第四章　语法分析

语法分析是整个编译过程中最重要的一个环节。语法分析阶段需扫描单词序列形式的源程序，分析源程序在语法上是否合法。任何一个高级程序设计语言都有一套规则（称为语法规则）来描述合法程序的语法结构，例如一个 C 语言程序是由一个或多个函数构成，每个函数又是由预处理指令和函数体构成，函数体由系列语句构成等等。语法规则一般采用文法的形式，上下文无关文法可以描述高级语言大部分的语法结构。描述语法结构的文法是程序设计者和编译器开发者共同的界面，即程序设计者根据文法书写应用程序，编译器开发者根据文法实现编译器。本章首先对两类主要的语法分析方法进行了概述，然后重点介绍自顶向下的语法分析方法——LL(1)分析和自底向上的语法分析方法——LR 分析，最后简要介绍了语法分析器的自动生成工具——YACC。

4.1 语法分析概述

语法分析器的主要功能是识别由词法分析给出的单词序列是否是给定的（上下文无关）文法的正确的句子，如果是，为源程序构造反映其语法结构的分析树，如果不是，需要指出错误在源程序中出现的位置和错误的性质。语法分析器在编译器模型中的位置见图 4-1。

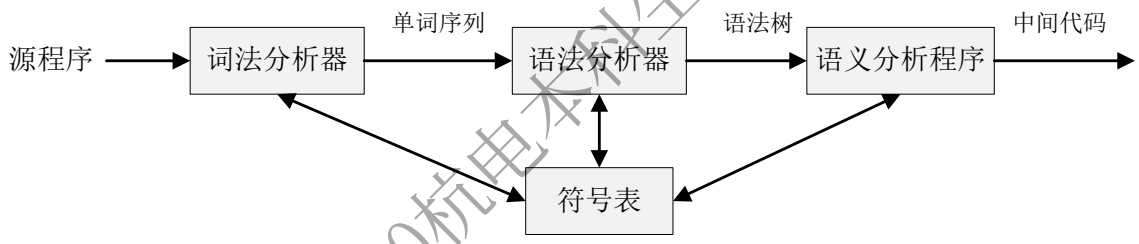

图 4-1 编译器模型中语法分析器的位置

常见的语法分析方法有两种类型，一种是自顶向下的语法分析方法（Top-Down Parsing），一种是自底向上的语法分析方法（Bottom-Up Parsing）。

自顶向下语法分析方法

从左到右扫描单词序列（句子），一次读入一个单词符号，按自顶向下的顺序构造分析树，即首先构造分析树的根结点，然后构造根结点的子结点，一层一层往下构建，直到构建完成所有的叶子结点。这个自顶向下构造分析树的过程，对应了一个从文法的开始符号开始到生成句子结束的推导序列。

【例 4-1】：考虑以下文法，

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F \qquad (4.1)$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

这个文法是表达式文法，其中 E 是表达式，表达式由一组项以“+”号连接构成；T 是项，项由一组因子以“*”号连接构成；F 表示因子，可以是括号括起来的表达式，也可以是变量标识符。

试分析该文法的一个句子：id*id。该句子分析树的一个自顶向下的构造过程见图 4-2。

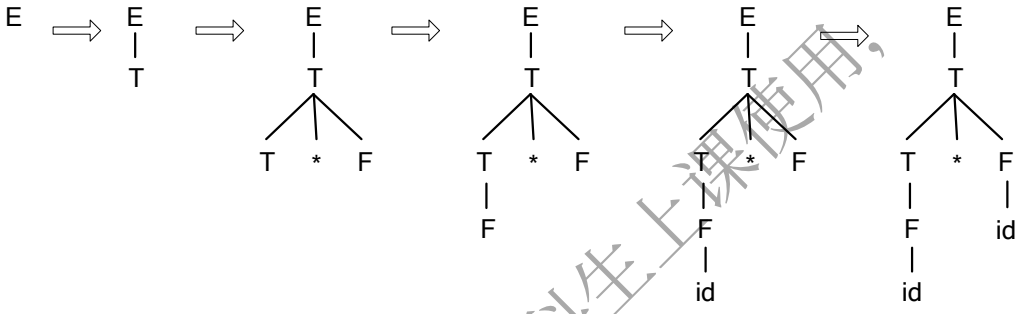

图 4-2 例【4-1】自顶向下构造分析树

这个分析树的构造过程对应了一个推导序列，见（4.2）。

$$E \Rightarrow T \Rightarrow T*F \Rightarrow F*F \Rightarrow id*F \Rightarrow id*id \qquad (4.2)$$

对于合法的句子，一个自顶向下分析方法可以给出句子的一个自顶向下（一般还同时要求从左到右）构造分析树的过程，或者给出一个从开始符号开始推导出句子的推导序列（一般还要求是最左推导序列）。

自底向上语法分析方法

从左到右扫描单词序列（句子），一次读入一个单词符号，按自底向上的顺序构造分析树，即首先构建叶子结点，然后一层一层往上构建内部结点，最后构建根结点。这个自底向上构造分析树的过程，对应了一个从句子开始到文法开始符号结束的归约序列。

【例 4-2】：考虑文法（4.1），分析该文法的一个句子：id*id。

解：该句子分析树的一个自底向上的构造过程见图 4-3。

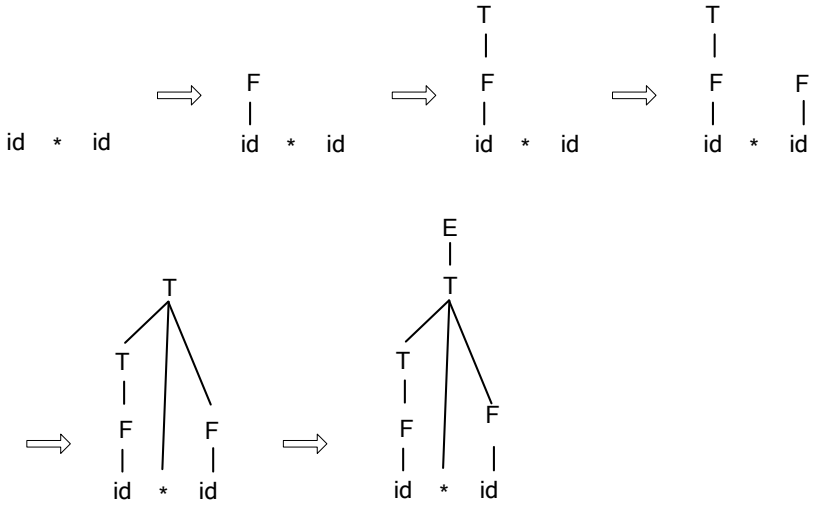

图 4-3 例【4-2】自底向上构造分析树

这个分析树的构造过程对应了一个归约序列，见（4.3）。

$$id*id \Leftarrow F*id \Leftarrow T*id \Leftarrow T*F \Leftarrow T \Leftarrow E \qquad (4.3)$$

对于合法的句子，一个自底向上分析方法可以给出句子的一个自底向上（一般还同时要求从左到右）构造分析树的过程，或者给出一个将句子归约为开始符号的归约序列（一般还要求是最左归约序列）。

这两种分析方法都是按照从左向右的方式来扫描待分析的句子，而且是每次读入一个单词（或者向前看一个单词）。最高效的自顶向下分析方法和自底向上分析方法只能处理某些满足特殊要求的文法，比如 LL(1)文法和 LR 文法。好在这些文法的表达能力已经足以描述现代程序设计语言的大部分语法结构了。手工实现的语法分析器通常基于 LL(1)文法，比如预测分析器，对于处理较大的 LR 文法的语法分析器通常是使用自动化工具构造得到的。

在本章及后续章节，我们经常采用算术表达式文法作为例子，因为算术表达式是高级语言中的典型结构，而且由于运算符的结合性和优先级特性使得分析算术表达式更具挑战性。实际上，处理算术表达式的语法分析技术可以用来处理程序设计语言的大部分结构。在后续讨论中，除非特别指出我们一般讨论的都是无二义的文法，对于每一个合法的句子，都只有一棵分析树，也只对应一个最左（最右）推导或者最左（最右）归约。

4.2 自顶向下语法分析方法

4.2.1 不确定的自顶向下分析

从为句子构造最左推导的角度来看，一个自顶向下分析过程就是一个为句型最左边的非终结符号选择产生式一步步往下推导的过程。由于限定是做最左推导，因此在每步推导过程中需要展开的非终结符号是确定的，唯一不确定的是如何为该非终结符号选择推导要用的产生式，因为该非终结符号可能会有多条候选产生式。

在选择产生式时如果采用的是随机策略，这样一种自顶向下分析方法称为**不确定的自顶向下分析**。由于每一个合法的句子只有一棵分析树，也只有一个最左推导序列，每步推导选择的产生式只有一条是正确的。如果选择产生式是随机的，不能保证每次的选择是正确的。如果在某步推导中选择了不正确的产生式，分析过程将无法完成，当发现推导无法匹配句子的时候需回过头再随机选择另外一条产生式往下推导。这样一个回头选择另外一条产生式进行推导的操作，称为**回溯**。不确定的自顶向下分析可能会有大量的回溯操作。

【例 4-3】：考虑以下文法，

S →AB

A → aA | ε

B → b | bB

解：试分析输入 aaabb。一个可能的分析过程见表 4-1。

表 4-1 句子 aaabb 的不确定性自顶向下分析

步骤	推导结果	输入串	说明
1	S	aaabb	推导从开始符号 S 开始
2	⇒ AB	aaabb	选择产生式 S→AB
3	⇒ aAB	aaabb	选择产生式 A→aA，匹配第 1 个 a
4	⇒ aaAB	aaabb	选择产生式 A→aA，匹配第 2 个 a
5	⇒ aaaAB	aaabb	选择产生式 A→aA，匹配第 3 个 a
6	⇒ aaaaAB	aaabb	选择产生式 A→aA，无法匹配下一符号
6’	⇒ aaaεB	aaabb	回溯，取消上一步骤，选择另一条产生式 A→ε
7	⇒ aaab	aaabb	选择产生式 B→b，匹配第 1 个 b，推导结束，无法匹配第 2 个 b
7’	⇒ aaabB	aaabb	回溯，取消上一步骤，选择另一条产生式 B→bB，匹配第 1 个 b
8	⇒ aaabb	aaabb	选择产生式 B→b，匹配第 2 个 b，推导结束

在以上分析过程中，执行了两次回溯操作。分别取消了步骤 6 的这次推导，代之以步骤 6’，取消了步骤 7 的这次推导，代之以步骤 7’。最终寻找到了需要的最左推导序列，推出了要分析的句子。

不确定的自顶向下分析本质上是一种基于穷举原理的试探方法，是一个反复使用不同的候选产生式谋求匹配句子的过程。其不确定性体现在每次选择的产生式不一定是正确的，在分析过程中可能会有大量的回溯操作。不确定的自顶向下分析方法虽然对文法几乎没有什么

要求，适用的文法很广，但是由于回溯造成分析的效率低下，实现代价极高，因此这种方法只有理论价值，实践中很少采用这种方法。

4.2.2 确定的自顶向下分析

在自顶向下分析过程中，每步推导都能根据句子中下一个要匹配的符号，及与文法相关的一些信息，选出唯一正确的产生式，这样一种自顶向下分析方法称为**确定的自顶向下分析**。采用确定的自顶向下分析方法，无论待分析的句子是否合法，都不会有回溯，分析的效率高。如果句子是合法的，可以一次性地构造出最左推导的序列，如果句子是不合法的，在句子出现语法错误的位置，将无法挑选出正确的产生式继续往下推导。

确定的自顶向下分析，其确定性体现在分析过程中能选择正确的产生式。确定的自顶向下分析又称为**预测分析**。这种分析方法在选择产生式时，除了需要考察句子中下一个要匹配的符号之外，还需要利用事先计算的与文法相关的一些信息，只有满足一定的条件才能实现确定性的分析，因此对文法是有限制的，并不是任意的文法都适合这种分析方法。

那么在分析过程中需要哪些和文法相关的信息来帮助我们选择产生式呢？

【例 4-4】：考虑以下文法，

S →Ap

S →Bq

A → a

A → cA

B → b

试分析输入串 ccap。

解：正确的最左推导过程见表 4-2。

表 4-2 句子 ccap 的确定性自顶向下分析

步骤	推导结果	输入串	说明
1	S	ccap	推导从开始符号 S 开始
2	⇒ Ap	ccap	选择产生式 S→Ap
3	⇒ cAp	ccap	选择产生式 A→cA，匹配第 1 个 c
4	⇒ ccAp	ccap	选择产生式 A→cA，匹配第 2 个 c
5	⇒ ccap	ccap	选择产生式 A→a，匹配 ap

以上分析过程中共选择了 4 条产生式进行推导，每条产生式都是唯一正确的。在步骤 2 中，需要对 S 进行展开，而 S 有两条候选产生式，其右部分别是“Ap”和“Bq”，这时候为什么选择第 1 条产生式用“Ap”去替换 S 而不是选择第 2 条产生式用“Bq”去替换 S 呢？

因为句子中下一个要匹配的符号（第 1 个符号）是“c”，经过分析发现“Ap”往下推导是可以推出以“c”开始的串的，也就是说是可以匹配句子中的下一个符号的，而“Bq”往下推导无法推出以“c”开始的串，也就无法匹配句子中的下一个符号，因此选择 S 的第 1 条产生式用“Ap”去替换 S 是正确的。

同理，在步骤 3 中，需要对 A 进行展开，A 也有两条候选产生式，其右部分别是“a”和“cA”，这时候句子中的下一个要匹配的符号还是“c”，此时只能选择 A→cA 去匹配“c”，选择 A→a 是无法匹配下一个符号的。采用同样的分析，可以为步骤 4、步骤 5 确定正确的产生式。

以上例子说明，在确定产生式过程中每条产生式右部能推导出哪些终结符号（即能推导出哪些以终结符号开始的串）是很重要的信息。令句子中的下一个要匹配的符号是“x”，下面要对 A 进行展开，A 的一条候选产生式的右部为 α，如果 α 能推出以“x”开始的串，那么这条候选产生式是可选择的。α 能推出哪些终结符号为首的串，这些终结符号是事先可以计算的，α 能推出的所有终结符号构成一个集合，称为 α 的首符号集，记为 **FIRST(α)**。

假设 A 还有另外一条候选产生式 A→ β，若同时满足 x∈FIRST(β)，则 A→ β 也是可以选择的。如果存在这种情况，我们就无法确定唯一正确的产生式往下推导，也就无法实现确定的自顶向下分析。因此一个文法要做确定的自顶向下分析，同一个非终结符号的多个候选产生式，其右部的首符号集的交集必须为空集。

以上例子说明了计算产生式右部 FIRST 集的重要性，下面再看一个例子。

【例 4-5】：考虑以下文法，

S → aA

S → d

A → bAS

A → ε

试分析输入串 abd。

解：正确的最左推导过程见表 4-3。

表 4-3 句子 abd 的确定性自顶向下分析

步骤	推导结果	输入串	说明
1	S	abd	推导从开始符号 S 开始
2	⇒ aA	abd	选择产生式 S→aA，匹配 a
3	⇒ abAS	abd	选择产生式 A→bAS，匹配 b
4	⇒ ab ε S	abd	选择产生式 A→ ε，消除 A
5	⇒ abd	abd	选择产生式 S→d，匹配 d

在以上分析过程中，步骤 2、步骤 3、步骤 5 中的产生式选择可以根据当前候选产生式右部的 FIRST 集来确定（如【例 4-4】）。步骤 4 需要对 A 进行展开，此时句子中下一个要匹配的符号是“d”，A 有两个候选产生式，其中 A→bAS 显然推不出“d”，那么 A→ ε 这条产生式能采用吗？如何来判断？

如果用 A→ ε 往下推导，相当于是在当前句型中将 A 消除掉，如果我们事先知道，在包含 A 的文法的句型中紧跟在 A 的后面是可以出现下一个要匹配的符号“d”的，那么选择这条产生式是可能去匹配“d”，也即这条产生式是可以选择的。在文法的所有句型中，能出现在某个非终结符号 A 后面的所有终结符号构成的集合，称为 A 的后跟符号集，记为 FOLLOW(A)。

【例 4-5】说明，当某个非终结符号的候选产生式右部为空，计算该非终结符号的后跟符号集是非常重要的。一般地，令候选产生式为 A→ α （ α 等于 ε，或者 α 能推出 ε），句子中下一个要匹配的符号为“x”，如果 x∈FOLLOW(A)，则 A→ α 是可以选择的。要实现确定的自顶向下分析，除了 A 的多个候选产生式的右部的 FIRST 集的交集为空集，FOLLOW(A)还应该和其它 A 的非空候选产生式的 FIRST 集的交集也为空集，从而确保每次选择唯一正确的产生式往下推导。

下面给出 FIRST 集和 FOLLOW 集的形式定义。

定义 4-1：FIRST(α)是由文法符号串 α 推导出的所有以终结符号开始的符号串的第一个终结符号组成的集合，即：

$$FIRST(\alpha) = \{ a \mid \alpha \overset{*}{\Rightarrow} a\cdots, a \in V_T \},$$

如果 $\alpha \overset{*}{\Rightarrow} \varepsilon$，则规定 ε ∈FIRST(α)。

下面考虑如何构建一个文法符号串的 FIRST 集。令 $\alpha = X_1X_2\cdots X_n$（其中 $x_i \in V_T \cup V_N$，$1 \le i \le n$），首先将 X_1 能推出的首个终结符号加入 FIRST(α)；若 X_1 能推出 ε，则再将 X_2 能推出的首个终结符号加入 FIRST(α)；若 X_2 也能推出 ε，则接着考察 X_3；依此类推，直到找到一个 X_j（j=1, 2, ..., n）不能推出 ε。若每个 x_i（i=1, 2, ..., n）都能推出 ε，则将 ε 也加入 FIRST(α)。如果 α = ε，则将 ε 加入 FIRST(α)。

【例 4-6】：考虑以下文法，

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \varepsilon \\ T &\rightarrow F\,T' \\ T' &\rightarrow *F\,T' \mid \varepsilon \\ F &\rightarrow (E) \mid id \end{aligned} \qquad (4.4)$$

求每个产生式右部的 FIRST 集。

解：首先求单个非终结符号 E、E’、T、T’、F 的 FIRST 集：

FIRST（E）= FIRST（T）= FIRST（F）= {（，id }

FIRST（E’）={ +， ε }

FIRST（T’）={ *， ε }

再求产生式右部的 FIRST 集：

FIRST（TE’）={（，id }

FIRST（+TE’）={ + }

FIRST（ ε ）={ ε }

FIRST（F T’）={（，id }

FIRST（*F T’）={ * }

FIRST（（E））={（ }

FIRST（id）={ id }

定义 4-2：非终结符号 A 的 **FOLLOW 集**，定义为：

FOLLOW（A）= { a | S $\overset{*}{\Rightarrow}$ …Aa… ，a∈V_T }。

构造非终结符号的 FOLLOW 集，遵循以下几条规则：

（1）（特例）令文法的开始符号是 S，则将“$”符号加入 FOLLOW（S）中。 在实际的语法分析中，往往在句子的结尾加一个特殊符号（如“$”符号）作为句子的结束标志，当扫描到这个符号的时候就知道整个句子已经分析完成了。句子由 S 代表，“$”紧跟在句子后面，因此$∈FOLLOW（S）。

（2）若有产生式 A → α B β （其中 α 、 β 均是文法符号串），则将 FIRST(β)中的非空符号并入 FOLLOW(B)中。

（3）若有产生式 A → α B 或 A → α B β （满足 β $\overset{*}{\Rightarrow}$ ε ），则将 FOLLOW(A)中的符号并入 FOLLOW(B)中。

【例 4-7】：考虑【例 4-6】中的文法（4.4），求每个非终结符号的 FOLLOW 集。

解：非终结符号 E、E’、T、T’、F 的 FOLLOW 集如下：

FOLLOW（E）= FOLLOW（E’）={），$}

FOLLOW（T）= FOLLOW（T’）={+，），$}

FOLLOW（F）= {+，*，），$}

通过对【例 4-4】和【例 4-5】的分析，我们知道在做确定的自顶向下分析时，需要事先计算文法中每个产生式右部的 FIRST 集和非终结符号的 FOLLOW 集。只有根据这些信息才能在每步推导过程中选出唯一正确的产生式。为方便选择产生式，在 FIRST 集和 FOLLOW 集的基础上定义 SELECT 集。

定义 4-3：给定文法产生式 A → α（其中 A∈V_N，α ∈$(V_T \cup V_N)^*$），

SELECT（A → α）定义为：

（1）若 α 不能推出 ε，则 SELECT（A → α）= FIRST（α）；

（2）若 $\alpha \overset{*}{\Rightarrow} \varepsilon$，则 SELECT（A → α）= FIRST（α）∪FOLLOW（A）。

【例 4-8】：考虑【例 4-6】中的文法（4.4），求每条产生式的 SELECT 集。

解：文法产生式的 SELECT 集如下：

SELECT（E → TE’）={（，id }

SELECT（E’ → +TE’）={ + }

SELECT（E’ → ε ）={ ε，），$ }

SELECT（T → F T’）={（，id }

SELECT（T’ → *F T’）={ * }

SELECT（T’ → ε ）={ ε，+，），$ }

SELECT（F → （E））={（ }

SELECT（F → id）={ id }

有了 SELECT 集的定义，在选择产生式时只需要考察 SELECT 集就可以了。在自顶向下分析过程中，假设当前句子中下一个要匹配的符号是“x”，待展开的非终结符号是 A，A→ α 是该非终结符号的一条候选产生式，若 x∈SELECT（ A→ α），则产生式 A→ α 是可以选择的。如果要做确定的自顶向下分析，每次选出的产生式最多只能有一条，这就要求文法满足一定的条件。LL（1）文法就是这样的满足条件的文法。

定义 4-4：一个文法是 **LL（1）文法**的充分必要条件是对每个非终结符号 A 的两个不同产生式 A→ α， A→ β，满足：

SELECT（ A→ α）∩ SELECT（ A→ β）= Φ

LL（1）文法中的第一个 L 是指在分析句子时是从左到右进行扫描的，第二个 L 是指分析过程中是要生成一个句子的最左推导，1 代表在作出分析前（选择产生式的时候）要向前看 1 个符号。

根据定义，文法（4.4）是 LL（1）的。对于任何一个 LL（1）文法，都可以对其句子做确定的自顶向下分析。

【例 4-9】：基于【例 4-6】中的文法（4.4），对句子 id+id*id 做确定性的自顶向下分析。

解：分析过程见表 4-4。

表 4-4 句子 id+id*id 的自顶向下分析

步	推导结果	句子	下一个符号	选择的产生式	说明
1	E	id+id*id$	id	E→TE’	id∈SELECT(E→TE’)
2	⇒ T E’	id+id*id$	id	T→F T’	id∈SELECT(T→F T’)
3	⇒ F T’ E’	id+id*id$	id	F→id	id∈SELECT(F→id)
4	⇒ id T’ E’	id+id*id$	+	T’ → ε	+∈SELECT(T’ → ε)
5	⇒ id ε E’	id+id*id$	+	E’ →+TE’	+∈SELECT(T’ → ε)
6	⇒ id + T E’	id+id*id$	id	T→F T’	id∈SELECT(T→F T’)
7	⇒ id + F T’ E’	id+id*id$	id	F→id	id∈SELECT(F→id)
8	⇒ id + id T’ E’	id+id*id$	*	T’ →*F T’	*∈SELECT(T’ →*F T’)
9	⇒ id + id * F T’ E’	id+id*id$	id	F→id	id∈SELECT(F→id)
10	⇒ id + id * id T’ E’	id+id*id$	$	T’ → ε	$∈SELECT(T’ → ε)
11	⇒ id + id * id ε E’	id+id*id$	$	E’ → ε	$∈SELECT(E’ → ε)
12	⇒ id + id * id ε ε	id+id*id$			

分析树的构造过程见图 4-4。

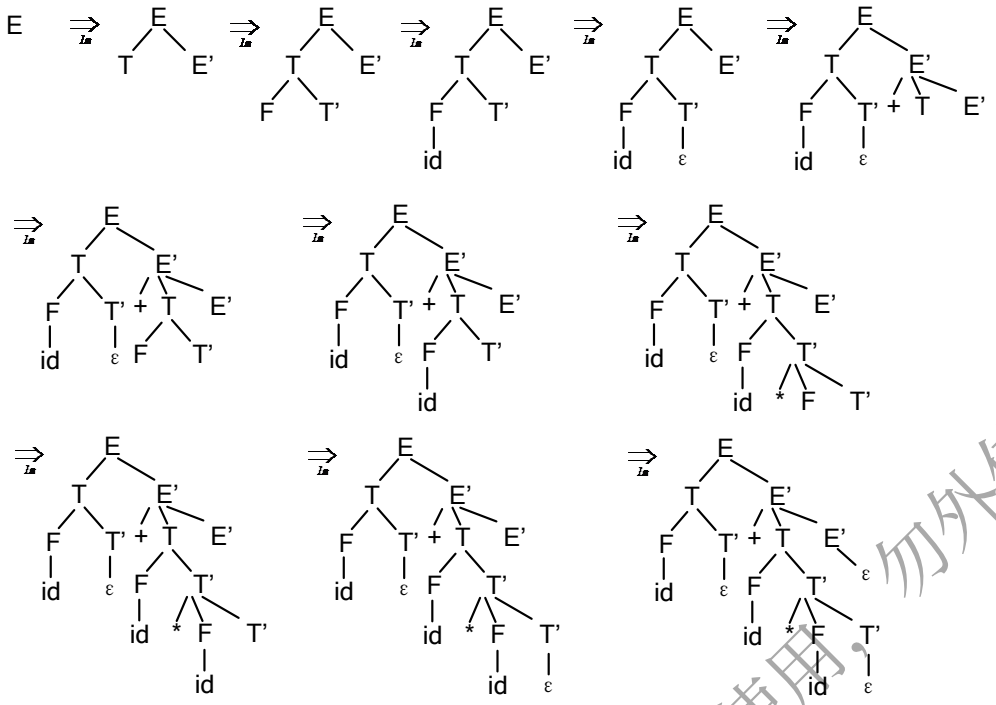

图 4-4 句子 id+id*id 的分析树的构造

4.2.3 非 LL（1）文法到 LL（1）文法的等价变换

一个语言可以有多个文法去描述它，但不是每个语言都一定会有 LL（1）文法。要对语言的句子做确定的自顶向下分析，必须基于此语言的 LL（1）文法。

有些文法带有非 LL（1）文法的特征，但是我们可以通过一定的变换操作，消除它的这个非 LL（1）特征，这样就有可能将它转化为等价的 LL（1）文法。

1、提取公共左因子

有些文法含有形如：A → α β | α γ（其中 α、β、γ 都是文法符号串）的产生式。其中 α 是产生式右部共同的部分，称为公共左因子。含有公共左因子的文法不是 LL（1）的，如上面这两条产生式的 SELECT 集里面都包含 FIRST（α），故：

SELECT（ A → α β）∩ SELECT（ A → α γ ） ≠ Φ

不满足 LL（1）文法定义的要求。

含有公共左因子的产生式的一般形式为：

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \mid \cdots \mid \alpha\beta_n \qquad (4.5)$$

在推导过程中，需要将 A 展开时，我们不知道应该将 A 展开为 $\alpha\beta_1$、$\alpha\beta_2$、.....，还是 $\alpha\beta_n$。一个解决方案是先将 A 展开为 αA’，从而将做出决定的时间往后延。在扫描完由 α 推导得到的符号串之后，我们再决定将 A’展开为 $\alpha\beta_1$、$\alpha\beta_2$、.....，还是 $\alpha\beta_n$。

基于以上分析，可以对含有公共左因子的产生式作如下变换：

（1）提取公共左因子，

$$A \rightarrow \alpha\ (\beta_1 |\ \beta_2 |\ \beta_3 | \cdots |\ \beta_n) \qquad (4.6)$$

（2）引入一个新的非终结符号 A’，对（4.6）进行等价变换，

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 |\ \beta_2 |\ \beta_3 | \cdots |\ \beta_n$$

【例 4-10】：考察如下含有公共左因子的文法，

$$S \rightarrow aSb$$

$$S \rightarrow aSa$$

$$S \rightarrow b$$

试提取该文法的公共左因子。

解：以上文法第 1、2 条产生式右部含有公共左因子“aS”，引入一个新的非终结符号 A，对以上文法进行等价变换，得：

$$S \rightarrow aSA$$

$$S \rightarrow b \qquad (4.7)$$

$$A \rightarrow a$$

$$A \rightarrow b$$

提取公共左因子之后，文法有可能转化为 LL（1）的。

如上例：

SELECT（ S → aSA）∩ SELECT（ S → b ）

$= \{a\} \cap \{b\}$

$= \Phi$

SELECT（ A → a）∩ SELECT（ A → b ）

$= \{a\} \cap \{b\}$

$= \Phi$

故（4.7）是 LL（1）文法。

【例 4-11】：考虑如下文法，

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

$$E \rightarrow b$$

解：该文法描述了高级语言中常见的条件语句，其中 i，t，e 分别表示 if，then 和 else，E 和 S 分别表示表达式和语句。

提取公共左因子之后，文法变换为：

$$S \rightarrow iEtSS' \mid a$$

$$S' \rightarrow eS \mid \varepsilon \qquad (4.8)$$

$$E \rightarrow b$$

提取公共左因子有可能将文法转化为 LL（1）的，也有可能文法仍然不是 LL（1）的。

如（4.8）中，

$$SELECT(S' \rightarrow eS) \cap SELECT(S' \rightarrow \varepsilon)$$

$$= \{e\} \cap \{e、\$\}$$

$$\neq \Phi$$

故（4.8）不是 LL（1）文法。

2、消除左递归

定义 4-5：含有形如 $A \rightarrow A\alpha \mid \beta$ 的产生式或 $A \overset{+}{\Rightarrow} A\alpha$ 形式的推导的文法称为**左递归（Left Recursive）文法**。左递归文法不是 LL（1）文法。

以含 $A \rightarrow A\alpha \mid \beta$ 产生式的文法为例，SELECT（$A \rightarrow A\alpha$）和 SELECT（$A \rightarrow \beta$）中均包含 FIRST（β），故：

$$SELECT(A \rightarrow A\alpha) \cap SELECT(A \rightarrow \beta) \neq \Phi$$

因此文法不是 LL（1）的。

左递归文法又分为直接左递归的和间接左递归的。

直接左递归文法的一般形式如下：

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \qquad (4.9)$$

其中 β_i（i=1, 2, ..., n）第一个符号都不是 A，α_j（j=1, 2, ..., m）不等于 ε。

那么可以把（4.9）改写为如下等价形式：

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

$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon \qquad (4.10)$$

（4.10）之所以和（4.9）等价，是因为（4.10）中非终结符号 A 能推出的符号串和（4.9）中非终结符号 A 能推出的符号串是一样的。（4.10）不再包含左递归，虽然 A’的产生式是右递归的，但是右递归并不影响它是否是 LL（1）文法。

【例 4-12】：考虑如下表达式文法，

$$E \rightarrow E + T$$

$$E \rightarrow T$$

$$T \rightarrow T * F$$

$$T \rightarrow F$$

$$F \rightarrow (E)$$

$$F \rightarrow id$$

解：显然此文法是直接左递归文法，按照上述消除左递归的方法，将此文法进行等价变换，得到的文法如下：

$$E \rightarrow TE'$$

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

$$T \rightarrow F\,T'$$

$$T' \rightarrow *F\,T' \mid \varepsilon$$

$$F \rightarrow （E） \mid id$$

上一节已证明该文法是 LL（1）的。

间接左递归文法是含有形如 $A \overset{+}{\Rightarrow} A\alpha$ 推导的文法。

【例 4-13】：考虑如下文法，

$$A \rightarrow aB \mid Bb \qquad (4.11)$$

$$B \rightarrow Ac \mid d$$

解：由于存在推导：A ⇒ Bb ⇒ Acb，所以文法（4.11）是间接左递归的。间接左递归文法也不是 LL（1）的，对于文法（4.11），其关于非终结符号 A 的产生式的 SELECT 集计算如下：

SELECT（A →aB）= { a }

SELECT（A →Bb ）= { a, d }

故：SELECT（A →aB）∩ SELECT（ A →Bb ） ≠ Φ

要消除一个文法中的间接左递归，前提条件是文法不含环路，即无形如 $A \overset{+}{\Rightarrow} A$ 的推导，还要求没有 A → ε 形式的产生式。

算法 4-1：间接左递归消除算法。

输入：没有环或 ε 产生式的文法 G；

输出：一个等价的无左递归文法；

方法：对文法 G 执行以下算法。

1) 按任意顺序排列非终结符号：$A_1, A_2, ..., A_n$

2) for (从 1 到 n 的每个 i)

3) { for (从 1 到 i-1 的每个 j)

4) { 将每个形如的 $A_i \rightarrow A_j \gamma$ 产生式替换为产生式组：

5) $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \cdots \mid \delta_k \gamma$ ，

6) 其中 $A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k$ 是所有关于 A_j 的产生式

7) }

8) 消除产生式的直接左递归；

9) }

10) 对获得的文法化简。

【例 4-14】：消除以下文法的左递归，

S → Ac | c

A → Bb | b

B → Sa| a

解：首先给出一个非终结符号的排列：B，A，S，

（1）先写出 B 的产生式，B → Sa| a，无直接左递归；

（2）写出 A 的产生式，将 B 的产生式代入 A 的产生式的右部，得：

A → Sab | ab | b

这组产生式也无直接左递归；

（3）写出 S 的产生式，将（2）中得到的 A 的产生式带入 S 产生式的右部，得：

S → Sabc | abc | bc | c

这组产生式有直接左递归，消除这组产生式的直接左递归得：

S → abcS’ | bcS’ | cS’

S’ → abcS’ | ε

（4）以上步骤获得的文法如下：

S → abcS’ | bcS’ | cS’

S’ → abcS’ | ε

A → Sab | ab | b

B → Sa| a

最后对此文法化简。文法开始符号是 S，A 和 B 的产生式在推导过程中不会被用到，可删除。化简之后的文法如下：

S → abcS’ | bcS’ | cS’

S’ → abcS’ | ε

对非终结符号的排序不同，最后得到的无左递归的文法可能也是不同的，但它们都是等价的。

例如：给出另一个非终结符号的排列：S，A，B，消除左递归之后得到文法：

S → Ac | c

A → Bb | b

B → bcaB’|caB’|aB’

B’ → bcaB’ | ε

需要注意的是，一个文法没有公共左因子，没有左递归，并不一定能保证文法是 LL（1）的，无左因子产生式和无左递归只是文法为 LL（1）文法的必要条件，而非充分条件。

4.2.4 无回溯递归下降分析法

递归下降分析法为自顶向下语法分析的实现提供了一个框架。一个递归下降语法分析器由一组过程构成，文法中的每个非终结符号 U（它们分别代表一个语法单位）都对应一个过程，该过程完成对 U 所对应的语法单位的分析和识别任务。U 对应的过程的结构和 U 的产生式右部的结构是一致的，对于产生式右部的终结符号调用一个辅助过程直接匹配，对于非终结符号，则调用该非终结符号对应的过程来对它进行分析和识别。由于在分析（构造推导序列）过程中，可能会有同一个非终结符号对应的过程的多个实例同时在运行，即所谓的过程递归调用，另外这种方法是自顶向下分析的具体实现方法，以自顶向下顺序构建分析树，因此这种方法称为**递归下降分析法**。

采用递归下降分析法分析句子时是从调用开始符号 S（对应整个句子）对应的过程开始的。作为自顶向下分析的具体实现，递归下降分析法也要解决一个如何选择产生式的问题。如果是实现不确定的自定向下分析，那么在分析过程中会产生回溯，影响分析效率。如果是实现确定的自顶向下分析，那么可以在 SELECT 集计算的基础上，以无回溯的方式来实现句子的分析。下面结合一个实例来介绍无回溯的递归下降分析法。

【例 4-15】：考虑以下文法，

type → simple

| ↑ id

| array [simple] of type

simple → integer

| char

| num dotdot num

试分析句子： array [num dotdot num] of integer

解：首先计算每条产生式的 SELECT 集：

SELECT(type → simple) = { integer, char, num }

SELECT(type → ↑ id) = { ↑ }

SELECT(type → array [simple] of type) = { array }

SELECT(simple → integer) = { integer }

SELECT(simple → char) = { char }

SELECT(simple → num) = { num }

下面为文法构建无回溯的递归下降分析程序。该文法有两个非终结符号 type 和 simple，分别为它们构建识别过程。

```
procedure type ;
begin
        if lookahead in  { integer, char, num }  then
                simple ( )
            else  if  lookahead = □ '↑'  then
                begin
                    match (□ '↑'  □);
                    match (id);
                end
                    else  if  lookahead =  'array'  then
                        begin
                            match ( 'array' );
                            match (  '  [  ' );
                            simple ( );
                            match (  '  [  '  );
                            match ( 'of'  );
                            type ( )
                        end
        else error( )
end;

procedure simple ;
begin
        if  lookahead =  'integer'  then
```

```
            match (‘integer’)
        else  if  lookahead = ‘char’  then
            match (‘char’)
        else  if  lookahead = ‘num’  then
            begin
                match (‘num’);
                match (dotdot);
                match (‘num’);
            end
        else  error( )
end;
```

以上过程中的 lookahead 是一个全局变量，指向句子中的下一个待匹配的符号，开始时指向句子的第一个符号。以上过程的基本思想是，若 lookahead 指向的符号在当前非终结符号某条候选产生式的 SELECT 集中，则进入这条产生式右部的分析，同时输出这条产生式（用于最左推导）。

另外需要一个辅助过程 match，输入参数为 t，如果 t 与 lookahead 指向的符号匹配，lookahead 指向句子中的下一个输入符号，如果不匹配就报错。

```
procedure match (t : token);
    begin
        if  lookahead = t  then
            lookahead  :=  nexttoken( )
        else  error ( )
    end;
```

调用以上分析程序，对句子“array [num dotdot num] of integer”进行分析，分析过程如下：

（1）调用 type ，输出 type → array [simple] of type；

（2）调用 simple ，输出 simple → num dotdot num；

（3）调用 type ，输出 type → simple；

（4）调用 simple ，输出 simple ➞ integer。

用以上步骤输入的 4 条产生式作最左推导，可以推出句子“array [num dotdot num] of integer”，同时可以自顶向下构建句子的分析树。

构建一个递归下降分析程序，相对来说比较简单，程序结构比较直观，可读性好。如果要对文法进行扩充，实现起来也比较方便。基于以上优点，递归下降分析法在许多高级语言的编译器设计中得到了应用。

4.2.5 非递归预测分析器

确定的自顶向下分析又叫预测分析。预测分析的一个实现方案是上一小节讨论的无回溯递归下降分析。在这种分析方法中，将文法产生式的 SELECT 集的信息直接写在代码里，也就是说代码和数据是集成在一起的。下面介绍预测分析的另一个实现方案——非递归预测分析器模型，见图 4-5。

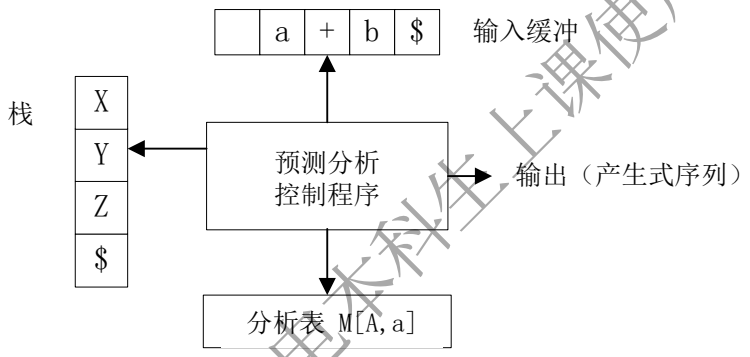

图 4-5 非递归预测分析器模型

该模型包含几个组成部分：

（1）输入缓冲：待分析的句子末尾跟上一个“$”符号，放在输入缓冲区中，在分析过程中有一个指针指向句子中下一个待匹配的符号；

（2）栈：用于存放分析的中间结果，栈底是“$”符号，其它是文法符号；

（3）分析表：是一张二维表，用于存放 SELECT 集的信息，一个分析表的例子见表 4-5；

（4）预测分析控制程序：是一个算法，根据当前非终结符号及句子中下一个要匹配的符号查询分析表，确定下一步用哪条产生式进行推导；

（5）输出：算法输出一个产生式序列，用这个产生式序列作最左推导可以推出待分析的句子。

下面介绍分析表的构造方法。分析表是一张二维表，记为 M [A，a]，A 是非终结符号，a 是终结符号或“$”符号，表中存放的是产生式，如果：

a∈SELECT（A→α），

则把 A→α 放入 M [A，a]中。空白表项表示出错。本质上，分析表是以表格的形式存放了 SELECT 集的信息。

考虑文法（4.4），前面已经计算了每条产生式的 SELECT 集，则：

（1）由 SELECT(E→ TE')= {(, id}，将产生式“E→ TE'”放到“E”这行的“(”这列和“id”这列；

（2）由 SELECT(E'→ +TE')={+}，将产生式“E'→ +TE'”放到“E'”这行的“+”这列；

（3）由 SELECT(E'→ ε)={), $}，将产生式“E'→ ε ”放到“E'”这行的“)”这列和“$”这列；

（4）由 SELECT(T→FT') ={(, id}，将产生式“T→FT'”放到“T”这行的“(”这列和“id”这列；

（5）由 SELECT(T'→*FT')={*}，将产生式“T'→*FT'”放到“T'”这行的“*”这列；

（6）由 SELECT(T'→ ε)={)，+，$}，将产生式“T'→ ε ”放到“T'”这行的“)”这列、“+”这列和“$”这列；

（7）由 SELECT(F→(E))={(}，将产生式“F→(E)”放到“F”这行的“(”这列；

（8）由 SELECT(F→id)={id}，将产生式“F→id”放到“F”这行的“id”这列；

最后构造的分析表见表 4-5。

表 4-5 文法（4.4）的分析表

非终结符号	输入符号					
	id	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→ +TE'			E'→ ε	E'→ ε
T	T→FT'			T→FT'		
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id			F→(E)		

定义 4-6：如果一个文法的预测分析表没有多重定义入口，则该文法是 **LL（1）文法**。

所谓多重定义入口是指在分析表的某个表项中存在多条产生式。定义 4-6 是 LL（1）文法的另一个定义。定义 4-4 从 SELECT 集的角度来定义了 LL（1）文法，定义 4-6 是通过考

察分析表来定义 LL（1）文法的。由于分析表是 SELECT 集的一种存储方式，这两个定义本质上是一样的。

下面讨论无递归预测分析器是如何分析句子的，即预测分析控制程序的工作过程。

令当前栈顶为 X、当前输入符为 a，由(X，a)决定分析动作，共 3 种可能：

（1）若 X=a≠$，则 X 从栈中弹出，输入指针下移，指向下一个符号；

（2）若 $X \in V_N$，则去查分析表 M 的元素 M[X，a]，该元素或为 X 的产生式，或为一个“空白”（出错）。

a）如果 M[X, a]=X→$Y_1Y_2\cdots Y_k$，则把 X 从栈中弹出，并依次将 Y_k, Y_{k-1},…,Y_1 压入栈中；

b）如果 M[X, a]=“空白”（出错），调用出错处理程序。

（3）若 X=a=$，分析停止，宣告成功完成分析。

基于非递归预测分析器模型，下面给出具体的预测语法分析算法。

算法 4-2：预测语法分析算法。

输入：一个串 w，文法 G 的预测分析表 M。

输出：如果 w 在 L（G）中，输出 w 的一个最左推导，否则给出一个错误指示。

方法：预测分析器初始设置：将 w$放入输入缓冲区中，将“$”符号压栈，将 G 的开始符号 S 压栈，然后执行以下算法。

```
置 ip 指向 w$ 的第一个符号;
repeat
        令 X 是栈顶符号, a 是 ip 所指向的符号;
        if X 是终结符号或 $ then
              if  X=a  then
                    把 X 从栈中弹出,  ip 指向下一符号;
              else  error()
        else     /*  X 是非终结符号  */
              if  M[X,a] = X→Y1Y2…Yk  then begin
                    把 X 从栈中弹出;
                    依次把 Yk, Yk-1, … , Y1  压入栈中, 即 Y1 在顶上;
                    输出产生式  X→Y1Y2…Yk
                    end
              else  error()
until X=$   /* 栈为空 */
```

【例 4-16】：用预测分析器分析文法（4.4）的句子“id+id*id”，分析表见表 4-5。

解：分析过程见表 4-6。

表 4-6 句子 id+id*id 的预测分析

步骤	已匹配	栈	输入串	动作
1		E$	id+id*id$	
2		T E’$	id+id*id$	输出 E→TE’
3		F T’ E’$	id+id*id$	输出 T→F T’
4		id T’ E’$	id+id*id$	输出 F→id
5	id	T’ E’$	+id*id$	匹配 id
6	id	E’$	+id*id$	输出 T’ → ε
7	id	+ T E’$	+id*id$	输出 E’ →+TE’
8	id+	T E’$	id*id$	匹配+
9	id+	F T’ E’$	id*id$	输出 T→F T’
10	id+	id T’ E’$	id*id$	输出 F→id
11	id+id	T’ E’$	*id$	匹配 id

12	id+id	* F T' E'$	*id$	输出 T' →*F T'
13	id+id*	F T' E'$	id$	匹配*
14	id+id*	id T' E'$	id$	输出 F→id
15	id+id*id	T' E'$	$	匹配 id
16	id+id*id	E'$	$	输出 T' → ε
17	id+id*id	$	$	输出 E' → ε

表 4-6 内容分为 4 列，“已匹配”列显示的是句子中已扫描过的部分，“输入串”列显示的是剩余的待扫描的部分，每一行的这两列并起来构成正在被分析的句子。“栈”这列显示的是在分析过程中获得的中间结果，每一行的“已匹配”列并上“栈”这列构成推导得到的句型。“动作”列有两个动作，一个是输出产生式（用于最左推导），一个是当推导出终结符号时与句子中的终结符号进行匹配。分析过程输出一个产生式的序列，用这个产生式序列作最左推导可以推出句子，同时可以按自顶向下、从左到右的顺序构建分析树，见图 4-4。

4.2.5 预测分析中的错误处理

错误处理是任何一个语法分析方法必须考虑的问题。在分析过程中一旦发现源程序存在语法错误，首先要报告错误出现的位置，然后要判断该错误的性质与类别，并给出提示信息。同时要有错误处理机制，使得分析可以继续，以便在一次扫描中尽可能多地发现源程序中的错误。一种有效的错误处理机制是所谓的恐慌模式。恐慌模式错误处理的基本思想是：发现错误后忽略掉输入串中的一些符号，直到下一个输入符号是事先确定的一个同步单词集合中的元素，分析过程再恢复进行。

对于预测分析器模型，在两种情况下会发现错误（见算法 4-2）：

（1）栈顶的终结符号和句子中的下一个输入符号不匹配；

（2）根据栈顶符号 X 和句子中下一个符号 a，去查分析表，发现是空白。

针对这两种情况，结合恐慌模式错误处理的基本思想，一个启发式的错误处理机制如下：

（1）若栈顶终结符号和下一个输入符号不匹配，则弹出栈顶终结符号；

（2）在分析表中加入同步化入口（synch），如果查询分析表时是“synch”，则从栈中弹出 A；

（3）如果查询分析表时是“空白”，则跳过输入符号 a。

根据非终结符号的 FOLLOW 集可以在分析表中添加同步化入口（synch）。计算每个非终结符号 A 的 FOLLOW 集，令 x∈FOLLOW（A），若在分析表中 M[A, x]是空白，则在这个表项中添加“synch”。

【例 4-17】：为文法（4.4）构建含同步化入口的分析表，并分析一个输入串“+id*+id”。

解：

（1）由 FOLLOW(E)={), $ }，在 M[E,)]和 M[E, $]添加“synch”；

（2）由 FOLLOW(T)={), +, $ }，在 M[T,)]、M[T, +]和 M[T, $]添加“synch”；

（3）由 FOLLOW(F)={ +, *,), $ }，在 M[F, +]、M[F, *], M[F,)]和 M[F, $]添加“synch”。

含同步化入口的分析表见表 4-7。

表 4-7 文法（4.4）含同步化入口的分析表

非终结符号	输入符号					
	id	+	*	(	)	$
E	E→TE'			E→TE'	**synch**	**synch**
E'		E'→ +TE'			E'→ ε	E'→ ε
T	T→FT'	**synch**		T→FT'	**synch**	**synch**
T'		T'→ ε	T'→*FT'		T'→ ε	T'→ ε
F	F→id	**synch**	**synch**	F→(E)	**synch**	**synch**

句子“+id*+id”有两个错误，一个是表达式不能以运算符“+”开头，一个是两个二目运算符“*”、“+”之间必须要有运算对象。句子分析过程见表 4-8，分析过程中发现了这两个错误，确定了具体位置，并作了处理以保证分析过程能继续下去。

表 4-8 预测分析错误处理示例

步骤	栈	输入串	动作
1	E$	+id*+id$	错误，忽略+，继续分析
2	E$	id*+id$	
3	T E’$	id*+id$	
4	F T’ E’$	id*+id$	
5	id T’ E’$	id*+id$	
6	T’ E’$	*+id$	
7	* F T’ E’$	*+id$	
8	F T’ E’$	+id$	错误，弹出 F，继续分析
9	T’ E’$	+id$	
10	E’$	+id$	
11	+ T E’$	+id$	
12	T E’$	id$	

13	F T' E'$	id$	
14	id T' E'$	id$	
15	T' E'$	$	
16	E'$	$	
17	$	$	

4.3 自底向上语法分析——LR 分析

4.3.1 自底向上语法分析的关键——识别句柄

如 4.1 节所述，一个自底向上的语法分析过程是从左到右扫描待分析的句子，构造一个归约序列（一般要求是最左归约序列），将句子最终归约为文法的开始符号。这个（最左）归约序列的逆序是一个（最右）推导序列。最右推导序列对应的是这个句子的分析树的一个自顶向下、从右到左的分析树的构造过程。这个构造过程的逆序是自底向上、从左到右构造分析树，和最左归约的顺序是一致的。参见【例 4-2】。

归约是用某条产生式的左部非终结符号替换其右部文法符号串的过程。在自底向上分析过程中，关键是要解决如何确定可归约串的问题，如果是做最左归约就是要解决一个如何确定最左可归约串的问题。

定义 4-7：从文法的开始符号出发进行零步或多于零步的最右推导得到的文法符号串称为**右句型**，右句型又称为**规范句型**。

令文法 G 的开始符号是 S，给出以下最右推导序列：

$$S = \gamma_0 \underset{rm}{\Rightarrow} \gamma_1 \underset{rm}{\Rightarrow} \gamma_2 \underset{rm}{\Rightarrow} \cdots \underset{rm}{\Rightarrow} \gamma_{n-1} \underset{rm}{\Rightarrow} \gamma_n = w \qquad (4.12)$$

S 做零步最右推导，得到 γ_0，因此 γ_0 是右句型，做一步最右推导，得到 γ_1，γ_1 是右句型。以此类推，做 n 步最右推导得到 γ_n，γ_n 只包含终结符号，不能够继续推导，就是我们要分析的句子 w。反过来看（4.12）就是一个最左归约的过程，每一步归约都是将右句型 γ_i 归约为右句型 γ_{i-1}。自底向上的语法分析可以看作是一个从句子出发，不断对右句型（句子也是右句型）中的最左可归约串进行归约的过程。

考虑（4.12）中的某步推导（$S \overset{*}{\underset{rm}{\Rightarrow}}$）αAx $\underset{rm}{\Rightarrow}$ αβx。由于是最右推导，x 不含非终结符号，这步推导用到的产生式是 A→β，用产生式左部 A 去替换右部 β。对应分析树的构造过程，这步推导之后分析树结构如图 4-6 所示。

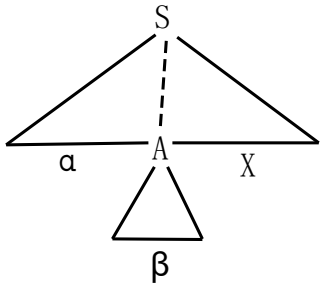

图 4-6 句柄的识别

从图 4-6 可知，A 及其子结点 β 构成的子树是分析树中最左边的只有父子两代的子树。根据定义 2-18，β 是右句型 αβx 的句柄。于是将右句型 αβx 归约为右句型 αAx，是对句柄 β 进行归约，αβx 的最左可归约串就是句柄 β。

需要注意的是文法可能是二义性的，αβx 可能存在多个最右推导，就可能存在多个句柄。如果一个文法是无二义性的，那么文法的每个右句型都有且只有一个句柄。

句柄首先必须和某个产生式右部匹配，但是和某个产生式右部匹配的未必是句柄。如【例 4-2】中的归约序列：

$$id*id \Leftarrow F*id \Leftarrow T*id \Leftarrow T*F \Leftarrow T \Leftarrow E \qquad (4.3)$$

在 3 步归约之后得到 T*F，T 是产生式 E→T 的右部，T*F 是产生式 T→T*F 的右部。这时候有两个选择，一个是将 T 归约为 E，另一个是将 T*F 归约为 T。显然第一个选择是错误的，如果将 T 归约为 E，则后续无法将句子归约到开始符号。也就是说对于右句型 T*F 来说，T*F 是它的句柄，而 T 不是。

因此自底向上语法分析的关键问题是如何识别右句型的句柄。

4.3.2 自底向上语法分析的实现方法——移进-归约法

在具体实现自底向上分析的时候往往会借助一个符号栈。在从左到右对待分析的句子进行扫描的过程中，将输入符号逐个移进一个先进后出的符号栈中。一边移进一边分析栈顶，一旦栈顶出现了当前右句型的句柄就选择一条产生式进行归约。重复这一过程，直到归约到栈中只剩下文法的开始符号，同时输入符号全部移入了栈中时分析成功。

分析过程中通常使用$来标记栈的底部，在输入串的末尾也加上一个$符号作为输入串的结束标志。开始的时候栈中除了一个$符号之外没有其它元素，输入串 w 后跟一个$符号存放在输入缓冲区中，见如下格局。

符号栈	输入串
$	*w* $

首先将 w 的第一个符号移进栈中，同时分析栈顶有没有出现 w 的句柄，如果有就选择一条产生式进行归约，否则就继续移进第二个符号并判断，直到栈顶出现 w 的句柄并进行归约。反复执行这样的移进-归约操作，如果 w 是一个正确的句子，分析过程中栈中的文法符号串拼接上剩余的输入串将构成一个右句型，一旦这个右句型的句柄出现在了栈顶，就选择一条产生式进行归约。如果句子是正确的，分析将在如下格局出现后结束。

符号栈	输入串
$S	$

如果句子有错误，分析将在发现错误的时候报错。虽然语法分析动作主要是移进和归约，但实际上一个移进-归约语法分析器可以有四个动作：

（1）移进（shift）：将下一个输入符号移进到栈顶；

（2）归约（reduce）：应用一条产生式对出现在栈顶的句柄进行归约（句柄从栈中弹出，产生式左部压入栈中）；

（3）接受（accept）：宣布语法分析过程成功完成；

（4）报错（error）：发现一个语法错误，并调用一个错误处理子程序。

【例 4-17】：考虑以下文法：

$$\begin{aligned}&(1)\ S \rightarrow aAcBe\\&(2)\ A \rightarrow b\\&(3)\ A \rightarrow Ab\\&(4)\ B \rightarrow d\end{aligned} \qquad (4.12)$$

试用移进-归约法分析句子：abbcde。

解：分析过程见表 4-9。

表 4-9 对句子 abbcde 的移进-归约分析

步骤	符号栈	输入串	动作
1	$	abbcde$	移进
2	$a	bbcde$	移进
3	$ab	bcde$	归约（A→b）
4	$aA	bcde$	移进
5	$aAb	cde$	归约（A→Ab）
6	$aA	cde$	移进
7	$aAc	de$	移进
8	$aAcd	e$	归约（B→d）
9	$aAcB	e$	移进
10	$aAcBe	e$	归约（S→aAcBe）
11	$S	$	接受

在分析过程中，依次使用了四条产生式进行归约。每次都是对右句型进行归约，归约的结果也是右句型。每一时刻栈中的文法符号串拼接上剩余的输入串构成当前右句型，每次都是当当前右句型的句柄出现在栈顶时立刻进行归约。

需要注意的是移进-归约法只是自底向上分析具体实现的一个框架，并没有解决句柄如何识别这一自底向上分析的关键问题，反映到分析过程中就是并没有解决何时移进，何时归约，如果要归约用哪条产生式进行归约的问题。

4.3.3 LR 分析器模型

LR 分析器模型是一个采用移进-归约法的自底向上语法分析器模型，它通过查询预先构造的 LR 分析表来决定在分析过程中下一步是作移进还是作归约，归约的话又选择哪条产生式进行归约。

LR 分析器模型的一般结构见图 4-7。

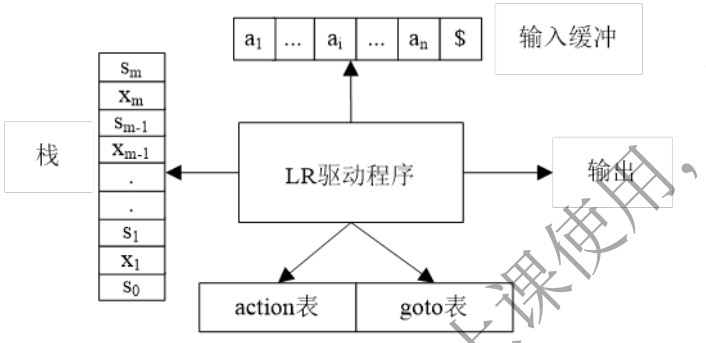

图 4-7 LR 分析器模型

LR 分析器模型由一个输入、一个输出、一个栈、一个驱动程序、一个 LR 分析表（action 子表、goto 子表）构成。待分析的句子后跟$符号放在输入缓冲区里面，分析的时候从输入缓冲区逐个读入符号。栈中包含两类符号，x_i 是文法符号，可以是终结符号或非终结符号，s_i 是状态符号，栈底是 s_0，代表初始状态，栈顶是 s_m，代表当前状态，文法符号和状态符号交替出现。所有 LR 语法分析器的驱动程序都是一样的，驱动程序是 LR 分析算法的一个实现，而分析表是和文法相关的，文法不同分析表也不同。LR 分析器输出的是一个产生式的序列，如果句子是正确的，可以用该产生式序列对句子进行最左归约，将句子归约为文法的开始符号，实现句子的自底向上分析。

LR 分析表由两个子表组成：一个动作表 action、一个转换表 goto。LR 分析表的行表头是状态，action 表的列表头是终结符号或$符号，goto 表的列表头是非终结符号。

令 i 是状态，a 是终结符号或者$符号，则 action[i, a]的取值可以有下列四种形式：

（1）移进（s_j），其中 j 是一个状态，分析器的下一步动作将是把输入符号 a 移进栈中，接着把 j 也移进栈中。

（2）归约（r_k），其中 k 是产生式的编号，假设文法第 k 条产生式是 A→β，则分析器下一步动作是把栈顶的 β（句柄）归约为产生式的左部 A。

（3）接受（Acc），分析器宣告成功完成对句子的分析。

（4）报错（空白），分析器发现了输入串中的错误。

令 A 是非终结符号，若 goto[i, A]=j，表示状态 i 面临 A 时转换到状态 j。

表 4-10 是文法（4.12）的 LR（0）分析表，如何构建分析表后续再介绍。

表 4-10 LR 分析表示例

	action						goto		
	a	b	c	d	e	$	S	A	B
0	s_2						1		
1						Acc			
2		s_4						3	
3		s_6	s_5						
4	r_2	r_2	r_2	r_2	r_2	r_2			
5				s_8					7
6	r_3	r_3	r_3	r_3	r_3	r_3			
7					s_9				
8	r_4	r_4	r_4	r_4	r_4	r_4			
9	r_1	r_1	r_1	r_1	r_1	r_1			

LR 分析器的工作过程可以看作是栈中的符号序列和输入缓冲区中剩余符号串所构成的二元式的变化过程。这个二元式称为格局（configuration）。初始格局为：

（s_0, $a_1a_2\cdots a_n$$）

其中 s_0 为初始状态，$a_1a_2\cdots a_n$ 是待分析的符号串。

假设分析过程中当前格局为：

（$s_0x_1s_1x_2s_2\cdots x_ms_m$, $a_ia_{i+1}\cdots a_n$$）

其中栈中的文法符号串拼接上剩余输入串（$x_1x_2\cdots x_ma_ia_{i+1}\cdots a_n$）构成当前正在分析的右句型，当该右句型的句柄出现在栈顶，就进行归约，否则就移进。在这个格局下，下一步的动作由栈顶状态 s_m 和下一个输入符号 a_i 决定：

（1）若 action [s_m, a_i] = s_j，则移进，即将 a_i 和 j 依次压入栈中，格局变为：

（$s_0x_1s_1x_2s_2\cdots x_ms_ma_ij$, $a_{i+1}\cdots a_n$$）

（2）若 action [s_m, a_i] = r_k，则归约。令文法第 k 条产生式为 A→β，假设 β 的长度为 r，则 $\beta=x_{m-r+1}x_{m-r+2}\ldots x_m$。查询转换表，若 goto[$s_{m-r}$, A]=s，则格局变为：

（$s_0x_1s_1x_2s_2\cdots x_{m-r}\,s_{m-r}\,As$, $a_ia_{i+1}\cdots a_n$$）

（3）若 action [s_m, a_i] = Acc，则宣告分析成功。

（4）action [s_m, a_i] = 空白（error），则报告错误。

上面的讨论可以归纳为如下的 LR 分析算法，LR 驱动程序即是对 LR 分析算法的实现。

算法 4-3：LR 分析算法。

输入：一个输入串 w 和文法 G 的 LR 分析表。

输出：若 w 是 L（G）中的句子，则输出对 w 的自底向上分析过程，否则给出一个错误提示。

方法：将初始状态 s_0 置于栈底，将 w$置于输入缓冲区中，然后执行以下程序。

```
置 ip 指向 w$的第一个符号;
repeat forever begin
    令 s 是栈顶状态符号, a 是 ip 指向的当前符号;
    if action[s,a] = shift s' then
        begin
            把 a 和 s' 依次压入栈中;
            是 ip 指向下一个符号
        end
    else if action[s,a] = reduce A→β then
        begin
            从栈中弹出 2 * |β| 个符号;
            令 s'是当前的栈顶状态;
            把 A 和 goto[s', A]依次压入栈中;
            输出产生式 A→β
        end
    else if action[s,a] = Acc then
            return
        else error()
end
```

图 4-8 LR 分析程序

基于表 4-9 调用 LR 分析算法对句子 abbcde 进行分析，分析过程如表 4-11 所示。

表 4-11 基于 LR 分析表对句子 abbcde 进行分析

步骤	栈	输入串	action	goto
1	0	abbcde$	s_2	
2	0a2	bbcde$	s_4	
3	0a2b4	bcde$	r_2	3
4	0a2A3	bcde$	s_6	
5	0a2A3b6	cde$	r_3	3
6	0a2A3	cde$	s_5	
7	0a2A3c5	de$	s_8	
8	0a2A3c5d8	e$	r_4	7
9	0a2A3c5B7	e$	s_9	
10	0a2A3c5B7e9	e$	r_1	1
11	0S1	$	Acc	

基于 LR 分析器模型的语法分析又称为 LR（k）分析，其中 L 表示对输入串进行从左到右的扫描，R 表示构造一个最右推导的反向过程（即最左归约过程），k 是在做出分析决定时向前看的输入符号的个数。K 通常为 0 或 1，省略 k 时，默认 k=1。LR（k）分析是当前最广义的“移进-归约”方法，现今的能用上下文无关文法描述的程序设计语言一般都可以用 LR 分析方法进行有效的分析。相对于 LL（1）分析来说它适用的文法更广，而分析效率却不逊色。

LR（k）分析的关键问题是如何构造分析表，不同的构造方法形成不同的 LR 分析方法，主要有 LR（0）、SLR（1）、LR（1）、LALR（1）等。

4.3.4 构造 LR（0）分析表

1、LR 分析过程中的性质与特点

总结一下，在调用 LR 分析器模型分析句子过程中，忽略掉栈中的状态符号，剩下的文法符号串并上输入缓冲区中剩余的输入串构成一个右句型，当该右句型的句柄出现在栈顶时归约，否则就移进。栈中的文法符号串是当前右句型的前缀，这个前缀有一个重要的性质，那就是它不可能包含当前右句型的句柄右边的符号，因为一旦在栈顶出现了完整的句柄，就进行归约，归约之后当前右句型就转换成了另一个右句型。栈中的这个不含右句型句柄右边符号的前缀称为**活前缀**。

【例 4-18】：考虑文法（4.12），作最右推导：

$$S \underset{rm}{\Rightarrow} \underline{aAcBe} \underset{rm}{\Rightarrow} aAc\underline{d}e \underset{rm}{\Rightarrow} a\underline{Ab}cde$$

每步最右推导推出的都是右句型，考虑右句型 aAbcde，其中 Ab 是其句柄，这个右句型的不含句柄后面符号的活前缀有 4 个： ε , a, aA, aAb，其中 aAb 包含了完整的句柄。

包含完整句柄的活前缀称为**可归前缀**。将活前缀作为考察对象，当栈中的活前缀是可归前缀时，下一步动作是归约，否则移进。如果有识别活前缀（特别是可归前缀）的方法，将可以帮助我们决定 LR 分析中的下一步动作。

2、构造识别活前缀的 DFA

对于某文法 G，其能推导出哪些右句型是确定的。一个右句型又对应若干个活前缀，因而每个文法 G 都对应了一个特定的活前缀的集合。可以证明：任一文法 G 的右句型的所有活前缀构成一个语言，这个语言是正规语言，可以用确定的有限自动机（DFA）来识别。证明这个定理已超出本书范围，在这里不作介绍。

在 LR 分析中一般会对文法 G 进行一个拓广，令 G 的开始符号是 S，那么 G 的拓广文法 G'就是为 G 引入一个新的开始符号 S'，同时增加产生式 S'→S 而得到的文法。拓广文法具有以下特点：

（1）拓广文法与原文法等价。对于同一个句子，基于拓广文法作自顶向下分析，第一步是用 S'→S 推导，其它步骤和原来一样；作自底向上分析，最后一步是用 S'→S 归约，其它步骤和原来一样；

（2）文法的开始符号（S'）只在第一条产生式的左部出现，不会出现在其它位置。这样可以区别在分析过程中是归约到了文法的最初开始符号还是产生式右边出现的开始符号；

（3）在分析过程中，如果用 S'→S 进行归约，则分析结束，输入串被分析器接受。

【例 4-19】：文法（4.12）的拓广文法如下：

(0) S' → S
(1) S → aAcBe
(2) A → b　　（4.13）
(3) A → Ab
(4) B → d

识别该文法所有活前缀的 DFA 见图 4-9。

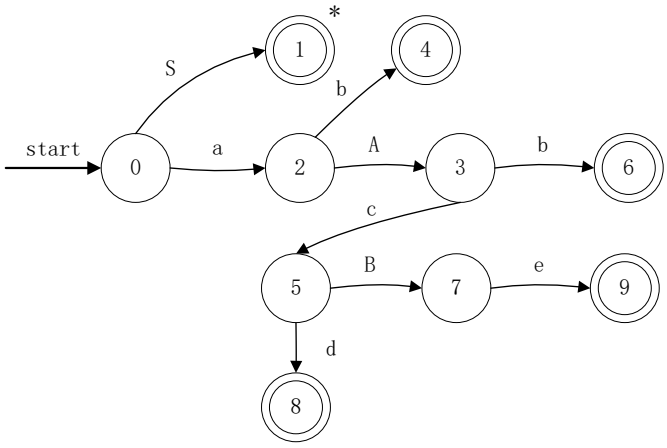

图 4-9 识别文法（4.13）所有活前缀的 DFA

在这个 DFA 中每个状态都是活前缀的识别态，双圈状态是可归前缀（句柄）识别态，标识了“*”的双圈状态是句子识别态。例如：识别 ε，a，aA，aAb 这几个活前缀的分别是状态 0、2、3、6。其中状态 6 识别可归前缀 aAb。

下面讨论如何构建识别文法所有活前缀的 DFA。

首先介绍 LR（0）项目。

定义 4-8：在文法产生式右部的适当位置添加一个圆点“·”构成 **LR（0）项目**（item）。

【例 4-20】：产生式 A→XYZ 的 LR（0）项目包括：

A→·XYZ；A→X·YZ；A→XY·Z；A→XYZ·

特殊地，对于空产生式 A→ε，它有唯一的一个项目：A→·。

项目的含义：圆点的左部表示分析过程中的某时刻用该产生式归约时句柄已识别过的部分，圆点右部表示待识别部分。

项目又分为 4 类：

（1）移进项目：形如 A →α· aβ，即圆点“·”后面是一个终结符号的项目。

（2）待约项目：形如 A →α· Bβ，即圆点“·”后面是一个非终结符号的项目。

（3）归约项目：形如 A →α·，即圆点“·”位于产生式右部最后面的项目。

（4）接受项目：形如 S’ → S·，是一个特殊的归约项目，和文法开始符号 S’的唯一产生式有关。

定义 4-9：令 I 是一个项目集合，按如下步骤构建的项目集合称为 I 的项目集闭包（closure），记为 closure（I）：

（1）I 中的每个项目在 closure（I）中；

（2）若 A→ α ・B β 在 closure（I）中，且 B→ γ 是产生式，则将 B→ ・ γ 添加到 closure（I）中；

（3）反复执行（2），直到没有新的项目添加到 closure（I）中为止。

【例 4-21】：考虑以下拓广的表达式文法：

(0) E’ → E

(1) E → E + T

(2) E → T

(3) T → T * F （4.14）

(4) T → F

(5) F → (E)

(6) F → id

试求 closure（{ E’ → ・E }）。

解：closure（{ E’ → ・E }）= { E’ → ・E

E→ ・E+T

E→ ・T

T→ ・T*F

T→ ・F

F→ ・(E)

F→ ・id }

定义 4-10：令 I 是一个项目集合，X 是一个文法符号，**转换函数 go（I, X）**定义为：

go（I, X）= closure（J）

其中 J 按如下规则构建：若 I 中存在形如 A→ α ・X β 的项目，则把项目 A→ α X・ β 添加到 J 中。

【例 4-22】：考虑文法（4.14），求： go（{ E’ → E・， E → E・+ T }, +）。

解：go（{ E’ → E・， E → E・+ T }, +）

= closure（{ E → E+・T }）

= { E → E+・T

T→・T*F

T→・F

F→・(E)

F→・id }

以项目 S’ →・S 为源头，使用闭包运算 closure 和转换函数 go，可以构建拓广文法的**规范的 LR（0）项目集族**。见算法 4-4。

算法 4-4：规范的 LR（0）项目集族构建算法。

令 G’为拓广文法，开始符号为 S’，其唯一产生式为 S’ →・S，执行以下程序求规范的 LR（0）项目集族 C：

```
procedure items(G');
        begin
          C := { closure ( { S' →・S } ) } ;
          repeat
              for 每一 I∈C 和每一 X ∈ { VT∪VN }
                  把 go(I, X)加入到 C 中
              until  C 不再增大
        end;
```

【例 4-23】：试求文法（4.14）的规范的 LR（0）项目集族。

解：

I_0 = closure ({ E’ →・E })

= { E’ →・E；E→・E+T；E→・T；T→・T*F；T→・F；F→・(E)；F→・id }

I_1 = go (I_0, E)

= closure ({ E’ →E・；E→E・+T })

= { E’ →E・；E→E・+T }

I_2 = go (I_0, T)

= closure ({ E→T・； T→T・*F })

= { E→T・； T→T・*F }

I_3 = go (I_0, F)

= closure ({ T→F・})

= { T→F・ }

I_4 = go (I_0, ( )

= closure ({ F→(・ E) })

= { F→(・ E)； E→・E+T； E→・T； T→・T*F； T→・F； F→・(E)； F→・id }

I_5 = go (I_0, id)

= closure ({ F→id・ })

= { F→id・ }

I_6 = go (I_1, +)

= closure ({ E→E+・T })

= { E→E+・T； T→・T*F； T→・F； F→・(E)； F→・id }

I_7 = go (I_2, *)

= closure ({ T→T*・F })

= { T→T*・F； F→・(E)； F→・id }

I_8 = go (I_4, E)

= closure ({ F→(E・)； E→E・+T })

= { F→(E・)； E→E・+T }

I_9 = go (I_6, T)

= closure ({ E→E+T・； T→T・*F })

= { E→E+T・； T→T・*F }

I_{10} = go (I_7, F)

= closure ({ T→T*F・ })

= { T→T*F · }

I_{11} = go (I_8,))

= closure ({ F→(E) · })

= { F→(E) · }

可以在规范的 LR（0）项目集族及项目集之间 go 函数映射关系的基础上构建识别拓广文法 G'所有活前缀的 DFA：

（1）以规范的 LR（0）项目集族作为 DFA 的状态集，I_0 作为初始状态；

（2）go (I, X) 作为 DFA 的状态转换函数；

（3）含有归约项目的项目集是可归前缀识别态；

（4）含有项目“S' → S•”的是“句子”识别态。

识别文法（4.14）所有活前缀的 DFA 见图 4-10。

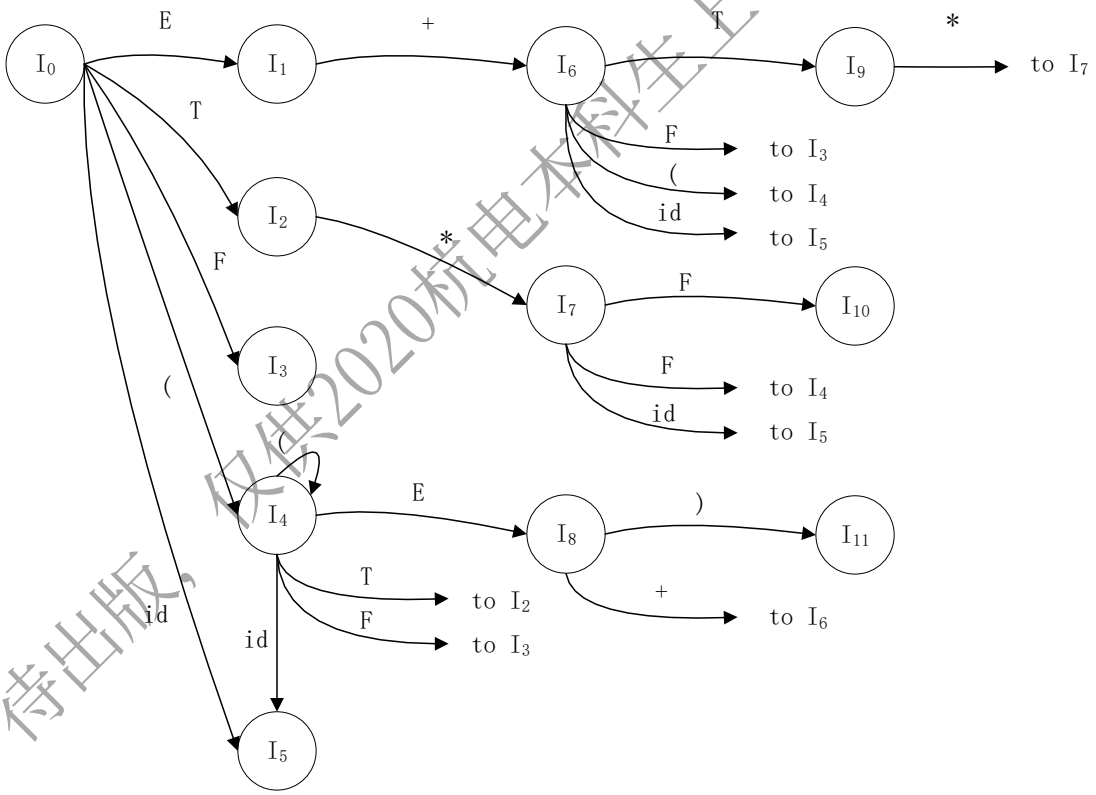

图 4-10 识别文法（4.14）所有活前缀的 DFA

在这个 DFA 中，每个状态都是活前缀的识别态，识别的是从初态 I_0 到该状态的通路上标记的文法符号构成的活前缀。

定义 4-11：令 γ 是活前缀，从初态出发经过 γ 路径所到达的那个状态（即 γ 的识别态）所代表的项目集称为活前缀 γ 的**有效项目集**。

特殊地，初态是活前缀 ε 的有效项目集。

在 LR 分析过程中，栈中文法符号串是当前右句型的活前缀，可以根据当前活前缀的有效项目集确定下一步的分析动作。有以下 3 种情况：

（1）有效项目集中包含移进项目 A → α · aβ，且句子中下一个符号是 a，则移进；

（2）有效项目集中包含归约项目 A → α ·，则用产生式 A → α 归约；

（3）有效项目集中包含接受项目 S' → S ·，则用产生式 S' → S 归约，同时宣告分析成功完成。

有了识别所有活前缀的 DFA，就可以分析句子了。

【例 4-24】：基于文法（4.14）的规范的 LR（0）项目集族和如图 4-10 所示的 DFA 分析句子 id*id。

分析过程见表 4-12。

表 4-12 基于识别活前缀的 DFA 对句子 id*id 进行分析

步骤	栈	输入串	活前缀（识别态）	动作
1	0	id*id $	ε（I_0）	移进
2	0id5	*id $	id（I_5）	归约（F→id）
3	0F3	*id $	F（I_3）	归约（T→F）
4	0T2	*id $	T（I_2）	移进
5	0T2*7	id $	T*（I_7）	移进
6	0T2*7id5	$	T*id（I_5）	归约（F→id）
7	0T2*7F10	$	T*F（I_{10}）	归约（T→T*F）
8	0T2	$	T（I_2）	归约（E→T）
9	0E1	$	E（I_1）	接受

步骤 1：分析开始前栈中没有文法符号，或者说文法符号串为 ε， ε 是任何一个右句型的活前缀，识别 ε 的是状态 I_0，故将状态编号 0 压入栈中。当前句子中下一个符号是“id”，I_0 中包含移进项目“F→ · id”，因此下一个分析动作就是移进。移进“id”之后，栈中活前缀为“id”，识别它的是 I_5，故将 5 压入栈中。反映到 DFA 中就是当前状态从 I_0 沿着标记为“id”的有向边到达 I_5。

步骤 2：当前栈顶状态是 I_5，I_5 中包含归约项目“F→id ·”，于是用产生式 F→id 归约。从栈中弹出句柄“id”，同时弹出“id”对应的状态编号 5，然后将产生式的左部 F 压入栈

中。此时栈中活前缀为“F”，识别该活前缀的是 I_3，故将 3 压入栈中。反映到 DFA 中的就是从状态 I_5，沿着标记为“id”的有向边退回到 I_0，然后沿着标记为“F”的有向边到达 I_3。

步骤 3：当前栈顶状态是 I_3，I_3 中包含归约项目“T→F・”，于是用产生式 T→F 归约。从栈中弹出句柄“F”，同时弹出“F”对应的状态编号 3，然后将产生式的左部 T 压入栈中。此时栈中活前缀为“T”，识别该活前缀的是 I_2，故将 2 压入栈中。反映到 DFA 中的就是从状态 I_3，沿着标记为“F”的有向边退回到 I_0，然后沿着标记为“T”的有向边到达 I_2。

步骤 4：当前栈顶状态是 I_2，I_2 中包含移进项目“T→T・*F”，当前句子中下一个符号是“*”，因此下一个分析动作就是移进。移进“*”之后，栈中活前缀为“T*”，识别它的是 I_7，故将 7 压入栈中。反映到 DFA 中就是当前状态从 I_2 沿着标记为“*”的有向边到达 I_7。

步骤 5：当前栈顶状态是 I_7，I_7 中包含移进项目“F→・id”，当前句子中下一个符号是“id”，因此下一个分析动作就是移进。移进“id”之后，栈中活前缀为“T*id”，识别它的是 I_5，故将 5 压入栈中。反映到 DFA 中就是当前状态从 I_7 沿着标记为“id”的有向边到达 I_5。

步骤 6：当前栈顶状态是 I_5，I_5 中包含归约项目“F→id・”，于是用产生式 F→id 归约。从栈中弹出句柄“id”，同时弹出“id”对应的状态编号 5，然后将产生式的左部 F 压入栈中。此时栈中活前缀为“T*F”，识别该活前缀的是 I_{10}，故将 10 压入栈中。反映到 DFA 中的就是从状态 I_5，沿着标记为“id”的有向边退回到 I_7，然后沿着标记为“F”的有向边到达 I_{10}。

步骤 7：当前栈顶状态是 I_{10}，I_{10} 中包含归约项目“ T→T*F・”，于是用产生式 T→T*F 归约。从栈中弹出句柄“T*F”，同时弹出“T*F”对应的状态编号 2、7、10，然后将产生式的左部 T 压入栈中。此时栈中活前缀为“T”，识别该活前缀的是 I_2，故将 2 压入栈中。反映到 DFA 中的就是从状态 I_{10}，依次沿着标记为“F”、“*”、“T”的有向边退回到 I_0，然后沿着标记为“T”的有向边到达 I_2。

步骤 8：当前栈顶状态是 I_2，I_2 中包含归约项目“E→T・”，于是用产生式 E→T 归约。从栈中弹出句柄“T”，同时弹出“T”对应的状态编号 2，然后将产生式的左部 E 压入栈中。此时栈中活前缀为“E”，识别该活前缀的是 I_1，故将 1 压入栈中。反映到 DFA 中的就是从状态 I_2，沿着标记为“T”的有向边退回到 I_0，然后沿着标记为“E”的有向边到达 I_1。

步骤 9：当前栈顶状态是 I_1，I_1 中包含接受项目“E’ →E・”，于是接受输入串，分析成功。

3、构造 LR（0）分析表

基于识别所有活前缀的 DFA 来分析句子在实现的时候不是很方便，可以在 DFA 的基础上构建 LR（0）分析表，然后调用 LR 分析器模型来分析句子。

假设已构建识别拓广文法 G'所有活前缀的 DFA，按以下规则可构造 G'的 LR（0）分析表：

（1）若 go (I_k，a)= I_j，则 action[k，a]= s_j；

（2）若 go (I_k，A)= I_j，则 goto[k，A]= j；

（3）若 I_k 包含 A→ α ·，则 aciton[k，a]= r_j，a 为任何终结符号或$，j 为产生式 A→ α 的编号；

（4）若 I_k 包含 S' → S·，则 action [k，$] = Acc。

【例 4-25】：构造以下拓广文法的 LR（0）分析表。

(0) S' → S

(1) S → aA

(2) S → bB

(3) A → cA （4.15）

(4) A → d

(5) B → cB

(6) B → d

解：首先构造文法（4.15）规范的 LR（0）项目集族及识别其所有活前缀的 DFA，见图 4-11。

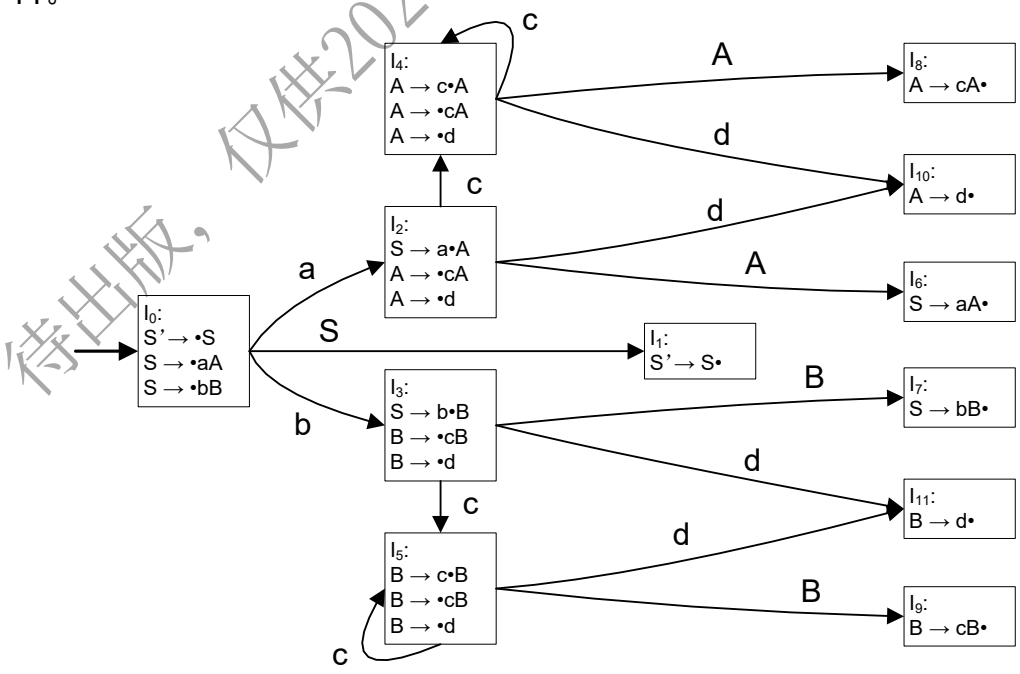

图 4-11 识别文法（4.15）所有活前缀的 DFA

文法（4.15）的 LR（0）分析表见表 4-13。

表 4-13 文法（4.15）的 LR（0）分析表

	action					goto		
	a	b	c	d	$	S	A	B
0	s_2	s_3				1		
1					Acc			
2			s_4	s_{10}			6	
3			s_5	s_{11}				7
4			s_4	s_{10}			8	
5			s_5	s_{11}				9
6	r_1	r_1	r_1	r_1	r_1			
7	r_2	r_2	r_2	r_2	r_2			
8	r_3	r_3	r_3	r_3	r_3			
9	r_5	r_5	r_5	r_5	r_5			
10	r_4	r_4	r_4	r_4	r_4			
11	r_6	r_6	r_6	r_6	r_6			

对于拓广文法 G’，若其 LR（0）分析表没有多重定义入口（在一个表项中最多只有一个移进/归约/接受动作），则称 G’为 **LR（0）文法**，基于 LR（0）分析表的 LR 分析称为 **LR（0）分析**。之所以称为 LR（0）分析，是因为在构建 LR（0）分析表时，若某项目集合 I_k 包含归约项目 $A\rightarrow \alpha\cdot$，则无论句子中下一个符号是什么（或者说向前看了 0 个符号）aciton[k，a]均填入 r_j，其中 j 为产生式 $A\rightarrow \alpha$ 的编号。文法（4.15）就是 LR（0）文法。

4.3.5 构造 SLR（1）分析表

为一般的上下文无关文法构建的 LR（0）分析表并不总是没有多重定义入口的。

例如：对于文法（4.14），其 LR（0）分析表如表 4-14 所示。

表 4-14 文法（4.14）的 LR（0）分析表

	action						goto		
	id	+	*	(	)	$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				Acc			
2	r_2	r_2	**r_2、s_7**	r_2	r_2	r_2			
3	r_4	r_4	r_4	r_4	r_4	r_4			
4	s_5			s_4			8	2	3
5	r_6	r_6	r_6	r_6	r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9	r_1	r_1	**r_1、s_7**	r_1	r_1	r_1			
10	r_3	r_3	r_3	r_3	r_3	r_3			

11	r_5	r_5	r_5	r_5	r_5	r_5			

在表 4-13 中存在两个多重定义入口，即：

（1）action[2，*]= { r_2、s_7 }；

（2）action[9，*]= { r_1、s_7 }。

多重定义入口是由 I_2、I_9 中包含的项目造成的：

（1）I_2 = go (I_0, T) = { E→T・；T→T・*F }，其中既包含移进项目，又包含归约项目；

（2）I_9 = go (I_0, T) = { E→E+T・；T→T・*F }，其中既包含移进项目，又包含归约项目。

一个项目集中既含移进项目又含归约项目，我们称该项目集存在**冲突**。项目集存在冲突必然造成 LR（0）分析表存在多重定义入口。基于有冲突的 LR（0）分析表的 LR 分析无法对某些句子作正确的分析，因为如果在查询 action 表时发现了多重定义入口，将无法决定下一步的动作。

项目集的冲突有两种情况：

（1）**移进-归约冲突**：项目集合中同时含有形如 A→ α ・a β 和 B→ γ ・的项目；

（2）**归约-归约冲突**：项目集合中同时含有形如 A→ α ・和 B→ β ・的项目。

对于规范的 LR（0）项目集族中有冲突的项目集，有的可以通过向前看一个符号（即考察句子中下一个符号）来解决冲突。

解决冲突的方法基于以下分析：含有归约项目的项目集，无论句子中下一个符号是什么都进行归约是不合理。假设句子中下一个符号为 a，项目集包含的归约项目是 A→ α ・，则 a 在 A 的后跟符号集（Follow 集）中才能用 A→ α 进行归约。因为执行归约之后，栈顶的句柄 α 被替换为 A，a 自然跟在 A 后面，如果 a 不在 Follow（A）中，后续分析将无法进行。另外，若项目集中包含形如 X→ α ・b β 的移进项目，若 a≠b，则不能执行移进操作，因为移进之后栈中的文法符号串不是活前缀，后续分析也将无法进行。

假设文法的规范 LR（0）项目集族中存在如下项目集合：

{ X→ α ・b β，A→ γ ・，B→ δ ・}

即该项目集合既存在移进-归约冲突，又存在归约-归约冲突。

如果 FOLLOW（A）∩ FOLLOW（B）∩ { b } = Φ，则可以按如下规则解决冲突（假设句子中下一个符号是 a）：

（1）若 a = b，则移进；

（2）若 a∈ FOLLOW（A），则用产生式 A→ γ 归约；

（3）若 a∈ FOLLOW（B），则用产生式 B→ δ 归约；

（4）否则，报错。

如果文法 G'的规范 LR（0）项目集族中的移进-归约冲突和归约-归约冲突可以用上述方法解决，则称文法 G'为 **SLR（1）文法**。

构造分析表过程中，如果在填入归约项时考察了产生式左边非终结符号的 Follow 集（即句子中下一个符号 a 属于产生式左部非终结符号 A 的 Follow 集才归约），这样形成的分析表称为 **SLR（1）分析表**。基于 SLR（1）分析表的 LR 分析称为 **SLR（1）分析**。

SLR（1）分析是在 LR（0）项目集中存在冲突的时候才通过向前看一个符号（考察句子中下一个符号）来解决冲突，是一种简单的 LR（1）分析方法，即 Simple LR（1）分析。

文法（4.14）中各非终结符号的 FOLLOW 集计算如下：

（1）FOLLOW（E）={ +,), $ };

（2）FOLLOW（T）={ +, *,), $ };

（3）FOLLOW（F）={ +, *,), $ }。

为文法（4.14）构建 SLR（1）分析表，如图 4-15 所示。

表 4-15 文法（4.14）的 SLR（1）分析表

	action						goto		
	id	+	*	(	)	$	E	T	F
0	s_5			s_4			1	2	3
1		s_6				Acc			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

文法（4.14）的 SLR（1）分析表没有多重定义入口，因此是 SLR（1）文法，基于该分析表可以对文法（4.14）的句子作 SLR（1）分析。

【例 4-26】：基于文法（4.14）的 SLR 分析表（表 4-14）对句子 id*id 作 SLR（1）分析。

解：分析过程见表 4-16，这个分析过程和表 4-11 的基于识别活前缀的 DFA 对句子 id*id 进行分析的过程是一致的。

表 4-16 对句子 id*id 的 SLR（1）分析

步骤	栈	输入串	action	goto
1	0	id*id $	s_5	
2	0id5	*id $	r_6	3
3	0F3	*id $	r_4	2
4	0T2	*id $	s_7	
5	0T2*7	id $	s_5	
6	0T2*7id5	$	r_6	10
7	0T 2*7F10	$	r_3	2
8	0T2	$	r_2	1
9	0E1	$	Acc	

4.3.6 LR（1）和 LALR（1）分析表的构造

每个 SLR（1）文法都是无二义性的，但是也存在很多不是 SLR（1）的无二义性文法。

【例 4-27】：考察如下拓广文法：

(0)S’ → S

(1)S → L=R

(2)S → R （4.16）

(3)L → *R

(4)L → id

(5)R → L

构造该文法的规范的 LR（0）项目集族及识别它所有活前缀的 DFA，见图 4-12。

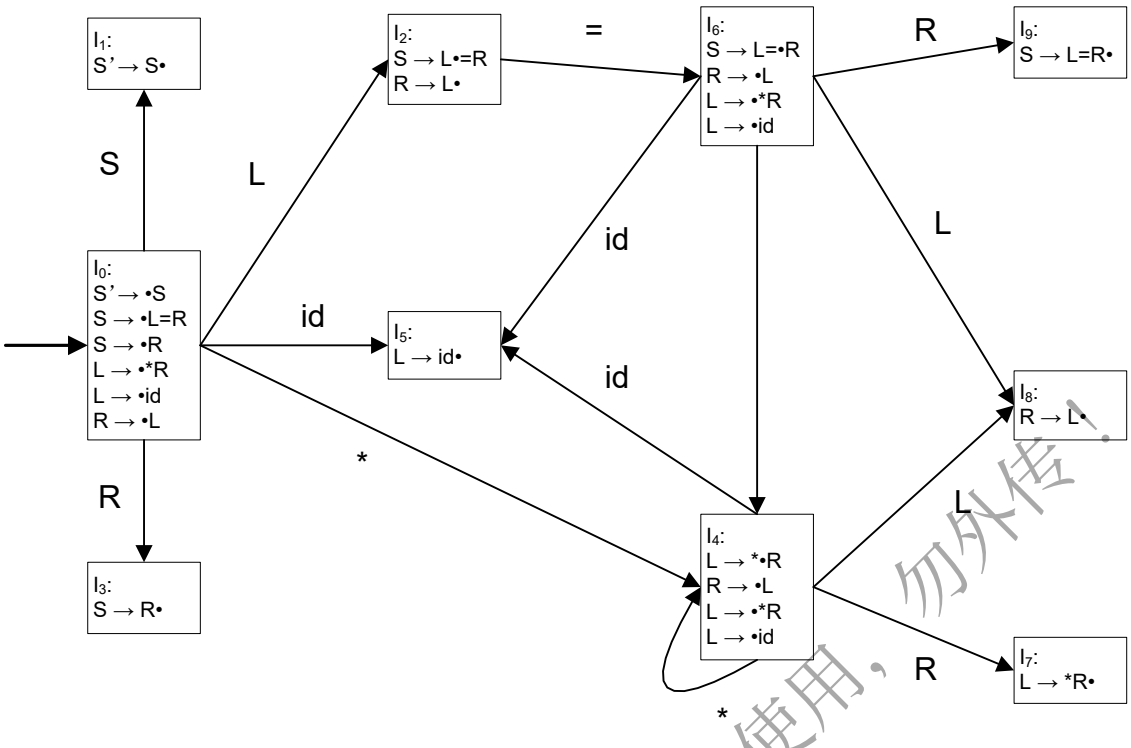

图 4-12 识别文法（4.16）所有活前缀的 DFA

其中 I_2 = { S → L•=R；R → L• }存在移进-归约冲突。由于归约项目“R → L•”左部非终结符号 R 的 FOLLOW 集中包含“=”，而移进项目“S → L•=R”圆点后面的符号也为“=”，故构造 SLR（1）分析表时，当状态 I_2 面临“=”时既可以作移进（将“=”压入栈中），也可以作归约（用产生式“R → L”归约），于是 SLR（1）分析表存在多重定义入口，文法（4.16）不是 SLR（1）文法。

造成冲突无法消除的原因是 SLR（1）分析器功能还不够强大，不能记住足够多的上下文信息。当它看到栈顶有一个可归约为 L 的串时，不能确定语法分析器应该对输入“=”采取什么动作。

从对文法(4.16)的分析可知，当尝试用某个产生式 A→ α 对栈顶符号串进行归约时，不仅要向前看一个输入符号，还需要考虑当前格局下，栈中所有的符号串 δ α 。只有当把 α 归约为 A 后得到的符号串 δ A 和后续的输入符号 a 能构成该文法的某一个规范句型的前缀时，才能够用该产生式对 α 进行归约。那么问题是怎么才能保证 δ Aa 是文法的某个规范句型的前缀呢？

实际上从前面的分析，我们可以简单地总结为，SLR(1)在解决冲突的时候引入非终结符号的 FOLLOW 集，考察的范围过大。为此，可以考虑为每一条产生式的归约设置一个向前看的搜索符，即在原来的 LR(0)项目 A→ α • β 中增加一个搜索符，代表当右部的符号串出现在栈顶时，紧跟在 A 后面的符号，从而得到如下形式的 LR(1)项目：[A→ α • β，a]， a 必须保证 δ Aa 是文法某个规范句型的前缀，即 LR(1)项目对应的活前缀 δ α 必须是有效的。

形式地说，LR(1)项目[A→ α • β ， a]对活前缀 γ 有效，如果存在着最右推导序列：$S \Rightarrow^*_{rm} \delta Aw \Rightarrow_{rm} \delta\alpha\beta w$，其中：

(1) $\gamma = \delta\alpha$，且，

(2) a 是 w 的第一个符号，或者 w 是空串 ε 且 a 为$。

【例 4-28】：考虑文法：

S→BB

B→aB | b

可以发现，该文法存在一个最右推导 $S \Rightarrow^*_{rm} aaBab \Rightarrow_{rm} aaaBab$。根据上述定义，令 δ=aa，A=B，w=ab，α=a，β=B，可知 LR(1)项目[B→a•B，a]对于活前缀 γ=aaa 是有效的。另外还有一个最右推导 $S \Rightarrow^*_{rm} BaB \Rightarrow_{rm} BaaB$。根据这个推导，我们知道 LR(1)项目[B→a•B，$]是活前缀 Baa 的有效项目。

构造规范的 LR(1)项目集族的方法实质上与构造规范的 LR(0)项目集族的方法是一样的，只需要对 closure 和 go 函数进行相应的修改。具体构造过程如下：

（1）构造 LR(1)项目集的闭包函数 CLOSURE(I)。

(a) I 中的项目都在 CLOSURE(I)中；

(b) 若[A→α·Bβ, a]在 CLOSURE(I)中， B→γ 是文法的一条产生式，b∈FIRST(βa)，则将[B→·γ, b]加到 CLOSURE(I)中；

(c) 重复第 2 步，直到项目集不再增大。

（2）构造转换函数 GO(I, X)。

(a) 初始化 J 为空集；

(b) 对于 I 中任何形如[A→α·Xβ, a]的项目，将[A→αX·β, a]项目加入到 J 中；

(c) GO(I, X) = CLOSURE(J)。

构造规范的 LR（1）项目集族 items(G)：

（1）初始化集合 C 为 CLOSURE({[S'→·S, $]})；

（2）对于 C 中的每个项目集 I，每个文法符号 X，如果 GO(I, X)不为空且不在 C 中，将 GO(I, X)加入 C 中；

（3）重复 2，直到 C 不再增大。

对拓广文法（4.16），令 I=[S'→·S，$]为初始项目集，构造出它的规范的 LR(1)项目集族，结果如图 4-13 所示。

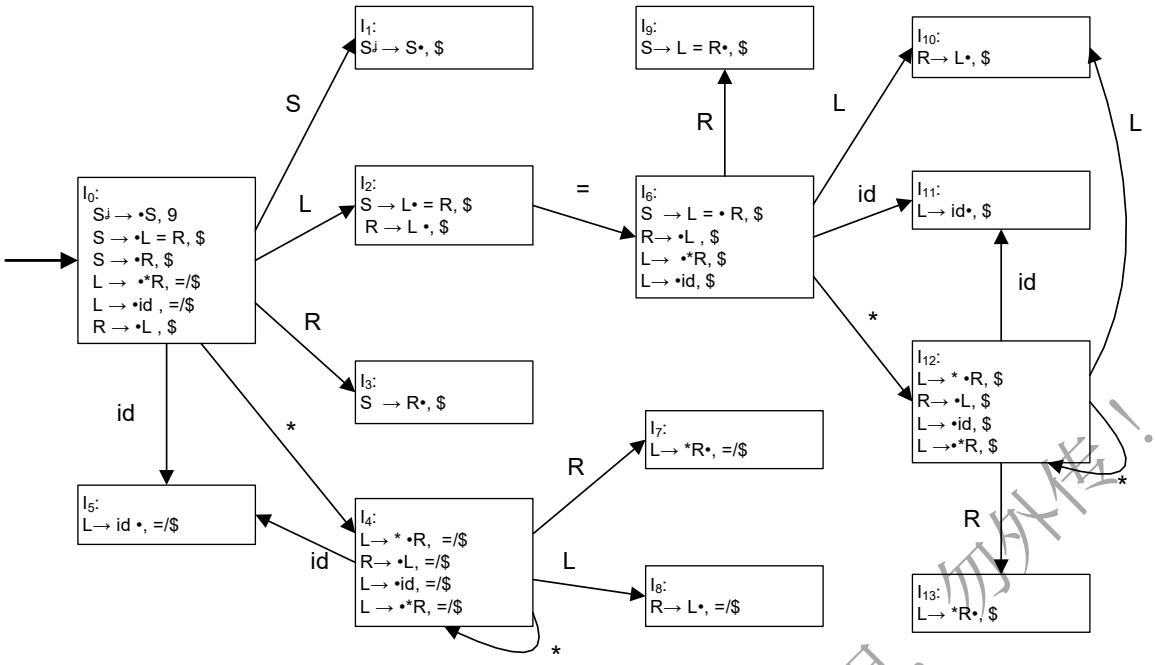

图 4-13 文法（4.16）的规范的 LR(1)项目集族及转换函数

需特别注意的是，根据算法流程，项目集 I_0 中的项目[L→·*R, =/$]，其向前搜索符=和$的计算是经过两次计算后合并得到的：由项目[S→·L=R，$]可以得到[L→·*R, =]；由项目[S→·R, $]得到[R→·L, $]，进一步得到[L→·*R, $]。同理，项目[L→·id, =/$]通过类似方式得到。

下面给出从规范的 LR(1)项目集族构造 LR(1)分析表的算法。

算法 4-5：构造 LR(1)分析表。

输入：一个拓广文法 G’；

输出: LR(1)语法分析表(ACTION 和 GOTO)；

步骤：

（1）构造 G’的规范的 LR(1)项目集族 C，令 C = {I_0, I_1, …, I_n}；

（2）分析器的状态 i 对应于项目集 I_i，0 为开始状态，状态 i 的语法分析动作按以下规则确定：

(a) 若项目[A→α·aβ, b]在 I_i 中，并且 GO(Ii, a) = I_j，a 为终结符号，那么设置 ACTION[i, a] = shift j，即移进；

(b) 若项目[A→α·, a]在 I_i 中，并且 A≠ S’，那么设置 ACTION[i, a] = reduce A→α，即归约；

(c) 如果项目[S’→S· , $]在 I_i 中，那么设置 ACTION[i, $]=“accept”；

(d) 对于非终结符号 A，如果 GO(I_i , A) = I_j，设置 GOTO[i, A] = j。

（3）所有没有按第 2 步获得填充的位置均表示出错。

基于以上算法可以得到文法（4.16）的 LR(1)分析表，见表 4-17。

表 4-17 文法（4.16）的 LR(1)分析表

	action				goto		
	*	id	=	$	S	L	R
0	s_4	s_5			1	2	3
1				Acc			
2			s_6	r_6			
3				r_3			
4	s_4	s_5				8	7
5			r_5	r_5			
6	s_{12}	s_{11}				10	9
7			r_4	r_4			
8			r_6	r_6			
9				r_2			
10				r_6			
11				r_5			
12	s_{12}	s_{11}				10	13
13				r_4			

从 LR(1)分析表可以看出，对于所有的 LR(1)归约项目均不存在无效的归约。如果一个文法的 LR(1)分析表不存在多重定义入口，或者任何一个 LR(1)项目集中没有“移进-归约”冲突或“归约-归约”冲突，我们称该文法为 **LR(1)文法**。基于 LR(1)分析表的 LR 分析称为 **LR(1)分析**。

在大多数情况下，同一个文法的 LR(1)项目集个数比 LR(0)项目集个数要多得多。这是因为对于同一个 LR(0)项目集，由于向前搜索符不同而对应多个 LR(1)项目集。项目集个数的急剧增长，带来的是分析时时间效率和空间效率的降低。

为了克服这一缺点，Frank DeRemer 提出了一种折中的方法，即 LALR(1)分析法。这种方法的基本思想是将 LR(1)项目集中的同心项目集合并，以减少项目集的个数。如图 4.13 中的 I_5={ [L→id・, =/$]}和 I_{11}={ [L→ id・, $]}，它们的第一个分量，即 LR(0)项目部分相同，只是搜索符不一样，它们就是一对同心项目集，其中的心即为{ L→id・}。同样的，I_4 和 I_{12}， I_7 和 I_{13}，I_8 和 I_{10}，均为为同心集。

如果将一个文法 G 的所有同心项目集进行合并，而且合并后的项目集中不存在“移进-归约”冲突或“归约-归约”冲突，则得到的规范项目集族即为规范的 LALR(1)项目集族。根据该项目集族构建的分析表就是 **LALR(1)分析表**，该文法称为 **LALR(1)文法**，基于 LALR(1)分析表进行的 LR 分析称为 **LALR(1)分析**。

同心集合并后的项目集族具有以下几个特点：

（1）同心集合并之后，心仍相同，只是向前搜索符集合为各同心集的原有向前搜索符的并集；

（2）合并同心集后，转换函数自动合并；

（3）可能会有冲突，但只会出现归约-归约冲突，而不会是移进-归约冲突；

（4）合并同心集后，可能会推迟发现错误的时间，但位置仍然是准确的。

经过同心集合并，可将图 4.13 转换为图 4.14。

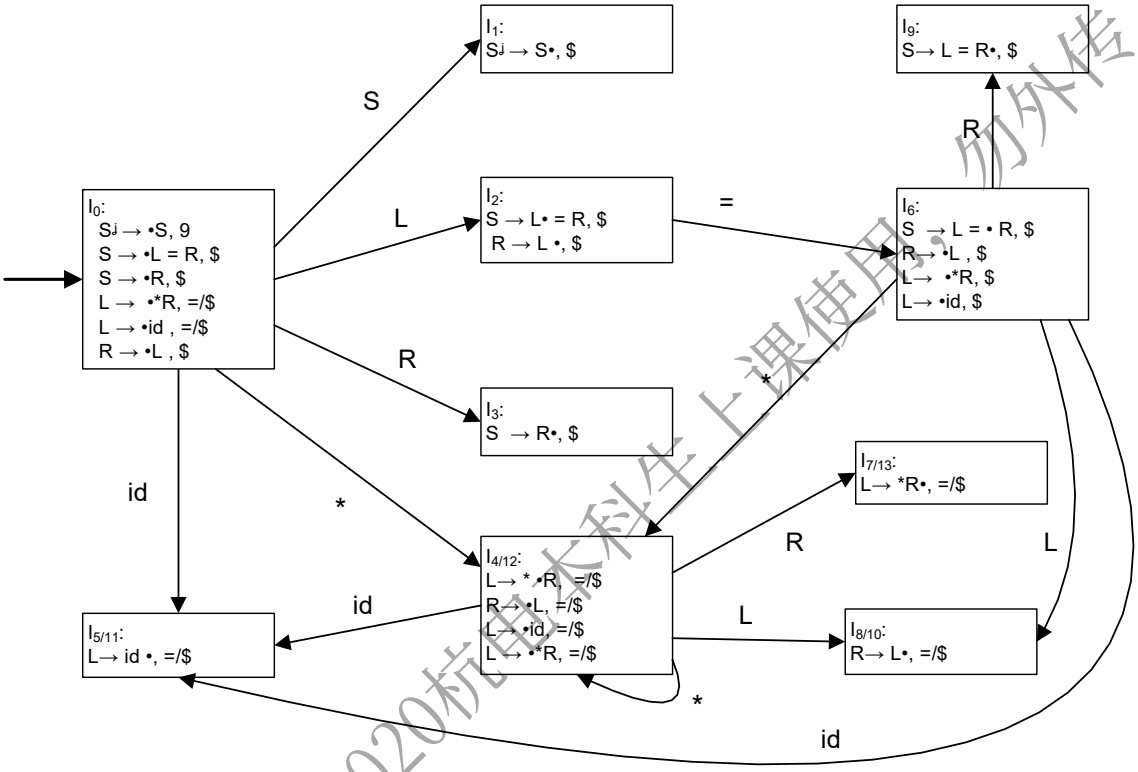

图 4-14 文法（4.16）的规范的 LALR(1)项目集族及转换函数

基于图 4-14 可构造文法（4.16）的 LALR(1)分析表，如表 4-18 所示。

表 4-18 文法（4.16）的 LALR(1)分析表

	action				goto		
	*	id	=	$	S	L	R
0	s_4	s_5			1	2	3
1				Acc			
2			s_6	r_6			
3				r_3			
4	s_4	s_5				8	7
5			r_5	r_5			
6	s_4	s_5				8	9
7			r_4	r_4			
8			r_6	r_6			
9				r_2			

4.4 语法分析程序的自动构造工具

本节介绍语法分析器的生成器 YACC（Yet Another Compiler Compiler）。YACC 结合 LEX 可以帮助我们构造编译器的前端。YACC 产生于 20 世纪 70 年代初，现在仍然是 UNIX 和 Linux 等系统下的流行工具。YACC 生成的编译器主要是用 C 语言写成的语法分析器，需要与词法分析生成器 LEX 结合在一起使用，再将两部分生成的 C 程序一并编译，其构造分析器的过程如图 4-15 所示。

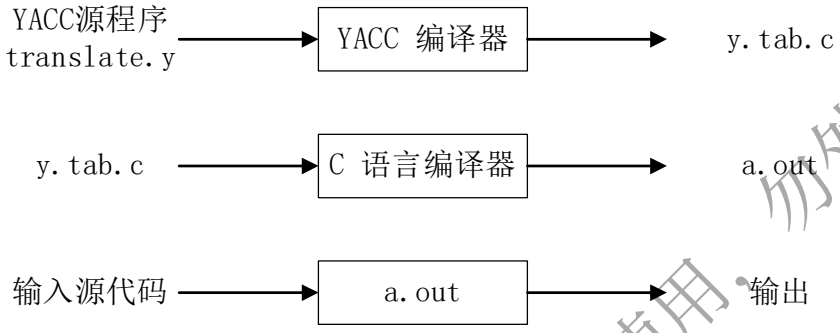

图 4-15 YACC 构建分析器的过程

首先，用 YACC 语言规范建立一个 YACC 源程序文件，如 translate.y。该文件通过 YACC 编译器处理，生成 C 语言程序 y.tab.c。 y.tab.c 文件包含了 LALR(1)分析器的 C 实现代码以及用户的辅助代码。C 语言程序 y.tab.c 经过 C 编译器后可得到目标程序 a.out。

YACC 源程序由三个部分构成：声明、翻译规则、辅助过程，各部分之间用%%行分割，其格式如下：

声明部分

%%

翻译规则部分

%%

辅助过程部分

YACC 声明部分包含两种可选的内容，一种是用{%和%}括起来的 C 语言声明，通常是声明后面内容可用的常量和变量信息。另外一种内容是声明文法的终结符号信息。

翻译规则部分是多个候选的产生式规则，如：

左部 ： A_1 {语义动作程序 1}

| A_2 {语义动作程序 2}

…

| A_n {语义动作程序 n}

;

其中的语义动作程序是 C 语言程序段，用来描述当使用该条候选式进行归约时，需要执行的语义动作。

YACC 源程序的第三部分是一些 C 语言辅助代码。

下面通过一个简单计算器例子来说明 YACC 源程序的基本结构，见图 4-16。

```
%{
#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;
         }
```

图 4-16 简单计算器的 YACC 源程序

在 YACC 的产生式中，没有加引号的字母数字串，如果没有被声明为终结符号，则被看作非终结符号；所有加了单引号的字符，如‘+’会被看成字符+所代表的单词符号。右部各个候选式及其语义动作之间通过竖线隔开，最后一个用分号结尾。第一个左部非终结符号默认为开始符号。在语义动作的 C 程序段中，符号$$代表左部非终结符号的属性值，而$i 表示右部第 i 个文法符号的属性值。

在图 4-16 给出的 YACC 程序中，表达式 E 有两个产生式，即 E→E+T|T，其语句包括：

```
expr :   expr  ‘+’  term { $$=$1+$3;}
       | term
       ;
```

在第一个产生式中，非终结符号 term 是右部第三个符号，‘+’是第二个符号，其语义动作是把右部 expr 的属性值和 term 的属性值相加，把结果赋给左边 expr 的属性值。第二个产生式的语义动作为缺省动作，默认将 term 的属性值直接赋给左部 expr 的属性值。

针对简单计算器工作特点，该 YACC 程序增加了一条开始产生式：

```
Line : expr  ‘\n’  { printf( “%d\n”, $1; )
```

该产生式的作用是代表该计算器的输入为一个表达式后面跟一个换行符，对应的语义动作是输出该表达式的计算结果并换行。

YACC 的第三部分是 C 语言代码段，函数 yylex()的主要作用是进行词法分析，也可以由 LEX 来产生。yylex 返回单词符号二元组（单词类别码，值），返回的单词类别（如 DIGIT）必须在 YACC 程序第一部分声明，值则必须通过内置变量 yyval 传给分析器。

上述例子中的词法分析器是比较粗糙的，它使用 C 语言库函数 getchar()每次读取一个字符，如果读入的是数字字符，则将其值存入 yyval，返回类别码 DIGIT，否则把字符本身作为单词符号返回。如果输入非法字符，则自动退出程序。

4.5 小结

语法分析的任务是为合法的句子构造分析树，构造分析树的方法有很多。按构造分析树结点的顺序来分类，可以将语法分析技术分为两大类，即自顶向下语法分析技术和自底向上语法分析技术。

自顶向下语法分析按照自上而下的顺序构建分析树。首先构造根结点，再构造根结点的子结点及子结点的子结点，直到构建完所有的叶子结点，叶子结点从左至右排列就是要分析的句子。从推导的角度来看，自顶向下语法分析就是构造一个以句子为目标的从开始符号出发的（最左）推导序列。如何选择每步推导的产生式是自顶向下语法分析的关键问题。在每步推导时随机选择一条候选产生式（不确定的自顶向下分析）会造成回溯，因而效率低下，

没有实用价值。对于 LL(1)文法，可以通过计算每条产生式的 SELECT 集来确定每步推导唯一正确的产生式，这就是确定的自顶向下分析（即 LL(1)分析）。只有 LL(1)文法才可以做确定的自顶向下分析，含左公因子和左递归的文法不是 LL(1)的。对这类文法实施提取左公因子、消除左递归操作之后有可能把它们转换成 LL(1)的。确定性自顶向下分析的具体实现方法有两种：递归下降子程序法和基于非递归预测分析器模型的方法。

自底向上语法分析按照自下而上的顺序构建分析树。首先构造分析树的叶子结点，然后一层层往上构建内部结点，最后构建根结点。从归约的角度来看，自底向上语法分析就是要构造一个以开始符号为目标的从句子出发的（最左）归约序列。从实现的角度通常又将自底向上语法分析称为移进-归约分析，移进和归约都是相对于一个分析栈来说的，LR 分析器模型是一个采用移进-归约法的自底向上的语法分析器模型。在从左至右扫描句子的过程中，判断何时移进、何时归约是移进-归约分析中要解决的核心问题。LR 分析器模型通过查 LR 分析表来决定什么时候移进、什么时候归约，如果归约又用哪条产生式归约。LR 分析表的构造有多种方法，不同的构造 LR 分析表的方法形成不同的 LR 分析方法。根据构造分析表方法的不同，LR 分析又分为 LR(0)分析、SLR 分析、LR(1)分析、LALR 分析等。

本章的重点是理解掌握自顶向下语法分析和自底向上语法分析中涉及的关键概念和算法。自顶向下分析部分需掌握 FIRST 集、FOLLOW 集和 SELECT 集的计算方法；提取左公因子和消除左递归的算法；无递归预测分析器模型及预测分析算法。自底向上分析部分需理解活前缀在分析过程中的关键作用，掌握构造识别文法所有活前缀的 DFA 的方法及各类 LR 分析表构造方法，理解 LR 分析器模型及 LR 分析算法。

习题

4.1 试消除下列文法 G[E]中存在的左递归：

E→ET+ | ET- | T

T→TF* | TF/ | F

F→(E) | i

4.2 设有文法 G[S]（o, a , d, e, f, b 是终结符号）：

S→MH | a

H→LSo | ε

K→dML | ε

L→eHf

M→K | bLM

求非终结符号的 FIRST 集与 FOLLOW 集。

4.3 设有文法 G[S]：

S→a | ^ | (T)

T→T,S | S

(1) 改写文法（消除左递归或左公共因子）。

(2) 判断改写后文法是否是 LL(1)的，如果是，构造预测分析表。

(3) 给出输入串(a, a)的分析过程。

4.4 设有文法 G[A]：

A→aABe | a

B→Bb | d

(1) 改写文法（消除左递归或左公共因子）。

(2) 判断改写后文法是否是 LL(1)的，如果是，构造预测分析表。

4.5 考虑简化了的 C 语言声明语句的文法 G[<declaration>]，其中<、>括起来的串表示非终结符号，其它符号都是终结符号（注意：int，float，id 均为终结符号）。

<declaration> → <type><var_list>

<type> → int | float

<var_list> → id, <var_list> | id

(1) 在该文法中提取左公共因子。

(2) 为改造后文法的非终结符号构造 FIRST 集和 FOLLOW 集。

(3) 说明改造后的文法是 LL(1)文法。

(4) 为改造后的文法构造 LL(1)分析表。

(5) 给出输入串 int x, y, z 所对应的 LL(1)分析过程。

4.6 设有拓广文法 G[S']：

[0] S' → S

[1] S → S(S)

[2] S → a

(1) 计算该文法的 LR(0)项目集规范族，构造识别器所有规范句型活前缀的 DFA。

(2) 该文法是 LR(0)文法吗？请说明理由。

(3) 构造该文法的 SLR(1)分析表。

(4) 给出识别句子 a(a(a)) 的自底向上分析过程。

4.7 证明如下拓广文法 G[S']不是 LR(0)，但是 SLR(1)文法。

(0) S' → S

(1) S → A

(2) A → Ab | bBa

(3) B → aAc | a | aAb

4.8 若有定义二进制数的文法，G[S']：

S' →S

S → L.L | L

L→LB | B

B→0 | 1

(1) 证明该文法是 SLR(1)文法，但不是 LR(0)文法。

(2) 构造其 SLR(1)分析表。

(3) 给出输入串 101.110$ 的分析过程。

4.9 设有如下文法 G[S']：

[0] S' →S

[1] S→aAD

[2] S→aBe

[3] S→bBS

[4] S→bAe

[5] A→g

[6] B→g

[7] D→d

[8] D→ ε

试构造该文法的 LR(1)项目集规范族（包括项目集及状态图）。该文法是 LALR(1)文法吗？

4.10 设有文法 G[S]:

S → (S) | ε

试判断该文法是否 SLR(1)文法，若不是，给出理由；若是，则构造出其 SLR(1)分析表。

4.11 设有文法 G[S]:

S → aA | bB

A → 0A | 1

B → 0B | 1

(1) 构造识别该文法活前缀的 DFA；

(2) 判断该文法是否 LR(0)文法，若是，请给出其 LR(0)分析表，若不是，给出理由。

4.12 设有文法 G[S]：

S → rD

D → D,i | i

(1) 构造识别该文法活前缀的 DFA；

(2) 该文法是 LR(0)文法吗？请说明理由；

(3) 该文法是 SLR(1)文法吗？若是，构造其 SLR(1)分析表。