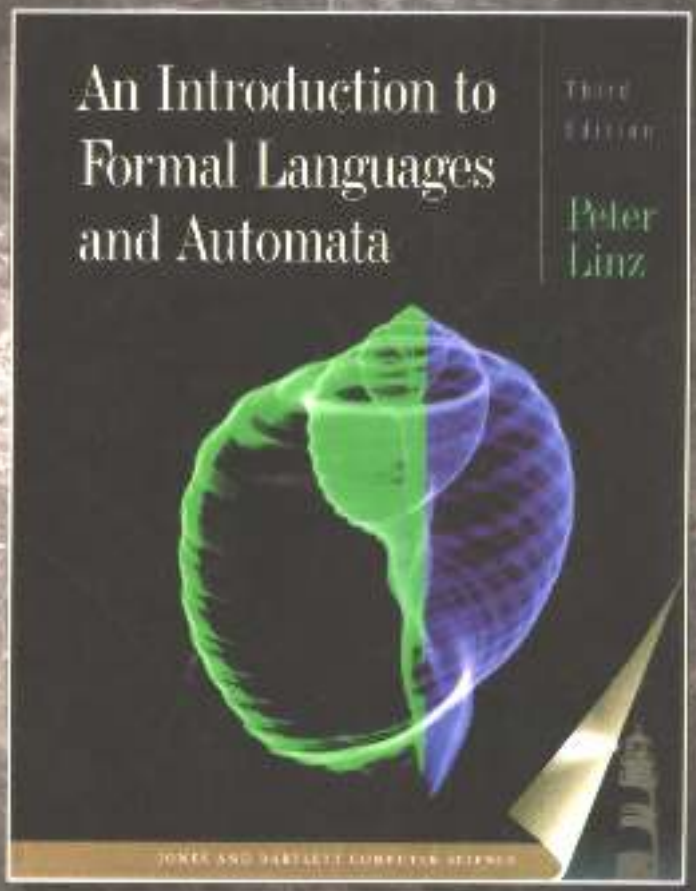
计算机科学丛书

原书第3版

形式语言与自动机导论

（美） Peter Linz 著　孙家骏 等译



An Introduction to
Formal Languages and Automata, Third Edition

机械工业出版社
China Machine Press

本书是理论计算机科学方面的优秀教材，主要介绍形式语言、自动机、可计算性和相关内容。本书特别注意定义、定理的准确性和严格性，在定理的证明中给出了直观的动机和框架，避免多余的数学细节，这有利于培养学生形式化和严格的数学推理能力，加强对问题的理解；本书通过精心设计的大量示例，生动剖析了各种定理和定义，概念清晰，深入浅出。每章后面还给出了难度不同的习题，并给出部分习题的解答，可使学生加深对基本原理的理解并增强应用能力。

作者简介

Peter Linz 加利福尼亚大学戴维斯分校计算机科学系的荣誉教授，研究重点是：开发数值分析理论，以构建可靠的数值方法并将其用于科学计算中的问题求解环境的设计。Linz教授的相关著作还有 *Exploring Numerical Methods: An Introduction to Scientific Computing*。

译者简介

孙家骕 北京大学软件学院教学指导委员会成员，北京大学信息科学技术学院计算机系教授、博士生导师。主要研究方向：计算机语言、程序理解、软件逆向工程、再工程。主持、参加过多项国家科研项目；发表论文30余篇；编写教材10余部；曾获北京大学优秀教学奖、机电部“七五”攻关荣誉证书。

ISBN 7-111-16788-0

9 787111 167884

华章图书

上架指导：计算机基础与计算理论/自动机

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

投稿热线：(010) 88379604

购书热线：(010) 68995259, 68995264

读者信箱：hzjsj@hzbook.com

ISBN 7-111-16788-0/TP・4335

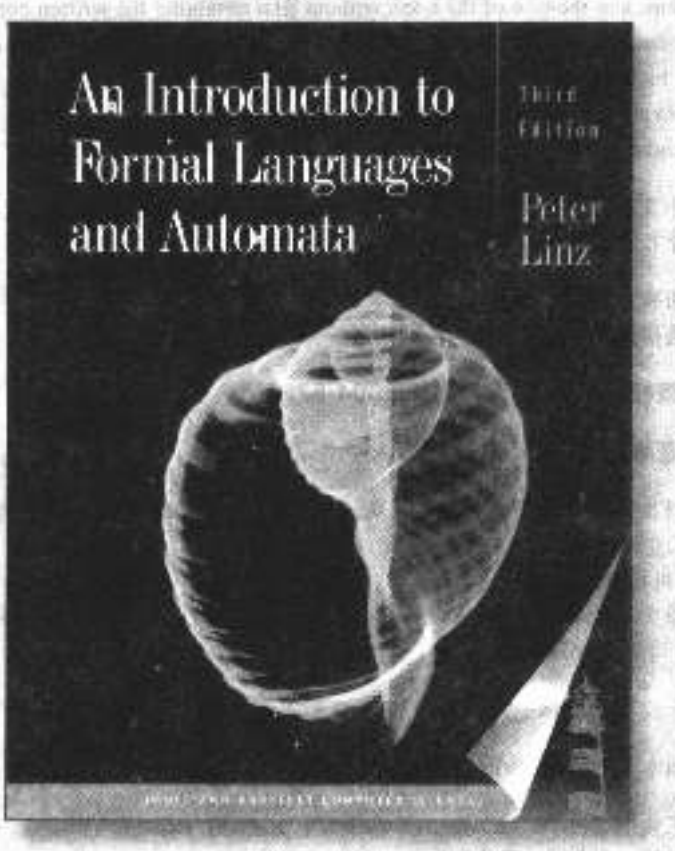
定价：36.00 元

计 算 机 科 学 丛 书

原书第3版

形式语言与自动机导论

(美) Peter Linz 著　孙家骕 等译



An Introduction to Formal Languages and Automata, Third Edition

机械工业出版社
China Machine Press

本书主要介绍形式语言、自动机、可计算性和相关内容。主要内容包括：计算理论导引、有穷自动机、正则语言与正则文法、上下文无关语言及文法、下推自动机、图灵机、形式语言和自动机的层次结构、计算复杂性等。每节后面都给出了习题，并包含部分习题的解答，方便教学。

本书是理论计算机科学方面的优秀教材之一，可作为高等院校计算机专业的教材，也可作为计算机系统研发人员的参考书。

Peter Linz: An Introduction to Formal Languages and Automata, Third Edition (ISBN 0-7637-1422-4).

Copyright © 2001 by Jones and Bartlett Publishers, Inc.

Original English language edition published by Jones and Bartlett Publishers, Inc., 40 Tall Pine Drive, Sudbury, MA 01776.

All rights reserved. No change may be made in the book including, without limitation, the text, solutions, and the title of the book without first obtaining the written consent of Jones and Bartlett Publishers, Inc. All proposals for such changes must be submitted to Jones and Bartlett Publishers, Inc. in English for his written approval.

Chinese simplified language edition published by China Machine Press.

Copyright © 2005 by China Machine Press.

本书中文简体字版由Jones and Bartlett Publishers, Inc.授权机械工业出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01-2004-5721

图书在版编目（CIP）数据

形式语言与自动机导论（原书第3版）/（美）林兹（Linz, P.）著；孙家骕等译. －北京：机械工业出版社，2005.9

（计算机科学丛书）

书名原文：An Introduction to Formal Languages and Automata, Third Edition

ISBN　7-111-16788-0

Ⅰ.形…　Ⅱ.①林…　②孙…　Ⅲ.①形式语言　②自动机理论　Ⅳ.TP301

中国版本图书馆CIP数据核字（2005）第070358号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：朱起飞　冯春丽

北京京北制版厂印刷 · 新华书店北京发行所发行

2005年9月第1版第1次印刷

787mm × 1092mm 1/16 · 18.75印张

印数：0 001 - 4 000册

定价：36.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下:

电子邮件: hzjsj@hzbook.com
联系电话: (010) 68995264
联系地址: 北京市西城区百万庄南街1号
邮政编码: 100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周克定	周傲英	孟小峰	岳丽华	范　明
郑国梁	施伯乐	钟玉琢	唐世渭	袁崇义
高传善	梅　宏	程　旭	程时端	谢希仁
裘宗燕	戴　葵			

译 者 序

理论计算机科学是推动计算机技术向前发展的强大动力。形式语言、自动机、可计算性和相关方面内容构成的计算理论，是理论计算机科学的基础内容之一。学习、研究这些内容，不仅为进一步学习、研究理论计算机科学所必需，而且对增强形式化能力和推理能力有重要作用，这些能力对从事计算机技术中的软件形式化等研究，是不可缺少的。

本书是由美国加利福尼亚大学戴维斯分校的Peter Linz教授编写的高等学校教材，主要介绍形式语言、自动机、可计算性和相关内容。本书特别注意定义、定理的准确性和严格性，这有利于培养学生形式化和严格的数学推理的能力；本书强调讲述问题、定理时的直觉性，这有利于学生对问题的理解；本书在定理的证明中往往只给出框架，而略去细节，这有利于学生进一步思考，加强对问题的理解；本书在每节后面都给出了习题，对部分习题还给出了解答，这有利于学生通过做习题加深对基本原理的理解并增强应用能力。本书是理论计算机科学的优秀教材之一。

本书可以用作计算机理论专业和计算机工程专业的教材，也可以作为计算机系统研发人员的参考书。

引进国外的优秀计算机教材，无疑会对我国的计算机教育事业的发展起积极的推动作用，也是与世界接轨、建立世界一流大学不可缺少的条件。我们把本书介绍给国内从事计算机教育事业的同行们，以供参考。

本书第1版于1990年发行，这个译本是我们根据它的第3版翻译的。参加本书翻译的有：孙家骕同志负责各章译稿的详细修改和全书的统稿；郝丹同志负责第1章到第5章的翻译；罗景同志负责第6章到第9章的翻译；李炎同志负责第11章到第14章的翻译；孙宏涛同志负责第10章、习题解答及索引的翻译。

由于我们的能力有限，译文中难免有不当之处，敬请读者不吝赐教。

译 者

2005年6月

前　言

本书主要介绍形式语言、自动机和可计算性等相关内容。这些内容形成计算理论的主体。这方面的课程现在是计算机科学课程体系中必有的一部分，并且这门课程通常开设在计算机课程的早期。因此，这本书面向的读者主要包括计算机科学系或计算机工程系大学二年级和三年级的学生。

学习本书的预备知识包括了解某种高级程序设计语言（通常包括C、C++或Java），熟悉数据结构和算法的基础知识。另一个预备课程是离散数学，包括集合论、函数、关系、逻辑和数学推理的基础。这门课程也是标准的计算机科学导论课程的一部分。

计算理论的研究有几个目的，其中最为重要的目的包括：（1）使学生熟悉计算机科学的基础和原则，（2）为后续课程做准备，（3）增强学生进行规范和严格数学推理的能力。尽管我在这本书中采用的表达方式主要是为了达到前两个目的，但是我认为这本书也达到了第三个目的。为了表达得更加清晰，使得读者能够认识事物的本质，本书强调直觉上的动机，并通过例子阐述这些想法。只要可以选择，我就会选择那些容易掌握的论据，而不是那些简洁、优雅，可是在概念上却难以理解的论据。我在这本书中准确地声明定义和定理，给出证明的动机，不过省略了程序和冗长的细节。这样做是出于教育学的考虑。许多证明是归纳或反证的一般应用，它们对于不同的特殊问题而言也不同。这样的详细论证不仅是不必要的，而且会影响到整体的流畅性。因此，我们略过很多论证的细节，可能会有人出于完整性的考虑而质疑这种做法，我本人并不认为这是一个缺陷。数学技巧无法通过阅读别人的论证过程而培养出来，而应该是考虑问题的本质，发现能够证明论点的想法，然后通过精确的细节实现它。这后一种技巧是必学的。我认为这本书的论证框架提供了一个这样实践的正确起点。

从事计算机科学研究的学生有时把计算理论方面的课程视为非必需的，且认为它缺乏实践意义。为了说服他们，我们需要吸引他们的兴趣和精力，比如处理难解问题的坚韧性和创造性。为此，我的方法强调通过解决问题来学习。

使用这种解决问题的学习方法，我想让学生主要通过按问题分类的示例来学习知识。正如它们同定理和定义之间的关联性一样，这种例子通过隐藏其后的概念可以显现其动机。同时，这些例子还有其不平凡的一面，它们有助于学生发现解决方案。在这种方法中，课后习题更有助于学习过程。每个章节后面的习题阐明和解释了本章的内容，在不同的层次上调动学生解决问题的能力。其中一些习题相当简单，挑选了一些书中没有完成讨论的内容让学生继续进行思考。另一些习题非常难，即使对比较聪明的学生也是一种挑战。这两种习题的适当结合会成为非常有效的教学手段。学生不必解决所有的问题，但是应该根据课程和教师的要求完成部分习题。不同的学校开设的计算机科学课程不同，有的强调理论方面，有的几乎完全面向实际应用。倘若课程选择考虑到了学生的背景和兴趣，那么我相信这本书可以用于上述任何学校。同时，教师需要告诉学生他们期望学生能够做到的抽象程度。对于面向论证的习题更是如此。当我说“证明”或“表明”时，我认为学生可以构造出一个论证过程，然后产生一个清晰的结论。一个论证过程要形式化到什么程度由教师决定，而且这种指导应该

在课程的早期给出。

这本书的内容适合在一学期内完成。教师可以讲授这本书的大部分内容，不过，重点就由教师自己决定了。尽管这本书中证明不多，不过我在我的班级里通常都是简略地讲述这些证明。我通常只给出足够的讲解，使结论可信，然后让学生独自去阅读剩下的部分。总的来说，如果不想以后碰到困难，还是尽可能不要跳过某些章节。标有星号的章节可以略过，略过这部分内容不会影响你读后面的内容。尽管如此，大部分的内容还是必需的，而且不能够跳过。

这本书的第1版出版于1990年，第2版出版于1996年。对新版本的需求是令人高兴的，这表明我的这种通过语言而不是计算的教学方法是可行的。第2版相对于第1版的变化不是彻底的变化，而是在原有基础上的改进。它修改了第1版中晦涩和错误的内容。无论如何，第2版看起来已经相对稳定，需要进行修改的地方也很少。所以，第3版的大部分内容和第2版差不多。第3版主要的新特点是增加了一些带解答的习题。

最初，我并不愿意给出习题的解答，因为这样就会减少可以用来作为课后练习的习题数量。然而，几年来，我收到来自世界各地学生的求助请求，因此我决定在这一版中给出一部分习题的解答。我又增加了一些新的习题从而保证没有解答的习题的数量。我只挑选那些有重要指导意义的习题来给出解答。因此，我没有仅仅给出最终的答案，而是给出最终结果的推理依据。许多习题使用了同样的原理，我通常从中选择具有代表性的习题给出解答，希望学生能够举一反三。我相信挑选出来的这部分习题的解答可以帮助学生提高他们解决问题的能力，并且保留一部分比较好的习题给教师作为课后作业使用。在本书中，有解答和提示的习题都标有 ◉ 。

此外，为响应读者的建议，我把一些比较难的习题做了标注。这样做不太容易，毕竟习题跨越的难度范围很大，而且同样一道习题对于某个学生而言简单，另一个学生可能就会认为很难。但是仍然存在一部分的习题，它们对于我的大多数学生而言都是有挑战性的。这部分的习题我用一个星号（★）标明。当然还存在一些特殊的习题，它们没有明确的答案。要想解决它们，可能需要思索，阅读一些附加内容，或者需要一些计算机编程。虽然它们不适合作为常规的作业来布置，但是它们可以作为进一步学习的切入点。这种习题我标了双星号（★★）。

在过去的10年里，我得到了很多评论家、教师和学生的有益建议。这样的人太多，我在这里无法一一列举他们的名字。我衷心感谢他们对我的帮助。他们的反馈对于我改进这本书而言是极其宝贵的。

Peter Linz

目　录

出版者的话
专家指导委员会
译者序
前言

第1章　计算理论导引 ……1
1.1　数学预备知识和表示 ……2
1.1.1　集合 ……2
1.1.2　函数和关系 ……3
1.1.3　图和树 ……5
1.1.4　证明方法 ……7
1.2　三个基本概念 ……10
1.2.1　语言 ……11
1.2.2　文法 ……13
1.2.3　自动机 ……18
1.3　一些应用* ……21
第2章　有穷自动机 ……27
2.1　确定型有穷接受器 ……27
2.1.1　确定型接受器和转换图 ……27
2.1.2　语言和dfa对应的语言 ……29
2.1.3　正则语言 ……32
2.2　非确定型有穷接受器 ……35
2.2.1　非确定型接受器的定义 ……35
2.2.2　为什么需要非确定型 ……38
2.3　确定型有穷接受器和非确定型有穷接受器的等价性 ……40
2.4　减少有穷自动机中状态的化简* ……45
第3章　正则语言与正则文法 ……51
3.1　正则表达式 ……51
3.1.1　正则表达式的形式化定义 ……51
3.1.2　和正则表达式相关的语言 ……51
3.2　正则表达式和正则语言之间的联系 ……55
3.2.1　正则表达式表示正则语言 ……55
3.2.2　正则语言的正则表达式 ……56
3.2.3　描述简单模式的正则表达式 ……59
3.3　正则文法 ……62
3.3.1　右线性文法和左线性文法 ……62
3.3.2　右线性文法生成正则语言 ……63
3.3.3　正则语言的右线性文法 ……64
3.3.4　正则语言和正则文法的等价性 ……66
第4章　正则语言的性质 ……69
4.1　正则语言的封闭性质 ……69
4.1.1　简单集合运算的封闭性 ……70
4.1.2　其他运算的封闭性 ……71
4.2　正则语言的基本问题 ……77
4.3　识别非正则语言 ……78
4.3.1　使用鸽巢原理 ……79
4.3.2　泵引理 ……79
第5章　上下文无关语言 ……85
5.1　上下文无关文法 ……85
5.1.1　上下文无关语言的例子 ……86
5.1.2　最左推导和最右推导 ……87
5.1.3　推导树 ……88
5.1.4　句型和推导树之间的关系 ……89
5.2　分析和二义性 ……92
5.2.1　分析和成员资格判定 ……92
5.2.2　文法和语言的二义性 ……95
5.3　上下文无关文法和程序设计语言 ……99
第6章　上下文无关文法的化简与范式 ……101
6.1　文法变换方法 ……101
6.1.1　一个有用的代入规则 ……101
6.1.2　删除无用产生式 ……103
6.1.3　消除λ产生式 ……106
6.1.4　消除单位产生式 ……107
6.2　两个重要的范式 ……111
6.2.1　乔姆斯基范式 ……112
6.2.2　格里巴克范式 ……114
6.3　上下文无关文法的成员资格

判定算法* ……116
第7章 下推自动机 ……119
7.1 非确定型下推自动机 ……119
7.1.1 下推自动机的定义 ……120
7.1.2 下推自动机接受的语言 ……121
7.2 下推自动机与上下文无关语言 ……125
7.2.1 上下文无关语言相应的下推自动机 ……125
7.2.2 下推自动机相应的上下文无关文法 ……129
7.3 确定型下推自动机和确定型上下文无关语言 ……133
7.4 确定型上下文无关语言的文法* ……136
第8章 上下文无关语言的性质 ……141
8.1 两个泵引理 ……141
8.1.1 上下文无关语言的泵引理 ……141
8.1.2 线性语言的泵引理 ……144
8.2 上下文无关语言的封闭性质和判定算法 ……146
8.2.1 上下文无关语言的封闭性质 ……146
8.2.2 上下文无关语言的可判定性质 ……149
第9章 图灵机 ……153
9.1 标准图灵机 ……153
9.1.1 图灵机的定义 ……153
9.1.2 作为语言接受器的图灵机 ……157
9.1.3 作为转换器的图灵机 ……160
9.2 完成复杂任务的组合图灵机 ……164
9.3 图灵论题 ……168
第10章 图灵机的其他模型 ……171
10.1 对图灵机的较小修改 ……171
10.1.1 自动机类的等价性 ……171
10.1.2 带不动选择的图灵机 ……172
10.1.3 单向无穷带图灵机 ……174
10.1.4 离线图灵机 ……175
10.2 具有更复杂存储的图灵机 ……177
10.2.1 多带图灵机 ……177
10.2.2 多维图灵机 ……179
10.3 非确定型图灵机 ……181
10.4 通用图灵机 ……183
10.5 线性有界自动机 ……186
第11章 形式语言和自动机的层次结构 ……189
11.1 递归语言和递归可枚举语言 ……189
11.1.1 非递归可枚举的语言 ……190
11.1.2 非递归可枚举语言 ……191
11.1.3 递归可枚举但非递归的语言 ……192
11.2 无限制文法 ……193
11.3 上下文相关文法和语言 ……198
11.3.1 上下文相关语言和线性有界自动机 ……198
11.3.2 递归语言和上下文相关语言的关系 ……199
11.4 乔姆斯基层次结构 ……201
第12章 算法计算的限制 ……205
12.1 图灵机所不能解决的问题 ……205
12.1.1 可计算性和可判定性 ……205
12.1.2 图灵机停机问题 ……206
12.1.3 将一个不可判定问题简化成另外一个问题 ……208
12.2 递归可枚举语言的不可判定问题 ……211
12.3 波斯特对应问题 ……213
12.4 上下文无关语言的不可判定问题 ……218
第13章 其他的计算模型 ……223
13.1 递归函数 ……224
13.1.1 原始递归函数 ……225
13.1.2 Ackermann函数 ……227
13.1.3 μ递归函数 ……228
13.2 波斯特系统 ……229
13.3 重写系统 ……232
13.3.1 矩阵文法 ……232
13.3.2 马尔科夫算法 ……233
13.3.3 L系统 ……234
第14章 计算复杂性介绍 ……237
14.1 计算的效率 ……237
14.2 图灵机和复杂性 ……239
14.3 语言族和复杂性类 ……241
14.4 复杂性类P和NP ……243
部分习题的解答和提示 ……247
参考文献 ……283
索引 ……285

第1章 计算理论导引

计算机科学是一门实践学科。从事这一领域的人经常是那些偏爱解决理论中有用而切实的问题的人。同样地，计算机科学方向的学生主要对解决现实世界中的困难感兴趣。只有当理论问题有助于他们找到好的解决方案的时候，他们才对理论问题感兴趣。这种态度是正确的，因为没有应用，谁还会对计算机感兴趣呢。但是，既然认为计算机科学是面向实践的，那么人们不禁会问“为什么要研究理论？”

第一，理论提供有助于理解学科一般本质的概念和原则。计算机科学领域包含了广泛的特殊课题，从机器设计到编程。在现实世界中成功地应用计算机需要学习大量特殊的细节。这使得计算机科学成为一门涉及知识宽泛的学科。尽管计算机科学涉及多方面的知识，可是仍然存在一些通用的基本原则。为了研究这些基本原则，我们构造了计算机和计算的抽象模型。这些模型体现了硬件和软件的一些共同的重要特点，并且和我们在使用计算机工作的过程中碰到的许多特殊和复杂的构造是本质相关的。即使当这些模型太简单而不能立即应用到现实世界中时，我们仍然可以通过研究它们获得一些知识，为特殊的应用提供基础。这种方法当然不是仅仅应用在计算机科学中。模型构造是任何科学学科的本质之一，并且一门学科的有效性往往取决于其简单而强大的理论和规律。

第二，我们讨论的想法有即时和重要的应用，这点或许不是很明显。但是数字设计 、程序设计语言和编译器都是最好的例子，当然还存在着很多其他例子。我们这里研究的概念像线一样贯穿整个计算机科学，从操作系统到模式识别。

第三，是我们想要说服读者。这个学科在智力上充满刺激和乐趣。它提出了许多有挑战性的、让人迷惑不解的题目，这些题目能让人废寝忘食。本质上，这就是解决问题。

在这本书中，我们关注那些能够表现所有计算机和它们的应用特点的模型。为了能够给计算机的硬件建模，我们引入了*自动机*（automaton）（复数：automata）的表示。自动机的构造包含了数字计算机所有必不可少的特点。它接受输入，产生输出，可能还有临时存储空间，并在把输入转化成输出的过程中做出决定。*形式语言*（formal language）是对程序设计语言的一般特点的抽象。形式语言包括符号集和把符号组成称为句子的实体的构成规则。一种形式的语言是所有符合构成规则的符号串的集合。尽管我们这里研究的某些形式语言，它们要比程序设计语言简单，但是它们有着相同的本质特点。通过形式语言，我们可以获得很多关于程序设计语言的知识。最后，我们通过*算法*（algorithm）一个精确的定义来把机械计算的概念形式化，研究哪些问题适合用机械方式解决，哪些不适合。学习的过程中，我们将展示这些抽象表示之间的密切关系，研究我们从中可以获得的结论。

在第1章中，我们从一个宽广的视角去看这些基本想法，从而为后面的内容做准备。在1.1节中，我们复习必需的数学基本知识。因为直觉经常成为我们探究问题的指南，所以我们得出的结论应该以严格的推理为基础。这就要涉及很多数学工具，但是并不会涉及得过多。读者需要掌握集合论、函数和关系的术语及基本结论。书中还会经常用到树结构和图结构，不

2 过仅仅需要知道带标记图和有向图的定义。或许最重要的要求是能够理解证明和形成一个合理的数学推理。这包括熟悉演绎、归纳和反证法的基本证明技巧。我们假设读者已经具备了这些必需的背景知识。1.1节用来复习一些在后面的章节中使用的主要结论，并建立符号表示基础。

在1.2节中，我们首先关注语言、文法和自动机的核心概念。这些概念以各种不同的形态出现在书中。在1.3节中，我们给出了这些基本概念的一些简单应用，从而解释这些概念在计算机科学中的广泛应用。这两节的讨论是直观的，并不严格。在后面，我们会使这些概念更准确。但是暂时这样做的目的是为了更加清晰地描述我们要用到的概念。

1.1 数学预备知识和表示

1.1.1 集合

集合（set）是元素的组合，除去成员资格关系，无其他结构。我们用$x \in S$表示x是集合S的元素，$x \notin S$表示x不是集合S中的元素。集合在大括号内列出它的元素，例如，整数0，1，2的集合表示成

$$S = \{0, 1, 2\}$$

当集合含义清楚时，我们可以用省略号和其内的元素来表示集合。因此，$\{a, b, \cdots, z\}$代表所有的英文小写字母，而$\{2, 4, 6, \cdots\}$代表所有的正偶数。如果需要的话，我们会更多地使用直接表示，对于上面最后一个例子，我们可以写成

$$S = \{i : i > 0，i\text{是偶数}\} \tag{1-1}$$

通过这个式子我们可以知道“S是所有i的集合，其中，i大于零且为偶数”。当然这里暗示i是整数。

通常的集合运算包括并（union，表示成$\cup$）、交（intersection，表示成$\cap$）和差（difference，表示成$-$），它们的定义分别是

$$S_1 \cup S_2 = \{x : x \in S_1 \text{或} x \in S_2\}$$
$$S_1 \cap S_2 = \{x : x \in S_1 \text{且} x \in S_2\}$$
$$S_1 - S_2 = \{x : x \in S_1 \text{且} x \notin S_2\}$$

3 另一个基本运算是补（complementation）。集合S的补表示成$\overline{S}$，包含所有不在集合S中的元素。为了使这个概念有意义，我们首先需要知道全集（universal set）U包含哪些元素。如果给定U，那么

$$\overline{S} = \{x : x \in U, x \notin S\}$$

没有元素的集合称为空集（empty set或null set），表示成$\emptyset$。根据集合的定义，显然有

$$S \cup \emptyset = S - \emptyset = S$$
$$S \cap \emptyset = \emptyset$$
$$\overline{\emptyset} = U$$
$$\overline{\overline{S}} = S$$

下面这些有用的恒等式被称为德摩根定律（DeMorgan's laws），在有些情况下需要使用它们:

$$\overline{S_1 \cup S_2} = \overline{S}_1 \cap \overline{S}_2 \tag{1-2}$$

$$\overline{S_1 \cap S_2} = \overline{S}_1 \cup \overline{S}_2 \tag{1-3}$$

如果集合S_1的元素都是集合S的元素，那么集合S_1是集合S的子集（subset），记作

$$S_1 \subseteq S$$

如果$S_1 \subseteq S$，但是S包含一个不是S_1中的元素，那么S_1是S的真子集（proper subset），记作

$$S_1 \subset S$$

如果S_1和S_2没有共同的元素，即$S_1 \cap S_2 = \varnothing$，那么，这两个集合称为不相交（disjoint）。

如果集合包含有限个元素，那么这个集合是有限的；否则，就是无限的。一个有限集合的大小指的是它包含的元素个数，记作$|S|$。

一个给定的集合通常有很多子集。集合S的所有子集的集合称作集合S的幂集（powerset），记成2^S。注意2^S是集合的集合。

例1.1 如果集合$S = \{a, b, c\}$，那么它的幂集就是

$$2^S = \{\varnothing, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$$

这里$|S| = 3$，$|2^S| = 8$。这是一个例子的一般结果。如果S是有限的，那么

$$|2^S| = 2^{|S|}$$ □ 4

在我们的很多例子中，一个集合的元素是其他集合元素的有序排列。这个集合称为其他集合的笛卡儿积（Cartesian product）。两个集合的笛卡儿积是有序对的集合，表示成

$$S = S_1 \times S_2 = \{(x, y) : x \in S_1, y \in S_2\}$$

例1.2 设$S_1 = \{2, 4\}$，$S_2 = \{2, 3, 5, 6\}$。那么

$$S_1 \times S_2 = \{(2,2),(2,3),(2,5),(2,6),(4,2),(4,3),(4,5),(4,6)\}$$

注意上述书写的元素对的顺序。有序对（4, 2）属于集合$S_1 \times S_2$，但是（2, 4）就不属于。这种表示显然可以扩展到多个（大于两个）集合的笛卡儿积。通用的定义为

$$S_1 \times S_2 \times \cdots \times S_n = \{(x_1, x_2, \cdots, x_n) : x_i \in S_i\}$$ □

1.1.2 函数和关系

函数（function）是建立一个集合的元素和另一个集合的唯一的一个元素对应关系的规则。如果f表示一个函数，那么第一个集合称为函数f的定义域（domain），第二个集合称为它的值域（range）。我们用

$$f : S_1 \to S_2$$

表示函数f的定义域是集合S_1的子集，值域是集合S_2的子集。如果函数f的定义域就是集合S_1本

身，我们就说函数f是集合S_1上的全函数（total function）；否则就是部分函数（partial function）。

在许多应用中，函数的定义域和值域是正整数集合。而且，我们经常只对函数的自变量变大时所表现出来的函数行为感兴趣。在这种情况下，理解变化率就足够了，我们只需要使用普通的数量级符号表示。假设函数$f(n)$和$g(n)$的定义域是正整数的子集。如果存在一个正整数c满足：对于所有的n，

$$f(n) \leqslant cg(n)$$

那么我们就说g是f的最大阶（order at most），记作

5

$$f(n) = O(g(n))$$

如果

$$|f(n)| \geqslant c|g(n)|$$

那么g是f的最小阶（order at least），记作

$$f(n) = \Omega(g(n))$$

最后，如果存在常数c_1和c_2，并且

$$c_1|g(n)| \leqslant |f(n)| \leqslant c_2|g(n)|$$

则称f和g等价（same order of magnitude），表示成

$$f(n) = \Theta(g(n))$$

在数量级表示中，我们忽略乘法的常数，以及随着n增大可以忽略的低阶项。

例1.3 设

$$f(n) = 2n^2 + 3n$$
$$g(n) = n^3$$
$$h(n) = 10n^2 + 100$$

那么

$$f(n) = O(g(n))$$
$$g(n) = \Omega(h(n))$$
$$f(n) = \Theta(h(n))$$

在数量级表示中，＝号不应该被解释成相等，数量级表达式不能像普通的表达式那样对待。诸如这样的写法

$$O(n) + O(n) = 2O(n)$$

就不是很明智，很可能会导致错误的结论。然而如果使用正确的话，数量级的命题是有效的，这一点我们可以在后面关于算法分析的章节中看到。 □

一些函数被表示成有序对的集合

$$\{(x_1, y_1), (x_2, y_2), \cdots\}$$

这里，x_i是函数定义域的元素，y_i是函数值域中对应的元素。使用这样的集合来定义函数，任

意x_i至多出现在有序对的第一个元素处一次。如果不满足这一点，那么这个集合称为关系（relation）。关系比函数的概念更宽泛：在函数中，定义域中的每一个元素在值域中都恰好有一个关联元素；在关系中，可能有多个值域中的元素与之相关联。 6

另一种特殊的关系是等价（equivalence），它就是通常的恒等关系。表示有序对（x,y）是等价关系可以写成

$$x \equiv y$$

如果一个标有≡的关系满足下面三个规则，那么这个关系是等价关系：

自反性规则

$$\text{对于所有的}x\text{，都有}x \equiv x$$

对称性规则

$$\text{如果}x \equiv y\text{，那么}y \equiv x$$

传递性规则

$$\text{如果}x \equiv y\text{并且}y \equiv z\text{，那么}x \equiv z$$

例1.4 定义在非负整数集合上的关系

$$x \equiv y$$

成立，当且仅当

$$x \bmod 3 = y \bmod 3$$

根据上面的定义可以得到$2 \equiv 5$，$12 \equiv 0$和$0 \equiv 36$。很明显，这是一个等价关系，因为它满足自反性、对称性和传递性。 □

1.1.3 图和树

一个图（graph）包括两个有限集合，顶点（vertex）集合$V = \{v_1, v_2, \cdots, v_n\}$和边（edge）集合$E = \{e_1, e_2, \cdots, e_m\}$。每条边由顶点集$V$中的一对顶点构成，例如

$$e_i = (v_j, v_k)$$

是一条从顶点v_j到顶点v_k的边。我们认为边e_i对于v_j而言是输出边，对于v_k而言是输入边。这种构造实际上是有向图（digraph），因为我们把每条边都关联了一个方向（从v_j到v_k）。图也可以被标记，这个标记可以是名字或和图的部分相关联的其他信息。顶点和边都可以被标记。 7

图的这种结构通常以图表的方式来表现。在图表中，顶点表示成圆圈，边表示成连接顶点的带箭头的直线。图1-1中描绘的图包含顶点集合$\{v_1, v_2, v_3\}$和边集合$\{(v_1, v_3), (v_3, v_1), (v_3, v_2), (v_3, v_3)\}$。

一个边的序列$(v_i, v_j), (v_j, v_k), \cdots, (v_m, v_n)$称为从$v_i$到$v_n$的通道（walk）。通道的长度指的是从起点到终点经过的边的数目。没有重复边的通道称为路径（path）。没有重复顶点的路径成为简单（simple）路径。一个从顶点v_i出发又回到该顶点的无重复边的通道称为以v_i为始点（base）的回路（cycle）。如果回路中除了作为始点的顶点以外没有重复的顶点，那么这个回路是简单的。在图1-1中，$(v_1, v_3), (v_3, v_2)$是从v_1到v_2的简单路径。边序列$(v_1, v_3), (v_3, v_3), (v_3, v_1)$是回路，

但不是简单回路。如果图中的边被标记，那么我们就可以谈论通道的那个标记了。这个标记就是遍历路径时经过的边的顺序标号。最后，一条从某个顶点到它自身的边称作环（loop）。在图1-1中，顶点v_3有个环。

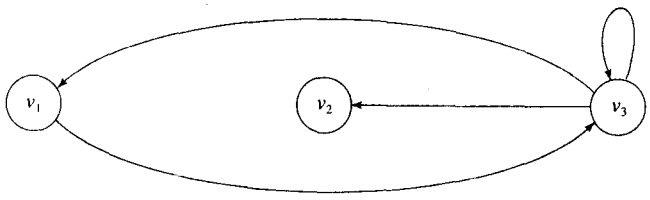


图 1-1

在几种情况下，我们会用到一个寻找两个给定顶点间所有简单路径（或以某一个顶点为始点的所有简单回路）的算法。如果不考虑效率的话，我们可以使用下面这个方法。从某个给定的顶点（假设是v_i）出发，列出所有的输出边 (v_i, v_k), (v_i, v_l), …这样，我们就获得了所有起点为v_i，长度为1的路径。对于所有这样到达的顶点v_k, v_l, …，只要输出边的另一个顶点不是已经构造出的路径中的某个顶点，我们就可以列出所有的输出边。之后，我们就获得了所有始点为v_i，长度为2的简单路径。继续上述过程直到不可能增加新的顶点为止。因为顶点的数目是有限的，所以我们最终会列出所有起点是v_i的简单路径。从这些路径中，我们选出那些以给定的另一顶点为终点的路径。

树是一种特殊的图。树是一个没有回路的有向图。在这个图中，有一个唯一的顶点，称为*根结点*（root），从根结点到任何一个其他顶点只有唯一的一条路径。这个定义告诉我们，根结点是没有输入边的，而且树中存在一些没有输出边的顶点。这些没有输出边的顶点就是树的*叶结点*（leave）。如果从顶点v_i到顶点v_j存在边，那么v_i就是v_j的*父结点*（parent），v_j是v_i的*子结点*（child）。每个结点的*层数*（level）指的是从根结点到该结点的路径的边数。树的*高度*（height）指的是所有结点的最大层数。这些术语在图1-2中以图例的方式进行了解释。

8

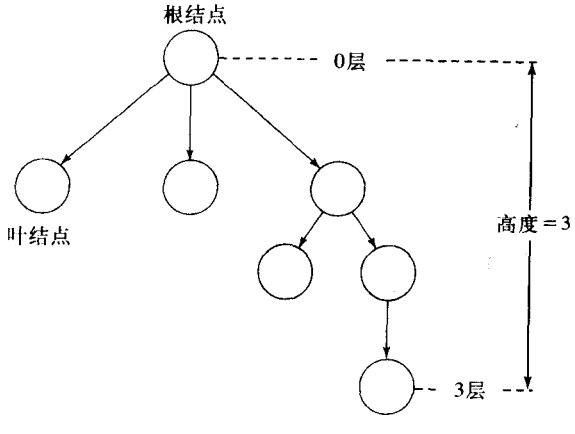


图 1-2

有时，我们想要把每层的结点和一定的顺序相联系，这种情况的树，称为*顺序树*（ordered tree）。

关于图和树的更多知识可以在离散数学书中获得。

1.1.4 证明方法

阅读这本书的一个重要目的是培养证明问题的能力。在数学证明中，我们应用演绎推理中许多已被接受的规则，而且许多证明只不过是这些规则的一个序列。有两种特殊的证明方法经常被使用，因此有必要先简要地复习一下它们。它们是*归纳证明*（proof by induction）和*反证法*（proof by contradiction）。

归纳证明是根据一些特殊的例子的正确性来推出更多命题的正确性的方法。假设我们需要证明一个命题序列$P_1, P_2, \cdots$是正确的。而且假设下面的条件成立:

1. 对于某个$k \geqslant 1$，我们已知$P_1, P_2, \cdots, P_k$是正确的。
2. 对于任何$n \geqslant k$，$P_1, P_2, \cdots, P_n$正确都有P_{n+1}正确。

我们可以使用这种归纳的方法来证明这个序列中的每一个命题都是正确的。

在归纳证明中，我们是这样证明的：根据条件1，我们知道前k个命题都是正确的。然后条件2告诉我们P_{k+1}也一定是正确的。既然我们知道前$k+1$个命题是正确的，我们应用条件2就可以证明P_{k+2}也一定是正确的，依此类推。我们不必详细地完成这个证明过程，因为这个证明模式很清楚。这样的系列推理可以扩展到任何命题。因此，每个命题都是正确的。 9

开始的这些命题$P_1, P_2, \cdots, P_k$是归纳的*基础*（basis）。从P_n到P_{n+1}的步骤之间的联系是*归纳步骤*（inductive step）。使用$P_1, P_2, \cdots, P_n$是正确的*归纳假设*（inductive assumption），使得归纳步骤变得更加容易。在形式化归纳证明中，我们更加明确地表现这三部分。

例1.5 二元树指的是不存在父结点有两个以上的子结点的特殊树。证明高度为n的二元树至多有2^n个叶结点。

证明：如果我们将一个高度为n的二元树的最大的叶结点数表示为$l(n)$，那么我们需要证明的就是

$$l(n) \leqslant 2^n$$

归纳基础：因为高度为0的树除了根结点以外没有其他结点，即：这棵树至多有一个叶结点。所以，显然$l(0) = 1 = 2^0$。

归纳假设：对于$i = 0, 1, \cdots, n$，有$l(i) \leqslant 2^i$。

归纳步骤：由一个高度为n的二元树构造一个高度为$n+1$的二元树。我们最多可以用两个叶结点来代替以前的每一个叶结点。因此，

$$l(n+1) = 2l(n)$$

这样，使用归纳假设就可以得到

$$l(n+1) \leqslant 2 \times 2^n = 2^{n+1}$$

因此，我们证明了$n+1$时结论也是正确的。既然n可以是任何数，这个命题对于所有的n都成立。■ □

我们在这里引进■符号来表示证明的结束。

例1.6 证明

$$S_n = \sum_{i=0}^{n} i = \frac{n(n+1)}{2} \tag{1-4}$$

首先，我们已知

10

$$S_{n+1} = S_n + n + 1$$

然后我们建立归纳假设，认为式（1-4）对于S_n成立。如果是这样的话，就有

$$\begin{aligned} S_{n+1} &= \frac{n(n+1)}{2} + n + 1 \\ &= \frac{(n+2)(n+1)}{2} \end{aligned}$$

因此式（1-4）对于S_{n+1}成立。至此我们已经完成了归纳步骤的证明。由于式（1-4）对于$n = 1$是显然成立的，因此我们有了归纳基础，至此，通过归纳证明了对于所有的n都成立。 □

在最后的例子中，我们在验证归纳基础、归纳假设和归纳步骤上都不是十分形式化，但是，我们仍然列出了它们，因为它们是必需的。为了防止后续的讨论过于形式化，我们会更倾向于使用第二个例子的风格。无论如何，如果你在完成或构造证明过程方面有困难的话，可以回头看形式更加清楚的例1.5。

归纳推理可能并不容易掌握，但注意到程序设计中的递归和归纳的紧密联系会对我们有所帮助。例如，函数$f(n)$（n是任意正整数）的递归定义包含两部分。一部分是根据$f(n), f(n-1), \cdots, f(1)$来定义$f(n+1)$，这个相当于归纳步骤。第二部分是通过非递归的定义$f(1), f(2), \cdots, f(k)$来从递归中“跳出”。这个相当于归纳基础。正如归纳证明一样，递归给定某些初值，依赖问题的递归本质，从问题的所有实例中得出结论。

反证法是另外一种强有力的技巧，在其他证明方法都不奏效的情况下，它可以胜任。假设我们想要证明某个命题P是正确的。我们就假设，在某一时刻P是错误的，然后看这个假设会得出什么样的结论。如果我们获得一个已知错误的结论，那么我们就会怀疑最初的假设，从而证明P一定是正确的。下面是一个经典的例子。

例1.7 一个有理数可以表示成两个正整数n和m的比，并且这两个正整数没有大于1的公共因子。如果某个数是实数，但不是有理数，我们称它为无理数。证明$\sqrt{2}$就是无理数。

按照反证法，假设我们要证明的命题的反命题。这里我们假设$\sqrt{2}$是有理数，那么它就应该可以写成

11

$$\sqrt{2} = \frac{n}{m} \tag{1-5}$$

这里n和m都是整数，并且没有大于1的公共因子。重新整理式（1-5），我们得到

$$2m^2 = n^2$$

因此n^2一定是偶数，n一定是偶数。既然n是偶数，那么我们就可以写成$n = 2k$，则可以得到

$$2m^2 = 4k^2$$

即

$$m^2 = 2k^2$$

因此m是偶数，但这就和我们的假设——n和m没有大于1的公共因子——相矛盾。所以，式（1-5）中的m和n不存在，即$\sqrt{2}$不是有理数。 □

这个例子展示了反证法的基本证明过程。通过作某种假设，我们得到一个与假设或已知事实相矛盾的结论。如果我们论证的所有步骤都是符合逻辑的，那么就会得出我们的初始假设是错误的。

习题

1. 对S的大小作归纳，证明：如果S是有限集合，则$|2^S| = 2^{|S|}$。
2. 证明：如果S_1和S_2是有限集合，并且$|S_1| = n$，$|S_2| = m$，那么$|S_1 \cup S_2| \leqslant n + m$。
3. 如果S_1和S_2是有限集合，证明$|S_1 \times S_2| = |S_1||S_2|$。
4. $S_1 \equiv S_2$当且仅当$|S_1| = |S_2|$，证明这是个等价关系。
5. 证明德摩根定律，即式（1-2）和式（1-3）。●
6. 有时交运算和并运算的符号与求和符号Σ类似，我们用类似的表示法来定义几个集合的交运算

$$\bigcup_{p \in \{i, j, k, \cdots\}} S_p = S_i \cup S_j \cup S_k \cdots$$

使用这种表示，通用的德摩根定律可以写成

$$\overline{\bigcup_{p \in P} S_p} = \bigcap_{p \in P} \overline{S_p}$$

12

和

$$\overline{\bigcap_{p \in P} S_p} = \bigcup_{p \in P} \overline{S_p}$$

证明当P是有限集合时这些恒等式成立。●

7. 证明 $S_1 \cup S_2 = \overline{\overline{S_1} \cap \overline{S_2}}$。
8. 证明$S_1 = S_2$当且仅当 $(S_1 \cap \overline{S_2}) \cup (\overline{S_1} \cap S_2) = \varnothing$。
9. 证明 $S_1 \cup S_2 - (S_1 \cap \overline{S_2}) = S_2$。●
10. 证明 $S_1 \times (S_2 \cup S_3) = (S_1 \times S_2) \cup (S_1 \times S_3)$。
11. 证明：如果 $S_1 \subseteq S_2$，那么有 $\overline{S_2} \subseteq \overline{S_1}$。
12. 给出满足下面等式的集合S_1和S_2，

$$S_1 = (S_1 \cup S_2) - S_2 \bullet$$

13. 证明：如果$f(n) = O(g(n))$，并且$g(n) = O(f(n))$，那么$f(n) = \Theta(g(n))$。
14. 证明：$2^n = O(3^n)$，但$2^n \neq \Theta(3^n)$。
15. 证明下面的数量级结果成立：
    (a) $n^2 + 5 \log n = O(n^2)$
    (b) $3^n = O(n!)$
    (c) $n! = O(n^n)$●
16. 证明：如果$f(n) = O(g(n))$，并且$g(n) = O(h(n))$，那么$f(n) = O(h(n))$。
17. 证明：如果$f(n) = O(n^2)$，并且$g(n) = O(n^3)$，那么

$$f(n) + g(n) = O(n^3)$$

$$f(n)g(n) = O(n^6)$$

13 18. 在习题17中，$g(n)/f(n) = O(n)$ 是否成立？

19. 假设$f(n) = 2n^2 + n$, $g(n) = O(n^2)$，下面的证明哪里有问题？

由于

$$f(n) = O(n^2) + O(n)$$

因此

$$f(n) - g(n) = O(n^2) + O(n) - O(n^2)$$

所以

$$f(n) - g(n) = O(n)$$

20. 以$\{v_1, v_2, v_3\}$为顶点，以$\{(v_1, v_1), (v_1, v_2), (v_2, v_3), (v_2, v_1), (v_3, v_1)\}$为边绘制一幅图。列举所有以$v_1$为始点的回路。

21. 已知图$G = (V, E)$，证明：

如果在$v_i \in V$和$v_j \in V$之间存在通道，那么在这两个顶点之间一定存在长度不大于$|V| - 1$的路径。

22. 任何两个顶点之间至多有一条边的图，证明在这个图中有n个顶点的情况下，它至多有n^2条边。

23. 证明：

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

24. 证明：

$$\sum_{i=1}^{n} \frac{1}{i^2} < 2 - \frac{1}{n}$$

25. 证明：对于所有$n \geqslant 4$，都有不等式$2^n < n!$成立。

26. 证明$\sqrt{8}$不是有理数。

27. 证明$2 - \sqrt{2}$是无理数。

28. 证明下面命题的正确性或不正确性。

(a) 有理数和无理数的和一定是无理数。

(b) 两个正无理数的和一定是无理数。

(c) 有理数和无理数的乘积一定是无理数。

29. 证明每个正整数都可以表示成质数的乘积。

★30. 证明所有质数的集合是无限的。

31. 一个质数对由两个相差为2的质数构成。存在很多质数对，比如，11和13，17和19，等等。质数三元组指的是三个数n, $n + 2$, $n + 4$都是质数。证明仅有的质数三元组是（1, 3, 5）和
14 （3, 5, 7）。

1.2 三个基本概念

这三个概念是这本书的主题，它们分别是：语言（language），文法（grammar）和自动

机（automata）。在学习这门课的过程中，我们将探讨关于这些概念的许多结论以及它们之间的关系。首先，我们必须理解这些术语的含义。

1.2.1 语言

我们都熟悉自然语言（比如：英语和法语）的表示。然而，我们大多数人可能发现准确解释“语言”是什么仍然很困难。字典中非形式化地定义语言为一个用来表达某些想法、事实或概念的系统，这个系统包括符号集和制定的规则。尽管这个定义给我们提供了语言的直观描绘，但是对于形式化语言的研究而言这个定义还是不够的。我们需要一个更准确的定义。

首先我们给出一个有限的、非空的符号集合Σ，称为*字母表*（alphabet）。使用单个符号可以构造出*符号串*（string），这个符号串是字母表中符号的有穷序列。例如，如果字母表是$\Sigma = \{a, b\}$，那么$abab$和$aaabbba$都是Σ上的符号串。除了特殊情况，我们一般使用小写字母$a, b, c, \cdots$来表示Σ中的元素，使用$u, v, w, \cdots$来表示符号串的名字。例如，我们使用

$$w = abaaa$$

来表示名为w的符号串，它的值为$abaaa$。

符号串w和v的*连接*（concatenation）是把v添到w的右端。即：如果

$$w = a_1a_2\cdots a_n$$
$$v = b_1b_2\cdots b_m$$

那么w和v的连接，表示成wv，就是

$$wv = a_1a_2\cdots a_nb_1b_2\cdots b_m$$

符号串的*逆*（reverse）是把符号串中的符号按照相反的顺序列出。如果w是上面提到的那个符号串，那么它的逆w^R就是

$$w^R = a_n\cdots a_2a_1$$

符号串w的*长度*（length），记成$|w|$，指的是符号串中包含的符号个数。我们经常使用到*空串*（empty string）。空串中没有任何符号，用λ表示。对于所有的符号串w，都有如下关系成立 15

$$|\lambda| = 0$$
$$\lambda w = w\lambda = w$$

符号串w中的任何连贯的符号构成的符号串称作w的*子串*（substring）。如果$w = vu$，那么子串v和子串u分别称为w的*前缀*（prefix）和*后缀*（suffix）。例如，如果$w = abbab$，那么$\{\lambda, a, ab, abb, abba, abbab\}$是$w$所有前缀的集合，而$bab, ab, b$都是它的后缀。

符号串的简单属性，比如它们的长度，是非常直观的，因此可能不需要详细介绍。例如，如果u和v都是符号串，那么它们的连接长度就是它们各自的长度的和，即：

$$|uv| = |u| + |v| \tag{1-6}$$

尽管这个关系是显然的，但是如果能够使它更精确并证明就更有帮助了。这样做的方法对于处理更加复杂的情况是重要的。

例1.8 证明对于任何u和v，式（1-6）都成立。为了证明这一点，我们首先需要一个符号串长度的定义。我们使用递归的方式定义如下：对于所有$a \in \Sigma$，w是Σ上的任意符号串，都有

$$|a| = 1$$
$$|wa| = |w| + 1$$

这个定义是我们根据直观理解符号串长度而得出的一个形式化的命题。单个符号的长度是1。符号串增加一个符号，符号串的长度也相应地增加1。使用这个形式化的定义，我们就可以使用归纳推理的方法证明式（1-6）。

根据定义，对于所有任意长度的符号串u和所有长度为1的符号串v，式（1-6）都成立，这样我们就有了归纳基础。作为归纳假设，我们认为对于所有任意长度的符号串u和所有长度为$1, 2, \cdots, n$的符号串v，式（1-6）都成立。那么，我们考虑长度为$n+1$的符号串v，并且把它写成$v = wa$。则有

$$|v| = |w| + 1$$
16
$$|uv| = |uwa| = |uw| + 1$$

但是根据归纳假设（因为w的长度为n，所以可以使用这个归纳假设的结论）

$$|uw| = |u| + |w|$$

因此，

$$|uv| = |u| + |w| + 1 = |u| + |v|$$

所以，根据上面的推导步骤和证明，我们证明了对于所有符号串u，和所有长度达到$n+1$的符号串v，式（1-6）都成立。 □

如果w是符号串，那么w^n表示重复w这个符号串n次。我们定义
对于所有的w都有

$$w^0 = \lambda$$

如果Σ是字母表，那么Σ^*表示连接Σ中的零个或多个符号获得的所有符号串的集合。集合Σ^*总是包含λ。为了把空串排除在外，我们定义

$$\Sigma^+ = \Sigma^* - \{\lambda\}$$

尽管根据假设知道Σ是有限的，但是因为Σ^*和Σ^+包含的符号串的长度是没有限制的，所以他们两个集合总是无限的。

一种语言通常定义为Σ^*上的子集。一种语言L中的一个符号串称作这个语言L的一个*句子*（sentence）。这个定义非常宽泛，字母表Σ上的符号串的任意集合都可以看成是一种语言。后面我们会学到某些特殊语言定义和描绘的方法，这样就能给出这些语言的结构，而不仅仅是宽泛的定义。不过，我们还是先看几个特殊的例子。

例1.9 设$\Sigma = \{a, b\}$，那么

$$\Sigma^* = \{\lambda, a, b, aa, ab, ba, bb, aaa, aab, \cdots\}$$

集合$\{a, aa, aab\}$是Σ上的一种语言。因为它有有限个句子，所以我们称之为有限语言。集合

$$L = \{a^n b^n : n \geqslant 0\}$$

也是Σ上的一种语言。符号串$aabb$和$aaaabbbb$属于语言L，而符号串abb不属于语言L。这种语言是无限的。我们已知的大多数语言都是无限的。 □ 17

因为语言也是集合，所以也可以类似地定义两种语言的并、交和差。在全集Σ^*上定义语言L的补集为

$$\bar{L} = \Sigma^* - L$$

一种语言的逆，指的是所有的符号串的逆的集合，即

$$L^R = \{w^R : w \in L\}$$

两种语言L_1和L_2的连接指的是L_1中的任意元素和L_2中的任意元素通过连接形成的所有符号串的集合。具体表示为

$$L_1L_2 = \{xy : x \in L_1, y \in L_2\}$$

我们定义L^n为L自身连接n次，特殊地，对于任意语言L，都有

$$L^0 = \{\lambda\}$$

和

$$L^1 = L$$

最后，我们定义一种语言的星闭包（star-closure）为

$$L^* = L^0 \cup L^1 \cup L^2 \cdots$$

和正闭包（positive closure）为

$$L^+ = L^1 \cup L^2 \cdots$$

例1.10 如果

$$L = \{a^n b^n : n \geqslant 0\}$$

那么有

$$L^2 = \{a^n b^n a^m b^m : n \geqslant 0, m \geqslant 0\}$$

注意上面的n和m没有关系，符号串$aabbaaabbb$属于L^2。

L的逆使用集合的方式描述很容易，

$$L^R = \{b^n a^n : n \geqslant 0\}$$

但是以这种方式描述$\bar{L}$或L^*比较困难。尝试几次，你就会发现集合表示对于一些复杂的语言比较受限。 □ 18

1.2.2 文法

为了精确地研究语言，我们需要一种机制来描述它们。平常的语言是不准确的、含糊的，因此，英语的非形式化描述经常是不够的。在例1.9和例1.10中使用的集合表示更合适，但是这种表示还是有局限性的。接下来我们将学习几种应用在不同的环境中的语言定义机制。首

先，我们介绍一种最常见、最有用的表示，文法（grammar）。

英语的文法告诉我们一个特殊的句子是否是一种好的句式。英语文法的一个典型规则是“句子由名词短语和谓语构成”。我们把它写成一种更简明的形式，是

<句子>→<名词短语><谓词>

上式带有清楚的解释。这种形式对于处理实际的句子当然是不够的。我们必须对新引进的构成成分<名词短语>和<谓词>给出如下定义

<名词短语>→<冠词><名词>
<谓词>→<动词>

我们把上述结构与实际的单词相联系，就是“a”和“the”是<冠词>，“boy”和“dog”是<名词>，“runs”和“walks”是<动词>。这样根据文法我们就得到正确形式的句子“a boy runs”和“the dog walks”。如果我们要在理论上给出一个完整的文法，那么每个正确的句子就都能够按照这种方式解释了。

这个例子使用简单的句子解释了一个通用的概念定义。我们这里从顶层概念开始，逐步简化到语言的不可再化简的部分。这些概念的概括帮助我们导出文法的形式化概念。

定义1.1 文法G是一个四元组

$$G = (V, T, S, P)$$

其中，V是对象的有限集合，称为变量（variable），
T是对象的有限集合，称为终结符（terminal symbol），
$S \in V$是一个特殊符号，称为开始符（start variable），
19 P是产生式（production）的有限集合。
如果不特殊指出，集合V和T是不相交且非空的集合。

产生式规则是文法的核心。它们指出文法如何把一个符号串转化成另一个符号串。通过这个过程，产生式规则定义了一个和这个文法相关的语言。在我们的讨论中，我们假设所有的产生式规则都是按照下面的形式表示的

$$x \to y$$

这里，x是$(V \cup T)^+$中的元素，y属于$(V \cup T)^*$。这些产生式可以按照下面的各种方式来应用：如果一个符号串w可以写成

$$w = uxv$$

我们就说产生式$x \to y$可以应用到这个符号串中。我们可以用y替换x，由此获得一个新的符号串

$$z = uyv$$

上述过程可以写成

$$w \Rightarrow z$$

这时我们称w推导（derive）出z，或者说z由w推导出。应用文法的产生式我们可以以任意顺序推导出连续的符号串。只要可以，我们就可以使用产生式，按照我们的意愿应用产生式。如果

$$w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n$$

我们就说w_1推导出w_n，并写成

$$w_1 \overset{*}{\Rightarrow} w_n$$

*表示从w_1推导出w_n经过没有指定的数量的步骤（包括零步）。因此

$$w \overset{*}{\Rightarrow} w$$

总是正确的。

以不同的顺序应用产生式规则，一个给定的文法可以生成很多符号串，其中所有仅由终结符构成的符号串的集合就是这个文法定义或生成的语言。 20

定义1.2 设$G = (V, T, S, P)$是一个文法，那么集合

$$L(G) = \{w \in T^* : S \overset{*}{\Rightarrow} w\}$$

就是文法G生成的语言。

如果$w \in L(G)$，那么序列

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n \Rightarrow w$$

就是句子w的一个推导（derivation）。包含变量和终结符的符号串$S, w_1, w_2, \cdots, w_n$，称作推导的句型（sentential form）。

例1.11 已知文法

$$G = (\{S\}, \{a, b\}, S, P)$$

其中，P定义为

$$S \to aSb$$
$$S \to \lambda$$

因此有

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

所以，我们可以得到

$$S \overset{*}{\Rightarrow} aabb$$

符号串$aabb$是G生成的语言的一个句子，而$aaSbb$是一个句型。

文法G可以完全定义语言$L(G)$，但是从文法上却不容易获得对语言的清楚描述。然而，这里的这个答案是相当明显的。不难推测

$$L(G) = \{a^n b^n : n \geqslant 0\}$$

证明这个结论很容易。如果我们注意到规则$S \to aSb$是递归的，那么就可以使用归纳证明的方法来证明它。我们首先指出所有的句型都有下面的形式

$$w_i = a^i S b^i \tag{1-7}$$ 21

假设对于所有长度小于或等于$2i + 1$的句型w_i，式（1-7）都成立。那么，为了获得另一个句型（不是句子），我们就只能应用产生式$S \to aSb$。这样我们就得到

$$a^iSb^i \Rightarrow a^{i+1}Sb^{i+1}$$

于是长度为$2i+3$的每个句型也都是式（1-7）的形式。因为对于$i=1$，式（1-7）显然是成立的，所以根据归纳对于所有的i它都成立。最后，为了获得一个句子，我们必须应用产生式$S \to \lambda$，此时

$$S \overset{*}{\Rightarrow} a^nSb^n \Rightarrow a^nb^n$$

代表了所有可能的推导。因此，文法G只能推导出a^nb^n这样形式的符号串。

我们还必须指出所有这种形式的符号串都能够被这个文法推导出。这个是很容易的。我们只要按需要的次数简单地应用产生式$S \to aSb$，最后应用$S \to \lambda$就可以了。 □

例1.12 寻找能够生成下面语言的文法

$$L = \{a^nb^{n+1} : n \geqslant 0\}$$

解决以前例子的想法可以扩展到这个例子。我们只需要生成一个特殊的b。这个可以通过产生式$S \to Ab$和其他几个产生式来完成，其他的产生式选择的是使用A推导出前一个例子生成语言时使用的产生式。按照这种方式推理，通过下面的产生式，我们可以获得文法$G = (\{S, A\}, \{a, b\}, S, P)$

$$S \to Ab$$
$$A \to aAb$$
$$A \to \lambda$$

我们可以使用这些产生式推导出一些特殊的句子从而更加确认结果的正确性。 □

上述例子都相当简单，因此太严格的证明反而显得多余。但是为一种以非形式化的方式描述的语言寻找相应的文法，或是为某个文法描述的语言指出直观的特点，通常都不是容易的。为了证明某个语言是由文法G生成的，我们必须要证明（a）每个$w \in L$都可以使用G中的产生式由S推导出，（b）每个这样推导出的符号串都是语言L中的句子。

22

例1.13 已知$\Sigma = \{a, b\}$，并且$n_a(w)$和$n_b(w)$分别表示符号串w中a和b的个数。文法G的产生式是

$$S \to SS$$
$$S \to \lambda$$
$$S \to aSb$$
$$S \to bSa$$

它生成的语言是

$$L = \{w : n_a(w) = n_b(w)\}$$

这个证明不是很明显，我们需要详细地写出证明的过程。

首先，因为生成a的产生式即$S \to aSb$和$S \to bSa$，同时生成b。所以，G中的每个句型都有相同数目的a和b。因此，$L(G)$中的每个元素都属于语言L。但是L中的每个符号串都可以由G推导出，这个命题的正确性就很难一眼看出。

看看上面的问题，我们考虑一下$w \in L$可能存在的各种形式。假设w是以a开头，b结束。那么它的形式就是

$$w = aw_1b$$

其中，w_1也属于L。如果S确实可以推导出L中的任意符号串，那么我们可以认为上面的形式可以由

$$S \Rightarrow aSb$$

开始推导出。如果以b开头，a结束，那么可以类似地证明。因为L中符号串的开头符号和结束符号可以相同，所以这种证明并不能覆盖所有的情况。如果我们写出一个这样的符号串，比方说$aabbba$，那么我们可以把它看成是两个更短符号串$aabb$和ba的连接，其中这两个符号串都属于L。这个对于一般情况是否也是正确的呢？为了证明它是正确的，我们可以像下面这样证明：假设从符号串的左端开始，当遇到一个a时我们加1，当遇到一个b时我们减1。如果符号串开始和结束都是a，那么在最左边的符号之后的符号串的计算得到的值应该是1，在最右边的符号之前的符号串的值应该是－1。因此，在符号串中间的某个位置上，计算的值会为0。这就暗示这个符号串一定可以写成这样的形式

$$w = w_1 w_2$$

其中，w_1，w_2都属于L。这种情况我们可以使用产生式$S \rightarrow SS$。

既然我们已经直观上知道如何证明了，下面我们准备进行严格的证明。我们再次使用归纳证明。假设对于所有满足$w \in L$，并且$|w| \leqslant 2n$都可以由G推导出。证明对于所有长度为$2n+2$的结论也成立。如果$w = aw_1b$，那么w_1属于L，并且$|w_1| = 2n$。因此，根据假设有 23

$$S \overset{*}{\Rightarrow} w_1$$

于是，可能有

$$S \Rightarrow aSb \overset{*}{\Rightarrow} aw_1b = w$$

所以w可以由G推导出。显然，如果$w = bw_1a$，那么可以类似地证明。

如果w不是这种形式，即：如果它开始和结束的符号相同，那么根据上面计算值的方法可以得出，它可以写成$w = w_1w_2$的形式。其中，w_1，w_2都属于L，并且长度都小于或等于$2n$。因此，我们可以得到

$$S \Rightarrow SS \overset{*}{\Rightarrow} w_1S \overset{*}{\Rightarrow} w_1w_2 = w$$

上面的归纳假设对于$n = 1$显然也成立，这样我们就有了归纳基础。最终证明了对于所有的n，命题都是正确的。 □

通常，一个语言可以由多个文法生成。尽管这些文法是有区别的，但是它们在某种程度上是等价的。如果两个文法G_1和G_2产生相同的语言，即

$$L(G_1) = L(G_2)$$

我们就说这两种文法是*等价的*（equivalent）。

我们后面会谈到，两种文法的等价性并不容易看出。

例1.14 已知文法$G_1 = (\{A, S\}, \{a, b\}, S, P_1)$，产生式$P_1$包括

$$S \rightarrow aAb \mid \lambda$$
$$A \rightarrow aAb \mid \lambda$$

这里我们引入了一种方便的速记表示。如果几个产生式有相同的左部，那么它们的右部可以

写在同一个产生式的右边，中间用|隔开。上面表示中，产生式$S \to aAb|\lambda$表示产生式$S \to aAb$和$S \to \lambda$。

上面这个文法等价于例1.11的文法G。只要证明

$$L(G_1) = \{a^n b^n : n \geqslant 0\}$$

24 成立，那么它们的等价性就容易证明了。我们把这个证明过程留作习题。 □

1.2.3 自动机

自动机是数字计算机的抽象模型。正因为如此，每个自动机都包含一些本质的特征。它们都有读入装置。一般都假设输入是给定字母表上的一个符号串，将符号串写在输入文件（input file）上。自动机只能读入输入文件，但不能够修改。输入文件可以分成若干个单元，每个单元存放一个符号。输入装置从左到右读入输入文件，一次读入一个符号。输入装置能够识别输入串的尾部（通过读出文件结束条件来识别）。自动机能够生成某种形式的输出。自动机有一个临时存储（storage）设备。这个临时存储设备包含无限个单元，每个单元存放着字母表上的一个符号（不一定是输入字母表上的符号）。自动机可以读入存储单元中的内容，并修改。另外，自动机还有控制部件（control unit）。这个控制部件可以处于有限个内部状态（internal state）中的任何一个上，并且按照某个指定的方式改变状态。图1-3给出了一个通用自动机的示意图。

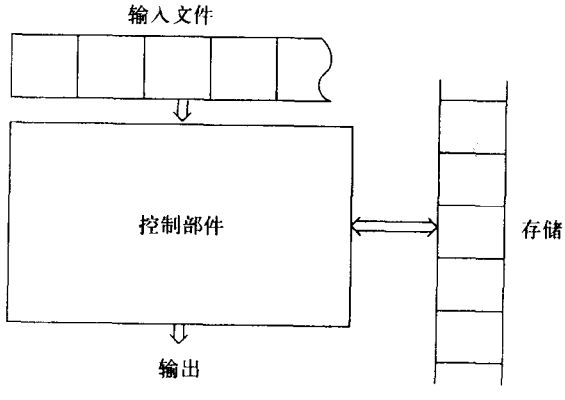


图 1-3

根据假定，自动机是在离散时间框架上操作的。在任意给定的时间点上，控制部件处于某个内部状态上，输入装置正在搜索输入文件上的一个特定符号。下一时刻控制部件会出现在哪一个内部状态上由下一个状态（nextstate）或转移函数（transition function）决定。转移函数根据当前状态、当前读入符号和临时存储空间中的当前信息，来决定下一个状态。从一个时间间隔向下一个时间间隔转移的过程中，自动机会产生输出，或改变临时存储空间中的信息。格局（configuration）这个术语指的是控制部件、输入文件和临时存储空间的某一特定状态。自动机从一个格局到另一个格局的转换称为一个迁移（move）。

这个通用模型覆盖了我们在这本书讨论到的所有自动机。有穷状态控制对于一些特殊类
25 型的自动机来说是通用的，但是它们在输出的方式和临时存储的本质上是不同的。临时存储的本质对于特殊类型的自动机有着强大的效果。

为了后续讨论的方便，我们必须区分确定型自动机（deterministic automata）和非确定型自动机（nondeterministic automata）。确定型自动机指的是根据当前的格局，自动机的每一步迁移都是唯一确定的。如果我们知道内部状态、输入和临时存储空间的内容，我们就可以准确地预测自动机的下一步行为。在非确定型自动机中，就不是这样的。在每一点上，非确定型自动机都有几个可能的迁移，因此，我们只能得到一个可能行为的预测集合。各种类型的确定型自动机和非确定型自动机间的关系在我们的研究中扮演着很重要的角色。

如果一个自动机的输出响应仅仅限定在简单的“yes”和“no”之间的话，这个自动机就是接受器（accepter），提供一个输入串，接受器要么接受它，要么拒绝它。一个更通用的自动机，能够输出和输入相同的符号，这种自动机叫转换器（transducer）。尽管我们在这本书的主要关注点是接受器，不过我们在下一节还是给出了一些简单的转换器例子。

习题

1. 对n进行归纳，证明：对于所有的符号串u和所有的n，都有$|u^n| = n|u|$。
2. 上面已经非形式化地介绍了一个符号串的逆，它可以使用递归的方法更准确地定义为：
   对于任意$a \in \Sigma$，$w \in \Sigma^*$，都有

$$a^R = a$$
$$(wa)^R = aw^R$$

   使用这个来证明：对于任意u，$v \in \Sigma^*$，都有

$$(uv)^R = v^R u^R$$ ●

3. 证明：对于所有的$w \in \Sigma^*$，都有$(w^R)^R = w$。
4. 设$L = \{ab, aa, baa\}$。下面哪些符号串属于语言L^*？

$$abaabaaabaa, aaaabaaaa, baaaaabaaaab, baaaaabaa$$ ●

5. 对于例1.12和例1.13中的语言，等式$L = L^*$是否成立？
6. 是否存在语言满足$(\overline{L})^* = (\overline{L^*})$？
7. 证明：对于任意语言L_1和L_2，都有

$$(L_1 L_2)^R = L_2^R L_1^R$$

8. 证明对于任意语言都有$(L^*)^* = L^*$。
9. 证明下列命题成立或不成立。
   (a) 对于任意语言L_1和L_2，有$(L_1 \cup L_2)^R = L_1^R \cup L_2^R$。
   (b) 对于任意语言L，有$(L^R)^* = (L^*)^R$。
10. 给出在$\Sigma = \{a, b\}$上能够分别满足下列条件的语言的文法：
    (a) 只有一个a的所有符号串
    (b) 至少有一个a的所有符号串
    (c) a的个数不多于3的所有符号串
    (d) 至少有3个a的所有符号串 ●

    针对每种情况，证明你给出的文法是能够产生相应语言的文法。

26

11. 简单描述由下面的产生式对应的文法生成的语言。●

$$S \to aA$$
$$A \to bS$$
$$S \to \lambda$$

12. 下面产生式对应的文法可以生成什么样的语言?

$$S \to Aa$$
$$A \to B$$
$$B \to Aa$$

13. 分别为下面的各种语言，构造生成这种语言的文法:

(a) $L_1 = \{a^n b^m : n \geqslant 0, m > n\}$ ●

(b) $L_2 = \{a^n b^{2n} : n \geqslant 0\}$

(c) $L_3 = \{a^{n+2} b^n : n \geqslant 1\}$

(d) $L_4 = \{a^n b^{n-3} : n \geqslant 3\}$ ●

(e) $L_1 L_2$

(f) $L_1 \cup L_2$

(g) L_1^3

(h) L_1^*

(i) $L_1 - \overline{L_4}$

★14. 给出在$\Sigma = \{a\}$上构造产生下面语言的文法:

(a) $L = \{w : |w| \bmod 3 = 0\}$

(b) $L = \{w : |w| \bmod 3 > 0\}$ ●

(c) $L = \{w : |w| \bmod 3 \neq |w| \bmod 2\}$

27 (d) $L = \{w : |w| \bmod 3 \geqslant |w| \bmod 2\}$

15. 构造可以生成下面语言的文法

$$L = \{ww^R : w \in \{a, b\}^+\}$$

并且给出一个完整的证明。

16. 使用例1.13的表示，构造下面语言的文法。假设$\Sigma = \{a, b\}$。

(a) $L = \{w : n_a(w) = n_b(w) + 1\}$ ●

(b) $L = \{w : n_a(w) > n_b(w)\}$

★(c) $L = \{w : n_a(w) = 2n_b(w)\}$

(d) $L = \{w \in \{a, b\}^* : |n_a(w) - n_b(w)| = 1\}$

17. 假设$\Sigma = \{a, b, c\}$，重复习题16（a）和习题16（d）。

18. 完成例1.14的证明，证明$L(G_1)$确实能生成给定的语言。

19. 下面两个产生式对应的文法等价吗？假设两个文法都是以S为开始符的。

$$S \to aSb | ab | \lambda$$

和

$$S \to aAb|ab$$
$$A \to aAb|\lambda$$

20. 证明文法$G = (\{S\}, \{a, b\}, S, P)$ 和例1.13中的文法等价，其中，产生式是

$$S \to SS|SSS|aSb|bSa|\lambda$$

★21. 到目前为止，我们给出的例子都是相对简单的文法，每个产生式的左部只有一个变量。正如我们看到的，这种文法非常重要，但是定义1.1允许更一般的形式。

已知文法$G = (\{A, B, C, D, E, S\}, \{a\}, S, P)$，产生式是

$$S \to ABaC$$
$$Ba \to aaB$$
$$BC \to DC|E$$
$$aD \to Da$$
$$AD \to AB$$
$$aE \to Ea$$
$$AE \to \lambda$$

根据这个文法推导出$L(G)$ 中3个不同的句子，然后根据这些句子，对$L(G)$ 进行猜测。 28

1.3 一些应用*

尽管我们强调形式语言和自动机的抽象与数学本质，但是，这些概念在计算机科学方面仍然存在着广泛的应用。事实上，它们还是和很多特殊领域相联系的普遍主题。在这一书中，我们给出一些简单的例子，从而让读者清楚我们这里研究的内容不是许多抽象内容的罗列，而是一些对于我们理解许多重要的现实问题很有帮助的东西。

形式语言和文法广泛应用于和程序语言相关的地方。在编程的大部分过程中，我们或多或少都需要直观地理解我们写的语言。偶尔，当使用不熟悉的特征时，我们需要去参考语言的准确描述，比如大多数程序文本中的文法图。如果我们写一个编译器，或者想要推敲一个程序的正确性，几乎每一步都需要知道语言的准确描述。程序语言可以用多种方式来精确定义，可是，文法却很可能是其中应用最广的一种。

描述像Pascal或C这样典型语言的文法很庞大。我们以一个较小的语言为例来介绍文法的应用，其中较小的语言是这个大语言的一个组成部分。

例1.15 Pascal中所有合法标识符构成的集合是一种语言。通俗地说，合法标识符指的是所有以字母开头，后面可以跟任意个数的字母或数字的符号串。下面的文法使得这个非形式化的定义更加准确。

$$\langle id\rangle \to \langle letter\rangle\ \langle rest\rangle$$
$$\langle rest\rangle \to \langle letter\rangle\ \langle rest\rangle|\langle digit\rangle\ \langle rest\rangle|\lambda$$
$$\langle letter\rangle \to a|b|\cdots|z$$
$$\langle digit\rangle \to 0|1|\cdots|9$$

在这个文法中，变量是<*id*>、<*letter*>、<*digit*>和<*rest*>，$a, b, \cdots, z, 0, 1, \cdots, 9$是终结符。标识符$a0$的推导过程是

$$
\begin{aligned}
<id> &\Rightarrow <letter>\ <rest> \\
&\Rightarrow a<rest> \\
&\Rightarrow a<digit><rest> \\
&\Rightarrow a0<rest> \\
&\Rightarrow a0
\end{aligned}
$$

29

程序设计语言通过文法的形式给出的定义是普通的，且更有用。但是仍然有其他一些更方便的定义。例如，我们使用接受器来描述语言——把接受到的每一个符号串都当成是语言的组成部分。如果以更精确的方式来讨论这个问题，我们需要给自动机一个更形式化的定义。我们可以很快就完成这件事。但暂时我们还是先给出一个更直观的定义。

自动机可以表示成一个图，结点表示内部状态，边表示转移。边上的标记表示转移中发生了什么（在输入和输出方面）。例如，图1-4表示当输入符号是a时，发生从状态1到状态2的转移。头脑中有了这样一个直观的图，我们看看下面对Pascal标识符的另一种描述方式。 □

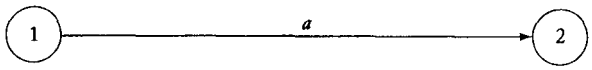
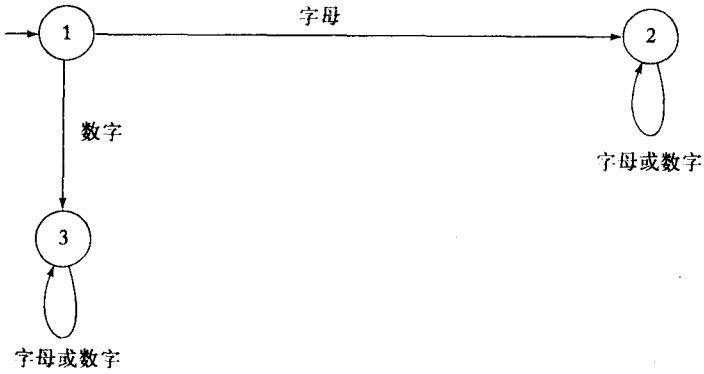


图 1-4

例1.16 图1-5是一个接受Pascal所有合法标识符的自动机。这里做些必要的解释。我们假设初始时自动机位于状态1，我们用一个指向这个状态的箭头（没有起始结点）表示。与以往相同，按照从左到右的顺序读取待检查的符号串，一次检查一个。当第一个符号是字母时，自动机跳到状态2，后面的符号串就无关紧要了。状态2因此表示接受器的“yes”状态。相反，如果第一个符号是数字，自动机就会跳转到状态3，表示接受器的“no”状态，并且永远停留在这个状态。在我们的解决方案中，我们假设输入只有字母和数字。 □



30

图 1-5

编译器和其他的翻译器，广泛利用我们已经接触的这几个例子的想法，把一种语言的程序转化成另一种语言的程序。正如例1.15所示，程序设计语言可以通过文法被精确地定义。在判断某段代码是否能够被一种程序设计语言接受时，文法和自动机都起到了举足轻重的作用。上面的例子给出了如何处理这一过程的提示。后续的例子也将以此为基础扩展。

另一个重要的应用领域是数字设计，在这个领域，转换器的概念很流行。尽管对于这个领域我们在这里不会深入探讨，但是我们还是给出一个简单例子。原则上，任何数字计算机

都可以被看成是自动机，但是这种观点不一定合适。假设我们把计算机的内部寄存器和主存看成是自动机的控制部件，那么，如果寄存器和内存共有n位，则自动机总共有2^n个内部状态。即使对于一个很小的n，结果仍然太大而无法处理。但是如果我们关注的是一个小得多的单元，那么自动机理论就会成为一个有用的设计工具。

例1.17 二进制加法器是通用计算机的一个重要部分。这种加法器把两个表示数字的位符号串作为输入，把它们的和作为输出。简单地说，假设我们仅仅处理正整数的加法，我们使用

$$x = a_0a_1\cdots a_n$$

表示正整数

$$v(x) = \sum_{i=0}^{n} a_i 2^i$$

这是一个通常的二进制表示的逆。

一个串行加法器将两个这样的数$x = a_0a_1\cdots a_n$和$y = b_0b_1\cdots b_n$从左开始，逐位相加。每个位加法产生和的一个数字和一个向高位的进位。二进制加法表（图1-6）总结了这个过程。

a_i \ b_i	0	1
0	0 无进位	1 无进位
1	1 无进位	0 进位

图 1-6

图1-7是我们学习计算机时首先看到的一种块图表。从图中可以知道，加法器是一个接受两个位，产生和数位与可能的进位的盒子。它解释了加法器能完成什么样的计算，但是却没有解释它的内部工作过程。自动机（这里是一个转换器）可以使这个过程更加准确。

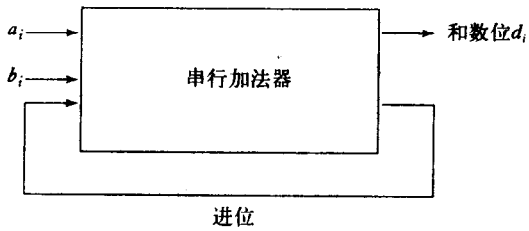


图 1-7

转换器的输入是位偶（a_i, b_i），输出是和数位d_i。我们再次用边上标有（a_i, b_i)/d_i的图表示这个自动机。从一步运算到下一步运算的进位在自动机中是用两个标有“carry”和“no carry”的内部状态来表示的。最初，转换器位于“no carry”状态。除非遇到位偶（1, 1），否则它将

保持不变。如果遇到位偶（1，1），自动机就进入“carry”状态。当读入下一个位偶的时候，计算时要考虑进位。图1-8给出了串行加法器的完整描述图。按照这个图，通过几个小例子，你就可以确定这个加法器是正确的。

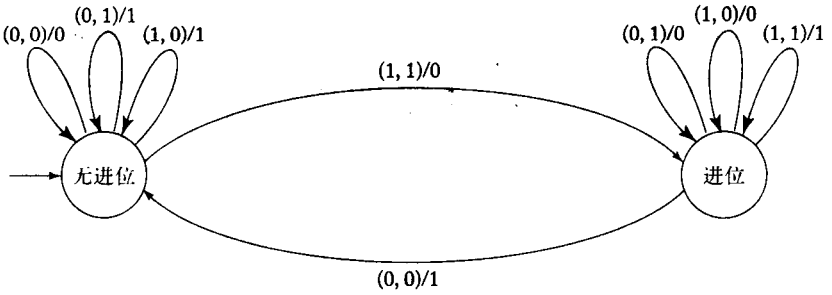


图 1-8

31～32

正如这个例子指出的，在电路的高层功能描述和它的晶体管、门电路和开关构成的逻辑实现中，自动机起到了桥梁的作用。自动机清晰地表示了决策逻辑，它的形式化有利于它进行精确的数学实现。由于这个原因，数字设计主要依赖于自动机理论的概念。对这方面感兴趣的读者可以看这方面的经典文章，如Kovahi 1978。 □

习题

1. 给出C语言中整数集合的文法。●
2. 设计一个接受C语言中整数的接受器。
3. 设计一个能生成C语言中所有实数的文法。
4. 假设某种程序设计语言只允许标识符以字母开始，可以包含的数字至少一个、至多三个，字母的数目任意。给出一个接受这种标识符集合的文法和接受器。
5. 给出Pascal语言中var声明的文法。
6. 在罗马数字系统中，数字是定义在$\{M, D, C, L, X, V, I\}$这个字母表上的符号串。设计一个只接受形式正确的罗马数字的接受器。出于简单考虑，我们不使用“化减”规定中的IX表示9，而是使用与之等价的VIIII来表示。
7. 我们假设自动机工作在离散时间域中，这点对于我们后续的讨论几乎没有影响。然而，在数字设计中，时间因素的假设就很重要了。

   为了能够使计算机的不同部分的信号同步到达，延迟电路是必需的。单位延迟转换器指的是在一个单位时间后，简单地复制输入（可以看成是符号的连续流）。特殊地，如果转换器在时间t读入符号a，它会在时间$t+1$复制并输出这个符号。当时间为$t=0$时，转换器什么也没输出。因此，转换器把输入为$a_1a_2\cdots$，变成输出$\lambda a_1a_2\cdots$。

   设计一个$\Sigma=\{a, b\}$上的单位延迟转换器的工作图。●
8. 一个n单位延迟转换器是在n个时间单位后，将输入复制n次。即：把输入$a_1a_2\cdots$转化成$\lambda^n a_1a_2\cdots$。这就意味着在前n个时间槽，转换器没有输出。

   (a) 构造一个$\Sigma=\{a, b\}$上的二单位延迟转换器。

   (b) 证明一个n单位延迟转换器至少有$|\Sigma|^n$个状态。

9. 一个二进制串的二进制补码表示的是一个正整数。它首先对符号串的每一位取补，然后在最低位加1。设计一个能够把位符号串转化成其相应的二进制补码的转换器。假设二进制数按照例1.17中的方式表示，最低位位于符号串的左侧。 33

10. 设计一个能把二进制符号串转化成八进制数的转换器。例如，二进制串001101110应该产生输出156。●

11. 已知输入$a_1a_2\cdots$是一个位符号串。每三个位为一个子串，设计一个转换器计算每一个子串的奇偶性。明确的定义为

$$\pi_1 = \pi_2 = 0$$
$$\pi_i = (a_{i-2} + a_{i-1} + a_i)\bmod 2,\ i = 3, 4, \cdots$$

例如，输入110111应该产生输出000001。●

12. 设计一个转换器，接受位符号串$a_1a_2a_3\cdots$，计算该符号串中所有连续的3个位形成的数值模5。假设输出为$m_1, m_2, m_3, \cdots$，则

$$m_1 = m_2 = 0$$
$$m_i = (4a_i + 2a_{i-1} + a_{i-2})\bmod 5,\ i = 3, 4, \cdots$$

13. 数字计算机一般使用某种类型的编码，来表示位符号串的所有信息。例如，符号信息可以使用有名的ASCII码系统来编码。

例如，分别已知字母表$\{a, b, c, d\}$和$\{0, 1\}$，从第一个字母表向第二个字母表译码，规则是：$a\to 00$, $b\to 01$, $c\to 10$, $d\to 11$。构造一个建立在$\{0, 1\}$上的译码转换器。例如：输入010011应该产生输出bad。

14. 设x和y是两个正二进制数。设计一个输出为$\max(x, y)$的转换器。 34

第2章 有穷自动机

我们在第1章简要地、非正式地介绍了计算理论的基本概念，还特意对自动机进行了讨论。至此，我们仅仅概要地了解了什么是自动机，以及如何用图来表示自动机。现在，我们需要进一步提供它的更精确的形式化的定义，以便得出一些严格的结论。首先介绍有穷接受器。它是我们将在最后一章介绍的通用模型的一个简单又特殊的例子。这种类型的自动机的特点是没有临时存储。因为输入文件不可改写，所以有穷自动机在计算过程中所能“记住”的东西是相当有限的。通过把控制部件放在某一个特殊状态上，自动机可以把有限的信息保存到这个控制部件中。但是由于这种特殊状态是有限的，有穷自动机只能处理这种情况——每一时刻可以存储的信息都是有严格限制的。例1.16中的自动机是一个有穷接受器的例子。

2.1 确定型有穷接受器

我们首先详细研究的自动机是操作确定的有穷接受器。我们先给出确定型接受器的准确的形式化定义。

2.1.1 确定型接受器和转换图

定义2.1 确定型有穷接受器（deterministic finite accepter，dfa）是一个五元组

$$M = (Q, \Sigma, \delta, q_0, F)$$

其中，Q是内部状态（internal state）的有限集合，

Σ是符号的有限集合，叫做输入字母表（input alphabet），

$\delta: Q \times \Sigma \rightarrow Q$是一个全函数，叫转移函数（transition function），

$q_0 \in Q$是初态（initial state），

$F \subseteq Q$是终态（final state）集合。

确定型有穷接受器按照下面的方式工作。开始时假设接受器处于初态q_0，输入装置位于输入符号串最左端的符号上。自动机每次迁移的时候，输入装置向右迁移一个位置，因此，每次迁移读入一个输入符号。当输入装置读完符号串时，如果自动机处于它的一个终态，那么表示自动机接受了这个符号串。否则，表示自动机拒绝接受这个符号串。输入装置只能从左向右迁移，并且每次只能读入一个符号。转移函数δ决定自动机从一个状态转移到另一个状态。例如，如果

$$\delta(q_0, a) = q_1$$

dfa处于状态q_0，并且当前的输入符号是a，那么dfa会进入状态q_1。

在讨论自动机时，我们应该使用一个清晰直观的图来表示它。为了可视化地表示有穷自动机，我们使用转换图（transition graph）。在转换图中，顶点表示状态，边表示转移。顶点

上标有状态的名字，边上标有输入符号的当前值。例如，如果q_0和q_1都是某个确定型有穷接受器M的内部状态，那么M对应的图上就应该有标有q_0和q_1的两个顶点。标有a的边（q_0, q_1）表示转移函数$\delta(q_0, a) = q_1$。有一个没有起始于任何顶点的箭头指向初态，这是初态的特征。终态用双圈表示。

36

进一步地，如果$M = (Q, \Sigma, \delta, q_0, F)$是一个确定型有穷接受器，那么它对应的转换图$G_M$就有$|Q|$个顶点，每个顶点上标着不同的$q_i \in Q$。对应于每个转移规则$\delta(q_i, a) = q_j$，在图中都存在着一条标有$a$的边（$q_i$, q_j）。标有q_0的顶点是初态顶点，而标有$q_f \in F$的顶点是终态顶点。把一个dfa的（$Q, \Sigma, \delta, q_0, F$）定义转化成转换图表示很简单，反之亦然。

例2.1 图2-1表示dfa

$$M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$$

其中，δ定义为

$$\delta(q_0, 0) = q_0, \delta(q_0, 1) = q_1$$
$$\delta(q_1, 0) = q_0, \delta(q_1, 1) = q_2$$
$$\delta(q_2, 0) = q_2, \delta(q_2, 1) = q_1$$

下面看一下这个dfa接受符号串01的过程。自动机开始处于状态q_0，首先读入符号0。看图中的边，我们知道自动机仍然处于状态q_0。然后，读入1，自动机进入状态q_1。这时，我们处于符号串结束的位置，同时，自动机进入终态q_1。至此，符号串01被接受。这个dfa不接受符号串00，因为自动机连续读入两个0之后，会进入状态q_0。用类似的推理，我们会看到自动机接受符号串101、0111和11001，而不接受100和1100。 □

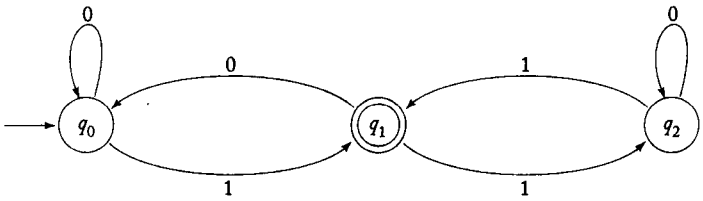


图 2-1

如果引入扩展的转移函数$\delta^*: Q \times \Sigma^* \to Q$，就更方便了。$\delta^*$的第二个参数是一个符号串，而不是一个单一符号。这个函数的值是自动机读入这个符号串后进入的状态。例如，如果

37

$$\delta(q_0, a) = q_1$$

并且

$$\delta(q_1, b) = q_2$$

那么

$$\delta^*(q_0, ab) = q_2$$

我们可以递归地给出形式化的定义

对于所有的$q \in Q, w \in \Sigma^*, a \in \Sigma$，都有

$$\delta^*(q, \lambda) = q \tag{2-1}$$

$$\delta^*(q, wa) = \delta(\delta^*(q, w), a) \tag{2-2}$$

为了说明为什么这些式子是正确的，我们把这几个定义用到上面介绍过的几个简单例子上。首先，我们使用式（2-2）得到

$$\delta^*(q_0, ab) = \delta(\delta^*(q_0, a), b) \tag{2-3}$$

但是

$$\begin{aligned}\delta^*(q_0, a) &= \delta(\delta^*(q_0, \lambda), a)\\ &= \delta(q_0, a)\\ &= q_1\end{aligned}$$

把这个替换到式（2-3）中，得到

$$\delta^*(q_0, ab) = \delta(q_1, b) = q_2$$

上式和我们期望的结果相同。

2.1.2 语言和dfa对应的语言

给出接受器的准确定义后，我们下一步将给出和它相关联的语言的形式化定义。这种关联性是显而易见的。这里的语言指的是被自动机接受的所有符号串的集合。

定义2.2 dfa $M = (Q, \Sigma, \delta, q_0, F)$ 接受的语言是定义在Σ上被M接受的所有符号串的集合。形式化地表示为

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \in F\}$$

38

注意我们要求δ和δ^*是全函数。因为自动机的每一步迁移都是通过定义而唯一确定的，所以我们称这个自动机为确定型的。dfa处理Σ^*上的所有符号串，并给出是否接受该符号串的结果。不接受意味着停在非终态上，即：

$$\overline{L(M)} = \{w \in \Sigma^* : \delta^*(q_0, w) \notin F\}$$

例2.2 考虑图2-2中的dfa。

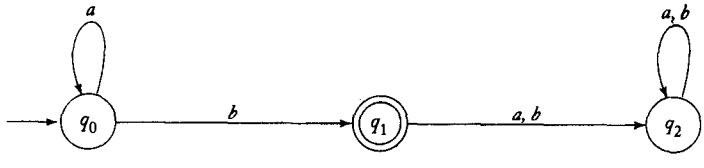


图 2-2

在画图2-2时，我们允许一条边标有两个标记。这种标有多个标记的边是两个或两个以上转移的速记：当输入符号和边标记上的任意一个符号匹配时，转移就发生。

图2-2中的自动机除非遇到b，否则将会一直停留在它的初态q_0上。如果这第一个b也是输入的最后一个符号，那么这个符号串会被接受。如果不是最后一个符号，那么自动机就会进入状态q_2，并且永远停留在这个状态中出不来。这个状态是陷阱状态（trap state）。从图中我们可以清楚地看出，这个自动机接受前面是任意个a，后面跟着一个b的所有符号串。其他类型的符号串都不接受。在集合表示中，这个自动机接受的语言表示为

$$L = \{a^n b : n \geqslant 0\}$$ □

从这些例子都可以看出，使用转换图表示有穷自动机很方便。尽管（根据例2.1和例2.2）可以按照转移函数及其扩展形式来画转换图，但是至此我们还没有证明图的结果确实是遵循了这些函数。在我们的讨论中，我们尽可能地使用图这种更直观的表示方法。既然如此，我们就必须保证我们没有被这种表示方法误导，并且保证图的方法和使用δ的形式化属性的方法是同样有效的。下面的初步结论确保了这一点。

39

定理2.1 设$M = (Q, \Sigma, \delta, q_0, F)$是确定型有穷接受器，$G_M$是和它相关联的转换图。那么对于任意$q_i, q_j \in Q$和$w \in \Sigma^+$，$\delta^*(q_i, w) = q_j$当且仅当在$G_M$中有一条从$q_i$到$q_j$标有$w$的通道。

证明：以例2.1为例可以看出这个命题显然是正确的。我们可以对w的长度作归纳来严格证明这个命题。假设这个命题对于所有$|v| \leqslant n$的符号串v都成立。对于长度为$n + 1$的任意符号串w，我们都可以把它写成

$$w = va$$

假设$\delta^*(q_i, v) = q_k$。因为$|v| = n$，所以在G_M中从q_i到q_k一定存在一条标有v的通道。但是如果$\delta^*(q_i, w) = q_j$，那么一定有转移函数$\delta(q_k, a) = q_j$，因此构造G_M时会存在标有a的边（q_k, q_j）。所以，G_M中，q_i和q_j之间有一条标有$va = w$的通道。因为对于$n = 1$结论显然为真，根据归纳证明，我们得出对于任意$w \in \Sigma^+$，都有

$$\delta^*(q_i, w) = q_j \tag{2-4}$$

即在G_M中，从q_i到q_j有一条标有w的通道。

反过来可以说明，如果存在着这样的通道，则有式（2-4），从而完成了证明。■

这个定理的结论很显然，所以似乎根本就不需要形式化的证明。可是我们却提供了证明，原因有两点。第一，尽管证明很简单，可是这是一个和自动机相关的归纳证明的典型例子。第二，这个命题的结论会被反复用到，因此声明并证明它有助于增强我们使用图的信心。和使用δ^*的属性相比，图使得我们的例子和证明更加清晰。

尽管图能够方便地表示自动机，但是其他类型的表示也仍然是有用的。例如，我们可以使用表格来表示函数δ。图2-3中的表格和图2-2是等价的。在图2-3中，行表示当前状态，列表示当前输入符号。表格的交叉处表示下一个状态。

	a	b
q_0	q_0	q_1
q_1	q_2	q_2
q_2	q_2	q_2

图 2-3

从这个例子我们显然可以看出，一个dfa可以很容易地用计算机程序实现。例如，写成一个简单的表格查找或一组“if”语句序列来实现。不同的应用存在不同的最佳实现方式或表示方式。转换图对于我们要完成的各种各样的命题来说都是一种方便的表示形式，所以我们在大多数论述中都会使用它。

在构造非形式化定义的语言对应的自动机时，我们都使用类似于高级程序设计语言中用到的推理。但是这种dfa的编程是乏味的，而且有时由于自动机功能不够强大，故而在概念上把问题复杂化了。

例2.3 找到一个确定型有穷接受器识别所有前缀为ab，定义在$\Sigma = \{a, b\}$上的符号串。

这里需要考虑的只有符号串的前两个符号；读入这两个符号后，符号串的余下部分就不必理会了。因此，我们可以设计一个四个状态的自动机来解决这个问题。这四个状态分别是：初态、两个识别ab并最终进入终态的状态，和一个非终态的陷阱状态。如果第一个符号是a，第二个符号是b，那么自动机就进入终态。而且，不管剩下的输入是什么，它都会停留在这个状态中。另外，如果第一个符号不是a，或者第二个符号不是b，那么自动机会进入非终态的陷阱状态。图2-4呈现了这个简单的解决方案。 □

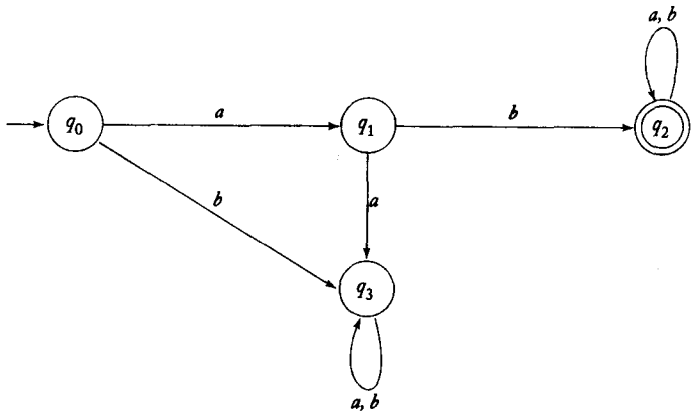


图 2-4

40 ~ 41

例2.4 构造一个dfa，接受{0, 1}上除包含有001子串以外的所有符号串。

为了判断是否出现了子串001，我们不仅需要知道当前的输入符号，而且要记得刚刚接受了一个还是两个0。我们让自动机进入一个特殊状态，一边跟踪这种情况，一边标记这种情况。就像程序设计语言中的变量名一样，状态名也可以是任意的，可以使用那些好记的名字。例如，已经接受了两个0的状态就叫00。

如果符号串以001开始，那么自动机会拒绝它。这就意味着必定存在一条从初态到非终态的路径标有001。为了方便，这个非终态记成001。这个状态一定是一个陷阱状态，因为一旦进入这个状态，后面是什么符号都无关紧要了。所有其他状态都是接受态。

这样我们就有了一个解决方案的基本框架。但是我们还需要考虑子串001出现在输入串中间的这种情况。我们必须定义Q和δ，这样无论我们做什么样的决定，自动机都会记得。在这种情况下，当读入一个符号时，我们需要知道在符号串中位于它左边的符号是什么，比如，之前的两个符号是不是00。如果我们用相关符号来标记状态，那么就很容易知道转移函数是什么样的了。例如，

$$\delta(00, 0) = 00$$

因为只有出现连续三个0时，才会使用这个转移函数。我们关心的仅仅是和当前符号最近的那两个符号，我们在dfa中用状态00来记录。图2-5是得到的完整的结果。从这个例子中，我们明白了便于记忆的状态名对于跟踪某些信息是很有帮助的。使用这个自动机跟踪一些符号串，比如100100和1010100，我们可以知道这个结果的确是正确的。 □

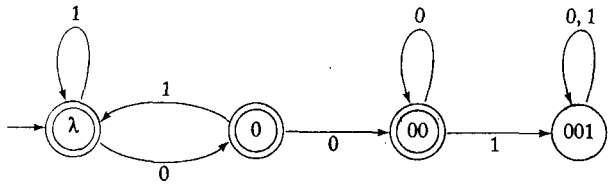


图 2-5

2.1.3 正则语言

每个有穷自动机都接受某种语言。如果我们考察所有可能的有穷自动机，我们就获得了一个和这些自动机相关的语言集合。我们把这些语言集合称作*族*（family）。能够被确定型有穷接受器接受的语言族是相当有限的。随着我们研究的深入，这个语言族中的语言的结构和性质将更加清晰。现在我们先简单地给出这个语言族的名字。

42

定义2.3 *一种语言L称为正则（regular）语言，当且仅当存在某个确定型有穷接受器M满足*

$$L = L(M)$$

例2.5 证明语言

$$L = \{awa : w \in \{a, b\}^*\}$$

是正则语言。为了证明这种或其他某种语言是正则语言，我们要为这种语言构造一个dfa。这种语言的dfa构造类似于例2.3，但是比它要更复杂些。这个dfa需要检查符号串是不是由a开始，以a结束，符号串的中间是什么无所谓。这个问题的复杂之处在于没有一个明确的方法可以判断符号串的结尾符号。解决这个困难的方法是当dfa遇到第二个a时，就进入了终态。如果符号串还没有结束，并且又发现一个b，自动机就从终态中跳出。然后继续按照这种方式扫描输入，每碰到a就进入终态。图2-6显示了这个问题的完整结果。我们仍然可以使用几个例子来验证这个自动机是否符合要求。经过一两个测试，显然就可以知道当且仅当符号串开始符号和结束符号都是a时，dfa才接受。因为我们构造出了接受这种语言的自动机，所以我们根据定义可以知道这种语言是正则语言。 □

例2.6 设L是例2.5中的语言。证明L^2也是正则语言。我们再次使构造dfa的方法来证明正则语言。我们首先写出语言L^2中的表达式形式，即

$$L^2 = \{aw_1\, aaw_2a : w_1, w_2 \in \{a, b\}^*\}$$

因此，我们需要一个自动机，能够识别由两个形式完全一样的符号串（只是形式相同，不要求值相同）顺序连接成的符号串。我们可以以图2-6为基础，把顶点q_3进行修改。这个状态不再是终态，我们从这个状态开始寻找第二个形式为awa的子串。为了识别第二个子串，我们复制第一部分的状态（分别取了新的名字），并且以q_3为第二部分的开始顶点。在aa出现的地方，

43

我们把整个符号串切分成两部分，自动机开始识别第二部分。这部分功能可以通过函数$\delta(q_3, a) = q_4$来实现。图2-7给出了完整结果。因为这个dfa接受L^2，所以它是正则语言。 □

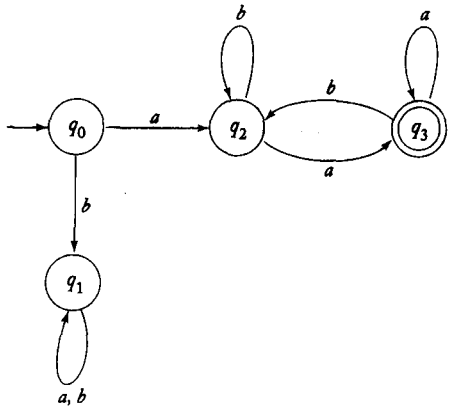


图 2-6

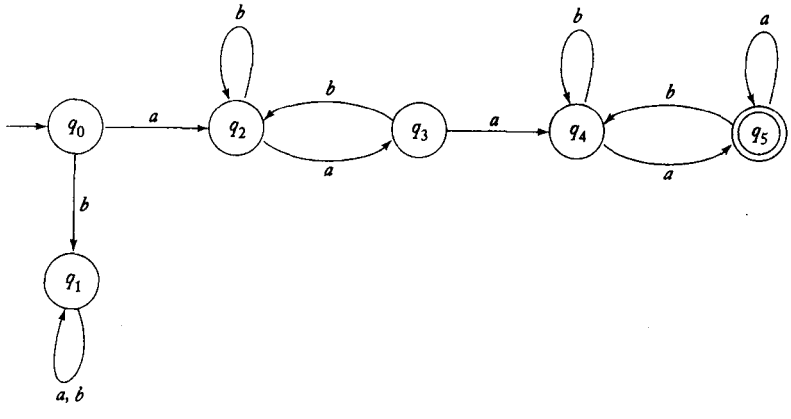


图 2-7

44

最后一个例子证实了我们的猜测，如果语言L是正则语言，那么它自身的连接L^2, L^3, …就都是正则语言。我们会在后面证明这个结论的正确性。

习题

1. 符号串0001，01001，0000110中，哪些可以被图2-1中的dfa接受？
2. 已知$\Sigma = \{a, b\}$，分别构造接受下面集合的dfa:
   (a) 只有一个a的符号串
   (b) 至少有一个a的符号串
   (c) a的个数不多于3的符号串●
   (d) 至少有一个a并且恰有两个b的符号串
   (e) 恰有2个a并且有多于两个b的符号串
3. 如果我们修改图2-6，使得q_3成为非终态而q_0, q_1, q_2成为终态。证明这样得到的dfa接受语言L。

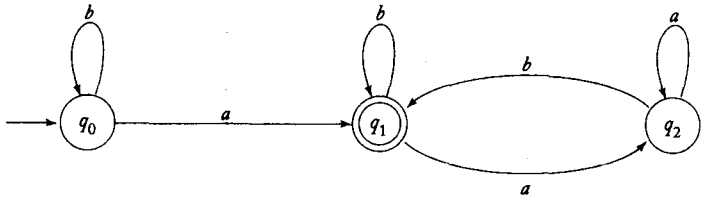
4. 归纳前一个习题的结论。特别地，证明如果$M = (Q, \Sigma, \delta, q_0, F)$和$\hat{M} = (Q, \Sigma, \delta, q_0, Q - F)$是两个确定型有穷接受器，那么就有$L(M) = L(\hat{M})$。

5. 构造分别接受下面语言的dfa:

(a) $L = \{ab^5wb^2 : w \in \{a, b\}^*\}$ ●

(b) $L = \{w_1abw_2 : w_1 \in \{a, b\}^*, w_2 \in \{a, b\}^*\}$

6. 给出下图描述的自动机接受的语言集合的描述。你能够口头描述出这种语言吗？



7. 在$\Sigma = \{a, b\}$上，分别构造接受下面语言的dfa:

(a) $L = \{w : |w| \bmod 3 = 0\}$ ●

(b) $L = \{w : |w| \bmod 5 \neq 0\}$

(c) $L = \{w : n_a(w) \bmod 3 > 1\}$

45 (d) $L = \{w : n_a(w) \bmod 3 > n_b(w) \bmod 3\}$ ●

(e) $L = \{w : (n_a(w) - n_b(w)) \bmod 3 > 0\}$

(f) $L = \{w : (n_a(w) + 2n_b(w)) \bmod 3 < 2\}$

★8. 一个符号串上的“run”指的是符号串中包含由至少两个相同符号构成的子串。比如，符号串$abbbaab$包含一个符号b的长度为3的“run”和一个符号a的长度为2的“run”。在$\{a, b\}$上，分别构造下列语言的dfa。

(a) $L = \{w : w$不包含长度小于4的“run”$\}$

(b) $L = \{w :$ 符号的每个“run”的长度或者是2或者是3$\}$

(c) $L = \{w :$ 每个符号串至多包含两个长度为3的符号的“run”$\}$

(d) $L = \{w :$ 每个符号串就包含两个长度为3的符号的“run”$\}$

9. 在$\{0, 1\}$上，分别构造接受下列语言的dfa。

(a) 每个00后面都紧跟着1的符号串。例如，符号串101，0010，0010011001都属于这种语言，而0001和00100不属于。●

(b) 所有包含00，却不包含000的符号串。

(c) 最左边的符号和最右边的符号不同的符号串。

(d) 每4个符号构成的子串至多有两个0。例如，001110和011001都属于这种语言，而10010不属于，因为它的一个子串0010有三个0。●

(e) 所有长度不小于5的符号串，从左边起第4个符号跟最右边的符号不同。

(f) 所有最左边的两个符号和最右边的两个符号相同的符号串。

★10. 在$\{0, 1\}$上构造dfa完成下面的功能。把输入的符号串看成是二进制数，如果这个二进制数能够被5整除，那么就接受这个符号串。例如，0101和1111，分别表示整数5和15，因此可以被接受。

11. 证明：语言$L = \{vwv : v, w \in \{a, b\}^*, |v| = 2\}$是正则语言。

12. 证明语言$L = \{a^n : n \geqslant 4\}$是正则语言。
13. 证明语言$L = \{a^n : n \geqslant 0, n \neq 4\}$是正则语言。●
14. 证明语言$L = \{a^n : n = i + jk, i, k$固定，$j = 0, 1, 2, \cdots\}$是正则语言。
15. 证明C语言中的所有实数的集合是正则语言。
16. 证明：如果L是正则语言，那么$L - \{\lambda\}$也是正则语言。
17. 使用式（2-1）和式（2-2），证明对于所有$w, v \in \Sigma^*$，都有

$$\delta^*(q, wv) = \delta^*(\delta^*(q, w), v)$$

46

18. 假设L是图2-2中自动机接受的语言。构造接受语言L^2的dfa。
19. 假设L是图2-2中自动机接受的语言。构造接受语言$L^2 - L$的dfa。
20. 假设L是例2.5中的语言，证明L^*是正则语言。
21. 设G_M是某个确定型有穷接受器M的转换图，证明下列结论：
    (a) 如果$L(M)$是无穷的，那么在G_M中一定存在至少一个回路。在这个回路中，存在从起点到回路中某个顶点的路径，以及从这个顶点到某个终点的路径。
    (b) 如果$L(M)$是有穷的，那么不存在这样的回路。●
22. 我们定义截短（truncate）运算为从任何符号串的最右端删除一个符号。例如truncate（*aaaba*）是*aaab*。把这个操作扩展到语言上的定义有

$$\text{truncate}(L) = \{\text{truncate}(w) : w \in L\}$$

    对于接受任一给定正则语言L的dfa，给出如何构造一个接受truncate(L) 的dfa。由此过程来证明：如果L是不包含λ的正则语言，那么truncate(L) 也是正则语言。
23. 设$x = a_0a_1\cdots a_n$，$y = b_0b_1\cdots b_n$，$z = c_0c_1\cdots c_n$，都是按照例1.17定义的二进制数。证明下面用三元组构成的符号串集合是正则语言

$$\begin{pmatrix} a_0 \\ b_0 \\ c_0 \end{pmatrix}\begin{pmatrix} a_1 \\ b_1 \\ c_1 \end{pmatrix}\cdots\begin{pmatrix} a_n \\ b_n \\ c_n \end{pmatrix}$$

    其中a_i, b_i, c_i使得$x + y = z$。
24. 某一给定dfa接受的语言是唯一的，然而接受同一种语言的dfa却有很多。构造一个只有6个状态的dfa，使它接受图2-4中dfa接受的语言。●

2.2 非确定型有穷接受器

如果我们允许有穷接受器以非确定的方式工作，那么有穷接受器就会变得更加复杂。非确定论虽然强大，然而最初看来却是一个不寻常的想法。因为我们通常认为计算机的工作过程是完全确定的，每一步都没有选择的余地。然而，正如我们接下来将要看到的一样，非确定论是一个有用的表示法。

47

2.2.1 非确定型接受器的定义

非确定论意味着自动机器的每一步迁移都要选择。在非确定型接受器中，每一步的迁移不是唯一的，而是一个由所有可能的迁移构成的集合。我们使用转移函数这种形式化的方法

来定义，所以转移函数的值域是所有可能状态的集合。

定义2.4 非确定型有穷接受器（nondeterministic finite accepter, nfa）是一个五元组

$$M = (Q, \Sigma, \delta, q_0, F)$$

其中，Q, Σ, q_0和F同确定型有穷接受器的定义相同，但δ的定义为

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \rightarrow 2^Q$$

注意这个定义和dfa的定义有三个主要的不同点。在非确定型接受器中，δ的值域是幂集2^Q。因此，这个函数的函数值不是Q中的一个元素，而是它的一个子集。这个子集定义了通过转移函数可以得到的所有可能的状态。例如，如果当前状态是q_1，读入符号a，并且有

$$\delta(q_1, a) = \{q_0, q_2\}$$

那么，q_0或q_2都可能是nfa的下一个状态。同样地，我们允许λ作δ的第二个参数。这意味着nfa可以不使用任何输入符号就可以发生转移。尽管我们假定输入机制能够只能向右迁移，但是它也可能在某些迁移步骤上保持不动。最后，在nfa中，集合$\delta(q_i, a)$ 可以为空。这种情况表示这种特殊情况的转移函数没定义。

像dfa一样，非确定型接受器也可以用转移图表示。Q决定顶点。当且仅当集合$\delta(q_i, a)$ 包含q_j时，图中才存在标有a的边（q_i, q_j）。注意既然a可以是空串，图中就可以存在标记为λ的边。

如果机器读完一个符号串，经过一系列的迁移后进入终态，那么我们称这个符号串可以被这个nfa接受。只有无论自动机如何迁移，都无法进入终态时，我们称这个符号串被拒绝（不被接受）。因此，非确定性被看成具有“直觉的”洞察力，因为它每次都能选择最佳的迁移方案（假设nfa想接受每一个符号串）。

例2.7 观察图2-8中的转移图。这是一个非确定型接受器，因为有两条从q_0出发的转移边标有a。 □

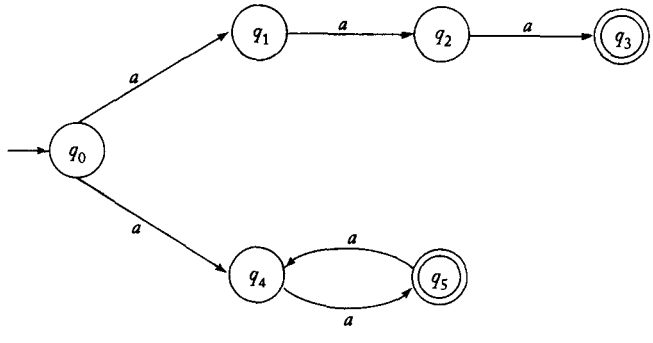


图 2-8

例2.8 图2-9表示的是一个非确定型自动机。之所以说它是非确定的，不仅因为从同一个顶点出发的几条边都有相同的标记，而且因为它有λ转移。一些转移函数，比如$\delta(q_2, 0)$，在这个图中没有定义。这个可以看成是向空集定义的转移，即：$\delta(q_2, 0) = \varnothing$。这个自动机接受$\lambda$，1010和101010，但不接受110和10100。注意10存在两种不同的通道，一种到q_0，另一种到q_2。尽管q_2不是终态，但是自动

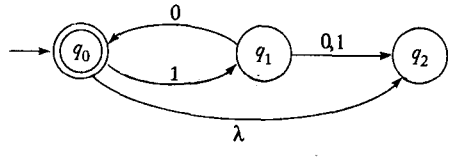


图 2-9

机仍然接受这个符号串，因为另外一种通道可以到终态。

此外，可以扩展转移函数的定义，这样的话，它的第二个参数就可以是符号串。我们规定扩展转移函数δ^*为，如果

$$\delta^*(q_i, w) = Q_j$$

那么自动机开始于状态q_i，读入w后，自动机可能处于的所有状态构成的集合为Q_j。同式（2-1）和式（2-2）类似，可以对δ^*进行递归定义，这里就不特别强调了。通过转移图可以给出一个更有价值的定义。 □ 48~49

定义2.5 对于nfa，扩展的转移函数定义为：$\delta^*(q_i, w)$ 包含q_j，当且仅当转移图中从顶点q_i到顶点q_j存在标记为w的通道。这个定义对于所有$q_i, q_j \in Q$和$w \in \Sigma^*$都成立。

例2.9 图2-10表示一个nfa。在这个nfa中，存在几个λ转移和像$\delta(q_2, a)$ 这样的没有定义的转移。

假设我们想要找到$\delta^*(q_1, a)$ 和$\delta^*(q_2, \lambda)$。从q_1到它自身有一条标记为a，包含两个λ转移的通道。使用两次λ边，我们就可以发现到q_0和q_2也有包含λ转移的通道。因此，

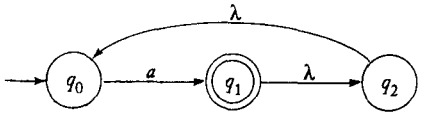


图 2-10

$$\delta^*(q_1, a) = \{q_0, q_1, q_2\}$$

既然在q_2和q_0之间存在λ边，我们就此可以知道$\delta^*(q_2, \lambda)$ 包含q_0。同样，因为任何状态如果不迁移都可以到达它自身，且不用读入任何符号，所以$\delta^*(q_2, \lambda)$ 也包含q_2。

因此

$$\delta^*(q_2, \lambda) = \{q_0, q_2\}$$

尽可能地使用λ转移，我们就可以得到

$$\delta^*(q_2, aa) = \{q_0, q_1, q_2\}$$ □

尽管通过带有标记的通道来定义δ^*不太正式，但是这样有助于我们近距离地理解这个名词。定义2.5是正确的，因为在任何两个顶点v_i和v_j之间，要么存在一条标记为w的通道，要么不存在。这说明δ^*是完全定义的。我们总是可以使用这个定义来找$\delta^*(q_i, w)$，这一点也许很难看出来。

在1.1节中，我们给出了一个找出两个顶点间所有简单路径的算法。但是在这里我们不能直接使用这个算法，因为正如例2.9所示，带有标记的通道不总是简单路径。我们可以修改这个简单路径的算法，去掉不能有重复的顶点和边的限制。这个新的算法就可以产生一系列长度为1, 2, 3, …的通道。 50

这里仍然存在着困难。给定w，一个标记为w的通道有多长？答案不是显而易见的。在例2.9中，顶点q_1和q_2之间标记为a的通道长度为4。这是由于λ转移造成的，因为它，通道的长度增加了，可是标记的符号没有变化。通过观察可以得出，如果两个顶点v_i和v_j之间存在标记为w的通道，那么它们之间就一定存在长度不大于$\Lambda + (1 + \Lambda)|w|$标记为$w$的通道。其中，$\Lambda$是图中$\lambda$边的数目。证明如下：因为$\lambda$边可以重复，所以总存在这样一条通道：每一个重复的$\lambda$边都被一个标记为非空符号的边隔开。否则，通道中包含标记为λ的回路，这条通道可以由一个简单路径替代，并且不改变原通道上的标记。这个命题的详细证明我们留作习题。

有了上面的基础，我们给出计算$\delta^*(q_i, w)$的方法。我们考察所有起点为v_i，长度至多是$\Lambda + (1 + \Lambda)|w|$的通道。从中选出标记为$w$的通道。选出的通道的终点就是集合$\delta^*(q_i, w)$中的元素。

正如确定型接受器一样，我们也可以使用递归的方式来定义δ^*。不幸的是，结果并不像想象中的那么透明，非确定型接受器的扩展转移函数很难按照确定型的方法来定义。因此，我们更倾向于使用定义2.5，因为它更直观，更容易操作。

和dfa一样，通过扩展转移函数给出nfa接受的语言的形式化定义。

定义2.6 nfa $M = (Q, \Sigma, \delta, q_0, F)$ *接受的语言L是按照上述方式被接受的所有符号串的集合。即*

$$L(M) = \{w \in \Sigma^* : \delta^*(q_0, w) \cap F \neq \emptyset\}$$

总之，语言包含的所有符号串w都满足：在转移图中，从起始顶点到某个终点存在标记为w的通道。

例2.10 图2-9中自动机接受的是什么语言？从图中不难看出，只有当输入是重复的符号串10，或者空串时，nfa才会停在终态。因此，自动机接受的语言是$L = \{(10)^n : n \geq 0\}$。

51

当输入是符号串$w = 110$时，自动机会怎么办呢？读入前缀11后，自动机进入状态q_2，可是转移函数$\delta(q_2, 0)$是没有定义的。我们称这种状况是*死格局*（dead configuration），并且可以把这种情况简单地看作是自动机的没有进一步行动的停机。但是我们必须记住这种状况是不严密的，并且可能带来误解释的危险。我们认为准确的定义是

$$\delta^*(q_0, 110) = \emptyset$$

因此，处理$w = 110$，最终不能进入终态。所以，这个符号串不被接受。 □

2.2.2 为什么需要非确定型

在非确定型机器推理的过程中，我们要谨慎使用直觉表示。直觉很容易使我们误入歧途，我们必须给出准确的论断来证实我们的结论。非确定型是一个较难的概念。数字计算机完全是确定型的；在任何时刻，根据它的输入和当前状态都可以预测它的下一个状态，并且这个状态是唯一的。因此，人们很自然地就会问，为什么我们要研究非确定型机器呢？我们试图为现实系统建模，那么我们为什么还要考虑这些非确定型的因素呢？我们通过以下几种方式来回答这个问题。

许多确定型算法要求在某些阶段做出选择。一个典型的例子是决策程序。我们通常并不知道最佳的策略，但是我们可以使用具有回溯功能的穷举查找来找到最佳策略。当同时存在几种策略时，我们选择其中的一种走下去，直到我们可以判断这种策略是否是最佳的。如果不是，我们就回退到最后的决策点，探索其他的策略。非确定型算法可以做出最佳的选择来解决这个问题，并且不需要回溯。然而，确定型算法必须要增加一些额外工作才能完全模拟非确定型算法。因此，非确定型机器可以看作是查找回溯算法的模型。

非确定型有时可以有助于简单地解决问题。看图2-8中的nfa。很明显，开始的时候就需要做出一个选择。选择上面，则符号串a^3会被接受。相反，选择下面，则只能接受包含偶数个a的符号串。这个nfa接受的语言是$\{a^3\} \cup \{a^{2n} : n \geq 1\}$。当然我们也可以找到接受这种语言的dfa，但是非确定型接受器看起来更自然。这种语言是两个不同集合的并。非确定型接受器就让我

们在这个并集合上进行选择。确定型的解不像非确定型这样和定义有明显的联系。接下来，我们将给出一些证明非确定型有用的更有说服力的例子。 52

同样地，非确定型可以简明地描述一些复杂语言。注意包含非确定元素的语法定义。已知

$$S \to aSb|\lambda$$

我们在任何时候都面临着选择：使用第一个产生式，还是第二个产生式。使用这两条推导规则，我们可以区分很多不同的符号串。

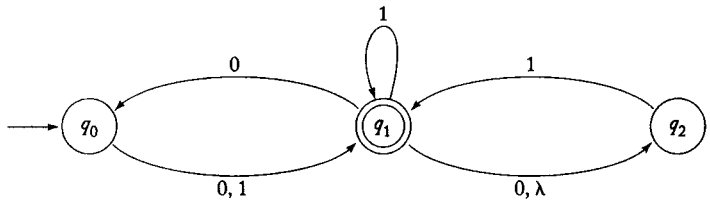
最后，引入非确定型还有着技术方面的原因。正如我们看到的，有些结论使用nfa更容易得到。我们下一个主要命题是指明这两种自动机并没有什么本质上的区别。因此，允许非确定型常常可以简化形式化证明，而且又不影响结论的一般化。

习题

1. 详细证明前一节的命题：如果在转移图中存在标记为w的通道，那么一定存在长度不大于$\Lambda + (1 + \Lambda)|w|$标记为$w$的通道。
2. 构造一个dfa，使得它能接受图2-8中nfa接受的语言。
3. 在图2-9中，确定$\delta^*(q_0, 1011)$和$\delta^*(q_1, 01)$。
4. 在图2-10中，确定$\delta^*(q_0, a)$和$\delta^*(q_1, \lambda)$。●
5. 在图2-9的nfa中，确定$\delta^*(q_0, 1010)$和$\delta^*(q_1, 00)$。
6. 设计一个状态数不超过5的nfa，接受集合$\{abab^n : n \geqslant 0\} \cup \{aba^n : n \geqslant 0\}$。
7. 构造一个3个状态的nfa，接受语言$\{ab, abc\}^*$。●
8. 你认为可以找到状态少于3个的nfa来解决习题7的问题吗？●
9. (a) 构造3个状态的nfa，接受语言

$$L = \{a^n : n \geqslant 1\} \cup \{b^m a^k : m \geqslant 0, k \geqslant 0\}$$

   (b) 你认为存在状态数少于3的nfa能接受（a）中的语言吗？
10. 构造4个状态的nfa，接受语言$L = \{a^n : n \geqslant 0\} \cup \{b^n a : n \geqslant 1\}$。
11. 符号串00，01001，10010，000，0000中，哪些可以被下面的nfa接受？ 53



12. 给出图2-10中nfa接受的语言的补集。
13. 设L是图2-8中nfa接受的语言。构造接受语言$L \cup \{a^5\}$的nfa。
14. 给出习题12中语言的简单描述。
15. 构造接受$\{a\}^*$的nfa，并且满足：若从它的转移图中去掉一条边（没有任何其他变化），则得到的自动机可以接受$\{a\}$。●
16. 习题15可以用一个dfa来解决吗？如果可以，给出这个解。否则，证明你的结论。
17. 定义2.6修改后如下所示：一个带有多个初态的nfa是一个五元组

$$M = (Q, \Sigma, \delta, Q_0, F)$$

其中，$Q_0 \subseteq Q$是所有可能的初态集合。这个自动机接受的语言定义为

$$L(M) = \{w : \delta^*(q_0, w) \text{ 包含} q_f\text{，对于任意} q_0 \in Q_0, q_f \in F\}$$

证明：对于任何一个多初态nfa，总存在某个单初态nfa接受同样的语言。●

18. 假设习题17增加一条限制条件$Q_0 \cap F = \varnothing$。这个限制条件会影响结论吗？

19. 使用定义2.5证明：对于所有的$q \in Q$和w，$v \in \Sigma^*$，都有

$$\delta^*(q, wv) = \bigcup_{p \in \delta^*(q, w)} \delta^*(p, v)$$

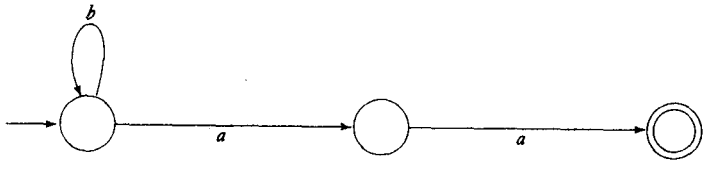
20. 满足下列条件的dfa称为不完全（incomplete）dfa。

(a) 没有λ转移；

(b) 对于所有的$q \in Q$和$a \in \Sigma$，$\delta(q, a)$至多包含一个元素。

54 显然，根据这样的定义，自动机的迁移选择是唯一的。

已知$\Sigma = \{a, b\}$，把不完全dfa转化成一个标准dfa。●



2.3 确定型有穷接受器和非确定型有穷接受器的等价性

现在我们问一个基本问题。dfa和nfa有哪些不同？显然，它们的定义不同，但是这并不意味着二者之间存在着本质差异。为了进一步探讨这个问题，我们引入自动机等价的定义。

定义2.7 对于两个有穷接受器M_1和M_2，如果

$$L(M_1) = L(M_2)$$

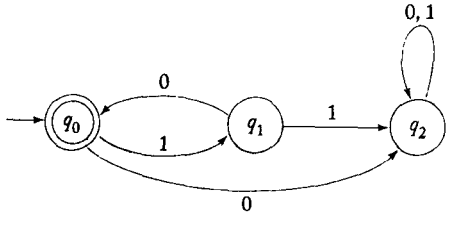
即：它们接受相同的语言。那么，这两个有穷接受器等价。

正如以前提到的，对于一种给定的语言，接受它的接受器很多，因此，任何一个dfa或nfa都有多个等价的接受器。

例2.11 图2-11中表示的dfa和图2-9的nfa等价，因为它们都接受语言

$$\{(10)^n : n \geqslant 0\}$$

□



55

图 2-11

当我们比较两类自动机时，我们总会问哪种自动机更强大。更强大指的是哪种自动机可以完成另外一种自动机无法实现的东西。让我们针对有穷接受器考虑这个问题。既然dfa在本质上是一种更加严格的nfa，那么显然，dfa接受的任何语言nfa也应该可以接受。但是，反之就不是那么明显了。我们既然增加了非确定型这种机制，那么起码应该存在一种nfa能接受的语言，我们找不到dfa来接受。但事实证明不是这样的。dfa和nfa这两类是同样强大的：nfa能接受的语言，dfa也可以接受。

这个结论不是很明显，因此需要证明。这个命题，正如本书中大多数命题一样，要采取构造性证明的方法。这就意味着我们可以给出一种把nfa转化成等价的dfa的方法。这个构造方法不难理解。一旦原则清晰了，那么这就可以成为严格证明的开始点。构造证明的基本原理如下。nfa读入符号串w后，我们并不确切地知道自动机会进入哪一个状态，但是我们可以说它一定是处于可能的状态集合的某一个状态中，记作$\{q_i, q_j, \cdots, q_k\}$。而一个等价的dfa读入同样的符号串后，一定会处于某一个确定的状态。那么，我们如何把这两种情况对应起来呢？这里有个小窍门：把上面的状态集合标记成dfa的状态，使得读入符号串w后，等价的dfa进入一个标记为$\{q_i, q_j, \cdots, q_k\}$的状态中。既然$|Q|$个状态的集合只有$2^{|Q|}$个子集，那么对应的dfa就有有穷个状态。

上面介绍的构造性方法的主要工作集中于nfa的分析，从而获得输入和可能的状态之间的对应关系。在形式化地描述上面的方法之前，我们先使用下面的简单例子解释。

例2.12 把图2-12中的nfa转化成等价的dfa。这个nfa的初态为q_0，因此dfa的初态标记为$\{q_0\}$。读入符号a后，nfa处于状态q_1中，或者经过λ转移，进入状态q_2。因此，对应的dfa一定存在一个标记为$\{q_1, q_2\}$的状态，并包含一个转移函数

$$\delta(\{q_0\}, a) = \{q_1, q_2\}$$

在状态q_0，输入为b时，nfa没有指定转移函数，因此，

$$\delta(\{q_0\}, b) = \varnothing$$

状态标记为$\varnothing$表示nfa不可能迁移，也就是说，自动机不接受这个符号串。因此，这个状态应该对应于dfa的一个非终态的陷阱状态。

56

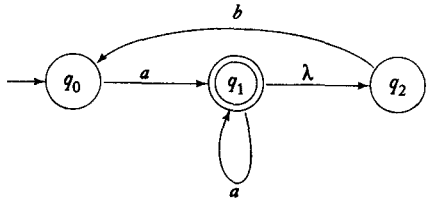


图 2-12

现在我们已经引入了dfa的状态$\{q_1, q_2\}$，因此我们需要确定这个状态的输出转移函数。我们已知dfa的这个状态对应于nfa中的两个可能状态，因此我们可以回来看看nfa。如果nfa处于状态q_1，读入符号a，那么它会进入状态q_1。而且，nfa可以从状态q_1做λ转移进入状态q_2。如果nfa处于状态q_2，读入同样的符号，那么就找不到指定的转移函数。因此，

$$\delta(\{q_1, q_2\}, a) = \{q_1, q_2\}$$

类似地，

$$\delta(\{q_1, q_2\}, b) = \{q_0\}$$

至此，对每个状态都定义了所有的转移函数。最后的结果，如图2-13所示，是一个和我们开始提出的nfa等价的dfa。图2-12中的nfa接受所有满足条件$\delta^*(q_0, w)$包含q_1的符号串。为了使对应的dfa接受每一个这样的符号串w，任何包含状态q_1的标记都必须设为终态。 □

57

定理2.2 设L是被非确定型有穷接受器$M_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$接受的语言。那么，一定存在一个确定型有穷接受器$M_D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$，满足

$$L = L(M_D)$$

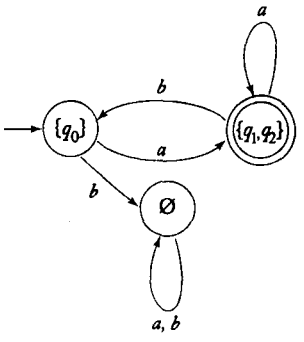


图 2-13

证明：给定M_N，我们使用下面的程序nfa_to_dfa来构造接受器M_D的转移图G_D。为了便于理解构造函数，记住G_D必须具有某些性质。每个顶点必须有$|\Sigma|$条输出边，并且每条边必须标有不同的Σ元素。在构造的过程中，一些边可能会漏掉，但是我们不断地使用这个程序从而保证不漏掉任何边。

程序：nfa_to_dfa

1. 从顶点$\{q_0\}$开始构造图G_D。把这个顶点作为起始顶点。

2. 重复下面的步骤，直到没有边被漏掉。

取G_D的一个顶点$\{q_i, q_j, \cdots, q_k\}$，该顶点没有标记为$a \in \Sigma$的输出边。

计算$\delta^*(q_i, a), \delta^*(q_j, a), \cdots, \delta^*(q_k, a)$。

然后，根据所有这些δ^*的并集得到集合$\{q_l, q_m, \cdots, q_n\}$。

如果G_D中不存在标记为$\{q_l, q_m, \cdots, q_n\}$的顶点，那么在$G_D$中添加一个标记为$\{q_l, q_m, \cdots, q_n\}$的顶点。

在G_D中增加一条从$\{q_i, q_j, \cdots, q_k\}$到$\{q_l, q_m, \cdots, q_n\}$的边，标记为$a$。

3. 对于G_D中的每一个标记包含$q_f \in F_N$的状态，都把它作为终点。

4. 如果M_N接受λ，那么G_D中的顶点$\{q_0\}$也是终点。

很明显，这个程序总会停止。每次经过步骤2的循环时，G_D都会增加一条边。但是G_D至多有$2^{|Q_N|}|\Sigma|$条边，所以，循环总会停下来。为了证明这个构造的正确性，我们会通过对输入符号串的长度进行归纳来证明。

假设对于长度不超过n的所有符号串v，在G_N中存在从q_0到q_i的标记为v的通道，这意味着在G_D中存在从$\{q_0\}$到状态$Q_i = \{\cdots, q_i, \cdots\}$的标记为$v$的通道。现在考虑任何符号串$w = va$，以及在$G_N$中的从$q_0$到$q_i$的标记为$w$的通道。从$q_0$到$q_i$一定存在标记为$v$的通道，而从$q_i$到$q_l$有标记为$a$的边（或一系列的边）。根据归纳假设可知，在$G_D$中一定存在一条从$\{q_0\}$到$Q_i$的标记为$v$的通道。但根据构造可知，在$G_D$中一定存在从$\{q_0\}$到某个状态的边标记包含$v$。因此，对于长度为$n + 1$的所有符号串归纳假设也成立。因为对于$n = 1$命题显然成立，所以对于所有的$n$命题都成立。于是得到结论，如果$\delta_N^*(q_0, w)$包含终态$q_f$，那么$\delta_D^*(q_0, w)$也包含终态。为了证明的完整性，我们把命题倒过来，即证明：如果$\delta_D^*(q_0, w)$包含q_f，那么$\delta_N^*(q_0, w)$也包含q_f。 ■

58

尽管已经证明了这个命题的正确性，但是为了简洁的需要，我们还是只列出了主要的步骤。我们会在这本书剩下的部分继续这方面的练习。我们只强调证明过程中的主要想法，至

于一些细节，读者可以自己补上。

上面的构造证明虽然枯燥，但却十分重要。我们再举一个例子来帮助读者理解构造证明的所有步骤。

例2.13 把图2-14的nfa转化成等价的确定型自动机。因为$\delta_N(q_0, 0) = \{q_0, q_1\}$，所以我们在$G_D$中引入状态$\{q_0, q_1\}$，在$\{q_0\}$和$\{q_0, q_1\}$之间增加一条标记为0的边。同样地，已知$\delta_N(q_0, 1) = \{q_1\}$，我们增加状态$\{q_1\}$，在它和$\{q_0\}$之间增加标记为1的边。

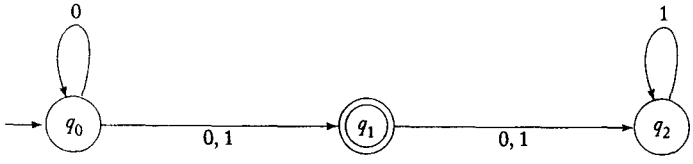


图 2-14

现在还有很多漏掉的边，因此我们继续使用定理2.2中的构造方法。使$a = 0, i = 0, j = 1$，我们可以得到

$$\delta_N^*(q_0, 0) \cup \delta_N^*(q_1, 0) = \{q_0, q_1, q_2\}$$

这就形成了一个新的状态$\{q_0, q_1, q_2\}$，和一个新的转移函数

$$\delta_D(\{q_0, q_1\}, 0) = \{q_0, q_1, q_2\}$$

接下来，使$a = 1, i = 0, j = 1, k = 2$，得到

$$\delta_N^*(q_0, 1) \cup \delta_N^*(q_1, 1) \cup \delta_N^*(q_2, 1) = \{q_1, q_2\}$$

这样又有必要引入另一个状态$\{q_1, q_2\}$。在这点上，我们构造了部分的自动机，如图2-15所示。因为还有漏掉的边，所以我们继续这个过程直到得到如图2-16所示的最终的完整结果。 □ 59

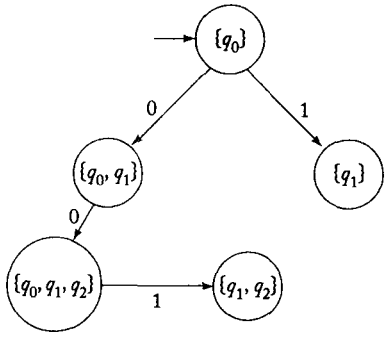


图 2-15

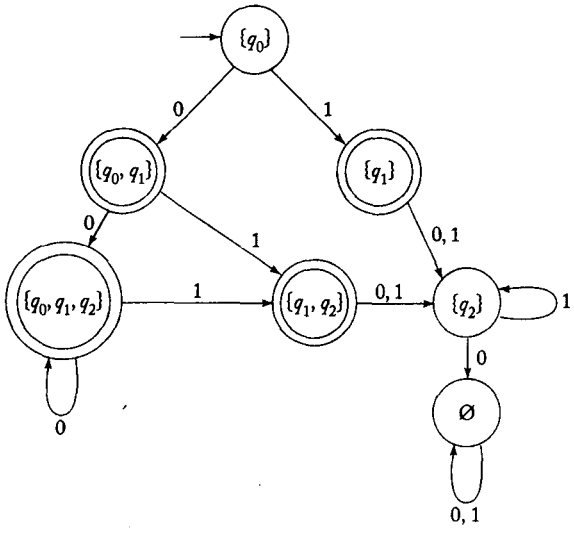


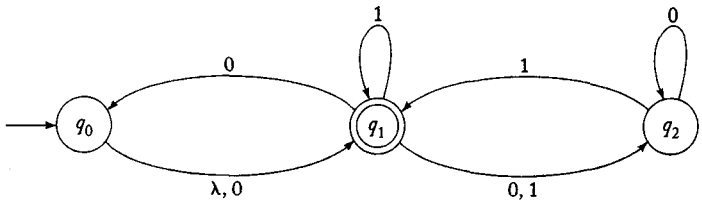
图 2-16

60

我们从定理2.2中得到的一个重要结论是被nfa接受的语言都是正则的。

习题

1. 使用定理2.2种的构造方法把图2-10中的nfa转化成dfa。你能够更直接地看出一个更简单的答案吗？
2. 把2.2节习题11的nfa转化成为一个等价的dfa。●
3. 把下面的nfa转化成等价的dfa。



4. 完成定理2.2的详细证明。给出下面结论的详细证明：如果$\delta_D^*(q_0, w)$ 包含q_f，那么$\delta_N^*(q_0, w)$ 也包含q_f。
5. 对于任意nfa $M = (Q, \Sigma, \delta, q_0, F)$，$L(M)$ 的补集等于集合$\{w \in \Sigma^* : \delta^*(q_0, w) \cap F = \varnothing\}$是否成立？如果成立，给出证明。否则，给出反例。
6. 对于任意nfa $M = (Q, \Sigma, \delta, q_0, F)$，$L(M)$ 的补集与集合$\{w \in \Sigma^* : \delta^*(q_0, w) \cap (Q - F) \neq \varnothing\}$相等是否成立？如果成立，给出证明。否则，给出反例。
7. 证明：不论一个nfa有多少个终态，总存在一个与之等价的只有一个终态的nfa。dfa是否存在类似的结论？●
8. 构造一个没有λ转移，且只有一个终态的nfa，接受集合$\{a\} \cup \{b^n : n \geq 1\}$。●

★9. 设L是不包含λ的正则语言。证明：存在一个没有λ转移，且只有一个终态的nfa可以接受语言L。

10. 根据2.2节习题17中关于nfa的定义，类似给出一个具有多个初态的dfa的定义。是否总存在一个等价的单初态dfa？
11. 证明所有的有穷语言都是正则语言。●
12. 证明：如果L是正则语言，那么L^R也是。
13. 给出图2-16中dfa接受的语言的简单口头描述。根据这个描述构造另一个与之等价的dfa，要求构造的dfa的状态数要更少。

61

★14. 设L是任何语言。定义$even(w)$ 为抽取符号串w中偶数位置的符号构成的符号串。

即：如果

$$w = a_1a_2a_3a_4\cdots$$

那么

$$even(w) = a_2a_4\cdots$$

根据上述定义，我们可以定义语言

$$even(L) = \{even(w) : w \in L\}$$

证明：如果L是正则语言，那么$even(L)$ 也是。●

15. 已知语言L，我们把L中每个符号串的最左端的两个符号去掉，获得的新语言是$chop2(L)$。

形式化的定义为

$$chop2(L) = \{w : vw \in L, \text{其中} |v| = 2\}$$

证明：如果L是正则的，那么$chop2(L)$也是正则的。●

2.4 减少有穷自动机中状态的化简*

任何一种dfa都定义了唯一的一种语言，但是反之就不一定对了。对于一种给定的语言，有很多dfa都可以接受它。这些等价的自动机，它们的状态数目可能差别很大。到目前为止，我们考虑的问题，无论什么样的解都一样能使人满意。但是如果把这些结果应用到实际背景中，就会存在孰优孰劣的问题了。

例2.14 输入一些测试符号串，会很快发现图2-17a和图2-17b中描述的两个dfa等价。我们可能会注意到图2-17a具有一些明显但并不是很本质的特点。在自动机中，状态q_5根本没有用，因为从初态q_0根本到不了它。这种状态是不可达的，因此，从自动机中去掉它（以及所有和它相关的转移）并不会影响自动机接受的语言。但是即使去掉了q_5，第一个自动机仍然存在着冗余。第一次迁移不管是使用$\delta(q_0, 0)$，还是使用$\delta(q_0, 1)$，后面的可达状态都是相同的。第二个自动机把这两个迁移合并了。 □ 62

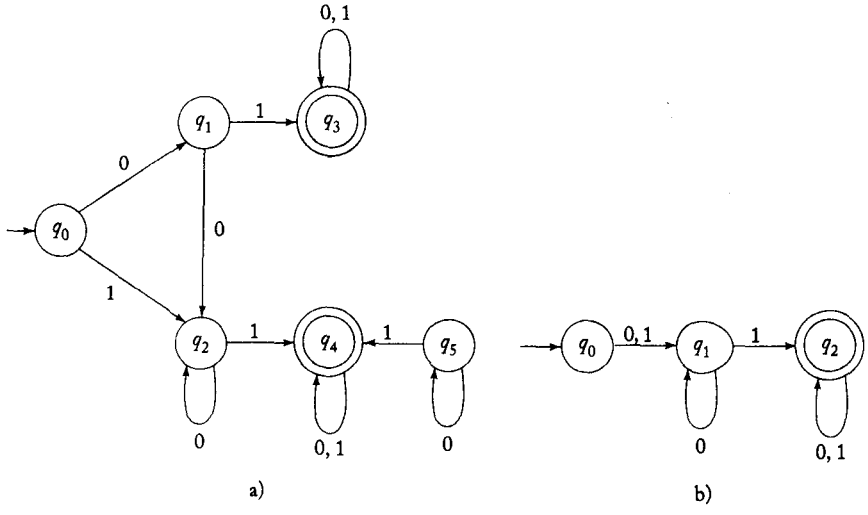


图 2-17

从严格的理论意义上讲，选择图2-17b中的自动机而不选择图2-17a中的自动机，几乎是没有理由的。但是无论如何，从简单的角度上讲，第二个是更值得推荐的。以计算为目的的自动机的表示，需要的空间大小与其状态数成正比。考虑到存储效率，就应该尽可能地减少状态数。我们现在描述一种能够达成这个目的的算法。

定义2.8 对于所有的$w \in \Sigma^*$，如果

$$\delta^*(p, w) \in F \text{意味着} \delta^*(q, w) \in F$$

并且

$$\delta^*(p, w) \notin F \text{意味着} \delta^*(q, w) \notin F$$

则称dfa的两个状态p和q是不可区分的（indistinguishable）。反之，如果存在符号串$w \in \Sigma^*$，使得

$$\delta^*(p, w) \in F \text{并且} \delta^*(q, w) \notin F$$

63 或者，反之亦然，那么状态p和q称为依靠符号串w可区分的（distinguishable）。

很明显，两个状态要么可区分，要么不可区分，不可区分具有一个等价关系的属性：如果p和q是不可区分的，并且q和r也是不可区分的，那么p和r就是不可区分的，于是这三个状态都是不可区分的。

化简dfa状态的一个方法是基于寻找和合并不可区分状态的。因此，我们首先描述一种寻找可区分状态对的方法。

程序：mark

1. 去掉所有的不可达状态。通过列举dfa图中所有起点为初态的简单通道，就可以完成这一点。任何不在这种通道上的状态就是不可达状态。

2. 考察所有的状态对（p, q）。如果$p \in F$并且$q \notin F$，或者反之，把状态对（p, q）标记为可区分的。

3. 重复下述过程，直到不再有前面未标记过的状态对。

对于所有状态对（p, q）和所有$a \in \Sigma$，计算$\delta(p, a) = p_a$和$\delta(q, a) = q_a$。如果状态对（p_a, q_a）标记为可区分的，那么把状态对（p, q）也标记为可区分的。

我们称这个程序完成了标记所有可区分状态对的算法。

定理2.3 把程序mark应用到任何dfa $M = (Q, \Sigma, \delta, q_0, F)$上，程序都会停止，并且可以确定这个dfa所有的可区分状态对。

证明：很显然，由于标记的可区分状态对是有穷的，所以程序一定会停止。而且很容易看出，如此标记的状态对一定是可区分的。唯一需要证明的是，这个程序能找到所有的可区分对。

注意，首先状态q_i和q_j可以依靠长度为n的符号串区分，当且仅当对于某个$a \in \Sigma$存在有转移函数

$$\delta(q_i, a) = q_k \tag{2-5}$$

和

$$\delta(q_j, a) = q_l \tag{2-6}$$

其中通过长度为$n-1$的符号串区分两个状态q_k和q_l。

我们首先使用这个来证明完成n次第3步的循环，所有可以依靠长度不超过n的符号串区分的状态都被作以标记。在第2步中，我们标记所有的依靠符号串λ区分的状态对，因此我们就
64 有了$n = 0$作为归纳基础。我们现在假设对于$i = 1, 0, \cdots, n-1$的所有命题都成立。根据这个归纳假设，在第n轮循环开始时，所有可以依靠长度不超过$n-1$的符号串来区分的状态都已经被标记了。因为上面提到的式（2-5）和式（2-6），这轮循环结束后，所有可以依靠长度不超过n的符号串来区分的状态也都被标记了。根据归纳，我们可以认为，对于任何n，完成第n次循环后，所有可以被长度不超过n的符号串区分的状态对都被标记了。

为了证明这个程序能够标记所有可区分的状态，假设循环执行n次后停止。这就意味着在第n次执行过程中，没有新的状态可被标记了。依据式（2-5）和式（2-6），不存在只可以被长度为n的符号串区分的（而不可以被长度少于n的符号串区分的）状态。但是如果不存在只可以被长度为n的符号串区分的状态，那么也就不会存在只可以被长度为$n+1$的符号串区分的

状态了，依此类推。因此，当循环停止的时候，所有可以区分的对都应该被标记了。■

执行这个标记程序之后，我们使用这个结果来把dfa的状态集合Q划分成不相交的子集$\{q_i, q_j, \cdots, q_k\}$，$\{q_l, q_m, \cdots, q_n\}$，…，其中，任何$q \in Q$都只存在于一个子集中，每个子集中的元素都是不可区分的。但是不同子集中的任意两个元素是可以区分的。使用本节后面的习题11描述的结论，可以证明这种划分总是存在的。根据这些子集，我们可以使用下一个程序来构造最小的自动机。

程序：reduce

给定dfa $M = (Q, \Sigma, \delta, q_0, F)$，我们按照下面的方法构造简化的dfa $\hat{M} = (\hat{Q}, \Sigma, \hat{\delta}, \hat{q}_0, \hat{F})$。

1. 使用程序mark来找到所有的可区分状态对。然后根据这个，按照上面提到的方法，把状态集合划分成由所有不可区分状态构成的集合，比如，$\{q_i, q_j, \cdots, q_k\}$，$\{q_l, q_m, \cdots, q_n\}$等。

2. 对于上面由不可区分状态构成的每个集合$\{q_i, q_j, \cdots, q_k\}$，创建一个新的状态，标志为$ij\cdots k$。

3. 对于M中的每条具有下述形式的转移规则

$$\delta(q_r, a) = q_p$$

找到q_r和q_p属于的集合。如果$q_r \in \{q_i, q_j, \cdots, q_k\}$，并且$q_p \in \{q_l, q_m, \cdots, q_n\}$，那么增加一条转移规则 $\hat{\delta}$

$$\hat{\delta}(ij\cdots k, a) = lm\cdots n$$

65

4. 初态 $\hat{q}_0$是 $\hat{M}$ 中标记包含0的状态。

5. $\hat{F}$是所有标记包含i（其中i满足$q_i \in F$）的状态集合。

例2.15 考察图2-18中描绘的自动机。

在步骤2中，程序mark识别出了所有的可区分对（q_0, q_4），（q_1, q_4），（q_2, q_4）和（q_3, q_4）。在执行步骤3的循环过程中，程序计算了$\delta(q_1, 1) = q_4$和$\delta(q_0, 1) = q_3$。

既然（q_3, q_4）是可区分对，那么对（q_0, q_1）也应该被标记。按照这种方式继续，这个标记算法最终会标记（q_0, q_1），（q_0, q_2），（q_0, q_3），（q_0, q_4），（q_1, q_4），（q_2, q_4）和（q_3, q_4）这些对为可区分的，而（q_1, q_2），（q_1, q_3）和（q_2, q_3）是不可区分的。因此，状态q_1, q_2, q_3都是不可区分的。所有的状态分成了三个集合$\{q_0\}$，$\{q_1, q_2, q_3\}$和$\{q_4\}$。应用程序reduce中的步骤2和步骤3，得到了图2-19中的dfa。 □

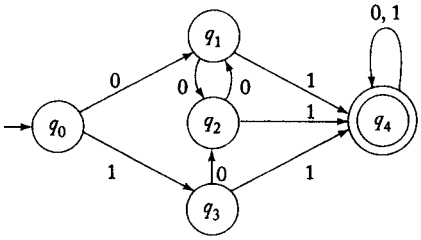


图 2-18

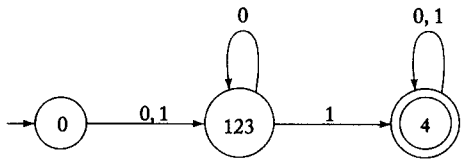


图 2-19

66

定理2.4 给定dfa M，应用程序reduce可以得到另一个dfa $\hat{M}$，满足

$$L(M) = L(\hat{M})$$

而且，$\hat{M}$是最小的。这里的最小指的是它是所有接受$L(M)$的dfa中状态数最少的。

证明：证明包括两部分。第一部分证明使用reduce生成的dfa和原始的dfa等价。这个相对容易

一些。我们可以使用证明dfa与nfa等价中用到的类似的归纳证明。我们需要做的仅仅是证明$\delta^*(q_i, w) = q_j$当且仅当$\hat{\delta}^*(q_i, w)$的标志是$\cdots j \cdots$这样的形式。我们把这个证明留作习题。

第二部分来证明$\hat{M}$是最小的，这个有点难度。假设$\hat{M}$有状态$\{p_0, p_1, p_2, \cdots, p_m\}$，$p_0$是初态。假设存在一个等价的dfa M_1。它的转移函数是δ_1，初态是q_0。它和$\hat{M}$等价，但是状态数更少。因为$\hat{M}$中没有不可达状态，所以一定存在不同的符号串$w_1, w_2, \cdots, w_m$，满足

$$\hat{\delta}^*(p_0, w_i) = p_i, i = 1, 2, \cdots, m$$

但是既然M_1的状态数比$\hat{M}$少，那么一定至少存在两个符号串，假设w_k和w_l，满足

$$\delta_1^*(q_0, w_k) = \delta_1^*(q_0, w_l)$$

因为p_k和p_l是可区分的，所以，一定存在某个符号串x，使得$\hat{\delta}^*(p_0, w_k x) = \hat{\delta}^*(p_k, x)$是终态，并且$\hat{\delta}^*(q_0, w_l x) = \hat{\delta}^*(p_l, x)$是非终态（或恰好相反）。换句话说，$w_k x$可以被$\hat{M}$接受，而$w_l x$则不能被$\hat{M}$接受。但是注意

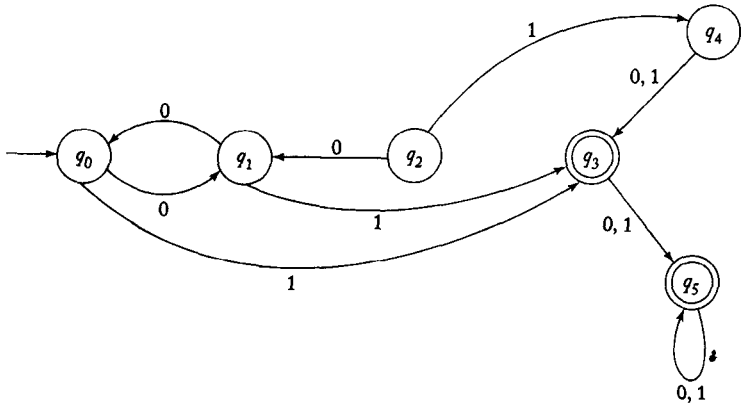
$$\begin{aligned}\delta_1^*(q_0, w_k x) &= \delta_1^*(\delta_1^*(q_0, w_k), x) \\ &= \delta_1^*(\delta_1^*(q_0, w_l), x) \\ &= \delta_1^*(q_0, w_l x)\end{aligned}$$

因此，M_1要么接受$w_k x$和$w_l x$，要么都不接受。这与$\hat{M}$和M_1等价的假设矛盾。这个矛盾证明了M_1不存在。■

67

习题

1. 最小化图2-16中dfa的状态数。
2. 寻找接受下面语言的最小dfa，分别证明它们的结果是最小的:
   (a) $L = \{a^n b^m : n \geq 2, m \geq 1\}$
   (b) $L = \{a^n b : n \geq 0\} \cup \{b^n a : n \geq 1\}$
   (c) $L = \{a^n : n \geq 0, n \neq 3\}$ ●
   (d) $L = \{a^n : n \neq 2, n \neq 4\}$
3. 证明使用程序reduce生成的自动机是确定的。
4. 最小化下图中描绘的dfa的状态数。●



5. 证明：如果L是非空语言，并且L中的任何符号串w的长度至少为n，那么，接受L的dfa至少有$n+1$个状态。
6. 证明下面的猜想成立，或不成立。如果$M=(Q,\Sigma,\delta,q_0,F)$是接受正则语言$L$的最小dfa，那么$\hat{M}=(Q,\Sigma,\delta,q_0,Q-F)$是接受语言$\overline{L}$的最小dfa。●
7. 证明：不可区分性是等价关系，而可区分性不是。
8. 给出定理2.4第一部分的详细证明。即：证明$\hat{M}$和原始dfa等价。
★★9. 对于任意给定的dfa，给出产生与之等价的最小dfa的程序。
10. 证明下面命题：如果状态q_a和q_b是不可区分的，并且q_a和q_c是可区分的，那么，q_b和q_c一定是可区分的。● 68
11. 完成程序mark之后，进行下面过程。从某个状态开始，比如说q_0。把所有没有标记为与q_0可区分的状态放到q_0的等价集合中。然后，再取出另一个状态（不属于前面已经划定的等价集合），重复同样的操作。反复进行上述过程，直到处理完所有的状态。接着，把上述过程形式化，写成一个算法。证明这个算法确实能够把原始的状态集合划分成若干个等价的状态集合。 69

第3章 正则语言与正则文法

根据我们的定义，如果一种语言能够被一个有穷接受器接受，那么这种语言就是正则语言。因此，每种正则语言都可以被某个dfa或nfa描述。这种描述很有用，比如，我们想通过逻辑证明一个给定的符号串是否属于某种语言。但是很多时候，我们需要更加简明的方法来描述正则语言。在本章中，我们介绍描述正则语言的其他方式。这些表示方法有着重要的实际应用，我们将通过一些例子和习题来了解这些应用。

3.1 正则表达式

一种描述正则语言的方式是使用*正则表达式*（regular expression）的表示方法。这种表示方法包含了字母表Σ上的符号构成的符号串、圆括号及操作符+，·和*的组合。最简单的情况是语言$\{a\}$，这个语言用正则表达式a来表示。稍微复杂点的语言$\{a, b, c\}$，是用+来表示并集，它的正则表达式是$a + b + c$。按照类似的方式，我们使用·表示连接，*表示星闭包。表达式$(a + b \cdot c)^*$表示$\{a\} \cup \{bc\}$的星闭包。即：它代表的语言是$\{\lambda, a, bc, aa, abc, bca, bcbc, aaa, aabc, \cdots\}$。 71

3.1.1 正则表达式的形式化定义

我们对基本构成成分重复地使用某种递归规则来构造正则表达式，这和我们构造算术表达式的方式是类似的。

定义3.1 *设Σ是给定的字母表，那么*

1. *$\emptyset$，λ和$a \in \Sigma$都是正则表达式。这些叫做基本正则表达式（primitive regular expression）。*
2. *如果r_1和r_2是正则表达式，那么$r_1 + r_2$，$r_1 \cdot r_2$，r_1^*和（r_1）也是正则表达式。*
3. *当且仅当把规则2有限次地应用于基本正则表达式得到的符号串，才是正则表达式。*

例3.1 已知$\Sigma = \{a, b, c\}$，符号串

$$(a + b \cdot c)^* \cdot (c + \emptyset)$$

是正则表达式，因为它是按照上面的规则构造的。例如，如果我们取$r_1 = c$，$r_2 = \emptyset$，我们会发现$c + \emptyset$和（$c + \emptyset$），也都是正则表达式。重复这个过程，我们最终产生了所有的符号串。另外，$(a + b +)$不是正则表达式，因为它不能用基本的正则表达式来构造。 □ 72

3.1.2 和正则表达式相关的语言

正则表达式可以用来描述一些简单语言。如果r是正则表达式，我们就用$L(r)$表示和r相关的语言。这个语言的定义如下：

定义3.2 *使用正则表达式r表示的语言$L(r)$，由下面规则定义：*

1. *$\emptyset$是表示空集的正则表达式。*
2. *λ是表示$\{\lambda\}$的正则表达式。*

3. 对于每个$a \in \Sigma$，a是表示$\{a\}$的正则表达式。

如果r_1和r_2是正则表达式，那么

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$。

5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$。

6. $L((r_1)) = L(r_1)$。

7. $L(r_1^*) = (L(r_1))^*$。

定义的后4条规则递归地把$L(r)$ 转化成更简单的成分。前三条是这个递归的终止条件。为了弄清楚给定的表达式表示的是什么语言，我们反复使用上面这些规则。

例3.2 用集合形式表示语言$L(a^* \cdot (a + b))$。

$$\begin{aligned} L(a^* \cdot (a+b)) &= L(a^*)L(a+b) \\ &= (L(a))^*(L(a) \cup L(b)) \\ &= \{\lambda, a, aa, aaa, \cdots\}\{a, b\} \\ &= \{a, aa, aaa, \cdots, b, ab, aab, \cdots\} \end{aligned}$$

73 □

定义3.2中的规则4至规则7存在一个问题。如果给定r_1和r_2，它们可以准确地定义语言，但是把一个复杂的表达式拆分成几部分，拆分可以按照很多种方式进行，因此会造成拆分的歧义性。例如，正则表达式$a \cdot b + c$。我们可以把它看成是由$r_1 = a \cdot b$和$r_2 = c$组成的。这样的话，我们就得到$L(a \cdot b + c) = \{ab, c\}$。但是根据定义3.2，我们也可以拆分成$r_1 = a$和$r_2 = b + c$。这样就得到了一个不同的结果，$L(a \cdot b + c) = \{ab, ac\}$。为了避免这种情况的发生，我们就需要将所有的表达式都用圆括号括起来，但是这样得到的结果又过于麻烦了。于是，我们使用数学和程序设计语言中类似的规定，建立一套优先级规则，从而确定星闭包要优先于连接，连接又优先于并。另外，连接的符号可以省略，因此我们使用r_1r_2代替$r_1 \cdot r_2$。

稍做练习之后，我们很快就可以清楚一个特殊的正则表达式表示的是什么样的语言。

例3.3 已知$\Sigma = \{a, b\}$，表达式

$$r = (a + b)^*(a + bb)$$

是正则的。它表示语言

$$L(r) = \{a, bb, aa, abb, ba, bbb, \cdots\}$$

分别考察r的各部分，我们就可以得出上面的结论。第一部分是 $(a + b)^*$，代表a和b构成的任意符号串。第二部分是 $(a + bb)$，表示一个a，或者两个b。因此，$L(r)$ 表示$\{a, b\}$上的所有以a或bb结尾的符号串。 □

例3.4 表达式

$$r = (aa)^*(bb)^*b$$

表示偶数个a，后面跟着奇数个b的所有符号串的集合。即:

$$L(r) = \{a^{2n}b^{2m+1} : n \geqslant 0, m \geqslant 0\}$$

74 根据非形式化的描述或集合表示来确定其正则表达式会更困难。 □

例3.5 已知$\Sigma = \{0, 1\}$，给出一个满足下面描述的正则表达式r

$$L(r) = \{w \in \Sigma^* : w\text{至少有一对连续的}0\}$$

根据下面的推理可以得到结果: $L(r)$ 中的每个符号串一定包含00，但是00之前与之后的符号

完全是任意的。{0, 1}上的任意符号串可以表示为 $(0+1)^*$。把这些分析放在一起，我们得到结果为

$$r=(0+1)^*00(0+1)^* \qquad \square$$

例3.6 为下面的语言构造正则表达式

$$L=\{w\in\{0,1\}^*: w\text{中不存在连续的}0\}$$

尽管这个看起来和例3.5类似，但是构造起来要更困难些。根据观察得到一个有帮助的结论是：0的后面一定紧跟着1。这个子串的前面和后面还可以有任意个1。这就暗示我们结果应该包含1…101…1形式的重复的符号串，即：这种语言应该用正则表达式 $(1^*011^*)^*$ 来表示。然而，结果还不完整，因为我们还没有考虑以0结束的符号串或全部由1构成的符号串这两种情况。把上面的这些特殊情况考虑在内，我们得到结果为

$$r=(1^*011^*)^*(0+\lambda)+1^*(0+\lambda)$$

如果我们将推理稍做变化，就可以得到另一个结果。如果我们把L看成是符号串1和01的重复，那么可以得到比较短的正则表达式

$$r=(1+01)^*(0+\lambda)$$

尽管两个结果看起来不同，但它们都是正确的，因为它们表示的是同一种语言。通常，任意给定一种语言都可以由无限种正则表达式来表达。

注意这个语言是例3.5中的语言的补。然而，这两个正则表达式并不非常相似，不能轻易看出这两种语言之间的紧密联系。 □

最后的例子中引入了正则表达式的等价表示。如果两个正则表达式表示的是同一种语言，那么我们就称它们是等价的。我们可以推导出大量简化正则表达式的规则（详见本节的习题18），但是因为我们很少需要使用这种处理，所以我们不再谈这方面的问题。 75

习题

1. 在$L((a+b)^*b(a+ab)^*)$ 中找到长度少于4的所有符号串。
2. 表达式 $((0+1)(0+1)^*)^*00(0+1)^*$表示的是例3.5中的语言吗？
3. 证明$r=(1+01)^*(0+1^*)$ 也可以表示例3.6中的语言。找到另外两个等价的表达式。
4. 为集合$\{a^nb^m:(n+m)$ 是偶数$\}$构造一个正则表达式。
5. 为下列语言构造正则表达式：
   (a) $L_1=\{a^nb^m, n\geqslant 4, m\leqslant 3\}$
   (b) $L_2=\{a^nb^m, n<4, m\leqslant 3\}$
   (c) L_1的补集
   (d) L_2的补集
6. 表达式 $(\varnothing^*)^*$和$a\varnothing$分别表示的是什么语言？
7. 对语言$L((aa)^*b(aa)^*+a(aa)^*ba(aa)^*)$，给出一个简单的口头描述。
8. 已知L是习题1中的语言，给出L^R的正则表达式。
9. 给出$L=\{a^nb^m: n\geqslant 1, m\geqslant 1, nm\geqslant 3\}$的正则表达式。
10. 给出$L=\{ab^nw: n\geqslant 3, w\in\{a,b\}^+\}$的正则表达式。

11. 为例3.4中的语言的补集构造正则表达式。
12. 为$L = \{vwv : v, w \in \{a, b\}^*, |v| = 2\}$构造正则表达式。●
13. 为$L = \{w \in \{0, 1\}^* : w$有且只有一对连续的0$\}$构造正则表达式。
14. 为下面定义在$\Sigma = \{a, b, c\}$上的语言构造正则表达式:
    (a) 有且只有一个a的所有符号串。
    (b) 仅有3个a的所有符号串。
    (c) 包含Σ上所有字母的符号串。●
    (d) a的长度不超过2的所有符号串。
    ★(e) a的长度为3的倍数的所有符号串。

76

15. 分别写出下列定义在$\{0, 1\}$上的语言的正则表达式:
    (a) 结束符为01的所有符号串。
    (b) 结束符不是01的所有符号串。
    (c) 包含偶数个0的所有符号串。●
    (d) 至少包含两个00子串的所有符号串（注意，通常认为000包含两个这样的子串）。
    (e) 子串00至多出现两次的所有符号串。
    ★(f) 不包含子串101的所有符号串。
16. 为下列定义在$\{a, b\}$上的语言构造正则表达式:
    (a) $L = \{w : |w| \bmod 3 = 0\}$ ●
    (b) $L = \{w : n_a(w) \bmod 3 = 0\}$
    (c) $L = \{w : n_a(w) \bmod 5 > 0\}$
17. 把习题16中的字母表变成$\Sigma = \{a, b, c\}$，分别构造能够接受上面（a）、（b）和（c）的正则表达式。
18. 已知任意正则表达式r_1和r_2，确定下列结论是否成立。符号≡表示正则表达式等价，即：两种正则表达式表示的是同一种语言。
    (a) $(r_1^*)^* \equiv r_1^*$
    (b) $r_1^*(r_1 + r_2)^* \equiv (r_1 + r_2)^*$
    (c) $(r_1 + r_2)^* \equiv (r_1^* r_2^*)^*$ ●
    (d) $(r_1 r_2)^* \equiv r_1^* r_2^*$
19. 给出一个通用的方法，能够把正则表达式r变成$\hat{r}$，满足$(L(r))^R = L(\hat{r})$。
20. 严格证明例3.6中的表达式确实表示的是指定的语言。
21. 已知正则表达式r不包含λ或$\emptyset$，如果$L(r)$是无穷的，给出r必须满足的充分必要条件。●
22. 形式语言可以用来描绘二维图形。循环码语言是定义在字母表$\Sigma = \{u, d, r, l\}$上的，其中，字母表中的这些符号分别表示方向为上、下、右和左的单位长度的直线。使用这个表示的一个例子$urdl$，表示的是边为单位长度的正方形。画出表达式$(rd)^*$，$(urddru)^*$和$(ruldr)^*$表示的图形。
23. 根据习题22中的定义，表达式要满足什么样的充分条件，才能使得画出的图形是闭合的（即起点和终点是相同的）？这些条件是必需的吗？●

77

24. 构造接受语言$L(aa^*(a + b))$的nfa。
25. 构造正则表达式表示所有的位串：当把这个位串解释成二进制数时，它的值大于或等于40。●

26. 构造正则表达式表示所有的位串，其开始位为1，并且把这个位串解释成二进制数时，它的值不在10和30之间。

3.2 正则表达式和正则语言之间的联系

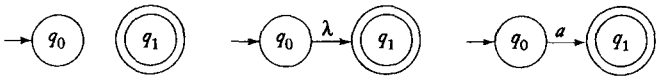
从术语字面上看来，正则表达式和正则语言之间的联系应该很紧密。这两个概念从本质上说是相同的。对于每一个正则语言，都可以找到一个正则表达式来表示它。而每一种正则表达式表示也都对应着一种正则语言。我们将分别证明这两部分。

3.2.1 正则表达式表示正则语言

我们首先证明如果r是正则表达式，那么$L(r)$ 则是正则语言。根据我们的定义有，如果一个语言可以被某个dfa接受，那么它就是正则语言。由于nfa和dfa等价，所以，如果一个语言可以被某个nfa接受，那么它也是正则语言。我们首先证明：已知任意正则表达式r，我们都能构造一个接受$L(r)$ 的nfa。构造的方法依赖于$L(r)$ 的递归定义。我们首先构造简单的自动机满足定义3.2的（1）、（2）和（3）这3部分。然后证明使用这几个简单的自动机，我们可以构造更加复杂的自动机满足第（4）、第（5）和第（7）部分。

定理3.1 *设r是正则表达式。那么，一定存在某个非确定型有穷接受器接受$L(r)$。因此，$L(r)$ 是正则语言。*

证明：我们首先构造接受简单正则表达式$\varnothing$，λ和$a\in\Sigma$表示的语言的自动机。它们分别如图3-1的a)、b)、c)所示。假设我们分别用自动机$M(r_1)$ 和$M(r_2)$ 接受正则表达式r_1和r_2表示的语言。我们不必直接构造这些自动机，可以示意性地表示它们，如图3-2所示。在这个示意图中，图中左边的顶点表示初态，右边的顶点表示终态。在2.3节习题7中，我们证明了对于任何nfa，总存在一个与之等价的单终态nfa。因此，我们这里可以假设只有一个终态。我们用这种方式表示$M(r_1)$和$M(r_2)$，接下来，为正则表达式$r_1 + r_2$，r_1r_2和r_1^*构造自动机。构造过程如图3-3至图3-5所示。正如图中指出的，作为构成成分的自动机的初态和终态分别被新的初态和终态代替。按照上面的这几个步骤，我们可以为任意复杂的正则表达式构造自动机。 78



a) 接受$\varnothing$的nfa　　b) 接受$\{\lambda\}$的nfa　　c) 接受$\{a\}$的nfa

图 3-1

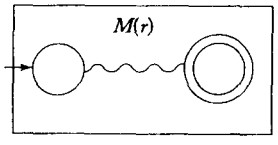


图3-2 nfa接受$L(r)$ 的示意图

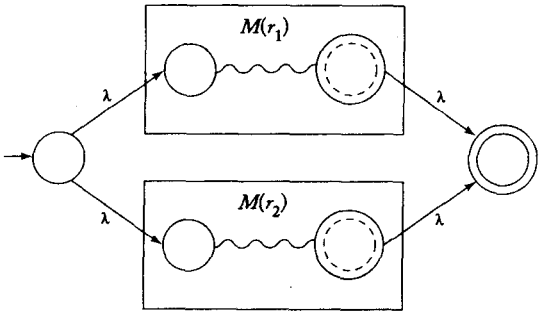


图3-3 接受$L(r_1 + r_2)$ 的自动机

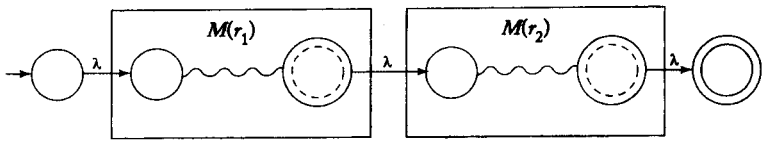


图3-4 接受$L(r_1r_2)$的自动机

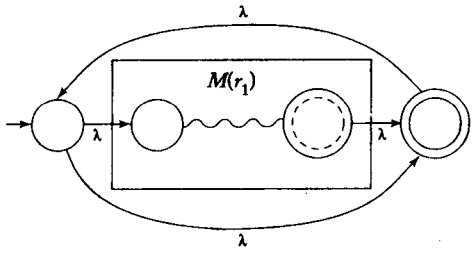


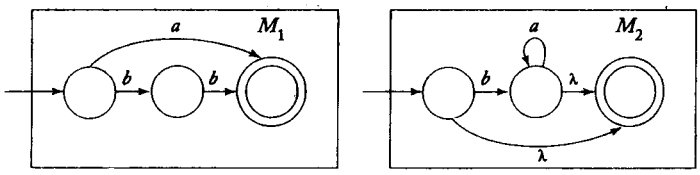
图3-5 接受$L(r_1^*)$的自动机

通过图3-3至图3-5的解释，我们可以更加清楚这个构造过程。为了论证的严密性，我们给出由构成成分的状态和转移函数来构造合并的自动机的状态和转移函数的形式化方法。然后通过对构造过程中运算的数目作归纳，来证明这样可以构造一个接受用某种正则表达式表示的语言的自动机。对这点我们不再详细讨论，因为这个结论显然总是正确的。■

例3.7 构造一个nfa接受$L(r)$，其中

$$r = (a + bb)^*(ba^* + \lambda)$$

$(a + bb)$ 和 $(ba^* + \lambda)$ 的自动机，直接按照第一个原则构造，具体如图3-6所示。使用定理3.1把这些组合在一起，就得到图3-7中的结果。 □



a) M_1接受$L(a + bb)$　　b) M_2接受$L(ba^* + \lambda)$

图 3-6

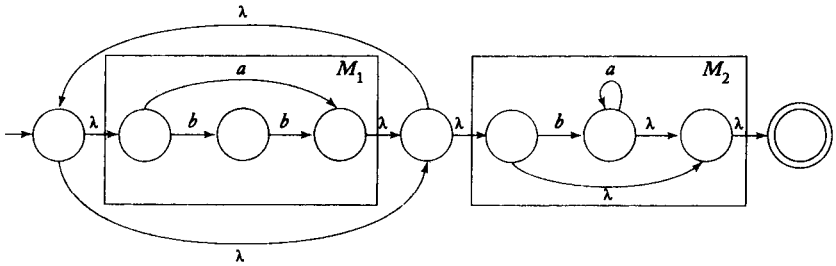


图3-7 接受$L((a + bb)^*(ba^* + \lambda))$的自动机

79 ~ 80

3.2.2 正则语言的正则表达式

从直观上看，定理3.1的逆命题也应该成立。即：对于任何一个正则语言，都存在与它相

应的正则表达式。因为任何正则语言都存在着与之相关的nfa，因此也存在着与之对应的转移图。我们需要做的就是找到一个正则表达式，使得它能够生成从q_0到任意终态的所有通道上的标记。这点看起来并不困难，但是由于回路的存在可能会把问题变复杂，因为回路可以按照任何顺序重复经过任意次。这就产生了一个必须仔细处理的簿记问题。解决这个问题的方法很多，一个比较直观的方法是利用所谓的*通用转移图*（generalized transition graph）。因为这种思想在这里是按照某种受限制的方式使用的，对于将来的讨论没起到什么作用，所以，我们在这里只是非形式化地介绍一下。

通用转移图是边标有正则表达式的转移图。它的其他方面和一般的转移图是一样的。从初态到终态的通道上的标记形成了几个正则表达式的连接。因此，它本身就是一个正则表达式。被上述的正则表达式表示的符号串，就是能够被通用转移图接受的语言的子集。所有这样的子集的并，就是该通用转移图接受的语言的全集。

例3.8 图3-8表示一个通用转移图。考察这个图可以清晰地得到它接受的语言是$L(a^* + a^*(a + b)c^*)$。标有a的边（q_0, q_0）是一个回路，这个回路能够产生任意个数的a，即：它可以生成$L(a^*)$。我们可以改用a^*标记这条边，而且不改变图接受的语言。 □

非确定型有穷接受器的图都可以看成是通用转移图，如果边的标记都可以正确地解释的话。一个标有单符号a的边可以解释成标有表达式a的边，而标有多个符号$a, b, \cdots$的边可以解释成标有表达式$a + b + \cdots$的边。根据这个观察结果，我们可以得出结论：对于任何正则语言，总存在一个通用转移图接受它。反之，每个被通用转移图接受的语言都是正则的。因为通用转移图中通道上的标记都是正则表达式，因此可以应用定理3.1立即得到这个结论。无论如何，证明中存在一些精巧之处，我们就不在这里进一步的介绍了，把它的详细证明留给读者作为习题，见4.3节习题16。

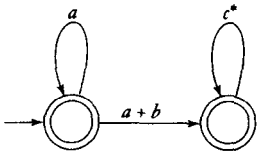


图 3-8

81

通用转移图的等价性可以通过它们接受的语言来定义。考虑一个具有状态$\{q, q_i, q_j, \cdots\}$的通用转移图，其中，q既不是终态也不是初态。我们想要构造一个去掉状态q的通用转移图，使它与原通用转移图等价，如果我们不改变从q_0到q_f的过程中所产生的标记集合表示的语言，那么我们可以做到这点。这个构造过程可以参照图3-9中的解释。其中，在图3-9中，要去掉状态q，边标记$a, b, \cdots$表示一般的表达式。由于q有到三个顶点q_i, q_j, q的输出边，在这个意义上，该情况描述的是一种最通用的情况。若要去掉图3-9a中的边，我们就要去掉其在图3-9b中对应的边。

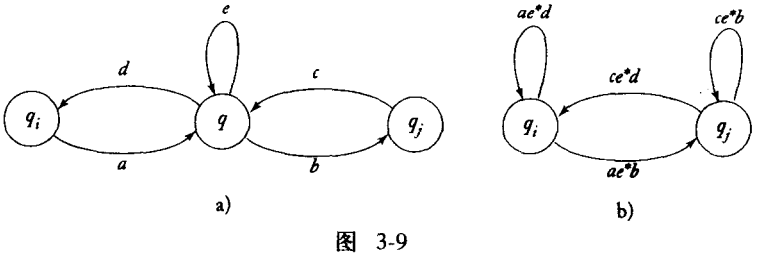


图 3-9

图3-9的构造过程证明了需要引入一些边，从而保证当我们去掉q及其所有的输入、输出边时，通用转移图对应的语言不会改变。完整的处理过程需要我们在去掉q之前，对于$Q - \{q\}$中的所有对（q_i, q_j），都要完成上述过程。尽管我们不会形式化地证明这点，但是这点可以通

过构造一个等价的通用转移图来证明。接受这点之后，我们准备开始证明如何把任意一个nfa和一个正则表达式关联起来。

定理3.2 *设L是正则语言。那么总存在正则表达式r，使得$L = L(r)$。*

证明：设M是接受L的nfa。我们可以不损失任何通用性地作出假设：M只有一个终态，并且$q_0 \notin F$。我们把M的图解释成通用转移图，并应用上述构造方法。为了去掉标有q的顶点，我们使用图3-9中的方案处理所有的对（q_i, q_j）。加入所有的新边后，去掉q以及所有与之相关的边。我们继续这个过程，一个顶点接着一个顶点地移除，直到变成图3-10中的情形。被上述图接受的语言对应的正则表达式是

82

$$r = r_1^* r_2 (r_4 + r_3 r_1^* r_2)^* \tag{3-1}$$

因为所有通用转移图的推导序列都等价于初始的那一个。所以我们对通用转移图中的状态数作归纳，从而证明式（3-1）中的正则表达式表示的是语言L。■

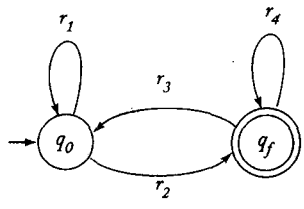


图 3-10

例3.9 考虑图3-11a中的nfa。除去状态q_1，得到的相应通用转移图如图3-11b所示。识别出它的组成部分分别为$r_1 = b + ab^*a$，$r_2 = ab^*b$，$r_3 = \varnothing$，$r_4 = a + b$，我们最终得到这个原始自动机对应的正则表达式为

$$r = (b + ab^*a)^* ab^* b(a + b)^*$$

定理3.2的构造过程冗长，而且给出的解比较长，但这是一个有固定步骤的方法，而且它总是有效的。□

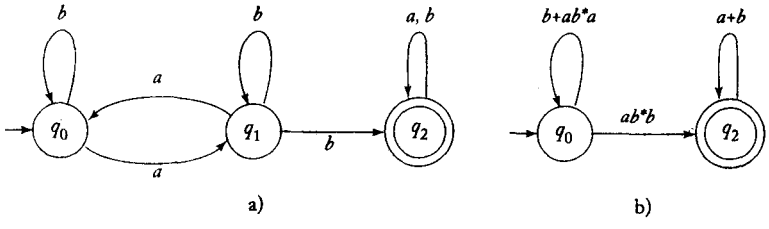


图 3-11

83

例3.10 为下面语言构造正则表达式

$$L = \{w \in \{a, b\}^* : n_a(w)\text{是偶数，并且}n_b(w)\text{是奇数}\}$$

直接根据描述去构造正则表达式会遇到各种困难。另一方面，只要我们有效地利用顶点标记，我们就能够容易地构造出接受它的nfa。我们用标记为EE的顶点来表示a和b的个数都是偶数个。用标记为OE的顶点表示的a个数是奇数，b的个数是偶数，等等。使用这种方法，我们可以很容易得到相应的nfa解，如图3-12所示。

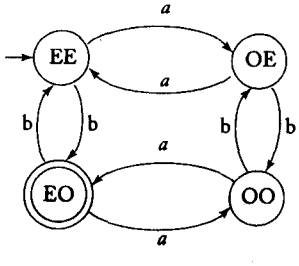


图 3-12

我们现在使用机械的方式把它转化成正则表达式。首先，我们去掉标记为OE的状态，得到的通用转移图见图3-13。

然后，我们去掉标有OO的顶点，得到的结果是图3-14。最后，我们应用式（3.1）得到

$$r_1 = aa + ab(bb)^*ba$$
$$r_2 = b + ab(bb)^*a$$
$$r_3 = b + a(bb)^*ba$$
$$r_4 = a(bb)^*a$$

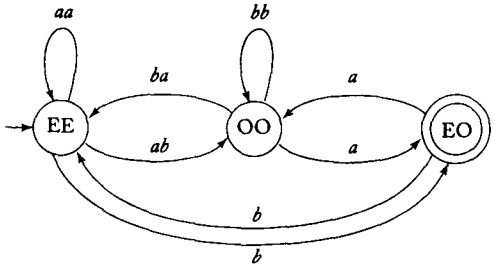


图 3-13

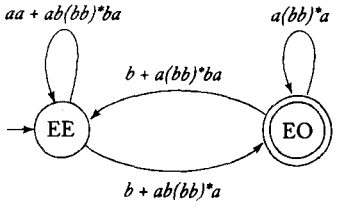


图 3-14

最终的表达式又长又复杂，但是，得到它的方式确是相对直接的。 □

3.2.3 描述简单模式的正则表达式

在例1.15和2.1节的习题15中，我们探讨了有穷接受器和程序设计语言的简单组成部分，如标识符、整数和实数之间的关系。有穷自动机和正则表达式之间的关系意味着我们可以把正则表达式看成是描述这些特征的一种方式。这点很容易看出来，比如，所有可以被接受的Pascal整数的集合用正则表达式描述，可以表示成sdd^*。这里，s表示符号，可能的取值范围为$\{+, -, \lambda\}$，d表示从0到9的数字。

Pascal整数是所谓的“模式”的一个简单例子。模式指的是具有共同属性的对象集合。模式匹配指的是把一个给定的对象赋值给某几个分类中的一个。成功模式匹配的关键常常在于找到一个有效的描述模式的方法。这是计算机科学中一个复杂并且广阔的领域，我们在这里只能简单间接地涉及而已。下面是一个简化但有启发意义的例子，它演示了我们到目前为止探讨过的想法是如何在模式匹配领域发挥作用的。

例3.11 模式匹配可以应用于文本编辑。所有的文本编辑器都允许在文本中搜索某个给定的符号串。大多数编辑器还扩展了这项功能，使得它能够搜索模式。例如，UNIX操作系统中的编辑器*ed*可以识别命令

$$/aba^*c/$$

84 ~ 85

这条命令是要在文档中搜索满足下面条件的符号串：符号串ab后面跟着任意数目的a，之后跟着一个c。从这个例子，我们可以看出UNIX编辑器能够识别正则表达式（尽管它有时候使用的识别正则表达式的方式和这里使用的规则有所不同）。

这个领域具有挑战性的任务就是写一个有效的程序来识别符号串模式。在文档中寻找一个指定的符号串是个非常简单的程序设计练习，但是这里的情况更加复杂。我们必须处理无限长的任意复杂模式，而且模式不是事先固定的，是在运行时产生的。模式描述是输入的一部分，因此，识别过程必须要灵活。为了解决这个问题，经常需要使用自动机理论。

如果模式可以被正则表达式识别出来，那么模式识别程序就可以采用这种描述方式，

使用定理3.1中的构造方法把它转化成等价的nfa。然后可以使用定理2.2来把它还原成dfa。将这个dfa写成一个转移表格的形式，就成为一个有效的模式匹配算法。程序员需要做的仅仅是使用这个表格给出一个通用的框架。这样，我们可以自动处理在运行时刻定义的大量模式。

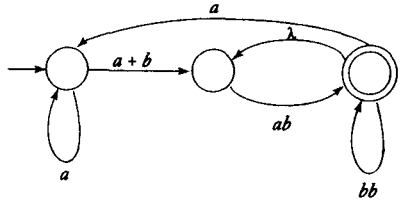
还要考虑程序的效率。使用定理2.1和定理3.1，由正则表达式构造有穷自动机的过程会产生具有很多状态的自动机。如果内存空间有问题的话，就使用2.4节中减少状态的方法。 □

习题

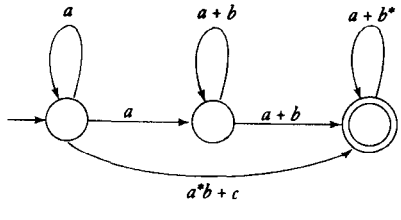
1. 使用定理3.1的构造方法，构造接受语言$L(ab^*aa + bba^*ab)$ 的nfa。
2. 构造一个nfa，接受习题1中的语言的补集。
3. 设计一个接受语言$L((a + b)^*b(a + bb)^*)$ 的nfa。●
4. 分别构造接受下列语言的dfa:
   (a) $L(aa^* + aba^*b^*)$ ●
   (b) $L(ab(a + ab)^*(a + aa))$
   (c) $L((abab)^* + (aaa^* + b)^*)$
   (d) $L(((aa^*)^*b)^*)$

86

5. 分别构造接受下列语言的dfa:
   (a) $L = L(ab^*a^*)\cup L((ab)^*ba)$
   (b) $L = L(ab^*a^*)\cap L((ab)^*ba)$
6. 为3.1节习题15(f) 构造nfa。使用这个来生成该语言的正则表达式。
7. 给出图3-9构造过程中，当图3-9a中的边逐条去掉时，使用的构造规则。●
8. 考虑下面的通用转移图

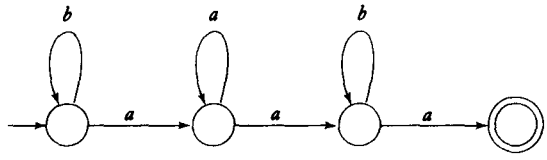


   (a) 给出一个与之等价的只有两个状态的通用转移图。●
   (b) 这个图接受的语言是什么？●
9. 下面的通用转移图接受的是什么语言？

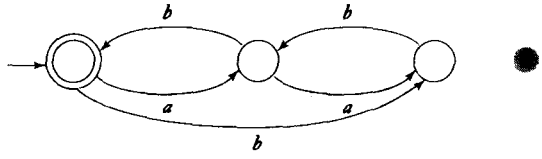


87

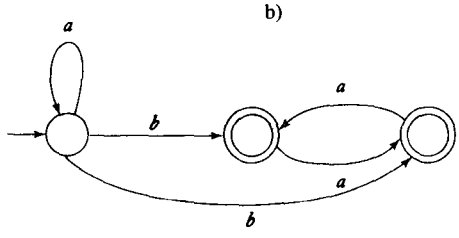
10. 分别给出下列自动机接受的语言的正则表达式:



a)



b)



c)

11. 重做例3.10，不过这次我们首先去掉状态OO。
12. 为下面定义在$\{a, b\}$上的语言构造正则表达式:
    (a) $L = \{w : n_a(w)$ 和$n_b(w)$ 都是偶数$\}$
    (b) $L = \{w : (n_a(w) - n_b(w))\bmod 3 = 1\}$
    (c) $L = \{w : (n_a(w) - n_b(w))\bmod 3 \neq 0\}$
    (d) $L = \{w : 2n_a(w) + 3n_b(w)$ 是偶数$\}$
13. 给出能够生成2.1节习题23中定义二进制加法的三元符号串的正则表达式。
14. 证明图3-9的构造方法可以生成等价的通用转移图。
15. 给出描述Pascal实数集合的正则表达式。
16. 给出描述Pascal整数集合的正则表达式。
17. 很多应用中，如：程序拼写检查，我们不需要进行模式的精确匹配，只需要模糊匹配。一旦模糊匹配变得精确了，自动机理论就可以应用到构造模糊模式匹配机中。比如，在原有的模式匹配中插入一个符号，从而获得的模式。设L是定义在Σ上的正则语言，定义 88

$$insert(L) = \{uav : a \in \Sigma, uv \in L\}$$

效果上，对于L中的每一个单词，在单词的任意位置插入一个伪字母，$insert(L)$ 就包含这样生成的所有单词。

★(a) 对给定的一个接受L的nfa，说明如何为$insert(L)$ 构造nfa。

★★(b) 讨论如何使用这个来写一个$insert(L)$ 的模式识别程序，把L的正则表达式作为输入。

★18. 与前面的习题类似，去掉符号串中的一个符号，考虑所有按照这样的办法由L生成的单词。给出语言的去掉操作$drop$的形式化定义。给定接受L的nfa，构造接受$drop(L)$ 的nfa。
19. 使用定理3.1的构造方法，构造接受$L(a\emptyset)$ 和$L(\emptyset^*)$ 的nfa。这个结果和这些语言的定义一致吗？

3.3 正则文法

描述正则语言的第三种方法是使用某种简单的文法。文法经常用来表示某个特定语言。每当我们用自动机或其他方式定义一个语言族的时候，我们都会有兴趣了解与语言族相关联的文法是什么样的。首先，我们看看能够产生正则语言的文法。

3.3.1 右线性文法和左线性文法

定义3.3 文法$G=(V,T,S,P)$称为右线性（right-linear）文法，如果所有的产生式都满足下面的形式

$$A\to xB$$
$$A\to x$$

其中，$A,B\in V$，并且$x\in T^*$。所有产生式满足如下两种形式的文法称为左线性（left-linear）文法

89

$$A\to Bx$$

或

$$A\to x$$

右线性文法和左线性文法都是正则文法（regular grammar）。

注意在正则文法中，任何产生式的右侧至多有一个变量。而且，这个变量要么在产生式的最右端，要么在最左端。

例3.12 文法$G_1=(\{S\},\{a,b\},S,P_1)$，按照右线性定义$P_1$如下

$$S\to abS|a$$

文法$G_2=(\{S,S_1,S_2\},\{a,b\},S,P_2)$，产生式是左线性的，定义为

$$S\to S_1ab$$
$$S_1\to S_1ab|S_2$$
$$S_2\to a$$

G_1和G_2都是正则文法。

G_1的推导得出的推导序列

$$S\Rightarrow abS\Rightarrow ababS\Rightarrow ababa$$

根据这个简单例子，容易推测出$L(G_1)$是正则表达式$r=(ab)^*a$表示的语言。类似地，我们可以看出$L(G_2)$就是正则语言$L(aab(ab)^*)$。 □

例3.13 文法$G=(\{S,A,B\},\{a,b\},S,P)$，产生式

$$S\to A$$
$$A\to aB|\lambda$$
$$B\to Ab$$

90 不是正则的。尽管每个产生式要么是右线性形式，要么是左线性形式，但是文法本身就不是右线性的，也不是左线性的。因此，这个文法不是正则的。这个文法是线性文法（linear

grammar)。线性文法指的是在产生式的右侧至多有一个变量的文法，而且不限制这个变量的位置。很明显，正则文法总是线性的，但是并不是所有的线性文法都是正则的。 □

我们的下一个目标是证明正则文法和正则语言是相关的，对于每种正则语言，总存在一个表示它的正则文法。因此，正则文法是正则语言的另一种表示方法。

3.3.2 右线性文法生成正则语言

首先，我们证明右线性文法产生的语言总是正则的。为了证明这一点，我们要构造一个nfa，让它来模仿右线性文法的推导过程。注意右线性文法的句型，仅有一个变量，并且这个变量出现在最右端。假设我们使用产生式$D \to dE$得到一个推导

$$ab\cdots cD \Rightarrow ab\cdots cdE$$

相应的nfa可以模仿这个步骤。遇到符号d，nfa从状态D移动到状态E。按照这个方法，自动机中的状态就对应着句型中的变量，而符号串的前缀保持不变。上述的这个简单想法是下面定理的基础。

定理3.3 设$G = (V, T, S, P)$是右线性文法。那么，$L(G)$是正则语言。

证明：假设$V = \{V_0, V_1, \cdots\}$，并且$S = V_0$，产生式的形式为$V_0 \to v_1V_i$, $V_i \to v_2V_j$, $\cdots$或$V_n \to v_l$, $\cdots$。如果w是$L(G)$中的一个符号串，那么因为G中的产生式的形式，推导一定是按照下面的形式进行

$$\begin{aligned} V_0 &\Rightarrow v_1V_i \\ &\Rightarrow v_1v_2V_j \\ &\overset{*}{\Rightarrow} v_1v_2\cdots v_kV_n \\ &\Rightarrow v_1v_2\cdots v_kv_l = w \end{aligned} \tag{3-2}$$

需要构造的自动机将通过顺序地使用这些v来再现这个推导。自动机的初态将标为V_0。对于每个变量V_i，都将对应地存在一个标有V_i的非终态。对于每一个产生式 91

$$V_i \to a_1a_2\cdots a_mV_j$$

自动机中都存在一个连接V_i和V_j的转移。即：δ将被定义使得

$$\delta^*(V_i, a_1a_2\cdots a_m) = V_j$$

对于每一个产生式

$$V_i \to a_1a_2\cdots a_m$$

自动机中相应的转换就是

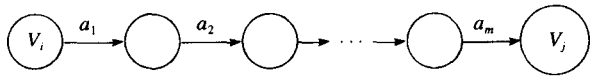
$$\delta^*(V_i, a_1a_2\cdots a_m) = V_f$$

其中，V_f是终态。需要处理的中间状态无关紧要，可以任意标记。通用的方法如图3-15所示。把这些单独的部分组装起来就构成了完整的自动机。

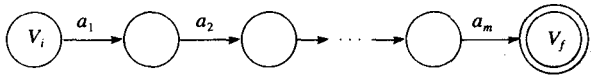
现在假设$w \in L(G)$，以便满足式（3-2）。在nfa中，根据构造，存在从V_0到V_i标记为v_1的路径，从V_i到V_j标记为v_2的路径，等等。因此，很明显就有

$$V_f \in \delta^*(V_0, w)$$

其中w是被M接受的。



表示$V_i \to a_1a_2\cdots a_mV_j$



表示$V_i \to a_1a_2\cdots a_m$

图 3-15

反之，假设w是被M接受的。因为构造M的方式，接受w的自动机必须经过V_0, V_i, …到V_f这个状态序列和标记为v_1, v_2, …的路径。因此，w必须具有形式

92

$$w = v_1v_2\cdots v_kv_l$$

并且，可能得到推导

$$V_0 \Rightarrow v_1V_i \Rightarrow v_1v_2V_j \overset{*}{\Rightarrow} v_1v_2\cdots v_kV_k \Rightarrow v_1v_2\cdots v_kv_l$$

因此，w属于$L(G)$，定理得以证明。■

例3.14 构造接受由下面文法生成的语言的有限自动机:

$$V_0 \to aV_1$$
$$V_1 \to abV_0 | b$$

我们先确定顶点V_0, V_1和V_f，作为构造转移图的开始。根据第一个产生规则，在顶点V_0和V_1之间建立标记为a的边。对于第二条规则，我们要引入一个附加的顶点，使得在顶点V_1和V_0之间存在一条标记为ab的路径。最后，我们需要在V_1和V_f之间增加一条标记为b的边，得到的自动机如图3-16所示。由这个文法产生，并被自动机接受的语言$L((aab)^*ab)$ 就是正则语言。 □

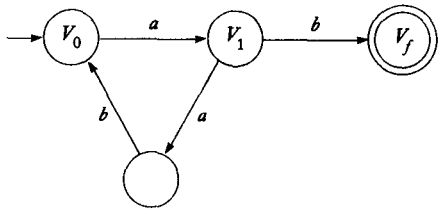


图 3-16

3.3.3 正则语言的右线性文法

为了证明任何正则语言都可以由某个右线性文法生成，我们从语言的dfa出发，把定理3.3的构造过程逆过来。现在，dfa的状态变成了文法的变量，转移函数中的终结符变成了产生式中的符号。

定理3.4 *如果L是定义在字母表Σ上的正则语言，那么存在右线性文法$G = (V, \Sigma, S, P)$，使得$L = L(G)$。*

93

证明: 设$M = (Q, \Sigma, \delta, q_0, F)$ 是接受L的dfa。我们假设$Q = \{q_0, q_1, \cdots, q_n\}$和$\Sigma = \{a_1, a_2, \cdots, a_m\}$。构造右线性文法$G = (V, \Sigma, S, P)$，其中

$$V = \{q_0, q_1, \cdots, q_n\}$$

并且$S = q_0$。对于M中的每个转移函数

$$\delta(q_i, a_j) = q_k$$

我们把下面的产生式放入P中

$$q_i \rightarrow a_j q_k \tag{3-3}$$

另外，如果q_k属于F，则向P中增加产生式

$$q_k \rightarrow \lambda \tag{3-4}$$

我们首先证明，按这种方式定义的G能够产生L中的每个符号串。已知$w \in L$，并且具有形式

$$w = a_i a_j \cdots a_k a_l$$

若M接受这个符号串，则它要经过下面的迁移

$$\begin{aligned}
&\delta(q_0, a_i) = q_p \\
&\delta(q_p, a_j) = q_r \\
&\quad\vdots \\
&\delta(q_s, a_k) = q_t \\
&\delta(q_t, a_l) = q_f \in F
\end{aligned}$$

通过构造，对于每个δ，文法都有一个相应的产生式。因此，我们可以用文法G和$w \in L(G)$ 进行推导

$$q_0 \Rightarrow a_i q_p \Rightarrow a_i a_j q_r \overset{*}{\Rightarrow} a_i a_j \cdots a_k q_t \Rightarrow a_i a_j \cdots a_k a_l q_f \Rightarrow a_i a_j \cdots a_k a_l \tag{3-5}$$

反之，如果$w \in L(G)$，那么它的推导一定具有式（3-5）的形式。这就意味着

$$\delta^*(q_0, a_i a_j \cdots a_k a_l) = q_f$$

从而完成了证明。■

94

为了构造文法，我们需要注意把M限制成dfa对于证明定理3.4而言不是必需的。略加修改后，如果M是nfa，可以使用同样的构造过程。

例3.15 为$L(aab^*a)$ 构造右线性文法。一个nfa的转移函数和相应的文法产生式在图3-17中给出。按照定理3.4的构造方法可以容易地得到结果。按照构造的文法可以推导出符号串$aaba$，过程为

$$q_0 \Rightarrow aq_1 \Rightarrow aaq_2 \Rightarrow aabq_2 \Rightarrow aabaq_f \Rightarrow aaba$$ □

$\delta(q_0, a) = \{q_1\}$	$q_0 \longrightarrow aq_1$
$\delta(q_1, a) = \{q_2\}$	$q_1 \longrightarrow aq_2$
$\delta(q_2, b) = \{q_2\}$	$q_2 \longrightarrow bq_2$
$\delta(q_2, a) = \{q_f\}$	$q_2 \longrightarrow aq_f$
$q_f \in F$	$q_f \longrightarrow \lambda$

图 3-17

3.3.4 正则语言和正则文法的等价性

前面的两个定理建立了正则语言和右线性文法之间的联系。下面的定理建立了正则语言和左线性文法之间的类似关系。进而证明了正则文法和正则语言之间的完全等价性。

定理3.5 *语言L是正则的，当且仅当存在一个左线性文法G，使得$L = L(G)$。*

证明：我们只列出证明的主要思想。给出一个左线性文法的产生式形式

$$A \to Bv$$

或

95

$$A \to v$$

我们把G中的每个产生式分别用下面$\hat{G}$的右线性文法的产生式代替

$$A \to v^R B$$

或

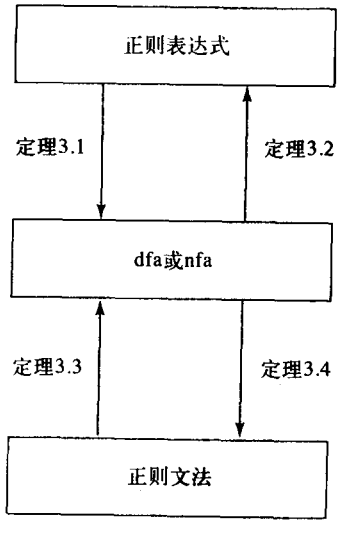
$$A \to v^R$$

举几个例子可以更清楚地说明$L(G) = (L(\hat{G}))^R$。接下来，2.3节习题12告诉我们任何正则语言的逆也是正则的。因为$\hat{G}$是右线性的，所以$L(\hat{G})$ 是正则的。同样地，$(L(\hat{G}))^R$和$L(G)$ 也有同样的结论。■

把定理3.4和定理3.5放在一起，我们可以得到正则语言和正则文法的等价性。

定理3.6 *语言L是正则的，当且仅当存在正则文法G，满足$L = L(G)$。*

现在，我们有好几种描述正则语言的方式了，包括dfa、nfa、正则表达式和正则文法。尽管在有些例子中，某种或某几种最适合，但是它们都是同样有效的表示方法。它们都对正则语言给出了完整的无二义性的定义。可以使用本章中的4个定理来建立这些概念之间的关系，如图3-18所示。



96

图 3-18

习题

1. 构造dfa接受由下面文法生成的语言:

$$S \to abA$$
$$A \to baB$$
$$B \to aA|bb$$

2. 构造一个能生成语言$L(aa^*(ab+a)^*)$的正则文法。
3. 为习题1中的语言构造左线性文法。
4. 为下面的语言构造左线性文法和右线性文法:

$$L = \{a^n b^m : n \geqslant 2, m \geqslant 3\}$$ ●

5. 为语言$L((aab^*ab)^*)$构造右线性文法。
6. 构造一个正则文法，使之产生的语言定义在$\Sigma = \{a, b\}$上，并且这个语言中的每一个符号串至多有3个a。
7. 在定理3.5中，证明$L(\hat{G}) = (L(G))^R$。 ●
8. 给出一个直接从nfa获得左线性文法的构造方法。
9. 为习题5中的语言构造一个左线性文法。
10. 为语言$L = \{a^n b^m : n + m$是偶数$\}$构造一个正则文法。 ●
11. 构造正则文法，使它可以生成语言$L = \{w \in \{a, b\}^* : n_a(w) + 3n_b(w)$ 是偶数$\}$。
12. 分别为下列定义在$\{a, b\}$上的语言构造正则文法。
    (a) $L = \{w : n_a(w)$ 和$n_b(w)$ 都是偶数$\}$ ●
    (b) $L = \{w : (n_a(w) - n_b(w)) \bmod 3 = 1\}$
    (c) $L = \{w : (n_a(w) - n_b(w)) \bmod 3 \neq 0\}$
    (d) $L = \{w : |n_a(w) - n_b(w)|$ 是奇数$\}$
13. 证明：对于任何一种不包含λ的正则语言，都存在一个产生它的右线性文法，并且产生式满足下面的形式

$$A \to aB$$

或

$$A \to a$$

其中，$A, B \in V$且$a \in T$。 97

14. 证明：对于任意文法G，若$L(G) \neq \emptyset$，则一定至少存在一个下面形式的产生式

$$A \to x$$

其中，$A \in V$且$x \in T^*$。

15. 构造一个正则文法，使它生成所有Pascal实数的集合。
16. 设$G_1 = (V_1, \Sigma, S_1, P_1)$ 是右线性文法，$G_2 = (V_2, \Sigma, S_2, P_2)$ 是左线性文法。假设V_1和V_2不相交。考察线性文法$G = (\{S\} \cup V_1 \cup V_2, \Sigma, S, P)$，其中，$S$不属于$V_1 \cup V_2$，并且$P = \{S \to S_1 | S_2\} \cup P_1 \cup P_2$。证明$L(G)$ 是正则的。 ● 98

第4章 正则语言的性质

我们已经定义了正则语言，研究了几种表示正则语言的方法，并且通过一些例子了解了它们的用途。现在提出一个新的问题：正则语言作为一种语言有多通用呢？是不是所有的形式语言都是正则的？或许我们指定的任何集合都可以被某些可能很复杂的有穷自动机接受。我们不久就会看到，这个猜测的答案是否定的。但是为了便于理解为什么是这样的，我们必须深入正则语言的本质，看看整个语言族到底有哪些性质。

我们提出的第一个问题是，如果我们对正则语言进行运算，结果会怎么样。这里提到的运算指的是简单的集合运算，如连接，以及会使语言中的每个符号串都发生变化的运算，像2.1节习题22中的例子。这样得到的语言是否仍然是正则的？我们把这个问题看成是封闭性（closure）问题。尽管大多数理论都关注这个性质，但是我们对它感兴趣的原因在于它能够帮助我们区分遇到的不同语言族。

关于语言族的第二个问题讨论我们决定某些性质的能力。例如，我们是否能够判断出一种语言是有穷的，还是无穷的？我们下面会看到，这个问题对于正则语言而言是很容易回答的，但是对于别的语言族而言就不容易回答。

最后，我们考虑最重要的问题：如何判断某种语言是否为正则的？如果这种语言确实是正则的，我们可以通过给出它的dfa、正则表达式或正则文法来证明。但是如果它不是正则的，我们需要使用其他方法来说明它不是正则的。证明一种语言不是正则的，一种办法是研究正则语言的一般性质，即：所有正则语言都具有的特点。如果我们知道这些性质，且能证明候选的语言不具备某种上述性质，那么我们就可以说这种语言不是正则的。

在本章里，我们考察正则语言的各种性质。这些性质告诉我们正则语言能够做什么，不能够做什么。以后，当我们用这些类似问题考察其他语言族时，可以使用这些性质的异同点来对照不同的语言。

4.1 正则语言的封闭性质

考虑下面的问题：给定正则语言L_1和L_2，它们的并集是否仍然是正则语言？在一些特殊的例子中，结论是显而易见的。但是这里我们想要强调的是这个问题的普遍性。对于所有的正则语言L_1和L_2，它是否都成立？事实证明这个问题的答案是肯定的，于是我们说正则语言族在并运算下是封闭（closed）的。我们对于语言的其他运算可以问类似的问题，这就使得我们要从普遍意义上去研究语言的封闭性质。

各种语言族在不同运算下的封闭性具有一定的理论研究价值。首先，我们可能还不清楚这些性质的实际意义。我们必须承认某些性质的意义不大，但是很多结论却是十分有用的。封闭性质使我们深入到语言族的更一般的本质中，从而帮助我们解决更多的实际问题。本章的后面将介绍这些例子（定理4.7和例4.13）。

4.1.1 简单集合运算的封闭性

我们首先看看普通集合运算上，如并和交，正则语言的封闭性。

定理4.1 如果L_1和L_2是正则语言，那么$L_1 \cup L_2$, $L_1 \cap L_2$, L_1L_2, $\overline{L_1}$和L_1^*也都是正则的。于是，我们认为正则语言族在并、交、连接、补和星闭包上都是封闭的。

证明：如果L_1和L_2是正则的，那么一定存在正则表达式r_1和r_2，使得$L_1 = L(r_1)$, $L_2 = L(r_2)$。根据定义，$r_1 + r_2$, r_1r_2和r_1^*分别是表示语言$L_1 \cup L_2$，L_1L_2和L_1^*的正则表达式。因此，并、连接和星闭包上都是封闭的。

100

为了证明补运算的封闭性，设接受L_1的一个dfa为$M = (Q, \Sigma, \delta, q_0, F)$。那么dfa

$$\hat{M} = (Q, \Sigma, \delta, q_0, Q - F)$$

接受$\overline{L_1}$。这个结论是相当直接的，我们已经在2.1节习题4中提到这个结论。注意dfa的定义，我们假设δ^*是个全函数，因此$\delta^*(q_0, w)$是定义在所有$w \in \Sigma^*$上的。所以，要么$\delta^*(q_0, w)$是终态，在这种情况下，$w \in L$；要么$\delta^*(q_0, w) \in Q - F$，在这种情况下，$w \in \overline{L}$。

演示交运算下的封闭性要麻烦些。设$L_1 = L(M_1)$, $L_2 = L(M_2)$，其中，$M_1 = (Q, \Sigma, \delta_1, q_0, F_1)$和$M_2 = (P, \Sigma, \delta_2, p_0, F_2)$都是dfa。我们把$M_1$和$M_2$合并起来构造出组合自动机$\hat{M} = (\hat{Q}, \Sigma, \hat{\delta}(q_0, p_0), \hat{F})$，状态集为$\hat{Q} = Q \times P$，它的元素为状态对$(q_i, p_j)$。当$M_1$处于状态$q_i$，$M_2$处于状态$p_j$时，则转移函数$\hat{\delta}$定义$\hat{M}$位于状态$(q_i, p_j)$。即：当

$$\delta_1(q_i, a) = q_k$$

并且

$$\delta_2(p_j, a) = p_l$$

则有

$$\hat{\delta}((q_i, p_j), a) = (q_k, p_l)$$

定义$\hat{F}$为所有的(q_i, p_j)构成的集合，其中，$q_i \in F_1$, $p_j \in F_2$。那么容易证明：$w \in L_1 \cap L_2$当且仅当它被$\hat{M}$接受。因此，$L_1 \cap L_2$是正则的。■

交运算封闭性的证明是构造法证明的一个典型例子。不仅仅因为它构造出了理想的结果，而且是因为它清晰地给出了如何为两个正则语言的交集构造有穷接受器。构造性证明贯穿了整本书，它很重要，因为它使我们能深入研究结果，并且可以作为研究实际算法的入手点。这里有一些短小的、非构造性的证明（或者至少不是明显的构造性证明）。为了证明交运算的封闭性，我们首先使用德摩根定律，对式（1-3）两边分别取补，于是对于任意语言L_1和L_2，都有

101

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

如果L_1和L_2是正则的，那么根据补运算的封闭性，有$\overline{L_1}$和$\overline{L_2}$也是正则的。使用并运算的封闭性，我们得到$\overline{L_1} \cup \overline{L_2}$也是正则的。再次使用补运算的封闭性，可以得到

$$\overline{\overline{L_1} \cup \overline{L_2}} = L_1 \cap L_2$$

是正则的。

下面的例子是同样想法的一个变换。

例4.1 证明正则语言族在差运算下是封闭的。换句话说，我们要证明：如果L_1和L_2是正则的，那么$L_1 - L_2$一定也是正则的。

根据集合差运算的定义可以立刻得到所需的集合等式，即

$$L_1 - L_2 = L_1 \cap \overline{L_2}$$

L_2是正则的，意味着$\overline{L_2}$也是正则的。然后，因为正则语言在交运算下的封闭性，我们可以知道$L_1 \cap \overline{L_2}$也是正则的，于是证明完成了。 □

根据这些基本命题可以推导出一些其他的封闭性质。

定理4.2 正则语言族在转置运算上也是封闭的。

证明：这个命题的证明任务是2.3节的一个习题，这里给出一些细节。假设L是正则语言，那么我们构造一个只有一个终态的nfa接受它。根据2.3节习题7，这总是可能的。在这个nfa的转移图中，我们把初态的顶点作为终态的顶点，把终态的顶点作为初态的顶点，把所有边的方向调转过来。可以很直观地证明，修改后的nfa接受w^R，当且仅当原来的nfa接受w。因此，修改的nfa接受L^R，从而证明了转置运算的封闭性。■ 102

4.1.2 其他运算的封闭性

除了语言的标准运算外，还可以定义语言的其他运算，并研究这些运算的封闭性。有很多这样的研究结果，这里我们只选择两个典型的例子，其他的留作本节的习题。

定义4.1 假设Σ和Γ是字母表。那么函数

$$h : \Sigma \rightarrow \Gamma^*$$

称为同态（homomorphism）。换句话说，同态就是把一个字母用一个符号串代替。h函数的定义域可按照某一种明显的形式扩展成符号串。如果

$$w = a_1 a_2 \cdots a_n$$

那么

$$h(w) = h(a_1)h(a_2)\cdots h(a_n)$$

如果L是定义在Σ上的语言，那么它的同态映像（homomorphic image）就定义为

$$h(L) = \{h(w) : w \in L\}$$

例4.2 已知$\Sigma = \{a, b\}$和$\Gamma = (a, b, c)$，定义h为

$$h(a) = ab$$
$$h(b) = bbc$$

那么$h(aba) = abbbcab$。$L = \{aa, aba\}$的同态映像为语言$h(L) = \{abab, abbbcab\}$。 □

如果我们有一种语言L的正则表达式r，那么对r中的每个Σ符号简单地应用同态，就可以得到$h(L)$的正则表达式。 103

例4.3 已知$\Sigma = \{a, b\}$和$\Gamma = \{b, c, d\}$。定义h为

$$h(a) = dbcc$$
$$h(b) = bbc$$

如果L是正则语言，使用正则表达式

$$r = (a + b^*)(aa)^*$$

表示，那么

$$r_1 = (dbcc + (bdc)^*)(dbccdbcc)^*$$

表示的是正则语言$h(L)$。 □

考察了这个典型的例子之后，我们将得到正则语言在同态运算上封闭的一般结论。

定理4.3 *设h是同态函数。如果L是正则的，那么它的同态映像$h(L)$也是正则的。因此，正则语言族在任意同态运算上都是封闭的。*

证明：设L是用正则表达式r表示的正则语言。对于r中的每个符号$a \in \Sigma$对应的$h(a)$，我们都用$h(r)$来替换。根据正则表达式的定义，我们可以直接证明得到的结果也是正则表达式。很容易看出，得到的表达式表示的是$h(L)$。我们需要证明的仅仅是，对于每个$w \in L(r)$，对应的$h(w)$都属于$L(h(r))$。相反，对于$L(h(r))$中的每个v，在L中都存在一个w，使得$v = h(w)$。我们把这段证明的细节留作习题，声明$h(L)$是正则的。■

定义4.2 *设L_1和L_2是定义在同一个字母表上的语言，那么L_1和L_2的右商（right quotient）定义为*

$$L_1/L_2 = \{x : xy \in L_1 \text{对于某个} y \in L_2\} \tag{4-1}$$

为了计算L_1和L_2的右商，我们取出所有满足下面要求的符号串：L_1中所有其后缀属于L_2的符号串。由每个这样的符号串去掉后缀后形成的符号串都属于L_1/L_2。

104

例4.4 如果

$$L_1 = \{a^n b^m : n \geqslant 1, m \geqslant 0\} \cup \{ba\}$$

并且

$$L_2 = \{b^m : m \geqslant 1\}$$

那么

$$L_1/L_2 = \{a^n b^m : n \geqslant 1, m \geqslant 0\}$$

L_2中的符号串至少包含一个b。因此，我们可以通过这样的方法来得到答案：将L_1中后缀至少有一个b的符号串去掉一个或多个b。

这里注意L_1，L_2和L_1/L_2都是正则的。这就暗示我们两个正则语言的右商也是正则的。我们将在下一个定理中，通过使用L_1和L_2的dfa构造L_1/L_2的dfa来证明这一点。在我们完全描述这个构造过程之前，我们先看看下面这个例子。首先构造接受L_1的dfa，比如$M_1 = (Q, \Sigma, \delta, q_0, F)$，即图4-1中的自动机。因为$L_1/L_2$的自动机必须接受$L_1$中的任何前缀，所以我们修改$M_1$，以便于如果$y$满足图4-1，$M_1$就接受$x$。

困难在于确定是否存在某一个y，使得$xy \in L_1$，并且$y \in L_2$。为了解决这个困难，我们要确定，对于任意一个$q \in Q$，是否存在一条从q到某个终态的标记为v的通道，其中v满足$v \in L_2$。如果存在的话，任何满足$\delta(q_0, x) = q$的x，都将属于L_1/L_2。我们相应地修改自动机，使q成为终态。

为了把这点应用到我们的例子中去，我们逐个检查状态$q_0, q_1, q_2, q_3, q_4, q_5$，判断是否存在一条到$q_1, q_2$或$q_4$的通道，其标记为$bb^*$。我们发现只有$q_1$和$q_2$满足条件，而$q_0, q_3, q_4$不满足条件。

最终得到的接受L_1/L_2的自动机如图4-2所示。通过对该自动机进行检查可看出，这个构造方法是可行的。这个想法在下一个定理中将得到推广。 □

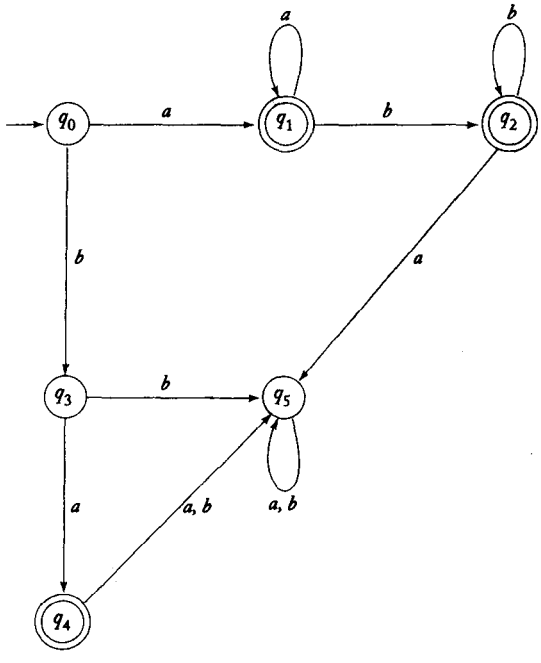


图 4-1

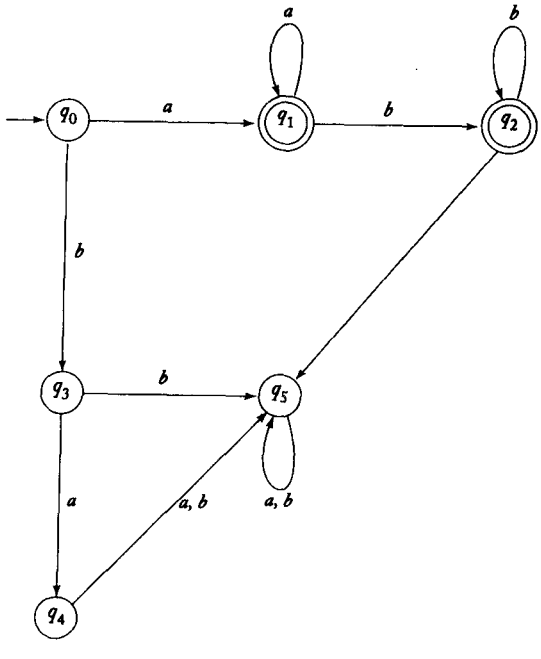


图 4-2

定理4.4 如果L_1和L_2是正则语言，那么L_1/L_2也是正则的。于是我们称正则语言族与正则

语言在右商运算上是封闭的。

证明：设$L_1 = L(M)$，其中$M = (Q, \Sigma, \delta, q_0, F)$是dfa。我们按照下面的方法构造另一个dfa $\hat{M} = (Q, \Sigma, \delta, q_0, \hat{F})$。对于每个$q_i \in Q$，判断是否存在一个$y \in L_2$，使得

$$\delta^*(q_i, y) = q_f \in F$$

105~106 这个是否存在的判断可以通过考察另一个dfa $M_i = (Q, \Sigma, \delta, q_i, F)$来进行。自动机$M_i$是把$M$中的初态$q_0$由$q_i$代替。现在我们判断是否存在一个$y$，既属于$L(M_i)$，又属于$L_2$。为了解决这个问题，我们使用定理4.1中两个正则语言交集的构造方法，构造$L_2 \cap L(M_i)$的转移图。如果在初态和任何终态之间存在一条路径，那么$L_2 \cap L(M_i)$就非空。在这种情况下，把q_i加到$\hat{F}$中。对于任意$q_i \in Q$，重复这一过程，从而确定$\hat{F}$，构造$\hat{M}$。

为了证明$L(\hat{M}) = L_1/L_2$，我们假设x是L_1/L_2中的任意元素。那么一定存在着一个$y \in L_2$，使得$xy \in L_1$。这就意味着

$$\delta^*(q_0, xy) \in F$$

因此，一定存在某个$q \in Q$，使得

$$\delta^*(q_0, x) = q$$

并且

$$\delta^*(q, y) \in F$$

因此，根据构造可知$q \in \hat{F}$，并且$\hat{M}$接受x，因为，$\delta^*(q_0, x)$属于$\hat{F}$。

相反，对于$\hat{M}$接受的任意x，我们都有

$$\delta^*(q_0, x) = q \in \hat{F}$$

但是根据构造法，我们得到一定存在一个$y \in L_2$，满足$\delta^*(q_0, y) \in F$。因此，xy属于L_1，并且x属于L_1/L_2。我们于是得到结论

$$L(\hat{M}) = L_1 / L_2$$

由此可知，L_1/L_2是正则的。■

例4.5 构造L_1/L_2，已知

$$L_1 = L(a^*baa^*)$$
$$L_2 = L(ab^*)$$

我们首先找到接受L_1的dfa。这个容易，图4-3给出了一种解。这个例子很简单，所以我们可以跳过构造的过程。根据图4-3，显然有

$$L(M_0) \cap L_2 = \emptyset$$
$$L(M_1) \cap L_2 = \{a\} \neq \emptyset$$
$$L(M_2) \cap L_2 = \{a\} \neq \emptyset$$
$$L(M_3) \cap L_2 = \emptyset$$

107

因此，确定了接受L_1/L_2的自动机，结果如图4-4。这个自动机接受由正则表达式$a^*b + a^*baa^*$表示的语言，这个正则表达式可以简化成a^*ba^*。因此$L_1/L_2 = L(a^*ba^*)$。□

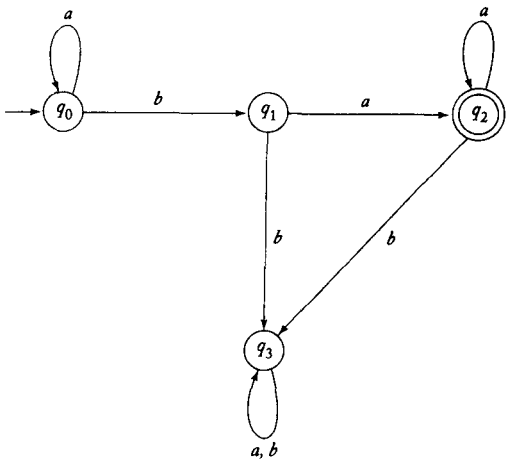


图 4-3

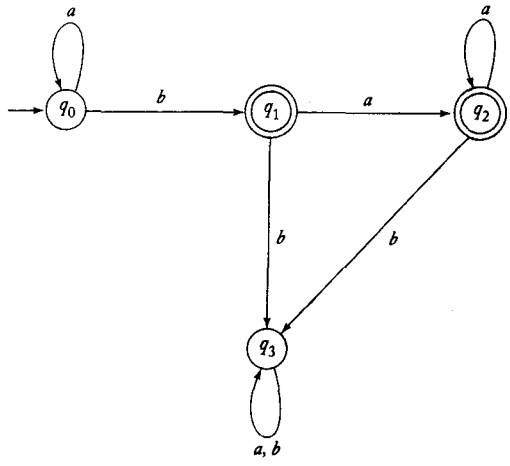


图 4-4

108

习题

1. 完成定理4.1中交运算封闭性的构造性证明的细节。
2. 使用定理4.1中的构造方法来分别构造接受下面语言的nfa。
   (a) $L((a+b)a^*)\cap L(baa^*)$ ●
   (b) $L(ab^*a^*)\cap L(a^*b^*a)$
3. 在例4.1中，我们证明了正则语言差运算的封闭性，但是证明方法不是构造性的。提供一个构造性的证明。
4. 在定理4.3中，证明$h(r)$是正则表达式。然后证明$h(r)$表示的是$h(L)$。
5. 证明正则语言族在有穷并运算和有穷交运算上是封闭的。即：如果$L_1, L_2, \cdots, L_n$是正则的，那么

$$L_U = \bigcup_{i=\{1,2,\cdots,n\}} L_i \text{ 和 } L_I = \bigcap_{i=\{1,2,\cdots,n\}} L_i$$

   都是正则的。
6. 两个集合S_1和S_2的*对称差*（symmetric difference）定义如下：

$$S_1 \ominus S_2 = \{x : x\in S_1\text{或者}x\in S_2\text{，但是}x\text{不同时存在于}S_1\text{和}S_2\text{中}\}$$

   证明：正则语言族在对称差运算上是封闭的。
7. 两个语言的*nor*定义为

$$nor(L_1, L_2) = \{w : w\notin L_1\text{并且}w\notin L_2\}$$

   证明：正则语言族在运算*nor*上是封闭的。●
8. 两个语言的补或*cor*运算定义为

$$cor(L_1, L_2) = \{w : w\in \overline{L}_1\text{或者}w\in \overline{L}_2\}$$

   证明：正则语言族在*cor*运算下是封闭的。
9. 下面的结论对于所有的正则语言和同态都成立吗？

(a) $h(L_1 \cup L_2) = h(L_1) \cup h(L_2)$

(b) $h(L_1 \cap L_2) = h(L_1) \cap h(L_2)$

109 (c) $h(L_1L_2) = h(L_1)h(L_2)$

10. 已知$L_1 = L(a^*baa^*)$ 和$L_2 = L(aba^*)$，构造L_1/L_2。

11. 证明：对于所有语言L_1和L_2，都有$L_1 = L_1L_2/L_2$成立。

★12. 假设我们已知$L_1 \cup L_2$是正则的，L_1是有穷的。我们可以由此推断L_2是正则的吗？●

13. 如果L是正则语言，证明$L_1 = \{uv : u \in L, |v| = 2\}$也是正则的。

14. 如果L是正则语言，证明语言$\{uv : u \in L, v \in L^R\}$也是正则的。●

15. L_1相对于L_2的左商定义为

$$L_2/L_1 = \{y : x \in L_2, xy \in L_1\}$$

证明：正则语言族在左商上是封闭的。

16. 证明：如果命题“如果L_1和$L_1 \cup L_2$是正则的，那么L_2一定也是正则的”对于所有的L_1和L_2是正确的，那么所有的语言都是正则的。●

17. 语言的尾（tail）定义为符号串所有后缀的集合，即：

$$tail(L) = \{y : 对于某个 x \in \Sigma^*, xy \in L\}$$

证明：如果L是正则的，那么$tail(L)$ 也是正则的。

18. 语言的头（head）定义为符号串所有前缀构成的集合。即：

$$head(L) = \{x : 对于某个 y \in \Sigma^*, xy \in L\}$$

证明：正则语言族在这个运算下是封闭的。●

19. 定义语言和符号串的运算*third*如下

$$third(a_1a_2a_3a_4a_5a_6\cdots) = a_3a_6\cdots$$

这个定义的定义域可以扩展到整个语言范围。证明正则语言族在这个运算下是封闭的。

20. $a_1a_2\cdots a_n$为任意符号串，定义移位（shift）运算如下

$$shift(a_1a_2\cdots a_n) = a_2\cdots a_na_1$$

由此，我们可以定义一个语言上的运算如下

$$shift(L) = \{v : 对于某个 w \in L, v = shift(w)\}$$

证明正则性在*shift*下是封闭的。

21. 定义符号串和语言上的运算交换（exchange）分别为

$$exchange(a_1a_2\cdots a_{n-1}a_n) = a_na_2\cdots a_{n-1}a_1$$

和

$$exchange(L) = \{v : 对于所有 w \in L, v = exchange(w)\}$$

110 证明正则语言族在*exchange*下是封闭的。

★22. 两个语言L_1和L_2的运算*shuffle*定义为

$$shuffle(L_1, L_2) = \{w_1v_1w_2v_2\cdots w_mv_m : 对于所有 w_i,\ v_i \in \Sigma^*, w_1w_2\cdots w_m \in L_1, v_1v_2\cdots v_m \in L_2\}$$

证明：正则语言在*shuffle*运算下是封闭的。

★23. 把语言L上的运算min*us*5定义为：把L所有符号串中左面起第5个符号去掉获得的符号串的集合（长度小于5的符号串保持不变）。证明：正则语言族在这个运算上是封闭的。

★24. 定义在L上的左边（leftside）运算为

$$leftside(L) = \{w : ww^R \in L\}$$

正则语言族在这个运算上是否是封闭的？

25. 语言的min运算定义为

$$\min(L) = \{w \in L : 不存在 u \in L, v \in \Sigma^+, 使得 w = uv\}$$

证明正则语言族在min运算上是封闭的。

26. 设G_1和G_2是两个正则文法。如何从这两个文法给出下列语言的正则文法：

(a) $L(G_1) \cup L(G_2)$ ●

(b) $L(G_1)L(G_2)$ ●

(c) $L(G_1)^*$ ●

4.2 正则语言的基本问题

现在我们来看一个很基本的问题：给定语言L和符号串w，我们是否能够判断w是不是L的元素？这是一个成员资格（membership）问题，解决这一问题的方法称为成员资格算法。我们对这个问题无能为力，因为找不到有效的成员资格算法来解决语言问题。我们将在后续的讨论中关注这个问题和成员资格算法的本质。这通常是一个难题。但是对于正则语言而言，就比较容易。

首先考虑当我们说“给定语言……”时，我们到底是什么意思。在很多命题中，这点很重要，这样才能够意思明确。我们有几种方式来描述正则语言：非形式化口头描述、集合表示、有穷自动机、正则表达式和正则文法。只有后三项是充分定义的，并在定理中使用过的。因此，我们说一个正则语言是以标准表示法（standard representation）给定的，当且仅当它是被有穷自动机、正则表达式或正则文法表示的。 111

定理4.5 *给定一种标准表示法，表示任何定义在Σ上的正则语言L和任何$w \in \Sigma^*$。这种标准表示法都有判断w是否属于L的算法。*

证明：我们用某个dfa表示这个语言，然后测试w看它是否能够被自动机接受。■

另一个重要的问题是一个语言是有穷的，还是无穷的；两个语言是否相同；一个语言是否是另一个语言的子集。至少对于正则语言而言，这几个问题都容易回答。

定理4.6 *对于使用标准表示法给出的任何一个正则语言，都存在算法可以判断这个语言是空的、有穷的，还是无穷的。*

证明：如果我们用dfa的转移图表示这种语言，结论是显而易见的。如果从初态顶点到任何一个终态顶点之间存在一条简单路径，那么这个语言就是非空的。

为了判断语言是否是无穷的，先找到构成某个回路的所有顶点。如果这些顶点中的任何一个位于从初态顶点到终态顶点的路径上，那么语言就是无穷的。否则，语言就是有穷的。■

两个语言的等价问题也是一个很重要的常用问题。一个语言通常有几种定义方式，我们需要知道的是，尽管它们的外在形式不同，但它们表达的是否是同一个语言。这通常是一个

难题。即使对于正则语言，这个命题的结论也不是明显的。我们不可能用逐句比较来证明，因为这种办法只适用于有穷语言。即便使用正则表达式、文法或者dfa，也是不容易看到结论的。一个比较合适的解决方法是使用已有的封闭性质。

定理4.7 *对使用标准表示法给定的任意两个正则语言L_1和L_2，总存在判断$L_1 = L_2$是否成立的算法。*

证明：使用L_1和L_2，我们定义语言

112

$$L_3 = (L_1 \cap \overline{L_2}) \cup (\overline{L_1} \cap L_2)$$

根据封闭性质，L_3是正则的，因此我们构造dfa$M$接受L_3。一旦我们有了M，我们就可以使用定理4.6中的算法来判断L_3是否为空。但是根据1.1节习题8，我们知道，$L_3 = \varnothing$当且仅当$L_1 = L_2$。■

这些都是基本却显然的结论。对于正则语言，定理4.5到定理4.7提出的问题可以轻松地回答，但是当我们处理更大的语言族时，就不那么容易了。后面，我们将会碰到几个类似的问题。我们唯一可以预料到的是，问题会越来越难回答，甚至可能根本没法回答。

习题

本节的所有习题都假设给定的正则语言使用标准表示法。

1. 证明：对于任意w，任意正则语言L_1和L_2，都存在判断$w \in L_1 - L_2$是否成立的算法。●
2. 证明：对于任意正则语言L_1和L_2，都存在判断 $L_1 \subseteq L_2$是否成立的算法。●
3. 证明：对于任意正则语言L，都存在判断$\lambda \in L$是否成立的算法。
4. 证明：对于任意正则语言L_1和L_2，都存在算法判断$L_1 = L_1/L_2$是否成立。
5. 如果$L = L^R$，我们称这个语言是回文（palindrome）语言。构造一个算法判断一个给定的正则语言是否是回文语言。●
6. 给出一个算法，用来判断正则语言L是否包含任意符号串w，满足$w^R \in L$。
7. 已知三个正则语言L, L_1, L_2，给出一个判断$L = L_1L_2$是否成立的算法。
8. 给定任意正则语言L，给出一个判断$L = L^*$是否成立的算法。
9. 设L是定义在Σ上的正则语言，$\hat{w}$是Σ^*中的任意符号串。给出一个算法判断L是否包含w，满足$\hat{w}$是它的子串。即满足 $w = u\hat{w}v$，其中$u, v \in \Sigma^*$。
10. 证明：对于任意正则语言L，都存在判断$L = shuffle(L, L)$是否成立的算法。
11. 运算$tail(L)$定义为

113

$$tail(L) = \{v : uv \in L, u, v \in \Sigma^*\}$$

证明对于任意正则语言L，都存在判断$L = tail(L)$是否成立的算法。

12. 设L是定义在$\Sigma = \{a, b\}$上的正则语言。证明：存在判断L包括所有长度为偶数的符号串是否成立的算法。●
13. 构造一个算法，判断正则语言L是否包含长度为偶数的无穷长的符号串。
14. 给出一个算法，判断给定一个正则文法G，$L(G) = \Sigma^*$是否成立。

4.3 识别非正则语言

从我们举的大多数例子中，可以看出正则语言是无穷的。事实上正则语言和具有有穷存

储器的自动机有关，但是在正则语言结构上做了一定的限制。如果想保证正则性，就一定要遵循一些限制。直觉告诉我们，只有在处理符号串的过程中，每个阶段记录的信息都要严格限制，这样得到的语言才是正则的。这种想法是正确的，但是，这个想法应该用更有意义的方式来准确证明，可以用有种方式来获得这种准确性。

4.3.1 使用鸽巢原理

术语“鸽巢原理”用于数学领域，主要内容如下。如果我们把n个物品放到m个盒子（鸽巢）中，并且$n>m$，那么至少有一个盒子里装了一个以上的物品。这是一个明显的事实，但是根据这个事实，我们可以得到更多深刻的结论。

例4.6 语言$L=\{a^nb^n : n\geqslant 0\}$是正则的吗？答案是否定的，我们将通过反证法证明答案。

假设L是正则的。那么，一定存在接受它的某个dfa $M=(Q,\{a,b\},\delta,q_0,F)$。对于$i=1,2,3,\cdots$，考察$\delta^*(q_0,a^i)$。因为有无穷个$i$，但是$M$中的状态数是有穷的，鸽巢原理告诉我们一定存在某个状态，设为q，满足

$$\delta^*(q_0,a^n)=q$$

和

$$\delta^*(q_0,a^m)=q$$

114

其中，$n\neq m$。但是因为M接受a^nb^n，所以我们有

$$\delta^*(q,b^n)=q_f\in F$$

因此，我们可以得出

$$\begin{aligned}\delta^*(q_0,a^mb^n)&=\delta^*(\delta^*(q_0,a^m),b^n)\\&=\delta^*(q,b^n)\\&=q_f\end{aligned}$$

这就和最初的假设——只有当$n=m$时，M接受a^mb^n——矛盾。因此，我们得到结论L不可能是正则的。 □

在上面的论证中，鸽巢原理只是我们准确阐明问题的一种方法。通过它，我们阐明了有穷自动机具有有穷存储器意味着什么。为了接受所有的a^nb^n，自动机需要区别所有的前缀a^n和a^m。但是因为自动机只有有穷个内部状态，一定存在一些n和m，不能够区别。

为了在不同的情况下使用上述结论，我们把它整理成有普遍意义的定理。表现形式有几种，我们只给出最著名的一种。

4.3.2 泵引理

下面的结论，是另一种形式的鸽巢原理，一般称为正则语言的*泵引理*（pumping lemma）。它的证明是基于具有n个顶点的转移图，任何长度不少于n的通道一定重复了某个顶点，即：存在回路。

定理4.8 *设L是无穷的正则语言。那么一定存在某个正整数m，满足，任意$w\in L$, $|w|\geqslant m$，可以分解成*

$$w=xyz$$

其中

$$|xy| \leqslant m$$
$$|y| \geqslant 1$$

对于所有$i = 0, 1, 2, \cdots$，满足

$$w_i = xy^iz \tag{4-2}$$

115 也属于L。

为了释义这一点，L中每个足够长的符号串都要拆分成3部分，任意重复中间部分，形成了L的另一个符号串。我们称中间的符号串为“被抽出”，因此，这个结论称为泵引理。

证明：如果L是正则语言，那么一定存在一个dfa识别它。设这个dfa的状态分别标有$q_0, q_1, q_2, \cdots, q_n$。现在我们取$L$中的任意符号串$w$，满足$|w| \geqslant m = n + 1$。因为假设$L$是无穷的，所以一定可以这样做。考虑自动机处理$w$时，经过的状态集合，比如

$$q_0, q_i, q_j, \cdots, q_f$$

因为这个序列共有$|w| + 1$项，所以至少有一个状态会被重复，并且这个重复的发生一定不晚于第n次移动。因此，这个序列应该形如

$$q_0, q_i, q_j, \cdots, q_r, \cdots, q_r, \cdots, q_f$$

这就意味着一定存在w的子串x, y, z，满足

$$\delta^*(q_0, x) = q_r$$
$$\delta^*(q_r, y) = q_r$$
$$\delta^*(q_r, z) = q_f$$

其中$|xy| \leqslant n + 1 = m$, $|y| \geqslant 1$。由此，我们可以立刻得到

$$\delta^*(q_0, xz) = q_f$$

和

$$\delta^*(q_0, xy^2z) = q_f$$
$$\delta^*(q_0, xy^3z) = q_f$$

等等。从而完成了定理的证明。■

我们只给出无穷语言的泵引理。尽管有穷语言也是正则语言，但是因为泵自动生成无穷集合，所以不能按照上面这样抽出相应的符号串。这个定理对于有穷语言依然成立，但是没什么意义。泵引理中的m要比最长的符号串的长度还大，所以不能够抽出符号串。

同例4.6中的鸽巢原理类似，泵引理可以用于证明某种语言不是正则的。证明也是使用反
116 证法。正如我们阐述的，泵引理不能用于证明一种语言是正则的。即使我们能证明（通常很困难）被抽出的符号串仍然属于原语言，但是在定理4.8中，我们找不到得出这种语言是正则的依据。

例4.7 使用泵引理证明$L = \{a^nb^n : n \geqslant 0\}$不是正则的。假设$L$是正则的，那么泵引理一定成立。我们不知道$m$的值，但是不论它取什么值，我们都可以取$n = m$。因此，子串$y$一定仅包含$a$。假设$|y| = k$。那么，根据式（4-2）使$i = 0$获得的符号串就是

$$w_0 = a^{m-k}b^m$$

显然，这个符号串不属于L。这与泵引理矛盾。因此，假设不成立，L不是正则的。 □

在应用泵引理时，我们必须记住定理的内容。定理给我们的保证是：存在m，且可以分解成xyz的形式，但是我们不知道它们具体是什么。我们不能仅仅因为m或xyz的某个特殊值使得泵引理不成立就说矛盾。另外，泵引理对每个$w \in L$和i都成立。因此，即便有一个w或i使得泵引理不成立，这种语言都不是正则的。

正确的证明过程可以看成是我们和对手玩一个游戏。我们的目的是要通过确定泵引理的矛盾来赢得比赛，而对手是要阻挡我们。游戏共有四步。

1. 对手选一个m。

2. 对给定的m，我们在L中找出一个长度不小于m的符号串w。我们可以任意挑w，只要满足$w \in L$并且$|w| \geqslant m$。

3. 对手选择一种分解方式，分解成xyz，要求$|xy| \leqslant m$，$|y| \geqslant 1$。我们必须假设对手做出的选择一定最不利于我们赢得比赛。

4. 我们使用某种方式挑选一个i，使得被抽出的符号串w_i，根据式（4-2），不属于L。如果我们可以完成这一步，那么我们就赢了。

无论对手的选择是什么我们都可以获胜的方法等价于证明语言不是正则的。这里，第2步是关键。尽管我们不可以对对手分解w的方式作要求，但是我们可以选择w，使对手在第3步受到限制，x, y和z的选择会使得我们在下一步推出违反泵引理的结论。 117

例4.8 设$\Sigma = \{a, b\}$。证明$L = \{ww^R : w \in \Sigma^*\}$不是正则的。

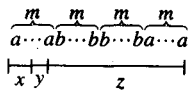


图 4-5

不论对手在第1步选择什么样的m，我们都能够选择如图4-5所示的w。再加上要求$|xy| \leqslant m$，所以对手在第3步受到限制的选择y只能包含字母a。第4步，我们令$i = 0$。这样获得的符号串左侧比右侧多了几个a，因此，就不具备ww^R的形式了。所以，L不是正则的。

注意如果我们选择的w太短，那么对手就会选出具有偶数个b的y。这样，我们在最后一步就不能推出和泵引理矛盾的结论了。如果我们选择全部由a构成的符号串，比如$w = a^{2m}$，它属于L。这样也无法推出和泵引理矛盾的结论。因为，为了打败我们，对手只需选出$y = aa$，这样对于所有的i，w_i都属于L。我们就输了。

为了应用泵引理，我们不可以假设对手会犯错误。比如，如果我们选择$w = a^{2m}$，对手会选择$y = a$，然后选择长度为奇数的符号串w_0，因此，得到的符号串不属于L。但是任何假设对手会如此配合我们的证明都是不正确的。 □ 118

例4.9 设$\Sigma = \{a, b\}$。语言$L = \{w \in \Sigma^* : n_a(w) < n_b(w)\}$不是正则的。

假设我们给定m。因为我们有完全的自由选择，所以我们选择$w = a^m b^{m+1}$。现在，因为$|xy|$不可能比m大，所以对手只能选出一个全部由a构成的y。即：

$$y = a^k, 1 \leqslant k \leqslant m$$

我们现在泵出，令$i = 2$，得到的符号串是

$$w_2 = a^{m+k}b^{m+1}$$

不属于L。因此，违反泵引理，L不是正则的。 □

例4.10 语言$L = \{(ab)^n a^k : n > k, k \geqslant 0\}$不是正则的。

给定m，我们定义符号串为

$$w = (ab)^{m+1}a^m$$

属于L。因为约束$|xy| \leqslant m$，所以x和y都必须是由ab构成的符号串中的一部分。x的选择不影响证明，因此，我们考虑y的选择。如果我们的对手选择$y = a$，我们就选择$i = 0$，从而获得一个不属于$L((ab)^* a^*)$的符号串。如果对手选择$y = ab$，我们就还选择$i = 0$。这时得到$(ab)^m a^m$，它仍然不属于L。同样，我们可以处理对手的所有可能选择，从而证明我们的断言。 □

例4.11 证明$L = \{a^{n!} : n \geqslant 0\}$不是正则的。

119 给定对手选择m，我们定义符号串为$a^{m!}$（除非对手选择$m < 3$，我们可以用$a^{3!}$代替w）。w的各种分解只是在子串的长度上有区别。假设对手挑选y，满足

$$|y| = k \leqslant m$$

那么我们就考虑长度为$m! - k$的xz。只有存在一个j，满足

$$m! - k = j!$$

时，这个符号串才属于L。但是这是不可能的，因为对于$m > 2$和$k \leqslant m$我们都有

$$m! - k > (m-1)!$$

因此，这种语言不是正则的。 □

在某些情况下，封闭性质可以用来把一个给定问题和我们已经分过类的某个问题联系起来，这样做，比直接应用泵引理简单得多。

例4.12 证明语言$L = \{a^n b^k c^{n+k} : n \geqslant 0, k \geqslant 0\}$不是正则的。

直接应用泵引理不难，但是使用同态下的封闭性会更容易。令

$$h(a) = a, h(b) = a, h(c) = c$$

则有

$$\begin{aligned} h(L) &= \{a^{n+k}c^{n+k} : n + k \geqslant 0\} \\ &= \{a^i c^i : i \geqslant 0\} \end{aligned}$$

但是因为我们知道这种语言不是正则的，所以L也不可能是正则的。 □

120 **例4.13** 证明语言$L = \{a^n b^l : n \neq l\}$不是正则的。

这里我们需要灵活地直接应用泵引理。选择具有$n = l + 1$或$n = l + 2$的符号串都不行，因为我们的对手总会找到一种分解方式使我们无法抽出不属于该语言的符号串（即：抽出的符号串中，a和b的个数相同）。我们必须更有创造力才行。我们令$n = m!$，$l = (m+1)!$。如果对手选出长度$k < n$的y（必须是只包含a），我们抽取i次，得到具有$m! + (i-1)k$个a的符号串。如果我们选择的i满足

$$m! + (i-1)k = (m+1)!$$

就可以推出和泵引理矛盾的结论。这个等式总是有可能成立的，因为

$$i = 1 + \frac{mm!}{k}$$

并且$k \leqslant m$。等式右侧是整数，因此我们成功地推出和泵引理相矛盾。

还有一种更好的解决方案。假设L是正则的，那么，根据定理4.1，$\overline{L}$和语言

$$L_1 = \overline{L} \cap L(a^*b^*)$$

也都是正则的。但是，我们知道$L_1 = \{a^nb^n : n \geqslant 0\}$不是正则的，所以$L$不是正则的。 □

下面几方面的原因使得泵引理的应用并不简单。首先，泵引理的声明复杂，所以应用时容易出错。然而，即使我们掌握了这种技术，仍然很难弄清楚具体如何使用它。泵引理就像一个具有复杂规则的游戏。了解规则是必需的，但是，仅仅知道这一点还不足以玩好这个游戏。你还需要有一个好的策略才能够赢得游戏。如果你可以正确应用泵引理来解决本书中的一些难题，那么就要祝贺你了。

习题

1. 证明下面版本的泵引理。如果L是正则的，那么存在一个m，满足每个长度大于m的$w \in L$都可以分解成$w = xyz$，其中 121

$$|yz| \leqslant m$$
$$|y| \geqslant 1$$

满足对于所有的i，xy^iz都属于L。

2. 下面是泵引理的一般化推广，定理4.8和习题1都是它的特例。证明这个普遍适用的泵引理。

   如果L是正则的，那么存在m满足对于任意足够长的$w \in L$及它的任意一种分解$w = u_1vu_2$，其中$u_1, u_2 \in \Sigma^*$，$|v| \geqslant m$，下面的结论都成立。中间的符号串v都可以写成$v = xyz$的形式，其中$|xy| \leqslant m$，$|y| \geqslant 1$，满足对于所有的$i = 0, 1, 2, \cdots$，都有$u_1xy^izu_2 \in L$。●

3. 证明语言$L = \{w : n_a(w) = n_b(w)\}$不是正则的。$L^*$是正则的吗？
4. 证明下列语言不是正则的：
   (a) $L = \{a^nb^la^k : k \geqslant n + l\}$ ●
   (b) $L = \{a^nb^la^k : k \neq n + l\}$
   (c) $L = \{a^nb^la^k : n = l$或$l \neq k\}$
   (d) $L = \{a^nb^l : n \leqslant l\}$
   (e) $L = \{w : n_a(w) \neq n_b(w)\}$ ●
   (f) $L = \{ww : w \in \{a, b\}^*\}$
   (g) $L = \{wwww^R : w \in \{a, b\}^*\}$
5. 判断下面定义在$\Sigma = \{a\}$上的语言是否是正则的：
   (a) $L = \{a^n : n \geqslant 2$，$n$是质数$\}$ ●
   (b) $L = \{a^n : n$不是质数$\}$
   (c) $L = \{a^n :$ 对于某个$k \geqslant 0$，$n = k^2\}$
   (d) $L = \{a^n :$ 对于某个$k \geqslant 0$，$n = 2^k\}$
   (e) $L = \{a^n : n$是两个质数的乘积$\}$
   (f) $L = \{a^n : n$要么是质数，要么是两个或更多质数的乘积$\}$
6. 直接应用泵引理来证明例4.12中的结论。
7. 证明下面的语言不是正则的：

$$L = \{a^nb^k : n > k\} \cup \{a^nb^k : n \neq k - 1\}$$

8. 证明下面的命题成立或不成立。

122 如果L_1和L_2是非正则语言，那么$L_1 \cup L_2$也是非正则的。●

9. 考察下列语言。对于每一个，判断它是否是正则的，并证明你的结论。

(a) $L = \{a^n b^l a^k : n + l + k > 5\}$ ●

(b) $L = \{a^n b^l a^k : n > 5, l > 3, k < l\}$ ●

(c) $L = \{a^n b^l : n/l$是整数$\}$

(d) $L = \{a^n b^l : n + l$是质数$\}$

(e) $L = \{a^n b^l : n < l < 2n\}$

(f) $L = \{a^n b^l : n \geqslant 100, l < 100\}$

(g) $L = \{a^n b^l : |n - l| = 2\}$

10. 下面的语言是正则的吗？

$$L = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^*, w_1 \neq w_2\}$$

11. 设L_1和L_2是正则语言。语言$L = \{w : w \in L_1, w^R \in L_2\}$一定是正则的吗？ ●

12. 直接将鸽巢原理应用到例4.8的语言中。

13. 下面的语言是正则的吗？

(a) $L = \{uww^R v : u, v, w \in \{a, b\}^+\}$ ●

★(b) $L = \{uww^R v : u, v, w \in \{a, b\}^+, |u| \geqslant |v|\}$ ●

14. 下面的语言是正则的吗？

$$L = \{ww^R v : v, w \in \{a, b\}^+\}$$

15. 设P是一个无穷可数的集合，并且和语言L_p中的每个$p \in P$相联系。包含所有L_p的最小集合为无穷集P上的并，表示成$\cup p \in_P L_P$。使用例子证明正则语言族在无穷并运算下不封闭。●

★16. 考察3.2节中的命题——和任何通用转移图相关的语言都是正则的。和这个图相关的语言是

$$L = \bigcup_{p \in P} L(r_p)$$

其中，P是图中所有的通道构成的集合。r_p是和通道p相关的表达式。通道的集合一般是无穷的，因此根据习题15，不会立刻有L是正则的结论。在这里证明：因为P的特殊本

123 质，无穷并也是正则的。

★17. 正则语言族在无穷交运算下是封闭的吗？ ●

18. 假设我们已知$L_1 \cup L_2$和L_1是正则的。是否可以因此得出结论L_2是正则的？

19. 在3.1节习题22的循环码语言中，设L是由所有描述矩形的$w \in \{u, r, l, d\}^*$构成的集合。证明

124 L不是正则语言。

第5章 上下文无关语言

在上一章中，我们发现不是所有的语言都是正则的。因为正则语言可以有效地描述某种简单模式，所以不必花很大气力去找那些非正则语言的例子。如果我们重新解释这些例子，程序设计语言的这些限制的相关性就会变得明显。如果在$L = \{a^n b^n : n \geqslant 0\}$中，我们用左括号代替$a$，右括号代替$b$，那么括号串（()）和（(()）)就属于$L$，但是（()不属于$L$。因此，这种语言描述了程序设计语言中一种简单的嵌套结构，这就意味着程序设计语言的某些属性需要某些正则语言不具有的性质。为了满足这个要求和其他的复杂特点，我们必须扩大语言族。这就导致了我们必须考虑上下文无关（context-free）语言及其文法。

我们首先在这章的开头定义了上下文无关文法和语言，并用几个简单的例子解释这个定义。接下来，我们考虑重要的成员关系问题，特别地，我们探讨如何判断一个给定的符号串是否可以由某个文法推出。大多数人在学习自然语言的时候都熟悉一种方式，就是通过文法推导解释一条语句，我们称之为分析（parsing）。分析是描述语句结构的一种方式。每当我们需要理解语义的时候，例如要把一种语言翻译成另一种语言时，分析是非常重要的。在计算机科学领域，分析和解释器、编译器及其他翻译程序都是相关的。 125

上下文无关语言恐怕是形式语言理论中最重要的一个方面，因为它可以应用到程序设计语言中。实际程序设计语言有很多特点可以通过使用上下文无关语言的方法来描述。形式语言理论告诉我们，上下文无关语言在程序设计语言的设计和有效编译器的构造方面都有重要的应用。我们在5.3节将简单接触这方面的问题。

5.1 上下文无关文法

正则文法的产生式有两方面的限制：左部必须是一个简单变量，并且右侧有固定的形式。为了创造出更强大的文法，我们必须放宽这些限制。保留左部的限制，但是允许右侧可以是任何形式，这样我们就获得了上下文无关文法。

定义5.1 *文法$G = (V, T, S, P)$称为上下文无关的（context-free），如果P中的产生式具有形式*

$$A \to x$$

其中，$A \in V$，$x \in (V \cup T)^*$。

语言L是上下文无关的，当且仅当存在一个上下文无关的文法G，满足$L = L(G)$。

每个正则文法都是上下文无关的，因此正则语言也是上下文无关的。但是，我们从简单例子如$\{a^n b^n\}$，可以知道存在非正则语言。我们在例1.11中指出这种语言可以由一种上下文无关文法生成。因此，我们得出正则语言族是上下文无关语言族的真子集。

上下文无关文法得名于句型中出现的产生式的左部变量可以在任何时候被替换，这种替换不依赖于句型中的其他符号（上下文）。这个特点就是只允许产生式左部有一个简单变量的理由。 126

5.1.1 上下文无关语言的例子

例5.1 文法$G=(\{S\},\{a,b\},S,P)$，产生式

$$S\to aSa$$
$$S\to bSb$$
$$S\to\lambda$$

这个文法是上下文无关的。它的一个典型推导是

$$S\Rightarrow aSa\Rightarrow aaSaa\Rightarrow aabSbaa\Rightarrow aabbaa$$

这样就可以清楚知道

$$L(G)=\{ww^R : w\in\{a,b\}^*\}$$

这个语言是上下文无关的，但是正如例4.8所示，它不是正则的。 □

例5.2 文法G具有产生式

$$S\to abB$$
$$A\to aaBb$$
$$B\to bbAa$$
$$A\to\lambda$$

这个文法是上下文无关的。我们把这个留给读者去完成，看看它生成的语言是不是

$$L(G)=\{ab(bbaa)^n bba(ba)^n : n\geqslant 0\}$$ □

上面两个有关文法的例子不仅是上下文无关的，而且还是线性的。正则和线性文法明显都是上下文无关的，但是上下文无关文法不一定是线性的。

127

例5.3 语言$L=\{a^n b^m : n\neq m\}$是上下文无关的。

为了证明这一点，我们需要一个产生这种语言的上下文无关文法。$n=m$的情况在例1.11中已经解决，我们可以利用它来构造结果。令$n>m$。我们首先生成一个具有相同个数a和b的符号串，然后在左部加入额外的a。这样的得到产生式为

$$S\to AS_1$$
$$S_1\to aS_1b|\lambda$$
$$A\to aA|a$$

对于$n<m$的情况，我们使用类似的推理，得到结果

$$S\to AS_1|S_1B$$
$$S_1\to aS_1b|\lambda$$
$$A\to aA|a$$
$$B\to bB|b$$

得到的这个文法是上下文无关的，因此L是上下文无关语言，但是，文法不是线性的。

这里给出的特殊文法形式，是为了方便解释。还有很多与之等价的上下文无关文法。事实上，这种语言还存在着简单的线性文法。在本节后面的习题25中，我们会要求你构造一个线性文法。 □

例5.4 考虑具有下列产生式的文法

$$S \to aSb|SS|\lambda$$

这也是个上下文无关文法，但不是线性文法。$L(G)$ 中有串$abaabb$，$aababb$和$ababab$。不难猜测和证明

$$L = \{w \in \{a, b\}^* : n_a(w) = n_b(w) \text{ 且} n_a(v) \geqslant n_b(v)\text{，这里，}v\text{是}w\text{的任意前缀}\} \tag{5-1}$$

如果我们分别用左右括号代替a和b，我们就可以清晰地看出语言L与程序设计语言的联系。语言L包括符号串（()）和 ()()()。事实上，这种语言就是通用程序设计语言所有正确嵌套的括号结构的集合。 128

对该文法也存在很多与其等价的文法。但是，和例5.3不同的是，很难看出它是否有线性文法。我们将在第8章回答这个问题。 □

5.1.2 最左推导和最右推导

在非线性的上下文无关文法中，推导可能包括由多个变量构成的句型。这种情况下，为了替换变量我们可以进行选择。以文法$G = (\{A, B, S\}, \{a, b\}, S, P)$ 为例，其产生式为

1. $S \to AB$
2. $A \to aaA$
3. $A \to \lambda$
4. $B \to Bb$
5. $B \to \lambda$

很容易看出这个文法产生的是语言$L(G) = \{a^{2n}b^m : n \geqslant 0, m \geqslant 0\}$。

考虑下面这两种推导:

$$S \overset{1}{\Rightarrow} AB \overset{2}{\Rightarrow} aaAB \overset{3}{\Rightarrow} aaB \overset{4}{\Rightarrow} aaBb \overset{5}{\Rightarrow} aab$$

和

$$S \overset{1}{\Rightarrow} AB \overset{4}{\Rightarrow} ABb \overset{2}{\Rightarrow} aaABb \overset{5}{\Rightarrow} aaAb \overset{3}{\Rightarrow} aab$$

为了表示使用的是哪个产生式，我们将产生式进行编号，并在符号⇒上标出相应的编号。由此我们可以看到，这两种推导不仅得到了同一个句子，而且使用了相同的产生式。它们不同的仅仅是产生式的使用顺序。为了去掉这些不相关的因素，我们经常要求变量要按照固定的顺序被替换。

定义5.2 *如果每次都替换句型中最左端的变量，那么这种推导称为最左的*（leftmost）。*如果每次都替换最右端的变量，那么就称这种推导为最右的*（rightmost）。 129

例5.5 考虑具有下面产生式的文法

$$S \to aAB$$
$$A \to bBb$$
$$B \to A|\lambda$$

那么

$$S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abAbB \Rightarrow abbBbbB \Rightarrow abbbbB \Rightarrow abbbb$$

是符号串*abbbb*的最左推导。这个符号串的最右推导为

$$S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abAb \Rightarrow abbBbb \Rightarrow abbbb$$ □

5.1.3 推导树

还有另一种表示推导过程的方式，叫做推导树（derivation tree），这种方式与产生式的使用顺序无关。推导树是一个有序树，其中结点标记的产生式是左部的符号，结点的子结点表示的是该产生式右部的符号。例如，图5-1表示下面产生式的推导树的那部分。

$$A \rightarrow abABc$$

在推导树中，标记有产生式左部变量的结点是根结点，标记有相应产生式右部的符号的结点是该根结点的子结点。以标有开始符的根结点开始，到标有终结符的叶结点结束。推导树可以表示推导过程中每个变量是如何被替换的。下面的定义给出了一个准确的表示。

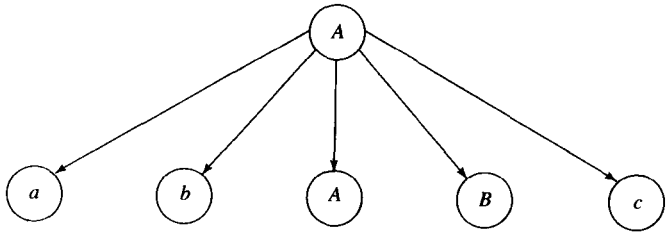


图 5-1

定义5.3 $G = (V, T, S, P)$ 是一个上下文无关文法。当且仅当下列性质成立时，一个有序树才能称为是G的推导树。

130

1. 根结点标记为S。
2. 每个叶结点有一个属于$T \cup \{\lambda\}$的标记。
3. 每个中间结点（不是叶结点的结点）都有一个属于V的标记。
4. 如果一个结点有标记$A \in V$，它的子结点标记（从左到右）为$a_1, a_2, \cdots, a_n$，那么，P中一定包含下面形式的产生式

$$A \rightarrow a_1 a_2 \cdots a_n$$

5. 标记为λ的叶结点没有兄弟，即：一个具有标记为λ的子结点的结点只能有一个子结点。

如果一棵树拥有性质3、4和5，但是1不一定成立，并且把2替换成：

2a. 每个叶结点都有属于$V \cup T \cup \{\lambda\}$的标记。

那么这棵树就是部分推导树（partial derivation tree）。

通过从左向右地读树上的叶结点获得的符号串，忽略遇到的任何λ，称为树的果（yield）。“从左到右”的术语描述可以给出一个准确的意义。果是终结符构成的符号串，是按照深度优先遍历树时遇到的字母顺序构成的，深度优先遍历总是先遍历最左部没有被遍历过的分支。

例5.6 已知文法G，它的产生式为

$$S \rightarrow aAB$$
$$A \rightarrow bBb$$

$$B \to A|\lambda$$

图5-2中的树是G的部分推导树，而图5-3中的树是推导树。符号串$abBbB$是第一棵树的果，它是G的句型。第二棵树的果$abbbb$，是$L(G)$ 的句子。 □ 131

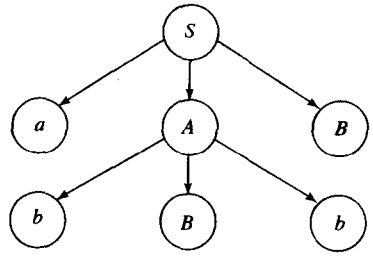


图 5-2

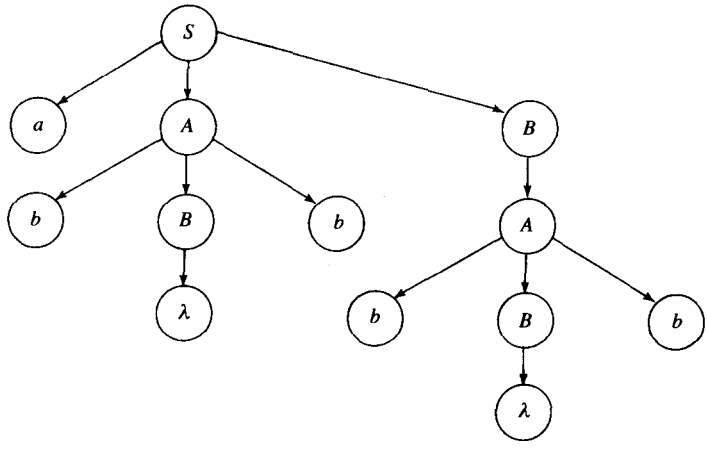


图 5-3

5.1.4 句型和推导树之间的关系

推导树给推导过程提供了一个清楚易懂的描述方式。与有穷自动机的转移图类似，这种清楚的描述对于证明有很大帮助。尽管这样，我们必须首先建立推导和推导树之间的联系。

定理5.1 设$G = (V, T, S, P)$ 是上下文无关文法。那么对于每个$w \in L(G)$，G中总存在一个推导树，它的果是w。反过来，任何推导树的果都属于语言$L(G)$。同样，如果t_G是G的任何一个部分推导树，且根结点标记为S，那么t_G的果就是G的一个句型。

证明：首先我们证明对于$L(G)$ 中的任何一个句型，都存在一个相应的部分推导树。我们对推导步数作归纳。作为归纳基础，我们注意对于一步推导产生的句型，上述结论都是成立的。因为$S \Rightarrow u$意味着存在产生式$S \to u$，根据定义5.3可以立即得到这个结论。

假设对于所有经过n步推导得出的句型，都存在一个对应的部分推导树。现在我们考虑任何需要$n+1$步推导才能产生的w，一定满足 132

$$S \overset{*}{\Rightarrow} xAy,\ x, y \in (V \cup T)^*,\ A \in V$$

这是经过n步推导得到的。而且

$$xAy \Rightarrow xa_1a_2\cdots a_my = w,\ a_1 \in V \cup T$$

因为根据归纳假设，存在一个具有果xAy的部分推导树。并且因为文法一定有产生式$A \to a_1a_2\cdots a_m$，我们按照这个文法扩展标有A的叶结点，就得到了具有果$xa_1a_2\cdots a_my = w$的部分推导树。根据归纳，我们可以说对于所有的句型，这个结论都成立。

类似地，我们可以证明每个部分推导树都表示某个句型。我们把这个证明留作习题。

因为推导树也是部分推导树，只不过它的叶子都是终结符。因此，$L(G)$ 中的每个句子都是G的某个推导树的果，并且所有推导树的果都属于$L(G)$。■

推导树指出了获取句子的过程中用到了哪些产生式，但是没有给出这些产生式应用的顺序。推导树能够表示任何推导，反映了顺序是无关的，这就拉近了我们和下一步讨论之间的距离。根据定义，任何$w \in L(G)$ 都有一个推导，但是我们不能说它也一定有最左推导和最右推导。然而，一旦有了推导树，我们就可以通过把树看成是由这种方法构造的，即总是由树的最左边变量开始扩展的，来得到最左推导。完成了这些细节，我们得到了一个不是很惊人的结论：任何$w \in L(G)$，既有最左推导，也有最右推导（细节详看本节末的习题24）。

习题

1. 完成例5.2中的证明，证明给出的语言是由该文法产生的。
2. 画出例5.1中推导对应的推导树。
3. 按照练习5.2中的文法，给出$w = abbbaabbaba$的推导树。使用推导树找到它的最左推导。
4. 证明例5.4中的文法确实产生了式（5-1）中描绘的语言。●
5. 例5.2中的语言是正则的吗？

133 6. 通过证明每个以S为根的部分推导树的果都是G中的句型，来完成定理5.1的证明。

7. 分别为下面语言构造上下文无关文法。（其中$n \geqslant 0, m \geqslant 0$）
   (a) $L = \{a^nb^m : n \leqslant m + 3\}$ ●
   (b) $L = \{a^nb^m : n \neq m - 1\}$
   (c) $L = \{a^nb^m : n \neq 2m\}$
   (d) $L = \{a^nb^m : 2n \leqslant m \leqslant 3n\}$ ●
   (e) $L = \{w \in \{a, b\}^* : n_a(w) \neq n_b(w)\}$
   (f) $L = \{w \in \{a, b\}^* : n_a(v) \geqslant n_b(v)$，其中$v$是$w$的任意前缀$\}$
   (g) $L = \{w \in \{a, b\}^* : n_a(w) = 2n_b(w) + 1\}$
8. 为下列语言构造上下文无关文法。（其中$n \geqslant 0, m \geqslant 0, k \geqslant 0$）
   (a) $L = \{a^nb^mc^k : n = m$或$m \leqslant k\}$ ●
   (b) $L = \{a^nb^mc^k : n = m$或$m \neq k\}$
   (c) $L = \{a^nb^mc^k : k = n + m\}$
   (d) $L = \{a^nb^mc^k : n + 2m = k\}$
   (e) $L = \{a^nb^mc^k : k = |n - m|\}$ ●
   (f) $L = \{w \in \{a, b, c\}^* : n_a(w) + n_b(w) \neq n_c(w)\}$
   (g) $L = \{a^nb^mc^k, k \neq n + m\}$
   (h) $L = \{a^nb^nc^k : k \geqslant 3\}$

9. 为$head(L)$ 构造上下文无关文法，L是上面习题7(a) 中的语言。$head$的定义参见4.1节习题18。

10. 为定义在$\Sigma = \{a, b\}$上的语言$L = \{a^n w w^R b^n : w \in \Sigma^*, n \geqslant 1\}$构造上下文无关文法。

★11. 给定L的上下文无关文法G。给出如何根据文法G，构造文法 $\hat{G}$，使得 $L(\hat{G}) = head(L)$。

12. 设$L = \{a^n b^n : n \geqslant 0\}$，

(a) 证明L^2是上下文无关的。

(b) 证明对于给定$k \geqslant 1$，L^k是上下文无关的。

(c) 证明$\bar{L}$和L^*是上下文无关的。

13. L_1是习题8 (a) 中的语言，L_2是习题8 (d) 中的语言，证明$L_1 \cup L_2$是上下文无关语言。

14. 证明下面语言是上下文无关的

$$L = \{uvwv^R : u, v, w \in \{a, b\}^+, |u| = |w| = 2\}$$

★15. 证明例5.1中的语言的补是上下文无关的。 134

16. 证明习题8 (c) 中的语言的补是上下文无关的。

17. 证明语言$L = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^+, w_1 \neq w_2^R\}$，它的字母表为$\Sigma = \{a, b, c\}$，是上下文无关的。

18. 给出符号串$aabbbb$的推导树，已知文法为

$$S \to AB \mid \lambda$$
$$A \to aB$$
$$B \to Sb$$

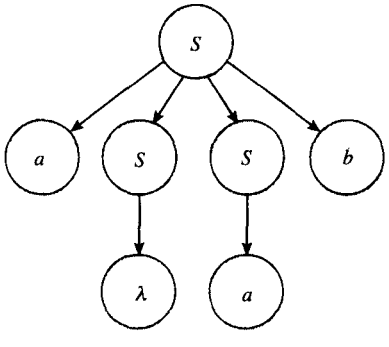
口头描述这个文法生成的语言。

19. 已知具有下面产生式的文法

$$S \to aaB$$
$$A \to bBb \mid \lambda$$
$$B \to Aa$$

证明符号串$aabbabba$不属于这个文法生成的语言。

20. 已知下面的推导树



这是符号串aab的推导树，构造它的简单文法G。然后再给出两个$L(G)$ 中的句子。

21. 定义包括两种括号()和[]的正确嵌套的括号结构。直观上，([])，([[ ]])[()]是正确的嵌套结构，而([)]或(([]都不是。根据你给出的定义，给出一个可以生成所有正确嵌套括号结构的

上下文无关文法。

22. 给出定义在字母表$\{a, b\}$上的所有正则表达式集合的上下文无关文法。●

23. 给出一个上下文无关文法，使之能够生成$T = \{a, b\}$, $V = \{A, B, C\}$的上下文无关文法的所有产生式。

24. 证明：如果G是上下文无关文法，那么任意一个$w \in L(G)$存在最左推导和最右推导。给出一个根据推导树进行推导的算法。

135

25. 为例5.3中的语言构造线性文法。

26. 设$G = (V, T, S, P)$是上下文无关文法，满足它的每个产生式都具有$A \to v$的形式，其中，$|v| = k > 1$。证明任何$w \in L(G)$的推导树，有高度为h，满足

$$\log_k |w| \leqslant h \leqslant \frac{(|w|-1)}{k-1}$$

5.2 分析和二义性

我们一直以来关注的是文法的一般性质。给定文法G，我们学习用G可以推导出的符号串的集合。在实际应用中，我们也关注文法的分析性质：给定终结符构成的符号串w，我们想要知道w是否属于$L(G)$。如果属于，那么我们想要得到w的推导过程。成员资格判定算法可以告诉我们w是否属于$L(G)$。术语分析（parsing）描述的是寻找$w \in L(G)$的推导过程中使用的一系列产生式。

5.2.1 分析和成员资格判定

给定$L(G)$中的符号串w，我们可以按照下面这种清晰的方式分析它。我们系统地构造所有可能的（比如：最左）推导，看看推导出的结果是否能够匹配w。特别地，第一回合，我们首先看所有具有下面形式的产生式

$$S \to x$$

从而发现所有可以由S一步推出的x。如果所有这些结果都不能匹配w，那么下一个回合我们就把所有可以应用的产生式都用在每个x的最左边的变量上。这样我们就获得了一个句型的集合，或许其中有的句型就能推出w。在每个后续的回合中，我们考察所有最左边的变量，应用所有可以应用的产生式。如果句型不能推出w，就抛弃它。但是，一般来说，我们每回合都会获得一些可能的句型。第一回合后，我们获得了使用一个简单产生式而生成的句型；第二回合后，我们获得了两步推导可以得到的句型……如果$w \in L(G)$，那么它一定有有限长度的最左推导。因此，这种方法最后会给出w的最左推导。

以后再提到这个分析方法，我们就称它为穷举搜索分析（exhaustive search parsing）。它是自顶向下分析（top-down parsing）的一种形式，我们可以把它看成是自根部向下构造推导树。

136

例5.7 已知文法

$$S \to SS|aSb|bSa|\lambda$$

和符号串$w = aabb$。第一回合，我们得到

$$1.\ S \Rightarrow SS$$
$$2.\ S \Rightarrow aSb$$
$$3.\ S \Rightarrow bSa$$
$$4.\ S \Rightarrow \lambda$$

根据结果，显然要去掉最后两个推导。第二回合，产生如下句型

$$S \Rightarrow SS \Rightarrow SSS$$
$$S \Rightarrow SS \Rightarrow aSbS$$
$$S \Rightarrow SS \Rightarrow bSaS$$
$$S \Rightarrow SS \Rightarrow S$$

这些句型是通过把句型1中最左边的S尽可能地替换掉而得到的。类似地，根据句型2，我们又得到几个句型

$$S \Rightarrow aSb \Rightarrow aSSb$$
$$S \Rightarrow aSb \Rightarrow aaSbb$$
$$S \Rightarrow aSb \Rightarrow abSab$$
$$S \Rightarrow aSb \Rightarrow ab$$

再去掉几个句型。下一个回合，我们从推导中得到最后的实际目标符号串

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

因此，$aabb$属于该文法在这种情况下生成的语言。 □

穷举搜索分析存在严重的缺陷。最明显的缺陷是过程冗长，所以不适用于需要使用高效率分析的地方。但是即使效率不是首要的因素，还存在其他的原因使它可能不被使用。当使用这种方法分析$w \in L(G)$ 时，若得不到属于$L(G)$ 中的符号串可能就永远都不会停止。这点对于前面的例子是显然的。对于$w = abb$，这种方法会不断地产生试验性的句型，除非我们使用某种方式使它停下来。 137

如果我们限制文法的形式，那么穷举搜索分析的不终止性问题就相对容易解决。如果考察例5.7，我们就可以看出产生式$S \to \lambda$带来的麻烦。使用这个产生式可以减少推导出的后继句型的长度，因此我们就不容易判断出什么时候停止推导。如果没有这样的产生式，那么我们的困难就少些。事实上，我们想要消除两种形式的产生式：$A \to \lambda$和$A \to B$。在下章中我们将看到，这个限制并不影响最终的文法表达能力。

例5.8 文法

$$S \to SS|aSb|bSa|ab|ba$$

满足给定的需求。它能够生成例5.7中提到的语言，除了空串。

已知任何$w \in \{a, b\}^+$，穷举搜索分析方法总会在不超过$|w|$回合时停止。这很明显，因为每回合句型的长度至少增长1。$|w|$回合后，我们或者产生一个分析，或者我们可以知道$w \notin L(G)$。 □

这个例子的想法可以一般化，从而生成上下文无关语言的定理。

定理5.2 假设$G = (V, T, S, P)$ 是上下文无关文法，并且不具备下面形式的产生式

$$A \to \lambda$$

或

$$A \to B$$

其中，$A, B \in V$。那么由穷举搜索分析法可以生成一个算法，它能完成：对于任何$w \in \Sigma^*$，要么生成w的一种分析，要么告诉我们不可能存在分析。

证明：对于每个句型，考虑它的长度和终结符的个数。推导的每一步中，上述两个因素至少有一个会增长。因为句型的长度和终结符的个数都不能超过$|w|$，所以推导过程不能超过$2|w|$个回合。那么至多经过这么多回合后，我们要么会得到一个成功的分析，要么会知道文法不能产生w。■

138

穷举搜索分析法从理论上保证了分析总是可以进行的。但是它的实践意义是有限的，因为通过这种分析得到的句型的长度可能特别大。不同的情况到底能产生多少句型，没有人能给出准确的一般结果，但是我们知道它的严格上限。如果我们只限制为最左推导，那么，第一回合后，我们得到的句型不超过$|P|$个，第二回合后，得到的句型个数不超过个$|P|^2$……在定理5.2的证明中，我们可以观察到分析包括的回合数不可能超过$2|w|$，因此，获得的句型数量不超过

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} \tag{5-2}$$

这就意味着穷举搜索分析的工作量是符号串长度的指数数量级，使得这个方法的代价太大。当然，式（5-2）只是一个范围，句型的数量通常要小得多。但是，实际的观察结果表明穷举搜索分析在大多数情况下效率是很低的。

上下文无关文法的更有效的分析方法的构造是一件复杂的事情，它属于编译器这门课程。我们在这里就不深究它了，除非对某些孤立的结论。

定理5.3 每个上下文无关文法都存在一个算法，它对于任何$w \in L(G)$，都能以和$|w|^3$成正比的步数进行分析。

有几种方法可以解决这个问题，但是它们都太复杂了，以至于不先证明某些额外的结论我们就无法描述它们。在6.3节中，我们将再次简要地谈及这个问题。有关这个问题的细节参见Harrison 1978、Hopcroft和Ullman 1979。这里不深究它们的细节的原因在于这些算法也并不令人满意。工作量与符号串长度的三次幂成比例增加的方法，虽然比指数的方法好些，但是也不是太有效，以此为基础的编译器即使对于中等长度的程序也要花费大量的时间分析。我们想要的是和符号串长度成正比的时间复杂度的分析法。我们把这种方法称为线性时间（linear time）分析算法。我们还不知道时间复杂度为线性时间的上下文无关文法的一般分析法，但是在一些有限制的特殊例子中可以找到这样的算法。

139

定义5.4 上下文无关文法$G = (V, T, S, P)$是简单文法（simple grammar或s-grammar），如果它的产生式形式都是

$$A \to ax$$

其中，$A \in V$，$a \in T, x \in V^*$，并且，任何对（A, a）在P中至多出现一次。

例5.9 文法

$$S \to aS|bSS|c$$

是简单文法。文法

$$S \to aS|bSS|aSS|c$$

不是简单文法，因为对（S, a）既出现在$S \to aS$中，又出现在$S \to aSS$中。 □

尽管简单文法相当严格，但是它们是有意义的。我们在下一节将会看到，一般程序设计语言的许多特点都可以使用简单文法来描述。

如果G是简单文法，那么$L(G)$中的任何符号串w都可以在与$|w|$成正比的时间复杂度范围内分析完。为了弄清楚这一点，考虑穷举搜索法和符号串$w = a_1a_2\cdots a_n$。因为左部为S且右部由a_1开头的产生式至多有一个，推导过程首先为

$$S \Rightarrow a_1A_1A_2\cdots A_m$$

然后，替换变量A_1，但是又因为至多只有一个选择，我们就得到

$$S \overset{*}{\Rightarrow} a_1a_2B_1B_2\cdots A_2\cdots A_m$$

由此，我们可知道每步都产生一个终结符号，于是，整个过程一定会在$|w|$步内完成。 140

5.2.2 文法和语言的二义性

根据我们的命题，可以知道任何给定的$w \in L(G)$，穷举搜索分析一定会产生一棵w的推导树。我们说“一棵”，而不是特指某棵的原因在于可能存在大量不同的推导树。这种情况称为二义性（ambiguity）。

定义5.5 *如果对某个$w \in L(G)$至少存在两棵不同的推导树，那么上下文无关文法G称作是二义性的（ambiguous）。也就是说，二义性意味着存在两个或两个以上的最左或最右推导。*

例5.10 例5.4中的文法，产生式为$S \to aSb|SS|\lambda$，有二义性。因为句子$aabb$有两棵推导树，如图5-4所示。 □

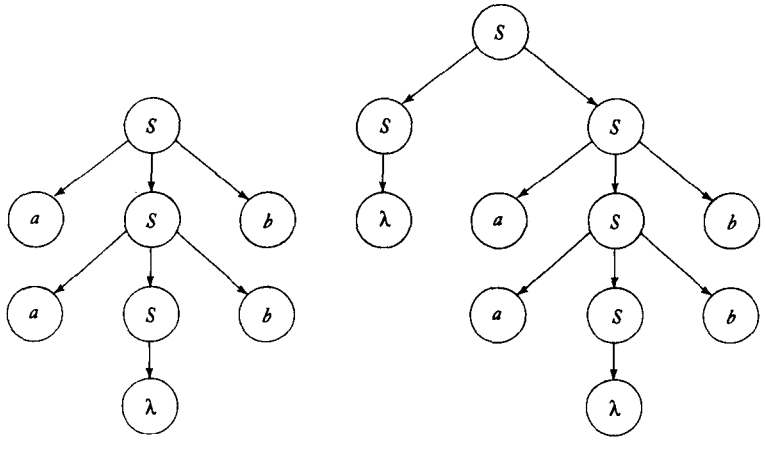


图 5-4

二义性是诸多自然语言的共同特色，自然语言容许二义性并以不同的方法加以解决。在程序设计语言中，每条语句只应该有一种解释，要去除二义性。我们通常通过重写一个等价的、无二义性的文法来达到这个目的。 141

例5.11 已知文法$G = (V, T, E, P)$，其中

$$V = \{E, I\}$$
$$T = \{a, b, c, +, *, (,)\}$$

产生式为

$$E \to I$$
$$E \to E + E$$
$$E \to E * E$$
$$E \to (E)$$
$$I \to a|b|c$$

符号串 $(a + b)*c$和$a*b + c$属于$L(G)$。很容易看出这个文法生成了C语言和类Pascal语言的数学表达式的严格集合。这个文法是有二义性的。例如，符号串$a + b*c$有两棵推导树，如图5-5所示。 □

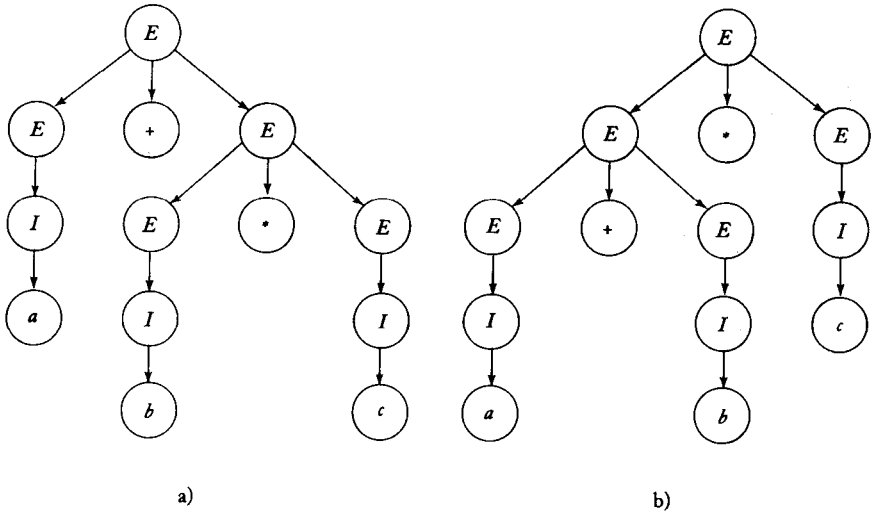


图5-5 $a + b*c$的两个推导树

解决这个二义性的一种方法是，像程序设计手册一样，建立操作符*和+之间的优先级。因为通常*比+优先级要高，因此我们认为图5-5a是正确的分析，这就意味着在执行加法前，先要验证子表达式$b*c$。然而，这个解出现在文法范围之外。最好重写这个文法，使得只存在一种分析。

142

例5.12 重写例5.11中的文法，我们引入新的变量，让V为$\{E, T, F, I\}$，并且将产生式变成

$$E \to T$$
$$T \to F$$
$$F \to I$$
$$E \to E + T$$
$$T \to T*F$$
$$F \to (E)$$
$$I \to a|b|c$$

句子$a + b*c$的推导树如图5-6所示。这个符号串不存在其他的推导树。于是文法是无二义性的。它和习题5.11中的文法等价。证明这些命题在特殊情况下成立是不难的。但是，一般情况下，一个给定的上下文无关文法是否是二义性的，或者两个上下文无关文法是否等价，这两个问题都非常难回答。事实上，在后面我们会证明不存在通用的算法来解决这些问题。 □

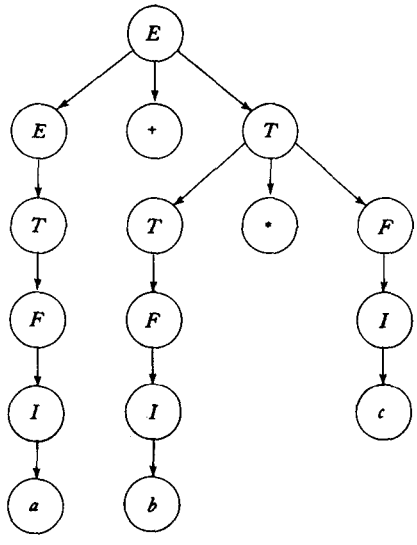


图 5-6

在前面的例子中，文法的二义性可以通过找一个等价的无二义性的文法来消除。但是某些情况下无法这样做，因为语言本身具有二义性。 143

定义5.6 *如果上下文无关语言L存在一个无二义性文法，那么L就是无二义性的。如果每个生成L的文法都是二义性的，那么这个语言是固有二义性的*（inherently ambiguous）。

表示一个具有固有二义性的语言有点困难。我们能做的仅仅是给出一个例子说明，说明它是固有二义性的。

例5.13 语言

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\}$$

其中，n和m是非负整数。这是一个具有固有二义性的上下文无关语言。

容易证明语言L是上下文无关的。注意

$$L = L_1 \cup L_2$$

其中，L_1是由

$$S_1 \to S_1 c | A$$
$$A \to aAb | \lambda$$

生成的。而L_2是通过一个类似的文法给出的，它的开始符为S_2，产生式为

$$S_2 \to aS_2 | B$$
$$B \to bBc | \lambda$$

于是，L可以由这两个文法合并生成，需要一个附加的产生式

$$S \to S_1 | S_2$$

这个文法是二义性的，因为符号串$a^n b^n c^n$有两棵推导树，一个开始时使用$S \Rightarrow S_1$，另一个开始时使用$S \Rightarrow S_2$。当然不能因此就说L是固有二义性的，因为可能存在某个非二义性的文法可以产生它。但是在某方面，L_1和L_2存在着冲突的需求，第一个对a和b的数目加限制，而第二个对b和c的数目加限制。尝试几次就会使你确信，不能把这两个需求合并在同一个规则集合里，唯一地满足$n = m$。但是这个命题的严格证明是很需要技巧的，一种证明方法参见Harrison 1978。 □

144

习题

1. 为$L(aaa^*b + b)$ 构造简单文法。
2. 为$L = \{a^n b^n : n \geqslant 1\}$构造简单文法。●
3. 为$L = \{a^n b^{n+1} : n \geqslant 2\}$构造简单文法。
4. 证明每个简单文法都是无二义性的。
5. 已知$G = (V, T, S, P)$ 是简单文法。使用$|V|$和$|T|$，给出一个描述$|P|$的最大值的表达式。
6. 证明下面的文法是有二义性的

$$S \to AB | aaB$$
$$A \to a | Aa$$
$$B \to b$$ ●

7. 构造一个和习题6等价的无二义性的文法。
8. 使用例5.12，给出$(((a + b)*c)) + a + b$的推导树。
9. 证明正则语言不可能是固有二义性的。●
10. 给出一个生成$\Sigma = \{a, b\}$上的所有正则表达式集合的无二义性文法。
11. 正则文法可能是二义性的吗？
12. 证明语言$L = \{ww^R : w \in \{a, b\}^*\}$不是固有二义性的。
13. 证明下面文法是有二义性的

$$S \to aSbS | bSaS | \lambda$$

14. 证明例5.4中的文法是有二义性的，但是它表示的语言是无二义性的。●
15. 证明例1.13中的文法是有二义性的。
16. 证明例5.5中的文法是无二义性的。
17. 使用穷举搜索分析法，按照例5.5中的文法，分析符号串$abbbbbb$。一般地，分析这种语言中的任何符号串w需要多少回合？

145

18. 证明例1.14中的文法是无二义性的。
19. 证明下面结论，上下文无关文法$G = (V, T, S, P)$ 中，每个$A \in V$最多在一个产生式的左侧出现一次。那么G是无二义性的。
20. 构造一个和例5.5中的文法等价的文法，使它满足定理5.2的要求。●

5.3 上下文无关文法和程序设计语言

形式语言理论的一个最重要的应用是程序设计语言的定义和它们的解释器、编译器的构造。这里的基本问题是准确地定义程序设计语言，并使用这个定义作为书写有效、可靠的翻译程序的起点。正则语言和上下文无关语言都是能够达到这个目的重要方法。正如我们将看到的，识别程序设计语言中的简单模式时，我们可以使用正则语言，但是就像我们在本章的介绍中提到的，我们需要上下文无关语言来给更复杂的情况建模。

和其他语言一样，我们可以根据文法来定义程序设计语言。通常，我们习惯使用一种特殊的文法来定义语言，这种文法称为*巴克斯-诺尔形式*（Backus-Naur form, BNF），这种形式在本质上和我们使用过的表示方式相同，但是表现形式不同。在BNF中，变量放在尖括号中表示，终结符不用特殊标记。BNF也使用辅助符号，比如 | 。因此，例5.12中的文法使用BNF表示为

<expression>::=<term>|<expression> + <term>

<term>::=<factor>|<term>*<factor>

等等。符号+和*是终结符。符号|在我们的表示中用来表示或的关系，同时用::=代替→。程序设计语言的BNF描述倾向于使用更多的含义明确的变量标识符，从而清楚地表示产生式的含义。但是，另一方面这两种表示方式之间却又不存在重大差异。

类Pascal程序设计语言的很多部分由于使用了上下文无关文法的严格方式表示，所以容易受定义的影响。例如，Pascal语言中的if-then-else语句可以定义为

<if_statement>::= *if* <expression><then_clause><else_clause>

146

这里关键词*if*是终结符。所有其他的术语都是变量，必须定义这些变量。如果我们根据定义5.4检查这个，我们可以看出它看起来就像个简单文法的产生式。左边的变量<if_statement>总是和右边的终结符*if*相关。由于这个原因，这样的语句容易有效地分析。这里我们就知道了为什么要在程序设计语言中使用关键字。关键字不仅仅为程序阅读者提供可视化的结构信息，而且使得构造编译器的工作变得大为容易。

不幸的是，并不是一个典型程序设计语言的所有特性都可以被简单文法表示的。上面<expression>的规则就不是这个类型的，因此，分析变得不太明显。于是就出现了一个问题：我们允许什么样的文法规则，什么样的文法规则可以有效地分析。在编译器领域中，进一步扩展所谓的LL文法和LR文法，这两种文法可以表达程序设计语言的某些不明显的特性。但是它们可以在线性时间内完成分析。这不是个简单的事情，这个问题涉及的很多内容都不属于我们要探讨的范围。我们将在第6章简单地接触这方面的问题，但是从我们的目的出发已足够了：认识到这样的文法存在并且已经被广泛地研究。

与此相联系的是，二义性问题承担了另一个重要意义。程序设计语言的规约必须是无二义性的，否则，被不同的编译器编译，或运行在不同系统上的同一个程序，可能产生非常不同的结果。如同例5.11所示，一个天真的方法很容易把二义性引入到文法中。为了避免这样的错误，我们必须能够识别和消除二义性。一个相关的问题是一种语言是否是固有二义性的。为了实现这个目的，我们需要的是能够在上下文无关文法中识别和消除二义性，并且判断一个上下文无关文法是否是固有二义性的算法。不幸的是，这都是十分困难的问题，在后面，

大家会看到对于通常情况下的这种问题是不可能解决的。

程序设计语言可以通过上下文无关文法建模的特点通常是和它的语法（syntax）相关的。无论如何，它通常是这样的情况：不是所有语法正确的程序都可以被接受的。对于Pascal而言，通常BNF的定义允许按照下面这样的构造

$$var\ x, y : real$$
$$x, z : integer$$

或

$$var\ x : integer$$
$$x := 3.2$$

147 这两种构造都不被Pascal编译器接受，因为它们都违反了其他的限制，比如“整型变量不能被赋值为实数”。这种规则是程序设计语言语义的一部分，因为它和我们如何解释一个具体结构的意义相关。

程序设计语言的语义是一个复杂的问题。程序设计语言语义的规约不像上下文无关文法这样优雅而简洁。结果，一些语义特性的定义简单、或有二义性。无论从程序设计语言方面，还是从形式语言理论方面而言，定义程序设计语言语义都还是一个正在研究中的问题。虽然已经提出了几种解决方法，但是它们都不能够被普遍接受，而且不能像上下文无关语言中的语法一样成功地给出语义的定义。

习题

1. 给出Pascal语言中的<*expression*>完整定义。
2. 给出Pascal语言中while语句的BNF定义。（不用定义通用的<*statement*>概念）
3. 给出一个BNF文法描述Pascal程序和它的子程序之间的关系。
4. 给出FORTRAN语言中do语句的BNF定义。
5. 给出定义C语言中的if-else语句的正确形式。

148 6. 给出不能够用上下文无关文法描述的C语言的特性的例子。

第6章 上下文无关文法的化简与范式

在深入学习上下文无关语言之前，我们首先需要关注一些技术方面的事情。在上下文无关文法的定义中，它没有对产生式右部做任何限制，而这种完全自由的形式并不总是必要的，事实上，在某些情况下，它甚至会对文法产生不良影响。从定理5.2即可以看出，文法的某些限制形式在应用中是方便的；而且通过消除诸如$A\to\lambda$和$A\to B$这样的产生式也可以简化文法，因而，上下文无关文法的限制形式是必要的。因此，我们就有必要关注文法的转换方法，通过它们可以将任意的上下文无关文法转换成某种等价的受限形式。在本章中，我们将学习几种这样的转换方法和替换规则，在以后的讨论中将会用到它们。

针对上下文无关文法，我们也将学习一些相应的*范式*（normal form）。范式指的是一种受限制但又具备足够表达能力的文法形式，任意文法都可以表示成相应的等价范式。本章我们将介绍两种有用的范式：*乔姆斯基范式*（Chomsky normal form）与*格里巴克范式*（Greibach normal form）。它们不仅在理论上具有重要意义，同时在实际中也被广泛应用。6.3节给出了乔姆斯基范式在语法分析中的直接应用。

由于本章大多数证明过程可操作性强而缺乏直观性，因此，在某种程度上，这些内容将会显得比较枯燥。但就我们学习形式语言的目的而言，技术方面的东西相对来说是不重要的，可随意阅读。但最终得到的各种结论却相当重要，在我们以后的学习中，它们会被反复用到。

6.1 文法变换方法

首先，就一般的文法与语言而言，它们都会存在一个麻烦的问题：空串现象。而且在很多的定理及其证明中，空串往往表现为一种特殊情况，需要予以特别考虑。因此，为了化简，我们可以选择消除空串，使得待考察的语言中不包含λ。从下面的讨论中就可以看出，消除空串不会使文法失去一般性。设L是某一上下文无关语言，而$G=(V,T,S,P)$是表示$L-\{\lambda\}$的上下文无关文法。通过在V中增加变量S_0，并使它作为开始变量，同时在P中增加产生式

$$S_0\to S|\lambda$$

这样得到的新文法产生的语言即为L。因此，我们可以得到一个重要结论：$L-\{\lambda\}$总是可以转换为L。同样，对于任意的上下文无关文法G，存在一种方法可以得到$\hat{G}$并使得$L(\hat{G})=L(G)-\{\lambda\}$（见本节习题13）。基于这样的结论，在实际应用中，我们认为包含λ的上下文无关文法与不包含λ的上下文无关文法是没有区别的。在本章的后续内容中，除非特别指出，否则我们讨论的都是不包含λ的上下文无关语言。

6.1.1 一个有用的代入规则

通过代入方法，很多规则能够用于生成等价文法。在这里我们将给出其中一种，它可以用于化简文法。所谓*化简*（simplification），我们虽然无法给出它的准确定义，但仍将使用它。

在这里，我们认为化简就是对某些不必要产生式的消除，但是在实际的化简过程中，它并不必然导致产生式数目的减少。

定理6.1 设$G=(V,T,S,P)$是一个上下文无关文法，P中包含一个形如

$$A\rightarrow x_1Bx_2$$

的产生式，其中A与B是不同的变量并且

150

$$B\rightarrow y_1|y_2|\cdots|y_n$$

是P中所有以B为左部的产生式集合。而对于上下文无关文法$\hat{G}=(V,T,S,\hat{P})$，其中$\hat{P}$是通过删除P中的产生式

$$A\rightarrow x_1Bx_2 \tag{6-1}$$

并增加产生式

$$A\rightarrow x_1y_1x_2|x_1y_2x_2|\cdots|x_1y_nx_2$$

而得到的。则

$$L(\hat{G})=L(G)$$

证明：假定$w\in L(G)$，则可得

$$S\overset{*}{\Rightarrow}_G w$$

作为推导符号⇒下标的G用于区分使用不同文法进行的推导。如果该推导没有使用式（6-1），则

$$S\overset{*}{\Rightarrow}_{\hat{G}} w$$

显然成立。

否则考虑在推导中第一次使用式（6-1）的情况。由于引入的变量B最终必须被替换，因此，如果我们假定在引入B之后就立即对其进行替换（见本节习题17），而这么做并没有丢失任何信息。因此可得

$$S\overset{*}{\Rightarrow}_G u_1Au_2\Rightarrow_G u_1x_1Bx_2u_2\Rightarrow_G u_1x_1y_jx_2u_2$$

而在文法$\hat{G}$中我们可得

$$S\overset{*}{\Rightarrow}_{\hat{G}} u_1Au_2\Rightarrow_{\hat{G}} u_1x_1y_jx_2u_2$$

这样通过文法G与$\hat{G}$可以得到相同的句型。如果在后续推导中重复使用了式（6-1），则可以采用同样的证明方法。依此类推，对使用该产生式的次数进行归纳，可以得到

$$S\overset{*}{\Rightarrow}_{\hat{G}} w$$

因此，如果$w\in L(G)$，则$w\in L(\hat{G})$。

类似地，可以证明如果$w\in L(\hat{G})$，则$w\in L(G)$。证毕。■

定理6.1是一个简单而又直观的代入规则。它通过将所有产生式中的B替换成B一步推导可

151 得到的符号串，而得到一组新的产生式作为文法的组成部分，而原产生式$A\rightarrow x_1Bx_2$可以从中删除。该结论的一个必要条件是A与B是不同的变量，对$A=B$的情况，在本节习题22与习题23中将部分地进行阐述。

例6.1 考虑文法$G=(\{A,B\},\{a,b,c\},A,P)$，其中$P$包含产生式

$$A\to a|aaA|abBc$$
$$B\to abbA|b$$

通过对变量B运用本节的代入规则，我们将得到文法$\hat{G}$，它的产生式如下

$$A\to a|aaA|ababbAc|abbc$$
$$B\to abbA|b$$

根据定理可知，文法$\hat{G}$与G是等价的。对于串$aaabbc$，它在G中存在推导

$$A\Rightarrow aaA\Rightarrow aaabBc\Rightarrow aaabbc$$

而在$\hat{G}$中的推导为

$$A\Rightarrow aaA\Rightarrow aaabbc$$

我们注意到，在这种情况下，变量B和与之相关的产生式仍在文法中，虽然它们在任何推导中都不再起作用。在后面的讨论中我们将会看到这种产生式是怎样从文法中删除的。 □

6.1.2 删除无用产生式

我们总是希望将在任何推导中不起作用的产生式从文法中删除。例如在某个文法中，它的产生式集合如下:

$$S\to aSb|\lambda|A$$
$$A\to aA$$

可以看出，产生式$S\to A$是无用的，因为不能从A推导出一个终结符串。虽然A可以出现在从S推导出的某个串中，但是它最终无法生成一个句子。删除这种形式的产生式，对语言没有任何影响，并且从定义上来看，这也是对文法的化简。 152

定义6.1 设$G=(V,T,S,P)$是一个上下文无关文法，如果变量$A\in V$是有用的（useful），则当且仅当至少存在一个$w\in L(G)$使得

$$S\overset{*}{\Rightarrow}xAy\overset{*}{\Rightarrow}w \tag{6-2}$$

其中x，y属于$(V\cup T)^*$。总而言之，某个变量是有用的当且仅当它至少在一个句子的推导中出现，而对于不出现在任何句子推导中的变量，我们称之为无用的（useless）。如果某个产生式包含无用变量，则该产生式是无用的。

例6.2 一个变量无用的可能原因是无法从其得到一个终结符串，上面讨论的就是这种情况。而通过下面的文法，我们将看到一个变量是无用的另外一个可能原因。在该文法中，开始符为S，它的产生式如下

$$S\to A$$
$$A\to aA|\lambda$$
$$B\to bA$$

其中变量B是无用的，因而产生式$B\to bA$也是无用的。虽然能够从B推导得到一个终结符串，但是我们无法得到$S\overset{*}{\Rightarrow}xBy$。 □

上述例子说明了一个变量无用的两个原因：或者无法从开始符达到它，或者无法从其推导出一个终结符串。去除无用变量与产生式的过程就是在认识到这样两种情况的基础上进行的。在给出一般情形以及相应的定理之前，我们将给出另外一个例子。

例6.3 消除文法$G = (V, T, S, P)$中的无用符号与产生式，其中$V = \{S, A, B, C\}$，$T = \{a, b\}$，产生式P如下：

$$S \to aS|A|C$$
$$A \to a$$
$$B \to aa$$
$$C \to aCb$$

153

首先我们将标识出可以得到终结符串的变量集合。由于$A \to a$且$B \to aa$，因此A，B属于该集合。同理由于$S \Rightarrow A \Rightarrow a$，因此$S$也属于该集合。但对于$C$，由于无法得到这样的一个结论，因而它是无用的。通过删除C以及与其相关的产生式，可以得到文法G_1，其中变量$V_1 = \{S, A, B\}$，终结符集$T = \{a\}$，产生式集合为

$$S \to aS|A$$
$$A \to a$$
$$B \to aa$$

接下来，我们将消除无法从开始变量达到的变量。针对这种情况，我们可以通过给出变量依赖图（dependency graph）来完成。依赖图作为对复杂关系的一种可视化表示而被广泛应用。对于上下文无关文法的依赖图，其中的顶点表示变量，而从顶点C到D之间存在边当且仅当存在如下形式的产生式

$$C \to xDy$$

图6-1所示为V_1的依赖图。如果一个变量是有用的，仅当在依赖图中存在一条路径，它从以S为标记的顶点出发能够到达以该变量标记的顶点。在我们的例子中，从图6-1可以看出B是无用的。通过删除它以及受到其影响的产生式与终结符，我们将得到最终文法$\hat{G} = (\hat{V}, \hat{T}, S, \hat{P})$，其中$\hat{V} = \{S, A\}$，$\hat{T} = \{a\}$，产生式$\hat{P}$如下

$$S \to aS|A$$
$$A \to a$$

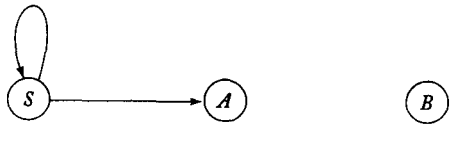


图 6-1

根据上述的形式化过程可以得到一个一般性的构造方法以及相应的定理。 □

定理6.2 *设$G = (V, T, S, P)$是一个上下文无关文法，则存在一个与之等价的文法$\hat{G} = (\hat{V}, \hat{T}, S, \hat{P})$，它不包含任何的无用符号或产生式。*

154

证明：文法$\hat{G}$能够通过一个算法根据G生成，该算法包含两个部分。在第一部分中，将构造一

个中间形式的文法$G_1 = (V_1, T_2, S, P_1)$，使得对于V_1中的每一个变量A，都能够找到如下形式的推导

$$A \overset{*}{\Rightarrow} w \in T^*$$

第一部分的算法步骤如下:

1. 置V_1为空。

2. 重复下面的步骤直到再也没有变量能够加到V_1中为止。

对于任一$A \in V$，如果P中存在形如

$$A \to x_1 x_2 \cdots x_n，其中所有x_i都在V_1 \cup T中$$

的产生式，则将A加入到V_1中。

3. 对于P中的每一个产生式，如果它的符号都在（$V_1 \cup T$）中，则将其加入到P_1中。

显然，这一过程总会终止。同样可以看出，如果$A \in V_1$，则$A \overset{*}{\Rightarrow} w \in T^*$将是$G_1$中的一个可能的推导。则剩下的问题就是是否对存在诸如$A \overset{*}{\Rightarrow} w = ab\cdots$推导出的每一个$A$，在本过程结束之前都将被加入到$V_1$中。为了说明这一问题，可以考虑相应的$A$以及与该推导相关的部分推导树（如图6-2）。在$k$层上，只有终结符，因此$k-1$层的每一个变量$A_i$都将通过算法的第2步在第一轮中被加入到$V_1$，$k-2$层的每个变量将通过算法的第2步在第二轮中被加入$V_1$，而$k-3$层的所有变量都将通过第2步的第三轮被加入，依此类推。只要在树中还有变量没有加入被到V_1中，该算法就不会终止。因此，所有这种形式的A最终都会被加入到V_1中。

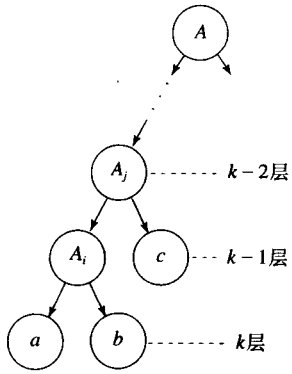


图 6-2

在构造方法的第二部分中，我们将从G_1生成结果文法$\hat{G}$。基于G_1的变量依赖图，可以得到所有不能从S到达的变量。通过删除这些变量以及与其相关的产生式，同时也删除不出现在有用产生式中的终结符。这样得到的结果就是文法$\hat{G} = (\hat{V}, \hat{T}, S, \hat{P})$。 155

根据构造过程，可以看出$\hat{G}$中将不包含任何的无用符号与产生式。针对任意$w \in L(G)$，我们有推导

$$S \overset{*}{\Rightarrow} xAy \overset{*}{\Rightarrow} w$$

由于$\hat{G}$的构造过程保留了A以及所有相关的产生式，因而我们有充分的条件进行如下推导

$$S \overset{*}{\Rightarrow}_{\hat{G}} xAy \overset{*}{\Rightarrow}_{\hat{G}} w$$

同时，由于文法 $\hat{G}$ 是通过删除G中的产生式而生成的，所以 $\hat{P} \subseteq P$。因而 $L(\hat{G}) \subseteq L(G)$ 成立。综合上述结论，可以得到，G与 $\hat{G}$ 是等价的。■

6.1.3 消除λ产生式

在上下文无关文法中，如果某个产生式的右部是空串，则该产生式有时是不必要的。

定义6.2 *上下文无关文法中任意形如*

$$A \to \lambda$$

的产生式称为λ产生式（λ-production）。任何存在推导

$$A \overset{*}{\Rightarrow} \lambda \tag{6-3}$$

的变量A称为可空的（nullable）。

一个文法生成的语言可以不包含λ，但它仍可能存在λ产生式与可空变量。在这种情况下，λ产生式是可以消除的。

156

例6.4 考虑文法

$$S \to aS_1b$$
$$S_1 \to aS_1b|\lambda$$

该文法生成的语言$\{a^nb^n : n \geqslant 1\}$不包含λ。因而，通过增加一些新的产生式，这些产生式是用λ替换出现在产生式右部的S_1得到的，同时λ产生式$S_1 \to \lambda$也就可以从文法中消除了。这样将得到新文法

$$S \to aS_1b|ab$$
$$S_1 \to aS_1b|ab$$

显而易见，上述新文法与原始文法生成的语言相同。

在更一般的情况下，λ产生式可以由一个类似上述过程但更复杂一点的方法消除。 □

定理6.3 *设G是任意的上下文无关文法且λ不在$L(G)$中，则存在一个等价的不包含λ产生式的文法 $\hat{G}$。*

证明：首先我们通过如下步骤找到所有G的可空变量的集合V_N

1. 对于所有形如$A \to \lambda$的产生式，将A加入V_N中。
2. 重复下述步骤直到再也没有新的变量可以加入到V_N中。对于所有形如

$$B \to A_1A_2 \cdots A_n$$

的产生式，如果$A_1, A_2, \cdots, A_n$都属于V_N，则将B加入V_N中。

一旦找到集合V_N，就可以构造产生式 $\hat{P}$ 了。我们考察P中所有形如

$$A \to x_1x_2 \cdots x_m, m \geqslant 1$$

的产生式，其中$x_i \in V \cup T$。对于P中每一个上述形式的产生式，我们将该产生式以及使用λ替换其可空变量所有可能组合而得到的产生式加入 $\hat{P}$。例如，如果x_i和x_j是可空的，则在 $\hat{P}$ 存在一个使用λ替换x_i的产生式，一个使用λ替换x_j的产生式，以及一个x_i和x_j都被λ替换的产生式。存在一种特殊情况，如果所有的x_i都是可空的，则产生式$A \to \lambda$将不被加入 $\hat{P}$ 中。

157

对 $\hat{G}$ 与G等价性的证明是简单的，我们将它留给读者来完成。■

例6.5 针对文法

$$S \to ABaC$$
$$A \to BC$$
$$B \to b|\lambda$$
$$C \to D|\lambda$$
$$D \to d$$

构造一个等价的上下文无关文法，使得它不包含λ产生式。

根据定理6.3中构造方法的第1步，我们可以得到可空变量A, B, C。然后通过应用构造方法的第2步可以得到

$$S \to ABaC|BaC|AaC|ABa|aC|Aa|Ba|a$$
$$A \to B|C|BC$$
$$B \to b$$
$$C \to D$$
$$D \to d$$

□

6.1.4 消除单位产生式

从定理6.2可以看出，如果某个产生式的两边都是单个变量的话，则这个变量有时是不必要的。

定义6.3 上下文无关文法中形如

$$A \to B$$

的产生式，其中$A, B \in V$，我们称为单位产生式。 158

为了消除单位产生式，我们运用定理6.1中讨论的代入规则。根据下面定理中的构造方法可以看出，只要在过程中细心，这一点是完全可以做到。

定理6.4 设$G = (V, T, S, P)$是一个任意的上下文无关文法且不包含λ产生式，则存在一个上下文无关文法$\hat{G} = (\hat{V}, \hat{T}, S, \hat{P})$，它与$G$等价且不存在单位产生式。

证明：显而易见，任意形如$A \to A$的单位产生式都可以直接从文法中消除，不会影响文法产生的语言。因此我们只需要考虑形如$A \to B$的产生式，其中A与B是不同的变量。乍一看，通过令$x_1 = x_2 = \lambda$，可以直接运用定理6.1，将

$$A \to B$$

替换成

$$A \to y_1|y_2|\cdots|y_n$$

但实际上这并不总是可行的；因为在存在形如

$$A \to B$$
$$B \to A$$

产生式的特殊情况中，单位产生式将无法消除。为了解决这一问题，我们首先针对每一个A，找到所有的变量B，它能使得

$$A \overset{*}{\Rightarrow} B \tag{6-4}$$

这一工作我们可以通过变量依赖图来完成，如果文法中存在单位产生式$C \to D$，则在依赖图中存在边（C, D）；而如果依赖图中A与B之间存在一条通道，则式（6-4）成立。新文法$\hat{G}$通过如下方式生成，它首先将所有的非单位产生式P加入$\hat{P}$，然后针对所有满足式（6-4）的A与B，在$\hat{P}$中加入产生式

$$A \to y_1|y_2|\cdots|y_n$$

其中$B \to y_1|y_2|\cdots y_n$是$\hat{P}$中所有B为左部的产生式集合。注意到由于$B \to y_1|y_2|\cdots|y_n$属于$\hat{P}$，所以任意一个y_i都不会是单个变量，因此在最后这一步中不会生成单位产生式。

159 为了说明结果文法与原始文法是等价的，可以采用与定理6.1同样的证明过程。■

例6.6 消除文法

$$S \to Aa|B$$
$$B \to A|bb$$
$$A \to a|bc|B$$

中的所有单位产生式。

图6-3给出了针对单位产生式的依赖图；从该图我们可以得到$S \overset{*}{\Rightarrow} A$，$S \overset{*}{\Rightarrow} B$，$B \overset{*}{\Rightarrow} A$以及$A \overset{*}{\Rightarrow} B$。因此通过在新文法中加入原文法的非单位产生式

$$S \to Aa$$
$$A \to a|bc$$
$$B \to bb$$

以及替换单位产生式而得到的新产生式

$$S \to a|bc|bb$$
$$A \to bb$$
$$B \to a|bc$$

这样我们就可以得到等价的新文法

$$S \to a|bc|bb|Aa$$
$$A \to a|bb|bc$$
$$B \to a|bb|bc$$

注意单位产生式的消除同时导致了B以及与之相关的产生式成为无用的。 □

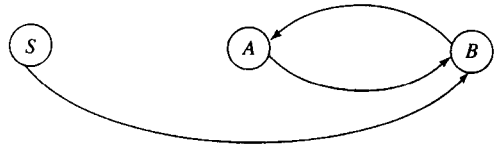


图 6-3

通过将所有的前述结论综合起来，我们将看到，对于上下文无关语言的文法可以消除其中的无用产生式、λ产生式以及单位产生式。

定理6.5 设L是不包含λ的上下文无关语言。则存在一个产生L的上下文无关文法，该文法不含无用产生式、λ产生式或单位产生式。

160

证明：定理6.2、定理6.3以及定理6.4中的方法分别消除了这些形式的产生式。这里唯一需要考虑的问题是，消除某一种形式的产生式可能会引入其他形式的产生式；比如，消除λ产生式可能会引入单位产生式。同时，定理6.4中要求文法中不存在λ产生式。但是注意到消除单位产生式不会引入λ产生式（见本节习题15）以及消除无用产生式不会引入λ产生式或单位产生式（见本节习题16）。因此我们可以通过如下步骤来消除所有这些不合要求的产生式：

1. 消除λ产生式。
2. 消除单位产生式。
3. 消除无用产生式。

通过上述步骤得到的结果，消除了所有这些形式的产生式，定理得证。■

习题

1. 通过说明从

$$S \overset{*}{\Rightarrow}_{\hat{G}} w$$

可以得到

$$S \overset{*}{\Rightarrow}_{G} w$$

来完成定理6.1的证明。

2. 在例6.1中，分别使用原始文法与修改文法给出符号串$ababbac$的一个推导。
3. 证明文法

$$S \to abAB|ba$$
$$A \to aaa$$
$$B \to aA|bb$$

与文法

$$S \to abAaA|abAbb|ba$$
$$A \to aaa$$

是等价的。

161

4. 在定理6.1中，为什么需要假定变量A与B是不同的？
5. 消除如下文法的所有无用产生式，并给出由该文法产生的语言。

$$S \to aS|AB$$
$$A \to bA$$
$$B \to AA$$

6. 消除如下文法

$$S \to a|aA|B|C$$

$$A \to aB|\lambda$$
$$B \to Aa$$
$$C \to cCD$$
$$D \to ddd$$

的无用产生式。

7. 消除如下文法

$$S \to AaB|aaB$$
$$A \to \lambda$$
$$B \to bbA|\lambda$$

的所有λ产生式。

8. 消除如下文法中的所有λ产生式、单位产生式以及无用产生式。并给出由该文法生成的语言。●

$$S \to aA|aBB$$
$$A \to aaA|\lambda$$
$$B \to bB|bbC$$
$$C \to B$$

9. 消除习题7中文法的所有单位产生式。
10. 完成定理6.3的证明。
11. 完成定理6.4的证明。
162 12. 使用定理6.3中的构造方法消除例5.4中文法的λ产生式，并给出由该文法生成的语言。
13. 假定G是一个上下文无关文法且$\lambda \in L(G)$，证明如果我们针对它应用定理6.3中的构造方法，将能够得到新的文法$\hat{G}$并满足$L(\hat{G}) = L(G) - \{\lambda\}$。
14. 给出一个文法实例，说明在消除λ产生式后引入了以前不存在的单位产生式的情况。●
15. 设G是一个没有λ产生式的上下文无关文法，但可能存在单位产生式，通过它证明定理6.4的构造方法不会引入λ产生式。
16. 证明：如果一个文法不存在λ产生式与单位产生式，则根据定理6.2中的构造方法，在消除无用产生式的过程中不会引入这些形式的产生式。●
17. 验证定理6.1证明中的命题：变量B在它一出现时就可以被替换。
18. 假定上下文无关文法$G = (V, T, S, P)$中存在一个如下形式的产生式

$$A \to xy$$

其中$x, y \in (V \cup T)^+$，试证明如果该产生式被

$$A \to By$$
$$B \to x$$

替换，其中$B \notin V$，则得到的新文法与原文法是等价的。

19. 考虑定理6.2中用于消除无用产生式的方法，颠倒其两个部分的顺序，也就是先消除S不能到达的变量，然后消除那些不会生成终结符串的变量。则采用这一新的方法是否可行？如

果可行，证明之，否则给出一个反例。

20. 通过介绍文法复杂性的概念，我们有可能给出化简的准确定义。针对复杂性，存在多种的定义方法，其中一种就是考虑所有产生式中所有符号串的长度，比如，我们可以使用

$$complexity(G) = \sum_{A \to v \in P} \{1 + |v|\}$$

作为复杂性定义。基于这种定义，试说明消除无用产生式总是能够降低复杂性的，并给出消除λ产生式以及单位产生式对复杂性的影响。 163

21. 对于给定语言L的上下文无关文法G以及G的任意等价形式$\hat{G}$，如果$complexity(G) < complexity(\hat{G})$，则$G$是最小的。通过实例说明消除无用产生式并不必然生成最小文法。●

★22. 证明：设$G = (V, T, S, P)$是一个上下文无关文法，将其中左部给定变量（比如A）的产生式集合分成两个不相交的子集

$$A \to Ax_1 | Ax_2 | \cdots | Ax_n$$

和

$$A \to y_1 | y_2 | \cdots | y_m$$

其中$x_i, y_i \in (V \cup T)^*$，且$A$不是任何$y_i$的前缀。考虑文法$\hat{G} = (V \cup \{Z\}, T, S, \hat{P})$，其中$Z \notin V$且$\hat{P}$是通过采用

$$A \to y_i | y_i Z, \quad i = 1, 2, \cdots, m$$
$$Z \to x_i | x_i Z, \quad i = 1, 2, \cdots, n$$

替换所有A在左边的产生式而得到的。则$L(G) = L(\hat{G})$。

23. 使用习题22的结论重写文法

$$A \to Aa | aBc | \lambda$$
$$B \to Bb | bc$$

使得它不再存在形如$A \to Ax$或者$B \to Bx$的产生式。

★24. 将左边包含变量A的产生式集合分成两组不相交的子集

$$A \to x_1 A | x_2 A | \cdots | x_n A$$

和

$$A \to y_1 | y_2 | \cdots | y_m$$

其中A不是任意y_i的后缀。证明通过将这些产生式用

$$A \to y_i | Zy_i, \quad i = 1, 2, \cdots, m$$
$$Z \to x_i | Zx_i, \quad i = 1, 2, \cdots, n$$

替换而得到的文法与原文法等价。 164

6.2 两个重要的范式

针对上下文无关文法，我们可以建立多种类型的范式，其中一些由于它们广泛的用途而被深入研究，本节我们将简单介绍其中的两种。

6.2.1 乔姆斯基范式

我们将看到，存在一种类型的范式，其中产生式右部的符号数目是严格限制的。特别地，我们可以限制产生式右部的符号数不超过两个，这种形式的一个实例即是乔姆斯基范式。

定义6.4 如果一个上下文无关文法的所有产生式都形如

$$A \to BC$$

或

$$A \to a$$

其中，A, B, C属于V，a属于T。则该文法属于乔姆斯基范式。

例6.7 文法

$$S \to AS|a$$
$$A \to SA|b$$

是乔姆斯基范式；而文法

$$S \to AS|AAS$$
$$A \to SA|aa$$

165 不是，因为产生式$S \to AAS$和$A \to aa$与定义6.4中的条件发生冲突。 □

定理6.6 对于任意的上下文无关文法$G = (V, T, S, P)$其中$\lambda \notin L(G)$，都存在一个等价的上下文无关文法$\hat{G} = (\hat{V}, \hat{T}, S, \hat{P})$属于乔姆斯基范式。

证明：根据定理6.5，不失一般性地，可以假定文法G中没有λ产生式与单位产生式。则$\hat{G}$的构造过程包括如下两个步骤：

步骤1：根据文法G构造文法$G_1 = (V_1, T, S, P_1)$。首先考察P中形如

$$A \to x_1 x_2 \cdots x_n \tag{6-5}$$

的产生式，其中每个x_i都是V或者T中的符号。如果$n = 1$，由于不存在单位产生式，则x_1必然是一个终结符。在这种情况下，将该产生式加到P_1中。如果$n \geqslant 2$，那么对于每一个$a \in T$，引入新的变量B_a。对于P中所有形如式（6-5）的产生式，我们在P_1中加入产生式

$$A \to C_1 C_2 \cdots C_n$$

其中

$$C_i = x_i \text{ 若 } x_i \text{ 在 } V \text{ 中}$$

且

$$C_i = B_a \text{ 若 } x_i = a$$

对于每一个B_a，我们也将在P_1中加入产生式

$$B_a \to a$$

算法的这一步消除了右部长度大于1的产生式中的所有终结符，并用新引入的变量来替换它们。通过这一步骤，我们将得到文法G_1，其中的产生式形如

$$A \to a \tag{6-6}$$

或

$$A \to C_1C_2\cdots C_n \tag{6-7}$$

其中$C_i \in V_1$

根据定理6.1，很容易得到

$$L(G_1) = L(G)$$

166

步骤2：在第2步中，通过引入新的变量来减少产生式右部的长度，首先我们将所有形如式（6-7）且$n=2$的产生式和形如式（6-6）的产生式加到$\hat{P}$中。对于$n>2$的产生式，我们引入新变量$D_1, D_2, \cdots$，并将产生式

$$\begin{gathered} A \to C_1D_1 \\ D_1 \to C_2D_2 \\ \vdots \\ D_{n-2} \to C_{n-1}C_n \end{gathered}$$

加到$\hat{P}$中。显然，最后得到的结果文法$\hat{G}$是乔姆斯基范式。重复应用定理6.1可以得到$L(G_1) = L(\hat{G})$，因而

$$L(\hat{G}) = L(G)$$

上述证明过程并不规范，但很容易给出更加准确的证明，我们将这一工作留给读者。■

例6.8 将具有产生式

$$\begin{gathered} S \to ABa \\ A \to aab \\ B \to Ac \end{gathered}$$

的文法转换成乔姆斯基范式。

该文法满足定理6.6的构造条件，不存在λ产生式与单位产生式。

第一步，我们引入新的变量B_a, B_b, B_c，并通过应用上述算法可以得到

$$\begin{gathered} S \to ABB_a \\ A \to B_aB_aB_b \\ B \to AB_c \\ B_a \to a \\ B_b \to b \\ B_c \to c \end{gathered}$$

167

第二步，通过引入另外的变量将上步得到的前两个产生式转换成范式，这样得到的最终结果如下：

$$\begin{gathered} S \to AD_1 \\ D_1 \to BB_a \\ A \to B_aD_2 \end{gathered}$$

$$D_2 \to B_a B_b$$
$$B \to AB_c$$
$$B_a \to a$$
$$B_b \to b$$
$$B_c \to c$$

□

6.2.2 格里巴克范式

另一种有用的文法形式是格里巴克范式。它不是限制产生式右部的长度，而是限制终结符与变量可以出现的位置。验证格里巴克范式的证明过程相对比较复杂并且不是很明显。类似地，对给定的上下文无关文法构造与其等价的格里巴克范式也是一个枯燥的过程。因此这里我们只做简要的介绍，但格里巴克范式有很多理论与实践性的结论。

定义6.5 对于一个上下文无关文法，如果其所有的产生式都具有如下形式

$$A \to ax$$

其中$a \in T$且$x \in V^*$，则它属于格里巴克范式。

通过与定义5.4比较，我们可以看出$A \to ax$的形式在简单文法与格里巴克范式中是相同的，但在格里巴克范式中，没有限制（A, a）对只能出现一次。这种形式上的自由使得格里巴克范式具有简单文法所没有的一般性。

168

如果一个文法不是格里巴克范式，则我们可以运用前述学习的一些技巧来重写文法使之满足格里巴克范式的要求。这里将给出两个简单的例子。

例6.9 文法

$$S \to AB$$
$$A \to aA|bB|b$$
$$B \to b$$

不是格里巴克范式，但通过运用定理6.1中的替换方法，我们可以直接得到属于格里巴克范式的等价形式

$$S \to aAB|bBB|bB$$
$$A \to aA|bB|b$$
$$B \to b$$

□

例6.10 将文法

$$S \to abSb|aa$$

转换成格里巴克范式。

这里我们可以采用与构造乔姆斯基范式类似的方法，首先引入变量A和B，它们在实际含义上分别与a和b是相同的。这样通过将原产生式中的终结符替换成相应的变量即可以得到属于格里巴克范式的等价文法。

$$S \to aBSB|aA$$
$$A \to a$$
$$B \to b$$

□

一般来说，对给定的文法，将其转换成格里巴克范式或进行相应的证明都不是简单的事情。但通过引入格里巴克范式，可以使下一章中一个重要结论的技术讨论大大简化。然而从概念的角度出发，格里巴克范式在我们的讨论中并没有起到更进一步的作用，因此这里我们只是简单地给出它的相关结论而没有提供证明。 169

定理6.7 *对于每一个$\lambda \notin L(G)$的上下文无关文法G，都存在一个与其等价的属于格里巴克范式的文法$\hat{G}$。*

习题

1. 给出定理6.6的详细证明过程。
2. 给出文法$S \to aSb|ab$的乔姆斯基范式。
3. 给出文法$S \to aSaA|A$，$A \to abA|b$的乔姆斯基范式。
4. 给出文法

$$S \to abAB$$
$$A \to bAB|\lambda$$
$$B \to BAa|A|\lambda$$

的乔姆斯基范式。

5. 给出文法

$$S \to AB|aB$$
$$A \to aab|\lambda$$
$$B \to bbA$$

的乔姆斯基范式。

6. 设$G = (V, T, S, P)$是任意一个没有单位产生式或λ产生式的上下文无关文法，设k为P中产生式右部符号数目的最大值。证明存在一个等价的乔姆斯基范式，使得它的产生式规则数目不超过$(k-1)|P| + |T|$。
7. 画出习题4中文法的依赖图。
8. 如果一个语言存在线性文法，则称之为线性语言（定义见例3.13）。设L是不包含λ的线性语言，证明存在一个文法$G = (V, T, S, P)$，它的产生式具有如下的某种形式

$$A \to aB$$
$$A \to Ba$$
$$A \to a$$

其中$a \in T$, $A, B \in V$，满足$L = L(G)$。 170

9. 证明对于任意的上下文无关文法$G = (V, T, S, P)$都存在一个等价的上下文无关文法形式，其产生式形如$A \to aBC$或$A \to \lambda$，其中$a \in \Sigma \cup \{\lambda\}$, $A, B, C \in V$。
10. 给出文法

$$S \to aSb|bSa|a|b$$

的格里巴克范式。

11. 给出文法

$$S \to aSb|ab$$

的格里巴克范式。

12. 给出文法

$$S \to ab|aS|aaS$$

的格里巴克范式。

13. 给出文法

$$S \to ABb|a$$
$$A \to aaA|B$$
$$B \to bAb$$

的格里巴克范式。

14. 是否每一个线性文法都能够转换成这样一种形式：产生式形如$A \to ax$，其中$a \in T$，$x \in V \cup \{\lambda\}$。

15. 如果一个上下文无关文法的所有产生式满足如下模式

$$A \to aBC$$
$$A \to aB$$
$$A \to a$$

其中$a \in T, A, B, C \in V$，则该文法属于双标准式。

根据双标准式的定义，将文法$G = (\{S, A, B, C\}, \{a, b\}, S, P)$转换成双标准式，其中$P$为

$$S \to aSA$$
$$A \to bABC$$
$$B \to b$$
$$C \to aBC$$

★16. 证明双标准式的一般性，即对于任意的上下文无关文法G且$\lambda \notin L(G)$，都存在一个等价的文法满足双标准式。

171

6.3 上下文无关文法的成员资格判定算法*

在5.2节中，我们已经指出，如果没有采用特别的策略，基于现有上下文无关文法的成员资格判定算法和分析算法需要$|w|^3$步才能完成符号串w的分析。现在我们将验证这一命题。这里我们描述的算法称为CYK算法，它是根据其提出者J. Cocke，D. H. Younger和T. Kasami进行命名的。使用该算法的一个前提条件是上下文无关文法必须满足乔姆斯基范式，它是通过将一个问题采用如下方法分解成一系列的小问题来实现的。给定文法$G = (V, T, S, P)$，它满足乔姆斯基范式，对于符号串

$$w = a_1 a_2 \cdots a_n$$

我们定义其子串

$$w_{ij} = a_i \cdots a_j$$

以及V的子集

$$V_{ij} = \{A \in V : A \overset{*}{\Rightarrow} w_{ij}\}$$

显然，$w \in L(G)$ 当且仅当$S \in V_{1n}$。

为了计算V_{ij}，注意到$A \in V_{ii}$当且仅当G中存在产生式$A \to a_i$。因此对于$1 \leqslant i \leqslant n$的所有$V_{ii}$，可以通过检测$w$和文法中的所有产生式来计算。接下来，注意由于$j > i$，$A$能推导出$w_{ij}$当且仅当存在产生式$A \to BC$，其中对于某个$k$（$i \leqslant k, k < j$），$B \overset{*}{\Rightarrow} w_{ik}$且$C \overset{*}{\Rightarrow} w_{k+1j}$。换句话说，

$$V_{ij} = \bigcup_{k \in \{i, i+1, \cdots, j-1\}} \{A : A \to BC, \text{其中} B \in V_{ik}, C \in V_{k+1,j}\} \tag{6-8}$$

通过对式（6-8）中下标的检查，可以发现按照如下的顺序类推，能够计算出所有的V_{ij}:

1. 计算$V_{11}, V_{22}, \cdots, V_{nn}$
2. 计算$V_{12}, V_{23}, \cdots, V_{n-1,n}$
3. 计算$V_{13}, V_{24}, \cdots, V_{n-2,n}$

等等。

172

例6.11 确定符号串$w = aabbb$是否属于由文法

$$S \to AB$$
$$A \to BB \mid a$$
$$B \to AB \mid b$$

生成的语言。

首先注意到$w_{11} = a$，因而V_{11}是所有能够直接推导出a的变量的集合，也就是说，$V_{11} = \{A\}$。同样，由于$w_{22} = a$，我们也可以得到$V_{22} = \{A\}$。

类似地，可以得到

$$V_{11} = \{A\}, V_{22} = \{A\}, V_{33} = \{B\}, V_{44} = \{B\}, V_{55} = \{B\}$$

现在可以应用式（6-8）计算

$$V_{12} = \{A : A \to BC, B \in V_{11}, C \in V_{22}\}$$

由于$V_{11} = \{A\}$，$V_{22} = \{A\}$，该集合由所有右部为AA的产生式的左部变量组成。由于不存在这样的产生式，所以V_{12}为空，接下来计算

$$V_{23} = \{A : A \to BC, B \in V_{22}, C \in V_{33}\}$$

它要求产生式的右部是AB，因此我们可以得到$V_{23} = \{S, B\}$，依此类推我们将得到

$$V_{12} = \varnothing, V_{23} = \{S, B\}, V_{34} = \{A\}, V_{45} = \{A\}$$
$$V_{13} = \{S, B\}, V_{24} = \{A\}, V_{35} = \{S, B\}$$
$$V_{14} = \{A\}, V_{25} = \{S, B\}$$
$$V_{15} = \{S, B\}$$

因此$w \in L(G)$。 □

上述的CYK算法可以用于确定由乔姆斯基范式生成语言的成员资格。通过更多地了解元素V_{ij}是怎样被推导出来的，CYK算法可以转换成一个分析方法。可以看出CYK算法需要$O(n^3)$

步，注意只有$n(n+1)/2$个V_{ij}的集合需要计算。并且它们中的每一个在式（6-8）中至多涉及n个项目的求值，因此结论成立。

173

习题

1. 使用CYK算法确定符号串*aabb*，*aabba*以及*abbbb*是否属于由例6.11给出的文法生成的语言。
2. 根据例6.11中的文法，使用CYK算法找到符号串*aab*的分析。●
3. 根据习题2中采用的方法证明CYK算法可以转换成一个分析方法。

★★4. 使用习题3的结果写一个计算程序，它能够针对所有具有格里巴克范式的上下文无关文法进行语法分析。

174

第7章 下推自动机

采用上下文无关文法描述上下文无关语言是方便的，在程序设计语言的定义中使用BNF就说明了这一点。针对上下文无关语言的下一个问题就是是否存在一类自动机能够与它关联起来。在前面的讨论中，我们已经看到，有穷自动机并不能识别所有的上下文无关语言。直观上，这是由于有穷自动机存在严格的有穷存储限制，而上下文无关语言的识别可能需要存储没有边界的信息。例如，在扫描语言$L=\{a^nb^n : n \geqslant 0\}$中的一个符号串时，不仅需要检查所有在第一个$b$之前的$a$，而且必须记录$a$的个数；同时由于$n$是无限的，使得这种计数不能在有穷的存储中完成，因此我们需要一种能够无限计数的机器。而从另外一个例子，比如$\{ww^R\}$中可以看出，仅仅有无限制计数的能力还是不够的，它还需要能够逆序地存储并匹配一个符号序列，这表明我们可以尝试使用栈作为存储机制，并允许以与栈类似的方式操作无限的存储。这样我们就可以得到一类称之为*下推自动机*（pushdown automata, pda）的机器。 175

在这一章中，我们将介绍下推自动机与上下文无关语言之间的联系。我们首先证明如果允许下推自动机以非确定型的方式工作，那么就能够得到恰好可以接受上下文无关语言族的自动机类。但同时我们也将看到，它不再存在一种确定型的等价形式。确定型下推自动机类定义了新的语言族，即确定型上下文无关语言，它是上下文无关语言的一个真子集。在程序设计语言的处理中，确定型上下文无关语言是一个重要的语言族，所以我们将通过与确定型上下文无关语言对应的文法的简要介绍来结束本章。

7.1 非确定型下推自动机

图7-1给出了下推自动机的图示。其中，控制部件的每一次迁移都将从输入文件中读入一个字符，同时通过常规的栈操作来改变栈的内容。控制部件的每一次迁移都是由当前输入符号和当前栈顶符号同时决定的。而迁移的结果将导致控制部件状态以及栈顶符号发生改变。在本章中，我们的讨论只限于可以作为接受器的下推自动机。

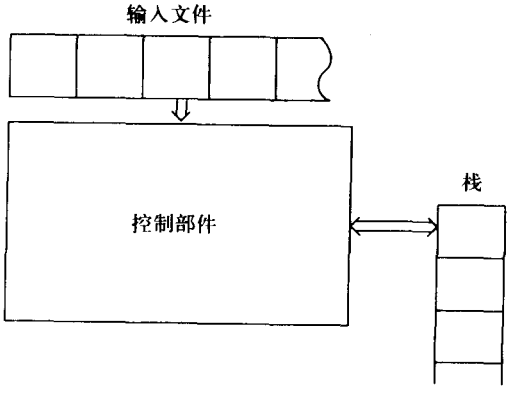


图 7-1

7.1.1 下推自动机的定义

176 通过对直观表示的形式化，我们将得到下推自动机的一个准确定义。

定义7.1 非确定型下推接受器（nondeterministic pushdown accepter, npda）由一个七元组定义

$$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$$

其中，Q是控制部件内部状态的有穷集；

Σ是输入字母表；

Γ是一个有穷符号集，称为栈字母表（stack alphabet）；

$\delta: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow Q \times \Gamma^*$的有穷子集，称为转移函数；

$q_0 \in Q$是控制部件的初始状态；

$z \in \Gamma$是栈开始符（stack start symbol）；

$F \subseteq Q$是终止状态集合。

δ的定义域与值域采用的复杂形式化表示便于封闭性检查。δ的参数是控制部件的当前状态、当前输入符号以及当前栈顶符号，δ的返回结果是对集合(q, x)，其中q是控制部件的下一个状态，x是用于替换原单个栈顶符号的符号串。注意到δ的第二个参数可以是λ，它表明自动机的一次迁移可以不需要输入符号，我们称之为λ转移。同时根据定义，δ的执行总需要一个栈符号；如果栈为空，则相应的转移将无法完成。最后一点，定义中对δ的值域为一有穷子集的要求是必要的，这是由于$Q \times \Gamma^*$是一无穷集合因而也就存在无穷子集。而当npda存在几种迁移选择时，则这一选择必须限制到一有穷可能性集合中。

例7.1 假定一个npda的转移规则集合包含

$$\delta(q_1, a, b) = \{(q_2, cd), (q_3, \lambda)\}$$

如果某一时刻，控制部件的状态是q_1，输入符号是a，当前的栈顶符号是b，则可能出现两种情况中的一种：（1）控制部件进入状态q_2，用符号串cd替换栈顶符号b，或者（2）控制部件进入状态q_3，其中栈顶符号b被删除。在表示法中，我们假定向栈中插入一个符号串的操作是
177 从符号串的右端开始逐个完成的。 □

例7.2 考虑某一npda，其中

$$Q = \{q_0, q_1, q_2, q_3\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{0, 1\}$$
$$z = 0$$
$$F = \{q_3\}$$

并且

$$\delta(q_0, a, 0) = \{(q_1, 10), (q_3, \lambda)\}$$
$$\delta(q_0, \lambda, 0) = \{(q_3, \lambda)\}$$
$$\delta(q_1, a, 1) = \{(q_1, 11)\}$$
$$\delta(q_1, b, 1) = \{(q_2, \lambda)\}$$

$$\delta(q_2, b, 1) = \{(q_2, \lambda)\}$$
$$\delta(q_2, \lambda, 0) = \{(q_3, \lambda)\}$$

那么，该自动机是如何工作的?

首先，我们注意到并不是输入符号与栈符号的所有可能组合都在转移函数中做了定义。比如，对$\delta(q_0, b, 0)$，就不存在相应的入口。这与非确定型有穷状态自动机的情形是相同的，一个没有定义的转移函数其值为空集合，而在npda中它表示死格局。

上述npda中的关键转移是:

$$\delta(q_1, a, 1) = \{(q_1, 11)\}$$

它表示当自动机读到a时，在栈顶增加一个1；以及

$$\delta(q_2, b, 1) = \{(q_2, \lambda)\}$$

它表示读到b时从栈顶删除一个1。这两步转移分别完成对a的计数以及a的个数与对b个数的匹配。控制部件一直保持状态q_1直到碰到第一个b，此时进入状态q_2，这就保证了最后一个a之前没有任何b。通过分析其他转移函数，我们将看到当且仅当输入符号串在语言

$$L = \{a^n b^n : n \geqslant 0\} \cup \{a\}$$

中时，npda才将以最终状态q_3结束。

与有穷自动机类似，我们可以说该npda能够接受上述语言。当然，在给出这样一个命题之前，我们需要首先定义npda能够接受一个语言的具体含义。 □ 178

为了简化讨论，我们引入一种方便的表示法，它用于描述一个npda在处理一个符号串的过程中经历的连续格局。它涉及的相关因素包括某一时刻控制部件的当前状态，输入符号串的未读入部分以及当前的栈内容。综合这些因素就完全决定了npda所能够进行处理的全部可能途径。设有三元组

$$(q, w, u)$$

其中，q是控制部件的状态，w是输入符号串的未读入部分，u是栈的当前内容（最左边的符号表示栈顶符号），我们称该三元组为下推自动机的*瞬时描述*（instantaneous description）。从一个瞬时描述到另一个瞬时描述的迁移采用符号$\vdash$来表示；这样，

$$(q_1, aw, bx) \vdash (q_2, w, yx)$$

当且仅当

$$(q_2, y) \in \delta(q_1, a, b)$$

才是可能的。

包含任意多个步骤的迁移，可以用$\vdash^*$表示。在存在多个自动机的情况下，我们将采用$\vdash_M$表明该迁移是由特定自动机M完成的。

7.1.2 下推自动机接受的语言

定义7.2 设$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$为一非确定型下推自动机，则$M$接受的语言是集合

$$L(M) = \{w \in \Sigma^* : (q_0, w, z) \vdash^*_M (p, \lambda, u), p \in F, u \in \Gamma^*\}$$

总之，由M接受的语言就是所有这些符号串的集合，它们都能够在结束接受符号串时将M置为终态。而栈的最终内容u与这种接受形式的定义是无关的。

例7.3 针对语言

$$L = \{w \in \{a, b\}^* : n_a(w) = n_b(w)\}$$

构造一个npda。

在例7.2中，针对这一问题的解决方案涉及a与b的计数，这点很容易用栈完成。同时，这里我们不需要考虑a与b的顺序问题，因而可以在读到一个a时插入一个计数器符号（比如0）到栈中，在读到一个b时，就从栈中弹出一个计数器符号。这样做存在的一个难题是：如果w的一个前缀中a的个数比b的个数少，则在栈中将找不到足够的0来完成转移。为了解决这一问题，我们可以采用一个负计数器符号（比如1）用于对还没有与a匹配的b计数。这样得到的完整解决方案就是一个npda $M = (\{q_0, q_f\}, \{a, b\}, \{0, 1, z\}, \delta, q_0, z, \{q_f\})$，其中$\delta$为

179

$$\begin{aligned}
\delta(q_0, \lambda, z) &= \{(q_f, z)\} \\
\delta(q_0, a, z) &= \{(q_0, 0z)\} \\
\delta(q_0, b, z) &= \{(q_0, 1z)\} \\
\delta(q_0, a, 0) &= \{(q_0, 00)\} \\
\delta(q_0, b, 0) &= \{(q_0, \lambda)\} \\
\delta(q_0, a, 1) &= \{(q_0, \lambda)\} \\
\delta(q_0, b, 1) &= \{(q_0, 11)\}
\end{aligned}$$

在处理符号串$baab$的过程中，该npda能够执行如下的迁移

$$\begin{aligned}
(q_0, baab, z) &\vdash (q_0, aab, 1z) \vdash (q_0, ab, z) \\
&\vdash (q_0, b, 0z) \vdash (q_0, \lambda, z) \vdash (q_f, \lambda, z)
\end{aligned}$$

因此该符号串可以被接受。 □

例7.4 构造一个npda能够接受语言

$$L = \{ww^R : w \in \{a, b\}^+\}$$

我们将运用这样的一个事实：通过栈能够得到与插入顺序相反的符号。当读取符号串的第一部分时，我们在栈中压入连续的符号。对于第二部分，我们比较当前符号与栈顶符号，只要它们匹配就一直进行下去。由于从栈中获取的符号与其插入的顺序相反，因此当且仅当输入形式为ww^R时，才能得到一个完全的匹配。

上述方案中，一个明显的难点是我们并不知道符号串的中间位置，也就是w结束和w^R开始的地方。但是自动机的非确定型特性可以帮助我们解决这一问题；npda可以正确地猜测中间位置并转换状态。这样，针对该问题的解决方案是：$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$，其中

$$\begin{aligned}
Q &= \{q_0, q_1, q_2\} \\
\Sigma &= \{a, b\} \\
\Gamma &= \{a, b, z\} \\
F &= \{q_2\}
\end{aligned}$$

180

它的转移函数可以分成几个部分：在栈中压入w的转移函数集合

$$\delta(q_0, a, a) = \{(q_0, aa)\}$$
$$\delta(q_0, b, a) = \{(q_0, ba)\}$$
$$\delta(q_0, a, b) = \{(q_0, ab)\}$$
$$\delta(q_0, b, b) = \{(q_0, bb)\}$$
$$\delta(q_0, a, z) = \{(q_0, az)\}$$
$$\delta(q_0, b, z) = \{(q_0, bz)\}$$

猜测符号串中间位置的转移函数集合（其中npda的状态从q_0转至q_1）为

$$\delta(q_0, \lambda, a) = \{(q_1, a)\}$$
$$\delta(q_0, \lambda, b) = \{(q_1, b)\}$$

栈中内容匹配w^R的转移函数集合为

$$\delta(q_1, a, a) = \{(q_1, \lambda)\}$$
$$\delta(q_1, b, b) = \{(q_1, \lambda)\}$$

最后

$$\delta(q_1, \lambda, z) = \{(q_2, z)\}$$

可用于识别一个成功的匹配。

上述自动机接受$abba$的迁移序列为：

$$(q_0, abba, z) \vdash (q_0, bba, az) \vdash (q_0, ba, baz)$$
$$\vdash (q_1, ba, baz) \vdash (q_1, a, az) \vdash (q_1, \lambda, z) \vdash (q_2, z)$$

针对符号串中间位置定位的非确定型选择在第三个迁移中完成。在这一阶段，pda的瞬时描述为（q_0, ba, baz），并且对下一步迁移存在两种选择：一种是运用$\delta(q_0, b, b) = \{(q_0, bb)\}$并执行迁移

$$(q_0, ba, baz) \vdash (q_0, a, bbaz)$$

另一种就是在上述迁移序列中所采用的，也就是$\delta(q_0, \lambda, b) = \{(q_1, b)\}$。只有后一选择才能实现输入的接受。 □ 181

习题

1. 构造一状态数少于4的pda，使得它能够接受例7.2中pda所接受的语言。
2. 证明：例7.4中的pda不能接受$\{ww^R\}$以外的任何符号串。●
3. 构造能够接受下列正则语言的npda：
   (a) $L_1 = L(aaa^*b)$
   (b) $L_1 = L(aab^*aba^*)$
   (c) L_1与L_2的并
   (d) $L_1 - L_2$

4. 构造npda能够接受下列定义在$\Sigma = \{a, b, c\}$上的语言:

(a) $L = \{a^n b^{2n} : n \geqslant 0\}$

(b) $L = \{wcw^R : w \in \{a, b\}^*\}$

(c) $L = \{a^n b^m c^{n+m} : n \geqslant 0, m \geqslant 0\}$

(d) $L = \{a^n b^{n+m} c^m : n \geqslant 0, m \geqslant 1\}$

(e) $L = \{a^3 b^n c^n : n \geqslant 0\}$

(f) $L = \{a^n b^m : n \leqslant m \leqslant 3n\}$

(g) $L = \{w : n_a(w) = n_b(w) + 1\}$

(h) $L = \{w : n_a(w) = 2n_b(w)\}$

(i) $L = \{w : n_a(w) + n_b(w) = n_c(w)\}$

(j) $L = \{w : 2n_a(w) \leqslant n_b(w) \leqslant 3n_a(w)\}$

(k) $L = \{w : n_a(w) < n_b(w)\}$

5. 构造一个能够接受语言$L = \{a^n b^m : n \geqslant 0, n \neq m\}$的npda。

6. 找一个定义于$\Sigma = \{a, b, c\}$上的npda，它能够接受语言

$$L = \{w_1 c w_2 : w_1, w_2 \in \{a, b\}^*, w_1 \neq w_2^R\}$$

7. 找一个npda，它能够接受由$L(a^*)$与习题6中语言连接而得到的语言。

8. 为语言$L = \{ab(ab)^n b(ba)^n : n \geqslant 0\}$构造一个能够接受它的npda。

9. 是否存在一个dfa接受的语言与下面的pda所接受的语言相同

182

$$M = (\{q_0, q_1\}, \{a, b\}, \{z\}, \delta, q_0, z, \{q_1\})$$

其中

$$\delta(q_0, a, z) = \{(q_1, z)\}$$
$$\delta(q_0, b, z) = \{(q_0, z)\}$$
$$\delta(q_1, a, z) = \{(q_1, z)\}$$
$$\delta(q_1, b, z) = \{(q_0, z)\}$$

10. 什么语言能够被pda

$$M = (\{q_0, q_1, q_2, q_3, q_4, q_5\}, \{a, b\}, \{0, 1, a\}, \delta, q_0, z, \{q_5\})$$

接受？其中

$$\delta(q_0, b, z) = \{(q_1, 1z)\}$$
$$\delta(q_1, b, 1) = \{(q_1, 11)\}$$
$$\delta(q_2, a, 1) = \{(q_3, \lambda)\}$$
$$\delta(q_3, a, 1) = \{(q_4, \lambda)\}$$
$$\delta(q_4, a, z) = \{(q_4, z), (q_5, z)\}$$

11. 给出由npda $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{a, b, z\}, \delta, q_0, z, \{q_2\})$接受的语言，它的转移函数定义如下:

$$\delta(q_0, a, z) = \{(q_1, a), (q_2, \lambda)\}$$
$$\delta(q_1, b, a) = \{(q_1, b)\}$$

$$\delta(q_1, b, b) = \{(q_1, b)\}$$
$$\delta(q_1, a, b) = \{(q_2, \lambda)\}$$ ●

12. 针对例7.3中的npda，如果$F = \{q_0, q_f\}$，给出由该自动机接受的语言。
13. 针对习题11中的npda，如果$F = \{q_0, q_1, q_2\}$，给出由该自动机接受的语言。
14. 给出一个能够接受语言$L(aa^*ba^*)$的npda，且其内部状态不超过两个。●
15. 假设在例7.2中，我们用

$$\delta(q_2, \lambda, 0) = \{(q_0, \lambda)\}$$

替换$\delta(q_2, \lambda, 0)$给定的值，给出这个新pda接受的语言。

16. 我们可以定义一个受限制的npda，在每一次迁移中，它将至多在栈中增加一个符号，改变定义7.1，使得

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow 2^{Q \times (\Gamma\Gamma \cup \Gamma \cup \{\lambda\})}$$

也就是说，δ的值域由形如(q_i, ab), (q_i, a)或(q_i, λ)的对集合构成。证明对任意npda M都存在这样一个受限制的npda $\hat{M}$满足$L(M) = L(\hat{M})$。● 183

17. 对于自动机是如何接受语言的，除了定义7.2，还存在另一种形式的定义，这种定义要求：当符号串输入完毕时，栈为空。形式上，如果其满足如下条件，则npda可以以空栈方式接受语言$N(M)$。

$$N(M) = \{w \in \Sigma^* : (q_0, w, z) \vdash^*_M (p, \lambda, \lambda)\}$$

其中p是Q中的任意元素。证明这种表示方法与定义7.2是等价的，也就是说，对于任意的npda M，都存在一个npda $\hat{M}$与之等价使得$L(M) = N(\hat{M})$，反之亦然。

7.2 下推自动机与上下文无关语言

在前一节的例子中，我们可以看到，对于一些常见的上下文无关语言存在相应的下推自动机，这并不是偶然的。实际上，在上下文无关语言与非确定型下推接受器之间存在着普遍性的关系，也就是接下来要建立的两个主要结论。我们将证明，对于每一个上下文无关语言都存在一个npda能够接受它，反过来，任何npda接受的语言都是上下文无关语言。

7.2.1 上下文无关语言相应的下推自动机

我们将首先证明，对于每一个上下文无关语言都存在一个npda可以接受它。基本思想就是构造一个npda能够以某种方式对于该语言中任何符号串产生一个最左推导。为了对命题稍做简化，我们可以假定上下文无关语言是由格里巴克范式生成的。

将要构造的pda是采用如下方式来表示推导的：将句型右部变量压入栈中，而使完全由终结符组成的左部作为读输入。首先，我们将开始符压入栈中，然后为了模拟产生式$A \rightarrow ax$，必须使得这时的栈顶符号为变量A，输入符号为终结符a。模拟过程中，栈顶符号A将被变量串x替换。我们将很容易看出需要什么样的δ，才能得到这样的结果。在给出一般性的论证之前，我们首先来看一个简单的例子。

例7.5 构造一个pda能够接受具有如下产生式的文法生成的语言：

184

$$S \to aSbb|a$$

首先我们将该文法转换成格里巴克范式，得到的新产生式为:

$$S \to aSA|a$$
$$A \to bB$$
$$B \to b$$

相应的自动机将包含三个状态$\{q_0, q_1, q_2\}$，其中q_0为初态，q_2为终态。首先，开始符号S将通过转移函数

$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$$

被压入栈中。当从输入中读入a时，在pda中，通过将S替换为SA来模拟产生式$S \to aSA$。类似地，规则$S \to a$将导致pda读入一个a同时从栈顶简单地删除S。因此，这两个产生式在pda中表示如下

$$\delta(q_1, a, S) = \{(q_1, SA), (q_1, \lambda)\}$$

类似地，根据其他产生式将得到

$$\delta(q_1, b, A) = \{(q_1, B)\}$$
$$\delta(q_1, b, B) = \{(q_1, \lambda)\}$$

如果栈顶为开始符，则表明推导完成并通过

$$\delta(q_1, \lambda, z) = \{(q_2, \lambda)\}$$

将pda置于终态。

针对这一实例的构造方法同样适用于其他情况，可以得出一个普遍性的结论。 □

定理7.1 *对于任何的上下文无关语言L，存在一个npda M使得*

$$L = L(M)$$

证明: 如果L是一个无λ的上下文无关语言，则存在一个生成它且满足格里巴克范式的上下文无关文法。设$G=(V, T, S, P)$是这样的文法，根据它我们构造一个npda，它能模拟文法中的最左推导。如上所述，为了完成模拟，要使句型中未被处理的部分保留在栈内，同时使句型的由终结符组成的前缀与输入符号串相应的前缀匹配。

185

具体地，npda可表示为:

$$M = (\{q_0, q_1, q_f\}, T, V \cup \{z\}, \delta, q_0, z, \{q_f\})$$

其中$z \notin V$。注意M的输入字母表与文法G的终结符集合相同，栈字母表包括文法G中所有变量组成的集合。

其转移函数将包括

$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\} \tag{7-1}$$

它使得M在第一次转移后，栈中将包含推导的开始符S。(其中栈开始符z是一个标志，用于探测推导的结束。) 另外，针对P中形如

$$A \to au$$

的产生式，转移规则集合将满足

$$(q_1, u) \in \delta(q_1, a, A) \tag{7-2}$$

它读入a，并将栈顶的变量A替换成u。通过这种方法生成转移函数将允许pda能够模拟所有的推导。最后，我们将通过

$$\delta(q_1, \lambda, z) = \{(q_f, z)\} \tag{7-3}$$

使M进入终态。

为了证明M能够接受任意的$w \in L(G)$，考虑局部最左推导

$$\begin{aligned} S &\overset{*}{\Rightarrow} a_1a_2\cdots a_nA_1A_2\cdots A_m \\ &\Rightarrow a_1a_2\cdots a_nbB_1\cdots B_kA_2\cdots A_m \end{aligned}$$

如果M能够模拟这一推导，则在读入$a_1a_2\cdots a_n$后，栈中一定包含$A_1A_2\cdots A_m$。为了能进行下一步推导，G中必须包含产生式

$$A_1 \to bB_1\cdots B_k$$

而根据构造方法，可知M中存在转移规则，其中

$$(q_1, B_1\cdots B_k) \in \delta(q_1, b, A_1)$$

通过它，在读入$a_1a_2\cdots a_nb$后，栈中将包含$B_1\cdots B_kA_2\cdots A_m$。

对推导步骤的步数进行简单的归纳论证，证明如果

$$S \overset{*}{\Rightarrow} w$$

186

则

$$(q_1, w, Sz) \overset{*}{\vdash} (q_1, \lambda, z)$$

通过运用式（7-1）与式（7-3）有

$$(q_0, w, z) \vdash (q_1, w, Sz) \overset{*}{\vdash} (q_1, \lambda, z) \vdash (q_f, \lambda, z)$$

使得 $L(G) \subseteq L(M)$。

为了证明 $L(M) \subseteq L(G)$，设$w \in L(M)$，根据定义可得

$$(q_0, w, z) \overset{*}{\vdash} (q_f, \lambda, u)$$

但是由于从q_0到q_1以及从q_1到q_f都仅存在一条路径，因而必然存在

$$(q_1, w, Sz) \overset{*}{\vdash} (q_1, \lambda, z)$$

现在通过设$w = a_1a_2a_3\cdots a_n$，则

$$(q_1, a_1a_2a_3\cdots a_n, Sz) \overset{*}{\vdash} (q_1, \lambda, z) \tag{7-4}$$

中的第一步必然采用形如式（7-2）的规则以得到

$$(q_1, a_1a_2a_3\cdots a_n, Sz) \vdash (q_1, a_2a_3\cdots a_n, u_1z)$$

但另一方面，文法中存在形如$S \to a_1u_1$的规则，因而

$$S \Rightarrow a_1u_1$$

如此重复，设$u_1 = Au_2$，我们将得到

$$(q_1, a_2a_3\cdots a_n, Au_2z) \vdash (q_1, a_3\cdots a_n, u_3u_2z)$$

它表明在文法中存在$A \to a_2u_3$，因而也就可得

$$S \overset{*}{\Rightarrow} a_1a_2u_3u_2$$

这使得任一时刻栈的内容（z除外）与句型中没有匹配的部分是一致的，因此根据式（7-4）可得

$$S \overset{*}{\Rightarrow} a_1a_2\cdots a_n$$

结果有$L(M) \subseteq L(G)$，如果语言不包含λ，则证毕。

如果$\lambda \in L$，我们可以在待构造的npda中加入转移函数

$$\delta(q_0, \lambda, z) = \{(q_f, z)\}$$

187 使得空串也能够被接受。■

例7.6 考虑文法

$$S \to aA$$
$$A \to aABC|bB|a$$
$$B \to b$$
$$C \to c$$

由于该文法已经是格里巴克范式了，所以我们可以直接应用前面定理中的构造方法。除了规则

$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$$

和

$$\delta(q_1, \lambda, z) = \{(q_f, z)\}$$

pda还将包含转移规则

$$\delta(q_1, a, S) = \{(q_1, A)\}$$
$$\delta(q_1, a, A) = \{(q_1, ABC), (q_1, \lambda)\}$$
$$\delta(q_1, b, A) = \{(q_1, B)\}$$
$$\delta(q_1, b, B) = \{(q_1, \lambda)\}$$
$$\delta(q_1, c, C) = \{(q_1, \lambda)\}$$

通过使用M处理$aaabc$的迁移序列为

$$\begin{aligned}(q_0, aaabc, z) &\vdash (q_1, aaabc, Sz)\\ &\vdash (q_1, aabc, Az)\\ &\vdash (q_1, abc, ABCz)\\ &\vdash (q_1, bc, BCz)\end{aligned}$$

$$\vdash (q_1, c, Cz)$$
$$\vdash (q_1, \lambda, z)$$
$$\vdash (q_f, \lambda, z)$$

与之对应的推导是

$$S \Rightarrow aA \Rightarrow aaABC \Rightarrow aaaBC \Rightarrow aaabC \Rightarrow aaabc \qquad \square$$

为了简化论证，在定理7.1的证明中假定了文法是格里巴克范式，但这并不是必要的。对于一般性的上下文无关文法，我们同样可以类似地得到稍微复杂一点的构造方法。例如对形如 188

$$A \rightarrow Bx$$

的产生式，通过在栈中使用Bx替换A，但不读入任何输入符号来模拟用该产生式进行的推导；而对于形如

$$A \rightarrow abCx$$

的产生式，应首先将输入中的ab与栈中的相似符号串进行匹配，然后用Cx替换A。我们将这种形式的构造方法以及相关的证明留做习题。

7.2.2 下推自动机相应的上下文无关文法

定理7.1的逆命题也成立。相关的构造方法已经说明了这一点：颠倒定理7.1中的构造过程，使用文法模拟pda中的转移。这也就是说使句型中的变量部分反映栈的内容，而已经处理的输入作为句型的仅含终结符前缀。为了做到这点，我们只需稍做说明。

为了使讨论尽可能地简单，我们将假定问题中的npda满足如下要求：

1. 它只有一个终态q_f，且当且仅当栈为空时才进入终态；
2. 所有转移函数的形式应该是$\delta(q_i, a, A) = \{c_1, c_2, \cdots, c_n\}$，其中

$$c_i = (q_j, \lambda) \tag{7-5}$$

或

$$c_i = (q_j, BC) \tag{7-6}$$

也就是说，每一次迁移对栈进行的修改都是要么增加一个符号，要么减少一个符号。

上述限制看起来似乎有点严格，但实际上并不是这样的。可以证明对于任意的npda，都存在一个满足特性1与2的等价形式。7.1节的习题16与习题17就已经部分地阐述了这种等价形式。这里需要做进一步的阐述，但同样我们将其留做习题（本节习题16）。假定这是成立的，我们将为npda接受的语言构造上下文无关文法。

如前所述，我们要用句型表示栈的内容。但npda的格局还包含一个内部状态，而且这个状态又必须记录在句型中。对于这点，很难说明它是怎么完成的，下面我们给出的构造方法将会比较复杂。 189

假设现在我们可以找到一个文法使得其变量形如（q_iAq_j）而产生式为

$$(q_iAq_j) \overset{*}{\Rightarrow} v$$

则当且仅当在读到v并且从状态q_i向q_j转换时，npda擦除A。这里的“擦除”指的是将A以及A的影响（比如，所有替换它的后继符号串）从栈中删除，并使得原来A下面的符号被置于栈顶。如果我们能够找到这样一个文法，并以（q_0zq_f）作为它的开始符，则

$$(q_0zq_f) \overset{*}{\Rightarrow} w$$

当且仅当在读入符号串w并从q_0转换到q_f时，npda删除z（创建一个空栈）。这就是npda接受w的过程。因此，由文法生成的语言与npda接受的语言是相同的。

为了构造一个文法能够满足这些条件，我们将检查由npda执行的不同类型的转移函数。由于式（7-5）包含对A的立即擦除，所以文法中存在一个对应的产生式

$$(q_iAq_j) \to a$$

式（7-6）类型的产生式将生成如下的规则集合

$$(q_iAq_k) \to a(q_jBq_l)(q_lCq_k)$$

其中q_k和q_l可取Q中所有可能的值。这是由于为了擦除A，当读到a并且从q_i到q_j转换时，我们首先需要用BC替换A。接下来状态从q_j转换到q_l并擦除B，然后从q_l转换到q_k并擦除C。

在最后一步中，我们似乎加入了太多产生式，这是由于在擦除B时，可能存在一些无法从q_j达到的状态q_l。的确如此，但是这并不影响该文法。因为即使得到的结果文法中变量（q_jBq_l）是无用的，也不会影响文法接受的语言。

190 最后，我们以（q_0zq_f）作为文法的开始变量，其中q_f是npda的唯一终态。

例7.7 考虑npda，它的转移函数如下

$$\delta(q_0, a, z) = \{(q_0, Az)\}$$
$$\delta(q_0, a, A) = \{(q_0, A)\}$$
$$\delta(q_0, b, A) = \{(q_1, \lambda)\}$$
$$\delta(q_1, \lambda, z) = \{(q_2, \lambda)\}$$

用q_0作为初态，q_2作为终态，该npda满足前述条件1但并不满足条件2。为了满足后者，我们引入了一个新的状态q_3和一个中间步骤，在该步骤中，我们首先从栈中删除A，然后在下一次迁移中替换它。新的转换规则集合为

$$\delta(q_0, a, z) = \{(q_0, Az)\}$$
$$\delta(q_3, \lambda, z) = \{(q_0, Az)\}$$
$$\delta(q_0, a, A) = \{(q_3, \lambda)\}$$
$$\delta(q_0, b, A) = \{(q_1, \lambda)\}$$
$$\delta(q_1, \lambda, z) = \{(q_2, \lambda)\}$$

后三个转移函数符合式（7-5）的形式，因而可以直接得到相应的产生式

$$(q_0Aq_3) \to a, \quad (q_0Aq_1) \to b, \quad (q_1zq_2) \to \lambda$$

根据前两个转移函数，可以得到产生式集合

$$(q_0zq_0) \to a(q_0Aq_0)(q_0zq_0) | a(q_0Aq_1)(q_1zq_0) |$$
$$a(q_0Aq_2)(q_2zq_0) | a(q_0Aq_3)(q_3zq_0)$$

$$
\begin{aligned}
(q_0zq_1)&\to a(q_0Aq_0)(q_0zq_1)|a(q_0Aq_1)(q_1zq_1)|\\
&\quad a(q_0Aq_2)(q_2zq_1)|a(q_0Aq_3)(q_3zq_1)\\
(q_0zq_2)&\to a(q_0Aq_0)(q_0zq_2)|a(q_0Aq_1)(q_1zq_2)|\\
&\quad a(q_0Aq_2)(q_2zq_2)|a(q_0Aq_3)(q_3zq_2)\\
(q_0zq_3)&\to a(q_0Aq_0)(q_0zq_3)|a(q_0Aq_1)(q_1zq_3)|\\
&\quad a(q_0Aq_2)(q_2zq_3)|a(q_0Aq_3)(q_3zq_3)
\end{aligned}
$$

$$
\begin{aligned}
(q_3zq_0)&\to(q_0Aq_0)(q_0zq_0)|(q_0Aq_1)(q_1zq_0)|(q_0Aq_2)(q_2zq_0)|(q_0Aq_3)(q_3zq_0)\\
(q_3zq_1)&\to(q_0Aq_0)(q_0zq_1)|(q_0Aq_1)(q_1zq_1)|(q_0Aq_2)(q_2zq_1)|(q_0Aq_3)(q_3zq_1)\\
(q_3zq_2)&\to(q_0Aq_0)(q_0zq_2)|(q_0Aq_1)(q_1zq_2)|(q_0Aq_2)(q_2zq_2)|(q_0Aq_3)(q_3zq_2)\\
(q_3zq_3)&\to(q_0Aq_0)(q_0zq_3)|(q_0Aq_1)(q_1zq_3)|(q_0Aq_2)(q_2zq_3)|(q_0Aq_3)(q_3zq_3)
\end{aligned}
$$

191

其中开始变量为（q_0zq_2）。符号串aab通过如下的连续格局能够被pda接受

$$
\begin{aligned}
(q_0, aab, z) &\vdash (q_0, ab, Az)\\
&\vdash (q_3, b, z)\\
&\vdash (q_0, b, Az)\\
&\vdash (q_1, \lambda, z)\\
&\vdash (q_2, \lambda, \lambda)
\end{aligned}
$$

而它在G中相应的推导为

$$
\begin{aligned}
(q_0zq_2) &\Rightarrow a(q_0Aq_3)(q_3zq_2)\\
&\Rightarrow aa(q_3zq_2)\\
&\Rightarrow aa(q_0Aq_1)(q_1zq_2)\\
&\Rightarrow aab(q_1zq_2)\\
&\Rightarrow aab
\end{aligned}
$$

如果你注意到了pda的连续瞬时描述与推导中句型间的对应关系，就更易于理解下面定理的证明步骤。其中每个句型最左变量中的第一个q_i是pda的当前状态，而中间的符号序列与栈内容相同。

虽然这一构造方法得到的是一个非常复杂的文法，但是它可以应用于任何pda，只要该pda的转移规则满足给定条件。同时，这也为一般性结论的证明提供了基础。 □

定理7.2 对于任何一个npda M，如果$L=L(M)$，则L是上下文无关语言。

证明：假定$M=(Q, \Sigma, \Gamma, \delta, q_0, z, \{q_f\})$满足前述条件1与条件2，我们使用上面的构造方法得到文法$G=(V, T, S, P)$，其中$T=\Sigma$，V由形如（q_icq_j）的元素组成。我们将证明这样得到的文法满足如下条件，对于所有的$q_i, q_j\in Q, A\in\Gamma, X\in\Gamma^*, u, v\in\Sigma^*$，根据

$$(q_i, uv, AX)\overset{*}{\vdash}(q_j, v, X) \tag{7-7}$$

可得

$$(q_iAq_j)\overset{*}{\Rightarrow}u$$

反之亦然。

第一部分证明的是，npda任何时候都满足在读到u并从状态q_i到q_j转换时，都能将符号A及

192 其影响从栈中删除，则变量(q_iAq_j)就能导出u。显而易见，文法的构造就是用于完成这一工作的。我们只需对迁移的步数做归纳就能清楚地说明这一点。

针对逆命题，考虑推导中单独的一步，比如

$$(q_iAq_k)\Rightarrow a(q_jBq_l)(q_lCq_k)$$

对npda使用相应的转移函数

$$\delta(q_i, a, A) = \{(q_j, BC), \cdots\} \tag{7-8}$$

我们可以看到在控制部件读入a并从状态q_i迁移到q_j时，可以从栈中删除A，压入BC。类似地，如果

$$(q_iAq_j)\Rightarrow a \tag{7-9}$$

则必定存在相应的转移函数

$$\delta(q_i, a, A) = \{(q_j, \lambda)\} \tag{7-10}$$

通过它，可以从栈中弹出A。从这可以看出，由q_iAq_j导出的句型定义了npda的一个可能的格局序列，通过它也就可以得到式（7-7）。

注意某些q_jBq_l，q_lCq_k不存在式（7-8）或式（7-10）形式的转移函数，但可能存在$(q_iAq_j)\Rightarrow a(q_jBq_l)(q_lCq_k)$。但在这种情况中，至少有一个右部的变量是没用的。而由于所有的句型都将有一个终结符串，因此命题成立。

如果我们现在将结论应用到

$$(q_0, w, z)\overset{*}{\vdash}(q_f, \lambda, \lambda)$$

将看到上式成立当且仅当

$$(q_0zq_f)\overset{*}{\Rightarrow}w$$

因而$L(M) = L(G)$。■

习题

1. 证明例7.5中构造的pda能够接受由给定文法生成的语言中的符号串$aaabbbb$。

193 2. 证明例7.5中的pda能接受语言$L = \{a^{n+1}b^{2n} : n \geqslant 0\}$。

3. 构造一个npda能够接受由如下文法生成的语言:

$$S\rightarrow aSbb|aab$$

4. 构造一个npda能够接受由文法$S\rightarrow aSSS|ab$生成的语言。

5. 构造文法

$$S\rightarrow aABB|aAA$$
$$A\rightarrow aBB|a$$
$$B\rightarrow bBB|A$$

相应的npda。

6. 构造一个npda能够接受由文法$G = (\{S, A\}, \{a, b\}, S, P)$生成的语言，其中产生式为$S \to AA|a$，$A \to SA|b$。
7. 证明根据定理7.1与定理7.2可以得到：对于任意npda $\hat{M}$，都存在一个至多包含三个状态的npda $\hat{M}$满足$L(M) = L(\hat{M})$。
8. 说明怎样才能将上面习题中$\hat{M}$的三个状态简化成两个。
9. 给出一个只有两个状态的npda接受语言$L = \{a^n b^{n+1} : n \geqslant 0\}$。
10. 给出一个只有两个状态的npda接受语言$L = \{a^n b^{2n} : n \geqslant 1\}$。
11. 证明例7.7中的npda能够接受语言$L(aa*b)$。
12. 证明例7.7中的文法能够生成语言$L(aa*b)$。
13. 证明例7.7中的变量（q_0Bq_0）与（q_0zq_1）是无用的。
14. 根据定理7.1中的构造方法，构造npda能够接受7.1节中例7.5的语言。
15. 给出一个上下文无关文法，能够生成由npda $M = (\{q_0, q_1\}, \{a, b\}, \{A, z\}, \delta, q_0, z, \{q_1\})$接受的语言。它的转移函数如下

$$\delta(q_0, a, z) = \{(q_0, Az)\}$$
$$\delta(q_0, b, A) = \{(q_0, AA)\}$$
$$\delta(q_0, a, A) = \{(q_1, \lambda)\}$$

16. 说明对于任意npda，都存在一个等价的形式满足定理7.2导引中的条件1和条件2。
17. 给出定理7.2的完整证明。
18. 给出一个构造方法，使得任意上下文无关文法都能用于定理7.1的证明。 194

7.3 确定型下推自动机和确定型上下文无关语言

确定型下推接受器（deterministic pushdown accepter, dpda）是没有迁移选择的下推自动机。它可以通过修改定义7.1得到。

定义7.3 一个下推自动机$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$称为确定型的，如果它是根据定义7.1定义的自动机并且满足如下的限制条件：对于任意$q \in Q$, $a \in \Sigma \cup \{\lambda\}$，以及$b \in \Gamma$，有

1. $\delta(q, a, b)$最多包含一个元素；
2. 如果$\delta(q, \lambda, b)$非空，则对于每个$c \in \Sigma$，$\delta(q, c, b)$都必须为空。

第一个条件要求对于任意给定的输入符号与栈顶符号，最多只能执行一种迁移。第二个条件说明，如果对某一格局存在λ迁移，则不能有读入输入符号的迁移。

我们来看一下这个定义与确定型有穷自动机定义的区别。转移函数仍采用定义7.1中的定义域而不是$Q \times \Sigma \times \Gamma$，这是因为我们想保留$\lambda$转移。由于栈顶符号在决定下一次迁移中发挥作用，因而λ转移的存在并不必然导致非确定型。同时，dpda的一些转移函数也允许为空集，也就是说它们没有定义，因而可能存在死格局。但这并不对确定型的定义造成影响；因为对确定型的唯一标准是：任何时候都至多存在一种可能的迁移。

定义7.4 语言L是确定型上下文无关语言（deterministic context-free language）当且仅当存在一个dpda M满足$L = L(M)$。 195

例7.8 语言

$$L = \{a^n b^n : n \geqslant 0\}$$

是确定型上下文无关语言。由于pda $M = (\{q_0, q_1, q_2\}, \{a, b\}, \{0, 1\}, \delta, q_0, 0, \{q_0\})$ 能够接受语言L，其中M的产生式为

$$\delta(q_0, a, 0) = \{(q_1, 10)\}$$
$$\delta(q_1, a, 1) = \{(q_1, 11)\}$$
$$\delta(q_1, b, 1) = \{(q_2, \lambda)\}$$
$$\delta(q_2, b, 1) = \{(q_2, \lambda)\}$$
$$\delta(q_2, \lambda, 0) = \{(q_0, \lambda)\}$$

并且M满足定义7.4中的条件，所以L是确定型的。 □

例7.4中的npda不是确定型的，因为

$$\delta(q_0, a, a) = \{(q_0, aa)\}$$

和

$$\delta(q_0, \lambda, a) = \{(q_1, a)\}$$

与定义7.3中的条件2冲突。当然，这并不表明语言$\{ww^R\}$本身一定是非确定型的，因为可能存在一个等价的dpda。但事实上，该语言确实是非确定型的。从这个例子和下一个要举的例子中，我们将看到，与有穷自动机不同的是确定型下推自动机与非确定型下推自动机不是等价的。因为存在非确定型上下文无关语言。

例7.9 设

$$L_1 = \{a^n b^n : n \geqslant 0\}$$

和

$$L_2 = \{a^n b^{2n} : n \geqslant 0\}$$

其中L_1是一个上下文无关语言，通过对其证明做一些明显的修改，可证明L_2也是上下文无关语言。语言$L = L_1 \cup L_2$同样是上下文无关的。这个结论可以从下一章中的一个一般性定理得到，但现在只能简单地说明它似乎是合理的。设$G_1 = (V_1, T, S_1, P_1)$ 和$G_2 = (V_2, T, S_2, P_2)$ 是上下文无关文法，满足$L_1 = L(G_1)$ 和$L_2 = L(G_2)$ ，如果我们假定V_1与V_2是不相交的，并且$S \notin V_1 \cup V_2$，则合并它们可得文法$G = (V_1 \cup V_2 \cup \{S\}, T, S, P)$，其中

196

$$P = P_1 \cup P_2 \cup \{S \to S_1 | S_2\}$$

生成语言$L_1 \cup L_2$。讲到这里证明思路已经相当清楚了，但我们在第8章中才会给出论证的详细内容，由上述，可得L是上下文无关的。但是，L不是确定型上下文无关语言，这看起来是合理的，因为pda必须根据每个a匹配一个或者两个b，所以不论输入的是L_1还是L_2，它都需要做出一个初始选择，而在符号串的开始部分没有足够的信息可以用于确定型地做出哪一种选择。当然，这种推理基于我们了解的某个特定算法；它有可能让我们得到正确的推测，但并不能证明任何事情。而且总是有可能存在一种完全不同的方法可以避免初始选择，但上述语言的确不存在这样的方法，因而L是非确定型的。为了说明这一问题，我们首先假定：L是一个确定型上下文无关语言，则

$$\hat{L} = L \cup \{a^n b^n c^n : n \geqslant 0\}$$

也是上下文无关的。我们将通过在给定L的dpda M的情况下，构造$\hat{L}$的npda $\hat{M}$来进行说明。

构造的方法是：在M的控制部件中增加一个类似的部件，把由b引起转移改为由c引起转移。在M读入$a^n b^n$后，新部件将在控制部件中发挥作用。由于第二个部件对c^n的响应与第一个部件对b^n的响应是相同的，所以处理$a^n b^{2n}$的过程同样也能够接受$a^n b^n c^n$。图7-2生动地描述了这一构造。下面给出形式化的证明。

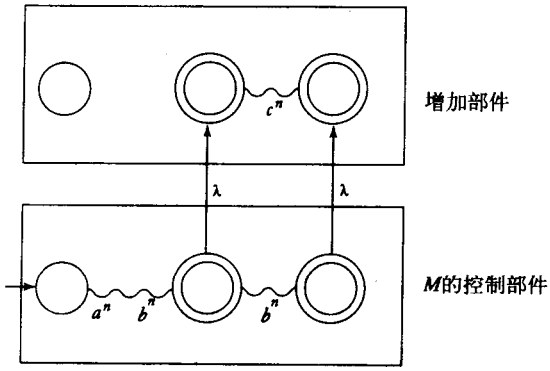


图 7-2

设$M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$，其中

$$Q = \{q_0, q_1, \cdots, q_n\}$$

然后考虑$\hat{M} = (\hat{Q}, \Sigma, \Gamma, \delta \cup \hat{\delta}, z, \hat{F})$，其中

$$\hat{Q} = Q \cup \{\hat{q}_0, \hat{q}_1, \cdots, \hat{q}_n\}$$
$$\hat{F} = F \cup \{\hat{q}_i : q_i \in F\}$$

并通过如下方式在δ中加入转移函数构造$\hat{\delta}$。首先对所有$q_f \in F, s \in \Gamma$，在$\hat{\delta}$中加入

$$\hat{\delta}(q_f, \lambda, s) = \{(\hat{q}_f, s)\}$$

然后对所有

$$\delta(q_i, b, s) = \{(q_j, u)\}$$

其中，$q_i \in Q, s \in \Gamma, u \in \Gamma^*$，在$\hat{\delta}$中加入

$$\hat{\delta}(\hat{q}_i, c, s) = \{(\hat{q}_j, u)\}$$

因为M能够接受$a^n b^n$，所以我们一定能得到

$$(q_0, a^n b^n, z) \vdash^*_M (q_i, \lambda, u)$$

其中$q_i \in F$。由于M是确定型的，所以

$$(q_0, a^n b^{2n}, z) \vdash^*_M (q_i, b^n, u)$$

也必然成立，使得M能够接受a^nb^{2n}，这样，我们就能进一步地得到

$$(q_i, b^n, u) \vdash_M^* (q_j, \lambda, u_1)$$

其中$q_j \in F$。然后通过构造

$$(\hat{q}_i, c^n, u) \vdash_{\hat{M}}^* (\hat{q}_j, \lambda, u_1)$$

使得$\hat{M}$能够接受$a^nb^nc^n$。除此之外，仍需要说明$\hat{M}$不能接受非$\hat{L}$中的符号串；这将在本节的几个习题中阐述。据此可得结论$\hat{L} = L(\hat{M})$，因而$\hat{L}$是上下文无关的。但是在下一章（例8.1）中我们将得到$\hat{L}$不是上下文无关的。因此，对L是确定型上下文无关语言的这一假设是错误的。 □

197 ~ 198

习题

1. 证明$L = \{a^nb^{2n} : n \geqslant 0\}$是确定型上下文无关语言。
2. 证明$L = \{a^nb^m : m \geqslant n + 2\}$是确定型的。
3. 语言$L = \{a^nb^n : n \geqslant 1\} \cup \{b\}$是否是确定型的？
4. 例7.2中的语言$L = \{a^nb^n : n \geqslant 1\} \cup \{a\}$是否是确定型的？ ●
5. 证明例7.3中的下推自动机不是确定型的，但该例中的语言是确定型的。
6. 针对习题1中的语言L，说明L^*是确定型上下文无关语言。
7. 给定语言

$$L = \{a^nb^mc^k : n = m \text{或} m = k\}$$

为什么能推测出它是非确定型的，试给出理由。

8. 语言$L = \{a^nb^m : n = m \text{或} n = m + 2\}$是否是确定型的？
9. 语言$L = \{wcw^R : w \in \{a, b\}^*\}$是否是确定型的？ ●
10. 如果习题9中的语言是确定型的，而与之密切相关的语言$L = \{ww^R : w \in \{a, b\}^*\}$却是非确定型的，针对这一问题，给出合理的解释。
11. 证明语言$L = \{w \in \{a, b\}^* : n_a(w) \neq n_b(w)\}$是确定型上下文无关语言。 ●
12. 证明例7.9中的$\hat{M}$不能接受$a^nb^nc^k$，其中$k \neq n$。
13. 证明例7.9中的$\hat{M}$不能接受$L(a^*b^*c^*)$以外的符号串。
14. 证明例7.9中的$\hat{M}$不能接受$a^nb^{2n}c^k$（其中$k > 0$）。证明只有当$m = n$或$m = 2n$并且$k = 0$时，它才能接受$a^nb^mc^k$。
15. 证明所有的正则语言都是确定型上下文无关语言。 ●
16. 如果L_1是确定型上下文无关语言，而L_2是正则语言，则$L_1 \cup L_2$是确定型上下文无关语言。 ●
17. 如果L_1是确定型上下文无关语言，而L_2是正则语言，则$L_1 \cap L_2$是确定型上下文无关语言。
18. 给出一个确定型上下文无关语言的实例，而其转置不是确定型的。

199

7.4 确定型上下文无关语言的文法*

由于确定型上下文无关语言能够被有效地分析，因而它在实际应用中显得特别重要。在直观上，我们可以将下推自动机看成一个语法分析设备。由于不存在回溯，因而很容易写出

与它相应的计算机程序，并期望它能有效地工作。但由于λ转移的存在，我们不能立刻得出结论：它是一个线性时间的语法分析器。但无论如何，它已经把我们引入了正确的途径。为了实现这一目标，我们将关注什么形式的文法适用于描述确定型上下文无关语言。这里我们将进入编译器研究的一个重要主题，但这不是我们的兴趣所在。因此本节只简单介绍了其中的一些重要结论，更多的内容可参见与编译器相关的书籍。

假定我们自顶向下地进行语法分析，并试图为特定的句子找到其最左推导。为了便于讨论，将采用图7-3中阐述的方法。我们从左至右地扫描输入符号串w，当到达某一句型时，它的终结符前缀将与截止到当前所扫描符号的w前缀匹配。为了能够继续匹配后续符号，我们需要确切地知道在每一步中应当用哪一条产生式规则，这将避免回溯从而得到高效的语法分析器。现在的问题在于是否存在这样的文法允许我们这么做。对于一般性的上下文无关文法，这可能并不成立，但如果上下文无关文法的形式是受限的，我们就能够达到这一目标。

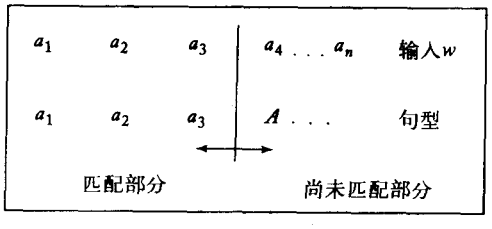


图 7-3

首先，考虑定义5.4中介绍的简单文法。根据其讨论，我们可以很清楚地看到在分析的每一步骤中，它都能够准确地知道使用了哪一个产生式。假定$w = w_1w_2$并且已经到达了句型w_1Ax，为了得到与w中下一个符号匹配的句型中的下一个符号，我们只需要看w_2的最左符号，比如说a。如果在文法中不存在形如$A \to ay$的规则，则符号串w不属于该语言，如果存在这样的规则，则分析过程可以继续。在这里，它只存在一个这样的规则，所以没有其他的选择。 200

尽管简单文法是有用的，但它太严格了，以至于不能描述出程序设计语言的所有语法现象。我们需要将这个思想一般化，使得它更为强大而又不失语法分析的基本性能。*LL文法*（LL grammar）即是这样的一种文法。在*LL*文法中，我们仍然保持在输入中看有限个字符（由扫描的符号加上跟在它后面的有限个符号组成），即可准确预测使用哪一个产生式的特性。在有关编译器的书中，术语*LL*是标准用法；第一个*L*表明输入采用的是从左至右的扫描方式，第二个*L*表明采用的是最左推导。每个简单文法都是*LL*文法，但是*LL*文法更具一般性。

例7.10 文法

$$S \to aSb|ab$$

不是简单文法，但它是*LL*文法。为了确定使用哪个产生式，我们只需看输入符号串中两个连续的符号。如果第一个是a第二个是b，我们就必须使用产生式$S \to ab$，否则就必须采用规则$S \to aSb$。 □

一个文法，如果给定当前输入的符号以及“向前看”$k-1$个符号，我们就能够唯一地确定使用哪一个产生式，则该文法是$LL(k)$文法。例7.10中的文法是$LL(2)$文法。

例7.11 文法

$$S \to SS|aSb|ab$$

生成例7.10中语言的正闭包，正如例5.4中所说的那样，这是一个描述正确嵌套括号结构的语言。该文法对任意k都不是$LL(k)$ 文法。

为了说明这一原因，让我们来看长度大于2符号串的推导。一开始，我们有两个可能的产生式$S \to SS$和$S \to aSb$作为选择，但是根据当前扫描到的符号并不能确定使用哪个产生式。假定我们采用向前看的方式，并且考虑前两个符号，如果它们是aa，是否就能据此做出正确的决定呢？答案仍是否定的，因为它可以是该语言中多个符号串的的前缀，比如$aabb$，$aabbab$。在第一种情况下，我们应该必须选择$S \to aSb$，而第二种情况则必须使用$S \to SS$。因此该文法不是LL（2）文法。采用同样的方式，我们可以看到无论向前看多少个符号，总是存在一些无法解决的情况。

201

上述情形中的文法并不能说明该语言就是非确定型的或者不存在产生该语言的LL文法。如果我们能分析出原文法不满足LL文法的原因，就可以为该语言构造一个LL文法。该问题的难度在于直到符号串的末尾，我们才能够知道到底存在多少个重复的基本模式$a^n b^n$，然而文法需要立刻对此做出决定。我们可以通过重写文法来解决这一问题，文法

$$S \to aSbS|\lambda$$

是一个LL文法，它与原文法几乎是等价的。

为了说明这一点，考虑$w = abab$的最左推导，可得

$$S \Rightarrow aSbS \Rightarrow abS \Rightarrow abaSbS \Rightarrow ababS \Rightarrow abab$$

从中可以看到我们没有任何其他的选择，如果输入的符号是a，我们就必须使用产生式$S \to aSbS$，如果输入的符号是b，或者这时已经达到符号串的末尾，就必须使用$S \to \lambda$。

到现在为止，问题还没有完全解决，因为新的文法能够产生空串。我们通过引入一个新的开始变量S_0以确保仅产生非空符号串的产生式来解决这一问题。这样将得到最终结果

$$S_0 \to aSbS$$
$$S \to aSbS|\lambda$$

它是LL文法并且与原文法是完全等价的。 □

虽然上面对LL文法的非形式化描述对于理解简单的例子已经足够了，但是为了得到严格的结论，我们需要一个更准确的LL文法定义。我们以下面的定义来结束讨论。

定义7.5 设$G = (V, T, S, P)$是一个上下文无关文法，如果对每个最左推导对

$$S \overset{*}{\Rightarrow} w_1 A x_1 \Rightarrow w_1 y_1 x_1 \overset{*}{\Rightarrow} w_1 w_2$$
$$S \overset{*}{\Rightarrow} w_1 A x_2 \Rightarrow w_1 y_2 x_2 \overset{*}{\Rightarrow} w_1 w_3$$

其中$w_1, w_2, w_3 \in T^*$，w_2和w_3最左的k个符号的等同性意味着$y_1 = y_2$，则称G为$LL(k)$ 文法。（如果$|w_2|$或$|w_3|$的长度小于k，则取它们的最小长度代替k。）

202

通过定义，我们可以清楚地看到，在最左推导（w_1Ax）的任意阶段，如果我们能够知道输入的下k个字符，则推导的下一步是唯一确定的（如同$y_1 = y_2$所表示的）。

LL文法是编译器学习的一个重点，很多程序设计语言都可以采用LL文法定义，很多编译器也是采用LL分析器写的，但是LL文法还不够一般化，不能处理所有的确定型上下文无关语言，因此要关注其他更具一般性的确定型文法。LR文法就是其中特别重要的一种，它同样支

持有效的语法分析，但采用的是自底向上的推导树构造方式，在关于编译器（例如，Hunter 1981）或者形式语言语法分析方法（例如Aho和Ullman 1972）的书籍中存在大量关于这一主题的资料。

习题

1. 证明例7.11中的第二个文法是LL文法并且它与原始文法是等价的。
2. 证明例1.13中的产生语言$L = \{w : n_a(w) = n_b(w)\}$的文法不是$LL$文法。
3. 为习题2中的语言构造一个LL文法。
4. 为语言$L(a^*ba) \cup L(abbb^*)$ 构造相应的LL文法。
5. 证明任意的LL文法都是无二义性的。
6. 证明如果G是$LL(k)$ 文法，则$L(G)$ 是确定型上下文无关语言。
7. 证明一个确定型上下文无关语言不是固有二义性的。
8. 设G是一个上下文无关文法，且符合格里巴克范式。设计一个算法，对于任意给定的k，它都能够确定文法G是不是$LL(k)$ 文法。
9. 给出下列语言的LL文法，其中$\Sigma = \{a, b, c\}$。
   (a) $L = \{a^n b^m c^{n+m} : n \geq 0, m \geq 0\}$
   (b) $L = \{a^{n+2} b^m c^{n+m} : n \geq 0, m \geq 0\}$
   (c) $L = \{a^n b^{n+2} c^m : n \geq 0, m > 1\}$
   (d) $L = \{w : n_a(w) < n_b(w)\}$
   (e) $L = \{w : n_a(w) + n_b(w) \neq n_c(w)\}$

第8章 上下文无关语言的性质

上下文无关语言族在形式语言体系中占有首要的位置。一方面，上下文无关语言本身包括了一些重要的受限语言族，比如正则语言与确定型上下文无关语言。另一方面，上下文无关语言又是更大的语言族的一个特例。为了理清不同语言族之间的关系以及展现它们的相似性与区别，我们将研究不同语言族的特性。像第4章一样，我们要考察语言在各种运算下的封闭性、用于确定语言族中成员性质的算法以及诸如泵引理这样的结构性结论。所有这些为我们理解不同语言族之间的关系提供了途径，同时它们也有助于我们将某一特定语言归入合适的族中。

8.1 两个泵引理

定理4.8给出的泵引理为证明某些语言不是正则语言提供了一种有效手段。对其他语言族也存在着类似的泵引理。这里我们将讨论两个这样的结论，一个是针对一般性的上下文无关语言的，另一个是针对一种受限形式的上下文无关语言的。

8.1.1 上下文无关语言的泵引理

定理8.1 *设L是一个无穷上下文无关语言，则存在一个正整数m使得对于任意满足$|w| \geq m$的$w \in L$都能够被分解为*

$$w = uvxyz \tag{8-1}$$

其中

$$|vxy| \leq m \tag{8-2}$$

且

$$|vy| \geq 1 \tag{8-3}$$

则对于所有的$i = 0, 1, 2, \cdots$，满足

$$uv^ixy^iz \in L \tag{8-4}$$

这就是上下文无关语言的泵引理。

证明：考虑语言$L - \{\lambda\}$，设生成它的文法为G，并且G没有单位产生式与λ产生式。由于产生式右部符号串长度总是有界的，比如设界为k，则任意$w \in L$的推导长度至少为$|w|/k$。而由于L是无穷的，所以它存在任意长的推导以及对应的任意高度的推导树。

现在来看一个具备这样高度的推导树以及某个从根到叶的足够长的路径。由于G中的变量是有限的，因而必然会在该路径中存在重复出现的变量，如图8-1所示。与图8-1中推导树对应的推导为

$$S \overset{*}{\Rightarrow} uAz \overset{*}{\Rightarrow} uvAyz \overset{*}{\Rightarrow} uvxyz$$

其中u, v, x, y以及z都是终结符串。从上可得$A \overset{*}{\Rightarrow} vAy$和$A \overset{*}{\Rightarrow} x$，因而所有的符号串$uv^ixy^iz$, $i = 0$, 1, 2, …，都能够根据文法生成，因此它们也属于L。此外，可以假定推导$A \overset{*}{\Rightarrow} vAy$和$A \overset{*}{\Rightarrow} x$中没有重复变量（如果有的话，我们就以它作为$A$），因此符号串$v$, x与y的长度仅与文法中的产生式有关，且它们受到w长度的限制，因而式（8-2）成立。最后由于不存在单位产生式与λ产生式，所以v与y不能同时为空，则可得式（8-3）。

206

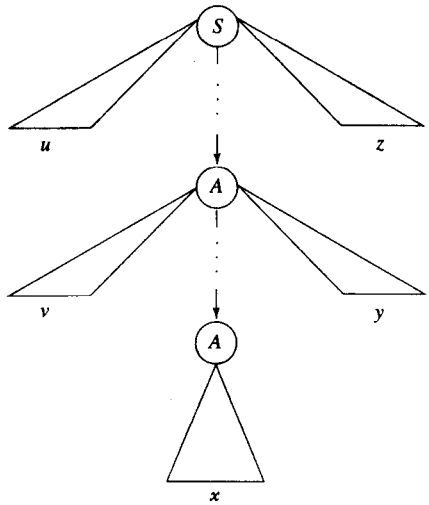


图8-1 长符号串的推导树

这样我们就证明了式（8-1）至式（8-4）都成立。■

该泵引理在证明一个语言不属于上下文无关语言族方面非常有用。一般而言，这也是泵引理的典型应用；它们用于确定一个给定语言不属于某一族。如定理4.8所说，正确的证明过程可以看成是一个与智能对手的游戏，只是现在的规则变得更难了一些。对于正则语言，长度不超过m的子串xy是从w的左端开始的，因此能够抽取的子串y也就在从w左端开始的m个符号中。而对于上下文无关语言，我们仅限制$|vxy|$的上界。vxy之前的子串u可以是任意长度的，这也就给予了对手更多的自由，同时也使得与定理8.1相关的证明更加复杂。

例8.1 证明语言

$$L = \{a^nb^nc^n : n \geqslant 0\}$$

207

不是上下文无关的。

一旦对手选定m，我们就挑选L中的符号串$a^mb^mc^m$。现在对手存在几种可能的选择。如果他选定的vxy中只包含a，则抽取得到的符号串显然不在L中；如果它选定的vxy包含相同数目的a与b，则抽取得到的符号串为$a^kb^kc^m$，其中$k \neq m$，同样它也不在L中。事实上，对手唯一可以阻止我们成功的办法就是选取vxy，使得vy有同样多的a，b以及c。但根据式（8-2）的限制，这是不可能的。因此，L不是上下文无关的。

但如果对语言$L = \{a^nb^n\}$采用同样的证明方法，它将不起作用，因为L是上下文无关的。如果我们选取L中诸如$w = a^mb^m$的任意符号串，对手就可以选取$v = a^k$与$y = b^k$。那么，不管我

们选取什么样的i，抽取得到的结果符号串w_i总是属于L的。要记住，虽然如此，但是这并不能证明L就是上下文无关的，对此我们只能说，这里我们通过泵引理不能得到任何结论。而要证明L是上下文无关语言，需要采用其他的证明方法，比如构造产生L的上下文无关文法。

上述证明验证了例7.9中给出的命题，并使我们对那个例子的理解能够连贯起来。语言

$$\hat{L}=\{a^nb^n\}\cup\{a^nb^{2n}\}\cup\{a^nb^nc^n\}$$

不是上下文无关的。这是因为符号串$a^mb^mc^m$在$\hat{L}$中，但其抽取的结果不在$\hat{L}$中。 □

例8.2 考虑语言

$$L=\{ww : w\in\{a,b\}^*\}$$

虽然上述语言与例5.1中的上下文无关语言十分类似，但它不是上下文无关语言。

考虑符号串

$$a^mb^ma^mb^m$$

现在对手存在多种选取vxy的方式，但是对于他的每一个选择，我们都有获胜的对抗方法。比如，采用图8-2中的选择，我们就可以使用$i=0$来得到符号串

$$a^kb^ja^mb^m,\ k<m \text{ 或 } j<m$$

它不在L中。对于对手的其他选择，我们可以用类似的论证方法获胜。因此可得：L不是上下文无关语言。 □ 208

例8.3 证明$L=\{a^{n!} : n\geqslant 0\}$不是上下文无关语言。

在例4.11中，我们已经证明了该语言不是正则的。然而，对于字母表只有一个符号的语言，在定理8.1与正则语言的泵引理之间几乎是不存在区别的。无论何种情况，由于抽取的符号串都是由a组成的，因而能够通过定理8.1得到的符号串，通过定理4.8同样也可以得到。因此，我们可以使用例4.11中的方法来证明L不是上下文无关语言。 □

例8.4 证明$L=\{a^nb^j : n=j^2\}$不是上下文无关语言。

根据定理8.1，给定m，我们可以选取$a^{m^2}b^m$作为待考察的符号串。现在对手有几种可能的选择，而只有图8-3中给出的选择需要更多思考。抽取i次将得到的新符号串有$m^2+(i-1)k_1$个a与$m+(i-1)k_2$个b。如果对手选取$k_1\neq 0$，$k_2\neq 0$，我们就可令$i=0$。由于

$$\begin{aligned}(m-k_2)^2&\leqslant(m-1)^2\\&=m^2-2m+1\\&<m^2-k_1\end{aligned}$$

因此结果不属于L。而如果对手选取$k_1=0$，$k_2\neq 0$或者$k_1\neq 0$，$k_2=0$，那么同样可令$i=0$，抽取得到的符号串也不在L中。至此可得L不是上下文无关语言。 □

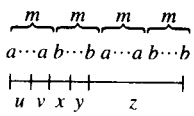


图 8-2

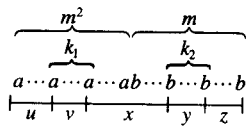


图 8-3

8.1.2 线性语言的泵引理

我们在前面对线性上下文无关文法与非线性上下文无关文法做了区分，这里我们将对相应的语言做类似的区分。

定义8.1 对一个上下文无关语言，如果存在线性上下文无关文法G满足$L=L(G)$，则该语言被称为线性语言。

显然，每一个线性语言都是上下文无关语言，但是我们还没有给出它的逆命题是否成立的结论。

例8.5 语言$L=\{a^nb^n : n\geqslant 0\}$是线性语言，例1.10给出了它的线性文法。而例1.12中给定的语言$L=\{w : n_a(w)=n_b(w)\}$的文法不是线性的，因此语言L不一定是线性的。 □

当然，仅仅根据一个特定的文法是非线性的，不能说明由该文法生成的语言都是非线性的。如果我们需要证明一个语言是非线性的，就必须证明不存在等价的线性文法。为了达到这一目的，我们采用常规方法，通过建立线性文法的结构特性，然后再证明某些上下文无关语言没有这样的特性。

定理8.2 设L是一个无穷线性语言，存在某个正整数m，使得任意$w\in L$（其中$|w|\geqslant m$）都能够被分解为$w=uvxyz$，其中

209 ~ 210

$$|uvyz|\leqslant m \tag{8-5}$$

$$|xy|\geqslant 1 \tag{8-6}$$

则对于所有的$i=0,1,2,\cdots$满足

$$uv^ixy^iz\in L \tag{8-7}$$

注意，上述定理得到的结论与定理8.1是存在区别的，这是由于使用式（8-5）替换了式（8-2）。这表明可以抽取的符号串v与y必须分别位于w中自左端开始的长度为m的符号串中和w中自右端开始的长度为m的符号串中。而中间符号串x可以为任意长度。

证明：我们的推理过程遵循定理8.1的证明。由于语言是线性的，因而对它存在某个线性文法。为了使用定理8.1中的证明，我们还必须保证文法G中没有单位产生式与λ产生式。通过考察定理6.3与定理6.4的证明可以看出，消除单位产生式与λ产生式将不会破坏文法的线性特性，因此我们可以假定文法G满足相应的要求。

现在考虑图8-1中的推导树。由于文法是线性的，所以变量只能出现在从S到第一个A的路径中、从第一个A到第二个A的路径中，以及从第二个A到树的某个叶结点的路径中。因为从S到第一个A的路径中只有有穷个变量，且它们都只能产生有穷数目的终结符，所以u与z的长度一定是有穷的。同理，v与y也是有穷的，可知式（8-5）成立。

剩下的证明过程与定理8.1相同。■

例8.6 语言$L=\{w : n_a(w)=n_b(w)\}$不是线性的。

为了证明上述命题，假定此语言是线性的并且对符号串

$$w=a^mb^{2m}a^m$$

应用定理8.2。根据不等式（8-5）可知，在这种情况下，符号串u, v, y, z都必须只包含a。这样

抽取该符号串将得到$a^{m+k}b^{2m}a^{m+l}$，其中$k \geqslant 1$或$l \geqslant 1$，而它不在L中，这与定理8.2矛盾，也就证明该文法不是线性的。 □ 211

上例回答了上下文无关语言族与线性语言族之间关系的一般性问题。线性语言族是上下文无关语言族的真子集。

习题

1. 采用与例4.11中类似的推理过程给出完整证明：例8.3中的语言不是上下文无关的。
2. 证明语言$L = \{a^n : n$是质数$\}$不是上下文无关语言。
3. 证明语言$L = \{ww^Rw : w \in \{a, b\}^*\}$不是上下文无关语言。●
4. 证明语言$L = \{w \in \{a, b, c\}^* : n_a^2(w) + n_b^2(w) = n_c^2(w)\}$不是上下文无关语言。
5. $L = \{a^nb^m : n = 2^m\}$是否是上下文无关语言？
6. 证明语言$L = \{a^{n^2} : n \geqslant 0\}$不是上下文无关语言。
7. 证明下列定义$\Sigma = \{a, b, c\}$上的语言不是上下文无关的。
   (a) $L = \{a^nb^j : n \leqslant j^2\}$ ●
   (b) $L = \{a^nb^j : n \geqslant (j-1)^3\}$
   (c) $L = \{a^nb^jc^k : k = jn\}$
   (d) $L = \{a^nb^jc^k : k > n, k > j\}$
   (e) $L = \{a^nb^jc^k : n < j, n \leqslant k \leqslant j\}$
   (f) $L = \{w : n_a(w) < n_b(w) < n_c(w)\}$(S)
   (g) $L = \{w : n_a(w)/n_b(w) = n_c(w)\}$
   (h) $L = \{w \in \{a, b, c\}^* : n_a(w) = n_b(w) = 2n_c(w)\}$
8. 判断下述语言是否为上下文无关的。
   (a) $L = \{a^nww^Ra^n : n \geqslant 0, w \in \{a, b\}^*\}$
   (b) $L = \{a^nb^ja^nb^j : n \geqslant 0, j \geqslant 0\}$ ●
   (c) $L = \{a^nb^ja^jb^n : n \geqslant 0, j \geqslant 0\}$
   (d) $L = \{a^nb^ja^kb^l : n + j \leqslant k + l\}$
   (e) $L = \{a^nb^ja^kb^l : n \leqslant k, j \leqslant l\}$
   (f) $L = \{a^nb^nc^j : n \leqslant j\}$
9. 在定理8.1中，根据文法G的性质给出m的范围。
10. 判断语言

$$L = \{w_1cw_2 : w_1, w_2 \in \{a, b\}^*, w_1 \neq w_2\}$$

是否是上下文无关的。● 212

11. 证明语言$L = \{a^nb^na^mb^m : n \geqslant 0, m \geqslant 0\}$是上下文无关的，但不是线性的。
12. 证明语言$L = \{w : n_a(w) \geqslant n_b(w)\}$不是线性的。●
13. 证明语言$L = \{w \in \{a, b, c\}^* : n_a(w) + n_b(w) = n_c(w)\}$是上下文无关的，但不是线性的。
14. 判断语言$L = \{a^nb^j : j \leqslant n \leqslant 2j - 1\}$是否是线性的。
15. 判断例5.12中的语言是否是线性的。●

16. 在定理8.2中，根据文法G的性质给出m的范围。
17. 证明定理8.2中的命题：对于任意的线性语言（不包含λ），都存在一个不包含单位产生式与λ产生式的线性文法。
18. 考虑所有形如a/b的符号串集合，其中a与b都是十进制正整数且$a<b$，该集合表示了所有正的十进制的小数。判断该集合代表的语言是否是上下文无关的。
★19. 证明习题6中语言的补不是上下文无关的。
20. 判断如下语言是否是上下文无关的？

$$L = \{a^{nm} : n\text{与}m\text{是质数}\}$$ ●

8.2 上下文无关语言的封闭性质和判定算法

在第4章中，我们学习了某些运算的封闭性以及用于判定正则语言族性质的算法。总的来说，在那里提出的问题易于回答。但要针对上下文无关语言回答同样问题，相对来说要更加困难。首先，正则语言中成立的封闭性在上下文无关语言中并不一定成立。即使成立，这些命题的证明也常常是非常复杂的。其次，在上下文无关语言中，一些直观上简单而又重要的问题目前无法解决。乍看起来令人难以置信，对此我们将在后续的学习中予以详细阐述。本节只是给出了一些重要结论的样例。

8.2.1 上下文无关语言的封闭性质

定理8.3 *上下文无关语言族对并运算、连接运算和闭包运算是封闭的。*

213 **证明**：设L_1和L_2是两个上下文无关语言，G_1和G_2分别是生成L_1与L_2的上下文无关文法。其中$G_1 = (V_1, T_1, S_1, P_1)$，$G_2 = (V_2, T_2, S_2, P_2)$。不失一般性地，我们可以假定集合$V_1$与$V_2$是不相交的。

考虑语言$L(G_3)$，生成它的文法为

$$G_3 = (V_1 \cup V_2 \cup \{S_3\}, T_1 \cup T_2, S_3, P_3)$$

其中S_3是不属于$V_1 \cup V_2$的变量。G_3中的产生式包括G_1的产生式、G_2的产生式以及一个用于决定使用文法G_1还是G_2的选择性开始产生式。更为准确地描述为

$$P_3 = P_1 \cup P_2 \cup \{S_3 \to S_1 | S_2\}$$

显然，G_3是一个上下文无关文法，因而$L(G_3)$是一个上下文无关语言。同时容易看出

$$L(G_3) = L_1 \cup L_2 \tag{8-8}$$

假定某一$w \in L_1$，则

$$S_3 \Rightarrow S_1 \overset{*}{\Rightarrow} w$$

是文法G_3的一个可能推导。对于$w \in L_2$可以得到同样的结论。而如果$w \in L(G_3)$，则推导的第一步一定是

$$S_3 \Rightarrow S_1 \tag{8-9}$$

或

$$S_3 \Rightarrow S_2 \tag{8-10}$$

如果使用式（8-9），由于自S_1导出的句型只包含V_1中的变量，且V_1与V_2是不相交的，所以推导

$$S_1 \overset{*}{\Rightarrow} w$$

只与P_1中的产生式相关，因此w也就必然属于L_1。另一种情况，如果首先使用式（8-10），则w必然在L_2中。由此可得$L(G_3)$是L_1与L_2的并。

接下来，考虑

$$G_4 = (V_1 \cup V_2 \cup \{S_4\}, T_1 \cup T_2, S_4, P_4)$$

这里S_4是一个新的变量，并且

$$P_4 = P_1 \cup P_2 \cup \{S_4 \rightarrow S_1 S_2\}$$

214

则容易得到

$$L(G_4) = L(G_1)L(G_2)$$

最后，考虑$L(G_5)$，生成它的文法为

$$G_5 = (V_1 \cup \{S_5\}, T_1, S_5, P_5)$$

其中S_5是一个新的变量，且

$$P_5 = P_1 \cup \{S_5 \rightarrow S_1 S_5 | \lambda\}$$

则可得

$$L(G_5) = L(G_1)^*$$

这样，我们就已经证明了上下文无关语言族对并运算、连接运算和闭包运算是封闭的。■

定理8.4 上下文无关语言族对交运算和补运算是不封闭的。

证明：考虑语言

$$L_1 = \{a^n b^n c^m : n \geqslant 0, m \geqslant 0\}$$

与

$$L_2 = \{a^n b^m c^m : n \geqslant 0, m \geqslant 0\}$$

存在几种方式可以证明L_1与L_2都是上下文无关的。例如，生成L_1的文法为

$$S \rightarrow S_1 S_2$$
$$S_1 \rightarrow a S_1 b | \lambda$$
$$S_2 \rightarrow c S_2 | \lambda$$

另外，注意到L_1是两个上下文无关语言的连接，根据定理8.3，可知它是上下文无关的。但我们已经证明

$$L_1 \cap L_2 = \{a^n b^n c^n : n \geqslant 0\}$$

215 不是上下文无关的。因此上下文无关语言族对交运算是不封闭的。

定理的第二部分可以根据定理8.3以及集合恒等式

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

进行证明。如果上下文无关语言类对补运算是封闭的，则对于任意的上下文无关语言L_1与L_2，上述恒等式右边的语言是上下文无关语言。但是这与我们刚刚证明的结论相矛盾，也就是两个上下文无关语言的交运算不一定是上下文无关的。因而可以得到，上下文无关语言对补运算是不封闭的。■

虽然两个上下文无关语言的交运算生成的语言可能不是上下文无关的，但如果其中一个语言是正则的，交运算的封闭性就成立。

定理8.5 设L_1是一个上下文无关语言，L_2是一个正则语言，则$L_1 \cap L_2$是一个上下文无关语言。

证明：设$M_1 = (Q, \Sigma, \Gamma, \delta_1, q_0, z, F_1)$是接受语言$L_1$的一个npda，$M_2 = (P, \Sigma, \delta_2, p_0, F_2)$是接受语言$L_2$的dfa。我们构造一个下推自动机$\hat{M} = (\hat{Q}, \Sigma, \Gamma, \hat{\delta}, \hat{q}_0, z, \hat{F})$，它能够模拟$M_1$与$M_2$的并行动作：当读入输入符号串的一个符号时，$\hat{M}$同步地执行$M_1$与$M_2$的迁移。最后设

$$\hat{Q} = Q \times P$$
$$\hat{q}_0 = (q_0, p_0)$$
$$\hat{F} = F_1 \times F_2$$

并定义$\hat{\delta}$使得

$$((q_k, p_l), x) \in \hat{\delta}((q_i, p_j), a, b)$$

216 当且仅当$(q_k, x) \in \delta_1(q_i, a, b)$与$\delta_2(p_j, a) = p_l$。

在这里我们同样要求如果$a = \lambda$，则$p_j = p_l$。换句话说，$\hat{M}$的状态由状态对(q_i, p_j)标记，它们分别表示M_1与M_2在读入某一个输入符号串后能够到达的状态。通过一个简单的归纳证明可以得到：

$$((q_0, p_0), w, z) \vdash^*_{\hat{M}} ((q_r, p_s), x)$$

其中$q_r \in F_1$且$p_s \in F_2$，当且仅当$(q_0, w, z) \vdash^*_{M_1} (q_r, x)$和$\delta^*(p_0, w) = p_s$成立。

因此，如果一个符号串可以被$\hat{M}$接受，当且仅当它能够被M_1与M_2接受，也就是说，它属于$L(M_1) \cap L(M_2) = L_1 \cap L_2$。■

本定理给出的性质被称为正则交运算（regular intersection）的封闭性。根据该定理的结论，我们可以得到，上下文无关语言族对正则交运算是封闭的。在简化针对特定语言的相关证明中，这一封闭性有时是有用的。

例8.7 证明语言$L = \{a^n b^n : n \geqslant 0, n \neq 100\}$是上下文无关的。

针对这一命题，可以采用构造该语言的pda或者上下文无关文法的方式证明，但证明过程是繁琐的。根据定理8.5，我们可以得到一个更加简洁的证明。

设

$$L_1 = \{a^{100} b^{100}\}$$

由于L_1是有穷的，因而它是正则的。同样，容易看出

$$L = \{a^n b^n : n \geqslant 0\} \cap \overline{L}_1$$

因此，根据正则语言补运算的封闭性以及上下文无关语言的正则交运算的封闭性，可以得到需要的结论。 □ 217

例8.8 证明语言

$$L = \{w \in \{a, b, c\}^* : n_a(w) = n_b(w) = n_c(w)\}$$

不是上下文无关的。

可以采用泵引理来证明这一命题。但是根据正则交运算的封闭性，我们可以得到一个更加简洁的证明。假设L是上下文无关的，则

$$L \cap L(a^* b^* c^*) = \{a^n b^n c^n : n \geqslant 0\}$$

也将是上下文无关的，但我们已知它不是上下文无关的。因此可得L不是上下文无关的。 □

语言的封闭性质在形式语言理论中发挥着重要的作用，针对上下文无关语言，可以建立更多的封闭性质。本节习题中将介绍另外一些相关的结论。

8.2.2 上下文无关语言的可判定性质

结合定理5.2与定理6.6，我们已经确立了上下文无关语言的成员资格判定算法。这对任何语言族都是一个在实际中很有用的基本特性。上下文无关语言的其他一些简单性质也是可以确定的。根据讨论的需要，我们假定语言都由相应的文法描述。

定理8.6 *给定一个上下文无关文法$G = (V, T, S, P)$，存在判定$L(G)$是否为空的算法。*

证明：为了简化，假设$\lambda \notin L(G)$。如果不是这样的，证明过程只需要做一些小的改动。我们将对其应用消除无用符号与无用产生式的算法。如果S是无用的，则$L(G)$为空；否则$L(G)$至少包含一个元素。■

定理8.7 *给定一个上下文无关文法$G = (V, T, S, P)$，存在判定$L(G)$是否有穷的算法。*

证明：假设G中没有λ产生式、单位产生式以及无用符号。如果该文法中有一个重复的变量，也就是说存在某一个$A \in V$，它有推导

$$A \overset{*}{\Rightarrow} xAy$$ 218

由于G中没有单位产生式与λ产生式，因此x与y不能同时为空。又因为A既不是可空变量也不是无用符号，所以可得

$$S \overset{*}{\Rightarrow} uAv \overset{*}{\Rightarrow} w$$

和

$$A \overset{*}{\Rightarrow} z$$

其中u，v与z都属于T^*。则对所有的n

$$S \overset{*}{\Rightarrow} uAv \overset{*}{\Rightarrow} ux^n Ay^n v \overset{*}{\Rightarrow} ux^n zy^n v$$

都是可能的，所以$L(G)$是无穷的。

如果文法中没有重复的变量，则任意推导的长度都不会超过$|V|$，在这种情况下，$L(G)$显然是有穷的。

因此，为了得到判定$L(G)$是否有穷的算法，我们仅需要判定文法中是否存在重复的变量。我们可以通过变量依赖图来完成，在该方法中，如果存在产生式

$$A \to xBy$$

则存在相应的边（A, B）。在得到的依赖图中，回路中的任意一个开始符号都是可重复的。因而，一个文法中存在重复变量当且仅当依赖图中存在回路。

我们有了判定文法中是否存在重复变量的算法，就有判定一个语言$L(G)$是否有穷的算法。■

令人奇怪的是，上下文无关语言中的其他简单性质却不这么容易处理。如定理4.7，我们希望能够找到判定两个上下文无关文法生成的语言是否相同的算法，但事实证明不存在这样的算法。就目前而言，虽然这在直观上很明确，但我们还没有相应的技术机制能够准确地给出“没有算法”的准确含义。这是一个重点，后面我们还会回来讨论它。

习题

1. 习题8.8中语言的补是否是上下文无关的？●
219 2. 考虑定理8.4中的语言L_1，证明该语言是线性的。
3. 证明上下文无关语言族在同态下具有封闭性。
4. 证明线性语言族在同态下是封闭的。
5. 证明上下文无关语言族在逆下是封闭的。●
6. 指出目前讨论的语言族中，哪些在逆下是封闭的。
7. 证明上下文无关语言族对一般性的差运算不封闭，但对正则差运算封闭。也就是说，如果L_1是上下文无关语言，L_2是正则语言，则$L_1 - L_2$是上下文无关语言。
8. 证明确定型上下文无关语言族在正则差运算下是封闭的。
9. 证明线性语言族对并运算是封闭的，而对连接运算不封闭。●
10. 证明线性语言对交运算是不封闭的。
11. 证明确定型上下文无关语言族对并运算和交运算是不封闭的。
12. 给出一个上下文无关文法的实例，使得它的补不是上下文无关语言。
★13. 证明如果L_1是线性语言，L_2是正则语言，则L_1L_2是线性语言。●
14. 证明无二义上下文无关语言族对并运算不封闭。
15. 证明无二义上下文无关语言族对交运算不封闭。●
16. 设L是一个确定型上下文无关语言，根据它定义一个新的语言$L_1 = \{w : aw \in L, a \in \Sigma\}$，判定$L_1$是否一定是确定型上下文无关语言。
17. 证明语言$L = \{a^n b^n : n \geqslant 0$且$n$不是5的倍数$\}$是上下文无关的。
18. 证明下述语言是上下文无关的：

$$L = \{w \in \{a, b\}^* : n_a(w) = n_b(w), w\text{不包含子串}aab\}$$

19. 确定型上下文无关语言族在同态下是否是封闭的。

20. 给出定理8.5的详细推理论证。
21. 给出一个算法，对于任意给定的上下文无关文法G，它都能够判定是否有$\lambda \in L(G)$。
22. 证明存在一个算法，它能够判定某一上下文无关文法生成的语言是否包含长度小于某一给定数n的字。
23. 设L_1是一个上下文无关语言，L_2是一个正则语言，证明存在一个算法，它能够判定L_1与L_2是否包含某一相同元素。

220

第9章 图 灵 机

在前述的讨论中，我们碰到了一些基本概念，特别是正则语言与上下文无关语言及其与有穷自动机与下推接受器的关联。我们的研究发现，正则语言是上下文无关语言的一个真子集，因而下推自动机也就比有穷自动机能力更强大。同时也已经看到，虽然上下文无关语言对于程序设计语言的研究是重要的，但它在应用范围上受到限制。这一点在上一章中已经说得很清楚，比如一些简单的语言$\{a^nb^nc^n\}$与$\{ww\}$就不是上下文无关的。这促使我们考察、研究在上下文无关语言之外，如何定义一个新的语言族使之能包含这些例子。因此，我们需要回到自动机的一般性描述上来。如果我们比较有穷自动机与下推自动机，就可以发现它们之间的区别在于临时存储空间。如果没有存储，得到的就是有穷自动机；而如果存储机制为栈，将得到更强大的下推自动机。根据这一观察推测，如果我们给自动机更加灵活的存储机制，将可能发现更加强大的语言族。例如，在图1-3所示的一般性方案中，如果采用两个栈、三个栈、队列，或者其他的一些存储机制，将会发生什么情况？是否每一种存储机制都定义了一种新的自动机，并且能够通过它得到一个新的语言族？这种方式引发了大量的问题，而它们中的大多数是没有意义的。如果能提出一个更具挑战性的问题以及考虑能将自动机的概念推向多远，将更具指导意义。对于自动机的最强能力以及计算的极限，我们可以得到一些什么样的结论？这就奠定了图灵机（Turing machine）概念的基础，由此也可以得出机械计算或算法的准确定义。

221

首先我们将研究图灵机的形式化定义，然后通过一些简单的程序，对图灵机的作用提供一个感性的认识，接下来，将说明图灵机机制是非常基本的，它的概念宽泛足以解决非常复杂的过程。图灵论题的讨论结果表明，图灵机可以实现任何计算过程，比如目前计算机中执行的程序，都能在图灵机中完成。

9.1 标准图灵机

虽然我们可以想像出各种具有复杂存储机制的自动机，但图灵机的存储是非常简单的。可以将其看成一个一维单元组，其中每一个单元可以包含一个符号。该单元组可以在两端无限扩展，因此它能够存储无穷的信息。这些信息可以以任意顺序读取和修改，我们称这样的存储机制为带（tape），因为它与实际计算机中的磁带十分类似。

9.1.1 图灵机的定义

图灵机是一种自动机，它采用带作为临时存储。带可以被划分为若干单元，其中每一个单元包含一个符号。与带关联的是读写头（read-write head），它能够在带上左右移动，并且每次移动能够读写一个符号。与第1章中描述的一般方案存在细微的区别，在这里作为图灵机的自动机既没有输入文件也没有特别的输出机制。无论输入还是输出都将在自动机的带上进行。后面我们将看到对1.2节中的一般模型做这种修改不会有什么影响。我们也可以保留其输

入文件与输出机制而不会影响后续的结论，但我们没有这么做，因为以这种方式得到的自动机更易于描述。

222

图9-1给出了图灵机的一个直观描述，在定义9.1中，我们给出它的准确定义。

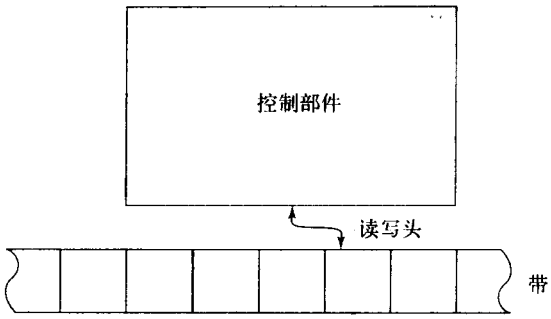


图 9-1

定义9.1 图灵机M由

$$M=(Q,\Sigma,\Gamma,\delta,q_0,\square,F)$$

定义，其中

Q是内部状态的集合，

Σ是输入字母表，

Γ是符号的有穷集合，我们称之为带字母表（tape alphabet），

δ是转移函数，

$\square\in\Gamma$是一个特殊的符号，我们称之为空白符（blank），

$q_0\in Q$是初态，

$F\subseteq Q$是终态集合。

在图灵机的定义中，我们假定$\Sigma\subseteq\Gamma-\{\square\}$，也就是说输入字母表是带字母表的子集，但它不包括空白符。空白符不作为输入的原因将在下面的讨论中揭示。转移函数δ定义如下

$$\delta: Q\times\Gamma\rightarrow Q\times\Gamma\times\{L,R\}$$

一般而言，δ是$Q\times\Gamma$上的部分函数；它们给出了图灵机操作的规则。δ的参数是控制部件的当前状态以及即将读入的带符号。它返回的结果是一个新的控制部件状态、一个替换旧带符号的新符号以及一个迁移符号L或者R。迁移符号用于表明在新的带符号被记录在带上后，读写头的移动方向：向左或者向右。

223

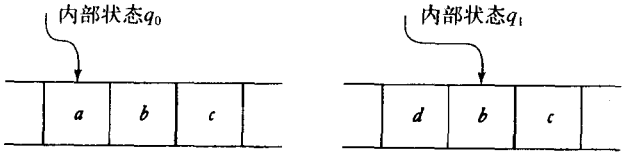
例9.1 图9-2显示了转移函数

$$\delta(q_0,a)=(q_1,d,R)$$

执行之前与之后的情况。

我们可以将图灵机看成一个简单的计算机。它有一个存储有限的处理部件，而带上的二级存储容量是无限的。这种计算机所能处理的指令是有限的：识别带上的符号并根据它确定下一步的动作。图灵机能够完成的动作就是覆盖当前符号，改变控制部件的状态并移动读写头。这么小的指令集看起来不足以处理复杂的事情，但实际情况并不是这样的。通

常，图灵机是非常强大的。而转移函数δ定义了计算机是怎样工作的，也就是我们通常所说的“程序”。



a) 迁移前的情形　　b) 迁移后的情形

图 9-2

通常，自动机以一个给定的初态和带上的一些信息开始。然后在转移函数δ的控制下完成一系列的动作。在这一过程中，带上任何一个单元的内容可能会被多次检查与修改。最后通过将图灵机置于停机状态（halt state）使整个过程结束。当图灵机到达一个δ中没有定义的格局时，将会停机；这种情况是可能的，因为δ只是部分函数。实际上，我们假定对于任意的终态都没有定义其转移函数，那么当图灵机到达终态时，也将停机。 □ 224

例9.2 考虑由

$$Q = \{q_0, q_1\}$$
$$\Sigma = \{a, b\}$$
$$\Gamma = \{a, b, \square\}$$
$$F = \{q_1\}$$

和

$$\delta(q_0, a) = \{q_0, b, R\}$$
$$\delta(q_0, b) = \{q_0, b, R\}$$
$$\delta(q_0, \square) = \{q_1, \square, L\}$$

定义的图灵机。如果该图灵机的初态为q_0，并且读写头位于符号a上，则可以应用转移函数$\delta(q_0, a) = (q_0, b, R)$。因此读写头将$a$替换成$b$，然后向右移动，图灵机状态仍为$q_0$。接下来的任意$a$也将被替换成$b$，而对$b$将不作任何改动。当该图灵机读到一个空白符，它将向左移动一个单元，然后停机在终态q_1上。

图9-3描述了一个简单初始格局迁移过程的几个阶段。 □

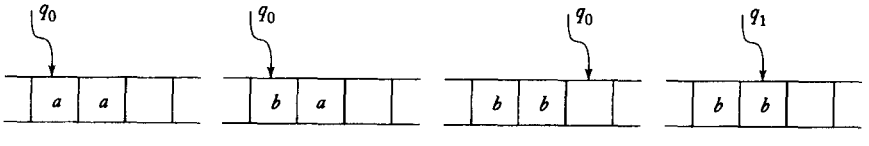


图9-3 迁移序列

例9.3 Q, Σ, Γ采用例9.2中的定义，并设F为空，δ定义如下

$$\delta(q_0, a) = \{q_1, a, R\}$$
$$\delta(q_0, b) = \{q_1, b, R\}$$
$$\delta(q_0, \square) = \{q_1, \square, R\}$$

$$\delta(q_1, a) = \{q_0, a, L\}$$
$$\delta(q_1, b) = \{q_0, b, L\}$$
225 $$\delta(q_1, \square) = \{q_0, \square, L\}$$

为了解这里发生的情况，我们可以跟踪一种典型的情形。假定初始时带上包含$ab\cdots$，并且读写头位于a上。然后图灵机读入a，但并不改变它。图灵机的下一个状态将转换为q_1且读写头向右移动使之位于b上。接下来这一符号同样被读入也不作改变。而图灵机将进入状态q_0并且读写头向左移动，现在它又回到了初态，迁移序列也将重新开始。显见，无论带上初始信息如何，该图灵机都将不停地工作，使得读写头不断地左右移动，而对带信息不作任何改变。这是一个不会停机的图灵机，和程序设计术语中的情况类似，我们说图灵机进入了*无穷循环*(infinite loop)。 □

由于图灵机存在多种不同形式的定义，因而有必要总结一下我们关于图灵机模型的主要特性，对于具备这些特性的图灵机，我们称之为*标准图灵机*（standard Turing machine）。图灵机的主要特性有:

1. 图灵机的带在左右方向上都没有限制，允许任意数目的左迁移和右迁移。
2. 由于对每一个格局，δ最多只定义了一种迁移，因而它是确定型的。
3. 没有特定的输入文件。我们假定在初始时刻，带上存在特定的内容，其中部分内容可以作为图灵机的输入。同样，没有特定的输出设备，当图灵机停机时，带的部分或者全部内容可以作为输出。

上述约定主要是为了后续讨论的方便。在第10章中我们将看到图灵机的其他形式，并且要讨论它们与标准模型之间的关系。

为了展示图灵机的格局情况，我们将使用瞬时描述的概念。每一个格局都完全地由当前控制部件的状态、带的内容以及读写头位置决定。我们采用形如

$$x_1 q x_2$$

或

$$a_1 a_2 \cdots a_{k-1} q a_k a_{k+1} \cdots a_n$$

的表示形式，它是对处于状态q且带的内容如图9-4所示的图灵机的瞬时描述。其中符号$a_1, \cdots,$
226 a_n表示带的内容，而q定义了控制部件的状态。这种约定形式使得读写头位于紧跟在q后的符号所在的单元上。

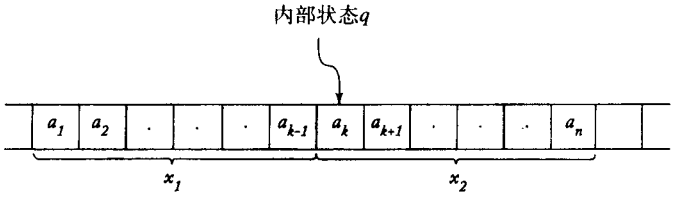


图 9-4

瞬时描述只给出了读写头左右有限的信息。而假定带上没有说明的部分为空白符；通常这种空白是不相关的，并且不在瞬时描述中予以显式地说明。但如果空白符的位置与讨论相

关，则其可以出现在瞬时描述中。例如瞬时描述$q\square w$表示读写头恰在w左边符号所在的单元上且其内容为空白符。

例9.4 图9-3中的几个图片依次与瞬时描述序列q_0aa, bq_0a, $bbq_0\square$, bq_1b对应。

从一种格局到另一种格局的迁移可采用$\vdash$进行表示。因此，如果

$$\delta(q_1, c) = (q_2, e, R)$$

则当图灵机内部状态为q_1，带内容为$abcd$，读写头位于c上时，将执行迁移

$$abq_1cd \vdash abeq_2d$$

符号$\vdash$通常表示任意数目的迁移，而$\vdash_M$用于区分不同机器上的格局的迁移。 □

例9.5 图9-3中图灵机的动作可以采用

$$q_0aa \vdash bq_0a \vdash bbq_0\square \vdash bq_1b$$

或者$q_0aa \vdash^* bq_1b$表示。 □ 227

为了进一步讨论，采用形式化的方法概括观察到的各种情况是很方便的。

定义9.2 设图灵机$M = \{Q, \Sigma, \Gamma, \delta, q_0, \square, F\}$，则任意串$a_1\cdots a_{k-1}q_1a_ka_{k+1}\cdots a_n$其中$a_i \in \Gamma$且$q_1 \in Q$，是$M$的一个瞬时描述。迁移

$$a_1\cdots a_{k-1}q_1a_ka_{k+1}\cdots a_n \vdash a_1\cdots a_{k-1}bq_2a_{k+1}\cdots a_n$$

是可能的，当且仅当

$$\delta(q_1, a_k) = (q_2, b, R)$$

迁移

$$a_1\cdots a_{k-1}q_1a_ka_{k+1}\cdots a_n \vdash a_1\cdots q_2a_{k-1}ba_{k+1}\cdots a_n$$

是可能的，当且仅当

$$\delta(q_1, a_k) = (q_2, b, L)$$

针对某一初始格局$x_1q_ix_2$，如果

$$x_1q_ix_2 \vdash^* y_1q_jay_2$$

由于对任意q_j与a，$\delta(q_j, a)$没有定义，则M将停机。我们把导致停机状态的格局序列称为计算(computation)。

例9.3说明了图灵机永不停机是可能的，可能处于一个无穷循环处理中。这种情况在图灵机的讨论中起着十分重要的作用，因此我们对其使用一个特殊的表示法。将其表示成

$$x_1qx_2 \vdash^* \infty$$

它表示着从初始格局x_1qx_2开始，将不会停机。 228

9.1.2 作为语言接受器的图灵机

在下面的讨论中，图灵机可以被看成识别器。符号串w被记录在带上，并且其他没有使用的部分为空白符。图灵机从初态q_0开始并使读写头位于w的最左符号上。如果通过一系列的迁

移，图灵机最终进入终态并且停机，则认为w能够被接受。

定义9.3 设$M=(Q,\Sigma,\Gamma,\delta,q_0,\square,F)$是一个图灵机，则被$M$接受的语言为

$$L(M)=\{w\in\Sigma^+ : q_0w\vdash^* x_1q_fx_2\text{对于某个}q_f\in F, x_1, x_2\in\Gamma^*\}$$

这一定义表明输入串w将被记录在带上且其两边均为空白符。将空白符排除在输入之外的原因现在清楚了：它保证了所有输入都被限制在一个带上明确定义的区域中，且其左右均为空白符。如果没有这一约定，图灵机将不能限制输入串的放置区域；因而无论读入了多少空白符，都不能确定是否在带上还存在非空白的输入。

定义9.3给出了当$w\in L(M)$时将会出现的情况。而对于其他输入，它没有对结果给出说明。当w不在$L(M)$中时，可能会发生两种情况：图灵机将在一个非终态下停机，或者将进入一个无穷循环而永不停机。根据定义，任何使M不会停机的符号串都不在$L(M)$中。

例9.6 基于$\Sigma=\{0,1\}$，设计一个图灵机，它能够接受正则表达式00*表示的语言。

这是图灵机程序设计的一个简单习题。从输入符号串的左端开始，我们读入每一个符号并检查它是否为0。如果是，我们将继续向右迁移。如果在到达一个空白符之前碰到的都是0，我们将终止并接受该符号串。而如果在输入中包含1，则它不在语言$L(00^*)$中，图灵机将在一个非终态下停机。为了跟踪这一计算，我们只需设置两个内部状态$Q=\{q_0,q_1\}$和一个终态$F=\{q_1\}$就可以了。我们定义转移函数为

$$\delta(q_0,0)=(q_0,0,R)$$

229

$$\delta(q_0,\square)=(q_1,\square,R)$$

只要读写头下的符号为0，读写头就将向右迁移。而一旦读到1，由于$\delta(q_0,1)$没有定义，则图灵机将在非终态q_0下停机。注意到，如果图灵机在空白符上以状态q_0开始，它也将进入到终态下并停机。我们可以认为图灵机接受了λ，但是由于技术上的原因，定义9.3中并没有包含空串。 □

语言越复杂，对它的识别也就越难。由于图灵机采用的是基本指令集，因此通过高级语言可以简单编程的计算在图灵机上通常就是相当复杂的。尽管如此，在下一个例子中，我们将看到，这仍然是可能的且其概念也易于明白。

例9.7 对$\Sigma=\{a,b\}$，设计一个图灵机，它能够接受

$$L=\{a^nb^n : n\geqslant 1\}$$

直观上，我们可以用下面的方式来解决这一问题。从最左边的a开始，我们对其进行核对并以某一符号比如x替换它。然后我们让读写头继续右移直到碰到最左边的b，对它同样进行核对并以某一符号比如y替换它。在这之后，我们又左移到最左边的a上，用x替换它，然后移至最左边的b，用y替换它，依此继续。通过这种来回移动，对每个a，我们用b去和它匹配。如果最后没有a和b了，则认为该符号串一定在语言L中。

通过对上述内容的详细描述，我们得到一个完整的解决方案，其中

$$Q=\{q_0,q_1,q_2,q_3,q_4\}$$
$$F=\{q_4\}$$
$$\Sigma=\{a,b\}$$
$$\Gamma=\{a,b,x,y,\square\}$$

它的转移函数可以分成几个部分，集合

$$\delta(q_0, a) = (q_1, x, R)$$
$$\delta(q_1, a) = (q_1, a, R)$$
$$\delta(q_1, y) = (q_1, y, R)$$
$$\delta(q_1, b) = (q_2, y, L)$$

用x替换最左边的a，然后，使读写头右移至第一个b，并用y替换它。当y被写入带上时，图灵机将进入状态q_2，表明一个a已经成功地与一个b相匹配了。 230

转移函数的下一个集合使读写头向相反的方向移动直到碰到一个x，重新将读写头定位在最左边的a上，并使控制返回到它的初态。

$$\delta(q_2, y) = (q_2, y, L)$$
$$\delta(q_2, a) = (q_2, a, L)$$
$$\delta(q_2, x) = (q_0, x, R)$$

现在我们回到了初态q_0，为处理下一个a与b做好了准备。

在计算的这一部分执行完一遍后，图灵机将完成如下的局部计算

$$q_0aa\cdots abb\cdots b \vdash^* xq_0a\cdots ayb\cdots b$$

它使得一个a与一个b进行匹配。在第二遍后，我们将完成局部计算

$$q_0aa\cdots abb\cdots b \vdash^* xxq_0\cdots ayy\cdots b$$

依此类推，可以看出a与b的匹配过程能够正确执行。

当输入为符号串a^nb^n时，重写过程将以上述方式继续执行直到再没有a可以擦除为止。这时为了找到最左边的a，读写头将在状态q_2下左移。当碰到一个x时，将逆转方向来读取a。但是现在，不是为了找到一个a，而是为了找到y。为了终止，将进行一个终结检查，以确认是否所有的a与b都已经被替换了（检测在输入中是否一个a后有一个b）。这可以通过下式完成。

$$\delta(q_0, y) = (q_3, y, R)$$
$$\delta(q_3, y) = (q_3, y, R)$$
$$\delta(q_3, \square) = (q_4, \square, R)$$

如果我们输入一个不在语言中的符号串，计算将在一个非终态下停止。例如，如果我们输入a^nb^m，其中$n > m$，则图灵机将在状态q_1下碰到一个空白符。由于没有针对这种情况定义转移函数，将导致图灵机停机。其他不在语言中的符号串同样将导致非终结的停机状态（本节习题3）。

对于特定输入$aabb$的连续瞬时描述序列如下:

$$\begin{aligned}
q_0aabb &\vdash xq_1abb \vdash xaq_1bb \vdash xq_2ayb \\
&\vdash q_2xayb \vdash xq_0ayb \vdash xxq_1yb \\
&\vdash xxyq_1b \vdash xxq_2yy \vdash xq_2xyy \\
&\vdash xxq_0yy \vdash xxyq_3y \vdash xxyyq_3\square \\
&\vdash xxyy\square q_4\square
\end{aligned}$$

231

针对上述情况，图灵机将在终态下停机，因此符号串$aabb$能够被接受。

为了更好地了解图灵机的工作情况，我们建议读者对几个在L中与不在L中的符号串进行跟踪。 □

例9.8 设计一个图灵机能够接受语言

$$L = \{a^n b^n c^n : n \geqslant 1\}$$

例9.7中的思想很容易用在这里。我们通过将a，b，c分别替换成x, y, z来对其进行匹配。最后，我们检查是否所有的原始符号都已经被重写了。虽然这仅仅是对前一例子的简单扩展，但是编写实际程序却非常繁琐。我们将它留做一个简单但是有点冗长的习题。注意，虽然$\{a^n b^n\}$是上下文无关语言而$\{a^n b^n c^n\}$并不是，但是可以采用结构非常类似的图灵机接受它们。 □

从上例可以得到一个结论，图灵机能够识别不是上下文无关的语言，它的一个最直接含义就是，图灵机比下推自动机更强大。

9.1.3 作为转换器的图灵机

到目前为止，我们似乎没有研究转换器的必要，因为在语言理论中，接受器就已经足够了。但是我们很快就会看到，图灵机并不仅仅在语言接受上有用，它同样为我们提供了通用电子计算机的一个简单的抽象模型。由于计算机的基本目标在于将输入转换为输出，它就像一个转换器的工作一样。如果我们想要使用图灵机为计算机建模，就有必要深入了解这一方面的情况。

一个计算的输入就是初始时刻带上的非空白符号。而计算完成时，带上所有的符号就是输出。这样，我们可以将一个图灵机转换器看成函数f的实现。f定义如下：如果对某一终态q_f，有

$$\hat{w} = f(w)$$

则

232

$$q_0 w \vdash_M^* q_f \hat{w}$$

定义9.4 *对定义域为D的函数f，如果存在一个图灵机$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$使得对所有的$w \in D$，都有*

$$q_0 w \vdash_M^* q_f f(w), \quad q_f \in F$$

成立，则称该函数是图灵可计算的（Turing-computable）或可计算的（computable）。

很快我们将得到，所有一般性的数学函数，无论它多么复杂，都是图灵可计算的。首先，我们将看几个诸如加法和算术比较的简单运算。

例9.9 给定两个正整数x和y，设计一个能够计算$x + y$的图灵机。

首先，我们需要选取某一约定来表示正整数。为了简化，我们采用一元表示法，它将任意的正整数x表示成$w(x) \in \{1\}^+$，满足

$$|w(x)| = x$$

同样，我们也将确定x与y是怎样被初始地放置在带上的，在计算结束时，它们的和是怎

样表示的。这里，我们假定$w(x)$与$w(y)$以一元表示法放置在带上并由一个0隔开，同时读写头位于$w(x)$的最左符号上。计算结束后，$w(x+y)$将以0结尾出现在带上，读写头位于结果的左端。因此，我们需要设计一个图灵机来执行如下计算

$$q_0w(x)0w(y)\vdash^* q_fw(x+y)0$$

其中q_f是终态。针对这些构造一个程序是相对简单的。我们需要做的是首先将分隔0移至$w(y)$的右端，使得加法就只将两个符号串进行连接，此外什么也不做。为了实现该程序，我们构造$M=(Q,\Sigma,\Gamma,\delta,q_0,\square,F)$，其中 233

$$\begin{aligned}
Q &= \{q_0, q_1, q_2, q_3, q_4\}\\
F &= \{q_4\}\\
\delta(q_0, 1) &= (q_0, 1, R)\\
\delta(q_0, 0) &= (q_1, 1, R)\\
\delta(q_1, 1) &= (q_1, 1, R)\\
\delta(q_1, \square) &= (q_2, \square, L)\\
\delta(q_2, 1) &= (q_3, 0, L)\\
\delta(q_3, 1) &= (q_3, 1, L)\\
\delta(q_3, \square) &= (q_4, \square, R)
\end{aligned}$$

注意，在我们将0向右移时，临时创建了一个1，这一情况通过将图灵机置于状态q_1来进行记录。计算结束时，需要使用转移函数$\delta(q_2, 1)=(q_3, 0, R)$消除它。这可以从如下111加11的瞬时描述序列中看出:

$$\begin{aligned}
q_0111011 &\vdash 1q_011011 \vdash 11q_01011 \vdash 111q_0011\\
&\vdash 1111q_111 \vdash 11111q_11 \vdash 111111q_1\square\\
&\vdash 11111q_21 \vdash 1111q_310\\
&\vdash^* q_3\square111110 \vdash q_4111110
\end{aligned}$$

虽然一元表示法对于实际的计算比较繁琐，但对于图灵机编程它是方便的。通过它得到的结果程序比采用其他表示法，如二进制或者十进制要简短得多。 □

加法是任何计算机的基本操作之一，它可以作为更加复杂指令合成的组成部分。其他的基本操作还有拷贝符号串及简单比较，它们也很容易通过图灵机实现。

例9.10 设计一个图灵机用于拷贝由1构成的符号串。更准确地说，也就是构造一个机器能够执行计算

$$q_0w \vdash^* q_fww$$

其中$w\in\{1\}^+$。 234

为了解决这一问题，我们实现如下的直观过程:

1. 用x替换每一个1。
2. 找到最右边的x，用1替换它。
3. 移至当前带上非空区域的最右端，并创建一个1。
4. 重复步骤2与3直到再也没有x为止。

实现它的一个图灵机为

$$\delta(q_0, 1) = (q_0, x, R)$$
$$\delta(q_0, \square) = (q_1, \square, L)$$
$$\delta(q_1, x) = (q_2, 1, R)$$
$$\delta(q_2, 1) = (q_2, 1, R)$$
$$\delta(q_2, \square) = (q_1, 1, L)$$
$$\delta(q_1, 1) = (q_1, 1, L)$$
$$\delta(q_1, \square) = (q_3, \square, R)$$

其中q_3是唯一的终态。首先，要理解它是有点难度的，我们可以用简单的符号串11来跟踪该程序，这种情况下，计算执行如下:

$$\begin{aligned} q_0 11 &\vdash x q_0 1 \vdash x x q_0 \square \vdash x q_1 x \\ &\vdash x 1 q_2 \square \vdash x q_1 11 \vdash q_1 x 11 \\ &\vdash 1 q_2 11 \vdash 11 q_2 1 \vdash 111 q_2 \square \\ &\vdash 11 q_1 1^1 \vdash 1 q_1 111 \\ &\vdash q_1 1111 \vdash q_1 \square 1111 \vdash q_3 1111 \end{aligned}$$

□

例9.11 设x与y是采用一元表示法表示的两个正整数。构造一个图灵机使得如果$x \geqslant y$，则在终态q_y下停机，而如果$x < y$，则在非终态q_n下停机。更为形式化地，也就是使得该机器能够执行计算

$$q_0 w(x) 0 w(y) \vdash^* q_y w(x) 0 w(y), \quad 如果 x \geqslant y$$
$$q_0 w(x) 0 w(y) \vdash^* q_n w(x) 0 w(y), \quad 如果 x < y$$

235

为了解决这一问题，我们可以对例9.7中的思想做一些小的修改。不匹配a与b，而匹配分隔符0左边的1与右边的1。通过匹配，在带上我们可以根据$x > y$还是$y > x$得到结果

$$xx \cdots 110 xx \cdots x \square$$

或者

$$xx \cdots xx0xx \cdots x11 \square$$

在第一种情况下，当我们试图匹配下一个1时，在工作空间的右端碰到了空白符，它可以作为图灵机进入状态q_y的信号。在第二种情况下，当左边的1都被替换完后，右边仍然有1，这种情况下图灵机将进入状态q_n。针对这一计算的完整程序很简单，我们将其留为习题。

通过这个例子可以得到一个重要的结论，可以给图灵机编程，使它能根据算术比较进行决策。这种简单的决策在计算机的机器语言中是常见的，它根据算术运算的结果选择进入分支指令流。 □

习题

★★1. 使用高级程序设计语言设计一个图灵机模拟器。该模拟器能够以任何图灵机的描述和一个初始格局作为输入，并输出相应的计算结果。

2. 设计一个状态个数不超过3的图灵机，它能够接受语言$L(a(a+b)^*)$。如果假定$\Sigma=\{a,b\}$，两状态的图灵机是否能够接受该语言？

3. 在例9.7中，如果输入的是aba与$aaabbbb$，说明该图灵机是如何工作的。

4. 在例9.7中，是否存在输入使该图灵机进入无穷循环？ 236

5. 确定由图灵机$M=(\{q_0,q_1,q_2,q_3\},\{a,b\},\{a,b,\square\},\delta,q_0,\square,\{q_3\})$接受的语言，其中

$$\delta(q_0,a)=(q_1,a,R)$$
$$\delta(q_0,b)=(q_2,b,R)$$
$$\delta(q_1,b)=(q_1,b,R)$$
$$\delta(q_1,\square)=(q_3,\square,R)$$
$$\delta(q_2,b)=(q_2,b,R)$$
$$\delta(q_2,a)=(q_3,a,R)$$

6. 在例9.10中，如果符号串w中包含1以外的符号，将会发生什么情况？

7. 为如下基于$\{a,b\}$的语言构造图灵机：

(a) $L=L(aba^*b)$

(b) $L=\{w:|w|$是偶数$\}$

(c) $L=\{w:|w|$是3的倍数$\}$

(d) $L=\{a^nb^m:n\geqslant 1,n\neq m\}$

(e) $L=\{w:n_a(w)=n_b(w)\}$

(f) $L=\{a^nb^ma^{n+m}:n\geqslant 0,m\geqslant 1\}$

(g) $L=\{a^nb^na^nb^n:n\geqslant 0\}$

(h) $L=\{a^nb^{2n}:n\geqslant 1\}$

对于上述每一个问题，给出δ的详细内容，然后通过跟踪几个测试实例来检验答案。

8. 设计图灵机使之能够接受语言

$$L=\{ww:w\in\{a,b\}^+\}$$

9. 构造图灵机使之能计算函数

$$f(w)=w^R$$

其中$w\in\{0,1\}^+$。

10. 设计一个图灵机，它能够找到偶数长度符号串的中间位置。例如，如果$w=a_1a_2\cdots a_na_{n+1}\cdots a_{2n}$且$a_i\in\Sigma$，则图灵机将输出结果$\hat{w}=a_1a_2\cdots a_nca_{n+1}\cdots a_{2n}$其中$c\in\Gamma-\Sigma$。

11. 设计一批能计算如下函数的图灵机，其中x与y是正整数且采用一元表示法。

(a) $f(x)=3x$

(b) $f(x,y)=x-y,\quad x>y$

$\qquad\qquad =0,\quad x\leqslant y$ 237

(c) $f(x,y)=2x+3y$

(d) $f(x)=\dfrac{x}{2},\quad$ 如果x是偶数

$\qquad\quad =\dfrac{x+1}{2},\quad$ 如果x是奇数

(e) $f(x) = x \bmod 5$

(f) $f(x) = \left\lfloor \frac{x}{2} \right\rfloor$，其中$\left\lfloor \frac{x}{2} \right\rfloor$表示小于或者等于$\frac{x}{2}$的最大整数

12. 设计一个图灵机，其中$\Gamma = \{0, 1, \square\}$，使得如果图灵机开始于任意包含空白符或1的单元，则当且仅当带上有一个0时，它才停机。
13. 写出例9.8的完整解决方案。
14. 针对例9.10中的图灵机，当输入为111时，给出相应的瞬时描述序列。当输入为110时，情况又会怎样？
15. 证明例9.10中的图灵机确实能够执行指定的计算。
16. 给出例9.11的详细内容。
17. 假定在例9.9中，x与y以二进制作为表示法，设计一个图灵机能够在该表示法下完成指定的计算。
18. 如果例9.9中，x与y采用十进制表示法，给出针对它的解决方案梗概。
19. 注意到本节实例中的图灵机都仅有一个终态。是否存在一般性的结论：对于任意的图灵机，都存在另一个只有一个终态的图灵机能够接受同样的语言。
20. 在定义9.2中，图灵机接受的语言不包含空串。试改变该定义使之能够接受包含有λ的语言。

9.2 完成复杂任务的组合图灵机

我们已经明确说明了所有那些计算机中都提供的重要操作是怎样通过图灵机实现的。既然在数字计算机中，这些基本操作是组成更为复杂指令的构造块，那么让我们来看看这些基本操作在图灵机中是怎样组合在一起的。为了说明图灵机是如何组合的，我们将遵循程序设计中的一个共同惯例。从程序的一个高层描述开始，不断地细化，直到程序能够用我们使用的实际语言描述为止。我们能够以多种方式在高层描述图灵机，后续的讨论中，我们将主要以块图或伪码作为描述方式。在块图中，我们将计算封装在盒子中，并描述了它的功能，但其内部细节是不可见的。通过使用这种盒子，我们已经隐含地表明了它们确实是可以构造的。作为第一个例子，我们把例9.9与例9.11中的图灵机加以组合。

238

例9.12 设计一个图灵机能够计算函数

$$\begin{aligned} f(x, y) &= x + y, \quad \text{如果} x \geqslant y \\ &= 0, \quad \text{如果} x < y \end{aligned}$$

为了便于讨论，我们假定x与y都是正整数并且采用一元表示法。值零将在带上表示成0且带上的其他部分为空白符。

采用基于图的高层描述方式，$f(x, y)$ 的计算可以可视化地表示为图9-5。在该图中，我们将首先使用一个类似例9.11中的比较机器来确定是否有$x \geqslant y$。如果$x \geqslant y$，则比较器给加法器发一个开始信号，开始计算$x + y$。否则，启动一个擦除程序，将所有的1替换成空白符。

在接下来的讨论中，我们将经常使用图灵机的这种高层的块图表示形式。这比用相应δ的扩展集合更加快捷、清楚。在我们接受这一高层视图之前，首先需要验证它的可行性，比如比较器给加法器发送一个开始信号到底是什么意思？定义9.1没有对这种可能性给出说明。然

而我们可以通过一种简单的方式来进行说明。

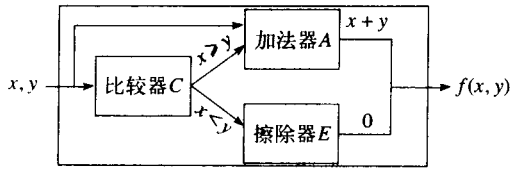


图 9-5

比较器C的程序是根据例9.11进行编写的，使该图灵机的状态下标为C。对于加法器，我们使用例9.9中的思想并使自动机的状态下标为A。而对于擦除器E，我们构造了一个状态下标为E的图灵机。由C完成的计算是

$$q_{C,0}w(x)0w(y) \vdash^* q_{A,0}w(x)0w(y), \quad 如果x \geqslant y$$

与

$$q_{C,0}w(x)0w(y) \vdash^* q_{E,0}w(x)0w(y), \quad 如果x < y$$

239

如果我们分别以$q_{A,0}$与$q_{E,0}$作为A与E的初态，可以看出C将启动A或者E。

由加法器完成的计算为

$$q_{A,0}w(x)0w(y) \vdash^* q_{A,f}w(x+y)0$$

而擦除器E将完成

$$q_{E,0}w(x)0w(y) \vdash^* q_{E,f}0$$

结果将得到单个图灵机，如图9-5所示，它组合了C，A与E的工作。 □

图灵机另一个有用的高层视图是采用伪码作为描述。在计算机程序设计中，伪码采用一些易于理解的描述性词汇简要地描述一个计算。然而，这种描述在计算机中是不可用的，但我们假定它能够转换成一种我们需要的合适语言。可以用宏指令做例子来介绍一种简单的伪码，宏指令是单个语句的缩写，但是代表了一个低层语句的序列。首先，我们根据低层语言定义宏指令，并假定相关的低层代码将替换该宏指令的每一次出现，这样就可以在程序中使用宏指令了。这一思想在图灵机程序设计中非常有用。

例9.13 考虑宏指令

$$\textit{if } a \text{ then } q_j \text{ else } q_k$$

它的含义是：如果图灵机读到a，无论当前处于什么状态，都将进入状态q_j，且不改变带的内容或读写头的位置。如果读到的符号不是a，该图灵机将进入状态q_k，同样不改变带的内容或读写头的位置。

实现该宏指令只需要图灵机的几个简单相关操作：

$$\delta(q_i, a) = (q_{j0}, a, R) \quad 对任意的q_i \in Q$$
$$\delta(q_i, b) = (q_{k0}, b, R) \quad 对任意的q_i \in Q及任意的b \in \Gamma - \{a\}$$
$$\delta(q_{j0}, c) = (q_j, c, L) \quad 对任意的c \in \Gamma$$
$$\delta(q_{k0}, c) = (q_k, c, L) \quad 对任意的c \in \Gamma$$

状态q_{j0}与q_{k0}是新引入的状态，它们用于处理标准图灵机中每一步迁移读写头都将移动这一事实所带来的问题，因为在该宏指令中，我们需要改变状态但不移动读写头。这样就可以先让读写头右移并使图灵机进入状态q_{j0}或者q_{k0}，也就使得在进入要求的状态q_j或者q_k之前需要执行一次左移。 □

240

我们可以进一步使用子程序来替换宏指令。一般而言，宏指令在其每次出现的地方将会被实际代码替换，而子程序作为一段代码块，可以在需要的地方被反复地调用。子程序是高级程序设计语言的基本成分，它同样可以用在图灵机中。为了说明其合理性，我们将简要地描述一个图灵机是怎样作为一个子程序被其他图灵机反复调用的。这里图灵机需要一个新特性：能够存储调用程序的格局信息，使得从子程序返回时能够重建该格局。例如，图灵机A在状态q_i下调用图灵机B。当B完成时，我们将希望在状态q_i下恢复程序A，并使得其读写头（它可能在B的操作中发生了移动）保持原来的位置。其他时候，A可能会在状态q_j下调用B，调用完成后，控制应该返回到状态q_j，为了解决这一控制转移问题，我们必须能够从A到B、从B到A地传递信息并且保证临时挂起的计算A不会受正在执行的B的影响。为了能这么做，我们将带分成如图9-6所示的几个区域。

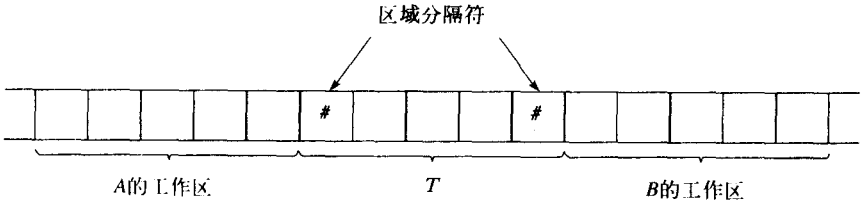


图 9-6

在A调用B之前，它在带区域T中放置B需要的信息（比如A的当前状态，B的参数等）。然后A通过一个转移函数进入B的开始状态并将控制交给B。在转移完成后，B将在T中找到其相应的输入。由于B的工作空间与T以及A的工作空间是分开的，所以它们之间不会出现干扰。当B完成时，它将相关的结果返回到区域T中，A将在这里找到相应的信息。通过这种方式，这两个程序能够以所需的方式相互作用。可以看出这与实际计算机中的子程序调用十分类似。

241

现在，如果我们知道（至少在理论上知道）怎样将伪码转换成一个实际的图灵机程序，就可以采用伪码对图灵机进行编程了。

例9.14 设计一个图灵机，它能够将两个以一元表示法表示的正整数相乘。

乘法图灵机可以通过组合加法与拷贝的思想来进行构造。我们假定初始与最终的带内容如图9-7所示。乘法过程可以被看成是对于乘数x中每一个1重复地拷贝被乘数y的过程，使得符号串y每次都能够加入到部分的计算乘积中。下面的伪码说明了该过程的主要步骤：

1. 重复下述过程直到x中没有1为止：

在x中找一个1并用另一个符号a替换它；

将最左边的0替换成0y。

2. 将所有的a替换成1。

尽管上述伪码很简略，它的思想也足够简单，但却可以毫无疑问地完成计算。 □

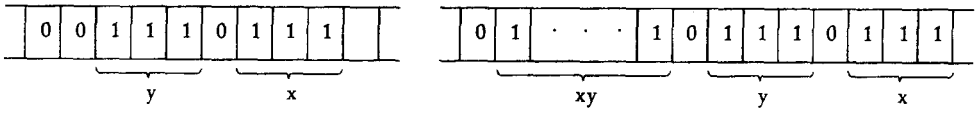


图 9-7

无论这些例子采用了什么样的描述方式，我们都可以据其推测：基本的图灵机能够以多种方式组合而使其非常强大。但同时由于我们给出的例子不是一般性的，也不足够详细，因此我们也就不能说通过它们得到了什么证明。但对于图灵机可以完成一些非常复杂的事情这一论点，我们似乎是可以接受的。

习题

1. 给出例9.14的完整解决方案。 242
2. 给出一元表示法表示正负整数的约定。基于这一约定，概述用于计算$x-y$的减法器的构造。
3. 利用加法器、减法器、拷贝器、比较器以及乘法器，画出能够计算如下函数的图灵机块图。其中n是正整数。
   (a) $f(n) = n(n+1)$
   (b) $f(n) = n^5$
   (c) $f(n) = 2^n$
   (d) $f(n) = n!$
   (e) $f(n) = n^{n!}$
4. 使用块图概要地描述函数f的实现，f定义如下：对于所有$w_1, w_2, w_3 \in \{1\}^+$，有

$$f(w_1, w_2, w_3) = i$$

   其中i满足$|w_i| = \max(|w_1|, |w_2|, |w_3|)$，条件是没有两个$w$的长度是相同的；否则$i=0$。
5. 给出在$\{a, b\}$上能够接受如下语言的图灵机的高层描述，并对每一个问题定义一组合适的且易于实现的宏指令集合，并在高层描述的解决方案中使用它们。
   (a) $L = \{ww^R\}$
   (b) $L = \{w_1w_2 : w_1 \neq w_2 : |w_1| = |w_2|\}$
   (c) (a) 中语言的补
   (d) $L = \{a^nb^m : m = n^2, n \geqslant 1\}$
   (e) $L = \{a^n : n$是一个质数$\}$
6. 给出用图灵机表示有理数的方法，在此基础上概述如何针对这些数执行加法与减法操作。
7. 概述能够对十进制表示的正整数x与y执行加法与乘法操作的图灵机的结构。
8. 给出宏指令

$$\text{searchright}\ (a, q_i, q_j)$$

的实现。该指令用于从带的当前位置向右搜索第一个符号a，如果在一个空白符出现之前碰到a，则该图灵机将进入状态q_i，否则进入状态q_j。 243
9. 使用习题8中的宏指令设计一个基于$\Sigma = \{a, b\}$的图灵机能够接受语言$L(ab^*ab^*a)$。

10. 使用习题8中的宏指令创建一个图灵机，它能够将最左边的a左边紧邻的符号替换成空白符。如果输入中没有a，则最右的非空白符号被替换成b。

9.3 图灵论题

前述讨论不仅说明图灵机能够利用一些比较简单的部件进行构造，而且也说明了采用这种低级自动机进行工作的不方便之处。将块图或者伪码转换成相应的图灵机程序几乎不需要什么想像力与创造性，它只是一个耗时且容易犯错的过程，而对我们的理解没有什么帮助。由于图灵机指令集合的限制过于严格，使得对任何稍复杂一点的问题，它们的解决方案与证明都将非常繁琐。

现在我们面临一个两难的处境：为了说明图灵机不仅能执行一些可以明确给出程序的简单操作，而且也能解决一些可用块图或伪码描述的更为复杂的问题，我们需要明确地给出相应程序，而这却是一件痛苦且没有任何吸引力的工作，可能的话，应该避免这么做。因此，我们希望能够找到一种关于图灵机的合理的严格讨论，而又不需要编写冗长而低级的代码。不幸的是，还没有一种完美的方案能够使我们摆脱这样的困境；我们只能对此给出一个合理的折中方案。为了说明我们是怎样得到这一折中方案的，我们将转向有点哲学性的话题。

从前一节的例子中，我们可以得到一些简单的结论。第一点就是图灵机比下推自动机更为强大（对于这点的讨论，见本节后边的习题2）。例9.8对一个非上下文无关语言，概述了接受它的图灵机的构造，但不存在下推自动机能够接受这个语言。而例9.9、例9.10与例9.11说明了图灵机能够完成一些基本的算术运算，符号串处理以及一些简单的比较。同样也说明了基本操作是怎样组合起来解决更为复杂的问题的，多个图灵机是怎样组合起来的以及一个程序是怎样作为另一个程序的子程序的。由于图灵机具备这种建立复杂操作的方法，我们猜想图灵机在功能上能够与一个典型的计算机相当。

244 我们可能在某种意义上推测图灵机在功能上与典型的电子计算机是等价的。但我们怎样能证明或者反驳它呢？对于证明，我们可能会采用一系列更为复杂的问题来说明图灵机是怎样解决它们的。我们也可能选取某一特定计算机的机器语言指令集，然后设计一个图灵机使其能够执行该集合中的所有指令。毫无疑问，这么做需要巨大的耐心，但如果这一假定是正确的，也是可以这么做的。在这一方向上的每一个成功都能进一步验证我们的推测，但它并不能够证明。困难在于我们并不能准确地知道什么是“典型的数字计算机”，也无法给出它的准确定义。

我们也可以从另一个方面来解决这一问题。努力找出一些过程，对这些过程能够编写出相应的计算机程序，如果没有相应的程序则表明，这些过程不存在相应的图灵机。如果能够找到没有相应程序的过程，我们就有了反驳这一推测的根据。但是，目前还没有人能够找到这样的一个反例，所有的这种尝试都失败了，这个事实可以作为例证来说明是不可能找到这样的反例的。所有的这些都在说明着图灵机与任何计算机基本上是同样强大的。

对这一类型问题的争论，导致A. M. Turing和其他的一些计算机科学家在20世纪30年代中期给出了称为*图灵论题*（Turing thesis）的著名猜想。该猜想认为任何可以采用机械方式完成的计算都可以采用图灵机来完成。

这是一个总括性的陈述，因此对我们而言，了解图灵论题是很重要的。但它还没有得到证明。要想证明这个论题，我们需要准确地定义术语“机械方式”，为此，需要某种其他的抽象模型，而这使得我们无法前行。将图灵论题看成是对机械计算的定义将更为合理，而一个计算被称之为机械计算当且仅当它能够被某一图灵机执行。

如果我们相信这一观点并简单地认为图灵论题是一个定义，则对这一定义是否足够的宽泛，我们将提出一些问题。是否它能够覆盖我们现在采用计算机能完成（以及在未来可能的）的所有事情？对此，还没有一个明确的肯定回答，但是存在一些很强的、有利于它的证据。对于接受图灵论题作为机械计算的定义，我们存在如下一些论点：

1. 任何能够被现有数字计算机完成的事情，图灵机同样可以完成。

2. 到目前为止，还没有人能够给出这样一个问题，它能通过我们直观上能够想到的算法进行解决，却写不出它相应的图灵机程序。

3. 曾经提出的一些其他机械计算模型，都不比图灵机更强大。 245

这些论据都是基于具体情况的，因而图灵论题并不能由此得到证明。尽管图灵论题看起来是合理的，但它仍是一个假设。将图灵论题简单地看作是一个定义会遗漏一个重要的论点，那就是在某种意义上，图灵论题在计算科学中发挥的作用就像化学与物理中的基本规律一样。比如，经典物理学主要是基于牛顿运动定律。虽然我们称之为定律，但它们并没有逻辑上的必然性，更确切地说，这些似是而非的理论能够解释物理世界中的大部分现象。由于根据它们得到的结论与我们的经验以及观察相符合，因而我们接受了它们。但我们并不能证明这些规律是正确的，尽管它们甚至可能是错误的。如果一个实验的结果与根据这些规律得到的结论冲突，我们就会对其有效性提出怀疑。另一方面，试图推翻一个规律的努力反复失败增强了我们对这个规律的信心。图灵论题也是同样的情况，因此我们有理由将其作为计算机科学的一条基本规律。从它得到的结论与现实计算机得到的结论是符合的，而且到目前为止，很多试图推翻这一规律的尝试都失败了。也许有一天有人能够给出一个新的定义，它能够解决一些图灵机所不能解决的问题，但目前，这仍不在我们对于机械理论认识的直观概念范围之内。如果出现这种情况，则我们接下来的很多讨论都需要做重大的修改，但这种情况的可能性很小。

如果我们接受图灵论题，现在就能够给出算法的准确定义了。

定义9.5 *实现函数$f: D \to R$的算法是一个图灵机M，它以带上的任意$d \in D$作为输入，最终以带上的正确结果$f(d) \in R$作为输出并停机。形式化地，我们可以要求对于所有的$d \in D$，有*

$$q_0 d \vdash_M^* q_f f(d), q_f \in F$$

采用图灵机程序来识别一个算法，我们就能够严格地证明诸如“存在一个算法”或者“不存在一个算法”这样的命题。然而，要明确构造一个算法来解决相对简单的问题都将是一件繁琐的工作。为了避免这种情况，我们就可以根据图灵论题宣称，在任何计算机上可以完成的事情同样可以在图灵机上完成。因而，我们也就能够在定义9.5中使用“Pascal程序”替 246
换“图灵机”，这将明显减轻给出相关算法的负担。实际上，我们已经这么做了，更进一步地，假设对于口头描述或块图，我们能够根据需要给出与它们相应的图灵机程序，我们就能接受其为算法。这大大简化了我们的讨论，但同时这也使我们面临一个困境。“Pascal程序”是良定义的，而“易懂的口头描述”却不是，这使我们处在把不存在算法说成存在的危险中。但

由于我们可以通过它保证讨论简单、清晰，能够对相当复杂的过程给出简洁的描述，使我们的冒险得到一定的补偿。如果读者怀疑这些命题的正确性，可以通过使用某种程序设计语言编写一个合适的程序来进行反驳。

习题

★★1. 考虑一个计算机的机器语言集合。简要说明集合中的各种指令是如何在图灵机中执行的。

2. 在上面的讨论中，我们已经指出图灵机比下推自动机更为强大。由于图灵机的带能够像栈一样地工作，从这似乎可以看出，图灵机确实比下推自动机更为强大。指出在上述论据中，哪一个重要的因素没有考虑？●

★★3. 在通俗文献中有很多有意思的关于图灵机的论文。其中由J. E. Hopcroft于1984年5月发表在《科学美国人》(*Scientific American*)上的，名为“图灵机”的论文就是很好的一篇。这篇论文讨论了我们本章介绍的关于图灵机的思想，并且给出了一些图灵和其他人工作的历史背景。阅读这篇文章，并写一个简要的评论。

247

第10章 图灵机的其他模型

我们在上一章中给出的标准图灵机的定义并不是唯一可行的，还存在着若干等价的定义。我们所能得出的关于图灵机能力的结论在很大程度上独立于为之选定的特定结构。在本章中，我们将再介绍几个更加复杂的图灵机模型，在某种意义上它们和标准图灵机的能力是等价的。

如果我们接受图灵论题，则我们可以认为，通过赋予标准图灵机更加复杂的存储设备，并不会对这种自动机的能力有任何影响。任何能被这种新型图灵机执行的计算，均属于机械计算（mechanical computation）的范畴，因此，也能够通过标准图灵机来完成。仅仅从证明我们所预期的标准图灵机的能力从而增加我们对图灵论题的信心这一点看，研究更加复杂的模型也是有益的。定义9.1中所介绍的基本模型的许多变种都是可行的，比如有多条带的图灵机或者多维带的图灵机。本章我们将考虑在后续讨论中有用的图灵机的变种。 249

我们还将研究非确定型图灵机，并证明它们和确定型图灵机的能力等价。这是出乎我们意料之外的，因为图灵论题仅仅覆盖了机械计算的内容，并没有讨论在非确定型中所隐含的智能猜测。图灵论题中没有直接解决的另一个问题是关于同一台机器在不同的时间执行不同的程序，这导致了“可再编程的”或者“通用的”图灵机思想的出现。

最后，我们还将介绍*线性有界自动机*（linear bounded automata），这个概念将在后续章节中用到。所谓线性有界自动机，是指具有一条*无穷带*（infinite tape），但只能以一种受限的方式使用这条带的一类图灵机。

10.1 对图灵机的较小修改

我们首先考虑对定义9.1做一些较小的改动，并探讨这些改动是否导致了一般概念上的差异。每次我们改动一个定义都会引入一种新型的自动机，并引起如下问题：这些新型的自动机和以前我们所遇到的自动机是否有实质上的区别？我们所说的一类自动机和另一类自动机之间的“实质上的区别”是指什么呢？虽然它们的定义之间可能有着明显的差异，但是这些差异并不一定能导致令我们感兴趣的结果。我们可以在前面章节中介绍的确定型和非确定型有穷自动机中看到这一点：这两种自动机的定义迥然不同，但在它们都精确对应于正则语言族这一意义上，这两种自动机是等价的。依此推断，我们可以在一般意义上定义*自动机类*（classes of automata）的等价性或不等价性。

10.1.1 自动机类的等价性

当我们定义两个自动机或者自动机类的等价性时，我们必须首先明确这种“等价性”意味着什么。在本章的后续部分中，我们将遵循为确定型有穷接受器（dfa）和非确定型有穷接受器（nfa）所建立的先例，用接受语言的能力来定义“等价性”。

定义10.1 *两个自动机是等价的，如果它们能接受同样的语言。考虑两个自动机类C_1和C_2，如果对于C_1中的任意自动机M_1，都存在C_2中的自动机M_2，满足* 250

$$L(M_1) = L(M_2)$$

我们就称C_2具有至少和C_1相同的能力。反之亦成立，即对于C_2中的任意自动机M_2，都存在C_1中的自动机M_1，满足$L(M_1) = L(M_2)$，我们就称C_1和C_2是等价的。

有多种方式可以证明自动机的等价性。定理2.2中的构造性方法被用于证明dfa和nfa的等价性。为了证明图灵机的等价性，我们经常采用一种重要的称作模拟（simulation）的技术。

设M是一个自动机，我们称自动机$\hat{M}$可以模拟M的一个计算，如果$\hat{M}$可以按照如下的方式模仿M的计算。设$d_0, d_1, \cdots$是M计算的瞬时描述序列，即

$$d_0 \vdash_M d_1 \vdash_M \cdots \vdash_M d_n \cdots$$

则称$\hat{M}$模拟这个计算，如果$\hat{M}$可以执行和M类似的如下计算:

$$\hat{d}_0 \vdash^*_{\hat{M}} \hat{d}_1 \vdash^*_{\hat{M}} \cdots \vdash^*_{\hat{M}} \hat{d}_n \cdots$$

其中$\hat{d}_0, \hat{d}_1, \cdots$是瞬时描述，并且每个瞬时描述都对应于$M$的唯一一个状态。换句话说，如果我们知道$\hat{M}$执行的计算以及初始格局，我们就能准确地确定$M$所执行的计算。

应该注意的是，M的一次迁移$d_i \vdash_M d_{i+1}$在$\hat{M}$的模拟中可能对应于若干次迁移。迁移$\hat{d}_i \vdash^*_{\hat{M}} \hat{d}_{i+1}$的中间格局可能不对应于$M$的任何格局，但如果我们能确定$\hat{M}$对应的格局，就不会有任何影响。只要我们能从$\hat{M}$的计算确定$M$的计算，这种模拟就是正确的。如果$\hat{M}$能够模拟$M$的任何计算，则我们称$\hat{M}$能够模拟$M$。我们应该清楚，如果$\hat{M}$能够模拟$M$，那么问题就转化为它们能够接受相同的语言，于是它们是等价的。为了证明两个自动机类的等价性，我们需要证明对于一类中的任何一个自动机，在另一类中都存在着可以模拟它的自动机。

10.1.2 带不动选择的图灵机

251 在标准图灵机定义中，读写头要么是向左移动要么是向右移动，而有时为了方便，我们引入第三种选择，即在读写头重写带上单元后位置保持不动。于是，我们可以修改定义9.1，得到新的带不动选择的图灵机（Turing machine with stay-option）的定义，办法是将定义9.1中的δ替换为

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$$

S表示读写头位置保持不动。这一改动并没有增强标准图灵机的能力。

定理10.1 带不动选择的图灵机类与标准图灵机类等价。

证明：因为带不动选择的图灵机显然是标准图灵机的扩展，于是任何标准图灵机无疑可以被带不动选择的图灵机模拟。

为了证明逆命题（即标准图灵机可以模拟带不动选择的图灵机），我们令$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$为一带不动选择的图灵机，用一个标准图灵机$\hat{M} = (\hat{Q}, \Sigma, \Gamma, \hat{\delta}, \hat{q}_0, \square, \hat{F})$模拟它。对于$M$的每一步，模拟机$\hat{M}$都做如下工作：如果$M$的读写头向左移动或者向右移动，则$\hat{M}$的读写头也相应地向左或向右移动；如果$M$执行了动作$S$（即读写头保持不动），则$\hat{M}$执行两步动作：读写头首先重写单元格并向右移动，然后对新移动到的单元格不做任何修改并向左移动。这一模拟机可以通过定义$\hat{\delta}$由M构造如下:

对于每一步转移

$$\delta(q_i, a) = (q_j, b, L \text{ or } R)$$

我们把$\hat{\delta}$解释为

$$\hat{\delta}(\hat{q}_i, a) = (\hat{q}_j, b, L \text{ or } R)$$

对于每一步读写头不动的转移

$$\delta(q_i, a) = (q_j, b, S)$$

我们把$\hat{\delta}$解释为相应的两步转移

$$\hat{\delta}(\hat{q}_i, a) = (\hat{q}_{js}, b, R)$$

和

$$\hat{\delta}(\hat{q}_{js}, c) = (\hat{q}_j, c, L)$$

其中$c \in \Gamma$。

很明显，对M的每一个计算，都有对应的$\hat{M}$的一个计算，因此$\hat{M}$可以模拟M。■ 252

模拟是证明自动机等价的一种标准技术。我们在上一个定理中所描述的形式体系使得准确地讨论过程和证明有关等价性的定理成为可能。在后续讨论中，我们会频繁地使用模拟的思想，但一般我们不会试图严格并具体地描述所有细节。图灵机的完全模拟通常是很繁琐的，为了避免这一点，我们的讨论只限于描述性的，而非定理证明的形式。虽然对于模拟我们只给出总体描述，但这些描述同样能够易于转化为严格的证明。读者将会发现用高级语言或者伪码描述每个模拟是有益的。

在介绍其他模型之前，我们再对标准图灵机做一点评述。从定义9.1可以看出，每个带符号都可以是一些符号的组合体，而不仅仅是单个符号。通过对图9-1的扩展（图10-1）我们可以更清楚地认识到这一点。在图10-1中，每一个带符号都是三个较简单的字母构成的三元组。

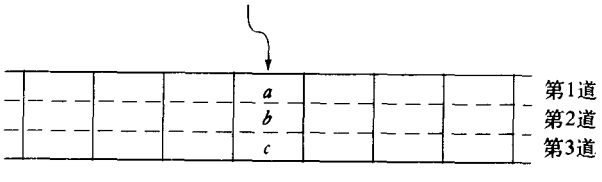


图 10-1

在图10-1中，我们将带上的每个单元划分为三个部分，每部分都称为*道*（track），每个道包含三元组的一个成员。基于这一直观认识，这样的图灵机有时被称为*多道图灵机*（Turing machine with multiple tracks）。但这一观点并没有扩展定义9.1，因为我们所要做的只是将Γ改为一个新的字母表，其中每个符号包含几个部分。

然而，其他图灵机模型就需要对定义做一些修改，因此这些图灵机与标准图灵机的等价性是需要证明的。我们来看两个这样的模型，这两个模型有时也被作为图灵机的标准定义。一些较少见的图灵机变形将在本节末的习题中提出。

10.1.3 单向无穷带图灵机

很多作者并不把图9-1中的模型作为标准图灵机模型，而是采用单向无穷带图灵机（Turing machine with semi-infinite tape）。我们可以把这一图灵机直观地理解为带只有左边界（图10-2）。这一模型与我们的标准模型同样是等价的，区别只在于当读写头位于带的左边界时不能向左移动。

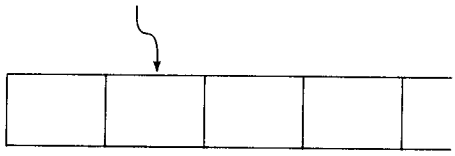


图 10-2

不难看到，这一限制并没有影响图灵机的能力。为了用单向无穷带图灵机 $\hat{M}$ 模拟标准图灵机M，我们采用图10-3所示的带。

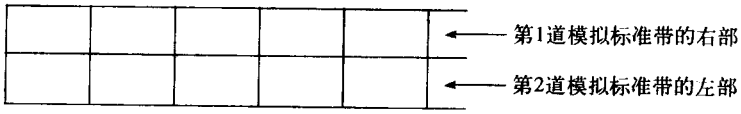


图 10-3

模拟机 $\hat{M}$ 有一条双道带。在上道中，我们保存位于M带上某一参考点右侧单元的内容，例如可以选择计算开始时读写头的位置作为参考点。在下道中，我们以逆序保存位于M带上参考点左侧的内容。我们可以对 $\hat{M}$ 编程，使得当M的读写头位于参考点右侧时，$\hat{M}$ 使用上道的信息；当M的读写头位于参考点左侧时，$\hat{M}$ 使用下道的信息。M的读写头是否位于参考点右侧可以通过将 $\hat{M}$ 的状态分为两部分以辅助进行判断，比如设这两类状态分别为Q_U和Q_L：当 $\hat{M}$ 工作于上道时使用前者，当 $\hat{M}$ 工作于下道时使用后者。左边界处的上下道中均包含特殊的结束符#，以便于两个道之间的切换。例如，被模拟的图灵机M和模拟机 $\hat{M}$ 分别处于图10-4所示的格局，并且将要模拟的迁移由

$$\delta(q_i, a) = (q_j, c, L)$$

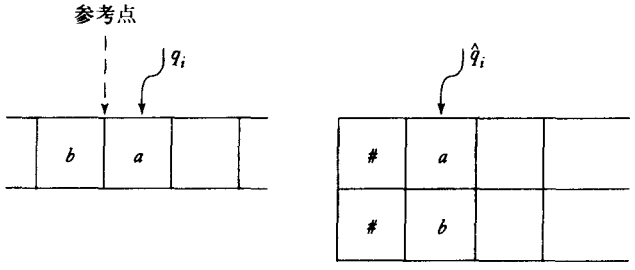
生成。模拟机 $\hat{M}$ 先通过转移

$$\hat{\delta}(\hat{q}_i, (a, b)) = (\hat{q}_j, (c, b), L)$$

来进行迁移，其中 $\hat{q}_i \in Q_U$ 。因为 $\hat{q}_i$ 属于Q_U，所以此处只考虑上道的信息。此时，模拟机在状态 $\hat{q}_j \in Q_U$ 看到了符号（#, #），接下来作转移

$$\hat{\delta}(\hat{q}_j, (\#, \#)) = (\hat{p}_j, (\#, \#), R)$$

其中 $\hat{p}_j \in Q_L$ ，产生格局如图10-5所示。现在模拟机的状态属于Q_L并将工作于下道。由此我们可以直接得到此模拟过程的更详细的描述。



a) 被模拟的图灵机　　b) 模拟机

图 10-4

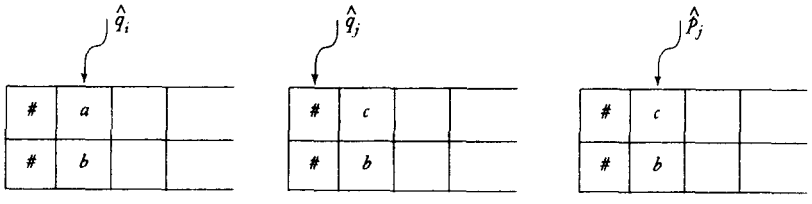


图10-5　模拟$\delta(q_i, a) = (q_j, c, L)$的格局序列

10.1.4 离线图灵机

第1章中自动机的一般性定义中包含一个输入文件和一个临时存储空间。在定义9.1中，为了简单起见，我们去掉了输入文件，并声明这样做并不会改变图灵机的概念。我们现在就详细地讨论这一点。

如果我们在模型中加入输入文件，我们所得到的就是离线图灵机（off-line Turing machine）。在这类机器中，每一步转换都是由内部状态、从输入文件当前读到的符号以及读写头所见到的带上的符号控制的。离线图灵机的图示如图10-6所示。离线图灵机的形式化定义是很容易给出的，但我们把这留作一个习题。我们只想简要地说明为何离线图灵机类与标准图灵机类是等价的。

首先，任何标准图灵机的行为都可由离线图灵机模拟。模拟机所要做的只是将输入文件的内容拷贝到带上，然后便能以与标准图灵机相同的方式工作。

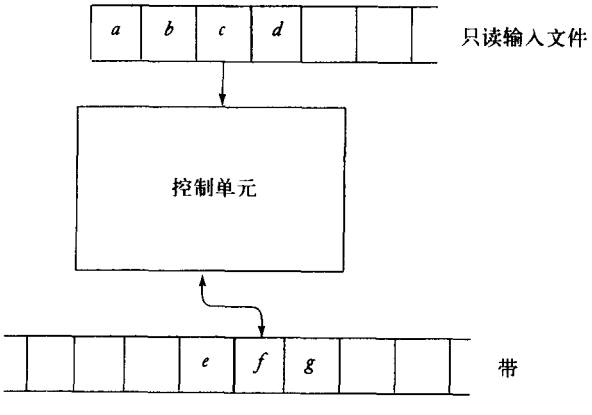


图 10-6

对标准图灵机 $\hat{M}$ 如何模拟离线图灵机M的描述要稍长一些。标准图灵机可以通过使用图10-7所示的四道带模拟离线图灵机。图10-7所示的带的内容代表了图10-6中的格局。$\hat{M}$ 的四道中的每一道在模拟中都有特定的功能：第一道包含M的输入；第二道记录M的输入文件的当前输入位置；第三道代表M的带；第四道记录M的读写头的位置。

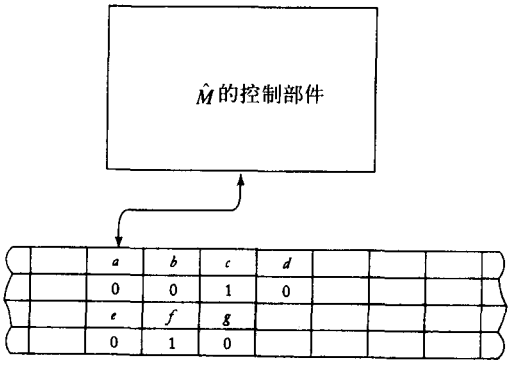


图 10-7

$\hat{M}$ 对M的每步转移的模拟都需要若干步。从某个标准位置开始，比如左边界，借助于特殊结束符所标记的相关信息，$\hat{M}$ 在第二道寻找M输入文件的输入位置。找到第一道上的相应单元后，$\hat{M}$ 将此信息记录在我们为此目的而设置的控制部件状态中。接下来，$\hat{M}$ 在第四道上寻找M读写头的位置。综合在状态中记录的输入信息和第三道上的符号，我们就知道M要做的转移了。这一信息再次被记录在 $\hat{M}$ 的相应内部状态中。然后，$\hat{M}$ 将对其所有4条道进行改写，以反映M所作的转移。最后，$\hat{M}$ 的读写头返回标准位置以进行对M下一转移的模拟。

习题

253 ~ 256

1. 给出单向无穷带图灵机的形式化定义。并证明单向无穷带图灵机类与标准图灵机类等价。
2. 给出离线图灵机的形式化定义。
3. 证明任何能够被离线图灵机接受的语言也能被标准图灵机接受。
4. 考虑如下图灵机模型，在每一次迁移中，图灵机可以改写带符号，也可以移动读写头，但不能二者都做。
   (a) 给出此类图灵机的形式化定义。
   (b) 证明这种图灵机类与标准图灵机类等价。
5. 考虑如下图灵机模型，在每一次迁移中，读写头可以向左或向右移动一个以上的单元，移动距离和方向作为转移函数δ的参数。给出此类图灵机的准确定义，并描述如何用标准图灵机模拟此类图灵机。
6. 不可擦除图灵机（nonerasing Turing machine）不能将非空白符改写为空白符。这一点可以通过如下限制实现，如果

$$\delta(q_i, a) = (q_j, \square, L \text{ or } R)$$

那么a一定是□。证明这类图灵机与标准图灵机等价。

7. 考虑不能将带符号改写为空白符的图灵机，即对所有$\delta(q_i, a) = (q_j, b, L \text{ or } R)$，$b$一定属于$\Gamma - \{\square\}$。证明此图灵机模型能够模拟标准图灵机。
8. 假设我们要求图灵机只能在终态停机，即对于所有（q, a），$a \in \Gamma$且$q \notin F$，$\delta(q, a)$ 都要有定义。这一要求是否限制了图灵机的能力？
9. 假设我们要求图灵机在带上写的符号不能和带上该位置的原来符号相同，即如果

$$\delta(q_i, a) = (q_j, b, L \text{ or } R)$$

则a与b必须相异。这一限制是否降低了图灵机的能力？

10. 考虑图灵机的如下变形，转换不仅取决于读写头当前所在的单元符号，还取决于当前单元左边的一个单元符号及右边的一个单元符号。给出此类图灵机的形式化定义，并描述如何用标准图灵机模拟此类图灵机。
11. 考虑下类图灵机，其转移的决策过程不同于标准图灵机，转移仅在当前带符号不属于特定的符号集合时发生。例如

$$\delta(q_i, \{a, b\}) = (q_j, c, R)$$

257

仅在当前带符号既非a也非b时允许迁移发生。给出这一概念的形式化描述，并证明此类图灵机与标准图灵机模型等价。

10.2 具有更复杂存储的图灵机

标准图灵机的存储装置是如此简单，以致人们可能以为通过将存储装置复杂化有可能增强图灵机的能力。但事实并非如此，我们现在用两个例子加以说明。

10.2.1 多带图灵机

多带图灵机（multitape Turing machine）是一个有多条带的图灵机，每一条带都有一个被独立控制的读写头（图10-8）。

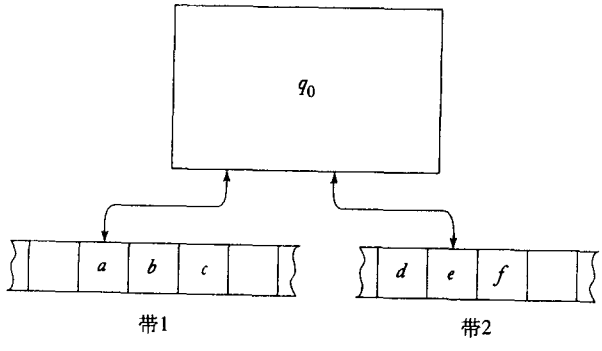


图 10-8

多带图灵机的形式化定义已经超出了定义9.1的形式，因为多带图灵机的转移函数不同于标准图灵机。一般我们如下定义n-带图灵机：$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$，其中$Q, \Sigma, \Gamma, q_0, F$的

定义与定义9.1相同，不同的是δ

$$\delta : Q \times \Gamma^n \rightarrow Q \times \Gamma^n \times \{L, R\}^n$$

它定义了发生在所有带上的转移。例如，若$n = 2$，当前格局如图10-8所示，则

$$\delta(q_0, a, e) = (q_1, x, y, L, R)$$

的解释如下：此条转移规则只能在图灵机处于状态q_0，第一个读写头当前所见符号为a，第二个读写头当前所见符号为e时应用，第一条带上的读写头将当前单元内容改写为x并向左移动，同时第二条带上的读写头将当前单元内容改写为y并向右移动，然后控制部件将状态改为q_1，图灵机进入下一个格局，如图10-9所示。

258

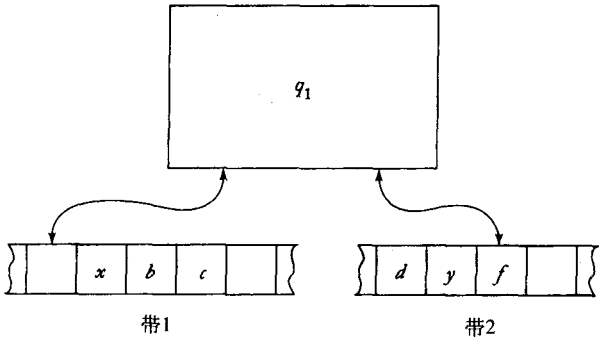


图 10-9

为了证明多带图灵机与标准图灵机的等价性，我们只需证明任何给定多带图灵机M都能被标准图灵机$\hat{M}$模拟，并且任何标准图灵机也都能被多带图灵机模拟。后者是显而易见的，因为我们总是可以让一个多带图灵机只在其一条带上做有用的工作。用只有一个带的标准图灵机模拟多带图灵机有些复杂，但在概念上仍是简单的。

考虑图10-10所示的2-带图灵机。模拟此2-带图灵机的单带图灵机的带有4个道（图10-11）。第一道表示M的第一条带；第二道的非空白单元除了表示M的第一条带读写头当前位置的那个单元是1外，其他单元都是0；类似地，第三道和第四道用来模拟M的第二条带。通过图10-11，我们可以清楚地看到，对于$\hat{M}$的每一个相关的格局（具有图10-11所示形式的格局），M都有唯一一个格局与之对应。

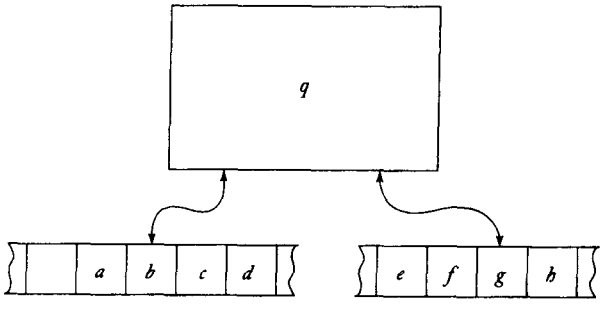


图 10-10

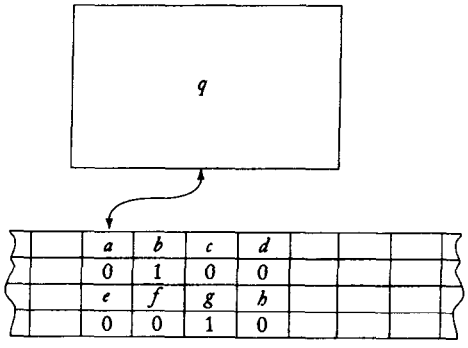


图 10-11

多带图灵机用单带图灵机表示，类似于离线图灵机用标准图灵机表示。实际的模拟步骤也非常相似，唯一的区别就是模拟多带图灵机时需要考虑多条带。我们可以沿用与离线图灵机的标准图灵机表示大体相似的结构，同时做一些小的改动。在沿用的同时，我们已经可以清楚地知道如何修改M的转移函数δ从而获得$\hat{M}$的转移函数$\hat{\delta}$了。虽然给出详细的描述并不困难，但很繁琐。当然，繁琐并不影响结论的正确性，即M能做的事情$\hat{M}$也能做。

需要注意的一点是，当我们说多带图灵机并不比标准图灵机功能更强大时，我们只是指它们所能做的工作相同，即能接受相同的语言。

例10.1 考虑语言$\{a^n b^n\}$。在例9.7中，我们构造了一个单带图灵机来接受这个语言，但方法繁琐。而用一个2-带图灵机就会使工作大为简化。假设在计算的初始状态，第一条带上的初始符号串为$a^n b^m$。然后，我们把所有a拷贝到第二条带上。当我们拷贝完最后一个a时，我们开始将第一条带上的b与第二条带上的a进行匹配。这样我们就可以判断a的个数和b的个数是否相等，同时无须读写头反复地来回移动。 □

259 ~ 260

我们应记住，图灵机的各种变形被认为是等价的，只是从能做什么事情这一意义上考虑，而不是从是否容易编程或我们可能关心的其他效率度量方面考虑。在第14章我们还会考虑这个重点。

10.2.2 多维图灵机

多维图灵机（multidimensional Turing machine）是一种其带在多个维度上都可以无限扩展的图灵机，图10-12所示为一个二维图灵机的图。

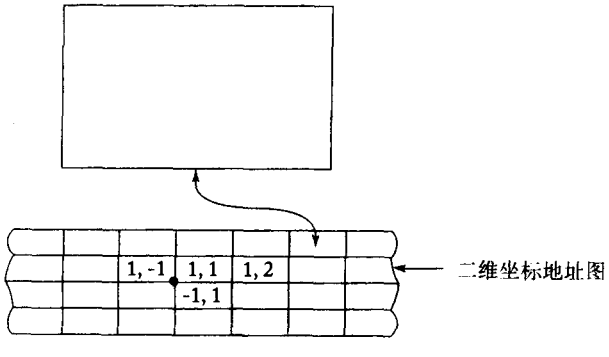


图 10-12

在二维图灵机的形式化定义中，转移函数δ具有下面的形式

$$\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, U, D\}$$

其中U和D分别表示读写头向上移动和向下移动。

为了用标准图灵机模拟二维图灵机，我们可以采用图10-13所示的2-道模型。首先，我们给二维图灵机的二维带的每个单元一个地址。这可以有很多种方法，例如图10-12所示二维坐标表示法。模拟机的两道之一用于记录单元内容，另一道用于保存相应单元的地址（称这个道为地址道）。假设在图10-12所示的格局中，单元（1，2）内容为a，单元（10，－3）内容为b，则相应的模拟机的格局如图10-13所示。注意一点：单元地址可以是无限大的整数，所以地址道不能用固定区域来存储地址，而应采用可变区域来存储，并用特殊符号分隔区域边界，如图10-13所示。

	a				b						
	1	#	2	#	1	0	#	-	3	#	

图 10-13

我们假设在每个迁移模拟的开始，二维图灵机M的读写头与模拟机$\hat{M}$的读写头总是分别位于相应的两个单元上。要模拟M的一步迁移，模拟机$\hat{M}$首先计算M的读写头将要移到的单元地址。采用二维坐标地址图，这一计算是非常简单的。当地址计算出来后，$\hat{M}$就在第二道上找到这一地址，然后按照M的迁移修改单元内容。此外，如果给定M，则构造$\hat{M}$的过程是相当简单的。

习题

我们对图灵机讨论的目的主要是想通过证明标准图灵机能够模拟更复杂的图灵机来增加图灵论题的可信度。不幸的是，具体的模拟总是很繁琐无趣。在下面的练习中，读者只需大致描述模拟过程即可，但要足够清楚以便能够从中看出如何构造具体的模拟。

1. 多头图灵机（multihead Turing machine）可以看成是一个有单一带、单一控制部件和多个独立读写头的图灵机。给出多头图灵机的形式化定义，并描述如何用标准图灵机模拟此类图灵机。
2. 给出多头-多带图灵机的形式化定义。并描述如何用标准图灵机模拟此类图灵机。
3. 给出单带多控制部件图灵机的形式化定义，其中每个控制部件都有一个自己的读写头。并说明如何用多带图灵机模拟此类图灵机。

★4. 队列自动机（queue automaton）的临时存储部分是一个队列。假设这个自动机是一个在线自动机，即它没有输入文件，而是在计算开始之前将要处理的符号串置于队列中。给出此类自动机的形式化定义，并将其能力与图灵机进行比较。

★5. 证明任何一个图灵机都可以被一个不超过六个状态的标准图灵机模拟。

6. 尽你的所能将习题5中所需的状态数目减少（提示：最少的可能数目是3）。

★7. 计数器（counter）是一个字母表只含两个符号的栈，一个栈开始符（stack start symbol）和一个计数符（counter symbol）。只有计数符能够入栈或出栈。计数器自动机（counter automaton）是一个以一个或多个计数器作为存储部件的确定型自动机。证明任意一个图

灵机都能被一个有4个栈的计数器自动机模拟。 261 ~ 262

8. 证明任何一个可以由标准图灵机完成的计算也能被一个带不动选择的、拥有至多两个状态的多带图灵机完成。●

9. 给出完成例10.1中计算的详细程序。

10.3 非确定型图灵机

虽然根据图灵论题似乎可以得出特定的带结构并不会影响图灵机能力的结论，但这一结论对于非确定型图灵机来说则是不成立的。因为非确定型引入了选择，从而具有某种非机械化的特性，此时就不能再用图灵论题了。为了证明非确定型图灵机与标准图灵机能力相同，我们需要更仔细地考察非确定性带来的影响。我们再次用模拟技术证明非确定性行为可以通过确定的方式加以处理。

定义10.2 非确定型图灵机类似于定义9.1中的图灵机，只是转移函数δ变为

$$\delta : Q \times \Gamma \rightarrow 2^{Q \times \Gamma \times \{L, R\}}$$

只要含有非确定性，δ的值域就是一个由可能发生的转移构成的集合。对每一步转移，图灵机都可以从集合中任意选择一个。

例10.2 如果一个图灵机有如下的转移函数

$$\delta(q_0, a) = \{(q_1, b, R), (q_2, c, L)\}$$

那么这个图灵机就是非确定型的。迁移

$$q_0aaa \vdash bq_1aa$$

和

$$q_0aaa \vdash q_2\square caa$$

都是可行的。 □ 263

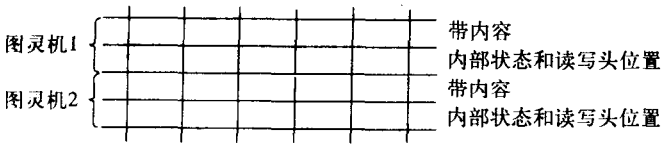
因为非确定性在计算函数中的作用并不明显，所以我们通常将非确定型自动机看作符号串的接受器。非确定型图灵机接受符号串w，如果存在某个可行的迁移序列满足

$$q_0w \vdash^* x_1q_fx_2$$

其中$q_f \in F$。一个非确定型自动机可能会在一系列迁移后进入非终态，也可能其所作的一系列转换是循环的从而没有结束的时候，但我们对诸如此类的转换序列并不感兴趣，我们感兴趣的是是否存在一个导致接受的迁移序列。

为了证明非确定型图灵机与确定型图灵机的能力相同，我们需要给出非确定过程的确定型等价形式。其实我们已经间接提供了一种等价形式。非确定性可以被看作确定型的回溯算法。确定型图灵机只要能记录回溯过程中的状态，就可以模拟非确定型图灵机。其实这很简单，为了理解这一点，让我们换一个角度看非确定性，这种观点在很多证明中都是有用的：一个非确定型图灵机可以被看作是一个在任何必要的时候都能复制自身的图灵机。当有多于一种迁移可供选择时，这种图灵机就会把自身复制成多个图灵机，然后让每个图灵机去执行一种可能的迁移。

从这一角度看非确定性，它便显得尤为非机械化。当然，当代计算机并不具备无限复制自身的能力。然而，确定型图灵机是能够模拟这一过程的。考虑一个具有二维带的图灵机(图10-14)。每两个横向的道代表一个图灵机。上道代表带，下道表示内部状态和读写头位置。每当要复制出一个新的图灵机的时候，就会在两个新的道上写入相应的信息。图10-15表示了例10.2中图灵机的初始格局以及后继格局。模拟机寻找所有活跃的道（由于这些道被特殊符号所包围，所以总能被找到），执行指定的迁移，如果有必要再复制自身。至此我们已经给出了模拟的大致过程，我们已经可以认识到这种模拟的正确性了，至于具体细节就留给读者自己完成。



264

图 10-14

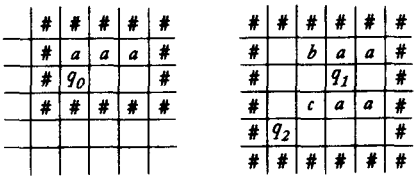


图10-15 一个非确定型迁移的模拟

基于上述模拟，我们可以得出结论：对于任何一个非确定型图灵机，都存在一个等价的确定型图灵机。由于这一结论非常重要，我们将其表述为定理。

定理10.2 *确定型图灵机类和非确定型图灵机类是等价的。*

证明：对于一个非确定型图灵机，用前面讲过的方法构造一个确定型图灵机，证明后者可以模拟前者。■

习题

1. 详细说明如何用确定型图灵机模拟非确定型图灵机。说明图灵机如何复制自身，如何找到当前活跃的图灵机，以及如何将已经停机的图灵机排除在后续考虑之外。
2. 说明如何用一个确定型图灵机模拟一台二维非确定型图灵机。
3. 为一个非确定型图灵机编写程序，使它接受如下语言

$$L = \{ww : w \in \{a, b\}^+\}$$

把它与确定型解决方法进行比较。●

4. 概要说明如何编写一个非确定型图灵机程序，使得此图灵机接受语言

$$L = \{ww^R w : w \in \{a, b\}^+\}$$

5. 为接受下述语言的非确定型图灵机编写一个简单的程序

$$L = \{xww^Ry : x, y, w \in \{a, b\}^+, |x| \geq |y|\}$$

如何确定性地解决这个问题?

6. 设计一个非确定型图灵机，使其接受语言 265

$$L = \{a^n : n\text{不是质数}\}$$ ●

★7. 双栈自动机（two-stack automaton）是一个具有两个独立的栈的非确定型下推自动机。为了定义这样的自动机，我们对定义7.1进行修改，使得

$$\delta : Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \times \Gamma \rightarrow \text{有限子集合} Q \times \Gamma^* \times \Gamma^*$$

一个迁移依赖这两个栈的栈顶值，迁移的结果把两个新值压入到这两个栈中。证明双栈自动机类与图灵机类等价。●

10.4 通用图灵机

让我们来看下面这段与图灵论题相悖的论断："一个如定义9.1所描述的图灵机是一台服务于特殊目的的计算机。一旦δ确定了，这台机器就被限制于执行某种特定的计算。而数字计算机却是通用计算机，人们可以对它们编程，使其在不同的时间做不同的工作。因此，图灵机与通用数字计算机是不等价的。"

我们可以通过设计一台可重编程的图灵机（reprogrammable Turing machine）来解决上述问题，我们把这类图灵机叫做通用图灵机（universal Turing machine）。通用图灵机M_u是这样一台自动机，它以任意一台图灵机M的描述和符号串w作为输入，并可以模拟M在w上的计算。为了建造这样一个M_u，我们首先选择一种图灵机的标准描述方式。我们可以不失一般性地假设

$$Q = \{q_1, q_2, \cdots, q_n\}$$

其中q_1是初态，q_2是唯一的终态。假设

$$\Gamma = \{a_1, a_2, \cdots, a_m\}$$

其中a_1表示空格符。然后我们选择一种编码方式使得q_1被编码为1，q_2被编码为11，依此类推。类似地，a_1被编码为1，a_2被编码为11，依此类推。符号0作为1串（仅由1构成的串）之间的分隔符。有了如上对初态、终态以及空格符的定义，我们就可以用δ完全描述任何一个图灵机了。我们用上述编码方式对转移函数进行编码，转移函数的参数和结果的顺序是事先确定的。例如，$\delta(q_1, q_2) = (q_2, a_3, L)$ 被编码为

$$\cdots 10110110111010 \cdots$$ 266

于是任何一个图灵机都可以被编码为一个有限的$\{0, 1\}^+$上的符号串，并且对任给的一个图灵机M的编码，我们都可以按唯一的方式解码。有一些符号串不代表任何图灵机（比如00011），但我们可以轻易地识别出这类符号串，所以可以不考虑它们。

一个通用图灵机M_u包括一个$\{0, 1\}$上的字母表和一个多带图灵机结构，如图10-16所示。

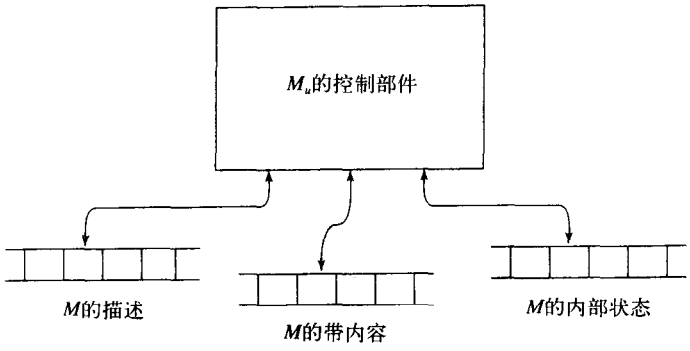


图 10-16

对于任意给定的输入M和w，1带用于记录M的定义编码，2带记录M的带内容，3带记录M的内部状态。M_u首先查看2带和3带的内容以决定M的当前格局，然后查看1带以决定M在此格局下的动作，最后，修改2带和3带以反映此次迁移的结果。

有理由建造一台实际的通用图灵机（例如Denning、Dennis和Qualitz 1978），但这一建造过程并无乐趣。相比之下我们更喜欢使用图灵假设。显然我们可以用某种程序设计语言实现这一点；实际上，9.1节中的习题1提出的那个程序就是一台通用图灵机在较高级的语言上的实现。因此，我们完全可以期望用一台标准图灵机完成上述工作。于是，我们可以断言下述图灵机的存在：对于任意给定的一个程序，它都可以完成此程序规定的计算。这种图灵机就是通用计算机的模型。

每一个图灵机都可以由0和1的符号串表示这一点对我们有重要的启示。但在我们探讨这些启示之前，我们需要先回顾一下集合论的一些结果。

某些集合是有限的，但大多数有趣的集合（和语言）是无限的。对于无限集合，我们将其划分为可数（countable）集和不可数（uncountable）集两类。如果一个集合的元素可以一一映射到正整数集，就称这个集合可数。这也就是说我们可以按照某种顺序写出这个集合的元素，如x_1、x_2、x_3等等，于是此集合的每个元素都有一个有限的序号。例如，我们可以按照0，2，4，…的顺序写出所有偶数。因为每一个正整数$2n$都出现在第$n+1$个位置上，所以偶数集是可数的。这一点并不会令人感到意外，但有一些更复杂的例子却是有悖直觉的。例如所有形如p/q的分数构成的集合，其中p和q是正整数。怎样排列这一集合中的元素才能证明此集合是可数的呢？我们不能写成

267

$$\frac{1}{1},\frac{1}{2},\frac{1}{3},\frac{1}{4},\cdots$$

因为2/3永远不会出现。但这并不意味着此集合是不可数的。对于这个例子，有一种聪明的排列集合元素的方法能够证明此集合实际上是可数的。如图10-17所示的方法，我们可以按照箭头所指的顺序写下集合元素。那就是

$$\frac{1}{1},\frac{1}{2},\frac{2}{1},\frac{1}{3},\frac{2}{2},\cdots$$

在这个序列中，2/3出现在第八个位置上，并且每一个元素在序列中都有相应位置。因此这个

集合是可数的。

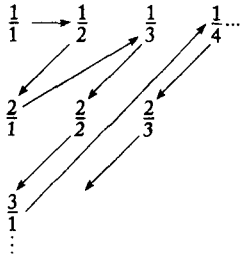


图 10-17

从这个例子可以看出，如果我们能按某种方法顺序地写出集合中的元素，我们就能够证明此集合是可数的。我们将这类方法称作*枚举过程*（enumeration procedure）。因为一个枚举过程是某种机械的过程，所以我们可以用图灵机模型来形式化地定义它。

定义10.3 *令S为字母表Σ上的符号串集合。则S的枚举过程就是一个执行下列计算步骤的图灵机*

$$q_0\square \vdash^* q_s x_1 \# s_1 \vdash^* q_s x_2 \# s_2 \cdots$$

其中$x_i \in \Gamma^* - \{\#\}$，$s_i \in S$。用这种方式，$S$中的每一个符号串$s$都会在有限步内被产生。状态$q_s$用于表示$S$的成员状态。也就是说每当机器进入状态$q_s$时，#后面的符号串一定是$S$中的成员。 268

并非每个集合都是可数的，我们在下一章会看到一些不可数的集合。但任何一个可以被枚举的集合都是可数的，因为枚举给出了集合中所有元素之间的顺序。

严格地讲，一个枚举过程并不能被称为算法，因为当S是无穷集合时，枚举过程是不终止的。但枚举过程依然是有意义的，因为枚举的结果是定义清楚且可预测的。

例10.3 令$\Sigma = \{a, b, c\}$。如果我们能找到按某种顺序（比如字典序）枚举出集合$S = \Sigma^+$中所有元素的枚举过程，那么S就是可数的。但是字典序是不可取的，除非我们对它作一些修改。在字典中，所有以a开头的单词都出现在以b开头的单词之前。但是，当有无穷多个以a开头的单词时，我们就永远无法枚举出以b开头的单词。这违反了定义10.3中规定的任何元素都必须在有限步内枚举出的条件。

然而，我们可以采用修改后的字典序，即把符号串的长度作为考虑的首要因素，其次是相同长度符号串在字典中出现的顺序。下面就是S的一个枚举过程

$$a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, aaa, \cdots$$

由于我们在后面还会用到这一顺序的定义，所以我们将它称为*良序*（proper order）。 □

上述讨论的一个重要结论就是图灵机是可数的。

定理10.3 *所有图灵机构成的无穷集合是可数的。*

证明：我们可以用0和1对每个图灵机进行编码。利用这样的编码，我们就可以构造出如下的枚举过程：

1. 按良序生成$\{0, 1\}^+$中的下一个符号串。

2. 检查生成的符号串，看它是否定义了一个图灵机。如果是，则将此符号串按照定义10.3所要求的形式写在带上；如果否，则忽略此符号串。

3. 转到第1步。

因为每一个图灵机都有一个有限的描述，所以任何一个具体的图灵机最终都能够被这一过程枚举出来。■

269

图灵机集合元素之间的具体顺序取决于我们所用的编码方式。如果采用另外一种编码方式，我们就能获得另一种顺序。然而，采用何种顺序是不重要的，重要的是必须存在某种顺序。

习题

1. 大致给出一个算法，检查一个$\{0, 1\}^+$中的符号串是否是一个图灵机的编码。
2. 用我们前面讲过的编码方式，给具有如下转移函数的图灵机编码

$$\delta(q_1, a_1) = (q_1, a_1, R)$$
$$\delta(q_1, a_2) = (q_3, a_1, L)$$
$$\delta(q_3, a_1) = (q_2, a_2, L)$$

3. 大致给出一个图灵机程序，使其按良序枚举出集合$\{0,1\}^+$中的元素。
4. 在习题3中，0^i1^j位于枚举序列中的第几个？
5. 设计一台图灵机按良序枚举出集合

$$L = \{a^nb^n : n \geqslant 1\}$$

6. 对于例10.3，找到一个函数$f(w)$，该函数能给出符号串每一个w在良序枚举序列中的序号。
7. 证明由所有形如（i, j, k）的正整数三元组构成的集合是可数的。
8. 设S_1和S_2是可数集。证明$S_1 \cup S_2$和$S_1 \times S_2$也是可数集。
9. 证明有限个可数集的笛卡儿积也是可数的。

10.5 线性有界自动机

虽然我们不能通过使图灵机的带结构复杂化而使其功能变强，但我们可以通过限制对带的使用方式而限制图灵机的能力。我们在前面已经见过一个例子——下推自动机。我们可以将下推自动机看作是有一条带的非确定型图灵机，而且这条带必须以栈的方式使用。我们还可以用其他方式限制对带的使用。比如可以将带上的工作空间限制在一个有限的范围内。可以证明，加上这种限制的图灵机就是有限状态自动机（见本节习题3），因此我们不讨论此种限制。但还有一种我们感兴趣的限制：我们要求图灵机只能工作于带的输入部分。因此，较长的输入符号串意味着较多的工作空间。这一限制定义了另一种自动机：线性有界自动机（linear bounded automata, lba）。

270

像标准图灵机一样，线性有界自动机也有一条无限长的带，但带上能够使用的部分的长度是输入部分的函数。一种特殊情况就是，我们将带的可用部分正好限制在输入部分所占用的单元上。为了做到这一点，我们可以将输入部分包含在两个特殊符号——*左端标记*（left-end marker）（[）和*右端标记*（right-end marker）（]）——之间。对于一个输入w，图灵机的初始格局由瞬时描述$q_0[w]$给出。两个端点标记“[”和“]”所在的单元不能被重写，读写头也不能移动到“[”的左边或“]”的右边。有时我们称读写头被端点标记“弹回”。

定义10.4 一个线性有界自动机是一个非确定型图灵机$M=(Q,\Sigma,\Gamma,\delta,q_0,\square,F)$，具有如定义10.2所述的限制，并且字母表$\Sigma$必须包含两个特殊符号“[”和“]”，且$\delta(q_i,[)$只能包含形如$(q_j,[,R)$的元素，$\delta(q_i,])$只能包含形如$(q_j,],L)$的元素。

定义10.5 符号串w被一个线性有界自动机接受，如果对于某个$q_f\in F, x_1, x_2\in\Gamma^*$，存在一个可行的迁移序列

$$q_0[w]\vdash^* [x_1q_fx_2]$$

被lba接受的语言就是所有被接受符号串构成的集合。

注意，在这一定义中，我们假设线性有界自动机是非确定性的。这并不只是出于方便的考虑，而且本质上对于讨论lba是必需的。当然我们也可以定义确定型的lba，但我们还不知道它是否与非确定型的lba等价。本节习题8对这一问题做了一些讨论。 271

例10.4 语言

$$L=\{a^nb^nc^n : n\geqslant 1\}$$

被某个线性有界自动机接受。从例9.8中的讨论，我们可以直接得出这一结论。例9.8中图灵机的计算并不需要原始输入部分以外的空间，因此也可以被线性有界自动机完成。 □

例10.5 找到一个线性有界自动机，使其接受语言

$$L=\{a^{n!} : n\geqslant 0\}$$

解决这一问题的一种方法是依次用2，3，4，…去整除a的个数，直到我们得出是接受还是拒绝这一符号串。如果输入符号串属于L，则最后必会剩下一个a；如果不属于，则某一个整除必会出现一个长度不为0的余数部分。我们简要描述这一解决方法从而揭示定义10.4中暗含的一个策略。线性有界自动机的带可以是多道的，多出来的道可以作为草稿区（scratch space）。对于这一问题，我们可以采用2-道带。第一道包含在整除过程中剩余a的个数；第二道包含当前的除数（图10-18）。具体的解法是十分简单的。我们用第二道上的除数去整除第一道上a的个数，比如我们可以删掉除了位于除数整数倍位置上的a以外的所有a。然后，我们将除数加1，并继续上述过程，直到遇到一个非0的余数或者只剩一个a为止。 □

[	a	a	a	a	a	a	]	要检查的a
[	a	a	a				]	当前的除数

图 10-18

前两个例子表明线性有界自动机比下推自动机的功能更强大，因为两个例子中的语言都不是上下文无关的。为了证实这一猜想，我们还需要证明任何一个上下文无关语言都可以被一个线性有界自动机接受。在后面我们会用一种间接的方法证明这一点。本节的习题5和习题6提出了一种更为直接的方法。要推测图灵机和线性有界自动机之间的关系并非易事。例10.5中的问题无疑可以用线性有界自动机解决，因为我们可以使用与输入内容长度成比例的草稿区。事实上，要想给出一个具体明确的不能被线性有界自动机接受的语言是相当困难的。在第11章我们将会证明线性有界自动机类的能力弱于未受限制的图灵机类，但要给出一个这样的例子仍有许多工作要做。 272

习题

1. 给出例10.5中解决方案的细节。
2. 给出例10.5的一个不需要第二道草稿区的解决方案。●
3. 考虑一个离线图灵机，其输入只能被从左至右地读一次，并且不能被重写。其在带上至多可以使用n个多余的单元作为工作空间，且n对于所有输入都是固定的。证明此类自动机等价于有穷自动机。
4. 给出接受下列语言的线性有界自动机。
   (a) $L = \{a^n : n = m^2, m \geqslant 1\}$
   (b) $L = \{a^n : n$是质数$\}$
   (c) $L = \{a^n : n$不是质数$\}$
   (d) $L = \{ww : w \in \{a, b\}^+\}$
   (e) $L = \{w^n : w \in \{a, b\}^+, n \geqslant 1\}$ ●
   (f) $L = \{www^R : w \in \{a, b\}^+\}$
5. 给出一个识别例10.5中语言的补语言的lba，设$\Sigma = \{a, b\}$。
6. 证明对于每一个上下文无关语言，都存在一个接受它的pda，且栈中符号的个数从不超过输入符号串的长度加1。●
7. 利用上题的结论证明任何一个不包含λ的上下文无关语言都能被某个线性有界自动机接受。
8. 要定义一个确定型的线性有界自动机，我们可以用定义10.4，只是要求图灵机是确定型的。检查你给出的习题4的解决方案，它们都是确定型的线性有界自动机吗？如果不是，请找出确定型的解决方案。

273

第11章 形式语言和自动机的层次结构

现在我们把注意力转回到主要的研究兴趣——形式语言上来。在本章中，我们的直接目标是研究与图灵机相关的形式语言和它们的一些约束条件。由于图灵机具有很强的计算能力，能够完成所有的算法计算，因此我们会发现与它相关联的语言族拥有很多的成员。这其中不仅包括了正规语言和上下文无关语言，还包括了我们掌握的其他各种语言。然而，存在这样一个重要的问题——是否存在某种不能被图灵机所接受的语言。为了回答这个问题，我们首先要证明存在比图灵机所能接受的语言种类更多的语言。也就是要证明必然存在某些不能被图灵机所接受的语言。虽然这个证明方法很简单清晰，但是它对问题并没有进行深入研究讨论，所以对我们理解问题的本质并不是很有帮助。因此，我们将建立具体的不能被图灵机识别的语言，通过这个明确的例子来证明这类语言的存在性。除此之外，另外一条研究这个问题的途径是考察图灵机和一些特定文法之间的关系，然后再在这些文法、正规文法和上下文无关文法之间建立起联系。这样我们可以建立起所有文法的层次结构。通过这个层次结构可以得到划分语言族的标准。借助集合论中的图，我们可以很清楚地解释不同语言之间的关系。 275

严格地讲，在本章中出现的很多论证仅对不包含空符号串的语言才成立。这个限制是因为我们定义图灵机不接受空符号串引起的。为了避免对定义的修改或者增加一个重复的否定性信息（关于空串），我们在这里假定：如果没有特殊说明，本章中所有讨论的语言都不包含λ。在语言包含λ的情况下重新阐述我们给出的每个结论是不难的，把这些留给读者。

11.1 递归语言和递归可枚举语言

首先介绍与图灵机相关语言的术语。我们需要对图灵机所接受的语言和存在一个成员资格判定算法的语言进行严格地区分。这是因为图灵机对于本身不接受的输入不一定总会停机，因此前者并不能包含后者。

定义11.1 *一个语言L，如果存在一个接受它的图灵机，则称这个语言是递归可枚举的*（recursively enumerable）。

这个定义表明存在一个图灵机M，对于所有的$w \in L$，满足

$$q_0 w \vdash_M^* x_1 q_f x_2$$

这里q_f是终态。这个定义没有说明对于任何不在L中的w会出现什么样的情况。因此，机器在这时候可能在非终态上停机，也可能进入了一个无穷的循环而永不停止。所以，我们可以要求图灵机告诉我们，一个给定的输入是否在它的语言之中。 276

定义11.2 *在Σ上的语言L被称为递归的*（recursive），*如果存在一个图灵机M，该图灵机接受语言L并且对Σ^+中的所有w都能停机。换句话说，一个语言是递归的当且仅当这个语言存在一个成员资格判定算法。*

如果一个语言是递归的，那么一定存在一个极易构造的枚举过程。假设M是一个判定递归语言L的成员资格的图灵机。我们可以首先建立另外一个图灵机，比如说$\hat{M}$，$\hat{M}$按照良序来产生Σ^+中的符号串，我们将这些符号串记为$w_1, w_2, w_3, \cdots$。当这些符号串被生成之后，它们成为了图灵机M的输入。而M被修改为仅仅把L中的串写在它的带上。

对于所有的递归可枚举语言都有一个类似的枚举过程这点，并不像看到的那样简单。我们不能按上述的方法那样去做。因为如果某个w_j不在L中，那么当M以w_j作为带上的内容开始时，M可能永不停止。因此就不能枚举L中排列到w_j后面的符号串。为了保证不会出现这种情况，我们用另外一种方式来执行计算。我们首先用$\hat{M}$来生成w_1，并且使用M来对w_1执行一次迁移。然后我们用$\hat{M}$来生成w_2，并且使用M来对w_2执行一次迁移，紧接着使用M对w_1执行第二次迁移。在这之后我们生成w_3，并且依次令M执行w_3的第一次迁移，w_2的第二次迁移以及w_1的第三次迁移，依此类推。具体的执行顺序在图11-1里面进行了解释。从这里，我们可以清楚地看到，M是不会进入无限循环的。既然L中任意的符号串w都能由$\hat{M}$生成并且在有限步骤内被M所接受，那么L中的所有符号串最终都能被M识别。

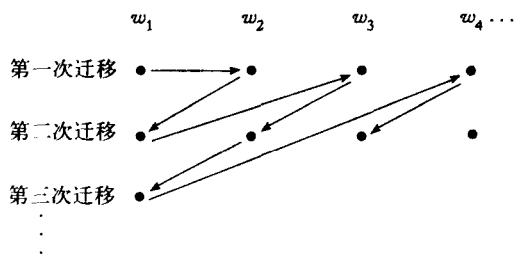


图 11-1

如果一个语言存在一个枚举的步骤，那么它是递归可枚举的。我们可以将给定的输入符号串和由枚举过程连续产生的符号串序列进行简单的比较。如果$w \in L$，我们最终能够得到一个匹配，那么这个过程就可以停止了。

定义11.1和定义11.2并没有告诉我们递归语言或者递归可枚举语言的本质性信息。这些定义只是将与图灵机相关的语言族中的成员和这些名称关联起来，而并没有展示这些语言族成员的典型语言本质。同样，这些定义也没有向我们展示这些语言之间的关系或者这些语言和我们以前遇到的语言之间的联系。因此，我们马上就会遇到类似这样的问题“是否存在语言是递归可枚举的但却不是递归的？”和“是否存在语言可以被清楚地描述但却不是递归可枚举的”。对于这些问题，虽然我们可以给出一些答案，但是我们却没有办法给出具体的例子来解释这些问题，特别是第二个。

277

11.1.1 非递归可枚举的语言

我们可以通过多种不同的方法来表明这种非递归可枚举的语言的存在。其中有一种方法非常简洁，它利用了数学中一个非常基础的结论。

定理11.1 *如果S是一个无穷可数集合，那么它的幂集2^S不是可数的。*

证明：令$S = \{s_1, s_2, s_3, \cdots\}$，那么它的幂集$2^S$中的任意元素$t$都可以被表示为一个0和1组成的序列。当且仅当$s_i$在$t$中出现，相应的第$i$个位置上取1。例如，$\{s_2, s_3, s_6\}$可以被表示为

01100100…，而$\{s_1, s_3, s_5, \cdots\}$则会被表示为10101…。显然，$2^S$中的任意元素都可以被表示为一个序列，同样这样一个序列也唯一地代表了2^S中的一个元素。假设2^S是可数的，那么它的元素可以按照某种顺序写出来，如$t_1, t_2, t_3\cdots$。我们可以把它们写入一个表中，如图11-2所示。在这个表中，将主对角线上的元素替换为它的反，即将主对角线上的0替换为1，1替换为0。在图11-2所示的例子中，对角线上的序列是1100…，因此我们得到0011…。比如对于某个t_i来说，替换之后的结果代表了2^S中的某个元素。但是这个结果不会是t_1，因为它和t_1在元素s_1上不同，同样的道理它不可能是t_2、t_3或者是其他的t_i。这样就造成了逻辑上的矛盾。因此，2^S是可数的假设是错误的。■

t_1	1	0	0	0	0	...
t_2	1	1	0	0	0	...
t_3	1	1	0	1	0	...
t_4	1	1	0	0	1	...
⋮						

图 11-2

这种论证因为包含了对于一个表中的对角线上元素的操作，因此被称为对角线化(diagonalization)。这种方法是由数学家G. F. Cantor提出的，他用这种方法证明了实数集合是不可数的。在接下来的几章中，我们会在几个不同的上下文中看到类似的论证。定理11.1是对角线化最直接的使用。

作为本结论的直接推论，我们可以看到，在某种意义上，图灵机的数目少于语言的数目。因此，必然存在非递归可枚举的语言。

定理11.2 *对于任何非空的集合Σ，都存在非递归可枚举的语言。*

证明：某个语言是Σ^*的一个子集，并且每个子集都是一个语言。因此，所有语言的集合数目是2^{Σ^*}。既然Σ^*是无限的，那么定理11.1告诉我们，Σ上所有语言构成的集合是不可数的。但是所有图灵机的集合是可以被枚举的，因此所有递归可枚举的语言的集合是可数的。通过本节后的习题16，可以证明在Σ上必然存在非递归可枚举的语言。■

这个证明虽然简洁，但是在很多方面却不尽如人意。虽然它告诉我们非递归可枚举的语言的存在，但对我们完全没有建设性的帮助。它并没有告诉我们任何关于这种语言的特征。在下一节中，我们会更加清楚地阐述这个结论。

11.1.2 非递归可枚举语言

既然所有能够被图灵机接受的语言都能用一种直接的算法方式描述，并且都是递归可枚举的，那么对于非递归可枚举语言的描述必然是非直接的。但是，这种描述是可能的。证明的过程使用了对角线化理论的变种。

278～279

定理11.3 *存在某种递归可枚举语言，它的补不是递归可枚举的。*

证明：令$\Sigma = \{a\}$，并且考虑所有以此为输入字母表的图灵机构成的集合。根据定理10.3，这

个集合是可数的，因此，我们可以按照$M_1, M_2 \cdots$的顺序来对这些元素进行排序。对于每个图灵机M_i，都会有一个相关联的递归可枚举语言$L(M_i)$。同样，对于Σ上每个递归可枚举语言，必定存在某个能够接受它的图灵机。

现在我们来定义一种新的语言L。对于每个$i \geqslant 1$，符号串a^i属于L当且仅当$a^i \in L(M_i)$。因为语句$a^i \in L(M_i)$，所以$a^i \in L$，或者为真，或者为假，所以L是良定义（well defined）的。接下来，我们来看L的补，

$$\overline{L} = \{a^i : a^i \notin L(M_i)\} \tag{11-1}$$

同样，L的补也是良定义的，接下来我们将证明L的补是非递归可枚举的。

我们通过反证法来证明L的补是非递归可枚举的。首先假设$\overline{L}$是递归可枚举的。如果这个假设成立，那么必然存在某个图灵机，比如说M_k，满足

$$\overline{L} = L(M_k) \tag{11-2}$$

那么对于符号串a^k，它究竟是在L中还是在$\overline{L}$中呢？假设$a^k \in \overline{L}$，那么由式（11-2）可得$a^k \in L(M_k)$，但是式（11-1）意味着$a^k \notin \overline{L}$。相反，如果我们假设a^k在L中，那么$a^k \notin \overline{L}$，那么式（11-2）意味着$a^k \notin L(M_k)$，但是从式（11-1），我们又可以得到$a^k \in \overline{L}$。通过这个矛盾，我们可以得出假设不成立。因此$\overline{L}$是非递归可枚举的。

为了完成这条定理的证明，我们必须证明L是递归可枚举的。我们可以使用图灵机的枚举过程来证明这一点。给定a^i，我们可以首先通过a的数目来确定i。接下来，可以使用图灵机的枚举过程来查找M_i。最后，我们可以把这个描述和a^i交给一个通用的图灵机M_u，让M_u来模拟a^i在M上的处理过程。如果a^i在L中，那么M_u最终会停机。这样的话，存在一个接受所有$a^i \in L$的图灵机。因此，根据定义11.1，L是递归可枚举的。■

280

这个定理的证明明确表示了一个良定义的、非递归可枚举的语言。虽然解释$\overline{L}$是不容易的，我们可能只能展示其中的一小部分成员。但是，我们毕竟得到了$\overline{L}$的确切定义。

11.1.3 递归可枚举但非递归的语言

接下来，我们会证明有些语言是递归可枚举的，但是却不是递归的。同样，我们这次也要通过一个迂回的方法来证明。我们首先要建立一个辅助性的结论。

定理11.4 *如果一个语言L和它的补$\overline{L}$都是递归可枚举的，那么两者都是递归的。如果L是递归的，那么$\overline{L}$也是递归的，同时两者都是递归可枚举的。*

证明：如果L和它的补$\overline{L}$都是递归可枚举的，那么存在图灵机M和$\hat{M}$来分别作为L和$\overline{L}$的枚举过程。第一个图灵机会产生L中的$w_1, w_2, \cdots$，第二个产生$\overline{L}$中的$\hat{w}_1, \hat{w}_2, \cdots$。现在假设任意给定的$w \in \Sigma^+$，我们首先可以通过$M$来生成$w_1$并和$w$进行比较。如果它们不一样，我们可以通过$\hat{M}$来生成$\hat{w}_1$并和$w$再进行比较。如果仍不相同，我们可以通过$M$来生成$w_2$并和$w$进行比较，$\hat{M}$来产生$\hat{w}_2$，依此类推。任意的$w \in \Sigma^+$都会被$M$或者$\hat{M}$生成，这样我们最终会得到一个匹配。如果匹配的符号串是由M产生的，那么w属于L，否则w在$\overline{L}$中。这个处理过程是L和$\overline{L}$的成员资格判定算法，因此它们都是递归的。

相反，如果我们假设L是递归的。那么对它必然存在一个成员资格判定算法。我们可以通

过对这个算法的结论求补来使得它成为L补的成员资格判定算法。因此$\bar{L}$也是递归的。既然任意的递归语言都是递归可枚举的，因此结论成立。■

从这里，我们可以直接得出递归可枚举语言族与递归语言族是不等同的。定理11.3中的语言L属于第一个族，但是不属于第二个族。

定理11.5 *存在一个递归可枚举语言L但却不是递归的，换句话说，递归语言族是递归可枚举语言族的真子集。*

证明：定理11.3中的语言L是递归可枚举的，但它的补不是。因此，根据定理11.4，它不是递归的，这个例子就可以证明结论。■ 281

从这里，我们还可以得出，确实存在无法为其构造成员资格判定算法的良定义的语言。

习题

1. 证明实数集合是不可数的。
2. 证明所有非递归可枚举语言组成的集合是不可数的。●
3. 设L是有限语言。证明L^+是递归可枚举的。提示建立L^+的枚举过程。
4. 设L是上下文无关语言，证明L^+是递归可枚举的。提示建立L^+的枚举过程。
5. 证明一个非递归可枚举语言L的补不是递归的。
6. 证明递归可枚举语言族在并运算下是封闭的。●
7. 递归可枚举语言族在交运算下是否是封闭的？
8. 证明递归语言族在并运算和交运算下是封闭的。
9. 证明递归语言族和递归可枚举语言族在逆运算下是封闭的。
10. 递归语言族在连接运算下是封闭的吗？
11. 证明上下文无关语言的补必然是递归的。●
12. 设L_1是递归语言、L_2是递归可枚举语言。证明L_2-L_1一定是递归可枚举的。
13. 假设存在某个图灵机能够按照适当的顺序枚举出L中的元素，证明这意味着L是递归的。
14. 设L是递归语言，是否L^+也是递归的？●
15. 选择一个特定编码方式的图灵机，并利用它来产生定理11.3中语言$\bar{L}$的元素。
16. 设S_1是可数集合，S_2是不可数集合，并且$S_1 \subset S_2$。证明S_2中必然包含无数个不属于S_1的元素。
17. 证明在习题16中产生的集合S_2-S_1不可能是可数的。
18. 为什么定理11.1的证明当S是无穷时不能成立？●
19. 证明所有无理数组成的集合是不可数的。 282

11.2 无限制文法

为了考察递归可枚举的语言和文法之间的联系，我们再回过头来看看第1章中对于文法的一般性定义。在定义1.1中，产生式规则可以采用任意形式，但是后来我们看到的特定文法都被加上了各种各样的限制。如果采用一般性的方式并且不强加任何的限制，我们会得到无限制文法。

定义11.3 文法$G = (V, T, S, P)$被称为无限制文法（unrestricted grammar）当且仅当它的每个产生式都有形式：

$$u \to v$$

这里u属于$(V \cup T)^+$，而v属于$(V \cup T)^*$。

在一个无限制文法中，基本上没有对产生式强加任何条件。任意数目的变量和终结符都可以出现在左端或者右端，并且可以以任意顺序出现。这里唯一的一个限制就是λ不能作为一个产生式的左部。

正如我们所见，无限制文法比我们目前所研究的受限制文法，如上下文无关文法和正则文法具有更强的表达能力。事实上，无限制文法对应了我们使用机械方法所能识别的最大语言族。即无限制文法生成了递归可枚举的语言族。我们将通过两个部分来证明这一点；第一部分相当直接，但是第二部分则需要一个较长的构造过程。

定理11.6 任何由无限制文法生成的语言都是递归可枚举的。

证明：文法实际上定义了如何系统地枚举相应语言中所有符号串的过程。例如，我们可以将L中所有满足下面推导的w列出来

$$S \Rightarrow w$$

283 即w是一步推出的。既然文法产生式的集合是有限的，那么这种符号的数目就是有限的。接着，我们可以列举出所有在L中w，并且由两步推导出来的符号串

$$S \Rightarrow x \Rightarrow w$$

依此类推。我们可以在图灵机上模拟这种推导过程。因此，我们存在一个这种语言的枚举过程，所以它是递归可枚举的。■

无限制文法和递归可枚举的语言之间的这种对应关系是很自然的。文法可以通过一个良定义的算法步骤生成符号串，因此，这个推导过程可以在图灵机上实现。为了显示其反面，我们描述如何用无限制文法模拟任给的图灵机。

给定一个图灵机$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$，我们来构造一个无限制文法$G$，满足$L(G) = L(M)$。构造的思想相对比较简单，但是具体实现却比较麻烦。

既然图灵机的计算过程可由如下瞬时描述序列表示

$$q_0 w \overset{*}{\vdash} x q_f y \tag{11-3}$$

我们可以排列它使得当且仅当式（11-3）成立时，相应的文法具有如下特性：

$$q_0 w \overset{*}{\Rightarrow} x q_f y \tag{11-4}$$

接下来，对于所有满足条件式（11-3）的w，我们需要在式（11-4）和我们需要的形式

$$S \overset{*}{\Rightarrow} w$$

之间建立关联。为了完成这个关联的建立，我们定义一个文法，它大体上有如下特性：

1. 对于所有的$w \in \Sigma^+$，S都可以推导出$q_0 w$。

2. 当且仅当式（11-3）成立时，才可能有式（11-4）。

3. 当符号串xq_fy ($q_f \in F$) 生成时，文法将符号串转换为最初的w。 284

完整的推导过程如下:

$$S \overset{*}{\Rightarrow} q_0 w \overset{*}{\Rightarrow} x q_f y \overset{*}{\Rightarrow} w \tag{11-5}$$

上述推导过程中的第三步比较麻烦。问题在于，如果w在第二步里被修改了，那么文法如何保存w？我们通过对符号串进行编码来解决这个问题。在开始时，让系统拥有w的两份拷贝。第一个被保存，第二个在式（11-4）中使用。当进入最终格局时，文法将清除w之外的所有符号。

为了产生w的两份拷贝，并且为了处理M的状态符号（最终会被文法清除），对于所有的$a \in \Sigma \cup \{\square\}$, $b \in \Gamma$以及所有满足$q_i \in Q$的i，我们引入变量V_{ab}和V_{aib}。变量V_{ab}对符号a和b编码，而V_{aib}则同时对a、b和状态q_i进行编码。

式（11-5）中的第一步可以这样得到:

$$S \rightarrow V_{\square\square} S | S V_{\square\square} | T \tag{11-6}$$

$$T \rightarrow T V_{aa} | V_{a0a} \tag{11-7}$$

其中$a \in \Sigma$。这种推导允许文法对于任意的符号串$q_0 w$产生编码的版本。这里的$q_0 w$前面和后面都可以有任意多的空格符。

对于第二步，对每个关于M的转换

$$\delta(q_i, c) = (q_j, d, R)$$

我们都添加文法产生式

$$V_{aic} V_{pq} \rightarrow V_{ad} V_{pjq} \tag{11-8}$$

其中所有的a，$p \in \Sigma \cup \{\square\}$、$q \in \Gamma$。对于$M$的每个

$$\delta(q_i, c) = (q_j, d, L)$$

我们都在G中添加

$$V_{pq} V_{aic} \rightarrow V_{pjq} V_{ad} \tag{11-9}$$

其中所有的a，$p \in \Sigma \cup \{\square\}$，$q \in \Gamma$。

如果在第二步中，M就进入了一个终态，则文法必须清除w之外的所有符号，w保存在V中第一个索引中。因此，对于每一个$q_j \in F$，$a \in \Sigma \cup \{\square\}$，$b \in \Gamma$，我们引入产生式

$$V_{ajb} \rightarrow a \tag{11-10}$$ 285

这样就得到了符号串中的第一个终结符，对于所有的a，$c \in \Sigma \cup \{\square\}$，$b \in \Gamma$，上式可以重写为

$$c V_{ab} \rightarrow ca \tag{11-11}$$

$$V_{ab} c \rightarrow ac \tag{11-12}$$

我们还需要另外一个特殊的产生式

$$\square \rightarrow \lambda \tag{11-13}$$

最后一个产生式处理了当M移出输入w所占的带区域的情况。为了在这种情况下能够正常工作，我们首先使用式（11-6）和式（11-7）产生

$$\square \cdots \square q_0 w \square \cdots \square$$

来代表所有使用过的带区域。其他无关的空格符都由式（11-13）除去。

接下来的例子揭示了这个复杂的构造过程。仔细检查例子的每一步，看看各产生式做些什么，为什么需要它们。

例11.1 对于图灵机$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$，令

$$Q = \{q_0, q_1\}$$
$$\Gamma = \{a, b, \square\}$$
$$\Sigma = \{a, b\}$$
$$F = \{q_1\}$$
$$\delta(q_0, a) = (q_0, a, R)$$
$$\delta(q_0, \square) = (q_1, \square, L)$$

该图灵机接受$L(aa^*)$。

现在考虑接受符号串aa的计算

$$q_0 aa \vdash a q_0 a \vdash aa q_0 \square \vdash a q_1 a \square \tag{11-14}$$

为了能够用G推导出这个符号串，我们首先使用式（11-6）和式（11-7）中的规则来得到相应的初始符号串。

$$S \Rightarrow S V_{\square\square} \Rightarrow T V_{\square\square} \Rightarrow T V_{aa} V_{\square\square} \Rightarrow V_{a0a} V_{aa} V_{\square\square}$$

286 最后一个句型用于模拟图灵机上计算的推导部分，是推导的出发点。它在第一个索引中包含了原始输入$aa\square$，在其余的索引中包含初始瞬时描述$q_0 aa\square$。接下来，我们使用

$$V_{a0a} V_{aa} \rightarrow V_{aa} V_{a0a}$$
$$V_{a0a} V_{\square\square} \rightarrow V_{aa} V_{\square 0 \square} \text{（式（11-8）的特例）}$$

以及

$$V_{aa} V_{\square 0 \square} \rightarrow V_{a1a} V_{\square\square} \text{（来自式（11-9））}$$

由上述三个产生式，我们得到如下推导步骤

$$V_{a0a} V_{aa} V_{\square\square} \Rightarrow V_{aa} V_{a0a} V_{\square\square} \Rightarrow V_{aa} V_{aa} V_{\square 0 \square} \Rightarrow V_{aa} V_{a1a} V_{\square\square}$$

第一个索引序列保持不变，始终记录着原始的输入。其他索引的序列为

$$0aa\square, a0a\square, aa0\square, a1a\square$$

它和式（11-14）中的瞬时描述序列等价。

最后，将式（11-10）至式（11-13）用在最后一步中，可得如下推导

$$V_{aa}V_{a1a}V_{\square\square}\Rightarrow V_{aa}aV_{\square\square}\Rightarrow V_{aa}a\square\Rightarrow aa\square\Rightarrow aa$$

式（11-6）至式（11-13）中描述的构造是证明下面结论的基础。 □

定理11.7 对于每一个递归可枚举语言L，存在一个无限制文法G，满足$L=L(G)$。

证明：上述构造保证了如果$x\vdash y$，那么$e(x)\Rightarrow e(y)$，这里$e(x)$代表了一个符号串根据给定的规定得到的编码。根据对转换步数的归纳，我们可以得到

$$e(q_0w)\overset{*}{\Rightarrow}e(y)$$

287

当且仅当$q_0w\overset{*}{\vdash}y$成立。我们还必须证明，我们可以生成每个可能的初始格局，并且当且仅当M进入一个终止格局时重构w。具体的证明过程不是很难，留做习题。■

这两个定理表明：无限制文法相应的语言族与递归可枚举的语言族是等同的。

习题

1. 下面的无限制文法可推导出什么样的语言？

$$S\rightarrow S_1B$$
$$S_1\rightarrow aS_1b$$
$$bB\rightarrow bbbB$$
$$aS_1b\rightarrow aa$$
$$B\rightarrow\lambda$$

2. 在无限制文法中，如果我们允许空串作为产生式的左部，那么我们会遇到什么样的困难？
3. 如果对文法进行修改，使得推导的开始点是一个符号串的有限集合，而不是一个变量。把这个概念形式化，然后研究如何将这种文法和我们这里用过的无限制文法联系起来。
4. 证明例11.1中的文法不能生成含有b的句子。
5. 给出定理11.7的详细证明。
6. 为语言$L(01(01)^*)$构造图灵机，然后根据定理11.7找出相应的无限制文法，并利用该文法给出0101的推导过程。
7. 证明对于每个无限制文法都存在一个等价的无限制文法，其所有的产生式都有如下的形式

$$u\rightarrow v$$

这里$u,v\in(V\cup T)^+$，并且$|u|\leqslant|v|$，或

$$A\rightarrow\lambda$$

这里$A\in V$。

288

8. 证明如果对习题7中的文法增加更多的条件$|u|\leqslant 2$和$|v|\leqslant 2$，那么结论仍然成立。
9. 对于无限制文法，有些学者给出了与定义11.3截然不同的定义。在他们的定义中，对于无限制文法的产生式都要有如下形式

$$x\rightarrow y$$

这里$x\in(V\cup T)^*V(V\cup T)^*$且$y\in(V\cup T)^*$。区别在于这里的产生式左部至少要有一个变量。

证明这种定义和我们所使用的定义在下面意义上基本是相同的：对一种类型的每个文法都存在等价的另一种类型的文法。

11.3 上下文相关文法和语言

在限制的、上下文无关文法和一般的、无限制文法之间，存在大量的在一定程度上受限制的文法。不是所有的情况都令人感兴趣，在这其中，上下文相关文法值得我们关注。这些文法生成的语言和受限的图灵机类相关联，即我们在10.5节中介绍的线性有界自动机。

定义11.4 文法$G=(V,T,S,P)$是上下文相关的（context-sensitive），如果它的所有产生式都有如下形式

$$x\rightarrow y$$

这里$x,y\in(V\cup T)^+$，并且

289

$$|x|\leqslant|y| \tag{11-15}$$

这个定义清楚地表明了这类文法的一个特征：在推导过程中，后继句型的长度是不会减小的，在这个意义上，上下文相关文法是不会收缩的（noncontracting）。虽然为什么称这种文法为上下文相关文法不是很明显，但是可以看到这种文法可以用一种范式（如Salomaa 1973）来改写，其中所有的产生式都有如下形式

$$xAy\rightarrow xvy$$

这也是说产生式$A\rightarrow v$仅当在A的左边为x，右边为y的这种上下文情况下才能被使用。

11.3.1 上下文相关语言和线性有界自动机

顾名思义，上下文相关文法和同名的语言族是联系在一起的。

定义11.5 语言L是上下文相关语言当且仅当存在一个上下文相关文法G，满足$L=L(G)$或者$L=L(G)\cup\{\lambda\}$。

在定义中，我们又引入了空串。定义11.4暗示了不允许有$x\rightarrow\lambda$，因此一个上下文相关文法是不会生成包含有空串的语言的。而每个不包含λ的上下文无关语言都可以由一个上下文相关文法的特例，比如乔姆斯基范式或格里巴克范式（这两个范式都符合定义11.4的条件）来生成。通过在上下文相关语言（而不是在文法中）中引入空串，我们可以说上下文无关语言的集合是上下文相关语言集合的一个子集。

290

例11.2 $L=\{a^nb^nc^n: n\geqslant 1\}$是上下文相关语言。我们通过为它构造一个上下文相关文法来表明这点。具体文法如下：

$$S\rightarrow abc|aAbc$$
$$Ab\rightarrow bA$$
$$Ac\rightarrow Bbcc$$
$$bB\rightarrow Bb$$
$$aB\rightarrow aa|aaA$$

我们通过展示$a^3b^3c^3$的推导过程来看看该文法是如何起作用的。

$$
\begin{aligned}
&S \Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc \\
&\Rightarrow aBbbcc \Rightarrow aaAbbcc \Rightarrow aabAbcc \\
&\Rightarrow aabbAcc \Rightarrow aabbBbccc \\
&\Rightarrow aabBbbccc \Rightarrow aaBbbbccc \\
&\Rightarrow aaabbbccc
\end{aligned}
$$

这个解决方案有效地使用了变量A和B作为消息的传递者。每当左边产生一个A，它就一直向右移动直到遇到第一个c，此时又产生了另外的b和c。接下来它将B送回到左边以便产生相应的a。整个过程与接受语言L的图灵机的工作过程非常类似。 □

因为上面例子中的语言不是上下文无关的，所以我们可以看到上下文无关语言族是上下文相关语言族的一个真子集。例11.2同样也表明了：即使是为一个相对简单的例子构造相应的上下文相关文法也不是很容易的事。通常，最容易的解决办法是首先在图灵机上获得程序，然后再找到一个等价的文法。一些例子表明对于上下文相关语言，相应的图灵机有可预测空间的要求。实际上，它可以被看作是一种线性有界自动机。

定理11.8 *对于每个不包括λ的上下文相关语言L，都存在某个线性有界自动机M，满足L = L(M)。*

证明：如果L是上下文相关的，那么对于$L-\{\lambda\}$存在一个上下文相关文法。我们将说明如何将文法中的推导过程在线性有界自动机上模拟出来。线性有界自动机有两个栈，一个包含了输入符号串w，另外一个包含了用G推导出来的句型。证明的关键点在于所有可能的句型长度都不会超过$|w|$。另外一个值得注意的是这个线性有界自动机是非确定型的。在证明中，这一点是必需的，这样，我们就可以说正确的产生式总能被猜测出来而不必去追踪无效的产生式选择。因此，定理11.6中描述的计算过程可以被执行。而且这种执行过程，除了最初w占据的空间之外，不需要额外的空间。即它可以由线性有界自动机来执行。■ 291

定理11.9 *如果语言L被某个线性有界自动机M接受，那么就存在某个能够产生L的上下文相关文法。*

证明：文法的构造过程和定理11.7中的过程类似。定理11.7中生成的所有产生式里，除了式（11-13）□→λ以外，都是不会收缩的。但是式（11-13）这个产生式可以被忽略掉。这个产生式仅当图灵机读写头移出了原始输入的边界时才会使用到，但在这里不会出现这种情况。这样，由不含非必需产生式的构造过程得到的产生式都是不会收缩的，证明完成。■

11.3.2 递归语言和上下文相关语言的关系

定理11.9告诉我们每个上下文相关语言都能被某个图灵机所接受，因此是递归可枚举的。由此极易得到定理11.10。

定理11.10 *每个上下文相关语言L都是递归的。*

证明：给定上下文相关语言L和相关的上下文相关文法G，我们来看一下w的推导过程

$$S \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \cdots \Rightarrow x_n \Rightarrow w$$

不失一般性地，我们可以假设在一单个的推导过程中，所有的句型都是不同的；即对于所有

的$i \neq j$，有$x_i \neq x_j$。证明的难点在于要证明任意推导过程的步数都是$|w|$的一个有界函数。我们知道G是不会收缩的，所以

292

$$|x_j| \leq |x_{j+1}|$$

此外，我们唯一要做的就是根据G和w来找到某个m，使得对于所有的j，都满足

$$|x_j| < |x_{j+m}|$$

其中$m = m(|w|)$是$|V \cup T|$和$|w|$的有界函数。由于$|V \cup T|$的有界性意味着只有有限个给定长度的符号串，所以这样的m是存在的。因此$w \in L$的推导长度是不会超过$|w|m(|w|)$ 的。

通过上面的步骤，我们可以很快给出L的一个成员资格判定算法。我们检查所有在长度上界$|w|m(|w|)$ 之内的推导。因为G的产生式是有限的，所以这些步骤也是有限的。如果在这些步骤里面能够产生w，那么$w \in L$，否则w就不在L中。■

定理11.11 *存在不是上下文相关的递归语言。*

证明：考虑$T = \{a,b\}$上的所有上下文相关文法。我们用常规的方式来表示文法，每个文法都有一个形如

$$V = \{V_0, V_1, V_2, \cdots\}$$

的变量集合。每个上下文相关文法都由它们的产生式来说明。我们可以把这些产生式写成一个单独的符号串

$$x_1 \rightarrow y_1;\ x_2 \rightarrow y_2;\ \cdots;\ x_m \rightarrow y_m$$

用下面的同态关系作用于该符号串

$$\begin{aligned} h(a) &= 010 \\ h(b) &= 01^20 \\ h(\rightarrow) &= 01^30 \\ h(;) &= 01^40 \\ h(V_i) &= 01^{i+5}0 \end{aligned}$$

这样，任何上下文相关文法都可以被唯一地表示为$L((011^*0)^*)$ 中的符号串。不仅如此，这种转换是可逆的，即对于$L((011^*0)^*)$ 中的任意符号串，至多有一个上下文相关文法与其对应。

让我们引入$\{0, 1\}^+$上的适当排序，以便我们可以将符号串按顺序写成$w_1, w_2, \cdots$。对于一个给定的符号串w_j可能没有上下文相关文法与之对应，如果有，我们将它称为G_j。接下来，我们定义语言$L = \{w_i: w_i$定义了上下文相关文法G_i并且$w_i \notin L(G_i)\}$。L是良定义的，并且实际上是递归的。为了证明这一点，我们可以为它构造一个成员资格判定算法。对给定的符号串w_i，我们检查它是否定义了一个上下文相关文法G_i，如果没有，那么$w_i \notin L$。如果该符号串定义了一个上下文相关文法，那么$L(G_i)$ 是递归的，我们可以利用定理11.10中的成员资格判定算法查明是否有$w_i \in L(G_i)$。如果没有，则w_i在L中。

293

但是L不是上下文相关的。如果它是上下文相关的，那么会存在某个G_j，满足$L = L(G_j)$。那么我们要问，w_j是否在$L(G_j)$ 中。如果假设$w_j \in L(G_j)$，那么根据定义可知w_j不在L中。但是$L = L(G_j)$，因此出现了矛盾。相反，如果我们假设$w_j \notin L(G_j)$，那么根据定义可知$w_j \in L(G_j)$，又

会出现另一个矛盾。所以L不是上下文相关的。■

定理11.11的结论表明了线性有界自动机确实不如图灵机的能力强，因为它们接受的只是递归语言的真子集。同时，我们也可以看到线性有界自动机确实比下推自动机的能力强。上下文无关文法产生的上下文无关语言是上下文相关语言的子集；而且正如例子所示，是真子集。由于线性有界自动机和上下文相关语言在本质上是等价的，且下推自动机和上下文无关语言本质上也是等价的，所以我们可以看到：所有下推自动机接受的语言同样也能被线性有界自动机所接受。但是，有些语言能够被线性有界自动机所接受却不能被下推自动机接受。

习题

★1. 为下列语言构造上下文相关文法：

(a) $L = \{a^{n+1}b^nc^{n-1} : n \geqslant 1\}$

(b) $L = \{a^nb^na^{2n} : n \geqslant 1\}$

(c) $L = \{a^nb^mc^nd^m : n \geqslant 1, m \geqslant 1\}$ ●

(d) $L = \{ww : w \in \{a, b\}^+\}$

★2. 为下列语言构造上下文相关文法：

(a) $L = \{w : n_a(w) = n_b(w) = n_c(w)\}$

(b) $L = \{w : n_a(w) = n_b(w) < n_c(w)\}$

3. 证明上下文相关语言族对于并运算是封闭的。

4. 证明上下文相关语言族对于逆运算是封闭的。● 294

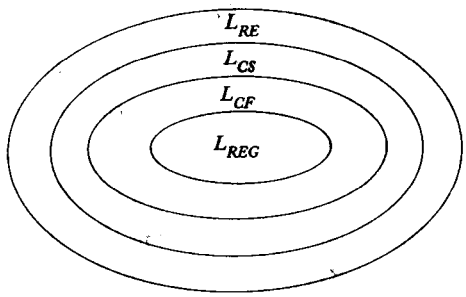
5. 对于定理11.10中的m，给出以$|w|$和$|V \cup T|$函数形式表述的关于m的上界。

6. 证明：对于$L = \{wuw : w, u \in \{a, b\}^+\}$存在相应的上下文无关文法（不需要给出严格的构造过程）。●

11.4 乔姆斯基层次结构

我们现在已经见过了很多种语言族，其中包括递归可枚举的语言族（L_{RE}）、上下文相关语言族（L_{CS}）、上下文无关语言族（L_{CF}）和正则语言族（L_{REG}）。它们之间的关系可以采用乔姆斯基层次结构（Chomsky hierarchy）来表示。Noam Chomsky，形式语言理论的奠基人，给出了形式语言最初的分类。他把这些语言分成4种类型，从0型到3型。人们还在使用最初的术语，用0型到3型来命名我们所研究的各种语言。0型语言是由无限制文法生成的，即递归可枚举的语言。1型语言由上下文相关语言构成，2型语言由上下文无关语言构成，3型语言由正则语言构成。正如我们看到的，每个i型语言族是$i-1$型语言族的真子集。图示（图11-3）可以明确地展示它们之间的关系。图11-3展示了最初的乔姆斯基层次结构，在我们学过的语言族中，还有几种语言族适合于分配到这个图中。包括确定型上下文无关语言族（L_{DCF}）和递归语言族（L_{REC}），将这两个语言族加入到最初的层次结构中进行扩充，得到的结果如图11-4所示。

可以定义其他的语言族并且研究它们在图11-4中的位置，虽然它们之间不一定具有图11-3和图11-4中所示的那种嵌套关系。在一些实例中，它们之间的关系到现在还不是很明确。



295

图 11-3

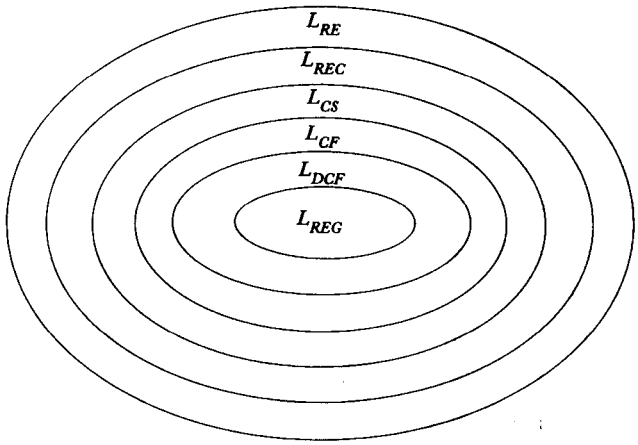


图 11-4

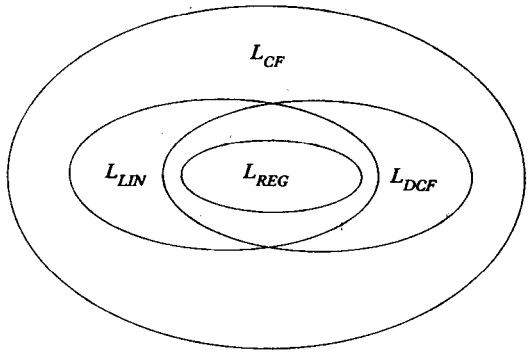
例11.3 我们以前介绍过上下文无关语言

$$L = \{w : n_a(w) = n_b(w)\}$$

并且证明了它是确定型的，但不是线性的。另一方面，语言

$$L = \{a^n b^n\} \cup \{a^n b^{2n}\}$$

是线性的，但却不是确定型的。这表明了正则语言、线性语言、确定型上下文无关语言和非确定型上下文无关语言之间具有如图11-5所示的关系。 □



296

图 11-5

仍然还有未解决的问题。我们在10.5节中的习题8中引入了确定型线性有界自动机的概念。我们自然要提出与其他自动机类似的问题：非确定型在这里起什么作用？不幸的是，还没有一个简单的答案。到目前为止，由确定型线性有界自动机所接受的语言族是否是上下文相关语言族的真子集还是未知的。

总结：我们现在已经知道了几个语言族及与它们相关的自动机之间的关系。我们建立了语言层次结构，并且把自动机按照它们的接受能力进行了分类。图灵机具有比线性有界自动机更强的能力。而这两者都具有比下推自动机更强的能力。在这个层次结构中，处于最底层的是有穷接受器，它是我们研究的出发点。

习题

1. 收集本书中能够说明图11-4中所示的子集关系是正确的例子。
2. 找出两个这样的例子（不包含例11.3中的语言）：语言是线性的，但却不是确定型上下文无关的。
3. 找出两个这样的例子（不包含例11.3中的语言）：语言是确定型上下文无关的，但却不是线性的。

297

第12章 算法计算的限制

我们已经讨论过图灵机能够完成什么样的工作，现在我们来看图灵机所不能完成的工作。虽然图灵论题让我们相信图灵机的计算能力几乎没有限制，但是我们仍然在不同的场合提到过，对于一些特定的问题不存在相应的解决算法。现在我们把这个问题更加明确地表述出来。其中的一些结论可以表述得非常简单；如果一个语言是非递归的，那么根据定义可知不存在与该语言相应的成员判定算法。但这还不是问题的全部，因为非递归语言几乎没有什么实际价值。而问题将进一步深入。例如，我们已经提出（还没有证明），不存在能判定上下文无关文法是否是无二义性的算法。这个问题对我们进行程序设计语言的研究还是有着很重要的实际意义的。

我们首先定义*可判定性*（decidability）和*可计算性*（computability）的概念，当我们说图灵机所不能完成某个事情时，我们通过这些概念来明确说明我们的意思。然后，我们来看几个这种类型的经典问题，这其中包括著名的图灵机停机问题。在图灵机停机问题之后是一系列与图灵机和递归可枚举语言相关的问题。在这之后，我们来看一些关于上下文无关语言的问题。不幸的是，这里我们会发现一些没有相应算法的重要问题。 299

12.1 图灵机所不能解决的问题

机械计算能力是有限的——这一论断并不令人吃惊。直观上看，我们知道很多模糊玄妙的问题需要特别的洞察力和良好的推理能力，而这些已经超出了我们目前能够构造的任何计算机、甚至是可以预见的将来计算机的能力。对于计算机科学家来说，更有趣的是，确实有一些问题可以清晰、简单地陈述，而且很可能有相应的解决算法，但是却被认为是任何计算机都不能解决的。

12.1.1 可计算性和可判定性

在定义9.4中，我们说明了对于一个函数f，如果存在一个图灵机，使得f在某个定义域上的所有的值都能够被计算出来，那么就称函数f在这个定义域上是可计算的。如果不存在这样的图灵机，那么则称函数f是不可计算的。如果一个图灵机只能够计算定义域上的部分函数值，仍然不能称f是可计算的。从这里可以看出，若我们想将一个函数归类为可计算的或不可计算的，就必须清楚函数所在的定义域是什么。

我们在这里关心的是能不能给出一个简单的判定，来判断一个计算的结果是“是”还是“否”。这样，我们可以说一个问题是*可判定的*（decidable）还是*不可判定的*（undecidable）。通过一个问题，我们将了解到一系列相关的语句，其中每一个语句或者是真或者是假。例如，我们考虑“对于上下文无关的文法G，它的语言$L(G)$是二义性的”这句话。对于某些G，这句话是正确的，但对于另外一些文法，它就是错误的。然而，很明显，对于这句话的判断必须两者取一。问题是要判定对于任意给定的G，语句是否成立。在这里，同样有一个潜在的定义

域，即所有上下文无关文法的集合。对于一个问题，如果存在一个图灵机能够对问题的定义域中的每个取值都能够给出正确的答案，我们则称这个问题是可判定的。

当我们下可判定或不可判定的结论时，我们必须知道问题的定义域，因为定义域直接影响结果。问题可能在某个定义域上是可判定的，但是在另外一个定义域上却不是可判定的。具体到问题的一个实例，总是可以判定的，因此结果要么是“真”，要么是“假”。在第一种情况中，总是回答“真”的图灵机，给出的答案是正确的；而在第二种情况中，总是回答“假”的图灵机是适当的。这看起来是个滑稽的回答，但是它强调了重点。我们不知道正确的回答是什么这一事实，不影响图灵机能给出正确响应的本质。

300

12.1.2 图灵机停机问题

我们首先从一些具有历史意义的问题开始，同时以这些问题作为开发后面结论的起点，其中最著名问题是图灵机停机问题（halting problem）。简单地说，问题是这样的：给定一个图灵机M的描述和输入w；当图灵机M从初始格局q_0w出发执行一个计算时，最终会停机吗？用简略的形式表达，当M作用到w上，或者说（M，w）是停机还是不停机。问题的定义域是所有图灵机和所有w的集合，即我们要找到一个图灵机，对给定的任意M和w的描述，它都可以预测用于w的M计算是否会停机。

我们不能通过模拟M在w上的处理过程，即用通用图灵机执行它来得到答案，因为没有计算长度的限制。如果M进入了无穷循环，那么不管我们等待多长时间，我们都不能确定M是否在循环中。这可能只是一个非常长的计算。因此，我们需要的是某个算法。这个算法能够根据图灵机M的描述和输入w来分析确定正确答案。但是，正如我们所表明的，这样的算法并不存在。

为了后续讨论，对停机问题的含义有个准确的概念是方便的。为此，我们对上面所述的内容给出一个具体的定义。

定义12.1 令w_M是描述图灵机$M=(Q,\Sigma,\Gamma,\delta,q_0,\square,F)$的符号串，令$w$是$M$字母表中的一个符号串。我们假设将$w_M$和$w$编码为0和1的符号串，如同10.4节中所用的方法。将停机问题的解决方案表示为一个图灵机H，对任意的w_M和w，如果M作用到w上会停机，则图灵机H执行计算

$$q_0w_Mw \vdash^* x_1q_yx_2$$

如果M作用到w上不会停机，图灵机H执行计算

$$q_0w_Mw \vdash^* y_1q_ny_2$$

301 这里q_y和q_n都是H的终态。

定理12.1 不存在满足定义12.1的图灵机H。因此，图灵机的停机问题是不可判定的。

证明：我们使用反证法，即假设存在一个算法，相应地存在某个能够解决停机问题的图灵机H。H的输入将为符号串w_Mw，要求图灵机H对给定的任意w_Mw，不管回答为“是”，还是回答为“否”，都停机。我们通过H停止时候的终态，比如说是q_y还是q_n，来判定得到的回答内容。这种情形可以用块图来描述，如图12-1所示。图示表明：如果M的初态是q_0，输入是w_Mw，那么它最终会停在状态q_y或q_n上。正如定义12.1中所要求的，我们希望H具有这样的行为：如果当

M作用于w时停机，那么

$$q_0 w_M w \vdash^*_H x_1 q_y x_2$$

如果M作用于w时不停机，则

$$q_0 w_M w \vdash^*_H y_1 q_n y_2$$

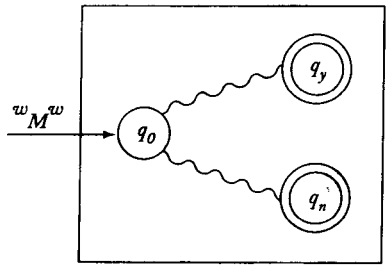


图 12-1

接下来，我们修改H的行为以便得到另一个图灵机H'。H'的结构如图12-2所示。通过在图12-2中新增加的状态，我们要表达的是：状态q_y与新状态q_a和q_b，三者之间存在不依赖带符号的转换，并且带保持不变。这种方法是很简单的。通过对比H和H'，我们可以发现，如果H到达了状态q_y并且停机，那么在H'上它会进入一个无穷循环。形式化地，H'的行为可以表示为：如果M作用于w上时停机，那么

$$q_0 w_M w \vdash^*_{H'} \infty$$

如果M作用于w上时不停机，则

$$q_0 w_M w \vdash^*_{H'} y_1 q_n y_2$$

302

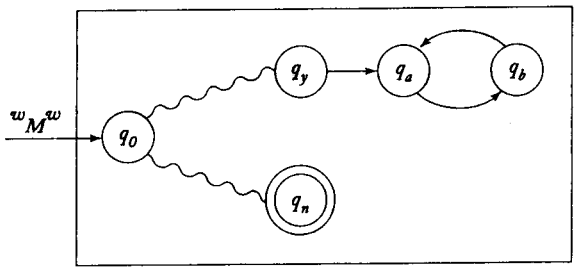


图 12-2

我们从H'出发构造另外一个图灵机$\hat{H}$。这个新的图灵机用w_M作为输入，并把输入复制一份，终止于初态q_0，之后，执行和H'相同的动作。那么对于$\hat{H}$的行为可以表示为：如果M作用到w_M上时停机，那么

$$q_0 w_M \vdash^*_{\hat{H}} q_0 w_M w_M \vdash^*_{\hat{H}} \infty$$

如果M作用到w_M上时不停机，则

$$q_0 w_M \vdash^*_{\hat{H}} q_0 w_M w_M \vdash^*_{\hat{H}} y_1 q_n y_2$$

既然$\hat{H}$是图灵机，就可以用$\{0, 1\}^*$中的一个符号串来描述，比如说$\hat{w}$。这个符号串除了能够作为$\hat{H}$的描述，还可以作为输入符号串。此时，我们理所当然地会问：如果将$\hat{H}$作用于$\hat{w}$，会出现什么样的结果。用$\hat{H}$标识M，从上边我们可以得到，如果$\hat{H}$作用到$\hat{w}$上时停机，那么

$$q_0\hat{w} \vdash^*_{\hat{H}} \infty$$

如果$\hat{H}$作用到$\hat{w}$上时不停机，则

$$q_0\hat{w} \vdash^*_{\hat{H}} y_1 q_n y_2$$

这显然是无意义的。这个矛盾告诉我们，我们假设的图灵机H是不存在的。因此关于停机问题可判定性的假设是不成立的。■

有人可能会对定义12.1提出质疑，因为为了证明停机问题，我们要求H必须在非常明确的格局下开始或结束。但是，不难发现，这种任意选定的情况在证明过程中仅仅起到很小的作用；本质上，同样的推理过程对于任何其他的开始格局和终止格局都适用。出于讨论的目的，我们已经把问题和具体的定义联系到了一起，但是这并不影响结论。

记住定理12.1的内容是很重要的。定理12.1并不排除对于特定的情况，停机问题是可解的。通常，我们通过对M和w的分析可以给出图灵机是否停机。定理12.1的含义是：并不是对于所有的情况都能够给出那样的结论；没有一种算法可以对所有的w_M和w都能给出正确的判定。

303

支持定理12.1的论点已经给出，它们是经典的并且有历史意义。该定理的结论实际上隐含在以下述形式出现的先前的研究成果中。

定理12.2 *如果停机问题是可判定的，那么所有的递归可枚举语言都将是递归的。因此，停机问题是不可判定的。*

证明：设L是Σ上的递归可枚举语言，M是接受L的图灵机。设H是解决停机问题的图灵机，那么对于停机问题，我们可以构造如下的步骤：

1. 将H作用到$w_M w$上，如果H返回“否”，则根据定义可知w不在L中。

2. 如果H返回“是”，就将M作用到w上。而M是一定会停机的，所以M会最终告诉我们w是否在L中。

这样就为L构造了一个成员资格判定算法，使得L成为了递归的。但是我们已经知道有些递归可枚举的语言不是递归的。这个矛盾表明H不存在，即停机问题是不可判定的。■

由定理11.5可知停机问题和递归可枚举语言的成员资格判定问题几乎是等价的。唯一的不同在于：对于停机问题，我们不区分终态停机还是非终态停机；而对于成员资格判定问题，我们却区分。定理11.5的证明（通过定理11.3证明的）和定理12.1的证明是密切相关的，两者都是对角线化的一个版本。

12.1.3 将一个不可判定问题简化成另外一个问题

上述论证通过将停机问题和成员资格判定问题关联起来阐明了一个非常重要的简化技术。如果问题A的可判定性和问题B的可判定性是一致的，那么我们说问题A可以简化（reduced）成问题B。同时，如果我们知道了A是不可判定的，那么也就知道了B也是不可判定的。我们

将通过一些例子来说明这一点。

例12.1 状态进入问题（state-entry problem）：给定任意的图灵机$M=(Q,\Sigma,\Gamma,\delta,q_0,\square,F)$和任意的$q\in Q$，$w\in\Sigma^+$，来判定当$M$作用到$w$上时，会不会进入状态$q$。这个问题是不可判定的。 304

为了将停机问题简化为状态进入问题，我们首先假设有一个算法A能够解决状态进入问题。那么我们就可以用它来解决停机问题。例如，给定任意的M和w，我们可以修改M来得到$\hat{M}$，使得当且仅当M停机时，$\hat{M}$进入状态q并且停机。通过M的转换函数δ，我们可以知道实现这样的修改很简单。如果M停机，那么必然存在某些$\delta(q_i,a)$没有定义。因此我们对每个这样的、没有定义的δ进行修改，使得

$$\delta(q_i,a)=(q,a,R)$$

这里q为终态。这样就得到了$\hat{M}$。我们将状态进入算法A应用到（$\hat{M}$，q，w）上。如果A回答“是”，那么就进入了q状态，同时（M,w）停机。如果A回答“否”，则（M,w）不停机。

这样，状态进入问题是可判定的——这一假设就给出了停机问题的算法。因为停机问题是不可判定的，所以状态进入问题也一定是不可判定的。 □

例12.2 空白带停机问题（blank-tape halting problem）是另外一个可以通过停机问题简化来得到的问题。给定一个图灵机M，如果从空白带开始，判定M会不会停机。这个问题是不可判定的。

为了表明这个简化是如何得到的，假设我们给定M和w，首先利用M构造一个从空白带开始的新图灵机M_w，然后将w写到M_w上，同时将M_w置于格局q_0w上。之后，M_w和M就一致了。显然，M_w在空白带上停机当且仅当M在w上停机。

假设现在空白带停机问题是可判定的。对于任意给定的（M,w），我们首先构造M_w，然后对它应用空白带停机的算法。结论会告诉我们，当M作用到w上时，M是否会停机。由于对于所有的M和w，这个假设都成立，所以空白带停机问题的算法就可以被转换成停机问题的算法。又由于后者是不可判定的，所以空白带停机问题也同样是不可判定的。 □

这两个例子证明过程中的构造方法展示了一种建立不可确定性结论的通用方法。块图通常可以帮助我们表示这个过程。图12-3解释了例12.2中的构造过程。在图中，我们首先使用算法将（M,w）转化为M_w；这个算法显然已经存在了。接下来，我们使用假定存在的解决空白带停机问题的算法。将这两个算法结合就得到了停机问题的算法。但停机问题的算法是不可能存在的，因此我们得出A是不存在的。 305

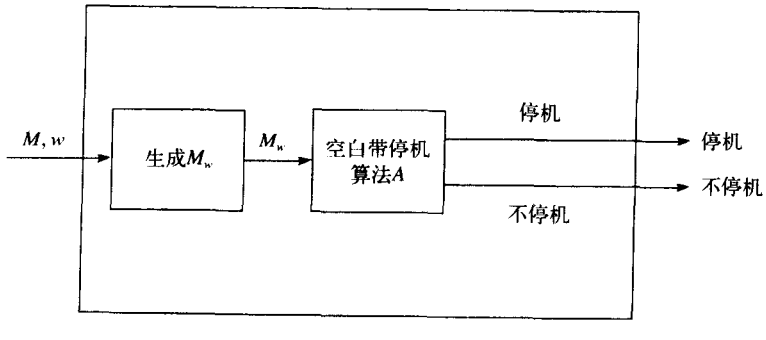


图12-3 停机问题的算法

实际上，判定问题是值域为{0, 1}的函数，即结果是一个是或否的回答。我们也可以来看一些更一般性的函数，看它们是否是可计算的。为此，我们需要根据已经建立起来的方法，将停机问题（或者是其他已知的不可判定的问题）简化成计算问题中的函数的问题。根据图灵论题，我们猜想在实际环境中遇到的函数都是可计算的，因此对于不可计算的函数的例子，必须看得更远一些。绝大多数不可计算的函数的例子一般都和预测的图灵机行为有关。

例12.3 设$\Gamma = \{0, 1, \square\}$。定义一个函数$f(n)$，它的值是所有$n$态图灵机从空白带开始，到进入停机状态所进行的最大迁移数。这个函数是不可计算的。

在我们说明这一点之前，我们先来确认$f(n)$对于所有的n都能被定义。首先，注意具有n个状态的图灵机的个数是有限的。这是因为Q和Γ都是有限的，所以δ的定义域和值域也是有限的。这也就意味着不同的δ的个数也是有限的，因此不同的n态图灵机的个数也是有限的。

在所有的n态图灵机中，有些总是停止的。例如，有些图灵机只具有终态，因此不存在迁移。而有些n态图灵机从空白带开始永不会停机，但是它们不会影响到f的定义。所有能够停机的图灵机都将进行确定数目的迁移，在这些迁移数中，我们选取最大值赋给$f(n)$。

取任意的图灵机M和正数m。我们把可以很容易地将M修改为$\hat{M}$。$\hat{M}$会在如下的两种情况下停机：作用到空白带上的M停机时，迁移次数不超过m次；或者作用到空白带上的M进行的迁移超过了m次。我们所要做的就是用M对迁移次数进行计数，当计数超过m次后停止计数。假设现在$f(n)$在某个图灵机F上是可计算的。我们把$\hat{M}$和F通过图12-4所示的方法结合在一起。首先计算$f(|Q|)$，这里的Q是M的状态集。这个值是M停机时所进行的迁移次数的最大值。我们用得到的这个值作为m来构造$\hat{M}$，而$\hat{M}$的描述用于构造通用图灵机M_u。这个M_u可以告诉我们当M作用于空白带时，M是否会停机或者在少于$f(|Q|)$次内不停机。如果我们发现M作用到空白带上的时候进行了超过$f(|Q|)$次的迁移，那么根据f的定义，这意味着M永远不会停机。这样，我们就得到了空白带停机问题的解决方案，结论的不可能性强迫我们接受事实f是不可以计算的。 □

306

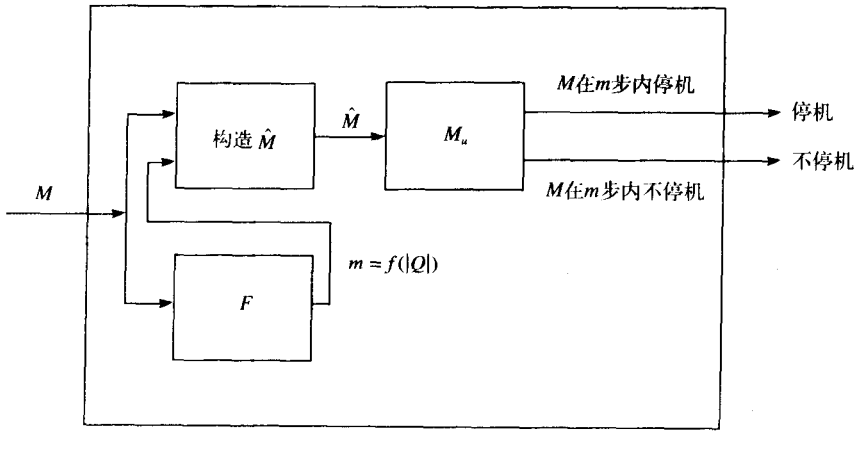


图12-4 空白带停机问题的算法

习题

1. 详细描述怎样将定理12.1中的H转化为H'。
2. 假如我们修改定义12.1，按照M作用到w上是否会停机来要求$q_0w_Mw\vdash^* q_yw$或者$q_0w_Mw\vdash^* q_nw$。重新检查定理12.1的证明过程，证明定义的不同不影响定理证明的关键过程。
3. 证明下面的问题是不可判定的。给定任意的图灵机M, $a\in\Gamma$且$w\in\Sigma^+$，判断当M作用到w上的时候，是否会写出符号a。●
4. 对于一般性的停机问题，我们要求针对任意的M和w都给出正确的算法。我们可以放宽这种一般性，例如对于所有的M和一个单独的w，找出正确的算法。如果对于任意的w都存在判定（M, w）是否停机的算法，那么我们称这个问题是可以判定的。证明即使在这种情况下，问题仍然是不可判定的。 307
5. 证明没有算法可以判定是否一个任意的图灵机会在所有的输入上停机。
6. 我们来看这个问题："图灵机在计算过程中会再次访问它的起始单元（即计算开始时，读写头所在的单元）吗？"这是一个可判定的问题吗？
7. 证明没有算法可以判定任意两个图灵机M_1和M_2是否会接受同样的语言。●
8. 如果习题7中的M_2是一个有穷自动机，那么结论如何？
9. 停机问题对于确定型下推自动机是否是可解的？即给定一个定义7.2中的pda，我们是否能够判定对于输入w，自动机是否会停机？
10. 设M是任意的图灵机，x和y是它的两个可能的瞬时描述。证明$x\vdash^*_M y$是否成立是不可判定的。●
11. 在例12.3中，给出$f(1)$和$f(2)$的值。
12. 证明一个图灵机是否会在任一输入上停机是不可判定的。
13. 设B是所有从空白带开始能够停机的所有图灵机的集合。证明这个集合是递归可枚举的，但是却不是递归的。●
14. 考虑所有具有带字母表$\Gamma = \{0, 1, \square\}$的$n$态图灵机构成的集合。给出具有这种带字母表的图灵机的个数的函数$m(n)$。
15. 设$\Gamma = \{0, 1, \square\}$，且$b(n)$是$n$态图灵机从空白带开始到停机时所检查过的最大带单元个数。证明$b(n)$是不可计算的。
16. 判断下面的结论是否成立：问题的定义域是有限的，任何一个这样的问题都是可判定的。●

12.2 递归可枚举语言的不可判定问题

我们已经确定递归可枚举语言不存在成员资格判定算法。对于递归可枚举语言来说，在决定一些属性的时候缺少算法不是异常情况，而是普遍的情况。正如我们所看到的，我们对这类语言了解得不是很多。因为递归可枚举语言具有的一般性太强，所以我们提出的任何关于它的问题基本上都是不可判定的。同样，当我们对递归可枚举语言提出问题的时候，我们 308
发现能够通过某种方法将停机问题简化成当前问题。在这里，我们给出几个例子来说明这个过程，并且从这些例子中得到处理一般情况的提示。

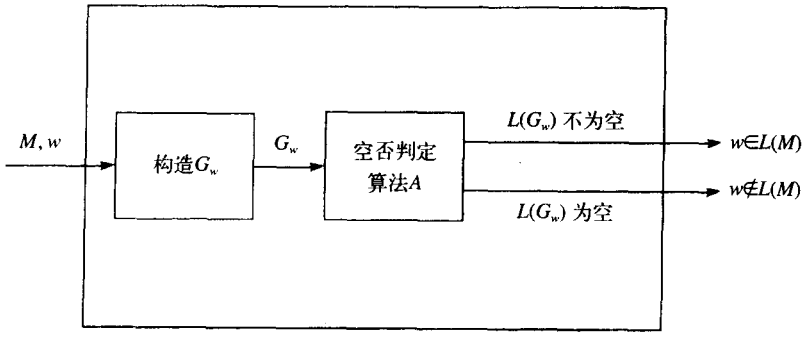
定理12.3 *设G是无限制文法。那么$L(G)=\varnothing$是否成立是不可判定的。*

证明：我们将递归可枚举语言的成员资格判定问题简化成这个问题。假设给定了图灵机M和某个符号串w。我们这样修改M：M首先用其带上某个特殊部分保存输入。然后，无论它什么时候进入终态，它都会检查自己保存的输入，并且，当且仅当它的输入是w的时候，才接受。这一点可以通过修改δ来做到。对于每个w构造一个图灵机M_w满足

$$L(M_w)=L(M)\cap\{w\}$$

利用定理11.7，我们可以构造相应的文法G_w。显然，从M和w得到G_w的构造总能够被完成。同样，我们还会知道$L(G_w)$不为空当且仅当$w\in L(M)$。

假设现在存在算法A能够判定$L(G)=\varnothing$是否成立。如果我们用T来表示产生G_w的算法，那么我们可以把T和A结合在一起，如图12-5所示那样。图12-5表示了一个图灵机，该图灵机能够针对任意的M和w来判定w是否属于$L(M)$。如果这样的图灵机存在，那么我们就会有一个递归可枚举语言的成员资格判定算法，这和前面建立的结论是矛盾的。因此，我们得出问题“$L(G)=\varnothing$”是不可判定的。■



309

图12-5 成员资格判定算法

定理12.4 *设M是任意的图灵机。那么问题$L(M)$是否是有限的——是不可判定的。*

证明：考虑停机问题（M, w）。从M出发，我们能够按照如下的步骤构造另外一个图灵机$\hat{M}$。首先，修改M的停机状态，使得这些状态在$\hat{M}$中为终态，即能够接受任意的输入。其次，修改M，使得$\hat{M}$在自己的带上首先产生w，然后再执行和M一样的计算，这其中利用新生成的w和其他未被使用的空间。换句话说，即在产生w之后，$\hat{M}$所有的迁移都和M在初始格局为q_0w的情况下的迁移一样。如果M在任意格局停机，则$\hat{M}$会在终态停机。

因此，如果（M, w）停机，那么$\hat{M}$对于所有的输入都会进入终态。如果（M, w）不停机，那么$\hat{M}$也不会停机，因此，没有接受到任何语言。即，$\hat{M}$或者接受无限语言Σ^+或者接受有限语言$\varnothing$。

如果我们假设存在算法A，它能够告诉我们$L(\hat{M})$是否是有限的，那么我们就可以按照图12-6中的方法，针对停机问题构造算法。因此，不存在算法能够判定$L(M)$是否是有限的。■

我们注意到，在定理12.4的证明中，问题“$L(M)$是有限的吗”的特定性质是非实质性的。改变该问题的性质不会对论点产生有意义的影响。

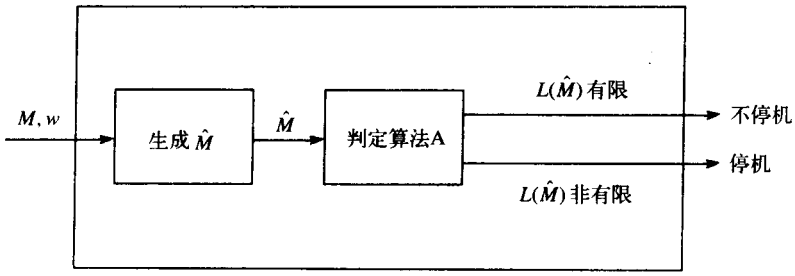


图 12-6

例12.4 证明：对于任意的图灵机M，其中$\Sigma = \{a, b\}$，问题“$L(M)$ 包含两个相同长度的不同符号串”是不可判定的。

为了证明这一点，我们使用定理12.4中所用的方法。只要$\hat{M}$没有到达停机格局，它都能被修改来接受两个符号串a和b。为了做到这一点，保存初始输入，并在计算结束的时候，将输入与a、b进行比较，$\hat{M}$仅接受这两个符号串。因此，如果（M, w）停机，那么$\hat{M}$就接受了两个同样长度的符号串。否则$\hat{M}$没有接受任何符号串。剩下的证明过程与定理12.4相同。 □ 310

用同样的方法，我们可以将问题替换为“$L(M)$ 能否接受长度为5的符号串”或“$L(M)$ 是否是正则的”，本质上，这样并不影响证明。这些问题（及与其类似的问题）都是不可判定的。对于这些问题，正式的一般性结论是Rice定理。定理说递归可枚举语言的所有的非平凡性质都是不可判定的。这里“非平凡的”（nontrivial）是指只被部分递归可枚举语言具有的属性。关于Rice定理的准确陈述和证明可以在Hopcroft &Ullman(1979) 上找到。

习题

1. 详细描述定理12.4中的$\hat{M}$的构造过程。
2. 证明在本节最后提到的两个问题是不可判定的，即
   (a) $L(M)$ 能否接受长度为5的符号串；
   (b) $L(M)$ 是否是正则的。
3. 设M_1和M_2是任意的两个图灵机。证明问题“$L(M_1) \subseteq L(M_2)$”是不可判定的。
4. 设G是任意的无限制文法。那么是否存在算法能够判定$L(G)^R$是否是递归可枚举的？
5. 如果G是任意的无限制文法，那么是否存在算法来判定$L(G)=L(G)^R$是否成立？
6. 设G_1是任意的无限制文法，G_2是任意的正则文法。证明问题$L(G_1) \cap L(G_2) = \emptyset$是不可判定的。
7. 证明习题6中的问题对于任意固定的G_2，只要$L(G_2)$不为空，问题就是不可判定的。
8. 对于一个无限制的文法G，证明问题“$L(G)=L(G)^*$是否成立”是不可判定的。(a) 用Rice定理证明；(b) 用基本原理（first principles）证明。 311

12.3 波斯特对应问题

停机问题的不可判定性在实际问题中有很多相关的结论，特别是在上下文无关语言领域

中。但是在很多情况下，直接利用停机问题工作是很困难的。需要在停机问题和其他问题之间建立一系列的中间结果来弥补两者之间的差距。这些中间结果由停机问题的不可判定性产生，但是更加接近于我们要研究的问题。因此，这些中间结果能够使得问题的证明更加简单。其中的一个中间结果是*波斯特对应问题*（Post correspondence problem）。

下面给出波斯特对应问题的定义。给定在某个字母表Σ上的两个由n个字符串组成的序列，

$$A = w_1, w_2, \cdots, w_n$$
$$B = v_1, v_2, \cdots, v_n$$

对于（A, B）对，如果存在一个非空的整数序列$i, j, \cdots, k$，满足

$$w_i w_j \cdots w_k = v_i v_j \cdots v_k$$

那么我们称（A, B）对存在*波斯特对应解*（Post correspondence solution, PC-solution）。波斯特对应问题就是要找出一种算法来判定对于任意给定的（A, B）对，是否存在一个波斯特对应解。

例12.5 设$\Sigma = \{0, 1\}$，并且取A和B为

$$w_1 = 11, w_2 = 100, w_3 = 111$$
$$v_1 = 111, v_2 = 001, v_3 = 11$$

这种情况存在图12-7中所示的波斯特对应解。

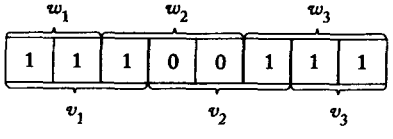


图 12-7

如果我们取

$$w_1 = 00, w_2 = 001, w_3 = 1000$$
$$v_1 = 0, v_2 = 11, v_3 = 011$$

312

就不存在波斯特对应解了。因为由元素A组成的任意符号串都要比相应的由元素B组成的符号串长。 □

在特定的例子中，我们可以通过具体的构造或者证明来说明波斯特对应解的存在与否。但是通常，我们无法对所有的情况给出一般性的解决算法。所以波斯特对应问题是不可判定的。

证明上述论断的过程有点长，为了清楚起见，我们把证明过程分为两部分。在第一部分中，我们首先介绍*修改过的波斯特对应问题*（modified Post correspondence problem）。如果存在整数序列$i, j, \cdots, k$，满足

$$w_1 w_i w_j \cdots w_k = v_1 v_i v_j \cdots v_k$$

那么我们说（A, B）对存在*修改过的波斯特对应解*（modified Post correspondence solution,

MPC-solution)。在修改过的波斯特对应问题中，序列A、B中的首个元素扮演了很特别的角色。一个修改过的波斯特对应解必须由v_1和w_1开头。注意：如果存在MPC-solution，那么必然存在PC-solution，但是反之则不成立。

修改过的波斯特对应问题是设计算法来判定任意的（A, B）对是否存在MPC-solution。这个问题也是不可判定的。我们通过将已知的不可判定问题——递归可枚举语言的成员资格判定问题——简化成修改过的波斯特对应问题来证明其不可判定性。最后，我们介绍如下构造。假设给定无限制文法$G = (V, T, S, P)$ 和一个目标符号串w。之后，我们按照图12-8所示的方法来产生（A, B）对。在图12-8中，符号串$FS\Rightarrow$被作为w_1，符号串F作为v_1。其他符号串的顺序是随意的。

A	B	
$FS\Rightarrow$	F	F是不在$V\cup T$中的符号
a	a	对每个$a\in T$
V_i	V_i	对每个$V_i\in V$
E	$\Rightarrow wE$	E是不在$V\cup T$中的符号
y_i	x_i	对P中的每个$x_i\rightarrow y_i$
$\Rightarrow$	$\Rightarrow$	

图 12-8

我们最终断言$w\in L(G)$ 当且仅当用这种方法构造的集合A和集合B具有MPC-solution。由于这个结论不是显而易见的，所以我们通过一个简单的例子来说明一下。

例12.6 设$G = (\{A, B, C\}, \{a, b, c\}, S, P)$，产生式是 313

$$S\rightarrow aABb|Bbb$$
$$Bb\rightarrow C$$
$$AC\rightarrow aac$$

并取$w = aaac$。按照给定方法生成的A和B的序列在图12-9中给出。$L(G)$ 中的符号串$w = aaac$的推导过程如下：

$$S\Rightarrow aABb\Rightarrow aAC\Rightarrow aaac$$

在图12-10中，给出了这个推导的过程，同时给出了相应的MPC-solution，这里给出了推导的前两步。推导符号串上下的整数分别展示了用于构造符号串的w和v的索引。

i	w_i	v_i
1	$FS \Rightarrow$	F
2	a	a
3	b	b
4	c	c
5	A	A
6	B	B
7	C	C
8	S	S
9	E	$\Rightarrow aaacE$
10	$aABb$	S
11	Bbb	S
12	C	Bb
13	aac	AC
14	$\Rightarrow$	$\Rightarrow$

314

图 12-9

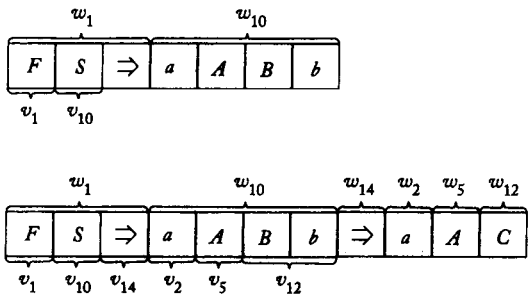


图 12-10

从图12-10中，我们仔细观察会发现，如果要构造一个MPC-solution，那么就必须从w_1开始，即从$FS\Rightarrow$开始。这个符号串包括S，因此为了匹配它，我们必须使用v_{10}或者v_{11}。在这个例子中，我们使用了v_{10}，这使得w_{10}被引入，w_{10}的引入使我们得到了部分推导中的第二个符号串。再多看几步，我们会看到符号串$w_1w_iw_j\cdots$总比$v_1v_iv_j\cdots$要长。在推导过程中，第一个序列总是领先一步。唯一的例外是最后一步，w_9使得v组成的符号串能够赶上来。图12-11显示了完整的MPC-solution。构造的过程和例子表明了下一个结论是如何建立的。 □

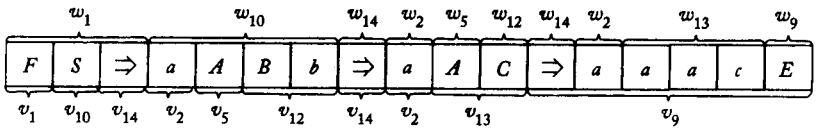


图 12-11

定理12.5 设$G=(V, T, S, P)$是任意的无限制文法，w是T^+上的任意符号串。设(A, B)是由G构造的相应的对，w是图12-8所示的过程中的w。那么(A, B)对中存在MPC-solution当且仅当$w\in L(G)$。

证明：证明过程中包含了形式化的归纳推理。我们这里省略了细节。■

根据这个结论，我们可以从递归可枚举语言的成员资格判定问题推导出修改过的波斯特对应问题，因而论证了后者的不可判定性。

315

定理12.6 修改过的波斯特对应问题是不可判定的。

证明：给定任意的无限制文法$G = (V, T, S, P)$和$w \in T^+$，我们按照前面介绍的方法来构造集合A和集合B。根据定理12.5，(A, B)对存在MPC-solution当且仅当$w \in L(G)$。

现在我们假定修改过的波斯特对应问题是可判定的。我们可以根据图12-12中描述的过程来构造G的成员资格判定问题的算法。根据G和w来构造A和B的算法是存在的，但是G和w的成员资格判定算法是不存在的。因此，我们得出修改过的波斯特对应问题是不可判定的。■

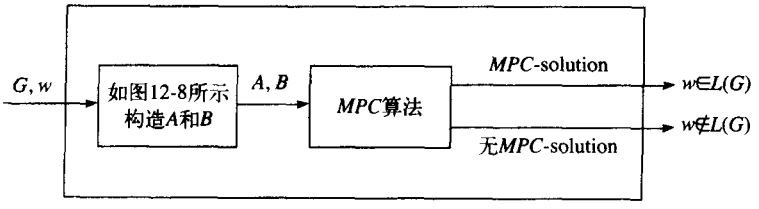


图12-12 成员资格判定算法

利用上述的工作，我们可以证明原有形式的波斯特对应问题。

定理12.7 波斯特对应问题是不可判定的。

证明：我们将证明如果波斯特对应问题是可判定的，那么修改过的波斯特对应问题也是可判定的。

我们假设给定了在某个字母表Σ上的符号串序列$A = w_1, w_2, \cdots, w_n$和$B = v_1, v_2, \cdots, v_n$。然后，我们引入新符号¢和§，并且引入新序列$C$和$D$的定义

$$C = y_0, y_1, \cdots, y_{n+1}$$
$$D = z_0, z_1, \cdots, z_{n+1}$$

对于$i = 1, 2, \cdots, n$

$$y_i = w_{i1}¢w_{i2}¢\cdots w_{im_i}¢$$
$$z_i = ¢v_{i1}¢v_{i2}¢\cdots v_{ir_i}$$

这里w_{ij}和v_{ij}分别代表了w_i和v_i中的第j个字母，且$m_i = |w_i|$，$r_i = |v_i|$。即y_i是由在w_i的每个字母后加上¢得到的，而z_i是由在v_i的每个字母前加上¢得到的。为了完成对C和D的定义，我们取 316

$$y_0 = ¢y_1$$
$$y_{n+1} = §$$
$$z_0 = z_1$$
$$z_{n+1} = ¢§$$

现在考虑(C, D)对，假设它存在PC-solution。由于¢和§的位置，这个解的左边必须有y_0，右边必须有y_{n+1}，因此这个解肯定有如下形式：

$$¢w_{11}¢w_{12}\cdots¢w_{j1}¢\cdots¢w_{k1}\cdots¢§ = ¢v_{11}¢v_{12}\cdots¢v_{j1}¢\cdots¢v_{k1}\cdots¢§$$

不考虑字符¢和§，我们可以看到

$$w_1w_j\cdots w_k = v_1v_j\cdots v_k$$

因此（A, B）对就存在MPC-solution。

我们可以倒过来证明如果（A, B）对存在MPC-solution，那么（C, D）对就存在PC-solution。

现在假设波斯特对应问题是可判定的，我们按照图12-13中所示来构造机器。机器显然可以判定修改过的波斯特对应问题。但是修改过的波斯特对应问题是不可判定的，因此，就不存在判定波斯特对应问题的算法。■

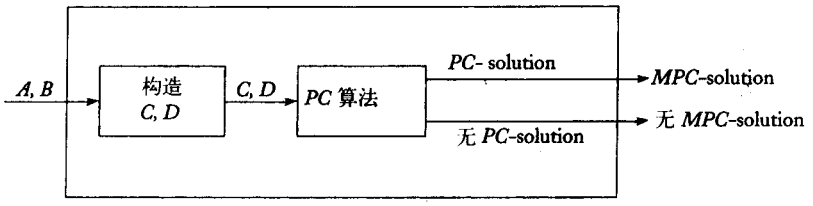


图12-13 MPC算法

习题

1. 设$A = \{001, 0011, 11, 101\}$，$B = \{01, 111, 111, 010\}$。那么（$A$, B）对是否存在一个PC-solution？是否存在MPC-solution？
2. 给出定理12.5的证明细节。
3. 证明对于$|\Sigma| = 1$，波斯特对应问题是可判定的，即对于任意给定的Σ上的（A, B）对，存在算法能够判定它是否存在一个PC-solution。
4. 假设我们限制波斯特对应问题的定义域，使得它的字母表只包含两个符号。那么相应的结果是否可判定？
5. 证明下列问题是不可判定的。
   (a) 如果存在整数序列满足$w_iw_j\cdots w_kw_1 = v_iv_j\cdots v_kv_1$，那么存在MPC-solution。
   (b) 如果存在整数序列满足$w_1w_2w_iw_j\cdots w_k = v_1v_2v_iv_j\cdots v_k$，那么存在MPC-solution。
6. 我们称给定的（A, B）对具有偶数个PC-solution当且仅当存在一个非空的偶数个整数的序列$i, j, \cdots, k$，满足$w_iw_j\cdots w_k = v_iv_j\cdots v_k$。证明对于任意（$A$, B）对是否存在偶数个PC-solution的判定问题是不可判定的。

317

12.4 上下文无关语言的不可判定问题

波斯特对应问题是学习上下文无关语言的不可判定问题的有力工具。我们将通过一些结论来说明这一点。

定理12.8 *不存在算法能够判定给定的上下文无关文法是否是无二义性的。*

证明：给定某个字母表Σ上的两个符号串序列$A = (w_1, w_2, \cdots, w_n)$和$B = (v_1, v_2, \cdots, v_n)$，选取另外一组符号$a_1, a_2, \cdots, a_n$构成新的集合，满足

$$\{a_1, a_2, \cdots, a_n\} \cap \Sigma = \emptyset$$

且给定两个语言

$$L_A = \{w_iw_j\cdots w_lw_ka_ka_l\cdots a_ja_i\}$$

和

$$L_B = \{v_i v_j \cdots v_l v_k a_k a_l \cdots a_j a_i\}$$

现在来看上下文无关文法

$$G = (\{S, S_A, S_B\}, \Sigma \cup \{a_1, a_2, \cdots, a_n\}, P, S)$$ 318

这里产生式集合P是两个子集的并：第一个集合P_A包括

$$S \to S_A$$
$$S_A \to w_i S_A a_i | w_i a_i，其中 i = 1, 2, \cdots, n$$

第二个集合P_B则有产生式

$$S \to S_B$$
$$S_B \to v_i S_B a_i | v_i a_i，其中 i = 1, 2, \cdots, n$$

现在取

$$G_A = (\{S, S_A\}, \Sigma \cup \{a_1, a_2, \cdots, a_n\}, P_A, S)$$
$$G_B = (\{S, S_B\}, \Sigma \cup \{a_1, a_2, \cdots, a_n\}, P_B, S)$$

那么，显然

$$L_A = L(G_A)$$
$$L_B = L(G_B)$$

而且$L(G) = L_A \cup L_B$。

很容易看出G_A和G_B自身都是无二义性的。如果给定$L(G)$中一个以a_i结尾的符号串，那么它在G_A中的推导必然以$S \Rightarrow S_A$开始。同样，我们能够知道在接下来的步骤中必须使用哪个产生式规则。因此，如果说G是二义性的，那么一定是因为存在某个w有两种推导形式

$$S \Rightarrow S_A \Rightarrow w_i S_A a_i \overset{*}{\Rightarrow} w_i w_j \cdots w_k a_k \cdots a_j a_i = w$$

和

$$S \Rightarrow S_B \Rightarrow v_i S_B a_i \overset{*}{\Rightarrow} v_i v_j \cdots v_k a_k \cdots a_j a_i = w$$

相应地，如果G是二义性的，那么（A, B）对的波斯特对应问题就有解。

如果存在能够解决二义性问题的算法，那么我们就可以按照图12-14将它修改成能够解决波斯特对应问题的算法。但是波斯特对应问题是不可解的，因此二义性问题也是不可判定的。■ 319

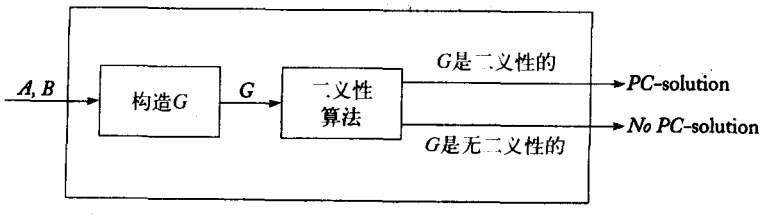


图12-14 PC算法

定理12.9 对于任意两个上下文无关文法G_1和G_2，不存在算法能够判定下面的结论是否成立

$$L(G_1)\cap L(G_2)=\varnothing$$

证明：将G_1和G_2分别视为定理12.8证明里面的G_A和G_B。假设$L(G_A)$ 和$L(G_B)$ 有一个共有的元素，即

$$S_A \overset{*}{\Rightarrow} w_i w_j \cdots w_k a_k \cdots a_j a_i$$
$$S_B \overset{*}{\Rightarrow} v_i v_j \cdots v_k a_k \cdots a_j a_i$$

那么（A, B）对就有了一个PC-solution。反之，如果（A, B）对没有一个PC-solution，那么$L(G_A)$和$L(G_B)$ 就没有一个共有的元素。因此，我们得出$L(G_A)\cap L(G_B)$为空当且仅当（A, B）对有一个PC-solution。因此定理得证。■

还有很多其他的类似结论。其中一些可以被简化成波斯特对应问题，而另一些则可以通过先建立不同的中间结果来解决（见本节习题6和习题7）。在这里我们不给出证明。

有很多与上下文无关语言相关联的不可判定问题，这一点表面上看起来让人有些吃惊，它表明在这个领域中，一些我们希望建立算法的问题是受计算限制的。例如，如果我们能够知道用BNF定义的程序设计语言是否是二义性的，或者知道两个用不同规范定义的语言实际上是否是等价的，都将会对我们有所帮助。但是，已知的结论告诉我们，这是不可能的，因此为这两个问题寻找算法是在浪费时间。然而，一定要记得对于特定的问题、甚至有可能是最有意思的那些问题，解决的方法可能是存在的。不可判定性结论告诉我们的是没有完全通用的算法，不管一个方法能够处理多少不同的情况，总有不能处理的情况。

320

习题

1. 证明定理12.8中的G_A和G_B自身都是无二义性的。

★2. 证明：对于上下文无关文法G_1和G_2来说，$L(G_1)\subseteq L(G_2)$是否成立是不可判定的。

★3. 证明：对于任意的上下文无关文法G_1和G_2，问题“$L(G_1)\cap L(G_2)$ 是否是上下文无关的”是不可判定的。

★4. 证明：如果定理12.8中的$L(G_A)\cap L(G_B)$ 是正则的，那么它肯定是空的。并用这个结论来证明：对于上下文无关G来说，问题“$L(G)$ 是正则的”是不可判定的。

★5. 设L_1是正则语言，G是上下文无关文法。证明问题“$L_1\subseteq L(G)$”是不可判定的。

★6. 设M是任意的图灵机。不失一般性地，我们可以假设每个计算都包含有偶数个迁移。对于每个这样的计算

$$q_0 w \vdash x_1 \vdash x_2 \vdash \cdots \vdash x_n$$

我们都可以构造符号串

$$q_0 w \vdash x_1^R \vdash x_2 \vdash x_3^R \vdash \cdots \vdash x_n$$

这个计算被称为是有效计算（valid computation）。

证明对于每个M，我们都可以构造三个上下文无关的文法G_1、G_2和G_3，满足：

(a) 所有有效计算的集合是$L(G_1)\cap L(G_2)$;

(b) 所有无效计算的集合（即有效计算集合的补）是$L(G_3)$。

★7 利用上面习题中的结论来证明：在所有上下文无关文法G的定义域上，“$L(G) = \Sigma^*$”是不可判定的。

★8 设G_1是上下文无关文法，G_2是正则文法。那么$L(G_1) \cap L(G_2) = \emptyset$是否是可判定的？ 321

★9 设G_1和G_2是文法，其中G_1是正则文法，那么对于下列情况，问题$L(G_1) = L(G_2)$是否是可判定的？

(a) G_2是无限制的

(b) G_2是上下文无关的

(c) G_2是正则的 322

第13章 其他的计算模型

虽然图灵机是我们所构造的最一般的计算模型，但是它们不是仅有的模型。在不同时期，提出了很多其他模型，其中一些模型乍看起来和图灵机有着很大的不同。然而，最终发现所有的这些模型都是等价的。这个领域中的很多开创性的工作是在1930年到1940年之间由一些数学家完成的，其中包括了著名的A. M. Turing。他们创立的结论不仅使机械计算概念变得更加清晰，而且使整个数学理论都变得更加清晰了。

Turing的工作成果发表于1936年，而当时还没有任何商用计算机。事实上，他的全部思想都是独立于计算机之外的。虽然Turing的思想最终在计算机科学领域中变得非常重要，但是他最初的目标并不是要为数字计算机的研究提供一个基础。为了理解Turing的意图，我们必须简单地看一下当时数学界的状态。

随着牛顿和莱布尼茨在17世纪和18世纪发现了微分和积分，人们对于数学的研究产生了极大的兴趣，使得整个学科以爆炸性的速度发展。产生了很多不同的研究领域，并且在大多数领域中都取得了显著的进展。在19世纪末，数学学科的体系已经非常庞大。数学家也开始成熟起来，他们发现一些逻辑问题需要用更加仔细的方法来处理。这就需要更加严谨的推理，因而引发了对于数学基础知识的检验。来看看为什么这种检验是必需的。在很多数学方面的书籍和论文中，我们常会碰到如下的典型情况：作者做了一些似乎正确的论述，其中间有“显而易见”和“由此可得”之类的词语。在数学推理中，这样的词语是很常见的，它们表达的意思是：如果对由此得出的结论有质疑，则可以做进一步的推理。当然，这么做是很危险的，因为这样容易遗漏问题，导致我们引入了隐含有错误的假设，或者做出了不正确的推论。每当我们看到这样的论证时，都不可避免地会质疑证明过程的正确性。通常没有有效的办法（用来检验论证的正确性），因此，那些又长又复杂的证明要在公布后，经历相当长的时间，才能发现它们的错误。然而由于实际限制，这种类型的推理被大部分数学家所接受。这种推理能够将问题解释清楚，至少增加了我们对结论正确性的信心。但是从可靠性的角度出发，这些论证是不可以接受的。

相对于这种“宽松”的数学证明，另外一种方法就是尽可能地形式化。我们从一组称为公理（axioms）的假设和一组准确定义的逻辑推理、推论规则开始。顺序使用这些规则，其中每一个规则都可以使我们由一个已证事实得到另外一个事实。这些规则必须满足：它们的正确性可以用常规的、完全机械化的方法检验。对于一个命题，如果我们从公理出发，可以用有限的逻辑步骤推导出这个命题，那么我们认为这个命题被证实是正确的。如果一个命题与另外一个正确的命题相矛盾，那么这个命题是错误的。

寻找这样的一个形式化系统是19世纪末数学家的主要目标。对于这个系统，有两个值得关注的地方。首先，系统应该是一致的（consistent）。即不能有这样的命题：经过一系列的步骤证明是正确的，但是在另外一个有效的证明下是错误的。一致性在数学中是不可或缺的，从不一致系统中推导得到的所有命题都是不被承认的。其次，系统应该是完备的（complete）。

这意味着系统中任意可表达的命题都能够被证明是正确的还是错误的。曾有一段时间，人们希望这种一致而完备的系统能够开启严格而完整的数学证明之门。但是这种希望却因K. Gödel的工作而破灭了。在他著名的不完备性定理（Incompleteness Theorem）中，Gödel证明了任何有意义的一致系统必定是不完备的，即必然有些不能被证明的命题。Gödel的革命性结论发表于1931年。

324

Gödel的工作并没有回答不可被证明的问题是否可以通过某种方式与其他可以证明的问题区别开来。因此，数学家们仍然有希望能够对大部分数学问题给出精确而且可以验证的证明过程。Turing和当时其他的一些数学家，特别是A. Church、S. C. Kleene和E. Post，对这个问题进行了深入的研究。为了研究这个问题，数学家们建立了很多的计算模型。在这些模型中，比较著名的是Church与Kleene的递归函数和（Post的）波斯特系统（Post system），当然还有很多其他值得研究的系统。在本章中，我们简要回顾一下围绕着这些研究提出的一些观点。在这里我们只能给出一些极其简单的描述，读者需要从其他的参考文献中获得具体的细节。关于递归函数（Recursive function）和波斯特系统可以参考Denning, Dennis, and Qualitz (1978)。而在Salomaa (1973) 和Salomaa (1985) 中，有很好的关于各种其他重写系统（rewriting system）的讨论。

我们在这里介绍的计算模型和其他的一些模型有着不同的起源。但是最终发现它们在执行计算的能力上都是等同的。这些结论最初源于丘奇论题（Church's thesis）。这个论题表明所有可能的计算模型，如果它们足够宽广，那么必然是等同的。这也隐含着，对于模型来说，总是会有各种各样的限制，使得某个函数不能很好地表述。这个论题和图灵机论题十分接近，将它们组合在一起称为丘奇-图灵论题（Church-Turing thesis）。它提出了关于算法计算的一个基本的原则（不可证明的），同时，它为不可能存在更强计算模型这一理论提供了强有力的证据。

13.1 递归函数

函数是数学中很基础的概念。正如在1.1节中总结的，函数中有一个被称为定义域的集合和一个被称为值域的集合，函数是将定义域中的一个元素映射到值域上一个元素的规则。它的概念是很宽泛的，因此，如何表示这种规则随之产生。有很多种方法可以定义函数。有些被经常使用，而另一些则不然。

325

我们对如$f(n) = n^2 + 1$的函数表示方法已经很熟悉了。它将任意的参数n和自身相乘，然后再加一。函数的定义很明确，我们可以用完全机械化的方式来计算函数值。但是，为了完善f的定义，我们还必须说明它的定义域。例如对于这个函数，我们可以把它的定义域定义为所有整数的集合，那么相应的值域就是正整数集合的一些子集。

既然我们可以用这种方法来表示很复杂的函数，那么我们就想知道这种表达形式是否具有一般性。对于一个定义好的函数（即，我们知道函数的定义域元素和值域元素之间的关系），是否能够用这种函数形式表示？为了回答这个问题，我们首先必须明确什么样的形式可以被接受。为此，我们首先引入一些基本的函数，然后再给出一些规则，使得它们能够构造出更复杂的函数。

13.1.1 原始递归函数

为简单起见，我们只考虑有一个或者两个变量的函数。函数的定义域或者是非负整数集合I或者是$I \times I$，函数的值域属于I。在这些设定下，我们来看几个基本函数:

1. *零函数*（zero function）$z(x) = 0$，其中所有的$x \in I$。

2. *后继函数*（successor function）$s(x)$，函数的值是与x相邻的下一个正整数，即使用常规表示法为$s(x) = x + 1$。

3. *投影函数*（projector function）

$$p_k(x_1, x_2) = x_k \text{（这里}k = 1, 2\text{）}$$

基于上述基本函数，我们介绍两种构造较复杂函数的方法:

1. *组合*（composition），即通过如下方法构造函数，这里g_1、g_2和h都是函数。

$$f(x, y) = h(g_1(x, y), g_2(x, y))$$

2. *原始递归*（Primitive recursion），这里通过如下方法递归地定义函数，这里g_1、g_2和h都是函数。

$$f(x, 0) = g_1(x)$$
$$f(x, y + 1) = h(g_2(x, y), f(x, y))$$

326

我们将通过一些例子来说明如何使用上述的基本函数和构造方法。

例13.1 整数x与y的和可以由函数$add(x, y)$来实现，这个函数可以递归地定义为

$$add(x, 0) = x$$
$$add(x, y + 1) = add(x, y) + 1$$

为了得到3加2的结果，我们可以递归地使用这些规则:

$$\begin{aligned} add(3, 2) &= add(3, 1) + 1 \\ &= (add(3, 0) + 1) + 1 \\ &= (3 + 1) + 1 \\ &= 4 + 1 = 5 \end{aligned}$$

□

例13.2 利用例13.1中定义的add函数，我们可以定义乘法为

$$mult(x, 0) = 0$$
$$mult(x, y + 1) = add(x, mult(x, y))$$

形式化地，第二步是原始递归的应用，这里的add相当于h，而$g_2(x, y)$是投影函数$p_1(x, y)$。□

例13.3 减法不是特别明显。考虑到我们的系统中不允许有负数，所以我们必须首先定义减法。通过普通的减法来定义（系统允许的）减法

$$x \dot{-} y = x - y, \text{ 当}x \geqslant y\text{时}$$
$$x \dot{-} y = 0, \text{ 当}x < y\text{时}$$

我们有时也把$\dot{-}$称为*真减*（monus），通过它定义的减法的值域是I。

327

现在定义前驱（predecessor）函数

$$pred(0) = 0$$
$$pred(y + 1) = y$$

通过它，可以这样定义减法

$$subtr(x, 0) = x$$
$$subtr(x, y + 1) = pred(subtr(x, y))$$

为了证明5 − 3 = 2，我们通过重复使用定义来简化命题以求证

$$\begin{aligned} subtr(5, 3) &= pred(subtr(5, 2)) \\ &= pred(pred(subtr(5, 1))) \\ &= pred(pred(pred(subtr(5, 0)))) \\ &= pred(pred(pred(5))) \\ &= pred(pred(4)) \\ &= pred(3) \\ &= 2 \end{aligned}$$

□

同样，我们可以定义整数的除法，但是我们把它作为一个习题。如果我们接受上述内容，那么我们会发现所有基本的算术运算都可以通过上面描述的基本过程来构造。通过准确定义的代数运算，可以构造其他的更为复杂的运算，十分复杂的运算也可以通过简单运算来构造。我们把可以通过这种方式构造的函数称为原始递归的。

定义13.1 *一个函数可以被称为原始递归的（primitive recursive），当且仅当它可以由基本函数z, s, p_k通过组合和原始递归来得到。*

注意：如果g_1，g_2和h都是全函数，那么通过它们定义得到的f也是全函数。由此可知每一个定义在I和$I \times I$上的原始递归函数都是全函数。

328

原始递归函数的表达能力是很强的。大多数常见函数都是原始递归的。但是，并非所有的函数都属于这种类型。

定理13.1 *用F来表示所有从I到I的函数构成的集合。那么在F中有些函数不是原始递归的。*

证明：每个原始递归函数都可以通过有限的符号串来表示它是如何定义的。可以将这些符号串编码并按标准顺序排列。因此，所有的原始递归函数都是可数的。

现在假设所有函数构成的集合也是可数的，那么我们可以按照某种顺序将所有的函数列举出来，比如说，$f_1, f_2, \cdots$接下来，我们构造函数g并定义为

$$g(i) = f_i(i) + 1, i = 1, 2, \cdots$$

显然，g是符合函数定义的，因此应该在F中。但同样明显的是g和每个f_i在对角线的位置上不同。这个矛盾表明F是不可数的。

将上面的两个结论和在一起，我们证明了F中必然存在某个函数不是原始递归的。■

事实上，这个定理可以进一步扩展：不仅有些函数不是原始递归的，而且有些实际可算函数也不是原始递归的。

定理13.2 *用C来表示所有从I到I的可算函数集合。那么在C中有些函数不是原始递归的。*

证明：通过上一个定理的证明，我们已经知道所有原始递归函数构成的集合是可数的。我们把这个集合中的函数表示为$r_1, r_2, \cdots$并且定义一个函数g

$$g(i) = r_i(i) + 1$$

通过构造，我们可以看到g和每一个r_i都是不同的，因此g不是原始递归的。但是显然g是可计算的。因此定理得证。■

在对角理论中，证明有些可算函数不是原始递归的——这种非构造证明是相当简单的练习。因为这种函数的实际构造是一件很复杂的事情。在这里，我们还会给出一个看起来十分简单的例子，但是关于它不是原始递归的证明却是相当长的。 329

13.1.2 Ackermann函数

Ackermann函数是一个从$I \times I$到I上的函数，定义如下：

$$A(0, y) = y + 1$$
$$A(x, 0) = A(x - 1, 1)$$
$$A(x, y + 1) = A(x - 1, A(x, y))$$

很难看出这里的函数A是一个完全的、可计算的函数。事实上，我们可以很容易地写出这个函数计算的递归的计算机程序。然而，尽管表面上看起来简单，但是，Ackermann函数却不是原始递归的。

当然，我们无法直接从A的定义来推导出来。尽管我们在这里给出的A的定义并不是一个原始递归函数的形式，但是很可能存在另外一个合适的形式。这里的情形类似于我们前面遇到的：证明一个语言不是正则的或者不是上下文无关的。我们需要从所有原始递归函数的类里面提取一些共有的属性，来表明Ackermann函数是不满足这些属性的。对于原始递归函数来说，它们的一个属性就是增长率。随着$i \to \infty$，原始递归函数的增长速度会出现一个极限，而Ackermann函数则违反了这个限制。很容易证明：Ackermann函数增长得十分迅速。这个在习题9至习题11中会看到。而关于原始递归函数的增长速度的限制，将由下面的定理准确给出。它的证明比较冗长而且专业化，所以这里被省略了。

定理13.3 *设f是任意的原始递归函数。那么必然存在某个整数n，对于所有的$i = n, n + 1, \cdots$满足*

$$f(i) < A(n, i)$$

证明：证明的细节可以参考Denning, Dennis, and Qualitz (1978, p.534)。■

如果我们接受这个结论，那么很容易证明Ackermann函数不是原始递归的。

定理13.4 *Ackermann函数不是原始递归的。*

证明：考虑函数

$$g(i) = A(i, i)$$ 330

如果A是原始递归的，那么g也是。此外，根据定理13.3，必然存在某个n，对于所有的i，满足

$$g(i) < A(n, i)$$

现在如果我们取$i = n$，那么我们得到了矛盾

$$\begin{aligned} g(n) &= A(n, n) \\ &< A(n, n) \end{aligned}$$

这样就证明了Ackermann函数不是原始递归的。■

13.1.3 μ递归函数

为了扩展递归函数的思想使其包含Ackermann函数和其他可计算函数，我们必须向规则中添加一些内容，使修改后的规则能够构造这样的（递归）函数。其中的一个方法就是引入μ或者是最小化（minimalization）运算符，定义为

$$\mu y(g(x, y)) = \text{满足} g(x, y) = 0 \text{的最小} y$$

在这个定义中，我们假设g是全函数。

例13.4 设函数

$$g(x, y) = x + y - 3$$

这是一个全函数。如果$x \leqslant 3$，那么

$$y = 3 - x$$

是最小化的结果。如果$x > 3$，那么就没有（$y \in I$）能满足$x + y - 3 = 0$。因此

$$\begin{aligned} \mu y(g(x, y)) &= 3 - x \quad (x \leqslant 3) \\ &= \text{无定义} \quad (x > 3) \end{aligned}$$

我们可以从这里看到即使$g(x, y)$是一个全函数，$\mu y(g(x, y))$也可能只是部分定义的。□

如上例所示，最小化运算符使得递归地定义部分函数成为可能。但是结果表明它也增强了定义全函数的能力（能够定义所有可计算函数了）。这里，我们同样只给出主要的结论和如何得到详细证明的参考文献。

331

定义13.2 *如果一个函数的构造可以通过使用μ运算符、组合运算及原始递归运算来完成，那么我们称这个函数是μ递归的。*

定理13.5 *一个函数是μ递归的当且仅当它是可计算的。*

证明：证明细节请参考Denning, Dennis, and Qualitz (1978, Chapter 13)。■

μ递归的函数因此给出了另外一个计算模型。

习题

1. 使用例13.1和例13.2中的定义来证明$3 + 4 = 7$和$2 \times 3 = 6$。
2. 定义函数

$$\begin{aligned} greater(x, y) &= 1 \quad x > y \\ &= 0 \quad x \leqslant y \end{aligned}$$

证明这个函数是原始递归的。●

3. 定义函数

$$equals(x, y) = 1 \quad x = y$$
$$= 0 \quad x \neq y$$

证明这个函数是原始递归的。

4. 定义函数f为

$$f(x, y) = x \quad x \neq y$$
$$= 0 \quad x = y$$

证明f是原始递归的。 332

★5. 整数除法可以通过两个函数div和rem来定义:

$$div(x, y) = n$$

这里的n是满足$x \geqslant ny$的最大整数。而

$$rem(x, y) = x - ny$$

证明div和rem都是原始递归的。

6. 证明函数$f(n) = 2^n$是原始递归的。
7. 证明函数$g(x, y) = x^y$是原始递归的。
8. 编写一个计算Ackermann函数的计算机程序，并用它来计算$A(2, 5)$ 和$A(3, 3)$。
9. 证明Ackermann函数的下列结论成立。
   (a) $A(1, y) = y + 2$
   (b) $A(2, y) = 2y + 3$
   (c) $A(3, y) = 2^{y+3} - 3$
10. 利用习题9来计算$A(4, 1)$ 和$A(4, 2)$。
11. 给出$A(4, y)$ 的一个一般表达式。
12. 给出$A(5, 2)$ 的递归调用步骤。
13. 证明Ackermann函数是一个定义在$I \times I$上的全函数。
14. 利用习题8中的构造程序来计算$A(5, 5)$。能解释你所观察到的过程吗?
15. 对于下面的每个g，计算$\mu y(g(x, y))$，并确定它的定义域。
   (a) $g(x, y) = xy$
   (b) $g(x, y) = 2^x + y - 3$
   (c) $g(x, y) = (x - 1)/(y + 1)$ 的整数部分
   (d) $g(x, y) = x \bmod (y + 1)$
16. 在例13.3里面定义的$pred$函数，虽然看起来很清楚，但是它却没有严格遵循原始递归函数的定义。请重新定义这个函数，使其具有正确的形式。 333

13.2 波斯特系统

波斯特系统看起来很像无限制文法，它包含了一个字母表和几个产生式规则，通过这些可以推导出连续的符号串。但是波斯特系统和无限制文法在产生式的使用上有很大的区别。

定义13.3 波斯特系统∏可以定义为

$$\Pi = (C, V, A, P)$$

这里

C是一个由常量构成的有限集合，这个集合又包含了两个不相交的集合C_N和C_T，C_N称为非终结常量（nonterminal constants），C_T称为终结常量（terminal constants）；

V是一个变量的有限集；

A是一个从C^*得到的有限集，称为公理；

P是一个产生式的有限集。

波斯特系统中的产生式必须满足一些特定的限制。它们必须有如下形式

$$x_1V_1x_2\cdots V_nx_{n+1}\rightarrow y_1W_1y_2\cdots W_my_{m+1} \tag{13-1}$$

这里$x_i, y_i\in C^*$，$V_i, W_i\in V$，而且要求任何变量最多能在产生式的左部出现一次，所以

$$V_i \neq V_j (i \neq j)$$

而所有在右部出现的变量必须也在左部出现，即

$$\bigcup_{i=1}^{m} W_i \subseteq \bigcup_{i=1}^{n} V_i$$

假设我们有一个终结符的符号串形如$x_1w_1x_2w_2\cdots w_nx_{n+1}$，这里，子串$x_1, x_2\cdots$匹配了式（13-1）里面的相应字符串，$w_i\in C^*$。我们可以使$w_1 = V_1$, $w_2 = V_2$, …，然后用这些值替换式（13-1）右边的W。既然每个W都有相应的V_i（在左部），并且都有一个唯一的值，那么我们可以得到新的符号串$y_1w_iy_2w_j\cdots y_{m+1}$。我们可以将过程写成

334

$$x_1w_1x_2w_2\cdots x_{n+1}\Rightarrow y_1w_iy_2w_j\cdots y_{m+1}$$

现在波斯特系统可以被视为一个文法了，我们来看一下由波斯特系统得到的语言。

定义13.4 由波斯特系统$\Pi = (C, V, A, P)$生成的语言是$L(\Pi) = \{w\in C_T^* : w_0\overset{*}{\Rightarrow}w, w_0\in A\}$

例13.5 给定波斯特系统

$$C_T = \{a, b\}$$
$$C_N = \varnothing$$
$$V = \{V_1\}$$
$$A = \{\lambda\}$$

和产生式

$$V_1\rightarrow aV_1b$$

这个系统允许如下推导

$$\lambda\Rightarrow ab\Rightarrow aabb$$

第一步，我们使用式（13-1），让$x_1 = \lambda$, $V_1 = \lambda$, $x_2 = \lambda$, $y_1 = a$, $W_1 = V_1$, $y_2 = b$。在第二步，我们重定义$V_1 = ab$，其他保持不变。如果继续推导下去，你很快就会发现这个特殊的波斯特系统

生成的语言是$\{a^n b^n : n \geqslant 0\}$。 □

例13.6 对于波斯特系统

$$C_T = \{1, +, =\}$$
$$C_N = \varnothing$$
$$V = \{V_1, V_2, V_3\}$$
$$A = \{1 + 1 = 11\}$$

335

和产生式

$$V_1 + V_2 = V_3 \rightarrow V_1 1 + V_2 = V_3 1$$
$$V_1 + V_2 = V_3 \rightarrow V_1 + V_2 1 = V_3 1$$

那么这个系统允许如下推导

$$1 + 1 = 11 \Rightarrow 11 + 1 = 111$$
$$\Rightarrow 11 + 11 = 1111$$

如果我们把这里面由1组成的符号串解释为整数的一元表示，那么推导可以写成

$$1 + 1 = 2 \Rightarrow 2 + 1 = 3 \Rightarrow 2 + 2 = 4$$

这个波斯特系统生成的语言是所有整数加法的等式集合，如：$2 + 2 = 4$，这些等式都是从公理$1 + 1 = 2$得到的。 □

例13.6展示了波斯特系统可以通过公理的集合来进行严格的数学证明。这也是设计波斯特系统的意图。同时，例13.6也展示了如此严密的方式的根本性缺陷和为什么它得不到广泛的使用。虽然波斯特系统对于证明复杂定理来说有些繁琐，但却是通用的计算模型，像下面的定理表述的那样。

定理13.6 *一个语言是递归可枚举的，当且仅当存在能够产生它的波斯特系统。*

证明：证明过程相对简单，因此，我们在这里只做简要描述。首先，因为波斯特系统的推导过程是完全机械的，所以可以在图灵机上来实现这一过程。因此，任何由波斯特系统生成的语言都是递归可枚举的。

反过来，记住：任何递归可枚举语言都可以由某个无限制文法G生成，产生式都具有如下形式：

$$x \rightarrow y$$

这里$x, y \in (V \cup T)^*$。给定任意的无限制文法G，我们可以得到一个波斯特系统$\Pi = (V_\Pi, C, A, P_\Pi)$，这里$V_\Pi = \{V_1, V_2\}$，$C_N = V$，$C_T = T$，$A = \{S\}$。对于文法中的每个产生式$x \rightarrow y$，构造波斯特系统中的产生式为：

$$V_1 x V_2 \rightarrow V_1 y V_2$$

接下来要证明一个w能够被波斯特系统生成，当且仅当它属于由G生成的语言，这个证明就简单了。■

336

习题

1. 对于$\Sigma = \{a, b, c\}$，构造一个波斯特系统，使之能够生成下列语言。
   (a) $L(a^*b + ab^*c)$
   (b) $L = \{ww\}$ ●
   (c) $L = \{a^n b^n c^n\}$
2. 构造一个波斯特系统，使之能够产生$L = \{ww^R : w \in \{a, b\}^*\}$。
3. 对于一个波斯特系统，如果$\Sigma = \{a\}$，公理为$\{a\}$，并且具有产生式$V_1 \to V_1V_1$，那么它最终生成的是什么语言？ ●
4. 如果习题3中的公理集合为$\{a, ab\}$，那么这个波斯特系统生成的是什么样的语言？
5. 构造一个波斯特系统能够证明整数乘法的等式，从公理$1 \times 1 = 1$开始。 ●
6. 给出定理13.6的证明细节。
7. 对于具有产生式

$$V \to aVV$$

   和公理集合$\{ab\}$的波斯特系统能够生成什么样的语言？
8. 一个严格的波斯特系统是这样定义的：系统中的每个产生式$x \to y$除了满足通常的要求之外，还要满足出现在产生式两边的变量的数目是相同的，即式（13-1）中的$n = m$。证明对于每个由波斯特系统生成的语言L，存在某个严格的波斯特系统也能够生成L。

13.3 重写系统

我们学过的各种文法与波斯特系统有着很多相同之处：它们都是基于某个字母表的，都能够从一个符号串得到另外一个符号串。即使是图灵机也可以归入此类，因为它的瞬时描述也是一个能够完全定义其格局的符号串。而程序则是由一个符号串得到另外一个符号串的一组规则。我们观察到的这些结果能够被形式化地表示为*重写系统*（rewriting system）。一般来说，一个重写系统包含了一个字母表Σ和一个产生式的集合。产生式能够将Σ^+中的一个符号串转换为另外一个符号串。一个重写系统与另外一个重写系统的区别在于字母表Σ的种类和产生式应用的限制。

337

这个概念很宽泛，除了我们以前碰到的各种情况以外，还有很多特殊情况都能被归入到这类系统中。这里我们简要地介绍一些不太为人所知但却比较有趣的内容，同时提供通用的计算模型。具体的内容请参考Salomaa (1973) 和Salomaa (1985)。

13.3.1 矩阵文法

矩阵文法（matrix grammars）与我们以前学习过的文法（也被称为*短语结构文法*（phrase-structure grammars））不同，它们主要的区别在于对产生式的使用上。对于矩阵文法，它的产生式集合包含了一系列的子集$P_1, P_2, \cdots, P_n$，其中每一个子集都是一个有序的序列

$$x_1 \to y_1, x_2 \to y_2, \cdots$$

每当某个集合P_i中的第一个产生式被使用的时候，我们必须对刚刚产生的符号串使用第二个

产生式，接下来是第三个，依此类推。如果我们要使用集合P_i中的第一个产生式，那么整个集合中的产生式都必须能够同样被使用。否则，就不能使用集合P_i中的第一个产生式。

例13.7 给定矩阵文法

$$P_1 : S \to S_1S_2$$
$$P_2 : S_1 \to aS_1,\ S_2 \to bS_2c$$
$$P_3 : S_1 \to \lambda,\ S_2 \to \lambda$$

我们能够得出如下推导:

$$S \Rightarrow S_1S_2 \Rightarrow aS_1bS_2c \Rightarrow aaS_1bbS_2cc \Rightarrow aabbcc$$

注意，每当我们使用集合P_2中的第一个产生式规则来得到a时，都必须同时使用集合中的第二个产生式规则，产生相应的b和c。这使得我们可以很容易地看出由这个矩阵文法生成的终结符串的集合是

$$L = \{a^nb^nc^n : n \geqslant 0\}$$ □

矩阵文法包含了短语结构文法，可以将短语结构文法视为矩阵文法中每个集合P_i都只包含一个产生式的特殊情况。同样，由于矩阵文法表示了运算过程，所以它们也受制于丘奇论题。我们从这里得出矩阵文法和短语结构文法作为计算模型具有同样的表达能力。但是，如例13.7所示，有时使用矩阵文法给出的解决方案要比用不受限的短语结构文法得到的简单很多。 338

13.3.2 马尔科夫算法

马尔科夫算法（markov algorithm）是一个重写系统，它的产生式$x \to y$是有顺序的。在一个推导过程中，第一个可使用的产生式必须被使用。而且，在最左边出现的子串x必须被替换为y。有些产生式可以被单独提出来作为终结产生式（terminal production），将它们表示为$x \to .y$。一个推导从某个符号串$w \in \Sigma$开始并继续，直到使用了终结产生式或者没有可以使用的产生式为止。

对于语言接受，集合$T \subseteq \Sigma$被确定为终结符集合。从终结符串开始，一直使用产生式直到产生了空串。

定义13.5 设M是一个具有字母表Σ上和终结符T的马尔科夫算法，那么它接受的语言是

$$L(M) = \{w \in T^* : w \overset{*}{\Rightarrow} \lambda\}$$

例13.8 考虑具有$\Sigma = T = \{a, b\}$的马尔科夫算法，其产生式为

$$ab \to \lambda$$
$$ba \to \lambda$$

在推导的过程中，每一步都会消除一个子串ab或者ba，因此

$$L(M) = \{w \in \{a, b\}^* : n_a(w) = n_b(w)\}$$ □ 339

例13.9 构造一个马尔科夫算法接受语言$L = \{a^nb^n : n \geqslant 0\}$。

答案是:

$$ab \to S$$
$$aSb \to S$$
$$S \to .\lambda$$

在这个例子中，如果我们取前两个产生式，颠倒它们的左部和右部，那么我们就得到了一个上下文无关文法，它能生成语言L。在某种意义上，马尔科夫算法是简单的短语结构文法。然而我们还不能随便下这个结论，因为现在我们还不知道如何来处理最后一个产生式。但是，这个观察结果的确为我们提供了一个起始点——证明下面描述马尔科夫能力的定理。 □

定理13.7 一个语言是递归可枚举的当且仅当对它存在一个马尔科夫算法。

证明：请参考Salomaa (1985, p.35) ■

13.3.3 L系统

L系统（L-systems）的起源比较出乎我们的意料。它们的创始人A. Lindenmayer使用它们来对特定器官的增长模式进行建模。L系统本质上是并行的（parallel）重写系统。根据这一点，我们知道在每一步推导中，每个符号都必须被重写。因此，我们可以得出L系统的产生式必须具有如下形式

$$a \to u \tag{13-2}$$

这里$a \in \Sigma$，$u \in \Sigma^*$。当一个符号串被重写的时候，在新符号串产生之前，必须对（符号串中的）每个符号都应用一个这样的产生式。

340

例13.10 设$\Sigma = \{a\}$且$a \to aa$，定义一个L系统。从符号串a开始，我们可以进行这样的推导

$$a \Rightarrow aa \Rightarrow aaaa \Rightarrow aaaaaaaa$$

这样推导出的符号串集合显然是

$$L = \{a^{2n} : n \geqslant 0\}$$

注意这个特定的重写系统能够处理那些用短语结构文法难以解决的问题。 □

现在我们已经知道具有类似于式（13-2）形式的产生式的L系统不足以表示所有的算法计算。我们可以对它进行扩充来保证它的一般性。在一个扩展的L系统中，产生式具有形式

$$(x, a, y) \to u$$

这里的$a \in \Sigma$且$x, y, u \in \Sigma^*$，其中a仅当作为符号串xay的一部分的时候，才可以被替换为u。众所周知，这个扩展的L系统是通用的计算模型。具体的细节请参考Salomaa (1985)。

习题

1. 为$L = \{ww : w \in \{a, b\}^*\}$构造矩阵文法。
2. 下面的矩阵文法能够产生什么样的语言？

$$P_1 : S \to S_1S_2$$
$$P_2 : S_1 \to aS_1b,\ S_2 \to bS_2a$$
$$P_3 : S_1 \to \lambda,\ S_2 \to \lambda$$

3. 假设我们将例13.7里的最后一组产生式修改为

$$P_3 : S_1 \rightarrow \lambda, S_2 \rightarrow S$$

那么这个修改后的矩阵文法能够产生什么样的语言？

4. 为什么例13.9中的马尔科夫算法不接受$abab$？ 341

5. 找到一个马尔科夫算法能够得到语言$L = \{a^n b^n c^n : n \geqslant 1\}$。

★6. 构造一个马尔科夫算法使得它能够接受$L = \{a^n b^m a^{nm} : n \geqslant 1, m \geqslant 1\}$。

7. 构造一个L系统能够生成$L(aa^*)$。

8. 当具有如下产生式的L系统从符号串a开始推导时，能够得到一个什么样的符号串集合？

$$a \rightarrow aa$$

$$a \rightarrow aaa$$

342

第14章 计算复杂性介绍

在学习算法和计算理论的时候，我们并不太关心如何将这些想法应用到具体的计算机上。我们似乎只关心对于特定的问题，相应的算法存在与否。这个对于我们研究理论是一个很好的起点，但是却缺乏实际的意义。对于实际的计算，我们不仅需要知道解决问题的一般原则，而且需要构造具有足够效率的算法。我们称可以被有效解决的问题为可解的（tractable）。对于这个概念，我们将在本章中给出一个更加详尽的定义。

在软件开发的实际过程中，效率涉及很多方面。有时候，我们关心如何有效地利用计算机资源，例如处理机时间和存储空间。也有时候，我们可能会比较关心如何能够高效地开发软件，如何有效地对软件进行维护，如何保证它是可靠的。还有一些时候，我们比较强调解决用户问题的效率。所有这些都太复杂了，不能通过抽象的理论来描述。我们所能做的就是集中考虑一些具体的问题，并由这些问题抽象出一个合适的框架结构。大多数已建立的结论都会强调空间效率和时间效率，从而得出了复杂性理论（complexity theory）这个重要主题。在研究复杂性时，首要考虑的是计算效率，它是通过计算在时间和空间上的需求来衡量的。我们把它们分别称为算法的时间复杂度（time-complexity）和空间复杂度（space-complexity）。

计算复杂性理论的范畴很宽泛，其中大部分都超出了本书的范畴。但是有些结论可以被简单地表述且易于理解，而这些结论又进一步地揭示了语言和计算的本质。在本节中，我们给出复杂性理论的一个简要概述。这其中，大部分结论的证明都比较难，我们没有提供证明，只给出相应的参考文献。我们在这里只是给出这个领域里面的主要研究对象，并且说明它是怎么和我们知道的语言及自动机联系起来的。由此，我们在主题的选择和讨论的正式程度上就有了很大的自由。

我们这里仅限于时间复杂度的讨论。空间复杂度与它有着类似的结论，但是时间复杂度更容易理解。

14.1 计算的效率

我们首先从一个具体的例子开始。给定一个由1000个整数构成的表，我们想要把它们按照某种顺序（比如说，升序）排列起来。排序问题是简单的，但是它也是计算机领域的一个非常基础性的问题。如果要回答“执行排序需要多长时间”，那么我们就需要很多的信息。很显然，表中项的个数在执行时间的长短上起着很重要的作用，但是还有很多其他的因素。这里还会涉及我们所用的计算机和执行排序的程序。此外，有若干种排序方法，因此对于排序算法的选择也是很重要的。其实，即使是对上面的问题进行简单的估计也会需要考虑很多的事情。如果我们要对排序进行一个大致的描述，那么我们就不得不忽略大多数的细节问题，而集中思考那些最基础的问题。

对于计算复杂性的讨论，我们需要做下列假设来简化问题:

1. 我们讨论的模型将是图灵机。具体采用的图灵机类型将在后边的讨论中给出。

344 2. 我们用n来表示问题的规模。对于我们排序问题，显然，n就代表了表中项的个数。尽管一个问题的规模不一定总能这么容易地描述，但是我们通常可以把它映射为一个正整数。

3. 在分析算法的时候，我们更重视它在一般情况下的效率，而不是在一个特定情况中的效率。我们特别关心当问题的规模增大的时候，算法效率会有什么变化。因此，主要的问题就是当n增大的时候，对资源需求的增长有多快。

我们的目标是把问题的时间需求表示为问题规模的函数。在这里，我们使用图灵机作为计算模型。

首先，我们给出一些关于图灵机的时间的概念。我们认为图灵机在一个时间单元内执行一次迁移。因此，一个计算的执行时间就是图灵机的迁移次数。正如前面所述，我们重点考察计算需求是如何随着问题规模的增大而增长的。一般来说，即便给定问题规模的大小，问题在不同情况下的增长速度也是不同的。我们在这里只关心有着最高资源要求的最坏情况。我们说一个计算的时间复杂度为$T(n)$，是指对于任意规模为n的问题，图灵机将在$T(n)$次迁移之内完成计算。

在制定了作为计算模型的具体的图灵机之后，我们可以写出具体的算法，并且通过计算具体的执行步数来分析算法。但是，由于各种各样的原因，这种方法不是很有效。首先，具体的执行步数会随着程序的一些细节而变化，因此，步数的多少很大程度上取决于程序员（编写的程序）。其次，从实际的观点出发，我们关心的是算法在现实世界中的表现，而这和它在图灵机上的表现是很不一样的。我们所能希望的最好的情况就是在图灵机上做的分析能够体现实际特性中的主要方面。例如，理论分析能够给出实际计算的时间复杂度的渐进增长速率。因而，我们最初对于算法资源需求的理解总是尝试用数量级（order-of-magnitude）分析，这里我们要用到第1章中介绍过的符号O、Θ和Ω。尽管我们的分析不是特别正规，但却经常能够得到有用的信息。

例14.1 给定n个数$x_1, x_2, \cdots, x_n$的集合和一个键值x，判定集合中是否包含x。

除非集合中的数是按照某种顺序排列的，否则最简单的算法是线性查找（linear search）。在这种算法中，我们要将x与集合中的数进行一一比较，直到找到某个与x匹配的数或者到了集合中的最后一个数字。由于我们要找的数可能在第一次比较时就能匹配，也可能到最
345 后一次才能匹配，所以我们不能预计这需要多少工作。但是我们知道，在最坏的情况下，需要进行n次比较。我们因此可以说这个线性查找的时间复杂度是$O(n)$，或者更好一点是$\Theta(n)$。在对这个问题进行分析的时候，我们并没有指定具体的机器或者假设算法是如何实现的。 □

习题

1. 假设给定一个由n个数$x_1, x_2, \cdots, x_n$构成的集合，并要求判定集合中是否存在重复的数。
   (a) 给出一个算法并找出这个算法的时间复杂度的数量级表达式。
   (b) 检查在图灵机上实现该算法是否会影响你的结论。
2. 对于上面习题中的集合，现在要求判定集合中是否存在三个相同的数，请给出算法。并且

判断算法是否是最高效的。

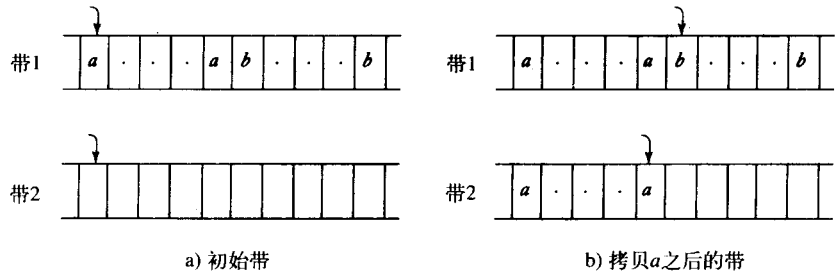
3. 回顾算法的选择是如何影响排序效率的。

14.2 图灵机和复杂性

在第10章中，我们证明了不同类型的图灵机在解决问题的能力上是等价的。这一点使得我们在一个具体的证明过程中可以选择最方便的图灵机，甚至可以选择使用高级计算机语言的程度，从而避免因为使用标准图灵机模型而造成的复杂情况。但是，当我们涉及复杂性这个问题的时候，不同类型图灵机之间的等价性就不再成立了。

例14.2 在例9.4中，我们构造了一个单带图灵机来接受语言$L = \{a^n b^n : n \geqslant 1\}$。通过观察算法，我们得出对于$w = a^n b^n$，图灵机匹配每个$a$和相应的$b$大致需要$2n$步来完成。因此，这个过程需要$O(n^2)$时间。

但是，在后来的例10.1中，我们利用了双带图灵机，并且使用了不同的算法。我们首先将所有的a拷贝到第二个带上，然后利用它们来和第一个带上的b进行匹配。拷贝前后的情况如图14-1所示。拷贝和匹配都能在$O(n)$时间内完成，因此，我们可以看出双带图灵机的时间复杂度为$O(n)$。 □ 346



a) 初始带　　b) 拷贝a之后的带

图 14-1

例14.3 在5.2节和6.3节中，我们讨论了上下文无关语言的成员判定问题。如果我们取输入符号串w的长度作为问题的规模n，那么穷举查找算法的复杂度为$O(n^M)$，这里的M取决于文法。而更有效的CYK算法的时间复杂度为$O(n^3)$这两种算法都是确定型的。

这个问题的非确定型算法是通过对构造w所使用的产生式的顺序进行推测来进行的。如果我们使用的文法没有单元或者λ产生式，那么推导的长度就基本上是$|w|$，这样我们就得到了一个具有时间复杂度$O(n)$的算法。 □

例14.4 我们现在介绍*可满足性问题*（satisfiability problem），这个问题在复杂性理论里扮演着很重要的角色。

布尔（逻辑）常量或者布尔（逻辑）变量都是只能取两个值——“真”或者“假”，我们用“1”和“0”来分别表示它们。布尔运算符将布尔常量和布尔变量组合成布尔表达式。最简单的布尔运算符有“或”（or）运算符，表示为v，它的定义为:

$$0 \vee 1 = 1 \vee 0 = 1 \vee 1 = 1$$

$$0 \vee 0 = 0$$

347

和“与”（and）运算符，表示为∧，它的定义为:

$$0\wedge 0 = 0\wedge 1 = 1\wedge 0 = 0$$
$$1\wedge 1 = 1$$

同样的还有“非”（negation）运算符，也称否定运算符，我们用一条横线来表示，它的定义为:

$$\bar{0} = 1$$
$$\bar{1} = 0$$

我们现在考虑合取范式（conjunction normal form）的布尔表达式。在这种形式中，我们先利用变量$x_1, x_2, \cdots, x_n$来构造如下形式的布尔表达式

$$e = t_i \wedge t_j \wedge \cdots \wedge t_k \tag{14-1}$$

式中的$t_i, t_j, \cdots, t_k$是由变量或其否定经过或运算得到的，即

$$t_i = s_l \vee s_m \vee \cdots \vee s_p \tag{14-2}$$

这里的每一个$s_l, s_m, \cdots, s_p$代表了某个变量或其否定。

可满足性问题是指：给定一个合取范式的表达式e，然后对其中的每个变量$x_1, x_2, \cdots, x_n$进行赋值，那么是否存在某种情况下的赋值，使得表达式e的最后取值为真。举一个具体的例子，我们来看这个式子

$$e_1 = (\bar{x}_1 \vee x_2) \wedge (x_1 \vee x_3)$$

如果我们令$x_1 = 0, x_2 = 1, x_3 = 1$，那么表达式$e_1$就会取真，那么表达式是可满足的。另一方面，表达式

$$e_2 = (x_1 \vee x_2) \wedge \bar{x}_1 \wedge \bar{x}_2$$

是不可满足的，因为无论如何对变量x_1和x_2进行赋值，e_2总是取值为假。

对于可满足性问题很容易发现一个确定的算法。我们可以对变量$x_1, x_2, \cdots, x_n$取所有可能的值，然后计算表达式的值。由于有2^n种不同的情况，这种穷举方法也就相应地具有指数的时间复杂度。

此外，非确定型的算法能够简化问题。如果e是可满足的，我们对于每个x_i的取值进行猜测，然后计算e的值。这是个$O(n)$ 量级的算法。正如在例14.3中所示，我们有一个确定型的穷举查找算法，而这个算法的复杂度是指数量级的。相应地，我们还有一个线性的非确定型的算法。然而，和前一个例子不同，我们这里不知道任何非指数量级的确定型的算法。 □

348

这些例子表明了复杂性问题的讨论是受选定的图灵机的类型影响的，而其中确定型和非确定型的不同是很关键的一点。例14.1表明多带图灵机上的算法和我们使用的具体编程语言的算法是很接近的。因此，我们将使用多带图灵机作为我们研究复杂性问题的模型。

习题

对于本节的所有习题，假设使用的图灵机都是确定型的。

1. 使用双带图灵机来为$\{ww : w \in \{a, b\}^*\}$中的成员构造一个线性时间的算法。如果使用单带图灵机，那么能够得到最好的结果是什么样的？
2. 证明任何在单带、离线图灵机上能够在$O(T(n))$ 时间内完成的算法，在标准图灵机上同样能够在$O(T(n))$ 时间内完成。
3. 证明任何在标准图灵机上能够在$O(T(n))$ 时间内完成的算法，在单向无穷带图灵机上同样能够在$O(T(n))$ 时间内完成。
4. 证明任何在双带图灵机上能够在$O(T(n))$ 时间内完成的算法，在标准图灵机上需要$O(T^2(n))$ 时间来完成。
5. 将布尔表达式 $(x_1 \wedge x_2) \vee x_3$重写为合取范式。
6. 判定下面的表达式

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$$

是否是可满足的。

7. 在例14.2中，我们说第一个算法的时间复杂度为$O(n^2)$，而第二个算法的时间复杂度为$O(n)$。能否更加准确地表述为：对于第一个算法有$T(n) = \Theta(n^2)$，而对于第二个算法有$T(n) = \Theta(n)$？这个对于我们证明例14.2有什么帮助？ 349

14.3 语言族和复杂性类

在关于语言分类的乔姆斯基层次结构中，我们将语言族和自动机类联系起来了，那里，自动机类是根据它的临时存储能力来分类的。另外一种分类的方法是利用某种特定种类的图灵机，同时将时间复杂度作为区分的因素。为此，我们首先来定义一个语言的时间复杂度。

定义14.1 *我们说一个图灵机在时间$T(n)$ 内能够接受语言L，是指对于每个属于L的w，其中$|w| \leq n$，都能在$O(T(n))$次迁移内被图灵机M所接受。如果M是非确定型的，这就意味着对于每个$w \in L$，都至少存在一个迁移长度为$O(T(|w|))$的序列，使得M能够接受这个符号串。*

定义14.2 *如果存在一个确定型的多带图灵机能够在时间$T(n)$ 内接受语言L，那么我们称这个语言是类$DTIME(T(n))$ 的成员。*

如果存在一个非确定型的多带图灵机能够在时间$T(n)$ 内接受语言L，那么我们称这个语言是类$NTIME(T(n))$ 的成员。

这些复杂性类之间的一些关系是明显的，例如：$DTIME(T(n)) \subseteq NTIME(T(n))$和$T_1(n) = O(T_2(n))$ 表明了 $DTIME(T_1(n)) \subseteq DTIME(T_2(n))$。但是从这里开始很快就变得模糊了。我们在这里能得到的是随着$T(n)$ 级数的增加，我们接受了更多的语言。 350

定理14.1 *对于任何整数$k \geq 1$，有*

$$DTIME(n^k) \subset DTIME(n^{k+1})$$

证明：具体的证明请参考Hopcroft和Ullman（1979，p.299）。■

从这个定理，我们可以得出结论：一些能够在n^2时间量级内被接受的语言并没有线性时间的算法；确实也存在一些属于$DTIME(n^3)$ 而不属于$DTIME(n^2)$ 的语言，等等。这样一来，我

们能够给出无穷个嵌套的复杂性类。我们甚至能够得到更多指数时间复杂性类（如果允许的话）。事实上，在这里并没有限制。无论复杂性函数$T(n)$增长得有多迅速，总是有些函数超出了$DTIME(T(n))$的范围。

定理14.2 *不存在一个完全的图灵可计算函数$f(n)$，满足每个递归语言都属于$DTIME(f(n))$。*

证明：给定一个字母表$\Sigma = \{0, 1\}$，其中，Σ^+中的所有符号串都是按照固定顺序$w_1, w_2, \cdots$排列的。同时，我们假设将所有的图灵机也按照固定顺序$M_1, M_2, \cdots$进行排列。

假设现在定理中所说的$f(n)$是存在的。那么我们可以定义语言

$$L = \{w_i : M_i\text{在}f|w_i|\text{步内不接受}w_i\} \tag{14-3}$$

我们将证明L是递归的。为此，对于任意的$w \in L$首先计算$f(|w|)$。由假设f是一个完全的图灵可计算函数可知，这一点是可能的。我们接下来寻找w在序列$w_1, w_2, \cdots$中的位置i。因为整个序列是按照固定顺序排列的，所以，这也是可以做到的。当我们有了i，我们就能够找到M_i，使其在w上执行$f(|w|)$步。这就能够判断w是否属于L，因此，L是递归的。

但是我们现在要证明L是不属于$DTIME(f(n))$的。先假设L属于$DTIME(f(n))$。既然L是递归的，那么就存在某个M_k，满足$L = L(M_k)$。那么w_k是否属于L？如果我们说w_k属于L，那么M_k就会在$f(|w_k|)$步内接受w_k。这是因为$L \in DTIME(f(n))$，而且每个$w \in L$都被M_k在$f(|w|)$步内接受了。但是这与式（14-3）是矛盾的。相反地，如果我们假设$w_k \notin L$也会得到矛盾。这样一个无法解决的问题使我们得出最初的假设，即可计算函数$f(n)$的存在，必然是不成立的。■

定理14.1和定理14.2可以帮助我们得出很多结论，例如存在语言属于$DTIME(n^4)$，但是却不属于$DTIME(n^3)$。尽管这个结论可能比较有理论意义，但是却不清楚它有没有实际意义。这里，我们并不太清楚属于$DTIME(n^4)$的语言的特性是什么。我们可以将复杂性的分类和乔姆斯基层次结构中的语言联系起来，那么就可以更加深入地了解问题。我们下面会通过几个简单的例子来给出更加明确的结论。

351

例14.5 每个正则语言都能够被一个确定型有穷自动机接受，而且接受的时间是和输入的长度成正比的。因此，$L_{REG} \subseteq DTIME(n)$。但是$DTIME(n)$不仅包含了$L_{REG}$。我们在例13.7中就知道，上下文无关的语言$\{a^n b^n : n \geqslant 0\}$可以在$O(n)$时间内被接受。那里给出的证明甚至可以用于更为复杂的语言。□

例14.6 非上下文无关语言$L = \{ww : w \in \{a, b\}^*\}$是属于$NTIME(n)$的。这一点比较直观，我们可以对这个语言中符号串进行识别，使用下面的算法来完成：

1. 将输入从输入文件拷贝到第一个带上，非确定地猜测符号串的中间位置。
2. 将第二部分拷贝到第二个带上。
3. 逐个比较第一个带上的符号和第二个带上的符号。

显然，所有这些步骤都可以在$O(|w|)$时间内完成，因此$L \in NTIME(n)$。

实际上，我们可以证明如果能够构造算法在$O(n)$时间内找到符号串的中间位置，那么$L \in DTIME(n)$。这个寻找中间位置的算法可以这么实现：我们观察第一个带上的每个字符，同时在第二个带上保存个数，但是只是在偶数个符号的时候计数。我们将具体的细节留作习题。□

352

例14.7 从例14.2我们可以得到$L_{CF} \subseteq DTIME(n^3)$和$L_{CF} \subseteq NTIME(n)$。现在，我们来考虑

上下文相关语言族。由于每一步可以使用的产生式是有限的，所以穷举查找算法也是可行的。因此，每个长度为n的符号串都可以在时间n^M内被分解，这里的M是依赖于文法的。注意，尽管如此，我们却不能根据这个得出

$$L_{CS} \subseteq DTIME(n^M)$$

这是因为我们不能给出M的上界。 □

从这些例子中，我们观察得出一个趋势：随着$T(n)$ 的增加，越来越多的语言族，如L_{REG}、L_{CF}、L_{CS}被包含进来。但是乔姆斯基层次结构和复杂性类之间的联系确实是薄弱的、不很清晰的。

习题

1. 完成例14.5中的证明。
2. 证明$L = \{ww^Rw : w \in \{a, b\}^+\}$是属于$DTIME(n)$ 的。
3. 证明$L = \{www : w \in \{a, b\}^+\}$是属于$DTIME(n)$ 的。
4. 证明存在不属于$NTIME(2^n)$ 的语言。

14.4 复杂性类P和NP

既然通过具有不同增长率的时间复杂度来构造有意义的层次结构是没有成效的，那么我们就需要忽略一些次要的因素，例如去掉一些没有意义的区别，如$DTIME(n^k)$ 与$DTIME(n^{k+1})$之间的区别。我们可以证明这种区别，比如说$DTIME(n)$ 和$DTIME(n^2)$ 之间的区别，是非本质性的，因为它可能依赖于具体的图灵机模型（例如图灵机的带的个数），并且我们不知道究竟哪个模型最适合一个真实的计算机。这使得我们得出了著名的复杂性类

$$\mathrm{P} = \bigcup_{i \geq 1} DTIME(n^i)$$

这个类包含了可以被某个确定型图灵机在多项式时间内接受的所有语言，这里没有考虑多项式的幂级。正如我们所看到的，L_{REG}和L_{CF}都属于P。 353

由于确定型复杂度和非确定型复杂度的区别是本质性的，所以我们引入

$$\mathrm{NP} = \bigcup_{i \geq 1} NTIME(n^i)$$

显然 $\mathrm{P} \subseteq \mathrm{NP}$，但是我们不知道这个包含是否是真包含。虽然普遍认为存在属于NP但是不属于P的语言，但是到目前为止，还没有人发现这样一种语言。

对于复杂性类的研究，尤其是对P类问题，是源于对真实和非真实计算的区分。某些计算虽然在理论上是可行的，但是它对资源的要求过高，超过了一般计算机甚至是超级计算机的能力。我们有时称这样的问题为难解的（intractable），以此来表明这种问题虽然原则上是可解的，但是在真实的计算环境下，是没有希望来完成这个算法的。为了更好地理解这类问题，计算机科学家们尝试着为这种难解性构造一个形式化的基础。其中在Cook-Karp论题（cook-karp thesis）中，就曾经尝试着对这个概念进行定义。在Cook-Karp论题中，一个问题如果属

于P，那么就称之为可解的，若不属于P，则称之为难解的。

那么Cook-Karp论题是否能够很好地区分可以实际解决的问题和难解问题呢？答案是不清晰的。显然，任何不属于P的计算具有的时间复杂度的增长速度都要比多项式要快，而它的需求会随着问题规模的增大而非常迅速地增大。即使对于像$2^{0.1n}$这样的函数，取比较大的n，如$n \geq 1000$，其需求都会过高。对于将这种问题称为难解问题，也许我们认为是正当的。但是对于属于$DTIME(n^{100})$的问题呢？虽然Cook-Karp论题认为这样一个问题是可解的，但是即使对很小的n，我们也无法处理。Cook-Karp论题的正确性好像来源于对于一些实际问题的观察，如P类中绝大多数问题属于$DTIME(n)$、$DTIME(n^2)$、$DTIME(n^3)$，而不在此类中的问题往往具有指数复杂度。在实际的问题中，属于P类的问题和不属于P类的问题之间存在着明显的差异。

复杂性类P和NP之间的关系的研究引起了很多科学家的兴趣。这个研究的根本在于问题

$$P = NP$$

是否成立。

这个问题也是计算理论中尚未解决的基础性问题。为了研究这个问题，计算机科学家已经引入了各种相关的概念和问题。其中一个就是NP完全问题。粗略地讲，一个NP完全问题和任何NP问题一样难，从某种意义上讲，它们都是等价的。我们将在后边给出详细解释。

354

定义14.3 *如果存在一个确定型图灵机能够将L_1中字母表上的w_1在多项式时间内转换为L_2中的字母表上的w_2，同时使得$w_1 \in L_1$当且仅当$w_2 \in L_2$。那么我们称语言L_1可以在多项式时间归约（polynomial-time reducible）为语言L_2。*

从这里我们可以得出，如果L_1可以多项式时间归约到L_2，并且如果$L_2 \in P$，那么$L_1 \in P$。同样，如果$L_2 \in NP$，那么$L_1 \in NP$。

定义14.4 *如果语言$L \in NP$，而且每个$L' \in NP$都可以多项式时间归约到L，那么我们称L是NP完全的。*

从定义我们可以很容易地得出，如果某个L_1是NP完全的，而且可以多项式时间归约到L_2，那么L_2也是NP完全的。定义隐含了这样结论：如果我们能够为任意一个NP完全的语言构造一个确定型的多项式时间算法，那么所有的属于NP的语言同时也属于P，即

$$P = NP$$

这使得NP完全性在这个问题的研究中扮演了核心的角色。

例14.8 可满足性问题可以被视为一个语言问题。我们可以将每个实例作为字符串进行编码，使得当且仅当表达式可满足的时候，符号串才被接受。这个问题是NP完全的。可满足性问题是NP完全，这个命题被称为Cook定理。具体的讨论请参阅Hopcroft & Ullman (1979)。 □

355

除了可满足性问题，人们还发现了很多其他的NP完全问题。对于所有这些问题，我们都能够构造指数级的算法，但是对于这些问题，至今没有发现它们中任何一个的多项式时间的算法。这个事实使得人们认为很可能$P \neq NP$。但是，除非能够构造出一个实际的属于NP但是却不属于P的语言，或者证明没有这样的语言存在，否则这个问题始终是悬而未决的。

习题

1. 证明：如果一个语言L_1是NP完全的，并且可以在多项式时间归约到L_2，那么L_2也是NP完全的。

★★2. 参考复杂性理论的相关书籍，编制一个NP完全的列表。

3. P = NP是否有可能是不可判定的？ 356

部分习题的解答和提示

第1章

1.1

5. 要证明两个集合相等，必须证明一个元素属于第一个集合当且仅当它属于第二个集合。假设 $x \in \overline{S_1 \cup S_2}$，则$x \notin S_1 \cup S_2$，意思是$x$不可能在$S_1$或$S_2$中，即 $x \in \bar{S}_1 \cap \bar{S}_2$。相反地，若 $x \in \bar{S}_1 \cap \bar{S}_2$，则$x$不在$S_1$中也不在$S_2$中，即 $x \in \overline{S_1 \cup S_2}$。

6. 可以通过对集合的数目进行归纳来证明。设$Z = S_1 \cup S_2 \cdots \cup S_n$，则$S_1 \cup S_2 \cdots \cup S_n \cup S_{n+1} = Z \cup S_{n+1}$。由标准德摩根定律得

$$\overline{Z \cup S_{n+1}} = \bar{Z} \cap \bar{S}_{n+1}$$

由归纳假设，此关系对前n个集合成立，即

$$\bar{Z} = \bar{S}_1 \cap \bar{S}_2 \cap \cdots \overline{\cap S}_n$$

所以

$$\overline{Z \cup S_{n+1}} = \bar{S}_1 \cap \bar{S}_2 \cap \cdots \overline{\cap S}_n \cap \bar{S}_{n+1}$$

归纳完毕。

8. 假设$S_1 = S_2$，则 $S_1 \cap \bar{S}_2 = \bar{S}_1 \cap S_2 = S_1 \cap \bar{S}_1 = \varnothing$，整个表达式等于空集。现在假设$S_1 \neq S_2$，并设有一个元素$x$在$S_1$中但不在$S_2$中，则 $x \in \bar{S}_2$，于是 $S_1 \cap \bar{S}_2 \neq \varnothing$。于是整个表达式不可能等于空集。

12. 如果x既在S_1中又在S_2中，则x不属于 $(S_1 \cup S_2) - S_2$。因此，充分必要条件是两个集合不相交。

15. (c) 因为

$$\frac{n!}{n^n} = \frac{n}{n}\frac{n-1}{n}\cdots\frac{2}{n}\frac{1}{n}$$

是小于等于1的因子的乘积，所以$n! = O(n^n)$。

27. 用反证法。假设 $2-\sqrt{2}$ 是有理数，则

$$2-\sqrt{2} = \frac{n}{m}$$

即

$$\sqrt{2} = \frac{2m-n}{m}$$

这与 $\sqrt{2}$ 不是有理数的事实相矛盾。

29. 用归纳法。设所有小于n的整数都可写成质数的乘积。如果n是质数，立即得证；如果n不是质数，则它可写成

$$n = n_1 n_2$$

其中两个因子都小于n。由归纳假设，它们都可以写成质数的乘积，于是n也可以写成质数的乘积。

1.2

2. 很多关于符号串的等式都可以用归纳法证明。设对于所有的$u \in \Sigma^*$和长度为n的符号串v，有$(uv)^R = v^R u^R$。现在取一个长度为$n+1$的符号串，比如说$w = va$，则

$$\begin{aligned}(uw)^R &= (uva)^R \\ &= a(uv)^R \text{（由逆的定义）} \\ &= av^R u^R \text{（由归纳假设）} \\ &= w^R u^R\end{aligned}$$

由归纳法，结论对所有符号串都成立。

4. 由于$abaabaaabaa$可以被分解为子串ab，aa，baa，ab，aa，其中每个子串都属于L，所以原符号串属于L^*。类似地，$baaaaabaa$属于L^*。但对于$baaaaabaaaab$不存在可能的分解，因此它不属于L^*。

10. (d) 我们先生成三个a，然后再在任意位置添加任意个a和b。

$$S \to AaAaAaA$$
$$A \to aA|bA|\lambda$$

第一个产生式生成三个a，第二个产生式在任意位置产生任意个a和b。从而上述文法可以产生任意符号串$w \in \{a, b\}^*$，只要$n_a(w) \geqslant 3$。

11.

$$S \Rightarrow aA \Rightarrow abS \Rightarrow abaA \Rightarrow ababS$$

由此可得

$$L(G) = \{(ab)^n : n \geqslant 0\}$$

13. (a) 生成一个b，然后再生成相同数目的a和b，最后可根据需要生成任意多个b。

$$S \to AbA$$
$$A \to aAB|\lambda$$
$$B \to bB|\lambda$$

13. (d) 如果你注意到

$$L_4 = \{a^{m+3} b^m : m \geqslant 0\}$$

问题就比较容易了。

可以很容易得到答案

$$S \to aaaA$$
$$A \to aAb|\lambda$$

14. (b) 我们可以将问题分成两种情况来讨论，从而将问题化简，即$|w| \bmod 3 = 1$和$|w| \bmod 3 = 2$。第一种情况由下式产生

$$S_1 \to aaaS_1|a$$

第二种情况由下式产生

$$S_2 \to aaaS_2|aa$$

上述两种情况可以统一为一个文法

$$S \to S_1|S_2$$

16. (a) 我们可以利用例1.13中的方法和结果。设L_1为例1.13中的语言，修改其语法使得开始符号是S_1。然后考虑符号串$w \in L$。如果w的起始符号是a，则w形如$w = aw_1$，其中$w_1 \in L_1$。这种情况可以用产生式$S \to aS_1$处理。如果w以b开头，则可以由产生式$S \to S_1S$产生。

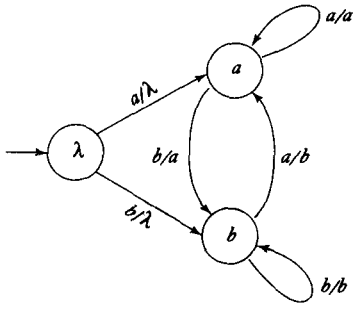
1.3

1.

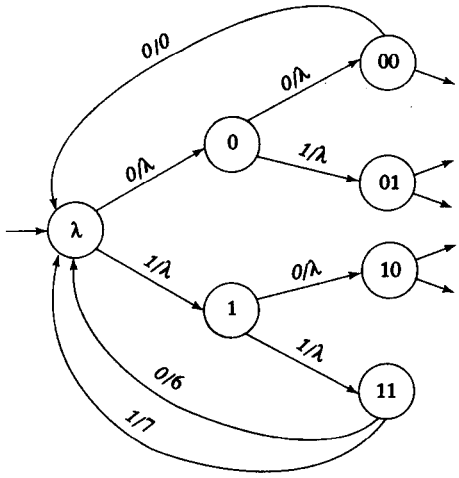
整数→符号　数量
符号→+|-|λ
数量→数字|数字　数量
数字→0|1|2|3|4|5|6|7|8|9

我们可以将上述语法看作理想的C，因为它对整数的长度没有作限制。大多数实际的编译器都会对整数的长度作出限制。

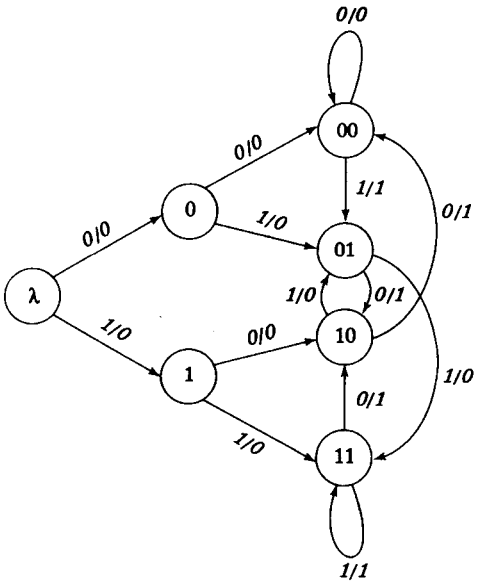
7. 自动机必须在一段时间内记住输入，从而可以在稍后将其重新产生以便输出。我们可以通过用合适的信息来标记状态以达到这种记忆的功能。状态的标记稍后将被产生并用作输出。



10. 我们通过将状态加上标记从而记住输入。当处理完三个位（bit）时，我们产生输出并回到开始以便处理接下来的三个位。下面的解只是部分的，但完整的解是显而易见的。



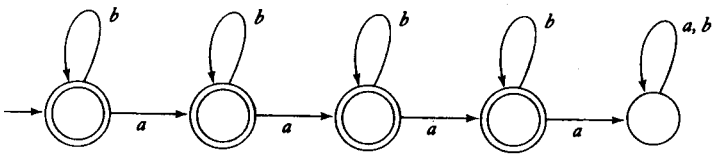
11. 在本题中，传感器必须记住输入的前两个符号并作出相应的转换以记住所需的信息。



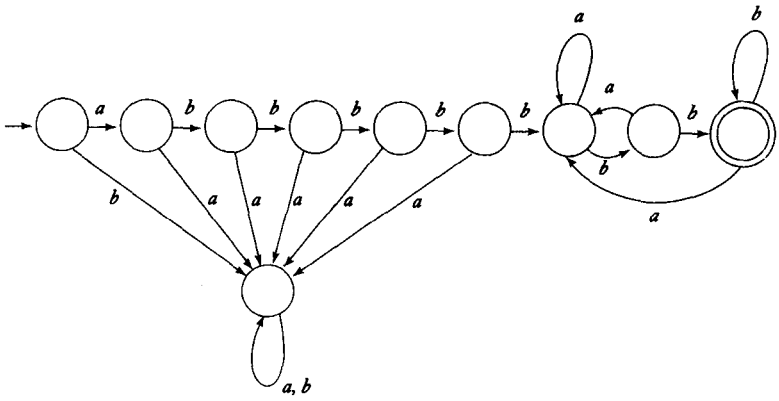
第2章

2.1

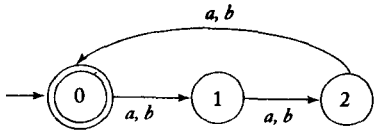
2. (c) 将其分为4种情况，每种包含一个接受状态：没有a的，有一个a的，有两个a的，有三个a的。第四个a会使dfa进入非接受的陷阱状态。下面是一种解：



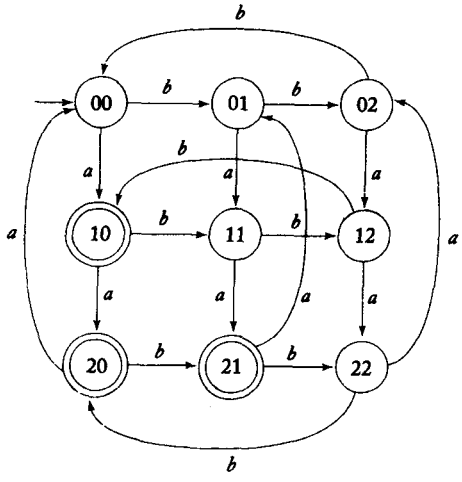
5. (a) 检查前六个符号。如果它们不正确则拒绝。如果前缀正确，则我们检查最后两个输入符号，如果是bb，dfa就进入接受状态。



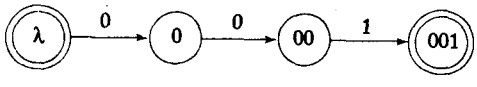
7. (a) 使用标记为$|w|$mod 3的状态，则解是相当简单的。



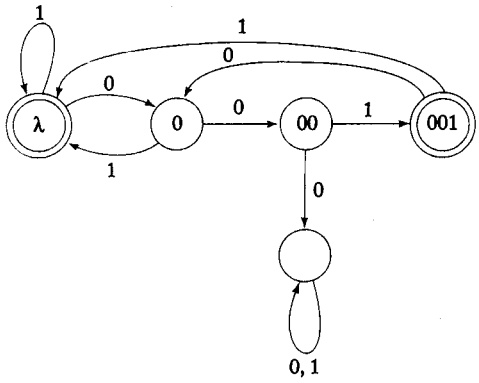
(d) 对于此种情况，我们使用9个状态，每个标记的第一部分是$n_a(w)$mod 3，第二部分是$n_b(w)$mod 3。于是很容易得出转换和终态。



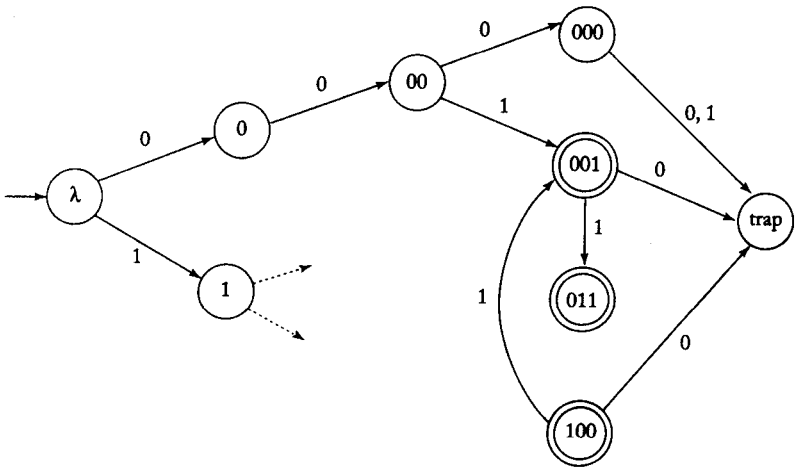
9. (a) 对连续的0的个数进行计数，从而得到dfa的主要部分。



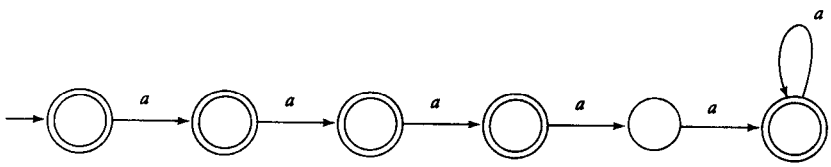
然后加入其他转换以处理连续的0，并为不可接受的符号串设置陷阱。



（d）这里我们需要记住三个位的所有组合情况。这需要8个状态和一个初态。解比较长但并不难。下面给出的是部分解。



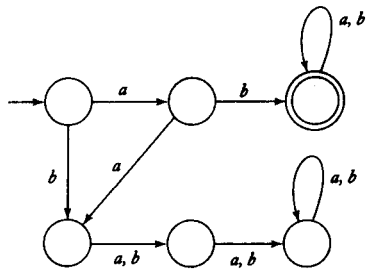
13. 解决这个问题的最简单的方法就是构造一个dfa使其接受语言$L = \{a^n : n = 4\}$，然后计算这个dfa的补。



21. （a）用反证法。设G_M从初态到任何一个终态都没有回路。于是每一个通道都有有限的步数，因而每一个被接受的符号串的长度都一定是有限的。但这就意味着被接受的语言是有限的。

（b）用反证法。设G_M从初态到某个接受状态存在某个回路。于是我们可以利用这个回路产生任意长的、标记有可接受符号串的通道。但一个有限的语言是不能包含任意长的符号串的。

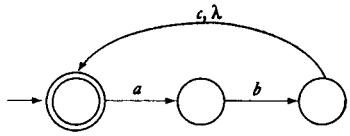
24. 本题有很多不同的解。此处给出其中的一个。



2.2

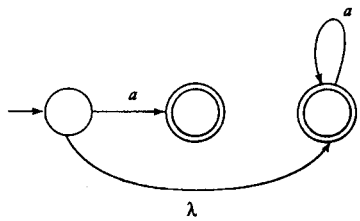
4. $\delta^*(q_0, a) = \{q_0, q_1, q_2\}$, $\delta^*(q_1, \lambda) = \{q_0, q_2\}$。

7. 具有4个状态的解是容易得出的，但是具有三个状态的解稍难一些。下面给出一个三个状态的解:



8. 否。符号串abc有三个不同的符号，于是它不可能被少于三个状态的自动机接受。

15. 对于此类问题，你只需要尝试不同的方法即可找到答案。可能你的大部分尝试都会失败。下面给出一个解。

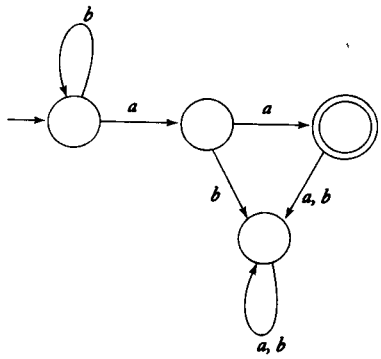


17. 引入一个唯一的开始状态p_0。然后加入一个转移函数

$$\delta(p_0, \lambda) = Q_0$$

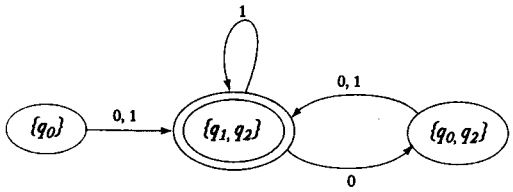
接下来，从Q_0中去掉开始状态。易见，新的nfa与原来的等价。

20. 引入一个非接受的陷阱状态，然后将所有没有定义的转换都连接到这个新状态上。解:



2.3

2. 只需遵照nfa_to_dfa过程进行变换即可。得到的dfa为

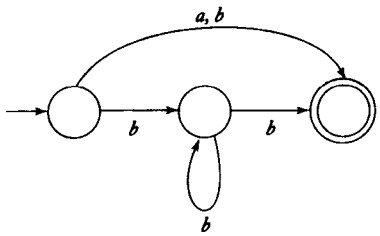


7. 引入一个新的终态p_f，并对于所有$q \in F$，添加一个转移函数

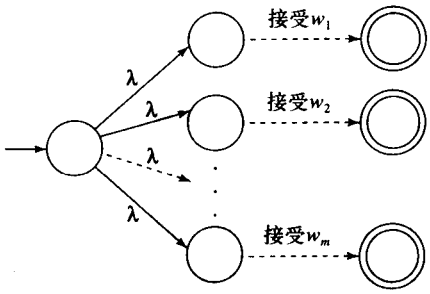
$$\delta(q, \lambda) = \{p_f\}$$

然后将p_f作为唯一的终态。于是很容易证明若原来$\delta^*(q_0, w) \in F$，则修改后为$\delta^*(q_0, w) = \{p_f\}$。因此原nfa与修改后的nfa是等价的。因为这一构造需要使用λ转移，因而不能被用于dfa。一般情况下，dfa不可能只有一个终态，比如接受语言$\{\lambda, a\}$的dfa。

8. 此题有一定难度。下面是一个解

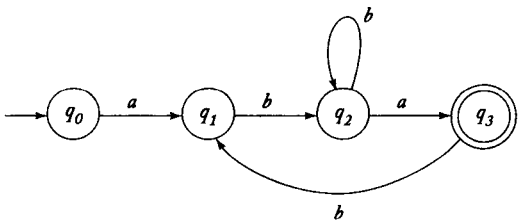


11. 设$L = \{w_1, w_2, \cdots, w_m\}$。则有nfa

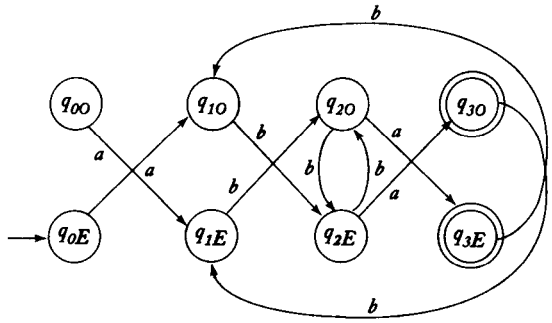


接受语言L，于是此语言是正则的。

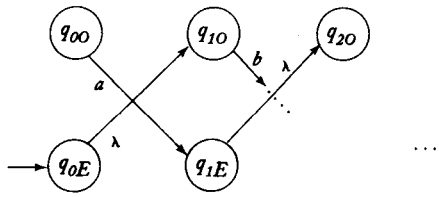
14. 这一点并不显而易见。方法是对接受语言L的dfa进行修改使其记住已经读入的符号的个数是偶数还是奇数。这可以通过将每一个状态变为两个状态，将O或E添加到标记中来解决。例如，如果dfa的局部如下



它的等价变换如下



现在将所有从E状态到O状态的转换均换成λ转移。



通过几个例子，你一定会确信：如果原dfa接受$a_1a_2a_3a_4$，则新的自动机将接受$\lambda a_2\lambda a_4\cdots$，从而接受$even(L)$。

15. 假设有一个dfa接受L。我们

(a) 找出所有从q_0状态读入含两个符号的前缀v能达到的状态的集合$\overline{Q}$，即

$$\overline{Q}=\{q\in Q:\delta^*(q_0,v)=q\}$$

(b) 引入一个新的初态p_0并添加转换

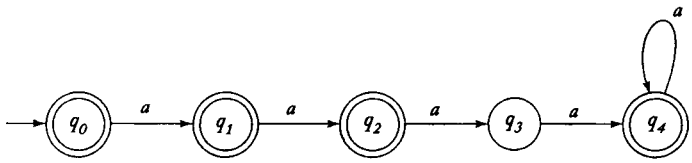
$$\delta(p_0,\lambda)=\overline{Q}$$

不难看出新的nfa能够接受$chop\ 2(L)$。

虽然上述构造是可行的，但完整的答案还应该包括对上面最后一个陈述的证明。

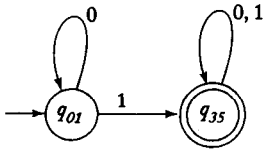
2.4

2. (c)



此dfa是最小的。原因如下：$q_3\notin F$且$q_4\in F$，因此q_3和q_4是可区分的。然后，$\delta^*(q_2, a)\notin F$且$\delta^*(q_4, a)\in F$，因此q_2和q_4是可区分的。类似地，$\delta^*(q_1, aa)\notin F$且$\delta^*(q_3, aa)\in F$，因此$q_1$和$q_3$是可区分的。照此继续下去，可见所有状态都是可区分的，于是此dfa是最小的。

4. 首先，去掉不可达状态q_2和q_4，然后用*mark*过程找到不可区分的状态对（q_0, q_1）和（q_3, q_5），便得到了最小的dfa。



6. 用反证法。假设 $\hat{M}$ 不是最小的。于是我们便可以构造出一个更小的dfa $\tilde{M}$ 使其接受语言 $\bar{L}$。在 $\tilde{M}$ 中，对终态求补使其接受语言L。但这个dfa比M小，与假设M是最小的矛盾。

10. 用反证法。假设q_b和q_c是不可区分的。因为q_a和q_b是不可区分的，并且不可区分性是一种等价关系（习题7），于是q_a和q_c一定是不可区分的。

第3章

3.1

2. 是。因为 $((0+1)(0+1)^*)^*$可以表示任何一个由0和1构成的符号串，而 $(0+1)^*$也是如此。

5. （a）按$m = 0, 1, 2, 3$分成4种情况。产生4个或更多的a，后面按要求产生一定数量的b。解: $aaaaa^*(\lambda + b + bb + bbb)$。

（c）5（a）中语言的补更难找到。如果一个符号串形如a^nb^m（$n<4$或者$m>3$），则它不在L中，但这并不是对 $\bar{L}$ 的完全描述。$\bar{L}$ 中还应包括这样的符号串：其中一个b后面跟着一个a。解:

$$(\lambda + a + aa + aaa)b^* + a^*bbbbb^* + (a+b)^*\,ba(a+b)^*$$

9. 分为三种情况：$m = 1$，$n \geqslant 3$；$n \geqslant 2$，$m \geqslant 2$；$n = 1$，$m \geqslant 3$。每一种情况有一个直接的解。

12. 枚举所有$|v| = 2$的情况，得到

$$aa(a+b)^*\,aa + ab(a+b)^*\,ab + ba(a+b)^*\,ba + bb(a+b)^*bb$$

14. （c）符号串只需要包含每种符号至少一个。下式即可:

$$(a+b+c)^*\,a(a+b+c)^*\,b(a+b+c)^*\,c(a+b+c)^*$$

但此式并不完全，比如其中一定出现的a总是在一定出现的b之前。要给出完全的解，必须枚举三个符号的所有排列。我们可以再增加六个项。答案虽然很长，但概念上并不难。

15. （c）产生两个0，中间填入一些1，然后重复。但不要忘记一个0都没有的情况。解: $(1^*01^*01^*)^* + 1^*$。

16. （a）产生所有长度为3的符号串并重复。一个简短的解是 $((a+b+c)(a+b+c)(a+b+c))^*$。

18. （c）断言

$$(r_1 + r_2)^* \equiv (r_1^*r_2^*)^*$$

是真的。根据给定的规则，$(r_1 + r_2)^*$表示语言 $(L(r_1) \cup L(r_2))^*$，即由$L(r_1)$ 中符号串和$L(r_2)$ 中符号串构成的所有连接。而 $(r_1^*r_2^*)^*$表示 $((L(r_1))^*(L(r_2))^*)^*$，也是由$L(r_1)$ 中符号串和$L(r_2)$ 中符号串构成的所有连接。

21. 一个无穷语言的表达式必须包含至少一个带星号的子表达式，否则它只能表示有限多个符号串。如果有一个带星号的子表达式表示了一个非空符号串，则此子表达式就可以被重复任意多次从而产生任

意长的符号串。

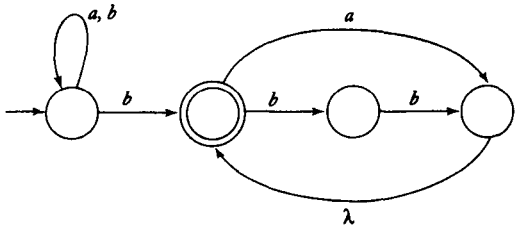
23. 表达式r能产生闭包，当且仅当$n_l(r) = n_r(r)$ 并且$n_u(r) = n_d(r)$。

25. 注意以下几点：位串必须至少有6位长；如果比6位长，它的值至少是64，则一定满足要求；如果正好6位，则要么从左边数第二位（16）是1，要么从右边数第三位（8）是1。如果认识到上述几点，则很容易得到解

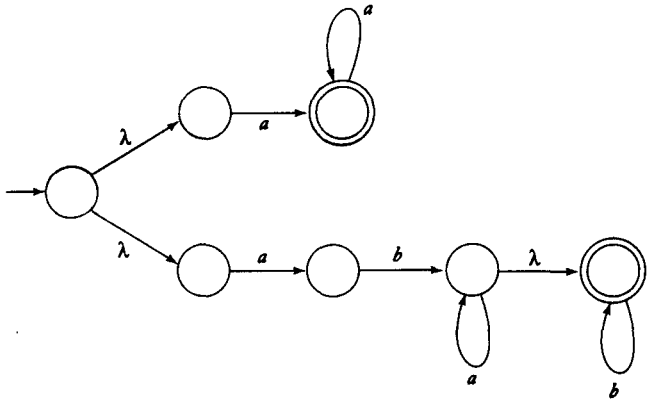
$$(111 + 110 + 101)(0 + 1)(0 + 1)(0 + 1) +$$
$$1(0 + 1)(0 + 1)(0 + 1)(1 + 0)(1 + 0)(1 + 0)(1 + 0)^*$$

3.2

3. 本题可以根据前几个规则解答，而不必用从表达式到dfa的标准构造过程。用后者显然可以解题，但答案要复杂一些。解:

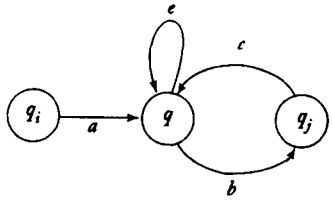


4. (a) 先构造出

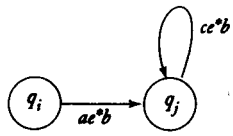


然后应用标准的nfa到dfa的构造算法。

7. 一种情况是

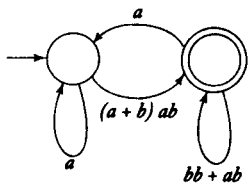


因为从q_j到q_i没有路径，所以我们省去了通常情况下由此路径生成的边。通过查看所有可能的路径，我们得到的结果是



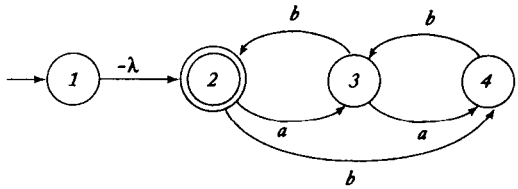
我们对其他情况可以进行类似的分析。

8. 去掉中间的顶点，得到

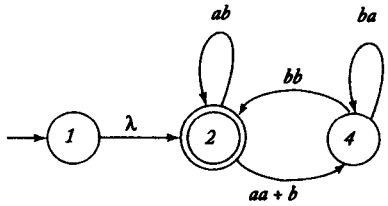


所接受的语言是$L(r)$，其中$r = a^*(a + b)ab(bb + ab + aa^*(a + b)ab)^*$。

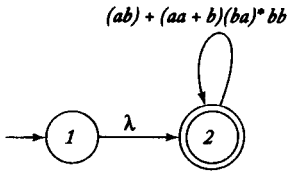
10. (b) 首先我们必须修改nfa使其满足定理3.2中构造过程要求的条件，其中之一是$q_0 \notin F$。这一过程很简单。



然后去掉状态3。

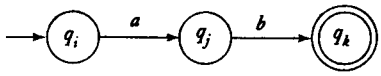


接下来去掉状态4。

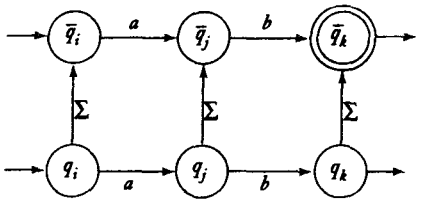


于是正则表达式为$r = (ab + (aa + b)(ba)^*bb^*)^*$。

17. (a) 这是一个难题，除非你能看出其中的窍门。首先构造一个dfa，其状态为$q_0, q_1, \cdots$，并引入一个“平行”自动机，其状态为$\bar{q}_0, \bar{q}_1, \cdots$。然后在原自动机的各个状态与平行自动机的相应状态之间建立转换，使得伪字母可以非确定地从原自动机中的任何一个状态转换到平行自动机的相应状态。例如，如果原dfa的局部如下



则此dfa加上其平行部分得到的新nfa的局部相应为



不难证明，原dfa接受语言L当且仅当新的nfa接受语言insert(L)。

3.3

4. 右线性文法:

$$S \to aaA$$
$$A \to aA|B$$
$$B \to bbbC$$
$$C \to bC|\lambda$$

左线性文法:

$$S \to Abbb$$
$$A \to Ab|B$$
$$B \to aaC$$
$$C \to aC|\lambda$$

7. 我们可以通过归纳证明：如果w是由G生成的句型，则w^R可以被$\hat{G}$在同样步数之内生成。

因为w是由左线性推导构造的，它的形式必为$w = Aw_1$，其中$A \in V$，$w_1 \in T^*$。根据归纳假设可知$w^R = w_1{}^R A$可以由$\hat{G}$生成。现在，如果我们应用$A \to Bv$，则

$$w \Rightarrow Bvw_1$$

但$\hat{G}$包含产生式$A \to v^R B$，因此有

$$\begin{aligned} w^R &\Rightarrow w_1^R v^R B \\ &= (Bvw_1)^R \end{aligned}$$

归纳完毕。

10. 分为两种情况：（i）n和m都是偶数，（ii）n和m都是奇数。答案很容易得出，下面给出情况（i）的产生式

$$S \to aaS|A$$
$$A \to bbA|\lambda$$

12. （a）首先构造一个接受语言L的dfa。这一过程是直接的，得到转换如下

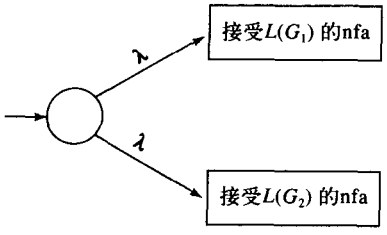
$$\delta(q_0, a) = q_1, \delta(q_0, b) = q_2$$
$$\delta(q_1, a) = q_0, \delta(q_1, b) = q_3$$

$$\delta(q_2, a) = q_3,\ \delta(q_2, b) = q_0$$
$$\delta(q_3, a) = q_2,\ \delta(q_3, b) = q_1$$

其中q_0既是初态也是终态。然后根据定理3.4得答案如下

$$q_0 \to aq_1 | bq_2 | \lambda$$
$$q_1 \to bq_3 | aq_0$$
$$q_2 \to aq_3 | bq_0$$
$$q_3 \to aq_2 | bq_1$$

16. 显然，S_1和S_2都是正则的。我们可以通过构造如下dfa证明它们的并也是正则的



V_1和V_2不相交的条件是必要的，这保证了两个dfa是不同的。

第4章

4.1

2. (a) 构造过程是直接的，但冗长。接受语言$L((a+b)a^*)$ 的dfa是由下列转移函数得到的:

$$\delta(q_0, a) = q_1,\quad \delta(q_0, b) = q_1,\quad \delta(q_1, a) = q_1,\quad \delta(q_1, b) = q_t$$

其中q_t是陷阱状态，q_1是终态。接受语言$L(baa^*)$ 的dfa是由下列转移函数得到的:

$$\delta(p_0, a) = p_t,\ \delta(p_0, b) = p_1,\ \delta(p_1, a) = p_2$$
$$\delta(p_1, b) = p_t,\ \delta(p_2, a) = p_2,\ \delta(p_2, b) = p_t$$

其中p_2是终态。由此我们得到

$$\delta((q_0, p_0), a) = (q_1, p_t),\ \delta((q_0, p_0), b) = (q_1, p_1)$$
$$\delta((q_1, p_1), a) = (q_1, p_2),\ \delta((q_1, p_2), a) = (q_1, p_2)$$

其他转移函数类似。当我们完成上述构造时，会发现唯一的终态是 (q_1, p_2)，且$L((a+b)a^*) \cap L(baa^*) = baa^*$。

7. 注意到

$$nor(L_1, L_2) = \overline{L_1 \cup L_2}$$

于是结果可以由交运算和补运算的封闭性得到。

12. 答案是肯定的。可以由下面的集合等式得到

$$L_2 = ((L_1 \cup L_2) \cap \overline{L_1}) \cup (L_1 \cap L_2)$$

关键要注意到由于L_1是有限的，所以$L_1 \cap L_2$是有限的，于是所有的L_2都是正则的。然后根据已知的并运算和补运算下的封闭性就可以很容易地得到其余答案。

14. 正则语言L在反运算下是封闭的，L^R是正则的。然后可以根据连接运算下的封闭性得到答案。

16. 使用$L_1 = \Sigma^*$。则对于任意的L_2，$L_1 \cup L_2 = \Sigma^*$都是正则的，于是由题述可知所有的L_2都是正则的。

18. 我们可以按如下方式构造：找到一个状态的集合P，满足从初始顶点到P中的任何一个元素都有一条路径，并且从P中任一元素到终态也有一条路径。然后将P中的每一个元素都改为终态。

26. 设$G_1 = (V_1, T, S_1, P_1)$，$G_2 = (V_2, T, S_2, P_2)$，不失一般性地，设V_1和V_2是不相交的。合并两个文法并且

(a) 将S作为新的开始符，并添加产生式$S \to S_1 | S_2$。

(b) 将P_1中所有形如$A \to x$（$A \in V_1$，$x \in T^*$）的产生式都替换为$A \to xS_2$。

(c) 将P_1中所有形如$A \to x$（$A \in V_1$，$x \in T^*$）的产生式都替换为$A \to xS_1$。

4.2

1. 根据例4.1可知，$L_1 - L_2$是正则的，所以存在一个成员资格判定算法。

2. 如果$L_1 \subseteq L_2$，则$L_1 \cap L_2 = L_2$。因为$L_1 \cup L_2$是正则的，并且我们有判断集合相等的算法，也有判断集合之间包含关系的算法。

5. 从接受L的dfa出发，利用定理4.2中的方法构造接受L^R的dfa。然后利用定理4.7中判断等价性的算法。

12. 本题需要一点技巧。如果L不包含长度为偶数的符号串，则

$$L \cap L((aa + ab + ba + bb)^*) = \emptyset$$

左边是正则的，因此我们可以应用定理4.6。

4.3

2. 接受L的dfa要处理中间的符号串v，则在转换图中需要有一条长度为$|v|$的通道。如果$|v|$比dfa的状态数大，则此通道中必包含一个标记为y的回路。但显然，此回路可以被重复任意多次而不改变对输入串的可接受性。

4. (a) 给定m，选$w = a^m b^m a^{2m}$。于是符号串y一定是a^k，被抽取的符号串将是

$$w_i = a^{m+(i-1)k} b^m a^{2m}$$

如果取$i \geqslant 2$，则$m + (i-1)k > m$，则w_i不在L中。

(e) 直接应用泵引理似乎不太容易，因此我们采取间接的方法。假设L是正则的，则根据正则语言在补运算下的封闭性，$\overline{L}$也将是正则的。但$\overline{L} = \{w : n_a(w) = n_b(w)\}$，显然不是正则的。根据反证法，$L$不是正则的。

5. (a) 取p为大于等于m的最小质数，选$w = a^p$。于是y是由a构成的长度为k的符号串，因此

$$w_i = a^{p+(i-1)k}$$

如果取$i - 1 = p$，则$p + (i-1)k = p(k+1)$是合数，w_{p+1}不在此语言中。

8. 此命题是不成立的。举一个反例，设$L_1 = \{a^n b^m : n \leqslant m\}$，$L_2 = \{a^n b^m : n > m\}$，两个都是非正则的。

但$L_1 \cup L_2 = L(a^*b^*)$却是正则的。

9. (a) 此语言是正则的。通过分情况讨论，可以很容易地看到这一点，如$l = 0$，$k = 0$，$n > 5$，在此基础上可以很容易地建立正则表达式。

(b) 此语言不是正则的。如果我们选$w = aaaaaab^m a^m$，我们的对手会有几种选择。如果y只包含a，我们用$i = 0$来破坏$n > 5$的条件。如果对手选择由b构成的y，则我们可以破坏$k < l$的条件。

11. L是正则的。我们可以通过$L = L_1 \cap L_2^R$以及已知的正则语言在相关运算下的封闭性得到此结论。

13. (a) 此语言是正则的，因为任何一个包含两个连续的相同符号的符号串都在此语言中。L的正则表达式是$(a+b)(a+b)^*(aa+bb)(a+b)(a+b)^*$。

(b) 此语言不是正则的。取$w = (ab)^m aa(ba)^m$，此时对手有几种选择，比如$y = (ab)^k$或$y = (ab)^k a$。在第一种情况中

$$w_0 = (ab)^{m-k} aa(ba)^m$$

因为唯一可能与之相等的就是$ww^R = b^l aab^l$，所以w_0不在L中。在第二种情况中，w_0的长度是奇数，因此也不会在L中。

15. 取$L_i = a^i b^i$, $i = 0, 1, \cdots$。对于每一个i，L_i都是有穷的，因此也是正则的。但所有这些语言的并是非正则语言$L = \{a^n b^n : n \geqslant 0\}$。

17. 不是。举一个反例，取语言

$$L_i = \{v_i u v_i^R : |v_i| = i\} \cup \{v_i v_i^R : |v_i| < i\},\ i = 0, 1, 2, \cdots$$

我们断言所有L_i的并是集合$\{ww^R\}$。为了证明这一点，取任意符号串$z = ww^R$，其中$|w| = n$。若$n \geqslant i$，则$z \in \{v_i u v_i^R : |v_i| = i\}$，于是$z$在$L_i$中。如果$n < i$，则$z \in \{v_i v_i^R : |v_i| < i\}$，$i = \{0, 1, 2, \cdots\}$，于是$z$也在$L_i$中。因此，$z$在所有$L_i$的并中。

另一方面，任取L_i中长为m的符号串z，若取i大于m，则z不会在$\{v_i u v_i^R : |v_i| = i\}$中，因为$z$不够长。于是$z$一定在$\{v_i v_i^R : |v_i| < i\}$中，因此$z$的形式必为$ww^R$。

最后我们还必须证明对于每一个i，L_i是正则的。可以根据如下事实得证：对于每一个i，只有有限多个子串v_i。

第5章

5.1

4. 显然，由此文法产生的任何一个符号串都包含相同个数的a和b。为了证明前缀条件$n_a(v) \geqslant n_b(v)$成立，我们对推导过程的长度进行归纳证明。假设对于任何一个由S在n步中产生的句型此条件都成立。要在$n+1$步内得到一个句型，我们可以应用产生式$S \to \lambda$或$S \to SS$。因为这两个产生式都不会改变a和b的个数，也不会改变已经产生的a和b的位置，所以前缀条件依然成立。或者我们使用产生式$S \to aSb$，它增加了一个a和一个b。但因为增加的a是在增加的b的左边，前缀条件也是成立的。因此，若在n步后前缀条件成立，则在$n+1$步后前缀条件依然成立。显然，前缀条件在一步后是成立的，因此我们有了归纳基础，于是归纳成立。

7. (a) 首先解决$n = m + 3$的情况。然后增加更多的b，这可以通过下列产生式完成

$$S \to aaaA$$

$$A \to aAb|B$$
$$B \to Bb|\lambda$$

但这是不完整的，因为它生成了至少三个a。为了解决$n = 0, 1, 2$的情况，我们添加产生式

$$S \to \lambda|aA|aaA$$

(d) 本题有一个出乎意料的简单的解

$$S \to aSbb|aSbbb|\lambda$$

这些产生式对于每一个已经生成的a非确定地产生了bb或bbb。

8. (a) 对于第一种情况，$n = m$，k是任意的，我们可以用下列产生式

$$S_1 = AC$$
$$A \to aAb|\lambda$$
$$C \to Cc|\lambda$$

在第二种情况中，n是任意的，$m < k$，我们使用

$$S_2 = BD$$
$$B \to aB|\lambda$$
$$D \to bDc|E$$
$$E \to Ec|\lambda$$

最后，我们添加产生式$S \to S_1|S_2$。

(e) 将问题分解为两种情况：$n = k + m$和$m = k + n$。用下列产生式处理第一种情况

$$S \to aSc|S_1|\lambda$$
$$S_1 \to aS_1b|\lambda$$

12. (a) 如果S产生L，则$S_1 \to SS$产生L^2。

15. 通常情况下不能由L的文法直接得出 $\bar{L}$ 的文法，因此我们希望能够通过一种递归的方式描述 $\bar{L}$ 。这一点在现在看来还有点难。$\bar{L}$ 的一个显而易见的子集包括长度为奇数的符号串，但这不是全部的。

假设我们有一个长度为偶数的符号串，且不具有形式ww^R。我们从中间开始同时向左和向右对比相应的符号，当中点附近的子串形如ww^R时，我们找到了左边的一个a和右边相应位置上的一个b，或者与此相反。于是此符号串的形式必为$uaww^Rbv$或$ubww^Rav$，其中$|u| = |v|$。一旦我们认识到这一点，我们就可以构造产生此类符号串的文法了。一个解是

$$S \to ASA|B$$
$$A \to a|b$$
$$B \to bCa|aCb$$
$$C \to aCa|bCb|\lambda$$

前两个产生式生成u和v，第三个生成两个不同的符号，最后一个生成中间的回文。

19. 唯一可能的推导始于

$$S \Rightarrow aaB \Rightarrow aaAa \Rightarrow aabBba \Rightarrow aabAaba$$

但这个句型的后缀是aba，因此它不能生成句子$aabbabba$。

22. $E \to E + E|E.E|E^*|(E)|\lambda|\emptyset$。

5.2

2. 一个解是

$$S \to aA,\ A \to aAB|b,\ B \to b$$

注意，另一个较为明显的文法

$$S \to aS_1B$$
$$S_1 \to aS_1B|\lambda$$
$$B \to b$$

不是简单文法。

6. $w = aab$有两个最左推导

$$S \Rightarrow aaB \Rightarrow aab$$
$$S \Rightarrow AB \Rightarrow AaB \Rightarrow aaB \Rightarrow aab$$

9. 我们可以用定理3.4中的方法，从一个接受正则语言的dfa得到一个正则文法。除去$q_f \to \lambda$，此文法是简单文法。但这一规则不会产生任何二义性，因为dfa从不会选择，所以在产生式中也不会有选择。

14. 从下面的推导中可以清楚地看到二义性

$$S \Rightarrow aSb \Rightarrow ab$$
$$S \Rightarrow SS \Rightarrow abS \Rightarrow ab$$

一个等价的非二义性文法是

$$S \to A|\lambda$$
$$A \to aAb|ab|AA$$

此文法的非二义性并不是显而易见的。为了使其可信，先来看两种特殊情况：要产生$w = aabb$，只能从产生式$A \to aAb$开始推导；要产生$w = abab$，只能从产生式$A \to AA$开始推导。更为复杂的符号串也都包含在这两种情况之中，所以产生它们的方式也是唯一的。

20. 解:

$$S \to aA|aAA$$
$$A \to bAb|bb$$

第6章

6.1

3. 用定理6.1对第一个文法中的B进行替换。然后B就变成了无用的，其相关的产生式便可以被删除了。由定理6.1和定理6.2可知，这两个文法是等价的。

8. 唯一的可空变量是A，因此消除λ产生式，得到

$$S \to aA|a|aBB$$
$$A \to aaA|aa$$
$$B \to bC|bbC$$
$$C \to B$$

$C \to B$是唯一的单位产生式，消除它得到

$$S \to aA|a|aBB$$
$$A \to aaA|aa$$
$$B \to bC|bbC$$
$$C \to bC|bbC$$

最后，B和C是无用变量了，所以去除它们可以得到

$$S \to aA|a$$
$$A \to aaA|aa$$

此文法生成的语言是$L((aa)^*a)$。

14. 一个例子是

$$S \to aA$$
$$A \to BB$$
$$B \to aBb|\lambda$$

消除λ产生式以后，我们可以得到

$$S \to aA|a$$
$$A \to BB|B$$
$$B \to aBb|ab$$

16. 显然是这样的。因为删除无用产生式后，文法不会增加任何内容。

21. 文法$S \to aA$; $A \to a$没有无用产生式，也没有单位产生式或是λ产生式。但它并不是最小的，因为$S \to aa$与其等价。

6.2

5. 首先我们必须消除λ产生式，得到

$$S \to AB|B|aB$$
$$A \to aab$$
$$B \to bbA|bb$$

这引入了一个单位产生式，不符合定理6.6构造过程的要求。消除这个单位产生式很简单，得到

$$S \to AB|bbA|aB|bb$$
$$A \to aab$$
$$B \to bbA|bb$$

我们现在可以应用定理6.6的构造过程，得到

$$S\to AB|V_bV_bA|V_aB|V_bV_b$$
$$A\to V_aV_bV_b$$
$$B\to V_bV_bA|V_bV_b$$

和

$$S\to AB|V_cA|V_aB|V_bV_b$$
$$A\to V_dV_b$$
$$B\to V_cA|V_bV_b$$
$$V_c\to V_bV_b$$
$$V_d\to V_aV_b$$
$$V_a\to a$$
$$V_b\to b$$

8. 考虑线性文法中产生式的一般形式

$$A\to a_1a_2\cdots a_nBb_1b_2\cdots b_m$$

然后引入新的变量V_1及产生式

$$V_1\to a_2\cdots a_nBb_1b_2\cdots b_m$$

和

$$A\to a_1V_1$$

继续这一过程，引入V_2和

$$V_2\to a_3\cdots a_nBb_1b_2\cdots b_m$$

依此类推，直到左边不再含有终结符。然后用类似的方法去掉右边的终结符。

9. 此范式可以很容易地由CNF得到。形如$A\to BC$的产生式是被允许的，因为$a=\lambda$是可能的。对于$A\to a$，引入新的变量V_1、V_2及产生式$A\to aV_1V_2$、$V_1\to\lambda$和$V_2\to\lambda$。

12. 解：$S\to aV_b|aS|aV_aS$, $V_a\to a$, $V_b\to b$。

15. 只有$A\to bABC$不具有我们要求的形式，因此我们引入$A\to bAV$和$V\to BC$。后者的形式不正确，但经过对B的替换后，我们有

$$S\to aSA$$
$$A\to bAV$$
$$V\to bC$$
$$C\to aBC$$

6.3

2. 因为aab是例6.11中符号串的前缀，我们可以利用在那里计算出来的V_{ij}。由于$S\in V_{13}$，所以符号串aab在此文法产生的语言中，因此我们可以对其进行语法分析。

为了做语法分析，我们要确定在证明$S\in V_{13}$中用到的产生式:

$$S\in V_{13}，因为S\to AB，其中A\in V_{11}, B\in V_{23}$$
$$A\in V_{11}，因为A\to a$$
$$B\in V_{23}，因为B\to AB，其中A\in V_{22}, B\in V_{33}$$
$$A\in V_{22}，因为A\to a$$
$$B\in V_{33}，因为B\to b$$

上述产生式证明了$S\in V_{13}$，于是我们可以将它们用于语法分析

$$S\Rightarrow AB\Rightarrow aB\Rightarrow aAB\Rightarrow aaB\Rightarrow aab$$

第7章

7.1

2. 证明的关键是从q_0到q_1的非确定性的转换，此转换不需要在符号串的中间发生。当然，如果转换在其他位置发生或者输入的符号串的形式不是ww^R，自动机就不能进入接受格局。假设转换发生时栈的内容是$x_1x_2\cdots x_kz$，自动机若要接受一个符号串则必须进入格局（q_1, λ, z）。我们通过检查转移函数知道，我们可以进入上述格局当且仅当符号串的未读部分形如$x_1x_2\cdots x_k$，即原输入形如ww^R，且此转换正好发生在符号串的中间。

4.（a）可以这样得到解：每读入一个a向栈中放两个标志，每读入一个b则从栈中去掉一个标志。解:

$$\delta(q_0, \lambda, z) = \{(q_f, z)\}$$
$$\delta(q_0, a, z) = \{(q_1, 11z)\}$$
$$\delta(q_1, a, 1) = \{(q_1, 111)\}$$
$$\delta(q_1, b, 1) = \{(q_1, \lambda)\}$$
$$\delta(q_1, \lambda, z) = \{(q_f, z)\}$$

（f）此处我们利用非确定性及下述函数来生成一个、两个或三个符号

$$\delta(q_0, a, z) = \{(q_1, 1z), (q_1, 11z), (q_1, 111z)\}$$
$$\delta(q_1, a, 1) = \{(q_1, 11), (q_1, 111), (q_1, 1111)\}$$

解的其余部分基本上与4（a）类似。

9. 这是一个没有用栈的pda，所以从功能上讲，它等同于一个有穷接受器。于是由此pda可以直接得到状态转换

$$\delta(q_0, a) = q_1$$
$$\delta(q_0, b) = q_0$$
$$\delta(q_1, a) = q_1$$
$$\delta(q_1, b) = q_0$$

11. 跟踪此过程，每次选择一条路径。从q_0到q_2的路径上可以只包含一个a。其他路径还可以包括一

个a后面跟随一个或多个b并以a结尾。这些是全部的（仅有的）选择。所以此pda接受语言

$$L = \{a\} \cup L(abb^*a)$$

14. 我们没有足够的状态用于跟踪a的序列到b的序列的转换。为了解决这个问题，我们在栈中放一个符号以记录我们在符号串序列中所处的位置。例如，一个解是:

$$\delta(q_0, a, z) = \{(q_0, 1)\}$$
$$\delta(q_0, a, 1) = \{(q_0, 1)\}$$
$$\delta(q_0, b, 1) = \{(q_0, 2)\}$$
$$\delta(q_0, a, 2) = \{(q_0, 2)\}$$
$$\delta(q_0, \lambda, 2) = \{(q_f, 2)\}$$

我们只有两个状态，初态q_0和接受状态q_f。因此，我们一般不再需要用不同的状态进行跟踪，而只需要用栈中的一个符号。

16. 此处我们用内部状态记录即将被放入栈中的符号。例如，

$$\delta(q_i, a, b) = \{(q_j, cde)\}$$

被

$$\delta(q_i, a, b) = \{(q_{jc}, de)\}$$
$$\delta(q_{jc}, \lambda, d) = \{(q_j, cd)\}$$

取代。由于δ只能有有限个元素，且每个元素只能向栈中添加有限的信息，因而这一构造过程可以被任何pda完成。

7.2

3. 可以仿照定理7.1中的构造过程，或者注意到此语言正是$\{a^{n+2}b^{2n+1} : n \geq 0\}$。通过后者我们得到如下解

$$\delta(q_0, a, z) = \{(q_1, z)\}$$
$$\delta(q_1, a, z) = \{(q_2, z)\}$$
$$\delta(q_2, a, z) = \{(q_2, 11z)\}$$
$$\delta(q_2, a, 1) = \{(q_2, 111)\}$$
$$\delta(q_2, b, 1) = \{(q_3, 1)\}$$
$$\delta(q_3, b, 1) = \{(q_3, \lambda)\}$$
$$\delta(q_3, \lambda, z) = \{(q_f, z)\}$$

其中q_0是初态，q_f是终态。

4. 首先将文法转换为格里巴克范式，得到$S \to aSSS$; $S \to aB$; $B \to b$。然后按照定理7.1中的构造过程进行，得到

$$\delta(q_0, \lambda, z) = \{(q_1, Sz)\}$$
$$\delta(q_1, a, S) = \{(q_1, SSS), (q_1, B)\}$$

$$\delta(q_1, b, B) = \{(q_1, \lambda)\}$$
$$\delta(q_1, \lambda, z) = \{(q_f, z)\}$$

7. 由定理7.2，任给一个npda，我们都能构造一个等价的上下文无关文法。然后根据定理7.1，我们可以由此文法构造一个具有三状态的npda。由等价关系的传递性，初始的npda和最后的npda也是等价的。

9. 首先我们得到L的格里巴克范式，例如$S \to aSB|b$, $B \to b$。然后，我们应用定理7.1中的构造过程得到一个具有三个状态q_0、q_1和q_f的npda。我们可以通过用一个特殊的栈符号标记状态q_1从而将q_1省去。一个完整的解是

$$\delta(q_0, \lambda, z) = \{(q_0, Sz_1)\}$$
$$\delta(q_0, a, S) = \{(q_0, SB)\}$$
$$\delta(q_0, b, S) = \{(q_0, \lambda)\}$$
$$\delta(q_0, b, B) = \{(q_0, \lambda)\}$$
$$\delta(q_0, \lambda, z_1) = \{(q_f, \lambda)\}$$

11. 开始必须至少有一个a。在此之后，$\delta(q_0, a, A) = \{(q_0, A)\}$只是简单地读入$a$而不改变栈的内容。最后，当遇到第一个$b$时，pda进入状态$q_1$，而从此状态只能通过$\lambda$转换进入终态。因此，一个符号串可以被接受当且仅当它包含一个或多个a并且以一个b结尾。

7.3

4. 乍一看，此语言似乎是非确定型的，因为前缀a可以连接两种不同的后缀。然而，此语言确实是确定型的，因为我们能够由其构造一个dpda。当输入的第一个符号是a时，此dpda进入终态。如果有更多的输入符号，则离开此状态并接受a^nb^n。完整的解:

$$\delta(q_0, a, z) = \{(q_3, 1z)\}$$
$$\delta(q_3, a, 1) = \{(q_1, 11)\}$$
$$\delta(q_1, a, 1) = \{(q_1, 11)\}$$
$$\delta(q_1, b, 1) = \{(q_1, \lambda)\}$$
$$\delta(q_1, \lambda, z) = \{(q_2, z)\}$$

这里$F = \{q_2, q_3\}$。

9. 可以很直接地得到解。将a和b压入栈中。符号c标志着从保存到匹配的转换，因此所有过程都可以确定性地进行。

11. 有两个状态: 初始的、非接受状态q_0及终态q_1。pda会保持在状态q_1上直到栈顶为z。当栈顶为z时，pda将转换到状态q_0。其余部分基本上与例7.3相同。于是我们有$\delta(q_0, a, z) = \{(q_1, 0z)\}$, $\delta(q_1, a, 0) = \{(q_1, 00)\}$等等，其中$\delta(q_1, \lambda, z) = \{(q_0, z)\}$。当你写出全部上述产生式时，你将会认识到此pda是确定型的。

15. 这是很明显的。因为任何一个正则语言都可以被一个dfa所接受，并且每一个这样的dfa都是一个没有用到栈的dpda。

16. 此处的基本思想是将一个dpda与一个dfa结合到定理4.1的构造过程中，而对栈的处理与此自动机接受语言L_1时相同。不难看出，结果是一个dpda。

7.4

2. 考虑符号串$aabb$和$aabbbbaa$。在前者中，推导必须由$S \Rightarrow aSB$开始；而在后者中，推导必须由$S \Rightarrow SS$开始。但如果我们仅看前4个符号，我们不能确定是哪一种情况。于是此文法不在$LL(4)$中。因为对于任意长的符号串，我们都可以构造类似上面的例子，所以对于任意的k，此文法都不在$LL(k)$中。

4. 观察前三个符号。如果它们是aaa、aab或aba，则此符号串只能在$L(a^*ba)$中。如果前三个符号是abb，则任何这样的可分析的符号串都一定在$L(abbb^*)$中。对于每一种情况，我们都可以找到一个LL文法，并以一种明显的方式结合二者。一个解是

$$S \to S_1 | S_2$$
$$S_1 \to aS_1 | ba$$
$$S_2 \to abbB$$
$$B \to bB | \lambda$$

观察前三个字符，我们就可以知道$S \Rightarrow S_1$或者$S \Rightarrow S_2$是否是必需的。所以此文法是$LL(3)$。

7. 对于一个确定型的CFL，存在一个dpda。当把此dpda转换为一个文法时，此文法是非二义性的。

9. (a)

$$S \to aSc | S_1 | \lambda$$
$$S_1 \to bS_1c | \lambda$$

这基本上是一个简单文法。只要当前被扫描的符号是a，我们就必须应用$S \to aSc$；如果是b，就必须应用$S \to S_1$；如果是c，则只能用$S \to \lambda$。此文法是$LL(1)$。

第8章

8.1

3. 取$w = a^m b^m b^m a^m a^m b^m$。此时对手有几种选择可以考虑。例如，若$v = a^k$，$y = a^l$，且$v$和$y$位于前缀$a^m$中，则

$$w_0 = a^{m-k-l} b^m b^m a^m a^m b^m$$

不在L中。还存在许多其他可能的选择，但在所有的情况下，我们都可以从该语言中抽取出这个符号串。

7. (a) 用泵引理。给定m，取$w = a^{m^2} b^m$。我们需要仔细考虑的v和y的选择只是$v = a^k, y = b^l$，其中k与l都不为0。设$l = 1$，然后选$i = 2$，于是w_2有$m^2 + k$个a和$m + 1$个b。但是

$$(m+1)^2 = m^2 + 2m + 1$$
$$> m^2 + k$$

因为w_2不在该语言中，所以该语言不是上下文无关的。可类似地证明$l > 1$的情况。

(f) 给定m，取$w = a^m b^{m+1} c^{m+2}$，则可以很容易地从该语言中抽取出w。

8. (b) 该语言不是上下文无关的。用泵引理，$w = a^m b^m a^m b^m$，并检查v和y的不同选择。

10. 可能出乎你的意料，该语言是上下文无关的。构造一个npda，使其计数到某个值k（通过将k个符号放入栈中），并记住第k个符号。然后它在w_2中检查第k个符号。如果该符号不同于记住的符号，则

接受该符号串。若$w \in L$，则必定会有某个k使得此种情况发生。该npda非确定性地选择上述k。

12. 使用线性语言的泵引理。对于给定的m，选取$w = a^m b^{2m} a^m$。现在v和y全部是由a构成的，因此可以很容易地从该语言中抽取出w。

15. 该语言不是线性的。用泵引理，选

$$w = (\cdots(a)\cdots) + (\cdots(a)\cdots)$$

其中（…(与)…）分别表示m个左括号或右括号。如果$|u| \geq 1$，则我们可以很容易地抽取，使得对于某个前缀v，$n((v) < n)(v)$是一个不正确的表达式。可以类似地证明其他的分解方式。

20. 使用$w = a^{pq}$，其中p和q是质数且$p > m$，$q > m$。如果$|vy| = k$，则

$$|w_{i+1}| = pq + ik$$

如果我们选$i = pq$，则

$$w_{i+1} = a^{pq(1+k)}$$

不在该语言中。

8.2

1. 补语言是上下文无关的。补语言包括两种情况：$n_a(w) \neq n_b(w)$ 和$n_a(w) \neq n_c(w)$ 。这些又可以被分解为$n_a(w) > n_b(w)$，$n_a(w) > n_c(w)$，$n_a(w) < n_b(w)$和$n_a(w) < n_c(w)$。其中每一个都是上下文无关的，这可以通过构造一个CFG得到证明。因此，完整的语言是上述四种情况的并，根据并运算的封闭性，该语言是上下文无关的。

5. 给定一个上下文无关文法G，通过将每一个产生式$A \to x$替换为$A \to x^R$构造一个上下文无关文法$\hat{G}$。对推导的步骤数进行归纳，我们可以证明，若w是G的一个句型，则w^R是$\hat{G}$的一个句型。

9. 给定两个线性文法$G_1 = (V_1, T, S_1, P_1)$ 和$G_2 = (V_2, T, S_2, P_2)$，且$V_1 \cap V_2 = \varnothing$，构造二者合并后的文法$\hat{G} = (V_1 \cup V_2, T, S, P_1 \cup P_2 \cup S \to S_1 \mid S_2)$。则$\hat{G}$是线性的且$L(\hat{G}) = L(G_1) \cup L(G_2)$。
为了证明线性语言在连接运算下是不封闭的，选取线性语言$L = \{a^n b^n : n \geq 1\}$，则可以通过泵引理证明语言$L^2$不是线性的。

13. 设$G_1 = (V_1, T, S_1, P_1)$ 是L_1的一个线性文法，$G_2 = (V_2, T, S_2, P_2)$ 是L_2的一个左线性文法。通过把G_2中形如$V \to x$（$x \in T^*$）的产生式替换为$V \to S_1 x$来构造一个文法$\hat{G}_2$。合并文法G_1和$\hat{G}_2$，选取S_2为开始符。可以很容易地证明，在该文法中

$$S_2 \Rightarrow S_1 w \Rightarrow uw$$

当且仅当$u \in L_1$且$w \in L_2$。

15. 语言$L_1 = \{a^n b^n c^m\}$和$L_2 = \{a^n b^m c^m\}$都是非二义性的。但它们的交甚至不是上下文无关的。

21. $\lambda \in L(G)$ 当且仅当S是可空的。

第9章

9.1

2. 检查整个输入的一个具有三个状态的解是

$$\delta(q_0, a) = (q_1, a, R)$$
$$\delta(q_1, a) = \delta(q_1, b) = (q_1, a, R)$$
$$\delta(q_1, \square) = (q_2, \square, R)$$

其中$F = \{q_2\}$。

还可以通过只检查输入的第一个符号而忽略其余输入符号来得到一个具有两个状态的解，例如

$$\delta(q_0, a) = (q_2, a, R)$$

7. (a)

$$\delta(q_0, a) = (q_1, a, R)$$
$$\delta(q_1, b) = (q_2, b, R)$$
$$\delta(q_2, a) = (q_2, a, R)$$
$$\delta(q_2, b) = (q_3, b, R)$$

其中$F = \{q_3\}$。

(b)

$$\delta(q_0, a) = \delta(q_0, b) = (q_1, \square, R)$$
$$\delta(q_0, \square) = (q_2, \square, R)$$
$$\delta(q_1, a) = \delta(q_1, b) = (q_0, \square, R)$$

其中$F = \{q_2\}$。

10. 在概念上，本题的解是简单的，但要写出细节不免有些冗长。大致结构如下:

(i) 在每一个符号串的末尾放置一个标记符号c。

(ii) 将左边的两个符号的组合ca替换为ac，将右边的两个符号的组合ac替换为ca。重复此过程，直到两个c在符号串的中间相遇。

(iii) 去掉一个c并移动符号串的其余部分将空位补上。

显然，这是一个很长的过程，但它却是图灵机经常以一种笨拙的方式做一些简单的工作这一事实的典型情况。

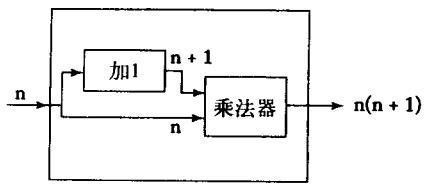
12. 我们不能只从一个方向查找，因为我们不知道何时能停下来。我们必须以一种往返的方式工作，将标志置于查找区间的右边界和左边界，并向外移动标记。

19. 如果终态集F包含的元素多于一个，则对于所有的$q \in F$和$a \in \Gamma$，可以引入一个新的终态q_f及转换

$$\delta(q, a) = (q_f, a, R)$$

9.2

3. (a) 我们可以将该机器视为由两个主要部分构成的: 一个只对输入加1的加1器; 一个将两个数相乘的乘法器。示意性地它们以一种简单的方式结合。



5. (c) 首先，将输入分解为两个相等的部分。这可以仿照9.1节的习题10进行。然后按顺序比较一个部分中的一个符号和另一个部分中与其对应的符号，直到发现不同的两个符号。

8. 一个解:

$$\delta(q_0, a) = (q_i, a, R)$$
$$\delta(q_0, c) = (q_0, c, R) \text{ 对所有的} c \in \Sigma - \{a\}$$
$$\delta(q_0, \square) = (q_j, \square, R)$$

状态q_0是可以应用*searchright*指令的任何一个状态。

9.3

2. 我们忽略了到目前为止定义的图灵机是确定型的，而pda可以是非确定型的这一事实。因此，我们还不能断言图灵机比下推自动机更强大。

第10章

10.1

4. (a) 该自动机有一个转移函数

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$$

其中限制: 对于所有的转换$\delta(q_i, a) = (q_j, b, L \text{ or } R)$，条件$a = b$必须成立。

(b) 为了模拟标准图灵机的转换$\delta(q_i, a) = (q_j, b, L)$，其中$a \neq b$，对于所有的$c \in \Gamma$，我们引入新的转换$\delta(q_i, a) = (q_{jL}, b, S)$和$\delta(q_{jL}, b) = (q_j, b, L)$，等等。

6. 我们引入一个伪空白符（pseudo-blank）B。每当原机器要写□时，新的机器就写B。因此，对于每一个$\delta(q_i, \square) = (q_j, b, L)$，我们增加$\delta(q_i, B) = (q_j, b, L)$，等等。当然，我们应该保留原来的转换 $\delta(q_i, \square) = (q_j, b, L)$以处理带上原有的空白符。

9. 这不会限制自动机的能力。对于每一个符号$a \in \Gamma$，我们引入一个伪符号（pseudo-symbol），比如说A。每当我们需要保留a时，我们先写下A，然后回到该单元，将A替换为a。

11. 对于所有的$d \in \Gamma - \{a, b\}$，我们将

$$\delta(q_i, \{a, b\}) = (q_j, c, R)$$

替换为

$$\delta(q_i, d) = (q_j, c, R)$$

10.2

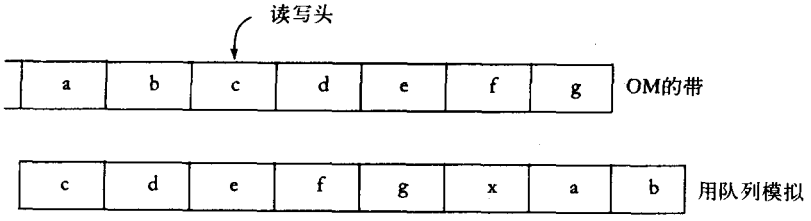
1. 为了给出形式化的定义，我们使用$\Gamma_T = \Gamma \times \Gamma \times \cdots \times \Gamma$和$\delta : Q \times \Gamma_T \rightarrow Q \times \Gamma_T \times \{L, R\}^m$，这里$m$是读写头的个数。一个需要考虑的问题是当两个读写头位于同一个单元时会发生什么情况。形式化的定义必须给出可能发生的冲突的解决方案。

要用标准图灵机（SM）模拟原图灵机（OM），我们让SM具有$m + 1$个道。在一个道上我们保存OM带上的内容，而我们用其余的m个道记录OM的带头的位置。

	□	a	b	c	d	□ OM的带内容
	□		x			□ 带读写头#1的位置
	□				x	□ 带读写头#2的位置
						⋮

SM通过扫描并更新它的活动区来模拟OM的每一个迁移。

4. 本练习说明了队列自动机与标准图灵机是等价的，因此，队列是比栈更强大的存储设备。要用队列自动机模拟标准图灵机，我们可以将OM的右侧保持在队列的前面，将左侧保持在队列的后面。



对右移的模拟很简单，我们只需去掉队列的首符号并将相应符号置于队列尾部。而对左移的模拟是相对复杂的，我们需要让队列的内容循环移动几次以使得所有符号都在正确的位置上。我们可以借助于另外的标记Y和Z来表示边界。例如，为了模拟

$$\delta(q_i, c) = (q_j, z, L)$$

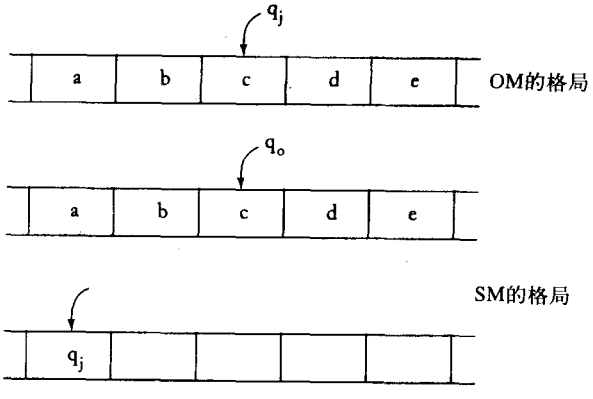
我们执行下列步骤:

(i) 去掉队列首符号c，将zY添加到队列尾部。

(ii) 循环队列中的内容直到得到$bzYdefgXa$。

(iii) 将Z添加至队列尾部，然后循环，当Y和Z移至队列首部时将其去掉。

8. 我们只需两个带，一个用于存储OM的带内容，另一个用于存储OM的状态。



SM只需要两个状态: 一个接受状态和一个非接受状态。

10.3

3. (i) 从输入的左侧开始。通过将机器置于相应的状态来记住符号。然后将其替换为X。

(ii)将读写头右移，并停在（非确定地）输入的中间。

(iii)比较该位置上的符号与记住的符号。如果匹配，则在该单元上写Y。如果不匹配，则拒绝输入。

(iv)由于输入的中间用Y做了标记，我们就可以确定地进行下面的步骤：交替地左移和右移，比较相应的符号。

要得到一个完全确定型的解，我们首先要找到输入的中点（比如在两个端点置标记，然后将其移向中间，直至相遇）。

6. 非确定性地为n选择一个值。判断输入的长度是否是n的整数倍。如果是，则接受。若$a^n \in L$，则存在某个n使上述过程有效。

10.4

3. 大致算法如下：

(i) 首先，拷贝前面的符号串。

(ii) 找到最右边的0，将其改为1。然后将这个1右边的1全都改为0。

(iii) 如果没有0了，则将所有的1改为0，并在左侧添加一个1。

(iv) 转到 (i)。

8. 设$S_1 = \{s_1, s_2, \cdots\}$，$S_2 = \{t_1, t_2, \cdots\}$，则它们的并可以由下式枚举

$$S_1 \cup S_2 = \{s_1, t_1, s_2, t_2, \cdots\}$$

如果某个$s_i = t_j$，则我们只列出其中一个。因此，上述两个集合的并是可数的。对于$S_1 \times S_2$，使用图10-17中的排序。

10.5

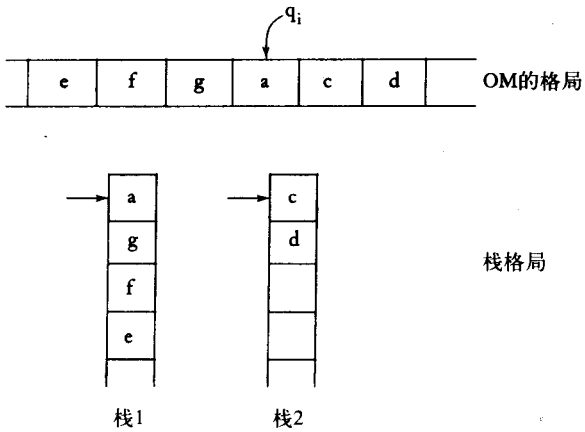
2. 首先，将输入除以2，并将结果移至带的一个部分上，这部分空白空间最初是由输入占据的。此部分空间可以被用于存储后续的除数。

4. （e）使用如下图所示的3-道自动机。在第三道上，我们保留$|w|$的当前试验值。在第二道上，我们每隔$|w|$个单元放置一个分隔符。然后我们比较标记之间的单元内容。

	a	b	c	d	b	c	d		输入
			x			x			分隔符
	1	1	1						$\|w\|$的试验值

6. 用6.2节的练习16找到一个Ⅱ型（two-standard form）文法。然后使用定理7.1中的构造过程。由此得到的pda每一次迁移读入一个输入符号，并且每次增加的栈的内容都不超过一个符号。

7. 例：



栈1包含OM读写头下面的符号及其左侧的所有符号。栈2包含读写头右侧的所有信息。对OM读写头的左移和右移的模拟都很简单。例如，要模拟$\delta(q_i, a) = (q_j, b, L)$，可以将$a$从栈1弹出，并将$b$压入栈2。

第11章

11.1

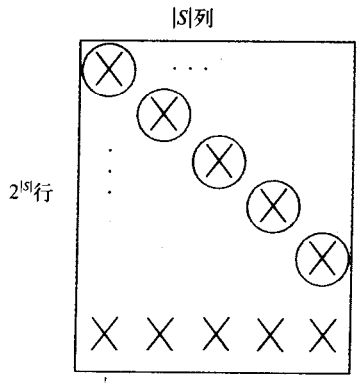
2. 我们知道两个可数集合的并还是可数的，所有递归可枚举语言的集合是可数的。如果所有非递归可枚举的语言的集合也是可数的，那么所有语言的集合就是可数的。但我们知道，事实并非如此。

6. 设L_1和L_2是两个递归可枚举语言，M_1和M_2分别是接受这两个语言的图灵机。当用输入w表示时，我们非确定地选择M_1或M_2来处理w。结果是一个接受$L_1 \cup L_2$的图灵机。

11. 因为上下文无关语言是递归的，所以由定理11.4可知，该语言的补也是递归的。但应该注意，该补语言并不一定是上下文无关的。

14. 对于任意给定的$w \in L^+$，考虑w的所有拆分$w = w_1 w_2 \cdots w_m$。对于每一种拆分，确定是否有$w_i \in L$。因为对于每一个w，只存在有限个拆分方式，因而我们能够确定是否有$w \in L^+$。

18. 试图通过对角线化证明2^S是不可数的（S是有限集合）之所以失败，是因为图11-2中的表不是正方形的，它有$|2^S|$行和$|S|$列。



当我们对角化时，对角线上的结果可能出现在下面的某一行上。

11.2

1. 看一个典型的推导:

$$S \overset{*}{\Rightarrow} aS_1bB \Rightarrow aaS_1bbB \overset{*}{\Rightarrow} a^nS_1b^nB \Rightarrow a^{n+1}b^{n-1}B \Rightarrow a^{n+1}b^{n+1}B \Rightarrow \cdots$$

由此不难推测出该文法生成语言为

$$L = \{a^{n+1}b^{n+k}, n \geqslant 1, k = -1, 1, 3, \cdots\}$$

3. 形式上，该文法可以被描述为$G = (V, S, T, P)$，其中$S \subseteq (V \cup T)^+$，且

$$L(G) = \{x \in T^* : s \Rightarrow_G x \text{ 对于任意的 } s \in S\}$$

定义11.3中的无限制文法与此扩展文法是等价的，因为任给一个无限制文法，对于所有$s_i \in S$，我们总可以添加开始规则$S_0 \to s_i$。

7. 为了得到无限制文法的这种形式，每当$|u| > |v|$时，都在右侧插入哑变量（dummy variable）。例如:

$$AB \to C$$

可被替换为

$$AB \to CD$$
$$D \to \lambda$$

对于等价性的证明是简单的。

11.3

1. (c) 处理上下文相关文法并不总是容易的。例11.2中介绍的消息传递者的想法通常是有用的。在此问题中，第一步是要产生句型a^nBc^nD。变量B和D将起到标记和消息传递者的作用，以保证在合适的位置上生成正确数目的b和d。第一部分用如下产生式很容易得到

$$S \to aAcD | aBcD$$
$$A \to aAc | aBc$$

下一步，将B向右移动以便和D相遇，所用产生式为

$$Bc \to cB$$
$$Bb \to bB$$

当B和D相遇时，我们便可以生成一个d和一个返回的消息传递者，该消息传递者将在正确的位置上生成一个b并停止于该位置。

$$BD \to Ed$$
$$cE \to Ec$$
$$bE \to Eb$$
$$aE \to ab$$

另外，我们也可以生成一个d和一个标记D，并用一个不同的消息传递者生成一个b，如此继续下去：

$$BD \rightarrow FDd$$
$$cF \rightarrow Fc$$
$$bF \rightarrow Fb$$
$$aF \rightarrow abB$$

4. 最简单的证明来自于lba。假设一个语言是上下文相关的，则存在一个lba M能够接受该语言。给定w，我们先将其重写为w^R，然后再将M作用于它。因为$L^R = \{w : w^R \in L\}$，所以M接受w^R当且仅当$w \in L^R$。将符号串重写并将M应用于其上的自动机是一个lba。因此L^R是上下文相关的。

6. 我们可以通过lba来证明。显然存在一个lba可以识别任何形如wuw的符号串。只需从另一端开始比较相应的符号直到得到一个匹配的符号串。因为存在一个lba，所以该语言是上下文相关的且一定存在一个上下文相关文法。

第12章

12.1

3. 给定M和w，修改M从而得到 $\hat{M}$， $\hat{M}$停机当且仅当写下一个特殊符号（比如一个新引入的符号#）。要实现这一点，我们可以修改M的停机格局使得每次写下#后便停止。于是，M停机就意味着 $\hat{M}$写下了#，而 $\hat{M}$ 写下了#就意味着M停机了。因此，如果有一个算法可以告诉我们自动机是否写下了一个特殊符号a，我们就可以令$a = \#$并将其作用于 $\hat{M}$ 。这就解决了停机问题。

7. 给定（M, w），将M修改为 $\hat{M}$ ，使得（M, w）停机当且仅当 $\hat{M}$ 接受了某个简单语言，比如$\{a\}$。为了达到此目的，M可以首先检查输入并记住其是否是a。然后M执行其正常的计算。当它停机时，检查输入是否是a。如果是则接受，否则就拒绝。因此 $\hat{M}$ 接受$\{a\}$当且仅当M停机。现在构造一个接受a的简单的图灵机M_1。如果我们有一个算法能够检查两个语言的等价性，我们就可以用它判断是否$L(\hat{M}) = L(M_1)$。如果$L(\hat{M}) = L(M_1)$，则（M, w）停机。如果$L(\hat{M}) \neq L(M_1)$，则（M, w）不停机，于是我们就得到了停机问题的解决方案。

10. 给定（M, w），我们修改M使其在格局$q_f w$下总是停机。如果给定的问题是可判定的，我们就可以将假定的算法用于修改后的具有格局$q_0 w$和$q_f w$的机器。这样我们就得到了停机问题的一个解决方案。

13. 取一台通用图灵机并用它模拟空白带上的计算。一旦被模拟的计算停机，通用图灵机就接受正在被模拟的图灵机。因而此通用图灵机是一个接受器，它接受所有在空白带上停机的图灵机。于是该集合是递归可枚举的。

现在假设该集合是递归的，则存在一个算法A，该算法可以按程序长度的递增顺序列举出所有在空白带上停机的图灵机。检查原图灵机是否在由A列举出的图灵机之中。因为原程序的长度是固定的，所以当被检查的图灵机的程序长度超过该长度时，比较将会停止。于是我们就有了一个空白带停机问题的解决方案。

16. 设该问题的实例是$p_1, p_2, \cdots, p_n$，我们构造一个图灵机使其按如下方式运行：

如果 问题 $= p_1$ 则返回false

如果 问题 $= p_2$ 则返回true

⋮

如果 问题 = p_n 则返回true

无论这些实例的真值如何，总会有某个图灵机能给出正确的答案。记住，我们无须知道该图灵机实际是什么样子，只需保证它存在即可。

12.2

3. 假设我们有一个算法能够判定是否 $L(M_1) \subseteq L(M_2)$。则我们可以构造一个图灵机M_2，满足$L(M_2) = \emptyset$，然后应用此算法。于是 $L(M_1) \subseteq L(M_2)$ 当且仅当$L(M_1) = \emptyset$。但这与定理12.3矛盾，因为我们可以由任给的文法G构造M_1。

6. 如果我们取$L(G_2) = \Sigma^*$，则该问题就变成了定理12.3的问题，因而是不可判定的。

8. 因为存在一些文法使得$L(G) = L(G)^*$，且存在另一些文法使得$L(G) \neq L(G)^*$，所以由Rice定理可知该问题是不可判定的。而要由基本原理得到此结果会比较难。取停机问题（M, w）并修改之（如定理12.4所述），使得如果（M, w）停机则 $\hat{M}$ 接受$\{a\}^*$，如果（M, w）不停机则 $\hat{M}$ 接受$\emptyset$。使用我们借以得到定理11.7的构造方法来由 $\hat{M}$ 构造文法 $\hat{G}$。如果 $L(\hat{M}) = \{a\}^*$，则 $L(\hat{G}) = L(\hat{G})^* = \{a\}^*$。但如果 $L(\hat{M}) = \emptyset$，则 $L(\hat{G}) = \emptyset$ 且 $L(\hat{G})^* = \{\lambda\}$。因此，如果此问题是可判定的，我们就可以得到停机问题的一个解决方案。

12.3

1. 一个PC-solution是$w_3w_4w_1 = v_3v_4v_1$。没有MPC-solution，因为一个符号串的前缀是001，另一个符号串的前缀是01。

3. 如果对于所有的i，都有$|w_i| > |v_i|$或者都有$|w_i| < |v_i|$，那么显然无解。如果此条件不成立，则要么对于某个i有$|w_i| = |v_i|$，该情况下的解是显然的，要么存在一个j和一个k，满足$|w_j| > |v_j|$且$|w_k| < |v_k|$。在后一种情况中，波斯特对应问题有一个解 $w_j^r w_k^s$，这里$r = |v_k| - |w_k|$，$s = |w_j| - |v_j|$。

5. （a）该问题是不可判定的。如果它是可判定的，我们就会有一个判定原修改过的波斯特对应问题（MPC-problem）的算法。给定$w_1, w_2, \cdots, w_n$，我们构造 $w_1^R, w_2^R, \cdots, w_n^R$并使用假定算法。由于$w_1w_i\cdots w_k = (w_k^R \cdots w_i^R w_1^R)^R$，所以原MPC-problem有解当且仅当新的MPC-problem有解。

第13章

13.1

2. 使用例13.3中的函数*subtr*，我们得到解

$$greater(x, y) = subtr(1, subtr\ (1, subtr\ (x, y)))$$

7.

$$g(x, y) = mult(x, g(x, y-1))$$
$$g(x, 0) = 1$$

9. （a）

$$\begin{aligned} A(1, y) &= A(0, A(1, y-1)) \\ &= A(1, y-1) + 1 \end{aligned}$$

$$
\begin{aligned}
&= A(1, y-2) + 2 \\
&\vdots \\
&= A(1,0) + y \\
&= y + 2
\end{aligned}
$$

（b）有了（a）的结果，我们可以用归纳法证明下一个等式。假设对于$y = 1, 2, \cdots, n-1$，我们有$A(2,y) = 2y + 3$。则

$$
\begin{aligned}
A(2, n) &= A(1, A(2, n-1)) \\
&= A(1, 2n+1) \\
&= 2n + 3, \quad \text{由(a)}
\end{aligned}
$$

因为

$$
\begin{aligned}
A(2, 0) &= A(1, 1) \\
&= 3
\end{aligned}
$$

于是我们有了归纳基础，因而该等式对所有的y均成立。

15. 如果$2^x + y - 3 = 0$，则$y = 3 - 2^x$。唯一能给出具有正值的y的x的值是0和1，因此μ的值域是$\{0, 1\}$，当$y = 1$时取最小值。因此

$$\mu y(2^x + y - 3) = 1$$

13.2

1. （b）使用$C_T = \{a, b, c\}$, $C_N = \{x\}$以及$A = \{x\}$。非终结符x被用作目标符号串的左边界和右边界，两个w由下式同时产生

$$V_1xV_2 \rightarrow V_1axV_2a | V_1bxV_2b | V_1cxV_2c$$

最后，x由下式去掉

$$V_1xV_2 \rightarrow V_1V_2$$

3. 在每一步唯一可能的对V_1的识别是用整个生成的符号串。这导致了符号串数目的加倍，且

$$L = \{a^{2n} : n \geqslant 1\}$$

5. 一个解是

$$
\begin{aligned}
V_1 * V_2 = V_3 &\rightarrow V_1 1 * V_2 = V_3V_2 \\
V_1 * V_2 = V_3 &\rightarrow V_1 * V_2 1 = V_3V_1
\end{aligned}
$$

例如

$$1*1 = 1 \Rightarrow 11*1 = 11 \Rightarrow 11*11 = 1111$$

等等。

13.3

1.

$$P_1: S \to S_1S_2$$
$$P_2: S_1 \to aS_1, S_2 \to aS_2$$
$$P_3: S_1 \to bS_1, S_2 \to bS_2$$
$$P_4: S_1 \to \lambda, S_2 \to \lambda$$

5. 回忆我们对于上下文相关文法使用的消息传递者的方法，就可以得出本题的解。

$$ab \to x$$
$$xb \to bx$$
$$xc \to \lambda$$

8. 虽然此结果不是显而易见的，但它却是解决习题7的一个方法。任取一个符号串，比如a^{255}，该符号串可以通过对a^{127}使用1次产生式$a \to aaa$以及126次产生式$a \to aa$而得到。而a^{127}也可以通过类似的方式由a^{63}得到，等等。因此$L(aa^*)$中的每一个符号串都可以被生成。

参考文献

A. V. Aho and J. D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling.* Vol. 1, Englewood Cliffs, N.J.: Prentice Hall.

P. J. Denning, J. B. Dennis, and J. E. Qualitz. 1978. *Machines, Languages, and Computation.* Englewood Cliffs, N.J.: Prentice Hall.

M. A. Harrison. 1978. *Introduction to Formal Language Theory.* Reading, Mass.: Addison-Wesley.

J. E. Hopcroft and J. D. Ullman. 1979. *Introduction to Automata Theory, Languages and Computation.* Reading, Mass.: Addison-Wesley.

R. Hunter. 1981. *The Design and Construction of Compilers.* Chichester, New York: John Wiley.

R. Johnsonbaugh. 1996. *Discrete Mathematics.* Fourth Ed. New York: Macmillan.

Z. Kovahi. 1978. *Switching and Finite Automata Theory.* Second Edition. New York: McGraw-Hill.

A. Salomaa. 1973. *Formal Languages.* New York: Academic Press.

A. Salomaa. 1985. "Computations and Automata," in *Encyclopedia of Mathematics and Its Applications.* Cambridge: Cambridge University Press.

索　引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

A

accepter（接受器），26
Ackerman's function（Ackerman函数），330
algorithm（算法），2，246
alphabet（字母表），15
ambiguity（二义性），136
　of a grammar（文法的），141
　inherent（固有的），144
automaton（自动机），2，25
　deterministic（确定型的），26
　nondeterministic（非确定型的），26
axioms（公理），324，334

B

Backus-Naur form（巴克斯-诺尔形式），146
base of a cycle（回路的始点），8
blank（空白符），223
blank-tape halting problem（空白带停机问题），305

C

Cartesian product of sets（集合的笛卡儿积），5
child-parent relation in a tree（树中的父子关系），8
Chomsky hierarchy（乔姆斯基层次结构），295
Chomsky normal form（乔姆斯基范式），149，165
Church's thesis（丘奇论题），325
Church-Turing thesis（丘奇-图灵论题），325
closure（闭包），99
　positive（正的），18
　star（星），18
closure properties（封闭性质）
　of context-free languages（上下文无关语言的），213
　of regular languages（正则语言的），100
complement（补）
　of a set（集合的），3
　of a language（语言的），18
complete systems（完整系统），324
complexity（复杂性），343
　of a grammar（文法的），163
　space（空间），344
　time（时间），344
complexity class P（复杂性类P），353
complexity class NP（复杂性类NP），354
composition（组合），326
computability（可计算性），299
computable function（可计算函数），233
computation（计算），228
　models（模型），323
　valid（有效），321
concatenation（连接）
　of languages（语言的），18
　of strings（符号串的），15
configuration of an automaton（自动机的格局），25
conjunctive normal form（合取范式），348
consistent systems（一致系统），324
context-free grammars（上下文无关文法），126
context-free languages（上下文无关语言），125
　deterministic（确定型的），195
context-sensitive grammars（上下文相关文法），289
context-sensitive languages（上下文相关语言），290
control unit of an automaton（自动机的控制部件），25
Cook-Karp thesis（Cook-Karp论题），354
Cook's theorem（Cook定理），355
cycle in a graph（图中的回路），8
　simple（简单的），8
CYK algorithm（CYK算法），172

D

dead configuration（死格局），52
decidability（可判定性），299
DeMorgan's laws（德摩根定律），4
dependency graph（依赖图），154

derivation（推导），21
leftmost（最左的），129
rightmost（最右的），129
derivation tree（推导树），130
partial（部分），131
yield（果），131
deterministic finite accepter（确定型有穷接受器），36
dfa（确定型有穷接受器），36
diagonalization（对角线化），279
disjoint sets（不相交的集合），4
distinguishable sets in a dfa（确定型有穷接受器的可区分的集合），63
dpda（确定型下推自动机），195

E

empty set（空集），4
end markers for an lba（lba的端标记），271
enumeration procedure（枚举过程），268
equivalence（等价性），7
of automata classes（自动机类的），250
of dfa's and nfa's（dfa和nfa的），55
of grammars（文法的），24

F

family of languages（语言族），42
final state（终态），36
finite automata（有穷自动机），35
formal languages（形式语言），2
functions（函数），5
computable（可计算的），233
domain（定义域），5
partial（部分的），5
range（值域），5
total（全），5

G

grammar（文法），19
context-free（上下文无关的），126
context-sensitive（上下文相关的），289
left-linear（左线性的），89
linear（线性的），91
regular（正则的），89
right-linear（右线性的），89
simple（简单的），140
unrestricted（无限制的），283
graph（图），7
labeled（带标记的），7
Greibach normal form（格里巴克范式），149，168

H

halting problems for Turing machines（图灵机的停机问题），301
halt state of a Turing machine（图灵机的停机状态），224
hierarchy of language families（语言族的层次结构），275
homomorphic image of a language（语言的同态映像），103
homomorphism（同态），103

I

incompleteness theorem（不完整性定理），324
indistinguishable states in a dfa（dfa中的不可区分状态），66
inherent ambiguity（固有二义性），144
initial state（初态），36
input file（输入文件），25
instantaneous description（瞬时描述）
of a pushdown automaton（下推自动机的），179
of a Turing machine（图灵机的），226
internal states of an automaton（自动机的内部状态），25，36
intractable problems（难解问题），354

L

lambda-productions（lambda产生式），156
language（语言），15，17
accepted by a dfa（dfa接受的），38
accepted by a dpda（dpda接受的），196
accepted by an lba（lba接受的），271
accepted by an nfa（nfa接受的），51
accepted by a Turing machine（图灵机接受的），229
associated with regular expressions（正则表达式相应的），73
generated by a grammar（文法生成的），21
generated by a Post system（波斯特系统生成的），335

language families（语言族），42
lba（线性有界自动机），270
left-linear grammar（左线性文法），89
leftmost derivation（最左推导），129
linear bounded automata（线性有界自动机），270
linear grammar（线性文法），91
LL-grammars（LL文法），201
L-systems（L-系统），340

M

Markov algorithm（马尔科夫算法），339
matrix grammar（矩阵文法），338
membership algorithm（成员资格判定算法），111
  for context-free languages（上下文无关语言的），172
  for context-sensitive languages（上下文相关语言的），293
minimal dfa（最小dfa），67
minimalization operator（最小化运算符），331
monus（真减），327
move of an automaton（自动机的迁移），25
MPC-solution（修改过的波斯特对应解），313
mu-recursive functions（mu-递归函数），331

N

nfa（非确定型有穷接受器），48
non-contracting grammars（不可收缩文法），290
nondeterministic finite accepter（非确定型有穷接受器），47
nondeterminism（非确定性），52
nonterminal constant（非终结常量），334
normal form of a grammar（文法的范式），149，165
NP-complete problems（NP完全问题），355
npda（非确定型下推自动机），177
null set（空集），4

O

order（序）
  proper（良），269
  relation in a tree（树中的关系），8

P

parsing（分析），136
  exhaustive search（穷举搜索），136
  top-down（自顶向下的），136
path（路径）
  in a graph（图中的），8
  labeled（带标记的），8
  simple（简单的），8
pattern matching（模式匹配），85
PC-solution（波斯特对应解），313
pda（下推自动机），175
phrase-structure grammar（短语结构文法），338
pigeonhole principle（鸽巢原理），114
polynomial-time reduction（多项式时间归约），355
Post correspondence problem（波斯特对应问题），312
  modified（修改过的），313
Post system（波斯特系统），334
powerset（幂集），4
primitive recursion（原始递归），326
primitive recursive functions（原始递归函数），328
primitive regular expressions（基本正则表达式），72
productions of a grammar（文法的产生式），19
program of a Turing machine（图灵机的程序），224
projector function（投影函数），326
proof techniques（证明方法），9
  contradiction（反证法），11
  induction（归纳法），9
proper order（良序），269
proper subset（真子集），4
pumping lemma（泵引理）
  for context-free languages（上下文无关语言的），206
  for linear languages（线性语言的），210
  for regular languages（正则语言的），115
pushdown automata（下推自动机），175
  deterministic（确定型的），195
  nondeterministic（非确定型的），176

R

read-write head of a Turing machine（图灵机的读写头），222
recursive function（递归函数），325
recursive language（递归语言），277
recursively enumerable languages（递归可枚举的语言），276
reduction of states in a dfa（dfa的状态化简），62
reduction（归约）
  of undecidable problems（不可判定问题的），304

polynomial-time（多项式时间），355
regular expressions（正则表达式），71
regular grammar（正则文法），89
regular language（正则语言），43
relation（关系），5
reverse（逆）
of a language（语言的），18
of a string（符号串的），15
rewriting systems（重写系统），337
Rice's theorem（Rice定理），311
right-linear grammars（右线性文法），89
rightmost derivation（最右推导），129
right quotient of languages（语言的右商），104
root of a tree（树的根），8

S

satisfiability problem（可满足性问题），347
semantics of programming languages（程序设计语言的语义），148
sentence（句子），17
sentential form（句型），21
set（集合），3
countable（可数的），267
uncountable（不可数的），267
set operations（集合运算），3
s-grammar（简单文法），140
simulation（模拟），251
space-complexity（空间复杂度），344
stack（栈），175
alphabet（字母表），177
start symbol（开始符），177
standard representation for regular languages（正则语言的标准表示），112
state-entry problem（状态进入问题），304
storage of an automaton（自动机的存储），25
string（符号串），15
empty（空的），15
length（长度），15
operations（操作），15
prefix（前缀），16
suffix（后缀），16
subset（子集），4
proper（真），4
substring（子串），16
successor function（后继函数），326
symmetric difference of two sets（两个集合的对称差），109
syntax of a programming language（程序设计语言的语法），147

T

tape alphabet（带字母表），223
tape of a Turing machine（图灵机的带），222
terminal constant（终结常量），334
terminal symbol（终结符），19
time-complexity（时间复杂度），344
tracks on a tape（带上的道），253
tractable problems（可解问题），343，354
transducer（转换器），26
transition function（转移函数），25，36
extended（扩展的），37
transition graph（转换图），36
generalized（通用的），81
trap state（陷阱状态），39
trees（树），8
Turing-computable function（图灵可计算函数），233
Turing machine（图灵机），221
multidimensional（多维的），261
with multiple tracks（多道的），253
multitape（多带），258
nondeterministic（非确定型的），263
off-line（离线），255
with semi-infinite tape（半无穷带），253
standard（标准），226
with stay-option（带不动选择的），251
universal（通用的），266
Turing's thesis（图灵论题），244

U

undecidable problem（不可判定问题），300
for context-free languages（上下文无关语言的），318
for recursively enumerable languages（递归可枚举语言的），308
unit productions（单位产生式），158
universal set（全集），4
universal Turing machine（通用图灵机），266
unrestricted grammar（无限制文法），283
useless productions（无用产生式），153

V

variable（变量）
of a grammar（文法的），19
nullable（可空的），156
start（开始的），19
useless（无用的），153

W

walk in a graph（图中的通道），8

Y

yield of a derivation tree（推导树的果），131

Z

zero function（零函数），326