Lecture 7: LR分析法

Xiaoyuan Xie 谢晓园
xxie@whu.edu.cn
计算机学院E301

复习与回顾

自底向上的语法分析

- **为一个输入串构造语法分析树的过程；**
  - 从叶子（输入串中的终结符号，将位于分析树的底端）开始，向上到达根结点
- **将一个串*w*归约为文法符号的过程**
  - 在每一步的归约中，一个与某产生式体相匹配的特定子串被替换为该产生式左部的非终结符号，一次归约实质上是一个推导的反向操作
  - 选择哪部分进行归约?
  - 应用哪个产生式进行归约?

自底向上分析

- **示例**

句柄

重要

- **最右句型γ的一个句柄**
  - 最左简单短语，即最右推导的反向过程中被规约的那些部分
  - 对句柄的规约，代表了相应的最右推导中的一个反向步骤

- 正确的被归约的对象称为句柄 (handle)，即它能保证被归约后一定还保持着最右句型. 例子

  1) exp PLUS term TIMES ID $\underset{rm}{\Leftarrow}$ exp PLUS term TIMES fac

  2) exp PLUS term TIMES ID $\underset{rm}{\Leftarrow}$ exp PLUS exp TIMES ID

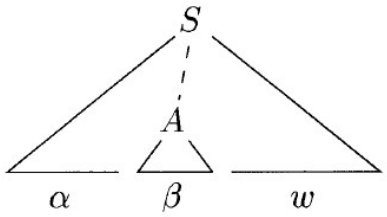
  3) exp PLUS term TIMES ID $\underset{rm}{\Leftarrow}$ exp TIMES ID

  只有 (1) 是句柄.

- 如果文法是没有二义性的，则一个最右句型的句柄是唯一的.
- 寻找句柄是解决自底向上分型的关键.

自底向上句法分析概述: 句柄剪枝

- **Bottom-Up Parsing – 归约过程 – 反向最右推导过程 – 句柄剪枝过程**
  - 句柄：和某个产生式体相匹配的子串，对它的归约代表了相应的最右推导的一个反向步骤
  - 句柄的形式定义：$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$，那么紧跟$\alpha$的$\beta$可以一步归约到$A$。 w是所有的终结符号串。
- **注意：**
  - 归约前后都是最右句型
  - 和某个产生式匹配的最左子串不一定是句柄
  - 无二义性的文法，其每个最右句型都有且只有



自底向上语法分析概述: 句柄剪枝（续）

- 通过“句柄剪枝”可以得到一个反向的最右推导。从句子w开始，令$w=\gamma_n$，γ_n是未知最右推导的第n个最右句型

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

- 以相反的顺序重构这个推导（实际上是重构归约序列），我们在γ_n中寻找句柄β_n，并将替换为相应产生式$A \rightarrow \beta_n$的头部，得到前一个最右句型γ_{n-1}；重复这个过程，直到S
- 和推导类似，如果能重现这个序列，则可以完成语法分析任务（其中可能存在什么问题）
  - 如何找到句柄

移进-归约语法分析框架

- 一种自底向上的分析形式，
- 使用一个栈保存文法符号，一个输入缓冲区存放将要进行分析的剩余符号
- 初始栈： $　　初始输入 *w*$
- 对输入串的一次从左到右的扫描过程中，语法分析器将零个或多个输入符号移到栈的顶端，直到它可以**对栈顶的一个文法符号串**进行**归约**为止。它将归约为某个产生式的头。不断重复这个循环，直到它检测到一个语法错误，或者栈中包含了开始符号且输入缓冲区为空。
- 分析成功时： 栈 $ S,　输入 $

自顶向下的非递归预测分析中栈是如何工作的？

主要分析动作

- **移入：将下一个输入符号移动到栈顶**
- **归约：将句柄归约为相应的非终结符号**
  - 句柄总是在栈顶
  - 具体操作时弹出句柄，压入被归约到的非终结符号
- **接受：宣布分析过程成功完成**
- **报错：发现语法错误，调用错误恢复子程序**

移进-规约主要分析动作

- **示例**

最基本的移进-规约技术-LR(0)

- **规范LR(0)项集族: 提供构建LR(0)自动机的基础**
  - LR(0)自动机可用于做出语法分析决定
  - LR(0)自动机中每个状态代表了LR(0)项集族中的一个项集
- **增广文法**
- **符号栈 → 状态栈+符号栈(辅助)**

规范LR(0)项集族的构造

- **构造过程中用到的子函数**
  - **CLOSURE(I)：I的项集闭包**
    - **对应于DFA化算法的 ε-CLOSURE**
  - **GOTO(I,X)：I的X后继**
    - **对应于DFA化算法的MOVE(I,X)**

LR(0)项集中的内核项和非内核项

- 内核项：初始项S’ →.S、以及所有点不在最左边的项
- 非内核项：除了S’ →.S之外、点在最左边的项
- 由于表可能很庞大，实现算法时可以考虑只保存相应的非终结符号；甚至可以只保存内核项，而在要使用非内核项时调用CLOSURE函数重新计算

LR语法分析器的格局

- **LR语法分析器的格局包含了栈中内容和余下输入**

  $(s_0 s_1 \ldots s_m,\ a_i a_{i+1} \ldots a_n \$)$

  - 第一个分量是栈中的内容（右侧是栈顶）
  - 第二个分量是余下输入
- **LR语法分析器的每一个状态都对应一个文法符号（s0除外）**
  - 如果进入状态s的边的标号为X，那么s就对应于X
- **令Xi为si对应的符号，那么**
  - $X_1 X_2 \ldots X_m\ a_i a_{i+1} \ldots a_n$对应于一个最右句型

LR语法分析器的行为

- **对于格局($s_0s_1\ldots s_m$, $a_ia_{i+1}\ldots a_n$\$)，LR语法分析器查询条目ACTION[s_m,a_i]确定相应动作**
  - 移入s：执行移入动作，将s（对应a）移入栈中，新格局($s_0s_1\ldots s_ms$，$a_{i+1}\ldots a_n$\$)
  - 归约$A \rightarrow \beta$，将栈顶的β归约为A，压入状态s: ($s_0s_1\ldots s_{m-r}s$，$a_ia_{i+1}\ldots a_n$\$)
    其中r是β的长度，$s=GOTO[s_{m-r},A]$
  - 接受：语法分析过程完成
  - 报错：发现语法错误，调用错误恢复例程

LR语法分析过程

- **示例**

7.1 LR(0)局限性

7.1 LR(0)分析的局限

- LR(0)文法仅凭符号栈里的内容来确定可归约活前缀, 非常容易产生冲突;
- LR(0)文法易于产生冲突的原因在于在确定分析动作时没有考虑输入串信息。
- 事实上,只有有限的文法能满足LR(0)文法的条件;
- LR(0)分析不是一种实用的方法，只是为介绍LR分析的思想而引进的；

LR(0)自动机的移入-归约冲突

$V_T = \{a, b\}$

$V_N = \{S, A\}$

S = S

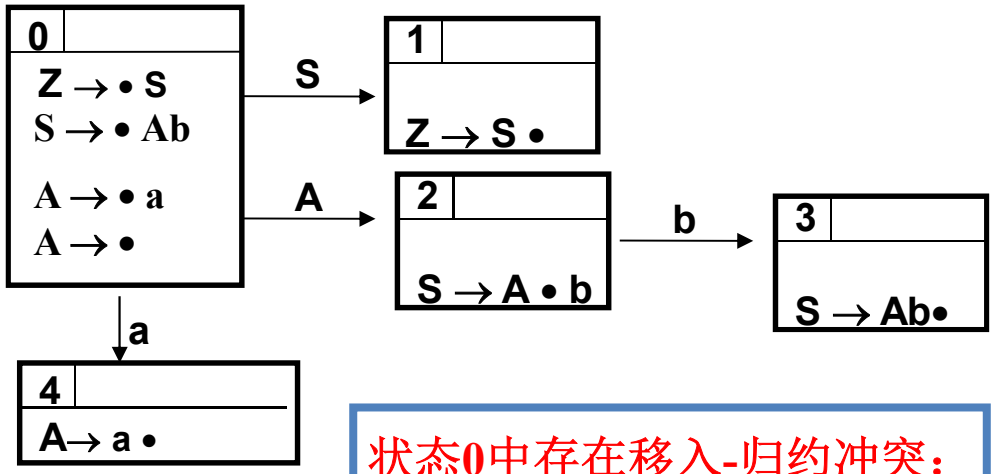
P:

{ (1) S→ Ab

(2) A → ε

(3) A → a

}



状态0中存在移入-归约冲突：

(1) 移入项目:A → • a

(2) 归约项目: A → •

LR(0)自动机的归约-归约冲突

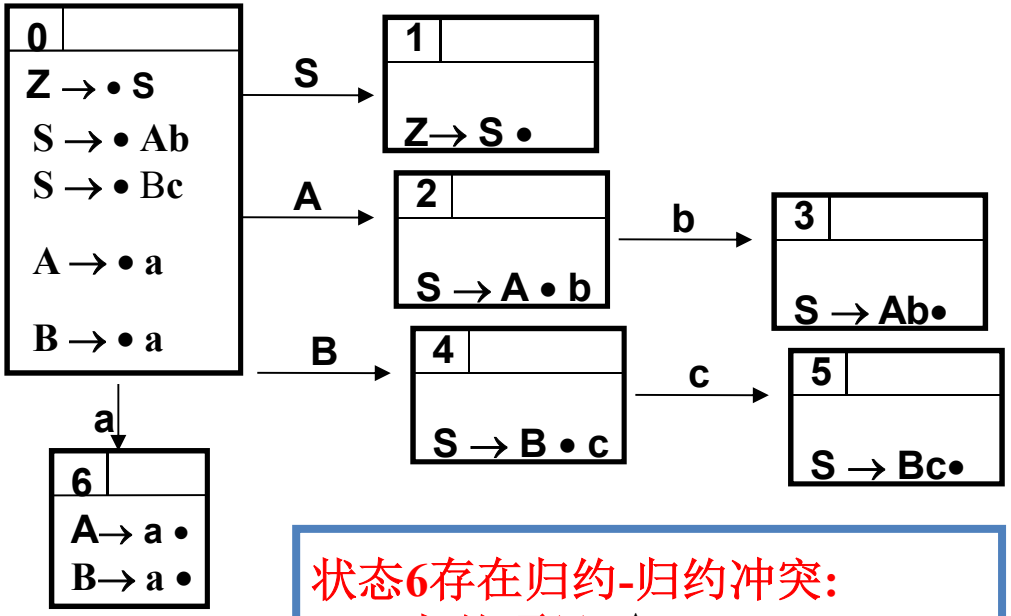
V_T = {a, b, c}

V_N = {S, A, B}

S = S

P:

{(1) S→ Ab
(2) S→ Bc
(3) A → a
(4) B → a
}



状态6存在归约-归约冲突:

(1) 归约项目: A → a •

(2) 归约项目: B → a•

如何消除冲突？

- **改进LR(0)**
  - SLR
  - LR(1)
  - LALR

7.2 SLR

改进策略之SLR

- **消除冲突：向前看一个输入符号来选择分析动作(考虑现实)**
  - 对于任何形如I={X→γ• bβ，A→ α• ，B→ α•}的LR(0)项目集，
    - 假设下一个输入符号是: a，则存在规约冲突；但
    - 如果Follow(A)∩Follow(B) = Φ, 且b∉Follow(A)，b∉Follow(B)，则可以通过如下方法进行冲突消除

改进策略之SLR

- **消除冲突：向前看一个输入符号来选择分析动作(考虑现实)**
  - 移入-归约冲突(S-R冲突)
    - 移入: 如存在移入项目 $A\rightarrow \alpha\bullet a\beta$
    - 归约: 如果存在归约项目 $B\rightarrow \pi\bullet$, 且 $a \in follow(B)$
  - 归约-归约冲突(R-R冲突)
    - 归约(P1): 如果存在归约项目 $A\rightarrow \pi\bullet$, $a\in follow(A)$, 产生式P1= $A\rightarrow \pi$
    - 归约(P2): 如果存在归约项目 $B\rightarrow \pi'\bullet$, $a\in follow(B)$, 产生式P2= $B\rightarrow\pi'$

LR(0)分析表 (带有S-R 冲突)

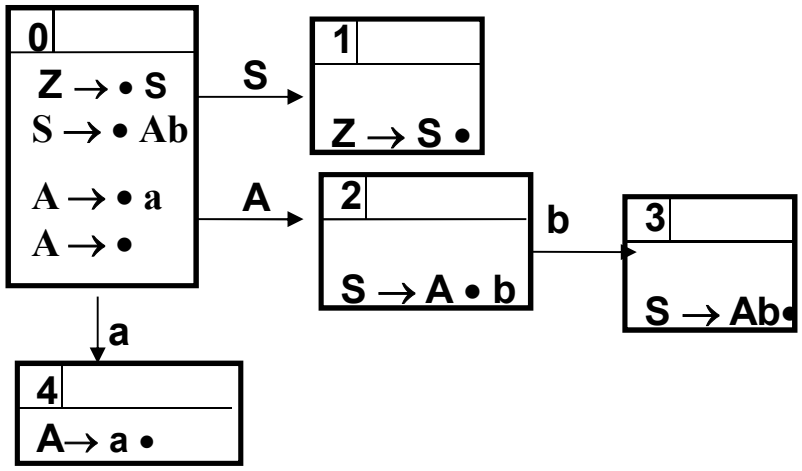
V_T = {a, b}

V_N = {S, A}

S = S

P:

{ (1) S→ Ab
(2) A → ε
(3) A → a
}



状态0中存在移入-归约冲突：
(1) 移入项目: A → • a
(2) 归约项目: A → •

	Action 表				Go to 表	
	a	b	#		S	A
0	S5;R2	R2	R2		1	3
1			Accept			
2		S3				
3	R1	R1	R1			
4	R3	R3	R3			

LR(0)分析表 (没有冲突)

$V_T = \{a, b\}$

$V_N = \{S, A\}$

S = S

P:

{ (1) S→ Ab

(2) A → ε

(3) A → a

}

	Action 表				Goto 表	
	a	b	#		S	A
0	S5	R2			1	3
1			Accept			
2		S3				
3			R1			
4		R3				

冲突用follow(A)解决掉了

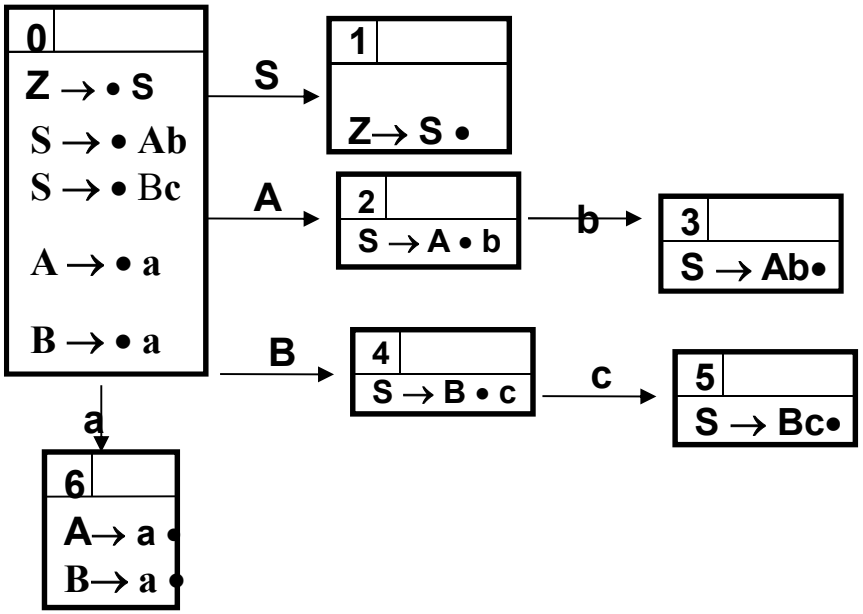
LR(0)分析表 (带有R-R 冲突)

V_T = {a, b, c}

V_N = {S, A, B}

S = S

P:
{(1) S→ Ab
(2) S→ Bc
(3) A → a
(4) B → a
}



	Action 表					Goto 表		
	a	b	c	#		S	A	B
0	S7					1	3	5
1				Accept				
2		S4						
3	R1	R1	R1	R1				
4			S6					
5	R2	R2	R2	R2				
6	R3 R4	R3 R4	R3 R4	R3 R4				

状态6存在归约-归约冲突:
(1) 归约项目: A → a •
(2) 归约项目: B → a•

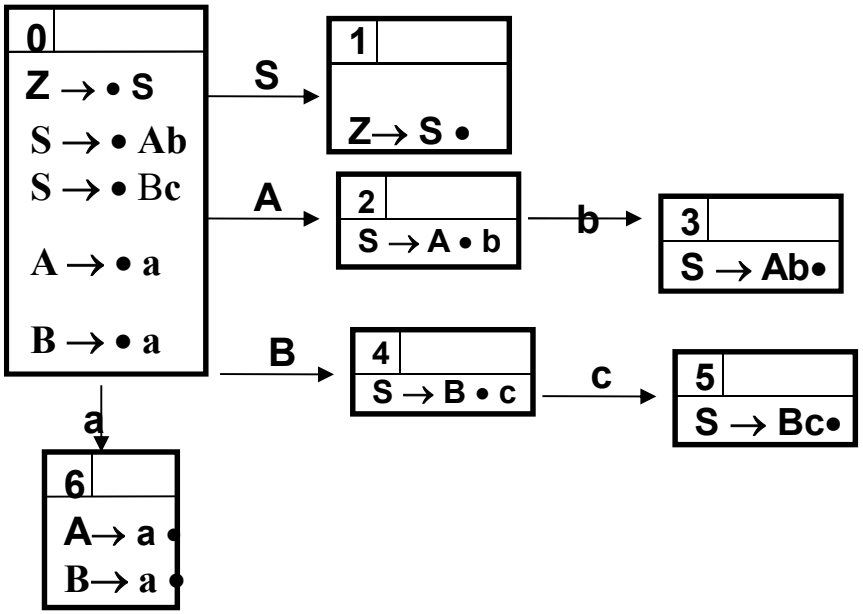
LR(0)分析表 (带有R-R 冲突)

V_T = {a, b, c}

V_N = {S, A, B}

S = S

P:
{(1) S→ Ab
(2) S→ Bc
(3) A → a
(4) B → a
}



	Action 表					Goto 表		
	a	b	c	#		S	A	B
0	S7					1	3	5
1				Accept				
2		S4						
3	R1	R1	R1	R1				
4			S6					
5	R2	R2	R2	R2				
6		R3	R4					

状态6存在归约-归约冲突:
(1) 归约项目: A → a •
(2) 归约项目: B → a•

冲突用 follow(A) 和 follow(B)消除了

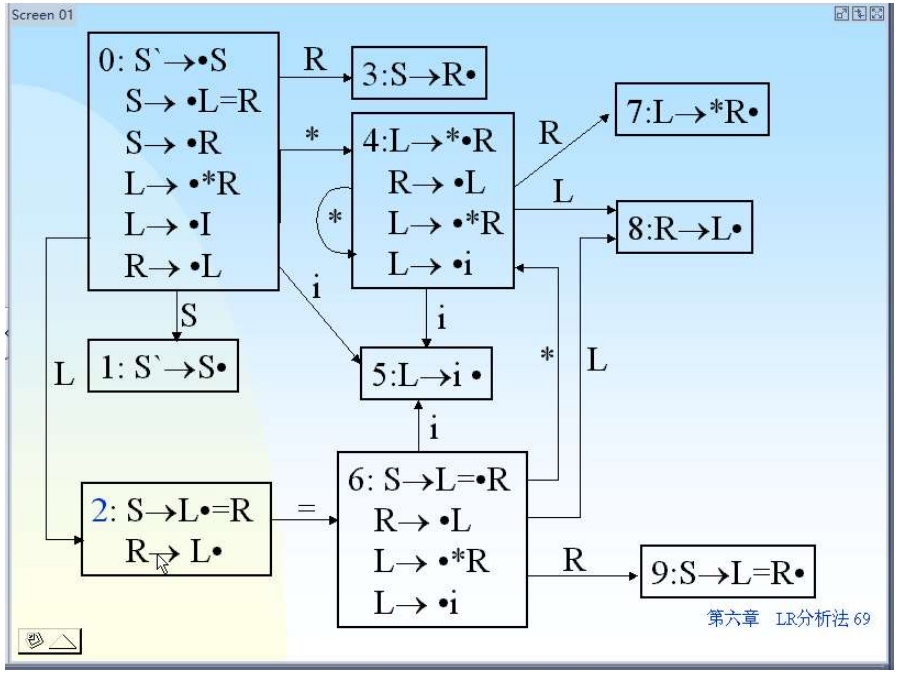
SLR(1) 分析

- **SLR(1) 分析的思想**
  - **向前看一个输入符号;**
  - **用非终极符的follow集合解决冲突;**
  - **如果能将LR(0)自动机中的所有冲突消除掉，则该文法称为*SLR(1)文法*，否则称为非SLR(1)文法。**
- **SLR(1)文法**
  - **文法G的*SLR(1)自动机*与它的LR(0)自动机完全相同**
  - ***SLR(1)分析表*中Action表与LR(0)分析表的Action表稍有不同(移入动作没有区别，归约动作有区别)**
  - ***SLR(1)驱动程序*与LR(0)驱动程序也相同**

非SLR(1)文法的例子

- 例：一个非SLR文法的例子。有如下文法:
- 1. S` →S　　2. S →L=R
- 3. S →R　　4. L →*R
- 5. L → i　　6. R →L

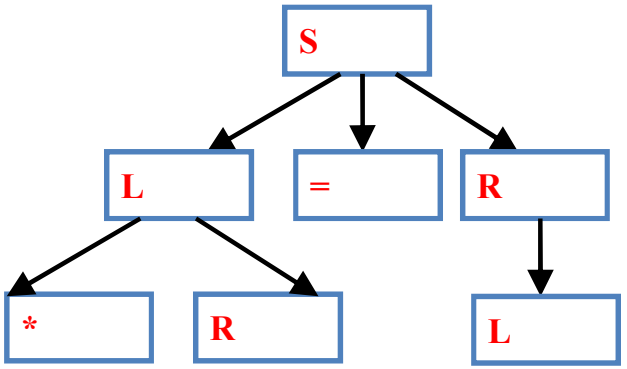
follow(R) = {#, =}



SLR(1)分析的局限性

- **SLR(1)分析存在的问题**
  - 通过引入follow集，解决了一部分文法的冲突问题，但满足SLR(1)文法条件的文法仍有限；
  - 原因在于：对于同一个非终极符可能出现在不同的位置，不同位置的后继符号(follow)是不同的，统一看待，仍有可能引起冲突。

- 例：一个非SLR文法的例子。有如下文法:
- 1. S` →S　　2. S →L=R
- 3. S →R　　4. L →*R
- 5. L → i　　6. R →L



SLR(1)分析的局限性

- 对SLR分析的思考
  - 在构造SLR分析表的方法中，若项目集I_k中含有$A\rightarrow\alpha\bullet$，那么在状态k时，只要面临 输入符号$a \in Follow(A)$，就确定采用$A\rightarrow\alpha$产生式进行归约。但是，在某种情况下，当状态k呈现于栈顶时，栈里的符号串所构成的活前缀$\beta\alpha$未必允许把α归约为A。因为可能没有一个规范句型含有前缀βAa。因此此时用$A\rightarrow\alpha$产生式进行归约未必有效。

7.3 LR(1)

解决的办法

- **结论**
  - 并非Follow集中的符号都会出现在规范句型中
- **对策**
  - 给每个LR(0)项目添加展望信息，即添加句柄之和可能跟的终结符，因为这些终结符确实是规范句型中跟在句柄之后的
  - 不要看Follow集，而是要看展望符(First集)

解决的办法

- **LR(1) 分析**
  - 基本思想:
    - 对于非终极符的每个不同出现求其*后继终极符*, 称为*展望符*;

      $S' \rightarrow \delta A\omega \rightarrow \delta\alpha\beta\omega$，如果$a \in First(\omega)$，则有 $A \rightarrow \bullet\alpha\beta$, $A \rightarrow \alpha\bullet\beta$, $A \rightarrow \alpha\beta\bullet$ 的展望符a，记$(A \rightarrow \bullet\alpha\beta, a)$, $(A \rightarrow \alpha\bullet\beta, a)$, $(A \rightarrow \alpha\beta\bullet, a)$都是有效的LR(1)项目；

      如果$a \in Follow(A)$但$a \notin First(\omega)$，a不属于$A \rightarrow \bullet\alpha\beta$, $A \rightarrow \alpha\bullet\beta$, $A \rightarrow \alpha\beta\bullet$的展望符集
    - 一个非终极符的一个出现的所有后继终极符构成的集合称为*展望符集*;
    - 示意动画（pp.243-247）

解决的办法

- **LR(1) 分析**
  - 分析步骤:
    - 构造 LR(1) 自动机
      - LR(1) 项目 = LR(0)项目+展望符集
    - 生成 LR(1)分析表 (action & goto)
    - LR(1)驱动程序 = LR(0)驱动程序

LR(1) 自动机

- **LR(1) 项目**
  - 两个部分： (A→ α•β, {a, …})
    - (1) LR(0) 项目: A→ α•β
    - (2) 展望符集: {a, …}，表示非终极符A此次出现的所有可能follow符号。
  - 例如：
    - S→ L•=R , {#}
    - A → α• , {a, b}
  - 展望符集的作用:
    - 对于移入型项目, 不起作用,但是需要保存;
    - 对于归约型项目, 表示只有当下一个输入符是其中一个展望符时, 才可以进行归约动作

LR(1) 自动机

- **LR(1) 项目集合*关于符号X的投影***
  - IS 是 LR(1) 项目的集合;
  - X 是一个符号;
  - $IS_{(X)}$ 表示项目集IS关于X的投影:
  - $IS_{(X)} = \{(S \rightarrow \alpha X \bullet \beta, ss) \mid (S \rightarrow \alpha \bullet X \beta, ss) \in IS, X \in V_T \cup V_N\}$

投影对展望符集没有影响！

- **IS = {(A → A•Bb, {a,b}), (B → a•, #), (B → b•B, {b})}**
- **X = B**
- **$IS_{(B)}$ = {(A → AB•b, {a, b}), (B → bB•, {b})}**

LR(1) 自动机

- **LR(1)项目集合的*闭包***
  - IS 是LR(1)项目的集合;
  - CLOSURE(IS)是一个LR(1)项目集合, 按照下面的步骤计算:

    [1]初始, CLOSURE(IS) = IS;

    [2] 对于CLOSURE(IS)没有处理的LR(1)项目,

    如果其形式为 $(B\rightarrow \beta\bullet A\pi, ss)$,

    而且A的全部产生式是$\{A\rightarrow\alpha 1, \ldots, A\rightarrow\alpha n\}$

    则增加如下LR(1)项目到CLOSURE(IS)

    $\{(A\rightarrow \bullet\alpha 1, ss'), \ldots, (A\rightarrow \bullet\alpha n, ss')\}$,

    其中 $ss' = first(\pi)$, 如果符号串π不导出空;

    $ss' = (first(\pi)-\{\varepsilon\}) \cup ss$, 如果符号串$\pi$导出空;

    [3] 重复[2]直到 CLOSURE(IS)收敛;

LR(1)自动机

- **goto函数()**
  - IS 是 LR(1) 项目集;
  - X 是一个符号;
  - goto(IS, X) = CLOSURE(IS $_{(X)}$)

LR(1)自动机的构造过程

- [1]增广产生式 $Z \rightarrow S$
- [2]$\Sigma = V_T \cup V_N \cup \{\#\}$
- [3]S0 = CLOSURE($\{(S' \rightarrow \bullet S, \{\#\})\}$)
- [4]ISS = {S0}
- [5]对于ISS中的每一个项目集合IS, 和每个符号$X \in \Sigma$,

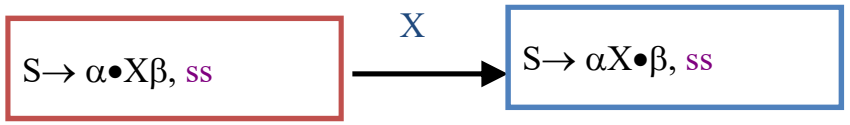
  计算IS' = goto(IS, X),

  如果IS'不为空, 则 建立 IS $\xrightarrow{X}$ IS',

  如果IS'不为空且IS'不属于ISS,则把IS'加入ISS;
- [6]重复[5]直到ISS收敛;

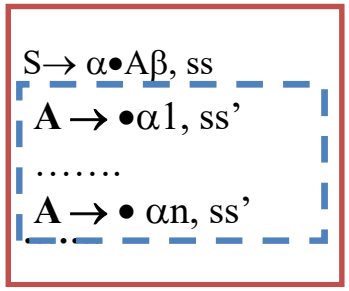
如何计算展望符集?

- **投影得到的项目**
  - **继承**



1 闭包新产生的项目

– 扩展



ss' = first(β), 如果β不导出空;

ss' = (first(β)-{ε}) ∪ss, 如果β导出空;

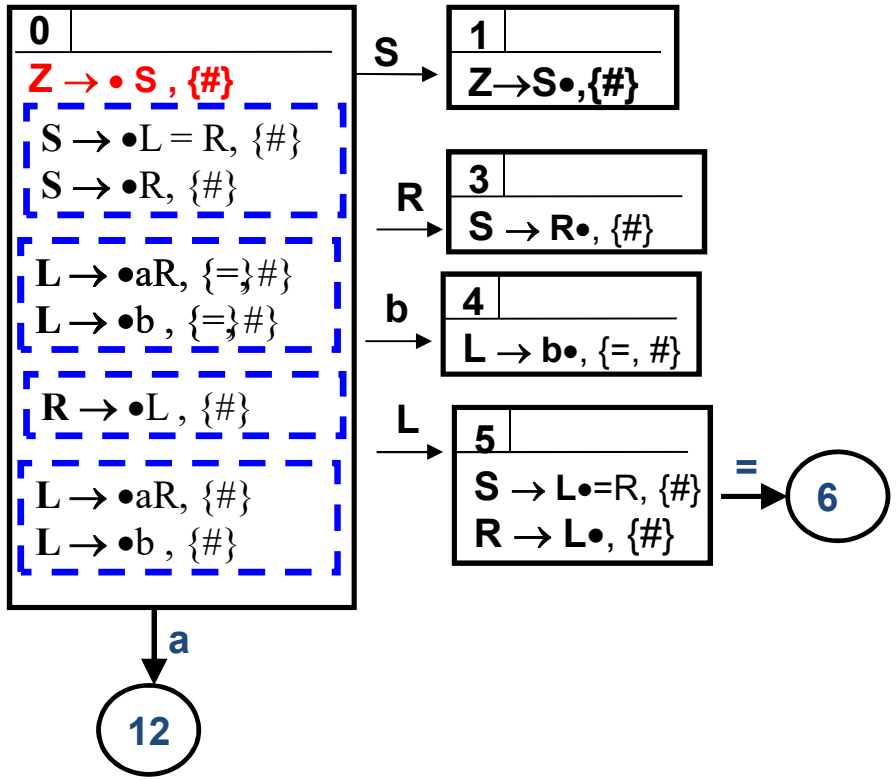
LR(1)自动机的构造

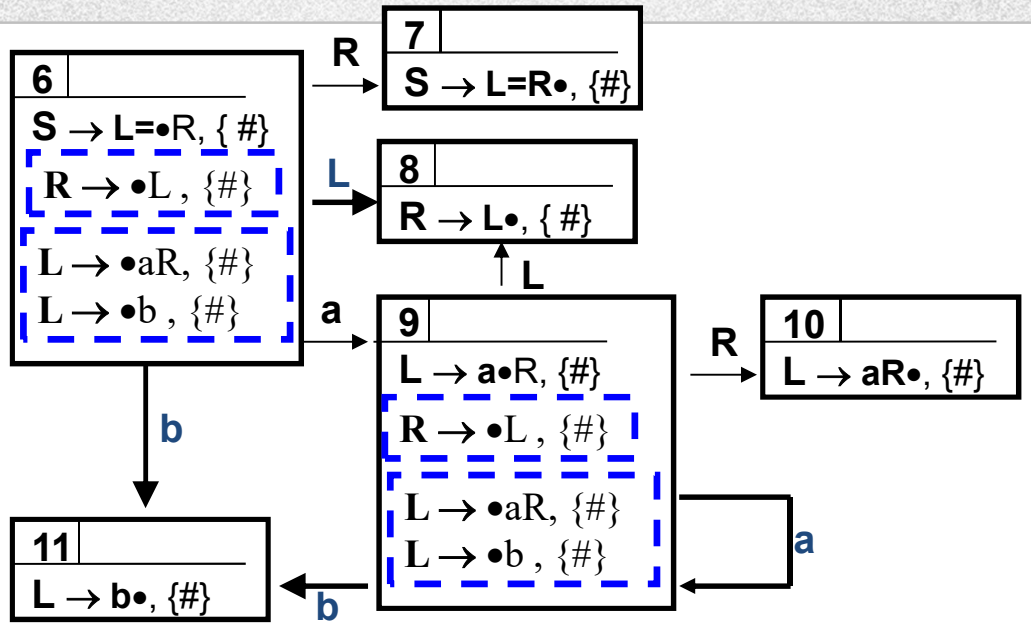
V_T = {a, b, =}

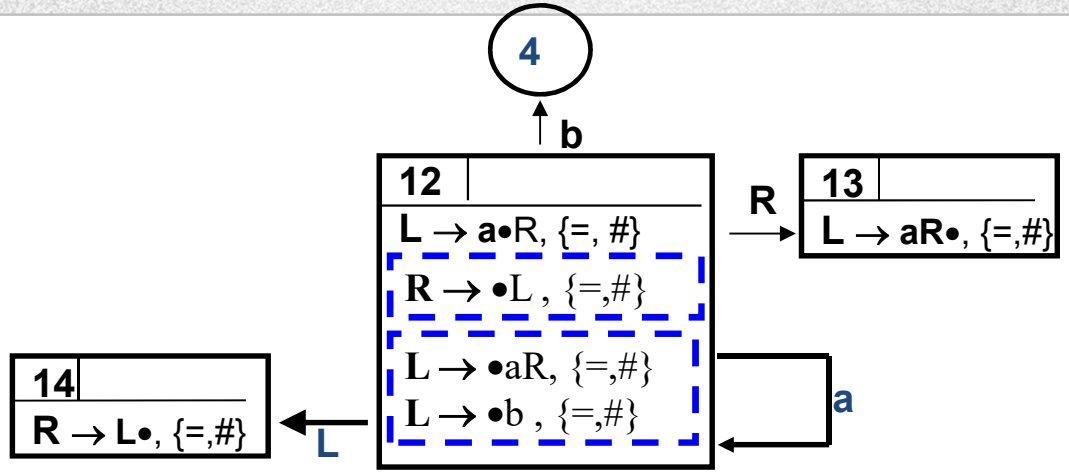
V_N = {S, L, R}

S = S

P:
{(1) S → L = R
(2) S → R
(3) L → aR
(4) L → b
(5) R → L
}




6
S → L=•R, { #}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
R
7
S → L=R•, {#}
L
8
R → L•, { #}
L
a
9
L → a•R, {#}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
R
10
L → aR•, {#}
a
b
11
L → b•, {#}
b


4
b
12
L → a•R, {=, #}
R → •L , {=,#}
L → •aR, {=,#}
L → •b , {=,#}
R
13
L → aR•, {=,#}
a
L
14
R → L•, {=,#}

LR(1) 分析表

- action表

(1)action(S_i,a) = S_j, 如果S_i到S_j有a输出边
(2)action(S_i,a) = R_p, 如果S_i中包含这样LR(1)项目,
(A→α•, ss),其中A→α是产生式P,且a∈ss;
(3)action(S_i,#) = accept, 如果S_i是接受状态
(4)action(S_i,a) = error, 其他情形

终极符 / 状 态	a_1	…	#
S_1			
…			
S_n			

LR(1) 分析表

- goto表

goto (S_i, A) = S_j, 如果S_i到S_j有A输出边
goto (S_i, A) = error,如果S_i没有A输出边

非终极符 / 状态	A_1	…	A_n
S_1			
…			
S_n			

LR(1) 分析表

	Action 表					Goto 表		
	a	b	=	#		S	L	R
0	S12	S4				1	5	3
1				Accept				
3				R2				
4			R4	R4				
5			S6	R5				
6	S9	S11					8	7
7				R1				

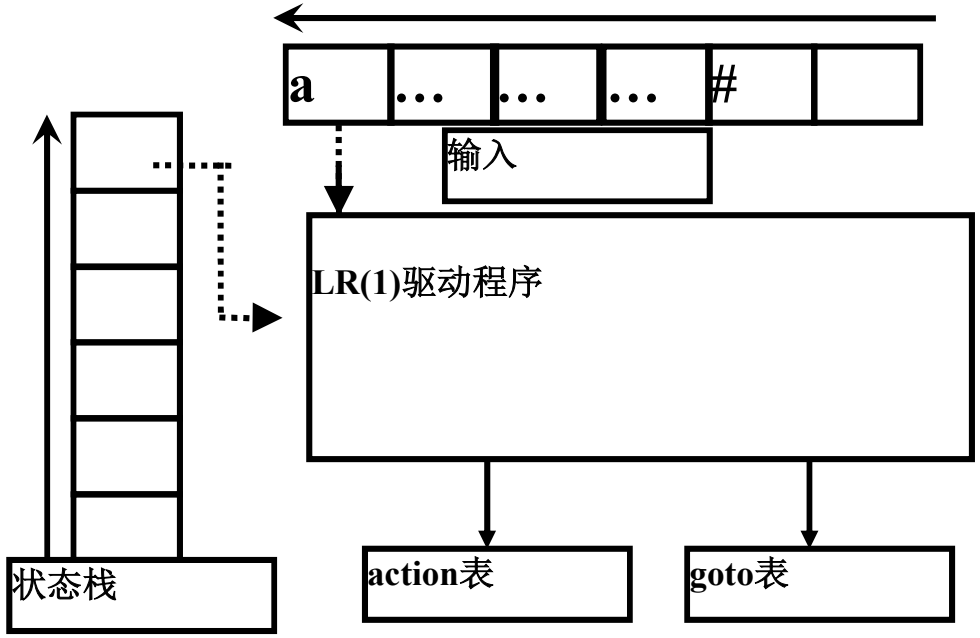
LR(1) 分析表 (接上页.)

	Action 表					Goto 表		
	a	b	=	#		S	L	R
8				R5				
9	S9	S11						10
10				R3				
11				R4				
12	S12	S4					14	13
13			R3	R3				
14			R4	R4				
15								

LR(1) 文法

- **给定一个上下文无关文法 *G***
  - LR_1 是文法G的 LR(1) 自动机
  - A_1 *是G的action表*
  - 如果对于任意一个状态s和任意的一个终极符a, A_1(s,a)只有一个唯一的动作, 则文法G称为*LR(1)文法*;
    - Shift
    - Reduce
    - Accept
    - Error

LR(1) 分析方法



LR(1) 分析驱动程序

- **初始化: push(S0); a = readOne();**
- **L: Switch action(stack(top), a)**
  - Case *error*: error();
  - Case *accept*: return true;
  - Case *Si*: push(Si), a=readOne(); goto L;
  - Case R_P: pop(|P|);
    push(goto(stack(top), P_A));
    goto L;

如何在LR分析时生成语法分析树？

LR(1)分析过程

V_T = {a, b, =}

V_N = {S, L, R}

S = S

P:
{(1) $S \rightarrow L = R$
(2) $S \rightarrow R$
(3) $L \rightarrow aR$
(4) $L \rightarrow b$
(5) $R \rightarrow L$
}

状态栈	输入流	分析动作
0	b=b#	S4
04	=b#	R4, Goto(0,L)=5
05	=b#	S6
056	b#	S11
056(11)	#	R4, Goto(6, L)=8
0568	#	R5 , Goto(6, R)=7
0567	#	R1, , Goto(0, S)=1
01	#	Accept

作业

- **教材P164: 4.6.2，4.6.3，4.6.5，4.6.6**

7.4 LALR(1)

LR(1)分析存在的问题

- 为消除冲突，引入太多的状态;
- 有些状态含有完全相同的LR(0)项目部分，只有展望符部分是不同的;
- LR(1)**项目的心**：如果(A→α•β, ss) 是一个LR(1)项目，则其中的LR(0)项目部分称为它的心;
- LR(1)自动机**状态的心**：一个状态所含有的所有LR(1)项目的心;
- **同心状态**：如果两个LR(1)状态具有相同的心，则称这两个状态为同心状态。

LALR(1) 分析

- **主要思想**
  - 合并文法G的LR(1)自动机中的同心状态，得到的自动机称为LALR(1)自动机；
  - 若这个得到的LALR(1)自动机没有冲突，则称文法G是LALR(1)文法。
- **LALR(1)分析过程**
  - 构造LALR(1)自动机
  - 构造LALR(1)分析表(同LR(1)分析表构造方法)
  - LALR(1)驱动程序 = LR(1)驱动程序

如何构造LALR(1)自动机

- **第一种途径：**
  - 首先构造LR(1)自动机
  - 然后合并其中的同心状态
  - 该方法简单，但不现实(not practical)!

Step1:构造LR(1)自动机

$V_T = \{a, b, =\}$

$V_N = \{S, L, R\}$

$S = S$

P:

{(1) $S \rightarrow L = R$

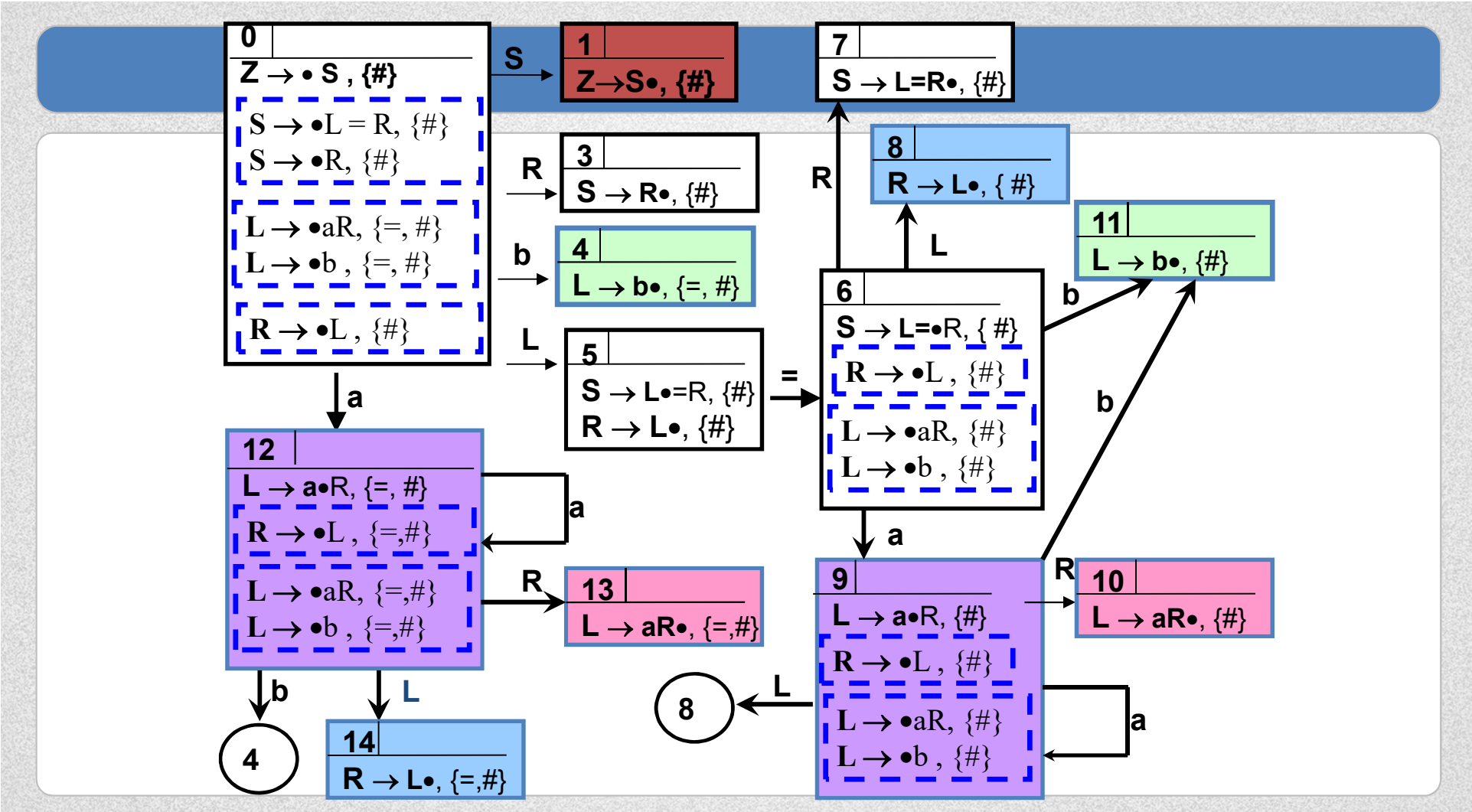
(2) $S \rightarrow R$

(3) $L \rightarrow aR$

(4) $L \rightarrow b$

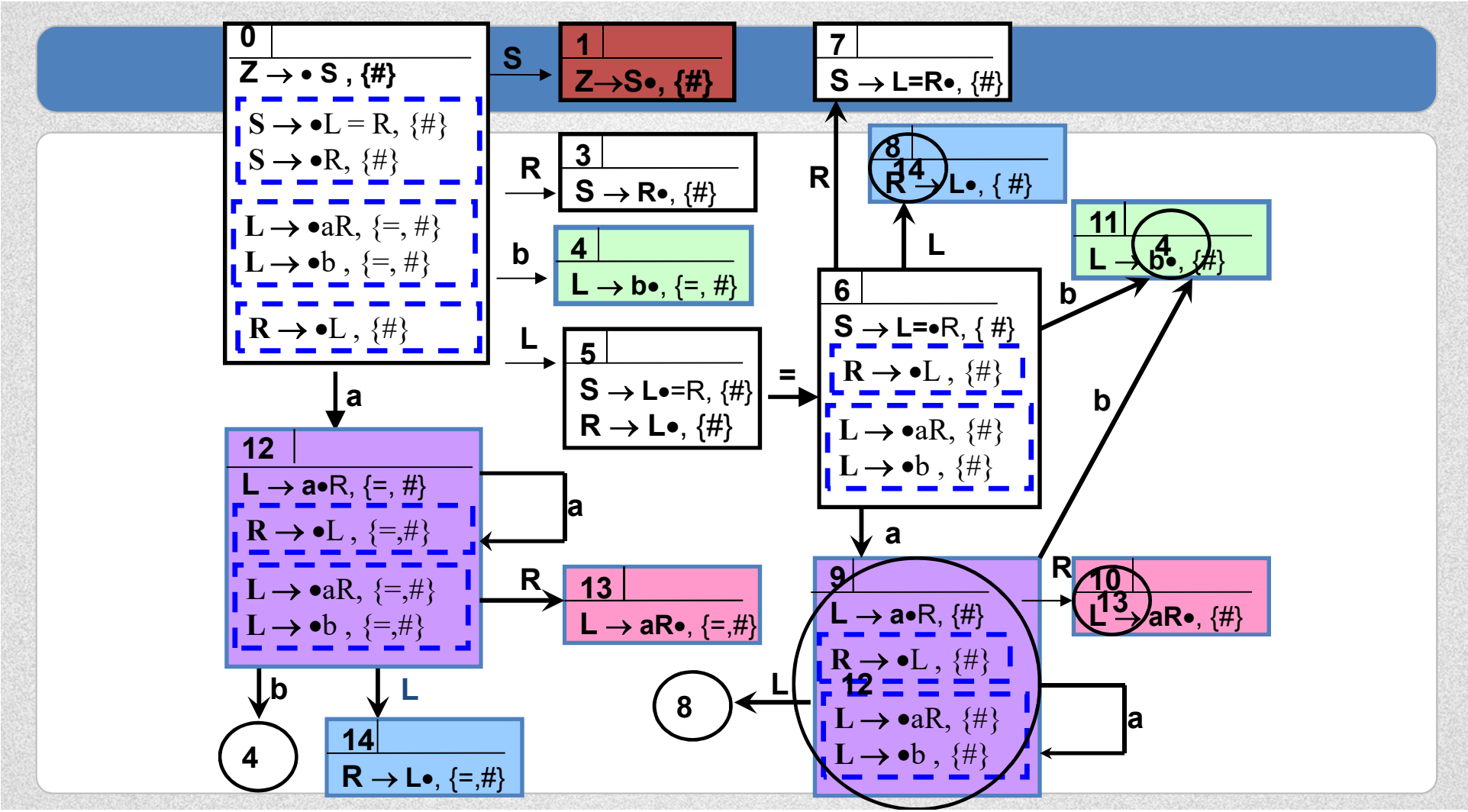
(5) $R \rightarrow L$

}


0
Z → • S , {#}
S → •L = R, {#}
S → •R, {#}
L → •aR, {=, #}
L → •b , {=, #}
R → •L , {#}
S
1
Z→S•, {#}
R
3
S → R•, {#}
b
4
L → b•, {=, #}
L
5
S → L•=R, {#}
R → L•, {#}
=
6
S → L=•R, { #}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
7
S → L=R•, {#}
R
8
R → L•, { #}
L
b
11
L → b•, {#}
b
a
12
L → a•R, {=, #}
R → •L , {=,#}
L → •aR, {=,#}
L → •b , {=,#}
a
R
13
L → aR•, {=,#}
b
L
4
14
R → L•, {=,#}
a
9
L → a•R, {#}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
R
10
L → aR•, {#}
L
8
a
b

Step2: 合并同心状态

- 状态4和11同心
- 状态8和14同心
- 状态10和13同心
- 状态9和12同心


0
Z → • S , {#}
S → •L = R, {#}
S → •R, {#}
L → •aR, {=, #}
L → •b , {=, #}
R → •L , {#}
S
1
Z→S•, {#}
R
3
S → R•, {#}
b
4
L → b•, {=, #}
L
5
S → L•=R, {#}
R → L•, {#}
=
6
S → L=•R, { #}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
7
S → L=R•, {#}
R
8
14
R → L•, { #}
L
11
4
L → b•, {#}
b
b
a
12
L → a•R, {=, #}
R → •L , {=,#}
L → •aR, {=,#}
L → •b , {=,#}
a
R
13
L → aR•, {=,#}
a
9
L → a•R, {#}
R → •L , {#}
12
L → •aR, {#}
L → •b , {#}
R
10
13
L → aR•, {#}
b
L
8
L
b
4
L
14
R → L•, {=,#}

Step2: 合并Action表和GOTO表

算法 4.59 一个简单，但空间需求大的 LALR 分析表的构造方法。

输入： 一个增广文法 G'。

输出： 文法 G' 的 LALR 语法分析表函数 ACTION 和 GOTO。

方法：

1）构造 LR(1)项集族 $C=\{I_0, I_1, \cdots, I_n\}$。

2）对于 LR(1)项集中的每个核心，找出所有具有这个核心的项集，并将这些项集替换为它们的并集。

3）令 $C'=\{J_0, J_1, \cdots, J_m\}$ 是得到的 LR(1)项集族。状态 i 的语法分析动作是按照和算法 4.56 中的方法根据 J_i 构造得到的。如果存在一个分析动作冲突，这个算法就不能生成语法分析器，这个文法就不是 LALR(1)的。

4）GOTO 表的构造方法如下。如果 J 是一个或多个 LR(1)项集的并集，也就是说 $J=I_1 \cup I_2 \cup \cdots \cup I_k$，那么 GOTO($I_1$, X)，GOTO(I_2, X)，…，GOTO(I_k, X)的核心是相同的，因为 I_1、I_2、…、I_k 具有相同的核心。令 K 是所有和 GOTO(I_1, X)具有相同核心的项集的并集，那么 GOTO(J, X)=K。 □

LR(1) 分析表

	Action 表					Goto 表		
	a	b	=	#		S	L	R
0	S12	S4				1	5	3
1				Accept				
3				R2				
4			R4	R4				
5			S6	R5				
6	S9	S11					8	7
7				R1				

LR(1) 分析表 (接上页.)

	Action 表					Goto 表		
	a	b	=	#		S	L	R
8				R5				
9	S9	S11						10
10				R3				
11				R4				
12	S12	S4					14	13
13			R3	R3				
14			R4	R4				
15								

LALR(1) 分析表

	Action 表					Goto 表		
	a	b	=	#		S	L	R
0	S12	S4				1	5	3
1				Accept				
3				R2				
4			R4	R4				
5			S6	R5				
6	S12	S4					14	7
7				R1				
12	S12	S4					14	13
13			R3	R3				
14			R4	R4				

Step2: 合并Action表和GOTO表

- **合并Action表**
  - 出错与出错合并：结果仍为出错，无冲突；
  - 移进与移进合并；
  - 出错与移进合并：不会出现此类情况，因为出错项目与移进项目不同心
  - 移进与规约合并：不会出现此类情况(不可能引入此类冲突)，因为不可能不同心
    - 例如I1：L→ i• a；I2：L→ i•a xxx, 说明合并前已有冲突

Step2: 合并Action表和GOTO表

- **合并Action表**
  - 规约与出错合并：规定它做规约
    - 由此可见，LALR与LR(1)相比，放松了报错条件，但由于移进预测能力没有减弱，所以在下一个符号进栈前总能报错，所以对错误的定位能力没有减弱
  - 规约与规约合并：
    - 按同一个产生式规约，无冲突；
    - 按不同产生式规约，将造成冲突，因此LALR能力弱于LR(1)

Step2: 合并Action表和GOTO表

- **合并时发生规约-规约冲突，例如：语言{acd，ace，bcd，bce}**
  - 可行前缀ac的有效项集{[A⟶c ·,d],[B⟶c ·,e]}
  - 可行前缀bc的有效项集{[A⟶c ·,e],[B⟶c ·,d]}
  - 合并之后的项集为{[A⟶c ·,d/e],[B⟶c ·,d/e]}
  - 当输入为d或e时，用哪个归约？

$$S' \rightarrow S$$
$$S \rightarrow a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e$$
$$A \rightarrow c$$
$$B \rightarrow c$$

综上所述，可得：只要合并同心之后，不存在按不同产生式的规约-规约冲突，由LR(1)项目集族总能构造成LALR分析表

Step2: 合并Action表和GOTO表

- **合并GOTO表**
  - 直接合并，无冲突，因为一定同心项目的GOTO表指向的项目也同心

LALR(1)构造示例2

- **示例**

LALR(1) 自动机

- **对于给定的上下文无关文法 *G***
  - G的LALR(1)项目跟LR(1)项目形式相同;
  - LALR(1)自动机中每个状态S中各个项目的展望符集是把LR(1)自动机中所有和S同心的状态的对应项目的展望符集合合并后得到的;
  - 如果每个LALR(1)的状态都用该状态的心(LR(0)项目)替换，则LALR(1)自动机和它的LR(0)自动机相同;
  - G的LALR(1) 自动机的状态数同LR(0)自动机的状态数相同；
  - 能不能用LR(0)自动机构造LALR(1)自动机呢？

如何构造LALR(1)自动机

- **第二种途径：**
  - 首先构造LR(0)自动机
  - 然后为每个状态的每个LR(0)项目计算展望符集；
  - 是实际应用中采用的方法！
  - 关键是如何计算展望符（向前看符号）呢？

如何构造LALR(1)自动机

- **第二种途径：**

算法 4.62 确定向前看符号。

输入：一个 LR(0) 项集 I 的内核 K 以及一个文法符号 X。

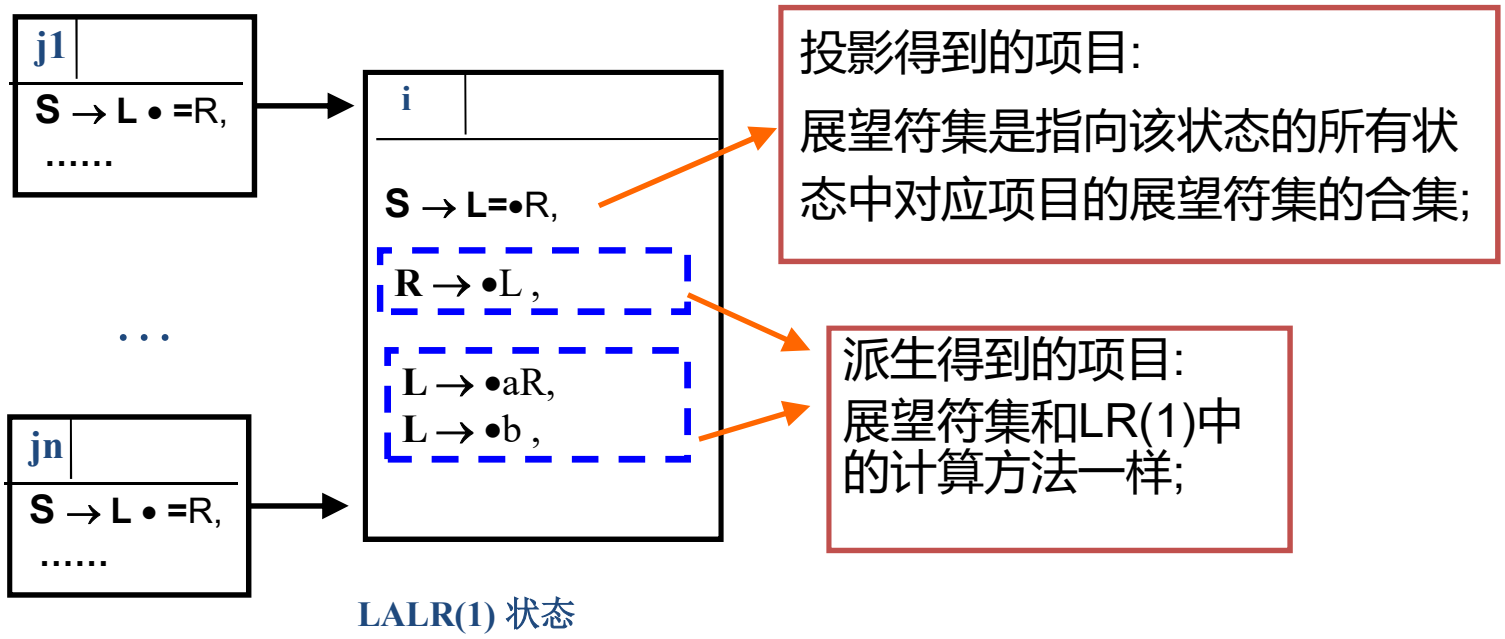
输出：由 I 中的项为 GOTO(I, X) 中内核项自发生成的向前看符号，以及 I 中将其向前看符号传播到 GOTO(I, X) 中内核项的项。

方法：算法在图 4-45 中给出。 □

```
for ( K 中的每个项 A → α·β ) {
    J := CLOSURE({[A → α·β,#]} );
    if ( [B → γ·Xδ,a] 在 J 中，并且 a 不等于 # )
        断定 GOTO(I,X) 中的项 B → γX·δ 的向前看符号 a
        是自发生成的；
    if ( [B → γ·Xδ,#] 在 J 中 )
        断定向前看符号从 I 中的项 A → α·β 传播到了 GOTO(I,X) 中的项
        B → γX·δ 之上；
}
```

图 4-45 发现传播的和自发生成的向前看符号

LALR(1)展望符的计算方法



LALR(1) 状态

构造LALR(1) 自动机

V_T = {a, b, =}

V_N = {S, L, R}

S = S

P:

{(1) $S \rightarrow L = R$

(2) $S \rightarrow R$

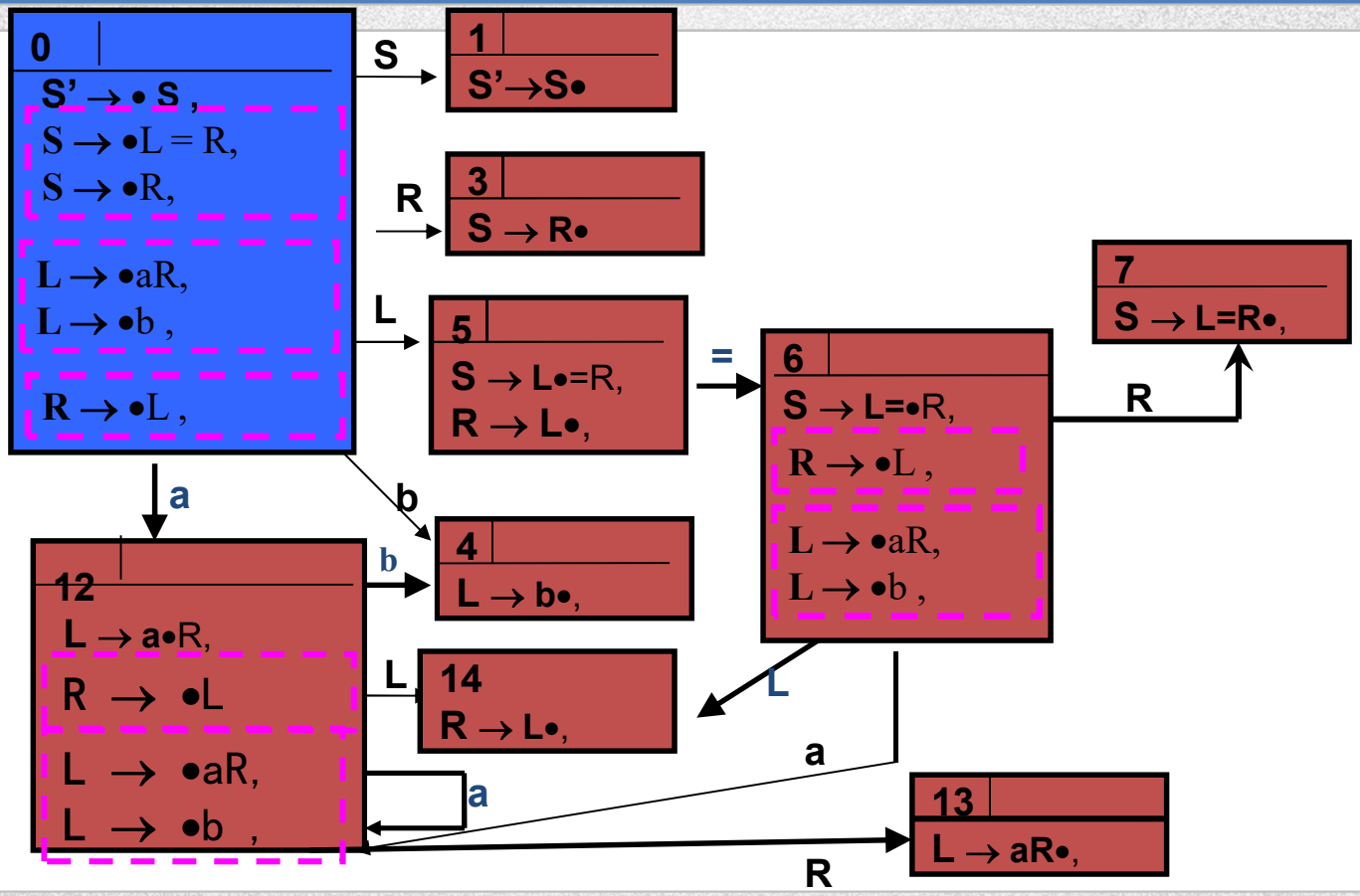
(3) $L \rightarrow aR$

(4) $L \rightarrow b$

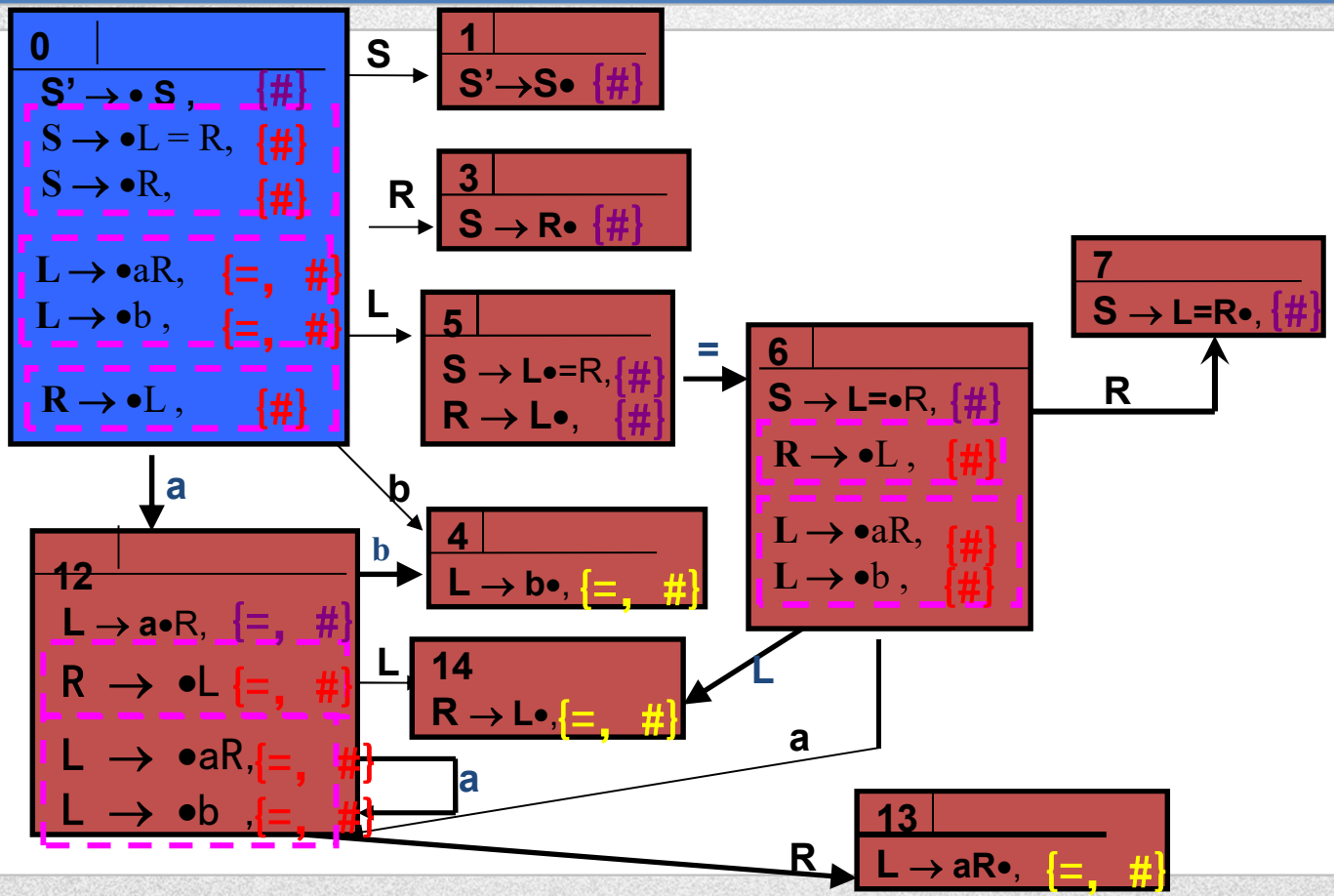
(5) $R \rightarrow L$

}

Step1: 构造LR(0) 自动机



Step2:计算展望符


0
S' → • S , {#}
S → •L = R, {#}
S → •R, {#}
L → •aR, {=, #}
L → •b , {=, #}
R → •L , {#}
1
S'→S• {#}
3
S → R• {#}
5
S → L•=R, {#}
R → L•, {#}
6
S → L=•R, {#}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
7
S → L=R•, {#}
4
L → b•, {=, #}
12
L → a•R, {=, #}
R → •L {=, #}
L → •aR, {=, #}
L → •b , {=, #}
14
R → L•, {=, #}
13
L → aR•, {=, #}
S
R
L
=
R
a
b
b
L
L
a
a
R

合并可能产生的冲突

- 如果文法G的LALR(1) 自动机中没有冲突,则文法G称为*LALR(1) 文法*;
- 合并引起的冲突是指：**本来的LR(1)项集没有冲突**，而合并具有相同核心的项集后有冲突
- 合并同心状态只能产生归约-归约冲突，不会产生移入-归约冲突---已经解释过了

LALR分析器和LR分析器

- **若构造出的LALR分析表中没有冲突，则可以用LALR分析表进行语法分析**
  - 如果是正确的输入串
    - LALR和LR分析器执行完全相同的移入和归约动作序列，只是栈中的状态名字有所不同
  - 如果是错误的输入串
    - 则LALR分析器可能会比LR分析器晚一点报错，定位一样，能力一样
    - LALR分析器不会在LR分析器报错后再移入任何符号，但是可能会继续执行一些归约

LALR分析器和LR分析器

- “i=*i=#” 的LR(1)分析过程

步骤	状态栈	符号栈	输入串
0	0	#	i=*i=#
1	0，5	#i	=*i=#
2	0，2	#L	=*i=#
3	0，2，6	#L=	*i=#
4	0，2，6，12	#L=*	i=#
5	0，2，6，12，10	#L=*i	=#
	报错		

LALR分析器和LR分析器

“i=*i=#” 的LALR(1)分析过程

步骤	状态栈	符号栈	输入串
0	0	#	i=*i=#
1	0，5	#i	=*i=#
2	0，2	#L	=*i=#
3	0，2，6	#L=	*i=#
4	0，2，6，4	#L=*	i=#
5	0，2，6，4，5	#L=*i	=#
6	0，2，6，4，8	#L=*L	=#
7	0，2，6，4，7	#L=*R	=#
8	0，2，6，8	#L=L	=#
9	0，2，6，9	#L=R	=#
	报错		

LR(1)方法遇到输入串有错立即报错，LALR没有立即报错，而是多做了几步规约后才报错，但他们对错误定位能力相同，故LALR报错能力并没有减弱。

LALR分析器和LR分析器

识别能力

- $L(G) = \{c^m d c^n d \mid m, n \in \mathbb{N}\}$.
- LALR 和 LR(1) 分析法均接受正确的输入：c*dc*d.
- LR(1)DFA 的状态 I_4 接受第一个 d，状态 I_7 接受第二个 d.
- 合并后状态 I_{47} 对应的项目 $[C \to d\bullet, \$]$ 对活前缀 cd 不再是有效项目.
- 如果输入是 cd，LR(1) 分析法将在状态 I_4，面对输入 \$，分析器将立刻报错.
- 如果输入是 cd，LALR 分析法将在状态 I_{47}，面对输入 \$，分析器将在两步归约操作后报错.

文法 G

1/ $S \to CC$ 2/ $C \to cC$ 3/ $C \to d$

LR(1) 分析法

stack			action	rest
I_0	I_3	I_4	error	\$
\$	c	d		

LALR 分析法

stack			action	rest
I_0	I_{36}	I_{47}	reduce	\$
\$	c	d		
I_0	I_{36}	I_{89}	reduce	\$
\$	c	C		
I_0	I_2		error	\$
\$	C			

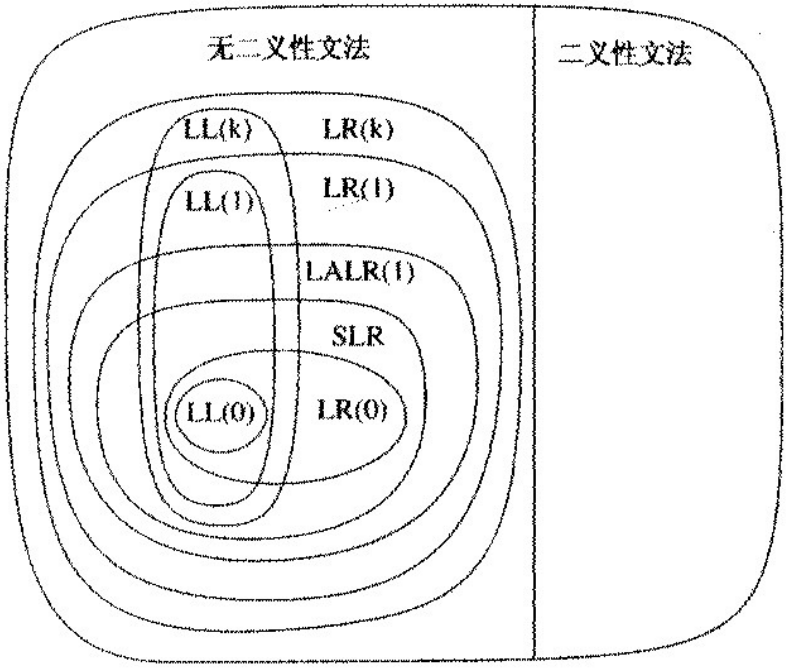
综合比较

- **状态数:**
  - 对LR(1)项集规范族中所谓的同心项集进行合并，从而使得分析表既保持了LR(1)项中向前看符号的信息，又使状态数减少到与SLR分析表的一样多
  - LR(1) > LALR(1) = SLR(1) = LR(0)
- **展望符的确定:**
  - LR(0)没有展望符；
  - SLR(1)取follow集；
  - LR(1)取不同位置的follow集
  - LALR(1)取同心项的展望符的并集；
- **向前看输入符:**
  - SLR(1),LR(1)和LALR(1)向前看一个输入符；
  - LR(0)不看;
- **分析能力: LR(1) ⊃ LALR(1) ⊃ SLR(1) ⊃ LR(0)**

综合比较

- 如果原来LR(1) DFA规范项目集中没有移进-规约冲突，则LALR(1)项目集不可能有移进-规约冲突；即，如果LALR(1)项目集有移进-规约冲突，则LR(1) 一定也有相同的S-R冲突
- 但是LALR(1)可能出现R-R冲突
- LALR分析表移进操作与LR(1)和SLR一样，唯一不同的在于归于操作
- LALR(1) 可能解决SLR的S-R冲突，例如复制表达式文法是LALR文法，而不是SLR文法 --- 绝大多数程序语言是LALR文法

综合比较



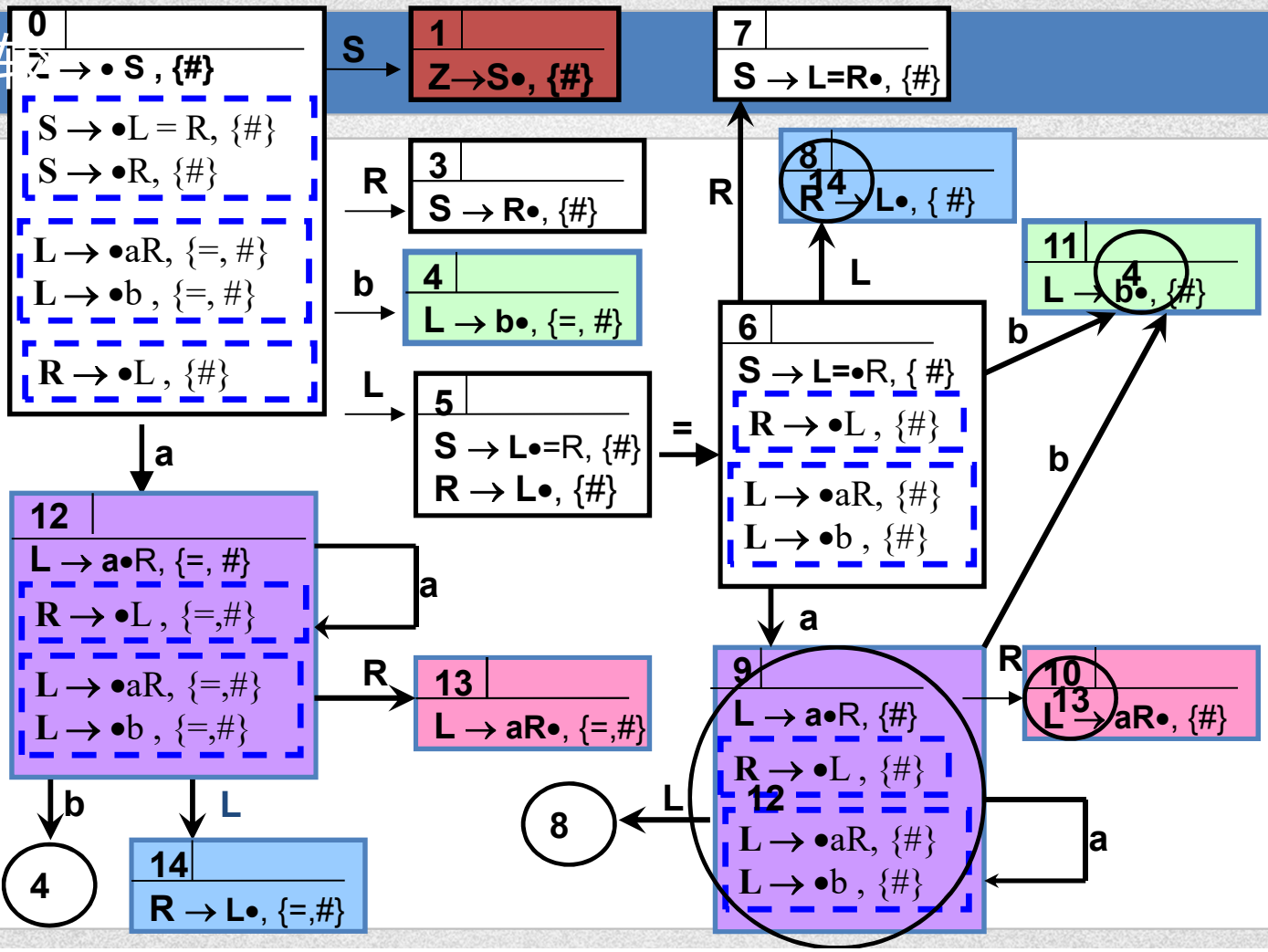
LR(1)-LALR(1) = 规约-规约冲突的情况

综合比较

Example of LALR gammar, but not SLR grammar

- $$G: \begin{cases} S \rightarrow Aa \mid bAc \mid dc \mid bda \\ A \rightarrow d \end{cases}$$
- LR(0) 规范项目集中有：$I = \{S \rightarrow d \bullet c,\ A \rightarrow d\bullet\}$，并且，$c \in Follow(A)$，所以有移进 - 归约冲突.
- LALR 项目集中对应的是：$I = \{[S \rightarrow d \bullet c, \$],\ [A \rightarrow d\bullet, a]\}$，避免了上述冲突，所以是 LALR 文法.

综合比较


0
Z → • S , {#}
S → •L = R, {#}
S → •R, {#}
L → •aR, {=, #}
L → •b , {=, #}
R → •L , {#}
S
1
Z→S•, {#}
R
3
S → R•, {#}
b
4
L → b•, {=, #}
L
5
S → L•=R, {#}
R → L•, {#}
=
6
S → L=•R, { #}
R → •L , {#}
L → •aR, {#}
L → •b , {#}
7
S → L=R•, {#}
R
8
14
R → L•, { #}
L
11
4
L → b•, {#}
b
a
12
L → a•R, {=, #}
R → •L , {=,#}
L → •aR, {=,#}
L → •b , {=,#}
a
R
13
L → aR•, {=,#}
b
L
4
14
R → L•, {=,#}
a
9
L → a•R, {#}
R → •L , {#}
12
L → •aR, {#}
L → •b , {#}
R
10
13
L → aR•, {#}
b
L
8
a

综合比较

Example of LR(1) gammar, but not LALR grammar

- $$G: \begin{cases} S \rightarrow aAa \mid bBb \mid aBb \mid bAa \\ A \rightarrow c \\ B \rightarrow c \end{cases}$$
- LR(1) 规范项目集中有：$I = \{[A \rightarrow c\bullet, a], [B \rightarrow c\bullet, b]\}$，
  $I = \{[A \rightarrow c\bullet, b], [B \rightarrow c\bullet, a]\}$，没有冲突.
- 合并后，LALR 项目集中对应的是：
  $IJ = \{[A \rightarrow c\bullet, a/b], [B \rightarrow c\bullet, a/b]\}$，产生了 R-R 冲突.

综合比较

Example of LL(1) gammar, but not SLR grammar

- $$G: \begin{cases} S \rightarrow AaAb \mid BbBa \\ A \rightarrow \varepsilon \\ B \rightarrow \varepsilon \end{cases}$$
- G 是 LL(1) 文法.
- G 不是 SLR 文法，存在规范 LR(0) 项目集：$I = \{S \rightarrow \bullet AaAb, S \rightarrow \bullet BbBa, A \rightarrow \bullet, B \rightarrow \bullet\}$，有两个归约项目 $A \rightarrow \bullet, B \rightarrow \bullet$，而 $a, b \in Follow(A) = Follow(B)$，从而产生 R-R 冲突.
- 还存在 LL(1) 文法，但不是 LALR 文法.

作业

- **教材P177: 4.7.3，4.7.4，4.7.5**

Thank you!