# 第五讲　自底向上语法分析

2024-10

## 1. 基本思想

对于给定语言的文法和一个单词符号串（终结符串），一般的自底向上分析过程是：从所要分析的终结符串开始进行归约，每一步归约是在当前串中找到与某个产生式的右部相匹配的子串，然后将该子串用这一产生式的左部非终结符进行替换；如果找不到这样的子串，则回退到上一步归约前的状态，选择不同的子串或不同的产生式重试；重复这一过程，直到归约至文法开始符号；如果不存在任何一个这样的归约过程，则表明该单词符号串存在语法错误。

例如，给定如下文法 $G[S]$：

$$
\begin{aligned}
&S \to AB\\
&A \to aA \mid \varepsilon\\
&B \to b \mid bB
\end{aligned}
$$

针对单词符号串 $aaab$ 的一个自底向上分析过程为

$$
\begin{aligned}
&aaab && // \text{ 使用产生式 } A \to \varepsilon\\
\Leftarrow\ &aaaAb && // \text{ 使用产生式 } A \to aA\\
\Leftarrow\ &aaAb && // \text{ 使用产生式 } A \to aA\\
\Leftarrow\ &aAb && // \text{ 使用产生式 } B \to b\\
\Leftarrow\ &aAB && // \text{ 使用产生式 } A \to aA\\
\Leftarrow\ &AB && // \text{ 使用产生式 } S \to AB\\
\Leftarrow\ &S
\end{aligned}
$$

这里，我们用“$\Leftarrow$”表示一步归约，与“$\Rightarrow$”表示一步推导形成对照。

然而，这种一般的自底向上分析过程存在很大程度的非确定性：在每一步归约中，选择哪个产生式，以及匹配哪个位置上的子串都可能是非确定的。这种非确定性导致归约过程需要不断地进行试探和回溯，导致分析过程会有很高的复杂性。例如，对于文法 $G[S]$ 单词符号串 $aaab$，以下是一个试探的归约过程：

$$
\begin{aligned}
&aaab && // \text{ 使用产生式 } A \to \varepsilon\\
\Leftarrow\ &aaAab && // \text{ 使用产生式 } A \to aA\\
\Leftarrow\ &aAab && // \text{ 使用产生式 } A \to aA\\
\Leftarrow\ &Aab && // \text{ 使用产生式 } B \to b\\
\Leftarrow\ &AaB && // \text{ 使用产生式 } A \to \varepsilon\\
\Leftarrow\ &AaAB && // \text{ 使用产生式 } A \to aA\\
\Leftarrow\ &AAB && // \text{ 使用产生式 } A \to aA\\
\Leftarrow\ &AS && // \text{ 必须回溯}
\end{aligned}
$$

那么，这种回溯的根源是什么呢？其实，我们很快就会发现，第一步将第二个 *a* 和第三个 *a* 之间的 ε 归约为 *A* 是一个错误的选择，无论在此之后选择什么样的归约过程，都不可能最终归约至开始符号 *S*。也就是说，*aaAab* 实际上不是一个句型。对于文法的某个句型或句子，如果一个子串根据某个产生式归约后，得到文法的另一个句型，那么我们称这样的子串为**可归约串**；如果没有这样的产生式，那么它就是不可归约的串。上面的例子中，*aaab* 中第二个 *a* 和第三个 *a* 之间的 ε 为不可归约串，而第三个 *a* 和 *b* 之间的 ε 为可归约串。

在实用的自底向上分析中，总是选择某个“可归约串”进行归约，可大大减少回溯。

### 1.1 短语和直接短语

与可归约串紧密相关的概念是短语和直接短语，下面我们先给出短语的定义：

对于文法 $G = (V_N, V_T, P, S)$，若 $S \Rightarrow^* \alpha A \delta$ 且 $A \Rightarrow^+ \beta$，则称 $\beta$ 是句型 $\alpha\beta\delta$ 的一个**短语**。其中，$\alpha, \delta, \beta \in (V_N \cup V_T)^*$。

**例 1** 设有文法 *G*[*S*]：

$$S \to AB$$
$$A \to aA \mid \varepsilon$$
$$B \to b \mid bB$$

试分别指出句子 *aaab* 和句型 *aaAb* 的全部短语。

**解** 根据短语的定义，我们可以先给出以所考虑的句型为果实的分析树，在此基础上不难给出该句型的全部短语。对于一棵分析树，其果实为所有叶子结点的符号（可以是终结符，也可以是非终结符）从左到右进行连接所得到的串。关于果实（yield）的概念，大家可参考有关书籍[5]。值得注意的是，句型的分析树必须是以开始符号为根结点。如果文法 *G*[*S*] 为无二义的，则对于每一句型来说，这样的分析树是唯一的。

图 1（*a*）为以 *aaab* 为果实的分析树，图 1（*b*）为以 *aaAb* 为果实的分析树。

实际上，在这样的分析树中，每个非叶子结点（内部结点）与该句型的一个短语是一一对应的。从图 1（*a*），我们可以得出句子 *aaab* 的所有短语为：

| | | |
|---|---|---|
| ε ： | *aaa***ε***b* | // 对应果实为 ε 祖先结点为 *A* 的子树 |
| *a* ： | *aa***a**ε*b* | // 对应果实为 ***a*ε** 祖先结点为 *A* 的子树 |
| *aa* ： | *a***aa**ε*b* | // 对应果实为 ***aa*ε** 祖先结点为 *A* 的子树 |
| *aaa* ： | ***aaa***ε*b* | // 对应果实为 ***aaa*ε** 祖先结点为 *A* 的子树 |
| *aaab* ： | ***aaa*ε*b*** | // 对应果实为 ***aaa*ε*b*** 祖先结点为 *S* 的子树 |
| *b* ： | *aaa***b** | // 对应果实为 ***b*** 祖先结点为 *B* 的子树 |

其中，最左边的一列给出了 *aaab* 的所有短语，第二列的加重部分指出了短语在句型中的位置。类似地，根据图 1（*b*），我们可以得出句型 *aaAb* 的全部短语为：

| | | |
|---|---|---|
| *aA* ： | *aa***A***b* | // 对应果实为 ***aA*** 祖先结点为 *A* 的子树 |
| *aaA* ： | ***aaA***b* | // 对应果实为 ***aaA*** 祖先结点为 *A* 的子树 |
| *aaAb* ： | ***aaAb*** | // 对应果实为 ***aaAb*** 祖先结点为 *S* 的子树 |
| *b* ： | *aaA***b** | // 对应果实为 ***b*** 祖先结点为 *B* 的子树 |

同样，最左列给出了 *aaAb* 的所有短语，第二列加重部分指出了短语在句型中的位置。

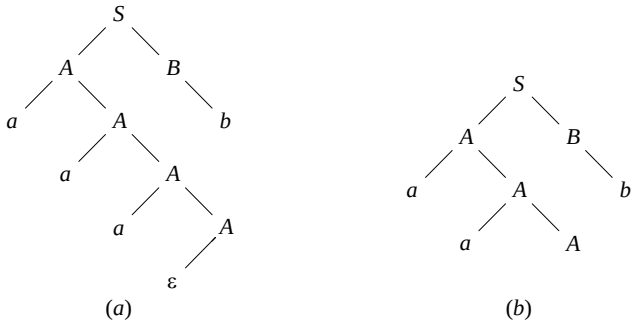



**图 1　句型的分析树**

根据短语的定义，$\beta$ 是句型 $\sigma\beta\delta$ 的一个短语，实际上相当于 $\beta$ 可以多步归约到某个非终结符 $A$，其中 $A$ 是从开始符号可达的，即 $S \Rightarrow^* \alpha A\delta$。对于定句型 $\alpha\beta\delta$ 来说，可以将短语 $\beta$ 进行多步归约至 $A$，每一步归约的子串都是当前句型的一个可归约串。

进一步，如果我们限定只能进行一步归约，则这样的短语称为直接短语，定义如下：

对于文法 $G = (V_N, V_T, P, S)$，若 $S \Rightarrow^* \alpha A\delta$ 且 $A \Rightarrow \beta$，则称 $\beta$ 是句型 $\alpha\beta\delta$ 的一个**直接短语**，其中，$\alpha, \delta, \beta \in (V_N \cup V_T)^*$。

**例 2** 对于例 1 中的文法 $G[S]$，试分别指出句子 *aaab* 和句型 *aaAb* 的全部直接短语。

**解**　例 1 已经得出句子 *aaab* 和句型 *aaAb* 的全部短语。在此基础上，我们只需找到那些只使用一步就可归约至相应非终结符的那些短语。

实际上，在句型的分析树中，只包含直接子孙的非叶子结点（内部结点）与该句型的直接短语是一一对应的。从图 1（*a*），我们可以得出句子 *aaab* 的所有直接短语为：

| | | |
|---|---|---|
| ε ： | *aaa**ε**b* | // 对应果实为 ε 父结点为 *A* 的子树 |
| *b* ： | *aaa**b*** | // 对应果实为 ***b*** 父结点为 *B* 的子树 |

类似地，根据图 1（*b*），我们可以得出句型 *aaAb* 的全部直接短语为：

| | | |
|---|---|---|
| *aA* ： | *a**aA**b* | // 对应果实为 ***aA*** 父结点为 *A* 的子树 |
| *b* ： | *aaA**b*** | // 对应果实为 ***b*** 父结点为 *B* 的子树 |

对于特定句型来说，它的每个直接短语就是该句型的一个可归约串。因此，在每一步只使用一个产生式的自底向上分析中，总是选择某个直接短语进行一步归约。

### 1.2　句柄

在某个句型的直接短语不唯一时，应该对哪一个进行归约呢？这个选择是非确定的。为解决这种非确定性，通常的做法是选择最左边的直接短语进行归约。这种做法的另一原因是，在从左到右读进输入符号的过程中尽可能早的进行归约，可以大大改善分析算法的时空效率。为了更好地描述这种归约过程，我们引入一个关键概念—句柄。

对于文法 $G = (V_N, V_T, P, S)$，若 $S \Rightarrow_{rm}^* \alpha Aw$ 且 $A \Rightarrow \beta$，则称 $\beta$ 是右句型 $\alpha\beta w$ 的一个相对于非终结符 $A$ 的**句柄**，其中，$\alpha, \beta \in (V_N \cup V_T)^*$，而 $w \in V_T^*$。

注意：句柄一定是相对于特定的非终结符而言的，但有时为方便简洁，当上下文中的这个非终结符是明确的或者不影响到所讨论的问题时，我们经常会将忽略不提。

句柄的概念只适合于右句型，它是右句型的一个直接短语。我们知道：一个终结符串是合法的，即属于 $L(G)$，当且仅当它可以由开始符号通过一系列的最右推导得到。因此，我们可以在自底向上分析的每一步中仅使用句柄进行归约。如果我们把这种按照句柄进行归约的过程逆过来，实际上就相当于一个最右推导过程。本章要重点讨论的 *LR* 系列分析方法就是建立在最右推导的基础上。

**例 3** 对于例 1 中的文法 $G[S]$，试分别指出句子 *aaab* 和右句型 *aaAb* 的句柄。

**解** 任何一个句子一定是右句型，句子 *aaab* 的一个最右推导是：

$$S \Rightarrow_{rm} AB \Rightarrow_{rm} Ab \Rightarrow_{rm} aAb \Rightarrow_{rm} aaAb \Rightarrow_{rm} aaaAb \Rightarrow_{rm} aaab$$

*aaAb* 也是一个右句型，它的一个最右推导是：

$$S \Rightarrow_{rm} AB \Rightarrow_{rm} Ab \Rightarrow_{rm} aAb \Rightarrow_{rm} aaAb$$

该文法是无二义的，因而每个右句型的最右推导是唯一的。从上述最右推导可以看出，句子 *aaab* 唯一的句柄是 ε （句柄在句型中的位置表示为 *aaaεb* 的加深部分），而右句型 *aaAb* 唯一的句柄是 *aA*。

我们也可以从分析树中找到句柄：先找出所有的直接短语，最左边的直接短语就是句柄。从例 2，我们得知句子 *aaab* 的所有直接短语为：

ε ： *aaaεb* // 对应果实为 ε 父结点为 *A* 的子树
*b* ： *aaa**b*** // 对应果实为 ***b*** 父结点为 *B* 的子树

由图 1（*a*）可看出，最左边的直接短语是 ε，它是 *aaab* 唯一的句柄。

同理，由图 1（*b*）可看出，最左边的直接短语是 *aA*，所以右句型 *aaAb* 的句柄是 *aA*。

一般情况下，如果所考虑的文法是无二义的，那么每个右句型有唯一的最右推导，也对应有唯一的以根结点为开始符号的分析树，因而其句柄是唯一的。同时，句柄是位于右句型中最左边的直接短语。这是因为，如果 $\beta$ 是右句型 $\alpha\beta w$ 的相对于非终结符 $A$ 的句柄，同时存在 $\beta'$ 满足 $\alpha = \alpha'\beta'$ 且 $\beta'$ 是相对于非终结符 $A'$ 的直接短语，即 $\beta'$ 是从左边紧邻 $\beta$ 的直接短语，那么右句型 $\alpha\beta w$ 的最右推导过程形如 $S \Rightarrow_{rm}^* \alpha'A'Aw \Rightarrow_{rm} \alpha'A'\beta w \Rightarrow_{rm} \alpha'\beta'\beta w = \alpha\beta w$，这样 $\alpha\beta w$ 的句柄应该是 $\beta'$ 而不是 $\beta$。

然而，若是对于二义文法，那么右句型可能存在多个分析树，也就可能有多个句柄。参见下面的例子。

**例 4** 设有文法 $G[S]$：

$$S \to AB$$
$$A \to aA \mid aaA \mid \varepsilon$$
$$B \to b \mid bB$$

试分别指出句子 *aaab* 和句型 *aaAb* 的全部句柄。

**解** 图 2（*a*）和（*b*）分别给出句子 *aaab* 和句型 *aaAb* 的根结点为 *S* 的所有分析树。

图 2（*a*）有三个分析树，无论从哪个分析树我们均可观察到，句子 *aaab* 最左边的直接短语均为第 3 个 *a* 和 *b* 之间的 ε，即句子 *aaab* 有唯一的句柄：

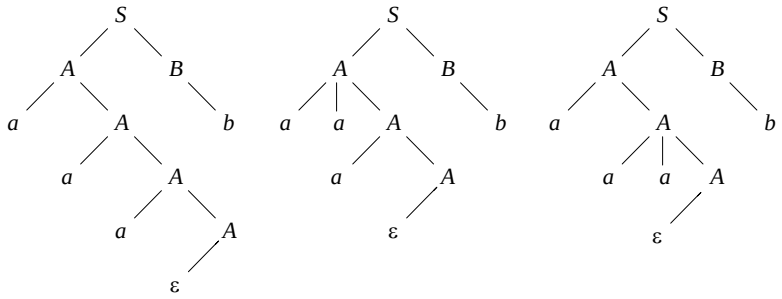
ε ： *aaa**ε**b* // 对应果实为 ε 祖先结点为 *A* 的子树

从图 2（*b*）的两个分析树，我们可以得到句型 *aaAb* 的所有直接短语为：

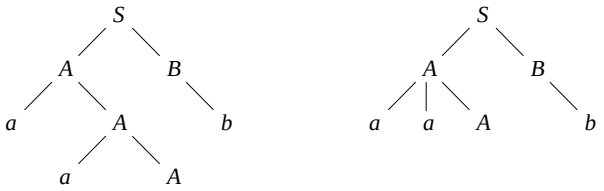
*aA* ： *a**aA**b* // 对应左边分析树中果实为 ***aA*** 祖先结点为 *A* 的子树
*aaA* ： ***aaA**b* // 对应右图中果实为 ***aaA*** 祖先结点为 *A* 的子树
*b* ： *aaA**b* // 对应两个图中果实为 ***b*** 祖先结点为 *B* 的子树

我们可以从图 2（*b*）的每个分析树中观察到最左的直接短语，找到句型 *aaAb* 有如下两个句柄：

*aA* ： *a**aA**b* // 对应左图中果实为 ***aA*** 祖先结点为 *A* 的子树
*aaA* ： ***aaA**b* // 对应右图中果实为 ***aaA*** 祖先结点为 *A* 的子树




(*a*)




(*b*)

**图 2　二义文法中句型的分析树**

# 2. 移进–归约分析

实现自底向上分析最常用的技术是**移进-归约分析**。它的基本思想是借助一个栈（称为分析栈或下推栈）和一个基于有限状态控制的分析引擎，分析引擎根据当前状态、分析栈当前内容、余留的输入符号串来唯一确定如下动作之一，然后进入新状态：

- *Reduce*：依确定的方式对位于栈顶的短语进行归约。
- *Shift*：从输入序列移进一个单词符号。
- *Error*：发现语法错误，进行错误处理/恢复。
- *Accept*：分析成功。

不同的自底向上分析方法在进行 *Reduce* 动作时可将短语、直接短语、或句柄替换为某个非终结符。若是替换一般的短语，则可能会使用多个产生式进行多步归约；若是替换直接短语或句柄，则只使用一个产生式进行一步归约。在本书重点讨论的 *LR* 分析方法中，是对

句柄进行归约，*Reduce* 动作是将位于当前栈顶部分的句柄替换为该句柄相应的非终结符。

前面也反复提到，实用的语法分析过程应该消去各种非确定因素。移进-归约分析过程确定化的关键是解决好两类冲突：移进–归约冲突和归约–归约冲突。

**移进–归约冲突**是指到达一个不能确定下一步应该移进还是应该归约的状态。例如，某文法中有如下产生式：

$S \rightarrow$ if $E$ then $S$ | if $E$ then $S$ else $S$

考虑正在对于如下句型进行移进–归约分析：

if $E$ then if $E$ then $S$ else $S$

当 if $E$ then if $E$ then $S$ 出现在栈中时，是移进 else，还是归约 if E then S？

**归约–归约冲突**是指到达一个可以对多个短语进行归约的状态。例如，某文法中有如下产生式：

$A \rightarrow aA \mid aaA \mid \varepsilon$

考虑对于包含 $aaa$ 的串进行的移进–归约分析过程。当分析到某一步时，可能有 $aaA$ 出现在分析栈中，是用产生式 $A \rightarrow aA$ 归约 $aA$，还是用产生式 $A \rightarrow aaA$ 归约 $aaA$?

在移进–归约分析中，为了解决这些冲突，经常会采用向前查看确定数目的单词符号，规定特定的优先关系等技术。然而，无论采用什么技术，不可能找到通用的解决方法适合于任何文法。例如，对于如下文法 $G[S]$，不存在确定的移进–归约分析方法（即确保在移进–归约分析过程中不出现任何冲突）：

$S \rightarrow a\,S\,a \mid b\,S\,b \mid \varepsilon$

我们不关心这个结果的证明，有兴趣的读者可参考有关书籍或文献。

在下面一节里，我们将介绍基于移进–归约分析技术的 LR 系列分析方法，其中的 LR（1）分析方法可以达到所有确定的分析方法中最强的分析能力。

为简化状态控制，在实现移进–归约分析时通常会借助于一张分析表，称之为表驱动的方法。图 3 描述了一种表驱动移进–归约分析模型。图中，*Input#* 表示余留的输入符号串，# 为串结束符；*Output* 表示可能的分析结果，如分析树。分析引擎根据当前状态，分析栈当前内容、余留的输入符号串，通过查询分析表，来确定下一步动作。下一步动作若是 *Shift*，则从余留的输入符号串中移入一个单词符号至分析栈；若是 *Reduce*，则按照分析表中记录的归约方案进行归约，归约过程通常会访问记录了所有产生式的表。动作完成后，分析引擎进入下一个状态，重复以上工作。

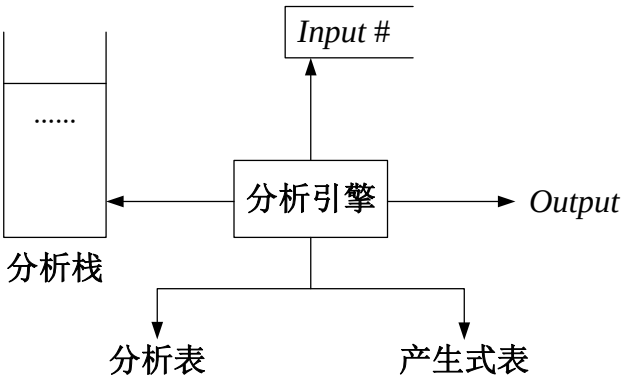



**图 3　表驱动移进–归约分析模型**

我们先通过例子初步了解移进–归约分析的具体过程。设有如下文法 *G*[*E*]：

(1) $E \rightarrow E + T$

(2) $E \rightarrow T$

(3) $T \rightarrow T * F$

(4) $T \rightarrow F$

(5) $F \rightarrow ( E )$

(6) $F \rightarrow v$

(7) $F \rightarrow d$

假设待分析的输入符号串为 *v*＋*v***d*，图 4 和 图 5 分别描述了采用两种不同分析方法的移进–归约分析过程。开始时，假设“分析栈”中只有一个开始符号 #，将“余留符号串”置为 *v*＋*v***d*#。分析引擎根据通过查询分析表得到下一步要进行的“动作”。动作为 *Shift*，表示移进；动作为 *Reduce*，表示归约。

执行移进动作 *Shift* 时，当前的输入单词符号从余留符号串移出，移至分析栈的栈顶位置，然后进入到下一状态。

在图 4 的移进–归约分析过程中，*Reduce* 动作是按句柄进行归约。表的“动作”一栏里标出了归约时使用的产生式，如 *Reduce*(3) 表示使用产生式 $T \rightarrow T * F$ 进行归约。从步骤（11）到步骤（12）执行 *Reduce*(3) 后，分析栈顶部的句柄 $T * F$ 被归约至 *T*。动作结束后进入到下一状态。

| 步骤 | 分析栈 | 余留符号串 | 动作 |
|---|---|---|---|
| (0) | # | *v + v * d #* | *Shift* |
| (1) | *# v* | *+ v * d #* | *Reduce*(6) |
| (2) | *# F* | *+ v * d #* | *Reduce*(4) |
| (3) | *# T* | *+ v * d #* | *Reduce*(2) |
| (4) | *# E* | *+ v * d #* | *Shift* |
| (5) | *# E +* | *v * d #* | *Shift* |
| (6) | *# E + v* | ** d #* | *Reduce*(6) |
| (7) | *# E + F* | ** d #* | *Reduce*(4) |
| (8) | *# E + T* | ** d #* | *Shift* |
| (9) | *# E + T ** | *d #* | *Shift* |
| (10) | *# E + T * d* | *#* | *Reduce*(7) |
| (11) | *# E + T * F* | *#* | *Reduce*(3) |
| (12) | *# E + T* | *#* | *Reduce*(1) |
| (13) | *# E* | *#* | *Accept* |

**图 4　按句柄归约的移进-归约分析过程**

在图 5 的移进–归约分析过程中，分析引擎是根据运算优先关系来控制移进或归约的。例如，从步骤（5）到步骤（6），由于预先假设了运算 * 优先于运算 +，所以执行 *Shift* 将移进。从步骤（8）到步骤（9），*Reduce* 动作是将短语 $F*F$ 归约至 $T$；显然，这是一个多步的归约，涉及到产生式 $T \to F$ 和 $T \to T*F$ 进行归约。动作结束后进入到下一状态。

| 步骤 | 分析栈 | 余留符号串 | 动作 |
|---|---|---|---|
| (0) | # | *v + v * d #* | *Shift* |
| (1) | *# v* | *+ v * d #* | *Reduce* |
| (2) | *# F* | *+ v * d #* | *Shift* |
| (3) | *# F +* | *v * d #* | *Shift* |
| (4) | *# F + v* | ** d #* | *Reduce* |
| (5) | *# F + F* | ** d #* | *Shift* |
| (6) | *# F + F ** | *d #* | *Shift* |
| (7) | *# F + F * d* | *#* | *Reduce* |
| (8) | *# F + F * F* | *#* | *Reduce* |
| (9) | *# F + T* | *#* | *Reduce* |
| (10) | *# E* | *#* | *Accept* |

**图 5　采用另一种归约方式的移进-归约分析过程**

图 4 和图 5 的移进–归约分析中分别采用的是 LR 分析方法和算符优先分析方法。前者是我们要重点介绍的内容，见下一节。后者特别适合于表达式的分析，但限于其适用范围的局限性以及篇幅所限，本课程将不覆盖相关的内容，有兴趣的同学可查阅相关书籍。

# 3. LR 分析方法

## 3.1 LR 分析基础

LR 分析方法是应用最普遍的自底向上分析方法。

LR 中的“L”代表从左（Left）向右扫描单词符号，“R”代表产生的是最右（Rightmost）推导。LR 分析是一种确定的自底向上分析方法。回顾1.2节，最右推导的逆过程对应于自底向上的每一步对句柄进行归约。

图4的移进-归约分析采用的就是LR分析方法。每一步中，如果处于正常状态（非出错状态），则后续动作必定取决于当前所期待的句柄可能有哪些以及这些句柄的哪些部分已经出现在栈顶，因此，我们将当前全部所期待的句柄已出现在栈顶的部分和未出现的部分抽象为一个状态。初始状态记为“0”，后续状态分别标以“1”，“2”，“3”，...。

状态“0”可刻画为下列序偶的集合：<ε, E>, <ε, E+T>, <ε, T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d>。

初始状态比较特殊，当前分析栈上未出现任何文法符号，而我们期待着分析结束时栈上出现唯一的文法开始符号，即E。这里，我们将E看作是一种所期待的特殊“句柄”。其实，很快会看到，后续将引入“增广文法”的概念，原文法的开始符号自然可在新文法中作为句柄。因此，状态“0”中包含<ε, E>，即期待的句柄E已出现在栈顶的部分为ε。若要出现E，则必然先要出现E+T或T，因此E+T和T也是当前状态下所期待的句柄，故状态“0”中也包含<ε, E+T>和<ε, T>。以此类推，状态“0”中还包含<ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> 和<ε, d>。

在某个状态，若相关句柄中的下一个符号入栈，则会转移至另一状态。

对于**状态“0”**：{ <ε, E>, <ε, E+T>, <ε, T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d> }

若“E”入栈，则转移至 { <E, ε>, <E, +T> }，记为状态“1”；

若“T”入栈，则转移至 { <T, ε>, <T, *F> }，记为状态“2”；

若“F”入栈，则转移至 { <F, ε> }，记为状态“3”；

若“(”入栈，则转移至 { <(, E)>, <ε, E+T>, <ε, T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d> }，记为状态“4”；

若“v”入栈，则转移至 { <v, ε> }，记为状态“5”；

若“d”入栈，则转移至 { <d, ε> }，记为状态“6”。

对于**状态“1”**：{ <E, ε>, <E, +T> }

若“+”入栈，则转移至 { <E+, T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d> }，记为状态“7”。

类似地，我们有：

对于**状态“2”**：{ <T, ε>, <T, *F> }

若“*”入栈，则转移至 { <T*, F>, <ε, (E)> , <ε, v> , <ε, d> }，记为状态“8”。

对于**状态“4”**：{ <(, E)>, <ε, T>,  <ε, E+T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d> }

若“E”入栈，则转移至 { <(E, )> , <E, +T> }，记为状态“9”；

若“T”入栈，则转移至 { <T, ε>, <T,*F> }，即为状态“2”；

若“F”入栈，则转移至 { <F, ε> }，即为状态“3”；

若“(”入栈，则转移至 { <(, E)>, <ε, T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d> }，即为状态“4”；

若“v”入栈，则转移至 { <v, ε> }，即为状态“5”；

若“d”入栈，则转移至 { <d, ε> }，即为状态“6”。

对于**状态“7”**：{ <E+, T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d> }

若“T”入栈，则转移至 { < E+T, ε>, <T,* F> }，记为状态“10”；

若“F”入栈，则转移至 { <F, ε> }，即为状态“3”；

若“(”入栈，则转移至 { <(, E)>, <ε, T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d> }，即为状态“4”；

若“v”入栈，则转移至 { <v, ε> }，即为状态“5”；

若“d”入栈，则转移至 { <d, ε> }，即为状态“6”。

对于**状态“8”**：{ <T*, F>, <ε, (E)> , <ε, v> , <ε, d> }

若“F”入栈，则转移至 { < T*F, ε> }，记为状态“11”；

若“(”入栈，则转移至 { <(, E)>, <ε, T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d> }，即为状态“4”；

若“v”入栈，则转移至 { <v, ε> }，即为状态“5”；

若“d”入栈，则转移至 { <d, ε> }，即为状态“6”。

对于**状态“9”**：{ <(E, )> , <E, +T> }

若“)”入栈，则转移至 { < (E), ε> }，记为状态“12”；

若“+”入栈，则转移至 { <E+, T>, <ε, T*F> , <ε, F>, <ε, (E)> , <ε, v> , <ε, d> }，即为状态“7”。

对于**状态“10”**：{ < E+T, ε>, <T, *F> }

若“*”入栈，则转移至 { <T*, F>, <ε, (E)> , <ε, v> , <ε, d> }，即为状态“8”。

上述转移关系，可刻画为如图6所示的状态转移图。

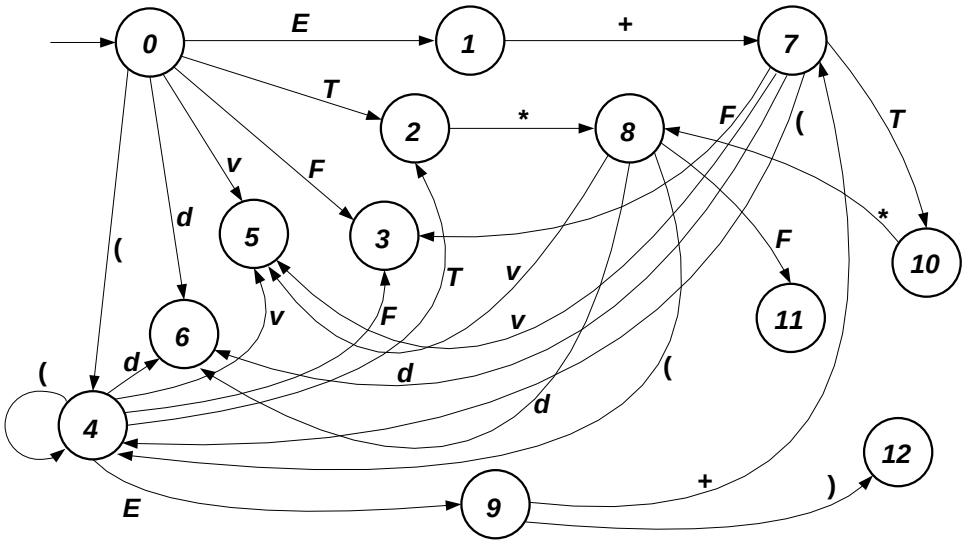



**图 6　刻画栈顶句柄出现情况的状态转移图**

为简洁，对于图6的状态，我们没有考虑当前所期待的句柄是针对哪个产生式的（或针对哪个非终结符的），因为文法G[E]中所有产生式右部的符号串均不同。此类文法称为 **UI** (uniquely invertible)**文法**或者**反向确定**(backwards deterministic)**文法**，所以不会影响到当前所讨论的问题，但在正式算法中是要考虑句柄是针对哪个产生式的，参见后面的LR（0）有限状态机构造（见3.2.3节）。

与图 3 对应，图 7 描述了一种表驱动的 LR 分析模型，分析过程中使用的分析表称为 LR 分析表。值得注意的是：在设计 LR 分析程序时，通常的做法是在分析栈中存放分析引擎的当前状态（如图 6 的状态转移图中的状态），这样的分析栈我们称之为状态分析栈，简称**状态栈**。相应地，我们把如图 4 所示的分析栈称为符号分析栈，简称**符号栈**。

有关 LR 分析表的构造过程，我们将在随后的小节里介绍。这一小节主要介绍 LR 分析的过程，因此我们假定已经存在构造好的 LR 分析表。对于各种不同的 LR 分析方法，LR 分析表的结构都是通用的，LR 分析的过程也是通用的。

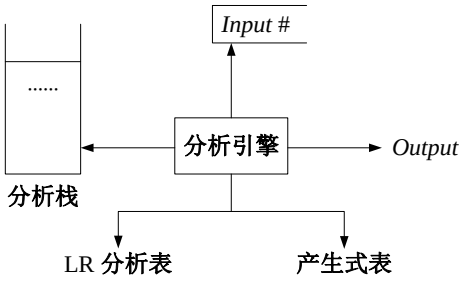



**图 7　表驱动 LR 分析模型**

LR分析表由两部分构成：

- **ACTION 表**　告诉分析引擎：在栈顶状态为 *k*, 当前的输入单词符号是 *a* 时应该完成的动作。ACTION 表的表项有如下 4 种取值：

  ACTION [*k*,a] = s*i*，　*Shift*：状态 *i* 移进栈顶
  ACTION [*k*,a] = r*j*，　*Reduce*：按第 *j* 条产生式归约

ACTION [*k*,a] = *acc*， *Accept*：分析完成

ACTION [*k*,a] = *err*，*Error*：发现错误 （常标为空白）

- **GOTO 表** GOTO[*i*,*A*] = *j* 告诉分析引擎：在依产生式 $A\rightarrow\beta$ 归约之后，位于栈顶的状态如何改变。依产生式 $A\rightarrow\beta$ 归约时，要将栈顶的 $|\beta|$ 个状态弹出，假设此时位于栈顶的状态是 *i*，那么就将新状态为 *j* 移进栈顶。

我们先看一个LR分析表的例子。图8是前面的文法 *G*[*E*] 的一个LR分析表。

参考图8，LR分析表的每一行对应分析引擎的一个状态；ACTION 表的每一列对应一个输入单词符号或#，后者表示输入结束符号；GOTO 表的每一列对应一个文法非终结符。分析引擎的初始状态通常表示为 0，其余状态依次表示为 1, 2, 3, …。若不是特别指明，今后 ACTION 表中表项取值为 *err* 时用都空白表示。实际上，这些状态与图6的转移图中的状态完全对应。

| 状态 | ACTION | | | | | | | GOTO | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *v* | *d* | * | + | ( | ) | # | *E* | *T* | *F* |
| 0 | s5 | s6 | | | s4 | | | 1 | 2 | 3 |
| 1 | | | | s7 | | | acc | | | |
| 2 | | | s8 | r2 | | r2 | r2 | | | |
| 3 | | | r4 | r4 | | r4 | r4 | | | |
| 4 | s5 | s6 | | | s4 | | | 9 | 2 | 3 |
| 5 | | | r6 | r6 | | r6 | r6 | | | |
| 6 | | | r7 | r7 | | r7 | r7 | | | |
| 7 | s5 | s6 | | | s4 | | | | 10 | 3 |
| 8 | s5 | s6 | | | s4 | | | | | 11 |
| 9 | | | | s7 | | s12 | | | | |
| 10 | | | s8 | r1 | | r1 | r1 | | | |
| 11 | | | r3 | r3 | | r3 | r3 | | | |
| 12 | | | r5 | r5 | | r5 | r5 | | | |

**图 8 LR 分析表举例**

假设待分析的输入符号串为 *v*＋*v***d*。图 9 描述了基于图 8 中分析表的 LR 分析过程。开始时，状态分析栈中包含分析引擎的初始状态 0，余留符号串置为 *v*＋*v***d*#。图中“分析动作”一列给出了通过查表得到的下一步要进行的“动作”。

在步骤（0），下一步动作为 ACTION [0, *v*] = s5，执行该动作的结果是：将当前输入单词符号，即余留符号串的第一个符号移出，将状态 5 移入状态分析栈成为新的栈顶状态。在步骤（1），下一步动作为 ACTION [5, +] = r6，即按照第 6 条产生式 $F\rightarrow v$ 进行归约，执行该动作的结果是：将状态 5 从栈中移出，然后将状态 GOTO [0, *F*] = 3 入栈并成为栈顶状态。

我们通过看一个更具代表性例子来解释归约动作的执行过程。考虑从步骤（11）到步骤（12），执行归约动作 ACTION [11, #] = r3，即按照第 3 条产生式 $T\rightarrow T*F$ 进行归约。由于该产生式右部的符号串长度为 3，因此要从栈顶移出 3 个状态符号，依次为 11，8，和 10，结果新的栈顶状态变为 7。通过查 GOTO 表，得到 GOTO [7, *T*] = 10，因此将状态 10 压入状态分析栈。

图 9 中其它步骤类似。到达步骤（13）时，ACTION [1, #] = acc，表示分析成功，分析过

程结束。

若在某一步骤中，遇到 ACTION 表中的空白表项，则表明输入符号串存在语法错误，转出错处理过程。

| 步骤 | 状态分析栈 | 余留符号串 | 分析动作 |
|---|---|---|---|
| (0) | 0 | *v* + *v* ∗ *d* # | ACTION [0, *v*] = s5 |
| (1) | 0 5 | + *v* ∗ *d* # | ACTION [5, +] = r6, GOTO [0, *F*] = 3 |
| (2) | 0 3 | + *v* ∗ *d* # | ACTION [3, +] = r4, GOTO [0, *T*] = 2 |
| (3) | 0 2 | + *v* ∗ *d* # | ACTION [2, +] = r2, GOTO [0, *E*] = 1 |
| (4) | 0 1 | + *v* ∗ *d* # | ACTION [1, +] = s7 |
| (5) | 0 1 7 | *v* ∗ *d* # | ACTION [7, *v*] = s5 |
| (6) | 0 1 7 5 | ∗ *d* # | ACTION [5, ∗] = r6, GOTO [7, *F*] = 3 |
| (7) | 0 1 7 3 | ∗ *d* # | ACTION [3, ∗] = r4, GOTO [7, *T*] = 10 |
| (8) | 0 1 7 10 | ∗ *d* # | ACTION [10, ∗] = s8 |
| (9) | 0 1 7 10 8 | *d* # | ACTION [8, *d*] = s6 |
| (10) | 0 1 7 10 8 6 | # | ACTION [6, #] = r7, GOTO [8, *F*] = 11 |
| (11) | 0 1 7 10 8 11 | # | ACTION [11, #] = r3, GOTO [7, *T*] = 10 |
| (12) | 0 1 7 10 | # | ACTION [10, #] = r1, GOTO [0, *E*] = 1 |
| (13) | 0 1 | # | ACTION [1, #] = acc |

**图9　LR分析过程举例**

为方便参考，图 10 描述了 LR 分析的一般过程。

---

**置 *ip* 指向输入串 *w* 的首符号，置初始栈顶状态为** 0

**令 *i* 为栈顶状态，*a* 是 *ip* 指向的符号，重复如下步骤：**

```
if ( ACTION[i, a] = sj ) {
   PUSH j；  /*进栈*/      ip 前进；/*指向下一输入符号*/
}
else if （ACTION[i, a] = rj ) {   /*设第 j 条产生式为 A → β*/
                  POP |β| 项; /*位于栈顶部的 |β| 个状态退栈*/
                  令当前栈顶状态为 k；  PUSH GOTO[k, A];
          }
          else if （ACTION[i, a] = acc）  return；   /*成功*/
                    else error；     /*报错/错误恢复*/
```

---

**图 10　LR 分析算法**

为了方便讨论，我们有时会让分析栈中既包含状态又包含符号，可以认为既有状态栈又有符号栈，参见图11中给出的分析模型。同样，可以将图10 描述的 LR 分析过程扩展为图12 描述的同时含有状态栈和符号栈的 LR 分析过程。

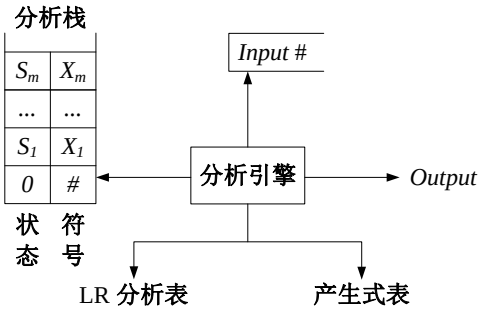



**图 11　同时含有状态栈和符号栈的 LR 分析模型**

---

**置 *ip* 指向输入串 *w* 的首符号，置状态栈顶为 0，状态栈顶为 #**

**令 *i* 为栈顶状态，*a* 是 *ip* 指向的符号，重复如下步骤：**

```
if ( ACTION[i, a] = sj ) {
   PUSH j, a ;  /*进栈*/     ip 前进 ; /*指向下一输入符号*/
}
else if （ACTION[i, a] = rj ) {      /*设第 j 条产生式为 A → β */
        POP |β| 项;       /*位于两个栈顶部的 |β| 个状态 或符号退栈*/
        设当前状态栈顶为 k;
        PUSH GOTO[k, A]，A ;
     }
     else if （ACTION[i, a] = acc ) return;  /*成功*/
          else error;     /*报错/错误恢复*/
```

---

**图 12　同时含有状态栈和符号栈的 LR 分析算法**

| 步骤 | 状态分析栈 | 余留符号串 | 分析动作 |
|---|---|---|---|
| (0) | 0# | *v* + *v* ∗ *d* # | ACTION [0, *v*] = s5 |
| (1) | 0# 5*v* | + *v* ∗ *d* # | ACTION [5, +] = r6, GOTO [0, *F*] = 3 |
| (2) | 0# 3*F* | + *v* ∗ *d* # | ACTION [3, +] = r4, GOTO [0, *T*] = 2 |
| (3) | 0# 2*T* | + *v* ∗ *d* # | ACTION [2, +] = r2, GOTO [0, *E*] = 1 |
| (4) | 0# 1*E* | + *v* ∗ *d* # | ACTION [1, +] = s7 |
| (5) | 0# 1*E* 7+ | *v* ∗ *d* # | ACTION [7, *v*] = s5 |
| (6) | 0# 1*E* 7+ 5*v* | ∗ *d* # | |
| (7) | 0# 1*E* 7+ 3*F* | ∗ *d* # | ACTION [3, ∗] = r4, GOTO [7, *T*] = 10 |
| (8) | 0# 1*E* 7+ 10*T* | ∗ *d* # | ACTION [10, ∗] = s8 |
| (9) | 0# 1*E* 7+ 10*T* 8∗ | *d* # | ACTION [8, *d*] = s6 |
| (10) | 0# 1*E* 7+ 10*T* 8∗ 6*d* | # | ACTION [6, #] = r7, GOTO [8, *F*] = 11 |
| (11) | 0# 1*E* 7+ 10*T* 8∗ 11*F* | # | ACTION [11, #] = r3, GOTO [7, *T*] = 10 |
| (12) | 0# 1*E* 7+ 10*T* | # | ACTION [10, #] = r1, GOTO [0, *E*] = 1 |
| (13) | 0# 1*E* | # | ACTION [1, #] = acc |

**图 13　LR 分析过程举例（同时含有状态栈和符号栈）**

**例 5** 根据图 12 描述的算法，扩展图 9 的分析过程，使分析栈中同时包含状态和符号。

**解**　本例的目的主要是为了读者能够将状态栈和符号栈对应起来，也可以认为分析栈的元素

是状态和符号构成的二元组。我们直接给出如图 13 的结果，无须更多的解释。

### 3.2 LR（0） 分析

我们将在随后的各小节里依次讨论几种不同的 LR 分析方法，包括 LR（0）、SLR（1）、LR（1） 和 LALR（1） 等分析方法。它们有着不同的分析能力，所适合的文法分别称为 LR（0） 文法、SLR（1） 文法、LR（1） 文法和 LALR（1） 文法。这些方法采用了结构上是一致的分析表，即 LR 分析表，只是构造方法上存在差异。它们在 LR 分析方法的发展中扮演了重要角色，但我们在这里不去追究这些分析方法的发展历史。

这些分析方法的名称中，“0” 代表向前查看 0 个符号，“1” 代表向前查看 1 个符号，SLR（1）代表 Simple LR（1），LALR（1） 代表 Look Ahead LR（1）。

LR（0） 分析是这几种方法中最弱的一个，它对文法的要求非常严格，但它是所有方法中最基础的，对其它方法的设计有直接的影响。

下面我们来介绍 LR（0） 分析方法的原理。首先，需要引入三个核心概念：增广文法，活前缀，和 LR（0）有限状态机。然后，在此基础上介绍 LR（0） 分析表的构造。

#### 3.2.1 增广文法

对于文法 $G = (V_N, V_T, P, S)$，增加如下产生式

$$S' \to S$$

其中，$S' \notin V_N \cup V_T$，得到 $G$ 的**增广文法**

$$G' = (V_N, V_T, P, S')$$

注：增广文法等价于原文法。增广文法的开始符号不会出现在任何产生式的右部，这一点在构造 LR（0）有限状态机时有特殊作用。若原文法的开始符号已经满足这个条件，那么可以不作这样的改造。

#### 3.2.2 活前缀

对于文法 $G = (V_N, V_T, P, S)$，若 $S \Rightarrow_{rm}^* \alpha Aw$ 且 $A \Rightarrow \beta$，其中 $\alpha, \beta \in (V_N \cup V_T)^*$，$w \in V_T^*$，即 $\beta$ 是右句型 $\alpha\beta w$ 的一个相对于非终结符 $A$ 的句柄，则 $\alpha\beta$ 的任何前缀 $\gamma$ 都是文法 $G$ 的**活前缀**。

例如，对于如下文法 $G[S]$：

$$S \to AB$$
$$A \to aA \mid \varepsilon$$
$$B \to b \mid bB$$

由例 3 可知，句子 $aaab$ 唯一的句柄是 $\varepsilon$，右句型 $aaAb$ 唯一的句柄是 $aA$，所以 $aaa$ 和 $aaA$ 的任何前缀都是文法 $G[S]$的活前缀：$\varepsilon$，$a$，$aa$，$aaa$，$aaA$。

可以这样来理解活前缀：在 LR 分析中，文法 $G = (V_N, V_T, P, S)$ 中的符号串 $\gamma \in (V_N \cup V_T)^*$ 是活前缀，当且仅当存在 $w \in L(G)$，$\gamma$ 恰好是在针对 $w$ 的分析过程中的某个时刻分析栈上的符号串（初始栈顶符号 # 除外）。

通过活前缀和句柄二者之间的关系，可以加深对两个重要概念的认识。一个活前缀是某

一右句型的前缀，它不超过该右句型的某个句柄，二者之间存在如下关系：

- 活前缀已含有该句柄的全部符号：表明该句柄对应的产生式 $A \to \alpha$ 的右部 $\alpha$ 已出现在栈顶。
- 活前缀只含该句柄的一部分符号：表明该句柄对应的产生式 $A \to \alpha_1\alpha_2$ 的右部子串 $\alpha_1$ 已出现在栈顶，期待从输入串中看到可由 $\alpha_2$ 推导出的符号串。
- 活前缀不含有该句柄的任何符号：此时期待从输入串中看到该句柄对应的产生式 $A \to \alpha$ 的右部所推导出的符号串。

### 3.2.3 LR（0）有限状态机

LR（0） 有限状态机是一种特定的有限自动机，用于 LR（0） 分析过程的有限状态控制。我们先来看LR（0） 有限状态机的构造过程。

每个上下文无关文法 $G = (V_N, V_T, P, S)$ 都对应有一个 LR（0） 有限状态机，可以由 $G$ 的增广文法 $G' = (V_N, V_T, P, S')$ 直接构造得到，它是一个以 $V_N \cup V_T$ 为字母表的 $DFA$。

LR（0） 有限状态机状态的定义基于 LR（0） 项目的概念。一个**LR（0） 项目**是在右端某一位置有圆点的产生式。如，产生式 $A \to xyz$ 对应如下 4 个 LR（0） 项目：

$A \to .xyz$
$A \to x.yz$
$A \to xy.z$
$A \to xyz.$

若是 ε 产生式 $A \to \varepsilon$，则唯一的 LR(0) 项目表示为 $A \to .$。

圆点标志着已分析过的串与该产生式匹配的位置。根据圆点所在的位置和圆点后是终结符还是非终结符或为空，把项目分为以下几种：

移进项目: 形如 $A \to \alpha . a\beta$, 其中 $a \in V_T$, $\alpha, \beta \in (V_N \cup V_T)^*$。

待约项目: 形如 $A \to \alpha . B\beta$

归约项目: 形如 $A \to \alpha .$

接受项目: 形如 $S' \to S .$

LR（**0**） **有限状态机**的任一**状态**是某个 **LR（0） 项目集** I 的**闭包** CLOSURE(I)。可以通过如下步骤计算 LR（0） 项目集 I 的闭包 CLOSURE(I)：

（1）置 J = I；

（2）Repeat For J 中的每个项目 $A \to \alpha . B\beta$ 和文法的每个产生式 $B \to \gamma$

Do 若 $B \to .\gamma$ 不在 J 中，则加 $B \to .\gamma$ 到 J 中；

Until 上一次循环不再有新项目加到 J 中；

（3）置 CLOSURE(I) = J 。

特别地，设文法 $G[S]$ 的增广文法为 $G'[S']$，则该文法之 LR（0） 有限状态机的**初态**为

$I_0$ = CLOSURE({S′ →. S})

**例 6** 设如下文法 *G*[*E*] 的增广文法为 *G′*[*E′*]，试给出该文法之 LR（0）有限状态机的初态：

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to ( E )$
(4) $T \to d$

**解** 根据上述计算项目集闭包的步骤，可得 LR（0） 有限状态机的初态

$I_0$ = CLOSURE({$E' \to . E$})
= { $E' \to .E$，
$E \to .E+T$，
$E \to . T$，
$T \to . ( E )$,
$T \to . d$
}

下面，我们定义 LR（0） 有限状态机的**转移函数**为：

GO (I, X) = CLOSURE(J)

其中，I 为 LR（0） 有限状态机的状态，X 为文法符号，J = { $A \to \alpha X . \beta \mid A \to \alpha . X\beta \in I$ }。

从 LR（0） 有限状态机的初态出发，应用上述转移函数，可逐步构造出完整的 LR（0）有限状态机，其所有状态的集合 C 可由如下步骤计算：

（1）置 C:= { CLOSURE ({ $S' \to .S$ })}，其中 *S* 为文法开始符号，*S′* 为增广文法开始符号；

（2）Repeat for C 中每一项目集 I 和每一文法符号 X

Do 若 GO(I, X) 非空且不属于 C，则加 GO(I, X) 到 C 中；

Until C 不再增大。

应该注意到，给定一个 LR(0) 有限状态机的状态，以及每一个文法符号，根据这一转移函数所到达的下一个状态（如果存在）是确定的。

**例 7** 针对例 6 中的文法 *G*[*E*] 及其增广文法 *G′*[*E′*]，试构造该文法之 LR（0）有限状态机。

**解** 从例 6 中得到的 LR（0） 有限状态机的初态开始，根据上述步骤可逐步构造出完整的 LR（0） 有限状态机，如图 14 所示。

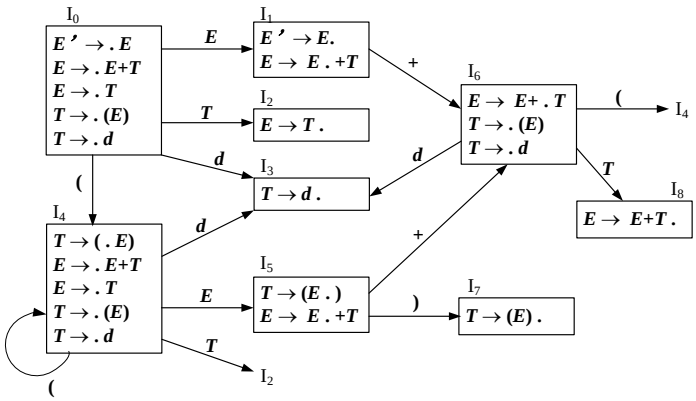



**图 14　一个 LR（0） 有限状态机**

对于文法 $G=(V_N, V_T, P, S)$，其增广文法为 $G'=(V_N, V_T, P, S')$，由上述方法构造的 LR（0）有限状态机可以看作一个以 $V_N \cup V_T$ 为字母表的 *DFA*。如果我们假定，除了（没有绘出的）死状态是非终态外，其余所有状态都是终态，那么可以有如下结论：该 *DFA* 的语言是增广文法为 *G′* 的所有活前缀的集合。由此可知，对于任何句型来说，按照这个 *DFA* 构造的分析引擎不会错过任何可归约的句柄，或者说不会错过任何最右推导；另一方面，该 *DFA* 的所有可接受的符号串都是 *G′* 的活前缀，因此只要不进到死状态，那么这些符号串就会是某个右句型的活前缀。对于这一结论，有兴趣的同学可以参考附录A。

进一步分析会发现，对于 LR（0）有限状态机任何状态 I，项目$A \to \beta_1 . \beta_2 \in \mathrm{I}$，当且仅当存在最右推导 $S' \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha\beta_1\beta_2 w$，且从初态$\mathrm{I}_0$ 经符号串$\alpha\beta_1$可达状态I。这一结论是LR分析算法的核心结论之一，但超出本课程范围，这里不进行证明。

注：如果存在最右推导 $S \Rightarrow_{rm}^* \alpha A w \Rightarrow_{rm} \alpha\beta_1\beta_2 w$，我们称**项目$A \to \beta_1 . \beta_2$ 针对活前缀$\alpha\beta_1$是有效的**（*valid*）。

所以，一个LR（0）有限状态机的状态I可以看作是针对其所对应的活前缀有效的所有项目的集合。由所有这些项目集合构成的集合也称为文法*G*的LR（0）**项目集规范族**（the *canonical collection* of sets of LR(0) items for *G*）。

那么，分析引擎的构造是如何使用这个 *DFA* 的呢？读者可以结合 3.1 中的 LR 分析算法和下面关于 LR（0） 分析表的构造方法找到答案。

### 3.2.4　LR（0） 分析表

LR（0） 分析表可以直接从 LR（0） 有限状态机得到，参见图 15 所描述的构造过程。

假定文法开始符号为 $S$，其增广文法的开始符号为 $S'$。 $C = \{I_0, I_1, \cdots, I_n\}$ 是该文法 LR(0) 有限状态机的全部状态集合，在其中，我们令状态 $I_k$ 对应于 LR(0) 分析表中的状态为 $k$；令含有项目 $S' \to .S$ 的状态为 $I_0$，因此 LR(0) 分析表中的状态 0 对应初态。ACTION 表项和 GOTO 表项可按如下方法构造：

- 若项目 $A \to \alpha . a\beta$ 属于 $I_k$ 且 $GO(I_k, a) = I_j$，$a$ 为终结符，则置 ACTION[$k$, $a$] 为“把状态 $j$ 和符号 $a$ 移进栈”，简记为“s$j$”；
- 若项目 $A \to \alpha .$ 属于 $I_k$，那么，对任何终结符 $a$，置 ACTION[$k$, $a$] 为“用产生式 $A \to \alpha$ 进行归约”，简记为“r$j$”；其中，假定 $A \to \alpha$ 为文法的第 $j$ 个产生式；
- 若项目 $S' \to S .$ 属于 $I_k$，则置 ACTION[$k$, #] 为“接受”，简记为“acc”；
- 若 $GO(I_k, A) = I_j$，，$A$ 为非终结符，则置 GOTO[$k$, $A$] = $j$；
- 分析表中凡不能用上述规则填入信息的空白格均置上“出错标志”。

**图 15　从 LR（0） 有限状态机构造 LR（0） 分析表**

**例 8** 利用例 7 得到的 LR（0） 有限状态机，构造例 6 中文法的 LR（0） 分析表。

**解** 根据图 15 所描述的构造过程，所得到的 LR（0） 分析表如图 16 所示。

| 状态 | ACTION | | | | | GOTO | |
|---|---|---|---|---|---|---|---|
| | *d* | + | ( | ) | # | *E* | *T* |
| 0 | s3 | | s4 | | | 1 | 2 |
| 1 | | s6 | | | acc | | |
| 2 | r2 | r2 | r2 | r2 | r2 | | |
| 3 | r4 | r4 | r4 | r4 | r4 | | |
| 4 | s3 | | s4 | | | 5 | 2 |
| 5 | | s6 | | s7 | | | |
| 6 | s3 | | s4 | | | | 8 |
| 7 | r3 | r3 | r3 | r3 | r3 | | |
| 8 | r1 | r1 | r1 | r1 | r1 | | |

**图 16　从 LR（0） 有限状态机构造 LR（0） 分析表举例**

值得注意的是，LR（0） 分析表中，代表“按某个产生式进行归约”的表项一定是整行出现的，意味着在 LR（0） 分析过程中遇到这一行对应的状态时，不向前查看任何输入符号就能确定需要进行归约。

可能细心的读者已经注意到，图 15 中的算法实际上存在一个关键的问题：一个文法的 LR（0） 分析表中的一些表项有可能会被**多重定义**，即被不止一个 r*j* 或 s*j* 赋值。这种情况下，是不合适进行如图 10 和图 12 所描述的 LR 分析过程的。

按照图 15 中的算法构造分析表，如果各表项均无多重定义，则称该文法为一个 **LR（0） 文法**。

可以通过 LR（0） 有限状态机直接判断相应的文法是否 LR（0）的。文法 *G* 是 LR（0） 文法，当且仅当它 的 LR（0） 有限状态机中的每个状态都满足：

- 不同时含有移进项目和归约项目，即不存在移进-归约冲突。
- 不含有两个以上归约项目，即不存在归约-归约冲突。

读者可以验证：图 14 中的状态都满足这两个条件。需要注意的是，状态 $I_1$ 中的 $E' \to E$ . 是接受项目，不是归约项目，因此 $I_1$ 中不存在移进-归约冲突。

### 3.3 SLR（1） 分析

LR（0）分析方法只适合于 LR（0）文法，但满足 LR（0）要求的文法不多。

**例 9** 我们为 3.1 节中的表达式文法 $G[E]$ 增加产生式 $S \to E$，得到如下增广文法 $G'[S]$：

(0) $S \to E$
(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to ( E )$
(6) $F \to v$
(7) $F \to d$

试给出该文法的 LR（0） 有限状态机，并在此基础上分析 $G[E]$ 是否 LR（0）文法。

**解** 根据 3.2.3 节介绍的方法，我们得到该文法的 LR（0） 有限状态机，如图17所示。 不难发现，这个LR（0） 有限状态机与图6中的状态转移图完全对应。

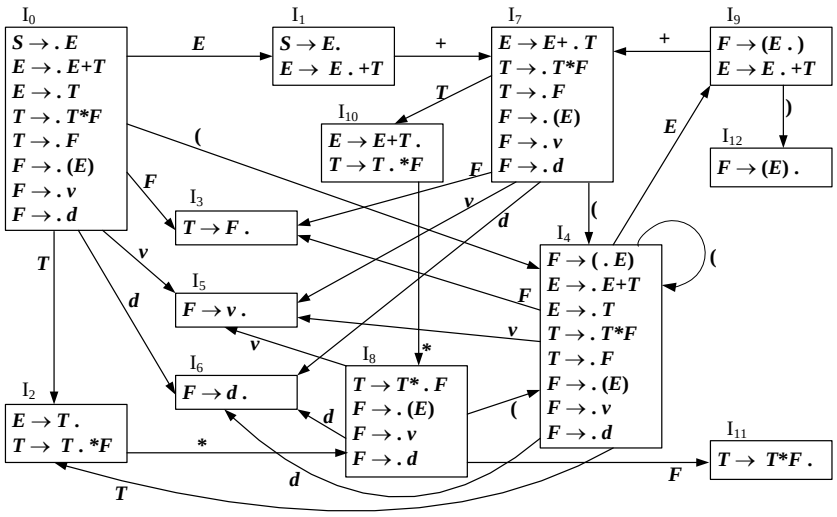



**图 17 一个有冲突状态的 LR（0） 有限状态机**

可以验证，图 17 中的 LR（0） 有限状态机中，状态 $I_2$ 和 $I_{10}$ 存在移进-归约冲突，因此文法 $G[E]$ 不是 LR（0）文法。

与例 6 的 LR（0）文法相比，例 9 中的文法只是使产生的表达式可以含乘法运算以及变量标识符 $v$。

在文法 $G[E]$中，我们有 Follow($E$) = { +, ), # }。$I_2$ 和 $I_{10}$ 中的移进项目指示要移进的输入符号为 *，不属于 Follow($E$)；而 $I_2$ 和 $I_{10}$ 中的归约项目都是指示归约后的非终结符为 $E$，因此所期望的下一个输入符号为 Follow($E$)中的符号。这样，通过向前查看一个符号可以解决 $I_2$ 和 $I_{10}$ 存在的冲突。

通常，通过向前查看一个输入符号，可以根据这一符号是否属于要归约的非终结符的 Follow 集来决定是否进行归约。如果通过这种做法可以使 LR（0） 有限状态机中所有状态中的冲突问题得到解决，那么我们就可以进行正常的 LR 分析。我们称这种方法为 **SLR（1）分析方法**，它的基本原理是检查 LR（0） 有限状态机的所有状态是否有冲突，并借助于以下两个方面来解决冲突问题：

- 如果每一状态的所有归约项中要归约的非终结符的 Follow 集互不相交，则可以解决归约–归约冲突。
- 如果每一状态的所有归约项中要归约的非终结符的 Follow 集与该状态所有移进项目要移进的符号集互不相交，则可以解决移进–归约冲突。

只需对 LR（0） 分析表进行简单修改，使得归约表项只适用于相应非终结符 Follow 集中的输入符号，就可得到 SLR（1） 分析表。从 LR（0） 有限状态机构造 SLR（1） 分析表的过程参见图 18。

---

**假定文法开始符号为** $S$，**其增广文法的开始符号为** $S'$。 $C = \{I_0, I_1, \cdots, I_n\}$ **是该文法** LR(0) **有限状态机的全部状态集合，在其中，我们令状态** $I_k$ **对应于** LR(0) **分析表中的状态为** $k$；**令含有项目** $S' \to .S$ **的状态为** $I_0$，**因此** LR(0) **分析表中的状态** 0 **对应初态。**ACTION **表项和** GOTO **表项可按如下方法构造：**

- **若项目** $A \to \alpha . a\beta$ **属于** $I_k$ **且** $GO(I_k, a) = I_j$，$a$ **为终结符，则置** ACTION[$k$, $a$] **为“把状态** $j$ **和符号** $a$ **移进栈”，简记为“**s$j$**”；**
- **若项目** $A \to \alpha .$ **属于** $I_k$，**那么，对任何** $a \in Follow(A)$，**置** ACTION[$k$, $a$] **为“用产生式** $A \to \alpha$ **进行归约”，简记为“**r$j$**”；其中，假定** $A \to \alpha$ **为文法的第** $j$ **个产生式；**
- **若项目** $S' \to S .$ **属于** $I_k$，**则置** ACTION[$k$, #] **为“接受”，简记为“**acc**”；**
- **若** $GO(I_k, A) = I_j$，$A$ **为非终结符，则置** GOTO[$k$, $A$] = $j$；
- **分析表中凡不能用上述规则填入信息的空白格均置上“出错标志”。**

---

**图 18　从 LR（0） 有限状态机构造 SLR（1） 分析表**

**例 10** 对于例 9 中的文法以及所得到的 LR（0） 有限状态机，构造相应的 SLR（1） 分析表。

**解** 根据图 18 所描述的构造过程，得到的 SLR（1） 分析表如图 8 所示。

同样，图 18 中的算法也存在这样的问题：一个文法的 SLR（1） 分析表中的一些表项有可能会被多重定义，即被不止一个 r*j* 或 s*j* 赋值。这种情况不合适进行 LR 分析的。

按照图 18 中的算法构造分析表，如果各表项均无多重定义，则称该文法为一个 **SLR（1）文法**。

可以通过 LR（0） 有限状态机直接判断相应的文法是否 SLR（1） 的。文法 $G$ 是 SLR（1） 文法，当且仅当它 的 LR（0） 有限状态机中的每个状态都满足：

- 对该状态的任何项目 $A \to \alpha . a\beta$ （$a$ 为终结符），不存在项目 $B \to \gamma .$，使得 $a \in$ Follow($B$) ，即不存在移进-归约冲突。
- 对该状态的任何两个项目 $A \to \alpha .$ 和 $B \to \beta .$，满足 Follow($A$) $\cap$ Follow($B$) = $\Phi$，即不存在归约-归约冲突。

读者可以验证：图 17 中 LR(0)有限状态机的状态都满足这两个条件。

SLR（1） 分析和 LR（0）分析的主要差别就是在归约一个句柄时，前者要向前查看一个输入符号，看是否是要归约的非终结符的 Follow 符号，若是则完成归约，否则进行出错处理。而对于 LR（0）分析来说，则无论下一个输入符号是什么，都进行归约。从出错处理的角度来看，可能对于某些错误报告来说，LR（0）分析会比 SLR（1） 分析晚一些，大家可以思考一下其中的原因。

体现在 LR（0） 表和 SLR（1） 表上，二者有如下区别：

- 在 LR（0） 表的 ACTION 表中，归约表项总是整行出现的，即每一个归约动作都是对应所有输入符；同时，在任何一行，不会既有移进表项又有归约表项。
- 而在 SLR（1） 表的 ACTION 表中，归约表项只适用于相应非终结符 Follow 集中的输入符号；在同一行，可以既有移进表项又有归约表项。

### 3.4 LR（1） 分析

SLR（1） 分析方法也有一定的局限性：只考虑到所归约非终结符的 Follow 符号。但一个输入符号属于所归约非终结符的 Follow 集合，未必就是句柄可以后跟的符号。

**例 11** 验证如下文法 *G*[*E*] 不是 SLR（1） 文法：

(1) $E \to (L \parallel E)$
(2) $E \to F$
(3) $L \to L \parallel E$
(4) $L \to E$
(5) $F \to (F)$
(6) $F \to d$

**解** 为文法 *G*[*E*] 增加产生式 $S \to E$，得到增广文法*G'*[*S*]。根据 3.2.3 节介绍的方法，我们得到该文法的 LR（0）有限状态机，如图19所示。

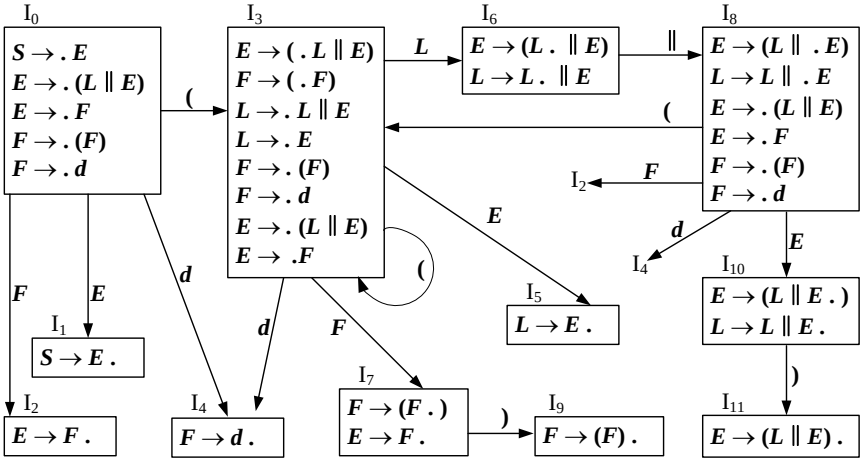



**图 19 一个非 SLR（1） 文法的 LR（0） 有限状态机**

可以验证，图 19 中的 LR（0） 有限状态机中，状态 $I_7$ 和 $I_{10}$ 存在移进-归约冲突，因此文法 *G*[*E*] 不是 LR（0） 文法。

计算 *G*[*E*] 中非终结符的 Follow 集合，我们得到：Follow(*E*) = {) , ‖ , #}，Follow(*L*) = { ‖ }。

可以看出，) ∉ Follow(*L*)，所以状态 $I_{10}$ 的移进-归约冲突可以通过 SLR（1） 方法解决。但，因为有 ) ∈ Follow(*E*)，所以状态 $I_7$ 的移进-归约冲突不可用这种方法解决。因此，我们得知，文法 *G*[*E*]不是 SLR（1） 文法，SLR（1） 分析方法不再适用。

我们来分析一下，当到达状态 $I_7$ 时，分析栈顶部是‘(*F*’（栈顶为‘*F*’）。如果将句柄 *F* 归约为 *E*，那么根据分析过程，下一状态会进入状态 $I_5$；接着将句柄 *E* 归约为 *L*，进入状态 $I_6$；此时就会看到，期望的下一个输入符号将是 ‖ 。这样，我们就得到一个解决状态 $I_7$ 中移进-归约冲突的方案：遇 ) 时，移进；遇 ‖ 时，归约；遇其它符号时，出错处理。

由此可知，当到达状态 $I_7$ 时，句柄 *F* 所期望的下一个输入符号实际上是 ‖ ，而不是 )。这两个符号都属于 Follow(*E*)，所以 SLR（1） 分析方法失去了作用。在这一小节里，我们介绍 LR（1） 分析方法，可以弥补这一不足。

下面我们来介绍 LR（1） 分析方法的原理。首先，需要引入一个核心概念 LR（1）有限状态机，不同于前面的 LR（0）有限状态机。然后，在此基础上介绍 LR（1） 分析表的构造。

### 3.4.1 LR（1） 有限状态机

前面已经知道，每个上下文无关文法 $G=(V_N, V_T, P, S)$ 都对应有一个 LR（0） 有限状态机。同样，它也对应有一个 LR（1） 有限状态机，也可以由 *G* 的增广文法 $G'=(V_N, V_T, P, S')$ 直接构造得到，它是一个以 $V_N \cup V_T$ 为字母表的 *DFA*。

类似地，LR（1） 有限状态机状态的定义基于 LR（1） 项目的概念。LR（1） 项目是在 LR（0）项目基础上增加一个终结符，称为**向前搜索符**，表示产生式右端完整匹配后所允许在余留符号串中的下一个终结符。文法 *G* 的一个 **LR（1） 项目**形如：

$$A \to \alpha . \beta \ , \quad a$$

其中，$A \to \alpha . \beta$ 是一个 LR（0） 项目，$a \in V_T \cup \{\#\}$ 为向前搜索符，’ ，’为分隔符。这里，#为输入结束标志符。

形如

$$A \to \alpha . \ , \quad a$$

的 LR（1） 项目，相当于 LR（0） 的归约项目，但只有当下一个向前搜索符是 *a* 时才可以进行归约。

对于其它形式的LR（1）项目，向前搜索符 *a* 只起到信息传承的作用，参见随后的 LR（1） 有限状态机构造过程。

为简化表示，若形如

$$A \to \alpha . \beta \ , \quad a_1$$
$$A \to \alpha . \beta \ , \quad a_2$$
$$\dots$$
$$A \to \alpha . \beta \ , \quad a_m$$

的 LR（1）项目序列需要出现在同一个项目集中时，将其简记为

$$A \to \alpha . \beta \ , \quad a_1 / a_2 / \dots / a_m$$

其中，'/' 为分隔符。

**LR（1） 有限状态机**的任一**状态**是某个 **LR（1） 项目集** I 的**闭包** CLOSURE(I)。可以通过如下步骤计算 LR（1） 项目集 I 的闭包 CLOSURE(I)：

（1）置 J = I；

（2）Repeat For J 中的每个项目 $[A \to \alpha . B\beta \ ， a]$ 和文法的每个产生式 $B \to \gamma$

Do 将所有形如 $[B \to . \gamma \ , b]$ 的项目加到 J 中，这里 $b \in \text{First}(\beta a)$；

Until 上一次循环不再有新项目加到 J 中；

（3）置 CLOSURE(I) = J 。

特别地，设文法 $G[S]$ 的增广文法为 $G'[S']$，则该文法之 LR（1） 有限状态机的**初态**为

$I_0$ = CLOSURE({[ $S' \to . S$ ， # ]})

**例 12** 设例 11 中文法 $G[E]$ 的增广文法为 $G'[S]$，试给出该文法之 LR（1）有限状态机的初态。

**解** 根据上述计算项目集闭包的步骤，可得 LR（1） 有限状态机的初态

$I_0$ = CLOSURE({[$S \to . E$ ， #]})
= { [ $S \to . E$ ， # ]，
[ $E \to . (L \parallel E)$ ， # ]，
[ $E \to . F$ ， # ]，
[ $F \to . (F)$ ， # ]，
[ $F \to . d$ ， # ]
}

下面，我们定义 LR（1） 有限状态机的**转移函数**为：

GO (I, X) = CLOSURE(J)

其中，I 为 LR（1）有限状态机的状态，X 为文法符号，J={[ $A \to \alpha X . \beta$，$a$] | [$A \to \alpha . X\beta$，$a$]∈ I}。

从 LR（1） 有限状态机的初态出发，应用上述转移函数，可逐步构造出完整的 LR（1）有限状态机，其所有状态的集合 C 可由如下步骤计算：

（1）置 C:= { CLOSURE ({[ $S' \to . S$ ， # ]})}，其中 $S$ 为文法开始符号，$S'$ 为增广文法开始符号；

（2）Repeat for C 中每一项目集 I 和每一文法符号 X

Do 若 GO(I, X) 非空且不属于 C，则加 GO(I, X) 到 C 中；

Until C 不再增大。

根据这一步骤构造的 LR（1） 有限状态机是一个 *DFA*，并省略了死状态。

**例 13** 针对例 11 中的文法 $G[E]$ 及其增广文法 $G'[S]$，试构造该文法之 LR（1）有限状态机。

**解** 从例 11 中 LR（1） 有限状态机的初态开始，根据上述步骤可逐步构造出完整的 LR（1）有限状态机，如图 20 所示。值得注意的是，在手工计算时，对于向前搜索符号一定要细心计算状态闭包，它是一个反复计算的过程，直到到达不动点为止，即不再有新项目产生为主。我们仅代表性地解释一下状态 $I_3$ 中的项目。从状态 $I_0$ 到状态 $I_3$，首先得到的项目为 [ $F\rightarrow$ ( . $F$) ， # ] 和 [ $E\rightarrow$ ( . $L$ ‖ $E$ ) ， # ]。从项目 [ $F\rightarrow$ ( . $F$) ， # ]，可知需要增加项目 [ $F\rightarrow$ . ( $F$) ， ) ] 和 [ $F\rightarrow$ . $d$ ， ) ]，其中的向前搜索符号 ) 属于 )# 的 First 集合。从项目 [ $E\rightarrow$ ( . $L$ ‖ $E$ ) ， # ]，可知需要增加项目 [ $L\rightarrow$ . ( ‖ $E$ ) ， ‖ ] 和项目 [ $L\rightarrow$ . $E$， ‖ ]，其中的向前搜索符号 ‖ 属于 First（‖ $E$ ) #）中的符号。类似地，我们又可从项目 [ $L\rightarrow$ . $E$， ‖ ] 得到新项目 [ $E\rightarrow$ . ($L$ ‖ $E$ ) ， ‖ ] 和 [ $E\rightarrow$ . $F$ ， ‖ ]。最后，从项目 [ $E\rightarrow$ . $F$ ， ‖ ]，我们新增项目 [ $F\rightarrow$ . ($F$) ， ‖ ] 和 [ $F\rightarrow$ . $d$ ， ‖ ]。采用缩写记法，我们将 [$F\rightarrow$ . ($F$) ， ) ] 和 [ $F\rightarrow$ . ($F$) ， ‖ ] 简写为 [$F\rightarrow$ . ($F$)， ‖ / ) ]，将 [$F\rightarrow$ . $d$ ， ) ] 和 [$F\rightarrow$ . $d$ ， ‖ ] 简写为 [ $F\rightarrow$ . $d$， ‖ / ) ] 。这样，我们就得到如图 20 所示状态 $I_3$。

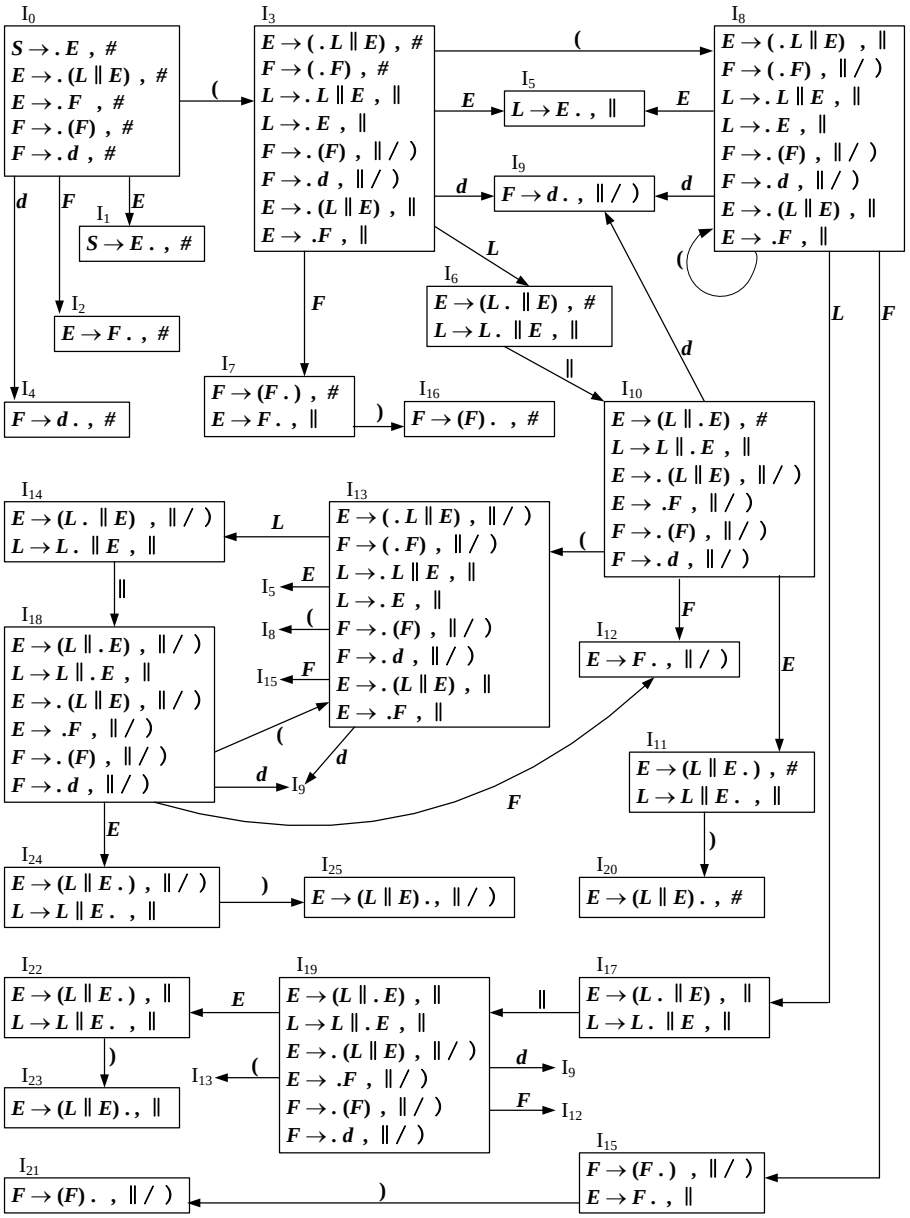



**图 20　一个 LR（1） 有限状态机**

### 3.4.2 LR（1） 分析表

LR（1） 分析表可以直接从 LR（1） 有限状态机得到，参见图 21 所描述的构造过程。

---

**假定文法开始符号为 $S$，其增广文法的开始符号为 S′。 C = {$I_0$, $I_1$,…， $I_n$} 是该文法 LR(0) 有限状态机的全部状态集合，在其中，我们令状态 $I_k$ 对应于 LR(0) 分析表中的状态为 $k$；令含有项目 $S' \to .S$ 的状态为 $I_0$，因此 LR(0) 分析表中的状态 0 对应初态。**ACTION **表项和** GOTO **表项可按如下方法构造：**

- **若项目** $[A \to \alpha . a\beta, b]$ **属于** $I_k$ **且** GO($I_k$, $a$) = $I_j$, $a$ **为终结符，则置** ACTION[$k$, $a$] **为“把状态** $j$ **和符号** $a$ **移进栈”，简记为“**s$j$**”；**
- **若项目** $[A \to \alpha ., b]$ **属于** $I_k$，**那么，置** ACTION[$k$, $b$] **为“用产生式** $A \to \alpha$ **进行归约”，简记为“**r$j$**”; 其中，假定** $A \to \alpha$ **为文法的第** $j$ **个产生式；**
- **若项目** $S' \to S .$ **属于** $I_k$，**则置** ACTION[$k$, #] **为“接受”，简记为“**acc**”；**
- **若** GO($I_k$, $A$) = $I_j$，$A$ **为非终结符，则置** GOTO[$k$, $A$] = $j$ **；**
- **分析表中凡不能用上述规则填入信息的空白格均置上“出错标志”。**

---

**图 21 从 LR（1） 有限状态机构造 LR（1） 分析表**

**例 14** 利用例 13 得到的 LR（1） 有限状态机，构造例 11 中文法的 LR（1） 分析表。

**解** 根据图 21 所描述的构造过程，所得到的 LR（1） 分析表如图 22 所示。

| 状态 | ACTION | | | | | GOTO | | |
|---|---|---|---|---|---|---|---|---|
| | *d* | ‖ | ( | ) | # | *E* | *L* | *F* |
| 0 | s4 | | s3 | | | 1 | | 2 |
| 1 | | | | | acc | | | |
| 2 | | | | | r2 | | | |
| 3 | s9 | | s8 | | | 5 | 6 | 7 |
| 4 | | | | | r6 | | | |
| 5 | | r4 | | | | | | |
| 6 | | s10 | | | | | | |
| 7 | | r2 | | s16 | | | | |
| 8 | s9 | | s8 | | | 5 | 16 | 14 |
| 9 | | r6 | | r6 | | | | |
| 10 | s9 | | s13 | | | 11 | | 12 |
| 11 | | r3 | | s20 | | | | |
| 12 | | r2 | | r2 | | | | |
| 13 | s9 | | s8 | | | 5 | 14 | 15 |
| 14 | | s18 | | | | | | |
| 15 | | r2 | | s21 | | | | |
| 16 | | | | | r5 | | | |
| 17 | | s19 | | | | | | |
| 18 | s9 | | s13 | | | 24 | | 12 |
| 19 | s9 | | s13 | | | 22 | | 21 |
| 20 | | | | | r1 | | | |
| 21 | | r5 | | r5 | | | | |
| 22 | | r3 | | s23 | | | | |
| 23 | | r1 | | | | | | |
| 24 | | r3 | | s25 | | | | |
| 25 | | r1 | | r1 | | | | |

**图 22 从 LR（1） 有限状态机构造 LR（1） 分析表举例**

同样，图 21 中的算法也存在多重定义的问题：一个文法的 LR（1） 分析表中的一些表项有可能会被不止一个 r*j* 或 s*j* 赋值。这种情况不合适进行 LR 分析的。

按照图 21 中的算法构造分析表，如果各表项均无多重定义，则称该文法为一个 **LR（1）文法**。

可以通过 LR（1） 有限状态机直接判断相应的文法是否 LR（1） 的。文法 *G* 是 LR（1） 文法，当且仅当它 的 LR（1） 有限状态机中的每个状态都满足：

- 如果该状态含有项目 $[A \to \alpha . a\beta, b]$ （$a$ 为终结符），那么就不会有项目$[B \to \gamma ., a]$ ；反之亦然。这表明该状态不存在移进-归约冲突。
- 该状态中的所有归约项目的向前搜索符互不相交，即不同时含有项目 $[A \to \alpha ., a]$ 和 $[B \to \beta ., a]$。这表明该状态不存在归约-归约冲突。

读者可以验证：图 20 中的状态都满足这两个条件。

我们来回顾一下例 11 的 LR（0） 有限状态机（图 19）中的冲突状态 $I_7$，该冲突不能用 SLR（1） 方法解决。不难看出，在图 20 的 LR（1） 有限状态机中，对应于同样活前缀的状态有两个：$I_7$ 和 $I_{15}$。在这两个状态中，归约项目的向前搜索符是 ‖，而移进项目指示要移进的输入符号为 )，不存在移进-归约冲突。

### 3.5 LALR（1） 分析

我们先来简单对比一下 LR（1） 和 SLR（1） 分析。首先，LR（1） 分析比 SLR（1） 分析强大，可以采用 SLR（1） 分析的文法一定可以采用 LR（1） 分析；但反之不成立。实际上，有结果表明：在所有以确定方式工作的分析程序中，LR（1） 分析程序已经达到了最强的分析能力。

然而，相比 SLR（1） 分析，LR（1） 分析的一个大的问题就是构造复杂度高得多。通常，一个 SLR（1） 文法的 LR（0） 有限状态机的状态数目，要比它的 LR（1） 有限状态机的状态数目少得多。

在这一小节里，我们介绍另外一种构造复杂度折衷的方法，称为 LALR（1） 分析。这种方法得到与 LR（0） 有限状态机相同数目的状态，但保留了 LR（1） 方法的部分能力。

#### 3.5.1 LALR（1） 有限状态机

通过合并 LR（1） 有限状态机中的同芯状态，便可得到相应的 LALR（1） 有限状态机。LR（1） 项目 $[A \to \alpha . \beta,\ a]$ 中的 $A \to \alpha . \beta$ 部分称为该项目的 **芯**。对于LR（1） 有限状态机的两个状态，如果只考虑每个项目的芯时它们是完全相同的项目集合，那么这两个状态就是**同芯状态**。

**例 15** 指出图 20 中 LR（1） 有限状态机的所有同芯状态。

**解** 根据定义，图 22 中 LR（1） 有限状态机的同芯状态共有 9 组：（1）$I_2$ 和 $I_{12}$；（2）$I_4$ 和 $I_9$；（3）$I_3$，$I_8$ 和 $I_{13}$；（4）$I_6$，$I_{14}$ 和 $I_{17}$；（5）$I_7$ 和 $I_{15}$；（6）$I_{10}$，$I_{18}$ 和 $I_{19}$；（7）$I_{11}$，$I_{22}$ 和 $I_{24}$；（8）$I_{16}$ 和 $I_{21}$；（9）$I_{20}$，$I_{23}$ 和 $I_{25}$。

**合并同芯状态**是将相互之间是同芯的状态合并为同一个状态，即将所有这些状态的项目集合全部并起来。

对于 LR（1） 有限状态机的两个同芯状态，由 GO 函数得到的针对任何符号的后继状态仍然是同芯的（请读者分析一下其中原因）。据此，我们很容易改造原有的GO 函数使其适应新的状态集合。这样，所有合并后的新状态和未合并的状态，以及改造后的GO 函数一起构成了一个新的LR（1） 有限状态机，称之为 **LALR（1） 有限状态机**。

**例 16** 从图 20 的 LR（1） 有限状态机构造相应的 LALR（1） 有限状态机。

**解** 将例 15 的 9 组同芯状态合并，得到 9 个新状态：（1）$I_{2\text{-}12}$（合并 $I_2$ 和 $I_{12}$）；（2）$I_{4\text{-}9}$（合并 $I_4$ 和 $I_9$）；（3）$I_{3\text{-}8\text{-}13}$（合并 $I_3$，$I_8$ 和 $I_{13}$）；（4）$I_{6\text{-}14\text{-}17}$（合并 $I_6$，$I_{14}$ 和 $I_{17}$）；（5）$I_{7\text{-}15}$（合并 $I_7$ 和 $I_{15}$）；（6）$I_{10\text{-}18\text{-}19}$（合并 $I_{10}$，$I_{18}$ 和 $I_{19}$）；（7）$I_{11\text{-}22\text{-}24}$（合并 $I_{11}$，$I_{22}$ 和 $I_{24}$）；（8）$I_{16\text{-}21}$（合并 $I_{16}$ 和 $I_{21}$）；（9）$I_{20\text{-}23\text{-}25}$（合并 $I_{20}$，$I_{23}$ 和 $I_{25}$）。最后，与没有合并的状态一起，并改造 GO 函数，我们得到相应的 LALR（1） 有限状态机，如图 23 所示。

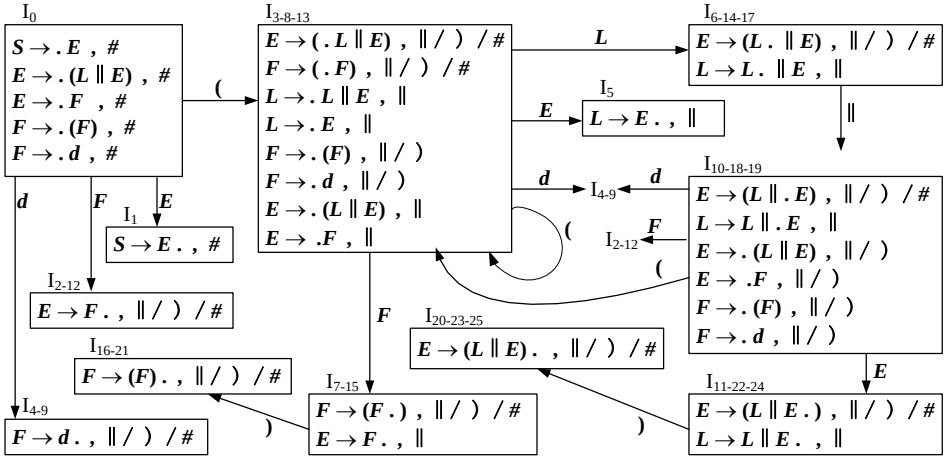



**图 23　从 LR（1） 有限状态机构造相应的 LALR（1） 有限状态机**

在实际实现时，是不会先构造 LR（1） 有限状态机，然后再通过合并状态来构造相应的 LALR（1） 有限状态机，而是会采取更有效的方法，如一边构造一边合并：LALR（1）FSM 的状态和 GO 函数的构造过程同LR（1）FSM，但每产生一个新状态后，需判断其是否与已有状态“同芯”；若是，就并到“同芯”的状态（合并向前搜索符），否则才加入此状态。

### 3.5.2　LALR（1） 分析表

LALR（1） 分析表可以直接从 LALR（1） 有限状态机得到，构造过程与图 21 所描述的完全一致。

**例 17** 利用例 16 得到的 LALR（1） 有限状态机，构造例 11 中文法的 LALR（1） 分析表。

**解** 根据图 21 所描述的构造过程，所得到的 LALR（1） 分析表如图 24 所示。

| 状态 | ACTION | | | | | GOTO | | |
|---|---|---|---|---|---|---|---|---|
| | *d* | ‖ | ( | ) | # | *E* | *L* | *F* |
| 0 | s4-9 | | s3-8-13 | | | 1 | | 2-12 |
| 1 | | | | | acc | | | |
| 2-12 | | r2 | | r2 | r2 | | | |
| 3-8-13 | s4-9 | | s3-8-13 | | | 5 | 6-14-17 | 7-15 |
| 4-9 | | r6 | | r6 | r6 | | | |
| 5 | | r4 | | | | | | |
| 6-14-17 | | s10-18-19 | | | | | | |
| 7-15 | | r2 | | s16-21 | | | | |
| 10-18-19 | s4-9 | | s3-8-13 | | | 11-22-24 | | 2-12 |
| 11-22-24 | | r3 | | s20-23-25 | | | | |
| 16-21 | | r5 | | r5 | r5 | | | |
| 20-23-25 | | r1 | | r1 | r1 | | | |

**图 24　从 LALR（1） 有限状态机构造 LALR（1） 分析表举例**

一个 LR（1） 文法，如果将其 LR（1） 有限状态机的同芯状态合并后所得到的 LALR（1） 有限状态机中的全部状态无归约–归约冲突，则该文法是一个 **LALR（1） 文法**。注：合并同芯状态后，不会产生新的移进–归约冲突（读者可分析一下为什么？），但有可能产生新的归约–归约冲突。

可以验证，图 23 中的状态没有归约–归约冲突，所以例 11 中的文法是 LALR（1） 文法。

由于 LALR（1） 有限状态机的状态可由 LR（1） 有限状态机的同芯状态合并后得到，因此其状态数目实际上与 LR（0） 有限状态机相同。

那么，LALR（1） 方法的分析能力又如何呢？首先，由于向前搜索符号都是对应非终结符 Follow 集中的符号，所以 LALR（1）分析不弱于 SLR（1） 分析。进一步，我们来回顾例 11 中文法的 LR（0） 有限状态机（图 19）中的冲突状态 $I_7$，该冲突不能用 SLR（1） 方法解决。而在图 23 的 LALR（1） 有限状态机中，对应于同样活前缀的状态是 $I_{7\text{-}15}$，该状态不存在移进-归约冲突。由此可知，LALR（1）分析要强于 SLR（1） 分析。

另一方面，LALR（1） 分析要弱于 LR（1） 分析。例如，对于下列文法 *G*(*S*)：

$$S \to A\,a \mid cA\,b \mid B\,b \mid c\,B\,a$$
$$A \to d$$
$$B \to d$$

可以验证：该文法是一个 LR（1） 文法，但不是 LALR（1） 文法。验证过程留作练习。

LALR（1）分析方法在实际的自动构造工具中使用较多，正是因为它有比较强的分析能力，同时构造复杂度要比 LR（1）分析方法低很多。如常用的自动构造工具 *Yacc*，可以生成一个基于 LALR（1）分析方法的语法分析及语法制导翻译程序。

# 4. 二义文法在 LR 分析中的应用

对于一些非 LR 文法，即不是 LR（1） 文法，如果我们附加一些合理的限定，是有可能采用 LR 分析方法进行语法分析的。例如，我们考虑如下文法 *G*(*E*)：

(1) $E \to E + E$

(2) $E \to E * E$

(3) $E \to ( E )$

(4) $E \to v$

(5) $E \to d$

这是一个二义文法，它一定不是 LR 文法（因为最右推导不唯一）。但是，如果对它的语义（即可以产生的语言）进行一些合理的限定，那么是有可能构造出相应的 LR 分析程序。例如，我们对二元算符 '+' 和 '*' 规定一种优先级和结合性后，就可以实现一种基于该文法的 SLR（1） 分析。

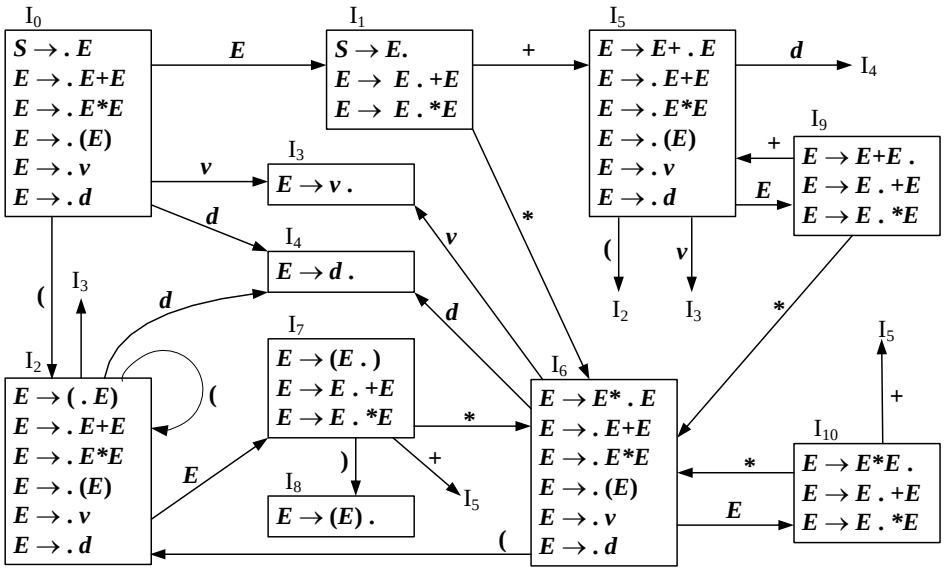



**图 25　一个二义文法的 LR（0） 有限状态机**

我们先给出 $G(E)$ 的 LR（0） 有限状态机（假设增广文法开始符号为 $S$），如图 25 所示。

可以看出，状态 $I_9$ 和 $I_{10}$ 存在移进–归约冲突。因为 + 和 * 属于 Follow($E$) = {+, *, ), #}，所以，该文法不是 SLR（1）文法。

但是，如果我们规定 * 的优先级高于 +，以及 * 和 + 都服从左结合性，则可以解决 $I_9$ 和 $I_{10}$ 中的移进–归约冲突：

- 对于 $I_9$：若遇*，则移进；若遇+，则归约。
- 对于 $I_{10}$：无论遇*，还是遇+，都归约。

这样，我们便可给出 $G(E)$ 的 一个 SLR（1） 分析表，如图 26 所示。

| 状态 | ACTION | | | | | | | GOTO |
|---|---|---|---|---|---|---|---|---|
| | *v* | *d* | * | + | ( | ) | # | *E* |
| 0 | s3 | s4 | | | s2 | | | 1 |
| 1 | | | s6 | s5 | | | acc | |
| 2 | s3 | s4 | | | s2 | | | 7 |
| 3 | | | r4 | r4 | | r4 | r4 | |
| 4 | | | r5 | r5 | | r5 | r5 | |
| 5 | s3 | s4 | | | s2 | | | 9 |
| 6 | s3 | s4 | | | s2 | | | 10 |
| 7 | | | s6 | s5 | | s8 | | |
| 8 | | | r3 | r3 | | r3 | r3 | |
| 9 | | | s6 | r1 | | r1 | r1 | |
| 10 | | | r2 | r2 | | r2 | r2 | |

**图 26　一个二义文法的 SLR（1） 分析表**

又如，对以下一个二义文法 *G*[S]：

(1) S → if *S* else *S*
(2) S → if *S*
(3) S → *a*

其中 if ，else ，*a* 为终结符。若规定最近嵌套匹配原则：else 优先匹配左边靠近它的未匹配的 if，则可以构造该文法的一个 LR 分析表。我们将这项工作留作练习。

# 5. LR 分析中的错误处理

与自顶向下分析过程的错误处理相比，自底向上分析处理语法错误更准确。原因在于推导和归约过程有如下差别：推导时仅观察可推导出的输入串的一部分，而归约时可归约的输入串整体已全部出现，并且输入符号是在查看后才被移进的。

对于 LR 分析而言，在当前扫描过的输入部分不存在正确的后跟符号时，就测试到一个错误。这样，仅使用 LR 分析表的 *action* 部分就能检测所有的错误。下面，我们简单介绍两种常用的 LR 分析错误恢复方法。

类似于介绍 LL（1）分析错误处理时的情形，一种简单的错误恢复措施是**应急恢复**。这种方法的基本原理是试图隔离由所选非终结符推导的最短可能出错的子串，然后跳过这个子串，进而恢复分析过程。比如，以 SLR（1）分析为例，可以采取的一种方案是：找出一个最短的串 $\alpha w$，以及一个重要的非终结符 $A$，使得 $\alpha$ 是符号栈顶部的符号串，$w$ 是余留输入符号串中较短的前缀，$A \Rightarrow_{rm}^{*} \alpha w$，并且余留输入符号串中 $w$ 后面的符号 $a \in \text{Follow}(A)$；为了恢复，可以在符号栈顶部将 $\alpha$ 替换为 $A$，同时将状态栈顶部的 $|\alpha|$ 个状态替换为一个新状态 GOTO [*k*, *A*]，其中 $k$ 为去掉 $|\alpha|$ 个顶部的状态后状态栈的栈顶状态。

我们要介绍的另外一种错误恢复方法是基于 LR 分析表的**即兴恢复**，它的基本思想是：对于每一个等于 *Error* 的 ACTION 表项，确定一个最有可能的出错情形，并针对这个出错情形设计一种有创意的恢复过程用来执行恰当的恢复动作。虽然这一设计需要编译器设计者较高的智慧，比如需要借助于如计算心理学或人工智能等其它许多领域的综合知识，但无疑这是一个有进一步发展前景的构架。下面，我们只通过例子简单介绍这一方法，涉及到一些典

型的错误恢复动作，如对分析栈或者余留输入符号串中的一些符号进行修改、插入、或删除，但一定要确保这种修改可以避免无限循环，以使分析程序最终可以步入正常过程。

| 状态 | ACTION | | | | | | | GOTO |
|---|---|---|---|---|---|---|---|---|
| | *v* | *d* | * | + | ( | ) | # | *E* |
| 0 | s3 | s4 | e1 | e1 | s2 | e2 | e1 | 1 |
| 1 | e3 | e3 | s6 | s5 | e3 | e2 | acc | |
| 2 | s3 | s4 | e1 | e1 | s2 | e2 | e1 | 7 |
| 3 | e3 | e3 | r4 | r4 | e3 | r4 | r4 | |
| 4 | e3 | e3 | r5 | r5 | e3 | r5 | r5 | |
| 5 | s3 | s4 | e1 | e1 | s2 | e2 | e1 | 9 |
| 6 | s3 | s4 | e1 | e1 | s2 | e2 | e1 | 10 |
| 7 | e3 | e3 | s6 | s5 | e3 | s8 | e4 | |
| 8 | e3 | e3 | r3 | r3 | e3 | r3 | r3 | |
| 9 | e3 | e3 | s6 | r1 | e3 | r1 | r1 | |
| 10 | e3 | e3 | r2 | r2 | e3 | r2 | r2 | |

**图 27　一个含错误处理信息的 LR 分析表**

图 27 中的 LR 分析表，是在第 4 节的图 26 基础上添加了错误处理信息。每一个错误处理信息包括可能的报错信息和可能的恢复措施两个部分，分别列举如下：

- e1 – 可能的报错信息：缺少运算数；可能的恢复措施：插入 *d* 到符号栈顶部，插入 *j* 到状态栈顶部，其中 *j* 满足 ACTION [*k*, *d*] = s*j*，这里 *k* 为 ACTION 中当前 e1 所在行的状态。

- e2 – 可能的报错信息：右括号未匹配；可能的恢复措施：删除余留符号串的当前符号 )。

- e3 – 可能的报错信息：缺少运算符；可能的恢复措施：插入 * 到符号栈顶部，插入 *j* 到状态栈顶部，其中 *j* 满足 ACTION [*k*, *] = s*j*，这里 *k* 为 ACTION 中当前 e3 所在行的状态。

- e4 – 可能的报错信息；缺少右括号；可能的恢复措施：插入 ) 到符号栈顶部，插入 *j* 到状态栈顶部，其中 *j* 满足 ACTION [*k*, )] = s*j*，这里 *k* 为 ACTION 中当前 e4 所在行的状态。

我们来看图 27 中的 LR 分析表中错误处理信息的几个例子：

- ACTION [2, *] = e1。查看图 25 中 LR（0）有限状态机的状态 $I_2$，此状态下所期望的动作是移进 *v*，*d*，或 (，因而缺少运算数是很有可能的错误之一。对于缺少运算数时的恢复措施，可以将 *v* 或 *d* 插入到栈顶，这样分析就可以继续下去了。

- ACTION [5, ) ] = e2。查看图 25 中 LR（0）有限状态机的状态 $I_5$，此状态下所期望的动作是移进 *v*，*d*，或 (。由于当前遇到了符号 )，因而很有可能是由于前面缺了左括号而导致的错误，因此报告这个右括号很可能是多余的。此时，若将这个多余的右括号从余留符号串删掉，则分析就可以继续进行了。

- ACTION [3, *d*] = e3。查看图 25 中 LR（0）有限状态机的状态 $I_3$，此状态下的动作是按照产生式 $E \to d$ 进行归约，由于采用的是 SLR(1)分析方法，所以期望余留符号串的当前符号应当属于 Follow(*E*) = {+, *, ), #}。那么为什么只报告右括号未匹配，而不报其它错误呢？比如，为什么不报缺少右括号（e4），是因为到达状态 $I_3$ 时的活前缀中不可能有左括号，所以此时期望的不会是右括号。这一点，从图 25 中 LR（0）有限状态机看得出来。如果使用 LR(1)分析，则这种情况下所期望的向前搜索符号集合为 {+, *, #}。可见，报告错误信息 e3 是合适的。没有按照所期望的向前搜索符号为 # 进行报告，是因为程序结束前误写一个常量的可能性不大。对于缺少运算符时的恢复措施，可以将 + 或 * 插入到栈顶，这样就可以继续分析了。

- ACTION [7, #] = e4。查看图 25 中 LR（0）有限状态机的状态 $I_7$，此状态下所期望的动作是移进 +，*，或 )，因而缺少右括号是很有可能的错误之一。之所以没有报告缺少运算符，是因为程序结束前漏写一个右括号的概率要大很多。同样，对于缺少右括号的错误进行恢复时，可以将 ) 插入到栈顶。

显然，应当可以设计出更精确的报错信息和恢复措施，但这不是一件轻松的事，需要编译器设计者更多的智能支持。

# 6. LR(k) 文法以及几类分析文法之间的关系（选讲）

## 6.1 LR（k） 文法[1]

LR(1) 分析中是向前查看 1 个单词符号，然后进行移进或归约（或者，接受/出错）。这种方法要求文法必须是 LR(1)文法。如果允许向前查看任意 $k$（$k \geq 1$）个单词符号，可扩展至 LR($k$) 文法。

D. E. Knuth 首先提出 LR(k)文法[1]。我们先大致了解一下 D. E. Knuth 在论文[1]中 LR(k)文法的含义。

先引入关于句柄（handle）的一个记号：设$\alpha=X_1 \ldots X_n \ldots X_t$是某一句型，假设某分析树对应的句柄是 $X_{r+1} \ldots X_n$（$0 \leq r \leq n \leq t$），使用的是第 $p$ 个产生式，则记这个$\alpha$的**句柄**为（$n, p$）。

此外，令符号"#"是文法符号（终结符和非终结符）集合之外的符号。***k*-句型**是后跟 $k$ 个"#"的句型，这里 $k \geq 0$。

下面是论文[1]中描述的 **LR(*k*)文法**应满足的条件：

设有 $k$-句型 $\alpha=X_1 \ldots X_n X_{n+1} \ldots X_{n+k} Y_1 \ldots Y_u$ 和$\beta=X_1 \ldots X_n X_{n+1} \ldots X_{n+k} Z_1 \ldots Z_v$，其中 $u \geq 0$，$v \geq 0$，以及 $X_{n+1}$，…，$X_{n+k}$，$Y_1$，…，$Y_u$，$Z_1$，…，和 $Z_v$ 均不是非终结符（即为终结符或者"#"）。若（$n, p$）是$\alpha$的句柄，（$m, q$）是$\beta$的句柄，那么这个文法是 LR($k$)文法的条件是：$n=m$，以及 $p=q$。也就是说，一个文法是 LR($k$)文法，当且仅当任何句柄总可以由左边的串 $X_1 \ldots X_n$ 和右边的终结符或"#"长度为 $k$ 的符号串所唯一确定。

以下是关于 LR($k$) 文法的一些重要结论：

- 对于整数 $k>0$，一个上下文无关文法是否为 LR($k$) 文法是可判定的。

[1] 注：如同 Aho 和 Ullman 教材[2]，在讨论的全程，预设所有 CFG 中均不存在无用符号（useless symbols）。

- 对于一个上下文无关文法，是否存在一个整数 $k>0$ 使得该文法是 LR(k) 文法，是不可判定的。
- LR($k$) 文法是无二义文法。
- 如果 G 是一个 LR(K)文法，且 L = L(G)，则一定存在某个 DPDA ，其语言为 L。
- 如果语言 L 是确定的（deterministic）上下文无关语言，即 L 是某个 DPDA 的语言，那么一定存在一个 LR(1)文法 G，L = L(G)。
- 两个 LR(k) 文法的语言是否相等是可判定的。
- 任何 LL(k)文法都是 LR(k)文法。[4]

这些结论在本课程的课堂讲稿（ppt）中有列举，目的是引起大家关注，但只是作为选讲，不属于课程的主体内容，也不会进行考察。

为满足一些同学的兴趣，以下对其中个别结论的由来及其与本讲前面部分较为相关的内容进行一些介绍，对于其余结论仅指出可进一步阅读的参考材料。

为方便研究，人们使用了比 Knuth 原来定义[1]更方便的方式定义 LR($k$)文法[2,4,6]。下面，我们引用 Aho 和 Ullman 教材[2] 5.2.2 节中的 LR($k$) 文法定义。

设 CFG $G=(V, T, P, S)$，CFG $G'=(V', T, P', S')$ 是相应的增广文法，则 $G$ 是 ***LR(k)*文法**（$k\geq 0$），如果以下三个条件

（1）$S' \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w$

（2）$S' \Rightarrow^*_{rm} \gamma Bx \Rightarrow_{rm} \alpha\beta y$ 和

（3）$\text{First}_k(w) = \text{First}_k(y)$

蕴含 $\alpha Ay = \gamma Bx$, 即$\alpha=\gamma$，$A = B$，和$y = x$。

若存在某个 $k\geq 0$，文法 $G$ 是 $LR(k)$ 文法，则称 $G$ 是一个 ***LR* 文法**。

上述 $LR(k)$ 文法的定义中，$\text{First}_k$ 的含义同文档Lecture03中的定义。为方便，在引入两个新记号的同时，重新描述一下 $\text{First}_k$ 的定义：

$$\text{First}_k(\alpha) = \{ x\in T^*/k \mid \alpha \Rightarrow_{lm}^* x\beta \wedge |x|=k \text{ or } \alpha \Rightarrow^* x \wedge |x|<k \}$$

引入的两个新记号如下：

设有符号串 $w$ 以及非负整数 $k>0$，定义 $\boldsymbol{w/k}$ 为：若$|w|\leq k$，则 $w/k$ 为 $w$；否则，$w/k$ 为 $w$ 的长度为 $k$ 的前缀。

设 $R$ 为符号串的集合，定义 $\boldsymbol{R/k} = \{w/k \mid w \in R\}$。

从这一定义，我们可以直接推论出上面所列的第 3 个结论。

**推论**：任何 $LR(k)$ 文法一定是无二义文法。

**证明** 设 $G=(V, T, P, S)$ 是 $LR(k)$ 文法，CFG $G'=(V', T, P', S')$ 是相应的增广文法，需要证明在$G'$或$G$中，任何 $z\in T^*$均有唯一的最右推导，若不然，则一定存在如下两个不同的最右推导：

$S' \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta w \Rightarrow^*_{rm} z$ 和

$S' \Rightarrow^*_{rm} \gamma Bx \Rightarrow_{rm} \alpha\beta w \Rightarrow^*_{rm} z$

满足 $\alpha Aw \neq \gamma Bx$，其中 $x,y \in T^*$。针对$LR(k)$定义中的条件，取$y$为$w$，显然有$\text{First}_k(w) = \text{First}_k(y)$，因此这三个条件蕴含着 $\alpha Aw = \gamma Bx$。所以上述假设不成立。证毕。

然而，该结论的逆命题不成立，如 Knuth 论文[1]中给出的如下例子：

$S \to aAc$
$A \to bAb$
$A \to b$

显然，这是一个无二义文法。但，该文法不是任何 $LR(k)$ 文法（$k \geq 0$）。

设 CFG $G=(V, T, P, S)$，$A \to \beta_1 \beta_2$ 是 $P$ 中的产生式，以及 $u \in T^*/k$，称 $[A \to \beta_1 \cdot \beta_2, u]$ 为一个**针对 $G$ 和 $k$ 的 $LR(k)$ 项目**，通常在上下文足够清楚时简称 **$LR(k)$ 项目**。

**$LR(k)$ 项目** $[A \to \beta_1 \cdot \beta_2, u]$ 针对 $G$ 的活前缀$\alpha\beta_1$是**有效的**，如果存在推导 $S \Rightarrow^*_{rm} \alpha Aw \Rightarrow_{rm} \alpha\beta_1\beta_2 w$，使得 $u = \text{First}_k(w)$。注：这里的 $\beta_1$ 可为ε；显然，任何活前缀至少有一个有效的 LR(k) 项目。

定义 ε-*free first* 函数 $\text{EFF}^G_k(\alpha)$（如果上下文无疑义，可忽略 $k$ 和 $G$）如下：

（1）若$\alpha$不以非终结符开头，则 $\text{EFF}_k(\alpha) = \text{First}_k(\alpha)$；

（2）若$\alpha$以非终结符开头，则 $\text{EFF}_k(\alpha) = \{ w \mid$ 存在推导 $\alpha \Rightarrow^*_{rm} \beta \Rightarrow_{rm} wx$，其中$\beta \neq Awx$（$A$为任意非终结符），以及有 $w = \text{First}_k(wx) \}$。

以下结论对理解 LR 分析技术很有帮助，也可以看作 **$LR(k)$ 文法的一个充要条件**。（参见 Aho 和 Ullman 教材[2] 中 5.2.3 节定理 5.9）

**结论**: CFG $G=(V, T, P, S)$ 是 $LR(k)$ 文法，当且仅当对每个 $u \in T^*/k$，下列条件成立：设 $\alpha\beta$ 为 $G$ 的增广文法 $G'$中右句型$\alpha\beta w$的活前缀，若 $LR(k)$ 项目$[A \to \beta \cdot, u]$ 针对$\alpha\beta$是有效的，则不存在任何其他 $LR(k)$ 项目 $[A_1 \to \beta_1 \cdot \beta_2, v]$，该项目针对活前缀$\alpha\beta$也是有效的，且满足 $u \in \text{EFF}_k(\beta_2 v)$。(注：$\beta_2$ 可为 ε)

基于这一结论，Aho 和 Ullman 教材[2] 中 5.2.4 节给出了 $LR(k)$ 文法的一个判定算法（本讲 3.4 节的 LR(1)文法的判定是其特例）。从而，可以证明上面所列的第 1 个结论，“对于整数 $k>0$，一个上下文无关文法是否为 LR($k$) 文法是可判定的”。另外，教材[2]5.2.5 节给出了规范 $LR(k)$分析算法， 3.4 节关于规范 LR(1)分析算法是其特例。同样，这一结论也可以参考 Knuth 论文[1]中第 II 节的讨论。

教材[2]的 $LR(k)$ 文法定义与 Knuth 的原始 $LR(k)$ 文法以及其他典型的定义[6]相比，当 $k>0$ 时是相互等价的，但对于 $k=0$ 时，教材[2]的定义相对严格一些。这一差异我们在此不多论述，不会对我们关于其他问题的讨论产生影响。

上面所列的第 2 个结论，“对于一个上下文无关文法，是否存在一个整数 $k>0$ 使得该文法是 LR(k) 文法，是不可判定的”，可参见 Knuth 论文[1]中第 IV 节的相应定理，也可参见教材[2] 中的习题 5.2.12。

上面所列的第 4 和第 5 个结论，刻画了 LR(K)文法和 DPDA 之间的等价性，即从 LR(K)文

法可以构造与之等价的 DPDA（第 4 个结论），而从 DPDA 可以构造与之等价的 LR(1)文法（第 5 个结论）。能被 DPDA 接受的语言称为确定的（deterministic）上下文无关语言[7]，因此，与 LR(K)文法对应的语言类即为确定的上下文无关语言类。这两个结论的证明可参考 Knuth 论文[1]的第 V 节。第 5 个结论也可参考 Aho 和 Ullman 教材[2]的定理 8.16，同时在该定理中也叙述了如下结论：每一个满足前缀性质（prefix property）的确定的上下文无关语言，均存在一个可以接受它的 LR(0)文法。

值得注意的是，从（上面所列的第 4 和第 5）这两个结论，可以推论出：任何 LR(K)文法可以构造与之等价的 LR(1)文法。这一点与 LL(*k*)文法之间的层次关系有显著的差异。

上面所列的倒数第 2 个结论，“两个 LR(k) 文法的语言是否相等是可判定的”，其正确性至少可以从列表中第 4 个结论（从 LR(K)文法可以构造与之等价的 DPDA）以及“两个 DPDA 的 等价性是可判定的”这一结论推论出来，后者的一个证明参见论文[3]。

上面所列的最后一个结论，“任何 LL(k)文法都是 LR(k)文法”，其证明可参考 Aho 和 Ullman 教材[2]的定理 8.1。该结论说明 LL(k)语言类包含于 LR(k)语言类。同时，参考该教材的习题 8.1.11 给出一个确定的上下文无关语言的例子，没有 LL(k)文法可以接受这一语言。这说明，LL(k)语言类严格包含于 LR(k)语言类。

## 6.2 几类分析文法之间的关系

本课程第 6 讲和第 8 讲中涉及到多种类别的分析文法，作为一个小结，我们在图 28 中给出了这些文法类之间的关系。从图中可知：

- 所有的 LR（0）文法都是 SLR（1）文法。
- 所有的 SLR（1）文法都是 LALR（1）文法。
- 所有的 LALR（1）文法都是 LR（1）文法。
- 所有的 LL（1）文法都是 LR（1）文法。
- LL（1）文法类与 LR（0）、SLR（1）及 LALR（1）文法类之间没有相互包含关系。

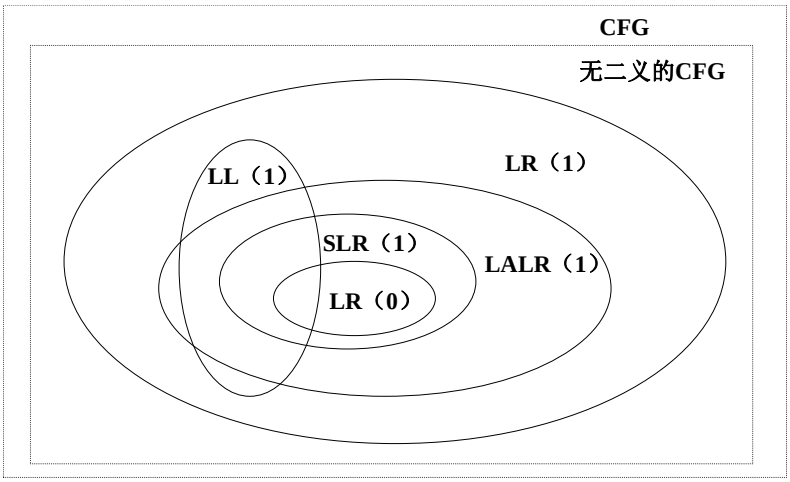



**图 28　若干文法类之间的关系**

关于上述最后一条的讨论，下列两个例子可供参考。

一个例子是下列文法 *G*[*S*]：

$$S \to AB$$
$$A \to Aa \mid B$$
$$B \to b$$

该文法是 LR(0)文法，自然也是 SLR(1)和 LALR(1)文法，但不是 LL(1)文法（因为含左递归）。

另一个例子是 J. C. Beatty 论文[4]中的下列文法

$$S \to a\,F \mid b\,G$$
$$F \to X\,c \mid Y\,d$$
$$G \to X\,d \mid Y\,c$$
$$X \to I\,A$$
$$I \to \varepsilon$$
$$A \to \varepsilon$$
$$Y \to I\,B$$
$$B \to \varepsilon$$

可以验证，该文法是 LL(1)文法，但不是 LALR(1)文法。

# 课后作业

1. 下面的文法 G[S] 描述由命题变量 p、q ，联结词 ∧（合取）、∨（析取）、¬（否定）构成的命题公式集合：

   S → S∨T | T
   T → T∧F | F
   F → ¬F | p | q

   试分别指出句型 ¬F ∨ ¬ q ∧ p 和 ¬F ∨ p ∧ ¬F ∨ T 的所有短语，直接短语。如果这些句型同时也是右句型，那么还要给出其句柄 . 请将结果填入下表中:

| 句型 | 短语 | 直接短语 | 句柄 |
|---|---|---|---|
| ¬F∨¬q∧p | | | |
| ¬F∨p∧¬F∨T | | | |

2. （1）给定文法 *G*[S]：

   $S \to A\,B$
   $A \to a\,A \mid \varepsilon$
   $B \to B\,b \mid \varepsilon$

（*a*）构造该文法 *G*[S] 的 LR（0）有限状态机。

（*b*）说明该文法不是 LR(0) 文法。

（*c*）该文法是否 SLR(1) 文法？为什么？

（2）给定文法 *G*[S]：

$S \to S\,S \mid (\,S\,) \mid a$

完成同（1）一样三个问题（*a*），（*b*），（*c*）。

3. 试构造下列文法的 LR（0）FSM，并判别它们是否 LR(0)或 SLR(1)文法:

a) 文法 G[E]:

E → E + T | T
T → ( E ) | id | id [ E ]

其中 E,T 为非终结符,其余符号其余符号 id，（，），[，] 为终结符

b) 文法 G[S]:

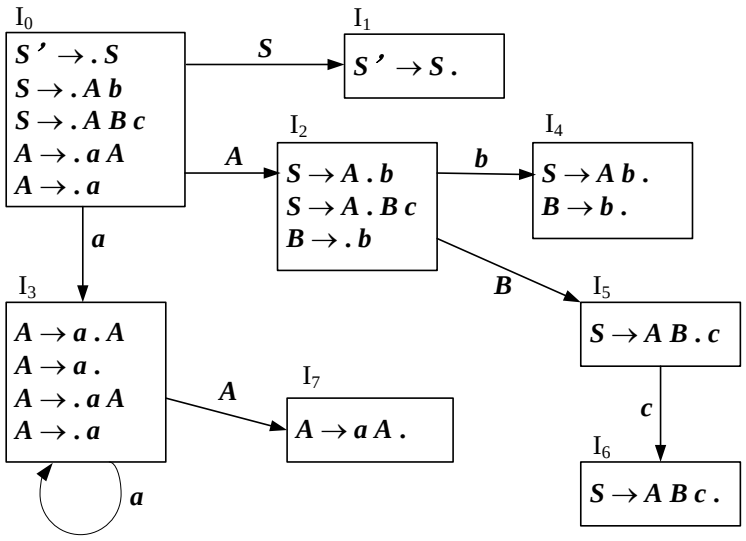
S → Ab | ABc
A → aA | a
B → b

其中 S,A,B 为非终结符,其余符号为终结符

4. 给定 SLR(1) 文法 G[S]：

（1）$S \to A\,b$
（2）$S \to A\,B\,c$
（3）$A \to a\,A$
（4）$A \to a$
（5）$B \to b$

其 LR(0) 有限状态机如下图所示：



（*a*）构造该文法的 SLR(1) 分析表。

（*b*）若采用 SLR(1) 方法对于 *L*(*G*) 中的某一输入串进行分析，当栈顶出现句柄 *a* 时，余留输入符号串中的第一个符号是什么？

5. 给定 SLR(1) 文法 G[S]：

（1）$S \rightarrow a\ S\ a$
（2）$S \rightarrow b\ S\ b$
（3）$S \rightarrow c$

（*a*）构造该文法的 LR(0) 有限状态机

（*b*）构造该文法的 SLR(1) 分析表。

（*c*）若根据以上 SLR(1)分析表对于 *L*(*G*) 中的某一输入串执行 SLR(1) 分析过程，初始时符号栈存放符号 #。当扫描过串 *abbcb* 后，分析栈中的符号串是什么（以进栈先后次序给出）？当前可归约的句柄是什么？

6. 给定文法 *G*[S]:

$S \rightarrow SS \mid aSb \mid \varepsilon$

（*a*）构造该文法的 LR（1）有限状态机。

（*b*）该文法是否 LR(1) 文法？为什么？

7. 对于下列文法 *G*(*S*)：

$S \rightarrow Aa \mid cAb \mid Bb \mid cBa$
$A \rightarrow d$
$B \rightarrow d$

试验证：该文法是一个 LR（1）文法，但不是LALR（1） 文法。

8. （1）构造下列增广文法 G[P’]的 LR(1) FSM，验证原文法是 LR(1)文法:

(0) P’→ P
(1) P → P(P)
(2) P → Aa
(3) P → ε
(4) A → ε

其中 P’,P,A 为非终结符

（2） 通过合并同芯集（状态）的方法构造相应于上述 LR(1) FSM 的 LALR(1) FSM，并判断原文法是否 LALR(1)文法？

9. 给定文法 *G*[S]:

（1）$S \rightarrow A\ a$
（2）$S \rightarrow b\ A\ c$
（3）$S \rightarrow d\ c$
（4）$S \rightarrow b\ d\ a$

（5）$A \rightarrow d$

（*a*）构造该文法的 LR（1）有限状态机。

（*b*）验证该文法是 LR(1) 文法。

（c）该文法是否 LALR(1) 文法?

（d）给出对应的 LR(1) 分析表。

10. 给定增广文法 G[S’]:

(0) S’ → S
(1) S → do S or S
(2) S → do S
(3) S → S;S
(4) S → act

其中 S’，S 为非终结符,其余符号为终结符。

a） 构造 LR(0)FSM；
b） 判别原文法是否 LR(0)或 SLR(1)文法；
c） 若对一些终结符的优先级或结合性作如下规定：

or 优先性大与 do;
; 优先性大与 do
; 优先性大与 or
; 服从左结合性

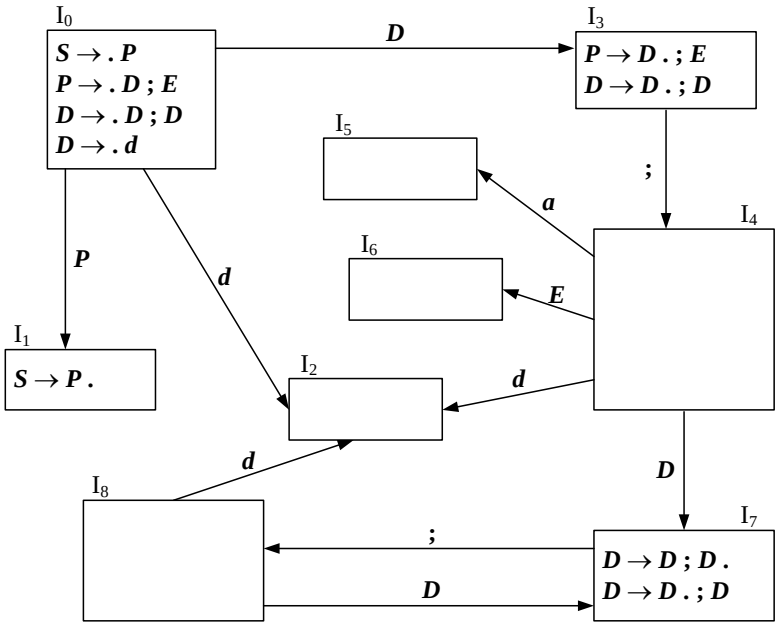
试构造原文法的一个 LR 分析表。

11. 给定如下文法 G(P)：

(1) $P \rightarrow D ; E$
(2) $D \rightarrow D ; D$
(3) $D \rightarrow d$
(4) $E \rightarrow a$

为文法 $G(P)$ 增加产生式 $S \rightarrow P$，得到增广文法$G'[S]$。

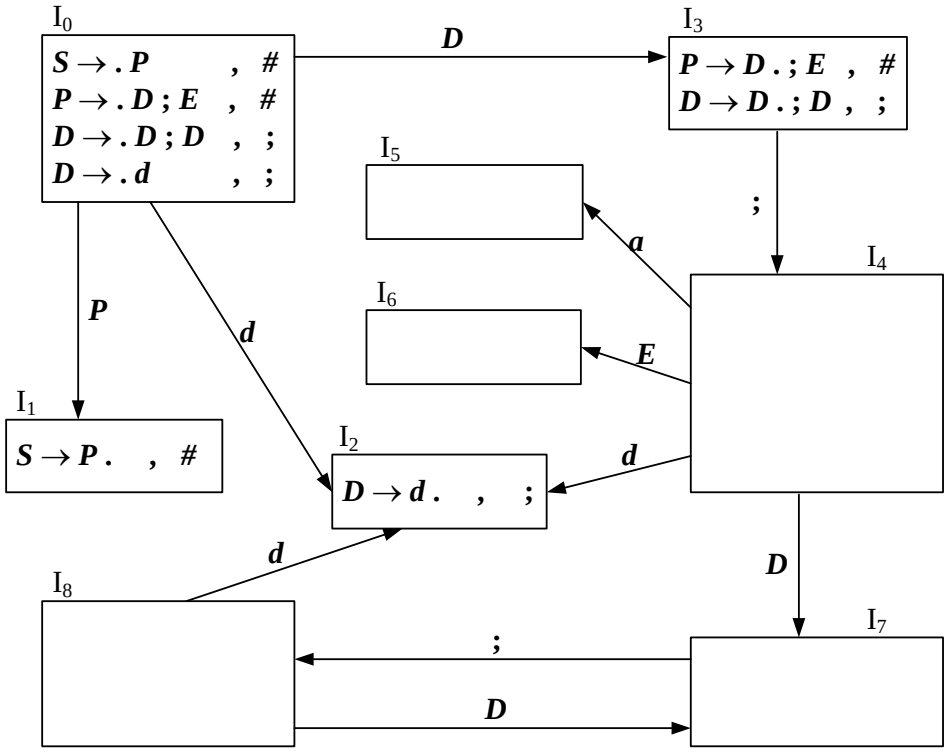
a) 下图表示该文法的LR(0)自动机，部分状态所对应的项目集未给出，试补齐之（即分别给出状态 $I_2$，$I_4$，$I_5$ ，$I_6$，和 $I_8$ 对应的项目集。




b) 指出上图中LR(0)自动机中所有冲突的状态（并指出是哪种类型的冲突），以说明该文法不是 LR(0) 文法。

c) 针对所有冲突的状态，说明这些冲突不适合采用 SLR(1) 分析方法解决。

d) 下图表示该文法的LR(1)自动机，部分状态所对应的项目集未给出，试补齐之（即分别给出状态 $I_4$，$I_5$，$I_6$ ，$I_7$，和 $I_8$ 对应的项目集。




e) 指出上图中LR(1)自动机中所有冲突的状态（并指出是哪种类型的冲突），以说明该文法不是 LR(1) 文法。

f) 尽管该文法不是LR文法，但仍有可能采用LR分析方法完成语法分析。（1）试给出一种采用SLR(1)分析方法的解决方案；（2）根据你所给的方案完成下图的 SLR(1) 分析表，状态 4、5、7 和8 对应的行未给出，试补齐之 （若你的方案与下图不匹配，请给出相应的完整SLR(1) 分析表）。

| 状态 | ACTION | | | | GOTO | | |
|---|---|---|---|---|---|---|---|
| | *a* | *d* | ; | # | *P* | *D* | *E* |
| 0 | | $s_2$ | | | 1 | 3 | |
| 1 | | | | acc | | | |
| 2 | | | $r_3$ | | | | |
| 3 | | | $s_4$ | | | | |
| 4 | | | | | | | |
| 5 | | | | | | | |
| 6 | | | | $r_1$ | | | |
| 7 | | | | | | | |
| 8 | | | | | | | |

12. 已知某文法 *G*[S] 的 LALR(1)分析表如下：

| 状态 | ACTION | | | | | GOTO |
|---|---|---|---|---|---|---|
| | *a* | *t* | *g* | *c* | # | *S* |
| 0 | s11 | s8 | | s4 | | 1 |
| 1 | | | | s2 | acc | |
| 2 | | | s3 | | | |
| 3 | s11 | s8 | | s4 | | 16 |
| 4 | s5 | | | | | |
| 5 | s6 | | | | | |
| 6 | | | | s7 | | |
| 7 | | | r1 | r1 | R1 | |
| 8 | | | s9 | | | |
| 9 | | | | s10 | | |
| 10 | s11 | s8 | | s4 | | 14 |
| 11 | s11 | s8 | | s4 | | 12 |
| 12 | | | s13 | s2 | | |
| 13 | s11 | s8 | | s4 | | 15 |
| 14 | | | r4 | s2 | R4 | |
| 15 | | | r2 | s2 | R2 | |
| 16 | | | r3 | s2 | R3 | |

并且已知各规则右边语法符号的个数以及左边的非终结符如下：

| 规则编号 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 右部长度 | 4 | 4 | 4 | 4 |
| 左部符号 | S | S | S | S |

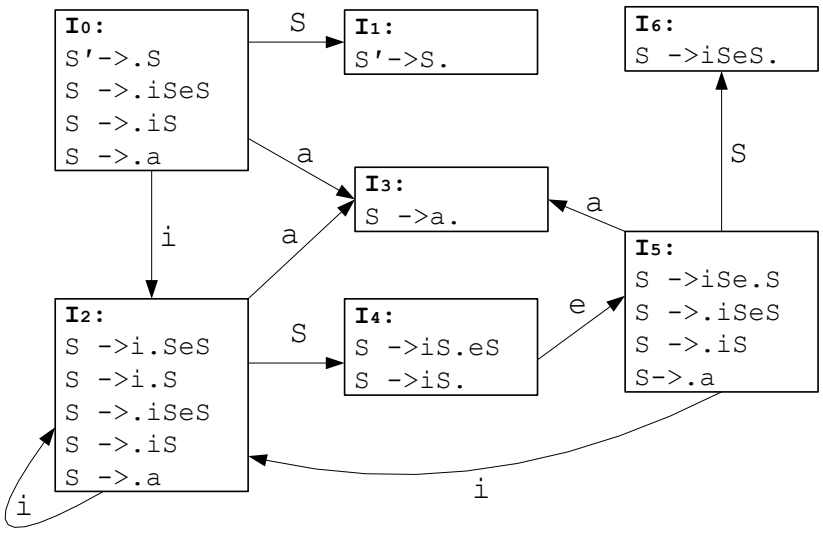
写出使用上述 LALR(1)分析器分析下面串的过程（只需写出前 10 步，列出所有可能的 r*i* ，s*j* 序列，注意先后次序）：

*acaaccgtgccaacgatgccaa* …

13. 给定如下文法 *G*[*S*]：

(1) $S \rightarrow$ if $S$ else $S$
(2) $S \rightarrow$ if $S$
(3) $S \rightarrow$ a

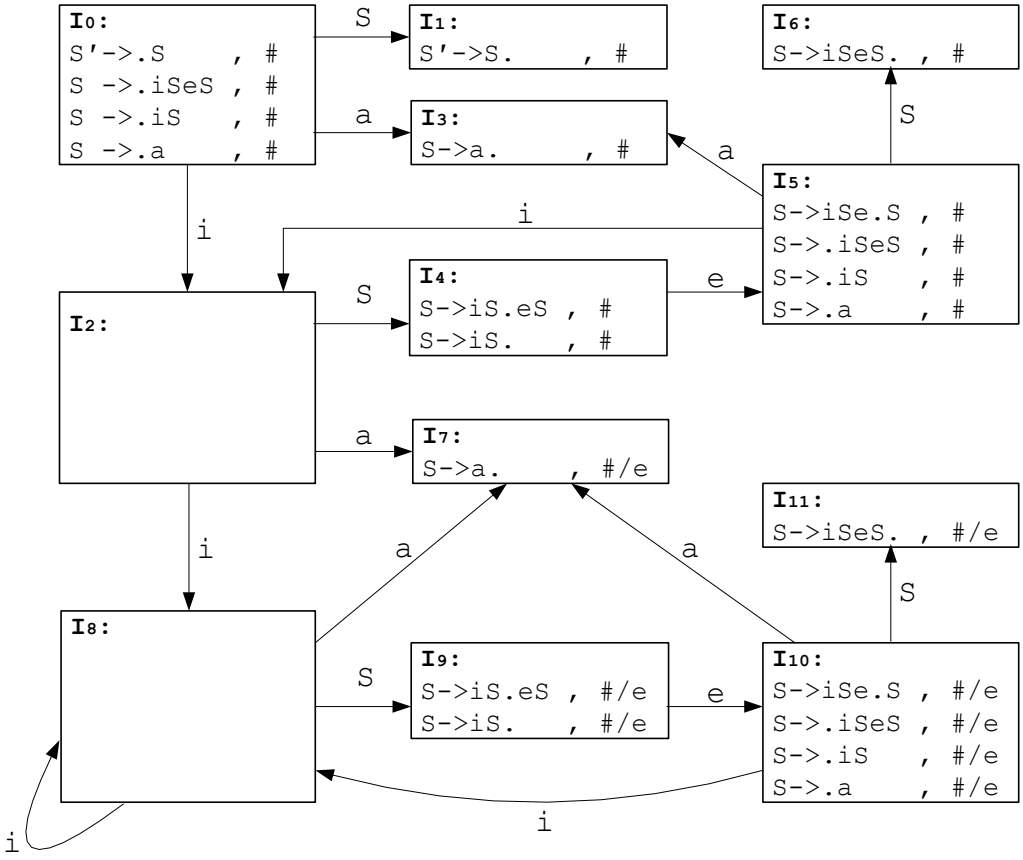
为文法 G[S] 增加产生式 $S' \rightarrow S$，得到增广文法 *G*'[*S*']，下图是相应的LR(0)自动机（i 表示 if，e 表示 else）：




（1）指出LR(0)自动机中的全部冲突状态及其冲突类型，以说明文法G[S]不是LR(0)文法。

（2）文法*G*[*S*]也不是SLR(1)文法。为什么？

（3）下图表示文法G[S]的LR(1)自动机，部分状态所对应的项目集未给出，试补齐之（即分别给出状态 $I_2$，$I_8$， 和 $I_{10}$ 对应的项目集。




（4）指出LR(1)自动机中的全部冲突状态，这说明文法 G[S] 也不是 LR(1) 文法。

（5）若规定最近匹配原则，即 else 优先匹配左边靠近它的未匹配的if，则可以解决上述2个自动机中的状态冲突。下图表示文法G[S]在规定这一规则情况下的SLR(1)分析表，状态 4~6 对应的行未给出，试补齐之 。

| 状态 | ACTION | | | | GOTO |
|---|---|---|---|---|---|
| | i | e | a | # | S |
| 0 | s2 | | s3 | | 1 |
| 1 | | | | acc | |
| 2 | s2 | | s3 | | 4 |
| 3 | | r3 | | r3 | |
| 4 | | | | | |
| 5 | | | | | |
| 6 | | | | | |

下图表示文法G[S] 在规定这一规则情况下的LR(1)分析表，状态 4，7 和 9 对

应的行未给出，试补齐之 。

| 状态 | ACTION | | | | GOTO |
|---|---|---|---|---|---|
| | i | e | a | # | S |
| 0 | s2 | | s3 | | 1 |
| 1 | | | | acc | |
| 2 | s8 | | s7 | | 4 |
| 3 | | | | r3 | |
| 4 | | | | | |
| 5 | s2 | | s3 | | 6 |
| 6 | | | | r1 | |
| 7 | | | | | |
| 8 | s8 | | s7 | | 9 |
| 9 | | | | | |
| 10 | s8 | | s7 | | 11 |
| 11 | | r1 | | r1 | |

（6）对于文法G[S]中正确的句子，基于上述两个分析表均可以成功进行LR分析。然而，对于不属于文法G[S]中的句子，两种分析过程发现错误的速度不同，即发现错误时所经过的移进/归约总步数有差异。试给出一个长度不超过10的句子（即所包含的终结符个数不超过10），使得两种分析过程发现错误的速度不同。哪一个更快？对于你给的例子，两种分析过程分别到达哪个状态会发现错误？

# 附录 A　理解 LR 分析理论的一个重要引理： LR(0) FSM 可以也只能读进所分析文法的增广文法的活前缀。

**证明思路：**

需要证明两个方面：

1　**所有活前缀一定都可由 LR(0) FSM 读进，即不会错过合法的归约。**

2　**LR(0) FSM 只能读进增广文法的活前缀。**

我们首先了解增广文法的活前缀是如何产生的。增广文法活前缀的集合 *Prefix* 可归纳定义如下：

（1） 设 $S'$是增广文法的开始符号，即有产生式 $S' \to S$（$S$ 是原文法 $G$ 的开始符号），令 $S \in Prefix$。

（2） 若 $v \in Prefix$，则 $v$ 的任一前缀 $u$ 都是活前缀，即 $u \in Prefix$。

（3） 若 $v \in Prefix$，且 $v$ 中至少包含一个非终结符，即可以将 $v$ 写成 $\alpha B\gamma$，其中 $B$ 为非终结符。若有产生式 $B \to \beta$，则$\alpha\beta$ 的任一前缀 $u$ 都是活前缀，即 $u \in Prefix$。

（4） *Prefix* 中的元素只能通过上述步骤产生。

这里，我们先假定 *Prefix* 的确可以表示增广文法所有活前缀的集合（证明附后），然后通过证明下列两个命题来分别说明上述两个方面均成立。

命题 1 欲证：*Prefix* 中所有活前缀一定都可由 LR(0) FSM 读进。我们根据 *Prefix* 的定义进行归纳证明。

基础：对于由规则（1）产生的活前缀 $S \in Prefix$，由于在 LR(0) FSM 的初始项目集（状态）$I_0$ 中含有项目 $S' \to \cdot S$，显然 $S$ 是可以被 LR(0) FSM 读进的。

归纳：若 $v \in Prefix$，且 $v$ 可以被 LR(0) FSM 读进，则 $v$ 的前缀可以被 LR(0) FSM 读进也是显然的。

若 $v \in Prefix$，且可以将 $v$ 写成 $\alpha B\gamma$，其中 $B$ 为非终结符并有产生式 $B \to \beta$。设 LR(0) FSM 在从初态 $I_0$ 读进 $\alpha$ 后进入状态 I。因为 $v=\alpha B\gamma$ 可以被 LR(0) FSM 读进，所以在状态 I 可以读进 $B$，根据 LR(0) FSM 的构造过程，一定存在项目 $C \to \alpha \cdot B\gamma' \in I$。由闭包的计算过程，可知一定有 $B \to \cdot \beta \in I$，因此 $\beta$ 是从 I 开始后续的可读串。所以，从初态 $I_0$ 开始，$\alpha\beta$ 的任一前缀 $u$ 都是可读进的。

命题 1 证毕。

命题 2 LR(0) FSM 能读进的序列均属于 *Prefix*。我们归纳于 LR(0) FSM 可读序列的长度 $n$ 来证明。

基础：$n$=0。显然，空序列 $\varepsilon \in Prefix$。

归纳：设 $\alpha X$ 是 LR(0) FSM 可读序列，且 $|\alpha X| = n+1$，其中 X 是某个文法符号。在读过 $\alpha$ 后，LR(0) FSM 一定处于某状态 I，在此状态下 X 是可读的。根据 LR(0) FSM 的构造过程，一定存在项目 $C \to \beta \cdot X\gamma \in I$。该项目或者是基本项目，或者是通过闭包计算得到的项目。下面分这两种情形来讨论。

1） $C \to \beta \cdot X\gamma$ 是基本项目。

若 $|\beta| = 0$，则该项目只能是 $S' \to \cdot S$，此时$\alpha X = S$ 显然是属于 *Prefix* 的。

若 $|\beta| = m \neq 0$，则 LR(0) FSM 从状态 $I_0$ 读进 $n-m$ 个符号的序列 $\mu$ 后进入状态 I′，且必定有 $C \to \cdot \beta X\gamma \in I'$。根据 LR(0) FSM 的构造过程，一定存在项目 $B \to \cdot C\gamma' \in I'$。这样在状态 I′下，可以读进 $B$。因为 $|\mu B| \leq n$，由归纳假设，可知 $\mu B$ 属于 *Prefix*。因此，由 *Prefix* 的归纳定义，得知 $\mu\beta X = \alpha X$ 是属于 *Prefix* 的。

2） $C \to \beta \cdot X\gamma$ 是通过闭包计算得到的项目。

此时，$\beta$ 一定为空序列，而项目 $C \to \cdot X\gamma$一定是由 I 中的某个基本项目 $B \to \mu \cdot C_0\nu$ 直接或间接地通过闭包计算序列得到的：$C_0 \to \cdot C_1\gamma_1$，$C_1 \to \cdot C_2\gamma_2$，…，

$C_k \to \cdot C\,\gamma_{k+1}$。由 1）的讨论结果，可知 $\alpha C_0$ 属于 *Prefix*。从而 $\alpha C_1$，$\alpha C_2$，…，$\alpha C_k$ 都属于 *Prefix*，$\alpha C$ 也属于 *Prefix*。所以，$\alpha X$ 是属于 *Prefix* 的。

命题 2 证毕。

下面，我们补充证明 *Prefix* 的确可以表示增广文法所有活前缀的集合。

对于文法 $G = (V_N, V_T, P, S)$，设 $S'$是其增广文法 $G'$ 的开始符号。

任取 $G'$ 的活前缀 $\gamma$，则存在$\alpha, \beta \in (V_N \cup V_T)^*$，$w \in V_T^*$，满足 $S' \Rightarrow_{rm}^* \alpha Aw$ 以及 $A \Rightarrow \beta$，且 $\gamma$ 是 $\alpha\beta$ 的前缀。现在，我们归纳于最右推导 $S' \Rightarrow_{rm}^* \alpha Aw$ 的推导步数 $n$，证明 $\gamma \in Prefix$。

若 $n$=0，则一定有 $\alpha Aw$ 为 $S'$，以及 $S'$唯一的产生式 $S' \to S$，满足 $S' \Rightarrow_{rm}^* S' \Rightarrow S$，此时 $\gamma$ 是 $S$ 的前缀。根据 *Prefix* 定义的归纳基础，$S \in Prefix$。根据 *Prefix* 定义的第 2 条归纳规则，我们有 $\gamma \in Prefix$。

若 $n$>0，则一定有 $S' \Rightarrow_{rm}^* \alpha' A' w' \Rightarrow_{rm} \alpha A w$，其中 $A' \Rightarrow \beta'$，满足 $\beta' = \alpha'' A w''$，$\alpha = \alpha' \alpha''$ 和 $w = w'' w'$，以及 $A \Rightarrow \beta$，且 $\gamma$ 是 $\alpha\beta$ 的前缀。根据归纳假设，$\alpha'\beta'$的所有前缀均属于 *Prefix*，$\alpha'\beta'$本身即$\alpha'\alpha''Aw''$也属于 *Prefix*。根据 *Prefix* 定义的第 3 条归纳规则，$\alpha'\alpha''\beta$ 即 $\alpha\beta$ 的任一前缀均属于 *Prefix*，所以有 $\gamma \in Prefix$。

另一方面，假设 $\gamma \in Prefix$，我们归纳于 *Prefix* 的定义，来证明 $\gamma$ 是 $G'$ 的活前缀。

若 $\gamma$ 为 $S$，因为 $S' \Rightarrow_{rm}^* S' \Rightarrow S$，即 $S$ 是相对 $S'$ 的句柄，所以 $S$ 本身即 $\gamma$ 是 $G'$ 的活前缀。

若 $\gamma$ 由 *Prefix* 定义的规则（2）产生，即存在 $v \in Prefix$，$\gamma$ 是 $v$ 的前缀。根据归纳假设，我们有 $S' \Rightarrow_{rm}^* \alpha Aw$ 且 $A \Rightarrow \beta$，$v$ 是 $\alpha\beta$ 的某个前缀，因此 $\gamma$ 也是 $\alpha\beta$ 的前缀，即 $\gamma$ 是 $G'$ 的活前缀。

若 $\gamma$ 由 *Prefix* 定义的规则（3）产生，即存在 $v \in Prefix$，可以将 $v$ 写成 $\alpha B\gamma'$，其中 $B$ 为非终结符，以及 $B \to \beta$ 是 $G'$ 的产生式，$\gamma$ 是 $\alpha\beta$ 的某个前缀。根据归纳假设，我们有 $S' \Rightarrow_{rm}^* \alpha' Aw'$ 且 $A' \Rightarrow \beta'$，$v$ 是 $\alpha'\beta'$ 的前缀。也就是说，$\alpha B\gamma'$ 是 $\alpha'\beta'$ 的前缀。这样，$S' \Rightarrow_{rm}^* \alpha' Aw' \Rightarrow_{rm} \alpha'\beta' w' = \alpha B\gamma'\beta' w' \Rightarrow_{rm}^* \alpha B w''$（注：我们默认所有文法符号都是有用符号）。因 $\gamma$ 是 $\alpha\beta$ 的前缀，所以 $\gamma$ 是 $G'$ 的活前缀。

（如发现问题，请及时讨论）

# 参考文献


[1] DONALD E. KNUTH, On the Translation of Languages from Left to Right, INFORMATION AND CONTROL 8, 607-639, 1965.

[2] Alfred V. Aho, Jefferey D. Ullman. The Theory of Parsing, Translation, and Compiling, Volume 1 & Volume 2, Prentice-Hall Series in Automatic Computation，1972.

[3] Géraud Sénizergues, The Equivalence Problem for Deterministic Pushdown Automata is Decidable, ICALP '97 Proceedings of the 24th International Colloquium on Automata, Languages and Programming, Lecture Notes in Computer Science Volume 1256, pp 671-681, 1997.

[4] John C. Beatty. On the relationship between LL(1) and LR(1) grammars, Journal of the ACM, Vol. 29, No. 4, pp 1007–1022, 1982.

[5] John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman. Introduction to Automata Theory, Languages, and Computation, 2nd / 3nd editions, Addison Wesley, 2001/. 清华大学出版社影印（第 2 版）, 2002, 机械工业出版社影印（第 3 版），2008.

[6] M.M. Geller and M.A. Harrison, On LR(k) grammars and languages, Theor. Comput. Sci. 4(3)，pp 245-276，1977.

[7] Seymour Ginsburg and SheilaGreibach, Deterministic context free languages, Information and Control，Volume 9, Issue 6, pp 620-648, 1966.