电子计算机教学丛书

计算机程序设计语言的翻译

(英) R.E. Berrg 著

胡学联 译

樊国咸 校

JI SUAN JI CHENG XU SHE JI YU YAN DE FAN YI

安徽教育出版社

计算机程序设计语言的翻译

〔英〕R.E.Berry 著

胡学联 译 樊国咸 校

安徽教育出版社出版

（合肥市跃进路1号）

安徽省新华书店发行 安徽新华印刷厂印刷

*

开本：787×1092 1/16 印张：10 字数225,000

1986年12月第1版 1986年12月第1次印刷

印数：3,000

统一书号：7276·508 定价：1.70元

内 容 简 介

编译程序和汇编程序是计算机软件的一个极为重要的组成部分，本书较为系统地介绍了这方面的知识。全书分成两个部分。第一部分共九章，其中第一—第六章讲述一般的编译技术，第七—第九章讲述汇编程序，各章均附有习题。第二部分共两章，其中第十章给出了用Pascal语言书写的完整的Pascal S(它是标准Pascal的一个子集)编译程序和解释程序，并且做了较为详细的说明。第十一章讨论了Pascal S解释程序的实现。

本书浅显、易懂；理论与实际联系得很紧；所给出的Pascal S编译程序简短、紧凑、易读，是一个较为理想的编译程序教学模型。本书既可供高等学校，中等专科学校有关专业师生作为教学参考书，也是广大科技工作者的一本较为理想的参考书。

本书还适合广大计算机用户学习、了解编译程序和汇编程序时使用。

为配合本书的使用，书中所给的教学模型即pascal S编译程序，已在IBM—PC微型机上实现。

中 译 本 序

这本《程序设计语言的翻译》中译本的问世，对国内计算机界来说，的确是一件喜事。

翻译技术，特别是编译技术，是计算机软件工作者必需具备的基本知识，它是计算机软件专业的重要必修课程。其讲授途径大致有三。一种是着重讲授基本原理和技术，而不涉及具体的编译程序；另一种是着重讲授某一具体编译程序。前者可能使读者“见林而不见树”，后者反之，往往会使读者“见树而不见林”。《程序设计语言的翻译》一书的作者采取了第三种途径，一面讲解基本技术，同时又给出了一个可用的Pascal S编译程序，这样，便可使读者以Pascal S编译程序为工具，自行设计并实现各种编译算法，既便于提高实践能力，又便于加深对基本原理与方法的理解，从而，既可见树，又可见林。这是一种讲授编译技术成功的途径。

胡学联同志从事计算机软件教学与研究多年，学有根底，成绩斐然，近又译成是书，措词严谨，行文通畅，余乐而为之序。

徐家福

1985年仲春识于南京大学

原　序

本书介绍了在编译程序和汇编程序的构造中所用的一些较为重要的技术. 编译程序或汇编程序怎样处理用户编写的程序呢？日益增多的计算机用户迫切希望了解这方面的知识，为此，我编写了这本书. 为了便于初学者学习，书中所述是一些一般性的概念和方法，而不是对一些特殊论题进行详尽的研究. 我所给出的例子，大多取自Pascal S. 绝大多数Pascal编译程序都是用Pascal语言编写的. 然而，一个语言的编译程序用该语言本身来编写，这一特性并不只限于Pascal编译程序. 这种特性是十分有利于编译程序教学的. 若干年来的教学中，我在不同的环境下使用了Pascal S，实践表明，它作为教学媒介是很有价值的，因为它能引起学生的兴趣. 本书还给出了Pascal S编译程序和解释程序清单，这是经N. Wirth教授同意的，在此表示感谢.

我还感谢我的妻子、同事和一些大学生，他们直接或间接地帮助我准备了这份资料，然而，责任由我个人来负. 我特别感谢丛书编辑B. Meek，他对本书的初稿提出了细致、中肯、有益的意见.

R. E. Berry

引　言

一个程序的翻译程序(指编译程序或汇编程序)输入的是用户编写的程序，输出的是一个能由计算机直接或间接执行的程序文本．这种翻译处理应包括检查用户程序是否符合所用语言的语法，检查所有由用户定义的符号的使用是否前后一致，以及生成一个适合于执行或解释的程序文本.用户希望翻译程序产生一个程序，这个程序在执行时所产生的效果要与假设他的初始程序能被直接执行时所产生的效果相同．把一种形式的程序翻译成另一种形式的程序，这是一个复杂的任务．通过列举那些为完成这一任务所需要的技术来讲解翻译程序，而很少考虑指明这些技术之间的内部关系，这是十分通常的现象.为了避免这种弊病，我将讨论翻译处理的各个方面，并且指明它们在一个特定的翻译程序—Pascal S编译程序中的实现.为了分清主次，我把本书分成两个部分.第一部分考虑一般的翻译，但行文及例子均先行假定读者熟悉在第二部分中给出的编译程序.第二部分给出了Pascal S编译程序清单和文档资料.这两部分均假定读者熟悉Pascal语言.

Pascal S是Pascal的一个子集.因此，Pascal S编译程序比目前使用的许多Pascal编译程序要小一些．不过，还要求付出有效的努力，以便通过研究 Pascal S编译程序清单和所给的有用的文档资料，透彻地了解编译程序的作用．如果这种努力有助于读者理解一个复杂而重要的软件，那是很值得一试的．

目　录

中译本序
原　序
引　言

第一部分……(1)
第一章　词法分析……(1)
1.1　用户界面……(1)
1.2　字母表……(1)
第二章　语法定义和语法分析……(6)
第三章　符号表——结构与存取……(15)
第四章　运行时刻环境……(20)
4.1　过程过口，过程出口……(24)
第五章　语义处理……(32)
第六章　运行时刻支持……(38)
6.1　执行还是解释……(38)
第七章　汇编程序……(40)
7.1　标号字段……(41)
7.2　符号表的组织……(43)
7.3　操作符或指令字段……(43)
7.4　操作数字段……(44)
7.5　内部段通讯……(46)
第八章　宏指令……(49)
第九章　装入程序……(52)
第二部份……(55)
第十章　Pascal　S 编译程序……(55)
10.1　Pascal　S语法图……(55)
10.2　Pascal　S编译程序……(60)
10.3　过程描述……(64)
10.3.1　实用程序……(64)
10.3.2　词法分析……(65)
10.3.3　语法分析……(66)
10.3.4　语义程序……(74)
10.3.5　代码生成……(76)
10.4　例子……(124)
10.5　Pascal　S错误摘要……(127)

第十一章 Pascal S 解释程序……（130）
11.1 运行时刻栈……（130）
11.2 运行时刻DISPLAY的作用……（135）
11.3 Pascal S操作代码 ……（141）
索引……（143）
译者后记

第一部分

第一章　词法分析

1.1　用户界面

在考虑编译程序或汇编程序的工作时，假定所有那些项目都是以读者所熟悉的同样方式工作的，这就过于简单化了.

一个用户能以若干方式与系统软件的这些项目进行通讯.卡片、纸带及交互式终端都可以用来准备一个随后提交处理的程序.而且，卡片可以按不同方式编码，纸带可以具有不同的尺寸(8单位、7单位及当前不怎么通用的5单位)和不同的编码，而终端可以一次传送一行或一个字符以供处理.无论选择什么样的媒介，用户都只是为处理而提交他的程序，并且自然要期望他所选择的输入媒介的形式不会影响他的程序的完整性.同样，一个编译程序的大部分工作是判定一个提交处理的程序是否是一个符合所用程序设计语言规则的正确程序.无论使用什么输入媒介，都要进行这种判定.希望读者注意：从一个终端一次接收一个字符，将意味着把rub—out或者可能是backspace视为擦去前面的字符；相反，每次接收一行则没有这样的删除可进行.

有关各种输入媒介的一些问题，可参阅Hopgood与Bell[6].

1.2　字母表

根据上述观点，就要假定我们所存取的字符流和任何准备数据的设备无关.但在进一步处理之前，我们需要确定在编译处理时，哪些字符是合法输入.比如，一个Fortran编译程序必须标记字符“[”是非法字符，因为在Fortran语言中这个字符没有用处.编译程序接受大小写字母吗？一个Pascal编译程序能接受字符“{”和字符对“(*”作为注解的开始吗？这些问题必须解决，以便能辨认出所有那些不构成编译程序的合法输入部分的字符，并且把它们标记为错误.一旦认定了我们准备处理的字符集或字母表，再问我们对这些字符还要做些什么工作，就很自然了.

在词法分析中，我们只是简单地把用户源程序考虑成字符流.我们必须扫描这个输入字符流以找出能称为**正文单元**(textual elements)的那些字符.这些字符是：组成用户程序的单词，标点，单字符和多字符运算符，注解，空格等.在最简单的情形中，扫描程序(词法分析程序)找出这些字符组，对每一个都根据其正文单元进行分类，并为后续处理回送一个该初始程序的非常简化的形式.特别是，识别出来的注解应被掠过，空格、回车换行以及其它被辨别出来的编辑符号也都要掠过去.请注意，空格可能起重要作用，因为在某些语言中，特别是在PL/1和Cobol中，它们是界限符，亦即用它们标记象保留字这类正文单元的结束.大部分(虽然不是所有的)Pascal实现均采用这种方法.在其它一些语言的实现中，保留字可以前置和后置一个不是空格的特殊字符.

例如，在大多数ALGOL 60和ALGOL 68的实现中，begin表示为 'begin'. 不管使用什么样的分界符，扫描程序的作用都是汇集处于相邻分界符(空格字符只是分界符的一种)之间的那些字符，并判定该字符组是否是一个正文单元.如果是，则是属于哪一类. 一旦辨认出一个字符组是一个正文单元，则用一个记号(token)或内部符号替换该字符组，使得在后续处理中易于识别.记号取什么样的形式，是特定实现者考虑的事.但是，引用特殊表格连同一个位置编号，或者一个特殊的在一个字节或字中的字位模式，或者象本书中的Pascal S 那样，能给出一个在预先定义的符号集中的相应元素，这些都是可以考虑的形式. 为了帮助弄清记号的作用，考虑下列程序设计语言的语句：

```
if  x>y  then  x:=x-y  else  y:=y-x
```

该语言用这种形式表示一条语句，显然，对用户来说它是易读的也是易于理解的. 同一语句还可用下列形式等价地写出：

```
|x>y~x\x-y@y\y-x
```

这就不是那么易于理解. 但实际上这只是第一个例子中的由一个以上字符组成的复合符号(亦即 if, then, else, :=)，在第二个例子中都用单个字符进行了等价替换.简而言之，保留字被换成了记号.对此例，记号是直观的，但构造我们所用的扫描程序时，显然不必如此进行.

这类正文单元的识别以及用记号对它们进行替换，使得我们能够辨别出任何一个可能的保留字.这就意味着扫描程序必须能访问一个存放了所有保留字的表，并且能够正确地把一个字符组标识为该表的一个项.这个问题涉及到可以用几种不同方式进行的查表技术.暂时我们认为查表能够进行，具体如何进行，将在后面讨论.因此，我们可以假定，我们已经把每个保留字都简化成一个简单的记号了.

对于许多高级语言，一个能按直接方式处理的项是串.如果我们考虑Pascal语句：

```
write('this is a sting')
```

那么在撇号之间的若干个项的汇集是一个串.串中是什么字符或有多少字符通常并不重要，因为在大多数语言中，这样的串并不能被加工处理.因此，一个扫描程序在遇到一个串时，几乎只要把这个串存放在串表中，并且存放一个记号在程序源中，用以指出串的出现以及存放该串的位置，便表明已成功地识别了这个串.

在用户源程序中，非常频繁出现的那些正文单元是标识符或者用户定义的符号. 这些标识符所具有的字符个数是从一到某个实现或者语言定义的最大字符长度数. 因为在编译程序的其余部分，经常不断地涉及多字符标识符要耗费时间，因此让扫描程序为所有被识别出的标识符造一个表，并且在用户源程序中用一个记号连同一些足以指出在标识符表中该相应符号存放情况的信息来替换那些标识符，这是有益处的.应该注意的是，对于语言中的保留字，扫描程序试图在先前构造的表中查找相应的字符组；然而，在标识符的情况下，需要某些稍微不同的操作.首先必须扫描标识符表(或者我们当前所访问的标识符表的一个部分)，查看对于该字符组是否已经有一个登记项存在.如果已经存在，那很好；否则，就要产生一个登记项.我们不打算细叙诸如':='，'..'这类符号的处理(虽然它们包含一个以上的符号，但处理是容易的，因为它们数量少，可以作为特殊情况对待)，现在唯一未予讨论的多字符正文单元是数值常量.

数值常量分为两种类型：具有小数部分的数值常量和不具有小数部分的数值常量. 识别在正被处理的源程序中的每一个数值常量的合法表示是扫描程序的任务，它通常可以通

过使用一个**状态表**来完成.考虑使用这种技术的一个例子可以得到启示，但是此刻为了能够这样做，我们需预先涉及某些以后将会详细叙述的素材，并假定读者熟悉用如下形式给出的定义：

〈无正负号数〉::＝数字〈余留无正负号数〉
〈无正负号数〉::＝.〈十进小数〉
〈无正负号数〉::＝E〈指数部分〉
〈余留无正负号数〉::＝数字〈余留无正负号数〉
〈余留无正负号数〉::＝.〈十进小数〉
〈余留无正负号数〉::＝E〈指数部分〉
〈余留无正负号数〉::＝△
〈十进小数〉::＝数字〈余留十进小数〉
〈余留十进小数〉::＝△
〈余留十进小数〉::＝E〈指数部分〉
〈余留十进小数〉::＝数字〈余留十进小数〉
〈指数部分〉::＝正负号〈整指数〉
〈指数部分〉::＝数字〈余留整指数〉
〈整指数〉::＝数字〈余留整指数〉
〈余留整指数〉::＝数字〈余留整指数〉
〈余留整指数〉::＝△

这不是表示这些定义的最紧致的形式，但依据这种形式来构造表格的方法是最容易理解的.注意，括在尖括号中有七个不同的项，其中每一个都至少在所列定义的左端出现一次.这些项是：

1.无正负号数
2.余留无正负号数
3.十进小数
4.余留十进小数
5.指数部分
6.整指数
7.余留整指数

它们决定了状态个数或者表的行数. 列数是由终结符号的个数或我们需在所列定义中使用的字符个数决定的.注意：把〈正负号〉和〈数字〉看成终结符号，不会引起什么问题.符号△用于表示其它任何字符.根据这些信息，扫描程序所用的表格构造如下.

	正负号	数字	.	E	△
1	/	2	3	5	/
2	/	2	3	5	出口
3	/	4	/	/	/
4	/	4	/	5	出口
5	6	7	/	/	/
6	/	7	/	/	/
7	/	7	/	/	出口

第一行是〈无正负号数〉的登记项.在规则定义表中，该项在定义的左端出现三次.相应的右部要求识别出“数字”、“·”或“E”.对应于那些以这些字符为列首的列，在其下的第一行上要有登记项存在.对于每一种情况，其登记项均是在相应定义中该符号之后的括在尖括号中的项的行号.因此，在E下的第一行，登记项是5，它引用第5行，因为括在尖括号中的那个项编号是5，即指数部分.

该表格的使用是直接了当的，所要求的都是当前状态(行号)和当前输入流中的当前字符.这些信息用来确定表中的某个元素的位置，它是一个新的状态或行号.从状态1开始，可以看到：只有那些在处理的那一位置上可能得到的字符才能产生符合要求的状态变化；任何其它的字符都会使我们得到一个注明为/的登记项，它可以视为一个出错出口.只要被处理的语言中的浮点常量的定义能以象上面给出的那种规则集来重新定义，那就可以使用这种技术. 这是有效的.虽然它需要表空间，但可以产生好的诊断.

我们取3·141*作为被分析的字符串.开始状态是1，第一个字符是数字，由状态表的第一行可知新状态是2.在此状态下，我们考虑输入串的下一个字符，即小数点；在状态表的第二行中小数点之下的登记项是3，现在它是新状态.这个过程可用下表较为简要地表示出来：

当前状态	当前符号	新状态
1	3	2
2	·	3
3	1	4
4	4	4
4	1	4
4	*	出口

有穷状态表的唯一用途是帮助扫描程序识别一个特定项.在所描述的例子中，使用该表的结果，应能对“被处理的数是一个合法的浮点数吗？”这一问题给出“是”或“不是”的回答.一个扫描程序通常把这样一个项的识别和对它求值及转换成**内部形式**这些工作结合起来.然而，如果承担附加的工作，则必须对运行编译程序的计算机作出假定；这是因为浮点数的存放方式将随机器的不同而不同.最后，可以进一步劝你接受一个论点，即：在编译处理中，把这样的转换操作延迟到后面，使得该编译程序尽可能多的部分能够以独立于特殊机器的方式来编制.

就我们所讨论的扫描程序而言，假定一个浮点数的成功识别会导致用一个适当的记号和表的引用来替代该浮点数，这是合理的.请注意，整型常量也能用这个构造出来的表进行处理，因为它们也是无正负号常数.扫描程序的作用可以看成是接收字符流，即用户源程序作为输入，产生一个长度大大缩短了的由记号序列组成的串，连同一个或多个表格一起作为输出.然而这种简单的观点出自孤立地看待扫描程序.实际上，常常是当编译处理的其余部分需要时，便调用扫描程序.这可以通过考察Pascal S编译程序中所给出的NEXTCH和INSYMBOL所起的作用清楚地看出来.

练　习

1.指出并列出文中所叙有穷状态识别程序接收的浮点数与Pascal S语法图所定义的浮点数之间的不同之处.

2.描述Pascal S编译程序使用哪些记号，如果有的话，作为下列符号':=', '<>', '..', '(*', '<=' 的记号.

3.表达式的运算分量可以是单个数字或单个字符，所允许的运算符是+，－，*，/.分别为不含单目运算符的表达式和含单目运算符的表达式构造一个有穷状态识别程序.

4.在Pascal S中，一个注解处于另一个注解中是合法的吗？如果该编译程序处理具有这类结构的程序会是什么结果？

5.为使大、小写字符能作为编译程序的输入，对Pascal S编译程序需做哪些必要的修改？请列出来.修改应当保证能正确地识别例如BeGIN之类的输入.

第二章　语法定义和语法分析

编写一个计算机程序时，用户必须使用某种程序设计语言.大约在1950年以后，才有这样的语言.人们根据他们自己的需要设计出这些语言.我们的自然语言已经发展了几千年，从时间的角度看，程序设计语言即不那么丰富，也不象我们所希望的那样灵活，也就不足为奇了.然而，程序设计语言已经进展到这样的阶段，即非常多的地方都用到它们；而那些难以实现或难以使用的语言特色很少能通过语言的设计阶段.

设计一个语言时，对将要使用的字符集、把这些字符组合成符号的方式以及这些符号如何组合以产生该语言的句子的规则等，均要给以认真的考虑.编译程序有各种各样的方法检查一个程序在语法上是否正确.所谓语法上正确，也就是根据所用语言的规则进行正确的构造.如何表示决定语言语法的那些规则呢？考察一下这个问题，可以得到某些启发.考虑下述规则：

1.〈数〉::＝〈数目〉

2.〈数目〉::＝〈数目〉〈数字〉|〈数字〉

3.〈数字〉::＝0|1|2|3|4|5|6|7|8|9

注意，规则2可以简单地写成：

〈数目〉::＝〈数字〉{〈数字〉}

规则3只是定义了一个称为〈数字〉的项，它是字符0—9之一.记住，规则的目的是精确地、简要地定义如何构成新的符号. 因此，不能依赖于一个数字或一个数应该是什么这样的直观概念.规则2对于如何形成数目给出了两个候选.它可以是一个数字，或者是一个数目(因而也是一个数字)后面跟着一个数字.简单地说，一个数目可以包含一个或多个数字. 第一条规则只是指出数是一个数目. 括在尖括号中的项是非终结符号；那些数字本身是终结符号，亦即那些能由识别程序识别出来的字符.这三条规则构成了定义数的文法.它们是文法的规则，也常称作产生式，因为它们指明了为产生一规则左部的项需要做些什么. 我们称这种表示文法规则的形式为Backus Naur范式；它再加上它的某些变形，已在计算科学文献中广泛使用.在定义 Pascal 的报告〔6〕中， N.wirth 用BNF和语法图两种方式定义了Pascal语法，因此用户可以针对特定的语言构造来鉴别这两种形式的优缺点.

表示文法规则的这两种方式的目的均在于为语言使用者定义组合语言各成份的方式.然而，在编译程序中我们面临的问题是：考察一个符号序列并且试图判定这样一个序列是否能由该文法的规则合理地产生出来.如果答案为“是”，则该程序是有效的；否则，不是有效的.记住，我们在词法分析中已说过，诸如保留字，标识符以及数值常量这样一些项，都已用记号或单一符号替换过了，现在注意力可以集中到这些内部符号的最终组合是否构成一个有效程序的问题上.

此处，我们遵照 Gries[1](2.4节)，考虑一个和数的文法规则相关联的语法 树.由〈数〉的文法，可做如下推导：

〈数〉⟶〈数目〉　　(规则1)

〈数目〉⟶〈数目〉〈数字〉　　(规则2)

⟶〈数字〉〈数字〉　　(规则3)

⟶ 3〈数字〉　　（规则 3）

⟶ 3 4　　（规则 3）

（把⟶读成“产生”）

这表明34是能由该文法的规则产生出来的一个数.这种推导可用图来表示，见图2-1.

它是这样构造的：当一条规则要求一个项由另外的项（不管它是不是在尖括号中）进行简单替换时，向下画一条垂线.使用规则2的第一个候选，就要求一个项（〈数目〉）由两个项（〈数目〉〈数字〉）来替换.在这种情况下，水平线用于连接两个项.最后得到的图是一棵语法树.该树的分支是线的集合加上这些线下的结点（符号）.末端结点是那些其下不再有分支的结点. 从左到右读取末端结点就得到一个串，该串是通过用树表示规则的应用之后推导出来的（上例为34）.在编译的语法分析阶段，我们打算要做的工作是，确定一个正被处理的符号串是否能通过应用该程序设计语言的文法规则产生出来. 可以用两种不同方式来完成这个工作. 我们能够从树的顶端开始，试图通过严格地使用该文法的规则，正确地产生我们已知的那个符号串.第二种方法路子则完全相反：先考虑该符号串，探寻与该文法的某一条规则的右部相同的符号组，如果能够找到，则这些符号可以归约到（或替换成）相应规则的左部符号.通过反复应用这种归约处理，我们能最终到达一个单一符号.第一种方法称作自顶向下或下降分析；第二种方法称作自底向上或上升分析.为什么如此命名，看一看语法树就清楚了.这两种方法在编译程序中都已广泛地应用，它们的实现技术，近年来有明显的改进.因为 Pascal S 编译程序使用了自顶向下分析，我们在后面还将进行较多的讨论，因此这里我们只简短地考虑一下在实现自底向上分析中所涉及到的问题. 我们使用下列取自Bolliett[7]的文法：

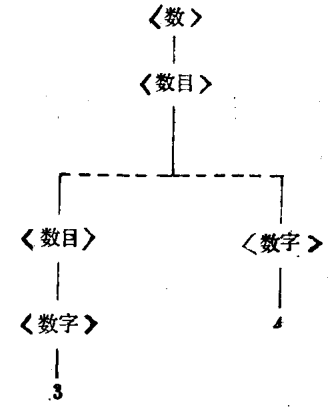

图2——1　一棵简单的语法树

〈P〉::=〈b〉

〈b〉::=l; 〈b〉|〈n〉|s

〈n〉::=〈t〉; 〈f〉

〈t〉::=begin d|〈t〉; d

〈f〉::=〈b〉; 〈f〉|〈b〉end

这里，为使文法易于理解，我们假定

〈p〉是程序

〈b〉是分程序

〈n〉是无标号分程序

〈t〉是分程序首部

〈f〉是分程序尾部

终结符号（我们期望在输入流中看到的符号）是begin, end, l（标号）, d（说明）, s（语句）. 我们试图识别出规则的右部，以把它们归约到左部的单个符号，因此按下列方法重写这些规则可带来方便：

1.　〈b〉⇒〈p〉

2.　l; 〈b〉⇒〈b〉

3. ⟨n⟩⇒⟨b⟩

4. ⟨t⟩; d⇒⟨t⟩

4a. ⟨t⟩; ⟨f⟩⇒⟨n⟩

5. **begin** d⇒⟨t⟩

6. ⟨b⟩; ⟨f⟩⇒⟨f⟩

6a. ⟨b⟩**end**⇒⟨f⟩

7. s⇒⟨b⟩

注意，在这一受限的文法中，有两条规则，其序列的开头有相同的两个符号(⟨t⟩;).事实上，这是一个一般性的问题.为了避免这个问题和其它能引起选择错误归约的问题，使用了一个后继矩阵.我们的目标当然是为了证实我们识别了一个程序(⟨p⟩)，为了成功地完成这个任务，途中必须达到各种子目标.因此，后继矩阵的作用是:使我们只做那些将达到当前子目标的归约.

	p	b	n	t	f
p	/	/	/	/	/
b	t	/	/	/	t
n	t	t	/	/	t
t	t	t	t	t	t
f	/	/	/	/	/
l	t	t	/	/	t
s	t	t	/	/	t
begin	t	t	t	t	t
d	/	/	/	/	/
end	/	/	/	/	/
:	/	/	/	/	/
;	/	/	/	/	/

注: t=true(真).

“后继矩阵为每一个符号(行首端)，终结符或非终结符，指明了在当前情况下，应用由该符号打头的规则是否能达到最终的或中间的目标(列首端).”(Bolliet[7])

下面，我们使用上面那个表和一些特定的规定，自底向上地分析符号串:

l; **begin** d; d; **begin** d; s **end end**

输入串那一栏中的最右符号是当前符号,在目标栈那一栏中的最右符号是目标栈的栈顶.假定无论何时下推一个子目标到栈中，有关分析的当前状态的信息同时保留.这意味着,当该子目标已达到时，通过返回到该状态，可使分析重新开始.第1步，我们的目标是⟨p⟩，当前输入符号是'l'.由'l'打头的唯一规则是规则2，因此该分析开始时试图识别出在输入串中能组成该规则的那些符号.

步骤	输入串	目标栈	规则
1	l	⟨P⟩	2l; ⟨b⟩ ↑
2	l;	⟨P⟩	2l; ⟨b⟩ ↑

'1'和其后的冒号在输入串中可以找到.

3　　1: begin ⟨P⟩　　21: ⟨b⟩

↑

冒号之后的输入符号并不匹配该规则中的相应符号.该规则要求非终结符号⟨b⟩，作为下一个成份.使用后继矩阵，查看目标⟨b⟩是否能够通过由当前符号begin打头的规则达到.答案为“是”，因此我们取⟨b⟩作为一个目标，并且找一条由begin打头的规则.

4　1: begin　　⟨P⟩⟨b⟩　　5begin d

↑

5　1: begin d　　⟨P⟩⟨b⟩　　5begin d

↑

规则5的最后的符号匹配当前输入符号.我们已经识别出⟨t⟩. 对应于规则5左部的输入串的部分用规则5的右部来替换，亦即把begin d归约到⟨t⟩.但我们的目标是⟨b⟩.我们能通过由⟨t⟩打头的某规则达到该目标吗？使用后继矩阵，可以看到答案为“是”，因此我们寻找一条由⟨t⟩打头的规则继续进行下去.

6　1: ⟨t⟩;　　⟨P⟩⟨b⟩　　4⟨t⟩; d

↑

7.　1: ⟨t⟩; d　　⟨P⟩⟨b⟩　　4⟨t⟩; d

↑

又一个⟨t⟩被识别出来了.同前面一样，我们再进行一次归约. 这个⟨t⟩还不是我们的目标，但它将通往我们的目标.为了继续进行下去，我们寻找一条由⟨t⟩打头的规则.

8　1: ⟨t⟩;　　⟨P⟩⟨b⟩　　4⟨t⟩; d

↑

9　1: ⟨t⟩; begin　　⟨P⟩⟨b⟩　　4⟨t⟩; d

↑

出现了一个匹配错误：输入符号是begin，但规则4要求一个'd'.规则4并未体现该输入串的这个部分. 还有另外的选择吗？在步骤8中，选用了规则4，因为要求有一条由⟨t⟩打头的规则.规则4a同样是由⟨t⟩打头的，因而分析从步骤8重新开始.这个新的步骤用8′表示.

8′　　1: ⟨t⟩;　　⟨P⟩⟨b⟩　　4a⟨t⟩; ⟨f⟩

↑

9′　　1: ⟨t⟩; begin　　⟨P⟩⟨b⟩　　4a⟨t⟩; ⟨f⟩

↑

我们再一次看到，当前输入符号不匹配所用规则的符号；然而这次情况是，该规则的符号是一个非终结符.因为当前输入符号能够达到该非终结符，所以我们取此非终结符作为一个新的子目标，寻找一条由当前输入符号打头的规则.

10　　1: ⟨t⟩; begin　　⟨P⟩⟨b⟩⟨f⟩　　5begin d

↑

11　　1: ⟨t⟩; begin d　　⟨P⟩⟨b⟩⟨f⟩　　5begin d

↑

该输入串的部分现能匹配规则5，因此可归约到⟨t⟩.因为⟨t⟩可以达到⟨f⟩，故找一条由⟨t⟩打头的规则.