第5章 自底向上的语法分析

自底向上的分析方法的主要思想是从待分析的输入单词串出发，根据文法规则逐步进行归约，直到归约到文法的开始符号为止。在编译器的语法分析中，存在多种自底向上的分析法，这些方法归根结底都遵循相同的原理，即“移进-归约”的思想。

5.1 自底向上的语法分析方法

5.1.1 “移进-归约”分析

自底向上的语法分析方法通常采用“移进-归约”的分析过程处理输入串。该过程需要预先设置一个存放文法符号的符号栈和存放输入串的缓冲区，初始化时，栈底存放标记符号#，输入缓冲区也以#作为结尾标记。在分析过程中，自左至右扫描输入串，把输入字符逐一移入符号栈内，当栈顶出现某个产生式的右部时就进行归约，即把栈顶的可归约串弹出，产生式的左部符号入栈，然后继续检查栈顶是否又出现了归约串，若出现，则继续归约，否则从输入缓冲区中继续移进下一个符号，这一过程直到整个输入串扫描完毕。此时若栈顶为文法的开始符号，说明整个输入串最终归约成文法开始符号，则可以确认所分析的符号串是文法的句子，否则，符号串不合法，要报告语法错误。下面通过一个例子说明“移进-归约”分析过程。

【例 5-1】 设有文法 $G[S]$：

$S \rightarrow aAcBe$

$A \rightarrow b \mid Ab$

$B \rightarrow d$

对输入符号串 *abbcde* 进行“移进-归约”分析，即判定它是否为该文法的句子。

用“移进-归约”分析方法对输入符号串 *abbcde* 的分析过程如表 5-1 所示。符号#作为输入符号串 *abbcde* 的左右界符，即输入符号串的开始和结束标志。初始化时，将左界符#首先放入符号栈底，然后从第一个符号开始分析。

表 5-1 输入符号串 *abbcde* 的分析过程

步骤	符号栈	输入串	分析动作
初始化	#	*abbcde*#	移进
(1)	#*a*	*bbcde*#	移进
(2)	#*ab*	*bcde*#	用 $A \rightarrow b$ 归约
(3)	#*aA*	*bcde*#	移进
(4)	#*aAb*	*cde*#	用 $A \rightarrow Ab$ 归约
(5)	#*aA*	*cde*#	移进
(6)	#*aAc*	*de*#	移进
(7)	#*aAcd*	*e*#	用 $B \rightarrow d$ 归约
(8)	#*aAcB*	*e*#	移进
(9)	#*aAcBe*	#	用 $S \rightarrow aAcBe$ 归约
(10)	#*S*	#	接受(分析成功)

注：表中第 2 列“符号栈”左端表示栈底，右端表示栈顶；第 3 列“输入串”左端表示当前要读入的输入串的字符，右端是输入串的结束。

从上述分析过程可知，“移进-归约”分析过程主要采取的分析动作包括移进、归约、接受或报错，上例因为分析的符号串没有语法错误，所以没有出现“报错”。“报错”通常调用出错处理子程序进行处理。进一步观察表 5-1 给出的归约顺序可知，归约的逆过程恰好是从文法开始符号出发，进行最右推导(规范推导)的过程。例如，符号串 *abbcde* 的最右推导序列为

$$S \Rightarrow a\,AcBe \Rightarrow a\,Acde \Rightarrow a\,Abcde \Rightarrow a\,bbcde$$

每次归约时呈现在栈顶的要归约的子串称为可归约串。归约时，第一步把最左边的可归约串 b 归约为 A，第二步把最左边的可归约串 Ab 归约为 A，第三步把最左边的可归约串 d 归约为 B，最后一步仍然是把最左边的可归约串 $aAcBe$ 归约为 S。因此最右推导与最左归约对应，互为逆过程。

“移进-归约”分析方法之所以采用最左归约是和扫描顺序对应的，因为对输入字符串自左至右逐一扫描，所以很自然地总是对最左边的可归约串进行归约。

5.1.2 规范归约与句柄

上述“移进-归约”分析实施的是规范归约(最左归约)，每步归约的是最左可归约串，显然，如何定义和确定它是自底向上分析的关键问题。下面介绍一些相关的概念和定义。

定义 5-1 短语

已知文法 $G[S]$，S 是文法 G 的开始符号，$\alpha\beta\delta$ 是文法 G 的一个句型，即 $S \overset{*}{\Rightarrow} \alpha A\delta$，若 $A \overset{+}{\Rightarrow} \beta$，则 β 是句型 $\alpha\beta\delta$ 相对于非终结符 A 的短语。

定义 5-2 直接短语

已知文法 $G[S]$，S 是文法 G 的开始符号，$\alpha\beta\delta$ 是文法 G 的一个句型，若有 $S \overset{*}{\Rightarrow} \alpha A\delta$ 且 $A \Rightarrow \beta$，则 β 是句型 $\alpha\beta\delta$ 相对于非终结符 A 的直接短语。

定义 5-3 句柄

一个句型的最左直接短语称为该句型的句柄，句柄是一个重要的概念。“移进-归约”分

析中的“可归约串”就是当前句型的句柄。如果文法是无二义性的，则规范句型的最右推导是唯一的，也就是说每步归约至多存在一个句柄。

下面通过例子说明短语、直接短语和句柄的概念以及句柄在“移进-归约”分析中的作用。

【例 5-2】 设有文法 $G[S]$：

(1) $S \rightarrow aAcBe$

(2) $A \rightarrow b \mid Ab$

(3) $B \rightarrow d$

符号串 $abbcde$ 的最右推导序列为 $S \Rightarrow aAcBe \Rightarrow aAcde \Rightarrow aAbcde \Rightarrow abbcde$，分析每一步推导过程中各句型中的短语、直接短语和句柄。

下面根据短语、直接短语和句柄的定义进行分析，句型 $aAcBe$ 中，$aAcBe$ 是相对 S 的短语、直接短语，也是句柄，这是因为 $S \Rightarrow S$，$S \Rightarrow aAcBe$，且 $aAcBe$ 是唯一直接短语，因此也是句柄。句型 $aAcde$ 中，d 是相对于 B 的直接短语，$aAcde$ 是相对于 S 的短语。同理可以分析剩余的两个句型。从定义出发寻找每个句型中的短语、直接短语和句柄不直观，容易遗漏，下面介绍一种简单直观的方法。

在自顶向下分析中，可通过分析树直观地了解从文法开始符号 S（树根）到输入串（叶子结点）的推导过程。同样，在自底向上分析中，从输入串（叶子结点）出发，可通过对分析树逐层修剪直到根结点来了解归约过程，从而加深对规范归约、句柄等概念的理解。

下面对例 5-2 进行分析，句子 $abbcde$ 的自底向上归约过程的分析树如图 5-1 所示。

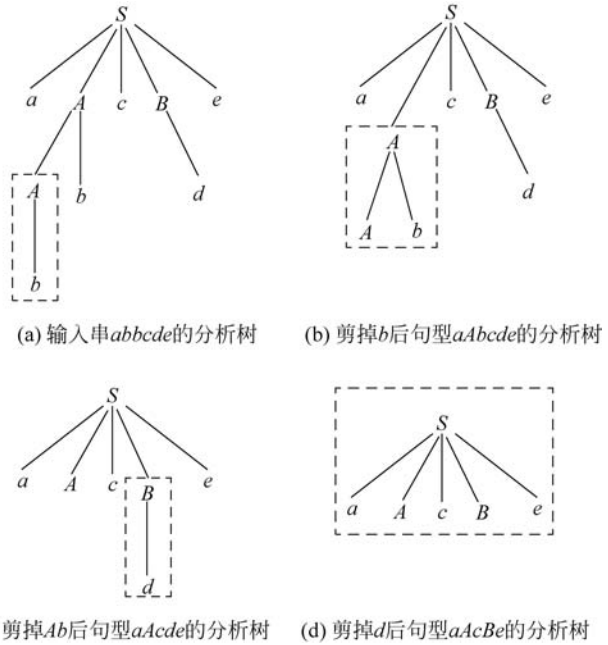

图 5-1　句子 $abbcde$ 的自底向上分析分析树的修剪过程

句子 $abbcde$ 的最左子树是图 5-1(a)中方框勾出的部分。这样子树的叶子结点 b 即为句子 $abbcde$ 的句柄，将其剪掉（归约）后如图 5-1(b)所示。图 5-1(b)中方框勾出的部分为句型 $aAbcde$ 的分析树中的最左子树，其叶子结点 Ab 为句柄，将其剪掉如图 5-1(c)所示。此

时,句型 *aAcde* 的句柄为 *d*,将其剪掉得到图 5-1(d)所示的分析树。此时,句型 *aAcBe* 的最左子树即为它自身的分析树,它的叶子结点 *aAcBe* 为句型 *aAcBe* 的句柄,将其剪掉,只剩下根结点 *S*,即文法的开始符号,至此,完成了输入符号串 *abbcde* 的分析。

仍然对例 5-2 进行分析,可以看出,一个句型的分析树中最左边只有父子两代的子树的所有叶子的自左至右排列形成该句型的句柄。而句型的直接短语是仅有父子两代的子树的所有叶子结点自左至右排列起来所形成的符号串。句型的短语是分析树中子树的所有叶子结点从左到右的排列,形成相对于子树根的短语。

下面对例 5-2 每步归约对应的句型进行分析,通过分析树找出短语、直接短语和句柄。各句型的分析树中所有子树用方框标识,如图 5-2 所示。句子 *abbcde* 的子树是图 5-2(a)中方框勾出的部分,共 4 棵子树,叶子结点 *b* 是相对于子树根 *A* 的短语和直接短语,*bb* 是相对于子树根 *A* 的短语,因为从 *A* 出发经过两步而不是直接推导出 *bb*,*d* 是相对于子树根 *B* 的短语和直接短语。句子 *abbcde* 的分析树是其自身的子树,因此,*abbcde* 是相对于子树根 *S* 的短语,*b* 是分析树中最左那棵只有父子两代的子树的所有叶子自左至右排列形成的符号串,即句柄。句型 *aAbcde* 的分析树中包含 3 棵子树,其中 *Ab* 是相对于 *A* 的短语、直接短语,也是句柄。*d* 是相对于 *B* 的短语和直接短语。*aAbcde* 是相对于子树根 *S* 的短语,如图 5-2(b)所示。句型 *aAcde* 包含两棵分析树,*d* 和 *aAcde* 是短语,*d* 也是直接短语和句柄,如图 5-2(c)所示。句型 *aAcBe* 的短语、直接短语和句柄均为 *aAcBe*,如图 5-2(d)所示。

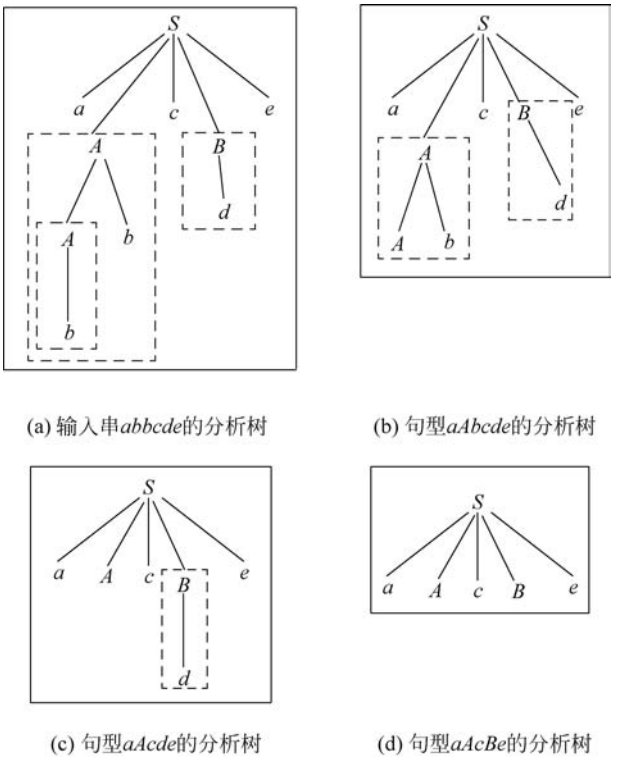

图 5-2 各句型的分析树及其子树

从上述例子可以看出,“移进-归约”分析方法的关键问题是如何确定可归约串。一是判断栈顶字符串的可归约性,即判断处在栈顶的子串是否是可归约串;二是确定使用文法中

的哪一个规则进行归约，因为一个可归约串可以归约到多个不同的非终结符，即可归约串是不同的产生式的候选式。依照寻找可归约串的策略不同形成了不同的自底向上分析方法。

综上所述，各种自底向上分析方法的共同点是，都采用“移进-归约”分析思想，不同点是确定可归约串的方法不同。下面将介绍几种典型的自底向上的分析方法。

5.2　LR 分析法

LR(k)分析法简称为 LR 分析法，1965 年由 D. Knuth 提出，是目前编译程序的语法分析中最常用且有效的自底向上的分析方法，这里的 L 指自左至右扫描输入符号串，R 指分析过程是最右推导的逆过程。k 表示根据分析栈当前已移进和归约出的全部文法符号，再向前查看 k($k\geqslant 0$)个输入符号，就可以确定分析器的动作是移进还是归约。LR 分析的过程是规范推导的逆过程，因此是一种规范归约过程。采用 LR 方法构造的语法分析程序统称为 LR 分析器。

自顶向下的 LL(1)分析对文法有较强的限制，而 LR 分析法对于文法的限制则少得多，因此也更通用，适用于绝大多数上下文无关语言分析，理论上也比较完善，另外这种方法还具有分析速度快、报错准确等优点。其缺点是对于高级程序语言构造一个 LR 分析器工作量相当大，手工实现困难。因此，需要借助贝尔实验室最早推出的语法分析自动生成器 YACC 辅助构造来自动生成。

下面重点介绍 LR(k)当 $k\leqslant 1$ 时分析器的基本构造原理和方法，其中 LR(0)分析器是最简单的一种 LR 分析法，分析过程中不需要向后看任何符号，但是它对文法的限制较大，对绝大多数高级程序语言不适用，其他 LR 分析法都是在 LR(0)的基础上进行完善的，特别地，当 $k=1$ 时就能够满足绝大多数高级程序语言语法分析的需要。本节重点讨论 LR(0)、SLR(1)、LR(1)和 LALR(1)这 4 种分析器的构造。

1. LR 分析器的逻辑结构和工作过程

在逻辑上，LR 分析器的结构如图 5-3 所示。它有一个输入缓冲区，存放 LR 分析器处理的字符串，有一个下推分析栈，以及一个 LR 分析总控程序和 LR 分析表。

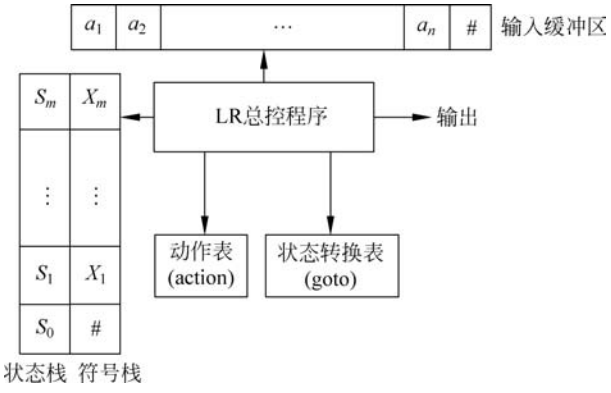

图 5-3　LR 分析器逻辑结构示意图

(1) 分析栈。包括文法符号栈 $X[i]$和相应的状态栈 $S[i]$两部分(状态是指能识别活前缀的自动机状态,后面会详细说明),LR 分析器通过判断状态栈的栈顶元素和输入符号查询分析表确定下一步分析动作,对符号栈进行相应更新。文法符号栈存放在分析过程中移进或归约的文法符号,类似"移进-归约"分析中符号栈的作用。状态栈的栈顶状态概括了栈中位于下边的全部信息,即无须关心状态栈中从开始状态到当前栈顶状态之间经历的一系列状态,仅用当前栈顶状态和输入符号就可以决定下一步的具体工作。换句话说,栈顶状态刻画了分析过程的"历史"情况和"展望"信息。初始化时,符号栈压入初始状态,文法符号栈压入输入串的左界符 #。

(2) 总控程序。LR 分析器在总控程序的控制下自左至右扫描输入串,根据当前分析栈顶所存放的文法符号状态及当前扫描读入的符号,查询 LR 分析表完成分析动作。

(3) 分析表。分析表是 LR 分析器的核心,它与具体文法有关,包括动作表(action)和状态转换表(goto)两部分,均使用二维数组表示,为总控程序提供决策依据。两个表的结构如表 5-2 和表 5-3 所示。

表 5-2 LR 分析表的动作表

状态	a_1	a_2	…	a_n
S_0	action$[S_0, a_1]$	action$[S_0, a_2]$	…	action$[S_0, a_n]$
S_1	action$[S_1, a_1]$	action$[S_1, a_2]$	…	action$[S_1, a_n]$
⋮	⋮	⋮	⋮	⋮
S_n	action$[S_n, a_1]$	action$[S_n, a_2]$	…	action$[S_n, a_n]$

表 5-3 LR 分析表的状态转换表

状态	X_1	X_2	…	X_n
S_0	goto$[S_0, X_1]$	goto$[S_0, X_2]$	…	goto$[S_0, X_n]$
S_1	goto$[S_1, X_1]$	goto$[S_1, X_2]$	…	goto$[S_1, X_n]$
⋮	⋮	⋮	⋮	⋮
S_n	goto$[S_n, X_1]$	goto$[S_n, X_2]$	…	goto$[S_n, X_n]$

其中,状态转换表的元素 goto $[S_i, X_i]$是一个状态,它表示根据状态栈栈顶状态 S_i 和文法符号栈栈顶符号 X_i($X_i \in V_N$)转移到下一个状态。例如,若有 goto $[S_i, X_i] = S_k$,表示当前栈顶状态为 S_i 和符号为 X_i 时转移到的下一个状态为 S_k。

动作表的元素 action$[S_i, a_i]$表示栈顶当前状态为 S_i,和当前输入符号为 a_i($a_i \in V_T$)时完成的分析动作。具体的分析动作可分为 4 类:移进、归约、接受或出错。

(1) 移进。

当 action$[S_i, a_i] = S_j$ 时(这里 S_i 和 S_j 中的 i、j 为状态编号,S 表示 shift,移进),即当前栈顶状态为 S_i,当前缓冲区输入符号为 a_i 时,将 S_j 移进状态栈,a_i 移进文法符号栈。"移进"分析动作表示句柄尚未在分析栈顶形成,正期待继续移进符号以形成句柄。

(2) 归约。

当 action$[S_i, a_i] = r_j$ 时(r_j 指按文法的第 j 个产生式进行归约,r 表示 reduce,归约),即表示当前栈顶状态为 S_i,当前输入符号为 a_i 时,用第 j 个产生式归约。设第 j 个产生式为 $A \rightarrow \beta$,字符串 β 的长度为 r,分析动作为"归约"时,表明当前分析栈的栈顶已形成

当前句型的句柄β，要立即进行归约。不仅要把文法符号栈栈顶包含r个符号的句柄β弹出，同时，状态栈也要弹出对应的r个状态，并把A移入文法符号栈，同时查询状态转换表goto[S_m，A]=S_j（S_m表示状态栈弹出r个状态后的栈顶状态），把S_j移进状态栈，形成当前的新状态。

（3）接受。

当action[S_i，a_i]='acc'时('acc'表示accept，接受），表示当前输入串已经归约到文法的开始符号，即文法符号栈栈顶为开始符号S，输入缓冲区扫描到最后的结尾标记#时，分析成功，终止分析工作。

（4）报错。

当action[S_i，a_i]='error'（也可用空白表示）时，分析动作出错，表示当前输入串有语法错误，调用相应的出错处理程序。

算法5-1给出了LR分析过程的具体描述。

算法5-1　LR分析算法

输入：LR分析表和输入符号串

输出：若输入符号串合法，输出自底向上构造的分析树，否则报错。

算法：

（1）初始化，将初始状态S_0压入状态栈，输入串左界符#压入文法符号栈。

（2）循环执行下面的步骤，直至分析成功或报错。

（3）根据当前状态栈栈顶状态S_i以及当前输入符号a_i查询动作表。

若action[S_i，a_i]=S_j，将S_j移进状态栈，a_i移进文法符号栈。

若action[S_i，a_i]=r_j，用第j个产生式归约，归约过程如前面所述。

若action[S_i，a_i]='acc'，分析成功。若action[S_i，a_i]='error'，进行出错处理。

下面通过具体例子介绍LR分析器的工作过程。

【例5-3】　设有文法G[S]：

（1）$S\rightarrow L,S$

（2）$S\rightarrow L$

（3）$L\rightarrow x$

（4）$L\rightarrow y$

已知文法G[S]的LR分析表如表5-4所示。以输入串“x,y,x”为例，给出LR分析器的分析过程。

表5-4　文法G[S]的LR分析表

状态	action表				goto表	
	x	y	,	#	S	L
0	S_3	S_4			1	2
1				acc		
2			S_5	r_2		
3			r_3	r_3		
4			r_4	r_4		
5	S_3	S_4			6	2
6				r_1		

输入串“x,y,x”的 LR 分析过程如表 5-5 所示。

表 5-5 输入串“x,y,x”的 LR 分析过程

步骤	栈中状态	栈中符号	输入符号串	分 析 动 作
1	0	#	x,y,x#	S_3 移进“x”,状态转到 3(3 入栈)
2	03	#x	$,y,x$#	r_3(用第三个产生式 $L\rightarrow x$ 归约)
3	02	#L	$,y,x$#	S_5 移进“,”,状态转到 5(5 入栈)
4	025	#L,	y,x#	S_4 移进“y”,状态转到 4(4 入栈)
5	0254	#L,y	$,x$#	r_4(用第四个产生式 $L\rightarrow y$ 归约)
6	0252	#L,L	$,x$#	S_5 移进“,”,状态转到 5(5 入栈)
7	02525	#L,L,	x#	S_3 移进“x”,状态转到 3(3 入栈)
8	025253	#L,L,x	#	r_3(用第三个产生式 $L\rightarrow x$ 归约)
9	025252	#L,L,L	#	r_2(用第二个产生式 $S\rightarrow L$ 归约)
10	025256	#L,L,S	#	r_1(用第一个产生式 $S\rightarrow L\ ,S$ 归约)
11	0256	#L,S	#	r_1(用第一个产生式 $S\rightarrow L\ ,S$ 归约)
12	01	#S	#	acc

从表 5-5 所示的分析过程可看出,根据归约的顺序依次归约的句柄是“x”“y”“x”“L”“L,S”“L,S”。刚好构成了最右推导(规范推导)的逆过程,即上述分析过程每一行的符号栈的文法符号和输入缓冲区的所有符号加在一起,刚好构成文法的一个右句型,因此,LR 的分析过程是(最左归约)规范归约的过程。

对于例 5-3,本书预先给出了 LR 分析表,实际上,LR 分析法的关键恰恰是分析表的构造。对于不同的文法,LR 分析算法都是不变的,唯一的区别是 LR 分析表构造的方法不同。

2. LR 文法

定义 5-4 LR 文法

一个文法 G,若构造出的 LR 分析表不含多重定义,即它的每一入口的动作都是唯一确定的,则文法 G 称为 LR 文法。

定义 5-5 LR(k)文法

一个文法 G,若使用 LR 方法进行分析,分析过程中的每一步最多向前查看 k 个输入符号,就能唯一确定当前分析动作,即可以构造出无多重定义的 LR(k)分析表,则称文法 G 为 LR(k)文法。

LR(k)是从分析过程需要利用输入字符数目的角度来定义的。对于当前流行的大多数程序设计语言来说,$k=1$ 就可以满足它们 LR 分析的需要。根据定义不难推出,任何 LR(k)文法都是无二义性的文法,任何二义性文法都不是 LR(k)文法。对于二义性文法而言,可通过对二义性文法进行修改或者施加一些限制来克服分析表中出现的冲突动作,从而使 LR 分析适用于这些二义性文法。

当 $k=0$ 时对应的 LR(0)分析是最简单的一种 LR 分析法,仅适用于 LR(0)文法。LR(0)分析仅根据状态栈当前状态即可决定当前符号栈是否已构成句柄,而不需要查看下一个输入符号,即只根据状态就可以确定是否归约。值得注意的是,这里的向前看一个符号的作用

主要是确定是否归约，因为移进动作总是需要向前看一个符号，也就是说，每个 LR(k)分析法确定移进动作时肯定需要向前看一个符号，但是确定是否归约时就有区别了，LR(0)不需要向前看任何符号，SLR(1)找出所有跟在句柄后可能的第一个终结符，LR(1)找出规范句型中跟在句柄之后可能的第一个终结符。

各种 LR 分析的实现思想和分析过程是相同的，区别仅在于 LR 分析表中确定归约动作的方法不同，下面从 LR(0)分析作为出发点，逐步展开对 LR 分析的讨论。首先引入一些重要的概念、术语和定义。

定义 5-6　规范句型的活前缀

规范句型的一个前缀若不含句柄之后的任何符号，则称为该句型的一个活前缀。活前缀的形式化定义为：设有文法 $G[S]$，$S\Rightarrow \alpha Aw\Rightarrow \alpha\beta w$ 是文法 G 的一个规范推导(α，$\beta\in V^{+}$，$w\in V_{\mathrm{T}}^{*}$，$A\in V_{\mathrm{N}}$)，若 γ 是符号串 $\alpha\beta$ 的前缀，则称 γ 是 G 的一个活前缀，如果符号串 γ 是含句柄的活前缀，则称 γ 是 G 的可归前缀(最长的活前缀)。

需要注意的是，活前缀所在句型一定是规范句型。从例 5-3 中对输入串“x，y，x”的分析过程可以看出，在分析的每一步，若将符号栈中的全部文法符号与扫描剩余的输入串连接起来，刚好构成语法的一个规范句型。符号栈中全部文法符号刚好是某一规范句型的活前缀。

例如，如表 5-6 所示，句子“x，y，x”分析的第 3 步，符号栈中的文法符号串是“L”，剩余字符串为“，y，x”，连接在一起刚好形成文法一个规范句型“L，y，x”。该句型的句柄为“y”，栈中符号串为“L”，则此句型的活前缀为“ε”“L”“L，”“L，y”，最长活前缀“L，y”恰好含有句柄“y”。但是在第 3 步还没有把“y”移入符号栈，因此，需要把输入符号串的字符继续逐一移入符号栈，在第 5 步句柄“y”出现在栈顶时，把“y”归约为“L”。从这个例子可以看出，符号栈中的符号串是当前规范句型的活前缀，如果还没有包括句柄，则继续移入符号，当把句柄完全移入后产生了最长的活前缀，然后紧接着进行归约。因此，我们最关心的是最长活前缀，因为最长活前缀刚好包含句柄。不难看出，活前缀的特点是它不含句柄之右的任何符号。

表 5-6　规范句型的活前缀

步骤	栈中状态	栈中符号	输入符号串	活前缀	最长活前缀
1	0	#	x，y，x #	ε　x	x
2	03	#x	，y，x #	ε　x	x
3	02	#L	，y，x #	ε　L　L，　L，y	L，y
4	025	#L，	y，x #	ε　L　L，　L，y	L，y
5	0254	#L，y	，x #	ε　L　L，　L，y	L，y
6	0252	#L，L	，x #	ε　L　L，　L，L　L，L，　L，L，x	L，L，x
7	02525	#L，L，	x #	ε　L　L，　L，L　L，L，　L，L，x	L，L，x
8	025253	#L，L，x	#	ε　L　L，　L，L　L，L，　L，L，x	L，L，x
9	025252	#L，L，L	#	ε　L　L，　L，L　L，L，　L，L，L	L，L，L
10	025256	#L，L，S	#	ε　L　L，　L，L　L，L，　L，L，S	L，L，S
11	0256	#L，S	#	ε　L　L，　L，S	L，S
12	01	#S	#		

上例也说明,一个 LR 分析器的“移进-归约”工作过程其实是符号栈逐步产生文法 G 的规范句型的活前缀的过程。当生成最长活前缀时,此时分析栈顶部形成句柄,可立即归约。

这就为确定分析动作提供了依据,若能找出文法所有的活前缀,并且确定当前分析句型中最长的活前缀,那么就可以在活前缀生长到最长时归约,在其余状态时移进或报错。

5.2.1 LR(0)

如前所述,找出文法所有的活前缀并确定最长的活前缀是确定分析动作的关键,为此,首先分析活前缀与句柄的关系,然后提出识别文法所有活前缀的方法。

在一个规范句型的活前缀中,决不会含有句柄右边的任何符号。因此,活前缀与句柄间的关系有 3 种情况:

(1) 活前缀中不包含句柄的任何符号。

(2) 活前缀中只含有句柄的一部分符号。

(3) 活前缀中已含有句柄的全部符号,即最长活前缀,通常也称为可归前缀。

第一种情况表明,句柄中的文法符号没有一个出现在栈顶,期望从输入符号串中移进某一产生式 $A\rightarrow\beta$ 中的 β 符号串。第二种情况意味着形如产生式 $A\rightarrow\beta_1\beta_2$ 的子串 β_1 已出现在栈顶,下一步准备从输入串中移进由 β_2 推出的符号串。第三种情况表明,某一产生式 $A\rightarrow\beta$ 的右部 β 已完全出现在栈顶,下一步分析动作应是用该产生式进行归约。为了刻画分析过程中文法的每一个产生式的右部符号已有多大一部分被识别(出现在栈顶),很自然地可以在产生式的右部加上一个圆点指示位置。

(1) $A\rightarrow\ \cdot\beta$ 表示当前句型活前缀中不包含句柄 β 的任何符号。

(2) $A\rightarrow\beta_1\cdot\beta_2$ 表示当前句型活前缀中只含有句柄的一部分符号 β_1。

(3) $A\rightarrow\beta\cdot$ 表示当前句型活前缀中已含有句柄的全部符号。

因此,需要定义 LR(0)项目来描述这些情况。

定义 5-7 LR(0)项目

为了描述分析状态,在文法 G 的每个产生式的右部符号串的任何位置上添加一个圆点所构成的每个产生式称为 LR(0)项目。

特别地,若产生式形为 $A\rightarrow\varepsilon$,则其 LR(0)项目为 $A\rightarrow\cdot$ 。

【例 5-4】 已知文法 $G(S)$:

$S\rightarrow aAcBe$

$A\rightarrow Ab\mid b$

$B\rightarrow d$

找出其所有 LR(0)项目。

$G(S)$的 LR(0)项目有:

$S\rightarrow\cdot aAcBe$, $S\rightarrow a\cdot AcBe$, $S\rightarrow aA\cdot cBe$, $S\rightarrow aAc\cdot Be$, $S\rightarrow aAcB\cdot e$, $S\rightarrow aAcBe\cdot$,

$A\rightarrow\cdot Ab$, $A\rightarrow A\cdot b$, $A\rightarrow Ab\cdot$, $A\rightarrow\cdot b$, $A\rightarrow b\cdot$,

$B\rightarrow\cdot d$, $B\rightarrow d\cdot$

从直观意义上讲,一个 LR(0)项目对应一种分析状态,即在分析过程中分析到了圆点所在的位置,圆点可看成是符号栈栈顶与输入串当前输入符号的分界点,圆点左边为已进入符号栈的部分,右边是当前需要分析的符号串。

根据 LR(0)项目的定义，圆点所在不同位置反映了分析栈的不同情况。因此根据不同的位置可以把 LR(0)项目分为 4 类，表示 4 种不同状态。

(1) 移进项目。

形式：$A\rightarrow\alpha\cdot a\beta(a\in V_T)$

这类 LR(0)项目表示 α 出现在符号栈中，活前缀包含句柄的部分符号，为构成可归前缀，需继续将圆点后的终结符号 a 移进符号栈。因此这种形式的 LR(0)项目称为移进项目，它对应的状态为移进状态。

(2) 待约项目。

形式：$A\rightarrow\alpha\cdot B\beta(B\in V_N)$

这类 LR(0)项目表示 α 出现在符号栈中，活前缀包含句柄的部分符号，为构成可归前缀，期待着把当前输入字符串中的相应内容先归约到 B。因此这种形式的 LR(0)项目称为待约项目，它对应的状态为待约状态。

(3) 归约项目。

形式：$A\rightarrow\alpha\cdot$

这类 LR(0)项目表示句柄 α 刚好出现在符号栈栈顶，活前缀刚好包含句柄，即当前符号栈中的符号串正好为可归前缀，应按 $A\rightarrow\alpha$ 进行归约。因此这种形式的 LR(0)项目称为归约项目，它对应的状态为归约状态。

(4) 接受项目。

对任何文法 G，在 G 中加进一个新的符号 S'和一个产生式 $S'\rightarrow S$，并以 S'代替 S 作为文法的开始符号，这样得到的一个与 G 等价的文法 G'称为 G 的拓广文法。对于拓广文法 $G[S']$，有项目 $S'\rightarrow S\ \cdot$，它是一个特殊的归约项目，我们称它为接受项目，它所对应的状态为接受状态。引入拓广文法的目的在于，拓广文法 G'的接受项目是唯一的，即 $S'\rightarrow S\cdot$。下面举一个例子，例如，已知文法 $G(S)$：$S\rightarrow aS|b$，若分析句子 ab，当把 b 归约为 S 时，好像是归约到了文法的开始符号 S，整个分析就结束了，但实际上还需要进一步归约，因此，引入拓广文法就可以避免这一问题。

对例 5-4 的文法进行拓广，引入一个新的开始符号 S'，则拓广后的文法为 $G(S')$：$S'\rightarrow S$，$S\rightarrow aAcBe$，$A\rightarrow Ab|b$，$B\rightarrow d$，对其所有 LR(0)项目进行编号：

(1) $S\rightarrow\cdot S'$　(2) $S\rightarrow S'\cdot$　(3) $S\rightarrow\cdot aAcBe$　(4) $S\rightarrow a\cdot AcBe$　(5) $S\rightarrow aA\cdot cBe$

(6) $S\rightarrow aAc\cdot Be$　(7) $S\rightarrow aAcB\cdot e$　(8) $S\rightarrow aAcBe\cdot$　(9) $A\rightarrow\cdot Ab$　(10) $A\rightarrow A\cdot b$

(11) $A\rightarrow Ab\cdot$　(12) $B\rightarrow\cdot d$　(13) $B\rightarrow d\cdot$　(14) $A\rightarrow\cdot b$　(15) $A\rightarrow b\cdot$

其中(2)、(8)、(11)、(13)、(15)为归约项目，(2)为接受项目，(1)、(4)、(6)、(9)为待约项目，(3)、(5)、(7)、(10)、(12)、(14)为移进项目。

前面已经提到，找出文法所有的活前缀并确定最长的活前缀是确定分析动作的关键，在给出 LR(0)项目的定义和分类之后，可以定义文法的所有 LR(0)项目，每个项目对应一个分析状态。因此，利用这些 LR(0)项目，可以构造能识别文法所有可归前缀的有限自动机。具体地，构造有限自动机可以从初始 LR(0)项目出发，通过引入闭包运算和 GO 转移函数来实现。

定义 5-8　LR(0)项目集的闭包运算

假定 I 是文法 G 的任一 LR(0)项目集，I 的项目集闭包 closure(I)通过以下步骤生成：

(1) I 中的每一个 LR(0)项目皆属于 closure(I)。

(2) 若项目 $A \rightarrow \alpha \cdot B\beta \in \text{closure}(I)$ ($B \in V_N$),则把文法 G 中形如 $B \rightarrow \eta$ 对应的 LR(0)项目 $B \rightarrow \cdot \eta$ 加进 closure(I)中。

(3) 重复执行(2)直到 closure(I)不再增大为止。

LR(0)项目集的闭包运算的含义是把项目集当中所有形如 $A \rightarrow \alpha \cdot B\beta$ 和 $B \rightarrow \cdot \eta$ 的 LR(0)项目放在一起,因为 $A \rightarrow \alpha \cdot B\beta$ 和 $B \rightarrow \cdot \eta$ 对应的分析状态是相同的,$A \rightarrow \alpha \cdot B\beta$ 表示将要开始识别 B,马上要进行分析,$B \rightarrow \cdot \eta$ 也表示还没有分析 B 能推导出的 η,两个 LR(0)项目均表示当前要分析 B,因此它们表示同一种状态,使用闭包运算来合并这些相同含义的状态。

【例 5-5】 已知文法 $G(S)$:

$S \rightarrow ABC$

$A \rightarrow Aa \mid a$

$B \rightarrow b \mid bB$

$C \rightarrow c$

假设 $I_1 = \{S \rightarrow \cdot ABC\}$,$I_2 = \{S \rightarrow A \cdot BC\}$求 I_1 和 I_2 的 LR(0)项目集闭包。

根据定义不难求出:

$\text{closure}(I_1) = \{ S \rightarrow \cdot ABC, A \rightarrow \cdot Aa, A \rightarrow \cdot a \}$

$\text{closure}(I_2) = \{ S \rightarrow A \cdot BC, B \rightarrow \cdot bB, B \rightarrow \cdot b \}$

定义 5-9 GO 转移函数

若 I 是 G 的一个 LR(0)项目集, $X \in \{V_T \cup V_N\}$,则

$\text{GO}(I, X) = \text{closure}(J)$, $J = \{A \rightarrow \alpha X \cdot \beta \mid A \rightarrow \alpha \cdot X\beta \in I\}$

$\text{GO}(I, X)$称为转移函数,项目 $A \rightarrow \alpha X \cdot \beta$ 称为 $A \rightarrow \alpha \cdot X\beta$ 的后继。

转移函数用来刻画不同 LR(0)项目集状态之间的关系,或者说从一个状态如何转移到下一个状态。例如,例 5-5 中的 I_1 和 I_2 之间的关系可以使用 GO 转移函数表示:

$\text{GO}(I_1, B) = \text{closure}(I_2) = \text{closure}(\{ S \rightarrow A \cdot BC\}) = \{ S \rightarrow A \cdot BC, B \rightarrow \cdot bB, B \rightarrow \cdot b \}$

结合 LR(0)项目集闭包的定义,可以很好地理解这两个定义的作用,即 LR(0)项目集闭包首先把表示相同状态的 LR(0)项目合并在一起形成 LR(0)项目集,然后利用转移函数建立不同 LR(0)项目集之间的跳转关系,从而形成识别文法所有可归前缀的有限自动机。

定义 5-10 LR(0)项目集规范族

识别文法 G 可归前缀的 DFA 中的所有 LR(0)项目集的全体称为文法 G 的 LR(0)项目集规范族。

根据定义 5-10,构造出了识别文法所有可归前缀的 DFA,也就构造出了 LR(0)项目集规范族。下面给出 LR(0)项目集规范族的构造算法。

算法 5-2 文法 $G[S]$的 LR(0)项目集规范族的构造算法

输入:文法 $G[S]$的拓广文法 G'

输出:文法 $G[S]$的 LR(0)项目集规范族 C

算法:

/ * 通过加入 $S' \rightarrow S$ 对文法进行拓广形成拓广文法 G',从而使接受状态易于识别 * /

BEGIN

$C=\{\text{closure}(S'\rightarrow \cdot S)\}$;
DO
　　FOR　$\forall I\in C$ 和$\forall X\in\{V_T\cup V_N\}$
　　　　把 go(I,X)加入到 C 中
　　WHILE C 不再增大
END;

算法 5-2 从文法的初始 LR(0)项目 $S'\rightarrow \cdot S$ 开始求其闭包 I,再通过 GO 函数求其所有的后继项目集,然后逐步扩展,且将其各项目集连成一个 DFA,最终求得的所有项目集 C 即为文法 G'的 LR(0)项目集规范族。

需要注意的是,用前述方法所构造的 LR(0)项目集中可能包括多个 LR(0)项目,每个 LR(0)项目表征了在分析过程中一个特定的分析动作,因此每个项目集中的各项目应是相容的,即没有冲突的。下面从项目集相容的角度出发,对 LR(0)文法加以定义。

定义 5-11　LR(0)文法

若一个文法 G 的识别可归前缀的 DFA 的每个状态对应的项目集均不存在以下情况:

(1) 既含移进项目又含归约项目。

(2) 含有多个归约项目。

则称项目集是相容的,称 G 是一个 LR(0)文法。

项目集是相容的也称为项目集是没有冲突的,而不相容的项目集一定包含"移进-归约"冲突或者"归约-归约"冲突。根据定义 5-11 可以得出:对任何一个 LR(0)文法对应的 LR(0)分析表一定不含多重定义。

至此,已经完成了所有构造 LR(0)分析表的准备工作,当识别其所有可归前缀的 DFA 构造出来以后,可以方便地构造出 LR(0)分析表及相应的 LR 分析器,因为从 DFA 中可以清楚地确定每个状态的含义,即确定是移进、归约、接受或者报错。所以 LR 分析器实质是一个带栈的确定的有限自动机,也称为下推自动机(关于下推自动机的定义不再介绍,读者可以参考可计算理论的相关内容)。

下面给出从识别可归前缀的 DFA 构造 LR(0)分析表的算法。

算法 5-3　利用识别可归前缀的 DFA 构造 LR(0)分析表

输入:文法 G;文法 G 的识别可归前缀的 DFA

输出:文法 G 的 LR(0)分析表

算法:

(1) 对于 $A\rightarrow\alpha\cdot x\beta\in S_i$,$GO(S_i,x)=S_j$:若 $x\in V_T$,则置 action$[S_i,x]=S_j$;若 $x\in V_N$,则置 goto$[S_i,x]=j$;

(2) 对于 $A\rightarrow\alpha\cdot\in S_i$,若 $A\rightarrow\alpha$ 是 G 中第 k 个产生式,则对所有输入符号 $x\in V_T$(包括 #),均置 action$[S_i,x]=r_k$。

(3) 若 $S'\rightarrow S\cdot\in S_i$,则置 action$[S_i,\#]=$acc(# 表示输入串右界符)。

(4) 其他情况均置错。

算法 5-3 构造的分析表称为 LR(0)分析表,使用 LR(0)分析表的 LR 分析器称为 LR(0)分析器。下面举例说明 LR(0)分析表的构造过程。

【例 5-6】 已知文法 $G[E]$：

$E \rightarrow aA \mid bB$

$A \rightarrow cA \mid d$

$B \rightarrow cB \mid d$

构造该文法的 LR(0)分析表。

首先对文法进行拓广，并对产生式进行编号：

(0) $S' \rightarrow E$ (1) $E \rightarrow aA$ (2) $E \rightarrow bB$ (3) $A \rightarrow cA$ (4) $A \rightarrow d$ (5) $B \rightarrow cB$ (6) $B \rightarrow d$

根据算法 5-2，从 $S' \rightarrow \cdot E$ 出发，构造识别文法所有可归前缀的 DFA，如图 5-4 所示。

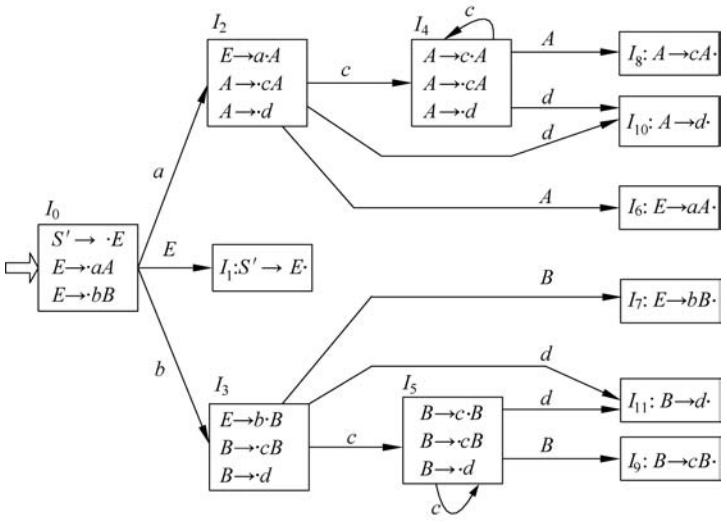

图 5-4 识别文法 $G[E]$所有可归前缀的 DFA

再根据算法 5-3，构造文法 $G[E]$的 LR(0)分析表如表 5-7 所示。

表 5-7 文法 $G[E]$的 LR(0)分析表

状态	action					goto		
	a	***b***	***c***	***d***	**#**	***E***	***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			

下面以句子 acd 为例，对表 5-7 进行说明。DFA 初始状态 0 经过终结符 a 到达状态 2，所以在 action[0,a]位置填写 S_2，表示当读入 a 时，分析器的动作为移进，把 a 移进符号栈，状态 2 压入状态栈。接着从状态 2 出发，经过终结符 c 到达状态 4，所以在 action[2,c]位置填写 S_4，表示当读入 c 时，继续移进，c 移进符号栈，状态 4 压入状态栈。同理，在 action[4,d]位置填写 S_{10}，状态 10 中对应归约项目 $A\rightarrow d\cdot$，其编号为 4，所以在 action 表中状态 10 所在的行填入 r_4，说明栈顶出现了句柄，需要进行归约，因此，d 从符号栈弹出，10 从状态栈弹出，然后 A 入栈。此时说明 A 已经产生，当前状态栈栈顶状态为 4，说明状态应该从状态 4 转到识别出 A 之后到达的状态 8，因此，在 goto 表的 goto[4,A]填入状态 8。分析表其他位置的元素也都是根据 DFA 进行构造。特别地，action[1,#]=acc，这是因为状态 1 对应LR(0)项目 $S'\rightarrow E\cdot$，说明要把 E 归约为 S'，如果缓冲区当前扫描到输入串右界符#，说明句子合法，分析动作为接受。

5.2.2　SLR(1)

LR(0)文法是一类非常简单的文法，它的项目集规范族中的每一项目集均不包含冲突，但是对于高级程序设计语言来说，构造出的项目集中往往包括“移进-归约”或者“归约-归约”冲突，因此 LR(0)分析的适用性受到很大的限制。下面通过例子进行说明。

【例 5-7】 设有文法 $G[S]$：

$S\rightarrow rD$

$D\rightarrow D,i\mid i$

试构造文法 G 的 LR(0)分析表。

首先将文法进行拓广，并对产生式进行编号：

(0) $S'\rightarrow S$　(1) $S\rightarrow rD$　(2) $D\rightarrow D,i$　(3) $D\rightarrow i$

构造项目集规范族和识别文法可归前缀的 DFA，如图 5-5 所示。

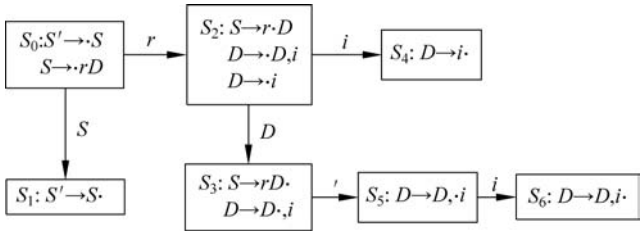

图 5-5　识别文法 $G[S]$所有可归前缀的 DFA

从图 5-5 中可看到，在项目集 S_3 中，既有移进项目($D\rightarrow D\cdot,i$)，又有规约项目($S\rightarrow rD\cdot$)，按照 LR(0)分析表的构造方法，在分析表中必有元素 action[3,,]=$\{S_5,r_1\}$，出现多重定义，如表 5-8 所示。这说明当分析到状态 3 时，下一个输入符号如果是“,”，或者将“,”移进符号栈，或者按文法的第一个产生式 $S\rightarrow rD$ 进行归约。于是出现了“移进-归约”冲突。其他项目集中没有出现冲突，所以要解决含有冲突动作的项目集 S_3 的问题，就可以使用 LR 分析方法。下面首先对 LR(0)分析表构造算法进行分析，找出不足，然后稍加修改，使其仍能适用于出现冲突的文法。其解决方法就是下面要研究的 SLR(1)分析法(即简单的 LR(1)分析法)。

表 5-8 文法 $G[S]$的 LR(0)分析表

状态	action				goto	
	r	,	*i*	#	*S*	*D*
0	S_2				1	
1				acc		
2			S_4			3
3	r_1	r_1,S_5	r_1	r_1		
4	r_3	r_3	r_3	r_3		
5			S_6			
6	r_2	r_2	r_2	r_2		

LR(0)分析表构造算法的问题在于,对于 $A\rightarrow\alpha\cdot\in S_i$,若 $A\rightarrow\alpha$ 是 G 中第 k 个产生式,则对所有输入符号 $x\in V_T$(包括#),均置 action$[S_i,x]=r_k$,也就是说,不管当前输入符号是什么,action 表中相应于状态 S_i 的那一行的元素都指定为 r_k。这是不合理的,因为只有当输入缓冲区中出现了紧跟在 A 之后的终结符号时才应该归约,因此,LR(0)分析法并没有进一步确定哪些符号紧跟在 A 之后,所以有可能即使出现了错误的句子,仍然进行归约。SLR(1)分析法的改进就是,对于归约项目 $A\rightarrow\alpha\cdot$,要确定紧跟在 A 之后的终结符号,即计算 FOLLOW(A)。

下面对这一处理过程进行描述,假定一个 LR(0)的规范族中存在如下含有冲突的项目集:

$$S_i=\{x\rightarrow\alpha\cdot b\beta,A\rightarrow\gamma\cdot,B\rightarrow\delta\cdot,\alpha,\beta,\gamma,\delta\in V^*,b\in V_T\}$$

显然,项目集中含有一个移进项目和两个归约项目。出现了“移进-归约”和“归约-归约”冲突。为解决冲突,对于归约项目 $A\rightarrow\gamma\cdot$ 和 $B\rightarrow\delta\cdot$,分别求 FOLLOW(A)和 FOLLOW(B)。若 FOLLOW(A)、FOLLOW(B)和$\{b\}$互不相交,则冲突可以解决,即对下一个输入符号 a 有:

(1) 当 $a=b$ 时,置 action$[S_i,b]$=“移进”。

(2) 当 $a\in$FOLLOW(A)时,置 action$[S_i,a]$=$\{$按产生式 $A\rightarrow\gamma$ 归约$\}$。

(3) 当 $a\in$FOLLOW(B)时,置 action$[S_i,a]$=$\{$按产生式 $B\rightarrow\delta$ 归约$\}$。

(4) 当 a 不属于上述 3 种情况时,置 action$[S_i,a]$=“error”。

上述方法仅对出现冲突的数据集进行处理,在冲突的地方简单地向前看一个符号,因此称为简单的 LR(1)分析法,即 SLR(1)方法。

下面给出从识别可归前缀的 DFA 构造 SLR(1)分析表的算法。

算法 5-4 利用识别可归前缀的 DFA 构造 SLR(1)分析表

输入:文法 G;文法 G 的识别可归前缀的 DFA

输出:文法 G 的 SLR(1)分析表

算法:

(1) 对于 $A\rightarrow\alpha\cdot x\beta\in S_i$,GO($S_i,x$)$=S_j$:若 $x\in V_T$,则置 action$[S_i,x]=S_j$;若 $x\in V_N$,则置 goto$[S_i,x]=j$。

(2) 对于归约项目 $A\rightarrow\alpha\cdot\in S_i$,若 $A\rightarrow\alpha$ 为文法的第 j 个产生式,则对于任意输入符号 a,若 $a\in$FOLLOW(A),则置 action$[S_i,a]=r_j$。

(3) 若 $S'\rightarrow S\cdot \in S_i$，则置 action[$S_i$，#]=acc（#表示输入串右界符）。

(4) 其他情况均置错。

比较 LR(0)和 SLR(1)分析表的构造算法可以看出，后者仅仅对于归约项目 $A\rightarrow\alpha\cdot$ 进行了进一步处理，计算 FOLLOW(A)。具体地，对于例 5-7，需要计算 FOLLOW(S)，FOLLOW(S)={#}，与{，}不相交，所以冲突可以解决。构造出的 SLR(1)分析表如表 5-9 所示。

表 5-9　文法 $G[S]$的 SLR(1)分析表

状态	action				goto	
	r	**，**	***i***	**#**	***S***	***D***
0	S_2				1	
1				acc		
2			S_4			3
3		S_5		r_1		
4	r_3	r_3	r_3	r_3		
5			S_6			
6	r_2	r_2	r_2	r_2		

从表 5-9 可以看出，SLR(1)分析方法比 LR(0)分析方法更确定，报错更及时。例如，在状态 3 下，下一个输入符号如果是 r 或者 i，则直接报错，而 LR(0)分析表（表 5-8）中对应的动作却是归约，因此，错误可能在若干步之后才会发现。

定义 5-12　SLR(1)文法

文法 G 按照 SLR(1)方法构造的分析表称为 SLR(1)分析表。如果每个入口不含多重定义，则文法 G 称为 SLR(1)文法。使用 SLR(1)分析表的 LR 分析器称为 SLR(1)分析器。

下面进一步讨论 LR(0)文法与 SLR(1)文法之间的关系。

【例 5-8】 已知文法 $G[E]$：

$E\rightarrow aA\mid bB$

$A\rightarrow cA\mid d$

$B\rightarrow cB\mid d$

构造该文法的 SLR(1)分析表。

首先对文法进行拓广，并对产生式进行编号：

(0) $S'\rightarrow E$　(1) $E\rightarrow aA$　(2) $E\rightarrow bB$　(3) $A\rightarrow cA$　(4) $A\rightarrow d$　(5) $B\rightarrow cB$　(6) $B\rightarrow d$

根据算法 5-2，从 $S'\rightarrow\cdot E$ 出发，构造识别文法所有可归前缀的 DFA，如图 5-6 所示。

状态 I_6、I_7、I_8、I_9、I_{10} 和 I_{11} 中含有归约项目，计算 FOLLOW(E)、FOLLOW(A)和 FOLLOW(B)，FOLLOW(E)=FOLLOW(A)=FOLLOW(B)={#}。

再根据算法 5-4，构造文法 $G[E]$的 SLR(1)分析表，如表 5-10 所示。对比文法的LR(0)分析表（表 5-7）可以看出，文法 $G[E]$的 SLR(1)分析表删除了一些归约动作，说明 SLR(1)分析报错更及时。同时，这个例子也说明，一个文法如果是 LR(0)文法，那么也一定是 SLR(1)文法；反之，一个文法是 SLR(1)文法，则不一定是 LR(0)文法。

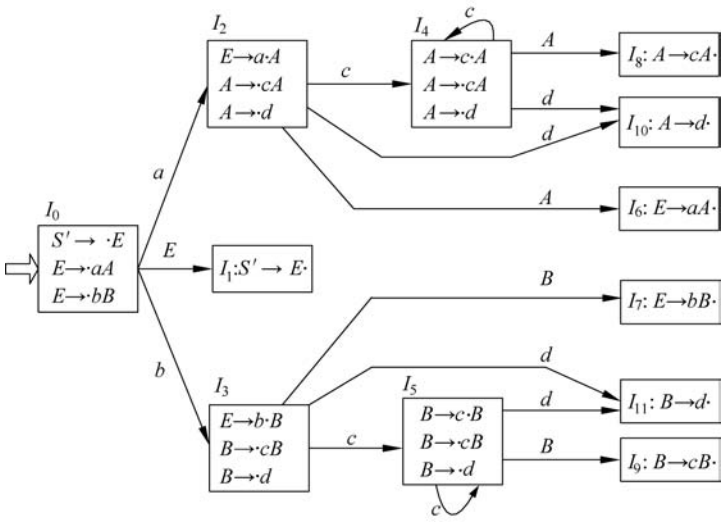

图 5-6 识别文法 $G[E]$所有可归前缀的 DFA

表 5-10 文法 $G[E]$的 SLR(1)分析表

状态	action						goto	
	a	***b***	***c***	***d***	**#**	***E***	***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			
7					r_2			
8					r_3			
9					r_5			
10					r_4			
11					r_6			

5.2.3 LR(1)

SLR(1)分析法是一种较为实用的方法。大多数程序设计语言基本上都可以用 SLR(1)文法来描述。在 SLR(1)分析法中,对于某状态 S_i,其项目集若不相容时,可根据 SLR(1)分析表的构造规则来解决冲突。然而,也确实存在许多无二义的文法,其项目集中的“移进-归约”和“归约-归约”冲突不能由 SLR(1)分析法解决,下面举例说明。

【例 5-9】 有文法 $G[S']$:

$S'\rightarrow S$

$S\rightarrow A=B$

$S \rightarrow B$

$A \rightarrow * B$

$A \rightarrow \mathrm{id}$

$B \rightarrow A$

判断该文法是否为 SLR(1)文法。

首先从 $S' \rightarrow \cdot S$ 出发，构造识别文法所有可归前缀的 DFA，如图 5-7 所示。

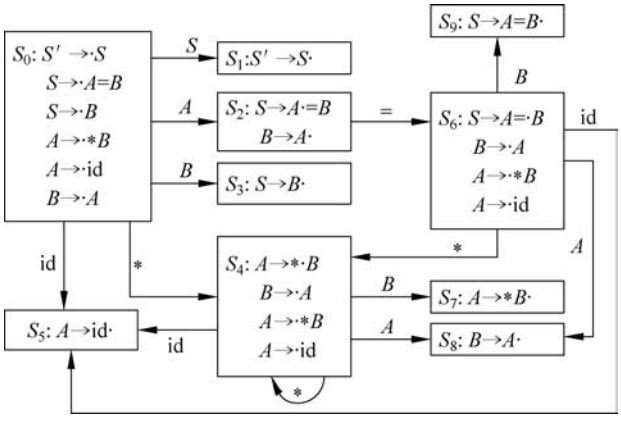

图 5-7　识别文法 $G[S']$所有可归前缀的 DFA

其中，状态 S_2 出现了移进-归约冲突，根据 SLR(1)分析法计算 FOLLOW(B)，FOLLOW(B)＝{＝，＃}，与状态 S_2 中的移进符号{＝}相交不为空集，冲突仍然没有解决，这说明 SLR(1)方法存在不足之处。

下面分析 SLR(1)方法存在的问题。SLR(1)分析法对于归约项目 $A \rightarrow \alpha \cdot$，要确定紧跟在 A 之后的终结符号，即计算 FOLLOW(A)。通俗地讲，就是要找出所有句型中紧跟在 A 之后的终结符号，因此，这就带来一个问题，分析过程中的一个状态往往只对应一个句型，我们关心的是在当前句型下紧跟在 A 之后的终结符号，而不是所有句型中紧跟在 A 之后的终结符号。例如，分析一下状态 S_2，S_2 是由初始状态 S_0 识别了 A 到达的，而 S_0 中有两个句型识别 A 之后到达 S_2。S_2 中的 $S \rightarrow A \cdot = B$ 对应的是句型 $S \Rightarrow A=B$，S_2 中的 $B \rightarrow A \cdot$ 对应句型 $S => A$ 而不是句型 $S \Rightarrow A=B$。SLR(1)方法的含义是在 S_2 状态下计算 FOLLOW(B)，既考虑句型 $A=B$ 中紧跟在 A 之后的＝，又考虑句型 A 中紧跟在 B 之后的＝，也就是说 SLR(1)方法仍然不够精确，没有细化到当前状态对应的句型。以上的分析说明，我们不仅关心紧跟在 A 之后的终结符号，更关心对当前状态有效的句型中紧跟在 A 之后的终结符号，因此，需要进一步确定 FOLLOW(A)中哪些终结符号对当前状态的具体句型有效。对状态 S_2 有效的句型是 $S \Rightarrow A$，因此在状态 S_2 下，只有下一个输入符号出现了＃才进行归约，如果有效句型是 $S \Rightarrow A=B$，出现＝时则移进，冲突可以解决。因此，通过把 LR(0)项目与有效的句型绑定，就可以设计出比 SLR(1)更有效的 LR 分析方法。

通过以上分析可知，问题的关键点在于如何确定当前句型中紧跟在 A 之后的终结符号。对于当前分析状态 S_i：$S \rightarrow \alpha \cdot A\omega$，$A \rightarrow \cdot \gamma$(假定 ω 推导不出 ε)，把 FIRST(ω)作为用产生式 $A \rightarrow \gamma$ 进行归约的搜索符，用于替代 SLR(1)分析法中的 FOLLOW(A)，并把搜索符号的集合放在项目的后面($[A \rightarrow . \gamma, a]$ $a \in$ FIRST(ω))。这种处理方法称为 LR(1)分

析法。

定义 5-13 LR(1)项目

在 LR(0)项目中放置一个向前搜索的符号 a，成为[$A\rightarrow\alpha\cdot\beta,a$]。$A\rightarrow\alpha\cdot\beta$ 是一个 LR(0)项目，a 是一个终结符号，这种形式的项目称为 LR(1)项目，a 称为向前搜索符。

向前搜索符仅对归约项目[$A\rightarrow\alpha\cdot,a$]有意义，当前输入符号是 a 时，才能用 $A\rightarrow\alpha$ 进行归约。

定义 5-14 LR(1)项目对活前缀的有效性

若文法 G 的一个 LR(1)项目[$A\rightarrow\alpha\cdot\beta,a$]对活前缀 γ 是有效的，当且仅当存在规范推导 $S\Rightarrow\delta Aw\Rightarrow\delta\alpha\beta w$ ($w\in V_T^*$，$A\in V_N$)，$\gamma=\delta\alpha$，$a\in$FIRST(w)或 $a=$'#'($w=\varepsilon$)。

定义 5-14 的本质在于把 $A\rightarrow\alpha\cdot\beta$ 与句型 δAw 对应起来，或者说进行绑定，这样就可以确定当前分析状态下的搜索符，即当前句型紧跟在 A 之后的终结符号。因此，必须准确计算出每个 LR(1)项目的向前搜索符，即构造出的每一个 LR(1)项目对活前缀(即对应一个句型)都必须是有效的。例 5-9 中，S_0 状态中的 $B\rightarrow\cdot A$ 对应的是句型 $S=>A$，所以称[$B\rightarrow\cdot A$，#]对活前缀 A 是有效的，换句话说，[$B\rightarrow\cdot A$，#]对句型 A 也是有效的。因此，定义 5-14 的目的是更好地增强 LR 分析的处理冲突的能力。

与 LR(0)分析的情况相类似，识别文法全部可归前缀的 DFA 的每一个状态也用一个 LR(1)项目集来表示，故构造 LR(1)项目集规范族的方法和构造 LR(0)项目集规范族的方法在本质上是一样的，同样需要用到函数 closure(I)和 GO(I，X)。

定义 5-15 LR(1)项目集的闭包运算

假定 I 是文法 G 的任一 LR(1)项目集，I 的项目集闭包 closure(I)通过以下步骤生成：

(1) I 中的每一个 LR(1)项目皆属于 closure(I)。

(2) 若项目[$A\rightarrow\alpha\cdot B\beta,a$]$\in$closure($I$)($B\in V_N$)，则把文法 G 中形如 $B\rightarrow\eta$ 对应的 LR(1)项目[$B\rightarrow\cdot\eta,b$] ($b\in$FIRST(βa))加进 closure(I)中。

(3) 重复执行(2)直到 closure(I)不再增大为止。

定义 5-16 GO 转移函数

若 I 是 G 的一个 LR(1)项目集，$X\in\{V_T\cup V_N\}$，则

$$GO(I,X)=closure(J)$$

其中，$J=\{[A\rightarrow\alpha X\cdot\beta,a]\mid$当[$A\rightarrow\alpha\cdot X\beta,a$]$\in I\}$，GO($I$，$X$)称为转移函数。项目 $A\rightarrow\alpha X\cdot\beta$ 称为 $A\rightarrow\alpha\cdot X\beta$ 的后继。

定义 5-17 LR(1)项目集规范族

识别文法 G 可归前缀的 DFA 中的所有 LR(1)项目集的全体称为文法 G 的 LR(1)项目集规范族。

下面给出 LR(1)项目集规范族的构造算法。

算法 5-5 文法 $G[S]$的 LR(1)项目集规范族的构造算法

输入：文法 $G[S]$的拓广文法 G'

输出：文法 $G[S]$的 LR(1)项目集规范族 C

算法：

/* 通过加入 $S'\rightarrow S$ 对文法进行拓广形成拓广文法 G'，从而使接受状态易于识别 */

BEGIN

$C=\{\text{closure}(\{[S'\rightarrow \cdot S,\#]\})\}$；

DO

FOR　$\forall I\in C$ 和$\forall X\in\{V_T\cup V_N\}$

把 go(I,X)加入到 C 中

WHILE C 不再增大

END；

【例 5-10】 有文法 $G[S']$：

(0) $S'\rightarrow S$

(1) $S\rightarrow A=B$

(2) $S\rightarrow B$

(3) $A\rightarrow *B$

(4) $A\rightarrow$id

(5) $B\rightarrow A$

构造该文法的 LR(1)项目集规范族。

根据算法 5-5,从$[S'\rightarrow \cdot S,\#]$出发,构造初始项目集,然后逐步扩展生成文法的 LR(1)项目集规范族,如图 5-8 所示。

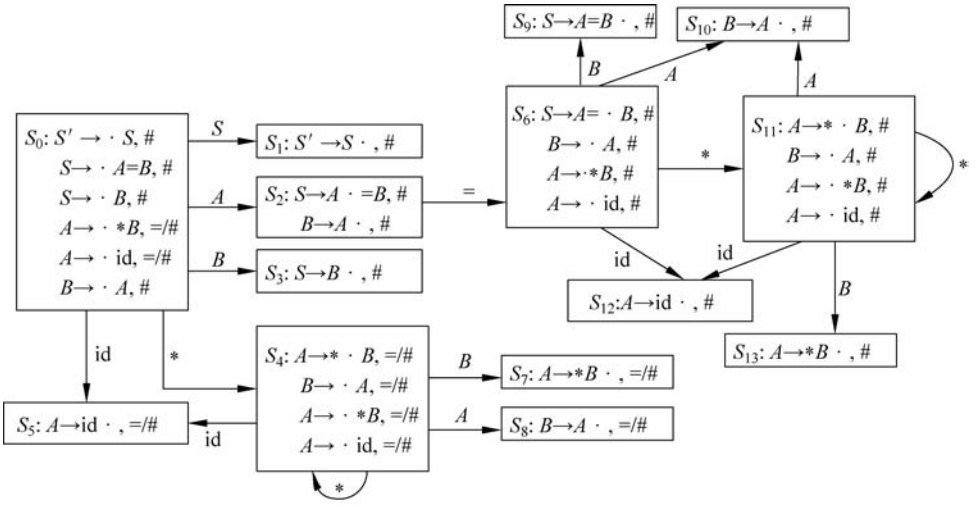

图 5-8　文法 $G[S']$的 LR(1)项目集规范族和识别可归前缀的 DFA

对于给定文法,当 LR(1)项目集规范族和识别可归前缀的 DFA 构造出来以后,可以方便地构造出文法的 LR(1)分析表。

算法 5-6　LR(1)分析表的构造

输入：文法 G 的 LR(1)项目集规范族和识别可归前缀的 DFA

输出：文法 G 的 LR(1)分析表

算法：

(1) 对于$[A\rightarrow\alpha\cdot x\beta,b]\in S_i$,$\text{GO}(S_i,x)=S_j$：若 $x\in V_T$,则置 action$[S_i,x]=S_j$；若 $x\in V_N$,则置 goto$[S_i,x]=j$。

(2) 对于$[A\rightarrow\alpha\cdot,a]\in S_i$,若$A\rightarrow\alpha$是$G$中第$k$个产生式,则置action$[S_i,a]=r_k$。

(3) 若$[S'\rightarrow S\cdot,\#]\in S_i$,则置action$[S_i,\#]$=acc(#表示输入串右界符)。

(4) 其他情况均置错。

算法5-6构造的分析表称为LR(1)分析表,若一个文法构造出的LR(1)分析表不含多重定义,则该文法称为LR(1)文法,使用LR(1)分析表的LR分析器称为LR(1)分析器。下面举例说明LR(1)分析表的构造过程。

【例5-11】 有文法$G[S']$:

(0) $S'\rightarrow S$

(1) $S\rightarrow A=B$

(2) $S\rightarrow B$

(3) $A\rightarrow *B$

(4) $A\rightarrow$id

(5) $B\rightarrow A$

构造该文法的LR(1)分析表。

在图5-8的基础上,根据算法5-6,构造文法$G[S']$的LR(1)分析表,如表5-11所示。与LR(0)和SLR(1)相比较,构造LR(1)分析表最大的不同在于算法的第二步,即利用归约项目中的向前搜索符就可以准确确定归约动作,也就是说,LR(1)分析法中,每一个LR(1)项目都对应向前看一个符号,向前搜索符虽然对非归约项目暂时没有作用,但它是为后面出现的归约项目提前做准备的,因此每一步都向前看一个符号,而不是出现冲突了才向前看一个符号。

表5-11 文法$G[S']$的LR(1)分析表

状态	action				goto		
	=	*	id	#	*S*	*A*	*B*
0		S_4	S_5		1	2	3
1				acc			
2	S_6			r_5			
3				r_2			
4		S_4	S_5			8	7
5	r_4			r_4			
6		S_{11}	S_{12}			10	9
7	r_3			r_3			
8	r_5			r_5			
9				r_1			
10				r_5			
11		S_{11}	S_{12}			10	13
12				r_4			
13				r_3			

前面已经说明,一个文法如果是LR(0)文法,那么也一定是SLR(1)文法,这是因为SLR(1)分析比LR(0)分析多计算了归约项目$A\rightarrow\alpha\cdot$的FOLLOW(A)集合,相当于删除了

LR(0)分析表中的一些归约动作。同理，一个文法如果是SLR(1)文法，那么也一定是LR(1)文法，这是因为LR(1)分析进一步确定了FOLLOW(A)中的对当前分析句型有效的终结符号，即向前搜索符。因此，LR(1)分析进一步确定了在FOLLOW(A)中哪些终结符号有效，哪些无效，即LR(1)分析表进一步删除了SLR(1)分析表中的一些归约动作。若一个文法的SLR(1)分析表中没有多重定义，则该文法的LR(1)分析表更不会有多重定义。

3种LR分析法的最大的区别就在于分析表的构造不同，是一个逐步确定化和细化的过程，它们之间的关系如表5-12所示。

表5-12　3种LR分析法的对比

LR分析法	优　　点	缺　　点
LR(0)分析法	无任何冲突，所以分析表构造最简单	所有终结符号均归约，查错慢，仅限于LR(0)文法
SLR(1)分析法	对FOLLOW(A)中的终结符号才归约，查错较快	没有考虑当前分析的句型，查错较LR(1)慢，仅限于SLR(1)文法
LR(1)分析法	对FOLLOW(A)中的对当前分析句型有效的终结符号才归约，每一步都求出这种终结符号，查错最快	构造LR(1)项目集规范族较复杂，识别可归前缀的DFA状态数较多

5.2.4　LALR(1)

前面已经提到，LR(1)分析法通过向前搜索符确切地指出归约时FOLLOW集中哪些终结符号有效，但向前搜索符的引入会带来状态数的增加。例如，假设SLR(1)中存在一个项目集$[A\rightarrow\alpha\ \cdot\ ]$，则在LR(1)中可能对应两个项目集$[A\rightarrow\alpha\ \cdot\ ,a]$和$[A\rightarrow\alpha\ \cdot\ ,b]$，优点是使分析更加确定，缺点是增加了状态。若对高级程序构造对应的LR(1)分析表，一般是比较庞大的，在机器的存储方面会遇到问题，从而影响到它的应用和推广。

LALR(1)分析法(Look-Ahead LR)是对LR(1)分析法的一种简化和改进，它的思想是对LR(1)中能够合并的项目集进行合并，从而减少状态。对同一个文法，合并LR(1)项目集之后得到的LALR(1)分析表比LR(1)分析表状态数少，具有和SLR(1)分析表相同数目的状态，但却能胜任SLR(1)所不能解决的问题。对目前常用的各类程序设计语言，LALR(1)分析法基本能够适用，但是缺点是状态的减少导致其比LR(1)报错要慢一点，所以从本质上讲，LALR(1)分析法是一种介于SLR(1)和LR (1)之间的折中的方法。

因此，LALR(1)分析法的关键问题在于LR(1)中哪些项目集能够合并，为此，给出如下同心集的定义。

定义5-18　同心集

若LR(1)的两个项目集的LR(0)项目全部相同，则称两个LR(1)项目集具有相同的心。具有相同心的项目集称为同心集。

【例5-12】　有文法$G[S']$：

(0) $S'\rightarrow S$

(1) $S\rightarrow CC$

(2) $C\rightarrow cC$

(3) $C \rightarrow d$

构造该文法的 LR(1)项目集规范族,并合并同心集。

从[$S' \rightarrow \cdot S$, #]出发,构造初始项目集,然后逐步扩展生成文法的 LR(1)项目集规范族,如图 5-9 所示。不难看出,I_4 和 I_7、I_3 和 I_6、I_8 和 I_9 是同心集,LALR(1)分析的思想就是在文法 LR(1)项目集规范族的基础上,将同心的项目集进行合并,合并之后得到的项目集规范族称为 LALR(1)项目集规范族。文法 $G[S']$的 LALR(1)项目集规范族如图 5-10 所示。

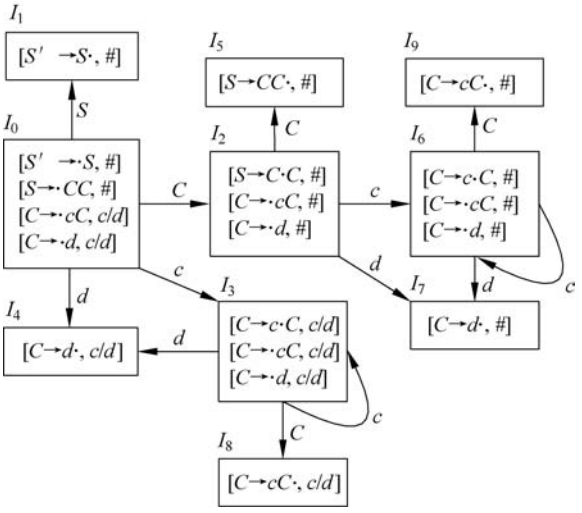

图 5-9 文法 $G[S']$的 LR(1)项目集规范族和识别可归前缀的 DFA

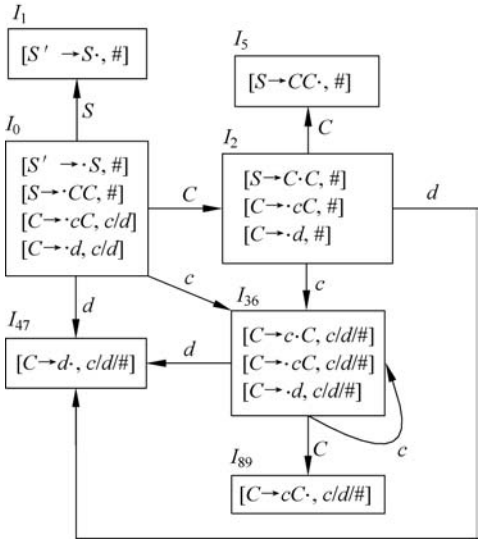

图 5-10 文法 $G[S']$的 LALR(1)项目集规范族和识别可归前缀的 DFA

一般来说,对于某个 LR(1)文法,若把同心项目集合并,可能会导致冲突,这种冲突不会是“移进-归约”冲突,仅可能产生新的“归约-归约”冲突。例如,假定 I_k 和 I_j 是两个同心

集，如下所示：

I_k：{[$A\to\alpha\cdot,u_1$] [$B\to\eta\cdot,u_2$] [$C\to\beta\cdot a\gamma,b$]}　$a\cap u_1=\varnothing$　$a\cap u_1=\varnothing$

I_j：{[$A\to\alpha\cdot,u_2$] [$B\to\eta\cdot,u_1$] [$C\to\beta\cdot a\gamma,c$]}　$a\cap u_2=\varnothing$　$a\cap u_1=\varnothing$

则合并之后的项目集 I_{kj} 如下：

I_{kj}：[$A\to\alpha\cdot,u_1/u_2$]　[$B\to\eta\cdot,u_1/u_2$]　[$C\to\beta\cdot a\gamma,b/c$]　$a\cap\{u_1,u_2\}=\varnothing$

因为原有项目集中假定没有“移进-归约”冲突，所以合并之后也不会出现“移进-归约”冲突，即 $a\cap\{u_1,u_2\}=\varnothing$。但是出现了“归约-归约”冲突。例如，下一个输入符号若是 u_1 或 u_2，均无法确定使用 $A\to\alpha\cdot$ 还是 $B\to\eta\cdot$ 进行归约。

若合并后的 LALR(1)项目集规范族不存在“归约-归约”冲突，则可按这个项目集规范族构造 LALR(1)分析表。构造 LALR(1)分析表的算法如下。

算法 5-7　LALR(1)分析表的构造

输入：文法 G 的 LALR(1)项目集规范族和识别可归前缀的 DFA

输出：文法 G 的 LALR(1)分析表

算法：

设 $C=\{S_0,S_1,\cdots,S_n\}$为文法 G 的 LALR(1)项目集规范族。

(1) 对于[$A\to\alpha\cdot x\beta,b$]$\in S_i$，GO(S_i,x)$=S_j$：若 $x\in V_T$，则置 action[S_i,x]$=S_j$；若 $x\in V_N$，则置 goto[S_i,x]$=j$。

(2) 对于[$A\to\alpha\cdot,a$]$\in S_i$，若 $A\to\alpha$ 是 G 中第 k 个产生式，则置 action[S_i,a]$=r_k$。

(3) 若[$S'\to S\cdot,\#$]$\in S_i$，则置 action[$S_i,\#$]=acc(#表示输入串右界符)。

(4) 其他情况均置错。

在 LALR(1)项目集规范族基础上，LALR(1)分析表的构造算法和 LR(1)分析表的构造完全相同，按上述算法构造的 LALR(1)分析表若无多重定义，则称文法 G 是 LALR(1)文法。使用 LALR(1)分析表的分析器叫 LALR(1)分析器。

例 5-12 的 LALR(1)项目集规范族中没有冲突，按上述算法对其构造的 LALR(1)分析表如表 5-13 所示。如前所述，LALR(1)分析表中的状态进行了合并，优点是减少了状态和存储空间，缺点是分析不够确定，报错较 LR(1)慢。

表 5-13　例 5-12 的 LALR(1)分析表

状态	action			goto	
	c	***d***	**#**	***S***	***C***
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		

上面介绍的4种LR分析法对应4种LR文法和LR分析器。那么假定给出任意一个无二义文法G,如何判断G属于哪一类LR文法呢?可以通过一个流程图给出判别的过程,如图5-11所示。

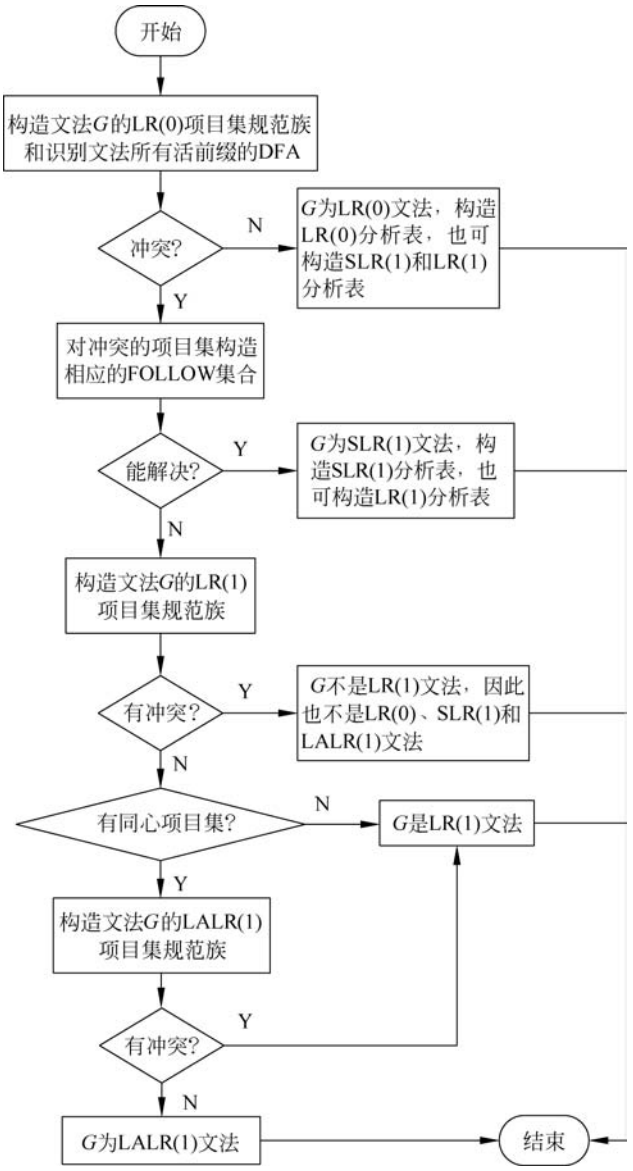

图5-11 判定文法G属于何种LR文法的流程图

文法的谱系如图5-12所示。下面不加证明地给出不同文法之间的关系。

LL(k)和LR(k)($k\geqslant 0$)文法均是无二义文法,LL(k)文法一定是LR(k)文法。一个文法如果是LR(0)文法,则一定是SLR(1)、LALR(1)、LR(1)和LR(k)($k>1$)文法。一个文法如果是SLR(1)文法,则一定是LALR(1)、LR(1)和LR(k)($k>1$)文法。一个文法如果是LALR(1)文法,则一定是LR(1)和LR(k)($k>1$)文法。

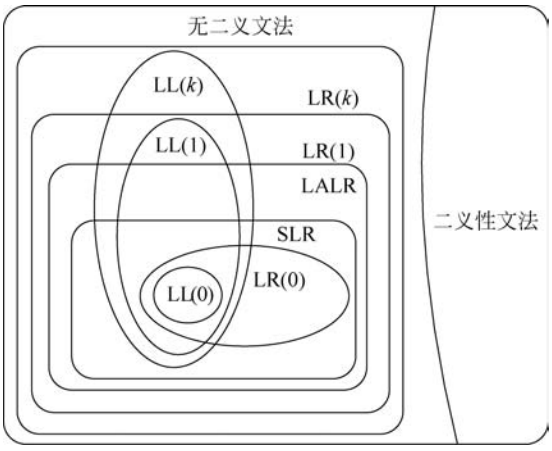

图 5-12　文法的谱系

5.3　语法分析程序自动生成器 YACC

本节介绍一个著名的语法分析器自动生成工具——YACC/Bison。它是以有限自动机理论为基础建立的。

1. YACC 综述与应用

YACC(Yet Another Compiler-Compiler)是一个 LALR(1)分析器自动生成器。YACC 与 Lex 一样,是贝尔实验室在 UNIX 上首先实现的,而且与 Lex 有直接的接口,是 UNIX 的标准应用程序。GNU 工程推出 Bison,是对 YACC 的扩充,同时也与 YACC 兼容。目前,YACC/Bison 与 Lex/Flex 一样,可以在 UNIX、Linux、MS-DOS 等环境运行,鉴于 YACC/Bison 的兼容,后面讨论中仅针对 YACC 进行介绍。

YACC 的功能是,为上下文无关文法自动生成基于 LALR(1)的方法的语法语义分析器(简称分析器),该分析器是使用 C 语言实现的。

YACC 自动构造分析器的模式及 YACC 的作用如图 5-13 所示。YACC 编译器接受 YACC 源程序,由 YACC 编译器处理 YACC 源程序,产生一个分析器作为输出。在 UNIX 环境中,YACC 编译器的输出是一个具有标准文件名 y.tab.c 的 C 程序,经过 C 编译器产生 a.out 文件,a.out 是一个实际可以运行的分析器。

图 5-13　YACC 自动构造分析器的模式及作用

使用 YACC 的步骤如图 5-14 所示。

(1) 编辑 YACC 源程序(例如,生成文本格式的关于 PAS 语言语法的 YACC 源程序文件 PAS.y)。

(2) 使用命令 yacc PAS.y 运行 YACC,若正确则输出 y.tab.c。

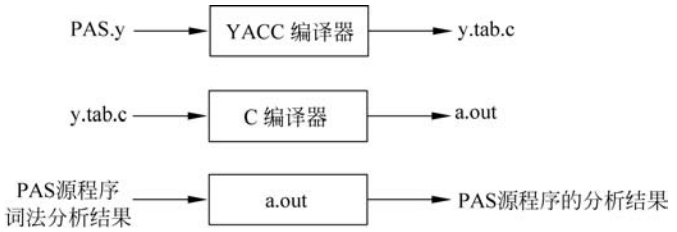

图 5-14 使用 YACC 的步骤

(3) 调用C编译器编译 y.tab.c,并与其他C语言模块连接产生执行文件。调试执行文件,直至获得正确输出。

为了使 LALR(1)分析表少占空间,可以用紧凑技术压缩分析表的大小。即使用命令

```
cc y.tab.c -ly
```

编译 y.tab.c,其中的 ly 表示使用 LR 分析器的库(名字 ly 随系统而定)。

用 BNF 对语言(设语言为 L1)的语法规则进行描述,然而 BNF 实际输入的是用 YACC 语言书写的源程序 L1.y,y 经 YACC 编译器翻译生成识别语言 L1 的语法分析器 y.tab.c,此分析器即能对 L1 源程序实现语法分析。

YACC 体系包括 YACC 语言和 YACC 编译器两部分。

2. YACC 语言

YACC 语言及 YACC 源程序是对语言的语法规则的描述,以解决文法规则的输入。YACC 语言作为分析器自动构造的专用语言。YACC 源程序由 3 部分组成,其结构如下:

```
说明部分
%%
翻译规则
%%
辅助过程
```

其中,说明部分通常包含两部分内容。一部分为通常的C语言程序的说明,该部分说明用一对符号%{和%}括起来;另一部分内容为文法符号(一般为终结符号)和文法规则的说明,以及对文法规则说明的一些限定规则和条件的说明。该部分的每一项均以%开头,其形式如下:

```
%说明
```

翻译规则部分是 YACC 源程序的主体部分,它以一对百分号%%标志该部分的开始,其内容是文法的全部规则及与每一文法规则相关的语义动作描述。对文法中某一文法规则:

<左部文法符号>→<候选式 1>|<候选式 2>|…|<候选式 n>

用 YACC 描述的一般形式为

<左部文法符号>: <候选式 1>{语义动作 1}

```
|<候选式 2>{语义动作 2}
⋮
|<候选式 n>{语义动作 n}
;
```

其中文法规则描述的候选式中，对文法的终结符号要用单引号括起来，以示与非终结符号的区别。该部分描述的第一个左部文法符号是开始符号。语义动作是完成语义处理的 C 语言程序。语义动作中的符号 $ $ 表示与文法规则的左部非终结符号相关的属性值，而 $ i 表示其右部候选式中的第 i 个文法符号的值。在分析过程中，每当选用某个产生式进行规约后，其产生式后的语义动作子程序即被执行，完成相应的语法范畴的翻译。

例如，对简单的表达式文法

$E \rightarrow E+T \mid T$

其 YACC 源程序的翻译规则描述部分可表示为

```
%%
    E: E'+'T { $$ = $1 + $3}
    |T
;
```

YACC 程序的第三部分，即辅助过程部分，是若干个 C 语言函数构成的，其中词法分析程序及错误诊断程序都是必不可少的。

【例 5-13】 设文法 $G(A)$：

$A \rightarrow E+E \mid E*E \mid \text{NUMBER}$

文法 $G(A)$的 YACC 源程序如下：

```
%{
    #include <ctype.h>
    #include <stdio.h>
%}
%token number
%%
lines:lines expr '\n'{printf("%g\n", $2);}
;
expr:expr '+' expr { $$ = $1 + $3}
    |expr '*' expr { $$ = $1 * $3}
;
%%
…     /*辅助过程*/
```

程序中省略了 YACC 程序的第三部分。在实际使用 YACC 时，这部分可根据 YACC 的使用说明及对文法翻译的具体要求编写所需要的辅助程序。

对于上述 YACC 说明的文法二义性，LALR(1)算法将产生分析动作的冲突。YACC 会报告产生的分析动作的冲突数目。项目集和分析冲突的描述可以在调用 YACC 时加-v 选择项得到。这个选择产生一个附加的文件 y.output，它包括分析时发现的项目集的心以及对 LALR(1)算法产生的分析动作冲突的描述。

带有冲突的 LR 分析表显然无法正确实施语法分析。考虑 YACC 的适用性，下面讨论

YACC 对二义性文法的处理。

3. YACC 处理二义性文法

YACC 自动生成的分析程序采用的是 LALR(1)分析法。按照 LR 分析应用于二义性文法的思想,即对二义性文法施加某些限定,YACC 同样可以适用于二义性文法分析器的自动生成。在 YACC 源程序的说明部分对规则进行描述,则 YACC 对二义性文法也可以产生 LR 分析器。

对例 5-13 的台式计算器文法 $G(A)$扩大其表达功能,给出如下文法 $G(A)'$:

$A \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid -E \mid \text{NUMBER}$

该文法是二义性文法,但只要对其中的终结符号+、—、*、/、NUMBER 规定优先级和结合规则,并在 YACC 源程序的说明部分中加以说明,YACC 就能自动生成文法 $G(A)$产生 LR 分析器。

$G(A)'$的 YACC 源程序如下:

```
%{
        #include<ctype.h>
        #include<stdio.h>
        #define YYSTYPE double
%}
        %token NUMBER
        %LEFT '+' '-'
        %LEFT '*' '/'
        %RIGHT UMINUS
%%
  lines:lines expr '\n'{printf("%g\n", $2);}
       |line '\n'
       |    /*ε*/
       ;
  expr: expr      '+'              {$$ = $1+ $3;}
       |expr      '-'expr          { $$ = $1- $3;}
       |expr      '*'expr          { $$ = $1* $3;}
       |expr      '/'expr          { $$ = $1/$3;}
       |'('expr')'                 {$$ = $2;}
       |'-'expr %&prec UMINUS      {$$ = -$2;}
       |NUMBER
;
%%
yylex()
int c;
while((c=getchar())==''){
    if((c=='.')||(isdigit(c)) {
        ungetc(c,stdin);
        scanf("%if",&yylval);
        return NUMBER;
    }
    return c;
}
```

该 YACC 程序的声明部分为终结符号规定了优先级和结合性。声明

```
%terminal '+''-'LEFT
%terminal '*''/'LEFT
```

使得＋和－有同样的优先级且为左结合。＊和/也是如此。

记号的优先级按它们在声明部分出现的次序确定，先出现的记号的优先级低，同一声明中的记号有相同的优先级，这样，上述 YACC 程序的声明中

```
%RIGHT UMINUS
```

使得 UMINUS 的优先级高于前面 5 个终结符号。

习题

1. 选择题。

(1) 文法识别符号经过任意步推导得到的结果是(　　)。

A. 句型　　B. 句柄　　C. 句子　　D. 短语

(2) 在自底向上的规范归约中，可归约串用(　　)来刻画。

A. 直接短语　　B. 句柄　　C. 终结符号串　　D. 最左短语

(3) 下面对自底向上分析描述错误的是(　　)。

A. 自底向上分析过程是对句子实施归约的过程

B. 自底向上分析是从给定的输入串 # 开始，逐步进行“归约”，直至归约到文法的开始符号

C. 各种自底向上分析方法的共同点是都采用移进-归约的思想，区别是确定可归约串的方法不同

D. 自底向上分析是规范归约的过程

(4) 下面(　　)不是自底向上的语法分析文法。

① LR(1)　② LALR(1)　③ 递归下降分析法　④ LL(1)　⑤ SLR(1)　⑥ LR(0)

A. ①②③④　　B. ③④　　C. ①②⑤⑥　　D. ③⑤⑥

(5) 若状态 j 含有项目“$X\rightarrow\alpha\cdot$ ”，且仅当输入符号 $b\in$FOLLOW(X)时，才用规则“$X\rightarrow\alpha$”归约的语法分析方法是(　　)。

A. LALR 分析法　　B. LR(0)分析法

C. LR(1)分析法　　D. SLR(1)分析法

(6) 若 b 为终结符，则 $X\rightarrow\alpha\cdot b\beta$ 为(　　)项目。

A. 归约　　B. 移进　　C. 接受　　D. 待约

(7) 若 Y 为非终结符，则 $X\rightarrow\alpha\cdot Y\beta$ 为(　　)项目。

A. 归约　　B. 移进　　C. 接受　　D. 待约

(8) 同心集合并之后若不是 LALR(1)文法，则会出现的情况是(　　)。

A. 产生二义性冲突　　B. 产生移进-移进冲突

C. 产生移进-归约冲突　　D. 产生归约-归约冲突

(9) 当分析文法 G 的某一个含有错误的符号串时,LALR(1)分析速度比 LR(1)分析速度要慢,主要原因是(　　)。

A. 合并同心集后做了不必要的移进动作

B. 合并同心集后做了不必要的归约动作

C. 合并同心集后做了不必要的移进与归约动作

D. 以上都不对

(10) 关于 LR 分析表,说法正确的是(　　)。

A. LR 分析表中的状态转换表指出了当状态 S_i 面临输入符号 a 时,移进之后转移到的新状态

B. LR 分析表中分析动作表指出了当状态 S_i 面临文法符号 X 时,应转移到的下一个状态

C. LR 分析表中的状态转换表指出了当状态 S_i 面临输入符号 a 时,应使用哪个规则进行归约

D. LR 分析表中分析动作表中符号为 acc 时,表示当前分析栈中只剩下开始符号 S'和待分析串结束符 #,表明分析成功

(11) 关于 LR 分析器,说法错误的是(　　)。

A. LR 分析器的每一步动作,都是由栈顶状态和现行输入符号所唯一确定的

B. LR 分析器的动作包括移进、归约、接受、报错四种动作

C. 不同的 LR 分析器,其总控程序也不同

D. LR 分析表包括分析动作表和状态转换表两部分,都是二维数组

(12) 关于文法规范句型的活前缀,说法错误的是(　　)。

A. 规范句型的活前缀不包含句柄右边的任何符号

B. LR 分析栈中的符号为规范句型的活前缀

C. 当 LR 分析栈中的栈顶符号串形成可归前缀,归约之后,栈中剩余符号不再是规范句型的活前缀

D. 活前缀可以是一个或者是若干个规范句型的前缀

(13) 设一个 LR(0)项目集 $I=\{[A\rightarrow\alpha\cdot b\beta],[B\rightarrow\gamma\cdot]\}$,该项目集可能含有的冲突项目是(　　)。

A. 移进-归约冲突　　B. 归约-归约冲突

C. 移进-接受冲突　　D. 不存在冲突

(14) 设 LR(1)项目集 $I=\{[A\rightarrow\alpha\cdot b\beta,a],[B\rightarrow\gamma\cdot,a]\}$,说法正确的是(　　)。

A. 存在移进-归约冲突　　B. 存在归约-归约冲突

C. 存在移进-接受冲突　　D. 不存在冲突

(15) 关于二义性文法与 LR 类文法,说法错误的是(　　)。

A. 任何一个二义性文法绝不是 LR 类文法

B. 在 LR 分析表中加入无二义性规则,仍然无法构造出无多重定义的 LR 分析表

C. 对于二义性的算术表达式文法,在 LR 分析表中加入运算符的优先级及运算符的结合性可构造出无多重定义的 LR 分析表

D. 可以通过消除二义性文法中的二义性,从而使得文法可以满足 LR 类文法

(16) LR(1)文法都是(　　)。

A. 既不存在二义性,也不存在文法左递归

B. 可以是二义性的,但是没有左递归

C. 不可以是二义性的,但是可以存在左递归

D. 可以是二义性的,还可以存在左递归

2. 填空题。

(1) "移进-归约"分析过程主要采取的分析动作包括(　　)、(　　)、(　　)和(　　)。

(2) LR(k)分析法中,L 指(　　),R 指分析过程是(　　)的逆过程,其中最简单的是(　　)分析法。

(3) LR 文法指构造出的分析表(　　)。

(4) 可归前缀指活前缀中已经包含句柄的(　　)。

(5) 向前搜索符只对(　　)项目有意义,对(　　)项目没有影响。表示(　　)与向前搜索符相同时,进行(　　)。

(6) LR 分析表是 LR 分析器的核心部分,包括(　　)和(　　)两部分组成,均使用(　　)表示。

(7) 同心集是指两个 LR(1)项目集的(　　)相同,当同心集合并后,可能会产生新的(　　),而不可能产生新的(　　)。

(8) 对于文法 $G[E]$: $E\rightarrow T|E+T \quad T\rightarrow F|T*F \quad F\rightarrow(E)|i$,句型 $E+T*F+T*F+T*i$ 的句柄是(　　)。

3. 分析应用题。

(1) 文法 $G[E]$为:

$E\rightarrow T \mid EiT$

$T\rightarrow F \mid T+F$

$F\rightarrow)E* \mid ($

试给出句型 $Ei)E*i($ 的短语、简单(直接)短语、句柄。

(2) 文法 $G[S]$为:

$S\rightarrow SdE \mid E$

$E\rightarrow E<T \mid T$

$T\rightarrow (S) \mid a$

试给出句型$(SdE)dT<a$ 的短语、简单(直接)短语、句柄。

(3) 已知文法 $G(S)$:

$S\rightarrow a|(T)|*$

$T\rightarrow T,S|S$

① 给出$(a,(a,a))$和$(((a,a),,(a)),a)*$的最左和最右推导。

② 指出$(((a,a),,(a)),a)\wedge$的规范归约及每一步的句柄。根据这个规范归约,给出"移进-归约"的过程,并给出它的语法树自底向上的构造过程。

(4) 设有文法 $G[S]$:

$S\rightarrow a|(T)|*$

$T\rightarrow T,S|S$

① 构造此文法的 LR(0)项目集规范族,并给出识别活前缀的 DFA。

② 构造其 LR(0)分析表。

(5) 给定文法 $G[Z]$:

① $Z \rightarrow CS$

② $C \rightarrow$ if E then

③ $S \rightarrow A = E$

④ $E \rightarrow E \wedge A$

⑤ $E \rightarrow A$

⑥ $A \rightarrow i$

其中:Z、C、S、A、$E \in V_N$

if、then、=、∧、$i \in V_T$

① 构造此文法的 LR(0)项目集规范族,并给出识别活前缀的 DFA。

② 构造其 SLR(1)分析表。

(6) 已知文法

$A \rightarrow aAcd \mid aAb \mid \varepsilon$

判断该文法是否是 SLR(1)文法,若是,构造相应的分析表,并对输入串 $ab\#$ 给出分析过程。

(7) 设文法 $G(S)$如下:

$E \rightarrow E + T \mid T$

$T \rightarrow T\ F \mid F$

$F \rightarrow F/ \mid a \mid b$

① 构造其 SLR 分析表。

② 构造其 LALR 分析表。

③ 假定输入串为 $a/b+b$,请给出 SLR(1)分析过程(即按照步骤给出状态、符号、输入串的变化过程)。

④ 假定输入串为 $a/b+b$,请给出 LALR(1)分析过程(即按照步骤给出状态、符号、输入串的变化过程)。

(8) 若有定义二进制数的文法如下:

$S \rightarrow L.L \mid L$

$L \rightarrow LB \mid B$

$B \rightarrow 0 \mid 1$

① 构造其 SLR 分析表。

② 假定输入串 101.110 ,请给出 SLR(1)分析过程(即按照步骤给出状态、符号、输入串的变化过程)。

(9) 给定文法 $G(S)$:

$S \rightarrow bAd \mid Aa \mid bc \mid BB$

$A \rightarrow a$

$B \rightarrow Bb \mid c$

构造其 LALR 分析表。

(10) 设已构造出文法 $G(S)$：

① $S \rightarrow AA$

② $A \rightarrow aA$

③ $A \rightarrow b$

要求：

① 构造 LR(1)分析表。

② 假定输入串为 $aabab$，请给出 LR(1)分析过程(即按照步骤给出状态、符号、输入串的变化过程)。

(11) 证明下面的文法是 LL(1)的，但不是 SLR(1)的。

$S \rightarrow AaAb \mid BbBa$　　$A \rightarrow \varepsilon$　　$B \rightarrow \varepsilon$

(12) 证明下列文法是 LR(1)的，但不是 SLR(1)的。

$S \rightarrow Aa \mid bAc \mid Bc \mid bBa$

$A \rightarrow d$

$B \rightarrow d$

(13) 考虑文法 $S \rightarrow AS \mid a$　　$A \rightarrow SA \mid b$

① 列出这个文法的所有 LR(0)项目。

② 构造这个文法的 LR(0)项目集规范族及识别活前缀的 DFA。

③ 这个文法是 SLR 的吗？若是，构造出它的 SLR 分析表。

④ 这个文法是 LALR 或 LR(1)的吗？