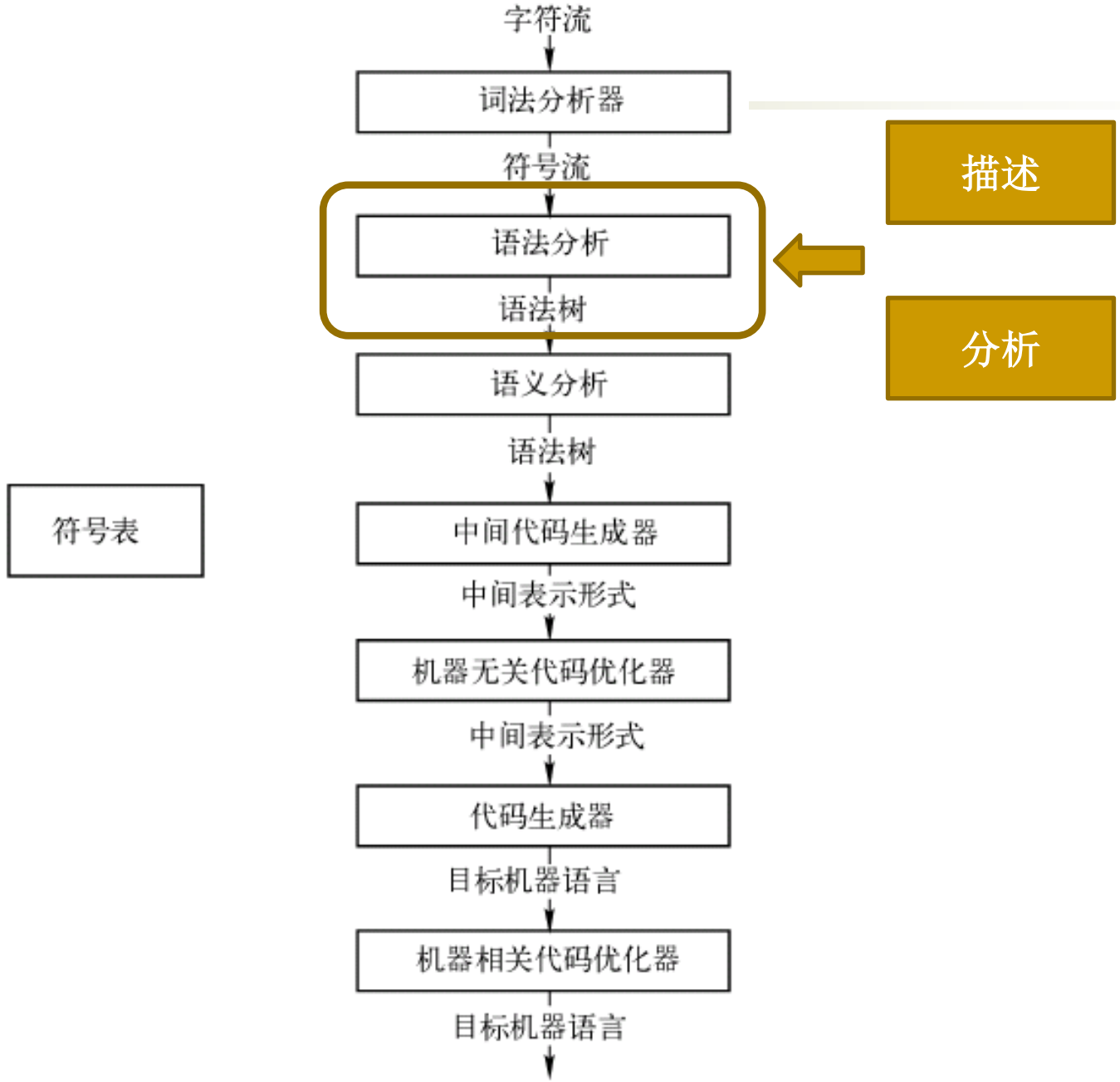
第四章 语法分析

陈 林

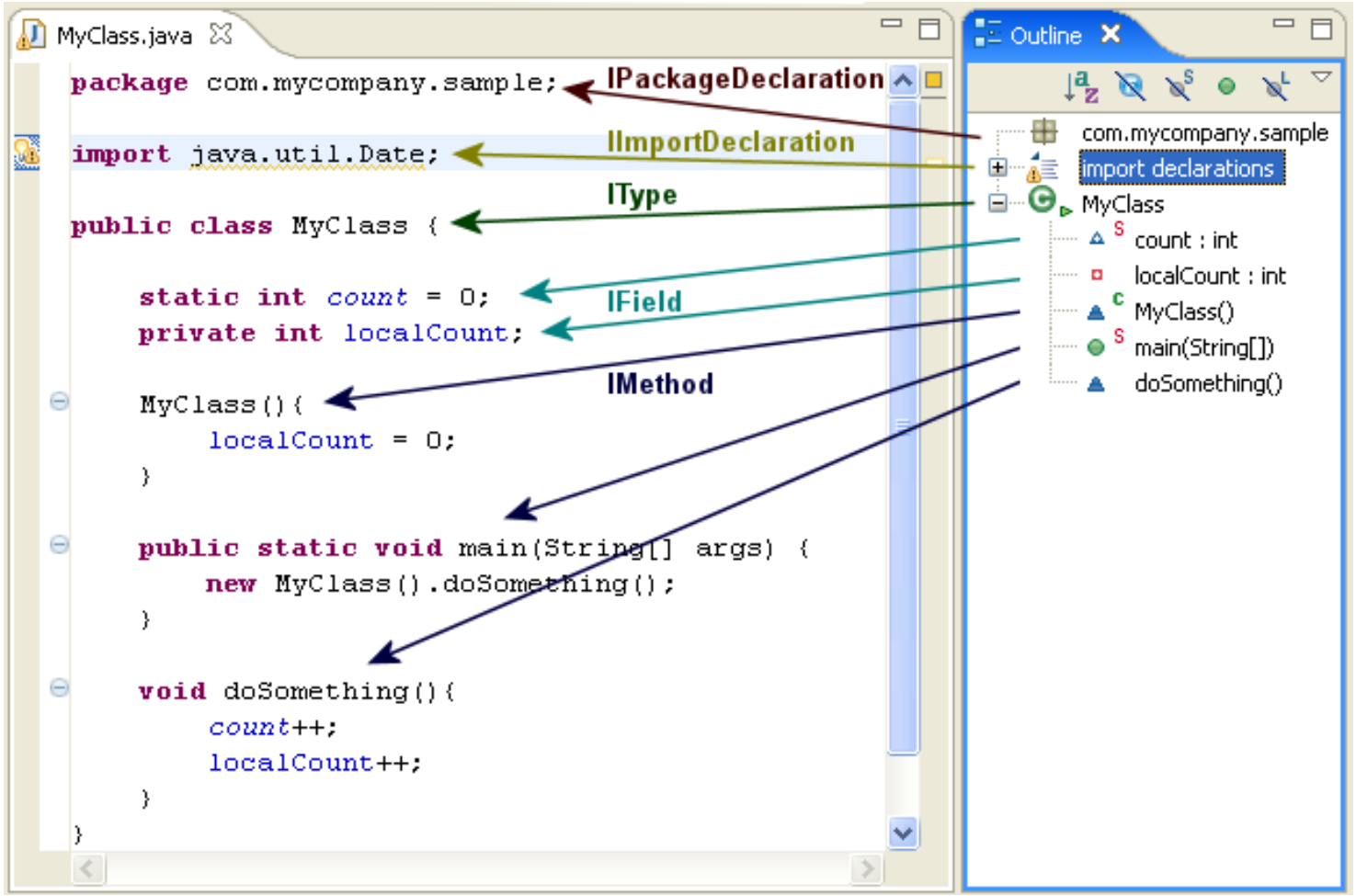
语法分析：提纲

- 引言
- 上下文无关文法
- 语法分析技术
  - 自顶向下
  - 自底向上
- 语法分析器生成工具


字符流
词法分析器
符号流
语法分析
语法树
语义分析
语法树
中间代码生成器
中间表示形式
机器无关代码优化器
中间表示形式
代码生成器
目标机器语言
机器相关代码优化器
目标机器语言
符号表
描述
分析

引言

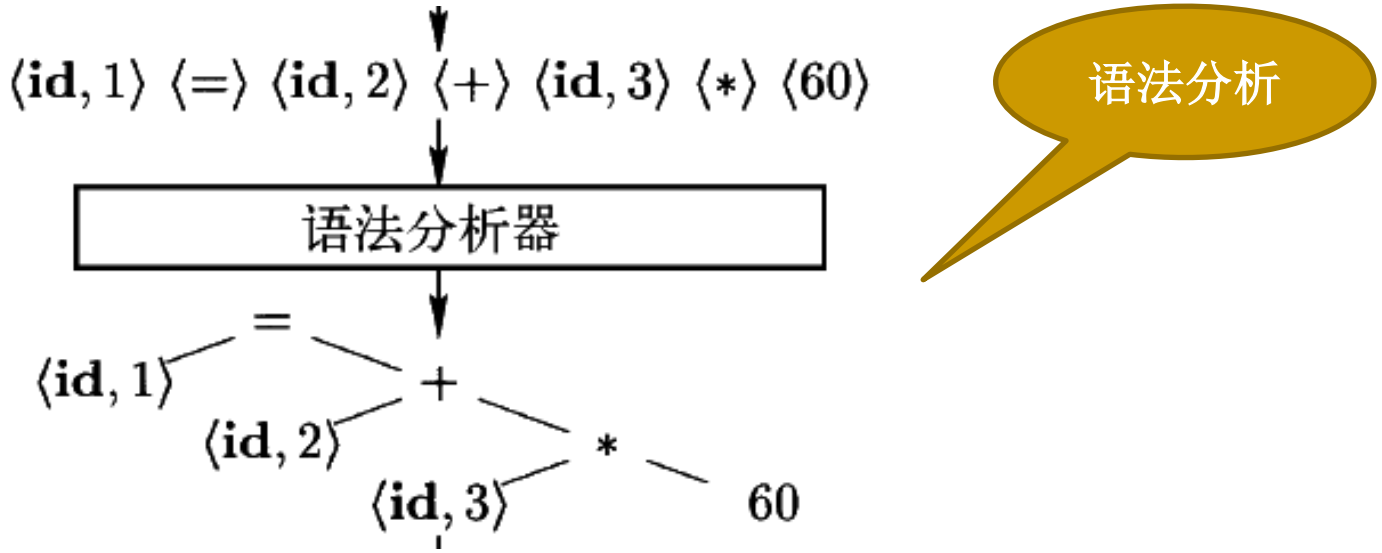
- 程序设计语言源程序的**构成**：语法结构



一个实例

- position = initial + rate * 60

<id,1> <=, > <id, 2> <+, > <id,3> <*, > <number, 4>



引言

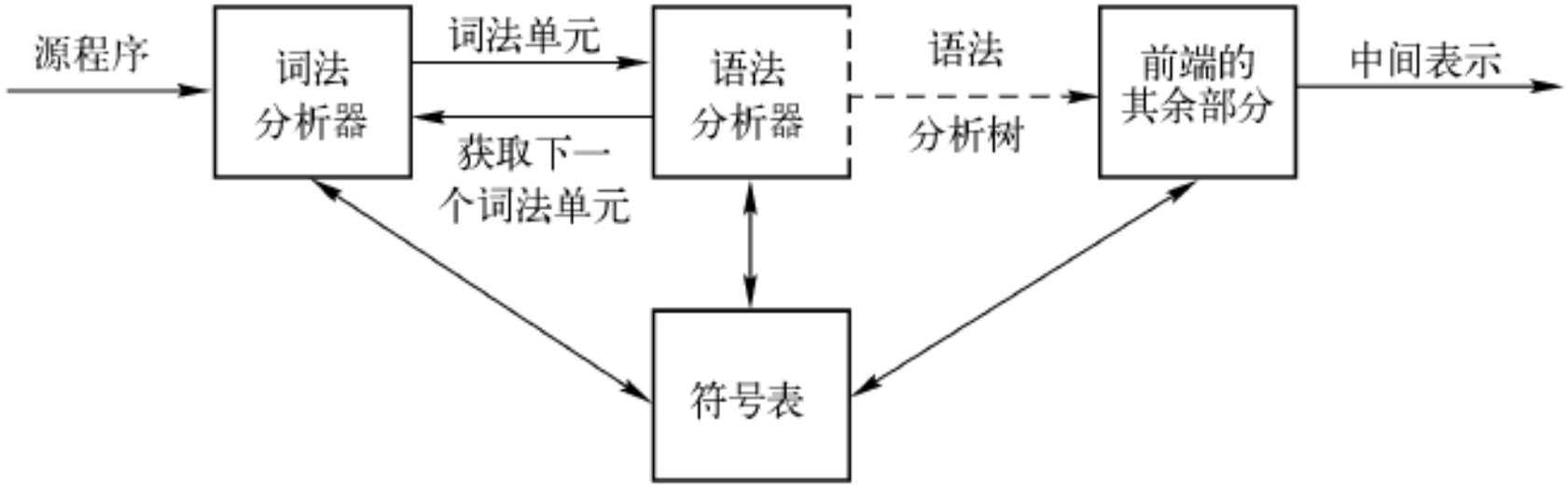
- 文法：一种用于描述程序设计语言语法的表示方法，能够自然地描述程序设计语言构造的层次化语法结构
  - 文法给出了一个程序设计语言的精确易懂的语法规约
  - 可以基于文法构造语法分析器，帮助确定源程序的语法结构
  - 语法结构有助于把源程序翻译为正确的目标代码
  - 文法的扩展性，有助于迭代的语言演化

语法分析器

- 输入：词法分析器输出的词法单元序列
- 输出：语法树表示
- 语法分析器功能：
  - 验证输入源程序的合法性，输出良构程序的语法结构
  - 对于病构的程序，能够报告语法错误，进行错误恢复
- 语法分析器的类型：
  - 通用型
  - 自顶向下：通常处理**LL**文法
  - 自底向上：通常处理**LR**文法

从左到右扫描字符

语法分析器



- 类型检查，语义分析，翻译生成中间代码等往往和语法分析过程交错完成，实践中往往和语法分析放入一个模块，图上用“前端的其余部分”表示上述活动

代表性文法

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

LR文法类
自底向上分析

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & +\ T\ E' \mid \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & *\ F\ T' \mid \epsilon \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

LL类文法
（无左递归）
自顶向下分析

上下文无关文法

- 上下文无关文法 （Context Free Grammar, CFG）

- 上下文无关文法是一种能够很好描述程序设计语言的表示方法

  stmt → if (*expr*) *stmt* else *stmt*

  expr → ……

  stmt → ……

上下文无关文法的定义

- 一个CFG由以下几个部分构成
  - **终结符号**
    - 组成串的基本符号，与“词法单元名字”同义
  - **非终结符号**
    - 语法变量，表示特定串的集合
    - 给出了语言的层次结构，这种层次结构是语法分析和翻译的关键
  - 一个**开始符号**
    - 某个特定的非终结符号，其表示的串集合是这个文法生成的语言
  - 一组**产生式**
    - 描述将终结符号和非终结符号组合成串的方法
    - 产生式左部（头）是一个非终结符号
    - 符号“→”
    - 一个由零个或多个终结符号与非终结符号组成的产生式右部（体）

用于描述算术表达式的文法定义

expression	→	*expression* + *term*
expression	→	*expression* − *term*
expression	→	*term*
term	→	*term* * *factor*
term	→	*term* / *factor*
term	→	*factor*
factor	→	(*expression*)
factor	→	**id**

符号表示约定

- 终结符号：a b + 3 **id**…
- 非终结符号: *A B C…, S, stmt*
- 文法符号: *X Y* …
- 终结符号串:*u v w*…
- 文法符号串:*α β*…
- 不同可选体: α_1 α_2 α_3…
- 开始符号: *S*

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

推导

- 产生式又可称为重写规则（Rewriting rule）
- 推导
  - 将待处理的串中的某个非终结符号替换为这个非终结符号的某个产生式的体
  - 从开始符号出发，不断进行上面的替换，就可以得到文法的不同句型
- 推导的实例
  - 文法：E → -E | E+E | E*E | (E) | id
  - 从E到-(id)的推导序列：E => -E => -(E) => -(id)

推导

- 推导的一般性定义
  - 如果$A \to \gamma$是一个产生式，那么$\alpha A \beta \Rightarrow \alpha \gamma \beta$
- 经过零步或者多步推导出：$\overset{*}{\Rightarrow}$
  - 对于任何串α，$\alpha \overset{*}{\Rightarrow} \alpha$
  - 如果$\alpha \overset{*}{\Rightarrow} \beta$且$\beta \Rightarrow \gamma$，那么$\alpha \overset{*}{\Rightarrow} \gamma$
- 经过一步或者多步推导出：$\overset{+}{\Rightarrow}$
  - $\alpha \overset{*}{\Rightarrow} \beta$且$\alpha$不等于$\beta$等价于$\alpha \overset{+}{\Rightarrow} \beta$
- 最左(右)推导
  - 符号：$\overset{*}{\underset{lm}{\Rightarrow}}$ $\overset{*}{\underset{rm}{\Rightarrow}}$

推导的例子

- 文法实例

E → E+E | E*E | -E | (E) | **id**

从上述文法推导出串 –(id+id)

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

句型/句子/语言

- 句型（sentential form）：
  - 如果$S \overset{*}{\Rightarrow} \alpha$，那么α就是文法的一个句型
  - 可能既包含非终结符，又包含终结符号；可以是空串
- 句子（sentence）
  - 文法的句子就是不包含非终结符号的句型
- 语言
  - 文法G的语言就是G的句子的集合，记为L(G)
  - ω在L(G)中当且仅当ω是G的句子，即$S \overset{*}{\Rightarrow} \omega$

推导的问题

- 从推导的角度看，**语法分析的任务**是：接受一个终结符号串作为输入，找出从文法的开始符号推导出这个串的方法

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

推导出串 –(id+id*id)

推导的问题

- 推导中可能遇到的两个问题
  - 每一步替换哪个非终结符号？
  - 若以这个非终结符号为头的产生式有多个，用哪个产生式的右部替换？

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

推导出串 –(id+id*id)

非终结符号的替换顺序

- 通常使用两种方式进行推导
  - 最左推导：总是选择每个句型的最左非终结符号。记作 $\underset{lm}{\Rightarrow}$
  - 最右推导：总是选择最右边的非终结符号。记作 $\underset{rm}{\Rightarrow}$
- 每个最左推导步骤可以写成 $wA\gamma \underset{lm}{\Rightarrow} w\delta\gamma$，

  应用产生式： $A \rightarrow \delta$
- 如果α经过最左推导得到β，记作 $\alpha \overset{*}{\underset{lm}{\Rightarrow}} \beta$
- 最左句型：S是文法G的识别符号，如果 $S \overset{*}{\underset{lm}{\Rightarrow}} \alpha$ ，则α是G的最左句型

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

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

语法分析树

- 推导的图形表示形式
  - 根结点的标号是文法的开始符号
  - 每个叶子结点的标号是非终结符号、终结符号或 ε
  - 每个内部节点的标号是非终结符号
  - 每个内部结点表示某个产生式的一次应用
    - 内部结点的标号为产生式头，结点的子结点从左到右是产生式的体
- 有时允许树的根不是开始符号（对应于某个短语）
- 树的叶子组成的序列是根的文法符号的句型
- 一棵分析树可对应多个推导序列，但是分析树和最左（右）推导序列之间具有一一对应关系

语法分析树

- 推导的图形表示形式，树上看不出来推导的顺序
- 能够反映串的语法层次结构

- 语法分析树
  - 内部节点：对应于一个非终结符号
  - 子节点：对应于其父节点为头的产生式体
  - 叶子节点：可以是终结符号或非终结符号，从左到右排列可以得到一个句型，称为这棵树的结果

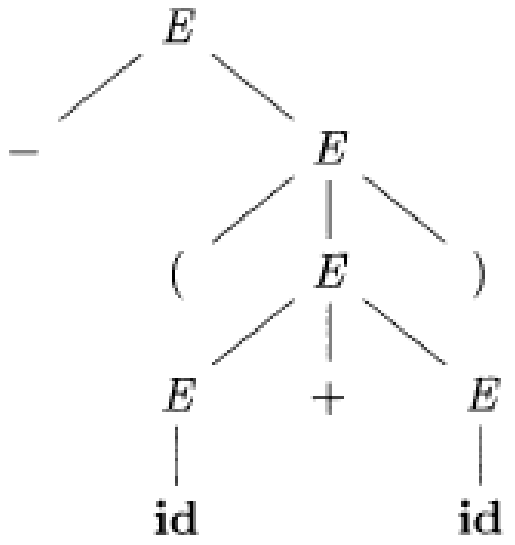


图 4-3　−(id + id)的语法分析树

E→E+E | E*E | -E | (E) | id

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

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

推导和语法树

- 对于推导中的每个句型 α_i，我们都可以构造出一个结果为 α_i 的语法树
  - 构造过程是对i的一次归纳过程
  - 假设已经构造出 $\alpha_{i-1} = X_1 X_2 \cdots X_k$

    $X_j \rightarrow \beta$

    $\beta = Y_1 Y_2 \cdots Y_m$

    $\alpha_i = X_1 X_2 \cdots X_{j-1} \beta X_{j+1} \cdots X_k$
  - 在推导的第i步，对α_i 应用规则 $X_j \rightarrow \beta$

从推导序列构造分析树

- 假设有推导序列：
  - $A \Rightarrow \alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_n$
- 算法：
  - 初始化：α_1的分析树是标号为A的单个结点
  - 假设已经构造出$\alpha_{i-1}=X_1X_2\cdots X_k$的分析树，且$\alpha_{i-1}$到$\alpha_i$的推导是将$X_j$替换为β，那么在当前分析树中找出第j个非ε结点，向这个结点增加构成β的子结点。（如果β=ε，则增加一个标号为ε的子结点）

推导与语法树的例子

E→E+E | E*E | -E | (E) | **id**

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

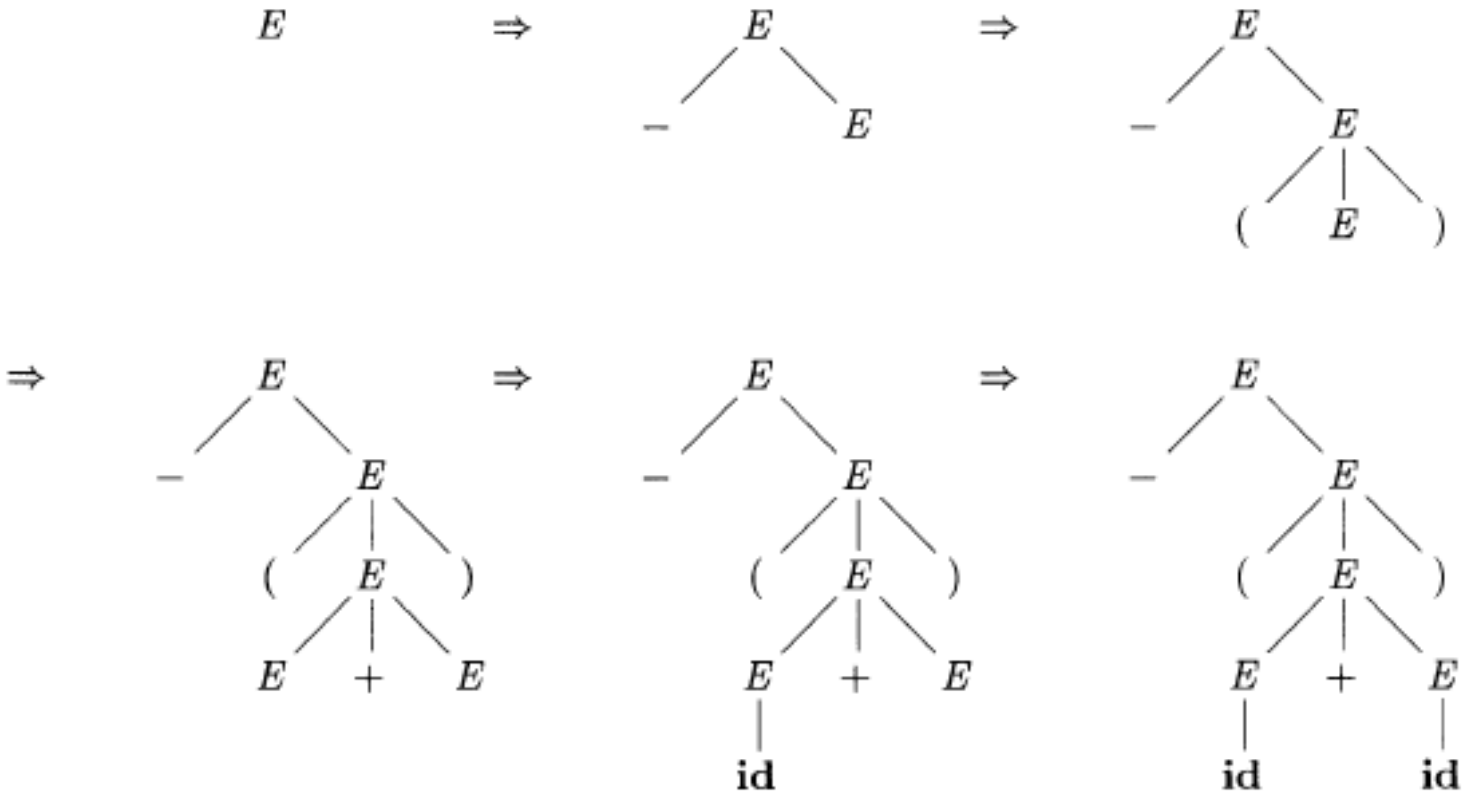


图 4-4　推导(4.8)的语法分析树序列

二义性文法

E→E+E | E*E | -E | (E) | **id**

- 如果一个文法可以为一个句子生成多棵不同的语法分析树，则该文法为二义性文法

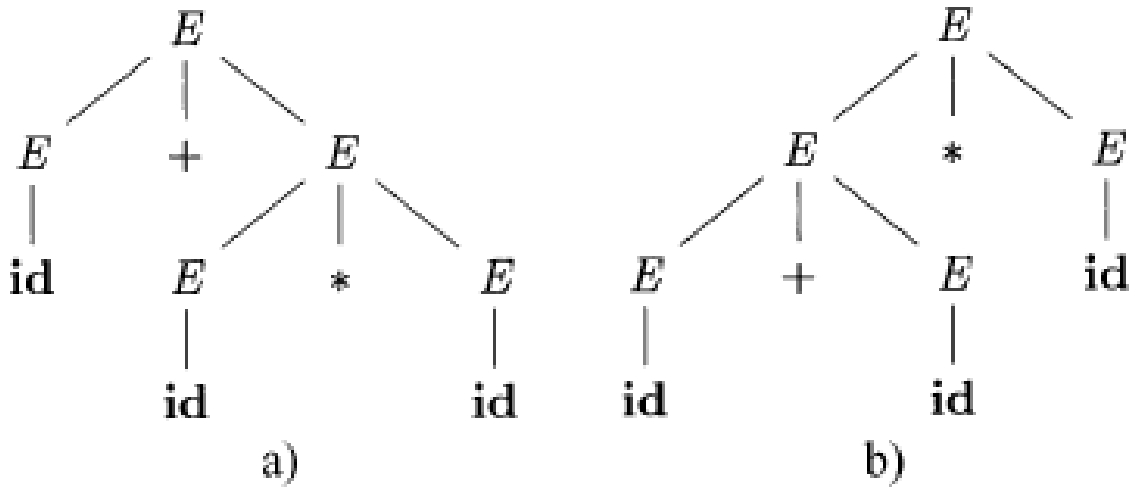


图 4-5　**id** + **id** * **id** 的两棵语法树

- 通常情况下，我们需要无二义性的文法

二义性

- 程序设计语言的文法通常都应该是无二义性的
  - 否则就会导致一个程序有多种“正确”的解释。
  - 比如文法： E → -E | E+E | E*E | (E) | id 的句子 a+b*c
- 但有些二义性的情况可以方便文法或语法分析器的设计
  - 需要消二义性规则来剔除不要的语法分析树
  - 比如：先乘除后加减

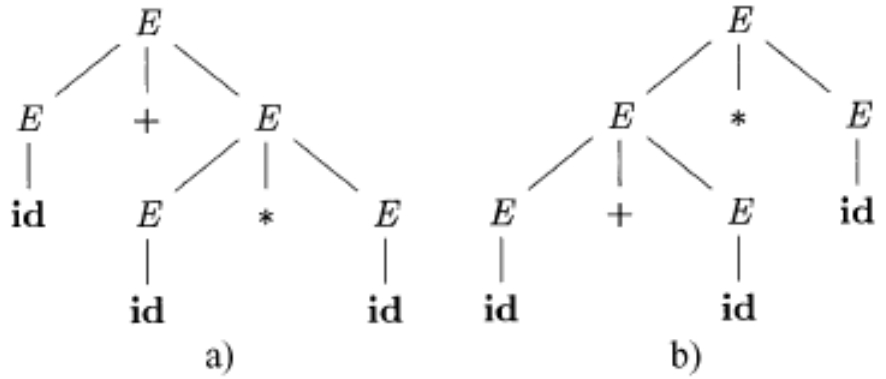


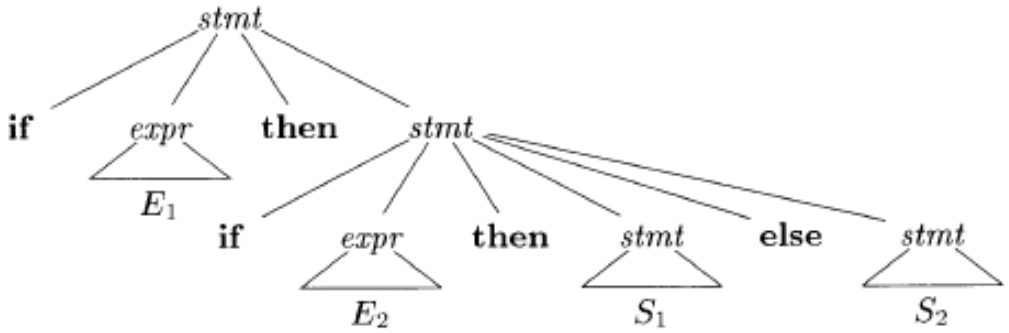
图 4-5　**id + id * id** 的两棵语法树

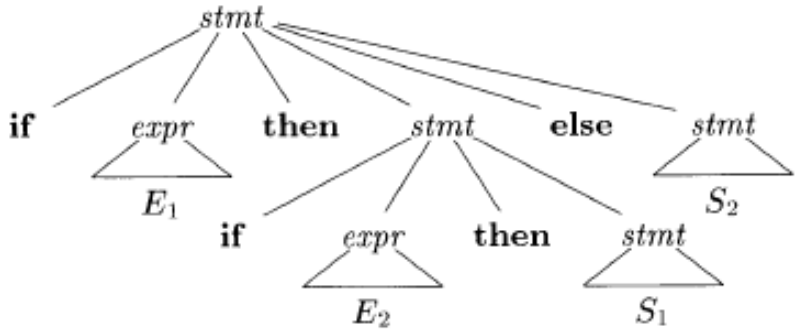
文法的二义性

- 例子：

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

- **if** E_1 **then if** E_2 **then** S_1 **else** S_2 的两棵语法树
- 这个文法是可以被改造成等价的无二义性的文法





文法及其生成的语言

- 语言是由文法的开始符号出发，能够推导得到的所有句子的集合

- 文法G: $S \rightarrow aS|a|b$, $L(G)=\{a^i(a|b), i>=0\}$

- 文法G：$S \rightarrow aSb|ab$

  $L(G)=\{a^nb^n, n>=1\}$

- 文法G：$S \rightarrow (S)S|\varepsilon$

  L(G)={所有具有对称括号对的串}

验证文法生成的语言

- 验证文法G生成语言L可以帮助我们了解文法可以生成什么样的语言
- 基本步骤：
  - 首先证明L(G)⊆L：G生成的每个串都在L中
  - 然后证明L⊆L(G)：L的每个串都能由G生成
  - 一般使用数学归纳法
- 证明L(G)⊆L：按照推导序列长度进行数学归纳
- 证明L⊆L(G)：按照符号串的长度来构造推导序列

- 实例：文法G：S → (S)S|ε, L(G)={所有具有对称括号对的串}

词法分析与语法分析

- 上下文无关文法和正则表达式
- 正则表达式 → 词法
- 上下文无关文法 → 语法

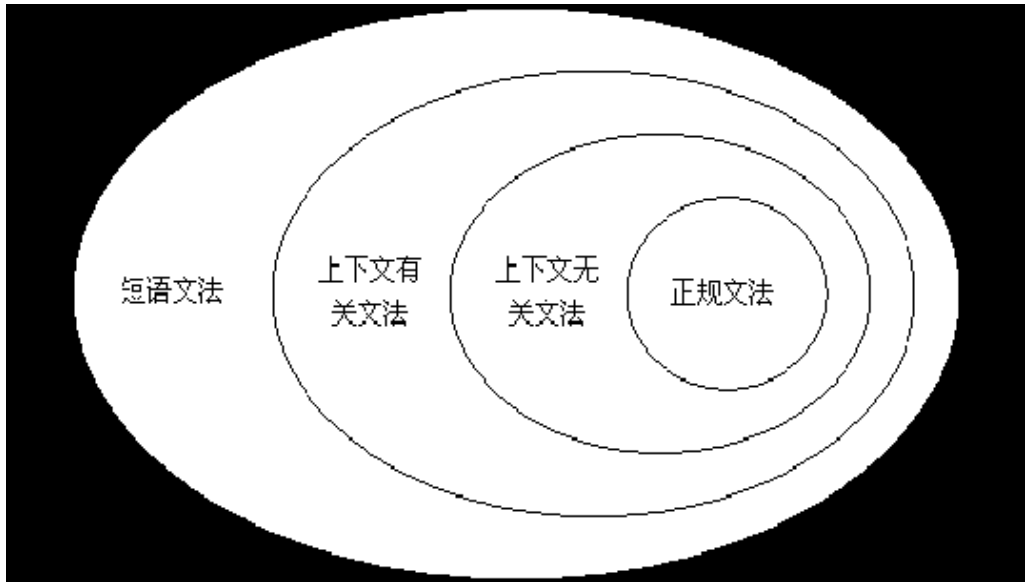
词法分析和语法分析比较

阶段	输入	输出	描述体系
词法分析	源程序符号串	词法单元序列	正则表达式
句法分析	词法单元序列	语法树	上下文无关文法

- 文法比正则表达式描述能力更强
- 正则表达式描述词法单元比较简洁
- 基于正则表达式构造的词法分析器效率更高
- 正则表达式适合描述词法结构，文法适合描述嵌套结构

补充

- 文法的类别，Chomsky文法类
  - 0型文法(短语结构文法)， $\alpha \rightarrow \beta$
  - 1型文法(上下文相关文法)， $\alpha A \beta \rightarrow \alpha \gamma \beta$
  - 2型文法(上下文无关文法)， $A \rightarrow \gamma$
  - 3型文法(正则文法)，$A \rightarrow t$, $A \rightarrow tB$



上下文无关文法和正则表达式

- 上下文无关文法的表达能力更强
- 每个正则表达式都可以用一个上下文无关文法来描述，反之不成立
- 每个正则语言都是一个上下文无关语言，反之不成立
- 正则表达式(a|b)*abb 等价于文法

$$A_0 \rightarrow a\ A_0 \mid b\ A_0 \mid a\ A_1$$
$$A_1 \rightarrow b\ A_2$$
$$A_2 \rightarrow b\ A_3$$
$$A_3 \rightarrow \varepsilon$$

上下文无关文法和正则表达式

- 上下文无关文法比正则表达式的能力更强，即：
  - 一些用文法描述的语言不能用正则文法描述
  - 所有的正则语言都可以使用文法描述
- 证明：
  - 首先S→aSb | ab 描述了语言{a^nb^n|n>0}，但是这个语言无法用DFA识别
    - 假设有DFA识别此语言，且这个文法有K个状态。那么在识别a^{k+1}…的输入串时，必然两次经过同一个状态。
    - 设自动机在第i个和第j个a时进入同一个状态，那么：因为DFA识别L，a^jb^j必然到达接受状态，因此a^ib^j必然也到达接受状态
    - 直观地讲：有穷自动机不能数（大）数

上下文无关文法和正则表达式

- 证明（续）
  - 其次证明：任何正则语言都可以表示为上下文无关文法的语言
  - 任何正则语言都必然有一个NFA。对于任意的NFA构造如下的上下文无关文法
    - 对NFA的每个状态i，创建非终结符号A_i
    - 如果有i在输入a上到达j的转换，增加产生式$A_i \rightarrow aA_j$
    - 如果i在输入ε上到达j，那么增加产生式$A_i \rightarrow A_j$.
    - 如果i是一个接受状态，增加产生式$A_i \rightarrow \varepsilon$
    - 如果i是开始状态，令A_i为所得文法的开始符号

为NFA构造等价文法

1）对于 NFA 的每个状态 i，创建一个非终结符号 A_i。

2）如果状态 i 有一个在输入 a 上到达状态 j 的转换，则加入产生式 $A_i \rightarrow aA_j$。如果状态 i 在输入 ϵ 上到达状态 j，则加入产生式 $A_i \rightarrow A_j$。

3）如果 i 是一个接受状态，则加入产生式 $A_i \rightarrow \epsilon$。

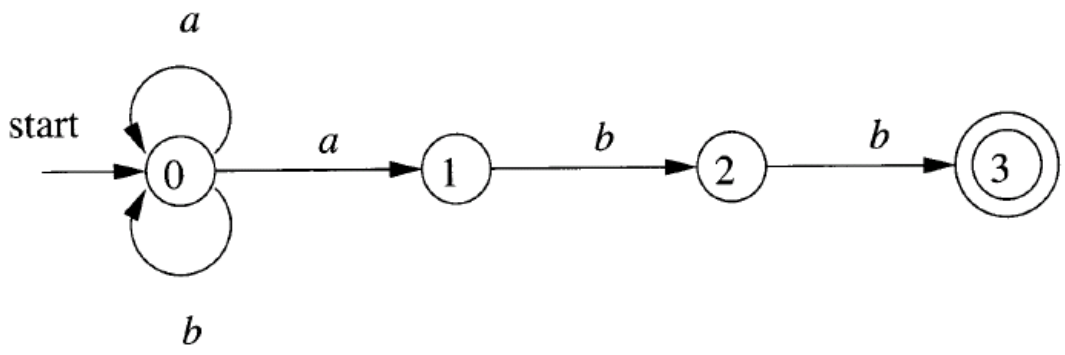
4）如果 i 是自动机的开始状态，令 A_i 为所得文法的开始符号。

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

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$



- **考虑baabb的推导和接受过程**
- **NFA接受一个句子的运行过程实际是文法推导出该句子的过程**

上下文无关文法和正则表达式

- 实例：

给出语言$\{a^n b^j c^i \mid n \geq 0, j \geq 0, 0 \leq i \leq n\}$

  - 能用正则文法描述吗？
  - 描述它的文法是怎样的？

文法的设计

- 程序设计语言所需要的文法
  - 上下文无关文法足够用来描述语法吗？
    - 标识符必须先声明后使用
  - 语法分析器能够完全按照上下文无关文法来构造吗？
    - 二义性、左递归的文法可能给语法分析器造成很大麻烦
- 可以怎么做？
  - 改造语法分析器，添加语义规则，使它可以做得更多
  - 改造上下文无关文法，消除二义性和左递归

文法的设计

- 在进行**高效**的语法分析之前，需要对文法做以下处理
  - 消除二义性
    - 文法的二义性：文法可以为一个句子生成多颗不同的树
  - 消除左递归
    - 左递归：文法中一个非终结符号A使得对某个串α，存在一个推导 $A \overset{+}{\Rightarrow} A\alpha$ ，则称这个文法是左递归的
  - 提取左公因子

消除文法的二义性

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

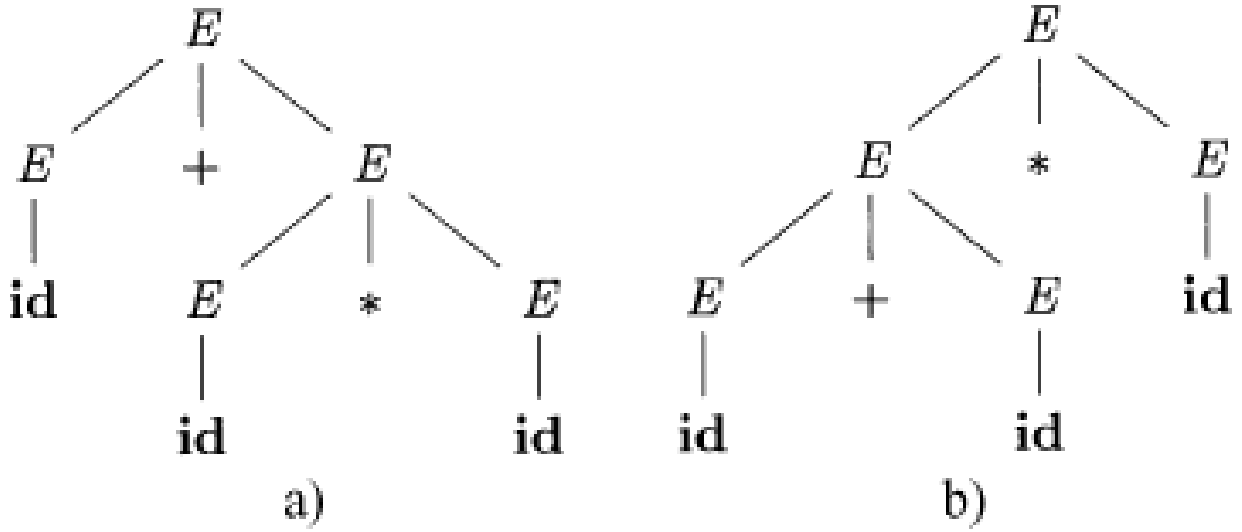


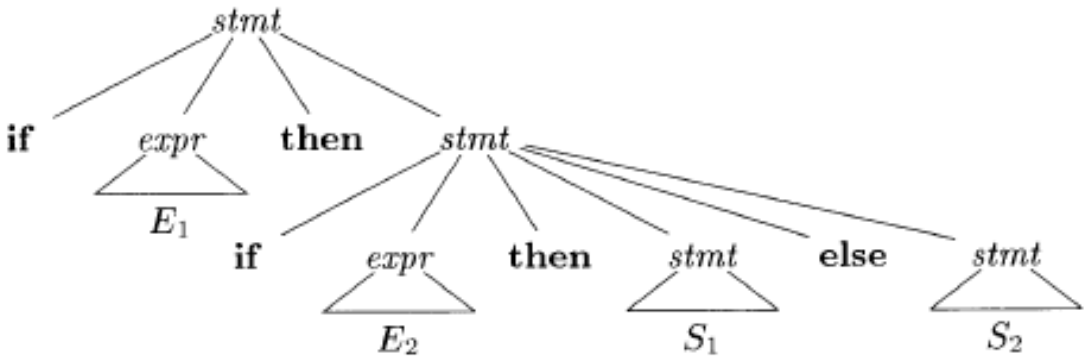
图 4-5　**id** + **id** * **id** 的两棵语法树

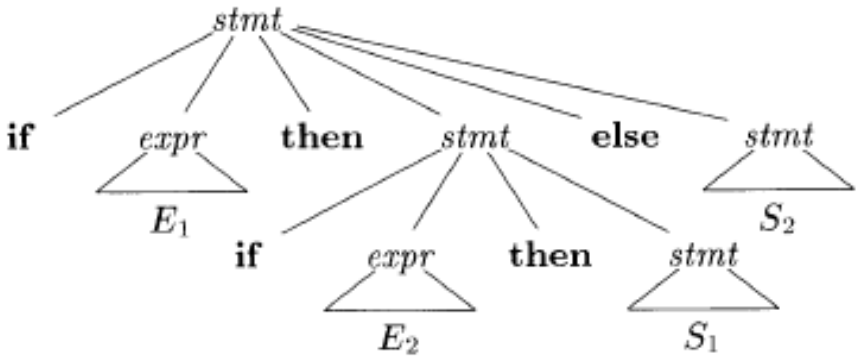
消除文法的二义性

- 一些二义性文法可以被改成等价的无二义性的文法
- 例子：

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

- **if** E_1 **then if** E_2 **then** S_1 **else** S_2 的两棵语法树





消除文法的二义性（续）

- 改写文法，基本思想: 在一个then和一个else之间出现的语句必须是“已匹配的”。也就是说then和else中间的语句不能以一个尚未匹配的then结尾
- 解决方案：引入新的非终结符号matched_stmt，用来区分是否是成对的then/else

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

if E_1 **then if** E_2 **then** S_1 **else** S_2

消除文法的二义性（续）

- 二义性的消除方法没有规律可循
- 通常并不是通过改变文法来消除二义性

消除左递归

- 左递归的定义
  - 文法中一个非终结符号A使得对某个串α，存在一个推导 $A \overset{+}{\Rightarrow} A\alpha$ ，则称这个文法是左递归的
- 立即左递归
  - 如果存在$A \rightarrow A\alpha$,则称为立即左递归
- 为什么要消除左递归？
  - 自顶向下的语法分析技术不能处理左递归的文法
- 立即左递归的消除实例：

$$A \rightarrow A\alpha \mid \beta \quad \Longrightarrow \quad \begin{array}{l} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{array}$$

立即左递归的消除

- 假设非终结符号A存在立即左递归的情形，假设以A为左部的规则有：

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

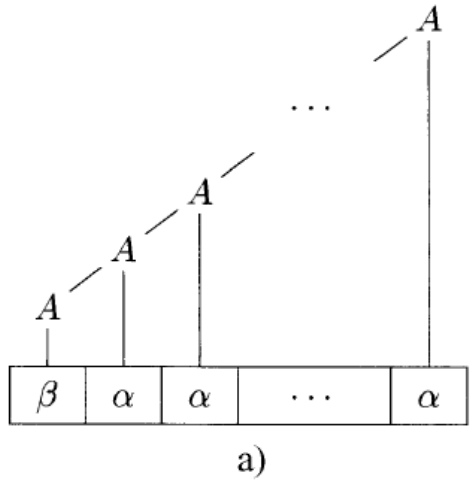
首先对A产生式**分组**(所有 α_i不等于 ε， β_i不以A开头)

可以**替换**为

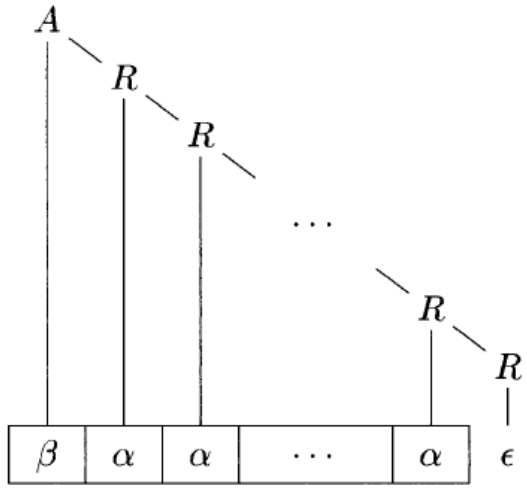
$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

- 由A生成的串总是以某个 β_i开头，然后跟上零个或者多个 α_i的重复



a)



b)

消除立即左递归

- $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
- 分组

$A \rightarrow$ $A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m$ $\mid$ $\beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

- 替换

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

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$

立即左递归消除示例

$$
\begin{aligned}
&E \longrightarrow E + T \mid T \\
&T \longrightarrow T * F \mid F \\
&F \longrightarrow (E) \mid \mathbf{id}
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
&E \longrightarrow TE' \\
&E' \longrightarrow + \ TE' \mid \epsilon \\
&T \longrightarrow FT' \\
&T' \longrightarrow * \ FT' \mid \epsilon \\
&F \longrightarrow (\ E \ ) \mid \mathbf{id}
\end{aligned}
$$

消除多步左递归

- 消除立即左递归的方法并不能消除因为两步或多步推导而产生的左递归

- 文法：$S \rightarrow Aa \mid b, \quad A \rightarrow Ac \mid Sd \mid \varepsilon$
- $S \Rightarrow Aa \Rightarrow Sda$

- 如何消除？

消除算法

- 输入：没有环A_i　　A_i和 $A \rightarrow \varepsilon$
- 输出：一个等价的无左递归的文法

- 算法原理：
  - 给非终结符号**排序**，A_1，A_2，…，A_n
  - 如果只有$A_i \Rightarrow A_j$ (i<j)，则不会有左递归
  - 如果发现$A_i \Rightarrow A_j$ (i>j)，**代入** A_j 的当前产生式，若替换后有 A_i 的直接左递归，再**消除**

通用的左递归消除方法

- 输入：没有环和ε产生式的文法G
- 输出：等价的无左递归的文法
- 步骤：

将文法的非终结符号任意排序为$A_1,A_2,\ldots,A_n$

```
for i=1 to n do {
    for j = 1 to i-1 do
    {
    将形如Ai→ Ajγ的产生式替换为Ai→δ1γ| δ2γ | … |δkγ，
    其中Aj→ δ1| δ2|…| δk是以Aj为左部的所有产生式；
    }
    消除Ai的立即左递归；
}
```

通用的左递归消除方法（续）

1. 内层循环的每一次迭代保证了A_i的产生式的右部首符号都比A_j更加靠后
2. 当内层循环结束时，A_i的产生式的右部首符号不先于A_i
3. 消除立即左递归就保证了A_i的产生式的右部首符号必然比A_i后。（不能有环和ε产生式）
4. 当外层循环结束时，所有的产生式都是如此。使用这些产生式当然不会产生左递归的情形

消除算法（示例）

- S → Aa | b,　　A → Ac | Sd | ε
- 排序：　S　A
- 替换：
  - i=1，没有处理
  - i=2, 替换 A→Sd 中的 S，得到 A → Ac | Aad | bd | ε
- 消除立即左递归

$$S \rightarrow A\ a \mid b$$

$$A \rightarrow b\ d\ A' \mid A'$$

$$A' \rightarrow c\ A' \mid a\ d\ A' \mid \epsilon$$

提取左公因子

- 在推导的时候，不知道该如何选择（自顶向下算法会详细描述）

$$stmt \longrightarrow \textbf{if}\ expr\ \textbf{then}\ stmt\ \textbf{else}\ stmt$$
$$\mid\ \textbf{if}\ expr\ \textbf{then}\ stmt$$

$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ $\Longrightarrow$ $A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 \mid \beta_2$

提取左公因子算法

- 输入：文法*G*
- 输出：一个等价的提取了左公因子的文法
- 方法：对于每个非终结符号*A*，找出它的两个或多个可选项之间的最长公共前缀α，且α≠ ε ，那么将*A*所有的产生式

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$$

替换为

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

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

提取左公因子

$$S \rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a$$

$$E \rightarrow b$$

对于S而言，最长的前缀是i E t S，因此：

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

$$S' \rightarrow e\ S \mid \epsilon$$

$$E \rightarrow b$$

非上下文无关语言的构造

- 抽象语言 L1={wcw | w在(a|b)*中}
- 这个语言不是上下文无关的语言
- 它抽象地表示了C或者Java中“标识符先声明后使用”的规则
  - 说明了C/Java这些语言不是上下文无关语言，不能使用上下文无关文法描述。
  - 通常用上下文无关文法描述其基本结构，不能用文法描述的特性在语义分析阶段完成。原因是上下文文法具有高效的处理算法。

自顶向下分析技术

- 自顶向下分析可以被看作是为输入串构造语法分析树的问题，也可以看作一个寻找输入串的最左推导的过程

- 问题
  - 在推导的每一步，对非终结符号A，应用哪个产生式，以可能产生于输入串相匹配的终结符号串

id+id*id的自顶向下分析

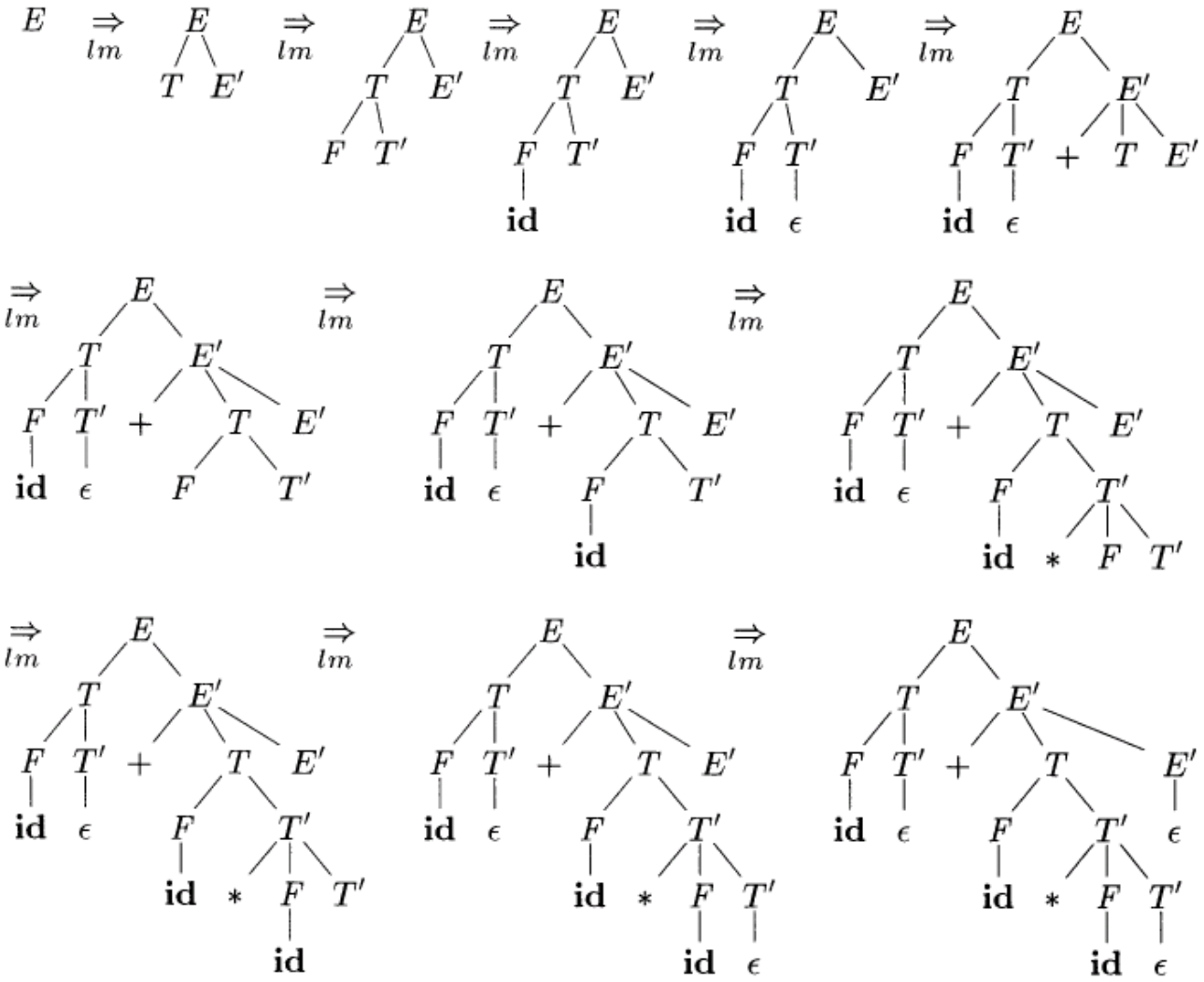
$$E \longrightarrow T\ E'$$

$$E' \longrightarrow +\ T\ E' \mid \epsilon$$

$$T \longrightarrow F\ T'$$

$$T' \longrightarrow *\ F\ T' \mid \epsilon$$

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



自顶向下语法分析

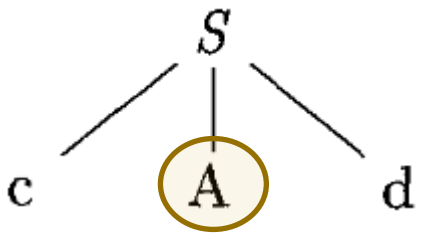
- 问题：对于非终结符号*A*，选择哪一个产生式
- 一种通用的递归下降分析框架
  - 由一组过程组成，每个非终结符号对应一个过程
  - 程序的执行从开始符号对应的过程开始
  - 每个过程的功能是：选择一个产生式体，扫描相应的句子。若遇到非终结符号，调用该符号对应的过程

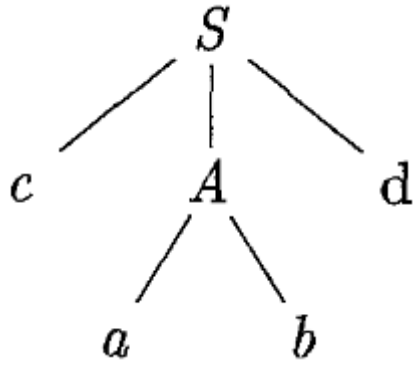
```
void A() {
1)      选择一个 A 产生式，A → X1X2···Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi 是一个非终结符号 )
4)              调用过程Xi();
5)          else if ( Xi 等于当前的输入符号a )
6)              读入下一个输入符号;
7)          else /* 发生了一个错误 */;
        }
}
```

递归下降分析过程示例

- 输入串$w = c\ a\ d$
- 文法：
  - $S \rightarrow cAd$
  - $A \rightarrow ab \mid a$

```
void A() {
1)      选择一个 A 产生式，A → X1X2···Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi 是一个非终结符号 )
4)              调用过程Xi();
5)          else if ( Xi 等于当前的输入符号a )
6)              读入下一个输入符号;
7)          else /* 发生了一个错误 */;
        }
}
```

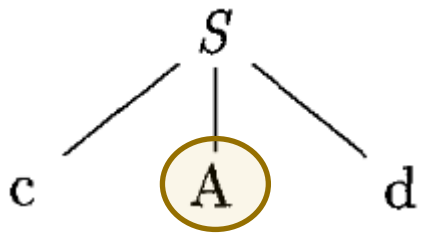


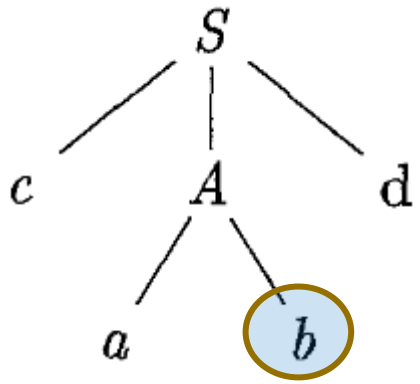


递归下降分析过程示例

- 输入串$w = c\ a\ d$
- 文法：
  - $S \rightarrow cAd$
  - $A \rightarrow ab \mid a$

```
void A() {
1)      选择一个 A 产生式，A → X1X2···Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi 是一个非终结符号 )
4)              调用过程Xi();
5)          else if ( Xi 等于当前的输入符号a )
6)              读入下一个输入符号;
7)          else /* 发生了一个错误 */;
        }
    }
```

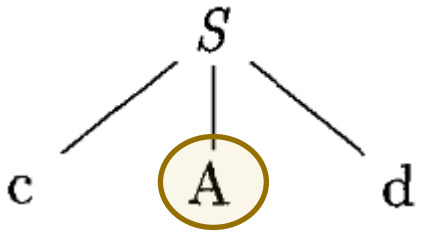


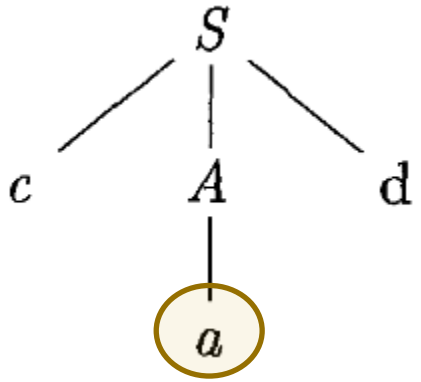


递归下降分析过程示例

- 输入串 $w = c\ a\ d$
- 文法：
  - $S \rightarrow cAd$
  - $A \rightarrow ab \mid a$

```
void A() {
1)      选择一个 A 产生式，A → X1X2···Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi 是一个非终结符号 )
4)              调用过程Xi();
5)          else if ( Xi 等于当前的输入符号a )
6)              读入下一个输入符号;
7)          else /* 发生了一个错误 */;
        }
}
```

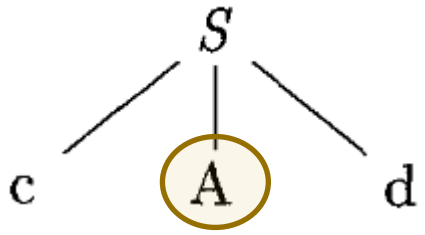


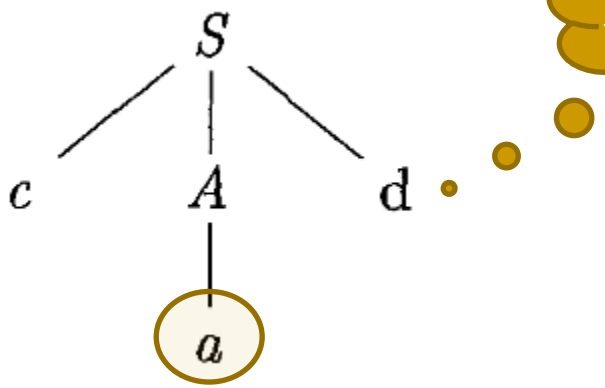


递归下降分析过程示例

- 输入串$w = c\ a\ d$
- 文法：
  - $S \rightarrow cAd$
  - $A \rightarrow ab \mid a$

```
void A() {
1)      选择一个 A 产生式，A → X1X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi 是一个非终结符号 )
4)              调用过程Xi();
5)          else if ( Xi 等于当前的输入符号a )
6)              读入下一个输入符号;
7)          else /* 发生了一个错误 */;
        }
}
```





匹配成功！

递归下降过程示例(续)

- 可能需要回溯，或者说，可能需要重复扫描输入
- “在回到A时，我们必须把输入指针重新设置到位置2，即我们第一次尝试展开A时该指针指向的位置。这意味着A的过程必须将输入指针存放在一个局部变量中。”

- 如果文法中存在左递归……

- 回溯（穷试 ☺）显然是笨办法

预测分析法简介

- 试图从开始符号推导出输入符号串
- 以开始符号作为初始的当前句型
- 每次为最左边的非终结符号选择适当的产生式
  - 通过查看下一个输入符号来选择这个产生式
  - 有多个可能的产生式时预测分析法无能为力
- 比如文法：E→ +EE | -EE | id；输入为+ **id - id id**
- 当两个产生式具有相同的前缀时无法预测
  - 文法：*stmt* → **if** *expr* **then** *stmt* **else** *stmt* | **if** *expr* **then** *stmt*
  - 输入：**if** a **then** …
  - 新文法：*stmt* → **if** expr **then** *stmt elsePart*
  - *elsePart* → **else** stmt | ε
- 需要提取公因子

预测分析技术

消除二义性
消除左递归
提取左公因子

- 一种确定性的、无回溯的分析技术
- 在每一步选择正确的产生式

  - 例1：文法G(S)：

    S→pA

    S→qB

    A→cAd

    A→a

  - 输入串 W=pccadd

预测分析技术（续）

- 例2：文法G(S)为：

  S →Ap

  S → Bq

  A →a|cA

  B→b|dB

- 输入W=ccap

预测分析技术（续）

- 通过在输入中向前看固定多个符号来选择正确的产生式
- 通常情况下，我们只需要向前看一个符号
  - 递归下降分析一般只适合于每个子表达式的第一个终结符能够为产生式选择提供足够信息的那些文法
- 给出两个与文法相关的两个函数
  - FIRST
  - FOLLOW
- 基于上述两个函数，可以根据下一个输入符号来选择应用哪个产生式

FIRST和FOLLOW（1）

- 在自顶向下的分析技术中，通常使用向前看几个符号来唯一地确定产生式（这里假定只看一个符号）
- 当前句型是xAβ，而输入是xa…。那么选择产生式A→α的必要条件是下列之一：
  - $A \overset{*}{\Rightarrow} \varepsilon$，且β以a开头；也就是说在某个句型中a跟在A之后；
  - $A \overset{*}{\Rightarrow}$ a…；
  - 如果按照这两个条件选择时能够保证唯一性，那么我们就可以避免回溯。
- 因此，我们定义FIRST和FOLLOW

FIRST和FOLLOW（2）

- FIRST(α):
  - 可以从α推导得到的串的首符号的集合；
  - 如果α $\overset{*}{\Rightarrow}$ ε，那么ε也在FIRST(α)中；
- FOLLOW(A):
  - 可能在某些句型中紧跟在A右边的终结符号的集合。

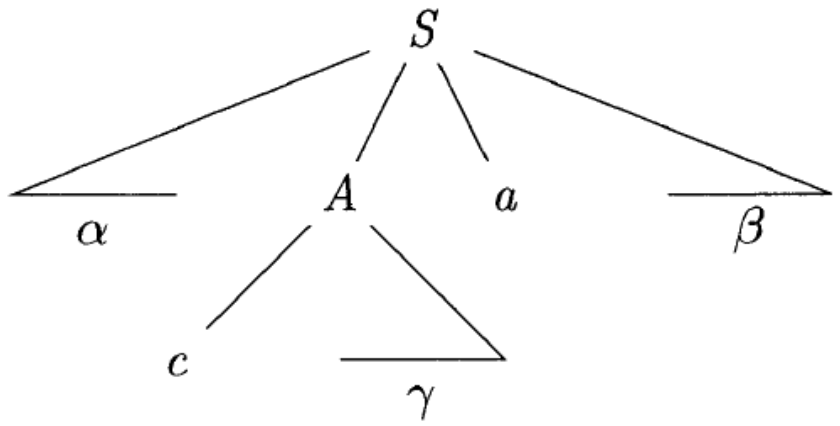


图 4-15　终结符号 c 在 FIRST(A) 中且 a 在 FOLLOW(A) 中

FIRST(α)

- 定义：可从α推导得到的串的首符号的集合，其中α是任意的文法符号串。如果$\alpha \overset{*}{\Rightarrow} \varepsilon$，那么ε也在FIRST(α)中
- FIRST函数的**意义**
  - 如果两个A产生式 $A \to \alpha \mid \beta$，其中First(α)和First(β)是不相交的集合。下一个输入符号是a，若a∈ First(α)，则选择$A \to \alpha$，若a∈ First(β)，则选择$A \to \beta$
  - ***用于预测产生式的选择***

First的计算

- 对于文法符号X的FIRST(X)，通过不断应用下列规则，直到没有新的终结符号或者ε可以加入到任何FIRST集合为止
  - 如果X是终结符号，那么FIRST(X)={X}
  - 如果X是非终结符号，且有规则$X \rightarrow a\ldots$,那么将a添加到FIRST(X)中; 如果$X \rightarrow \varepsilon$，那么ε也在FIRST(X)中
  - 对于规则$X \rightarrow Y_1Y_2\ldots Y_n$，把$FIRST(Y_1)$中的非ε符号添加到FIRST(X)中。如果ε在$FIRST(Y_1)$中，把$FIRST(Y_2)$中的非ε符号添加到FIRST(X)中…；如果ε在$FIRST(Y_n)$中，把ε添加到FIRST(X)中

First的计算（续）

- 对于文法符号串$X_1X_2...X_n$的First集合
  - 向First($X_1X_2...X_n$)加入First(X_1)中所有的非ε符号。
  - 如果ε在First(X_1)中，再加入First(X_2)中的所有非ε符号。如果ε在First(X_1)和First(X_2)中，再加入First(X_3)中的所有非ε符号。依次类推。
  - 如果对所有的i（1到n），ε都在First(X_i)中，则ε加入First($X_1X_2...X_n$)中。

First的计算示例

$$E \rightarrow T\ E'$$

$$E' \rightarrow +\ \ T\ E' \mid \epsilon$$

$$T \rightarrow F\ T'$$

$$T' \rightarrow *\ \ F\ T' \mid \epsilon$$

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

- First(F)=First(T)=First(E)
  ={(,id}
- First(E')={+, ε}
- First(T')={*, ε}
- First(TE')=First(T)={(,id}
- First(+TE')={+}
- ……

一个实例

- 文法G[S], 输入bcd

  - S → AB | CD
  - A →aD | ε
  - C →cD
  - B →bC
  - D →d

  - First(S) ={a, b, c}
  - First(A) = {a, ε }
  - First(B) = {b}
  - First(C) = {c}
  - First(D) = {d}

FOLLOW函数

- 对于非终结符号A，FOLLOW(A)定义为可能在某些句型中紧跟在A右边的终结符号的集合
  - 例如$S \overset{*}{\Rightarrow} \alpha Aa\beta$，终结符号$a \in$ Follow(A)
- 如果A是某些句型的最右符号，那么$ ∈Follow(A)。 $是特殊的输入串“结束标记”
- FOLLOW函数的意义：
  - 如果$A \rightarrow \alpha$，当$\alpha \rightarrow \varepsilon$或$\alpha \overset{*}{\Rightarrow} \varepsilon$时，FOLLOW(A)可以帮助我们做出选择恰当的产生式
  - 例如：如果$A \rightarrow \alpha$, b属于FOLLOW(A)，如果$\alpha \overset{*}{\Rightarrow} \varepsilon$，则若当前输入符号是b，可以选择$A \rightarrow \alpha$，因为A最终到达了ε，而且后面跟着b

文法G[S], 输入bcd

S → AB|CD
A →aD| ε
C →cD
B →bC
D →d

FOLLOW计算

- 计算各个非终结符号A的FOLLOW(A)集合，不断应用下列规则，直到没有新的终结符号可以被加入到任意FOLLOW集合中
  - 将$放入FOLLOW($S$)，$S$是开始符号，而$是输入串的结束标记
  - 如果存在产生式$A \rightarrow \alpha B\beta$，那么First($\beta$)中除ε之外的所有符号都在Follow(B)中
  - 如果存在一个产生式$A \rightarrow \alpha B$，或存在产生式$A \rightarrow \alpha B\beta$且First($\beta$)包含ε，那么Follow($A$)中的所有符号都在Follow($B$)中

Follow的计算示例

- 文法

  E → T E’　　E’ → +T E’ | ε　　T → F T’

  T’ → *F T’ | ε　　F → (E) | i

- FOLLOW(E)={) ， $}　← FIRST( )) U {$}
- FOLLOW(E’)={) ， $}　← FOLLOW(E)
- FOLLOW(T)={+，)，$}　← FIRST(E’) U FOLLOW(E’)
- FOLLOW(T’)={+，)，$}　← FOLLOW(T)
- FOLLOW(F)=(*，+，)，$)　← FIRST(T’) U FOLLOW(T)

- 将$放入FOLLOW(*S*)，*S*是开始符号，而$是输入串的结束标记
- 如果存在产生式*A* → *αBβ*，那么First(*β*)中除ε之外的所有符号都在Follow(B)中
- 如果存在一个产生式*A* → *αB*，或存在产生式*A* → *αBβ*且First(*β*)包含ε，那么Follow(*A*)中的所有符号都在Follow(*B*)中

课堂练习

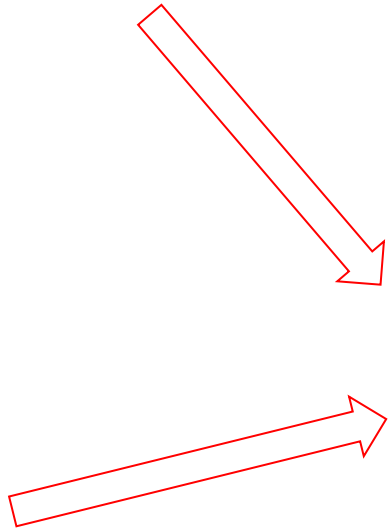
- 有如下文法：
  - **E → id X**
  - **X → ε | (A) | [E]**
  - **A → EY**
  - **Y → ε | ;A**

- 给出各个非终结符号的First集和Follow集合。

思考： 为什么

- 消除二义性
- 消除左递归
- 提取左公因子
- 求First和Follow集合

改造文法



LL(1)文法的预测分析

LL(1)文法（1）

- LL(1)文法，第一个L表示自左向右扫描，第二个L指产生最左推导，而1则表示向前看一个输入符号

- LL(1)文法可以利用不回溯的确定性的预测分析技术，因为只需要检查当前输入符号就可以为一个非终结符号选择正确的产生式

LL(1)文法（2）

- 定义:对于文法的任意两个不同的产生式A→α|β
  - 不存在终结符号a使得α和β都可以推导出以a开头的串
  - α和β最多只有一个可以推导出空串
  - 如果β可以推导出空串，那么α不能推导出以FOLLOW 中任何终结符号开头的串；
- 等价于：
  - FIRST(α)∩FIRST(β) = Φ;（条件一、二）
  - 如果ε∈FIRST(β)，那么FIRST(α) ∩FOLLOW(A) = Φ；反之亦然。（条件三）

LL(1)文法（3）

- 思考：对于一个LL(1)文法，我们期望从A**推导**出以终结符号a开头的符号串
  - 我们如何选择产生式？
  - 有多个选择吗？

LL(1)文法（4）

- 对于LL(1)文法，可以在自顶向下分析过程中，根据当前输入符号来确定使用的产生式

- 例如：
  - 产生式：*stmt* → **if (exp)** *stmt* **else** *stmt* | **while (exp)** *stmt* | **a**
  - 输入：**if (exp) while (exp) a else a**

  - 首先选择产生式stmt → **if (exp)** *stmt* **else** *stmt* 得到句型**if (exp)** *stmt* **else** *stmt*
  - 然后发现要把句型中的第一个stmt展开，选择产生式*stmt* → **while (exp)** *stmt*，得到句型**if (exp) while (exp)** *stmt* **else** *stmt* 。
  - 然后再展开第一个stmt，得到**if (exp) while (exp) a else** *stmt*
  - …

LL(1)文法的预测分析技术

- LL(1)文法 == > 预测分析表
- 将First和Follow集合中的信息放入一个预测分析表M[A,a]，该预测表告诉我们当非终结符号为A，当前输入符号为a时，要选择哪条产生式
- 预测分析表的构造
  - 若当前输入符号a在First(α)中，选择产生式A → α
  - 若$\alpha \overset{*}{\Rightarrow} \varepsilon$，如果a在Follow(A)中，选择产生式A → α
  - 如果当前符号a是$，且$在Follow(A)中，则选择产生式A → α

非终结符号	输入符号					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

预测分析表构造

- 输入：文法G
- 输出：预测分析表M

- 方法：对于文法G的每个产生式A→α，进行如下处理
  - 对于First(α)中的每个终结符号a，将A→α加入到M[A,a]
  - 如果ε在First(α)中，那么对于Follow(A)中的每个终结符号b，将A→α加入到M[A,b]中
  - 如果ε在First(α)中，且$在Follow(A)中，将A→α加入到M[A,$]中

- 完成上述操作后，若M[A,a]中没有产生式，填为Error

预测分析表的例子

- 文法：
  - E → TE’　　E’ → +TE’ | ε
  - T → FT’　　T’ → *FT’ | ε
  - F →(E) | **id**
- FIRST集：
  - F：{(, id}；　E,T：{(,id}；　E’：{+, ε}；　T’：{*, ε}
- FOLLOW集：
  - E：{$,)}；　E’：{$,)}；　T,T’：{+,), $}；　F：{+,*,),$}

这个例子恰巧使得每个产生式的右部的第一个符号的FIRST集合就等于产生式右部的FIRST集合。但是在一般情况下不总是这样的。

非终结符号	输入符号					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

预测分析方法

- 对于任何文法G，都可以构造预测分析表
- 对于LL(1)文法，预测表中每个条目都唯一地指定了一个产生式，或者标明Error

二义性文法的预测分析表

- 对于某些文法，表中可能会有一些多重定义的条目，比如左递归或二义性文法。

$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \epsilon$

$E \rightarrow b$

- **FIRST(eS)={e}，使得S’→eS在M[S’,e]中；**
- **FOLLOW(S’)={$,e}使得S’→ ε也在M[S’,e]中**

非终结符号	输入符号					
	a	b	e	i	t	$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow \epsilon$ $S' \rightarrow eS$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

文法和自顶向下分析

- 设计文法(保证描述能力，且适用于自顶向下分析)
  - 消除二义性
  - 消除左递归
  - 提取左公因子
- 自顶向下分析
  - 递归下降分析 （通用的可能需要回溯）
  - 预测分析
- LL(1)文法及其预测分析

非递归的预测分析（1）

- 在自顶向下分析的过程中，我们总是
  - 匹配掉句型中左边的所有终结符号
  - 对于最左边的非终结符号，我们选择适当的产生式展开
  - 匹配成功的终结符号不会再被考虑，因此我们只需要记住句型的余下部分，以及尚未匹配的输入终结符号串
  - 由于展开的动作总是发生在余下部分的左端，我们可以用**栈来存放这些符号**

非递归的预测分析（2）

- 分析处理过程
  - 初始化时，栈中仅包含开始符号
  - 如果栈顶元素是终结符号，那么进行匹配
  - 如果栈顶元素是非终结符号
    - 使用预测分析表来**选择**适当的产生式
    - 在栈顶用产生式右部**替换**产生式左部
- 对所有文法的预测分析都可以共用同样的驱动程序

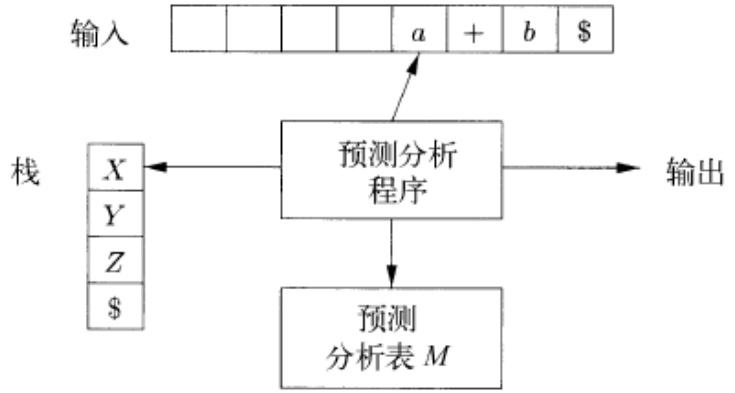


图 4-19　一个分析表驱动的预测分析器的模型

非递归的预测分析技术

- 文法符号栈
- 输入缓冲区
- 控制程序
  - 在任何时刻，栈顶符号X与当前输入符号a决定了控制程序所应执行的分析动作。四种可能的动作
    - 如果X=a=$，则分析成功结束
    - 如果X=a≠$，则从栈中退去X，并把输入指针推进到指向下一个输入符号
    - 如果X是一个非终结符号，且分析表A的元素A[X][a]="$X \rightarrow X_1X_2\ldots X_k$"，则把栈顶的X替换为$X_kX_2\ldots X_1$（反向下推入栈，使得$X_1$在栈顶）
    - 除上述情况外的其它情况，调用出错程序

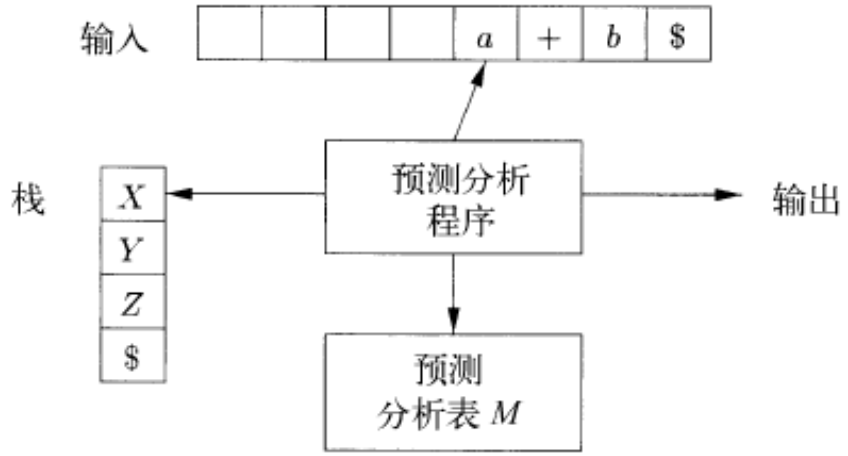


图 4-19　一个分析表驱动的预测分析器的模型

分析表驱动的预测分析器

- 性质1：如果栈中的符号序列为**α**，而**w**是已经被读入的部分输入，**w’**是尚未处理的输入；那么
  - S推导出wα
  - 我们试图从α推导出余下的输入终结符号串w’
- 预测分析程序使用M[X,a]来扩展X，将此产生式的右部按倒序压入栈中
- 请注意：这样的操作使得分析过程中性质1得到保持。

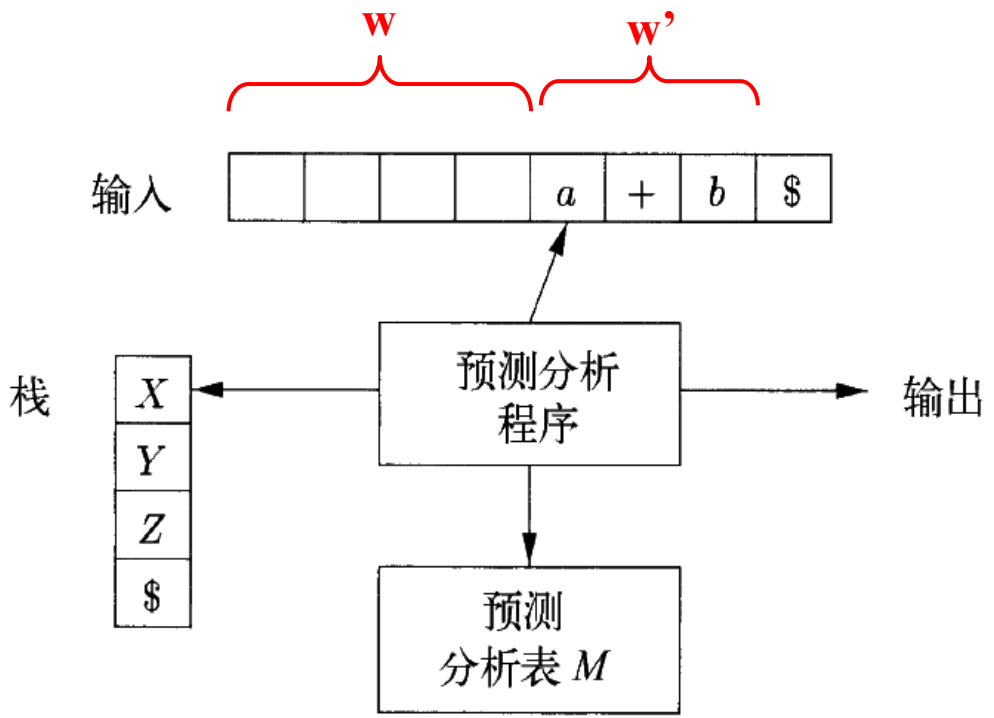


图 4-19　一个分析表驱动的预测分析器的模型

表驱动的非递归的预测语法分析

- 输入：一个串*w*，文法*G*的预测分析表*M*。
- 输出：如果*w*在*L(G)*中，输出*w*的一个最左推导；否则报错。
- 方法：初始化输入缓冲区为*w$*, 栈顶是*G*的开始符号*S*，下面是*$* 。

设置 ip 使它指向 w 的第一个符号，其中 ip 是输入指针;
令 X= 栈顶符号;
while ($X \neq \$$) { /* 栈非空 */
    if (X 等于 ip 所指向的符号 a) 执行栈的弹出操作，将 ip 向前移动一个位置;
    else if (X 是一个终结符号) $error()$;
    else if ($M[X,a]$ 是一个报错条目) $error()$;
    else if ($M[X,a] = X \rightarrow Y_1 Y_2 \cdots Y_k$) {
        输出产生式 $X \rightarrow Y_1 Y_2 \cdots Y_k$;
        弹出栈顶符号;
        将 $Y_k, Y_{k-1}, \ldots, Y_1$ 压入栈中，其中 Y_1 位于栈顶。
    }
    令 X= 栈顶符号;
}

分析表驱动预测分析的例子

$E \rightarrow T\ E'$

$E' \rightarrow +\ T\ E' \mid \epsilon$

$T \rightarrow F\ T'$

$T' \rightarrow *\ F\ T' \mid \epsilon$

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

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

已匹配	栈	输入	动作
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	输出 $E \rightarrow TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	输出 $T \rightarrow FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	输出 $F \rightarrow \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	匹配 $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	输出 $T' \rightarrow \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	输出 $E' \rightarrow +\ TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	匹配 $+$

已经匹配部分加上栈中符号必然是一个**最左**句型

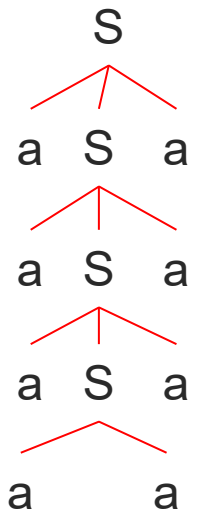
- 文法4.28，输入：id+id*id;

习题

- 文法：S → aSa | aa 生成了所有由a组成的长度为偶数的串。可以为这个文法设计一个带回溯的递归下降分析器。如果我们选择先使用产生式S → aa展开，那么只能识别到串aa。因此，任何合理的递归下降分析器将首先尝试S → aSa 。

- 说明这个递归下降分析器不能识别aaaaaa。

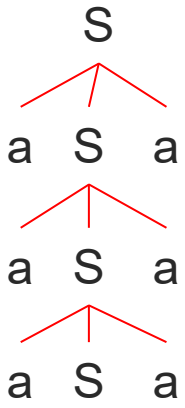
- $S \rightarrow aSa \mid aa$



21	S^4→aa	aaaaaa$	匹配成功，返回调用上一层；
22	S^3→ aSa	aaaaaa$	匹配成功，返回调用上一层；
23	S^2→ aSa	aaaaaa$	匹配失败，换产生式；
24	S^2→ aa	aaaaaa$	匹配成功；
25	S^2→aa	aaaaaa$	匹配成功，返回调用上一层；
26	S^1→ aSa	aaaaaa$	匹配成功；
27	隐藏符号$	aaaaaa$	匹配失败，换产生式；
28	S^1→ aa	aaaaaa$	匹配成功；
29	S^1→ aa	aaaaaa$	匹配成功；
30	隐藏符号$	aaaaaa$	匹配失败，程序结束，识别失败；

若要正确识别6个a，则应依次使用产生式S^1→aSa，S^2→aSa以及S^3→ aa。但是根据分析过程发现，第22步时使用产生式S^3→ aSa，a成功匹配后返回调用上一层。当第23步发现S^2→ aSa中a不能匹配时，实际上是S^3→ aSa这个产生式用错了，但是程序此时不会尝试（也无法尝试）使用S^3→ aa，只会尝试使用S^2→aa。因此导致最后错误发生。

- $S \rightarrow aSa \mid aa$

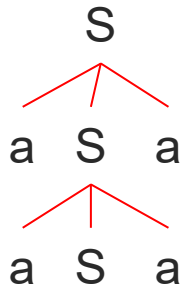


关键的一步
匹配成功，函数执行完毕
回不去继续尝试其他产生式了

21	$S^4 \rightarrow aa$	aaaaaa$	匹配成功，返回调用上一层；
22	$S^3 \rightarrow aSa$	aaaaaa$	匹配成功，返回调用上一层；
23	$S^2 \rightarrow aSa$	aaaaaa$	匹配失败，换产生式；
24	$S^2 \rightarrow aa$	aaaaaa$	匹配成功；
25	$S^2 \rightarrow aa$	aaaaaa$	匹配成功，返回调用上一层；
26	$S^1 \rightarrow aSa$	aaaaaa$	匹配成功；
27	隐藏符号$	aaaaaa$	匹配失败，换产生式；
28	$S^1 \rightarrow aa$	aaaaaa$	匹配成功；
29	$S^1 \rightarrow aa$	aaaaaa$	匹配成功；
30	隐藏符号$	aaaaaa$	匹配失败，程序结束，识别失败；

若要正确识别6个a，则应依次使用产生式$S^1 \rightarrow aSa$，$S^2 \rightarrow aSa$以及$S^3 \rightarrow aa$。但是根据分析过程发现，第22步时使用产生式$S^3 \rightarrow aSa$，a成功匹配后返回调用上一层。当第23步发现$S^2 \rightarrow aSa$中a不能匹配时，实际上是$S^3 \rightarrow aSa$这个产生式用错了，但是程序此时不会尝试（也无法尝试）使用$S^3 \rightarrow aa$，只会尝试使用$S^2 \rightarrow aa$。因此导致最后错误发生。

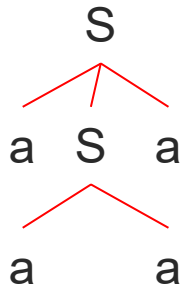
- $S \rightarrow aSa \mid aa$



21	S^4→aa	aaaaaa$	匹配成功，返回调用上一层；
22	S^3→ aSa	aaaaaa$	匹配成功，返回调用上一层；
23	S^2→ aSa	aaaaaa$	匹配失败，换产生式；
24	S^2→ aa	aaaaaa$	匹配成功；
25	S^2→aa	aaaaaa$	匹配成功，返回调用上一层；
26	S^1→ aSa	aaaaaa$	匹配成功；
27	隐藏符号$	aaaaaa$	匹配失败，换产生式；
28	S^1→ aa	aaaaaa$	匹配成功；
29	S^1→ aa	aaaaaa$	匹配成功；
30	隐藏符号$	aaaaaa$	匹配失败，程序结束，识别失败；

若要正确识别6个a，则应依次使用产生式S^1→aSa，S^2→aSa以及S^3→ aa。但是根据分析过程发现，第22步时使用产生式S^3→ aSa，a成功匹配后返回调用上一层。当第23步发现S^2→ aSa中a不能匹配时，实际上是S^3→ aSa这个产生式用错了，但是程序此时不会尝试（也无法尝试）使用S^3→ aa，只会尝试使用S^2→aa。因此导致最后错误发生。

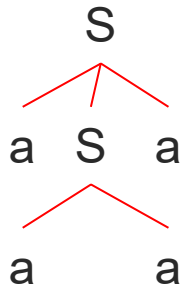
- $S \rightarrow aSa \mid aa$



21	S^4→aa	aaaaaa$	匹配成功，返回调用上一层；
22	S^3→ aSa	aaaaaa$	匹配成功，返回调用上一层；
23	S^2→ aSa	aaaaaa$	匹配失败，换产生式；
24	S^2→ aa	aaaaaa$	匹配成功；
25	S^2→aa	aaaaaa$	匹配成功，返回调用上一层；
26	S^1→ aSa	aaaaaa$	匹配成功；
27	隐藏符号$	aaaaaa$	匹配失败，换产生式；
28	S^1→ aa	aaaaaa$	匹配成功；
29	S^1→ aa	aaaaaa$	匹配成功；
30	隐藏符号$	aaaaaa$	匹配失败，程序结束，识别失败；

若要正确识别6个a，则应依次使用产生式S^1→aSa，S^2→aSa以及S^3→ aa。但是根据分析过程发现，第22步时使用产生式S^3→ aSa，a成功匹配后返回调用上一层。当第23步发现S^2→ aSa中a不能匹配时，实际上是S^3→ aSa这个产生式用错了，但是程序此时不会尝试（也无法尝试）使用S^3→ aa，只会尝试使用S^2→aa。因此导致最后错误发生。

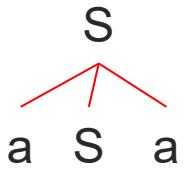
- $S \rightarrow aSa \mid aa$



21	S^4→aa	aaaaaa$	匹配成功，返回调用上一层；
22	S^3→ aSa	aaaaaa$	匹配成功，返回调用上一层；
23	S^2→ aSa	aaaaaa$	匹配失败，换产生式；
24	S^2→ aa	aaaaaa$	匹配成功；
25	S^2→aa	aaaaaa$	匹配成功，返回调用上一层；
26	S^1→ aSa	aaaaaa$	匹配成功；
27	隐藏符号$	aaaaaa$	匹配失败，换产生式；
28	S^1→ aa	aaaaaa$	匹配成功；
29	S^1→ aa	aaaaaa$	匹配成功；
30	隐藏符号$	aaaaaa$	匹配失败，程序结束，识别失败；

若要正确识别6个a，则应依次使用产生式S^1→aSa，S^2→aSa以及S^3→ aa。但是根据分析过程发现，第22步时使用产生式S^3→ aSa，a成功匹配后返回调用上一层。当第23步发现S^2→ aSa中a不能匹配时，实际上是S^3→ aSa这个产生式用错了，但是程序此时不会尝试（也无法尝试）使用S^3→ aa，只会尝试使用S^2→aa。因此导致最后错误发生。

- $S \rightarrow aSa \mid aa$



21	$S^4 \rightarrow aa$	aaaaaa$	匹配成功，返回调用上一层；
22	$S^3 \rightarrow aSa$	aaaaaa$	匹配成功，返回调用上一层；
23	$S^2 \rightarrow aSa$	aaaaaa$	匹配失败，换产生式；
24	$S^2 \rightarrow aa$	aaaaaa$	匹配成功；
25	$S^2 \rightarrow aa$	aaaaaa$	匹配成功，返回调用上一层；
26	$S^1 \rightarrow aSa$	aaaaaa$	匹配成功；
27	隐藏符号$	aaaaaa$	匹配失败，换产生式；
28	$S^1 \rightarrow aa$	aaaaaa$	匹配成功；
29	$S^1 \rightarrow aa$	aaaaaa$	匹配成功；
30	隐藏符号$	aaaaaa$	匹配失败，程序结束，识别失败；

若要正确识别6个a，则应依次使用产生式$S^1 \rightarrow aSa$，$S^2 \rightarrow aSa$以及$S^3 \rightarrow aa$。但是根据分析过程发现，第22步时使用产生式$S^3 \rightarrow aSa$，a成功匹配后返回调用上一层。当第23步发现$S^2 \rightarrow aSa$中a不能匹配时，实际上是$S^3 \rightarrow aSa$这个产生式用错了，但是程序此时不会尝试（也无法尝试）使用$S^3 \rightarrow aa$，只会尝试使用$S^2 \rightarrow aa$。因此导致最后错误发生。

自底向上语法分析

- 概述
- 移进-规约技术
- LR语法分析技术
  - 简单LR技术
  - LR技术

自底向上的语法分析

- 为一个输入串构造语法分析树的过程；
- 从叶子（输入串中的终结符号，将位于分析树的底端）开始，向上到达根结点
- 重要的自底向上语法分析的通用框架
  - 移入-归约
- LR：最大的可以构造出移入-归约语法分析器的语法类

自底向上语法分析概述 - 归约

- Bottom-Up Parsing

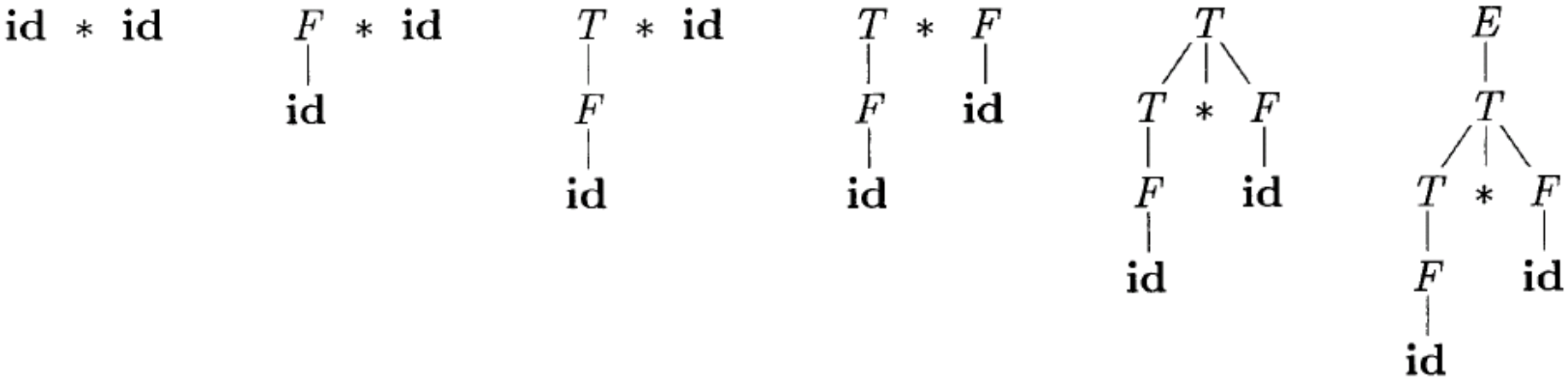


图 4-25　**id** * **id** 的自底向上分析过程

自底向上语法分析概述 - 归约

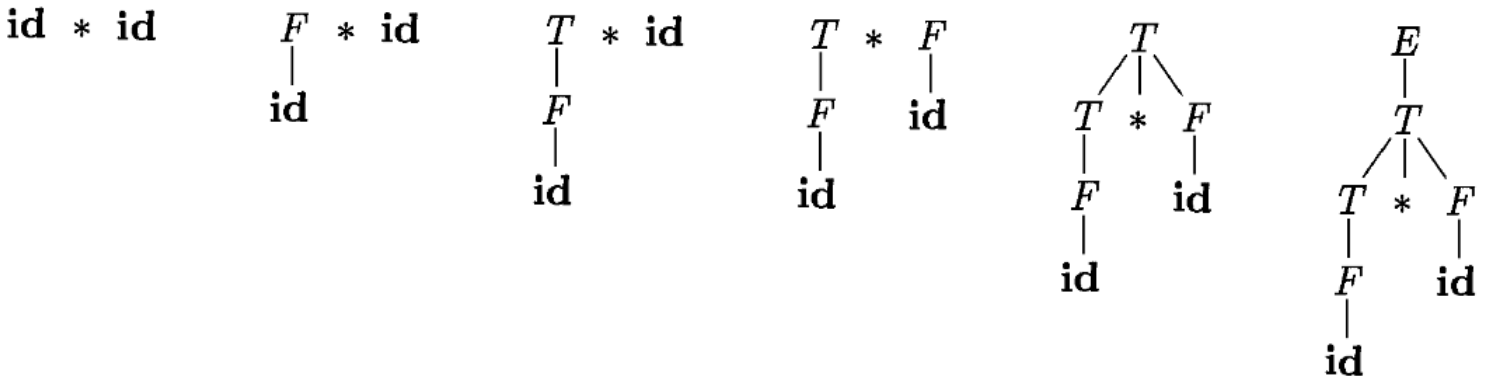
- Bottom-up parsing – 将一个串*w*归约为文法符号的过程

- 在每一步的归约中，一个与某产生式体相匹配的特定子串被替换为该产生式头部的非终结符号，一次归约实质上是一个推导的反向操作

归约的例子

- id * id的归约过程
  - id * id ，F*id，T*id，T*F，T，E
- 对于句型T*id，有两个子串和某产生式右部匹配
  - T是E → T的右部
  - id是F → id的右部
  - 为什么选择将id归约为F，而不是将T归约为E？
    - T归约为E之后，E*id不再是句型

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



自底向上分析- 规约

- 自底向上分析的过程也是规约的过程

- 规约的问题：
  - 选择哪部分进行归约?
  - 应用哪个产生式进行归约?

句型/句子/语言（回顾）

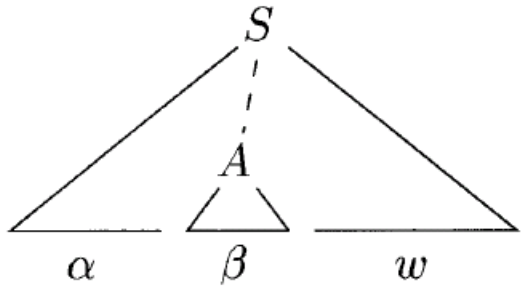
- 句型（sentential form）：
  - 如果$S \overset{*}{\Rightarrow} \alpha$，那么α就是文法的句型
  - 可能既包含非终结符号，又包含终结符号；可以是空串
- 句子（sentence）
  - 文法的句子就是不包含非终结符号的句型
- 语言
  - 文法G的语言就是G的句子的集合，记为L(G)
  - w在L(G)中当且仅当w是G的句子，即$S \overset{*}{\Rightarrow} w$

句柄

- 最右句型γ的一个句柄
  - 满足下述条件的产生式A→β及串β在γ中出现的位置
  - 条件：将这个位置上的β替换为A之后得到的串是γ的某个最右推导序列中出现在位于γ之前的最右句型
- 通俗地说
  - 句柄是最右推导的反向过程中被规约的那些部分
- 句柄的作用
  - 对句柄的规约，代表了相应的最右推导中的一个反向步骤

自底向上句法分析概述: 句柄剪枝

- Bottom-Up Parsing – 归约过程 – 反向最右推导过程 – 句柄剪枝过程
- 句柄：和某个产生式体相匹配的子串，对它的归约代表了相应的最右推导的一个反向步骤
- 句柄的形式定义：$S \overset{*}{\underset{rm}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha \beta w$，那么紧跟$\alpha$的$\beta$可以一步归约到$A$。 w是所有的终结符号串。



- 注意：
  - 归约前后都是最右句型
  - 和某个产生式匹配的最左子串不一定是句柄
  - 无二义性的文法，其每个最右句型都有且只有一个句柄

自底向上语法分析概述: 句柄剪枝（续）

- 通过“句柄剪枝”可以得到一个反向的最右推导。从句子w开始，令$w=\gamma_n$, γ_n是未知最右推导的第n个最右句型

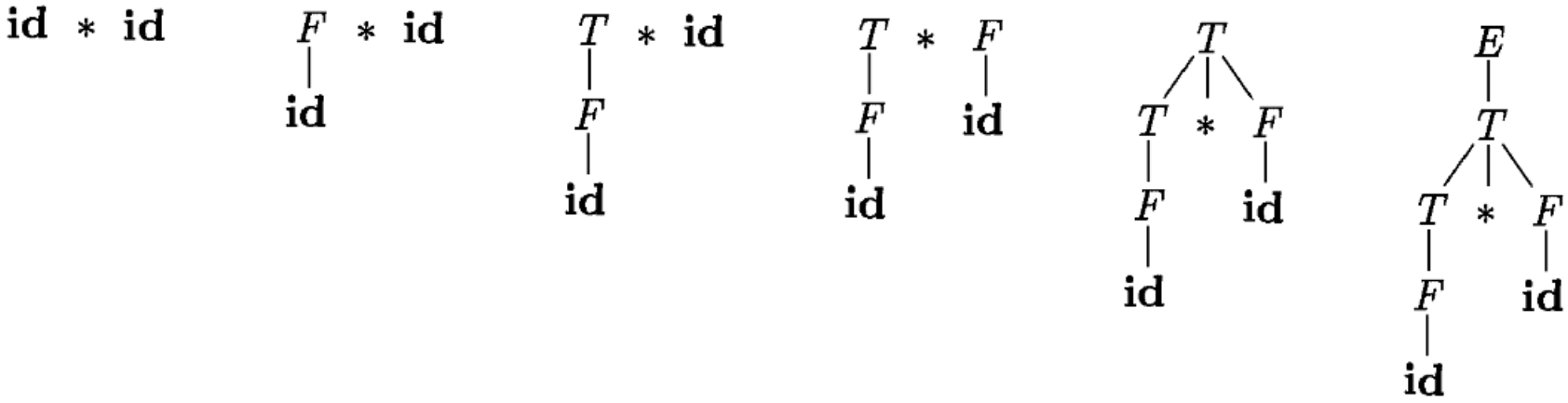
$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w$$

- 以相反的顺序重构这个推导（实际上是重构归约序列），我们在γ_n中寻找句柄β_n，并将替换为相应产生式$A \rightarrow \beta_n$的头部，得到前一个最右句型γ_{n-1}；重复这个过程，直到S
- 和推导类似，如果能重现这个序列，则可以完成语法分析任务（其中可能存在什么问题）
  - 如何找到句柄

句柄的例子

- 输入：id *id

最右句型	句柄	归约用的产生式
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$T \to T * F$



句柄的例子

- 已知文法

  E → E + T | E – T | T

  T → T * F | T / F | F

  F → (E) | id

- 句型 **T / (F – id)** 的句柄是什么？

- 句柄是**最右推导**的**反向**过程中**被规约的**那些部分

移进归约语法分析框架

- 一种自底向上的分析形式
- 使用一个栈保存文法符号，一个输入缓冲区存放将要进行分析的剩余符号
- 初始栈： $　　初始输入 *w*$
- 对输入串的一次从左到右的扫描过程中，语法分析器将零个或多个输入符号移到栈的顶端，直到它可以**对栈顶的一个文法符号串**进行**归约**为止。它将归约为某个产生式的头。不断重复这个循环，直到它检测到一个语法错误，或者栈中包含了开始符号且输入缓冲区为空。
- 分析成功时： 栈 $ S,　输入 $

自顶向下的非递归预测分析中栈是如何工作的？

移入-归约分析技术

- 使用一个栈来保存归约/扫描移入的文法符号
- 栈中符号（从底向上）和待扫描的符号组成了一个最右句型
- 开始时刻：栈中只包含$，而输入为w$
- 成功结束时刻：栈中$S，而输入$
- 在分析过程中，不断地移入符号，并在识别到句型时进行归约

主要分析动作

- 移入：将下一个输入符号移动到栈顶
- 归约：将句柄归约为相应的非终结符号
  - **句柄总是在栈顶**
  - 具体操作时弹出句柄，压入被归约到的非终结符号
- 接受：宣布分析过程成功完成
- 报错：发现语法错误，调用错误恢复子程序

归约分析过程的例子

栈	输入	动作
$	$\mathbf{id}_1 * \mathbf{id}_2$ $	移入
$ $\mathbf{id}_1$	$* \mathbf{id}_2$ $	按照 $F \to \mathbf{id}$ 归约
$ F	$* \mathbf{id}_2$ $	按照 $T \to F$ 归约
$ T	$* \mathbf{id}_2$ $	移入
$ $T *$	$\mathbf{id}_2$ $	移入
$ $T * \mathbf{id}_2$	$	按照 $F \to \mathbf{id}$ 归约
$ $T * F$	$	按照 $T \to T * F$ 归约
$ T	$	按照 $E \to T$ 归约
$ E	$	accept

图 4-28　一个移入－归约语法分析器在处理输入 $\mathbf{id}_1 * \mathbf{id}_2$ 时经历的格局

移进归约语法分析框架（续）

- 可行性分析：句柄总是出现在栈的顶端，绝不会出现在栈的中间

1）$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Az \underset{rm}{\Rightarrow} \alpha\beta Byz \underset{rm}{\Rightarrow} \alpha\beta\gamma yz$

2）$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha BxAz \underset{rm}{\Rightarrow} \alpha Bxyz \underset{rm}{\Rightarrow} \alpha\gamma xyz$

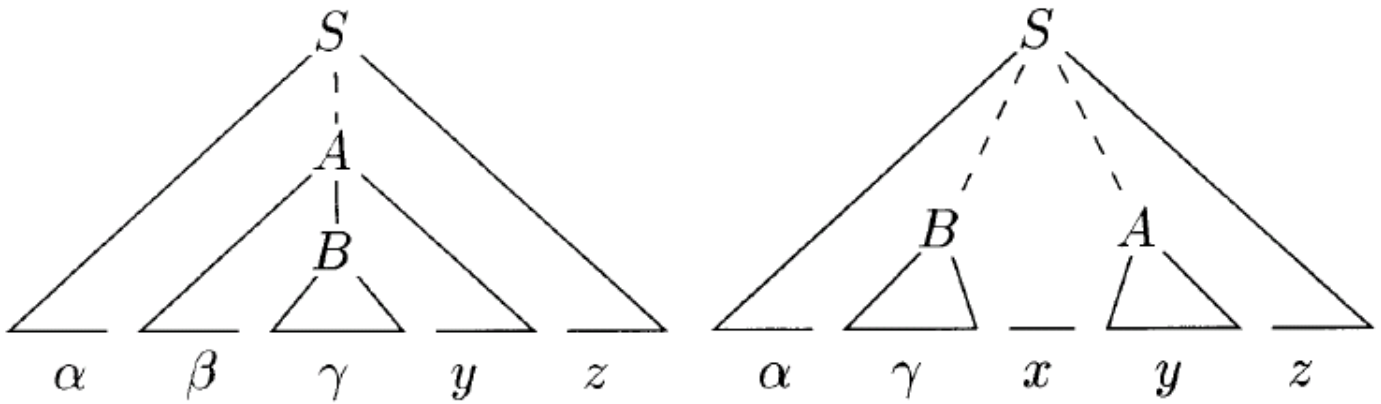
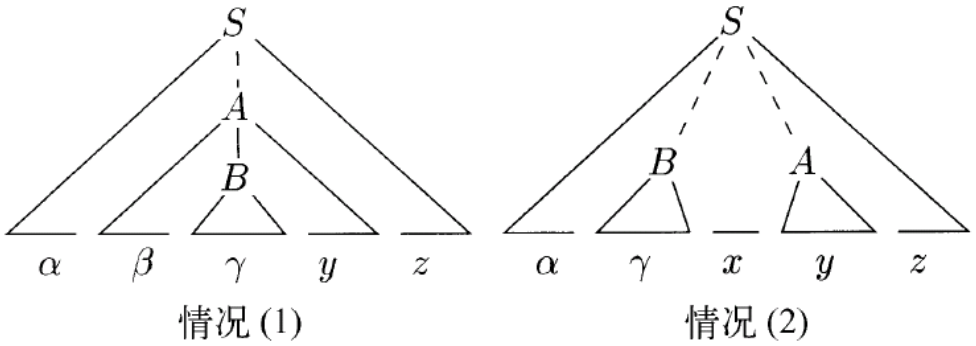


图 4-29　一个最右推导中两个连续步骤的两种情况

移进归约语法分析框架（续）



1）$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha Az \underset{rm}{\Rightarrow} \alpha\beta Byz \underset{rm}{\Rightarrow} \alpha\beta\gamma yz$

2）$S \underset{rm}{\overset{*}{\Rightarrow}} \alpha BxAz \underset{rm}{\Rightarrow} \alpha Bxyz \underset{rm}{\Rightarrow} \alpha\gamma xyz$

图 4-29　一个最右推导中两个连续步骤的两种情况

STACK	INPUT	STACK	INPUT
$\$\alpha\beta\gamma$	$yz\$$	$\$\alpha\gamma$	$xyz\$$
$\$\alpha\beta B$	$yz\$$	$\$\alpha Bxy$	$z\$$
$\$\alpha\beta By$	$z\$$		

- 语法分析器进行一次归约以后，都必须接着移入零个或多个符号才能在栈顶找到下一个句柄
- 不需要去栈中间去寻找句柄

移进/归约冲突

- 移入-归约技术并不能处理所有上下文无关文法
- 某些上下文无关文法（比如二义性文法）
  - 移入/归约冲突：栈中的内容和接下来的k个输入符号，都不能确定进行移入还是归约操作(不能确定是否是句柄)
  - 归约/归约冲突：存在多个可能的归约到不同非终结符号的归约(不能确定句柄归约到那个非终结符号)

移进/归约冲突

stmt → **if** *expr* **then** *stmt*

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

| **other**

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

规约为**stmt**？
还是移入**else**？

归约/归约冲突

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

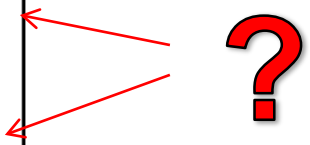


图 4-30　有关过程调用和数组引用的产生式

栈	输入
… **id** (**id**	, **id**) …

解决冲突问题

- 让词法分析区分id和procid
- 用[]表示数组，用()表示函数
- ……
- 但是并不能完全解决冲突问题
- 有一类文法可以解决句柄查找及归约唯一性的问题

自底向上分析-回顾

- 从输入串出发构造一个反向最右推导序列（归约）
- 每一步是对**句柄**进行归约 -- 寻找句柄
- 一种移进-归约的分析框架
  - 一个栈存放文法符号，一个输入缓冲区
  - 移进 or 归约
  - 总是能够在栈的顶端找到句柄
  - 通过证明是可行的
- 有时候会有冲突，是移进还是归约？ 怎么归约？

LR语法分析技术

- LR(k)的语法分析概念
  - L表示最左扫描，R表示反向构造出最右推导
  - k表示最多向前看k个符号
- 当k的数量增大时，相应的语法分析器的规模急剧增大
  - K=2时，程序设计语言的语法分析器的规模通常非常庞大
  - 当k=0、1时已经可以解决很多语法分析问题，因此具有实践意义
  - 因此，我们只考虑k<=1的情况

LR语法分析器的优点

- 表格驱动
  - 虽然手工构造表格工作量大，但表格可以自动生成
- 对于几乎所有的程序设计语言，只要写出上下文无关文法，就能够构造出识别该构造的LR语法分析器
- 最通用的无回溯移入-归约分析技术，且和其它技术一样高效
- 可以尽早检测到错误
- 能分析的文法集合是LL(k)文法的超集

LR分析相关概念

- 移入-归约语法分析器如何知道何时该移入、何时该归约呢？
  - LR语法分析器试图用一些**状态**来表明我们在移进归约语法分析过程中所处的位置，从而做出移入-归约决定
- 项的意义
  - 指明在语法分析过程中的给定点上，我们已经看到了一个产生式的哪些部分。或者说，如果我们想用这个产生式进行归约，还需要看到哪些文法符号
- 项的集合（项集）对应于一个状态

LR(0)项

- 文法的一个产生式加上在其产生式体中某处的一个点
  - **A →.XYZ，A → X.YZ，A → XY.Z，A → XYZ.**
  - 注意：**A → ε**只对应一个项**A →.**
- 直观含义
  - 项**A → α.β**表示已经扫描/归约到了**α**，并期望接下来的输入中经过扫描/归约得到**β**，然后把**αβ**归约到**A**
  - 如果**β**为空，表示我们可以把**α**归约为**A**
- 项也可以用一对整数表示
  - **(i,j)**表示第**i**条规则，点位于右部第**j**个位置

规范LR(0)项集族

- **规范LR(0)项集族提供构建LR(0)自动机的基础**
  - **LR(0)自动机可用于做出语法分析决定**
  - **LR(0)自动机中每个状态代表了LR(0)项集族中的一个项集**

以项为基础构造自动机

- 构造以项为状态的自动机
  - 开始状态S’ →.S
  - 转换
    - A → α.Bβ到B →. γ有一个ε转换
    - 从A → α.Xβ到A → αX.β有一个X转换
  - 接受状态：A → α.，即点在最后的项

规范LR(0)项集族的构造

- 三个概念
  - 增广文法
  - 项集闭包：**CLOSURE**
  - **GOTO**

规范LR(0)项集族的构造

- 增广文法
  - G的增广文法G’是在G中增加新开始符号S’，并加入产生式S’→S而得到的
  - G’和G接受相同的语言，且按照S’→ S进行归约实际上就表示已经将输入符号串归约成为开始符号
  - 引入的目的是告诉语法分析器何时宣布接受输入符号串，即用S’→S进行归约时，表明分析结束。

规范LR(0)项集族的构造

- **构造过程中用到的子函数**
  - **CLOSURE(I)：I的项集闭包**
    - **对应于DFA化算法的 ε-CLOSURE**
  - **GOTO(I,X)：I的X后继**
    - **对应于DFA化算法的MOVE(I,X)**

CLOSURE(I)的构造算法

- **如果I是文法G的一个项集，那么CLOSURE(I)就是根据下列规则从I构造得到的项集**
  - **将I中的各个项加入到CLOSURE(I)中**
  - **如果A → α.Bβ在CLOSURE(I)中，那么对B的任意产生式B → γ，将B →.γ加到CLOSURE(I)中**
  - **不断重复第二步，直到收敛**
- **第二步的意义**
  - **项A → α.Bβ表示期望在接下来的输入中归约到B**
  - **显然，要归约到B，首先要扫描归约到B的某个产生式的右部**
  - **因此对每个产生式B → γ，加入B →.γ**
    - **表示它期望能够扫描归约到γ**

项集闭包构造的例子

- **增广文法：**
  - **E’ → E**
  - **E → E+T | T**
  - **T → T*F | F**
  - **F →(E) | id**

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for (J中的每个项 A → α·Bβ)
                        for (G 的每个产生式 B → γ)
                                if (项 B → ·γ 不在J中)
                                        将 B → ·γ 加入J中;
        until 在某一轮中没有新的项被加入到J中;
        return J;
}
```

- **项集 I={[E’ →.E]} 的闭包**
  - **[E’ →.E]在闭包中**
  - **[E →.E+T]，[E →.T]在闭包中**
  - **[T →.T*F]，[T →.F]在闭包中**
  - **[F →.(E)]，[F →.id]在闭包中**

LR(0)项集中的内核项和非内核项

- 内核项：初始项S’→.S、以及所有点不在最左边的项
- 非内核项：除了S’→.S之外、点在最左边的项

- 由于表可能很庞大，实现算法时可以考虑只保存相应的非终结符号；甚至可以只保存内核项，而在要使用非内核项时调用CLOSURE函数重新计算

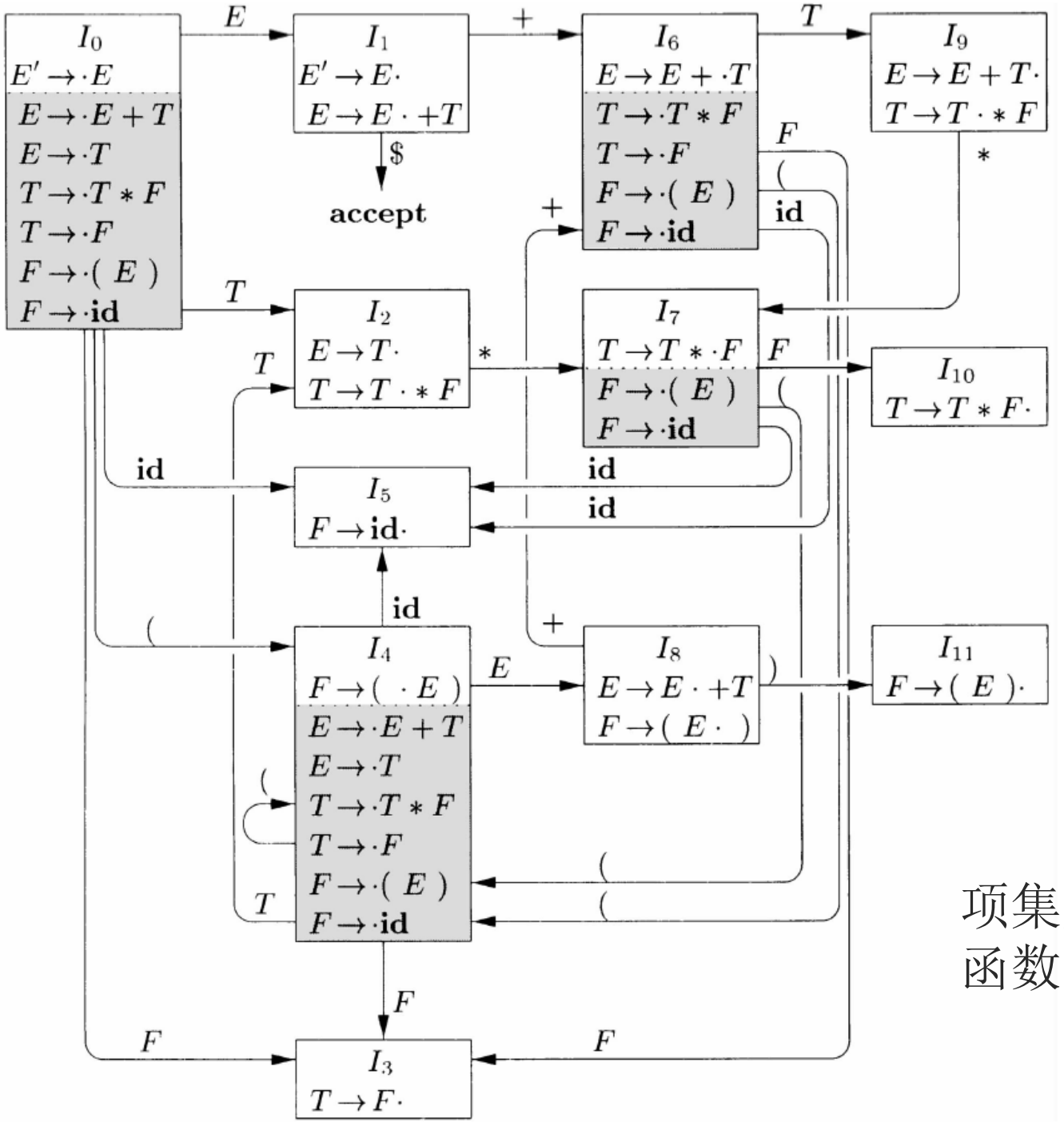
GOTO函数

- I是一个项集，X是一个文法符号，GOTO(I,X)定义为I中所有形如的项[A → α·Xβ]所对应的项[A → αX·β]的集合的闭包
  - 根据项的历史-期望的含义，GOTO(I,X)表示读取输入中的X或者归约到一个X之后的情况
  - GOTO(I,X)定义了LR(0)自动机中状态I在X之上的转换
- 例如：
  - I={[E' → E.], [E → E.+T]}
  - GOTO(I,+)计算如下
    - I中只有一个项的点后面跟着+，对应的项为[E → E+.T]
    - CLOSURE({[E → E+.T]}) = {[E → E+.T]，[T →.T*F], [T→.F]，[F →.(E)]，[F →.id]}

求LR(0)项集规范族的算法

- **为文法G构造LR(0)项集规范族C**
  - **步骤一：构造增广文法G’: S’ →S ……**
  - **步骤二：构造I0=CLOSURE(S’ →·S)，I0是C的初始项集**
  - **步骤三：对C中的每个项集I_i及每个文法符号X_j，将GOTO(I_i,X_j)加入到C中**
  - **重复步骤三，直到没有新的项集可以加入**

```
void items(G') {
    C = CLOSURE({[S' → ·S]});
    repeat
        for (C中的每个项集I)
            for (每个文法符号X)
                if (GOTO(I, X)非空且不在C中)
                    将GOTO(I, X)加入C中;
    until 在某一轮中没有新的项集被加入到C中;
}
```



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

项集规范族和GOTO函数的例子

LR(0)自动机的构造

- 构造方法
  - 规范LR(0)项集族中的项集可以作为LR(0)自动机的状态
  - GOTO(I,X)=J，则从I到J有一个标号为X的转换
  - 初始状态为CLOSURE({S’ →.S})对应的项集
  - 接受状态：包含形如A → α.的项集对应的状态

LR(0)自动机的作用

- 根据文法构造出LR(0)自动机，通过自动机的运行进行语法分析

- 假设文法符号串γ使LR(0)自动机从开始状态0运行到某个状态j，LR(0)自动机按照如下方式决定移入或归约
  - 如果下一个输入符号为a，且状态j上有a的转换，就移入a
  - 否则就归约，状态j中的项会告诉我们使用那个产生式进行归约

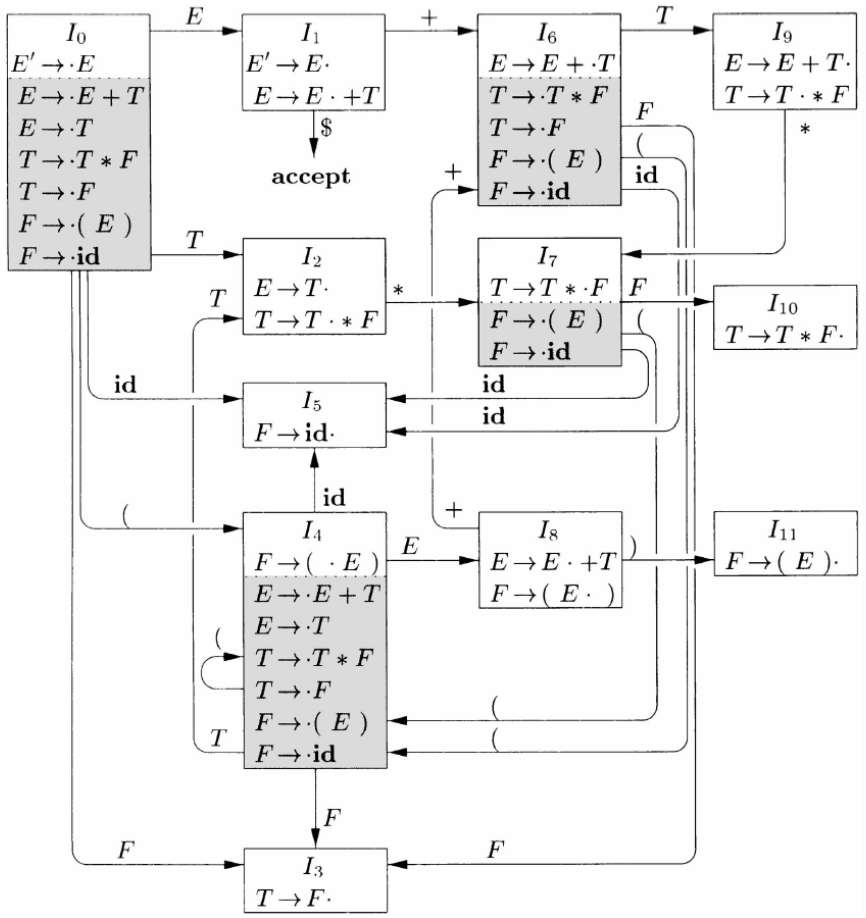
LR(0)自动机的作用（1）

- 假设文法符号串γ使LR(0)自动机从开始状态运行到状态（项集）j
  - 如果j中有一个形如A → α.的项，那么
    - 在γ之后添加一些终结符号可以得到一个最右句型
    - α是γ的后缀，且A → α是这个句型的句柄
    - 表示可能找到了当前最右句型的句柄
  - 如果j中存在一个项B → α.Xβ，那么
    - 在γ之后添加Xβ，然后再添加一个终结符号串可得到一个最右句型
    - 在这个句型中B → αXβ是句柄
    - 此时表示还没有找到句柄，需要移入

LR(0)自动机的作用（2）

- LR(0)自动机的使用
  - 移入-归约时，LR(0)自动机被用于识别句型
  - 已经归约/移入得到的文法符号序列对应于LR(0)自动机的一条路径
  - 如果路径到达接受状态，表明栈上端的某个符号串可能是句柄
- 不需要每次用归约/移入得到的串来运行LR(0)自动机
  - 路径被放到栈中；且文法符号可以省略，因为由LR(0)状态可以确定相应文法符号
  - 在移入后，根据原来的栈顶状态即可知道新的状态
  - 在归约时，根据归约产生式的右部长度弹出相应状态，仍然可以根据此时的栈顶状态计算得到新状态

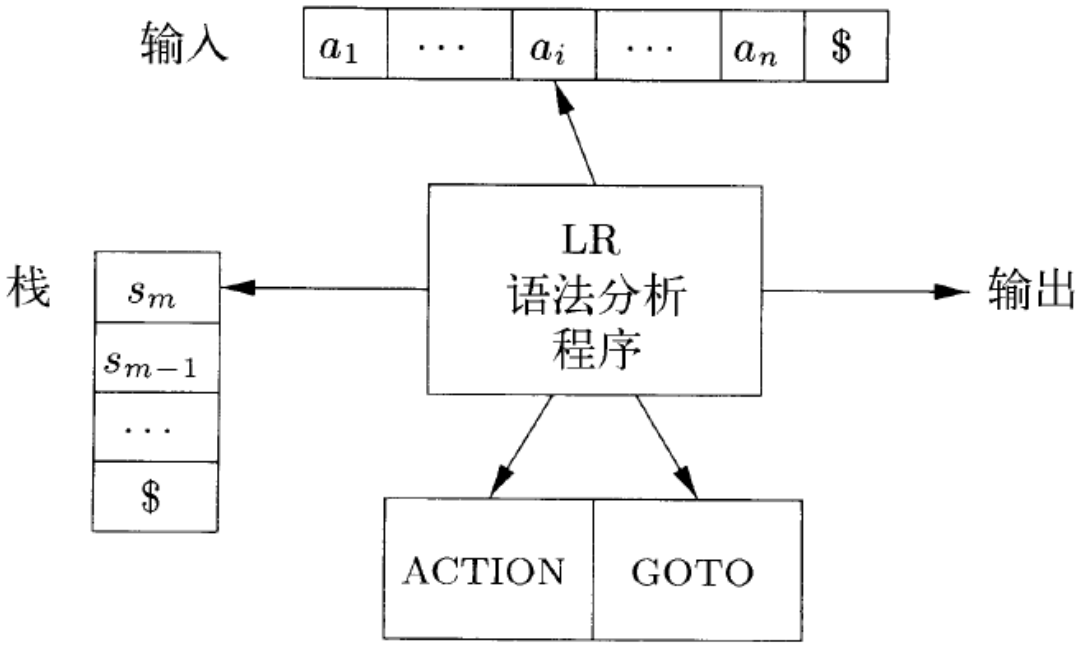
LR(0)自动机作用示例：分析id*id



行号	栈	符号	输入	动作
(1)	0	\$	**id** $*$ **id** \$	移入到 5
(2)	0 5	\$ **id**	$*$ **id** \$	按照$F \to$ **id** 归约
(3)	0 3	\$ F	$*$ **id** \$	按照 $T \to F$ 归约
(4)	0 2	\$ T	$*$ **id** \$	移入到 7
(5)	0 2 7	\$ $T *$	**id** \$	移入到 5
(6)	0 2 7 5	\$ $T *$ **id**	\$	按照$F \to$ **id**归约
(7)	0 2 7 10	\$ $T * F$	\$	按照$T \to T * F$归约
(8)	0 2	\$ T	\$	按照$E \to T$ 归约
(9)	0 1	\$ E	\$	接受

- **根据LR(0)自动机状态和符号的对应关系，可以得到符号栏中的符号串**
  - **路径被放到栈中；且文法符号可以省略，因为由LR(0)状态可以确定相应文法符号**

LR语法分析器的结构



- 所有的分析器都使用相同的驱动程序
- 分析表随文法以及LR分析技术的不同而不同
- 栈中存放的是状态序列；可以由状态序列求出符号序列
- 分析程序根据栈顶状态、当前输入，通过分析表确定语法分析动作

LR语法分析表的结构

- 两个部分：动作ACTION，转换GOTO
- ACTION有两个参数：状态i、终结符号a
  - 移入j：j是一个状态。把j压入栈
  - 归约$A \rightarrow \beta$：把栈顶的β归约为A
  - 接受：接受输入、完成分析
  - 报错：在输入中发现语法错误
- 状态集上的GOTO函数
  - 如果$GOTO[I_i,A]=I_j$，那么GOTO[i,A] = j

LR语法分析器的格局

- LR语法分析器的格局包含了栈中内容和余下输入

  $(s_0s_1\ldots s_m,\ a_ia_{i+1}\ldots a_n\$)$

  - 第一个分量是栈中的内容（右侧是栈顶）
  - 第二个分量是余下输入
- LR语法分析器的每一个状态都对应一个文法符号（s_0除外）
  - 如果进入状态s的边的标号为X，那么s就对应于X
- 令X_i为s_i对应的符号，那么
  - $X_1X_2\ldots X_m\ a_ia_{i+1}\ldots a_n$对应于一个最右句型

LR语法分析器的行为

- 对于格局($s_0s_1\ldots s_m$, $a_ia_{i+1}\ldots a_n$\$)，LR语法分析器查询条目ACTION[s_m,a_i]确定相应动作
  - 移入s：执行移入动作，将s（对应a）移入栈中，新格局($s_0s_1\ldots s_ms$，$a_{i+1}\ldots a_n$\$)
  - 归约$A \rightarrow \beta$，将栈顶的β归约为A，压入状态s

    ($s_0s_1\ldots s_{m-r}s$，$a_ia_{i+1}\ldots a_n$\$)，

    其中r是β的长度，s=GOTO[s_{m-r},A]
  - 接受：语法分析过程完成
  - 报错：发现语法错误，调用错误恢复例程

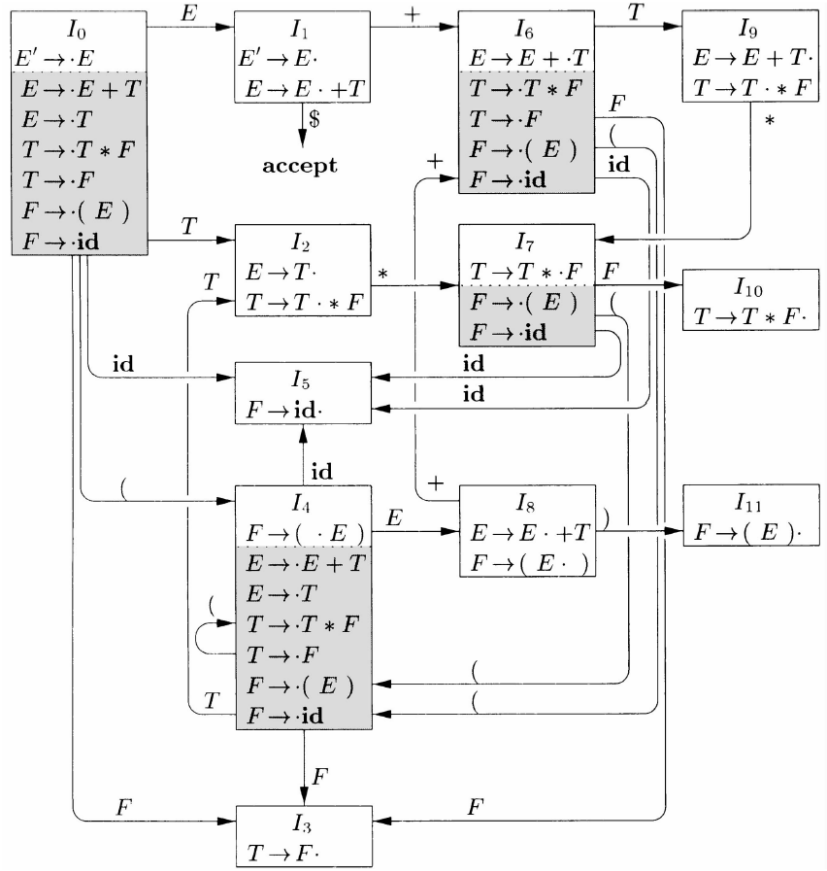
LR语法分析算法

- 输入：文法G的LR语法分析表，输入串ω
- 输出：如果ω在L(G)中，输出最左归约步骤（最右推导的反向过程），否则输出错误指示
- 算法

```
令 a 为 w$ 的第一个符号;
while(1) { /* 永远重复 */
    令 s 是栈顶的状态;
    if ( ACTION[s, a] = 移入 t ) {
        将 t 压入栈中;
        令 a 为下一个输入符号;
    } else if ( ACTION[s, a] = 归约 A → β ) {
        从栈中弹出 |β| 个符号;
        令 t 为当前的栈顶状态;
        将 GOTO[t, A] 压入栈中;
        输出产生式 A → β;
    } else if ( ACTION[s, a] = 接受 ) break; /* 语法分析完成 */
    else 调用错误恢复例程;
}
```

LR分析表的例子

- 文法：　(1)E → E+T　　(2)E → T
  (3)T → T*F　　(4)T → F
  (5)F →(E)　　(6)F → id



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

LR分析

- 输入：**id*id+id**

	栈	符号	输入	动作
(1)	0		id * id + id $	移入
(2)	0 5	id	* id + id $	根据 $F \to$ id 归约
(3)	0 3	F	* id + id $	根据 $T \to F$ 归约
(4)	0 2	T	* id + id $	移入
(5)	0 2 7	T *	id + id $	移入
(6)	0 2 7 5	T * id	+ id $	根据$F \to$ id 归约
(7)	0 2 7 10	$T * F$	+ id $	根据$T \to T * F$归约
(8)	0 2	T	+ id $	根据$E \to T$ 归约
(9)	0 1	E	+ id $	移入
(10)	0 1 6	E +	id $	移入
(11)	0 1 6 5	E + id	$	根据 $F \to$ id 归约
(12)	0 1 6 3	$E + F$	$	根据 $T \to F$ 归约
(13)	0 1 6 9	$E + T$	$	根据 $E \to E + T$归约
(14)	0 1	E	$	接受

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

SLR分析表构造

- 以LR(0)项和LR(0)自动机为基础，基于文法*G*的规范项集族*C*和GOTO函数就可以构造出SLR的语法分析表

SLR分析表构造算法

- 输入：一个增广文法*G'*
- 输出：*G'*的SLR语法分析表函数ACTION和GOTO
- 方法：

1）构造G' LR(0)项集规范族C={I0, I1,… In,}

2）对于状态i中的项：

- **[A → α·aβ]且GOTO[I_i ,a]=I_j, 那么ACTION[i,a]=移入j**
- **[A → α·]，那么对于FOLLOW(A)中的所有a， ACTION[i,a]=归约A → α**
- [S' →S·], 那么ACTION[i,$]=接受

3）状态i对于非终结符号A的转换规则：

如果 GOTO[I_i ,A]=I_j，那么GOTO[i ,A]=j

4) 规则2)和3)没有定义的所有条目设置为“报错”

SLR(1)

- 根据上一算法构造的语法分析表，若表中各位置没有多个条目，则称为文法G的**SLR(1)分析表**。使用该分析表的分析器，称为G的**SLR(1)语法分析器**。G称为**SLR(1)文法**

- 若表中某位置存在多个条目（多个可选的动作，冲突），则该文法是非SLR(1)文法

构造示例

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow (E)$

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

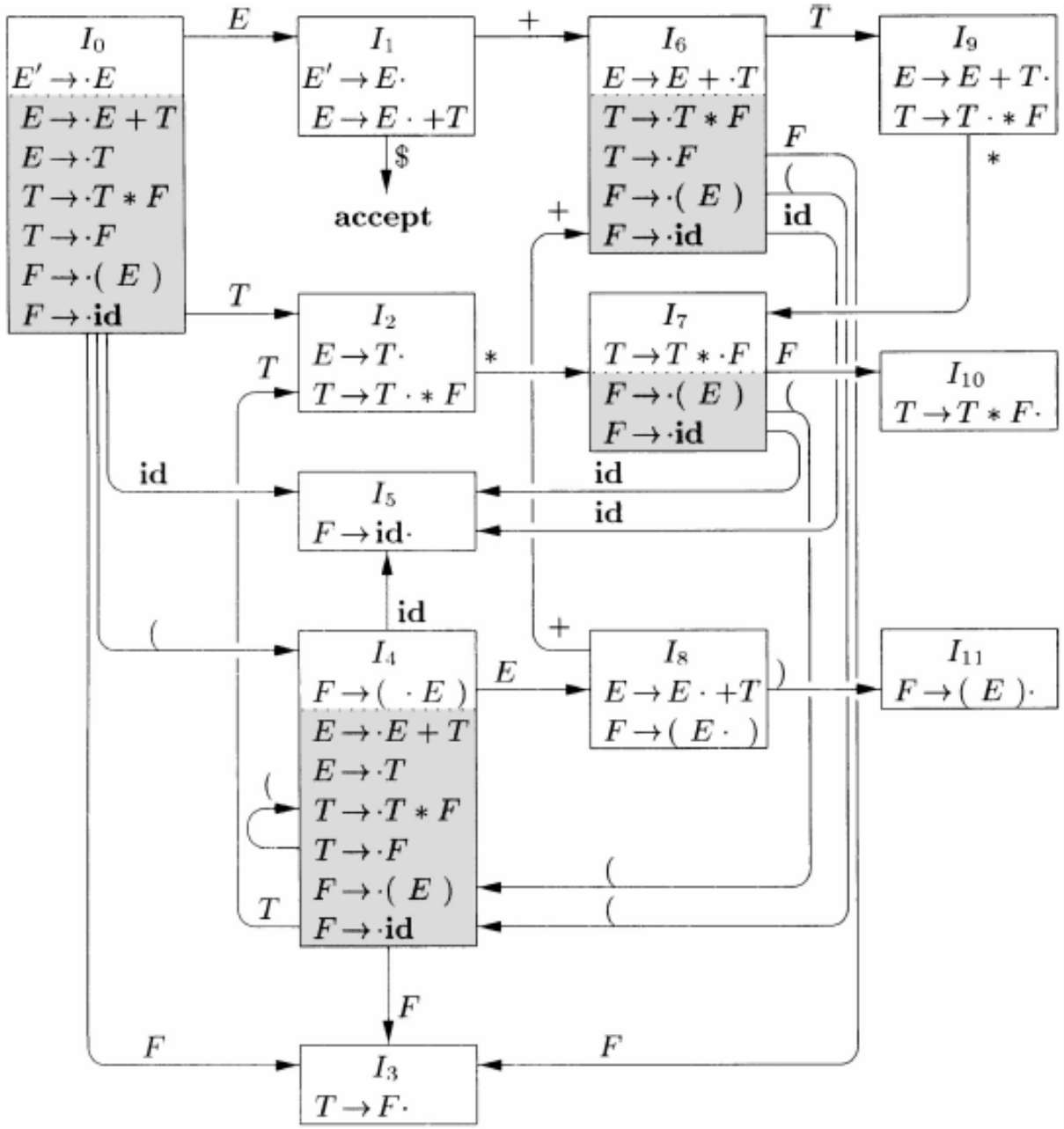


图 4-31　表达式文法(4.1)的 LR(0) 自动机

SLR的原理：可行前缀（1）

- LR(0)自动机刻画了可能出现在语法分析栈中的文法符号串 (能够识别可行前缀)
  - 直接把项当作状态，可以构造得到一个NFA
  - 然后确定化得到DFA就是LR(0)自动机
- 可行前缀
  - 可以出现在移入-归约语法分析器栈中的最右句型的前缀
  - 定义：某个最右句型的前缀，且没有越过该句型的句柄的右端

$$E \underset{rm}{\overset{*}{\Rightarrow}} F * \mathbf{id} \underset{rm}{\Rightarrow} (E) * \mathbf{id}$$

可能的前缀：
(，(E，(E)，(E)*

SLR的原理：可行前缀（2）

- 有效项
  - 如果存在 $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$，那么我们说项

    $A \rightarrow \beta_1 . \beta_2$对$\alpha\beta_1$有效
- 有效项的意义
  - 帮助我们决定是进行规约还是移入
  - 如果我们知道$A \rightarrow \beta_1 . \beta_2$对$\alpha\beta_1$有效
    1. 当β_2不等于空，表示句柄尚未出现在栈中，应该移入或者等待归约
    2. 如果β_2等于空，表示句柄出现在栈中，应该归约

SLR的原理：可行前缀（3）

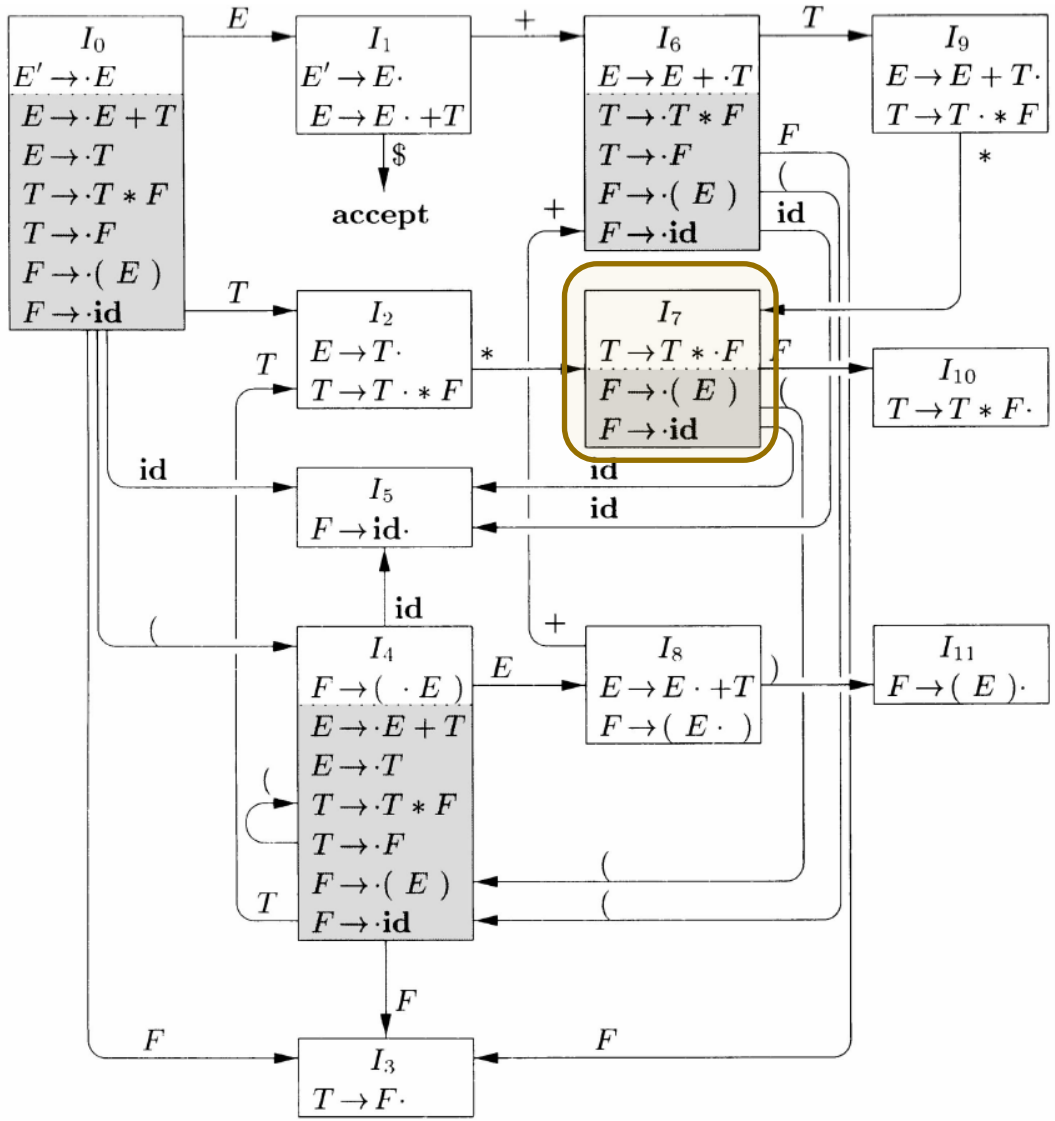
- 如果某个时刻存在两个有效项要求执行不同的动作，那么就应该设法解决冲突
  - 冲突实际上表示可行前缀可能是两个最右句型的前缀，第一个包含了句柄，而另一个尚未包含句柄
  - SLR解决冲突的思想：假如要按照A → β进行归约，那么得到的新句型中A后面跟随下一个输入符号。因此只有当下一个输入在FOLLOW(A)中时才可以归约

SLR的原理：可行前缀（4）

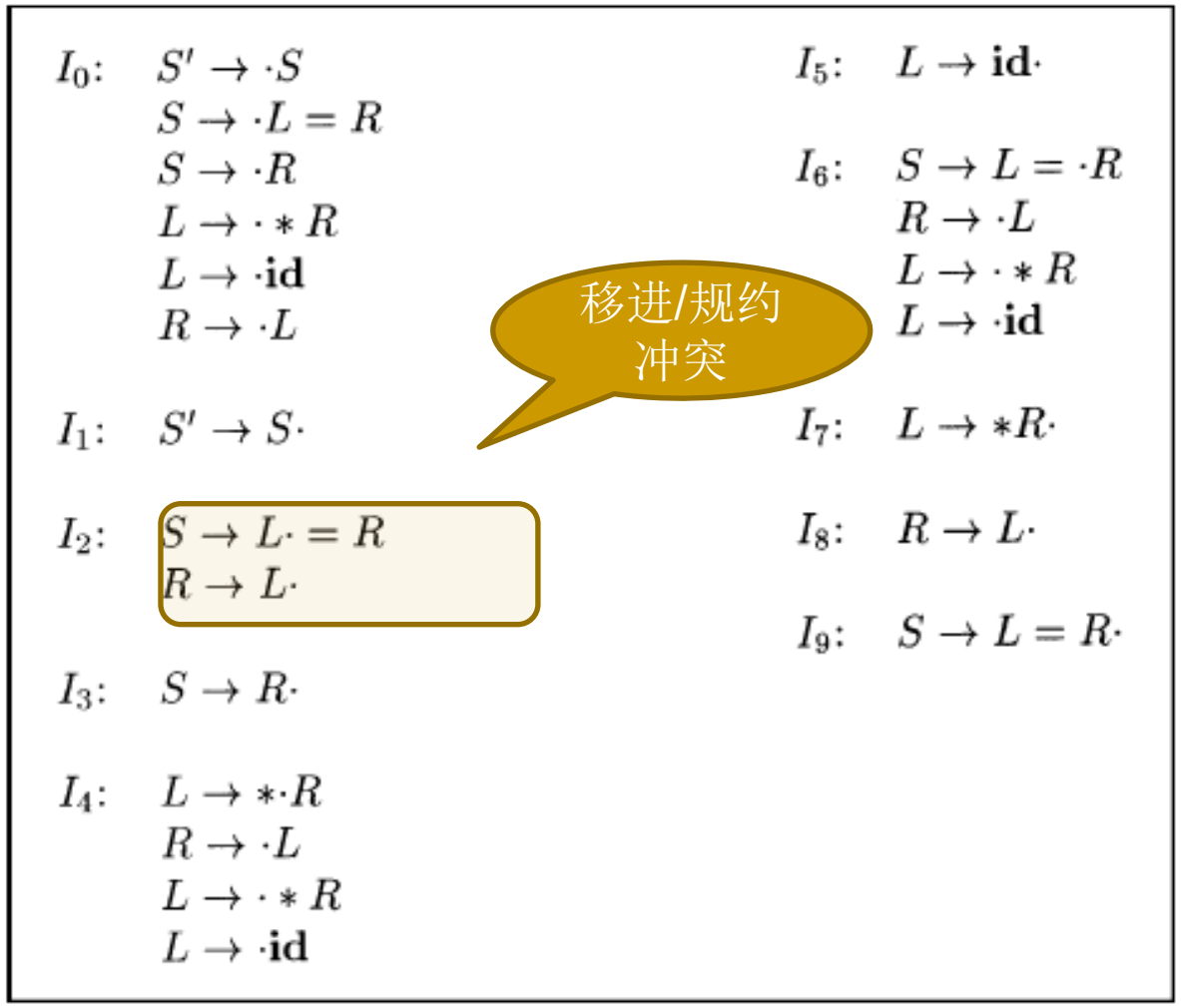
- 如果在文法G的LR(0)自动机中，从初始状态出发，沿着标号为γ的路径到达一个状态，那么这个状态对应的项集就是γ的有效项集
- 回顾确定分析动作的方法，就可以知道我们实际上是按照有效项来确定的
  - 为了避免冲突，归约时要求下一个输入符号在FOLLOW(A)中

可行前缀/有效项的例子

- 可行前缀E+T*
- 对应的LR(0)项
  - T → T*.F
  - F →.(E)
  - F →.id
- 对应的最右推导
  - E’➔E➔E+T➔E+T*F
  - E’➔E➔E+T➔E+T*F➔E+T*(E)
  - E’➔E➔E+T➔E+T*F➔E+T*id



示例二



$$S \to L = R \mid R$$

$$L \to * R \mid \mathbf{id}$$

$$R \to L$$

- 状态2，输入符号为“=”时
  - 移进
  - Follow(R)=？，规约

SLR语法分析器的弱点（1）

- SLR技术解决冲突的方法
  - 项集中包含[A → α.]时，按照A → α进行归约的条件是下一个输入符号a可以在某个句型中跟在A之后
    - **如果对于a还有其它的移入/归约操作，则出现冲突**

$$S \rightarrow L = R \mid R$$
$$L \rightarrow * R \mid \mathbf{id}$$
$$R \rightarrow L$$

$$I_2: \quad S \rightarrow L \cdot = R$$
$$R \rightarrow L \cdot$$

  - 假设此时栈中的符号串为βα
    - **如果βAa不可能是某个最右句型的前缀，那么即使a在某个句型中跟在A之后，仍然不应该按照A → α归约**
    - **进行归约的条件更加严格可以降低冲突的可能性**

SLR语法分析器的弱点（2）

- [A → α.]出现在项集中的条件：
  - 首先[A →. α]出现在某个项集中，然后逐步移入/归约到α中的符号，点不断后移，到达末端
  - 而[A →. α]出现的条件是B → β.Aγ出现在项中
    - 期望首先按照A → α归约，然后将B → β.Aγ中的点后移到A之后
  - 显然，在按照A → α归约时要求下一个输入符号是γ的第一个符号
  - 但是从LR(0)项集中不能确定这个信息

习题

- 给出如下文法的语法分析表，判断该文法是否SLR文法。

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

更强大的LR语法分析器

- 规范LR方法。充分利用向前看符号。这种方法是用了一个更大的项集，称为LR(1)项集

- 向前看LR，又称LALR

LR(1)项

- 在SLR中，如果项集I_i中包含项[A → α·]，且当前符号a在FOLLOW(A)中，那么就可以按照A→α进行归约
- 有时，在任何最右句型中a都不可能跟在βA之后，这时输入为a不能按照A → α进行归约
- FOLLOW集提供的可归约条件过松

$$S \rightarrow L = R \mid R$$
$$L \rightarrow * R \mid \mathbf{id}$$
$$R \rightarrow L$$

$$I_2: \quad S \rightarrow L \cdot = R$$
$$R \rightarrow L \cdot$$

- 每个状态能否明确指出哪些输入符号可以跟在句柄α后面，从而使句柄α可能被归约为A

规范LR(1)项

- 在状态中包含更多的信息
  - 让LR分析器的每个状态精确指明哪些输入符号可以跟在句柄α后面，使α可能被规约为A
- 让项包含第二个分量
  - 形式如[A → α·β,a]，其中A → αβ是一条产生式，a是一个终结符号或者$符
  - a是向前看符号，长度为1

规范LR(1)项

- 项[A → α·β,a]
  - 如果β≠ε时， a没有任何作用
  - a的意义是若栈顶状态中存在LR(1)项[A → α·,a]， 在输入符号为a时才会进行归约
  - **a的集合总是FOLLOW(A)的子集**

- **a指出能够进行归约的准确判断**

向前看符号与确定归约有关

规范LR(1)项

- 从可行前缀和有效项的角度看
- LR(1)项[A → α·β,a]对于一个可行前缀δα有效的条件是存在一个推导$S \underset{rm}{\Rightarrow}{}^{*} \delta A\omega \underset{rm}{\Rightarrow} \delta\alpha\beta\omega$，其中a∈First(ω)。当ω为ε，a=$

$$S \rightarrow L = R \mid R$$
$$L \rightarrow * R \mid \mathbf{id}$$
$$R \rightarrow L$$

$$I_2: \quad S \rightarrow L\cdot = R$$
$$R \rightarrow L\cdot$$

LR(0)：如果存在 $S' \underset{rm}{\overset{*}{\Rightarrow}} \alpha A w \underset{rm}{\Rightarrow} \alpha\beta_1\beta_2 w$ **，那么我们说项A → $\beta_1 . \beta_2$对$\alpha\beta_1$有效**

LR(1)项集的构造

- 过程类似于LR(0)项集构造
- 多了一个**向前看符号**分量的计算
- CLOSURE
- GOTO

LR(1)项集闭包CLOSURE

- CLOSURE(I)=I U {[B → .η,**b**] | [A → α·Bβ,**a**] ∈CLOSURE(I)，b∈First(βa)，且B →η为文法规则}

  为什么b必须在First(βa)中？

  - 对于可行前缀γ的有效项[A → α·Bβ,a]，存在最右推导 $S \underset{rm}{\overset{*}{\Rightarrow}} \delta Aax \underset{rm}{\Rightarrow} \delta\alpha B\beta ax$ ，其中γ = δα
  - 假设βax推导出终结符号串bγ，那么对于某个形如B → η的产生式，有推导 $S \underset{rm}{\overset{*}{\Rightarrow}} \gamma Bby \underset{rm}{\Rightarrow} \gamma\eta by$ ， 因此， {[B → .η, b]是γ的有效项
  - **b是First(βa)中的终结符号**
    - **b可能是从β推导出的第一个终结符号**
    - **也可能β推导出了ε，b就是a**

LR(1)项集闭包CLOSURE

SetOfItems CLOSURE(I) {
    repeat
        for (I 中的每个项 $[A \to \alpha \cdot B\beta, a]$)
            for (G' 中的每个产生式 $B \to \gamma$)
                for (FIRST(βa)中的每个终结符号 b)
                    将 $[B \to \cdot\gamma, b]$ 加入到集合 I 中;
    until 不能向 I 中加入更多的项;
    return I;
}

LR(1)项集GOTO函数

SetOfItems GOTO(I, X) {
    将 J 初始化为空集；
    for (I 中的每个项 $[A \to \alpha \cdot X \beta, a]$)
        将项 $[A \to \alpha X \cdot \beta, a]$ 加入到集合 J 中；
    return CLOSURE(J);
}

LR(1)项集规范族构造算法

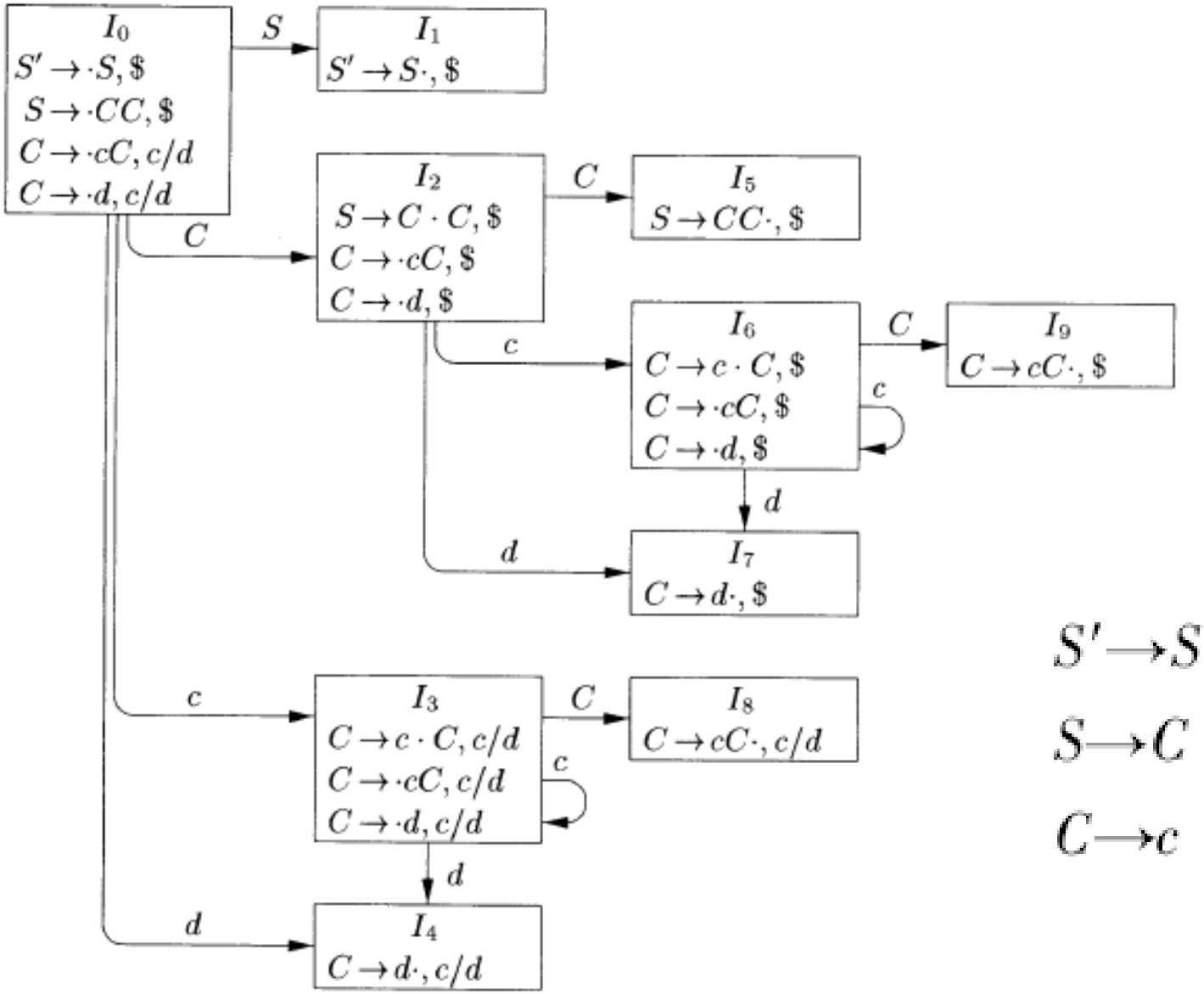
```
SetOfItems CLOSURE(I) {
    repeat
        for ( I中的每个项 [A → α·Bβ, a] )
            for ( G′中的每个产生式 B → γ )
                for ( FIRST(βa)中的每个终结符号 b )
                    将 [B → ·γ, b] 加入到集合 I中;
    until不能向 I 中加入更多的项 ;
    return I;
}

SetOfItems GOTO(I, X) {
    将 J 初始化为空集 ;
    for ( I 中的每个项 [A → α·Xβ, a])
        将项 [A → αX·β, a]加入到集合 J 中;
    return CLOSURE(J);
}

void items(G′) {
    将 C 初始化为CLOSURE({[S′ → ·S, $]});
    repeat
        for ( C 中的每个项集 I )
            for (每个文法符号 X )
                if ( GOTO(I, X) 非空且不在 C 中)
                    将 GOTO(I, X) 加入 C 中;
    until 不再有新的项集加入到 C 中 ;
}
```

图 4-40　为文法 G' 构造 LR(1)项集族的算法

LR(1)规范项集族构造示例



$$S' \to S$$

$$S \to C\ C$$

$$C \to c\ C \mid d$$

规范LR(1)语法分析表

- 输入：一个增广文法G'
- 输出：G'的规范LR语法分析表函数ACTION和GOTO
- 方法：

1）构造G' LR(1)项集规范族$C=\{I_0, I_1, \ldots I_n\}$

2）对于状态i中的项：

- $[A \rightarrow \alpha \cdot a\beta, b]$且$GOTO[I_i, a]=I_j$，那么$ACTION[i,a]$=移入$j$
- $[A \rightarrow \alpha \cdot, a]$，且$A \neq S'$， $ACTION[i,a]$=归约$A \rightarrow \alpha$
- $[S' \rightarrow S \cdot, \$]$，那么$ACTION[i,\$]$=接受

3）状态i对于非终结符号A的转换规则：如果

$GOTO[I_i, A]=I_j$，那么$GOTO[i, A]=j$

4）规则2)和3)没有定义的所有条目设置为“报错”

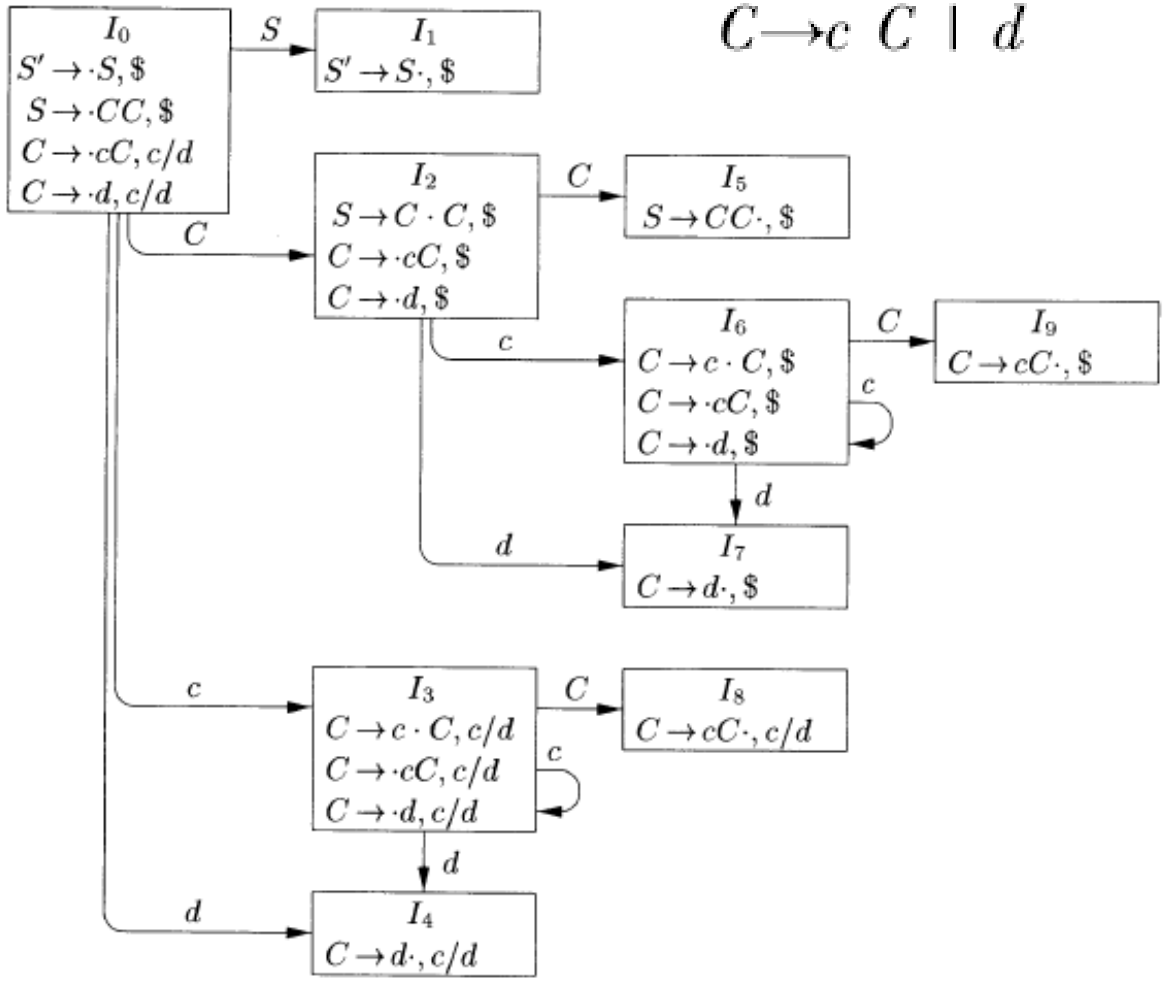
语法分析表示例

$$S' \rightarrow S$$

$$S \rightarrow C\ C$$

$$C \rightarrow c\ C \mid d$$

状态	ACTION			GOTO	
	c	d	\$	S	C
0					
1					
2					
3					
4					
5					
6					
7					
8					
9					

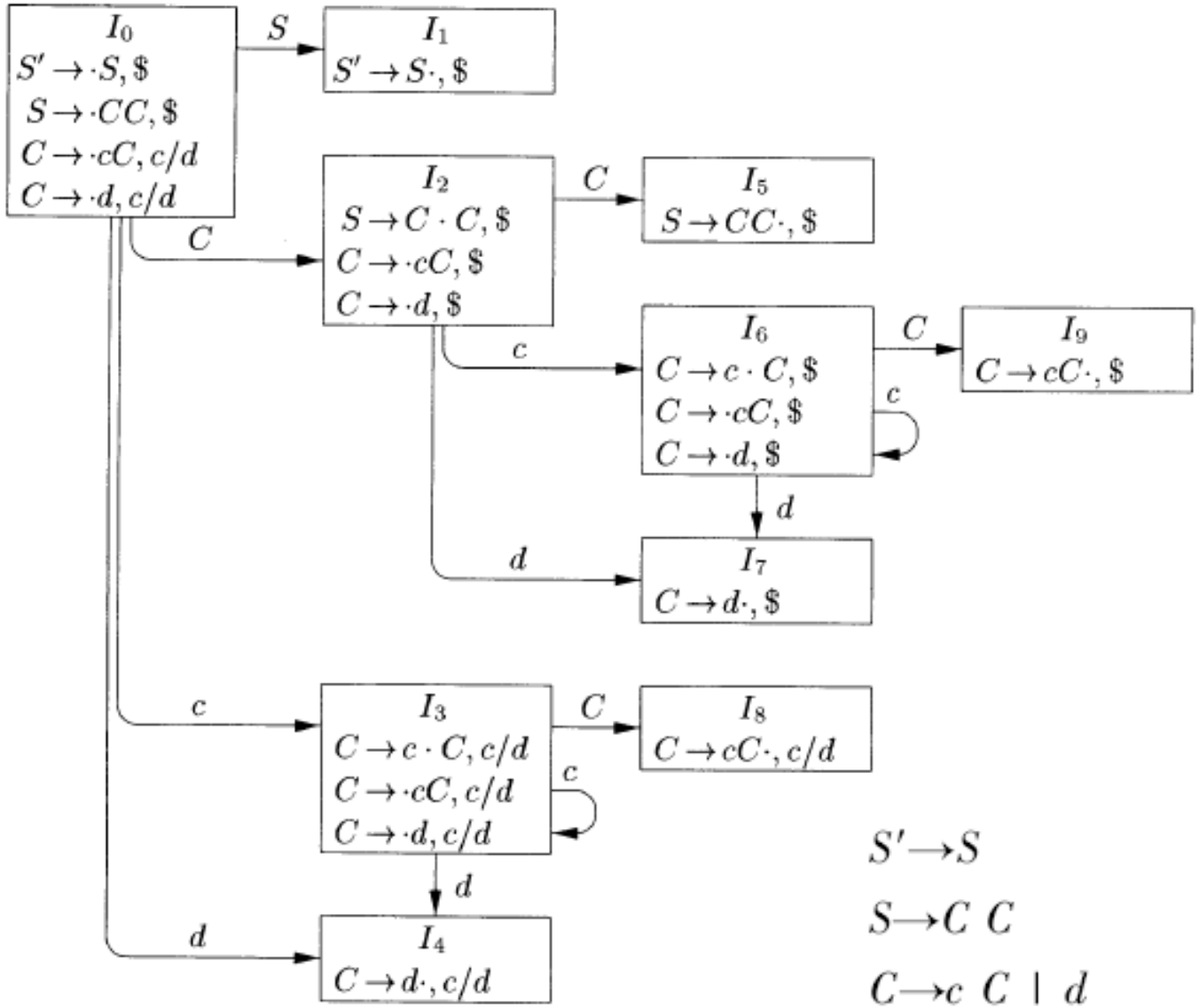


LR(1)文法

- 通过上述算法构造的LR(1)语法分析表中若不存在冲突条目，则给定的文法称为LR(1)文法

LR(1)语法分析器状态再观察

在(3,6) (4,7) (8,9)中，项的第一个分量是一样的，实质上它们来自于同样的LR(0)状态



LALR分析

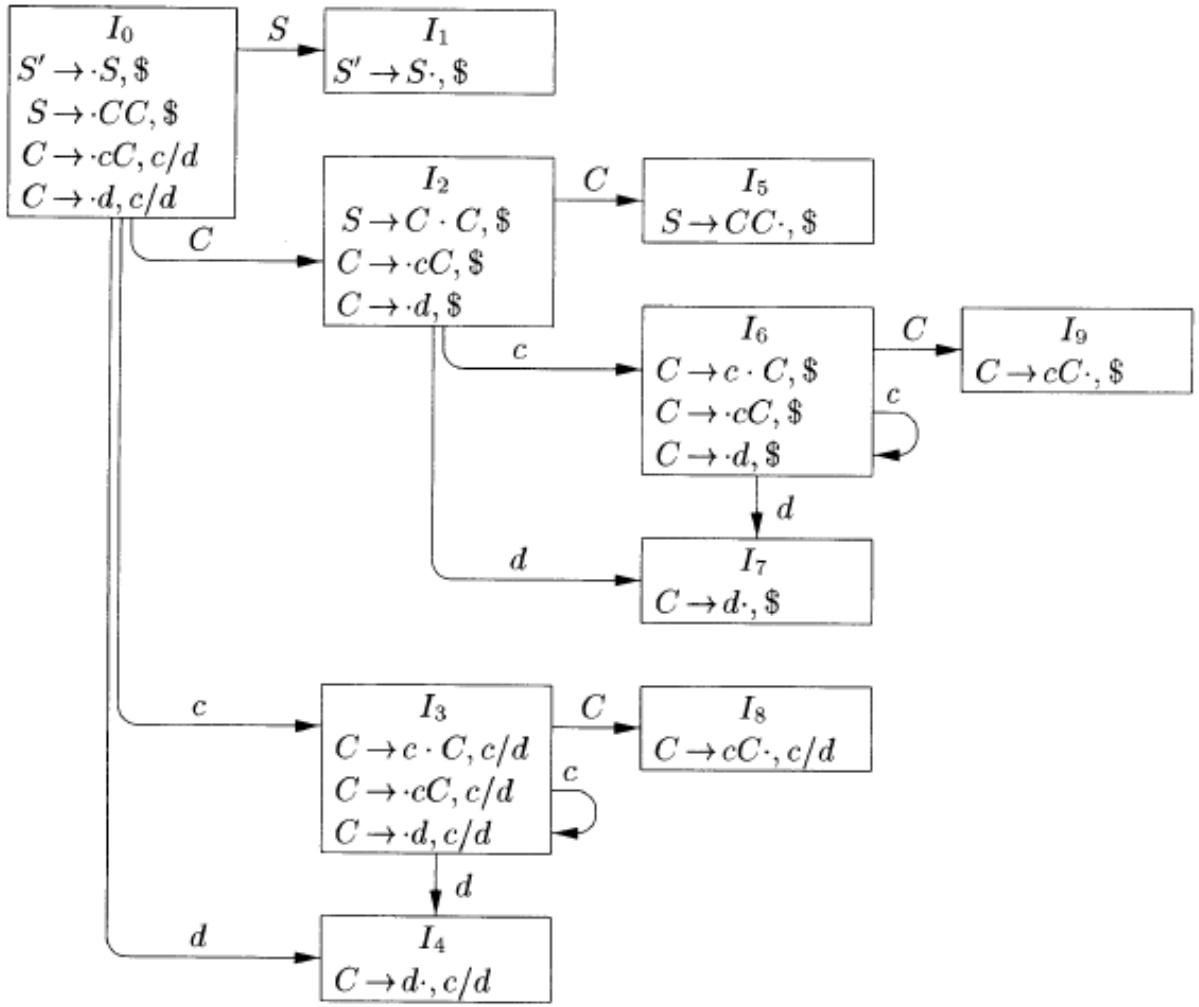
- LR(1)语法分析器的状态过多
- 但LR(1)的分析能力强于SLR(1)
- LALR分析
  - 状态和SLR一样多
  - 能力又比SLR稍微强一些

项集合并

- 将LR(1)项集中具有相同核心的项集合并为一个项集
- 项集核心是指LR(1)项集中第一个分量的集合
- 被合并项集的GOTO目标显然也可以被合并

项集合并

- I3={ [C→c·C, c/d],
  [C→·cC, c/d],
  [C→·d, c/d] }
- I6={ [C→c·C, $],
  [C→·cC, $],
  [C→·d, $] }
- 合并后：
  - I36= { [C→c·C, c/d/$],
    [C→·cC, c/d/$],
    [C→·d, c/d]/$ }
- GOTO(I3,C)和GOTO(I6,C)显然也可以合并……



合并可能产生的冲突

- 合并引起的冲突是指：本来的LR(1)项集没有冲突，而合并具有相同核心的项集后有冲突
- 不可能引入**归约-移入**型冲突
  - 假定合并后有移入_归约冲突，就是说：有项[A→α·, a]和项[B→β·aγ, b]。显然，原来的项集中都有[B→β·aγ, c]。而[A→α·, a]必然也在某个原来的项集中。这样，合并前的LR(1)项集已经存在移入-归约冲突
- 但是可能引起**归约_归约**冲突

归约-归约冲突

$S' \rightarrow S$

$S \rightarrow a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e$

$A \rightarrow c$

$B \rightarrow c$

- 语言{acd，ace，bcd，bce}
- 可行前缀ac的有效项集{[A⟶c ·,d],[B⟶c ·,e]}
- 可行前缀bc的有效项集{[A⟶c ·,e],[B⟶c ·,d]}
- 合并之后的项集为{[A⟶c ·,d/e],[B⟶c ·,d/e]}
  - 当输入为d或e时，用哪个归约？

- 从引起新的冲突可以看出：LALR的分析能力比规范LR(1)弱一些

LALR语法分析表构造方法

- 朴素的方法
- 高效的方法

LALR语法分析表构造方法 – 朴素的方法

- 基本思想：
  - 先构造出LR(1)项集，若没有出现冲突，就将核心的项集进行合并。然后根据合并后得到的项集规范组构造语法分析表
- 输入：一个增广文法*G'*
- 输出：文法*G'*的LALR语法分析表
- 方法：
  - 得到的分析表成为LALR语法分析表。构造得到LR(1)项集族C={$I_0,I_1,...,I_n$}
  - 对于LR(1)项集中的每个核心，找出所有具有这个核心的项集，并把这些项集替换为它们的并集
  - 令C'={$J_0,J_1,...,J_n$}是得到的LR(1)项集族。按照LR(1)分析表的构造方法得到ACTION表。（注意检查若存在冲突，则这个文法不是LALR的）
  - GOTO表的构造：若J是一个或者多个LR(1)项集的并集，即$J=I_1 \cup I_2 \cup ... \cup I_K$，令$K$是所有和GOTO($I_1$,$X$)具有相同核心的项集的并集，那么GOTO($J$,$X$)=$K$

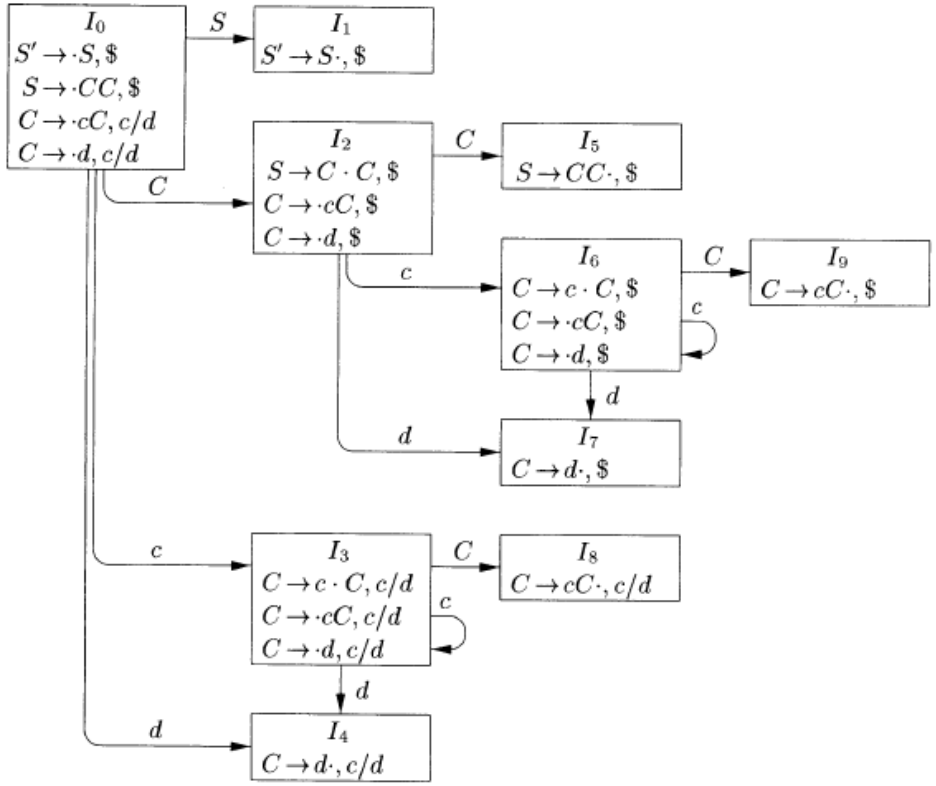
LALR分析表构造示例

$$S' \rightarrow S$$
$$S \rightarrow C\ C$$
$$C \rightarrow c\ C \mid d$$

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

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

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



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

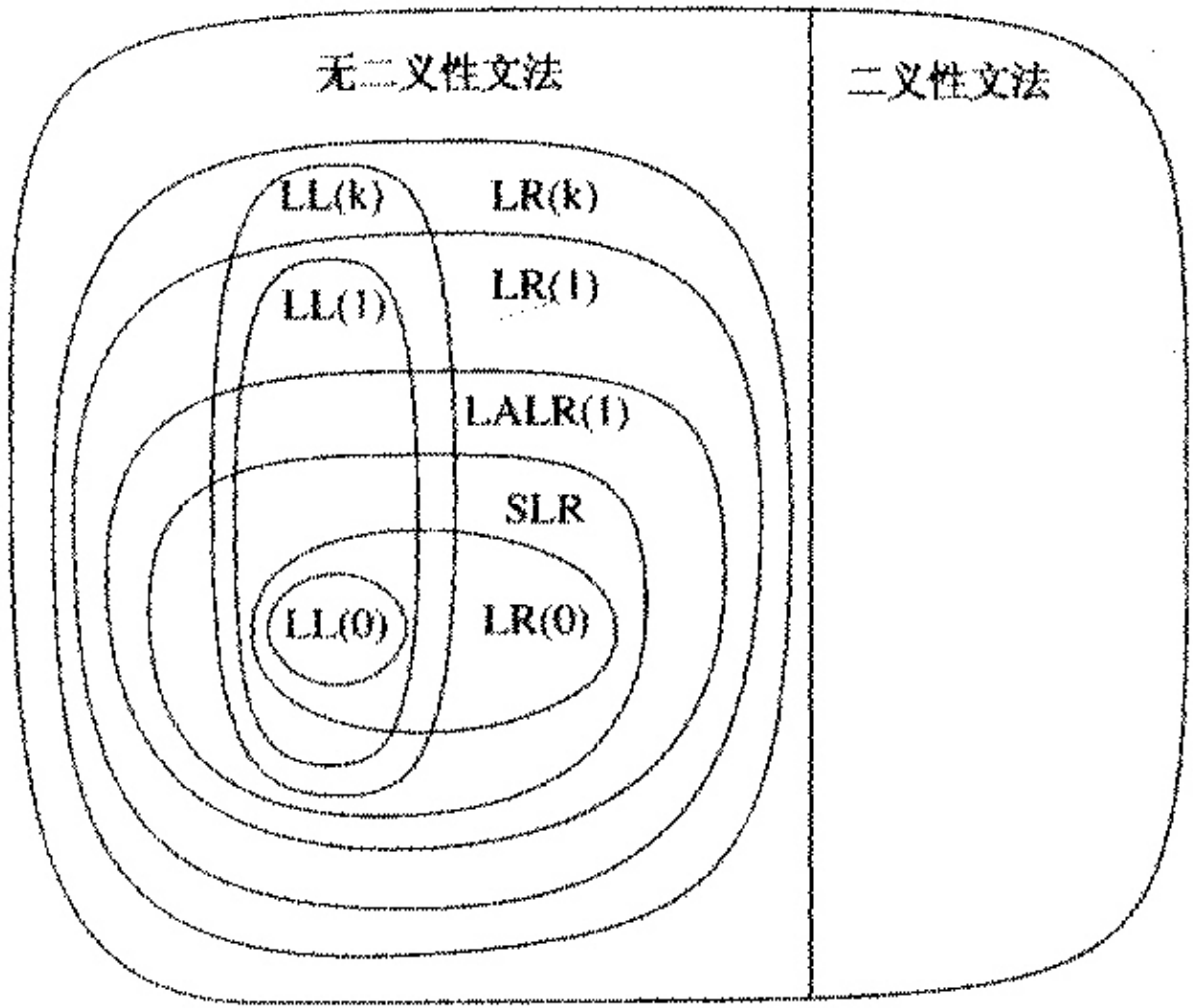
LALR分析器和LR分析器

- 若构造出的LALR分析表中没有冲突，则可以用LALR分析表进行语法分析
- 如果是正确的输入串
  - LALR和LR分析器执行完全相同的移入和归约动作序列，只是栈中的状态名字有所不同
- 如果是错误的输入串
  - 则LALR分析器可能会比LR分析器晚一点报错
  - LALR分析器不会在LR分析器报错后再移入任何符号，但是可能会继续执行一些归约

LALR技术本质

- 对LR(1)项集规范族中所谓的同心项集进行合并，从而使得分析表既保持了LR(1)项中向前看符号的信息，又使状态数减少到与SLR分析表的一样多

文法比较



习题

- 已知文法

  E → E + T | T

  T → T F | F

  F → F * | (E) | a | b

- ① 给出LR(1)项集{ [F → (·E), +] }的闭包；
- ② 给出①中项集闭包相对于文法符号E的后继LR(1)项集闭包。

二义性文法的自底向上分析

- 二义性文法都不是LR的
- 二义性文法却有其存在的必要
- 对于某些二义性文法
  - 可以通过消除二义性规则来保证每个句子只有一棵语法分析树
  - 且可以在LR分析器中实现这个规则

优先级/结合性消除冲突

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

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

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

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

- 二义性文法的优点
  - 容易修改算符的优先级和结合性
  - 简洁：较少的非终结文法符号
  - 高效：不需要处理E → T这样的归约

二义性表达式文法的LR(0)项集

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

- I_7，I_8中有冲突，在输入+或*时，不能确定是归约还是移入，且不可能通过向前看符号解决

I_0:
$E' \rightarrow \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_1:
$E' \rightarrow E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_2:
$E \rightarrow (\cdot E)$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_3:
$E \rightarrow \mathbf{id} \cdot$

I_4:
$E \rightarrow E + \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_5:
$E \rightarrow E * \cdot E$
$E \rightarrow \cdot E + E$
$E \rightarrow \cdot E * E$
$E \rightarrow \cdot (E)$
$E \rightarrow \cdot \mathbf{id}$

I_6:
$E \rightarrow (E \cdot)$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_7:
$E \rightarrow E + E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_8:
$E \rightarrow E * E \cdot$
$E \rightarrow E \cdot + E$
$E \rightarrow E \cdot * E$

I_9:
$E \rightarrow (E) \cdot$

基于优先级解决冲突

前缀	栈	输入
$E + E$	0 1 4 7	$*$ **id** $

- 如果*的**优先级**大于+，且+是**左结合**的
  - 下一个符号为+时，我们应该将E+E归约为E
  - 下一个符号为*时，我们应该移入*，期待移入下一个符号

解决冲突之后的SLR(1)分析表

- 对于状态7，输入
  - +时归约
  - *时移入
- 对于状态8
  - 执行归约

I_7: $E \to E + E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_8: $E \to E * E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

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

悬空else的二义性

- 栈中内容if *expr* then *stmt*，是输入else，还是归约？
- 答案是移入

$S' \rightarrow S$

$S \rightarrow i\ S\ e\ S \mid i\ S \mid a$

Follow(S)={e,$}

I_0:
$S' \rightarrow \cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_1:
$S' \rightarrow S\cdot$

I_2:
$S \rightarrow i\cdot SeS$
$S \rightarrow i\cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_3:
$S \rightarrow a\cdot$

I_4:
$S \rightarrow iS\cdot eS$
$S \rightarrow iS\cdot$

I_5:
$S \rightarrow iSe\cdot S$
$S \rightarrow \cdot iSeS$
$S \rightarrow \cdot iS$
$S \rightarrow \cdot a$

I_6:
$S \rightarrow iSeS\cdot$

语法错误的处理

- 错误难以避免
- 编译器需要具有处理错误的能力
- 程序中可能存在不同层次的错误
  - 词法错误
  - 语法错误
  - 语义错误
  - 逻辑错误
- 语法错误相对容易发现，语义和逻辑错误较难精确的检测到
- 语法分析器中错误处理程序的设计目标
  - 清晰准确地**报告**出现错误，并指出错误的**位置**
  - 能从当前错误中**恢复**，以继续检测后面的错误
  - 尽可能减少处理正确程序的开销

预测分析中的错误恢复

- 错误恢复
  - 当预测分析器报错时，表示输入的串不是句子
  - 对于使用者而言，希望预测分析器能够进行恢复处理后继续语法分析过程，以便在一次分析中找到更多的语法错误
  - 但是有可能恢复得并不成功，之后找到的语法错误有可能是假的
  - 进行错误恢复时可用的信息：栈里面的符号，待分析的符号
- 两类错误恢复方法
  - 恐慌模式
  - 短语层次的恢复

恐慌模式

- 基本思想
  - 语法分析器忽略输入中的一些符号，直到出现由设计者选定的某个同步词法单元
  - 解释
    - 语法分析过程总是试图在输入的前缀中找到和某个非终结符号对应的串；出现错误表明现在已经不可能找到这个非终结符号（程序结构）的串
    - 如果编程错误仅仅局限于这个程序结构，那么我们可以考虑跳过这个程序结构，假装已经找到了这个结构；然后继续进行语法分析处理
  - 同步词法单元就是这个程序结构结束的标志

同步词法单元的确定

- 非终结符号A的同步集合的启发式规则
  - FOLLOW(A)的所有符号在非终结符A的同步集合中
    - 这些符号的出现可能表示之前的那些符号是错误的A的串
  - 将高层次的非终结符号对应串的开始符号加入到较低层次的非终结符号的同步集合
    - 比如语句的开始符号，if，while等可以作为表达式的同步集合
  - FIRST(A)中的符号加入到非终结符号A的同步集合
    - 碰到这些符号时，可能意味着前面的符号是多余的符号
  - 如果A可以推导出空串，那么把此产生式当作默认值
  - 在栈顶的终结符号出现匹配错误时，可以直接弹出相应的符号，并且发出消息称已经插入了这个终结符号
- 哪些符号需要确定同步集合需要根据特定的应用而定

恐慌模式的例子（1）

- 对文法4.28对表达式进行语法分析时，可以直接使用FIRST、FOLLOW作为同步集合
  - 我们为E、T和F设定同步集合
  - 空条目表示忽略输入符号；synch表示忽略到同步集合，然后弹出非终结符号
  - 终结符号不匹配时，弹出栈中终结符号

非终结符号	输入符号					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$	synch	synch	$F \to (E)$	synch	synch

恐慌模式的例子（2）

- 错误输入：

  +id * + id

非终结符号	输入符号					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**	synch	synch	$F \to (E)$	synch	synch

栈	输入	说明
E \$	) **id** * + **id** \$	错误，略过)
E \$	**id** * + **id** \$	**id** 在 FIRST(E)中
TE' \$	**id** * + **id** \$	
$FT'E'$ \$	**id** * + **id** \$	
id $T'E'$ \$	**id** * + **id** \$	
$T'E'$ \$	* + **id** \$	
* $FT'E'$ \$	* + **id** \$	
$FT'E'$ \$	+ **id** \$	错误，$M[F, +] =$ synch
$T'E'$ \$	+ **id** \$	F 已经被弹出栈
E' \$	+ **id** \$	
+ TE' \$	+ **id** \$	
TE' \$	**id** \$	
$FT'E'$ \$	**id** \$	
id $T'E'$ \$	**id** \$	
$T'E'$ \$	\$	
E' \$	\$	
\$	\$	

短语层次的恢复

- 在预测语法分析表的空白条目中插入错误处理例程的函数指针
- 例程可以改变、插入或删除输入中的符号；发出适当的错误消息

LR语法分析中的错误恢复（1）

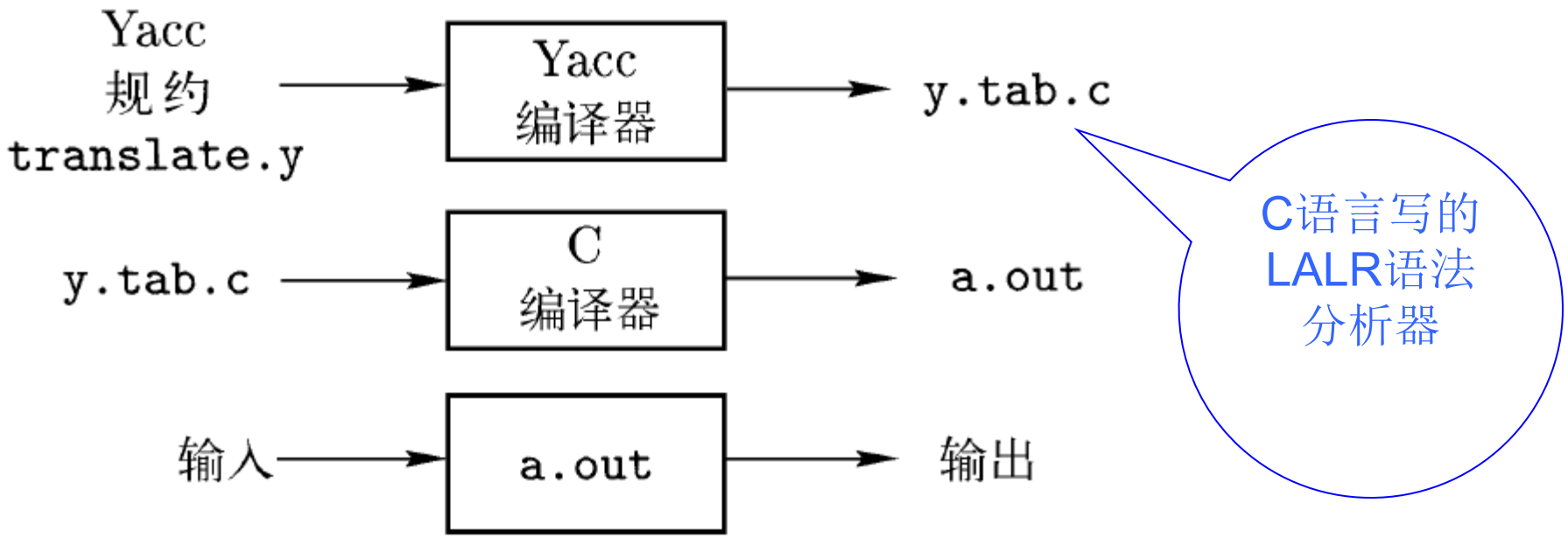
- LR语法分析器查询GOTO表时不会发现错误
- 但是可能在查询ACTION表时发现报错条目
  - 假设栈中的符号串为α，当前输入符号为a；报错表示不可能存在终结符号串x使得αax是一个最右句型
- 恐慌模式的错误恢复策略
  - 从栈顶向下扫描，找到状态s，s有一个对应于非终结符号A的GOTO目标；（s之上的状态被丢弃）
  - 在输入中读入并丢弃一些符号，直到找到一个可以合法跟在A之后的符号a（不丢弃a）
  - 将GOTO(s,A)压栈；继续进行正常的语法分。
  - **基本思想**：假定当前正在试图归约到A且碰到了语法错误，因此设法扫描完包含语法错误的A的子串，假装找到了A的一个实例

LR语法分析中的错误恢复（2）

- 短语层次的恢复
  - 检查LR分析表中的每个报错条目，根据语言的特性来确定程序员最可能犯了什么错误，然后构造适当的恢复程序

语法分析器生成工具YACC

- YACC的使用方法如下：



YACC源程序的结构

- 声明
  - 分为可选的两节：第一节放置C声明，第二节是对词法单元的声明。
- 翻译规则：
  - 指明产生式及相关的语义动作
- 辅助性C语言例程
  - 被直接拷贝到生成的C语言源程序中，
  - 可以在语义动作中调用。
  - 其中必须包括yylex()。这个函数返回词法单元，可以由LEX生成

声明
%%
翻译规则
%%
辅助性C语言程序

翻译规则的格式

<产生式头>　：　<产生式体>1 {<语义动作>1}
| <产生式体>2 {<语义动作>2}
… … … …
| <产生式体>n {<语义动作>n}
;

- 第一个产生式的头被看作开始符号；
- 语义动作是C语句序列；
- $$表示和产生式头相关的属性值，$i表示产生式体中第i个文法符号的属性值。
- 当我们按照某个产生式归约时，执行相应的语义动作。通常可以根据$i来计算$$的值。
- 在YACC源程序中，可以通过定义YYSTYPE来定义$$，$i的类型。

YACC 源程序的例子

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

%token DIGIT

%%
line   : expr '\n'        { printf("%d\n", $1); }
       ;
expr   : expr '+' term    { $$ = $1 + $3; }
       | term
       ;
term   : term '*' factor  { $$ = $1 * $3; }
       | factor
       ;
factor : '(' expr ')'     { $$ = $2; }
       | DIGIT
       ;
%%
yylex() {
    int c;
    c = getchar();
    if (isdigit(c)) {
        yylval = c-'0';
        return DIGIT;
    }
    return c;
}
```

YACC对于二义性文法的处理

- 缺省的处理方法
  - 对于归约/归约冲突，选择前面的产生式
  - 对于归约/移入冲突，总是移入（悬空-else的解决）
- 运行选项-v可以在文件y.output中看到冲突的描述及其解决方法；
- 可以通过一些命令来确定终结符号的优先级/结合性，解决移入/归约冲突。
  - 结合性：%left　　%right　　%nonassoc
  - 终结符号的优先级通过它们在声明部分的出现顺序而定。
  - 产生式的优先级设定为它的最右终结符号的优先级。也可以加标记%prec<终结符号>，指明产生式的优先级等同于该终结符号
  - 移入符号a/按照A→α归约：当A→α的优先级高于a，或者两者优先级相同但产生式是左结合时，选择归约，否则移入。

YACC的错误恢复

- 使用错误产生式的方式来完成语法错误恢复
  - 错误产生式A → **error** α
  - 例如：*stmt* → **error** ;
- 首先定义哪些“主要”非终结符号有相关的错误恢复动作；
  - 比如：表达式，语句，块，函数定义等对应的非终结符号
- 当语法分析器碰到错误时
  - 不断弹出栈中状态，直到有一个状态包含A →.**error** α;
  - 分析器将error移入栈中
  - 如果α为空，分析器直接执行归约，并调用相关的语义动作；否则向前跳过一些符号，直到找到可以归约为α的串

实例

- LR(0)项集族构造
- SLR分析表
- SLR分析
- LR(1)项集族构造
- LR(1)分析表
- LR(1)分析
- LALR项集族构造
- LALR分析表
- LALR分析

$$S' \rightarrow S$$

$$S \rightarrow C\ C$$

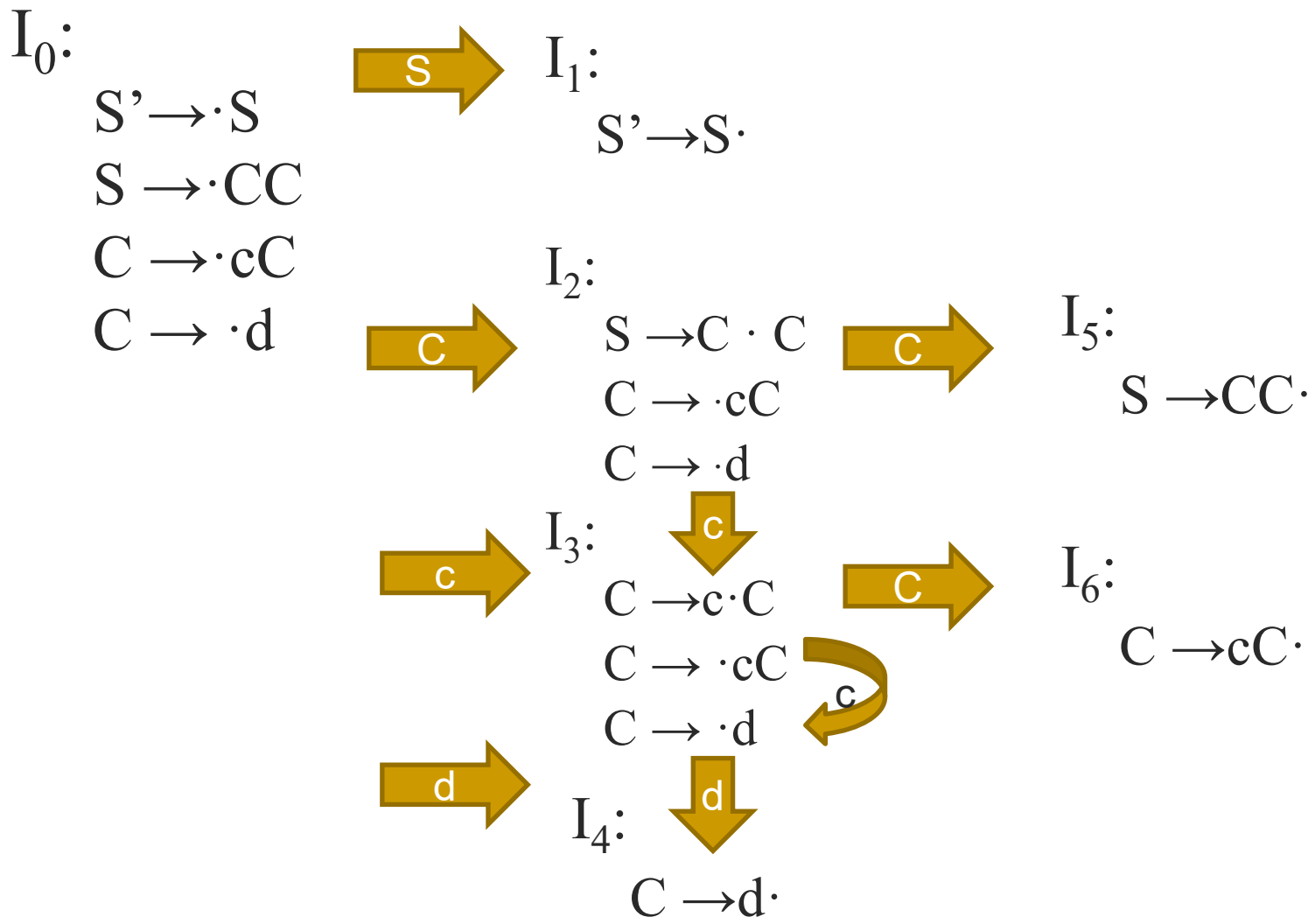
$$C \rightarrow c\ C \mid d$$

LR(0)

$S' \rightarrow S$

$S \rightarrow C\ C$

$C \rightarrow c\ C \mid d$



SLR分析表

状态	Action			Goto	
	c	d	$	S	C
0	s3	s4		1	2
1			accept		
2	s3				5
3	s3	s4			6
4	r	r	r		
5			r		
6	r	r	r		

Follow(C)={c,d,$}

Follow(S)={$}