高等学校计算机科学与技术专业系列教材

编译原理教程

(第五版)

主　编　李玉军　胡元义

副主编　邓亚玲　谈姝辰　赵明华

西安电子科技大学出版社

内 容 简 介

本书系统地介绍了编译程序的设计原理及实现技术。在内容的组织上，本书强调知识的实用性，将编译的基本理论与具体的实现技术有机地结合起来，既注重了理论的完整性，化繁为简，又将理论融于具体的实例中，化难为易，以达到准确、清楚地阐述相关概念和原理的目的。在具体内容讲述中，思路清晰、条理分明，给出的示例丰富并具有实用性与连贯性，使读者对编译的各个阶段有一个全面、直观的认识。本书采用的算法全部由 C 语言描述，各章均附有习题。

本书可作为本科计算机专业的教材，也可作为计算机软件工程人员的参考资料。

图书在版编目(CIP)数据

编译原理教程 / 李玉军，胡元义主编. —5 版. —西安：西安电子科技大学出版社，2021.7
ISBN 978–7–5606–6076–9

Ⅰ. ①编… Ⅱ. ①李… ②胡… Ⅲ. ①编译程序—程序设计—高等学校—教材
Ⅳ. TP314

中国版本图书馆 CIP 数据核字(2021)第 103618 号

策划编辑　马乐惠
责任编辑　陈　婷
出版发行　西安电子科技大学出版社(西安市太白南路 2 号)
电　　话　(029)88242885　88201467　　邮　　编　710071
网　　址　www.xduph.com　　电子邮箱　xdupfxb001@163.com
经　　销　新华书店
印刷单位　咸阳华盛印务有限责任公司
版　　次　2021 年 7 月第 5 版　2021 年 7 月第 20 次印刷
开　　本　787 毫米×1092 毫米　1/16　印　张　17
字　　数　400 千字
印　　数　71 501～74500 册
定　　价　38.00 元
ISBN 978-7-5606-6076-9 / TP
XDUP 6378005-20

前　言

计算机语言之所以能由单一的机器语言发展到现今的数千种高级语言，就是因为有了编译技术。编译技术是计算机科学中发展得最成熟的一个分支，它集中体现了计算机发展的成果与精华。

“编译原理”是计算机专业的一门核心课程，在计算机本科教学中占有十分重要的地位。编译原理课程具有很强的理论性与实践性，读者学习起来普遍感到内容抽象、不易理解。为此，本书采取了由浅入深、循序渐进的方法来介绍编译原理的基本概念和实现方法。在内容的组织上，本书将编译的基本理论与具体的实现技术有机地结合起来，既注重了理论的完整性，化繁为简，又将理论融于具体的实例中，化难为易，以达到准确、清晰地阐述相关概念和原理的目的。除了各章节对理论阐述的条理性之外，书中给出的例子也具有实用性与连贯性，使读者对编译的各个阶段有一个全面、直观的认识，从而透彻地领悟编译原理的精髓。

本书立足于“看得懂、学得会、用得上”，以编译核心知识为纲，以实用实现技术为主，以丰富的示例实践为线，侧重于编译理论的具体实现。书中的算法全部采用 C 语言描述，文法也尽可能采用 C 语言的文法。

本书在前四版的基础上又做了进一步修改，增加了对并行编译技术的介绍，对第四版书中存在的不足之处进行了修订、补充和完善。

本书共分九章。第 1 章简要介绍了编译的基本概念。第 2 章介绍了词法分析的相关内容，主要涉及正规表达式与有限自动机。第 3 章主要介绍语法分析，首先简要地介绍了文法的有关概念，然后介绍了自顶向下语法分析方法——递归下降分析法和 LL(1)分析法，最后介绍了自底向上语法分析方法——算符优先分析法和 LR 分析法。第 4 章介绍了语法制导翻译与中间代码生成的有关内容，给出了如何在语法分析的同时进行语义加工并产生出中间代码的方法。第 5 章介绍了代码优化的有关内容，主要涉及基本块优化和循环优化；此外，还增加了“全局优化概述”一节，以便读者对代码优化有一个全面、完整的了解。第 6 章介绍了程序运行时存储空间的组织。第 7 章讨论目标代码生成的有关内容，讲述了如何由中间代码产生出最终目标代码。第 8 章简要地介绍了符号表的组织与错误处理的方法。第 9 章概要地介绍了并行编译技术。

为了便于读者正确理解有关概念，各章配有一定数量的习题。这些习题大多选自本科生和研究生的考试试题，也包括作者结合多年教学实践经验设计出来的典型范例，力求使读者抓住重点、突破难点，进一步全面、深入地巩固所学知识。

由于水平所限，书中难免存在一些缺点和错误，恳请广大读者批评指正。

编　者

2021 年 3 月

目　录

第1章 绪 论

计算机的诞生是科学发展史上的一个里程碑。经过半个多世纪的发展，计算机已经改变了人类生活、工作的各个方面，成为人类不可缺少的工具。计算机之所以能够如此广泛地被应用，应当归功于高级程序设计语言。计算机语言之所以能由最初单一的机器语言发展到现今数千种高级语言，就是因为有了编译程序。没有高级语言，计算机的推广应用是难以实现的；而没有编译程序，高级语言就无法使用。编译理论与技术也是计算机科学中发展得最迅速、最成熟的一个分支，它集中体现了计算机发展的成果与精华。

1.1 程序设计语言和编译程序

为了处理和解决实际问题，每一种计算机都可以实现其特定的功能，而这些功能是通过计算机执行一系列相应的操作来实现的。计算机所能执行的每一种操作对应为一条指令，计算机能够执行的全部指令集合就是该计算机的指令系统。

计算机硬件的器件特性决定了计算机本身只能直接接受由 0 和 1 编码的二进制指令和数据，这种二进制形式的指令集合称为该计算机的机器语言，它是计算机唯一能够直接识别并接受的语言。

用机器语言编写程序很不方便且容易出错，编写出来的程序也难以调试、阅读和交流。为此，出现了用助记符代替机器语言(二进制编码)的另一种语言，这就是汇编语言。汇编语言是建立在机器语言之上的，因为它是机器语言的符号化形式，所以较机器语言直观；但是计算机并不能直接识别这种符号化语言，用汇编语言编写的程序必须翻译成机器语言之后才能执行，这种“翻译”是通过专门的软件——汇编程序实现的。

尽管汇编语言与机器语言相比在阅读和理解上有了长足的进步，但其依赖具体机器的特性是无法改变的，这给程序设计增加了难度。随着计算机应用需求的不断增长，出现了更加接近人类自然语言的功能更强、抽象级别更高的面向各种应用的高级语言。高级语言已经从具体机器中抽象出来，摆脱了依赖具体机器的问题。用高级语言编制的程序几乎能够在不改动的情况下在不同种类的计算机上运行且不易出错，这是汇编语言难以做到的，但高级语言程序翻译(编译)成最终能够直接执行的机器语言程序其难度却大大增加了。

由于汇编语言和机器语言一样都是面向机器的，故相对于面向用户的高级语言来说，它们都被称为低级语言，而 FORTRAN、PASCAL、C、ADA、Java 这类面向应用的语言则称之为高级语言。因此，编译程序就是指这样一种程序，通过它能够将用高级语言编写的源程序转换成与之在逻辑上等价的低级语言形式的目标程序，见图 1-1。

一个高级语言程序的执行通常分为两个阶段，即编译阶段和运行阶段，如图 1-2 所示。

编译阶段将源程序变换成目标程序；运行阶段则由所生成的目标程序连同运行系统(数据空间分配子程序、标准函数程序等)接收程序的初始数据作为输入，运行后输出计算结果。

图 1-1　编译程序的功能

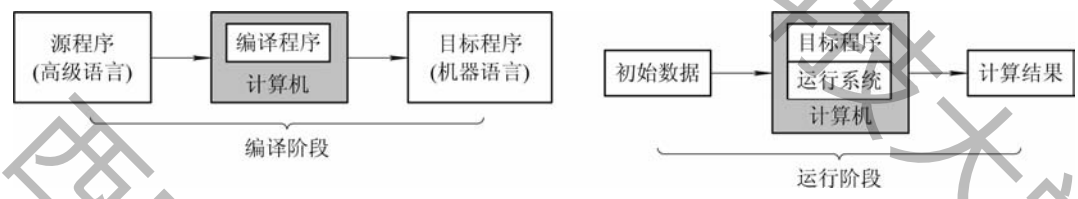

图 1-2　源程序的编译和运行阶段

如果编译生成的目标程序是汇编语言形式的，那么在编译与运行阶段之间还要添加一个汇编阶段，它将编译生成的汇编语言目标程序再经过汇编程序变换成机器语言目标程序，如图 1-3 所示。

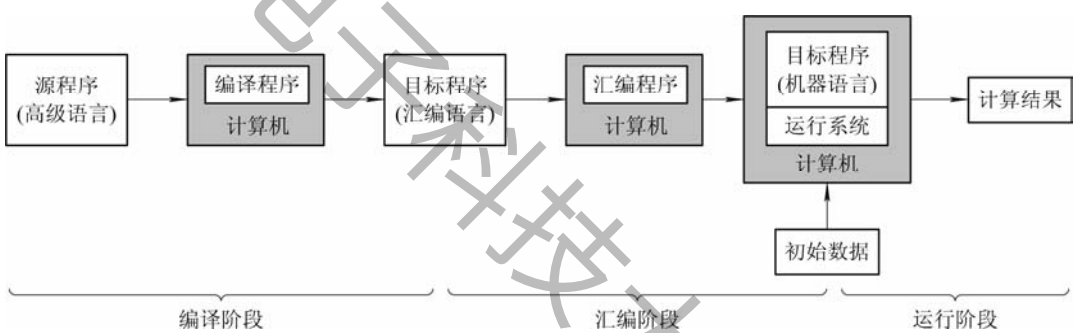

图 1-3　源程序的编译、汇编和运行阶段

用高级语言编写的程序也可通过解释程序来执行。解释程序也是一种翻译程序，它将源程序作为输入，一条语句一条语句地读入并解释执行，如图 1-4 所示。解释程序与编译程序的主要区别是：编译程序将源程序翻译成目标程序后再执行该目标程序；而解释程序则逐条读出源程序中的语句并解释执行，即在解释程序的执行过程中并不产生目标程序。典型的解释型高级语言是 BASIC 语言。

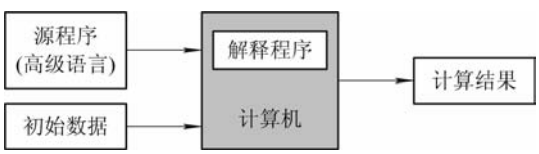

图 1-4　解释程序解释执行过程示意

1.2　编译程序的历史及发展

20 世纪 40 年代，由于冯·诺伊曼在存储程序计算机方面的开创性工作，计算机可以

执行编写的一串代码或程序，这些程序最初都是用机器语言(Machine Language)编写的。机器语言就是计算机能够执行的全部指令集合的二进制形式，例如：

C7 06 0000 0002

表示在 IBM PC 上使用的 Intel 8x86 处理器将数字 2 移至地址 0000(十六进制)的指令。用机器语言编写程序很不方便且容易出错，因此这种代码形式很快就被汇编语言(Assembly Language)取代。在汇编语言中，指令和存储地址都以符号形式给出。例如，汇编语言指令

MOV X, 2

就与前面的机器指令等价(假设符号存储地址 X 是 0000)。汇编程序(Assembler)再将汇编语言的符号代码和存储地址翻译成与机器语言相对应的二进制代码。汇编语言大大提高了编程的速度和准确性，至今人们仍在使用它，在有存储容量小和速度快的要求时尤其如此。但是，汇编语言依赖于具体机器的特性是无法改变的，这给编程和程序调试增加了难度。很明显，编程技术发展的下一个重要步骤就是用更简洁的数学定义或自然语言来描述和编写程序，它应与任何机器无关，而且也可通过一个翻译程序将其翻译为可在计算机上直接执行的二进制代码。例如，前面的汇编语言代码“MOV X, 2”可以写成一个简洁的与机器无关的形式“X=2”。

1954～1957 年，IBM John Backus 带领的一个研究小组对 FORTRAN 语言及其编译器进行了开发。但是，由于对编译程序的理论及技术研究刚刚开始，这个语言的开发付出了巨大的辛劳。与此同时，波兰语言学家 Noam Chomsky 开始了他的自然语言结构研究。Noam Chomsky 根据文法(Grammar，产生语言的规则)的难易程度及识别它们所需的算法对语言进行了分类，定义了 0 型、1 型、2 型和 3 型这四类文法及其相应的形式语言，并分别与相应的识别系统相联系。2 型文法(上下文无关文法，Context-free Grammar)被证明是程序设计语言中最有用的文法，它代表着目前程序设计语言结构的标准。Noam Chomsky 的研究结果最终使得编译器结构异常简单，甚至还具有自动化功能。有限自动机(Finite Automata)和正规表达式(Regular Expression)与上下文无关文法紧密相关，它们与 Noam Chomsky 的 3 型文法相对应，并引出了表示程序设计语言的单词符号形式。接着又产生了生成有效目标代码的方法——这就是最初的编译器，它们被沿用至今。随着对语法分析研究的深入，重点转移到对编译程序的自动生成的研究上。开发的这种程序最初被称为编译程序的编译器，但因为它们仅仅能够自动完成编译器的部分工作，所以更确切地应称为分析程序生成器(Parser Generator)；这些程序中最著名的是 Steve Johnson 在 1975 年为 UNIX 系统编写的语法分析器自动生成工具 YACC(Yet Another Compiler-Compiler)。类似地，有限自动机的研究也产生出另一种称为词法分析器自动生成工具(Scanner Generator)Lex。

20 世纪 70 年代后期和 80 年代初，大量的研究都关注于编译器其他部分的自动生成，其中包括代码生成。这些努力并未取得多少成功，这是由于这部分工作过于复杂，对其本质不甚了解，在此不再赘述。

现今编译器的发展包括了更为复杂的算法应用程序，它用于简化或推断程序中的信息，这又与具有此类功能的更为复杂的程序设计语言发展结合在一起。其中典型的有用于函数语言编译 Hindley-Milner 类型检查的统一算法。目前，编译器已经越来越成为基于窗口的交互开发环境(Interactive Development Environment，IDE)的一部分，这个开发环境包括了编辑器、链接程序、调试程序以及项目管理程序。尽管近年来对 IDE 进行了大量的研究，

但是基本的编译器设计在近 40 年中都没有多大的改变。

现代编译技术已转向并行编译的研究，由于本书重点是介绍经典的编译理论和技术，因此，对并行编译的发展仅做综述介绍。

1.3　编译过程和编译程序结构

编译程序的工作过程是指从输入源程序开始到输出目标程序为止的整个过程。此过程是非常复杂的。一般来说，整个编译过程可以划分成五个阶段：词法分析阶段、语法分析阶段、语义分析和中间代码生成阶段、优化阶段和目标代码生成阶段。

1．词法分析

词法分析的任务是输入源程序，对构成源程序的字符串进行扫描和分解，识别出一个个单词符号，如基本字(if、for、begin 等)、标识符、常数、运算符和界符(如“(”、“)”、“=”、“;”)等，将所识别出的单词用统一长度的标准形式(也称内部码)来表示，以便于后继语法工作的进行。因此，词法分析工作是将源程序中的字符串变换成单词符号流的过程，词法分析所遵循的是语言的构词规则。

2．语法分析

语法分析的任务是在词法分析的基础上，根据语言的语法规则(文法规则)把单词符号流分解成各类语法单位(语法范畴)，如“短语”、“子句”、“句子(语句)”、“程序段”和“程序”。通过语法分析可以确定整个输入串是否构成一个语法上正确的“程序”。语法分析所遵循的是语言的语法规则，语法规则通常用上下文无关文法描述。

3．语义分析和中间代码生成

语义分析和中间代码生成的任务是对各类不同语法范畴按语言的语义进行初步翻译，包含两个方面的工作：一是对每种语法范畴进行静态语义检查，如变量是否定义、类型是否正确等；二是在语义检查正确的情况下进行中间代码的翻译。注意，中间代码是介于高级语言的语句和低级语言的指令之间的一种独立于具体硬件的记号系统，它既有一定程度的抽象，又与低级语言的指令十分接近，因此转换为目标代码比较容易。把语法范畴翻译成中间代码所遵循的是语言的语义规则，常见的中间代码有四元式、三元式、间接三元式和逆波兰记号等。

4．优化

优化的任务主要是对前阶段产生的中间代码进行等价变换或改造(另一种优化是针对目标机即对目标代码进行优化)，以期获得更为高效(节省时间和空间)的目标代码。常用的优化措施有删除冗余运算、删除无用赋值、合并已知量、循环优化等。例如，其值并不随循环而发生变化的运算可提到进入循环前计算一次，而不必在循环中每次循环都进行计算。优化所遵循的原则是程序的等价变换规则。

5．目标代码生成

目标代码生成的任务是把中间代码(或经优化处理之后)变换成特定机器上的机器语言程序或汇编语言程序，实现最终的翻译工作。最后阶段的工作因为目标语言的关系而十分

依赖硬件系统，即如何充分利用机器现有的寄存器，合理地选择指令，生成尽可能短且有效的目标代码，这些都与目标机器的硬件结构有关。

上述编译过程的五个阶段是编译程序工作时的动态特征，编译程序的结构可以按照这五个阶段的任务分模块进行设计。编译程序的结构示意如图 1-5 所示。

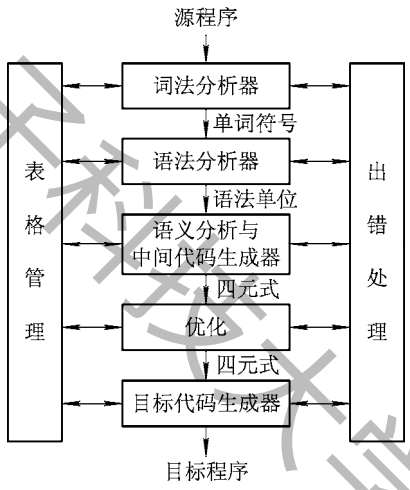

图 1-5　编译程序的结构示意

编译过程中源程序的各种信息被保留在不同的表格里，编译各阶段的工作都涉及构造、查找或更新有关的表格，编译过程的绝大部分时间都用在造表、查表和更新表格的事务上，因此，编译程序中还应包括一个表格管理程序。

出错处理与编译的各个阶段都有联系，与前三个阶段的联系尤为密切。出错处理程序应在发现错误后，将错误的有关信息如错误类型、出错地点等向用户报告。此外，为了尽可能多地发现错误，应在发现错误后还能继续编译下去。

一个编译过程可分为一遍、两遍或多遍完成，每一遍完成所规定的任务。例如，第一遍只完成词法分析的任务，第二遍完成语法分析和语义加工工作并生成中间代码，第三遍再实现代码优化和目标代码生成。当然，也可一遍即完成整个编译工作。至于一个编译程序究竟应分为几遍，如何划分，这和源程序语言的结构与目标机器的特征有关。分多遍完成编译过程可以使整个编译程序的逻辑结构更清晰，但遍数多势必增加读写中间文件的次数，从而消耗过多的时间。

1.4　编译程序的开发

由于计算机语言功能的完善、硬件结构的发展、环境界面的友好等都对编译程序提出了更多、更高的要求，因而构造一个编译系统并非易事。虽然编译理论和编译技术的不断发展已使编译程序的生产周期不断缩短，但是要研制完成一个编译程序仍需要相当长的时间，工作也相当艰巨。因此，如何高效、高质量地生成一个编译程序一直是计算机系统设计人员追求的目标。

编译程序的任务是把源程序翻译成某台计算机上的目标程序，因此，开发人员首先要熟悉这种源程序语言，对源程序语言的语法和语义要有准确无误的理解。此外，开发人员还需确定编译程序的开发方案及方法，这是编译开发过程中最关键的一步，其作用是使编译程序具有易读性和易改性，以便将来对编译程序的功能进行更新和扩充。选择合适的语言编写编译程序也是非常重要的，语言选择不当会使开发出来的编译程序的可靠性变差，难以维护且质量也无法保证。目前大部分编译程序都是用 C 语言之类的高级语言编写的，这不仅减少了开发的工作量，也缩短了开发周期。最后，开发人员对目标机要有深入的研究，这样才能充分利用目标机的硬件资源和特点，产生质量较高的目标程序。

编译程序的开发常常采用自编译、交叉编译、自展和移植等技术实现。

1．自编译

用某种高级语言编写自己的编译程序称为自编译。例如，假定 A 机器上已有一个 PASCAL 语言可以运行，则可以用 PASCAL 语言编写出一个功能更强的 PASCAL 语言编译程序，然后借助于原有的 PASCAL 编译程序对新编写的 PASCAL 编译程序进行编译，从而在编译后即得到一个能在 A 机器上运行的功能更强的 PASCAL 编译程序。

2．交叉编译

交叉编译是指用 A 机器上的编译程序来产生可在 B 机器上运行的目标代码。例如，若 A 机器上已有 C 语言可以运行，则可用 A 机器中的 C 语言编写一个编译程序，它的源程序是 C 语言程序，而产生的目标程序则是基于 B 机器的，即能够在 B 机器上执行的低级语言程序。

以上两种方法都假定已经有了一个系统程序设计语言可以使用，若没有可使用的系统程序设计语言，则可采用自展或移植的方法来开发编译程序。

3．自展

自展的方法是：首先确定一个非常简单的核心语言 L_0，然后用机器语言或汇编语言编写出它的编译程序 T_0；再把语言 L_0 扩充到 L_1，此时有 $L_0 \subset L_1$，并用 L_0 编写 L_1 的编译程序 T_1(即自编译)；然后再把语言 L_1 扩充为 L_2，此时有 $L_1 \subset L_2$，并用 L_1 编写 L_2 的编译程序 T_2……这样不断扩展下去，直到完成所要求的编译程序为止。

4．移植

移植是指 A 机器上的某种高级语言的编译程序稍加改动后能够在 B 机器上运行。一个程序若能较容易地从 A 机器上搬到 B 机器上运行，则称该程序是可移植的。移植具有一定的局限性。

用系统程序设计语言来编写编译程序虽然缩短了开发周期并提高了编译程序的质量，但实现的自动化程度不高。实现编译程序的最高境界是能够有一个自动生成编译程序的软件工具，只要把源程序的定义以及机器语言的描述输入到该软件中，就能自动生成这个语言的编译程序，如图 1-6 所示。

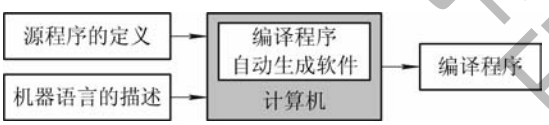

图 1-6　编译程序自动生成示意

计算机科学家和软件工作者为了实现编译程序的自动生成进行了大量研究工作，随着形式语言学研究的发展和编译程序自动生成工作的进展，已经出现了一些编译程序中某一部分的自动生成系统，如 UNIX 操作系统下的软件工具 Lex 和 YACC 等。

1.5　构造编译程序所应具备的知识内容

要在某一台机器上为某种语言构造一个编译程序，必须掌握下述三方面的内容。

(1) 对被编译的源语言(如 C、PASCAL 等)，要深刻理解其结构(语法)和含义。例如，

下面的 for 循环语句：

```
for(i=1;i<=10+i;i++)
    x=x+1;
```

就存在对循环终值的理解问题。一种理解是以第一次进入 for 循环的 i 值计算出循环终值，此循环终值在循环中不再改变，也即循环终值为 11；而另一种理解则是循环终值表达式 10+i 中的 i 值随循环在不断地改变，此时 for 语句将出现死循环。如 TURBO PASCAL 就是按第一种语义进行翻译的，而 TURBO C 和 VC++6.0 则是按第二种语义进行翻译的。此外，如果出了循环后还要引用 i 值，那么这个 i 值究竟是循环终值还是循环终值加 1？因此，对语义的不同理解可以得到不同的编译程序。

其次，C 语言中的 for 循环语句还可以写成下面的形式：

```
for ( i=0, j=5; i<3, j>0; i++, j-- )
    printf ("%d,%d\n", i, j );
```

逗号表达式“i<3,j>0”作为循环终值表达式，C 语言规定该表达式的值取逗号表达式中最右一个表达式的值，因此当 i 值为 3 时循环并不终止，只有当 j 值为 0 时循环才结束。不了解 C 语言的逗号表达式的功能就会得到错误的运行结果。

此外，下面 C 语言的函数：

```
#include<stdio.h>
int f(int x,int y)
{
    return x+y;
}
void main()
{
    int i=3;
    printf("%d\n",f(i,++i));
}
```

C 编译程序对函数参数传递的处理是由右向左进行的，因此，先传递的是第二个参数++i，即 i 值先进行自增，由 3 变为 4，也即这时函数 f 的两个实参的值都为 4，最终程序的运行结果是 8，而不是 7。不了解 C 语言的函数传递方式就很容易得到错误的结果。

即使是同一个 C 语言语句，不同版本的编译系统翻译的结果也是不一样的。例如：

```
i=3;
k=(++i)+(++i)+(++i);
```

对 VC++ 6.0 版本的 C 语言来说，执行语句“k=(++i)+(++i)+(++i);”的过程是先执行前两个“++i”，即先对 i 进行两次自增，i 值变为 5，然后相加得到 10；接下来再将这个 10 与第三个“++i”相加，也是先对 i 进行自增，其值由 5 变为 6，最后 10 加 6 得到 16，所以 k 值最终为 16。对 Turbo C 版本的 C 语言来说，执行语句“k=(++i)+(++i)+(++i);”的过程是先执行三次“++i”，即先对 i 进行三次自增，i 值变为 6，然后再将自增后的三个 i 值相加，其结果为 18，所以 k 值最终为 18。也即，不同的 C 编译程序给出了不同的解释。

(2) 必须对目标机器的硬件和指令系统有深刻的了解。例如，两个数相加的指令在

8086/8088 汇编中假定用下面两种指令实现：

ADD　AX，06 或 ADD　BX，06

粗略看来，这两条加法指令除了寄存器不同外没有本质上的差别，其实不然。由于 AX 是累加器，因此，从机器指令的代码长度来说(见附录 1)，第一条指令比第二条指令节省一个字节。此外，从 PC 硬件结构来看，AX 本身就是累加器且相加的结果也在累加器中，这就节省了传送的时间；而第二条指令则先要将 BX 中的值送到累加器中，相加后又要从累加器中取出结果再送回寄存器 BX 中。显然，第二条指令要比第一条指令费时，因此，在可能的情况下，应尽量生成像第一条指令这样的目标代码。

(3) 必须熟练掌握编译方法，编译方法掌握的如何将直接影响到编译程序的成败，一个好的编译方法可能得到事半功倍的效果。

由于编译程序是一个极其复杂的系统，故在讨论中只好把它分解开来，一部分一部分地进行研究。在学习编译程序的过程中，应注意前后联系，切忌用静止的、孤立的观点看待问题；作为一门技术课程，学习时还必须注意理论联系实际，多练习、多实践。

习　题　1

1.1　完成下列选择题：

(1) 下列叙述中正确的是_____。

A．编译程序是将高级语言程序翻译成等价的机器语言程序的程序

B．机器语言因其使用过于困难，所以现在计算机根本不使用机器语言

C．汇编语言是计算机唯一能够直接识别并接受的语言

D．高级语言接近人们的自然语言，但其依赖具体机器的特性是无法改变的

(2) 将编译程序分成若干个“遍”是为了_____。

A．提高编译程序的执行效率

B．使编译程序的结构更加清晰

C．利用有限的机器内存并提高机器的执行效率

D．利用有限的机器内存但降低了机器的执行效率

(3) 构造编译程序应掌握_____。

A．源程序　　B．目标语言

C．编译方法　　D．以上三项

(4) 编译程序绝大多数时间花在_____上。

A．出错处理　　B．词法分析

C．目标代码生成　　D．表格管理

(5) 编译程序是对_____。

A．汇编程序的翻译　　B．高级语言程序的解释执行

C．机器语言的执行　　D．高级语言的翻译

1.2　计算机执行用高级语言编写的程序有哪些途径？它们之间主要区别是什么？

1.3　请画出编译程序的总框图。如果你是一个编译程序的总设计师，设计编译程序时应当考虑哪些问题？

第2章 词法分析

词法分析是编译的第一个阶段，其任务是：从左至右逐个字符地对源程序进行扫描，产生一个个单词符号，把字符串形式的源程序改造成为单词符号串形式的中间程序。执行词法分析的程序称为词法分析程序，也称为词法分析器或扫描器。词法分析器的功能是输入源程序，输出单词符号。

词法分析可以采用如下两种处理结构：

(1) 把词法分析程序作为主程序。将词法分析工作作为独立的一遍来完成，即把词法分析与语法分析明显分开，由词法分析程序将字符串形式的源程序改造成单词符号串形式的中间程序，以这个中间程序作为语法分析程序的输入。在这种处理结构中，词法分析和语法分析是分别实现的，如图 2-1(a)所示。

(2) 把词法分析程序作为语法分析程序调用的子程序。在进行语法分析时，每当语法分析程序需要一个单词时便调用词法分析程序，词法分析程序每一次调用便从字符串源程序中识别出一个单词交给语法分析程序。在这种处理结构中，词法分析和语法分析实际上是交替进行的，如图 2-1(b)所示。

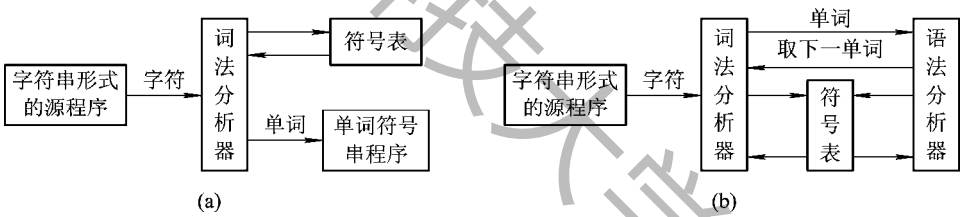

图 2-1 词法分析的两种处理结构

(a) 词法分析程序作为主程序；(b) 词法分析程序作为子程序

由于把词法分析器安排成一个子程序比较自然，因此，词法分析程序通常采用第二种处理结构。

2.1 词法分析器的设计方法

2.1.1 单词符号的分类与输出形式

1. 单词符号分类

词法分析程序简单地说就是读单词程序，该程序扫描用高级语言编写的源程序，将源程序中由单词符号组成的字符串分解出一个个单词来。因此，单词符号是程序语言的基本

语法单位，具有确定的语法意义。程序语言的单词符号通常可分为下面五种。

(1) 保留字(也称基本字)：如 C 语言中的 if、else、while 和 do 等，这些字保留了语言所规定的含义，是编译程序识别各类语法成分的依据。几乎所有程序语言都限制用户使用保留字来作为标识符。

(2) 标识符：用来标记常量、数组、类型、变量、过程或函数名等，通常由用户自己定义。

(3) 常数：包括各种类型的常数，如整型常数 386、实型常数 0.618、布尔型常数 TRUE 等。

(4) 运算符：如“+”、“−”、“*”、“/”、“>”、“<”等。

(5) 界符：在语言中是作为语法上的分界符号使用的，如“，”、“；”、“(”、“)”等。

注意：一个程序语言的保留字、运算符和界符的个数是确定的，而标识符或常数的使用则不限定个数。

2．词法分析程序输出单词的形式

我们知道，词法分析程序的输入是源程序字符串，而输出是与源程序等价的单词符号序列，并且所输出的单词符号通常表示成如下的二元式：

(单词种别，单词自身的值)

(1) 单词种别。单词种别表示单词的种类，它是语法分析所需要的信息。一个语言的单词符号如何划分种类、分为几类、如何编码都属于技术性问题，主要取决于处理上的方便。通常让每种单词对应一个整数码，这样可最大限度地把各个单词区别开来。对于保留字，可将其全体视为一种，也可一字一种，采用一字一种的分类方法处理起来比较方便；标识符一般统归为一种；常数可统归为一种，也可按整型、实型、布尔型等分为几种；运算符和界符可采用一符一种的分法，也可统归为一种。

(2) 单词自身的值。单词自身的值是编译中其它阶段所需要的信息。对于单词符号来说，如果一个种别只含有一个单词符号，那么对于这个单词符号，其种别编码就完全代表了它自身的值。如果一个种别含有多个单词符号，那么对于它的每个单词符号，除了给出种别编码之外还应给出单词符号自身的值，以便把同一种类的不同单词区别开来。注意，标识符自身的值就是标识符自身的字符串，而常数自身的值是常数本身的二进制数值。此外，我们也可用指向某类表格中一个特定项目的指针来区分同类中的不同单词。例如，对于标识符，可以用它在符号表的入口指针作为它自身的值；而常数也可用它在常数表的入口指针作为它自身的值。

2.1.2　状态转换图

在词法分析中，可以用状态转换图来识别单词。状态转换图是有限的有向图，结点代表状态，用圆圈表示；结点之间可由有向边连接，有向边上可标记字符。例如，图 2-2 表示在状态 i 下，若输入字符为 x，则读入 x 并转换到状态 j；若输入字符为 y，则读入 y 并转换到状态 k。

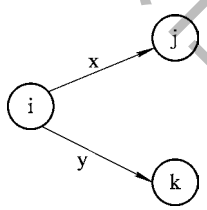

图 2-2　不同输入字符的状态转换

状态(即结点)数是有限的，其中必有一初始状态以及若干终止状态，终止状态(终态)的结点用双圈表示以区别于其它状态。图 2-3 给出了用于识别标识符、无符号整数、无符号数的状态转换图，其初始状态均用 0 状态表示。

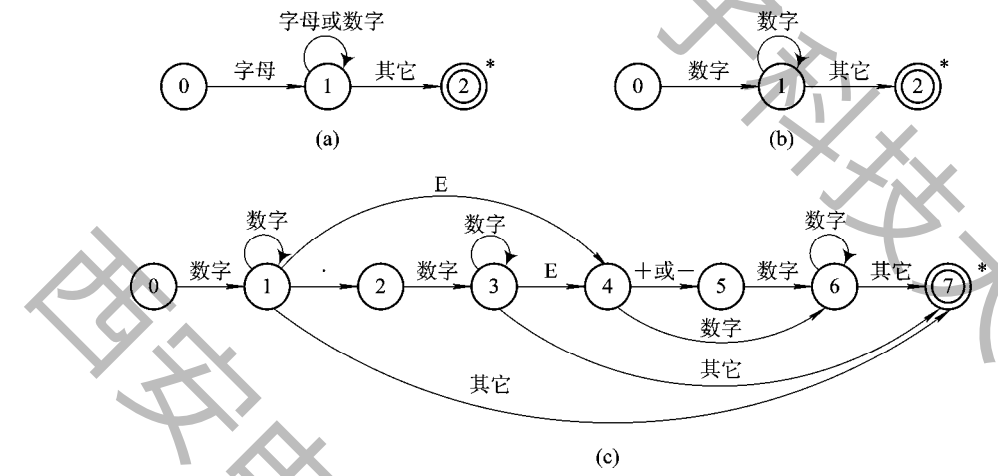

图 2-3　标识符及无符号数的状态转换图

(a) 标识符；(b) 无符号整数；(c) 无符号数

当到达一类单词符号的终止状态时即可给出相应的单词编码。某些终止状态是在读入了一个其它不属于该单词的符号后才得到相应的单词编码的，这表明在识别单词的过程中多读入了一个符号，所以识别出单词后应将最后多读入的这个符号予以回退；我们对此类情况的处理是在终态上以“*”作为标识。

对于不含回路的分支状态来说，可以让它对应一个 switch( ) 语句或一组 if-else 语句。例如，图 2-4(a) 的状态 i 所对应的 switch 语句如下：

```
s=getchar( );
switch(s)
{   case 'a':
    case 'b':
      …
    case 'z':
      … ;                          //实现状态 j 功能的语句
    case '0':
    case '1':
      …
    case '9':
      … ;                          //实现状态 k 功能的语句
}
```

对于含回路的状态来说，可以让它对应一个 while 语句。例如，图 2-4(b) 的状态 i 所对应的 while 语句如下：

```
getchar( );
```

```
while( letter()||digit())
    getchar();
…;                                    //实现状态 j 功能的语句
```

终态一般对应一个 return()语句。return 意味着从词法分析器返回到调用段，一般指返回到语法分析器。

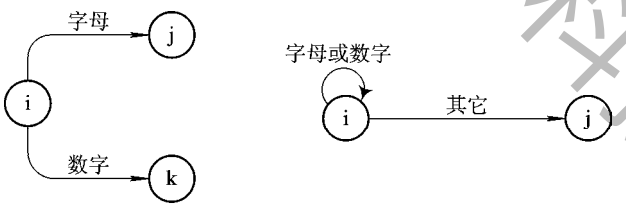

图 2-4　含有分支或回路的状态示意

(a) 含分支的状态 i；(b) 含回路的状态 i

2.2　一个简单的词法分析器示例

2.2.1　C 语言子集的单词符号表示

一个非常重要的事实是：大多数程序语言的单词符号都可以用状态转换图予以识别。作为一个综合例子，我们来构造一个 C 语言子集的简单词法分析器。表 2.1 列出了这个 C 语言子集的所有单词符号以及它们的种别编码和内码值。由于直接使用整数编码不利于记忆，故该例中用一些特殊符号来表示种别编码。

表 2.1　C 语言子集的单词符号及内码值

单词符号	种别编码	助记符	内码值
while	1	while	—
if	2	if	—
else	3	else	—
switch	4	switch	—
case	5	case	
标识符	6	id	id 在符号表中的位置
常数	7	num	num 在常数表中的位置
+	8	+	—
−	9	−	—
*	10	*	—
<=	11	relop	LE
<	11	relop	LT
= =	11	relop	EQ
=	12	=	—
;	13	;	—

2.2.2 C 语言子集对应的状态转换图

在设计的状态转换图中，首先对输入串做预处理，即剔除多余的空白符(在实际的词法分析中，预处理还包括剔除注释和制表换行符等编辑性字符的工作)，使词法分析工作既简单又清晰。其次，将保留字作为一类特殊的标识符来处理，也即对保留字不专设对应的状态转换图，当转换图识别出一个标识符时就去查对表 2.1 的前五项，确定它是否为一个保留字。当然，也可以专设一个保留字表来进行处理。

图 2-5 就是对应表 2.1 这个简单词法分析的状态转换图。

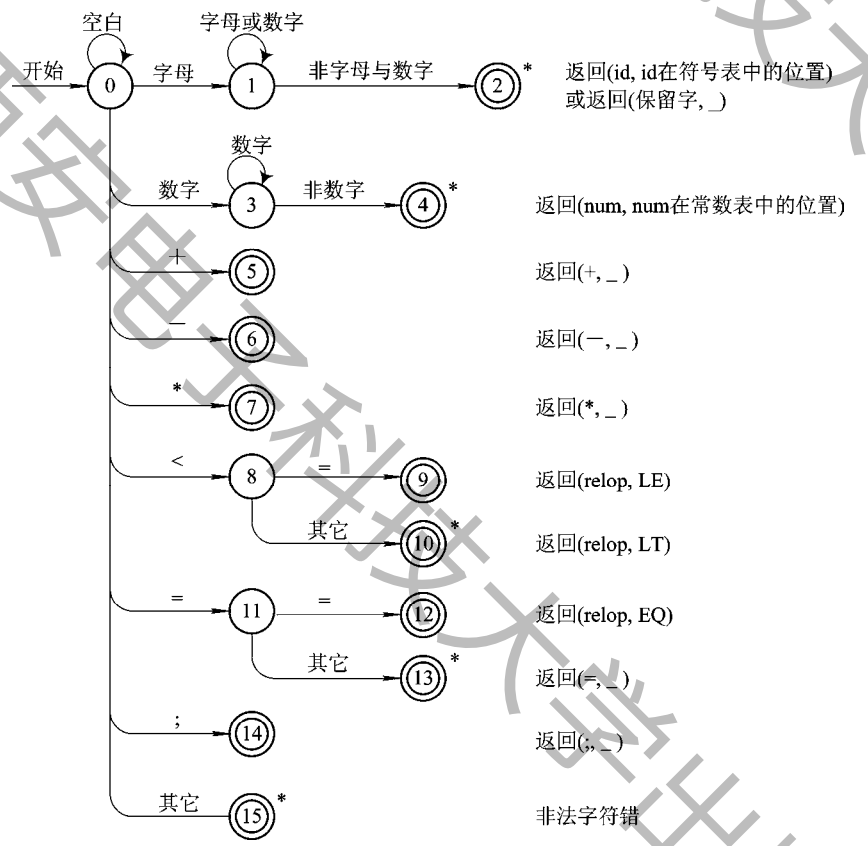

图 2-5 简单词法分析的状态转换图

注意：在状态 2 时，所识别出的标识符应先与表 2.1 的前五项逐一比较，若匹配，则该标识符是一个保留字，否则就是标识符。如果是标识符，应先查符号表，看表中是否有此标识符。若表中无此标识符，则将它登录到符号表中，然后返回其在符号表中的入口指针(地址)作为该标识符的内码值；若表中有此标识符，则给出重名错误信息。在状态 4 时，应将识别的常数转换成二进制常数并将其登录到常数表中，然后返回其在常数表中的入口指针作为该常数的内码值。

2.2.3 状态转换图的实现

状态转换图非常容易用程序实现，最简单的办法是让每个状态对应一小段程序。对于

图 2-5 所示的状态转换图，我们首先引进一组变量和函数如下：

(1) character：字符变量，存放最新读入的源程序字符。

(2) token：字符数组，存放构成单词符号的字符串。

(3) getbe()：若 character 中的字符为空白，则调用 getchar()，直至 character 为非空白字符为止。

(4) concatenation()：将 token 中的字符串与 character 中的字符连接并作为 token 中新的字符串。

(5) letter()和 digit()：判断 character 中的字符是否为字母和数字的布尔函数，是则返回 true(即 1)，否则返回 false(即 0)。

(6) reserve()：按 token 数组中的字符串查表 2.1 中的前五项(即判别其是否为保留字)，若是保留字则返回它的种别编码，否则返回 0 值。

(7) retract()：扫描指针回退一个字符，同时将 character 置为空白。

(8) buildlist()：将标识符登录到符号表中或将常数登录到常数表中。

(9) error()：出现非法字符，显示出错信息。

相对于图 2-5 的词法分析器构造如下：

```
token=' ';                          //对 token 数组初始化
character=getchar();
getbe();                            //滤除空格
switch(character)
{
    case 'a':
    case 'b':
    …
    case 'z':
        while( letter() || digit())
        {
            concatenation();        //将当前读入的字符送入 token 数组
            character =getchar();
        }
        retract();                  //扫描指针回退一个字符
        c=reserve();
        if(c==0)
        {
            buildlist();            //将标识符登录到符号表中
            return(id,指向 id 的符号表入口指针);
        }
        else
             return(C 语言子集中的保留字种别编码,null);
        break;
```

```
case '0':
case '1':
…
case '9':
    while(digit())
    {
        concatenation();
        character=getchar();
    }
    retract();
    buildlist();                          //将常数登录到常数表中
    return(num,num 的常数表入口指针);
    break;
case '+':
    return('+',null);
    break;
case '−':
    return('−',null);
    break;
case '*':
    return('*',null);
    break;
case '<':
    character=getchar();
    if(character=='=')
        return(relop,LE);
    else
    {
        retract();
        return(relop,LT);
    }
    break;
case '=':
    character=getchar();
    if(character== '=')
        return(relop,EQ);
    else
    {
        retract();
```

```
            return('=', null);
        }
        break;
    case ';':
        return(';', null);
        break;
    default:
        error( );
}
```

2.3　正规表达式与有限自动机简介

2.3.1　正规表达式与正规集

状态转换图对构造词法分析程序是行之有效的，为了便于词法分析器的自动生成，还须将状态转换图的概念加以形式化。正规表达式就是一种形式化的表示法，它可以表示单词符号的结构，从而精确地定义单词符号集。正规表达式简称为正规式，它表示的集合即为正规集。

为了理解正规式与正规集的含义，我们以程序语言中的标识符为例予以说明。程序语言中使用的标识符是一个以字母开头的字母数字串，如果字母用 letter 表示，数字用 digit 表示，则标识符可表示为

letter(letter | digit)*

其中，letter 与(letter | digit)*的并置表示两者的连接；括号中的“ | ”表示 letter 或 digit 两者选一；“ * ”表示零次或多次引用由“ * ”标记的表达式；(letter | digit)*是 letter | digit 的零次或多次并置，即表示一长度为 0、1、2、…的字母数字串；letter(letter | digit)*表示以字母开头的字母数字串，也即标识符集。letter(letter | digit)*就是表示标识符的正规式，而标识符集就是这个正规式所表示的正规集。

对于给定的字母表 Σ，正规式和正规集的递归定义如下：

(1) ε 和 Φ 都是 Σ 上的正规式，它们所表示的正规集分别为{ ε }和 Φ 。

(2) 对任一个 a∈ Σ，a 是 Σ 上的一个正规式，它所表示的正规集为{a}。

(3) 如果 R 和 S 是 Σ 上的正规式，它们所表示的正规集分别为 L(R)和 L(S)，则：

① R | S 是 Σ 上的正规式，它所表示的正规集为 L(R) ∪ L(S)；

② R • S 是 Σ 上的正规式，它所表示的正规集为 L(R)L(S)；

③ (R)* 是 Σ 上的正规式，它所表示的正规集为(L(R))*；

④ R 也是 Σ 上的正规式，它所表示的正规集为 L(R)。

(4) 仅由有限次使用规则(1)～(3)得到的表示式是 Σ 上的正规式，它所表示的集合是 Σ 上的正规集。

在上述定义中，规则(1)、(2)为基础规则，规则(3)为归纳规则，规则(4)是界限规则或终

止规则。此外，Σ上的一个字是指由Σ中的字符所构成的一个有穷序列；不包含任何字符的序列称为空字，用ε表示。我们用Σ^*表示Σ上所有字的全体，则空字ε也在其中。例如，若Σ={a,b}，则Σ^*={ε,a,b,aa,ab,ba,bb,aaa,…}。我们还用Φ表示不含任何元素的空集{}。这里需要注意ε、{}和{ε}的区别：{ε}是由空字组成的集合，而{}则表示不含任何字的集合。

正规式间的运算符“|”表示或，“•”表示连接(通常可省略)，“*”表示闭包，使用括号可以改变运算的次序。如果规定“*”优先于“•”，“•”优先于“|”，则在不出现混淆的情况下括号也可以省去。注意，Σ^*的正规式R和S的连接可以形式化地定义为

$$RS=\{\alpha\beta \mid \alpha\in R\&\beta\in S\}$$

即集合RS中的字是由R和S中的字连接而成的，且R自身的n次连接记为

$$R^n=\underbrace{RR\cdots R}_{n\text{个}}$$

我们规定$R^0=\{\varepsilon\}$，并令$R^*=R^0\cup R^1\cup R^2\cup R^3\cup\cdots$，则称$R^*$是R的闭包；此外，令$R^+=RR^*$，并称$R^+$是R的正则闭包。闭包$R^*$中的每个字都是由R中的字经过有限次连接而生成的。

对于Σ上的正规式R和S，如果它所表示的正规集L(R)=L(S)，则称R和S等价并记为R=S。不难证明，正规式具有下列性质：

(1) 交换律：R|S=S|R。

(2) 结合律：R|(S|T)=(R|S)|T；　　R(ST)=(RS)T。

(3) 分配律：R(S|T)=RS|RT；　　(R|S)T=RT|ST。

(4) 同一律：εR=Rε=R。

例2.1 令Σ={a,b}，设R=a(a|b)*是Σ上的正规式，试求其表示的正规集。

[解答] $L(R)=L(a(a\mid b)^*)=L(a)L((a\mid b)^*)=L(a)(L(a\mid b))^*=L(a)(L(a)\cup L(b))^*$

$=\{a\}(\{a\}\cup\{b\})^*=\{a\}\{a,b\}^*=\{a\}\{\varepsilon,a,b,aa,ab,ba,bb,aaa,\cdots\}$

$=\{a,aa,ab,aaa,aab,aba,abb,aaaa,\cdots\}$

例2.2 判断下述正规式之间是否等价：

(1) $(a\mid b)^*$与$a^*\mid b^*$　　(2) $(ab)^*$与a^*b^*　　(3) $(a\mid b)^*$与$(a^*b^*)^*$

[解答] (1) $(a\mid b)^*$对应的正规集其a、b可任意交替出现，如abbaaaba…；而$(a^*\mid b^*)$对应的正规集只可出现任意个a或者任意个b；因此两者不等价。

(2) $(ab)^*$对应的正规集是以任意个ab对出现的，即ababab…；而a^*b^*对应的正规集则是先出现任意个a后接任意个b，即a…ab…b；因此两者不等价。

(3) 由于$(a\mid b)^*$对应的正规集其a、b可任意交替出现，如aababbb；而$(a^*b^*)^*$可采用如下构造方法得到字aababbb：

$$(a^*b^*)^2=(a^*b^*)^0\cup(a^2b^1)^1\cup(a^1b^3)^2=aababbb$$

反之，对$(a^*b^*)^*$产生的任意字也可由$(a\mid b)^*$得到，即两者是等价的。

例2.3 证明：设$L(a^+)=\{a\}^*-\{\varepsilon\}$，则有$a^+=aa^*$。

[证明] $L(a^+)=\{a\}^*-\{\varepsilon\}=\{\varepsilon,a,a^2,a^3,\cdots\}-\{\varepsilon\}$

$=\{a,a^2,a^3,\cdots\}=\{a\}\cdot\{\varepsilon,a,a^2,\cdots\}$

$=\{a\}\{a\}^*=L(a)L(a^*)=L(aa^*)$

故　　$a^+=aa^*$

2.3.2　有限自动机

有限自动机(FA)是更一般化的状态转换图，它分为确定有限自动机 DFA 和非确定有限自动机 NFA 两种。

1．确定有限自动机(DFA)

一个确定的有限自动机 M_d(记为 DFA M_d)是一个五元组 $M_d=(S, \Sigma, f, s_0, Z)$，其中：

(1) S 是一个有限状态集，它的每一个元素称为一个状态。

(2) Σ是一个有穷输入字母表，它的每一个元素称为一个输入字符。

(3) f 是一个从 $S \times \Sigma$ 到 S 的单值映射，即 $f(s_i, a)=s_j$ 且有 s_i、$s_j \in S$ 和 $a \in \Sigma$。

(4) $s_0 \in S$，是唯一的一个初态。

(5) $Z \subset S$，是一个终态集。

注意：$f(s_i, a)=s_j$ 表示当前状态为 s_i 且输入字符为 a 时，自动机将转换到下一个状态 s_j，也即 s_j 称为 s_i 的一个后继状态。状态转换函数 f 是单值函数，$f(s_i, a)$ 唯一确定了下一个要转换的状态，即由每个状态发出的有向边(输出边)上所标记的输入字符各不相同。

例如，对图 2-6 所给出的状态 s_1 有：

$$f(s_1, a)=s_2$$
$$f(s_1, b)=s_3$$
$$f(s_1, c)=s_4$$

因此，f 是单值映射函数。

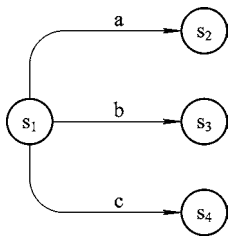

图 2-6　DFA 的状态转换示意

2．非确定有限自动机(NFA)

一个非确定有限自动机 M_n(记为 NFA M_n)是一个五元组 $M_n=(S, \Sigma, f, Q, Z)$，其中：

(1) S、Σ、Z 的意义与 DFA 相同。

(2) f 是一个从 $S \times \Sigma^*$ 到 S 的子集映射。

(3) $Q \subset S$，是一个非空初态集。

NFA 和 DFA 的区别主要有两点：其一是 NFA 可以有若干个初始状态，而 DFA 仅有一个初始状态；其二是 NFA 的状态转换函数 f 不是单值函数，而是一个多值函数，即 $f(s_i, a)=$ {某些状态的集合}($s_i \in S$)，它表示不能由当前状态和当前输入字符唯一地确定下一个要转换的状态，也即允许同一个状态对同一个输入字符可以有不同的输出边。

例如，对图 2-7 所给出的状态 s_1 有：

$$f(s_1, a)=\{s_1, s_2, s_3\}$$

即 f 是一个从 $S \times \Sigma^*$ 到 S 的子集映射；Σ^* 表示输出边上所标记的不仅是字符，也可以是字。此外，NFA 还允许 $f(s_1, \varepsilon)=$ {某些状态的集合}，即在 NFA 的状态转换图中输出边上的标记还可是 ε(空字)。

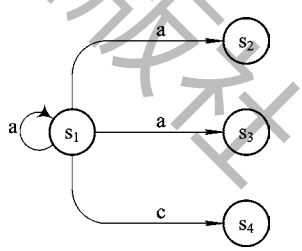

图 2-7　NFA 的状态转换示意

3．状态转换图与状态转换矩阵

DFA 和 NFA 都可以用状态转换图表示。假定 DFA(或 NFA)有 m 个状态、n 个输入字符(或字)，则这个状态转换图含有 m 个状态，每个状态最多有 n 条输出边与其它状态相连接，每一条输出边用 Σ(或 Σ^*)中的一个不同的输入字符(或一个输入字)作标记，整个图含有唯一一个初态(或多个初态)和若干个终态。

DFA 和 NFA 也可以用状态转换矩阵表示。状态转换矩阵的行表示状态，列表示输入符号，矩阵元素表示 $f(s_i, a)$ 的值。

例 2.4 假定 DFA $M_d=(\{s_0, s_1, s_2\}, \{a, b\}, f, s_0, \{s_2\})$，且有：

$f(s_0, a)=s_1$　　$f(s_0, b)=s_2$

$f(s_1, a)=s_1$　　$f(s_1, b)=s_2$

$f(s_2, a)=s_2$　　$f(s_2, b)=s_1$

试给出 DFA M_d 的状态转换图与状态转换矩阵。

[解答] DFA M_d 的状态转换图见图 2-8，状态转换矩阵见表 2.2。

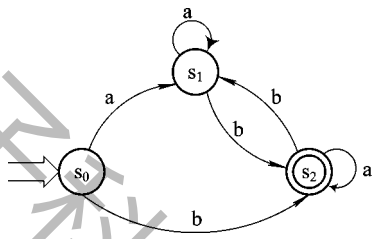

图 2-8 例 2.4 的 DFA M_d 状态转换图

表 2.2 状态转换矩阵

状态 \ 字符	a	b
s_0	s_1	s_2
s_1	s_1	s_2
s_2	s_2	s_1

例 2.5 假定 NFA $M_n=(\{s_0, s_1, s_2\}, \{a,b\}, f, \{s_0, s_2\}, \{s_1\})$，且有：

$f(s_0, a)=\{s_2\}$　　$f(s_0, b)=\{s_0, s_1\}$

$f(s_1, a)=\Phi$　　$f(s_1, b)=\{s_2\}$

$f(s_2, a)=\Phi$　　$f(s_2, b)=\{s_1\}$

试给出 NFA M_n 的状态转换图与状态转换矩阵。

[解答] NFA M_n 的状态转换图见图 2-9，状态转换矩阵见表 2.3。

对于 FA M 和任一 Σ 上的字符串 α(即 $\alpha \in \Sigma^*$)，如果存在一条从初始状态到终止状态的通路，通路上有向边(输出边)所标识的字符依次连接所得到的字符串恰为 α，则称 α 可以为 FA M 所接受或称 α 为 FA M 所识别。FA M 所能识别的字符串集称为 FA M 所识别的语言，记为 L(M)。

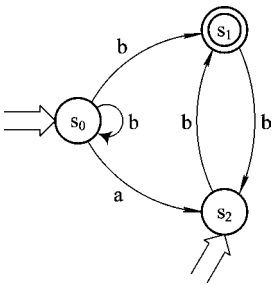

图 2-9　例 2.5 的 NFA M_n 的状态转换图

表 2.3　状态转换矩阵

状态 \ 字	a	b
s_0	$\{s_2\}$	$\{s_0, s_2\}$
s_1	Φ	$\{s_2\}$
s_2	Φ	$\{s_1\}$

注意：对于任何两个 FA M 和 FA M'，如果 L(M) =L(M')，则称有限自动机 M 和 M'等价。此外，对于任一给定的 NFA M，一定存在一个 DFA M'，使 L(M) =L(M')。因此，DFA 是 NFA 的特例，NFA 可以有 DFA 与之等价，即两者描述能力相同。DFA 便于识别，易于计算机实现，而 NFA 便于定理的证明。

2.4　正规表达式到有限自动机的构造

由正规表达式与有限自动机的等价性可知：如果 R 是 Σ 上的一个正规表达式，则必然存在一个 NFA M，使得 L(M)= L(R)；反之亦然。

2.4.1　由正规表达式构造等价的非确定有限自动机(NFA)

由正规表达式构造等价的 NFA M 的方法如下：

(1) 将正规表达式 R 表示成如图 2-10 所示的拓广转换图。

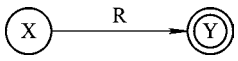

图 2-10　拓广转换图

(2) 对正规表达式采用如图 2-11 所示的三条转换规则来构造 NFA M。

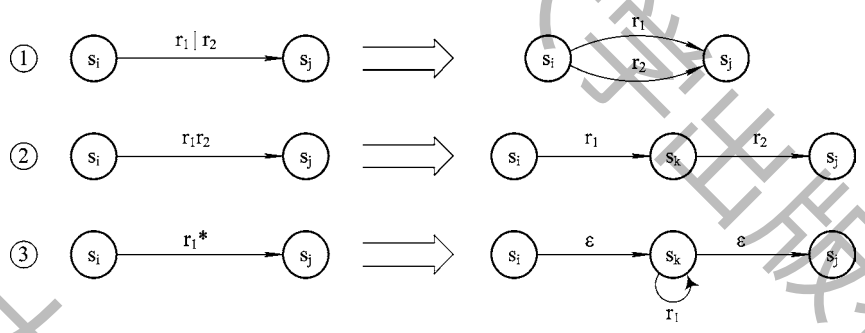

图 2-11　转换规则

对于给定的正规表达式 R，首先将其表示成如图 2-10 所示的形式，其中 X 为初始状态，Y 为终止状态；然后逐步将这个拓广转换图运用图 2-11 的三条转换规则不断加入新结点进行分解，直至每条有向边上仅标识有 Σ 的一个字母或 ε 为止，则 NFA M 构造完成。

例 2.6　对给定正规表达式 $b^*(d | ad)(b | ab)^+$构造其 NFA M。

[解答]　先用 $R^+=RR^*$改造题设的正规表达式为 $b^*(d | ad)(b | ab)(b | ab)^*$，然后构造其 NFA M，如图 2-12 所示。

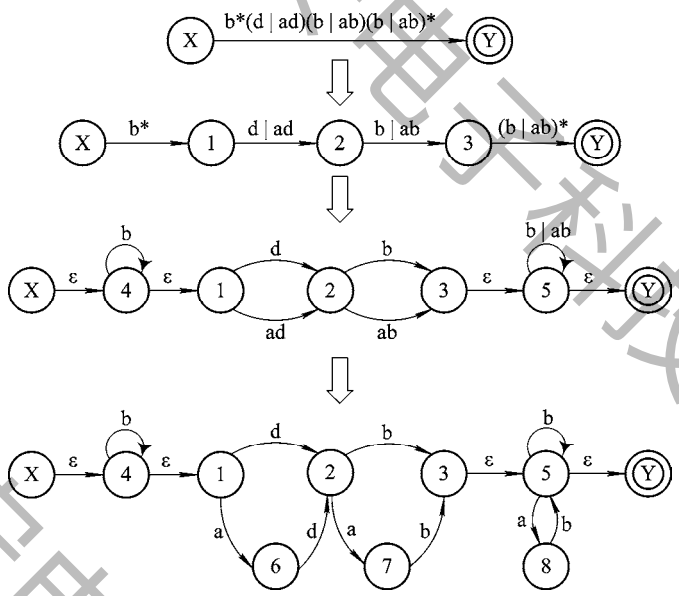

图 2-12　例 2.6 的 NFA M

注意，拓广后的状态转换图中如果有下面两种情况之一存在，则为 NFA M：

(1) 有 ε 边存在。

(2) 某结点对同一输入字符存在多条输出边(即为多值映射)。

2.4.2　NFA 的确定化

NFA 的确定化是指对给定的 NFA M 都能相应地构造出一个与之等价的 DFA M，使它们能够识别相同的语言。我们采用子集法来对 NFA M 确定化。

首先定义 NFA M 的一个状态子集 I 的 ε_闭包，即 ε_CLOSURE(I)，则：

(1) 若 $s_i \in I$，则 $s_i \in \varepsilon\_CLOSURE(I)$。

(2) 若 $s_i \in I$，则对从 s_i 出发经过 ε 通路所能到达的任何状态 s_j，都有 $s_j \in \varepsilon\_CLOSURE(I)$。

其次，对 FA M 的一个状态子集 I，若 a 是 Σ 中的一个字符，定义

$$I_a=\varepsilon\_CLOSURE(J)$$

其中，J 是所有那些可以从 I 中的某一状态出发经过有向边 a 而能到达的状态集。

例 2.7　已知一状态转换图如图 2-13 所示，且假定 I = ε_{1} = {1, 2}，试求从状态 I 出发经过一条有向边 a 而能到达的状态集 J 和 ε_CLOSURE(J)。

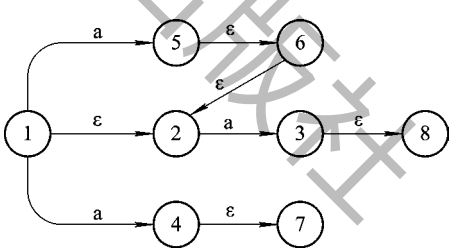

图 2-13　例 2.7 的状态转换图

[解答]　从状态 I 中的状态 1 或状态 2 出发经过一条 a 弧而能到达的状态集 J 为

{5，3，4}

若 $s_i \in J$，则由 s_i 出发经过任意条 ε 有向边而能到达的任何状态 $s_j \in \varepsilon\_CLOSURE(J)$，因此 ε_CLOSURE(J) 为

{5，6，2，3，8，4，7}

用子集法对 NFA M 确定化的方法如下：

(1) 构造一张转换表，其第一列为状态子集 I，对不同的 a(a∈Σ)在表中单设一列 I_a。

(2) 表的第一行第一列其状态子集 I 为 ε _CLOSURE(s_0)；其中，s_0 为初始状态。

(3) 根据第一列中的 I 为每一个 a 求其 I_a 并记入对应的 I_a 列中，如果此 I_a 不同于第一列已存在的所有状态子集 I，则将其顺序列入空行中的第一列。

(4) 重复步骤(3)直至对每个 I 及 a 均已求得 I_a，并且无新的状态子集 I_a 加入第一列时为止；此过程可在有限步后终止。

(5) 重新命名第一列的每一状态子集，则转换表便成为新的状态转换矩阵，其状态转换函数 f 是 S × Σ 到 S 的单值映射，即为与 NFA M 等价的 DFA M'。

例 2.8　正规表达式$(a|b)^*(aa|bb)(a|b)^*$的 NFA M 如图 2-14 所示，试将其确定化为 DFA M'。

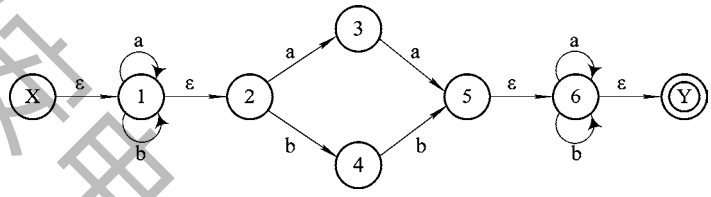

图 2-14　例 2.8 的 NFA M

[解答]　用子集法将图 2-14 所示的 NFA M 确定化为表 2.4。

表 2.4　例 2.8 的转换表

I	I_a	I_b
{X,1,2}	{1,2,3}	{1,2,4}
{1,2,3}	{1,2,3,5,6,Y}	{1,2,4}
{1,2,4}	{1,2,3}	{1,2,4,5,6,Y}
{1,2,3,5,6,Y}	{1,2,3,5,6,Y}	{1,2,4,6,Y}
{1,2,4,5,6,Y}	{1,2,3,6,Y}	{1,2,4,5,6,Y}
{1,2,4,6,Y}	{1,2,3,6,Y}	{1,2,4,5,6,Y}
{1,2,3,6,Y}	{1,2,3,5,6,Y}	{1,2,4,6,Y}

对表 2.4 中的所有子集重新命名，得到表 2.5 的状态转换矩阵及对应的状态转换图(见图 2-15)。注意，状态 3、4、5、6 因其原来对应的子集中含有终态 Y 而均为终态。

表 2.5　例 2.8 的状态转换矩阵

S	a	b
0	1	2
1	3	2
2	1	4
3	3	5
4	6	4
5	6	4
6	3	5

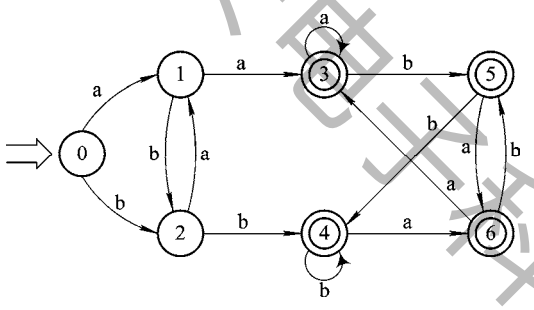

图 2-15 例 2.8 未化简的 DFA M'

2.4.3 确定有限自动机(DFA)的化简

对 NFA 确定化后所得到的 DFA 可能含有多余的状态，因此还应对其进行化简。所谓 DFA M 的化简，是指寻找一个状态数比 M 少的 DFA M'，使得 L(M)=L(M')。化简了的 DFA M' 满足下述两个条件：

(1) 没有多余状态。

(2) 在其状态集中，没有两个相互等价的状态存在。

所谓两个状态相互等价是指：对一给定的 DFA M，若存在状态 s_1、$s_2 \in S$ 且 $s_1 \neq s_2$，如果从 s_1 出发能识别字符串 α 而停于终态，从 s_2 出发也同样能够识别这个 α 而停于终态；反之，若从 s_2 出发能识别字符串 β 而停于终态，则从 s_1 出发也能识别这个 β 而停于终态，则称 s_1 和 s_2 是等价的，否则就是可区分的。

一个 DFA M 的状态最小化过程是将 M 的状态集分割成一些不相交的子集，使得任何不同的两个子集其状态都是可区分的，而同一子集中的任何两个状态都是等价的。最后，从每个子集中选出一种状态，同时消去其它等价状态，就得到最简的 DFA M' 且 L(M)=L(M')。

DFA M 的化简方法如下：

(1) 首先将 DFA M 的状态集 S 中的终态与非终态分开，形成两个子集，即得到基本划分。

(2) 对当前已划分出的 $I^{(1)}$、$I^{(2)}$、…、$I^{(m)}$子集(属于不同子集的状态是可区分的)，看每一个 I 是否能进一步划分；也即对某个 $I^{(i)}=\{s_1, s_2, \cdots, s_k\}$，若存在一个输入字符 $a(a \in \Sigma)$ 使得 $I_a^{(i)}$不全包含在当前划分的某一子集 $I^{(j)}$中(即跨越到两个子集)，就将 $I^{(i)}$一分为二(见图 2-16)。

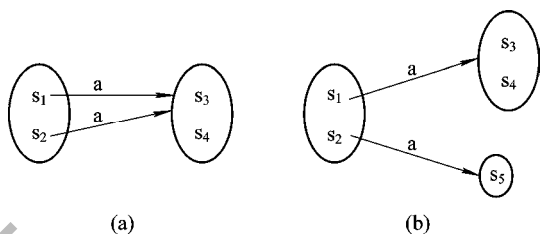

图 2-16 是否划分示意

(a) 无需划分；(b) 需要划分

(3) 重复步骤(2)，直到子集个数不再增加为止(即每个子集已是不可再分的了)。所谓不

能划分，是指该子集或者仅有一个状态，或者虽有多个状态但这些状态均不可区分(即等价)。

那么，如何进行划分呢？假定当前子集 $I^{(i)}=\{s_1, s_2, \cdots\}$，且状态 s_1 和 s_2 经过有向边 a 分别到达状态 t_1 和 t_2，而 t_1 和 t_2 又分属于当前已划分出的两个不同子集 $I^{(j)}$和 $I^{(k)}$，则此时应将 $I^{(i)}$分为两部分，使得一部分含有 s_1：

$$I^{(i1)}=\{s \mid s\in I^{(i)}\text{且 } s \text{ 经有向边 a 到达 } t_1\}$$

而另一部分含有 s_2：

$$I^{(i2)}=I^{(i)}-I^{(i1)}$$

由于 t_1 和 t_2 是可区分的，即存在一个字符串 α，t_1 将读出 α 而停于终态，而 t_2 或读不出 α 或可以读出 α 但不能到达终态。因此，字符串 α 将把状态 s_1 和 s_2 区分开来，也就是说，$I^{(i1)}$中的状态与 $I^{(i2)}$中的状态是可区分的。至此，我们已将 $I^{(i)}$ 划分为两个子集 $I^{(i1)}$和 $I^{(i2)}$，形成了新的划分。

当子集个数不再增加时，就得到一个最终划分。对最终划分的每一个子集，我们选取子集中的一个状态作为代表。例如，假定 $I^{(i)}=\{s_1, s_2, s_3\}$是这样一个子集，且我们挑选了 s_1 代表这个子集。这时，凡在原来 DFA M 中有指向 s_2 和 s_3 的有向边均改为在新的 DFA M' 中指向 s_1。改向之后，就可将 s_2 和 s_3 从原来的状态集 S 中删除了。若 $I^{(i)}$中含有原来的初态，则 s_1 就是 DFA M' 中的新初态；若 $I^{(i)}$中含有原来的终态，则 s_1 就是 DFA M' 中的新终态。此时，DFA M' 已是最简的了(含有最少的状态)。

例 2.9　化简由例 2.8 得到的 DFA M。

[解答]　(1) 首先将状态集 S={0,1,2,3,4,5,6}划分为终态集{3,4,5,6}和非终态集{0,1,2}。

(2) 考察$\{0,1,2\}_a$：因 $0_a=2_a=\{1\}$，而 $1_a=\{3\}$，分属于非终态集和终态集，故将{0,1,2}划分为{0,2}和{1}。

(3) 考察$\{0,2\}_b$：$0_b=\{2\}$，$2_b=\{4\}$，它们分属于两个不同的状态集，故{0,2}划分为{0}和{2}。

(4) 考察$\{3,4,5,6\}_a$：$3_a=6_a=\{3\}\subset\{3,4,5,6\}$；$4_a=5_a=\{6\}\subset\{3,4,5,6\}$，即都属于终态集，故不进行划分。

(5) 考察$\{3,4,5,6\}_b$：$3_b=6_b=\{5\}\subset\{3,4,5,6\}$；$4_b=5_b=\{4\}\subset\{3,4,5,6\}$，即都属于终态集，故不进行划分。

(6) 按顺序重新命名状态子集{0}、{1}、{2}、{3,4,5,6}为 0、1、2、3，则得到化简后的状态转换矩阵(见表 2.6)和 DFA M''(见图 2-17)。

表 2.6　例 2.9 的状态转换矩阵

S	a	b
0	1	2
1	3	2
2	1	3
3	3	3

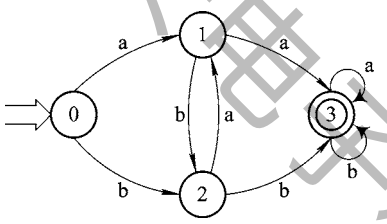

图 2-17 例 2.9 化简后的 DFA M"

2.4.4 正规表达式到有限自动机构造示例

例 2.10 试用 DFA 的等价性证明正规表达式$(a|b)^*$与$(a^*b^*)^*$等价。

[解答] (1) 正规表达式$(a|b)^*$对应的 NFA M 如图 2-18 所示。

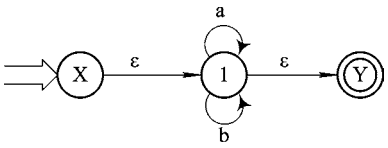

图 2-18 $(a|b)^*$的 NFA M

用子集法将图 2-18 所示的 NFA M 确定化得到如表 2.7 所列的转换表，重新命名后得到如表 2.8 所列的状态转换矩阵。

表 2.7 (a | b)*的转换表

I	I_a	I_b
{X,1,Y}	{1,Y}	{1,Y}
{1,Y}	{1,Y}	{1,Y}

表 2.8 (a | b)*的状态转换矩阵

S	a	b
0	1	1
1	1	1

由于状态 0 和状态 1 均为终态，无论输入什么字符，其下一状态仍是终态，故最简 DFA M 如图 2-19 所示。

(2) 正规表达式$(a^*b^*)^*$对应的 NFA M 如图 2-20 所示。

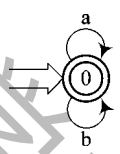

图 2-19 $(a|b)^*$的最简 DFA M

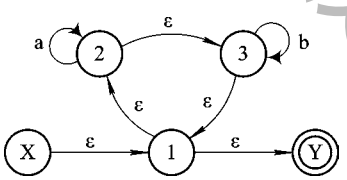

图 2-20 $(a^*b^*)^*$的 NFA M

用子集法将图 2-20 所示的 NFA M 确定化得到如表 2.9 所列的转换表，重新命名得到如表 2.10 所列的状态转换矩阵。

表 2.9　$(a^*b^*)^*$的转换表

I	I_a	I_b
{X,1,2,3,Y}	{2,3,1,Y}	{2,3,1,Y}
{2,3,1,Y}	{2,3,1,Y}	{2,3,1,Y}

表 2.10　$(a^*b^*)^*$的状态转换矩阵

S	a	b
0	1	1
1	1	1

由于表 2.8 和表 2.10 的状态转换矩阵相同，故$(a \mid b)^*$和$(a^*b^*)^*$等价。

注意，对正规表达式 a^*b^*构造 DFA M，则首先画出 NFA M 如图 2-21 所示，用子集法将图 2-21 所示的 NFA M 确定化得到如表 2.11 所列的转换表，重新命名得到如表 2.12 所列的状态转换矩阵。

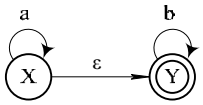

图 2-21　a^*b^*的 NFA M

表 2.11　a*b*的转换表

I	I_a	I_b
{X,Y}	{X,Y}	{Y}
{Y}	—	{Y}

表 2.12　a*b*的状态转换矩阵

S	a	b
0	0	1
1	—	1

(注：表中的空集Φ一律用“—”表示)

虽然状态 0 和状态 1 都是终态，但两者面对字符 a 转换的下一状态是不一样的：0_a=0，1_a=Φ(即“—”)；也即状态 0 和状态 1 不等价，故不可将图 2-21 的 DFA　M 合并成图 2-19 的 DFA M。因此，正规表达式 a^*b^*根据表 2.12 最终得到的 DFA M 如图 2-22 所示。

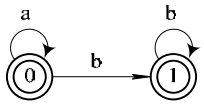

图 2-22　a^*b^*的 DFA M

例 2.11　C 语言可接受的合法的文件名为 device:name.extension，其中第一部分(device:)和第三部分(.extension)可缺省。若 device、name 和 extension 都是字母串，长度不限，但至少为 1，试画出识别这种文件名的 DFA M。

[解答]　以字母“c”代表字母，则所求正规式为

$$(cc^*: \mid \varepsilon)cc^*(.cc^* \mid \varepsilon)$$

应用例 2.10 由图 2-18 化简为图 2-19 的结论，去掉多余的 ε 弧，得到的 NFA M 如图 2-23 所示。

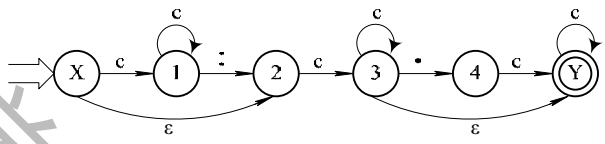

图 2-23　例 2.11 的 NFA M

用子集法将 NFA　M 确定化，得到如表 2.13 所列的转换表；重新命名后得到如表 2.14 所列的状态转换矩阵和如图 2-24 所示的 DFA M'。

表 2.13 例 2.11 的转换表

I	I_c	$I_:$	$I_.$
{X, 2}	{1, 3, Y}	—	—
{1, 3, Y}	{1, 3, Y}	{2}	{4}
{2}	{3, Y}	—	—
{4}	{Y}	—	—
{3, Y}	{3, Y}	—	{4}
{Y}	{Y}	—	—

重新命名

表 2.14 例 2.11 的状态转换矩阵

S	c	:	.
0	1	—	—
1	1	2	3
2	4	—	—
3	5	—	—
4	4	—	3
5	5	—	—

由表 2.14 可以看出，终态集中的状态 1 对应三种输入字符的下一状态均存在，状态 4 对应三种输入字符的下一状态存在两个，而状态 5 对应三种输入字符的下一状态仅有一个存在，故状态 1、4、5 都是可区分的。相应地，非终态 0、2、3 对应输入字母 c 的下一状态也都是可区分的了。因此，图 2-24 的 DFA M' 已为最简。

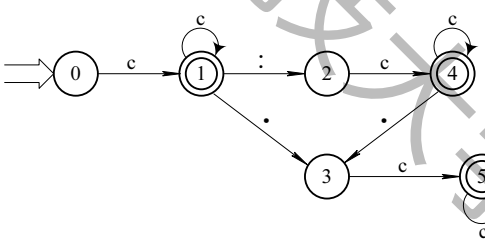

图 2-24 例 2.11 的 DFA M'

例 2.12 某高级程序语言无符号数的正规表达式为

$$\text{digit}^{+}[.\text{digit}^{+}][e[+|-]\text{digit}^{+}]$$

其中，digit 表示数字，“[]”表示“[]”中的内容可有可无，试给出其 DFA M。

[解答] 我们用 d 代表 digit，“[]”中的内容为无时用 ε 表示，则本题的正规式又可表示为

$$dd^{*}(.\,dd^{*}\mid\varepsilon)(e(+|-\mid\varepsilon)dd^{*}\mid\varepsilon)$$

由此，用状态转换图表示接受无符号数的 NFA M 如图 2-25 所示。

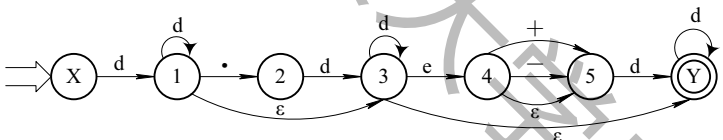

图 2-25 例 2.12 的 NFA M

用子集法将 NFA M 确定化，得到如表 2.15 所列的转换表(I_+和 I_-用 $I_\pm$表示在同一列中)；重新命名后得到如表 2.16 所列的状态转换矩阵和如图 2-26 所示的 DFA M'。

表 2.15 例 2.12 的转换表

I	$I_\pm$	I_d	$I_.$	I_e
{X}	—	{1, 3, Y}	—	—
{1, 3, Y}	—	{1, 3, Y}	{2}	{4, 5}
{2}	—	{3, Y}	—	—
{4, 5}	{5}	{Y}	—	—
{3, Y}	—	{3, Y}	—	{4, 5}
{5}	—	{Y}	—	—
{Y}	—	{Y}	—	—

重新命名

表 2.16 例 2.12 的状态转换矩阵

S	±	d	.	e
0	—	1	—	—
1	—	1	2	3
2	—	4	—	—
3	5	6	—	—
4	—	4	—	3
5	—	6	—	—
6	—	6	—	—

根据表 2.16 的化简过程如下：首先分为非终态和终态两个集合{0，2，3，5}和{1，4，6}。由非终态集{0，2，3，5}可知，状态 3 面对输入符号“+”、“−”的下一状态与状态 0、2、5 不同，故将状态 3 划分出来。终态集{1，4，6}中的状态 1、4、6 面对不同输入符号的下一非空状态分别有 3、2、1 种，故它们都是可区分的，并由此导致状态 0、2、5 面对输入符号“d”的下一状态也不相同，即状态 0、2、5 均可区分。故图 2-26 的 DFA M'已为最简。

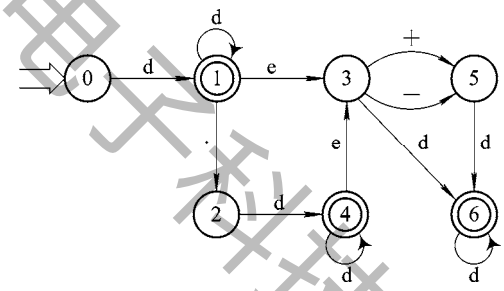

图 2-26　例 2.12 的 DFA M'

例 2.13　构造一个 DFA M，它接收 Σ={a,b}上所有满足下述条件的字符串：该字符串中的每个 a 都有至少一个 b 直接跟在其右边。

[解答]　已知 Σ={a,b}，根据题意得到正规表达式为 $b^*(abb^*)^*$。根据此正规表达式画出相应的 NFA M 如图 2-27 所示。

用子集法将 NFA M 确定化，得到如表 2.17 所列的转换表；重新命名后得到如表 2.18 所列的状态转换矩阵和如图 2-28 所示的 DFA M'。

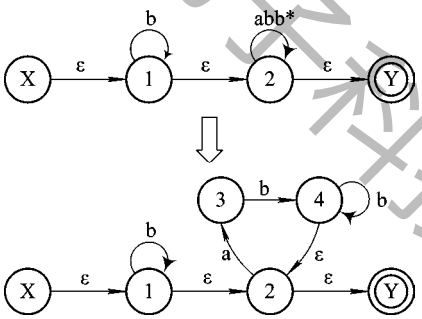

图 2-27　例 2.13 的 NFA M(一)

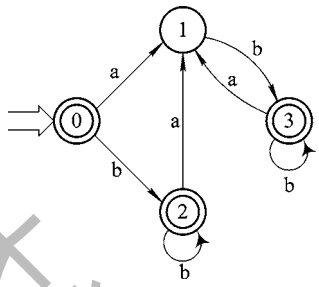

图 2-28　例 2.13 的 DFA M'

表 2.17　例 2.13 的转换表(一)

I	I_a	I_b
{X, 1, 2, Y}	{3}	{1, 2, Y}
{3}	—	{4, 2, Y}
{1, 2, Y}	{3}	{1, 2, Y}
{4, 2, Y}	{3}	{4, 2, Y}

重新命名

表 2.18　例 2.13 的状态转换矩阵

S	a	b
0	1	2
1	—	3
2	1	2
3	1	3

用 DFA M 的化简方法先得到一个初始划分，即终态集为{0,2,3}，非终态集为{1}；由化简方法可知这已是最终划分(状态 0、状态 2、状态 3 均为等价状态)，重新命名终态集为 0、非终态集为 1 后得到最终化简的 DFA M" 如图 2-29 所示。

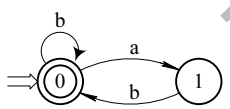

图 2-29　例 2.13 最终化简后的 DFA M"

实际上，根据正规表达式 $b^*(abb^*)$ 可以看出，当 b^* 和 $(abb^*)^*$(在此指括号外的这个“*”)的闭包*都取 0 时，则由初态 X 到终态 Y 有一条 ε 通路，即得到图 2-30。由图 2-30 的状态

Y 上的正规式 abb*可以看出，当 b*的闭包*取 0 时，则由状态 Y 出发，经过 a 和 b 又应回到状态 Y，而 b*则可描述为由状态 Y 出发又回到状态 Y 的一条标记为 b 的有向边，这样就得到图 2-31。

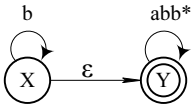

图 2-30　例 2.13 的 NFA M(二)

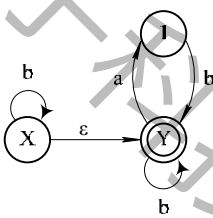

图 2-31　例 2.13 的 NFA M(三)

用子集法对图 2-31 进行确定化，得到表 2.19，重新命名并进行化简，最终可得到图 2-29。

表 2.19　例 2.13 的转换表(二)

I	I_a	I_b
{X,Y}	{1}	{X,Y}
{1}	—	{Y}
{Y}	{1}	{Y}

此题根据题意也可得到正规表达式为(ab｜b)*，最终同样可化简为图 2-29。

例 2.14　构造一个 DFA M，它接收 Σ={a,b}上所有含奇数个 a 的字符串。

[解答]　根据题意，我们首先可以构造出字符串中含偶数个 a 的正规表达式：(b｜ab*a)*，然后在其之前添加一个 a 即为奇数个 a。因此，得到含奇数个 a 的字符串的正规表达式为：b*a(b｜ab*a)*。根据此正规表达式画出相应的 NFA M 如图 2-32 所示。图 2-32 中每条边上为单个字符且是单值映射，并且无 ε 边出现，故已是 DFA M；体现在表 2-20 中，对每个输入字符的输出，即子集映射此时已全部是单个字符，即为单值映射。

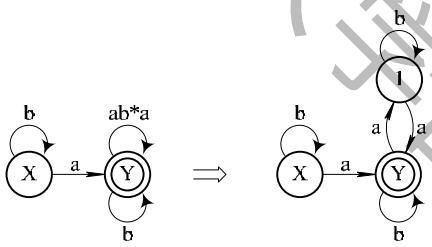

图 2-32　例 2.14 的 NFA M

用子集法将图 2-32 的 NFA M(由于是单值映射，故已为 DFA M)确定化，得到如表 2.20 所列的转换表；重新命名后得到如表 2.21 所列的状态转换矩阵。

表 2.20　例 2.14 的转换表

I	I_a	I_b
{X}	{Y}	{X}
{Y}	{1}	{Y}
{1}	{Y}	{1}

重新命名

表 2.21　例 2.14 的状态转换矩阵

S	a	b
0	1	0
1	2	1
2	1	2

根据表 2.21 的状态转换矩阵进行最小化，得到两个初始划分：{0,2}和{1}；由于状态 0 和状态 2 为等价状态，删去状态 2 即得到最简 DFA M，如图 2-33 所示。

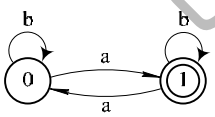

图 2-33　例 2.14 最终化简后的 DFA M

最后需要说明的是，采用如图 2-34 所示的三条转换规则可以实现从 FA M 到正规表达式的转换。

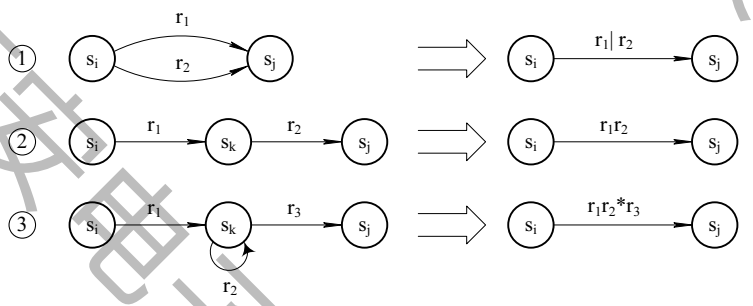

图 2-34　转换规则

2.5　词法分析器的自动生成

由于各种不同的高级程序语言中单词的总体结构大致相同，基本上都可用一组正规表达式描述，因而人们希望构造这样的自动生成系统：只要给出某高级语言各类单词词法结构的一组正规表达式以及识别各类单词时词法分析程序应采取的语义动作，该系统便可自动产生此高级程序语言的词法分析程序。所生成的词法分析程序的作用如同一台有限自动机，可以用来识别和分析单词符号。正规表达式与有限自动机的理论研究产生了自动生成词法分析程序的技术和工具。

Lex 是由美国 Bell 实验室的 M.Lesk 和 Schmidt 于 1975 年用 C 语言研制的一个词法分析程序的自动生成工具。对任何高级程序语言，用户必须用正规表达式描述该语言的各个词法类(这一描述称为 Lex 的源程序)，Lex 就可以自动生成该语言的词法分析程序。Lex 及其编译系统的作用如图 2-35 所示。

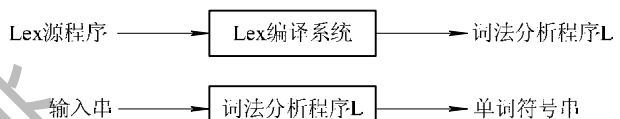

图 2-35　Lex 及其编译系统的作用

一个 Lex 源程序由用“%%”分隔的三部分组成：第一部分为正规式的辅助定义式，第二部分为识别规则，最后一部分为用户子程序。其书写格式为

辅助定义式
%%
识别规则
%%
用户子程序

其中，辅助定义式和用户子程序是任选的，而识别规则是必需的。如果用户子程序缺省，则第二个分隔符号“%%”可以省去；但如果无辅助定义式部分，第一个分隔符号“%%”不能省去，因为第一个分隔符号用于指示识别规则部分的开始。

下面给出一个简单语言的单词符号的 Lex 源程序例子，其输出单词的类别编码用整数编码表示：

```
Auxiliary Definitions                                    /*辅助定义*/
letter→A | B | C | … | Z | a | b | c | … | z
digit→0 | 1 | 2 | 3 | … | 9
%%
Recognition Rules                                        /*识别规则*/
1 while                  {return(1,null)}
2 do                     {return(2,null)}
3 if                     {return(3,null)}
4 else                   {return(4,null)}
5 switch                 {return(5,null)}
6 {                      {return(6,null)}
7 }                      {return(7,null)}
8 (                      {return(8,null)}
9 )                      {return(9,null)}
10 +                     {return(10,null)}
11 −                     {return(11,null)}
12 *                     {return(12,null)}
13 /                     {return(13,null)}
14 =                     {return(14,null)}
15 ;                     {return(15,null)}
16 letter (letter | digit)*      {if(keyword(id)==0)
                                    {return(16,null);
                                     return(id)};
                                  else return(keyword(id))}
17 digit (digit)*        {val=int(id);
                          return(17,null);
                          return(val)}
18 (letter | digit | { | } | ( | ) | + | − | * | / | = | ; )*
                         {return(18,null);
```

inslit(id)；
return(pointer, lenth)}

该 Lex 源程序中用户子程序为空；其中识别规则{A_{18}}语句中调用过程“inslit(id)”是指将字符串常量 id 存放到字符表中，“pointer”中存放该串的起始位置，“lenth”存放该串的长度。

Lex 有两种使用方式：一种是将 Lex 作为一个单独的工具，用以生成所需的识别程序；另一种是将 Lex 和语法分析器自动生成工具(如 YACC)结合起来使用，以生成一个编译程序的扫描器和语法分析器。

习　题　2

2.1　完成下列选择题：

(1) 词法分析所依据的是_____。

A．语义规则　　B．构词规则

C．语法规则　　D．等价变换规则

(2) 词法分析器的输入是_____。

A．单词符号串　　B．源程序

C．语法单位　　D．目标程序

(3) 词法分析器的输出是_____。

A．单词的种别编码　　B．单词的种别编码和自身的值

C．单词在符号表中的位置　　D．单词自身值

(4) 状态转换图(见图 2-36)接受的字集为 _____。

A．以 0 开头的二进制数组成的集合

B．以 0 结尾的二进制数组成的集合

C．含奇数个 0 的二进制数组成的集合

D．含偶数个 0 的二进制数组成的集合

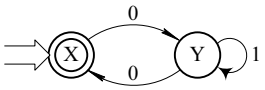

图 2-36　习题 2.1 的 DFA M

(5) 对于任一给定的 NFA M，_____一个 DFA M'，使 L(M) =L(M')。

A．一定不存在　　B．一定存在　　C．可能存在　　D．可能不存在

(6) DFA 适用于_____。

A．定理证明　　B．语法分析　　C．词法分析　　D．语义加工

(7) 下面用正规表达式描述词法的论述中，不正确的是_____。

A．词法规则简单，采用正规表达式已足以描述

B．正规表达式的表示比上下文无关文法更加简洁、直观和易于理解

C．正规表达式描述能力强于上下文无关文法

D．有限自动机的构造比下推自动机简单且分析效率高

(8) 与 (a | b)*(a | b) 等价的正规式是_____。

A．(a | b)(a | b)*　　B．a* | b*

C．(ab)*(a | b)*　　D．(a | b)*

(9) 在状态转换图的实现中，____一般对应一个循环语句。

A．不含回路的分叉结点　　B．含回路的状态结点

C．终态结点　　D．A～C 都不是

(10) 已知 DFA M_d= ({s_0, s_1, s_2}, {a, b}, f, s_0, {s_2})，且有：

$f(s_0, a) = s_1$　　$f(s_1, a) = s_2$

$f(s_2, a) = s_2$　　$f(s_2, b) = s_2$

则该 DFA M 所能接受的语言可以用正规表达式表示为____。

A．(a | b)*　　B．aa (a | b)*

C．(a | b)*aa　　D．a (a | b)*a

2.2 什么是扫描器？扫描器的功能是什么？

2.3 设 M= ({x, y}, {a, b}, f, x, {y}) 为一非确定的有限自动机，其中 f 定义如下：

f(x, a) = {x, y}　　f(x, b) = {y}

f(y, a) = Φ　　f(y, b) = {x, y}

试构造相应的确定有限自动机 M'。

2.4 正规式 (ab)*a 与正规式 a (ba)* 是否等价？请说明理由。

2.5 设有 $L(G) = \{a^{2n+1}b^{2m}a^{2p+1} \mid n \geqslant 0, p \geqslant 0, m \geqslant 1\}$。

(1) 给出描述该语言的正规表达式；

(2) 构造识别该语言的确定有限自动机(可直接用状态图形式给出)。

2.6 有语言 $L=\{w \mid w \in (0, 1)^+$，并且 w 中至少有两个 1，又在任何两个 1 之间有偶数个 0}，试构造接受该语言的确定有限状态自动机(DFA M)。

2.7 已知正规式 ((a | b)* | aa)*b 和正规式 (a | b)*b。

(1) 试用有限自动机的等价性证明这两个正规式是等价的；

(2) 给出相应的正规文法。

2.8 构造一个 DFA M，它接收 Σ = {a, b}上所有不含子串 abb 的字符串。

2.9 构造一个 DFA M，它接收 Σ = {a, b}上所有含偶数个 a 的字符串。

2.10 下列程序段以 B 表示循环体，A 表示初始化，I 表示增量，T 表示测试：

```
I=1;
while( I<=n )
{
   sum=sum+a[I];
   I=I+1;
}
```

请用正规表达式表示这个程序段可能的执行序列。

2.11 将图 2-37 所示的非确定有限自动机(NFA M)变换成等价的确定有限自动机(DFA M′)。其中，X 为初态，Y 为终态。

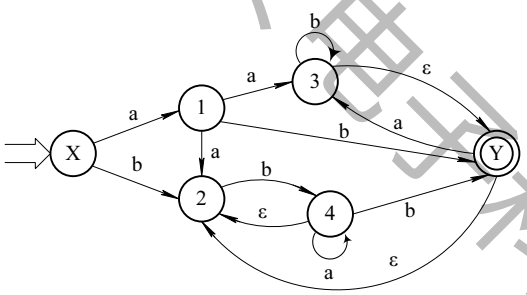

图 2-37　习题 2.11 的 NFA M

2.12　有一台自动售货机，接受 1 分和 2 分硬币，出售 3 分钱一块的硬糖。顾客每次向机器中投放大于等于 3 分的硬币，便可得到一块糖(注意：只给一块并且不找钱)。

(1) 写出售货机售糖的正规表达式；

(2) 构造识别上述正规式的最简 DFA M。

第3章 语 法 分 析

语法分析是编译过程的核心部分，其基本任务是根据语言的语法规则进行语法分析，若不存在语法错误则给出正确的语法结构并为语义分析和代码生成做准备。在描述程序语言的语法结构时，需借助于上下文无关文法，而文法是描述程序语言的依据。语法分析的方法通常分为两类，即自顶向下分析方法和自底向上分析方法。所谓自顶向下分析法，就是从文法的开始符号出发，根据文法的规则进行推导，最终推导出给定的句子来。自底向上的分析法则是从给定的输入串开始，根据文法规则逐步进行归约，直至归约到文法的开始符号为止。

3.1 文法和语言

文法是程序语言的生成系统，而自动机则是程序语言的识别系统：用文法可以精确地定义一个语言，并依据该文法构造出识别这个语言的自动机。因此，文法对程序语言和编译程序的构造具有重要意义，如程序语言的词法可用正规文法描述，语法可用上下文无关文法描述，而语义则要借助于上下文有关文法描述。

3.1.1 文法和语言的基本概念

1. 语言

通常我们用 Σ 表示字母表，字母表中的每个元素称为字符或符号。不同语言的字母表可能是不同的，程序语言的字母表通常是 ASCII 字符集。由字母表 Σ 中的字符所组成的有穷系列称为 Σ 上的字符串或字，字母表 Σ 上的所有字符串(包括空串)组成的集合用 Σ^*表示。那么，对字母表 Σ 来说，Σ^*上的任意一个子集都称为 Σ 上的一个语言，记为 $L(L\subset\Sigma^*)$，该语言的每一个字符串称为语言 L 的一个语句或句子。

例如，设 $\Sigma=\{a,b,c\}$，则 $L=\{\varepsilon,a,aa,ab,aaa,aab,aba,abb,\cdots\}$为 Σ 上的一个语言。如果 a 表示字母，b 表示数字，c 看做其它符号，我们可以将 L 看作是程序语言中的标识符集(当然，该标识集中的标识符允许以数字开头)，其中的每个标识符就是标识符集中的一个句子。

2. 文法

文法通常表示成四元组 $G[S]=(V_T, V_N, S, \xi)$，其中：

(1) V_T 为终结符号集，这是一个非空有限集，它的每个元素称为终结符号。

(2) V_N 为非终结符集，它也是一个非空有限集，其每个元素称为非终结符号，且有 $V_T\cap V_N=\Phi$。

(3) S 为一文法开始符，是一个特殊的非终结符号，即 $S\in V_N$。

(4) ξ是产生式的非空有限集，其中每个产生式(或称规则)是一序偶(α,β)，通常写作

$$\alpha \rightarrow \beta \text{ 或 } \alpha ::= \beta$$

读作“α是β”或“α定义为β”。在此，α为产生式的左部，而β为产生式的右部，α、β是由终结符和非终结符组成的符号串，$\alpha \in (V_T \cup V_N)^+$且至少有一个非终结符，而$\beta \in (V_T \cup V_N)^*$。

终结符号是指语言不可再分的基本符号，通常是一个语言的字母表；终结符代表了语法的最小元素，是一种个体记号。非终结符号也称语法变量，它代表语法实体或语法范畴；非终结符代表一个一定的语法概念，因此，一个非终结符是一个类、一个集合。例如，在程序语言中，可以把变量、常数、“+”、“*”等看作是终结符，而像“算术表达式”这个非终结符则代表着一定算术式组成的类，如 i*(i+i)、i+i+i 等；也即每个非终结符代表着由一些终结符和非终结符且满足一定规则的符号串组成的集合。

文法开始符号是一个特殊的非终结符，它代表文法所定义的语言中我们最终感兴趣的语法实体，即语言的目标，而其它语法实体只是构造语言目标的中间变量。如表达式文法的语言目标是表达式，而程序语言的目标通常为程序。

产生式(也称产生规则或规则)是定义语法实体的一种书写规则。一个语法实体的相关规则可能不止一个。例如，有：

$$P \rightarrow \alpha_1$$
$$P \rightarrow \alpha_2$$
$$\vdots$$
$$P \rightarrow \alpha_n$$

为书写方便，可将这些有相同左部的产生式合并为一个，即缩写成

$$P \rightarrow \alpha_1 \mid \alpha_2 \mid \cdots \mid \alpha_n$$

其中，每个$\alpha_i(i = 1,2,\cdots,n)$称为 P 的一个候选式，直竖“｜”读为“或”，它与“→”一样是用来描述文法的元语言符号(即不属于Σ的字符)。

例 3.1　试构造产生标识符的文法。

[解答]　首先，标识符是以字母开头的字母数字串，我们用 L 表示“字母”类非终结符，用 D 表示“数字”类非终结符，而用 T 表示“字母或数字”类非终结符，则有：

$$L \rightarrow a \mid b \mid \cdots \mid z$$
$$D \rightarrow 0 \mid 1 \mid \cdots \mid 9$$
$$T \rightarrow L \mid D$$

其次，如果用 S 表示“字母数字串”类，则 T 是一字母或数字，ST 也是字母数字串，即

$$S \rightarrow T \mid ST$$

其中，产生式 S→T｜ST 是一种左递归形式，由它可以产生一串 T。

最后，作为“标识符”的非终结符 I，它或者是一单个字母，或者为一字母后跟字母数字串，即

$$I \rightarrow L \mid LS$$

因此，产生标识符的文法 G[I]为：

$$G[I]=(\{a,b,\cdots,z,0,\cdots,9\},\{I,S,T,L,D\},I,\xi)$$

ξ: I→L｜LS
S→T｜ST
T→L｜D
L→a｜b｜…｜z
D→0｜1｜…｜9

例 3.2 写一文法，使其语言是奇数集合，但不允许出现以 0 打头的奇数。

[解答] 根据题意，我们可以将奇数划分为如图 3-1 所示的三个部分，即最高位允许出现 1～9，用非终结符 B 表示；中间部分可以出现任意多位数字 0～9，每一位用非终结符 D 表示；最低位只允许出现 1、3、5、7、9 等奇数，用 A 表示。

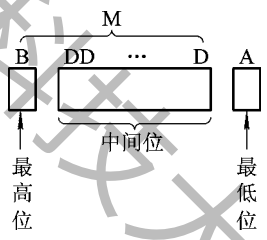

图 3-1 奇数划分示意

由于中间部分可出现任意位，所以另引入了一个非终结符 M，它包括最高位和中间位部分。假定开始符为 N，则可得到文法 G[N] 为

G[N]=({0,1,…,9},{N,A,M,B,D},N,ξ)

ξ: N→A｜MA //一位数字｜多位数字
M→B｜MD //仅两位数字(无中间位)｜多于两位数字
A→1｜3｜5｜7｜9
B→1｜2｜3｜4｜5｜6｜7｜8｜9
D→0｜B

3．文法产生的语言

设文法 $G[S]=(V_T,V_N,S,\xi)$ 且 α、$\beta\in(V_T\cup V_N)^*$，如果存在产生式 $A\to\delta(\delta\in(V_T\cup V_N)^*)$，则称 αAβ 可直接推出 αδβ，即

$$\alpha A\beta \Rightarrow \alpha\delta\beta$$

其中“⇒”表示直接推导出，是应用产生规则进行推导的记号。注意“⇒”与“→”不同，“→”是产生式中的定义记号。直接推导是指对文法符号串 αAβ 中的非终结符 A 用相应的产生式 A→δ 的右部 δ 来替换，从而得到 αδβ。我们给出推导的说明如下：

(1) 如果 α_1 可直接推出 α_2，α_2 可直接推出 α_3，…，α_{n-1} 可直接推出 α_n，即存在一个自 α_1 至 α_n 的推导序列：$\alpha_1\Rightarrow\alpha_2\Rightarrow\alpha_3\Rightarrow\cdots\Rightarrow\alpha_n(n>0)$，则我们称 α_1 可推导出 α_n，记为 $\alpha_1\overset{+}{\Rightarrow}\alpha_n$，它表示从 α_1 出发经过一步或若干步可推导出 α_n。

(2) 如果记 $\alpha_1\overset{0}{\Rightarrow}\alpha_1$，则 $\alpha_1\overset{*}{\Rightarrow}\alpha_n$ 表示从 α_1 出发，经过 0 步或若干步可推导出 α_n；也即 $\alpha_1\overset{*}{\Rightarrow}\alpha_n$ 意味着或者 $\alpha_1=\alpha_n$，或者 $\alpha_1\overset{+}{\Rightarrow}\alpha_n$。

例如，对下面的文法 G[E]：

$$E\to E+E\mid E*E\mid (E)\mid i \tag{3.1}$$

其中，唯一的非终结符 E 可以看成是代表一类算术表达式。我们可以从 E 出发进行一系列的推导，如表达式 i+i*i 的推导如下：

$$E\Rightarrow E+E\Rightarrow E+E*E\Rightarrow E+E*i\Rightarrow E+i*i\Rightarrow i+i*i$$

注意：在每一步推导过程中，只能对其中的一个非终结符用其对应的产生式右部的一

个候选式来替换。

假定 G[S]是一个文法，S 是它的开始符号，如果 $S \overset{*}{\Rightarrow} \alpha$，$\alpha \in (V_T \cup V_N)^*$，则称 α 是文法 G[S]的一个句型；如果 $\alpha \in V_T^*$，则称 α 是文法 G[S]的一个句子。仅含终结符的句型是一个句子。因此，句型和句子的定义如下：

(1) 句型：由文法的开始符 S 出发，经过 0 步或有限步推导出来的符号串 α(即 $S \overset{*}{\Rightarrow} \alpha$，$\alpha \in (V_T \cup V_N)^*$)。

(2) 句子：由文法的开始符 S 出发，经过 1 步或有限步推导出来的符号串 α 且该符号串 α 全部由终结符组成($S \overset{+}{\Rightarrow} \alpha$ 且 $\alpha \in V_T^*$)。

由定义可知，开始符 S 本身只能是文法的一个句型而不可能是一个句子。此外，上面推导出的 i+i*i 是文法 G[E]的一个句子(当然也是一个句型)，而 E+E、E+E*E、E+E*i 和 E+i*i 都是文法 G[E]的句型。

对于文法 G[S]，它所产生的句子的全体称为由文法 G[S]产生的语言，记为 L(G)，即

$$L(G) = \{\alpha \mid S \overset{+}{\Rightarrow} \alpha \text{ 且 } \alpha \in V_T^*\}$$

在此需要注意：① S 至少进行一次推导；② S 所推导出的 α 必须全部由终结符组成。

3.1.2　形式语言分类

语言学家 Noam Chomsky 于 1956 年首先建立了形式语言的描述，定义了四类文法及相应的形式语言，并分别与相应的识别系统相联系，它对程序语言的设计、编译方法、计算复杂性等方面都产生了重大影响。

1. 0 型文法与 0 型语言(对应图灵机)

如果文法 G[S]的每一个产生式具有下列形式：

$$\alpha \rightarrow \beta$$

其中，$\alpha \in (V_T \cup V_N)^* V_N (V_T \cup V_N)^*$，即至少含有一个非终结符；$\beta \in (V_T \cup V_N)^*$；则称文法 G[S]为 0 型文法或短语文法，记为 PSG。0 型文法相应的语言称为 0 型语言或称递归可枚举集，它的识别系统是图灵(Turing)机。

2. 1 型文法与 1 型语言(对应线性界限自动机，自然语言)

文法 G[S]的每一个产生式 $\alpha \rightarrow \beta$，均在 0 型文法的基础上增加了字符长度上满足 $|\alpha| \leqslant |\beta|$ 的限制，则称文法 G[S]为 1 型文法或上下文有关文法，记为 CSG。1 型文法相应的语言称为 1 型语言或上下文有关语言，它的识别系统是线性界限自动机。

1 型文法的另一种定义方法是文法 G[S]的每一个产生式具有下列形式：

$$\alpha A \delta \rightarrow \alpha \beta \delta$$

其中，α、$\delta \in (V_T \cup V_N)^*$，$A \in V_N$，$\beta \in (V_T \cup V_N)^+$。显然，它满足前述定义的长度限制，但它更明确地表达了上下文有关的特性，即 A 必须在 α、δ 的上下文环境中才能被 β 所替换。

3. 2 型文法与 2 型语言(对应下推自动机，程序设计语言)

文法 G[S]的每一个产生式具有下列形式：

$$A \rightarrow \alpha$$

其中，$A \in V_N$，$\alpha \in (V_T \cup V_N)^*$，则称文法 G[S]为 2 型文法或上下文无关文法，记为 CFG。2 型文法相应的语言称为 2 型语言或上下文无关语言，它的识别系统是下推自动机。

4. 3 型文法与 3 型语言(对应有限自动机)

文法 G[S]的每个产生式具有下列形式：

$$A \to a \text{ 或 } A \to aB$$

其中，$A、B \in V_N$，$a \in V_T^*$，则文法 G[S]称为 3 型文法、正规文法或右线性文法，记为 RG。3 型文法相应的语言为 3 型语言或正规语言，它的识别系统是有限自动机。3 型文法还可以呈左线性形式：

$$A \to a \text{ 或 } A \to Ba$$

5. 四类文法的关系与区别

由四类文法的定义可知，从 0 型文法到 3 型文法逐渐增加限制。1～3 型文法都属于 0 型文法，2、3 型文法不一定属于 1 型文法(如果存在形如 A→ε 的产生式，则不属于 1 型文法)，3 型文法属于 2 型文法。四类文法的区别如下：

(1) 1 型文法中不允许有形如“A→ε”的产生式存在，而 2、3 型文法则允许形如“A→ε”的产生式存在；

(2) 0、1 型文法的产生式左部存在含有终结符号的符号串或两个以上的非终结符，而 2 型和 3 型文法的产生式左部只允许是单个的非终结符号。

例 3.3 试判断下列产生式集所对应的文法和产生的语言：

(1) ① S→ACaB ② Ca→aaC ③ CB→DB ④ CB→E ⑤ aD→Da ⑥ AD→AC ⑦ aE→Ea ⑧ AE→ε

(2) ① S→aSBC ② S→aBC ③ CB→DB ④ DB→DC ⑤ DC→BC ⑥ aB→ab ⑦ bB→bb ⑧ bC→bc ⑨ cC→cc

(3) ① S→Ac ② S→Sc ③ A→ab ④ A→aAb

(4) ① S→aS ② S→aA ③ A→bA ④ A→bB ⑤ B→cB ⑥ B→c

[解答] 由四类文法的定义与区别可知，(1)～(4)分别为 0～3 型文法。

(1) 该 0 型文法产生的 0 型语言为 $L_0(G)=\{a^{2n} \mid n>0\}$。例如：当 n=2 时，句子 $a^{2\times 2}$=aaaa，是通过下列推导得到的：

$$S \overset{(1)}{\Rightarrow} ACaB \overset{(2)}{\Rightarrow} AaaCB \overset{(3)}{\Rightarrow} AaaDB \overset{(5)}{\Rightarrow} AaDaB \overset{(5)}{\Rightarrow} ADaaB \overset{(6)}{\Rightarrow} ACaaB \overset{(2)}{\Rightarrow} AaaCaB \overset{(2)}{\Rightarrow} AaaaaCB$$
$$\overset{(4)}{\Rightarrow} AaaaaE \overset{(7)}{\Rightarrow} AaaaEa \overset{(7)}{\Rightarrow} AaaEaa \overset{(7)}{\Rightarrow} AaEaaa \overset{(7)}{\Rightarrow} AEaaaa \overset{(8)}{\Rightarrow} aaaa$$

(2) 该 1 型文法产生的 1 型语言为 $L_1(G)=\{a^nb^nc^n \mid n\geq 1\}$。例如，当 n=2 时，句子 $a^2b^2c^2$=aabbcc 是通过下列推导得到的：

$$S \overset{(1)}{\Rightarrow} aSBC \overset{(2)}{\Rightarrow} aaBCBC \overset{(6)}{\Rightarrow} aabCBC \overset{(3)}{\Rightarrow} aabDBC \overset{(4)}{\Rightarrow} aabDCC \overset{(5)}{\Rightarrow} aabBCC \overset{(7)}{\Rightarrow} aabbCC \overset{(8)}{\Rightarrow}$$
$$aabbcC \overset{(9)}{\Rightarrow} aabbcc$$

(3) 该 2 型文法产生的 2 型语言为 $L_2(G)=\{a^nb^nc^m \mid m、n\geq 1\}$。例如当 n=2、m=3 时，

句子 $a^2b^2c^3$=aabbccc 是通过下列推导得到的：

$$S \overset{(2)}{\Rightarrow} Sc \overset{(2)}{\Rightarrow} Scc \overset{(1)}{\Rightarrow} Accc \overset{(4)}{\Rightarrow} aAbccc \overset{(3)}{\Rightarrow} aabbccc$$

(4) 该 3 型文法产生的 3 型语言为 $L_3(G)=\{a^mb^nc^k \mid m、n、k\geqslant 1\}$。例如当 m=2、n=3、k=4 时，句子 $a^2b^3c^4$=aabbbcccc 是通过下列推导得到的：

$$S \overset{(1)}{\Rightarrow} aS \overset{(2)}{\Rightarrow} aaA \overset{(3)}{\Rightarrow} aabA \overset{(3)}{\Rightarrow} aabbA \overset{(4)}{\Rightarrow} aabbbB \overset{(5)}{\Rightarrow} aabbbcB \overset{(5)}{\Rightarrow} aabbbccB \overset{(5)}{\Rightarrow}$$
$$aabbbcccB \overset{(6)}{\Rightarrow} aabbbcccc$$

由例 3.3 可知：$\{a^nb^nc^n \mid n\geqslant 1\} \subset \{a^nb^nc^m \mid m、n\geqslant 1\} \subset \{a^mb^nc^k \mid m、n、k\geqslant 1\}$，这说明对文法规则定义形式的限制虽然加强了，但相应的语言反而更大了。因此，不能主观认定文法限制越大则语言越小，也即下述结论是不成立的：

3 型语言⊂2 型语言⊂1 型语言⊂0 型语言

在编译方法中，通常用 3 型文法来描述高级程序语言的词法部分，然后用有限自动机 FA 来识别高级语言的单词；利用 2 型文法来描述高级语言的语法部分，然后用下推自动机 PDA 来识别高级语言的各种语法成分。

例 3.4　给出字母表 Σ={a,b}上的同时只有奇数个 a 和奇数个 b 的所有字符串集合的正规文法。

[解答]　为了构造字母表 Σ={a,b}上同时只有奇数个 a 和奇数个 b 的所有字符串的正规表达式，参考例 2.14 的 DFA M，我们画出如图 3-2 所示的 DFA M，即由开始符 S 出发，经过奇数个 a 到达状态 A，或经过奇数个 b 到达状态 B。再由状态 A 出发，经过奇数个 b 到达状态 C(终态)；同样，由状态 B 出发，经过奇数个 a 到达终态 C。

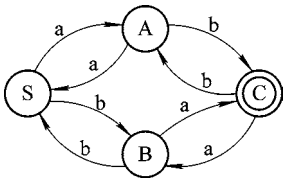

图 3-2　例 3.4 的 DFA

由图 3-2 可直接得到正规文法 G[S]如下：

G[S]：S→aA｜bB
A→aS｜bC｜b
B→bS｜aC｜a
C→bA｜aB｜ε

3.1.3　正规表达式与上下文无关文法

1. 正规表达式到上下文无关文法的转换

正规表达式所描述的语言结构均可以用上下文无关文法描述，反之则不一定。由正规表达式构造上下文无关文法的一种方法如下：

(1) 构造正规表达式的 NFA。

(2) 若 0 为初始状态，则 A_0 为开始符号。

(3) 如果存在映射关系 f(i,a)=j，则定义产生式 $A_i \rightarrow aA_j$。

(4) 如果存在映射关系 f(i,ε)=j，则定义产生式 $A_i \rightarrow A_j$。

(5) 若 i 为终态，则定义产生式 $A_i \rightarrow \varepsilon$。

例 3.5　用上下文无关文法描述正规表达式 $(a \mid b)^*abb$。

[解答]　首先构造识别正规表达式 $(a \mid b)^*abb$ 的 NFA M 如图 3-3 所示。

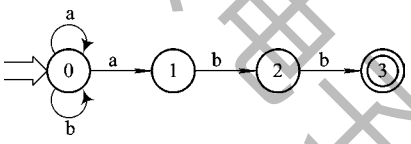

图 3-3 例 3.5 的 NFA M

由构造上下文无关文法的方法得到上下文无关文法 G[A_0]如下：

G[A_0]：$A_0 \rightarrow aA_0 \mid bA_0 \mid aA_1$

$A_1 \rightarrow bA_2$

$A_2 \rightarrow bA_3$

$A_3 \rightarrow \varepsilon$

事实上，由正规表达式构造上下文无关文法还可以采用另一种方法，即通过分析正规表达式的特性凭经验直接构造。如可把$(a \mid b)^*abb$ 看作由$(a \mid b)^*$和 abb 两部分组成，第一部分是由 0 个或若干个 a 和 b 组成的字符串，而第二部分则仅由 abb 字符串组成，由此得到上下文无关文法 G[A]如下：

G[A]：$A \rightarrow HT$

$H \rightarrow aH \mid bH \mid \varepsilon$

$T \rightarrow abb$

2．正规表达式与上下文无关文法描述的对象

上下文无关文法既可以描述程序语言的语法，又可以描述程序语言的词法，但基于下述原因，应采用正规表达式描述词法：

(1) 词法规则简单，采用正规表达式已足以描述。

(2) 正规表达式的表示比上下文无关文法更加简洁、直观和易于理解。

(3) 有限自动机的构造比下推自动机简单且分析效率高。

贯穿词法分析和语法分析始终如一的思想是：语言的描述和语言的识别是表示一个语言的两个不同的侧面，二者缺一不可。用正规表达式和上下文无关文法描述语言时的识别方法(即自动机)不同。通常，正规表达式适合于描述线性结构，如标识符、关键字和注释等；而上下文无关文法则适合于描述具有嵌套(层次)性质的非线性结构，如不同结构的语句 if-else、while 等。

3.2 推导与语法树

在此后的讨论中，我们用大写字母 A、B、S、E 等表示非终结符，用小写字母 a、b、i、j 等表示终结符，并用希腊字母等表示文法符号串，即 α、β、δ、γ 等均属于$(V_T \cup V_N)^*$。

3.2.1 推导与短语

1．规范推导

在 3.1.1 节中，所给句子 i+i*i 推导序列中的每一步推导都是对句型中的最右非终结符

用相应产生式的右部进行替换，这样的推导称为最右推导。如果每一步推导都是对句型中的最左非终结符用相应产生式的右部进行替换，则称为最左推导。例如句子 i+i*i 按文法(3.1)的最左推导是

$$E\Rightarrow E+E\Rightarrow i+E\Rightarrow i+E*E\Rightarrow i+i*E\Rightarrow i+i*i$$

从一个句型到另一个句型的推导过程并不是唯一的，为了对句子的结构进行确定性分析，我们往往只考虑最左推导或最右推导，并且称最右推导为规范推导。规范推导的逆过程便是规范归约(最左归约)。

2．短语

设 αβδ 是文法 G[S] 的一个句型，如果有：

$$S\overset{*}{\Rightarrow}\alpha A\delta \text{ 且 } A\overset{+}{\Rightarrow}\beta$$

则称 β 是句型 αβδ 关于非终结符 A 的一个短语，或称 β 是 αβδ 的一个短语。特别是有 A→β 产生式时，β 为句型 αβδ 的一个直接短语或简单短语。

短语的两个条件缺一不可。仅有 $A\overset{+}{\Rightarrow}\beta$ 未必意味着 β 就是句型 αβδ 的一个短语，还需要有 $S\overset{*}{\Rightarrow}\alpha A\delta$ 这一前提条件加以限制，即由开始符 S 出发能够推导出 αAδ 句型，而句型 αAδ 中的非终结符 A 又能够推导出短语 β；这时 β 既是非终结符 A 的一个短语，也是句型 αβδ 的一个短语。也即，短语属于句型的组成部分。

例如，对文法(3.1)，因为 $E\overset{*}{\Rightarrow}E+E*E\Rightarrow E+E*i$，所以 i 是 E+E*i 的一个短语，并且是直接短语；而由 $E\overset{*}{\Rightarrow}E+E\overset{+}{\Rightarrow}E+E*i$ 知 E*i 是 E+E*i 的一个短语，但非直接短语；而 E+E*i 是 E+E*i 自身的短语。

3．句柄

一个句型的最左直接短语称为该句型的句柄。注意，一个句型的直接短语可能不只一个，但最左直接短语则是唯一的。如对 $S\overset{*}{\Rightarrow}\alpha A\delta\Rightarrow\alpha\beta\delta$ 来说，将句型 αβδ 中的句柄 β 用产生式的左部符号代替便得到新句型 αAδ，这是一次规范归约，恰好与规范推导相反。

4．素短语

含有终结符的短语，如果它不存在也具有同样性质的真子串(即素短语中不得含有其它素短语)，则该短语为素短语。例如，在 $E\overset{+}{\Rightarrow}E+E*i$ 中，i、E*i 和 E+E*i 是句型 E+E*i 的三个短语；其中，i 为素短语；E*i 虽为短语且含有终结符，但它的真子串 i 是素短语，故 E*i 不是素短语；同样 E+E*i 也不是素短语。

3.2.2　语法树与二义性

1．语法树

对程序语言来说，有两个问题需要解决：其一是判别程序在语法上是否正确；其二是句子的识别或分析。在编译方法中，为了便于识别或分析句子而引入了语法树这一重要的辅助工具。语法树以图示化的形式把句子分解成各个组成部分来描述或分析句子的语法结构，这种图示化的表示与所定义的文法规则完全一致，但更为直观和完整。

对文法 $G[S]=(V_T, V_N, S, \xi)$，满足下列条件的树称为 G[S] 的语法树：

(1) 每个结点用 G[S]的一个终结符或非终结符标记。

(2) 根结点用文法开始符 S 标记。

(3) 内部结点(指非树叶结点，内部结点也包括根结点)一定是非终结符，如果某内部结点 A 有 n 个分支，它的所有子结点从左至右依次标记为 x_1、x_2、…、x_n，则 $A \rightarrow x_1x_2\cdots x_n$ 一定是文法 G[S]的一条产生式。

(4) 如果某结点标记为 ε，则它必为叶结点且是其父结点的唯一子结点。

相应于一个句型的语法树是以文法的开始符 S 作为根结点的，并随着推导逐步展开；当某个非终结符被它对应的产生式右部的某个候选式所替换时，这个非终结符所对应的结点就产生出下一代新结点，即候选式中从左至右的每一个符号都依次顺序对应一个新结点，且每个新结点与其父结点之间都有一条连线(树枝)。在一棵语法树生长过程中的任何时刻，所有那些没有后代的树叶结点自左至右排列起来就是一个句型。

例如，与文法(3.1)的句子 i+i*i 相应的语法树如图 3-4 所示。

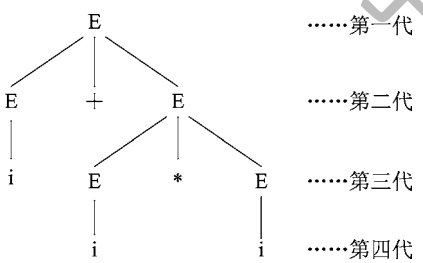

图 3-4 句子 i+i*i 的语法树

在构造语法树时可以发现，一个句型的最左推导及最右推导只决定先生长左子树还是先生长右子树，句型推导结束时相应的语法树也随之完成，这时已不能看出是先生长左子树还是先生长右子树，所呈现的仅仅是已经长成的这个句子或句型的语法树。这与使用文法规则进行推导是有差异的，即使用文法规则的推导过程是有先后之分的。因此，一棵语法树表示了一个句型种种可能的(但未必是所有的——见下面文法的二义性)不同推导过程，包括最左(最右)推导。如果我们坚持使用最左(或最右)推导，那么一棵语法树就完全等价于一个最左(或最右)推导，这种等价性也包括语法树的每一步生长和推导的每一步展开的这种完全一致性。

2．子树和短语

语法树的某个内部结点(即非树叶结点)连同它的所有后代组成了一棵子树，子树的根结点即为此内部结点。只含有单层分枝的子树称为简单子树。

子树与短语的关系十分密切，根据子树的概念，句型的短语、直接短语、句柄和素短语的直观解释如下：

(1) 短语：子树的末端结点(即树叶)组成的符号串是相对于子树根的短语。

(2) 直接短语：简单子树的末端结点组成的符号串是相对于简单子树根的直接短语。

(3) 句柄：最左简单子树的末端结点组成的符号串为句柄。

(4) 素短语：子树的末端结点组成的符号串含终结符，且在该子树中不再有含有终结符的更小子树。

短语和直接短语的进一步解释是：

(1) 短语：由内部结点向下生长的全部树叶自左至右的排列。每个内部结点都有一个短语，也可能有些内部结点的短语是同一个。

(2) 直接短语：内部结点直接一步生长出来的结点全部都由树叶组成(即以该内部结点为根的子树是简单子树)，该全部树叶自左至右的排列即为直接短语。

因此，直接短语是在短语基础上增加了只能向下生长一步且向下生长一步所产生的结点全部都由树叶组成这一限制；而句柄则是在直接短语的基础上增加了“最左”这一条件限制。

对素短语来说，首先要求素短语本身必须是一个短语，并在短语基础上又增加了两条限制(求素短语的另一种方法见本章 3.4.2 节)：① 构成素短语的全部树叶中至少含有一个终结符；② 该素短语的全部树叶中不得含有其它素短语。

显然，从语法树出发寻找短语、直接短语、句柄和素短语要直观得多。此外，要注意的是子树末端结点组成的符号串是指由该子树根开始向下生长的所有末端结点(即树叶)自左至右的排列，该子树的部分末端结点并不是该子树的短语。

对图 3-5 所示的关于句型 E+E*i 的语法树来说，它有 3 个内部结点(对应 3 棵子树)，即有 3 个短语，分别为 i、E*i 和 E+E*i；直接短语、句柄和素短语均为 i；而 E+E 由于在句型 E+E*i 的限制下只是树根 E 的部分末端结点，因而不是短语。但是，在图 3-6 中给出的 E+E+E*i 的句型下，E+E 却是子结点 E 的一个直接短语。

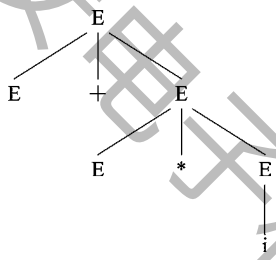

图 3-5　E+E*i 的语法树

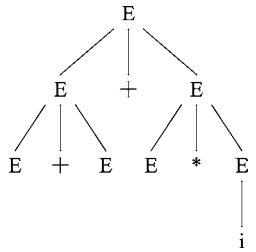

图 3-6　E+E+E*i 的语法树

例如，对本节下面将要介绍的无二义性算术表达式文法：

G[E]：E→E+T｜T
　　T→T*F｜F
　　F→(E)｜i

句型(T+T)*i 对应的语法树如图 3-7 所示。

对图 3-7 所示关于句型(T+T)*i 的语法树，它有 7 个内部结点(对应 7 棵子树)，相应就有 7 个短语；但根结点(内部结点)E 和第二层内部结点 T 向下生长出来的是同一树叶序列，故两者短语相同，都是(T+T)*i。第三层的内部结点 T 和第四层的内部结点 F 向下生长的也是同一树叶序列，即两者短语也相同，均为(T+T)。其余内部结点对应的短语各不相同，因此该句型(语法树)共有 5 个短语：T、T+T、(T+T)、i、(T+T)*i。

图 3-7 对应的直接短语有两个：一个是最下面的内部结点 E 直接一步生长成的树叶 T，另一个是第三层的内部结点 F 直接一步生长成的树叶 i。由于 T 在 i 的左边，故 T 为句柄。

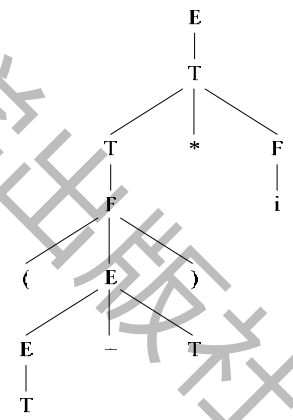

图 3-7　(T+T)*i 对应的语法树

素短语首先必须是一个短语，其次要至少含有一个终结符并且该素短语中不再含有其它更小的素短语。因此，直接短语 T 因其是非终结符(即该短语不含终结符)而不是素短语；直接短语 i 因其本身是终结符且不含其它更小的素短语，故为素短语；短语 T+T 满足至少含有一个终结符的条件且 T+T 中不含其它更小的素短语，故为素短语；短语(T+T)虽然满

足至少含有一个终结符的条件，但因其含有更小的素短语 T+T 而不是素短语；短语(T+T)*i 也因同样的原因不是素短语。

现在，可以将句型和短语的关系归纳如下：由根结点(即文法开始符)向下生长的任何时候，其生长出的全部树叶自左至右的排列就是一个句型。语法树的每个内部结点生长出的全部树叶自左至右的排列就是短语。由于根结点也是内部结点，所以根结点生长出的全部树叶自左至右的排列既是句型，又是短语。

3．文法的二义性

文法 G[S]的一个句子如果能找到两种不同的最左推导(或最右推导)，或者存在两棵不同的语法树，则称这个句子是二义性的。一个文法如果包含二义性的句子，则这个文法是二义文法，否则是无二义文法。

例如，对文法(3.1)，句子 i+i*i 存在着两种最左推导或最右推导，所形成的两棵不同的语法树如图 3-8 所示。

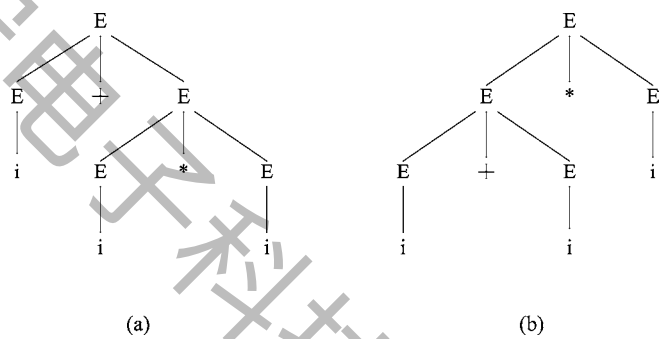

图 3-8 句子 i+i*i 的两棵不同的语法树

再如，条件语句的文法 G[S]为

G[S]：S→if b S
S→if b S else S
S→a

其中，V_N= {S}，V_T= {a,b,if,else}，则句子 if b if b a else a 所对应的两棵不同语法树见图 3-9。因此，文法 G[S]是二义性文法。

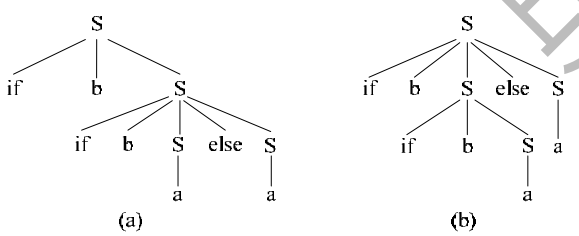

图 3-9 句子 if b if b a else a 的两棵不同语法树

显然，二义性文法将给编译程序的执行带来麻烦。对于二义性文法的句子，当编译程序对它的结构进行语法分析时，就会产生两种甚至多种不同的解释；由于语法结构的这种不确定性，因而必将导致语义处理上的不确定性。

4．文法二义性的消除

一个文法是二义性的，并不说明该文法所描述的语言也是二义性的。也即，对于一个二义性文法 G[S]，如果能找到一个非二义性文法 G'[S]，使得 L(G')=L(G)，则该二义性文法的二义性是可以消除的。如果找不到这样的 G'[S]，则二义性文法描述的语言为先天二义性的。

文法二义性消除的方法如下：

(1) 不改变文法中原有的语法规则，仅加进一些语法的非形式规定。如对文法(3.1)，不改变已有的四条规则(即四个产生式)，仅加进运算符的优先顺序和结合规则，即*优先于+，且*、+都服从左结合。这样，对文法(3.1)中的句子 i+i*i 就只有如图 3-8(a)所示的唯一一棵语法树。

(2) 构造一个等价的无二义性文法，即把排除二义性的规则合并到原有文法中，改写原有的文法。

方法(2)是通过添加新的非终结符来消除文法中的二义性，以达到将原文法改造成一个等价的无二义性文法的目的。

那么，如何才能将文法(3.1)改写为无二义性的文法呢？在构造算术表达式文法时可以按运算符的优先级将产生式分为三个层次：“+”、“-”类为一层，“*”、“/”类为一层，而括号“()”和运算对象“i”为另一层；这三层的优先级依次递增。由此，我们在文法(3.1)原有的非终结符 E 基础上再添加两个非终结符 T 和 F，即以 E、T、F 来分别代表三个层次的划分。并且，我们可以由后面将要介绍的归约看出：离根结点越远的短语先被归约，离根结点越近的短语后被归约。体现在文法上就是离开始符(对应根结点)越远的产生式其优先级越高，离开始符越近的产生式其优先级越低，开始符所在的产生式其优先级最低。据此，我们可以将文法(3.1)改写为无二义性的文法 G'[E] 如下：

G'[E]：E→E+T｜T
T→T*F｜F
F→(E)｜i

此时，句子 i+i*i 就只有如图 3-10 所示的唯一一棵语法树。

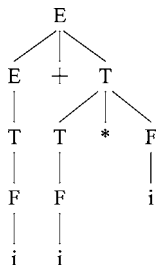

图 3-10　句子 i+i*i 的语法树

例 3.6　试将如下的二义性文法 G[S] 的二义性消除：

G[S]：S→if b S｜if b S else S｜a

[解答]　消除 G[S] 的二义性可采用如下两种方法：

(1) 不改变已有规则，仅加进一项非形式的语法规定：else 与离它最近的 if 匹配(即最近匹配原则)，这样，文法 G[S] 的句子 if b if b a else a 只对应唯一的一棵语法树(见图 3-11)。

(2) 改写文法 G[S] 为 G'[S]：$S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow$ if b S_1 else $S_1 \mid$ a

$S_2 \rightarrow$ if b S $\mid$ if b S_1 else S_2

这是因为引起二义性的原因是 if-else 语句的 if 后可以是任意 if 型语句，所以改写文法时规定 if 和 else 之间只能是 if-else 语句或其它语句。这样，改写后文法 G'[S] 的句子 if b if b a else a 只对应唯一的一棵语法树(如图 3-12 所示)。

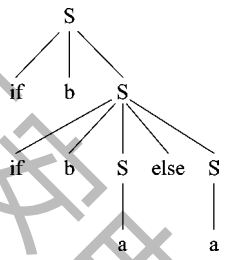

图 3-11 复合 if 语句的语法树

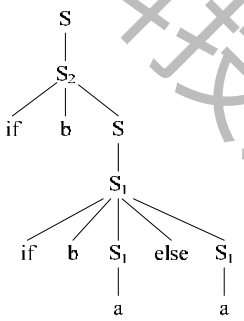

图 3-12 G'[S] 的复合 if 语句的语法树

我们总希望一个文法是无二义性的，这样，句子的分析可以按唯一确定的方式进行。但是，文法的二义性是不可判定的，即不存在一种算法，能够在有限步内判定一个文法是否为二义性的。有时候，二义性文法也可带来一定的好处，如语法分析中二义性文法的应用。

3.3 自顶向下的语法分析

自顶向下分析就是从文法的开始符出发并寻找出这样一个推导序列：推导出的句子恰为输入符号串；或者说，能否从根结点出发向下生长出一棵语法树，其叶结点组成的句子恰为输入符号串。显然，语法树的每一步生长(每一步推导)都以能否与输入符号串匹配为准，如果最终句子得到识别，则证明输入符号串为该文法的一个句子；否则，输入符号串不是该文法的句子。

3.3.1 递归下降分析法

递归下降分析法是一种自顶向下的分析方法，文法的每个非终结符对应一个递归过程(函数)。分析过程就是从文法开始符出发执行一组递归过程(函数)，这样向下推导直到推出句子；或者说从根结点出发，自顶向下为输入串寻找一个最左匹配序列，建立一棵语法树。

1. 自顶向下分析存在的不确定性

假定文法 G[S] 为

G[S]：$S \rightarrow xAy$

$A \rightarrow ab \mid a$

若输入串为 xay，则其分析过程如下：

(1) 首先建立根结点 S。

(2) 文法关于 S 的产生式只有一个，也即由 S 生长的语法树如图 3-13(a)所示，它的第一个终结符 x 与输入串待分析的字符 x 匹配。此时，下一个待分析的字符为 a，期待着与语法树中在 x 右侧且与 x 相邻的叶结点 A 匹配。

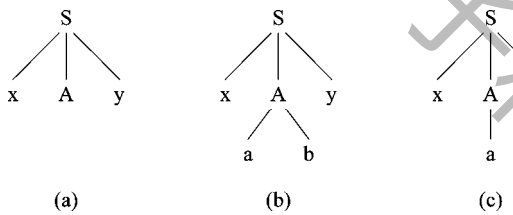

图 3-13　试探分析对应的语法树

(3) 非终结符 A 有两个候选式，先选用第一个候选式生长的语法树(如图 3-13(b)所示)；这时语法树的第二个叶结点 a 恰与待分析的字符 a 匹配。

(4) 输入串中下一个待分析的字符为 y，它期待与第三个叶结点 b 匹配，但匹配时发现这两个字符是不同的，即匹配失败，这是因为生成 A 的子树时所选用的是其第一个候选式，即(3)中对字符 a 的匹配是虚假匹配。

(5) 因不匹配而将 A 所生成的这棵子树注销，即把匹配指针回退(回溯)到输入串的第二个字符 a，也就是恢复与 A 匹配时的现场，即(3)之前。

(6) 此时 A 选用第二个候选式并生长语法树如图 3-13(c)所示，这时第二个叶结点 a 与输入串的第二个字符 a 匹配。

(7) 此时输入串的下一个待分析字符指向 y，而语法树的下一个未匹配的叶结点也为 y，两者恰好匹配。因此，图 3-13(c)所示的语法树即为输入串 xay 的语法树。

显然，这种自顶向下的分析是一个不断试探的过程；也即，在分析过程中，如果出现多个产生式(即候选式)可供选择，则逐一试探每一候选式进行匹配，每当一次试探失败，就选取下一候选式再进行试探；此时，必须回溯到这一次试探的初始现场，包括注销已生长的子树及将匹配指针调回到失败前的状态。这种带回溯的自顶向下分析方法实际上是一种穷举的试探方法，其分析效率极低，在实用的编译程序中很少使用。

2．确定的自顶向下分析

为了实现确定的(即无回溯的)自顶向下分析，则要求文法满足下述两个条件：

(1) 文法不含左递归，即不存在这样的非终结符号 A：有 A→A…存在或者有 $A \overset{+}{\Rightarrow} A\alpha$；

(2) 无回溯，对文法的任一非终结符号，当其产生式右部有多个候选式可供选择时，各候选式所推出的终结符号串的首字符集合要两两不相交。

左递归是程序语言的语法规则中并不少见的形式，例如前述消除了二义性的算术表达式文法的一个规则：

E→E+T　　//简单表达式→简单表达式+项

如果对该左递归文法采用自顶向下分析法，即首先以“E+T”中的 E 为目标对“E+T”进行试探，进而又以其中的 E 为目标再对选择“E+T”进行试探；也即 E+T⇒E+T+T⇒E+T+T+T⇒…，这种左递归的文法使自顶向下分析工作陷入无限循环。也就是说，当试图用 E 去匹配输入符号串时会发现：在没有吃进任何输入符号的情况下，又得重新要求 E 去进行新的匹配。因此，使用自顶向下分析法首先要消除文法的左递归性。

对于回溯，从上述不确定的自顶向下分析示例可知，由于回溯的存在，可能在已经做了大量的语法分析工作之后，才发现走了一大段错路而必须回头，要把已经做的一大堆语义工作(指中间代码产生工作和各种表格的簿记工作)推倒重来。回溯使得自顶向下语法分析只具有理论意义而无实际使用的价值。因此，要使自顶向下语法分析具有实用性，就必须要消除回溯。

3. 消除左递归

直接消除借助于产生式中的左递归比较容易，其方法是引入一个新的非终结符，把含有左递归的产生式改为右递归。

设关于非终结符 A 的直接左递归的产生式形如

$$A \rightarrow A\alpha \mid \beta$$

其中，α、β 是任意的符号串且 β 不以 A 开头。该产生式所能推导的句子如下：

$$A \Rightarrow \beta$$
$$A \Rightarrow A\alpha \Rightarrow \beta\alpha$$
$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow \beta\alpha\alpha$$
$$A \Rightarrow A\alpha \Rightarrow A\alpha\alpha \Rightarrow A\alpha\alpha\alpha \Rightarrow \beta\alpha\alpha\alpha$$
$$\cdots$$

我们再看下面不含 A 的直接左递归的产生式

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

这两个产生式所能推导的句子如下：

$$A \Rightarrow \beta A' \Rightarrow \beta$$
$$A \Rightarrow \beta A' \Rightarrow \beta\alpha A' \Rightarrow \beta\alpha$$
$$A \Rightarrow \beta A' \Rightarrow \beta\alpha A' \Rightarrow \beta\alpha\alpha A' \Rightarrow \beta\alpha\alpha$$
$$A \Rightarrow \beta A' \Rightarrow \beta\alpha A' \Rightarrow \beta\alpha\alpha A' \Rightarrow \beta\alpha\alpha\alpha A' \Rightarrow \beta\alpha\alpha\alpha$$
$$\cdots$$

推导结果与产生式 $A \rightarrow A\alpha \mid \beta$ 的推导结果相同(实际上都能描述正规表达式 $\beta\alpha^*$)。因此，可将产生式 $A \rightarrow A\alpha \mid \beta$ 改写为

$$\begin{cases} A \rightarrow \beta A' \\ A' \rightarrow \alpha A' \mid \varepsilon \end{cases}$$

其中，A'为新引入的非终结符。另外，ε 不可缺少，否则上面的推导过程将无法完成。

例如，含有直接左递归的表达式文法 G[E] 为

$$G[E]: \quad E \rightarrow E+T \mid T$$
$$T \rightarrow T*F \mid F$$
$$F \rightarrow (E) \mid i$$

经过消去直接左递归后得到文法 G'[E] 为

$$G'[E]: \quad E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$

$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid i \qquad (3.2)$$

将产生式 $A \rightarrow A\alpha \mid \beta$ 中的 α 和 β 拓广为多项时，则文法中关于 A 的产生式为

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

其中，每个 α 都不等于 ε 且每个 β 都不以 A 开头，则消除 A 的直接左递归性就是将其改写为

$$\begin{cases} A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A' \\ A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \varepsilon \end{cases}$$

例如，对产生式 E→E+T｜E-T｜T，消除直接左递归后为

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

注意：也有些文法是含有间接左递归的，如下述文法 G[S]：

$$G[S]: S \rightarrow Qc \mid c$$
$$Q \rightarrow Rb \mid b$$
$$R \rightarrow Sa \mid a \qquad (3.3)$$

该文法虽不具有直接左递归，但 S、Q、R 都是左递归的，有

$$S \Rightarrow Qc \Rightarrow Rbc \Rightarrow Sabc$$

如何消除一个文法的一切左递归呢？如果一个文法不含回路(形如 $A \overset{+}{\Rightarrow} A$ 的推导)，且产生式的右部也不含 ε 的候选式，那么，下述算法将消除文法的左递归：

(1) 将文法 G[S] 的所有非终结符按一给定的顺序排列：A_1、A_2、…、A_n；

(2) 执行下述循环语句将间接左递归改为直接左递归：

```
for( i=1;i<=n;i++)
    for(j=1;j<=i−1;j++)
    { 把一个形如：
```

$$\begin{cases} A_i \rightarrow A_j\gamma \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \\ A_j \rightarrow \delta_1 \mid \delta_2 \mid \cdots \mid \delta_k \end{cases}$$

```
      的产生式改写为
```

$$A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \cdots \mid \delta_k\gamma \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n;$$

```
      按消除直接左递归的方法消除 Ai 的直接左递归；
    }
```

(3) 化简由(2)所得的文法，即去掉那些从开始符号 S 出发，在推导中无法出现的非终结符的产生式(去掉多余产生式)。

注意：消除左递归之前的文法不允许有 ε 产生式，否则无法得到等效的无左递归文法。因此，如果原文法中有 ε 的产生式，则需将文法改写为无 ε 的产生式的文法。此外，此算法并未对非终结符的排列顺序加以规定，不同的排列可能得到不同的结果，但彼此是等价的。

例如，将文法(3.3)的非终结符排序为 R、Q、S。对 R 来说，它不存在直接左递归；把 R 代入到 Q 的有关候选式后得到改变的 Q 产生式为

$$Q\rightarrow Sab \mid ab \mid b$$

现在的 Q 同样不含直接左递归，再把它代入到 S 的有关候选式后得到改变的 S 产生式为

$$S\rightarrow Sabc \mid abc \mid bc \mid c$$

经过消除了 S 的直接左递归后，即得到了整个文法 G'[S]为

$$\begin{aligned} G'[S]:\ & S\rightarrow abcS' \mid bcS' \mid cS' \\ & S'\rightarrow abcS' \mid \varepsilon \\ & Q\rightarrow Sab \mid ab \mid b \\ & R\rightarrow Sa \mid a \end{aligned}$$

显然，关于 Q 和 R 的产生式已为多余，因此化简后的最终文法 G"[S]为

$$\begin{aligned} G''[S]:\ & S\rightarrow abcS' \mid bcS' \mid cS' \\ & S'\rightarrow abcS' \mid \varepsilon \end{aligned} \tag{3.4}$$

实际上，我们也可以用数学中的分配律来消除文法中的左递归。对文法(3.3)，首先将 R 的产生式代入到 Q 的产生式中(注：“(”和“)”为元语言符号)

$$Q\rightarrow (Sa \mid a)b \mid b$$

并按分配律展开得到

$$Q\rightarrow Sab \mid ab \mid b$$

再将改变后 Q 的产生式代入到 S 的产生式中

$$S\rightarrow (Sab \mid ab \mid b)c \mid c$$

并按分配律展开得到

$$S\rightarrow Sabc \mid abc \mid bc \mid c$$

在消除 S 的直接左递归后同样得到文法(3.4)。

4. 消除回溯

回溯发生的原因在于候选式存在公共的左因子，如产生式 A 如下：

$$A\rightarrow \alpha\beta_1 \mid \alpha\beta_2$$

此时，如果输入串待分析的字符串前缀为 α，则选用哪个候选式以寻求与输入串匹配就难以确定。倘若候选式不含公共左因子，则推导出的首字符能与输入串匹配的那个候选式便是唯一的匹配。在文法 G[S]中的每个非终结符相应的产生式右部其候选式均不含公共左因子的情况下，语法分析的匹配过程都是唯一匹配，无需试探；这时若匹配失败，则意味着输入串不是句子。

一般情况下，设文法中关于 A 的产生式为

$$A\rightarrow \delta\beta_1 \mid \delta\beta_2 \mid \cdots \mid \delta\beta_i \mid \beta_{i+1} \mid \cdots \mid \beta_j \tag{3.5}$$

那么，可以把这些产生式改写为

$$\begin{cases} A\rightarrow \delta A' \mid \beta_{i+1} \mid \cdots \mid \beta_j \\ A'\rightarrow \beta_1 \mid \cdots \mid \beta_i \end{cases} \tag{3.6}$$

经过反复提取左因子，就能把每个非终结符(包括新引进者)的所有候选首字符集变为两两不相交(即不含公共左因子)。

我们也可以用数学中提取公共因子的办法来提取公共左因子。如对式(3.5)提取公共(左)因子后得

$$A\rightarrow \delta(\beta_1 \mid \beta_2 \mid \ldots \mid \beta_i) \mid \beta_{i+1} \mid \ldots \mid \beta_j$$　　(注：“(”与“)”为元语言符号)

将产生式中由“(”和“)”括起的部分以非终结符 A'命名则得到式(3.6)。

例如，对文法 G[A]：A→aAb｜a｜b 提取公共左因子后的文法 G'[A]为

G'[A]：A→aA'｜b

A'→Ab｜ε

5．递归下降分析器

在不含左递归和每个非终结符的所有候选终结首符集都两两不相交的条件下，我们就可能构造一个不带回溯的自顶向下的分析程序，这个分析程序是由一组递归过程(或函数)组成的，每个过程(或函数)对应文法的一个非终结符。这样的一个分析程序称为递归下降分析器。

例如，文法(3.2)对应的递归下降分析器如下：

```
void match(token t)
{
   if (lookahead == t)
      lookahead = nexttoken;
   else error( );
}
void E( )
{
   T( );
   E'( );
}
void E'( )
{
   if (lookahead == '+')
   {
      match ('+');
      T( );
      E'( );
   }
}
void T( )
{
   F( );
```

```
    T'( );
  }
  void T'( )
  {
    if (lookahead =='*')
    {
      match ('*');
      F( );
      T'( );
    }
  }
  void F( )
  {
    if (lookahead == 'i')
      match ('i');
    else if (lookahead =='(')
      {
        match ('(');
        E( );
        if (lookahead ==')')
            match (')');
        else error( );
      }
      else error( );
  }
```

我们知道，关于 E'的产生式是

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

即 E'只有两个候选式：第一个候选式的开头终结符为+，第二个候选式为 ε 。这就是说，当 E'面临输入符号“+”时就令第一个候选式进入工作，而当面临任何其它符号时，E'就自动认为获得了匹配(匹配于 ε)。递归函数 E'就是根据这一原则设计的。

例如，我们将递归函数的调用以栈的形式模拟来分析输入串 #i_1*(i_2+i_3)#的语法分析过程；在此，“#”为输入串 i_1*(i_2+i_3)的分隔符。进行语法分析时，首先将“#”和文法开始符 E 压入栈中，当语法分析进行到栈中仅剩“#”而输入串扫描指针已指向输入串尾部的“#”时，则语法分析成功，分析过程如图 3-14 所示。

注意：图 3-14 中第 (5) 步执行函数 F( ) 时，因当前扫描的字符为“(”，故匹配后应执行 E( )(用栈模拟即为将 E 压栈)；并且，在执行完 E( ) 后还应执行其后的判断“)”与匹配“)”语句，这在栈的模拟中则是标出此时 E 压栈之前的位置(见图 3-14 的第 (7)～(14) 步)，即弹栈至此时(第(14) 步结束时)应执行这个判断“)”与匹配“)”语句。

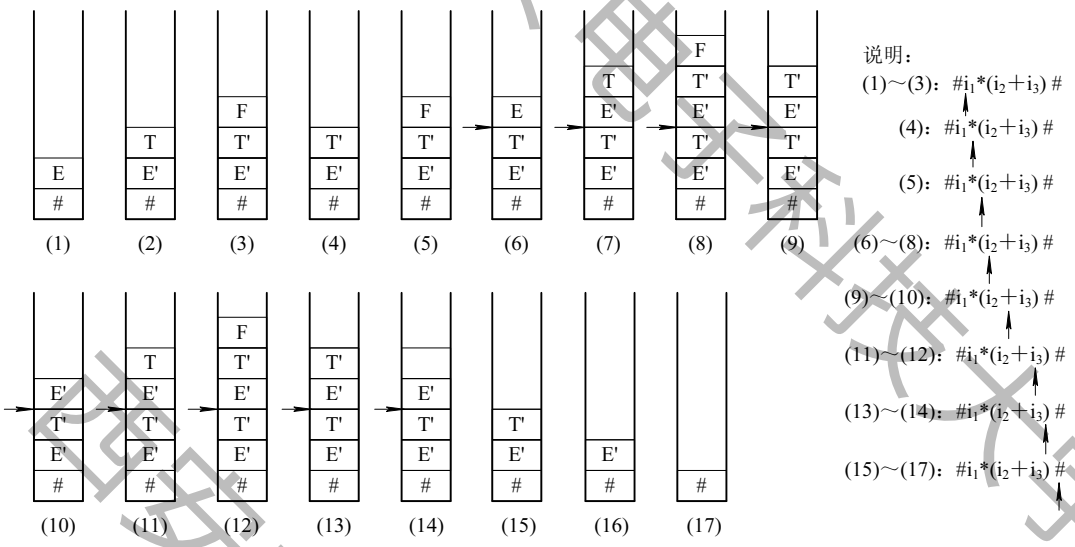

图 3-14　输入串 $i_1*(i_2+i_3)$的语法分析

我们也可以用消除了二义性的算术表达式文法来得到递归下降分析器：

G[E]：E→E+T｜T

T→T*F｜F

F→(E)｜i

由 E 的产生式 E→E+T｜T 可推出：

E⇒T

E⇒E+T⇒T+T

E⇒E+T⇒E+T+T⇒T+T+T

E⇒E+T⇒E+T+T⇒E+T+T+T⇒ T+T+T+T

…

即可得到 E 的正规表达式为：$T(+T)^*$，从而得到图 3-15 中 E 的状态转换图(DFA M)。同理，由 T 的产生式 T→T*F｜F 也可得到 T 的正规表达式为：$F(+F)^*$，从而得到图 3-15 中 T 的状态转换图。而产生式 F→(E)｜i 的状态转换图则可直接画出(见图 3-15 中 F 的状态转换图)。然后，借助于状态转换图来得到递归下降分析器。这种方法的特点是无需消除文法的左递归，但仍然要消除文法中的回溯。

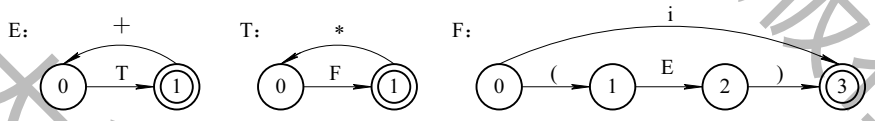

图 3-15　非终结符对应的状态转换图

图 3-15 中三个状态转换图的工作是以一种相互递归的方式进行的；因此，每个状态转换图的作用就如同一个递归过程(函数)。这时，前面的递归下降分析器程序可删除函数 E'()和 T'()，而将 E()和 T()改为

```
void E()
{
```

```
        T();
        while( lookahead=='+')
        {
            match('+');
            T();
        }
    }
    void T()
    {
        F();
        while( lookahead=='*')
        {
            match('*');
            F();
        }
    }
```

3.3.2 LL(1)分析法

LL(1)分析法又称预测分析法，是一种不带回溯的非递归自顶向下分析法。LL(1)的含义是：第一个 L 表明自顶向下分析是从左至右扫描输入串的；第二个 L 表明分析过程中将用最左推导；“1”表明只需向右查看一个符号就可决定如何推导(即可知用哪个产生式进行推导)。类似地，也可以有 LL(k)文法，也就是向前查看 k 个符号才能确定选用哪个产生式，不过 LL(k)(k>1)在实际中极少使用。

1. 表驱动的 LL(1)分析器

LL(1)分析法的基本思想是根据输入串的当前输入符号来唯一确定选用某条规则(产生式)来进行推导；当这个输入符号与推导的第一个符号相同时，再取输入串的下一个符号，继续确定下一个推导应选的规则；如此下去，直到推导出被分析的输入串为止。

一个 LL(1)分析器由一张 LL(1)分析表(也称预测分析表)、一个先进后出分析栈和一个控制程序(表驱动程序)组成，如图 3-16 所示。

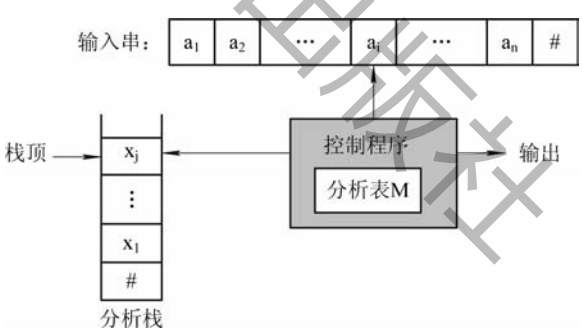

图 3-16 LL(1)分析器

对图 3-16 所示的 LL(1)分析器说明如下：

(1) 输入串是待分析的符号串，它以界符“#”作为结束标志(注：#∈V_T但不是文法符号，是由分析程序自动添加的)。

(2) 分析栈(又称符号号栈)中存放分析过程中的文法符号。分析开始时栈底先放入一个“#”，然后再压入文法的开始符号；当

分析栈中仅剩“#”，输入串指针也指向串尾的“#”时，分析成功。

(3) 分析表用一个矩阵(或二维数组)M 表示，它概括了相应文法的全部信息。矩阵的每一行与文法的一个非终结符相关联，而每一列与文法的一个终结符或界符“#”相关联。对 M[A,a]来说，A 为非终结符，而 a 为终结符或“#”。分析表元素 M[A,a]中的内容为一条关于 A 的产生式，表明当 A 面临输入符号 a 时当前推导所应采用的候选式；当元素内容为空白(空白表示“出错标志”)时，则表明 A 不应该面临这个输入符号 a，即输入串含有语法错误。

(4) 控制程序根据分析栈顶符号 x 和当前输入符号 a 来决定分析器的动作：

① 若 x = a = “#”，则分析成功，分析器停止工作。

② 若 x = a≠“#”，即栈顶符号 x 与当前扫描的输入符号 a 匹配；则将 x 从栈顶弹出，输入指针指向下一个输入符号，继续对下一个字符进行分析。

③ 若 x 为一非终结符 A，则查 M[A,a]：

i．若 M[A,a]中为一个 A 的产生式，则将 A 自栈顶弹出，并将 M[A,a]中的产生式右部符号串按逆序逐一压入栈中；如果 M[A,a]中的产生式为 A→ε，则只将 A 自栈顶弹出。

ii．若 M[A,a]中为空，则发现语法错误，调用出错处理程序进行处理。

控制程序描述如下：

```
将“#”和文法开始符依次压入符号栈中；
把第一个输入符号读入 a;
do{
    把栈顶符号弹出并放入 x 中;
    if(x∈V_T)
    {
        if(x==a)
        {
            if(a!='#')将下一输入符号读入 a;
        }
        else error();
    }
    else
        if(M[x,a] = "x→y1y2…yk")
        {
            按逆序依次把 yk、yk-1、…、y1 压入栈中;
            输出"x→y1y2…yk";
        }
        else error();
  }while(x!='#');
```

例 3.7　算术表达式文法(3.2)的 LL(1)分析表如表 3.1 所示，试给出输入串 $i_1+i_2*i_3$ 的分析过程。

表 3.1 算术表达式的 LL(1) 分析表

	i	+	*	(	)	#
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→i			F→(E)		

[解答] 输入串 $i_1+i_2*i_3$ 按控制程序进行的分析过程如表 3.2 所示。

表 3.2 输入串 $i_1+i_2*i_3$ 的分析过程

符号栈	输入串	当前输入符号	说 明
#E	$i_1+i_2*i_3$#	i_1	弹出栈顶符号 E，将 M[E,i]中 E→TE'的 TE'逆序压栈
#E'T	$i_1+i_2*i_3$#	i_1	弹出栈顶符号 T，将 M[T,i]中 T→FT'的 FT'逆序压栈
#E'T'F	$i_1+i_2*i_3$#	i_1	弹出栈顶符号 F，将 M[F,i]中 F→i 的 i 压栈
#E'T'i	$i_1+i_2*i_3$#	i_1	匹配，弹出栈顶符号 i 并读出输入串的下一个输入符号+
#E'T'	$+i_2*i_3$#	+	弹出栈顶符号 T'，因 M[T',+]中为 T'→ε，故不压栈
#E'	$+i_2*i_3$#	+	弹出栈顶符号 E'，将 M[E',+]中 E'→+TE'的+TE'逆序压栈
#E'T+	$+i_2*i_3$#	+	匹配，弹出栈顶符号+并读出输入串的下一个输入符号 i_2
#E'T	i_2*i_3#	i_2	弹出栈顶符号 T，将 M[T,i]中 T→FT'的 FT'逆序压栈
#E'T'F	i_2*i_3#	i_2	弹出栈顶符号 F，将 M[F,i]中 F→i 的 i 压栈
#E'T'i	i_2*i_3#	i_2	匹配，弹出栈顶符号 i 并读出输入串的下一个输入符号*
#E'T'	$*i_3$#	*	弹出栈顶符号 T'，将 M[T',*]中 T'→*FT'的*FT'逆序压栈
#E'T'F*	$*i_3$#	*	匹配，弹出栈顶符号*并读出输入串的下一个输入符号 i_3
#E'T'F	i_3#	i_3	弹出栈顶符号 F，将 M[F,i]中 F→i 的 i 压栈
#E'T'i	i_3#	i_3	匹配，弹出栈顶符号 i 并读出输入串的下一个输入符号#
#E'T'	#	#	弹出栈顶符号 T'，因 M[T',#]中为 T'→ε，故不压栈
#E'	#	#	弹出栈顶符号 E'，因 M[E',#]中为 T'→ε，故不压栈
#	#	#	匹配，分析成功

2．LL(1)分析表的构造

在表驱动的 LL(1)分析器中，除了分析表因文法的不同而异之外，分析栈、控制程序都是相同的。因此，构造一个文法的表驱动 LL(1)分析器实际上就是构造该文法的分析表。为了构造分析表 M，需要预先定义和构造两族与文法有关的集合 FIRST 和 FOLLOW。

假定 α 是文法 G[S]的任一符号串($\alpha\in(V_T\cup V_N)^*$)，可定义

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

如果 $\alpha \overset{*}{\Rightarrow} \varepsilon$，则规定 $\varepsilon\in\mathrm{FIRST}(\alpha)$；也即，FIRST(α)是 α 的所有可能推导的开头终结符或可能的 ε。这里的 FIRST(α)只是一般的公式，具体到文法中，只需求出文法中所有非终结符的 FIRST 集即可。

假定 S 是文法 G[S]的开始符号，对 G[S]的任何非终结符 A，可定义

$$FOLLOW(A)=\{a \mid S\overset{*}{\Rightarrow}\cdots Aa\cdots, a\in V_T\}$$

如果 $S\overset{*}{\Rightarrow}\cdots A$，则规定界符#∈FOLLOW(A)；也即，FOLLOW(A)是所有句型中出现在紧随 A 之后的终结符或“#”。

(1) FIRST 集构造方法：对文法中的每一个非终结符 X 构造 FIRST(X)，其方法是连续使用下述规则，直到每个集合的 FIRST 不再增大为止。(注：对终结符 a 而言，FIRST('a')={a}，因而无需构造。)

① 若有产生式 X→a…，且 a∈V_T ，则把 a 加入到 FIRST(X)中；若存在 X→ε，则将 ε 也加入到 FIRST(X)中。

② 若有 X→Y…，且 Y∈V_N，则将 FIRST(Y)中的所有非 ε 元素(记为“\{ε}”)都加入到 FIRST(X)中；若有 X→$Y_1Y_2\cdots Y_k$，且 Y_1～Y_i都是非终结符，而 Y_1～Y_i的候选式都有 ε 存在，则把 FIRST(Y_j)(j=1,2,…,i)的所有非 ε 元素都加入到 FIRST(X)中；特别是当 Y_1～Y_k均含有 ε 产生式时，应把 ε 也加到 FIRST(X)中。

(2) FOLLOW 集构造方法：对文法 G[S]的每个非终结符 A 构造 FOLLOW(A)的方法是连续使用下述规则，直到每个 FOLLOW 不再增大为止。

① 对文法开始符号 S，置#于 FOLLOW(S)中(由语句括号“#S#”中的 S#得到)。

② 若有 A→αBβ(α 可为空)，则将 FIRST(β)\{ε}加入到 FOLLOW(B)中。

③ 若有 A→αB 或 A→αBβ，且 $\beta\overset{*}{\Rightarrow}\varepsilon$(即 ε∈FIRST(β))，则把 FOLLOW(A)加到 FOLLOW(B)中(此处的 α 也可为空)。

注意：对 FIRST 集构造方法 ②，若有 a∈FIRST(Y)，即有 Y→a…；如果存在 X→Y…，则由 X⇒Y…⇒a…可知 a∈FIRST(X)；即若有 X→Y…，则有 FIRST(Y)\{ε}⊂FIRST(X)。

对于 FOLLOW 集构造方法 ②，可理解为在文法中有形如…Aα…的产生式，则有 FIRST(α)\{ε}⊂FOLLOW(A)；而对构造方法 ③，对形如 A→…B 的产生式，如果存在…Aa…的符号串，则用 A→…B 可推得：…Ba…，也即原属于 FOLLOW(A)的字符 a 此时必定属于 FOLLOW(B)，即有 FOLLOW(A)⊂FOLLOW(B)。

此外，构造 FIRST 集和 FOLLOW 集的过程有可能要反复进行多次，直到每一个非终结符的 FIRST 集和 FOLLOW 集都不再增大为止。

FIRST 集确定了每一个非终结符在扫描输入串时所允许遇到的输入符号及所应采用的推导产生式(该非终结符所对应的产生式中的哪一个候选式)。

FOLLOW 集是针对文法中形如“A→ε”这样产生式的，即在使用 A 的产生式进行推导时，面临输入串中哪些输入符号时则此时有一空字(即 ε)匹配而不出错；当然，此时的扫描指针仍指向当前扫描的输入符号上，并不向前推进。

(3) 构造分析表 M。

① 对文法 G[S]的每个产生式 A→α 执行以下②、③步。

② 对每个终结符 a∈FIRST(A)，把 A→α 加入到 M[A,a]中，其中 α 为含有首字符 a 的候选式或为唯一的候选式。

③ 若ε∈FIRST(A)(或文法中有A→ε的产生式)则对任何属于FOLLOW(A)的终结符b，将 A→ε 加入到 M[A,b]中。

④ 把所有无定义的 M[A,a]标记为“出错”。

一个文法 G[S]，若它的分析表 M 不含多重定义入口，则称它是一个 LL(1)文法，它所定义的语言恰好就是它的分析表所能识别的全部句子。一个上下文无关文法是 LL(1)文法的充分必要条件是：对每一个非终结符 A 的任何两个不同产生式 A→α｜β，有下面的条件成立：

(1) FIRST(α)∩FIRST(β)＝Φ；

(2) 假若 $\beta \overset{*}{\Rightarrow} \varepsilon$，则有 FIRST(α)∩FOLLOW(A)=Φ。

条件(1)意味着 A 的每个候选式都不存在相同的首字符，由 LL(1)分析表的构造方法可知：它避免了在分析表的同一栏目内出现多个产生式的情况，即避免了多重入口。

条件(2)避免了在分析表的同一栏目内出现 A→α 和 A→ε(这同样是多重入口)的情况。例如对文法：

G[S]：S→Aa｜b

A→a｜ε

则有 FIRST(A)={a, ε}、FOLLOW(A)={a}，此时文法对应的分析表 M[A,a]栏里必然有两个产生式 A→α 和 A→ε 存在，即形成了多重入口；而此时由条件(2)也可以得知：FIRST('a')∩FOLLOW(A)={a}≠Φ。

注意：LL(1) 文法首先是无二义的，这一点可以从分析表不含多重定义入口得知；并且，含有左递归的文法绝不是 LL(1) 文法，所以必须首先消除文法的一切左递归。其次，应该消除回溯(即提取公共左因子)。但是，文法中不含左因子只是 LL(1) 文法的必要条件，一个文法提取了公共左因子后，只解决了非终结符对应的所有候选式不存在相同首字符的问题(即每个候选式的 FIRST 集互不相交)，只有当改写后的文法不含 ε 产生式且无左递归时才可立即断定该文法是 LL(1) 文法，否则必须用上面 LL(1) 文法的充要条件进行判定(或者看 LL(1) 分析表中是否存在多重入口来判定)。

例 3.8 试构造表达式文法 G[E]的 LL(1)分析表，其中：

G[E]：E→TE'

E'→+TE'｜ε

T→FT'

T'→*FT'｜ε

F→(E)｜i

[解答] 首先构造 FIRST 集，步骤如下：

(1) FIRST(E')={+,ε}；FIRST(T')={*,ε}；FIRST(F)={(,i}；

(2) 由 T→F… 和 E→T… 知 FIRST(F)⊂FIRST(T)⊂FIRST(E)，即有 FIRST(T)=FIRST(E)= {(,i}。

其次构造 FOLLOW 集，步骤如下：

(1) FOLLOW(E)={#}；

(2) 由 E→TE'知 FIRST(E')\{ε}⊂FOLLOW(T)，即 FOLLOW(T)={+}；

由 T→FT'知 FIRST(T')\{ε}⊂FOLLOW(F)，即 FOLLOW(F)={*}；

由 F→(E)知 FIRST(')')⊂FOLLOW(E)，即 FOLLOW(E)={),#}；

(3) 由 E→TE'知 FOLLOW(E)⊂FOLLOW(E')，即 FOLLOW(E')={),#}；

由 E→TE'且 E'→ε 知 FOLLOW(E)⊂FOLLOW(T)，即 FOLLOW(T)={+,),#}；

由 T→FT'知 FOLLOW(T)⊂FOLLOW(T')，即 FOLLOW(T')={+,),#}；

由 T→FT'且 T'→ε 知 FOLLOW(T)⊂FOLLOW(F)，即 FOLLOW(F)={*,+,),#}；

最后得到文法 G[E]的 LL(1)分析表如表 3.1 所示。

例 3.9　程序语言中 if-else 语句的文法 G[S]为

G[S]：S→iESeS｜iES｜a

E→b

其中，e 遵从最近匹配原则。试改造文法 G[S]并为之构造 LL(1)分析表。

[解答]　提取公共左因子后得到文法 G'[S]：

G'[S]：S→iESS'｜a

S'→eS｜ε

E→b

求出每个非终结符的 FIRST 集和 FOLLOW 集。

FIRST 集构造：

FIRST(S)={i,a}；　　FIRST(S')={e,ε}；

FIRST(E)={b}。

FOLLOW 集构造：

(1) FOLLOW(S)={#}；

(2) 由 S→…ES… 知 FIRST(S)\{ε}⊂FOLLOW(E)，即 FOLLOW(E)={i,a}；

由 S→…S S' 知 FIRST(S')\{ε}⊂FOLLOW(S)，即 FOLLOW(S)={e,#}；

(3) 由 S→…S S'知 FOLLOW(S)⊂FOLLOW(S')，即 FOLLOW(S')={e,#}。

构造分析表见表 3.3。

表 3.3　例 3.9 的分析表

	i	e	a	b	#
S	S→iESS'		S→a		
S'		S'→eS S'→ε			S'→ε
E				E→b	

我们看到，分析表 M 含有多重入口冲突项 M[S',e]，因此文法 G[S]不是 LL(1)文法(实际上 G[S]是一个二义文法，二义文法构造的 LL(1)分析表必定含有冲突项)。我们可以通过图 3-17 来解决 M[S',e]栏的冲突。对图 3-17，iES 面临输入符号 e 时绝不能用 ε 匹配；如果用 ε 匹配，则认为 iES 是一个句型而丢掉了其后的 eS(这样做将认为 eS 是一个新的句型，而 eS 本身却无法构成一个句型)，即最终不可能得到句型 iESeS。因此，应继续扫描 eS 以便形成最终的句型 iESeS；这也意味着，在分析表的 M[S',e]栏内应舍弃 S'→ε 而保留 S'→eS。由此得到无二义的 LL(1)分析表见表 3.4。

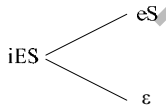

图 3-17　iES 面临输入符号 e 时可能出现的情况

表 3.4 例 3.9 的 LL(1) 分析表

	i	e	a	b	#
S	S→iESS'		S→a		
S'		S'→eS			S'→ε
E				E→b	

例 3.10 证明下述文法 G[S]不是 LL(1)文法，并给出其等价的 LL(1)文法。

G[S]：S→LA
L→i:｜ε
A→i=e

[解答] FIRST 集构造：

(1) FIRST(L)={i,ε};
FIRST(A)={i};

(2) 由 S→L…得 FIRST(L)\{ε}⊂FIRST(S)，即 FIRST(S)={i};
又由 S→LA 和 L→ε 推出 S→A，则 FIRST(A)\{ε}⊂FIRST(S)，即 FIRST(S)={i}

FOLLOW 集构造：

(1) FOLLOW(S)={#};

(2) 由 S→LA 得：FIRST(A)\{ε}⊂FOLLOW(L)，即 FOLLOW(L)={i};

(3) 由 S→LA 得：FOLLOW(S)⊂FOLLOW(A)，即 FOLLOW(A)={#};

对产生式 L→i:｜ε，由于有 FIRST(i:)∩FOLLOW(L)={i}∩{i}≠Φ，即不满足 LL(1)文法充分必要条件(2)，所以文法 G[S]不是 LL(1)文法。

为了满足 LL(1)文法的条件，需对文法 G[S]进行如下改造：

(1) 消去非终结符 L 和 A，得到：

G'[S]：S→i:i=e｜i=e

由此我们也可以看出 G'[S]存在着回溯(即含有公共左因子)，故不是 LL(1)文法。

(2) 提取公共左因子 i 得到：

G"[S]：S→iA
A→:i=e｜=e

这时有：

FIRST(S)={i};
FIRST(A)={:,=};
FOLLOW(S)={#};

由 S→iA 得：FOLLOW(S)⊂FOLLOW(A)，即

FOLLOW(A)={#};

而此时有：

FIRST(iA)∩FOLLOW(S)={i}∩{#}=Φ

故文法 G"[S]是与 G[S]等价的 LL(1)文法。

3.4　自底向上的语法分析

自底向上的语法分析与自顶向下的语法分析相比，它无需消除左递归和回溯；对某些二义文法，也可以采用自底向上分析方法。因此，自底向上分析的适用范围更大。

3.4.1　自底向上分析原理

所谓自底向上分析就是自左至右扫描输入串，自底向上进行分析；通过反复查找当前句型的句柄(最左直接短语)，并使用产生式规则将找到的句柄归约为相应的非终结符。这样逐步进行“归约”，直到归到文法的开始符号；或者说，从语法树的末端开始，步步向上“归约”，直到根结点。

自底向上分析法是一种“移进-归约”法，这是因为在自底向上分析过程中采用了一个先进后出的分析栈。分析开始后，把输入符号自左至右逐个移进分析栈，并且边移入边分析，一旦栈顶的符号串形成某个句型的句柄就进行一次归约，即用相应产生式的左部非终结符替换当前句柄。接下来继续查看栈顶是否形成新的句柄，若为句柄则再进行归约；若栈顶不是句柄则继续向栈中移进后续输入符号。不断重复这一过程，直到将整个输入串处理完毕。若此时分析栈只剩有文法的开始符号则分析成功，即确认输入串是文法的一个句子；否则，即认为分析失败。

我们举一个简单的例子来说明这种分析过程。假设一文法 G[S] 为

G[S]：S→aAbB
　　A→c | Ac
　　B→d

试对输入串 accbd 进行分析，检查该符号串是否是 G[S] 的一个句子。

我们仍将“#”作为输入串的界符(括号)，并将输入串前的“#”放入分析栈，接着将输入串的符号依次进栈，具体分析过程如表 3.5 所示。

表 3.5　对输入串 accbd 自底向上的分析过程

步骤	分析栈	句　柄	输入串	动　作
1	#		accbd#	移进
2	#a		ccbd#	移进
3	#ac		cbd#	移进
4	#aA	c	cbd#	归约(A→c)
5	#aAc		bd#	移进
6	#aA	Ac	bd#	归约(A→Ac)
7	#aAb		d#	移进
8	#aAbd		#	移进
9	#aAbB	d	#	归约(B→d)
10	#S	aAbB	#	归约(S→aAbB)

上述语法分析过程可以用一棵分析树来表示。在自底向上分析过程中，每一步归约都可以画出一棵子树来，随着归约的完成，这些子树被连成一棵完整的分析树。根据表 3.5 构造分析树的过程如图 3-18 所示。

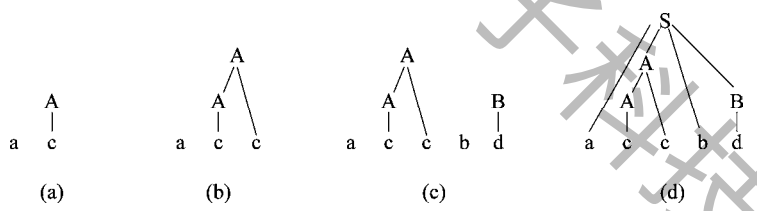

图 3-18 自底向上构造分析树的过程

从建立分析树的过程可以清楚地看出，自底向上分析过程的每一步归约确实都是归约当前句型的句柄；也就是说句柄一旦形成，则它总是出现在分析栈的栈顶而不会出现在栈的中间。由于每一步归约都是把栈顶的一串符号(此时，这串符号已经是某个产生式中的一个候选式)，用该产生式的左部符号(一个终结符号)替换，因而可以把栈顶的这样一串符号称为“可归约串”。初看起来，“移进-归约”似乎很简单，其实不然。在上例分析进行到第 6 步时，如果我们不是选择规则 A→Ac 而是选择规则 A→c 进行归约，也即把 c 看作句柄的话，则最终就无法达到归约到 S 的目的，从而也就无法得知输入串 accbd 是一个符合文法的句子。为什么知道此时栈顶的 Ac 形成“可归约串”而 c 不是“可归约串”呢？这就需要精确定义“可归约串”这个直观概念，这也是自底向上分析的关键问题。

我们在 3.2.1 节知道了推导与短语的概念，并且知道归约是推导的逆过程。这里，我们进一步指出：如果文法 G[S]是无二义的，那么规范推导(最右推导)的逆过程必是规范归约(最左归约)。

请注意句柄的“最左”特征，这一点对于“移进-归约”来说很重要，因为句柄的“最左”性和分析栈的栈顶两者是相关的。对于规范句型(规范推导所得的句型)来说，句柄的后面不会出现非终结符(也即句柄的后面只能出现终结符)。否则，之前找到的句柄必定不是最左直接短语，这也意味着前面所进行的寻找句柄及归约过程有误。基于这一点，我们可用句柄来刻画“移进-归约”过程的“可归约串”。因此，规范归约的实质是，在移进过程中，当发现栈顶呈现句柄时就用相应产生式的左部符号进行替换(即归约)。

上述分析有助于理解为什么规范归约所得的分析树恰好就是语法树。

为了加深对“句柄”和“归约”这些重要概念的理解，我们使用修剪语法树的办法来进一步阐明自底向上的分析过程。

一棵语法树的一个子树是由该树的某个内部结点(作为子树的根)连同它的所有子孙(如果有的话)组成的，也即一个子树的所有树叶自左至右排列起来形成一个相对于子树根的短语，并且一个句型的句柄是这个句型所对应的语法树中最左那个子树树叶自左至右的排列；这个子树只有(而且必须有)父子两代，没有第三代。

注意：如果一个树叶序列由左至右的排列可以向上归结到某一个内部结点(比如说 A)，并且由结点 A 向下生长出的全部树叶也恰是这个树叶序列，则这个树叶序列就是结点 A 的短语。如果这种向上归结只需要一层(即树叶与结点 A 都为父子关系)，则该树叶序列为直接短语。需要说明的是，如果一个树叶序列最终无法全部归结到一个结点上，或者归结到

某结点上的树叶序列并不完整，则该树叶序列不是短语。

例如，对图 3-18(d)所示的语法树，我们采用修剪语法树的方法来实现归约，也即每次寻找当前语法树的句柄(在语法树中用虚线勾出)，然后将句柄中的树叶剪去(即实现一次归约)；这样不断地修剪下去，当剪到只剩下根结点时，就完成了整个归约过程，如图 3-19 所示。

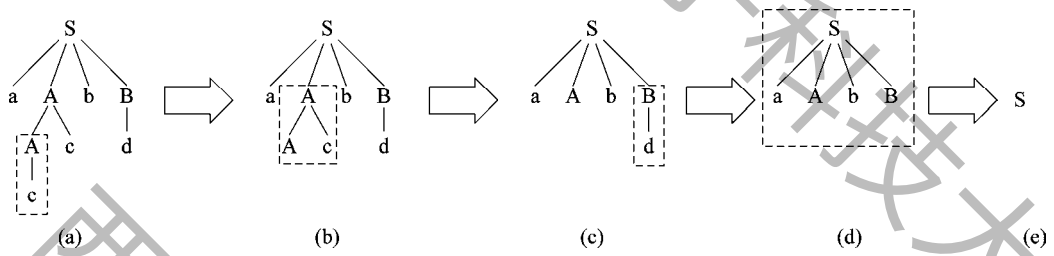

图 3-19　修剪语法树实现归约

至此，我们简单地讨论了“句柄”和“规范归约”这两个基本概念，但并没有解决规范归约的问题，因为我们并没有给出寻找句柄的算法。事实上，规范归约的中心问题就是如何寻找或确定一个句型的句柄。给出了寻找句柄的不同算法就给出了不同的规范归约方法，这一点我们将在 LR 分析器中讨论。

3.4.2　算符优先分析法

算符优先分析法是一种简单且直观的自底向上分析方法，它特别适合于分析程序语言中的各类表达式，并且宜于手工实现。

所谓算符优先分析，就是依照算术表达式的四则运算过程来进行语法分析，即这种分析方法要预先规定运算符(确切地说是终结符)之间的优先关系和结合性质，然后借助于这种关系来比较相邻运算符的优先级，以确定句型的“可归约串”来进行归约。因此，算符优先分析法不是一种规范归约，在整个归约过程中起决定性作用的是相继两个终结符的优先关系。

1．算符优先文法

如果一个文法存在…QR…的句型(Q 和 R 都是非终结符)，则这种形式的句型意味着两个算符之间操作数的个数是不确定的，也就意味着每个算符的操作数是不确定的。因此，我们首先定义一个任何产生式的右部都不含两个相继(并列)的非终结符的文法为算符文法，即算符优先文法首先应是一个算符文法；其次，我们还要定义任何两个可能相继出现的终结符 a 与 b (它们之间最多插有一个非终结符)的优先关系。

假定 G[S]是一个不含 ε 产生式的算符文法，对于任何一对终结符 a、b，有：

(1) a$\doteq$b，当且仅当 G[S]中含有形如 R→…ab…或 R→…aQb…的产生式；

(2) a$\lessdot$b，当且仅当 G[S]中含有形如 R→…aP…的产生式，而 P$\doteq$b…或 P$\doteq$Qb…；

(3) a$\gtrdot$b，当且仅当 G[S]中含有形如 R→…Pb…的产生式，而 P$\doteq$…a 或 P$\doteq$…aQ。

如果一个算符文法 G[S]中的任何终结符对(a,b)至多满足下述三种关系之一(相同、低于、高于)：

$$a \doteq b，\ a \lessdot b，\ a \gtrdot b$$

则称 G[S]是一个算符优先文法。

例 3.11 试说明下述算术表达式文法 G[E]是一个算符文法，但不是算符优先文法：

G[E]：E→E+E｜E*E｜(E)｜i

[解答] 由于文法 G[E]中的任何产生式右部都不含两个相邻的非终结符，所以 G[E]是算符文法。此外，因为

(1) 由于存在 E→E+E，而 E+E 中的第二个 E 可推出 E⇒E*E，即有+⋖*；

(2) 由于存在 E→E*E，而 E*E 中的第一个 E 可推出 E⇒E+E，即有+⋗*。

此即运算符 + 和 * 之间同时存在着两种不同的优先关系，故文法 G[E]不是一个算符优先文法。

2. 算符优先关系表的构造

我们通过图 3-20 的语法树来说明相继两个终结符之间的优先关系。对图 3-20(a)、(b)，根据语法树自底向上的归约方法，首先应该把 ab 或 aQb 归约为 R，然后再将 cRd 归约为 T；即对相继两个终结符 a 与 b 有：a⋍b，而对相继两个终结符 c 与 a 和 b 与 d 应分别有：c⋖a 和 b⋗d。

同样，对图 3-20(c)、(d)来说，应先将 b…或 Qb…归约为 P，然后再将 aP 归约为 R，因此对相继两个终结符 a 与 b 应有：a⋖b。对图 3-20(e)、(f)来说，则应先将…a 或…aQ 归约为 P，然后再将 Pb 归约为 R，即对相继两个终结符 a 与 b 应有 a⋗b。

对于图 3-20(c)、(d)来说，我们要找出形如 P $\overset{+}{\Rightarrow}$ b…或 P $\overset{+}{\Rightarrow}$ Qb…推导中出现的所有不同的终结符 b，则对这些不同的终结符 b 均应有：a⋖b。

对于图 3-20(e)、(f)也是如此，我们要找出形如 P $\overset{+}{\Rightarrow}$ …a 或 P $\overset{+}{\Rightarrow}$ …aQ 这种推导中出现的所有不同的终结符 a，则对这些不同的终结符 a 均应有：a⋗b。

由图 3-20 可以看出，位于同层的相继两个终结符比较，其优先级相同；位于不同层的相继两个终结符比较，则层次在上的优先级低，层次在下的优先级高。

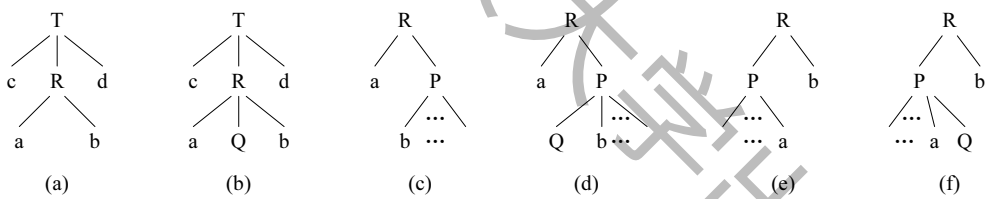

图 3-20 不同优先关系的语法树示意

因此，为了找出所有满足关系“⋖”和“⋗”的终结符对，我们首先需要对 G[S]的每个非终结符 P 构造两个集合 FIRSTVT(P)和 LASTVT(P)：

$$\text{FIRSTVT}(P)=\{a \mid P\overset{+}{\Rightarrow}a\cdots \text{或 } P\overset{+}{\Rightarrow}Qa\cdots,\ a\in V_T \text{ 而 } Q\in V_N\}$$

$$\text{LASTVT}(P)=\{a \mid P\overset{+}{\Rightarrow}\cdots a \text{ 或 } P\overset{+}{\Rightarrow}\cdots aQ,\ a\in V_T \text{ 而 } Q\in V_N\}$$

由此，得到 FIRSTVT 集的构造方法如下：

(1) 若有产生式 P→a…或 P→Qa…，则 a∈FIRSTVT(P)；

(2) 若有产生式 P→Q…，则 FIRSTVT(Q)⊂FIRSTVT(P)。

得到 LASTVT 集的构造方法如下：

(1) 若有产生式 P→…a 或 P→…aQ，则 a∈LASTVT(P)；

(2) 若有 P→…Q，则 LASTVT(Q)⊂LASTVT(P)。

通过检查 G[S]的每个产生式的每个候选式，我们还可以找出所有满足 a$\doteq$b 的终结符对。

注意：对于 FIRSTVT 集构造方法(1)，FIRSTVT(P) 包含形如 P→a…(以 a 打头)或 P→Qa…(以 Qa 打头)的终结符“a”；而 LL(1) 分析法中如果没有 Q→ε，则 FIRST 集不含 P→Qa…中的“a”。对于 FIRSTVT 集构造方法(2)，若有 P→Q…(以 Q 打头)，并且存在 Q→a…或 Q→Ra…，则 a∈FIRSTVT(Q)；由 P→Q…可得到：P⇒Q…⇒a…或者 P⇒Q…⇒Ra…，即有 a∈FIRSTVT(P)；因此，若有 P→Q…，则必有 FIRSTVT(Q)⊂FIRSTVT(P)，这与 LL(1) 分析法中若有 P→Q…，则必有 FIRST(Q)⊂FIRST(P) 的解法类似。由 FIRSTVT 集的构造方法(1)、(2)可知，算符优先文法中的 FIRSTVT 集要大于 LL(1) 分析法中的 FIRST 集。

对于 LASTVT 集构造方法(1)，LASTVT(P) 包含形如 P→…a(以 a 结尾)或 P→…aQ(以 aQ 结尾)的终结符“a”。对于 LASTVT 集构造方法(2)，若有 P→…Q(以 Q 结尾)，并且存在 Q→…a 或 Q→…aK，则 a∈LASTVT(Q)；由 P→…Q 可得到：P⇒…Q⇒…a 或者 P⇒…Q⇒…aK，即有 a∈LASTVT(P)；因此，若有 P→…Q，则必有 LASTVT(Q)⊂LASTVT(P)。对比 LL(1) 分析法中 FOLLOW 集，FOLLOW(P) 是指紧跟非终结符 P 之后所有可能出现的第一个终结符构成的集合，它与 LASTVT 集的概念是完全不同的。

至此，我们得到从算符优先文法 G[S] 构造优先关系表的方法如下：

(1) 对形如 R→…ab…或 R→…aQb…的产生式，有 a$\doteq$b；

(2) 对形如 R→…aP…的产生式，若有 b∈FIRSTVT(P)，则 a$\lessdot$b；

(3) 对形如 R→…Pb…的产生式，若有 a∈LASTVT(P)，则 a$\gtrdot$b。

此外，若将语句括号“#”作为终结符对待，且 S 是文法 G[S] 的开始符，则应有#S#存在；也即，可由上述构造方法得到：#$\doteq$#；#$\lessdot$FIRSTVT(S)；LASTVT(S)$\gtrdot$#(此处指“#”与 FIRSTVT(S) 或 LASTVT(S) 集合中的元素即终结符之间的优先关系)。

例 3.12　试构造下述算术表达式文法 G[E] 的算符优先关系表：

G[E]：E→E+T｜T
　　T→T*F｜F
　　F→(E)｜i

[**解答**]　(1) 构造 FIRSTVT 集。

① 根据规则(1)知：

由 E→E+…得 FIRSTVT(E)={+}；

由 T→T*…得 FIRSTVT(T)={*}；

由 F→(…和 F→i 得 FIRSTVT(F)={(,i}。

② 根据规则(2)知：

由 T→F 得 FIRSTVT(F)⊂FIRSTVT(T)，即 FIRSTVT(T)={*,(,i}；

由 E→T 得 FIRSTVT(T)⊂FIRSTVT(E)，即 FIRSTVT(E)={+,*,(,i}。

(2) 构造 LASTVT 集。

① 根据规则(1)知：

由 E→…+T 得 LASTVT(E)={+}；

由 T→…*F 得 LASTVT(T)={*}；

由 F→…) 和 F→i 得 LASTVT(F)={),i}。

② 根据规则(2)知：

由 T→F 得 LASTVT(F)⊂LASTVT(T)，即 LASTVT(T)= {*,),i}；

由 E→T 得 LASTVT(T)⊂LASTVT(E)，即 LASTVT(E)= {+,*,),i}。

(3) 构造优先关系表。

① 根据规则(1)知：由“(E)”得(=)。

② 根据规则(2)知：

由 E→…+T 得+⋖FIRSTVT(T)，即：+⋖*，+⋖(，+⋖i；

由 T→…*F 得*⋖FIRSTVT(F)，即：*⋖(，*⋖i；

由 F→(E…得(⋖FIRSTVT(E)，即：(⋖+，(⋖*，(⋖(，(⋖i。

③ 根据规则(3)知：

由 E→E+…得 LASTVT(E)⋗+，即：+⋗+，*⋗+，)⋗+，i⋗+；

由 T→T*…得 LASTVT(T)⋗*，即：*⋗*，)⋗*，i⋗*；

由 F→…E)得 LASTVT(E)⋗)，即：+⋗)，*⋗)，)⋗)，i⋗)。

此外，由#E#得#=#；#⋖FIRSTVT(E)，即#⋖+，#⋖*，#⋖(，#⋖i；LASTVT(E)⋗#，即+⋗#，*⋗#，)⋗#，i⋗#。

最后得到算术表达式的优先关系表如表 3.6 所示。

表 3.6 优 先 关 系 表

	+	*	i	(	)	#
+	⋗	⋖	⋖	⋖	⋗	⋗
*	⋗	⋗	⋖	⋖	⋗	⋗
i	⋗	⋗			⋗	⋗
(	⋖	⋖	⋖	⋖	=	
)	⋗	⋗			⋗	⋗
#	⋖	⋖	⋖	⋖		=

3. 算符优先分析算法的设计

由于算符优先分析法不是一种规范归约的分析方法，它仅在终结符之间定义了优先关系而未对非终结符定义优先关系，这样就无法使用优先关系表去识别由单个非终结符组成的可归约串(如 E→T)。因此，算符优先分析法实际上不是用句柄来刻画“可归约串”，而是用最左素短语来刻画“可归约串”。

在 3.2.1 节我们已经给出了素短语的定义，并在 3.2.2 节给出了素短语的求解方法。在此，我们须再次强调：所谓句型的素短语，是指这样一种短语，它至少包含一个终结符，并且除自身之外，不再包含其它更小的素短语。最左素短语则是指处于句型最左边的那个素短语。下面，我们给出求解素短语的另一种方法。

对算符优先文法，其句型的一般形式可表示为(括在两个#之间)

$$\#N_1a_1N_2a_2\cdots N_na_nN_{n+1}\#$$

其中，每个 a_i 都是终结符，而 N_i 则是可有可无的非终结符。算符文法的特点决定了该句型这 n 个终结符的任何两个相邻的终结符之间顶多只有一个非终结符。

由上述句型可找出该句型中的所有素短语，每个素短语要素(指仅包含终结符的序列)

都具有下述形式：

$$a_{j-1} \lessdot \underbrace{a_j \doteq a_{j+1} \doteq \cdots \doteq a_i}_{\text{素短语要素}} \gtrdot a_{i+1}$$

实际上，我们通过拓展图 3-20(b)可得到素短语所对应的语法树如图 3-21 所示。由图 3-21 可以看出，需要进行的归约是将符号串 $N_ja_jN_{j+1}a_{j+1}\cdots a_iN_{i+1}$ 归约为 R，而这个符号串恰好就是素短语，并且存在优先关系：$a_{j-1} \lessdot a_j$，$a_j \doteq a_{j+1}$，…，$a_{i-1} \doteq a_i$，$a_i \gtrdot a_{i+1}$。

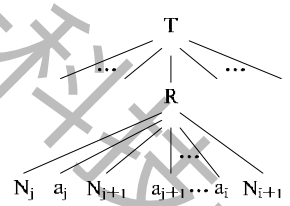

图 3-21　素短语对应的语法树示意

注意：素短语要素仅包含了构成素短语的终结符序列，再添加构成该短语的非终结符则形成了一个素短语；而最左素短语就是该句型中找到的最左边的那个素短语要素与该要素有关的非终结符所组成的短语。

最左素短语必须具备三个条件：

(1) 至少包含一个终结符(是否包含非终结符则按短语的要求确定)；

(2) 除自身外不得包含其它素短语(最小性)；

(3) 在句型中具有最左性。

此外，一定要注意最左素短语与句柄的区别。

查找最左素短语的方法如下：

(1) 最左子串法。找出句型中最左子串且终结符由左至右的对应关系满足 $a_{j-1} \lessdot a_j \doteq a_{j+1} \doteq \cdots \doteq a_i \gtrdot a_{i+1}$，然后检查文法中每个产生式的每个候选式，看是否存在这样一个候选式，该候选式中的所有终结符由左至右的排列恰为 $a_ja_{j+1}\cdots a_i$，即每一位终结符均对应相等，而非终结符仅对应位置存在即可。如果存在这样的候选式，则该候选式(包括其中的非终结符)即为该句型的最左素短语。

(2) 语法树法。设句型为 ω，先画出对应句型 ω 的语法树，然后找出该语法树中所有相邻终结符之间的优先关系。语法树确定相邻终结符之间优先关系的原则如下：

① 同层的优先关系为“$\doteq$”；

② 不同层时，层次在下的优先级高，层次在上的优先级低(这一点恰好验证了优先关系表的构造方法)；

③ 在句型 ω 两侧加上语句括号“#”，即#ω#，则有#$\lessdot$ω 和 ω$\gtrdot$#。

最后，按最左素短语必须具备的三个条件来确定最左素短语。

例 3.13　已知文法 G[E]为

$$\begin{aligned} G[E]:\ & E \to E+T \mid T \\ & T \to T*F \mid F \\ & F \to (E) \mid i \end{aligned}$$

试确定 F+T*i 的最左素短语。

[解答]　画出对应句型 F+T*i 的语法树并根据语法树确定相邻终结符之间的优先关系(见图 3-22)。

根据最左素短语必须具备的条件及短语的要求(即必须包含某内部结点向下生长全部

树叶)得到最左素短语为 i (该句型的最左直接短语为 F，注意两者的区别)。

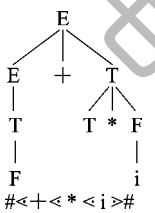

图 3-22 F+T*i 的语法树及其优先关系

根据上述查找最左素短语的方法，我们得到算符优先分析算法如下，在算法中使用了一个符号栈 S，用来存放终结符和非终结符，k 代表符号栈 S 的使用深度：

```
k=1; S[k]='#';
do{
    把下一个输入符号读进 a 中;
    if(S[k]∈VT)  j=k;
      //任何两终结符之间最多只有一非终结符，故若 S[k]∉VT 则 S[k−1]∈VT
    else  j=k−1;
    while(S[j]>a)
    {
      do{                          //找出最左子串 S[j]<S[j+1]…S[k]>a
          Q=S[j];
          if(S[j−1]∈VT)  j=j−1;
          else  j=j−2;
        }while(S[j]≠Q);
      把 S[j+1]…S[k]归约为某个 N;
      k=j+1;
      S[k]=N;                      //将归约后的非终结符 N 置于原 S[j+1]位置
    }
    if(S[j]<a||S[j]≠a)             //如果栈顶 S[j]<a 或 S[j]≠a 则将 a 压栈
    {
      k=k+1;
      S[k]=a;
    }
    else error();
}while(a!='#');
```

此算法工作过程中若出现 j 减 1 后其值小于或等于 0，则意味着输入串有错。在正确的情况下，算法工作完毕时符号栈将呈现#S#。

例 3.14 已知文法 G[E]和优先关系如例 3.13 所示，试给出输入串 i+i*i 的算符优先分析过程。

[解答] 输入串 i+i*i 的分析过程如表 3.7 所示。

表 3.7　i+i*i 算符优先分析过程

符号栈	输入串	动　　作
#	i+i*i#	#⋖i
#i	+i*i#	#⋖i⋗+ 用 F→i 归约
#F	+i*i#	#⋖+
#F+	i*i#	#⋖+⋖i
#F+i	*i#	…+⋖i⋗* 用 F→i 归约
#F+F	*i#	#⋖+⋖*
#F+F*	i#	#⋖+⋖*⋖i
#F+F*i	#	…*⋖i⋗# 用 F→i 归约
#F+F*F	#	…+⋖*⋗# 用 T→T*F 归约
#F+T	#	#⋖+⋗# 用 E→E+T 归约
#E	#	#E# 结束

输入串 i+i*i 的归约过程用语法树表示如图 3-23 所示。实际上，先画出输入串 i+i*i 的语法树，然后再根据语法树得到表 3.7 的分析过程将更加容易。

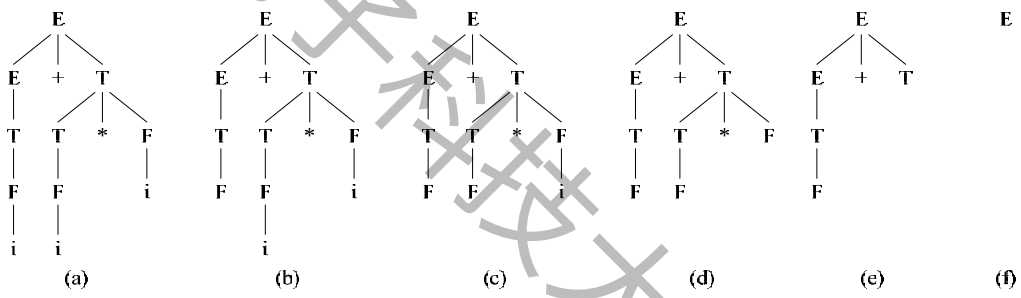

图 3-23　输入串 i+i*i 的语法树归约示意

由例 3.14 可知，算符优先分析的归约只关心句型中由左至右终结符序列的优先关系，而不涉及终结符之间可能存在的非终结符，即实际上可认为这些非终结符是同一个非终结符。采用例 3.13 中算符优先分析方法得到句子 i+i 的归约语法树如图 3-24 所示，先将第一个最左素短语 i 归为 F，然后把第二次归约的最左素短语 i (第二个 i)也归为 F，最后将第三次归约的最左素短语 F+F 归为 E，即认为 F+F 相当于 E+T。而对规范归约来说，句子 i+i 的归约过程是：先把第一个 i 归为 F，接着将 F 归为 T，再将 T 归为 E；然后重复相同的过程把第二个 i 归为 F，再将 F 归为 T；最后将 E+T 归为 E，如图 3-25 所示。

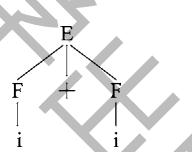

图 3-24　算符优先归约时 i+i 的语法树

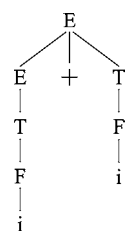

图 3-25　规范归约时 i+i 的语法树

因此，算符优先分析比规范归约要快得多，因为算符优先分析不考虑非终结符的形式(即认为所有非终结符都是一样的)，即跳过了所有形如 P→Q 的单非产生式(即右部仅含一个非终结符的产生式)所对应的归约步骤。这既是算符优先分析的优点，同时也是它的缺点；因为这样有可能把本来不成句子的输入串也误认为是句子，但这种缺点易于弥补。

算符优先分析法除了可用于分析算术表达式外，也可用于分析某些高级语言的语句。

例 3.15 试设计下面文法的算符优先分析表：

G[S]： S→iBtS｜iBtAeS｜a
A→iBtAeS｜a
B→b

[解答] 首先对文法 G[S]构造 FIRSTVT 和 LASTVT 集如下：

FIRSTVT(S)={i,a}；
FIRSTVT(A)={i,a}；
FIRSTVT(B)={b}；
LASTVT(S)={t,e,a}；
LASTVT(A)={e,a}；
LASTVT(B)={b}。

此外，由 A→…S 可知 LASTVT(S)⊂LASTVT(A)，即 LASTVT(A)={t,e,a}。优先关系如下：

(1) 由文法 G[S]的产生式 S→iBtAeS 和 S→iBtS 可知：

① 由 S→iB…得 i⋖FIRSTVT(B)，即 i⋖b；
② 由 S→…tS 得 t⋖FIRSTVT(S)，即 t⋖i，t⋖a；
③ 由 S→…Bt…得 LASTVT(B)⋗t，即 b⋗t；
④ 由 S→…tA…得 t⋖FIRSTVT(A)，即 t⋖i，t⋖a；
⑤ 由 S→…Ae…得 LASTVT(A)⋗e，即 e⋗e，a⋗e，t⋗e；
⑥ 由 S→…eS 得 e⋖FIRSTVT(S)，即 e⋖i，e⋖a；
⑦ 由 S→…iBt…得 i⋍t；
⑧ 由 S→…tAe…得 t⋍e。

(2) 由于存在 t⋗e 和 t⋍e(二义文法构造优先关系表必定含有冲突项)，根据 iBtAeS 知此时应将 iBtAeS 同时归约为 S 或 A，所以应选用 t⋍e 而舍弃 t⋗e。

最后得到的优先关系表见表 3.8。

表 3.8 例 3.15 的优先关系表

	i	t	e	a	b
i		⋍			⋖
t	⋖		⋍	⋖	
e	⋖		⋗	⋖	
a			⋗		
b		⋗			

4．优先函数

用优先关系表来表示每对终结符之间的优先关系会导致存储量大、查找费时。如果给每个终结符赋一个值(即定义终结符的一个函数 f)，值的大小反映其优先关系，则终结符对 a、b 之间的优先关系就转换为两个优先函数 f(a)与 f(b)值的比较。使用优先函数有两个明显的好处：一是节省空间；二是便于进行比较运算。

一个终结符在栈中(左)与在输入串中(右)的优先值是不同的。例如，既存在着+⋗)又存在着)⋗+。因此，对一个终结符 a 而言，它应该有一个左优先数 f(a)和一个右优先数 g(a)，这样就定义了终结符的一对函数。

如果根据一个文法的算符优先关系表，使得文法中的每个终结符 a 和 b 满足下述条件：

(1) 如果存在 a⋍b，则有 f(a)＝g(b)；

(2) 如果存在 a⋗b，则有 f(a)＞g(b)；

(3) 如果存在 a⋖b，则有 f(a)＜g(b)。

则称 f 和 g 为优先函数。其中，f 称为入栈函数，g 称为比较函数。注意，对应一个优先关系表的优先函数 f 和 g 不是唯一的；只要存在一对，就存在无穷多对。也有许多优先关系表不存在对应的优先函数。例如，表 3.9 给出的优先关系表就不存在优先函数。

表 3.9　不存在优先函数的优先关系表

	a	b
a	⋍	⋗
b	⋍	⋍

在表 3.9 中，假定存在 f 和 g，则应有：

f(a)=g(a)；　f(a)＞g(b)

f(b)=g(a)；　f(b)=g(b)

这将导致如下矛盾：

f(a)＞g(b)=f(b)=g(a)=f(a)

根据优先关系表构造优先函数 f 和 g 的方法有两种：一种是关系图法(也称 Bell 方法)；另一种由定义直接构造(也称 Floyd 方法)。

(1) 用关系图法构造优先函数的步骤如下：

① 对所有终结符 a(包括“#”)，用有下脚标的 f_a、g_a 表示结点名，画出全部 n 个终结符所对应的 2n 个结点；

② 若 a⋗b 或 a⋍b，则画一条从 f_a 到 g_b 的有向边；若 a⋖b 或 a⋍b，则画一条从 g_b 到 f_a 的有向边；

③ 对每个结点都赋予一个数，此数等于从该结点出发所能到达的结点(包括出发结点自身在内)的个数，赋给 f_a 的数作为 f(a)，赋给 g_b 的数作为 g(b)；

④ 检查所构造出来的函数 f 和 g，看它们同原来的关系表是否有矛盾。如果没有矛盾，则 f 和 g 就是所要的优先函数；如果有矛盾，那么就不存在优先函数。

(2) 由定义直接构造优先函数时，对每个终结符 a (包括“#”)，令 f(a)=g(a)=1(也可以是其它整数)，则：

① 如果 a⋗b 而 f(a)≤g(b)，则令 f(a)= g(b)+1；

② 如果 a⋖b 而 f(a)≥g(b)，则令 g(b)= f(a)+1；

③ 如果 a⋗b 而 f(a)≠g(b)，则令 f(a)= g(b)= max{f(a), g(b)}；

④ 重复①～③步直到过程收敛。如果重复过程中有一个值大于 2n，则表明该算符优先文法不存在优先函数。

注意：重复 ① ～ ③ 步的操作是对算符优先关系表逐行扫描，并按 ① ～ ③ 步修改函数 f(a)、g(b) 的值。这是一个迭代过程，一直进行到优先函数的值不再变化时为止，或当函数值大于 2n 时为止(表明无优先函数)。

用关系图法构造优先函数仅适用于终结符不多的情况，而由定义直接构造优先函数可适用于任何情况，并且也易于在计算机上实现。此外需要说明的是，对于一般表达式文法，优先函数通常是存在的。

例 3.16 试用关系图法和直接定义法求出表 3.6 的优先关系表所对应的优先函数(不含“#”)。

[解答] (1) 用关系图法构造的关系图如图 3-26 所示。

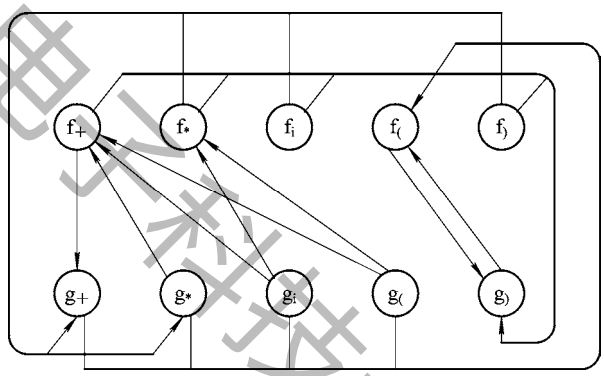

图 3-26 优先函数关系图

由图 3-26 所示的优先函数关系图求得优先函数如表 3.10 所示。

表 3.10 例 3.16 的优先函数表

	+	*	i	(	)
f	4	6	6	2	6
g	3	5	7	7	2

(2) 由定义直接计算出优先函数的过程如表 3.11 所示。

表 3.11 例 3.16 的优先函数计算过程

迭代次数	函数	+	*	i	(	)
0(初值)	f	1	1	1	1	1
	g	1	1	1	1	1
1	f	2	4	4	1	4
	g	2	3	5	5	1
2	f	3	5	5	1	5
	g	2	4	6	6	1
3	f	3	5	5	1	5
	g	2	4	6	6	1

表 3.11 最终迭代出来的优先函数的值与表 3.10 不同，这正说明了，如果优先函数存在，就可以有任意多个。

3.5　规范归约的自底向上语法分析方法

LR 分析法是一种自底向上进行规范归约的语法分析方法，LR 指“自左向右扫描和自底向上进行归约”。LR 分析法比递归下降分析法、LL(1)分析法和算符优先分析法对文法的限制要少得多，对大多数用无二义的上下文无关文法描述的语言都可以用 LR 分析器予以识别，而且速度快，并能准确、及时地指出输入串的任何语法错误及出错位置。LR 分析法的一个主要缺点是，若用手工构造分析器则工作量相当大，因此必须求助于自动产生 LR 分析器的产生器。

3.5.1　LR 分析器的工作原理

在第二章词法分析中，DFA(确定有限状态自动机)是根据正规表达式来识别字符串的；也即，词法分析中文法的作用就是描述字符串的规则。语法分析与词法分析相比则有较大的不同，这是因为语法分析中的文法含有非终结符，且非终结符在文法中的主要作用是用来表示嵌套和递归，以便形成比字符串更为复杂的句子。下面，我们通过两个产生式来阐述 LR 分析器的基本原理。

$$A \rightarrow aBc$$

$$B \rightarrow d \mid ef$$

为了识别非终结符 A，就要识别符号串 aBc。为此，构造一个 DFA 如图 3-27 所示。

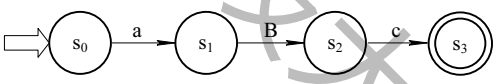

图 3-27　识别非终结符 A 的 DFA

当此 DFA 识别出符号串 aBc 后，非终结符 A 就被识别了。但是，与词法分析中 DFA 不同的是，这里的输入符号包含着非终结符 B，而语法分析中待分析的句子是不含非终结符的(句子全部由终结符组成)；因此，在图 3-27 中 DFA 必须从待分析的符号串中识别出非终结符 B 后方可到达状态 s_2。那么，如何识别非终结符 B 呢？我们可以为 B 的产生式右部构造一个 DFA(如图 3-28 所示)，这个 DFA 在识别了 B 的产生式右部后就可以为识别 A 的 DFA 提供一个非终结符 B 了。为了实现产生式的这种嵌套特性，我们把 DFA 也嵌套起来，即对图 3-27 和图 3-28，可以用 ε 弧将状态 s_1 和状态 s_4 连接起来。这样，就可以使 A 的 DFA 在识别字符 a 后自动进入到识别非终结符 B 的 NFA(如图 3-29 所示)。此时，状态 s_5 和状态 s_7 仅表示识别非终结符 B 的终态，而不再是识别 A 的整个 DFA 终态了；并且，非终结符 B 识别完后要自动返回到状态 s_1。此外要说明的是，状态 s_1 和状态 s_4 也可以合并为一个状态 s_1'(如图 3-30 所示)。

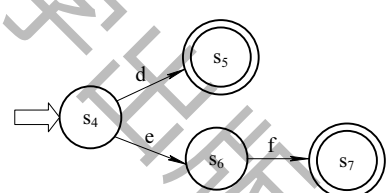

图 3-28　识别非终结符 B 的 DFA

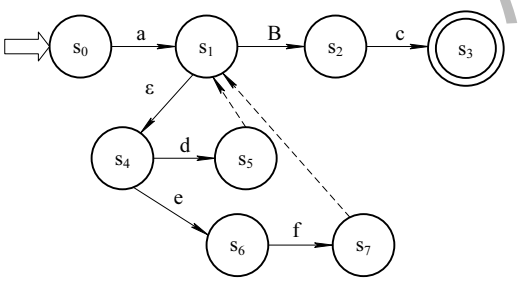

图 3-29　识别 A 的嵌套 NFA

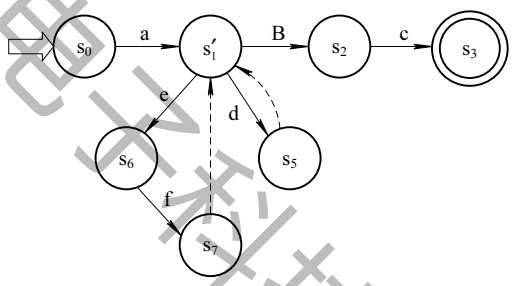

图 3-30　识别 A 的嵌套 DFA

对图 3-30，DFA 到达 s_5 或 s_7 时，表示在输入的符号串中有构成非终结符 B 的子串，但整个句子(符号串)还未分析完。这时，DFA 将回到状态 s_1' 并以刚识别(归约)出的 B 作为输入符号，使状态转换到 s_2；也即，由状态 s_1' 经过状态 s_5 或经过状态 s_6、s_7 再回到状态 s_1' 所识别的字符串为对应非终结符 B 的一个句子或句型，它等价于由状态 s_1' 识别字符 B 到达状态 s_2，故此时可用 B 来取代该句子或句型，而用 s_2 来取代 s_5 或 s_6、s_7，这意味着已经将该句子或句型归约为 B，并且已经读入了 B 并到达状态 s_2。接下来 DFA 在输入字符 c 后到达终态 s_3，从而识别出非终结符 A。如果以本例中的两个产生式来构成一个文法的话，则这个文法所定义的语言只有两个句子，即 adc 和 aefc，而图 3-30 的 DFA 恰好能够识别这两个句子。

如果用栈来记录图 3-30 DFA 输入句子(仅由终结符组成)和生成非终结符 A 的过程，我们就会发现：DFA 的输入过程与移进过程对应，而生成非终结符的过程则与归约过程对应。表 3.12 依据图 3-30 将句子 aefc 的“移进-归约”分析与 DFA 的状态转换过程对应起来。

表 3.12　句子 aefc 的“移进-归约”与 DFA 的状态转换

符号栈	输入符号串	移进-归约动作	DFA 状态转换
#	aefc#	移进	s_0 转到 s_1'
#a	efc#	移进	s_1' 转到 s_6
#ae	fc#	移进	s_6 转到 s_7
#aef	c#	用 B→ef 归约	s_7 回退到 s_1' 再根据归约后的 B 转到 s_2
#aB	c#	移进	s_2 转到 s_3
#aBc	#	用 A→aBc 归约	s_3 是接受状态
#A	#	分析成功	结束

从表面看，符号栈的“移进-归约”分析与 DFA 状态转换的句子识别是分别进行的，但二者之间却存在着密切联系。实际上，符号栈里实时记录着 DFA 的状态转换路径。符号栈里的每一个符号都对应着状态转换图中的一条有向边；这些有向边首尾相连，起始于 DFA 的初态 s_0，暂停于 DFA 的当前扫描状态 s_i(即表 3.12 中任何一行符号栈中的符号序列都对应图 3-30 中由 s_0 开始的有向边序列，且该有向边序列终止于 DFA 当前扫描的状态 s_i)。既然符号栈中的符号串能够与 DFA 的状态转换相对应，那么就可以用一个状态序列来替代符号栈的符号串：因为状态序列与符号栈中的符号串作用相同，都记录着 DFA 在识别句子过程中的状态转换路径。这样，符号栈就变成了状态栈，而 DFA 的状态转换可以用栈顶的状态与当前扫描的输入符号(两者一起构成一个状态转换函数 GO)来决定。由此，对符号的“移进”就变成了对状态的“移进”；对符号串的“归约”就变成了对状态序列的替换。

根据上述思想，我们来构造 LR 分析器。首先，LR 分析器是一个 DFA，它应该有一个

称为 GOTO 表的状态转换矩阵，GOTO[s,x]规定了状态栈栈顶 s 在输入符号为 x 时所应转换的下一状态是什么；其次，LR 分析器还必须完成“移进-归约”操作，因此它还应有一个称之为 ACTION 表的操作动作表，且每一项 ACTION[s,a]所规定的动作是以下四种情况之一：

(1) 移进：使栈顶状态 s 与当前扫描的输入符号 a (终结符)的下一状态 s'=ACTION[s,a]和输入符号 a 进栈，而下一个输入符号则变成当前扫描的输入符号。事实上，由于存在 s'=ACTION[s,a]，则 s'=GOTO[s,a]就可以不要了，因为它们二者的含义是一样的，即移进完全由 ACTION 表完成。这样，就使 GOTO 表省去了输入符号为终结符的那些表项，而仅保留输入符号为非终结符的那些表项。

(2) 归约：如果符号栈栈顶的符号串为 α(自栈顶向下则为 α 的逆序)且文法中存在 A→α，则将栈顶的符号串 α 用非终结符 A 替换，即实现将 α 归约为 A。对状态栈来说，假定 α 中有 γ 个符号(即 α 的长度为 γ)，则状态栈栈顶的 γ 个状态序列恰好能识别符号串 α，此时可用产生式 A→α 进行归约。由图 3-30 和表 3.12 可知，归约的动作是去掉栈顶的 γ 个状态项，假定原来 s_m 为栈顶状态，则此时状态 $s_{m-\gamma}$ 成为新的栈顶状态，然后使 $s_{m-\gamma}$ 与所归约的非终结符 A 的下一状态 s'=GOTO[$s_{m-\gamma}$,A]和 A 分别进入状态栈和符号栈。归约的动作不改变当前扫描的输入符号。由图 3-30 和表 3.12 可知；符号栈栈顶出现符号串 fe，状态栈栈顶出现 s_7s_6 时，fe 的逆序即为 B 的句柄(B→ef)。此时，应去掉符号栈栈顶的符号串 fe 以及去掉状态栈栈顶的状态 s_7s_6，使得状态 s_1'成为新的状态栈栈顶，并将归约后的 B 以及由 s_1'经过有向边 B 所达到的下一状态 s_2 分别压入符号栈和状态栈(表示已经识别了非终结符 B)。

(3) 接受：分析工作成功，表明所分析的句子为文法所识别，此时分析器停止工作。

(4) 报错：发现所分析的句子不是文法所允许的句子，调用出错处理程序进行处理。

我们知道，规范归约(最左归约，即最右推导的逆过程)的关键问题是寻找句柄。因此，LR 分析法的基本思想是：在规范归约过程中，一方面记住已移进和归约出的整个符号串，即记住“历史”；另一方面根据所用的产生式推测未来可能遇到的输入符号，即对未来进行“展望”。当一串貌似句柄的符号串呈现于分析栈(即状态栈和符号栈)的顶端时，LR 分析器能够根据所记载的“历史”和“展望”以及“现实”的输入符号等三方面的材料，来确定栈顶的符号是否构成相对某一产生式的句柄。

综上所述，一个 LR 分析器的结构可以用图 3-31 表示，它由分析栈、分析表和总控程序这三个部分组成，而 LR 分析表是 LR 分析器的核心。

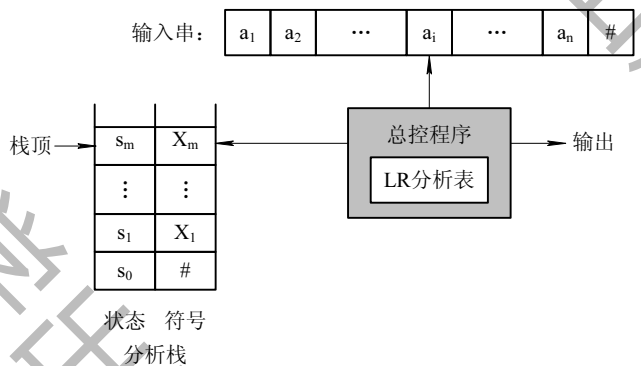

图 3-31　LR 分析器结构示意

注意：所有 LR 分析器的总控程序都是一样的，只是分析表各有不同。因此，构造 LR 分析器的主要任务就是产生分析表。

因此，LR 分析器实质上是一个带分析栈的 DFA。在这个 DFA 中，我们将把“历史”和“展望”材料综合抽象成某些“状态”，而分析栈则用来存放这些状态。栈里的每个状态概括了从分析开始直到某一归约阶段的全部“历史”和“展望”资料。任何时候，栈顶的状态都代表了整个“历史”和已推测出的“展望”。LR 分析器的每一步工作都是由栈顶状态和现行输入符号所唯一决定的。为了有助于明确归约手续，我们把已归约出的文法符号串也同时放在栈中(实际上可不必进栈)。栈的每一项内容包括状态 s 和文法符号 X 两部分(见图 3-31)。$(s_0,\#)$ 为分析开始前预先放入栈里的初始状态和句子括号；栈顶状态为 s_m，符号串 $X_1X_2\cdots X_m$ 是至今已移进归约出的文法符号串。而 LR 分析器的总控程序工作的任何一步只需按分析栈的栈顶状态 s_m 和当前扫描到的输入符号 a_i 执行 ACTION$[s_m,a_i]$所规定的动作即可(GOTO 表实质上只用于 ACTION 表执行归约后的处理，即对归约后的非终结符进行状态转换)。

例如，表达式文法 G[E]如下，它对应的 LR 分析表见表 3.13，则语句 i+i*i 的 LR 分析过程如表 3.14 所示：

G[E]：(1) E→E+T
(2) E→T
(3) T→T*F
(4) T→F
(5) F→(E)
(6) F→i

表 3.13 LR 分析表

状态	ACTION						GOTO		
	i	+	*	(	)	#	E	T	F
0	s_5			s_4			1	2	3
1		s_6				acc			
2		r_2	s_7		r_2	r_2			
3		r_4	r_4		r_4	r_4			
4	s_5			s_4			8	2	3
5		r_6	r_6		r_6	r_6			
6	s_5			s_4				9	3
7	s_5			s_4					10
8		s_6			s_{11}				
9		r_1	s_7		r_1	r_1			
10		r_3	r_3		r_3	r_3			
11		r_5	r_5		r_5	r_5			

其中，s_j 指把下一状态 j 和现行输入符号 a 移进栈；r_j 指按文法的第 j 个产生式进行归约；acc 表示分析成功；空白格为出错。

表 3.14　i+i*i 的 LR 分析过程

步骤	状态栈	符号栈	输入串	动 作 说 明
1	0	#	i+i*i#	ACTION[0, i]=s_5，即状态 5 入栈
2	0 5	# i	+i*i#	r_6: 用 F→i 归约且 GOTO(0, F)=3 入栈
3	0 3	# F	+i*i#	r_4: 用 T→F 归约且 GOTO(0, T)=2 入栈
4	0 2	# T	+i*i#	r_2: 用 E→T 归约且 GOTO(0, E)=1 入栈
5	0 1	# E	+i*i#	ACTION[1, +]=s_6，即状态 6 入栈
6	0 1 6	# E+	i*i#	ACTION[6, i]=s_5，即状态 5 入栈
7	0 1 6 5	# E+i	*i#	r_6: 用 F→i 归约且 GOTO(6, F)=3 入栈
8	0 1 6 3	# E+F	*i#	r_4: 用 T→F 归约且 GOTO(6, T)=9 入栈
9	0 1 6 9	# E+T	*i#	ACTION[9, *]=s_7，即状态 7 入栈
10	0 1 6 9 7	# E+T*	i#	ACTION[7, i]=s_5，即状态 5 入栈
11	0 1 6 9 7 5	# E+T*i	#	r_6: 用 F→i 归约且 GOTO(7, F)=10 入栈
12	0 1 6 9 7 10	# E+T*F	#	r_3: 用 T→T*F 归约且 GOTO(6, T)=9 入栈
13	0 1 6 9	# E+T	#	r_1: 用 E→E+T 归约且 GOTO(0, E)=1 入栈
14	0 1	# E	#	acc: 分析成功

输入串 i+i*i 的归约过程用语法树表示如图 3-32 所示。从表 3.14 的分析过程可以看出，每次归约恰好是图 3-32 语法树中的句柄，这种归约过程实际上就是修剪语法树的过程，直到归约到树根(也即文法开始符 E)为止。因此，LR 分析法解决了在语法分析过程中寻找每一次归约的句柄问题。

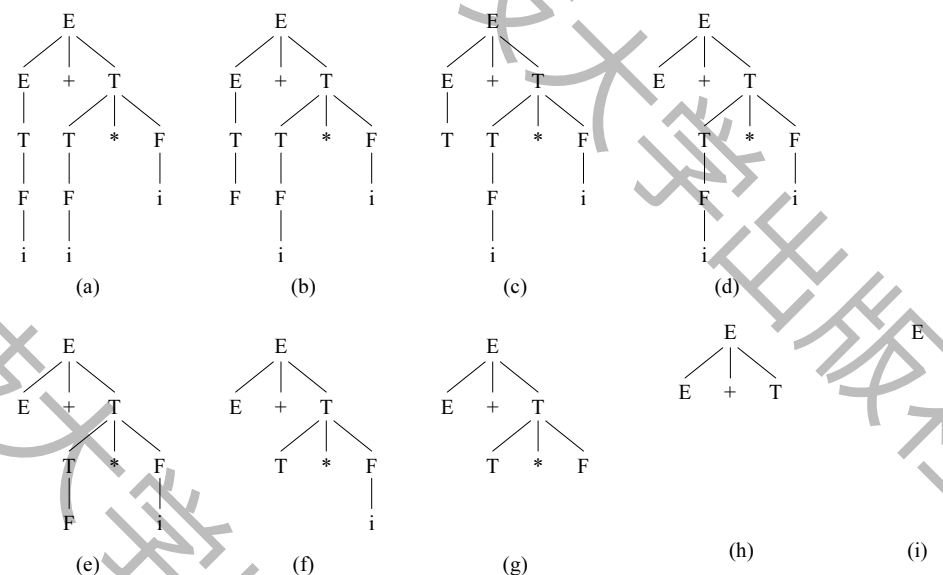

图 3-32　输入串 i+i*i 的归约示意

我们主要关心的问题是如何由文法构造 LR 分析表。对于一个文法，如果能够构造一张分析表，使得它的每个入口均是唯一确定的，则称这个文法为 LR 文法。对于一个 LR

文法，当分析器对输入串进行自左至右扫描时，一旦句柄呈现于栈顶，就能及时对它实行归约。

在有些情况下，LR 分析器需要“展望”和实际检查未来的 k 个输入符号才能决定应采取什么样的“移进-归约”决策。一般而言，一个文法如果能用一个每步最多向前检查 k 个输入符号的 LR 分析器进行分析，则这个文法就称为 LR(k)文法。

对于一个文法，如果它的任何“移进-归约”分析器都存在这样的情况：尽管栈的内容和下一个输入符号都已了解，但仍无法确定是“移进”还是“归约”，或者无法从几种可能的归约中确定其一，则该文法是非 LR 的。注意，LR 文法肯定是无二义的，一个二义文法绝不会是 LR 文法；但是，LR 分析技术可以进行适当修改以适用于分析一定的二义文法。

我们在后面将介绍四种分析表的构造方法，它们是：

(1) LR(0)表构造法，这种方法局限性很大，但它是建立一般 LR 分析表的基础；

(2) SLR(1)表(即简单 LR 表)构造法，这种方法较易实现又极有使用价值；

(3) LR(1)表(即规范 LR 表)构造法，这种表适用大多数上下文无关文法，但分析表体积庞大；

(4) LALR(1)表(即向前 LR 表)构造法，该表能力介于 SLR(1)和 LR(1)之间。

3.5.2 LR(0)分析器

我们希望仅由一种只概括“历史”资料而不包含推测性“展望”材料的简单状态就能识别呈现在栈顶的某些句柄，而 LR(0)项目集就是这样一种简单状态。

在讨论 LR 分析法时，需要定义一个重要概念，这就是文法规范句型的“活前缀”。字的前缀是指该字的任意首部，例如字 abc 的前缀有 ε、a、ab 或 abc。所谓活前缀，是指规范句型的一个前缀，这种前缀不含句柄之后的任何符号。在 LR 分析工作过程中的任何时候，栈里的文法符号(自栈底而上)$X_1X_2\cdots X_m$ 应该构成活前缀，把输入串的剩余部分匹配于其后即应成为规范句型(如果整个输入串确为一个句子的话)。例如，表 3.14 中符号栈就给出了对输入串 i+i*i 扫描过程中每一步的活前缀，而将表 3.14 符号栈中每行的活前缀与该行后面的输入串连接起来，就构成了一个规范句型。因此，只要输入串的已扫描部分保持可归约成一个活前缀，就意味着所扫描的部分没有错误。

1. LR(0)项目集规范族的构造

对于一个文法 G[S]，首先要构造一个 NFA，它能识别 G[S]的所有活前缀。这个 NFA 的每个状态就是一个“项目”。文法 G[S]中每一个产生式的右部添加一个圆点“•”，称为 G[S]的一个 LR(0)项目(简称项目)。例如，产生式 A→aBc 对应有四个项目：

A→•aBc

A→a•Bc

A→aB•c

A→aBc•

注意，如果产生式的右部有 n 个符号，则该产生式就有 n + 1 个项目；并且，产生式 A→ε 只对应一个项目 A→•。

一个项目指明了在分析过程的某个时刻我们能看到产生式的多大一部分。圆点“•”指出了分析过程中扫描输入串的当前位置；圆点“•”前的部分为已经扫描过的符号串，圆点

“·”后的部分为待扫描的符号串，且圆点“·”前的符号串构成了一个活前缀。对于项目 A→·aBc 来说，表示符号串 aBc 还未扫描；而对项目 A→aBc·来说，则已经扫描完符号串 aBc(形成了可归约为 A 的句柄)，此时可以将 aBc 归约为 A 了。由于产生式的项目与识别 NFA 的状态相对应，因此可以用项目来构造 NFA。产生式 A→aBc 所对应的 NFA 如图 3-33 所示。

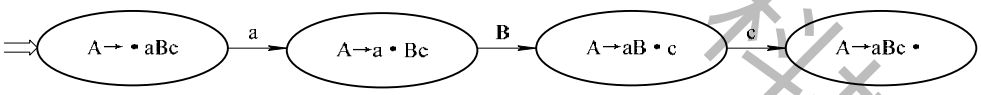

图 3-33　产生式 A→aBc 所对应的 NFA M

显然，这个 NFA M 能够识别产生式 A→aBc 的右部 aBc。圆点“·”右侧第一个符号即为 NFA 相应状态出发的有向边上的符号，它表示该状态能够识别的符号。

注意，凡圆点在最右端的项目，如 A→α·，称为一个“归约”项目；对文法的开始符号 S'的归约项目(S'的含义见下面的描述)，如 S'→α·称为“接受”项目；形如 A→α·aβ 的项目(圆点“·”右侧第一个符号是终结符)称为“移进”项目；形如 A→α·Bβ 的项目(圆点“·”右侧第一个符号是非终结符)称为“待约”项目，因为在 NFA 中发生状态转换的非终结符不可能由输入串(由终结符组成)提供，而必须由 NFA 的归约来产生，这就要启动另一路 NFA 来识别(归约出)所需要的非终结符。例如，在图 3-33 中若非终结符 B 的产生式为 B→ef，则在含项目 A→a·Bc 的状态中就应加入项目 B→·ef，称为 A 的闭包项。也即，要获得非终结符 B，就必须先识别 B 的产生式，这就意味着在识别 A 的 NFA M 中嵌入了识别 B 的 NFA M(如图 3-34 所示，图中 I_0～I_5 为状态编号)。

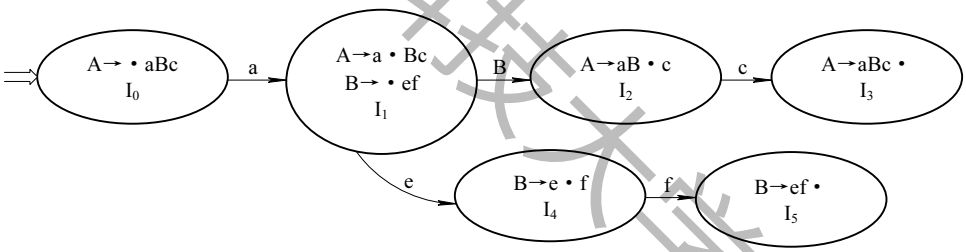

图 3-34　在识别 A 的 NFA 中嵌入识别 B 的 NFA M

如果闭包项 B→·ef 中“·”后的第一个符号是另一个非终结符 R，即闭包项为 B→·Rf 的话，则 B→·Rf 仍然是一个待约项目，也即在图 3-34 的状态 I_1 中还应加入识别 R 的闭包项；如此下去，不断在同一个状态中加入闭包项直至没有新的“待约项”出现为止。这样，就把识别活前缀的 NFA M 确定化，形成了一个识别产生式的 DFA M′(这种闭包项的方法等同于第二章的 ε_CLOSURE(闭包)方法)，此时的状态也可能不再是含有一个项目，而是含有多个项目，我们称其为项目集。

将上述方法拓展到整个文法，即可以使用这些项目状态构造一个 NFA M 来识别文法的所有活前缀；使用闭包项方法，就能够把识别活前缀的 NFA M 确定化，使之成为一个以项目集为状态的 DFA M，这个 DFA M 就是建立 LR 分析算法的基础。构成识别一个文法活前缀的 DFA M 的项目集(状态)的全体称为这个文法的 LR(0)项目集规范族，这个规范族提供了建立一类 LR(0)和 SLR(1)分析器的基础。

我们用第二章所引进的 ε_CLOSURE 来构造一个文法 G[S]的 LR(0)项目集规范族。假

定 I 是文法 G[S]的任一项目集，则定义和构造 I 的闭包 CLOSURE(I)的方法是：

(1) I 的任何项目都属于 CLOSURE(I)；

(2) 若 A→α•Bβ 属于 CLOSURE(I)，那么对任何关于 B 的产生式 B→γ，其项目 B→ •γ 也属于 CLOSURE(I)(设 A→α•Bβ 的状态为 i，则 i 到所有含 B→•γ 的状态都有一条 ε 有向边，即此规则仍与第二章的 ε_CLOSURE(I)定义一样)；

(3) 重复执行上述(1)～(2)步直至 CLOSURE(I)不再增大为止。

在构造 CLOSURE(I)时请注意一个重要的事实，那就是对任何非终结符 B，若某个圆点“•”在左边的项目 B→•γ 进入到 CLOSURE(I)，则 B 的所有其它圆点“•”在左边的项目 B→•β 也将进入同一个 CLOSURE 集。

此外，我们设函数 GO 为状态转换函数，GO(I,X)的第一个变元 I 是一个项目集，第二个变元 X 是一个文法符号(终结符或非终结符)，函数 GO(I,X)定义为

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

其中，如果 A→α•Xβ 属于 I，则 J={任何形如 A→αX•β 的项目}。如果由项目集 I 发出的字符为 X 的有向边，则到达的状态即为 CLOSURE(J)(这也类同于第二章 I_a= ε_CLOSURE(J)的定义，但这里相当于输入的字符是 X)。直观上说，若 I 是对某个活前缀 γ 有效的项目集(状态)，则 GO(I,X)就是对 γX 有效的项目集(状态)。

通过函数 CLOSURE 和 GO 很容易构造一个文法 G[S]的拓广文法 G'[S']的 LR(0)项目集规范族。如果已经求出了 I 的闭包 CLOSURE(I)，则用状态转换函数 GO 可以求出由项目集 I 到另一项目集状态必须满足的字符(即转换图有向边上的字符)；然后，再求出有向边到达的状态所含的项目集，即用 GO(I,X)=CLOSURE(J)求出 J，再对 J 求其闭包 CLOSURE(J)，也就是有向边到达状态所含的项目集。以此类推，最终构造出拓广文法 G'[S']的 LR(0)项目集规范族。

2．LR(0)分析表的构造

为了构造 LR(0)分析表，就必须对文法进行改造。我们察看文法 G[S]：S→aS｜bc，句子 abc 的语法树见图 3-35(a)。当把句柄 bc 归约为 S 时，由语法树可以看出该 S 不是树根；但在图 3-35(b)中，句柄 bc 归约为 S 时该 S 为树根。由机器进行语法分析时无法判断归约的 S 是否为树根，即是否到达分析成功的“接受”状态。为了使“接受”状态易于识别，总是将文法 G[S]进行拓广。假定文法 G[S]以 S 为开始符号，我们构造一个 G'[S']，它包含了整个 G[S]并引进了一个不出现在 G[S]中的非终结符 S'，同时加进了一个新产生式 S'→S，这个 S'是 G'[S']的开始符号，称 G'[S']是 G[S]的拓广文法，并且会有一个仅含项目 S'→S•的状态，这就是唯一的“接受”态。如在图 3-35(c)、(d)中，当把句柄 bc 归约为 S 时，这个 S 一定不是树根，仅当把 S 归约为 S'时，这个 S'才是唯一的树根，也是我们进行语法分析的“接受”状态。

假若一个文法 G[S]的拓广文法 G'[S']的活前缀识别自动机(DFA)中的每个状态(项目集)不存在下述情况：既含移进项目又含归约项目或者含有多个归约项目，则称 G[S]是一个 LR(0)文法。换言之，LR(0)文法规范族的每个项目集不包含任何冲突项目。

对于 LR(0)文法，我们可直接从它的项目集规范族 C 和活前缀自动机的状态转换函数 GO 构造出 LR 分析表。下面是构造 LR(0)分析表的算法。

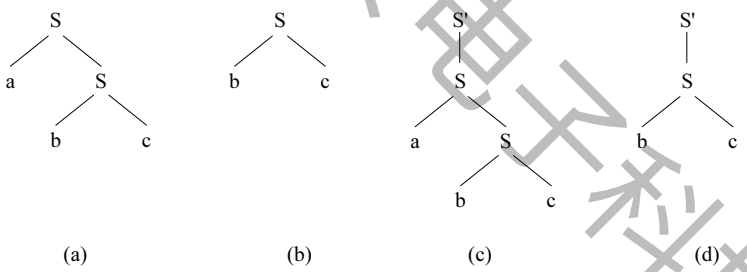

图 3-35　句柄 bc 归约情况分析

假定 $C = \{I_0, I_1, \cdots, I_n\}$。由于我们已经习惯用数字表示状态，因此令每个项目集 I_k 的下标 k 作为分析器的状态，特别地，令包含项目 S'→ • S(表示整个句子还未输入)的集合 I_k 的下标 k 为分析器的初态。分析表的动作子表 ACTION 和状态转换子表 GOTO 可按如下方法构造：

(1) 若项目 A→α • aβ 属于 I_k 且 $GO(I_k, a)=I_j$，a 为终结符，则置 ACTION[k, a] 为“将(j, a)移进栈”，简记为“s_j”。

(2) 若项目 A→α • 属于 I_k，则对任何终结符 a (包括结束符#)，置 ACTION[k, a] 为“用产生式 A→α 进行归约”，简记为“r_j”(注意：j 是产生式的编号，而不是项目集的状态号，即 A→α 是文法 G'[S']的第 j 个产生式)。

(3) 若项目 S'→S • 属于 I_k(S • 表示整个句子已输入且已分析归约结束)，则置 ACTION[k, #] 为“接受”，简记为“acc”。

(4) 若项目 A→α • Bβ 属于 I_k 且 $GO(I_k, B)=I_j$，B 为非终结符，则置 GOTO[k, B] = j，即意为将(j, B)移进栈。

(5) 分析表中凡不能用规则(1)～(4)填入的空白格均置为“出错标志”。

由于假定 LR(0) 文法规范族的每个项目集不含冲突项目，因此按上述方法构造的分析表的每个入口都是唯一的(即不含多重定义)。我们称如此构造的分析表是一张 LR(0) 表，使用 LR(0) 表的分析器叫做一个 LR(0) 分析器。

对于(2)的说明如下，假定有产生式 A→abc 且句子序列为…abcd…，当前刚扫描完字符串 abc 且扫描指针指向字符 d(此时 d 为当前输入符号)。由于刚扫描过的 abc 形成了一个句柄，这时应将 abc 归约为非终结符 A。那么，当面临什么输入符号时可以将 abc 归约为 A 呢？由于 LR 分析法采用的是规范归约，即在句柄之后是不会出现非终结符的。为了简单起见，LR(0) 采取的办法是句柄遇见文法的任何终结符(包括终结符#)都进行归约，而不管该终结符是否真正会在句柄之后出现。这种情况反映在 LR(0) 分析表中，就是在 ACTION 表的状态 k(设归约项目 A→abc • 属于项目集 I_k)这一行全部填满产生式 A→abc 的归约编号 r_j，这是因为 ACTION 表恰好对应文法的全部终结符(包括终结符#)，而 GOTO 表因只对应文法的非终结符故其状态 k 这一行无需填入 r_j。

对于(3)，当把 S 归约为 S'时则表明整个句子分析成功，此时扫描到的当前输入符号只能是语句结束符#，否则出错。因此，反映在 ACTION 表中，就是在状态 k (设 S'→S • 属于 I_k)这一行对应终结符#的栏目 ACTION[k, #] 被置为 acc。

例 3.17　已知文法 G[S] 如下，试构造该文法的 LR(0) 分析表：

G[S]：S→BB

B→aB｜b

[解答] 将文法 G[S]拓广为文法 G'[S']：

G'[S']：(0) S'→S

(1) S→BB

(2) B→aB

(3) B→b

列出 LR(0)的所有项目：

1．S'→·S　　5．S→BB·　　9．B→·b

2．S'→S·　　6．B→·aB　　10．B→b·

3．S→·BB　　7．B→a·B

4．S→B·B　　8．B→aB·

用 ε_CLOSURE 办法构造文法 G'[S']的 LR(0)项目集规范族如下：

I_0：S'→·S　　I_1：S'→S·　　I_3：B→a·B　　I_5：S→BB·

S→·BB　　I_2：S→B·B　　B→·aB　　I_6：B→aB·

B→·aB　　B→·aB　　B→·b

B→·b　　B→·b　　I_4：B→b·

根据状态转换函数 GO 构造出文法 G'[S']的 DFA，如图 3-36 所示。

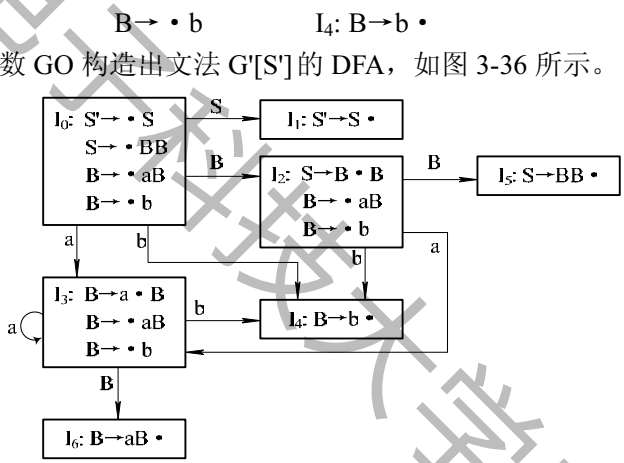

图 3-36　例 3.17 文法 G'[S']的 DFA M(即 LR(0)项目集和 GO 函数)

也可以用项目集规范族和图 3-37 所示的状态转换图来共同表示 G'[S']的 DFA M。

项目集的构造原则如下：

(1) 对一个项目集中的某个项目，只要在“·”之后的第一个符号是非终结符，则该非终结符所有以“·”开头的项目全都纳入到该项目集；如果这些新纳入的项目中，又在“·”后紧随出现新的非终结符，则这些新的非终结符所有以“·”开始的项目也全部纳入该项目集。如图 3-36 中的项目集 I_0，在 S'→·S 中的“·”后出现了非终结符 S，故 S→·BB 纳入

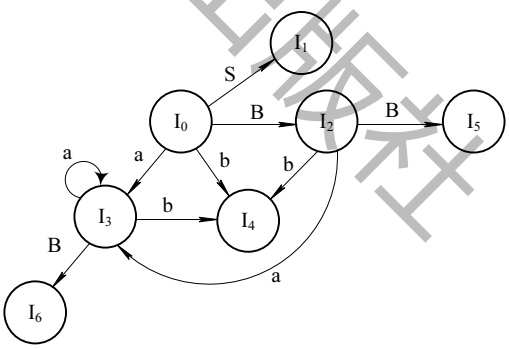

图 3-37　G'[S']的状态转换图(即 G'[S']的 DFA M)

到 I_0；而 S→•BB 的“•”后第一个符号又出现了新的非终结符 B，则相应地 B→•aB 和 B→•b 也都纳入到 I_0。

(2) 对一个项目集来说，除了归约项目之外，对于其余移进项目，“•”之后有多少个不同的首字符(包括非终结符)，就要引出多少条有向边到不同的项目集(也可能有一条有向边到此项目集自身)，这是检查所构造的 DFA M 是否完整的一种方法。

(3) 在项目集中根据某一项目“•”后的首字符，引出一有向边到达另一项目集，实际上是将形如 A→α•aβ 或 A→α•Bβ 的项目读入字符 a 或字符 B 并同时引出一有向边到达另一项目集。而所到达的项目集中应含有读入字符 a 或字符 B 后的新项目 A→αa•β 或者 A→αB•β，这时要分两种情况考虑：一种是项目 A→αa•β 或 A→αB•β 在目前已存在的所有项目集中均未出现，则引出的有向边到达一新产生的项目集，该项目集纳入新项目 A→αa•β 或 A→αB•β；另一种是项目 A→αa•β 或 A→αB•β 在目前已存在的所有项目集中的某一个已经出现，则不产生新的项目集，引出的有向边到达这个含有项目 A→αa•β 或 A→αB•β 的项目集。

掌握了上述构造原则和方法后，则可在写出拓广文法后就直接画出该文法的 DFA，而无需再列出 LR(0)的所有项目和构造 LR(0)项目集规范族了。

根据图 3-36 构造 LR(0)分析表的方法如下：由于存在 I_0～I_6 这 7 个项目集，因此每个不同的 I_i 都分别对应一行且将这些不同 I_i 的下标 i 依次标记在该行的第一列上以表示不同的状态行(注意，I_0 因含有 S'→ • S 而标记在第 0 行上状态)。此外，ACTION 表中相对于文法的每一个终结符(包括终结符#)都对应一列，而 GOTO 表则相对文法的每一个非终结符(除 S'外)都对应一列。对每一个 I_i，如果由 I_i 发出的有向边上标记的是终结符(所对应的项目属于移进项)且该有向边指向 I_k，则在 ACTION 表的第 i 行及该终结符这一列所对应的栏中填上“s_k”；如果有向边上标记的是非终结符(所对应的项目属于待约项)且该有向边指向 I_k，则在 GOTO 表的第 i 行及该非终结符这一列所对应的栏中填上“k”(注意，如果由 I_i 发出的有向边共有 n 条，则相应第 i 行的 ACTION 表和 GOTO 表填入内容的栏合计为 n 个，否则就有缺项)。如果 I_i 中含有形如“A→α•”这样的归约项目，并且拓广文法 G'[S']中 A→α 所标记的产生式序号为“j”，则 ACTION 表的第 i 行全部填入“r_j”；如果 I_i 是含有形如“S'→S•”这样的归约项目，则在第 i 行对应 ACTION 表终结符“#”这一列的栏中填入“acc”。

最后得到 LR(0)分析表见表 3.15。

表 3.15　例 3.17 的 LR(0) 分析表

状态	ACTION			GOTO	
	a	b	#	S	B
0	s_3	s_4		1	2
1			acc		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5	r_1	r_1	r_1		
6	r_2	r_2	r_2		

下面，通过图 3-36 的 DFA M 来分析对输入串 aB…的识别过程(事实上，句子对应的输入串应全部由终结符组成，在此的输入串是指扫描及归约过程中某时刻所形成的一种形态)。首先，由初始状态 I_0 开始识别第 1 个字符 a，即与 I_0 中的项目 B→•aB 对应，所以应执行的动作是“移进”，故由状态 I_0 识别字符 a 后到达 I_3，即将状态 3 和字符 a (即(3,a))压入分析栈。然后，继续在状态 I_3 下扫描第 2 个字符 B，因与 I_3 中的项目 B→a•B 对应，故应执行的动作仍是“移进”，即由状态 I_3 识别字符 B 后到达状态 I_6，此时将(6,B)压入分析栈。接下来在状态 I_6 下扫描后继的无论什么字符，都因状态 I_6 中仅有一个归约项目 B→aB•而将刚扫描过的两个字符 aB 归约为 B；由 3.5.1 节识别非终结符的图 3-29 可知，此时应由状态 I_6 回退到扫描两个字符 aB 之前的状态 I_0，因此这种回退就意味着去掉刚才因移进字符 a 和 B 所压入分析栈的两项(3,a)和(6,B)，即用 B 取代 aB，而用 I_2 取代 aB 对应的状态 I_3 和 I_6。注意，由于是将 aB 归约为 B，即对于已读入的字符串 aB 相当于此时在状态 I_0 读入这个归约后的字符 B，故由状态 I_0 识别这个字符 B 后到达状态 I_2，此时将(2,B)压入分析栈。当然，对输入串 aB…的识别过程也可以由表 3.15 按照表 3.14 来分析更加容易地得到。同时，我们也理解了为什么当用某一产生式 A→β 进行归约时，则归约的动作是先去掉分析栈栈顶的 γ 个项(假若 β 的长度为 γ)，即回退到扫描字符串 β 之前的状态，然后再由该状态扫描这个归约后的字符 A 进行新的“移进”操作。

例 3.18 试构造下述文法的 LR(0)分析表：

G[E]：E→E+T｜T
T→(E)｜a

[解答] (1) 将文法 G[E]拓广为 G[E']：

G[E']：(0) E'→E
(1) E→E+T
(2) E→T
(3) T→(E)
(4) T→a

(2) 列出 LR(0)的所有项目：

1．E'→•E	8．E→T•
2．E'→E•	9．T→•(E)
3．E→•E+T	10．T→(•E)
4．E→E•+T	11．T→(E•)
5．E→E+•T	12．T→(E)•
6．E→E+T•	13．T→•a
7．E→•T	14．T→a•

(3) 用 ε_CLOSURE(闭包)方法得到文法 G'[E']的 LR(0)项目集规范族如下：

I_0: E'→•E	I_4: T→a•
E→•E+T	I_5: E→E+•T
E→•T	T→•(E)
T→•(E)	T→•a
T→•a	I_6: T→(E•)

I_1: E'→E•　　E→E•+T
　　E→E•+T　　I_7: E→E+T•
I_2: E→T•　　I_8: T→(E)•
I_3: T→(•E)
　　E→•E+T
　　E→•T
　　T→•(E)
　　T→•a

(4) 根据状态转换函数 GO 构造出文法 G'[E'] 的 DFA M，如图 3-38 所示。

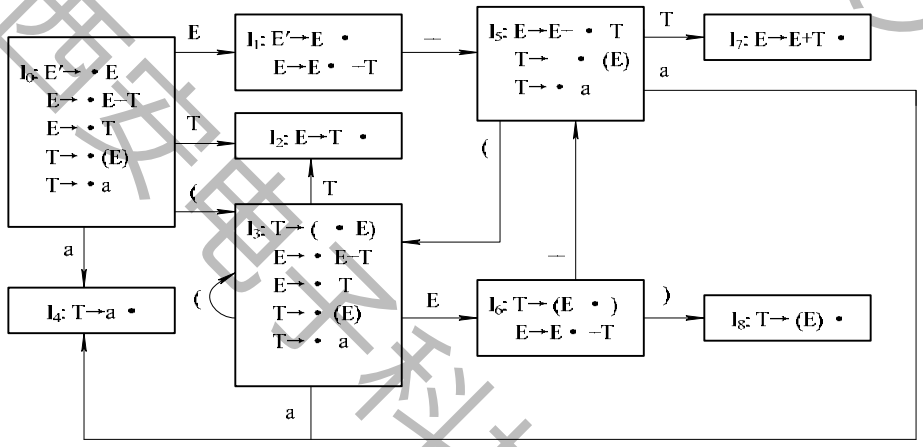

图 3-38　例 3.18 文法 G'[E'] 的 DFA M

(5) 构造 LR(0) 分析表如表 3.16 所示。

表 3.16　例 3.18 的 LR(0) 分析表

状态	ACTION					GOTO	
	a	+	(	)	#	E	T
0	s_4		s_3			1	2
1		s_5			acc		
2	r_2	r_2	r_2	r_2	r_2		
3	s_4		s_3			6	2
4	r_4	r_4	r_4	r_4	r_4		
5	s_4		s_3				7
6		s_5		s_8			
7	r_1	r_1	r_1	r_1	r_1		
8	r_3	r_3	r_3	r_3	r_3		

3.5.3　SLR(1) 分析器

LR(0) 文法是一类非常简单的文法，其特点是该文法的活前缀识别自动机的每一状态

(项目集)都不含冲突性的项目。但是，即使是定义算术表达式这样的简单文法也不是 LR(0)的，因此，需要研究一种带有简单“展望”材料的 LR 分析法，即 SLR(1)法。

由 LR(0)分析表可知，当出现形如 A→α•的归约项目时，ACTION 表状态 k (设 A→α•属于 I_k)这一行将全部填满产生式 A→α 的归约编号 r_j。但是，并非所有的终结符都允许跟在已归约的非终结符 A 之后，对于那些根本不可能出现在 A 之后的终结符 b，相应的 ACTION[k,b]就无需填入 r_j。这一做法的意义在于它可以减少“移进/归约”冲突的发生(如果 ACTION[k,b]又要填入移进 s_i 的话)，而非终结符 A 允许其后出现哪些终结符则完全可以采用 LL(1)分析法中的 FOLLOW 集求得。

一般而言，假定 LR(0)规范族的一个项目集 I 中含有 m 个移进项目：

$$A_1\rightarrow\alpha\bullet a_1\beta_1，\ A_2\rightarrow\alpha\bullet a_2\beta_2，\ \cdots，\ A_m\rightarrow\alpha\bullet a_m\beta_m$$

同时含有 n 个归约项目：

$$B_1\rightarrow\alpha\bullet，\ B_2\rightarrow\alpha\bullet，\ \cdots，\ B_n\rightarrow\alpha\bullet$$

如果集合 $\{a_1,\cdots,a_m\}$、FOLLOW(B_1)、…、FOLLOW(B_n)两两不相交(包括不得有两个 FOLLOW 集含有“#”)，则隐含在 I 中的动作冲突可通过检查现行输入符号 a 属于上述 n+1 个集合中的哪个集合而获得解决，即：

(1) 若 a 是某个 a_i，$i=1,2,\cdots,m$，则移进；

(2) 若 $a\in$FOLLOW(B_i)，$i=1,2,\cdots,n$，则用产生式 $B_i\rightarrow\alpha$ 进行归约；

(3) 对(1)、(2)项以外的情况，报错。

冲突性动作的这种解决办法叫做 SLR(1)解决办法。

对任给的一个文法 G[S]，我们可用如下办法构造它的 SLR(1)分析表：先把 G[S]拓广为 G'[S']，对 G'[S']构造 LR(0)项目集规范族 C 和活前缀识别自动机的状态转换函数 GO，然后再使用 C 和 GO 按下面的算法构造 G'[S']的 SLR(1)分析表。

假定 $C=\{I_0,I_1,\cdots,I_n\}$，令每个项目集 I_k 的下标 k 为分析器的一个状态，则 G'[S']的 SLR(1)分析表含有状态 0，1，…，n。令那个含有项目 S'→•S 的 I_k 的下标 k 为初态，则子表 ACTION 和 GOTO 可按如下方法构造：

(1) 若项目 A→α•aβ 属于 I_k 且 GO(I_k,a)=I_j，a 为终结符，则置 ACTION[k,a]为“将状态 j 和符号 a 移进栈”，简记为“s_j”。

(2) 若项目 A→α•属于 I_k，那么对任何输入符号 a，a 为终结符且 $a\in$FOLLOW(A)，置 ACTION[k,a]为“用产生式 A→α 进行归约”，简记为“r_j”。其中，j 是产生式的编号，即 A→α 是文法 G'[S']的第 j 个产生式。

(3) 若项目 S'→S•属于 I_k，则置 ACTION[k,#]为“接受”，简记为“acc”。

(4) 若项目 A→α•Bβ 属于 I_k 且 GO(I_k,B)=I_j，B 为非终结符，则置 GOTO[k,B]=j，即意为将(j,B)移进栈。

(5) 分析表中凡不能用规则(1)～(4)填入信息的空白格均置为“出错标志”。

按上述算法构造的含有 ACTION 和 GOTO 两部分的分析表，如果每个入口不含多重定义，则称它为文法 G[S]的一张 SLR(1)表，具有 SLR(1)表的文法 G[S]称为一个 SLR(1)文法。数字“1”的意思是在分析过程中最多只要向前看一个符号(实际上仅是在归约时需要向前看一个符号)，使用 SLR(1)表的分析器叫做 SLR(1)分析器。

若按上述算法构造的分析表存在多重定义的入口(即含有动作冲突)，则说明文法 G[S]

不是 SLR(1)的；在这种情况下，不能用上述算法构造分析器。

注意：SLR(1) 方法与 LR(0) 方法的区别仅在于步骤(2)。在 LR(0) 方法中，若项目集 I_k 含有 A→α•，则在状态 k 时，无论面临什么输入符号都采取“A→α 归约”的动作；假定 A→α 的产生式编号为 j，则在分析表 ACTION 部分，对应状态 k 这行所有栏目都填为“r_j”。而在 SLR(1) 方法中，若项目集 I_k 含有 A→α•，则在状态 k 时，仅当面临的输入符号为 a∈FOLLOW(A) 时，才确定采取“A→α 归约”的动作，这样将在分析表 ACTION 部分面对状态 k 这一行，所有 b∉FOLLOW(A) 的栏目将空出来。对空出来的栏目(假定该栏目对应的终结符就是 b)，如果恰好又存在项目 A→α•bβ 属于 I_k 且 $GO(I_k, b)=I_i$，则可置该栏目(即 ACTION[k, b])为“s_i”。但是这种情况在 LR(0) 就不行了，因为对应状态 k 这一行的所有栏目都已填入了“r_j”，此时若将 ACTION[k, b] 栏目填上“s_i”将产生冲突。因此，SLR(1) 方法比 LR(0) 优越，它可以解决更多的冲突。

例 3.19　试构造例 3.17 所示文法 G[S]的 SLR(1)分析表。

[解答]　构造 SLR(1)分析表必须先求出所有形如“A→α•”的 FOLLOW(A)，即对文法 G'[S']的归约项目：

S'→S•

S→BB•

B→aB•

B→b•

求 FOLLOW 集(实际上仍是对文法 G'[S']求 FOLLOW 集)，由 FOLLOW 集的构造方法得：

① FOLLOW(S')={#}；

② 由 S→BB 得 FIRST(B)⊂FOLLOW(B)，即 FOLLOW(B)={a,b}；

③ 由 S'→S 得 FOLLOW(S')⊂FOLLOW(S)，即 FOLLOW(S)={#}；由 S→BB 得 FOLLOW(S)⊂FOLLOW(B)，即 FOLLOW(B)={a,b,#}。

根据 SLR(1)分析表的构造方法的文法 G'[S']的 SLR(1)分析表见表 3.17。

表 3.17　例 3.19 的 SLR(1) 分析表

状　态	ACTION			GOTO	
	a	b	#	S	B
0	s_3	s_4		1	2
1			acc		
2	s_3	s_4			5
3	s_3	s_4			6
4	r_3	r_3	r_3		
5			r_1		
6	r_2	r_2	r_2		

例 3.20 已知文法 G[S]如下，试构造该文法的 SLR(1)分析表：

G[S]：S→bAS | bA

A→aSc

[解答] (1) 将文法 G[S]拓广为文法 G'[S']：

G'[S']：(0) S'→S

(1) S→bAS

(2) S→bA

(3) A→aSc

(2) 根据文法 G'[S']构造出 DFA M，如图 3-39 所示。

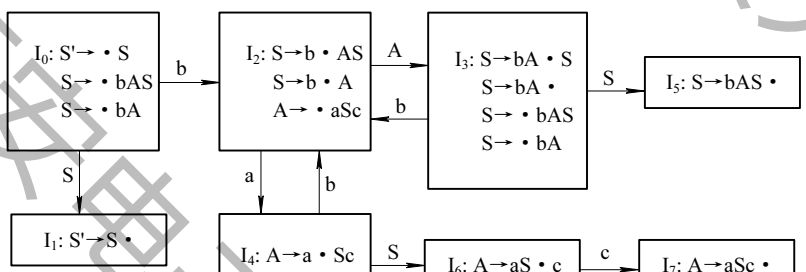

图 3-39 例 3.20 文法 G'[S']的 DFA M

(3) 对 G'[S']的所有归约项目：

S'→S•

S→bAS•

S→bA•

A→aSc•

求出 FOLLOW 集如下：

① FOLLOW(S')={#}；

② 由 S→…AS 得 FIRST(S)\{ε}⊂FOLLOW(A)，即 FOLLOW(A)={b}；

由 A→…Sc 得 FIRST('c') ⊂FOLLOW(S)，即 FOLLOW(S)={c}；

③ 由 S'→S 得 FOLLOW(S') ⊂FOLLOW(S)，即 FOLLOW(S)={c,#}；

由 S→…A 得 FOLLOW(S) ⊂FOLLOW(A)，即 FOLLOW(A)={b,c,#}。

由 SLR(1)分析表的构造方法可知：若移进项目 A→α•aβ(a 为终结符)属于 I_k 且由 I_k 发出的标记为 a 的有向边落到 I_j 上，则置 ACTION[k,a]为“s_j”；若待约项目 A→α•Bβ(B 为非终结符)属于 I_k 且由 I_k 发出的标记为 B 的有向边落到 I_j 上，则置 GOTO[k,B]为“j”；若归约项目 A→α•属于 I_k，则仅当 a∈FOLLOW(A)时，置 ACTION[k,a]为“r_i”(i 为文法 G'[S']中 A→α 的产生式编号)。所以，在 ACTION 表中可能发生“移进/归约”冲突，即在 ACTION[k,a]栏中，既要填入“s_j”又要填入“r_i”；而 GOTO 子表由于仅涉及待约项目，故不会发生任何冲突。

分析图 3-39 可知在 I_3 项目集中，S→bA•要求归约，而 S→bA•S 却要求移进 S，即对

移进的非终结符 S 和归约项 S→bA•左部非终结符 S 的 FOLLOW(S)有：

$$FIRST(S)\cap FOLLOW(S)=\{b\}\cap\{c,\#\}=\Phi$$

故 I_3 不产生冲突。

最后，根据图 3-39 可得到无冲突的 SLR(1)分析表见表 3.18。

表 3.18　例 3.20 的 SLR(1) 分析表

状态	ACTION				GOTO	
	a	b	c	#	S	A
0		s_2			1	
1				acc		
2	s_4					3
3		s_2	r_2	r_2	5	
4		s_2			6	
5			r_1	r_1		
6			s_7			
7		r_3	r_3	r_3		

例 3.21　已知算术表达式文法 G[E]如下，试构造该文法的 SLR(1)分析表。

G[E]：E→E+T｜T
T→T*F｜F
F→(E)｜i

[解答]　将文法 G[E]拓广为文法 G'[S']：

(0) S'→E
(1) E→E+T
(2) E→T
(3) T→T*F
(4) T→F
(5) F→(E)
(6) F→i

列出 LR(0)的所有项目：

1．S'→•E	6．E→E+T•	11．T→T*•F	16．F→(•E)
2．S'→E•	7．E→•T	12．T→T*F•	17．F→(E•)
3．E→•E+T	8．E→T•	13．T→•F	18．F→(E)•
4．E→E•+T	9．T→•T*F	14．T→F•	19．F→•i
5．E→E+•T	10．T→T•*F	15．F→•(E)	20．F→i•

用 ε_CLOSURE 方法构造出文法 G'[S']的 LR(0)项目集规范族，并根据状态转换函数 GO 画出文法 G'[S']的 DFA M，如图 3-40 所示。

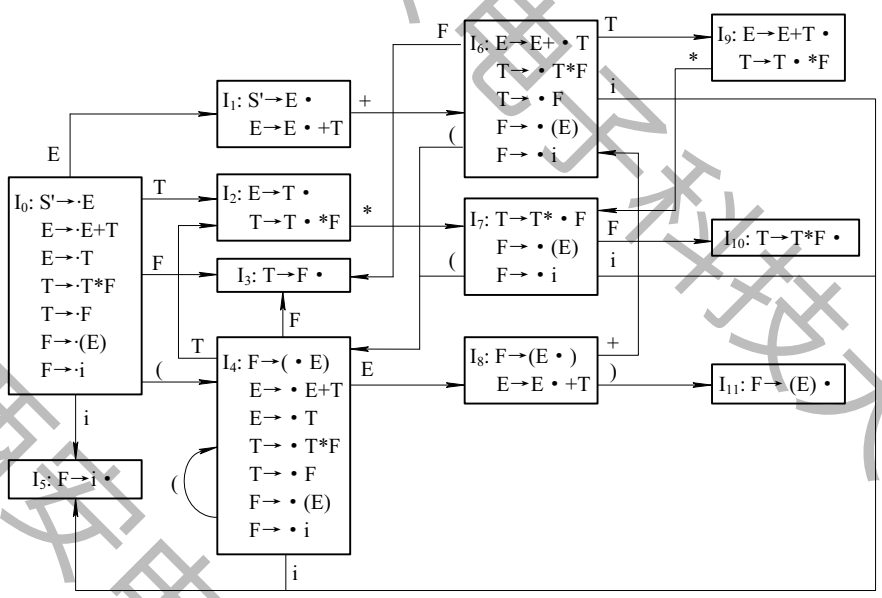

图 3-40 例 3.21 文法 G'[S'] 的 DFA M

根据 FOLLOW 集构造方法，构造文法 G'[S'] 中非终结符的 FOLLOW 集如下：

(1) 对文法开始符 S'，有#∈FOLLOW(S')，即 FOLLOW(S')={#}。

(2) 由 E→E+…得 FIRST('+')\{ε}⊂FOLLOW(E)，即 FOLLOW(E)={+}。

由 E→…E)得 FIRST(')')\{ε}⊂FOLLOW(E)，即 FOLLOW(E)={+,)}。

由 T→T*…得 FIRST('*')\{ε}⊂FOLLOW(T)，即 FOLLOW(T)={*}。

(3) 由 S'→E 得 FOLLOW(S')⊂FOLLOW(E)，即 FOLLOW(E)={+,),#}。

由 E→T 得 FOLLOW(E) ⊂FOLLOW(T)，即 FOLLOW(T) ={+, *,), #}。

由 T→F 得 FOLLOW(T) ⊂FOLLOW(F)，即 FOLLOW(F) ={+, *,), #}。

分析图 3-40 可知 I_1、I_2 和 I_9 存在移进和归约项目。在 I_1 项目集中，S'→E•要求归约，而 E→E•+T 却要求移进，即对移进的终结符+和归约项 S'→E•左部非终结符 S'的 FOLLOW(S')有：

FIRST('+')∩FOLLOW(S')={+}∩{#}=Φ

在 I_2 项目集中，E→T•要求归约，而 T→T•*F 却要求移进，即对移进的终结符*和归约项 E→T•左部非终结符 E 的 FOLLOW(E)有：

FIRST('*')∩FOLLOW(E)={*}∩{+,),#}=Φ

在 I_9 项目集中，E→E+T•要求归约，而 T→T•*F 却要求移进，即对移进的终结符*和归约项 E→E+T•左部非终结符 E 的 FOLLOW(E)有：

FIRST('*')∩FOLLOW(E)={*}∩{+,),#}=Φ

即 I_1、I_2 和 I_9 的移进和归约项目均不产生冲突。实际上在 SLR(1)分析表的构造过程中，如果分析表的所有栏目中均不存在冲突(即一个栏目中不存在两项及两项以上的内容)，则所构造的分析表即为 SLR(1)分析表。

根据图 3-40 可得到算术表达式文法 G[E]的 SLR(1)分析表见表 3.13。

3.5.4　LR(1)分析器

在 SLR(1)方法中，若项目集 I_k 含有 A→α•，那么在状态 k，只要所面临的输入符号 a ∈FOLLOW(A)时，就确定采取“用 A→α 归约”的动作。但是，也有很多文法不能采用 SLR(1)方法进行分析，如某文法的项目集规范族中含有如下的项目集 I_k：

I_k：S→α•aβ

A→α•

B→α•

若终结符 a 不属于 FOLLOW(A)且也不属于 FOLLOW(B)，就不会产生“移进/归约”冲突(即 ACTION[k, a] 只填移进 s_i)；但是，如果终结符 b 既属于 FOLLOW(A) 又属于 FOLLOW(B)(FOLLOW(A)∩FOLLOW(B)≠Φ)，则在 ACTION[k, b]栏产生了“归约/归约”冲突(该栏既要填 A→α 的归约编号 r_i，又要填 B→α 的归约编号 r_j)，而这种“归约/归约”冲突是无法用 SLR(1)方法解决的。

显然，对 LR(0)和 SLR(1)归约来说，其归约的考察范围只局限于短语，即只要找到形成某一非终结符的句柄(最左直接短语，且为该非终结符所对应的某一产生式的右部)时就进行归约；这时，如果某个项目集中有两个归约项目并含有相同的产生式右部(如上例中的 A→α•和 B→α•)，就必然产生“归约/归约”冲突。为了解决这个问题，我们将是否归约的考察范围由文法中的短语限定为语法分析过程中当前句型中的短语，即待归约的短语除了要求是句柄外，还必须是当前规范句型中的短语。这样，使归约的限制更加严格；当项目集 I_k 中出现归约项目 A→α•和 B→α•时，就会由当前规范句型来决定是将 α 归约为 A 还是 B，从而减少了“归约/归约”冲突的发生。但是，这种方法实现的前提却需要让每个项目(状态)含有更多有关归约的信息。

因此，我们可以进一步拓展 SLR(1)分析方法，即让每个状态含有更多的“展望”信息，在必要时也可以将一个状态分裂为两个或多个状态，使得 LR 分析器的每个状态都能确切指出在用 A→α 归约时，当 α 后跟哪些终结符时才允许把 α 归约为 A。也即，当状态 k 呈现于状态栈栈顶，而符号栈里的符号串为 βα 且当前面临的输入符号为 a，则只有存在含有活前缀 βAa 的规范句型时，才允许把 α 归约为 A，否则不允许归约；这就是 LR(1)分析方法。仅察看短语 α 后一个符号是为了提高分析的效率，当然也可以察看短语 α 后 k 个符号而形成 LR(k)分析方法。

LR(0)、SLR(1)和 LR(1)的区别就体现在“归约”操作上。若项目 A→α•属于 I_k，则当用产生式 A→α 归约时，LR(0)是无论面临什么输入符号(终结符)都进行归约；SLR(1)仅当面临的输入符号 a∈FOLLOW(A)时才进行归约；而 LR(1)则限定只有 α 后跟终结符 a，即形成当前规范句型的活前缀时才允许把 α 归约为 A。注意，在 SLR(1)里这个 a 是在整个文法中找出来的 a∈FOLLOW(A)，而 LR(1)则将其缩小到语法分析过程中当前句型中的短语 α(产生式 A→α)所允许出现在 A 之后的终结符 a，体现在 LR(1)项目集规范族中，就是找出每个项目集中属于各归约项目的向前搜索符。由于具体到某个项目集里特定的归约项目，显然其向前搜索符要少于 SLR(1)中相应归约项目 FOLLOW 集中的字符。因此，在 LR(1)中出现的归约比 SLR(1)要少；也即 LR(1)的效率更高，解决的冲突也多于 SLR(1)。

我们需要重新定义项目，使得每个项目都附带有 k 个终结符。现在每个项目的一般形

式为

$$[A\rightarrow\alpha\bullet\beta, a_1a_2\cdots a_k]$$

此处，A→α•β 是一个 LR(0)项目，每一个 a 都是终结符。这样的项目称为一个 LR(k)项目，项目中的 $a_1a_2\cdots a_k$ 称为它的向前搜索符串(或展望串)。向前搜索符串仅对归约项目[A→α•, $a_1a_2\cdots a_k$]有意义。对于任何移进或待约项目[A→α•β, $a_1a_2\cdots a_k$]，β≠ε，搜索符串 $a_1a_2\cdots a_k$ 不起作用。归约项目[A→α•, $a_1a_2\cdots a_k$]意味着当它所属的状态呈现在栈顶且后续的 k 个输入符号为 $a_1a_2\cdots a_k$ 时，才可以把栈顶的 α 归约为 A。这里，我们只对 k≤1 的情形感兴趣，因为对多数程序语言的语法来说，向前搜索(展望)一个符号就基本可以确定“移进”或“归约”了。

构造有效的 LR(1)项目集族的办法本质上和构造 LR(0)项目集规范族的办法是一样的，也需要两个函数：CLOSURE 和 GO。

假定 I 是一个项目集，它的闭包 CLOSURE(I)可按如下方法构造：

(1) I 的任何项目都属于 CLOSURE(I)。

(2) 若项目[A→α•Bβ,a]属于 CLOSURE(I)，B→γ 是一个产生式，那么对于 FIRST(βa)中的每个终结符 b，如果[B→•γ,b]原来不在 CLOSURE(I)中，则把它加进去。

(3) 重复执行步骤(2)，直到 CLOSURE(I)不再增大为止。

注意：b 可能是从 β 推出的第一个终结符，若 β 推出 ε，则 b 就是 a。

令 I 是一个项目集，X 是一个文法符号，则函数 GO(I,X)定义为

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

其中，如果[A→α•Xβ,a]∈I，则 J={任何形如[A→αX•β,a]的项目}。

下面根据文法 LR(1)项目集族 C 构造分析表。假定 $C=\{I_0,I_1,\cdots,I_n\}$，令每个 I_k 的下标 k 为分析表的状态，令那个含有[S'→•S,#]的 I_k 的 k 为分析器的初态。则子表 ACTION 和子表 GOTO 可按如下方法构造：

(1) 若项目[A→α•aβ,b]属于 I_k 且 $GO(I_k,a)=I_j$，a 为终结符，则置 ACTION[k,a]为“将状态 j 和符号 a 移进栈”，简记为“s_j”；

(2) 若项目[A→α•,a]属于 I_k，则置 ACTION[k,a]为“用产生式 A→α 归约”，简记为“r_j”；其中，j 是产生式的编号，即 A→α 是文法 G'[S']的第 j 个产生式；

(3) 若项目[S'→S•,#]属于 I_k，则置 ACTION[k,#]为“接受”，简记为“acc”；

(4) 若项目[A→α•Bβ,b]属于 I_k 且 $GO(I_k,B)=I_j$，B 为非终结符，则置 GOTO[k,B]=j，即意为将(j,B)移进栈。

(5) 分析表中凡不能用规则(1)～(4)填入信息的空白栏均置为“出错标志”。

按上述算法构造的分析表，若不存在多重定义入口(即动作冲突)的情形，则称它是文法 G[S]的一张规范的 LR(1)分析表，使用这种分析表的分析器叫做一个规范的 LR 分析器，具有规范的 LR(1)分析表的文法称为一个 LR(1)文法。

注意：构造有效的 LR(1)项目集在求闭包 CLOSURE(I)时与 LR(0)是有区别的。若 A→α•Bβ 属于 CLOSURE(I)，关于 B 的产生式是 B→γ，则对 LR(0)来说，项目 B→•γ 也属于 CLOSURE(I)；但对 LR(1)(假定 A→α•Bβ 的后续一个字符为 a)，则要求对 FIRST(βa)中的每个终结符 b，都有项目[B→•γ,b]属于 CLOSURE(I)。

其实，向前搜索符仅对归约项目有意义，因为只有当输入符号是归约项目的向前搜索

符时才表明该归约的操作是正确的操作，否则只能是移进操作或者出错。但是在构造 LR(1) 分析表中，直接为归约项目确定向前搜索符是困难的，所以只能先确定 A→•α,a，然后再推得 A→α•,a。

对 LR(1) 来说，其中的一些状态(项目集)除了向前搜索符不同外，其余都是相同的；也即 LR(1) 比 SLR(1) 和 LR(0) 存在更多的状态。因此，LR(1) 的构造比 LR(0) 和 SLR(1) 更复杂，占用的存储空间也更多。

例 3.22　试构造例 3.17 所示文法 G[S] 的 LR(1) 分析表。

[解答]　由 FOLLOW 集构造方法知

$$\text{FOLLOW(S')}=\{\#\};$$

且由 S'→S 知 FOLLOW(S')⊂FOLLOW(S)，即 FOLLOW(S)={#}；也即 S 的向前搜索字符为“#”(实际上可直接看出)，即 [S'→•S,#]。令 [S'→•S,#]∈CLOSURE(I_0)，我们根据 LR(1) 闭包 CLOSURE(I) 的构造方法求出属于 I_0 的所有项目。

① 已知 [S'→•S,#]∈CLOSURE(I_0)，S→BB 是一个产生式，且由构造方法“β = ε 得到 b=a='#'”求得 [S→•BB,#]∈CLOSURE(I_0)；

② 已知 [S→•BB,#]∈CLOSURE(I_0)，B→aB 是一个产生式，又 FIRST(B)={a,b}(此处的 B 是指 S→•BB 中的第二个 B)，即有 [B→•aB,a/b]∈CLOSURE(I_0)；

③ 已知 [S→•BB,#]∈CLOSURE(I_0)，B→b 是一个产生式，且 FIRST(B)={a,b}，即有 [B→•b,a/b]∈CLOSURE(I_0)。

由此可以得到项目集 I_0 如下：

I_0：S→•S′,#
S→•BB,#
B→•aB,a/b
B→•b,a/b

同理可求得全部 CLOSURE(I)，再根据状态转换函数 GO 的算法构造出文法 G'[S'] 的 DFA 如图 3-41 所示。

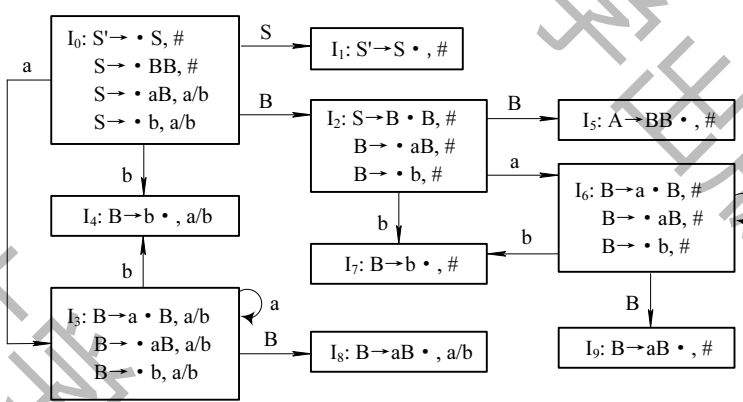

图 3-41　例 3.17 文法 G'[S'] 的 DFA M

LR(1) 分析表构造与 LR(0) 和 SLR(1) 的主要区别是构造算法步骤(2)，即仅当归约时搜索符才起作用。根据 LR(1) 分析表的构造算法得到 LR(1) 分析表见表 3.19。

表 3.19 例 3.22 的 LR(1) 分析表

状 态	ACTION			GOTO	
	a	b	#	S	B
0	s_3	s_4		1	2
1			acc		
2	s_6	s_7			5
3	s_3	s_4			8
4	r_3	r_3			
5			r_1		
6	s_6	s_7			9
7			r_3		
8	r_2	r_2			
9			r_2		

例 3.23 判断下述文法 G[S] 是哪类 LR 文法。

G[S]：(1) S→L=R
(2) S→R
(3) L→*R
(4) L→i
(5) R→L

[解答] 首先将文法 G[S] 拓广为 G'[S']：

G'[S']：(0) S'→S
(1) S→L=R
(2) S→R
(3) L→*R
(4) L→i
(5) R→L

构造文法 G'[S'] 的 LR(0) 项目集规范族如下：

I_0: S'→•S
S→•L=R
S→•R
L→•*R
L→•i
R→•L

I_1: S'→S•

I_2: S→L•=R
R→L•

I_3: L→*•R
R→•L
L→•*R
L→•i

I_4: L→i•

I_5: S→R•

I_6: S→L=•R
R→•L
L→•*R
L→•i

I_7: S→L=R•

I_8: L→*R•

我们知道，如果每个项目集中不存在既含移进项目又含归约项目或者含有多个归约项目的情况，则该文法是一个 LR(0) 文法。检查上面的项目集规范族，发现 I_2 存在既含移进项目 S→L•=R 又含归约项目 R→L•的情况，故文法 G[S] 不是 LR(0) 文法。

假定 LR(0) 规范族的一个项目集 I 中含有 m 个移进项目，即

$A_1 \rightarrow \alpha \bullet a_1\beta_1$，$A_2 \rightarrow \alpha \bullet a_2\beta_2$，…，$A_m \rightarrow \alpha \bullet a_m\beta_m$

同时 I 中含有 n 个归约项目，即

$$B_1\rightarrow\alpha\bullet，B_2\rightarrow\alpha\bullet，\cdots，B_n\rightarrow\alpha\bullet$$

如果集合$\{a_1,\cdots,a_m\}$、FOLLOW(B_1)、…、FOLLOW(B_n)任何两个出现相交的情况(包括存在两个 FOLLOW 集含有“#”)，则该文法不是 SLR(1)文法。

因此，构造文法 G'[S']的 FOLLOW 集如下：

(1) FOLLOW(S')={#}；

(2) 由 S→L=…得 FIRST('=')\{ε}⊂FOLLOW(L)，即 FOLLOW(L)={=}；

(3) 由 S'→S 得 FOLLOW(S')⊂FOLLOW(S)，即 FOLLOW(S)={#}；

由 S→R 得 FOLLOW(S)⊂FOLLOW(R)，即 FOLLOW(R)={#}；

由 L→…R 得 FOLLOW(L)⊂FOLLOW(R)，即 FOLLOW(R)={=,#}；

由 R→L 得 FOLLOW(R)⊂FOLLOW(L)，即 FOLLOW(L)={=,#}。

此外，由 I_2 的移进项目 S→L•=R 和归约项目 R→L•得

$$\{=\}\cap\text{FOLLOW(L)}=\{=\}\cap\{=,\#\}=\{=\}\neq\Phi$$

所以文法 G[S]不是 SLR(1)文法。

判断是否为 LR(1)文法则首先要构造 LR(1)项目集规范族。因此，构造文法 G'[S']的 LR(1)项目集规范族如下(项目集 I_0 由 S'→•S,#开始)：

I_0: S'→•S,#
S→•L=R,#
S→•R,#
L→•*R,=
L→•i,=
R→•L,#

I_1: S'→S•,#

I_2: S→L•=R,#
R→L•,#

I_3: S→R•,#

I_4: L→*•R,=
R→•L,=
L→•*R,=
L→•i,=

I_5: L→i•,=

I_6: S→L=•R,#
R→•L,#
L→•*R,#
L→•i,#

I_7: L→*R•,=

I_8: R→L•,=

I_9: S→L=R•,#

I_{10}: R→L•,#

I_{11}: L→*•R,#
R→•L,#
L→•*R,#
L→•i,#

I_{12}: L→i•,#

I_{13}: L→*R•,#

此时，I_2 的移进项目[S→L•=R,#]和归约项目[R→L•,#]有

$$\{=\}\cap\{\#\}=\Phi$$

故文法 G[S]是 LR(1)文法。

3.5.5　LALR(1)分析器

对 LR(1)来说，存在着某些状态(项目集)，这些状态除向前搜索符不同外，其核心部分都是相同的。能否将核心部分相同的诸状态合并为一个状态，这种合并是否会产生冲突？下面将对此进行讨论。

两个 LR(1)项目集具有相同的心是指除去搜索符之后这两个集合是相同的。如果把所有同心的 LR(1)项目集合并为一，将看到这个心就是一个 LR(0)项目集，这种 LR 分析法称为 LALR(1)方法。对于同一种文法，LALR(1)分析表和 LR(0)以及 SLR(1)分析表永远具有相同数目的状态。LALR(1)方法本质上是一种折中方法，LALR(1)分析表比 LR(1)分析表要小得多，能力也差一点，但它却能对付一些 SLR(1)所不能对付的情况。

由于 GO(I,X)的心仅仅依赖于 I 的心，因而 LR(1)项目集合并之后的转换函数 GO 可通过 GO(I,X)自身的合并而得到，因此在合并项目集时无需考虑修改转换函数的问题(假定 I_1 与 I_2 的心相同，即项目集相同，则 $GO(I_1,X)=GO(I_2,X)$，但这里的项目集是不包括搜索符的)。但是，动作 ACTION 必须进行修改，使之能够反映被合并的集合的既定动作。

假定有一个 LR(1)文法，它的 LR(1)项目集不存在动作冲突，但如果把同心集合并为一，就可能导致产生冲突。这种冲突不会是“移进/归约”间的冲突，因为若存在这种冲突，则意味着面对当前的输入符号 a，有一个项目[A→α•,a]要求采取归约动作，而同时又有另一项目[B→β•aγ,b]要求把 a 移进。这两个项目既然同处于(合并之前的)某一个集合中，则意味着在合并前必有某个 c 使得[A→α•,a]和[B→β•aγ,c]同处于(合并之前的)某一集合中，然而这又意味着原来的 LR(1)项目集已经存在着“移进/归约”冲突了。因此，同心集的合并不会产生新的“移进/归约”冲突(因为是同心合并，所以只改变搜索符，而不改变“移进”或“归约”操作，故不可能存在“移进/归约”冲突)。同时，这也说明，如果原 LR(1)存在着“移进/归约”冲突，则 LALR(1)必定也有“移进/归约”冲突。

但是，同心集的合并有可能产生新的“归约/归约”冲突。例如，假定有对活前缀 ac 有效的项目集为{[A→c•,d],[B→c•,e]}，对 bc 有效的项目集为{[A→c•,e],[B→c•,d]}。这两个集合都不含冲突，它们是同心的，但合并后就变成{[A→c•,d/e],[B→c•,d/e]}，显然这已是一个含有“归约/归约”冲突的集合了，因为当面临 e 或 d 时，我们不知道该用 A→c 还是用 B→c 进行归约。

下面给出构造 LALR(1)分析表的算法，其基本思想是首先构造 LR(1)项目集族，如果它不存在冲突，就把同心集合并在一起。若合并后的集族不存在“归约/归约”冲突(即不存在同一个项目集中有两个像 A→c•和 B→c•这样的产生式具有相同的搜索符)，就按这个集族构造分析表。构造分析表算法的主要步骤如下：

(1) 构造文法 G[S]的 LR(1)项目集族 $C=\{I_0,I_1,\cdots,I_n\}$。

(2) 把所有的同心集合并在一起，记 $C'=\{J_0,J_1,\cdots,J_m\}$为合并后的新族，那个含有项目[S'→•S,#]的 J_k 为分析表的初态。

(3) 从 C'构造 ACTION 子表。

① 若[A→α•aβ,b]∈J_k 且 $GO(J_k,a)=J_j$，a 为终结符，则置 ACTION[k,a]为“s_j”；

② 若[A→α•,a]∈J_k，则置 ACTION[k,a]为“用 A→α 归约”，简记为“r_j”，其中，j 是产生式的编号，即 A→α 是文法 G'[S']的第 j 个产生式；

③ 若[S'→S•,#]∈J_k，则置 ACTION[k,#]为“接受”，简记为“acc”。

(4) GOTO 子表的构造。假定 J_k 是 I_{i1}、I_{i2}、…、I_{it} 合并后的新集，由于所有这些 I_i 同心，因而 $GO(I_{i1},X)$、$GO(I_{i2},X)$、…、$GO(I_{it},X)$也同心；记 J_i 为所有这些 GO 合并后的集(即合并后为第 J_i 个状态(项目集))，那么就有 $GO(J_k,X)=J_i$。于是，若项目[A→α•Bβ,b]∈J_k 且 $GO(J_k,B)=J_j$，B 为非终结符，则置 GOTO[k,B]=j。

(5) 分析表中凡不能用(3)、(4)填入信息的空白格均置为“出错标志”。

注意，(3)、(4)中的 J_k、J_i 均为同心集合并后的状态编号。

经上述步骤构造的分析表若不存在冲突，则称它为文法 G[S]的 LALR(1)分析表，存在这种分析表的文法称为 LALR(1)文法。

LALR(1)与 LR(1)的不同之处是当输入串有误时，LR(1)能够及时发现错误，而 LALR(1)则可能还继续执行一些多余的归约动作，但决不会执行新的移进，即 LALR(1)能够像 LR(1)一样准确地指出出错的地点。就文法的描述能力来说，有下面的结论：

$$LR(0)\subset SLR(1)\subset LR(1)\subset 无二义文法$$

例 3.24　试根据例 3.22 中的 LR(1)项目集族构造 LALR(1)分析表。

[解答]　根据 LR(1)项目集族将同心集合并在一起，即将图 3-41 中的 I_3 与 I_6、I_4 与 I_7 以及 I_8 与 I_9 分别合并成：

I_{36}: [B→a•B, a/b/#]　　I_{47}: [B→b•, a/b/#]

[B→•aB, a/b/#]　　I_{89}: [B→aB•, a/b/#]

[B→•b, a/b/#]

由合并后的集族按照 LALR(1)分析表的构造算法得到 LALR(1)分析表如表 3.20 所示。

表 3.20　例 3.24 的 LALR(1) 分析表

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

表 3.20 所示的 LALR(1)分析表与表 3.17 所示的 SLR(1)分析表是相同的，这是因为文法 G'[S']既可以用 SLR(1)构造，又可以用 LALR(1)构造。注意有些文法只能用 LALR(1)构造。

3.5.6　二义文法的应用

任何二义文法绝不是一个 LR 文法，因而也不是 SLR(1)或 LALR(1)文法，这是一条定理。但是，某些二义文法是非常有用的。如算术表达式的二义文法远比无二义文法简单，因为无二义文法需要定义算符优先级和结合规则的产生式，这就需要使用比二义文法更多的非终结符，从而导致构造的 LR 分析表有更多的状态。但是，二义文法的问题是因其没有算符优先级和结合规则而产生了二义性。因此，我们要讨论的是使用 LR 分析法的基本思想，凭借一些其它条件来分析二义文法所定义的语言。下面介绍通常处理二义文法的方法。

如果某文法的拓广文法的 LR(0)项目集规范族存在“移进(或接受)/归约(或接受)”冲突，则可采用 SLR(1)的 FOLLOW 集的办法予以解决。如果无法解决，则只有借助其它条件，如使用算符的优先级和结合规则的有关信息。

此外，还可以赋予每个终结符和产生式以一定的优先级。假定在面临输入符号 a 时碰到“移进/归约”(假定用 A→α 归约)冲突，那么就比较终结符 a 和产生式 A→α 的优先级。若 A→α 的优先级高于 a 的优先级，则执行归约，反之则执行移进。

假如对产生式 A→α 不特别赋予优先级，就认为 A→α 和出现在 α 中的最右终结符具有相同的优先级。自然，那些不涉及冲突的动作将不理睬赋予终结符和产生式的优先级信息。特别重要的是，只给出终结符和产生式的优先级往往不足以解决所有冲突，这时可以规定结合性质，使“移进/归约”冲突得以解决。实际上，左结合意味着打断联系而实行归约，右结合意味着维持联系而实行移进。对于“归约/归约”冲突，一种极为简单的解决办法是：优先使用列在前面的产生式进行归约，也即列在前面的产生式具有较高的优先级。

例 3.25 已知算术表达式文法 G[E]如下，试构造该文法的 SLR(1)分析表。

G[E]：E→E+E｜E*E｜(E)｜i

[解答] 将文法 G[E]拓广为文法 G'[S']：

(0) S'→E
(1) E→E+E
(2) E→E*E
(3) E→(E)
(4) E→i

列出 LR(0)的所有项目：

1．S'→•E	5．E→E+•E	9．E→E*•E	13．E→(E•)
2．S'→E•	6．E→E+E•	10．E→E*E•	14．E→(E)•
3．E→•E+E	7．E→•E*E	11．E→•(E)	15．E→•i
4．E→E•+E	8．E→E•*E	12．E→(•E)	16．E→i•

用 ε_CLOSURE 方法构造出文法 G'[S']的 LR(0)项目集规范族，并根据状态转换函数 GO 画出文法 G'[S']的 DFA M，如图 3-42 所示。

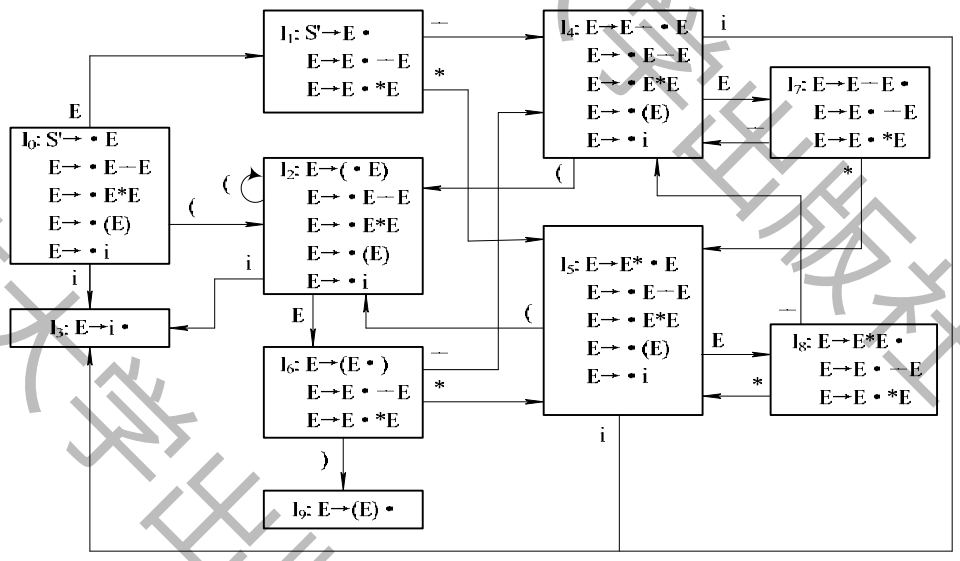

图 3-42 算术表达式文法 G'[S']的 DFA M

下面我们对文法 G'[S'] 中形如“A→α•”的项目求 FOLLOW 集。

I_1 ： S'→E•

I_7 ： E→E+E•

I_8 ： E→E*E•

I_9 ： E→(E)•

I_3 ： E→i•

根据 FOLLOW 集构造方法，构造文法 G'[S'] 中非终结符的 FOLLOW 集如下：

(1) 对文法开始符 S'，有#∈FOLLOW(S')，即 FOLLOW(S')={#}；

(2) 由 E→E+…得 FIRST('+')\{ε}⊂FOLLOW(E)，即 FOLLOW(E)={+}；

由 E→E*…得 FIRST('*')\{ε}⊂FOLLOW(E)，即 FOLLOW(E)={+,*}；

由 E→…E)得 FIRST(')')\{ε}⊂FOLLOW(E)，即 FOLLOW(E)={+,*,)}；

(3) 由 S'→E 得 FOLLOW(S')⊂FOLLOW(E)，即 FOLLOW(E)={+,*,),#}。

分析图 3-42 可知 I_7 和 I_8 存在矛盾：一方面它们存在“移进/归约”矛盾；另一方面，不论是“+”或“*”都属于 FOLLOW(E)。这说明文法 G'[S'] 是二义文法，即这种“移进/归约”冲突只有借助其它条件才能得到解决，这个条件就是使用算符“+”和“*”的优先级和结合规则的有关信息。

由于“*”的优先级高于“+”，则“+”遇见其后的“*”应移进，而“*”遇见其后的“+”应归约。此外，因服从左结合，故“+”遇见其后的“+”应归约，“*”遇见其后的“*”也应归约(注意，服从左结合则实行归约，服从右结合则实行移进)。

对于 I_7，因为是由活前缀…+E 转入 I_7 的(见图 3-42，由 I_0 开始到达 I_7 的有向边序列上形成的字符序列即为活前缀)，则其后遇到“+”应该归约，但遇到“*”则应移进。

对于 I_8，因为是由活前缀…*E 转入 I_8 的，则其后遇到“+”应该归约，遇到“*”同样应该归约。

上述两种情况由 I_0 开始到达 I_7 和由 I_0 开始到达 I_8 的有向边上形成的字符序列分别如图 3-43(a)、(b)所示。对图 3-43(a)，E+E 遇到后面的“+”则应将 E+E 归约为 E(即先计算 E+E)，而 E+E 遇到后面的“*”则不能先计算 E+E，必须将“*”移进；对图 3-43(b)，E*E 无论遇到后面的“+”还是“*”，都应先计算 E*E，也即将 E*E 归约为 E。

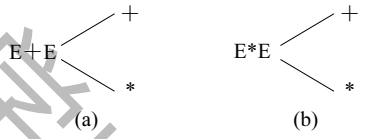

图 3-43　I_0 到 I_7 和 I_0 到 I_8 的有向边上形成的字符序列

由此得到算术表达式文法 G[E] 的 SLR(1) 分析表见表 3.21。

表 3.21　算术表达式文法 G[E] 的 SLR(1) 分析表

状 态	ACTION						GOTO
	i	+	*	(	)	#	E
0	s_3			s_2			1
1		s_4	s_5			acc	
2	s_3			s_2			6
3		r_4	r_4		r_4	r_4	
4	s_3			s_2			7

续表

状 态	ACTION						GOTO
	i	+	*	(	)	#	E
5	s_3			s_2			8
6		s_4	s_5		s_9		
7		r_1	s_5		r_1	r_1	
8		r_2	r_2		r_2	r_2	
9		r_3	r_3		r_3	r_3	

例 3.26 一程序语句的文法 G[S] 为

G[S]：S→if S else S

S→if S

S→S;S

S→a

该二义文法 G[S] 终结符 else 与最近的 if 结合，试用 LR 分析法的基本思想为 G[S] 构造 SLR(1) 分析表。

[解答] 为方便起见，用 i 代表 if，e 代表 else，然后将文法 G[S] 拓广为 G'[S']：

G'[S']：(0) S'→S

(1) S→iSeS

(2) S→iS

(3) S→S;S

(4) S→a

用 ε_CLOSURE 方法构造文法 G'[S'] 的 LR(0) 项目集规范族，并根据转换函数 GO 构造出文法 G'[S'] 的 DFA M，如图 3-44 所示。

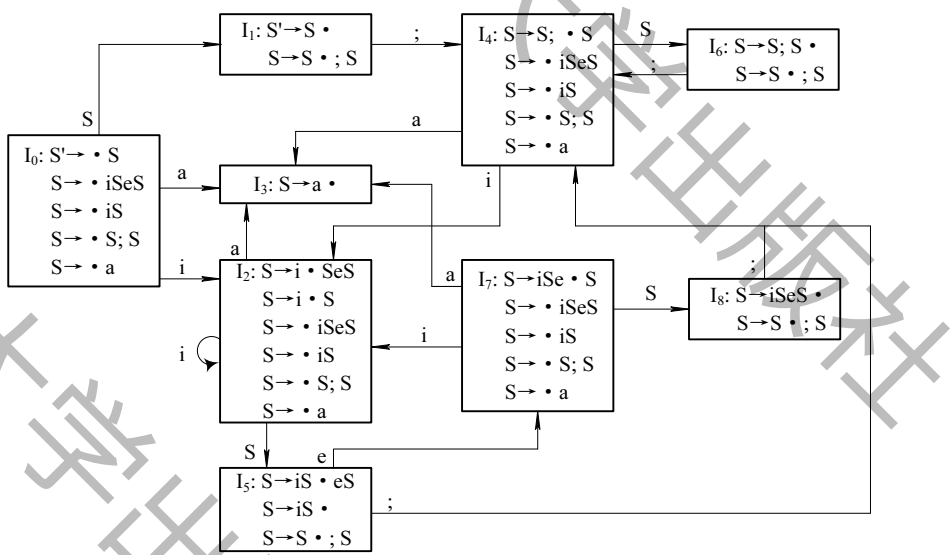

图 3-44 例 3.26 的文法 G'[S'] 的 DFA M

已知 FOLLOW(S')={#}，由 S'→S 得 FOLLOW(S')⊂FOLLOW(S)，即

$$FOLLOW(S)=\{\#\}$$

由 S→…Se…得 FIRST('e')⊂FOLLOW(S)，由 S→S;…得 FIRST(';')⊂FOLLOW(S)，即

$$FOLLOW(S)=\{\#,e,;\}$$

对 I_5，S→iS•要求归约，而 S→iS•eS 和 S→S•;S 却要求移进，即有：

$$FIRST('e')\cap FOLLOW(S)=\{e\}\neq\Phi$$

$$FIRST(';')\cap FOLLOW(S)=\{;\}\neq\Phi$$

也即冲突字符为“e”和“;”。

对于 I_5，由于是由活前缀“iS”达到 I_5 的，因此遇到后面的“e”，则应与前面的活前缀“iS”结合以便形成 iSeS 句型，故应移进；而遇到后面的“;”时，活前缀“iS”即为一个语句，故应归约。

I_6 和 I_8 引起冲突的字符是“;”。

对于 I_6，由于是由活前缀“S;S”到达 I_6 的，因此遇到后面的“;”时应将活前缀“S;S”归约为 S，故应归约。

对于 I_8，由于是由活前缀“iSeS”到达 I_8 的，因此遇到后面的“;”时应将活前缀“iSeS”归约为 S，故应归约。

最后，得到如表 3.22 所示的无冲突的 SLR(1) 分析表。

表 3.22　无冲突的 SLR(1) 分析表

状　态	ACTION					GOTO
	i	e	;	a	#	S
0	s_2			s_3		1
1			s_4		acc	
2	s_2			s_3		5
3		r_4	r_4		r_4	
4	s_2			s_3		6
5		s_7	r_2		r_2	
6		r_3	r_3		r_3	
7	s_2			s_3		8
8		r_1	r_1		r_1	

*3.5.7　LR 分析器的应用与拓展

LR 分析器除了应用于编译程序外，还可拓展到其它领域，如人工智能领域。

人工智能中采用的搜索方法如深度优先法、广度优先法、爬山法和 α–β 法等都可适用于一般的智能推测。由于要综合考虑各种因素，且推测的目标并不事先确定，故这些方法实质上是一种试探法或穷举法。因此，它们的共同特点是适用面广，但效率不高。

LR 分析器的推理过程是依据“历史”来展望“未来”，因此 LR 分析器同样具有智能性，但是这种智能却是有限的。作为归约过程的“历史”材料的积累虽不困难(已保存于栈中)，但是“展望”材料的汇集却很不容易。因为根据历史推测未来，即使是推测未来的一

个符号，也常常存在着非常多的不同可能性，所以在把“历史”和“展望”材料综合在一起时，复杂性就大大增加了。因此，具体实现 LR 分析器的功能时通常采用某种限制性措施，尽可能使出现的状态减少，只有这样才能提高效率并易于实现。如简单 LR——极有使用价值的 SLR(1)方法，它只在栈中保留已扫描过的那段输入符号的部分信息，即“历史”，并根据这些信息和未来的一串符号决定下一步的操作。SLR(1)的这种做法使得状态数大为减少，因此可以高效率地实现。

我们可以将上述方法和思想运用于人工智能领域。通过分析可以发现，人工智能的推理按目标可分为两类：一类是对未知目标的推理；一类是对已知目标的推理。对事先无法知道的目标的推理一般采用常规的智能搜索方法——试探法或穷举法；而对已知目标的推理，虽然仍可采用试探法或穷举法，但能否采用其它更加简单有效的方法呢？LR 分析器中的“历史”类同于人工智能中已搜索过的路径，而 LR 分析器对未来的“展望”恰好就是对已知目标的“展望”。LR 分析器是在“目标已知”这个特定条件下对人工智能常规搜索方法的一种简化。由于 LR 分析器推理的目标确定，故其效率远远高于常规的人工智能搜索方法。对那些事先可以分析出目标的推理问题，如智能化教学系统、智能化管理系统以及智能控制机床等，都可以采用 LR 分析器方法。LR 分析器是一种基于目标的智能推理方法，它适用于目标确定的智能化推理问题。

最后，我们通过下面的例子来进一步加深对本章 LR 分析表构造方法的掌握。

例 3.27 已知一 LR 分析表如表 3.23 所示，试根据该表求出与之相配的文法 G[S]。

表 3.23 LR 分析表

状态	ACTION			GOTO
	i	k	#	S
0	s_1	s_3		2
1	s_1	s_3		4
2			acc	
3			r_2	
4			r_1	

[解答] 根据表 3.23 可知 DFA M 含有 I_0～I_4 共 5 个项目集，同时可知各项目集之间的有向边及转换关系如图 3-45 所示。

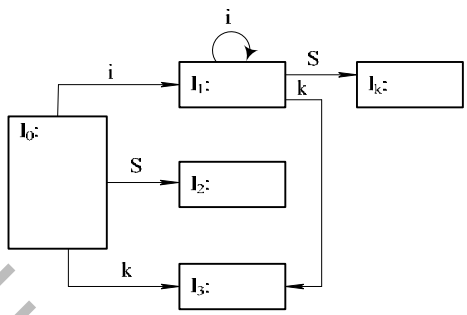

图 3-45 表 3.23 对应的 DFA M 结构图

(1) 对于 I_0：首先它含有 S'→•S 项目；此外根据由 I_0 发出的有向边上字母 i 和 k 得知 I_0 还含有如下两个项目：

S→•iα

S→•kβ

其中，α 和 β 待定。

(2) 对于 I_1：由于从 I_0 到 I_1 有一字母为 i 的有向边，则 I_1 含有 S→i•α 的项目(因 I_0 含有 S→•iα 项目)；又因 I_1 本身含有一字母为 i 的回路有向边，故判定 I_1 必有 S→i•S 和 S→•iS 两个项目，否则无法构成回路有向边，即得知 α 即为字母 S。

此外，I_1 还含有一字母为 k 的出边，参考 I_0 得知 I_1 有一 S→•kβ 项目，β 待定。

(3) 对于 I_2：由于表 3.23 状态 I_2 这一行仅有一项接受(acc)，故 I_2 仅含接受项目 S'→S•。

(4) 对于 I_3：由于 I_0 和 I_1 到达 I_3 有向边上字母均为 k，且 I_3 又无任何出边，故 I_3 只含 S→k•β 项目；由表 3.23 得知 I_3 仅含一个归约项，这里只有 β 为 ε 时 S→k•β 方可为归约项，故确定 I_0、I_1 和 I_3 中的 β 为 ε，即 I_3 只含 S→k•项目。

(5) 对于 I_4：由于从 I_1 经字母 S 的有向边到达 I_4，且 I_4 也仅含有归约项，故 I_4 只含 S→iS•项目。

归纳上述结果得到拓广文法 G'[S'] 为：

G'[S']：(0) S'→S

(1) S→iS

(2) S→k

故与表 3.23 相匹配的文法 G[S] 为：

G[S]：S→iS | k

习　题　3

3.1　完成下列选择题：

(1) 程序语言的语义需要用_____来描述。

A．上下文无关文法　　B．上下文有关文法

C．正规文法　　D．短语文法

(2) 2 型文法对应_____。

A．图灵机　　B．有限自动机

C．下推自动机　　D．线性界限自动机

(3) 下述结论中，_____是正确的。

A．1 型语言⊂0 型语言　　B．2 型语言⊂1 型语言

C．3 型语言⊂2 型语言　　D．A～C 均不成立

(4) 有限状态自动机能识别_____。

A．上下文无关文法　　B．上下文有关文法

C．正规文法　　D．短语文法

(5) 文法 G[S]：S→xSx | y 所识别的语言是_____。

A．xyx　　B．(xyx)*　　C．$x^nyx^n(n\geq 0)$　　D．x*yx*

(6) 只含有单层分枝的子树称为“简单子树”，则句柄的直观解释是_____。

A．子树的末端结点(即树叶)组成的符号串

B．简单子树的末端结点组成的符号串

C．最左简单子树的末端结点组成的符号串

D．最左简单子树的末端结点组成的符号串且该字符串必须含有终结符

(7) 下面对语法树错误的描述是_____。

A．根结点用文法 G[S] 的开始符 S 标记

B．每个结点用 G[S] 的一个终结符或非终结符标记

C．如果某结点标记为 ε，则它必为叶结点

D．内部结点可以是非终结符

(8) 由文法开始符 S 经过零步或多步推导产生的符号序列是_____。

A．短语　　B．句柄　　C．句型　　D．句子

(9) 设文法 G[S]：S→SA | A

A→a | b

则对句子 aba 的规范推导是_____。

A．S⇒ SA ⇒ SAA ⇒ AAA ⇒ aAA ⇒ abA ⇒ aba

B．S⇒ SA ⇒ SAA ⇒ AAA⇒ AAa ⇒ Aba ⇒ aba

C．S ⇒ SA⇒ SAA ⇒ SAa ⇒ Sba ⇒ Aba ⇒ aba

D．S ⇒ SA ⇒ Sa ⇒ SAa ⇒ Sba ⇒ Aba ⇒ aba

(10) 如果文法 G[S] 是无二义的，则它的任何句子 α 其_____。

A．最左推导和最右推导对应的语法树必定相同

B．最左推导和最右推导对应的语法树可能不同

C．最左推导和最右推导必定相同

D．可能存在两个不同的最右推导，但它们对应的语法树相同

(11) 一个句型的分析树代表了该句型的_____。

A．推导过程　　B．归约过程　　C．生成过程　　D．翻译过程

(12) 规范归约中的“可归约串”由_____定义。

A．直接短语　　B．最右直接短语　　C．最左直接短语　　D．最左素短语

(13) 规范归约是指_____。

A．最左推导的逆过程　　B．最右推导的逆过程

C．规范推导　　D．最左归约的逆过程

(14) 文法 G[S]：S→aAcB | Bd

A→AaB | c

B→bScA | b

则句型 aAcbBdcc 的短语是_____。

A．Bd　　B．cc　　C．a　　D．b

(15) 文法 G[E]：E→E+T | T

T→T*P | P

P→ (E) | i

则句型 P+T+i 的句柄和最左素短语是_____。

A．P+T 和 T　　B．P 和 P+T　　C．i 和 P+T+i　　D．P 和 P

(16) 采用自顶向下分析，必须_____。

A．消除左递归　　B．消除右递归

C．消除回溯　　D．提取公共左因子

(17) 对文法 G[E]：E→E+S | S

S→S*F | F

F→ (E) | i

则 FIRST(S) =_____。

A．{(}　　B．{(, i}　　C．{i}　　D．{(,)}

(18) 确定的自顶向下分析要求文法满足_____。

A．不含左递归　　B．不含二义性　　C．无回溯　　D．A～C 项

(19) 递归下降分析器由一组递归函数组成，且每一个函数对应文法的_____。

A．一个终结符　　B．一个非终结符

C．多个终结符　　D．多个非终结符

(20) LL(1) 分析表需要预先定义和构造两族与文法有关的集合_____。

A．FIRST 和 FOLLOW　　B．FIRSTVT 和 FOLLOW

C．FIRST 和 LASTVT　　D．FIRSTVT 和 LASTVT

(21) 设 a、b、c 是文法的终结符且满足优先关系 a⋹b 和 b⋹c，则_____。

A．必有 a⋹c　　B．必有 c⋹a　　C．必有 b⋹a　　D．A～C 都不一定成立

(22) 算符优先分析法要求文法_____。

A．不存在…QR…的句型且任何终结符对 (a, b) 满足 a⋹b、a⋖b 和 a⋗b 三种关系

B．不存在…QR…的句型且任何终结符对 (a, b) 至多满足 a⋹b、a⋖b 和 a⋗b 三种关系之一

C．可存在…QR…的句型且任何终结符对(a, b)至多满足 a⋹b、a⋖b 和 a⋗b 三种关系之一

D．可存在…QR…的句型且任何终结符对 (a, b) 满足 a⋹b、a⋖b 和 a⋗b 三种关系

(23) 任何算符优先文法_____优先函数。

A．有一个　　B．没有　　C．有若干个　　D．可能有若干个

(24) 在算符优先分析中，用_____来刻画可归约串。

A．句柄　　B．直接短语　　C．素短语　　D．最左素短语

(25) 下面最左素短语必须具备的条件中有错误的是_____。

A．至少包含一个终结符　　B．至少包含一个非终结符

C．除自身外不再包含其他素短语　　D．在句型中具有最左性

(26) 对文法 G[S]：S→b | ∧ | (T)

T→T, S | S

其 FIRSTVT(T) 为_____。

A．{b, ∧, (}　　B．{b, ∧,)}　　C．{, , b, ∧, (}　　D．{, , b, ∧,)}

(27) 对文法 G[E]：E→E*T | T
T→T+i | i
句子 1+2*8+6 归约的值为_____。
A．23 B．42 C．30 D．17

(28) 下述 FOLLOW 集构造方法中错误的是_____。
A．对文法开始符 S 有#∈FOLLOW(S)
B．若有 A→αBβ，则将 FIRST(β)\{ε}⊂FOLLOW(B)
C．若有 A→αB，则有 FOLLOW(B)⊂FOLLOW(A)
D．若有 A→αB，则有 FOLLOW(A)⊂FOLLOW(B)

(29) 若文法 G[S] 的产生式有…AB…出现，则对 A 求 FOLLOW 集正确的是_____。
A．FOLLOW(B)⊂FOLLOW(A) B．FIRSTVT(B)⊂FOLLOW(A)
C．FIRST(B)\{ε}⊂FOLLOW(A) D．LASTVT(B)⊂FOLLOW(A)

(30) 下面_____是自底向上分析方法。
A．预测分析法 B．递归下降分析器
C．LL(1) 分析法 D．算符优先分析法

(31) 下面_____是采用句柄进行归约的。
A．算符优先分析法 B．预测分析法
C．SLR(1) 分析法 D．LL(1) 分析法

(32) 一个_____指明了在分析过程中某时刻能看到产生式多大一部分。
A．活前缀 B．前缀 C．项目 D．项目集

(33) 若 B 为非终结符，则 A→α·Bβ 为_____项目。
A．接受 B．归约 C．移进 D．待约

(34) 在 LR(0) 的 ACTION 子表中，如果某一行中存在标记为“r_j”的栏，则_____。
A．该行必定填满 r_j B．该行未填满 r_j
C．其他行也有 r_j D．GOTO 子表中也有 r_j

(35) LR 分析法解决“移进/归约”冲突时，左结合意味着_____。
A．打断联系实行移进 B．打断联系实行归约
C．建立联系实行移进 D．建立联系实行归约

(36) LR 分析法解决“移进/归约”冲突时，右结合意味着_____。
A．打断联系实行归约 B．建立联系实行归约
C．建立联系实行移进 D．打断联系实行移进

(37) 若项目集 I_k 含有 A→α•，则在状态 k 时，仅当面临的输入符号 a∈FOLLOW(A) 时才采取“将 α 归约为 A”动作的一定是_____。
A．LALR(1) 文法 B．LR(0) 文法
C．LR(1) 文法 D．SLR(1) 文法

(38) 同心集合并又可能产生新的_____冲突。
A．“归约/接受” B．“移进/移进”
C．“移进/归约” D．“归约/归约”

3.2 令文法 G[N] 为

G[N]：N→D | ND

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

(1) G[N] 的语言 L(G) 是什么?

(2) 给出句子 0127、34 和 568 的最左推导和最右推导。

3.3　已知文法 G[S] 为 S→aSb | Sb | b，试证明文法 G[S] 为二义文法。

3.4　已知文法 G[S] 为 S→SaS | ε，试证明文法 G[S] 为二义文法。

3.5　按指定类型，给出语言的文法。

(1) L={a^ib^j | j>i≥1}的上下文无关文法；

(2) 字母表∑={a, b}上的同时只有奇数个 a 和奇数个 b 的所有串的集合的正规文法；

(3) 由相同个数 a 和 b 组成句子的无二义文法。

3.6　有文法 G[S]：S→aAcB | Bd

A→AaB | c

B→bScA | b

(1) 试求句型 aAaBcbbdcc 和 aAcbBdcc 的句柄；

(2) 写出句子 acabcbbdcc 的最左推导过程。

3.7　对于文法 G[S]：S→ (L) | aS | a

L→L, S | S

(1) 画出句型 (S,(a)) 的语法树；

(2) 写出上述句型的所有短语、直接短语、句柄、素短语和最左素短语。

3.8　下述文法描述了 C 语言整数变量的声明语句：

G[D]：D→TL

T→int | long | short

L→id | L, id

(1) 改造上述文法，使其接受相同的输入序列，但文法是右递归的；

(2) 分别以上述文法 G[D] 和改造后的文法 G'[D] 为输入序列 int a, b, c 构造分析树。

3.9　考虑文法 G[S]：S→ (T) | a+S | a

T→T, S | S

消除文法的左递归及提取公共左因子，然后对每个非终结符写出不带回溯的递归子程序。

3.10　已知文法 G[A]：A→aABl | a

B→Bb | d

(1) 试给出与 G[A] 等价的 LL(1) 文法 G'[A]；

(2) 构造 G'[A] 的 LL(1) 分析表；

(3) 给出输入串 aadl# 的分析过程。

3.11　将下述文法改造为 LL(1) 文法：

G[V]：V→N | N[E]

E→V | V+E

N→i

3.12　对文法 G[E]：E→E+T | T

T→T*P | P

P→i

(1) 构造该文法的优先关系表(不考虑语句括号#)，并指出此文法是否为算符优先文法；

(2) 构造文法 G[E] 的优先函数。

3.13 设有文法 G[S]：S→a | b | (A)

A→SdA | S

(1) 构造算符优先关系表；

(2) 给出句型 (SdSdS) 的短语、直接短语、句柄、素短语和最左素短语；

(3) 给出输入串 (adb)# 的分析过程。

3.14 在算符优先分析法中，为什么要在找到最左素短语的尾时才返回来确定其对应的头？能否按扫描顺序先找到头后再找到对应的尾，为什么？

3.15 试证明在算符文法中，任何句型都不包含两个相邻的非终结符。

3.16 给出文法 G[S]：S→aSb | P

P→bPc | bQc

Q→Qa | a

(1) 它是 Chomsky 的哪一型文法？

(2) 它生成的语言是什么？

(3) 它是不是算符优先文法？请构造算符优先关系表并证实之；

(4) 文法 G[S] 消除左递归、提取公共左因子后是不是 LL(1) 文法？请证实。

3.17 LR 分析器与优先关系分析器在识别句柄时的主要异同是什么？

3.18 什么是规范句型的活前缀？引进它的意义何在？

3.19 8086/8088 汇编语言对操作数域的检查可以用 LR 分析表实现。设 m 代表存储器，r 代表寄存器，i 代表立即数，并且为了简单起见，省去了关于 m、r 和 i 的产生式，暂且认为 m、r、i 为终结符，则操作数域 P 的文法 G[P]为

G[P]：P→m, r | m, i | r, r | r, i | r, m

试构造能够正确识别操作数域的 LR 分析表。

3.20 试构造下述文法的 SLR(1) 分析表：

G[E]：E→E+T | T

T→F* | F

F→ (E) | a

该文法能否构造 LR(0) 分析表？

3.21 试构造下述文法的 SLR(1) 分析表：

G[S]：S→bASB | bA

A→dSa | e

B→cAa | c

3.22 LR(0)、SLR(1)、LR(1) 及 LALR(1) 有何共同特征？它们的本质区别是什么？

3.23 请指出图 3-46 中的 LR 分析表 (a)、(b)、(c) 分属 LR(0)、SLR(1) 和 LR(1) 中的哪一种，并说明理由。

3.24 为二义文法 G[T] 构造一个 SLR(1) 分析表(详细说明构造方法)。其中终结符“,”满足右结合性，终结符“;”满足左结合性，且“,”的优先级高于“;”的优先级。

G[T]：T→TAT | bTe | a

A→, | ;

状态	ACTION		GOTO	
	b	#	S	B
0	s_3		1	2
1		acc		
2	s_4			5
3	r_2			
4		r_2		
5		r_1		

(a)

状态	ACTION			GOTO
	a	b	#	T
0	s_2	s_3		1
1			acc	
2	s_2	s_3		4
3	r_2	r_2	r_2	
4	r_1	r_1	r_1	

(b)

状态	ACTION			GOTO
	i	k	#	P
0	s_1	s_3		2
1	s_1	s_3		4
2			acc	
3			r_2	
4			r_1	

(c)

图 3-46　习题 3.23 的 LR 分析表

3.25　文法 G[T] 及其 LR 分析表(见表 3.24)如下，给出串 bibi 的分析过程。

G[T]：(1) T→EbH

(2) E→d

(3) E→ε

(4) H→i

(5) H→Hbi

(6) H→ε

表 3.24　习题 3.25 的 SLR(1) 分析表

状　态	ACTION				GOTO		
	b	d	i	#	T	E	H
0	r_3	s_3			1	2	
1				acc			
2	s_4						
3	r_2						
4	r_6		s_6	r_6			5
5	s_7			r_1			
6	r_4			r_4			
7			s_8				
8	r_5			r_5			

3.26　给出文法 G[S] 及图 3-47 所示的 LR(1) 项目集规范族中的 0、1、2、3、4。

G[S]：S→S; B | B

B→BaA | A

A→b (S)

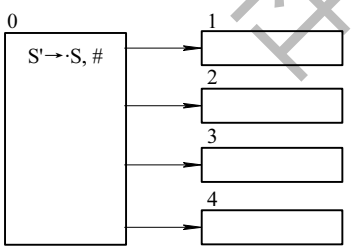

图 3-47　习题 3.26 的部分项目集

3.27　一个非 LR(1) 的文法如下：

G[L]：L→MLb | a

M→ε

请给出所有“移进/归约”冲突的 LR(1) 项目集，以说明该文法确实不是 LR(1) 的。

3.28 试证明任何一个 SLR(1) 文法一定是一个 LALR(1) 文法。

3.29 已知文法 G[S]：S→aAd | ; Bd | aB↑ | ; A↑

A→a

B→a

(1) 试判断 G[S] 是否为 LALR(1) 文法。

(2) 当一个文法是 LR(1) 而不是 LALR(1) 时，那么 LR(1) 项目集的同心集合并后会出现哪几种冲突，请说明理由。

3.30 给定文法 G[A]：A→ (A) | a

(1) 证明：LR(1) 项目 [A→ (A ·),)] 对活前缀“((a”是有效的；

(2) 画出 LR(1) 项目识别所有活前缀的 DFA；

(3) 构造 LR(1) 分析表；

(4) 合并同心集，构造 LALR(1) 分析表。

3.31 下述文法 G[S] 是哪类 LR 文法？构造相应 LR 分析表：

G[S]：(1) S→L=R

(2) S→R

(3) L→*R

(4) L→i

(5) R→L

3.32 已知布尔表达式的文法 G[B] 如下：

G[B]：B→AB | OB | not B | (B) | i rop i | i

A→B and

O→B or

试为 G[B] 构造 LR 分析表。

3.33 给出文法 G[S]：S→SaS | SbS | cSd | eS | f；

(1) 请证实这是一个二义文法；

(2) 给出什么样的约束条件可构造无冲突的 LR 分析表？请证实你的论点。

3.34 根据例 3.26 中的表 3.22 分析 ia;iaea#的语义加工过程。