第4章 语法分析（下）

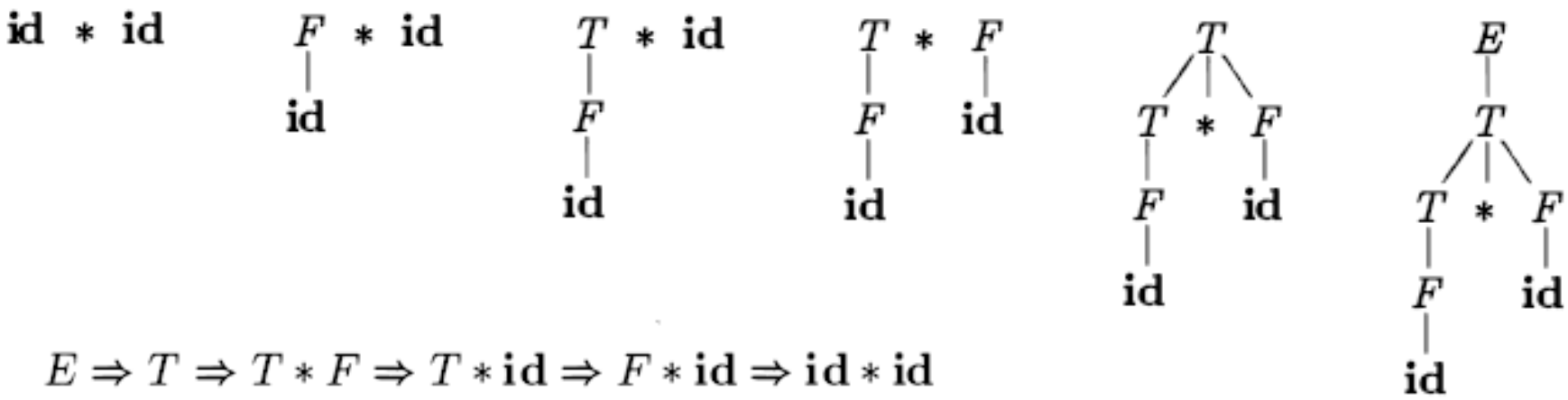
西北大学信息学院计算机科学系 付丽娜

自底向上的语法分析

自底向上的语法分析

- 自底向上的语法分析过程
  - 对应于为一个输入串构造语法分析树的过程
  - 从叶子结点（底部）开始逐渐向上到达根结点（顶部）
  - 例：*id*id*的自底向上语法分析过程

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$



$$E \Rightarrow T \Rightarrow T * F \Rightarrow T * \mathbf{id} \Rightarrow F * \mathbf{id} \Rightarrow \mathbf{id} * \mathbf{id}$$

归约

- 归约（*reduction*）
  - 自底向上语法分析过程可以看作是将一个串*w*归约为文法符号的过程
  - 在每个归约步骤中，一个与某产生式右部匹配的特定子串被替换为该产生式左部的非终结符号
  - 一次归约是一个推导步骤的逆操作，自底向上语法分析的目标是反向构造一个推导过程
- 自底向上语法分析的关键问题
  - 何时进行归约
  - 应用哪个产生式进行归约

自底向上规约中的问题

例：

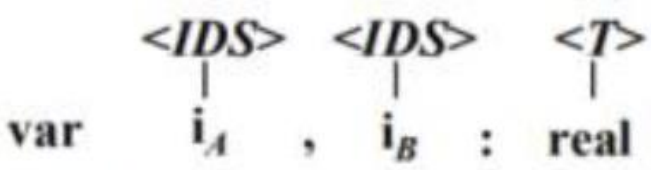
(1) <*S*>→var <*IDS*> : <*T*>

(2) <*IDS*>→i

(3) <*IDS*>→<*IDS*> , i

(4) <*T*>→real | int

栈	剩余输入	动作
$	var i_A, i_B : real $	
$ var	i_A, i_B : real $	移入
$ var i_A	, i_B : real $	移入
$ var <*IDS*>	, i_B : real $	归约
$ var <*IDS*> ,	i_B : real $	移入
$ var <*IDS*> , i_B	: real $	移入
$ var <*IDS*> , <*IDS*>	: real $	归约
$ var <*IDS*> , <*IDS*> :	real $	移入
$ var <*IDS*> , <*IDS*> : real	$	移入
$ var <*IDS*> , <*IDS*> :<*T*>	$	归约



自底向上规约中的问题（CONT）

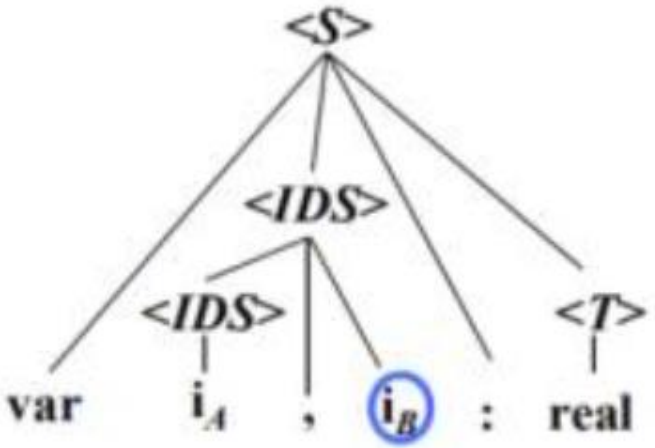
例：

(1) <S>→var <IDS> : <T>

(2) <IDS>→i

(3) <IDS>→<IDS> , i

(4) <T>→real | int



造成错误的原因：
错误地识别了句柄

栈	剩余输入	动作
$	var i_A, i_B : real $	
$ var	i_A, i_B : real $	移入
$ var i_A	, i_B : real $	移入
$ var <IDS>	, i_B : real $	归约
$ var <IDS> ,	i_B : real $	移入
$ var <IDS> , i_B	: real $	移入
$ var <IDS>	: real $	归约
$ var <IDS> :	real $	移入
$ var <IDS> : real	$	移入
$ var <IDS> :<T>	$	归约
$ <S>	$	归约

句柄：句型的最左直接短语

句柄

- 归约和句柄
  - 从左向右扫描输入串，并在扫描过程中进行自底向上的语法分析，就可以反向构造出一个最右推导
  - 句柄是和某个产生式右部匹配的子串
  - 但是当栈顶出现某个产生式的右部时，不一定就是句柄

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

句柄

- 短语、直接短语与句柄
  - 设 $\alpha\beta\delta$ 为G 的一个句型 ($S \overset{*}{\Rightarrow} \alpha\beta\delta$)，若有

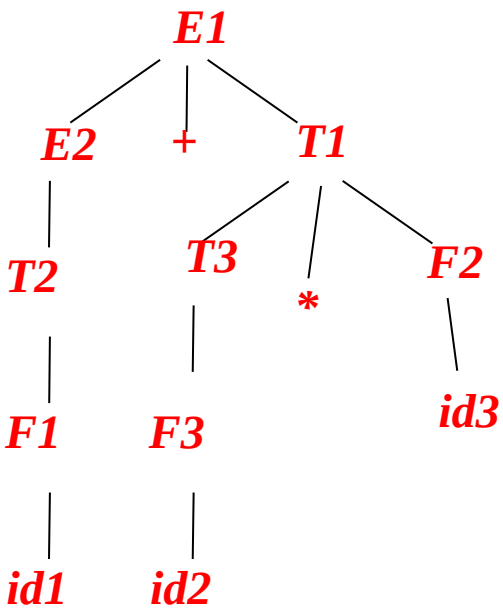
    $$S \overset{*}{\Rightarrow} \alpha A\delta \text{ 且 } A \overset{+}{\Rightarrow} \beta$$

    则称 β 是句型 $\alpha\beta\delta$ 相对于非终结符 A 的**短语**
  - 特别是，若有 $A \Rightarrow \beta$，

    则称β 是句型 $\alpha\beta\delta$ 相对于规则 $A \rightarrow \beta$的**直接短语**
  - 一个句型的最左直接短语称为该句型的**句柄**
  - *注意，作为短语的两个条件缺一不可*
    - 仅$A \overset{+}{\Rightarrow} \beta$不能说明$\beta$ 是句型 $\alpha\beta\delta$ 相对于 A 的短语
    - 还必须有 $S \overset{*}{\Rightarrow} \alpha A\delta$

句柄

- 为了找出一个句型的所有短语，可以构造语法分析树
  - 句型*id1+id2*id3*的短语、直接短语和句柄
- 语法树中的叶子与短语、直接短语和句柄有以下关系
  - 短语：以非终结符为根的子树中所有从左到右的叶子
    - *id1，id2，id3，id2 * id3，id1 + id2 * id3*
  - 直接短语：若子树仅为两代，则该短语为直接短语
    - *id1，id2，id3*
  - 句柄：最左仅为两代的子树的直接短语为句柄
    - id_1



句柄剪枝

- 句柄的特点
  - 句柄右边的串w一定只包含终结符号
  - 如果一个文法是无二义的，那么该文法的每个最右句型都有且只有一个句柄
- 句柄剪枝
  - 从被分析的终结符号串w开始，如果w是当前文法的句子，那么令$w=\gamma_n$，其中γ_n是某个未知最右推导的第n个最右句型
  - 为了以相反顺序重构这个推导，我们在γ_n中寻找句柄β_n，并将β_n替换为相关产生式$A_n \rightarrow \beta_n$的左部，得到前一个最右句型γ_{n-1}

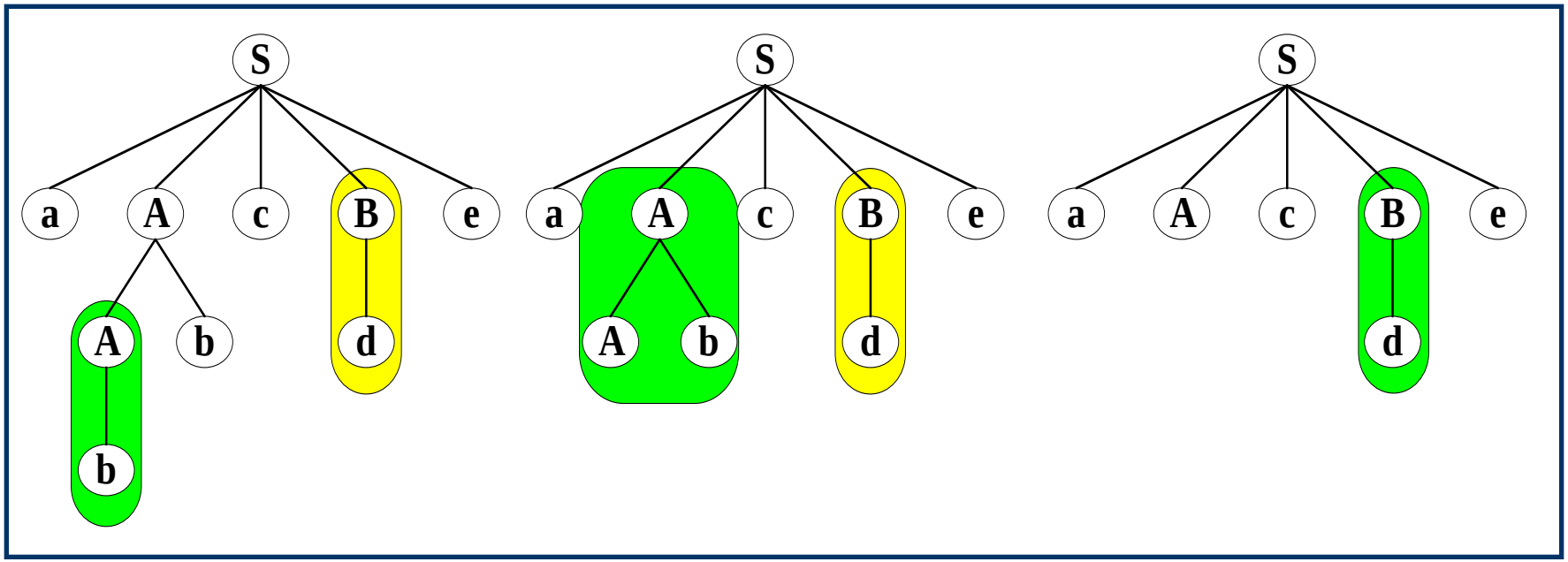
$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

例……句柄剪枝

- 例： 文法G如下：

  $S \rightarrow aAcBe \qquad A \rightarrow b \mid Ab \qquad B \rightarrow d$

  输入串$w = abbcde$，句柄剪枝过程如图



移入-归约语法分析技术

- 移入-归约语法分析是自底向上语法分析的通用框架
  - 用一个栈保存文法符号
  - 用一个输入缓冲区存放将要进行语法分析的其余符号
  - 用$标记栈底和输入串右端
- 移入-归约
  - 在对输入串的一次从左向右扫描过程中，语法分析器将输入符号移进栈的顶端，直到可以对栈顶的一个文法符号串β进行归约时为止（可规约串）
  - 语法分析器将β归约为某个产生式的左部非终结符
  - 重复上述过程，直到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时停止

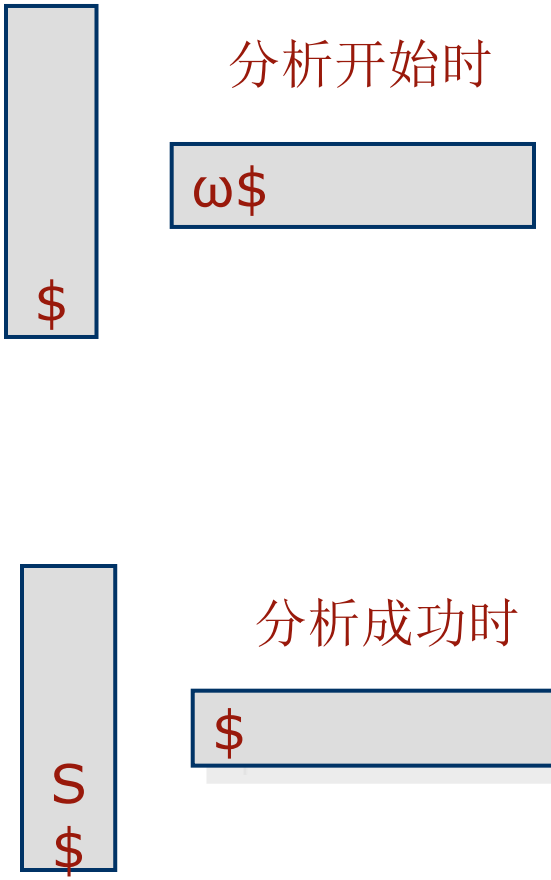
移入-归约语法分析技术

- 栈和输入缓冲区的布局
  - 分析开始时

栈	输入串
$	ω$

  - 分析成功时（否则ω有语法错误）

栈	输入串
$ S	$



例……移入-归约语法分析技术

❖ 例：输入串id_1*id_2的移入-归约过程

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

STACK	INPUT	ACTION
\$	$\mathbf{id}_1 * \mathbf{id}_2$ \$	shift
\$ $\mathbf{id}_1$	$* \mathbf{id}_2$ \$	reduce by $F \rightarrow \mathbf{id}$
\$ F	$* \mathbf{id}_2$ \$	reduce by $T \rightarrow F$
\$ T	$* \mathbf{id}_2$ \$	shift
\$ $T *$	$\mathbf{id}_2$ \$	shift
\$ $T * \mathbf{id}_2$	\$	reduce by $F \rightarrow \mathbf{id}$
\$ $T * F$	\$	reduce by $T \rightarrow T * F$
\$ T	\$	reduce by $E \rightarrow T$
\$ E	\$	accept

移入-归约语法分析技术

- 移入-归约语法分析器的动作
  - 移入（*shift*）将下一个输入符号移进栈的顶端
  - 归约（*reduce*）被归约的符号串（句柄）的右端必然是栈顶，语法分析器在栈中确定这个串的左端，并决定用哪个非终结符号来替换这个串
  - 接受（*accept*）宣布语法分析过程成功完成
  - 报错（*error*）发现一个语法错误，并调用错误恢复例程
- 移入-归约语法分析过程的性质
  - 句柄总是出现在栈的顶端，绝不会出现在栈的中间（对应最左端的二代子树）

移入-归约语法分析中的冲突

- 有些上下文无关文法不能使用移入-归约语法分析技术
  - 移入/归约冲突
    - 即使知道了栈中的所有内容(历史与当下）以及接下来k个输入符号(未来），分析器仍然无法判断应该进行移入还是归约
  - 归约/归约冲突
    - 无法在多个可能的归约方法中选择正确的归约动作
- LR(k)文法
  - 在分析过程中，在输入中至多向前看k个符号，就能确定分析器的动作
  - 存在一些文法不是LR(k)文法，即无论k取多大的值，都无法消除冲突

移入/归约冲突

- 二义性文法不是LR的
  - 经过修正的移入-归约语法分析技术可以对某些二义性文法进行语法分析

stmt	→	**if** *expr* **then** *stmt*
	|	**if** *expr* **then** *stmt* **else** *stmt*
	|	**other**

如果我们有一个移入－归约语法分析器处于格局

栈	输入
··· **if** *expr* **then** *stmt*	**else** ··· $

中，那么不管栈中 **if** *expr* **then** *stmt* 之下是什么内容，我们都不能确定它是否是句柄。这里就出现了一个移入/归约冲突。

根据输入中 **else** 之后的内容的不同，可能应该将 **if** *expr* **then** *stmt* 归约为 *stmt*，也可能应该将 **else** 移入然后再寻找另一个 *stmt*，从而找到完整的 *stmt* 产生式体 **if** *expr* **then** *stmt* **else** *stmt*。

归约/归约冲突

- 在已经确认找到了句柄的时候，不能根据栈中内容和下一个输入符号确定应该使用哪个产生式归约

	栈	输入
1	… id (id	, id) …

(1) *stmt* → id (*parameter_list*)
(2) *stmt* → *expr* := *expr*
(3) *parameter_list* → *parameter_list* , *parameter*
(4) *parameter_list* → *parameter*
(5) *parameter* → id
(6) *expr* → id (*expr_list*)
(7) *expr* → id

一个以 p(i, j)开头的语句将以词法单元流 id(id, id)的方式输入到语法分析器中。在将前三个词法单元移入到栈中后，移入－归约语法分析器将处于格局1中。显然，栈顶的 id 必须被归约，但使用哪个产生式呢？如果 p 是一个过程，那么正确的选择是产生式(5)；但如果 p 是一个数组，就该选择产生式(7)。栈中的内容并没有指出 p 是什么，必须使用从 p 的声明中获得的符号表中的信息来确定。

练习4.5

练习 4.5.1：对于练习 4.2.2(a)中的文法 $S \rightarrow 0\ S\ 1 \mid 0\ 1$，指出下面各个最右句型的句柄：

1) 000111　　2) 00*S*11

练习 4.5.2：对于练习 4.2.1 的文法 $S \rightarrow S\ S\ +\ \mid S\ S\ *\ \mid a$ 和下面各个最右句型，重复练习 4.5.1。

1) $SSS + a *\ +$

2) $SS + a * a +$

3) $aaa * a + +$

练习 4.5.3：对于下面的输入符号串和文法，说明相应的自底向上语法分析过程。

1) 练习 4.5.1 的文法的串 000111。

2) 练习 4.5.2 的文法的串 $aaa * a + +$。

简单LR技术

LR语法分析技术介绍

LR语法分析技术介绍

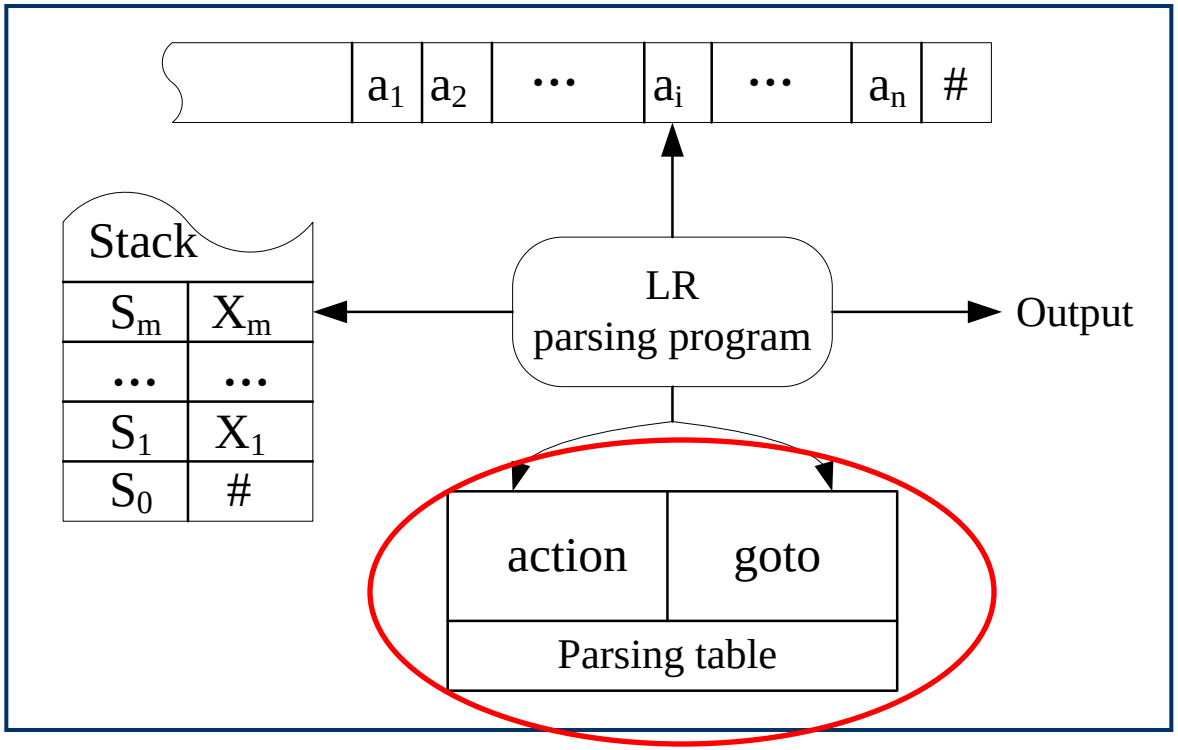
- LR(*k*)语法分析
  - “*L*”表示从左向右扫描输入串，“*R*”表示反向构造一个最右推导序列，*k*表示在语法分析决策时向前看*k*个符号
    - *k*<=*1*时有实用价值
  - LR(0)，SLR，LR(1)，LALR
- LR语法分析器是表驱动的
  - 如果能为一个文法构造出一个不含冲突的LR分析表，那么该文法就是LR文法

为什么使用LR语法分析器

- LR语法分析技术的优点
  - 对于几乎所有程序设计语言构造，只要能够写出其上下文无关文法，就能够构造出识别该构造的LR语法分析器
  - LR方法是已知的最通用的无回溯移入-归约分析技术，并且其实现可以和其他更原始的移入-归约方法一样高效
  - LR语法分析器可以在对输入进行从左到右扫描时尽可能早地检测到错误
  - 可以用LR方法进行语法分析的文法类是可以用预测方法或*LL*方法进行语法分析的文法类的真超集；即LR文法能够比*LL*文法描述更多的语言
- LR方法的缺点
  - 手工构造LR分析器的工作量非常大

LR分析器

- LR分析器的组成
  1. 分析程序
  2. 状态栈
  3. 分析表
  - 分析程序和状态栈对于所有的LR分析法都相同



例……LR分析器分析过程

(1) $E \rightarrow E + T$　(2) $E \rightarrow T$

(3) $T \rightarrow T * F$　(4) $T \rightarrow F$

(5) $F \rightarrow (E)$　(6) $F \rightarrow id$

状态	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				**acc**			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

步骤	状态	符号	输入串
1	0	$	id * id + id $
2	05	$ id	* id + id $
3	03	$ F	* id + id $
4	02	$ T	* id + id $
5	027	$ T *	id+ id$
6	0275	$ T * id	+ id $
7	02710	$ T * F	+ id $
8	02	$ T	+ id $
9	01	$ E	+ id $
10	016	$ E +	id $
11	0165	$ E + id	$
12	0163	$ E + F	$
13	0169	$ E + T	$
14	01	$ E	$

LR(0)项

- LR(0)项（*item*），简称项
  - LR语法分析器（DFA）要维护一些状态，用这些状态表明我们在语法分析过程中所处的位置，从而做出移入-归约决策
  - 每个状态是一个项的集合
  - 一个文法G的LR(0)项是G的一个产生式再加上一个位于产生式右部的某处的点
    - 产生式$A \rightarrow XYZ$有4个项

      $A \rightarrow \bullet XYZ$　$A \rightarrow X \bullet YZ$　$A \rightarrow XY \bullet Z$　$A \rightarrow XYZ \bullet$
    - 产生式$A \rightarrow \varepsilon$ 只有一个项$A \rightarrow \bullet$
  - 直观地讲，项指明了在语法分析过程中的给定点上，我们已经看到了一个产生式的哪些部分

规范LR(0)项集族

- 规范LR(0)项集族（*canonical LR(0) collectoin*）
  - *LR(0)*项集的集合，是构建一个*DFA*的基础，该*DFA*可以用于做出语法分析决策，叫做*LR(0)*自动机
- LR(0)自动机
  - 规范LR(0)项集族是LR(0)自动机的状态集，这个自动机中的每个状态是一个项集
  - LR(0)自动机是构造LR分析器的基础
  - 一个文法的LR(0)自动机也叫做识别该文法活前缀的DFA
    - 活前缀是指最右句型的一个前缀，它不含句柄右边的任何符号；在活前缀右部添加一些终结符号就可能形成一个最右句型

构造规范LR(0)项集族

- 增广文法（*augmented grammar*）
  - 也译作拓广文法
  - 如果G是一个以S为开始符号的文法，那么G的**增广文法**G'就是在G中加上新开始符号S'和产生式$S' \rightarrow S$而得到的文法
    - 引入新开始产生式的目的是告诉语法分析器何时应该停止语法分析并宣告接受输入符号串
    - 当且仅当语法分析器要使用规则$S' \rightarrow S$进行归约时，输入符号串被接受
- *CLOSURE*函数：项集的闭包
- *GOTO*函数：LR(0)自动机的状态转换函数

构造规范LR(0)项集族（NFA→DFA子集法）

S′→E

E→aA|bB

A→cA|d

B→cB|d

这个文法的项目有：

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·

构造规范LR(0)项集族（NFA→DFA子集法）

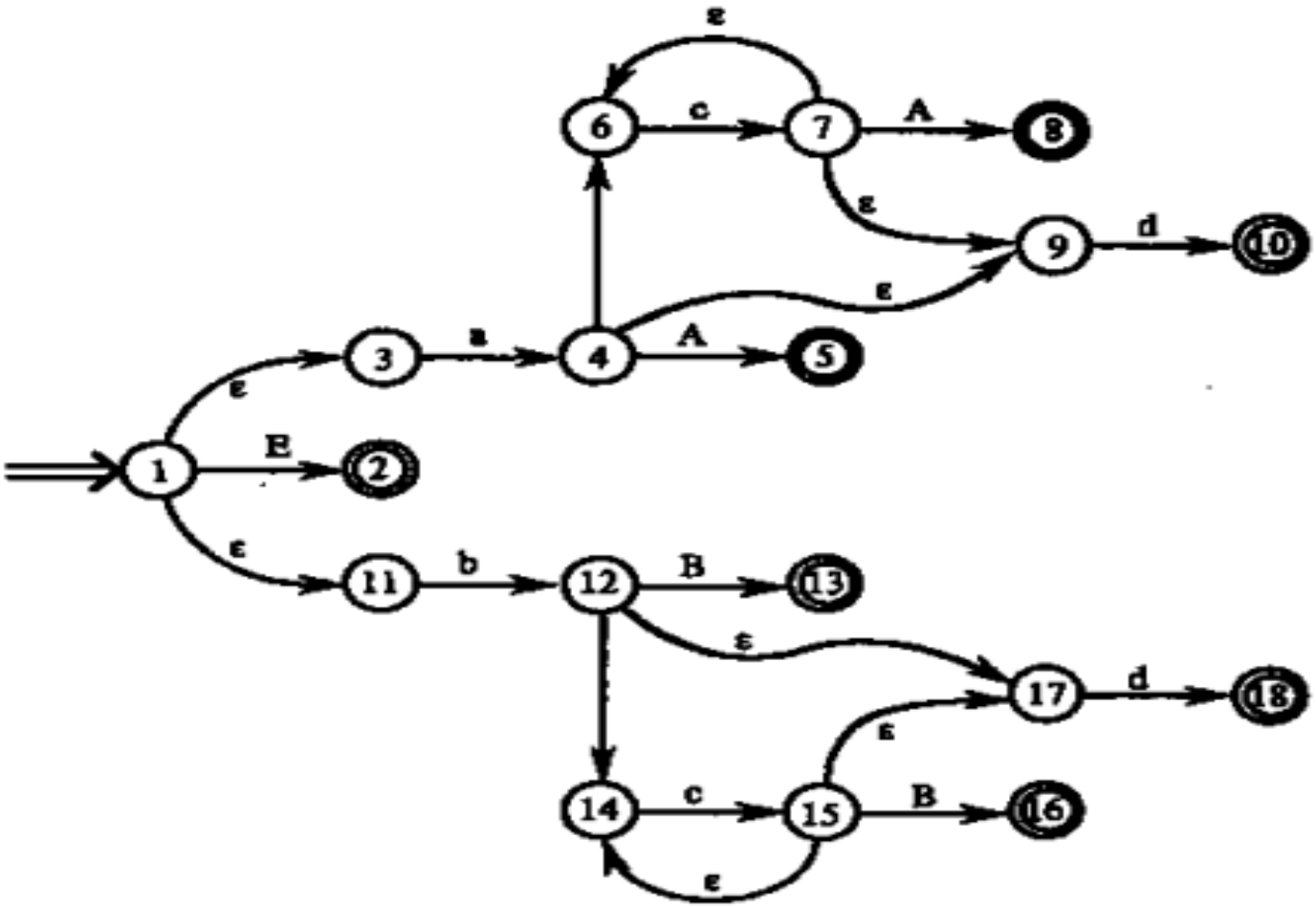


图 5.6　识别活前缀的 NFA

构造规范LR(0)项集族（NFA→DFA子集法）

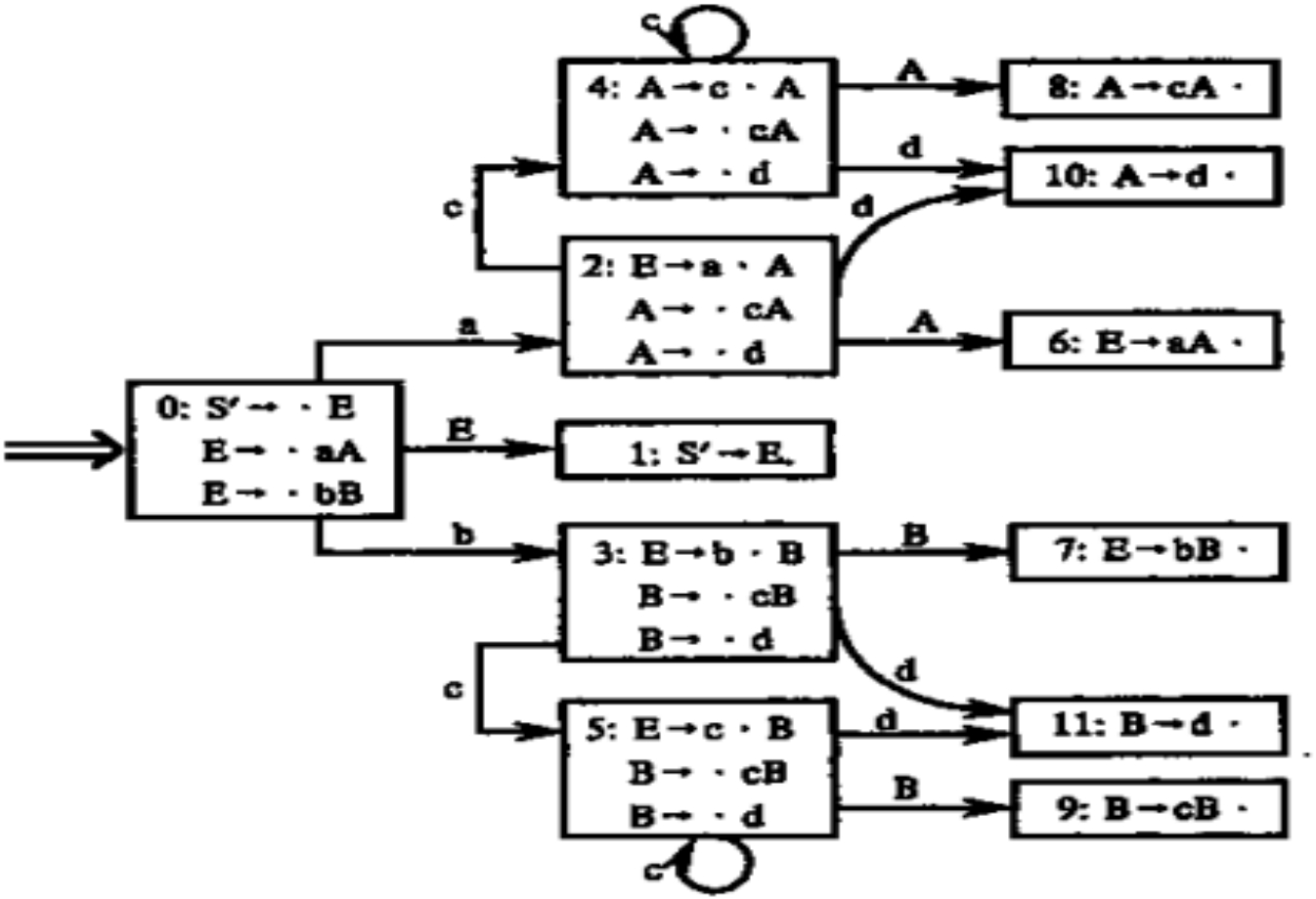


图 5.7　识别前缀的 DFA

构造规范LR(0)项集族（NFA→DFA子集法）

表 5.4　LR(0)分析表

状　态	ACTION					GOTO		
	a	b	c	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	r4	r4	r4	r4	r4			
10	r5	r5	r5	r5	r5			
11	r6	r6	r6	r6	r6			

CLOSURE函数

- 项集的闭包
  - 如果I是文法G的一个项集，那么$CLOSURE(I)$就是根据下面的规则从I构造得到的项集：
    1. 一开始，将I中的各个项加入到$CLOSURE(I)$中
    2. 如果$A\rightarrow\alpha\bullet B\beta$在$CLOSURE(I)$中，$B\rightarrow\gamma$是一个产生式，并且项$B\rightarrow\bullet\gamma$不在$CLOSURE(I)$中，就将这个项加入其中
    3. 不断应用规则②，直到没有新项可以加入到$CLOSURE(I)$中为止。
  - 直观地讲，$CLOSURE(I)$中的项$A\rightarrow\alpha\bullet B\beta$表明在语法分析过程的某点上，我们认为接下来可能会在输入中看到一个能够从$B\beta$推导得到的子串；这个子串的某个前缀可以从B推导得到，而推导时必然要应用到某个B产生式，因此我们加入了各个B产生式对应的项$B\rightarrow\bullet\gamma$

CLOSURE函数

- 例：

  如果$I = \{ [E' \rightarrow \bullet E] \}$

  $CLOSURE(I) = \{$

  $E' \rightarrow \bullet E$

  $E \rightarrow \bullet E+T$

  $E \rightarrow \bullet T$

  $T \rightarrow \bullet T*F$

  $T \rightarrow \bullet F$

  $F \rightarrow \bullet (E)$

  $F \rightarrow \bullet id$

  $\}$

$E' \rightarrow E$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid \mathbf{id}$

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

GOTO函数

- *GOTO(I , X)*
  - *I*是一个项集，*X*是一个文法符号
  - *GOTO(I , X)*被定义为I中所有形如[$A \rightarrow \alpha \bullet X\beta$]的项所对应的项[$A \rightarrow \alpha X \bullet \beta$]的集合的闭包
  - 直观地讲，*GOTO*函数用于定义一个文法的LR(0)自动机中的转换；这个自动机的状态对应于项集，而*GOTO(I , X)*描述了当输入为*X*时离开状态*I*的转换
  - 例：如果$I = \{ [E' \rightarrow E \bullet] , [E \rightarrow E \bullet + T]\}$，那么*GOTO(I , +)* 包含

$$
\begin{aligned}
E &\rightarrow E + \cdot T \\
T &\rightarrow \cdot T * F \\
T &\rightarrow \cdot F \\
F &\rightarrow \cdot (E) \\
F &\rightarrow \cdot \mathbf{id}
\end{aligned}
$$

构造规范LR(0)项集族的算法

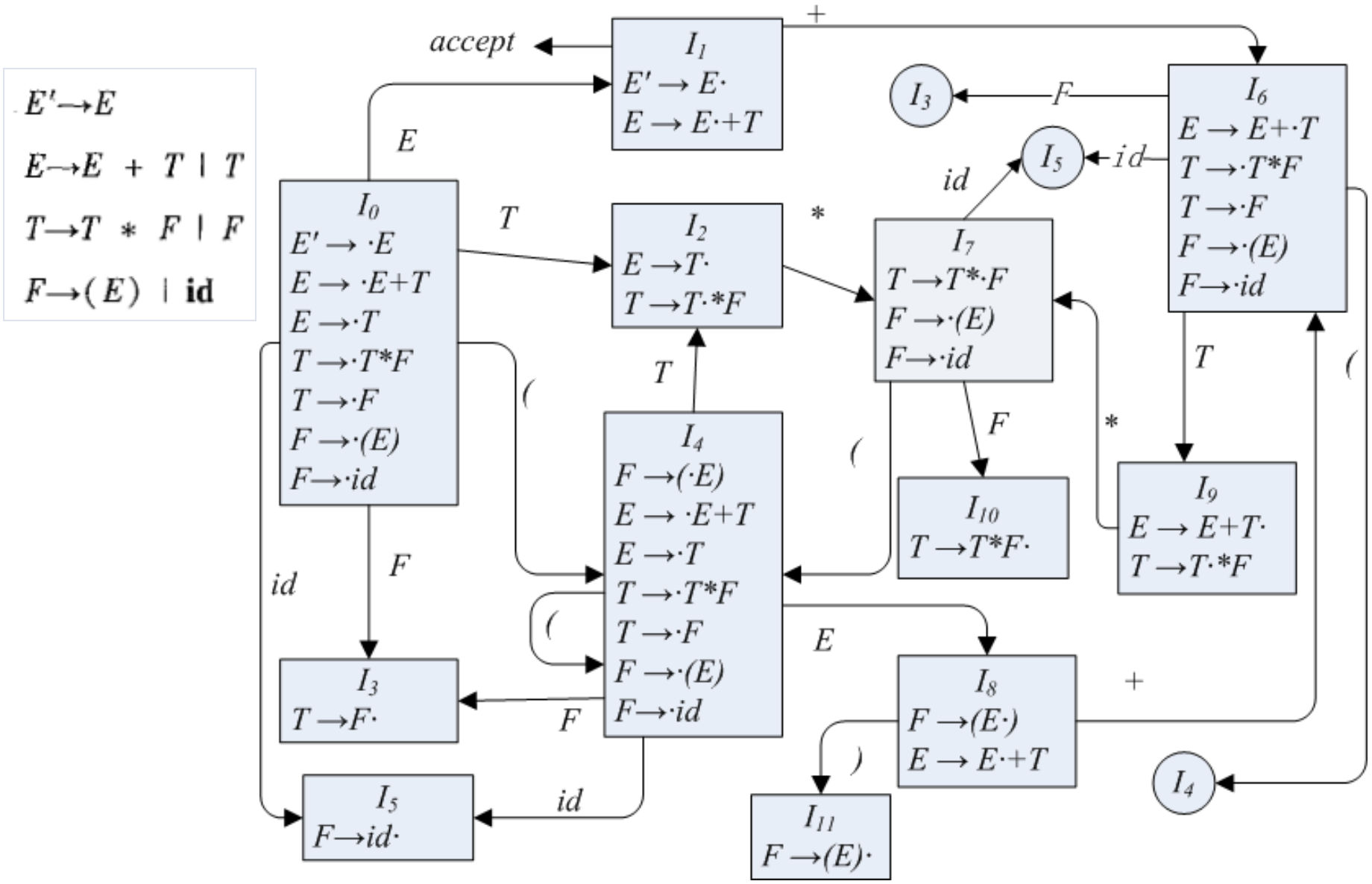
- 构造一个增广文法G'的规范$LR(0)$项集族C的算法

```
void items(G') {
        C = CLOSURE({[S' → ·S]});
        repeat
                for ( each set of items I in C )
                        for ( each grammar symbol X )
                                if ( GOTO(I, X) is not empty and not in C )
                                        add GOTO(I, X) to C;
        until no new sets of items are added to C on a round;
}
```

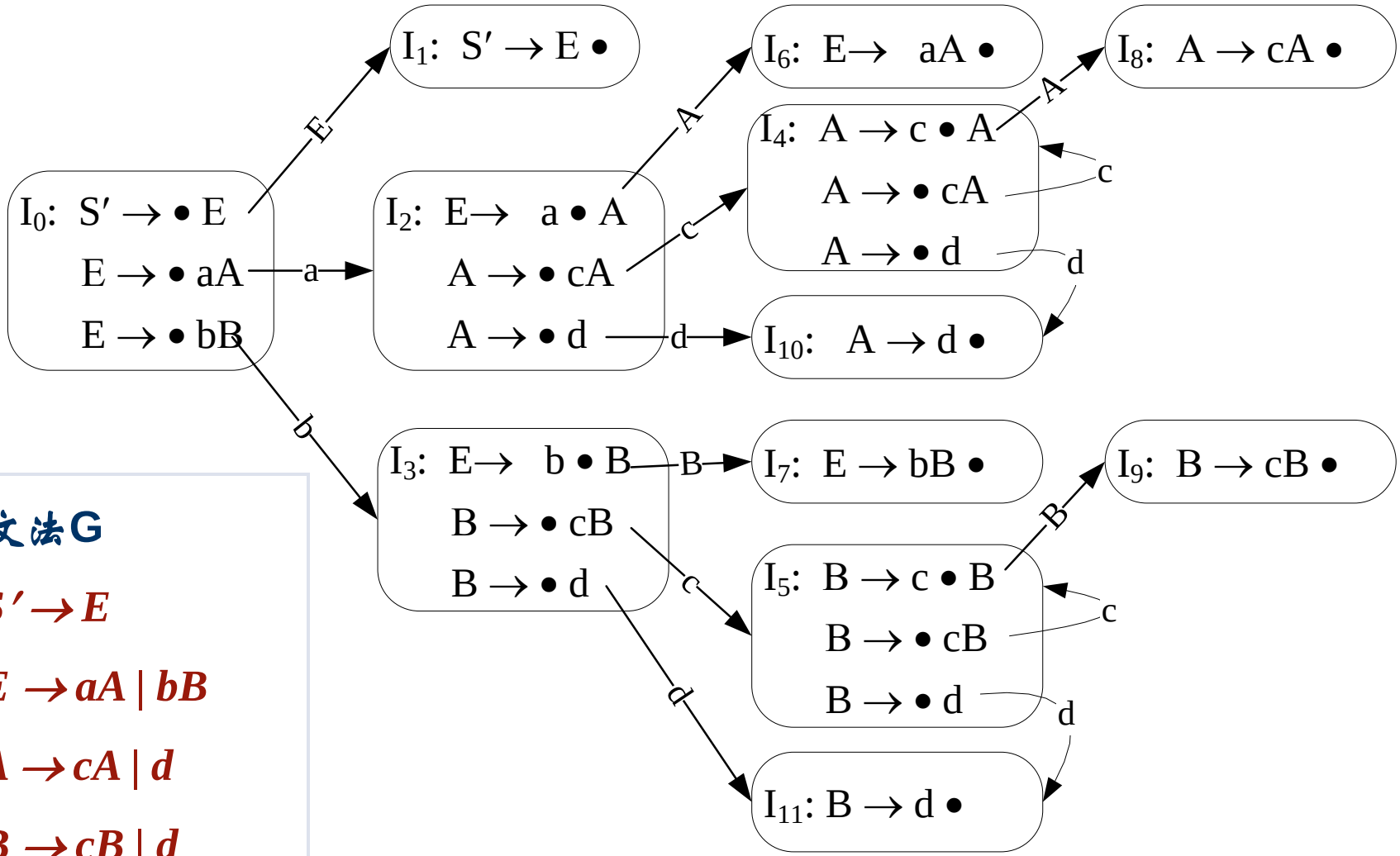
LR(0)项的分类

- 按照对LR(0)自动机动作的影响分类
  - 项 $A \to \alpha \bullet$ 称为归约项
  - 项 $S' \to S \bullet$ 称为接受项 (特殊的归约项)
  - 项 $A \to \alpha \bullet a\beta$ 称为移入项，a 为终结符号
  - 项 $A \to \alpha \bullet B\beta$ 称为待约项

例……构造规范LR(0)项集族



例……构造规范LR(0)项集族



文法G

$S' \rightarrow E$

$E \rightarrow aA \mid bB$

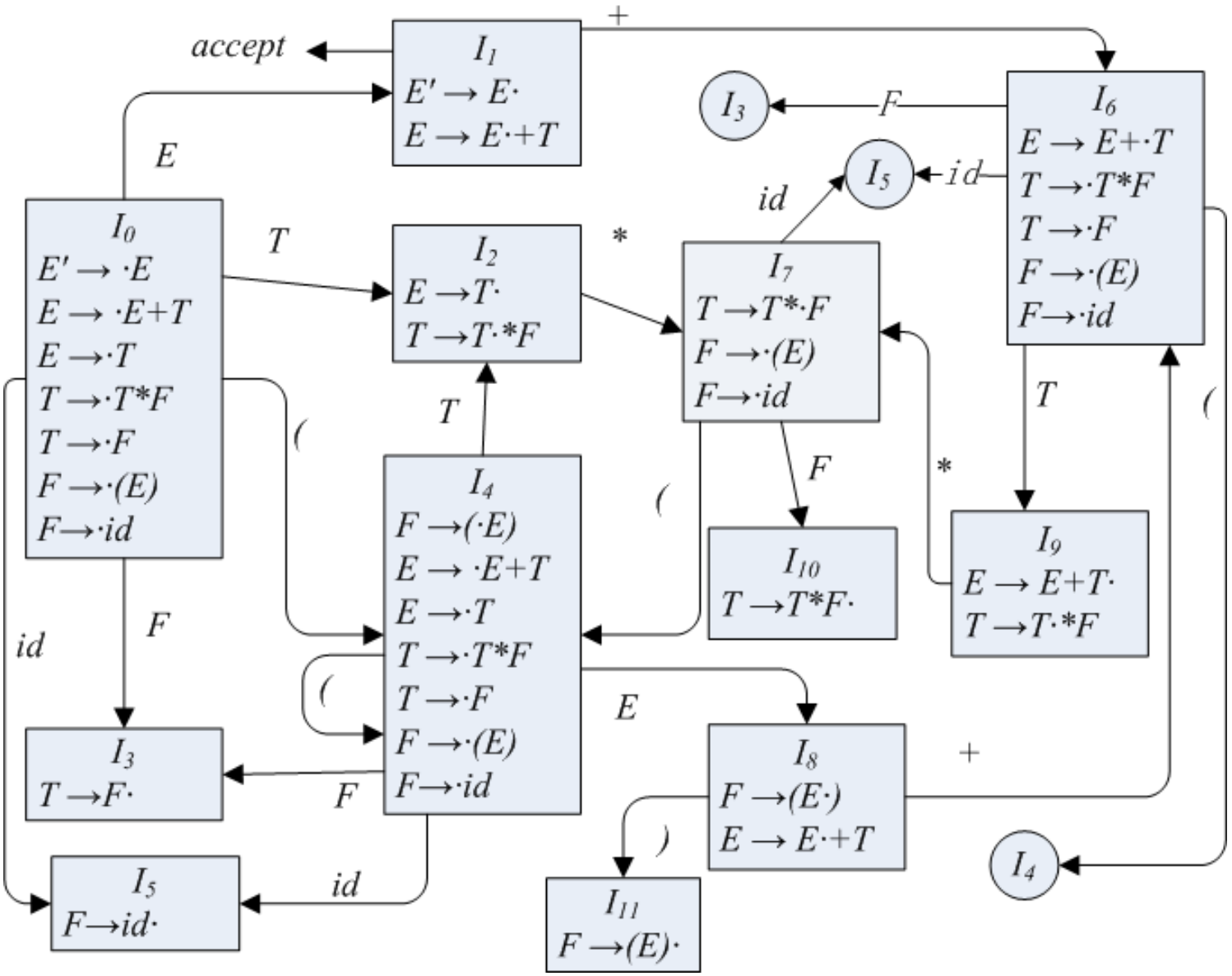
$A \rightarrow cA \mid d$

$B \rightarrow cB \mid d$

LR(0)自动机的用法

- LR(0)语法分析技术的中心思想是根据文法构造出LR(0)自动机，这个自动机的
  - 状态是规范LR(0)项集族中的元素，将项集I_k称为状态k
  - 转换由$GOTO$函数定义
  - 开始状态是$CLOSURE(\{[S' \rightarrow \bullet S]\})$
- LR(0)自动机对移入-归约决策的作用
  - 假设文法符号串γ使LR(0)自动机从开始状态0运行到某个状态k；如果下一个输入符号为a且状态k有一个在a上的转换，就移入a；否则选择归约动作，状态k的项将告诉我们用哪个产生式进行归约。

例……LR(0)自动机的用法


accept
I1
E′ → E·
E → E·+T
+
I6
E → E+·T
T →·T*F
T →·F
F →·(E)
F→·id
I3
F
I5
id
id
E
I0
E′ → ·E
E → ·E+T
E →·T
T →·T*F
T →·F
F →·(E)
F→·id
T
I2
E →T·
T →T·*F
*
I7
T →T*·F
F →·(E)
F→·id
T
(
T
I4
F →(·E)
E → ·E+T
E →·T
T →·T*F
T →·F
F →·(E)
F→·id
F
(
*
I10
T →T*F·
I9
E → E+T·
T →T·*F
id
F
I3
T →F·
(
F
E
I8
F →(E·)
E → E·+T
+
I4
)
I5
F→id·
id
I11
F →(E)·

例……LR(0)自动机的用法

LINE	STACK	SYMBOLS	INPUT	ACTION
(1)	0	\$	**id** $*$ **id** \$	shift to 5
(2)	0 5	\$ **id**	$*$ **id** \$	reduce by $F \to \mathbf{id}$
(3)	0 3	\$ F	$*$ **id** \$	reduce by $T \to F$
(4)	0 2	\$ T	$*$ **id** \$	shift to 7
(5)	0 2 7	\$ T $*$	**id** \$	shift to 5
(6)	0 2 7 5	\$ T $*$ **id**	\$	reduce by $F \to \mathbf{id}$
(7)	0 2 7 10	\$ T $*$ F	\$	reduce by $T \to T * F$
(8)	0 2	\$ T	\$	reduce by $E \to T$
(9)	0 1	\$ E	\$	accept

LR语法分析表的结构

- 语法分析动作函数$ACTION[i, a]$可以有4种形式的取值
  1. 移入j，其中j是一个状态。把输入符号a移入栈中，用状态j代表a
  2. 归约$A \rightarrow \beta$。把栈顶的β归约为产生式左部A
  3. 接受。语法分析器接受输入并完成语法分析过程
  4. 报错。发现输入中有错误并执行纠正动作
- 转换函数$GOTO$
  - 把定义在项集上的$GOTO$函数扩展为定义在状态集上的函数：如果$GOTO[I_i, A]=I_j$，那么$GOTO$也把状态i和一个非终结符号A映射到状态j。

例……表达式文法的LR语法分析表

各种动作在此图中的编码方法如下：

1）si 表示移入并将状态 i 压栈。

2）rj 表示按照编号为 j 的产生式进行归约。

3）acc 表示接受。

4）空白表示报错。

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

例……LR分析器分析过程

(1) $E \rightarrow E + T$　(2) $E \rightarrow T$

(3) $T \rightarrow T * F$　(4) $T \rightarrow F$

(5) $F \rightarrow (E)$　(6) $F \rightarrow id$

状态	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	s5			s4			1	2	3
1		s6				**acc**			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

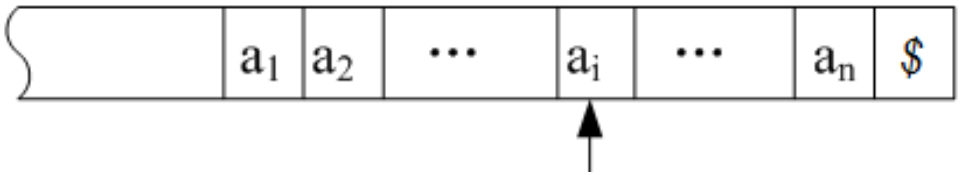
步骤	状态	符号	输入串
1	0	$	id * id + id $
2	05	$ id	* id + id $
3	03	$ F	* id + id $
4	02	$ T	* id + id $
5	027	$ T *	id+ id$
6	0275	$ T * id	+ id $
7	02710	$ T * F	+ id $
8	02	$ T	+ id $
9	01	$ E	+ id $
10	016	$ E +	id $
11	0165	$ E + id	$
12	0163	$ E + F	$
13	0169	$ E + T	$
14	01	$ E	$

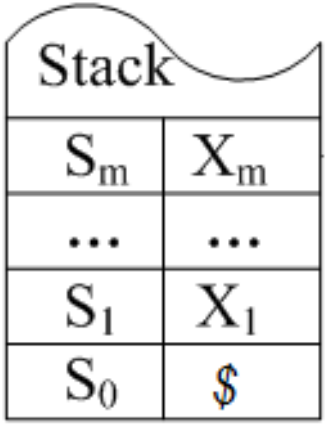
LR语法分析器的格局

- LR语法分析器的格局（*configuration*）是一个二元组

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

  - 用来表示LR语法分析器的完整状态，即栈和剩余输入
    - 第一个分量是栈中的内容，右侧是栈顶
    - 第二个分量是余下的输入，$表示输入串结束标志
  - 分析器的格局表示了一个最右句型：$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$
    - 栈中保存的状态代表文法符号
    - 状态s_0不对应文法符号，表示栈底





LR语法分析器的格局

- 语法分析器根据格局和分析表决定下一个动作
  - 当前格局为$(s_0s_1...s_m, a_ia_{i+1}...a_n\$)$，分析器读入当前输入符号$a_i$和栈顶状态$s_m$；在分析动作表中查询条目$ACTION[s_m, a_i]$，如果$ACTION[s_m, a_i]$
    1. = 移入s，执行一次移入动作，状态s进栈，进入格局$(s_0s_1...s_ms, a_{i+1}...a_n\$)$；当前输入符号变成$a_{i+1}$
    2. = 归约$A\rightarrow\beta$，执行一次归约动作，进入格局$(s_0s_1...s_{m-r}s, a_ia_{i+1}...a_n\$)$；其中$r=|\beta|$，$s=GOTO[s_{m-r}, A]$。语法分析器将$r$个状态出栈，然后将$s$进栈，当前输入符号不变
    3. = 接受，语法分析过程完成
    4. = 报错，出现语法错误，调用错误恢复例程

LR语法分析算法

输入：一个输入串 w 和一个 LR 语法分析表，这个表描述了文法 G 的 ACTION 函数和 GOTO 函数。

输出：如果 w 在 $L(G)$ 中，则输出 w 的自底向上语法分析过程中的归约步骤；否则给出一个错误指示。

方法：最初，语法分析器栈中的内容为初始状态 s_0，输入缓冲区中的内容为 $w\$$。然后，语法分析器执行图 4-36 中的程序。 □

```
let a be the first symbol of w$;
while(1) { /* repeat forever */
        let s be the state on top of the stack;
        if ( ACTION[s, a] = shift t ) {
                push t onto the stack;
                let a be the next input symbol;
        } else if ( ACTION[s, a] = reduce A → β ) {
                pop |β| symbols off the stack;
                let state t now be on top of the stack;
                push GOTO[t, A] onto the stack;
                output the production A → β;
        } else if ( ACTION[s, a] = accept ) break; /* parsing is done */
        else call error-recovery routine;
}
```

构造SLR语法分析表

输入：一个增广文法 G'。

输出：G'的 SLR 语法分析表函数 ACTION 和 GOTO。

方法：

1）构造 G'的规范 LR(0)项集族 $C=\{I_0, I_1, \cdots, I_n\}$。

2）根据 I_i 构造得到状态 i。状态 i 的语法分析动作按照下面的方法决定：

① 如果$[A\rightarrow\alpha\cdot a\beta]$在 I_i 中并且 $\text{GOTO}(I_i, a)=I_j$，那么将 ACTION$[i, a]$设置为“移入 j”。这里 a 必须是一个终结符号。

② 如果$[A\rightarrow\alpha\cdot]$在 I_i 中，那么对于 FOLLOW(A)中的所有 a，将 ACTION$[i, a]$设置为“归约 $A\rightarrow\alpha$”。这里 A 不等于 S'。

③ 如果$[S'\rightarrow S\cdot]$在 I_i 中，那么将 ACTION$[i, \$]$设置为“接受”。

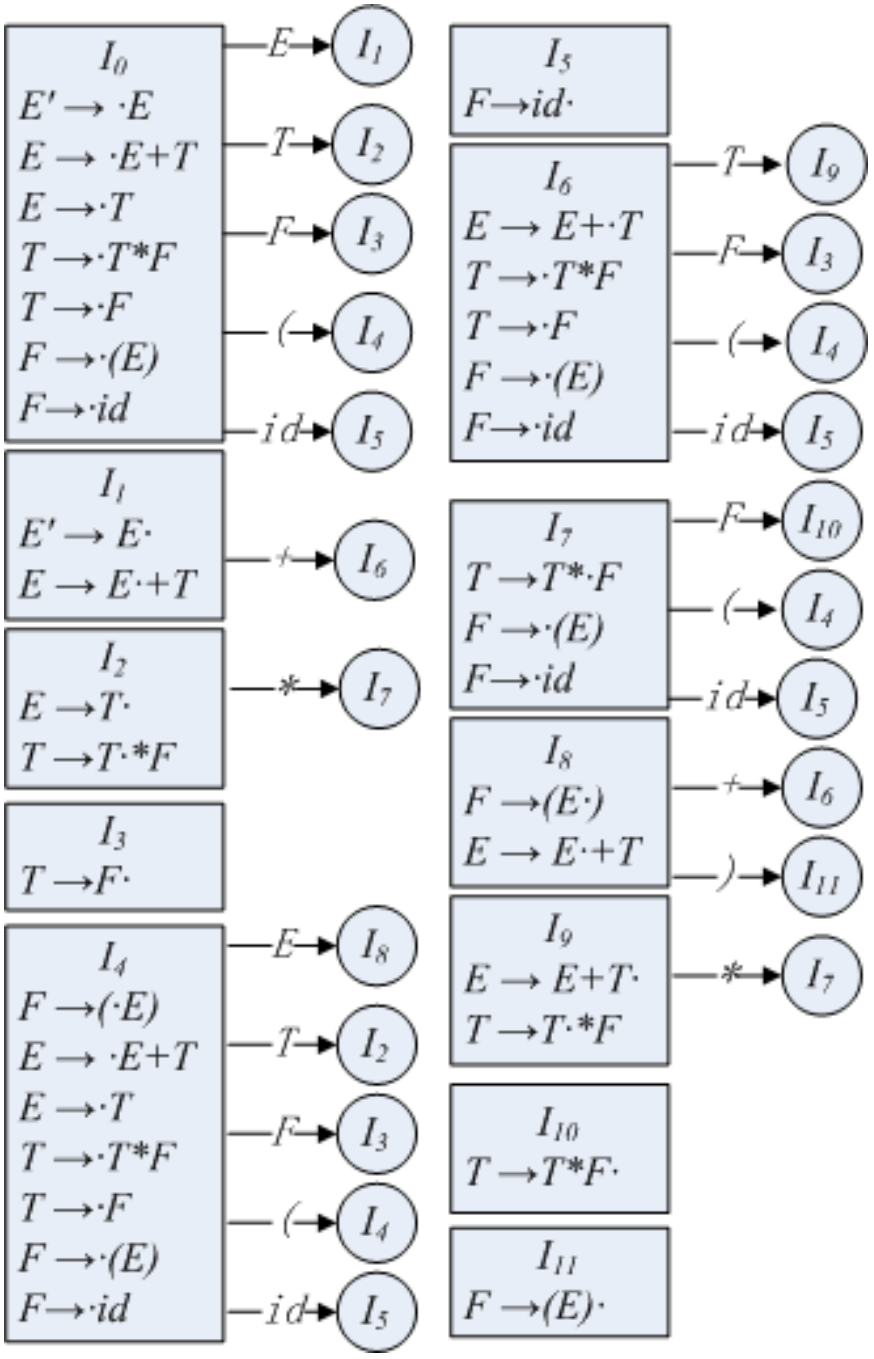
如果根据上面的规则生成了任何冲突动作，我们就说这个文法不是 SLR(1)的。在这种情况下，这个算法无法生成一个语法分析器。

3）状态 i 对于各个非终结符号 A 的 GOTO 转换使用下面的规则构造得到：如果 $\text{GOTO}(I_i, A)=I_j$，那么 GOTO$[i, A]=j$。

4）规则(2)和(3) 没有定义的所有条目都设置为“报错”。

5）语法分析器的初始状态就是根据$[S'\rightarrow\cdot S]$所在项集构造得到的状态。 □

例……构造SLR分析表



(1) $E \to E+T$
(2) $E \to T$
(3) $T \to T*F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

状态	ACTION						GOTO		
	id	+	*	(	)	$	**E**	**T**	**F**
0	s5			s4			1	2	3
1		s6				**acc**			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

构造SLR分析表……例题（课堂练习）

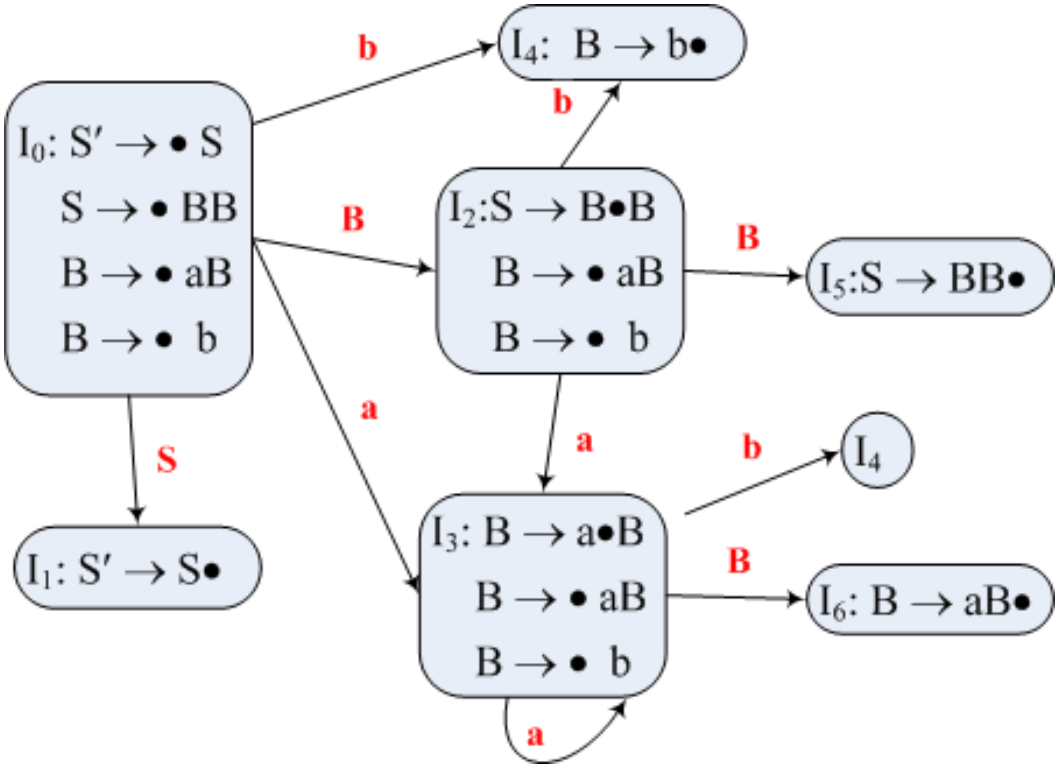
- 对文法 $S \rightarrow BB \quad B \rightarrow aB|b$
  - ① 构造此文法的规范LR(0)项集族
  - ② 构造SLR语法分析表
  - ③ 写出对输入串aabab的分析过程
- 解答
  - ① 增广文法
  - ② 构造LR(0)自动机
  - ③ 构造SLR语法分析表
  - ④ 按照分析表对*aabab*进行LR分析

例题……解答

解：增广文法，得G′如下

(0)S′→S (1)S→BB (2)B→aB (3) B→b

对G′构造LR(0) 自动机如下



则规范LR(0)项集族C={$I_0, I_1, \cdots I_6$}

构造SLR分析表

FOLLOW(S)={$}

FOLLOW(B)={$,a,b}

状态	*ACTION*			*GOTO*	
	a	b	$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5			r1		
6	r2	r2	r2		

输入串aabab的分析过程略

SLR(1)分析表的构造……例

- 文法G(S):

  $S \rightarrow bTc \mid a$

  $T \rightarrow R$

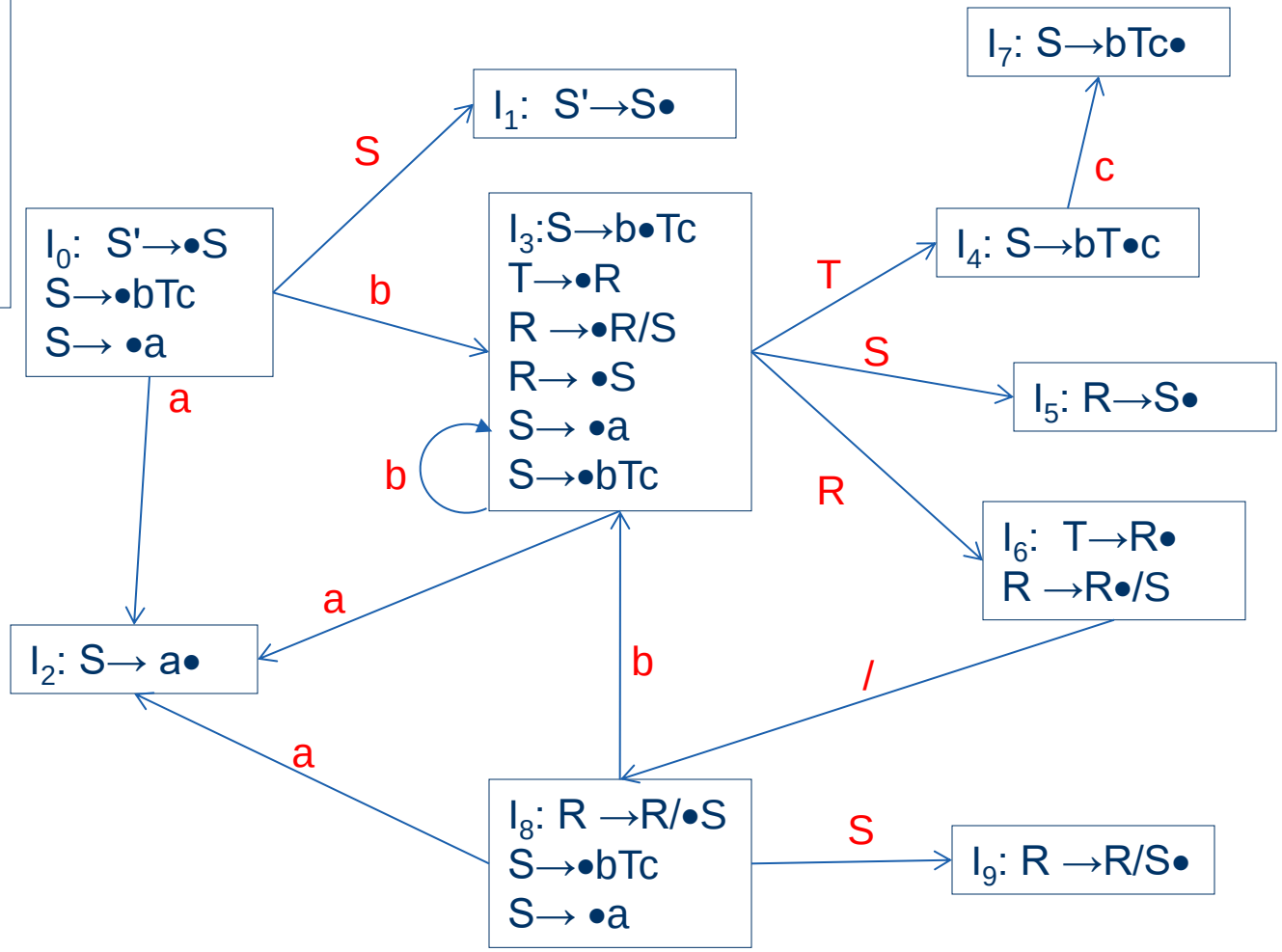
  $R \rightarrow R/S \mid S$

  该文法是否为SLR文法？请予以证实。

- 解答
  - 增广文法
  - 构造规范LR(0)项集族
  - 构造SLR分析表
  - 检查分析表中是否存在冲突如果不存在冲突是SLR文法，否则不是

SLR(1)分析表的构造……解答

0. S'→S
1. S→bTc
2. S→ a
3. T→R
4. R →R/S
5. R→ S



非SLR文法

- 二义性文法不是SLR文法，有些无二义文法也不是SLR

$$
\begin{array}{lcl}
S & \rightarrow & L = R \mid R \\
L & \rightarrow & *R \mid \mathbf{id} \\
R & \rightarrow & L
\end{array}
$$

FOLLOW(S) = {$}

FOLLOW(L) = { =, $}

FOLLOW(R) = { =, $}

$$
\begin{array}{lll}
I_0: & S' \rightarrow \cdot S & \\
 & S \rightarrow \cdot L = R & \\
 & S \rightarrow \cdot R & \\
 & L \rightarrow \cdot * R & \\
 & L \rightarrow \cdot \mathbf{id} & \\
 & R \rightarrow \cdot L & \\
I_1: & S' \rightarrow S \cdot & \\
I_2: & S \rightarrow L \cdot = R & \\
 & R \rightarrow L \cdot & \\
I_3: & S \rightarrow R \cdot & \\
I_4: & L \rightarrow * \cdot R & \\
 & R \rightarrow \cdot L & \\
 & L \rightarrow \cdot * R & \\
 & L \rightarrow \cdot \mathbf{id} & \\
I_5: & L \rightarrow \mathbf{id} \cdot & \\
I_6: & S \rightarrow L = \cdot R & \\
 & R \rightarrow \cdot L & \\
 & L \rightarrow \cdot * R & \\
 & L \rightarrow \cdot \mathbf{id} & \\
I_7: & L \rightarrow * R \cdot & \\
I_8: & R \rightarrow L \cdot & \\
I_9: & S \rightarrow L = R \cdot &
\end{array}
$$

- 项集I_2中存在移入-归约冲突
- 产生冲突的原因是构造SLR分析器的方法功能不够强大，不能记住足够多的上下文信息
- 规范LR方法和LALR方法可以成功处理更多的文法类型

活前缀

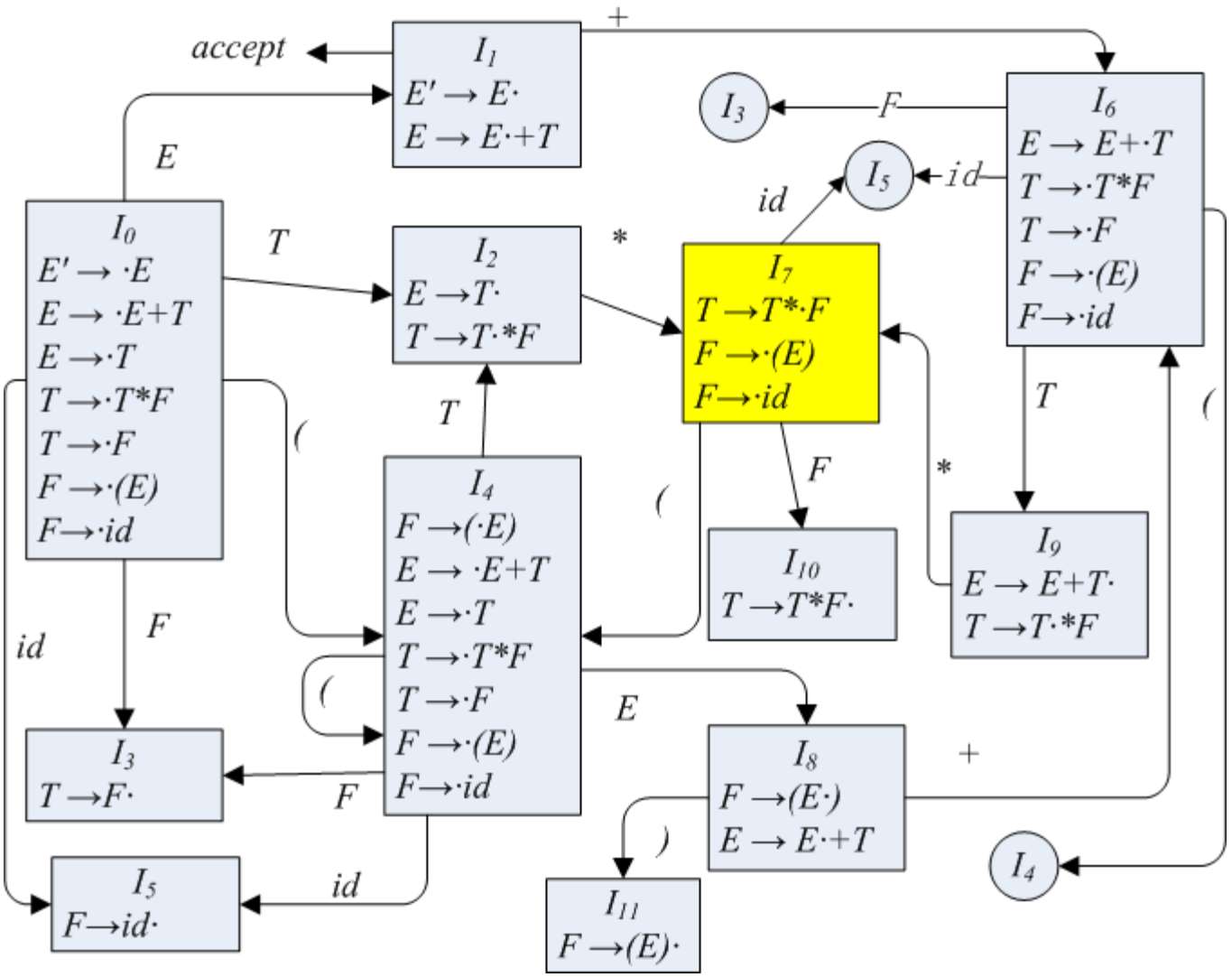
- 为什么可以用LR(0)自动机做出移入-归约决策？
  - 文法的LR(0)自动机可以识别可能出现在分析器栈中的文法符号串
  - 栈中的文法符号串一定是某个最右句型的前缀
    - 如果栈中内容是α，余下的输入是x，那么存在一个将αx归约为开始符号的归约序列，推导形式为：$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha x$
- 活前缀（可行前缀）
  - 是一个最右句型的前缀，并且它没有越过该最右句型的最右句柄的右端
    - 我们总是可以在一个可行前缀之后增加一些终结符号来得到一个最右句型

可行前缀

- LR(0)自动机能够识别可行前缀
  - 这是SLR分析技术的基础
- 有效项
  - 如果存在一个推导过程 $S' \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$（ω为终结符号串，可以为空串）

  我们就说项$A \rightarrow \beta_1 \bullet \beta_2$对可行前缀$\alpha\beta_1$有效
  - 一个项可以对多个可行前缀有效
  - 对一个可行前缀有效的项可以为多个
- LR语法分析理论的核心定理
  - 如果我们在某个文法的LR(0)自动机中从初始状态开始沿着标号为某个可行前缀γ的路径到达一个状态，那么该状态对应的项集就是γ的有效项集

可行前缀



$$E' \underset{rm}{\Rightarrow} E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F \underset{rm}{\Rightarrow}$$

$$E' \underset{rm}{\Rightarrow} E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F \underset{rm}{\Rightarrow} E+T*(E) \underset{rm}{\Rightarrow}$$

$$E' \underset{rm}{\Rightarrow} E \underset{rm}{\Rightarrow} E+T \underset{rm}{\Rightarrow} E+T*F \underset{rm}{\Rightarrow} E+T*\mathbf{id} \underset{rm}{\Rightarrow}$$

练习4.6（作业）

练习4.6.1：描述下列文法的所有可行前缀：

1）练习4.2.2(1)的文法 $S \rightarrow 0\ S\ 1 \mid 0\ 1$。

！2）练习4.2.1的文法 $S \rightarrow S\ S\ + \mid S\ S\ * \mid a$。

！3）练习4.2.2(3)的文法 $S \rightarrow S\ (\ S\ ) \mid \epsilon$。

练习4.6.2：为练习4.2.1中的(增广)文法构造SLR项集。计算这些项集的GOTO函数。给出这个文法的语法分析表。这个文法是SLR文法吗？

练习4.6.3：利用练习4.6.2得到的语法分析表，给出处理输入 $aa*a+$ 时的各个动作。

练习4.6.4：对于练习4.2.2(1)～(7)中的各个(增广)文法：

1）构造SLR项集和它们的GOTO函数。

2）指出你的项集中的所有动作冲突。

3）如果存在SLR语法分析表，构造出这个语法分析表。

练习4.6.5：说明下面的文法是LL(1)的，但不是SLR(1)的。

$$S \rightarrow A\ a\ A\ b \mid B\ b\ B\ a \qquad A \rightarrow \epsilon \qquad B \rightarrow \epsilon$$

练习4.6.6：说明下面的文法是SLR(1)的，但不是LL(1)的。

$$S \rightarrow S\ A \mid A \qquad A \rightarrow a$$

更强大的LR语法分析器

规范LR方法和LALR方法

更强大的LR语法分析技术

- 规范LR（*canonical-LR*）方法
  - 也称为LR方法或LR(1)方法
  - 利用向前看符号，使用LR(1)项集
- LALR（*lookahead-LR*）方法
  - 向前看LR
  - 基于LR(0)项集族，向LR(0)项中引入向前看符号
  - 和基于LR(1)项的典型语法分析器相比，状态数要少
  - 分析能力比SLR强，构造的分析表不比SLR分析表大

规范LR(1)项

- SLR方法的归约动作特点
  - 如果项集I_i中包含项$[A \rightarrow \alpha \bullet]$，当前输入符号$a \in FOLLOW(A)$，那么状态$i$就要按照$A \rightarrow \alpha$进行归约
  - 虽然SLR方法强于LR（0）的，但是还是有很多文法不是SLR文法，不能用SLR分析器进行分析。
  - 这是因为在LR(0)项中没有包含足够的信息，如果让状态包含更多的信息，就可能排除掉一些不正确的归约
- 解决方法
  - 分裂LR(0)自动机的某些状态，设法让LR语法分析器的每个状态更精确地指明哪些输入符号可以跟在句柄α的后面，从而使α可以被归约为A

LR(1)项

- LR(1)项
  - 形如$[A\rightarrow\alpha\bullet\beta, a]$，其中$A\rightarrow\alpha\beta$是一个产生式，$a$是一个终结符号或\$。
    - 第二个分量叫做向前看符号串，LR(1)的1是指串的长度为1
  - 在$[A\rightarrow\alpha\bullet\beta, a]$且$\beta\neq\varepsilon$时，向前看符号$a$没有任何作用；$a$只对形如$[A\rightarrow\alpha\bullet, a]$的归约项有效，意思是只有在下一个输入符号等于$a$时才按照$A\rightarrow\alpha$进行归约
  - 因此，只有栈顶状态中包含形如$[A\rightarrow\alpha\bullet, a]$项，且输入为$a$时才用$A\rightarrow\alpha$进行归约
  - 向前看符号a的集合总是FOLLOW(A)的子集，而且很可能是真子集

有效LR(1)项

- LR(1)项$[A\to\alpha\bullet\beta, a]$对可行前缀$\gamma$的有效性

正式地讲，我们说LR(1)项$[A\to\alpha\cdot\beta, a]$对于一个可行前缀$\gamma$有效的条件是存在一个推导 $S \overset{*}{\underset{rm}{\Rightarrow}} \delta Aw \underset{rm}{\Rightarrow} \delta\alpha\beta w$，其中

1）$\gamma = \delta\alpha$，且

2）要么a是w的第一个符号，要么w为ϵ且a等于\$。

- 例

考虑文法 $S\to B\ B$ $B\to a\ B \mid b$

该文法有一个最右推导 $S \overset{*}{\underset{rm}{\Rightarrow}} aaBab \underset{rm}{\Rightarrow} aaaBab$。

在上面的定义中，令$\delta = aa$，$A = B$，$w = ab$，$\alpha = a$且$\beta = B$，

可知项$[B\to\alpha\cdot B, a]$对于可行前缀$\gamma = aaa$是有效的。

另外还有一个最右推导 $S \overset{*}{\underset{rm}{\Rightarrow}} BaB \underset{rm}{\Rightarrow} BaaB$。

根据这个推导，我们知道项$[B\to\alpha\cdot B, \$]$是可行前缀$Baa$的有效项。

构造LR(1)项集

- 修改的*CLOSURE*函数和*GOTO*函数

SetOfItems CLOSURE(I) {
    repeat
        for (each item $[A \to \alpha \cdot B\beta, a]$ in I)
            for (each production $B \to \gamma$ in G')
                for (each terminal b in FIRST(βa))
                    add $[B \to \cdot\gamma, b]$ to set I;
    until no more items are added to I;
    return I;
}

SetOfItems GOTO(I, X) {
    initialize J to be the empty set;
    for (each item $[A \to \alpha \cdot X\beta, a]$ in I)
        add item $[A \to \alpha X \cdot \beta, a]$ to set J;
    return CLOSURE(J);
}

构造LR(1)项集

为了理解 CLOSURE 操作的新定义，特别是理解为什么 b 必须在 FIRST(βa) 中，我们考虑对某些可行前缀 γ 有效的项集合中的一个形如 $[A\rightarrow\alpha\cdot B\beta, a]$ 的项，那么必然存在一个最右推导 $S \overset{*}{\underset{rm}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta\alpha B\beta ax$，其中 $\gamma=\delta\alpha$。

假设 βax 推导出终结符号串 by，那么对于某个形如 $B\rightarrow\eta$ 的产生式，我们有推导 $S \overset{*}{\underset{rm}{\Rightarrow}} \gamma Bby \underset{rm}{\Rightarrow} \gamma\eta by$。 因此，$[B\rightarrow\cdot\eta, b]$ 是 γ 的有效项。

请注意，b 可能是从 β 推导得到的第一个终结符号，也可能在 $\beta ax \overset{*}{\underset{rm}{\Rightarrow}} by$ 的推导过程中 β 推导出了 ϵ，因此 b 也可能是 a。

总结这两种情况，我们说 b 可以是 FIRST(βax) 中的任意终结符号。

请注意，x 不可能包含 by 的第一个终结符号，因此 FIRST(βax) = FIRST(βa)。

LR(1)项集族的构造算法

输入：一个增广文法 G'。

输出：LR(1)项集族，其中的每个项集对文法 G' 的一个或多个可行前缀有效。

方法：过程 CLOSURE 和 GOTO，以及用于构造项集的主例程 items。

```
void items(G') {
    initialize C to CLOSURE({[S' → ·S, $]});
    repeat
        for ( each set of items I in C )
            for ( each grammar symbol X )
                if ( GOTO(I, X) is not empty and not in C )
                    add GOTO(I, X) to C;
    until no new sets of items are added to C;
}
```

例……LR(1)项集族的构造

$$
\begin{array}{lcl}
S' & \rightarrow & S \\
S & \rightarrow & C\ C \\
C & \rightarrow & c\ C \mid d
\end{array}
$$

I_0 :
$S \rightarrow \cdot S, \$$
$S \rightarrow \cdot CC, \ \$$
$C \rightarrow \cdot cC, \ c/d$
$C \rightarrow \cdot d, \ c/d$

I_1 : $S' \rightarrow S\cdot, \ \$$

I_2 :
$S \rightarrow C\cdot C, \ \$$
$C \rightarrow \cdot cC, \ \$$
$C \rightarrow \cdot d, \ \$$

I_3 :
$C \rightarrow c\cdot C, \ c/d$
$C \rightarrow \cdot cC, \ c/d$
$C \rightarrow \cdot d, \ c/d$

I_4 : $C \rightarrow d\cdot, \ c/d$

I_5 : $S \rightarrow CC\cdot, \$$

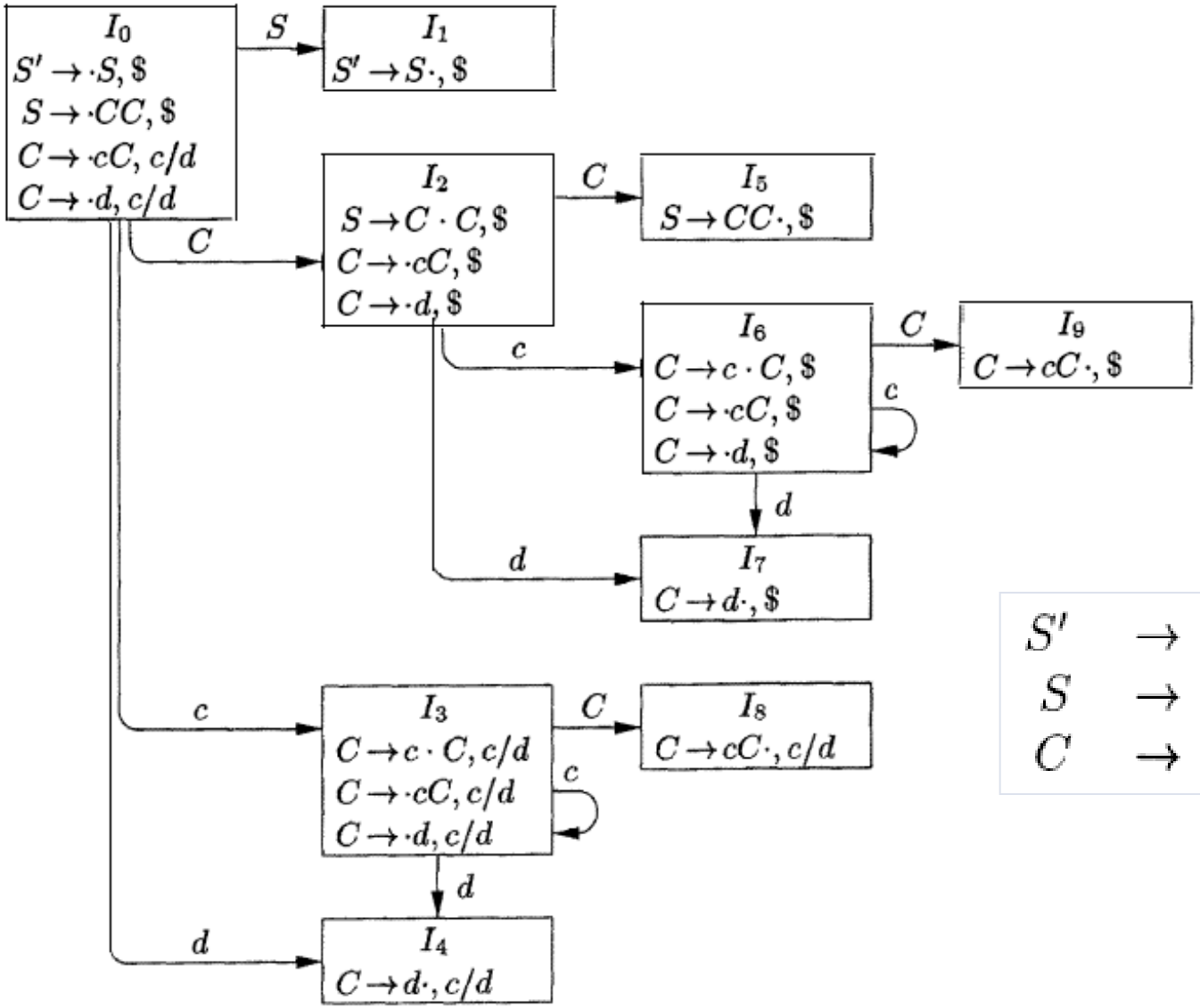
I_6 :
$C \rightarrow c\cdot C, \ \$$
$C \rightarrow \cdot cC, \ \$$
$C \rightarrow \cdot d, \ \$$

I_7 : $C \rightarrow d\cdot, \ \$$

I_8 : $C \rightarrow cC\cdot, \ c/d$

I_9 : $C \rightarrow cC\cdot, \ \$$

例……LR(1)项集族的构造



$$
\begin{array}{lcl}
S' & \to & S \\
S & \to & C\ C \\
C & \to & c\ C \mid d
\end{array}
$$

规范LR(1)语法分析表

输入：一个增广文法 G'。

输出：G'的规范 LR 语法分析表的函数 ACTION 和 GOTO。

方法：

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

2）语法分析器的状态 i 根据 I_i 构造得到。

状态 i 的语法分析动作按照下面的规则确定：

① 如果$[A \to \alpha \cdot a\beta, b]$在 I_i 中，并且 $\text{GOTO}(I_i, a) = I_j$，那么将 $\text{ACTION}[i, a]$ 设置为“移入 j”。这里 a 必须是一个终结符号。

② 如果$[A \to \alpha \cdot, a]$在 I_i 中且 $A \neq S'$，那么将 $\text{ACTION}[i, a]$设置为“归约 $A \to \alpha$”

③ 如果$[S' \to S \cdot, \$]$在 I_i 中，那么将 $\text{ACTION}[i, \$]$设置为“接受”。

如果根据上述规则会产生任何冲突动作，我们就说这个文法不是 LR(1)的。在这种情况下，这个算法无法为该文法生成一个语法分析器。

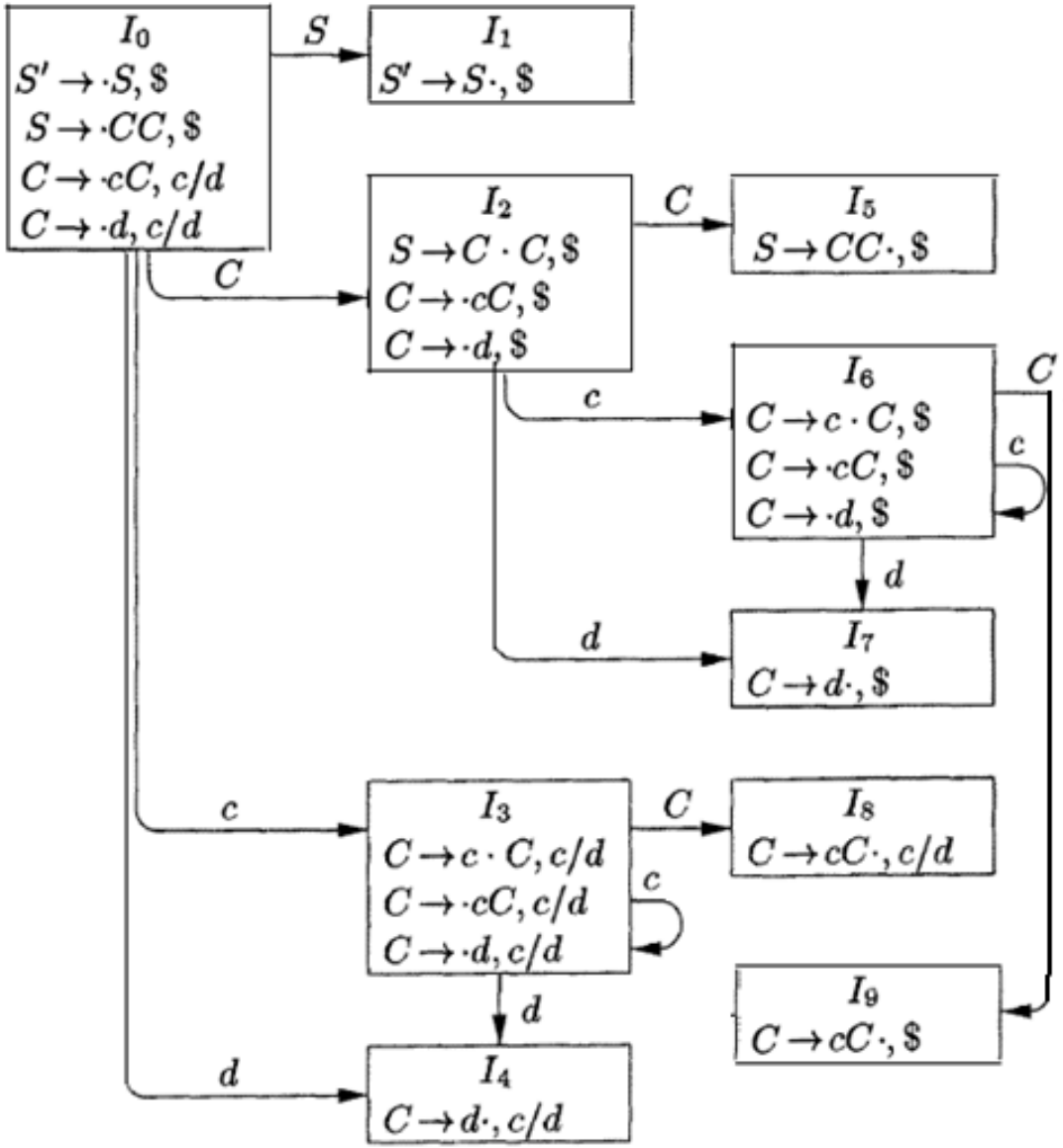
3）状态 i 相对于各个非终结符号 A 的 goto 转换按照下面的规则构造得到：

如果 $\text{GOTO}(I_i, A) = I_j$，那么 $\text{GOTO}[i, A] = j$。

4）所有没有按照规则(2)和(3)定义的分析表条目都设为“报错”。

5）语法分析器的初始状态是由包含$[S' \to \cdot S, \$]$的项集构造得到的状态。

例……规范LR(1)语法分析表



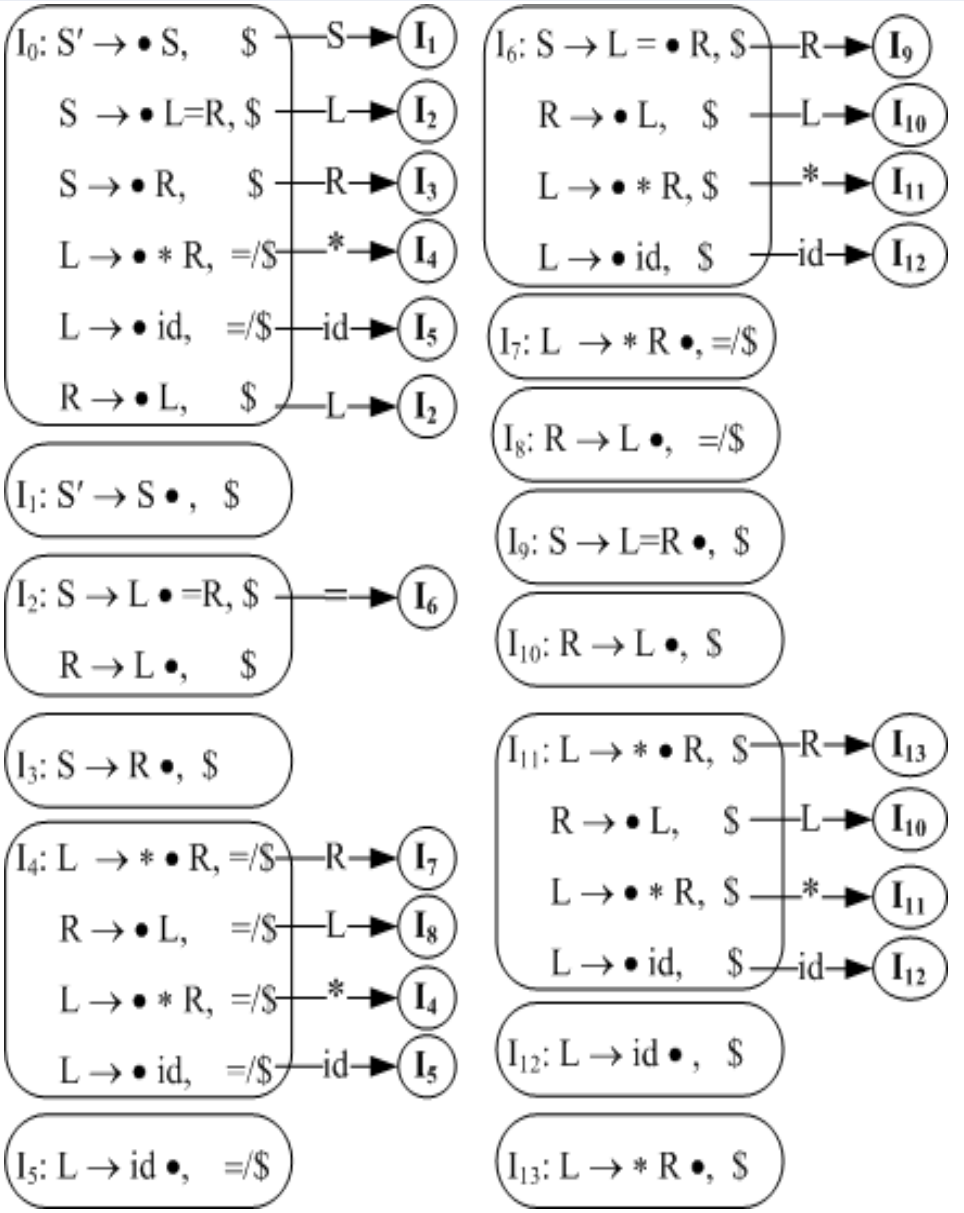
STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

LR(1)文法

- 规范LR(1)语法分析器
  - 以上算法生成的分析表称为规范LR(1)语法分析表
  - 使用规范LR(1)分析表的LR分析器称为规范LR(1)分析器
- LR(1)文法
  - 如果为文法G构造的LR(1)分析表中不含多重定义的条目，即无动作冲突，则G是LR(1)文法
- 每个SLR(1)文法都是LR(1)文法
  - 对于一个SLR(1)文法而言，规范LR(1)语法分析器的状态要比同一文法对应的SLR分析器的状态多

0. $S' \rightarrow S$　1. $S \rightarrow L = R$　2. $S \rightarrow R$

3. $L \rightarrow * R$　4. $L \rightarrow id$　5. $R \rightarrow L$



状态	ACTION				GOTO		
	=	*	id	$	S	L	R
0		s4	s5		1	2	3
1				acc			
2	s6			r5			
3				r2			
4		s4	s5			8	7
5	r4			r4			
6		s11	s12			10	9
7	r3			r3			
8	r5			r5			
9				r1			
10				r5			
11		s11	s12			10	13
12				r4			
13				r3			

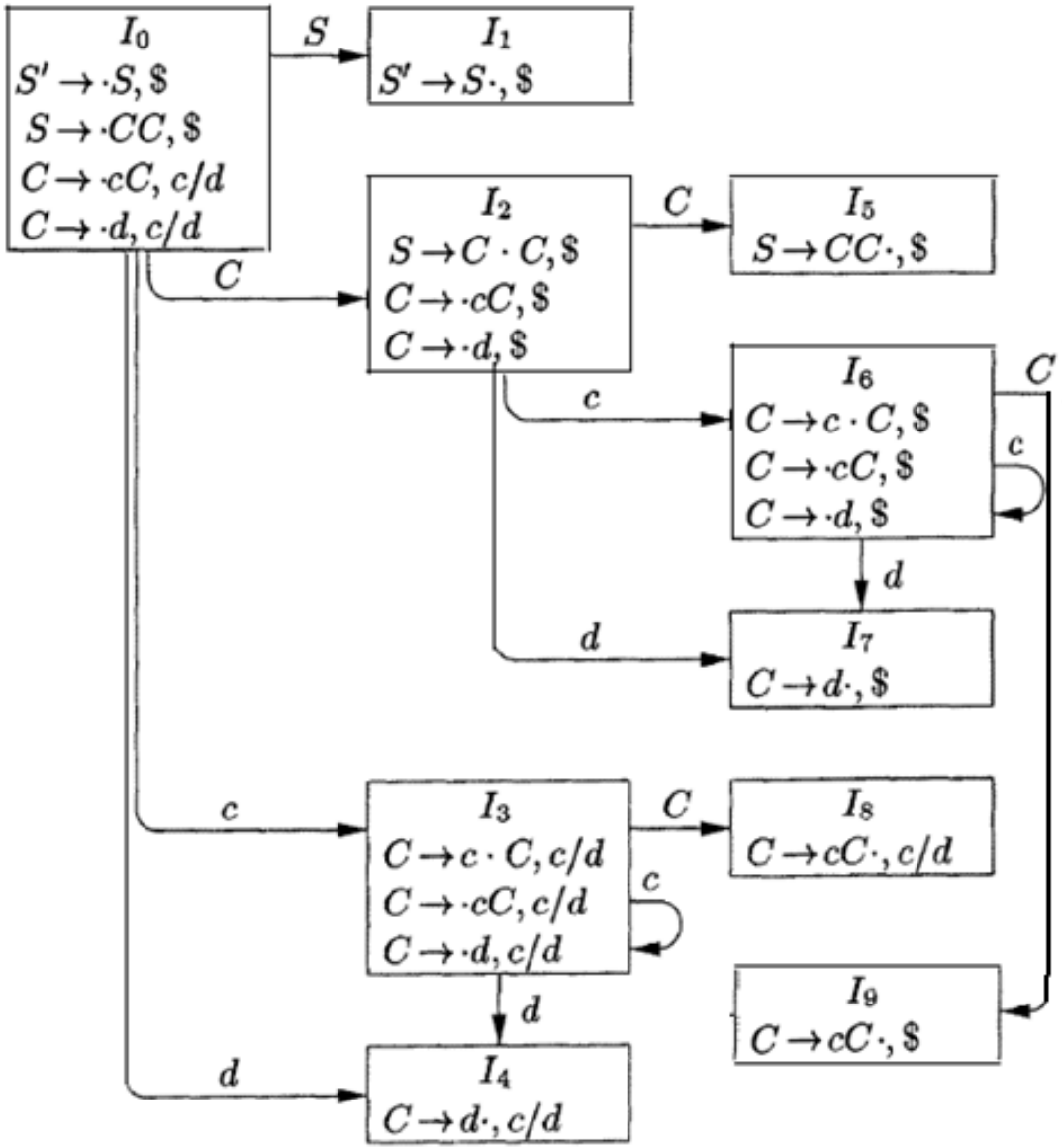
构造LALR语法分析表

- 合并相同核心的LR(1)项集
  - 项集的核心就是LR(1)项集的第一个分量的集合，如

$$I_4: \quad C \to d\cdot,\ c/d \qquad I_7: \quad C \to d\cdot,\ \$$$

  - 一个核心就是当前正处理的文法的LR(0)项集，一个LR(1)文法可能产生多个具有相同核心的项集
- 合并相同核心的LR(1)项集的影响
  - *GOTO*函数
    - 因为*GOTO(I,X)*的核心只由I的核心决定，所以一组被合并的项集的*GOTO*目标也可以被合并；即在合并项集时可以相应地修改*GOTO*函数
  - *ACTION*函数
    - 也要加以修改，以反映出被合并的所有项集的非报错动作

例……构造LALR分析表



I_{36}: $C \to c\cdot C,\ c/d/\$$
$C \to \cdot cC,\ c/d/\$$
$C \to \cdot d,\ c/d/\$$

I_{47}: $C \to d\cdot,\ c/d/\$$

I_{89}: $C \to cC\cdot,\ c/d/\$$

STATE	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

构造LALR语法分析表

- 合并相同核心的项集是否会引起动作冲突？
  - 不会产生新的移入-归约冲突
    - 移入动作仅由核心决定，只有归约动作才考虑向前看符号
  - 可能引起归约-归约冲突

假设有一个 LR(1)文法，我们将所有具有相同核心的状态替换为它们的并集，假设在并集中有一个项$[A\rightarrow\alpha\cdot, a]$要求按照$A\rightarrow\alpha$进行归约，同时另一个项$[B\rightarrow\beta\cdot a\gamma, b]$要求进行移入，那么就会出现在向前看符号 a 上的冲突。

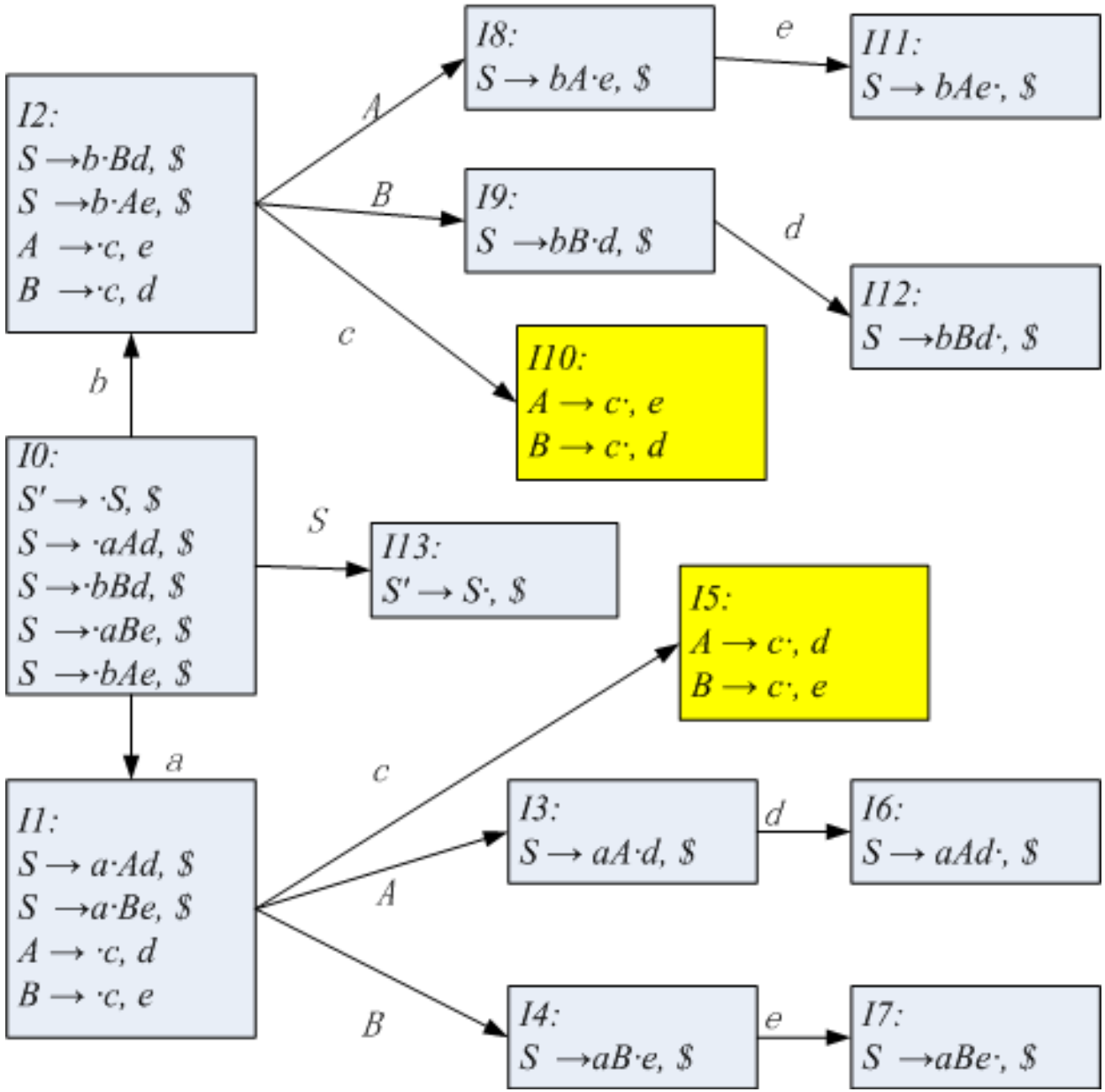
此时必然存在某个被合并进来的项集中包含项$[A\rightarrow\alpha\cdot, a]$，同时因为所有这些状态的核心都是相同的，所以这个被合并进来的项集中必然还包含项$[B\rightarrow\beta\cdot a\gamma, c]$，其中 c 是某个终结符号。如果这样的话，这个状态中同样也有在输入 a 上的移入/归约冲突，因此这个文法不是假设的 LR(1)文法。

因此，合并相同核心的状态不会产生出原有状态中没有出现的移入/归约冲突。

例……构造LALR语法分析表

$$
\begin{aligned}
S' &\rightarrow S \\
S &\rightarrow a\ A\ d \\
S &\rightarrow b\ B\ d \\
S &\rightarrow a\ B\ e \\
S &\rightarrow b\ A\ e \\
A &\rightarrow c \\
B &\rightarrow c
\end{aligned}
$$

$$
\begin{aligned}
A &\rightarrow c\cdot,\ d/e \\
B &\rightarrow c\cdot,\ d/e
\end{aligned}
$$



构造LALR分析表算法

❖ 基本思想

- 构造出LR(1)项集，如果没有出现冲突，就将具有相同核心的项集合并；根据合并后得到的项集构造语法分析表

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 $C=\{I_0, I_1, \cdots, I_n\}$。

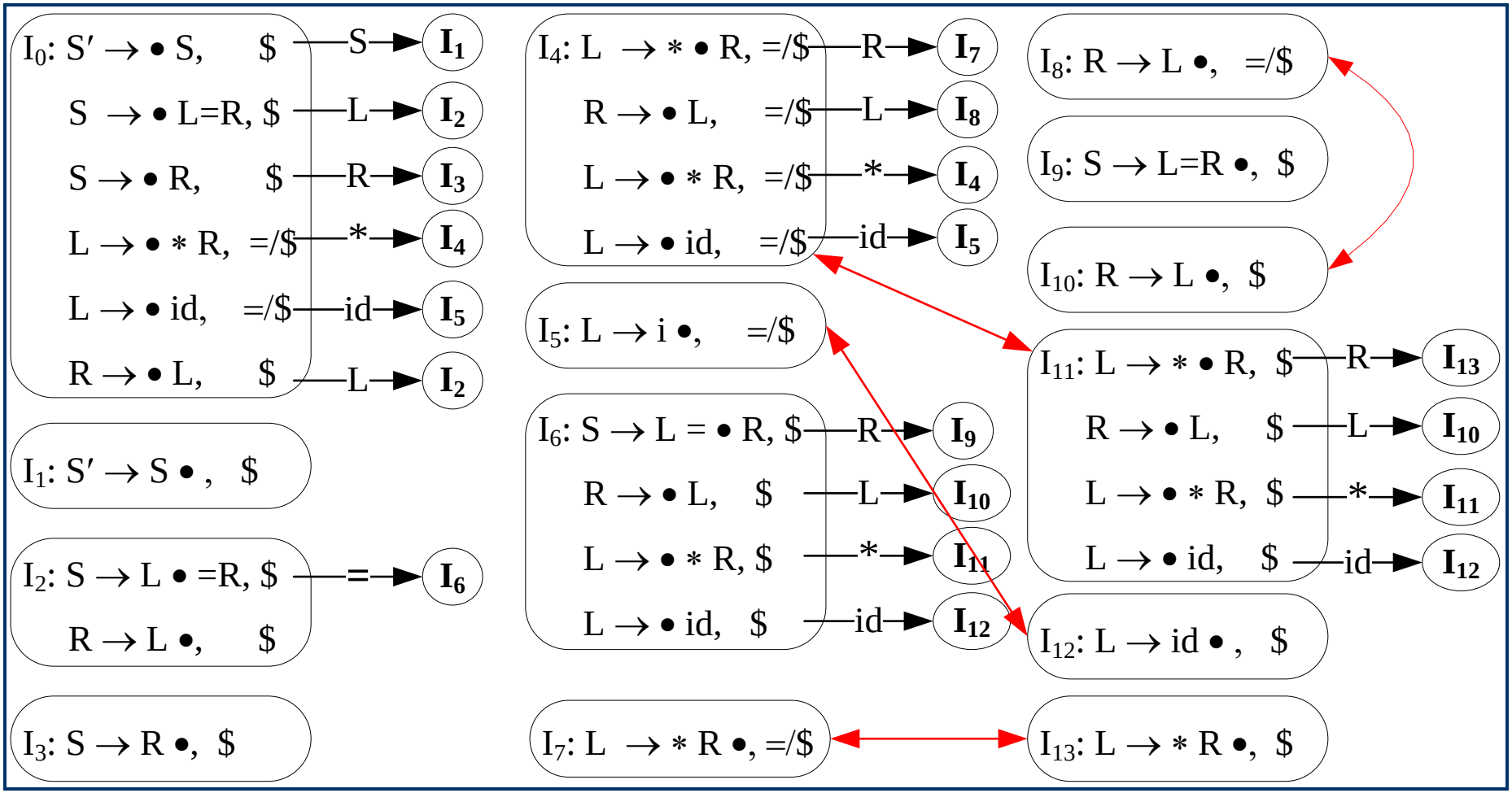
2）对于 LR(1)项集中的每个核心，找出所有具有这个核心的项集，并将这些项集替换为它们的并集。

3）令 $C'=\{J_0, J_1, \cdots, J_m\}$是得到的 LR(1)项集族。按照规范 LR 语法分析表的构造算法根据 J_i 构造得到状态 i 的语法分析动作。如果存在分析动作冲突，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。 □

例……构造LALR分析表

0. $S' \to S$　1. $S \to L = R$　2. $S \to R$
3. $L \to * R$　4. $L \to i$　5. $R \to L$



LALR文法

- LALR文法
  - 以上算法生成的分析表称为文法G的LALR分析表。
  - 如果对文法G能构造出没有动作冲突的LALR语法分析表，那么G就是LALR(1)文法。
  - 算法第（3）步构造得到的项集族被称为LALR(1)项集族。
- 与SLR和规范LR的比较
  - LALR分析表比规范LR分析表小，在实践中经常使用
  - LALR比LR(1)的分析能力弱，在遇到错误输入时可能多做无效的归约，但不会多移入任何符号
  - 一个文法的SLR和LALR分析表总是具有相同数量的状态，但LALR方法能处理某些SLR无法处理的情况

练习4.7（作业）

练习 4.7.1：为练习 4.2.1 的文法 $S \rightarrow S\ S\ +\ |\ S\ S\ *\ |\ a$ 构造

1）规范 LR 项集族。

2）LALR 项集族。

练习 4.7.2：对练习 4.2.2(1)～(7)的各个(增广)文法重复练习 4.7.1。

！ **练习 4.7.3**：对练习 4.7.1 的文法，使用算法 4.63，根据该文法的 LR(0)项集的内核构造出它的 LALR 项集族。

！ **练习 4.7.4**：说明下面的文法

$$S \rightarrow A\ a\ |\ b\ A\ c\ |\ d\ c\ |\ b\ d\ a \qquad A \rightarrow d$$

是 LALR(1)的，但不是 SLR(1)的。

！ **练习 4.7.5**：说明下面的文法

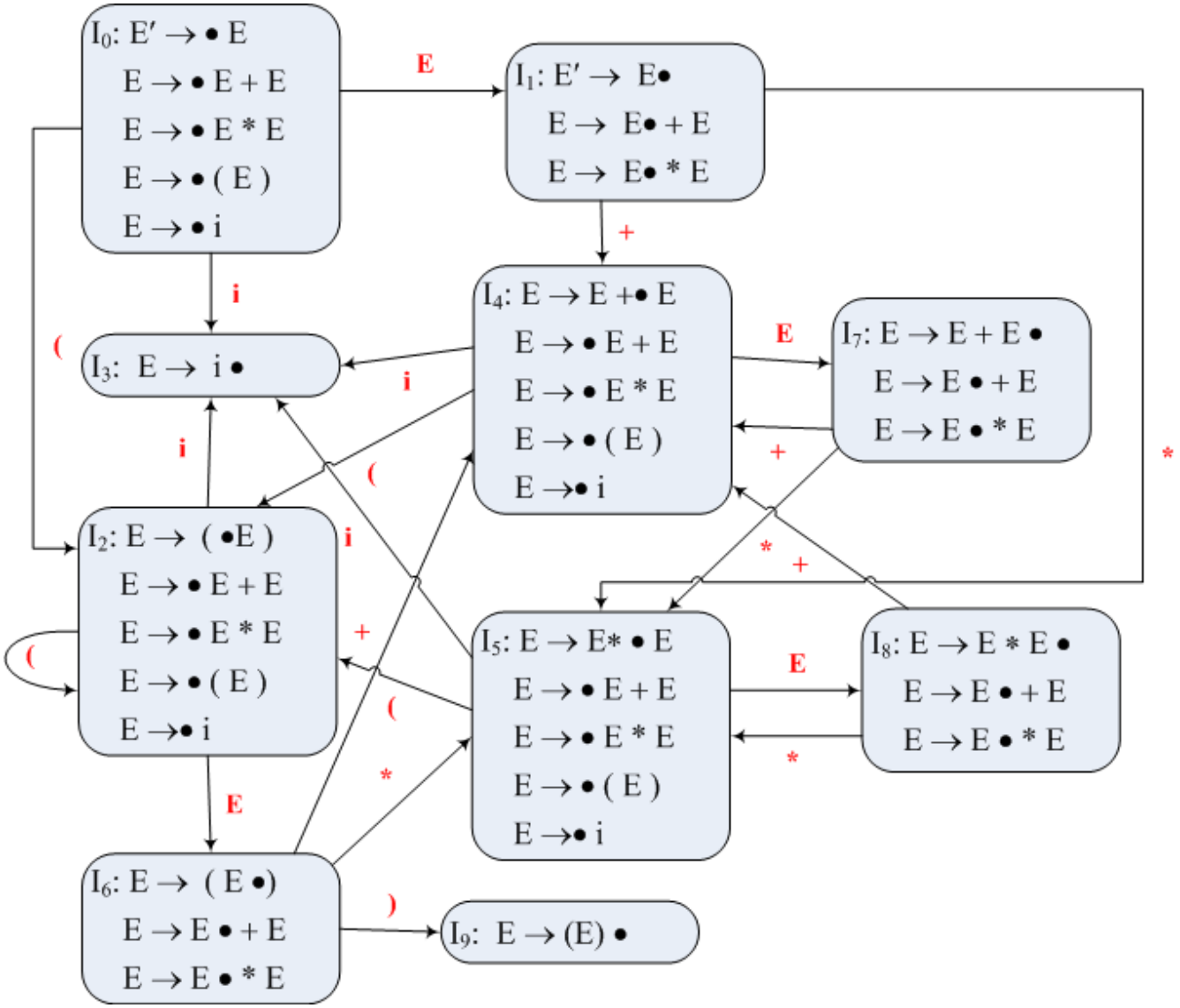
$$S \rightarrow A\ a\ |\ b\ A\ c\ |\ B\ c\ |\ b\ B\ a \qquad A \rightarrow d \qquad B \rightarrow d$$

是 LR(1)的，但不是 LALR(1)的。

LR方法和二义性文法

- 二义性文法都不是LR的
- 二义性文法的优点
  - 有时候比无二义文法更简短更自然，如表达式文法
- 对二义性文法的处理
  - 一般会给出消除二义性的规则，使得每个句子只有一棵语法分析树
  - 应用这些规则，语言的规约在整体上是无二义的，也可以遵循这些二义性解决方案构造出无冲突的LR语法分析器
  - 注意：谨慎使用二义性文法

例……二义性文法的应用



(0) $E' \rightarrow E$

(1) $E \rightarrow E + E$

(2) $E \rightarrow E * E$

(3) $E \rightarrow (E)$

(4) $E \rightarrow i$

例……二义性文法的应用

(0) E' → E
(1) E → E + E
*(2) E → E * E*
(3) E → (E)
(4) E → i

I_7: E → E + E •
E → E • + E
E → E • * E

I_8: E → E * E •
E → E • + E
E → E • * E

解决冲突的方法：
规定*比+的优先级高

状态	ACTION						GOTO
	i	+	*	(	)	$	E
0	s3			s2			1
1		s4	s5			acc	
2	s3			s2			6
3		r4	r4		r4	r4	
4	s3			s2			7
5	s3			s2			8
6		s4	s5		s9		
7		r1 ~~s4~~	~~r1~~ s5		r1	r1	
8		r2 ~~s4~~	r2 ~~s5~~		r2	r2	
9		r2	r3		r3	r3	

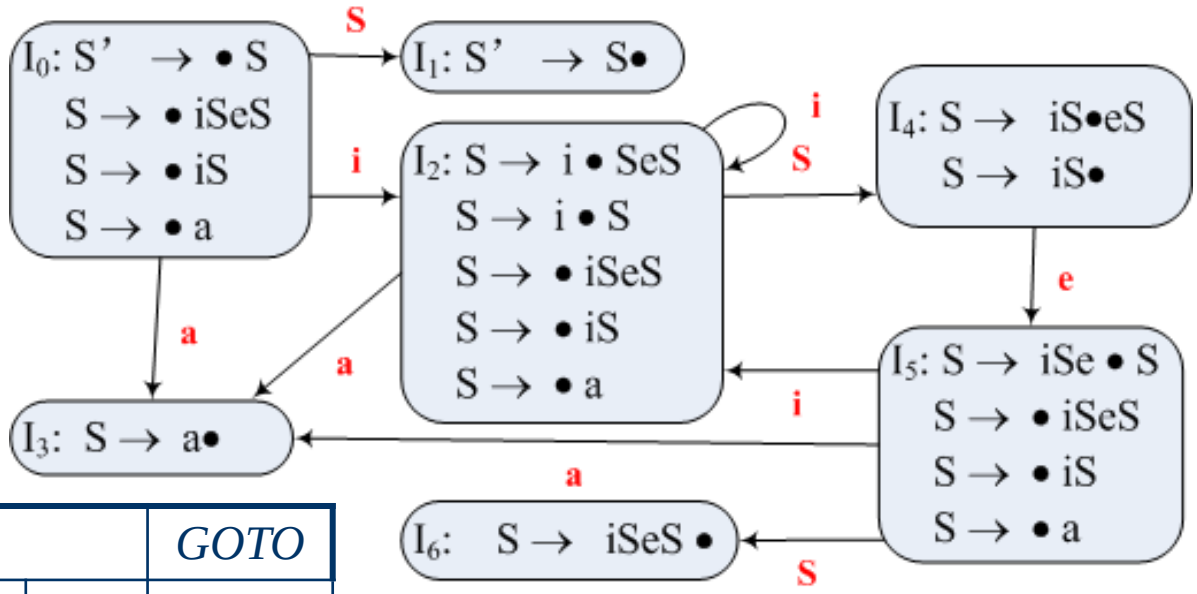
例……二义性文法的应用

(0) $S' \rightarrow S$

(1) $S \rightarrow iSeS$

(2) $S \rightarrow iS$

(3) $S \rightarrow a$



状态	ACTION				GOTO
	i	e	a	$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		~~r2~~ **s5**		r2	
5	s2		s3		6
6		r1		r1	

解决冲突的方法：

规定*else*与最近的*if*匹配

语法分析器生成工具

Yacc简介

Yacc

- 语法分析器生成工具Yacc(*yet another compiler-compiler*)

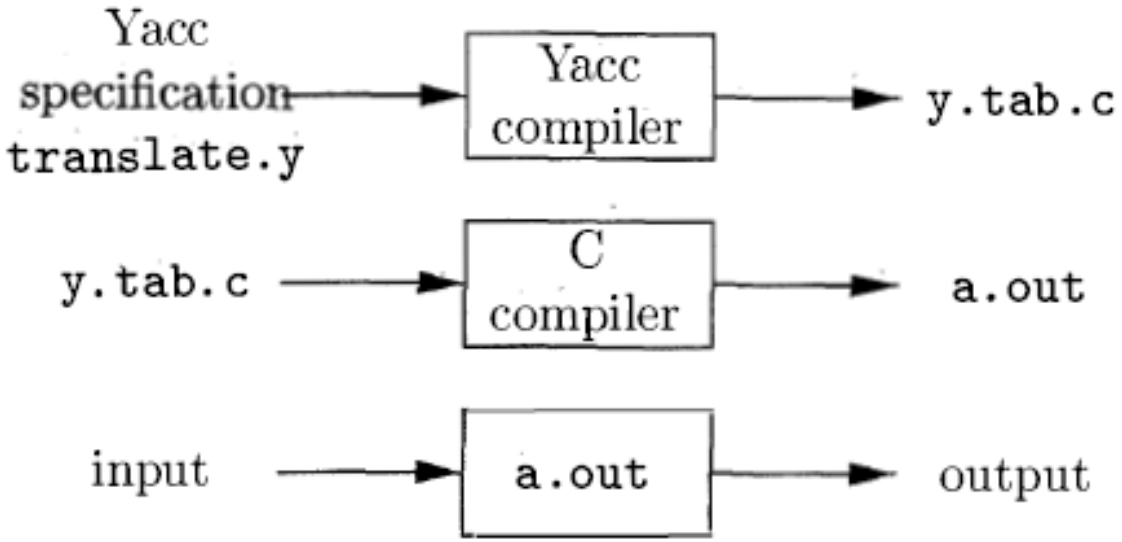


图 4-57　用 Yacc 创建一个输入/输出翻译器

Yacc源程序

- 声明
  - 通常的C声明
  - 词法单元的声明
- 翻译规则
  - 每个规则由一个文法产生式和一个相关联的语义动作组成
  - 一个Yacc语义动作是一个C语句序列
- 辅助性C语言例程
  - 必须提供一个名为yylex()的词法分析器，通常是由Lex生成的
  - 还可以添加错误恢复例程等

```
declarations
%%
translation rules
%%
supporting C routines
```

例……Yacc源程序

$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid T \\
T & \rightarrow & T * F \mid F \\
F & \rightarrow & (E) \mid \textbf{digit}
\end{array}
$$

```
%{
#include <ctype.h>
%}

%token DIGIT

%%
line   : expr '\n'          { printf("%d\n", $1); }
       ;
expr   : expr '+' term      { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor    { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'       { $$ = $2; }
       | DIGIT
       ;

%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

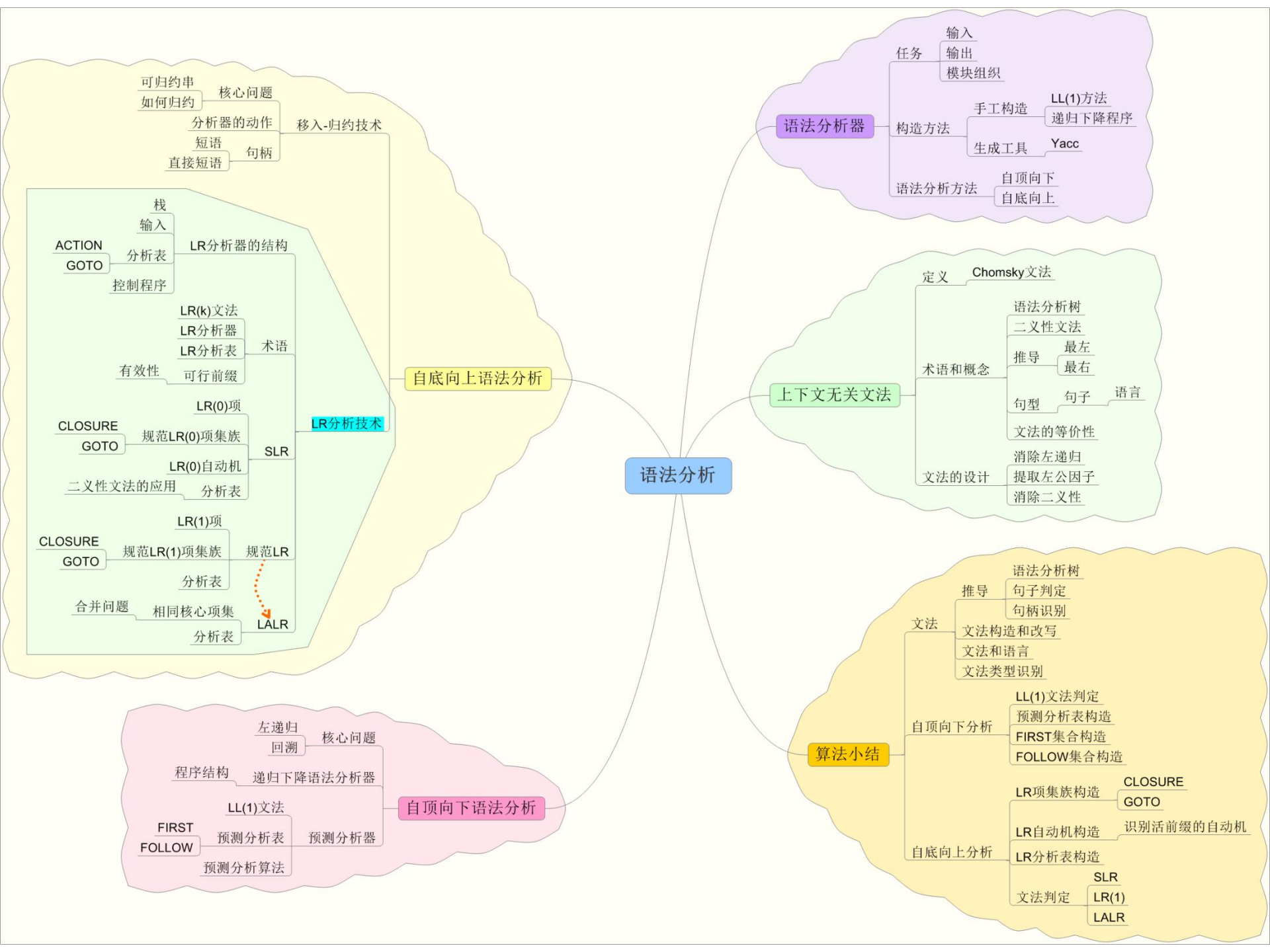
```
D:\temp>bison -dyv cal.y

D:\temp>flex cal.l

D:\temp>gcc lex.yy.c y.tab.c -o cal

D:\temp>cal
2+3
ans=5
2-3
ans=-1
2*3
ans=6
2/3
ans=0
```

本章小结


语法分析
语法分析器
任务
输入
输出
模块组织
构造方法
手工构造
LL(1)方法
递归下降程序
生成工具
Yacc
语法分析方法
自顶向下
自底向上
上下文无关文法
定义
Chomsky文法
术语和概念
语法分析树
二义性文法
推导
最左
最右
句型
句子
语言
文法的等价性
文法的设计
消除左递归
提取左公因子
消除二义性
算法小结
文法
推导
语法分析树
句子判定
句柄识别
文法构造和改写
文法和语言
文法类型识别
自顶向下分析
LL(1)文法判定
预测分析表构造
FIRST集合构造
FOLLOW集合构造
自底向上分析
LR项集族构造
CLOSURE
GOTO
LR自动机构造
识别活前缀的自动机
LR分析表构造
文法判定
SLR
LR(1)
LALR
自底向上语法分析
移入-归约技术
核心问题
可归约串
如何归约
分析器的动作
句柄
短语
直接短语
LR分析技术
LR分析器的结构
栈
输入
分析表
ACTION
GOTO
控制程序
术语
LR(k)文法
LR分析器
LR分析表
可行前缀
有效性
SLR
LR(0)项
规范LR(0)项集族
CLOSURE
GOTO
LR(0)自动机
分析表
二义性文法的应用
规范LR
LR(1)项
规范LR(1)项集族
CLOSURE
GOTO
分析表
LALR
相同核心项集
分析表
合并问题
自顶向下语法分析
核心问题
左递归
回溯
递归下降语法分析器
程序结构
预测分析器
LL(1)文法
预测分析表
FIRST
FOLLOW
预测分析算法