



中国科学技术大学  
University of Science and Technology of China

# 语 法 分 析

《编译原理和技术》

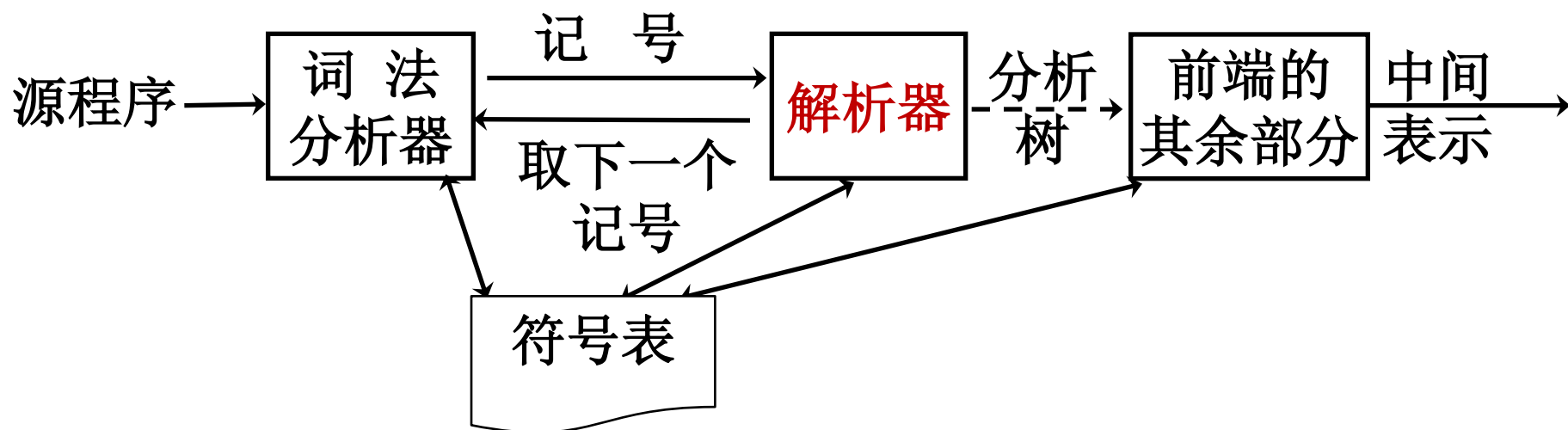
张昱

0551-63603804, [yuzhang@ustc.edu.cn](mailto:yuzhang@ustc.edu.cn)

中国科学技术大学  
计算机科学与技术学院



# 本章内容



- 语法的形式描述：上下文无关文法
- 语法分析：自上而下、自下而上
- 语法分析器(parser、syntax analyzer)的自动生成
  - LL(k)、LL(\*)、SLR、LR(k)、LALR



## 3.1 上下文无关文法

- 正规式的表达能力
- 上下文无关文法
  - 定义、推导、二义性
  - 名词：语言、文法等价、句型、句子



# 正规式的表达能力不足

## □ 正规式的表达能力

- 定义一些简单的语言，能表示给定结构的固定次数的重复或者没有指定次数的重复

例：  $a(ba)^5$ ,  $a(ba)^*$

- 不能用于描述配对或嵌套的结构

例1：配对括号串的集合，如不能表达  $(^n)^n, n \geq 1$

例2：  $\{w^c w \mid w \text{ 是 } a \text{ 和 } b \text{ 的串}\}$

原因：  $n$  不固定，且后面的串要依据前面不定长的串来确定；在有限的状态下不能表达



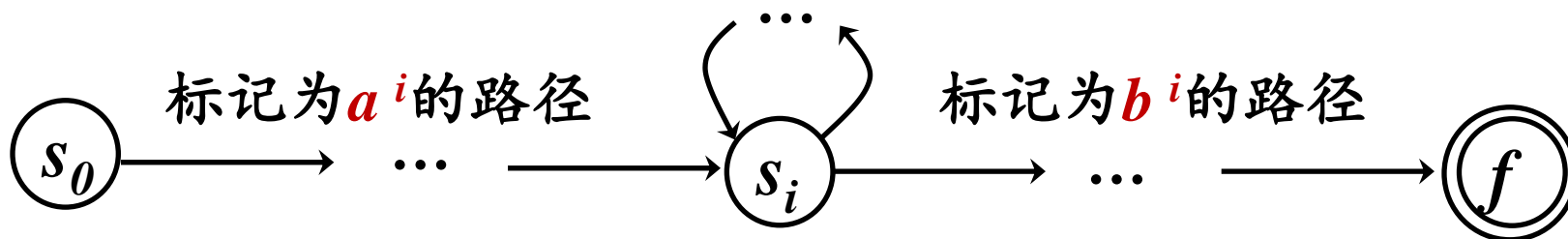
# 正规式的表达能力不足

例：  $L = \{a^n b^n \mid n \geq 1\}$ ， $L$  不能用正规式描述

## 反证法

- 若存在接受  $L$  的 DFA  $D$ ，状态数为  $k$  个（有限个）
- 设  $D$  读完  $\varepsilon, a, aa, \dots, a^k$  分别到达状态  $s_0, s_1, \dots, s_k$
- 至少有两个状态相同，例如是  $s_i$  和  $s_j$ ，则  $a^j b^i$  属于  $L$

标记为  $a^{j-i}$  的路径





# 上下文无关文法的定义

Context-free **Grammar** (CFG) 注: Syntax-语法

□ CFG是四元组 ( $V_T, V_N, S, P$ )

$V_T$  : 终结符(terminal, 记号token的第1元)集合

$V_N$  : 非终结符(nonterminal)集合

$S$  : 开始符号(start symbol), 是一个非终结符

$P$  : 产生式(production)集合

产生式的形式 :  $A \rightarrow \alpha$  , 有时用  $A ::= \alpha$

■ 例 (  $\{\text{id}, +, *, -, (, )\}, \{\text{expr}, \text{op}\}, \text{expr}, P$  )

$\text{expr} \rightarrow \text{expr op expr}$        $\text{expr} \rightarrow (\text{expr})$        $\text{expr} \rightarrow - \text{expr}$

$\text{expr} \rightarrow \text{id}$

$\text{op} \rightarrow +$

$\text{op} \rightarrow *$



# CFG的简化表示

## □ 表达式

### ■ 引入选择符 |

$expr \rightarrow expr\ op\ expr \mid (expr) \mid -expr \mid id$

$op \rightarrow + \mid *$

注：+, \*是 $op$ 的选择(alternatives)

### ■ 简化名称

$E \rightarrow E\ A\ E \mid (E) \mid -E \mid id$

$A \rightarrow + \mid *$



# 推导(derivation)

## □ 推导

把产生式看成重写规则，把符号串中的非终结符用其产生式右部的串来代替

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

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

上述代换序列称为从  $E$  到  $-(\text{id}+\text{id})$  的推导

$-(\text{id}+\text{id})$  是  $E$  的实例

## 记法

0步或多步推导  $S \Rightarrow^* \alpha$ 、一步或多步推导  $S \Rightarrow^+ w$





# 语言、文法、句型、句子

## □ 上下文无关语言

- 由上下文无关文法 $G$ 产生的语言：从**开始符号 $S$** 出发，经 $\Rightarrow^+$ 推导所能到达的**所有仅由终结符组成的串**
- **句型(sentential form)**:  $S \Rightarrow^* \alpha$ ,  $S$ 是开始符号,  $\alpha$ 是由**终结符和/或非终结符**组成的串, 则 $\alpha$ 是文法 $G$ 的句型
- **句子(sentence)**: 仅由**终结符**组成的句型

## □ 等价的文法

- 它们产生同样的语言



# 最左推导与最右推导

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

## □ 最左推导(leftmost derivation)

每步代换**最左边**的非终结符

$$\begin{aligned} E &\Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E + E) \\ &\Rightarrow_{lm} -(\text{id} + E) \Rightarrow_{lm} -(\text{id} + \text{id}) \end{aligned}$$

## □ 最右推导 (rightmost or canonical, 规范推导)

每步代换**最右边**的非终结符

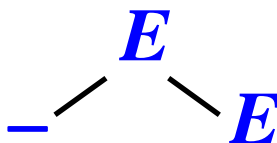
$$\begin{aligned} E &\Rightarrow_{rm} -E \Rightarrow_{rm} -(E) \Rightarrow_{rm} -(E + E) \\ &\Rightarrow_{rm} -(E + \text{id}) \Rightarrow_{rm} -(\text{id} + \text{id}) \end{aligned}$$



# 分析树(parse tree)

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

-(id+id)最左推导的分析树

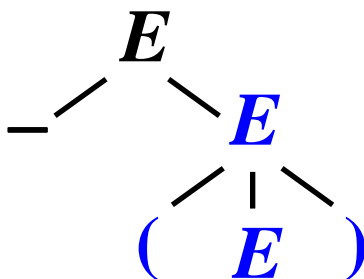




# 分析树(parse tree)

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

-(id+id)最左推导的分析树

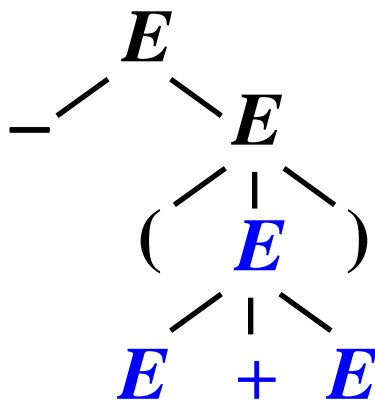




# 分析树(parse tree)

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

-(id+id)最左推导的分析树

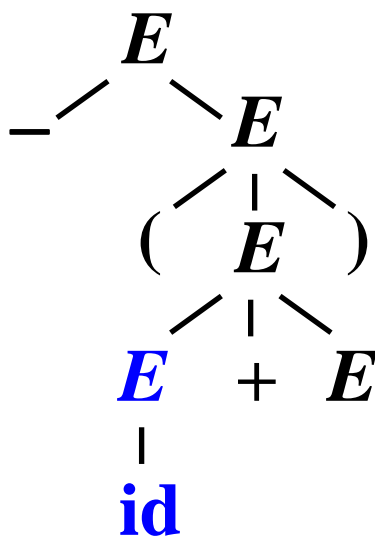




# 分析树(parse tree)

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

-(id+id)最左推导的分析树

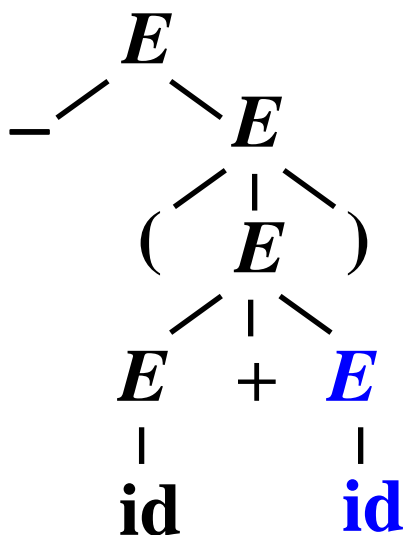




# 分析树(parse tree)

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

-(id+id)最左推导的分析树





# 文法的二义性

文法的某些句子存在不止一种最左(最右)推导, 或者不止一棵分析树, 则该文法是**二义**的。

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

**id\*id+id** 有两个不同的最左推导

$$E \Rightarrow E * E$$

$$\Rightarrow \text{id} * E$$

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

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

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

$$E \Rightarrow E + E$$

$$\Rightarrow E * E + E$$

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

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

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





# 文法的二义性

**id\*id+id** 有两棵不同的分析树

$E \Rightarrow E * E$

$\Rightarrow id * E$

$\Rightarrow id * E + E$

$\Rightarrow id * id + E$

$\Rightarrow id * id + id$

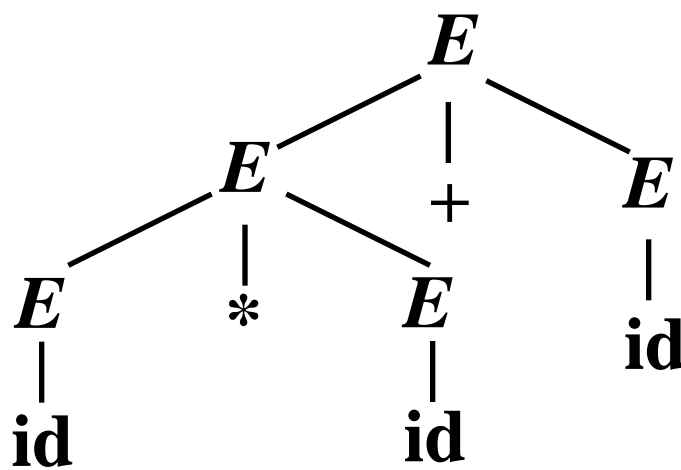
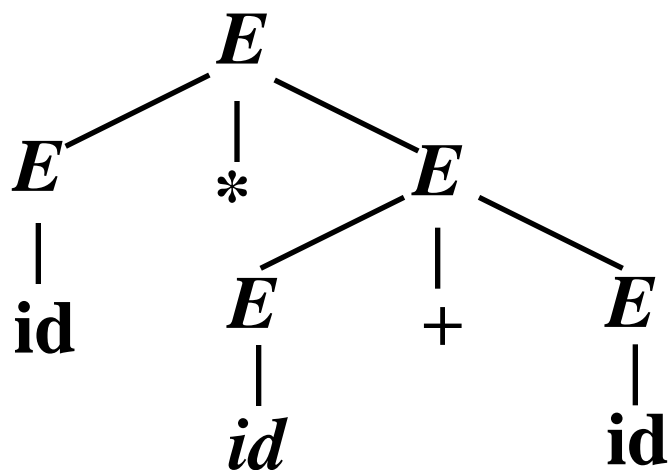
$E \Rightarrow E + E$

$\Rightarrow E * E + E$

$\Rightarrow id * E + E$

$\Rightarrow id * id + E$

$\Rightarrow id * id + id$





## 3.2 语言 and 文法

- 词法分析和语法分析的分离
- 语言和文法：验证、消除二义、消除左递归、提左因子

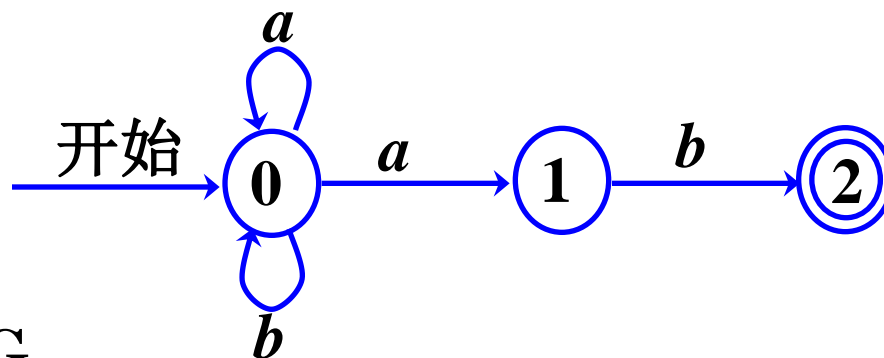


# 正规式和CFG的比较

- 都能表示语言
- 凡是能用正规式表示的语言，都能用CFG表示

- 正规式

$(a|b)^*ab$



- 上下文无关文法CFG

可机械地由NFA变换而得，为每个NFA状态引入一个非终结符，每条弧对应于产生式的一个分支（选项）

$A_0 \rightarrow a A_0 \mid b A_0 \mid a A_1$

$A_1 \rightarrow b A_2$

$A_2 \rightarrow \varepsilon$ （该产生式并不必要）



# 分离词法分析器的理由

## □ 为什么要用正规式定义词法

- 词法规则非常简单，不必用上下文无关文法
- 对于词法记号，正规式描述简洁且易于理解
- 从正规式构造出的词法分析器（DFA）效率高

## □ 分离词法分析和语法分析的好处（从软件工程看）

- 简化设计，便于编译器前端的模块划分
- 改进编译器的效率
- 增强编译器的可移植性，如输入字符集的特殊性等可以限制在词法分析器中处理



# 词法分析并入语法分析？

## □ 直接从字符流进行语法分析

- **文法复杂化**：文法中需有反映语言的注释和空白的规则
- **分析器复杂化**：处理包含注释和空白的分析器，比注释和空白符已被词法分析器过滤的分析器要复杂得多

## □ 分离但在同一遍（Pass）中进行

- 是通常编译器的做法



# 验证文法产生的语言

$G : S \rightarrow '(S)' S \mid \varepsilon \quad L(G) = \text{配对的括号串的集合}$

## □ 按推导步数进行归纳

按任意步推导，推出的是配对括号串

- 归纳基础(Basis):  $S \Rightarrow \varepsilon$
- 归纳 (Induction)假设: 少于 $n$ 步的推导都产生配对的括号串, 如  $S \Rightarrow^* x, S \Rightarrow^* y$
- 归纳步骤:  $n$ 步的最左推导如下:

$$S \Rightarrow '(S)' S \Rightarrow^* '(x)' S \Rightarrow^* '(x)' y$$



# 验证文法产生的语言

$G : S \rightarrow '(S)' S \mid \varepsilon \quad L(G) = \text{配对的括号串的集合}$

## □ 按串长进行归纳

任意长度的配对括号串均可由  $S$  推出

- 归纳基础(Basis):  $S \Rightarrow \varepsilon$
- 归纳 (Induction)假设: 长度小于  $2n$  的配对的括号串都可以从  $S$  推导出来
- 归纳步骤: 考虑长度为  $2n(n \geq 1)$  的  $w = '(x)'y$

$$S \Rightarrow '(S)' S \Rightarrow^* '(x)' S \Rightarrow^* '(x)' y$$



# 表达式的另一种文法

## □ 用一种层次的观点看待表达式

id \* id \* (id+id) + id \* id + id

左递归文法  
+ 是自左向右结合

## □ 无二义的文法

$expr \rightarrow \textcolor{red}{expr} + term \mid term$

$term \rightarrow \textcolor{red}{term} * factor \mid factor$

$factor \rightarrow id \mid (expr)$

如果改成

$expr \rightarrow term + \textcolor{red}{expr} \mid term$

呢?

+ 是自右向左结合





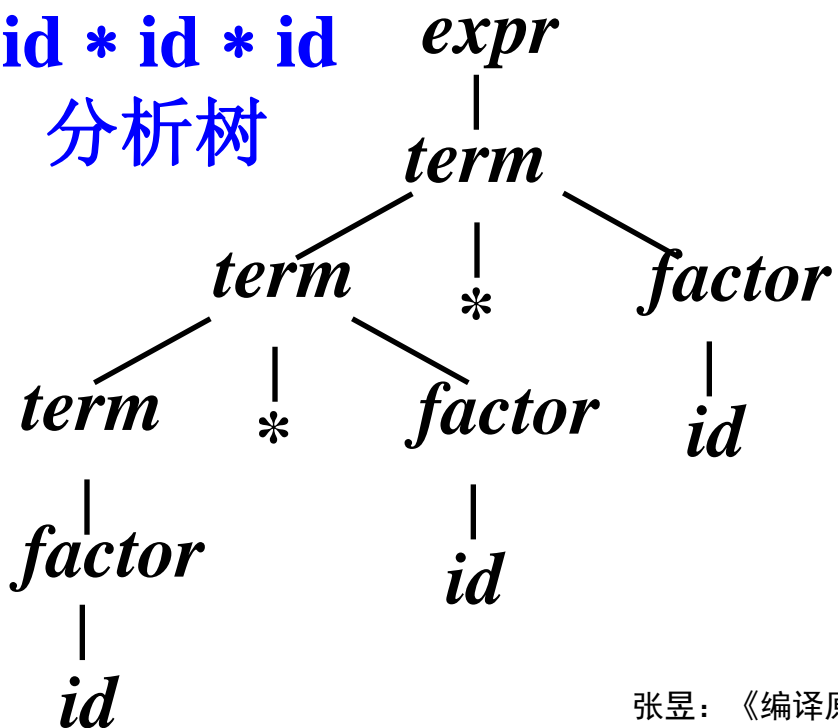
# 表达式的另一种文法

$expr \rightarrow expr + term \mid term$

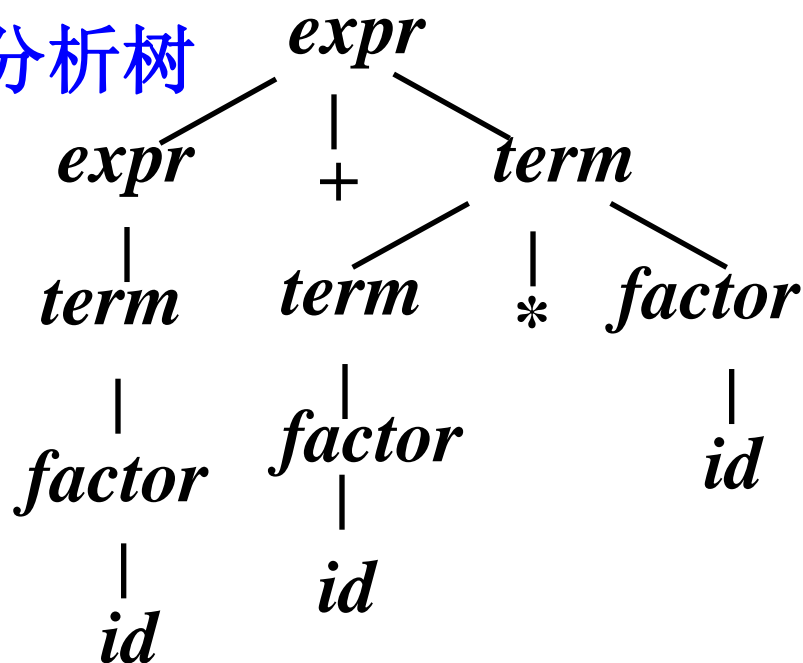
$term \rightarrow term * factor \mid factor$

$factor \rightarrow id \mid (expr)$

**id \* id \* id**  
**分析树**



**id + id \* id**  
**分析树**





# 消除二义性(Eliminating ambiguity)

$stmt \rightarrow$  if  $expr$  then  $stmt$   
          | if  $expr$  then  $stmt$  else  $stmt$   
          | other

□ 句型: if  $expr$  then if  $expr$  then  $stmt$  else  $stmt$

有两个最左推导:

$stmt \Rightarrow$  if  $expr$  then  $stmt$   
           $\Rightarrow$  if  $expr$  then if  $expr$  then  $stmt$  else  $stmt$

$stmt \Rightarrow$  if  $expr$  then  $stmt$  else  $stmt$   
           $\Rightarrow$  if  $expr$  then if  $expr$  then  $stmt$  else  $stmt$



# 消除二义性

## □ 无二义的文法

else 的就近匹配规则

$$\begin{aligned} stmt &\rightarrow matched\_stmt \\ &\quad | unmatched\_stmt \end{aligned}$$
$$\begin{aligned} matched\_stmt &\rightarrow \text{if } expr \text{ then } matched\_stmt \\ &\quad \text{else } matched\_stmt \\ &\quad | other \end{aligned}$$
$$\begin{aligned} unmatched\_stmt &\rightarrow \text{if } expr \text{ then } stmt \\ &\quad | \text{if } expr \text{ then } matched\_stmt \\ &\quad \quad \text{else } unmatched\_stmt \end{aligned}$$



# 消除左递归(Eliminating left recursion)

- 文法左递归  $A \Rightarrow^+ A \alpha$ 
  - 自上而下的分析不能用于左递归文法
- 直接左递归 (immediate left recursion)  
 $A \rightarrow A \alpha \mid \beta, \beta \text{ 不以 } A \text{ 开头}$ 
  - 串的特点  $\beta \alpha \dots \alpha$
- 消除直接左递归  $A \rightarrow A \alpha \mid \beta$   
 $A \rightarrow \beta A'$   
 $A' \rightarrow \alpha A' \mid \varepsilon$



# 消除左递归

## 例 算术表达文法

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

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

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

$$( T + T \dots + T )$$

$$( F * F \dots * F )$$

## 消除左递归后文法

$$E \rightarrow TE'$$

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

$$T \rightarrow FT'$$

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

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



# 消除非直接左递归

## □ 间接左递归

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Sd \mid \varepsilon$$

## □ 隐藏左递归

$$A \rightarrow \textcolor{red}{B}A$$

$$\textcolor{red}{B} \rightarrow \varepsilon$$

## □ 先变换成直接左递归

$$S \rightarrow Aa \mid b$$

$$A \rightarrow Aad \mid bd \mid \varepsilon$$

## □ 再消除左递归

$$S \rightarrow Aa \mid b$$

$$A \rightarrow bd A' \mid A'$$

$$A' \rightarrow adA' \mid \varepsilon$$



# 提左因子(left factoring)

- 有左因子的(left -factored)文法:  $A \rightarrow \alpha\beta_1 / \alpha\beta_2$ 
  - 自上而下分析时, 不清楚应该用A的哪个选择来代换
- 提左因子

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 / \beta_2$$

## 例 悬空else的文法

$$\begin{aligned} stmt &\rightarrow \text{if } expr \text{ then } stmt \text{ else } stmt \\ &\quad | \text{if } expr \text{ then } stmt \quad | \text{other} \end{aligned}$$

## 提左因子

$$\begin{aligned} stmt &\rightarrow \text{if } expr \text{ then } stmt \text{ optional\_else\_part} \quad | \text{other} \\ \text{optional\_else\_part} &\rightarrow \text{else } stmt \quad | \varepsilon \end{aligned}$$



# 非上下文无关的语言构造

$$L_1 = \{w c w \mid w \text{ 属于 } (a / b)^*\}$$

用来抽象：标识符的声明应先于其引用

C、Java都不是上下文无关语言

$$L_2 = \{a^n b^m c^n d^m \mid n \geq 0, m \geq 0\}$$

用来抽象：形参个数和实参个数应该相同

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

用来抽象：早先排版描述的一个现象

b e g i n: 5个字母键，5个回退键，5个下划线键





# 形似的上下文无关语言

$wcw$

$$L_1' = \{wcw^R \mid w \in (a/b)^*\}$$

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

$a^n b^m c^n d^m$

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

$$S \rightarrow aSd \mid aAd$$

$$A \rightarrow bAc \mid bc$$

$a^n b^n c^n$

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

$$S \rightarrow AB$$

$$A \rightarrow aAb \mid ab$$

$$B \rightarrow cBd \mid cd$$



# 形式语言鸟瞰

文法  $G = (V_T, V_N, S, P)$

□ 0型文法:  $\alpha \rightarrow \beta$ ,  $\alpha, \beta \in (V_N \cup V_T)^*$ ,  $|\alpha| \geq 1$

短语文法

□ 1型文法:  $|\alpha| \leq |\beta|$ , 但  $S \rightarrow \varepsilon$  可以例外

上下文有关文法

□ 2型文法:  $A \rightarrow \beta$ ,  $A \in V_N$ ,  $\beta \in (V_N \cup V_T)^*$

上下文无关文法

□ 3型文法:  $A \rightarrow aB$  或  $A \rightarrow a$ ,  $A, B \in V_N$ ,  $a \in V_T$

正规文法



# 上下文有关文法

例  $L_3 = \{a^n b^n c^n \mid n \geq 1\}$  的上下文有关文法

$S \rightarrow aSBC$        $S \rightarrow aBC$        $CB \rightarrow BC$        $aB \rightarrow ab$

$bB \rightarrow bb$        $bC \rightarrow bc$        $cC \rightarrow cc$

$a^n b^n c^n$  的推导过程如下：

$S \Rightarrow^* a^{n-1} S (BC)^{n-1}$       用  $S \rightarrow aSBC$   $n-1$  次

$S \Rightarrow^+ a^n (BC)^n$       用  $S \rightarrow aBC$  1 次

$S \Rightarrow^+ a^n B^n C^n$       用  $CB \rightarrow BC$  交换相邻的  $CB$

$S \Rightarrow^+ a^n b B^{n-1} C^n$       用  $aB \rightarrow ab$  1 次

$S \Rightarrow^+ a^n b^n C^n$       用  $bB \rightarrow bb$   $n-1$  次

$S \Rightarrow^+ a^n b^n c C^{n-1}$       用  $bC \rightarrow bc$  1 次

$S \Rightarrow^+ a^n b^n c^n$       用  $cC \rightarrow cc$   $n-1$  次



# 例题1 写等价的非二义文法

下面的二义文法描述命题演算公式的语法，  
为它写一个等价的非二义文法

$$S \rightarrow S \text{ and } S \mid S \text{ or } S \mid \text{not } S \mid p \mid q \mid '(' S ')'$$

解答

非二义文法的产生式如下：

$$E \rightarrow E \text{ or } T \mid T$$

$$T \rightarrow T \text{ and } F \mid F$$

$$F \rightarrow \text{not } F \mid '(' E ') ' \mid p \mid q$$



# 例题1 写等价的非二义文法

下面的二义文法描述命题演算公式的语法，  
为它写一个等价的非二义文法

$$S \rightarrow S \text{ and } S \mid S \text{ or } S \mid \text{not } S \mid p \mid q \mid '(' S ')'$$

解答

非二义文法的产生式如下：

$$E \rightarrow E \text{ or } T \mid T$$

$$T \rightarrow T \text{ and } F \mid F$$

$$F \rightarrow \text{not } \cancel{E} \mid '(' E ') ' \mid p \mid q \quad ?$$

not p and q  
not p and q

not p and q有两种不同的最左推导



## 例题2 写等价的不同文法

设计一个文法：字母表 $\{a, b\}$ 上  $a$  和  $b$  的个数相等的所有串的集合

□ 二义文法： $S \rightarrow a S b S \mid b S a S \mid \varepsilon$   
 $aabbabab$   $aabbabab$

□ 二义文法： $S \rightarrow a B \mid b A \mid \varepsilon$   
 $A \rightarrow a S \mid b A A$   
 $B \rightarrow b S \mid a B B$   
 $aabbabab$   $aabbabab$   $aabbabab$

□ 非二义文法： $S \rightarrow a B S \mid b A S \mid \varepsilon$   
 $A \rightarrow a \mid b A A$   
 $B \rightarrow b \mid a B B$   
 $a abb abab$   
 $a B S$



## 3.3 自上而下分析

- 文法:  $LL(1)$ 、 $LL(k)$ 、 $ALL(*)$ 不支持左递归
- 分析器: 递归下降的预测分析器  
非递归的预测分析器 (预测分析表)
- 错误恢复



# 自上而下分析的一般方法

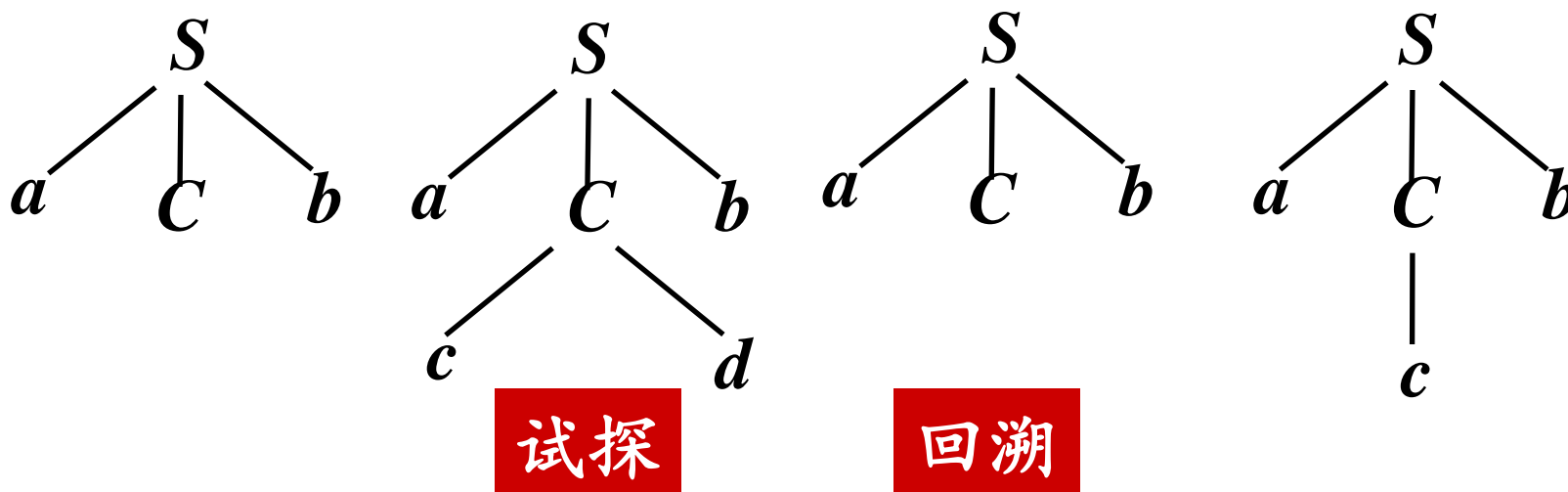
## □ 自上而下top-down分析

为输入串寻找**最左推导**：试探 - 回溯(效率低，代价高)

■ ANTLR: 引入带谓词的DFA使回溯不重新分析输入串

例 文法  $S \rightarrow aCb$   $C \rightarrow cd / c$

为输入串  $w = acb$  建立分析树







# 自上而下分析：左递归

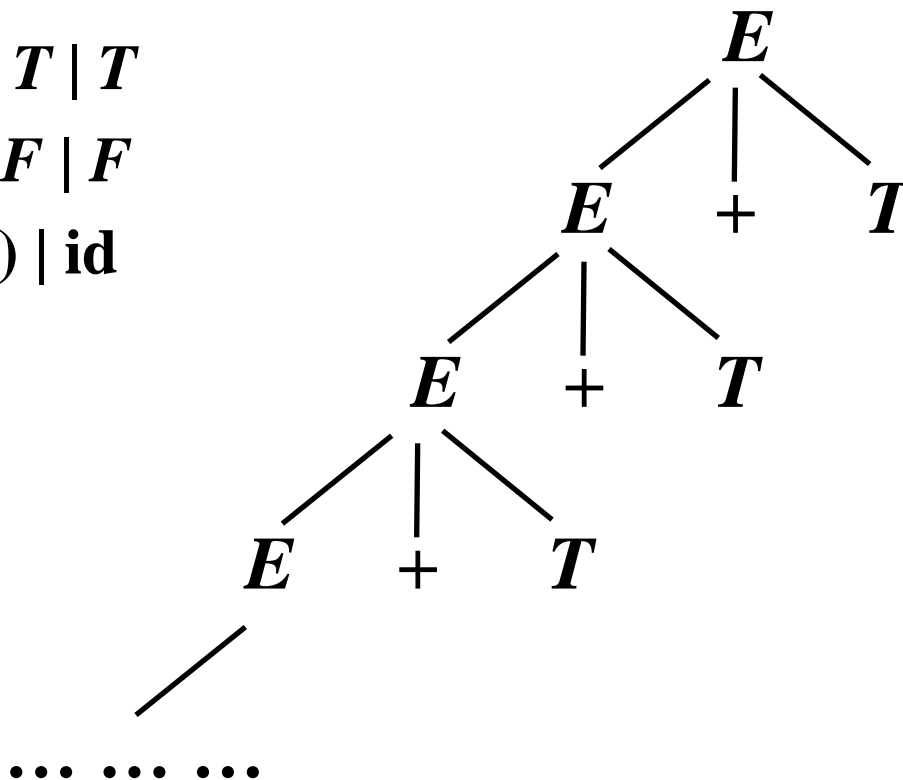
## □ 不能处理左递归文法

算术表达文法

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

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

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





# LL(1)文法

**L**-scanning from left to right; **L**- leftmost derivation

□ 对文法加什么样的限制可以保证没有回溯?

□ 先定义两个和文法有关的函数

■  $\text{FIRST}(\alpha) = \{a \mid \alpha \Rightarrow^* a..., a \in V_T\}$

特别地,  $\alpha \Rightarrow^* \varepsilon$  时, 规定  $\varepsilon \in \text{FIRST}(\alpha)$

■  $\text{FOLLOW}(A) = \{a \mid S \Rightarrow^* ...Aa..., a \in V_T\}$

如果  $A$  是某个句型的最右符号, 那么  $\$$  属于  $\text{FOLLOW}(A)$



# LL(1)文法：FIRST(X)

□ 计算FIRST(X),  $X \in V_T \cup V_N$

■  $X \in V_T$ ,  $\text{FIRST}(X) = \{X\}$

■  $X \in V_N$  且  $X \rightarrow Y_1 Y_2 \dots Y_k$

如果  $a \in \text{FIRST}(Y_i)$  且  $\varepsilon$  在  $\text{FIRST}(Y_1), \dots,$

$\text{FIRST}(Y_{i-1})$  中, 则将  $a$  加入到  $\text{FIRST}(X)$

如果  $\varepsilon$  在  $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_k)$  中, 则将  $\varepsilon$  加入到  $\text{FIRST}(X)$

■  $X \in V_N$  且  $X \rightarrow \varepsilon$

则将  $\varepsilon$  加入到  $\text{FIRST}(X)$



# LL(1)文法：FIRST, FOLLOW

- 计算 $\text{FIRST}(X_1 X_2 \dots X_n)$ ,  $X_i \in V_T \cup V_N$ , 它包含
  - $\text{FIRST}(X_1)$  中所有的非  $\varepsilon$  符号
  - $\text{FIRST}(X_i)$  中所有的非  $\varepsilon$  符号, 如果  $\varepsilon$  在  $\text{FIRST}(X_1), \dots, \text{FIRST}(X_{i-1})$  中
  - $\varepsilon$ , 如果  $\varepsilon$  在  $\text{FIRST}(X_1), \dots, \text{FIRST}(X_n)$  中
  
- 计算 $\text{FOLLOW}(A)$ ,  $A \in V_N$ 
  - $\$$  加入到  $\text{FOLLOW}(S)$  中
  - 如果  $A \rightarrow \alpha B \beta$ , 则  $\text{FIRST}(\beta)$  加入到  $\text{FOLLOW}(B)$
  - 如果  $A \rightarrow \alpha B$  或  $A \rightarrow \alpha B \beta$  且  $\varepsilon \in \text{FIRST}(\beta)$ , 则  $\text{FOLLOW}(A)$  的所有符号加入到  $\text{FOLLOW}(B)$



# LL(1)文法

## □ LL(1)文法的定义

任何两个产生式  $A \rightarrow \alpha / \beta$  都满足下列条件：

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
- 若  $\beta \Rightarrow^* \varepsilon$ ，那么  $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$

**例** 对于下面文法，面临  $a\dots$  时，第2步推导不知用  $A$  的哪个产生式选择

$S \rightarrow A B$

$A \rightarrow a b \mid \varepsilon$        $a \in \text{FIRST}(ab) \cap \text{FOLLOW}(A)$

$B \rightarrow a C$

$C \rightarrow \dots$



# LL(1)文法

## □ LL(1)文法的定义

任何两个产生式  $A \rightarrow \alpha / \beta$  都满足下列条件：

- $\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$
- 若  $\beta \Rightarrow^* \varepsilon$ ，那么  $\text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \emptyset$

## □ LL(1)文法有一些明显的性质

- 没有公共左因子
- 不是二义的
- 不含左递归



# 表达式文法：无左递归的

例

$$E \rightarrow T\mathbf{E'}$$

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

$$T \rightarrow FT'$$

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

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

$$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{ (, \text{id} \}$$

$$\text{FIRST}(E') = \{ +, \varepsilon \}$$

$$\text{FIRST}(T') = \{ *, \varepsilon \}$$

$$\text{FOLLOW}(E) = \mathbf{\text{FOLLOW}(E')} = \{ ), \$ \}$$

$$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{ \mathbf{+}, ), \$ \}$$

$$\text{FOLLOW}(F) = \{ +, \mathbf{*}, ), \$ \}$$



# 预测分析器

## □ 递归下降(recursive-descent)的预测分析

- 为每一个非终结符写一个分析过程
- 这些过程可能是递归的

### 例

*type*  $\rightarrow$  *simple*

|  $\uparrow$  id

| array [*simple*] of *type*

*simple*  $\rightarrow$  integer

| char

| num dotdot num





# 递归下降的预测分析器

*type*  $\rightarrow$  *simple* |  $\uparrow$  id | array [*simple*] of *type*  
*simple*  $\rightarrow$  integer | char | num dotdot num

```
void match (terminal t) {  
    if (lookahead == t) lookahead = nextToken( );  
    else error( );  
}  
  
void type( ) {  
    if ( (lookahead == integer) || (lookahead == char) || (lookahead == num) )  
        simple( );  
    else if ( lookahead == ' $\uparrow$ ' ) { match(' $\uparrow$ '); match(id); }  
    else if (lookahead == array) {  
        match(array); match( '[' ); simple( );  
        match( ']' ); match(of ); type( );  
    }  
    else error( );  
}
```



# 递归下降的预测分析器

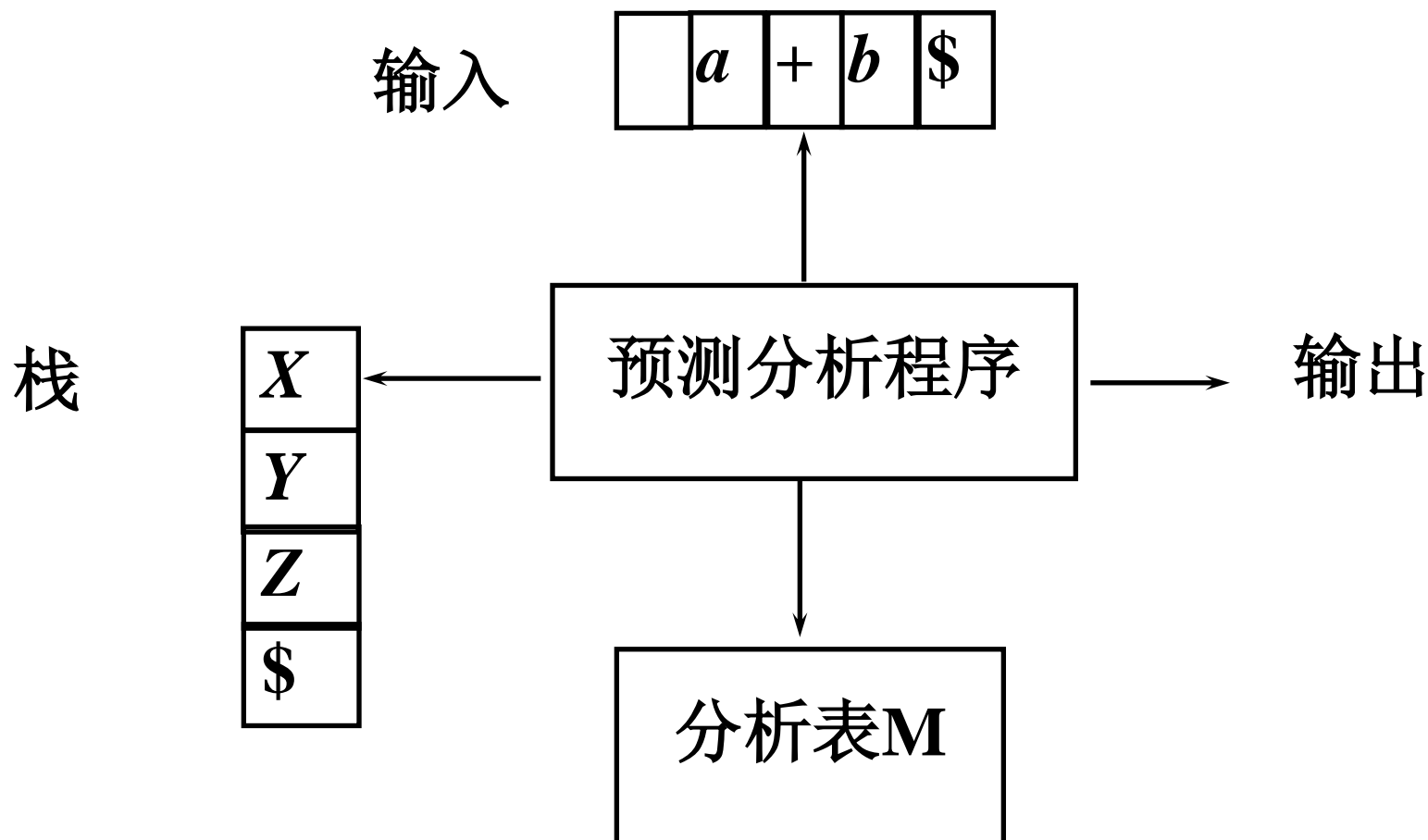
*type*  $\rightarrow$  *simple* |  $\uparrow$  id | array [*simple*] of *type*  
*simple*  $\rightarrow$  integer | char | num dotdot num

```
void simple( ) {  
    if ( lookahead == integer) match(integer);  
    else if (lookahead == char) match(char);  
    else if (lookahead == num) {  
        match(num); match(dotdot); match(num);  
    }  
    else error( );  
}
```



# 非递归的预测分析

## Nonrecursive Predictive Parsing





# 预测分析表

□ 行：非终结符；列：终结符 或\$；单元：产生式

□ 教材 表3.1 (P58)

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



# 预测分析举例

预测分析器接受输入  $id * id + id$  的前一部分动作

栈	输 入	输 出
$\$E$	$id * id + id\$$	



# 预测分析举例

预测分析器接受输入  $id * id + id$  的前一部分动作

栈	输 入	输 出
$\$E$	$id * id + id\$$	
$\$E'T$	$id * id + id\$$	$E \rightarrow TE'$



# 预测分析举例

预测分析器接受输入  $id * id + id$  的前一部分动作

栈	输 入	输 出
$\$E$	$id * id + id\$$	
$\$E 'T$	$id * id + id\$$	$E \rightarrow TE '$
$\$E 'T 'F$	$id * id + id\$$	$T \rightarrow FT '$



# 预测分析举例

预测分析器接受输入  $id * id + id$  的前一部分动作

栈	输 入	输 出
$\$E$	$id * id + id\$$	
$\$E 'T$	$id * id + id\$$	$E \rightarrow TE '$
$\$E 'T 'F$	$id * id + id\$$	$T \rightarrow FT '$
$\$E 'T ' id$	$id * id + id\$$	$F \rightarrow id$





# 预测分析举例

预测分析器接受输入  $id * id + id$  的前一部分动作

栈	输 入	输 出
$\$E$	$id * id + id\$$	
$\$E 'T$	$id * id + id\$$	$E \rightarrow TE '$
$\$E 'T 'F$	$id * id + id\$$	$T \rightarrow FT '$
$\$E 'T ' id$	$id * id + id\$$	$F \rightarrow id$
$\$E 'T '$	$* id + id\$$	



# 预测分析举例

预测分析器接受输入  $id * id + id$  的前一部分动作

栈	输 入	输 出
$\$E$	$id * id + id\$$	
$\$E 'T$	$id * id + id\$$	$E \rightarrow TE'$
$\$E 'T 'F$	$id * id + id\$$	$T \rightarrow FT'$
$\$E 'T 'id$	$id * id + id\$$	$F \rightarrow id$
$\$E 'T '$	$* id + id\$$	
$\$E 'T 'F *$	$* id + id\$$	$T' \rightarrow *FT'$



# 预测分析举例

预测分析器接受输入  $id * id + id$  的前一部分动作

栈	输 入	输 出
$\$E$	$id * id + id\$$	
$\$E 'T$	$id * id + id\$$	$E \rightarrow TE '$
$\$E 'T 'F$	$id * id + id\$$	$T \rightarrow FT '$
$\$E 'T ' id$	$id * id + id\$$	$F \rightarrow id$
$\$E 'T '$	$* id + id\$$	
$\$E 'T 'F *$	$* id + id\$$	$T' \rightarrow *FT '$
$\$E 'T 'F$	$id + id\$$	



# 预测分析举例

预测分析器接受输入  $id * id + id$  的前一部分动作

栈	输 入	输 出
$\$E$	$id * id + id\$$	
$\$E 'T$	$id * id + id\$$	$E \rightarrow TE '$
$\$E 'T 'F$	$id * id + id\$$	$T \rightarrow FT '$
$\$E 'T ' id$	$id * id + id\$$	$F \rightarrow id$
$\$E 'T '$	$* id + id\$$	
$\$E 'T 'F *$	$* id + id\$$	$T' \rightarrow *FT '$
$\$E 'T 'F$	$id + id\$$	
$\$E 'T ' id$	$id + id\$$	$F \rightarrow id$



# 预测分析表的构造

## □ predictive parsing table

行：非终结符；列：终结符 或\$；单元：产生式

$M[A, a]$  产生式  $A \rightarrow \alpha$  表示在面临  $a$  时，将栈顶符号  $A$  替换为  $\alpha$

## □ 构造方法

- (1) 对文法的每个产生式  $A \rightarrow \alpha$ ，执行(2)和(3)
- (2) 对  $\text{FIRST}(\alpha)$  的每个终结符  $a$ ，把  $A \rightarrow \alpha$  加入  $M[A, a]$
- (3) 如果  $\varepsilon$  在  $\text{FIRST}(\alpha)$  中，对  $\text{FOLLOW}(A)$  的每个终结符  $b$ （包括\$），把  $A \rightarrow \alpha$  加入  $M[A, b]$
- (4)  $M$  中其它没有定义的条目都是error



# 多重定义

例  $stmt \rightarrow \text{if } expr \text{ then } stmt \ e\_part \mid other$

$e\_part \rightarrow \text{else } stmt \mid \varepsilon$

$expr \rightarrow b$

非终结符	输入符号			
	other	$b$	else	...
$stmt$	$stmt \rightarrow other$			
$e\_part$			$e\_part \rightarrow$ else $stmt$ $e\_part \rightarrow \varepsilon$	
$expr$		$expr \rightarrow b$		

多重定义条目意味着文法左递归或者是二义的



# 多重定义的消除

例 删去 $e\_part \rightarrow \varepsilon$ ，这正好满足 else和最近的then配对

LL(1)文法  $\Leftrightarrow$  预测分析表无多重定义的条目

非终结符	输入符号			
	other	$b$	else	...
$stmt$	$stmt \rightarrow other$			
$e\_part$			$e\_part \rightarrow$ else $stmt$ <del><math>e\_part \rightarrow \varepsilon</math></del>	
$expr$		$expr \rightarrow b$		



# 预测分析的错误恢复

## □ 编译器的错误处理

- 词法错误，如标识符、关键字或算符的拼写错
- 语法错误，如算术表达式的括号不配对
- 语义错误，如算符作用于不相容的运算对象
- 逻辑错误，如无穷的递归调用

## □ 分析器对错误处理的基本目标

- 清楚而准确地报告错误的出现，并尽量少出现伪错误
- 迅速地从每个错误中恢复过来，以便诊断后面的错误
- 它不应该使正确程序的处理速度降低太多

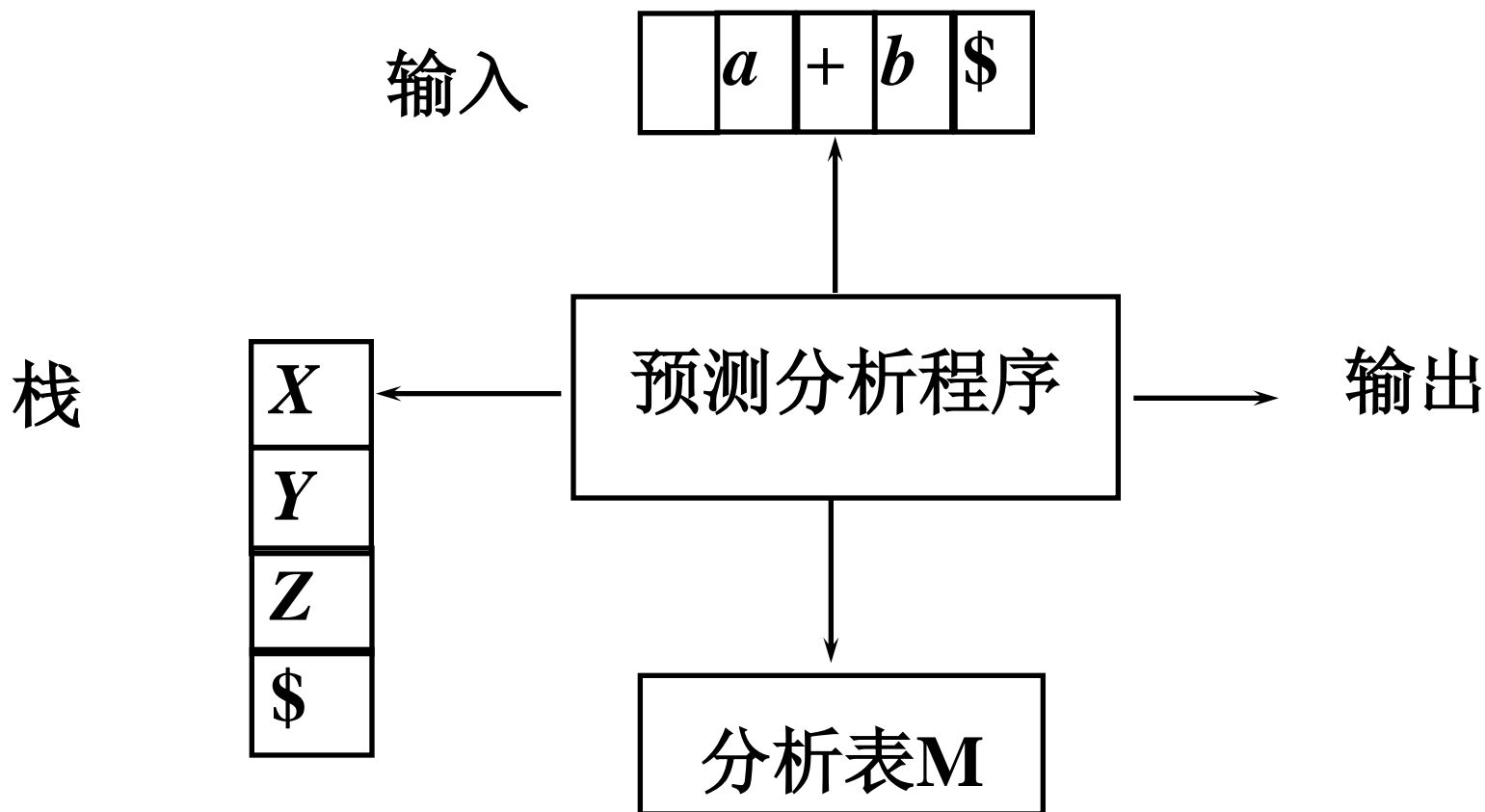




# 预测分析的错误恢复

## □ 非递归预测分析在什么场合下发现错误

- 栈顶的终结符和下一个输入符号不匹配

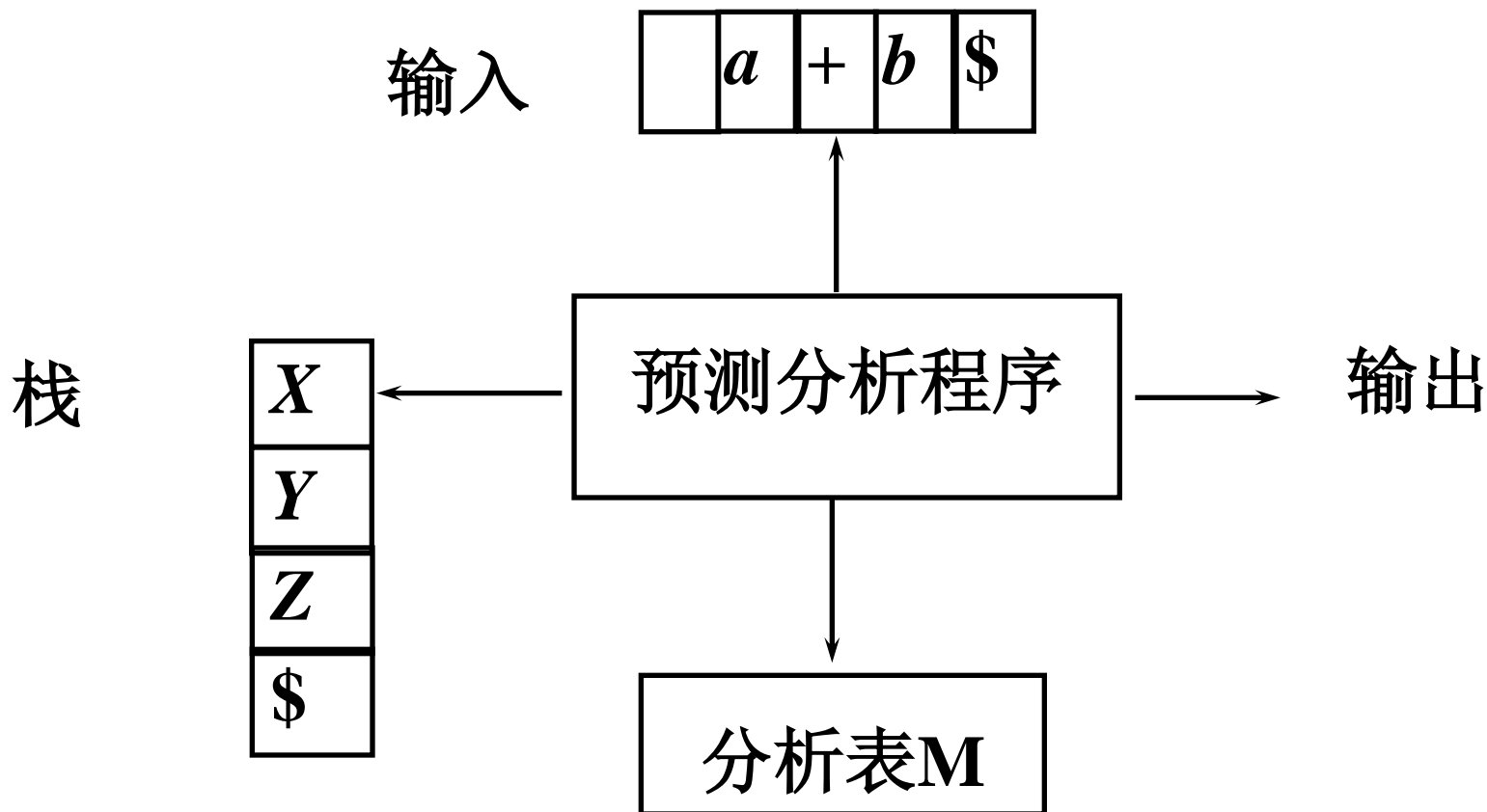




# 预测分析的错误恢复

## □ 非递归预测分析在什么场合下发现错误

- 栈顶是非终结符 $A$ ，输入符号是 $a$ ，而 $M[A, a]$ 是空白





# 预测分析的错误恢复

## □ 非递归预测分析

采用紧急方式(panic mode)的错误恢复

- 发现错误时，抛弃输入记号直到其属于某个指定的同步记号(synchronizing tokens)集合为止

## □ 同步(synchronizing)

- 同步：词法分析器当前提供的记号流能够构成的语法构造，正是语法分析器所期望的
- 不同步的例子  
语法分析器期望剩余的前缀构成过程调用语句，而实际剩余的前缀形成的是赋值语句



# 预测分析的错误恢复

## □ 同步记号集合的选择

- 把FOLLOW(A)的所有终结符放入非终结符A的同步记号集合

if *expr* then *stmt*

出错

(then和分号等记号是*expr*的同步记号)

- 把高层构造的开始符号加到低层构造的同步记号集中

*a* = *expr*; if ...

出错

同步记号

(语句的开始符号作为表达式的同步记号, 以免表达式出错又遗漏分号时忽略if语句等一大段程序)



# 预测分析的错误恢复

## □ 同步记号集合的选择

- 把FOLLOW(A)的所有终结符放入非终结符A的同步记号集合
- 把高层构造的开始符号加到低层构造的同步记号集中
- 把FIRST(A)的终结符加入A的同步记号集合

$a = \text{expr};$ ,  $\text{if} \dots$

出错

同步记号

(语句的开始符号作为语句的同步符号, 以免多出一个逗号时会把if语句忽略了)



# 预测分析的错误恢复

## □ 同步记号集合的选择

- 把FOLLOW(A)的所有终结符放入非终结符A的同步记号集合
- 把高层构造的开始符号加到低层构造的同步记号集中
- 把FIRST(A)的终结符加入A的同步记号集合
- 如果出错时栈顶是存在有产生空串选择的非终结符，则可以使用其推出空串的产生式选择



# 错误恢复举例

例 栈顶为  $T'$ ，面临  $id$  时出错

非终结符	输入符号			
	$id$	$+$	$*$	$\dots$
$E$	$E \rightarrow TE'$			
$E'$		$E' \rightarrow +TE'$		
$T$	$T \rightarrow FT'$			
$T'$	出错	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	
$\dots$				



# 错误恢复举例

例 栈顶为  $T'$ ，面临  $id$  时出错

非终结符	输入符号			
	$id$	$+$	$*$	$\dots$
$E$	$E \rightarrow TE'$			
$E'$		$E' \rightarrow +TE'$		
$T$	$T \rightarrow FT'$			
$T'$	出错 用 $T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	
$\dots$				





# 预测分析的错误恢复

## □ 同步记号集合的选择

- 把FOLLOW(A)的所有终结符放入非终结符A的同步记号集合
- 把高层构造的开始符号加到低层构造的同步记号集中
- 把FIRST(A)的终结符加入A的同步记号集合
- 如果出错时栈顶是存在有产生空串选择的非终结符，则可以使用其推出空串的产生式选择
- 如果终结符在栈顶而不能匹配，弹出此终结符



## 3.4 自下而上分析

(移进-归约分析)

- 归约(最右推导的逆过程)
- 句柄(可归约串),可能不唯一
- 冲突: 移进-归约、归约-归约



# 归约 ( Reduce )

把输入串归约成文法的开始符号，是最右推导的逆过程

例  $S \rightarrow aABe$   
 $A \rightarrow Abc / b$   
 $B \rightarrow d$



# 归约 ( Reduce )

把输入串归约成文法的开始符号，是最右推导的逆过程

例  $S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

$abbcde$  (读入 $ab$ )      寻找能匹配某产生式右部的子串

$a$        $b$



# 归约 ( Reduce )

把输入串归约成文法的开始符号，是最右推导的逆过程

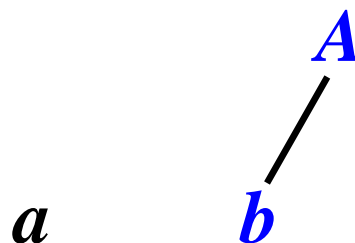
例  $S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

$abbcde$

$aAbcde$  (归约)





# 归约 ( Reduce )

把输入串归约成文法的开始符号，是最右推导的逆过程

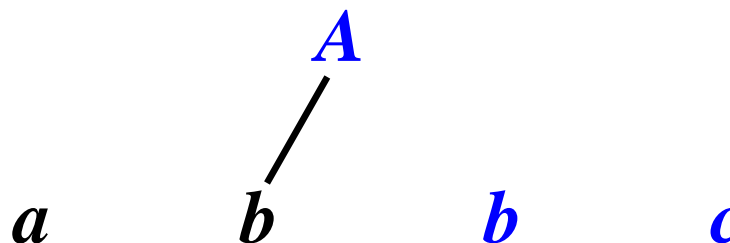
例  $S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

$abbcde$

$aAbcde$  (再读入 $bc$ )





# 归约 ( Reduce )

把输入串归约成文法的开始符号，是最右推导的逆过程

例  $S \rightarrow aABe$

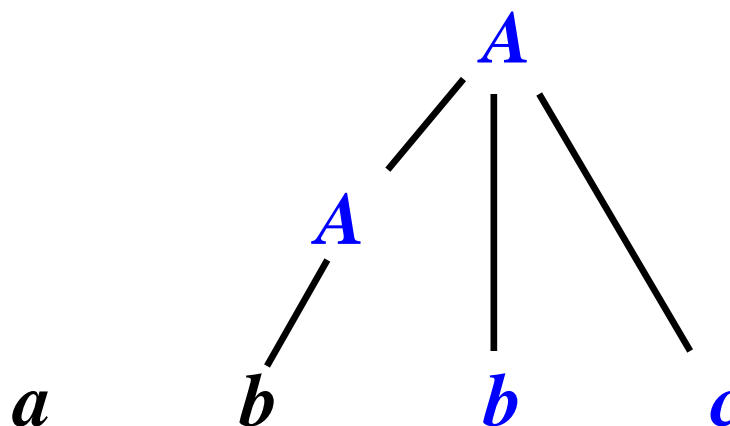
$A \rightarrow Abc / b$

$B \rightarrow d$

$abbcde$

$aAbcde$

$aAde$  (归约)





# 归约 ( Reduce )

把输入串归约成文法的开始符号，是最右推导的逆过程

例  $S \rightarrow aABe$

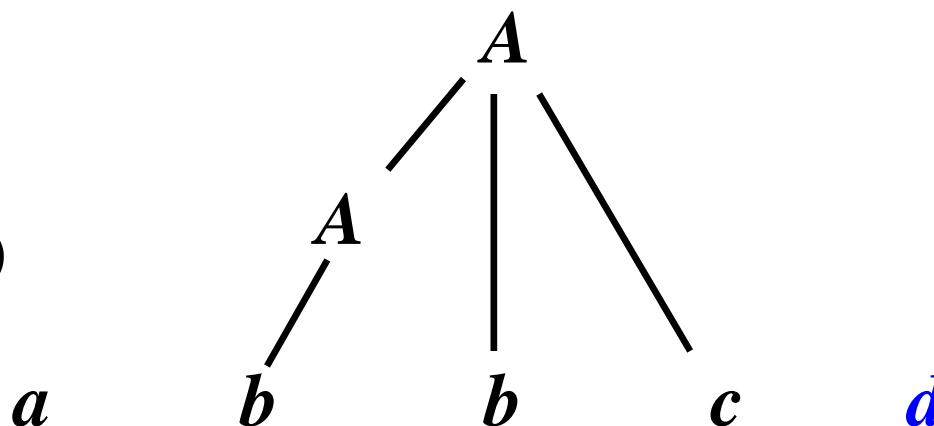
$A \rightarrow Abc / b$

$B \rightarrow d$

$abbcde$

$aAbcde$

$aAde$  (再读入  $d$ )







# 归约 ( Reduce )

把输入串归约成文法的开始符号，是最右推导的逆过程

例  $S \rightarrow aABe$

$A \rightarrow Abc / b$

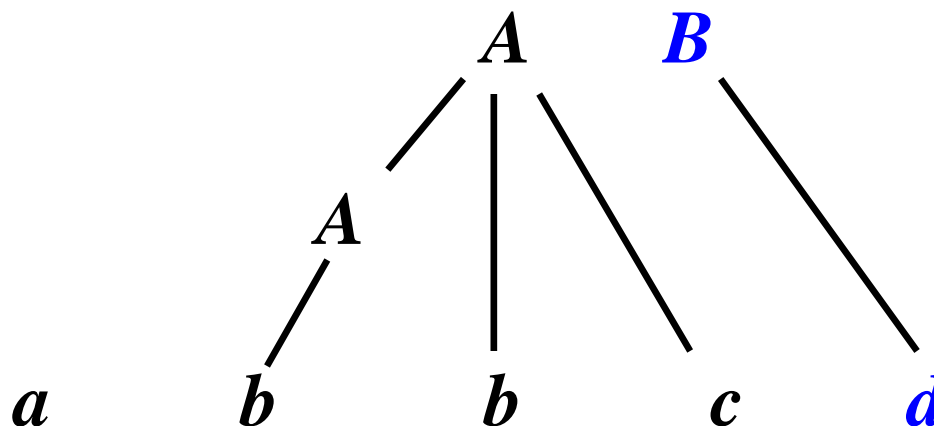
$B \rightarrow d$

$abbcde$

$aAbcde$

$aAde$

$aABe$  (归约)





# 归约 ( Reduce )

把输入串归约成文法的开始符号，是最右推导的逆过程

例  $S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

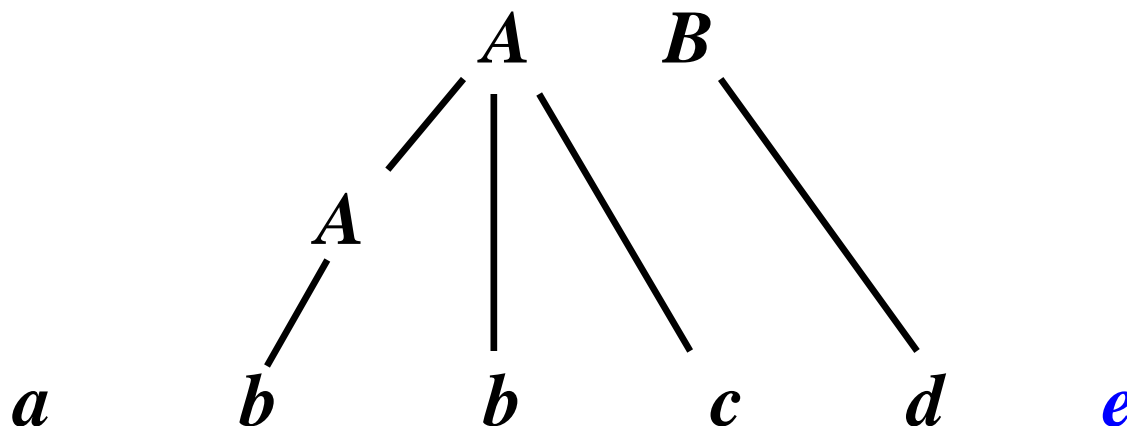
$abbcde$

$aAbcde$

$aAde$

$aABe$

$aABe$ (再读入 $e$ )





# 归约 ( Reduce )

把输入串归约成文法的开始符号，是最右推导的逆过程

例  $S \rightarrow aABe$

$A \rightarrow Abc / b$

$B \rightarrow d$

$abbcde$

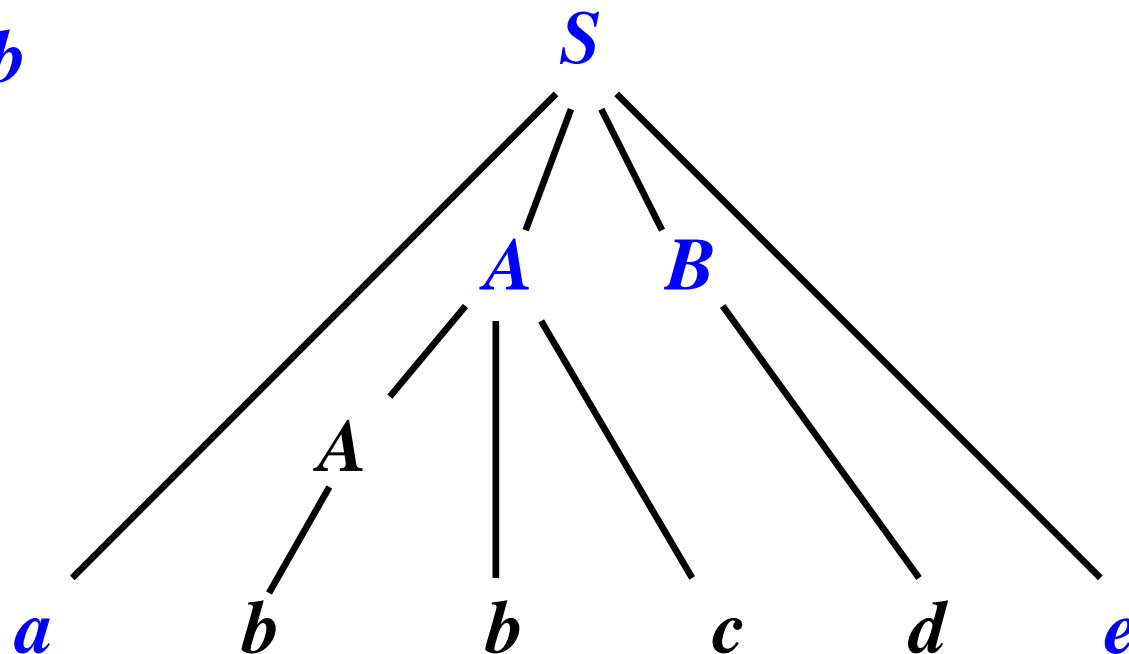
$aAbcde$

$aAde$

$aABe$

$aABe$

$S$  (归约)





# 句柄(handles)

## □ 句型的句柄（可归约串）

- 该句型中**和某产生式右部匹配**的子串, 并且
- 把它**归约**成该产生式左部的非终结符代表了**最右推导过程的逆过程的一步**

$$S \rightarrow aABe$$

$$A \rightarrow Abc / b$$

$$B \rightarrow d$$

$$S \Rightarrow_{rm} aABe \Rightarrow_{rm} aAde \Rightarrow_{rm} aAbcde \Rightarrow_{rm} abbcde$$

- 句柄的**右边仅含终结符**
- 如果文法二义, 那么句柄**可能不唯一**



# 例句柄不唯一

$$E \rightarrow E + E / E * E / (E) / \text{id}$$

$$\begin{aligned} E &\Rightarrow_{rm} E * E \\ &\Rightarrow_{rm} E * E + E \\ &\Rightarrow_{rm} E * E + \text{id}_3 \\ &\Rightarrow_{rm} E * \text{id}_2 + \text{id}_3 \\ &\Rightarrow_{rm} \text{id}_1 * \text{id}_2 + \text{id}_3 \end{aligned}$$



# 例句柄不唯一

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

$$E \Rightarrow_{rm} E * E$$

$$\Rightarrow_{rm} E * E + E$$

$$\Rightarrow_{rm} E * E + \text{id}_3$$

$$\Rightarrow_{rm} E * \text{id}_2 + \text{id}_3$$

$$\Rightarrow_{rm} \text{id}_1 * \text{id}_2 + \text{id}_3$$

$$E \Rightarrow_{rm} E + E$$

$$\Rightarrow_{rm} E + \text{id}_3$$

$$\Rightarrow_{rm} E * E + \text{id}_3$$

$$\Rightarrow_{rm} E * \text{id}_2 + \text{id}_3$$

$$\Rightarrow_{rm} \text{id}_1 * \text{id}_2 + \text{id}_3$$

在右句型  $E * E + \text{id}_3$  中，句柄不唯一

\* 右句型：最右推导可得的句型



# 用栈实现移进-归约分析

先通过“移进-归约分析器在分析输入串 $id_1 * id_2 + id_3$ 时的动作序列“来了解移进-归约分析的工作方式。



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3$ \$	





# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3$ \$	移进



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
\$ $E$	$* id_2 + id_3 \$$	



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	





# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
$\$E*\textcolor{blue}{E}$	$+ id_3 \$$	



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
$\$E*E$	$+ id_3 \$$	移进? 归约?



# 移进-归约需解决的一些问题

- 如何决策是选择移进还是归约？
- 进行归约时，怎么确定右句型中将要归约的子串  
(即句柄)
  - 句柄总是出现在栈顶
- 进行归约时，如何确定选择哪一个产生式？



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
$\$E*E$	$+ id_3 \$$	移进



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
$\$E*E$	$+ id_3 \$$	移进
$\$E*E+$	$id_3 \$$	



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
$\$E*E$	$+ id_3 \$$	移进
$\$E*E+$	$id_3 \$$	移进



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
$\$E*E$	$+ id_3 \$$	移进
$\$E*E+$	$id_3 \$$	移进
$\$E*E+id_3$	\$	





# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
$\$E*E$	$+ id_3 \$$	移进
$\$E*E+$	$id_3 \$$	移进
$\$E*E+id_3$	\$	按 $E \rightarrow id$ 归约
$\$E*E+E$	\$	



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
$\$E*E$	$+ id_3 \$$	移进
$\$E*E+$	$id_3 \$$	移进
$\$E*E+id_3$	\$	按 $E \rightarrow id$ 归约
$\$E*E+E$	\$	按 $E \rightarrow E+E$ 归约
$\$E*E$	\$	



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
$\$E$	$* id_2 + id_3 \$$	移进
$\$E*$	$id_2 + id_3 \$$	移进
$\$E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
$\$E*E$	$+ id_3 \$$	移进
$\$E*E+$	$id_3 \$$	移进
$\$E*E+id_3$	\$	按 $E \rightarrow id$ 归约
$\$E*E+E$	\$	按 $E \rightarrow E+E$ 归约
$\$E*E$	\$	按 $E \rightarrow E*E$ 归约
$\$E$	\$	



# 移进-归约分析： $id_1 * id_2 + id_3$

栈	输 入	动 作
\$	$id_1 * id_2 + id_3 \$$	移进
\$ $id_1$	$* id_2 + id_3 \$$	按 $E \rightarrow id$ 归约
\$ $E$	$* id_2 + id_3 \$$	移进
\$ $E*$	$id_2 + id_3 \$$	移进
\$ $E*id_2$	$+ id_3 \$$	按 $E \rightarrow id$ 归约
\$ $E*E$	$+ id_3 \$$	移进
\$ $E*E+$	$id_3 \$$	移进
\$ $E*E+id_3$	\$	按 $E \rightarrow id$ 归约
\$ $E*E+E$	\$	按 $E \rightarrow E+E$ 归约
\$ $E*E$	\$	按 $E \rightarrow E*E$ 归约
\$ $E$	\$	接受



# 移进-归约分析的冲突

□ 移进-归约冲突( shift/reduce conflict)

例

*stmt* → **if *expr* then *stmt***

| **if *expr* then *stmt* else *stmt***

| **other**

如果移进-归约分析器处于格局(configuration)

栈

输入

... **if *expr* then *stmt***

**else ... \$**



# 移进-归约分析的冲突

## □ 归约-归约冲突(reduce/reduce conflict)

$stmt \rightarrow id (parameter\_list) \mid expr = expr$

$parameter\_list \rightarrow parameter\_list, parameter \mid parameter$

$parameter \rightarrow id$

$expr \rightarrow id (expr\_list) \mid id$

$id(...)$ 是数组元素的引用

$expr\_list \rightarrow expr\_list, expr \mid expr$

由 $A(I, J)$ 开始的语句

归约成 $expr$ 还是  
 $parameter$ ?

栈

... id ( id

输入

, id )...



# 移进-归约分析的冲突

## □ 归约-归约冲突(reduce/reduce conflict)

$stmt \rightarrow \text{id} (parameter\_list) \mid expr = expr$

$parameter\_list \rightarrow parameter\_list, parameter \mid parameter$

$parameter \rightarrow \text{id}$

$expr \rightarrow \text{id} (expr\_list) \mid \text{id}$  id(...)是数组元素的引用

$expr\_list \rightarrow expr\_list, expr \mid expr$

由A(I, J)开始的语句 (词法分析查符号表, 区分第一个id)

栈

输入

... **procid**( id

, id )...

## ■ 需要修改上面的文法



## 3.5 LR分析器

(**L**-scanning from left to right; **R**- rightmost derivation in reverse)

□ LR分析算法：效率高

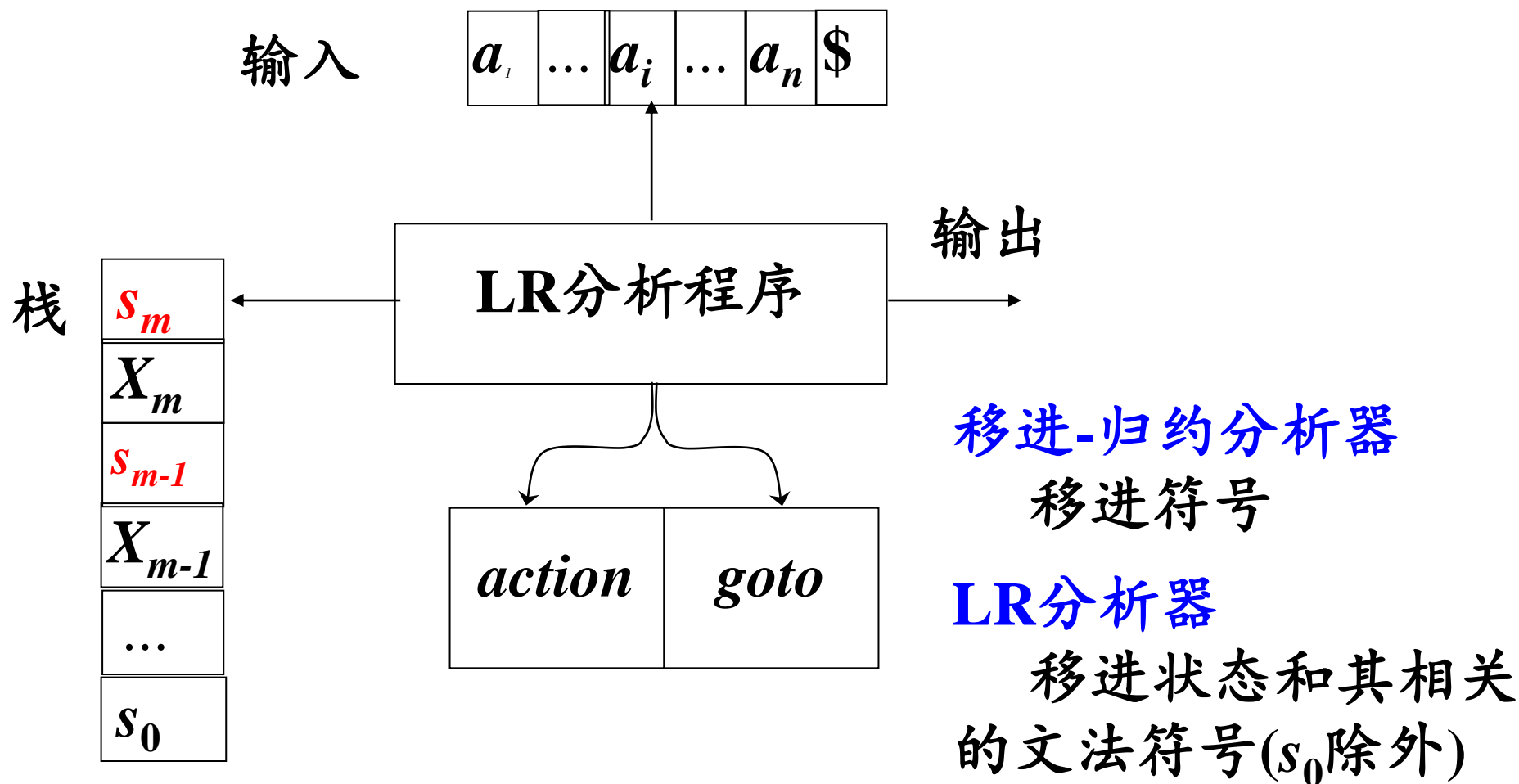
□ LR分析表的构造技术

简单的LR(SLR)、规范的LR、向前看LR(LALR)



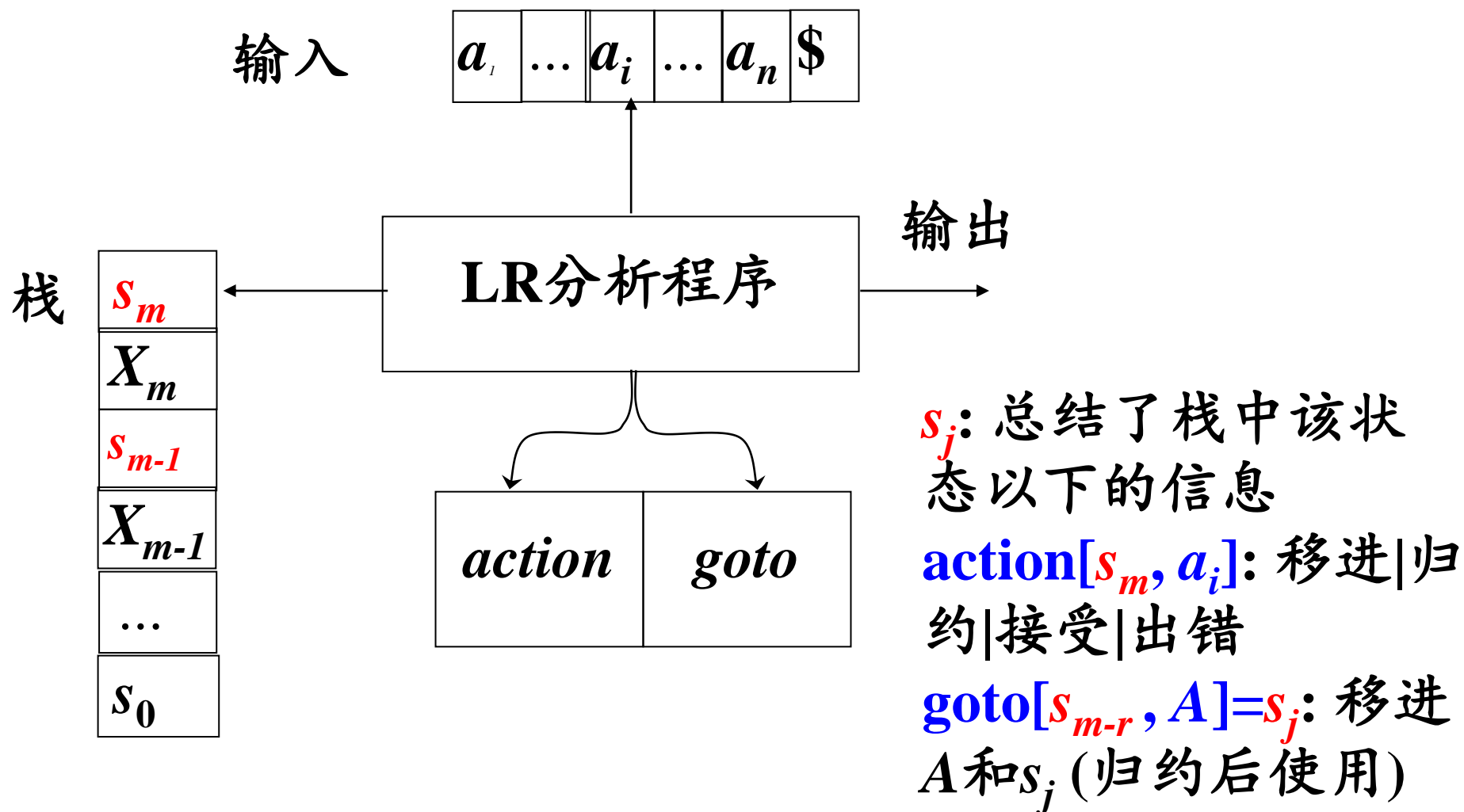


# LR分析算法: 分析器的模型





# LR分析算法: 分析器的模型





# LR分析算法：举例

例  $E \rightarrow E + T / E \rightarrow T$

**P69**  $T \rightarrow T * F / T \rightarrow E$

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

*si* 移进当前输入符号和状态*i*

*rj* 按第*j*个产生式进行归约

*acc* 接受

状态	动 作	转 移		
	id + * ( ) \$	<i>E</i>	<i>T</i>	<i>F</i>
0	<i>s5</i> <i>s4</i>	1	2	3
1	<i>s6</i> <i>acc</i>			
2	<i>r2</i> <i>s7</i> <i>r2</i> <i>r2</i>			
3	<i>r4</i> <i>r4</i> <i>r4</i> <i>r4</i>			
4	<i>s5</i> <i>s4</i>	8	2	3



# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	



# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	移进 (查action表)
0 id 5	* id + id \$	



# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	移进
0 id 5	* id + id \$	按 $F \rightarrow id$ 归约
0 F 3	* id + id \$	<div>1. 查<math>action[5, *] \Rightarrow</math>归约 2. 执行归约(<math>F \rightarrow \alpha</math>):<ul style="list-style-type: none"><li>从栈中弹出<math> \alpha </math>个状态-符号对</li><li>查<math>goto[0, F] \Rightarrow 3</math></li><li>将(<math>F, 3</math>)入栈</li></ul></div>



# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	移进
0 id 5	* id + id \$	按 $F \rightarrow id$ 归约
0 $F$ 3	* id + id \$	按 $T \rightarrow F$ 归约
0 $T$ 2	* id + id \$	



# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	移进
0 id 5	* id + id \$	按 $F \rightarrow id$ 归约
0 $F$ 3	* id + id \$	按 $T \rightarrow F$ 归约
0 $T$ 2	* id + id \$	移进
0 $T$ 2 * 7	id + id \$	





# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	移进
0 id 5	* id + id \$	按 $F \rightarrow id$ 归约
0 $F$ 3	* id + id \$	按 $T \rightarrow F$ 归约
0 $T$ 2	* id + id \$	移进
0 $T$ 2 * 7	id + id \$	移进
0 $T$ 2 * 7 id 5	+ id \$	



# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	移进
0 id 5	* id + id \$	按 $F \rightarrow id$ 归约
0 $F$ 3	* id + id \$	按 $T \rightarrow F$ 归约
0 $T$ 2	* id + id \$	移进
0 $T$ 2 * 7	id + id \$	移进
0 $T$ 2 * 7 id 5	+ id \$	按 $F \rightarrow id$ 归约
0 $T$ 2 * 7 $F$ 10	+ id \$	



# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	移进
0 id 5	* id + id \$	按 $F \rightarrow id$ 归约
0 $F$ 3	* id + id \$	按 $T \rightarrow F$ 归约
0 $T$ 2	* id + id \$	移进
0 $T$ 2 * 7	id + id \$	移进
0 $T$ 2 * 7 id 5	+ id \$	按 $F \rightarrow id$ 归约
0 $T$ 2 * 7 $F$ 10	+ id \$	按 $T \rightarrow T * F$ 归约
...	...	...
0 $E$ 1	\$	接受



# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	移进
0 id 5	* id + id \$	按 $F \rightarrow id$ 归约
0 $F$ 3	* id + id \$	按 $T \rightarrow F$ 归约
0 $T$ 2	* id + id \$	移进
0 $T$ 2 * 7	id + id \$	移进
0 $T$ 2 * 7 id 5	+ id \$	按 $F \rightarrow id$ 归约
0 $T$ 2 * 7 $F$ 10	+ id \$	按 $T \rightarrow T * F$ 归约
...	...	...



# LR分析算法：举例

栈	输 入	动 作
0	id * id + id \$	移进
0 id 5	* id + id \$	按 $F \rightarrow id$ 归约
0 $F$ 3	* id + id \$	按 $T \rightarrow F$ 归约
0 $T$ 2	* id + id \$	移进
0 $T$ 2 * 7	id + id \$	移进
0 $T$ 2 * 7 id 5	+ id \$	按 $F \rightarrow id$ 归约
0 $T$ 2 * 7 $F$ 10	+ id \$	按 $T \rightarrow T * F$ 归约
...	...	...
0 $E$ 1	\$	接受



# LR分析: 基本概念

## □ 活前缀 ( viable prefix )

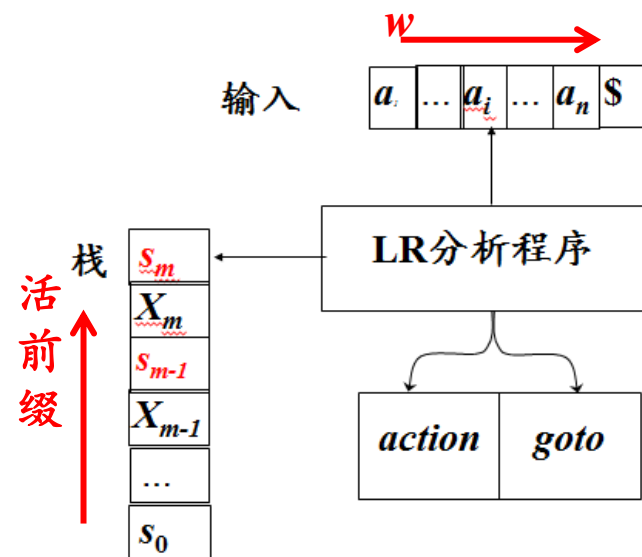
- 右句型的前缀，该前缀不超过最右句柄的右端

$$S \Rightarrow_{rm}^* \gamma A w \Rightarrow_{rm} \gamma \beta w$$

- $\gamma \beta$  的任何前缀 (包括  $\epsilon$  和  $\gamma \beta$  本身) 都是活前缀
- $w$  仅包含终结符

## ■ 对应到LR分析模型上的特点

- 活前缀: 是LR分析栈中从栈底到栈顶的**文法符号**连接形成的串
- $w$ : 输入缓冲区中剩余的记号串





# LR分析: 基本概念

## □ LR文法(LR grammar)

- 能为之构造出所有条目（若存在）**都唯一**的LR分析表

## □ LR分析表

- 移进+ goto（转移函数）：本质上是识别活前缀的DFA

状态	动 作	转 移
	id + * ( ) \$	<i>E</i> <i>T</i> <i>F</i>
0	<i>s5</i> <i>s4</i>	1 2 3
1	<i>s6</i> <i>acc</i>	
2	<i>r2</i> <i>s7</i> <i>r2</i> <i>r2</i>	
3	<i>r4</i> <i>r4</i> <i>r4</i> <i>r4</i>	
4	<i>s5</i> <i>s4</i>	8 2 3



# LR分析方法的特点

- 栈中的文法符号总是形成一个活前缀
- 分析表的转移函数本质上是识别活前缀的DFA
- 栈顶的状态符号包含了确定句柄所需要的一切信息
- 是已知的最一般的无回溯的移进-归约方法
- 能分析的文法类是预测分析能分析的文法类的真超集
- 能及时发现语法错误
  
- 手工构造分析表的工作量太大





# LR方法与LL方法的比较

	LR(1)方 法	LL(1)方 法
建立分析树的方式	自下而上	自上而下
归约还是推导	规范归约	最左推导
决定使用产生式的时机		

$S \Rightarrow \dots \Rightarrow \gamma A b w \Rightarrow \gamma l \beta b w$

$A \rightarrow l\beta$

LL(1)决定用该  
产生式的位置



# LR方法与LL方法的比较

	LR(1)方法	LL(1)方法
建立分析树的方式	自下而上	自上而下
归约还是推导	规范归约	最左推导
决定使用产生式的时机		

$S \Rightarrow \dots \Rightarrow \gamma A b w \Rightarrow \gamma l \beta b w$

$A \rightarrow l\beta$

LL(1)决定用该  
产生式的位置

LR(1)决定用该  
产生式的位置



# LR方法与LL方法的比较

	LR(1)方法	LL(1)方法
建立分析树的方式	自下而上	自上而下
归约还是推导	规范归约	最左推导
决定使用产生式的时机	看见产生式右部推出的 <b>整个</b> 终结字符串后，才确定用哪个产生式归约	看见产生式右部推出的 <b>第一个</b> 终结符后，便要确定用哪个产生式推导

$S \Rightarrow \dots \Rightarrow \gamma A b w \Rightarrow \gamma l \beta b w$

$A \rightarrow l\beta$

LL(1)决定用该  
产生式的位置

LR(1)决定用该  
产生式的位置

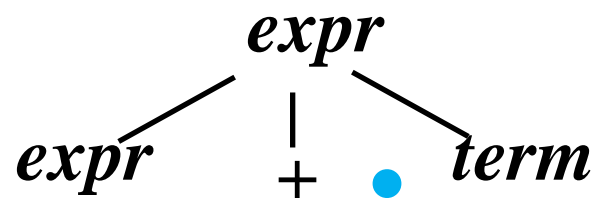


# SLR分析表的构造

## SLR (Simple LR)

### □ LR(0) 项目

- 在右部的某个地方加点的产生式
- 加点的目的是用来表示分析过程中的状态



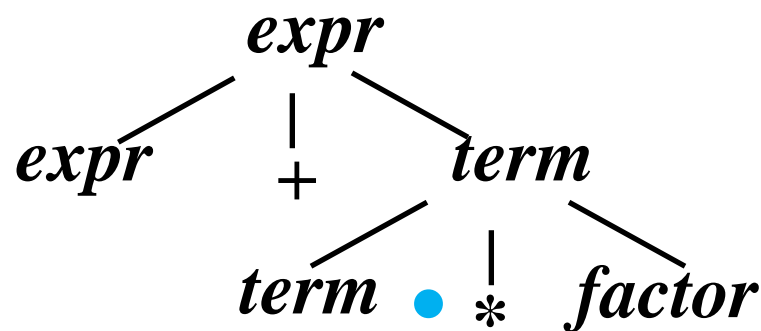


# SLR分析表的构造

## SLR (Simple LR)

### □ LR(0) 项目

- 在右部的某个地方加点的产生式
- 加点的目的是用来表示分析过程中的状态





# SLR分析表的构造

## SLR (Simple LR)

### □ LR(0) 项目

- 在右部的某个地方加点的产生式
- 加点的目的是用来表示分析过程中的状态

例  $A \rightarrow XYZ$  对应应有四个项目

$$A \rightarrow \cdot XYZ \quad A \rightarrow X \cdot YZ$$

$$A \rightarrow XY \cdot Z \quad A \rightarrow XYZ \cdot$$

例  $A \rightarrow \varepsilon$  只有一个项目和它对应

$$A \rightarrow \cdot$$



# SLR分析表的构造

## □ SLR分析表的构造

1. 从文法构造识别活前缀的DFA
2. 从上述DFA构造分析表



# 构造识别活前缀的DFA

## 1. 拓广文法 (augmented grammar)

$E' \rightarrow E$  指示分析器何时停止分析并宣布接受输入

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

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





# 构造识别活前缀的DFA

## 2. 构造LR(0)项目集规范族 (canonical LR(0) collection)

$I_0$ :

$E' \rightarrow \cdot E$



# 构造识别活前缀的DFA

## 2. 构造LR(0)项目集规范族 (canonical LR(0) collection)

$I_0$ :

求项目集的闭包closure( $I$ ) P75

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$



# 构造识别活前缀的DFA

## 2. 构造LR(0)项目集规范族 (canonical LR(0) collection)

$I_0$ :

求项目集的闭包closure( $I$ ) P75

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$



# 构造识别活前缀的DFA

## 2. 构造LR(0)项目集规范族 (canonical LR(0) collection)

$I_0$ :

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$

求项目集的闭包closure( $I$ ) P75

核心项目

1) 初始项目; 2) 点不在最左端的项目

非核心项目

非初始项目且点在最左端的项目

可以通过对核心项目求闭包来获得  
为节省存储空间, 可省去



# 构造识别活前缀的DFA

## 2. 构造LR(0)项目集规范族 (canonical LR(0) collection)

$I_0$ :

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$

$I_1$ :

$E' \rightarrow E \cdot$

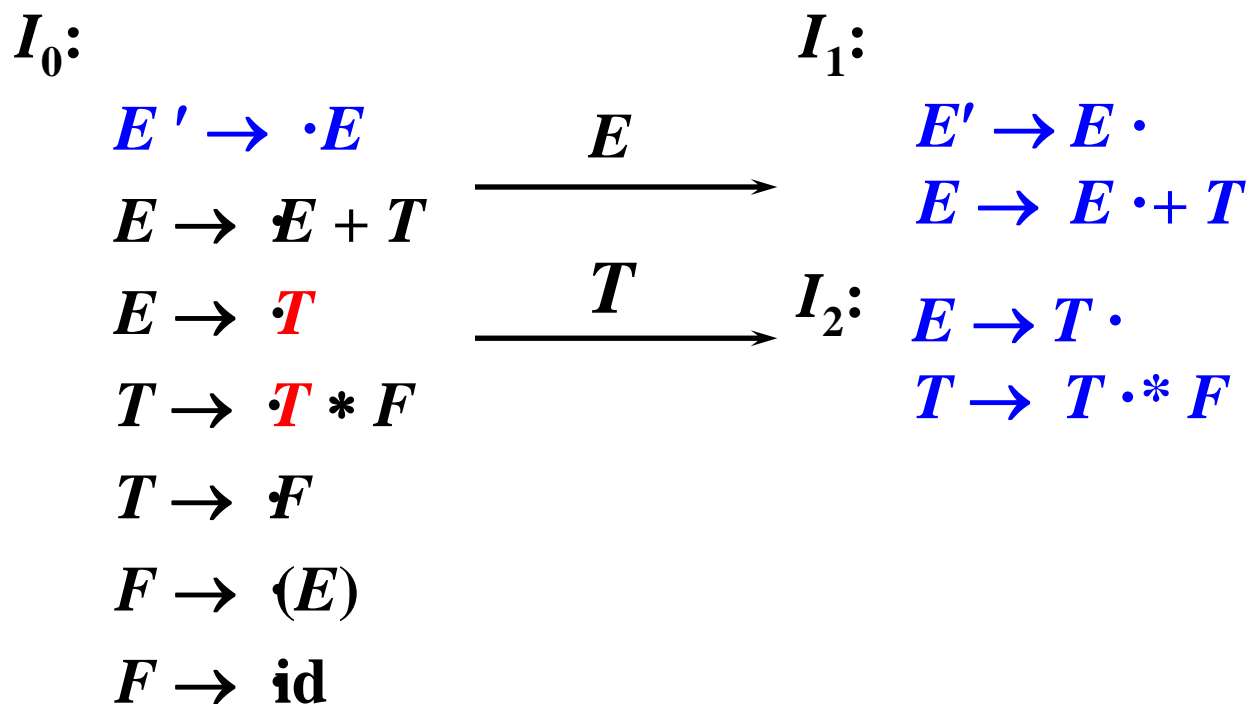
$E \rightarrow E \cdot + T$

$I_1 := \text{goto} (I_0, E)$



# 构造识别活前缀的DFA

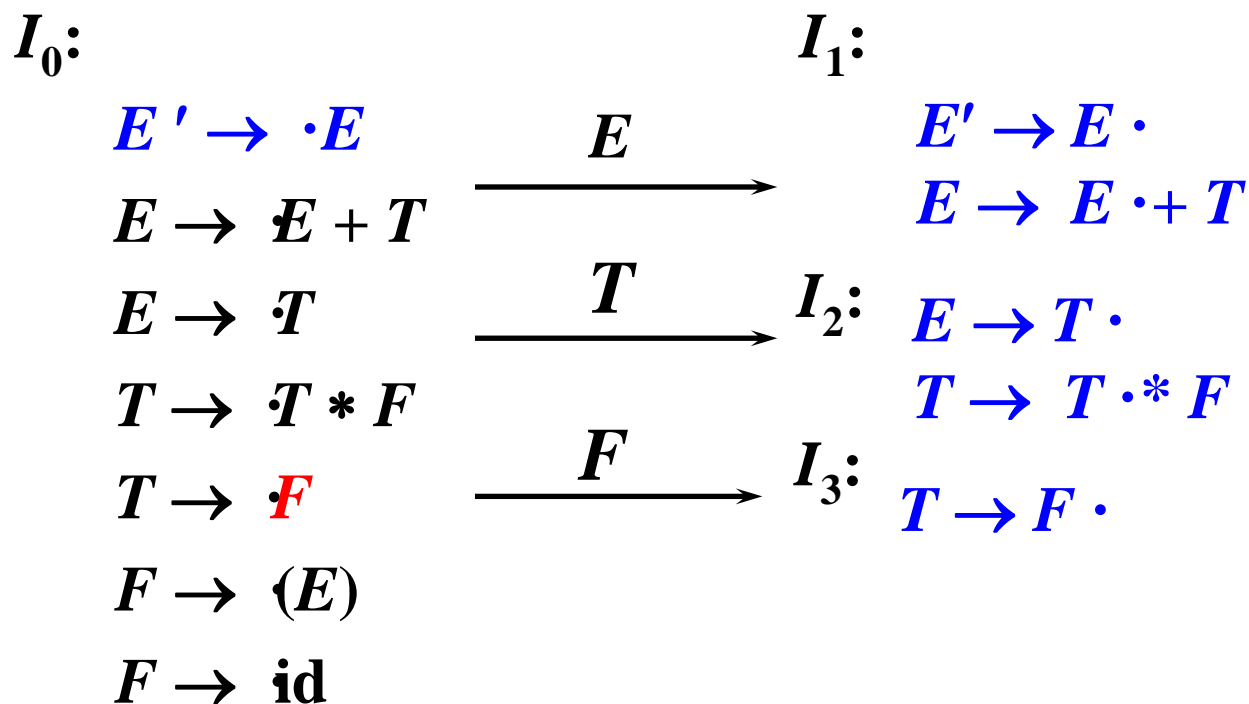
## 2. 构造LR(0)项目集规范族 (canonical LR(0) collection)





# 构造识别活前缀的DFA

## 2. 构造LR(0)项目集规范族 (canonical LR(0) collection)





# 构造识别活前缀的DFA

## 2. 构造LR(0)项目集规范族 (canonical LR(0) collection)

$I_0$ :

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$

$I_1$ :

$E' \rightarrow E \cdot$

$E \rightarrow E \cdot + T$

$I_2$ :

$E \rightarrow T \cdot$

$T \rightarrow T \cdot * F$

$I_3$ :

$T \rightarrow F \cdot$

$I_4$ :

$F \rightarrow ( \cdot E )$

$E \rightarrow \cdot E + T$

$T \rightarrow \cdot T * F$

$F \rightarrow \cdot (E)$

$E \rightarrow \cdot T$

$T \rightarrow \cdot F$

$F \rightarrow \cdot \text{id}$





# 构造识别活前缀的DFA

## 2. 构造LR(0)项目集规范族 (canonical LR(0) collection)

$I_0$ :

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$

$I_1$ :

$E' \rightarrow E \cdot$

$E \rightarrow E \cdot + T$

$I_2$ :

$E \rightarrow T \cdot$

$T \rightarrow T \cdot * F$

$I_3$ :

$T \rightarrow F \cdot$

$I_4$ :

$F \rightarrow ( \cdot E )$

$E \rightarrow \cdot E + T$

$T \rightarrow \cdot T * F$

$F \rightarrow \cdot (E)$

$E \rightarrow \cdot T$

$T \rightarrow \cdot F$

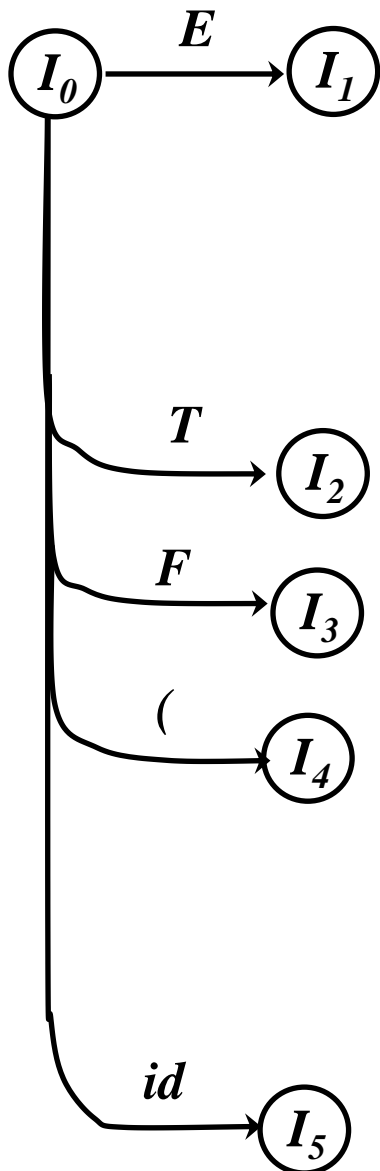
$F \rightarrow \cdot \text{id}$

$I_5$ :

$F \rightarrow \text{id} \cdot$



# 构造识别活前缀的DFA

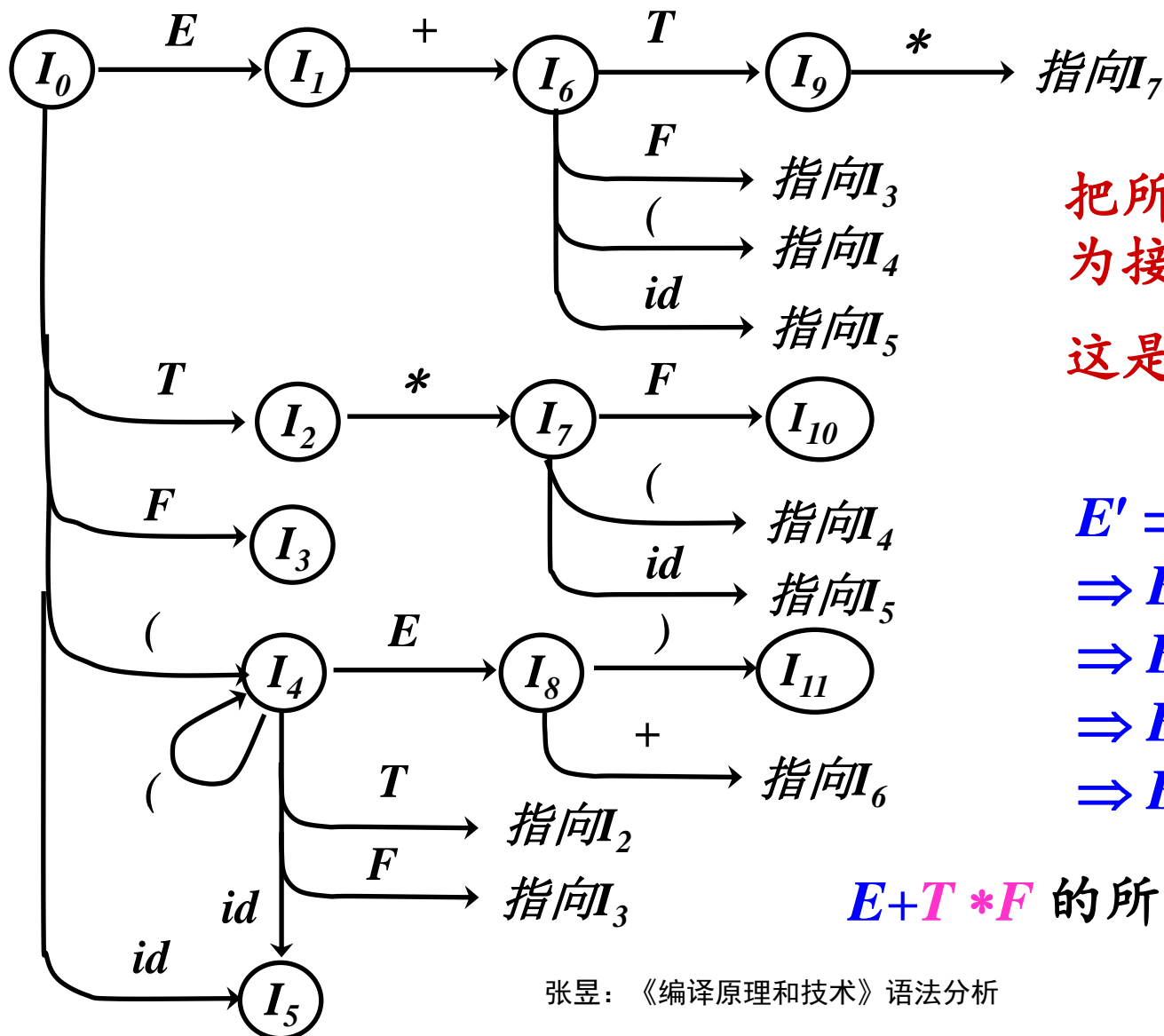


$I_3:$

$T \rightarrow F \cdot$  无状态转换



# 构造识别活前缀的DFA



把所有状态都作  
为接受状态

这是一个DFA

$E' \Rightarrow E$   
 $\Rightarrow E+T$   
 $\Rightarrow E+T * F$   
 $\Rightarrow E+T * id$   
 $\Rightarrow E+T * F * id$

$E+T * F$  的所有前缀都可接受



# 构造识别活前缀的NFA

$I_0$ :

每个项目一个状态

$E' \rightarrow \cdot E$

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$



# 构造识别活前缀的NFA

$I_0$ :

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

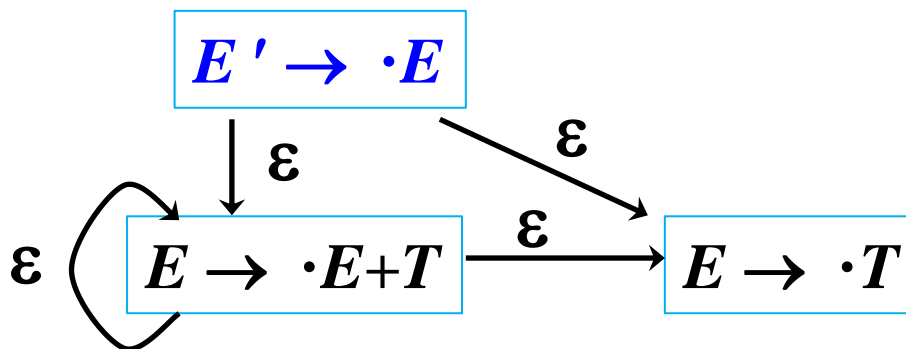
$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$

每个项目一个状态





# 构造识别活前缀的NFA

$I_0$ :

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

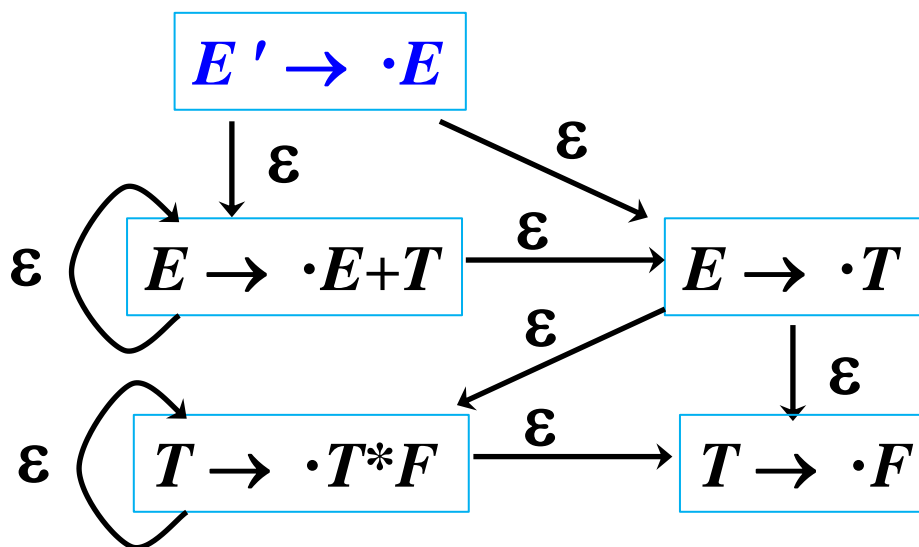
$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$

每个项目一个状态





# 构造识别活前缀的NFA

$I_0$ :

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

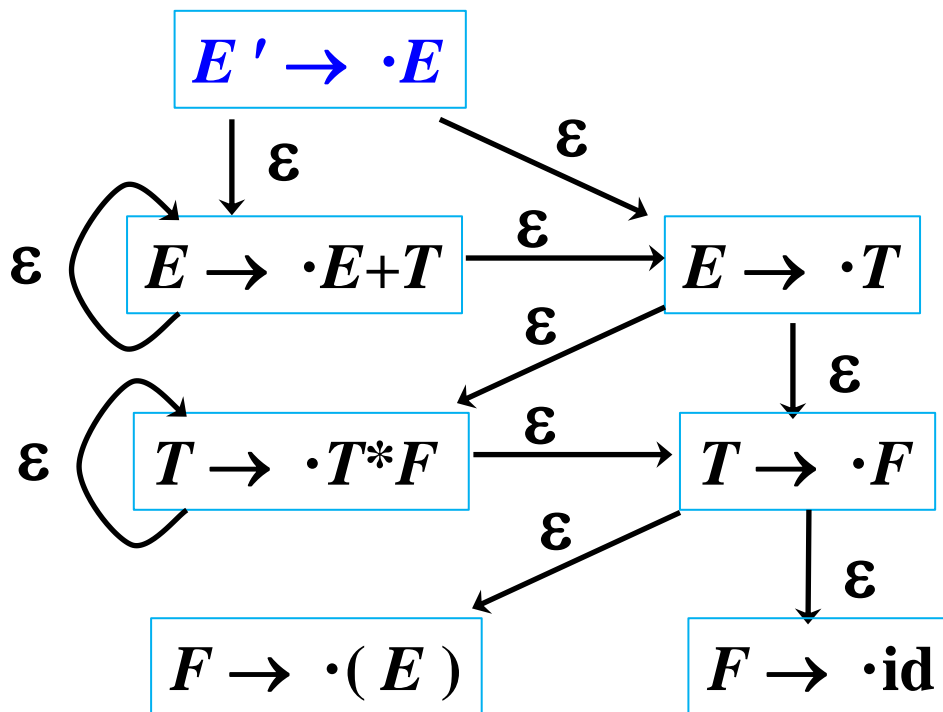
$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$

每个项目一个状态





# 构造识别活前缀的NFA

$I_0$ :

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

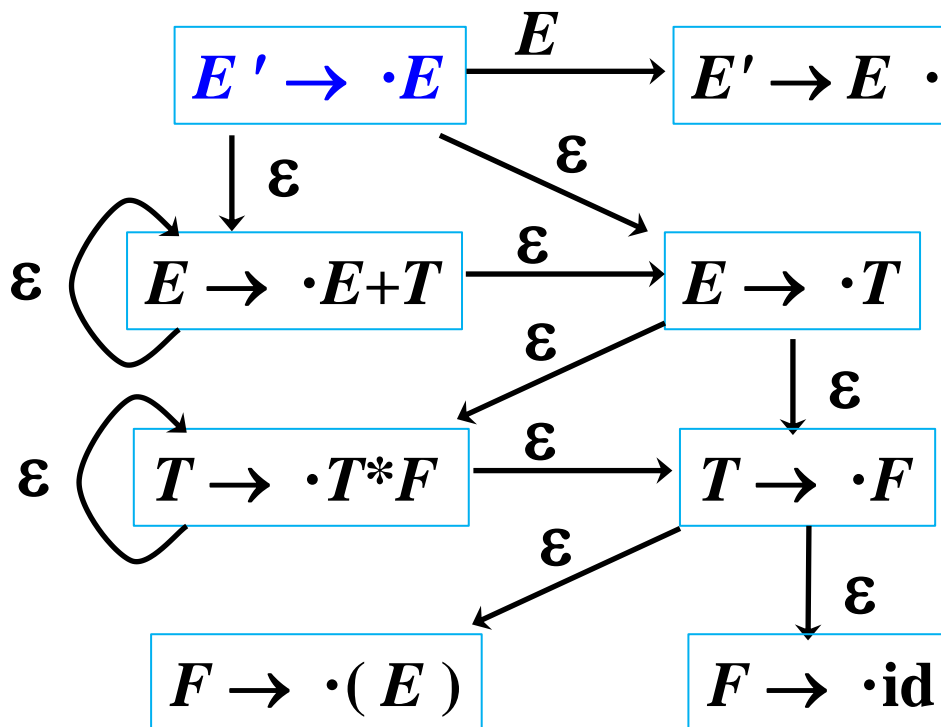
$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

$F \rightarrow \cdot \text{id}$

每个项目一个状态







# 有效项目

如果  $S' \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta_1 \beta_2 w$ , 那么就说项目  $A \rightarrow \beta_1 \beta_2$  对活前缀  $\alpha \beta_1$  是有效的

■ 一个项目可能对好几个活前缀都是有效的

$E \rightarrow E+T$  对  $\epsilon$  和  $($  这两个活前缀都有效

$E' \Rightarrow E \Rightarrow E+T$  ( $\alpha, \beta_1$  都为空)

$E' \Rightarrow E \Rightarrow (E) \Rightarrow (E+T)$  ( $\alpha = "($ ",  $\beta_1$  为空)

该DFA读过 $\epsilon$  和 $($ 后到达不同的状态, 那么项目  $E \rightarrow \cdot E+T$  就出现在对应的不同项目集中



# 有效项目

如果  $S' \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta_1 \beta_2 w$ , 那么就说项目  $A \rightarrow \beta_1 \beta_2$  对活前缀  $\alpha \beta_1$  是有效的

■ 一个项目可能对好几个活前缀都是有效的

从项目  $A \rightarrow \beta_1 \beta_2$  对活前缀  $\alpha \beta_1$  有效这个事实可以知道

- ✓ 如果  $\beta_2 \neq \varepsilon$ , 应该移进
- ✓ 如果  $\beta_2 = \varepsilon$ , 应该用产生式  $A \rightarrow \beta_1$  归约



# 有效项目

如果  $S' \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta_1 \beta_2 w$ , 那么就说项目  $A \rightarrow \beta_1 \beta_2$  对活前缀  $\alpha \beta_1$  是有效的

- 一个项目可能对好几个活前缀都是有效的
- 一个活前缀可能有多个有效项目

一个活前缀  $\gamma$  的**有效项目集**是从这个DFA的初态出发, 沿着标记为  $\gamma$  的路径到达的那个项目集 (状态)



# 有效项目

例 串  $E + T *$  是活前缀，读完它后，DFA处于状态  $I_7$

$I_7: \quad T \rightarrow T * F, F \rightarrow (E), F \rightarrow \text{id}$

$S' \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha \beta_1 \beta_2 w$       活前缀:  $\alpha \beta_1$

$E' \Rightarrow E$	$E' \Rightarrow E$	$E' \Rightarrow E$
$\Rightarrow E + T$	$\Rightarrow E + T$	$\Rightarrow E + T$
$\Rightarrow E + T * F$	$\Rightarrow E + T * F$	$\Rightarrow E + T * F$
$\Rightarrow E + T * \text{id}$	$\Rightarrow E + T * (E)$	$\Rightarrow E + T * \text{id}$
$\Rightarrow E + T * F * \text{id}$		



# 从DFA构造SLR分析表

- 状态  $i$  从  $I_i$  构造，按如下方法确定 *action* 函数：
  - 移进：如果  $[A \rightarrow \alpha \textcolor{red}{a}\beta]$  在  $I_i$  中，并且  $\text{goto}(I_i, \textcolor{red}{a}) = I_j$ ，那么置  $\text{action}[i, a]$  为  $sj$
  - 归约：如果  $[A \rightarrow \alpha \cdot]$  在  $I_i$  中，那么对 **FOLLOW(A)** 中的所有  $a$ ，置  $\text{action}[i, a]$  为  $rj$ ， $j$  是产生式  $A \rightarrow \alpha$  的编号
  - 接受：如果  $[S' \rightarrow S \cdot]$  在  $I_i$  中，那么置  $\text{action}[i, \$]$  为  $\text{acc}$

如果出现动作冲突，那么该文法就不是SLR(1)的



# 从DFA构造SLR分析表

- 状态  $i$  从  $I_i$  构造，按如下方法确定 *action* 函数：
  - 移进：如果  $[A \rightarrow \alpha \ a\beta]$  在  $I_i$  中，并且  $goto(I_i, a) = I_j$ ，那么置  $action[i, a]$  为  $sj$
  - 归约：如果  $[A \rightarrow \alpha \ ]$  在  $I_i$  中，那么对  $FOLLOW(A)$  中的所有  $a$ ，置  $action[i, a]$  为  $rj$ ， $j$  是产生式  $A \rightarrow \alpha$  的编号
  - 接受：如果  $[S' \rightarrow S \ ]$  在  $I_i$  中，那么置  $action[i, \$]$  为  $acc$
- 构造状态  $i$  的 *goto* 函数
  - 对所有的非终结符  $A$ ，如果  $goto(I_i, A) = I_j$ ，则  $goto[i, A] = j$
- 不能由上面两步定义的条目都置为 error
- 分析器的初始状态：包含  $[S' \rightarrow \ ]$  的项目集对应的状态



# SLR分析表构造举例

例  $I_2$ :

$E \rightarrow T \cdot$

$T \rightarrow T \cdot * F$

$E \rightarrow T \cdot$

因为

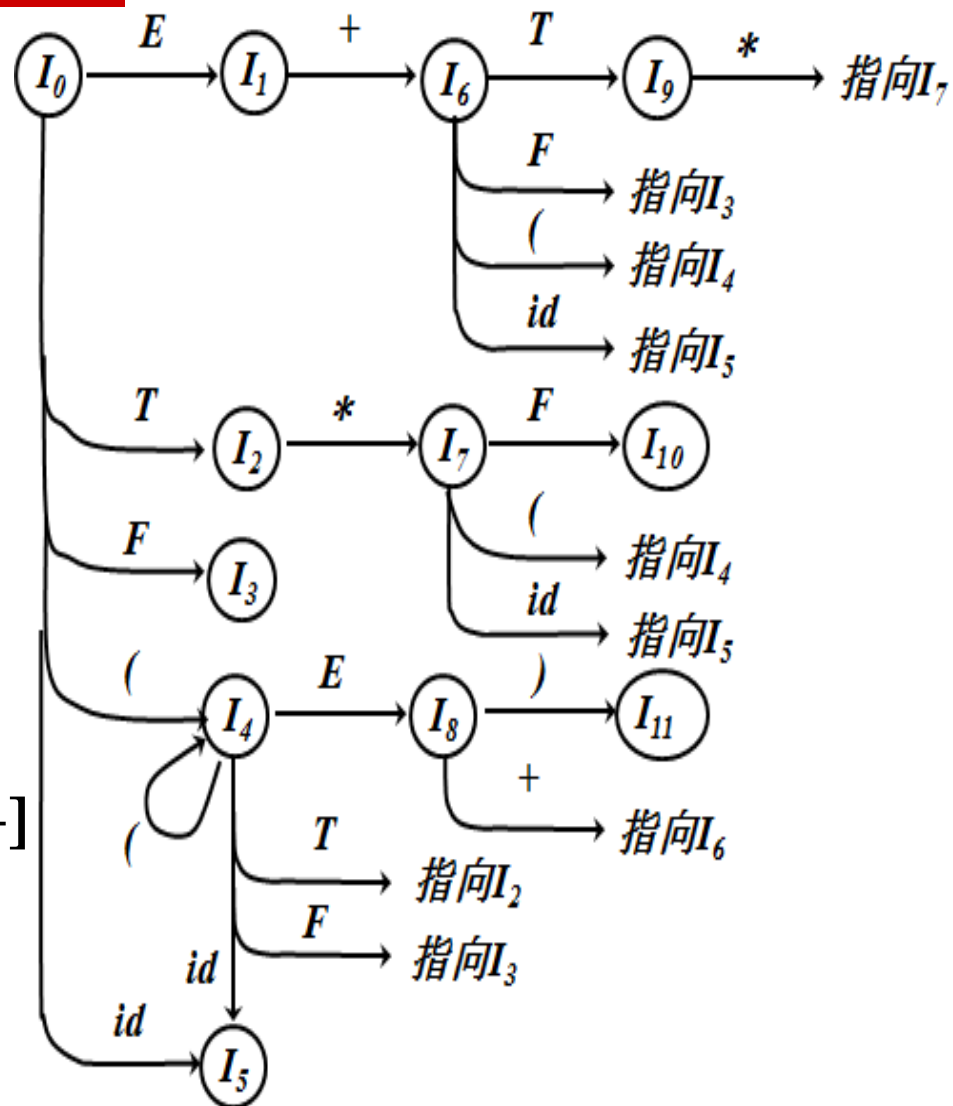
$\text{FOLLOW}(E) = \{\$, +, )\}$ ,

所以

$\text{action}[2, \$] = \text{action}[2, +]$   
 $= \text{action}[2, )] = r2$

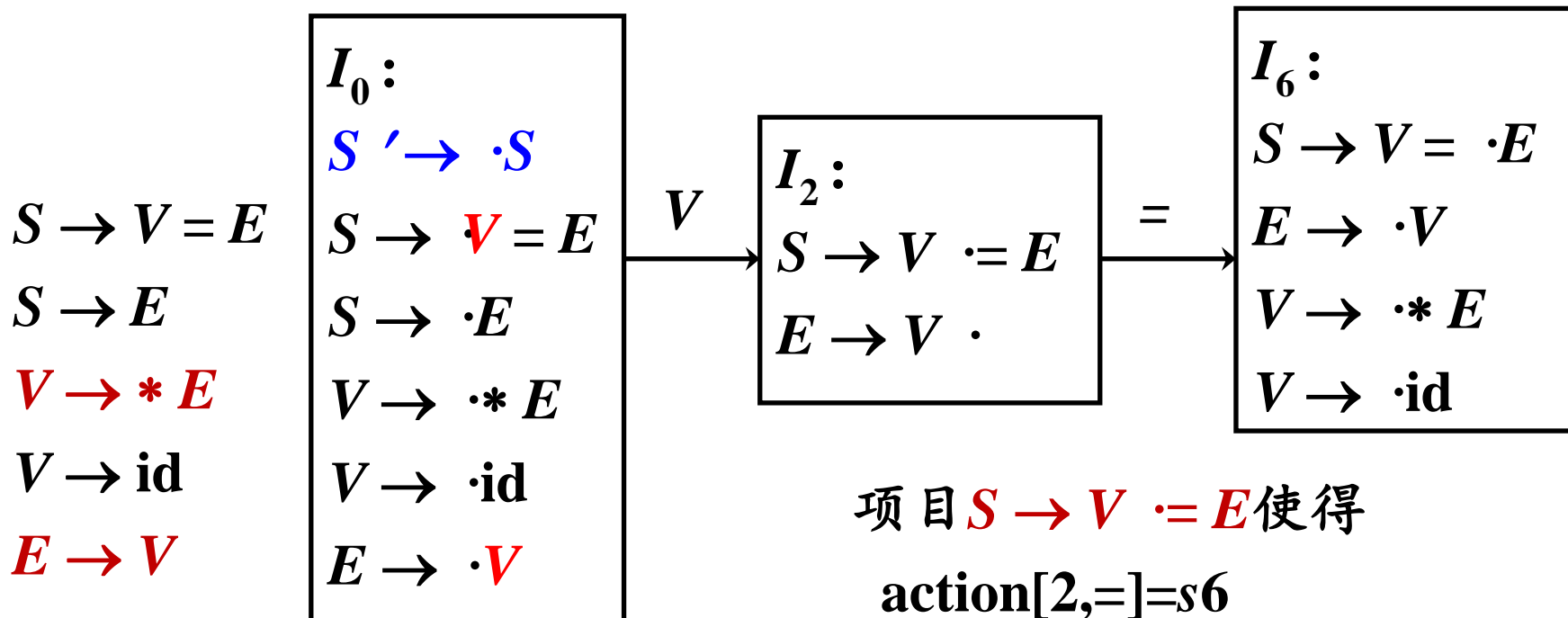
$T \rightarrow T \cdot * F$

$\text{action}[2, *] = s7$





# SLR(1)文法的描述能力有限



该文法并不是二义的

$S \$ \Rightarrow V = E \$ \Rightarrow * E = E \$$

$S \$ \Rightarrow V = E \$$  无句型  $E = E$  ☹️

$S \$ \Rightarrow E \$ \Rightarrow V \$$

项目  $S \rightarrow V \cdot = E$  使得

$\text{action}[2, =] = s6$

项目  $E \rightarrow V \cdot$  使得

$\text{action}[2, =]$  为按  $E \rightarrow V$  归约,

因为  $\text{Follow}(E) = \{=, \$\}$

产生移进-归约冲突





# 规范的LR分析

## □ LR(1)项目

重新定义项目，让它带上搜索符 (lookahead)，成为

$$[A \rightarrow \alpha \cdot \beta, a]$$

## □ LR(1)项目 $[A \rightarrow \alpha \cdot \beta, a]$ 对活前缀 $\gamma$ 有效：

如果存在着推导  $S \Rightarrow_{rm}^* \delta A w \Rightarrow_{rm} \delta \alpha \beta w$ ，其中：

■  $\gamma = \delta \alpha$ ;

■  $a$  是  $w$  的第一个符号，或者  $w$  是  $\varepsilon$  且  $a$  是  $\$$



# 规范的LR分析：举例

例  $S \rightarrow BB$

$B \rightarrow bB \mid a$

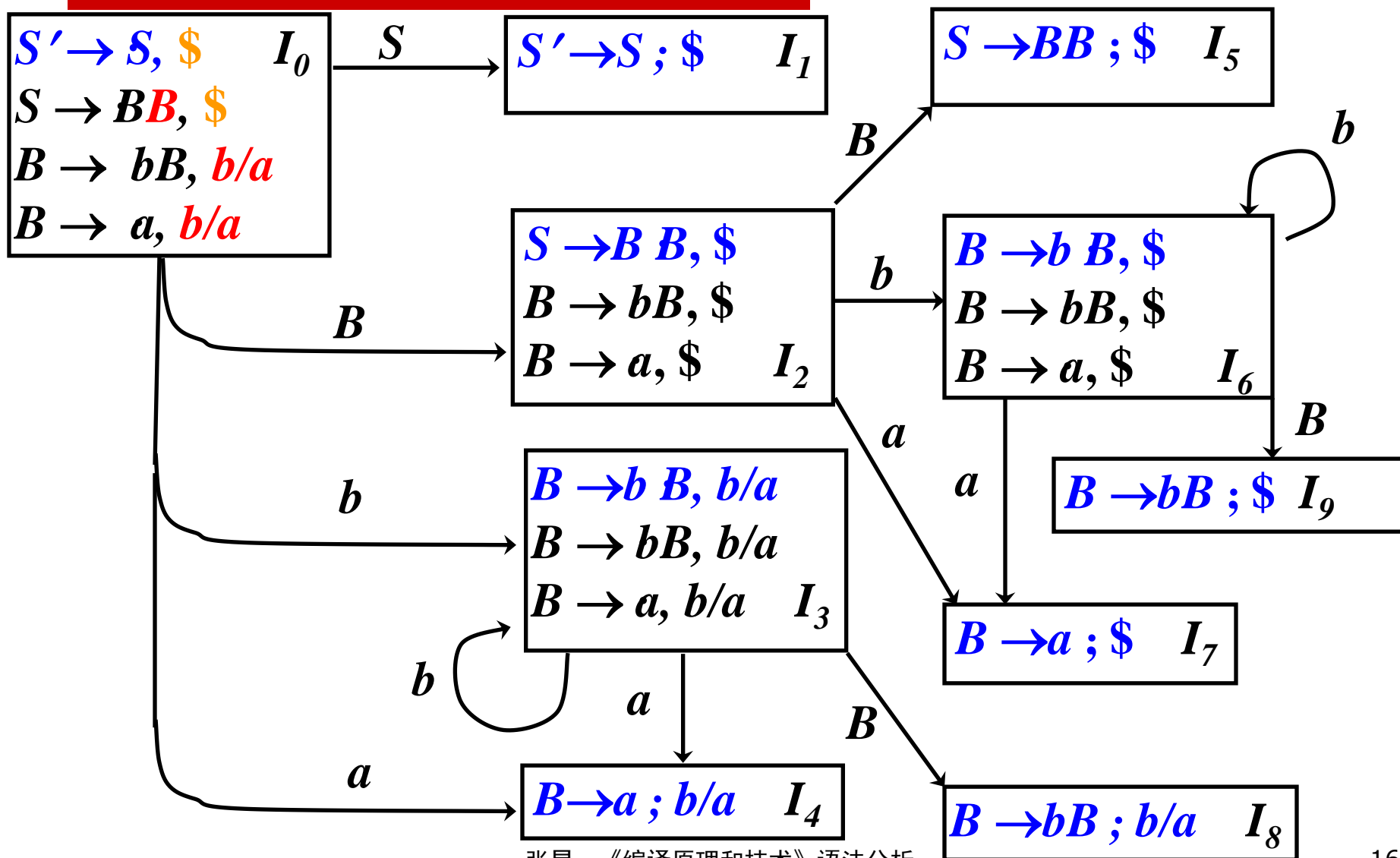
从最右推导  $S \Rightarrow_{rm}^* bbBba \Rightarrow_{rm} bbbBba$  看出：

$[B \rightarrow bB, b]$  对活前缀  $\gamma = bbb$  是有效的

对于项目  $[A \rightarrow \alpha ; a]$ ，是**根据搜索符  $a$  来填表**（归约项目），而不是根据  $A$  的后继符来填表



# 规范的LR分析：举例





# 构造规范的LR分析表

1. 基于LR(1)项目来构造识别  $G'$  活前缀的DFA
  2. 构造LR分析表, 状态  $i$  的 *action* 函数按如下确定
    - ① 如果  $[A \rightarrow \alpha \textcolor{red}{a}\beta, b]$  在  $I_i$  中, 且  $\text{goto}(I_i, a) = I_j$ , 那么置  $\text{action}[i, a]$  为  $sj$
    - ② 如果  $[A \rightarrow \alpha \cdot, \textcolor{red}{a}]$  在  $I_i$  中, 且  $A \neq S'$ , 那么置  $\text{action}[i, a]$  为  $rj$
    - ③ 如果  $[S' \rightarrow S ; \$]$  在  $I_i$  中, 那么置  $\text{action}[i, \$] = acc$
- 如果用上面规则构造, 出现了冲突, 则文法就不是LR(1)的
- *goto* 函数的确定: 如果  $\text{goto}(I_i, A) = I_j$ , 那么  $\text{goto}[i, A] = j$
  - 用上面规则未能定义的所有条目都置为 **error**
  - 初始状态是包含  $[S' \rightarrow \textcolor{blue}{S}, \$]$  的项目集对应的状态

SLR是根据Follow(A)来确定归约动作  
这里是根据搜索符 (上下文信息) 来确定

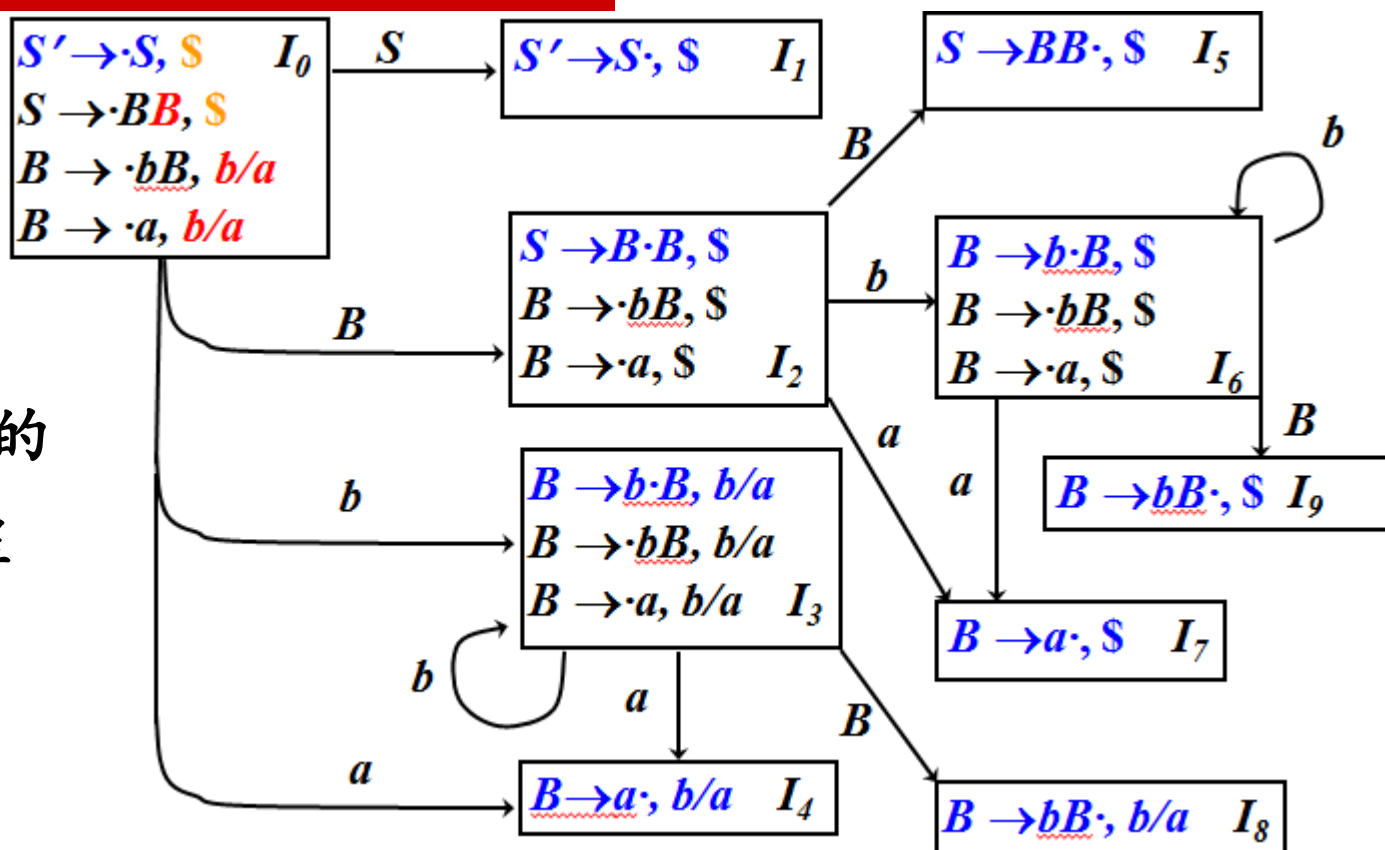


# 规范的LR分析

给出在以下  
两种输入下的  
LR分析过程

*bbabba\$*

*bba\$*





# 不是SLR(1)但是LR(1)的文法

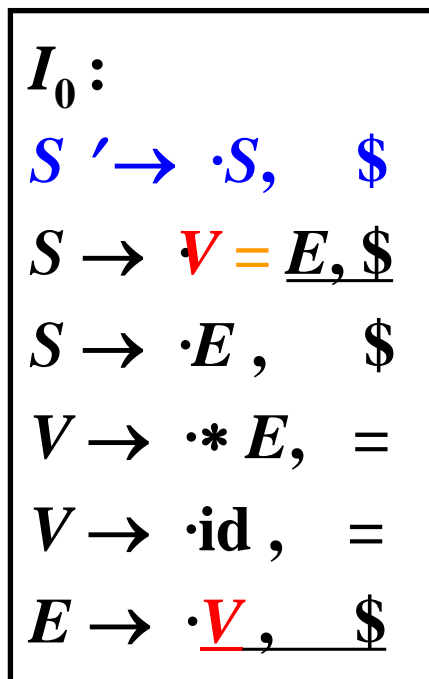
$S \rightarrow V = E$

$S \rightarrow E$

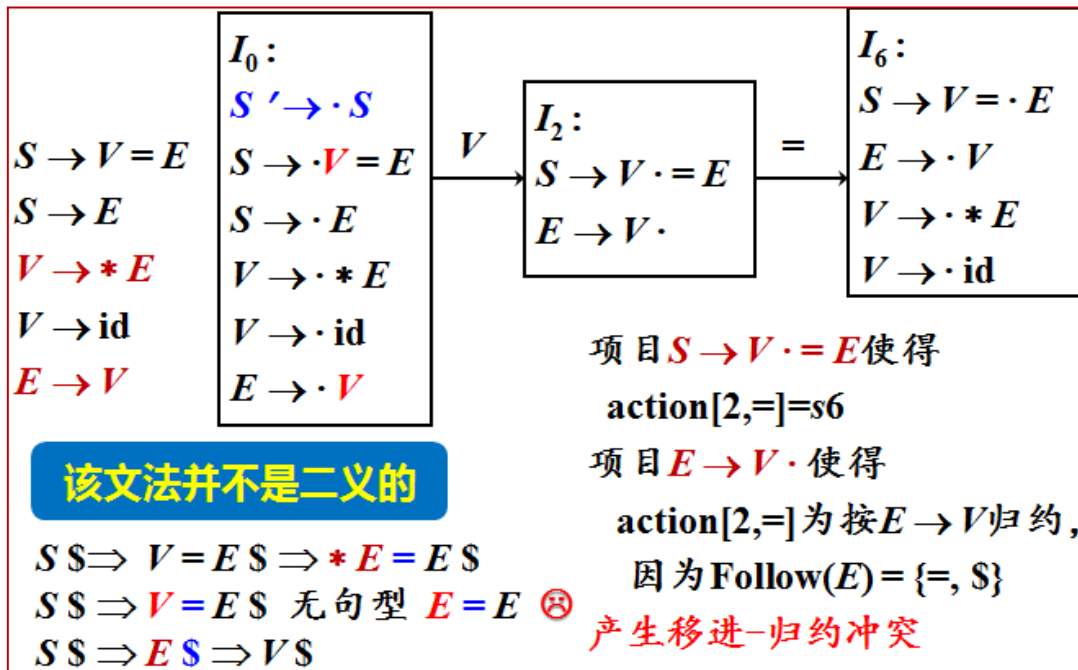
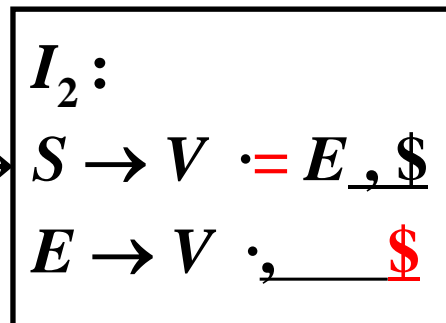
$V \rightarrow * E$

$V \rightarrow \text{id}$

$E \rightarrow V$



$V$



无移进-归约  
冲突



# LALR分析表

## □ 研究LALR的原因

- 规范的LR分析表的状态数偏多

## □ LALR特点

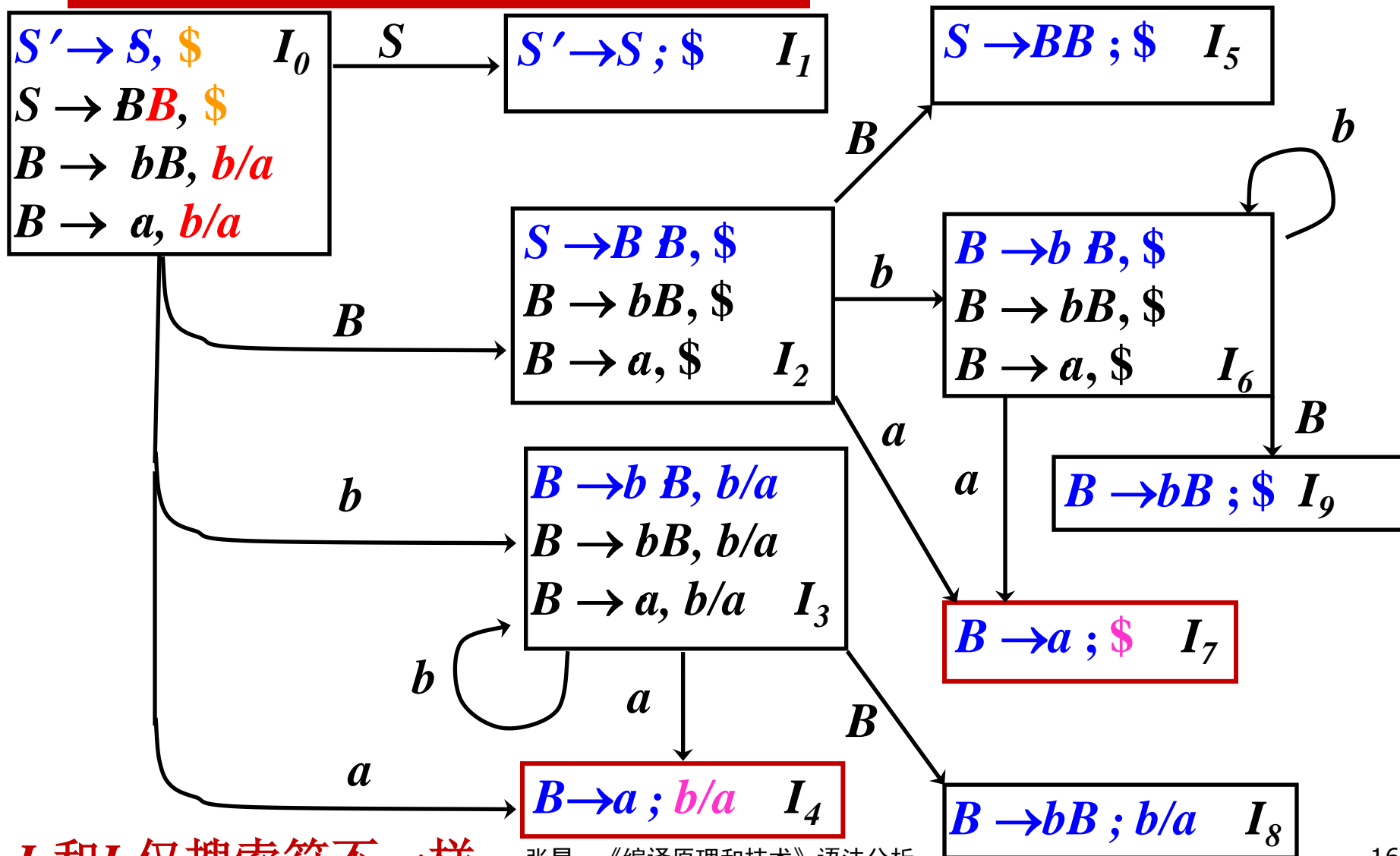
- LALR和SLR的分析表有同样多的状态，比规范的LR分析表要小得多
- LALR的能力介于SLR和规范LR之间
- LALR的能力在很多情况下已经够用

## □ LALR分析表构造方法

- 通过合并规范LR(1)项目集来得到



# 识别活前缀的DFA

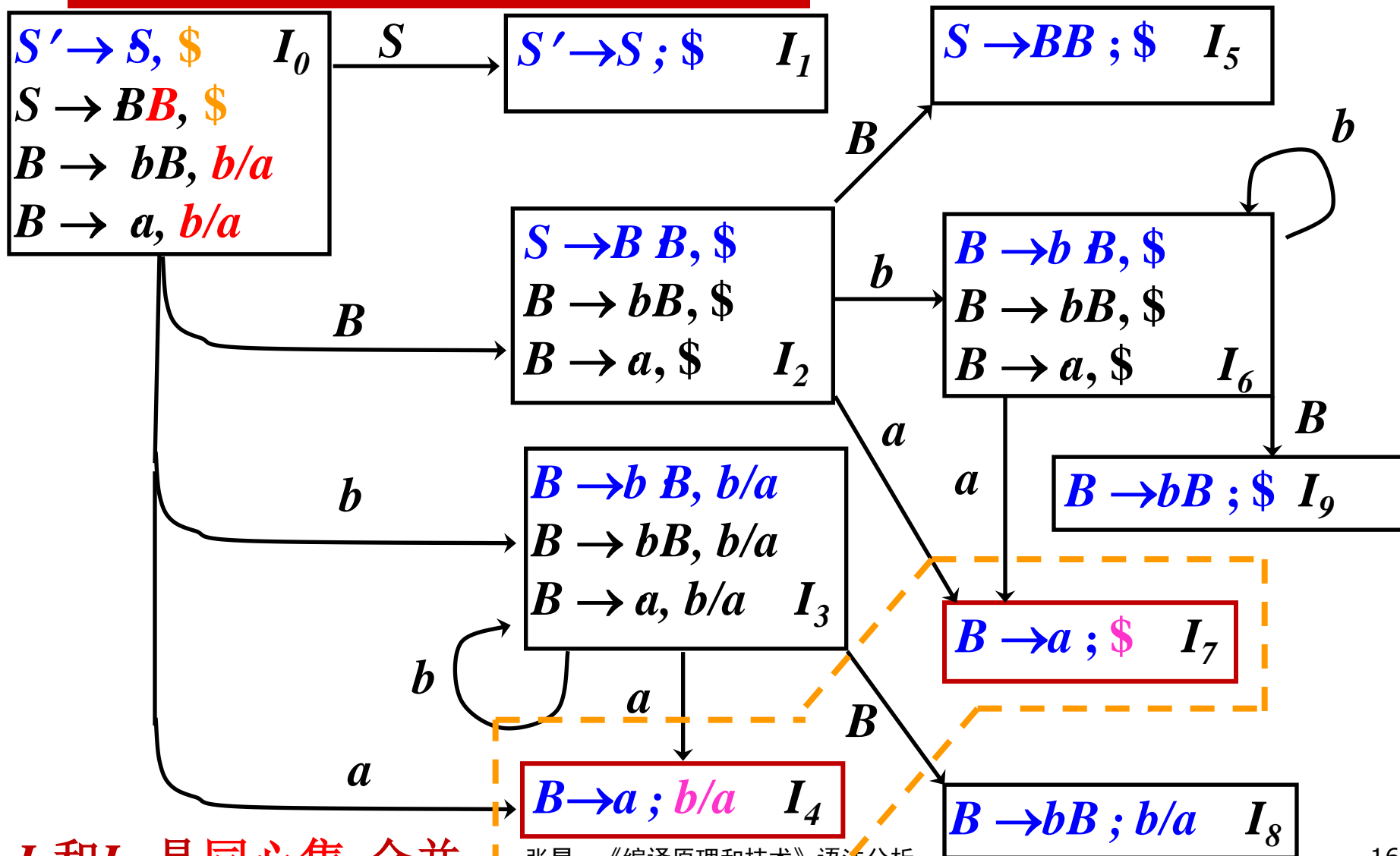


$I_4$ 和 $I_7$ 仅搜索符不一样





# 识别活前缀的DFA

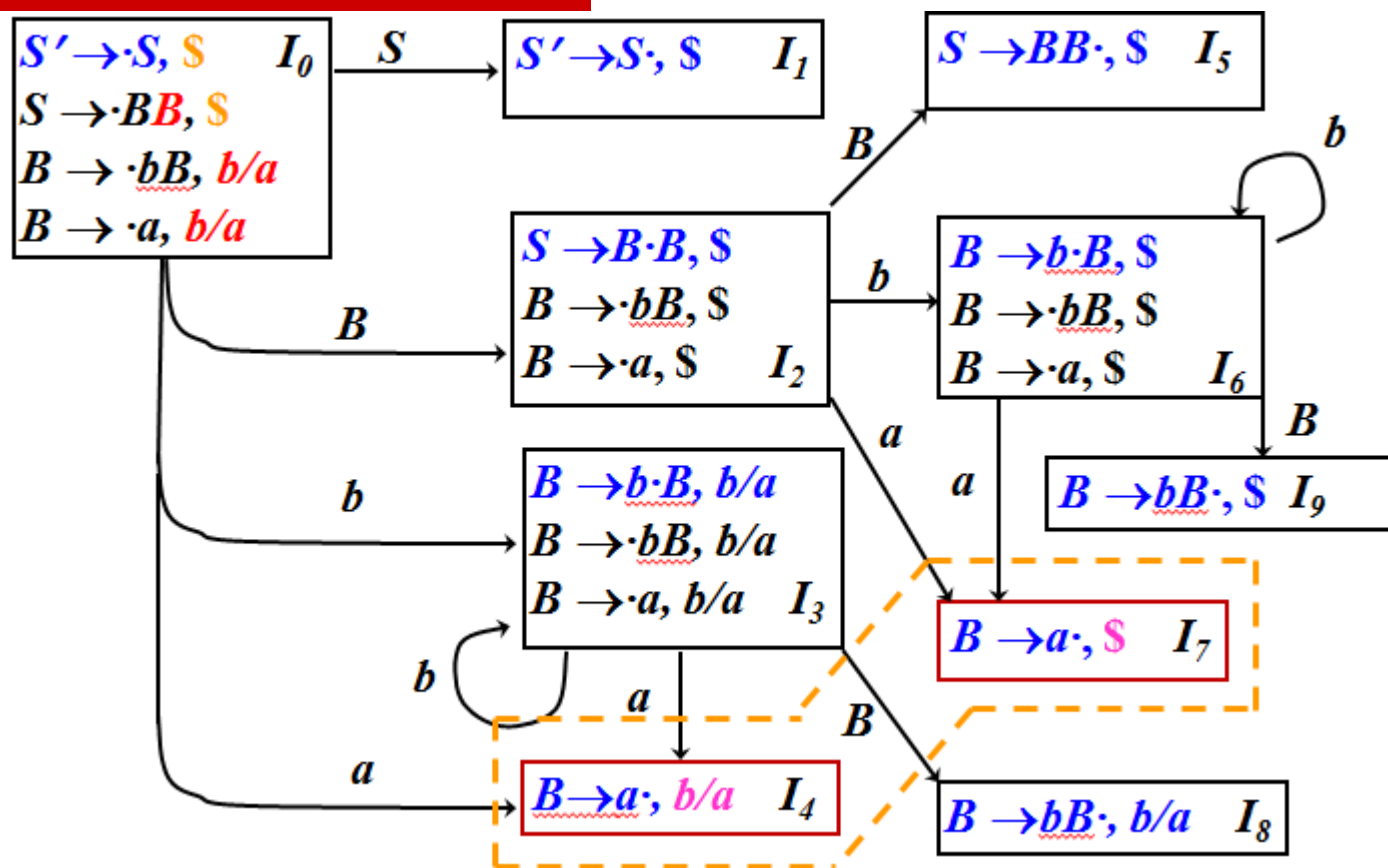


$I_4$ 和 $I_7$  是同心集,合并

输入

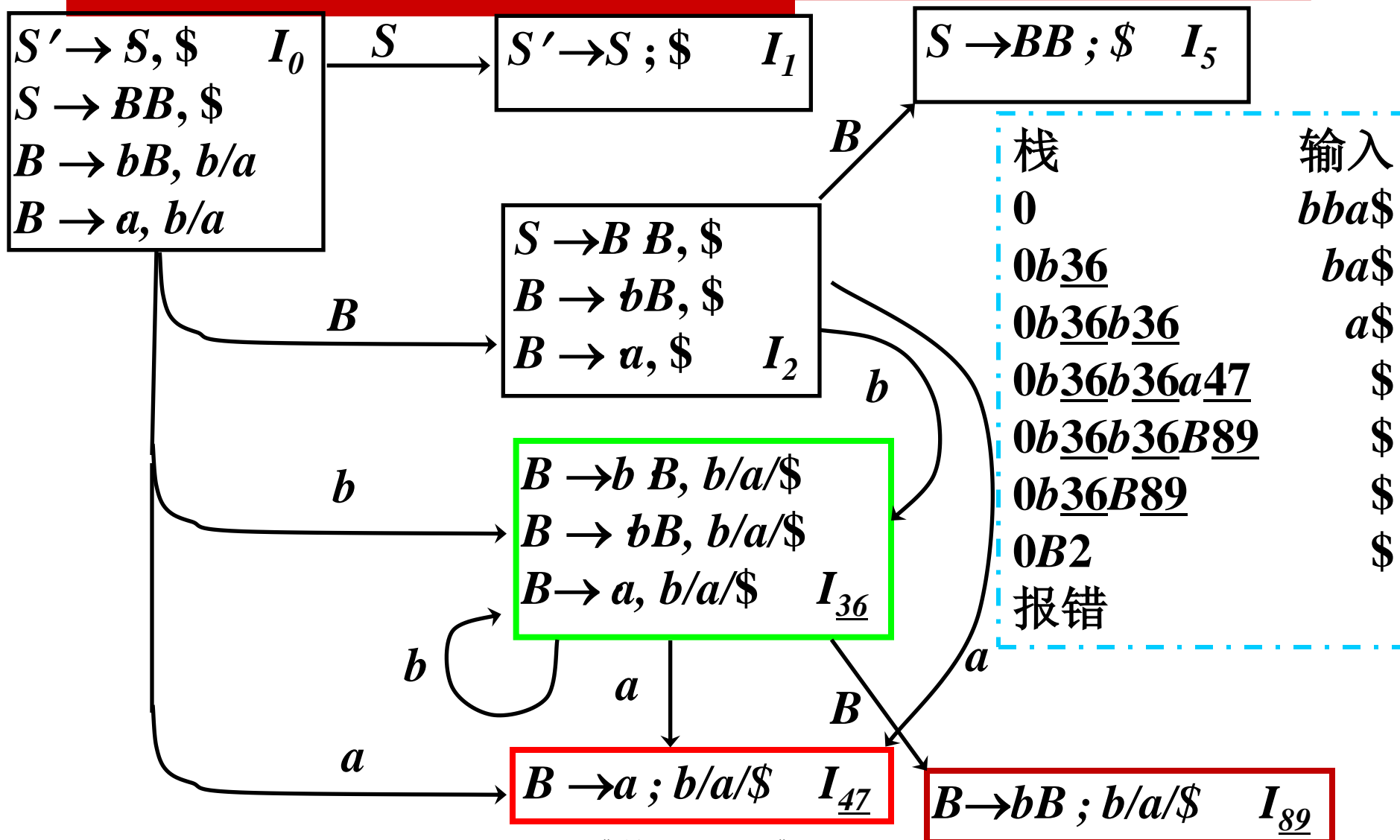
bbabba\$

bba\$





# 合并同心项目集





# LALR(1)分析表的构造

1. 构造LR(1)项目集规范族  $C = \{I_0, I_1, \dots, I_n\}$
2. 寻找LR(1)项目集规范族中同心的项目集，用它们的并集代替它们
3. 按构造规范LR(1)分析表的方式构造分析表



# LALR vs. LR

## □ 同心的LR(1)项目集

- 两个项目集在略去搜索符后是相同的集合

## □ 同心集的合并不会引起新的移进-归约冲突

项目集1

$[A \rightarrow \alpha ; a]$

...

项目集2

$[B \rightarrow \beta a \gamma, b]$

...

若合并后有冲突



# LALR vs. LR

## □ 同心的LR(1)项目集

- 两个项目集在略去搜索符后是相同的集合

## □ 同心集的合并不会引起新的移进-归约冲突

项目集1

$[A \rightarrow \alpha ; a]$

$[B \rightarrow \beta a \gamma, c]$

...

项目集2

$[B \rightarrow \beta a \gamma, b]$

$[A \rightarrow \alpha ; d]$

...

则合并前就有冲突



# LALR vs. LR

## □ 同心的LR(1)项目集

■ 两个项目集在略去搜索符后是相同的集合

## □ 同心集的合并不会引起新的移进-归约冲突

## □ 同心集的合并有可能产生新的归约-归约冲突

$S' \rightarrow S$

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

$A \rightarrow c$

$B \rightarrow c$

对 $ac$ 有效的项目集

$A \rightarrow c ; d$   
 $B \rightarrow c ; e$

对 $bc$ 有效的项目集

$A \rightarrow c ; e$   
 $B \rightarrow c ; d$

合并同心集之后

$A \rightarrow c ; d/e$   
 $B \rightarrow c ; d/e$

该文法是LR(1)的,  
但不是LALR(1)的



# 非LR的上下文无关结构

若自左向右扫描的移进-归约分析器能及时识别出现在栈顶的句柄，那么相应的文法就是LR的。

语言  $L = \{ww^R \mid w \in (a \mid b)^*\}$  的文法

$$S \rightarrow aSa \mid bSb \mid \varepsilon$$

不是LR的

*ababb**bb**aba*

语言  $L = \{w\mathbf{c}w^R \mid w \in (a \mid b)^*\}$  的文法

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

是LR的

*ababb**c**bbaba*

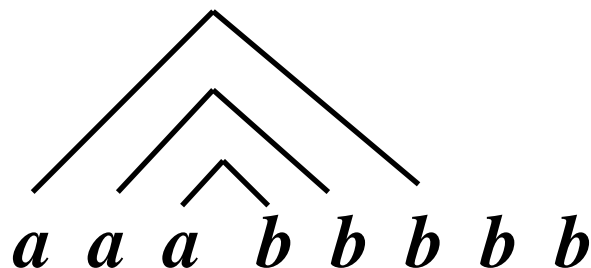




# 例题 写不同的文法

为语言  $L = \{ a^m b^n \mid n > m \geq 0 \}$  写三个文法, 它们分别是LR(1)的、二义的和非二义且非LR(1)的。

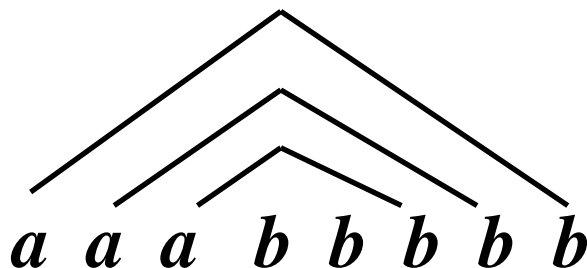
□ LR(1)文法:  $S \rightarrow AB$      $A \rightarrow aAb \mid \varepsilon$      $B \rightarrow Bb \mid b$





为语言  $L = \{ a^m b^n \mid n > m \geq 0 \}$  写三个文法, 它们分别是 LR(1) 的、二义的和非二义且非 LR(1) 的。

❑ 非二义且非LR(1)的文法:  $S \rightarrow aSb \mid B \quad B \rightarrow Bb \mid b$

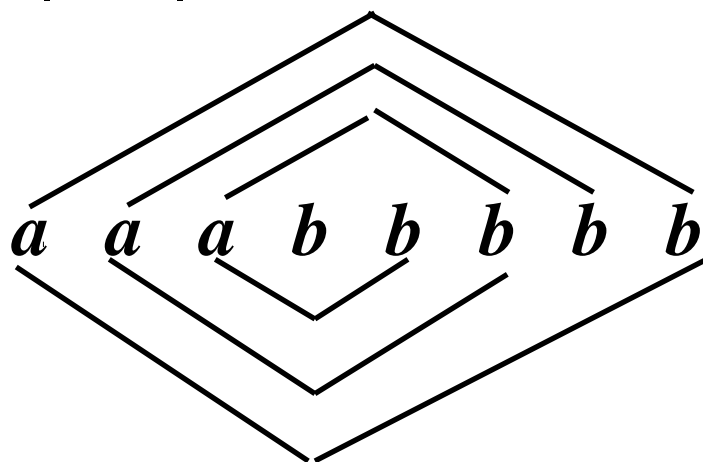




# 例题 写不同的文法

为语言  $L = \{ a^m b^n \mid n > m \geq 0 \}$  写三个文法, 它们分别是LR(1)的、二义的和非二义且非LR(1)的。

- LR(1)文法:  $S \rightarrow \textcolor{red}{A}\textcolor{blue}{B} \quad A \rightarrow aAb \mid \varepsilon \quad B \rightarrow Bb \mid b$
- 非二义且非LR(1)的文法:  $S \rightarrow a\textcolor{red}{S}b \mid \textcolor{blue}{B} \quad B \rightarrow Bb \mid b$
- 二义的文法:  $S \rightarrow a\textcolor{blue}{S}b \mid Sb \mid b$



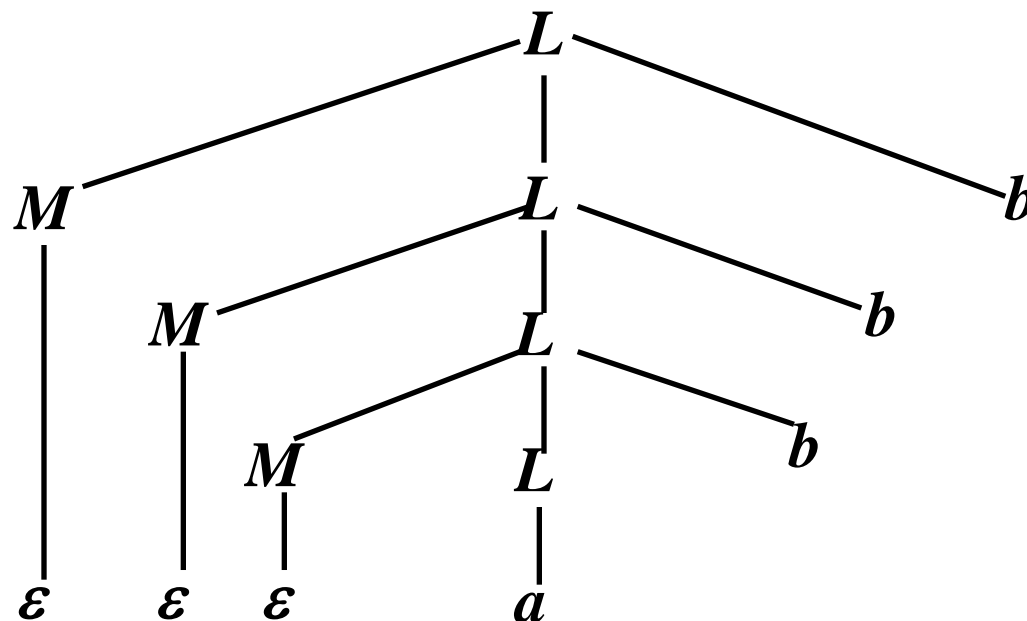


## 例题4

试说明下面文法不是LR(1)的：

$$L \rightarrow M L b \mid a$$

$$M \rightarrow \varepsilon$$



面临 $a$ 时，不知道该  
做多少次空归约 $M \rightarrow \varepsilon$

句子 $abbbb$ 的分析树



## 例题5

下面的文法不是LR(1)的，对它略做修改，使之成为一个等价的SLR(1)文法

*program*  $\rightarrow$  *begin declist ; statement end*

*declist*  $\rightarrow$  *d ; declist* | *d*

*statement*  $\rightarrow$  *s ; statement* | *s*

该文法产生的句子的形式是

*begin d ; d ; ... ; d ; s ; s ; ... ; s end*

修改后的文法如下：

*program*  $\rightarrow$  *begin declist statement end*

*declist*  $\rightarrow$  *d ; declist* | *d ;*

*statement*  $\rightarrow$  *s ; statement* | *s*



## 例题6

一个C语言的文件如下，第四行的if误写成fi：

```
long gcd(p,q)
long p,q;
{
    fi (p%q == 0)
        return q;
    else
        return gcd(q, p%q);
}
```

基于LALR (1) 方法的一个编译器的报错情况如下：

parse error before ‘return’ ( line 5).

是否违反了LR分析的活前缀性质？



中国科学技术大学  
University of Science and Technology of China

## 3.6 二义文法的应用



# 二义文法的特点

## □ 特点

- 绝不是LR 文法
- 简洁、自然

例 二义文法  $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$

非二义的文法:

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

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

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

单非产生式会增加  
分析树的高度  
 $\Rightarrow$  分析效率降低

该文法有单个非终结符为右部的产生式





# 二义文法的消除

## □ 特点

- 绝不是LR 文法
- 简洁、自然
- 可以用文法以外的信息来消除二义

例 二义文法  $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$

规定：\*优先级高于+，两者都是左结合

LR(0)项目集 $I_7$

$E \rightarrow E + E \cdot$

id + id

+ id

$E \rightarrow E + E$

$E \rightarrow E * E$

面临+, 归约



# 二义文法的消除

## □ 特点

- 绝不是LR 文法
- 简洁、自然
- 可以用文法以外的信息来消除二义

例 二义文法  $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$

规定：\*优先级高于+，两者都是左结合

LR(0)项目集 $I_7$

$E \rightarrow E + E \cdot$

id + id                      + id

$E \rightarrow E + E$

id + id                      \* id

$E \rightarrow E * E$

面临+，归约

面临\*，移进

面临)和\$，归约



# 二义文法的消除

## □ 特点

- 绝不是LR 文法
- 简洁、自然
- 可以用文法以外的信息来消除二义

例 二义文法  $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$

规定：\*优先级高于+，两者都是左结合

LR(0)项目集 $I_8$

$E \rightarrow E * E \cdot$

$\text{id} * \text{id} \quad \quad + \text{id}$

$E \rightarrow E + E$

$E \rightarrow E * E$

面临+, 归约



# 二义文法的消除

## □ 特点

- 绝不是LR 文法
- 简洁、自然
- 可以用文法以外的信息来消除二义

例 二义文法  $E \rightarrow E + E \mid E * E \mid (E) \mid \text{id}$

规定: \*优先级高于+, 两者都是左结合

LR(0)项目集 $I_8$

$E \rightarrow E * E \cdot$

id \* id                      + id

$E \rightarrow E + E$

id \* id                      \* id

$E \rightarrow E * E$

面临+, 归约

面临\*, 归约

面临)和\$, 归约



# 特殊情况引起的二义性

$$E \rightarrow E \text{ sub } E \text{ sup } E$$

$$E \rightarrow E \text{ sub } E$$

$$E \rightarrow E \text{ sup } E$$

$$E \rightarrow \{E\}$$

$$E \rightarrow c$$

从定义形式语言的角度说，第一个产生式是多余的  
但联系到语义处理，第一个产生式是必要的  
对  $a \text{ sub } i \text{ sup } 2$ ，需要下面第一种输出

$$a_i^2 \quad a_i^2 \quad a_{i^2}$$



# 特殊情况引起的二义性

$E \rightarrow E \text{ sub } E \text{ sup } E$

$E \rightarrow E \text{ sub } E$

$E \rightarrow E \text{ sup } E$

$E \rightarrow \{E\}$

$E \rightarrow c$

$I_{11}$ :

$E \rightarrow E \text{ sub } E \text{ sup } E \cdot$

$E \rightarrow E \text{ sub } E \cdot$

...

按前面一个产生式归约



# LR分析的错误恢复

- LR分析器在什么情况下发现错误
  - 访问action表时遇到出错条目
  - 访问goto表时绝不会遇到出错条目
  - 绝不会把不正确的后继移进栈
  - 规范的LR分析器在报告错误之前决不做任何无效归约
    - SLR和LALR在报告错误前有可能执行几步无效归约

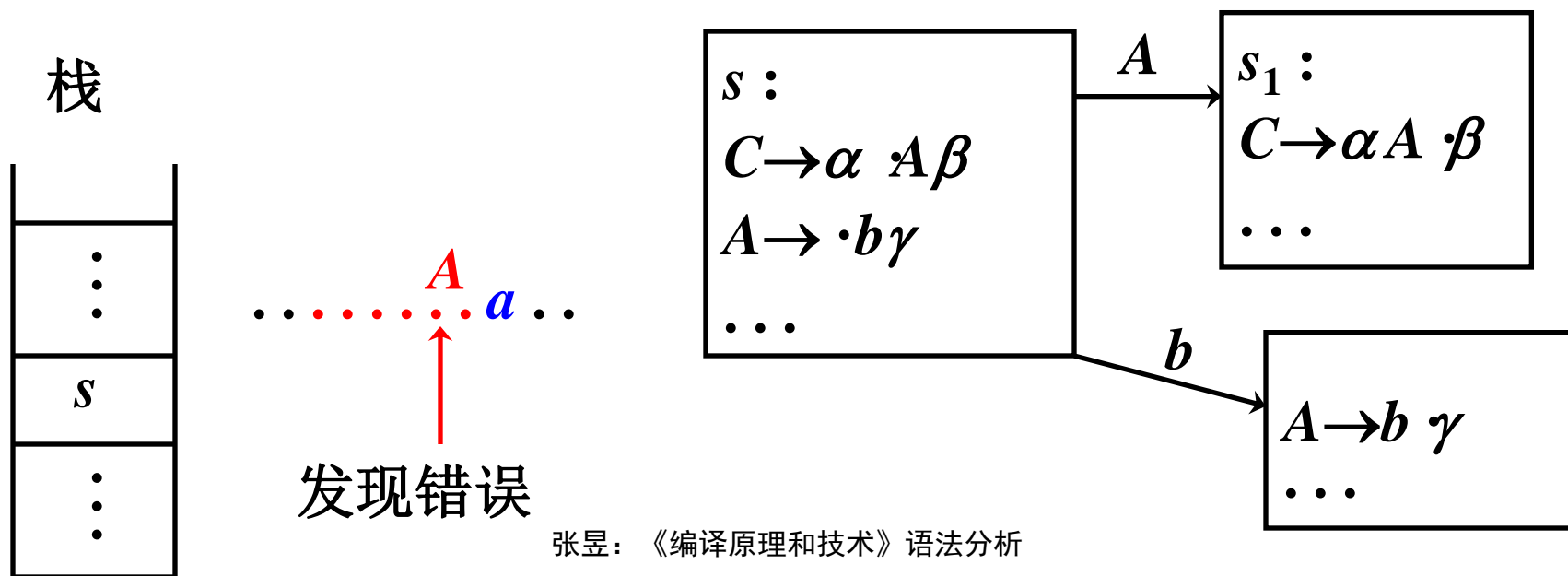


# 紧急方式的错误恢复

## □ 错误恢复策略

■ 试图忽略含语法错误的短语： $A$ 推出的串含错误

1. 退栈，直至出现状态  $s$ ，它有预先确定的  $A$  的转移





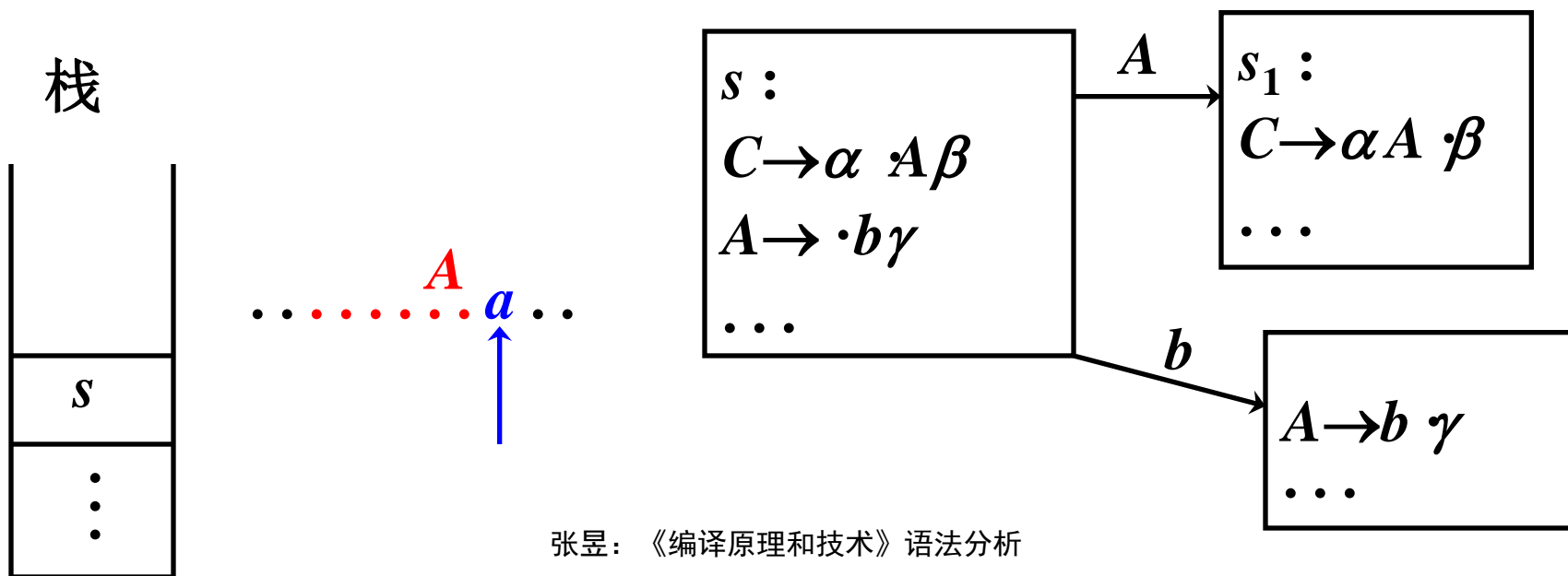


# 紧急方式的错误恢复

## □ 错误恢复策略

■ 试图忽略含语法错误的短语： $A$ 推出的串含错误

1. 退栈，直至出现状态  $s$ ，它有预先确定的  $A$  的转移
2. 抛弃若干输入符号，直至找到  $a$ ，它是  $A$  的合法后继



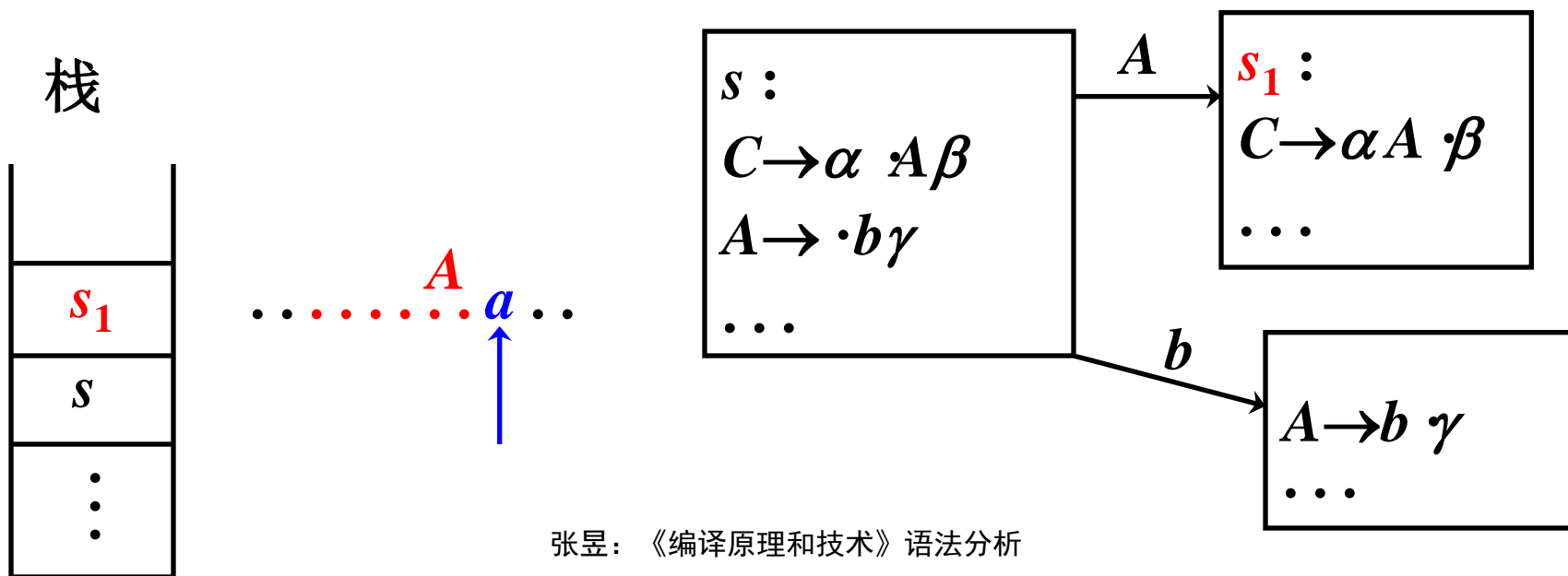


# 紧急方式的错误恢复

## □ 错误恢复策略

■ 试图忽略含语法错误的短语： $A$ 推出的串含错误

1. 退栈，直至出现状态  $s$ ，它有预先确定的  $A$  的转移
2. 抛弃若干输入符号，直至找到  $a$ ，它是  $A$  的合法后继
3. 再把  $A$  和状态  $goto[s, A]$  压进栈，恢复正常分析





# 短语级恢复

## □ 短语级恢复

- 发现错误时，对剩余输入作局部纠正

如用分号代替逗号, 删除多余的分号, 插入遗漏的分号

缺点：难以解决实际错误出现在诊断点以前的情况

- 实现方法

在action表的每个空白条目填上指示器，指向错误处理例程



中国科学技术大学  
University of Science and Technology of China

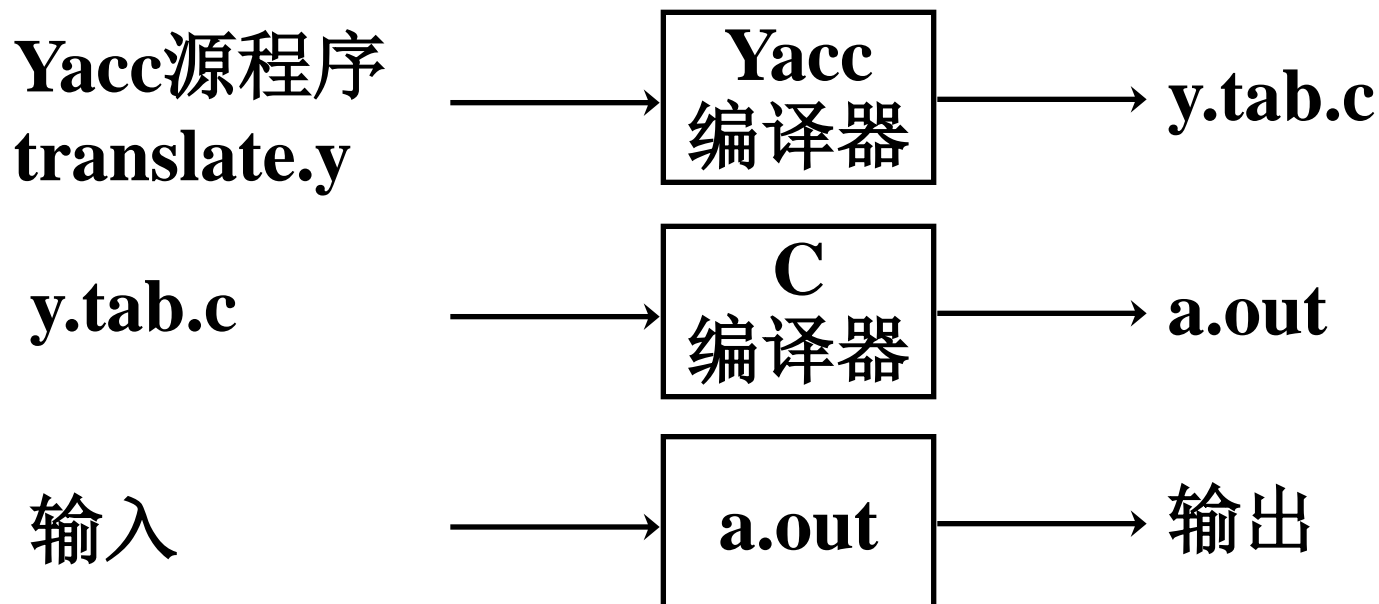
## 3.7 分析器的生成器

□ YACC



# YACC

## □ YACC (Yet Another Compiler Compiler)





# 例 简单计算器

- 输入一个表达式并回车，显示计算结果
- 也可以输入一个空白行

## 声明部分

```
%{  
# include <ctype.h>  
# include <stdio.h>  
# define YYSTYPE double /*将栈定义为double类型*/  
%}  
  
%token NUMBER  
%left '+' '-'  
%left '*' '/'  
%right UMINUS  
%%
```



# 例 简单计算器

## 翻译规则部分

```
lines      : lines expr '\n'    {printf ( "%g \n", $2 ) }  
           | lines '\n'  
           | /* ε */  
           ;  
  
expr       : expr '+' expr      { $$ = $1 + $3; }  
           | expr '-' expr      { $$ = $1 - $3; }  
           | expr '*' expr      { $$ = $1 * $3; }  
           | expr '/' expr      { $$ = $1 / $3; }  
           | '(' expr ')'        { $$ = $2; }  
           | '-' expr %prec UMINUS { $$ = -$2; }  
           | NUMBER  
           ;
```

%%



# 例 简单计算器

## 翻译规则部分

```
lines      : lines expr '\n'    {printf ( "%g \n", $2 ) }  
           | lines '\n'  
           | /* ε */  
           ;  
  
expr       : expr '+' expr      { $$ = $1 + $3; }  
           | expr '-' expr      { $$ = $1 - $3; }  
           | expr '*' expr      { $$ = $1 * $3; }  
           | expr '/' expr      { $$ = $1 / $3; }  
           | '(' expr ')'       { $$ = $2; }  
           | '-' expr %prec UMINUS { $$ = -$2; }  
           | NUMBER  
           ;
```

%%

-5+10看成是-(5+10), 还是(-5)+10? 取后者





# 例 简单计算器

## C例程部分

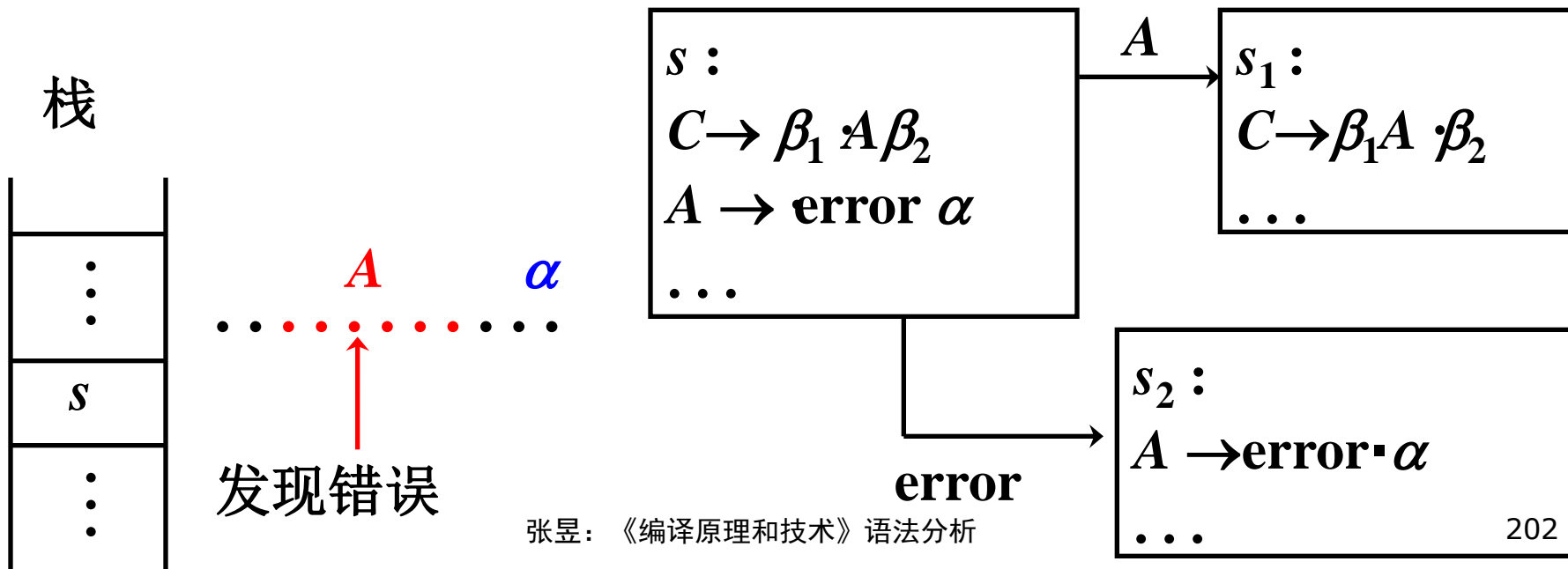
```
yylex ( ) {  
    int c;  
    while ( ( c = getchar ( ) ) == ' ' );  
    if ( ( c == '.' ) || (isdigit (c) ) ) {  
        ungetc (c, stdin);  
        scanf ( "%lf", &yylval);  
        return NUMBER;  
    }  
    return c;  
}
```

为了C编译器能准确报告yylex函数中错误的位置，  
需要在生成的程序y.tab.c中使用编译命令#line



# YACC的错误恢复

- 增加错误产生式  $A \rightarrow \text{error } \alpha$
- 遇到语法错误时

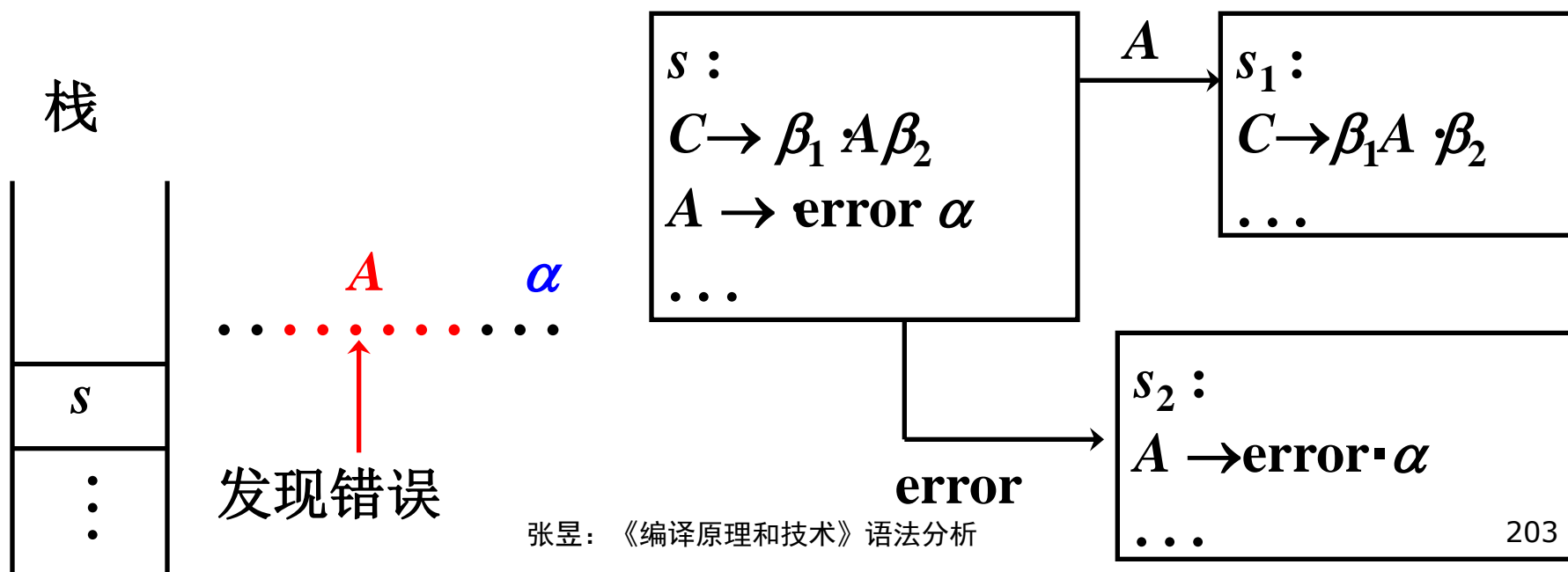




# YACC的错误恢复

## □ 遇到语法错误时

- 从栈中弹出状态，直到发现栈顶状态的项目集包含形为  $A \rightarrow \text{error } \alpha$  的项目为止

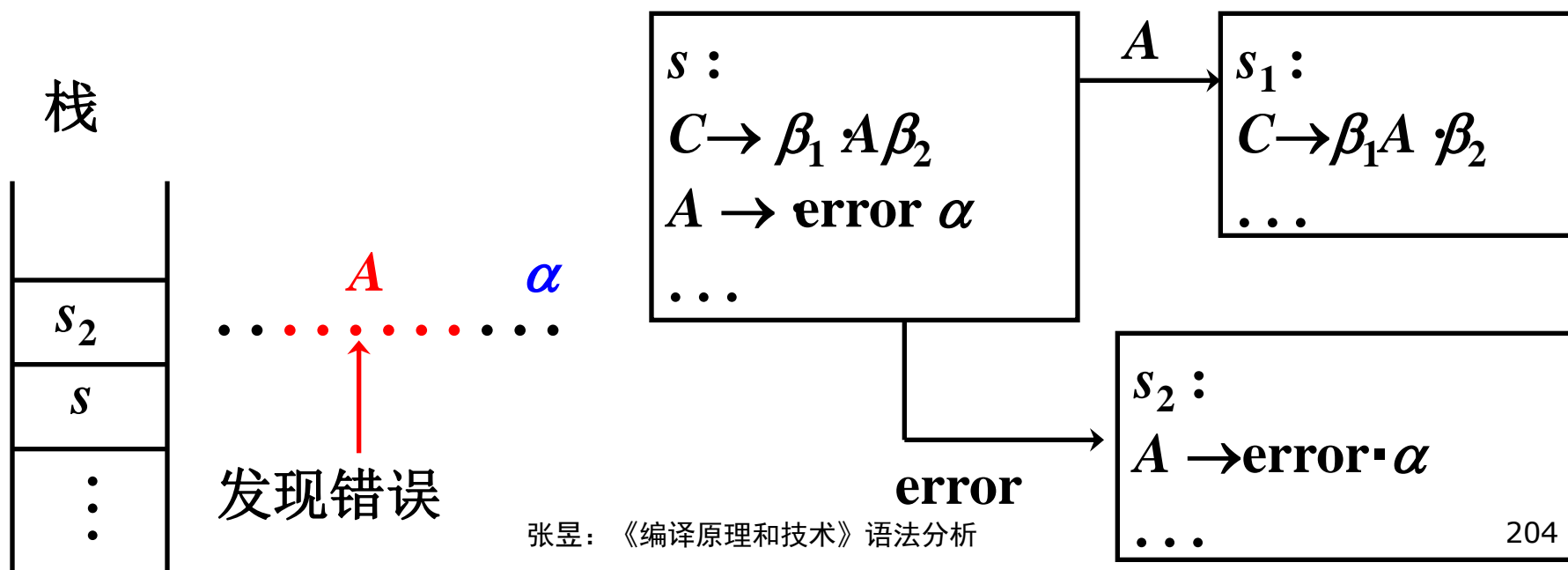




# YACC的错误恢复

## □ 遇到语法错误时

- 从栈中弹出状态，直到发现栈顶状态的项目集包含形为  $A \rightarrow \text{error } \alpha$  的项目为止
- 把虚构的终结符 **error** “移进” 栈

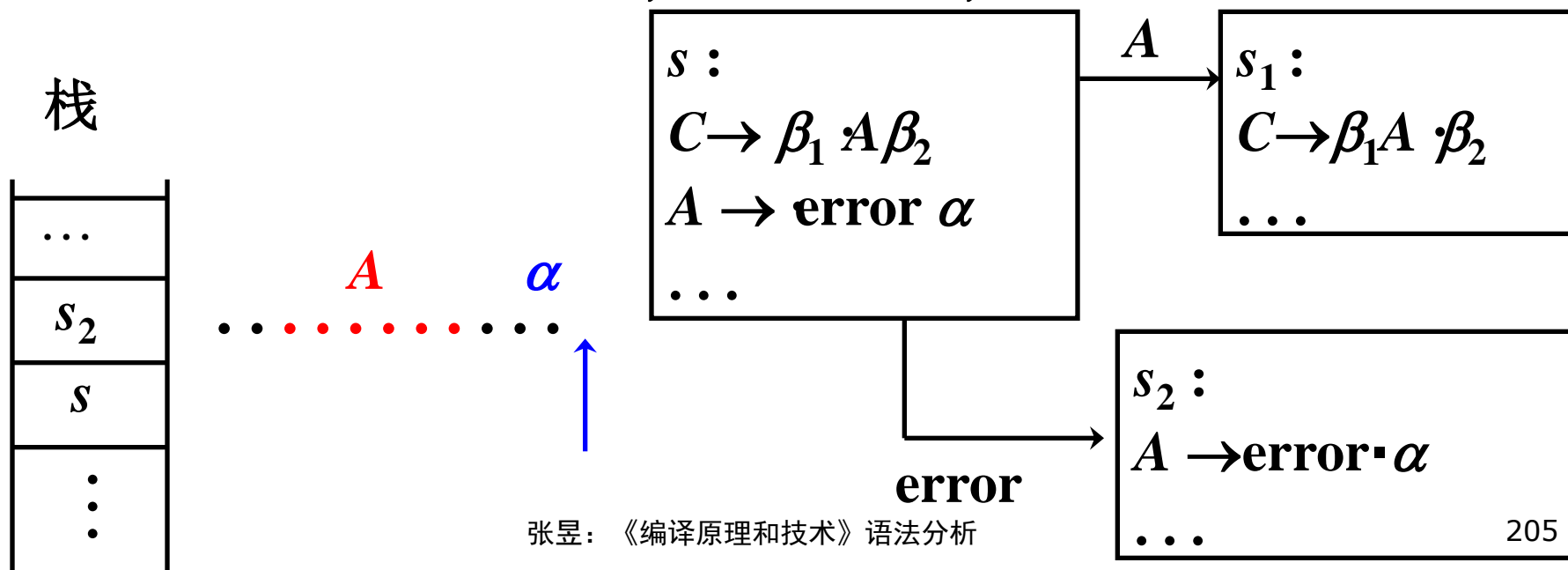




# YACC的错误恢复

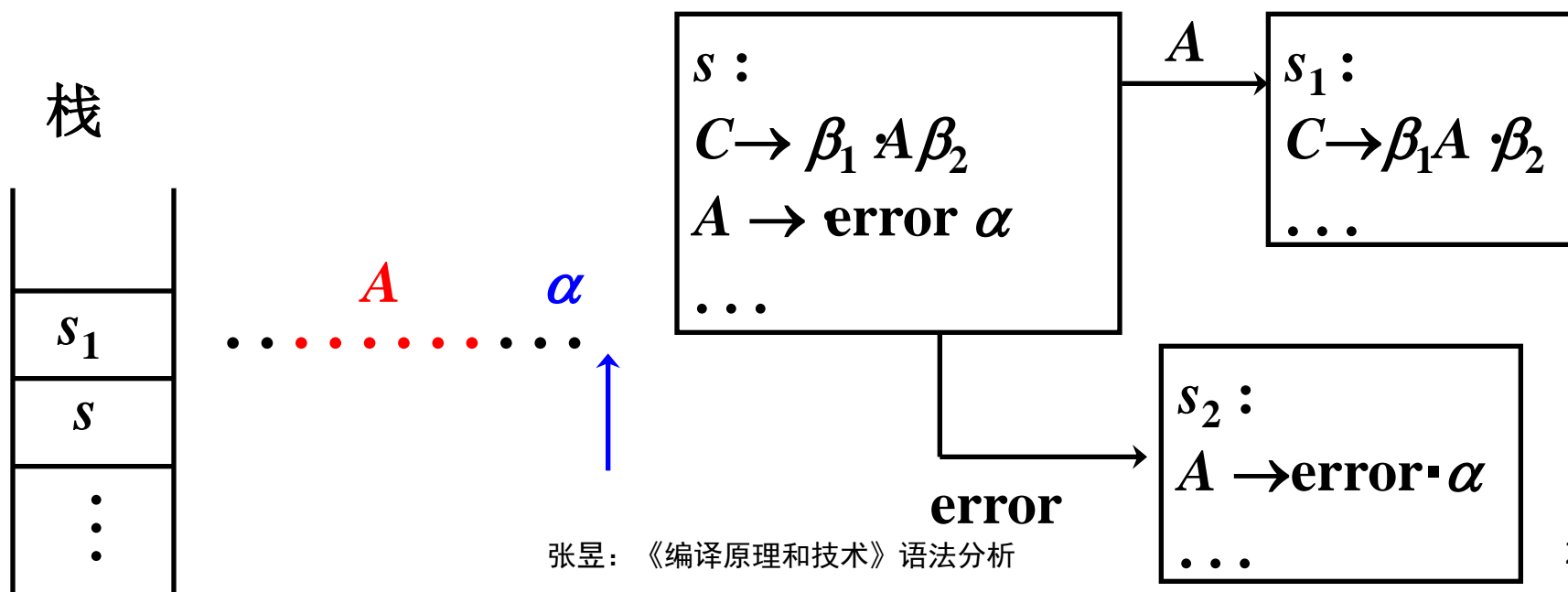
## □ 遇到语法错误时

- 从栈中弹出状态，直到发现栈顶状态的项目集包含形为  $A \rightarrow \text{error } \alpha$  的项目为止
- 把虚构的终结符 **error** “移进” 栈
- 忽略若干输入符号，直至找到  $\alpha$ ，把  $\alpha$  移进栈



# YACC的错误恢复

- 从栈中弹出状态，直到发现栈顶状态的项目集包含形为  $A \rightarrow \text{error } \alpha$  的项目为止
- 把虚构的终结符 **error** “移进” 栈
- 忽略若干输入符号，直至找到  $\alpha$ ，把  $\alpha$  移进栈
- 把 **error**  $\alpha$  归约为  $A$ ，恢复正常分析





# 例 简单计算器

## □ 增加错误恢复的简单计算器

```
lines      : lines expr '\n'      {printf ( "%g \n", $2 ) }  
| lines '\n'  
| /*  $\epsilon$  */  
| error '\n'{yyerror ( “重新输入上一行” );  
               yyerrok;}  
;
```