第四章 语法分析

《编译原理》

谭添

南京大学计算机系

2022年春季

版权所有 南京大学计算机科学与技术系 许畅 2022春季版（有修改）

概要

- 语法分析器
- 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

南大编译许勇

语法分析器的作用

- 基本作用
  - 从词法分析器获得词法单元的序列，确认该序列是否可以由语言的文法生成
  - 对于语法错误的程序，报告错误信息
  - 对于语法正确的程序，生成**语法分析树**(简称**语法树**)
    - 通常产生的是抽象语法树（AST）

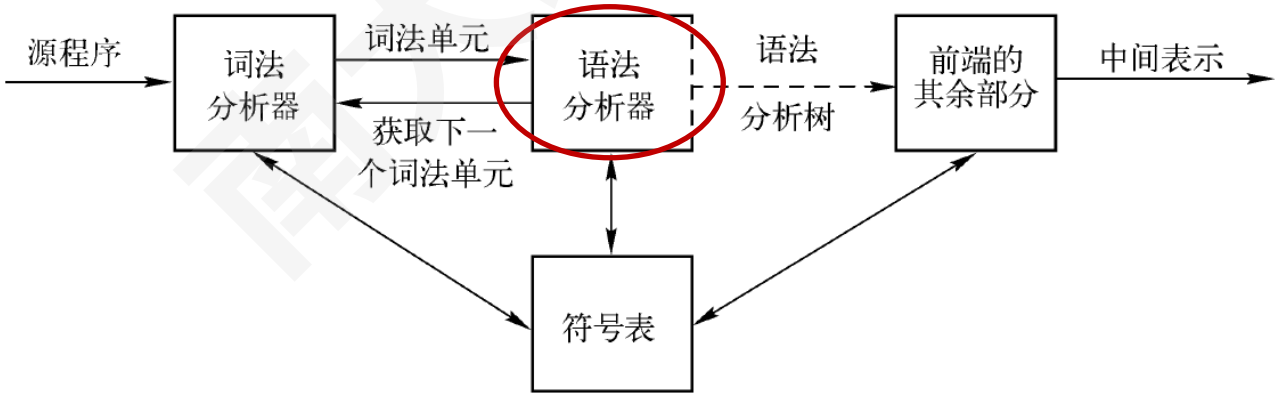


图 4-1　编译器模型中语法分析器的位置

程序设计语言构造的描述

- 程序设计语言构造的语法可使用**上下文无关文法**(Context-Free Grammar) 或BNF表示法来描述
  - 文法可给出精确易懂的语法规则
  - 可以自动构造出某些类型的文法的语法分析器
  - 文法指出了语言的结构，有助于进一步的语义处理/代码生成
  - 支持语言的演化和迭代

上下文无关文法

类比：正则定义
- *letter_* → A | … | Z | a | … | z | _
- *digit* → 0 | 1 | … | 9
- *id* → *letter_* (*letter_* | *digit*)*

- 一个**上下文无关文法** (CFG) 包含四个部分
  - **终结符号**：组成串的基本符号 (词法单元名字)
  - **非终结符号**：表示串的集合的语法变量
    - 在程序设计语言中通常对应于某个程序构造，比如*stmt* (语句)
  - **产生式**：描述将终结符号和非终结符号组成串的方法
    - 形式：头 (左) 部 → 体 (右) 部
    - 头部是一个非终结符号，右部是一个符号串
    - 例子：*expression* → *expression* + *term*
  - **起始符号**：某个被指定的非终结符号
    - 它对应的串的集合就是文法的语言

上下文无关文法的例子

- 简单算术表达式的文法
  - 终结符号：id, +, –, *, /, (,)
  - 非终结符号：*expression* (表达式), *term* (项), *factor* (因子)
  - 起始符号：*expression*
  - 产生式集合

expression → *expression* + *term*

expression → *expression* – *term*

expression → *term*

term → *term* * *factor*　　*term* → *term* / *factor*

term → *factor*

factor → id　　*factor* → (*expression*)

南大编译计算

文法简单形式的例子

$E \rightarrow E + T$

$\quad | \; E - T$

$\quad | \; T$

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

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

(*E*: *expression*, *T*: *term*, *F*: *factor*)

- 注意
  - |是**元符号**(即文法描述中的符号，而不是文法符号)
  - 这里的(和)不是元符号，而是终结符

推导 (1)

- **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（右部）
  - 从起始符号出发，不断进行上面的替换，就可以得到文法的不同句型
- 例子
  - 文法：$E \rightarrow -E$ ①
    $\mid E+E$ ②
    $\mid E*E$ ③
    $\mid (E)$ ④
    $\mid \mathrm{id}$ ⑤
  - 推导序列：$E \Rightarrow -E \Rightarrow -(E) \Rightarrow -(\mathrm{id})$

推导 (2)

- 推导的正式定义
  - 如果$A \rightarrow \gamma$是一个产生式，那么$\alpha A \beta \Rightarrow \alpha \gamma \beta$　例：$T \Rightarrow T * F$
  - **最左 (右) 推导**：$\alpha(\beta)$ 中不包含非终结符号　例：$E + T \Rightarrow T + T$
    - 符号：$\overset{*}{\underset{lm}{\Rightarrow}}$　$\overset{*}{\underset{rm}{\Rightarrow}}$
- 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：$\overset{+}{\Rightarrow}$
  - $\alpha \overset{+}{\Rightarrow} \beta$，即$\alpha \overset{*}{\Rightarrow} \beta$且$\alpha$不等于$\beta$

句型/句子/语言

- **句型** (Sentential form)
  - 如果$S \overset{*}{\Rightarrow} \alpha$，那么$\alpha$就是文法$S$的句型
  - 可能既包含非终结符号，又包含终结符号，也可以是空串
- **句子** (Sentence)
  - 文法的句子就是只包含终结符号的/空串句型
- **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$S \overset{*}{\Rightarrow} w$

文法及其生成的语言

- 语言是从文法的起始符号出发，能推导得到的所有句子的集合
  - 文法G：$S \rightarrow aS \mid a \mid b$，$L(G) = \{ a^i(a|b), i >= 0 \}$
  - 文法G：$S \rightarrow aSb \mid ab$，$L(G) = \{ a^n b^n, n >= 1 \}$
  - 文法G：$S \rightarrow (S)S \mid \varepsilon$，$L(G) = \{$所有具有对称括号对的串$\}$

- 如何验证文法G所确定的语言L
  - 证明G生成的每个串都在L中
  - 证明L中的每个串都能被G生成

语法分析树

- 推导的图形表示形式
  - **根结点**的标号是文法的起始符号
  - 每个**叶子结点**的标号是非终结符号、终结符号或ε
  - 每个**内部结点**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
    - 结点的标号为产生式头，其子结点从左到右是产生式的体
- 树的叶子组成的序列是根的文法符号的一个句型
- 一棵语法分析树可对应多个推导序列
  - 但只有唯一的最左推导及最右推导

语法分析树的例子

- 文法：$E \rightarrow -E \mid E+E \mid E*E \mid (E) \mid \mathrm{id}$
- 句子：$-(\mathrm{id}+\mathrm{id})$

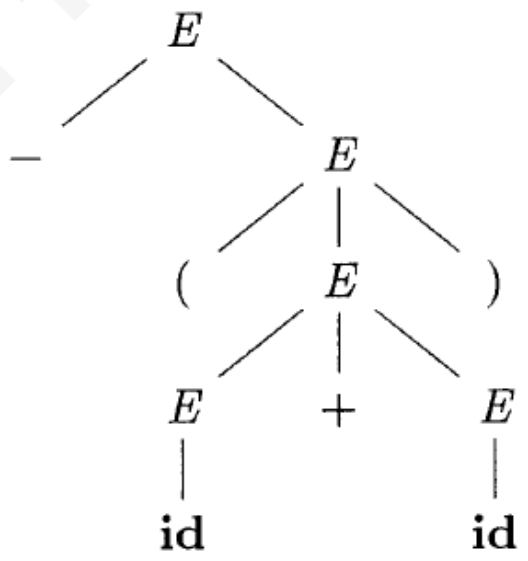


图 4-3　$-(\mathbf{id}+\mathbf{id})$的语法分析树

推导的图形表示形式

- **根结点**的标号是文法的起始符号
- 每个**叶子结点**的标号是非终结符号、终结符号或ε
- 每个**内部结点**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
  - 结点的标号为产生式头，其子结点从左到右是产生式的体

树的叶子组成的序列是根的文法符号的一个句型

从推导序列构造分析树的例子

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

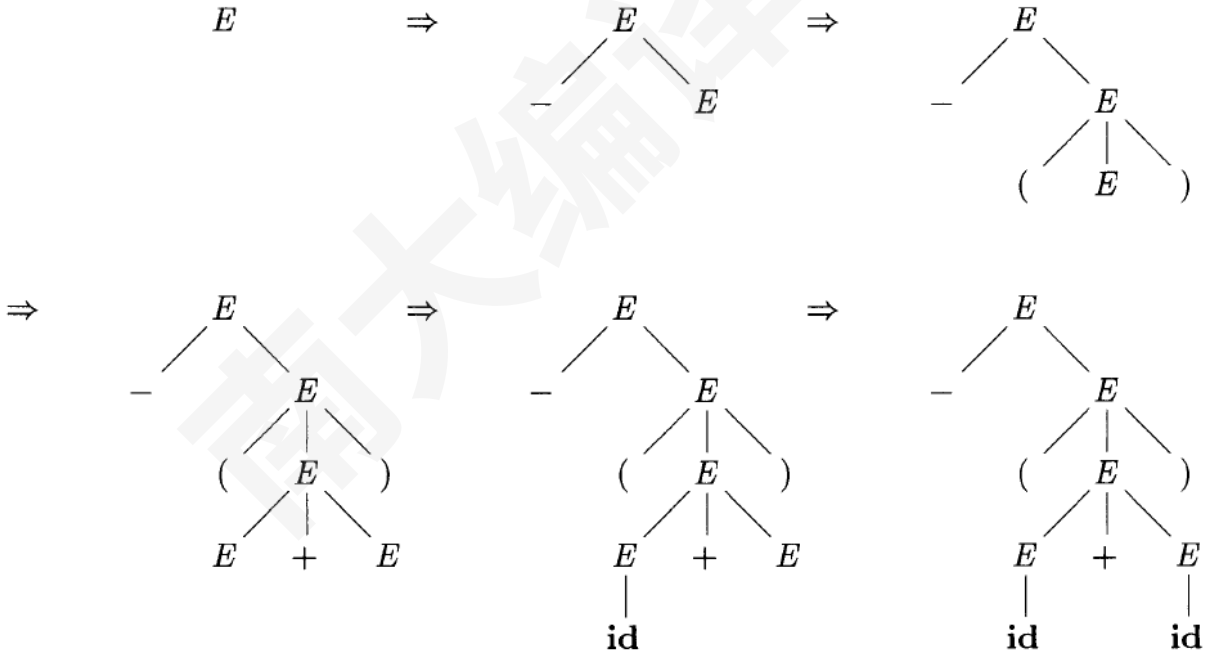
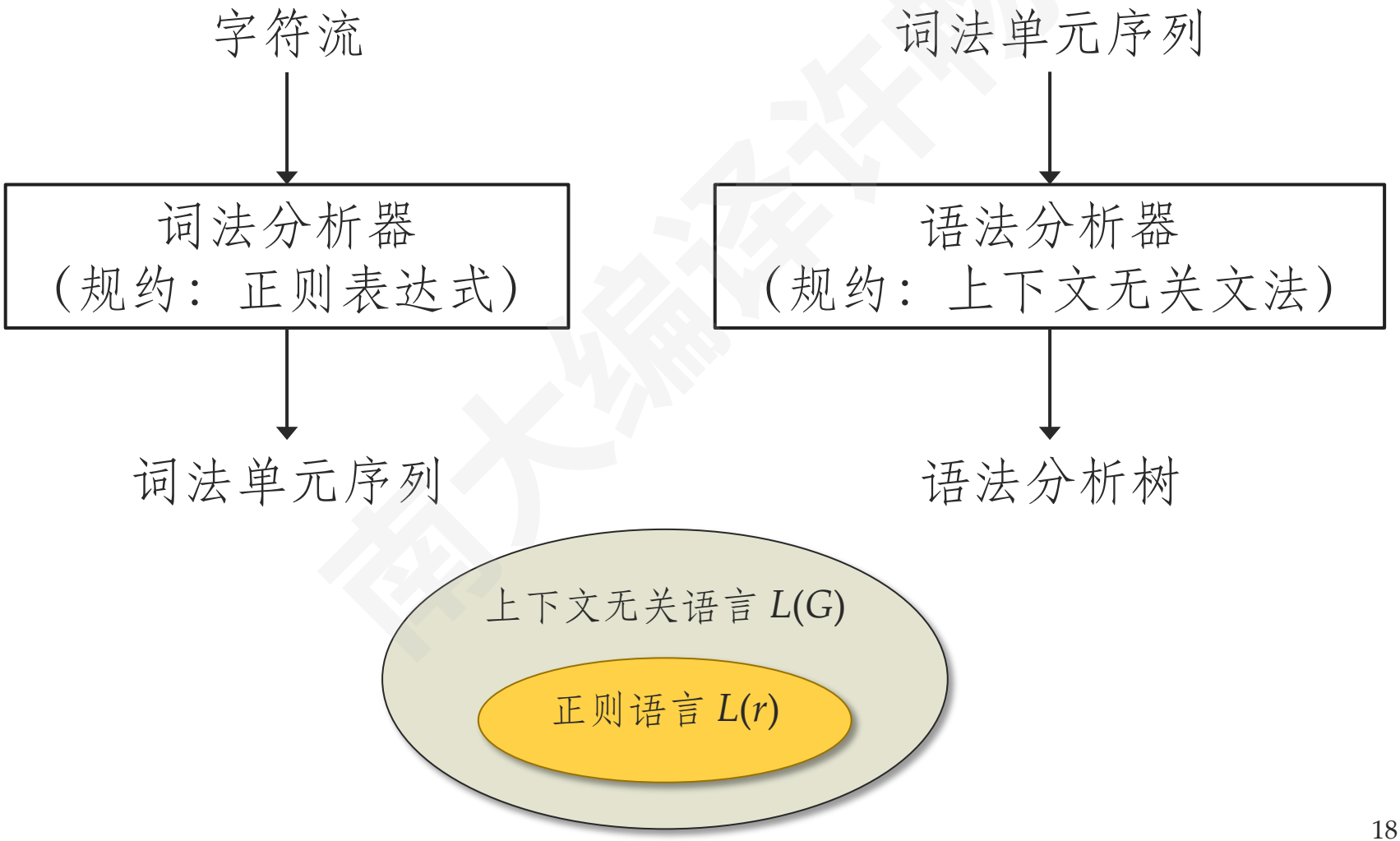


图 4-4　推导(4.8)的语法分析树序列

词法分析和语法分析的比较



上下文无关文法和正则表达式 (1)

- 上下文无关文法比正则表达式的能力更强
  - 所有的正则语言都可以使用文法描述
  - 但有一些用文法描述的语言不能用正则表达式描述
- 证明
  - 任何正则语言都能构造出一个能识别该语言的上下文无关文法

$L(r)$表示正则表达式r所表达的语言

- 基本部分
  - ε是一个正则表达式，$L(\varepsilon) = \{\varepsilon\}$
  - 如果a是Σ上的一个符号，那么a是正则表达式，$L(a) = \{a\}$
- 归纳步骤
  - 选择：$(x)|(y)$，$L((x)|(y)) = L(x) \cup L(y)$
  - 连接：$(x)(y)$，$L((x)(y)) = L(x)L(y)$
  - 闭包：$(x)^*$，$L((x)^*) = (L(x))^*$

X表示描述x的上下文无关文法

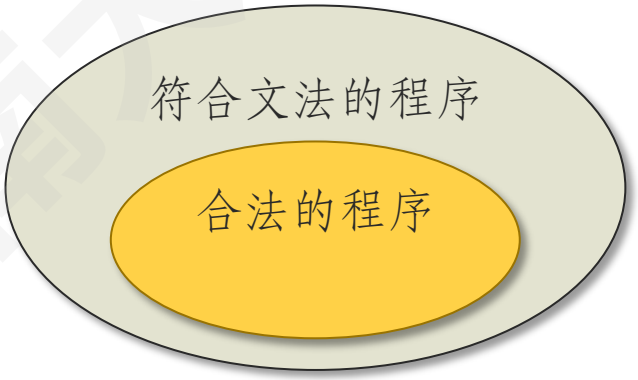
- 基本部分
  - $S \rightarrow \varepsilon$
  - $S \rightarrow a$
- 归纳步骤
  - 选择：$S \rightarrow X \mid Y$
  - 连接：$S \rightarrow XY$
  - 闭包：$S \rightarrow \varepsilon \mid XS$

上下文无关文法和正则表达式 (2)

- 证明 (续)
  - 有一些用文法描述的语言不能用正则表达式描述
  - 首先$S \rightarrow aSb \mid ab$描述了语言$L\ \{a^n b^n \mid n > 0\}$，但这个语言无法用DFA识别
    - 假设有DFA识别此语言L，且这个DFA有k个状态。那么在识别a^{k+1}...的输入串时，必然两次到达同一个状态。假设自动机在第i个和第j个a时进入同一个状态$(i \neq j)$，那么：因为DFA识别L，$a^i b^i$必然到达接受状态，因此$a^j b^i$必然也到达接受状态。
    - 直观地讲：有穷自动机不能无限计数（以有涯随无涯，殆已！）

设计文法 (1)

- 文法能够描述程序设计语言的大部分语法
  - 但不是全部，比如，标识符的先声明后使用则无法用上下文无关文法描述
  - 因此语法分析器接受的语言是程序设计语言的超集；必须通过语义分析来剔除一些符合文法、但不合法的程序



设计文法 (1)

- 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
    - **二义性**：文法可以为一个句子生成多颗不同的分析树
  - 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 $A \Rightarrow A\alpha$，则称这个文法是左递归的
  - 提取左公因子

二义性 (1)

- 二义性 (Ambiguity)：如果一个文法可以为某个句子生成多棵语法分析树，这个文法就是二义的
- 例子　文法：$E \rightarrow E+E \mid E*E \mid (E) \mid \text{id}$
  - $E \Rightarrow E+E \Rightarrow \text{id}+E \Rightarrow \text{id}+E*E \Rightarrow \text{id}+\text{id}*E$
    $\Rightarrow \text{id}+\text{id}*\text{id}$
  - $E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow \text{id}+E*E \Rightarrow \text{id}+\text{id}*E$
    $\Rightarrow \text{id}+\text{id}*\text{id}$

都是最左推导

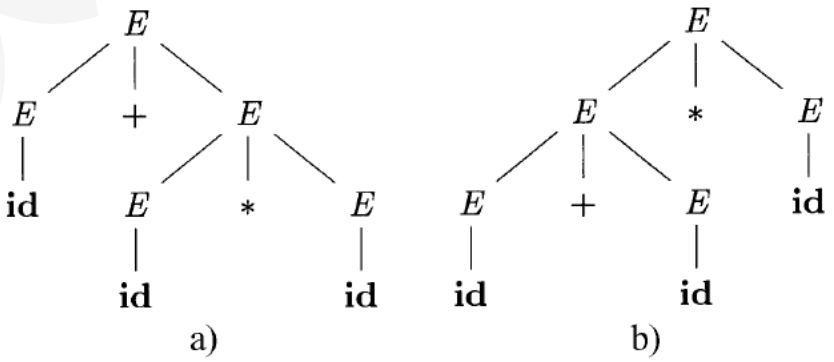


图 4-5　**id + id * id** 的两棵语法树

二义性 (2)

- 程序设计语言的文法通常是无二义的
  - 否则就会导致一个程序有多种“正确”的解释
  - 如文法 $E \rightarrow E+E \mid E*E \mid (E) \mid \text{id}$ 应对句子 a + b * c 时　a + (b * c) ok　(a + b) * c no-no
- 有些二义性的情况可以方便文法或语法分析器的设计
  - 但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

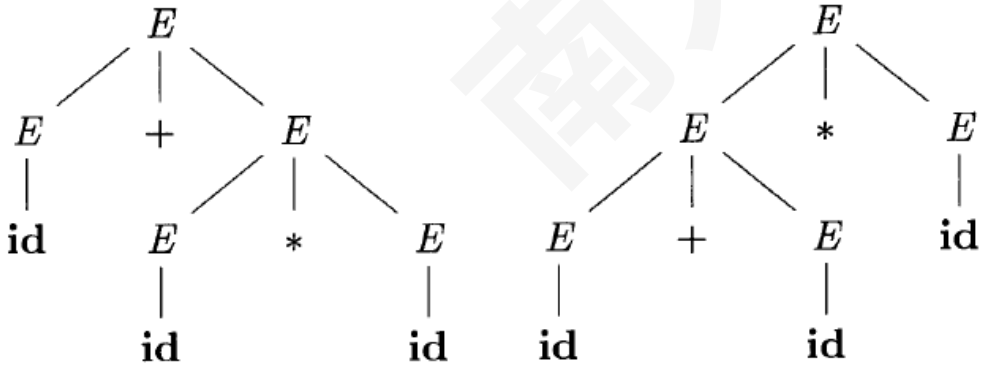
二义性的消除 (1)

- 二义性的根源：多种“正确”推导处于文法同一层
- 消除二义性的惯用技术：分层
  - 改造文法，对于引发二义性的多种推导处于文法同一层的情况，将真正想要的推导提取出来，放到更深的层次
  - 最左推导中，更深层的非终结符总是会被优先替换
  - 确保只有一种最左推导，消除二义性
- 例子
  - 原：$E \rightarrow E+E \mid E*E \mid (E) \mid \text{id}$
  - 新：$E \rightarrow E+T \mid T$
    $T \rightarrow T*F \mid F$
    $F \rightarrow (E) \mid \text{id}$

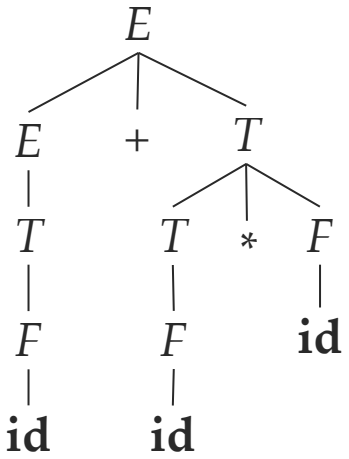
二义性的消除 (2)

- 原： $E \rightarrow E + E$
  $\mid E * E$
  $\mid (E) \mid \text{id}$
- $E \Rightarrow \underline{\boldsymbol{E + E}} \Rightarrow \underline{\text{id}} + E$
  $\Rightarrow \text{id} + \underline{E * E} \Rightarrow \text{id} + \underline{\text{id}} * E$
  $\Rightarrow \text{id} + \text{id} * \underline{\text{id}}$
- $E \Rightarrow \underline{\boldsymbol{E * E}} \Rightarrow \underline{E + E} * E$
  $\Rightarrow \underline{\text{id}} + E * E \Rightarrow \text{id} + \underline{\text{id}} * E$
  $\Rightarrow \text{id} + \text{id} * \underline{\text{id}}$

- 新： $E \rightarrow E + T \mid T$
  $T \rightarrow T * F \mid F$
  $F \rightarrow (E) \mid \text{id}$
- $E \Rightarrow \underline{\boldsymbol{E + T}} \Rightarrow \underline{T} + T \Rightarrow \underline{F} + T$
  $\Rightarrow \underline{\text{id}} + T \Rightarrow \text{id} + \underline{T * F}$
  $\Rightarrow \text{id} + \underline{F} * F \Rightarrow \text{id} + \underline{\text{id}} * F$
  $\Rightarrow \text{id} + \text{id} * \underline{\text{id}}$



a) b)



自顶向下的语法分析

- 为输入串构造语法分析树
  - 从分析树的根结点（即文法的起始符号）开始，自顶相下，按照先根次序，深度优先地创建各个结点
  - 对应于最左推导
  - 关键步骤：应用产生式创建新的子结点
    - $A \rightarrow \alpha \mid \beta \mid \gamma \ldots$ 如何知道应用哪个产生式?

递归下降的语法分析 (1)

- 每个非终结符号对应于一个过程/函数，该过程负责扫描此非终结符号对应的结构
  - $S \rightarrow \ldots$ bool S() { … }
    $A \rightarrow \ldots$ bool A() { … }
    $B \rightarrow \ldots$ bool B() { … }
- 程序执行从开起符号对应的过程开始
  - 当扫描整个输入串时宣布分析成功完成
- 关键步骤：应用产生式
  - $A \rightarrow X_1 \mid X_2 \mid X_3 \ldots$ 如何知道应用哪个产生式?
  - 依次猜X_1、X_2、X_3…猜对结束，猜错回溯

```
S();
bool S() { …
    return true;
}
```

递归下降的语法分析 (2)

- $A \rightarrow t$（终结符）

```
bool A() {
    if nextToken is t
        consume nextToken;
        return true;
    return false;
}
```

- $A \rightarrow XYZ$

如果X/Y/Z是终结符，则展开X()成识别终结符的代码

```
bool A() {
    if X() && Y() && Z()
        return true;
    return false;
}
```

- $A \rightarrow X \mid Y \mid Z$

回溯需要来回扫描
撤销已识别的语法结构

```
bool A() {
    if X() return true; else backtrack;
    if Y() return true; else backtrack;
    if Z() return true; else backtrack;
    return false;
}
```

递归下降分析中回溯的例子

- 文法：$S \rightarrow cAd$　$A \rightarrow ab \mid a$；输入串：cad
- 步骤
  - 调用函数S，选择唯一的产生式$S \rightarrow cAd$
  - 输入中的c与句型中的c匹配，继续调用函数A
  - A首先选择产生式$A \rightarrow ab$，a与输入的a匹配，但b和输入的d不匹配
  - 回溯并选择下一个产生式$A \rightarrow a$，a与输入的a相匹配，对函数A的调用返回到S的调用
  - $S \rightarrow cAd$中最后的d与下一个输入d匹配，结束
- 因此cad是S的句子

一个特殊的情况

- 文法： $A \rightarrow A\alpha$

```
bool A() {
    if A() …
}
```

南大编译许畅

左递归

- **左递归**的定义
  - 如果一个文法中有非终结符号A使得$A \overset{+}{\Rightarrow} A\alpha$，那么这个文法就是左递归的
- **直接左递归** (规则左递归)
  - 文法中存在一个形如$A \rightarrow A\alpha$的产生式

- 自顶向下的语法分析技术不能处理左递归的情况，因此需要消除左递归，但是自底向上的技术可以处理左递归

直接左递归的消除 (1)

- 假设非终结符号A存在直接左递归的情形

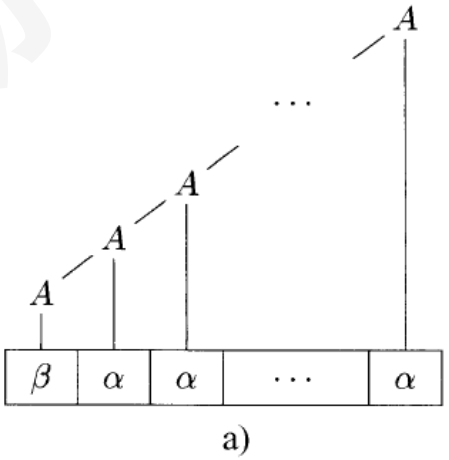
  $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$

- 可替换为

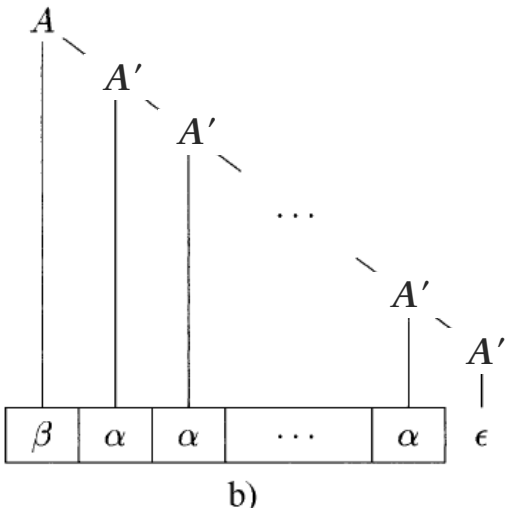
  $A \rightarrow \beta_1 A' \mid \ldots \mid \beta_n A'$

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

- 观察：由A生成的串以某个β_i开头，然后跟上零个或多个α_j



a)



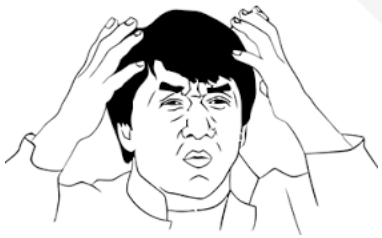
b)

$A \rightarrow A\alpha \mid \beta$ $\Longrightarrow$ $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \varepsilon$

直接左递归的消除 (2)

- 原：$A \rightarrow A\alpha \mid \beta$

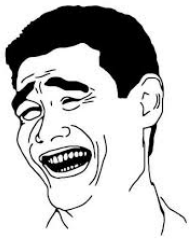
```
bool A() {
    if A() …
}
```



- 新：$A \rightarrow \beta A'$

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

```
bool A'() {
    if nextToken is α
        consume nextToken;
        return A'();
    return false;
}
```



南大编译课件

直接左递归消除示例

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

$\Longrightarrow$

$$E \rightarrow TE'$$
$$E' \rightarrow + \ TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow * \ FT' \mid \epsilon$$
$$F \rightarrow (\ E \ ) \mid \mathbf{id}$$

间接左递归

- 消除直接左递归的方法并不能消除因为多步推导而产生的间接左递归
  - 文法：$S \rightarrow Aa \mid b \quad A \rightarrow Ac \mid Sd \mid \varepsilon$
  - $S \Rightarrow Aa \Rightarrow Sda$

- 间接左递归也可消除
  - 消除算法见课本（龙书）4.3.3节。

递归下降分析的特点

- 优点
  - 易于实现（手写）
- 缺点
  - 需要回溯（影响效率）

$A \rightarrow X \mid Y \mid Z$

回溯需要来回扫描
撤销已识别的语法结构

```
bool A() {
    if X() return true; else backtrack;
    if Y() return true; else backtrack;
    if Z() return true; else backtrack;
    return false;
}
```

基于预测的语法分析

- 为输入串构造语法分析树
  - 从分析树的根结点开始，按照先根次序，深度优先地创建各个结点
  - 对应于最左推导
- 基本步骤
  - 确定对句型中最左边的非终结符号应用哪个产生式
  - 然后对该产生式与输入符号进行匹配
- 关键步骤：应用产生式
  - $A \rightarrow X_1 \mid X_2 \mid X_3 \ldots$ 如何知道应用哪个产生式?
  - 根据下一字符，预测正确的产生式，避免回溯

预测分析法简介

- 试图从开始符号推导出输入符号串
- 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则无能为力
- 例子
  - 文法：$A \rightarrow aX \mid bY \quad X \rightarrow xxx \quad Y \rightarrow yyy$
  - 输入：$axxx$
- 当两个产生式具有相同前缀时无法预测
  - 文法：$A \rightarrow aX \mid aY \quad X \rightarrow xxx \quad Y \rightarrow yyy$
  - 输入：$axxx$

需提取左公因子

提取公因子的文法变换

- 算法
  - 输入：文法G
  - 输出：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，找出它的两个或者多个可选产生式体之间的最长公共前缀
    - $A \rightarrow \alpha\beta_1 \mid \ldots \mid \alpha\beta_n \mid \gamma$
    - $A \rightarrow \alpha A' \mid \gamma$　　$A' \rightarrow \beta_1 \mid \ldots \mid \beta_n$
    - 其中γ是不以α开头的产生式体

预测分析法：LL(k)

- L: left-to-right 从左到右扫描
- L: left-most 最左推导
- k: 向前看k个符号
  - 实践当中，通常k=1，即LL(1)

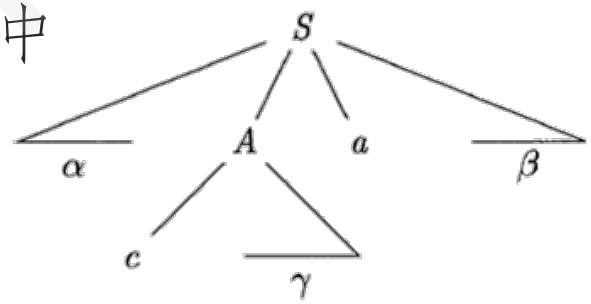
- 每次为最左边的非终结符号选择产生式时，向前看1个输入符号，预测要使用的产生式

FIRST和FOLLOW

- 在基于预测的分析技术中，使用向前看几个符号来确定产生式 (通常只看一个符号)
- 当前句型是$xA\beta$，而输入是xa …，那么选择产生式$A \rightarrow \alpha$的必要条件是下列之一
  - $\alpha \overset{*}{\Rightarrow}$ a …
  - $\alpha \overset{*}{\Rightarrow}$ ε，且β以a开头，即在某个句型中a跟在A之后
  - 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

- FIRST(α)
  - 可以从α推导得到串的首符号的集合
  - 如果$\alpha \overset{*}{\Rightarrow} \varepsilon$，那么$\varepsilon$也在FIRST($\alpha$)中



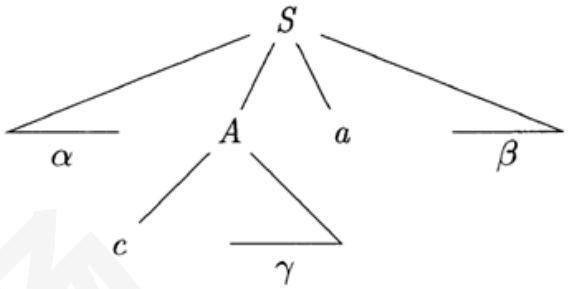
终结符号 c 在 FIRST(A)

- FIRST函数的意义
  - A的产生式 $A \rightarrow \alpha \mid \beta$，且FIRST($\alpha$)和FIRST($\beta$)不相交
  - 下一个输入符号是a，若$a \in$ FIRST(α)，则选择$A \rightarrow \alpha$，若$a \in$ FIRST(β)，则选择$A \rightarrow \beta$

FIRST的计算方法

- 计算FIRST(X)
  - X是终结符号，那么添加X
  - X是非终结符号，且$X \rightarrow Y_1Y_2\ldots Y_k$是产生式
    - 如果a在FIRST(Y_i)中，且ε在FIRST(Y_1), … , FIRST(Y_{i-1})中，那么也添加a
    - 如果ε在FIRST(Y_1), … , FIRST(Y_k)中，那么也添加ε
  - X是非终结符号且$X \rightarrow$ ε，那么也添加ε
- 计算FIRST($X_1X_2\ldots X_n$)
  - 加入FIRST(X_1)中所有非ε符号
  - 若ε在FIRST(X_1)中，加入FIRST(X_2)中所有非ε符号 …
  - 若ε在所有FIRST(X_i)中，也加入ε

FOLLOW



a 在 FOLLOW(A)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
  - 例如：$S \rightarrow \alpha A a \beta$，终结符号a ∈ FOLLOW(*A*)

- FOLLOW函数的意义
  - 如果$A \rightarrow \alpha$，当$\alpha \Rightarrow \varepsilon$时，FOLLOW(*A*)可以帮助我们选择恰当的产生式
  - 例如：$A \rightarrow \alpha$，而b属于FOLLOW(*A*)，如果$\alpha \Rightarrow \varepsilon$，而当前输入符号是b，则可以选择$A \rightarrow \alpha$，因为*A*最终到达了ε，而且后面跟着b

FOLLOW的计算方法

- 算法
  - 将**输入结束标记**$加入FOLLOW($S$)中
  - 按照下面两个规则不断迭代，直到所有的FOLLOW集合都不再增长为止
    - 如果存在产生式$A \rightarrow \alpha B\beta$，那么FIRST($\beta$)中所有非ε的符号都加入FOLLOW($B$)中
    - 如果存在一个产生式$A \rightarrow \alpha B$，或者$A \rightarrow \alpha B\beta$且FIRST($\beta$)包含ε，那么FOLLOW($A$)中所有符号都加入FOLLOW($B$)中

FIRST/FOLLOW的例子 (1)

- 文法
  - $E \rightarrow T\ E'$　　$E' \rightarrow +\ T\ E' \mid \varepsilon$
  - $T \rightarrow F\ T'$　　$T' \rightarrow *\ F\ T' \mid \varepsilon$　　$F \rightarrow (\ E\ ) \mid \mathrm{id}$
- FIRST集合的计算
  - FIRST(F) = { (, id }
  - FIRST(T) = FIRST(F) = { (, id }
  - FIRST(E) = FIRST(T) = { (, id }
  - FIRST(E') = { +, ε }　　FIRST(T') = { *, ε }
- 由此可推出各个产生式右部的FIRST集合
  - 如：FIRST($T\ E'$) = FIRST(T) = { (, id }

FIRST/FOLLOW的例子 (2)

- 文法
  - $E \rightarrow T\ E'$ $\quad E' \rightarrow +\ T\ E' \mid \varepsilon$
  - $T \rightarrow F\ T'$ $\quad T' \rightarrow *\ F\ T' \mid \varepsilon$ $\quad F \rightarrow (\ E\ ) \mid \text{id}$
- FOLLOW集合的计算
  - FOLLOW(E) = { \$,) } // E是开始符号
  - FOLLOW(E') = { \$,) } // $E \rightarrow T\ E'$
  - FOLLOW(T) = { +, \$,) } // $E' \rightarrow +\ T\ E'$
  - FOLLOW(T') = { +, \$,) } // $T \rightarrow F\ T'$
  - FOLLOW(F) = { *, +, \$,) } // $T \rightarrow F\ T'$

LL(1)文法 (1)

L: left-to-right 从左到右扫描
L: left-most 最左推导
1: 向前看1个符号

- 定义：对文法的任意两个产生式$A \rightarrow \alpha \mid \beta$
  - 不存在终结符号a使得α和β都可推导出以a开头的串
  - α和β最多只有一个可推导出空串
  - 如果β可推导出空串，那么α不能推导出以FOLLOW(A)中任何终结符号开头的串
- 等价于
  - FIRST(α)与FIRST(β)不相交(条件一、二)
  - 如果$\varepsilon \in$ FIRST(β)，即$\beta \stackrel{*}{\Rightarrow} \varepsilon$，那么FOLLOW($A$)与FIRST($\alpha$)与不相交；反之亦然 (条件三)

避免冲突，只看1个符号即可唯一、正确地预测产生式

LL(1)文法 (2)

- 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
  - 产生式：*stmt* → **if (exp)** *stmt* **else** *stmt*
    | **while (exp)** *stmt*
    | **a**
  - 输入：**if**[1] **(exp) while**[2] **(exp) a**[3] **else a**
    1. 首先根据输入**if**，选择产生式*stmt* → **if (exp)** *stmt* **else** *stmt*，得到句型**if (exp)** *stmt* **else** *stmt*
    2. 然后根据输入**while**，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*
    3. 再根据输入**a**，选择产生式开下个*stmt* → **a**，得到**if (exp) while (exp) a else** *stmt* …

预测分析表构造算法

- 输入：文法G
- 输出：预测分析表M
  - 二维表，非终结符 (行) × 终结符 (列) → 产生式
  - 展开非终结符时，根据输入终结符选择相应产生式
- 方法
  - 对于文法G的每个产生式$A \rightarrow \alpha$
    - 对于FIRST(α)中的每个终结符号a，将$A \rightarrow \alpha$加入到$M[A, a]$中
    - 如果ε在FIRST(α)，那么对于FOLLOW(A)中的每个符号b，将$A \rightarrow \alpha$也加入到$M[A, b]$中
  - 最后在所有的空白条目中填入error

预测分析表的例子

- 文法：$E \to TE'$　　$E' \to +TE' \mid \varepsilon$
  $T \to FT'$　　$T' \to *FT' \mid \varepsilon$　　$F \to (E) \mid \mathrm{id}$
- FIRST集
  - E, T, F: { (, id }　E': { +, ε }　T': { *, ε }
- FOLLOW集
  - E, E': { $,) }　T, T': { +, $,) }　F: { *, +, $,) }

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

预测分析表冲突的例子

- 文法： $S \rightarrow$ **i** E **t** S S' | a　$S' \rightarrow$ **e** S | ε　$E \rightarrow$ b
  - FIRST(eS) = { e }，使得$S' \rightarrow e S$在$M[S', e]$中
  - FOLLOW(S') = { $, e }，使得$S' \rightarrow \varepsilon$也在$M[S', e]$中
- LL(1)文法必然不是二义性的，而这个文法是二义性的

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		$E \rightarrow b$				

LL(1)文法的递归下降分析

- 递归下降语法分析程序由一组过程组成
- 每个非终结符号对应于一个过程，该过程负责扫描该非终结符号对应的结构
- 可以使用当前的输入符号来唯一地选择产生式

$A \rightarrow X \mid Y \mid Z$

```
bool A() {
    p = M[A, nextToken]; // 查表取出相应产生式
    if p is A → X return X();
    if p is A → Y return Y();
    if p is A → Z return Z();
    return false;
}
```

无回溯递归下降

非递归的预测分析 (1)

- 递归下降分析的输入（程序）太大时，容易导致（调用）栈溢出
- 可以用栈模拟递归过程，实现非递归的分析
- 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，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)

- 分析时的处理过程
  - 初始化时，栈中仅包含开始符号S (和$)
  - 如果栈顶元素是终结符号，那么跟输入进行匹配
  - 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用同样的驱动程序

分析表驱动的预测分析器

- 栈中符号序列为α，w是已读入的输入，w'是余下的输入，那么
  - S推导出$w\alpha$
  - 试图从α推导出余下的输入w'
- 预测分析程序使用$M[X, a]$来扩展X，将产生式的右部按倒序压入栈中

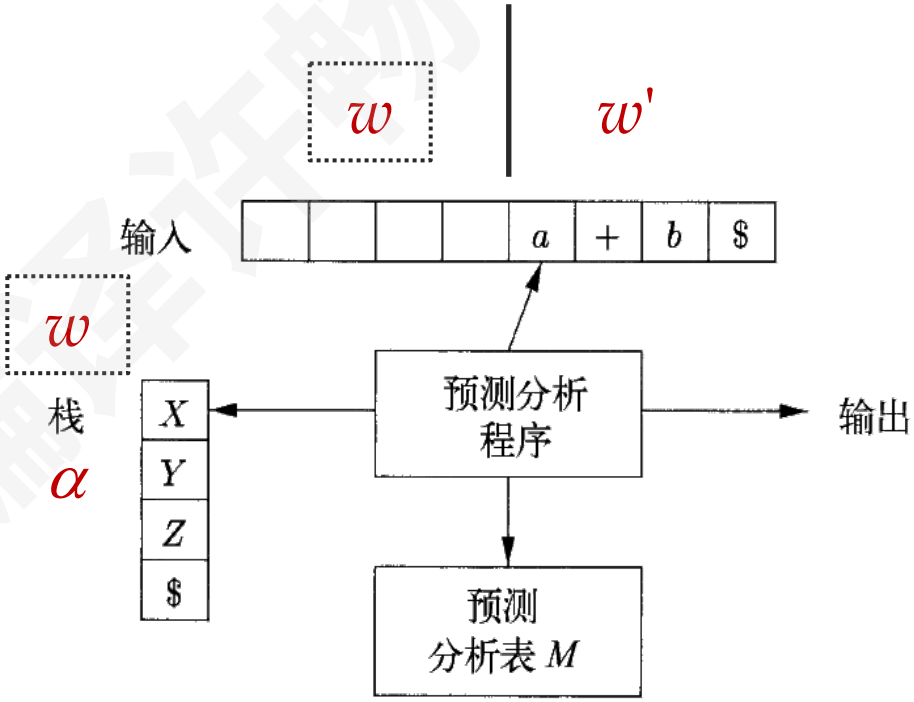


图 4-19　一个分析表驱动的预测分析器的模型

预测分析算法

- 输入：串w，预测分析表M
- 输出：如果w是句子，输出w的最左推导；否则报错

(1) 初始化：输入缓冲区中为w\$，栈中为S\$，ip指向w的第一个符号

(2) 令X = 栈顶符号，ip指向输入符号a

if ($X == a$) X出栈，ip向前移动 // 与终结符号匹配成功

else if (X是终结符号) error() // 失配

else if ($M[X, a]$是报错条目) error() // 无适当的产生式

else if ($M[X, a] = X \rightarrow Y_1Y_2\ldots Y_k$) {

输出产生式$X \rightarrow Y_1Y_2\ldots Y_k$ // 这一步可构造语法树

弹出栈顶符号X，并将$Y_k, Y_{k-1}, \ldots, Y_1$压入栈中

}

这一步代替了递归下降分析法“递归”的步骤

(3) 不断执行第二步，直到要么报错，要么栈中为空

分析表驱动预测分析的例子 (1)

- 输入：
  id + id * id

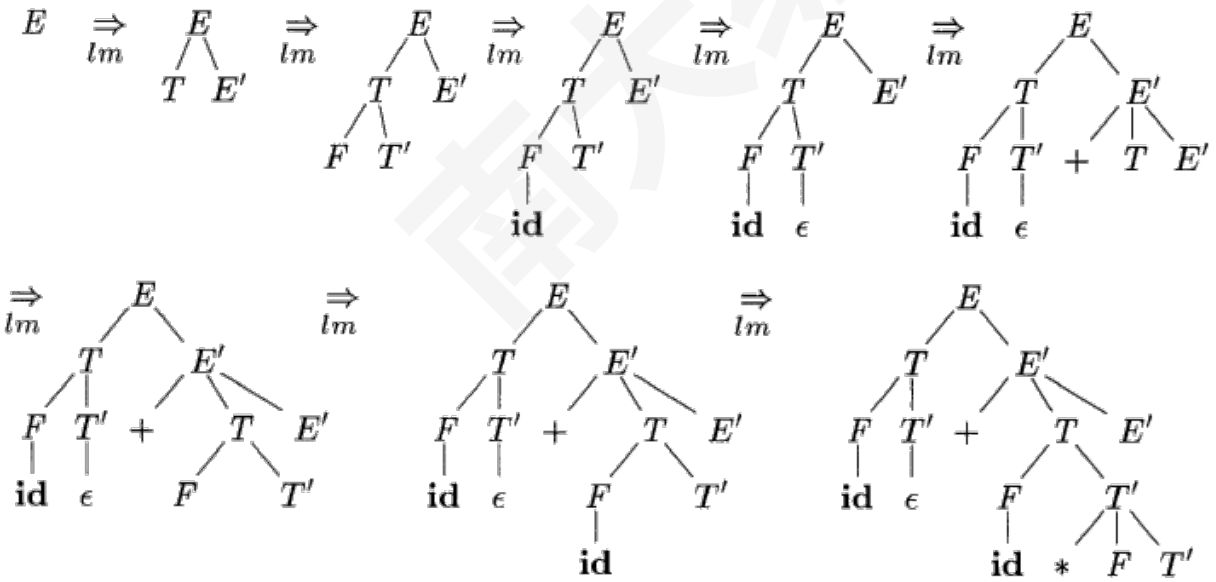
非终结符号	输入符号					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathbf{id}$			$F \to (E)$		

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

分析表驱动预测分析的例子 (2)

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

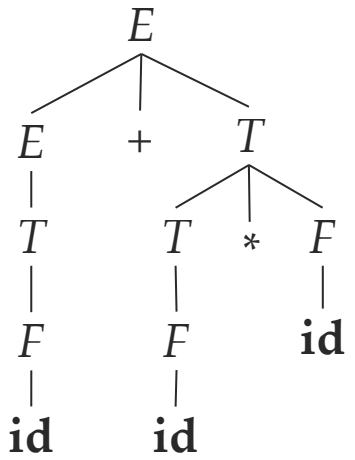
已匹配部分加上栈中符号必然是一个最左推导过程中的句型



自顶向下语法分析的局限

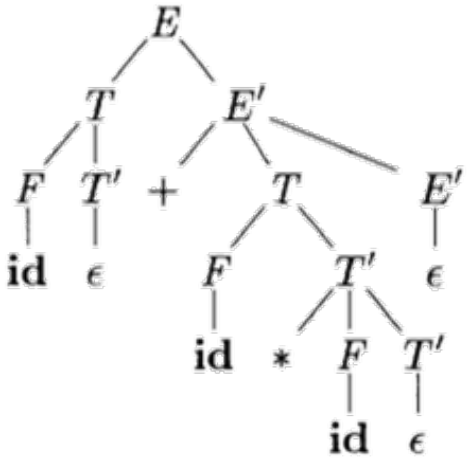
- 根据极为有限的信息预测(猜)产生式
- 能识别的文法有局限性
  - 需要改造文法消除左递归
  - 改造后的文法不直观
  - 生成的语法树不易理解和处理

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



- ✓ 反过来：自底向上
  - 先看足够输入，再选择产生式

$$E \to TE'$$
$$E' \to + \ TE' \mid \epsilon$$
$$T \to FT'$$
$$T' \to * \ FT' \mid \epsilon$$
$$F \to (\ E\ ) \mid \mathbf{id}$$



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
  - 在实际的语法分析过程中并不一定会构造出相应的分析树，但是用分析树的概念可以方便理解
- 重要的自底向上语法分析的通用框架
  - 移入-归约 (shift-reduce)
- 多种LR技术
  - 简单LR技术 (Simple LR)
  - 向前看LR技术(Look-Ahead LR)
  - …

分析过程示例

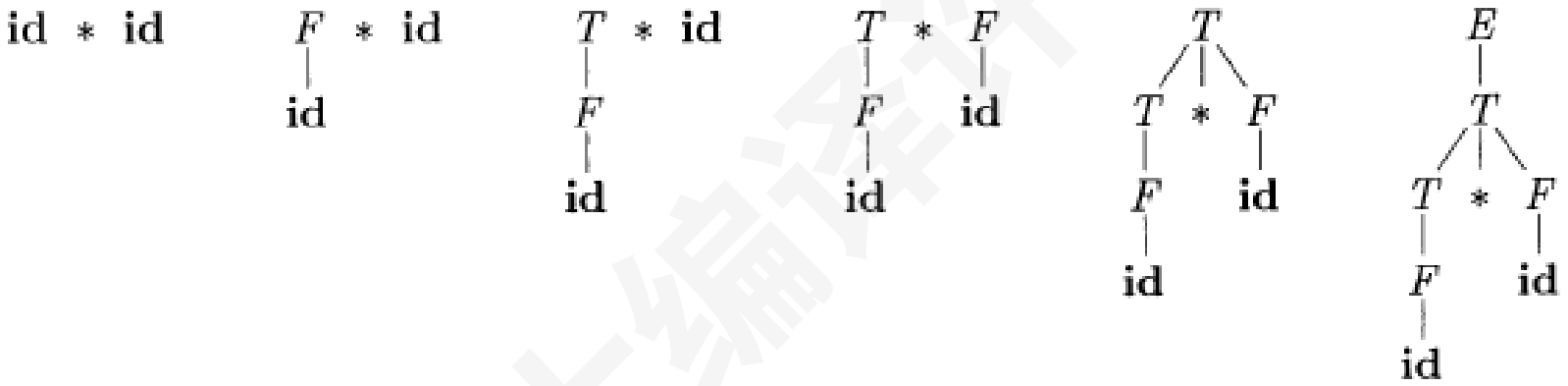


图 4-25　**id** * **id** 的自底向上分析过程

归约

- 自底向上的语法分析过程可以看成是从串w归约为文法开始符号S的过程 $S \overset{*}{\Rightarrow} w$
- **归约**步骤
  - 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
- 问题
  - 何时归约 (归约哪些符号串)?
  - 归约到哪个非终结符号?

归约的例子

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

- id * id的归约过程
  - id * id，$\underline{F}$ * id，$\underline{T}$ * id，T * $\underline{F}$，$\underline{T}$，$\underline{E}$

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

归约过程等于一个反向最右推导

- 对于句型T * id，有两个子串和某产生式右部匹配
  - T是$E \rightarrow T$的右部
  - id是$F \rightarrow$ id的右部
  - 为什么选择将id归约为F，而不是将T归约为E?
    - 原因：T归约为E之后，E * id不再是句型
    - 问题：如何确定这一点?

句柄

- 最右句型中和某个产生式体 (右部) 相匹配的子串，对它的归约代表了该最右句型的最右推导的最后一步

- 正式定义：对于最右句型$\alpha\beta w$，如果$S \overset{*}{\underset{rm}{\Longrightarrow}} \alpha A w \underset{rm}{\Longrightarrow} \alpha\beta w$，则$\beta$是$A \rightarrow \beta$的一个句柄 (方便起见，通常直接称$\beta$为句柄)
  - 在一个最右句型中，句柄右边只有终结符号
  - 如果文法是无二义性的，那么每个句型有且只有一个句柄

- 自底向上分析的过程就是识别和归约句柄的过程

句柄的例子

- 输入：id * id

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

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

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$
- 结束时刻：栈中为S，而输入为$
- 在分析过程中，不断移入符号，并在识别到句柄时进行归约
- 句柄被识别时总是出现在栈的顶部

主要分析动作

- **移入**：将下一个输入符号移入到栈顶
- **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

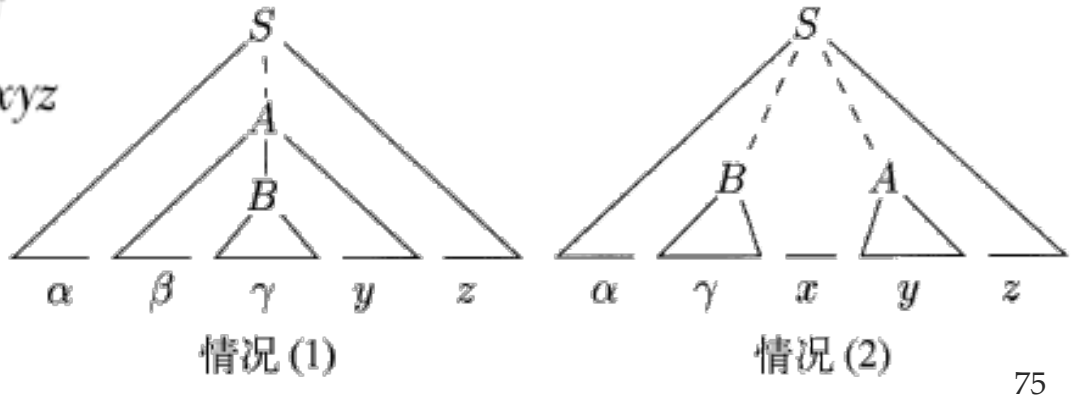
为什么句柄总是在栈顶?

- 为什么每次归约得到的新句型的句柄仍在栈顶?
- 考虑最右推导的两个连续推导的两种情况
  - 情况1) 嵌套：A被替换为βBy，然后产生式体中的最右非终结符号B被替换为γ (归约之后句柄为βBy)
  - 情况2) 非嵌套：A首先被展开，产生式体中只包含终结符号，下一个最右非终结符号B位于y左侧

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

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

直觉：如果一个句柄出现在栈中/底，那它之前就应该被归约了



移入-归约分析中的冲突

- 对于有些不能使用移入-归约分析的文法，不管用什么样的移入-归约分析器都会到达这样的格局
  - 即使知道栈中所有内容、以及后续k个输入符号，仍然
    - 无法知道是否该进行归约 (**移入-归约冲突**)，或
    - 无法知道按照什么产生式进行归约 (**归约-归约冲突**)
  - 设栈中符号串是$\alpha\beta$，接下来的k个符号是x，产生移入/归约冲突的原因是存在y和y'使得$a\beta xy$是最右句型且β是句柄 (需归约)，而$a\beta xy'$也是最右句型，但是句柄还在右边 (需移入)

移入-归约冲突的例子

stmt → **if** *expr* **then** *stmt*

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

| **other**

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

归约-归约冲突的例子

- 输入为id (id , id)
- 冲突时的格局
  - 栈：… id (id　　输入：, id) …

(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

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增大时，相应的语法分析器的规模急剧增大
  - $k=2$时，程序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
  - 我们只考虑$k<=1$的情况

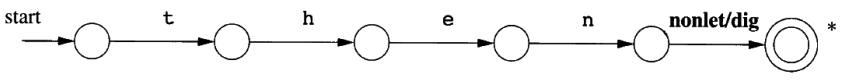
从最简单的LR(0)开始

LR语法分析器的优点

- 由表格驱动，虽然手工构造表格工作量很大，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语言的LR语法分析器
- 最通用的无回溯移入-归约分析技术
- 能分析的文法比LL(k)文法更多

LR语法分析思路 (1)

- 核心任务：识别句柄，归约句柄
  - 句柄：栈中内容（已扫描/归约的串）的最右可归约子串
- 关键问题：如何知道栈中内容可以归约了？
  - 维护一个状态，记录当前句柄识别的进度
- 项：一个文法产生式加上在其中某处的一个点
  - $A \rightarrow \cdot XYZ$，$A \rightarrow X \cdot YZ$，$A \rightarrow XY \cdot Z$，$A \rightarrow XYZ \cdot$
  - $A \rightarrow \alpha \cdot \beta$表示已扫描/归约到了$\alpha$，并期望在接下来的输入中经过扫描/归约得到$\beta$，然后把$\alpha\beta$归约到$A$
  - $A \rightarrow \alpha\beta \cdot$表示已扫描/归约得到了$\alpha\beta$，可以把$\alpha\beta$归约为$A$
  - 注意：$A \rightarrow \varepsilon$只对应一个项$A \rightarrow \cdot$
  - 类似词法分析DFA的状态



通常被称为LR(0)项

LR语法分析思路 (2)

- 项：一个文法产生式加上在其中某处的一个点
  - 类似词法分析DFA的状态
- 一个项读入一个符号后，可变为另一个项
  - $A \rightarrow \cdot xyz$ 读入x变为 $A \rightarrow x \cdot yz$，类似状态间的跳转

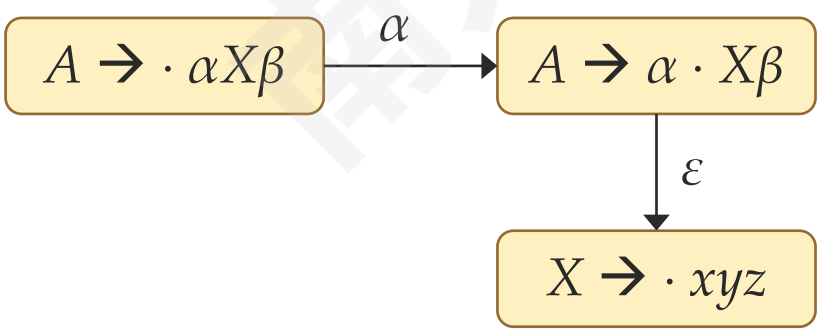
状态 + 跳转 → 自动机

- 文法产生式是有限的，每个产生式右部的长度也是有限的
  - 项的数量也是有限的
- 同一时刻只识别一个句柄，因此只需关注一个项

有穷自动机，被称为LR(0)自动机

LR语法分析思路 (3)

- 项即表示过去（已扫描/归约的串），也预示未来（想要扫描/归约的串）
  - $A \rightarrow \alpha \cdot \beta$表示已扫描/归约到了$\alpha$，并期望在接下来的输入中经过扫描/归约得到$\beta$，然后把$\alpha\beta$归约到$A$
- 识别句柄的过程中，可能有多个可选的产生式，因此可能同时满足多个项
  - 同时处于多个状态 NFA



LR(0)项集规范族的构造

- 三个相关定义
  - 增广文法（引入起始状态）
  - 项集闭包CLOSURE（状态机确定化）
  - GOTO函数（定义状态跳转）

- **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$S' \rightarrow S$而得到的
  - 显然G'和G接受相同的语言，且按照$S' \rightarrow S$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
  - 方便引入S的项，表示识别的起始($S' \rightarrow \cdot S$)/结束($S' \rightarrow S \cdot$)

项集闭包CLOSURE

- 根据一个项（状态），扩充读取相同输入后可能同时处于的多个项（状态） 类似NFA的ε-CLOSURE
- **项集闭包** (CLOSURE)：如果I是文法G的一个项集，CLOSURE(I)就是根据下列两条规则从I构造得到的项集
  - 将I中的各项加入CLOSURE(I)中
  - 如果$A \rightarrow \alpha \cdot B\beta$在CLOSURE($I$)中，而$B \rightarrow \gamma$是一个产生式，且项$B \rightarrow \cdot\gamma$不在CLOSURE($I$)中，就将该项加入其中，不断应用该规则直到没有新项可加入
- 意义
  - $A \rightarrow \alpha \cdot B\beta$，表示希望看到由$B\beta$推导出的串，那要先看到由$B$推导出的串，因此加上$B$的各个产生式对应的项

CLOSURE(*I*)的构造算法

```
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;
}
```

项集闭包构造的例子

增广文法

- $E' \rightarrow E$　$E \rightarrow E+T \mid T$　$T \rightarrow T*F \mid F$
- $F \rightarrow (E) \mid \text{id}$

项集{ [$E' \rightarrow \cdot E$] }的闭包

- [$E' \rightarrow \cdot E$]加入闭包
- [$E \rightarrow \cdot E+T$], [$E \rightarrow \cdot T$]加入闭包
- [$T \rightarrow \cdot T*F$], [$T \rightarrow \cdot F$]加入闭包
- [$F \rightarrow \cdot (E)$], [$F \rightarrow \cdot \text{id}$]加入闭包

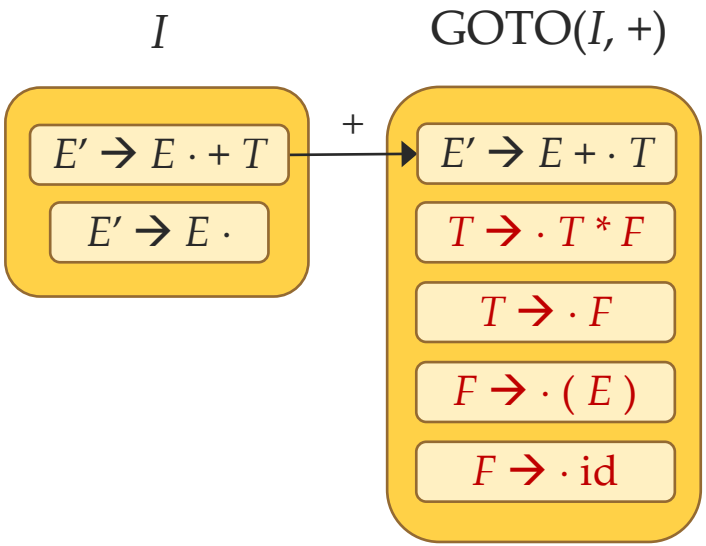
```
SetOfItems CLOSURE(I) {
    J = I;
    repeat
        for (J中的每个项 A → α·Bβ)
            for (G 的每个产生式 B → γ)
                if (项 B → ·γ 不在J中)
                    将 B → ·γ 加入J中;
    until 在某一轮中没有新的项被加入到J中;
    return J;
}
```

GOTO函数

$E' \rightarrow E$　　$E \rightarrow E+T$　　$E \rightarrow T$

$T \rightarrow T*F$　　$T \rightarrow F$　　$F \rightarrow (E)$　　$F \rightarrow \mathrm{id}$

- **GOTO函数（状态跳转表）**
  - I是一个项集，X是一个文法符号，则GOTO(I, X)定义为I中所有形如$[A \rightarrow \alpha \cdot X\beta]$的项所对应的项$[A \rightarrow \alpha X \cdot \beta]$的集合的闭包
- 例如
  - $I = \{ [E' \rightarrow E \cdot], [E \rightarrow E \cdot + T] \}$
  - GOTO(I, +)计算如下
    - I中只有一个项的点后面跟着+，对应的项为$[E \rightarrow E + \cdot T]$
    - CLOSURE($\{ [E \rightarrow E + \cdot T] \}$) = $\{$ $[E \rightarrow E + \cdot T]$, $[T \rightarrow \cdot T * F]$, $[T \rightarrow \cdot F]$, $[F \rightarrow \cdot (E)]$, $[F \rightarrow \cdot \mathrm{id}]$ $\}$

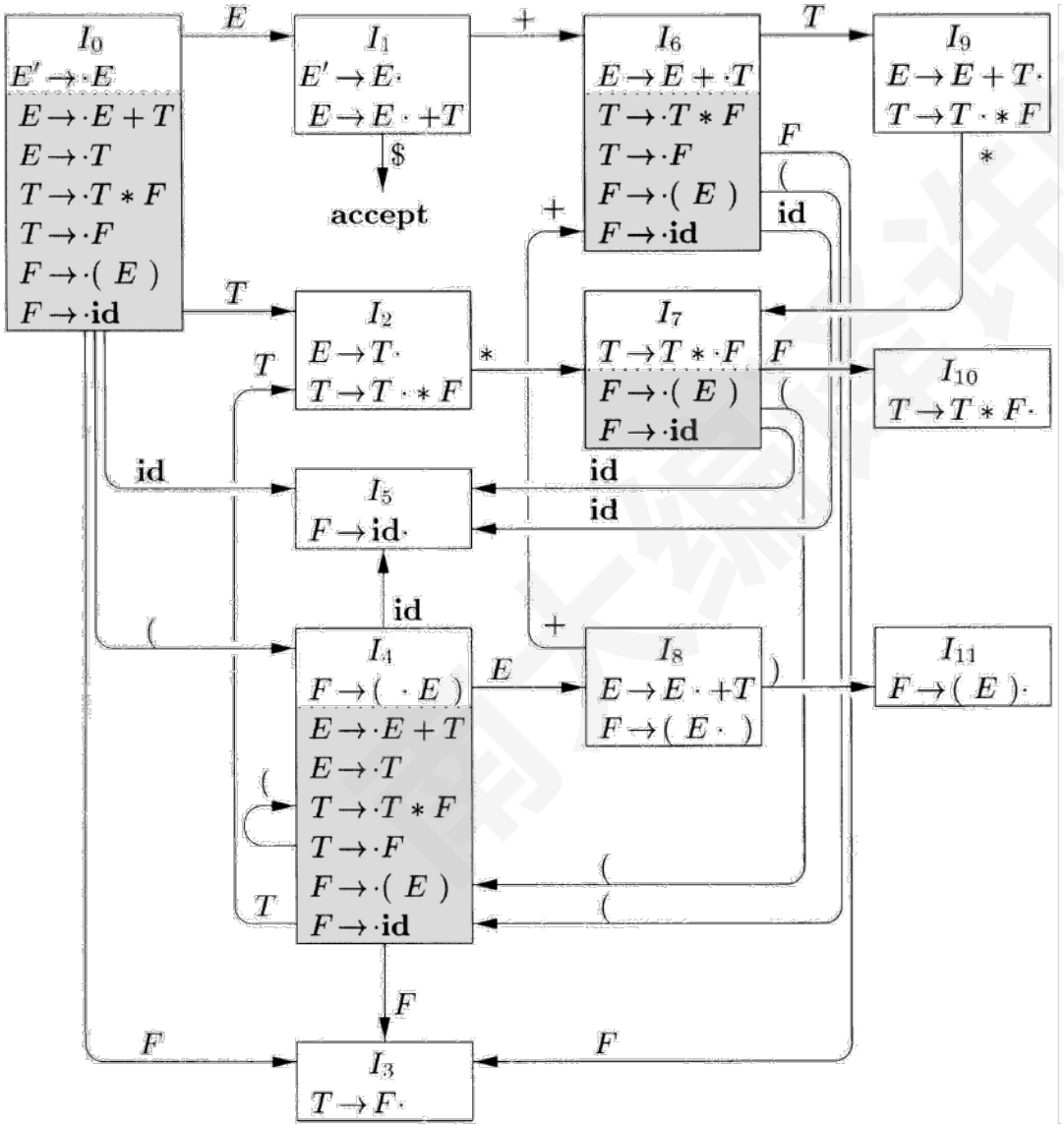


求LR(0)项集规范族的算法

- 从初始项集开始，不断计算各种可能的后继，直到生成所有的项集
  - 输入：增广文法G'
  - 输出：跳转表GOTO（以及相应项集闭包）

```
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;
}
```

项集规范族构造示例



(1) $E \to E + T$

(2) $E \to T$

(3) $T \to T * F$

(4) $T \to F$

(5) $F \to (E)$

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

LR(0)自动机的构造

- 构造方法
  - 基于规范LR(0)项集族可以构造LR(0)自动机
  - 规范LR(0)项集族中的每个项集对应于LR(0)自动机的一个**状态**
  - **状态转换**：如果GOTO(I, X) = J，则从I到J有一个标号为X的转换
  - **开始状态**为CLOSURE({ $S' \rightarrow \cdot S$ })对应的项集

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中存在项$A \rightarrow \alpha\cdot$，那么
  - α是γ的后缀，且是该句型的句柄 (对应于产生式$A \rightarrow \alpha$)
  - 表示可能找到了当前最右句型的句柄，可以归约
  - 在γ之后添加一些终结符号可以得到一个最右句型
- 如果j中存在项$B \rightarrow \alpha\cdot X\beta$，那么
  - 该句型中$\alpha X\beta$是句柄，但还没找到，还需移入
  - 在γ之后添加$X\beta$和一些终结符号可以得到一个最右句型

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
  - 已得到的文法符号序列对应于LR(0)自动机的一条路径

- 无需每次用该文法符号序列来运行LR(0)自动机
  - 文法符号可省略，由LR(0)状态可确定相应的文法符号
    - 状态间的转换是唯一的，转换边上有文法符号
  - 在移入后，根据原来的栈顶状态可以知道新的状态
    - 根据原栈顶状态和移入符号，查询GOTO
  - 在归约时，根据归约产生式的右部长度弹出相应状态，也可以根据此时的栈顶状态知道新的状态
    - 归约相当于先弹出，再移入

LR(0)自动机的作用的演示

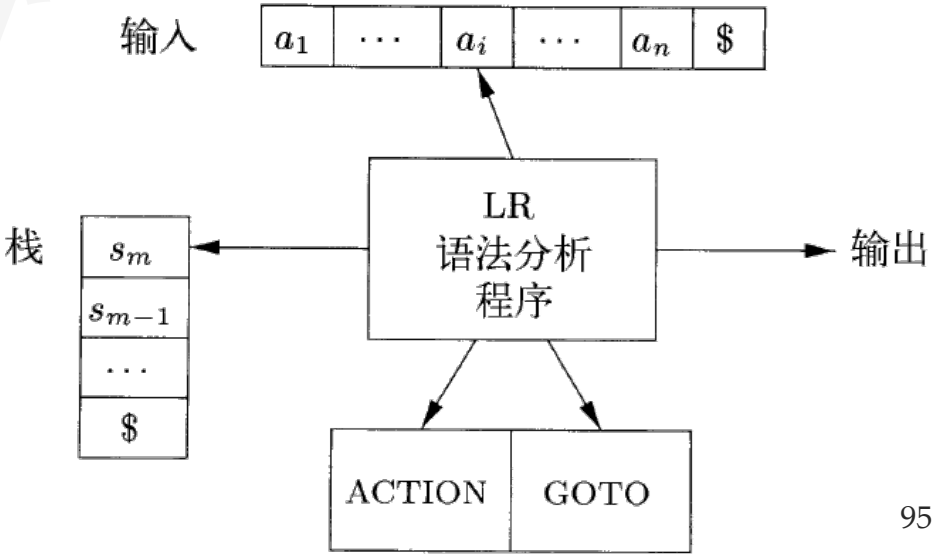
- 分析id * id
  - 栈中只保留状态，文法符号可以从相应的状态中获取

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

非必需

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
  - 分析表 = ACTION + GOTO
- 栈中存放的是状态序列，可求出相应的符号序列
- 分析程序根据栈顶状态和当前输入，通过分析表确定下一步动作



LR语法分析表的结构

- 两个部分：动作ACTION、转换GOTO
- ACTION表项有两个参数：状态i，终结符号a
  - 移入j：j是新状态，把j压入栈(同时移入a)
  - 归约$A \rightarrow \beta$：把栈顶的β归约为A(并根据GOTO表项压入新状态)
  - 接受：接受输入，完成分析
  - 报错：在输入中发现语法错误
- GOTO表项
  - 如果GOTO[I_i, A] = I_j，那么GOTO[i, A] = j

GOTO函数

GOTO表项

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 ($s_0 s_1 \ldots s_m$, $a_i a_{i+1} \ldots a_n$\$)
  - 第一个分量$s_0 s_1 \ldots s_m$是栈中的内容 (右侧s_m是栈顶)
  - 第二个分量$a_i a_{i+1} \ldots a_n$\$是余下的输入

- 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)
  - 如果状态s_{i-1}读取符号X跳转到状态s_i，那么s_i就对应于X
  - 令X_i为s_i对应的符号，那么$X_1 X_2 \ldots X_m a_i a_{i+1} \ldots a_n$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]确定相应的动作
  - 移入s：执行移入动作，将状态s(对应输入a_i)移入栈中，得到新格局($s_0s_1\ldots s_ms$，$a_{i+1}\ldots a_n$\$)
  - 归约$A \rightarrow \beta$：将栈顶的β归约为A，压入状态s，得到新格局($s_0s_1\ldots s_{m-r}s$，$a_ia_{i+1}\ldots a_n$\$)，其中r是β的长度，状态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w
- 输出：如果w在$L(G)$中，则输出自底向上语法分析过程中的归约步骤，否则输出错误指示
- 算法如下：

```
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) $E \rightarrow E + T$　　(2) $E \rightarrow T$

(3) $T \rightarrow T * F$　　(4) $T \rightarrow F$

(5) $F \rightarrow (E)$　　(6) $F \rightarrow$ id

状态	ACTION						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分析过程的例子

- 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 $F \rightarrow$ id 归约
(3)	0 3	F	* id + id $	根据 $T \rightarrow F$ 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$F \rightarrow$ id 归约
(7)	0 2 7 10	$T * F$	+ id $	根据$T \rightarrow T * F$归约
(8)	0 2	T	+ id $	根据$E \rightarrow T$ 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 $F \rightarrow$ id 归约
(12)	0 1 6 3	$E + F$	$	根据 $T \rightarrow F$ 归约
(13)	0 1 6 9	$E + T$	$	根据 $E \rightarrow E + T$归约
(14)	0 1	E	$	接受

Simple LR语法分析表的构造

- 以LR(0)自动机为基础的SLR语法分析表构造算法
  - 构造增广文法G'的LR(0)项集规范族$\{ I_0, I_1, \ldots, I_n \}$
  - 状态i对应项集I_i，相关的ACTION/GOTO表条目如下
    - $[A \rightarrow \alpha \cdot a\beta]$在$I_i$中，且$GOTO(I_i, a) = I_j$，则$ACTION[i, a]$ = "移入j"
    - $[A \rightarrow \alpha \cdot]$在$I_i$中，那么对$FOLLOW(A)$中所有$a$，$ACTION[i, a]$ = "按$A \rightarrow \alpha$归约"
    - 如果$[S' \rightarrow S \cdot]$在I_i中，那么将$ACTION[i, \$]$设为"接受"
    - 如果$GOTO(I_i, A) = I_j$，那么在GOTO表中，$GOTO[i, A] = j$
  - 空白的条目设为"error"
- 如果SLR分析表没有冲突，该文法就是SLR的

思想：把α归约成为A，后面需是$FOLLOW(A)$中的终结符号，否则只能移入

SLR分析表构造的例子

- 项集I_0

  $E' \rightarrow \cdot E$　　$E \rightarrow \cdot E+T$　　$E \rightarrow \cdot T$

  $T \rightarrow \cdot T*F$　　$T \rightarrow \cdot F$　　$F \rightarrow \cdot(E)$　　$F \rightarrow \cdot$id

  - ACTION[0, (] = s4　　ACTION[0, id] = s5
  - GOTO[0, E] = 1　　GOTO[0, T] = 2

    GOTO[0, F] = 3
- 项集I_1：　$E' \rightarrow E\cdot$　　$E \rightarrow E\cdot+T$
  - ACTION[1, +] = s6　　ACTION[1, $] = 接受
- 项集I_2：　$E \rightarrow T\cdot$　　$T \rightarrow T\cdot*F$
  - ACTION[2, *] = s7　　ACTION[2, +/)/$] = 归约$E \rightarrow T$

非SLR(1)文法的例子

- 文法
  - $S \rightarrow L = R \mid R$
  - $L \rightarrow *R \mid \text{id}$
  - $R \rightarrow L$
- 对于项集I_2
  - 第一个项使ACTION[2, =] = s6
  - 第二个项使ACTION[2, =] = 归约$R \rightarrow L$

I_0: $S' \rightarrow \cdot S$
$S \rightarrow \cdot L = R$
$S \rightarrow \cdot R$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$
$R \rightarrow \cdot L$

I_1: $S' \rightarrow S\cdot$

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

I_3: $S \rightarrow R\cdot$

I_4: $L \rightarrow *\cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_5: $L \rightarrow \mathbf{id}\cdot$

I_6: $S \rightarrow L = \cdot R$
$R \rightarrow \cdot L$
$L \rightarrow \cdot * R$
$L \rightarrow \cdot \mathbf{id}$

I_7: $L \rightarrow *R\cdot$

I_8: $R \rightarrow L\cdot$

I_9: $S \rightarrow L = R\cdot$

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[$A \rightarrow \alpha\cdot$]时，按照$A \rightarrow \alpha$进行归约的条件是下一个输入符号$x$可以在某个句型中跟在$A$之后
    - 如果此时对于x还有其它的移入/归约操作，则出现冲突
  - 假设此时栈中的符号串为$\beta\alpha$
    - 如果βAx不是任何最右句型的前缀，那么即使x在某个句型中跟在A之后，仍不应该按$A \rightarrow \alpha$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)

- $[A \rightarrow \alpha\cdot]$出现在项集中的条件
  - 首先$[A \rightarrow \cdot\alpha]$出现在某个项集中，然后逐步读入/归约到$\alpha$中的符号，点不断后移，到达末端
  - 而$[A \rightarrow \cdot\alpha]$出现的条件是$B \rightarrow \beta\cdot A\gamma$出现在项中
  - 期望首先按照$A \rightarrow \alpha$归约，然后将$B \rightarrow \beta\cdot A\gamma$中的点移到$A$之后
  - 显然，在按照$A \rightarrow \alpha$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

更强大的LR语法分析器

- **规范LR方法** (LR方法)
  - 添加项[$A \rightarrow \cdot\alpha$]时，把期望的向前看符号也加入项中(成为**LR(1)项集**)
  - 向前看符号(串)的长度即为LR(k)中的k
  - 这个做法可以充分利用向前看符号，但是状态很多

- **向前看LR** (LALR方法)
  - 基于LR(0)项集族，但每个LR(0)项都带有向前看符号
  - 分析能力强于SLR方法，且分析表和SLR分析表一样大
  - LALR已经可以处理大部分的程序设计语言

LR(1)项

- LR(1)项中包含更多信息来消除一些归约动作
- 实际的做法相当于“分裂”一些LR(0)状态，精确指明何时应该归约

- LR(1)项的形式$[A \rightarrow \alpha \cdot \beta, a]$
  - a称为**向前看符号**，可以是终结符号或者\$
  - a表示如果将来要按照$A \rightarrow \alpha\beta$进行归约，归约时的下一个输入符号必须是$a$　　$a \in \text{FOLLOW}(A)$
  - 当β非空时，移入动作不考虑a，a传递到下一状态

LR(1)项和可行前缀

- $[A \rightarrow \alpha \cdot \beta, a]$对一个可行前缀$\gamma$有效的条件
  - 存在一个推导$S \overset{*}{\underset{rm}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha\beta w$
  - 其中$\gamma = \delta\alpha$，且a是w的第一个符号，或w为空且$a = \$$
- 如果$[A \rightarrow \alpha \cdot B\beta, a]$对于可行前缀$\gamma$有效，那么
  - $[B \rightarrow \cdot\theta, b]$对于$\gamma$有效的条件是什么？
  - $S \overset{*}{\underset{rm}{\Rightarrow}} \delta A w \underset{rm}{\Rightarrow} \delta\alpha B\beta w \overset{*}{\underset{rm}{\Rightarrow}} \delta\alpha B x w \underset{rm}{\Rightarrow} \delta\alpha\theta x w$
  - b应该是xw的第一个符号，或xw为空且$b = \$$
  - 如果x为空，则$b = a$

$$b = \text{FIRST}(\beta a)$$

- 如果$[A \rightarrow \alpha \cdot X\beta, a]$对可行前缀$\gamma$有效，那么
  - $[A \rightarrow \alpha X \cdot \beta, a]$对$\gamma X$有效

LR(1)项的作用

- 形如$[A \rightarrow \alpha\cdot, a]$，辅助归约决策

- 形如$[A \rightarrow \alpha\cdot B\beta, a]$，辅助计算$[B \rightarrow \theta\cdot, b]$
  - $[A \rightarrow \alpha\cdot B\beta, a] \rightarrow [B \rightarrow \cdot\theta, b] \rightarrow [B \rightarrow \theta\cdot, b]$　$b = \mathrm{FIRST}(\beta)$
  - 产生式$A \rightarrow \alpha\cdot B\beta$的信息传递到了$[B \rightarrow \theta\cdot, b]$
  - 辅助归约$B \rightarrow \theta$

构造LR(1)项集

- **LR(1)项集族**的构造和LR(0)项集族类似，但是CLOSURE和GOTO有所不同
  - 在CLOSURE中，当由项[$A \rightarrow \alpha \cdot B\beta, a$]生成新项[$B \rightarrow \cdot\theta, b$]时，$b$必须在FIRST($\beta$a)中
  - 对LR(1)项集中的任意项[$A \rightarrow \alpha \cdot B\beta, a$]，总有：$a$在FOLLOW($A$)中，归纳法:
    - 初始项满足这个条件
    - 每次求CLOSURE项集时，新产生的项也满足这个条件

LR(1)项集的CLOSURE算法

- 注意在添加[$B \rightarrow \cdot\gamma, b$]时，$b$的取值范围

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

LR(0)项集闭包:

if (项 $B \rightarrow \cdot\gamma$ 不在 J 中)
    将 $B \rightarrow \cdot\gamma$ 加入 J 中;

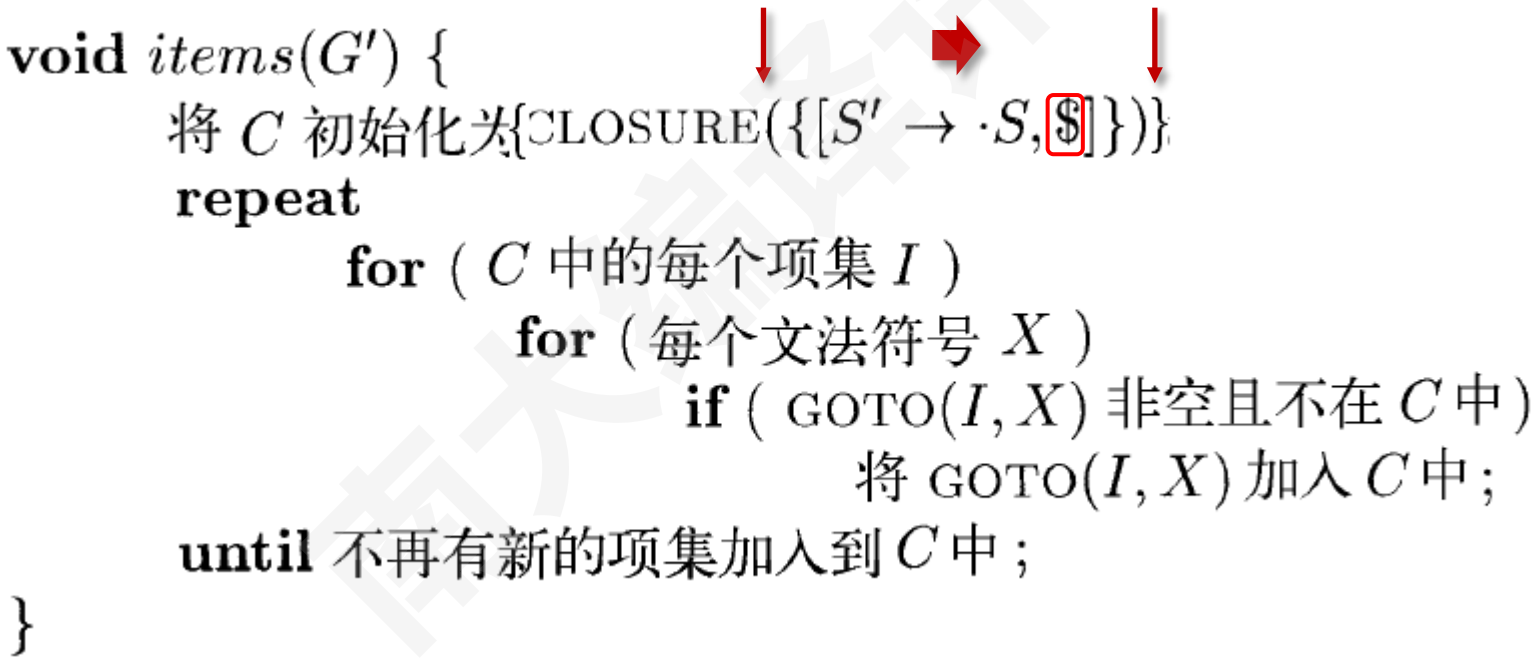
LR(1)项集的GOTO算法

- 和LR(0)项集的GOTO算法基本相同

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)项集族的构造算法

- 和LR(0)项集族的构造算法相同



LR(1)项集族例子

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;

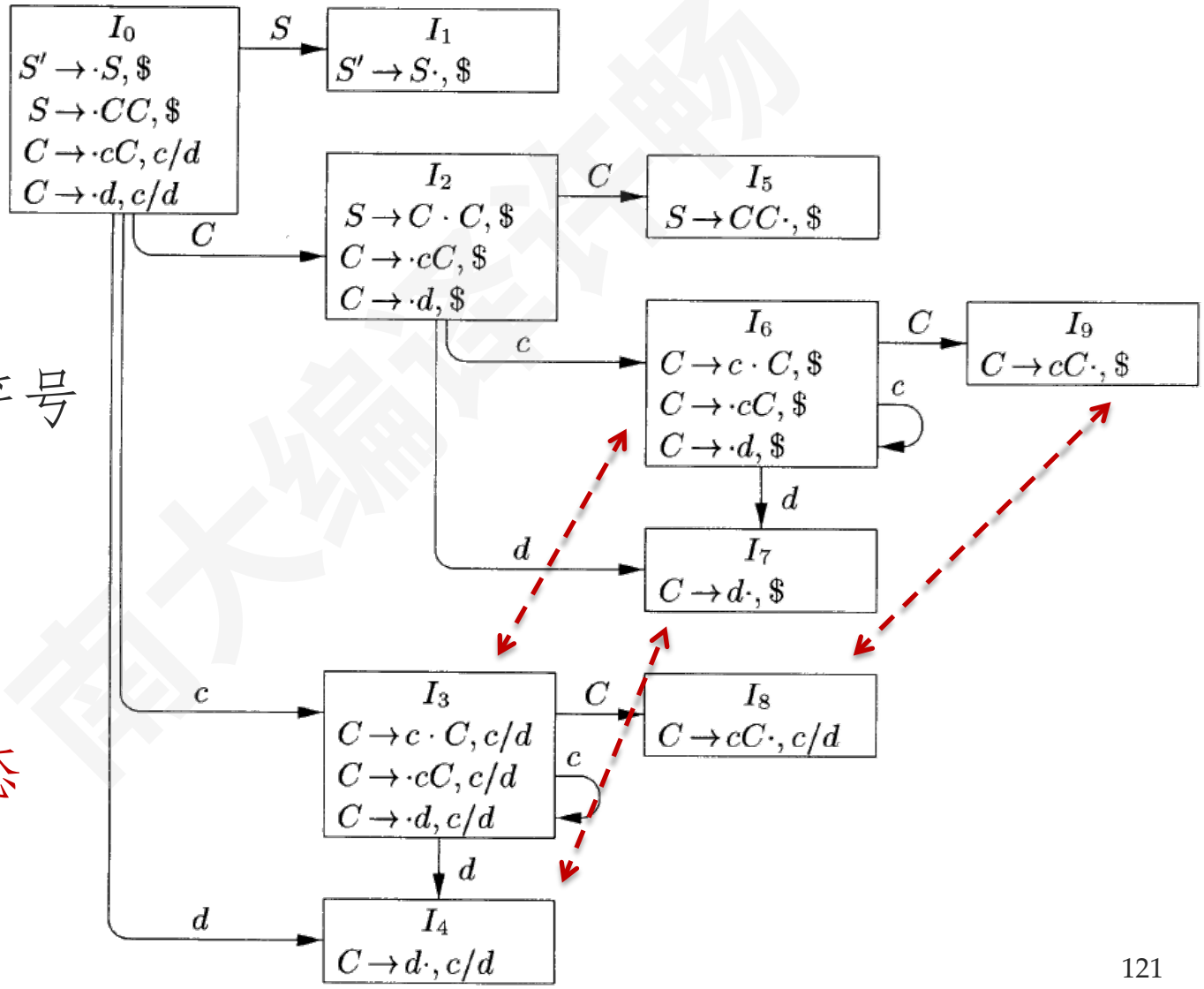
- 增广文法
  - $S' \to S$　　$S \to C\ C$　　$C \to c\ C \mid d$
- 构造I_0项集和GOTO函数
  - I_0 = CLOSURE{ [$S' \to \cdot S$, \$] } =
    - { [$S' \to \cdot S$, \$], [$S \to \cdot CC$, \$], [$C \to \cdot cC$, c/d], [$C \to \cdot d$, c/d] }
  - GOTO(I_0, S) = { [$S' \to S\cdot$, \$] }
  - GOTO(I_0, C) = CLOSURE{ [$S \to C\cdot C$, \$] } =
    - { [$S \to C\cdot C$, \$], [$C \to \cdot cC$, \$], [$C \to \cdot d$, \$] }
  - GOTO(I_0, c) = CLOSURE{ [$C \to c\cdot C$, c/d] } =
    - { [$C \to c\cdot C$, c/d], [$C \to \cdot cC$, c/d], [$C \to \cdot d$, c/d] }
  - GOTO(I_0, d) = { [$C \to d\cdot$, c/d] }

LR(1)项集的GOTO图

不计向前看符号

- I_3, I_6相同
- I_4, I_7相同
- I_8, I_9相同

共10个状态



LR(1)语法分析表的构造

- 步骤
  - 构造得到LR(1)项集族$C' = \{ I_0, I_1, \ldots, I_n \}$
  - 状态i对应于项集I_i，其分析动作如下
    - $[A \rightarrow \alpha \cdot a\beta, b]$在项集中，且$GOTO(I_i, a) = I_j$，那么$ACTION[i, a] =$ "移入j"
    - $[A \rightarrow \alpha \cdot, a]$在项集中，那么$ACTION[i, a]$ = "按照$A \rightarrow \alpha$归约"

SLR：$[A \rightarrow \alpha \cdot]$在$I_i$中，对FOLLOW($A$)中所有$a$，$ACTION[i, a]$ = "按$A \rightarrow \alpha$归约"

    - $[S' \rightarrow S \cdot, \$]$在项集中，那么$ACTION[i, \$]$ = "接受"
  - GOTO表项：$GOTO[i, A] = j$，如果$GOTO(I_i, A) = I_j$
  - 没有填写的条目为error
  - 如果条目有冲突，则不是LR(1)的
  - **初始状态**对应于$[S' \rightarrow \cdot S, \$]$所在的项集

LR(1)语法分析表的例子

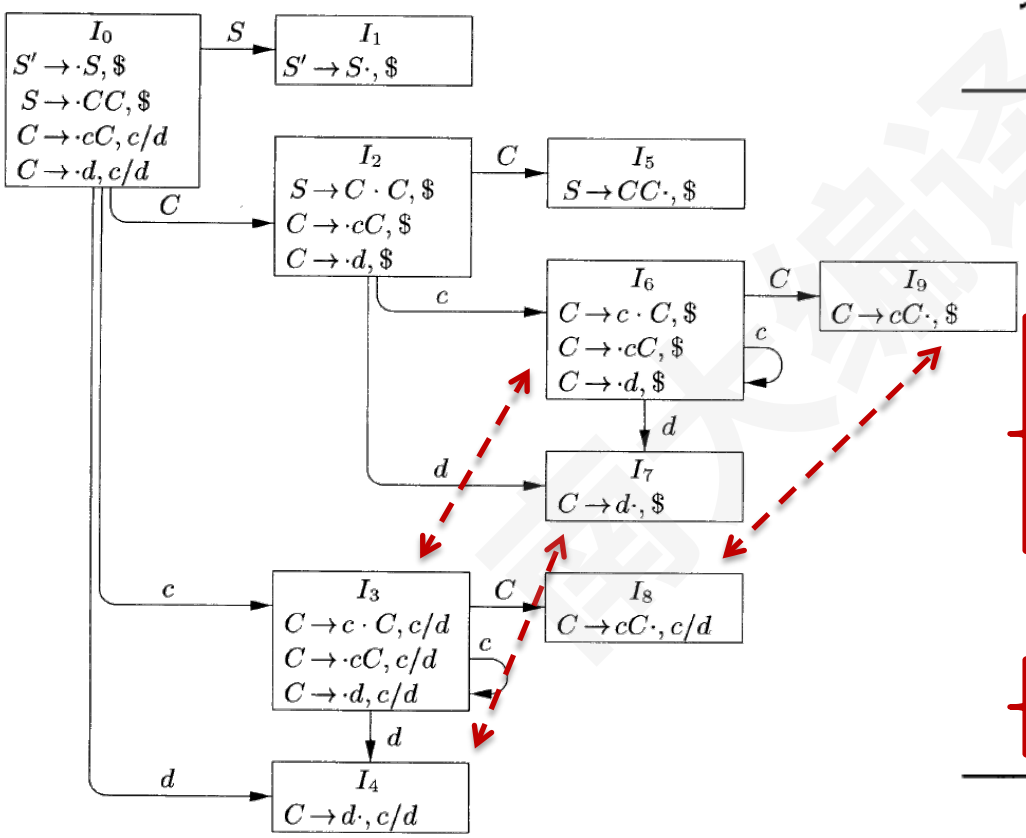
文法：

$S' \rightarrow S$

$S \rightarrow C\ C$

$C \rightarrow c\ C \mid d$

- (3, 6), (4, 7), (8, 9)可看作是由原来的一个LR(0)状态拆分而来



状态	ACTION			GOTO	
	c	d	$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

构造LALR语法分析表

- SLR(1)语法分析表的分析能力较弱
- LR(1)语法分析表的状态数量很大

- LALR(1)是实践中常用的方法
  - 状态数量和SLR(1)的状态数量相同
  - 能够方便地处理大部分常见程序设计语言的构造

LR(1)语法分析表的合并

- 4和7在向前看符号上不同
  - [C → d·, c/d] vs. [C → d·, $]
  - 状态4：下个符号为c/d则归约，为$则报错
  - 状态7：分析动作正好相反
- 如果将4和7中的项合并得47，则所有情况下都归约
  - 对这个文法，合并不会引起冲突，但有些文法会

状态	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

文法：

$S' \rightarrow S$

$S \rightarrow C\ C$

$C \rightarrow c\ C \mid d$

LALR分析技术的基本思想

- 寻找具有相同核心的LR(1)项集，并把它们合并成为一个项集
  - 项集的**核心**就是项的第一分量的集合
    - I_4和I_7的核心都是{ $C \rightarrow d\cdot$ }
    - I_3和I_6的核心{ $C \rightarrow c\cdot C$, $C \rightarrow \cdot cC$, $C \rightarrow \cdot d$ }
- 一个LR(1)项集的核心是一个LR(0)项集

- GOTO(I, X)的核心只由I的核心决定，因此被合并项集的GOTO目标也可以合并
  - 这表示合并之后，我们仍可以建立GOTO关系

合并引起的冲突

- 原来无冲突的LR(1)分析表在合并之后得到LALR(1)分析表，新表中可能存在冲突
- 合并不会导致移入/归约冲突
  - 假设合并之后在a上存在移入/归约冲突，即存在项$[B \rightarrow \beta \cdot a\gamma, _]$和$[A \rightarrow \alpha \cdot, a]$
  - 因为被合并的项集具有相同的核心，因此被合并的所有项集中都包括$[B \rightarrow \beta \cdot a\gamma, _]$，而$[A \rightarrow \alpha \cdot, a]$也必然在某个项集中，那么这个项集必然已经存在冲突！
- 合并会引起归约/归约冲突，即不能确定按照哪个产生式进行归约

合并引起归约/归约冲突的例子

- 文法：
  - $S' \rightarrow S$　　$S \rightarrow a\,A\,d \mid b\,B\,d \mid a\,B\,e \mid b\,A\,e$
  - $A \rightarrow c$　　$B \rightarrow c$
- 语言：{ acd, bcd, ace, bce }

- 可行前缀ac的有效项集：{ $[A \rightarrow c\cdot, d]$, $[B \rightarrow c\cdot, e]$ }
- 可行前缀bc的有效项集：{ $[B \rightarrow c\cdot, d]$, $[A \rightarrow c\cdot, e]$ }
- 合并之后的项集为：{ $[A \rightarrow c\cdot, d/e]$, $[B \rightarrow c\cdot, d/e]$ }
  - 包含了归约/归约冲突：应该把c归约成为A还是B?

LALR分析表构造算法

- 步骤
  - 构造得到LR(1)项集族$C = \{ I_0, I_1, \ldots, I_n \}$
  - 对于LR(1)中的每个核心，找出所有具有该核心的项集，并把这些项集替换为它们的并集
  - 令$C' = \{ J_0, J_1, \ldots, J_m \}$为得到的LR(1)项集族
    - 按照LR分析表的构造方法得到ACTION表 (如果存在冲突，则这个文法不是LALR的)
  - GOTO表的构造：设J是一个或者多个LR(1)项集 (包括I_1)的并集，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
- 得到的分析表称为**LALR语法分析表**

LALR分析表的例子

文法：

$S' \rightarrow S$

$S \rightarrow C\ C$

$C \rightarrow c\ C \mid d$

- LR(1)项集族中有三对可以合并
  - I_{36}: [C → c·C, c/d/$], [C → ·cC, c/d/$], [C → ·d, c/d/$]
  - I_{47}: [C → d·, c/d/$]
  - I_{89}: [C → cC·, c/d/$]
- GOTO(I_{36}, C) = I_{89} (原GOTO(I_3, C) = I_8)

状态	ACTION			GOTO	
	c	*d*	$	*S*	*C*
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

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

- 处理语法正确的输入时，LALR语法分析器和LR语法分析器的动作序列完全相同
  - 栈中的状态名字不同，但是状态序列之间有对应关系
  - 如果LR分析器压入状态I，那么LALR分析器压入I对应的合并项集
    - 当LR分析器压入状态I_3时，LALR分析器压入状态I_{36}

- 当处理错误的输入时，LALR可能多执行一些归约动作，但不会多移入一个符号

LALR技术本质

- 对LR(1)项集规范族中的同核心项集进行合并
  - 使得分析表保持了LR(1)项中的向前看符号信息
  - 又使状态数减少到与SLR分析表的一样多

二义性文法的使用

- 二义性文法都不是LR的
- 某些二义性文法是有用的
  - 可以简洁地描述某些结构
  - 隔离某些语法结构，对其进行特殊处理

- 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
  - 可以在LR分析器中实现这个规则

优先级/结合性消除冲突

- 二义性文法
  - $E \rightarrow E+E \mid E*E \mid (E) \mid \text{id}$
- 等价于
  - $E \rightarrow E+T \mid T$　　$T \rightarrow T*F \mid F$　　$F \rightarrow (E) \mid \text{id}$
- 二义性文法的优点
  - 容易修改算符的优先级和结合性
  - 简洁：如果有多个优先级，那么无二义性文法将引入太多的非终结符号
  - 高效：不需要处理像$E \rightarrow T$这样的归约

二义性表达式文法的LR(0)项集

- 文法
  - $E \rightarrow E+E \mid E*E \mid (E) \mid \text{id}$
- 冲突
  - I_7、I_8中有冲突，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:
$E \rightarrow (E) \cdot$

冲突的原因以及解决

- 当栈顶状态为7时，表明
  - 栈中状态序列对应的文法符号序列为：…$E+E$
  - 如果下一个符号为+或*，移入还是归约？
- 如果*的优先级大于+，且+是左结合的
  - 下一个符号为*时，我们应该移入*
  - 下一个符号为+时，我们应该将$E+E$归约为E
- 栈顶状态为8时，有类似情况

解决冲突之后的SLR(1)分析表

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

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	

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

- 这个表和等价的无二义性文法的分析表类似

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

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			

悬空else的二义性

- 文法

  $S' \rightarrow S$

  $S \rightarrow \mathbf{i}\ S\ \mathbf{e}\ S \mid \mathbf{i}\ S \mid a$

- 项集I_4包含冲突
  - 栈中符号为：$\mathbf{i}\ S$

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:
$S \rightarrow iSeS\cdot$

- 冲突
  - 下一个符号为**e**，因栈中**i**未匹配，该移入**e**
  - 如果下一个符号属于FOLLOW(S)，该归约
  - 解决：移入

语法错误的处理

- 错误难以避免，编译器需要有处理错误的能力
- 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
- 语法分析器中错误处理程序的设计目标
  - 清晰准确地报告出现的错误，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
  - 尽可能地减少开销

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
  - 使用者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
  - 可能恢复得并不成功，之后找到的语法错误是假的
  - 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
  - ~~摆烂~~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前缀中找到和某个非终结符号对应的串，出现错误表明已经不可能找到对应这个非终结符号(程序结构)的串了
    - 如果编程错误仅局限于这个程序结构，我们可考虑跳过该程序结构，假装已经找到了它，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

- 文法符号*A*的同步集合的启发式规则

非终结符：

- 将FOLLOW(*A*)中所有符号放入*A*的同步集合中
- 将高层次非终结符号对应串的开始符号加入到较低层次非终结符号的同步集合
  - 比如：语句的开始符号，if/while等可作为表达式的同步集合
- 将FIRST(*A*)中的符号加入到*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，把该产生式当作默认值

终结符：

- 在栈顶的终结符号出现匹配错误时，可直接弹出该符号，并且发出消息称已经插入了这个终结符号

- 根据特定应用来决定哪些符号需要确定同步集合

恐慌模式的例子 (1)

- 对E, T, F的表达式进行语法分析时，可直接使用其FOLLOW值作为同步集合
  - synch表示一直忽略到同步集合，然后弹出非终结符号
  - 空条目表示忽略输入符号 (多)
  - 终结符号不匹配时，弹出栈中终结符号 (漏)

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

恐慌模式的例子 (2)

- 错误输入
  - + id * + id

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，并发出适当的错误消息

LR语法分析中的错误恢复 (1)

- 查询ACTION表时可能发现报错条目
  - 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$\alpha a x$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有一个对应于某个非终结符号A的GOTO目标 (s之上的状态被丢弃)
  - 在输入中丢弃一些符号，直到一个可以跟在A之后的符号b (不丢弃b)，并将GOTO(s, A)压栈，继续进行分析
  - 基本思想：假定当前试图归约到A但碰到了语法错误，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

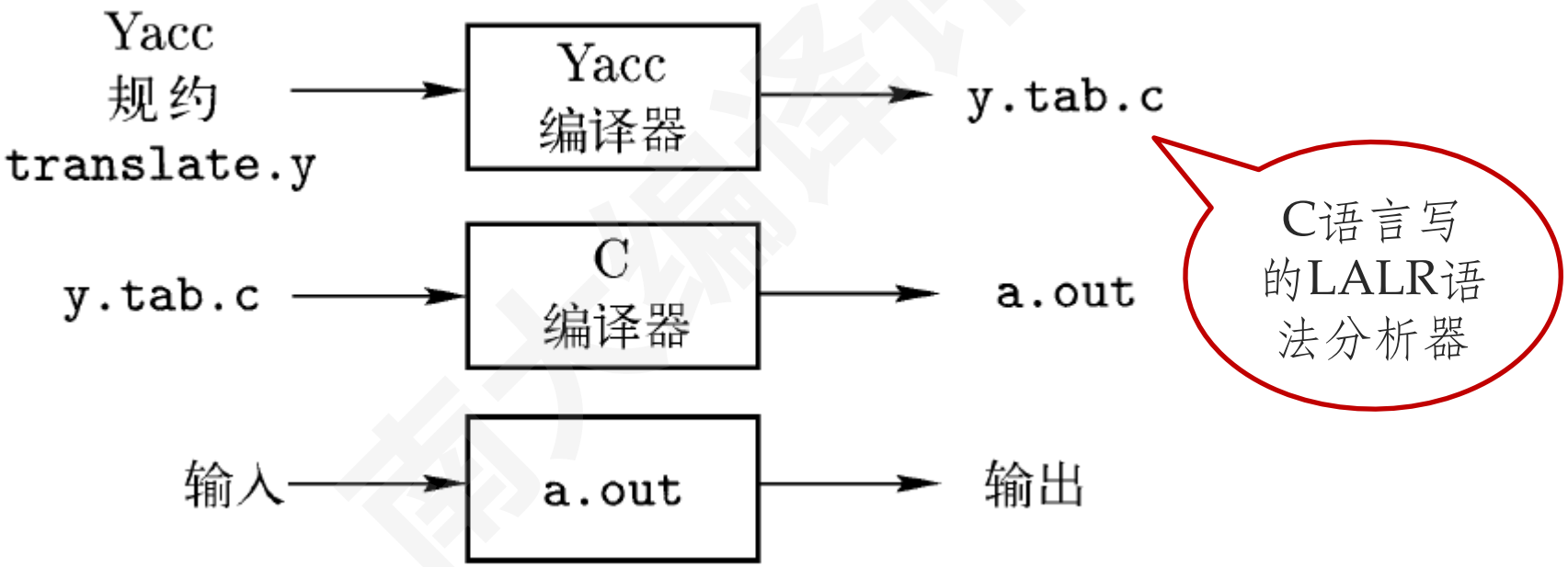
LR语法分析中的错误恢复 (2)

- 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

南大编译课件

语法分析器生成工具YACC/Bison

- YACC的使用方法如下



YACC源程序的结构

- **声明**
  - 放置C声明和对词法单元的声明
- **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
  - 可在语义动作中调用
  - 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语言例程
```

翻译规则的格式

- 说明
  - 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
  - $$表示和产生式头相关的属性值，$*i*表示产生式体中第*i*个文法符号的属性值
  - 在按照某个产生式归约时，执行相应的语义动作，可以根据$*i*来计算$$的值

<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*>}
　　;

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中的冲突处理

- 缺省处理方法
  - 归约/移入冲突：总是移入 (悬空else的解决)
  - 归约/归约冲突：选择列在前面的产生式
  - 选项-v可在文件y.output中看到冲突的描述和解决方法
- 通过确定终结符号的优先级/结合性来解决冲突
  - 结合性：%left, %right, %nonassoc
  - 移入a/按$A \rightarrow \alpha$归约：比较a和$A \rightarrow \alpha$的优先级再选择
    - 终结符号的优先级按在声明部分的出现顺序而定
    - 产生式的优先级设为它最右的终结符号的优先级，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

YACC的错误恢复

- 使用错误产生式来完成语法错误恢复
  - 错误产生式$A \rightarrow \text{error}\ \alpha$
  - 例如：*stmt* → error ;
- 定义哪些非终结符号有错误恢复动作
  - 比如：表达式、语句、块、函数定义等非终结符号
- 当语法分析器遇到错误时
  - 不断弹出栈中状态，直到栈顶状态包含项$A \rightarrow \cdot\ \text{error}\ \alpha$
  - 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；否则跳过一些符号，找到可以归约为α的串为止