柄是αβ的后端，则称αβ是可归前缀或可规范前缀。
- 在LR分析过程中，实际上是把αβ的前缀（即文法G的活前缀）列出放在符号栈中，一旦在栈中出现αβ（形成可归前缀），即句柄已经形成，则用产生式$A \to \beta$进行归约。

识别活前缀的有限自动机

在LR方法实际分析过程中并不是直接分析文法符号栈中的符号是否形成句柄，但它给我们一个启示：

我们可以把终结符号和非终结符号都看成一个有限自动机的输入符号，每把一个符号进栈时看成已识别了该符号，而状态进行转换。当识别到可归前缀时，相当在栈中形成句柄，则认为到达了识别句柄的终态。

例：文法G[S]的拓广文法为：

S’ →S[0]

S →aAcBe[1]

A →b[2]

A →Ab[3]

B →d[4]

句子abbcde的可归前缀为：

S [0]

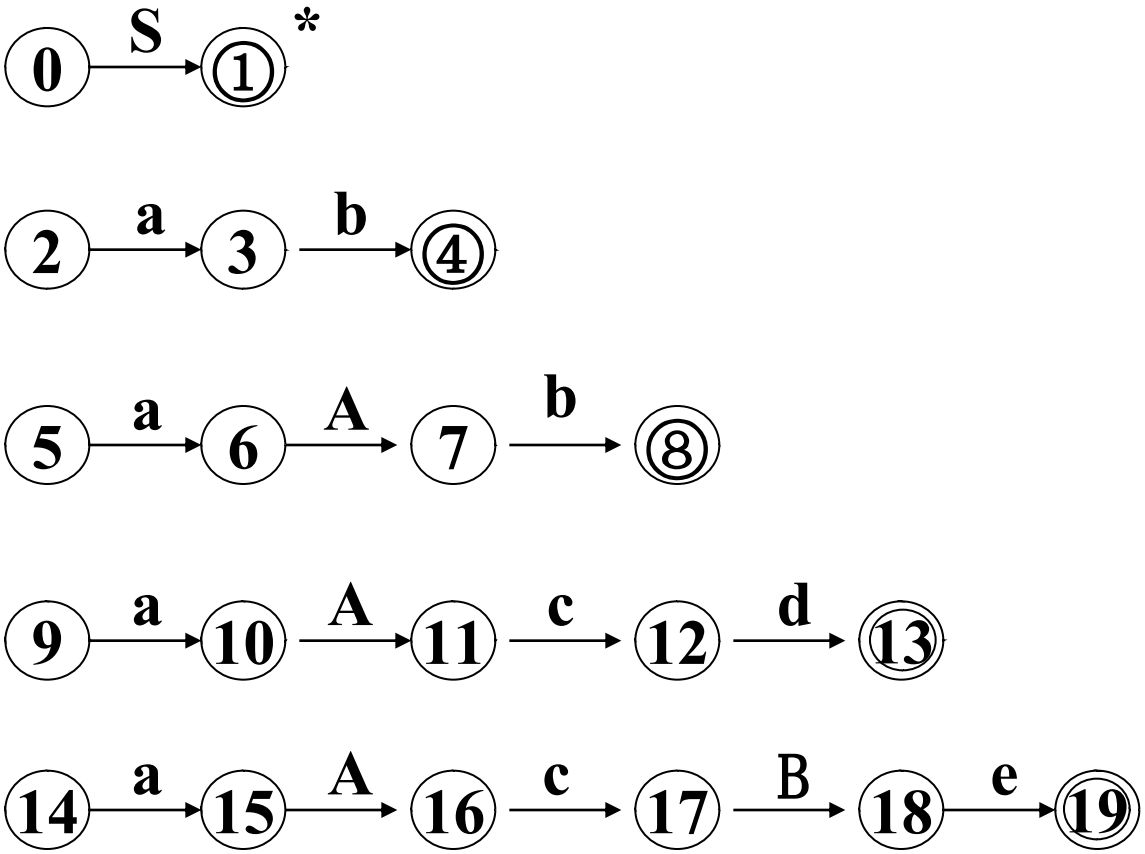
ab[2]

aAb[3]

aAcd[4]

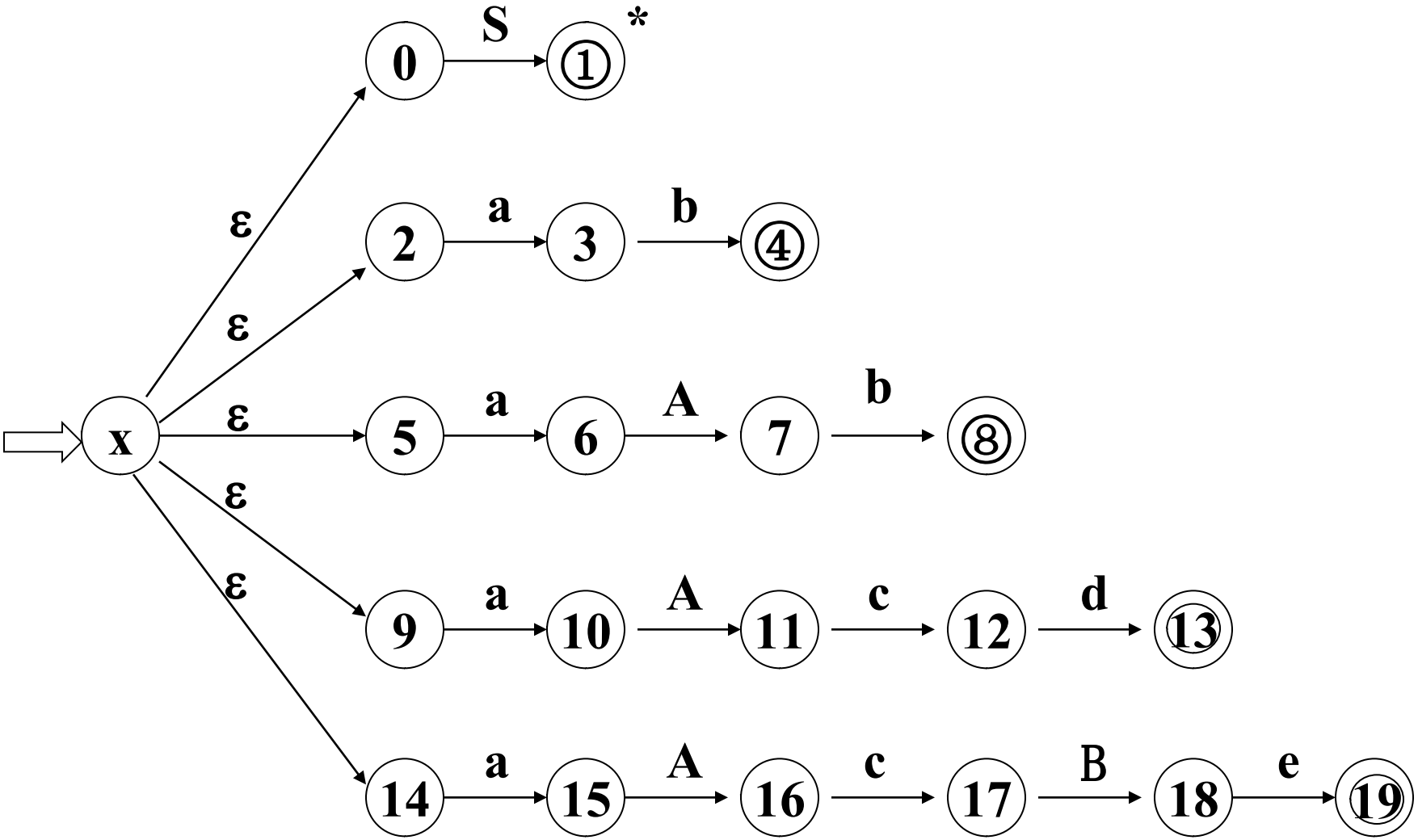
aAcBe[1]

识别活前缀及可归前缀的有限自动机

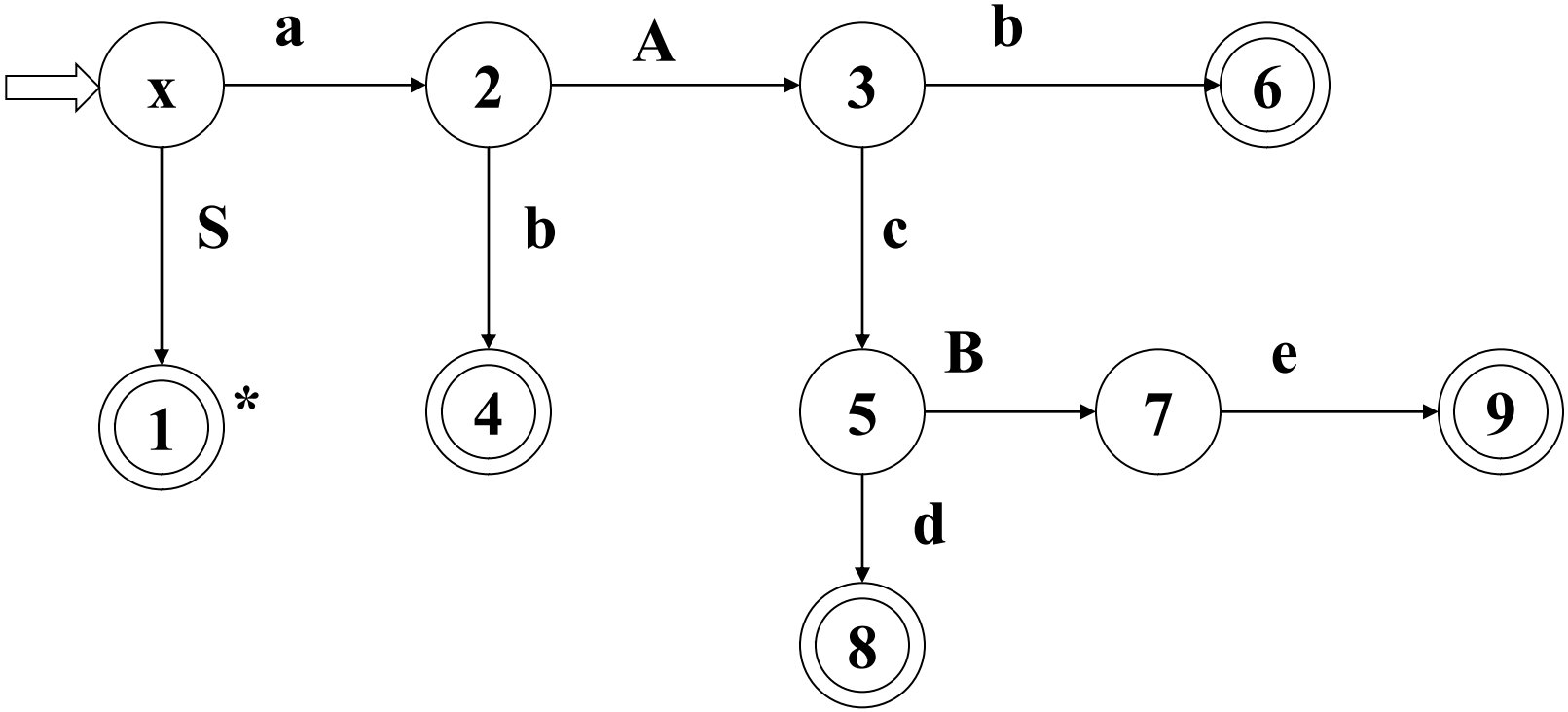


每一个终态都是句柄识别态，用ⓘ表示，仅有带“*”的状态既为句柄识别态又是句子识别态，句子识别态仅有唯一的一个。

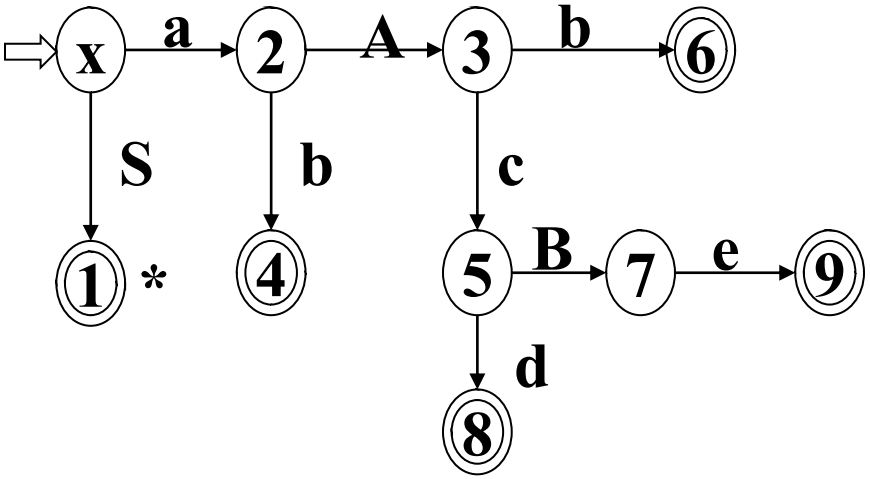
如果加一个开始状态X，并用 ε 弧和每个识别可归前缀的有限自动机连接，变为下图



确定化后得到下图：



比较以下两图



识别活前缀的确定有限自动机

用有限自动机识别时每当识别完句柄，则状态回退到句柄串长度的状态数。

LR分析表

对于任何一个复杂的文法，它的可归前缀不是很简单就能计算出来。下面给出求出一个任给的上下文无关文法所有活前缀和可归前缀的有限自动机的确定方法。

LC(A)的定义

- 启示: LR(0)分析使用的信息之一是句柄左部的内容.
- 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’:

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

若有$B \rightarrow \gamma A\delta$ 则:$LC(A) \supseteq LC(B).\{\gamma\}$因为:

$S' \overset{*}{\underset{R}{\Rightarrow}} \alpha B\omega \Rightarrow \alpha\gamma A\ \delta\ \omega$

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}

简写为：

- [S']= ε
- [S]=[S']
- [A]=[S]a+[A]
- [B]=[S]aAc

用代入法求解得：

- [S']= ε
- [S]= ε
- [A]=a+[A]
- [B]=aAc

令 ∑={[S']，[S]，[A]， [B]，a，A,c}

则方程两边都是∑上的正口式

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)CONTEXT(A→α)={γ| γ =βα且$S' \overset{*}{\underset{R}{\Rightarrow}} \beta A\omega \underset{R}{\Rightarrow} \beta\alpha\omega$, $\omega \in V_t^*$}

有推论：LR(0)C(A →α)=LC(A).{α}

- **[S']= ε**
- **[S]= ε**
- **[A]=a**
- **[B]=aAc**

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 **Σ (=Vn∪Vt)上的正规式**

再举一例：

G’[S’]: (0) S’→E (1) E →a A (2)E →bB (3) A →cA (4)A →d (5) B →cB (6) B →d

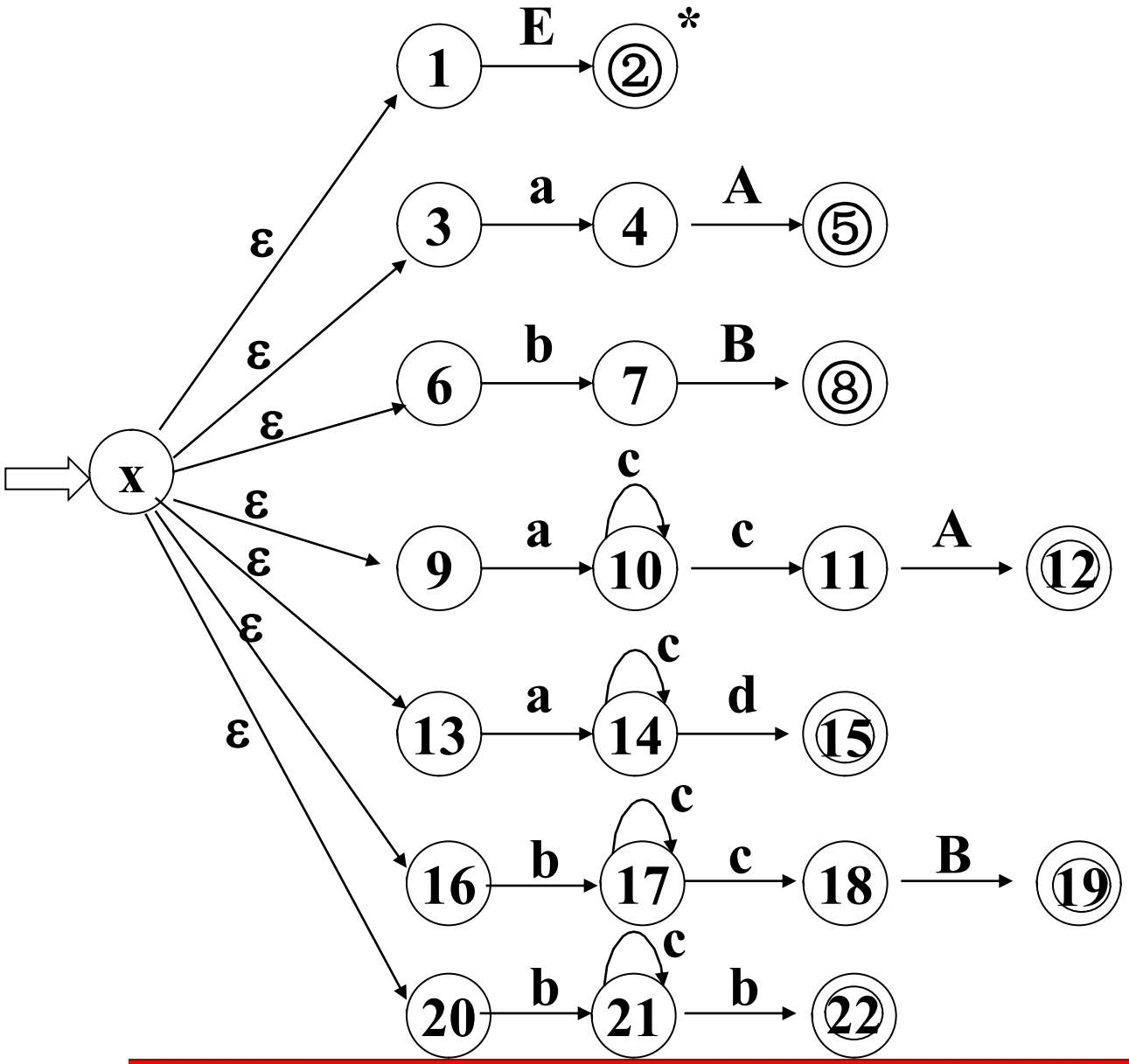
不包括句柄在内的活前缀方程组为：

- **LC(S’)={ε}**
- **LC(S)=LC(S’). ε**
- **LC(A)=LC(E). a | LC(A). c = ac***
- **LC(B)=LC(E). B | LC(B). c = bc***

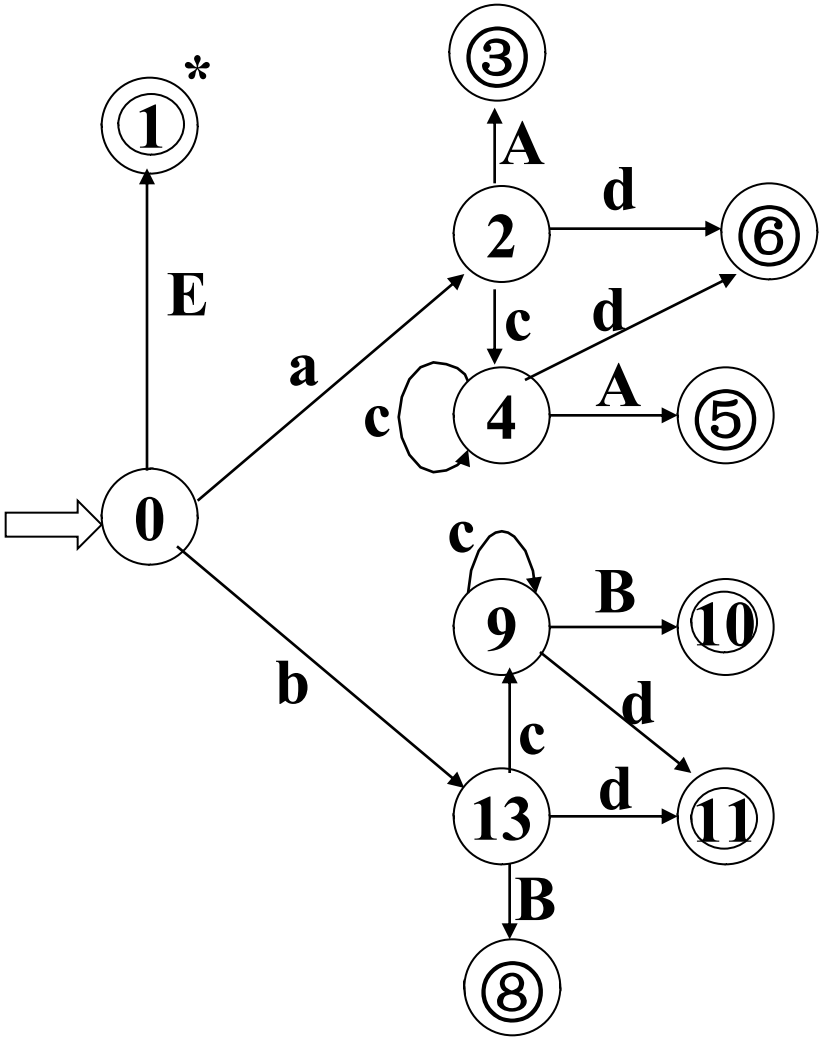
包含句柄的活前缀为：

- **LR(0)C(S’→E)= E**
- **LR(0)C(E→aA)=aA**
- **LR(0)C(E→bB)=bB**
- **LR(0)C(A→cA)=ac*cA**
- **LR(0)C(A→d)=ac*d**
- **LR(0)C(B→cB)=bc*cB**
- **LR(0)C(B→d)=bc*d**

识别可归前缀的不确定有限自动机



识别可归前缀的确定有限自动机



对任何一个上下文无关文法只要构造它的识别可归前缀的有限自动机，就可以构造其对应的分析表。用这种方法构造识别可归前缀的有限自动机从理论角度讲是很严格的，然而，对于一个实用的文法实现起来是很复杂的，下面再介绍一种实用的方法。

活前缀与句柄的关系

G[S]:

若$S \overset{*}{\underset{R}{\Rightarrow}} \alpha A \omega \underset{R}{\Rightarrow} \alpha \beta \omega$　r是$\alpha \beta$的前缀，则

称r是G的一个活前缀

1. 活前缀已含有句柄的全部符号，表明产生式$A \rightarrow \beta$的右部β已出现在栈顶
2. 活前缀只含句柄的一部分符号表明$A \rightarrow \beta_1 \beta_2$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
3. 活前缀不含有句柄的任何符号，此时期望$A \rightarrow \beta$的右部所推出的符号串

活前缀,与句柄 ,与 LR(0)项目

- 为刻划这种分析过程中的文法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，分别用标有圆点的产生式来指示位置。

  $A \rightarrow \beta \bullet$ 刻划产生式$A \rightarrow \beta$的右部β已出现在栈顶

  $A \rightarrow \beta_1 \bullet \beta_2$ 刻划$A \rightarrow \beta_1 \beta_2$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

  $A \rightarrow \bullet \beta$ 刻划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$A \rightarrow \bullet$

项目：文法G的每个产生式(规则)的右部添加一个圆点就构成一个项目。

例:产生式:A→XYZ

项　目: A→• XYZ

A→X• YZ

A→XY• Z

A→XYZ•

产生式:A→ε

项　目: A→.

项目的直观意义:
指明在分析过程中的某一时刻已经归约的部分和等待归约部分。

其中， ε、 X、XY、 XYZ为活前缀，XYZ是可归前缀。

例如：产生式S→aAcBe对应有6个项目。

[0] S→• aAcBe　　[1] S→a• AcBe

[2] S→aA• cBe　　[3] S→aAc• Be

[4] S→aAcB• e　　[5] S→aAcBe •

构造识别活前缀的NFA

文法G'[S']：

S'→E

E→aA | bB

A→cA |d

B→cB | d

把文法的所有产生式的项目都列出，每个项目都为NFA的一个状态。

该文法的项目有：

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

构造识别活前缀的NFA

- 由于S′仅在第一个产生式的左部出现，因此规定项目1为初态，其余每个状态都为活前缀的识别态，圆点在最后的项目为句柄识别态，第一个产生式的句柄识别态为句子识别态。状态之间的转换关系确定方法为如下：
- 若i项目为：$X \rightarrow X_1 X_2 \ldots X_{i-1} \cdot X_i \ldots X_n$

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

  i项目和j项目出于同一个产生式。对应于NFA为状态j的圆点只落后于状态I的圆点一个符号的位置，那么从状态i到状态连一条标记为X_i的箭弧。

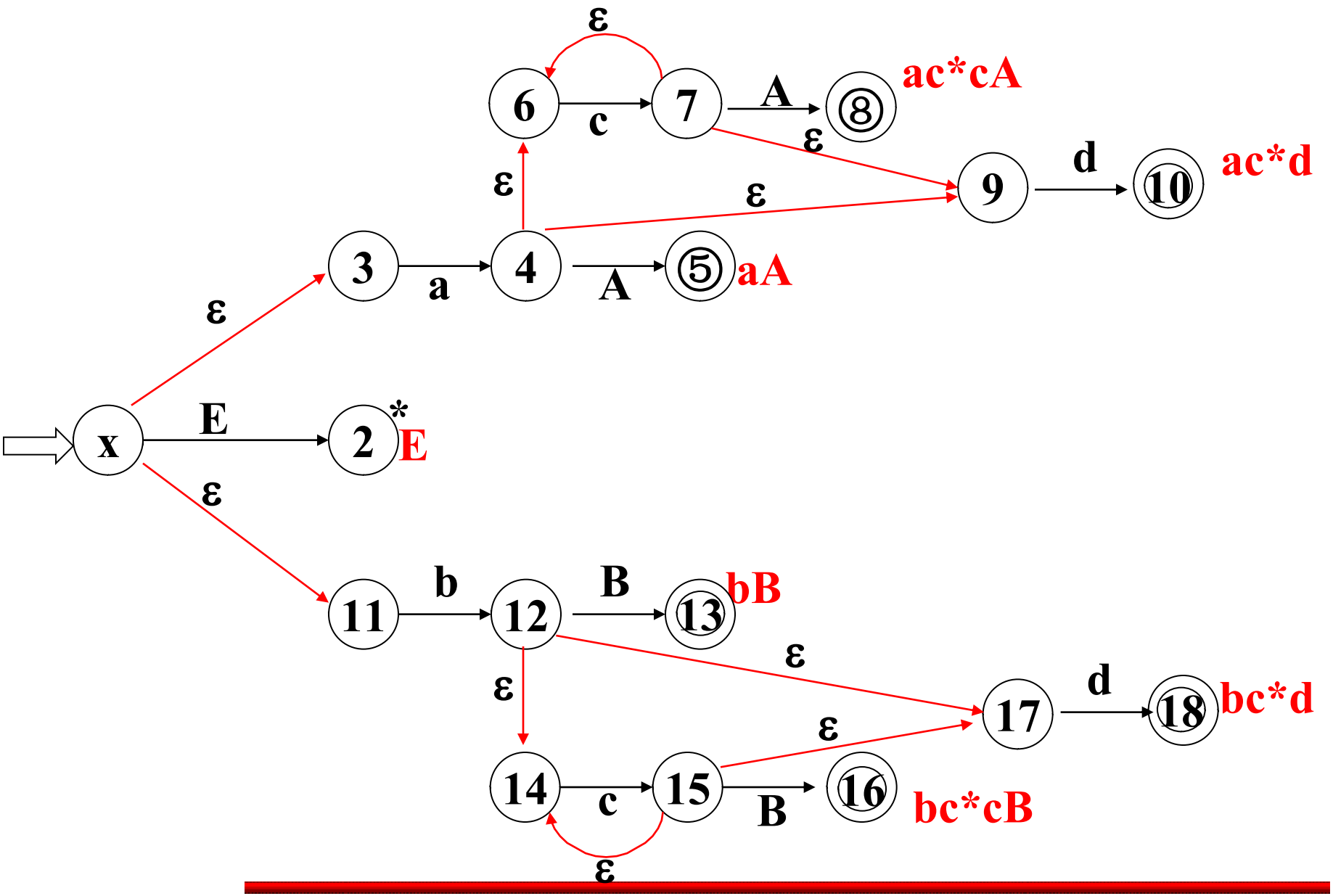
- 如果X_i为非终结符，则一定会有X_i为左部的有关项目及其相应的状态，例如：有项目形如:

  i项目: $X \rightarrow \gamma \cdot A\delta$

  k项目：$A \rightarrow \cdot \beta$

  则从状态i画标记为ε的箭弧到状态k，对于A的所有产生式圆点在最左边的的状态都连一条从i状态到该状态（这里是i和k)的箭弧，其上标记为ε

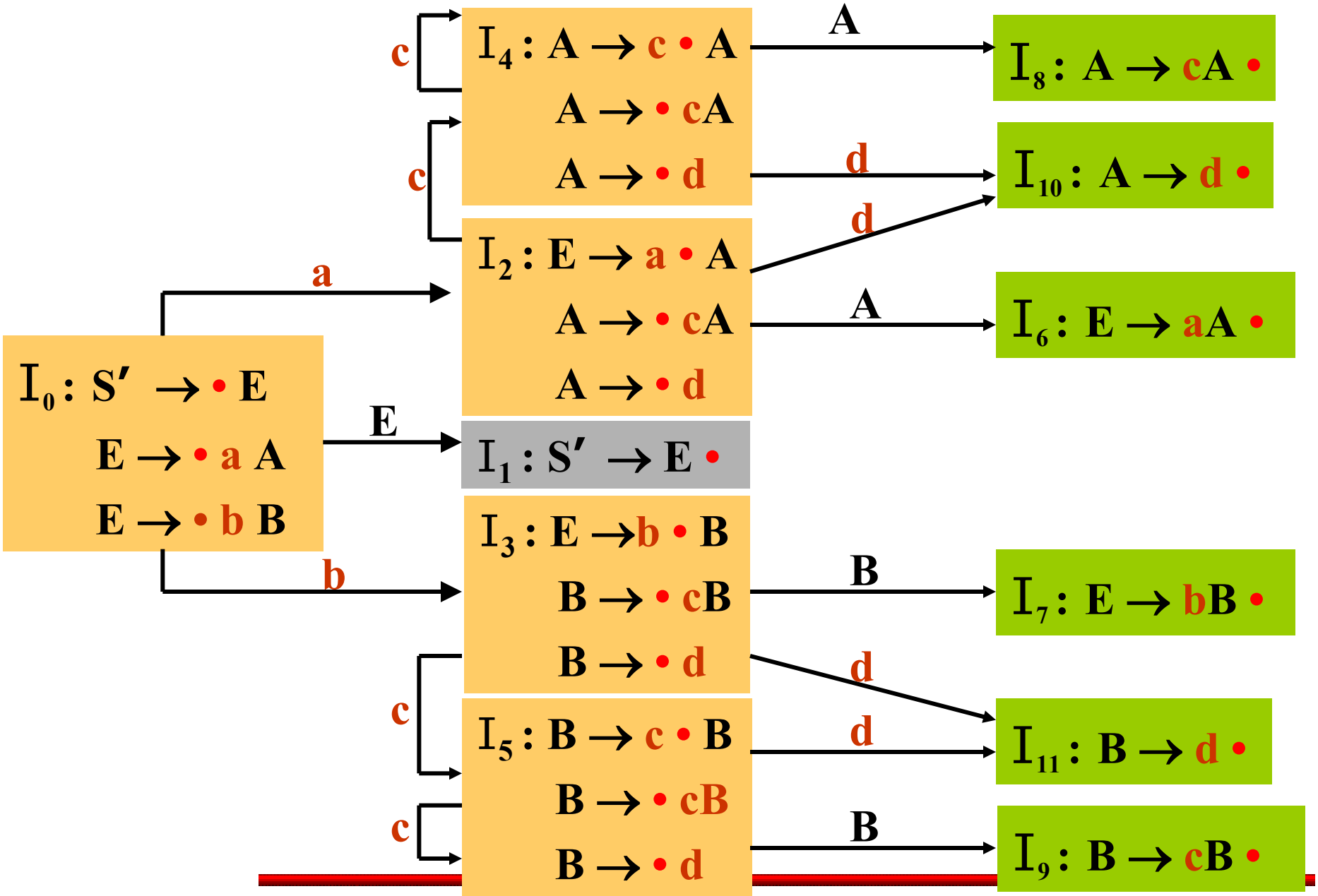
识别活前缀的不确定有限自动机



确定化：

		a	b	c	d	E	A	B	
I_0	1,3,11	4,6,9	12,14,17			2			0
I_1	2								1
I_2	4,6,9			7,6,9	10		5		0
I_3	12,14,17			15,14,17	18			13	0
I_4	7,6,9			7,6,9	10		8		0
I_5	15,14,17			15,14,17	18			16	0
I_6	5								1
I_7	13								1
I_8	8								1
I_9	16								1
I_{10}	10								1
I_{11}	18								1

识别可归前缀的DFA


I0 : S′ → • E
E → • a A
E → • b B
a
E
b
I1 : S′ → E •
I2 : E → a • A
A → • cA
A → • d
I3 : E → b • B
B → • cB
B → • d
I4 : A → c • A
A → • cA
A → • d
I5 : B → c • B
B → • cB
B → • d
c
c
c
c
A
d
d
A
B
d
d
B
I6 : E → aA •
I7 : E → bB •
I8 : A → cA •
I9 : B → cB •
I10 : A → d •
I11 : B → d •

项目类型：

项目类型有归约项目、移进项目、待约项目和接受项目。

① 归约项目：圆点在最右边的项目称为归约项目。如：$A \rightarrow \alpha \cdot$　此时已把α分析□束，　α已在□□，从而可按相应的产生式进行归约。

② 移进项目：圆点后面为终结符的项目称为移进项目。如$A \rightarrow \alpha \cdot a\beta$，　$a \in V_T$，此时把a移进，即a进符号栈。

③ 待约项目：圆点后面为非终结符的项目，称为待约项目。如$A \to \alpha \cdot B\beta$，$B \in V_N$，它表明所对应的状态等待着分析完非终结符B所能推出的串归约成B，才能继续分析A的右部。

④ 接受项目：当归约项目为$S' \to S \cdot$时则表明已分析成功，即输入串为该文法的句子，相应状态为接受状态。

LR(0)项目集规范族的构造

对于构成识别一个文法活前缀的DFA项目集(状态)的全体称为这个文法的LR(0)项目集规范族。

通过分析DFA每个状态中项目集的构成，可以发现如下规律：

若状态中包含$A \to \alpha \cdot B\beta$的□目，则形如$B \to \cdot\gamma$的□目也在此状□内。

项目集的闭包

若文法已拓广为G’，而S为文法G的开始符号，拓广后增加产生式S’→S。如果I是文法G’的一个项目集，定义和构造I的闭包CLOSURE(I)如下：

①I的项目均在CLOSURE(I)中

②若$A \rightarrow \alpha \cdot B\beta \in CLOSURE(I)$, 且$B \in V_N$则每一个形如$B \rightarrow \cdot \gamma$的项目也属于CLOSURE(I)

③重复②直到CLOSURE(I)不再增加为止。

我们可以很容易的构造出初态的项目集。由初态出发对其项目集的每个项目的圆点向右移动一个位置用箭弧转向不同的新状态，箭弧上用移动圆点经过的符号标记。

新状态的初始项目即圆点移动后的项目成为核。

转换函数定义

转换函数GO(I, X)定义如下：

GO(I, X)=CLOSURE(J)

其中：I为包含某一项目集的状态。

X为一文法符号，$X \in V_N \cup V_T$

J={任何形如$A \rightarrow \alpha X \cdot \beta$的□目| $A \rightarrow \alpha \cdot X\beta$属于I}

- □表明，若状□I□□活前□γ，□状□J□□活前□γX。
- □点不在□生式右部最左□的□目称□核，但S’→·S除外。
- 因此用GO(I, X)□□函数得到的J□□向后状□所含□目集的核。核可能是一个或多个□目□成。

构造LR(0)项目集规范族的步骤

1. 置项目S’→ · S□初□集的核，然后□核求□包，CLOSURE({S’→ • S})得到初□的□目集。
2. □初□集或其他构造的□目集□用□□函数GO(I, X)=CLOSURE(J)求出新状□J的□目集
3. 重复2直到不出□新的□目集□止。

构造识别文法活前缀DFA的三种方法的比较

第一种方法是根据形式定义求出活前缀的正规表达式，然后由此正规表达式构造NFA再确定化为DFA。

第二种方法是求出文法的所有项目，按一定规则构造识别活前缀的NFA再确定化为DFA。

第三种方法是把拓广文法的第一个项目{S'→ • S}作□初□集的核，通□求核的□包和和□□函数，求出LR(0)□目集□范族，再由□□函数建立状□之□的□接关系得到□□活前□的DFA。

第一种方法从理□上□比□□格确切，第2、3种方法比□直□，从直□上的分析与理□上□□的□果相吻合。

相同点：出发点都是把LR分析方法的归约过程，看成是识别文法规范句型活前缀的过程，因为只要分析到的当前状态时活前缀的识别态，则说明已分析过的部分是该文法某规范句型的一部分，也就说明了已分析过的部分是正确的。

LR(0)项目类型

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，$\alpha,\beta \in V^*$
- 归约项目，形如 $A \rightarrow \alpha \cdot$
- 待约项目，形如 $A \rightarrow \alpha \cdot B\beta$
- 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

项目集中的项目可能存在的两种冲突

- 移进和归约项目同时存在，形如：

  $A \rightarrow \alpha \cdot a\beta$

  $B \rightarrow \gamma \cdot$

  由于这是面临输入符号a时，不能确定移进a还是将γ归约为B，因为LR(0)分析是不向前看符号，所以对归约的项目不管当前符号是什么都应该归约。对于同时存在移进和归约项目的称移进-归约冲突。

- 归约和归约项目同时存在，形如：

  $A \rightarrow \beta \cdot$

  $B \rightarrow \gamma \cdot$

  因为这时不管面临什么输入符号都不能确定归约为A，还是归约为B，对同时存在两个以上归约项目的状态称为归约-归约冲突 。

LR(0)文法

对一个文法的LR(0)项目集规范族不存在移进-归约，或归约-归约冲突时，称这个文法为LR(0)文法。

LR(0)分析表的构造

假定已构造出来的LR(0)项目集规范族为$C=\{I_0, I_1, \ldots\ldots, I_n\}$，其中，$I_k$为项目集的名字，k为状态名。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为终结符，则置ACTION[k, a]为S_j，其动作意为将终结符a移进符号栈，状态j进入状态栈;
- 若项目A→α•属于I_k, 那么，对任何终结符a和“#”, 置ACTION[k, a]为“用产生式A→α进行规约”，简记为“r_j”;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填入信息的空白格均置上“出错标志”。

例子：文法G为：

(0) S’ →E

(1) E →aA

(2) E →bB

(3) A →cA

(4) A →d

(5) B →cB

(6) B →d

该文法的LR(0)分析表见下表：

状态	ACTION					GOTO		
	a	b	c	d	#	E	A	B
0	S_2	S_3				1		
1					acc			
2			S_4	S_{10}			6	
3			S_5	S_{11}				7
4			S_4	S_{10}			8	
5			S_5	S_{11}				9
6	r_1	r_1	r_1	r_1	r_1			
7	r_2	r_2	r_2	r_2	r_2			
8	r_3	r_3	r_3	r_3	r_3			
9	r_5	r_5	r_5	r_5	r_5			
10	r_4	r_4	r_4	r_4	r_4			
11	r_6	r_6	r_6	r_6	r_6			

LR(0)分析器的工作过程

对一个文法构造了LR(0)分析表后就可以在LR分析器的总控程序控制下对输入串进行分析，即根据输入串的当前符号和分析栈的栈顶状态查找分析表应采取的动作，对状态栈和符号栈进行相应的操作即移进、归约、接受或报错。具体说明如下：

- 若ACTION[S, a]=S_j，a为终结符，则把a移入符号栈，j移入状态栈
- 若ACTION[S, a]=r_j，a为终结符或“#”，则用第j个产生式进行归约，并将两个栈的指针减去k，其中k为第j个产生式右部的符号串长度，这时当前面临符号为第j个产生式左部的非终结符，设为A，归约后栈顶状态为n，再进行GOTO[n, A]。
- 若ACTION[S, a]=acc，a为“#”，则为接受，表示分析成功。
- 若GOTO[S, A]=j，A为非终结符，表明前一动作是关于A的产生式归约的，当前面临非终结符A应移入符号栈，j入状态栈。对于终结符GOTO[S, a]已和ACTION[S, a]重合。
- 若ACTION[S, a]=空白，则转向出错处理。

对于输入串#bccd#的分析过程如下：

步骤	状态栈	符号栈	输入符	action	goto
1	0	#	bccd#	S_3	
2	03	#b	ccd#	S_5	
3	035	#bc	cd#	S_5	
4	0355	#bcc	d#	S_{11}	
5	0355(11)	#bccd	#	r6	9
6	03559	#bccB	#	r5	9
7	0359	#bcB	#	r5	7
8	037	#bB	#	r2	1
9	01	#E	#	acc	

SLR(1)分析

1、问题的提出：

- 只有当一个文法G是LR(0)文法，即G的每一个状态项目集不存在移进-归约，或归约-归约冲突时，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得处理，以解决冲突。
- 因为只对有冲突的状态才向前查看一个符号，以确定做那种动作，因而称这种分析方法为简单的LR(1)分析法，用SLR(1)表示。

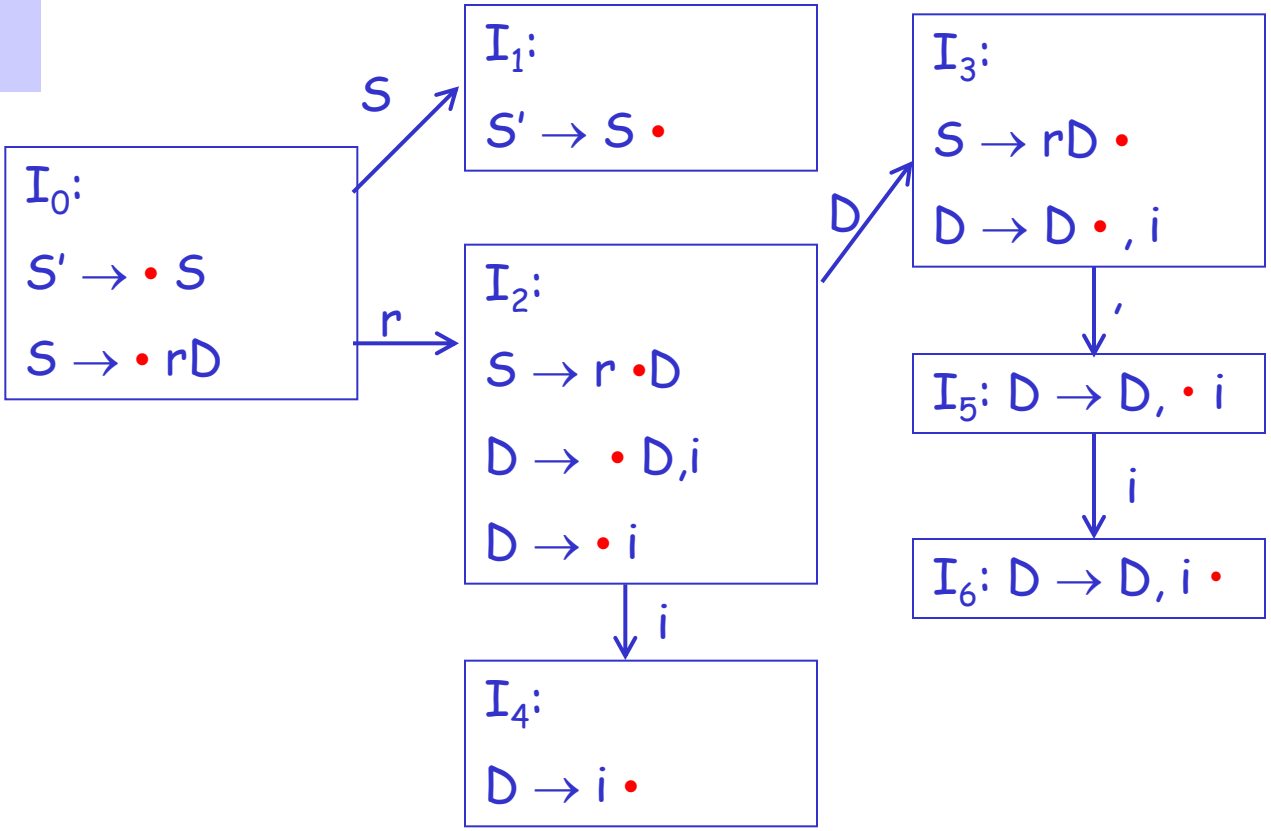
文法G':

(0) S' → S

(1) S → rD

(2) D → D,i

(3) D → i



分析每个状态包含的项目集，不难发现在状态I_3中含项目：

$S \rightarrow rD \cdot$　　为归约项目

$D \rightarrow D \cdot , i$　　为移进项目

也就是按$S \rightarrow rD \cdot$项目的动作认为用$S \rightarrow rD$产生式进行归约的句柄已形成，不管当前的输入符号是什么，都应把rD归约成S。但是按$D \rightarrow D \cdot ,i$项目当面临输入符为‘,’号时，应将‘,’号移入符号栈，状态转向I_5。显然该文法不是LR(0)文法，也可在构造它的LR(0)分析表时发现这个问题，如下表所示。

状态	ACTION				GOTO	
	r	,	i	#	S	D
0	S_2				1	
1				acc		
2			S_4			3
3	r_1	r_1, S_5	r_1	r_1		
4	r_3	r_3	r_3	r_3		
5			S_6			
6	r_2	r_2	r_2	r_2		

- 如何解决这种移进-归约冲突？
- LR（0）在归约时不向前看输入符号；
- 在LR（0）基础上，如果出现不相容的项目（存在移进-归约冲突或归约-归约冲突）则通过向前看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）

SLR(1)分析表的构造方法思想

- 在出现移进-归约冲突或归约-归约冲突时，通过观察归约成的非终结符的后跟符号集合，来区分移进-归约动作或不同的归约动作。
- 上例冲突的解决方法

  归约还是移进？

  如果归约，那么要构成一个合法的句子，该非终结符后都可以跟哪些终结符？

  而输入符号串中的当前符号是什么？

  是否匹配？

  匹配：则归约；不匹配：则不归约。

后跟符号集合的定义：

设G=（V_T，V_N，P，S）是上下文无关文法，A∈V_N，S是开始符号，

Follow(A)={a | $S \overset{*}{\Rightarrow} uA\beta$且a∈$V_T$，a ∈First(β)，u∈$V_T^*$，β∈$V^+$}。针对非终结符

若$S \overset{*}{\Rightarrow} uA\beta$，且$\beta \overset{*}{\Rightarrow} \varepsilon$，则#∈Follow(A)

（#表示输入串的结束符，或句子括号）

也可以写成为：

Follow(A)＝ {a|$S \overset{*}{\Rightarrow} \ldots Aa \ldots$， a∈$V_T$}

若$S \overset{*}{\Rightarrow} \ldots A$，则#∈Follow(A)。

Follow(A)是所有句型中出现在紧接A之后的终结符或“#”。

2、SLR(1)分析表的构造方法思想

在构造SLR(1)分析表时，根据不同的向前看符号，将S_i中的各项目所对应的动作加以区分，从而即可使冲突动作得到解决。

假定一个LR(0)规范族中含有如下的项目集（状态）I

$I=\{ X\rightarrow\alpha \cdot b\beta,\ A \rightarrow \gamma \cdot ,\ B\rightarrow\delta \cdot \}$

也就是在该项目集中含有移进-归约冲突和归约-归约冲突。其中$\alpha,\beta,\gamma,\delta$为文法符号串，b为终结符。

对于归约项目$A\rightarrow \gamma \cdot$，$B \rightarrow \delta \cdot$ 分别求Follow(A)和Follow(B)，

Follow(A)是所有句型中出现在紧接A之后的终结符或“#”。

如果满足如下条件：

$FOLLOW(A) \cap FOLLOW(B) = \varphi$

$FOLLOW(A) \cap \{b\} = \varphi$

$FOLLOW(B) \cap \{b\} = \varphi$

那么，当在状态S_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$a \in Follow(A)$,则用产生式$A \rightarrow \gamma$进行归约。

(3) 若$a \in Follow(B)$,则用产生式$B \rightarrow \delta$进行归约。

(4) 此外，报错。

通常对于LR(0)规范族的一个项目集I中可能含有多个移进项目和多个归约项目。假设项目集中有m个移进项目： $A_1 \rightarrow \alpha_1 \cdot a_1\beta_1$， $A_2 \rightarrow \alpha_2 \cdot a_2\beta_2$，…， $A_m \rightarrow \alpha_m \cdot a_m\beta_m$；同时含有n个归约项目：$B_1 \rightarrow \gamma_1 \cdot$, $B_2 \rightarrow \gamma_2 \cdot$ ，…， $B_n \rightarrow \gamma_n \cdot$ ，只要集合$\{a_1, a_2, \ldots, a_m\}$和$FOLLOW(B_1)$, $FOLLOW(B_2)$, …, $FOLLOW(B_n)$两两交集都为空，那么我们仍然可以用上述规则解决冲突即考查当前输入符号决定动作。

(1) 若$a \in \{a_1, a_2, \ldots, a_m\}$，则移进。

(2) 若$a \in Follow(B_i)$, i=1, 2, …, n，则用产生式$B_i \rightarrow \gamma_i$进行归约。

(3) 此外，报错。

- 如果对于一个文法的LR(0)项目集规范族所含有的动作冲突都能用以上方法来解决，则称该文法为SLR(1)文法，所构造的分析表为SLR(1)分析表，使用SLR(1)分析表的分析器为SLR(1)分析器。

3、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	I_1
	E→ · E+T	E	I_1
	E→ · T	T	I_2
	T → · T *F	T	I_2
	T → · F	F	I_3
	F→ · （E）	(	I_4
	F→ · i }	i	I_5
I_1	{ S'→E ·		
	E→E · +T }	+	I_6

状态	项目集	后继符号	后继状态
I_2	{E→T· T→T·*F}	*	I_7
I_3	{T→F·}		
I_4	{F→（·E） E→·E+T E→·T T→·T*F T→·F F→·（E） F→·i}	E E T T F (i	I_8 I_8 I_2 I_2 I_3 I_4 I_5

状态	项目集	后继符号	后继状态
I_5	{F→ i · }		
I_6	{E→E＋·T	T	I_9
	T→ · T*F	T	I_9
	T → · F	F	I_3
	F→·（E）	(	I_4
	F→· i }	i	I_5
I_7	{T→T* · F	F	I_{10}
	F→ ·（E）	(	I_4
	F→ · i }	i	I_5

状态	项目集	后继符号	后继状态
I_8	{F→(E ·) E→E · +T }	) +	I_{11} I_6
I_9	{E→E+T · T→T · *F }	*	I_7
I_{10}	{T→T*F · }		
I_{11}	{F→(E) · }		

由上图可见，I_1、I_2和I_9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
0	S_5			S_4			1	2	3
1		S_6				acc			
2	r_2	r_2	r_2 S_7	r_2	r_2	r_2			
3	r_4	r_4	r_4	r_4	r_4	r_4			
4	S_5			S_4			8	2	3
5	r_6	r_6	r_6	r_6	r_6	r_6			
6	S_5			S_4				9	3
7	S_5			S_4					10
8		S_6			S_{11}				
9	r_1	r_1	r_1 S_7	r_1	r_1	r_1			
10	r_3	r_3	r_3	r_3	r_3	r_3			
11	r_5	r_5	r_5	r_5	r_5	r_5			

从上表也可见在I_1, I_2, I_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I_1：{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I_1中的冲突可解决。

对于I_2：I_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。

对于I_9：I_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。

由以上考查，该文法在I_1，I_2，I_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。

进一步分析我们可以发现如下事实：例如在状态I_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。

但是很显然T的后跟符没有‘（’符号，如果当前面临输入符是‘（’，也进行归约显然是错误的。因此我们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)分析表我们称它为改进的SLR(1)分析表。

改进的SLR(1)分析表的构造方法如下：

- 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S_j;
- 若项目A→α•属于I_k, 那么，对任何终结符a和“#”,,且满足a∈Follow(A)时，置ACTION[k, a]=r_j，假定A→ α□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填入信息的空白格均置上“出错标志”。

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			

7.4 LR(1)分析表的构造

1、问题的提出

在SLR(1)方法中，对于某状态S_i，其项目集若不相容时，可根据SLR(1)分析表的构造规则来解决冲突分析动作，但如果不相容的项目集中的FOLLOW集合及其有关集合相交时，就不可能通过SLR(1)分析表构造规则来构造SLR(1)分析表。这时就用LR(1)分析。

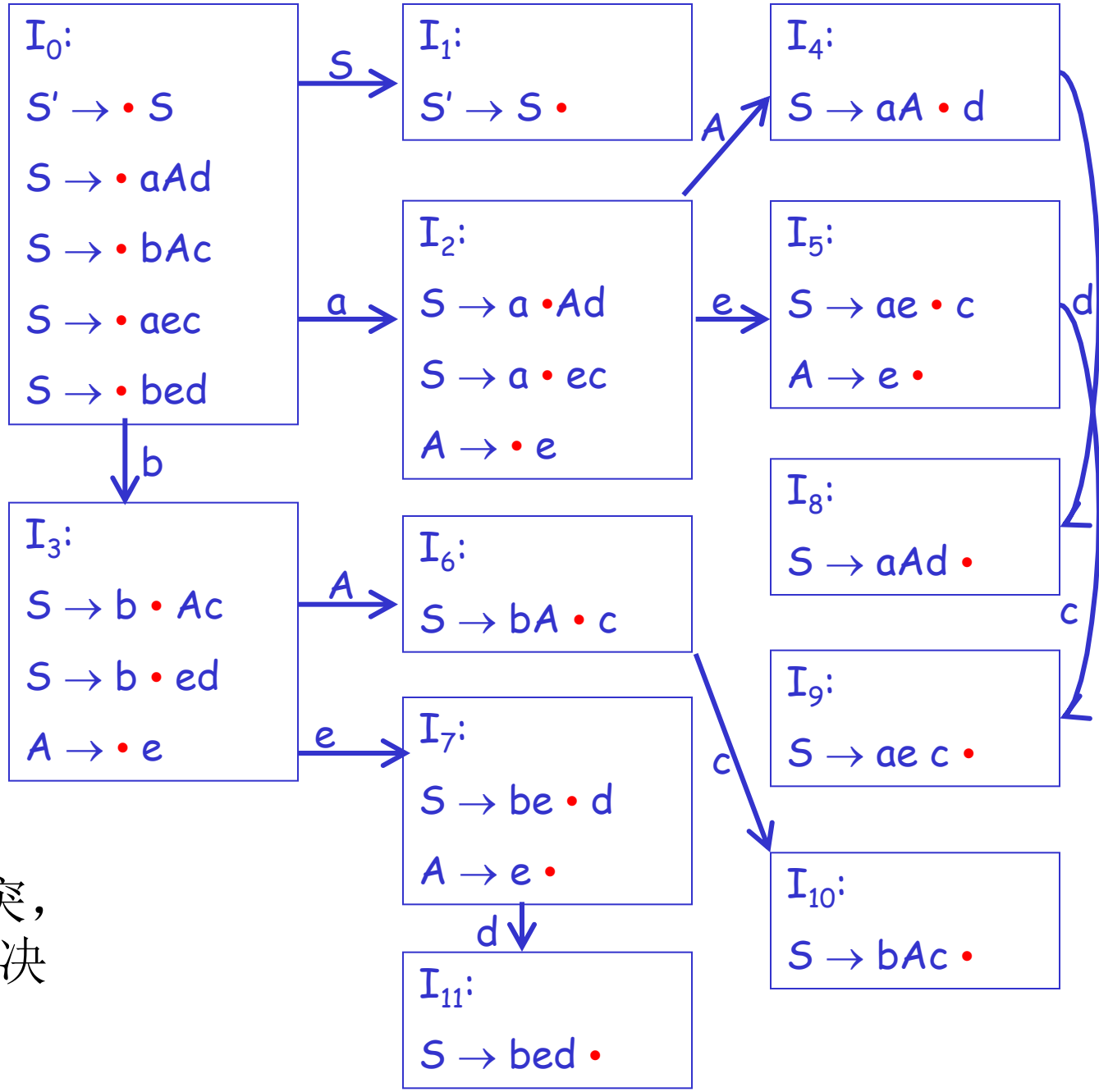
例：文法G:

(0)S`→S　　(1) S→aAd　　(2) S→bAc

(3) S→aec　　(4) S→bed　　(5) A→e

文法G':
(0) S' → S
(1) S → aAd
(2) S → bAc
(3) S → aec
(4) S → bed
(5) A → e

FOLLOW(A)={c, d}



查看I_5, I_7中的冲突，
体会LR(1)如何解决

I_5: S →ae•c

A →e•

$S' \underset{R}{==>} S \underset{R}{==>} aAd \underset{R}{==>} aed$

$S \underset{R}{==>} S \underset{R}{==>} aec$

活前缀ae遇到c应移进；遇到d应归约

I_7: S →be•d

A →e•

$S' \underset{R}{==>} S \underset{R}{==>} bAc \underset{R}{==>} bec$

$S' \underset{R}{==>} S \underset{R}{==>} bed$

活前缀be遇到d应移进；遇到c应归约

并不是Follow（A）的每个元素在含A的所有句型中都会在A的后面出现

LR（1）方法

- 在每个项目中增加向前搜索符
- 若项目集[A→α•Bβ]属于I时，则[B→•γ]也属于I
- 把FIRST(β)作为用产生式归约的搜索符（称为向前搜索符），作为用产生式B→γ□□□□看的符号集合（用以代替SLR(1)分析中的FOLLOW集），并把此搜索符号的集合也放在相应项目的后面，这种处理方法即为LR(1)方法

LR(1)项目集族的构造：针对初始项目S’→•S,#求闭包后再用转换函数逐步求出整个文法的LR(1)项目集族。

1）构造LR（1）项目集的闭包函数

a) I的项目都在CLOSURE(I)中

b) 若A→α•Bβ，a属于CLOSURE(I)，B→γ是文法的产生式，β∈V*，b∈FIRST(βa),则B→•γ,b也属于CLOSURE(I)

c) 重复b)直到CLOSURE(I)不再扩大

2)转换函数的构造

GOTO（I，X）= CLOSURE（J）

其中：I为LR(1)的项目集，X为一文法符号

J={任何形如A→αX • β,a的项目| A→α • X β,a属于I}

一个文法符号串的first集合计算方法：

如果文法符号串$\alpha \in V^*$, $\alpha = X_1X_2\ldots X_n$,

1、当$X_1 \overset{*}{\nRightarrow} \varepsilon$，则$first(\alpha)=first(X_1)$

2、当对任何j（$1 \le j \le i-1$，$2 \le i \le n$），$\varepsilon \in first(X_j)$

则$first(\alpha)=(first(X_1)-\{\varepsilon\}) \cup (first(X_2)-\{\varepsilon\}) \cup \ldots \cup (first(X_i\text{-}1)-\{\varepsilon\}) \cup first(X_i)$

3、当$first(X_j)$都含有ε时$(1 \le j \le n)$，则
$first(\alpha)=first(X_1) \cup first(X_2) \cup \ldots \cup first(X_j) \cup \{\varepsilon\}$

文法G':
(0) S' → S
(1) S → aAd
(2) S → bAc
(3) S → aec
(4) S → bed
(5) A → e

I_0:
S' → • S, #
S → • aAd, #
S → • bAc, #
S → • aec, #
S → • bed, #

I_1:
S' → S •, #

I_2:
S → a •Ad, #
S → a • ec, #
A → • e, d

I_3:
S → b • Ac, #
S → b • ed, #
A → • e, c

I_4:
S → aA • d, #

I_5:
S → ae • c, #
A → e •, d

I_6:
S → bA • c, #

I_7:
S → be • d, #
A → e •, c

I_8:
S → aAd •, #

I_9:
S → ae c •, #

I_{10}:
S → bAc •, #

I_{11}:
S → bed •, #

查看I_5, I_7中的冲突，
体会LR(1)如何解决

LR(1)分析表的构造

- LR(1)分析表的构造与LR(0)分析表的构造在形式上基本相同，不同之处在于：归约项目的归约动作取决于该归约项目的向前搜索符号集。

1) 若项目[**A→α • aβ,b**]属于I_k，且转换函数$GO(I_k,a)= I_j$，当a为终结符时，则置ACTION[k,a]为S_j
2) 若项目[**A→α •** ,a]属于I_k，则对a为任何终结符或‘#’，置$ACTION[k,a] = r_j$，j为产生式在文法G‘中的编号
3) 若$GO(I_k,A)= I_j$，则置GOTO[k,A]=j，其中A为非终结符，j为某一状态号
4) 若项目[**S‘→S •** ,#]属于I_k，则置ACTION[k,#] = acc
5) 其它填上“报错标志”

LR(1)分析表

	ACTION						GOTO	
	a	c	e	b	d	#	S	A
0	S2			S3			1	
1						acc		
2			S5					4
3			S7					6
4					S8			
5		S9			r5			
6		S10						
7		r5			S11			
8						r1		
9						r3		
10						r2		
11						r4		

- 如果对于一个文法的LR(1)分析表不含多重入口时，（即任何一个LR(1)项目集中无移进-归约冲突或归约-归约冲突），则称该文法为LR(1)文法，所构造的分析表为LR(1)分析表，使用LR(1)分析表的分析器为LR(1)分析器或称规范的LR分析器。
- 一个文法是LR(0)文法一定也是SLR(1)文法，也是LR(1)文法。反之则不一定成立。

LALR(1)分析

文法G’：

(0) S’ → S

(1) S → BB

(2) B → aB

(3) B → b

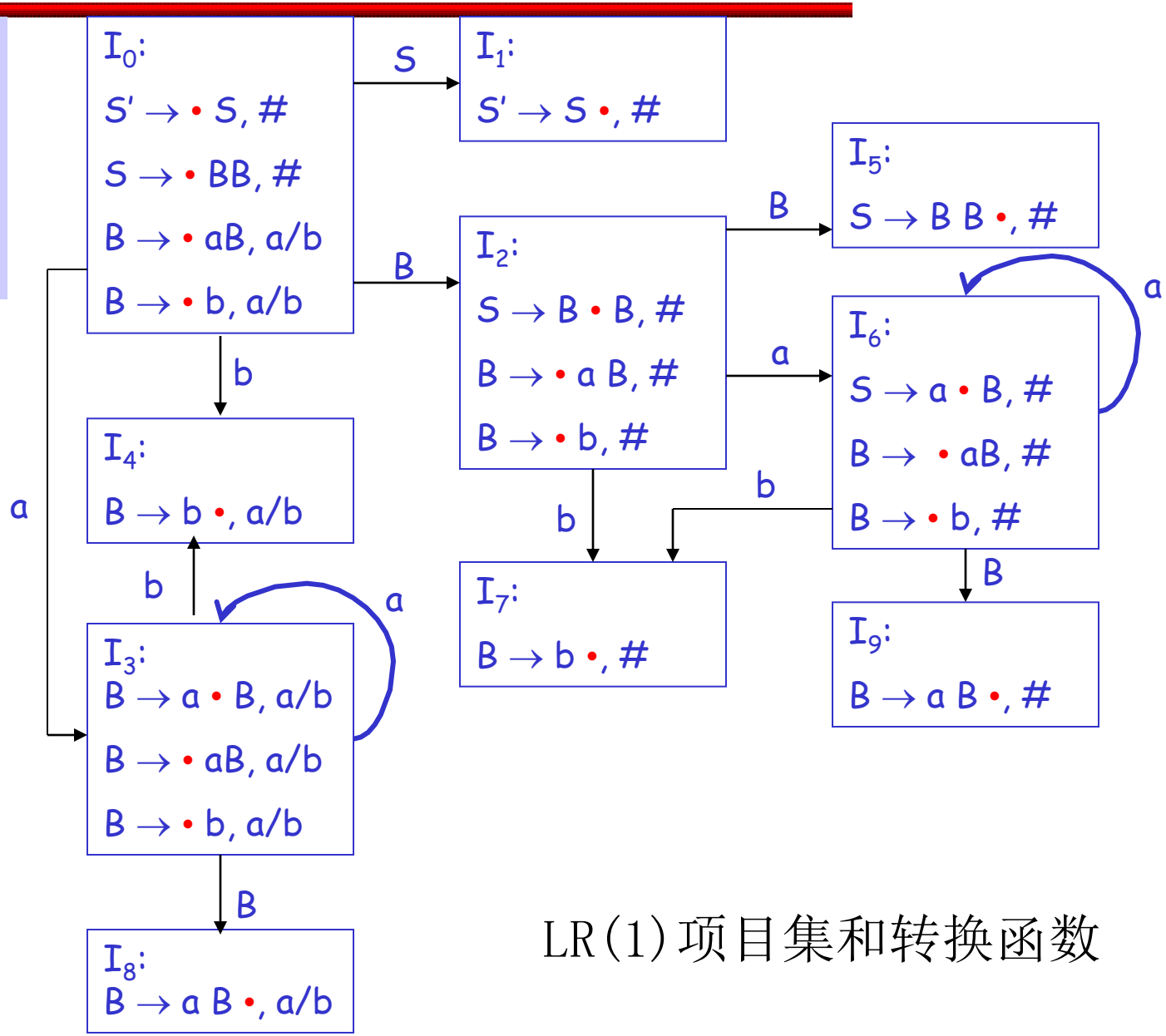
文法G'：

(0) S' → S

(1) S → BB

(2) B → aB

(3) B → b



LR(1)项目集和转换函数

如果两个LR(1)项目集去掉搜索符之后是相同的，则称这两个项目集具有相同的心。

分析可发现I_3和I_6，I_4和I_7，I_8和I_9分别为同心集

I_3:
$B \rightarrow a \cdot B, a/b$
$B \rightarrow \cdot aB, a/b$
$B \rightarrow \cdot b, a/b$

I_6:
$S \rightarrow a \cdot B, \#$
$B \rightarrow \cdot aB, \#$
$B \rightarrow \cdot b, \#$

合并为

$I_{3,6}$:
$S \rightarrow a \cdot B, a/b/\#$
$B \rightarrow \cdot aB, a/b/\#$
$B \rightarrow \cdot b, a/b/\#$

I_4:
$B \rightarrow b \cdot, a/b$

I_7:
$B \rightarrow b \cdot, \#$

合并为

$I_{4,7}$:
$B \rightarrow b \cdot, a/b/\#$

I_8:
$B \rightarrow a\ B \cdot, a/b$

I_9:
$B \rightarrow a\ B \cdot, \#$

合并为

$I_{8,9}$:
$B \rightarrow a\ B \cdot, a/b/\#$

LALR(1)分析

- 对LR(1)项目集规范族合并同心集，若合并同心集后不产生新的冲突，则为LALR(1)项目集。

合并同心集的几点说明

- 同心集合并后心仍相同，只是超前搜索符集合为各同心集超前搜索符的和集
- 合并同心集后转换函数自动合并
- LR(1)文法合并同心集后也只可能出现归约-归约冲突，而没有移进-归约冲突
- 合并同心集后可能会推迟发现错误的时间，但错误出现的位置仍是准确的

构造 LALR(1)分析表

1.构造文法G的规范 LR(1) 状态.

2.合并同心集的状态.

3.新 LALR(1) 状态的GO函数是合并的同心集状态的GO函数的并

4. LALR(1)分析表的ACTION 和 GOTO 登录方法与LR(1)分析表一样

经上述步骤构造的表若不存在冲突，则称它为G的LALR(1)分析表。

存在这种分析表的文法称为LALR（1）文法。

合并同心集后

状态	ACTION			GOTO	
	a	b	#	S	B
0	$S_{3,6}$	$S_{4,7}$		1	2
1			acc		
2	$S_{3,6}$	$S_{4,7}$			5
3,6	$S_{3,6}$	$S_{4,7}$			8,9
4,7	r_3	r_3	r_3		
5			r_1		
8,9	r_2	r_2	r_2		

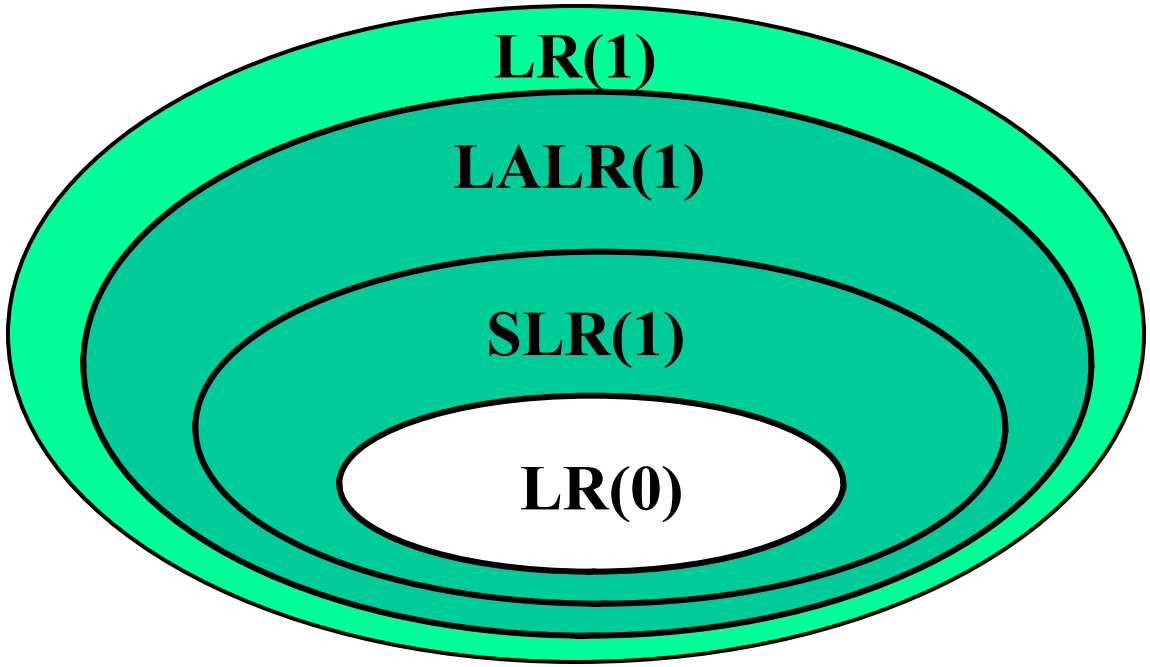
对 输入串ab#用LR(1)分析的过程

步骤	状态栈	符号栈	输入串	ACTION	GOTO
• 1	0	#	ab#	S_3	
• 2	03	#a	b#	S_4	
• 3	034	#ab	#	出错	

对输入串ab#用LALR(1)分析的过程

步骤	状态栈	符号栈	输入串	ACTION	GOTO
• 1	0	#	ab#	$S_{3,6}$	
• 2	0(3,6)	#a	b#	$S_{4,7}$	
• 3	0(3,6)(4,7)	#ab	#	r_3	(8,9)
• 4	0(3,6)(8,9)	#aB	#	r_2	2
• 5	02	#B	#	出错	

四种LR文法之间的关系



例

(0) S’→S

(1)S→L=R

(2)S→R

(3)L→ *R

(4)L→id

(5)R→L

不能用SLR(1)技术解决，但能用LR(1)和LALR(1)技术解决

例

(0) S’→S

(1)S→aAd

(2)S→bBd

(3)S→ aBe

(4)S→bAe

(5)A→c

(6)B→c

本文法是LR(1)文法而不是LALR(1)文法。

判断一个文法是何种LR文法的步骤

构造LR(0)项目集规范族

If 所有的项目集不含有移进-归约冲突和归约-归约冲突，则为LR(0)文法；

Else if 冲突项目可以通过考察非终结符的后跟符号集来解决，则为SLR(1)文法；

Else 构造LR(1)项目集规范族

If 任何项目集中都不存在动作冲突，则为LR(1)文法；

对LR(1)项目集规范族进行同心集的合并，如合并之后仍不存在冲突，则为LALR(1)文法。

几种文法的比较

- **LR(0)**
- **SLR(1): 生成的LR(0)项目集如有冲突，则根据非终结符的FOLLOW集决定**
- **LR(1)、LR(k): 项由 核心与向前搜索符组成，搜索符长度为1或k**
- **LALR(1): 对LR(1)项目集规范族合并同心集**
- **由弱到强：LR（0）、SLR（1）、LALR（1）、LR（1）**
- **LR（1）中的向前搜索符号集合是与该项目相关的非终结符号的Follow集的子集；**
- **LALR项目的搜索符一般是与该项目相关的非终结符号的Follow集的子集，这正是LALR分析法比SLR分析法强的原因。**

二义性文法在LR分析中的应用

- **表达式文法：**

  $E' \rightarrow E$

  $E \rightarrow E+E$

  $E \rightarrow E*E$

  $E \rightarrow (E)$

  $E \rightarrow i$

二义性文法不是LR文法，但是对某些二义性文法，人为地给出优先性和结合性可能构造出更有效的LR分析器

如：规定在表达式文法中i的优先性最高

‘*’ .＞ ‘＋’

‘*’和‘＋’都服从左结合

算术表达式二义性文法的 LR(0)项目集及状态转换矩阵

在I_1, I_7, I_8中 存在移进 -归约冲突

在I_1, I_7, I_8中 存在移进 -归约冲突

I_1： E′ →E•

E →E•+E

E →E•*E

I_7： E →E+E•

E →E•+E

E →E•*E

I_8： E →E*E•

E →E•+E

E →E•*E

I_1： 移进和接受无冲突

I_7, I_8用优先关系和结合性解决冲突。

规定：‘*’优先于‘+’，都服从左结合

I_7： 遇‘*’移进

遇‘+’归约

I_8： 遇‘+’，‘*’都归约

二义性表达式文法的LR分析表

状态	ACTION						GOTO
	+	*	(	)	i	#	E
0			S2		S3		1
1	S4	S5				acc	
2			S2		S2		6
3	r4	r4		r4		r4	
4			S2		S3		7
5			S2		S3		8
6	S4	S5		S9			
7	r1	S5		r1		r1	
8	r2	r2		r2		r1	
9	r3	r3		r3		r3	

对输入串i+i*i的分析过程

步骤	状态栈	符号栈	输入串	ACTION	GOTO
1	0	#	i+i*i#	S3	
2	03	#i	+i* i#	r4	1
3	01	#E	+i*i#	S4	
4	014	#E+	i*i#	S3	
5	0143	#E+i	*i#	r4	
6	0147	#E+E	*i#	S5	
7	01475	#E+E*	i#	S3	
8	014753	#E+E*i	#	r4	8
9	014758	#E+E*E	#	r2	
10	0147	#E+E	#	r1	1
11	01	#E	#	acc	

作业

- 7、9