第5章　自底向上的分析

本章要点

- 自底向上分析概览
- LR(0)项的有穷自动机及LR(0)分析
- SLR(1)分析
- 一般的LR(1)和LALR(1)分析
- Yacc: LALR(1)分析程序的生成器
- 使用Yacc生成TINY分析程序
- 自底向上分析程序中的错误校正

前一章涉及到了递归下降和预测分析的自顶向下分析的基本算法。在本章中，我们将描述主要的自底向上的分析技术及其相关的构造。如在自顶向下的分析一样，我们将主要学习利用最多一个先行符号的分析算法，此外再谈一下如何扩展这个算法。

与LL(1)分析程序的术语相类似，最普通的自底向上算法称作　LR(1)分析(LR(1) parsing)(L表示由左向右处理输入，R表示生成了最右推导，而数字1则表示使用了先行的一个符号)。自底向上分析的作用还表现在它使得LR(0)分析(LR(0) parsing)也具有了意义，此时在作出分析决定时没有考虑先行(因为可在先行记号出现在分析栈上之后再检查它，而且倘若是这样发生的，它也就不会被算作先行了，所以这是可能的　)。SLR(1)分析(SLR(1) parsing，可简写为LR(1)分析)是对LR(1)分析的改进，它使用了一些先行。　LALR(1)分析(LALR(1) parsing，意即先行LR(1)分析)是比SLR(1)分析略微强大且比一般的LR(1)分析简单的方法。

在本章节中将会谈到以上每一个分析方法的必要构造。这将包括　LR(0)的DFA构造以及LR(1)各项：SLR(1)、LR(1)和LALR(1)分析算法的描述，以及相关分析表的构造。我们还将描述Yacc(一个LALR(1)分析程序生成器)的用法，并为TINY语言使用Yacc生成一个分析程序，该TINY语言构造出与在第5章中由递归下降分析程序开发的相同的语法树。

一般而言，自底向上的分析算法的功能比自顶向下的方法强大(例如，左递归在自底向上分析中就不成问题)。但是这些算法所涉及到的构造却更为复杂。因此，需要小心对它们的描述，我们还需要利用文法的十分简单的示例来介绍它们。本章的开头先给出了这样的两个例子，本章还会一直用到这两个例子。此外还会用到一些前章中的例子(整型算术表达式、if语句等等)。但是由于为全TINY语言用手工执行任何的自底向上的分析算法非常复杂，所以我们并不打算完成它。实际上，所有重要的自底向上方法对于手工编码而言都太复杂了，但是对于诸如Yacc的分析程序生成器却很合适。但是了解方法的操作却很重要，这样作为编译程序的编写者就可对分析程序生成器的行为进行正确地分析了。由于分析程序生成器可以用BNF中建议的语言语法来识别可能的问题，所以程序设计语言的设计者还可从这个信息中获益不少。

为了掌握好自底向上的分析算法，读者还需要了解前面讲过的内容，这其中包括了有穷自动机、从一个NFA中构造DFA的子集(第2章的2.3节和2.4节)、上下文无关文法的一般特性、推导和分析树(第3章的3.2节和3.3节)。有时也需要用到Follow集合(第4章的4.3节)。这一章从自底向上的分析的概况开始谈起。

5.1 自底向上分析概览

自底向上的分析程序使用了显式栈来完成分析，这与非递归的自顶向下的分析程序相类似。分析栈包括记号和非终结符，以及一些后面将讨论到的其他信息。自底向上的分析开始时栈是空的，在成功分析的末尾还包括了开始符号。因此，可将自底向上的分析示意为：

```
$                    InputString $
    ...                  ...
    ...                  ...
$ StartSymbol            $        accept
```

其中，分析栈在左边，输入位于正中间，而分析程序的动作则在右边 (此时，“ 接受 ”是所指出的唯一动作)。

自底向上的分析程序有两种可能的动作(除“ 接受 ”之外)：

1) 将终结符从输入的开头移进到栈的顶部。

2) 假设有BNF选择$A \rightarrow \alpha$，将栈顶部的串α归约为非终结符A。

因此自底向上的分析程序有时称作是移进-归约分析程序㊀。移进动作是由书写单词shift指出的。归约动作则由书写 *reduce*单词给出且指出在归约中所用的 BNF选择㊁。自底向上的分析程序的另一个特征是：出于后面要讲到的技术原因，总是将文法与一个新的开始符号一同扩充(augmented)。这就意味着若S是开始符号，那么就将新的开始符号S' 增加到文法中，同时还添加一个单元产生式到前面的开始符号中：

$$S' \rightarrow S$$

下面是两个示例，之后再讨论这两个例子所表现出的自底向上分析的一些特性。

例5.1 考虑以下用于成对括号的扩充文法：

$$S' \rightarrow S$$
$$S \rightarrow (\ S \ ) \ S \mid \varepsilon$$

表5-1给出了使用该文法的串( )的自底向上的分析。

表5-1 例5.1中文法的自底向上分析程序的分析动作

	分 析 栈	输 入	动 作
1	$	( ) $	移进
2	$ (	) $	用$S \rightarrow \varepsilon$ 归约
3	$ (S	) $	移进
4	$ (S)	$	用$S \rightarrow \varepsilon$ 归约
5	$ (S) S	$	用$S \rightarrow$ (S) S归约
6	$ S	$	用$S' \rightarrow S$归约
7	$ S'	$	接受

㊀ 由于相同的原因，自顶向下的分析程序可被称作生成-匹配分析程序，但这并未成为惯例。

㊁ 在归约的情况中，可如同在自顶向下分析中为一个生成动作所做的一样，仅仅由 BNF选择自身写出即可，但是惯例却是添加*reduce*。

例5.2 考虑以下基本算术表达式的扩充文法(没有括号，只有一种运算)：

$$E' \rightarrow E$$

$$E \rightarrow E + n \mid n$$

表5-2是串$n + n$使用这个文法的自底向上的分析。

表5-2 **例**5.2**中文法的自底向上分析程序的分析动作**

	分 析 栈	输 入	动 作
1	$	$n + n$ $	移进
2	$ n	$+ n$ $	用$E \rightarrow n$归约
3	$ E	$+ n$ $	移进
4	$ $E +$	n $	移进
5	$ $E + n$	$	用$E \rightarrow E + n$归约
6	$ E	$	用$E' \rightarrow E$归约
7	$ E'	$	接受

在使用先行上，自底向上的分析程序比自顶向下的分析程序的困难要小。实际上，自底向上的分析程序可将输入符号移进到栈里直到它判断出要执行的是何种动作为止 (假设判断一个动作可以不要求将符号移进到输入中)。但是为了判断要完成的动作，自底向上的分析程序会需要在栈内看得更深，而不仅仅是顶部。例如，在表 5-1中的第5行，S位于栈的顶部，且分析程序用产生式$S \rightarrow (S)$ S归约，而第6行在栈的顶部也有S，但是分析程序却用$S' \rightarrow S$归约。为了能够了解$S \rightarrow (S)$ S在第5步中是一个有效的归约，我们必须知道实际上栈在该点包含了串(S) S。因此，自底向上的分析要求任意的“ 栈先行 ”。由于分析程序本身建立了栈且可使恰当的信息成为可用的，所以这几乎同输入先行一样重要。此时所用的机制是“ 项 ”的确定性的有穷自动机，下一节将会讲到它。

当然，仅仅了解栈的内容并不足以可唯一地判断出移进 - 归约分析的下一步，还需将输入中的记号作为一个先行来考虑。例如在表 5-2的第3行，E在栈中，且发生了一个移进；但在第 6 行中，E又在栈中，但却用了$E' \rightarrow E$ 归约。两者的区别在于：在第 3行中，输入的下一个记号是“ + ”；但在第6行中，下一个输入记号却是$。因此，任何执行那个分析的算法必须使用下一个输入记号(先行)来判断出适当的动作。不同的移进 - 归约分析方法以不同的方式使用先行，而这将导致具有不同能力和复杂性的分析程序。在看到单个算法之前，我们首先做一些普通的观察来看看自底向上分析的直接步骤是如何表现特征的。

首先，我们再次注意到移进 - 归约分析程序描绘出输入串的最右推导，但推导步骤的顺序却是颠倒的。在表5-1中，共有4个归约，它们与最右推导的4个步骤对应的顺序相反：

$$S' \quad S \quad (S) \quad S\,(S) \quad ( )$$

在表5-2中，相对应的推导是

$$E' \quad E \quad E + n \quad n + n$$

我们将这样的推导中的终结符和非终结符的每个中间串都称作右句型 (right sentential form)。在移进 - 归约分析中，每个这样的句型都被分析栈和输入分隔开。例如，发生在前面推导中的第3步的右句子格式$E + n$ 出现在表5-2的第3、第4和第5步。如果通过符号 ‖ 指出每一时刻栈的顶部位于何处(即：当在栈和输入之间发生了分隔)，则表5-2的第3步就由E ‖ $+ n$给出，而其第4步则由$E +$ ‖ n给出。在每一种情况下，分析栈的符号序列都被称作右句型的可行前缀

(viable prefix)。因此，E、$E+$ 和$E + n$ 都是右句型$E + n$ 的可行前缀，但右句子格式$n + n$ 却使 ε 和n 作为它的可行前缀(表5-2的第1步和第2步)。请注意，$n +$ 不是$n + n$ 的可行前缀。

移进-归约分析程序将终结符从输入移进到栈直到它能执行一个归约以得到下一个右句子格式。它发生在位于栈顶部的符号串匹配用于下一个归约的产生式的右边。这个串、它在右句子格式中发生的位置以及用来归约它的产生式被称作右句型的句柄 (handle)[⊖]。例如，在右句子格式$n + n$ 中，它的句柄是由最左边的单个记号n 与用来归约它以产生新的右句型$E + n$的产生式$E \to n$ 组成的串。这个新句型的句柄是整个串 $E + n$ (一个可行的前缀)以及产生式$E \to E + n$。有时由于表示法上的弊端，我们要用串本身来作为句柄。

判断分析中的下一个句柄是移进-归约分析程序的主要任务。请注意，句柄串总是为它的产生式(在下一步的归约中使用的产生式)构成一个完整的右部，而且当归约发生时，句柄串的最右边的位置将与栈的顶部相对应。所以对于移进-归约分析程序将要基于产生式右边的位置来判断它的动作这一点而言，就看起来有些似是而非了。当这些位置到达产生式的右边末端时，这个产生式就有可能是一个归约，而且句柄还有可能位于栈的顶部。但为了成为句柄，串位于栈的顶部来匹配产生式的右边并不够。实际上，若 ε 产生式可用于归约的话（如在例 5.1中)，那么它的右边(空串)总是位于栈的顶部。归约仅发生在结果串实际为一个右句型时。例如，在表5-1的第3步中，可完成用$S \to \varepsilon$ 归约，但是得到的串 $(S\ S)$并不是右句子格式，所以ε 在句子格式(S)中的这个位置也不是句柄。

5.2 LR(0)项的有穷自动机与LR(0)分析

5.2.1 LR(0)项

上下文无关文法的**LR(0)**项(LR(0) item)(或简写为项(item))是在其右边带有区分位置的产生式选择。我们可用一个句点 (当然它就变成了元符号，而不会与真正的记号相混淆)来指出这个区分的位置。所以若 $A \to \alpha$是产生式选择，且若 β和γ 是符号的任何两个串（包括空串 ε)，且存在着βγ =α，那么$A \to \beta.\gamma$ 就是LR(0)项。之所以称作 LR(0)项是由于它们不包括先行的显式引用。

例5.3 考虑例5.1的文法：

$$S' \to S$$
$$S \to (\ S\ )\ S \mid \varepsilon$$

这个文法存在着3个产生式选择和8个项目：

$$S' \to .S$$
$$S' \to S.$$
$$S \to .(\ S\ )\ S$$
$$S \to (.S\ )\ S$$
$$S \to (\ S.)S$$
$$S \to (\ S\ ).S$$
$$S \to (\ S\ )S.$$

⊖ 如果文法有二义性，那么由此就会存在多于一个的推导，则在右句型中就会有多于一个的句柄。如果文法没有二义性，则句柄就是唯一的。

$$S\rightarrow .$$

例5.4 例5.2的文法存在着以下的8个项目：

$$E'\rightarrow .E$$
$$E'\rightarrow E.$$
$$E\rightarrow .E + n$$
$$E\rightarrow E. + n$$
$$E\rightarrow E + .n$$
$$E\rightarrow E + n.$$
$$E\rightarrow .n$$
$$E\rightarrow n.$$

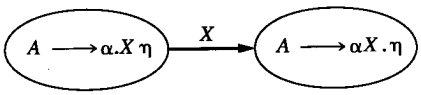
项目概念的思想就是指项目记录了特定文法规则右边识别中的中间步骤。特别地，项目 $A\rightarrow\beta.\gamma$ 是由文法规则选择$A\rightarrow\alpha$构成（其中$\alpha=\beta\gamma$），这一点意味着早已看到了β，且可能从下一个输入记号中获取γ。从分析栈的观点来看，这就意味着 β必须出现在栈的顶部。项目 $A\rightarrow.\alpha$意味着将要利用文法规则选择$A\rightarrow\alpha$识别A(将这样的项目称作初始项(initial item)。项目$A\rightarrow\alpha.$意味着α现在位于分析栈的顶部，而且若$A\rightarrow\alpha$在下一个归约中使用的话，它有可能就是句柄（将这样的项目称作完整项（complete item））。

5.2.2 项目的有穷自动机

LR(0)项可作为一个保持有关分析栈和移进 - 归约分析过程的信息的有穷自动机的状态来使用。这将从作为非确定性的有穷自动机开始。从这个 LR(0)项的NFA中可利用第2章的子集构造来构建出LR(0)项集合的DFA。正如将要看到的一样，直接构造 LR(0)项集合的DFA也是很简单的。

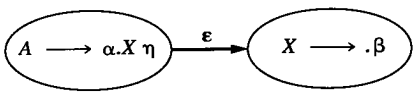
LR(0)项的NFA的转换是什么呢？若有项目 $A\rightarrow\alpha.\gamma$，且假设γ 以符号$X$开始，其中$X$可以是记号或非终结符，所以该项目就可写作 $A\rightarrow\alpha X.\eta$。那么在符号$X$上就有一个从由该项目代表的状态到由项目$A\rightarrow\alpha X.\eta$代表的状态的转换。把它写在一般格式中，就是：

若X是一个记号，那么该转换与 X的一个在分析中从输入到栈顶部的移进相对应。另一方面，



若X是一个非终结符，因为 X永远不会作为输入符号出现，所以该转换的解释就复杂了。实际上，这样的转换仍与在分析时将 X压入到栈中相对应，但是它只发生在由产生式 $X\rightarrow\beta$形成的归约时。由于这样的归约前面必须有一个 β的识别，而且由初始项$X\rightarrow.\beta$给出的状态代表了这个处理的开始(句点指出将要识别一个β)，则对于每个项目$A\rightarrow\alpha.X\eta$，必须为X的每个产生式$X\rightarrow\beta$添加一个ε 产生式，

它指示通过识别它的产生式的右边的任意匹配来产生 X。



这两种情况只表示 LR(0)项的NFA中的转换，它并未讨论到 NFA的初始状态和接受状态。

NFA的初始状态应与分析程序的初始状态相对应：栈是空的，而且将要识别一个 S，其中S 是文法的开始符号。因此，任何由S的产生式选择构造的初始项$S\rightarrow.\alpha$都可作为开始状态。不幸的是，S可能有许多这样的产生式，如何知道该选择哪一个呢？实际上，我们是无法做到这一点的。其解决办法是通过产生式 $S'\rightarrow S$ 扩充(augment)文法，其中S'是一个新的非终结符。接着，S' 成为扩充文法(augmented grammar)的开始状态，且初始项$S'\rightarrow.S$ 也成为NFA的开始状态。这就是为什么扩充前面例子文法的原因。

在这个NFA中，哪些状态将成为接受状态呢？读者此时必须记住 NFA的任务是用于了解分析的状态，而不是如第2章中自动机那样被设计为完全识别串。因此，分析程序本身将决定何时接受，而NFA则无需包含这一信息，所以 NFA实际上根本就没有接受状态 (NFA存在着有关接受的一些信息，但并不是在接受状态中。在描述使用自动机制分析算法时还会讨论到这一点)。

这样一来，LR(0)项的NFA的描述就完整了。现在来讨论前面两个例子的简单文法以及构造它们的LR(0)项的NFA。

例5.5 5.3列出了例5.1的文法的8个LR(0)项，所以这个NFA就有8个状态，如图5-1所示。请注意，图中在非终结符S之前带有一个句点的每个项目都有一个到S的每个初始项的ε- 转换。

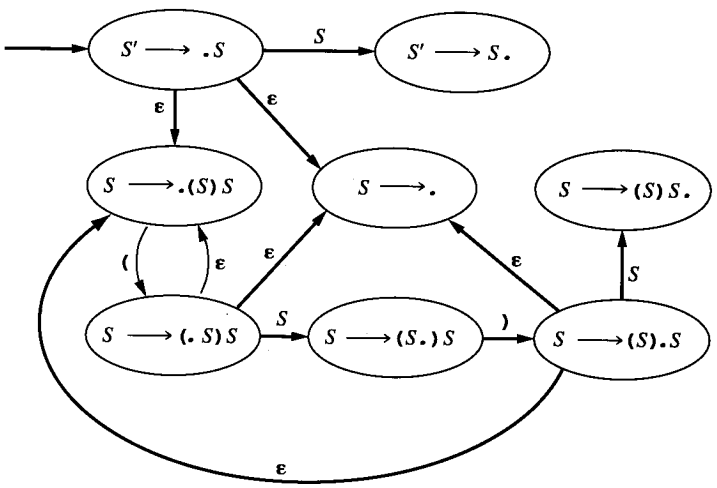


图5-1 例5.5中文法的LR(0)项的NFA

例5.6 在例5.4中，我们列出了与例5.2中的文法相关的LR(0)项。图5-2是这些项目的NFA。请注意，初始项$E\rightarrow.E + n$ 有一个到它自身的ε- 转换(这种情况会出现在带有直接左递归的所有文法中)。

为了完成项目用于了解分析状态的描述，我们必须根据第 2章的子集构造来构建与项目的NFA相对应的项目集合的DFA，这样之后就可说明LR(0)分析算法了。我们将执行前面刚给出的NFA的两个例子的子集构造。

例5.7 考虑图5-1的NFA。相关的DFA的开始状态是由项目$S'\rightarrow.S$组成的集合的ε-闭包，而这又是3个项目的集合{ $S'\rightarrow.S$, $S\rightarrow.$ **(** S **)** S, $S\rightarrow.$ }。由于S有一个从$S'\rightarrow .S$到$S'\rightarrow S.$的转换，所以也就存在着一个从开始状态到DFA状态{ $S'\rightarrow S.$ }的对应转换(不存在从$S'\rightarrow S.$到任何其他项目的ε

转换)。在(上也有一个从开始状态到DFA状态的转换{ S→ (. S) S, S→.(S) S, S→. }({ S→ (. S) S }的ε 闭包)。DFA状态{ S→ (. S) S, S→ . (S) S, S→ . }在(上有到其自身的转换，在S上也有到{ S→ (S.) S }的转换。这个状态在(上有到状态{ S→ (S). S, S→ .(S) S, S → . }的转换。最后，这一最终状态在(上有到前面所构造的状态{ S→ (. S) S, S→ .(S) S, S → . }的转换。图5-3是完整的DFA，为了便于引用还给各个状态编了号(按照惯例，状态0是开始状态)。

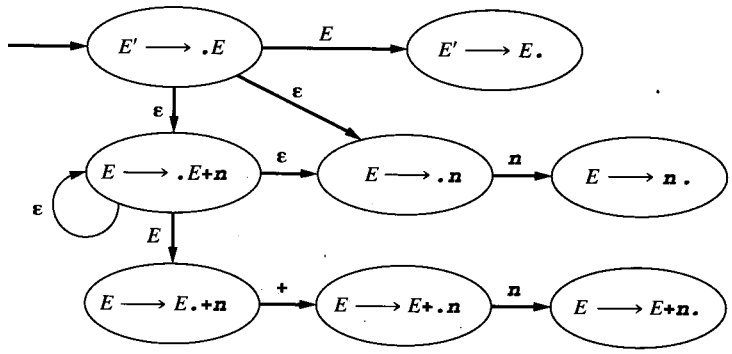


图5-2 例5.6中文法的LR(0)项的NFA

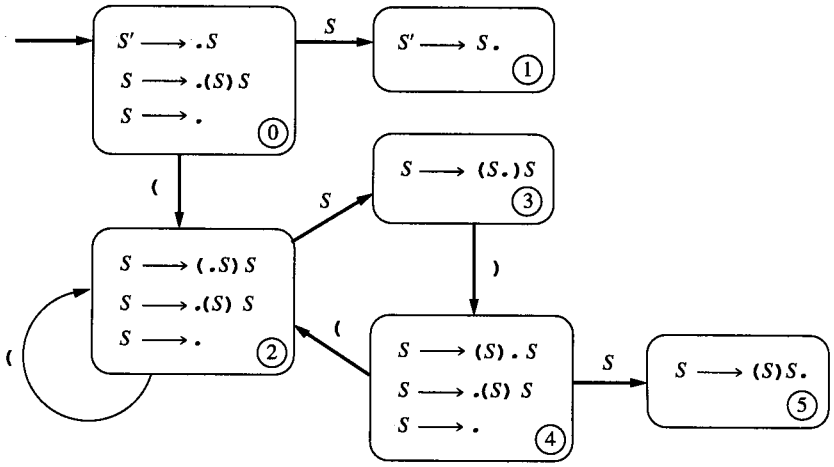


图5-3 与图5-1的NFA相对应的项目集合的DFA

例5.8 考虑图5-2的NFA。相关DFA的开始状态由3个项目的集合{*E*′→.*E*, *E*→.*E* + ***n***, *E* →.***n***}组成。在*E*上有一个从项目*E*′→.*E*到项目*E*′→ *E*.的转换，但在*E*上还有一个从项目*E*→.*E* + ***n*** 到项目*E*→*E*. + ***n***的转换。因此，在*E*上就有一个从DFA的开始状态到集合{ *E*′→ *E*., *E*→*E*. + ***n***}的闭包的转换。由于没有任何一个来自这些项目的 ε 转换，所以这个集合就是它自身的 ε 闭包，且构成了一个完整的 DFA状态。来自开始状态还有另一个转换，它与符号 ***n*** 上的从*E*→.***n***到*E* → ***n***.的转换相对应。因为没有来自项目 *E*→***n***.的ε 转换，所以这个项目是它自身的 ε 闭包，且构成了一个完整的DFA状态{ *E*→ ***n***. }。由于没有来自这个状态的转换，所以被计算的唯一转换仍来自于集合{*E*′→ *E*., *E*→*E*. + ***n***}。从该集合开始只有一个转换，它与符号 +上的从项目*E* →*E*. + ***n*** 到项目*E*→*E* +.***n*** 的转换相对应。项目*E*→*E* +.***n*** 也没有ε 转换，所以就在DFA中构造了一个单独的集合。最后， ***n*** 上有一个从集合{ *E*→*E* +.***n***}到集合{ *E* →*E* + ***n***. }的转换。图5-4

中是整个DFA。

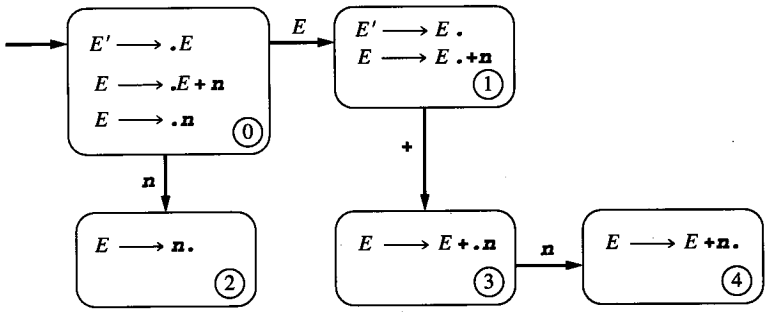


图5-4 与图5-2中的NFA相对应的项目集合的DFA

在LR(0)项集合的DFA中，有时需要区分在ε 闭包步骤中添加到状态中的项目与引起状态作为非ε- 转换的目标的那些项目。前者被称作闭包项 (closure item)，后者被称作核心项 (kernel item)。在图5-4的状态0中，$E'\rightarrow .E$是(唯一的)核心项，而$E\rightarrow .E + n$ 和$E\rightarrow .n$ 则是闭包项。在图5-3的状态2中，$S\rightarrow (.S)S$是核心项，而$S\rightarrow .(S)S$和$S\rightarrow .$是闭包项。根据项目的NFA的ε 转换的定义，所有的闭包项都是初始项。

区分核心项与闭包项的重要性在于：若有一个文法，核心项唯一地判断出状态以及它的转换，那么只需指出核心项就可以完整地表示出项目集合的 DFA来。因此，构造DFA的分析程序生成器只报告核心项(例如对于Yacc而言，这也是正确的)。

若项目集合的DFA是直接运算的，这样就比首先计算项目的 NFA之后再应用子集构造要更简化一些。从项目的集合确实可很容易地立即判断出什么是ε-转换以及初始项指向哪里。因此，如Yacc的分析程序生成器总是从文法直接计算 DFA，本章后面的内容也是这样做的。

5.2.3 LR(0)分析算法

现在开始讲述LR(0)分析算法。由于该算法取决于要了解项目集合的 DFA的当前状态，所以须修改分析栈以使不但能存储符号而且还能存储状态数。这是通过在压入一个符号之后再将新的状态数压入到分析栈中完成的。实际上，状态本身就包含了有关符号的所有信息，所以可完全将符号省掉而只在分析栈中保存状态数。但是为了方便和清晰起见，我们仍将在栈中保留了符号。

为了能开始分析，我们将底标记 $和开始状态0压入到栈中，所以分析在开始时的状况表示为：

分 析 栈	输 入
$ 0	*InputString* $

现在假设下一步是将记号**n** 移进到栈中并进入到状态2(当DFA如在图5-4中所示一样，且**n**是输入中的下一个记号时，结果就是这样的)。表示如下：

分 析 栈	输 入
$ 0 **n** 2	*InputString* $ 的剩余部分

LR(0)分析算法根据当前的DFA状态选择一个动作，这个状态总是出现在栈的顶部。

定义：LR(0)分析算法(LR(0) parsing algorithm)。令s为当前的状态(位于分析栈的顶部)。则动作定义如下：

1. 若状态s包含了格式$A\rightarrow\alpha.X\beta$的任何项目，其中$X$是一个终结符，则动作就是将当前的输入记号移进到栈中。若这个记号是X，且状态s包含了项目$A\rightarrow\alpha.X\beta$，则压入到栈中的新状态就是包含了项目$A\rightarrow\alpha X.\beta$的状态。若由于位于刚才所描述的格式的状态$s$中的某个项目，这个记号不是$X$，则声明一个错误。
2. 若状态s包含了任何完整的项目(格式$A\rightarrow\alpha.$的一个项目)，则动作是用规则$A\rightarrow\alpha$归约。假设输入为空，用规则$S'\rightarrow S$归约(其中S是开始状态)与接受相等价；若输入不为空，则出现错误。在所有的其他情况中，新状态的计算如下：将串α及它的所有对应状态从分析栈中删去(根据DFA的构造方式，串α必须位于栈的顶部)。相应地，在DFA中返回到由α开始构造的状态中（这须是由α的删除所揭示的状态）。由于构造DFA，这个状态就还须包含格式$B\rightarrow\alpha.A\beta$的一个项目。将$A$压入到栈中，并压入包含了项目$B\rightarrow\alpha A.\beta$的状态（作为新状态）。(请注意，由于正将$A$压入到栈中，所以这与在DFA中跟随$A$的转换相对应(这实际上是合理的)。

若以上的规则都是无歧义的，则文法就是LR(0)文法(LR(0) grammar)。这就意味着若一个状态包含了完整项目$A\rightarrow\alpha.$，那么它就不能再包含其他项目了。实际上，若这样的状态还包含了一个“移进的”项目$A\rightarrow\alpha.X\beta$（$X$是一个终结符），就会出现一个到底是执行动作(1)还是执行动作(2)的二义性。这种情况称作移进-归约冲突(shift-reduce conflict)。类似地，如果这样的状态包含了另一个完整项目$B\rightarrow\beta.$，那么也会出现一个关于为该归约使用哪个产生式($A\rightarrow\alpha$或$B\rightarrow\beta$）二义性。这种情况称作归约-归约冲突(reduce-reduce conflict)。所以，当仅当每个状态都是移进状态(仅包含了“移进”项目的状态)或包含了单个完整项目的归约时，该文法才是LR(0)。

我们注意到上面例子中所用到的两个文法都不是LR(0)文法。在图5-3中，状态0、状态2和状态4都包括了对于LR(0)分析算法的移进-归约冲突；在图5-4的DFA中，状态1包含了一个移进-归约冲突。由于几乎所有“真正的”文法都不是LR(0)，所以这并不奇怪。但下面将会给出一个文法是LR(0)的示例。

例5.9 考虑文法

$$A\rightarrow (\ A\ )\mid \mathbf{a}$$

扩充的文法具有如图5-5所示的项目集合的DFA，而这就是LR(0)。为了看清LR(0)分析算法是如何工作的，可考虑一下串**((a))**。该串的分析是根据表5-3各步骤所给出的LR(0)分析算法进行。分析开始于状态0，因为这个状态是一个移进状态，所以将第1个记号**(**移进到栈中。接着，由于DFA指出了在符号**(**上从状态0到状态3的转换，所以将状态3压入到栈中。状态3也是一个移进状态，所以下一个**(**也被移进到栈中，而且在**(**上的转换返回到状态3。移进再一次将**a**放入到栈中，而且**a**上的转换由状态3进入到状态2。现在位于表5-3的第4步，而且已到达了第1个归约状态。这里的状态2和符号**a**都是由栈弹出的，并回到处理中的状态3。接着，将A压入到栈中，且得到由状态3到状态4的A转换。状态4是一个移进状态，所以**)**被移进到栈中，且**)**上的转换使分析转到状态5。这里发生了一个由规则$A\rightarrow$ **(** A **)** 进行的归约，它从栈中弹出状态5、状态4、状态3以及符号**)**、A和**(**。现在的分析位于状态3中，而且A和

状态4又被压入到栈中。)再一次被移进到栈中，且压入状态 5。由*A*→ (*A*) 进行的另一个归约从栈中(向后地)删除了串(3 *A* 4) 5，而将分析留在状态0中。现在*A*已被压入且也得到了由状态0到状态1的*A*转换。状态1是接受状态。由于输入现在是空的，则分析算法承认它。

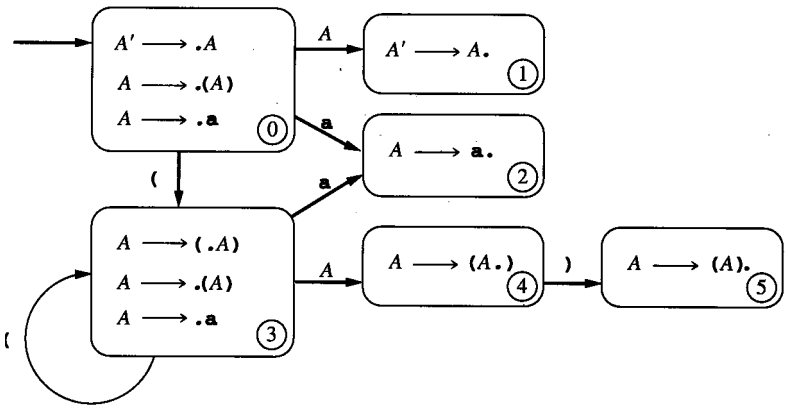


图5-5 例5.9的项目集合的DFA

表5-3 例5.9的分析动作

	分 析 栈	输 入	动 作
1	$0	((a)) $	移进
2	$0 (3	(a)) $	移进
3	$0 (3 (3	a)) $	移进
4	$0 (3 (3 a 2	)) $	用*A*→ a归约
5	$0 (3 (3 *A* 4	))$	移进
6	$0 (3 (3 *A* 4) 5	) $	用*A*→ (*A*) 归约
7	$0 (3 *A* 4	) $	移进
8	$0 (3 *A* 3) 5	$	用*A*→ (*A*) 归约
9	$0 *A* 1	$	接受

也可将项目集合的 DFA以及由LR(0)分析算法指定的动作合并到分析表中，所以 LR(0)分析就变成一个表驱动的分析方法了。其典型结构是：表的每行用 DFA的状态标出，而每一个列也如下标出。由于 LR(0)分析状态是“ 移进的 ”状态或“ 归约的 ”状态，所以专门利用一列来为每个状态指出它。若是“ 归约的 ”状态，就用另一个列来指出在归约中所使用的文法规则选择。若是移进的状态，将被移进的符号会判断出下一个状态 (通过DFA)，所以每个记号都会有一个列，而这些记号的各项都是有关该记号移进后将移到的新状态。由于不能在输入中真正地看到它们，所以尽管分析程序的行为好像是它们被移进了，非终结符上的转换 (在归约时被压入)仍代表着一个特殊情况。因此，在“ 移进的 ”状态中还需要为每个非终结符有一个列，而且按照惯例，这些列被列入到称为 goto部分的表的一个单独部分中。

表5-4就是这样的分析表的一个例子，它是例 5.9中文法的表格。读者可以证实这个表将引出如在表5-3中给出的示例的分析动作。

表5-4 例5.9中的文法的分析表

状 态	动 作	规 则	输 入			Goto
			(	**a**	)	*A*
0	移进		3	2		1
1	归约	*A*′→ *A*				
2	归约	*A*→ **a**				
3	移进		3	2		4
4	移进				5	
5	归约	*A*→ (*A*)				

在这样的分析表中的空白项表示的是错误。当必须要进行错误恢复时，则需要准确地指出分析程序要为每个这些空白项采取什么动作。后面一节将讨论这个问题。

5.3 SLR(1)分析

5.3.1 SLR(1)分析算法

简单LR(1)分析，或SLR(1)分析，也如上一节中一样使用了LR(0)项目集合的DFA。但是，通过使用输入串中下一个记号来指导它的动作，它大大地提高了LR(0)分析的能力。它通过两种方法做到这一点。首先，它在一个移进之前先考虑输入记号以确保存在着一个恰当的DFA。其次，它使用如4.3节所构造的非终结符的Follow集合来决定是否应执行一个归约。令人吃惊的是，先行的这个简单应用的能力强大得足以分析几乎所有的一般的语言构造。

定义：SLR(1)分析算法(SLR(1) parsing algorithm)。令s为当前状态(位于分析栈的顶部)。则动作可定义如下：

1. 若状态s包含了格式$A\rightarrow\alpha.X\beta$的任意项目，其中$X$是一个终结符，且$X$是输入串中的下一个记号，则动作将当前的输入记号移进到栈中，且被压入到栈中的新状态是包含了项目$A\rightarrow\alpha X.\beta$的状态。
2. 若状态s包含了完整项目$A\rightarrow\gamma.$，则输入串中的下一个记号是在Follow(A)中，所以动作是用规则$A\rightarrow\gamma$归约。用规则$S'\rightarrow S$归约与接受等价，其中S是开始状态；只有当下一个输入记号是$时，这才会发生㊀。在所有的其他情况中，新状态都是如下计算的：删除串α和所有它的来自分析栈中的对应状态。相对应地，DFA回到α开始构造的状态。通过构造，这个状态必须包括格式$B\rightarrow\gamma.A\beta$的一个项目。将$A$压入到栈中，并将包含了项目$B\rightarrow\alpha A.\beta$的状态压入。
3. 若下一个输入记号都不是上面两种情况所提到的，则声明一个错误。

若上述的SLR(1)分析规则并不导致二义性，则文法为**SLR(1)**文法(SLR(1) grammar)。特别地，当且仅当对于任何状态s，以下的两个条件：

1) 对于在s中的任何项目$A\rightarrow\alpha.X\beta$，当$X$是一个终结符，且$X$在Follow ($B$) 中时，$s$中没有完整的项目$B\rightarrow\gamma.$。

2) 对于在s中的任何两个完整项目$A\rightarrow\alpha.$和$B\rightarrow\beta.$，Follow(A) ∩ Follow(B)为空。

㊀ 实际上，任何文法扩充的开始状态S'的Follow集合总是只由$组成，这是因为$S'$只出现在文法规则$S'\rightarrow S$中。

均满足时，文法为SLR(1)。

若第1个条件不满足，就表示这是一个移进-归约冲突 (shift-reduce conflict)。若第2个条件不满足，就表示这是一个归约-归约冲突(reduce-reduce conflict)。

这两个条件同前一章中所述的 LL(1)分析的两个条件在本质上是类似的。但是如同使用所有的移进-归约分析方法一样，可将决定使用哪个文法规则推迟到最后，同时还可考虑一个更强大的分析程序。

SLR(1)分析的分析表也可以用与前一节所述的 LR(0)分析的分析表的类似方式构造。两者的差别如下：由于状态在SLR(1)分析程序中可以具有移进和归约 (取决于先行)，输入部分中的每项现在必须要有一个“移进”或“归约”的标签，而且文法规则选择也必须被放在标有“归约”的项中。这还使得动作和规则列成为多余。由于输入结束符号 $也可成为一个合法的先行，所以必须为这个符号在输入部分建立一个新的列。我们将 SLR(1)分析表的构造放在SLR(1)分析的第1个示例中。

例5.10 考虑例5.8中的文法，它的项目集合的DFA已列在了图5-4中。正如前面所述的，这个文法不是LR(0)，而是SLR(1)。非终结符的Follow集合是Follow(E') = {$}和Follow ($E$) = {$, +}。表5-5是SLR(1)分析表。在表中，移进由表项中的字母 s 指出，归约由字母 r 指出。因此，在输入+的状态1中，指出了一个移进，以及一个到状态 3的转换。另一方面，在输入 +的状态2中，指出了利用产生式$E \rightarrow n$ 归约。在输入$的状态1中还用动作“接受”代替了 r ($E \rightarrow E$)。

表5-5 例5.10的SLR(1)分析表

状 态	输 入			Goto
	n	+	$	E
0	s2			1
1		s3	接受	
2		r ($E \rightarrow n$)	r ($E \rightarrow n$)	
3	s4			
4		r ($E \rightarrow E + n$)	r ($E \rightarrow E + n$)	

这个示例的最后是串 $n + n + n$ 的分析。表5-6是它的分析步骤。该图的步骤1以输入记号 n 的状态0开始，接着分析表指出动作“s2”，即：将记号移进到栈中并进入到状态2。在表5-6中，将它与阶段“shift 2”一起指出来。在该图的步骤2中，分析程序是在状态2中且带有输入记号 +，表还指出了用规则 $E \rightarrow n$ 归约。此时，从栈中弹出状态2和记号 n。使状态0曝露出来。将符号 E 压入且将 E 的Goto从状态0带到状态1。第3步中的分析程序是带有输入记号 +的状态1，且表还指出了移进以及指向状态 3的转换。在输入 n 的状态3中，表也指出了一个移进和到状态 4的转换。在输入+的状态4中，表指出用规则 $E \rightarrow E + n$ 归约。这个归约是由将串 $E + n$ 和与它相结合的来自栈的状态弹出来完成的，并再一次暴露状态 0，将 E 压入并将Goto带到状态1中。分析的其他步骤是类似的。

表5-6 例5.10的分析动作

	分 析 栈	输 入	动 作
1	$ 0	$n + n + n$	移进2

（续）

	分 析 栈	输 入	动 作
2	$ 0 ***n*** 2	***+ n + n*** $	用*E*→***n***归约
3	$ 0 *E* 1	***+ n + n*** $	移进3
4	$ 0 *E* 1 ***+*** 3	***n + n*** $	移进4
5	$ 0 *E* 1 ***+*** 3 ***n*** 4	***+ n*** $	用*E*→*E* ***+ n*** 归约
6	$ 0 *E* 1	***+ n*** $	移进3
7	$ 0 *E* 1 ***+*** 3	***n*** $	移进4
8	$ 0 *E* 1 ***+*** 3 ***n*** 4	$	用*E*→*E* ***+ n*** 归约
9	$ 0 *E* 1	$	接受

例5.11　考虑成对括号的文法，图 5-3已给出了它的LR(0)项的DFA。一个直接的运算生成了Follow (*S*′) = {$}和Follow (*S*) = {$, **)**}。表5-7给出了它的SLR(1)分析表。请读者注意，非LR(0)状态0、2和4是如何通过ε- 产生式*S*→ε 具有移进和归约动作的。表 5-8给出了SLR(1)分析算法用来分析串( ) ( )的步骤。请注意，栈如何继续扩展到最终的归约。这是自底向上的分析程序在诸如*S*→ (*S*) *S*的右递归规则中的一个特征。因此，右递归可引起栈的溢出，所以若可能的话应尽量避免。

表5-7　**例**5.11**的**SLR(1)**分析表**

状 态	输 入			Goto
	(	**)**	$	*S*
0	s2	r (*S*→ε)	r (*S*→ε)	1
1			接受	
2	s2	r (*S*→ε)	r (*S*→ε)	3
3		s4		
4	s2	r (*S*→ε)	r (*S*→ε)	5
5		r (*S*→ **(** *S* **)** *S*)	r (*S*→ **(** *S* **)** *S*)	

表5-8　**例**5.11**的分析动作**

	分 析 栈	输 入	动 作
1	$ 0	**( ) ( )** $	移进2
2	$ 0 **(** 2	**) ( )** $	用*S*→ε 归约
3	$ 0 **(** 2 *S* 3	**( )** $	移进4
4	$ 0 **(** 2 *S* 3 **)** 4	**( )** $	移进2
5	$ 0 **(** 2 *S* 3 **)** 4 **(** 2	**)** $	用*S*→ε 归约
6	$ 0 **(** 2 *S* 3 **)** 4 **(** 2 *S* 3	**)** $	移进4
7	$ 0 **(** 2 *S* 3 **)** 4 **(** 2 *S* 3 **)** 4	$	用*S*→ε 归约
8	$ 0 **(** 2 *S* 3 **)** 4 **(** 2 *S* 3 **)** 4 *S* 5	$	用*S*→ **(** *S* **)** *S*归约
9	$ 0 **(** 2 *S* 3 **)** 4 *S* 5	$	用*S*→ **(** *S* **)** *S*归约
10	$ 0 *S* 1	$	接受

5.3.2 用于分析冲突的消除二义性规则

SLR(1)分析中以及所有的移进-归约分析方法中的分析冲突都可分为两类：移进-归约冲突和归约-归约冲突。在移进-归约冲突中，有一个自然的消除二义性规则，它总是选取移进而不是归约，因此大多数的移进-归约分析程序通过选择移进来取代归约，也就自动地解决了移进-归约冲突。但是归约-归约冲突就要复杂一些了：这样的冲突通常 (但并不是总是)指出文法设计中的一个错误(后面将给出这样冲突的示例)。在移进-归约冲突中选取移进取代归约自动地合并了用于在if语句中的悬挂else二义性的最近嵌套规则，例5.12就是这样的一个例子。这是为什么在程序设计语言的文法中仍保留有二义性的一个原因。

例5.12 考虑在前面章节中所用到的简化了的if语句的文法(例如可参见第3章的3.4.3节)：

statement → *if-stmt* | **other**
if-stmt → **if (** *exp* **)** *statement*
| **if (** *exp* **)** *statement* **else** *statement*
exp → **0** | **1**

由于这是一个有二义性的文法，所以在任何的分析算法中都会出现分析冲突。为了在 SLR(1)分析程序中看到这一点，我们将文法再简化一些使得项目集合的 DFA的构造更易处理。甚至还可将测试表达式全部省略，那么文法就如下所示(它仍包含了悬挂else二义性)：

$S \rightarrow I \mid \mathbf{other}$
$I \rightarrow \mathbf{if}\ S \mid \mathbf{if}\ S\ \mathbf{else}\ S$

图5-6是项目集合的DFA。构造SLR(1)分析动作需要S 和I 的Follow集合，它们是

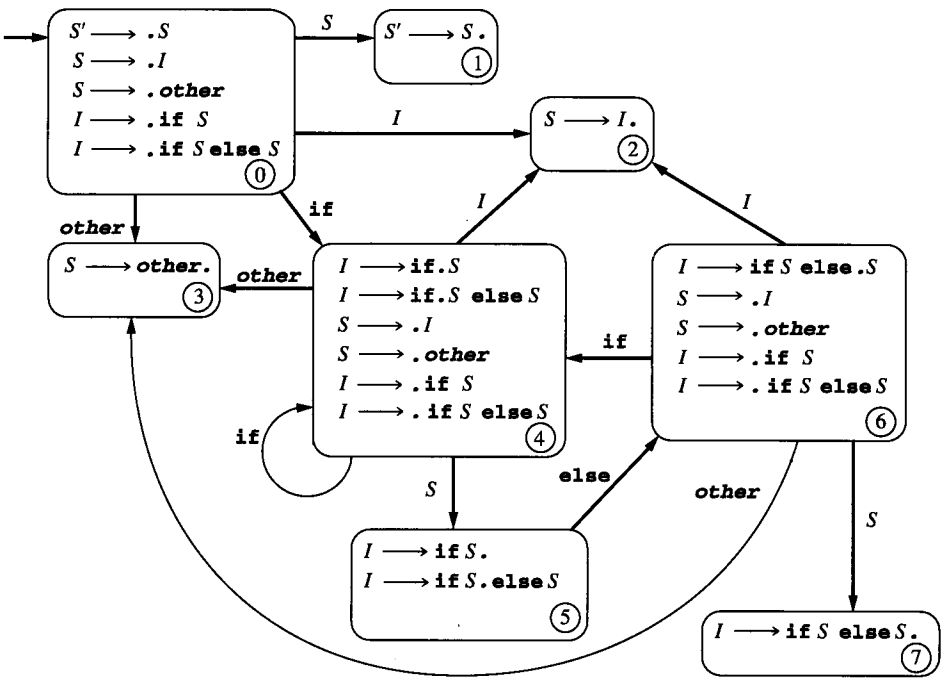


图5-6 例5.12中LR(0)项目集合的DFA

Follow (*S*) = Follow (*I*) = {$, **else** }

现在就可以看到由悬挂else问题引起的分析冲突了。当发生在DFA的状态5中时，其中的完整项目$I \rightarrow$ **if** S.指出规则$I \rightarrow$ **if** S的归约将发生在输入**else**和$中，但项目$I \rightarrow$ **if** S. **else** S 却指出输入记号的一个移进将发生在 **else**上。因此悬挂else将导致在SLR(1)分析表的移进-归约冲突。很明显，用移进取代归约的消除二义性的规则可以消除这个冲突，并会根据最近嵌套规则作出分析(若用归约取代移进，就没有办法在 DFA中输入状态6或状态7，这将导致虚假的分析错误)。

表5-9是由该文法引出的SLR(1)分析表。在该表中为归约动作中的文法规则选择使用了编号，用它来代替写出规则本身。编号如下：

(1) $S \rightarrow I$
(2) $S \rightarrow$ ***other***
(3) $I \rightarrow$ **if** S
(4) $I \rightarrow$ **if** S **else** S

请注意，无需为扩充产生式 $S' \rightarrow S$编号，这是由于用该规则实现的归约与接受相对应，且在表中已被写作“ 接受 ”了。

读者应注意到在归约项中使用的产生式编号容易引起与在移进和 Goto项中所用到的编号混淆。例如，在表5-9的表的状态5中，输入**else**下的项目是s6，它指出一个移进以及到状态6的转换，但在输入$下的项目却是r3，它指出用产生式编号3实现的归约(即：$I \rightarrow$ **if** S)。

表5-9还为了移进而删除了移进-归约冲突。我们将表中的项目渐渐减少以显示出在何处发生了冲突。

表5-9　例5.12的SLR(1)分析表(删除了分析冲突)

状　态	输　入				Goto	
	if	**Else**	**other**	**$**	*S*	*I*
0	s4		s3		1	2
1				接受		
2		r1		r1		
3		r2		r2		
4	s4		s3		5	2
5	s6		r3			
6	s4		s3		7	2
7		r4		r4		

5.3.3　SLR(1)分析能力的局限性

SLR(1)分析是LR(0)分析的一个简单但有效的扩展，而LR(0)分析的能力足以处理几乎所有实际的语言结构。不幸的是，在有些情况下， SLR(1)分析能力并不太强，而正由于这个原因，我们还需要学习更强大的一般的 LR(1)和LALR(1)分析。下一个例子是SLR(1)分析失败的典型情况。

例5.13　考虑语句的以下文法，它是从Pascal 中抽取和简化而得来的（在C中也有类似的情况

发生）：

stmt → *call-stmt* | *assign-stmt*
call-stmt → ***identifier***
assign-stmt →*var* **:=** *exp*
var → *var* [exp] | ***identifier***
exp → *var* | ***number***

这个文法模块语句既可是调用无参数的过程也可以是对变量的表达式的赋值。请注意，赋值和过程调用都是以一个标识符开头。只有到看到语句的结尾或记号 **:=** 时，分析才会决定该语句是一个赋值还是一个调用。将这种情形简化成以下的文法，在其中删去了变量的替换选择，并将语句选项简化为无需改变基本的情况：

S→ ***id*** | *V* **:=** *E*
V→ ***id***
E→ *V* | ***n***

为了显示这个文法在SLR(1) 分析中是如何引起一个分析冲突，请考虑项目集合的 DFA的开始状态：

S′→.*S*
S→**.*id***
S→.*V* **:=** *E*
V→**.*id***

这个状态在***id*** 上有一个到状态

S→***id*.**
V→***id*.**

的转换。现在有Follow (*S*) = {$}和Follow (*V*) = { **:=**, $ }(由于有规则*V*→*V* **:=** *E*所以有**:=**，又由于*E*可以是*V*，所以有$)。因此，SLR(1)分析算法要求在这个状态中有一个在输入符号$下的利用规则*S*→***id*** 和规则*V*→***id*** 实现的归约(这是一个归约-归约冲突)。这个分析冲突实际上是一个由SLR(1)方法的缺点所引起的“假冒”问题。实际上当输入为$时，用*V*→***id*** 实现的归约永远也不应该在这个状态中，这是由于只有到看到记号**:=**和被移进后，变量才会出现在语句的末端。

在下面的两节中，读者将会看到如何利用更强大的分析方法来解决这个分析问题。

5.3.4 SLR(k)文法

同其他分析算法一样， SLR(1)分析算法可被扩展为SLR(*k*)分析，其中的分析动作是基于*k*≥1个先行的符号之上。利用上一章定义的集合 $First_k$ 和$Follow_k$，Slr(*k*)分析程序使用以下两个规则：

1) 若状态*s* 包含了格式*A*→α.*X*β（*X*是一个记号），且*Xw* $First_k$ (*X*β)是输入串中之后的*k* 个记号，那么该动作就是将当前输入记号移进到栈中，而且被压入到栈中的新状态是包含了项目*A*→α*X*.β的状态。

2) 若状态*s* 包含了完整项目*A*→α.，且*w* $Follow_k$ (*A*)是输入串中之后的*k* 个记号，则动作用规则*A*→α归约。

当*k* >1时， SLR(*k*)分析比SLR(1)分析更强大，但由于分析表的大小将按 k的指数倍增长，

所以它又要复杂许多。非SLR(1)的典型语言构造可利用LRLA(1)分析程序处理得更好一些，它可使用标准的消除二义性的规则，或将文法重写。虽然例 5.13的简单非SLR(1)文法确实也可出现在SLR(2)中，但对于为任意值的k 而言，它所来自的程序设计语言问题却不是SLR(k)。

5.4 一般的LR(1)和LALR(1)分析

本节将研究LR(1)分析的最一般格式，有时它称作 LR(1)规范(canonical)分析。这种方法解决了上一节最后所提到的SLR(1)分析中出现的问题，但它却复杂得多。实际上在绝大多数情况下，通常地，一般的 LR(1)分析太复杂以至于不能在大多数情况下的分析程序的构造中使用。幸运的是，一般的LR(1)分析的一个修正——称作LRLA(1)(即“先行”LR分析)在保留了LR(1)分析的大多数优点之外还保留了 SLR(1)方法的有效性。LALR(1)方法已成为诸如用于诸如 Yacc 这样的分析程序生成器所选用的方法，本节稍后将会研究到它。但为了理解这个方法，首先应学习普通方法。

5.4.1 LR(1)项的有穷自动机

SLR(1)中的困难在于它在 LR(0)项的DFA的构造之后提供先行，而构造却又忽略了先行。一般的LR(1)方法的功能在于它使用了一个从开始时就将先行构建在它的构造中的新 DFA。这个DFA使用了LR(0)项的扩展的项目。由于它们包括了每个项目中的一个先行记号，所以就称作**LR(1)**项(LR(1) item)。说得更准确一些就是：LR(1)项应是由LR(0)项和一个先行记号组成的对。利用中括号将LR(1)项写作

$$[A\rightarrow\alpha.\beta\ a]$$

其中$A\rightarrow\alpha.\beta$是一个LR(0)项，而a 则是一个记号(先行)。

为了完成一般LR(1)分析所用的自动机的定义，我们需要首先定义 LR(1)项之间的转换。它们与LR(0)转换相类似，但它们还知道先行。同LR(0)项一样，它们包括了ε- 转换，此外还需建立一个DFA，它的状态是项目为ε- 闭包的集合。LR(0)自动机和LR(1)自动机的主要差别在于ε-转换的定义。我们首先给出较简单的情况下(非ε-转换)的定义，它们与LR(0)的情况基本一致。

定义：LR(1)转换(第1部分)的定义(definition of LR(1) transitions (part 1))。假设有LR(1)项目$[A\rightarrow\alpha.X\gamma,\ a]$，其中$X$是任意符号(终结符或非终结符)，那么$X$就有一个到项目$[A\rightarrow\alpha X.\gamma,\ a]$的转换。

请注意在这种情形下，两个项目中都出现了相同的先行 a，所以这些转换并不会引起新的先行的出现。只有ε- 转换才“创建”新的先行，如下所示。

定义：LR(1)转换(第2部分)的定义(definition of LR(1) transitions (part 2))。假设有LR(1)项目$[A\rightarrow\alpha.B\gamma,a]$，其中$B$是一个非终结符，那么对于每个产生式 $B\rightarrow\beta$和在First (γa) 中的每个记号b都有到项目$[B\rightarrow.\beta,b]$ 的ε- 转换。

请读者留意，这些ε- 转换是如何跟踪在其中结构B需要被识别的上下文。实际上项目 $[A\rightarrow\alpha.B\gamma,a]$说明了在分析的这一点上可能要识别 B，但这只有是当这个B后跟有一个从串γa 派生出的串时，且这样的串须以一个在First (γa)中的记号开始才可能。由于串γ 跟随在位于产生式$A\rightarrow\alpha B\gamma$ 中的B之后，所以若a是被构造在Follow (A)中，那么有First (γa) $\subset$ Follow (B)，且在项目[B

→.β,*b*] 中的*b* 将总是在Follow (*B*)中。一般的LR(1)方法的功能在于集合First (γ*a*) 可能是Follow (*B*)的一个恰当的子集(SLR(1)分析程序本质上从整个Follow集合中得到先行b)。请读者再注意，仅当γ 可派生出空串时，最初的先行*a* 才可作为*b* 中的一个元素。在大多数情况下(尤其是在实际情况中)，只有γ 当本身是ε 时它才发生，此时从格式[*A*→α.*B*,*a*]到[*B*→.β, *a*]可得到ε- 转换的特殊情况。

对LR(1)项目集合的DFA的构造还包括指定开始状态。这是通过如同在 LR(0)中一样用新的开始符号*S*′ 和新的产生式*S*′→*S*(其中*S*是最初的开始符号)来扩充文法。接着， LR(1)项目的NFA的开始符号就成为了项目[*S*′→.*S*,$]，其中$代表结尾标记(且是Follow (*S*′) 中的唯一符号)。这就有效地说明了是由识别一个从*S*派生出的串开始，其后则是$符号。

现在来看一下有关LR(1)项目的DFA的构造的一些示例。

例5.14 考虑例5.9中的文法

$$A \rightarrow \ (\ A\ )\ |\ \mathbf{a}$$

首先通过扩充文法以及构造初始的 LR(1)项目[*A*′→.*A*, $]来构建它的LR(1)项目集合的DFA。这个项目的ε- 闭包是DFA的开始状态。由于在这个项目中没有任何符号跟随在*A*之后(按照前面所讨论的有关转换的术语，串γ 就是ε)，就有到项目[*A*→. (*A*), $]和[*A*→.**a**, $]的ε- 转换(按照前面讨论过的有关术语，即为First (γ$) = {$})。接着，开始状态(状态0)就是这3个项目的集合：

状态0：
$$[A' \rightarrow .A,\ \$]$$
$$[A \rightarrow .\ (\ A\ ),\ \$]$$
$$[A \rightarrow .\ \mathbf{a},\ \$]$$

在*A*上有从这个状态到包含了项目[*A*′→*A*., $]的集合的闭包的转换，而且由于没有来自完整项目的转换，所以这个状态仅包含了单个项目 [*A*′→*A*., $]。除此之外再也没有来自这个状态的转换了，所以将它编号为状态1：

状态1： $[A' \rightarrow A.,\ \$]$

(这是LR(1)分析算法将从中生成接受动作的状态)。

再回到状态0上，在记号(之上也有一个到由项目[*A*→. (*A*), $]组成的集合的闭包。由于有来自这个项目的ε-转换，这个闭包也就并不是微不足道的了。实际上从这个项目上也有到项目[*A*→.(*A*),)]和[*A*→.**a**,)]的ε-转换。这是因为在项目[*A*→ (.*A*), $]中，在括号的上下文的右边识别*A*。也就是，右边*A*的Follow是First ( )$) = {)}，因此在这种情况下，就得到了一个新的先行记号，而且新的DFA状态是由以下的项目组成：

状态2：
$$[A \rightarrow (\ .A\ ),\ \$\ ]$$
$$[A \rightarrow .(\ A\ ),\ )\ ]$$
$$[A \rightarrow .\ \mathbf{a},\ )\ ]$$

再回到状态0，在其中找到由项目[*A*→**a** . , $]生成的状态的最后转换。由于这是一个完整的项目，所以它是一个项目的状态：

状态3： $[A \rightarrow \mathbf{a}\ .,\ \$\ ]$

现在回到状态2。在*A*上有一个从这个状态到[*A*→ (*A* .), $]的ε- 闭包的转换，它是带有一个项目的状态：

状态4： $[A \rightarrow (\ A.\ ),\ \$]$

在(上也有一个到[A→ (. A),)]的ε-闭包的转换。在这里也可根据与在状态2的构造中相同的原理，生成闭包项目[A→.(A),)]和[A→.a,)]。所以就得到新的状态：

状态5： [A→ (. A),)]
[A→ . (A),)]
[A→.a,)]

请注意，这个状态除了第1个项目的先行之外，其他都与状态2相同。

最后，在记号a上有一个从状态2到状态6的转换：

状态6： [A→a.,)]

请读者再次注意，这与状态3几乎一样，只在先行上略有不同。

具有一个转换的下一个状态是状态4，它有一个在记号) 上到状态

状态7： [A→ (A) ., $]

的转换。

再回到状态5，这里在 (上有一个从这个状态到它本身的转换，还有一个在A上的到状态

状态8： [A→ (A .),)]

的转换，以及一个在a上到早已构造好的状态6的转换。

最后，在)上有一个从状态8到

状态9： [A→ (A) .,)]

的转换。

因此，这个文法的LR(1)项目的DFA具有10个状态。图5-7是完整的DFA。将它与相同文法的LR(0)项目集合的DFA相比较(参见图5-5),我们发现LR(1)项目的DFA几乎是它的两倍大小。这也是正常的。实际上通过在带有多个先行记号的复杂情形中的 10因子， LR(1)状态的数量可超过LR(0)状态的数量。

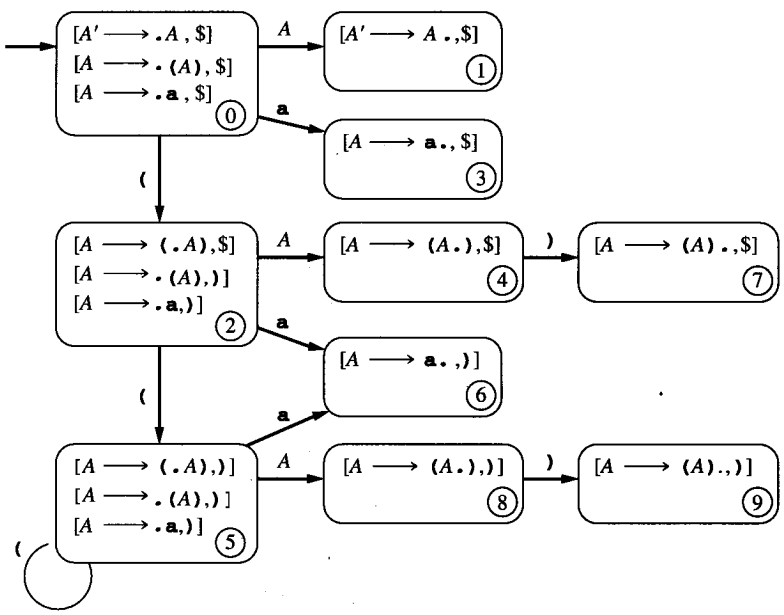


图5-7 例5.14的LR(1)项目集合的DFA

5.4.2 LR(1)分析算法

在考虑其他示例之前，我们需要先根据新的 DFA构造通过重新叙述分析算法而将一般的LR(1)分析进行完善。由于仅需重述 SLR(1)分析算法而无需使用 LR(1)项目中的先行记号代替Follow集合，所以这是容易办到的。

一般的LR(1)分析算法 令s 为当前状态(位于分析栈的顶部)，则动作定义如下：

① 若状态s 包含了格式$[A\rightarrow\alpha.X\beta, a]$的任意LR(1)项目，其中$X$是一个终结符且是输入串中的下一个记号，则动作就是将输入记号移进到栈中，且被压入到栈中的新状态是包含了 LR(1)项目$[A\rightarrow\alpha X.\beta, a]$的状态。

② 若状态s 包含了完整的LR(1)项目$[A\rightarrow\alpha., a]$，且输入串中的下一个记号是a，则动作就是用规则$A\rightarrow\alpha$归约。用规则$S'\rightarrow S$ (其中S是开始状态)实现的归约等价于接受(只有当下一个输入记号是$时才发生)。在其他情况下，新状态的计算如下：将串 α以及与它对应的所有状态从分析栈中删去。相应地 DFA返回到α开始构造的状态。通过构造，这个状态必须包括格式 $[B\rightarrow\alpha.A\beta, b]$的LR(1)项目。将$A$压入到栈中，并压入包含了项目$[B\rightarrow\alpha A.\beta, b]$ 的状态。

③ 若下一个输入记号不是上面所述的任何一种情况，则声明一个错误。

同使用前面的方法一样，若前面的一般 LR(1)分析规则的应用程序不引起二义性，则该文法就是**LR(1)**文法(LR(1) grammar)。特别地，当且仅当对于任何状态s，能够满足以下两个条件，该文法才是LR(1)文法：

① 对于在s 中的任何项目$[A\rightarrow\alpha.X\beta, a]$，且$X$是一个终结符，则在$s$ 中没有格式$[B\rightarrow\beta., X]$的项目(否则就有一个移进-归约冲突)。

② 在s 中没有格式$[A\rightarrow\alpha., a]$和$[B\rightarrow\beta., a]$的两个项目(否则就有一个归约-归约冲突)。

我们还注意到可从表达一般 LR(1)分析算法的LR(1)项目集合的DFA中构造出一个分析表。该表具有与SLR(1)分析程序的表格完全相同的格式，如下例所示。

例5.15 为表5-10中的例5.14的文法给出一般的LR(1)分析表。它可从图5-7的DFA中很方便地构造出来。在该表中，我们为在归约动作中的文法规则选择使用了以下的编号：

(1) $A\rightarrow(A)$

(2) $A\rightarrow\mathbf{a}$

因此在状态3中带有先行$的项r2指出了规则$A\rightarrow\mathbf{a}$ 实现的归约。

表5-10 例5.14的一般LR(1)分析表

状 态	输 入					Goto
	(	a	)	$	A	
0	s2	s3			1	
1					接受	
2	s5	s6				4
3					r2	
4			s7			
5	s5	s6				8
6			r2			
7				r1		
8			s9			
9			r1			

因为在一般的LR(1)分析中的特定串的分析步骤与它们在SLR(1)分析或LR(0)分析中的步骤相同，所以我们省略了一个这样的分析示例。由于也可从 DFA中很方便地得到，所以本节随后的例子也将省掉分析表的构造。

在实际中除非是有二义性的，几乎所有合理的程序设计语言的文法都是 LR(1)文法。当然也有可能构造出不能成为LR(1)文法的非二义性文法的示例来，但在这里就不这样做了 (参见练习)。在实践中人们总是试图避免它。实际上，程序设计语言甚至很少需要一般的 LR(1)分析所提供的能力。我们前面用来介绍LR(1)分析的示例(例5.14)其实早已是一个LR(0)文法了(而且因此也就是SLR(1)文法了)。

下面的示例表明一般的LR(1)分析解决了例5.13中不能成为SLR(1)的文法的先行问题。

例5.16　例5.13的文法在简化了的格式中是：

$$S \rightarrow \textbf{id} \mid V \textbf{:=} E$$
$$V \rightarrow \textbf{id}$$
$$E \rightarrow V \mid \textbf{n}$$

为这个文法构造LR(1)项目集合的DFA。其开始状态是项目[$S'\rightarrow .S$, \$]的闭包，它包括了项目[$S \rightarrow .\textbf{id}$, \$]和[$S \rightarrow .V$ **:=** E, \$]。因为$S \rightarrow .V$ **:=** E 指出可能识别了一个V，但是这只有是当它后面是一个赋值记号时，所以这个最后的项目还引起了闭包项目 [$V \rightarrow .\textbf{id}$, **:=**]。因此，记号**:=**作为初始项目$V \rightarrow .\textbf{id}$ 的先行出现。开始状态可总结为由以下的LR(1)项目组成：

状态0：　[$S'\rightarrow .S$, \$]
[$S \rightarrow .\textbf{id}$, \$]
[$S \rightarrow .V$ **:=** E, \$]
[$V \rightarrow .\textbf{id}$, **:=**]

在S上有一个从这个状态到一个项目的状态

状态1：　[$S'\rightarrow .S$, \$]

的转换，且在**id**上有一个到两个项目的状态

状态2：　[$S\rightarrow .\textbf{id}$, \$]
[$V \rightarrow .\textbf{id}$, **.=**]

的转换。在V上还有一个从状态0到一个项目的状态

状态3：　[$S \rightarrow .V$ **.:=** E, \$]

的转换。从状态1和状态2没有转换，但在**:=**上从状态3到项目[$S \rightarrow V$ **:=.** E, \$] 的闭包有一个转换。由于在这个项目中E之后没有符号，所以这个闭包包括了项目 [$E\rightarrow .V$, \$]和[$E \rightarrow .\textbf{n}$, \$]。最后，由于此时V之后也没有符号，项目[$E\rightarrow .V$, \$]可引起闭包项目[$V \rightarrow .\textbf{id}$,\$] (这与在状态0中的情形不同，在状态0中V后有一个赋值。而由于此时已经看到了一个赋值记号，所以这里的 V后不能有一个赋值)。那么最后的状态就是

状态4：　[$S \rightarrow V$ **:=** .E, \$]
[$E \rightarrow .V$, \$]
[$E \rightarrow .\textbf{n}$, \$]
[$V \rightarrow .\textbf{id}$, \$]

后面的状态和转换能够很容易地构造出来，我们将它留给读者。图 5-8中是LR(1)项目集合的完

整的DFA。

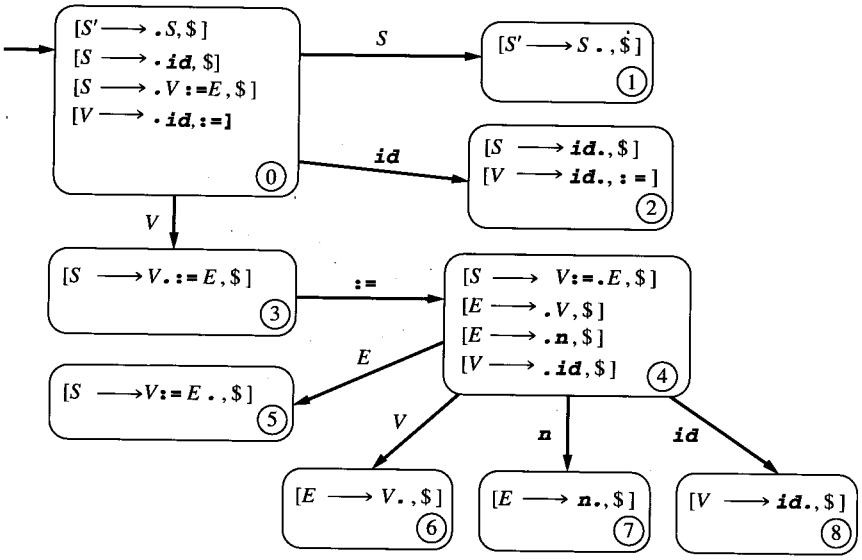


图5-8 例5.16中LR(1)项目集合的DFA

现在来考虑状态2。这是引起SLR(1)分析冲突的状态。LR(1)项目可由它的先行清晰地区分出两个归约：在**:=**上用规则$S \rightarrow$ ***.id*** 实现的归约和用$V \rightarrow$ ***.id*** 实现的归约。因此这个文法就是LR(1)文法。

5.4.3 LALR(1)分析

LALR(1)分析是基于以下的观察：在许多情况下， LR(1)项目集合的DFA的大小应部分地归于许多不同状态的存在，在这些状态的项目 (LR(0)项目)中，第1个成分的集合相同，只有第2个成分(先行符号)不同。例如，图5-7的LR(1)项目的DFA有10个状态，而相应的LR(0)项目的DFA(图5-5)只有6个。实际上在图5-7的状态2和5、状态4和8、状态7和9、状态3和6中，每对中的状态与其他的区别只在于它的项目的先行成分。例如状态 2和5：这两个状态只在其第1个项目上不同，且仅在那个项目的先行上：状态2有第1个项目[$A \rightarrow$ **(** A **)**, $]，且$作为它的先行；而状态5有第1个项目[$A \rightarrow$ **(** A **)**, **)**]，且**)**作为它的先行。

LALR(1)分析算法表明了它使得标识所有这样的状态和组合它们的先行有意义。在这样做时，我们总是必须以一个与LR(0)项目中的DFA相同的DFA作为结尾，但是每个状态都是以带有先行集合的项目组成。在完整项目的情况下，这些先行集合通常比相应的 Follow集合小；因此，LRLA(1)分析保留了LR(1)分析优于SLR(1)分析的一些特征，但是仍具有在LR(0)项目中的DFA尺寸较小的特点。

更正式地，在LR(1)项目的DFA中，状态的核心(core)是由状态中的所有LR(1)项目的第1个成分组成的LR(0)项目的集合。由于LR(1)项目的DFA的构造使用与在LR(0)项目的DFA的构造中相同的转换，但是它们在项目的先行部分上的作用不同，我们得到以下两个事实，它们构成了LALR(1)分析构造的基础。

(1) LALR(1)分析的第1个原则

LR(1)项目的DFA的状态核心是LR(0)项目的DFA的一个状态。

(2) LALR(1)分析的第2个原则

若有具有相同核心的LR(1)项目的DFA的两个状态s_1和s_2，假设在符号X上有一个从s_1到状态t_1的转换，那么在X上就还有一个从状态s_2到一个状态t_2的转换，且状态t_1和t_2具有相同的核心。

总之，这两个原则允许我们构造LALR(1)项目的DFA(DFA of LALR(1) items)，它是通过识别具有相同核心的所有状态以及为每个LR(0)项目构造出先行符号的并，而从LR(1)项目的DFA构造出来。因此，这个DFA的每个LALR(1)项目都将一个LR(0)项目作为它的第1个成分，并将一个先行记号的集合作为它的第2个成分⊖。在后面的示例中将在先行之间写一个/来表示多重先行。因此，LALR(1)项目[$A\rightarrow\alpha.\beta, a/b/c$]具有一个由符号$a$、$b$和$c$组成的先行集合。

我们给出一个展示这个构造的示例。

例5.17 考虑例5.14的文法，它的LR(1)项目的DFA在图5-7中。通过识别状态2和5、状态4和8、状态7和9、状态3和6，就可得出图5-9中的LALR(1)项目的DFA。在那个图中，我们保留了状态2、3、4和7的编号，并从状态5、6、8和9添加了先行。正如所期望的，除了先行之外，这个DFA与LR(0)项目的DFA相同。

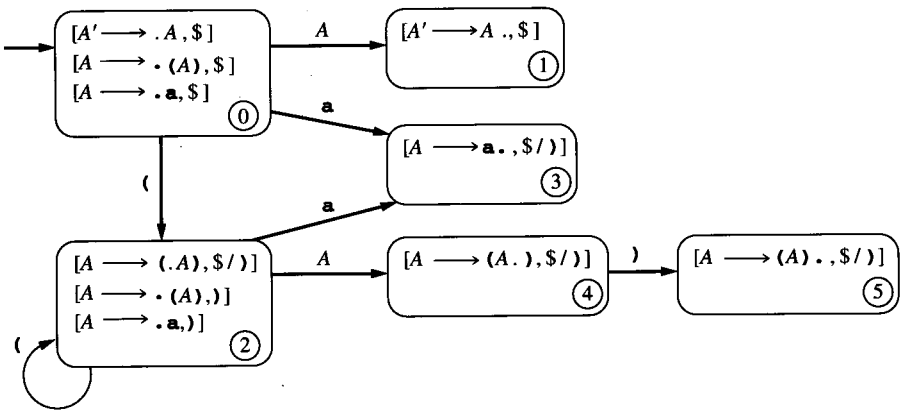


图5-9 例5.17的LALR(1)项目集合的DFA

使用了LALR(1)项目的压缩了的DFA的LALR(1)分析算法与上一节所描述的一般的LR(1)分析算法相同。同前所述，若在LALR(1)分析算法中没有出现分析冲突则称这个文法为LALR(1)文法(LALR(1) grammar)。在LALR(1)构造中，有可能制造出在一般的LR(1)分析中不存在的分析冲突来，但这在实际中却很少发生。实际上，若一个文法是LR(1)文法，则LALR(1)分析表就不能有任何的移进-归约冲突了；但是却有可能有归约-归约冲突(参见练习)。然而，若一个文法是SLR(1)，那么它肯定就是LALR(1)文法，而且LALR(1)分析程序在消除发生在SLR(1)分析中的典型冲突时通常与一般的LR(1)分析程序所做的相同。例如，例5.16的非SLR(1)文法是LALR(1)：图5-8的LR(1)项目的DFA也是LALR(1)项目的DFA。如在这个示例中一样，如果文法已经是LALR(1)了，则使用LALR(1)分析取代一般的LR分析的唯一结果是：在声明错误之前会作出一些虚假的归约。例如，从图5-9中可看出，若有错误的输入串**a**)，则LALR(1)分析程序将在声明错误之前执行归约$A\rightarrow$**a**，但是一般的LR(1)分析程序将在

⊖ LR(1)项目的DFA实际上还可以利用先行符号的集合代表共享它们的第1个成分的相同状态中的多重项目，但是我们发现为LALR(1)构造使用这种表示法也很方便，在这里它最适合。

移进记号**a**之后立即声明错误。

将LR(1)状态组合到LALR(1)项目的DFA解决了分析表尺寸较大的问题，但它仍要求计算LR(1)项目的整个DFA。实际上，通过传播先行 (propagating lookahead)的处理从LR(0)项目的DFA直接计算出LALR(1)项目的DFA是有可能的。尽管我们对此并不着意进行描述，但看看如何相对简单地做到它仍是不无好处的。

观察图5-9中的LALR(1)DFA。首先通过将结尾标记 $添加到状态0中的扩充项目$A' \rightarrow .A$ 的先行中($先行被称作被自发生成的(spontaneously generated))。接着再通过ε- 闭包的规则，将$传播到两个闭包项目 (核心项目$A' \rightarrow .A$ 的右边的A后是空串)。通过跟随状态0的3个转换，将$传播到状态1、3和2的核心项目。继续状态2，闭包项目得到先行)，再一次通过自发生成(因为位于核心项目$A' \rightarrow$ **(** $.A$ **)** 上的A在右括号之前)。现在在**a**上的到状态3的转换使得将)传送到那个状态中项目的先行 (上的从状态2到它本身的转换也使得) 。传送到核心项目的先行 (这就是为什么核心项目在它的先行集合中有 $和的原因)。现在先行集合$/传送到了状态4，之后再是到了状态7。因此通过这个处理就可直接从LR(0)项目的DFA得到图5-9中的LALR(1)的DFA了。

5.5 Yacc：一个LALR(1)分析程序的生成器

分析程序生成器(parser generator)是一个指定某个格式中的一种语言的语法作为它的输入，并为该种语言产生分析过程以作为它的输出的程序。在历史上，分析程序生成器被称作编译 - 编译程序(compiler-compiler)，这是由于按照规律可将所有的编译步骤作为包含在分析程序中的动作来执行。现在的观点是将分析程序仅考虑为编译处理的一个部分，所以这个术语也就有些过时了。合并 LALR(1)分析算法是一种常用的分析生成器，它被称作 Yacc(yet another compiler-compiler)。本节将给出Yacc的概貌来，下一节将使用 Yacc为TINY语言开发一个分析程序。由于Yacc有许多不同的实现以及存在着通常称作Bison的若干个公共领域版本⊖，所以在它的运算细节中有许多变化，而这又可能与这里所用的版本有些不同 ⊜。

5.5.1 Yacc基础

Yacc取到一个说明文件(通常带有一个**.y**后缀)并产生一个由分析程序的C源代码组成的输出文件(通常是在一个称作**y.tab.c**或**ytab.c**或更新一些的<文件名>**.tab.c**的文件中，而<文件名>**.y**则是输入文件)。Yacc说明文件具有基本格式

```
{definitions}
%%
{rules}
%%
{auxiliary routines}
```

因此就有3个被包含了双百分号的行分隔开来的部分——定义部分、规则部分和辅助程序部分。

定义部分包括了Yacc需要用来建立分析程序的有关记号、数据类型以及文法规则的信息。它还包括了必须在它的开始时直接进入输出文件的任何 C代码 (主要是其他源代码文件的**#include**指示)。说明文件的这个部分可以是空的。

⊖ 一个流行版本——Gnu Bison——是由Free Software Foundation发布的Gnu软件的一个部分，请参见“注意与参考”部分。

⊜ 实际上，我们已使用了若干个不同的版本来生成后面的示例。

规则部分包括修改的BNF格式中的文法规则以及将在识别出相关的文法规则时被执行的 C代码中的动作(即：根据LALR(1)分析算法，在归约中使用)。文法规则中使用的元符号惯例如下：通常，竖线被用作替换(也可分别写出替换项)。用来分隔文法规则的左右两边的箭头符号→在Yacc中被一个冒号取代了，而且必须用分号来结束每个文法规则。

第3点：辅助程序部分包括了过程和函数声明，除非通过 **#include**文件，否则它们会不适用，此外还需要被用来完成分析程序和 /或编译程序。这个部分也可为空，此时第 2个双百分号元符号可从说明文件中省略掉。因此，最小的 Yacc说明文件可仅由后面带有文法规则和动作(若仅是要分析文法也可省掉动作，本节稍后再讲到它)的%%组成。

Yacc还允许将C-风格的注解插入到说明文件的任何不妨碍基本格式的地方。

利用一个简单的示例，我们可将 Yacc说明文件的内容解释得更详细一些。这个示例是带有文法

exp → *exp addop term* | *term*
addop→ **+** | **-**
term → *term mulop factor* | *factor*
mulop → ***
factor → **(** *exp* **)** | ***number***

的简单整型算术表达式的计算器。这个文法在前面的章节中已用得很多了。在 4.1.2节中，我们为这个文法开发了一个递归下降计算器程序。程序清单 5-1给出了完全等价的Yacc说明。下面按顺序讨论这个说明中3个部分的内容。

程序清单5-1的定义部分有两个项目。第1个项目由要在Yacc输出的开始处插入的代码组成。这个代码是由两个典型的**#include**指示组成，且由在其前后的分隔符 **%{**和**%}**将它与这个部分中的其他Yacc说明分开(请注意百分号在括号之前)。定义部分的第2个项目是记号**NUMBER**的声明，它代表数字的一个序列。

程序清单5-1　一个简单计算器程序的Yacc定义

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%%

command : exp      { printf("%d\n",$1);}
        ; /* allows printing of the result */

exp     : exp '+' term   {$$ = $1 + $3;}
        | exp '-' term   {$$ = $1 - $3;}
        | term    {$$ = $1;}
        ;

term    : term '*' factor {$$ = $1 * $3;}
        | factor  {$$ = $1;}
        ;
```

```
factor      : NUMBER          {$$ = $1;}
            | '(' exp ')'     {$$ = $2;}
            ;
%%

main()
{ return yyparse();
}

int yylex(void)
{ int c;
  while((c = getchar()) == ' ');
  /* eliminates blanks */
  if ( isdigit(c) ) {
    ungetc(c,stdin);
    scanf("%d",&yylval);
    return(NUMBER);
  }
  if (c == '\n') return 0;
  /* makes the parse stop */
  return(c);
}

void yyerror(char * s)
{ fprintf(stderr,"%s\n",s);
} /* allows for printing of an error message */
```

Yacc用两种方法来识别记号。首先，文法规则的单引号中的任何字符都可被识别为它本身。因此，单字符记号就可直接被包含在这个风格的文法规则中，正如程序清单 5-1中的运算符记号**+**、**-**和***** (以及括号记号)。其次，可在Yacc的%记号(**%token**)中声明符号记号，如程序清单5-1中的记号**NUMBER**。这样的记号被Yacc赋予了不会与任何字符值相冲突的数字值。典型地，Yacc开始用数字258给记号赋值。Yacc将这些记号定义作为**#define**语句插入到输入代码中。因此，在输出文件中就可能会找到行

```
#define NUMBER 258
```

作为Yacc对说明文件中的**%token NUMBER**声明的对应。Yacc坚持定义所有的符号记号本身，而不是从别的地方引入一个定义。但是却有可能通过在记号声明中的记号名之后书写一个值来指定将赋给记号的数字值。例如，写出

```
%token NUMBER 18
```

就将给**NUMBER**赋值18 (不是258)。

在程序清单5-1的规则部分中，我们看到非终结符exp、term和factor的规则。由于还需要打印出一个表达式的值，所以还有另外一个称为 command的规则，而且将其与打印动作相结合。因为首先列出了command的规则，所以command则被作为文法的开始符号。若不这样，我们还可在定义部分中包括行

```
%start command
```

此时就不必将*command*的规则放在开头了。

Yacc中的动作是由在每个文法规则中将其写作真正的 C代码(在花括号中)来实现的。通常，尽管也有可能在一个选择中写出嵌入动作 (embedded action)(稍后将讨论它)，但动作代码仍是放在每个文法规则选择的末尾 (但在竖线或分号之前)。在书写动作时，可以享受到 Yacc伪变量

(pseudovariable)的好处。当识别一个文法规则时，规则中的每个符号都拥有一个值，除非它被参数改变了，该值将被认为是一个整型 (稍后将会看到这种情况)。这些值由Yacc保存在一个与分析栈保持平行的值栈 (value stack)中。每个在栈中的符号值都可通过使用以 $开始的伪变量来引用。**$$**代表刚才被识别出来的非终结符的值，也就是在文法规则左边的符号。伪变量**$1**、**$2**、**$3**等等都代表了文法规则右边的每个连续的符号。因此在程序清单 5-1中，文法规则和动作

```
exp    : exp '+' term { $$ = $1 + $3; }
```

就意味着当识别规则*exp*→*exp* **+** *term*时，就将左边的*exp* 的值作为*exp* 的值与右边的*term* 的值之和。

所有的非终结符都是通过这样的用户提供的动作来得到它们的值。记号也可被赋值，但这是在扫描过程中实现的。 Yacc假设记号的值被赋给了由 Yacc内部定义的变量**yylval**，且在识别记号时必须给**yylval**赋值。因此，在文法和动作

```
factor    : NUMBER    { $$ = $1;}
```

中，值**$1**指的是当识别记号时已在前面被赋值为**yylval**的**NUMBER**记号的值。

程序清单5-1的第3个部分(辅助程序部分)包括了3个过程的定义。第1个是**main**的定义，之所以包含它是因为 Yacc输出的结果可被直接编译为可执行的程序。过程 **main**调用**yyparse**，**yyparse**是Yacc给它所产生的分析过程起的名称。这个过程被声明是返回一个整型值。当分析成功时，该值总为 0；当分析失败时，该值为 1(即发生一个错误，且还没有执行错误恢复)。Yacc生成的**yyparse**过程接着又调用一个扫描程序过程，该过程为了与 Lex扫描程序生成器相兼容，所以就假设叫作**yylex** (参见第2章)。因此，程序清单5-1中的Yacc说明还包括了**yylex**的定义。在这个特定的情况下， **yylex** 过程非常简单。它所需要做的只有返回下一个非空字符；但若这个字符是一个数字，此时就必须识别单个元字符记号 **NUMBER**并返回它在变量**yylval**中的值。这里有一个例外：由于假设一行中输入了一个表达式，所以当扫描程序已到达了输入的末尾时，输入的末尾将由一个新行字符（在 C中的‘ **\n** ’）指出。Yacc希望输入的末尾通过**yylex**由空值0标出(这也是Lex所共有的一个惯例)。最后就定义了一个**yyerror**过程。当在分析时遇到错误时， Yacc就使用这个过程打印出一个错误信息 (典型地， Yacc打印串“ 语法错误 ”，但这个行为可由用户改变)。

5.5.2 Yacc选项

除了**yylex**和**yyerror**之外， Yacc通常需要访问许多辅助过程，而且它们经常是被放在外置的文件之中而非直接放在 Yacc说明文件中。通过写出恰当的头文件以及将 **#include**指示放在Yacc说明的定义部分中，就可以很容易地使 Yacc访问到这些过程。将 Yacc特定的定义应用到其他文件上就要复杂一些了，在记号定义时尤为如此。此时正如前面所讲的， Yacc坚持自己生成(而不是引入)，但是它又必须适用于编译程序的许多其他部分 (尤其是扫描程序)。正是由于这个原因， Yacc就有一个可用的选项，自动产生包含了该信息的头文件，而这个头文件将被包括在需要定义的任何其他文件中。这个头文件通常叫作 **y.tab.h**或**ytab.h**，并且它与**-d**选项(用于heaDer文件)一起生成。

例如，若文件**calc.y**包括了程序清单5-1中的Yacc说明，则命令

```
yacc -d calc.y
```

将产生(除了文件**y.tab.c**文件之外)内容各异的文件**y.tab.h**(或相似的名称)，但却总是包括

下示的内容：

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define     NUMBER     258
extern YYSTYPE yylval;
```

(稍后将详细地描述**YYSTYPE**的含义)。这个文件可被用来通过插入行

```
#include y.tab.h
```

到文件中而将**yylex**的代码放在另一个文件中⊖。

Yacc的另一个且极为有用的选项是详细选项(verbose option)，它是由命令行中的**-v**标志激活。这个选项也产生另一个名字为**y.output**(或类似名称的)的文件。这个文件包括了被分析程序使用的LALR(1)分析表的文本描述。阅读这个文件将使得用户可以准确地判断出 Yacc生成的分析程序在任何情况下将会采取的动作，而且这是处理文法中的二义性和不准确性的极为有效的方法。在向说明添加动作或辅助过程之前， Yacc与这个选项一起在文法上运行以确保 Yacc生成的分析程序将确实如希望的那样执行的确是一个好办法。

例如，程序清单5-2中的Yacc说明。这是程序清单5-1中Yacc说明的基本版本。当同时使用Yacc和详细选项：

```
yacc -v calc.y
```

程序清单5-2 使用-V选项的Yacc说明提纲

```
%token NUMBER
%%
command     : exp
            ;
exp         : exp '+' term
            | exp '-' term
            | term
            ;
term        : term '*' factor
            | factor
            ;
factor      : NUMBER
            | '(' exp ')'
            ;
```

时，这两个说明生成相同的输出文件。程序清单 5-3是该文法完整的典型**y.output**文件⊜。下面的段落将讨论如何解释这个文件。

Yacc输出文件列出了 DFA中的所有状态，此外还有内部统计的小结。状态由 0开始编号。输出文件在每个状态的下面列出了核心项目 (并未列出闭包项目)，其次是与各个先行对应的动作，最后则是各个非终结符的 goto动作。Yacc尤其使用一个下划线字符 _来标出项目中的显著

⊖ Yacc的早期版本可能只将记号的定义 (但没有**yylval**的定义)放置在**y.tab.h**中。这可能将要求一个共同的工作区或重新安排代码。

⊜ Bison的较新版本在输出文件中产生了一个根本不同的格式，但是内容却基本相同。

的位置，用它来代替本章所用到的句点，却用句点来指明缺省，或“不在意”每个状态列表的动作部分中的先行记号。

程序清单5-3 为程序清单5-1中的Yacc说明使用详细选项生成的典型y.output 文件

```
state 0
        $accept : _command $end

        NUMBER  shift 5
        (  shift 6
        .  error

        command  goto 1
        exp  goto 2
        term  goto 3
        factor  goto 4

state 1
        $accept :  command_$end

        $end  accept
        .  error

state 2
        command :  exp_    (1)
        exp :  exp_+ term
        exp :  exp_- term

        +  shift 7
        -  shift 8
        .  reduce 1

state 3
        exp :  term_    (4)
        term :  term_* factor

        *  shift 9
        .  reduce 4

state 4
        term :  factor_    (6)

        .  reduce 6

state 5
        factor :  NUMBER_    (7)

        .  reduce 7

state 6
        factor :  (_exp )

        NUMBER  shift 5
        (  shift 6
        .  error

        exp  goto 10
        term  goto 3
        factor  goto 4

state 7
        exp :  exp +_term

        NUMBER  shift 5
        (  shift 6
        .  error

        term  goto 11
        factor  goto 4

state 8
        exp :  exp -_term

        NUMBER  shift 5
        (  shift 6
        .  error

        term  goto 12
        factor  goto 4

state 9
        term :  term *_factor

        NUMBER  shift 5
        (  shift 6
        .  error

        factor  goto 13

state 10
     exp :  exp_+ term
     exp :  exp_- term
     factor :  ( exp_)

     +  shift 7
     -  shift 8
     )  shift 14
     .  error

state 11
     exp :  exp + term_    (2)
     term :  term_* factor

     *  shift 9
     .  reduce 2
```

```
state 12
     exp :  exp - term_    (3)
     term :  term_* factor

     *  shift 9
     .  reduce 3

state 13
     term :  term * factor_   (5)

     .  reduce 5

state 14
     factor :  ( exp )_    (8)

       .  reduce 8

8/127 terminals, 4/600 nonterminals
9/300 grammar rules, 15/1000 states
0 shift/reduce, 0 reduce/reduce conflicts reported
9/601 working sets used
memory: states, etc. 36/2000, parser 11/4000
9/601 distinct lookahead sets
6 extra closures
18 shift entries, 1 exceptions
8 goto entries
4 entries saved by goto default
Optimizer space used: input 50/2000, output 218/4000
218 table entries, 202 zero
maximum spread: 257, maximum offset: 43
```

Yacc通过列出扩充产生式的初始项目而从状态 0开始，而这通常在DFA的开始状态中只有核心项目。在上面示例的输出文件中，这个项目写作

```
$accept :_command $end
```

它与我们自己的术语中的项目 *command'→.command* 对应。Yacc为扩充的非终结符提供的名字为**$accept**。它还将输入结尾的伪记号显式地列为**$end**。

首先大致地看一下状态0的动作部分，它后面是核心项目的列表：

```
NUMBER shift 5
( shift 6
. error
command goto 1
exp goto 2
term goto 3
factor goto 4
```

上面的列表指出了DFA移进到先行记号**NUMBER**的状态5中、移进到先行记号 **(**的状态6中，并且说明其他所有先行记号中的错误。此外为了在归约到所给出的非终结符中使用还列出了 4个goto转换。这些动作与用这章中的方法手工构造分析表中的内容完全一样。

再看看状态2，它有输出列表

```
state 2
command : exp_ (1)
exp : exp_+ term
exp : exp_- term
+ shift 7
- shift 8
. reduce 1
```

这里的核心项目是一个完整的项目，所以在动作部分中有一个用相关的产生式选择实现的归约。为了提醒我们在归约中所使用的产生式的编号， Yacc在完整的项目之后列出了编号。在这种情

况下，产生式编号就是1，而且有一个表示用产生式 *command* → *exp*实现的**reduce 1**动作。Yacc总是按照它们在说明文件中所列的顺序为产生式编号。在我们的示例中，有 8个产生式(*command*的一个，*exp*的3个，*term*的两个以及*factor*的两个)。

请注意，在这个状态中的归约动作是一个缺省动作：一个将在任何不是 **+**或**-**的先行之上的归约。这里的Yacc与一个单纯的LALR(1)分析程序(而且甚至是SLR(1)分析程序)的不同在于它并不试着去检查归约上的合法先行(而是在若干个归约中决定)。Yacc分析程序将在最终声明错误之前在错误之上作出多个归约(这将是在任何的更多的移进发生之前的最后行为)。这就意味着错误信息可能不是像它们应该的那样有信息价值，但是分析表却会变得十分简单，这是因为情况发生得更少了(这点将在5.7节中再次讨论到)。

在这个示例最后，我们从Yacc输出文件中构造出一个分析表，该表与本章早些时候手工写出的完全一样。表5-11就是这个分析表。

表5-11 与程序清单5-3的Yacc输出对应的分析表

状 态	输 入							Goto			
	NUMBER	**(**	**+**	**-**	*****	**)**	$	*command*	*exp*	*term*	*factor*
0	s5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			s7	s8		S14					
11	r2	r2	r2	r2	s9	r2	r2				
12	r3	r3	r3	r3	s9	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				
14	r8	r8	r8	r8	r8	r8	r8				

5.5.3 分析冲突与消除二义性的规则

详细选项的一个重要用处是Yacc将在**y.output**文件中报告调查的分析冲突。Yacc在其中建立了消除二义性的规则，该规则将允许它甚至在分析冲突发生时产生一个分析程序 (因此，甚至这也是用于二义性文法的)。这些消除二义性的规则通常总能做对，但有时也会出错。对**y.output**文件的检查使用户判断出分析冲突是什么，以及由Yacc产生的分析程序是否可正确地解决问题。

程序清单5-1的示例中并没有分析冲突，在输出文件结尾的小结信息中， Yacc将它报告为

```
0 shift / reduce, 0 reduce / reduce conflicts reported
```

例5.12中的二义性悬挂else文法是一个更为有趣的示例。在表5-9中，我们为这个文法给出了SLR(1)分析表，在其中通过选取移进而不是归约将状态5中的移进-归约冲突消除了(它与最近嵌套消除二义性的规则对应)。Yacc以完全相同的术语报告了二义性，并且通过相同的消除二义性的规则解决了二义性。除了Yacc插入到错误项中的缺省归约之外，由Yacc报告的分析表确实与表5-9相同。例如， Yacc将状态5的动作在**y.output**文件中报告如下(记号在说明文件中的定义用小写字母写出，这样就避免了与C的保留字相冲突)：

```
5: shift / reduce conflict ( shift 6, red'n 3) on ELSE
state 5
      I : IF S_(3)
      I : IF S_ELSE S
      ELSE shift 6
      . reduce 3
```

在小结信息中， Yacc还报告了一个移进-归约冲突：

```
1 shift / reduce, 0 reduce / reduce conflicts reported
```

在归约-归约冲突的情况下， Yacc 通过执行由文法规则在说明文件中首先列出的归约来消除二义性。尽管它也会带来正确的分析程序，这仍与在文法中的错误很相似。下面是一个简单的示例。

例5.18 考虑以下的文法：

$$S \rightarrow A \mid B$$
$$A \rightarrow a$$
$$B \rightarrow a$$

由于单个合法串a有两个派生：$S \Rightarrow A \Rightarrow a$和$S \Rightarrow B \Rightarrow a$，所以这是一个有二义性的文法。程序清单5-4是这个文法完整的**y.output**文件。请注意，在状态4中的归约-归约冲突，它由在规则$B \rightarrow a$之前执行规则$A \rightarrow a$来解决。这样就导致了后面的这个规则永远不会用在归约之中(它很明确地指出文法的一个问题)。Yacc在最后报告了这项情况以及行

```
Rule not reduced : B : a
```

程序清单5-4 例5.18中文法的Yacc输出文件

```
   Rule not reduced:  B : a

state 0
      $accept : _S $end

      a  shift 4
      .  error

      S  goto 1
      A  goto 2
      B  goto 3

state 1
      $accept :  S_$end
```

```
        $end   accept
        .   error

state 2
        S :   A_     (1)

        .   reduce 1

state 3
        S :   B_     (2)

        .   reduce 2

4: reduce/reduce conflict (red'ns 3 and 4 ) on $end
state 4
        A :   a_     (3)
        B :   a_     (4)

        .   reduce 3

Rule not reduced:    B :   a

3/127 terminals, 3/600 nonterminals
5/300 grammar rules, 5/1000 states
0 shift/reduce, 1 reduce/reduce conflicts reported
...
```

除了前面已提到过的消除二义性的规则之外，Yacc为指定与一个有二义性的文法相分隔开的算符优先及结合性还具有特别的机制。它具有一些优点。首先，文法无需包括指定了结合性和优先权的显式构造，而这就意味着文法可以短一些和简单一些了。其次，相结合的分析表也可小一点且得出的分析程序更有效。

例如程序清单5-5中的Yacc说明。在那个图中，文法是用没有算符的优先权和结合性的具有二义性的格式书写。相反地，算符的优先权和结合性通过写出行

```
%left '+' '-'
%left '*'
```

在定义部分中给出来。这些行向 Yacc指出算符+和-具有优先权且是左结合的，而且运算符*是左结合且有比+和-更高的优先权(因为在说明中，它是列在这些算符之后)。在Yacc中，其他可能的算符说明是%right 和%nonassoc ("nonassoc" 意味着重复的算符不允许出现在相同的层次上)。

程序清单5-5　带有二义性文法和算符的优先权及结合性规则的简单计算器的 Yacc说明

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token NUMBER

%left '+' '-'
%left '*'
```

```
%%

command : exp       { printf("%d\n",$1);}
        ;

exp     : NUMBER        {$$ = $1;}
        | exp '+' exp   {$$ = $1 + $3;}
        | exp '-' exp   {$$ = $1 - $3;}
        | exp '*' exp   {$$ = $1 * $3;}
        | '(' exp ')'   {$$ = $2;}
        ;

%%
/* auxiliary procedure declarations as in Figure 5.10 */
```

5.5.4 描述Yacc分析程序的执行

除了在**y.output**文件中显示分析表的详细选项之外， Yacc生成的分析程序也可能打印出它的执行过程，这其中包括了分析栈的一个描述以及与本章早先给出的描述类似的分析程序动作。这是通过用符号定义的**YYDEBUG**编译**y.tab.c**(例如通过使用**-DYYDEBUG**编译选项)以及在需要描述信息的地方将Yacc整型变量**yydebug**设置为1。例如假设表达式**2+3**是输入，请将下面的行添加到程序清单5-1的**main**过程的开头

```
extern int yydebug;
yydebug = 1;
```

将会使得分析程序产生一个与程序清单 5-6相似的输出。我们希望读者利用表 5-11的分析表手工构造出分析程序的动作描述并将它与这个输出作一对比。

程序清单5-6 假设有输入2 + 3，利用yydebug为由程序清单5-1生成的Yacc分析程序描绘输出

```
Starting parse
Entering state 0
Input: 2+3
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0
Entering state 4
Reducing via rule 6, factor -> term
state stack now 0
Entering state 3
Next token is '+'
Reducing via rule 4, term -> exp
state stack now 0
Entering state 2
Next token is '+'
Shifting token '+', Entering state 7
Next token is NUMBER
Shifting token NUMBER, Entering state 5
Reducing via rule 7, NUMBER -> factor
state stack now 0 2 7
Entering state 4
Reducing via rule 6, factor -> term
```

```
state stack now 0 2 7
Entering state 11
Now at end of input.
Reducing via rule 2, exp '+' term -> exp
state stack now 0
Entering state 2
Now at end of input.
Reducing via rule 1, exp -> command
5
state stack now 0
Entering state 1
Now at end of input.
```

5.5.5 Yacc中的任意值类型

程序清单5-1使用与文法规则中的每个文法符号相关的 Yacc伪变量指出计算器的动作。例如，用写出**$$ = $1 + $3**将一个表达式的值设置为它的两个子表达式值之和(在文法规则exp → exp **+** term的右边的位置1和位置3)。由于这些值的Yacc缺省类型总是整型，所以只要处理的值是整型就可以了。但是若要用浮点值计算一个计算器，那就不合适了。在这种情况下，必须在说明文件中对 Yacc伪变量的值类型进行重定义。这个数据类型总是由 C预处理器符号**YYSTYPE**在Yacc中定义。重定义这个符号会恰当地改变 Yacc值栈的类型。因此若需要一个计算浮点值的计算器，就必须在Yacc说明文件的定义部分的括号**%{...%}**中添加行

```
#define YYSTYPE double
```

在更复杂的情况下，不同的文法规则就有可能需要不同的值。例如，假设希望将算符挑选的识别与用那些算符计算的规则分隔开，如在规则

$$exp \rightarrow exp\ addop\ term \mid term$$
$$addop \rightarrow \mathbf{+} \mid \mathbf{-}$$

(它们实际上是表达式文法的原始规则，我们把它们改变了以直接识别程序清单 5-1中的exp规则的算符)。现在addop必须返回算符(一个字符)，但exp必须返回计算的值(即一个**double**)，但这两个数据类型不同。我们所需要做的是将 **YYSTYPE**定义为**double**与**char**的联合。有两种方法可以办到。其一是在Yacc说明中利用Yacc的**%union**声明来直接声明一个联合：

```
%union { double val;
         char op; }
```

现在Yacc需要被告知每个非终结符的返回类型，这是利用定义部分中的**%type**指示来实现的：

```
%type <val> exp term factor
%type <op> addop mulop
```

请注意，在Yacc的**%type**声明中，联合域的名称是怎样被尖括号括起来的。接着将程序清单5-1的Yacc说明修改成按下开始(我们将详细的写法留在练习中)：

```
...
%token NUMBER

%union { double val;
         char op; }

%type <val> exp term factor NUMBER
%type <op> addop mulop
```

```
%%
comand : exp      { printf ( " %d\n " , $1 ) ; }
       ;
exp  : exp op term { swithc ( $2 ) {
                     case '+': $$ = $1 + $3; break;
                     case '-': $$ = $1 - $3; break;
                     }
                   }
     | term { $$ = $1; }
     ;
op : '+' { $$ '+'; }
   | '-'{ $$ '-'; }
   ;
```

第2种方法是在另一个包含文件 (例如，一个头文件)中定义一个新的数据类型之后再将YYSTYPE定义为这个类型。接着必须在相关的动作代码中手工地构造出恰当的值来。下一节中的TINY分析程序是它的一个示例。

5.5.6 Yacc中嵌入的动作

在分析时，有时需要在完整地识别一个文法规则之前先执行某个代码。例如考虑简单声明的情况：

decl → *typevar-list*
type → **int** | **float**
var-list → *var-list* **, id** | **id**

当识别var-list时，用当前类型(整型和浮点)将标签添加于每个变量标识符上。可按如下方法在Yacc中完成：

```
dec1  : type { current_type = $1 ; }
        var_list
      ;
type  : INT { $$ = INT_TYPE ; }
      | FLOAT { $$ = FLOAT_TYPE ; }
      ;
var_list : var_list ';' ID
           { setType (tokenString, current_type);}
         | ID
           { setType (tokenString, current_type);}
         ;
```

请注意，在**decl**规则中识别变量之前，一个嵌入的动作如何设置**current_type**。读者将在后面的章节中看到其他有关嵌入动作的示例。

Yacc将一个嵌入动作

```
A : B { /* embedded action */ } C ;
```

解释为与一个新的占位符非终结符相等价，且与在被归约时，执行嵌入动作的非终结符的 ε- 产生式等价：

```
A : B E C ;
E : { /* embedded action */ } ;
```

最后，在表5-12中小结了Yacc的定义机制以及已讨论过的一些内置名称。

表5-12 Yacc内置名称和定义机制

Yacc的内置名称	含义／用处
`y.tab.c`	Yacc输出文件名称
`y.tab.h`	Yacc生成的头文件，包含了记号定义
`yyparse`	Yacc分析例程
`yylval`	栈中当前记号的值
`yyerror`	由Yacc使用的用户定义的错误信息打印机
`error`	Yacc错误伪记号
`yyerrok`	在错误之后重置分析程序的过程
`yychar`	包括导致错误的先行记号
`YYSTYPE`	定义分析栈的值类型的预处理器符号
`yydebug`	变量，当由用户设置为1时则导致生成有关分析动作的运行信息
Yacc的定义机制	**含义／用处**
`%token`	定义记号预处理器符号
`%start`	定义开始非终结符符号
`%union`	定义和`YYSTYPE`，允许分析程序栈上的不同类型的值
`%type`	定义由一个符号返回的和类型
`%left %right %nonassoc`	定义算符的结合性和优先权(由位置)

5.6 使用Yacc生成TINY分析程序

TINY的语法已在3.7节中给出，且在4.4节中也已给出了一个手写的分析程序；我们希望读者能掌握这些内容。这里将描述Yacc说明文件`tiny.y`以及对全程定义`globals.h`的修改(我们已采用了一些方法将对其他文件的改变定为最小，这也将讲到)。整个`tiny.y`文件都列在了附录B的第4000行到第4162行中。

首先讨论TINY的Yacc说明中的定义部分。稍后将谈到标志YYPARSER(第4007行)。表示了Yacc在程序中的任何地方都需要的信息有4个`#include`文件(第4009行到第4012行)。定义部分有4个其他声明。第1个(第4014行)是`YYSTYPE`的定义，它定义了通过使Yacc分析过程为指向节点结构的指针返回的值(`TreeNode`本身被定义在`globals.h`中)，这样就允许了Yacc分析程序构造出一个语法树。第2个是全程savedName变量的定义，它被用作暂时储存要被插入到还没构造出的树节点中的标识符串，而此时已能在输入中看到这些串了(在TINY中只有在赋值中才需要)。变量`savedLineNo`也是被用作相同目的，所以那个恰当的源代码行数也将与标识符关联。最后，`savedTree`被用来暂时储存由`yyparse`过程产生的语法树(`yyparse`本身可以仅返回一个整型标志)。

下面讨论一下与TINY的每个文法规则相结合的动作(这些规则与第3章的程序清单3-1中所给出的BNF略有不同)。在绝大多数情况下，这些动作表示与该点上的分析树相对应的语法树的构造。特别地，需要从`util`包调用到`newStmtNode`和`newExpNode`来分配新的节点(这些已在4.4节中讲述过了)，而且也需要指派新树节点的合适的子节点。例如，与TINY的`write_stmt`(第4082ff行)相对应的动作如下所示：

```
write_stmt : WRITE exp
               { $$ = newStmtNode (WriteK);
                 $$ ->child [0] = $2;
               }
           ;
```

第1个指令调用**newStmtNode**并指派返回的值为**write_stmt**的值。接着**exp** (Yacc伪变量**$2**是指向将要被打印的表达式的树节点的指针)前面构造的值为write语句的树节点的第1个孩子。其他的语句和表达式的动作代码十分类似。

program、**stmt_seq**和**assign_stmt**的动作处理与每个这些构造相关的小问题。在**program**的文法规则中，相关的动作(第4029行)是

```
{savedTree = $1;}
```

它将**stmt_seq**构造的树赋给了静态变量**savedTree**。由于它使得语法树可由**parse**过程之后返回，所以这是必要的。

在**assign_stmt**的情况中，我们早已指出需要储存作为赋值目标的变量的标识符串，这样当构造节点时 (以及为了便于今后的描绘，还编制了它的行号)它就是恰当的了。通过使用**savedName**和**saveLineNO**静态变量(第4067行)可以做到这一点：

```
assign_stmt : ID { savedName = copyString ( tokenString ) ;
                   savedLineNo = lineno ; }
              ASSIGN exp
                 { $$ = newStmtNode ( AssignK );
                   $$ ->child [ 0 ] = $4 ;
                   $$ ->attr.name = savedName ;
                   $$ ->lineno = saveLineNo ;
                 }
            ;
```

由于作为被匹配的新记号， **tokenString**和**lineno**的值都被扫描程序改变了，所以标识符串和行号必须作为一个在 **ASSIGN**记号识别之前的一个嵌入动作储存起来。但是只有在识别出**exp**之后才能完全构造出赋值的新节点，因此就需要 **savedName**和**saveLineNo**。(实用程序过程**copyString**的使用确保了这些串没有共享存储。读者还需注意将 **exp**的值认为是**$4**。这是因为 Yacc认为嵌入动作在文法规则的右边是一个额外的位置——参见上一节的讨论)。

在**stmt_seq**(第4031行到第4039行)的情况中，它的问题是：属指针 (而不是孩子指针)将语句在TINY语法树中排在一起。因为人们将语法序列的规则写成左递归的，这也就要求为了在末尾附上当前的语句，代码应找出早先为左子集构造的属列表。这样做的效率并不高而且我们还可以通过将规则重写为右递归来避免它，但是这个解决方法也有它自身的问题：只要处理语句序列，其中的分析栈就会变得很大。

最后， Yacc说明的辅助过程部分 (第4144行到第4162行)包括了3个过程——**yyerror**、**yylex**和**parse**——的定义。**parse**过程是由主程序调用，它将调用 Yacc定义的分析过程**yyparse**并且返回保存的语法树。之所以需要 **yylex**过程是因为Yacc假设这是扫描程序过程的名称，而它又在外部被定义为**getToken**。写出了这个定义就使得Yacc生成的分析程序可在对别的代码文件只作出最小的改变的情况下就可与 TINY编译程序一起工作。有人可能希望对扫描程序作出恰当的改变并省掉这个定义，特别是在使用扫描程序的 Lex版本时。在出现错误时，**yyerror**过程由Yacc调用：它将一定的有用信息(如行号)打印到列表文件上。它使用的是

Yacc的内置变量**yychar**，**yychar**包含了引起错误的记号的记号号码。

我们还需要描述对TINY分析程序中的其他文件的改变，由于使用了Yacc来产生分析程序，所以这些改变也是必要的。正如前面已指出的，我们的目标是使这些改变成为最小的，而且将所有的改变都限制为**globals.h**文件的。修改过的文件列在了附录B的第4200行到第4320行中。其基本问题是由Yacc生成的包含了记号定义的头文件必须被包括在大多数的其他代码文件中，但它又不能被直接包括在Yacc生成的分析程序中，因为这样做会重复内置的定义。对上面问题的解决办法是用一个标志**YYPARSER**(第4007行)的定义作为Yacc说明部分的开头，而**YYPARSER**位于Yacc分析程序中且指出C编译程序何时处于分析程序之中。我们使用那个标志(第4226行到第4236行)有选择地将Yacc生成的头文件**y.tab.h**包括在**globals.h**中。

第2个问题出现在**ENDFILE**记号中，此时扫描程序要指出输入文件的结尾。Yacc假设这个记号总是存在着值0，因此也就提供了这个记号的直接定义(第4234行)并且将它包括在由**YYPARSER**控制的有选择性的编译部分之中，这是因为Yacc内部并不需要它。

因为所有的Yacc记号都有整型值，所以**globals.h**文件最后的改变是将**TokenType**重定义为int的一个同义字(第4252行)。这样就避免了无必要地替代前面所列的其他文件中的类型**TokenType**。

5.7 自底向上分析程序中的错误校正

5.7.1 自底向上分析中的错误检测

当在分析表中检测到一个空(或错误)项时，自底向上的分析程序将检测错误。尽可能快地检测到错误显然是有意义的，这样的错误信息可以更有意义且是确定的。因此，分析表应有尽可能多的空项。

不幸的是，这个目标与一个同等重要的目标相冲突，这就是缩小分析表的大小。我们早已看到(在表5-11中)Yacc尽可能多地用缺省归约来填充表项，所以在声明错误之前，大量的归约将占据分析栈。这样就会使错误的准确来源不清晰，而且会导致没有意义的错误信息。

自底向上的分析的另一个特征是所使用的特定算法的能力会影响到分析程序是否可早些检测出错误的能力。例如一个LR(1)分析程序能够比LALR(1)分析程序或SLR(1)分析程序更早地检测出错误来；而在这方面，LALR(1)分析程序和SLR(1)分析程序又比LR(0)分析程序能力强一些。例如，对比LR(0)分析表(表5-4)和LR(1)分析表(表5-10)中的相同文法。假设存在着错误的输入串**(a** $，表5-10中的LR(1)分析表就会将**(**和**a**移进到栈中的状态6。由于在状态6中没有项是位于$之下，所以就报告了一个错误。相反地，LR(0)算法(以及SLR(0)算法)在发现缺少右括号之前先用*A*→**a**归约。类似地，假设存在着错误串**a)** $，则一般的LR(1)分析程序会移进**a**，接着再声明来自右括号上的状态3的错误；而LR(0)分析程序在声明错误之前先用*A*→**a**归约。当然，任何的自底向上的分析程序总是可能会在若干个“错误的”归约之后最终报告出错误来。这些分析程序都不会移进错误的记号。

5.7.2 应急方式错误校正

在自底向上的分析中，通过明智地从分析栈或输入或以上这两者中删除符号有可能会适度地在自底向上的分析程序中得到较好的错误校正。与LL(1)分析程序相似，它有可能完成3种动作：

1) 从栈中弹出一个状态。

2) 在看到可重新开始分析的记号之后，就从输入中成功地弹出记号。

3) 将一个新的状态压入到栈中。

当发生错误时，用来选择以上哪个动作的特别有效的方法如下：

1) 在发现一个带有非空的Goto项的状态之后从分析栈中弹出状态。

2) 若在当前的输入记号上有一个来自Goto状态的合法动作，则将这个状态压入到栈中，并重新开始分析。如果存着在若干个这样的状态，则选择移进而不是归约。在归约动作中，则选择其结合的非终结符为最不一般的那一个。

3) 若在当前输入记号上没有来自Goto状态的合法动作，推进输入直到有一个合法的动作或到达了输入的末尾。

这些规则能够强迫一个构造的识别，当错误发生时这个构造位于正被识别的处理中，并由此立即重新开始分析。使用这些或类似规则的错误校正可被称作应急方式 (panic mode)错误校正，这是因为它与在4.5节中描述的自顶向下的应急方式类似。

不幸的是，由于步骤2将新的状态压入栈中，所以这些规则将会导致一个无穷循环。此时可有若干个可能的解决办法。其一是在步骤 2上坚持来自一个Goto状态的移进动作；但是这可能会有太大的限制。另一个办法是：若下一个合法的移动是归约，则设置一个引起分析程序跟踪在下面归约中的状态序列的标志；若相同的状态重现时，则直到在错误发生时将初始的状态删去之后再弹出栈状态，并且再次从步骤 1开始。若在任何时候都发生了移进，则分析程序重新设置标志并开始正常的分析。

例5.19 在一个简单的算术表达式文法中(它的Yacc分析表在表5-11中给出)，现在考虑错误的输入(**2+*)**。在看到*****之前，分析一直是正常进行。此时，应急方式会导致在分析栈中发生以下的动作：

分 析 栈	输 入	动 作
...	...	...
$ 0 **(** 6 *E* 10**+**7	***** **)** $	错误：
		压入*T*，goto 11
$ 0 **(** 6 *E* 10**+**7 T 11	***** **)** $	移进9
$ 0 **(** 6 *E* 10**+**7 T 11 ***** 9	**)** $	错误：
		压入*F*，goto 13
$ 0 **(** 6 *E* 10**+**7 T 11 ***** 9 F 13	**)** $	用T→*T* ***** *F* 归约
...	...	...

在第1个错误中，分析程序位于状态 7中，它有合法的Goto状态11和4。由于状态11在下一个输入记号*****上存在着一个移进，而这正是Goto倾向的，所以就将记号移进。此时分析程序位于状态9中，在输入中它有一个右括号。而这又是一个错误。在状态 9中，有一个单个的Goto项(到状态11)，且状态11在**)**上也确有一个合法的动作 (虽然是一个归约)。分析接着就正常地继续它的结论。

5.7.3 Yacc中的错误校正

不用应急方式，我们可以使用称作错误产生式 (error production)的方法。错误产生式就是

一个包括了伪记号**error**作为它的右边的唯一符号的产生式。错误产生式标志着一个上下文，直到看到恰当的同步记号时，在其中的错误记号才被删除，而此时又可重新开始分析。错误产生式可有效地允许程序员用手写标记出其Goto项将被用作错误校正的非终结符。

错误产生式是在Yacc中用于错误恢复的主要方法。当发生错误时，Yacc分析程序的行为以及它处理错误产生式的行为如下所示。

1) 当分析程序在分析中检测到错误时(即，它遇到分析表中的一个空项)，它会从分析栈中弹出状态直至到达一个其中的**error**伪记号是合法的先行的状态。其结果是将输入丢弃到错误的左边，并将输入看作是包括了**error**伪记号。如果没有错误伪记号，则**error**永远也不会是移进的合法先行，而且分析栈也将为空，它使分析在第 1个错误处中断(这是由程序清单5-1的输入Yacc生成的分析程序行为)。

2) 一旦分析程序找到了栈上的一个状态，在该状态中的**error**就是一个合法的先行，它以正常的风格继续移进和归约。其结果是好像在输入中看到了 **error**，它的后面是初始先行(也就是导致错误的先行)。如若需要，Yacc宏**yyclearin**可被用来丢弃掉引起错误的记号，并将它随后的记号作为下一个先行(在**error**之后)。

3) 如果分析程序在一个错误发生之后发现了更多的错误，则直到将 3个成功的记号合法地移进到分析栈中之后为止，引起错误的输入记号才会被无声地丢弃掉。此时认为分析程序是位于一个“错误状态”之中。将这个行为设计用来避免由相同错误引起的错误信息级联。但这样就会导致在分析程序退出错误状态之前丢掉大量的输入(同在应急方式中一样)。编译程序的编写者可以利用Yacc宏**yyerrok**将分析程序从错误状态中删除掉以不考虑这个行为，所以在没有新的错误校正的情况下就不会丢掉更多的输入了。

根据程序清单5-1中的Yacc输入，我们再描述这个行为的几个简单示例。

例5.20　考虑程序清单5-1中command规则的以下替换：

```
command : exp        { printf ("%d\n", $1);}
        | error      { yyerror ("incorrect  expression");}
        ;
```

再考虑错误的输入**2++3**。这个串的分析在到达第2个**+**之前一直是正常的。此时的分析是由以下的分析栈和输入给出的(虽然错误产生式的加法实际上将会导致分析表的少许变化，但我们还是用表5-11来描述它)：

PARSING STACK	**INPUT**
$0 *exp* 2 **+** 7	**+3**$

现在分析程序输入错误“状态”(生成一个诸如“语法错误”的错误信息)，从栈开始弹出状态直到发现状态0。此时，*command*的错误产生式提供**error**为合法的先行，而且将它移进到分析栈中，并立即归约到*command*上，它执行了相结合的动作(该动作打印出信息“不正确的表达式”)。最后的状况如下所示：

PARSING STACK	**INPUT**
$0 *command* 1	**+3**$

此时唯一的合法先行是输入的末尾(在这里由$指出，它与由**yylex**返回的0相对应)，而且分析程序在退出之前(但仍在“错误状态”中)将删除输入记号**+3**的剩余部分。因此，除了现在能够提供自己的错误信息之外，错误产生式的加法具有了同程序清单 5-1中版本相同的效果。

一个比这个更好的错误机制允许用户在错误输入之后重新输入行。此时需要一个同步的记

号，而且行标记的末尾是唯一敏感的。因此，扫描程序必须经过修改而返回新行字符 (而不是0)，经过这个修改可以写出(参见练习5.32)：

```
command : exp ' \n ' { printf (' %d\n" , $1); exit (0);}
        | error ' \n '
            {  yyerrok ;
               printf ( "reenter  expression : ");}
          command
        ;
```

这个代码的结果是：当发生错误且当它执行由**yyerrok**表示的动作和**printf**语句时，分析程序将跳过所有的记号而到达一个新行。接着它将试图识别出另一个 *command*。这里需要一个向**yyerrok**的调用来在看到新行之后删除“错误状态”；这是因为如若不然，则当新行的右边发生新的错误时，Yacc在直到发现3个正确的记号序列之后才会无声地删除掉记号。

例5.21 若按以下的表示将一个错误产生式添加到程序清单 5-1的Yacc定义中，那么会出现怎样的结果呢：

```
factor    : NUMBER          {$$ = $1;}
          | '( ' exp ')'    {$$ = $2;}
          | error {$$ = 0;}
          ;
```

首先考虑与前例相同的错误输入**2++3**。(尽管加法错误产生式使得表发生了略微的改变，但我们仍然使用表5-11。)分析程序与上述相同，将会到达以下的输入：

PARSING STACK	INPUT
$0 *exp* 2 **+** 7	**+3**$

现在*factor*的错误产生式将提供**error**为状态7中的一个合法先行，而且将**error**立即移进到栈中，并归约到引起返回值0的*factor*上。现在分析程序已到达了以下的点：

PARSING STACK	INPUT
$0 *exp* 2 **+** 7 *factor* 4	**+3**$

这是一个正常的情况，而且分析程序将继续正常到执行的结束。其结果是将输入解释为**2+0+3**，将0放在两个**+**符号之间是因为此处是**error**伪记号插入的地方，而且通过错误产生式的动作，**error**被看作是与带有0值的一个因子等价。

现在考虑错误的输入**2 3**(即缺少了运算符的两个数字)。分析程序到达了位置

PARSING STACK	INPUT
$0 *exp* 2	**3**$

此时(若*command*的规则并未改变)即使数字并不是*command*的合法后随符号，分析程序也将(错误地)用规则*command* → *exp*归约(并打印出值2)。接着，分析程序到达位置

PARSING STACK	INPUT
$0 *command* 1	**3**$

现在删除了一个错误，且在揭示状态 0的同时从分析栈中弹出状态 1。此点*factor*的错误产生式允许为**error**成为一个来自状态0的合法先行，而且**error**被移进到分析栈中，并导致打印归约的另一个级联以及值 0(由错误产生式返回的值)。现在分析程序又回到了分析的相同位置，而且数字**3**仍在输入中！幸运地是，分析程序早已在“错误状态”中并不再移进 **error**了，但

是却扔掉了数字**3**，它揭示了状态1的正确先行，因此分析程序仍然存在[⊖]。其结果是分析程序按如下所示打印(在第1行中重复用户的输入)：

```
> 2 3
2
syntax error
incorrect expression
0
```

该行为大致给出了Yacc中良好的错误恢复的困难(参见练习中的更多练习)。

5.7.4 TINY中的错误校正

附录B中的Yacc说明文件**tiny.y**包括了两个错误产生式，一个是**stmt**的(第4047行)，另一个是**factor**的(第4139行)，与之相关的动作将返回空的语法树。除了它不试图在错误中建立重要的语法树之外，这些错误产生式提供一个与上一章中的递归下降 TINY分析程序中相似的错误处理级别。这些错误产生式还不能为分析的重新开始提供同步，所以在多重错误中，会跳过多个记号。

练习

5.1 考虑以下的文法：

$$E \rightarrow \mathbf{(}\ L\ \mathbf{)} \mid \mathbf{a}$$
$$L \rightarrow L\ \mathbf{,}\ E \mid E$$

a. 为这个文法构造LR(0)项目的DFA。

b. 构造SLR(1)分析表。

c. 显示分析栈和输入串

```
((a), a, (a,a))
```

的SLR(1)分析程序的动作。

d. 这个文法是不是LR(0)文法？若不是，请描述出LR(0)冲突。如果是，则构造LR(0)分析表，并描述一个分析如何可以与SLR(1)分析不同。

5.2 考虑上面练习中的文法

a. 为这个文法构造LR(1)项目的DFA。

b. 构造一般的LR(1)分析表。

c. 为这个文法构造LALR(1)项目的DFA。

d. 构造LALR(1)分析表。

e. 描述任何可能出现在一般的 LR(1)分析程序的动作和LALR(1)分析程序的动作之间的区别。

5.3 考虑以下的文法：

$$A \rightarrow A\ \mathbf{(}\ A\ \mathbf{)} \mid \varepsilon$$

a. 为这个文法构造LR(0)项目的DFA。

b. 构造SLR(1)分析表。

⊖ Yacc的一些版本在删除任何输入之前先再次弹出分析栈。这样会导致更复杂的行为。参见练习。

c. 显示分析栈和输入串

```
(()())
```

的SLR(1)分析程序的动作。

d. 这个文法是不是 LR(0)文法？如果不是，请描述出 LR(0)冲突。如果是，则构造LR(0)分析表，并描述一个分析如何与SLR(1)分析相区别。

5.4 考虑上一个练习的文法

a. 为这个文法构造LR(1)项目的DFA。

b. 构造一般的LR(1)分析表。

c. 为这个文法构造LALR(1)项目的DFA。

d. 构造LALR(1)分析表。

e. 描述任何可能出现在一般的 LR(1)分析程序的动作和LALR(1)分析程序的动作之间的区别。

5.5 考虑简化了的语句序列的以下文法：

$$\textit{stmt-sequence} \rightarrow \textit{stmt-sequence}\ \texttt{;}\ \textit{stmt} \mid \textit{stmt}$$
$$\textit{stmt} \rightarrow \texttt{s}$$

a. 为这个文法构造LR(0)项目的DFA。

b. 构造SLR(1)分析表。

c. 显示分析栈和输入串

```
s ; s; s
```

的SLR(1)分析程序的动作。

d. 这个文法是不是 LR(0)文法？如果不是，请描述出 LR(0)冲突。如果是，则构造LR(0)分析表，并描述一个分析如何与一个SLR(1)分析相区别。

5.6 考虑上一个练习的文法

a. 为这个文法构造LR(0)项目的DFA。

b. 构造一般的LR(1)分析表。

c. 为这个文法构造LALR(1)项目的DFA。

d. 构造LALR(1)分析表。

e. 描述任何可能出现在一般的 LR(1)分析程序的动作和LALR(1)分析程序的动作之间的区别。

5.7 考虑以下的文法：

$$E \rightarrow \texttt{(}\ L\ \texttt{)} \mid \texttt{a}$$
$$L \rightarrow E\ L \mid E$$

a. 为这个文法构造LR(0)项目的DFA。

b. 构造SLR(1)分析表。

c. 显示分析栈和输入串

```
((a)a(a a))
```

的SLR(1)分析程序的动作。

d. 通过在LR(0)项目的DFA中传送先行来构造LALR(1)项目的DFA。

e. 构造LALR(1)分析表。

5.8 考虑以下的文法

declaration → *type var-list*
type → **int** | **float**
var-list → ***identifier***, *var-list* | *identifier*

a. 将它用一个更适于自底向上分析的格式重写一次。
b. 为重写的文法构造LR(0)项目的DFA。
c. 为重写的文法构造SLR(1)分析表。
d. 利用**c**部分的表为输入串**int x, y, z**显示分析栈和SLR(1)分析程序的动作。
e. 通过在**b**部分中的LR(0)项目的DFA中传送先行来构造LALR(1)项目的DFA。
f. 为重写的文法构造LALR(1)分析表。

5.9 为通过在LR(0)项目的DFA中传送先行而构造LALR(1)项目的DFA写出算法的正式描述(在5.4.3节中已经非正式地描述过这个算法了)。

5.10 本章显示的所有分析栈都包括了状态数和文法符号 (为了清楚起见)，但是分析栈仅需要储存状态数即可——无需将记号和非终结符存放在栈中。若只将状态数保存在栈中，请描述出SLR(1)分析算法。

5.11 **a.** 说明以下的文法不是LR(1)文法：

$$A \rightarrow a\ A\ a \mid \varepsilon$$

b. 这个文法有二义性吗？为什么？

5.12 说明以下的文法是LR(1)文法但不是LALR(1)文法：

$$S \rightarrow a\ A\ d \mid b\ B\ d \mid a\ B\ e \mid b\ A\ e$$
$$A \rightarrow c$$
$$B \rightarrow c$$

5.13 说明不是LALR(1)文法的LR(1)文法只能有归约-归约冲突。

5.14 说明当且仅当一个右句子格式的前缀不会扩展到句柄之外时，它才能是一个变量前缀。

5.15 有没有一个SLR(1)文法不是LALR(1)文法的？为什么？

5.16 说明如果不能最终地将下一个输入记号移进，则一般的LR(1)分析程序在声明一个错误之前不会归约。

5.17 SLR(1)分析程序会不会在声明错误之前实现的归约比 LALR(1)分析程序实现的少？请解释原因。

5.18 以下的有二义性的文法生成了与练习 **5.3**中的文法相同的串 (即：嵌套的括号的所有串)：

$$A \rightarrow AA \mid (\ A\ ) \mid \varepsilon$$

由Yacc生成的分析程序将使用这个文法识别所有的合法串吗？为什么？

5.19 假设有在其中有两个可能的归约 (在不同的先行上)的状态，Yacc将选择其中的一个归约作为它的缺省动作。请描述出Yacc在作出这个选择时所使用的规则(提示：使用练习**5.16**中的文法作为一个测试情况)。

5.20 假设从程序清单5-5的Yacc说明中删除了算符的结合性和优先权的指定 (因此就剩下了一个有二义性的文法)。请描述出Yacc缺省的消除二义性的规则产生了怎样的结合性和优先权。

5.21 同例5.21中脚注所描述的一样，有一些 Yacc分析程序在丢弃任何位于“错误状态”

之中的输入之前会再次弹出分析栈。假设错误输入是 **2 3**，则请描述出例5.21中的Yacc说明对这样的分析程序的行为。

5.22 **a.** 利用分析表5-11和串**(*2**描绘出如5.7.1节所描述的应急方式错误校正机制。

b. 对**a**部分中的行为建议一个改善办法。

5.23 假设程序清单5-1的Yacc计算器说明中的**command**规则是由一个**list**开始的非终结符替换的：

```
list   :       list ' \n' { exit (0);}
       |       list  exp ' \n' {printf ("%d\n", $2);}
       |       list  error ' \n' { yyerrok;}
       ;
```

且这个图中的**yylex**过程将行

```
if (c == ' \n ') return 0;
```

删除掉了。

a. 解释简单计算器的这个版本的行为与程序清单5-1中版本的行为之间的差别。

b. 解释这个规则最后一行的**yyerrok**的原因。给出一个显示若它不在这里的情况的示例。

5.24 **a.** 假设程序清单5-1中的Yacc计算器说明中的**command**的规则被以下的规则

```
command : exp error { printf (" %d\n:, $1);}
```

替代了，则若输入为**2 3**，请准确地解释出Yacc分析程序的行为。

b. 假设程序清单5-1中的Yacc计算器的说明的**command**的规则被以下的规则

```
command : error exp {printf (" %d\n", $2);}
```

替代了，则若输入为**2 3**，请准确地解释出Yacc分析程序的行为。

5.25 假设例5.21中的Yacc错误产生式被以下的规则

```
factor  : NUMBER          {$$ = $1;}
        | ' (' exp ')'    {$$ = $2;}
        | error {yyerrok; $$ = 0;}
        ;
```

替换了。

a. 解释在错误的输入**2++3**中，Yacc分析程序中的行为。

b. 解释在错误的输入**2 3**中，Yacc分析程序中的行为。

5.26 利用4.5.3节中的测试程序对比由Yacc生成的TINY分析程序的错误校正和第4章中的递归下降分析程序。解释二者行为中的不同之处。

编程练习

5.27 重写程序清单5-1中的Yacc说明以使用以下的文法规则(而不是**scanf**)计算出一个数的值(以及，因此舍弃掉**NUMBER**记号)：

number → *number digit* | *digit*

digit→**0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**

5.28 将以下添加到程序清单5-1中的Yacc整型计算说明中(确保它们具有正确的结合性和优先权)：

a. 带有符号**/**的除法。

b. 带有符号%的整型模。

c. 带有符号^的整型求幂(警告：这个算符的优先权比乘法的高，且为右结合)。

d. 带有符号-的一目减。

5.29 将程序清单5-1中的Yacc计算器说明重新改写以使计算器会接受浮点数(并执行浮点计算)。

5.30 重写程序清单5-1中的Yacc计算说明以使其可区别浮点值与整型值，而不是仅仅将任何东西都作为整型数或浮点数来计算。(提示：“值”现在是一个带有指出它是整型还是浮点的标志)。

5.31 **a.** 将程序清单5-1中的Yacc计算器说明重写以使它根据3.3.2节中的说明会返回一个语法树。

b. 写出一个函数，由**a**部分中的代码生成的语法树作为参数并返回由遍历语法树计算出的值。

5.32 为例5.20中的计算器程序建议的简单错误校正技术有一个缺点：它在许多错误之后可导致栈满溢。请改写它以解决这个问题。

5.33 重写程序清单5-1中的Yacc计算器说明以添加以下的有用的错误信息：

由串**(2 +3**生成的“丢失右括号”

由串**2+3)**生成的“丢失左括号”

由串**2 3**生成的“丢失算符”

由串**(2+)**生成的“丢失操作数”

5.34 以下的文法表示在一个类似于LISP的前缀表示法中的简单的算术表达式：

lexp → ***number*** | (*op lexp-seq*)
op → **+** | **-** | *
lexp-seq → *lexp-seq lexp* | *lexp*

例如，表达式(*** (- 2) 3 4**)具有值-24。为一个在这个语法中计算并打印表达式的值的程序写出一个Yacc说明(提示：这要求重写文法以及使用将算符分析为一个lexp-seq的机制)。

5.35 以下的文法代表了第2章曾讨论过的正则表达式：

rexp → *rexp* " | "*rexp*
| *rexp rexp*
| *rexp*" * "
| " **(**"*rexp*"**)** "
| ***letter***

a. 写出一个表达该运算的正确结合性和优先权的大致的Yacc说明(即：没有动作)。

b. 将**a**部分中的说明扩展到包括产生一个“正则表达式编译程序”的所有动作和辅助过程。也就是说，一个在编译时将正则表达式作为C程序的输入和输出的程序为第一次遇到的匹配正则表达式的子集搜索一个输入串(提示：可将表或两维数组代表状态和相关的NFA的转换。接着可利用一个列来储存状态来模拟NFA。只有表才需被Yacc动作生成：剩余的代码将总是相同的，参见第2章)。

5.36 将TINY的Yacc说明(附录B的第4000行到第4162行)用以下之一的方法在更简洁的格式中重写一次：

a. 通过为表达式使用有二义性的文法（和 Yacc对于优先权和结合性带有消除二义性的规则）

b. 通过将算符的识别成一个单一的规则，如在

$$exp \rightarrow exp\ op\ term \mid \ldots$$
$$op \rightarrow \mathbf{+} \mid \mathbf{-} \mid \ldots$$

以及通过使用Y acc的`%union`声明(允许op返回算符，但exp和其他的非终结符却返回指向树节点的指针)。确保你的分析程序产生了与前面相同的语法树。

5.37 将比较算符`<=` (小于或等于)、`>` (大于)、`>=` (大于或等到于)，和`<>` (不等到于)添加到TINY分析程序中的Yacc说明(这将要求添加这些记号并改变扫描程序，却不要求改变语法树)。

5.38 将布尔算符`and`、`or`和`not`添加到TINY分析程序的Yacc说明中。假设它们具有练习**3.5**中描述的特性以及比所有的算术算符都低的优先权，确保任何表达式都可为布尔或整型。

5.39 重写TINY分析程序的Yacc说明以使其可改进它的错误校正。

注意与参考

一般的LR是由Knuth [1965]发明的，但人们直到SLR和LALR技术被DeRemer [1969, 1971]开发出来之前一直都是认为它是不实际的。我们已重复了 LR(1)分析程序对于实际应用非常复杂的惯例。实际上，利用比 LALR(1)分析的技术更精致的状态结合技术可建立实际的 LR(1)分析程序[Pager, 1977]。然而却很少用到附加的能力。在 Aho 和 Ullman [1972]中可找到有关 LR分析技术理论的完整研究。

Yacc是在20世纪70年代由Steve Johnson在A T&T的贝尔实验室开发并包括在大多数的Unix实践中[Johnson, 1975]。它被用来开发便捷的C编译程序[Johnson,1978]以及其他许多编译程序。Bison是由Richard Stallman及其他人一同开发的；Gnu Bison是Free Software Foundation的Gnu软件分配的一部分且可在许多 Internet站点上得到。 Yacc用法的一个示例是在Kernighan 和Pike [1984]中开发的有用但简洁的计算器程序。在Schreiner 和 Friedman [1985]中可找到有关Yacc用法的完整研究。

LR错误校正技术是Graham、Haley 和 Joy [1979]、Penello 和 DeRemer [1978]和Burke 和 Fisher [1987]中的研究内容。在Fischer 和 LeBlanc [1991]中描述了一个LR错误修正技术。5.7.2节中描述的应急方式技术由Fischer 和 LeBlanc归结于James [1972]中。