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》

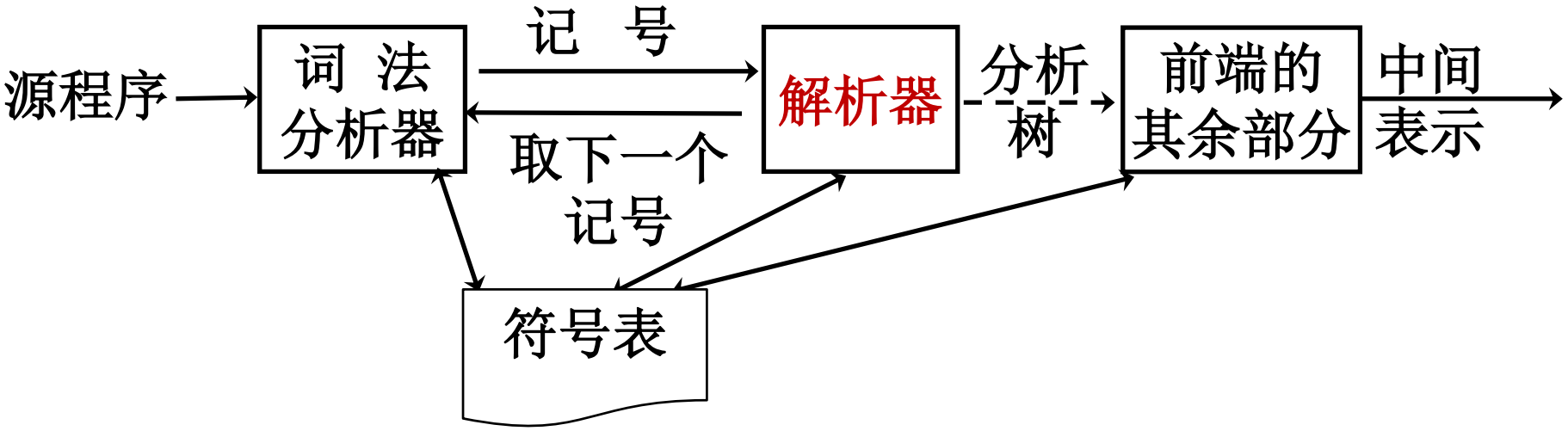
张昱

0551-63603804，yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

本章内容

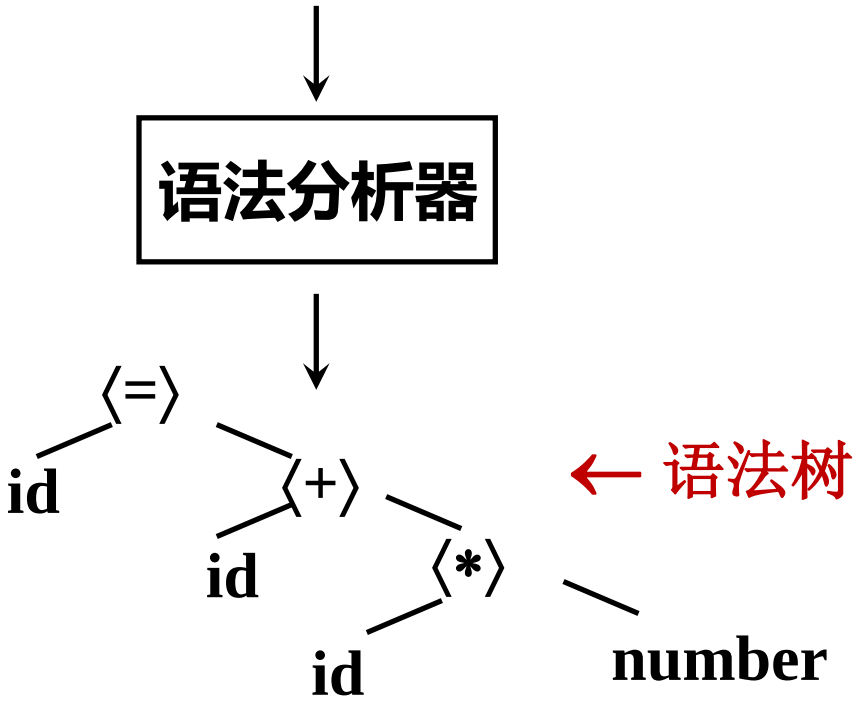


- 语法的形式描述：上下文无关文法
- 语法分析：自上而下、自下而上
- 语法分析器(parser、syntax analyzer)的自动生成
  - LL(k)、LL(*)、SLR(k)、LR(k)、LALR(k)

语法分析器

- **语法树(syntax tree)**
  - 源程序的**层次化**语法结构
  - 是程序的一种**中间表示**

⟨id, 1⟩ ⟨=⟩ ⟨id, 2⟩ ⟨+⟩ ⟨id, 3⟩ ⟨*⟩ ⟨60⟩ ←记号流



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

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：$\{wcw \mid w$是由a和b组成的串$\}$

  原因：n不固定，后面的串要依据前面不定长的串w来确定；

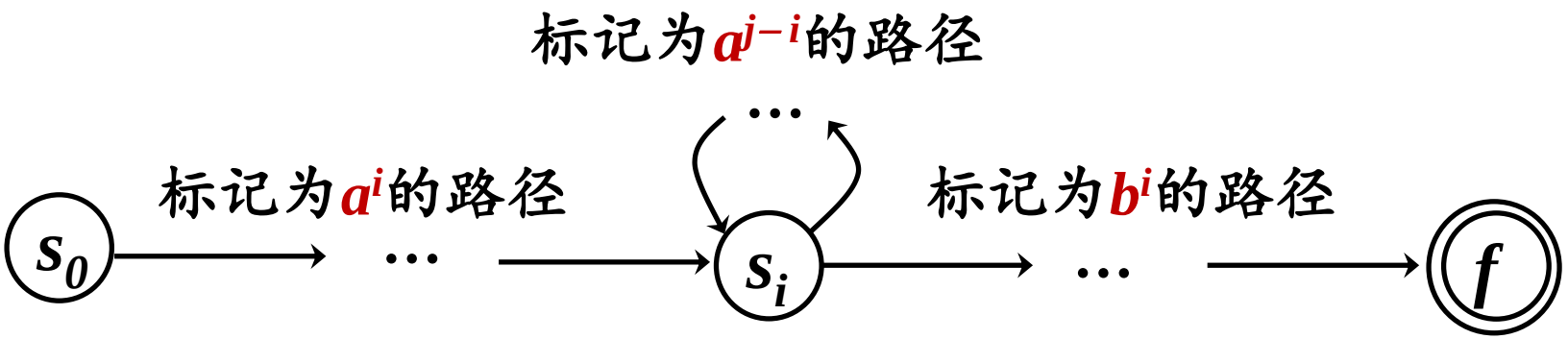
  有限自动机无法记录访问同一状态的次数

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+1个（有限个）
- 设 D 读完$\varepsilon, a, aa, \ldots, a^k$ 分别到达状态$s_0, s_1, \ldots, s_k$
- 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



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, 记号名，即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：$A \to \alpha$，　$A \in V_N, \alpha \in (V_T \cup V_N)^*$

  有时用 $A ::= \alpha$

  - 例 ({id, +, *, –, (,)}, {*expr*, *op*}, *expr*, *P*)

    expr → *expr op expr*　　*expr* → (*expr*)　　*expr* → – *expr*

    expr → id　　*op* → +　　*op* → *

CFG的简化表示

表达式的CFG的简化表示

- 引入选择符 |

$expr \rightarrow expr\ op\ expr \mid (expr) \mid -\ expr \mid \mathbf{id}$

$op \rightarrow + \mid *$

注：+, *是*op*的选择(alternatives)

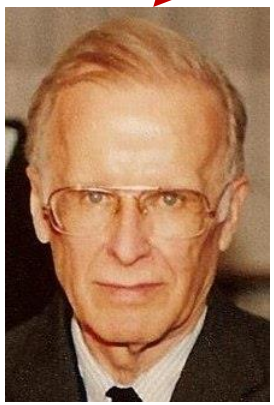
- 简化名称

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

$A \rightarrow + \mid *$

Backus–Naur Form

巴科斯-诺尔范式



John Backus

1977图灵奖

首次在ALGOL 58中实现BNF

Peter Naur

2005图灵奖

在ALGOL 60中发展和简化

图灵奖获得者

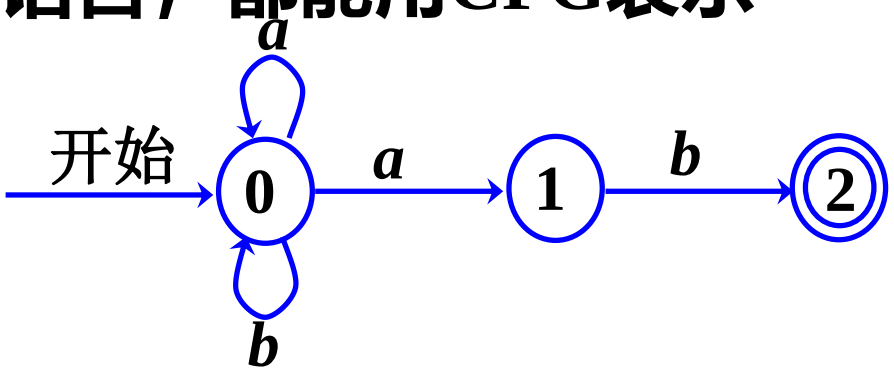
- John Backus (1924-2007)
  - 1977图灵奖获奖成就
    - FORTRAN发明组组长
    - 提出了BNF
  - 履历
    - 弗吉尼亚大学化学(因出勤率低而开除，随和入伍)
    - 哥伦比亚大学数学BS 1949, AM 1950
    - IBM
  - 获奖演说： Can Programming Be Liberated From the von Neumann Style?
- Peter Naur (1928-2016)
  - 2005图灵奖获奖成就
    - ALGOL 60
    - 发展BNF并简化
  - 履历
    - 哥本哈根大学天文学BS 1949、博士1957
    - 1950-51剑桥：天气恶劣破坏天文观测计划，但花很多时间编程以解决天文学中的扰动问题
    - 毕业后转向计算机科学
    - 获奖演说： Computing vs. Human Thinking

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，NFA中每条弧对应于产生式的一个分支（选项），

对于接受状态i，则引入$A_i \to \varepsilon$

$A_0 \to a\,A_0 \mid b\,A_0 \mid a\,A_1$

$A_1 \to b\,A_2$

$A_2 \to \varepsilon$　(该产生式并不必要)

CFG：推导(derivation)

推导

- 是从文法推出文法所描述的语言中**合法串**集合的动作
- 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 $E \to E+E \mid E*E \mid (E) \mid -E \mid \mathrm{id}$

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

上述代换序列称为从 E 到 -(id+id) 的推导

-(id+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 $S \Rightarrow^+ w$

思考

- 上下文无关是什么意思?

思考

- **上下文无关是什么意思?**
  - 指对于文法推导的每一步 $\alpha A \beta \Rightarrow \alpha \gamma \beta$

文法符号串γ仅依据A的产生式推导，而无需依赖A的上下文α和β

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，α是由终结符和/或非终结符组成的串，则α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 $E \to E + E \mid E * E \mid (E) \mid -E \mid \text{id}$

- 最左推导(leftmost derivation)

  每步代换最左边的非终结符

  $E \Rightarrow_{lm} -E \Rightarrow_{lm} -(E) \Rightarrow_{lm} -(E + E)$

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

- 最右推导（rightmost or canonical，规范推导)

  每步代换最右边的非终结符

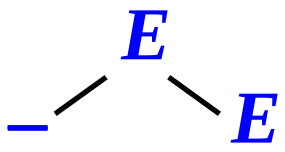
  $E \Rightarrow_{rm} -E \Rightarrow_{rm} -(E) \Rightarrow_{rm} -(E + E)$

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

分析树(parse tree)

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

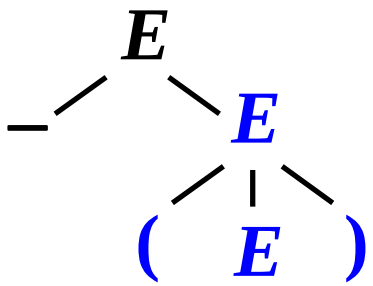
-(id+id)最左推导的分析树（parse tree)



分析树(parse tree)

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

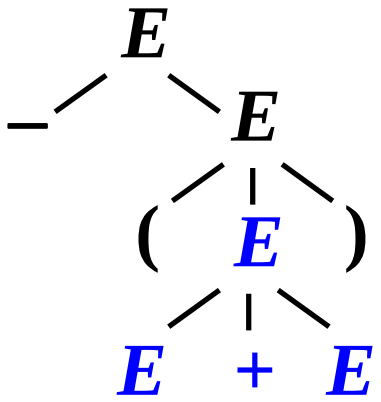
-(id+id)最左推导的分析树（parse tree)



分析树(parse tree)

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

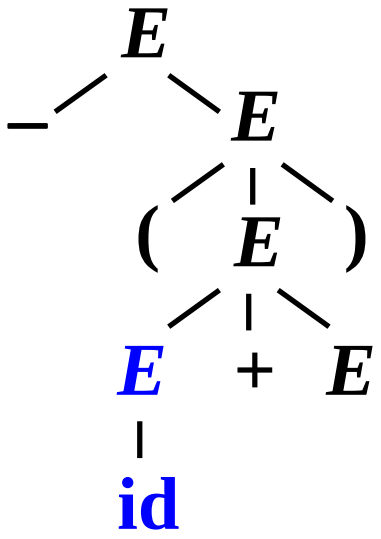
-(id+id)最左推导的**分析树（parse tree）**



分析树(parse tree)

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

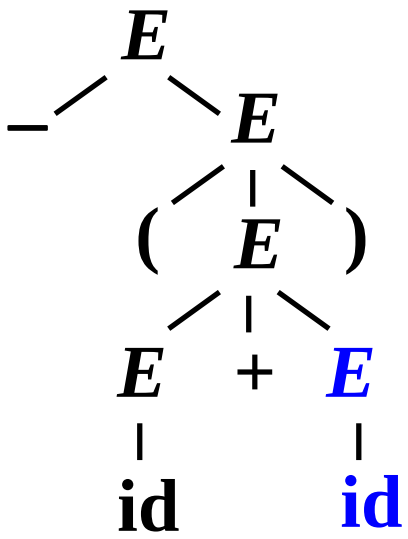
-(id+id)最左推导的分析树（parse tree)



分析树(parse tree)

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

-(id+id)最左推导的分析树（parse tree)



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 $E \to E+E \mid E*E \mid (E) \mid -E \mid \mathrm{id}$

id*id+id 有两个不同的最左推导

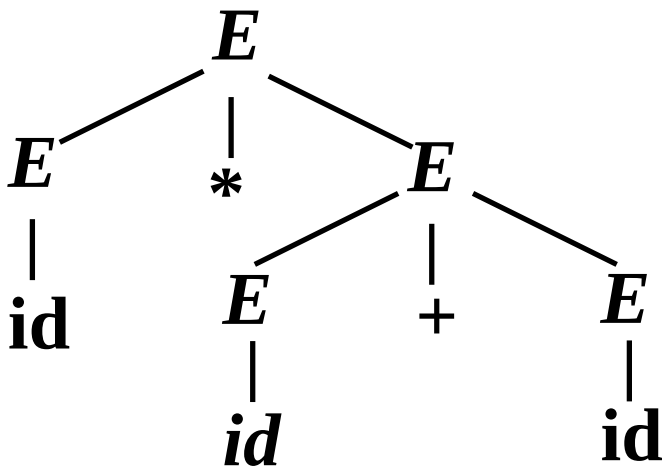
$$
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow \mathrm{id}*E \\
&\Rightarrow \mathrm{id}*E+E \\
&\Rightarrow \mathrm{id}*\mathrm{id}+E \\
&\Rightarrow \mathrm{id}*\mathrm{id}+\mathrm{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow E*E+E \\
&\Rightarrow \mathrm{id}*E+E \\
&\Rightarrow \mathrm{id}*\mathrm{id}+E \\
&\Rightarrow \mathrm{id}*\mathrm{id}+\mathrm{id}
\end{aligned}
$$

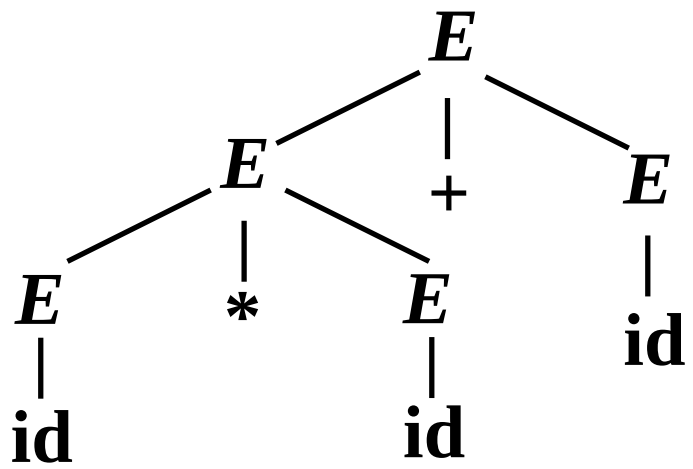
文法的二义性

id*id+id 有两棵不同的分析树

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

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





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

3.2 语言和文法

- 语言和文法：验证、消除二义
- 词法分析和语法分析的分离
- 非上下文无关文法

验证文法产生的语言

$G: S \rightarrow$ '('S ')' $S \mid \varepsilon$　$L(G)$ = 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：

$$S \Rightarrow \text{'('}S\text{')'}\ S \Rightarrow^* \text{'('}\ x\ \text{')'}\ S \Rightarrow^* \text{'('}\ x\ \text{')'}\ y$$

验证文法产生的语言

$G: S \rightarrow$ '('S ')' $S \mid \varepsilon$　$L(G)$ = 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

- 无二义的文法

  expr → *expr* + *term* | *term*

  term → *term* * *factor* | *factor*

  factor → id | (*expr*)

左递归文法
+ 是自左向右结合

如果改成
expr → *term* + *expr* | *term*
呢？
+ 是自右向左结合

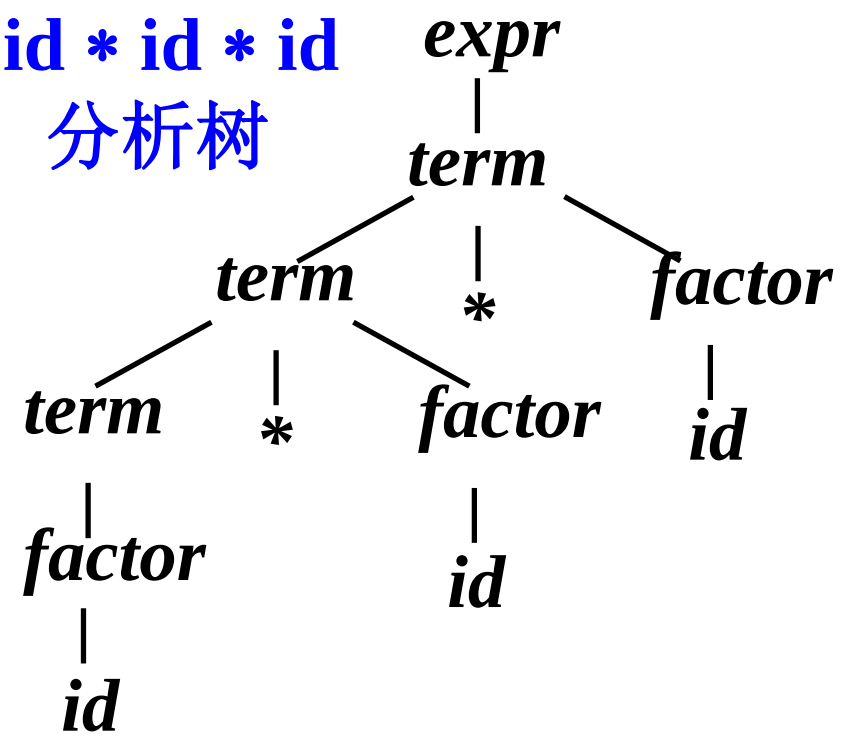
表达式的另一种文法

$expr \rightarrow expr + term \mid term$

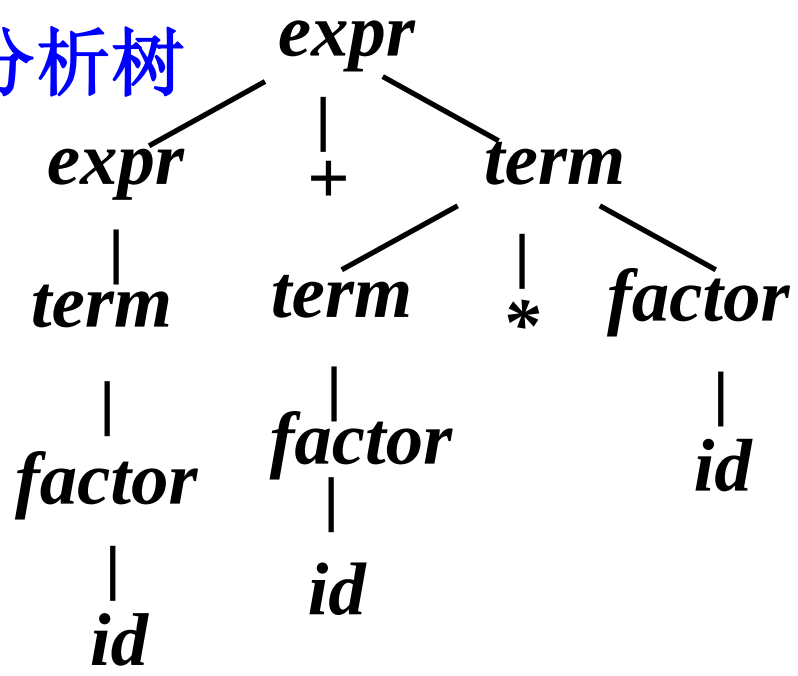
$term \rightarrow term * factor \mid factor$

$factor \rightarrow \mathbf{id} \mid (expr)$

id * id * id 分析树



id + id * id 分析树



消除二义性(Eliminating ambiguity)

stmt → 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 ⇒ if *expr* then *stmt*

  ⇒ if *expr* then if *expr* then *stmt* else *stmt*

  stmt ⇒ if *expr* then *stmt* else *stmt*

  ⇒ if *expr* then if *expr* then *stmt* else *stmt*

消除二义性

- 无二义的文法

stmt → *matched _stmt*
| *unmatched_stmt*

else 的就近匹配规则

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

unmatched_stmt → if *expr* then *stmt*
| if *expr* then *matched_stmt*
else *unmatched_stmt*

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

3.2 语言和文法

- 语言和文法：验证、消除二义
- 词法分析和语法分析的分离
- 非上下文无关文法

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

上下文无关文法的优缺点

- 优点
  - 文法给出了精确的、易于理解的语法说明
  - 可以给语言定义出层次结构
  - 可以基于文法自动产生高效的分析器
  - 以文法为基础实现语言便于对语言修改
- 缺点
  - 表达能力不足够，只能描述编程语言中的大部分语法

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

3.2 语言和文法

- 语言和文法：验证、消除二义
- 词法分析和语法分析的分离
- 非上下文无关文法

非上下文无关的语言构造

$L_1 = \{wcw \mid w$属于$(a \mid b)^*\}$

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

begin：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

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

$$S \to aSa \mid bSb \mid c$$

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

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

$$S \to aSd \mid aAd$$

$$A \to bAc \mid bc$$

$a^n b^n c^n$

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

$$S \to AB$$

$$A \to aAb \mid ab$$

$$B \to cBd \mid cd$$

形式语言鸟瞰

文法 $G = (V_T, V_N, S, P)$

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

  短语文法

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

  上下文有关文法

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

  上下文无关文法

- 3型文法：$A \to aB$或$A \to a$，$A, B \in V_N$，$a \in V_T$

  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

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

$bB \to bb$　　$bC \to bc$　　$cC \to cc$

$a^n b^n c^n$的推导过程如下：

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

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

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

$S \Rightarrow^+ a^n bB^{n-1}C^n$　　用$aB \to ab$　1次

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

$S \Rightarrow^+ a^n b^n cC^{n-1}$　　用$bC \to bc$　1次

$S \Rightarrow^+ a^n b^n c^n$　　用$cC \to cc$　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

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

解答

非二义文法的产生式如下：

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

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

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

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

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

解答

非二义文法的产生式如下：

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

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

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

?

not p and q有两种不同的最左推导

not p and q

not p and q

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：　$S \to a\ S\ b\ S \mid b\ S\ a\ S \mid \varepsilon$

  aabbabab　　aabbabab

- 二义文法：　$S \to a\ B \mid b\ A \mid \varepsilon$

  $A \to a\ S \mid b\ A\ A$

  $B \to b\ S \mid a\ B\ B$

  aabbabab　　aabbabab　　aabbabab

- 非二义文法：　$S \to a\ B\ S \mid b\ A\ S \mid \varepsilon$

  $A \to a \mid b\ A\ A$

  $B \to b \mid a\ B\ B$

  a abb abab

  a B S

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

3.3 自上而下分析

- 自上而下分析、左递归及消除、提左因子
- LL(1)文法和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

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

3.3 自上而下分析

- 自上而下分析、左递归及消除、提左因子
- LL(1)文法和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

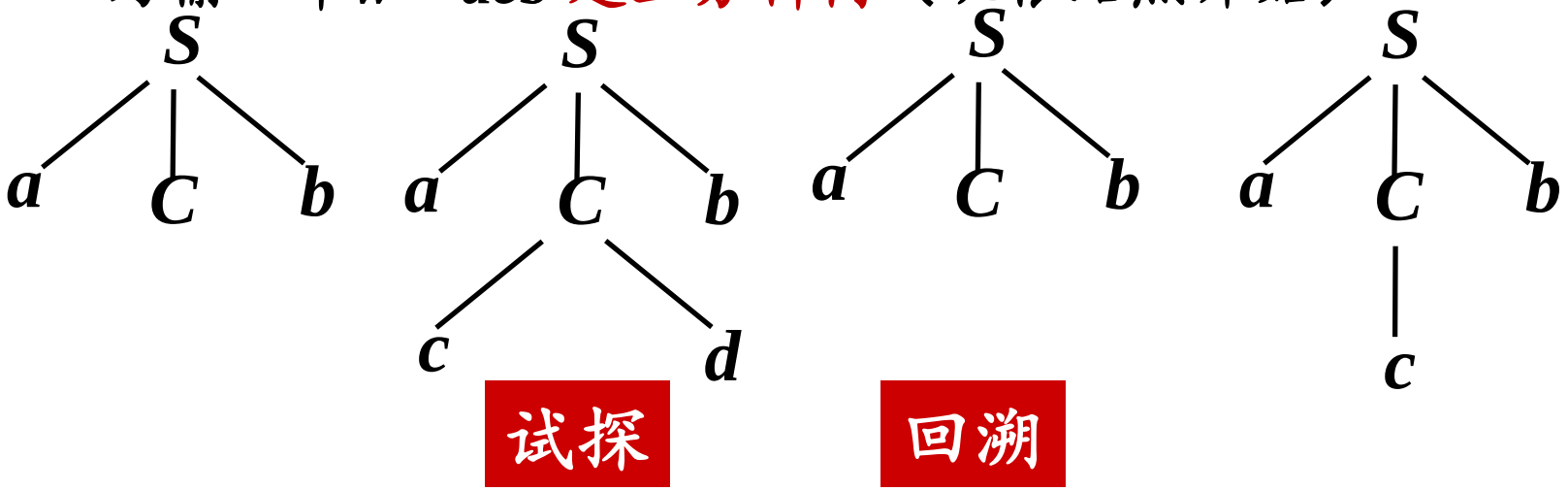
从开始符号出发，为输入串寻找最左推导

试探 产生式的选择 – 失败回溯(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　　$S \to aCb$　　$C \to cd \mid c$

为输入串$w = acb$ 建立分析树（从根结点开始）



自上而下分析：左递归

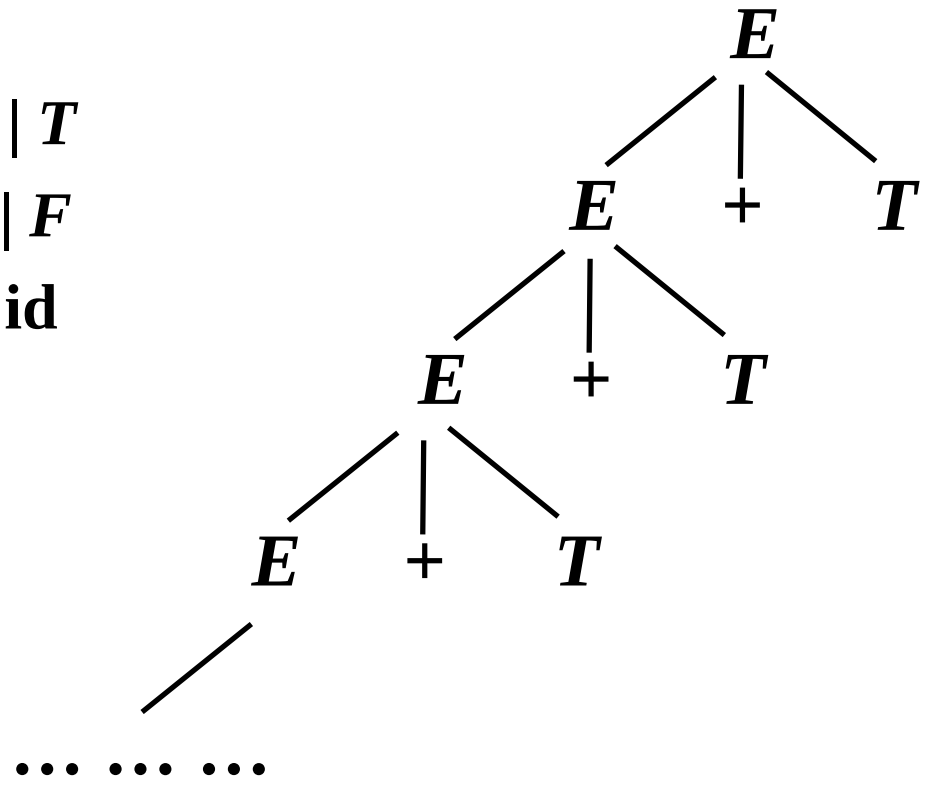
- 文法左递归 $A \Rightarrow^{+} A\alpha$
- 自上而下的分析不能处理左递归文法

算术表达文法

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid \mathrm{id}$



← 分析树

消除左递归(Eliminating left recursion)

- 文法左递归　　$A \Rightarrow^{+} A\alpha$

- 直接左递归 (immediate left recursion)

  $A \rightarrow A\alpha \mid \beta$, 其中$\beta$不以$A$开头

  - 由A推出的串的特点　　$\beta\alpha \ldots \alpha$

- 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

$$E \to E + T \mid T$$

$$T \to T * F \mid F$$

$$F \to (E) \mid \mathrm{id}$$

$$(T + T \ldots + T)$$

$$(F * F \ldots * F)$$

消除左递归后文法

$$E \to TE'$$

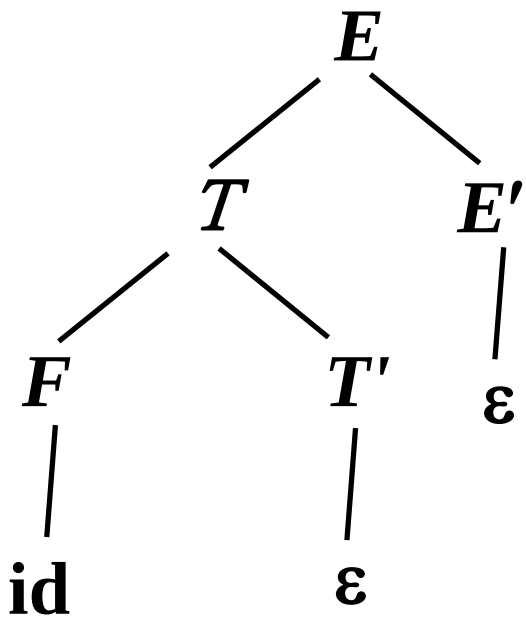
$$E' \to + TE' \mid \varepsilon$$

$$T \to FT'$$

$$T' \to * F T' \mid \varepsilon$$

$$F \to (E) \mid \mathrm{id}$$

自上而下识别id的分析树



消除非直接左递归

- 间接左递归

$$S \to Aa \mid b$$

$$A \to Sd \mid \varepsilon$$

- 先变换成直接左递归

$$S \to Aa \mid b$$

$$A \to Aad \mid bd \mid \varepsilon$$

- 再消除左递归

$$S \to Aa \mid b$$

$$A \to bdA' \mid A'$$

$$A' \to adA' \mid \varepsilon$$

- 隐藏左递归

$$A \to BA$$

$$B \to \varepsilon$$

提左因子(left factoring)

- 有左因子的(left -factored)文法：$A \to \alpha\beta_1 \mid \alpha\beta_2$
  - 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

$$A \to \alpha A' \qquad A' \to \beta_1 \mid \beta_2$$

例 悬空*else*的文法

stmt → **if** *expr* **then** *stmt* **else** *stmt*

| **if** *expr* **then** *stmt* | other

提左因子

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

optional_else_part → **else** *stmt* | ε

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

3.3 自上而下分析

- 自上而下分析、左递归及消除、提左因子
- **LL(1)文法和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

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**
  - **FIRST(α) = $\{a \mid \alpha \Rightarrow^* a\ldots, a \in V_T\}$** ← **文法符号串$\alpha$ 推出的串中首终结符集合**

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

  - **FOLLOW(A) = $\{a \mid S \Rightarrow^* \ldots Aa\ldots,\ a \in V_T\}$**

    如果A是某个句型的最右符号，那么$属于FOLLOW($A$)

    ↑ **在句型中紧跟在A之后的终结符或$组成的集合**

    $表示输入记号串的结束标志;S是开始符号

LL(1)文法：FIRST(X)

- 计算FIRST(X), $X \in V_T \cup V_N$
  - $X \in V_T$, FIRST(X) = {X}
  - $X \in V_N$ 且 $X \to Y_1 Y_2 \ldots Y_k$

    如果 $a \in$ FIRST(Y_i)且ε 在FIRST(Y_1), …, FIRST(Y_{i-1})中，则将 a加入到FIRST(X)

    如果 ε 在FIRST(Y_1), …, FIRST(Y_k)中，则将ε 加入到FIRST(X)
  - $X \in V_N$ 且 $X \to \varepsilon$，则将 ε 加入到 FIRST(X)

FIRST集合只包含终结符和ε

LL(1)文法：FIRST, FOLLOW

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

LL(1)文法

LL(1)文法的定义

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

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

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

$S \to A\ B$

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

$B \to a\ C$

$C \to \ldots$

LL(1)文法

LL(1)文法的定义

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

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

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

$E \to TE'$

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

$T \to FT'$

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

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

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }

FIRST(E') = {+, ε}

FRIST(T') = {*, ε}

FOLLOW(E) = FOLLOW(E') = {), $}

FOLLOW(T) = FOLLOW(T') = {+,), $}

FOLLOW(F) = {+, *,), $}

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 → *simple*
| ↑ id
| array [*simple*] of *type*

simple → integer
| char
| num dotdot num

递归下降的预测分析器

type → *simple* | ↑ id | array [*simple*] of *type*
simple → 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 == '↑' ) { match('↑'); 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 → *simple* | ↑ id | array [*simple*] of *type*
simple → 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( );
}
```

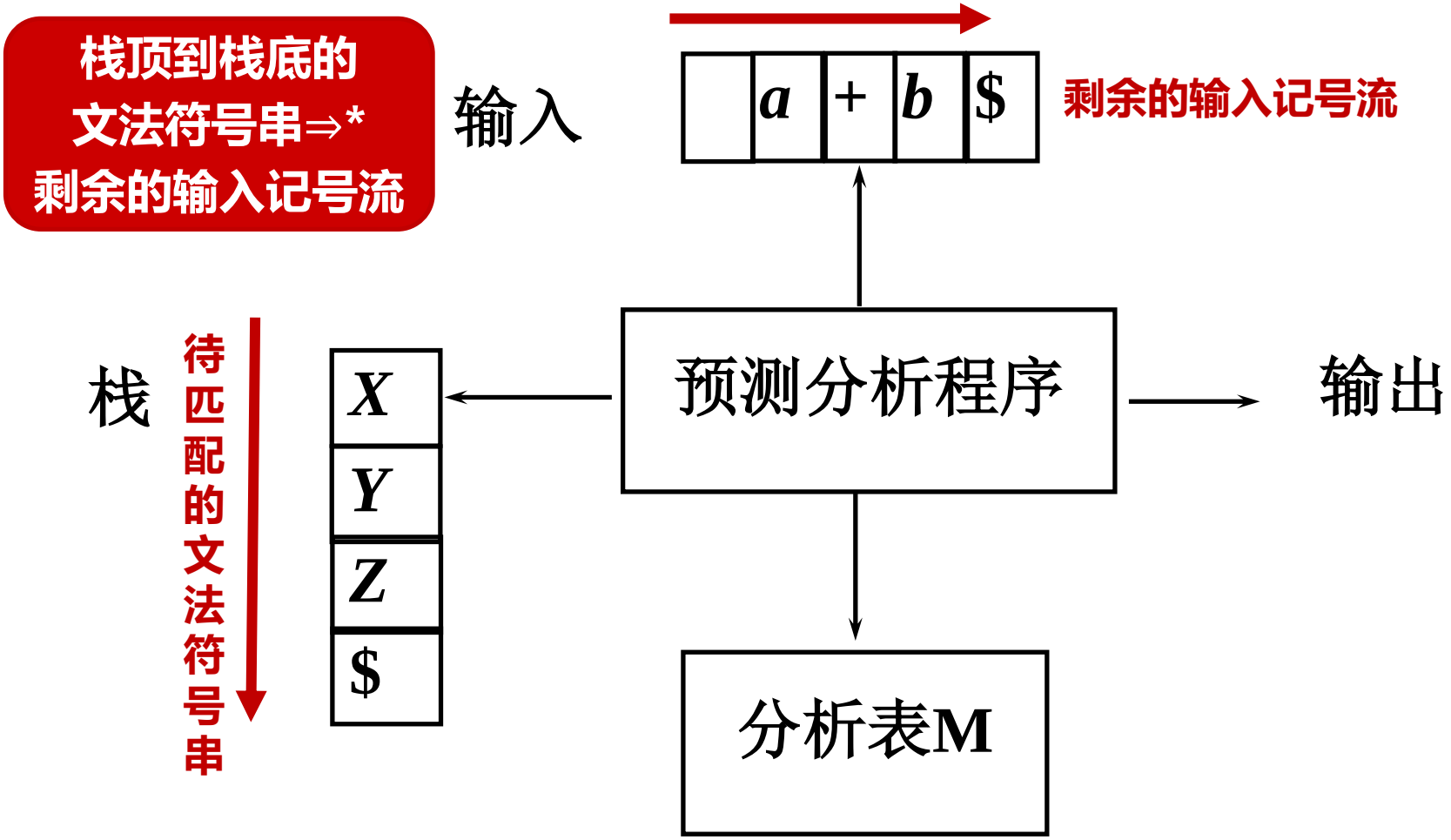
中国科学技术大学
University of Science and Technology of China

3.3 自上而下分析

- 自上而下分析、左递归及消除、提左因子
- LL(1)文法和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

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	$F \rightarrow$ 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$A \to \alpha$表示在面临a时，将栈顶符号A替换为α

- 构造方法

  （1）对文法的每个产生式$A \to \alpha$，执行(2)和(3)

  （2）对FIRST(α)的每个终结符a，把$A \to \alpha$加入$M[A, a]$

  （3）如果ε在FIRST(α)中，对FOLLOW(A)的每个终结符b（包括$），把$A \to \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* → if *expr* then *stmt e_part* | *other*

e_part → else *stmt* | ε　　*expr* → *b*

非终结符	输入符号			
	other	*b*	else	. . .
stmt	*stmt* → other			
e_part			*e_part*→ else *stmt* *e_part* → ε	
expr		*expr* → *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* → ε，这正好满足 else和最近的then配对

LL(1)文法 ⇔ 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* → other			
e_part			*e_part*→ else *stmt* ~~*e_part* → ε~~	
expr		*expr* → *b*		

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

3.3 自上而下分析

- 自上而下分析、左递归及消除、提左因子
- LL(1)文法和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

预测分析的错误恢复

编译器的错误处理

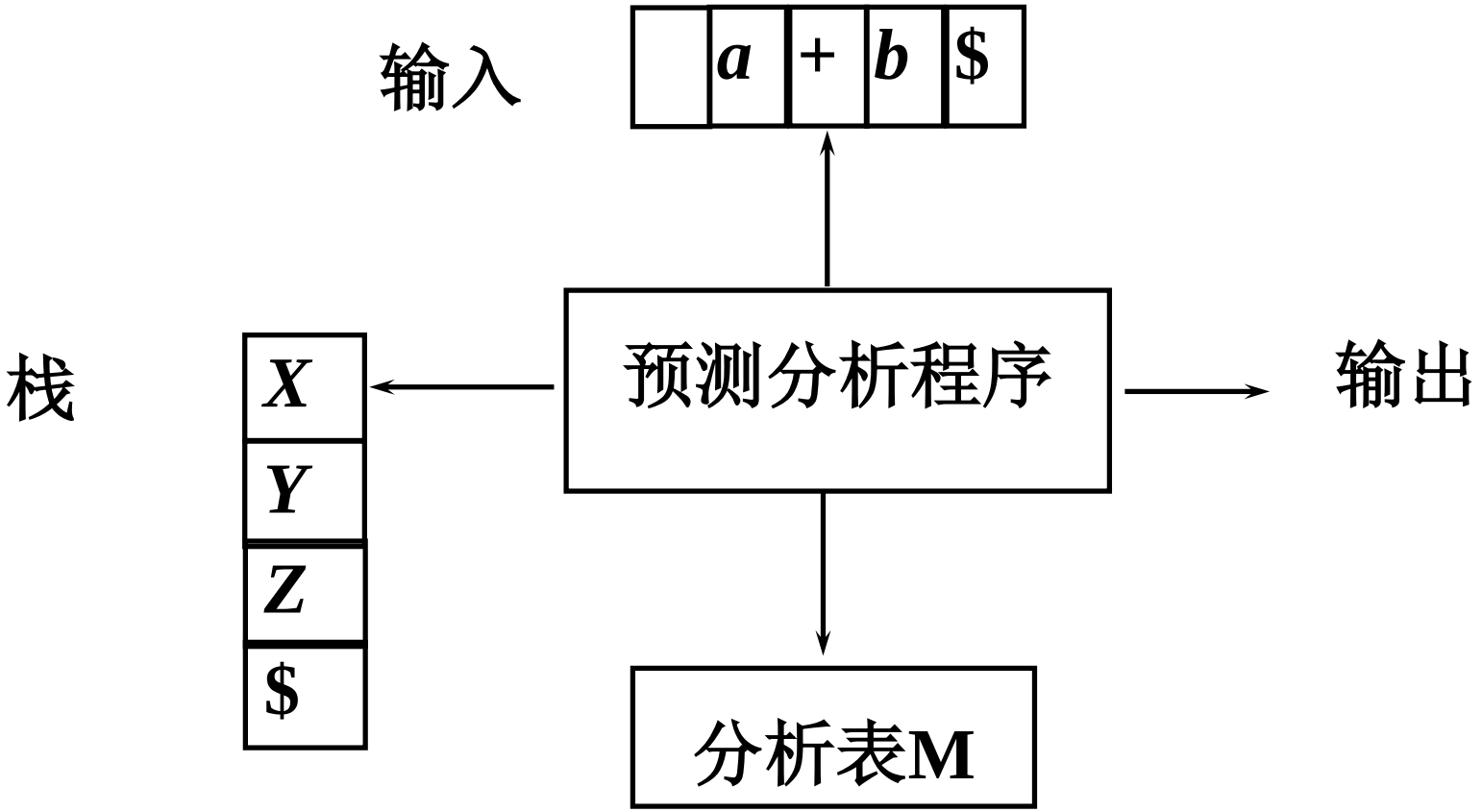
- 词法错误，如标识符、关键字或算符的拼写错
- 语法错误，如算术表达式的括号不配对
- 语义错误，如算符作用于不相容的运算对象
- 逻辑错误，如无穷的递归调用

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

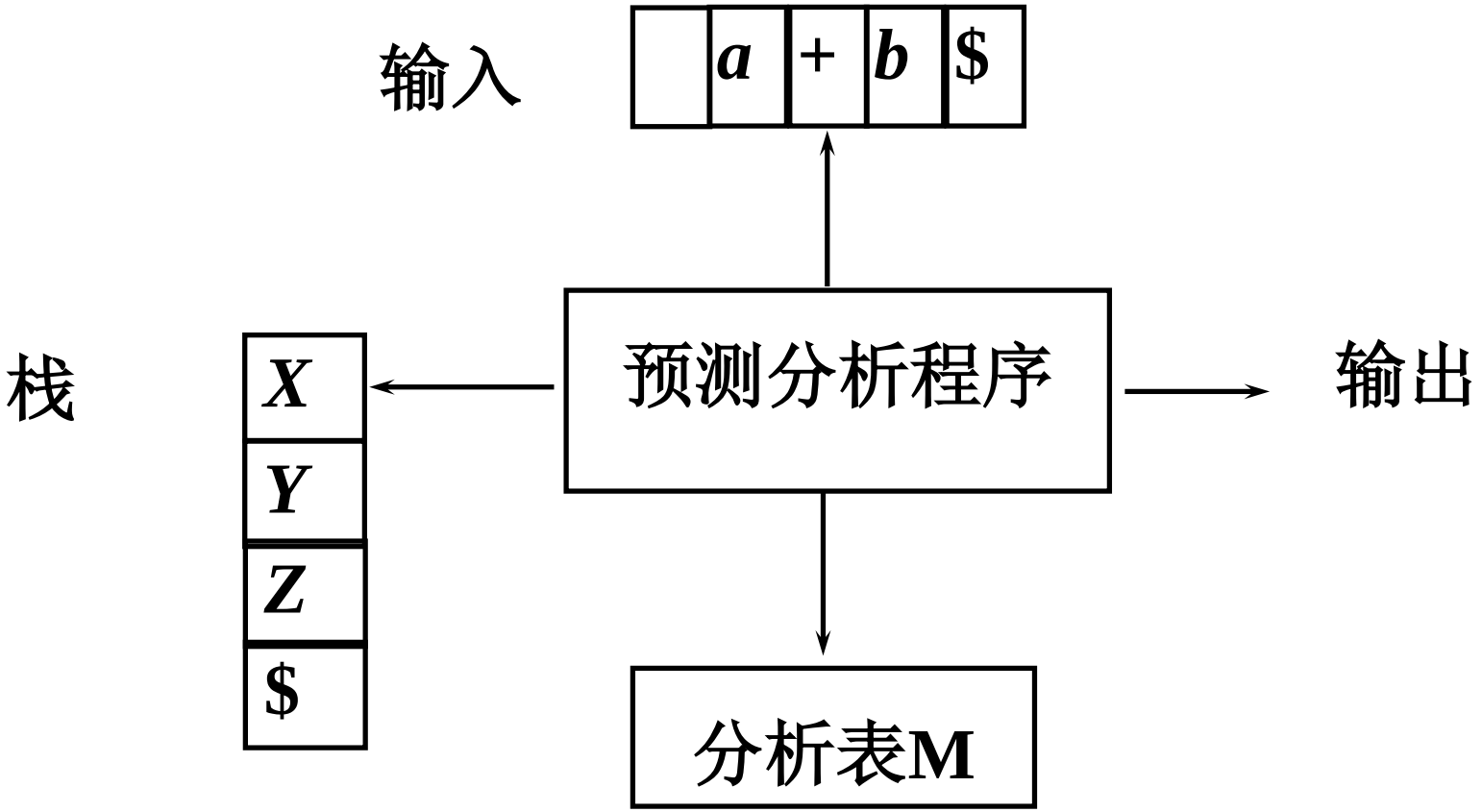
预测分析的错误恢复

- 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

同步记号

if *expr* then *stmt*

出错

（then和分号等记号是*expr*的同步记号）

- 把高层构造的首终结符加到低层构造的同步记号集中

a = *expr*; if ...

出错　　同步记号

(语句的首终结符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
- 把高层构造的首终结符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 = *expr*; , if …

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
  - 把高层构造的首终结符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	+	*	...
E	$E\rightarrow TE'$			
E'		$E'\rightarrow +TE'$		
T	$T\rightarrow FT'$			
T'	出错	$T'\rightarrow \varepsilon$	$T'\rightarrow *FT'$	
...				

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	+	*	...
E	$E\to TE'$			
E'		$E'\to +TE'$		
T	$T\to FT'$			
T'	出错 用$T'\to\varepsilon$	$T'\to\varepsilon$	$T'\to *FT'$	
...				

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
  - 把高层构造的首终结符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

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

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

语法分析的主要方法

- 自上而下（top-down）预测分析
  - 从开始符号出发，为输入串寻找最左推导
    是自上而下形成分析树的过程
  - 即便消除左递归、提取左公因子，仍然存在一些程序语言，它们对应的文法不是LL(1)
- 自下而上（bottom-up）移进-归约
  - 针对输入串，尝试根据产生式归约(reduce，将与产生式右部匹配的串归约为左部符号)，直至归约到开始符号
    是自下而上形成分析树的过程
  - 比top-down方法更一般化

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

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 $S \to aABe$

$A \to Abc \mid b$

$B \to d$

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 $S \to aABe$

$A \to Abc \mid b$

$B \to d$

输入串： $abbcde$

ab（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 $S \to aABe$

$A \to Abc \mid b$

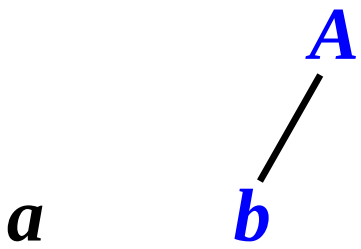
$B \to d$

输入串： $abbcde$

ab

aA（归约）

用产生式 $A \to b$
归约后仍能形成右句型
$aAbcde$ 是右句型



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

右句型：按最右推导推出的句型， $aABe$、$aAde$、$aAbcde$、$abbcde$ 都是右句型

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 $S \to aABe$

$A \to Abc \mid b$

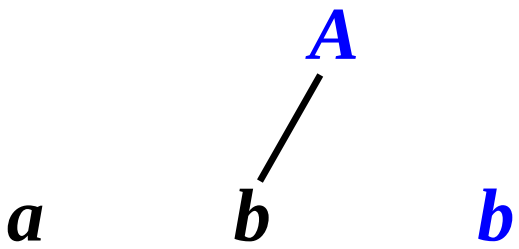
$B \to d$

输入串： $abbcde$

ab

aAb（再读入b）

b可以归约成A吗?



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 $S \to aABe$

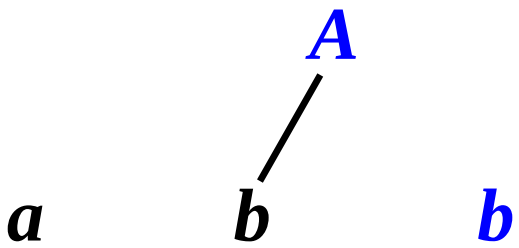
$A \to Abc \mid b$

$B \to d$

输入串： $abbcde$

ab

aAb（再读入b）



b可以归约成A吗?

归约后能否形成右句型?

aAAcde 不是右句型

故不能将b归约成A

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

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 $S \rightarrow aABe$

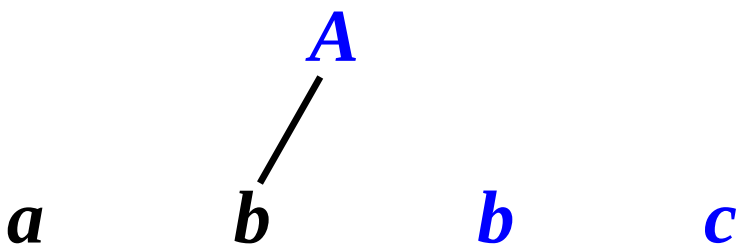
$A \rightarrow Abc \mid b$

$B \rightarrow d$

输入串： $abbcde$

ab

$aAbc$（再读入c）



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　$S \to aABe$

$A \to Abc \mid b$

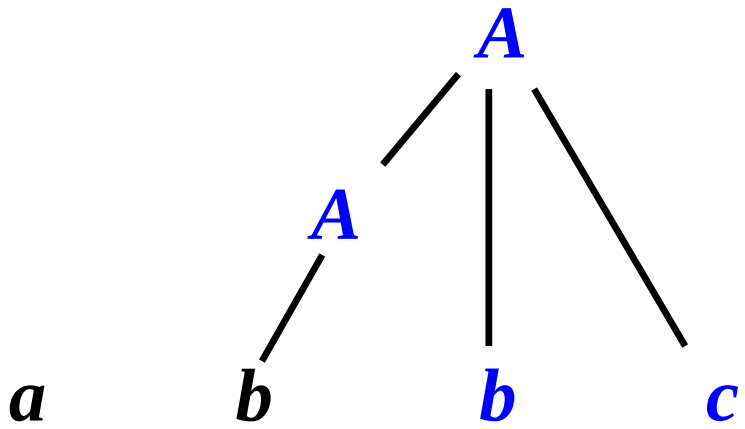
$B \to d$

输入串：$abbcde$

ab

$aAbc$

aA（归约）



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

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　$S \to aABe$

$A \to Abc \mid b$

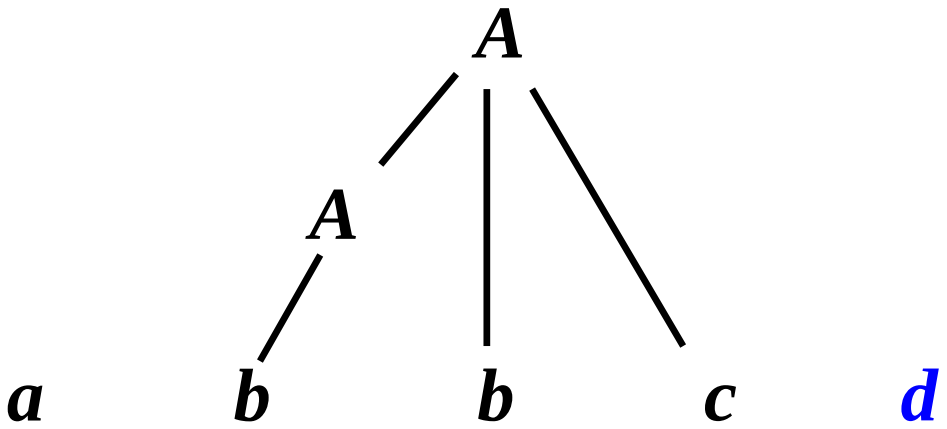
$B \to d$

输入串：$abbcde$

ab

$aAbc$

aAd（再读入d）



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

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 $S \to aABe$

$A \to Abc \mid b$

$B \to d$

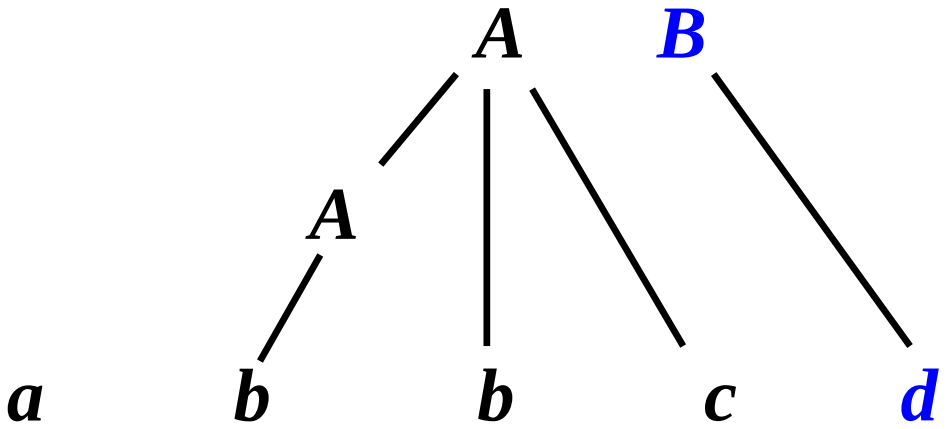
输入串： $abbcde$

ab

$aAbc$

aAd

aAB **(归约)**



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

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 $S \to aABe$

$A \to Abc \mid b$

$B \to d$

输入串： *abbcde*

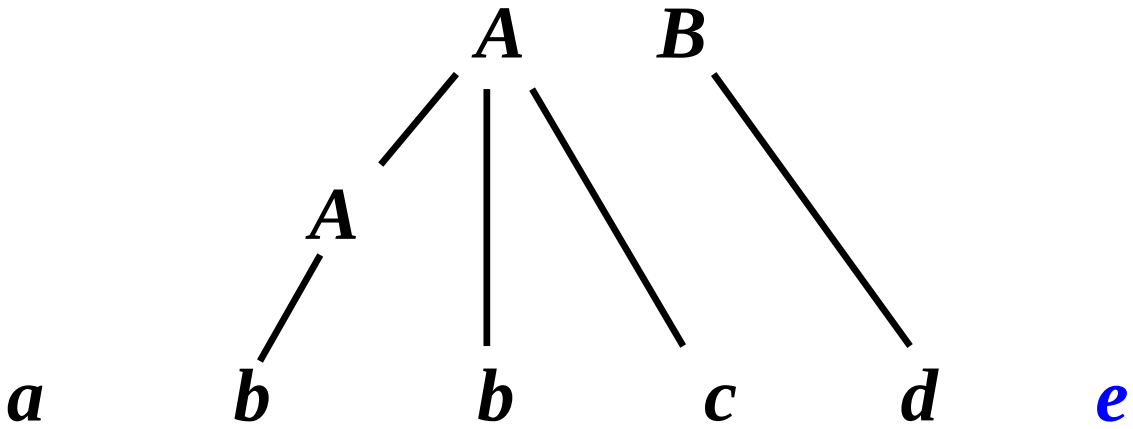
ab

aAbc

aAd

aAB

aABe(再读入*e*)



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

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 $S \to aABe$

$A \to Abc \mid b$

$B \to d$

输入串： *abbcde*

ab

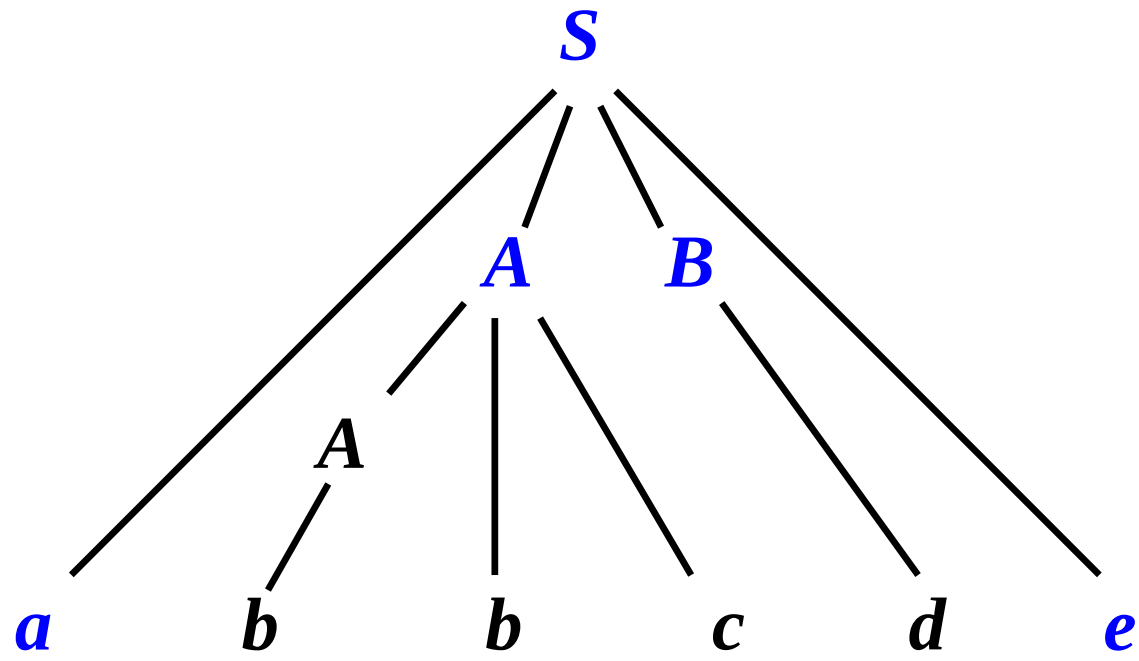
aAbc

aAd

aAB

aABe

S（归约）



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

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

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

句柄(handles)

右句型的句柄（可归约串）

- 是右句型$\alpha\delta\beta$中和某产生式 $B \to \delta$右部匹配的子串δ, 并且
- 把 δ 归约成 B 代表了最右推导的逆过程的一步

  右句型$\alpha\delta\beta$中将子串δ归约成B后得到的串$\alpha B\beta$仍是右句型

  $S \to aABe$

  $A \to Abc \mid b$

  $B \to d$

  * 右句型：最右推导可得的句型

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

- 句柄的右边仅含终结符 （是尚未处理输入串）
- 如果文法二义，那么句柄可能不唯一

例 句柄不唯一

$$E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$$

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

例 句柄不唯一

$$E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$$

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

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

在右句型$E * E + \mathrm{id}_3$中，句柄不唯一：id_3、$E * E$

* 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。

需要引入**栈**保存移进的文法符号

归约时需要从栈的顶部将形成**句柄**的文法符号串弹出，再将归约成的非终结符入栈

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
$$E$*$E$	$+ id_3$$	

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$E \to$ id归约
$$E$	$* id_2 + id_3$$	移进
$$E*$	$id_2 + id_3$$	移进
$$E*id_2$	$+ id_3$$	按$E \to$ id归约
$$E*E$	$+ id_3$$	移进？归约？

$$E \Rightarrow_{rm} E*E \Rightarrow_{rm} E*E+E \Rightarrow_{rm} E*E+id_3$$ 移进

$$E \Rightarrow_{rm} E+E \Rightarrow_{rm} E+id_3 \Rightarrow_{rm} E*E+id_3$$ 归约

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

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

移进–归约分析需要解决的一些问题

- 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 … $

一般地，优先移进
也满足else的
就近匹配原则

移进-归约分析的冲突

- **归约–归约冲突(reduce/reduce conflict)**

  stmt → id (*parameter_list*) | *expr* = *expr*　　id(…)是函数调用

  parameter_list → *parameter_list*, *parameter* | *parameter*

  parameter → id

  expr → id (*expr_list*) | id　　id(…)也表示数组元素的引用

  expr_list → *expr_list*, *expr* | *expr*

由*A*(*I*, *J*)开始的语句

归约成*expr*还是*parameter*？

栈	输入
… id (id	, id)…

方法1
一般用在前的
产生式进行归约

移进-归约分析的冲突

- **归约–归约冲突(reduce/reduce conflict)**

  stmt → id (*parameter_list*) | *expr* = *expr*　　id(…)是函数调用

  parameter_list → *parameter_list*, *parameter* | *parameter*

  parameter → id

  expr → id (*expr_list*) | id　　id(…)也表示数组元素的引用

  expr_list → *expr_list*, *expr* | *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

方法2
改写文法，区分
id是否是procid

3.5 LR分析器

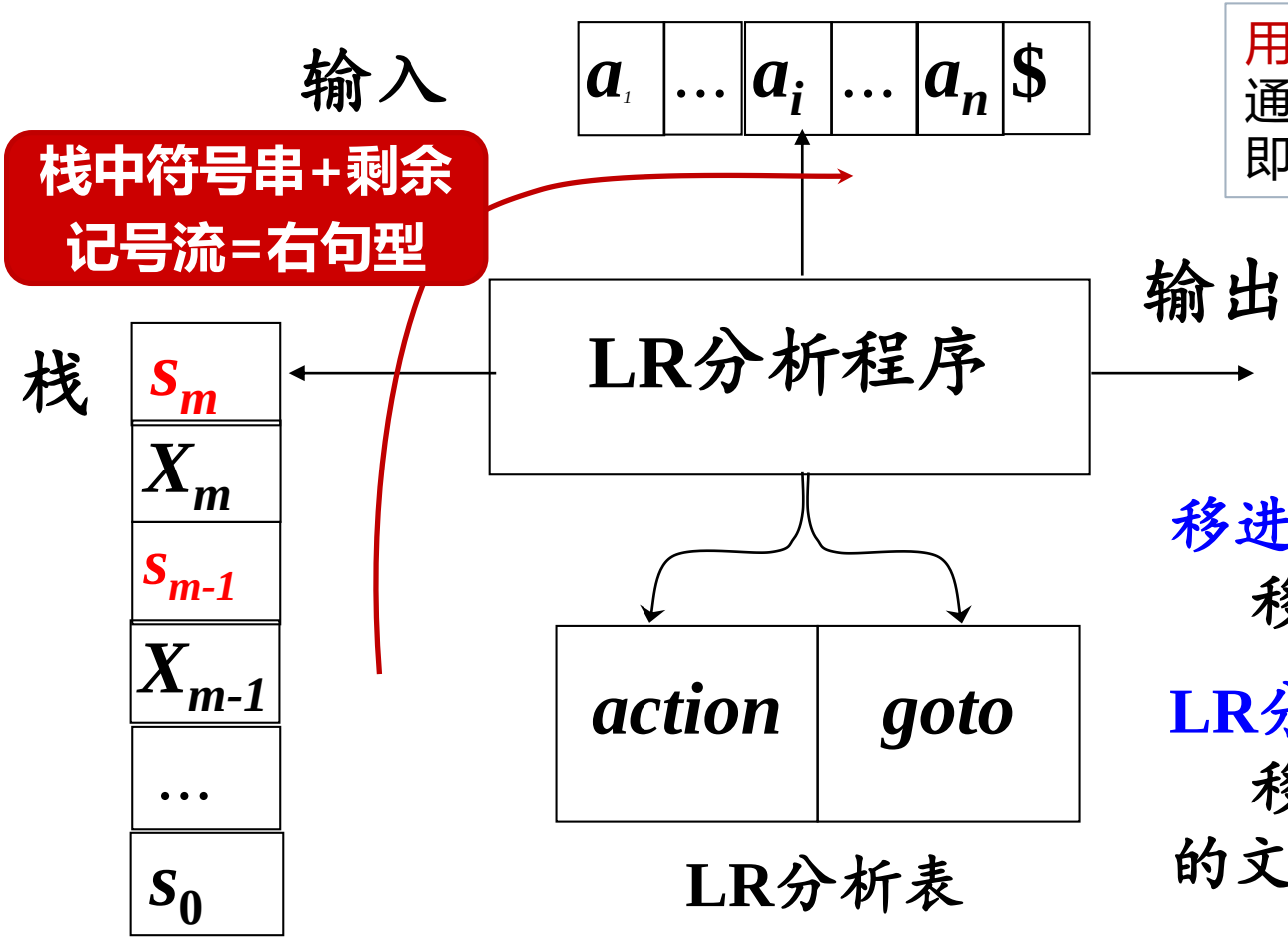
(L-scanning from left to right; R- rightmost derivation in reverse)

- 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

- 如何快速识别栈的顶部是否形成句柄? ➔引入状态

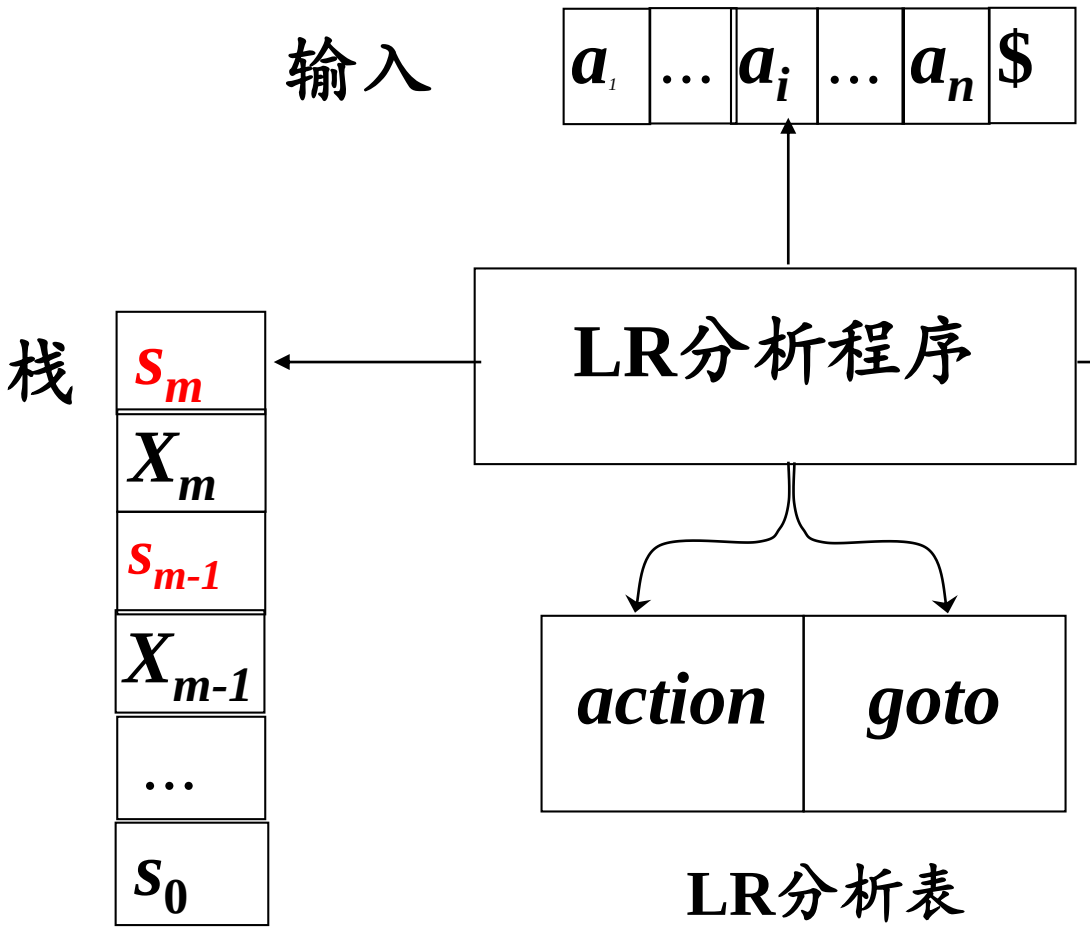


用状态刻画栈中的内容
通过查看栈顶状态，
即可判断是否形成句柄

移进-归约分析器
移进符号

LR分析器
移进状态和其相关的文法符号(s_0除外)

LR分析算法: 分析器的模型



s_j: 是栈中该状态到栈底信息的浓缩

action[s_m, a_i]:

移进|归约|接受|出错

goto[s_{m-r}, A]=s_j: 移进A和s_j (归约后使用)

LR分析算法：举例

例 P69

$E \to E + T \mid E \to T$

$T \to T * F \mid T \to E$

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

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

LR分析表

状态	动作						转移		
	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			

LR分析算法：举例

栈	输　入	动　作
0	id * id + id $	

状态	动作						转移		
	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			

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 $	

状态	动作						转移		
	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			

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 $	

1. 查action[5, *]= r6 归约
2. 按r6执行归约($F \rightarrow a$)：
   - 从栈中弹出| a |个状态-符号对
   - 查goto[0, F] =>3
   - 将(F, 3)入栈

状态	动作						转移		
	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			

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 $	

状态	动作						转移		
	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			

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 $	

状态	动作						转移		
	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			

LR分析算法：举例

状态	动作						转移		
	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			

栈	输　入	
0	id * id + id	
0 id 5	* id + id	
0 *F* 3	* id + id	
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

状态	动作						转移		
	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			

栈	输　入	
0	id * id + id	
0 id 5	* id + id	
0 *F* 3	* id + id	
0 *T* 2	* id + id $	移进
0 *T* 2 * 7	id + id $	移进
0 *T* 2 * 7 id 5	+ id $	按*F* → id归约
0 *T* 2 * 7 *F* 10	+ id $	

LR分析算法：举例

状态	动作						转移		
	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			

栈	输　入	
0	id * id + id	
0 id 5	* id + id	
0 *F* 3	* id + id	
0 *T* 2	* id + id $	移进
0 *T* 2 * 7	id + id $	移进
0 *T* 2 * 7 id 5	+ id $	按*F* → id归约
0 *T* 2 * 7 *F* 10	+ id $	按*T* → *T***F*归约
. . .	. . .	. . .

LR分析算法：举例

状态	动作						转移		
	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			

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

归约为 **开始符号**

完成合法输入串的分析

LR分析: 基本概念

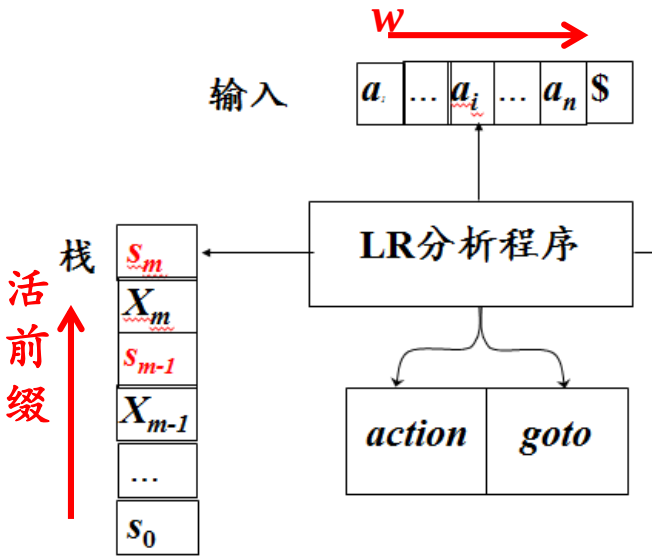
活前缀（viable prefix）

- 右句型的前缀$\gamma\beta$，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ε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	+	*	(	)	$	*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

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		

$$S \Rightarrow \ldots \Rightarrow \gamma A\, b\, w \Rightarrow \gamma l\, \beta\, b\, w$$

$A \rightarrow l\beta$

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		

$S \Rightarrow \ldots \Rightarrow \gamma A\, b\, w \Rightarrow \gamma\, l\, \beta\, b\, w$

$A \rightarrow l\beta$

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整个终结符串后，才确定用哪个产生式归约	看见产生式右部推出的第一个终结符后，便要确定用哪个产生式推导

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

$A \rightarrow l\beta$

LL(1)决定用该产生式的位置

LR(1)决定用该产生式的位置

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

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分析表的构造

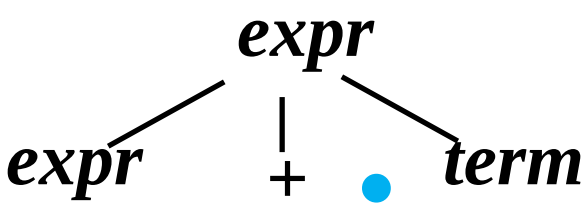
- LR(0)项目与LR(1)项目

- SLR：简单的LR
  - 构造LR(0) 项目集规范族➔形成DFA状态➔SLR分析表
- LR：规范的LR
  - 构造LR(1) 项目集规范族➔形成DFA状态➔LR分析表
- LALR：向前看的LR
  - 构造LR(1) 项目集规范族➔形成DFA状态➔合并同心项目集➔LR分析表

LR分析表的构造方法

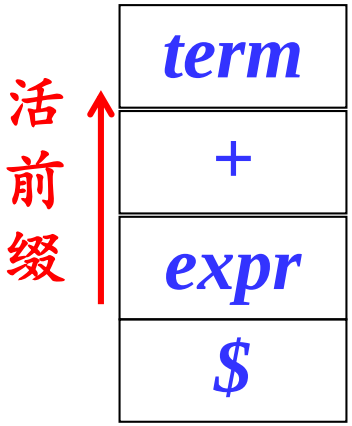
- **LR(0) 项目**
  - 在右部的某个地方加点的产生式
  - 加点的目的是用来表示分析过程所处的位置

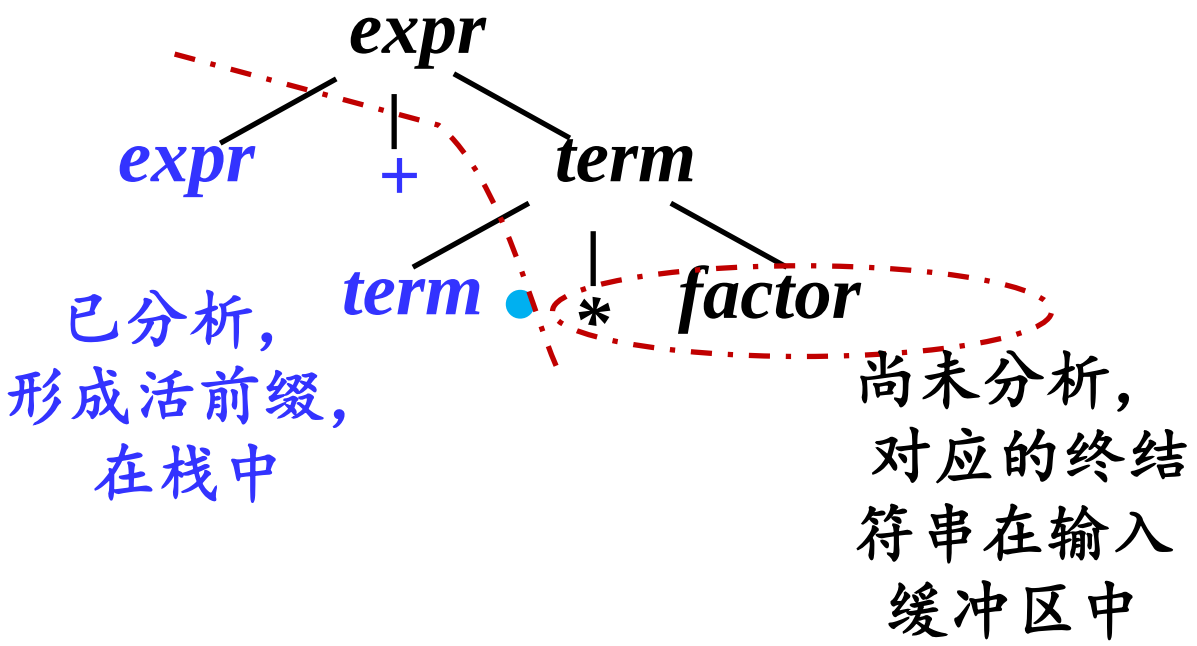
$expr \rightarrow expr + \cdot term$



LR分析表的构造方法

- 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所处的位置





LR分析表的构造方法

- **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所处的位置

例 $A \rightarrow XYZ$ 对应有四个项目

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

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

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

$A \rightarrow \cdot$

LR分析表的构造方法

- LR(0) 项目 $[A\to\alpha\cdot\beta]$， α和β为文法符号串(可能为ε)
- 状态：LR(0) 项目集规范族(canonical LR(0) collection)

1. 拓广文法 (augmented grammar)

新增产生式和
新的开始符号

$E'\to E$ 旨在指示分析器何时开始分析、何时完成分析

$E\to E+T\mid T$

$T\to T*F\mid F$

$F\to(E)\mid \mathrm{id}$

$[E'\to\cdot E]$ $[E'\to E\cdot]$

LR分析表的构造方法

- LR(0) 项目 $[A\rightarrow\alpha\ \cdot\beta]$， α和β为文法符号串(可能为ε)
- 状态：LR(0) 项目集规范族(canonical LR(0) collection)

初始状态

I_0:

$E'\rightarrow\ \cdot E$

LR分析表的构造方法

- LR(0) 项目 $[A\to\alpha\cdot\beta]$， α和β为文法符号串(可能为ε)
- 状态：LR(0) 项目集规范族(canonical LR(0) collection)

初始状态

I_0:

$E'\to\cdot E$

$E\to\cdot E+T$

$E\to\cdot T$

求项目集的闭包closure(I) P75

求LR(0)项目集的闭包closure(I) P75

$[A\to\alpha\cdot B\beta]\in I$

$\forall B\to\gamma : [B\to\cdot\gamma]\in I$

LR分析表的构造方法

- LR(0) 项目 $[A\to\alpha\cdot\beta]$， α和β为文法符号串(可能为ε)
- 状态：LR(0) 项目集规范族(canonical LR(0) collection)

初始状态

I_0:

$E'\to\cdot E$

$E\to\cdot E+T$

$E\to\cdot T$

$T\to\cdot T*F$

$T\to\cdot F$

求项目集的闭包closure(*I*) P75

求LR(0)项目集的闭包closure(*I*) P75

$[A\to\alpha\cdot B\beta]\in I$

$\forall B\to\gamma$: $[B\to\cdot\gamma]\in I$

LR分析表的构造方法

- LR(0) 项目 $[A\to\alpha\cdot\beta]$，α和β为文法符号串(可能为ε)
- 状态：LR(0) 项目集规范族(canonical LR(0) collection)

初始状态

I_0:

$E'\to\cdot E$

$E\to\cdot E+T$

$E\to\cdot T$

$T\to\cdot T*F$

$T\to\cdot F$

$F\to\cdot(E)$

$F\to\cdot id$

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

LR分析表的构造方法

- SLR（SimpleLR）
  - LR(0) 项目 $[A\rightarrow\alpha\cdot\beta]$， α和β为文法符号串(可能为ε)
  - 每个DFA状态：LR(0) 项目集规范族
- 规范的LR分析
  - LR(1) 项目：带搜索符(lookahead) $[A\rightarrow\alpha\cdot\beta, a]$

  表示A之后紧跟a. 如果存在$S \Rightarrow^*_{rm} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$，则
  - a是w的第一个符号，或者 w 是ε 且a是$
  - 问题：LR(1)项目数量庞大 => LR(1)分析的状态数偏多
- LALR分析（LookAhead LR）

  和SLR同样多的状态，通过合并规范LR(1)项目集来得到

LR分析表的构造

1. 拓广文法

$$S' \to S$$
$$S \to BB$$
$$B \to bB \mid a$$

2. 构造LR(0) 项目集规范族或LR(1)项目集规范族

=>构造识别活前缀的DFA

活前缀：某个右句型的一个前缀，该前缀不超过该右句型的最右句柄的右端

右句型：通过最右推导得到的句型

3. 从上述DFA构造LR分析表

注： LR(0)项目集规范族 => SLR分析表
LR(1)项目集规范族 => 规范的LR分析表

构造识别活前缀的DFA

- LR(0)项目集规范族

I_0:

$S' \to \cdot S$

$S \to \cdot BB$

$B \to \cdot bB$

$B \to \cdot a$

- LR(1)项目集规范族

I_0:

$S' \to \cdot S$, $\$$

$S \to \cdot BB$, $\$$

$B \to \cdot bB$, a/b

$B \to \cdot a$, a/b

首终结符

FIRST(B) ={a, b}

求LR(0)项目集的闭包closure(I)

P75

$[A \to \alpha \cdot B\beta] \in I$

$\forall\ B \to \gamma : [B \to \cdot\gamma\ ] \in I$

求LR(1)项目集的闭包closure(I)

P82

$[A \to \alpha \cdot B\beta, a] \in I$

$\forall\ B \to \gamma : [B \to \cdot\gamma, b] \in I,$

$b \in$ FIRST(βa)

构造识别活前缀的DFA (以LR(1)项目集为例)

$S' \to S$
$S \to BB$
$B \to bB \mid a$

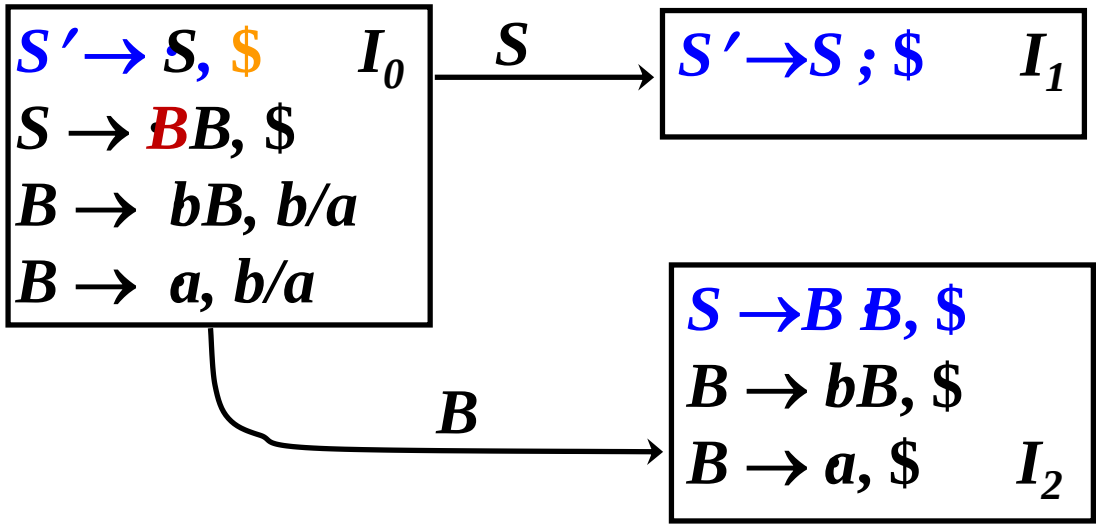
$S' \to S,\ \$ \quad I_0$
$S \to BB,\ \$$
$B \to bB,\ b/a$
$B \to a,\ b/a$

S

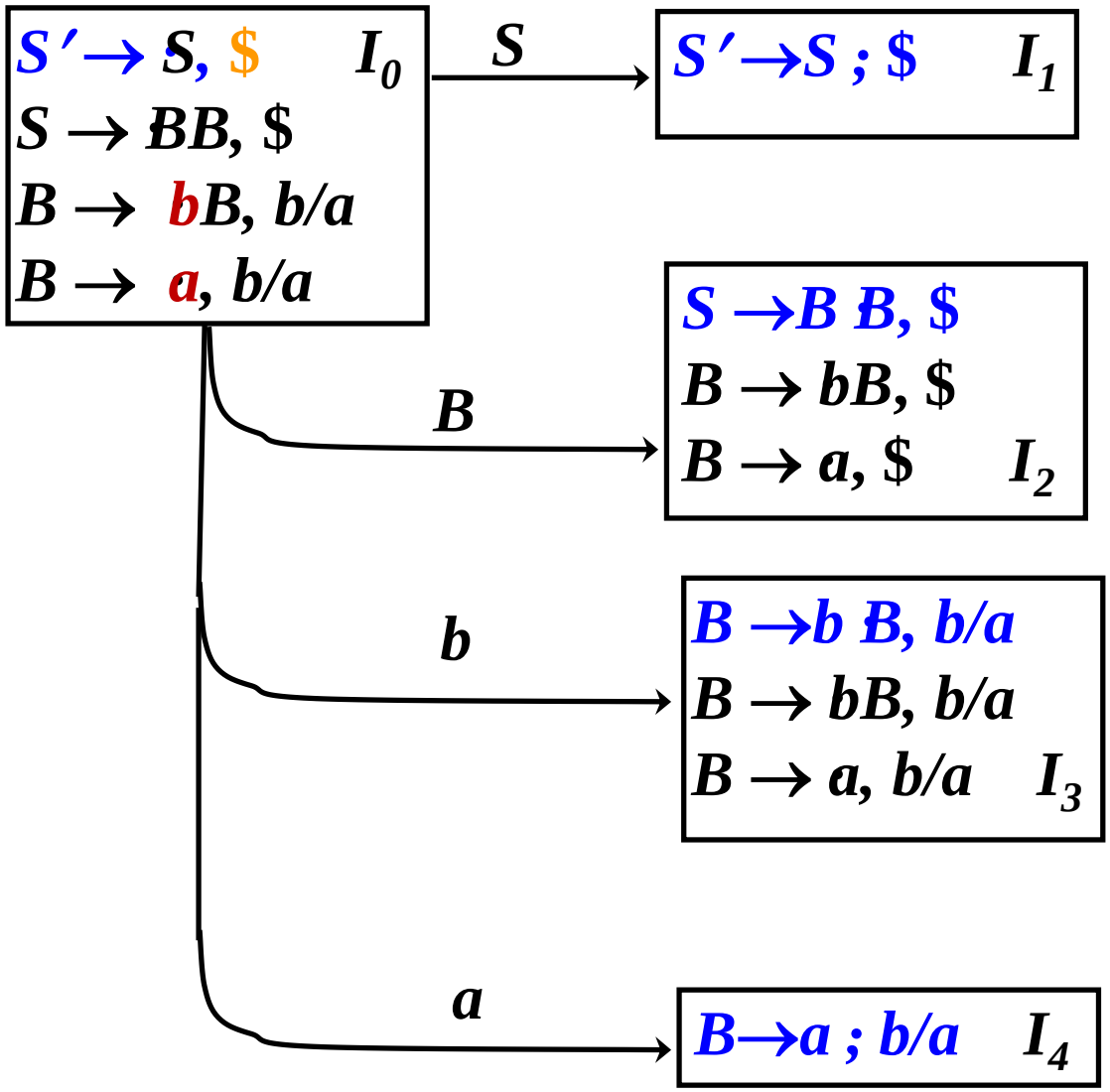
$S' \to S\ ;\ \$ \quad I_1$

$I_1 := goto\ (\ I_0,\ S\ )$

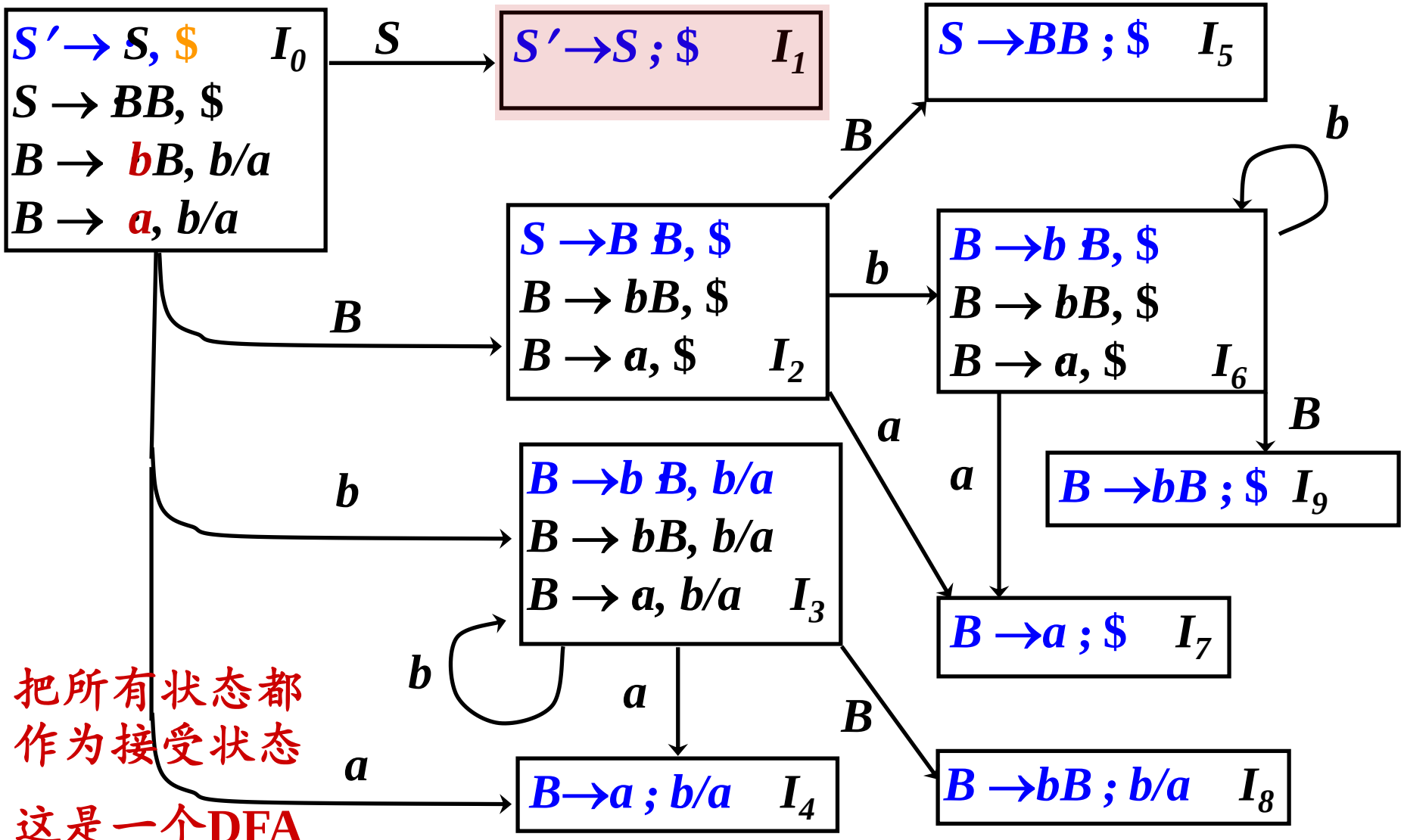
构造识别活前缀的DFA (以LR(1)项目集为例)



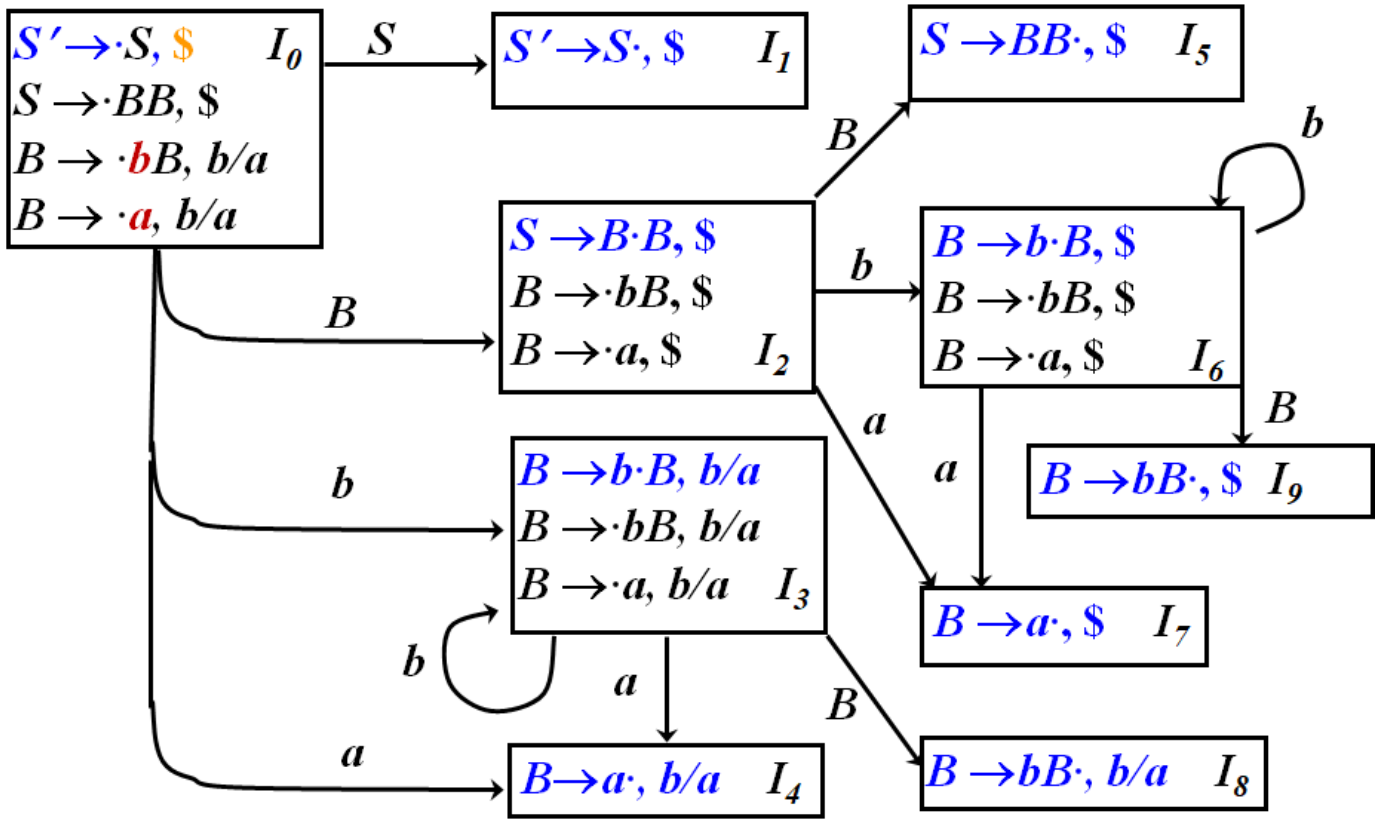
构造识别活前缀的DFA (以LR(1)项目集为例)



构造识别活前缀的DFA (以LR(1)项目集为例)



构造识别活前缀的DFA (以LR(1)项目集为例)



$S' \Rightarrow S$
$\Rightarrow BB$
$\Rightarrow BbB$
$\Rightarrow BbbB$

把所有状态都
作为接受状态

这是一个DFA

*bB*是最右句柄
BbbB 的所有前缀(活前缀)都可接受

构造识别活前缀的NFA

以LR(0)项目集为例说明

I_0:

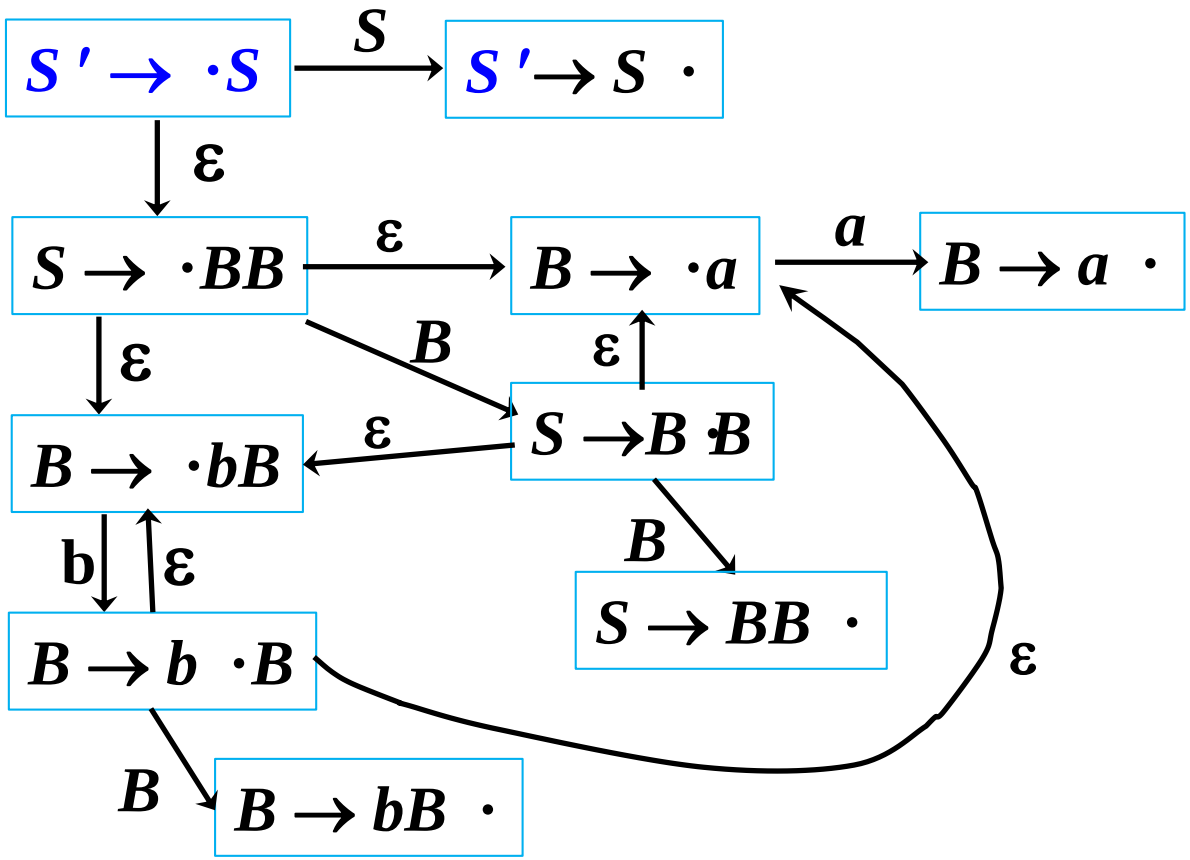
$S' \to \cdot S$

$S \to \cdot BB$

$B \to \cdot bB$

$B \to \cdot a$

每个项目一个状态

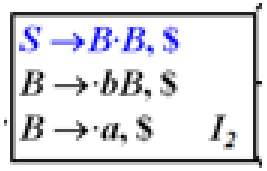


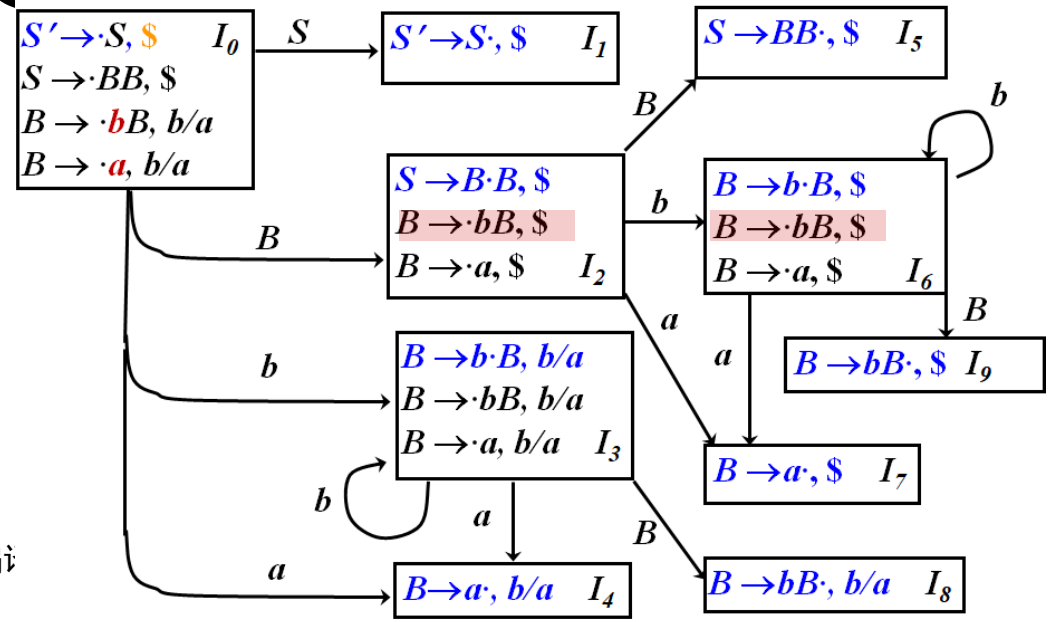
有效项目

- LR(0)项目[$A\to\alpha\cdot\beta$]对活前缀 $\gamma=\delta\alpha$有效：
  如果存在着推导$S'\Rightarrow^*_{rm}\delta Aw\Rightarrow_{rm}\delta\alpha\beta w$
- LR(1)项目[$A\to\alpha\cdot\beta, a$]对活前缀 $\gamma=\delta\alpha$ 有效：
  如果存在着推导 $S'\Rightarrow^*_{rm}\delta Aw\Rightarrow_{rm}\delta\alpha\beta w$，其中：
  - a是w的第一个符号，或者 w 是ε 且a是$

项目与活前缀之间的关系

- [$B\to\cdot bB$,$]对活前缀B、Bb、Bbb都有效
- 活前缀B有多个有效项目

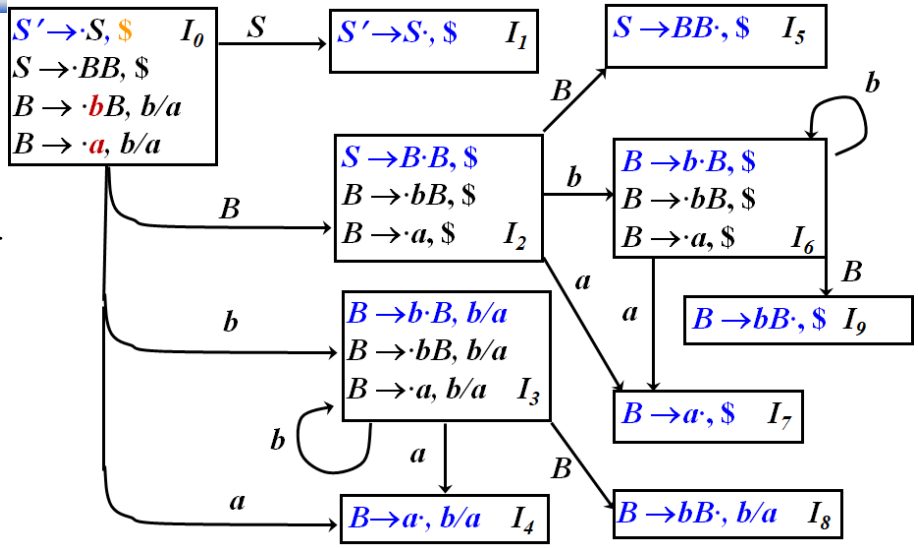




有效项目

项目与活前缀之间的关系

- 活前缀是DFA中从初始状态到项目所在状态路径上的文法符号串联形成的串



- 从项目$A \rightarrow \alpha \cdot \beta$对活前缀$\delta\alpha$有效这个事实可以知道
  - ✓ 如果$\beta \neq \varepsilon$，应该移进
  - ✓ 如果$\beta = \varepsilon$， 应该用产生式$A \rightarrow \alpha$归约

- 一个活前缀γ的有效项目集是从这个DFA的初态出发，沿着标记为γ的路径到达的那个项目集（状态）

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$[A\to\alpha\ a\beta]$在 I_i中，并且 $goto(I_i, a) = I_j$，那么置$action[i, a]$为sj
  - 归约：如果$[A\to\alpha\ \cdot]$在I_i中，那么对$FOLLOW(A)$中所有的a, 置$action[i, a]$为rj，j是产生式 $A\to\alpha$的编号
  - 接受：如果$[S'\to S\ \cdot]$在I_i中，那么置$action[\ i, \$\ ]$为$acc$

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$[A\to\alpha\ a\beta]$在 I_i中，并且 $goto(I_i, a) = I_j$，那么置$action[i, a]$为sj
  - 归约：如果$[A\to\alpha\ \cdot]$在I_i中，那么对$FOLLOW(A)$中所有的a, 置$action[i, a]$为rj，j是产生式 $A\to\alpha$的编号
  - 接受：如果$[S'\to S\ \cdot]$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$[S'\to \cdot S]$的项目集对应的状态

构造规范的LR分析表

- 构造LR分析表，状态 i 的*action*函数按如下确定

① 如果$[A \to \alpha \cdot a\beta, b]$在$I_i$中，且$goto(I_i, a) = I_j$，那么置$action[i, a]$为$sj$

② 如果$[A \to \alpha \cdot, a]$在I_i中，且$A \neq S'$，那么置$action[i, a]$为rj

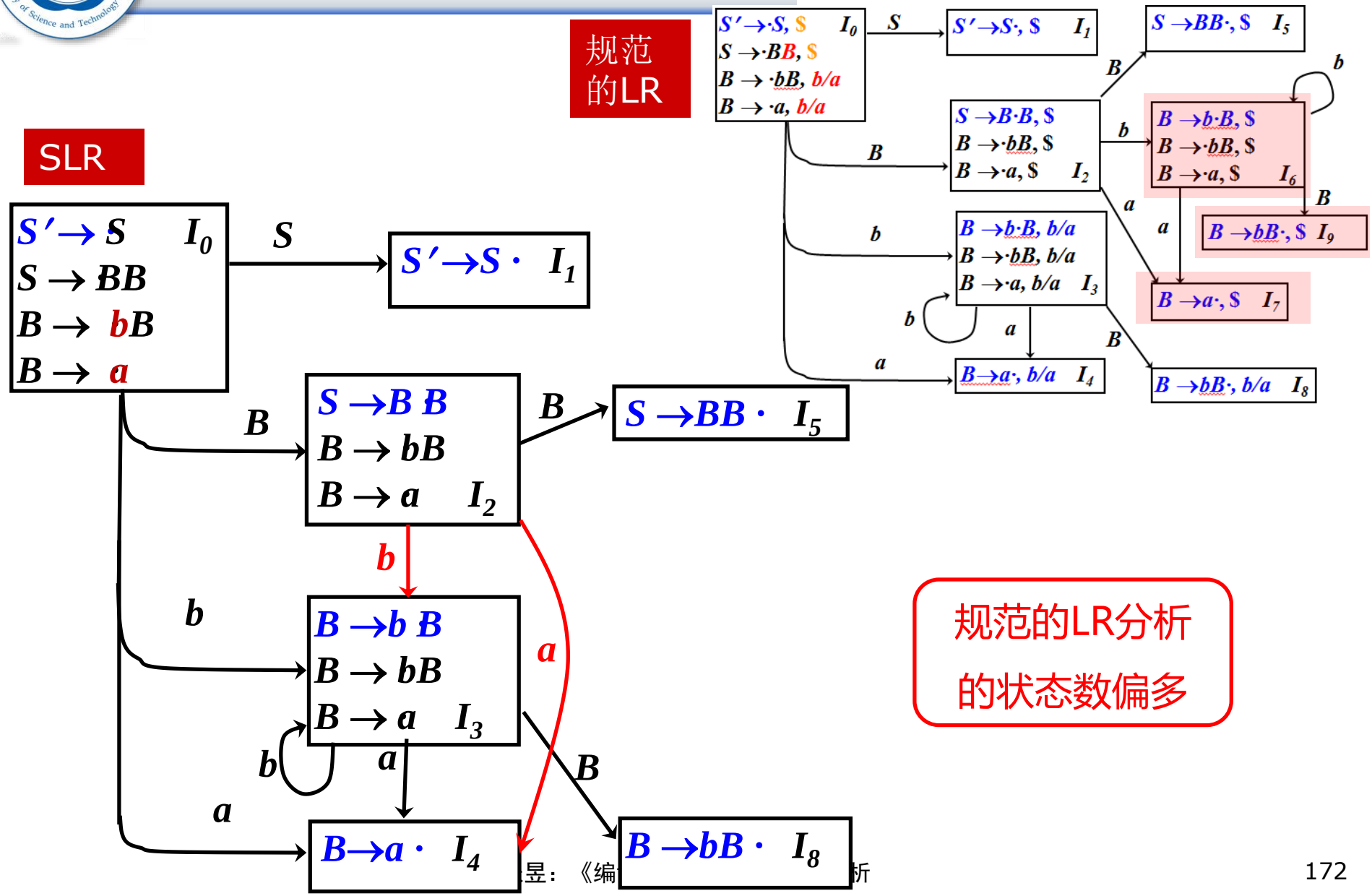
③ 如果$[S' \to S \cdot, \$]$在$I_i$中，那么置$action[i, \$] = acc$

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$goto(I_i, A) = I_j$，那么$goto[i, A] = j$
- 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$[S' \to \cdot S, \$]$的项目集对应的状态

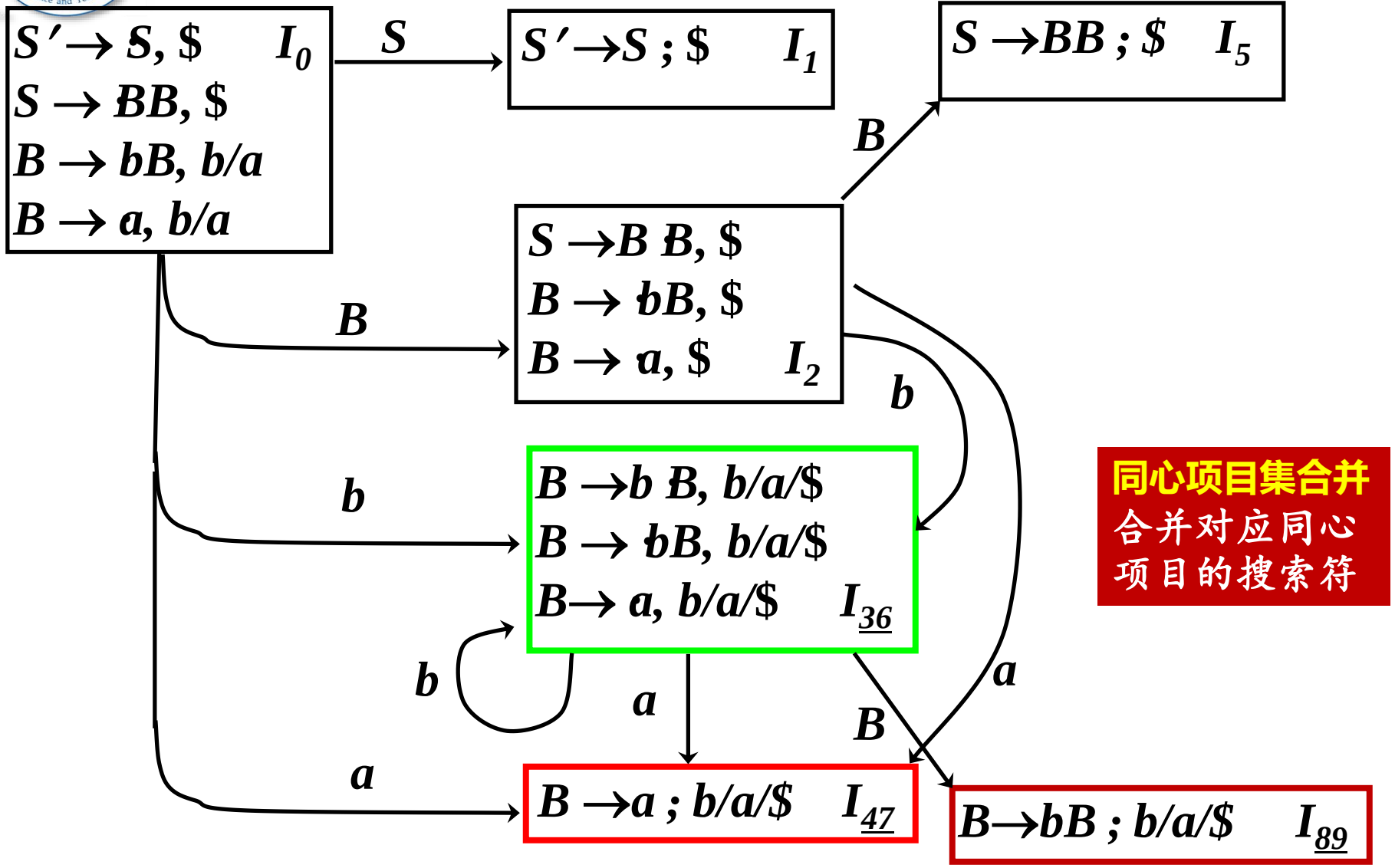
SLR是根据Follow(*A*)来确定归约动作
这里是根据搜索符（上下文信息）来确定

规范的LR vs. SLR 分析

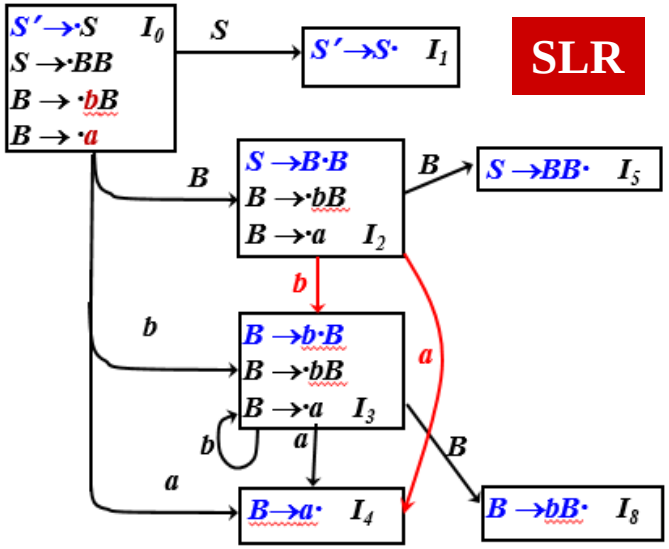


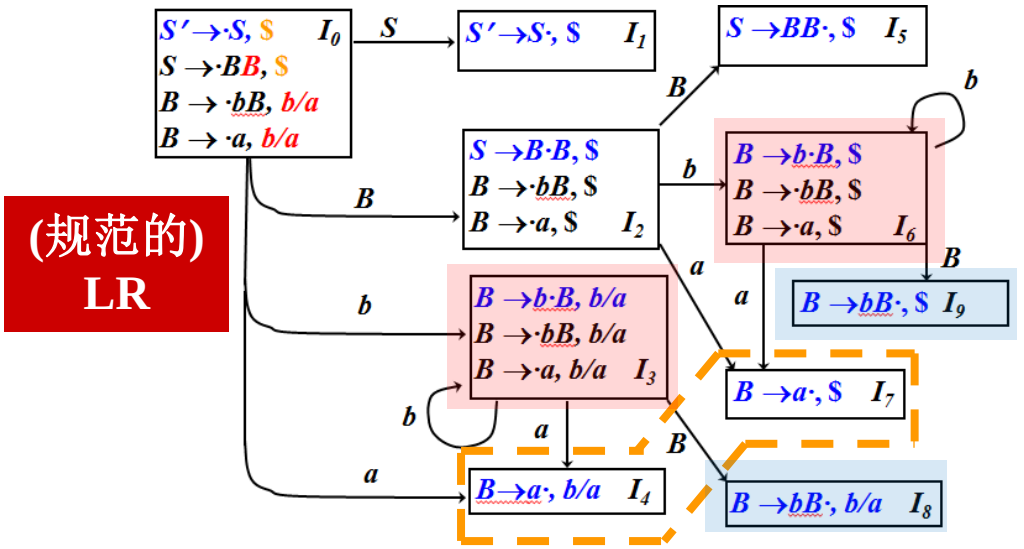
规范的LR分析的状态数偏多

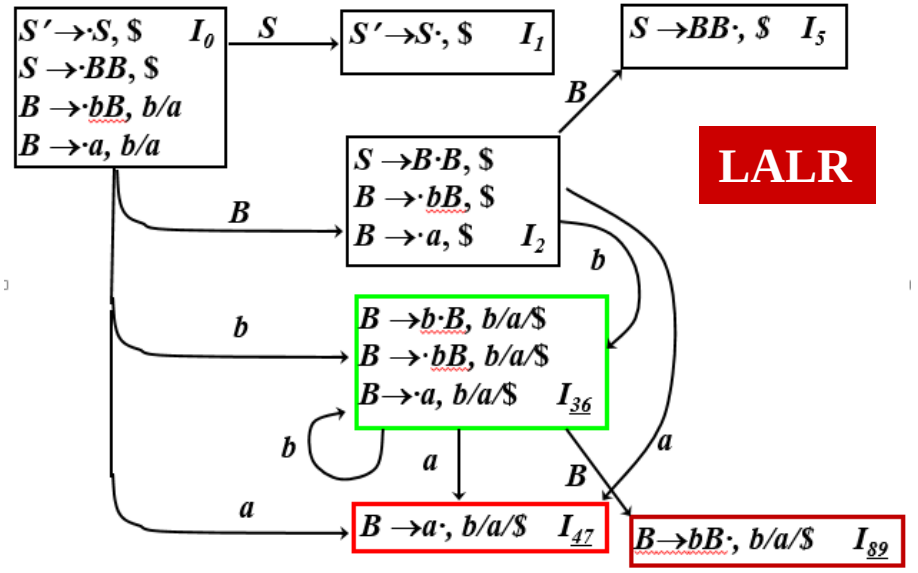
LALR分析


S′→S, $ I0
S→BB, $
B→bB, b/a
B→a, b/a
S
S′→S ; $ I1
S→BB ; $ I5
B
S→B B, $
B→bB, $
B→a, $ I2
B
b
B→b B, b/a/$
B→bB, b/a/$
B→a, b/a/$ I36
b
b
a
B
a
a
B→a ; b/a/$ I47
B→bB ; b/a/$ I89
同心项目集合并
合并对应同心项目的搜索符

SLR vs. LR vs. LALR分析







- 同心集的合并不会引起新的移进–归约冲突

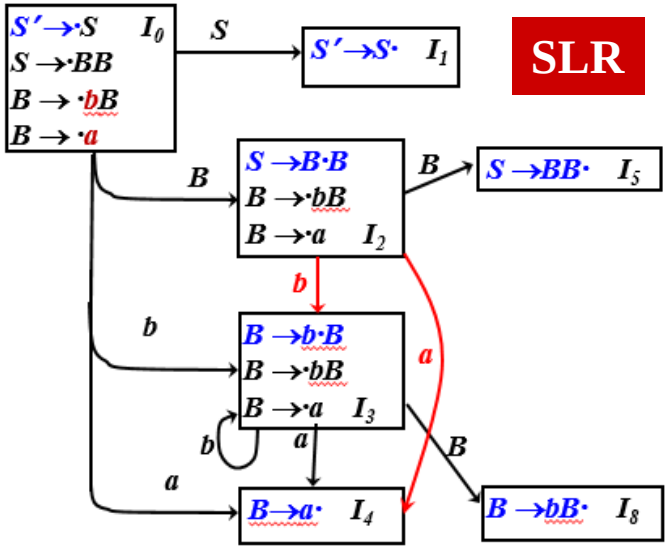
项目集1　　项目集2

$[A\to\alpha\ ;a]$　　$[B\to\beta a\gamma, b]$

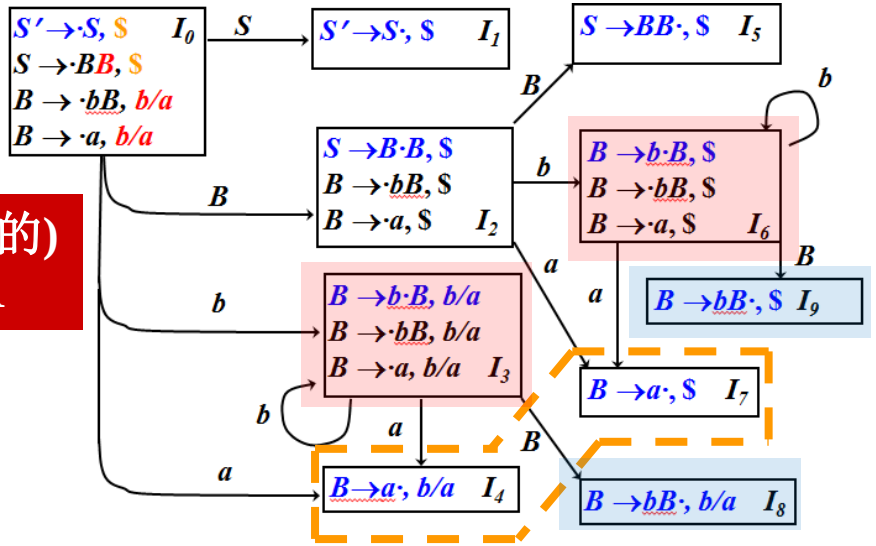
…　　…

若合并后有冲突

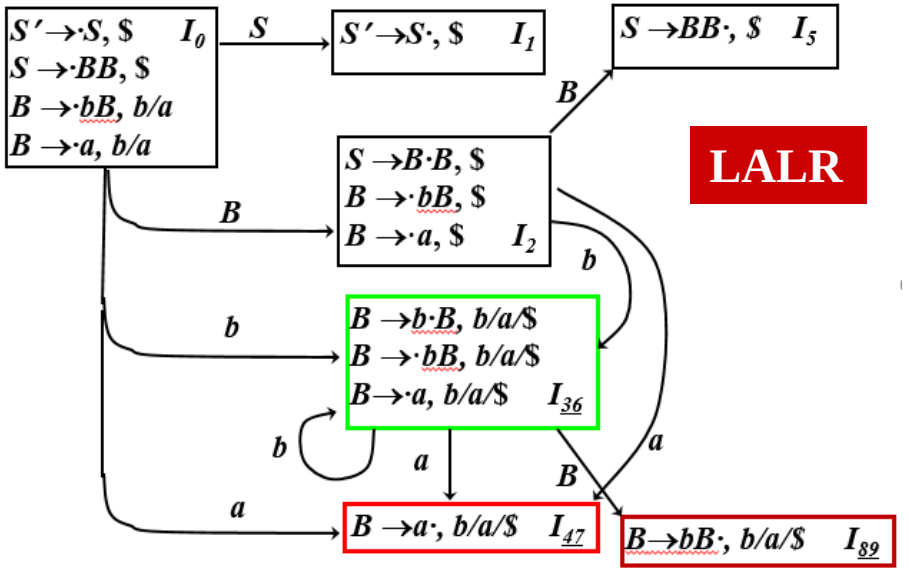
SLR vs. LR vs. LALR分析



SLR



(规范的) LR



LALR

- 同心集的合并不会引起新的移进–归约冲突

项目集1　　项目集2

$[A\to\alpha\ ;\ a]$　　$[B\to\beta a\gamma,\ b]$

$[B\to\beta a\gamma,\ c]$　　$[A\to\alpha\ ;\ d]$

…　　…

则合并前就有冲突

LALR vs. LR 分析

- 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

$S' \to S$
$S \to aAd \mid bBd \mid$
$\quad aBe \mid bAe$
$A \to c$
$B \to c$

对ac有效的项目集

$$\boxed{\begin{array}{l} A \to c \cdot, d \\ B \to c \cdot, e \end{array}}$$

对bc有效的项目集

$$\boxed{\begin{array}{l} A \to c \cdot, e \\ B \to c \cdot, d \end{array}}$$

合并同心集之后

$$\boxed{\begin{array}{l} A \to c \cdot ; d/e \\ B \to c \cdot ; d/e \end{array}}$$

该文法是LR(1)的，
但不是LALR(1)的

SLR(1)文法的描述能力有限

$S \to V = E$

$S \to E$

$V \to * E$

$V \to \text{id}$

$E \to V$

I_0:

$S' \to \cdot S$

$S \to \cdot V = E$

$S \to \cdot E$

$V \to \cdot * E$

$V \to \cdot \text{id}$

$E \to \cdot V$

V

I_2:

$S \to V \cdot = E$

$E \to V \cdot$

$=$

I_6:

$S \to V = \cdot E$

$E \to \cdot V$

$V \to \cdot * E$

$V \to \cdot \text{id}$

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$S \to V \cdot = E$使得

action[2,=]=s6

项目$E \to V \cdot$使得

action[2,=]为按$E \to V$归约，

因为Follow(E) = {=, \$}

产生移进-归约冲突

不是SLR(1)但是LR(1)的文法

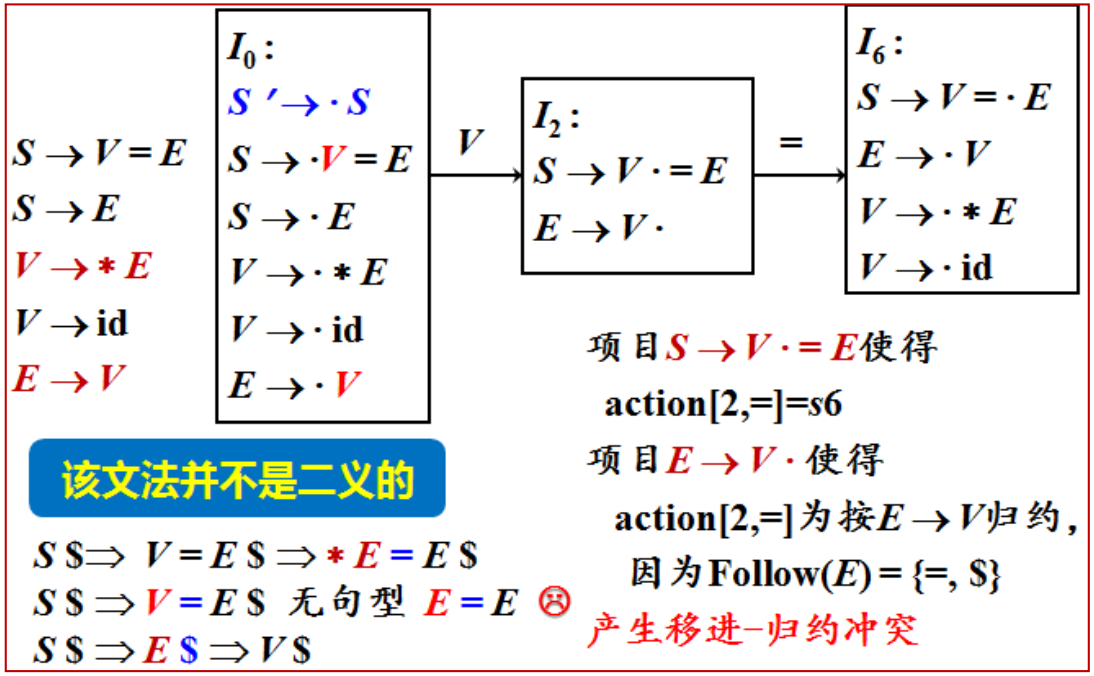
$S \to V = E$

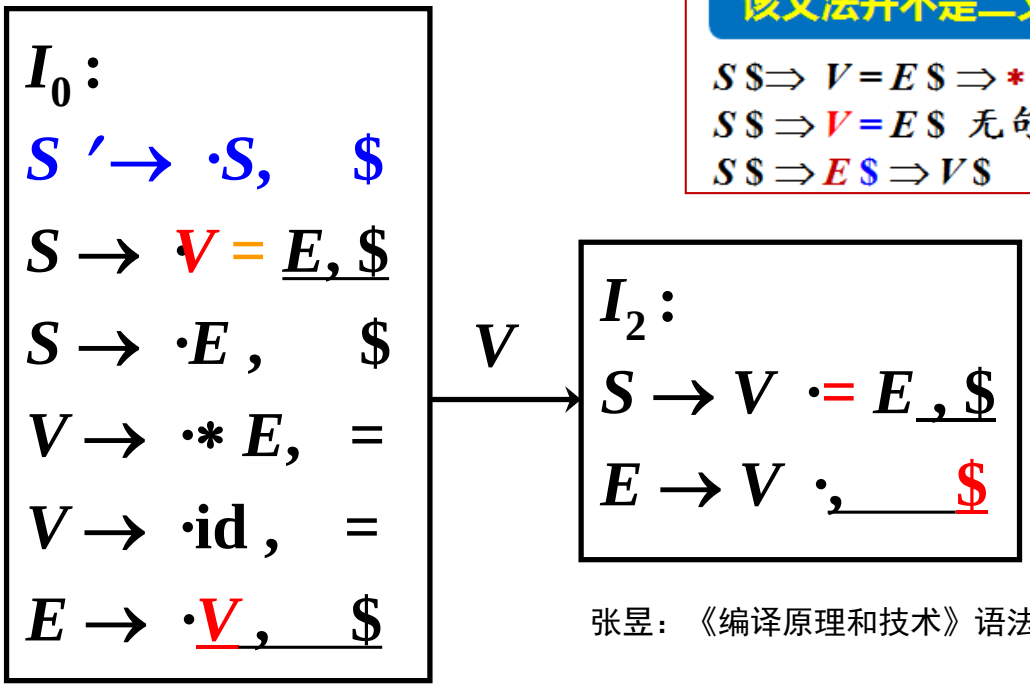
$S \to E$

$V \to * E$

$V \to \text{id}$

$E \to V$





LR(1)分析
无移进-归约
冲突

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（指规范的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　　*ababbbbaba*

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

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

是LR的　　*ababbcbbaba*

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（指规范的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

$ababbbbaba$

I_0:

$S' \rightarrow \cdot S$

$S \rightarrow a \cdot Sa$ ⟹ 面临a，*移进*

$S \rightarrow \cdot$ ⟹ $\because a \in Follow(S)$，$\therefore$面临a，*归约*

$S \rightarrow \cdot bSb$

存在移进-归约冲突

故不是SLR(1)文法

非LR的上下文无关结构

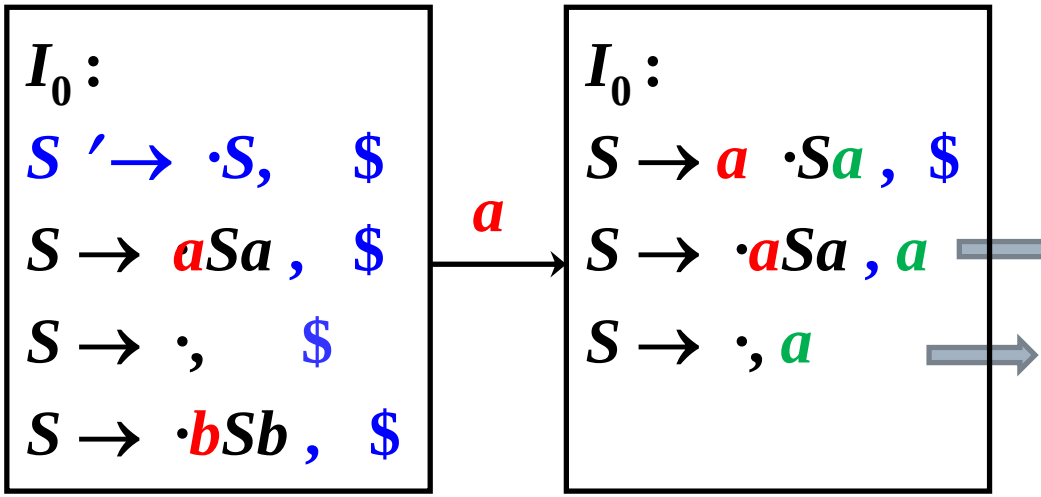
若自左向右扫描的移进–归约分析器能及时识别出现在栈顶的句柄，那么相应的文法就是LR（指规范的LR）的。

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

ababbbbaba



面临a，*移进*

∵ a在项目搜索符中

∴面临a，*归约*

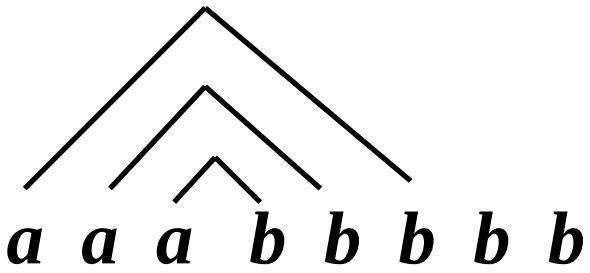
存在移进-归约冲突

故不是LR(1)文法

例题 写不同的文法

为语言$L=\{a^mb^n \mid n>m\geq 0\}$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。

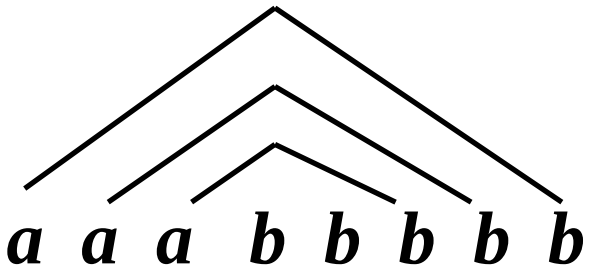
- LR(1)文法：$S \to AB$　　$A \to aAb \mid \varepsilon$　　$B \to Bb \mid b$



例题 写不同的文法

为语言$L=\{a^mb^n \mid n>m\geq 0\}$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。

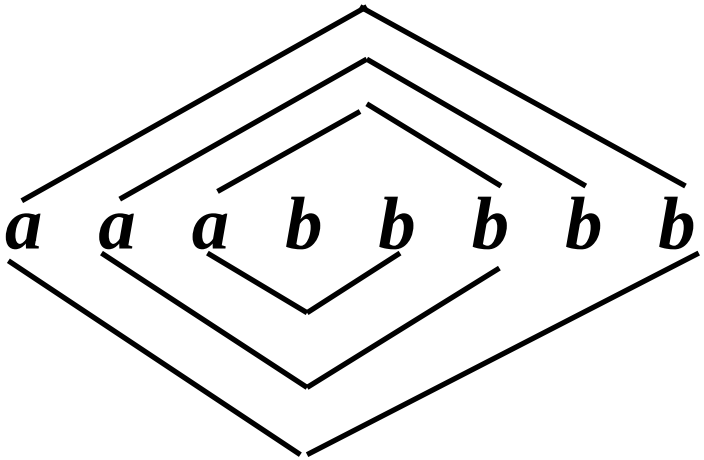
- LR(1)文法：$S \to AB \quad A \to aAb \mid \varepsilon \quad B \to Bb \mid b$
- 非二义且非LR(1)的文法：$S \to aSb \mid B \quad B \to Bb \mid b$



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：$S \to AB \quad A \to aAb \mid \varepsilon \quad B \to Bb \mid b$
- 非二义且非LR(1)的文法：$S \to aSb \mid B \quad B \to Bb \mid b$
- 二义的文法：$S \to aSb \mid Sb \mid b$

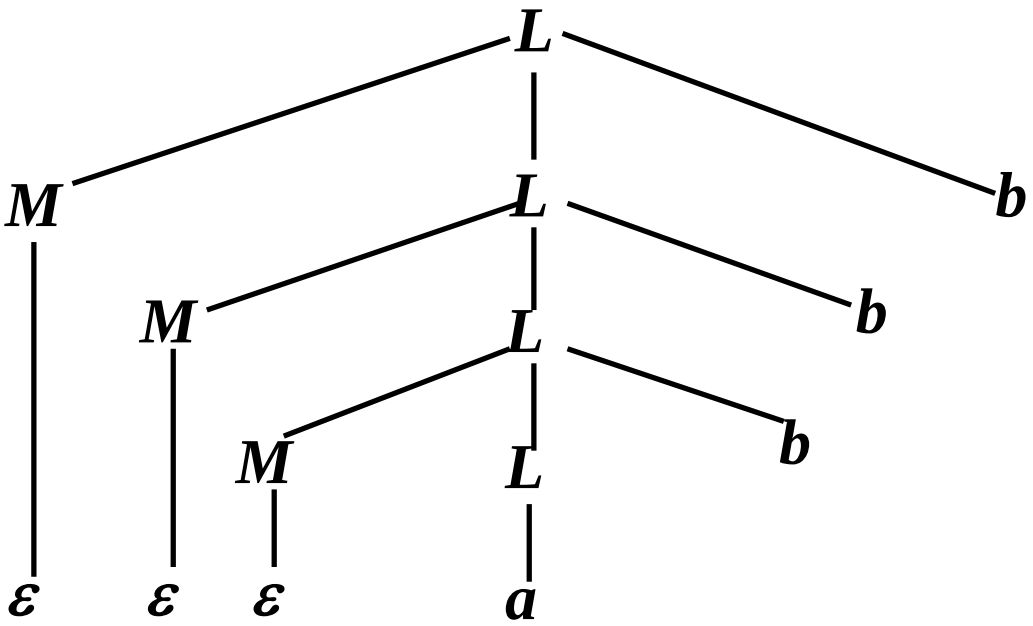


例题4

试说明下面文法不是LR(1)的：

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

$M \rightarrow \varepsilon$



句子*abbb*的分析树

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

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 → begin *declist* ; *statement* end

declist → d ; *declist* | d

statement → 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 → begin *declist* *statement* end

declist → d ; *declist* | d ;

statement → 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).

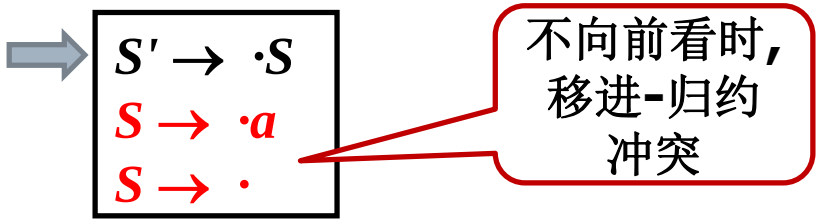
是否违反了LR分析的活前缀性质？

LR项目与LR文法小结

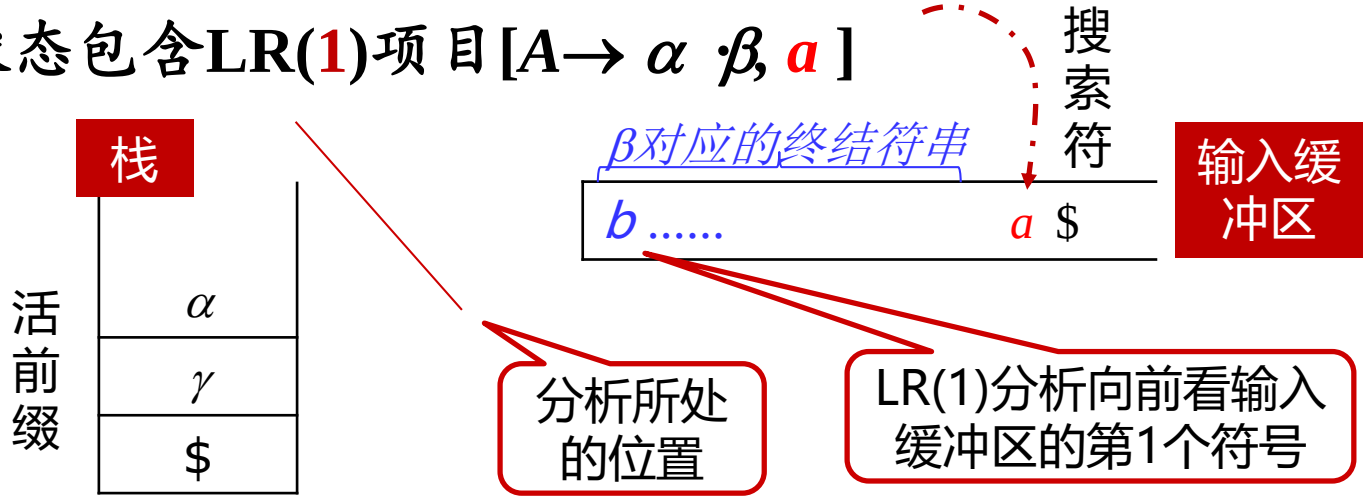
- LR(0)项目[$A \rightarrow \alpha \cdot \beta$]、LR(1)项目[$A \rightarrow \alpha \cdot \beta, a$]
  - 数字表示向前搜索的符号个数，0表示不向前搜索符号
- SLR(k)分析技术与SLR(k)文法
  - SLR(1)分析的状态：LR(0)项目集
  - k是指向前看输入缓冲区的k个符号
- [规范的]LR(k)分析技术与LR(k)文法
  - LR(1)分析的状态：LR(1)项目集
- LALR(k)分析技术与LALR(k)文法
  - LR(1)分析的状态：LR(1)项目集+同心项目集合并

LR项目与LR文法小结

- 不是SLR(0)文法，但是SLR(1)文法
  - 例：S → a | ε



- SLR(0)文法
  - S → a　S → a | b
- 理解LR(1)项目与LR(1)文法中的1
  - 若栈顶状态包含LR(1)项目[A→ α ·β, a]



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

3.6 二义文法的应用

- 通过其他手段消除二义文法的二义性
- LR分析的错误恢复

二义文法的特点

- 特点
  - 绝不是LR 文法
  - 简洁、自然

例 二义文法 $E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$

非二义的文法：

$E \to E + T \mid T$

$T \to T * F \mid F$

$F \to (E) \mid \mathrm{id}$

该文法有单个非终结符为右部的产生式（简称单非产生式）

单非产生式会增加
分析树的高度
=> 分析效率降低

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 $E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$

规定：*优先级高于+，两者都是左结合

LR(0)项目集I_7

$E \to E + E \cdot$

$E \to E \cdot + E$

$E \to E \cdot * E$

id + id + id

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 $E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$

规定：*优先级高于+，两者都是左结合

LR(0)项目集I_7

$E \to E + E \cdot$ id + id + id

$E \to E \cdot + E$ id + id * id

$E \to E \cdot * E$

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 $E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$

规定：*优先级高于+，两者都是左结合

LR(0)项目集I_8

$E \to E * E \cdot$ id * id + id

$E \to E \cdot + E$

$E \to E \cdot * E$

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 $E \to E + E \mid E * E \mid (E) \mid \mathrm{id}$

规定：*优先级高于+，两者都是左结合

LR(0)项目集I_8

$E \to E * E \cdot$

$E \to E \cdot + E$

$E \to E \cdot * E$

id * id + id

id * id * id

面临+，归约

面临*，归约

面临)和$，归约

特殊情况引起的二义性

$E \rightarrow E$ sub E sup E

$E \rightarrow E$ sub E

$E \rightarrow E$ 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 sub i sup 2，需要下面第一种输出

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

特殊情况引起的二义性

$E \rightarrow E$ sub E sup E

$E \rightarrow E$ sub E

$E \rightarrow E$ sup E

$E \rightarrow \{E\}$

$E \rightarrow c$

I_{11}:

$E \rightarrow E$ sub E sup E ·

$E \rightarrow E$ sup E ·

…

按前面一个产生式归约

LR分析的错误恢复

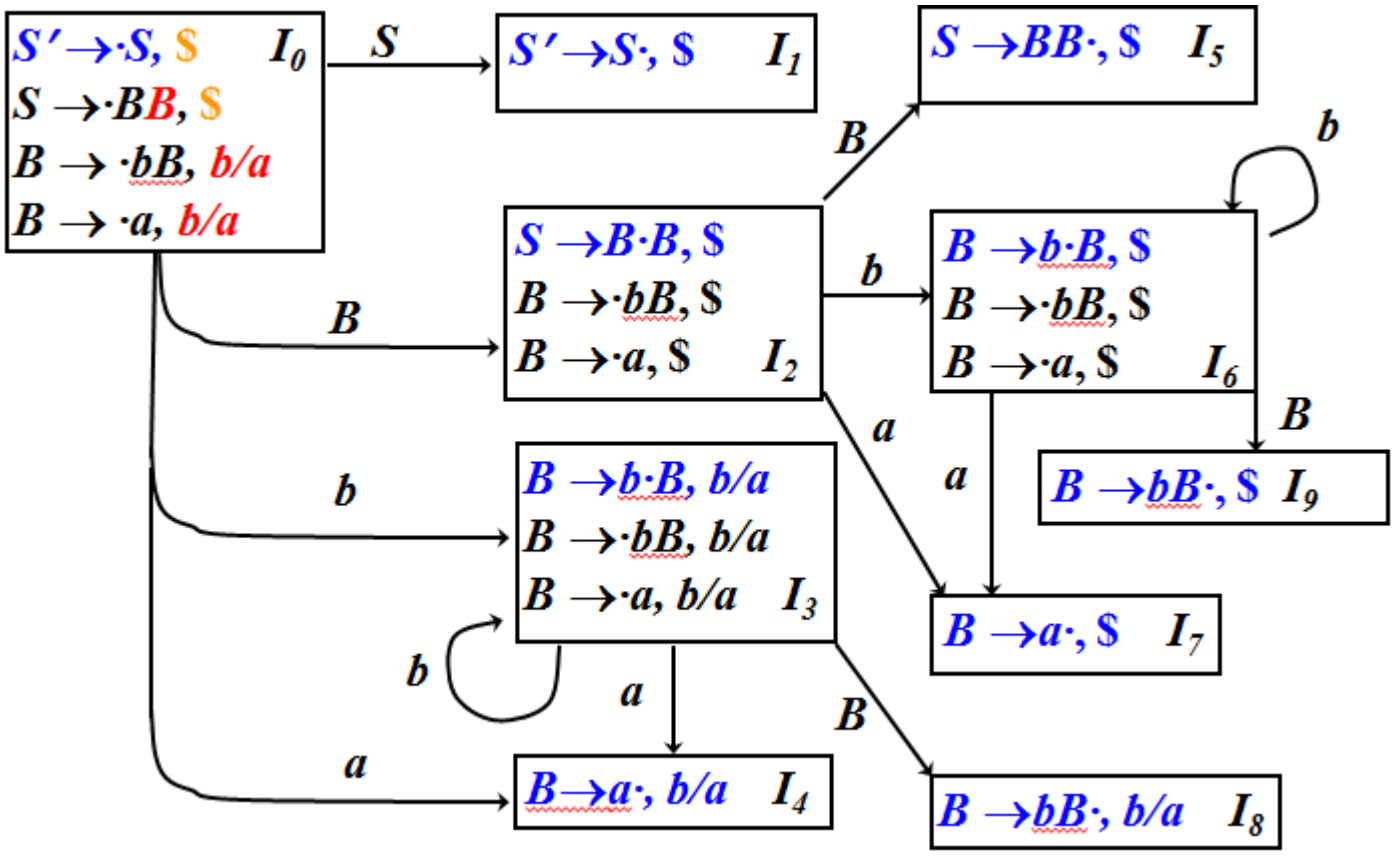
- LR分析器在什么情况下发现错误
  - 访问action表时遇到出错条目
  - 访问goto表时绝不会遇到出错条目
  - 绝不会把不正确的后继移进栈
  - 规范的LR分析器在报告错误之前决不做任何无效归约
    - SLR和LALR在报告错误前有可能执行几步无效归约

规范的LR分析不会把错误移进

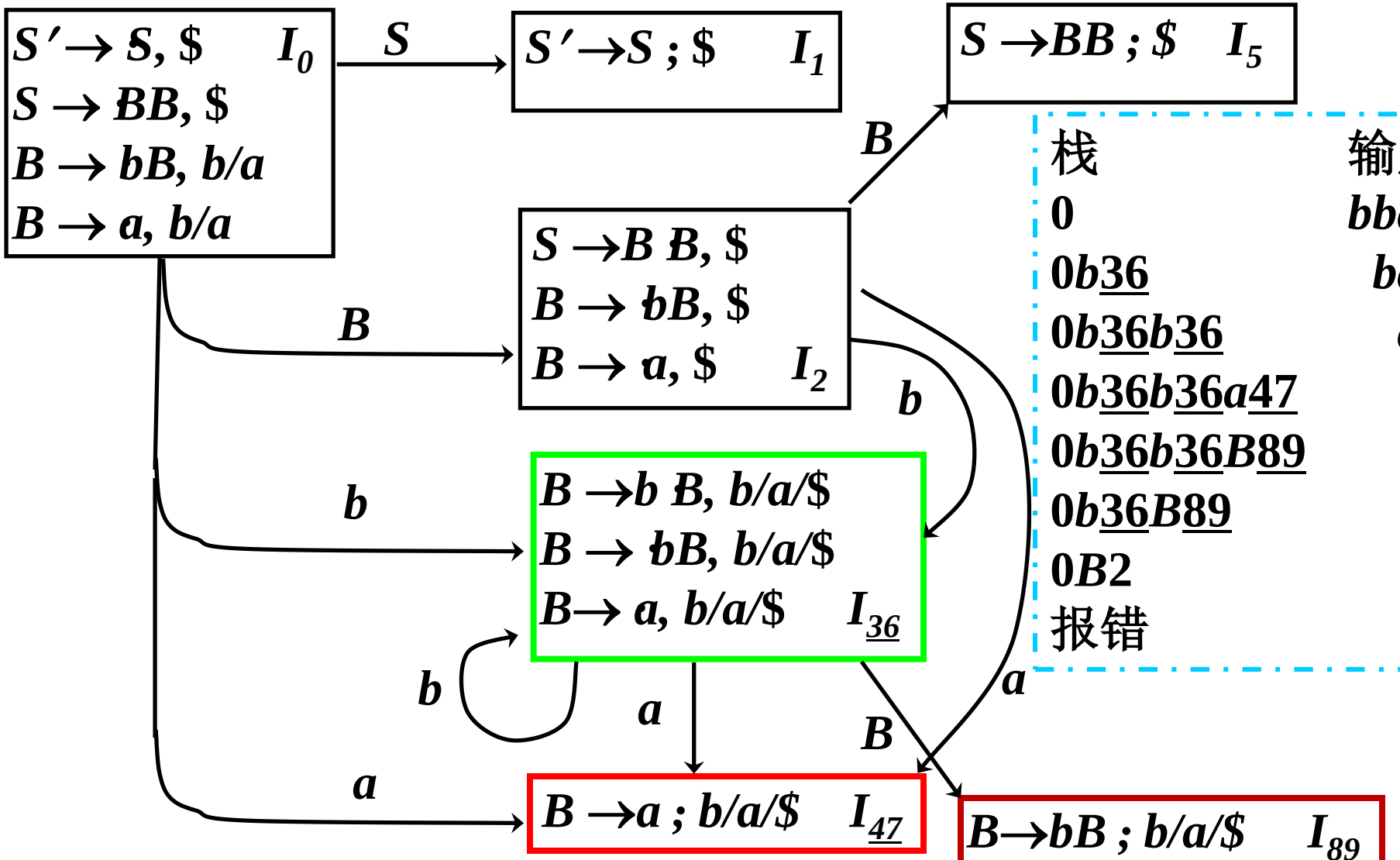
给出在以下两种输入下的LR分析过程

bbabba$

bba$



LALR分析也不会把错误移进



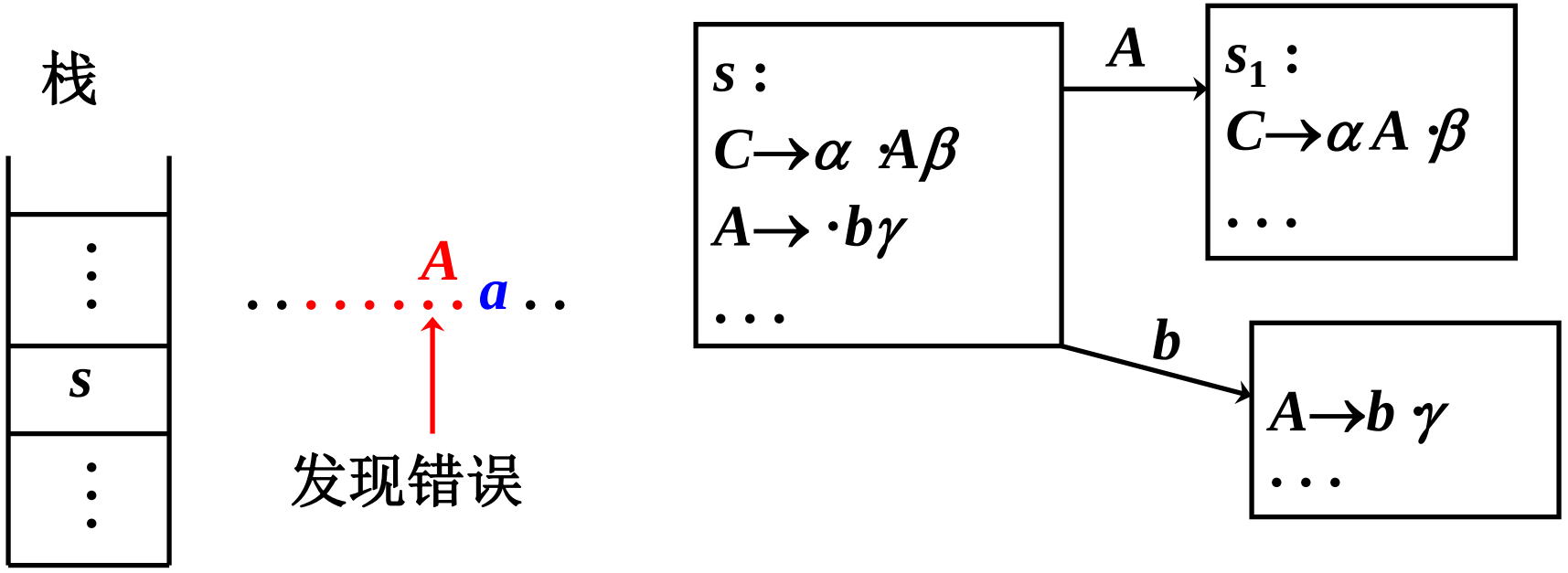
栈	输入
0	bba$
0b36	ba$
0b36b36	a$
0b36b36a47	$
0b36b36B89	$
0b36B89	$
0B2	$
报错	

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

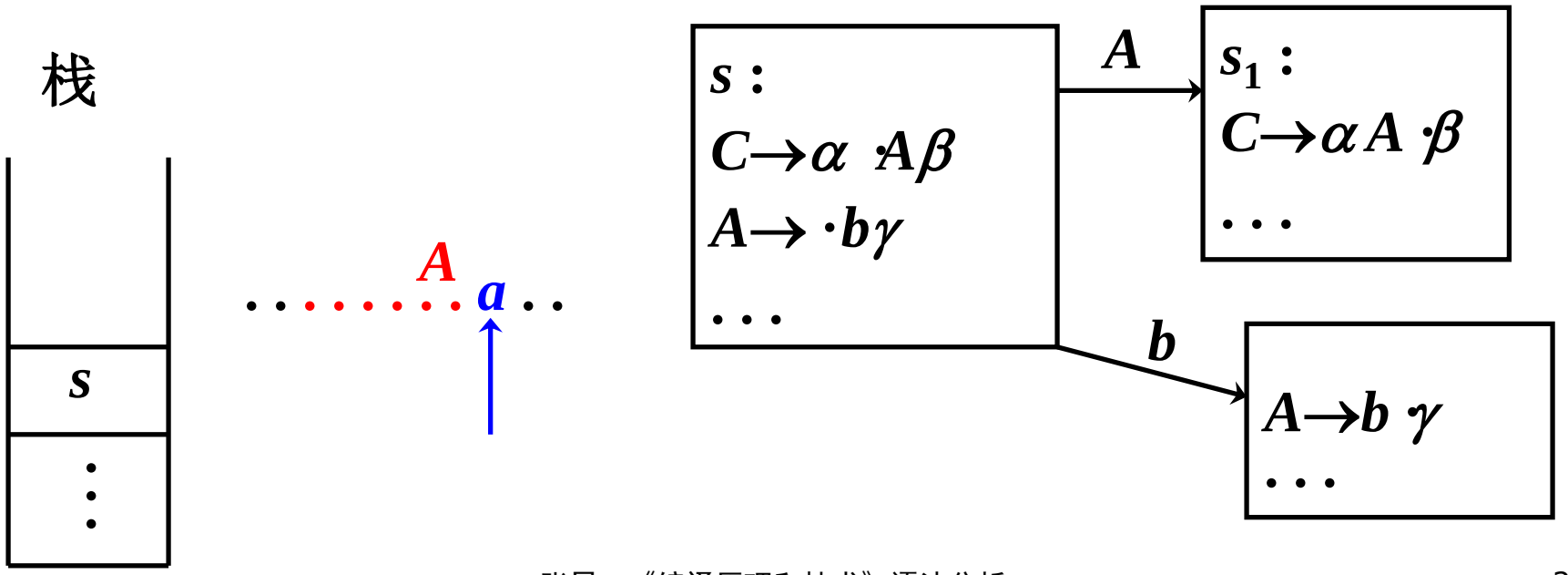


紧急方式的错误恢复

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

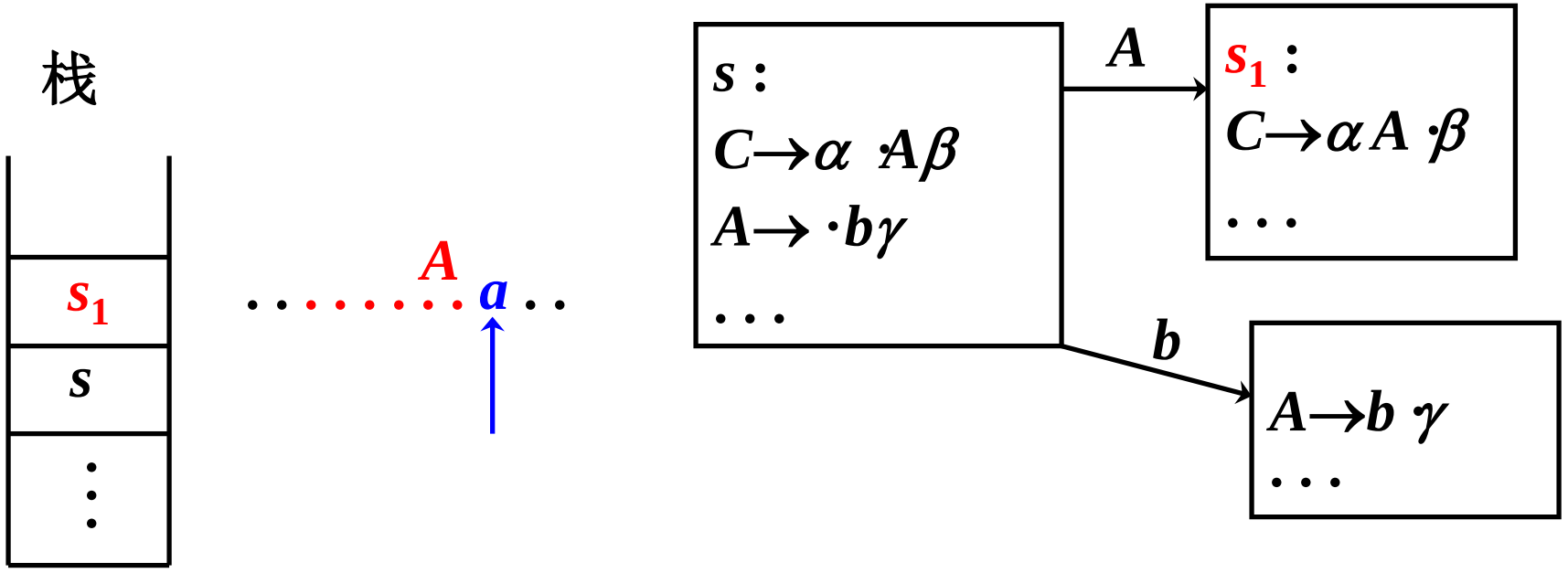


紧急方式的错误恢复

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

- 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

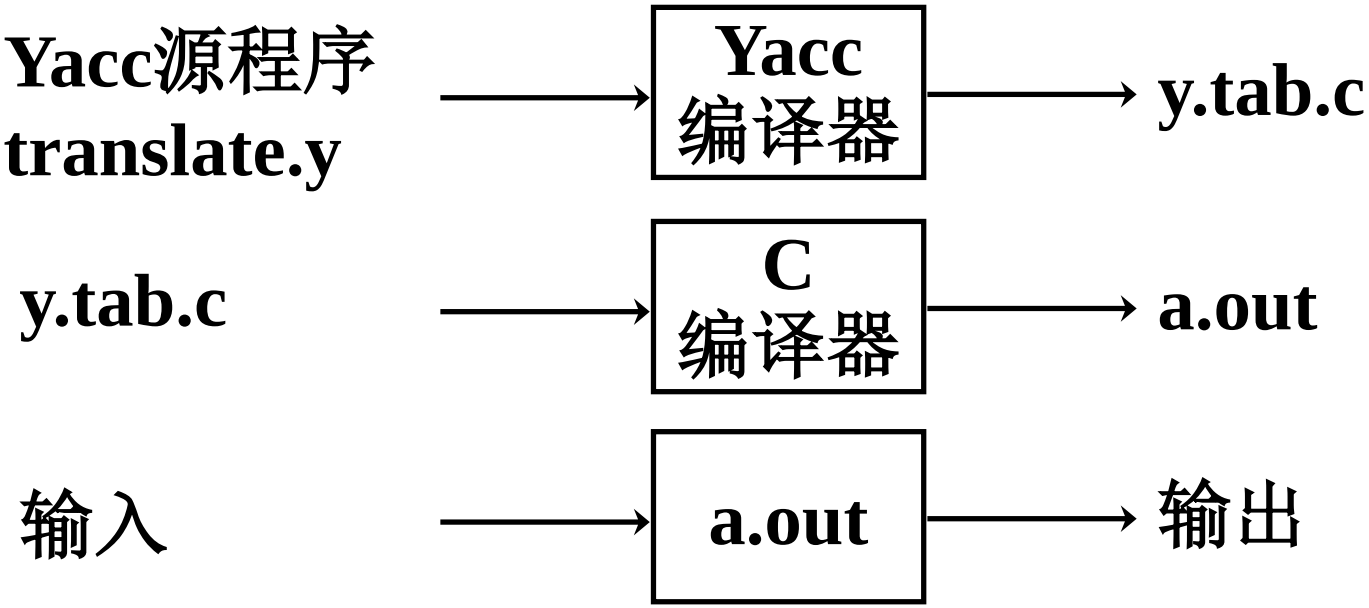
中国科学技术大学
University of Science and Technology of China

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

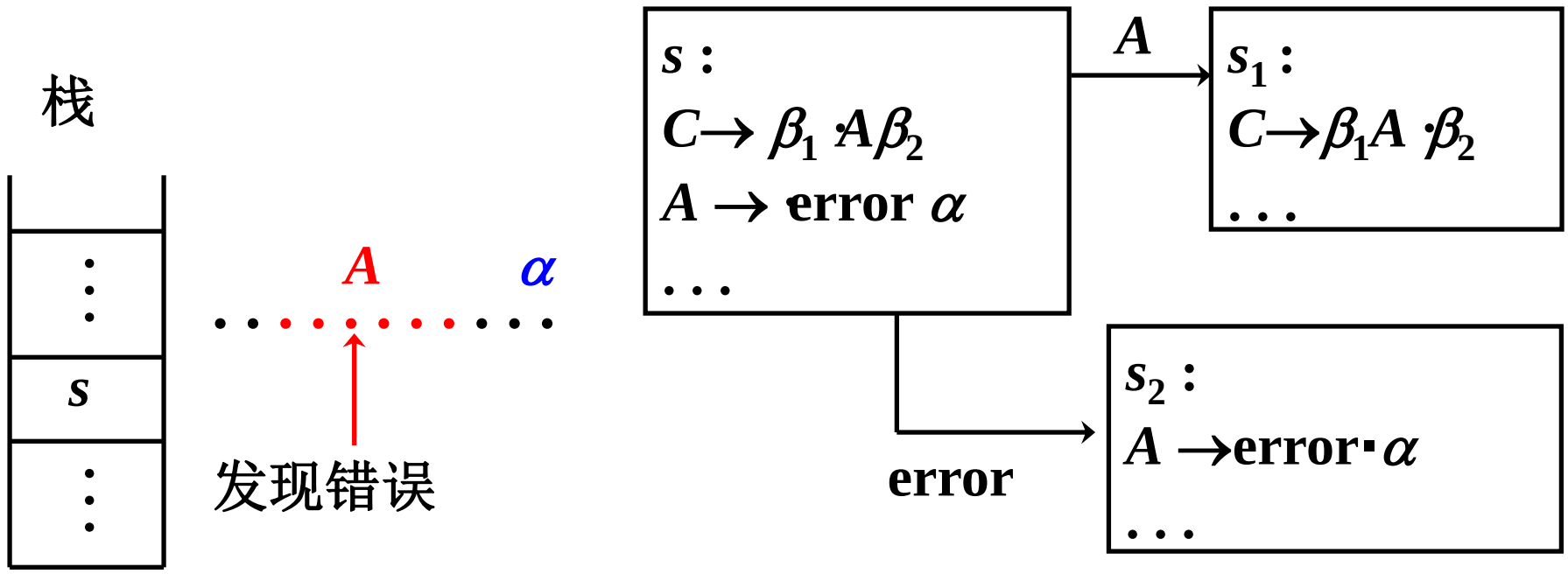
C例程部分

```
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

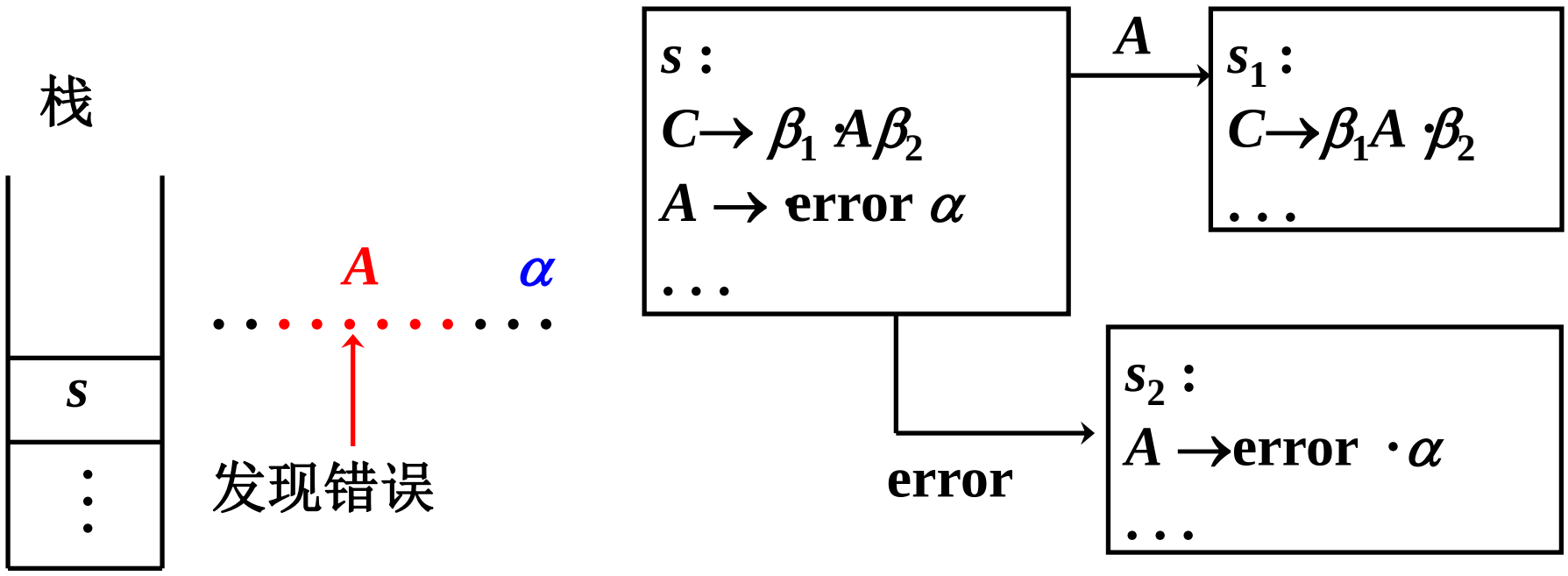
YACC的错误恢复

- 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

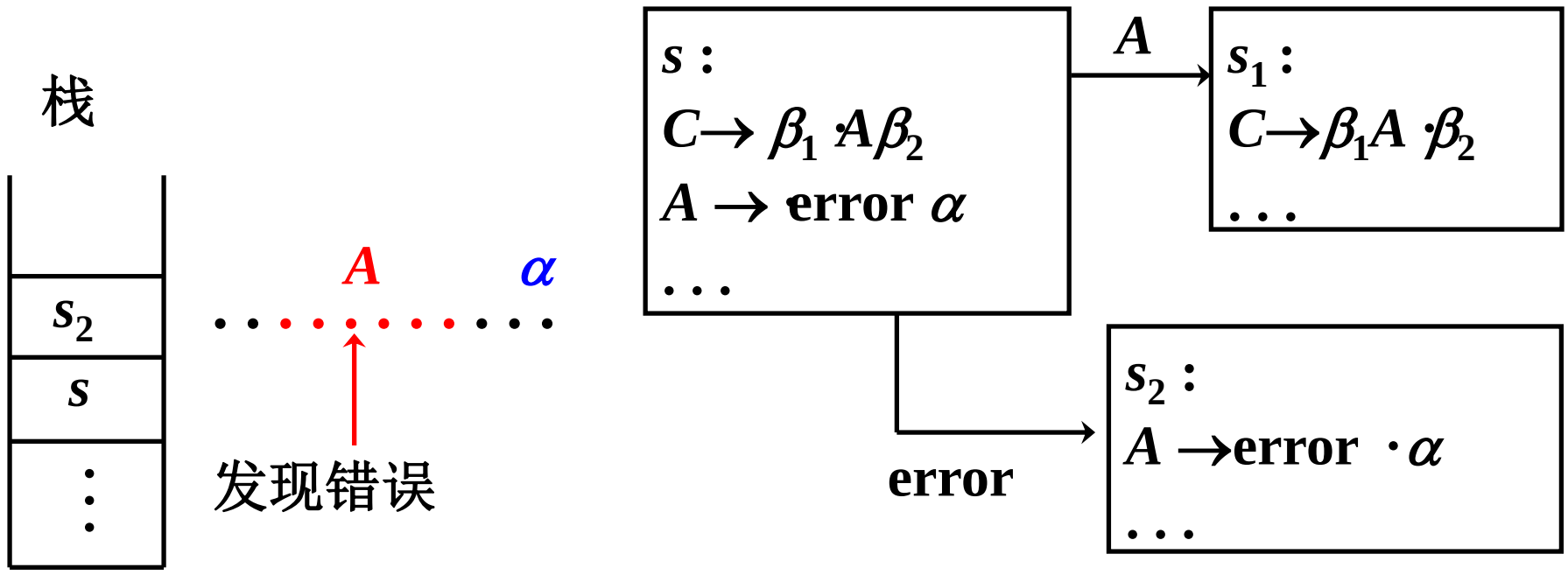
- 遇到语法错误时
  - 从栈中弹出状态，直到发现栈顶状态的项目集包含形为 $A \rightarrow \text{error } \alpha$ 的项目为止



YACC的错误恢复

遇到语法错误时

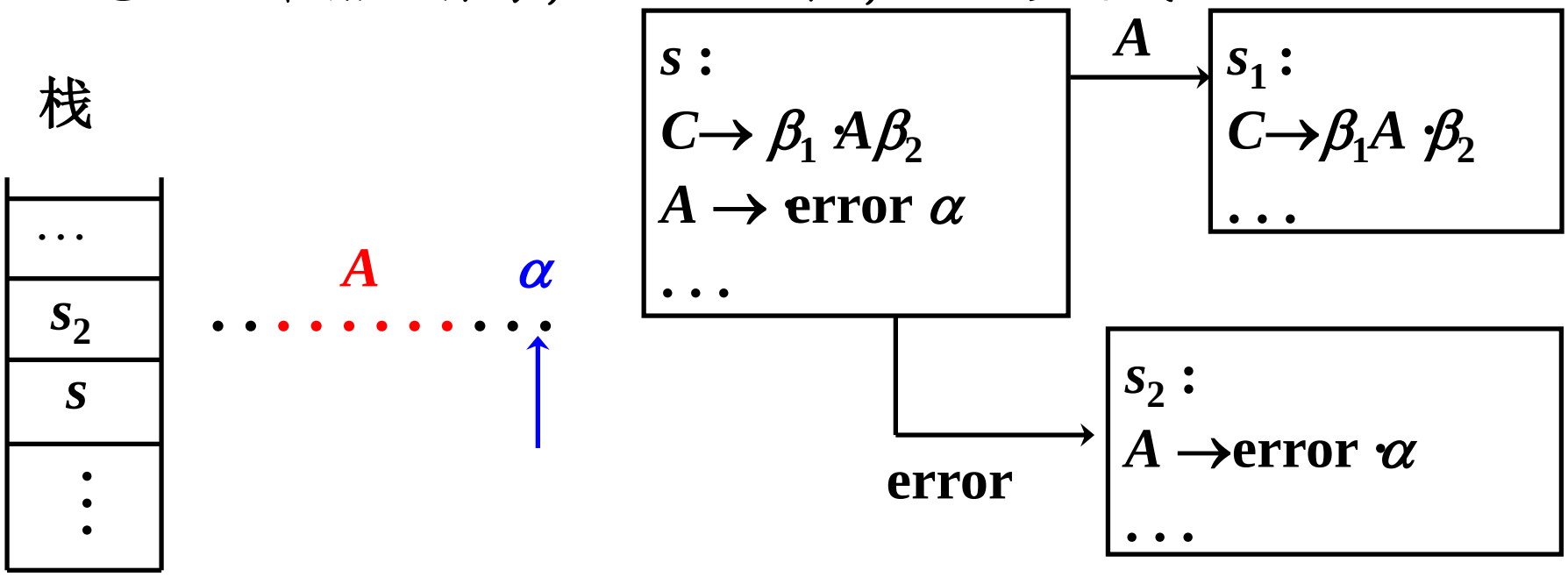
- 从栈中弹出状态，直到发现栈顶状态的项目集包含形为 $A \rightarrow \text{error}\ \alpha$的项目为止
- 把虚构的终结符error“移进”栈



YACC的错误恢复

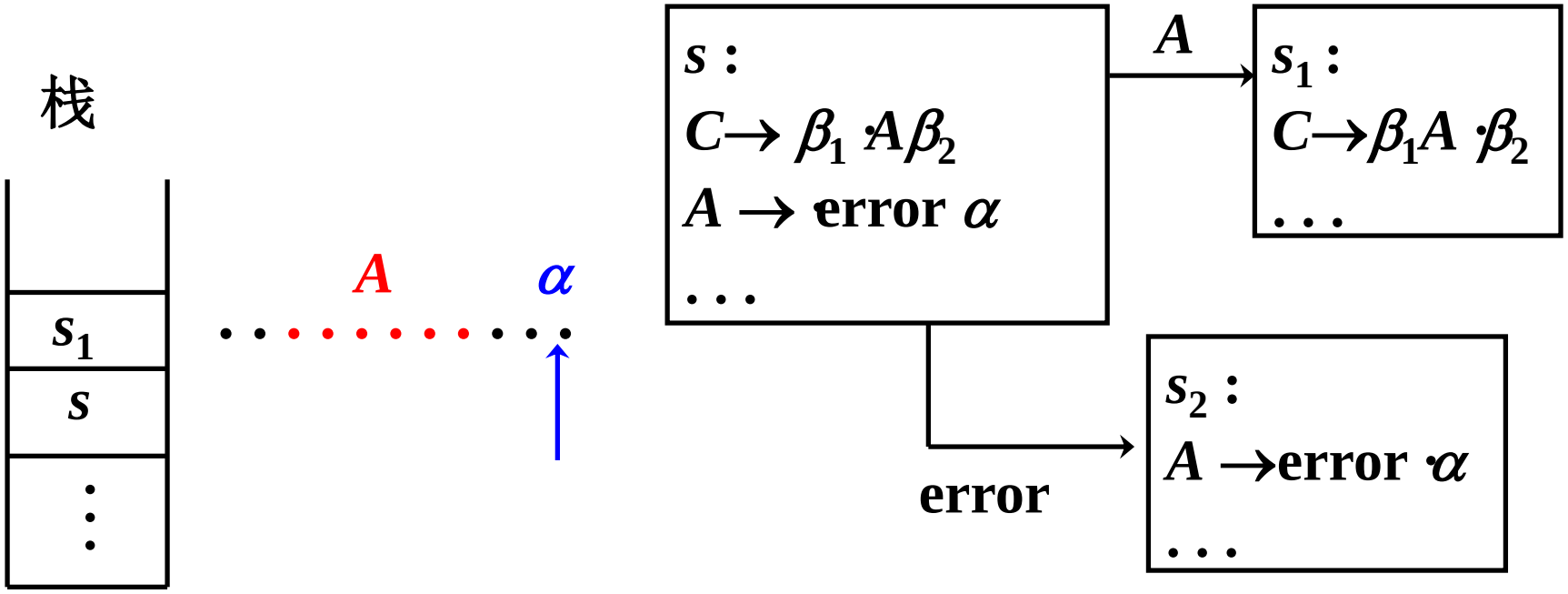
遇到语法错误时

- 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α，把α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α，把α移进栈
- 把error α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'
            |  /* ε */
            | error '\n'{yyerror ( "重新输入上一行" );
                                yyerrok;}
            ;
```