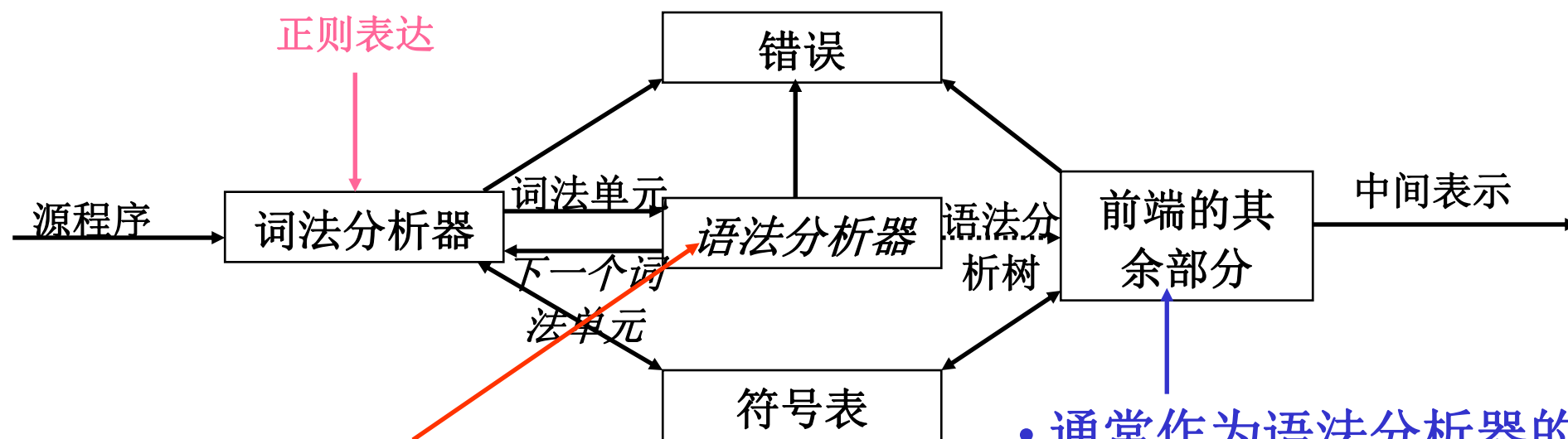


第四章 语法分析

学习内容

- 语法分析器概述
- 上下文无关文法
- 自顶向下分析方法：递归实现、表驱动
- 自底向上分析方法
 - LR分析方法
 - SLR
 - 规范LR
 - LALR

语法分析器的作用



- 利用语法检查单词流的语法结构
- 构造语法分析树
- 语法错误和修正
- 识别正确语法
- 报告错误

- 通常作为语法分析器的一部分实现
- 包括对单词扩充信息，以进行类型检查、语义分析等工作

语法错误处理

不同层次的错误

- 词法：拼写错误（`j=1.05e`，指数表示错误）
- 语法：单词漏掉、顺序错误（花括号不配对）
- 语义：类型错误（声明 `void f()` 和调用 `aa = f()`）
- 逻辑：无限循环/递归调用（`==` \longrightarrow `=`）

语法错误处理为重点

- 语法错误相对较多
- 编译器容易高效检测

错误处理目标

三个“简单”的目标

- 清楚、准确地检测、报告错误及其发生位置
- 快速恢复，继续编译，以便发现后续错误
- 不能对正确程序的编译速度造成很大影响

LL, LR, 可最快速度发现错误

- 可行前缀特性, **viable-prefix property**
- 一个输入前缀不是语言中任何符号串前缀——发生错误

错误恢复策略

1. 恐慌模式的恢复

- 丢弃单词，直到发现“同步”单词
- 设计者指定同步单词集，{**end**, “;”, “}”, ...}
- 缺点
 - 丢弃输入⇒遗漏定义，造成更多错误
 - 遗漏错误
- 优点
 - 简单⇒适合每个语句一个错误的情况

错误恢复策略

2. 短语层次的恢复

- 局部修正，继续分析
- “,” \Rightarrow “;”，删除“,”，插入“;”
- 同样由设计者指定修正方法
- 避免无限循环
- 有些情况不适用
- 与恐慌模式相结合，避免丢弃过多单词

错误恢复策略

3. 错误产生式

- 理解、描述错误模式
- 文法添加生成错误语句的产生式
- 拓广文法→语法分析器程序
- 如，对C语言赋值语句，为“:=”添加规则
报告错误，但继续编译
- 错误检测信息+自动修正

错误恢复策略

4. 全局纠正

- 错误程序 → 正确程序
- 寻找最少修正步骤，插入、删除、替换
- 不正确输入 x ，文法 G $\xrightarrow{\text{最少修正 } x \rightarrow y}$ y 对应的语法分析树
- 过于复杂，时空效率低

学习内容

- 语法分析器概述
- 上下文无关文法
- 语法分析
 - 自顶向下分析方法：递归实现、表驱动
 - 自底向上分析方法
 - LR分析方法
 - SLR
 - 规范LR
 - LALR

上下文无关文法

描述语言的语法结构的形式规则

定义：四元式(V_T, V_N, S, \mathcal{P})

- V_T : 终结符号（单词）集， T
- V_N : 非终结符号（语法变量）集， NT ，定义了文法/语言可生成的符号串集合
- S : $S \in NT$ ，开始符号，定义语言的所有符号串
- \mathcal{P} ，产生式集， PR ， $NT \rightarrow (T \mid NT)^*$
规则 $\rightarrow T$ 、 NT 如何组合，生成语言的合法符号串

例：简单表达式

$expr \rightarrow expr + term$

$expr \rightarrow expr - term$

$expr \rightarrow term$

$term \rightarrow term * factor$

$term \rightarrow term / factor$

$term \rightarrow factor$

$factor \rightarrow (expr)$

$factor \rightarrow id$

蓝色符号——T,
黑色符号——NT



例：利用符号约定简化文法

$expr \rightarrow expr + term$

$expr \rightarrow expr - term$

$expr \rightarrow term$

$term \rightarrow term * factor$

$term \rightarrow term / factor$

$term \rightarrow factor$

$factor \rightarrow (expr)$

$factor \rightarrow id$

表达式文法简化后结果

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T / F \mid F$

$F \rightarrow (E) \mid id$



推导

描述文法定义语言的过程

自顶向下构造语法分析树的精确描述

将产生式用作重写规则

- 由开始符号起始
- 每个步骤将符号串转换为另一个符号串
- 转换规则：利用某个产生式，将符号串中出现的其左部NT替换为其右部符号串



推导

$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$

$E \rightarrow -E$, E 可替换为 $-E$

$E \Rightarrow -E$, “ E 直接推出 $-E$ ”

$E * E \Rightarrow (E) * E$

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathbf{id})$

替换序列, $E \rightarrow -(\mathbf{id})$ 的一个推导



定义

形式化定义

- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ 仅当存在产生式 $A \rightarrow \gamma$
- $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n \xrightarrow{\quad} \alpha_1 \overset{*}{\Rightarrow} \alpha_n$
- 若 $\alpha \overset{*}{\Rightarrow} \beta$ 且 $\beta \rightarrow \gamma$, 则 $\alpha \overset{*}{\Rightarrow} \gamma$

\Rightarrow , “一步推导”, “直接推出”, 推导步数=1

$\overset{+}{\Rightarrow}$, “一步或多步推导”, 推导步数 ≥ 1

$\overset{*}{\Rightarrow}$, “0步或多步推导”, 推导步数 ≥ 0



推导与语言的关系

文法G，开始符号S，生成的语言L(G)

终结符号串w

$$w \in L(G) \Leftrightarrow S \xRightarrow{+} w$$

w: G的一个句子

CFG生成上下文无关语言

两个CFG生成相同语言，两个CFG等价

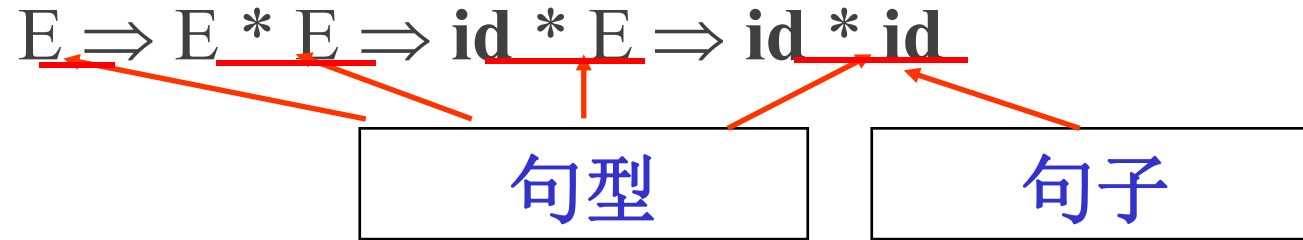
$S \xRightarrow{*} \alpha$ ， α 可能包含NT

α : G的一个句型

句子: 不包含NT的句型



例:



$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$

另一种推导过程

$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(E+\mathbf{id}) \Rightarrow -(\mathbf{id}+\mathbf{id})$



最左推导和最右推导

最左推导：总替换最左边的NT

$$E \Rightarrow -E \Rightarrow_{\text{lm}} -(E) \Rightarrow_{\text{lm}} -(E+E) \Rightarrow_{\text{lm}} -(\text{id}+E) \Rightarrow_{\text{lm}} -(\text{id}+\text{id})$$

最右推导：总替换最右边的NT

$$E \Rightarrow -E \Rightarrow_{\text{rm}} -(E) \Rightarrow_{\text{rm}} -(E+E) \Rightarrow_{\text{rm}} -(E+\text{id}) \Rightarrow_{\text{rm}} -(\text{id}+\text{id})$$

形式化定义： $A \rightarrow \delta$

$$wA\gamma \Rightarrow_{\text{lm}} w\delta\gamma, \text{ } w \text{ 只含 } T$$

$$\beta Aw \Rightarrow_{\text{rm}} \beta\delta w, \text{ } w \text{ 只含 } T$$

$$S \xRightarrow{*}_{\text{lm}} \alpha, \text{ } \alpha: \text{ 最左句型}$$



语法分析树和推导

语法树：推导的图示，但不体现推导过程的顺序

- 内部节点：非终结符A
- 内部结点A的孩子节点：左 \rightarrow 右，对应推导过程中替换A的右部符号串的每个符号
- 叶：由左至右 \rightarrow 句型，结果（**yield**），边缘（**frontier**）



语法树与推导的关系

一个推导过程: $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \dots \Rightarrow \alpha_n$

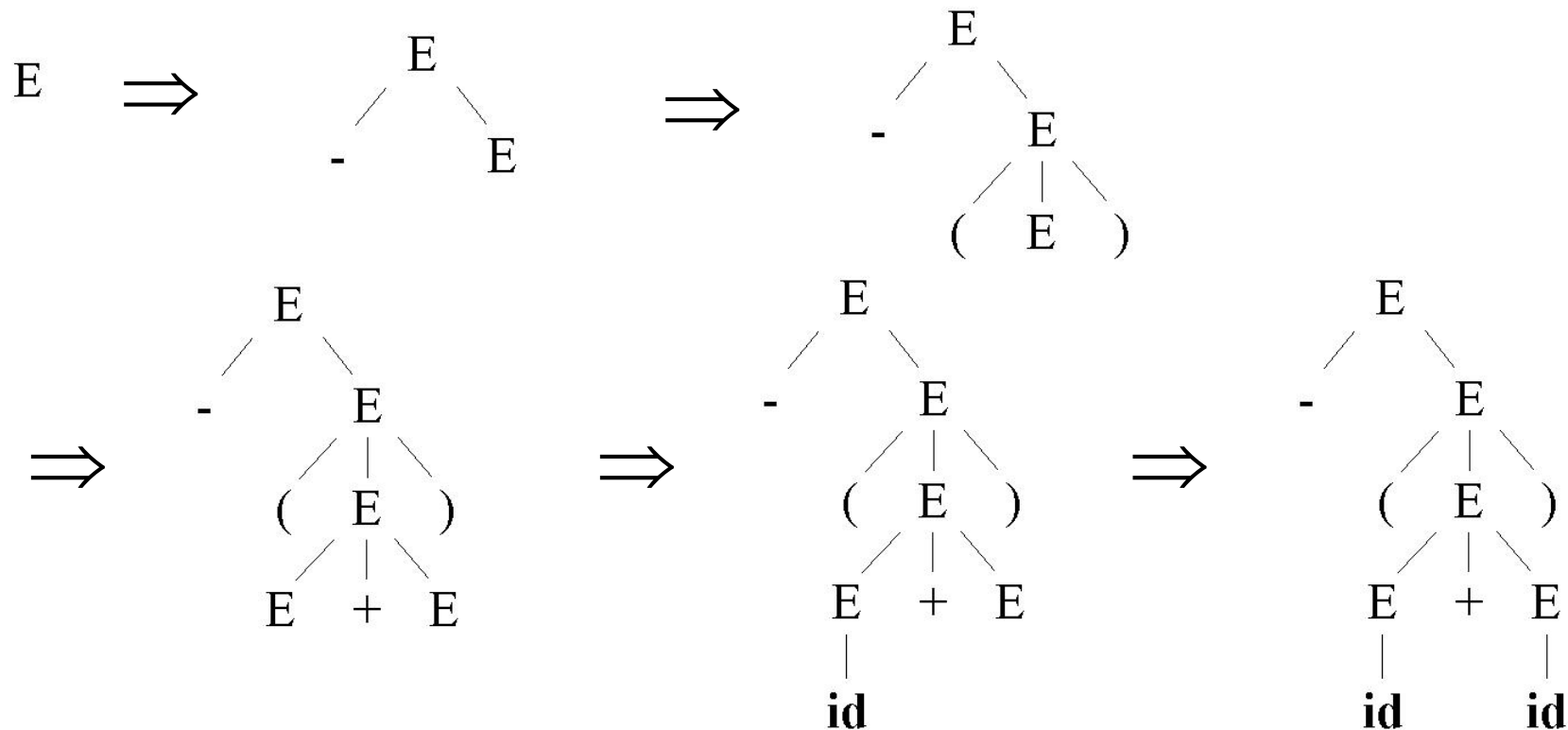
句型 α_i : 一个结果为 α_i 的语法分析树

- $\alpha_1 \equiv A$, 单节点, 标记为A
- $\alpha_{i-1} = X_1 X_2 \dots X_k$ 对应语法树T
- 第i步推导, $X_j \rightarrow Y_1 Y_2 \dots Y_r$
- T的第j个叶节点, 添加r个孩子节点 Y_1, Y_2, \dots, Y_r , 特殊情况, $r=0$, 一个孩子 ε



$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$$
$$\mathbf{-(id+id)}$$

例:

$$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(E+E) \Rightarrow -(\mathbf{id}+E) \Rightarrow -(\mathbf{id}+\mathbf{id})$$


一棵语法树 \leftrightarrow 多个推导

一棵语法树 \leftrightarrow 唯一最左推导, 唯一最右推导



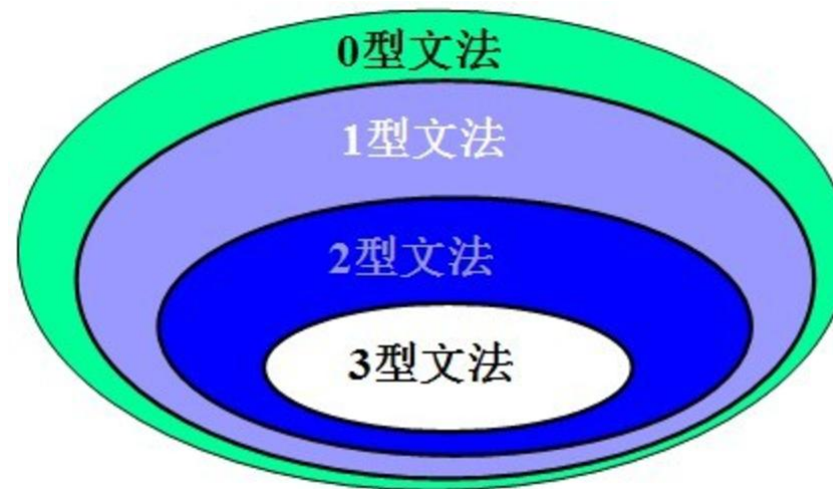
文法对比

正则表达式（3型）

- 词法分析的基础
- 描述正则语言
- 描述能力不够, $a^n b^n, n \geq 1$

上下文无关文法（2型）

- 语法分析的基础
- 描述程序语言结构
- 上下文无关语言



正则表达式与上下文无关文法

正则表达式可描述的语言CFG均可描述, $(a|b)^*abb$

$$A_0 \rightarrow aA_0 \mid aA_1 \mid bA_0$$

$$A_1 \rightarrow bA_2$$

$$A_2 \rightarrow bA_3$$

$$A_3 \rightarrow \varepsilon$$

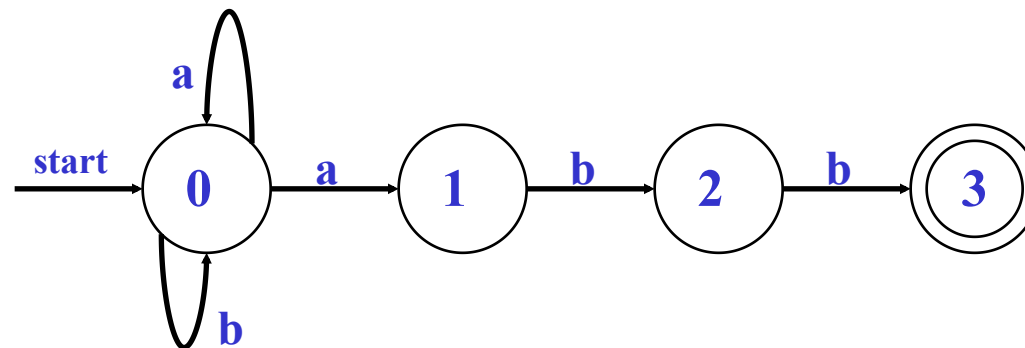
正则语言 \subset 上下文无关语言

Reg. Lang.

CFLs



NFA \rightarrow CFG

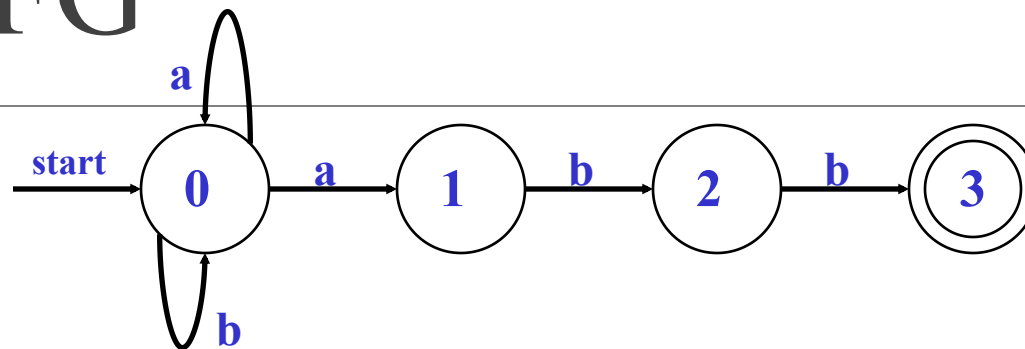


$(a|b)^*abb$

1. 状态 $i \rightarrow$ 非终结符 A_i : A_0, A_1, A_2, A_3
2. $i \xrightarrow{a} j \rightarrow A_i \rightarrow aA_j$
 $\left\{ \begin{array}{l} : A_0 \rightarrow aA_0, A_0 \rightarrow aA_1 \\ : A_0 \rightarrow bA_0, A_1 \rightarrow bA_2 \\ : A_2 \rightarrow bA_3 \end{array} \right.$
3. $i \xrightarrow{\epsilon} j \rightarrow A_i \rightarrow A_j$
4. 若 i 为终态 $\rightarrow A_i \rightarrow \epsilon$: $A_3 \rightarrow \epsilon$
5. 若 i 为初态, A_i 为开始符号: A_0



NFA \rightarrow CFG



$(a|b)^*abb$

A_i 的含义是什么?

状态 $i \rightarrow$ 终态路径上的符号串集合

A_i 能否表示“初态 \rightarrow 状态 i 路径上的符号串集合”?

$$A_2 \rightarrow bA_3$$

$$A_2 \rightarrow A_1b$$

$$A_0 \rightarrow A_0a$$

$$A_0 \rightarrow A_0b$$

$$A_1 \rightarrow A_0a$$

$$A_2 \rightarrow A_1b$$

$$A_3 \rightarrow A_2b$$

$$A_0 \rightarrow \epsilon$$

变换规则如何修改?

文法变成什么样?



为什么还需要正则表达式？

1. 词法规则很简单，正则表达式描述能力足够
2. 正则表达式更简洁、更容易理解
3. 能更自动构造更高效的词法分析器
4. 使编译器前端更模块化

词法、语法规则的划分没有固定准则

- 正则表达式更适合描述标识符、常量、关键字...的结构
- CFG更适合描述单词的结构化联系、层次化结构，如括号匹配，if-then-else, ...

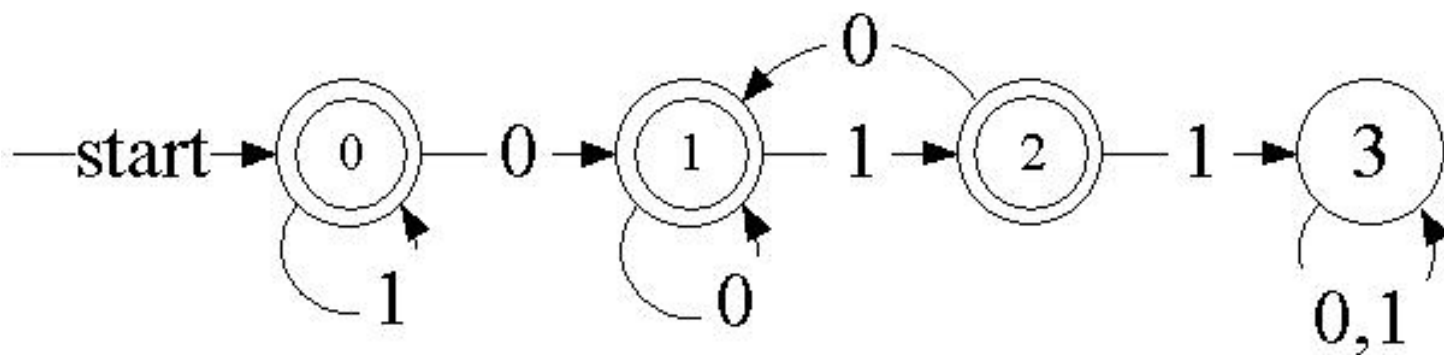


设计CFG练习

$$L = \{ a^n b b^{2n} \mid n \geq 0 \}$$

不包含子串011的0/1串

$$S \rightarrow b \mid aSbb$$



$$S \rightarrow 0 A \mid 1 S \mid \varepsilon$$

$$A \rightarrow 0 A \mid 1 B \mid \varepsilon$$

$$B \rightarrow 0 A \mid \varepsilon$$



设计CFG的难点

手工进行，无形式化方法

不同的语法分析方法对CFG有不同的特殊要求

- 如自顶向下分析方法和自底向上分析方法
- CFG设计完成后可能需要修改



CFG的修改

两个目的

- 去除“错误”
- 重写，满足特殊要求

不合要求的问题

- 二义性
- ϵ -moves
- 回路
- 左递归
- 左公因子



CFG的修改

- 二义性
- ϵ -moves
- 回路
- 左递归
- 左公因子



二义性文法

句子 \leftrightarrow 多个语法树, 多个最左(右)推导

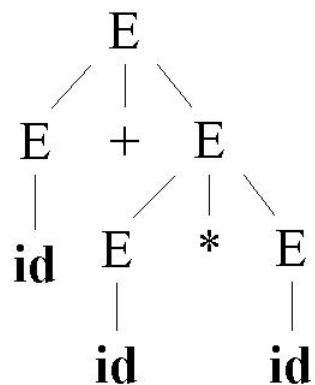
$$E \Rightarrow E + E$$

$$\Rightarrow \mathbf{id} + E$$

$$\Rightarrow \mathbf{id} + E * E$$

$$\Rightarrow \mathbf{id} + \mathbf{id} * E$$

$$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$



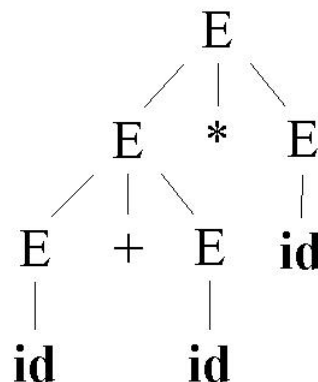
$$E \Rightarrow E * E$$

$$\Rightarrow E + E * E$$

$$\Rightarrow \mathbf{id} + E * E$$

$$\Rightarrow \mathbf{id} + \mathbf{id} * E$$

$$\Rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id}$$



$$E \rightarrow E + E \mid E * E \mid (E) \mid -E \mid \mathbf{id}$$

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

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



表达式文法简化后结果

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T / F \mid F$$
$$F \rightarrow (E) \mid \text{id}$$

消除二义性

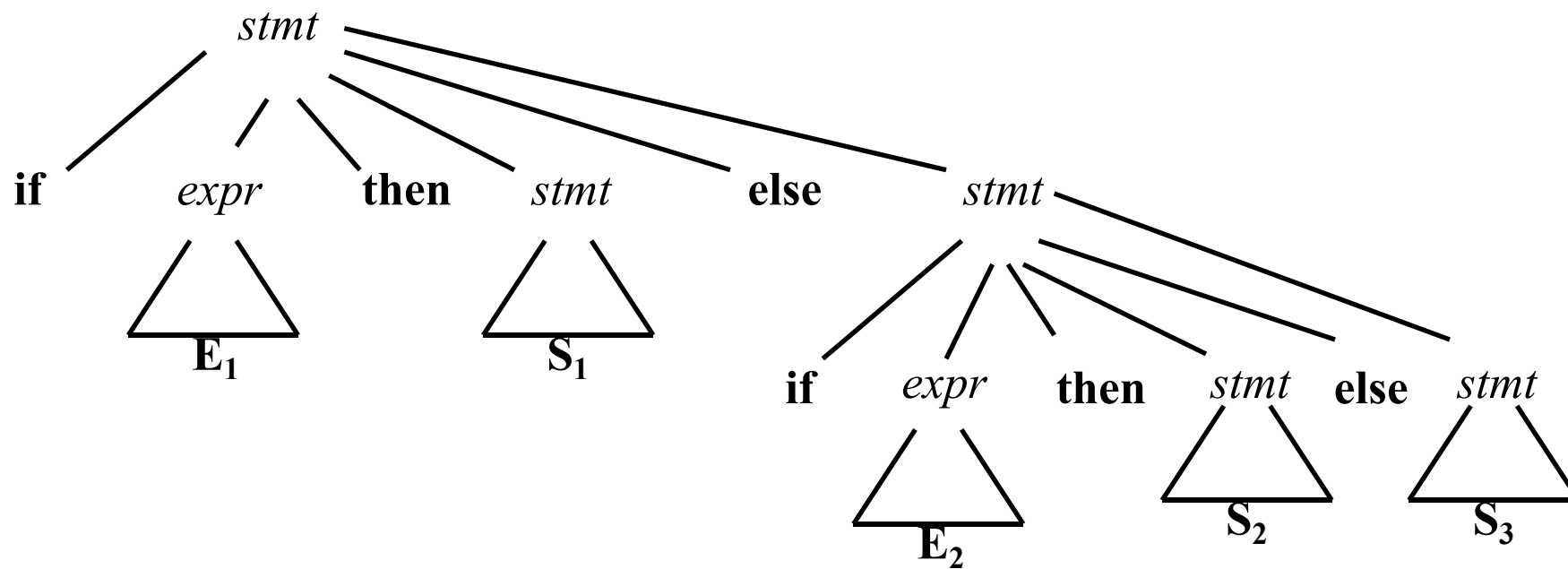
例子：条件分支语句

$stmt \rightarrow$ **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other** (任何其他形式的语句)



无二义性的句子

if E_1 then S_1 else if E_2 then S_2 else S_3 语法树如下



二义性句子

if E_1 then if E_2 then S_1 else S_2 有两种意义

if E_1 then
{ if E_2 then
S_1
else
S_2

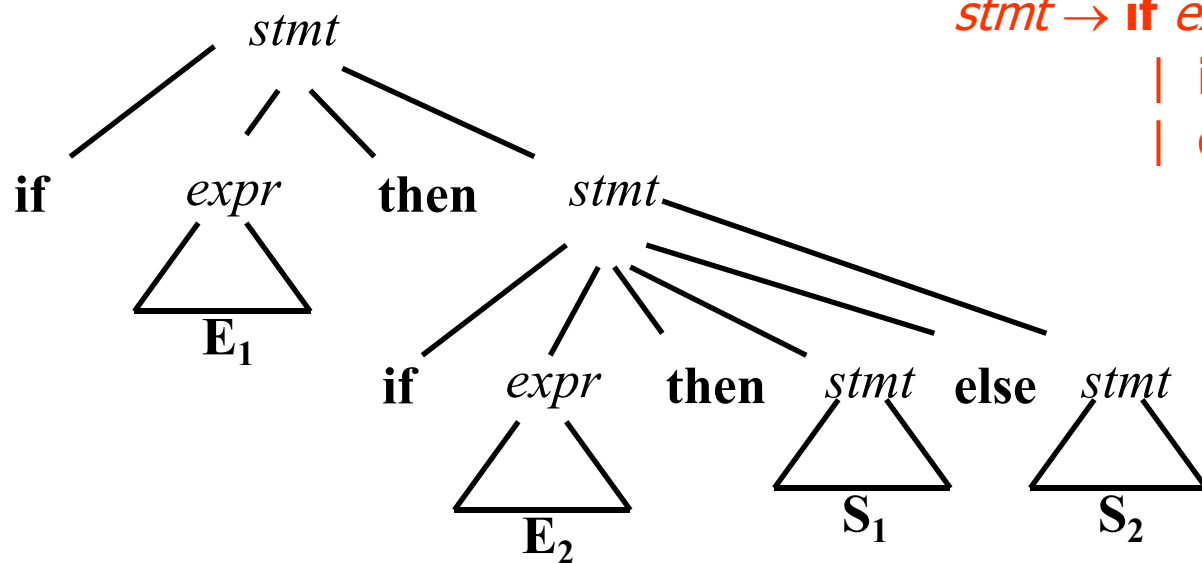
vs.

{ if E_1 then
if E_2 then
S_1
else
S_2



两个语法树

if E_1 then if E_2 then S_1 else S_2

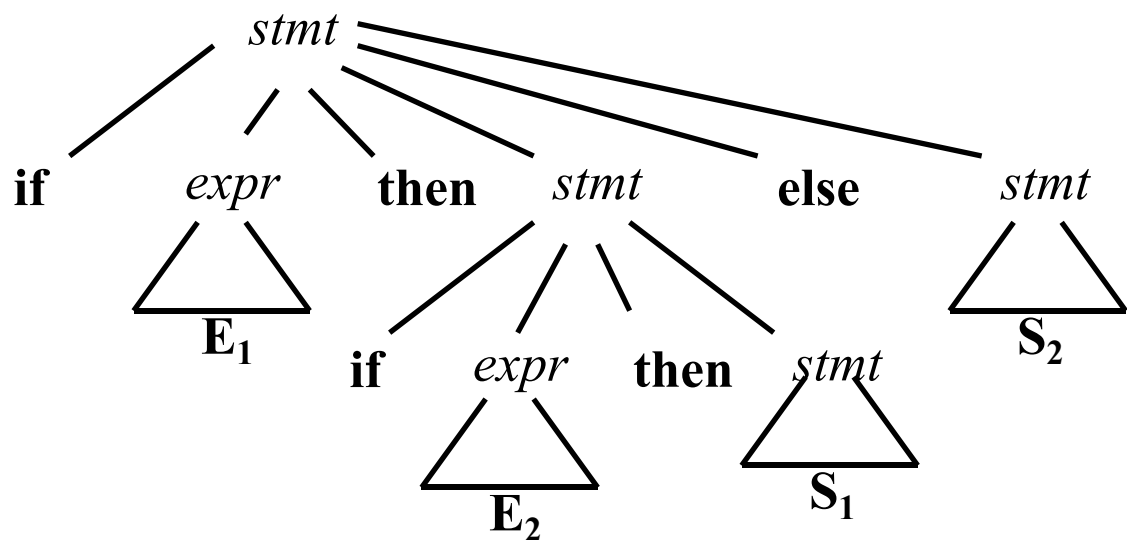


stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other** (任何其他形式的语句)



两个语法树（续）

if E_1 then if E_2 then S_1 else S_2



stmt → **if** *expr* **then** *stmt*
| **if** *expr* **then** *stmt* **else** *stmt*
| **other** (任何其他形式的语句)



消除二义性

“else与最近的未匹配的then相匹配”

修改文法—then和else间的语句必须平衡

$$\begin{aligned} stmt &\rightarrow matched_stmt \\ &| open_stmt \end{aligned}$$
$$\begin{aligned} matched_stmt &\rightarrow \mathbf{if\ expr\ then\ matched_stmt\ else\ matched_stmt} \\ &| \mathbf{other} \end{aligned}$$
$$\begin{aligned} open_stmt &\rightarrow \mathbf{if\ expr\ then\ stmt} \\ &| \mathbf{if\ expr\ then\ matched_stmt\ else\ open_stmt} \end{aligned}$$


消除左递归

$$A \stackrel{+}{\Rightarrow} A\alpha$$

自顶向下分析方法无法处理，死循环
直接左递归的消除

$$A \rightarrow A\alpha \mid \beta \quad \beta \alpha \alpha \alpha \alpha \alpha \alpha \alpha$$

β 不以A开头，改写为：

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$



例:

$$\mathbf{E} \rightarrow \mathbf{E} + \mathbf{T} \mid \mathbf{T} \longrightarrow \begin{cases} \mathbf{E} \rightarrow \mathbf{T}\mathbf{E}' \\ \mathbf{E}' \rightarrow + \mathbf{T}\mathbf{E}' \mid \varepsilon \end{cases}$$

$$\mathbf{T} \rightarrow \mathbf{T} * \mathbf{F} \mid \mathbf{F} \longrightarrow \begin{cases} \mathbf{T} \rightarrow \mathbf{F}\mathbf{T}' \\ \mathbf{T}' \rightarrow * \mathbf{F}\mathbf{T}' \mid \varepsilon \end{cases}$$

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

算法：消除间接左递归

输入：CFG G ，无环路，无 ϵ 产生式

输出：等价的、无左递归的文法

1. 非终结符按顺序排列 A_1, A_2, \dots, A_n
2. for ($i = 1; i < n; i++$)
 for ($j = 1; j < i - 1; j++$) {
 将所有形如 $A_i \rightarrow A_j \gamma$ 的产生式替换为
 $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \dots \mid \delta_k \gamma$ ，其中
 $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ 为其他对 A_j 的产生式
 }
3. 消除所有直接左递归

间接左递归

间接左递归

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Sd \mid \varepsilon \end{aligned}$$

先变换成直接左递归

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow Aad \mid bd \mid \varepsilon \end{aligned}$$

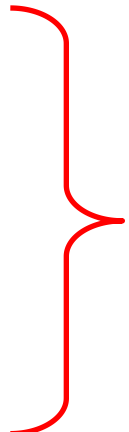
再消除左递归

$$\begin{aligned} S &\rightarrow Aa \mid b \\ A &\rightarrow bd A' \mid A' \\ A' &\rightarrow ad A' \mid \varepsilon \end{aligned}$$

消除 ε 产生式

方法：利用产生式进行代入

$$A \rightarrow \varepsilon, B \rightarrow uAv \rightarrow B \rightarrow uv \mid uAv$$

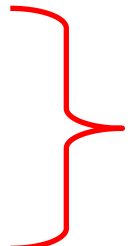
$E \rightarrow TE'$		$E \rightarrow TE' \mid T$
$E' \rightarrow + TE' \mid \varepsilon$		$E' \rightarrow + TE' \mid + T$
$T \rightarrow FT'$		$T \rightarrow FT' \mid F$
$T' \rightarrow * FT' \mid \varepsilon$		$T' \rightarrow * FT' \mid * F$
$F \rightarrow (E) \mid id$		$F \rightarrow (E) \mid id$

消除 ε 产生式

$$A_1 \rightarrow A_2 a \mid b$$

$$A_2 \rightarrow bd A_2' \mid A_2'$$


$$A_2' \rightarrow c A_2' \mid bd A_2' \mid \varepsilon$$


$$\begin{aligned} A_1 &\rightarrow A_2 a \mid b \mid a \\ A_2 &\rightarrow bd A_2' \mid A_2' \\ &\quad \mid bd \\ A_2' &\rightarrow c A_2' \mid bd A_2' \\ &\quad \mid c \mid bd \end{aligned}$$

消除回路

$$S \rightarrow SS \mid (S) \mid \varepsilon$$

回路: $S \Rightarrow SS \Rightarrow S$



$S \rightarrow \varepsilon$

如何消除回路?

保证每个产生式都加入终结符 (开始符号的 ε 产生式除外)

上面文法改写为:

$$S \rightarrow S(S) \mid (S) \mid \varepsilon$$

提取左公因子

预测分析方法要求——

向前搜索一个单词，即可确定产生式

$stmt \rightarrow \mathbf{if\ expr\ then\ stmt\ else\ stmt}$

$\quad | \mathbf{if\ expr\ then\ stmt}$ 不符合！

一般的

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$

改写为

$A \rightarrow \alpha A'$

$A' \rightarrow \beta_1 \mid \beta_2$

算法： 提取左公因子

输入：CFG G

输出：等价的、提取了左公因子的文法

方法：

对每个非终结符 A ，寻找多个候选式公共的最长前缀 α ，若 $\alpha \neq \varepsilon$ ，则将所有 A 的候选式

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$ (γ 表示所有其他候选式)，改写为

$A \rightarrow \alpha A' \mid \gamma$

$A' \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$

例:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

$i \rightarrow \text{if}$, $t \rightarrow \text{then}$, $e \rightarrow \text{else}$, $E \rightarrow \text{表达式}$, $S \rightarrow \text{语句}$

改写为:

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$E \rightarrow b$$

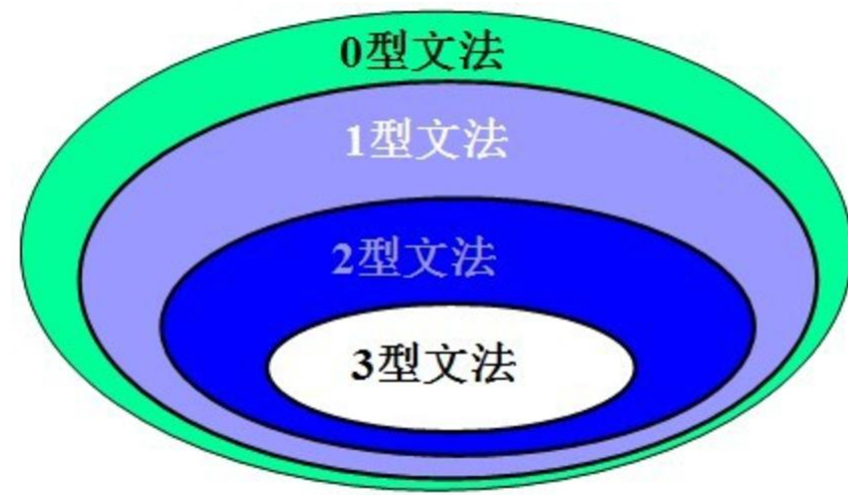
上下文无关文法

优点

- 给出精确的，易于理解的语法说明
- 自动产生高效的分析器
- 可以给语言定义出层次结构
- 以文法为基础的语言的实现便于语言的修改

问题

- 文法只能描述编程语言的大部分语法



CFG无法描述的语言结构

例1: $L_1 = \{ wcw \mid w \in (a \mid b)^* \}$

检查标识符必须在使用之前定义
语义分析

例2: $L_2 = \{ a^n b^m c^n d^m \mid n \geq 1 \text{ 且 } m \geq 1 \}$

检查函数的形参（声明）与实参（调用）的数目是否匹配
语法定义一般不考虑参数数目

CFG无法描述的语言结构

例3: $L_3 = \{ a^n b^n c^n \mid n \geq 0 \}$

排版软件，文本加下划线：n个字符，n个退格，n个下划线

另一种方式：字符—退格—下划线三元组序列， $(abc)^*$

类似语言可用CFG描述

$L_1' = \{ wcw^R \mid w \in (a \mid b)^*, w^R \text{为} w \text{的反转} \}$

$S \rightarrow aSa \mid bSb \mid c$

$L_2' = \{ a^n b^m c^m d^n \mid n \geq 1 \text{ 且 } m \geq 1 \}$

$S \rightarrow aSd \mid aAd \quad A \rightarrow bAc \mid bc$

$L_2'' = \{ a^n b^n c^m d^m \mid n \geq 1 \text{ 且 } m \geq 1 \}$

$S \rightarrow AB \quad A \rightarrow aAb \mid ab \quad B \rightarrow cBd \mid cd$

$L_3' = \{ a^n b^n \mid n \geq 0 \}$

$S \rightarrow aSb \mid ab$

学习内容

- 语法分析器概述
- 上下文无关文法
- 语法分析
 - 自顶向下分析方法：递归实现、表驱动
 - 自底向上分析方法
 - LR分析方法
 - LR(0)
 - SLR
 - LR(1)
 - LALR

语法分析器的类型

① 自顶向下分析器

② 自底向上分析器

自顶向下语法分析

- 递归下降分析, recursive-descent parsing
- LL(1), 无回溯
- 错误恢复
- 实现方法

递归下降分析方法

递归下降分析的基本方法

- 将一个非终结符 A 的文法规则看作识别 A 的过程的定义。
- A 的文法规则的右部指示过程的代码结构
 - 匹配文法规则中出现的终结符 a : $match(a)$
 - 调用文法规则中出现的非终结符

$E \rightarrow TE'\#$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

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

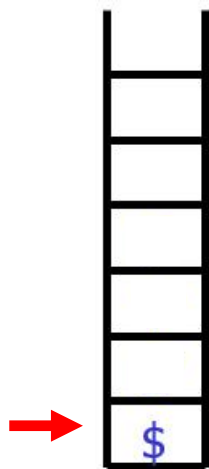
```
void T_quotation(token) {
    if (token == '*' )
    {
        token = getnext();
        F(token);
        T_quotation(token);
    }
    else
    {
        if (token != '#' || token != '+')
            ERROR();
    }
}
```

```
void F(token) {
    if (token == '(')
    {
        token = getnext();
        E(token);
        token = getnext();
        if (token != ')')
            ERROR();
    }
    else
    {
        if (token != 'id')
            ERROR();
    }
}
```

```
void T(token) {
    F(token);
    T_quotation(token);
}
```

```
void E_quotation(token) {
    if (token == '+')
    {
        token = getnext();
        T(token);
        E_quotation(token);
    }
    else
    {
        if (token != '#')
            ERROR();
    }
}
```

```
void E(token) {
    T(token);
    E_quotation(token);
}
```



\downarrow
 $id + id * id \$$

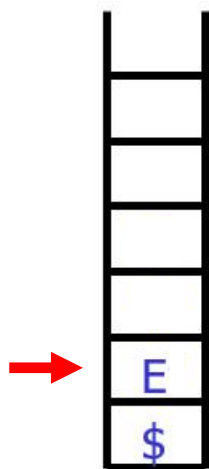
$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid id$$



\downarrow
 $id + id * id \$$

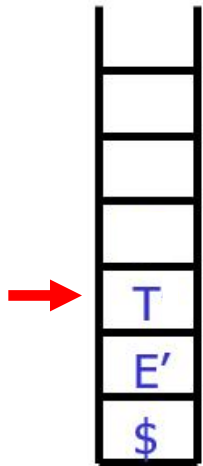
$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid id$$



\downarrow
 $id + id * id \$$

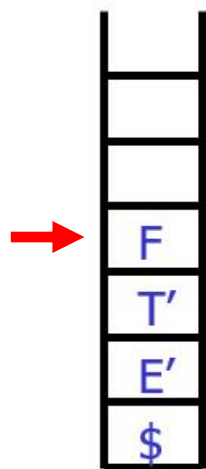
$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid id$$



\downarrow
 $id + id * id \$$

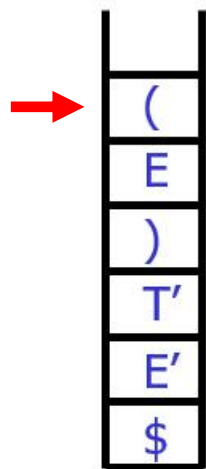
$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid id$$



\downarrow
 $id + id * id \$$

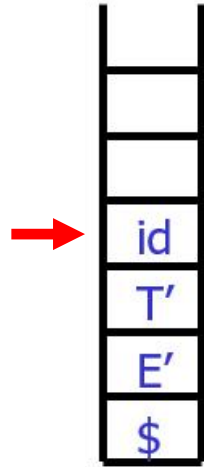
$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid id$$



\downarrow
 $id + id * id \$$

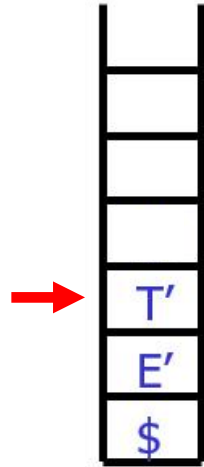
$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid id$$



$id \ + \ id \ * \ id \ \$$

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

$$F \rightarrow (E) \mid id$$

递归下降分析方法的缺点

- 不能处理左递归
- 复杂的回溯技术
 - 回溯导致语义工作推倒重来
 - 难以报告出错的确切位置
 - 效率低

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \varepsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \varepsilon \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

消除回溯

产生回溯的原因

进行推导时，若产生式存在多个候选式，选择哪个候选式进行推导存在不确定性。

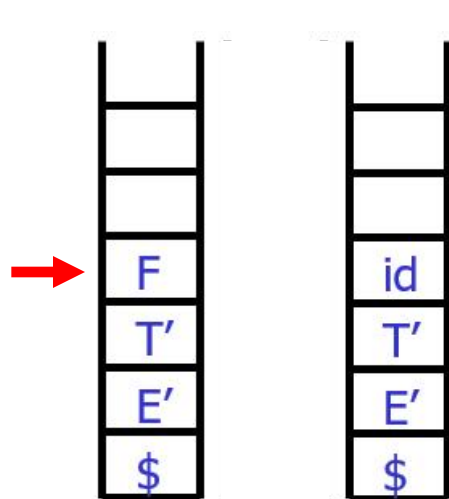
消除回溯的基本原则

对文法的任何非终结符，若能根据当前读头下的符号，准确的选择一个候选式进行推导，那么回溯就可以消除。

预测分析表

M[X,a]

非终结符	输入符号			
	id	+	*	...
E	$E \rightarrow TE'$			
E'		$E' \rightarrow +TE'$		
T	$T \rightarrow FT'$			
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	
F	$F \rightarrow \text{id}$			



$id + id * id \$$

预测分析表 $M[X, a]$

非终结符	输入符号			
	id	+	*	...
E	$E \rightarrow TE'$			
E'		$E' \rightarrow +TE'$		
T	$T \rightarrow FT'$			
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	
F	$F \rightarrow id$			

FIRST和FOLLOW

非终结符	输入符号			
	id	+	*	...
E	$E \rightarrow TE'$			
E'		$E' \rightarrow +TE'$		
T	$T \rightarrow FT'$			
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	
F	$F \rightarrow \text{id}$			

FIRST?

- $\text{FIRST}(\alpha)$: $\alpha \in (T \cup NT)^*$
 - 所有 α 可推导出的符号串的开头终结符的集合
 - $\alpha \xRightarrow{*} \epsilon \rightarrow \epsilon \in \text{FIRST}(\alpha)$

FOLLOW?

- $\text{FOLLOW}(A)$: $A \in NT$
 - 所有句型中紧接 A 之后的终结符的集合
 - $S \Rightarrow \alpha A a \beta \rightarrow a \in \text{FOLLOW}(A)$
 - $S \Rightarrow \alpha A \rightarrow \$ \in \text{FOLLOW}(A)$

$$\begin{aligned}
 E &\rightarrow TE' \\
 E' &\rightarrow +TE' \mid \varepsilon \\
 T &\rightarrow FT' \\
 T' &\rightarrow *FT' \mid \varepsilon \\
 F &\rightarrow (E) \mid id
 \end{aligned}$$

<p>Overall: $\text{First}(E) = \{ (, id \} = \text{First}(F)$</p> <p> $\text{First}(E') = \{ +, \varepsilon \} \quad \text{First}(T') = \{ *, \varepsilon \}$</p> <p> $\text{First}(T) = \text{First}(F) = \{ (, id \}$</p>

例

Given the production rules:

$$S \rightarrow i E t S S' \mid a$$

$$S' \rightarrow e S \mid \varepsilon$$

$$E \rightarrow b$$

Verify that

$$\text{First}(S) = \{ i, a \}$$

$$\text{First}(S') = \{ e, \varepsilon \}$$

$$\text{First}(E) = \{ b \}$$

				(
				E
			F)
		T	T'	T'
	E'	E'	E'	E'
\$	\$	\$	\$	\$

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

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

				(
				E
			F)
		T	T'	T'
	E'	E'	E'	E'
\$	\$	\$	\$	\$

$$E \rightarrow TE'$$

$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$

$$T' \rightarrow *FT' \mid \varepsilon$$

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

计算FOLLOW(A)——非终结符

- \$ 加入FOLLOW(S), S为开始符号, \$为输入串结束标记

计算FOLLOW(A)——非终结符

- \$ 加入FOLLOW(S), S为开始符号, \$为输入串结束标记
- 若 $A \rightarrow \alpha B \beta$,

计算FOLLOW(A)——非终结符

- \$ 加入FOLLOW(S), S为开始符号, \$为输入串结束标记
- 若 $A \rightarrow \alpha B \beta$,
则FIRST(β)中符号除 ϵ 外, 均加入FOLLOW(B)

计算FOLLOW(A)——非终结符

- \$ 加入FOLLOW(S), S为开始符号, \$为输入串结束标记
- 若 $A \rightarrow \alpha B \beta$,
则FIRST(β)中符号除 ϵ 外, 均加入FOLLOW(B)
因为: 若有 $\beta \Rightarrow a\gamma$, 显然会有 $S \Rightarrow \delta \alpha B a \gamma \eta$

计算FOLLOW(A)——非终结符

- \$ 加入FOLLOW(S), S为开始符号, \$为输入串结束标记
- 若 $A \rightarrow \alpha B \beta$,
则FIRST(β)中符号除 ϵ 外, 均加入FOLLOW(B)
因为: 若有 $\beta \Rightarrow a\gamma$, 显然会有 $S \Rightarrow \delta \alpha B a \gamma \eta$
- 若 $A \rightarrow \alpha B$ 或 $A \rightarrow \alpha B \beta$ 且 $\beta \Rightarrow \epsilon$

计算FOLLOW(A)——非终结符

- \$ 加入FOLLOW(S), S为开始符号, \$为输入串结束标记
- 若 $A \rightarrow \alpha B \beta$,
则FIRST(β)中符号除 ϵ 外, 均加入FOLLOW(B)
因为: 若有 $\beta \Rightarrow a\gamma$, 显然会有 $S \Rightarrow \delta \alpha B a \gamma \eta$
- 若 $A \rightarrow \alpha B$ 或 $A \rightarrow \alpha B \beta$ 且 $\beta \Rightarrow \epsilon$, FOLLOW(A)中所有符号加入FOLLOW(B)

计算FOLLOW(A)——非终结符

- \$ 加入FOLLOW(S), S为开始符号, \$为输入串结束标记
- 若 $A \rightarrow \alpha B \beta$,
则FIRST(β)中符号除 ϵ 外, 均加入FOLLOW(B)
因为: 若有 $\beta \Rightarrow a\gamma$, 显然会有 $S \Rightarrow \delta \alpha B a \gamma \eta$
- 若 $A \rightarrow \alpha B$ 或 $A \rightarrow \alpha B \beta$ 且 $\beta \Rightarrow \epsilon$, FOLLOW(A)中所有符号加入FOLLOW(B)
因为: 若有 $S \Rightarrow \gamma A a \eta$, 则有 $S \Rightarrow \gamma \alpha B a \eta$

二次扫描算法计算FOLLOW

1. 对所有非终结符 X , $FOLLOW(X)$ 置为空集。 $FOLLOW(S)=\{\$ \}$, S 为开始符号
2. 重复下面步骤, 直至所有FOLLOW集都不再变化
 - for 所有产生式 $X \rightarrow X_1 X_2 X_3 \dots X_m$
 - for $j = 1$ to m
 - if X_j 为非终结符, 则 $Follow(X_j)=Follow(X_j) \cup (First(X_{j+1}, \dots, X_m) - \{\epsilon\})$;
 - 若 $\epsilon \in First(X_{j+1}, \dots, X_m)$ 或 $X_{j+1}, \dots, X_m = \epsilon$, 则
 - $Follow(X_j)=Follow(X_j) \cup Follow(X)$;

例:

Compute Follow for: $E \rightarrow TE'$
 $E' \rightarrow + TE' \mid \varepsilon$
 $T \rightarrow FT'$
 $T' \rightarrow * FT' \mid \varepsilon$
 $F \rightarrow (E) \mid id$

FIRST

(id

$\varepsilon +$

(id

$\varepsilon *$

(id

FOLLOW

E \$)

E' \$)

T + \$)

T' + \$)

F + * \$)

$FOLLOW(T) = FOLLOW(T) \cup FIRST(E') \cup FOLLOW(E') \cup FOLLOW(E)$
 $FOLLOW(E') = FOLLOW(E') \cup FOLLOW(E)$
 $FOLLOW(F) = FOLLOW(F) \cup FIRST(T') \cup FOLLOW(T') \cup FOLLOW(T)$
 $FOLLOW(T') = FOLLOW(T') \cup FOLLOW(T)$
 $FOLLOW(E) = FOLLOW(E) \cup \{ \}$

	1	2	3	4	5
E	\$	\$)	\$)	\$)	\$)
E'		\$	\$)	\$)	\$)
T		+ \$	+ \$)	+ \$)	+ \$)
T'			+ \$	+ \$)	+ \$)
F		*	* + \$	* + \$)	* + \$)

例

Recall:

$$S \rightarrow i E t S S' \mid a$$

$$S' \rightarrow e S \mid \varepsilon$$

$$E \rightarrow b$$

$$\text{FIRST}(S) = \{ i, a \}$$

$$\text{FIRST}(S') = \{ e, \varepsilon \}$$

$$\text{FIRST}(E) = \{ b \}$$

$$\text{FOLLOW}(S) = \{ e, \$ \}$$

$$\text{FOLLOW}(S') = \text{FOLLOW}(S)$$

$$\text{FOLLOW}(E) = \{ t \}$$

FIRST&FOLLOW的作用

$A \rightarrow \alpha, \quad A \rightarrow \gamma$

栈

$\$ \beta A$

输入缓冲区

$\textcolor{blue}{d}y\$$

FIRST&FOLLOW的作用

$A \rightarrow \alpha, \quad A \rightarrow \gamma$

栈

输入缓冲区

$\$ \beta A$

$\mathbf{d} y \$$

- $\alpha \neq \varepsilon$ 或 $\alpha \xrightarrow{*} \varepsilon$, 且 $\mathbf{d} \in \text{FIRST}(\alpha)$

栈

输入缓冲区

$\$ \beta \mathbf{d}$

$\mathbf{d} y \$$

FIRST&FOLLOW的作用

$A \rightarrow \alpha, \quad A \rightarrow \gamma$

栈

输入缓冲区

$\$ \beta A$

$\mathbf{d} y \$$

- $\alpha \neq \varepsilon$ 或 $\alpha \xrightarrow{*} \varepsilon$, 且 $\mathbf{d} \in \text{FIRST}(\alpha)$

栈

输入缓冲区

$\$ \beta \mathbf{d}$

$\mathbf{d} y \$$

- $\alpha = \varepsilon$ 或 $\alpha \xrightarrow{*} \varepsilon$, $\mathbf{d} \in \text{FOLLOW}(\alpha)$

栈

输入缓冲区

$\$ \beta \mathbf{d}$

$\mathbf{d} y \$$

预测分析表的构造

输入：CFG G

输出：预测分析表 M

方法：

1. 对每个产生式 $A \rightarrow \alpha$ ，重复做2、3
2. 对所有的终结符 $a \in \text{FIRST}(\alpha)$ ，将 $A \rightarrow \alpha$ 加入 $M[A, a]$
3. 若 $\epsilon \in \text{FIRST}(\alpha)$ ：对所有终结符 $b \in \text{FOLLOW}(A)$ ，将 $A \rightarrow \alpha$ 加入 $M[A, b]$ ；
若 $\$ \in \text{FOLLOW}(A)$ ，将 $A \rightarrow \alpha$ 加入 $M[A, \$]$
4. 所有未定义的表项设置为错误

Non-terminal	INPUT SYMBOL					
	a	b	e	i	t	\$
S						
S'						
E						

例A:

$S \rightarrow i E t S S' \mid a$	$\text{First}(S) = \{ i, a \}$	$\text{Follow}(S) = \{ e, \$ \}$
$S' \rightarrow eS \mid \varepsilon$	$\text{First}(S') = \{ e, \varepsilon \}$	$\text{Follow}(S') = \{ e, \$ \}$
$E \rightarrow b$	$\text{First}(E) = \{ b \}$	$\text{Follow}(E) = \{ t \}$

$S \rightarrow i E t S S'$

$\text{First}(i E t S S') = \{ i \}$

$S \rightarrow a$

$\text{First}(a) = \{ a \}$

$E \rightarrow b$

$\text{First}(b) = \{ b \}$

$S' \rightarrow eS$

$\text{First}(eS) = \{ e \}$

$S' \rightarrow \varepsilon$

$\text{First}(\varepsilon) = \{ \varepsilon \}$

$\text{Follow}(S') = \{ e, \$ \}$

Non-terminal	INPUT SYMBOL					
	a	b	e	i	t	\$
S	<u>$S \rightarrow a$</u>			<u>$S \rightarrow i E t S S'$</u>		
S'			$S' \rightarrow \varepsilon$			$S' \rightarrow \varepsilon$
E		<u>$E \rightarrow b$</u>	$S' \rightarrow eS$			

LL(1)文法

定义

若文法G的预测分析表M中不含有多重定义项，则称G为 LL(1)文法。

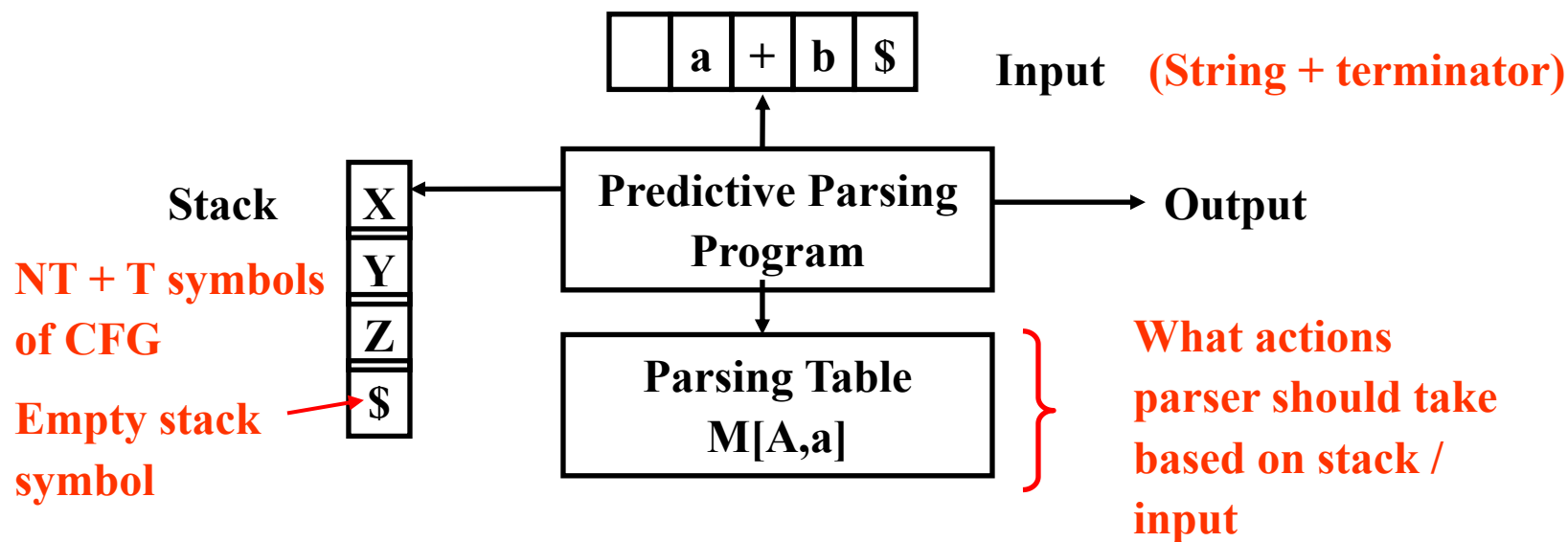
- L: 由左至右扫描输入
- L: 构造最左推导
- 1: 向前搜索一个输入符号，结合栈中符号，即可确定分析器动作

1. LL(1)文法是无二义的，二义文法一定不是LL(1)文法
2. LL的含义是从左到右扫描输入串，采用最左推导分析句子
3. 数字1表示分析句子时需向前看一个输入符号

LL(1)文法的特性

1. 无二义性，无左递归
 2. 若 $A \rightarrow \alpha \mid \beta$ ，则
 - 1) α 、 β 推导出的符号串，不能以同样的终结符 a 开头(多重定义项)
 - 2) α 、 β 至多有一个可推导出 ε
 - 3) 若 $\beta \xRightarrow{*} \varepsilon$ ， $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \Phi$
- 某些语言不存在LL(1)文法，例A

非递归预测分析方法



输入缓冲、栈、预测分析表、输出流

- 栈：语法符号序列，栈底符\$
- 预测分析表：二维数组 $M[A, a]$ ，A为NT，a为T或\$，其值为某种动作

预测分析器运行方法

考虑栈顶符号 X ，当前输入符号 a

1. $X=a=\$$ ，终止，接受输入串
2. $X=a\neq \$$ ， X 弹出栈，输入指针前移
3. X 为 NT :
 - $M[X, a] = \{X \rightarrow UVW\}$ ，将栈中 X 替换为 UVW （ U 在栈顶），输出可以是打印出产生式，表示推导
 - $M[X, a] = \mathbf{error}$ ，调用错误恢复函数

算法：非递归预测分析方法

输入：符号串 w ，文法 G 及其预测分析表 M

输出：若 $w \in L(G)$ ， w 的一个最左推导；否则，错误提示

方法：

初始：栈中为 SS （ S 在栈顶），输入缓冲区为 $w\$$ 。分析器运行算法：

设置 ip 指向输入缓冲区的第一个符号；

do {

 令 X 为栈顶符号， a 为 ip 指向符号

 if (X 为终结符或 $\$$) {

算法（续）

```
    if (X == a) {  
        X弹出栈, ip前移;  
    }  
    else error();  
} else if (M[X, a] =  $X \rightarrow Y_1 Y_2 \dots Y_k$ ) {  
    X弹出栈;  
    将 $Y_k, Y_{k-1}, \dots, Y_1$ 压栈,  $Y_1$ 置于栈顶;  
    输出产生式 $X \rightarrow Y_1 Y_2 \dots Y_k$ ;  
} else error();  
} while (X != $);
```

例:

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

Table M

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$	id + id * id\$	

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id			F→(E)		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	E → TE'

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$
\$E'T	id * id\$	

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	E → TE'
\$E'T'F	id + id * id\$	T → FT'
\$E'T'id	id + id * id\$	F → id
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	T' → ε
\$E'T+	+ id * id\$	E' → +TE'
\$E'T	id * id\$	
\$E'T'F	id * id\$	T → FT'

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	E → TE'			E → TE'		
E'		E' → +TE'			E' → ε	E' → ε
T	T → FT'			T → FT'		
T'		T' → ε	T' → *FT'		T' → ε	T' → ε
F	F → id			F → (E)		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$
\$E'T	id * id\$	
\$E'T'F	id * id\$	$T \rightarrow FT'$
\$E'T'id	id * id\$	$F \rightarrow id$

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$
\$E'T	id * id\$	
\$E'T'F	id * id\$	$T \rightarrow FT'$
\$E'T'id	id * id\$	$F \rightarrow id$
\$E'T'	* id\$	

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$
\$E'T	id * id\$	
\$E'T'F	id * id\$	$T \rightarrow FT'$
\$E'T'id	id * id\$	$F \rightarrow id$
\$E'T'	* id\$	
\$E'T'F*	* id\$	$T' \rightarrow *FT'$

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$
\$E'T	id * id\$	
\$E'T'F	id * id\$	$T \rightarrow FT'$
\$E'T'id	id * id\$	$F \rightarrow id$
\$E'T'	* id\$	
\$E'T'F*	* id\$	$T' \rightarrow *FT'$
\$E'T'F	id\$	

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$
\$E'T	id * id\$	
\$E'T'F	id * id\$	$T \rightarrow FT'$
\$E'T'id	id * id\$	$F \rightarrow id$
\$E'T'	* id\$	
\$E'T'F*	* id\$	$T' \rightarrow *FT'$
\$E'T'F	id\$	
\$E'T'id	id\$	$F \rightarrow id$

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$
\$E'T	id * id\$	
\$E'T'F	id * id\$	$T \rightarrow FT'$
\$E'T'id	id * id\$	$F \rightarrow id$
\$E'T'	* id\$	
\$E'T'F*	* id\$	$T' \rightarrow *FT'$
\$E'T'F	id\$	
\$E'T'id	id\$	$F \rightarrow id$
\$E'T'	\$	

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

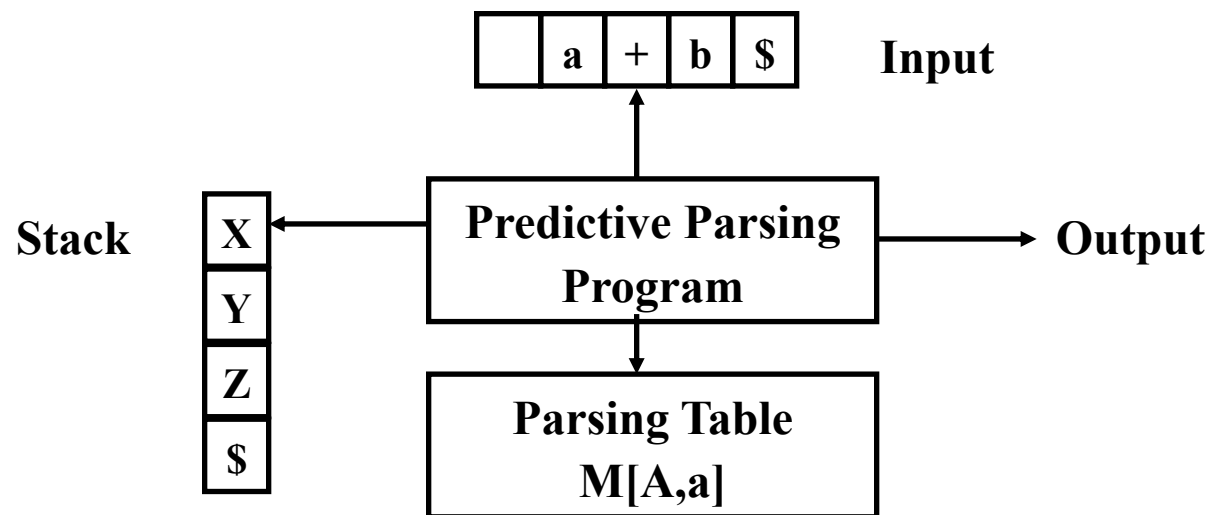
STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$
\$E'T	id * id\$	
\$E'T'F	id * id\$	$T \rightarrow FT'$
\$E'T'id	id * id\$	$F \rightarrow id$
\$E'T'	* id\$	
\$E'T'F*	* id\$	$T' \rightarrow *FT'$
\$E'T'F	id\$	
\$E'T'id	id\$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \epsilon$

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

STACK	INPUT	OUTPUT
\$E	id + id * id\$	
\$E'T	id + id * id\$	$E \rightarrow TE'$
\$E'T'F	id + id * id\$	$T \rightarrow FT'$
\$E'T'id	id + id * id\$	$F \rightarrow id$
\$E'T'	+ id * id\$	
\$E'	+ id * id\$	$T' \rightarrow \epsilon$
\$E'T+	+ id * id\$	$E' \rightarrow +TE'$
\$E'T	id * id\$	
\$E'T'F	id * id\$	$T \rightarrow FT'$
\$E'T'id	id * id\$	$F \rightarrow id$
\$E'T'	* id\$	
\$E'T'F*	* id\$	$T' \rightarrow *FT'$
\$E'T'F	id\$	
\$E'T'id	id\$	$F \rightarrow id$
\$E'T'	\$	
\$E'	\$	$T' \rightarrow \epsilon$
\$	\$	$E' \rightarrow \epsilon$

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$			$E \rightarrow TE'$		
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$			$T \rightarrow FT'$		
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$			$F \rightarrow (E)$		

预测分析法的错误恢复



何时发生错误？

1. $X \in T$, $X \neq$ 输入符号
2. $X \in NT$, $M[X, \text{输入符号}]$ 为空

两种策略：Panic模式、短语层次的恢复

恐慌模式恢复策略

考虑NT的同步单词集

1. FOLLOW(A)——略过A
2. 将高层结构的开始符号作为低层结构的同步集：语句的开始关键字——表达式的同步集，处理赋值语句漏掉分号情况
3. FIRST(A)——重新开始分析A

其他方法

4. 若 $A \xRightarrow{*} \epsilon$ ，使用它——推迟错误，减少NT
5. 若T不匹配，可弹出它，报告应插入符号——同步集设置为所有其他单词

例:

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	$E \rightarrow TE'$	_____	_____	$E \rightarrow TE'$	<u>sync</u>	<u>sync</u>
E'	_____	$E' \rightarrow +TE'$	_____	_____	$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$	<u>sync</u>	_____	$T \rightarrow FT'$	<u>sync</u>	<u>sync</u>
T'	_____	$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$	_____	$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$	<u>sync</u>	<u>sync</u>	$F \rightarrow (E)$	<u>sync</u>	<u>sync</u>

同步集——FOLLOW集

跳过输入符号

STACK	INPUT	Remark
\$E	+ id * + id\$	error, skip + id ∈ FIRST(E)
\$E	id * + id\$	
\$E'T	id * + id\$	
\$E'T'F	id * + id\$	
\$E'T'id	id * + id\$	
\$E'T'	* + id\$	error, M[F,+] = synch F has been popped
\$E'T'F*	* + id\$	
\$E'T'F	+ id\$	
\$E'T'	+ id\$	
\$E'	+ id\$	
\$E'T+	+ id\$	
\$E'T	id\$	
\$E'T'F	id\$	
\$E'T'id	id\$	
\$E'T'	\$	
\$E'	\$	
\$	\$	

Non-terminal	INPUT SYMBOL					
	id	+	*	()	\$
E	E → TE'	_____	_____	E → TE'	<u>sync</u>	<u>sync</u>
E'	_____	E' → +TE'	_____	_____	E' → ε	E' → ε
T	T → FT'	<u>sync</u>	_____	T → FT'	<u>sync</u>	<u>sync</u>
T'	_____	T' → ε	T' → *FT'	_____	T' → ε	T' → ε
F	F → id	<u>sync</u>	<u>sync</u>	F → (E)	<u>sync</u>	<u>sync</u>

产生错误信息

保存输入计数（位置）

每个非终结符符号化一个抽象语言结构

考虑例子的文法

- E表示表达式
 - E在栈顶，输入符号为+：“错误位置i，表达式不能以+开始”或“错误位置i，非法表达式”
 - E，*的情况类似
- E'表示表达式的结束
 - E'，*/id：“错误：位置j开始的表达式在位置i处结构错误”

产生错误信息（续）

- T表示加法项
 - T, *: “错误位置i, 非法项”
- T'表示项的结束
 - T', (: “位置j开始的项在位置i处结构错误”
- F表示加法/乘法项

同步错误

- F, +: “位置i缺少加法/乘法项”
- E,): “位置i缺少表达式”

产生错误信息（续）

栈顶终结符与输入符号不匹配

- **id, +:** “位置i缺少标识符”
- **)**, 其他符号
 - 分析过程中遇到 ‘(’, 都将位置保存在“左括号栈”中——实际可用符号栈实现
 - 当发现不匹配时, 查找左括号栈, 恢复括号位置
 - “错误位置i: 位置m处左括号无对应右括号”
——如 **(id * + (id id)\$**

短语层次错误恢复

预测分析表空位填入错误处理函数

- 修改栈和（或）输入流，插入、删除、替换
- 输出错误信息

问题

- 插入、替换栈符号应小心，避免错误推导
- 避免无限循环

与Panic模式结合使用，更完整的方式

自顶向下分析——预测分析法

预测分析法实现步骤

1. 构造文法
2. 改造文法：消除二义性、消除左递归、消除回溯
3. 求每个变量的 $FIRST$ 集和 $FOLLOW$ 集，构造预测分析表
4. 检查是不是 $LL(1)$ 文法
5. 对于递归的预测分析，为每一个非终结符编写一个过程；对于非递归的预测分析，实现表驱动的预测分析算法

缺点：不是所有文法满足 $LL(1)$ 要求

自底向上语法分析

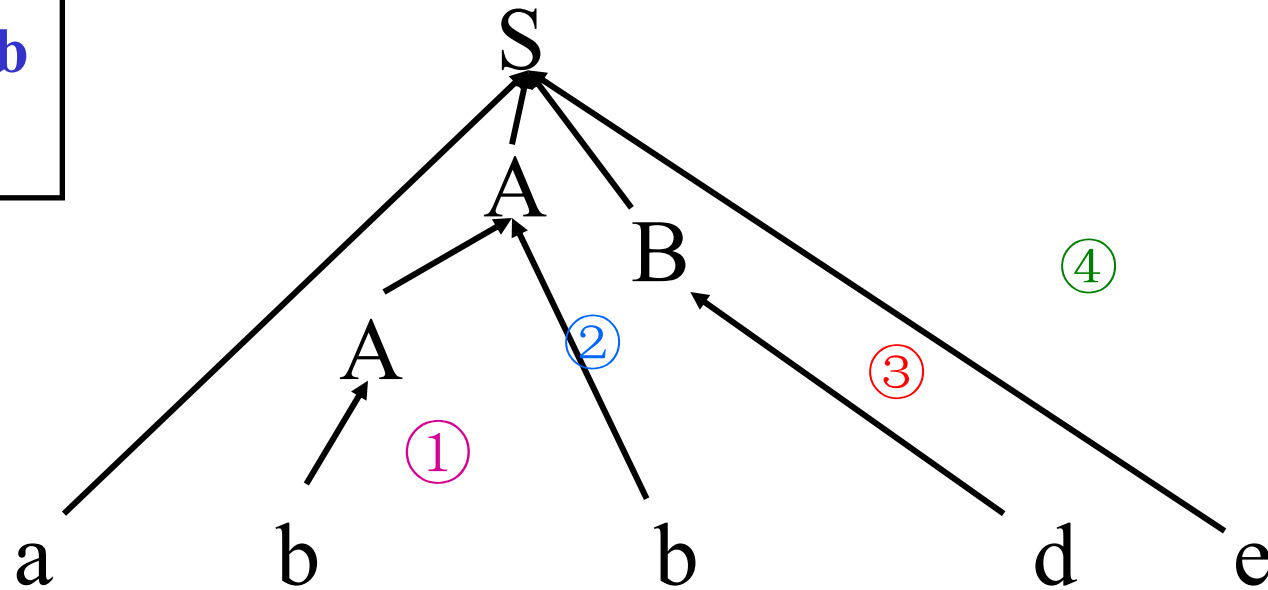
- 自底向上语法分析，是从输入符号串出发，试图把它归约成识别符号。
- 从图形上看，自底向上分析过程是以输入符号串作为末端结点符号串，向着根结点方向往上构造语法树，使识别符号正是该语法树的根结点。

某产生式体相匹配的特定子串被替换为该产生式头部的非终结符号

归约

例：句子abbde的归约过程

$S \rightarrow aABe$
 $A \rightarrow Ab \mid b$
 $B \rightarrow d$



abbcde
aAbde
aAde
aABe
S

对应最右推导 $S \xRightarrow{rm} aABe \xRightarrow{rm} aAde \xRightarrow{rm} aAbde \xRightarrow{rm} abbde$

自底向上语法分析

- 自底向上分析是一个不断进行直接归约的过程。任何自底向上分析方法的关键是要找出这种句柄。

句柄 (Handle)

符号串的“句柄”

- 与某个产生式右部匹配的子串
- 归约为产生式左部 $\leftarrow \rightarrow$ 最右推导逆过程

形式化定义：一个最右句型 γ 的句柄

- $\langle A \rightarrow \beta, \gamma \text{ 中 } \beta \text{ 的位置} \rangle$
- $S \xRightarrow{*}_{rm} \alpha A w \xRightarrow{rm} \alpha \beta w$, γ 中 α 之后即为句柄位置, w 只包含终结符

$$\begin{aligned} S &\rightarrow \underline{aABe} \\ A &\rightarrow Ab \mid b \\ \underline{B} &\rightarrow d \end{aligned}$$

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbde \Rightarrow_{rm} abbde$$

移入—归约语法分析技术

两个问题

- 定位句柄
- 确定用哪个产生式进行归约

一般实现方法——栈

1. 将输入符号“移进”栈，直至在栈顶形成一个句柄
2. 将句柄“归约”为相应的非终结符
3. 不断重复，直至栈中只剩开始符号，输入缓冲区为空——接受输入串
4. 错误处理

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id, * id, \$$	归约 $E \rightarrow id$

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id_2 * id_3 \$$	归约 $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	移进

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id_2 * id_3 \$$	归约 $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	移进
\$E +	$id_2 * id_3 \$$	移进

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id_2 * id_3 \$$	归约 $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	移进
\$E +	$id_2 * id_3 \$$	移进
\$E + id_2	$* id_3 \$$	归约 $E \rightarrow id$

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id_2 * id_3 \$$	归约 $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	移进
\$E +	$id_2 * id_3 \$$	移进
\$E + id_2	$* id_3 \$$	归约 $E \rightarrow id$
\$E + E	$* id_3 \$$	移进

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id_2 * id_3 \$$	归约 $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	移进
\$E +	$id_2 * id_3 \$$	移进
\$E + id_2	$* id_3 \$$	归约 $E \rightarrow id$
\$E + E	$* id_3 \$$	移进
\$E + E *	$id_3 \$$	移进

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id_2 * id_3 \$$	归约 $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	移进
\$E +	$id_2 * id_3 \$$	移进
\$E + id_2	$* id_3 \$$	归约 $E \rightarrow id$
\$E + E	$* id_3 \$$	移进
\$E + E *	$id_3 \$$	移进
\$E + E * id_3	\$	归约 $E \rightarrow id$

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id_2 * id_3 \$$	归约 $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	移进
\$E +	$id_2 * id_3 \$$	移进
\$E + id_2	$* id_3 \$$	归约 $E \rightarrow id$
\$E + E	$* id_3 \$$	移进
\$E + E *	$id_3 \$$	移进
\$E + E * id_3	\$	归约 $E \rightarrow id$
\$E + E * E	\$	归约 $E \rightarrow E * E$

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id_2 * id_3 \$$	归约 $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	移进
\$E +	$id_2 * id_3 \$$	移进
\$E + id_2	$* id_3 \$$	归约 $E \rightarrow id$
\$E + E	$* id_3 \$$	移进
\$E + E *	$id_3 \$$	移进
\$E + E * id_3	\$	归约 $E \rightarrow id$
\$E + E * E	\$	归约 $E \rightarrow E * E$
\$E + E	\$	归约 $E \rightarrow E + E$

$$E \rightarrow E + E \mid E * E \mid (E) \mid id$$

栈	输入	动作
\$	$id_1 + id_2 * id_3 \$$	移进
\$ id_1	$+ id_2 * id_3 \$$	归约 $E \rightarrow id$
\$E	$+ id_2 * id_3 \$$	移进
\$E +	$id_2 * id_3 \$$	移进
\$E + id_2	$* id_3 \$$	归约 $E \rightarrow id$
\$E + E	$* id_3 \$$	移进
\$E + E *	$id_3 \$$	移进
\$E + E * id_3	\$	归约 $E \rightarrow id$
\$E + E * E	\$	归约 $E \rightarrow E * E$
\$E + E	\$	归约 $E \rightarrow E + E$
\$E	\$	接受

基本操作

1. 移进 (**shift**)
 - 下个输入符号移到栈顶
2. 归约 (**reduce**)
 - 句柄的右端恰在栈顶，定位句柄左端
 - 确定选用的产生式，用产生式左部非终结符替换栈中的句柄
3. 接受 (**accept**)
 - ❑ 宣布分析成功结束
4. 错误 (**error**)
 - ❑ 发现语法错误，调用错误恢复函数

LR分析方法

- LR分析方法：当前最广义的无回溯的“移进-归约”方法。
- LR分析方法：“自左到右扫描和最左归约”的自底向上的分析方法。
- 根据栈中的符号串和向右顺序查看输入串中的 $k(k \geq 0)$ 个符号，就能唯一确定分析器的动作是移进还是归约，以及用哪个产生式进行归约。
- 优点：适用范围广；分析速度快；报错准确。
- 构造分析器的工作量很大，不大可能手工构造

LR(k)分析技术

L----是指从左至右扫描输入符号串

R----是指构造一个最右推导的逆过程

k----是指为了作出分析决定而向前看的输入符号的个数。若 $k=0$ ，就为LR(0)分析，说明分析动作时，不向前看任何符号，LR(1)分析，说明分析动作时只向前看一个符号。

- LR(0) ， SLR， 规范LR， LALR

LR分析器的组成

从逻辑上说，一个LR分析器包括两部分：一个总控程序和一张分析表。

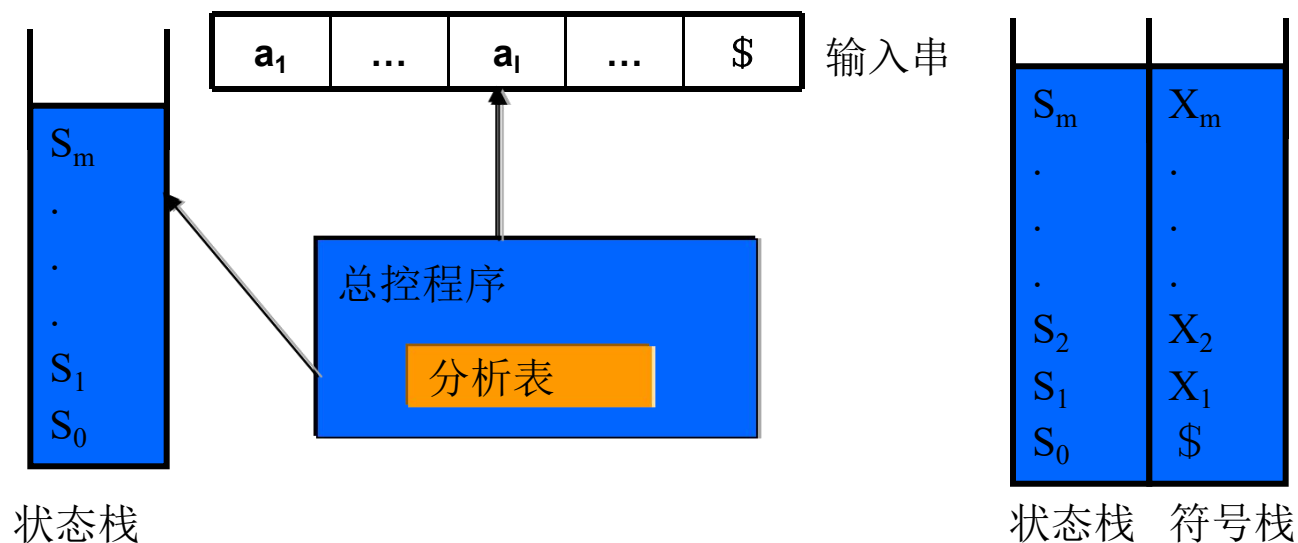
一般说来，所有LR分析器总控程序是一样的，只是分析表各不相同。

LR分析器工作原理

LR分析器的逻辑结构

在逻辑上，一个LR分析器结构如下图所示。它是由一个输入符号串，一个下推状态栈，以及一个总控程序和分析表组成。

实际上在分析时读入符号是不进栈的。为使分析解释更清楚，我们另设一个符号栈（实际上只有一个状态栈用于存放状态）。



LR分析表（LR分析器核心）

LR分析表：分析动作表+状态转换表

- 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（动作）						GOTO（状态转换）		
	id	+	*	()	\$	E	T	F
0	S ₅			S ₄			1	2	3
1		S ₆				acc			
2		r ₂	S ₇		r ₂	r ₂			
3		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			
6	S ₅			S ₄				9	3
7	S ₅			S ₄					10
8		S ₆			S ₁₁				
9		r ₁	S ₇		r ₁	r ₁			
10		r ₃	r ₃		r ₃	r ₃			
11		r ₅	r ₅		r ₅	r ₅			

LR(0)分析表的构造

LR(0)分析就是LR(k)分析当 $k=0$ 的情况，就是指在分析每一步，只要根据当前栈顶状态，就能确定应采取何种分析动作，而无需向前查看输入符号。为了构造LR分析表，首先引入可行前缀的概念。

(1) 可行前缀

前缀：是指字符的任意首部。如字abc的前缀有 ϵ , a, ab, abc.

可行前缀：规范句型（右句型）的一个前缀，如果它不含句柄后任何符号，则称它是该规范句型的一个可行前缀。也就是说在可行前缀右边增添一些终结符号之后，就可以成为规范句型。如： $S \Rightarrow abcdef$, 其中cd是句柄，则

$\epsilon, a, ab, abc, abcd$ 是该规范句型的可行前缀，而abcd是包含句柄的可行前缀。

在LR分析过程中的任何时候，栈里的文法符号 $X_1X_2...X_m$ 应该构成可行前缀，把输入串的剩余部分配上之后即成为规范句型（如果整个输入串确实构成一个句子的话。）

LR(0)分析表的构造

可行前缀与句柄之间的关系:

- ① 可行前缀已包含句柄全部符号, 这表明产生式 $A \rightarrow \beta$ 的右部符号串 β 已在分析栈顶, 因此相应的分析动作应是用此产生式进行归约, 称可归约的可行前缀。我们用 $A \rightarrow \beta \cdot$ 表示
- ② 可行前缀中只含句柄一部分符号, 意味着形如产生式 $A \rightarrow \beta_1 \beta_2$ 的右部子串 β_1 已出现在栈顶, 正期待着从余留输入串中看到能由 β_2 推出的符号串。
我们用 $A \rightarrow \beta_1 \cdot \beta_2$ 表示
- ③ 可行前缀不包含句柄的任何符号, 这表明产生式 $A \rightarrow \beta$ 的右部符号串 β 不在分析栈顶, 正期待从余留输入串中由产生式 $A \rightarrow \beta$ 的 β 所能推出的符号串。
我们用 $A \rightarrow \cdot \beta$ 表示

LR(0)分析表的构造

(2) LR(0) 项目

我们把右部某位置上标有圆点的产生式称为相应文法的一个LR(0)项目。特别地，对形如 $A \rightarrow \varepsilon$ 的产生式，相应LR(0)项目为 $A \rightarrow \cdot$ 。

例如：

$A \rightarrow \beta$	$A \rightarrow \beta \cdot$	一个LR(0)项目
$A \rightarrow \beta$	$A \rightarrow \cdot \beta$	一个LR(0)项目
$A \rightarrow \beta_1 \beta_2$	$A \rightarrow \beta_1 \cdot \beta_2$	一个LR(0)项目
$A \rightarrow \varepsilon$	$A \rightarrow \cdot$	一个LR(0)项目

LR(0)分析表的构造

(2) LR(0) 项目

产生式 $A \rightarrow aBC$ ，根据圆点的位置不同可以有四个LR(0)项目：

$A \rightarrow \cdot aBC$ 正期待着从余留输入串中由 aBC 推出的符号串进栈

$A \rightarrow a \cdot BC$ a 已进栈，正期待着从余留输入串中由 BC 推出的符号串进栈

$A \rightarrow aB \cdot C$ aB 推出的符号串进栈,正期待着从余留输入串中 C 推出的符号串进栈

$A \rightarrow aBC \cdot$ aBC 推出的符号串进栈

对于产生式 $A \rightarrow \beta$ 对应项目数为 $|\beta| + 1$ 个。（ $|\beta|$ 表示 β 所含字符的个数）显然，不同的LR(0)项目反映了分析过程中栈顶的不同情况。

后继项目：两个项目对应相同的产生式，圆点位置只差一个符号。例： $A \rightarrow a \cdot BC$ 称为 $A \rightarrow \cdot aBC$ 的后继项目

LR(0)分析表的构造

(3) 构造识别可行前缀的有穷自动机

1) 将一般文法G改写成增广文法G'

如果S是文法G的开始符号，则增广文法G'中增加一个产生式 $S' \rightarrow S$ ， S' 是文法G'开始符号，显然 $L(G)=L(G')$ ，这样就使得增广文法G'中有项目 $S' \rightarrow S \cdot$ 是唯一接受项目
例如上面我们举的例子中的增广文法G'为：

① $S' \rightarrow E$

② $E \rightarrow aA$

③ $E \rightarrow bB$

④ $A \rightarrow cA$

⑤ $A \rightarrow d$

⑥ $B \rightarrow cB$

⑦ $B \rightarrow d$

LR(0)分析表的构造

2) 写出增广文法G'LR(0)的全部项目

对于文法G'，其LR(0)项目有：

- | | | |
|-------------------------------|--------------------------------|--------------------------------|
| (1) $S' \rightarrow \cdot E$ | (7) $A \rightarrow c \cdot A$ | (13) $E \rightarrow bB \cdot$ |
| (2) $S' \rightarrow E \cdot$ | (8) $A \rightarrow cA \cdot$ | (14) $B \rightarrow \cdot cB$ |
| (3) $E \rightarrow \cdot aA$ | (9) $A \rightarrow \cdot d$ | (15) $B \rightarrow c \cdot B$ |
| (4) $E \rightarrow a \cdot A$ | (10) $A \rightarrow d \cdot$ | (16) $B \rightarrow cB \cdot$ |
| (5) $E \rightarrow aA \cdot$ | (11) $E \rightarrow \cdot bB$ | (17) $B \rightarrow \cdot d$ |
| (6) $A \rightarrow \cdot cA$ | (12) $E \rightarrow b \cdot B$ | (18) $B \rightarrow d \cdot$ |

- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{aA}$
- 2) $E \rightarrow \underline{bB}$
- 3) $A \rightarrow \underline{cA}$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{Cb}$
- 6) $B \rightarrow d$

LR(0)分析表的构造

3) 构造DFA

① 先求出DFA初态 I_0 的状态集

I_0 的状态集由基本项目 $J = S' \rightarrow \cdot E$ 开始求出

即 $I_0 = \text{CLOSURE}(J) = \text{CLOSURE}(\{S' \rightarrow \cdot E\})$

按构造 I 的闭包 $\text{CLOSURE}(I)$ 的方法,可求得

$$I_0 = \{S' \rightarrow \cdot E, E \rightarrow \cdot aA, E \rightarrow \cdot bB\}$$

0) $S' \rightarrow E$

1) $E \rightarrow \underline{a}A$

2) $E \rightarrow \underline{b}B$

3) $A \rightarrow \underline{c}A$

4) $A \rightarrow d$

5) $B \rightarrow \underline{C}b$

6) $B \rightarrow d$

LR(0)分析表的构造

(1) $S' \rightarrow \cdot E$	(7) $A \rightarrow \underline{c} \cdot A$	(13) $E \rightarrow \underline{b} B \cdot$
(2) $S' \rightarrow E \cdot$	(8) $A \rightarrow c \underline{A} \cdot$	(14) $B \rightarrow \cdot \underline{c} B$
(3) $E \rightarrow \cdot \underline{a} A$	(9) $A \rightarrow \cdot d$	(15) $B \rightarrow \underline{c} \cdot B$
(4) $E \rightarrow \underline{a} \cdot A$	(10) $A \rightarrow d \cdot$	(16) $B \rightarrow c \underline{B} \cdot$
(5) $E \rightarrow \underline{a} A \cdot$	(11) $E \rightarrow \cdot \underline{b} B$	(17) $B \rightarrow \cdot d$
(6) $A \rightarrow \cdot \underline{c} A$	(12) $E \rightarrow \underline{b} \cdot B$	(18) $B \rightarrow d \cdot$

② 由初态 I_0 构造其他状态 $I_1, I_2, I_3, \dots, I_{11}$

$I_0 = \{S' \rightarrow \cdot E, E \rightarrow \cdot \underline{a} A, E \rightarrow \cdot \underline{b} B\}$ $I_1 = GO(I_0, E) = CLOSURE(\{S' \rightarrow E \cdot\}) = \{S' \rightarrow E \cdot\}$

$I_2 = GO(I_0, a) = CLOSURE(\{E \rightarrow a \cdot A\}) = \{E \rightarrow a \cdot A, A \rightarrow \cdot c A, A \rightarrow \cdot d\}$

$I_3 = GO(I_0, b) = CLOSURE(\{E \rightarrow b \cdot B\}) = \{E \rightarrow b \cdot B, B \rightarrow \cdot c B, B \rightarrow \cdot d\}$

$I_4 = GO(I_2, c) = CLOSURE(\{A \rightarrow c \cdot A\}) = \{A \rightarrow c \cdot A, A \rightarrow \cdot c A, A \rightarrow \cdot d\}$

$I_5 = GO(I_3, c) = CLOSURE(\{B \rightarrow c \cdot B\}) = \{B \rightarrow c \cdot B, B \rightarrow \cdot c B, B \rightarrow \cdot d\}$

$I_6 = GO(I_2, A) = CLOSURE(\{E \rightarrow a A \cdot\}) = \{E \rightarrow a A \cdot\}$

$I_7 = GO(I_3, B) = CLOSURE(\{E \rightarrow b B \cdot\}) = \{E \rightarrow b B \cdot\}$

$I_8 = GO(I_4, A) = CLOSURE(\{A \rightarrow c A \cdot\}) = \{A \rightarrow c A \cdot\}$

$I_9 = GO(I_5, B) = CLOSURE(\{B \rightarrow c B \cdot\}) = \{B \rightarrow c B \cdot\}$

$I_{10} = GO(I_4, d) = CLOSURE(\{A \rightarrow d \cdot\}) = \{A \rightarrow d \cdot\}$

$I_{11} = GO(I_3, d) = CLOSURE(\{B \rightarrow d \cdot\}) = \{B \rightarrow d \cdot\}$

此外, 由于 $GO(I_4, c) = I_4$, $GO(I_2, d) = I_{10}$

$GO(I_5, c) = I_5$, $GO(I_5, d) = I_{11}$ 故它们不产生新项目集。

实际上, 我们可以直接画图求出 I_2, I_3, \dots, I_{11} , 这样可直接画出DFA图, 十分方便。

I_0 :
 $S' \rightarrow \bullet E$

0) $S' \rightarrow E$

1) $E \rightarrow \underline{aA}$

2) $E \rightarrow \underline{bB}$

3) $A \rightarrow \underline{cA}$

4) $A \rightarrow d$

5) $B \rightarrow \underline{Cb}$

6) $B \rightarrow d$

I_0 :

$S' \rightarrow \bullet E$

$E \rightarrow \bullet aA$

$E \rightarrow \bullet bB$

0) $S' \rightarrow E$

1) $E \rightarrow \underline{aA}$

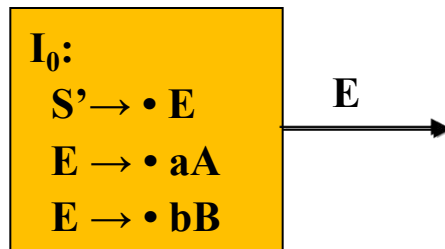
2) $E \rightarrow \underline{bB}$

3) $A \rightarrow \underline{cA}$

4) $A \rightarrow d$

5) $B \rightarrow \underline{Cb}$

6) $B \rightarrow d$



0) $S' \rightarrow E$

1) $E \rightarrow \underline{aA}$

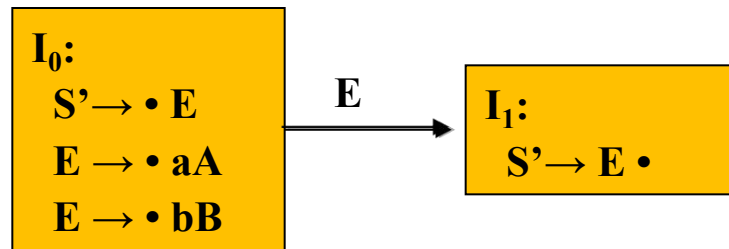
2) $E \rightarrow \underline{bB}$

3) $A \rightarrow \underline{cA}$

4) $A \rightarrow d$

5) $B \rightarrow \underline{Cb}$

6) $B \rightarrow d$



0) $S' \rightarrow E$

1) $E \rightarrow \underline{aA}$

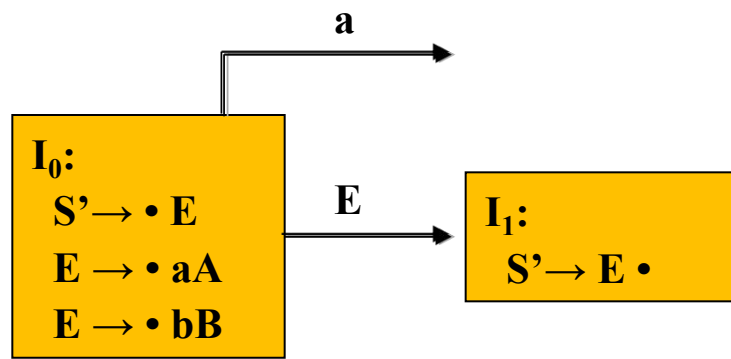
2) $E \rightarrow \underline{bB}$

3) $A \rightarrow \underline{cA}$

4) $A \rightarrow d$

5) $B \rightarrow \underline{Cb}$

6) $B \rightarrow d$



0) $S' \rightarrow E$

1) $E \rightarrow \underline{aA}$

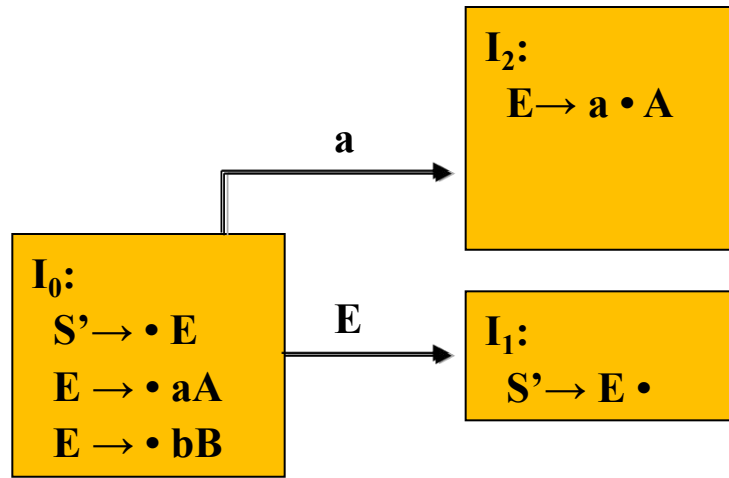
2) $E \rightarrow \underline{bB}$

3) $A \rightarrow \underline{cA}$

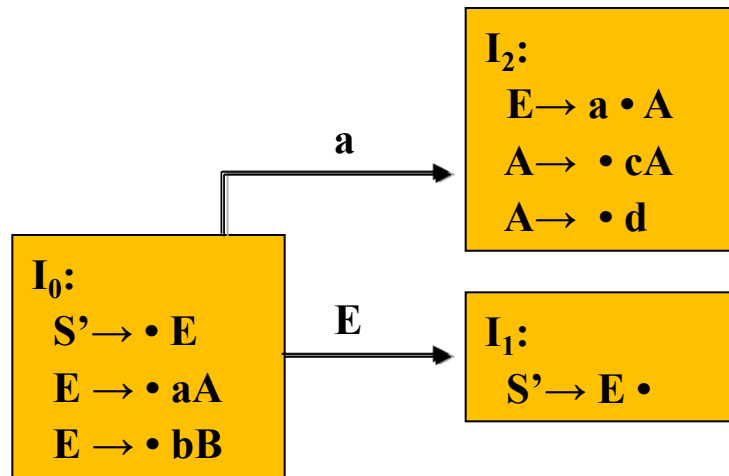
4) $A \rightarrow d$

5) $B \rightarrow \underline{Cb}$

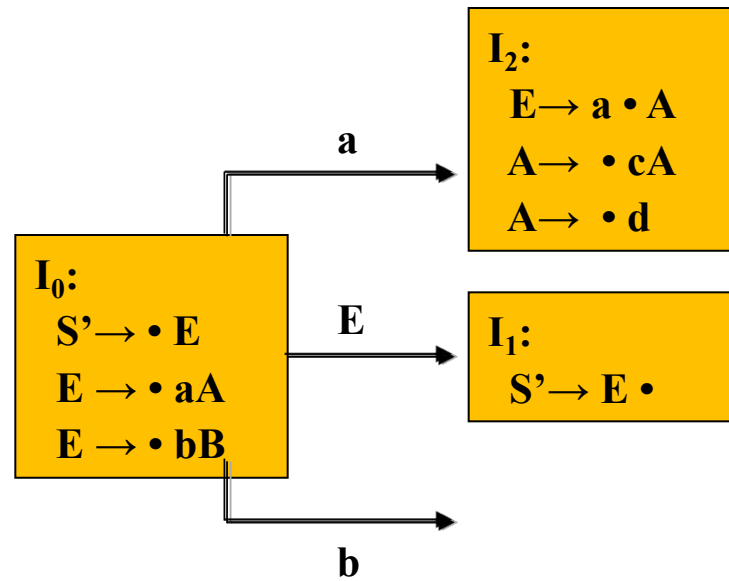
6) $B \rightarrow d$



- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



0) $S' \rightarrow E$

1) $E \rightarrow \underline{aA}$

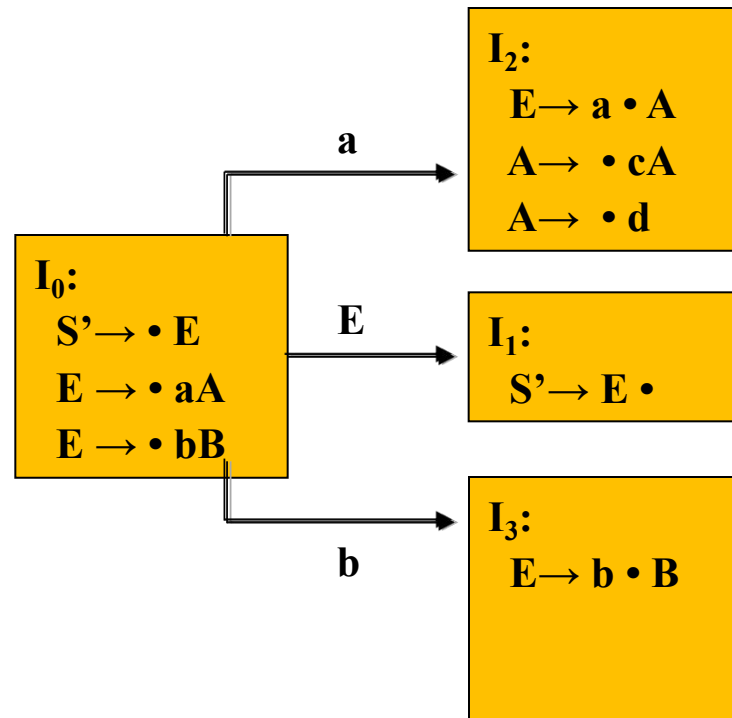
2) $E \rightarrow \underline{bB}$

3) $A \rightarrow \underline{cA}$

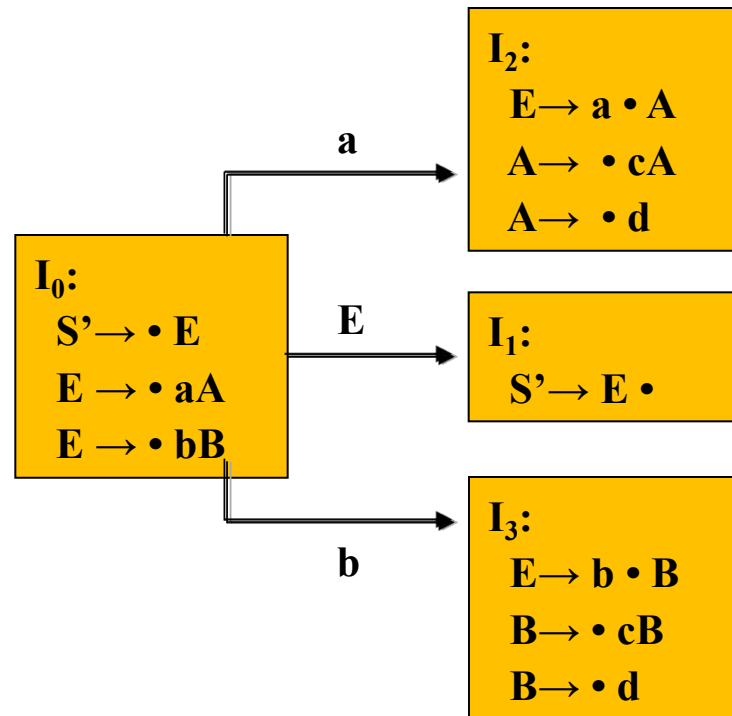
4) $A \rightarrow d$

5) $B \rightarrow \underline{Cb}$

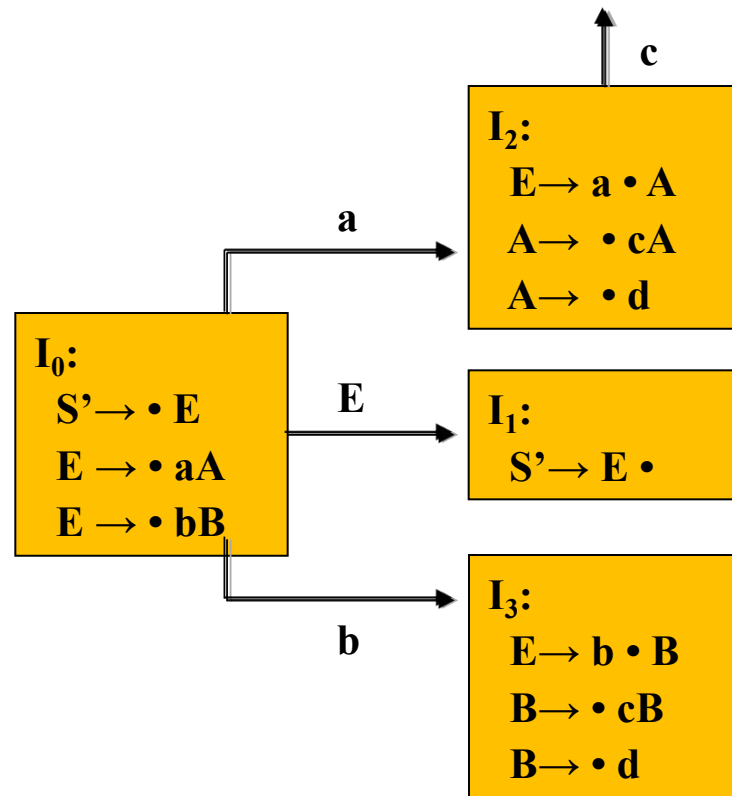
6) $B \rightarrow d$



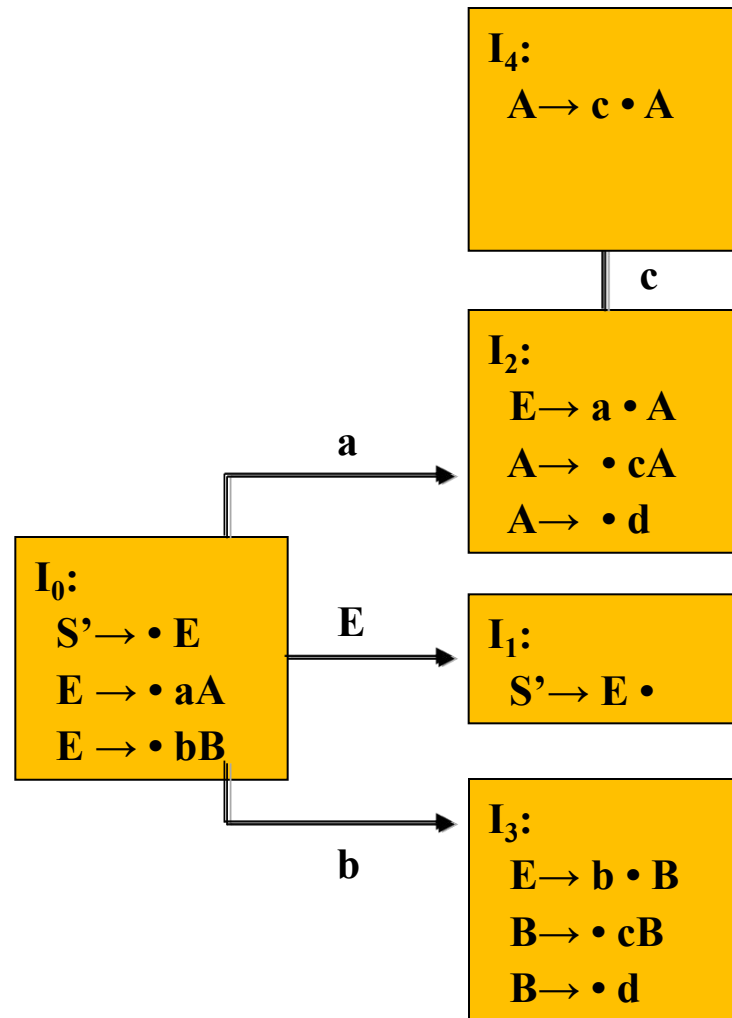
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{aA}$
- 2) $E \rightarrow \underline{bB}$
- 3) $A \rightarrow \underline{cA}$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{Cb}$
- 6) $B \rightarrow d$



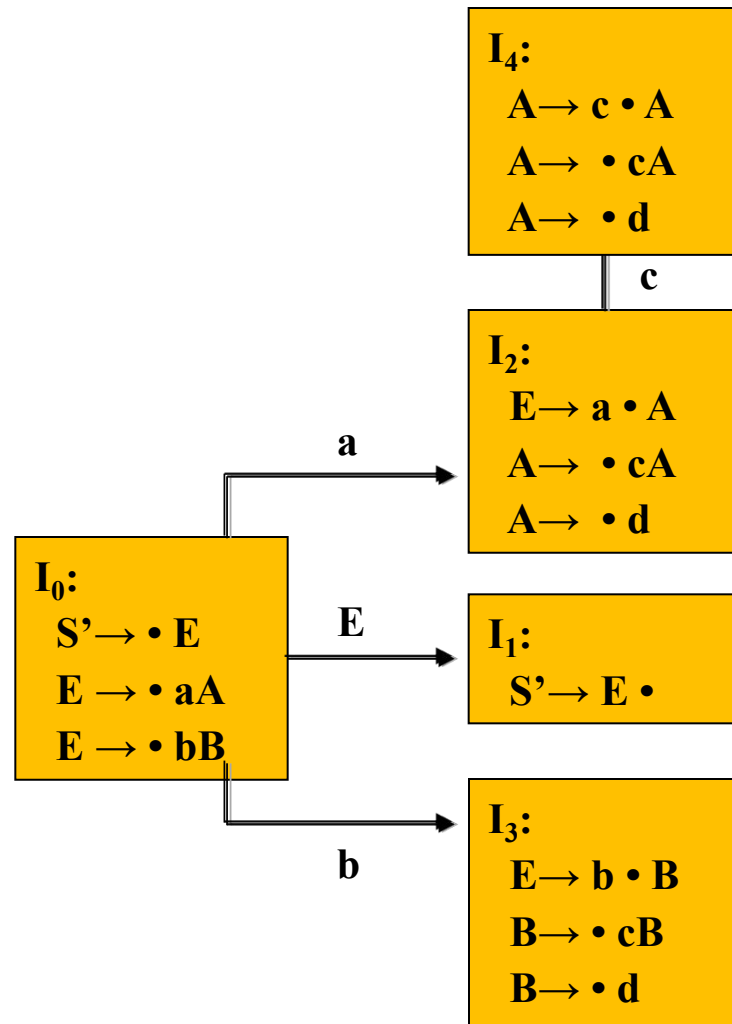
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{aA}$
- 2) $E \rightarrow \underline{bB}$
- 3) $A \rightarrow \underline{cA}$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{Cb}$
- 6) $B \rightarrow d$



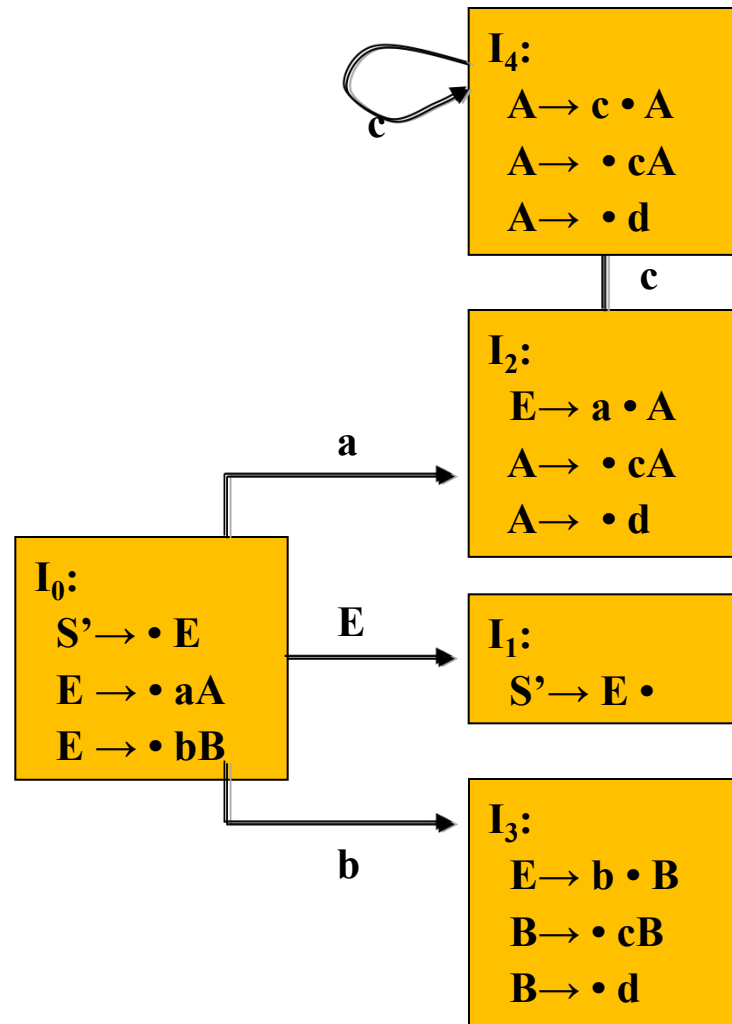
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{aA}$
- 2) $E \rightarrow \underline{bB}$
- 3) $A \rightarrow \underline{cA}$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{Cb}$
- 6) $B \rightarrow d$



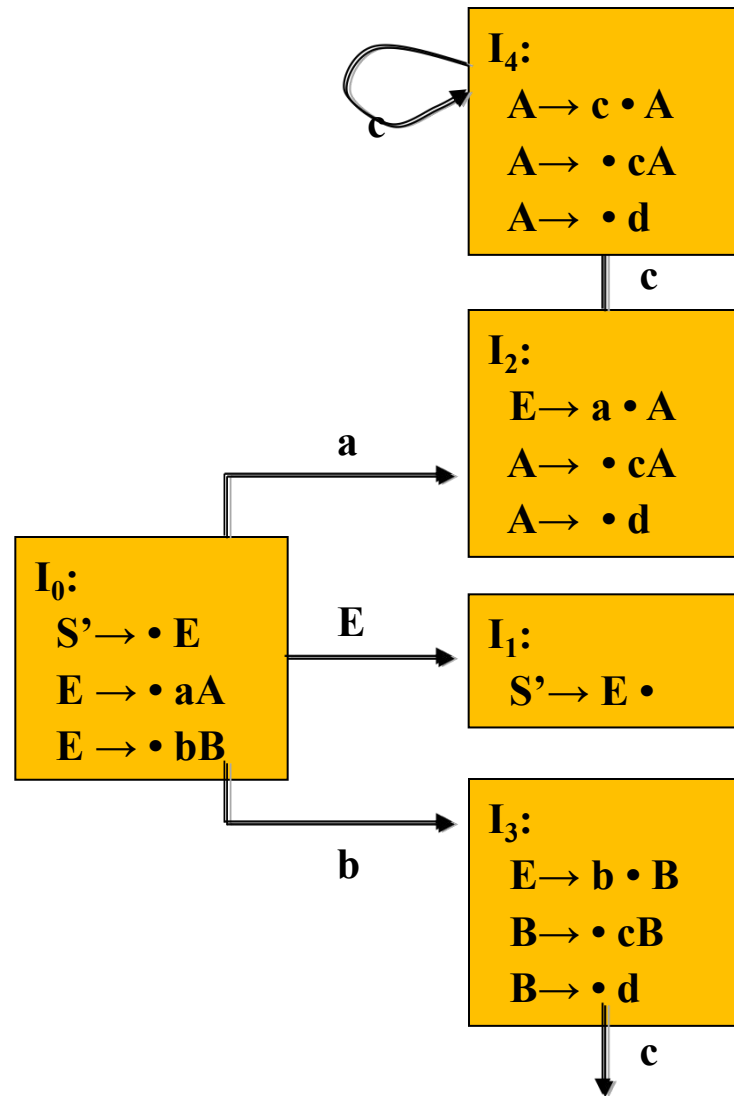
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



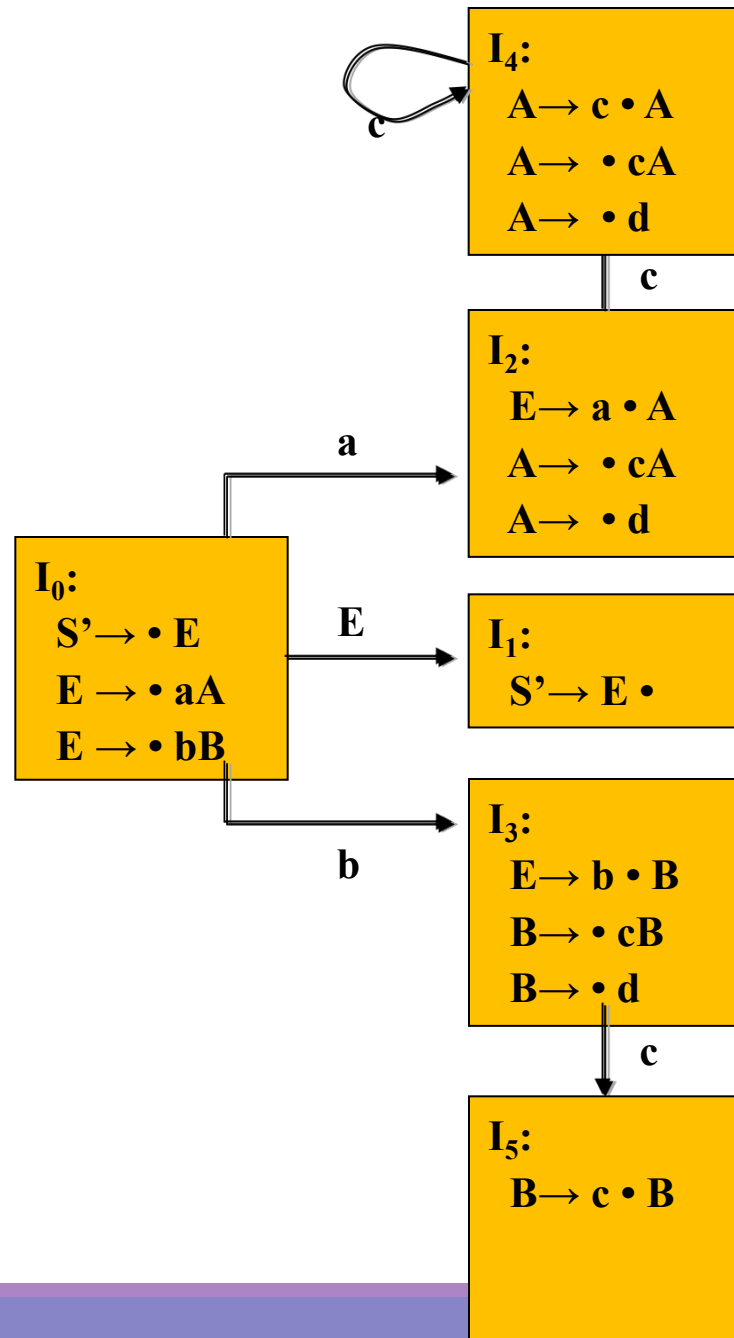
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{aA}$
- 2) $E \rightarrow \underline{bB}$
- 3) $A \rightarrow \underline{cA}$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{Cb}$
- 6) $B \rightarrow d$



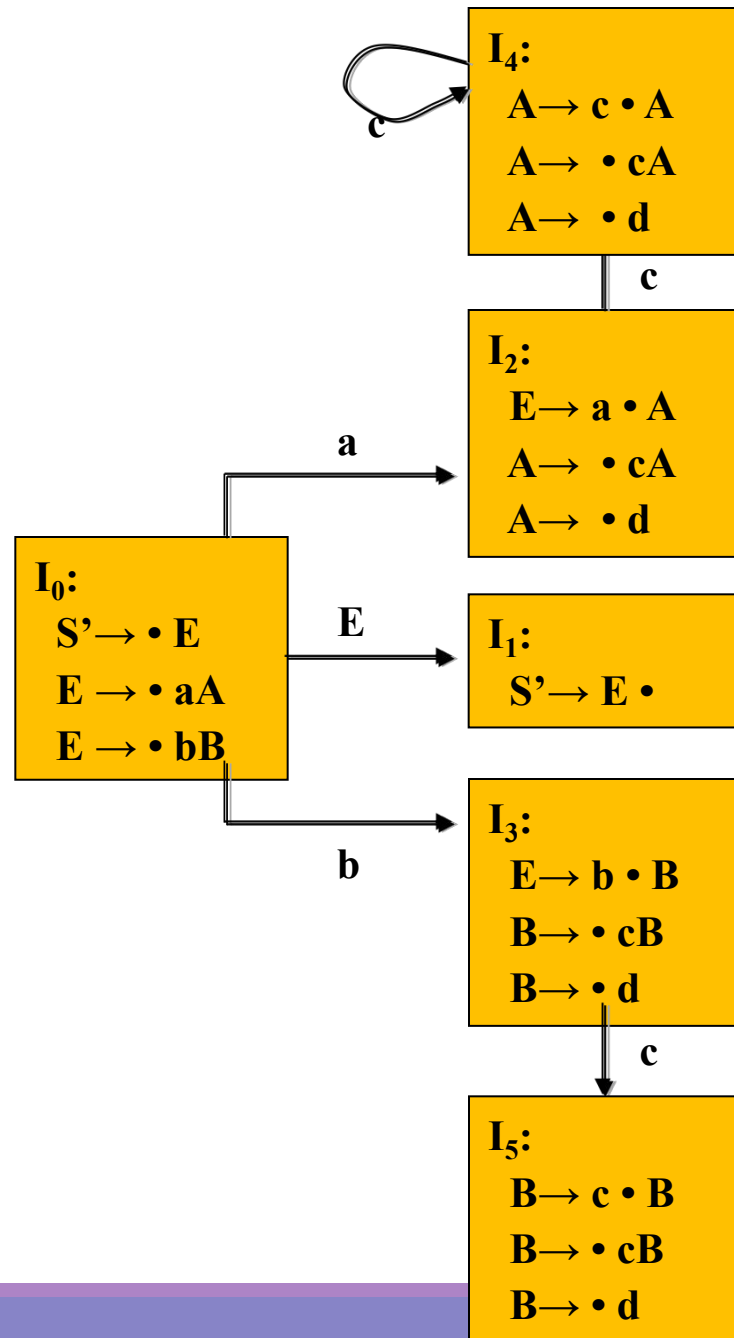
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



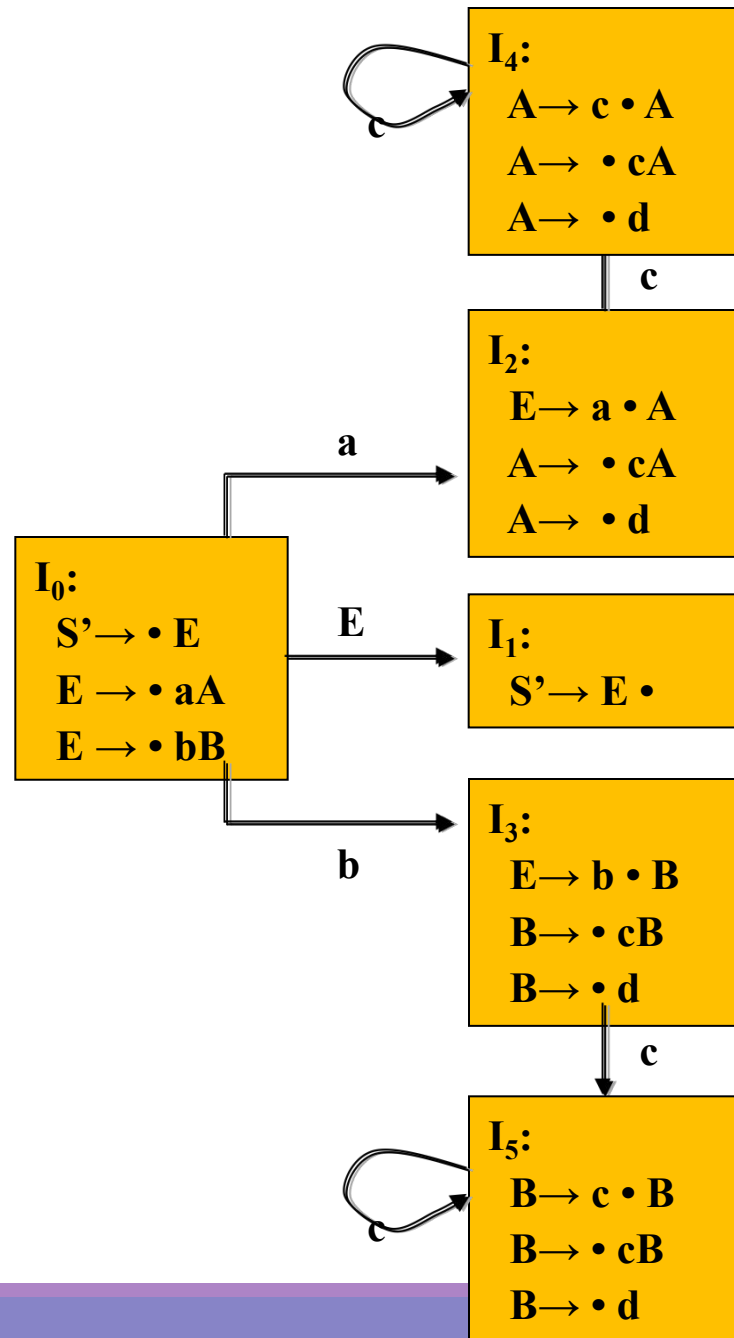
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



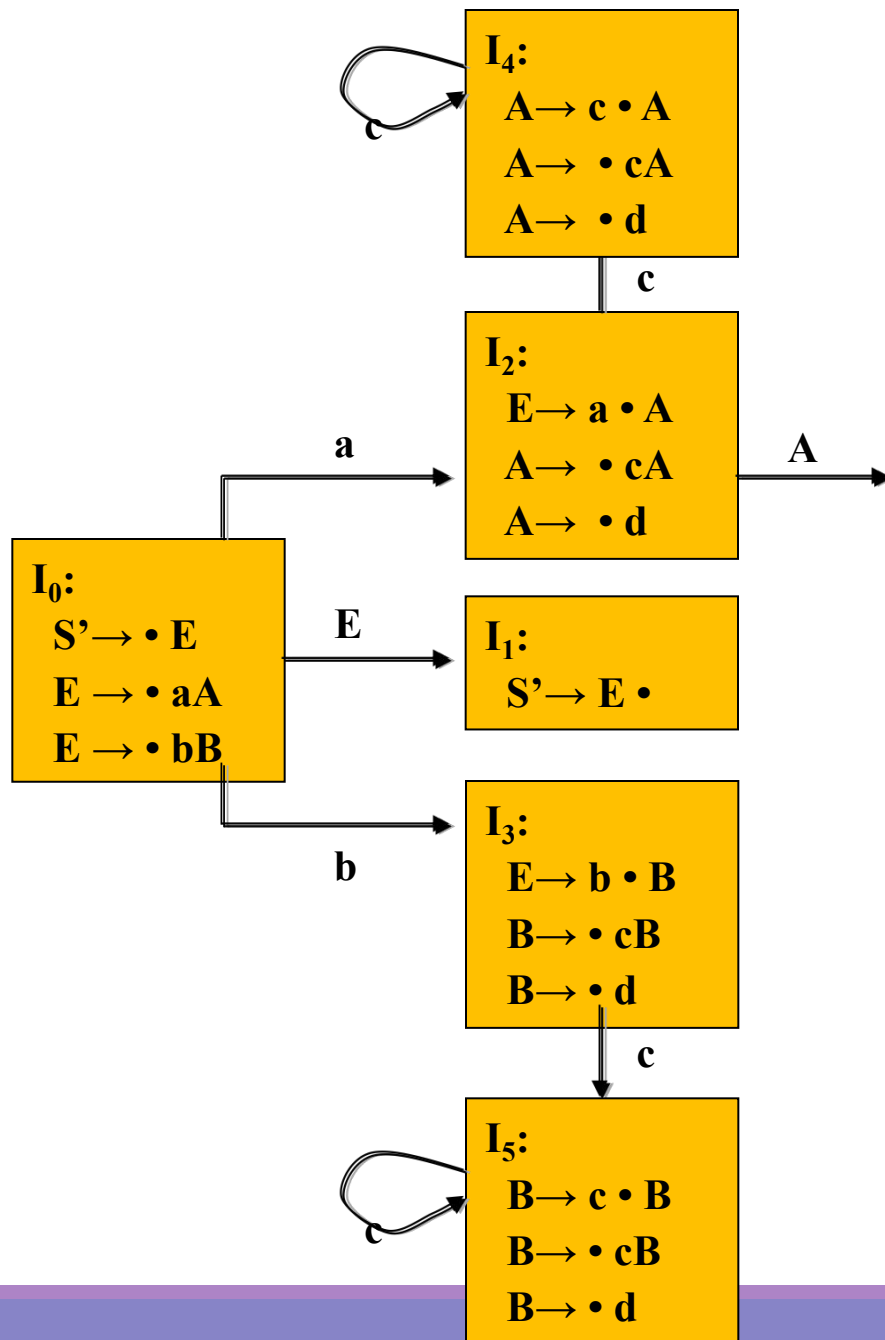
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



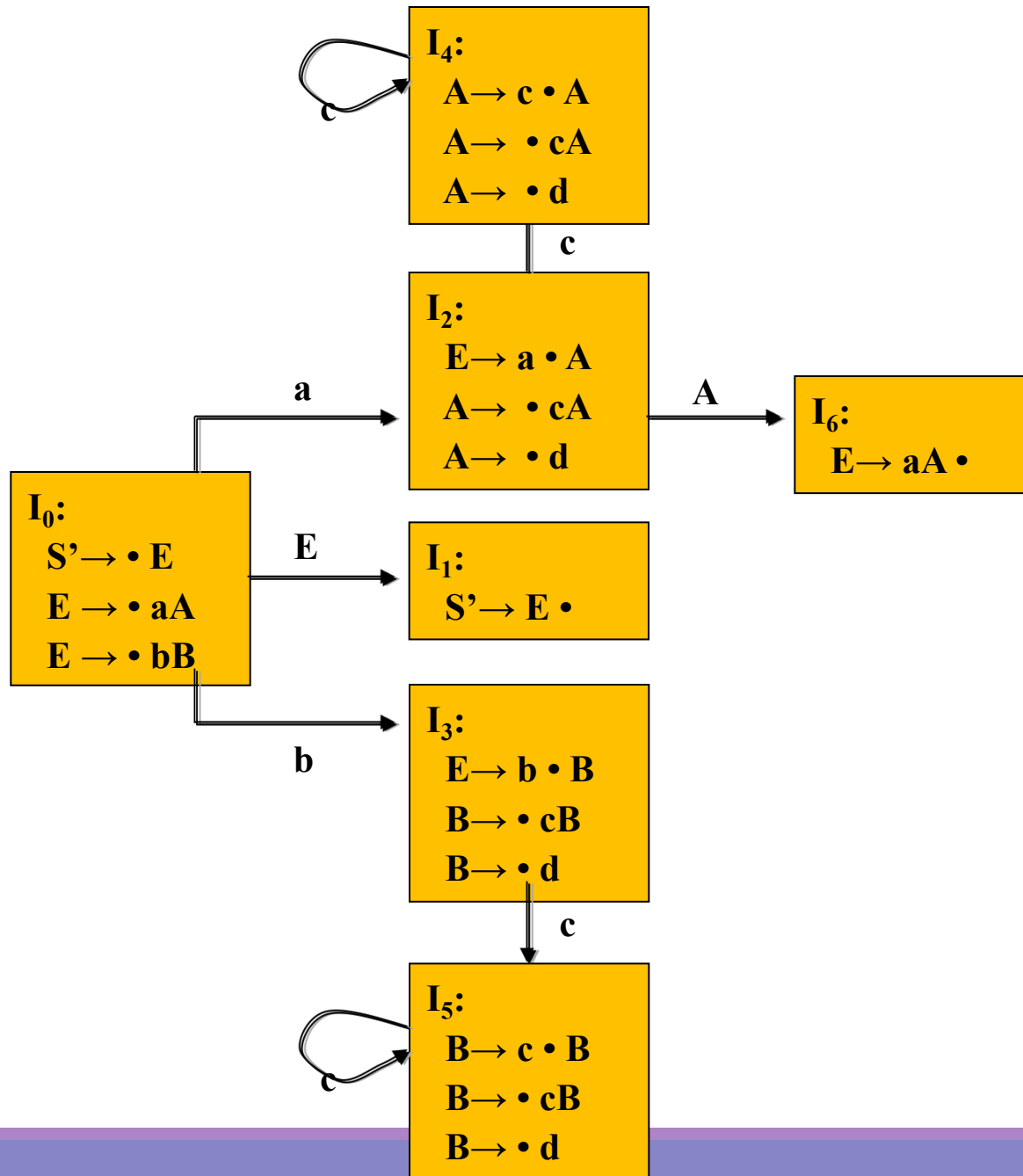
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



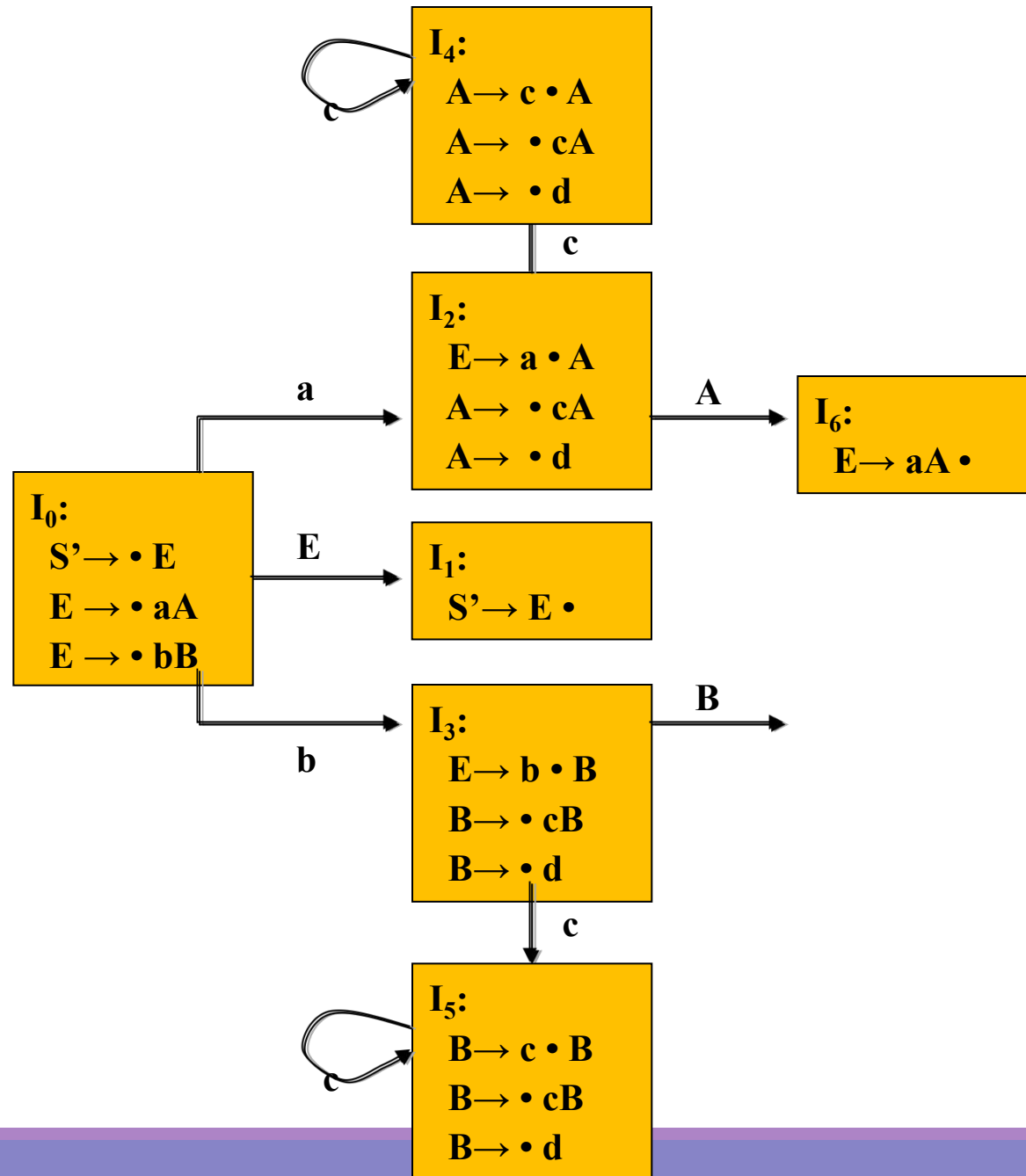
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



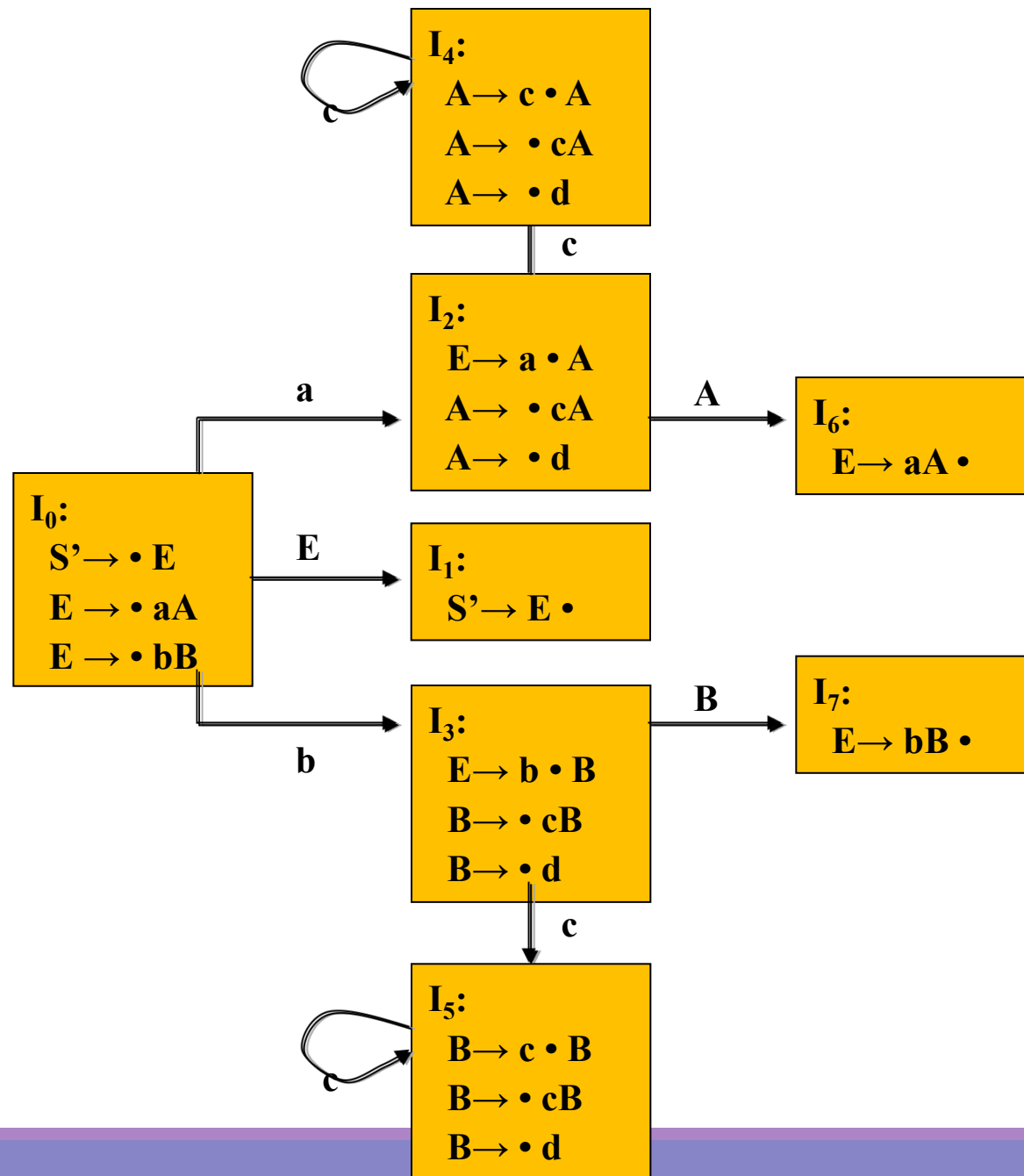
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



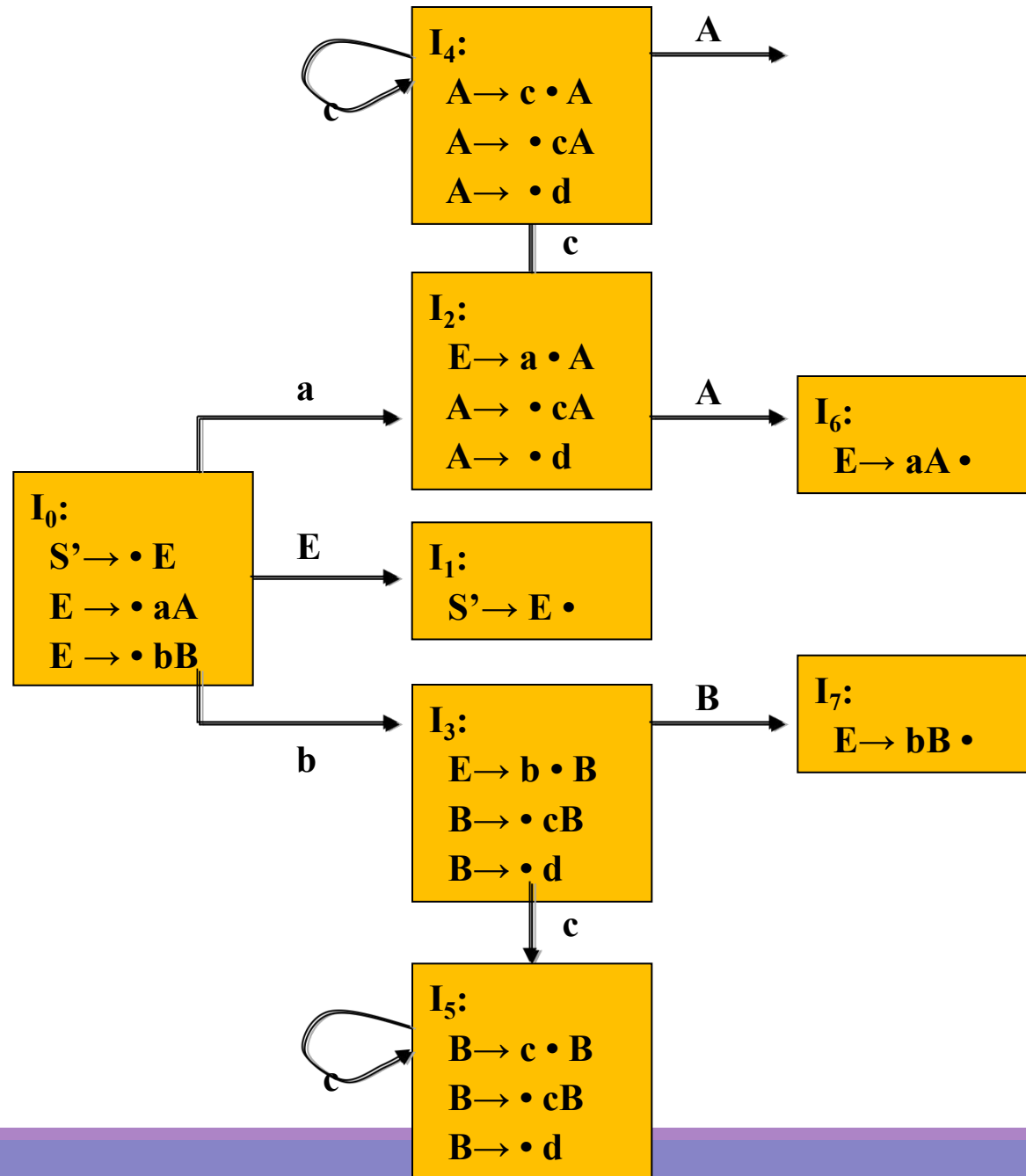
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



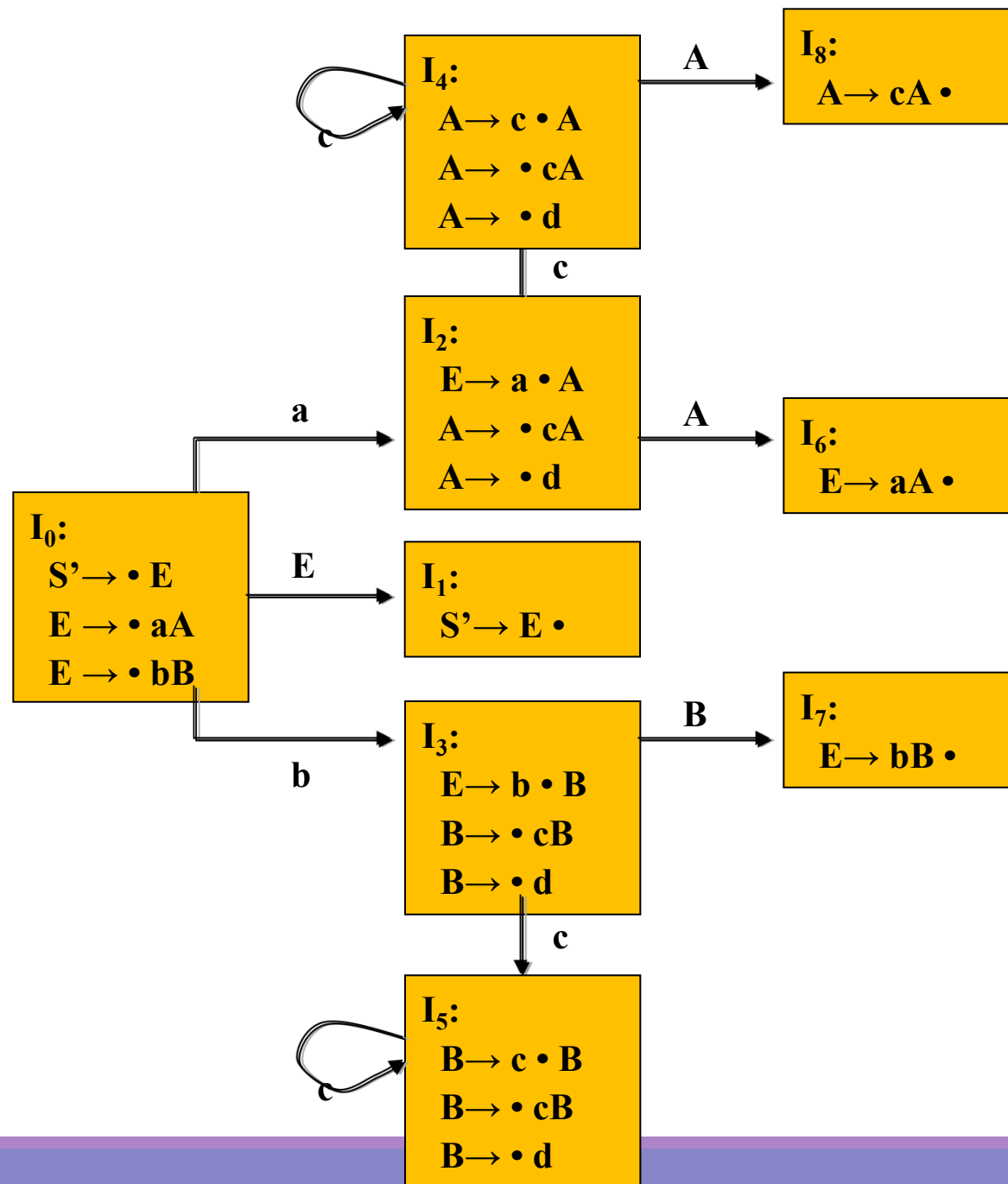
- 0) $S' \rightarrow E$
- 1) $E \rightarrow aA$
- 2) $E \rightarrow bB$
- 3) $A \rightarrow cA$
- 4) $A \rightarrow d$
- 5) $B \rightarrow Cb$
- 6) $B \rightarrow d$



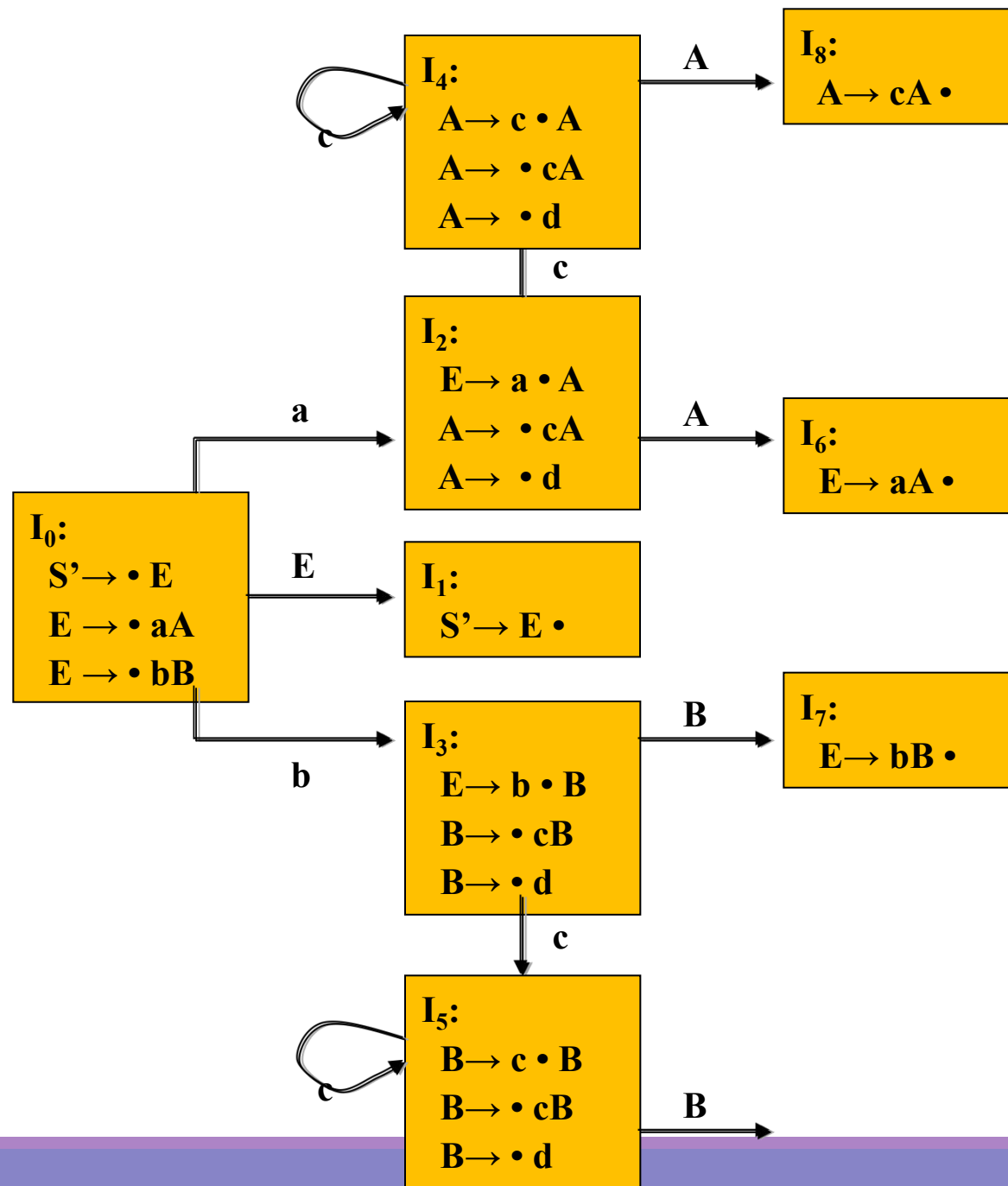
- 0) $S' \rightarrow E$
- 1) $E \rightarrow aA$
- 2) $E \rightarrow bB$
- 3) $A \rightarrow cA$
- 4) $A \rightarrow d$
- 5) $B \rightarrow Cb$
- 6) $B \rightarrow d$



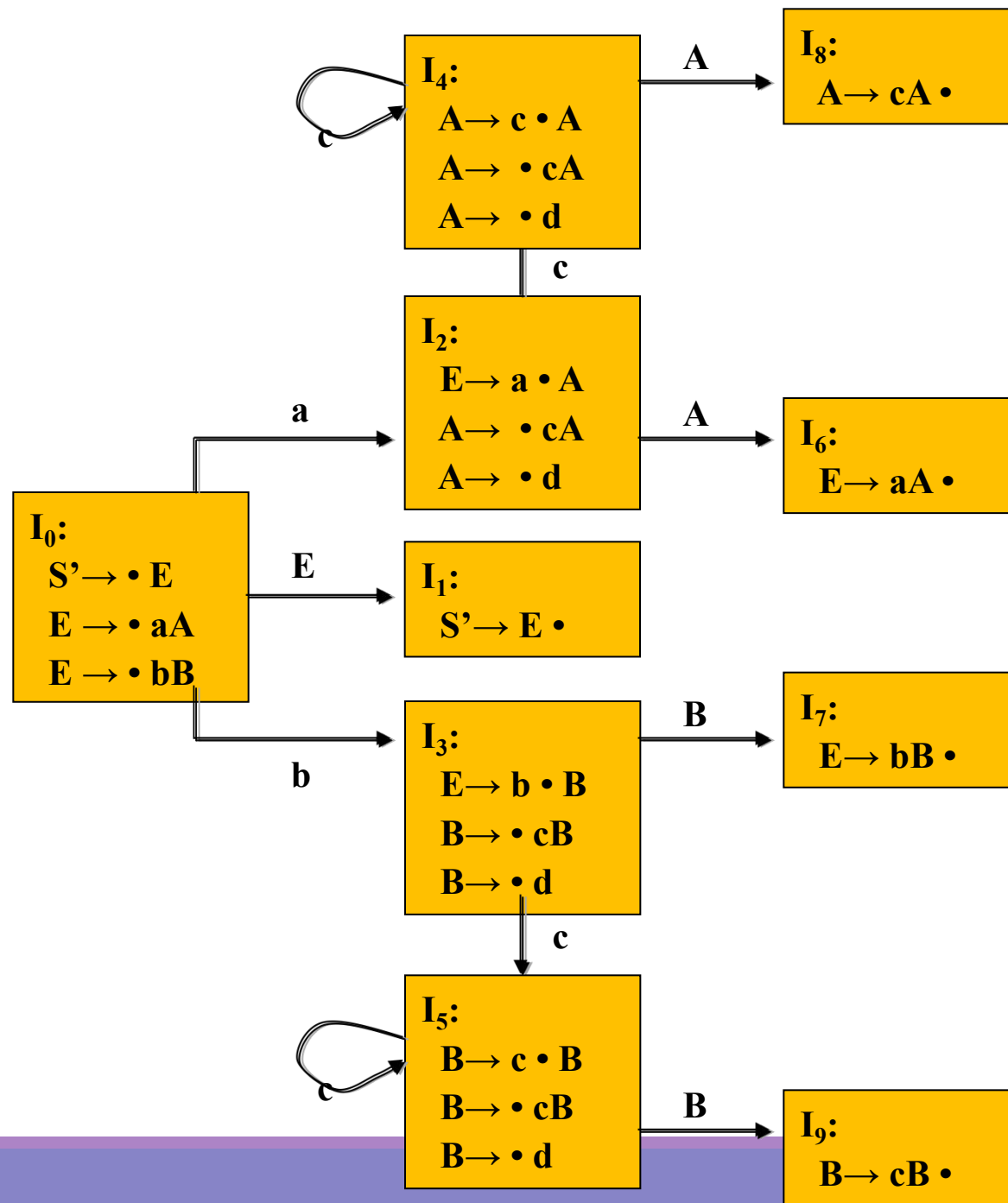
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



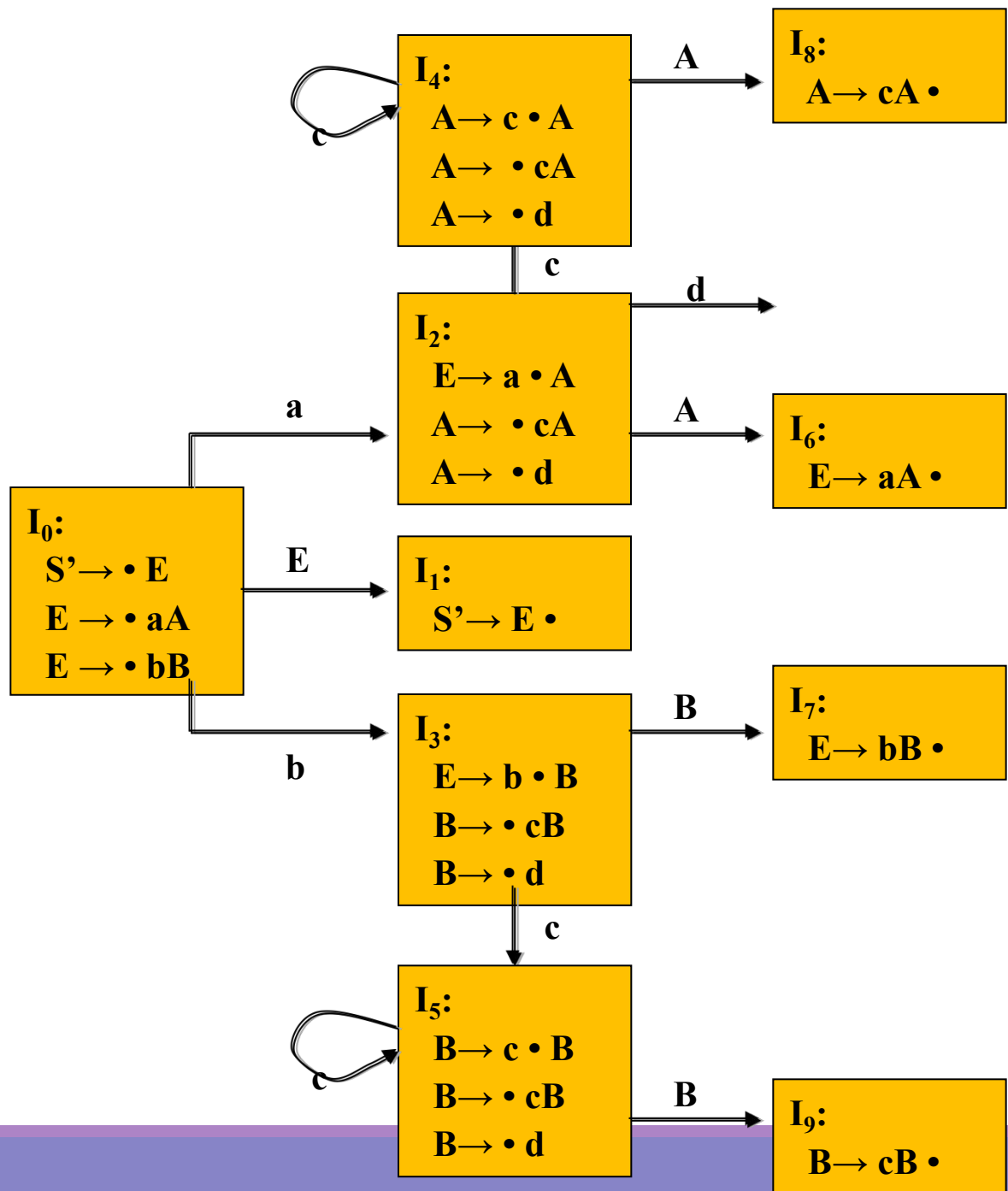
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



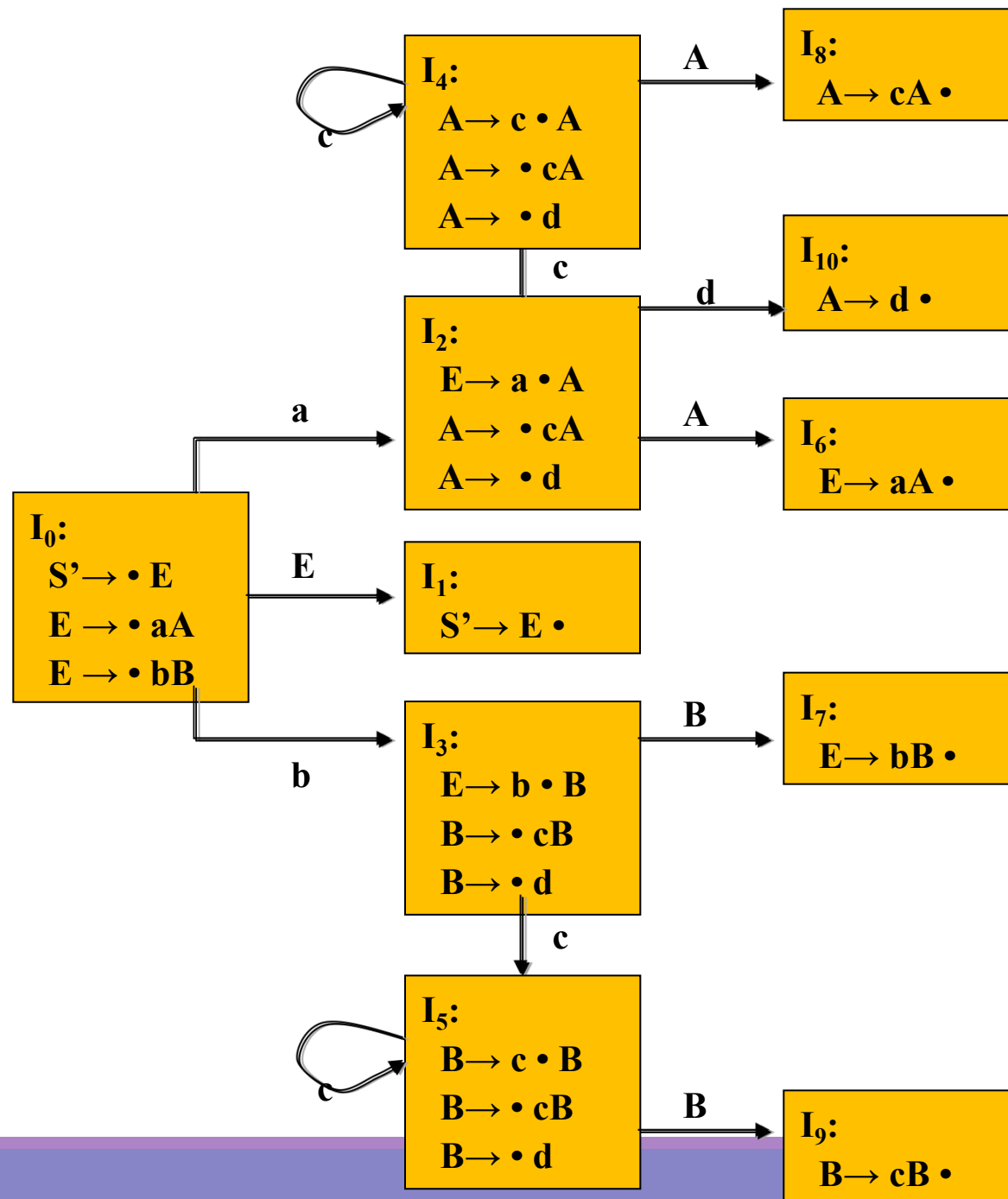
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



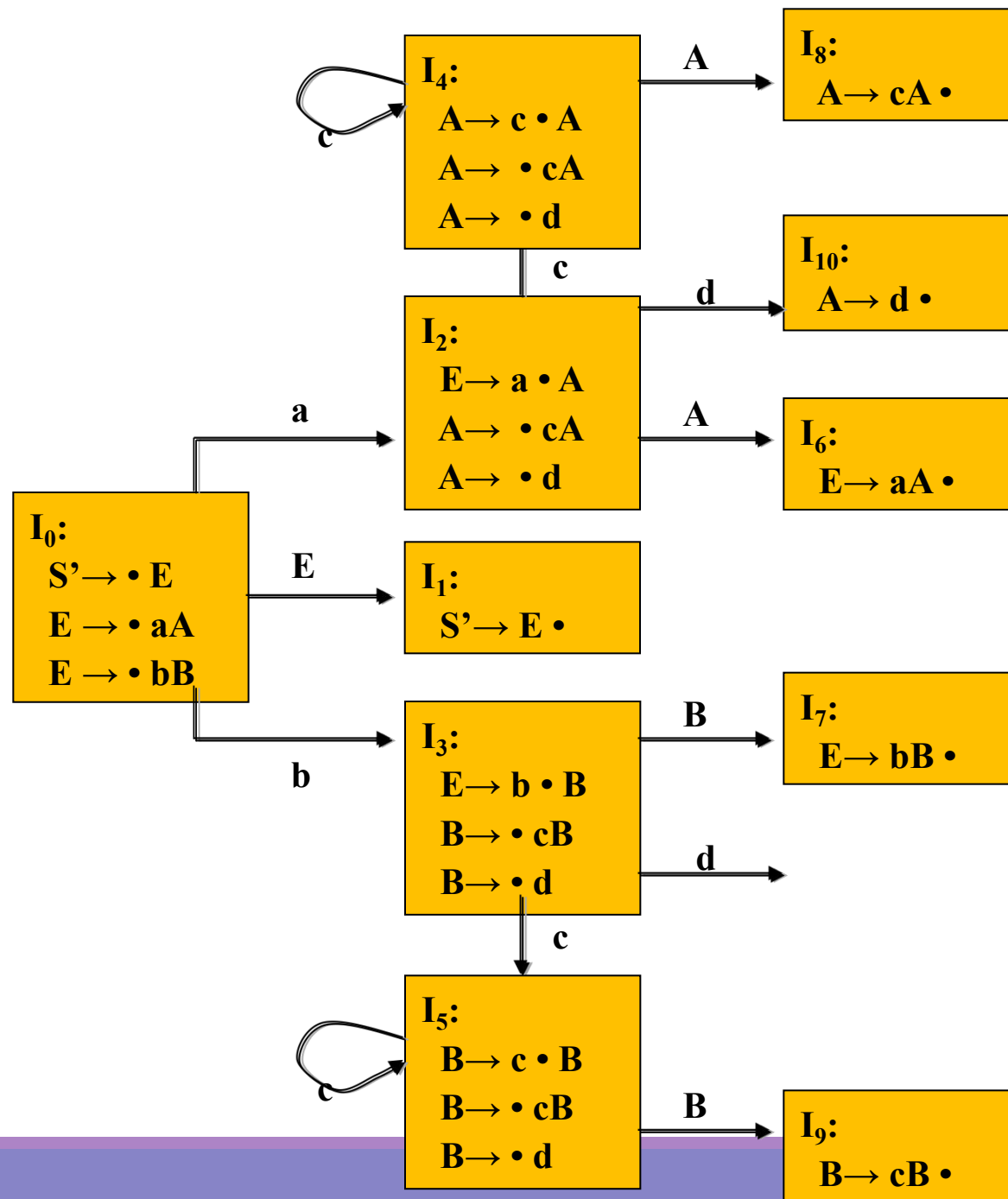
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



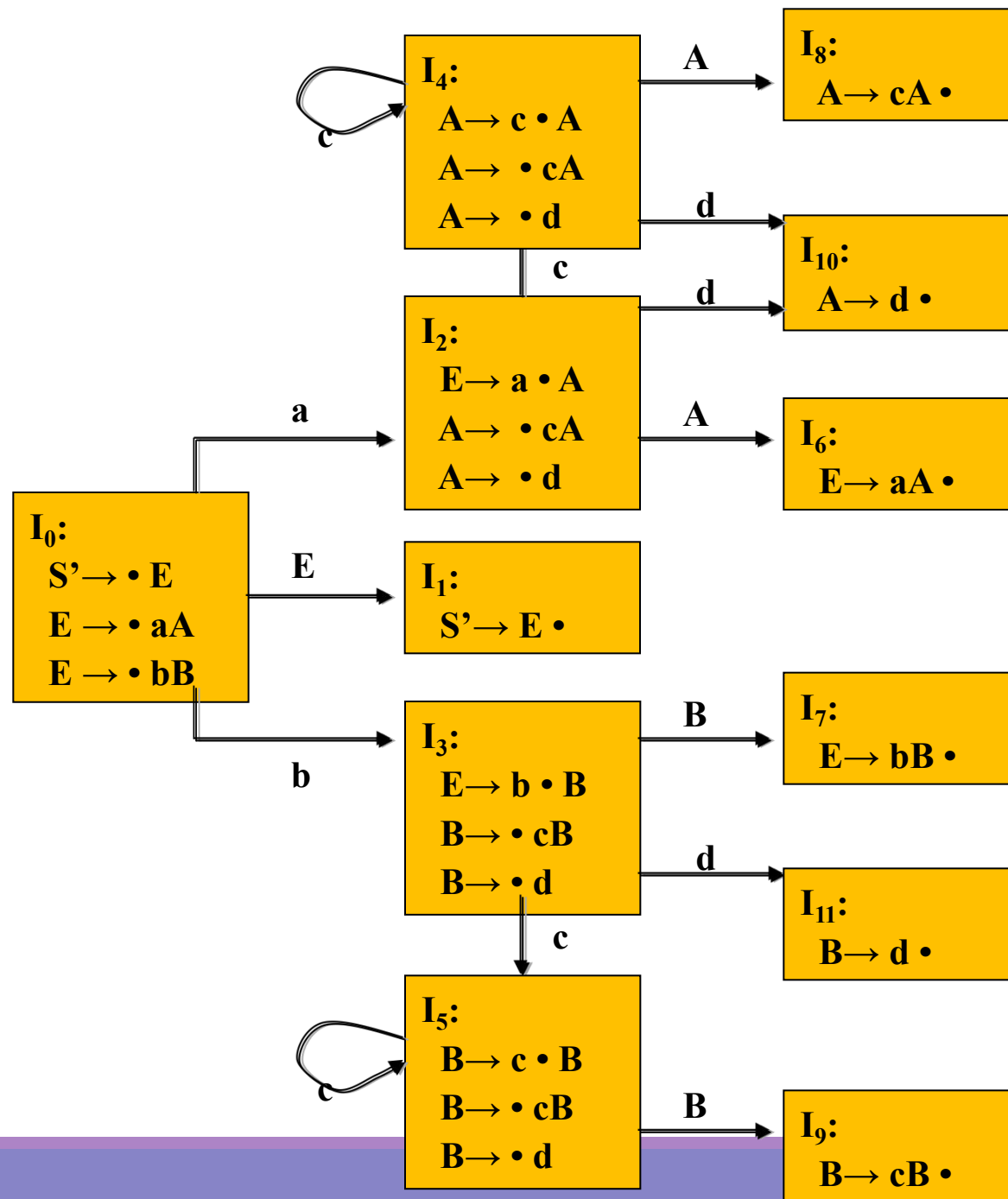
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



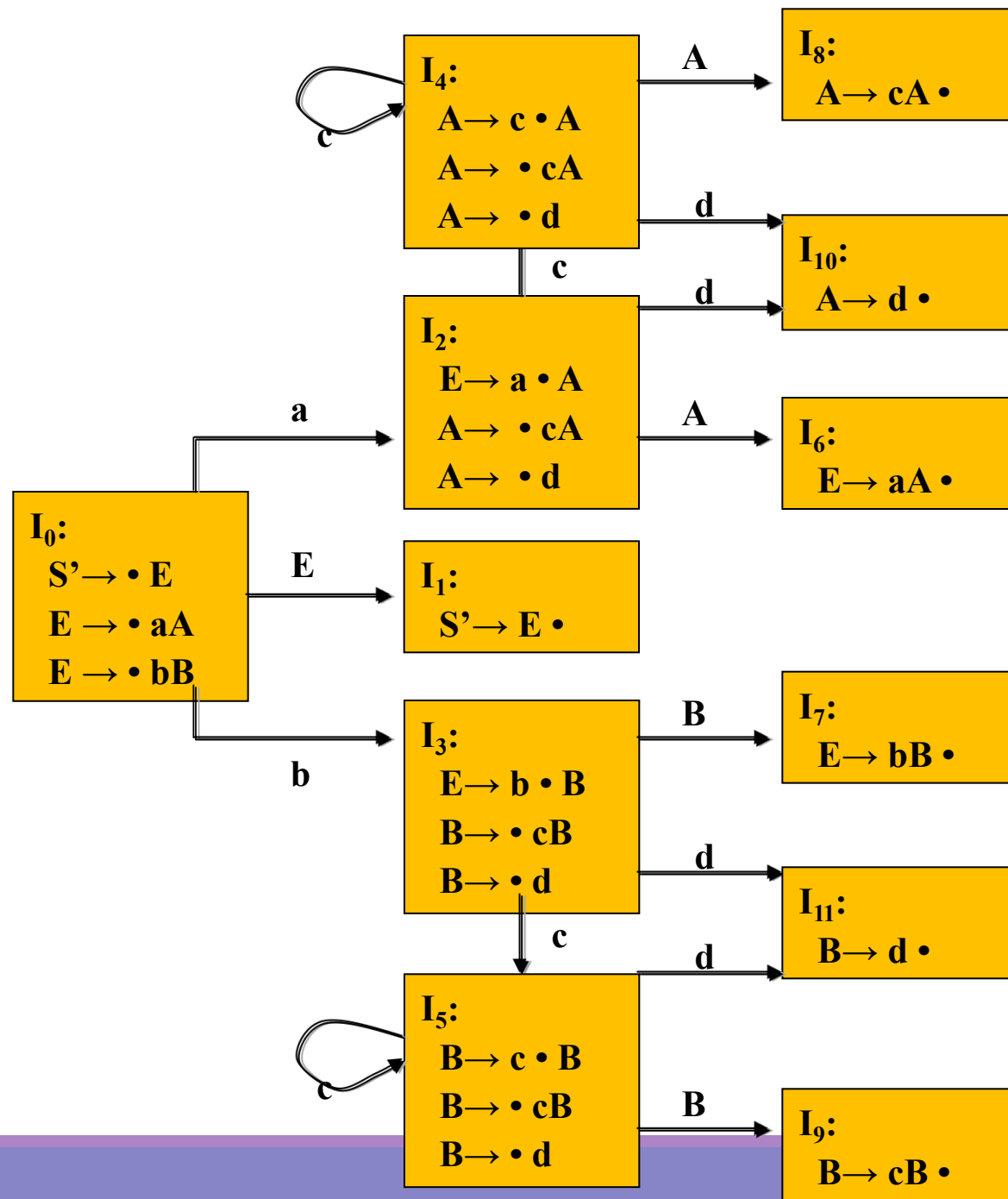
- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$



- 0) $S' \rightarrow E$
- 1) $E \rightarrow \underline{a}A$
- 2) $E \rightarrow \underline{b}B$
- 3) $A \rightarrow \underline{c}A$
- 4) $A \rightarrow d$
- 5) $B \rightarrow \underline{C}b$
- 6) $B \rightarrow d$

LR(0)分析表的构造

(4) LR(0)项目集规范族

构成识别一个文法的可行前缀的DFA的项目集(状态)的全体称为这个文法的LR(0)项目集规范族。上例中文法G的LR(0)项目集规范族为 $\{I_0, I_1, I_2, I_3, \dots, I_{11}\}$

$$I_i = \{ A \rightarrow \beta_1 \cdot b \beta_2, B \rightarrow \beta \cdot, C \rightarrow \beta \cdot \}$$

(5) LR(0)文法

1) 冲突项目

如果一个项目集中既有移进项目又含有归约项目，或一个项目集中有两个以上不同归约项目，则称这些项目是冲突项目。前面我们构造的项目集还没有冲突项目

2) LR(0)文法

如果一个文法的项目规范族的每个项目集不存在任何冲突项目，则称该文法为LR(0)文法。

如：上例文法的LR(0)项目集规范族的每个项目集中就不存在冲突项目，所以该文法就是LR(0)文法。

LR(0)分析表的构造

(6) LR (0) 分析表的构造

对于LR (0) 文法，我们构造出识别可行前缀DFA后，我们就可以根据DFA的状态转换图来构造LR (0) 分析表。下面给出构造LR (0) 分析表的算法

LR(0)分析表的构造

假定 $C = \{I_0, I_1, I_2, I_3, \dots, I_n\}$, 方便起见, 用整数 $0, 1, 2, 3, \dots, n$ 表示状态 $I_0, I_1, I_2, I_3, \dots, I_n$ 。

1) 对于每个项目集 I_i 中有形如 $A \rightarrow \beta_1 \cdot X \beta_2$ 项目, 且 $GO(I_i, X) = I_j$,
若 $X = a \in V_T$, 则置 $ACTION[i, a] = S_j$, 若 $X \in V_N$, 则置
 $GOTO[i, X] = j$

如:

- I_0 中有 $E \rightarrow \cdot aA$, $GO(I_0, a) = I_2, a \in V_T$,
所以置 $ACTION[0, a] = S_2$
- I_2 中有 $E \rightarrow a \cdot A$, $GO(I_2, A) = I_6, A \in V_N$
所以置 $GOTO[2, A] = 6$

LR(0)分析表的构造

2)若归约项目 $A \rightarrow \beta \cdot$ 属于 I_i , 设 $A \rightarrow \beta$ 是文法第 j 个产生式, 则对任意终结符 a 和句子右界符 $\$$, 均置 $\text{ACTION}[i, a \text{ 或 } \$] = r_j$, 表示按文法第 j 条产生式将符号栈顶符号串 β 归约为 A 。

如: I_6 有项目 $E \rightarrow aA \cdot$, 其产生式 $E \rightarrow aA$ 是文法的第一个产生式, 所以置

$\text{ACTION}[6, a] = \text{ACTION}[6, b] = \text{ACTION}[6, c] = \text{ACTION}[6, d] = \text{ACTION}[6, \$] = r_1$

LR(0)分析表的构造

3)若接受项目 $S' \rightarrow S \cdot$ 属于 I_i ，则置 $\text{ACTION}[i, \$] = \text{acc}$ ，表示接受，如：

$S' \rightarrow E \cdot$ 属于 I_1 ，所以置 $\text{ACTION}[1, \$] = \text{acc}$

4)分析表中，凡不能用前3个规则填入信息的空白格位置上，均表示出错。

LR(0)分析表的构造-总结

- (1) 写出给定文法G的增广文法G'并编号，同时写出全部项目
- (2) 写出G'初始状态 I_0 ，项目集基本项目 $S' \rightarrow \cdot S$ ，并由此基本项目求 I_0 项目集合。
- (3) 由 I_0 项目集合，再根据GO函数和CLOSURE求LR（0）项目其它项目集 $I_1, I_2 \dots$
- (4) 构造识别可行前缀的DFA
- (5) 由DFA根据算法构造LR（0）分析表

SLR分析表的构造

(1) 问题提出

上面介绍的LR(0)方法，是从左向右扫描源程序，当到达某产生式右部最右符号时，便识别出这条产生式，并且对于每一句柄，无需查看句柄之外任何输入符号。这种分析方法要求文法的每一个项目集都不含冲突性的项目。但通常程序设计语言文法不一定符合这种要求。

例如 LR(0)项目集规范族中有这样一个项目集 I_i

$$I_i = \{ A \rightarrow \beta_1 \cdot b \beta_2, B \rightarrow \beta \cdot, C \rightarrow \beta \cdot \}$$

其中第一个项目是移进项目，第二、三个项目是归约项目。仔细分析前面讨论的LR(0)分析表的构造可以知道，由于这三个项目相互冲突，因而使得LR(0)分析表中出现多重定义的分析动作。其原因在于

SLR分析表的构造

LR(0)分析表构造规则2)，当有归约项目 $B \rightarrow \beta \cdot$ 时，不论当前输入符号是什么，在LR(0)分析表的第i行上均置 r_j (假定 $B \rightarrow \beta$ 是文法第j个产生式)，同样道理，对于项目 $C \rightarrow \beta \cdot$ ，仍在LR(0)分析表第i行上置 r_m (假定 $C \rightarrow \beta$ 是文法第m个产生式)。而 I_i 中第一个项目 $A \rightarrow \beta_1 \cdot b \beta_2$ ，指出将下一输入符b移入分析栈，于是发生了是归约还是移进，如果归约是将栈顶 β 归约为B，还是归约为C。这样使得在LR(0)分析表第i行上造成多重定义。

	a	b	c	d	\$
0					
1					
2					
.					
.					
	r_j	r_j	r_j	$r_j \dots$	r_j
i	r_m	r_m / S_j	r_m	r_m	r_m

SLR分析表的构造

为什么LR(0)分析表构造会出现多重定义？

由于LR(0)分析表构造时，若是归约项目 $A \rightarrow \beta \cdot$ 属于 I_i 时， $A \rightarrow \beta$ 是文法第 j 个产生式，对任意终结符 a 和句子右界符 $\$$ ，均置 $\text{ACTION}[i, a \text{ 或 } \$] = r_j$ ，表示按文法第 j 条产生式将符号栈顶符号串 β 归约为 A 。由于不考虑句柄后任一符号，即不向前看符号，一律为 r_j ，所以当有两个以上归约项目时会出现冲突。

解决这种矛盾办法是在 i 行上根据输入符号 a 决定在第 i 行置上唯一元素。为此我们引入SLR分析法，下面介绍SLR(1)分析表的构造。

SLR分析表的构造

1) 解决冲突项目

对于 $I_i = \{ A \rightarrow \beta_1 \cdot b \beta_2, B \rightarrow \beta \cdot, C \rightarrow \beta \cdot \}$

如果集合 $\text{FOLLOW}(B)$ 和 $\text{FOLLOW}(C)$ 不相交，而且不包含 b ，那么，当状态 I_i 面临任何输入符号 a 时，可采用如下“移进---归约”的决策。

- ①当 $a=b$ 时，则移进，置 $\text{ACTION}[i,a] = S_j$
- ②当 $a \in \text{FOLLOW}(B)$ 时，置 $\text{ACTION}[i,a] = r_j$
- ③当 $a \in \text{FOLLOW}(C)$ 时，置 $\text{ACTION}[i,a] = r_m$
- ④当 a 不属于三种情况之一，置 $\text{ACTION}[i,a] = \text{“ERROR”}$

一般地，若一个项目集 I_i 含有多个移进项目和归约项目，例如

$I_i = \{ A_1 \rightarrow \alpha \cdot a_1 \beta_1, A_2 \rightarrow \alpha \cdot a_2 \beta_2, \dots, A_m \rightarrow \alpha \cdot a_m \beta_m,$

$B_1 \rightarrow \alpha \cdot, B_2 \rightarrow \alpha \cdot, \dots, B_n \rightarrow \alpha \cdot \}$

如果集合 $\{a_1, a_2, \dots, a_m\}$ ， $\text{FOLLOW}(B_1)$ ， $\text{FOLLOW}(B_2)$ ， \dots ， $\text{FOLLOW}(B_n)$

两两不相交时，可类似地根据不同的当前符号，对状态为 i 中的冲突动作进行区分。这种解决“移进---归约”冲突的方法称作SLR方法。

SLR分析表的构造

2) SLR (1) 分析表构造

有了SLR方法之后，只须对LR(0)分析表构造方法②进行修改，其它方法保持不变。即若归约项目 $A \rightarrow \alpha \cdot$ 属于 I_i ，设 $A \rightarrow \alpha$ 是文法第 j 个产生式，则对于属于 $FOLLOW(A)$ 的输入符号 a ，置于 $ACTION [i, a] = r_j$ ，表示按文法的第 j 条产生式 $A \rightarrow \alpha$ 将栈顶符号串 α 归约成 A 。

3) SLR(1)文法

对于给定的文法 G ，若按上述方法构造的分析表不含多重定义的元素，则称文法 G 是SLR(1)文法。这里SLR(1)中的S代表Simple(简单)的意思，而数字1代表查看句柄外一个输入符号，即在分析过程中至多只需要向前查看一个符号。

SLR分析表的构造

4) SLR (1) 分析表构造举例

例：设有文法 G

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

构造该文法SLR(1)分析表。

SLR分析表的构造

① 将文法G增广为G'，同时对每一产生式进行编号

(0) $S' \rightarrow E$

(4) $T \rightarrow F$

(1) $E \rightarrow E + T$

(5) $F \rightarrow (E)$

(2) $E \rightarrow T$

(6) $F \rightarrow \text{id}$

(3) $T \rightarrow T * F$

SLR分析表的构造

②对G'构造文法LR(0)项目集规范族如下:

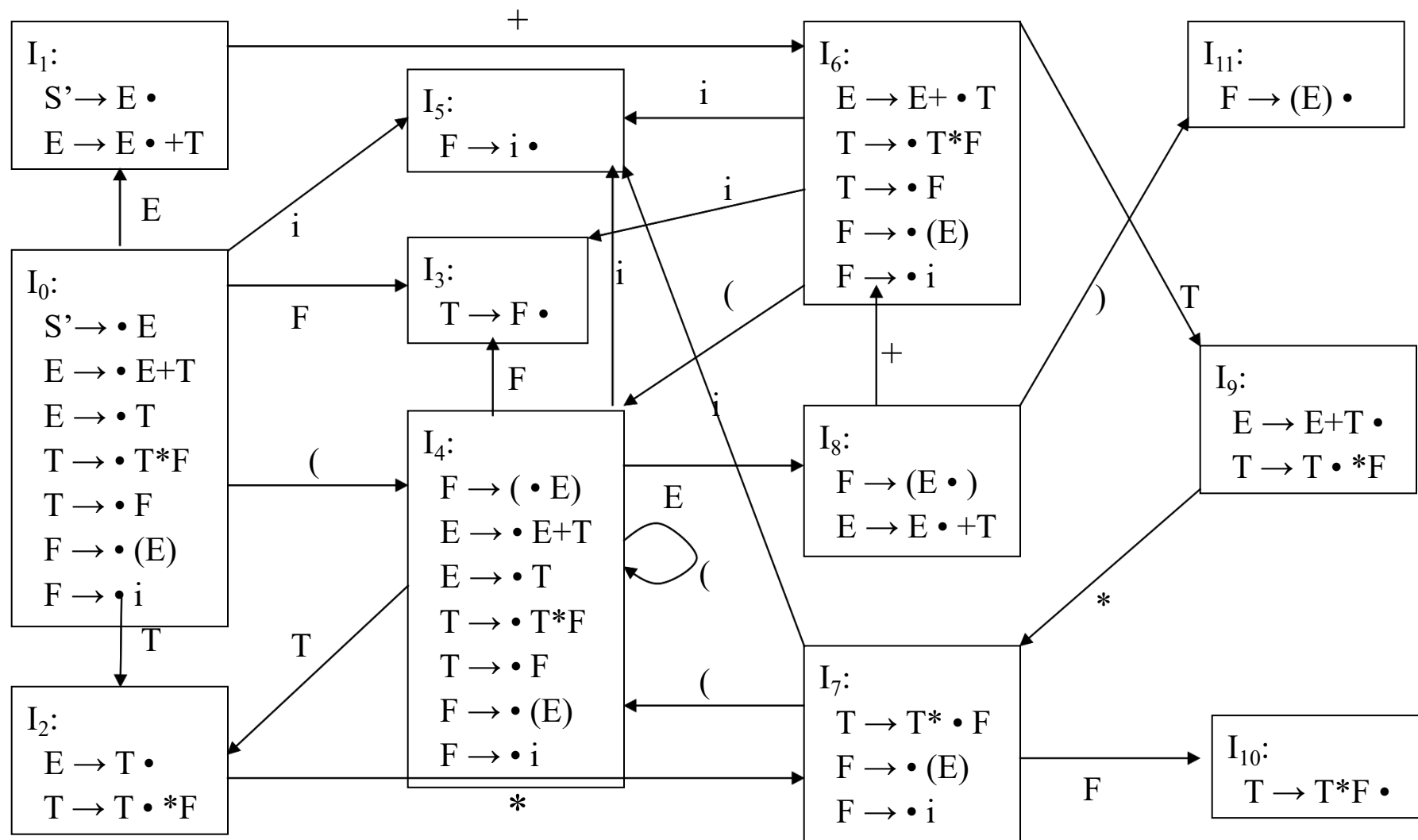
$I_0: S' \rightarrow \cdot E$
 $E \rightarrow \cdot E + T$
 $E \rightarrow \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot id$
 $I_1: S' \rightarrow E \cdot$
 $E \rightarrow E \cdot + T$

$I_2: E \rightarrow T \cdot$
 $T \rightarrow T \cdot * F$
 $I_3: T \rightarrow F \cdot$
 $I_4: F \rightarrow (\cdot E)$
 $E \rightarrow \cdot E + T$
 $E \rightarrow \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot id$

$I_5: F \rightarrow id \cdot$
 $I_6: E \rightarrow E + \cdot T$
 $T \rightarrow \cdot T * F$
 $T \rightarrow \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot id$
 $I_7: T \rightarrow T * \cdot F$
 $F \rightarrow \cdot (E)$
 $F \rightarrow \cdot id$

$I_8: F \rightarrow (E \cdot)$
 $E \rightarrow E \cdot + T$
 $I_9: E \rightarrow E + T \cdot$
 $T \rightarrow T \cdot * F$
 $I_{10}: T \rightarrow T * F \cdot$
 $I_{11}: F \rightarrow (E) \cdot$

③ 取这些项目集作为各状态，并根据转换函数G0画出识别文法 G' 的有穷自动机，



SLR分析表的构造

④ 用SLR方法解决“移进---归约”冲突。

在十二个项目集中， I_1 、 I_2 和 I_9 都含有“移进---归约”冲突，其解决办法是：

对于项目集 $I_1 = \{S' \rightarrow E \cdot, E \rightarrow E \cdot + T\}$ ，由于集合

$FOLLOW(S') = \{\$ \}$ 与集合 $\{+\}$ 不相交，

所以当状态为1时，面临着输入符号为+时便移进，而面临着输入符号为\$时，则按产生式 $S' \rightarrow E$ 归约。

对于项目集 $I_2 = \{E \rightarrow T \cdot, T \rightarrow T \cdot * F\}$ ，由于集合

$FOLLOW(E) = \{+,), \$ \}$ 与集合 $\{*\}$ 不相交，

因此状态2面临输入符号为*时移进，而面临输入符号为+或)或\$时，按产生式 $E \rightarrow T$ 归约。

对于项目集 $I_9 = \{E \rightarrow E + T \cdot, T \rightarrow T \cdot * F\}$ ，同样由于

$FOLLOW(E) = \{+,), \$ \}$ 与集合 $\{*\}$ 不相交，因此状态9面临着输入符号为*时移进，面临着输入符号为+或)或\$时，按产生式 $E \rightarrow E + T$ 归约。

⑤ 构造SLR(1)分析表

状态	ACTION (动作)						GOTO (状态转换)		
	i	+	*	()	\$	E	T	F
0	S ₅			S ₄			1	2	3
1		S ₆				acc			
2		r ₂	S ₇		r ₂	r ₂			
3		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			
6	S ₅			S ₄				9	3
7	S ₅			S ₄					10
8		S ₆			S ₁₁				
9		r ₁	S ₇		r ₁	r ₁			
10		r ₃	r ₃		r ₃	r ₃			
11		r ₅	r ₅		r ₅	r ₅			

LR(1)分析表的构造

(1) 问题的提出 2014607

SLR(1) 也存在不足, 即如果冲突项目的非终结符FOLLOW集与有关集合相交时, 就不能用SLR(1) 方法解决。

例: 设增广文法 G' :

(0) $S' \rightarrow S$

(1) $S \rightarrow C bBA$

(2) $A \rightarrow Aab$

(3) $A \rightarrow ab$

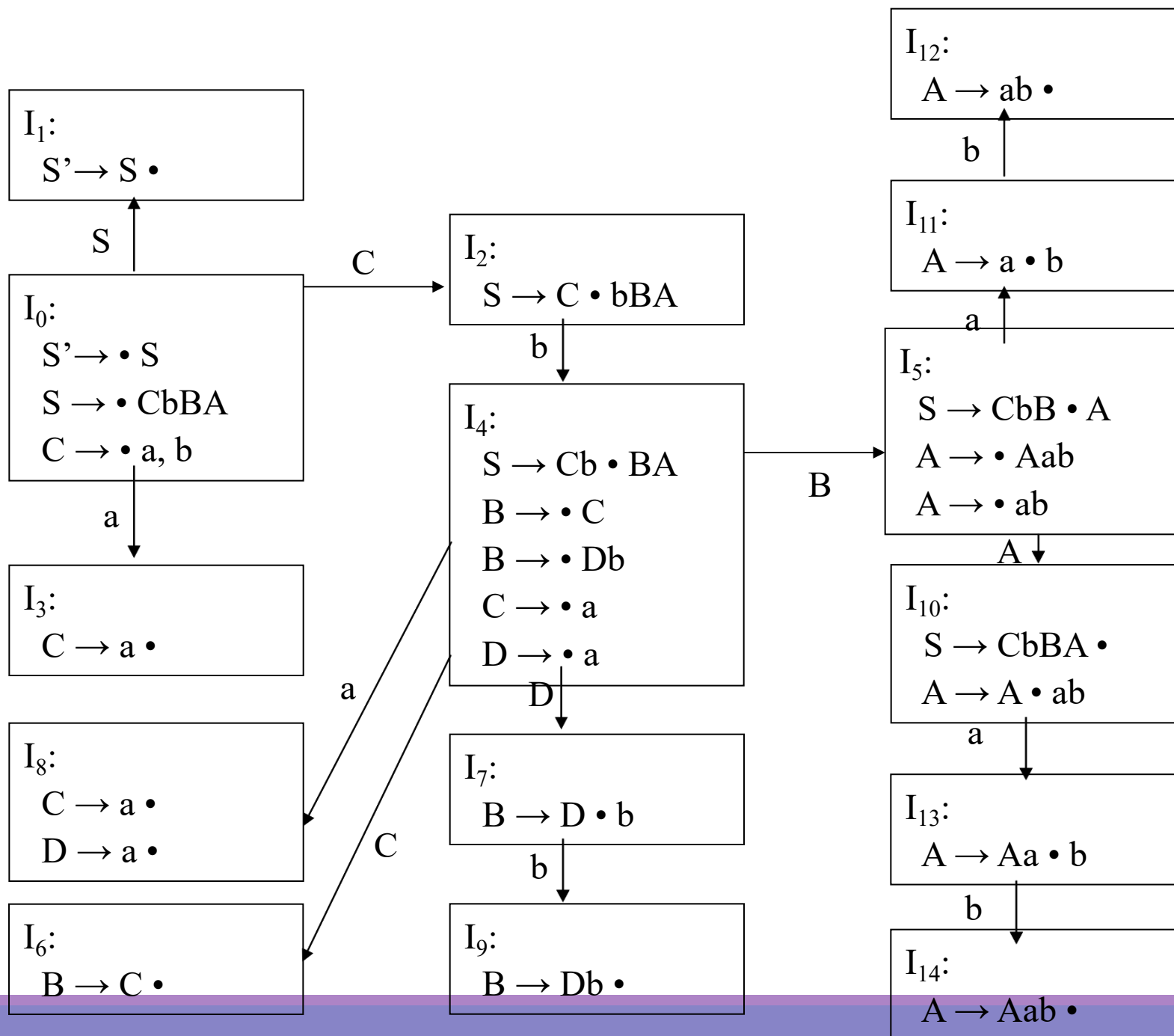
(4) $B \rightarrow C$

(5) $B \rightarrow D b$

(6) $C \rightarrow a$

(7) $D \rightarrow a$

识别此文法的全部可行前缀的DFA如下图所示



- (0) $S' \rightarrow S$
- (1) $S \rightarrow C bBA$
- (2) $A \rightarrow Aab$
- (3) $A \rightarrow ab$
- (4) $B \rightarrow C$
- (5) $B \rightarrow Db$
- (6) $C \rightarrow a$
- (7) $D \rightarrow a$

LR(1)分析表的构造

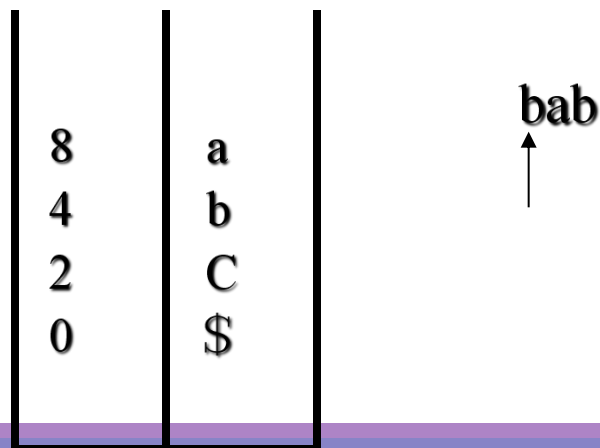
由图可知，项目集 $I_{10} = \{S \rightarrow C b B a \cdot, A \rightarrow A \cdot ab\}$

存在“移进---归约”冲突，由于 $FOLLOW(S) = \{\$ \}$ 与 $\{a\}$ 不相交，故上述冲突可通过SLR(1)产生式得到解决。

项目集 $I_8 = \{C \rightarrow a \cdot, D \rightarrow a \cdot\}$ 中，含有归约冲突项目，由于

$FOLLOW(C) = \{b, a\}$ 与 $FOLLOW(D) = \{b\}$ 相交，

故不能用SLR(1)方法简单地解决项目冲突。产生这种困境的原因是SLR(1)分析方法包含的信息还不够。例如，在分析某一个时刻：



LR(1)分析表的构造

此时栈顶状态为8(I8),栈顶符号为a, 对于下一个输入的符号为b时, 此时, 由产生式 $C \rightarrow a$ 和 $D \rightarrow a$, 是将a归约成C还是D呢?

如果将a归约成C,此时栈中就变成 \$ CbC,然后再读入b,栈中就变成 \$ CbCb,而该文法不存在可行前缀CbCb, 因为分析栈中应该全是可行前缀。

如果将a归约成D,则CbDb是可行前缀, Db是句柄 ($B \rightarrow Db$ 是文法产生式), 为什么用 $D \rightarrow a$ 归约, 而不用 $C \rightarrow a$ 归约, 这说明SLR(1)分析方法包含的信息还不够。

所以在归约时, 不但要向前看一个符号, 而且还要看栈中符号串情况, 才可以知道用某种产生式归约, 为了解决这个问题, 我们必须将原LR(0)的项目定义加以扩充, 而变成LR(1)项目, 也就是说, 还要看栈中可行前缀是什么, 再选择归约, 即项目 $D \rightarrow a \cdot$ 对Cba有效, 而项目 $C \rightarrow a \cdot$ 对Cba无效。

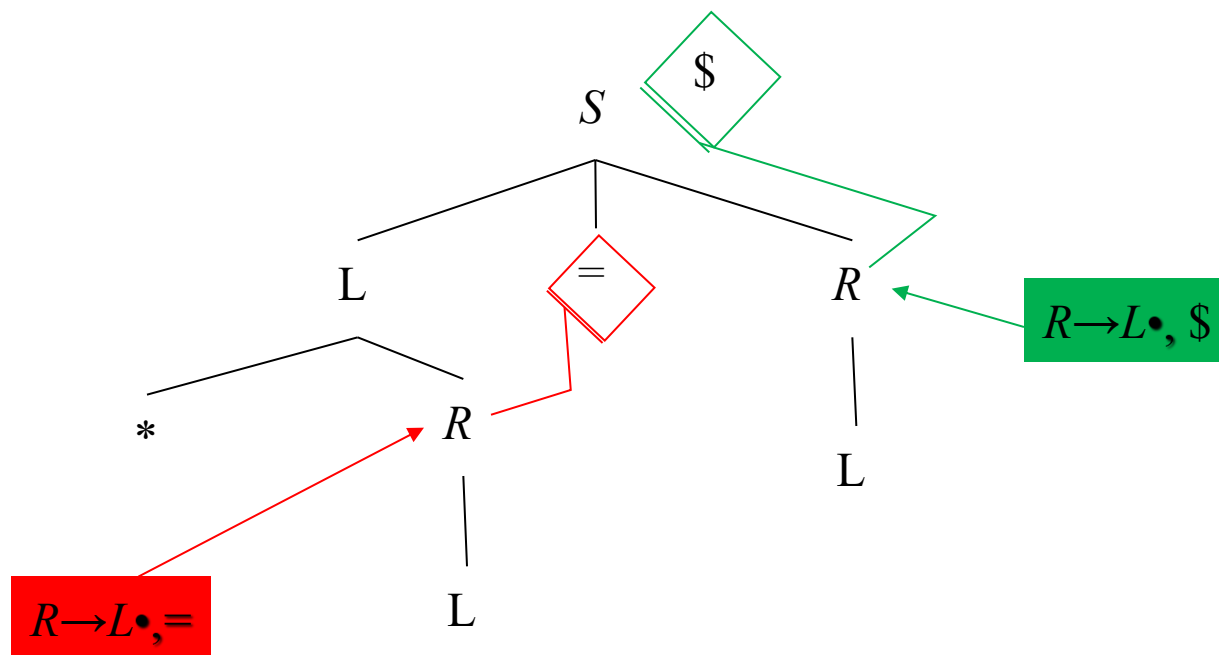
SLR分析存在的问题

SLR只是简单地考察下一个输入符号 **b** 是否属于与归约项目 **$A \rightarrow \alpha$** 相关联的 **$FOLLOW(A)$** ，但 **$b \in FOLLOW(A)$** 只是归约 **α** 的一个**必要条件**，而非**充分条件**

对于产生式 **$A \rightarrow \alpha$** 的归约，在**不同的使用位置**， **A** 会要求不同的后继符号

- 0) $S' \rightarrow S$
- 1) $S \rightarrow L = R$
- 2) $S \rightarrow R$
- 3) $L \rightarrow *R$
- 4) $L \rightarrow id$
- 5) $R \rightarrow L$

A	FOLLOW(A)
S	\$
L	=, \$
R	=, \$



在特定位置， **A** 的后继符集合是 **$FOLLOW(A)$** 的**子集**

LR(1)分析表的构造

- | | |
|---------------------------|-------------------------|
| (0) $S' \rightarrow S$ | (4) $B \rightarrow C$ |
| (1) $S \rightarrow C bBA$ | (5) $B \rightarrow D b$ |
| (2) $A \rightarrow Aab$ | (6) $C \rightarrow a$ |
| (3) $A \rightarrow ab$ | (7) $D \rightarrow a$ |

(2) LR (1) 项目

1) 定义

所谓一个LR (1) 项目 $[A \rightarrow \alpha \cdot \beta, a]$ 对可行前缀 $\gamma = \delta \alpha$ 有效, 是指存在规范推导

$$S \Rightarrow^* \delta A \omega \Rightarrow \delta \alpha \beta \omega \quad (\text{显然 } \delta \alpha \beta \text{ 是可归约可行前缀})$$

其中满足下列条件:

- ① 当 $\omega \neq \varepsilon$ 时, a 是 ω 首符号; ② 当 $\omega = \varepsilon$ 时, $a = \$$ 。

例如上例文法, 因有一个规范推导

$$S \Rightarrow C bBA \Rightarrow C bBab \Rightarrow CbDbab$$

$$\text{即 } S \Rightarrow^* C bBab \Rightarrow CbDbab$$

通过上面的定义中分别令

$$\delta = Cb, A = B, \alpha = D, \beta = b, \omega = ab$$

故LR (1) 项目 $[B \rightarrow D \cdot b, a]$ 对可行前缀 $\gamma = CbD$ 有效。

再看它的另一个规范推导

$$S \Rightarrow CbBA \Rightarrow CbBab \Rightarrow CbDbab \Rightarrow Cbabab$$

$$\text{即 } S \Rightarrow^* CbDbab \Rightarrow Cbabab$$

其中 $\delta = Cb, A = D, \alpha = a, \beta = \varepsilon, \omega = bab$, 故LR (1) 项目

$[D \rightarrow a \cdot, b]$ 对可行前缀 $\gamma = Cba$ 有效, 应将栈顶符号 a 归约为 D 。

存在 $A \rightarrow ab$ 和 $B \rightarrow Db$ 两条产生式

存在产生式 $D \rightarrow a$

8	a
4	b
2	C
0	\$

bab
↑

LR(1)分析表的构造

2) LR (1) 项目集规范族的构造

构造有效的LR (1) 项目集规范族的办法本质上和构造LR (0) 项目集规范族的办法是一样的。我们也需要两个函数CLOSURE和GO。假定 I 是一个项目集，它的闭包CLOSURE (I) 可按下述方式构造：

- ① I 的任何项目都属于CLOSURE (I) ；
- ② 若项目 $[A \rightarrow \alpha \cdot B \beta, a]$ 属于CLOSURE (I) ，并对可行前缀 $\gamma = \delta \alpha$ 有效，若有 $B \rightarrow \eta$ 产生式，那么对FIRST (βa) 中每个终结符b，形如 $[B \rightarrow \cdot \eta, b]$ 的所有项目也属于CLOSURE (I) ；
- ③ 重复执行步骤②，直到CLOSURE (I) 不再扩大，所得CLOSURE (I) 便是LR (1) 一个项目集。

LR(1)分析表的构造

3)转换函数的构造

至于函数 $GO(I, X)$ ，其中 I 为一LR(1)项目集， X 为一文法符号，与LR(0)文法相类似，我们将它定义为

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

其中

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

有了上述 $CLOSURE(I)$ 和 $GO(I, X)$ 的定义之后，采用与LR(0)类似方法可以构造出给定文法 G 的LR(1)项目规范族 C 及状态转换图。

例如，对于以下文法，其LR(1)项目集及状态转换图如下图所示

$$(0) S' \rightarrow S$$

$$(4) B \rightarrow C$$

$$(1) S \rightarrow C b B A$$

$$(5) B \rightarrow D b$$

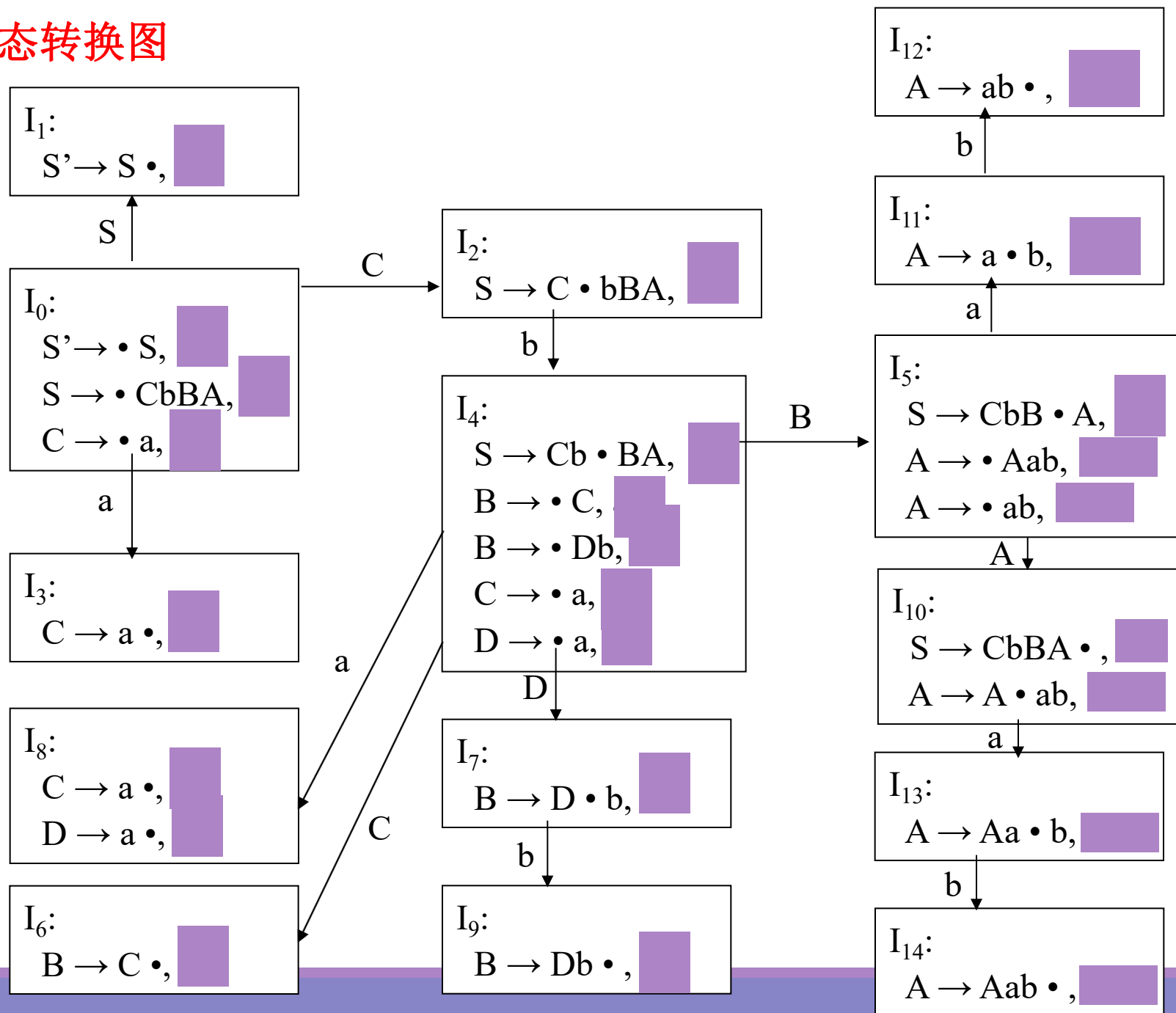
$$(2) A \rightarrow A a b$$

$$(6) C \rightarrow a$$

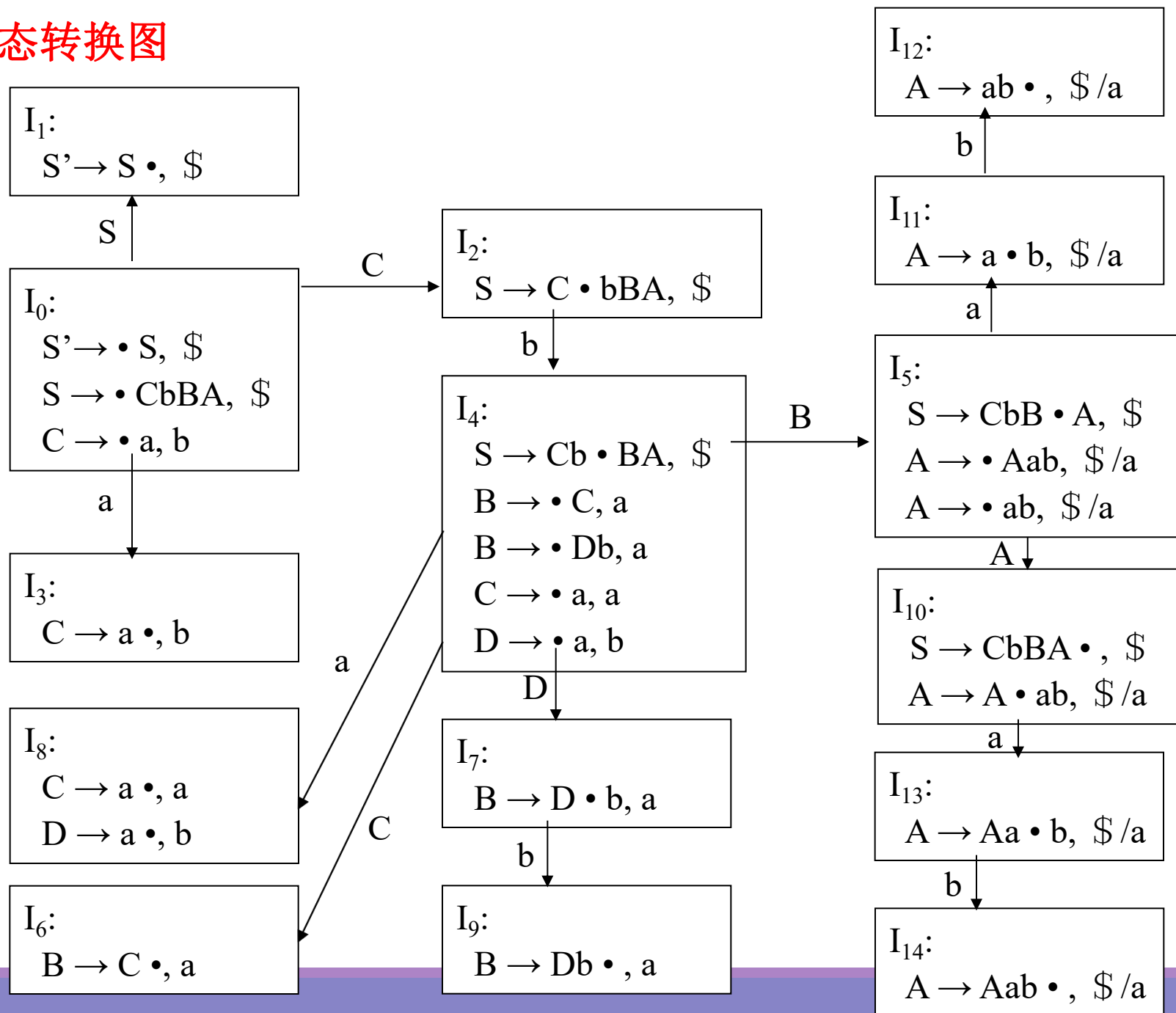
$$(3) A \rightarrow a b$$

$$(7) D \rightarrow a$$

LR(1)项目集及状态转换图



LR(1)项目集及状态转换图



LR(1)分析表的构造

- 1) 若项目 $[A \rightarrow \alpha \cdot X \beta, b]$ 属于 I_i , 且 $GO(I_i, X) = I_j$, 当 $X \in V_T$, 置 $ACTION[i, X] = S_j$, 当 $X \in V_N$, 则置 $GOTO[i, X] = j$;
可知 $[C \rightarrow \cdot a, b] \in I_0$, $a \in V_T$ 且 $GO[I_0, a] = I_3$
所以 $ACTION[0, a] = S_3$; 又如 $[S \rightarrow \cdot CbBA, \$] \in I_0$, $C \in V_N$
且 $GO[I_0, C] = I_2$ 所以 $GOTO[0, C] = 2$
- 2) 若项目 $[A \rightarrow \alpha \cdot, a]$ 属于 I_i , 设 $A \rightarrow \alpha$ 是文法第 j 个产生式, 则置 $ACTION[i, a] = r_j$, 表示按文法第 j 个产生式将 α 归约为 A ;
如: 上例中 $[C \rightarrow a \cdot, a] \in I_8$, 而 $C \rightarrow a$ 为文法的第 6 条产生式
则 $ACTION[8, a] = r_6$
- 3) 若项目 $[S' \rightarrow S \cdot, \$]$ 属于 I_i , 则置 $ACTION[i, \$] = acc$, 表示接受。
- 4) 分析表中不能按上述产生式填入信息的空白格位置上, 均表示出错 ERROR

LR(1)
分析表

状态	ACTION			GOTO				
	a	b	\$	S	A	B	C	D
0	S ₃			1			2	
1			acc					
2		S ₄						
3		r ₆						
4	S ₈					5	6	7
5	S ₁₁				10			
6	r ₄							
7		S ₉						
8	r ₆	r ₇						
9	r ₅							
10	S ₁₃		r ₁					
11		S ₁₂						
12	r ₃		r ₃					
13		S ₁₄						
14	r ₂		r ₂					

LR(1)分析表的构造

(3) 说明

1) 按照上述算法构造的分析表，若不存在多重定义的元素，则称此分析表为**规范LR（1）分析表**。使用这种分析表的分析器叫做规范LR分析器。具有规范LR（1）分析表的文法称为一个**LR（1）文法**。

2) LR（1）分析法比LR（0），SLR（1）适用范围更广，对多数程序设计语言来说有足够有效分析能力。

3) 若LR（1）分析法不可进行有效分析，即分析表项有多重定义，可继续向前搜k个符号，相应分析表称LR（K）分析表，具有规范LR（K）文法。

4) 任何二义性文法都不是LR（K）文法。

LALR分析表的构造

下面我们来介绍LALR (Look Ahead---LR)向前看LR分析法。

(1) 问题的提出

LALR分析法与SLR相类似,但功能比SLR(1)强,比LR(1)弱,LALR分析表比LR分析表要小得多。对于同一文法,LALR分析表与SLR分析表具有相同数目的状态(SLR是不区分向前搜索符的)。例如,对PASCAL语言来说,处理它的LALR分析表一般要设置几百个状态,若用规范LR分析表则可能要上千个状态。因此,构造LALR分析表要比构造LR分析表经济得多。

设文法G:

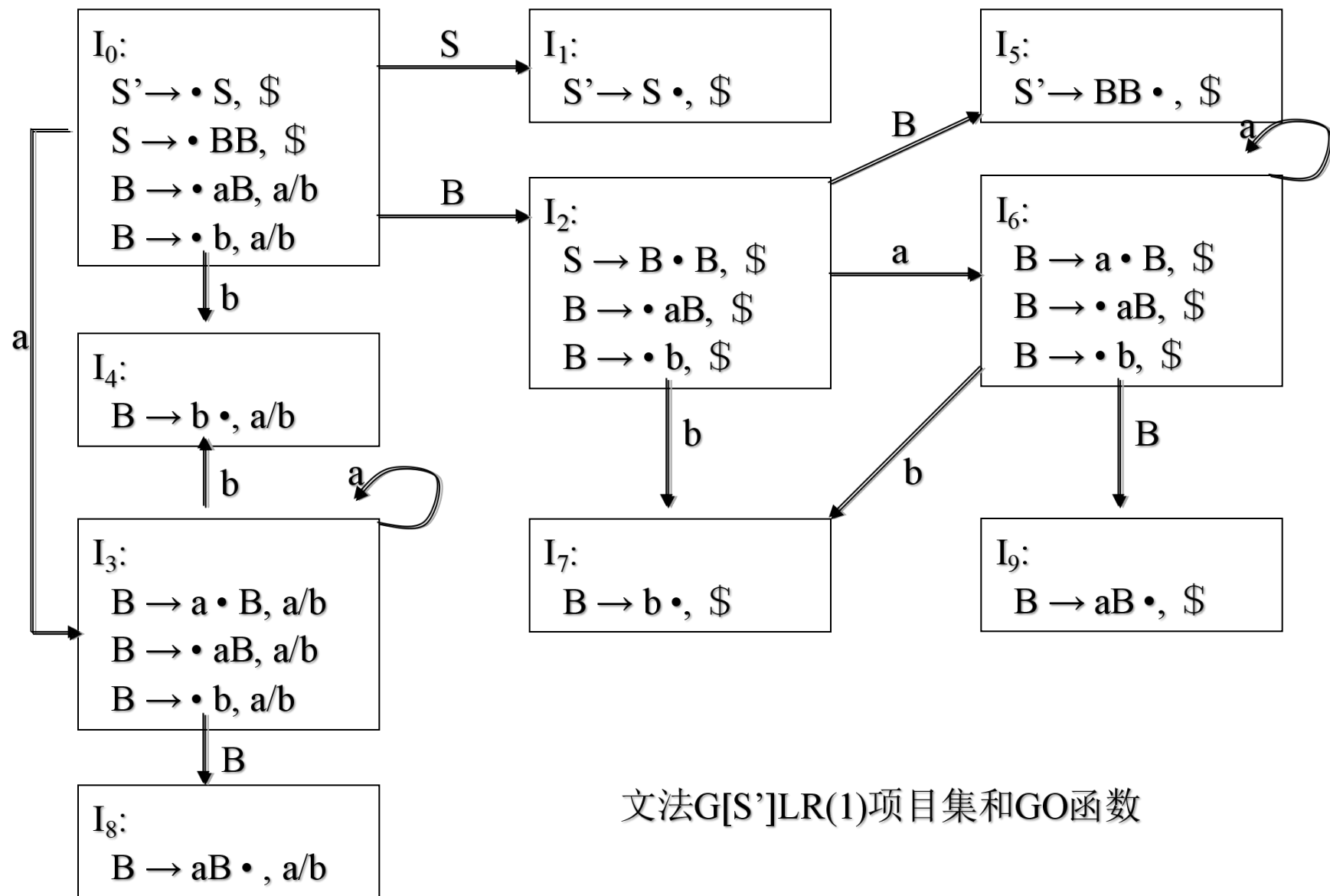
$$(0) S' \rightarrow S$$

$$(2) B \rightarrow aB$$

$$(1) S \rightarrow B B$$

$$(3) B \rightarrow b$$

由该文法,我们LR(1)项目集及转换函数如下图所示:



文法 $G[S']$ LR(1)项目集和GO函数

文法G[S']LR(1)分析表

状态	ACTION			GOTO	
	a	b	\$	S	B
0	S ₃	S ₄		1	2
1			acc		
2	S ₆	S ₇			5
3	S ₃	S ₄			8
4	r ₃	r ₃			
5			r ₁		
6	S ₆	S ₇			9
7			r ₃		
8	r ₂	r ₂			
9			r ₂		

LALR分析表的构造

从图中可以看出， I_4 和 I_7 ，它们只有一个项目，而且第一个成分（ $B \rightarrow b \cdot$ ）相同（核心项），不同的只是第二成分向前搜索符（分别为 a/b 和 $\$$ ）该文法语言为 a^*ba^*b 。假定规范LR分析器正在分析输入串 $aa...baa...b\$$ ，分析器把第一组 a 和后面第一个 b 移进栈，此时进入状态4（ I_4 ），如果下一个输入符号是 a 或 b 时，分析器将使用产生式 $B \rightarrow b$ 把栈顶的 b 归约为 B 。状态4的作用在于，若第一个 b 后是 $\$$ ，它就及时地予以报错。当读入第二个 b 后分析器进入状态7（ I_7 ），若状态7面临着输入符号不是 $\$$ ，而是 a 或 b 时，就立即报告错误；只有当它看到句末符 $\$$ 时，分析器选用产生式 $B \rightarrow b$ 将栈顶 b 归约成 B 。

现在我们把状态 I_4 和 I_7 合并成 I_{47}

$$I_{47} = \{ [B \rightarrow b \cdot, a/b/\$] \}$$

此时当栈顶为 b 时，在 I_{47} 状态下，不论遇 a, b 或 $\$$ 均将 b 归约为 B ，虽然未能及时发现错误，但输入下一个符号时就会发现错误，于是我们类似将以上同样状态合并使状态减少，变成LALR分析。

LALR分析表的构造

(2) 同心集的概念

1) 定义

如果除去搜索符以外，两个LR(1)项目集是相同的，则称为同心集。

如 I_4 与 I_7 ， I_3 与 I_6 ， I_8 与 I_9

2) 说明

① 同心集合并后，其转换函数 $GO[I, X]$ 可通过自身合成而得到

② 同心集合并后不会存在“移进—归约”冲突，但存在“归约—归约”冲突，因为移进和归约不同心，所以不会出现“移进—归约”冲突。

例如文法

$S' \rightarrow S$

$S \rightarrow aAd \mid bBd \mid aBe \mid bAe$

$A \rightarrow c$

$B \rightarrow c$

该文法一共产生四个句子acd,bcd,ace,bce

它的LR(1)分析表不会出现冲突，在

LR(1)项目集中

$\{ [A \rightarrow c \cdot, d], [B \rightarrow c \cdot, e] \}$

$\{ [A \rightarrow c \cdot, e], [B \rightarrow c \cdot, d] \}$ ，它们均不含有冲突且为同心的，将它们合并，则 $\{ [A \rightarrow c \cdot, d/e], [B \rightarrow c \cdot, d/e] \}$ 就产生“归约--归约”冲突，因为面对d或e时，不知道应该用 $A \rightarrow c$ 还是用 $B \rightarrow c$ 归约。

LALR分析表的构造

(3) LALR分析表构造算法

基本思想：首先构造LR（1）项目集，如果它不存在冲突，就把同心集合并在一起，若合并后项目集规范族不存在“归约---归约”冲突，就按照这个集族构造分析表，其步骤如下：

- （1）构造文法G的LR（1）的项目集族 $C = \{I_0, I_1, \dots, I_n\}$ 。
- （2）把全部同心集合并在一起，记为 $C' = \{J_0, J_1, \dots, J_m\}$ 为新的项目集族，其中含有项目 $[S' \rightarrow \cdot S, \$]$ 的 J_i 为分析表的初态。

LALR分析表的构造

(3) LALR分析表构造算法

(3) 从 C' 构造 ACTION 表:

- ①若 $[A \rightarrow \alpha \cdot a \beta, b] \in J_i$, 且 $GO(J_i, a) = J_j$, $a \in V_T$, 则置 $ACTION[i, a] = S_j$ 。
- ②若 $[A \rightarrow \alpha \cdot, a] \in J_i$, 则置 $ACTION[i, a] = r_j$, 其中假定 $A \rightarrow \alpha$ 是文法第 j 个产生式。
- ③ 若 $[S' \rightarrow S \cdot \$] \in J_i$, 则置 $ACTION[i, \$] = acc$ 。

(4) 构造 GOTO 表:

假定 $J_i = I_{i1} \cup I_{i2} \cup \dots \cup I_{it}$, 则 $GO(I_{i1}, X)$, $GO(I_{i2}, X)$, ..., $GO(I_{it}, X)$ 也是同心集, 令 J_j 是它们合并集, 则 $GO(J_i, X) = J_j$ 。所以, 若 $GO(J_i, A) = J_j, A \in V_N$, 则置 $GOTO[i, A] = j$ 。

(5) 分析表中凡不能用(3)和(4)填入信息空白格, 均代表出错标志。

LALR分析表的构造

例如： I_3 和 I_6 是同心集，令 $J_i = I_3 \cup I_6 = J_{36}$

$GO(I_3, B) = I_8$ $GO(I_6, B) = I_9$ 也是同心集，记为 J_j

$J_j = I_8 \cup I_9 = J_{89}$ ， $GO(J_i, B) = GO(J_{36}, B) = J_{89}$

所以，置 $GOTO(36, B) = 89$

根据LALR分析表构造方法，可得LALR分析表如下表所示

状态	ACTION			GOTO	
	a	b	\$	S	B
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

LALR分析表的构造

经上述步骤构造出的分析表若不存在冲突，则称它为LALR分析表，利用LALR分析表的LR分析器称为LALR分析器，能构成LALR分析表的文法称为LALR（1）文法。

当输入串正确时，不论是LR分析器，还是LALR分析器，都给出了同样的“移进---归约”的序列，所差别只是状态名不同而已。

但是当输入串不符合文法时，LALR可能比LR多做了一些不必要归约，但LALR和LR均能指出输入串出错位置。

例如对aab \$ 的LR分析过程可看出，在状态 4 遇 \$ 报告错误

步骤	状态栈	符号栈	输入串	分析动作	下一状态
1	0	\$	aab \$	S ₃	3
2	03	\$ a	ab \$	S ₃	3
3	033	\$ aa	b \$	S ₄	4
4	0334	\$ aab	\$	报错	

LALR分析表的构造

由对aab \$ 的LALR分析过程可看出，栈中47遇\$要归约为B，在状态2面临\$时才报错。
这说明 LALR在LR已发现错误之后，还继续执行一些多余的产生式后才发现错误。

步骤	状态栈	符号栈	输入串	分析动作	下一状态
1	0	\$	aab \$	S ₃₆	36
2	0 <u>36</u>	\$ a	ab \$	S ₃₆	36
3	0 <u>36</u> <u>36</u>	\$ aa	b \$	S ₄₇	47
4	0 <u>36</u> <u>36</u> <u>47</u>	\$ aab	\$	r ₃	GOTO[36, B]=89
5	0 <u>36</u> <u>36</u> <u>89</u>	\$ aaB	\$	r ₂	GOTO[36, B]=89
6	0 <u>36</u> <u>89</u>	\$ aB	\$	r ₂	GOTO[0, B]=2
7	02	\$ B	\$	报错	

LALR分析表的构造

构造LALR（1）分析表方法：首先构造完整的LR（1）项目集族，然后依据它再构造LALR（1）分析表。

值得一提的是，不管文法是LR（0），SLR（1），LALR（1），还是LR（1），它们语法分析算法基本上都是相同的，不同之处仅在于分析表的构造一个比一个复杂，但适用的文法一个比一个更广泛。

LR分析表的压缩

LALR分析表规模

- 对于程序设计语言的翻译，其文法规模大致为50—100个终结符，100个产生式
- 得到的LALR分析表规模大致为几百个状态，20000个action函数项

action表的压缩

[状态, 终结符]——二维数组

很多行（列表）是相同的

状态→指针数组，指向一维数组（行）

行→列表——每个状态一个列表

- (终结符, 动作), 发生频率——位置
- any—列表中未列出的任何其他终结符

例题：

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			

例题（续）

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			

状态0， 4， 6， 7的action行相同

- 列表：(id, s5), ((, s4), (any, error)

状态1：(+, s6), (\$, acc), (any, error)

8：(+, s6), (), s11), (any, error)

2：error→r2， 错误推迟， (*, s7), (any, r2)

9：(*, s7), (any, r1)

3：error→r4， (any, r4)， 5、 10、 11类似

goto表的压缩

与action表不同，按列压缩，表的每列用列表存储，即每个非终结符一个列表

- 若非终结符A的列表的某个表项为：

(current_state, next_state)

表示在goto表中

$\text{goto}[\text{current_state}, A] = \text{next_state}$

goto表的压缩方法的设计思路

每列表项（状态）很少

$\text{goto}(I_i, A) = I_j$, I_j 某些项目中, A 紧挨在 \cdot 左边

状态 j 出现在第 i 行第 A 列

不存在项目集, 对 $X \neq Y$, 既有项目 X 在 \cdot 左边, 也有项目 Y 在 \cdot 左边

➔每个状态只在一列中出现

正常项替换error项

例题

F: (7, 10), (any, 3)

T: (6, 9), (any, 2)

E: (4, 8), (any, 1)

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			

使用二义性文法

不适合LR，冲突：移进/归约, 归约/归约
某些情况下是有用的

- 表达式，更简洁、更自然
- 区分普通语法结构和特殊优化情况

特殊方法使二义性文法可以使用
文法二义性，语言描述无歧义

- 消除歧义→唯一语法树

LR分析器错误处理

分析表中空位

LR分析器一发现不能继续，就报告错误

规范LR分析器在错误发生后，立即报告

SLR和LALR分析器可能会进行若干归约后报告，但不会继续移进符号

Panic模式的实现

向下扫描栈，对某个特定非终结符A，找到一个状态s，对于A有goto函数

丢弃输入符号，直至遇到 $a \in \text{FOLLOW}(A)$

将goto[s, A]压栈，继续

A：表示语法结构，表达式、语句...

含义：

- 某个语法结构A分析了一部分，遇到错误
- 丢弃已分析部分（栈），未分析部分（输入）
- 假装已找到A的一个实例，继续分析

短语级模式的实现

对分析表每个错误项，根据语言使用特性分析错误原因，设计恢复函数

特点

- 容易实现，无需考虑错误归约
- 分析表空项填入错误处理函数即可
- 错误处理函数插入、删除、替换栈和输入
- 避免无限循环

例题: $E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$

状态	action						goto
	id	+	*	()	\$	E
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	e1	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

例题（续）

- e1: 状态0, 2, 4, 5期待运算数，但遇到+、*或\$
压栈id和状态3（goto[0/2/4/5, id]），
输出错误信息“缺少运算数”
- e2: 状态0, 2, 4, 5遇到 ‘)’
删除 ‘)’, 输出错误信息“不匹配的右括号”
- e3: 状态1, 6期待运算符，但遇到id或(
压栈+和状态4，输出错误信息“缺少运算符”
- e4: 状态6期待运算符或)，但遇到\$
压栈)和状态9，输出错误信息“缺少)”

STACK	INPUT	Remark
\$0	id +)\$	
\$0 id 3	+)\$	
\$0 E 1	+)\$	
\$0 E 1 + 4)\$	
\$0 E 1 + 4	\$	e2:“未匹配右括号” 将其删除
\$0 E 1 + 4 id 3	\$	e1:“缺少运算数” 压栈id和状态3
\$0 E 1 + 4 E 7	\$	
\$0 E 1	\$	

语法分析器

- 上下文无关文法
- 自顶向下分析方法：
 - 递归实现
 - 适合手动构造
- 自底向上分析方法：
 - LR分析方法(SLR、规范LR、LALR)
 - 适合自动构造（Yacc采用LALR）