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》

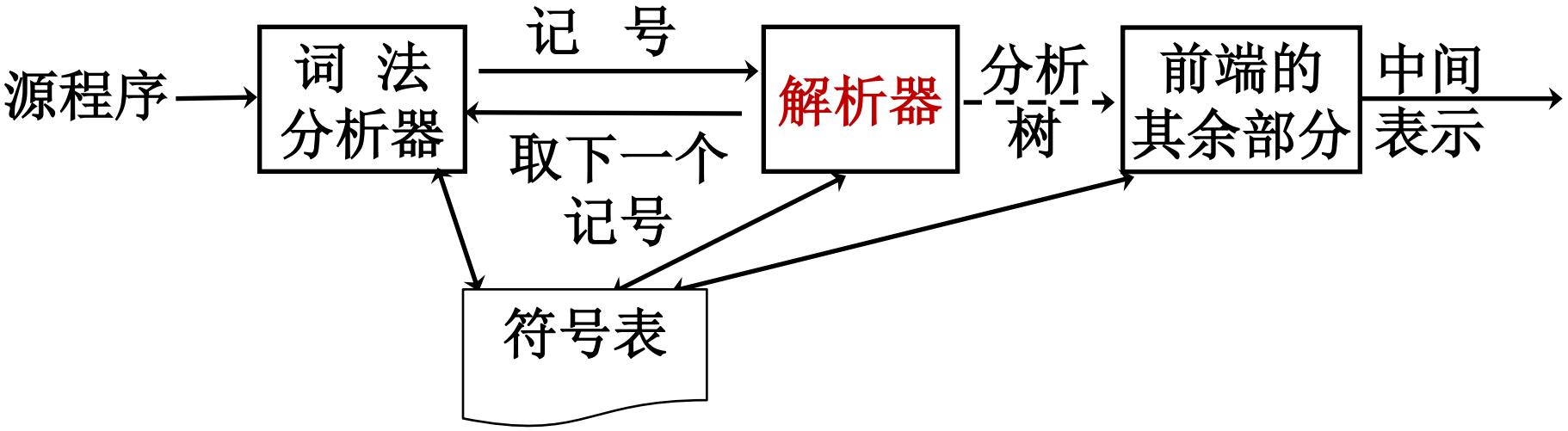
张昱

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)、SLR、LR(k)、LALR

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a和b的串$\}$

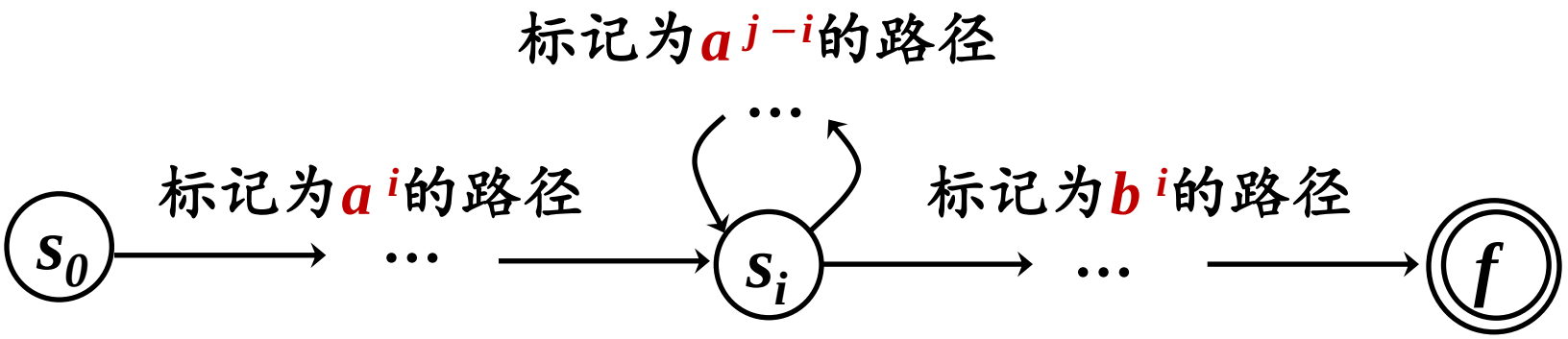
  原因：n不固定，且后面的串要依据前面不定长的串来确定；在有限的状态下不能表达

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$\varepsilon, a, aa, \ldots, a^k$ 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：$A \to \alpha$，有时用 $A ::= \alpha$

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

    $expr \to expr\ op\ expr$　　$expr \to (expr)$　　$expr \to - expr$

    $expr \to \mathrm{id}$　　$op \to +$　　$op \to *$

CFG的简化表示

表达式

- 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

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

$A \rightarrow + \mid *$

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

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

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 $S \Rightarrow^+ w$

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

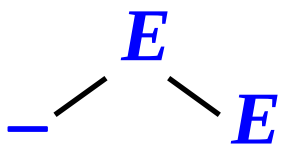
  $E \Rightarrow_{rm} -E \Rightarrow_{rm} -(E) \Rightarrow_{rm} -(E + E)$

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

分析树(parse tree)

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

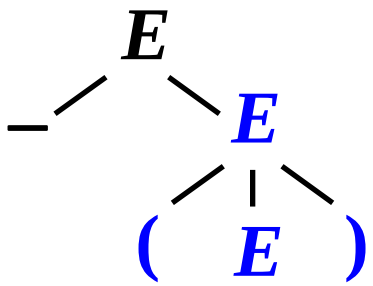
-(id+id)最左推导的分析树



分析树(parse tree)

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

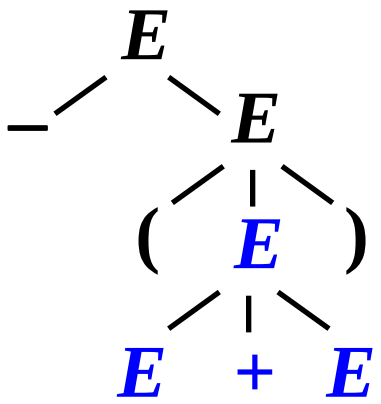
-(id+id)最左推导的分析树



分析树(parse tree)

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

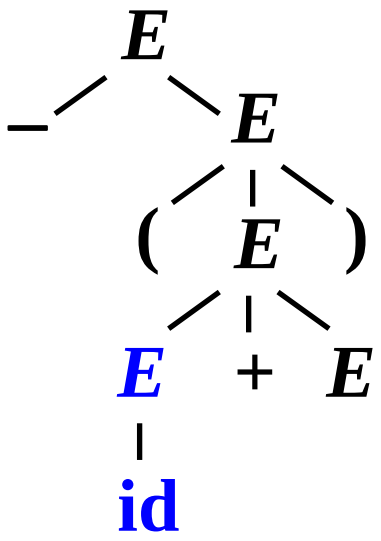
-(id+id)最左推导的分析树



分析树(parse tree)

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

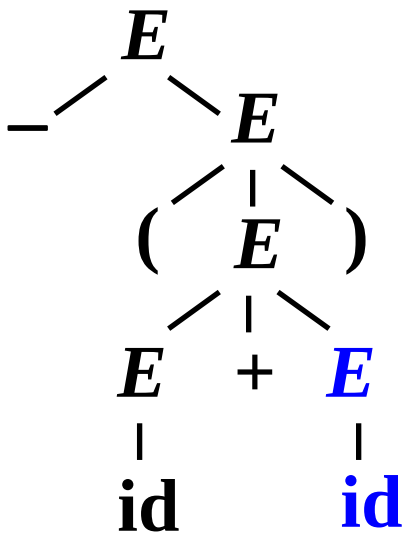
-(id+id)最左推导的分析树



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



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

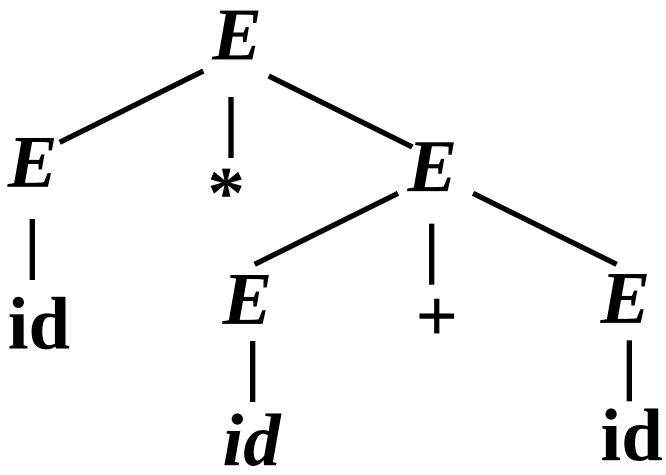
$$
\begin{aligned}
E &\Rightarrow E * E \\
&\Rightarrow \mathrm{id} * E \\
&\Rightarrow \mathrm{id} * E + E \\
&\Rightarrow \mathrm{id} * \mathrm{id} + E \\
&\Rightarrow \mathrm{id} * \mathrm{id} + \mathrm{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow E * E + E \\
&\Rightarrow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

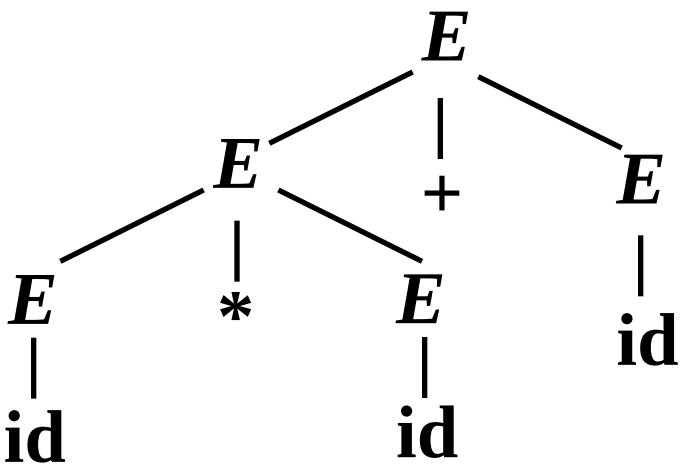
文法的二义性

id*id+id 有两棵不同的分析树

$E \Rightarrow E * E$
$\Rightarrow$ 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





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

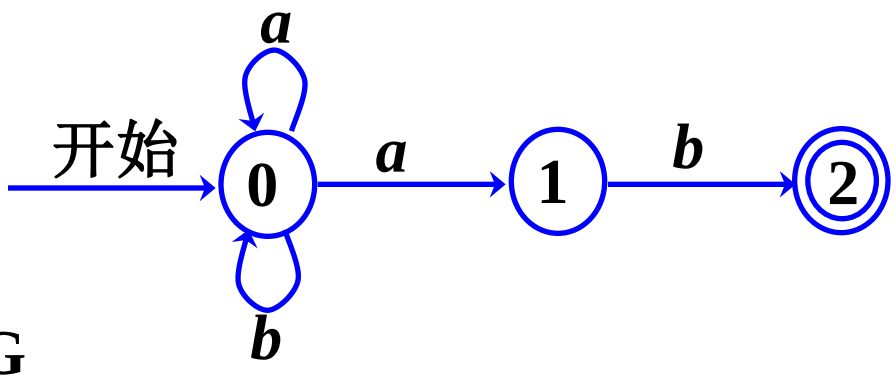
3.2 语言和文法

- 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 ')' $S \mid \varepsilon$　$L(G)$ = 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

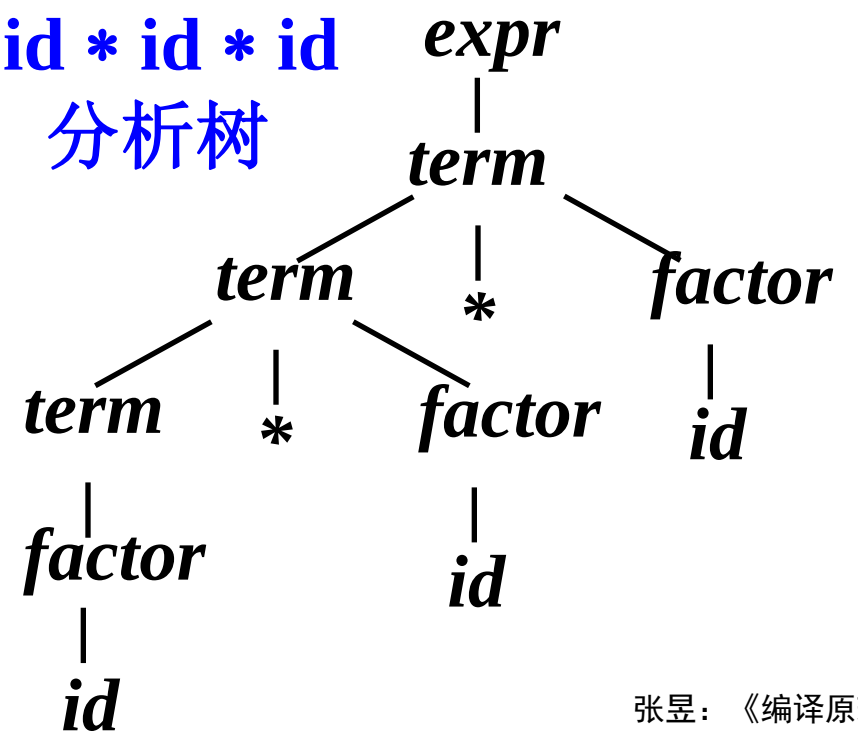
表达式的另一种文法

$expr \rightarrow expr + term \mid term$

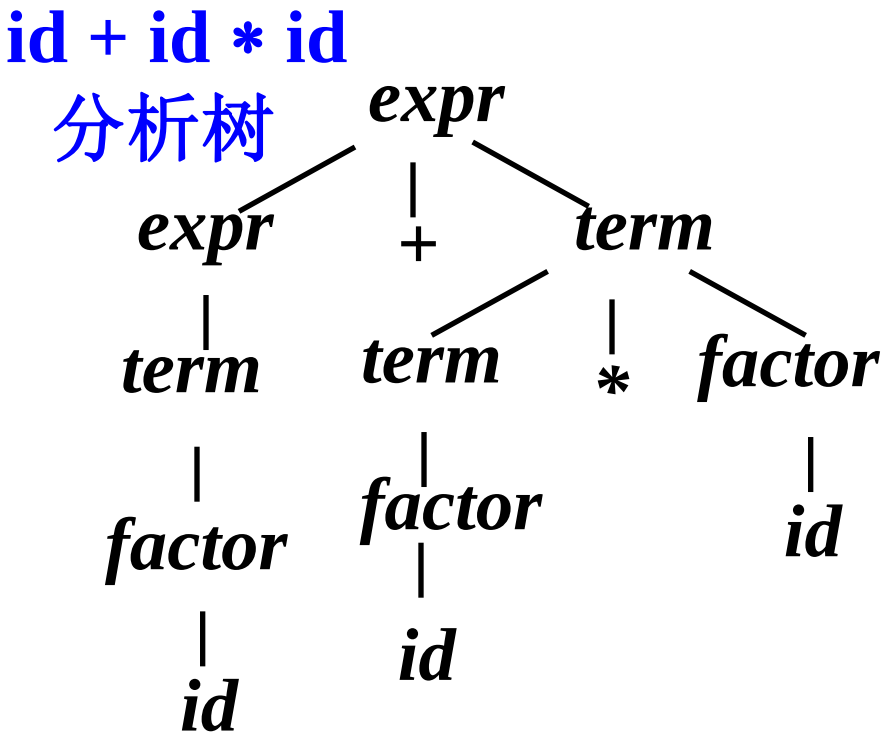
$term \rightarrow term * factor \mid factor$

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

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

else 的就近匹配规则

stmt → *matched _stmt*
| *unmatched_stmt*

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*

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)

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

$$E \to E + T \mid T \qquad (T + T \ldots + T)$$

$$T \to T * F \mid F \qquad (F * F \ldots * F)$$

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

消除左递归后文法

$$E \to TE'$$

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

$$T \to FT'$$

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

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

消除非直接左递归

- 间接左递归

$$S \rightarrow Aa \mid b$$

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

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

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

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

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

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

  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

- 文法：LL(1)、LL(k) 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

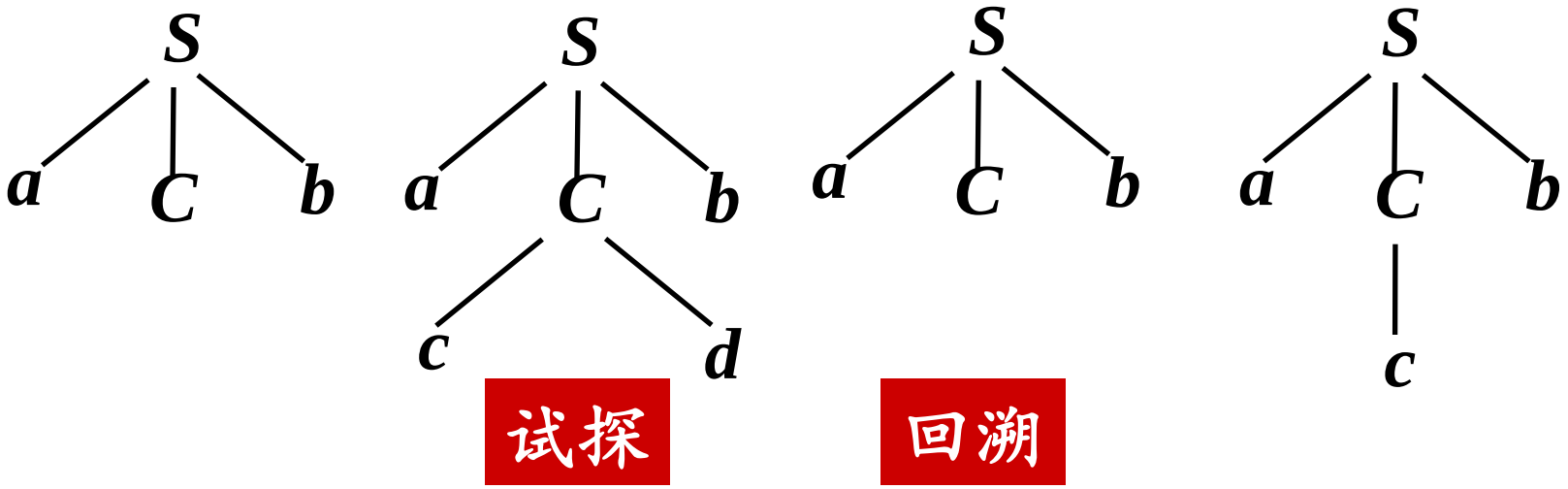
自上而下top-down分析

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　$S \to aCb$　$C \to cd \mid c$

为输入串$w = acb$建立分析树

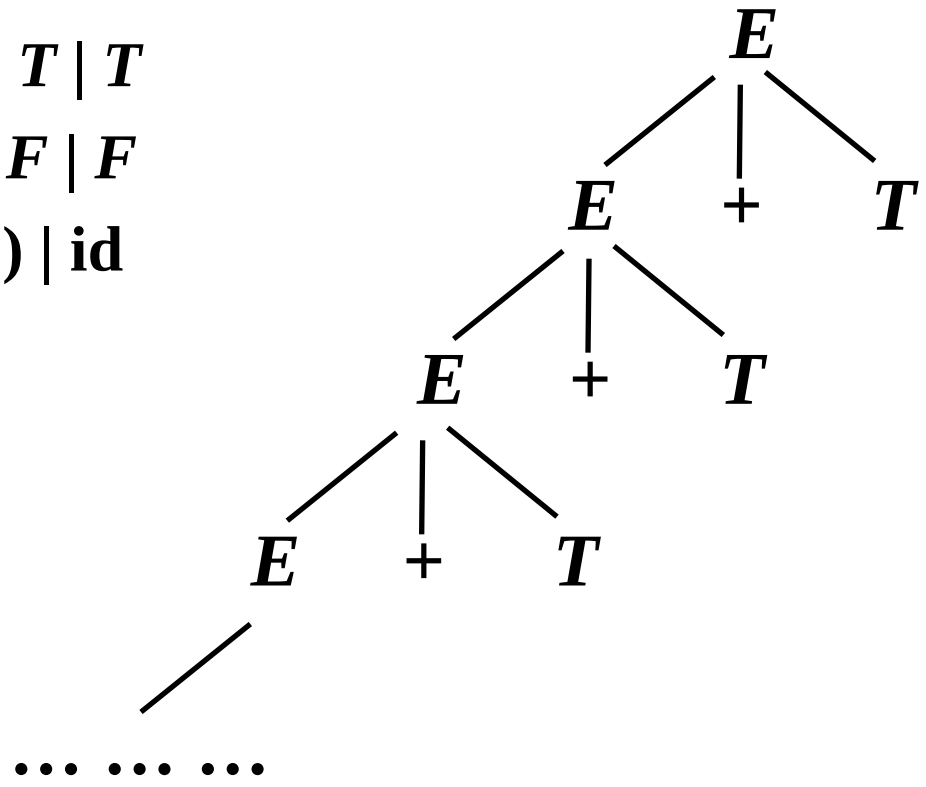


自上而下分析：左递归

- 不能处理左递归文法

算术表达文法

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



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
  - FIRST(α) = {$a \mid \alpha \Rightarrow^* a\ldots, a \in V_T$}

    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$)

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)

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

$S \to A\ B$

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

$B \to a\ C$

$C \to \ldots$

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

$$
\begin{aligned}
\textit{type} &\rightarrow \textit{simple} \\
&\quad \mid \uparrow \textbf{id} \\
&\quad \mid \textbf{array}\ [\textit{simple}]\ \textbf{of}\ \textit{type} \\
\textit{simple} &\rightarrow \textbf{integer} \\
&\quad \mid \textbf{char} \\
&\quad \mid \textbf{num dotdot num}
\end{aligned}
$$

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( );
}
```

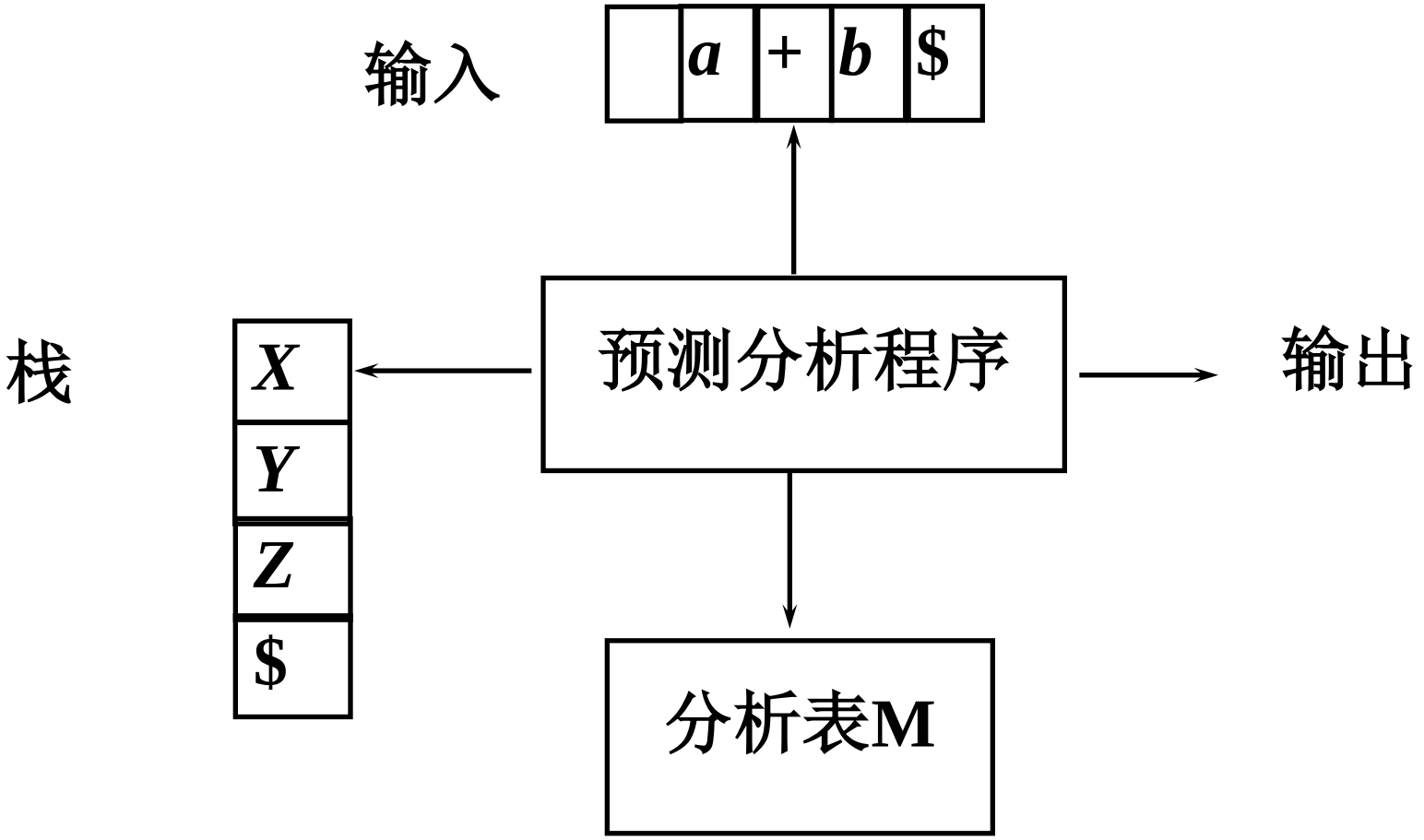
递归下降的预测分析器

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( );
}
```

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* → *TE '*
$*E 'T 'F*	id * id + id$	*T* → *FT '*
$*E 'T '* id	id * id + id$	*F* → id
$*E 'T '*	* id + id$	
$*E 'T 'F* *	* id + id$	*T '* → **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α

- **构造方法**

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

  （2）对FIRST(α)的每个终结符a，把$A \rightarrow \alpha$ 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 \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* → 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*		

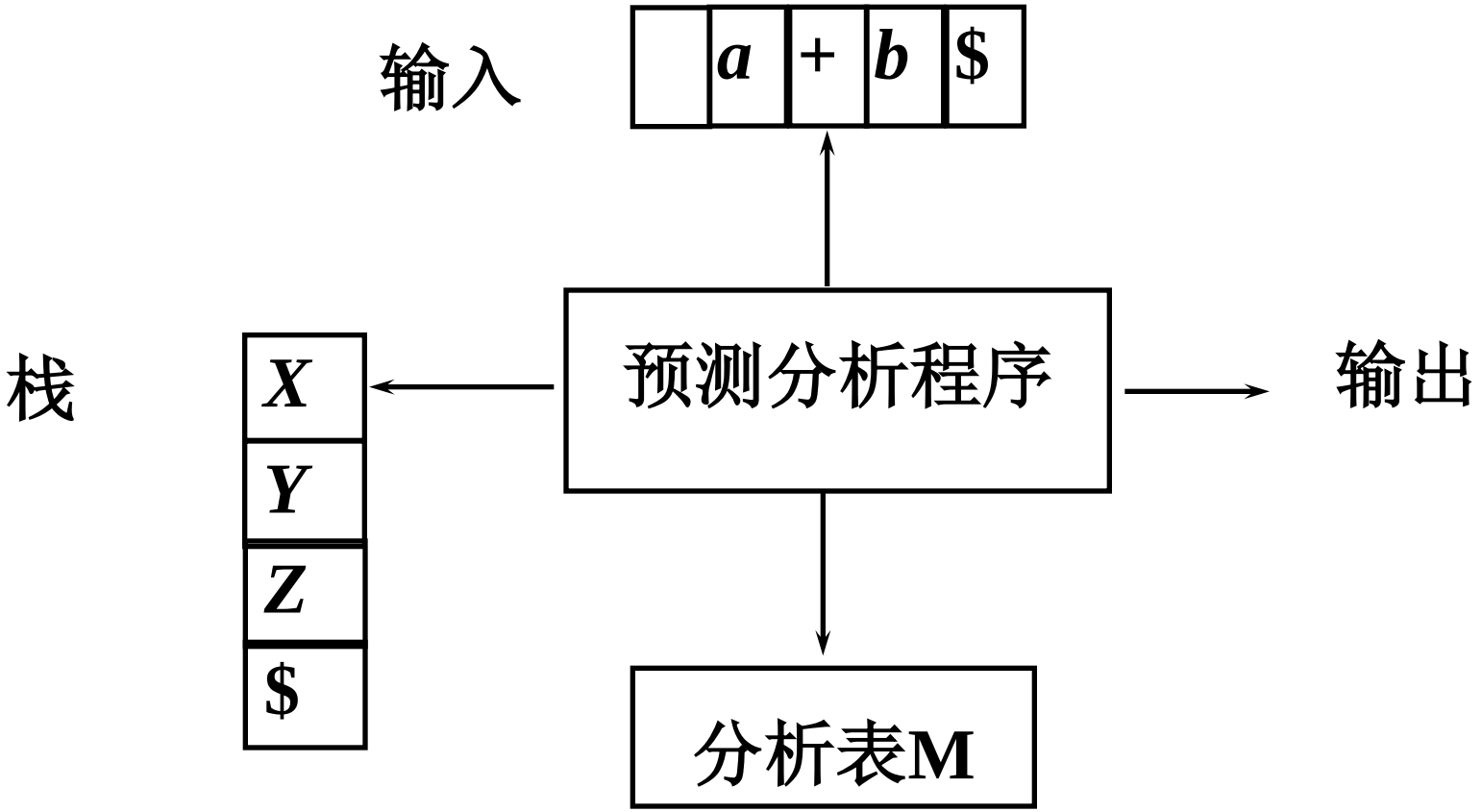
预测分析的错误恢复

- 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

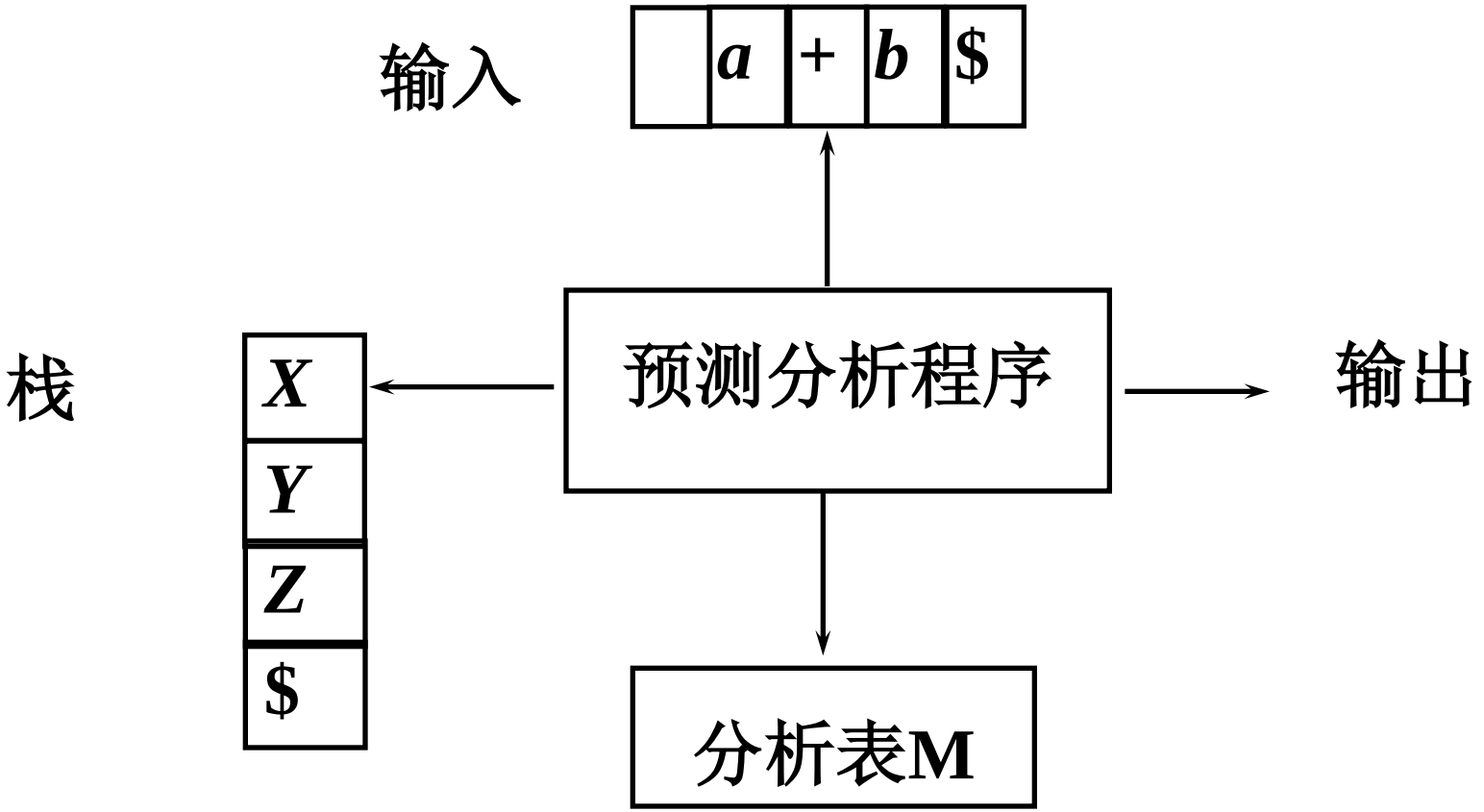
预测分析的错误恢复

- 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 = *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	+	*	...
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	$E\rightarrow TE'$			
E'		$E'\rightarrow +TE'$		
T	$T\rightarrow FT'$			
T'	出错 用$T'\rightarrow \varepsilon$	$T'\rightarrow \varepsilon$	$T'\rightarrow *FT'$	
...				

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 $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

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　$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$abbcde$（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

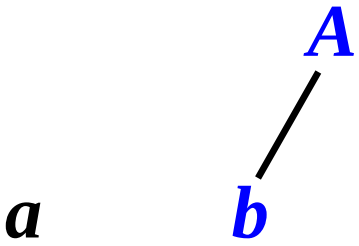
例　$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

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

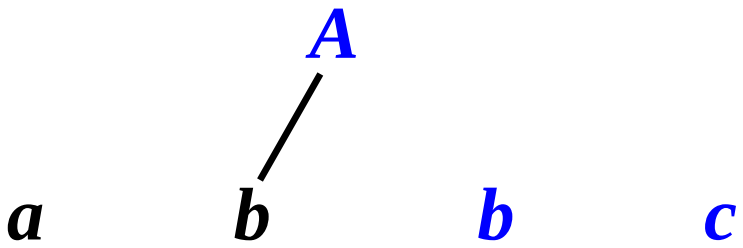
例 $S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

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　$S \rightarrow aABe$

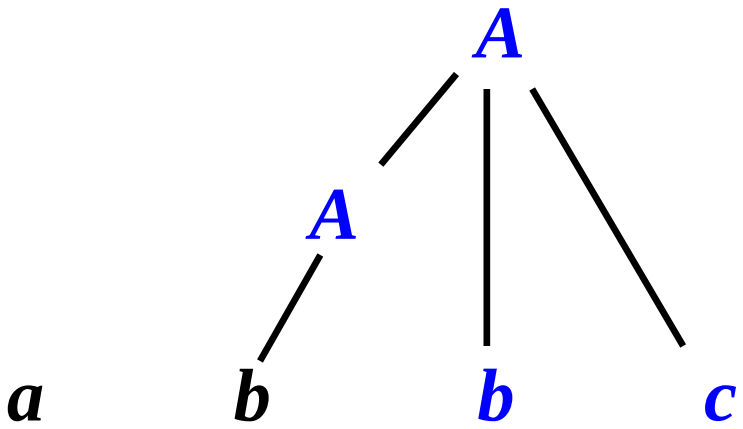
$A \rightarrow Abc \mid b$

$B \rightarrow d$

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　$S \rightarrow aABe$

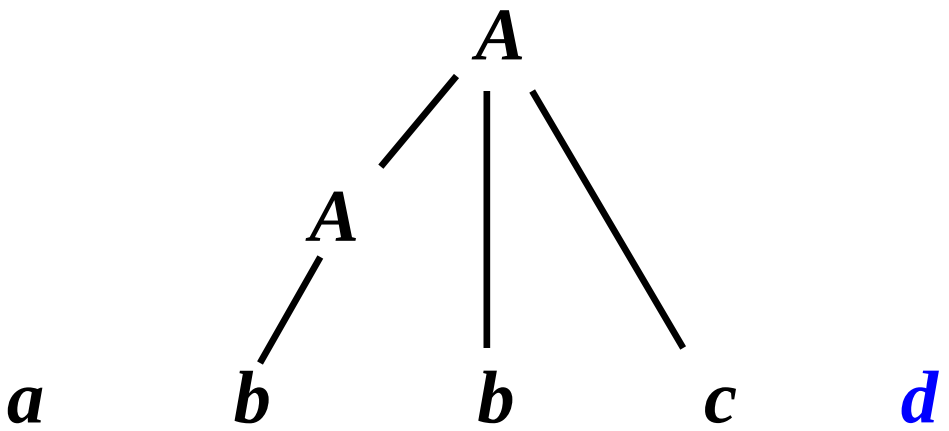
$A \rightarrow Abc \mid b$

$B \rightarrow d$

$abbcde$

$aAbcde$

$aAde$（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　$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

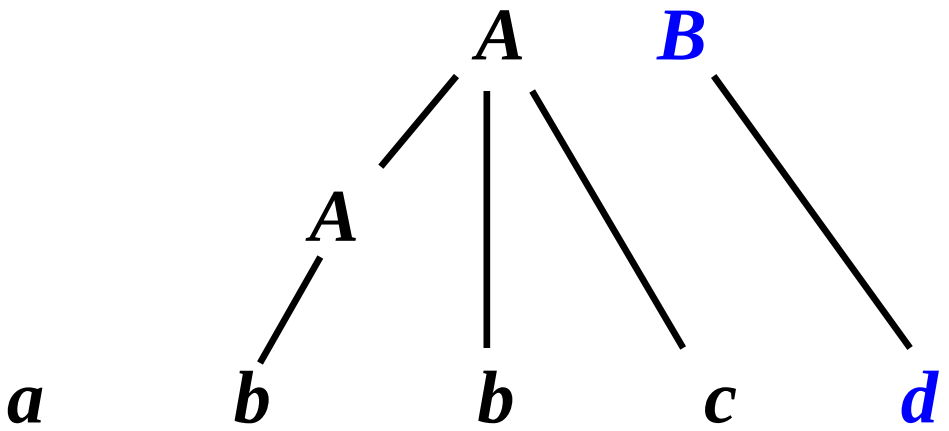
$B \rightarrow d$

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　$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

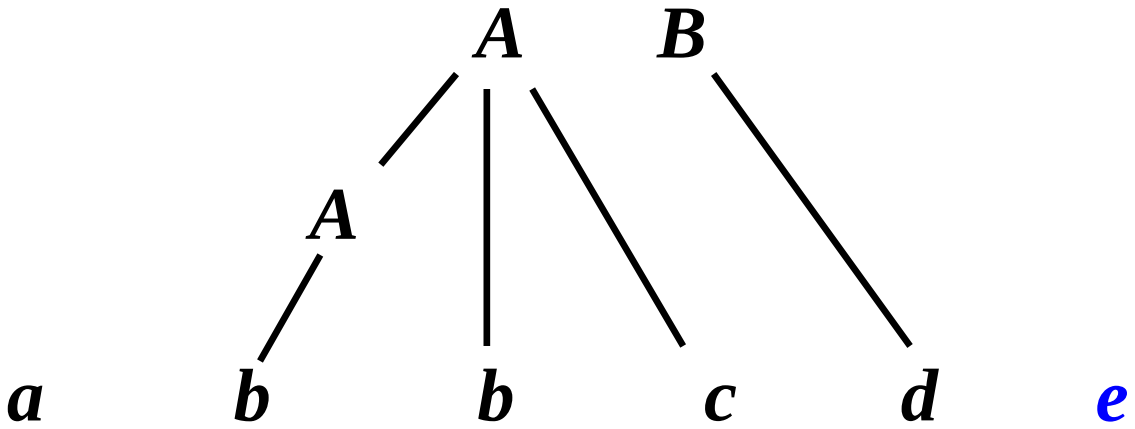
abbcde

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　$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

abbcde

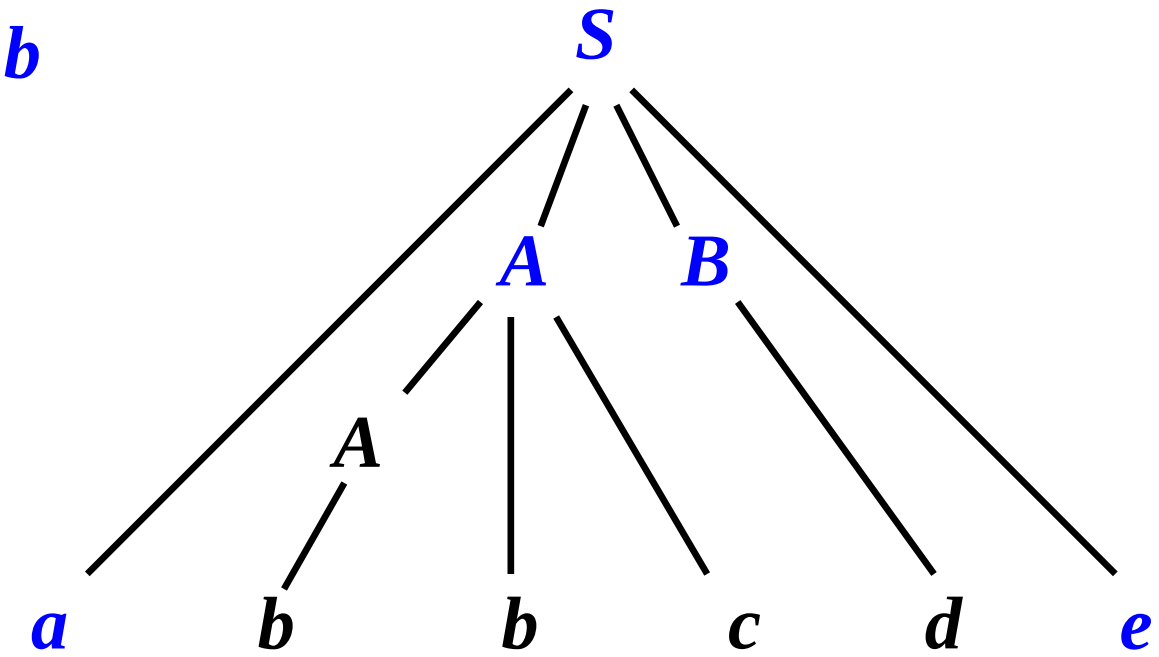
aAbcde

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

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

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

  - 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

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

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

例 句柄不唯一

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

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

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

在右句型$E * E + \mathrm{id}_3$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
$$E$*E$	$+ id_3$$	

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 → id (*parameter_list*) | *expr* = *expr*

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

  parameter → id

  expr → id (*expr_list*) | id　　id(…)是数组元素的引用

  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)…

移进-归约分析的冲突

- 归约–归约冲突(reduce/reduce conflict)

  stmt → id (*parameter_list*) | *expr* = *expr*

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

  parameter → id

  expr → id (*expr_list*) | id　　id(…)是数组元素的引用

  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

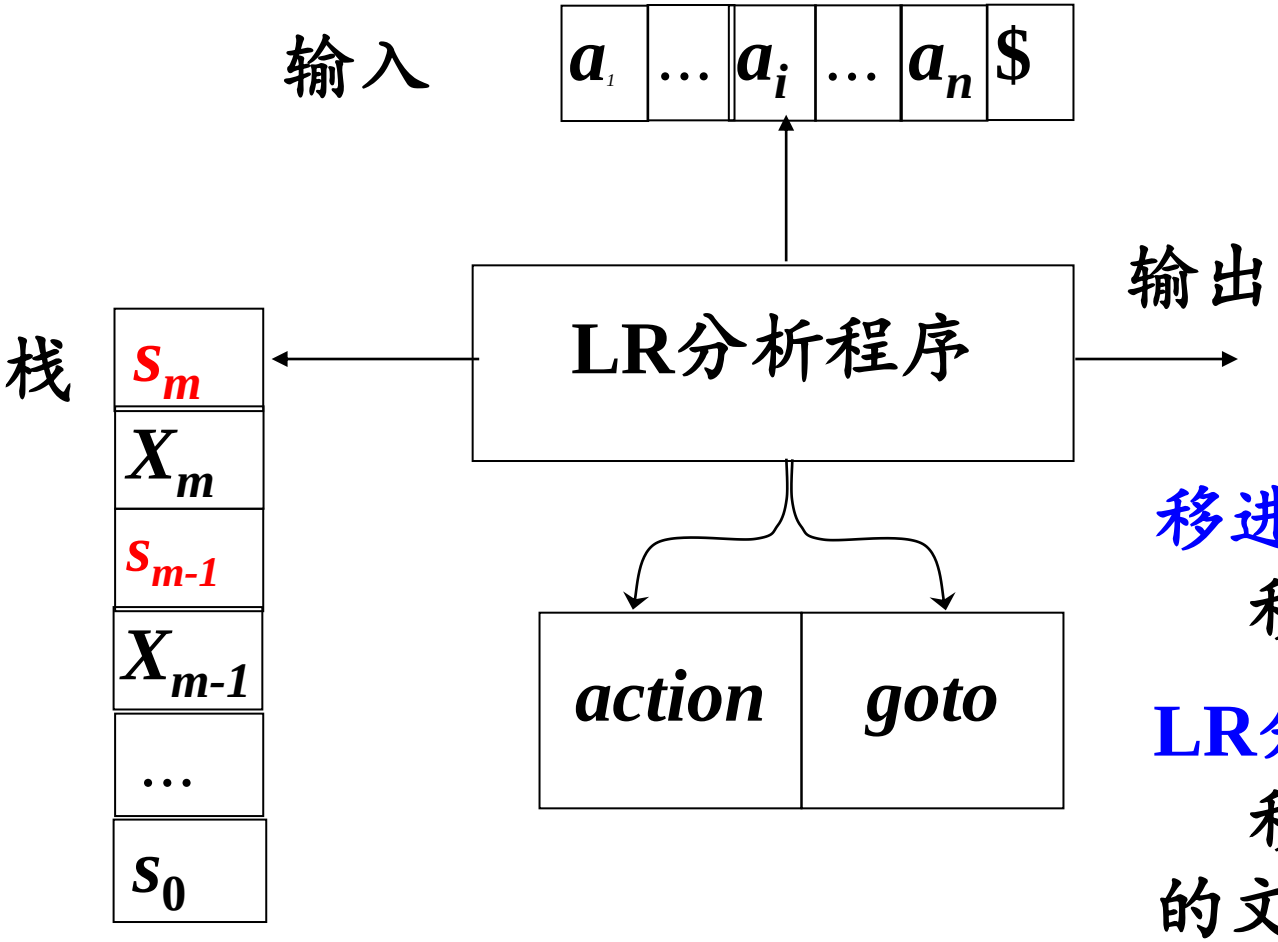
3.5 LR分析器

(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



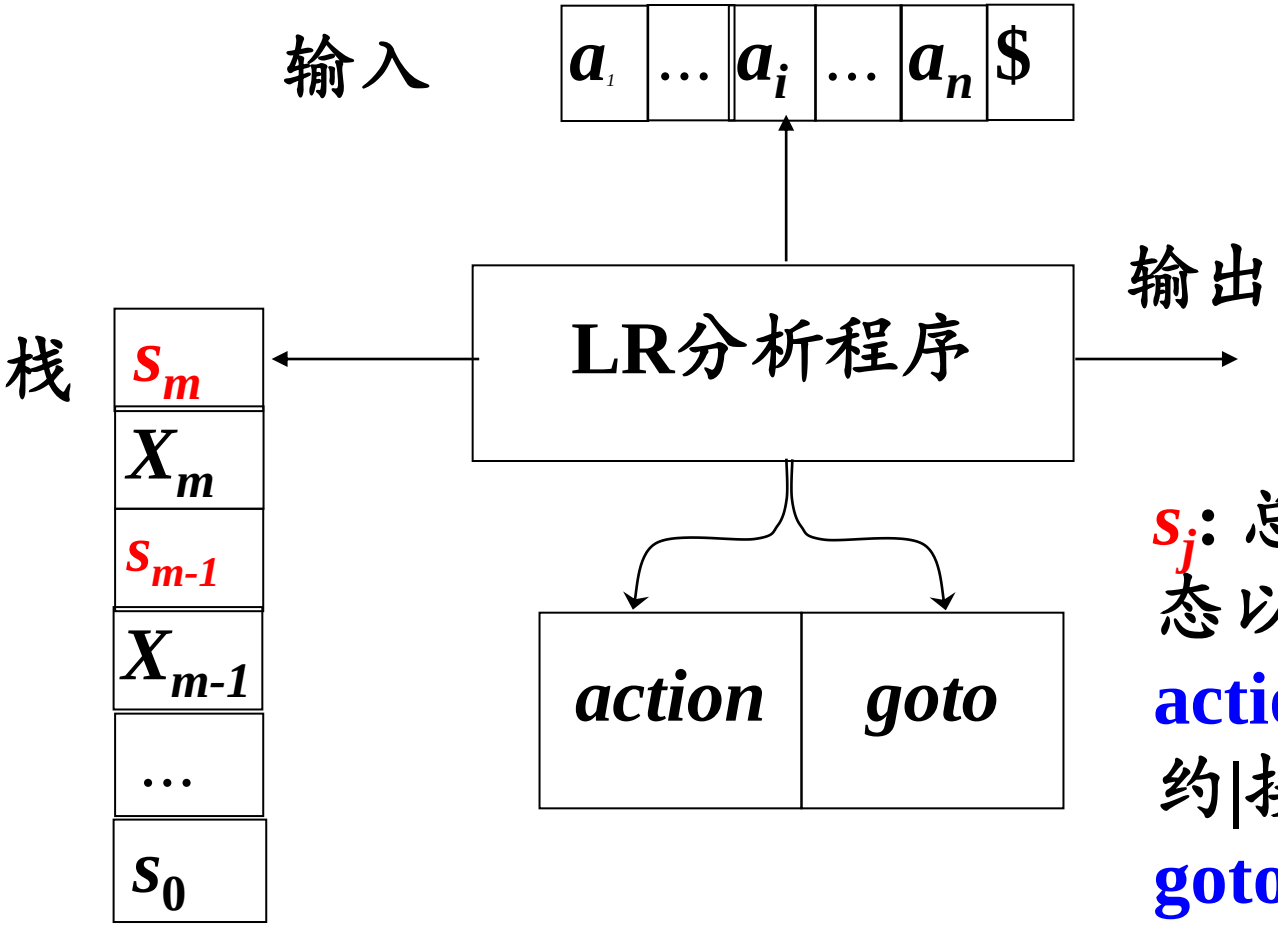
移进-归约分析器

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: 总结了栈中该状态以下的信息

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

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 $	

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

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

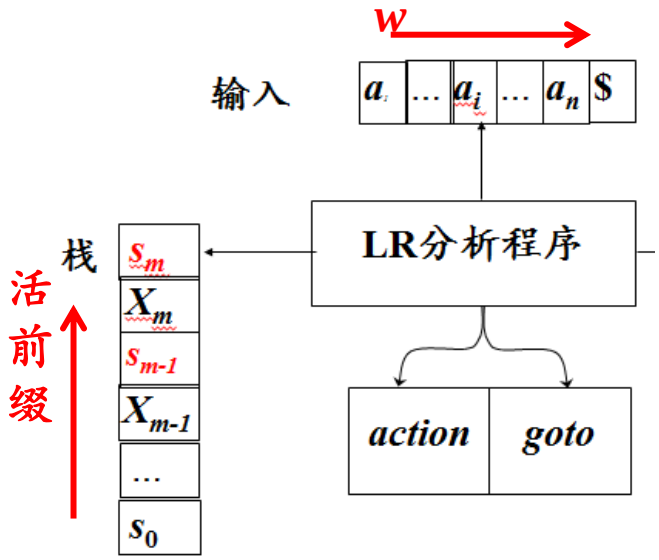
LR分析: 基本概念

- 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

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

$A \rightarrow l\beta$

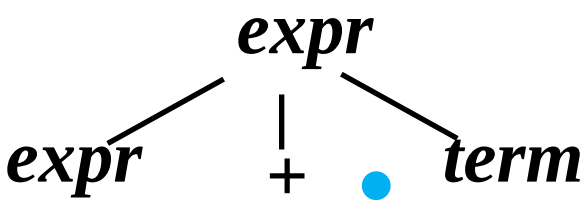
LL(1)决定用该产生式的位置

LR(1)决定用该产生式的位置

SLR分析表的构造

SLR （Simple LR）

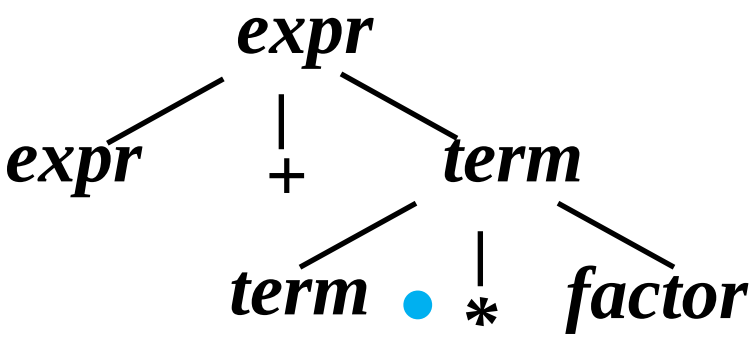
- LR(0) 项目
  - 在右部的某个地方加点的产生式
  - 加点的目的是用来表示分析过程中的状态



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

$A \rightarrow \cdot$

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)

$E' \to E$ 指示分析器何时停止分析并宣布接受输入

$E \to E + T \mid T$

$T \to T * F \mid F$

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

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:

求项目集的闭包closure(I) P75

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

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

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:

$E' \rightarrow \cdot E$

$E \rightarrow \cdot E + T$

$E \rightarrow \cdot T$

$T \rightarrow \cdot T * F$

$T \rightarrow \cdot F$

$F \rightarrow \cdot (E)$

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

$\xrightarrow{E}$

I_1:

$E' \rightarrow E \cdot$

$E \rightarrow E \cdot + T$

$I_1 := goto(I_0, E)$

构造识别活前缀的DFA

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

I_0:

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot \mathbf{id}$

E ⟶ I_1:

$E' \to E \cdot$

$E \to E \cdot + T$

T ⟶ I_2:

$E \to T \cdot$

$T \to T \cdot * F$

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:

$E' \to \cdot E$

$E \to \cdot E + T$

$E \to \cdot T$

$T \to \cdot T * F$

$T \to \cdot F$

$F \to \cdot (E)$

$F \to \cdot \mathbf{id}$

$\xrightarrow{E}$ I_1:

$E' \to E \cdot$

$E \to E \cdot + T$

$\xrightarrow{T}$ I_2:

$E \to T \cdot$

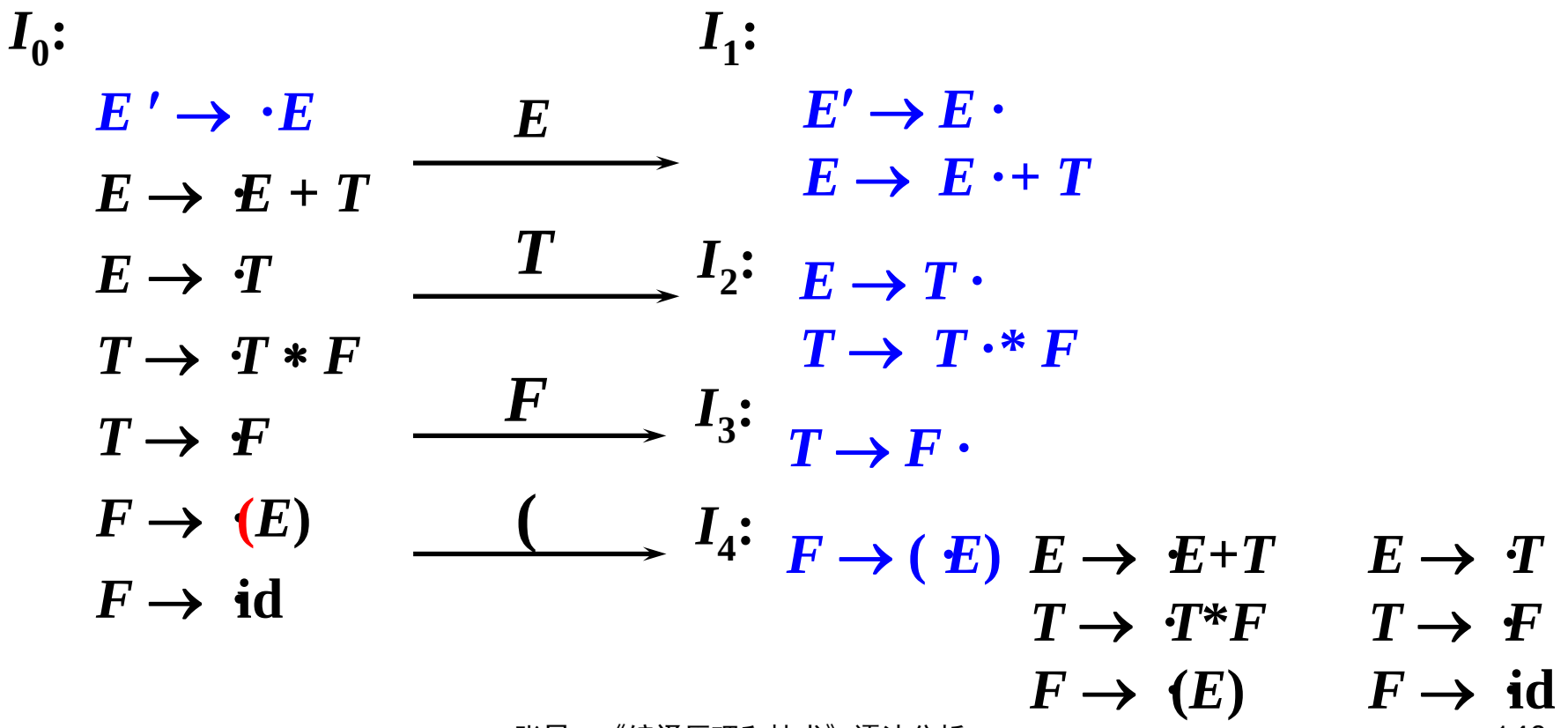
$T \to T \cdot * F$

$\xrightarrow{F}$ I_3:

$T \to F \cdot$

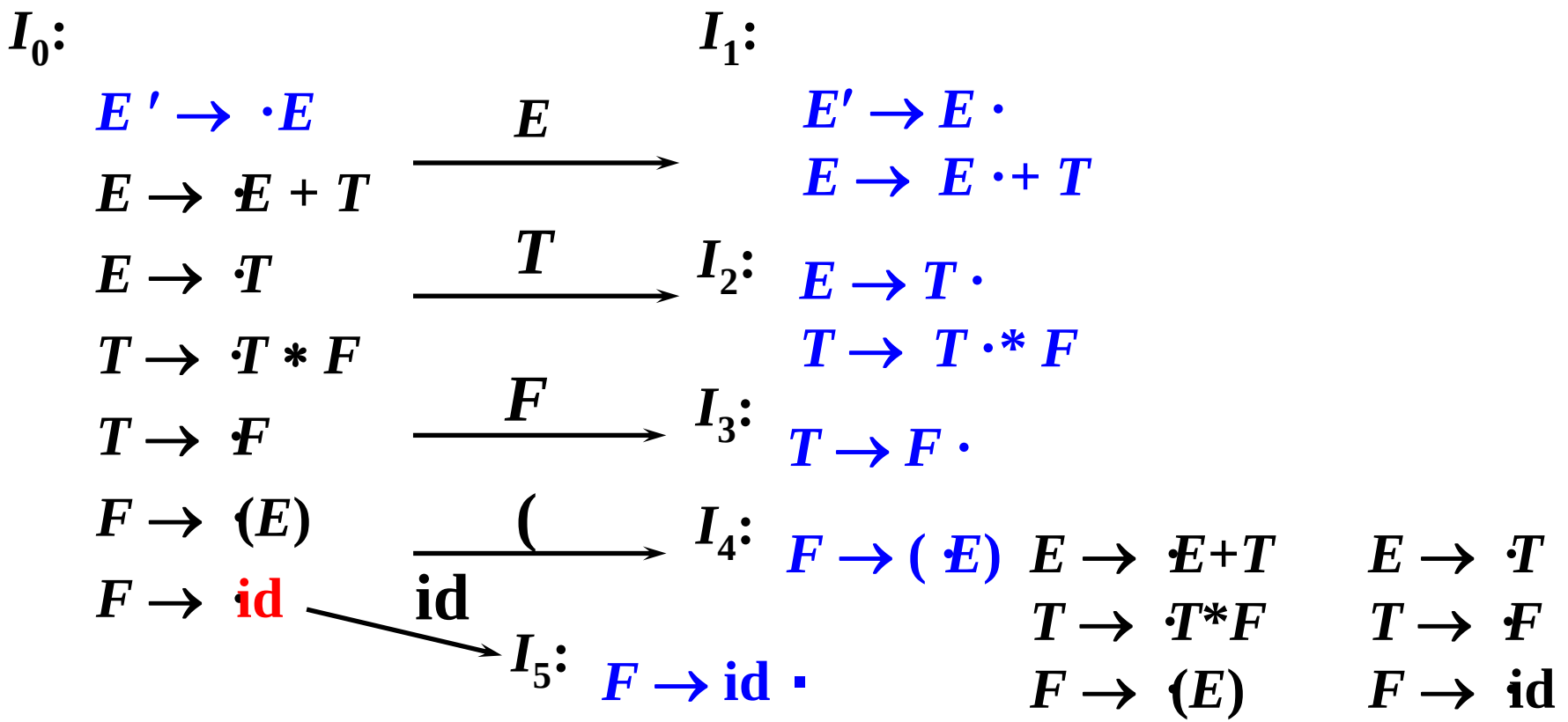
构造识别活前缀的DFA

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

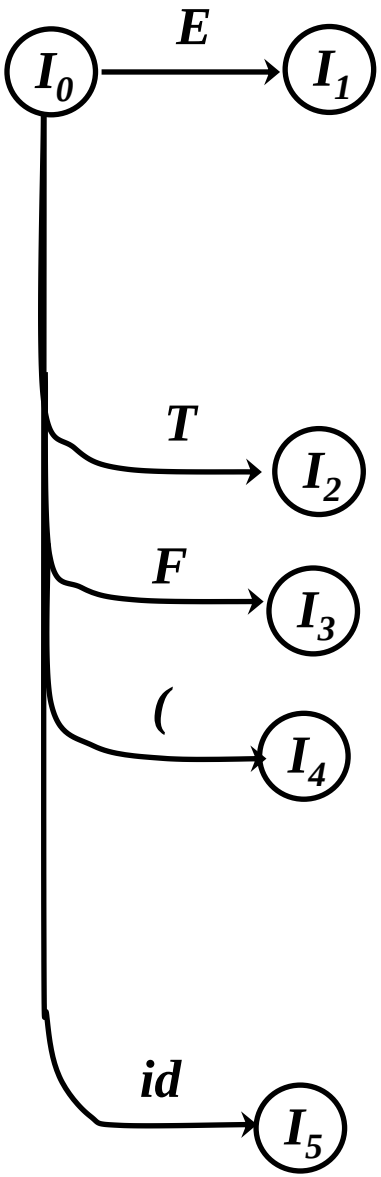


构造识别活前缀的DFA

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



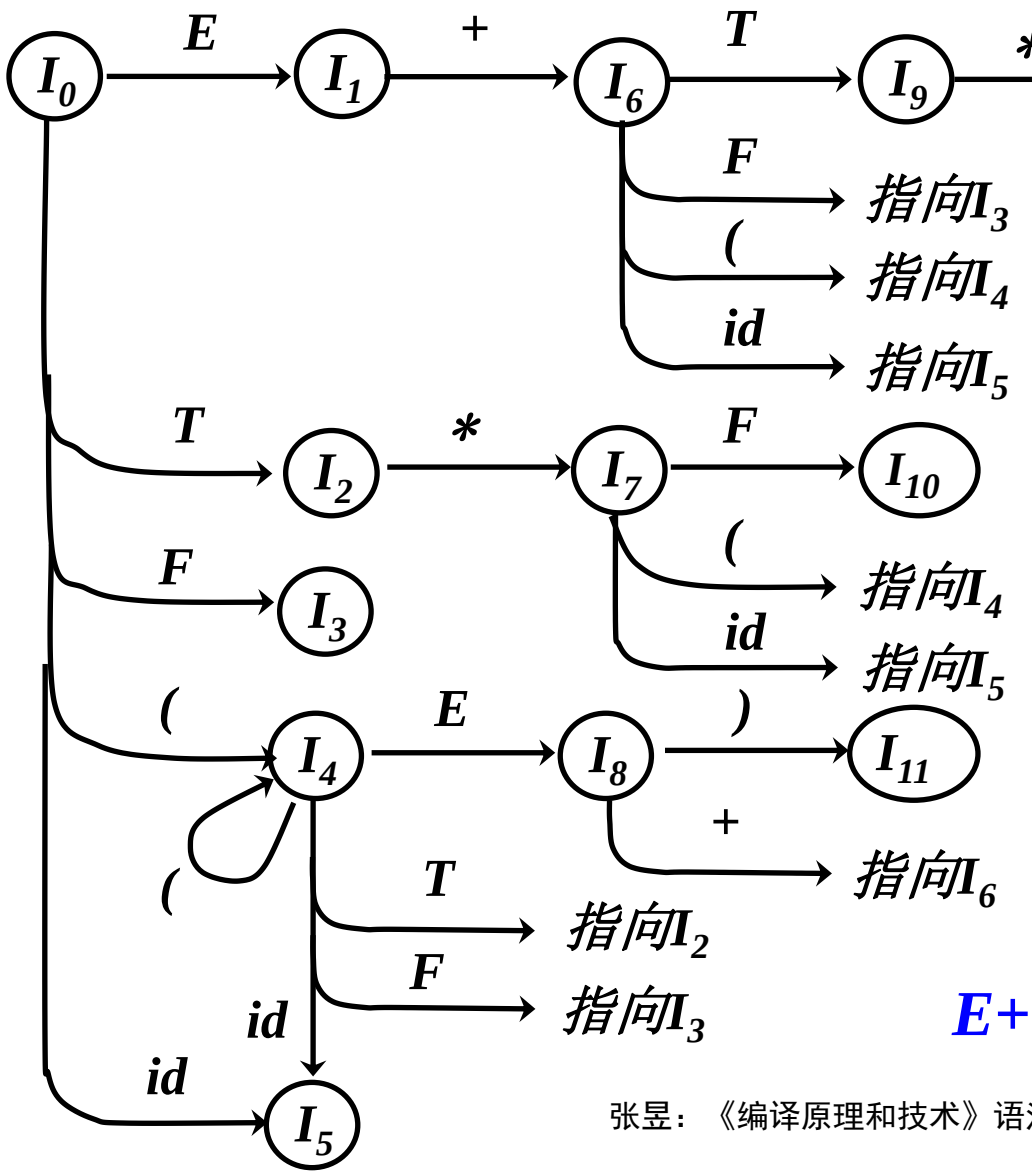
构造识别活前缀的DFA



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

$E' \Rightarrow E$

$\Rightarrow E+T$

$\Rightarrow E+T*F$

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

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

$E+T*F$ 的所有前缀都可接受

有效项目

如果$S' \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta_1\beta_2 w$，那么就说项目$A \rightarrow \beta_1 \cdot \beta_2$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

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

$E' \Rightarrow E \Rightarrow E+T$ （α，β_1都为空）

$E' \Rightarrow E \Rightarrow (E) \Rightarrow (E+T)$ （α="("，β_1为空）

该DFA读过ε 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$A \rightarrow \beta_1 \cdot \beta_2$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$A \rightarrow \beta_1 \cdot \beta_2$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$S' \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta_1\beta_2 w$，那么就说项目$A \rightarrow \beta_1 \cdot \beta_2$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）

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: $T \to T*F$, $F \to (E)$, $F \to id$

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

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

$E' \Rightarrow E$
$\Rightarrow E+T$
$\Rightarrow E+T*F$
$\Rightarrow E+T*(E)$

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

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$[A\rightarrow\alpha \cdot a\beta]$在 I_i中，并且 $goto(I_i, a) = I_j$，那么置$action[i, a]$为sj
  - 归约：如果$[A\rightarrow\alpha \cdot]$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$[S'\rightarrow S \cdot]$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$[A\rightarrow\alpha\ \cdot a\beta]$在 I_i中，并且 $goto(I_i, a) = I_j$，那么置$action[i, a]$为sj
  - 归约：如果$[A\rightarrow\alpha\ \cdot]$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$[S'\rightarrow S\ \cdot]$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$[S'\rightarrow \cdot S]$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

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

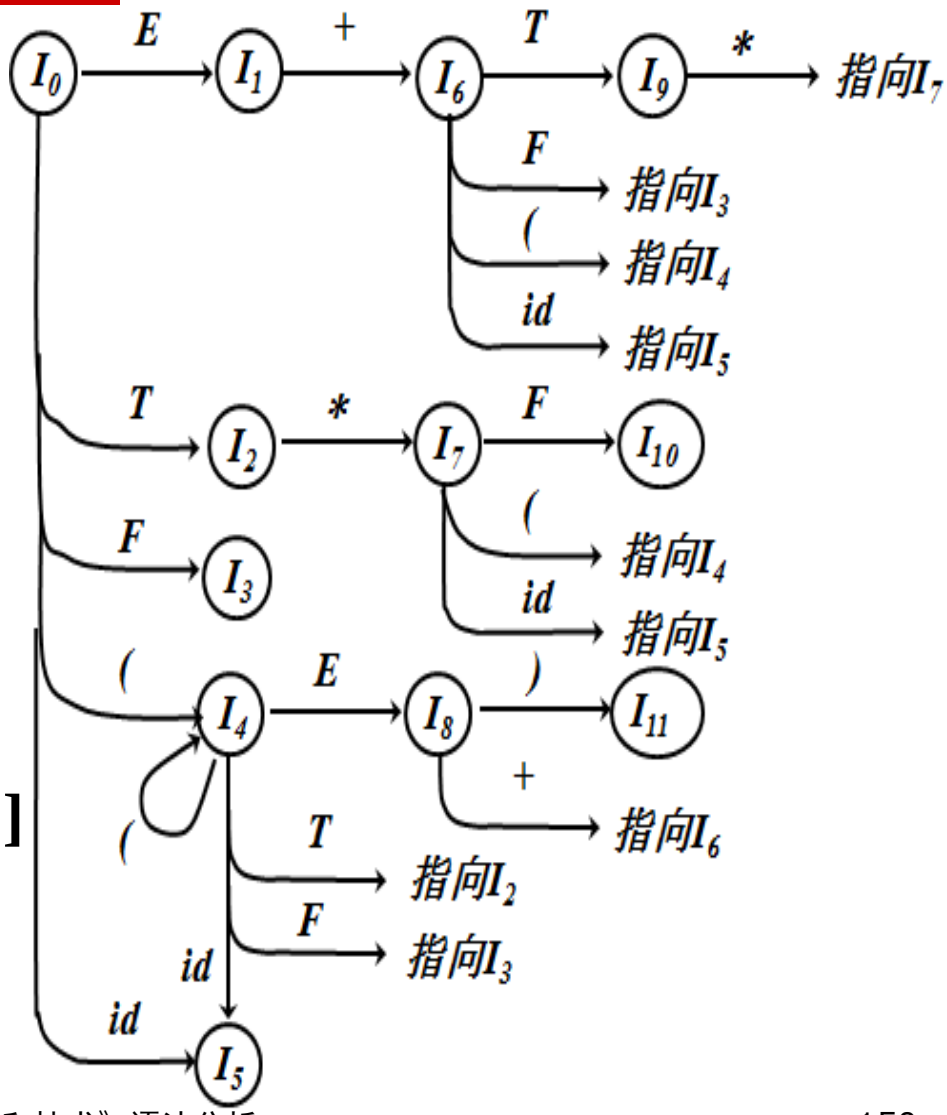
所以

$action[2, \$] = action[2, +]$

$= action[2,)] = r2$

$T \rightarrow T \cdot * F$

$action[2, *] = s7$



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

改进SLR(1)分析技术的办法

- 规范的LR分析表
  - 把LR(0)项目拓展为LR(1)项目

    让LR(0)项目带上搜索符，成为如下形式

    $[A \rightarrow \alpha \cdot \beta, a]$
  - 形式为$[A \rightarrow \alpha \cdot, a]$的项目告知在面临$a$时按产生式$A \rightarrow \alpha$进行归约
- 一个文法的LR(1)项目比它的LR(0)项目多得多，相应的状态转换图也大得多
- LALR分析：是一种折中

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　　*ababbbbaba*

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

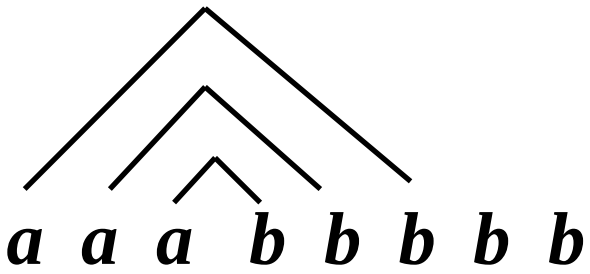
$S \rightarrow aSa \mid bSb \mid c$

是LR的　　*ababbcbbaba*

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。

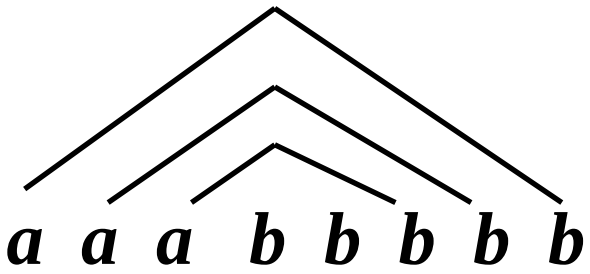
- LR(1)文法：$S \to AB \quad A \to aAb \mid \varepsilon \quad B \to Bb \mid b$



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。

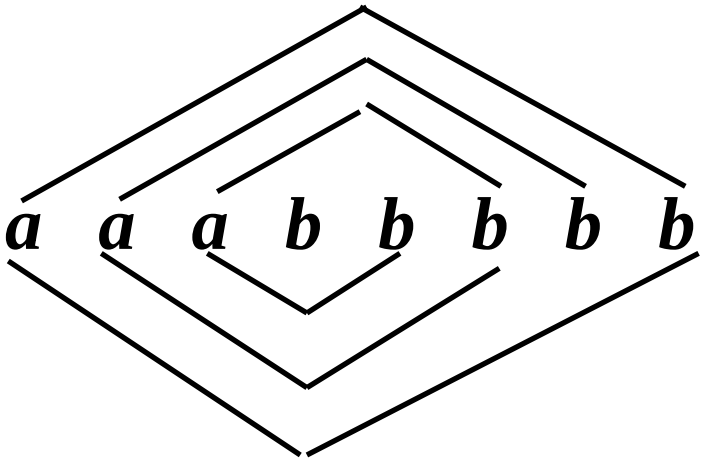
- LR(1)文法：$S \to AB \quad A \to aAb \mid \varepsilon \quad B \to Bb \mid b$
- 非二义且非LR(1)的文法：$S \to aSb \mid B \quad B \to Bb \mid b$



例题 写不同的文法

为语言$L=\{a^mb^n \mid n>m\geq 0\}$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：$S \to aSb \mid B \quad B \to Bb \mid b$
- 二义的文法：$S \to aSb \mid Sb \mid b$

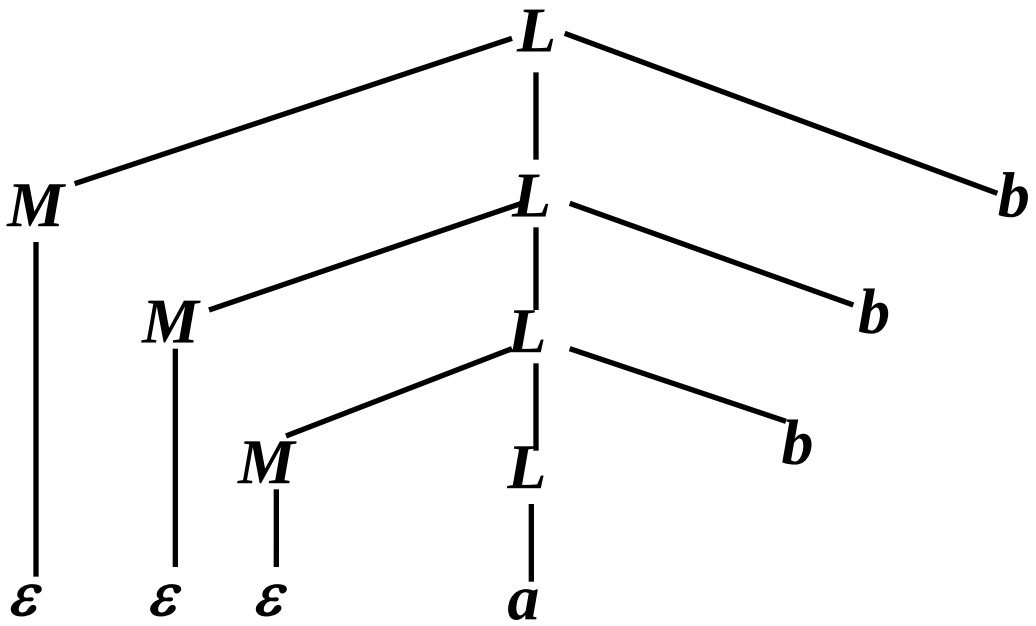


例题4

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？

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

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

$E \to E + E \cdot$

$E \to E \cdot + E$

$E \to E \cdot * E$

id + id + id

id + id * id

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

$E \to E * E \cdot$

$E \to E \cdot + E$

$E \to E \cdot * E$

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_8

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

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

$E \to E \cdot * E$

面临+，归约　面临*，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 ·

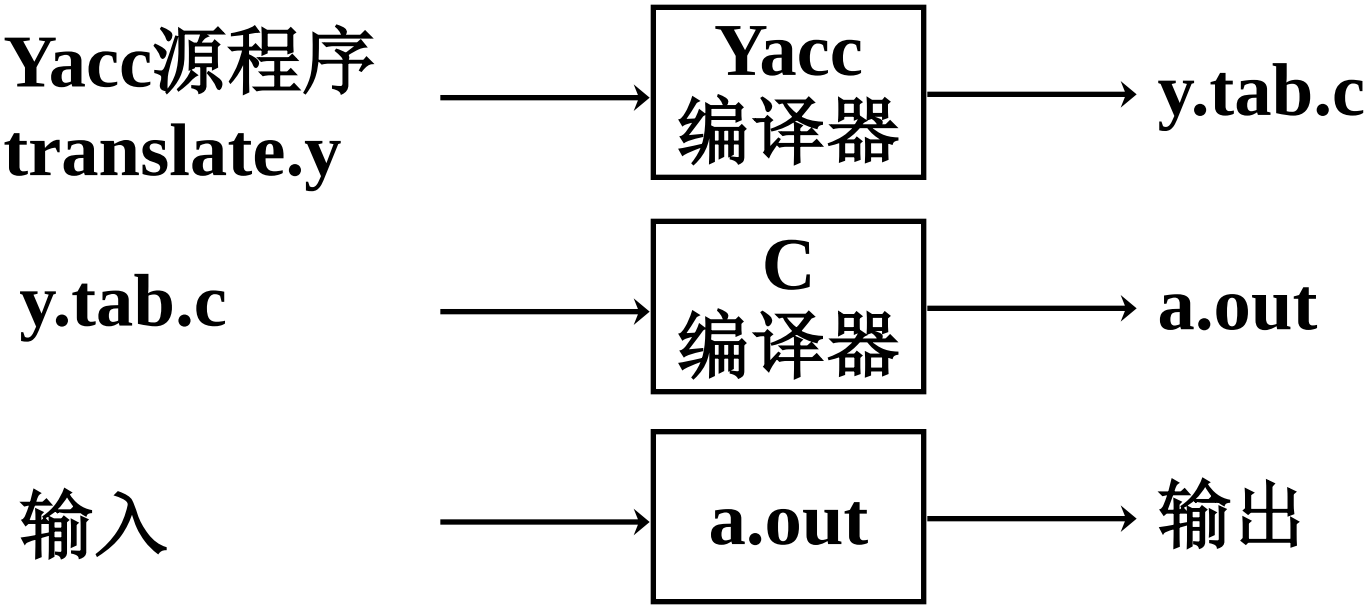
$E \rightarrow E$ sub E ·

…

按前面一个产生式归约

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

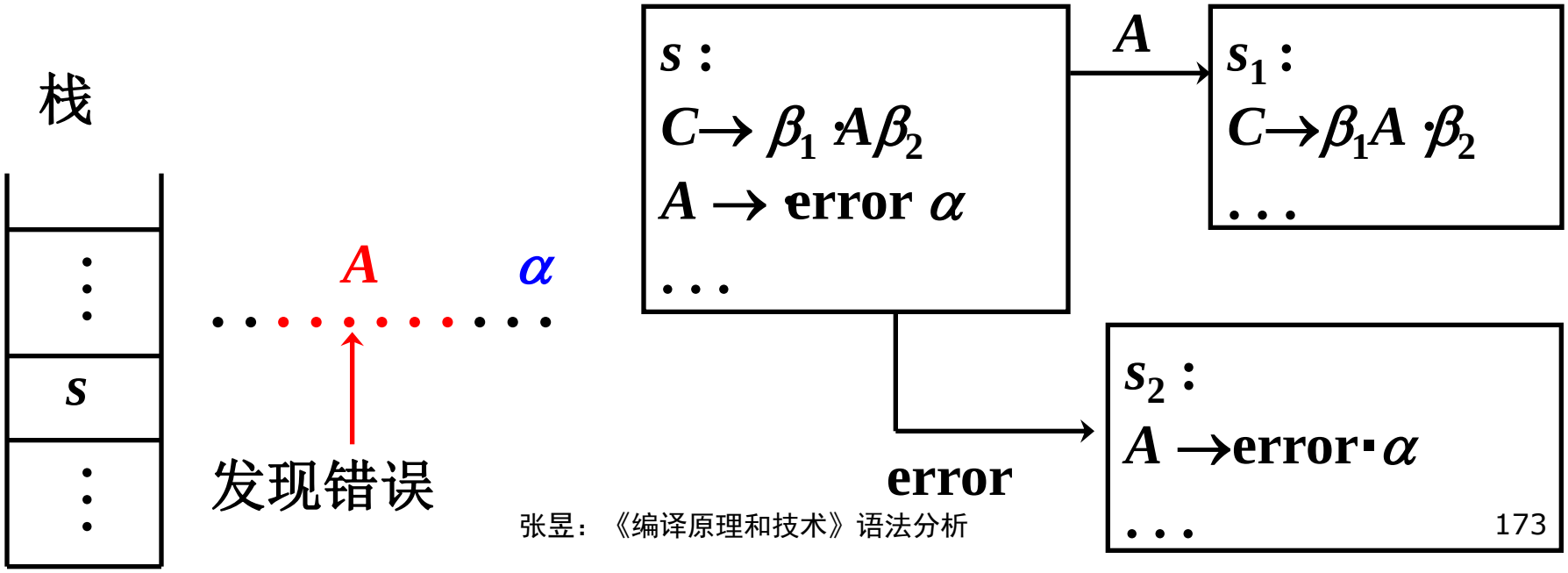
C例程部分

```
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

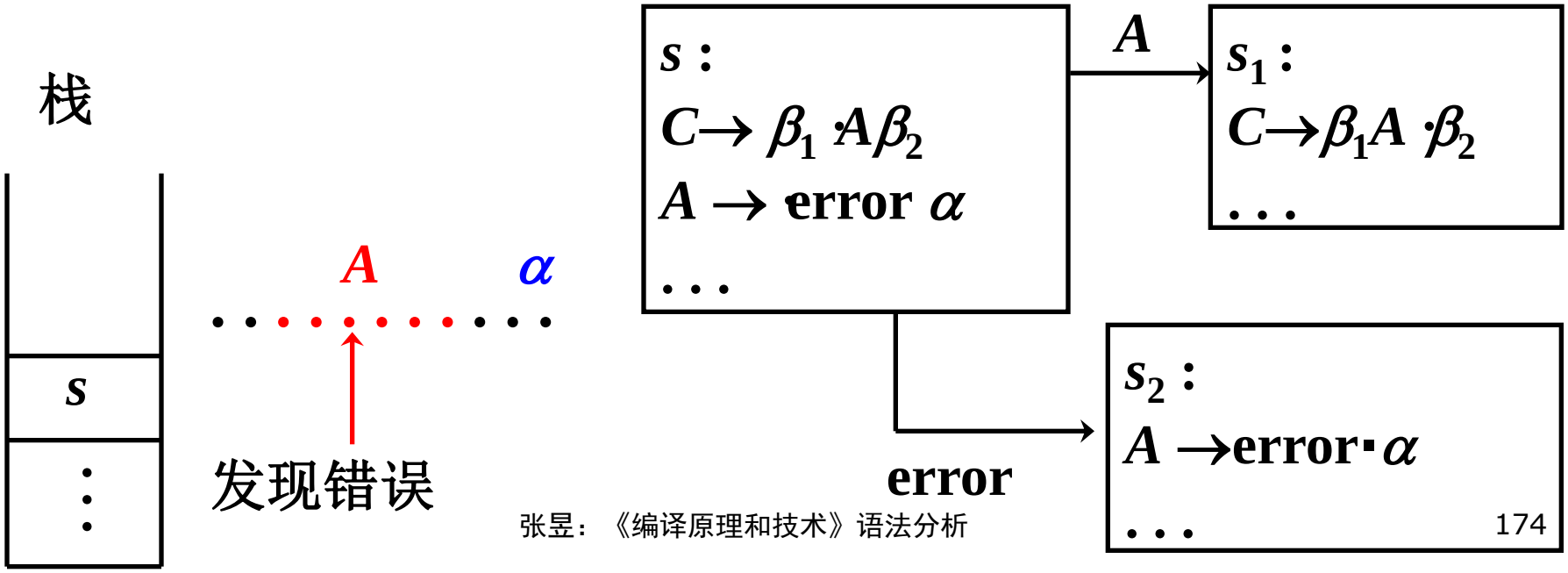
YACC的错误恢复

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



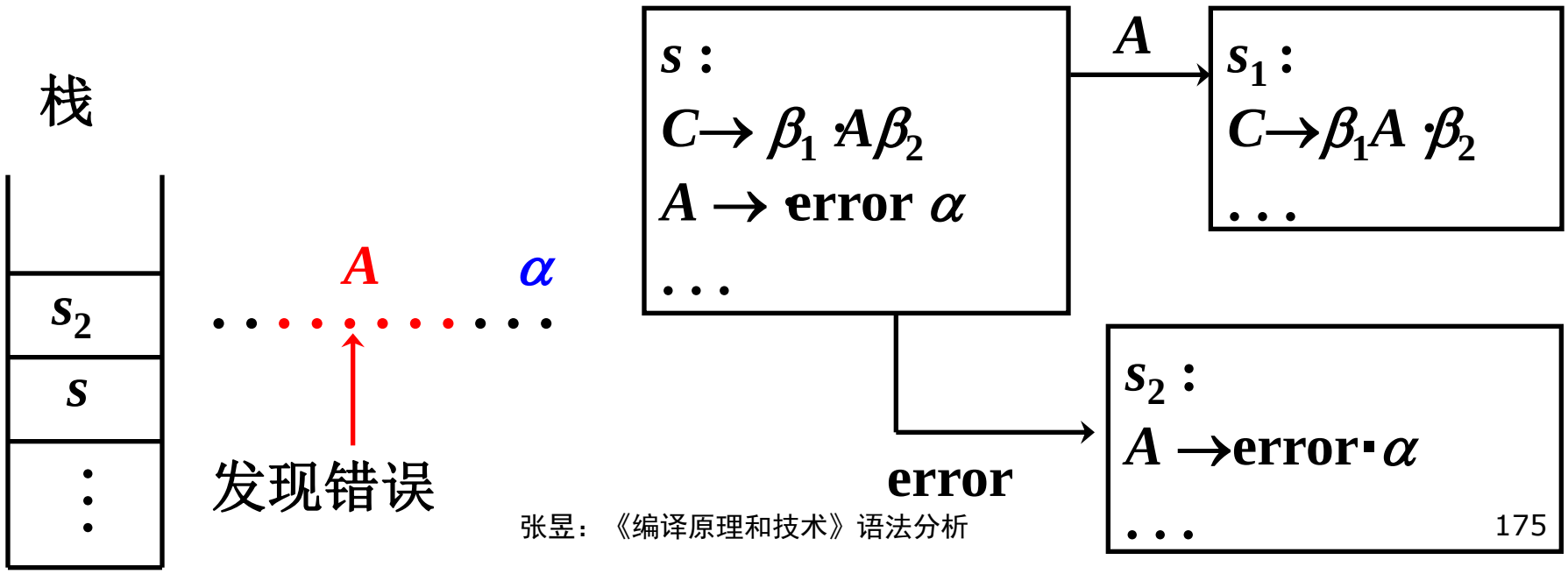
YACC的错误恢复

- 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



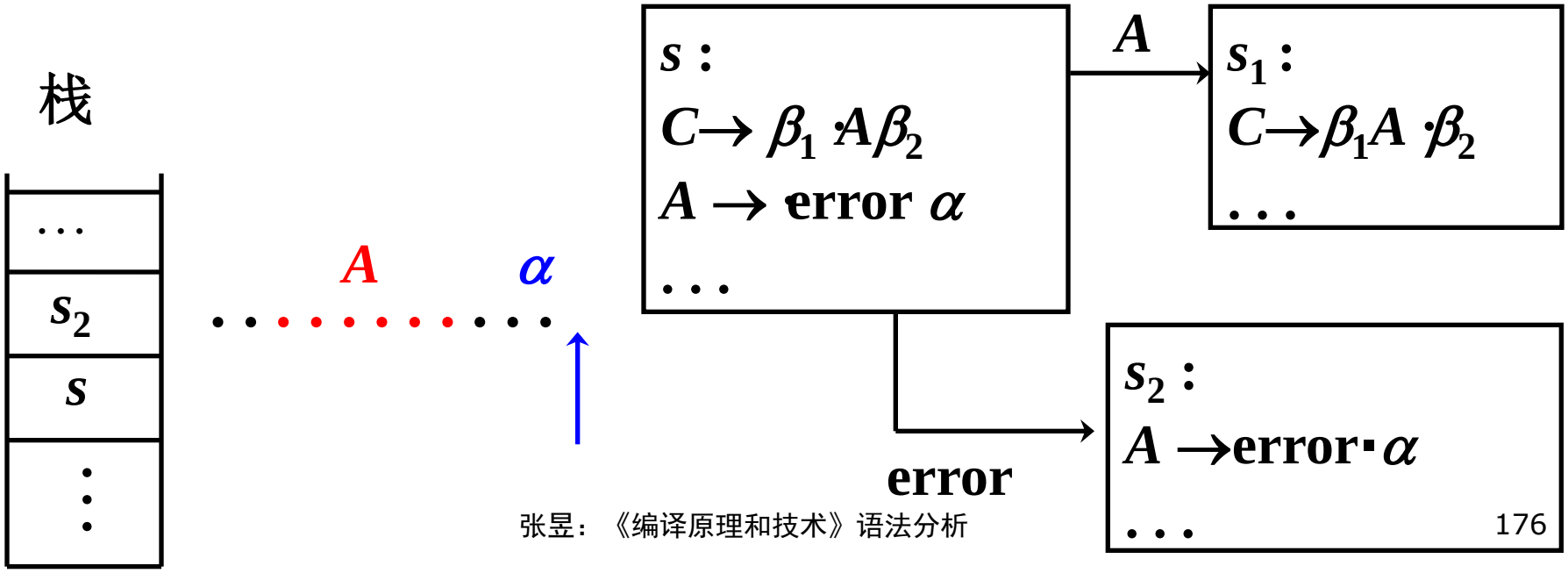
YACC的错误恢复

- 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



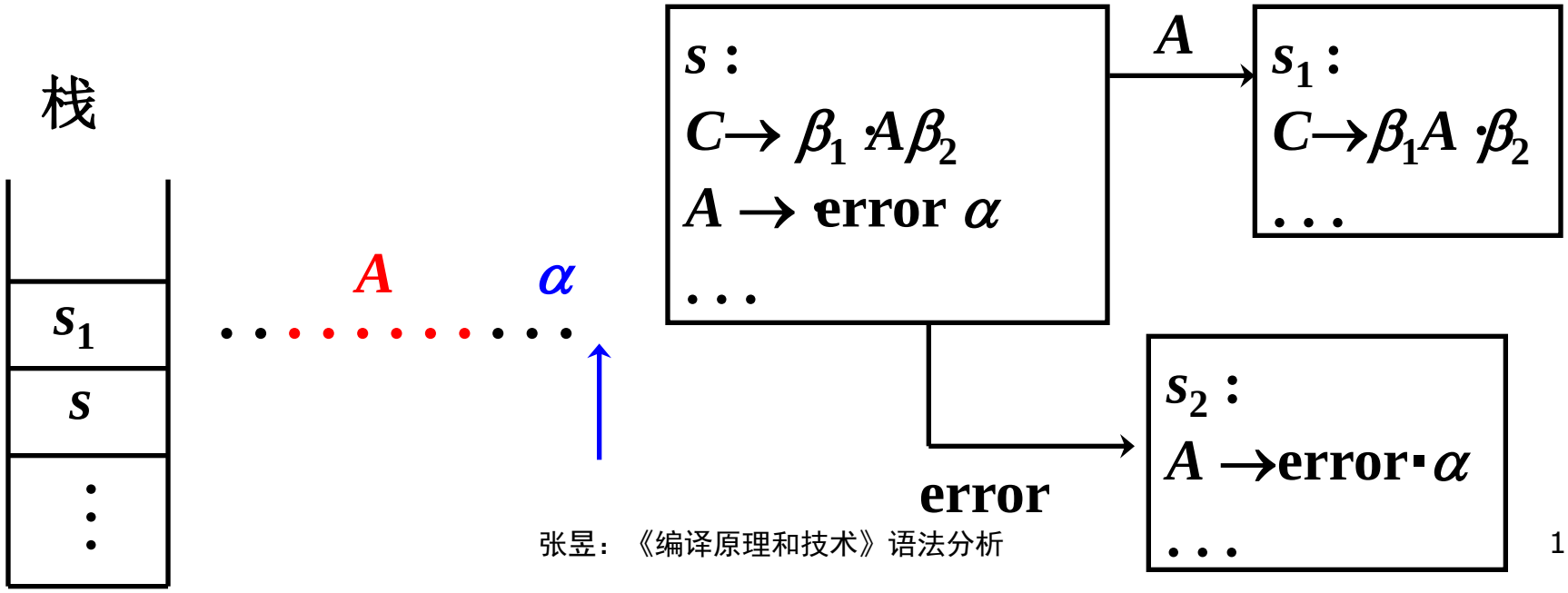
YACC的错误恢复

- 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;}
            ;
```