第四章　语法分析

上一章讨论了词法分析的任务和方法，是用正规式（它等价于正规文法）描述程序语言的单词符号结构，构造识别单词符号的有穷自动机，而描述程序语言的语法结构，则是基于上下文无关文法进行描述。文法是设计编译程序尤其是语法分析程序的依据。本章我们将讨论如何识别上下文无关文生成的语言。

4.1 语法分析概述

在编译程序对某个源程序完成了词法分析工作以后，就进入了语法分析阶段。由词法分析程序所产生的单词符号流，作为语法分析程序的输入串，按文法规则分析检查是否构成了合法的句子，即程序。语法分析是编译过程的核心部分。语法分析的一项重要任务是检查程序中的语法错误，在分析过程中，如果发现输入的单词符号不能构成语言的句子，则说明有语法错误。在这种情况下，语法分析要确定错误的性质和出错位置，向程序员提示，并在可能时做一些错误的修复工作或善后处理，以便程序能继续编译下去，进一步检查后面的语法错误。

在第二章中我们已经介绍过，通过语法分析可以建立起相应的语法树，按照构成语法树的顺序，语法分析方法可分为两大类，即自顶向下和自底向上分析方法。它们都要判断一输入串是否为一个合法的句子。

对于自顶向下语法分析而言，核心问题在于能否找到从文法开始符号开始的推导序列，使得推导出的句子恰为源程序输入串；或者说，能否从根结点出发，向下生长出一棵语法分析树，其叶结点组成的句子恰为输入串。对于用高级语言书写的源程序而言，自顶向下语法分析方法就是从文法开始符号如“程序”开始逐步推导出与输入源程序符号串相同的符号串。

对于自底向上语法分析而言，核心问题在于能否从输入串出发找到一个归约序列，且该序列能最终归约为文法开始符号；或者说，能否从叶结点出发，向上归约出文法开始符号为根结点的语法分析树，且每步归约，都以待处理的字符串是否已形成句柄为标准。对于用高级语言书写的源程序而言，自底向上语法分析方法就是从源程序的符号串出发，逐步归约到文法的开始符号，如“程序”。

语法分析的不同方法适合于不同形式的上下文无关文法，这些方法包括递归下降分析法、LL（1）文法、简单优先文法、算符优先文法和各种 LR 文法等分析方法。常用的自顶向下语法分析方法有递归下降分析法和 LL（1）分析法，而简单优先文法、算符优先分析法和 LR 系列分析法是典型的自底向上语法分析法，本章将重点介绍 LL（1）分析方法和 LR 系列分析方法。

4.2 自顶向下的语法分析

所谓自顶向下语法分析是基于推导的方法，从文法的开始符号出发，采用最左或者最右推导，试图一步一步地推出输入符号串 α。换句话说，就是以文法的开始符号 S 作为语法树的根部，试图自上而下地为输入符号串构造一棵语法树。若语法树的叶结点从左向右的排列恰好就是输入串 α，则表示该输入串 α 是文法的句子，而这棵语法树就是句子 α 的语法结构。如果构造不出这样的一棵语法树，则输入串 α 就不是这个文法的句子。这种分析过程本质上是一种试探的过程，是反复使用不同的产生式谋求匹配输入串的过程。

在某些文法的自顶向下的分析过程中，可能会遇到“回溯”和“左递归”的问题，为了能有效地运用自顶向下的语法分析方法，应使文法不含左递归及避免回溯。

4.2.1 消除文法的回溯和左递归

1. 回溯问题解决方案

在进行自顶向下语法分析时，当文法规则中具有同一左部而右部有不同规则候选式时，在分析时按候选式规则的次序一个个试探，若能分析下去则成，否则再退回到出错点更换另一候选式规则重新试探。这种方法称之为回溯分析方法。其实质就是使用不同候选式规则反复试探。

例 4.1 文法 G[S] S∷=cAd A∷=ab | a

要判断“cad”是否为该文法的句子，可以分别用 A∷=ab 和 A∷=a 代入产生式规则 S∷=cAd 中试探。第一次选择 ab 候选式，则试探失败，需要退回到出错点重新选择另一候选式 a 进行试探。

一般地，设 U 为文法 G 的任意非终结符号，若 U 有如下规则:

U∷=$\alpha_1 | \alpha_2 | \ldots, \alpha_i, \ldots | \alpha_n$，$\alpha_i \in V^+$

若定义：候选式 α_i 可能推出的所有终结符号串的首符号集合 FIRST(α_i)为 FIRST(α_i)＝{a|$\alpha_i \Rightarrow^* a\ldots$，$a \in V_T$}，显然 FIRST($\alpha_i$)$\subseteq V_T$。

为了避免回溯，我们对文法要求是：

FIRST(α_i) ∩ FIRST(α_j)＝∅ （i≠j）

即对文法中的任意一个非终结符号，其规则右部有多个候选式时，由各个候选式所推出的终结符号串首符号集合要两两不相交。这样，就可能根据此时读进的符号是属于哪个候选式的 FIRST(α_i)，来唯一地确定该选用哪个候选式来匹配输入串。我们将这种匹配方法称之为路标法。

例如：当前输入符号为 b（b∈V_T），如果 b∈FIRST(α_i)，则可以选择第 i 个候选式去匹配输入串。

当文法不满足上述路标法条件，即右部各候选式规则首符号相同时（例如 4.1 所示文法中的 A∷=ab | a），我们可以采用提取左因子法对文法进行改写。

一般地说，如果有规则

U∷=aβ1 | aβ2 |…| aβn | γ

则可以将这些规则写成

U∷=aU′ | γ

U′∷=β1 | β2 |…| βn

其中 a 称为左公因子，经过反复提取公因子即可将每个非终结符的所有候选式的首符号集合变得两两不相交。

例 4.2 对文法 G[S]: S∷=cAd　　A∷=ab | a 进行改写，消除回溯得到新的文法：

提取左公因子，得到新的 G[S]: S∷=cAd　　A∷=aB　　B∷=b | ε

2. **左递归问题解决方案**

在自顶向下语法分析过程中，假定现在要用非终结符号 U 去匹配输入串，而在文法中关于 U 的规则是 U∷=U…。

它是一条直接左递归规则，这种左递归文法将使上述自顶向下的语法分析过程陷入无限循环，即：当试图用 U 去匹配输入串时会发现，在没有吃进任何输入符号的情况下，又得重新要求 U 去匹配，如此循环下去而无终止。

若文法具有间接左递归，即有 U⇒+U…。那么，也会发生上述问题。

对于文法规则中的直接左递归，消除是比较容易的，一种方法被称为“重复表示法”，该方法用扩充的 BNF 表示法改写语法规则。

假定一个文法中有关于非终结符的规则为：

A∷=Aα | β

其中 α 非空，β 不以 A 开头，则等价地改写为：

A∷=β{α}

例 4.3 下述直接左递归规则：

E∷=E+T | T

可改写为：

E∷=T{+T}

还可用另一种被称为“改写法”的方法来改写形如文法规则 A∷=Aα | β 的直接左递归。

对 A 引入一个新的非终结符 A′，将 A∷＝Aα|β 等价写成：

A∷=βA′

A′∷=αA′ | ε

由于 β 不以 A 开头，α 不以 A′开头，因此改写后两条规则不是直接左递归规则。同样可以证明这种形式和原来的形式是等价的。

就一般而言，关于 A 的规则具有下面形式：

A∷=Aα1 | Aα2 |…| Aαn | β1 | β2 |…| βn

这时可改写成如下形式：

A∷＝A(α1 | α2 |…| αn)| β1 | β2 |…| βn

由“改写法”消除直接左递归方法，得：

A∷=(β1|β2|…|βn)A′

A′∷=(α1 | α2 |…| αn)A′ | ε

例 4.4 A∷＝Ac | Aad | bd | e 等价于 A ∷＝A(c | ad) | bd | e，所以可以改写成：

A∷＝(bd | e)A′ （即 A ∷＝bdA′ | eA′）

A′∷=(c | ad)A′ | ε （即 A′∷=cA′ | adA′ | ε）

上述两种方法可消除任意直接左递归，但不能消除两步或多步推导形成的左递归。

例如，有文法 G[S]

S∷=Qc | c

Q∷=Rb | b

R∷=Sa | a

该文法无直接左递归，但有间接左递归，例如有：

S⇒Qc⇒Rbc⇒Sabc

即，S⇒+Sabc

消除间接左递归的基本思路是：先将间接左递归变成直接左递归，然后消除直接左递归。

例 4.5 A∷＝aB | Bb （1）

B∷＝Ac | d （2）

先将（1）代入（2）中，得：

B∷＝Bbc | aBc | d （3）

由此将（3）改写为：

B∷＝（aBc | d）B′

B′∷＝bcB′ | ε

加入文法开始符号的产生式得消除左递归后的等价文法为：

A∷＝aB | Bb

B∷＝(aBc | d)B′

B′∷＝bcB′ | ε

消除文法递归的一般算法要求：文法不含形如 A⇒＋A 的推导，也不存在 A∷＝ε 这样的规则。具体算法思想如下：

（1）将文法 G 的所有非终结符整理成某种顺序 U 1, U2, … U n。

（2）从 U1 开始消除 U1 规则的直接左递归。

（3）假设规则形如 U1∷=x1 | x2 | x3 | … | xk，U2∷=U1(y1 | y2 | y3 | … | yt)，则

用左部为 U_1 的所有规则的右部 $x_1 | x_2 | x_3 | \ldots | x_k$ 替换左部为 U_2 规则的右部中的 U_1，形如 $U_1(y_1 | y_2 | y_3 | \ldots | y_t)$，结果变成 $U_2::=(x_1 | x_2 | x_3 | \ldots | x_k)(y_1 | y_2 | y_3 | \ldots | y_t)$，并消除 U_2 规则的直接左递归。

（4）用类似的方法把 U_1、U_2 的右部替换左部为 U_3、右部以 U_1、U_2 开始的规则后，消除 U_3 规则中的直接左递归。

（5）重复上一步，直到最后把左部为 U_1、U_2、…U_{n-1} 的右部带入 U_n 规则中，并消除 U_n 中的直接左递归。

（6）消除多余规则。

至于为什么这套算法中不允许有 A→ε 的规则存在，大家可以看这样一个例子：文法 G[A]: A→Bb | ε （1） B→AAb | a （2），将（1）代入（2）中将得到 B→BbAb | Ab | a 仍然存在间接左递归规则 B→Ab。

当然，我们也不允许有 A⇒＋A 的推导存在，例如文法 G[A]: A→B | a B→C | b C→A | c 也不能适用于这套消除左递归的算法。

4.2.2 LL（1）分析方法

LL（1）分析方法是一种自顶向下不带回溯的语法分析方法，LL 的意思分别是：从左（Left）到右扫描输入符号串并建立它的最左推导（Left most derivations）。数字 1 是指向前看一个符号来决定选择同一个非终结符号的不同规则。如果向前查看 K 个符号（K>1）才能确定应选规则时，这种语法分析方法就称 LL（K）分析法。LL（1）分析器借助于一张分析表及一个语法分析栈，在一个总控程序控制下实现。

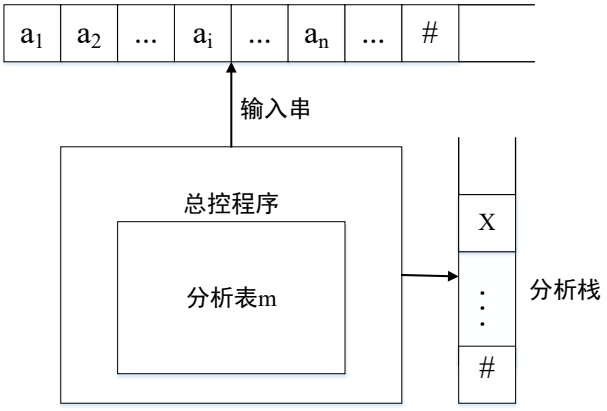

图 4.1 LL（1）分析器结构

1. 总控程序的逻辑结构和工作过程

我们通常把按 LL（1）方法执行语法分析任务的程序称为 LL（1）分析程序或 LL（1）分析器，它由一个总控程序、一张分析表和一个分析栈组成，如图 4.1 所示。其中“输入串”就是待分析的符号串（源程序），它以右界符#作为结尾。分析表 M 可用一个矩阵（或二维数组）来表示。它概括了相应文法的全部信息。分析表的每一行与文法的一个非终结符 A 相关联，而每一列则与文法的一个终

结符号或#相关联。分析表元素 M[A，a]（a∈V_TU {#}）或者指示了当前推导所应使用的产生式，或者指出了输入串中含有语法错误（参看表 4.1）。分析器对每个输入串的分析在总控程序控制下进行。总控程序的分析步骤如下：

（1）分析开始时，首先将符号#及文法的开始符号 S 依次置于分析栈的底部，并把各指示器调整至起始位置，即分别指向分析栈的栈顶元素和输入串的首字符。然后反复执行第（2）步；

（2）设在分析的某一步，分析栈及余留的输入符号串处于如下格局：

$\#X_1X_2\ldots X_{m-1}X_m$　　$a_ia_{i+1}\ldots\#$

其中，X_1，X_2，…，X_m为分析过程中所得的文法符号。此时，可视栈顶符号 X_m的不同情况，分别做如下的动作：

① 若 $X_m\in V_N$，则以 X_m及 a_i组成的符号对(X_m, a_i)查分析表 M，设 $M[X_m, a_i]$为一产生式，譬如说 $X_m\rightarrow UVW$，此时将 X_m从分析栈中退出，并将 UVW 按反序推入栈中（即用该产生式推导一步），从而得到新的格局：

$\#X_1X_2\ldots X_{m-1}WVU$　　$a_ia_{i+1}\ldots\#$

但若 $M[X_m, a_i]$=“ERROR”，则调用出错处理程序进行处理；

② 若 $X_m=a_i\neq\#$，则表明栈顶符号已与当前正扫视的输入符号得到匹配，此时应将 X_m（即 a_i）从栈中退出，并将输入符号指示器向前推进一个位置；

③若 $X_m=a_i=\#$，则表明输入串已完全得到匹配，此时即可宣告分析成功而结束分析工作。

例 4.6　利用 LL（1）分析方法分析 i+i*i 是否是文法 G[E]所定义的句子：

E→TE′

E′→+TE′ | ε

E→FT′

T′→*FT′ | ε

F→（E） | i

（1）建立文法 LL（1）的分析表

相应的分析表如表 4.1 所示（其构造方法，在后面叙述）。

表 4.1　文法 G[E]的分析表

	i	+	*	（	）	#
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→i			F→（E）		

其分析过程如表 4.2 所示。

表 4.2 符号串 i+i*i 的分析过程

步骤	分析栈	余留输入串	所用产生式
（1）	#E	i+i*i#	E→TE′
（2）	#E′T	i+i*i#	T→FT′
（3）	#E′T′F	i+i*i#	F→i
（4）	#E′T′i	i+i*i#	
（5）	#E′T′	+i*i#	T′→ε
（6）	#E′	+i*i#	E→+TE′
（7）	#E′T+	+i*i#	
（8）	#E′T	i*i#	T→FT′
（9）	#E′T′F	i*i#	F→i
（10）	#E′T′i	i*i#	
（11）	#E′T′	*i#	T′→*FT′
（12）	#E′T′F*	*i#	
（13）	#E′T′F	i#	F→i
（14）	#E′T′i	i#	
（15）	#E′T′	#	T′→ε
（16）	#E′	#	E′→ε
（17）	#	#	成功

由上述分析过程可以看出，在分析的每一时刻，当前已读过的符号与栈中的符号一起总是构成了当前的左句型，LL（1）分析器确实构造了输入串的一个最左推导。

下面给出总控程序实现的伪代码。

```
PROCEDURE Parserl;
    BEGIN
    p=0;
    Stack[p]='#';  p=p+1;
    Stack[p]=文法的开始符号;  p=p+1;  //#号和开始符号依次进栈
     get-symbol(a);    //读入输入串的首符号 a
     FLAG=true;        //设置标识位
     WHILE FLAG DO
      BEGIN
       x=Stack[p];  p=p-1;
            IF x∈VT    THEN
                IF x==a THEN
                    get-symbol(a)   //输入串指针进一
                ELSE
                    error
                END IF
            ENDIF
            IF x=='#'    THEN
                IF x==a      THEN
```

```
                FLAG=false      //说明符号栈和输入串都只剩下#，分析过程结束
            ELSE
                error
            END IF
        END IF
        IF M[x, a]=={x→x1x2…xk}  THEN
            依次将 xk, xk-1…x1 压入栈内;  //若 k=0, 即 x1, x2…xk=ε 则不进栈
        ELSE
            error
        ENDIF
    END
END-PROCEDURE
```

2. LL（1）分析表的构造

对于一个 LL（1）分析器，除了分析表因文法而异外，分析栈和总控程序都是相同的，由于总控程序十分简单，非常容易实现，因此，构造一个 LL（1）分析器问题，就主要归结为构造 LL（1）分析表的问题。

为了构造分析表，我们引进与文法有关的集合 FIRST 集和 FOLLOW 集的概念。

假定 α 是文法 G 的任一符号串，或者说 $\alpha\in(V_T\cup V_N)^*$ ，我们定义

FIRST(α)＝{b｜α⇒*b…，b∈V_T}

特别是，若 α⇒*ε，则规定ε∈FIRST(α)

即 FIRST(α)是 α 的所有可能推导的开始终结符号或可能的 ε。

例 4.7 设文法 G[T]:

T ∷＝AB

A ∷＝PQ | BC

P ∷＝pP | ε

Q ∷＝qQ | ε

B ∷＝bB | e

C ∷＝cC | f

求 FIRST(PQ)

由定义有 PQ ⇒ pPqQ = p…

PQ ⇒ εQ ⇒ Q ⇒ qQ ⇒ q…

PQ ⇒ εQ ⇒ Q ⇒ ε

所以 FIRST(PQ)={p, q, ε}

对于一个简单的文法我们用手工可以求得其 FISRT 集，对于复杂的文法我们通常使用下述自动化算法求解。

对于文法中的每一个文法符号串 $X\in(V_N\cup V_T)^*$，构造 FIRST(X)时，

只要连续使用下列规则，直至每个 FIRST 集不再扩大为止。

①若 $X\in V_T$，则 FIRST(X)={X}。

②若 $X\in V_N$，且有形如 X∷＝bα 规则（$b\in V_T$）或（和）X∷＝ε 的规则，把 b 或（和）ε 加入 FIRST(X)中。

③设文法 G 中有形如 $X::=Y_1Y_2\cdots Y_K$ 的规则，若 $Y_1\in V_N$，则将 FIRST(Y_1)中一切非 ε 符号加进 FIRST(X)中，对于一切 $2\leqslant i\leqslant k$，若 $Y_1\Rightarrow^*\varepsilon$，则把 Y_2 的首符号集（除 ε 外）也加进 FIRST（X）中，如此继续下去，直到 $Y_{k-1}\Rightarrow^*\varepsilon$，则把 Y_K 中首符号集（除 ε 外）也入 FIRST(X)中。

④若 $Y_1Y_2\cdots Y_k$ 每个非终结符号都可能推导出空符号串，即 $Y_1Y_2\cdots Y_k\Rightarrow^*\varepsilon$，则把 ε 也加进 FIRST(X)中。

现在，可以对文法 G 的任何符号串 $\alpha=X_1X_2\cdots X_n$，按如下步骤构造 FIRST(α)。

首先置 FIRST(α)=∅，然后将 FIRST(X_1)中一切非 ε 的符号加进 FIRST(α)，若 $\varepsilon\in$FIRST(X_1)，再将 FIRST(X_2)的一切非 ε 加进 FIRST(α)中，如此等等。最后，若对于 $1\leqslant i\leqslant n$，$\varepsilon\in$FIRST(X_i)，则再将 ε 加进 FIRST(α)中。

例 4.8 文法 G[E]，求每个产生式右部的 FIRST 集。

E∷＝TE′

E′∷＝+TE′ | ε

T∷＝FT′

T′∷＝*FT′ | ε

F∷＝（E） | i

我们先来构造该文法中每一个终结符号和非终结符号的 FIRST 集，在此基础上我们再来构造每个产生式右部的 FIRST 集合。

根据规则①，容易得到 FIRST(+)={+}，FIRST(*)={*}，FIRST(（)={（}，FIRST(）)={）}，FIRST(i)={i}；

根据规则②，可得到

FIRST(F)={（，i }

FIRST(T′)={*，ε }

FIRST(E′)={+，ε }

根据规则③，由于有产生式 E∷＝TE′和 T∷＝FT′，且 T 和 F 均不能推导出 ε，可得

FIRST(E)=FIRST(T)=FIRST(F)={（，i }

基于以上的求解结果，我们可以得到产生式右部的 FIRST 集合

FIRST(TE′)=FIRST(T)={（，i }

FIRST(+ TE′)={+}　　FIRST(ε)={ε}

FIRST(FT′)=FIRST(F)={（，i }

FIRST(*FT′)={*}　　FIRST(i)={i}

FIRST(（E）)={（}

这些求解结果将用于 LL（1）分析表的构造。

接下来，我们给出 FOLLOW 集的定义和构造方法。

假定 S 是文法 G 的开始符号，对于 G 中的任何非终结符 A，我们定义

FOLLOW(A)={ c ｜ S ⇒*…Ac…， c∈V_T}

特别的，若 S ⇒*…A，则规定#∈FOLLOW(A)。

即： FOLLOW(A)是所有句型中紧接 A 之后出现的终结符或#。

构造 FIRST 集的方法是正向思维的过程，可以直接用定义求解，而构造后继符号集合 FOLLOW 的方法是一个逆向思维的过程，因此，可应用下面算法进行计算。

对文法中每个非终结符 B，为了构造 FOLLOW(B)，可反复应用如下规则，直到每个 FOLLOW 集不再扩大为止。

① 对于文法的开始符号 S，令#∈FOLLOW(S)。

（因为 S ⇒* S，由定义#∈FOLLOW(S)）

②若文法中有形如 A∷=αBβ 的规则，且 β≠ε，则将 FIRST(β)中一切非 ε 符号加进 FOLLOW(B)中。

③若文法中有形如 A∷=αB 或 A∷=αBβ 的规则，且 β⇒*ε，则 FOLLOW(A)中全部符号均属于 FOLLOW(B)，其中 α 可以为 ε。

上述的②和③中所述的规则也可以合并描述如下：

若文法中有形如 A∷=αBβ 的规则，且 β≠ε，则将 FIRST(β)中一切非 ε 符号加进 FOLLOW(B)中；若 β=ε 或者 β ⇒*ε，则 FOLLOW(A)中全部符号均属于 FOLLOW(B)；其中 α 可以为 ε。

在具体求解时应关注非终结符号在哪条规则的右部出现。

例 4.9 对例 4.8 文法 G[E]再构造每一个非终结符号的 FOLLOW 集：

根据规则①，

#∈FOLLOW(E)，故 FOLLOW(E)={#}

根据规则②，

）∈FOLLOW(E)，故 FOLLOW(E)={），#}

FIRST(T′)-{ε}⊆FOLLOW(F)，故 FOLLOW(F)={*}

FIRST(E′)-{ε}⊆FOLLOW(T)，故 FOLLOW(T)={＋}

根据规则③，

FOLLOW(E)⊆FOLLOW(E′)，故 FOLLOW(E’)={），#}

FOLLOW(E)⊆FOLLOW(T)，故 FOLLOW(T)={＋，），#}

FOLLOW(T)⊆FOLLOW(T′)，故 FOLLOW(T’)={＋，），#}

FOLLOW(T)⊆FOLLOW(F)，故 FOLLOW(F)=｛＋，*，），#｝

最后的结果为

FOLLOW(E)=FOLLOW(E′)＝｛），#｝

FOLLOW(T)=FOLLOW(T′)＝｛＋，），#｝

FOLLOW(F)=｛＋，*，），#｝

求出 FIRST 集和 FOLLOW 集后，就可以构造文法 G 的 LL（1）分析表，对于 G 中每一个规则 A∷＝α，可按如下算法确定表中各元素：

① 对 FIRST(α)中每一终结符 a，置 M[A，a]＝“A→α”；

② 若 ε∈FIRST(α)，则对属于 FOLLOW(A)中的每一符号 b（b 为终结符或#），置 M[A，b]＝“A→α”；

③ 把 M 中所有不能按规则①、②定义的元素均置为出错。

按上述规则为例 4.8 文法 G[E]所构造的 LL（1）分析表如表 4.1 所示。例如 E∷＝TE′，我们已经求得 FIRST(TE′)=｛（，i｝，根据规则①，置 M[E，（]＝“E∷＝TE′”以及有 M[E，i]＝“E∷＝TE′”。此外，E′∷＝+TE′|ε，非终结符号 E′有两个候选式，其中 FIRST(+TE′)={+}，根据规则①置 M[E′，+]＝“E′∷＝+TE′”；由于 FIRST(ε)={ε}，规则①不适用，但是此时 ε∈FIRST(α)，所以要采用规则②，已经求得 FOLLOW(E’)={），#}，所以置 M[E′，）]＝“E′∷＝ε”和 M[E′，#]＝“E′∷＝ε”。

在此需要强调的是，在某些例子中，可能某条候选的产生式既适用于规则①又适用于规则②。例如文法 G[A]：A→X　X→aY|bAc|ε　Y→cY|c，显然我们可以求得 FIRST(X)={a, b, ε}，FOLLOW(A)={c, #}，那么对于候选式 A→X，根据规则①，置 M[A，a]＝“A→X”以及 M[A，b]＝“A→X”；但此时 ε∈FIRST(X)，因而根据规则②，对于 FOLLOW(A)中的 c 和#，需要置 M[A，c]＝“A→X”以及 M[A，#]＝“A→X”。

为了节省分析表的空间和提高分析效率，可以将 LL（1）分析表进一步简化。不需要将整个规则 A∷=α 记入 M[A，a]中，只需将规则右部存于分析表的元素中。如假设 $\alpha=X_1X_2\cdots X_n$，则记入表中的 α 可将它们倒序 $X_nX_{n-1}\cdots X_2X_1$ 存放，这样使总控程序工作方便。与此同时，若 X_1 为终结符，则它与输入符号 a 匹配时，因 X_1 无需入栈，而输入指针移向下一个输入字符，这样就减少了分析步骤。

一个文法 G，若它的分析表不含多重定义入口，则它是一个 LL（1）文法。可以证明，一个 LL（1）文法所定义的语言，恰好是它分析表 M 所识别全部句子。

LL（1）文法有一些明显的性质，它不是二义的，也不含左递归，且若文法 G 是 LL（1）文法，当且仅当文法 G 中同一非终结符号的任何两个规则 A∷=α|β 满足如下条件：

① FIRST(α)∩FIRST(β)=∅，也就是 α 和 β 推导不出以某同一终结符 a 为首的符号串。

② α 和 β 中最多只有一个可能推出空串。

③ 如果 β⇒*ε，那么 α 推出任何串不会以 FOLLOW(A)中的终结符开始，反之亦然。

根据 LL（1）的工作原理可以说明如上的三点条件。我们假设对于 A∷=α | β，有 FIRST(α)∩FIRST(β)={a}不为空，这也就意味着在构造的 LL（1）分析表中，置 M[A，a]=“A∷=α”还是“A∷=β”是无法确定的，违背了“自顶向下语法分析在推导的每一步期望能唯一指定某条产生式规则继续往下推导”的初衷。此外，我们还要求 A∷=α | β 中，α 和 β 中最多只有一个可能推出空串，如若不然，若 ε∈FIRST(α)同时有若 ε∈FIRST(β)，那么对属于 FOLLOW（A）中的每一符号 b（b 为终结符或#），置 M[A，b]=“A→α”还是“A→β”是无法确定的，因此存在多重定义入口问题。关于第③个条件，我们通过例 4.10 这个例子来加以说明。

例 4.10 文法 G[S]：

S∷=iCtSS′ | a

S′∷=eS | ε

C∷＝b

根据前面文法分析表构造方法，构造出它的分析表如表 4.3 所示。对于 S′∷=eS | ε，First(eS)={e}，因此在表 4.3 中需要置 M[S′ , e]=“S′∷=eS”；由于 ε⇒*ε，对于 FOLLOW(S′)= FOLLOW(S)={#，e}中的每一个符号需要置 M[S′, e]=“S′∷=ε”以及 M[S′, #]=“S′∷=ε”，这就造成了 M[S′, e]存在多重定义入口。这种情况恰好违背了上述的第③个条件，我们要求“如果 β⇒ *ε，那么 α 推出任何串不会以 FOLLOW(A)中的终结符开始，即若 β⇒ *ε，则 FIRST(α)∩FOLLOW(A)=∅”。

由表 4.3 可知，虽然该文法是 2 型文法却不是 LL（1）文法。因为从它的分析表可以看出，M[S′, e]处有两条规则，即 M[S′, e]有多重定义入口，所以该文法不是 LL（1）文法。

表 4.3　例 4.10 文法分析表

	a	b	e	i	t	#
S	S∷=a			S∷=iCtSS′		
S′			S′ =eS　S′ ∷=ε			S′ ∷=ε
C		C∷=b				

最后，还需要提及，在进行自顶向下语法分析时，若每一步推导不是向前看一个符号，而是需要看 K 个符号才能唯一的确定所选用的规则，那么我们就把此种语法分析称之为 LL（K）分析，把满足此种分析条件的文法称为 LL（K）

文法。例 4.11 是一个典型的 LL（K）文法。

例 4.11 有文法 G[S]: S ∷= aSb | aabb

问该文法是否是 LL（1）文法？若不是，至少应该向前看几个字符？

该文法显然不是LL(1)文法,至少向前看3个字符(分析任一个句型aaaabbbb就可以看出）才能确定到底是选用 S ∷= aSb 还是 S ∷= aabb 继续往下推导。

4.3 自底向上的语法分析

自底向上语法分析，是从输入符号串出发，试图把它归约成识别符号。从图形上看，自底向上语法分析过程是以输入符号串作为末端结点符号串，向着根结点方向往上构造语法树，使识别符号正是该语法树的根结点。自底向上语法分析是一个不断进行直接归约的过程。任何自底向上语法分析方法的关键是要找出这种句柄。

目前已有多种自底向上语法分析技术的实现方法。如简单优先分析法、算符优先分析法以及 LR（K）系列分析法等。几种语法分析方法对相应的文法都有一定的要求。例如：简单优先分析法和算符优先分析方法要求文法的各终结符号对间至多只有一种优先关系成立等。LR 分析法是 1965 年由 Knuth 提出的一种自底向上语法分析方法，可用于一大类上下文无关文法分析，这种技术叫做 LR（K）分析技术。LR（K）是目前最常使用的语法分析方法，因此本节重点介绍该方法，其他方法可参考其他编译原理书目。

4.3.1 LR 分析器的逻辑结构和分析过程

LR（K）分析器是这样一种分析程序，它总是从左至右扫描输入串，并按自底向上进行规范归约，在这种分析过程中，它至多只向前查看 K 个输入符号就能确定当前的动作是移进还是归约；若动作归约，它还能唯一地选中一个规则去归约当前已识别的句柄。若该输入串是给定文法的一个句子，则它总是可以把这个输入串归约到文法的开始符号，否则报错并指明它不是该文法的一个句子。

LR 分析是当前最一般的分析方法，其原因是能用上下文无关文法描述的程序设计语言结构一般可以构造 LR 分析器进行识别；其次，LR 分析法是一般性的无回溯的移进—归约分析法，并且能有效地实现；再者，LR 分析器在自左至右扫描输入串时，可以尽可能地发现语法错误。除此以外，还可用自动方式构造一个 LR（K）分析器的核心部分——分析表。

从逻辑上而言，一个 LR 分析器包括两部分：一个总控程序和一张分析表。一般说来，所有 LR 分析器总控程序是一样的，只是分析表各不相同。不同分析表有不同构造方法，共有四种不同分析表构造方法，它们分别是 LR（0）、SLR（1）、LR（1）及 LALR（1）分析表构造方法。LR（0）构造分析表功能太弱，

它的分析能力最低，但它是其它构造法的基础。SLR（1）是简单 LR 分析表构造法的简称，这是一种比较容易实现的方法，SLR（1）分析表的功能也不太强，但比 LR（0）稍强些。LR（1）分析表功能最强而且也适合于很多文法，对于通常的程序设计语言来说，虽然 LR（1）方法分析强大，但是实现它需要付出的代价也最大。LALR 称为向前看 LR 分析表构造法，这种方法构造分析表的功能介于 SLR（1）和 LR（1）之间，适用于大多数高级程序设计语言的结构，并且可以比较有效地实现。

LR 分析法的主要缺点是，用手工构造分析程序工作量太大，为此，人们已经设计出构造 LR 分析程序的专门工具——LR 分析程序自动生成器，比如 YACC，有了这样的生成器，只要写出上下文无关文法，就可以用它自动产生该文法的分析器，所以，这种分析方法更受人们的重视。

在逻辑上，一个 LR 分析器结构如图 4.2 所示。它是由一个输入符号串，一个下推状态栈，以及一个总控程序和分析表组成的。

实际上在分析时读入的符号是不进栈的。为使分析过程更加清晰，我们另设一个符号栈（实际上只有一个状态栈用于存放状态）。

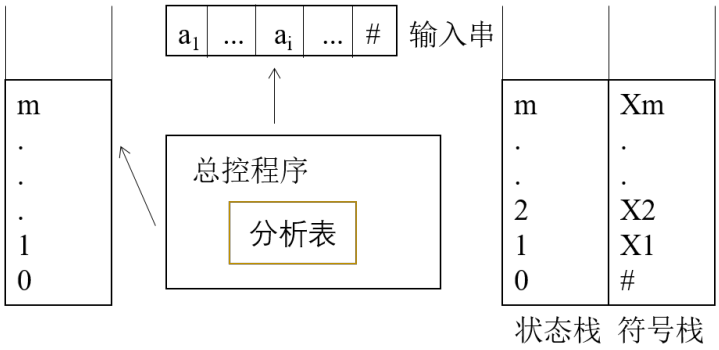

图 4.2　LR 分析器结构

LR 分析器的核心是分析表，分析表由两个子表组成：

1）分析动作表

表 4.4 分析动作表

	a_1	a_2	…	a_m
0	ACTION[0，a_1]	ACTION[0，a_2]	…	ACTION[0，a_m]
1	ACTION[1，a_1]	ACTION[1，a_2]	…	ACTION[1，a_m]
…	…	…	…	…
n	ACTION[n，a_1]	ACTION[n，a_2]	…	ACTION[n，a_m]

分析动作表如表 4.4 所示，其中：0，1，…，n 为分析器各状态，a_1，a_2，… a_m 为文法的全部终结符号，句子界限符 ACTION[i，a_j]指明：当状态 i 面临输入符号 a_j 时应采取的分析动作。ACTION[i，a_j]有如下四个取值：

ACTION[i，a_j]=S_j　　移进动作，当前输入符号 a_j 进入符号栈，状态 j 进入状态栈

ACTION[i，a_j]= r_j　　按第 j 个产生式进行归约

ACTION[i，a_j]= acc　　接受

ACTION[i，a_j]= ERROR　出错

2）状态转换表

表 4.5　状态转换表

	X_1	X_2	…	X_p
0	GOTO[0，X_1]	GOTO[0，X_2]	…	GOTO[0，X_p]
1	GOTO[1，X_1]	GOTO[1，X_2]	…	GOTO[1，X_p]
…	…	…	…	…
n	GOTO[n，X_1]	GOTO[n，X_2]	…	GOTO[n，X_p]

状态转换表如表 4.5 所示，其中：X_1，X_2，…，X_p 是文法字汇表中全部非终结符号，S_1，S_2，…，S_n 为分析器各状态，GOTO[S_i，X_j]指明当状态 S_i 面对文法非终结符号 X_j 时下一状态是什么。

LR 分析器的工作是在总控程序控制下进行的，其过程如下：

（1）分析开始时，首先将初始状态 0 及句子左界限符#推入分析栈，同时和输入串构成一个三元式为

（0，#，$a_1a_2\cdots a_n$#）

其中 0 为初态，#为句子左界限符，$a_1a_2\cdots a_n$ 是输入串，其后的#为句子右界限符

（2）设在分析的某一步，分析栈和余留输入符号串表示为

（01…m，#$X_1X_2\cdots X_m$，$a_ia_{i+1}\cdots a_n$#）

这时用当前栈顶状态 m 及正扫视的输入符号 a_i 组成符号对去查分析动作表，并根据分析表中元素 ACTION[m，a_i]所规定的动作进行分析。分析动作表每一元素 ACTION[m，a_i]所规定动作不外是下列四种可能之一：

①若 ACTION[m，a_i]＝S_k，为移进动作，这表明句柄尚未在栈顶部形成，正期待着移进输入符号以形成句柄，故将当前输入符号 a_i 推入符号栈中，同时将新状态 m+1 推入状态栈中，则三元式变为

（01…mk，#$X_1X_2\cdots X_ma_i$，$a_{i+1}a_{i+2}\cdots a_n$#）

② 若 ACTION[S_m，a_i]＝r_j，为归约动作，其中 r_j 是指文法中第 j 个规则 A∷＝β，r 是规则右部长度。此时按规则 A∷＝β 执行一次归约动作，这表明栈顶部的符号串 $X_{m-r+1}X_{m-r+2}\cdots X_m$ 已是当前句型（对非终结符 A）的句柄。按第 j 个产生式进行归约，此时将分析栈从顶向下的 r 个符号退出，使状态 m-r 变成栈顶状态，再将文法符号 A 推入栈中，其三元式为

（01…m-r，#$X_1X_2\cdots X_{m-r}$A，$a_ia_{i+1}\cdots a_n$#）

然后再以（m-r，A）查状态转换表，设 GOTO[m-r，A]＝k，将此新状态推入状态栈中则三元式变为

（01…m-r k，#$X_1X_2…X_{m-r}A$，$a_ia_{i+1}…a_n$#）

归约动作不改变现行输入符号，输入串指示器不向前推进，它仍然指向动作前的位置。

③ 若 ACTION[m，a_i]= acc，为接受动作，则表明当前输入串已被成功地分析完毕，则三元式不再变化，宣布分析成功。

④ 若 ACTION[m， a_i]= ERROR，则三元式变化过程终止，报告错误。

（3）重复上述（2），直到在分析某一步，栈顶出现“接受状态”或“出错状态”为止。

对于前者，其三元式变为（0z，#Z，#）

其中 Z 为文法开始符号，z 则为使动作 ACTION[z，#]=“accept”即接受的唯一状态。

一个 LR 分析器工作过程就是一步一步地变换三元式，直至执行“接受”或“报错”为止。

例 4.12 设已知文法 G[E]：(首先对每个文法规则要编号）

① E∷=E+T

② E∷=T

③ T∷=T*F

④ T∷=F

⑤ F∷=（E）

⑥ F∷=i

为了节省空间，我们将文法 G[E]分析动作表（ACTION）和状态转换表（GOTO）根据状态行进行合并，合并之后分析表如下表 4.6 所示（关于表的构造方法以后再讨论）。

表 4.6　例 4.12 文法分析表

状态	ACTION（动作）						GOTO（状态转移）		
	i	+	*	（	）	#	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			

表中所引用记号的意义是：

① S_j 把下一个状态 j 和现行输入符号 a_i 分别移进状态栈和符号栈

② r_j 按第 j 个规则进行归约

③ acc 接受

④ 空白格出错标志，报错

GOTO 表仅对所有非终结符 A 列出 GOTO[m，A]的值，表明所要到达的状态的值。

现以输入串为 i+i*i 为例，给出 LR 分析器对它进行分析的过程，具体如下表 4.7 所示。

现在我们主要关心的问题是，如何根据给定文法来构造 LR 分析表。对于一个文法，如果能够构造出一张分析表，且表中的每个条目都是唯一的（要么为空格，要么为唯一的某个动作），则把这个文法称为 LR 文法。并非所有上下文无关文法都是 LR 文法。但大多数高级程序设计语言一般都可用 LR 文法来描述。直观上说，对于一个 LR 文法，当分析器对输入串进行自左至右扫描时，一旦句柄出现在栈顶，就能及时对它实行归约。

LR 分析器能够用来作出移进-归约决定的一个信息源是向前看 K 个输入符号。一般而言，一个文法，如果能用一个每步顶多向前看 K 个输入符号的 LR 分析器进行分析，则这个文法称为 LR（K）文法。但对大多数高级程序设计语言而言，K=0 或 1 就足够了。因此本书只考虑 K≤1 的情形。

对于一个文法，如果它的任一“移进-归约”分析器存在如下情形：尽管栈的内容和向前看的下一个输入符号都已知悉，但依旧无法确定是采用移进动作还是归约动作；或者，无法从几种可能的归约动作中确定其一，那么这个文法就不是 LR 文法。此外，一个 LR 文法肯定是无二义的，一个不加改造的二义性文法绝不会是 LR 文法。

表 4.7　i+i*i 的 LR 分析过程

步骤	状态栈	符号栈	输入串	分析动作	下一状态
1	0	#	i+i*i#	S_5	5
2	05	#i	+i*i#	r_6	GOTO[0，F]=3
3	03	#F	+i*i#	r_4	GOTO[0，T]=2
4	02	#T	+i*i#	r_2	GOTO[0，E]=1
5	01	#E	i*i#	S_6	6
6	016	#E+	i*i#	S_5	5
7	0165	#E+i	*i#	r_6	GOTO[6，F]=3
8	0163	#E+F	*i#	r_4	GOTO[6，T]=9

9	0169	#E+T	*i#	S_7	7
10	01697	#E+T*	i#	S_5	5
11	016975	#E+T*i	#	r_6	GOTO[7，F]=10
12	0169710	#E+T*F	#	r_3	GOTO[6，T]=9
13	0169	#E+T	#	r_1	GOTO[0，E]=1
14	01	#E	#	acc	

4.3.2 LR(0)分析表的构造

LR（0）分析就是 LR（K）分析当 K=0 的情况，特指在分析的每一步，只要根据当前栈顶状态，就能确定应采取何种分析动作，而无需向前查看输入符号。为了构造 LR（0）分析表，首先引入规范句型活前缀的概念。

（1）规范句型的活前缀

字的前缀：是指字的任意首部。如字 abc 的前缀有 ε，a，ab，abc。

活 前 缀：规范句型（右句型）的一个前缀，如果它不含句柄后任何符号，则称它是该规范句型的一个活前缀。也就是说在活前缀右边增添一些终结符号之后，就可以成为规范句型。

如：S⇒+abcdef， 其中 cd 是句柄，则

ε，a，ab，abc，abcd 是该规范句型的活前缀，而 abcd 是包含句柄的活前缀。

在 LR 分析过程中的任何时候，符号栈里的文法符号 $X_1X_2\cdots X_m$ 应该构成活前缀，把输入串的剩余部分匹配上之后即成为规范句型（如果整个输入串确实构成一个句子的话）。

作为规范句型的活前缀不含有句柄后的任何符号。因此，前缀与句柄的关系可能有三种情况：

①活前缀已包含句柄全部符号，这表明规则 A∷=β 的右部符号串 β 已在符号栈顶形成，因此相应的分析动作应是用此规则进行归约。该可归约的活前缀用 A∷=β・表示。

②活前缀中只包含句柄的一部分符号，意味着形如规则 $A::=\beta_1\beta_2$ 的右部的左边子串 β_1 已出现在符号栈顶，正期待着从余留输入串中看到能由右边子串 β_2 推出的符号串依次入栈。

我们用 $A::=\beta_1\cdot\beta_2$ 表示。

③活前缀不包含句柄的任何符号，这表明规则 A∷=β 的右部符号串 β 中的符号均不在符号栈顶，正期待从余留输入串中由规则 A∷=β 的 β 所能推出的符号串依次入站。

我们用 A∷=・β 表示。

我们把右部某位置上标有圆点的规则称为相应文法的一个 LR（0）项目。特

别地，对形如 A∷=ε 的规则，相应 LR（0）项目为 A→・ 。

例如，

A∷=β	A∷=β・	一个 LR（0）项目
A∷=β1β2	A∷=β1・β2	一个 LR（0）项目
A∷=β	A∷= ・β	一个 LR（0）项目
若 A∷=ε	A∷= ・	一个 LR（0）项目

例如， 一个规则 A∷=aBC，根据圆点的位置不同可以产生四个 LR（0）项目：

A∷＝・aBC，正期待着从余留输入串中由 aBC 推出的符号串依次进栈

A∷＝a・BC，a 已进栈，正期待着从余留输入串中由 BC 推出的符号串依次进栈

A∷＝aB・C，aB 推出的符号串进栈，正期待着从余留输入串中由 C 推出的符号串依次进栈

A∷＝aBC・，aBC 推出的符号串均已进栈形成当前句柄

对于规则 A∷=β 对应项目数为 |β| ＋ 1 个（ |β| 表示 β 所含字符的个数）。显然，不同的 LR（0）项目反映了分析过程中栈顶的不同情况。

Knuth 证明了一个 LR 文法的右句型的所有活前缀能够为有穷自动机所接受。我们可以把文法 G 中每一个项目作为非确定有穷自动机的一个状态，通过构造 NFA，然后再将其确定化为 DFA。

例 4.12 设有文法 G[E]=（{E，A，B}，{a，b，c，d}，P，E）其中 P 由

下列规则组成:

① E∷=aA	④ A∷=d
② E∷=bB	⑤ B∷=cB
③ A∷=cA	⑥ B∷=d

为了方便起见，我们在上述文法中引入一个新的开始符号 S′，并将 S′∷=E 作为第 0 个规则，从而得到所谓 G 的拓广文法 G′，显然，L(G)=L(G′)。将文法拓广的目的是使文法只有一个开始符作为左部规则，这样构造出来的分析器有唯一的接受项目。否则 E∷=aA 和 E∷=bB 就有两个归约项目。

对于文法 G′，其 LR(0)项目有

(1)S′∷=・E	(7)A∷=c・A	(13)E∷=bB・
(2) S′∷=E・	(8)A∷=cA・	(14)B∷=・cB
(3)E∷=・aA	(9) A∷=・d	(15)B∷=c・B
(4)E∷=a・A	(10) A∷=d・	(16)B∷=cB・
(5)E∷=aA・	(11)E∷=・bB	(17)B∷=・d
(6)A∷=・cA	(12)E∷=b・B	(18)B∷=d・

我们可根据它们不同的作用，将一个文法全部的 LR(0)项目分为三类。

（1）对于形如 A∷=β·的项目，因为它表明右部符号串 β 已出现在符号栈顶，此时相应分析动作应按规则进行归约，称此种项目为归约项目。上例中的项目(2)，(5)，(8)，(10)，(13)，(16)，(18)都是归约项目。对于项目(2)显然仅用于分析过程中最后一次归约，它表明整个分析过程已成功地完成，是一个特殊的归约项目，称之为接受项目。

（2）对于形如 A∷=β_1·aβ_2，其中 β_1 可以是 ε，a 是终结符，相应分析动作应将当前输入符号 a 移入栈中，故称此项目为移进项目。上例中项目(3)，(6)，(9)，(11)，(14)，(17)都是移进项目。

（3）对于形如 A∷=β_1·Bβ_2，其中 β_1 可以是 ε，B 是非终结符，由于我们期待着从余留输入符号串中进行归约而得到 B，称此类项目为待约项目。上例中的(1)，(4)，(7)，(12)，(15)都是待约项目。

上述文法共有 18 个 LR(0)项目，所以可以构造一个具有 18 个状态的 NFA。并规定项目(1)为初始状态。

NFA 构造方法如下；

（1）如果状态 i 和 j 出自同一规则，而且状态 j 的圆点只落后于状态 i 的圆点一个位置，如状态 i 表示如下：

A∷=$X_1X_2\cdots X_{i-1}\cdot X_iX_{i+1}\cdots X_m$

而状态 j 表示如下：

A∷=$X_1X_2\cdots X_{i-1}X_i\cdot X_{i+1}\cdots X_m$

则从状态 i 出发，画一条标记为 X_i 的弧到状态 j，如图 4.3 所示。

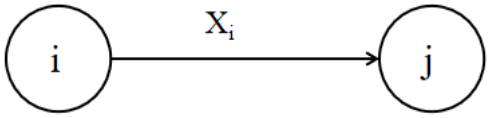

图 4.3　状态 i 和状态 j 的转换（1）

（2）如果状态 i 圆点之后那个符号 X_i 为非终结符，那么从状态 i 画ε弧到所有形如 X_i∷=·β 的项目（状态 j）。

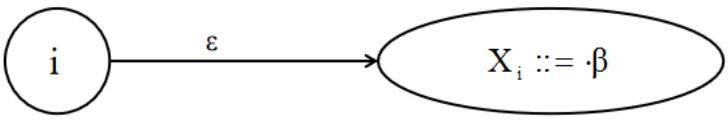

图 4.4　状态 i 和状态 j 的转换（2）

例如文法 G′中状态(1)圆点之后那个符号是 E，为非终结符号，那么从状态(1)画ε弧到所有形如 E∷=·β 的项目（状态），即(3)状态(E∷=·aA)和(13)状态(E∷=·bB)。

（3）凡是属于归约项目的状态是 NFA 的终态，即识别可归约活前缀的终态。

根据上述方法很容易构造出ε-NFA 状态转换图，图中状态(1)是唯一的初态。

画双圆圈者是可归约状态，即识别可归约活前缀的终态，如图 4.5 所示。

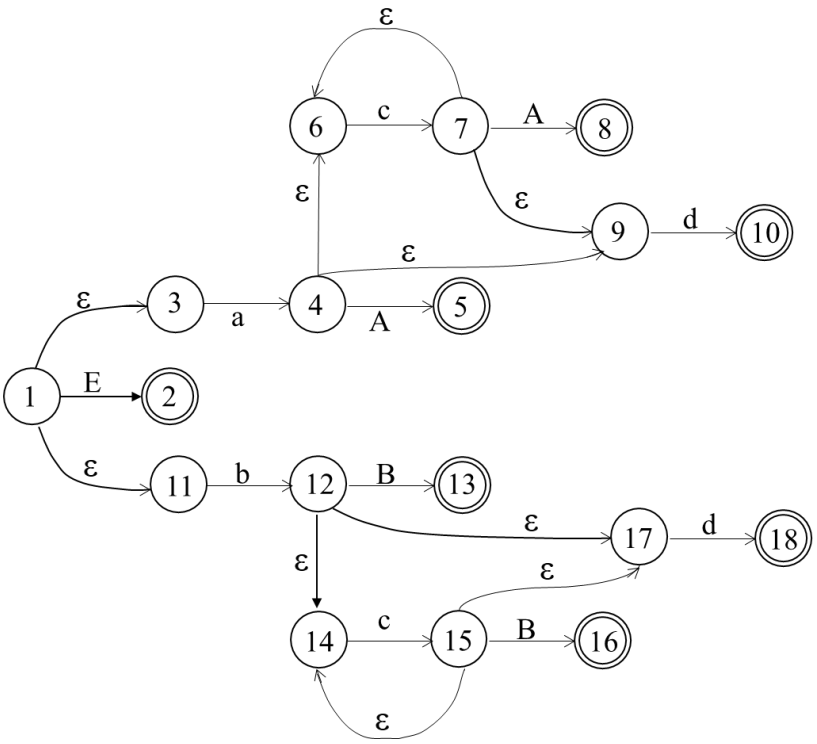

图 4.5 识别活前缀的 NFA

显然，NFA 可以识别文法 G 的活前缀：从 NFA 的开始状态(1)出发，沿着弧所指示的方向前进，当到达某一双圆圈状态时，把所经历的全部弧上的标记依次连成一符号串，此字符串就是某规范句型的一个可归约活前缀，若到达其他任一状态所得符号串就是规范句型活前缀。如可归约活前缀 bcB，规范句型活前缀 bc。

我们采用子集法，消去ε，将 NFA 确定化使其变成 DFA，然后将每一个集合(状态集合)作为 DFA 一个状态，如：I_0，I_1，I_2，……，这样，就可以画出 DFA 状态转换图，其转换图的每一个状态都是以项目集合来表示，如 I_0＝{S′∷=・E，E∷=・aA， E∷=・bB}，具体见图 4.6。

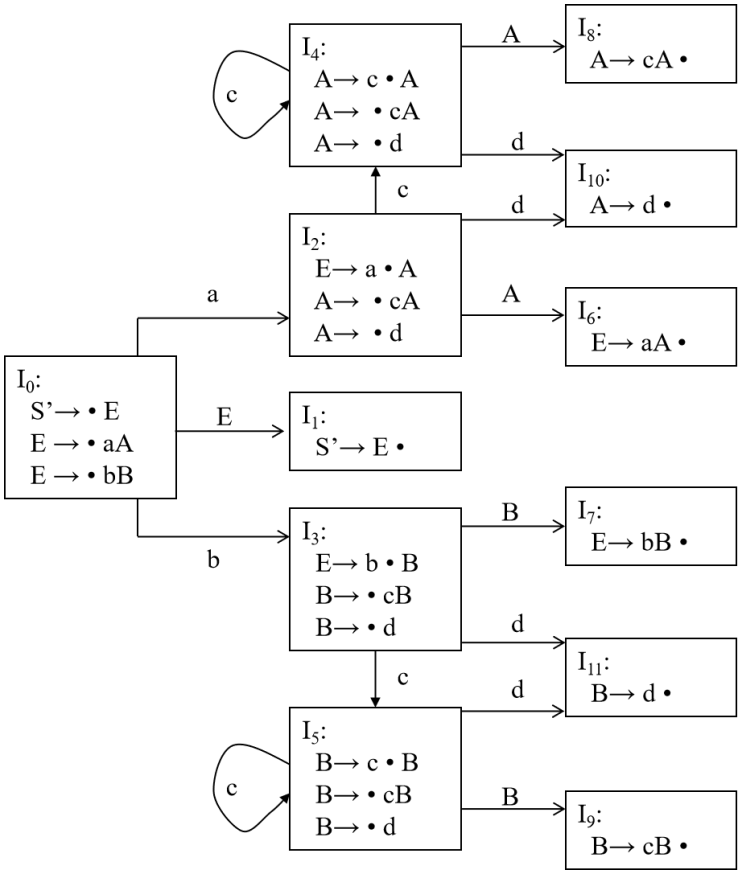

图 4.6　识别活前缀的 DFA

上述的过程较为复杂，尤其是子集法的采用，我们期望能够直接由文法项目构造出 DFA，然后再由 DFA 构造出分析表。下面首先考察 DFA 状态与文法项目的关系。

（1）DFA 状态与文法项目的关系

由 DFA 状态转换图可以看出：DFA 的每一状态都是一个项目子集。其中 I_0 ＝{S′∷＝・E , E∷＝・aA , E∷＝・bB}。

首先来看一下状态 I_0 中包含的各个项目是如何确定的：规定 S′∷＝・E 是属于 I_0 的一个项目，而 E 又是非终结符，E 的规则为：E∷＝aA | bB，所以状态 I_0 还包含 E∷＝・aA 和 E∷＝・bB。由此即得状态 I_0 的项目子集为{S′∷＝・E , E∷＝・aA , E∷＝・bB}。

根据上述分析，可以得到以下结论：

若 NFA 的某一状态的相应项目为 A∷＝β_1・Bβ_2，且 B 是非终结符号，如果有形如 B∷＝β 的规则，其所有相应项目 B∷＝・β 均为项目子集中的项目，如果新获得项目圆点之后的符号又为非终结符号，则又按如上所述方法派生出新的项目，如此继续下去，直到全部新派生项目的圆点之后的符号为终结符号或圆点之后无符号为止。

这时，所有派生出的新项目连同原有项目一起构成一个项目子集，作为 DFA 的一个状态（如：状态 I_0）。实际上，我们将项目 S′∷＝・E 称为项目集 I_0 的基

本项目，上述从项目 S′::=·E 出发构造项目集 I_0 的过程可看作是对基本项目集{S′::=·E}的闭包运算，用 CLOSURE({S′::=·E})来表示。一般地，设 I 为一项目集，则构造 I 的闭包 CLOSURE(I)的方法如下：

① I 中每一项目都属于 CLOSURE(I)；□

② 若形如 A::=β_1·Bβ_2 的项目属于 CLOSURE(I)，那么，对于任何有关 B 推导出的规则 B::=β，相应的项目 B::=·β 也属于 CLOSURE(I)；

③ 重复执行上述两步骤，直至 CLOSURE(I)不再增大为止。

显然，CLOSURE({S′::=·E})={S′::=·E, E::=·aA, E::=·bB}，这就是图 4.6 中的初态 I_0。

（2）DFA 转换函数与文法项目的关系（即从一个状态到另一个状态的弧上标记）

分析 DFA 状态图：状态 I_0 中有项目 S′::=·E，所以可以由状态 I_0 画一标记为 E 的弧指向下一状态 I_1，I_1 包含项目 S′::=E·，圆点向后移一位。

由此得出结论：设 X 为一个文法符号（终结符或非终结符），若 I_i 中有圆点位于 X 左边的项目，形如 A::=β_1·Xβ_2（β_1 可以是 ε），则可从 I_i 出发画一条弧，标记为 X，到下一状态；设此新状态为 I_j，其中项目 A::=β_1X·β_2 为 J，J 显然是新状态集合 I_j 中的一个基本项目，因此，按照上面构造项目集 I_0 的方法，我们就有 I_j=CLOSURE(J)。

例如：I_0 中有项目 S′::=·E（β_1 和 β_2 是 ε），从 I_0 出发画一条弧，标记为 E，到下一状态 I_1，圆点向后移一位，则 I_1 中有基本项目 J={S′::=E·}。由于项目 S′::=E·的圆点后无符号，所以 I_1={S′::=E·}。

同样 I_0 中有项目 E::=·aA（β_1 是 ε），从 I_0 出发画一条弧标记为 a，到下一状态 I_2，圆点向后移一位，则 I_2 中有基本项目 J={E::=a·A}。则 I_2=CLOSURE(J)=CLOSURE({E::=a·A})。那么按照构造 I 的闭包 CLOSURE(I)的方法，可求得 I_2={E::=a·A, A::=·cA, A::=·d}。

为了指明状态 I_j 和状态 I_i 之间的这种转换关系，我们定义一个状态转换函数：

$$GO(I_i, X) = CLOSURE(J)□$$

其中，I_i 为当前状态，X 为文法符号，J 是基本项目集，J={任何形如 A::=β_1X·β_2 的项目 | A::=β_1·Xβ_2 属于 I_i}。

例如：I_0 中有项目 E::=·bB，I_3=GO(I_0, b)=CLOSURE(J)，J={E::=b·B}，由于有文法规则 B::=cB 和 B::=d，所以 I_3={E::=b·B , B::=·cB , B::=·d}。

上面我们分析了 DFA 中状态和文法项目之间的关系、DFA 中转换函数和文法项目之间的关系，所以我们由文法构造 DFA 时，就不必先构造 NFA 然后再用子集法来构造 DFA，我们可以直接由文法来构造 DFA 了。

对于 LR（0）文法，我们构造出识别活前缀的 DFA 后，就可以根据 DFA 的状态转换图来构造 LR（0）分析表。下面给出构造 LR（0）分析表的算法：

假定 C＝{I_0,I_1,I_2,I_3,…,I_n}，为了方便起见，我们用整数 0，1，2，3，…，n 表示状态 I_0，I_1，I_2，I_3，…，I_n。

（1）若 GO(I_i, X)＝I_j，对于 I_i 中形如 A∷＝β_1・Xβ_2 的项目，若 X＝a∈V_T，则置 ACTION[i, a]＝S_j，若 X∈V_N，则置 GOTO[i, X]＝j。

如：I_0 中有 E∷＝・aA，GO(I_0, a)＝I_2，a∈V_T，所以置 ACTION[0, a]＝S_2，I_2 中有 E∷＝a・A，GO(I_2, A)＝I_6，A∈V_N，所以置 GOTO[2, A]＝6（见表 4.6）。

（2）若归约项目 A∷＝β・属于 I_i，假设 A∷＝β 是文法第 j 条规则，则对任意终结符号 a 和句子右界限符#，均置 ACTION[i, a/#]＝r_j，表示按文法第 j 条规则将符号栈顶的符号串 β 归约为 A。

如：I_6 中有项目 E∷＝aA・，其规则 E∷＝aA 是文法的第 1 条规则，所以置 ACTION[6, a] ＝ACTION[6, b]＝ACTION[6, c]＝ACTION[6, d]＝ACTION[6, #]＝r_1。

（3）若接受项目 S′∷＝S・属于 I_i，则置 ACTION[i, #]＝acc，表示接受。

如：S′∷＝E・属于 I_1，所以置 ACTION[1, #]＝acc。

（4）分析表中，凡不能用前 3 条规则填入信息的空白格位置，均表示出错。

构成识别一个文法的活前缀的 DFA 的项目集（状态）的全体称为这个文法的 LR（0）项目集规范族。上例中文法 G[E]的 LR（0）项目集规范族为{I_0,I_1,I_2,I_3,…,I_{11}}。

如果一个项目集中既有移进项目又有归约项目，或一个项目集中有两个以上的不同归约项目，则称这些项目是冲突项目。前面我们构造的项目集中还没有冲突项目。

如果一个文法的项目集规范族的每个项目集中均不存在任何冲突项目，则称该文法为 LR（0）文法。

如：上例文法的 LR（0）项目集规范族的每个项目集中就不存在冲突项目，所以该文法就是 LR（0）文法。

例 4.13　已知文法 G[S]：S∷＝aAa|aBb，A∷＝c，B∷＝c。构造识别活前缀的 DFA，并完成其分析表。

对文法 G[S]的规则进行排序：

① S∷＝aAa

② S∷＝aBb

③ A∷＝c

④ B∷＝c

构造识别活前缀的 DFA 如图 4.7 所示。

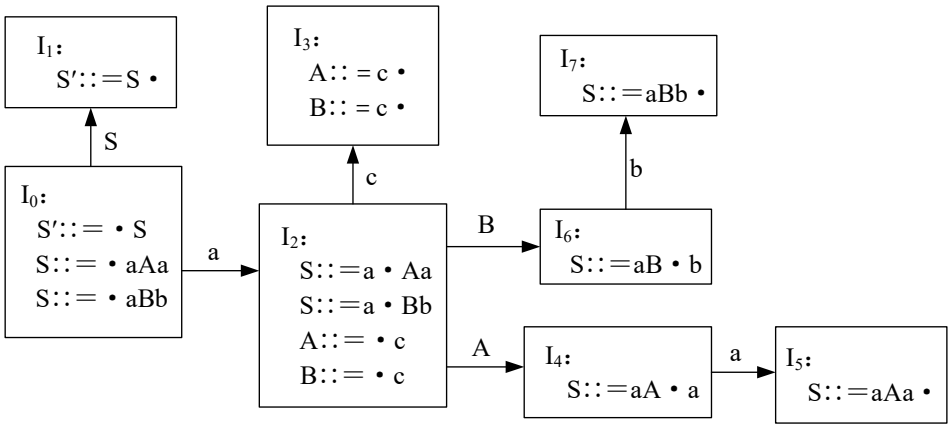

图 4.7 例 4.13 文法的 DFA

从表 4.8 可以看出，按照 LR（0）的方法构造出来的分析表含有多重定义入口，因此该文法不是 LR（0）文法。

表 4.8 例 4.13 文法的 LR（0）分析表

状 态	ACTION				GOTO		
	a	b	c	#	S	A	B
0	S_2				1		
1				acc			
2			S_3			4	6
3	r_3/r_4	r_3/r_4	r_3/r_4	r_3/r_4			
4	S_5						
5	r_1	r_1	r_1	r_1			
6		S_7					
7	r_2	r_2	r_2	r_2			

4.3.3 SLR（1）分析表的构造

上面介绍的 LR（0）方法，是从左向右扫描源程序，当到达某规则右部最右符号时，便识别出这条规则，并且对于每一句柄，无需查看句柄之外的任何输入符号。这种分析方法要求文法的每一个项目集都不含冲突性的项目。但通常程序设计语言文法不符合这种要求。

例如：LR（0）项目集规范族中有这样一个项目集 I_i：I_i＝{A∷＝β_1・bβ_2,B∷＝β・,C∷＝β・}。其中第一个项目是移进项目，第二和第三个项目是归约项目。仔细分析前面讨论的 LR（0）分析表的构造算法可以知道，由于这三个项目相互冲突，因而使得 LR（0）分析表中出现多重定义的分析动作。其原因在于 LR（0）分析表构造规则（2），当有归约项目 B∷＝β・时，无论当前输入符号是什么，在 LR（0）分析表的第 i 行动作子表上均置 r_j（假定 B∷＝β 是文法第 j 条规则）；同样道理，对于项目 C∷＝β・，仍在 LR（0）分析表第 i 行动作子表上均置 r_m（假定 C∷＝β 是文法第 m 条规则）。而 I_i 中第一个项目 A∷＝β_1・bβ_2，假设读入字符 b，I_i 将引申一条箭弧到 I_j，这意味着将下一输入符 b 移进分析符号

栈，于是又发生了是归约还是移进的冲突；如果归约，到底是将栈顶 β 归约为 B，还是归约为 C。这样使得在 LR（0）分析表第 i 行上产生了多重定义。具体如表 4.9 所示。

表 4.9 LR（0）分析表中的冲突示例

状 态	ACTION				GOTO		
	a	b	…	#	A	B	C
0							
…							
i	r_j/r_m	$S_j/r_j/r_m$	r_j/r_m	r_j/r_m			
…							

LR（0）分析表构造时，若是归约项目 A∷＝β・属于 I_i，且 A∷＝β 是文法第 j 条规则，则对任意终结符 a 和句子右界限符#，均置 ACTION[i, a/#]＝r_j，表示按文法第 j 条规则将符号栈顶符号串 β 归约为 A。由于不考虑句柄后任一符号，即不向前看符号，一律置为 r_j，所以当有两个以上归约项目时会出现冲突。

解决这种冲突的办法是在第 i 行上根据输入符号 a 决定唯一的分析动作。为此我们引入 SLR（1）分析法，下面介绍其分析表的构造。

首先解决冲突项目。对于项目集 I_i＝{A∷＝β_1・bβ_2, B∷＝β・ , C∷＝β・}，如果集合 FOLLOW(B)和 FOLLOW(C)不相交，而且不包含 b，那么，当 I_i 表示的状态遇到任何输入符号 a 时，可采取如下“移进—归约”的方法。

（1）当 a＝b 时，则执行移进动作，置 ACTION[i, a]＝S_j；

（2）当 a∈FOLLOW(B)时，执行归约动作且置 ACTION[i, a]＝r_j；

（3）当 a∈FOLLOW(C)时，执行归约动作且置 ACTION[i, a]＝r_m；

（4）当 a 不属于以上三种情况时，置 ACTION[i, a]＝ERROR。

一般地，若一个项目集 I_i 含有多个移进项目和归约项目，例如：I_i＝{A_1∷＝α・$a_1\beta_1$, A_2∷＝α・$a_2\beta_2$, … , A_m∷＝α・$a_m\beta_m$, B_1∷＝α・ , B_2∷＝α・ , … , B_n∷＝α・}。如果集合{$a_1, a_2, \ldots, a_m$}，FOLLOW(B_1)，FOLLOW(B_2)，…，FOLLOW(B_n)两两不相交时，可根据不同的当前符号，对 I_i 中的冲突动作进行区分。这种解决“移进—归约”冲突的方法称作 SLR 方法。

有了 SLR 方法之后，只须对 LR（0）分析表构造规则（2）进行修改，其它规则保持不变。即若归约项目 A∷＝α・属于 I_i，设 A∷＝α 是文法第 j 条规则，则对所有属于 FOLLOW(A)的输入符号 a，置 ACTION[i, a]＝r_j。

对于给定的文法 G，若按上述方法构造的分析表不含多重定义的元素，则称文法 G 是 SLR（1）文法。这里，SLR（1）中的 S 代表 Simple（简单）的意思，而数字 1 代表查看句柄外一个输入符号，即在分析过程中至多只需要向前查看一个符号。

将例 4.13 文法的 LR（0）分析表修改为 SL（1）分析表，见表 4.10 所示。

表 4.10 例 4.13 文法的 SLR（1）分析表

状 态	ACTION				GOTO		
	a	b	c	#	S	A	B
0	S_2				1		
1				acc			
2			S_3			4	6
3	r_3	r_4					
4	S_5						
5				r_1			
6		S_7					
7				r_2			

由 SLR（1）分析表的构造可知 SLR（1）分析法有如下优点：状态数少，造表简单；大多数高级程序设计语言都能用 SLR（1）文法描述。同时 SLR（1）分析法也存在不足，即如果冲突项目的非终结符 FOLLOW 集与有关集合相交时，就不能用 SLR（1）方法解决。下面再来看一个例子。

例 4.14　已知文法 G[S]：

① S∷＝aAa

② S∷＝aBb

③ A∷＝c

④ B∷＝c

⑤ S∷＝bAb

构造识别活前缀的 DFA 如图 4.8 所示和 SLR（1）分析表如表 4.11 所示。

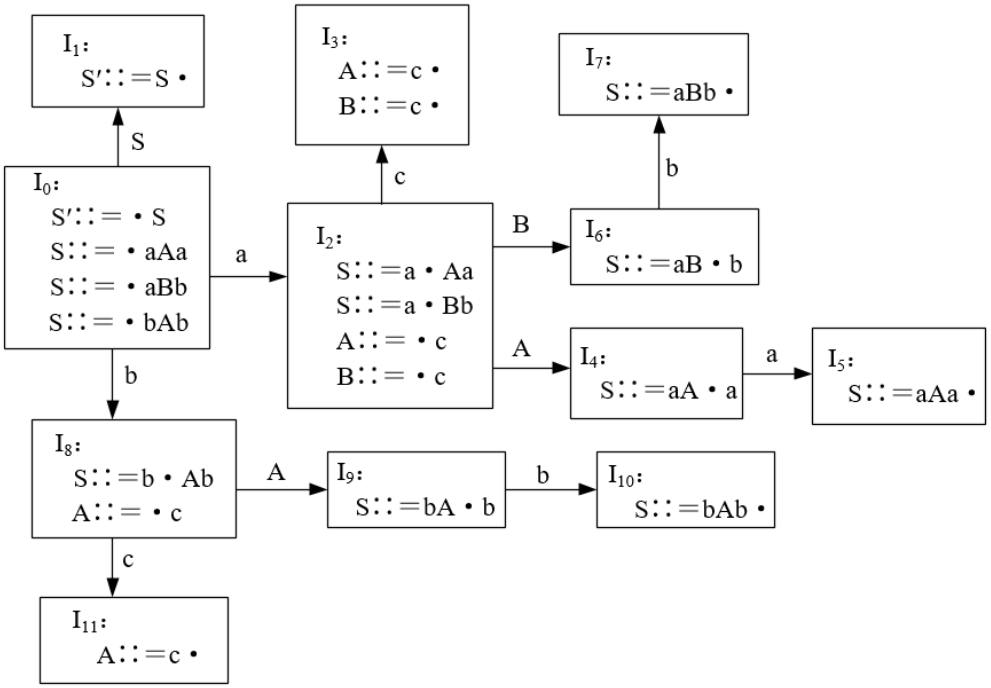

图 4.8 例 4.14 文法的 DFA

表 4.11 例 4.14 文法的 SLR（1）分析表

状 态	ACTION				GOTO		
	a	b	c	#	S	A	B
0	S_2	S_8			1		
1				acc			
2			S_3			4	6
3	r_3	r_3/r_4					
4	S_5						
5				r_1			
6		S_7					
7				r_2			
8			S_{11}			9	
9		S_{10}					
10				r_5			
11	r_3	r_3					

由图 4.8 和表 4.11 可知，项目集 I_3＝{A∷＝c・ , B∷＝c・}存在“归约—归约”冲突，由于 FOLLOW(A)＝{a, b}与 FOLLOW(B)＝{b}相交，故上述冲突不能通过 SLR（1）分析法得到解决。产生这种困境的原因是 SLR（1）分析法包含的信息还不够。所以在归约时，不但要向前看一个符号，而且还要看栈中符号串的情况，才可以知道用哪条规则归约。为了解决这个问题，我们必须将原 LR（0）的项目定义加以扩充，变成 LR（1）项目，也就是说，还要看符号栈中活前缀是什么，再选择相应规则进行归约。

4.3.4 LR（1）分析表的构造

一个 LR（1）项目[A∷＝α・β , a]对活前缀 γ＝δα 有效，是指存在规范推导 S∷＝*δAω ∷＝δαβω（显然 δαβ 是可归约活前缀），且满足下列条件：

（1）当 ω≠ε 时，a 是 ω 首符号；

（2）当 ω＝ε 时，a＝#。□

构造有效的 LR（1）项目集规范族的方法本质上和构造 LR（0）项目集规范族的方法是一样的。我们也需要两个函数 CLOSURE 和 GO。

假定 I 是一个项目集，它的闭包 CLOSURE(I)可按下述方式构造：

（1）I 的任何项目都属于 CLOSURE(I)；

（2）若项目[A∷＝α・Bβ , a]属于 CLOSURE(I)，并对活前缀 γ＝δα 有效，若有 B∷＝η 规则，那么对 FIRST(βa)中的每个终结符 b，形如[B∷＝・η, b]的所有项目也属于 CLOSURE(I)；

（3）重复执行步骤（2），直到 CLOSURE(I)不再扩大，最终得到的 CLOSURE(I)便是 LR（1）的一个项目集。

关于函数 GO(I, X)，其中 I 为一个 LR（1）的项目集，X 为一文法符号，与

LR（0）分析法类似，我们将它定义为：□

GO(I, X) ＝ CLOSURE(J)□

其中，J ＝ {任何形如[A∷＝αX・β , a]的项目 | [A∷＝α・Xβ , a]∈I}。

有了上述 CLOSURE(I)和 GO(I, X)的定义之后，采用与 LR（0）相类似的方法可以构造出给定文法 G 的 LR（1）项目集规范族及其状态转换图。LR（1）分析表的构造算法如下：□

（1）若项目[A∷＝α・Xβ , b]属于 I_i，且 GO(I_i, X)＝I_j，当 X∈V_T 时，置 ACTION[i, X]＝S_j；当 X∈V_N 时，则置 GOTO[i, X]＝j；

（2）若项目[A∷＝α・ , a]属于 I_i，设 A∷＝α 是文法第 j 条规则，则置 ACTION[i, a]＝r_j，表示按文法第 j 条规则将 α 归约为 A；

（3）若项目[S′∷＝S・ , #]属于 I_i，则置 ACTION[i, #]＝acc，表示接受；

（4）分析表中不能按上述规则填入信息的空白格位置，均表示出错。

按照上述算法构造的分析表，若不存在多重定义的元素，则称此分析表为规范 LR（1）分析表。使用这种分析表的分析器叫做规范 LR 分析器。具有规范 LR（1）分析表的文法称为 LR（1）文法。

LR（1）分析法比 LR（0），SLR（1）分析法适用范围更广，对多数高级程序设计语言而言有足够有效的分析能力。若 LR（1）分析法不可进行有效分析，即分析表项仍有多重定义，可继续向前搜索个符号（K≥2），相应分析表称 LR（K）分析表，具有 LR（K）分析表的文法称为规范 LR（K）文法。值得强调的是，任何二义性文法都不是 LR（K）文法。

例 4.15　对如下文法 G：

S∷＝S(S)　　S∷＝ε

构造 LR(1)项目集规范族以及 LR（1）分析表。

引入开始符号 S′。则拓广文法及规则编号如下：

（0）S′∷＝S

（1）S∷＝S(S)　（编号 r_1）

（2）S∷＝ε　　（编号 r_2）

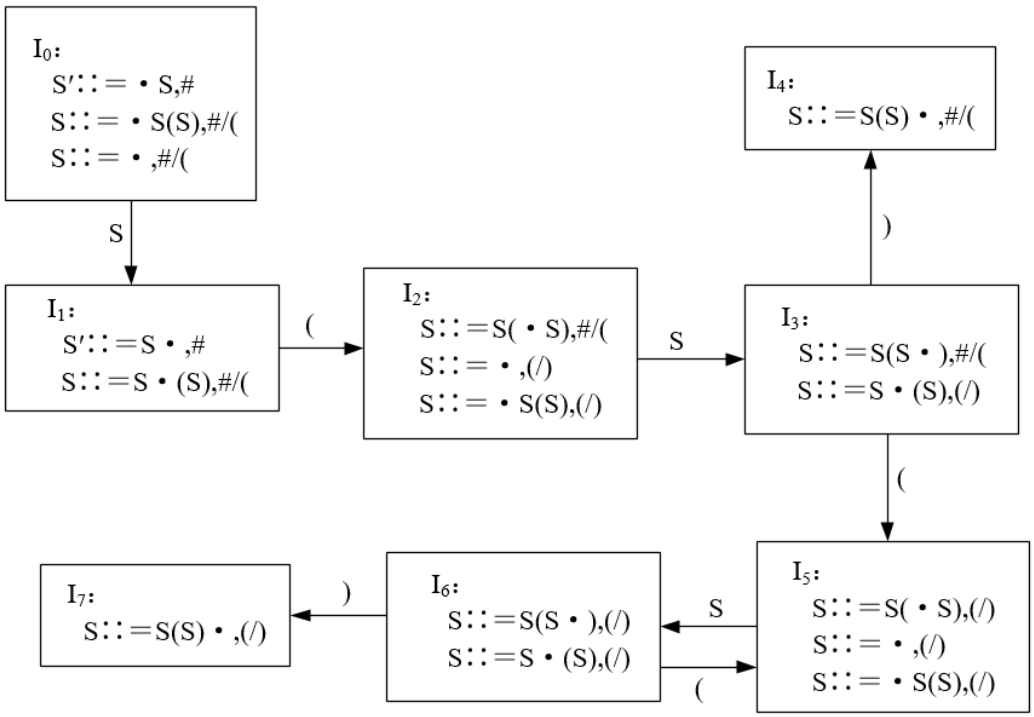

图 4.9 例 4.15 文法的 DFA

LR（1）分析表如下表 4.12 所示：

表 4.12 例 4.15 文法的 LR（1）分析表

状 态	ACTION			GOTO
	(	)	#	S
0	r_2		r_2	1
1	S_2		acc	
2	r_2	r_2		3
3	S_5	S_4		
4	r_1		r_1	
5	r_2	r_2		6
6	S_5	S_7		
7	r_1	r_1		

4.3.5 LALR（1）分析表的构造

LALR 分析法与 SLR 分析法类似，但功能比 SLR（1）强，比 LR（1）弱，且 LALR 分析表比 LR 分析表要小得多。对于同一文法，LALR 分析表与 SLR 分析表具有相同数目的状态。例如，对 PASCAL 语言来说，处理它的 LALR 分析表一般要设置几百个状态，若用规范 LR 分析表则可能要上千个状态。因此，构造 LALR 分析表要比构造 LR 分析表经济得多。

例 4.16　设文法 G[S′]：

（0）S′::＝S

（1）S::＝BB（编号 r_1）

（2）B::＝aB（编号 r_2）

（3）B∷=b （编号 r_3）

得到该文法的 LR（1）项目集如图 4.10 所示，LR（1）分析表如表 4.13 所示。

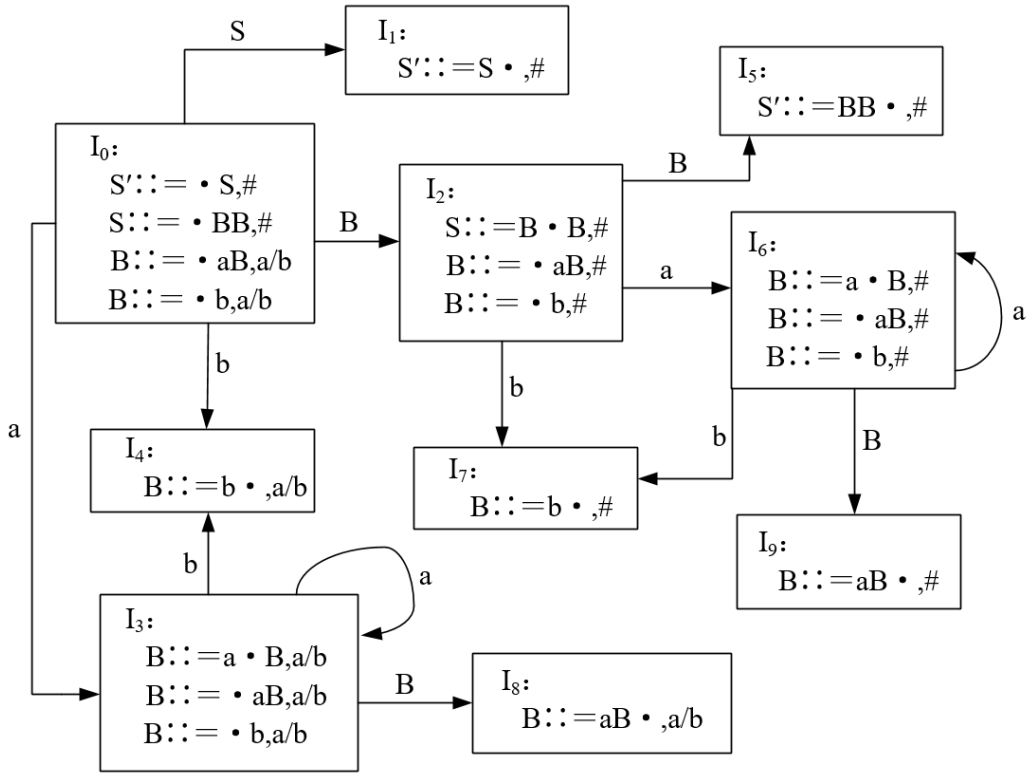

图 4.10 例 4.16 文法的 DFA

表 4.13 例 4.16 文法的 LR（1）分析表

状 态	ACTION			GOTO	
	a	b	#	S	B
0	S_3	S_4		1	2
1			acc		
2	S_6	S_7			5
3	S_3	S_4			8
4	r_3	r_3			
5			r_1		
6	S_6	S_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

从图 4.10 中可以看出，I_4 和 I_7，它们只有一个项目，而且第一个成分（核心项，B∷=b·）相同，不同的只是第二个成分（向前搜索符，分别为 a/b 和#）。该文法所能识别的语言集合为{a^*ba^*b}。假定规范 LR 分析器正在分析输入串

aa…baa…b#，分析器把第一组 a 和第一个 b 移进栈（即 aa…b 进栈），如图 4.10 所示，此时进入状态 4（I_4）。状态 4 的作用在于：如果下一个输入符号是 a 或 b 时，分析器将使用规则 B∷＝b 把栈顶的 b 归约为 B；如果下一个输入符号是#，它就及时地予以报错。当读入第二个 b 后分析器将进入状态 7（I_7），若状态 7 遇到的输入符号不是#，而是 a 或 b 时，就立即报告错误；只有当它遇到句末符#时，分析器才选用规则 B∷＝b 将栈顶 b 归约成 B。□

现在我们要把状态 I_4 和 I_7 合并成状态 I_{47}，I_{47}＝{[B∷＝b・, a/b/#]}。此时当符号栈的栈顶为 b 时，在 I_{47} 状态下，不论遇到 a、b 或#，均将 b 归约为 B，虽然未能及时发现错误，但输入下一个符号时就会发现。于是我们类似地合并状态，使状态数逐步减少，最终变成 LALR（1）分析。哪些状态能够合并，涉及到同心集的概念。

如果除去搜索符以外，两个 LR（1）项目集是相同的，则称为同心集。如本例中的 I_4 与 I_7，I_3 与 I_6，I_8 与 I_9。下面对同心集的特性进行简单说明：

（1）同心集合并后，其转换函数 GO[I, X]可通过自身合成而得到；

（2）同心集合并后不会存在“移进—归约”冲突，但有可能存在“归约—归约”冲突，因为移进和归约不同心，所以不会出现“移进—归约”冲突。

LALR（1）分析表构造算法的基本思想：首先构造 LR（1）项目集，如果它不存在冲突，就寻找同心集合并在一起，若合并后项目集规范族不存在“归约—归约”冲突，就按照合并后项目集规范族构造分析表，其步骤如下：

（1）构造文法 G 的 LR（1）项目集规范族 C＝{I_0,I_1,…,I_n}；□

（2）把所有同心集合并，记为{J_0,J_1,…,J_m}，成为新的项目集规范族，其中含有项目[S′∷＝・S, #]的 J_i 为分析表的初态；

（3）根据{J_0,J_1,…,J_m}构造 ACTION 表：

① 若[A∷＝α・aβ, b]∈J_i，且 GO(J_i, a)＝J_j，a∈V_T，则置 ACTION[i, a]=S_j；□

② 若[A∷＝α・, a]∈J_i，假定 A∷＝α 是文法的第 j 条规则，则置 ACTION[i, a]＝r_j；

③ 若[S′∷＝S・, #]∈J_i，则置 ACTION[i, #]＝acc。

（4）构造 GOTO 表：

假定 J_i＝I_{i1}∪I_{i2}∪…∪I_{it}，则 GO(I_{i1}, X)，GO(I_{i2}, X)，…，GO(I_{it}, X)也是同心集，令 J_j 是它们的合并集，则 GO(J_i, X)=J_j。所以，若 GO(J_i, A)=J_j，A∈V_N，则置 GOTO[i, A]＝j；

（5）分析表中凡不能用（3）和（4）填入信息的空白格，均代表出错。

例如：I_3 和 I_6 是同心集，令 J_i=I_3∪I_6=J_{36}，所以 GO(I_3, B)=I_8 与 GO(I_6, B)=I_9 也是同心集，记为 J_j。J_j=I_8∪I_9=J_{89}，GO(J_i, B)=GO(J_{36}, B)=J_{89}。所以置 GOTO(36, B)=89。

根据 LALR（1）分析表构造方法，可得例 4.16 文法 G[S′]的 LALR（1）分析表如下表 4.14 所示。

表 4.14 例 4.16 文法的 LALR（1）分析表

状 态	ACTION			GOTO	
	a	b	#	S	B
0	S_{36}	S_{47}		1	2
1			acc		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3	r_3		
5			r_1		
89	r_2	r_2	r_2		

经上述步骤构造出的分析表若不存在冲突，则称它为 LALR（1）分析表，利用 LALR（1）分析表的 LR 分析器称为 LALR 分析器，能构成 LALR 分析表的文法称为 LALR（1）文法。

当输入串正确时，不论是 LR 分析器，还是 LALR 分析器，都给出了同样的“移进—归约”的序列，差别只是状态名不同而已。但是当输入串不符合文法时，LALR 可能比 LR 多做了一些不必要的归约，延迟了发现错误的步骤，但 LALR 和 LR 均能指出输入串的出错位置。

任何 LR（K）文法都是无二义性文法，任何二义性文法都不是 LR（K）文法。对于 LR（K）文法，满足：LR（0）包含于 SLR（1）包含于 LALR（1）包含于 LR（1）。此外，对所有 K 都有 LR（K）包含于 LR（K+1）。给定文法 G 和某个固定的 K，G 是否是 LR（K）文法是可以判定的。给定文法 G，是否存在一个 K 使得 G 是一个 LR（K）文法的问题是不可判定的。

4.4 语法分析程序的自动生成

随着许多新语言的出现及计算机技术的发展，人们对开发编译程序的软件工具的需求大大增长。以 LR 文法及其分析方法为基础，从上个世纪 70 年代开始出现了自动生成语法分析程序的工具。YACC（Yet Another Compiler-Compiler）就是其中最杰出的代表。该系统是美国贝尔实验室的软件产品，是 UNIX 操作系统下的一个软件开发工具，它是由 S·C·Johnson 设计的。目前已经移植到多种操作系统上，并已成功开发了许多编译系统，深受软件工作者的喜爱。

YACC 是一个程序（软件工具），它接受 LALR 文法类，用户提供关于语法分析器的规格说明，基于 LALR 语法分析的原理，自动构造出一个语法分析器；并且能根据规格说明中给出的语义动作完成规定的语法制导翻译。

YACC 的工作过程如下：

（1）首先需要准备一个包含编译器性能规格的 YACC 说明文件，用 translate.y 表示。

（2）在 UNIX 环境下用命令 yacc translate.y，使用 LALR 分析法可把文件 translate.y 翻译成 C 语言程序。我们用 y.tab.c 表示，程序 y.tab.c 包含用 C 语言写的 LALR 分析器和用户准备的 C 语言程序。

（3）为了使 LALR 分析表少占空间，使用了合并技术压缩分析表规模，用命令 cc y.tab.c-ly 对 y.tab.c 进行编译，其中 ly 表示使用 LR 分析程序的库，编译结果得到目标程序 a.out。它完成了 YACC 程序指定的翻译。

（4）如果需要其它过程，它们可以和 y.tab.c 一起编译或装入，就和使用任何 C 程序一样。

ANTLR 集成了 YACC，在 ANTLR 中写一个语法文件和编写一个软件很相似，不同之处在于我们这里用到的是规则，而软件则是函数与过程。（需要记住的是，ANTLR 语法为每个规则生成一个函数。）但是，在关注规则内部之前，需要讨论的是整体语法结构以及如何形成一个初步语法框架。

语法文件包括语法的标题名称和可以相互调用的一组规则。

```
grammar MyG;
rule1 : <<stuff>>;
rule2 : <<morestuff>>;
…
```

就像写软件，我们必须弄清楚我们需要哪些规则，<<stuff>>是什么，哪些规则是开始规则（类似于 main()）。

正确的语法设计反映了功能分解或自顶向下设计的编程方法。这意味着我们的工作从粗到细，识别语言结构和编码语法规则。所以，第一个任务是找到粗略的语言结构的名称，这将成为我们的开始规则。在英语中，我们可以使用“sentence”。对于一个 XML 文件，我们可以使用“document”。在一个 Java 文件中，我们可以使用“compilationUnit”。

设计开始规则的内容是用来描述输入英文伪代码的整体格式的，有点像我们在编写软件时做的。例如，“a comma-separated-value (CSV) file is a sequence of rows terminated by newlines.”。对于这一句，“is a”左边的关键词“file”是规则的名称，“is a”右边的所有内容都是 <<stuff>> （规则的定义）。

```
file : <<sequence of rows that are terminated by newlines>>;
```

右边的名词通常引用 Tokens（标记）或 yet-to-be-defined 规则。这些标记是一些基本元素。正如单词在一个英语句子中是原子元素，标记在一个解析器语法中也是作为原子存在的。然而，规则的引用需要参考其他语言结构，需要分解成详细内容，就例如 row。

我们可以认为 row 是一个由逗号分隔的一序列的字段。这时，一个字段是一

个数字或字符串。伪代码如下所示：

file : <<sequence of rows that are terminated by newlines>>;

row : <<sequence of fields separated by commas>>;

field : <<number or string>>;

4.5 应用案例

智能英语语法分析软件是一个典型的词法分析和语法分析综合应用的软件，通过识别单词和英语句子结构，分析一条英文句子是否是合法的句子。

以英语中的简单句“Tom saw a mouse.”为例，词法分析先将句子中有意义的单词识别出来。在程序中，句子以字符流的形式顺序输入，词法分析器通过正规表达式(a|b|c|…|z)(a|b|c|…|z)*定义每个单词的结构，再根据定义的词法规则识别单词。与“C--”语言词法分析程序不同，识别英语单词不存在自定义的标识符，每一个单词都可以在英语词汇表中查询到，即关键字。若该单词在词汇表中不存在，则程序报告单词拼写错误。英语词汇表中包含单词及其词性，词法分析器除了识别单词之外，还会完成单词的分类如<名词>、<动词>、<冠词>等。上述例句通过词法分析后识别出的单词符号如图 4.11 和图 4.12 所示：

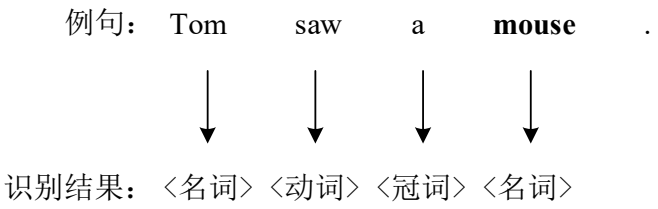

图 4.11 例句词法分析正确结果

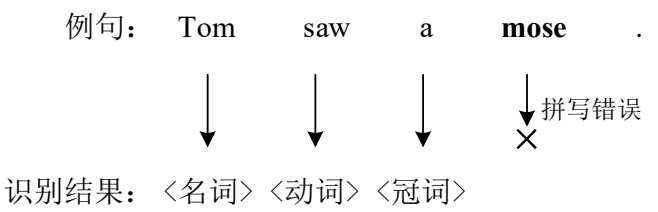

图 4.12 例句词法分析报错

词法分析完成后，语法分析根据英语的语法规则从单词流中识别出各个成分，例如<主语>、<宾语>、<表语>等，并检查各种语法成分在语法结构中是否正确。为了便于理解，我们以英语中基础的语法结构“主语+谓语+宾语”为例来说明语法分析过程。该结构定义的文法如下：

<句子>::=<主语><谓语><宾语>

<主语>::=<冠词><名词> | <名词> | ε

<谓语>::=<动词>

<宾语>::=<冠词><名词> | <名词> | ε

我们将<句子>简化为 E，<主语>简化为 S，<谓语>简化为 V，<宾语>简化为 O，<动词>简化为 v，<冠词>简化为 a，<名词>简化为 n，化简后的文法 G[E]如下：

E∷=SVO

S∷=an | n | ε

V∷=v

O∷=an | n | ε

上述例句“Tom saw a mouse.”经过词法分析得到“<名词><动词><冠词><名词>”，化简后的结果为“nvan”。程序将词法分析的结果顺序输入语法分析器中，并利用 LL（1）分析法分析句子是否符合文法 G[E]。构建上述文法的分析表，相应的分析表如表 4.15 所示：

表 4.15 文法 G[E]的 LL（1）分析表

	a	**n**	**v**	**#**
E	E∷=SVO	E∷=SVO	E∷=SVO	
S	S∷=an	S∷=n	S∷=ε	
V			V∷=v	
O	O∷=an	O∷=n		O∷=ε

其分析过程如表 4.16 所示：

表 4.16 分析例句“Tom saw a mouse.”

步骤	**分析栈**	**余留输入串**	**所用产生式**
(1)	#E	nvan#	E∷=SVO
(2)	#OVS	nvan#	S∷=n
(3)	#OVn	nvan#	
(4)	#OV	van#	V∷=v
(5)	#Ov	van#	
(6)	#O	an#	O∷=an
(7)	#na	an#	
(8)	#n	n#	
(9)	#	#	成功

由分析过程可以看出，例句“Tom saw a mouse.”是符合“主语+谓语+宾语”的语法结构的，所以该句子是合法的句子。上述过程是简化后的语法分析，忽略了英语中大量复杂的语法结构与句式，而智能英语语法分析软件将通过设计完整高效的文法规则实现英语句子的语法分析。

4.6 本章小结

本章主要针对编译过程中的语法分析方法及涉及到的相关概念（例如回溯、左递归、First 和 Follow 集合、活前缀等）进行了详细的阐述。目前，主流的语法分析方法分为两大类：一是自顶向下的语法分析方法，包括递归子程序法和 LL（1）分析法等；二是自底向上的语法分析方法，包括优先分析法和 LR 系列分析法等。本章重点讨论了 LL（1）分析法以及 LR（0）、SLR（1）、LR（1）和 LALR（1）的工作原理和语法分析过程，并通过具体的案例进行了应用分析。

习题

1. 试分别消除下列文法的直接或者间接左递归

（1）G[E]：

E∷＝T｜EAT

T∷＝F｜TMF

F∷＝(E)｜i

A∷＝+｜-

M∷＝*｜/

（2）G[S]：

S∷＝SA｜Ab｜b｜c

A∷＝Bc｜a

B∷＝Sb｜b

（3）G[Z]：

Z∷＝V_1

V_1∷＝V_2｜V_1iV_2

V_2∷＝V_3｜V_2+V_3

V_3∷＝)V_1*｜(

（4）G[S]：

S∷＝Qc｜c

Q∷＝Rb｜b

R∷＝Sa｜a

（5）G[Z]：

Z∷＝AZ｜b

A∷＝ZA｜a

2. 对下面文法 G[E]：

E∷＝TE′

E′∷＝+E｜ε

T∷＝FT′

T′∷＝T｜ε

F∷＝PF′

F′∷＝*F′｜ε

P∷＝(E)｜a｜b｜∧

（1）计算这个文法的每个非终结符号的 FOLLOW 集和所有规则右部的 FIRST 集。

（2）证明这个文法是 LL（1）文法。

（3）构造它的 LL（1）分析表并分析符号串 a*b+b。

3. 对下面文法，构造每个非终结符号相应的 FIRST 集和 FOLLOW 集

（1）S∷＝aAd

A∷＝BC

B∷＝b｜ε

C∷＝c｜ε

（2）A∷＝BCc｜gDB

B∷＝ε｜bCDE

C∷＝DaB｜ca

D∷＝ε｜dD

E∷＝gAf｜c

4. 给定文法：

S∷＝a｜∧｜(T)

T∷＝T,S｜S

（1）改写这个文法，消除左递归。

（2）改写后的文法是否是 LL（1）文法？若是，构造它的 LL（1）分析表。

（3）写出该文法所描述的语言是什么？

5. 设已给文法 G[S]：

S∷＝SaB｜bB

A∷＝Sa

B∷＝Ac

（1）将此文法改写为 LL（1）文法。

（2）对每一规则右部各候选式，构造 FIRST 集。

（3）对每一非终结符构造 FOLLOW 集。

（4）构造 LL（1）分析表。

6. 设有文法 G[Z]：

Z∷＝A｜B

A∷＝aAb｜c

B∷＝aBb｜d

（1）试构造能识别此 LR（0）文法全部活前缀的 DFA。

（2）试构造 LR（0）分析表。

（3）试分析符号串 aacbb 是否为此文法的句子。

7. 考虑文法：

S∷＝AS｜b

A∷＝SA｜a

（1）构造该文法 LR（0）的 DFA；

（2）判定其是否是 LR（0）文法？是否是 SLR（1）文法？若是，则构造 SLR（1）分析表。

（3）试分析符号串 bab 是否是该文法的句子。

8. 给定文法：

E∷＝EE+｜EE*｜a

（1）构造它的 LR（0）项目集规范族。

（2）它是 SLR（1）文法吗？若是，构造它的 SLR（1）分析表。

（3）它是 LR（1）文法吗？若是，构造它的 LR（1）分析表。

（4）它是 LALR（1）吗？若是，构造它的 LALR（1）分析表。

9. 给定文法 G[Z]：

Z∷＝AA

A∷＝Ab｜b

（1）试构造能识别此 LR（1）文法全部活前缀的 DFA。

（2）试构造 LR（1）分析表。

（3）请问该文法是否为 LR(1)文法？

10. 给出文法 G[E]：

E∷＝E+T｜T

T∷＝TF｜F

F∷＝F*｜(E)｜a｜b｜∧

构造该文法的 LR（1）项目集。

11. 证明文法 G[S]不是 LR（1）文法。

S∷＝1S0｜0S1｜10｜01

12. 将文法 G[S]

S∷＝E

E∷＝E+T｜T

T∷＝T*F｜F

F∷＝(E)｜x｜y

（1）构造其 SLR（1）分析表。

（2）试分析符号串 x+y*x 是否为此文法的句子。

13. 给出如下文法：

$G_1[S]$：

S::=aSbS | aS | c

$G_2[S]$：

S::=aAa | aBb

A::=x

B::=x

$G_3[S]$：

S::=aAa | aBb | bAb

A::=x

B::=x

$G_4[S]$：

S::=aAa | aBb | bAb | bBa

A::=x

B::=x

（1）证明二义性文法 $G_1[S]$不是 LR（0）文法。

（2）证明 $G_2[S]$是 SLR（1）文法但不是 LR（0）文法。

（3）证明 $G_3[S]$是 LR（1）文法但不是 SLR（1）文法。

（4）证明 $G_4[S]$是 LR（1）文法但不是 LALR（1）文法。