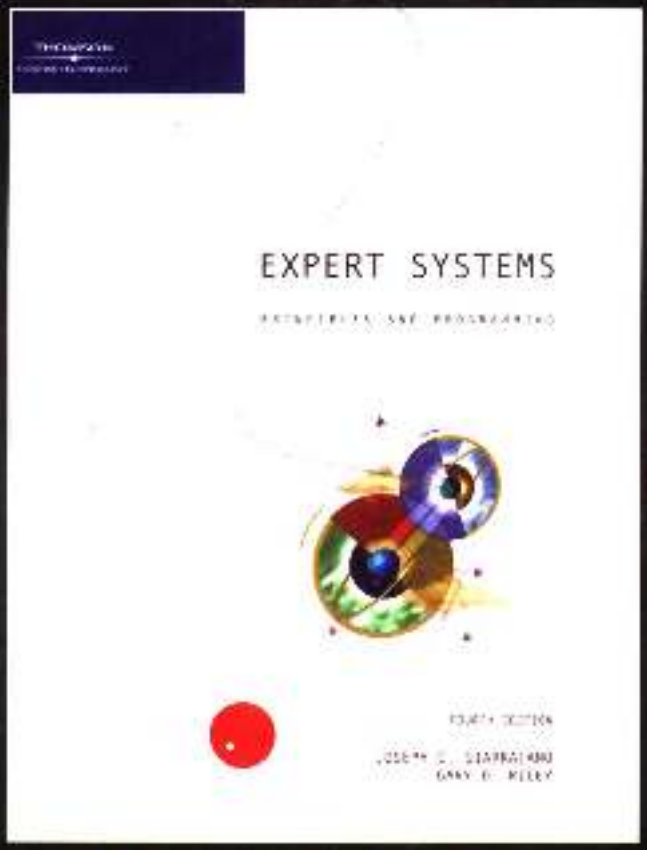
计 算 机 科 学 丛 书

原书第4版

专家系统

原理与编程

(美) Joseph C. Giarratano Gary D. Riley 著 印鉴 陈忆群 刘星成 译



Expert Systems

Principles and Programming

Fourth Edition

机械工业出版社
China Machine Press

THOMSON

专家系统 原理与编程 （原书第4版）

本书是人工智能领域里的著名教科书和参考书，详细介绍专家系统的基本原理与编程技术。本版在融合了前几个版本的理论知识和实际应用的基础上进行了改进。本书分两部分，第一部分介绍专家系统的基本理论，并对人工智能及其与专家系统的关系做了总体论述。第二部分集中介绍应用技术，包括CLIPS专家系统工具和新的面向对象语言COOL，读者将学习如何应用COOL语言通过定义规则和对象，开发一个完整的专家系统。

本书特点

- 覆盖面广，包括专家系统的理论知识和基于规则的应用。
- 内容全面更新，反映了快速发展的专家系统领域的最新趋势。
- 书中每一章的最后都设计了有针对性的习题，帮助读者加强对知识的理解。

作者简介

Joseph C. Giarratano 美国休斯敦大学明湖分校计算机科学系教授。作为NASA的顾问，他参与了专家系统工具CLIPS（包含在本书中）的开发。另外，他发表了30多篇研究论文并著有10多本书。

Gary D. Riley 于1984年在美国得克萨斯A&M大学获计算机科学硕士学位。他在NASA工作了11年，并为CLIPS专家系统语言开发了基于规则的特性部分。目前他居住在得克萨斯州，任职于PeopleSoft有限公司。

www.thomsonlearningasia.com



附赠光盘包括CLIPS程序、源代码以及其他相关文档。

ISBN 7-111-19203-6

9 787111 192039

HZ BOOKS 华章图书

上架指导：计算机/人工智能

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 7-111-19203-6

定价：65.00 元（附光盘）

TP18
760-2

计 算 机 科 丛 书

原书第4版

专家系统
原理与编程

(美) Joseph C. Giarratano Gary D. Riley 著 印鉴 陈忆群 刘星成 译



Expert Systems
Principles and Programming
Fourth Edition

机械工业出版社
China Machine Press

本书是一本关于专家系统的著名教科书，全面介绍了专家系统原理，并通过CLIPS详细讨论了其实际应用。本书内容包括：知识表示、推理方法、不确定性推理、不精确推理、CLIPS、高级模式匹配、模块化设计、执行控制和规则效率、过程化程序设计、类、实例和消息处理程序等。

本书理论与实际相结合，内容由浅入深，为了解和设计专家系统提供了理论基础和编程指导。随书光盘包括CLIPS程序、源代码以及其他相关文档。

本书适合作为计算机科学相关专业本科生和研究生的教材，也可供相关专业人员参考。

Joseph C. Giarratano, Gary D. Riley: Expert Systems: Principles and Programming, Fourth Edition.
ISBN: 0-534-38447-1

Copyright © 2005 by Course Technology, a division of Thomson Learning, Inc.

Original language published by Thomson Learning (a division of Thomson Learning Asia Pte Ltd). All rights reserved.

China Machine Press is authorized by Thomson Learning to publish and distribute exclusively this simplified Chinese edition. This edition is authorized for sale in the People's Republic of China only (excluding Hong Kong, Macao SAR and Taiwan). Unauthorized export of this edition is a violation of the Copyright Act. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

本书原版由汤姆森学习出版集团出版。版权所有，盗印必究。

本书中文简体字翻译版由汤姆森学习出版集团授权机械工业出版社独家出版发行。此版本仅限在中华人民共和国境内(不包括中国香港、澳门特别行政区及中国台湾)销售。未经授权的本书出口将被视为违反版权法的行为。未经出版者预先书面许可，不得以任何方式复制或发行本书的任何部分。

978-981-4195-37-9

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：01-2005-3046

图书在版编目（CIP）数据

专家系统：原理与编程(原书第4版)/(美)吉奥克(Giarratano, J.)等著；印鉴等译. －北京：机械工业出版社，2006.8

(计算机科学丛书)

书名原文：Expert Systems: Principles and Programming, Fourth Edition

ISBN 7-111-19203-6

Ⅰ. 专…　Ⅱ. ①吉…　②印…　Ⅲ. 专家系统　Ⅳ. TP319

中国版本图书馆CIP数据核字(2006)第052081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：李东震

北京京北制版印刷厂印刷 · 新华书店北京发行所发行

2006年8月第1版第1次印刷

184mm×260mm · 34印张

定价：65.00元(附光盘)

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 年开始，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. Berkeley，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

人工智能学科诞生于20世纪50年代，旨在研究如何利用计算机等工具来模仿人类的智能行为。自诞生以来，人工智能就一直是一个富有挑战性的领域，它以其诱人的目标和略显神秘的面纱，吸引了人类持久和狂热的追求，在众多的人工智能领域中，专家系统是一个最富有代表性和最重要的应用分支。

人工智能和专家系统曾取得过许多令人瞩目的成果，也走过不少弯路、经历过不少挫折。近几年来，随着计算机网络、通信等技术的发展，特别是 Internet 和 World Wide Web 的普及，人工智能与专家系统的研究再度活跃起来，并正向更为广泛的领域发展。

本书是一本关于专家系统的著名教科书。本书全面介绍了专家系统原理，并通过 CLIPS 详细讨论了其实际应用。内容涉及知识表示、推理方法、不确定性推理、不精确推理以及开发专家系统的一系列实用技术。本书还附送 1 张包括 CLIPS 程序、源代码以及其他相关文档的光盘。

本书理论与实际相结合，内容由浅入深，适用于计算机科学、管理信息系统、软件工程专业高年级本科生和研究生及相关专业人员，为其了解和设计专家系统提供了理论基础和编程指导。

本书前言及第1～5章、附录A～C由印鉴翻译，第6、10～12章及附录D～G由陈忆群翻译，第7～9章由刘星成翻译，全书由印鉴组织、校阅和统稿，中山大学胡菁同志也给予很多帮助，参加了文字校对工作。

限于译者水平，书中疏漏之处，敬请读者批评指正。

译者简介

印鉴：男，博士，教授，博士生导师，1968年生。1994年毕业于武汉大学计算机科学系，获工学博士学位。现任中山大学信息科学与技术学院计算机科学系副主任和计算机基础教育中心主任。1993年，曾在美国洛杉矶ALPHA OMEGA公司从事合作科研。1997年、2000年曾在香港浸会大学电脑学系作访问学者。2005年在美国东华盛顿大学（Eastern Washington University）作访问学者。目前主要从事数据挖掘、人工智能、知识工程等方面的研究工作。

陈忆群：女，硕士，助教，1979年生。2005年毕业于中山大学计算机科学系，获工学硕士学位。现任广东教育学院计算机科学系教师。2004年，曾在新加坡国立大学计算机学院访问学习。目前主要从事信息处理、数据挖掘、知识工程等方面的研究工作。

刘星成：男，博士，副教授。1964年生于江西安福。1989年毕业于华中理工大学（现华中科技大学）自动化专业，获硕士学位。2001年毕业于中山大学无线电物理专业，获博士学位。2002年～2003年，在英国Southampton大学作博士后研究；2004年～2005年，在美国俄勒冈州立大学（Oregon State University）作访问学者。现于中山大学信息科学与技术学院电子与通信工程系任教。目前主要从事智能信息处理、无线通信及安全等方面的研究工作。

前　言

如何有效地使用本书

本书第4版是对这本全球使用的专家系统与CLIPS专家系统工具编程课本的一个主要修订本。自从20世纪80年代进入商业应用以来，专家系统得到了巨大的发展。今天，专家系统已被广泛地运用到商业、科学、工程、农业、制造、医药、视频游戏以及实质上其他每一个领域。事实上，现在已很难举出一个没有应用专家系统的领域。

本书主要介绍专家系统原理与编程，适用于计算机科学、管理信息系统、软件工程专业以及其他一些对专家系统有兴趣的高年级本科生或研究生。一些新出现的术语用黑体字表示并给出了解释。书中还给出了大量的实例和参考资料帮助理解并指导更深层次的阅读。在新的第4版中，许多新的软件工具的免费和试用版本可以作为额外练习的基础和学习材料，它们的链接都在附录G中。

对新材料的讨论一般从其历史背景开始，这样便于学生理解为什么要开发它们，而不仅仅是学会如何使用它们。教育的核心应聚焦在为什么要创造新技术来解决问题，而不是简单地教授如何去应用。

本书分为两部分：第1～6章介绍原理，第7～12章介绍CLIPS专家系统工具编程。第一部分包括了专家系统所涉及的理论以及专家系统如何适合计算机科学范畴。

学过人工智能的有关课程会对学习本书帮助较大，在本书中，对与专家系统有关的一些人工智能知识也在第1章作了一个自我完备的介绍。单独一章自然无法涵盖人工智能的所有内容，但已足够概观人工智能以及专家系统所扮演的角色。本书第一部分包括了逻辑、概率、数据结构、人工智能概念和其他形成专家系统理论的内容。

我们尝试通过对专家系统理论的介绍来使学生学会对专家系统技术的运用。这里，要强调一点，专家系统和其他工具一样，既有优点，也有缺点。在理论部分还介绍了专家系统与其他编程方法的关系，如传统程序设计。另外，也希望对理论的介绍使学生能够阅读有关专家系统的现行研究文章，但由于专家系统涉及面很广，对初学者来说，仅凭了解就阅读是非常困难的。

本书第二部分介绍了CLIPS专家系统工具。这部分是专家系统编程的一个实例，可以补充和阐释第一部分的理论知识。有了第一部分的理论知识后，编程部分只要具有高级语言的编程经验就可以看懂。学生可以通过CLIPS这个功能强大的现代专家系统工具来了解专家系统开发中的一些实际问题。

在本版中讨论的一个新特性是COOL，即CLIPS面向对象语言。COOL允许完全使用对象，或者使用规则和对象的混合方法来开发专家系统。面向对象方法的优点在于知识集可以方便地组织成比单独规则大的集合。所有对象的一般性质（例如多继承）使得用更多专门知识来扩展对象变得更加容易，而不用像纯规则系统一样，每次都从头开始编写。本版还讨论了CLIPS的过程化编程功能，包括全局变量、函数和类属函数。

CLIPS最初是由Johnson太空中心NASA开发的。Gary Riley是开发基于规则组件的首席程序员。Joseph C.Giarratano作为顾问编写了NASA CLIPS的官方用户指南。现在，CLIPS已用于开发政府、商业、工业以及事实上任何部门的实际项目。使用因特网的任何搜索引擎都能返回成千上万个链接指向使用CLIPS编写的专家系统和采用了CLIPS的世界上很多大学的课程。

由于CLIPS代码是可移植的，它实质上可运行在任何支持ANSI C或C++编译器的机器或操作系统上。本书附带的光盘内容包括：CLIPS在Windows以及MacOS上的可执行程序；CLIPS参考手册和CLIPS用户指南；文档齐备的完整CLIPS C源程序代码。

有些专家系统课程包含一个课程设计，课程设计是提高专家系统开发技能的一个极好方法。学生们常常选择完成一个具有50～150条规则的小型专家系统作为一学期的课程设计。基于这本书已开发了成千上

万个课程设计，包括医疗、汽车维修、的士调度、个人安排、计算机网络管理、天气预报、股市预测、购物咨询等。使用因特网搜索引擎将得到由世界各地的大学开发的很多课程设计与资源，如 PowerPoint 幻灯片、提纲和作业等。

本书作为一学期的课程可安排如下：

1. 第 1 章简要介绍专家系统，习题 1、2、3 可作为练习。

2. 第 7～10 章介绍 CLIPS 基本编程。这部分内容对学生重新编写第 1 章习题 2 的程序，并比较专家系统方法与最初在第 1 章中所使用的语言方法非常有帮助。通过比较，可使学生发现基于规则的语言如 CLIPS、LISP、PROLOG 与在习题 2 中使用的其他语言的差别。可选地，讲授完第 10 章后，教师可返回到理论章节。如果学生具有较强的逻辑和 PROLOG 知识，可以跳过第 2、3 章的多数内容。对没有和只有一点人工智能课程中关于 LISP 简单知识的学生，如果希望重点加强逻辑和专家系统基础理论，则将从第 2、3 章得到很大提高。如果学生具有较强的概率和统计知识，则从第 4 章开始到第 4.11 节可以跳过。

3. 第 4 章和第 5 章讨论对不确定性的处理。这些非常重要，因为人类始终都在处理不确定性，如果没有它，专家系统并不比简单的判定树强多少。不确定性包括概率和贝叶斯推理、确定性因子、Dempster-Shafer 理论以及模糊理论。如果学生想阅读相关方面的现行文章或从事此领域研究工作，他们必须掌握好这些内容。

4. 第 6 章讨论专家系统中的知识获取和软件工程问题，这部分内容主要针对那些想开发大型专家系统的学生。在布置课程设计前，可以不学此章。事实上，可以最后讲述此章，以使学生更好地熟悉建造一个高质量专家系统的所有因素。

补充资源

在出版社的网站http://www.course.com 上可以下载具有单号习题和部分双号习题解答的手册，并有完整的 PowerPoint 幻灯片。另外，很多软件和其他资源的网址在本书中也随处可见。这些资源经过了筛选，学生使用软件可以对习题的关键部分进行实验，而不只是书面求解，这样能更好地理解书中的内容，例如逻辑和概率。大量有关人工智能、逻辑、概率、贝叶斯推理、模糊逻辑和其他主题的资源也都包含在内，以便学生对国际上人工智能和专家系统群体有更广泛的了解。（需要教辅资源的教师，可填写书后的教学支持服务表，并与原出版商联系。——编辑注）

感谢对 CLIPS 有贡献者

感谢所有对 CLIPS 的成功开发有贡献者。作为一个大的项目，CLIPS 凝聚了许多人的心血。其中，主要有：Robert Savely，JSC 高级软件技术首席科学家，是他构思出此项目，并自始至终给予指导与支持；Chris Culbert，软件技术分部主管，是他负责此项目并起草了 CLIPS 参考手册初稿；Gary Riley，设计开发了 CLIPS 中基于规则部分，合写了 CLIPS 参考手册、CLIPS 结构手册，开发了 Macintosh 上的 CLIPS 界面，并维护 CLIPS 的官方网站http://www.ghg.net/clips/CLIPS.html；Brian Donnell，开发了 CLIPS 中面向对象的语言（COOL），合写了 CLIPS 参考手册、CLIPS 结构手册；Bebe Ly，开发了 CLIPS 的 X Window 界面；Chris Ortiz，开发了 CLIPS 的 Windows 3.1 界面，Houston-clear Lake 大学的 Joseph Giarratano 博士，编写了 NASA 的每个 CLIPS 版本的官方用户指南；特别是，Frank Lopez，编写了 CLIPS 的最初原型版本。

致谢

在写作本书的过程中，很多人给予了大量的帮助，包括：Ted Leibfried，Jeanne Leslie，Mac Umphrey，Terry Feagin，Dennis Murphy，Jenna Giarratano 和 Melissa Giarratano。我们还要感谢反馈了信息的第 4 版审稿者：Akron 大学的 Chien-Chung Chan；Ohio 大学的 Constantine Vassiliadis；加拿大 Concordia 大学的 Jenny Scott；Villanova 大学的 Anthony Zygmont。

我们还要感谢许多从 1985 年 CLIPS 第一版发布起，20 年来一直致力于提高 CLIPS 的人们。通过提供免费的 CLIPS 完整源代码，开源组织极有效地提高了 CLIPS 的功能及影响力。这一切在我们 1985 年刚开

发 CLIPS 时是不可想像的。那时候专家系统只是新的未经试验的技术，没有人知道它能否经得起时间的考验。在过去的 20 年中，CLIPS 从 NASA 的最初谨慎开始，发展成为在世界各地有成千上万的人们使用，并证明了各个领域都从中受益。我们特别要感谢这些扩展了 CLIPS 功能和能力的开发者，是他们使 CLIPS 一开始在 NASA 中仅作为人工智能技术简单试验的充满风险的小项目发展成为世界范围的潮流。

Ernest Friedman-Hill，对专家系统的推广做出了重要的贡献，他独自开发了具有新特性的 CLIPS Java 版本，称为 JESS。他还写了一本关于 JESS 的书：《Jess in Action：Rule-Based Systems in Java》，里面有很多有趣的项目。

JESS：（http://herzberg.ca.sandia.gov/jess/）和 KAPICLIPS 1.0：（http://www.cs.umbc.edu/kqml/software/kapiclips.shtml）更加完备了 CLIPS。

CLIPS 的其他后代版本

PerlCLIPS（http://www.discomsys.com/~mps/dnld/clips-stuff/）

Protégé：CLIPS 的一个本体和基于知识的编辑器
（http://protege.stanford.edu/index.html）

Python-CLIPS interface（http://www.yodanet.com/portal/Products/download/clips-python.tar.gz/view）

TixClips：使用 Tix 的 CLIPS 专家系统集成开发环境（http://tix.sourceford.net/）

TclClips（www.eolas.net/tcl/clips），SWIG（http://www.swig.org/）wrapping
（http://starship.python.net/crew/mike/TixClips/）

WebCLIPS：作为 CGI 应用程序的 CLIPS 实现。

WebCLIPS：（http://www.monmouth.com/~km2580/wchome.htm）

wxCLIPS，一个使用图形用户界面的开发知识库系统应用程序的环境：
（http://www.anthemion.co.uk/wxclips/wxclips2.htm）

ZClips 0.1 允许 Zope 和 CLIPS 交互：
（http://www.zope.org/Members/raystream/zZCLIPS0.1）

CLIPS/R2，Production Systems Technologies 公司的：
（http://www.pst.com/clips_r2.htm）

可获取的其他 CLIPS 版本，如加拿大国家研究委员会的 FuzzyClips：
（http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyClips/fuzzyCLIPSIndex.html）

Togai InfraLogic 公司的 FuzzyClips：
（http://www.ortech-engr.com/fuzzy/fzyclips.html）

AdaCLIPS：（http://www.telepath.com/~dennison/Ted/AdaClips/AdaClips.html）

CLIPS 与 Perl 的扩展：（http ://cape.sourceforge.net/）

许多其他基于 CLIPS 的工具版本列在：
（http://www.ghg.net/clips/OtherWeb.html）

目　录

出版者的话
专家指导委员会
译者序
译者简介
前言

第 1 章　专家系统导论 …… 1
1.1　概述 …… 1
1.2　专家系统的定义 …… 1
1.3　专家系统的优点 …… 5
1.4　专家系统的基本概念 …… 5
1.5　专家系统的特点 …… 7
1.6　专家系统技术的发展 …… 8
1.7　专家系统的应用与领域 …… 12
1.8　语言、外壳、工具 …… 15
1.9　专家系统要素 …… 16
1.10　产生式系统 …… 20
1.11　过程化程序规范 …… 23
1.12　非过程化程序规范 …… 27
1.13　人工神经系统 …… 30
1.14　专家系统与归纳学习的关系 …… 34
1.15　人工智能的发展状况 …… 34
1.16　小结 …… 37
习题 …… 38
参考文献 …… 38
第 2 章　知识的表示 …… 41
2.1　概述 …… 41
2.2　知识的含义 …… 42
2.3　产生式 …… 45
2.4　语义网 …… 47
2.5　对象－属性－值三元组 …… 50
2.6　PROLOG 和语义网 …… 50
2.7　语义网的困难之处 …… 53
2.8　模式 …… 54
2.9　框架 …… 55
2.10　框架的困难之处 …… 57
2.11　逻辑与集合 …… 58
2.12　命题逻辑 …… 60
2.13　一阶谓词逻辑 …… 63
2.14　全称量词 …… 63
2.15　存在量词 …… 64
2.16　量词与集合 …… 65
2.17　谓词逻辑的局限性 …… 66
2.18　小结 …… 66
习题 …… 67
参考文献 …… 68
第 3 章　推理方法 …… 71
3.1　概述 …… 71
3.2　树、格、图 …… 71
3.3　状态与问题空间 …… 74
3.4　与或树和目标 …… 77
3.5　演绎逻辑与三段论 …… 79
3.6　推理规则 …… 83
3.7　命题逻辑的局限性 …… 89
3.8　一阶谓词逻辑 …… 90
3.9　逻辑系统 …… 91
3.10　归结 …… 93
3.11　归结系统与演绎 …… 95
3.12　浅推理和因果推理 …… 97
3.13　归结与一阶谓词逻辑 …… 99
3.14　正向链和反向链 …… 103
3.15　其他推理方法 …… 107
3.16　元知识 …… 112
3.17　隐马尔可夫模型 …… 113
3.18　小结 …… 114
习题 …… 114
参考文献 …… 117
第 4 章　不确定性推理 …… 119
4.1　概述 …… 119
4.2　不确定性 …… 119
4.3　误差种类 …… 121
4.4　误差与归纳 …… 122
4.5　经典概率 …… 124
4.6　经验主观概率 …… 127
4.7　复合概率 …… 128
4.8　条件概率 …… 129

4.9 假设推理与反向归纳 …… 133
4.10 时间推理与马尔可夫链 …… 135
4.11 信任几率 …… 138
4.12 充分性与必然性 …… 139
4.13 推论链中的不确定性 …… 141
4.14 证据组合 …… 144
4.15 推理网 …… 148
4.16 概率的传播 …… 155
4.17 小结 …… 158
习题 …… 158
参考文献 …… 161
第5章 不精确推理 …… 163
5.1 概述 …… 163
5.2 不确定性与规则 …… 163
5.3 确定性因子 …… 167
5.4 Dempster-Shafer 理论 …… 174
5.5 近似推理 …… 182
5.6 不确定性的现状 …… 210
5.7 模糊逻辑的一些商业应用 …… 211
5.8 小结 …… 212
习题 …… 212
参考文献 …… 215
第6章 专家系统设计 …… 217
6.1 概述 …… 217
6.2 选择合适的问题 …… 217
6.3 开发专家系统的步骤 …… 220
6.4 开发过程中的误区 …… 222
6.5 软件工程与专家系统 …… 225
6.6 专家系统生命周期 …… 226
6.7 详细生命周期模型 …… 229
6.8 小结 …… 232
习题 …… 232
参考文献 …… 233
第7章 CLIPS 介绍 …… 235
7.1 概述 …… 235
7.2 CLIPS …… 235
7.3 记号 …… 236
7.4 字段 …… 237
7.5 进入和退出 CLIPS …… 239
7.6 事实 …… 240
7.7 增加和删除事实 …… 242
7.8 修改和复制事实 …… 243
7.9 监视命令 …… 244
7.10 自定义事实结构 …… 245
7.11 规则的组成 …… 246
7.12 议程与执行 …… 247
7.13 结构处理命令 …… 250
7.14 打印输出命令 …… 252
7.15 使用复合规则 …… 252
7.16 设置断点命令 …… 253
7.17 调入和保存结构 …… 254
7.18 注释结构 …… 255
7.19 变量 …… 256
7.20 变量的复合用法 …… 257
7.21 事实地址 …… 257
7.22 单字段通配符 …… 259
7.23 块世界 …… 260
7.24 多字段通配符和变量 …… 263
7.25 小结 …… 267
习题 …… 267
参考文献 …… 271
第8章 高级模式匹配 …… 273
8.1 概述 …… 273
8.2 字段约束 …… 273
8.3 函数和表达式 …… 275
8.4 使用规则求和 …… 277
8.5 BIND 函数 …… 279
8.6 I/O 函数 …… 279
8.7 棍子游戏 …… 283
8.8 谓词函数 …… 284
8.9 测试条件元素 …… 285
8.10 谓词字段约束 …… 286
8.11 返回值字段约束 …… 287
8.12 棍子游戏程序 …… 288
8.13 OR 条件元素 …… 288
8.14 AND 条件元素 …… 290
8.15 NOT 条件元素 …… 291
8.16 EXISTS 条件元素 …… 292
8.17 FORALL 条件元素 …… 294
8.18 LOGICAL 条件元素 …… 295
8.19 小结 …… 298
习题 …… 298
第9章 模块化设计、执行控制和规则效率 …… 305
9.1 概述 …… 305
9.2 自定义模板属性 …… 305

9.3 优先级 …… 310
9.4 阶段和控制事实 …… 312
9.5 优先级属性的误用 …… 314
9.6 自定义模块结构 …… 316
9.7 输入、输出事实 …… 318
9.8 模块与执行控制 …… 320
9.9 Rete 模式匹配算法 …… 326
9.10 模式网络 …… 327
9.11 连接网络 …… 329
9.12 模式顺序的重要性 …… 331
9.13 排列模式以求高效 …… 335
9.14 多字段变量与效率 …… 335
9.15 测试条件元素与效率 …… 336
9.16 内置的模式匹配约束 …… 337
9.17 通用规则与专用规则 …… 337
9.18 简单规则与复杂规则 …… 339
9.19 小结 …… 340
习题 …… 341
参考文献 …… 346
第 10 章 过程化程序设计 …… 347
10.1 概述 …… 347
10.2 过程化函数 …… 347
10.3 自定义函数结构 …… 352
10.4 自定义全局变量结构 …… 358
10.5 自定义类属和自定义方法结构 …… 363
10.6 过程化结构和自定义模块 …… 374
10.7 有用的命令和函数 …… 375
10.8 小结 …… 379
习题 …… 380
第 11 章 类、实例和消息处理程序 …… 383
11.1 概述 …… 383
11.2 自定义类结构 …… 383
11.3 创建实例 …… 384
11.4 系统定义消息处理程序 …… 384
11.5 自定义实例结构 …… 386
11.6 类与继承 …… 386
11.7 对象模式匹配 …… 392
11.8 用户定义消息处理程序 …… 399
11.9 槽存取和处理程序创建 …… 403
11.10 BEFORE、AFTER 和 AROUND 消息处理程序 …… 405
11.11 实例创建、初始化和删除消息处理程序 …… 416
11.12 修改和复制实例 …… 418
11.13 类和类属函数 …… 420
11.14 实例集合查询函数 …… 421
11.15 多继承 …… 424
11.16 自定义类和自定义模块 …… 429
11.17 调入和保存实例 …… 430
11.18 小结 …… 431
习题 …… 431
第 12 章 专家系统设计实例 …… 433
12.1 概述 …… 433
12.2 确定性因子 …… 433
12.3 判定树 …… 436
12.4 反向链 …… 445
12.5 监视问题 …… 453
12.6 小结 …… 464
习题 …… 465
参考文献 …… 466
附录 A 一些有用的等式 …… 467
附录 B 一些基本量词公式及其含义 … 468
附录 C 一些集合性质 …… 469
附录 D CLIPS 支持信息 …… 470
附录 E CLIPS 命令与函数概要 …… 471
附录 F CLIPS BNF 范式 …… 492
附录 G 软件资源 …… 498

第 1 章　专家系统导论

1.1　概述

本章是对专家系统的一个概略介绍，主要介绍专家系统的基本原理。在本章中，将讨论专家系统的优缺点，描述专家系统应用的适用范围，并讨论专家系统与其他编程方法的关系。

1.2　专家系统的定义

20 世纪人们提出了很多种**人工智能**（artificial intelligence，AI）的定义，其中一个最早的定义之一“使计算机像人类一样思考”至今仍在使用。许多科幻电影验证了这一观点。事实上这种定义是基于英国数学家与计算机先驱 Alan Turing 的著名的“图灵测试”。在这个测试中，一个人试图判断与之通过远程键盘交谈的对象是人类还是计算机程序。如果计算机能在测试中顺利回答问题而不被发现是计算机，则说明计算机已经具有了**强人工智能**（Strong AI）。“强人工智能”被人们定义为基于严格逻辑基础，以区别于基于人工神经网络、遗传算法和进化方法的**弱人工智能**（Weak AI）。在今天，很明显没有一种人工智能技术能成功解决所有问题，而一些组合的方法会更有效一点。

第一个通过图灵测试的程序是 Steven Weizenbaum 在 1967 年写的心理测试程序。从那以后，人类知识和交互的研究得到很大发展，并举办了一个名为 Loebner 的奖金高达 100 000 美元的竞赛（http：//www.loebner.net/Prizef/loebner-prize.html）。当然，今天的交互多数用的是声音识别而不是旧式的电传打字机或键盘。如果你曾与人通过电话交谈但对方不理解你的话，也许该问问他是否通过了图灵测试。

从 20 世纪 60 年代开始，专家系统就作为一种研究工具而被开发，作为人工智能的一个特定部分，它可以成功解决某些领域如医疗诊断的复杂问题。但建立一个通用的人工智能程序解决一切问题难以实现，因为缺乏问题领域的专家知识，比如医疗诊断。自从 20 世纪 80 年代早期，专家系统展现了其商业用途之后，就越来越受到欢迎并得到发展。今天，专家系统已用于商业、科学、工程、制造和其他许多具有良定义问题的领域。实际上，如果你在为使用普通的支付手段还是使用信用卡而犹豫，专家系统可以帮你做出决定。

前面提到的术语“良定义”会在后面章节中做更多详细讨论。其最基本的含义是如果人类专家可以确定解决一个问题的推理步骤，那么专家系统也可以做到。如果不能确定推理，则可能只得靠运气。

作为一个反例，很多人尝试写一个专家系统来预测股票市场。事实上，华尔街一直在使用这些系统。然而，如果考察股票的变化，你不能发现明显的趋势，而专家系统也不比开发它的人强多少。这些系统的最大好处是，在现实交易中，一毫秒的延迟就会带来损失，竞争者的专家系统可以和你一样注意到股票趋势或下单买卖上亿美元价值的股票，但可以比人类快得多。但它不一定工作得好，1987 年声名狼藉的股市崩盘导致了一系列新的限制规定的出台，这些规定限制计算机不能仅为获得几百元的收益而销售上亿支股票，因为这类行为可能导致股市崩溃。

专家系统在人工智能技术领域有非常成功的应用。许多方法使其他技术与专家系统结合，例如遗传算法、人工神经网络等。使用了人工智能的系统通常称为**智能系统**（intelligent system）或自动系统（Hopgood 01）。

一般，解决任何问题的第一步是先划定要解决问题的范围或**领域**（domain）。不论是传统编程领域还是人工智能领域这一点都是相同的。然而，出于以前对人工智能感到神秘的缘故，人们往往相信这样一句过时的说法：“所谓人工智能问题就是该问题还没有解决”。另外一种流行的定义则是“人工智能就是使计算机像他们在电影中所做的一样行动起来”。在 20 世纪 70 年代，当人工智能仍处在研究阶

段时，这种想法就已经广为人知了。但是，今天人工智能已解决了许多现实问题，并且已应用到商业领域。在线杂志 PCAI.com 以及会议如 AAAI（http：//aaii.org/conferences/conferences.htm）还有其他许多书籍（Luger 02）都有相关内容讨论，更详细的资料可参看附录 G。

在更详细地讨论人工智能之前，让我们回头看看它是如何符合生命模式的。什么是生命？对于生命有很多不同的定义（Adami 98），从生理学、新陈代谢、基因和热力学等不同角度。哪一个是正确的呢？这要看你对生命的哪个角度感兴趣。也许最简单的是 Shakespeare 的定义“生命就是由傻瓜讲述的充满声音、愤怒而没有任何含义的故事。”

从计算机的角度来说，生命就如同软件一样。在 Adami 的书所附带的光盘中包含一个软件，用户可以创造人工的生命模式进行体验。也有抽象定义，如电影《黑客帝国》（Matrix）中描述的一样，代表人类的“行”生活在巨大的计算机程序中。其他数字计算机中的生命描述可以在 Helmreich 的书（Helmreich 98）中找到，这本书详细描述了计算机人工生命的生理学甚至精神细节。

从生理学的角度看，我们不再局限于从计算机系统中寻求人工生命模式。从 20 世纪 90 年代开始，已经可以成功地克隆动物，例如绵羊多利（Dolly），公司像销售母牛一样为失去宠物的主人克隆他们的宠物。但是通过克隆得到一个活的生物的复制品只是“改进”生命的第一步。例如，有一些研究人员正在尝试创造带有萤火虫基因的兔子，以便使它能在黑暗中生存。这种形式的生命模式在自然界没有祖先，是真正的人工生命。同时，因为这些生物是智能的，它们也就具有了人工智能，尽管这并不是用计算机的形式来表现的。

人工生命的扩展新领域是创造进化系统（creative evolutionary system），在这个系统里人工生命系统可以根据进化压力改变自己的程序（Bentley 02）。许多不同的技术，例如遗传算法等在实际应用中得到描述，例如音乐、艺术、电路设计、建筑、战斗机设计等。同样，Bentley 书中所附带的光盘允许用户体验创造进化系统。注意这本书是关于这些系统的计算机表现，而不是未来的新的设计者们所梦想的生物生命模式，例如一只成年兔子在黑暗中照顾小孩，或者一只比人更能胜任飞机驾驶的猴子。

de Silva 的书给出了智能的另一个定义：“智能就是学习、获取、适应、修正和扩展知识以便解决问题的能力”（de Silva 00）。从这个角度来说，我们的目标是通过机器人、工厂、工具和其他硬件建立能和现实世界交互的智能机器。其中的挑战是把现实世界中复杂的人类思维融合到机器中，例如歧义、含糊、一般、不精确、不确定、模糊、信任以及似然等。在这本书第 4 章和第 5 章中将有更多的讨论。注意前面这句话本身就带有歧义。“这本书”是指 de Silva 的书还是本书？作为高级生物，我们早已习惯于处理这些问题，但是对于机器和计算机来说，如果只使用传统的逻辑将会遇到很多困扰。

一个更具挑战性的问题是开发具有人工智能同时具有意识的系统。关于人脑我们已经了解了很多（Cotterill 98），但我们仍然不清楚意识体现在哪个部位，或者说，是什么让你具有了自己的特性。但是随着新的工具例如功能磁性共振图像（functional magnetic resonance imaging，fMRI）的使用，大脑可动态映像以查出本能反应时是激发了哪个部位。当然，如果我们使用人造克隆动物模型，那么我们早已成功了，因为克隆的绵羊、猫当然是具有意识的。然而，我们还是不知道如何使一台机器具有意识。更重要的是，当我们看过《终结者》（Terminator）或者 Matrix 等电影以后，想要一个智能机器的想法可能不是一个好的念头。毕竟，无论是人类还是机器，都不喜欢被毁灭掉。

尽管经典的人工智能问题，如：自然语言翻译、语音理解、视觉识别等仍未完美解决，但如果限制问题的范围则可能会找到一个有效的解决方法。例如，如果限定句子形式为主、谓、宾，那么建立简单的自然语言系统就不会很困难。目前，这类系统在为众多软件产品，如数据库系统和电子表格系统等提供友好的用户界面上做得很成功。与说话人无关的声音识别系统在今天也有了很高的精确率，不像以前的系统需要先用某个用户的声音做训练学习。与专家系统结合，这些智能系统一旦通过了图灵测试，将最终取代许多电话中心以记录用户订单（Luger 02）。

目前已有很多商用的声音识别系统可以在标准的个人电脑上运行，并且其价格也合理。有很多声音识别系统也广泛应用在汽车的免提电话中，只要问题领域限定在阿拉伯数字而不是所有单词，它就

有很好的识别能力。事实上，目前流行的计算机文字冒险游戏中的解析器展现了一种令人惊讶的自然语言理解能力，而这种能力正是局域网联网多人游戏所必需的，因为输入文字会使游戏速度减慢。

专家系统已与类似人类模式识别及自动决策等系统的数据库相结合，以便通过**数据挖掘**（data mining）来进行知识发现，这导致了**智能数据库**（intelligent database）的产生（Bramer 99）。一个重要的应用是在飞机安全系统中，使用可疑人士的人脸识别作为专家系统的前端，以决定是否需要向上级汇报。

另一个令人激动的人工智能领域和人工**发现系统**（discovery system）有关。这是一些能够真正发现某问题领域中的知识的计算机程序。例如，Automatic Mathematician（AM）程序发现新的数学定理，并再次发现了已经由人类归纳出来的如素数的特点等知识。BACON 3 系统发现新的科学知识，例如开普勒第三行星运动定理的一个版本，在 Wagman 的书中汇总了许多发现系统（Wagman 99）。

尽管人工智能最初在 20 世纪是作为计算机科学的一个分支被提出来的，它现在已经成为一个应用于许多领域，如计算机科学、心理学、生物学、神经系统科学等的基础学科。事实上，越来越多的大学设置了人工智能学位。

人工智能有许多备受关注的领域，如图 1.1 所示。**专家系统**（expert system）就是对传统人工智能问题中智能程序设计的一个非常成功的近似解决方法。专家系统早期先导者之一，斯坦福大学的 Edward Feigenbaum 教授，把专家系统定义为“一种智能的计算机程序，它运用知识和推理过程来解决只有专家才能解决的复杂问题”。也就是说，专家系统是一种**模拟**（emulate）专家决策能力的计算机系统，“模拟”一词表明专家系统要在所有方面都做得像专家一样。模拟比模仿（Simulation）更进一步，模仿只要求在某些方面做得像真正的事物一样。

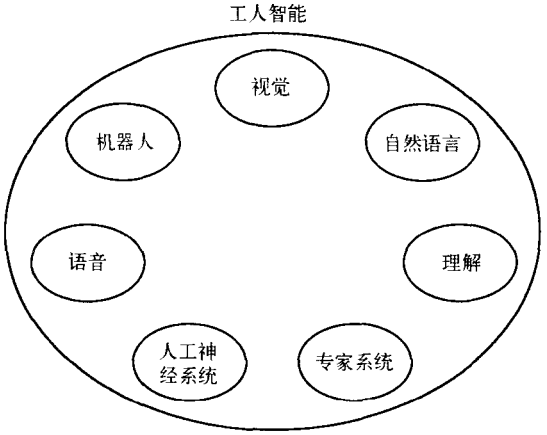


图 1.1 人工智能的一些领域

虽然我们仍未找到一种通用的解决问题的方法，然而专家系统在其限定的领域里做得很成功。你可以从附录 G 列出的书籍、期刊、会议、产品以及专家系统在商业、医学、科学、工程等领域的应用例子看到专家系统的成功应用。

专家系统大量利用专业知识以解决只有**专家**（expert）才能解决的问题。专家是一个在特定领域里具有专门知识的人。亦即，专家具有不为大多数人所知或所利用的专门技能或知识。专家能够解决大多数人所不能解决或是不能高效地（而不是低劣的）解决的问题。在最初发展起来时，专家系统特指包含专家知识。然而“专家系统”这一术语在今天适用于任何应用专家系统技术的系统。专家系统技术包括专门的专家系统语言、程序和为了辅助专家系统开发和执行而设计的硬件。

专家系统中的知识可以是专门知识或是从书籍、杂志、有学问的人处可获得的知识。从这个角度上说，知识也代表比更罕见的专家知识更低层次的内容。专家系统、**基于知识的系统**（knowledge-

based system）和**基于知识的专家系统**（knowledge-based expert system）这些术语经常同义地使用。多数人使用“专家系统”这一术语仅仅是因为它较短，即使在他们的专家系统中可能仅有一般的知识而没有专门知识。

图 1.2 描述了一个基于知识的专家系统的基本概念。用户提供事实或其他信息给专家系统，相应地收到专家建议或**专门知识**（expertise）。专家系统内部包括两个主要部分：知识库和推理机。知识库包含有为让**推理机**（inference engine）得出结论而使用的知识。这些结论是专家系统对用户询问的响应。

基于知识的系统还被设计成为专家的智能助手。由于开发上的优势，这些智能助手是用专家系统技术来设计的。随着添加进智能助手的知识越来越多，智能助手越来越像一位专家。因此开发智能助手将会成为设计一个完整的专家系统过程中的里程碑。此外，还可以通过加快解决问题的速度使专家空出更多时间。智能家教是人工智能的另一个应用。与原来的计算机辅导系统不同，新的系统可提供上下文相关的指导（Giarratano 91a）。

与通用问题求解技术方面的知识不同，专家知识是指特定**问题域**（problem domain）方面的知识。特定问题域是专家能成功解决问题的领域，例如医学、经济、科学或者工程学等。正如人类专家一样，专家系统是针对某一个问题域而设计的：比如，你通常不会期望一个拱师具有医学方面的专门知识。在一个问题域的专门知识不会自动地转向另一个问题域。

解决特定问题的专家知识称为专家的**知识域**（knowledge domain）：例如，诊断传染疾病的内科专家系统会有许多关于传染疾病症状方面的知识。在这种情况下，知识域是医学，包括疾病、症状、治疗方法等方面的知识。图 1.3 说明了问题域和知识域之间的关系。图中知识域完全包括在问题域之内，知识域之外的部分是一个对问题域内问题没有任何知识的区域。

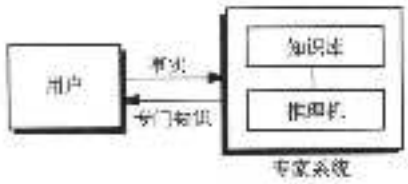


图 1.2 专家系统的基本作用



图 1.3 问题域和知识域的关系

一个传染病诊断专家系统一般不具有医学的其他分支的知识，例如外科或小儿科。尽管专家系统关于传染病的知识与一个人类专家相当或更多，但如果没有使用其他领域的知识编程，那么专家系统就不知道其他领域的任何知识。

专家系统在它所具有的知识中推导或**推理**（inferences）的方式与人类专家推断解决问题的方式是一样的。即是，给出一些事件，一个逻辑，然后推理出结论。比如，如果你的爱人已经一个月没有跟你说话了，你会推断他（或她）没有什么事值得说。然而，这只是几个可能推断中的一个。

表 1.1 对技术的不同视角

人	问 题
管理者	我能用它做什么？
技术人员	我怎样才能把它做得最好？
研究者	我如何去扩展它？
消费者	它会节省我的时间还是金钱？
商业人员	我能减少劳动力？
证券经纪人	它会如何影响季度利润？

对任何一种技术都有各种方法去衡量它的用处。表 1.1 总结了在一工程技术中参与者的不同观点，在表中，技术人员可以是工程师或软件设计者。工程技术可以是硬件或是软件。无论解决什么难题，这些问题都得回答，否则这一技术就不能成功地被使用。像其他任何工具一样，专家系统有其适宜或不适宜的应用领域。第 6 章将详细讨论如何选择合适的应用。

1.3 专家系统的优点

专家系统有许多吸引人的特征：

- 适应性强。专家知识在任何合适的计算机硬件上都是可利用的，实际上，专家系统是专家知识的集成体。
- 成本低。提供给用户的专家知识成本非常低。
- 危险性低。专家系统可用于那些可能对人有害的环境。
- 持久性。专家知识是持久的，不像人类专家那样会退休，或者死亡，专家系统的知识会无限地持续。
- 复合专家知识。复合专家知识可以做到在白天或晚上的任何时候同时、持续地解决某一问题。由几个专家复合起来的知识，其水平可能会超过一个单独的专家。
- 可靠性强。专家系统可增强正确决策的信心，这是通过向专家提供一个辅助观点而得到的。此外，专家系统还可协调多个专家的不同意见。不过，如果专家系统是由某一个专家编程设计的，那这个方法就不能奏效。如果专家没有犯错误的话，专家系统应该始终与专家意见一致。但是，如果专家很累或有压力就可能会犯错误。
- 解释、说明。专家系统能明确、详细地解释导出结论的推理过程。一个人可能会太厌烦、不愿意或是没有能力去这样做，但明确、详细的解释可增强得出正确决策的信心。
- 响应快。迅速或实时的响应对某些应用来讲是必要的。依靠所使用的软件或硬件，专家系统可以比专家反应得更迅速或更有效。某些突发的情况需要响应得比专家迅速，因此实时的专家系统是一个好的选择。
- 始终稳定、理智和完整的响应。实时和突发情况下，当专家由于压力或疲劳而不能高效地解决问题时，这一点是至关重要的。
- 智能家教。专家系统可以作为一个智能家教，让学生执行实例程序，解释系统的推理。
- 智能数据库。专家系统能以智能的方式来存取一个数据库，例如数据挖掘。

开发专家系统的过程也会有一个间接的益处，由于专家知识必须以精确的形式输入到计算机中，所以知识要被明确地了解而不是被隐含于专家的脑海中，这样，就必须对知识进行正确性、一致性和完整性检查，这就提高了知识的质量（专家可不理解这一点）。

1.4 专家系统的基本概念

专家系统的知识可以用多种方式描述。描述知识的一个常用方法是用 IF…THEN 型的**规则**(rules)，例如：

```
IF 灯是红的 THEN 停止
```

如果灯是红的这一事件出现，就与模式“灯是红的”相匹配，规则得到满足，执行“停止”行为。虽然这是一个非常简单的例子，但许多重要的专家系统都是通过规则来表达专家知识而建立的。实际上，开发专家系统的这种基于知识的方法已经完全取代了 20 世纪 50～60 年代早期人工智能的方法，那时人们致力于使用复杂的不依靠知识的推理技术。某些专家系统工具，如 CLIPS，允许使用**对象**(objects)，知识可以封装在规则和对象中。规则可以与对象及事件相匹配，而对象则可以独立于规则而操作。

自在如何配置计算机系统方面已胜过单个人类专家的专家系统第一次成功应用于 Digital Equipment 公司的 XCON/R1 系统以来，专家系统已不止一次显示出它们的价值和可用性。许多针对专门任务的小型系统也有几百条的规则，虽然这些小型系统的运作可能达不到专家的水平，但它们也充分利用了专家系统技术来处理那些知识密集型任务。对这些小型系统而言，其知识主要来源于书、期刊或其他公共资料。

与此相反，一个典型的专家系统主要依赖于没有写下的知识，这主要通过**知识工程师**（knowledge engineer）长期地与人类专家进行沟通而获得。建立一个专家系统的过程称为**知识工程**（knowledge engineering），这个过程是由知识工程师来完成的，知识工程师从专家或其他来源获取知识并把它们编码到专家系统中。

图1.4描述了开发专家系统的一般步骤。知识工程师首先通过与专家进行对话而获取专家知识，这个阶段与传统程序设计中系统设计人员与用户讨论系统需求相类似。然后知识工程师将知识编码到知识库中，随后专家评估系统并返回意见给知识工程师。这个过程一直循环，直到系统的性能为专家所满意为止。

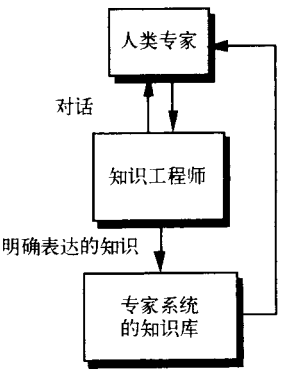


图1.4 专家系统的开发

对于采用了基于知识技术的应用而言，**基于知识的系统**（knowledge-based system）这一表述是较好的术语。因为它可被用于专家系统和基于知识的系统的创建。然而，正如“人工智能”这一术语一样，现今凡涉及到专家系统和知识系统，即便是知识未达到专家水平，人们也往往使用“专家系统”这一术语。

一般来说，专家系统设计不同于传统的程序设计，因为其问题通常没有算法去求解，而是依靠推理来获得一个合理的解决方法。算法是一种理想的解决方案，因为它会在有限的时间内给出答案（Berlinski 00）。然而，算法也许不能令人满意而问题的复杂性也会增加，所以需要使用人工智能。在没有任何可利用的算法帮助我们获得最佳方法时，一个合理的方法就是最好的。因为专家系统依赖于推理，它必须能够解释这个过程，所以它的推理过程是可以检查的。**解释机**（explanation facility）是复杂专家系统的一个必要部分，实际上，复杂的解释机可设计为允许用户深入探究 What if 类问题，称为**假设的推理**（hypothetical reasoning）问题。

有些专家系统甚至允许系统通过**规则归纳**（rule induction）从例子中学习规则，在归纳时，系统从数据中生成规则。把专家知识整理成规则并非易事，特别是当专家知识还未被系统化时。在一个专家系统中，专家知识可能会存在不一致性、模糊性、重复性或其他问题，除非可以形式地表示专家系统中的知识，否则这些问题都难以解决。

人类专家也知道自己知识的局限性，当问题达到他们的**未知界限**（limits of ignorance）时，他们会给建议打上一定折扣。人类专家也知道何时“打破规则”。如果专家系统没有专门设计来解决不确定性问题的话，即使它们处理的数据不精确、不完整，专家系统也会以同样的确信来给出建议。专家系统的建议与专家的建议一样，在其不知晓的范围内其合理性应降低。

现今许多专家系统的一个不足是缺乏**因果知识**（causal knowledge），也就是说专家系统并不能真正地理解系统中隐含的原因和结果。用基于经验和启发性的**浅**（shallow）知识来设计专家系统比用基于对象的基本结构、功能和行为的**深**（deep）知识要容易得多。例如，设计一个针对头痛开阿司匹林药方的专家系统比设计一个关于人体所有基本的生物化学、生理学、动物学、神经医学等知识的专家系统容易得多。一个人体机能模型的程序设计工作量非常之大，即使是成功了，由于系统要处理全部的信息，系统的响应时间也可能会很慢。

启发性知识（heuristic knowledge）是“浅”知识的一种类型，“启发性”（heuristic），这个词来源于希腊，意味着“探索”。启发性解决方法不能保证用同样的算法能取得成功。启发性知识是一种从实践中获得的经验性知识，它对问题的求解可起帮助作用，但不能保证一定有效。不过，在许多领域，如医学、工程学，启发性知识对问题的求解起着重要的作用。即使知道一个确切的解决方法，但由于花费或时间的限制，使得该方法不能实际应用。启发性知识能够提供有价值的捷径，可以帮助减少时间和花费。

专家系统的另一个不足是它的知识受限于系统的知识域。典型的专家系统不能像人那样，通过类

比（analogy）来一般化知识以获得求解新的问题的方法。虽然通过规则的归纳，专家系统可以获得少许新的知识。创建一个专家系统的常用方法，即知识工程师访问专家、设计原型、测试，然后再重复，这个过程是一件既费时、又费精力的事情。实际上，把人类知识转化到专家系统这一问题是如此的重要，以至于人们称它为**知识获取瓶颈**（knowledge acquisition bottleneck）。这是一个描述性的术语，因为如同瓶颈限制液体流入瓶子一样，知识获取的瓶颈限制着专家系统的创建。

尽管有一些局限性，但专家系统仍成功解决了现实世界的许多问题，这些问题是常规的程序设计方法学所不能解决的，尤其是那些需要处理不确定或不完全信息的问题。重要的一点是了解这种新技术的优点和局限性，只有这样才能够使之得到恰当的运用。

1.5 专家系统的特点

专家系统的设计具有以下一些特点：

- 高性能。系统能以此领域里专家的同等或更高水平响应，也就是说系统所给建议的质量必须很高。
- 适当的响应时间。系统必须能在合理的时间内工作，此时间与专家得出一个结论所需的时间相当或更适当。与专家需要一个小时的时间相比，需要一年才得出结论的专家系统是不会太有价值的。特别是必须在一特定时间间隔内做出响应的实时系统中，**时间限制**（time constraints）就显得更为苛刻。例如在雾中降落飞机。
- 好的可靠性。专家系统必须可靠，且不易崩溃，否则就不能使用。
- 可理解性。在执行过程中，系统能解释推理步骤，使之易于理解。专家系统不是一个推出不可思议答案的黑盒子，而是应该具有解释能力，其解释的方式应与专家解释他们推理的方式一样。对以下几个原因而言，这个特征非常重要。

原因之一是人的生命和财产可能会依靠专家系统的回答。由于危害的巨大潜在性，专家系统必须能够以专家解释如何得出某一结论的方式来证明它的结论是正确的。因此，解释为人提供了一个可理解的推理检测。

第二个原因出现在专家系统的发展阶段，这一阶段是让解释机证实知识已经被正确地获取并正被系统正确地使用。这在调试中是很重要的，因为知识可能会被不正确地带入或者由于知识工程师和专家间的误解而导致其不正确。一个好的解释机允许专家和知识工程师证实知识的正确性。而且由于传统专家系统创建的方法，读一个程序并理解它的操作将会很困难。

另外一个错误源也许是专家系统里没有预想到的相互影响，这些相互影响可以通过运行测试实例来检测到，这些测试实例按照系统必须遵循的推理方法来运行。正如后面将要详细讨论的内容一样，复合规则可以提供一个系统推理的环境。专家系统中的执行流不是按顺序的，因此仅仅一行行地阅读代码，是不能明白系统是如何工作的。也就是说，规则进入系统的次序并不一定是它们被执行的次序。专家系统就像一个规则独立于知识处理机的并行程序。

- 灵活性。专家系统可能有大量的知识，因此具有一个增加、修改、删除知识的高效机制是十分重要的。基于规则的系统得以普及的一个原因就是由于规则的高效和模块化存储能力。

视系统而定，一个解释机可以简单也可以复杂。在基于规则的系统中，一个简单的解释机可以显示使最近规则得以执行的所有事实。而在更为复杂的系统中可能按照如下方式来做：

- 列出所有支持和反对某个假设的原因。假设是将要被证明的目标，例如在一个医疗诊断专家系统中的“病人有破伤风感染”就是一个假设。在现实问题中可以有复合假设，正如一个病人可以同时有几种病一样。一个假设也可以被看作是一个事实，其正确性仍存在疑惑，需要被证实。一旦目标被假设将推出更简单的子目标，直到子目标可以被解决。这种解决问题的方法是典型的“分而治之”方法，也是后面章节介绍的向后链的基础。
- 列出所有可解释观测证据的假设。

- 解释假设的所有推断结果。例如，假设病人确实有破伤风，由于感染作用，就应该有发烧的迹象。如果后来观察到此症状，就会增强此假设正确的可信度，如果没有发现此症状，就会削弱假设的可信度。
- 给出当假设是正确时将发生事件的一个预测（prognosis）。
- 提供程序需要用户进一步信息的问题的依据。这些问题可以用来指导推理链朝着可能的诊断路径前进。在大多数现实问题中，探究所有的可能性花费太大或者需要太长的时间，并且要提供特定的方法引导正确搜索。例如，对一个报怨咽喉痛的病人，考虑一下进行所有内科检查所需的花费及所用的时间、效果（这对医疗收益很有利，但对病人很不利）。
- 提供程序所用知识正确的依据。例如，如果程序断言“病人有破伤风感染”这一假设是对的，用户可以要求解释，程序必须给出得到这一结论的依据是基于下面规则：如果病人进行血液检测，其破伤风是阳性，那么病人就患有破伤风。此时用户可要求程序提供此规则正确的依据，则程序可向用户说明：血液检测为阳性是患有疾病的证据。然而，正如后面章节将讨论的，这种方法忽略了假阳性。

在这个实例中，程序实际上在引用一个关于规则的知识——**元规则**（metarule）。“元”代表“之上”或“之外”。程序已经包含了使用机器学习的方法来推理新规则。假设通过知识而证实，知识通过正确的**根据**（warrant）来证实。根据实质上是一个解释专家系统推理说明的元解释。

在基于规则的系统中，知识可以很容易地**增加**（incrementally）。也就是说，知识库可以随着规则的添加而逐步增加，从而使得系统的性能和正确性得到持续地检查。这个过程有点类似于一个小孩每天学习新知识并检查其正确性。如果规则设计得好，那么规则间的相互影响作用就会非常小或者没有，从而消除那些难以预料的副作用。知识的这种逐步增加可以**快速原型化**（rapid prototyping），以使知识工程师可很快地演示专家系统的工作模式。这是一个重要的特征，因为它可保持专家和管理者对项目的兴趣。快速原型化还可迅速暴露出专家知识或系统中的缺陷、不一致性或错误，从而使之能够立即得到纠正。

1.6 专家系统技术的发展

专家系统的理论基础涉及到诸多学科，其中一个主要理论基础是**认知科学**（cognitive science）。认知就是研究人类如何认知处理信息，换句话说，就是研究人类如何思考，尤其是如何解决问题。许多认知工具已经被开发来提供更好的教学（Lajoic 00）。

另一个重要概念是人工智能机器必须具备识别符号的能力，由此建立了一个基础领域，称为**符号论**（semiotics）（Fetzer 01）。在符号论中，我们并不是指简单的如“停”这样的符号，而是指整个普遍意义上的符号。符号是表达某些其他含义的东西。例如，当你看一部有背景音乐的电影时，其中当令人激动的事情要发生时，音乐的节奏变快而高昂。同样，当令人不安的事情发生时，音乐变慢而低沉。在音乐、电影、电视和日常生活中存在许多非语言符号。例如，当一个人对一个问题撒谎或感到不安时，他通常会看着地下。这给设计能工作在现实世界的智能机器增加了极大的负担。简单地理解语言的程序远远不够，机器必须能够识别这些符号后面的含义。

如果我们想要计算机模拟专家，那么对认知的研究是非常重要的。通常，即使问题是由专家解决的，他们也不能解释自己是如何解决问题的。在一个基于精确知识的专家系统中，如果不能解释问题是如何解决的，则把知识译成代码是不可能的。在这种情况下，唯一的可行方法是设计通过自学习来模拟专家的程序。这些程序建立在归纳、人工神经系统和其他软件计算方法的基础之上，我们将在后面讨论。

人类问题求解与产生式

专家系统技术的发展有着广阔的背景，表1.2总结了现代专家系统的一些重要发展并最后发展出

CLIPS。只要可能，项目的开始日期都会给出，许多项目都延续了几年以上。在本章和其他章里都较详细地分析了这些发展。附录 G 中介绍了许多专家系统工具和现代专家系统。

表 1.2 导致 CLIPS 第一次发布的一些重要事件

年 份	事 件
1943	Post 产生式规则；McCulloch and Pitts 神经元模型
1954	控制规则执行的马尔可夫算法
1956	Dartmouth 会议；逻辑学家；启发性搜索；创立“AI”术语
1957	Rosenblatt 提出感知机；GPS（General Problem Solver，通用问题求解器），（Newell、Shaw 和 Simon）
1958	人工智能语言 LISP（McCarthy）
1962	Rosenblatt 关于感知的神经动力原理
1965	自动定理证明的归结方法（Robinson） 模糊对象的模糊推理逻辑（Zadeh） 开始建立 DENDRAL，第一个专家系统（Feigenbaum、Buchanan 等）
1968	语义网，联想记忆模型（Quillian）
1969	MACSYMA 数学专家系统（Martin 和 Moses）
1970	PROLOG（Colmerauer，Roussell 等）
1971	语音识别 HEARSAY Ⅰ 人类问题求解通用规则（Newell 和 Simon）
1973	MYCIN 医疗诊断专家系统（Shortliffe 等）及由此产生的 GUIDON，即智能家教（Clancey） TEIRESIAS，概念解释机（Davis） EMYCIN，第一个外壳（Van Melle、Shortliffe 和 Buchanan） HEARSAY Ⅱ，多协作专家的黑板模型
1975	框架，知识表示（Minsky）
1976	AM（Artificial Mathematician，人工数学家），数学概念的创造性发现（Lenat）不确定性推理的 Dempster-Shafer 证据理论 开始建立矿产探测的 PROSPECTOR 专家系统（Duda、Hart 等）
1977	OPS 专家系统外壳（Forgy），CLIPS 的一个祖先
1978	开始建立 XCON/R1 以配置 DEC 计算机系统 Meta-DENDRAL（McDermott、DEC），元规则，规则归纳（Buchanan）
1979	快速模式匹配的 Rete 算法（Forgy） 人工智能开始商业化 Inference 公司成立（1985 年发行 ART 专家系统工具）
1980	Symbolics 和 LMI 推出 LISP 机
1982	SMP 数学专家系统；Hopfield 神经网络； 开发智能计算机的日本第五代语言项目
1983	KEE 专家系统工具（IntelliCorp）
1985	CLIPS 专家系统工具第一版（NASA），可以在任何计算机上使用而不限制在特制的昂贵 LISP 机器

在 20 世纪 50 年代后期及 60 年代，人们编写了大量的以通用问题求解为目标的程序，其中最著名的是**通用问题求解器**（General Problem Solver）。这引起了巨大的关注，因为巨大的计算机充满了整个房间，并称之为“巨脑”（Giant Brains）。人们担心他们会因此失去工作，直到 IBM 宣布机器只会处理那些需要一百万年才能完成的成千个数学运算。在那些机器价值一百万美元而 CPU 时间以 1 毫秒 1 美元的价格售卖的日子，人们无法想像有一天自己可以有便宜的个人电脑在家庭或者工作中使用。

Newell 和 Simon 证明的最重要结果之一是大部分的人类问题求解或**认知**（cognition）可以用 IF…THEN 类型的**产生式规则**（production rule）表达。例如，“如果”看起来将要下雨，“那么”带上一把

雨伞，或者“如果”你的爱人心情不好，“那么”你不要显得很高兴。与一个小的、模块化的知识集相对应的规则称为**一块**（chunk），块以松散的形式连接、组织，并与相关的知识有联系。其原理之一是所有的人类记忆都以块的形式组织。下面是用一条规则表示一个知识块的例子：

```
IF 汽车运转不了并且
   油箱是空的
THEN 加油
```

Newell 和 Simon 用规则表示知识并显示了如何用规则推理。认知心理学家已经使用规则作为模型来解释人类信息处理，其基本思想是感官的接收对大脑产生刺激，刺激引发出适当的**长期记忆**（long-term memory）规则并生成恰当的响应，长期记忆是我们的知识存储处。例如，我们都有如下的规则：

```
IF 有火焰 THEN 有火灾
IF 有烟雾 THEN 可能有火灾
IF 有报警 THEN 可能有火灾
```

可以看到后两个规则在表述上不是完全肯定，火可能已经灭了，但空中可能仍有烟雾。同样，报警声并不证明就有火灾，因为可能是一个虚假的报警信号。看到火焰、闻到烟雾、听到报警声的刺激会诱发出这些或相似的规则。

长期记忆包括许多形如 IF…THEN（如果……那么）简单结构的规则。实际上一个技艺高超的棋师可能通晓 50 000 或更多的关于棋的模式的知识块。与长期记忆相反，**短期记忆**（short-term memory）是在解决问题过程中用来暂时存储知识的。尽管长期记忆能够容纳成千上万甚至更多的块，然而正工作着的记忆的容量是惊人地小——4～7 块。试着在脑海中浮现几个数字就是一个简单的例子，大多数人一次仅可以浮现 4～7 个数字，但他们能记住的远不只 4～7 位数字，只不过这些数字是存储在长期记忆中。

一种理论假设短期记忆表示那些可以同时活跃的数据块，并把人类问题的解决当作脑海中这些已激活块的传播。最后那个块被激活的强度如此之大以致产生出一个有意识的想像。

人类问题求解的另一个必要元素是**认知处理机**（cognitive processor），它尽力去发觉那些将被适当刺激**激活**（activated）的规则。但并不是任何规则都可激活，例如，你不会每次听到警报声就想到给油箱充油。只有与刺激相匹配的规则才会被激发。如果很多规则同时被激发，认知处理机必须处理冲突来决定哪一个规则有最高优先权，这个规则将会被执行。例如，如果以下两个规则是激发态的：

```
IF 有火灾 THEN 离开
IF 我的衣服着火了 THEN 扑灭火
```

此时，具有最高优先权的那条规则将会执行。对现代专家系统来说，**推理机**（inference engine）就相当于认知处理机。

Newell 和 Simon 把人类问题求解的模型归纳为：长期记忆（规则）、短期记忆（工作内存）、认知处理器（推理机），这三者是现代基于规则的专家系统的基础。

这里所说的规则是一类**产生式系统**（production system）。目前，基于规则的产生式系统是实现专家系统的一种流行方法。构成产生式系统的规则称为产生式**规则**（production rule）。在设计专家系统的过程中，一个重要的因素是知识的数量和规则的**粒度**（granularity）。粒度太小，则在没有其他规则参考的情况下很难理解规则，粒度太大，则专家系统很难修改，这是因为几个知识块被集成在一条规则中。能够像使用规则一样地使用对象是 CLIPS 的一个主要功能。

直到 20 世纪 60 年代中期，人工智能的主要目标就是依赖少量的知识和功能强大的推理方法来生成智能系统。所谓的通用问题求解器也是希望能解决大量的问题，而非针对某一特定领域。虽然通用问题求解器的推理功能非常强大，但它仍然只相当于一个初学者，当它面对一个新的领域时，它得从头做起，远比不上一个依赖于高性能领域知识的人类专家。

知识能力的一个例子是博弈。虽然计算机现在能与人匹敌，但计算机如果不是比人计算得快百万倍的话，人将比计算机下得好。研究表明，人类专家下棋并不是依赖于推理，而是依赖于多年下棋所积累的模式。正如前面所提到的一样，人类专家可保存 50 000 个模式。人类识别棋盘上模式的能力非常强，计算机一般通过推理来预测 50～100 步或者更多步后的情形，而人类则通过分析模式来发现潜在的威胁。这个策略比起强力的向前走步预测方法更强，这就使得计算机下棋程序，例如 IBM 的“深蓝（Big Blue）”，可以打败人类棋手。

虽然领域知识非常强大，但它一般局限于某一领域。例如，一个象棋高手不可能自动成为一个数学专家或是一个跳棋专家，虽然某些知识能借鉴到其他领域，如仔细地规划每一步走法，但那主要是技巧而不是知识。

直到 20 世纪 70 年代初期，人们才意识到领域知识才是建造具有人类专家水平的问题求解器的关键。虽然推理方法很重要，但研究表明，专家并不是首先依赖推理来求解问题。事实上，推理只是起到辅助的作用，人类专家主要依赖他们多年所积累的大量启发式知识和经验。只有当专家不能用他们的专家知识去求解问题时，他们才去推理（或是求教另一个专家）。在一个不熟悉的环境下，专家的推理能力与常人无异。早期基于推理建造问题求解器的尝试表明，完全依赖推理是行不通的。

领域知识是建造现实世界问题求解器的关键这一观点导致了专家系统的成功。因此，今天成功的专家系统是基于知识的专家系统而不是通用问题求解器。此外，开发专家系统的技术也同样适用于不需要专门知识的基于知识的系统的开发。

虽然专门知识被认为是专业化的且仅为少数人所知，但它一般可在书本里、网络上、期刊和其他广泛可获得的资料中找到。例如，如何求解一个二次方程式或做积分和微分的知识是可广泛获得的。基于知识的计算机程序如 Mathematica 和 MATLAB 可自动完成这些以及其他一些数字或符号上的数学操作，其他基于知识的程序可以完成工厂的生产控制。今天，**基于知识的系统**（knowledge-based system）和“专家系统”这些术语经常被同义地使用。实际上，现今专家系统被认为是常规程序设计的一个可选择的程序设计模型或**范例**（paradigm）。

基于知识系统的起源

20 世纪 70 年代随着基于知识系统的广为接受，产生了许多成功的专家系统。这些系统都具备完整的文档，许多论文、书籍阐释它们的工作和基础知识。对于现代专家系统，你将遇到的普遍问题是它们的知识都是私有的和保密的。使用专家系统的公司从为他们服务的人类专家中收集知识，他们不愿这些知识被竞争者、特别是律师们获得。想一下，一个医疗专家系统提供病人死亡或声称受伤的诊断，事实上，被检查的人也许没有受伤，只是他的行为符合受伤的人的情况。专家系统中的软件以及知识库被其他专家检查。正如许多判例一样，你可以看到一个专家反驳另一个专家。

这些经典的专家系统能够解释大量光谱图从而确定化学组成（DENDRAL）、诊断疾病（MYCIN）、分析石油（DIPMETER）和矿产的（PROSPECTOR）地质资料、配置计算机系统（XCON/R1）。到 1980 年为止，PROSPECTOR 探测到价值 1 亿美元的矿物淀积层和 XCON/R1 每年为 DEC 节省数百万美元的消息激起了人们对专家系统技术的强烈关注。作为对人类信息处理的研究，在 20 世纪 50 年代开始的人工智能这一分支已通过现实世界应用的实践性程序发展到取得商业性的成功。

MYCIN 专家系统之所以重要有几个原因。首先，它证明了人工智能可以应用到医疗诊断；其次，MYCIN 是新概念的试验，例如解释机、知识的自动获取和今天可在许多专家系统中找到的智能指导。第三个原因是它证实了专家系统**外壳**（shell）程序软件与数据分离的可行性。也就是说，数据和知识并没有被编写在程序中。事实上，外壳允许把一个领域知识方便地替换成另一个领域的知识。

以前的专家系统如 DENDRAL，是一个把知识库中的知识与推理机通过软件集成起来的单一系统。MYCIN 明确地把知识库与推理机分开，这对于专家系统技术的发展是极其重要的，因为这意味着专家系统的基本核心可以重用。也就是说，通过清空旧知识装入新领域的知识，新的专家系统可以比

DENDRAL 类型的系统创建得快速得多。处理推理和解释的 MYCIN 部分，可以用新系统的知识重装。去掉了医学知识的 MYCIN 被称为 EMYCIN（基本的或空的 MYCIN）。

到 20 世纪 70 年代后期，作为今天大多数专家系统基础的 3 个概念已合为一体，如图 1.5 所示，这些概念是规则、外壳、知识。

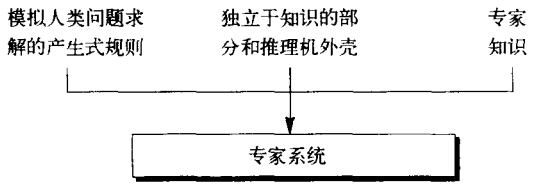


图 1.5 建造现代基于规则的专家系统的三个重要因素

到 1980 年，新的公司已开始把专家系统从大学实验室里带出并且生产商业性产品，而且引进了为专家系统开发服务的功能强大的新软件。直接用 LISP 开发的为专家系统开发服务的强大的新软件和专业软件也开始推出。但遗憾的是，随着个人计算机的功能越来越强大且 CLIPS 等专家系统工具业已开发出来，高昂的花费和可观的训练时间导致这些系统不可避免的灭亡。在 LISP 机器中，原始汇编语言、操作系统和其他基础代码都用 LISP 开发，因此导致大量的维护工作。

CLIPS 用 C 语言开发以提高速度和可移植性，同时使用强大的模式匹配算法 Rete。而且和其他专家系统工具不同，CLIPS 不仅是免费的且具有完备的源程序说明文档。CLIPS 可以安装在支持标准的 Kernigan 和 Richie C 语言的任何 C 编译器上。它已经被安装在许多计算机系列上。正如前言所述，许多从 CLIPS 衍生出来的语言已经开发出来，并带有如模糊逻辑和反向链等特性，它们用 Java 语言开发，称为 Jess。

1.7 专家系统的应用与领域

传统的计算机程序可用来解决许多类型的问题，这些问题通常有算法上的解决方法，这使得它们易于用传统的程序和程序设计语言如 C、C++、Java、C# 等来实现。在许多应用领域如工业和工程技术中，数值计算是极其重要的。与之相反，专家系统基本上是为符号推理而设计的。

虽然经典人工智能语言如 LISP 和 PROLOG 也常被用作符号操作的语言，但它们的应用并不局限于专家系统外壳，这并不意味着不可用 LISP 和 PROLOG 建造专家系统。在今天，**逻辑程序设计**（logic programming）这个词通常指用 PROLOG 编写的程序，虽然很多其他语言也以此为目标开发。事实上，许多专家系统都是用 PROLOG 和 LISP 建造的。PROLOG 用于诊断系统有很多的优点，这主要是由于它内含反向链推理。不过，使用专门为专家系统建造而设计的外壳和实用程序来建造大型专家系统更为方便，效率也更高。因为它不必每建造一个新的专家系统就从头做起。

专家系统的应用

专家系统已被应用到几乎每一个知识领域，其中有些被设计为研究工具，有些则履行着重要的商业和工业功能。专家系统应用于商业的一个例子是 20 世纪 70 年代 DEC 的 XCON 系统，XCON 系统（最初称为 R1）是与 Carnegie-Mellon 大学的 John McDermott 合作开发的，它是为 DEC 服务的一个计算机配置专家系统。

一个计算机系统的**配置**（configuration）意味着当用户下了一个订单时，所有的部件——软件、硬件和资料——都要提供。这至今仍是一个比较重要的问题，特别是当人们通过在 Internet 上填表格选购计算机、汽车和其他产品的时候。对一个计算机厂商或汽车制造商来说，花时间去检查每一个订单所需要的零部件是否齐全然后决定某一个订单是否能发送是非常没有效率的行为。今天，对任何需要配置并在一定时间内发送的事情来说，专家系统是一个符合逻辑和效益的必需品。

大量的专家系统已创建并且已报道在计算机期刊、Internet、书籍和会议上，但这些可能只是涉及到一些皮毛，因为许多公司和军事集团由于所有权或者包含在系统内的秘密知识的缘故而不会报道他们的系统。在已公开的系统基础上，我们可以看出专家系统应用的某些广义的分类，如表 1.3 所示。由于文档齐全，表 1.4～表 1.9 列出了一些经典的专家系统实例，现在的一些系统其应用也类似。

表 1.3 专家系统的广义分类

种 类	通用的领域
配置	以正确的方法配置系统组成
诊断	基于已观察到的迹象推断潜在的问题
教学	智能教学使得学生可以问为什么、怎么样和如果……会怎样之类的问题，如同人在教学一样
解释	解释观察到的数据
监测	比较观察数据和预测数据以判断性能
规划	规划行为以产生预期结果
预测	预测给定情况的结果
补救	对问题给定补救措施
控制	管理一个过程，可能要求解释、诊断、监测、设计、预测和补救

表 1.4 典型化学专家系统

名 称	化 学
CRYSALIS	解释蛋白质的三维结构
DENDRAL	解释分子结构
TQMSTUNE	矫正三重、四重、多重分光（使之协调）
CLONER	设计新型生物分子
MOLGEN	设计基因一克隆实验
SECS	设计复杂的有机分子
SPEX	规划分子生物学实验

表 1.5 典型电子学专家系统

名 称	电 子 学
ACE	诊断电话网络故障
IN-ATE	诊断示波器故障
NDS	诊断国内通信网
EURISKO	设计三维微电子学
PALLADIO	设计并检测新型 VLSI 电路
REDESIGN	重新设计数字电路
CADHELP	指导计算机辅助设计
SOPHIE	指导电路故障分析

表 1.6 典型医学专家系统

名 称	医 学
PUFF	诊断肺病
VM	监视特护病人
ABEL	分析酸/电解液
AI/COAG	诊断血液疾病
AI/RHEUM	诊断风湿疾病
CADUCEUS	诊断内科疾病
ANNA	监测洋地黄治疗法
BLUE BOX	诊断/矫正抑郁
MYCIN	诊断/治疗细菌感染
ONCOCIN	治疗/管理化疗病人
ATTENDING	指导麻醉管理
GUIDON	指导病菌感染分析

表 1.7 典型工程学专家系统

名 称	工 程 学
REACTOR	诊断/补救核反应堆事故
DELTA	诊断/补救 GE 火车头故障
STEAMER	指导电厂操作流

表 1.8 典型地质学专家系统

名 称	地 质 学
DIPMETER	解释钻井日志
LITHO	解释油井运转记录数据
MUD	诊断/补救钻探问题
PROSPECTOR	解释矿产地质资料

专家系统的适用领域

在开始建造专家系统之前，决定这是不是一个适当的模式是很必要的，例如，要考虑是否要用专家系统而不是用传统的程序设计。一个专家系统的适用领域取决于很多因素：

- 传统的程序设计能有效地解决此问题吗？如果答案为“是”，那么专家系统就不是最好的选择。例如，想一下一个诊断某种设备的问题。如果能预先知道所有功能失灵的全部症状，那么一个简单的图表或错误判定树就足够了。专家系统最适合于那些没有高效算法解决的情况，这些情况被称为**非结构化问题**（ill-structured problem），且推理可能会是好的解决方法的唯一希望。

举一个非结构化问题的例子，考虑一个不能决定去哪儿度假且决定去旅行社看看的人的情况。表1.10列出了这个人对旅行社问题的反应所显示出来的一个非结构化问题的一些特征。

表1.9 典型计算机专家系统

名 称	计算机系统
PTRANS	给出管理DEC计算机的预测
BDS	诊断交换网中的损坏部分
XCON	配置DEC计算机系统
XSEL	配置DEC计算机销售订单
XSITE	配置DEC计算机顾客位置
YES/MVS	监控IBM MVS操作系统
TIMM	诊断DEC计算机

表1.10 一个非结构化问题的例子

旅行社的问题	反 应
我能为你做什么吗？	我不确定
你想去哪里？	某些地方
有特别的目的地吗？	不确定
你能负担多少钱？	我不知道
你能弄到钱吗？	我不知道
你什么时候想去？	不定

尽管这可能是一个极端的例子，但它的确阐明了非结构化问题的基本概念，正如你所看到的，由于有如此之多的可能性，非结构化问题就不能有一个较好的算法方法来解决。在这种情况下，当所有其他方法都无效时就应给一个默认的选择。例如，旅行社会说：“哈哈！我有完美的旅程给你：环游世界。请填好下面的信用卡申请表，每一件事都会准备好。”

在处理非结构化问题时，有一点值得注意，那就是专家系统设计可能偶而反映了一种算法，即专家系统开发在不知不觉中使用了一种算法。一个明显的迹象就是硬性规定了一种**控制结构**（control structure），那就是，通过知识工程师明确地设定规则的优先级，这些规则可以按相应的顺序强制执行。在专家系统中强制性的控制结构抹杀了专家系统技术的一个主要优点，即不按照预定的模式来处理预料外的输入。也就是，专家系统随机地对输入作出反应，不管输入什么。传统程序通常希望按照一定的顺序输入。一个有大量控制的专家系统通常表明了一个模糊的算法，这对重写传统程序可能是一个好的选择。

- 这个域容易确定边界吗？在希望专家系统知道什么和它的能力应该是什么上有明确的界限时，这一点是很重要的。例如，假设你想建造一个诊断头痛的专家系统，显然医生的药理知识应该输入到知识库中。然而，为了更深刻地理解头痛，你可能还要输入关于神经化学的知识，然后是它的上一级领域：生物化学，接着是化学，分子生理学，这样一直推可能到亚原子物理。其他域比如生物反馈学、心理学、精神病学、生理学、运动学、瑜伽还有压力管理也可能包含与头痛相关的知识。问题是：你什么时候停止增加域？域越多，专家系统会变得越复杂。

尤其是，协调所有这些专业知识将成为一个主要的任务。从现实世界的经验可知，协调正在解决问题的专家组，特别当他们提出矛盾的建议时是多么的困难。如果我们知道怎样做好专家知识间的协调，那么我们就能尝试建立一个具有复合专家知识的专家系统。这种尝试已经在专家系统HEARSAY Ⅱ和HEARSAY Ⅲ中用到。举一个常见的例子，当有故障时把车子开进多个服务中心，你会得到许多截然不同的判断。但当专家变少时，选择的数量也将变少，这使得决定一个行动非常困难。

- 该专家系统是否有需求？虽然建立一个专家系统是伟大的体验，但如果没有人愿意用，它就是

毫无意义的。如果已经有大量的人类专家，那就很难用缺乏专家这一理由去为专家系统辩护。而且，如果专家或用户不想要这种系统，即使有需要，它也不会被接受。例如，许多模拟显示人工智能控制交通灯可以减少大约 50%的汽油消耗，但它同时会减少城市、省和政府汽油税收的 50%。

管理阶层应该特别愿意支持这种系统。相对于传统的程序，专家系统显得更重要，因为它被视为减轻工作压力的手段。工人必须确信引入专家系统之后不会造成失业，而是当专家知识更容易以低成本获取后自己会取得较大的收益。专家系统领域值得获取更多支持，因为它试图去解决传统程序不能解决的问题，风险是巨大的，回报也是巨大的。

- *至少有一个愿意合作的人类专家吗？*一定得有一个愿意合作且对这个项目充满热情的专家。并非所有专家都愿意把他们的知识接受错误检查后再输入计算机。即使以后开发中有很多专家愿意合作，限制加入专家的数目可能是明智的。不同的专家有不同的解决问题的方法，比如要求不同的诊断检测手段，有时他们甚至会得出不同的结论。应该努力去解决知识库中需要解决、可能引起内部冲突和不相容的问题。
- *专家能够解释他的知识以使知识工程师理解吗？*即使专家愿意合作，但用清晰的词语表达这些知识可能有困难。一个简单的例子是，你可以用语言表达你是怎样移动一个手指的吗？虽然你会说这是通过收缩手指上的肌肉来完成的，但接下来的问题是，你怎样收缩手指肌肉？专家和知识工程师沟通上的另一个困难是：知识工程师不懂专家的专门术语。对于医学术语，这个问题尤其严重，知识工程师可能要花费一年或者更长的时间仅仅去理解专家所谈论的内容，何况还要把那些知识转换成清晰的计算机代码。
- *求解问题的知识主要是启发式和模糊的吗？*当专家的知识大部分是启发式和模糊时，专家系统是非常适合的，这是因为，知识可以基于经验，这种知识叫**经验知识**（experiential knowledge），专家可以在一个方法无效时，尝试不同的方法，换一句话说，专家的知识与其说是基于逻辑和算法，不如说更多是一种“尝试—错误”反复的方法。然而，专家仍然比非专家解题快些，因此这是一个适于专家系统的应用。假如问题可以简单地用逻辑和算法去解决，那最好还是用传统程序去处理。

1.8 语言、外壳、工具

定义一个问题的一个基本决策是决定怎样才能最好地建立它的模型，有时经验可用来帮助选择最好的范例。例如，对工资表，经验建议最好用传统的过程程序设计，而且，如果可以的话，使用商业软件包比从头写一个更好。选择范例的一个普遍原则是首先考虑最传统的方法——通常的程序设计方法。这是因为，对传统的程序设计，我们已有丰富的经验和大量的商业软件包。如果一个问题不能用传统的程序设计方法来解决，此时，我们再来考虑非传统的方法，如人工智能方法，它要求有理论知识而不仅仅是 C 语言过程。

和关系数据库语言 SQL 一样，专家系统语言是一种比 LISP 或 C 语言层次更高的语言，因为它更容易去做某些事情，但能够解决问题的范围也更小。因此，专家系统语言的这种专门性使它们适合专家系统而不适合一般的编程。在很多情况下，甚至需要从专家系统语言中暂时退出以便去执行过程语言的一个函数。CLIPS 特意设计了使这种转换更容易的特性。

专家系统语言和过程语言之间的主要功能差异集中在表示上。过程语言着重提供一种灵活的、健壮的技术去表示数据。例如，诸如线性表、记录、链表、堆栈、队列和树的数据结构很容易被创建并熟练操作。现代语言如 Java 和 C#，通过对象、方法和软件包来更好地辅助数据抽象。这提供了一个抽象层。数据和操纵它的方法是紧密交织在对象中的。相反，专家系统语言提供灵活、健壮的方法去表示知识。专家系统允许两个抽象层，数据抽象和**知识抽象**（knowledge abstraction），专家系统语言特别地把数据和操作数据的方法分离开来，这种分离的一个例子是：在一个基于规则的专家系统语言中，

事实（数据抽象）和规则（知识抽象）是分离的。CLIPS还提供了对象并且具备真正的面向对象语言的特性。

着重点的不同也导致程序设计方法上的不同。由于在过程语言中数据和知识的紧密交织，程序员必须仔细描述执行的顺序。然而，在专家系统语言中数据和知识的明确分离使得其更少考虑执行顺序的严格控制。典型的，如推理机，就是一段完全独立的代码块，它主要用来把知识作用到数据上。知识和数据的分离允许更高程度的并行性和模块化。

决定一个专家系统是否需要的通常方式是决定你是否想把一个人类专家的专门知识编成程序。假如这样的专家存在而且愿意合作，那么专家系统的方法可能会获得成功。同样的，一个高度知识密集型并且不具备确定性的任务最好采用专家系统工具来解决。

选择一种专家系统语言让人难以抉择，因为有太多不同的选择。尽管CLIPS并不具备其他语言的所有特性，但它简单易学，因此是一本入门教科书的较好选择。而且，CLIPS还具备程序规模小和在实时响应要求严格时执行速度快等优点。

除了可用语言的选择困惑外，用来描述这些语言的术语也是令人困惑的。一些卖主称他们的产品为“工具”，其他一些则称作“外壳”，还有一些称为“集成环境”。本书为清楚起见，定义一些术语如下。

- **语言**（Language）：按照某种特定语法书写的命令的翻译器。专家系统语言会提供一个推理机去执行该语言的语句。依赖这种执行，推理机会提供正向链、反向链推理或两者都提供。在这种定义下，LISP不是一种专家系统语言而是PROLOG。然而，用LISP写一种专家系统语言和用PROLOG写人工智能语言是可能的。因此，你甚至可以用汇编语言写一个专家系统或人工智能语言，只是要考虑开发周期、便利、可维护、高效和速度这些问题。
- **工具**（Tool）：语言加上有关的实用程序，以使应用程序的开发、调试、交付更方便。实用程序可能包括文本图形编辑器、调试器、文件管理器、代码生成器等。有些工具允许使用不同的模板，比如在一个应用中可有正向和反向链推理。

有些工具可能和实用程序集成在一个环境中，以提供给用户一个统一的界面。这种方法使用户离开当前环境去执行任务的需求降到最小。例如，一个简单的工具可能不提供文件管理方面的功能，因此用户不得不退出这个工具去执行操作系统命令。一个集成环境使环境中实用程序之间的数据交换变得容易。一些工具甚至不需要用户写任何代码，只需按表和表格的例子输入知识，它就会自动生成适当的代码。

- **外壳**（Shell）：一种专门工具，为某些类型的应用而设计。在这些应用中用户仅仅只需提供知识库。一个典型的例子是EMYCIN外壳，它是通过去除MYCIN专家系统中的医学知识库而建立的。

MYCIN通过反向链推理来诊断疾病。通过去除医学知识，EMYCIN可作为一种外壳使用，它可用于那些使用反向链推理的咨询系统。EMYCIN外壳证明了MYCIN软件中一些基本部件如推理机和用户界面的可重用性。这是现代专家系统开发技术上非常重要的一步，因为这意味着对每一新的应用，再也不必从头开始建立一个专家系统。如今，专家系统工具领域已经具备大量可用的用户界面和组件。

专家系统的特点表现在很多方面，如知识的表示、正向或反向链推理、不确定性支持、假设推理、解释机等。除非建立过大量的专家系统，否则是难以体会所有这些特点的，尤其是那些昂贵工具上所具备的特点。学习专家系统技术的最好方法是用一种易学的语言去开发一些系统，然后当你需要更多功能时再使用更复杂的工具。

1.9 专家系统要素

一个典型专家系统的要素如图1.6所示。在一个基于规则的系统中，知识库包含以规则形式编码的解决问题的领域知识。虽然规则是表示知识的常用形式，但有些专家系统使用了不同的表示，本书将在第2章讨论。

专家系统由下列几部分组成：

- **用户界面**（user interface）——用户和专家系统之间的通信机制。
- **解释机**（explanation facility）——解释系统的推理给用户。
- **工作内存**（working memory）——被规则所使用的事实的**全局数据库**（global database）。
- **推理机**（inference engine）——通过决定那些规则满足事实或目标，并授予规则优先级，然后执行最高优先级规则来进行推理。
- **议程**（agenda）——由推理机创建的一个规则优先级表，这些规则匹配工作内存中的事实或目标。

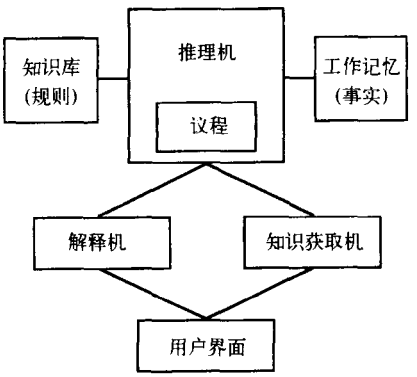


图 1.6 基于规则的专家系统的结构

- **知识获取机**（knowledge acquisition facility）——为用户建立的一个知识自动输入方法，以代替知识工程师去编码知识。

知识获取机在很多系统中是一个可选的功能。某些专家系统工具可以通过例子归纳学习，并自动生成规则。在机器学习中也使用其他方法如 ID3、C 4.5、C 5.1、人工神经网络和遗传算法等来产生规则。机器学习产生规则的主要问题是无法解释为什么会产生这样的结果。不像人类可以解释一个规则的产生原因，机器学习系统无法解释它们的行为，因此可能产生不可靠的结果。不过这些例子通常来自适于判断树形式的列表或表格数据。一般通过知识工程师建立的规则要比来自归纳的简单规则复杂得多。

对不同的系统，用户界面可以是简单的文本显示方式，也可以是复杂的、高分辨率的位映射显示方式。高分辨率显示一般用来模拟一个带按钮和显示器的控制板。

在基于规则的专家系统中知识库又叫**产生式存储库**（production memory）。作为一个简单的例子，考虑决定横过街道的问题。两条规则的产生式如下，箭头表示如果左边的条件为真则系统执行右边的行为。

```
灯是红色→停止
灯是绿色→走
```

这样的产生式规则可以用一条 IF…THEN 伪码表达，格式为：

```
规则:红灯
IF  灯是红色
THEN  停下来

规则:绿灯
IF  灯是绿色
THEN  走
```

每条规则用一个名字标识，后面跟着这条规则的 IF 部分。在规则 IF 和 THEN 之间的部分有许多不同的叫法，如**前件**（antecedent）、**条件部分**（conditional part）、**模式部分**（pattern part）、或**左部**（left-hand-side，LHS），单独的条件“灯变绿”称为**条件元素**（conditional element）或**一个模式**（pattern）。

下面给出几个真实系统中的规则例子。

诊断髓膜炎和菌血病症（细菌感染）的 MYCIN 系统：

```
IF
```

感染体是血液，并且
细菌的类别不能确切知道，并且
细菌的染色是革兰氏阴性，并且
细菌的外形是杆状，并且
病人有严重发烧

THEN

以不太充分的证据(可信程度 0.4)说明细菌的类别是假单胞细菌属

用来配置 DEC VAX 计算机系统的 XCON/R1 系统：

IF

当前环境是分配设备给总线组件，并且
有一个未分配的双端口磁盘驱动器，并且
所需控制器的类别是知道的，并且
每个控制器没有任何设备分配给它，并且
这些控制器能够支持的设备数目已知

THEN

分配磁盘驱动器给每一个控制器，并且
记下相关的控制器对，其中每一个控制器支持一个驱动器

在基于规则的系统中，推理机决定哪些规则的前件被事实满足。专家系统中用来求解问题的两个常用推理策略是**正向链**（forward chaining）和**反向链**（backward chaining）。为了特殊需要而使用的其他方法可能包括：手段-目的分析、问题简化、回溯、规划-生成-测试、逐级规划和最小满足原则、约束处理。

正向链是从事实到结论的推理，例如，如果你离家之前看到外面正在下雨（事实），那么你应该拿把伞（结论）。

反向链则是从假设，即要证明的结论，到事实的推理。例如，假如你看到外面有人穿着湿鞋，拿着伞进来，你的假设就是正在下雨。为了支持这个假设，你可以问这个人是否真的在下雨。如果回答是，那么假设证明为真而成为事实。正如前面所述，一个假设可以当作一个真伪尚未确定的事实，它可作为一个要证明的目标。

根据不同的设计，推理机既可以作正向链、也可以作反向链推理。例如，OPSS 和 CLIPS 是正向链推理，而 PROLOG 则是反向链推理。而由 Paul Haley 开发的另一个版本的 CLIPS，称为 Eclipse，两者都提供。推理机的选择取决于问题的类型。诊断问题最好用反向链推理，而预测、监视、控制则最好使用正向链推理。不管怎样，使用一个功能更强的工具需要存储容量和执行速度的牺牲，因为有更多的代码来构成该工具。设计 CLIPS 的目的是用来“学习和体验”，因此它会用最快的速度执行并适用于小存储器件例如 ROM 的应用开发。尽管使用具有 512 Mbyte 内存（RAM）台式计算机的人通常不会意识到这一点，但开发一个像智能远程控制或微波感应器等产品需要 ROM 便宜的永久存储。存储的价格随着容量下降，对消费产品来说，存储越小越具竞争力。CLIPS 可以产生一个足够小的可运行 C 代码以便嵌入到小型 ROM 中。

工作记忆可能包含关于交通灯当前状态的事实如“灯是绿”或“灯是红”，这些事实可能一个或者两个同时在工作记忆中，如果交通灯工作正常，则只有一个事实在记忆中。然而，如果交通灯有故障，那么有可能两个事实都在记忆中。注意知识库和工作内存的区别。事实彼此之间不会相互作用，“灯是绿”这个事实对“灯是红的”这个事实没有任何作用。不过，关于交通灯的知识会表明如果两个事实同时存在，那么灯有故障。

如果有一个事实“灯是绿”在工作记忆中，那么推理机会注意到这个事实满足绿灯规则的条件部分并将这条规则放到议程中。如果一条规则有多个模式，那么只有所有的模式同时得到满足时才能把它放到议程中。有些模式被满足还要通过在工作记忆中指定某些没有的事实。

使所有模式得到满足的规则称作是被激活的或者是被例化的，多个被激活的规则可同时在议程中。在这种情况下推理机必须选择一个规则去触发它。术语“触发”来自神经生理学。当受到刺激时，一个独立的神经细胞或神经元会产生一个电信号，除非达到一定数量的刺激才能够引起神经元在短时间内再次触发，这种现象叫“反射”。为了避免无谓的循环，基于规则的专家系统采用了反射方法。即，如果绿灯规则在同样的事实上一次又一次地被触发，则专家系统不做任何事情。

有很多不同的方法实现反射。在 OPS5 专家系统语言中，每一个事实进入工作记忆时，都赋予它一个叫时间标签的唯一标识。在规则被一个事实触发后，推理机就不会使它在那个事实上再次触发，因为它的时间标签已经用过了。

THEN 后面的部分是规则触发时将要执行的一系列**行为**（actions）。规则的这部分称作**后件**（consequent）或**右部**（right-hand side，RHS）。当红灯规则触发时，它的行为（停止）就被执行。类似地，当绿灯规则触发，它的行为是走。一些特定行为通常包含从工作记忆中增加、删除一些事实或是打印结果。这些行为的格式依赖于专家系统语言的语法。例如，在 CLIPS 中，增加一个叫“stop”的事实到工作记忆中的行为是（assert stop），由于 LISP 的缘故，其模式和动作需要用括号括起来。

推理机不停地**循环**（cycles）工作。描述这种循环有不同的名字，如**识别动作循环**（recognize-act cycle）、**选择执行循环**（select-execute cycle）、**环境反应循环**（situation-response cycle）和**环境行为循环**（situation-action cycle）等。不管叫什么名字，推理机都会重复执行一组任务，直到某种执行终止的判定标准出现。其循环的任务列在下面的伪码中，包括**冲突归结**（conflict resolution）、**动作**（act）、**匹配**（match）和**终止检测**（check for halt）。

```
WHILE 还没终止
```

冲突归结（conflict resolution）：如果有激活者，那么选择有最高优先级的一个。否则终止。

动作（act）：顺序执行选中的激活者右部的行为，在这个循环中那些改变工作记忆的激活者有即时的效果。从议程中删去刚刚触发过的激活者。

匹配（match）：通过检测任何一个规则的左部是否被满足来更新议程。如果满足，激活它们。如果规则的左部不再满足则删除它们。

终止检测（check for halt）：如果一个终止行为被执行或给出一个中断命令，那么终止。

```
END-WHILE
```

接受一条新的用户命令（Accept a new user command）

多个规则可能在一个循环中被激活并放到议程中。另外，有些激活者会由于前一次的循环而留在议程中，除非它们的左部不再被满足。因此激活者的数目在执行过程中会不同。有的程序，一个激活者可能总在议程中，但却从来没有被选中去触发。同样地，一些规则可能从来没有被激活。在这些情况下，应该重新检查这些规则，看其是否必要或其模式是否正确。

推理机首先执行在议程中有最高优先级的激活者的行为，然后是次优先级的激活者，这样直到没有激活者剩下。在专家系统外壳中，有各种优先级方式。一般，所有外壳都让知识工程师去定义优先级准则。

当不同的激活者有同一优先级而推理机必须决定其中哪一个被触发时，议程会发生冲突。不同的外壳有不同的解决方法。在早期的 Newell 和 Simon 范例里，那些首先进入系统的规则有最高的默认优先级，在 OPS5 里，对模式越复杂的规则给予越高的优先级。在 CLIPS 里，除非知识工程师分配不同的优先级给规则，否则它们有同样的默认优先级。

此时，控制返回到命令解释器的**顶层**（top-level），以便用户下达更多指令给专家系统外壳。顶层是用户与专家系统通信的默认模式，它的提示信息为“接受一条新的用户命令”，只有顶层才能接受新命令。

顶层是外壳的用户界面，通过它来开发专家系统应用。为了专家系统操作的便利，通常设计非常

复杂的用户界面。例如，为了一个制造车间的控制，专家系统可能需要用户界面以高精度、按位彩色模式来显示一个车间图表，警告和状态信息可能以模拟响铃和闪烁灯光的方式实现。事实上，花费在设计和实现用户界面上的努力比在专家系统知识库上的更多，特别是在设计原型时。根据专家系统功能的不同，用户界面可通过规则或可被专家系统调用的另一种语言来实现。例如，由 Ernest Friedman-Hill（Friedman Hill 03）开发的 JAVA 版 CLIPS，称为 Jess，能方便地调用 Java 类实现 GUI，且实现起来更加容易，因为 Java 语言带有很多支持这方面的对象。

专家系统的一个关键特征是解释机允许用户询问系统怎样得出某个结论和为什么需要某种信息。系统怎样得出某个结论这个问题在一个基于规则的系统中是容易回答的。因为激活规则的历史和工作记忆的内容能够保存在一个栈中。这一点在人工神经网络、遗传算法或其他相关系统中并不存在。尽管某些系统尝试提供解释能力，但不如专家系统清楚明白。复杂的解释机可允许用户问“如果……会怎样”类型的问题，以便通过假设去探索推理路径。

1.10 产生式系统

当今最流行的专家系统类型是基于规则的系统，基于规则的系统不是采用静态的断言来表达知识，而是以多个规则的方式说明在不同的情况下有什么样的结果。一个基于规则的系统包括 IF…THEN 规则、事实和一个解释器，该解释器根据工作记忆中的事实来决定哪条规则被激发。

基于规则的系统分为两大类：正向链和反向链。一个正向链系统从已知的初始事实出发，通过使用规则来得到新的结论或做出某种行动。反向链系统则从某个假设或要证明的目标出发，不断寻找能使假设成立的规则，这个过程可能产生新的子目标以便把大的问题分成更容易证明的小问题。正向链系统以数据驱动为主，而反向链系统是目标驱动的（Debenham 98）。在后面的章节中会有更多的相关讨论，也会给出一个反向链在 CLIPS 中是如何工作的复杂例子。

规则的流行主要有以下几个原因：

- *模块化特征*。规则使得知识容易封装并不断扩充。
- *解释机制*。通过规则容易建立解释机，这是因为一个规则的前件指明了激活这个规则的条件。通过追踪已触发的规则，解释机可以得到推出某个结论的推理链。
- *类似人类认知过程*。基于 Newell 和 Simon 的工作，规则似乎是模拟人类怎样解决问题的一个自然方法。规则的简单表示方式“IF…THEN”使得容易向获取他们知识的专家解释知识的结构。

规则的起源最早可追溯 20 世纪 40 年代，由于基于规则系统的重要性，有必要回顾一下它的发展，这将给你一个更全面的认识，为什么基于规则的系统对专家系统这么有用。

Post 产生式系统

产生式系统首先由 Post 在符号逻辑中使用，它证明了这一重要和令人惊奇的结果，即任何数学或逻辑系统都能被写成某种产生式规则系统。这一结论表明了大多数类型的知识都可以由产生式规则来表达。应用**重写规则**（rewrite rules），还可以定义语言的语法。计算机语言普遍使用 BNF 范式这一产生式规则来形式来定义像列在附录 D 中的 CLIPS。

Post 的基本思想是，任何数学或逻辑系统都只是一组特定规则，用来将一组符号转换成另一组符号。也就是说，提供一个输入字符串，即前件，产生式规则能够产生一个新的字符串，即结果。这一思想也适用于程序和专家系统，即初始符号串是输入数据，输出串则是转换后的输入。

举个简单的例子，比如输入串是“病人发烧”那么输出串将是“服用阿司匹林”。注意到这里的串并没有特别的含义。也就是说，串的使用是基于语法结构而非语义，你也不必懂得发烧，阿司匹林和病人所表示的意义。人们知道在现实世界里这些字符串所代表的意思，但是一个产生式系统只懂得将一个字符串转换成另一个字符串，这个例子的产生式规则如下：

```
前件→结果          人发烧了→服用阿司匹林
```

箭头表示从一个字符串到另一个字符串，我们可以用 IF…THEN 语句来解释，即如果发烧了，那么服用阿司匹林。

产生式规则还可包括多个前件。例如，

发烧了 并且
　　超过 102°F→看医生

连接的“并且”并不是字符串部分，它只表示规则有多个前件。

一个 Post 产生式系统包括一组产生式规则，如下：

(1) 汽车不能发动→检查电池
(2) 汽车不能发动→检查汽油
(3) 检查电池和电池坏了→换电池
(4) 检查汽油和没汽油了→加油

如果有字符串“汽车不能发动”，则规则（1）和（2）可以产生字符串“检查电池”和“检查汽油”。然而，并没有控制机制将这些规则作用于串，也许只有一个规则能被应用或者两个或者一个都没有。如果有另一个串“电池坏了”和串“检查电池”，则规则（3）将产生字符串“换电池”。

规则的次序并没有特别规定，这个例子的规则也能写成如下次序，但系统仍是同一系统。

(4) 检查汽油和没油了→加油
(2) 汽车不能发动→检查汽油
(1) 汽车不能发动→检查电池
(3) 检查电池和没电了→换电池

虽然 Post 产生式规则对建立专家系统的某些基本部分很有用，但对于编写实际的程序来说是不够的，其根本缺陷在于缺乏**控制策略**（control strategy）来指导规则应用，一个 Post 系统允许规则以任何方式作用于字符串，因为它没有规定如何运用规则。

举个例子，比如你去图书馆找某一本关于专家系统的书。在图书馆里，你从书架上查找你要的那本书，如果图书馆非常大，你得花很长时间找到这本书。即使你找到了一本书中关于专家系统的这一部分，但该书的下一部分可能会带你到另一个完全不同的部分，比如法国烹饪。如果你需要从第一本书中找到资料来确定你需要找的第二本书，情况会变得很糟，对第二本书的搜寻同样会花费你大量时间。

马尔可夫算法

使用产生式规则的另一个好处是 Markov 发现的，Markov 给出了一个产生式系统的控制结构。**马尔可夫算法**（Markov algorithm）就是把一组按优先级排序的产生式顺序作用于输入串。如果最高优先级的规则不适用，则尝试次高优先权的规则，如此类推。如果（1）最后的产生式规则不适用输入串，(2) 使用了一个终止产生式，则算法结束。

马尔可夫算法也适用于一个串中从左到右的子串，例如，产生式系统包含一条规则：

AB →HIJ

当输入字符串 GABKAB 时，产生新串 GHIJKAB。然后产生式再作用于新串，最后结果为 GHIJKHIJ

设∧代表**空串**（null string），则产生式

A → ∧

删除字符串中所有字母 A

有些特殊符号可表示任一单个符号，设由小写字母 a，b，c 等来表示，它们在专家系统中是一个重要的组成部分，例如，规则：

AxB → BxA

将互换 A 与 B。

希腊字母 α，β 等将用于表示字符串，使用它们是由于其不同于普通字母。

下面给出了一个使用马尔可夫算法把输入字符串的第一个字母移到末尾的例子，规则的优先次序为（1）最高，（2）其次，如此下去。

(1) αxy → yαx
(2) α → ∧
(3) ∧ → α

表 1.11 列出了对输入串 ABC 的执行顺序。

表 1.11 马尔可夫算法的执行过程追踪

规 则	成功与否	字 符 串
1	F	ABC
2	F	ABC
3	S	αABC
1	S	BαAC
1	S	BCαA
1	S	BCAα
2	S	BCA

这里 α 类似传统程序设计语言中的一个临时变量。但是，α 并不是用来存放一个值，而是用来标明一个位置，以使代换输入串的过程能进行下去。一旦完成任务，α 将通过规则 2 消除，当规则 2 被应用时，程序就结束，这是由于规则 2 是一个终止规则。隐马尔可夫模型（HMM）广泛使用于使用模式识别的应用中，例如语音识别。

Rete 算法

注意到马尔可夫算法具有明确的控制策略，它总是使用最高优先级的规则。只要最高优先级的规则被满足，就使用；否则，就尝试较低优先级的规则。虽然马尔可夫算法可作为专家系统的基本准则，但对具有大量规则的系统来说它是低效的。效率是一个最重要的问题，不管一个系统的其他方面如何好，如果用户响应时间长，则这个系统就不会被使用。我们需要这样一个算法，它不必去顺序尝试每一个规则。

这个问题的解决办法之一是 **Rete 算法**（Rete Algorithm），它是 1979 年由 Carnegie-Mellon 大学的 Charles L. Forgy 在其关于 OPS 专家系统外壳的博士论文中提出的。术语 Rete 来源于拉丁文 rete，是网的意思。Rete 算法通过在网络上存储规则信息来提高反应和规则激发速度，比起传统的一个一个检查大量的 IF…THEN 语句快得多。Rete 算法是一个动态的数据存储结构，类似标准的 B+ 树，可以自发优化查询。

Rete 算法是一个非常快速的模式匹配器，它是通过把有关规则的信息在存储器中以网络方式存放来获取速度的。Rete 算法通过限制一个规则激发后重新计算冲突集合的耗费来提高正向链规则系统的速度。其缺点是需要较多的存储空间，但在今天这已不成问题，因为存储器已非常便宜。Rete 利用下面两个经验总结，并把它们与其数据结构相结合。

时间冗余性——一个规则的激发通常只改动了少量事实，而每个改动只影响了少量规则。

结构相似性——相同的模式通常在不只一个规则的左边出现。

如果有成千上万条规则，那么计算机逐个去检查每条规则是否会被激发是非常低效的。Rete 算法使专家系统工具在计算速度低下的 20 世纪 70 年代可以投入实际使用。今天，Rete 算法仍对一个包含较多规则的专家系统的快速执行非常重要。

在每个识别动作循环中，Rete算法不是用事实去匹配每一个规则，而是仅仅考察那些有变化的匹配。由于在每一个循环中，那些没有变化的数据可忽略，因此大大提高了事实与前件的匹配速度。在CLIPS一章中，我们将进一步讨论这一点。快速模式匹配算法，如Rete算法等奠定了专家系统走向实际应用的基础。图1.7总结了现代基于规则的专家系统的一些基本技术。

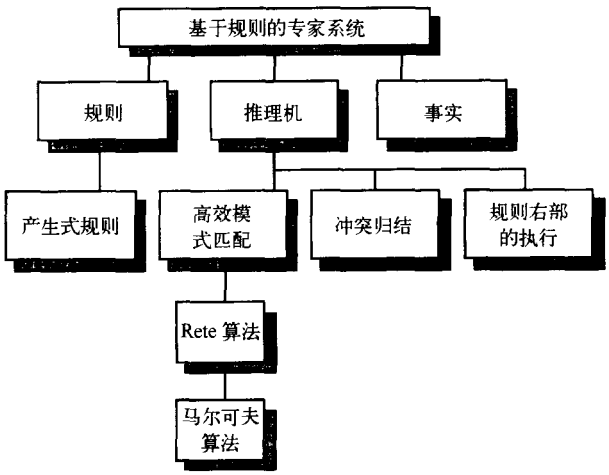


图1.7 现代基于规则的专家系统的一些基本技术

1.11 过程化程序规范

程序规范可分为过程化与非过程化。图1.8是语言的过程化规范的**分类**（taxonomy），图1.9是非过程化规范的分类。它们例示了其他规范与专家系统的关系，但只是一般导向而不是严格定义。例如CLIPS基于规则，但是也可以用CLIPS写出一个完全面向对象的专家系统，或混合了规则和对象的系统。一些规范与语言可以属于多种类别，例如，有些被认为是函数式而有些被认为是说明性。

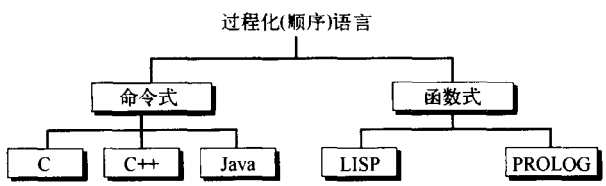


图1.8 过程化语言

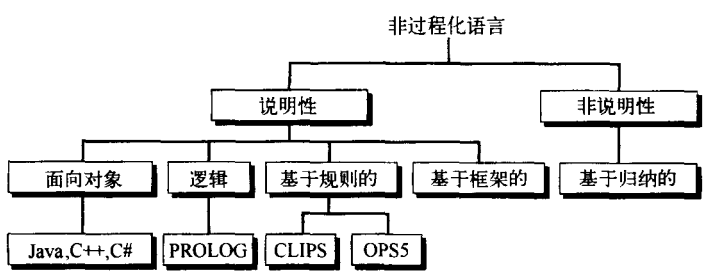


图1.9 非过程化语言

算法（algorithm）是一种在有限步内解决问题的方法。一个算法的程序实现就是一个**过程化程序**（procedural program）。术语算法程序、过程化程序以及传统程序经常交换使用，用于表达非人工智能

程序。过程化程序的一种通常定义是其按顺序执行。一个语句接着一个语句，直到遇到分支指令。过程化程序的另一个常用的代名词是顺序程序。然而，顺序程序这个术语当所有的现代程序语言支持递归时，其在应用上就受到很多限制。因为，这些程序不可能严格地按顺序执行。

过程化程序的一个独特特点是必须精确地说明一个问题的解决是如何编码的，甚至代码生成器也必须产生过程化代码。在某种意义上，代码生成器的使用是**非过程化**（nonprocedural）程序设计，因为它去掉了程序员所写的大部分或全部过程化程序代码。非过程化程序设计的目的就是只需程序员说明目的而让系统决定如何来实现它。

命令式程序设计

术语**命令式**（imperative）与**面向语句**（statement-oriented）是同义的。比如C语言就具有一种明显的特点，那就是语句是命令式的，它告诉计算机怎么做。在面向对象的C++语言中，对象可以被使用，对象之间可以传送信息。因此，一个面向对象程序的执行类似于一个事件驱动的系统，而不像一个命令驱动的程序流，从头到尾顺序执行。

命令驱动程序规范是基于图灵机和冯·诺伊曼寄存器和存储结构的机器的抽象。这类机器的核心是可更新**存储**（store）概念。变量和赋值是其在程序语言中的具体体现。存储是程序操作的对象。命令式程序语言具有丰富的命令以编码和操作存储。每一个命令程序语言都定义了一种特定的硬件视图。这些视图非常明确，就是通常所说的Pascal虚拟机或者Java虚拟机，其执行的字节码可以在任何硬件平台上运行。字节码的可移植性使Pascal在20世纪70年代和80年代获得巨大成功，直到Java使用了相同的系统。事实上，一个编译器就是实现了实际硬件和操作系统支持的程序语言定义的虚拟机。

在命令程序语言中，一个名字可以被赋予一个值，稍后又被赋予另一个值。所有名字和相关的值以及程序中的控制区域构成了**状态**（state）。状态是连接位置和值的存储的一个逻辑模型。在执行过程中，程序可以被描述为从一个初始状态通过一系列中间状态到达最终状态的过程。而这中间的状态变化取决于操作参数、输入和命令序列。

在冯·诺依曼体系中，命令式语言是程序员编写汇编语言的一种方式。命令语言非常支持变量、赋值操作及循环，这些都是低层操作，现代语言企图用递归、子程序、模块、包等来代替。命令语言还非常强调严格的控制结构并常常采用**自顶向下**（top-down）的程序设计。

所有语言都有这样一个重要的问题，那就是如何保证程序的正确性（在第2章中将详细讨论）。从人工智能的观点来看，另一个重大问题是命令式语言不是一种高效的工具。因为命令式语言是依据冯·诺依曼体系结构的，所以它非常适合数值计算而不适合于符号处理。然而命令式语言、C和面向对象语言C++、Java已被用作编写专家系统外壳的底层语言，这些语言和用它们所实现的外壳在一般通用机上的运行比用LISP实现的早期外壳要高效的多。

由于它们的顺序性，命令语言对于直接实现专家系统特别是基于规则的系统并不很有效。作为这个问题的说明，考虑一个具有成百上千条规则的现实世界问题。例如，DEC用于配置计算机系统的XCON系统目前其知识库具有7000条规则。早期使用FORTRAN、BASIC来编写代码，但不成功，直至使用专家系统方法才获得成功。一个专家系统工具，如CLIPS，使建立一个大型的基于规则的专家系统比使用第三代语言或面向对象语言Java、C++或C#更加容易。要想充分体验专家系统语言的优点，你得尝试编写代码。如何在CLIPS中编写代码将在本书的第2部分中讲解。

编写这些知识的代码在命令语言里需要7000个IF…THEN语句或一个很长的CASE语句，如果这样的话，效率将是一个主要的问题，因为在每一识别动作循环中，所有的7000个规则都需要匹配，而且推理机和所有的识别动作循环都得用命令语言来编写。而且情况比仅仅编写7000条规则复杂得多，因为许多规则将被其他的规则限制。例如，如果红灯时停车，这条规则仅当你遇到交通灯的时候应用，当你已经过了交通灯时便不再适用。在基于规则的系统中存在许多可能的交互。有一些规则证实了事实，有些则撤销了事实，有些则更改了事实。更复杂的规则可能产生新的规则或者在机器学习中删除了规则。

如果将规则排序，使得那些最可能执行的规则放在开头，则程序的效率将提高。然而，这需要对系统作大量调整、增加新规则，或删去及修改旧规则。一个提高效率的更好办法是建立规则模式树以减少决定哪个规则将被激活的搜索时间。规则模式树并不需要程序员构造，它可基于规则的 IF…THEN 结构让计算机自动建立。使用 IF…THEN 结构非常利于表达知识并进行高效的模式匹配，这需要开发一个语法分析器来分析输入结构并解释、编译、执行这种结构。

当实现了所有这些提高效率的技术，将会得到一个完整的专家系统，如果只要推理机、语法分析器以及解释器以便开发其他专家系统，则得到一个专家系统外壳。当然，你不必从头来进行这些开发，你可以使用一个现存的外壳，例如文档齐备，测试充分的 CLIPS。

函数式程序设计

正如 LISP 所体现出来的那样，**函数式程序设计**（functional programming）语言不同于面向语句的语言，面向语句的语言具有复杂的控制结构并采用自顶向下设计。传统语言通过概念上的限制使问题可以模块化。而函数式程序设计语言把这些限制推翻。函数式语言的 2 个特性，高阶函数、惰性求值，可以使模块化更加方便。其基本思想是组合简单函数来生成功能更强大的函数，这与命令式语言的**自顶向下**（top-down）设计不同，实际上是一种**自底向上**（bottom-up）的设计。

函数式语言以**函数**（functions）为中心，从数学上来讲，函数是从一个**域**（domain）到另一个**相关域**（codomain）的一种**关系**（association）或准则。下面是一个函数定义的例子：

cube(x)≡x * x * x，这里，x 是一个实数，而 cube 是一个具有实数值的函数。

这个函数定义的 3 个部分是：

(1) 关系 $x * x * x$

(2) 域，实数

(3) 相关域，实数

符号“≡”意谓着恒等于或定义为，下面是 cube 从一个域实数到另一个相关域实数映射的一种速写法，实数用符号$\Re$表示。

$cube:\Re \rightarrow \Re$

从域 S 到相关域 T 的一个函数 f 的一般表示方法为 f：$S \rightarrow T$。函数 f 的**值域**（range）是所有**映像**（images）$f(s)$ 的集合，其中 s 是 S 的一个元素。对于函数 cube，s 的映像是 $s * s * s$，值域是全体实体。这里值域与相关域是相同的。但这并不适用于其他函数，如平方函数，$x * x$ 域与相关域是实数，但其值域和相关域是负实数。

使用集合定义，函数的值域可被写成：

$R \equiv \{f(s) \mid s \in S\}$

花括号 { } 表示一个**集合**（set），竖线“|”可理解为“这里”。这样，上面的式子可理解为值域 R 等于 $f(s)$ 的集合，这里 s 是 S 中的一个元素。关系是有序对 (s, t) 的一个集合，其中 $s \in S$，$t \in T$，并且 $t = f(s)$，每一个 S 中的元素必须有且仅有 T 的一个元素与之对应。但是，多个 T 可能对应同一个 S，如对正数 n，其平方根值为 $\pm\sqrt{n}$。

函数可以递归定义，如：

```
factorial(n) ≡ n*factorial(n-1)
     where n is an integer and
     factorial is an integer function
```

这里，n 是一个整数，factorial 是一个整函数

递归函数通常用于函数语言如 LISP。

数学概念和表达式是**引用透明**（referentially transparent）的，因为其整体的含义完全取决于其组成部分，组成部分之间互相不影响。例如，函数表达式 $x + (2 * x)$，其结果是 $3 * x$，这两者是相同的，

不管 x 取什么值，甚至 x 是一个函数，其值也相同。例如，设 $h(y)$ 是一个任意函数，那么 $h(y) + (2 * h(y))$ 恒等于 $3 * h(y)$。

下面，考虑一个命令式语言，如C语言中的赋值语句：

```
sum = f(x) + x
```

如果参数 x 是引用型且其结果在函数调用 $f(x)$ 时改变，那么 x 取何值？这依赖编译程序是怎样写的，如果 x 的值在栈中保存，则可能是旧值，否则可能是新值。导致混乱的另一个原因是有的编译程序采用从右到左的次序计算表达式，而有的则采用从左到右的次序。在这种情况下，即使使用同样的语言，$f(x) + x$ 与 $x + f(x)$ 的值也可能不同。使用全局变量时也可能产生副作用。所以，与数学函数不同，程序函数的意义并不十分明确。

函数式程序语言在设计时，保证了其意义是明确的。一个函数式语言由5部分组成：

- **数据对象**（data objects），供语言函数操作
- **基本函数**（primitive functions），用于操作数据对象
- **函数格式**（functional forms），用于从函数生成新函数
- **应用操作**（application operations），作用于函数上，并返回一个值
- **命名过程**（naming procedures），用于标志一个新函数

函数式语言一般是解释型的，以便简化设计并实时响应用户。

在LISP（表处理）语言中，数据对象是**符号表达式（S-表达式，symbolic expressions）**，既可为**表（lists）**，也可为**原子（atoms）**。下面是一些表的例子，类似于CLIPS模式的编程。

```
(牛奶 鸡蛋 乳酪)
(购物(食品(牛奶、鸡蛋、乳酪)衣服(裤子)))
( )
```

这种类型的模式可以写在条件或规则**左部**（left-hand-side，LHS）中，表示事实或者一个列表，就如同购物单的例子。如果条件为真，则规则**右部**（right-hand-side，RHS）被执行并产生新的列表。其他的内容例如撤销事实等，也可以编写在规则右部中。表总是用配对的括号括着，其元素用空格分开。表的元素可能是原子，如牛奶、鸡蛋和乳酪，或者是子表如（牛奶、鸡蛋、乳酪）和（裤子）。表可拆分而原子不能，**空表（empty list）**，()，不包括任何元素，称为 **nil**。

LISP的早期版本称为“纯”LISP，因为它是纯粹函数式的。然而用它书写程序并不高效，为了提高书写效率，已加入了一些非函数化的东西。例如，SET是赋值操作，LET和PROG既可用于建立局部变量，又可执行一个S表达式序列。虽然这些操作类似函数，但它们并不是初始数学意义上的函数。

自从创立以来，LISP在美国一直居于人工智能语言的主导地位。由于它易于表达，许多早期专家系统外壳是用LISP建立的。但是，传统的计算机不能高效地执行LISP，对用LISP编写的外壳，情况更糟。当然，随着处理器和时钟周期的提高，LISP的效率也得到提高。

高费用使得开发和**发行**（delivery）成为一个问题。由于费用高，难以发行，因此就难以开发大型程序。一个好的开发平台，如果没有适宜的速度、功能、大小、重量、环境或价格，就将难以发行。一些应用甚至要求最后的代码放置于ROM就是由于费用和稳定的原因，把代码放入ROM将是困扰某些需要专门硬件运行的人工智能和专家系统工具的一个问题。不过，比起重写代码，这可能是一个好的选择。

另一个问题是在人工智能语言中嵌入传统的程序设计语言，如C、C++、C#和Java。例如对某些需要很多数值计算的应用，使用传统语言比用专家系统工具好。所以CLIPS使得用宿主语言开发自己的函数更加容易，可以从CLIPS的在线参考手册查询到更详细的内容（http://www.ghgcorp.com/clips/CLIPS.html）。

除非有特别的处理，用LISP编写的专家系统一般难以嵌入用其他语言编写的程序。选择AI语言的一个重要原因是它提供了一些工具。由于可移植性、效率、速度等原因，许多专家系统工具，现在

都用C语言编写或转换为C语言。这同时解决了原来的基于LISP的应用都需要昂贵的特制硬件的问题。

1.12 非过程化程序规范

非过程化程序规范不依赖于程序员给出求解问题的精确细节。与过程化规范不同，过程化规范着重于如何实现一个函数和一个语句序列，而非过程化规范则着重于实现什么并让系统决定怎样来实现它。

说明性程序设计

说明性规范（declarative paradigm）把**目标**（goal）与实现目标的方法分开。用户制定目标，而底层机制则尝试来满足这个目标。已建立了一些**规范**（paradigm）与相关程序语言用来实现说明性模型。

面向对象程序设计

面向对象（object-oriented）规范是一种既有命令式又有说明性程序设计特点的规范。术语面向对象用于程序设计语言C++、Java、C#等，其基本思想是设计程序时，把程序中的数据作为一个对象，然后实现这些对象上的操作。它与自顶向下的方法不同，自顶向下是逐步求精的。统一模型定义语言UML是流行的面向对象设计方法。

作为一个面向对象设计的例子，考虑用交互式菜单方式来编写管理账目的程序。重要的数据对象是当前的收支平衡、应收账、应付账。可定义各种操作作用于这些对象上，如增加应收、修改应付、增加月利润等。一旦定义了这些对象、操作、菜单界面，就可以开始编程。面向对象的设计方法特别适合于弱控制结构程序，它不适合于那些需要严格控制结构的程序，如工资表应用，对工资表应用程序。但面向对象程序设计在今天非常流行，因为它容易维护而且对象可以被重用。

术语**面向对象程序设计**（object-oriented programming）最早是指像Smalltalk这样的语言，它们专门为对象而设计。目前这个术语常常用于面向对象设计的程序，即使这种语言并不真正支持对象。

许多面向对象语言如C++、Java和C#本身具备支持对象的特性。Smalltalk起源于SIMULA 67，这是一种为仿真而设计的语言。SIMULA 67引入了**类**（class）的概念，继而导致了信息隐藏入模块中的概念。一个类是一种基本类型，是一个定义数据结构的模板。类的**实例**（instance）是一个能被操控的数据对象。实例这个术语已经被引入到专家系统中，它表示与模板相匹配的事实。类似地，如果一个规则的左部被满足，就称它被例化。在基于规则的系统中，术语被激活和例化是同义的。

另一个来自SIMULA 67的重要概念是**继承**（inheritance）。在SIMULA 67中，可定义一个**子类**（subclass）来继承类的属性。例如：可定义一个包含对象的类，这些对象可在栈中使用，再定义另一个复数类，有了这两个类，就很容易定义一个子类，其对象为可在栈中使用的复数。这些对象继承了上面这两个类的属性，这两个类叫**超类**（superclasses）。继承的概念使得能组织对象到一个层中，在这个层中，对象继承类，类又继承它上层的类，如此下去。由于对象能继承类的属性，而这又并不需要程序员去专门说明，因此继承的概念非常有用。但是多重继承的实现并不容易，所以Java和C#并不支持这个特性。CLIPS用C++编写，因此具备多重继承的特性。完全嵌入在CLIPS中的面向对象语言称为COOL（Common Object-Oriented Language）。COOL的操作对用户透明，所以用户只需要理解面向对象编程的特性。

逻辑程序设计

人工智能应用的一个最早程序之一是Newell和Simon设计的逻辑定理证明程序。这个程序于1956年在关于AI的Dartmouth会议上被公布并引起轰动，这是由于以前电子计算机只用于数字计算。现在计算机也能精确推理那些以前被认为只有数学家才能推导的定理。

在逻辑定理证明器以及它的后续版本——通用问题求解器（GPS）中，Newell 和 Simon 致力于实现能解决任何问题的计算算法。逻辑定理证明器只能证明数学定理，但 GPS 却被设计成能求解各种逻辑问题，包括游戏，智力难题如下棋、汉诺塔、传教士和野蛮人、猜谜等。一个著名的猜谜难题是：

```
  DONALD
+ GERALD
  ROBERT
```

其中已知 D=5。问题是求出其他字母的值，范围在 0～9 之间。

GPS 是第一个把求解知识与领域知识明确分开的问题求解程序，它已成为现今专家系统的一个基本范例。在现今的专家系统中，推理机决定什么知识应被使用并怎样去应用它。

人们继续致力于研究机器定理证明。到 20 世纪 70 年代初期，人们发现，计算只是逻辑推理的一个特例。当处理形如“如果条件则结论”的句子时，采用反向链推理非常易于定理证明。正如前面所讨论的产生式规则一样，条件代表所需匹配的模式。具有这种形式的句子成为 Horn 子句，它是 Alfred Horn 首先提出的。1972 年，Kowalski、Colmerauer 和 Roussel 创立了 PROLOG 语言，它使用 Horn 子句通过反向链推理实现了逻辑程序设计。

反向链既可以用说明方式来表达知识，又可以控制推理过程。反向链通过定义子目标来进行推理，如果要满足初始目标，这些**子目标**（subgoal）也必须满足。这些子目标经常分解成更小的子目标。

下面是一个说明性知识的典型例子：

所有人都会死
苏格拉底是一个人

这可用 Horn 子句表示为：

某人会死
　如果某人是一个人
苏格拉底是人
　如果(任何情况下)

后面一个句子在任何条件下，“if”条件都是真的，也就是说，关于苏格拉底的知识不需要任何模式匹配。而在前一个句子中，只有某人是一个人这个模式匹配，“if”条件才能满足。

注意，一个 Horn 子句能被解释成一个告知怎样满足目的的程序。那就是，决定某人是否会死，必须首先决定某人是否是人。下面是一个稍微复杂的例子：

汽车要有汽油、燃料和充气轮胎才能行驶。

用 Horn 子句表示为：

X 是汽车并能行驶
　如果 X 有汽油并且
　　X 有燃料并且
　　X 有充气轮胎

注意决定汽车是否能行驶这个问题将会缩小为 3 个更简单的子问题或子目标。现假定有下面的一些额外的说明性知识：

油箱计量器显示不空，如果汽车有汽油
量尺显示不空，如果汽车有燃料
空气压力计量器显示至少 20，如果汽车有充气轮胎
油箱计量器显示不空
量尺读数为空
空气压力计量器显示至少 15

这些知识能被转换成下面的 Horn 子句：

X有汽油
    如果油箱计量器显示不空
X有燃料
    如果量尺显示不空
X有充气轮胎
    如果空气压力计量器显示至少 20
油箱计量器不空
    如果(任何情况下)
量尺读数为空
    如果(任何情况下)
空气压力计量器显示至少 15
    如果(任何情况下)

从这些子句，一个机器定理证明器能证明汽车不会行驶，因为没有油和足够的空气压力。

反向链推理的一个优点是能并行执行。也就是说，如果有多处理器的话，它们可同时执行多个子目标。

PROLOG不只是一种语言，它还包含一个反向推理机。至少，PROLOG是一个外壳。因为它要求：

- 一个解释器或推理机
- 一个数据库（事实和规则）
- 一个模式匹配方法，称为**合一**（unification）
- 一个**回溯**（backtracking）机制，如果子目标搜索不成功的话

作为一个反向链推理的例子，设想如果你有现金或信用卡，你能买汽油而使汽车行驶。一个子目标去检测你是否有现金，如果你没有现金，回溯机制会探测另一个子目标，看你是否有信用卡，如果你有信用卡，买油的条件就满足了。注意缺少事实证明目标与具有相反的事实如“量尺读数为空”是等效的。相反的事实或缺乏事实都能导致目标不被满足。

如果问题不需要回溯和模式匹配机制，那么程序员可采用不同的语言编程。逻辑编程的一个优点是程序执行的说明化。也即，通过Horn子句说明问题的需求即可生成一个执行的程序。该程序不同于传统的程序，在传统的程序设计中，需求并不是最终的程序代码。

与产生式规则系统不同，在PROLOG程序中，子目标、事件和规则的次序并不起作用。效率（即速度）是由PROLOG搜索数据库的方式来决定。但是，有些程序，如果子目标、事件和规则以某种次序输入，则程序会执行正确执行，而如果以另外一种次序输入，则可能导致死循环或运行时错误。

专家系统

专家系统可以认为是一种说明性程序设计，这是因为程序员不必详细说明如何完成目标的具体算法。例如，在一个基于规则的专家系统中，任何一个规则，只要其左部与事实匹配，那它就可以被激活并被加入到议程中。规则的顺序并不影响它们的激活。这样，程序语句顺序也就不代表严格的控制流。还有其他类型的专家系统，我们将在第2章讨论基于**框架**（frames）的专家系统，在第4章讨论基于**推论网**（inference nets）的专家系统。

专家系统与传统的程序间有许多不同，表1.12列出了一些不同点：

表1.12 传统程序和专家系统的一些不同点

特　征	传统程序	专家系统
由……控制	语句次序	推理机
控制与数据	隐含在一起	明确分开
控制能力	强	弱

（续）

特　征	传统程序	专家系统
由……求解	算法	规则和推理机
求解搜索	少或没有	多
问题求解	算法的正确性	规则
输入	假设正确	不完整、错误
意外输入	难以处理	照样处理
输出	总是正确	依赖于问题的不同
解释	没有	通常有
应用	数值、文件和文本	符号推理
执行	一般是顺序	随机
程序设计	结构化设计	很少或没有结构
修改	难	较易
扩充	要作很大改动	可逐步增加

专家系统经常用于处理不确定性的问题，因为推理是处理不确定性的最好工具之一。不确定性既可体现在输入数据，也可体现在知识库中。使用传统程序设计的人可能会对此感到惊讶。然而，很多人类知识是启发性的，也就是说它只在有些时候才工作正常。此外，输入的数据可能会不正确、不完整、不一致，或者是其他错误。算法求解方法不足以处理这些情况，因为它必须保证在有限步内能解决问题。

对不同的输入数据和知识库，一个专家系统可以给出正确答案、好答案、差答案或者没有答案。乍看上去可能会觉得很惊讶，怎么会没有答案。不过，应记住的重要事情是一个好的专家系统并不比一个专家差，甚至会做得更好。如果我们能找到比专家系统更有效的计算方法，我们会使用它。最重要的是，使用最好的处理该项工作的工具。

非说明性程序设计

非说明性程序设计规范在 PROLOG 和 SQL 中越来越流行。PROLOG 的新版本专为 AI_1 开发而 SQL 是关系数据库基础。这些新的规范应用已越来越广泛。它们既可单独使用，也可与其他规范联合起来使用。

基于归纳的程序设计

作为人工智能的一个应用，**基于归纳**（induction-based）的程序设计，例如机器学习的经典 ID3 算法和新的 C4.5 和 C5.1，已越来越引起人们的兴趣。在这种规范中，程序通过例子来学习。这种方法已用于数据库存取，它不要求用户输入一个或几个要搜索的字段值，而是要求用户选择一个或几个具有该特征的相近事例。然后，程序在数据库中查找与该特征相匹配的数据。Oracle 和其他使用 SQL 语言的关系数据库管理系统是使用模式匹配查找的典型例子。

某些专家系统工具提供了归纳学习的功能，它们可通过接受事例来自动生成规则（参看附录 G）。

1.13 人工神经系统

在 20 世纪 80 年代，程序设计规范有了一个新的发展方向，即**人工神经系统**（ANS，artificial neural systems），它基于人脑如何处理信息。这种规范有时也称为**连接**（connectionism）系统，这是因为它求解问题的模型是通过训练在网络中相连的模拟神经元。许多研究人员都在从事神经网络方面的研究工作。今天人工神经系统大量活跃于各种应用包括人脸识别、医疗诊断、游戏、语音识别和汽车发动机。

连接研究者推测用人脑的工作方式来模拟计算比用数字计算机模拟更能深入智能行为的本质。隐藏在这种连接背后的基本思想是，如果我们从人脑方式的计算而不是基于规则的符号操作来看问题，我们有可能取得人工智能的显著进展。神经网络是一种基于生物神经系统，如人脑处理信息模式而建立的信息处理技术。神经网络的原理基础是结构化的信息处理系统。神经网络通过把许多相互高度连接的处理元素或神经元结合起来，能够以类似人类的技术通过学习例子来处理问题。对特定的问题，例如数据分类或模式识别，神经网络通过某种学习过程，称之为**训练**（training）来进行处理。如同生物系统一样，学习包括神经元间的连接修正。

有很多种方法可对人工神经系统进行分类，一个有用的方法是根据是否提供了输入和输出数据训练集来分类。如果提供了训练集，那么人工神经系统称为一个引导模型。如果是在不知道输出的情况下对输入进行分类学习，那么称为非引导型。人脸识别是一个很好的引导型人工神经系统例子。相反，如果你不知道输出数据是什么，人工神经系统也能很好地分类，如识别疾病的爆发。

不同的神经网络，其神经元间的连接方式、神经元计算方式、网络传播方式以及其学习的方式和效率可以不同。神经网络已经被应用于真实世界的许多问题。其最主要的优点是可以解决对于传统技术来说太过复杂的问题——没有算法可以解决或解决的算法太过复杂。通常，神经网络很好地适用于人们善于解决但无法解释如何做到的问题。这些问题包括模式识别和需要识别出数据趋势的预测预报。今天，**数据挖掘**（data mining）领域充分利用人工智能技术，从历史数据中找出模式以协助公司决定未来策略。例如，数据挖掘可以用来告诉一个公司因为季节变化在何时存储某类产品。

神经网也用于需要实时反应的从大量传感器获得输入的专家系统前端。超过 50 个免费的人工神经系统可以从新闻组 comp.ai.neural-nets 的 FAQ 中下载，其他的资源列在附录 G 中。

旅行售货者问题

ANS 在对复杂模式识别问题提供实时响应方面获得显著成功。例如，在 20 世纪 80 年代采用神经网络方法在一台普通微机上只需 0.1 秒就能获得一个好的解，而采用其他最好方法在大型机上也需一个小时。旅行售货者问题是重要的，因为它是电信系统中优化信号路线的一个典型问题。优化路线能减少运行时间，这样就提高了效率和速度。

基本的旅行售货者问题是计算给定城市间的最短路线。表 1.13 显示了从 1 到 4 个城市的可能路线。注意，路线数目是城市数目减 1 的阶乘，即 (N－1)!

表 1.13 旅行售货者问题的路线

城市数目	路　线
1	1
2	1－2－1
3	1－2－3－1
	1－3－2－1
4	1－2－3－4－1
	1－2－4－3－1
	1－3－2－4－1
	1－3－4－2－1
	1－4－2－3－1
	1－4－3－2－1

当有 10 个城市，则有 9! = 362880 条路线。当有 30 个城市则有 29! = 8.8E30 条路线。旅行售货者问题是一个典型的**组合爆炸**（combinatorial explosion）例子，因为可能路线的数目增长很快，使得对实际的城市数目没有现实的求解方法。如果在一个大型机上需要一个小时的 CPU 时间去求解 30 个城市的路线问题，则需要 30 个小时求解 31 个城市，330 个小时求解 32 个城市的路线问题。较之成千个电信交换开关以及真正所使用的城市数目，这实际上是非常少的一个数目。

一个神经网络解决 10 个城市案例的速度与解决 30 个城市的速度一样，但传统的计算方法则要花费更长的时间。对 10 个城市案例，神经网络会给出两条最好路线中的一条，而对 30 个城市，它会给出 100 000 000 条最好路线中的一条。这个结果是令人满意的，因为它比 1E22 条路线都要好。虽然神经网络并不能常常给出最优解，但它为实时响应提供了一个好的方案。在许多情况下，在一毫秒内给出一个 99.99999999999999999%正确的答案要比在 30 小时内给出一个 100%正确的答案要好。其他使用遗传算法解决问题的方法以及演变人工算法（Evolutionary Art Algorithm）

(Dorigo 04) 在附录 G 中。另一种方法是真实 DNA (Sipper 02)。

ANS 的单元

ANS 本质上是一个使用以高速并行方式相连的简单处理单元的模拟计算器，这些处理单元对它们的输入执行简单的布尔或算术运算操作。ANS 的关键之处在于与每个单元相联的**权值** (weights)。这些权值代表存储在系统中的信息。

一个典型的人工神经元如图 1.10 所示。神经元可有多个输入，也可只有一个输入。人脑大概包括 10^{11} 个神经元。每个神经元与其他神经元可能有数千种联系。神经元的各个输入信号与权值相乘，然后累加成为神经元的总输入。所有权值可用一个矩形来表示。

神经元的输出常常是输入的反曲函数 (sigmoid function)。反曲函数能够描述真正的神经元，当输入非常小和非常大时，它都能达到极限。反曲函数也称为**激活函数** (activation function)。常用的一个反曲函数是 $(1+e^{-x})^{-1}$。每个神经元还有一个相连的**阈值** (threshold value) θ，每个总输入都要减去阈值。图 1.11 显示了一个能计算异或 (XOR, exclusive-OR) 的 ANS，它采用了反向传播技术。所谓异或就是只有在输入不全为真或不全为假时，其值才为真。隐含层结点的数目根据应用和设计会有所不同。

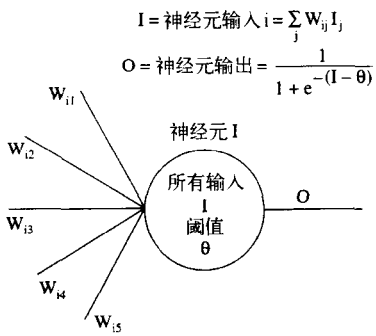


图 1.10 神经处理单元

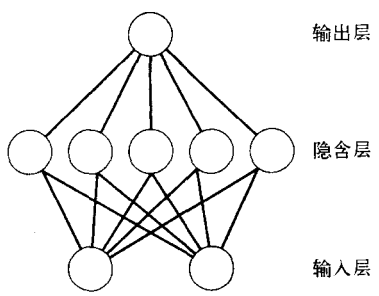


图 1.11 反向传播网络

神经网络一般不采用传统的方式编程。训练神经网络的学习算法有很多，如**对向传播** (counter propagation) 和反向传播算法等。程序员编程设计神经网络时，只需提供输入和相应的输出数据，网络通过调整相连神经元的权值来自动学习。神经元的权值和阈值决定通过网络数据的传播，使之能正确符合训练数据。训练神经网络可能需要几小时或几天，这取决于网络要学习的模式数、硬件和软件。然而一旦网络已训练好，网络反应将会很快。

如果用软件模拟不够快，ANS 还可以直接固化在芯片上以便实时响应。一旦网络训练好，权值已经确定，就可以构成芯片。

ANS 的特点

ANS 结构不同于传统的计算机结构。在传统计算机中，用存储单元来连接离散信息。例如，可在连续的存储单元中通过 ASCII 码来存放一个社会保险号。通过访问这些连续单元的内容，可直接得到一个社会保险号。这种方法是可行的，因为社会保险号与存储单元里的 ASCII 码都具有一个挨一个的关系。

ANS 是现今人脑理论的模型化，它用权值来表示信息。但是，在这种存放信息的权值之间并没有直接的联系。信息的这种分布表示类似于全息摄影，当激光扫过时，通过光的衍射作用，它可以重现所存储的影像。

当有大量经验数据并且没有保证足够精度和速度的算法时，神经网络是一个好的选择。与传统的

计算机存储相比，ANS有下面一些优点：

- 容错性（fault tolerant）。即使部分网络丢失，对网络存储数据的影响也不大。这是因为信息以分布的形式存储。
- 存储图像的质量降低并不与网络丢失数目成正比。信息不会受到灾难性丢失，存储质量具有全息摄影的特点。
- 数据以联想记忆方式存储。所谓联想记忆方式就是部分数据足够提供全部存储的信息。这与传统存储不同，其数据是通过特点地址确定的。部分或噪音输入仍可导出全部原始数据。
- 网络能根据其存储信息进行推断。训练使得网络能够发现数据的特征及其关系。据此，网络能够推断新数据的可能关系。在一个实验中，神经网络根据24个假设成员进行家庭关系的训练。之后，它能正确回答出没有训练过的关系。
- 网络有适应性（plasticity）。即使有些神经元丢失，只要剩有足够神经元，网络就能重新训练而达到它的初始水平。这也是人脑的一个特点：即使部分损坏，通过一定时间的再学习，也能恢复原来水平。

这些特性使得ANS在太空机器人、油田装置、水下设备、处理控制和其他一些需要长期处于难以恢复的恶劣环境中的应用方面，具有很大的吸引力。除了可靠性好外，ANS因其适应性也具有低维护费的潜力。虽然可做硬件修复，但重编辑一般比替换硬件要合算。

ANS一般不适用于精确计算或要求最优解的应用。同样，如果存在一个实际的求解算法，ANS也不是一个好的选择。

ANS技术的发展

与连接最密切的思想是在心理学理论中我们必须“严肃对待大脑”。这个思想源于古希腊的思考者在神经系统中联系动物的精神行为。后来，在笛卡儿的《*Treatise of Man*》一书中记载了著名的推断。ANS最早起源于1943年McCulloch和Pitts开始的关于神经元的数学模型。1949年Hebb给出了神经元学习的一个解释。在Hebb的研究中，神经元使用**触发**（firing）来刺激另一个神经元。触发指神经元发出一个电脉冲来刺激与其相连的神经元。这样神经元之间的传感连接，即**神经键**（synapses）就通过触发来引起。在ANS中，神经元间的连接权值变化即是模拟自然神经元之间的传感变化（Swingler 96）。

1961年，Rosenblatt出版了一本非常有影响的书，这本书阐述了他所从事的一项研究工作，该项工作是研究一个称之为**感知机**（perceptron）的新型人工神经系统。感知机具有学习和模式识别功能。它基本上由两层神经元和一个简单的学习算法组成。权值必须人工赋值，这一点与现代ANS中通过训练来设置权值不同。在20世纪60年代，许多研究者涉足ANS领域并开展对感知机的研究。

1969年，Minsky和Papert出版了一本书《*Perceptrons*》，该书指出了把感知机作为一种通用计算器的理论局限，这标志着早期感知机研究时代的结束。他们指出了感知机的缺点：它只能计算16个基本逻辑功能的14个，这意味着感知机不能作为一个通用目的计算器。特别是，他们证明了感知机不能识别XOR，虽然他们没有深入研究多层ANS，但他们认为，多层ANS也不可能解决XOR问题。政府基金停止了对ANS的资助，转而资助AI中使用LISP等语言的符号推理方法。20世纪70年代，由Minsky发明的一种新的框架表示法开始流行。框架又发展为现代的脚本。由于其简单性，感知机和其他ANS易于用现代集成电路技术构造。

在20世纪70年代，ANS的研究继续在小范围内进行。70年代末，Geoffrey Hinton、James McClelland、David Rumelhart、Paul Smolensky和其他并行分布式处理研究组成员致力于神经网络的理论认知。1986年发表划时代的书籍《*Parallel Distributed Processing: Explorations in the Microstructure of Cognition*》，标志着认知学的重大理论突破。Hopfield用Hopfield网使ANS建立在坚固的理论基础上，并证明了ANS如何解决不同的问题。Hopfield网的基本结构如图1.12。特别地，Hopfield说明了ANS在常数时间里怎样解决旅行售货者TSP问题，这在传统的求解算法中是一个组合爆炸问题。一个ANS电

路能在1微秒内解决TSP问题。其他的组合问题可用ANS轻易解决，如四色图、欧几里德匹配和转换代码问题。

ANS中的**反向传播**（back-propagation）网络很容易解决XOR问题，这就是著名的**一般化增量准则**（generalized delta rule）。即使附加层数能够指定，反向传播网络一般作为一个三层网络实现。在输入层和输出层之间的层称为**隐含层**（hidden layers），这是因为只有输入层和输出层对外界可见。另一流行的ANS是1986年Hecht-Nielsen发明的对向传播模型。从数学上得出的一个重要理论结果——Kolmogorov定理，可以证明，任何一个具有 n 个输入、隐含层具有 $2n+1$ 个神经元的三层网络能以任意精度逼近一个连续函数。

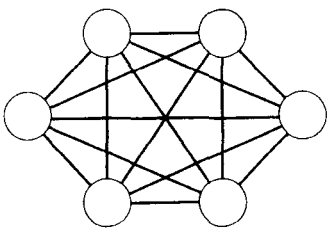


图1.12 Hopfield人工神经网

ANS技术的应用

使用反向传播学习算法的一个典型神经网络例子是学习课本中单词的正确发音。ANS通过训练用来校正DEC语音设备DECTalk的输出。设计校正DECTalk发音的规则需要20年的语言学研究。ANS仅仅通过听正确的课文发音来自学发音技巧，在ANS中，没有用到一点语言学知识。

通过光电计算机来识别雷达目标的ANS研究正在进行中。由光元件组成的光计算机实现的新的ANS，其速度要比用电子计算机实现的快上百万倍。ANS的光实现非常诱人，因为光具有并行性，即光线在传播时不互相干扰。大量的光子很容易用光元件如平面镜、透镜、高速可编程空间光调制器，多维光稳定设备等来生成和操纵，而且这些光元件还可作为光神经元和衍射光栅等使用。ANS的更进一步发展肯定会出现。

经典ANS应用引起广泛讨论（Giarratano 90a）。ANS特别适于那些用传统方法达不到满意效果的控制系统。(Giarratano 91b)。事实上，ANS广泛应用于许多工业控制系统（Hrycej 97），附录G列出的一些链接显示相关信息。

1.14 专家系统与归纳学习的关系

用ANS建立专家系统是可能的。ANS曾作为一个知识库，该知识库通过训练有关疾病的医学事例来构成。在这个系统中，专家系统根据训练所获得的症状来分类疾病。推理机也称MACIE（Matrix Controlled Inference Engine，阵列控制推理机）是用ANS知识库设计的，系统用正向链方式来进行推理，用反向链方式来询问用户所需的数据。虽然ANS自身不能解释权值是如何设置的，但MACIE能解释ANS并生成IF…THEN型规则来解释它的知识。

上面所述的这类ANS专家系统使用了**归纳学习**（inductive learning）。即，系统通过知识库中的事例来进行**归纳**（induces），归纳是从特殊推出一般的过程。除了ANS外，还用许多商用专家系统外壳从事例中生成规则，参见附录G中的讨论。归纳学习的目的是减少或消除知识获取瓶颈。通过把知识获取的任务交给专家系统，如果系统能够归纳出人们所不知道的规则，则既可以减少开发时间，又可以提高可靠性。现在，专家系统与ANS已经结合起来（Giarratano 90b）。

1.15 人工智能的发展状况

20世纪90年代以及21世纪，人工智能取得了许多进展。强人工智能观点，即认为只有逻辑和推理才能产生好的人工智能，在封闭的实验室环境外取得的进展不大。能适应残酷现实和竞争市场环境的成功人工智能系统多是基于生物学的系统。图1.13显示了从20世纪50年代以来人工智能的发展，图中的垂直线表示时间。在最初，人工智能分成两大类，基于模型的主要为符号逻辑，另一个大的分支为生物学处理方法。在这两个分支之间存在巨大的竞争，但是基于构造人工智能解决方案的经验，

现在我们可以说没有一种方法可以对一个困难的问题给出正确的答案。我们最高的期望只能是一个优化的解决方案，并且满足于一个好一点的解决方案。

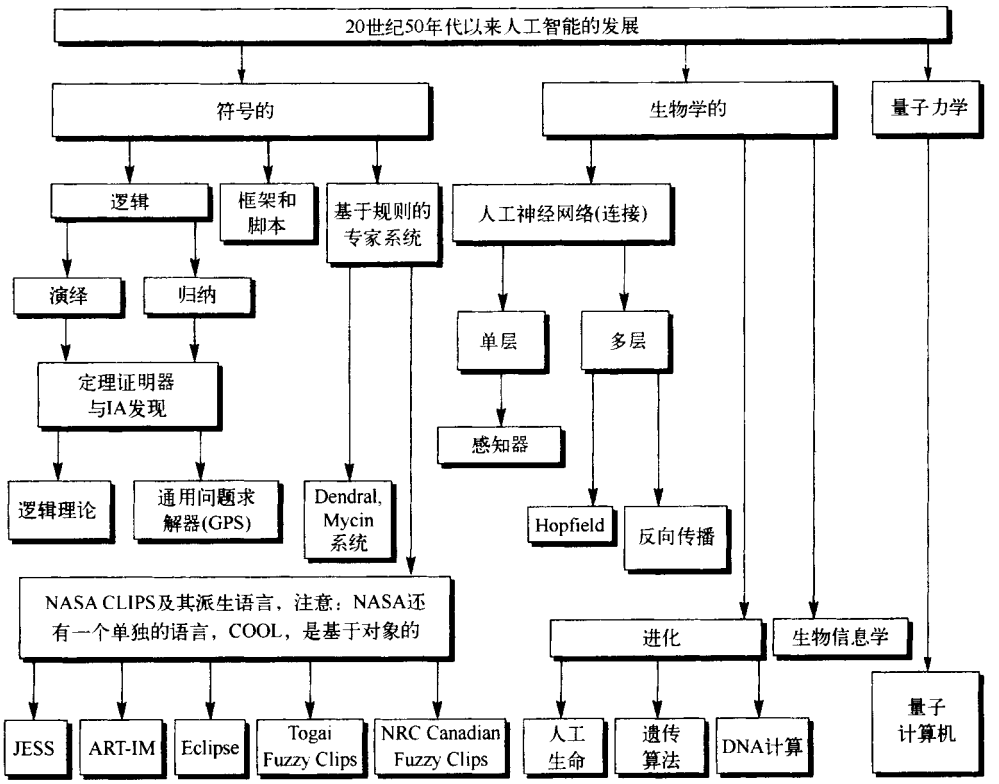


图 1.13 人工智能发展

在图 1.13 的右边显示了基于物理学，特别是使用量子计算机的新方法。在今天，我们仅是希望通过量子计算机来提高查找速度。从量子力学在 20 世纪基于 Schroedinger 等式和它的等价形式 Heisenberg 矩阵力学创立以来，许多人，包括爱因斯坦并不接受。爱因斯坦开玩笑说："上帝不会同宇宙掷骰子"。然而，量子力学不可避免地和概率紧密联结在一起，因此和经典的牛顿定理不同，没有什么是确定的。事实上，有一种理论认为意识是一种量子力学现象（Satinover 01）。

许多作者推测感知和意识本身可能是大脑的一个重要功能，而大脑由神经元组成，最终分解为原子和亚原子直到量子。它们之间的距离很小，10^{-60}米，而空间本身是纹理状的，不是数学和经典牛顿物理学所假设的平稳状态。物理学家推测这样小距离的空间不是通常的 3 维，而是 11 维。正如 2500 年前古希腊哲学家已经可以做到的一样，我们相信任何事情都可以用人类推理和思考的方式解决。如果忽略现实世界胡乱思考，可能得到以下结论：

前提：我没有移动，因为我没有感觉到自己在移动
前提：我看见太阳升落
所以，太阳绕着我运动。

在讨论进化算法时，术语**突现**（emergence）有一个特别的含义，用来表达一个事先未预料的行为不期望地发生。例如，在小溪中流动的水，看起来流动很平滑一直到水中出现石头。湍流的出现取决于水速、石头的大小和其他因素。通过湍流、退潮等其他不是由简单流体动力学而需要 2 维或更高维公式表达的动态行为，水展示了非直线的运动。这些公式的求解通常是很困难的，所以近似数值解是可

接受的最佳解答。许多重要的突现行为已被发现和应用于城市、管理和其他众多领域中（Johnson 01）。

突现行为的一个经典例子是蚁群，这导致了一种分布式智能技术，称为**群体智能**（swarm intelligence）（Kennedy 01）。这与人类和其他哺乳动物所展现的**集中智能**（centralized intelligence）不同，虽然集中智能在群体类型的昆虫中也应用得很好。人们正在研究群体智能的应用以便在其他星体中建立健壮的机器人群体，在这些地方，单个机器人的智能是不够的，但整个群体可以达到，而且还可节省能源和费用开销。群体智能在工业、网络和其他分布式系统的应用也显示了远大前景。事实上，因特网本身可以看作是群体智能的应用，因为它用分布的低层智能进行包分组，而用高层智能进行路由选择。这也正符合了20世纪60年代最初因特网（称为Arpanet网）出现的目标。

人工智能的一个新的主要方向是进化算法（Fogel 03），通常与其他人工智能技术如ANS一起使用，以提高结果的可靠性。ANN、遗传算法和其他类似方法的问题在于最后解容易陷入局部最优而得不到全局最优解。基于这个原因，有些方法保留那些“不适应”的解而不是全部采用最适应的解进行下一代的复制，以跳出局部循环。

比起取消或减少假设，研究连接系统是优化ANN结构的有效方法（Kasabov 02）。这些并行处理技术充分利用了计算机的功效且更具备可行性，因为现代计算机的计算能力越来越强大，而且互联网和其他私有公司也提供了连入许多计算机进行计算的接口（Foster 03）。

严格来说，计算机中的进化算法并不是基于达尔文进化理论，而是Lamarck的竞争理论。达尔文进化理论中，千万年才迈出进化的一小步。Lamarck是19世纪达尔文学说的支持者，他相信动物通过练习可以拥有一定限度内的强大力量，而这个动物的后代可以有更强大的分支。我们通过使用计算机来选择与我们的适应值标准最为接近的后代以使进化处理符合Lamarck的理论过程。

进化算法已经被作为计算机程序的一部分用于创造人工生命。但遗传算法或神经网络解决问题的领域并不仅限于此。它们还可用于广泛的领域，如生态学、经济学中研究野生肉食动物食物供应链关系，或更复杂的任何环境下的如全球经济中的生产消费关系。影像游戏也充分利用了环境算法来为用户定义新的虚拟生物。一个经典的例子是功能强大的游戏“The Sims”，以及其可供全球许多人共同参与的在线版本。

当在连接系统中使用进化技术时，计划外的意外事故可能会发生。不过，这只是增加了那些支持基于逻辑的人工智能者的批评。ANN算法不能解释其连接规则和权值是如何获取的。尽管许多人试图让ANS可以解释这些规则，事实证明这些规则和权重不是由人来规定的，而是对输入的反馈，以满足系统输出的需要。从这个意义上说，ANN或进化算法能产生正确的解决方案，否则它不会存在。

随着人类基因组的解码，**生物信息学**（bioinformatics）开始越来越重要。因为它使用计算机技术来处理巨大数量的信息。许多问题开始呈现在人工智能工作者面前。同时也产生了新的学科领域：**基因组学**（genomics）和**蛋白质组学**（proteomics）。人类基因组由大约30 000个基因组成。每个基因是一个生产携带机体功能蛋白质的工厂，蛋白质在细胞中起作用或在细胞间传递。目前的挑战是如何界定每一个基因和蛋白质，不仅仅是正常的也包括病变的基因。由于数据和可能解数目巨大，这些任务需要极大的计算能力。人工智能正是一个合适的工具。

人工智能在军事上也显示了其功效。1990年第一次海湾战争结束时，在军队中使用人工智能使国防部高级研究项目组（Defense Advanced Research Projects Agency，DARPA）从20世纪50年代开始对人工智能研究的资金投入（http：//www.au.af.mil/au/aul/school/acsc/ai02.htm）得到了成倍的回报。今天，人工智能通过计算机游戏模拟军队训练节省了大量花费，因为相对于真实的战争来说，它不需要燃料和军火。

在20世纪50年代难以想像人工智能可以在商业上取得如此成功。那时候以及接下来的四分之一世纪，人工智能研究者的重点在于寻找重要的、严肃的应用如定理证明和自动发现。尽管同时也使用人工智能程序来下西洋棋、象棋和玩其他游戏，但人工智能程序是用于严肃认真的目的如理解人类推理，而不是娱乐应用。

今天，人工智能在视频游戏的商业成功和电影中的特殊作用对于最初的研究者来说是难以想像的。视频游戏公司与电影和音乐共同竞争消费者市场。人们喜欢玩，特别是和不同国家、城市、地区的人一起玩交互游戏。先进的视频游戏高度依赖人工智能来使生物行为自然化，而不是简单的预定义模式。

事实上，越来越多的大学提供视频游戏课程以及学分，因为存在巨大的商业利益和工作机会。人工智能也在商业中发挥作用，最典型的如基于知识的筛选助手可以剔除 90% 的问题。具有专业知识的专家系统也应用在其他大量商业行为中。唯一的问题是，这些系统基于公司的利益信息，所以公司一般不会公布其系统细节以避免其竞争对手获益。

不过，在因特网上仍然有许多附录 G 中所列出的专家系统例子，大部分在万维网上，有一些在网站 CiteSeer 的文章里，还有一些在新闻组 comp.ai.shells 中，如 CLIPS 专家系统就在其中有很多讨论。新闻组的优点在于你可以发表问题期待别人回答。

除了人工智能和专家系统，**本体工程**（ontological engineer）也提供了一个新的工作机会。它可以看作是哲学工程的一个分支。在人工智能和专家系统中，本体（ontology）有不同于传统哲学意义上的含义。

本体有着明确的形式规范，其间存在着相关关系（Gruber 93）。本体广泛存在于网页中，尽管普通的网页使用者并没有意识到这一点。从巨大的组织分类本体，例如垂直的从上到下分类的组织相关信息的网站 Yahoo，到像 Amazon.com、eBay 和其他按照价格或拍卖时间及招标时间进行目录组织的网站。基本上，本体是一个标准的、符合某个规范的描述术语集，用来描述一个领域，无论是进行预订、拍卖或者其他内容。如果没有一个通用的词汇表，没有人知道在讨论领域中的是什么内容。许多组织为他们自己的领域开发本体以便更好地以明确的方式表明领域中讨论的内容。

一个标准化本体的最大优点之一在于它是机器可识别的，因此计算机可以帮助人类查找所需要的条目。一个类似的工作是为不同的领域定义扩展标记语言 XML。今天，如果要想成为一个本体工程师，为大量的知识分类，就必须具有人工智能背景。本体工程也应用于建立和维护一个专家系统的数据库。这不是一个微不足道的工作，特别是当知识库越来越大，并且允许用户输入新的知识为专家系统所用的时候。

1.16 小结

在这一章，我们回顾了专家系统的发展和一些问题。专家系统所解决的问题一般用传统程序难以解决，因为这些问题缺乏一个有效的算法。由于专家系统是基于知识的，它能有效解决现实中用别的方法难以解决的非结构化难题。同时，还讨论了一些知识表示规范以及其优缺点，关于这一点，在（Giarratano 04）中有更详细的讨论。

在选择专家系统所适宜的应用问题领域这部分内容中，我们讨论了专家系统的优缺点，同时也给出了选择适宜应用的标准。

以基于规则的专家系统为例，我们讨论了专家系统外壳的基本组成。通过一个简单的规则例子，我们描述并说明了基本的推理机识别动作循环。最后，描述了专家系统与其他程序范例的关系。所有这些最重要的一点是：专家系统是一种不同的程序设计工具，它适合于某些应用，而不适合于另外一些应用。以后章节将会更详尽地描述专家系统的特征与适用性。专家系统的优缺点也将在介绍如何为专家系统选择合适的领域时讨论。

正如计算机用户通过使用网格计算来完成大量计算功能，生物实验室利用 DNA 链，未来人工智能的发展将毫无疑问地使用量子计算机（Brown 00）。这两种方式可以在几天内解决非常复杂的旅行售货者问题，这比任何一个单一的超级计算机快得多。

附录 G 列出了许多人工智能的现行信息链接。人工智能信息和软件的最好资料来源是互联网。其他相关的新闻组有模糊逻辑、神经网、专家系统外壳等。除了可获取信息和软件外，这些新闻组还允许用户提出问题并得到来自于其他用户的答案。你可以按照章节上网浏览附录 G 中列出的资源。

习题

1.1 找一个专家或学者并拜访他。然后，根据“专家系统的优点”这一节的内容讨论，把该专家的知识模型化有哪些好处?

1.2 (a) 写出 10 条主要的规则来表示习题 1 中专家的知识。

(b) 编写一个能够给出专家建议的程序，测试上面的 10 条规则是否能给出正确的建议。为了简化编程，你可以菜单形式让用户输入。

1.3 (a) 在 Newell 和 Simon 的《*Human Problem Solving*》书中，他们提到一个九点问题。给定如下排列的 9 个点，画出 4 条直线通过这些点并满足（a）笔不能离开纸（b）不在任一点交叉。

. . .

. . .

. . .

(b) 解释你是如何找到求解方案的（如果可能的话），并讨论专家系统和其他程序设计方法哪个更适于求解该类问题。

1.4 编写一个程序求解密码问题。给出下面问题的结果，这里 D=5

```
  DONALD
+ GERALD
  ROBERT
```

1.5 写一组产生式规则集，来区别 5 种不同的燃料，如汽油、化学物质等，设 5 种燃料的类型已给定。

1.6 (a) 写一组产生式规则集，根据不同的症状来诊断 3 种类型的病毒。

(b) 修改上面程序，使得一旦病毒类型被确定，可以给出治疗建议。

1.7 给出 10 个规划假期的启发式 IF…THEN 型规则。

1.8 给出 10 个购买二手车的 IF…THEN 型启发式规则。

1.9 给出 10 个安排课表的 IF…THEN 型启发式规则。

1.10 给出 10 个购买汽车的 IF…THEN 型启发式规则。

1.11 给出 10 个购买证券或股票的 IF…THEN 型启发式规则。

1.12 给出 10 个为拖欠作业辩解的 IF…THEN 型启发式规则。

1.13 写一个关于现代专家系统的报告，资料来源可参考 *PCAI*，*IEEE Expert* 杂志，以及附录 G。

参考文献

(Adami 98). Christoph Adami, *Knowledge Introduction to Artificial Life*, Springer-Verlag, pp. 5, 1998.

(Bentley 02). Peter J. Bentley et al., *Creative Evolutionary Systems*, Academic Press, 2002.

(Berlinski 00). David Berlinski, *The Advent of the Algorithm*, Harcourt, Inc., pp. xvix, 2000.

(Brown 00). Julian Brown, *Minds, Machines, and the Multiverse*, Simon & Schuster, 2000.

(Bramer 99). Ed. by M.A. Bramer, *Knowledge Discovery and Data Mining*, IEEE Press, 1999.

(Cotterill 98). Rodney Cotterill, *Enchanted Looms*, Cambridge University Press, pp. 360, 1998.

(Debenham 98). John Debenham, *Knowledge Engineering: Unifying Knowledge base and Database Design*, Springer-Verlag, 1998.

(de Silva 00). Ed. by Clarence W. de Silva, *Intelligent Machines: Myths and Realities*, CRC Press, pp. 13, 2000.

(Dorigo 04). Marco Dorigo and Thomas Stützle, *Ant Colony Optimization* by MIT Press, 2004. Note the book also has a CD with software for solving the Traveling Salesman Problem using the Ant Evolutionary Algorithm.

(Fetzer 01). James H. Fetzer, *Computers and Cognition: Why Minds Are Not Machines,* Kluwer Academic Publishers, pp. 25-182, 2001.

(Foster 03). Ian Foster, "The Grid: Computing without Bounds" *Scientific American,* Volume 288, Number 4, pp. 78–85, April 2003.

(Friedman-Hill 03), Ernest Friedman-Hill, *Jess in Action: Rule-Based Systems in Java*, Manning Publications, 2003.

(Fogel 03). Ed. by Gary B. Fogel and David W. Corne, *Evolutionary Computation in Bioinformatics*, Morgan Kaufmann Publisher, 2003.

(Giarratano 90a). Joseph C. Giarratano, et al., "Future Impacts of Artificial Neural Systems on Industry," *ISA Transactions*, pp. 9-14, Jan. 1990.

(Giarratano 90b). Joseph C. Giarratano, et al., "The State of the Art for Current and Future Expert System Tools," *ISA Transactions*, pp. 17-25, Jan. 1990.

(Giarratano 91a). Joseph C. Giarratano, et al., "An Intelligent SQL Tutor," 1991 Conference on Intelligent Computer-Aided Training (ICAT '91), pp. 309-316, 1991.

(Giarratano 91b). Joseph C. Giarratano, et al.,"Neural Network Techniques in Manufacturing and Automation Systems," in *Control and Dynamic Systems*, Vol. 49, ed. by C.T. Leondes, Academic Press, pp. 37-98, 1991.

(Giarratano 04). Joseph C. Giarratano, (http://www.pcai.com/Paid/Issues/PCAI-Online-Issues/17.4_OL/New_Folder/So&9i2/17.4_PA/PCAI-17.4-Paid-pp.18-Art1.htm).

(Gruber 93). T.R. Gruber, "A Translation Approach to Portable Ontology Specification," *Knowledge Acquisition* 5: 199-220, 1993.

(Hecht-Nielsen 90). Robert Hecht-Nielsen, *Neurocomputing*, Addison-Wesley Publishing Co., pp. 147, 1990.

(Helmreich 98) Stefan Helmreich, *Silicon Second Nature*, University of California Press, pp. 180-202, 1998.

(Hopgood 01). Adrian A. Hopgood, *Intelligent Systems for Engineers and Scientists*, CRC Press, 2001.

(Hrycej 97). Tomas Hrycej, *Neurocontrol*, John Wiley & Sons, Inc., 1997.

(Luger 02). George F. Luger, *Artificial Intelligence,* Fourth Edition, Addison-Wesley, 2002.

(Jackson 99). Peter Jackson, *An Introduction to Expert Systems*, Addison-Wesley Publishing Co., 1999.

(Johnson 01). Steven Johnson, *Emergence: The connected lives of ants, brains, cities, and software*, Scribner, 2001.

(Kantardzic 03). Mehmed Kantardzic, *Data Mining,* IEEE Press, 2003.

(Kasabov 02). Nikola Kasabov, *Evolving Connectionist Systems*, Springer-Verlag, pp. 7-29, 2002.

(Kennedy 01). James Kennedy and Russel C. Eberhart, Swarm Intelligence, Morgan-Kaufmann Publishers, 2001.

(Lajoie 00). Ed. by Susanne P. Lajoie, *Computers As Cognitive Tools, Volume Two: No More Walls*, Lawrence Erlbaum Associates, Publishers, 2000.

(Lakemeyer 03) ed. by Gerhard Lakemeyer and Bernhard Nebel, *Artificial Intelligence in the New Millenium*, Morgan-Kaufmann Publishers, 2003.

(Mendel 01). Jerry M. Mendel, *"Introduction to Rule-Based Fuzzy Logic Systems,"* IEEE Press, 2001.

(Satinover 01). Jeffrey Satinover, *The Quantum Brain*, John Wiley, 2001.

(Sipper 02). Moshe Sipper, *Machine Nature: The Coming Age of Bio-Inspired Computing*, McGraw-Hill, 2002.

(Swingler 96). Kevin Swingler, *"Applying Neural Networks: A Practical Guide,"* 1996.

(Wagman 99). Morton Wagman, *The Human Mind According To Artificial Intelligence*, Praeger Publishers, p. 76, 1999.

第2章　知识的表示

2.1　概述

本章和下一章讨论**逻辑**（logic）。许多人把逻辑等同于合理。也就是说，如果一个人符合逻辑，那么自然他也是合理的。所以我们需要更加明确的定义以便处理基于逻辑的人工智能和专家系统。任何使用计算机的人都知道，计算机的优点在于它严格执行指令，但其缺点也在此。如果专家系统决定着你的信用评估，那么无论是即将被税务机关审计，还是有其他一样重要的事情，最好让所有东西都非常符合逻辑，没有一点模糊。从技术上讲，逻辑是研究有效的推理。即如果事实集是真的，那么结论也是真的。一个无效的推理意味着从一组真实的事实得到一个假的结论。

我们需要明确**形式逻辑**（formal logic）与**非形式逻辑**（informal logic）之间的区别。非形式逻辑是人们，特别是律师为了赢得辩论，比如一个案例而采用的一类逻辑。一个辩论应该与慷慨激昂无关，但在法庭上合法的辩论中，有好的律师通过采用煽动性的词语说服陪审团而获胜。一个复杂的逻辑辩论是一系列推理，其中一个结论推导出另一个结论，以此类推。在法庭中，也许会直到推出下述结论之一，或有罪、或无罪、或因精神错乱而无罪，或审判无效而上诉。

在形式逻辑也称为**符号逻辑**（symbolic logic）中，推理和其他通过有效方法证明最终结论真伪的因素是最重要的。计算机程序中的漏洞就是程序进行了无效符号推理的最佳例子。逻辑同样需要赋予符号以**语义**（semantics）。在形式逻辑中，我们不会采用受主观影响的词如“你喜欢百事可乐还是可口可乐?”。在一般的程序设计中，形式逻辑通常采用有一定含义的词作为变量名，以体现语义。

除了介绍逻辑，本章也将介绍常用的知识表示方法。知识表示（Knowledge representation，KR）由于其在使用知识的软件系统中起着决定性作用而被看作是人工智能的核心。这种重要性同样体现在计算机科学的数据库设计中。数据库通常用来存储当前数据，例如一个仓库产品的详细清单、已付账单、账号日期等。数据库虽然不是知识，但很多公司已利用数据挖掘来提取知识。

数据挖掘意味着使用存储在**数据仓库**（data warehouse）中的**档案数据**（archival data）来预测未来趋势。例如，一个公司可通过观察过去五年12月份的销售报告来预测存货清单的内容和数量。又如，他们可能通过应用数据挖掘而发现圣诞贺卡在12月份销售量很好，而情人贺卡则不是。一个更真实的案例是发现红色和绿色衣服在冬天比春天畅销，因为红色和绿色与圣诞节相呼应，而棕色、橙色和黄色衣服在秋季比较畅销。经理们可能会发现，数据挖掘可以协助提供需要购买的衣服数量，以及在换季时推出市场的时机。当然，数据挖掘真正的用途在于从公司存储的海量历史数据中识别出对于人类来说并不直观的模式。除了经典的统计学方法，人工智能技术如ANS、遗传算法、进化算法和专家系统等，都可单独或者组合地应用于数据挖掘（Werbos 94）。

KR在专家系统中非常重要主要有两个原因。首先，专家系统是专为某一类基于**逻辑规则**（rules of logic）即**推理**（inference）的知识表示设计的。通常，我们把推理理解为从事实推出结论。但遗憾的是人们并不擅长推理，因为我们容易把语义和推理过程本身混淆起来，从而导致错误的结论。政治选举是这方面很好的例子，政治逻辑的本质就是推理不使用事实，或者使用不可靠的事实，或者从同样的事实得到截然不同的结论。

推理是指不依赖语义的某种推理方法的形式说法。尽管现实世界中语义不可缺少，但专家系统专为基于逻辑的推理而设计，而不需要把主观语义引入其中。推理的目的是从事实和辩论方式上得到可靠的结论。从逻辑上来讲，这里辩论的含义是指从事实和推理规则中得到有效结论的形式方法。

有效的推理称为逻辑推理。在现实世界中，常识和概率推理（commonsense and probabilistic reasoning）是无法评估的，它们包含了不确定性，因为世界上没有一样事物是100%绝对的。如果有人说“天空是

蓝的”，那么过一会儿可能会变灰或绿。不确定性推论是我们在第4章和第5章要讨论的重要内容。

KR重要的第二个原因是，它影响着专家系统的开发、效率、速度和维护。这如同程序设计中对数据结构的选择。一个好的程序设计必须选择简洁的变量、数组、链表、队列、树、图、网络甚至外部数据库如Microsoft Access，SQL Server或者Oracle。在CLIPS中，KR可以是规则、自定义模板、对象和事实。

在接下来的一章中，我们将讨论如何应用推理从知识中得到有效结论，以及常常需要小心的看起来符合逻辑的谬论。这一点对第6章讨论的知识获取非常重要，当你为专家系统访问一名人类专家时，你必须能够从可以导致无效结论的语义中分辨真正的知识。

附录G中列出了很多利用人工智能来进行推理、定理证明和逻辑学习的程序。

2.2 知识的含义

知识一词，就像“爱”一样，每个人都知道它的意思，但却难以对它进行定义。如同“爱”一样，知识有许多意思。有些词，如数据、事实和信息常与知识互换使用。

人们可能会通过推理和经验来求解一个问题。一般把使用经验来求解问题叫做**启发式（heuristics）**。启发式经验如果是长的特殊事例就叫**基于案例的推理（case-based reasoning）**。这是法律、医学、修理等领域最主要的推理方法之一。律师、医生、机械师常试着用以前类似的案例，称为**先例（precedents）**来处理问题（Leake 96）。一些人已有和你同样的问题，你可能觉得基于案例的解决方法是昂贵的，但看看建一个新医疗的、法律的、汽车修理先例需花费多少！

一个专家系统也许含有成千上万个可以参考的小个案，每一个规则都可认为是一个可以找到解的小先例，或者是可用于有希望找到解的推理链中的一个小先例。

关于知识的研究称为**认识论**（epistemology）。它涉及知识的本质、结构和起源。图2.1描述了认识论的一些分类。除了亚里士多德、柏拉图、笛卡儿、哈莫、康德和其他一些人所代表的哲学类外，还有两个特别的类别：**先验知识**（priori）和**后验知识**（posteriori）。术语先验知识来自拉丁文，它的意思是“超前的”。这种知识不依赖于源于感觉器官所获得的知识，例如，“万事都有起因”和“平面内所有三角形的内角之和都是180°”就是先验知识。先验知识被认为是普遍正确的，没有反例，也不能被否定。逻辑语句、数学定律都是先验知识的例子。

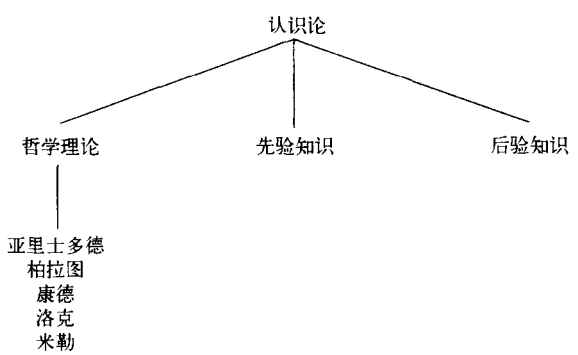


图2.1 认识论的一些分类

与先验知识相反的是由感觉器官所获得的知识，即后验知识。后验知识的正确与错误可以用感觉经验来证明，如“光是绿的”。但由于感觉经验并不是一直可信，后验知识可以在新知识的基础上被否定，而不必举反例。例如，如果你看到一个人有着棕色的眼睛，你或许会认为那人的眼睛是棕色的。但当你后来看到那个人拿下了棕色的眼镜片而露出蓝色的眼睛时，你的知识就必须修正了。

知识可以进一步被划分为**过程性知识**（procedural knowledge）、**说明性知识**（declarative knowledge）和**默认性知识**（tacit knowledge）。其中，过程性知识和说明性知识与第1章讨论的过程和说明性规范

相对应并在（Brewka 97）中有更深层讨论。

过程性知识常常是指知道如何做某事，例如知道如何去烧开一锅水。但如果你认为自己知道所有海拔下如何煮沸水则会导致常识推理的错误，一个真实的矛盾是，在常识下没有推理只有真实世界的经验（在本章末的煮沸水习题）。说明性知识是指知道某事是对的还是错的，它常用说明语句的形式来表达知识，例如“不要把你的手指放进一锅沸水中”。使用逻辑来表达知识的不同方式已开发出来，见（Sowa 00）。

默认性知识由于不能用语言来表达，常被称作**无意识的知识**（unconscious knowledge），例如你知道如何移动你的手。一般，你或许会说通过绷紧或放松你的肌肉和筋骨来移动你的手。但在更低的层次上，你是如何知道怎样绷紧或放松肌肉和筋骨的？走路和骑自行车也是这样的一些例子。在计算机系统中，ANS与默认性知识有关。因为神经网通常不能直接解释它的知识，但如果有一个适当的程序，或许就可以解释了（见第1.14小节）。

知识在专家系统中最为重要。事实上，与Wirth的经典表述

算法 + 数据结构 = 程序

相类似，在专家系统中，有：

知识 + 推理 = 专家系统

正如本书中所指，知识是图2.2中层次结构中的一部分。最底层是噪音，由几乎没有意义的事项和含糊难解的数据组成。上一层是数据，是一些有潜在意义的事项。信息或是经过加工后具有意义的数据在第三层。再上一层就是代表专门化信息的知识。这些重要信息是行为依据并且受到保护。在第1章中，基于规则的专家系统里的知识被定义为由事实激发而产生新的事实或结论的规则。推理过程是专家系统的另一个重要部分。推理（inferencing）一词通常用于像专家系统这样的机械系统中。而推导（Reasoning）一般用于人类的思维。

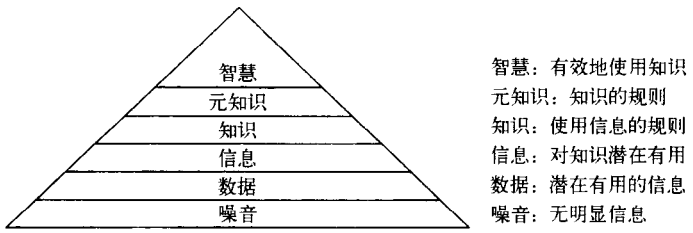


图2.2 知识的层次结构

ANS不进行推理，它寻找对于人类来说并不明显的数据中的隐含模式。从根本上来说，ANS是一个模式分类器。例如，人类读的能力是基于大脑中的神经网已经被训练得具备了对字母组合模式的识别。大脑的另一部分把这些模式翻译成脑海中所听过的词语，因为小时候就是这样教的，最后按照读音读出来。如果把这本书上下颠倒，最初可能很多人没有办法读。但是可以让你的读神经网重新适应以辨认出字体。在经典的心理学实验中，人们带上颠倒图像的眼镜。几天后，他们的大脑接受并适应了这种情况。事实上，我们的眼睛会把图像颠倒，大脑再将其恢复正常。

另一个显示神经网的变通能力的例子是把书本翻转30°（Saratchandran 96）。虽然慢一点，但你仍然可以阅读。如果你把书本翻转180°，也只是增加你阅读的难度。通过训练，有一些人可在翻转书本的情况下一样地阅读，这显示了人们神经网的令人惊异的适应能力。同样的，经过不同角度文字的训练，ANS可以阅读不正常显示的文字。ANS经过的训练角度越多，能够识别的能力就越强。类似地，如果对一个ANS训练不同的草写字样，ANS就可以辨别更多的手写体，就如同一个人可以阅读不同人的手写体一样。

事实（fact）一词指可靠的信息。专家系统使用事实推理。事实若随后证实为假，可以用CLIPS

的真值维持工具（Truth Maintenance facility）撤销，相关的结论、规则以及由此虚假事实产生的事实也自动撤销。专家系统也可以（1）从噪声中提取数据，（2）把数据转化为信息，（3）把信息转化为知识。在一个依赖事实的专家系统中使用原始数据是非常危险的，因为结论的可信性将很不可靠。这正如一句谚语所说："垃圾进，垃圾出"，除非某人想用垃圾来支持一个特别的议程。

以上概念可以用一个例子来说明，考察如下的24个数字：

```
13717876683252515643O015
```

如果没有知识，这些数字可能会被看作噪音。但如果知道这组数是有意义的，它就是数据。决定何者为数据，何者为噪音，就像那句关于花园的谚语："杂草是你不想要，却生长着的所有东西。"

在把数据转化为信息时，需要一定的知识存在。例如，下面这个算法可以将数据加工成信息：

```
每两个数字分为一组
忽略那些小于32的两位数
用ASCII字符代替两位数
```

把这个算法运用到前面的24个数字中，则可以产生信息：

```
GOLD 438+
```

现在知识可以被运用到这个信息中。例如，这里有一条规则：

```
IF 黄金价格低于500,
   且价格正在上涨(+)
THEN 买黄金
```

尽管在图2.2中没有明确的显示，**专家知识**（expertise）是专家所拥有的一种专业化的知识，如图2.2的元知识、知识、智慧。专家知识通常不能从图书、报纸那样的公共信息源中找到。例如，关于外科手术过程的细节可以在医学书籍中找到，但是即使价格减半，你是否愿意一个声称已经通过在线课程的人为你实施脑部手术呢？

专家知识是专家所暗含的知识，它必须被抽取才能明确地表示在一个专家系统中。知识的暗含性在于一个真正的专家深刻理解了知识以至于成为第二本能而不需要过多的思考。例如，从医学院毕业的实习生经过每周超过80小时的一年实习期之后才可以比较果断地进行各种治疗手术。这种实习是非常严格的，它使得知识深深刻在脑海中成为第二本能。最上面的知识是**元知识**（metaknowledge）。**元**（meta）的意思是"在……之上"。

元知识是关于知识和专家知识的知识。尽管一个专家系统可设计为具有几个不同领域的知识，但通常并不受欢迎，因为它使得系统定义不够明确。经验表明，大多数成功的专家系统是严格限于某个较小的领域的。例如，如果一个专家系统被设计用来诊断细菌疾病，那么用它来同时进行汽车故障诊断就不妥当了。参考现实世界中的例子，一个医生专攻某一方向而不是所有的医学领域。即使是家庭医师（称之为全科开业医师），也会在需要的时候建议病人去看专科医生。

在专家系统中，一个本体就是描述问题领域的元知识。理想情况下本体应该以形式的方式描述，以方便找出不一致性和不充分性。有很多免费的和商用的工具可以构建本体库。建立一个本体库应该在专家系统实施之前，否则由于有更多的领域信息、规则可能会重写，这样就会增加费用、开发时间和漏洞。

例如，一个专家系统可以具备关于修理GM轿车、GM SUV和GM柴油卡车的知识库。根据要修理汽车的类型，将使用相应的知识库。把所有知识库存放在内存中可能会使内存和速度方面效率较低，因为Rete网持续地修改内存中的规则网络。而且，当卡车和汽车的规则前件具有相同的模式，而后件不同时，可能导致冲突。例如，如果燃料读数为空，汽车专家系统会表明"为油箱添加汽油"，而卡车专家系统可能会表明"为油箱添加柴油"。为汽车添加柴油或者为卡车添加汽油都是不恰当的。当系统中的规则数目增加时，专家系统必然变慢，因为Rete网变大了。元知识可以用来决定哪个领域知识应

该被调入内存，也可以用来指导设计和维护专家系统以及本体。

在哲学意义上，**智慧**（wisdom）是所有知识的顶峰。智慧是确定生活的最佳目标并如何去获取它的元知识。一条智慧的规则可以是：

IF 我有足够的钱来保证我的配偶快乐
THEN 我会退休并享受生活

基于人工智能的智慧工程正在持续地发展。然而，由于这个世界上智慧的极端缺乏，我们应该将自己限制在基于知识的系统中，而把基于智慧的系统留给政客和其他专家。

2.3 产生式

已经有许多不同的知识表示技术。包括规则、语义网、框架、脚本、逻辑、概念组等。特别是还有许多知识表示语言如 KL-1 和它的模型基础发展，CLASSIC（Brachman 91）。还有许多其他语言被推荐包含可视化语言的。

正如在第一章中所描述的，产生式规则常被作为知识库而用在专家系统中，因为其优点大大超过了缺点。

定义产生式的一种形式方法是 Backus-Naur 范式（BNF）。这种方法是一种定义语法的**元语言**（metalanguage）。**语法**（syntax）定义了形式，而**语义**（semantics）则指出了含义。元语言是一种描述语言的语言。元意为“在……之上”，因此元语言是高于一般语言的。

语言的种类很多，有自然语言、逻辑语言、数学语言和计算机语言。如“一个句子由一个名词和一个动词及标点符号组成”，这个简单的英语语言规则的 BNF 如下：

＜句子＞::＝＜主语＞ ＜动词＞ ＜结束标志＞

这里的尖括号〈〉和::＝是元语言符号，而不是语言符号。符号“::＝”意为“被定义为”，它同第 1 章中产生式规则的箭头“—〉”意义是等价的。为了防止与 Pascal 语言的操作符“:＝”相混，我们将使用箭头。

尖括号中的项称为**非终结**（nonterminal）符号或简称为**非终结符**（nonterminal）。一个非终结符是表示其他项的变量。其他的项既可以是非终结符，也可以是**终结符**（terminal）。一个终结符不能被其他任何项所代替，因此它是常量。

〈句子〉是一个特殊的非终结符，因为它是一个定义其他符号的**开始符号**（start symbol）。在程序语言的定义中，开始符号常被命名为〈程序〉。产生式规则：

＜句子＞→＜主语＞＜动词＞＜结束标志＞

表示一个句子由一个主语跟一个动词，再跟一个结束标志组成。下面这些规则通过指出可能的终结符来定义非终结符。在元语言中|意为“或”。

＜主语＞→I|you|We
＜动词＞→left|came
＜结束标志＞→.|?|!

语言中所有可能的句子，都可以这样来生成，依次用右边的非终结符或终结符来代替左边的每个非终结符，直至所有的非终结符被消除为止。下面就是这样一些产生式：

```
I left.
I left?
I left!
You left.
You left?
You left!
We left.
We left?
We left!
```

一组终结符称为语言的**串**（string）。如果这个串是从开始符号通过使用产生式规则不断替换非终结符而获得的，那么它就是一个合法的**句子**（sentence）。例如“We”，“WeWe”和“leftcamecame”都是合法的串，但不是合法的句子。

语法（grammar）是无歧义地定义一种语言的一组完整的产生式规则。尽管前面的规则确实定义了一个语法，但它是很有限的，因为它几乎没有什么可能的产生式。例如，一个更加详细的语法还应该包括直接宾语，正如下面的产生式：

```
<句子>→<主语><动词><宾语><结束标志>
<宾语>→home|work|school
```

尽管这是一个正确的语法，但它对于实际应用还是太简单了。一个更为实际的语法可以是下面这样的，为了简单起见省略了它的结束标志：

```
<句子>→<主语短语><动词><宾语短语>
<主语短语>→<限定词><名词>
<宾语短语>→<限定词><形容词><名词>
<限定词>→a|an|the|this|these|those
<名词>→man|eater
<动词>→is|was
<形容词>→dessert|heavy
```

<限定词>用于指示一个特定的项目。用这种语法，就产生了像下边这样的句子：

```
the man was a dessert eater
an eater was the heavy man
```

一棵**语法分析树**（parse tree）或**派生树**（derivation tree）是一种把句子分解成所有非终结符和终结符，以便得出句子的图形表示法。图 2.3 是句子“the man was a heavy eater”的语法分析树。但是，串“man was a heavy eater”因为缺少主语短语中的限定词而不是一个合法的句子。当编译器判断一个程序中的语句是否符合某种语言的语法时，便产生了一棵语法分析树。

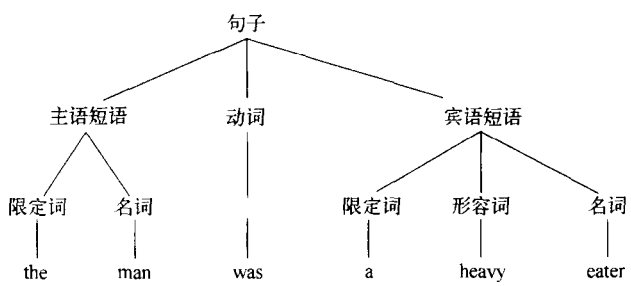


图 2.3 句子的语法分析树

图 2.3 中的树表明“the man was a heavy eater”这个句子可以通过运用合适的产生式而从开始符号获得。这个过程的步骤如下所示；双箭头“=>”表示所使用的产生式。

```
<句子> =><主语短语><动词><宾语短语>
<主语短语> =><限定词><名词>
<限定词> =>the
<名词> =>man
<动词> =>was
<宾语短语> =><限定词><形容词><名词>
<限定词> =>a
<形容词> =>heavy
```

<名词> =>eater

另一种使用产生式生成合法句子的方法就是前面讨论过的方法，即用合适的终结符代替所有非终结符。当然，并非所有生成的产生式，如“the man was the dessert”，都有意义。

有限状态机（FSM）非常适合识别句子结构。例如，编译器使用有限状态机把计算机语言源代码解析（parse）成最小的意义单元，称为标记（token）。FSM 和解析器是某些应用的基础，如编译器把源代码转换成汇编语言以及精确的语音识别等。术语编译器（compiler）的含义如今已经扩展到 Java 语言中，指使用 javac 编译器把源代码转换成平台无关的字节码，就如同 20 世纪 70 年代引入 Pascal 一样，其字节码可以在任何微处理器上执行。Pascal 转换成字节码被称为解释器而不是编译器，因为字节码跟汇编语言一样并不是特定的机器语言指令。不过，解释器比编译器要慢，所以称之为编译器要比称之为解释器的广告效果好。

Xerox 公司的一个有限状态机的在线演示在：（http：//www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fsinput.html）。关于 FSM 的例子如软饮料机的链接在：（http：//www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fsexamples.html）。FSM 的一个非常全面的参考在：（http：//odur.let.rug.nl/alfa/fsa _ stuff/）。

虽然 FSM 适于处理受限的符号集，如数字 0～9、字母表中的字母，但当处理如语音识别这样可能出现二义性的问题时就会出现困难，例如，下面两个句子：

(1) No one has let us read
(2) No one has lettuce red

在句子（1）中，某人抱怨他们没有读，在句子（2）中，没有人有红色的莴苣，一个消除词语 lettuce，let us，read，red 二义性的好的方法是使用隐马尔可夫机（hidden Markov Machine（HMM）），它为有限状态机的行为分配一个概率。通过考虑整个句子结构以及其他的句子，HMM 可断定正确的上下文（不管是读一本书还是在食品杂货店找一种蔬菜）。虽然专家系统不适合作为实现 HMM 的软件选择，但可作为语音识别的一个前端工具，因为像 CLIPS 这样的专家系统可用来触发适合的规则。

事实上，CLIPS 可以很方便地与 C、C++ 代码交互，因此用 C、C++ 写的 HMM 可以在 CLIPS 中调用，就像调用其他任何 CLIPS 关键词一样。CLIPS 的一个长处在于它是一个可扩展的语言，用户可以很容易地在编译时增加关键词以便获取最佳性能。而且，利用 COOL 的面向对象特性，通过对象的多继承性，对象可以用来扩展 CLIPS。其他软件，如 ANS、遗传算法以及用 C 或 C++ 写的软件既可以加在规则的左部以触发规则，也可以加在规则的右部用作输出。例如，对语音合成器、机器人受动器、执行器都可以通过定义一个适当的关键词函数在 CLIPS 中被调用。CLIPS 的源代码是可获取的意味着新的关键词可以编译到 CLIPS 中，与其他不提供源代码的专家系统工具相比，它不会降低速度。

2.4 语义网

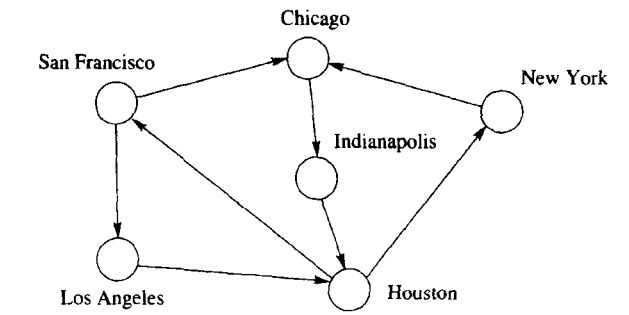
语义网络（semantic network）或语义网是用于表示命题信息的一种经典的人工智能表示技术（http：//www.pcai.com/web/T6110H2/R6.o1.h8/pcai.htm）。语义网也常常叫做**命题网**（propositional net）。正如前面所讨论过的，一个命题是一个或者为真或者为假的陈述，如“所有的狗都是哺乳动物”和“一个三角形有三条边”。命题是说明性知识的表达形式，因为它陈述了事实。根据数学的观点，一个语义网就是一张带有标记的有向图。一个命题通常或真或假，且由于其真值的不可再分而被称作**原子**（atomic）。原子这个词用于表示一个不可再分的对象。与之相反，我们在第 5 章所讨论的模糊命题通常不只为真或假。

语义网最初是作为人工智能中一种表达人类记忆和理解语言的方法由 Quillian 在 1968 年提出的。Quillian 用语义网来分析句子中的词意。注意，句子的意思与我们前一节所提到的通过 FSM 和 HMM 把句子解析成标记和词语结构不一样。从那时起，语义网就被运用到许多涉及知识表示的问题中。这种对意思的理解是必需的，它必须超越那些简单的专家系统或 AI 软件以避免二义性。在下一章的推理

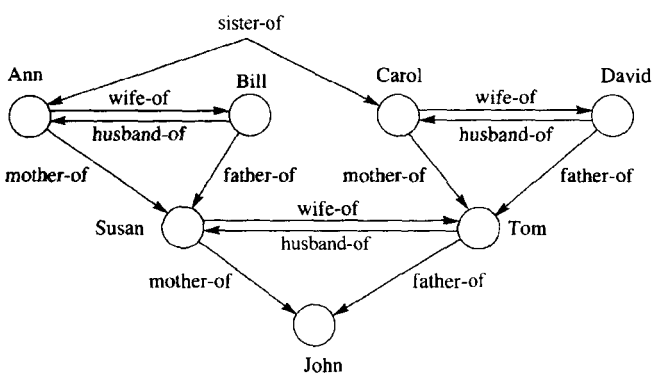
中，我们将看到专家系统如何从事实中得出结论，这些结论又可被其他的规则使用，形成一个推理链直到得到一个有效的结论。不使用语义，专家系统也会失败，就如同人们被二义性迷惑而失败一样。

语义网的结构可用图形表示为**结点**（nodes）以及连接结点的**弧**（arcs）。结点用来表示对象，而弧则称为**连接**（links）或**边**（edges）。

语义网的连接用来表示关系。结点则被用来表示物理实体、概念或势态（situation）。图 2.4（a）是一个一般的网，它实际上是一个用连接表示城市间航空路线的有向图。结点是一些圆圈，连线则连接结点，箭头指示飞机飞行的方向，因此便构成了有向图。在图 2.4（b）中，连接表示一个家庭各成员的关系。在一个语义网中，关系是十分重要的，因为它们提供了组织知识的基本结构。没有关系，知识只是无关事实的一个集合。有了关系，知识就是一个可推出其他知识的具有内聚力的结构。例如，在图 2.4（b）中，我们可以推出 Ann 和 Bill 是 John 的祖父母，尽管图中并没有明确地标明“是……的祖父母”这一连接。



(a) 一般网络



(b) 语义网络

图 2.4 两种形式的网

语义网有时也被称作**联系网**（associative nets），这是因为它的结点都和其他结点相联系或相关。事实上，Quillian 最初的工作就是把人的记忆模型转化为一个联系网，其中结点表示概念，而连接表示概念间的联系。根据这个模型，当读入一句话中的词语而激活一个概念结点时，它与其他概念之间的联系也就以传播的方式被激活了。如果其他的概念结点接收到充分的刺激，就会变成有意识的思想。例如，尽管你认识上千个词，可当你读一个句子时你只是关注正在读到的特定的词。

某些关系已被证实在许多种知识表示方法中非常有用。与其为不同的问题定义新的关系，不如使用这些标准的关系。这些标准关系的使用使不同的人更容易去理解一个不熟悉的网。

常用的两类连接是 IS-A 和 A-KIND-OF，有时写作 IS-A 和 AKO。图 2.5 就是使用这些连接的一个语义网例子。图中，IS-A 意为“是……的一个实例”，它指一个类的一个特定成员。一个**类**（class）是

一个与数学有关的概念，它表示一组对象。尽管一个集合可以包括任何类型的元素，一个类中的对象却彼此之间存在联系。例如，可以定义这样一个集合：

```
{ 3, eggs, blue, tires, art }
```

但它的成员间没有共同的联系。与之相反，飞机、火车和机动车就可以组成相关的一类，因为它们都属于运输工具。

AKO 连接在这里被用来连接一个类与另一个类。AKO 不用来表示特定个体间的联系，那是 IS－A 的功能。AKO 用来连接一个个体类和它的父类，这里的个体就是一个子类。

从另一种观点来看，AKO 用于连接类型结点，而 IS－A 则将一个实例或一个**个体**（individual）连接到一个一般类。在图 2.5 中，我们可以注意到越是一般的类越在上面，而越是特殊的类就越在下面。一个由 AKO 箭头指向的更一般的类称为**超类**（superclass）。如果一个超类也有指向另一个结点的 AKO，那么它同时也是 AKO 所指超类的一个类。另一种说法就是，一个 AKO 从一个**子类**（subclass）指向一个类。有时用 ARE 来代替 AKO，ARE 按照一般动词“are”的读法去读。

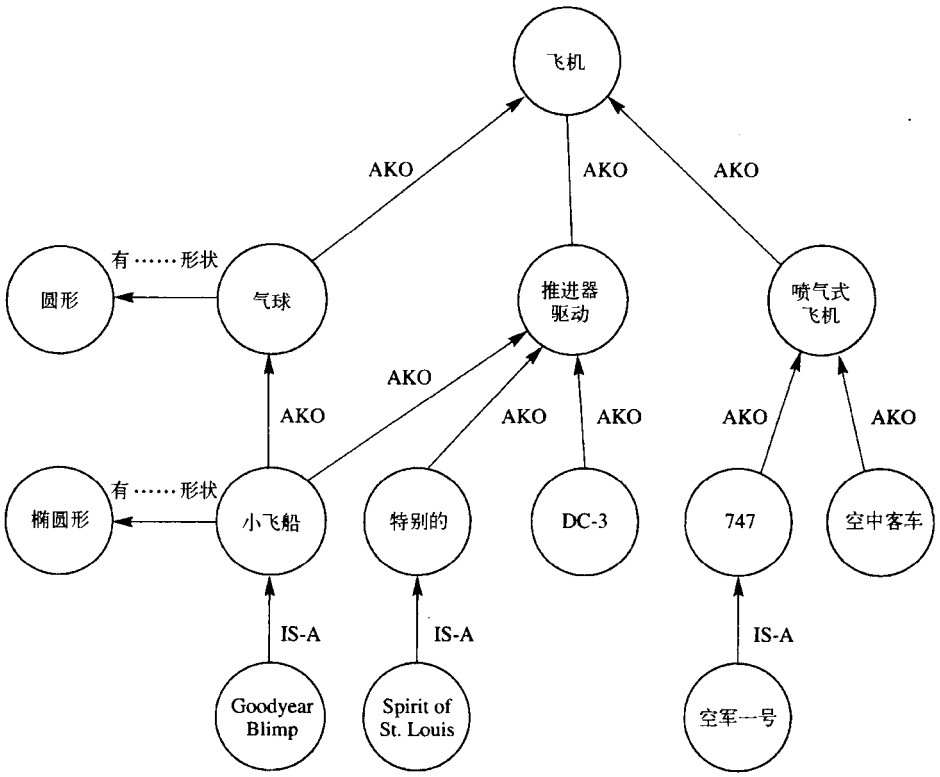


图 2.5　具有 IS-A 和 A-Kind-Of（AKO）联系的语义网

一个类的对象一般都有一个以上的**属性**（attribute），而每个属性又有一个值（value）。属性和值组合成**特性**（property）。例如，一个小飞船的属性有：尺寸、重量、形状、颜色等。形状属性的值是椭圆。语义网中也可以有其他类型的连接。IS-A 连接用于定义一个值。例如，总统的座架总是空军一号。如果总统在一架直升飞机上，那么空军一号便是一架直升飞机。而 **CAUSE** 连接则表达了因果知识。例如，热空气引起一个气球上升。

一个结点的属性被一个后代所复制，称作**继承**（inheritance）。除非有特例，否则我们可以假定一个类的所有成员都会继承它们超类的所有特性。例如，气球是圆形的。然而，由于小飞船类有一个指向椭圆形形状的连接，于是它便优先使用。继承在知识的表达中是一种非常有用的工具，因为它不必

重复表达相同的属性。由于许多复杂的关系都可以用一些结点和连接来表示，连接和继承就为知识表示提供了一个很有效的手段。

2.5 对象－属性－值三元组

使用语义网的一个问题就是没有命名连接的标准。例如，一些书既将 IS-A 作为类属关系，又作为个体关系。例如，IS-A 既用作表示一般的，又表示个体。所以 IS-A 既按一般的词用作“是一个”，又用作 AKO。

另一个常用的连接是 HAS-A，它连接一个类和一个子类。HAS-A 的指向与 AKO 相反，它常用来连接一个对象和这个对象的一部分。例如：

```
car HAS-A engine
car HAS-A tires
car IS-A ford
```

具体地，IS-A 连接一个值和一个属性，而 HAS-A 则连接一个对象和一个属性。

对象、属性和值这三个元素的出现是如此频繁，以致我们可以仅用它们来构建一个简化的语义网。可以用**对象—属性—值三元组**（OAV，object-attribute-value triple）来表示一个语义网中的所有知识，在 MYCIN 专家系统中就是使用它来诊断传染病。OAV 三元组便于以表格的形式列出知识，并通过规则推导将表格转化为计算机代码。OAV 三元组的一个例子如表 2.1 所示。

表 2.1 一个 OAV 表

对象	属性	值
apple	color	red
apple	type	mcintosh
apple	quantity	100
grapes	color	red
grapes	type	seedless
grapes	quantity	500

OAV 三元组在表达事实以及用来匹配事实与规则前件的模式时特别有用。这种系统的语义网由通过 HAS-A 和 IS-A 连接的对象结点、属性结点、值结点组成。如果只需表达简单对象而不必表达继承，那么一个更为简单的表达，即**属性—值对**（attribute-value pair），简称为 AV 就已足够用了。

2.6 PROLOG 和语义网

语义网很容易翻译成 PROLOG。例如，

```
is_a(goodyear_blimp,blimp).
is_a(spirit_of_st_louis,special).
has_shape(blimp,ellipsoidal).
has_shape(balloon,round).
```

这些都是表达了图 2.5 语义网中一些关系式的 PROLOG 语句 。句号表示语句的结束。

PROLOG 本质

上面的每一条语句都是 PROLOG 的**谓词表达式**（predicate expression）或简单地说一个谓词，因为它们都是基于谓词逻辑的。然而，PROLOG 并不是一种真正的谓词逻辑语言，它是一种带有可执行语句的计算机语言。在 PROLOG 里，一个谓词表达式包括谓词名称，如 is_a，后面要么不接任何东西，要么接用圆括号括起来并用逗号分开的变量。以下是一些 PROLOG 的例子，其谓词表达式和注释之间用分号隔开：

```
color(red).                ; red is a color is a
                             fact
mother(pat,ann).           ; pat is the mother of
                             ann
parents(jim,ann,tom)       ; jim and ann are
                             parents of tom
surrogatemother(pat,tom). ; pat is surrogatemother
                             of tom
```

如果你将谓词名跟在第一个变量之后，那么有两个变量的谓词将会更容易理解。有多个变量的谓词，比如双亲谓词，所表达的意思必须清晰地说明。因为连线只有两端，所以语义网主要用于表示二元关系。而双亲谓词有三个变量，因此语义网不可能只用一条有向边来表达双亲谓词。如果把 Tom 和 Susan 放在一个双亲结点而把 John 放在另一个结点，这也会导致新的困难。因为不可能用此双亲结点来表示其他二元关系，例如“mother_of”，因为它涉及 Tom。

谓词也可以表示 IS_A 和 HAS_A 连接。

```
is_a(red,color).
has_a(john,father).
has_a(john,mother).
has_a(john,parents).
```

注意到 has_a 谓词没有表达出和先前一样的意思，这是因为 John 的父亲、母亲和双亲没有明确地说出名字。为了给他们命名，必须添加另外的谓词。

```
is_a(tom,father).
is_a(susan,mother).
is_a(tom,parent).
is_a(susan,parent).
```

即使增加了这些额外的谓词也没有像原来的谓词一样表达相同意思。例如，我们知道 John 有一个父亲和 Tom 是一位父亲，但这并没表示出 Tom 是 John 的父亲。

前面所有的语句实际上都描述了 PROLOG 中的事实。PROLOG 程序按照下面**目标**（goals）的一般形式由事实和规则组成：

```
p:- p1,p2,...pN.
```

这里，p 是规则的头，而 p_k 是**子目标**（subgoals）。一般地，这种表达称为 Horn 子句，表示了当且仅当所有子目标都满足时，头目标 p 才能满足。当遇到某个失败的谓词时，则 p 不满足。由于不容易用经典的逻辑来证明否定，而 PROLOG 是以经典逻辑为基础的，因此使用失败的谓词是方便的。否定可看作是寻找证明的失败，如果有很多潜在的匹配那将可能是一个很长的搜索过程。这种**截断**（cut）和失败谓词通过减少对证明的搜索使得否定过程更有效。

用来分隔子目标的逗号表示了逻辑“与”。符号“:-”可解释为“IF”。只有当头存在以及右边没有子目标，如

```
p.
```

时，才认为头是真的，这就是以下谓词被认为是事实并且为真的原因：

```
color(red).
has_a(john,father).
```

事实还可看作是一个没有条件的结论，它不依赖于任何东西，因而不必有 IF 或“:-”。与之相反，PROLOG 的规则需要 IF 是因为它们是有条件的结论，其值依赖于一个或更多的条件，下面举一个双亲规则的例子：

```
parent(X,Y):- father(X,Y).
parent(X,Y):- mother(X,Y).
```

这意味着如果 X 是 Y 的父亲或者 X 是 Y 的母亲，那么 X 是 Y 的双亲。类似地，祖父母可以定义为：

```
grandparent(X,Y):- parent(X,Z),parent(Z,Y).
```

祖先可以被定义为：

```
(1) ancestor(X,Y):- parent(X,Y).
(2) ancestor(X,Y):- ancestor(X,Z),ancestor(Z,Y).
```

这里，(1) 和 (2) 只是用于区别。

在 PROLOG 中搜索

虽然有些系统可产生编译代码，但执行 PROLOG 语句的系统通常是一个解释器。PROLOG 系统的一般形式如图 2.6 所示。用户通过输入谓词查询并接收回答来与 PROLOG 进行交互。**谓词数据库**（predicate database）包含已输入的规则和事实谓词并形成知识库。解释器要判断用户输入的查询谓词是否在数据库中。如果在数据库中，回答“是”，否则回答“不是”。如果问题是一个规则，那么解释器会尝试用**深度优先搜索**（depth-first search）方式满足子目标，如图 2.7 所示。相应的，也给出了**广度优先搜索**（breadth-first search）方式，尽管这不是 PROLOG 的标准搜索方式，但也会有相同的结果。

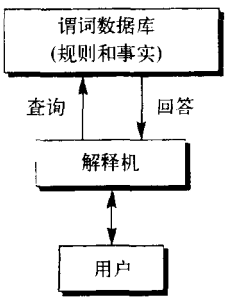
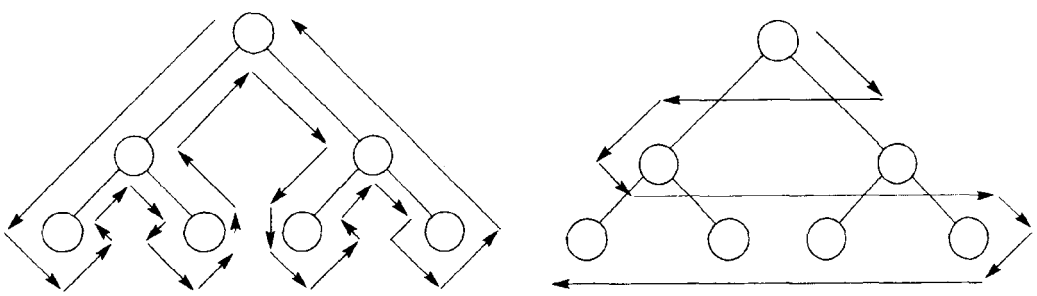


图 2.6 PROLOG 系统的一般组成

在深度优先搜索中，搜索尽可能地向下寻找然后返回。在 PROLOG 中，搜索也会按从左到右的次序进行。广度优先搜索方式在降到下一个更低层次之前会在同一层中执行搜索。



(a) 深度优先搜索　　(b) 广度优先搜索

图 2.7 任意树的深度优先和广度优先搜索

作为在 PROLOG 中搜索目标的一个例子，考虑（1）和（2）中定义的祖先规则。假设现在输入以下这些事实：

```
(3) parent(ann,mary).
(4) parent(ann,susan).
(5) parent(mary,bob).
(6) parent(susan,john).
```

现在假设向 PROLOG 提问以确定 Ann 是否是 Susan 的祖先。

```
:- ancestor(ann,susan).
```

开头的空缺意味这是一个**查询**（query），它是要由 PROLOG 证明的条件。事实、规则和查询是 PROLOG 三种不同类型的 Horn 子句。如果条件是一个子句实例的结论，那么条件是可证明的。当然，子句本身必须是可证明的，它是通过证明子句的条件来实现的。

接到这些输入后，PRLOLG 开始搜索那些头与输入模式 ancestor（ann，susan）相匹配的语句。这叫做**模式匹配**（pattern matching），就像在前面的产生式规则中，前件与事实的匹配。搜索从第一条语句（1），即**顶端**（top）开始直到最后语句（6），即**底端**（bottom）为止。可能存在与第一条祖先规则（1）的匹配。变量 X 匹配 ann、变量 Y 匹配 susan。由于头匹配，PROLOG 尝试匹配（1）的体部分，从而导出子目标 parent（ann，susan）。PROLOG 接着尝试找到与子目标匹配的子句，最后找到了事实（4）parent（ann，susan）。由于原来的查询为真并且已没有要寻找匹配的目标，所以 PROLOG 就会回答“是”。

作为另外一个例子，假设查询如下：

```
:- ancestor(ann,john).
```

它与祖先规则（1）相匹配，把 X 设为 ann，把 Y 设为 john。PROLOG 现在尝试匹配（1）的体，即 parent（ann，john）与每一个双亲语句。因为没有一个匹配，所以语句（1）不可能为真。因为（1）的体不为真，所以头也不为真。

因为语句（1）不为真，所以 PROLOG 尝试第二条祖先语句（2）。把 X 设为 ann，把 Y 设为 john。PROLOG 尝试通过证明（2）的两个子目标为真来证明（2）的头为真。也就是说，ancestor（ann，Z）和 ancestor（Z，john）必须被证明是正确的。PROLOG 以从左至右的顺序首先尝试匹配子目标 ancestor（ann，Z）。从顶端开始，与语句（1）匹配，所以 PROLOG 尝试证明（1）的体 parent（ann，Z）。再从顶端开始，这个体首先与语句（3）匹配，于是 PROLOG 把 Z 设为 mary。现在 PROLOG 尝试把（2）的第二个子目标当成 ancestor（mary，john）来找匹配。它与（1）匹配，所以 PROLOG 尝试匹配它的体 parent（mary，john）。然而，在（3）至（6）中没有一条语句为 parent（mary，john），所以这次搜索失败了。

PROLOG 然后重新考虑 Z 为 mary 这个猜测。既然这个选择不奏效，它会尝试寻找另一个奏效的值。另一个可能是将（4）中的 Z 设为 susan。这种当一条路径失败而倒回去找另一条不同搜索路径的技术叫做回溯，它经常被用来求解问题。

将 Z 设为 susan 后，PROLOG 尝试证明 ancestor（susan，john）为真。在（1）中，体 parent（susan，john）必须证明为真。事实（3）的确与之匹配，所以查询：

```
:- ancestor(ann,john)
```

被证明是真的，因为它的条件已全部证明为真。

注意到 PROLOG 的控制结构属于马尔可夫算法类型，模式匹配的搜索次序由 Horn 子句的输入次序决定。这与基于规则的专家系统相反，专家系统一般按照 Post 模式，在这种模式中规则输入的顺序并不影响搜索。

在这简短的介绍中还没有涉及 PROLOG 其他许多的特征和能力。从专家系统的观点来看，回溯和模式匹配是非常有用的。PROLOG 的这种说明性特征也特别有用，因为其程序规范就是一个可执行程序。

2.7 语义网的困难之处

尽管语义网在表示知识方面非常有用，但它有很多局限性。例如先前所提到的它缺乏连接命名标准，这就使人们难以理解语义网设计的意图以及它是否是以一种一致的方式来设计的。另外一个问题是结点的命名。如果一个结点被称为“chair”，它是代表：

一种专用椅子
椅子总类
椅子概念
会议主席

还是其他意思？由于语义网是表示**确切的知识**（definitive knowledge），也就是说，知识是能够定义的，所以必须严格定义连接和结点名字。当然，在程序语言中也存在同样的问题。幸运的是现在这个问题已经可以用可扩展标记语言（extensible markup language，XML）和本体论解决。XML 已经被证实在提供一种标准的方法把形式语义编码进任何语言方面是非常有用的。也有基于规则的 XML 版本，它如标记语言一样提供对数学、音乐、食品工业以及其他应用计算机进行信息处理的领域的支持。XML 和本体论都把万维网从简单的数据集变成一个更有用的机器可读的信息集合。由于能够辨别语义，如今网页被称为语义网，而不是简单的网页。

另一个问题是搜索结点时的组合爆炸，尤其在查询的回答是否定的时候。因为对一个产生否定结果的查询，可能要在语义网中搜索很多或所有的连接。正如第 1 章出现的旅行售货者问题，如果结点都是有联系的，那么连结数目将是结点数目减一的阶乘。尽管不是所有的表示都需要这种程度的连接，

但存在这种组合爆炸的可能。

语义网最初是作为人类联想记忆的模型提出来的，在这种模型中一个结点与其他结点相连而信息检索产生于结点的扩散性触发。但是，人脑中一定存在另一种机制，因为人不用花很长时间来回答“Pluto有一支足球队吗?”这类的查询。人脑中大约有10^{11}个神经元和10^{15}个连接。如果所有知识都用语义网表示，对答案为否定的查询，如以上的有关足球队的查询，它将用很长很长的时间来回答，因为所有的搜索涉及10^{15}个连接。

由于语义网不能像逻辑方法那样定义知识，所以语义网在逻辑上是不充分的。逻辑表示方法可以指定一种椅子、一些椅子、所有椅子、没有椅子等等，这将在本章的稍后进行讨论。另一个问题是语义网的启发性不足，因为没有办法可以把启发性的信息嵌入网中使之有效地搜索语义网。**启发性方法**(heuristic)是一种经验，它可以帮助找出解决方法但并不保证算法一定可解。启发性方法在人工智能中非常有用，因为典型的人工智能问题十分困难以致没有一种可解的算法或者求解算法十分低效。在网中唯一有用的控制策略是继承，但并非所有问题都有这种结构。

有一些方法尝试去改进语义网的内在问题。它们增强了逻辑性，尝试通过把过程附在结点上来增强启发性。当结点被激活时，就会执行相应的过程。但是，这种在语义网表达上所做的花费只得到了系统性能的少量提高。经过所有这些尝试后，得到的结论是，就像其他任何工具一样，语义网应该被用于它能做得最好的场合，比如表示二元关系，而不应将之看成是万能的工具。

2.8 模式

语义网是一种**浅知识结构**(shallow knowledge structure)。这种浅显性的产生是因为语义网的所有知识都包含在连接和结点之中。术语知识结构类似于数据结构，它表示有序知识的集合而不只是数据。**深**(deep)知识结构应可表示因果知识，以解释一些事情发生的原因。例如，可以用浅显知识来建立这样一个医学专家系统：

```
IF 某人发烧
THEN 吃阿司匹林
```

但该系统并不了解身体的生理基础以及阿司匹林退烧的原因。此规则还可定义为：

```
IF 某人有一只粉红色的猴子
THEN 吃冰箱
```

换句话说，专家系统的知识是浅显的，因为它以语法而不是以语义为基础，于是在以下规则中，可用任意两个词来代替X和Y：

```
IF 某人有 X
THEN 吃 Y
```

注意X和Y在此规则中不是变量，但可表示任意两个词。由于医生上过很多课程且在治疗病人方面有很多经验，所以医生有因果方面的知识。如果一个治疗方案不理想，医生能从此推理找到一个更好的治疗方案。换句话说，一个专家知道什么时候打破常规。

很多现实世界的知识不能用语义网的简单结构来表示。复杂的知识需要复杂的结构来更好地表示。在人工智能中，**模式**(schema)一词用来描述比语义网更加复杂的知识结构。模式源于心理学，它指生物根据刺激不断调整知识或反应。也就是说，生物通过学习了解事情的起因和结果之间的关系，若结果令人愉快，它们会重复此事；而若结果带来痛苦，它们会避免发生此事。

例如，吃喝行为是令人有愉快感觉的机制，模式就建立起吃喝感觉与相应肌肉运动之间的对应关系。一个人不需要思考，就知道该如何令肌肉运动，但却难以解释是如何准确地控制肌肉的。一个更难以解释的模式是如何骑单车，试试解释平衡的感觉!

另一种模式是**概念模式**(concept schema)。例如，每个人都有动物的概念。当很多人被问及何为

动物时，都可能会把动物描述成一些有四肢和毛的东西。当然，动物的概念会因人成长的环境在农场、城市、河边等不同而有所不同。不过，我们在脑子中总有概念的**刻板模式**（stereotype）。尽管“刻板模式”一词在口语中可能有否定意思，但在人工智能中它代表典型例子。因此动物的典型例子对于很多人来说可能是类似于狗这样的动物。

概念模式是一种抽象机制，它根据物体的一般特性对物体进行分类。例如，当你看见一个小的，带绿色茎的红色圆形物体，且上面标有“人工水果”时，那么你可能认为那是一个人工苹果。该物体具有苹果的特性因而使你联想到苹果的概念模式，当然，你不会认为是真苹果。

苹果的概念模式包含了一般苹果的特性，如大小、颜色、味道、用处等等。但模式中不会包括诸如以下的细节，如苹果在哪里采的，用哪辆货车运到超市的，或哪个人把它放到架子上的等等。这些细节对于形成苹果的抽象概念是不重要的。请注意对于一个盲人而言苹果的概念模式会十分不同，这时，苹果的手摸感觉最为重要。

把重点放在事物的一般性质上，就可更容易地推理而不会被不相关的细节所干扰。通常，模式有关于结点的内在结构信息而语义网没有。语义网的标记即是关于结点的所有知识。语义网在计算机科学中就像数据结构，其搜索的关键词也是存储在结点中的数据。而模式则像结点含有记录的数据结构，每一个记录可以包含数据、记录或指向其他结点的指针。

2.9 框架

在很多人工智能应用中使用的一种模式是**框架**（frame）。另一种是**脚本**（script），它基本上是一个框架的时序系列。作为理解视觉、自然语言和其他方面的方法，框架提供了一个方便的结构来表示如刻板模式这样的在特定环境中的典型对象。特别地，框架适于模拟常识，而常识是计算机非常难掌握的。语义网本质上是一种二维知识表示方法；框架通过允许结点有结构而增加了第三维，这些结构可以是简单类型的值或其他框架。

框架的基本特性是它可以表示与具有默认知识的狭窄主题有关的知识。框架系统可以很好地描述诸如汽车这样的机械装置。汽车的各部件如发动机、车身、刹车等相互关联便形成了汽车的完整概念。有关各部件的更详细细节可以通过检查框架结构而获得。尽管汽车的商标有很多，但大多数车有相同的特征如车轮、发动机、车身、传动系统等等。框架与语义网不同，它可以表示更广泛的知识。与语义网一样，它没有定义框架系统的标准。人们已经为框架设计了一些特殊的语言，如 FRL、SRL、KRL、KEE、HP-RL，及通过把框架的增强特性赋给 LISP 而得到的 LOOPS 和 FLAVORS。

槽	槽　值
manufacturer	General Motors
model	Chevrolet Caprice
year	1979
transmission	automatic
engine	gasoline
tires	4
color	blue

图 2.8　汽车框架

框架类似于 C 等高级语言的记录结构以及 LISP 中带有特性表的原子。与记录类型和记录值相对应的是**槽**（slot）和**槽值**（slot filler）。一个框架本质上是一组定义典型对象的槽和槽值。一个关于车的框架的例子如图 2.8 所示。用 OAV 术语来说，汽车是对象，槽名是属性，槽值是属性值。

很多框架并非如图 2.8 那样简单。框架的实用性在于层次结构和继承性。通过在槽值中使用框架和继承，可以建立起非常强大的知识表示系统。特别是基于框架的专家系统非常适于表示因果关系的知识，因为它们的信息由原因和结果组成。与之相反，基于规则的专家系统一般依赖于非因果关系的、无结构化知识。

有些基于框架的工具如 KEE 允许将各种项目存储在槽中，框架的槽可以包括规则、图形、注释、调试信息、给用户的问题、有关情景的假设或者其他框架等。

框架一般设计成既可以表示一般的又可以表示特殊的知识。图 2.9 说明了财产概念的一般框架。

槽	槽 值
name	property
specialization _ of	a _ kind _ of object
types	(car, boat, house)
	if-added: Procedure ADD _ PROPERTY
owner	default: government
	if-needed: Procedure FIND _ OWNER
location	(home, work, mobile)
status	(missing, poor, good)
under _ warranty	(yes, no)

图 2.9 财产的一般框架

槽值可以是一个值，如 name 槽的槽值为 property，也可以是一组值，如 types 槽的槽值。槽可以包含附在槽上的过程，称为附加过程（procedural attachments）。这时一般有 3 种类型，当需要槽值，但初始值不存在且默认（default）值也未设定时，执行 if-needed 过程。默认值对框架十分重要，因为它们模拟了大脑的某些方面。默认值相当于我们以经验为基础建立起来的某种期望。当我们遇到新情况时，就会修改最相近的框架以适应情况的变化。人们不需要在遇到每个新情况时都从头开始，而是可以修改默认值或其他槽值。默认值常用来表示常识。常识是众所周知的知识。当没有特殊知识可用时，我们就运用常识。

当将槽值加入槽中时，将运行 if-added 过程。在 types 槽中，当需增加一个新的财产类型时，将执行过程 ADD _ PROPERTY。例如，这个过程会在增加珠宝、电视机、立体声音响等时执行，因为 types 槽不含有这些槽值。

当要从槽中删除槽值时，将执行 if-removal 过程。如果槽值已无用，这个过程就会执行。

槽值也可以包含关系，比如上面的 specialization _ of 槽。在图 2.9、图 2.10、和图 2.11 中通过使用 a-kind-of 和 is-a 关系来说明这些框架具有何种层次联系。图 2.9 和图 2.10 是一般框架，而图 2.11 是特殊框架，因为它是汽车框架的实例。这里，我们按照一般惯例，认为 a-kind-of 关系是一般关系，而 is-a 关系是特殊关系。

槽	槽 值
name	car
specialization _ of	a-kind-of property
types	(sedan, sports, convertible)
manufacturer	(GM, Ford, Toyota)
location	mobile
wheels	4
transmission	(manual, automatic)
engine	(gasoline, hybrid gas/electric)

图 2.10 汽车框架——财产的一个一般子框架

槽	槽 值
name	John's car
specialization _ of	is _ a car
manufacturer	GM
owner	John Doe
transmission	automatic
engine	gasoline
status	good
under _ warranty	yes

图 2.11 汽车框架的一个实例

在框架系统中，越是一般的框架就越是位于层次结构的上层。假定能通过修改默认值和创建更多的特殊框架来定制特定情形下的框架。框架尝试用一般知识表示对象的主要属性，用特殊知识表示特殊情形来模拟现实世界中的对象。例如，我们通常认为鸟是一种能飞的生物，然而某些鸟，如企鹅和

驼鸟不能飞。这些鸟是特殊类型的鸟，于是在框架中它们的特征就会出现在比金丝雀和知更鸟更低的层次中。换句话说，鸟框架的顶层指明了所有鸟的最一般特征，而低层则反映了现实世界中对象的模糊边界。有着所有典型特征的对象称为**原型**（prototype），可从字面上理解为第一类型。

框架也可以按它们的应用情况分类，一个**情景框架**（situational frame）包含了在特定情景中所期望出现的知识，如生日派对。一个**行为框架**（action frame）包含在特定情景中所应执行的行为槽。即槽值是执行某些动作的过程，如从传输带上移走有缺陷的部件。一个行为框架表示了过程化的知识。情景和行为框架可组合成**因果知识框架**（causal knowledge frame），以表示因果关系。

人们已为各种各样的任务建立了非常复杂的框架系统。一个最能显示框架有效性且令人印象深刻的系统是：创造性地发现数学概念的 Doug Lenat 的自动数学家（Automated Mathematician，AM）系统。Lenat 的 AM 系统可以对有趣的新概念进行猜测并且探索它们。它能给出某些定理的全新数学证明。附录 G 中列出了更多的定理证明和发现系统。

2.10 框架的困难之处

框架最初被作为是表示刻板模式知识的一种范例。刻板模式知识的主要特点是它有定义得很好的特征，这使得它的很多槽有默认的槽值。数学概念是非常适合于框架的刻板模式知识。由于框架有组织化的知识表示，较之逻辑或有许多规则的产生式系统更易理解，所以框架系统具有直观吸引力（Jackson 99）。

然而，在框架系统中显示出来的主要问题是允许不受限制地修改或删除槽。这个问题的典型例子是描述大象的框架，如图 2.12 所示。

name	elephant
specialization _ of	a-kind-of mammal
color	grey
legs	4
trunk	a cylinder

图 2.12 大象框架

初看大象框架似乎就是对一般大象的描述。然而，假设存在一只名叫 Clyde 的三条腿大象。Clyde 可能因为打猎意外事件少了一条腿或者可能只是为了在本书中出现它的名字而假定失去了一条腿。重要的是大象框架宣称一只大象有四条腿而不是三条。所以我们不能认为上述的大象框架就是大象的定义。

当然，这个框架能修改为允许三条腿、二条腿、一条腿甚至没有腿。但这并不是一个很好的定义。其他槽也会出现一些问题。假设 Clyde 得了黄疸病，它的皮肤变成黄色，那么它就不再是大象了吗？

把框架作为定义的另一种做法是用它来描述一只典型的大象。然而继承会导致出现其他的问题。注意到大象框架说大象是一种哺乳动物。由于我们把框架看成是典型的，所以我们的框架系统认为典型的大象是典型的哺乳动物。尽管大象是哺乳动物，但它可能不是典型的哺乳动物。从本质上看，人、牛、羊或老鼠可能更具有典型哺乳动物的特征。

很多框架系统没有提供一种定义不可变槽的方法。因为任何槽都可改变，所以框架的继承特性可以在层次结构的任何地方改变或者删除。由于不能保证特性具有普遍性，而且每一种框架有它自己的规则，这就意味着每一个框架都是初始框架。在这种不受限制的系统中无法确定任何事情，所以不可能作出具有普遍性的描述，例如那些作为大象的定义。同样，从大象这样的简单定义，不可能可靠地建立起如三条腿大象这样的混合对象。带继承的语义网中具有同样的问题，如果任何结点的特性都能改变，那么没有事物是确定的。

然而，有另一种有效的方法来看待框架。如果框架的概念扩展到包括对象属性，那么一个对象可以被看做是一个框架。CLIPS 拥有嵌入其中的完全的面向对象语言 COOL，它可以看做是基于框架语言的一个扩展。所有对象的优点目前在 CLIPS 中都具有。面向对象的专家系统可以在 CLIPS 中构建，既具有把小知识块用规则处理的优点，也有把大知识块组织成对象处理以便于维护和开发的优点。规则可以操作事实或对象，这样 CLIPS 同时具有基于规则以及基于对象的专家系统工具的优点。在 CLIPS 中使用对象使之更容易组织、执行和维护大型知识库，而不是试图把所有的知识放在独立的成

千上万个规则和事实中。

2.11 逻辑与集合

除了规则、框架和语义网，知识也可以用**逻辑**（logic）符号来表示，逻辑主要研究规则的精确推理，推理主要是是从假设中推出结论。运用计算机进行推理便出现了**逻辑程序设计**（logic programming）和基于逻辑的语言开发，如 PROLOG 。逻辑在专家系统中非常重要，推理机通过对事实进行推理而得到结论。事实上，逻辑程序设计和专家系统也称为**自动推理系统**（automated reasoning systems）。

最早的形式逻辑是由公元前 4 世纪的希腊哲学家亚里士多德提出的。亚里士多德逻辑以**三段论**（syllogism）为基础，他提出了三段论的 14 种类型。另外 5 种类型在中世纪产生。三段论有两个**前提**（premises）和一个**结论**（conclusion）。下面就是三段论的典型例子：

前提：所有人都会死
前提：苏格拉底是人
结论：苏格拉底会死

在三段论中，前提给出了结论所必须的证据。三段论是一种表示知识的方法。另一种方法是**文氏图**（Venn diagram），如图 2.13 所示。

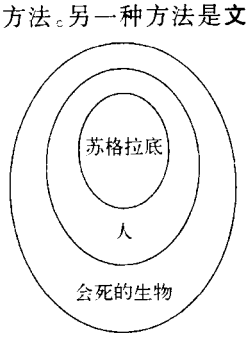


图 2.13 文氏图

图中外圈表示所有会死的生物。表示人的内圈完全在外圈之中表明所有的人都会死。既然苏格拉底是一个人，那么表示他的圈完全在第二个圈中。严格地讲，表示苏格拉底的圈应该是一个点，因为一个圈意味一种类型，但为了增强可读性，我们仍用圈表示。苏格拉底会死的结论是基于他位于会死生物的圈内而得出的。

按数学的观点，文氏图的一个圈表示一个包含对象的集合。在集合中的对象称为**元素**（elements）。以下是一些集合的例子：

A = {1,3,5}
B = {1,2,3,4,5}
C = {0,2,4 …}
D = {…, -4, -2,0,2,4, …}
E = {飞机,汽球,小飞船,喷气式飞机}
F = {飞机,汽球}

其中圆点…，称为省略符号，它表示尚未列出的项。

希腊字母∈表示一个元素属于某个集合。例如，1∈A，表示数字 1 是已定义集合 A 的一个元素。如果一个元素不属于集合，则用符号 $\notin$，如 $2 \notin A$。

对已定义的任意两个集合，如 X 和 Y，若 X 的每个元素是 Y 中的元素，那么 X 是 Y 的**子集**（subset），用 X⊂Y 或 Y⊃X 表示。从子集的定义可知，每一个集合都是它本身的子集。一个不是集合本身的子集称为**真子集**（proper subset），例如，前面定义的集合 X 是 Y 的真子集。在讨论集合时，把集合当作一个**论域**（universal set）的子集是很有用的。论域随着讨论主题的改变而变化。图 2.14 说明了“所有汽车”这个论域中两个子集所形成的**交集**（intersection）。在讨论数时，论域是所有数。论域画成一个包含其子集的矩形。使用论域不仅是为了方便，因为不分条件地定义集合会导致逻辑的自相矛盾。

假如我们定义 A 为所有蓝色汽车的集合，B 为所有手动汽车的集合，C 为所有手动的蓝色汽车。则可写

$$C = A \cap B$$

其中符号∩代表集合的交。这一表达式的另一种写法可用元素 x 表示如下：

$$C = \{x \in U \mid (x \in A) \wedge (x \in B)\}$$

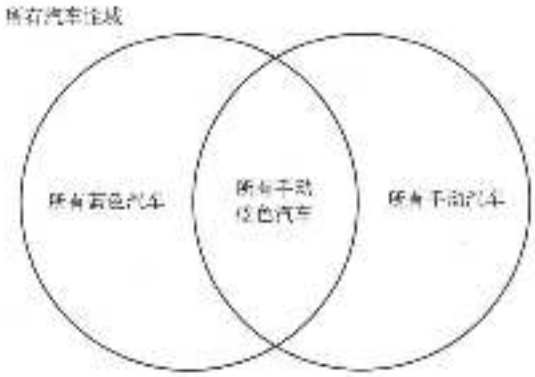


图 2.14 交集

其中

U 表示论域，

读为满足（有时用冒号：代替），

∧是逻辑与（logical AND）。

关于 C 的表达式读为：C 是由所有满足既属于 A 又属于 B 的元素 x 组成的集合。逻辑与来自布尔代数。这一表达式由两个操作数组成并由逻辑与运算符连接起来，该表达式当且仅当两个操作数为真时才为真。如果 A 和 B 没有公共元素，那么 $A \cap B = \varnothing$，其中希腊字母∅表示没有元素的集合，即空集（empty set 或 null set）。空集有时用希腊字母 Λ 表示。有时，用数字 1 表示论域而用 0 表示空集。尽管空集没有元素，它仍是一个集合。这好比一个具有顾客集合的餐馆，当没有人来时，顾客集合为空，但餐馆仍存在。

另一个集合运算是并（union）。它由或者属于 A 或者属于 B 的所有元素组成，这可表示成：

$$A \cup B = \{x \in U \mid (x \in A) \vee (x \in B)\}$$

其中，

∪是集合运算符并，

∨是逻辑或（logical OR）运算符。

集合 A 的补集（complement）是由所有不属于 A 的元素组成的集合。

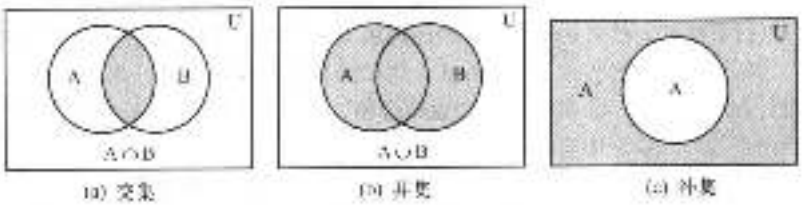
$$A' = \{x \in U \mid \neg(x \in A)\}$$

其中，

′表示一个集合的补集，

¬是逻辑非（logical NOT）运算符。

这些基本运算的文氏图如图 2.15 所示：



(a) 交集　(b) 并集　(c) 补集

图 2.15 基本集合运算的文氏图

2.12 命题逻辑

三段论是最古老且最简单之一的**形式逻辑**（formal logic），“形式”一词意味着逻辑所关心的是语句形式，而不管其意义。也就是说，形式逻辑关心的是句法而不是语义。这一点在建立专家系统中特别重要，因为你必须将推理与知识分开。以推理形式显示的可能是知识。例如，“总统永远是对的，因为他从来没错”以推理的形式出现，因为里面出现了词语“因为”连接着句子的两部分。事实上，这种类型的断言称为**重言式**（tautology）。在现实世界中，一个事实可能为真也可能为假，而重言式在纯逻辑意义上永远为真，因为它本身就证明了自己的正确性，表述为，“X 是 X”。

尽管形式逻辑一词听起来有点吓人，但它并不比代数难。事实上，代数就是数字的形式逻辑。例如，设想要你解决如下问题：一所学校有 25 台电脑，共有 60 块内存板，有些电脑有 2 块内存板，有些有 4 块，每一种类型有多少台电脑呢？这个解决方案可以写成下面的代数式

```
25 = X + Y
60 = 2X + 4Y
```

很容易解出 X=20，Y=5。

现在考虑这个问题。在院里有 25 只动物，共有 60 只脚。有些动物有 2 只脚，其他的有 4 只，每种类型的动物有多少只？你可以看到：可以写出同样的代数等式而不管我们是在讨论电脑、动物还是其他事物。代数等式使我们集中注意于符号的数学运算而不考虑这些符号代表什么，同样道理，形式逻辑使我们集中于推理而不困惑于所推理的对象。

作为形式逻辑的一个例子，用无意义的词 squeeg 和 moof 来考察三段论。

前提：所有 squeeg 是 moof
前提：John 是一个 squeeg
结论：John 是一个 moof

尽管词语 sqeeg 和 moof 是不存在的，无意义的，但这种辩论的形式仍是正确的。也就是说，因为这一三段论具有合法的形式，因此辩论是**有效的**（valid）而不管使用什么词。事实上，任一具有如下形式的三段论：

前提：所有 X 是 Y
前提：Z 是 X
结论：Z 是 Y

都是有效的而不管用什么取代 X、Y、Z。这个例子说明在形式逻辑中意义并不重要，只有形式才是重要的。把形式与意义或语义分开的观念使逻辑成为一个强有力的工具。通过分开形式与语义，就能更客观地考虑辩论的合法性，而避免了由语义引起的歧义。这有点像代数，表达式如 X+X=2X 的正确性始终是不变的，不管 X 是整数、苹果、飞机还是其他事物。

亚里士多德三段论直到 1847 年才成为逻辑学的基础，当时英国数学家乔治·布尔出版了第一本描述**符号逻辑**（symbolic logic）的书。尽管莱布尼兹在 17 世纪对自己的符号逻辑进行了修正，但它仍得不到普遍的应用。布尔提出了一个新观点，他对亚里士多德的主体存在观点，即**存在的重要性**（existential import），进行了修正。根据亚里士多德的经典观点，命题“所有美人鱼都善于游泳”既不能作为前提也不能作为结论使用，因为美人鱼不存在。布尔的现代观放宽了这一限制。现代观的重要性在于对空类也可进行推理。例如，除非有一个磁盘不合格，否则命题“所有不合格的硬盘都是便宜的”是不能用在亚里士多德的三段论中的。

布尔的另一个贡献是定义了一套**公理**（axiom），这些公理是由表达对象和类的符号以及操纵这些符号的代数运算组成的。公理是诸如数学、逻辑学这样的逻辑体系建立的基础。只能使用公理来创建定理。定理是一种可以从公理推导出而被证明的陈述。在 1910 到 1913 年，Whitehead 与 Russell 出版了他们不朽的，长达 2000 页的 3 卷著作：《数学原理》。书中指出了数学的基础是形式逻辑，这是一个重

要的里程碑。因为它为数学奠定了一个严格的基础，而不是仅凭借完全消除算术的含义来吸引人们的注意，也不是只集中精力于它们的形式和操作。正因为如此，数学一直被描述为纸上无意义的符号的集合。

然而，1931 年 Gödel 证明了建立在公理基础上的形式系统不能总被证明是一致的，因此无法避免矛盾。

命题逻辑，有时称为**命题演算**（propositional calculu），是一种用于命题操作的符号逻辑。特别地，命题逻辑针对**逻辑变量**（logical variable）进行运算，逻辑变量代表了命题。尽管大多数人按照牛顿和莱布尼兹发明的微积分学来理解 calculus，但 calculus 一词却有更一般的含义。它来自拉丁文中的词“calculus”，一种用于计算的小石头。calculus 的一般含义是：一种对符号进行操作的特别系统。此外命题逻辑有时也称为**语句演算**（statement calculus）或**句子演算**（sentential calculus）。句子一般分为 4 种，如表 2.2 所示。

命题逻辑主要考察那些或者为真或者假的陈述性句子。“一个正方形有四条边”这样一个句子的真值为真，而“乔治·华盛顿是第二任总统”这样的句子则真值为假。一个真值确定的句子称为一个**语句**（statement）或一个**命题**（proposition）。一个语句也叫做一个**封闭句子**（closed sentence），因为它的真值对任何问题都不会不确定。

如果我在这本书的序言中写道“这里的每句话都是谎言”，则这句话既不能说是真的也不能说是假的。如果它是真的，那么我在序言中讲的便是真的，这出现了矛盾。如果它是假的，那么每一句话都是真的，因此我说了谎——这又不可能是真的。诸如此类的陈述以说谎悖论最为著名，那些不能确切地回答的陈述称为**不确切句子**（open sentence），像“菠菜很美味”这样的句子也是一个不确切句子，因为它对于一些人是真的，而对另外一些人却是假的。句子“他是高的”也是一个不确切句子，因为句中包括一个变量“他”。只有当我们知道了变量所指的特定人或物时，我们才能得到该不确切句子的值，而这个句子的另外一个困难之处在于“高”一词的含义。对于一些人是高的，但对于另外一些人则不能称之为高。尽管在命题或谓词演算中这种“高”的歧义不能处理，但在模糊逻辑中却很容易处理，这一点我们将在第 5 章中讲述。

通过在语句间使用逻辑连接符，就可以形成**复合语句**（compound statement），常见的逻辑连接符列于表 2.3 中：

表 2.2 句子的类型

类 型	例 子
命令式	按我告诉你的做！
疑问式	这是什么？
感叹式	太棒了！
陈述式	一个正方形有四条边。

表 2.3 常用逻辑连接符

连接符	含 义
∧	AND；与
∨	OR；或
~	NOT，非
→	如果……则；条件
↔	当且仅当；充要条件

严格地讲，非不是连接符，因为它是一个一元运算符号，只对其后的单个操作数起作用，因此它实际上并没有连接什么。非的优先级比其他操作符都高，因此不需要对它加括号，也就是说，像～p∧q 这样一个语句的意思与（～p）∧q 是相同的。

条件符与产生式规则的箭头类似，他们都表示为 IF-THEN 的形式。例如

```
if it is raining then carry an umbrella
```

可以表示为

```
p → q
```

其中，

```
p = it is raining
q = carry an umbrella
```

有时也用“⊃”代替“→”，另外，条件符也称作**实质蕴含**（material implication）。

充要条件 p↔q 等价于

(p → q) ∧ (q → p)

只有当 p 和 q 有相同的真值时 p↔q 才成立，也就是说 p↔q 只有当 p 和 q 同时为真或同时为假时才为真，充要条件有下列的含义：

p 成立当且仅当 q 成立
q 成立当且仅当 p 成立
若 p 成立，则 q 成立，且若 q 成立，则 p 成立。

正如前面所提到的，重言式（tautology）是一永远为真的复合语句，无论组成它的每个语句是真还是假。**矛盾式**（contradiction）是一个永远为假的复合语句。**偶然式**（contingent）是一个既不是重言式也不是矛盾式的语句。重言式和矛盾式分别被称为解析真和解析假，因为它们的真值可以仅仅通过它们的形式就可决定。例如 p∨～p 的真值表说明它是重言式，而 p∧～p 是矛盾式。

若一个条件句是重言式，则称它为**蕴含式**（implication），并且用⇒代替→。若充要条件也是重言式的话，则称为**逻辑等价**（logical equivalence）或**实质等价**（material equivalence），并用符号⇔或≡来表示。两个逻辑等价的语句永远有相同的真值，例如，p≡～～p。

遗憾的是，这并不是蕴含式的唯一定义。因为对于两个可以取真或假的变量，有 16 个可能的真值表。实际上，在 20 世纪 80 年代早期的专家系统中讨论了应用于专家系统的蕴含操作符的 11 个不同定义。

在过程化语言或基于规则的专家系统中，条件句与 IF-THEN 并不完全相同。在过程化语言和专家系统中，IF-THEN 表示如果 IF 条件为真，则执行 THEN 后面的动作。在逻辑中，条件句由它本身的真值表所定义，它的意思可通过多种方法翻译成自然语言，例如，若：

p → q

这里，p，q 是任意语句，它可被翻译成：

p 蕴含 q
如果 p，则 q
p，仅当 q
p 是 q 的充分条件
q，如果 p
q 为 p 的必要条件

作为另一个例子，令 p 表示“你的年龄为 18 岁或以上”，q 表示“你有权投票”，条件句 p→q 可表示如下意思：

你的年龄为 18 岁或以上蕴含你有权投票
如果你的年龄为 18 岁或以上，那么你有权投票
你的年龄为 18 岁或以上，仅当你有权投票
你的年龄为 18 岁或以上是你有权投票的充分条件
你有权投票，如果你的年龄为 18 岁或以上
你有权投票是你的年龄为 18 岁或以上的必要条件

在某些情况下，措词上的变化对于使句子在语法上正确是必需的。最后一个例子说明，若 q 没有发生，那么 p 也不会发生。在语言中表示，你有权利投票但要求你在 18 岁或以上。

表 2.4 列出了二元逻辑连接符的真值，因为它们需要两个操作数，所以是二元连接符。非连接词“～”是一个作用于其后单个操作数的一元运算符，如表 2.5 所示。

如果每个真值函数都能仅用属于某一集合中的逻辑连接符表示，那么，这个逻辑连接符集合是**充分**（adequate）的。充分集合的例子有：{～，∧，∨}，{～，∧}，{～，∨} 和 {～，→}。

表 2.4 二元逻辑连接符的真值表

p	q	p∧q	p∨q	p→q	p↔q
T	T	T	T	T	T
T	F	F	T	F	F
F	T	F	T	T	F
F	F	F	F	T	T

表 2.5 非的真值表

p	~p
T	F
F	T

只含单个元素的集合称作**单集**（singleton），有两个充分的单集，它们是 NOT-OR（**NOR**）和 NOT-AND（**NAND**）。集合 NOR 表示为 {↓}，集合 NAND 表示为 {|}。运算符"|"称作竖线或**与非**（alternative denial），当 p 和 q 均为真时 p|q 为假，当 p 与 q 至少一个为假时，p|q 才为真。**或非**（joint denial）运算符"↓"表示当 p 或 q 为真时，p↓q 为假，也就是说，当 p 与 q 均为假时 p ↓q 才为真。

2.13 一阶谓词逻辑

虽然命题逻辑是有用的，但它具有局限性。最主要的问题是，命题逻辑仅能处理完整的语句。也就是说，它不能检查语句的内部结构。命题逻辑甚至不能证明以下三段论的合法性：

所有人会死
所有女人都是人
因此，所有女人都会死

为了分析更一般的情形，提出了**谓词逻辑**（predicate logic），其最简单的形式是**一阶**（first order）谓词逻辑，它是逻辑程序设计语言，如 PROLOG 语言的基础。在本节中，我们用“谓词逻辑”一词来表示一阶谓词逻辑。命题逻辑是谓词逻辑的一个子集。

谓词逻辑关心句子的内部结构。特别地，谓词逻辑使用一种特殊的词，即**量词**（quantifier）。如“所有”，“有些”，“没有”。这些量词是非常重要的，因为它们明确地量化了其他词，使句子的语义更为确切。所有量词均与“多少”有关，因而，比命题逻辑表示了更广阔的语义范围。

2.14 全称量词

一个使用**全称量词**（universal quantifier）的语句对于同一论域内所有成员都有相同的真值。全称量词用符号∀表示，后接一个或多个作为**域变量**（domain variable）的参数，符号∀解释为“对于每一个”或者“对于所有”。例如，在数域中：

```
(∀x) (x + x = 2x)
```

表示对于任一 x（x 为一个数），句子 x+x=2x 总为真，如果我们用符号 p 表示这个句子，那么，可更简短地表达为：

```
(∀x) (p)
```

又如，让 p 表示句子“所有狗都是动物”，则

```
(∀x) (p) ≡(∀x) (if x is a dog → x is an animal)
```

与之相反的语句为“没有狗是动物”，其写法如下，

```
(∀x) (if x is a dog → ~x is an animal)
```

此句也可以读作：

每一条狗都不是动物
所有狗都不是动物

再如，“所有三角形都是多边形”可以写成以下形式：

```
(∀x) (x is a triangle → x is a polygon)
```

读作“对于所有 x，若 x 是三角形，则 x 为多边形”具有谓词的逻辑语句可通过描述对象性质的**谓词函数**（predicate functions）来简化表达。上面的逻辑语句可写成：

$(\forall x)\ (\text{triangle}(x) \rightarrow \text{polygon}(x))$

谓词函数中的谓词通常用大写字母来表示，例如，令 T=三角形，P=多边形，那么上述语句也可简写为：

$(\forall x)\ (T(x) \rightarrow P(x))$ 或者 $(\forall y)\ (T(y) \rightarrow P(y))$

注意可用任何变量代替哑元 x 和 y，另一个例子是，用 H 表示谓词函数“人”，M 代替谓词函数“会死”。于是，语句“所有人会死”可写成：

$(\forall x)\ (H(x) \rightarrow M(x))$

读作，“对于所有 x，若 x 是人，则 x 会死”。此谓词逻辑语句可用一个语义网来表达，如图 2.16 所示。它也可以用规则表示为：

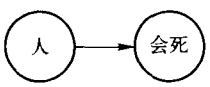


图 2.16 谓词逻辑语句的语义网表示

IF x 是人
THEN x 会死

全称量词也可解释为一个关于实例的谓词的合取。如前所述，实例是一个特定的例子。例如，一条名叫 Sparkler 的狗是狗类的一个特殊实例，于是可写成如下形式：

狗(Sparkler)

这里，“狗 ()”是一个谓词函数，Sparkler 是一个实例。

一个谓词逻辑语句，如

$(\forall x)\ P(x)$

可用实例解释为

$P(a_1) \wedge P(a_2) \wedge P(a_3) \wedge \cdots P(a_N)$

其中省略号表示谓词可延伸到类中所有成员。此语句说明谓词适用于类中的所有实例。

可使用多重量词。例如，数的加法交换律需要两个量词，如：

$(\forall x)\ (\forall y)\ (x + y = y + x)$

它表示“对于所有 x 和所有 y，x 与 y 的和等于 y 与 x 的和”。

2.15 存在量词

另一种量词叫**存在量词**（existential quantifier）。存在量词表示句子至少对论域中的一个成员为真。存在量词是表示对于论域中的所有成员句子都为真的全称量词的限制形式，存在量词写作 ∃，后接一个或多个参数。例如：

$(\exists x)\ (x \cdot x = 1)$
$(\exists x)\ (\text{elephant}(x) \wedge \text{name}(\text{Clyde}))$

第一个句子表示存在一些 x，它与自身相乘等于 1。第二个句子表示有一些大象名叫 Clyde。

存在量词有许多种读法，例如：

存在
至少一个
对某些
有一个
有些

又如，

$(\forall x)\ (elephant(x) \rightarrow four\text{-}legged(x))$

表示所有象都是四条腿的。而有一些大象是三条腿的，便可用逻辑“与”和存在量词表示为：

$(\exists x)\ (elephant(x) \wedge three\text{-}legged(x))$

正如全称量词可表示为一个合取，存在量词可表示为实例的一个析取。

$P(a_1) \vee P(a_2) \vee P(a_3) \vee \cdots P(a_N)$

用P表示“大象是哺乳动物”，在表2.6中列出了几个量词语句及其否定的例子，括号中的数字仅用于区别讨论中的各个例子。

表2.6 量词否定例子

例 子	含 义
(1a) $(\forall x)$ (P)	所有大象是哺乳动物
(1b) $(\exists x)$ $(\sim P)$	有些大象不是哺乳动物
(2a) $(\exists x)$ (P)	有些大象是哺乳动物
(2b) $(\forall x)$ $(\sim P)$	没有大象是哺乳动物

例（1a）与例（1b）互为否定，例（2a）和例（2b）也互为否定。注意，全称量词语句（1a）的否定就是存在量词语句（1b），类似地，存在量词语句（2a）的否定就是全称量词语句（2b）。

2.16 量词与集合

量词可用来定义一个论域U上的集合关系，如表2.7所示。

若A是B的一个真子集，则写作$A \subset B$，它表示所有属于A的元素都在B中，至少有一个在B中的元素不在A中。令E表示大象，M表示哺乳动物，集合关系为：

$E \subset M$

表示所有大象都是哺乳动物，但并不是所有哺乳动物都是大象。令G=灰色的，F=四足的，则“所有灰色的、四足的大象都是哺乳动物”可写作

$(E \cap G \cap F) \subset M$

表2.7 某些集合表达式及其等价逻辑表达式

集合表达式	等价逻辑表达式
$A=B$	$\forall x\ (x \in A \leftrightarrow x \in B)$
$A \subseteq B$	$\forall x\ (x \in A \rightarrow x \in B)$
$A \cap B$	$\forall x\ (x \in A \wedge x \in B)$
$A \cup B$	$\forall x\ (x \in A \vee x \in B)$
A'	$\forall x\ (x \in U \mid \sim\ (x \in A)$
U (Universe)	T (True)
$\varnothing$ (empty set)	F (False)

使用下面定义：

E=大象

R=爬行动物

G=灰色的

F=四足的

D=狗

M = 哺乳动物

以下是一些量词句例子：

没有大象是爬行动物
$E \cap R = \varnothing$
一些大象是灰色的
$E \cap G \neq \varnothing$
没有大象是灰色的
$E \cap G = \varnothing$
一些大象不是灰色的
$E \cap G' \neq \varnothing$
所有大象是灰色的和四足的
$E \subset (G \cap F)$
所有大象和狗都是哺乳动物
$(E \cup D) \subset M$
一些大象是四足的并且是灰色的
$(E \cap F \cap G) \neq \varnothing$

作为集合与逻辑形式的另一个比较，德·摩根律如表2.8所示，其中符号≡的意思是左边的语句与右边的语句有相同的真值，也就是说，两个语句是等价的。

表2.8 德·摩根定律的集合和逻辑形式

集　合	逻　辑
$(A \cap B)' \equiv A' \cup B'$	$\sim (p \wedge q) \equiv \sim p \vee \sim q$
$(A \cup B)' \equiv A' \cap B'$	$\sim (p \vee q) \equiv \sim p \wedge \sim q$

2.17 谓词逻辑的局限性

虽然谓词逻辑在许多情况下非常有用，但仍有某些类型的语句不能用使用了全称量词和存在量词的谓词逻辑来表达。例如，以下语句不能用谓词逻辑来表达：

大多数(most)同学得了A

在这个语句中，量词大多数表示多于一半。

量词大多数不能用全称量词和存在量词来表达。为了表达大多数，一个逻辑必须提供一些用于计算的谓词，如本书第5章的模糊逻辑。谓词逻辑的另一个局限是难以表达一些有时真但并非总是真的事情。这个问题也可通过模糊逻辑来解决。然而，引入计算的同时也把更多的复杂因素引进了逻辑系统中，并且使其更像数学了。

2.18 小结

在这一章，我们讨论了逻辑基础、知识表示和知识表示技术。知识表示对专家系统十分重要。讨论谬论是因为它对于知识工程师理解领域规则、避免混淆知识的形式和语义非常重要。除非规定了形式规则，专家系统才可能得到有效的结论，否则将在与人类生命和财产有关的重要系统中带来灾难。

从逻辑的角度，知识可以用许多方法来分类，例如，先验知识、后验知识；过程的、说明的、默认的知识。产生式规则、语义网、模式、框架和逻辑是专家系统中常用的知识表示方法。每一种方法都有优缺点。在设计一个专家系统之前，你应该决定用哪一种方法可以最好地解决问题。与其用一个工具去解决所有问题，倒不如对特定的问题用最好的工具。CLIPS的好处在于能够在表达知识时同时包含对象和规则。关于逻辑、知识和谬论的参考可以在附录G中获得。附录A、B和C包含了许多有用的等式、方程和集合性质。

习题

2.1 使用 AKO 和 IS-A 联系为计算机设计一个语义网。考虑以下计算机类别：微型机、小型机、大型机、巨型机、计算机系统、专用机、通用机、主板级、芯片级、单处理器、多处理器。包括特殊情况。

2.2 使用 AKO 和 IS-A 联系为计算机通讯设计一个语义网。考虑以下类别：局域网、广域网、令牌网、星形网、集中式、分散式、分布式、调制解调器、远程通讯、新闻组、电子邮件。包括特殊情况。

2.3 为建筑物设计一个框架系统。考虑办公室、教室、实验室等，你可以加入一些类别。包括要填入的槽。

2.4 设计一个可以运行的框架系统，并据此解释，如果你的计算机硬件坏了，你该怎么办。考虑软盘、电源、CPU、内存等出错情况。

2.5 画文氏图，并写出集合表达式：

(a) 两个集合的异或（exclusive-OR）。对两个集合 A、B，它由所有在一个集合中，但不在两个集合中的元素组成。异或也称**集合的对称差**（symmetric set difference），用符号/表示，例如：

$\{1, 2\} / \{2, 3\} = \{1, 3\}$

(b) 两个**集合的差**（set difference），用符号 ~ 表示。它由所有在第一个集合但不在第二个集合中的元素组成。例如：

$\{1, 2\} - \{2, 3\} = \{1\}$

这里，$\{1, 2\}$ 是第一个集合，$\{2, 3\}$ 是第二个集合。

2.6 写出真值表，并判定哪些是重言式、矛盾式、**偶然式语句**（contingent statement）。对（a）和（b），首先转换为用逻辑符号和连接符所表达的式子。

(a) 如果我及格并得了“A”那么

我及格或得了“A”

(b) 如果我及格则我得了“A”

并且

我及格且没有得“A”

(c) $((A \wedge \sim B) \rightarrow (C \wedge \sim C)) \rightarrow (A \rightarrow B)$

(d) $(A \rightarrow B) \wedge (\sim B \vee C) \wedge (A \wedge \sim C)$

(e) $A \rightarrow \sim B$ 偶然式

2.7 两个句子逻辑相等当且仅当它们具有相同的真值。因此，对两个表达式 A、B，它们的充要式 $A \leftrightarrow B$ 或恒等式 $A \equiv B$ 在任何情况下都为真，由此得到一个重言式。(a) 使用逻辑符号来表达下面两个句子并判定它们是否逻辑相等，(b) 判定它们的充要式是否为重言式。

如果你吃了香蕉片，那么你不能吃甜饼

如果你吃了甜饼，那么你不能吃香蕉片

2.8 写出与集合的差和对称差等价的逻辑表达式。

2.9 说明下列式子对任何集合 A、B、C 及空集$\varnothing$都是恒等的。

(a) $(A \cup B) \equiv (B \cup A)$

(b) $(A \cup B) \cup C \equiv A \cup (B \cup C)$

(c) $A \cup \varnothing \equiv A$

(d) $A \cap B \equiv B \cap A$

(e) $A \cap A' \equiv \varnothing$

2.10 用量词形式写出下列句子：

(a) 所有狗都是哺乳动物

(b) 没有狗是大象

(c) 有些程序有漏洞

(d) 我的程序没有一个有漏洞

(e) 所有你的程序都有漏洞

2.11 幂集（power set）P（S），是指集合 S 的所有子集的集合。P（S）至少含有空集∅，S 两个元素。

(a) 写出集合 A= {2，4，6} 的幂集。

(b) 一个具有 N 个元素的集合，它的幂集具有多少个元素？

2.12 (a) 写出下列式子的真值表：

含　义	定　义
要么 p 要么 q	$(p \vee q) \wedge \sim (p \wedge q)$
既不 p 也不 q	$\sim (p \vee q)$
p 除非 q	$\sim q \rightarrow p$
p 由于 q	$(p \wedge q) \wedge (q \rightarrow p)$
没有 p 是 q	$p \rightarrow \sim q$

(b) 说明 $(p \vee q) \wedge \sim (p \wedge q) \equiv p/q$，这里“/”即异或。

2.13 (a) 写出 NOR 和 NAND 的真值表。

(b) 通过真值表说明下面的等式，证明用 {↓} 或者 {|} 可以充分表示～、∧、∨。

$\sim\sim p \equiv p$

$(p \wedge q) \equiv (p \downarrow p) \downarrow (q \downarrow q)$

$\sim p \equiv p|p$

$(p \vee q) \equiv (p|p)|(q|q)$

(c) 由于 $p \rightarrow q \equiv \sim (p \wedge \sim q)$，用↓来表示 $p \rightarrow q$。

(d) 说明用充分单集（ⅰ）表示表达式（ⅱ）构造电路芯片的优缺点。

2.14 说明用多个领域的知识来设计专家系统的优缺点。

2.15 解释当你需要煮熟一个鸡蛋时，你如何在 Denver（丹佛）和 Houston（休斯顿）以不同的方式煮水，这是逻辑问题还是物理问题？

2.16 给出下面 PROLOG 语句，证明 Tom 是他自己的祖父：

mother (pat, ann).；pat 是 ann 的母亲

parents (jim, ann, tom)；jim 和 ann 是 tom 的双亲

surrogatemother (pat, tom).；pat 是 tom 的代孕母亲

参考文献

(Bergadano 96). *Inductive Logic Programming*, The MIT Press, 1996.

(Brewka 97). Gerhard Brewka, *Principles of Knowledge Representation,* CSLI Publications, 1997.

(Brachman 91). R. Brachman, D. McGuinness, P. Patel-Schneider, A. Borgida, and L. Resnick, Living with CLASSIC: When and How to Use a KL-ONE-Like Language, *Principles of Semantic Networks,* Morgan Kaufman, pp. 401–456, May, 1991.

(Huth 04). Michael Huth and Mark Ryan, Logic in Computer Science, 2nd Edition, Cam-

bridge University Press, 2004. NOTE: Also see software in following list for their book.

(Jackson 99). Peter Jackson, *Introduction to Expert Systems*, Addison-Wesley, Third Edition, 1999.

(Leake 96). Ed. By David Leake et al., *Case-Based Reasoning*, AAAI Press/MIT Press 1996.

(Kahane 03). Howard Kahane & Paul Tidman, *Logic and Philosophy: A Modern Introduction*, 9th edition, Wadsworth, 2003.

(Jacquette 01). Dale Jacquette, *Symbolic Logic*, Wadsworth, 2001.

(Saratchandran 96). P. Saratchandran, et al., *Parallel Implementations of Backpropagation Neural Networks on Transputers: A Study of Training Set Parallelism, Progress in Neural Processing 3*, World Scientific Pub. Co, July 1996

(Sowa 00). J.F. Sowa, *Knowledge Representation: Logical, Philosophical, and Computational Foundations*, Brooks-Cole, 2000.

(Russell 03). Stuart Russell and Peter Norvig, *Artificial Intelligence*, Second Edition, by Pearson Education, 2003.

(Werbos 94). Paul Werbos, *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting: Adaptive and Learning Systems for Signal Processing*, Wiley-Interscience, 1994.

第3章 推理方法

3.1 概述

本章将介绍关于知识的推理。由于把推理中所使用词语的含义与推理本身区别开来非常重要，所以我们采用形式的方法来讨论不同的推理方法（Russell 03）。特别地，我们将关注专家系统中的一种重要推理方法，即通过使用规则从事实推出结论。这一点在专家系统中尤其重要，因为使用推理是专家系统解决问题的基本方法（Klir 98）。

正如第1章所介绍的，专家系统广泛地用于没有合适的算法或无算法的情况下。特别是在信息不完整或缺乏的情况下，专家系统应该像人类一样能够从一系列的推理中得到结论。而一个简单的使用算法的计算机程序在缺乏参数的情况下是不能够得到问题解的。

无论如何，当无法找到最优解时，专家系统能够像人类一样做出最好的猜测。其中的缘由在于像人类一样得到一个有希望的解总比没有好。也许这个解只有最优解95%的效率，但至少比没有好得多。这样的处理方式并非首创。例如，数学中的数字计算如解微分方程的Runge-Kutta方法几个世纪来一直用于求解无法得到精确解的问题。

注意：附录A、B、C提供了本章的一些参考资料。

3.2 树、格、图

树（tree）是由**结点**（node）和**分枝**（branch）组成的层次数据结构，结点用于存储信息或知识，分枝连接各结点。有时分枝也称为**连接**（link）或**边**（edge），而结点称为**顶点**（vertice）。图3.1是一棵普通的二叉树，每个结点有0、1或2条分枝。在一棵**有向树**（oriented tree）中，**根结点**（root node）处于**最顶层**（hierarchy）而**叶结点**（leaf）在最底层。树可以看作是一种特殊类型的语义网，其中，除根结点外，每个结点只有一个**双亲**（parent）结点以及有0个或多个**子结点**（child node）。对于常用的二叉树，每个结点最多有2个子结点，并且左右子结点必须能够区分。

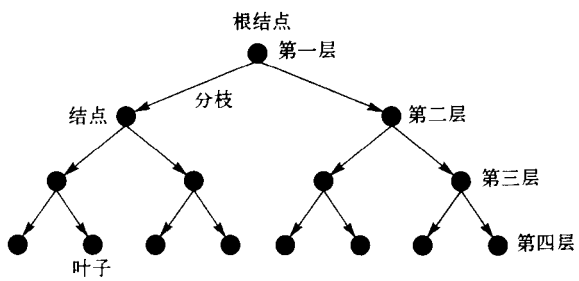


图3.1 二叉树

如果一个结点有超过一个的双亲结点，就成为网状结构。在图3.1中，从根结点到任何一个结点都只有一条**路径**（path），因为不可能逆着箭头的方向走。在有向树中，所有的箭头都是指向下的。

树是**图**（graph）的一个特例，图是一种更一般的数学结构。在用图描述特定的例子，如电话网络时，常将图称为“网络系统”或简称为“网络”。在图中结点之间可以没有或有多条连接，而且不区分双亲结点和子结点。图的一个简单例子就是地图，其中城市作为结点，公路作为连接，连接可以带有箭头或方向，也可以带有**权值**（weight）以表示连接的某些特性。比如，在一条单向行驶的路上，权值表示了此路上可通过多大的卡车。权值可代表各种类型的信息。如果图表示的是航空路线，那么权值可代表城市间的距离、飞行费用、燃料消耗等等。

人工神经系统是有环图的一个例子。因为在训练过程中，会出现从网的一层到另一层的信息反馈，并通过反馈的信息修改权值。如图 3.2（a）所示，这是一个简单的图，它没有任何边直接回到它所出发的那个结点。一个**回路**（circuit）或**环**（cycle）是指图中起点和终点都是同一个结点的路径，比如图 3.2（a）中的路径 ABCA。一个**无环图**（acyclic graph）是没有回路的。一个**连通图**（connected graph）是指所有的结点之间都有连接相通，如图 3.2（b）。带有向边的图称为**有向图**（digraph），在图 3.2（c）中有一个**自循环**（self-loop）。一个有向的无环图称为**格**（lattice），如图 3.2（d）所示。从根结点到一个叶子结点只有唯一一条路径的树叫**退化树**（degenerate tree），图 3.2（e）所示是只有三个结点的退化二叉树。一般，认为一棵树中箭头都是指向下的，所以箭头不需明确画出来。

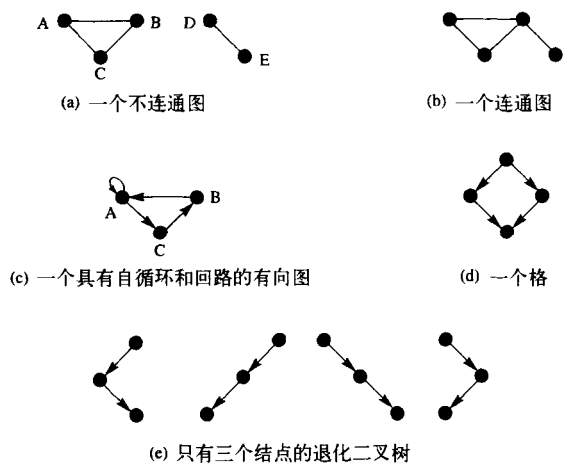


图 3.2 简单图

由于具有层次特性，即双亲在子结点之上，所以树和格通常用于对象分类。一个例子是家族树，它表明了家族的祖先以及家族成员之间的关系。树和格的另一个应用是做判定，此时称之为**判定树**（decision trees）或**判定格**（decision lattices）。我们将用**“结构”**（structure）一词来表示树和格。一个判定结构既可以作为知识表示模式，也可以作为推理的方法。图 3.3 是一个把动物分类的判定树。这个例子是为有 20 个问题的传统游戏设计的。结点包含着问题，分枝代表“是”和“否”，用以回答结点的问题。叶子则包含着猜测——这是什么动物。

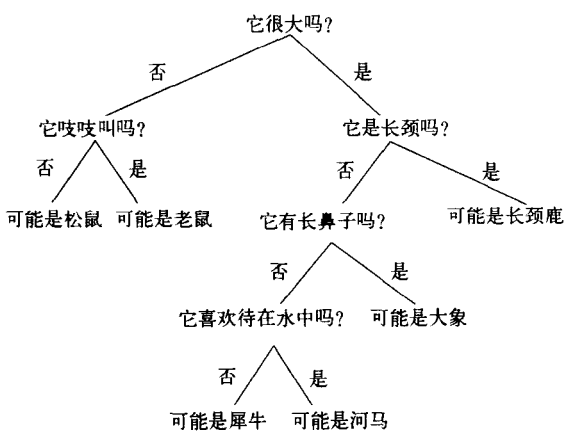


图 3.3 关于动物知识的判定树

图 3.4 是将草莓分类的判定树的一小部分。不像计算机科学中的树，分类树可把根画在下面。根

没有画出来，从根有到“单叶”结点的边和到“复叶”结点的边。判定过程从底部开始，这是通过确定粗略特性，例如叶子是单叶还是复叶来实现的。当我们向上遍历这棵树时，就需考虑更多特定的细节。于是，开始时要考虑许多可能，随着遍历的深入，可能的数目不断减少。判定树是把各种判定组织起来的好办法，它可以根据时间和精力做更详细的观察。

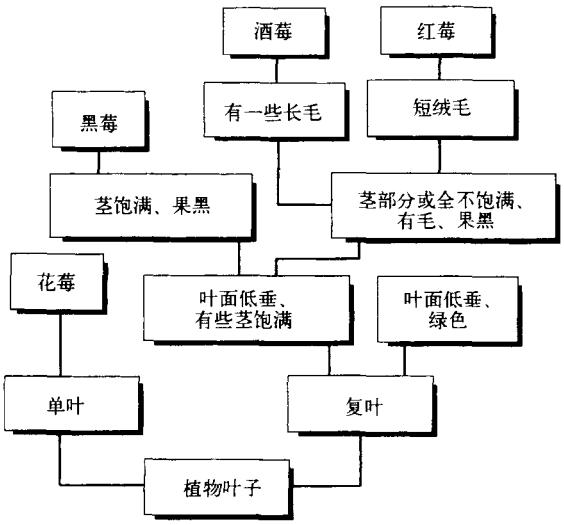


图 3.4 草莓分类判定树的一部分

如果判定是二元的，那么一棵二叉判定树不但易于构造，而且十分有效。在树中每回答一个问题就降到树的下一层。每一个问题有两个必居其一的答案，每两个问题就有 4 个可能答案中的一个，3 个问题就有 8 个可能答案中的一个等等。如果一棵二叉树的所有叶子都表示答案，而所有结点都表示问题，那么 N 个问题最多会有 2^N 个答案。如 10 个问题可分类为1 024种动物，而 20 个问题可从 1 048 576个答案中确定出一个。

判定树的另一个有用特点是可以**自学习**（self-learning）。如果推测错了，程序会询问用户一条新的、正确的分类问题，并要做出“是”或“否”的回答。新的结点、分枝、叶子会自动生成，并添加到树中。最初的动物程序是用 Basic 编写的，知识存放在 DATA 语句中。当用户告诉程序一种新的动物时，程序会自动学习并生成包含这个新动物的新的 DATA 语句。关于动物的知识可存于树中以便提高效率。在专家系统工具 CLIPS 中，程序使用建立（build）关键字来获得新的知识，因此会自动生成新的规则（Giarratano 93）。**自动知识获取**（automated knowledge acquisition）技术十分有用，因为它可以解决第 6 章中所提到的知识获取瓶颈问题。

判定结构可以机械地转变成产生式规则。这可以通过对结构进行广度优先搜索并在每个结点生成“IF-THEN”规则来实现。例如，图 3.3 判定树可以转化成以下规则：

```
IF 问题 = “它很大吗?”且回答 = “否”
THEN 问题: = “它吱吱叫吗?”
IF 问题 = “它很大吗?”且回答 = “是”
THEN 问题: = “它是长颈吗?”
```

其他的结点也是如此。一个叶子生成的是一个答案，而不是一个问题。如发现有错，程序会要求用户输入信息，并构造新的结点。

虽然判定结构是强有力的分类工具，但由于它们不能像专家系统那样处理变量，因而具有局限性。专家系统是通用的工具，而不只是简单的分类工具。

3.3 状态与问题空间

图可应用于许多实际问题上。描述对象行为的一个有效方法是定义一个称为**状态空间**（state space）的图。状态是用于确定一个对象身份或状况的特征集合，状态空间是指对象状态**转换**（transitions）所经历的所有状态，转换使对象从一种状态变到另一种状态。

状态空间例子

作为状态空间的一个简单例子，考虑要从一台机器购买饮料。当你把硬币投进机器时，机器就从一种状态转换到另一种状态。假设只有25美分及5美分的硬币是有效的，且一支饮料需55美分，图3.5说明了这种状态空间。如果允许投入诸如10美分和50美分的硬币，就会使这个例子更复杂，这里不作分析。

为了标识得更明确，开始和成功状态用双圈表示。状态以圆圈表示，从一种状态转换到另一种状态用箭头表示。注意，这个图是一个带权有向图，其中权值是各种状态下可能放进机器的硬币。

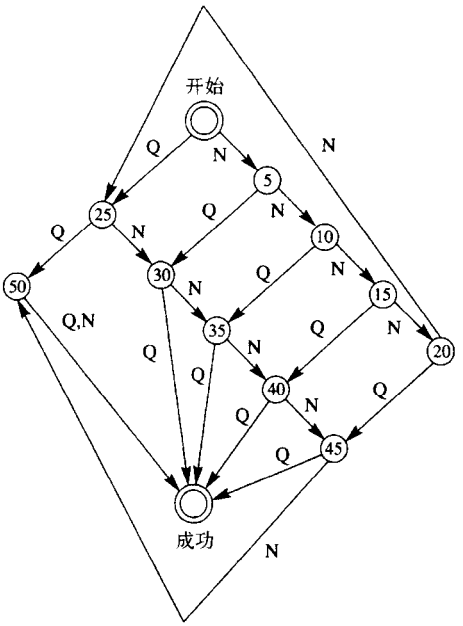


图3.5 一个只接受25美分（Q）和5美分（N）的饮料售卖机的状态图

因为描述的是一个机器的有限的状态，因而这个图也称为**有限状态机**（finite state machine）图。“机器”一词的用途非常广泛。“机器”可以是一个实在的对象、算法、概念等等。与每种状态相联系的是驱使它变成另一种状态的行为。任何时候，机器仅能处于一种状态下。当机器接受输入时，它将从一种状态推进到另一种状态。如果所给的是正确输入，机器将从开始状态不断前进到结束或成功的状态。一个状态如果没有设计为接受某个特定输入，那么机器将停止在那个状态。例如，饮料机器不能接收一角的硬币。如果某个人把一枚一角硬币投入机器中，机器的反应是无定义的。一个好的设计应考虑每种状态下所有非法输入的可能性，并使之在非法输入下转换到出错状态。出错状态要设计成可以给出适当的出错信息，并可采取某些必要的行动。

“有限状态机”常用在编译程序或其他程序中以判定输入的正确性。例如，图3.6给出了检测有效输入串的有限状态机的一部分。一次只检查一个输入字符。只有WHILE、WRITE和BEGIN这样的字符串才会被接受。图中画出了输入BEGIN正确时，从开始状态会到达成功状态，而输入不正确就会达到出错状态。为了提高效率，一些标有“L”和“T”的状态既可用于测试“WHILE”，也可用于测试“WRITE”。

注意：这里只显示了一部分出错状态转换。

状态图在描述问题的求解方法上也非常有用。在这些应用中，我们可以把状态空间看作**问题空间**（problem space），一些状态对应于问题解决的中间状态，一些状态对应于答案。在问题空间，可能有多种解决方法，相应地，有多种成功状态。在问题空间中寻找解决问题的方法就是寻找一条从开始状态（问题陈述）到成功状态（答案）的有效路径。动物判定树可作为一个问题空间，对问题“是”或“否”的回答决定了状态转换。

另一个问题空间的例子是经典的猴子和香蕉问题，如图3.7所示。这个问题是给猴子发出一些命令告诉它怎样取得从天花板上吊下来的香蕉。香蕉是够不着的，在房间里有一张椅子和一个梯子，最

初的状态是猴子坐在椅子上。所给的命令如下：

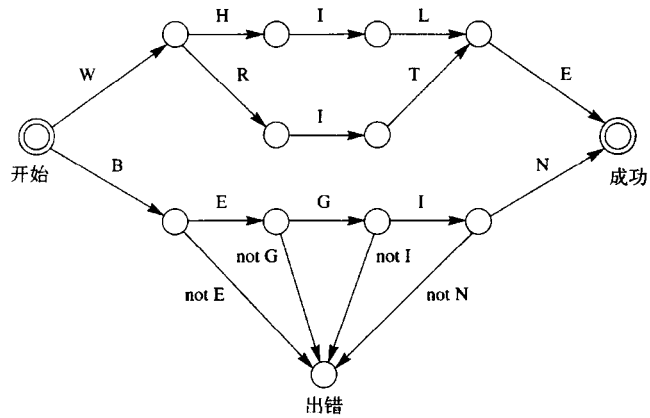


图 3.6 用于检测有效字符串 WHILE、WRITE 和 BEGIN 的有限状态机的一部分

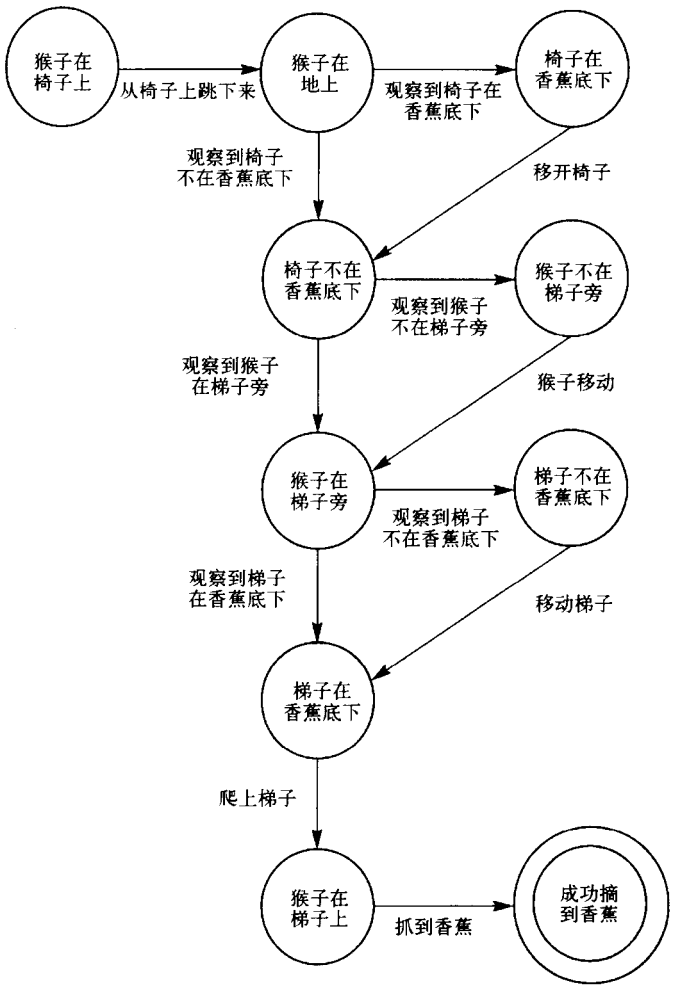


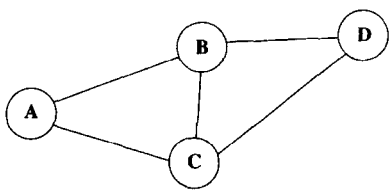
图 3.7 猴子与香蕉问题的状态空间

从椅子上跳下来
走向梯子
将梯子移到香蕉底下
爬上梯子
摘下香蕉

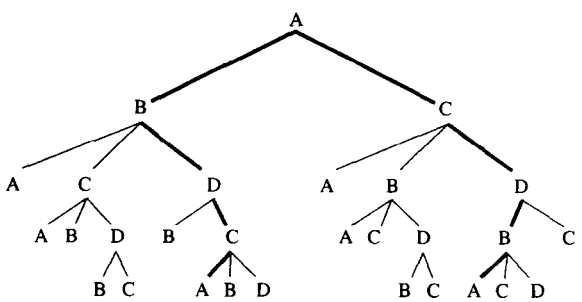
这些命令根据猴子、椅子、梯子的初始状态的不同而不同。因为有各种各样的初始状态，这里没有给出初始状态的用特殊双圈表示。例如，另一种可能的初始状态是猴子坐在香蕉底下的椅子上。猴子必须移开椅子，然后才能把梯子移到香蕉底下。最简单的初始态是猴子已经在香蕉底下的梯子上。

虽然这个问题在人看来是显而易见的，但它涉及相当多的推理。像这样的一个实用的推理系统告诉了机器人解决任务的指令。推理系统的通用方案是能够处理各种情况，而不是假定环境中的所有物体是固定不变的。一个基于规则的解决猴子和香蕉问题的方法已附在 CLIPS 光盘上。

图的另一个用途是探测路径以找到问题的解法。如图 3.8（a）所示，这个简单的网表示了旅行售货者问题（Traveling Salesman Problem）。在这个例子中，假定问题是寻找一条环路，使得从 A 点开始可以访问到所有其他结点。通常，在旅行售货者问题中，一个结点不能被访问两次。在图 3.8（b）中，以树型结构表示了所有可能从 A 点开始的路径，正确的路径是 ABDCA 和 ACDBA，图中以粗边表示。



(a) 旅行售货者问题的图



(b) 搜索路径（最佳路径用粗线表示）

图 3.8 旅行售货者问题

寻找正确路径的探测所需的回溯次数是由搜索算法决定的。例如，首先搜索路径 ABA 不成功，再退回到 B。从 B 点开始搜索 CA、CB、CDB 和 CDC 也不成功，下一个 BDB 也是不成功的搜索，直到找到第一条正确路径 ABDCA 为止。

非结构化问题空间

状态空间的一个应用是描述非结构化问题的特性。在第 1 章中，一个非结构化问题定义为该问题具有不确定性。这些不确定性能够由问题空间更加准确地指明。

作为一个非结构化问题的例子，让我们再次考虑第 1 章中讨论的例子：一个人正打算外出旅行，他到一个旅行社咨询。通过这个人对旅行社所提出问题的回答，表 3.1 列出了作为一个问题空间的这个非结构化问题的一些特性。

如果你拿表 3.1 和表 1.10 作比较，你将会发现问题空间的概念让我们更加准确地描述非结构化问题的特性。如果一个解法不行，准确地确定这些参数的特征以及考虑这个解法还需要什么是很重要的。一个问题不一定只因为它有一个、一些、甚至所有这些特性就是非结构化的，而主要取决于它的严格性（severity）。例如，所有要用定理证明方式解决的问题都有无穷多的潜在解法，但这并不能使之成为一个非结构化问题。

表 3.1 非结构化问题的旅行例子

特　性	回　答
目的地不明确	我正考虑去某个地方
没有约束的问题空间	我不知道去哪里
问题状态不离散	我只想去旅行，目的地并不重要
难以达到的必要状态	我没有足够的钱
状态算子未知	我不知道怎样弄到钱
时间限制	我必须尽快去

正如你从表 3.1 所看到的，存在着许多不确定性，然而旅行社每天都要处理它们。或许不是所有的情况都这么糟，但它表明了为什么写出一个解决问题的算法很难。

在结构化问题中我们很清楚问题、目标、运算符，以使从一个状态转到另一个状态。一个结构化问题是**确定的**（deterministic），因为当一个运算符用于一种状态时，就一定能确定下一个状态。问题空间是约束的，状态是离散的。这意味着只有数量有限的状态，且每种状态已被确定。

在旅行问题中，状态是无约束的，因为有各种可能的目的地供旅行者选择。一个相似的情况是一个模拟计程表，它可以指出无穷多的读数。如果我们将计程表的每一次读数当作一个状态，那么就有无穷多个状态，且这些状态没有很好定义，因为它们相当于实数。由于任意两个实数之间有无穷个实数，所以这些状态是不离散的，因为下一个状态有无穷多种可能。相反，数字计程表的读数是有约束的、离散的。

3.4 与或树和目标

许多类型的专家系统使用反向链来寻找问题的解法。PROLOG 是反向链系统的一个很好例子，它试图将一个问题分解成多个子问题，然后逐个解决。在 20 世纪 90 年代，PROLOG 开始广泛使用于商业和工业应用中（http://www.ddj.com/documents/s = 9064/ddj0212ai004/0212aie004.htm）解决问题就是要达到某一目标。为了达到目标，需要实现零个或多个子目标。

一种用于表示反向链的树或格是与或树。如图 3.9 所示，这是一个用与或格解决获取大学学位的简单例子。为了实现目标，必需上大学或通过函授。函授可以是邮寄形式或者通过用计算机和调制解调器联网来实现。

为了达到学位要求，必须实现 3 个子目标：(1) 申请入学；(2) 完成课程；(3) 申请毕业。请注意有一条弧横跨目标“满足要求”到其 3 个子目标的边，这条弧表明“满足要

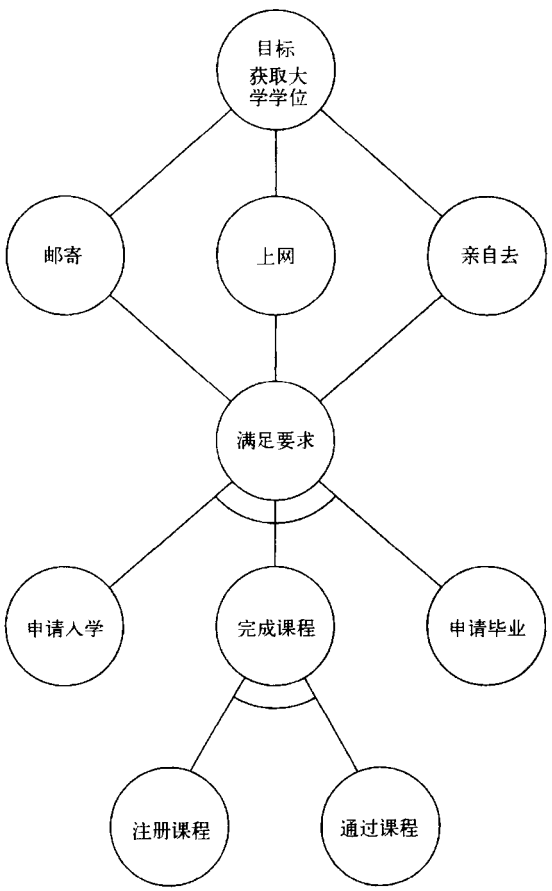


图 3.9 如何获得大学学位的与或格

求”是一个与结点，只有三个子目标都满足了它才满足。没有弧线的目标结点是或结点，例如图中的邮寄、上网、亲自去等结点。在这里只要实现其中一个子目标就可满足父结点目标，即获取大学学位。

这个图是一个格，因为“满足要求”的子目标有3个父结点：(1) 邮寄；(2) 上网；(3) 亲自去。注意可以将这个图画成一棵树，只要分别画出“满足要求”子目标与“邮寄”、“上网”、“亲自去”之间的目标子树。然而，因为“满足要求”子目标的3个父结点有相同的子目标，这样做没有优点，并且也浪费纸张。

另一个简单的例子，如图3.10所示，是用与—或树解决通过不同方式去上班的问题。为了表达完整，这棵树也可转换成格。例如，可以在“开车到火车站”结点和“汽车”结点之间加一条边，也可以在“步行到火车站”结点和“步行”结点之间加一条边。图3.11表示了一个与—异或类型的格。

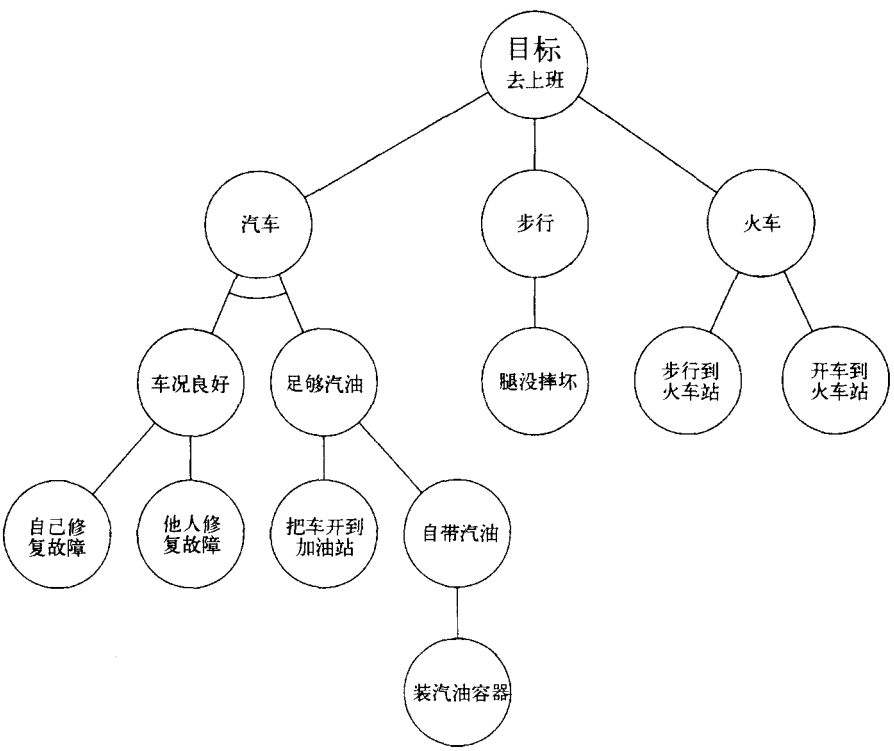


图3.10 一个显示去上班方法的简单与—或树

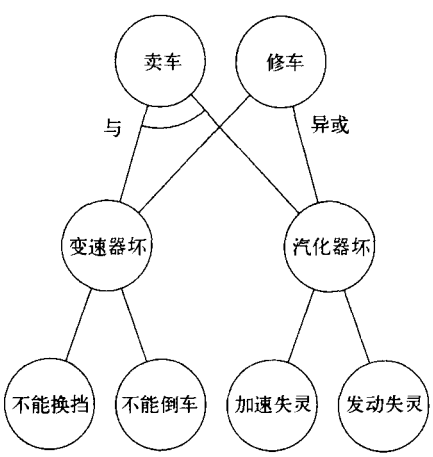


图3.11 决定卖车还是修车的与—或格

另一个描述问题解决方法的途径是与—或—非格，这种途径使用逻辑门符号代替与—或树型表达。与门、或门、非门的逻辑符号如图 3.12 所示。这些门实现了第 2 章中讨论的与、或、非真值表。图 3.13 表示了用与门及或门实现图 3.9。

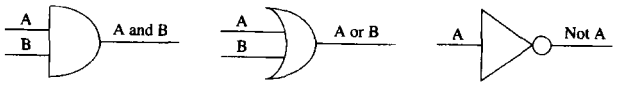


图 3.12 与、或、非逻辑门符号

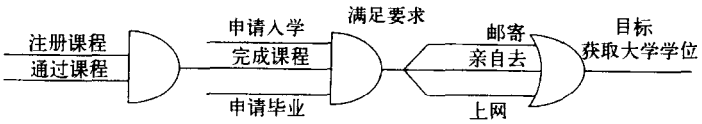


图 3.13 用与—或逻辑门表示图 3.9

与—或树和判定树基本上有共同的优缺点。与—或—非格的主要优点是它们可以在快速处理的硬件上实现。这些格可以做成集成电路。实际上，像与非（NOT-AND，NAND）这种类型的逻辑门常因制造成本较单独使用与门、或门、非门经济而使用。从逻辑角度可以证明任何逻辑功能都能用 NAND 门实现。但生产只有一种类型逻辑门的集成电路比生产具有多种类型逻辑门的集成电路便宜。

由于进行并行处理，采用正向链（forward chaining）的芯片可以把答案看作是输入的函数而快速计算出解答。这种芯片可用于对传感数据的实时监控并根据输入做出适当的反应。其主要缺点是像其他判定结构一样，一个逻辑芯片仅能处理已设计好的情况。但是，一个做在芯片上的 ANS 可以处理意料不到的输入。

3.5 演绎逻辑与三段论

在第 2 章中我们讨论了知识的逻辑表达。现在你将看到推理是如何导出新知识或信息的。在本章的剩余部分我们将讨论各种不同的推理方法。图 3.14 是推理方法的概观，可简短地归结如下：

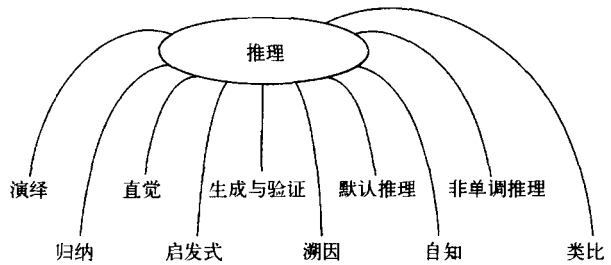


图 3.14 推理的类型

- *演绎*（Deduction）。由前提推出结论的逻辑推理。
- *归纳*（Induction）。从特殊到一般的推理。归纳是机器学习的主要方法，计算机不需要人的干预而学习。三种流行的机器学习方法是连接 Hopfield 网，ID3/ID5R 象征和基于探索的遗传算法（http：//ai.bpa.arizona.edu/papers/mlir93/mlir93.html）。
- *直觉*（Intuition）。没有证明的理论。可能无意识地发现了一个潜在的模式，于是便有了答案。专家系统还未实现这种类型的推理。ANS 可以保证这种类型的推理，因为它们能通过训练进行外推，而不只是提供一个条件响应或插值。也就是说，一个神经网络总是给出它最好的猜想作为解决办法。
- *启发式*（Heuristics）。基于经验的规则。

- 生成与验证（Generate and test）。试验与排错。为了提高效率，常与规划一起使用。
- 溯因（Abduction）。从一个正确的结论反推出可能导致这个结论的前提。
- 默认推理（Default）。在缺乏特定知识的情况下，假定通用知识。
- 自知（Autoepistemic）。自觉知识。
- 非单调推理（Nonmonotonic）。当获得新证据时，先前的知识可能不正确。
- 类比（Analogy）。基于与另一情况的相似性推出一个结论。神经元网络通过识别数据中的模式再推理出新的情况。

尽管在图 3.14 中没有明确列出，**常识性知识**（commonsense knowledge）可能是这些知识的任意组合。常识推理是人们通常使用的，却很难为计算机所掌握的推理。从 20 世纪 80 年代开始，Doug Lenat 主要尝试用常识性知识建立一个巨大的数据库供计算机推理用。而现在已经开始进行实践性的应用（http：//www.OpenCyc.org）。这个数据库包括很多断言和规则，可以用于更好地进行语音识别及更有效地构建本体等应用中。但有趣的是谚语“越大越好”并不总是正确。以下的内容来自 Doug Lenat 的私人信件，对此作了解释：

“手工输入了 300 万条规则，通过概化和精炼，我们把数量减少到 150 万。我们尽可能减少数量，而不是增加——我们可以很容易地扩展到 10 亿条规则。例如，考虑 2 种动物——有规则‘老鼠不是骆驼’。10 000种动物就可以写出 100 000 000 条这样的规则。但是，在 Cyc 中只有 10 001 条规则：Linnaean 分类关系学增加了一条规则：两种不存在从属关系的种类是相异的。‘成千上万条规则’指的是非常特别的规则，具有某种程度的复杂性，比如我刚才提到的就没有一条可以说是规则。

有一两百万条断言是由于我们不能再精简了。我们并不想将数量扩大，因为这样会使速度降低。但如果需要的话，我们可以从现有数据库中的 200 万条断言中拿出 10 001 个替换成 100 000 000 个不那么概化的规则。不过这并不好。”

关于模糊逻辑在常识性推理中的应用将在第 5 章中讨论。

最常用的推理方法是**演绎逻辑**（deductive logic），自远古时代起这种方法就已用于确定**辩论**（argument）的正确性。虽然人们一般使用辩论一词来描述一个令人生气的“意见交换”，但它在逻辑学中有非常不同的含义。一个**逻辑辩论**是一组语句，这组语句的最后一个语句基于前面**推理链**（chain of reasoning）中的语句而被证明。逻辑辩论中的一类是三段论，我们已在第 2 章中简要地讨论过了。三段论的一个例子如下：

前提：任何一个能编程的人都是聪明的
前提：约翰能编程
结论：所以，约翰是聪明的

在一个辩论中，前提是支持结论的证据。**前提**（premise）也称为**前件**（antecedent），**结论**（conclusion）也称为**后件**（consequent）。演绎逻辑的基本特征是正确的结论必须基于正确的前提。习惯上画一条线来分开前提和结论，如上所示，所以不必清晰地标出前提和结论。

以上辩论可以更简单地写成：

任何一个能编程的人都是聪明的
约翰能编程
∴约翰是聪明的

这里 3 个点，∴意思是“所以”。

让我们现在更仔细地研究一下三段式逻辑。三段论的主要好处在于它是一个简单、易理解的逻辑分支且能被完全地证明。而且，三段论是很常用的，因为它能用“IF…THEN”的规则来表达。例如，前面的三段论可改述成：

IF 任何一个能编程的人都是聪明的并且约翰能编程
THEN 约翰是聪明的

一般，三段论是一个有两个前提和一个结论的有效演绎辩论。经典的三段论是一种特殊的**范畴三段论**（categorical syllogism）。前提和结论被定义为下列4种形式的范畴语句，如表3.2所示。

表3.2 范畴语句

形 式	模 式	意 义
A	所有S是P	全肯定
E	没有S是P	全否定
I	有些S是P	部分肯定
O	有些S不是P	部分否定

注意在逻辑里，模式一词指定了语句的逻辑形式。这也说明了模式一词的另外一种用法，这种用法不同于在第2章讨论的在人工智能中的用法。在逻辑里，模式一词用于指出辩论的基本形式。模式也可以指定整个三段论的逻辑形式，如：

所有M是P
所有S是M
∴所有S是P

结论的主语S，称为**次要项**（minor term），而结论的谓语P，称为**主要项**（major term）。包含主要项的前提称为**大前提**（major premise），包含次要项的前提称为**小前提**（minor premise）。例如：

大前提:所有M是P
小前提:所有S是M
结论:所有S是P

是一个具有**标准形式**（standard form）的三段论，它已标明大、小前提。**主语**（subject）是被描述的对象，**谓词**（predicate）描述了主语的某些属性。例如，在以下语句中：

所有微机都是计算机

主语是“微机”，谓语是“计算机”。在下列语句中：

有1Gbyte的微机是有大量内存的计算机

主语是“有1Gbyte的微机”，谓语是“有大量内存的计算机”。

从远古时代起范畴语句的形式就已用字母A、E、I、O来标识。A和I表示肯定，它们来自拉丁文“affirmo”（我肯定）的最开头两个元音字母；而E和O来自“nego”（我否定）。A和I形式被认为是**本质上肯定的**（affirmative in quality），因为它们确认主语包含在谓语所在的类中。E和O形式是**本质上否定的**（negative in quality），因为主语不包含在谓语所在的类中。

动词is称为**连系词**（copula），它来自拉丁文，原意是“连接”。连系词连接语句的两部分。在标准的范畴三段论中，连系词是动词“是”（to be）的现在时形式。因此另外的一种形式是：

所有的S是(are)P

三段论的第三项M，称为**中间项**（middle term），是两个前提所共有的。中间项是必须有的，因为在一个三段论的定义中，结论不能单独从其中一个前提推出。所以辩论：

所有A是B
所有B是C
∴所有A是B

不是有效的三段论，因为它只根据第一前提推出。

量（quantity）或**量词**（quantifier）描述类所包含的部分。量词“所有”和“没有”是**全称的**（universal），因为它们指示了整个类别。量词“一些”是**特指的**（particular），因为它只指示了类的一部分。

三段论的**语态**（mood）由3个分别代表了大前提、小前提和结论形式的字母表示。例如，三段论：

所有M是P
所有S是M
∴所有S是P

是 AAA 语态。

排列 S、P、M 三项有 4 种可能的模式，如表 3.3 所示。每种模式称为一种型（figure），型的编号指出了它的类型。因此前一例子可完整地表示为 AAA-1 型三段论。一个辩论具有三段论的形式并不意味它是有效的。考虑一个 AEE-1 的三段论形式：

表 3.3 范畴语句的形式

	1 型	2 型	3 型	4 型
大前提	M P	P M	M P	P M
小前提	S M	S M	M S	M S

所有 P 是 P
没有 S 是 M
∴没有 S 是 P

它不是有效的三段论，正如从以下例子看到的：

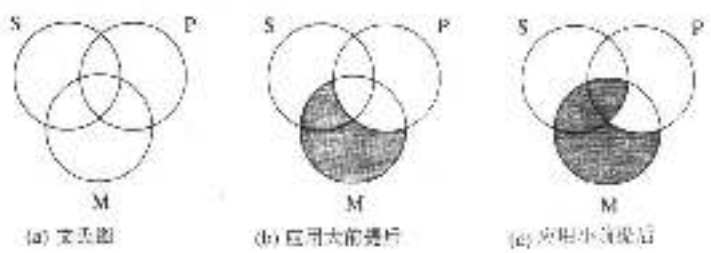
所有猫是计算机
没有人是三轮车是猫
∴没有人是三轮车是计算机

我们可以不必费神地去想例子来证明三段论辩论的有效性，现在已有判定过程（decision procedure）来进行这方面的工作。判定过程是一种证明有效性的方法，也是一种判定有效性的可自动通用机器算法。虽然已有用于三段论逻辑和命题逻辑的判定过程，但 Church 在 1936 年证明了没有可用于谓词逻辑的判定过程。人们必须应用创造力来进行证明。在 20 世纪 70 年代，如自动数学机和 Doug Lenat 的 Eurisko 等程序重新发现了哥德巴赫猜想和惟一分解定理理论的数学证明。只是数学期刊、图书馆期刊和杂志都不太报道由计算机完成的创造性工作，定理证明等。

命题的判定过程仅仅是构造一个真值表并检查它是否为重言式。三段论的判定过程可以通过使用文氏图来完成。图中用 3 个有重叠部分的圆圈代表 S、P 和 M，如图 3.15（a）所示。对于三段论形式 AEE-1：

所有 M 是 P
没有 S 是 M
∴没有 S 是 P

在图 3.15（b）中说明了其大前提。M 的画线部分表明在这部分里没有元素。在（c）中，小前提表示成在画线部分没有任何元素。由（c）可见，AEE-1 的结论是错误的，因为有一些 S 在 P 中。



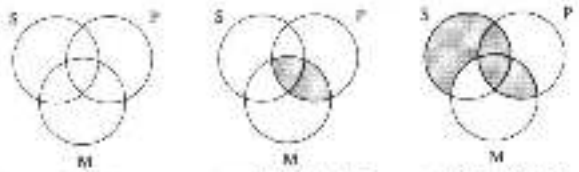
(a) 文氏图　(b) 应用大前提后　(c) 应用小前提后

图 3.15 三段论 AEE-1 的判定过程

作为另外一个例子，下面的 EAE-1 是有效的，如图 3.16（c）所示。

没有 M 是 P
所有 S 是 M
∴没有 S 是 P

具有量词"一些"的文氏图比较难画。根据布尔的观点，在 A 和 E 语句里可以没有成员，于是画范畴三段论的基本规则是：



(a) 文氏里　(b) 应用大前提后　(c) 应用小前提后

图 3.16　三段论 EAE-1 的判定过程

1. 如果类是空的，它要加上阴影。
2. 全称语句 A 和 E，要画在特指语句之前。
3. 如果类有至少一个成员，打上标记 ×。
4. 如果语句没有指明对象位于两个相邻类别中的哪一个，则在两个类间的线上打上 *。
5. 如果一个区域已打上阴影，就不能再打上 *。

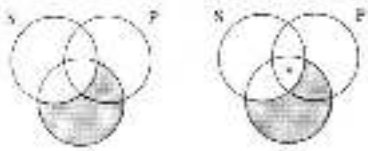
例如，

有些计算机是昂贵的(东西)
所有昂贵的(东西)是便携的(东西)
∴有些便携的(东西)是计算机

属于 IAI-4 类：

有些 P 是 M
所有 M 是 S
∴有些 S 是 P

根据文氏图的规则 2 和 1，我们从小前提的全称语句开始，对其加上阴影，如图 3.17 (a) 所示。下一步，根据规则 3，特指的大前提要打上 ×，如图 3.17 (b) 所示。由于结论“有些便携的(东西)是计算机”显示在图里，所以称论 IAI-4 是一个有效的三段论。



(a) 所有M是S　(b) 有些P是M

图 3.17　IAI-4 型的三段论

3.6　推理规则

虽然文氏图可作为三段论的一个判定过程，但对于更复杂的辩论，使用文氏图就不方便了，因为辩论越复杂文氏图就越难读。然而，对于三段论有一个更基本的问题，因为它们只代表了逻辑语句中的一小部分，特别地，范畴三段论只代表 A、E、I、O 形式的范畴语句。

命题逻辑提供了另一种描述辩论的方法。事实上，我们时常在没有意识的情况下使用了命题逻辑。例如，考虑下列的命题辩论：

如果有电源，计算机将工作
有电源
∴计算机将工作

这一辩论可以通过用字母代表命题的形式表达如下：

A－有电源
B－计算机将工作

这样辩论就可以写成

A → B
A
∴ B

这样的辩论经常都会出现。表示这类辩论的通用模式是：

```
p → q
p
∴ q
```

这里 p 和 q 是表示任一语句的逻辑变量。在命题逻辑中使用逻辑变量可以表示比 4 种三段论的形式 A、E、I、O 更复杂的语句类型。这种命题形式的推理模式有许多不同的名字：**直接推理**（direct reasoning），**假言推理**（modus ponens），**分离法则**（law of detachment），以及**假定前件推理**（assuming the antecedent）（Lakoff 00）。

注意这个例子也可以用三段论的形式来表示：

所有有电源的计算机将工作
这台计算机有电源
∴这台计算机将工作

这表明假言推理实际上是三段论逻辑的一个特殊情形。假言推理十分重要，因为它是基于规则的专家系统的基础。

然而，基于规则的系统并不仅仅依靠逻辑。因为人们不只是使用逻辑推理来解决问题。在现实世界中可能有几条竞争规则，而不仅仅是三段论中的单一规则。专家系统的规则引擎必须决定哪一个规则是合适的，就如同一个人必须决定“是吃最后一块糖果满足欲望，还是为了保持苗条而不吃?”

用 C 语言编写一系列规则并且得到专家系统工具的基本功能并不足够，尽管最简单的应用也必须这么做（http://www.ddj.com/documents/s=9064/ddj0301aie001/0301aie001.htm）。专家系统的真正威力体现在系统包括成千上万条规则并且其中含有冲突的时候。Rete 模式匹配算法是一种最强大且有效的解决规则冲突问题的算法之一，也是 CLIPS 模式匹配的基础（http://www.ddj.com/documents/s=9064/ddj0212ai002/0212aie002.htm）。

复合命题 p→q 对应于规则，而 p 对应于匹配规则前件的模式。然而，正如在第 2 章中讨论的，条件语句 p→q 并不完全等价于一个规则。因为条件语句是由真值表定义的一个逻辑定义，而这个条件语句有很多可能的定义。

通常我们会遵从常规的逻辑理论用大写字母如 A、B、C…来表示诸如“有电源”的命题常量。用小写字母如 p、q、r…表示逻辑变量，这些逻辑变量可代表不同的命题常量。注意这一习惯与 PROLOG 相反，PROLOG 用大写字母代表变量。

假言推理模式也可以使用不同名字的逻辑变量而写作：

```
r → s
r
∴ s
```

但模式的含义仍然相同。

也可用另一表示法：

```
r, r → s; ∴ s
```

这里逗号用于分开两个前提，分号表示前提的结束。尽管我们目前只讨论有两个前提的辩论，但辩论的更普遍形式是：

```
P1, P2, ... PN; ∴ C
```

这里大写字母 P_i 表示如 r，r→s 的前提，C 表示结论。注意，这与第 2 章中讨论的 PROLOG 的目标满足语句相似：

```
p :- p1, p2, ... pN.
```

如果所有子目标 p_1，p_2，…p_N 都满足，则目标 p 就满足。一个产生式规则的类似辩论可以写成如下的一般形式：

```
C1 ∧ C2 ∧ ... CN → A
```

表示如果一个规则的每个条件 C_i 都满足，那么这个规则的行为 A 就会执行。如前讨论，以上形式的逻辑语句并不严格等价于一个规则，因为条件语句的逻辑定义与产生式规则不同。然而，在考察规则时，这个逻辑形式是一个有用的直觉帮助。

与通常的 ∧ 和 ∨ 相比，逻辑运算符“与”和“或”的记法在 PROLOG 里有不同的形式。在 PROLOG 中子目标间的逗号表示合取符 ∧，而分号表示析取符 ∨。例如，

```
p :- p1; p2.
```

表示如果 p_1 或 p_2 满足，那么 p 就满足。合取符和析取符可以混合使用。例如，

```
p :- p1, p2; p3, p4.
```

与以下两个 PROLOG 语句是相同的。

```
p :- p1, p2.
p :- p3, p4.
```

尽管 PROLOG 具有强大的内置推理机制，但以这种方式创建的规则并不总是符合习惯。不过可使用 PROLOG 来执行规则，以便知识表达和推理策略可以协调起来，例如一个自动化小儿科办公系统（www.visualdatallc.com），在（Merritt 04）中有更详细的介绍。

一般地，如果前提和结论全部是模式，那么辩论：

```
P1, P2, ... PN; ∴ C
```

是一个形式有效的演绎辩论当且仅当

```
P1 ∧ P2 ∧ ... PN → C
```

是一个重言式。例如，

```
(p ∧q) → p
```

是一个重言式，因为它对 p 和 q 的任何值——T 或 F，都是真的。你可以通过构造真值表来证明这一点。

假言推理

```
p → q
p
∴ q
```

是有效的因为它能表示成一个重言式：

```
(p → q) ∧ p → q
```

注意我们假定箭头的优先级比合取符和析取符低。这样就不必另加括号而写成：

```
((p → q) ∧ p) → q
```

假言推理的真值表如表 3.4 所示。它是一个重言式，因为无论前提是什么值，位于最右一列的值都是真的。注意在第 3、4、5 列中，真值写在某一个运算符例如→和 ∧ 之下，这些运算符被称为**主要连接符**（main connective）因为它们连接了一个复合命题的两个主要部分。

表 3.4 假言推理的真值表

p	q	p→q	(p→q) ∧p	(p→q) ∧p→q
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

虽然这种判定有效辩论的方法是可行的，但它需要检查真值表的每一行。行数是 2^N，N 是前提数，因此随着前提数的增长行数迅速地增长。例如，5 个前提将需要检查 32 行，而 10 个前提需要检查 1024 行。一个判定有效辩论的较短方法是只考虑真值表里那些前提全为真的行。有效辩论的等价定义

是，辩论是有效的当且仅当结论对于这样的每一行都是真的。即，前提重言地蕴含着结论。对假言推理来说，前提 p→q 和 p 只在第一行同时取真，而结论在同一行也为真。因此，假言推理是有效的辩论。如果存在一行，其前提全是真的而结论为假，那么辩论是无效的。

表达假言推理的真值表的较短方法如表 3.5 所示，所有行都清晰地表示出来。实际上，只有那些前提为真的行，如第一行，是需要考虑的。

表 3.5 假言推理的短式真值表

		前提		结论
p	q	p→q	p	q
T	T	T	T	T
T	F	F	T	F
F	T	T	F	T
F	F	T	F	F

假言推理的真值表说明它是有效的，因为第一行有真的前提和真的结论，并且没有一行有真的前提和假的结论。

辩论有可能是欺骗性的。为了说明这点，首先考虑下例有效的假言推理：

如果没有错误,那么程序可编译
没有错误
∴程序可编译

把这和以下与假言推理有几分相似的辩论相比较：

如果没有错误,那么程序可编译
程序可编译
∴没有错误

这是不是一个有效的辩论呢？这类辩论的模式是：

p → q
q
∴ p

它的短式真值表如表 3.6 所示。

表 3.6 p→q，q; ∴p 的短式真值表

		前提		结论
p	q	p→q	q	p
T	T	T	T	T
T	F	F	F	T
F	T	T	T	F
F	F	T	F	F

注意该辩论不是有效的。虽然第一行表明如果所有前提为真，那么结论为真，但第三行却表明如果所有前提为真，而结论为假。因此该辩论不符合有效辩论的充要条件。虽然许多程序员希望这样的辩论是真的，但逻辑（和经验）证明它是**谬论**（fallacy），或者说是无效的辩论。这一特别的谬误辩论称为**逆谬论**（fallacy of the converse）。与之相反的定义在表 3.9 中。

作为另一个例子，辩论模式：

p → q
~q
∴ ~p

因为表 3.7 表示结论是真的只有当前提都是真的，所以是有效辩论。

表 3.7 $p\rightarrow q$，$\sim q$；$\therefore \sim p$ 的短式真值表

		前提		结论
p	q	$p\rightarrow q$	$\sim q$	$\sim p$
T	T	T	F	F
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

这一特别的模式有许多不同的名字：**间接推理**（indirect reasoning），**拒取式**（modus tollens），和**逆否法则**（law of contraposition）。

假言推理和拒取式是**推理规则**（rules of inference），有时也称作**推理法则**（laws of inference）。表 3.8 给出了一些推理法则。

表 3.8 命题逻辑的一些推理规则

推理法则	模式	
1. 分离法则	$p\rightarrow q$	
（Law of Detachment）	p	
	$\therefore q$	
2. 逆否命题法则	$p\rightarrow q$	
（Law of the Contrapositive）	$\therefore \sim q\rightarrow \sim p$	
3. 拒取法则	$p\rightarrow q$	
（Law of Modus Tollens）	$\sim q$	
	$\therefore \sim p$	
4. 链规则（三段论法则）	$p\rightarrow q$	
（Chain Rule，Law of the Syllogism）	$q\rightarrow r$	
	$\therefore p\rightarrow r$	
5. 析取推理法则	$p\vee q$	$p\vee q$
（Law of Disjunctive Inference）	$\sim p$	$\sim q$
	$\therefore q$	$\therefore p$
6. 双重否定法则	$\sim(\sim p)$	
（Law of the Double Negation）	$\therefore p$	
7. 德·摩根法则（定律）	$\sim(p\wedge q)$	$\sim(p\vee q)$
（De Morgan's Law）	$\therefore \sim p\vee \sim q$	$\therefore \sim p\wedge \sim q$
8. 简化法则	$p\wedge q$	$\sim(p\vee q)$
（Law of Simplification）	$\therefore p$	$\therefore q$
9. 合取法则	p	
（Law of Conjunction）	q	
	$\therefore p\wedge q$	
10. 析取附加法则	p	
（Law of Disjunctive Addition）	$\therefore p\vee q$	
11. 合取辩论法则	$\sim(p\wedge q)$	$\sim(p\wedge q)$
（Law of Conjunctive Argument）	p	q
	$\therefore \sim q$	$\therefore \sim p$

拉丁文 modus 意思是“方法”，而 ponere 意思是“肯定”，tollere 意思是“否定”。Modus 规则的真实名字和它们的字面含义如表 3.9 所示。Modus ponens 和 modus tollens 是前两种类型的简写。推理规则的编号与表 3.8 里的相对应。

表 3.9 Modus 规则的含义

推理规则的编号	名 称	含义“通过 来 ”
1	modus ponendo ponens	肯定，肯定
3	modus tollendo tollens	否定，否定
5	modus tollendo ponens	否定，肯定
11	modus ponendo tollens	肯定，否定

推理规则可应用于多于两个前提的辩论。例如，考虑下面的辩论：

芯片价格上升仅当日元上涨。
日元上涨仅当美元下跌，并且如果美元下跌则日元上涨。
因为芯片价格已经上升，所以美元一定已经下跌。

定义命题如下：

C = 芯片价格上升
Y = 日元上升
D = 美元下跌

回忆第 2.12 节，条件句的含义之一是“p，仅当 q”。像“芯片价格上升仅当日元上涨”这样的命题就有这样的含义，所以可以表示成 C→Y。整个辩论具有如下形式：

```
C → Y
(Y → D) ∧ (D → Y)
C
∴ D
```

第二个前提有一个有趣的形式，即能用条件句的变形来进行推导。条件句 p→q 有几个变形，**逆命题**（converse），**否命题**（inverse）和**逆否命题**（contrapositive）。为了完整起见，这些变形与条件句一起列在表 3.10 中。

表 3.10 条件句及其变形

条件句	p→q
逆命题	q→p
否命题	~p→~q
逆否命题	~q→~p

与通常的情况一致，否定运算符比其他逻辑运算符的优先级更高，所以~p 和~q 两边不用括号。

如果条件句 p→q 和它的逆 q→p 都是真的话，那么 p 和 q 是等价的。即 p→q∧q→p 和充要条件句 p↔q 及其等价式 p≡q 是相等的。也就是说，p 和 q 总是取相同的真值。如果 p 真那么 q 真，如果 p 假那么 q 也假。这样前面的辩论就变为：

```
(1) C → Y
(2) Y ≡ D
(3) C
    ∴ D
```

这里用数字来标识前提。由于从（2）得出 Y 和 D 是相等的，我们就可以用 D 代替（1）里的 Y 而得出

```
(4) C → D
```

这里（4）是在（1）和（2）的基础上得出的推论。现在前提（3）和（4）及结论是：

```
(4) C → D
(3) C
    ∴ D
```

这是一个假言推理的模式。因此该辩论是有效的。

用一个等价的变量来代替另一个变量的推理规则称作**代换规则**（rule of substitution）。假言推理规

则和代换规则是演绎逻辑的两个基本规则。

形式逻辑证明的书写通常都要给前提、结论和推论加上编号，如下所示：

```
1. C → Y
2. (Y → D) ∧ (D → Y)
3. C                    / ∴ D
4. Y ≡ D                2 等价
5. C → D                1 代换
6. D                    3,5 假言推理
```

1，2，3行是前提和结论，4，5，6行是得出的推论。在右边列出了推理中使用的行号和推理规则。

3.7 命题逻辑的局限性

考察一个我们熟悉的经典辩论：

所有的人(men)都会死
苏格拉底是人(man)
∴苏格拉底会死

我们知道这是一个有效的辩论，因为它是一个有效的三段论。能不能用命题逻辑来证明它的有效性呢？要回答这个问题，让我们首先把这个辩论写成模式：

令 p = 所有的人会死
q = 苏格拉底是人
r = 苏格拉底会死

那么辩论的模式是：

p
q
∴ r

注意在前提和结论中都没有逻辑连接词，所以每个前提和结论都必须有一个不同的逻辑变量。另外，命题逻辑没有提供量词，所以无法表达第一个前提中的量词“所有”。用命题逻辑对这个辩论的惟一表达方式就是以上由三个独立变量组成的模式。

为了判断这是否为一个有效的辩论，考察这三个独立变量分别取T或F的所有组合。由此组成的真值表如表3.11所示。这个表的第二行说明了辩论是无效的，因为前提都为真但结论为假。

表3.11 模式 p，q；∴r 的真值表

p	q	∴r
T	T	T
T	T	F
T	F	T
T	F	F
F	T	T
F	T	F
F	F	T
F	F	F

辩论的无效不应理解为结论不正确。任何人都知道这是一个正确的辩论。无效性只表明这个辩论不能用命题逻辑来证明。如果我们考察了前提的内部结构的话，这个辩论就能证明是有效的。例如，我们可以对“所有”赋予某种意义，还要把“men”认定为“man”的复数形式。但是，三段论和命题演算不允许人们考察命题的内部结构。可用谓词逻辑克服这个局限，这个辩论在谓词逻辑下是一个有效的辩论。事实上，所有三段论逻辑都是一阶谓词逻辑的一个有效子集，并且可以在一阶谓词逻辑下得到证明。

这个命题的唯一有效三段论形式是：

如果苏格拉底是人，那么苏格拉底会死
苏格拉底是人
∴苏格拉底会死

令：

p = 苏格拉底是人
q = 苏格拉底会死

这个辩论变成：

$p \rightarrow q$
p
$\therefore q$

这是一个假言推理的有效三段论形式。

作为另一个例子，考虑下面的经典辩论：

所有马是动物
∴一匹马的头是一只动物的头

我们知道这个辩论是正确的，但它不能由命题逻辑证明，而可以由谓词逻辑证明（见习题 3.12）。

3.8 一阶谓词逻辑

三段论逻辑可以完全由谓词逻辑来表达。表 3.12 列出了 4 种范畴语句以及它们在谓词逻辑中的表达。

表 3.12 使用谓词逻辑对 4 种范畴三段论的表达

类 型	模 式	谓 词 表 达
A	所有 S 是 P	$(\forall x)(S(x) \rightarrow P(x))$
E	没有 S 是 P	$(\forall x)(S(x) \rightarrow \sim P(x))$
I	有些 S 是 P	$(\exists x)(S(x) \wedge P(x))$
O	有些 S 不是 P	$(\exists x)(S(x) \wedge \sim P(x))$

除了具有先前讨论过的推理规则外，谓词逻辑还有处理量词的规则。

全称例化规则（the Rule of Universal Instantiation）实质上说明了可用个体来替换全称的变元。例如，如果 ϕ 是任意命题或**命题函数**（propositional function）：

$(\forall x)\ \phi(x)$
$\therefore \phi(a)$

是一个有效的推理，其中 a 是一个实例，即，a 是一个特殊的个体。x 是一个变量，可以取值为任一个体。例如，这可用于推导苏格拉底是人：

$(\forall x)\ H(x)$
$\therefore H(\text{Socrates})$

这里 H（x）是一个说明 x 是人的命题函数。上式说明对每一个 x，x 是人，由此可说明苏格拉底是人。

使用全称例化规则的其他例子还有：

$(\forall x)\ A(x)$
$\therefore A(c)$

$(\forall y)\ (B(y) \vee C(b))$
$\therefore B(a) \vee C(b)$

$(\forall x)\ [A(x) \wedge (\exists x)\ (B(x) \vee C(y))]$
$\therefore A(b) \wedge (\exists x)\ (B(x) \vee C(y))$

在第一个例子中，实例 c 替换了 x。在第二个例子，注意实例 a 替换了 y 而不是 b，因为 b 不在量词的**辖域**（scope）内。也就是说，量词如 $\forall x$ 只适用于变量 x。与量词一起使用的变量，如 x 和 y，是**约束的**（bound），而其余则是**自由的**（free）。在第三个例子中，量词 x 的辖域只是 A（x）。即 $\forall x$ 不能作用于 $\exists x$，$\exists x$ 的辖域是 B（x）$\vee$C（y）。类似这样的量词嵌套使用有如下规定：当使用一个新的量

词时，原来量词的辖域就到此结束，即使它们使用相同的变量，例如上例中的 x。在如下三段论：

所有人会死
苏格拉底是人
∴苏格拉底会死

中，令 H=人，M=会死，s=苏格拉底，则其形式证明可以写成：

```
1. (∀ x) (H(x) → M(x))
2. H(s)                    /∴ M(s)
3. H(s) → M(s)             1 全称例化
4. M(s)                    2,3假言推理
```

3.9 逻辑系统

一个逻辑系统是一个由规则、公理、语句等对象以一致的方式组织起来的集合。建立逻辑系统有如下几个目的。

第一个目的是规定辩论的形式。因为逻辑辩论从语义的角度看是无意义的，所以，如果要确定辩论的有效性，就必须有一个有效的形式。因此，逻辑系统的一个重要功能就是确定辩论中所使用的**合式公式**（well-formed formula，简写为 wff）。逻辑辩论中只能使用 wff。例如，在三段论逻辑中，

All S is P

是一个 wff，但

All
All is S P
Is S all

就不是 wff。虽然字母的符号是无意义的，但组成 wff 的符号序列是有意义的。

逻辑系统的第二个目的是规定有效的推理规则。第三个目的是通过发现新的推理规则来扩充其自身，并扩充其可以证明的辩论的范围。通过扩充辩论的范围，新的 wff，又称为**定理**（theorems），就可以由逻辑辩论来证明。

一旦逻辑系统适当地建立之后，辩论的有效性就可以由像算术、几何学、微积分学、物理学以及工程学等系统上的计算一样来确定。已经建立了一些如句子或命题演算、谓词演算等的逻辑系统。每一个系统都依赖于其**公理**（axiom）或**假定**（postulate）的形式定义，它们是系统的基本定义。从这些定义出发，人们（有时是计算机程序，如 AM）试图确定什么是可以被证明的。在中学里学过欧几里得几何学的人都会熟悉其公理和推导出的几何定理。正如几何定理可以由几何公理推导出来一样，逻辑定理也可以由逻辑公理推导出来。

一个公理只不过是一个不能在系统中被证明的事实或**断言**（assertion）。有时我们承认某些公理，因为它们符合常识或观察而显得有“意义”。其他公理，例如“平行直线相交于无穷远处”，在直觉上并不显得有意义，因为它们看来与欧几里得的平行直线永不相交的公理相矛盾。然而，这个平行直线相交于无穷远处的公理，从纯粹的逻辑观点来看，是和欧几里得的公理一样合理的，并且它是一类非欧几何学的基础。

一个形式系统要求具有如下的组成：

1. 一个符号字母表。
2. 一个由有限长的符号串，即 wff 组成的集合。
3. 公理，即系统的定义。
4. 推理规则，这些规则可以使一个 wff，A，成为从其他 wff 的有限集合 G 推出的结论，这里 $G=\{A_1, A_2, \cdots A_N\}$，这些 wff 必须是公理或逻辑系统中的其他定理。例如，一个命题逻辑系统可以只使用假言推理来得出新的定理。

如果辩论：

$A_1, A_2, \ldots A_N; \therefore A$

是有效的，那么称A是这个形式逻辑系统的一个定理，并用符号⊢表示。例如，Γ⊢A表示A是wff集合Γ的一个定理。以下是表明A为定理的一种更清楚的证明模式：

$A_1, A_2, \ldots A_N \vdash A$

符号⊢表示其后的wff是一个定理，它不是系统中的一个符号，而是一个元符号（metasymbol），因为它是用于描述系统自身的。一种类似的情况是计算机语言，如Java。虽然可以用Pascal的语法来规定一个程序，但Java中没有用语法具体规定什么是一个有效的程序。

形式系统中的推理规则明确地规定了新的断言，即定理，怎样从公理以及先前得到的定理中推得。一个定理的例子是我们关于苏格拉底的三段论，它可以用谓词逻辑的形式写为：

$(\forall x)\ (H(x) \rightarrow M(x)),\ H(s) \vdash M(s)$

这里H是代表人的谓词函数，M是代表会死的谓词函数。因为M（s）可以由它左边的公理推导得出，所以它是这些公理的一个定理。但是，要注意M（Zeus）不是一个定理，因为Zeus（宙斯）是希腊天神，不是人，也没有其他方法来证明M（Zeus）。

如果一个定理是一个重言式，那么Γ可以是一个空集，因为该wff总是为真，而不依赖于任何公理或定理。用符号⊨表示一个为重言式的定理，如⊨A。举个例子：如果A≡p∨~p，那么⊨p∨~p表示p∨~p是一个为重言式的定理。注意不论给p赋予T或F，定理p∨~p总是真的。指派一个wff的真值就是对其进行**解释**（interpretation）。一个**模型**（model）就是当wff为真时的解释。例如，p→q的一个模型是p=T且q=T。如果存在一个解释使一个wff为真，则称它是**相容的**（consistent）或**可满足的**（satisfiable）；如果这个wff在所有解释中都为假，则称为**不相容的**（inconsistent）或**不可满足的**（unsatisfiable）。p∧~p就是一个不相容的wff。

一个wff如果在所有解释中都为真，则它是**有效的**（valid），反之是**无效的**（invalid）。例如，p∨~p是一个有效的wff，而p→q是一个无效的wff，因为当p=T且q=F时它不为真。如果一个wff被论证为有效的，它就是**被证明了**（proved）。所有wff命题都可以用真值表的方法来证明，由于每个wff只存在有限个解释，所以命题演算是**可判定的**（decidable）。然而，谓词演算是不可判定的，因为对所有谓词演算的wff，不存在像真值表那样的通用的证明方法。

下面是一个可以证明的有效的谓词演算wff的例子：对任意谓词B，

$(\exists x)\ B(x) \rightarrow \sim [(\forall x) \sim B(x)]$

它说明了如何由全称量词替换存在量词。这个谓词演算wff是一个定理。

在表达式⊢A和⊨B之间，有一个很大的区别。A是一个定理，因此可以从公理出发通过推理规则证明。B是一个wff，可能不存在证明来说明它如何为真。命题逻辑是可判定的而谓词逻辑却不是。也就是，不存在机械的过程或算法可以在有限步内找出一个谓词逻辑定理的证明。事实上，理论上可以证明谓词逻辑不存在判定过程。但是，谓词逻辑的子集，比如三段论和命题逻辑，却存在判定过程。因此，谓词逻辑有时称为是**半可判定的**（semidecidable）。

注意PROLOG是基于谓词逻辑。在20世纪70年代PROLOG的第一轮狂热中，日本宣布了他们的第五代超级计算机系统计划。该系统将允许用自然语言输入和输出，并且能真正理解其中的含义。但是，当时既没有足够的计算机化常识知识，又没有如20世纪80年代那样足够强大的微处理器提供高精度的声音识别。现在，Open.Cyc系统既有常识本体又有高精度的无需训练的声音识别系统，这使得成功的机会大大增加（尽管不是使用PROLOG）。

作为完全的形式系统的一个很简单的例子，我们给出以下的定义：

符号字母表：单字符“1”

公理：字符串“1”（正好与符号1相同）

推理规则：如果任意字符串 $ 是定理，那么 $ 11 也是定理。这个规则可以写为一个 Post 产生式规则，

```
$ → $ 11,
如果 $ = 1,那么这个规则就给出 $ 11 = 111,
如果 $ = 111,那么根据这个规则就有 $ 11 = 11111,以此类推就有
      1,111,11111,1111111,...
```

这些字符串就是这个形式系统的定理。

虽然 11111 等看起来不像我们习惯见到的定理类型，但它们完全是有效的逻辑定理。这些特殊定理还有一个语义上的意义，因为它们都是在**一进制**（unary number system）中用 1 来描述的奇数。与二进制（binary number system）只有符号 0 和 1 类似，一进制只有单个符号 1。一进制和十进制数字的对应关系为：

一进制	十进制
1	1
11	2
111	3
1111	4
11111	5

以此类推。

注意根据我们的公理和推理规则，字符串 11，1111 等都不能由我们的形式系统来表达。即是说，虽然 11 和 1111 可由我们的形式符号表生成，但它们不是定理或 wffs，因为它们不能仅仅使用推理规则和公理就可证明。这个形式系统只能推导出奇数，不能推导出偶数。如果要推导出偶数，就必须增加一条公理“11”。

形式系统的另一个特性是**完备性**（completeness）。如果每个 wff 都可以被证明或**反驳**（refuted），那么这个公理集合就是**完备的**（complete）。“反驳”一词的意思是证明某些断言为假。在一个完备系统中，每个逻辑有效的 wff 就是一个定理。但是，谓词逻辑是不可判定的，要得到一个证明就要靠我们的运气和智慧。当然，另一个证明的方法就是写一个可用于得到证明的计算机程序，并让它不断运行。

另一个希望逻辑系统具备的特性是**合理性**（sound）。一个合理的系统是指每一个定理都是一个逻辑上有效的 wff。换句话说，一个合理的系统不会存在一个不是它前提的逻辑推论的结论。无效的辩论不会被推导为有效。

逻辑有不同的**阶**（order）。**一阶**（first-order）语言规定量词可以对对象，即变量进行操作，如 ∀x。**二阶**（second-order）语言增加了新的特征，比如它有两类变量和量词。除了具有序变量和量词外，二阶逻辑还可以具有作用于函数和谓词符号的量词。一个二阶逻辑的例子是**等式公理**（equality axiom），它表示如果两个对象的所有谓词都相等，那么这两个对象就相等。如果 P 是一个辩论中的任意谓词，那么

$$x = y \equiv (\forall P)\ [P(x) \leftrightarrow P(y)]$$

是一个使用二阶量词 ∀P 的等式公理语句，这里 ∀P 作用于所有谓词。

3.10 归结

Robinson 在 1965 年引入了一个十分有效的**归结**（resolution）规则，它常用于实现定理证明的 AI 程序中。事实上，归结是 PROLOG 语言中的主要推理规则。PROLOG 使用一个通用的归结推理规则而不是许多不同的适用性有限的推理规则，例如假言推理、拒取、合并、链规则等。应用归结使得自动定理证明器，如 PROLOG 实用工具可以求解问题。只需使用一个归结规则，而不必尝试不同的推理规则并希望其中有一个会成功，这种方法极大地缩小了搜索空间。

作为介绍归结的一种方法，让我们首先考虑关于苏格拉底的三段论，用 PROLOG 表述如下，其中用百分号表示注释：

```
mortal (X) :- man (X).          % 所有人会死
man (socrates).                 % 苏格拉底是人
:- mortal (socrates).           % 问:苏格拉底会死吗?
yes                             % PROLOG 回答是
```

PROLOG 使用一种**无量词**（quantifier-free）表示法。注意全称量词∀已隐含在语句“所有人都会死”中。

PROLOG 是基于一阶谓词逻辑的。但是，它也作了许多扩充以便更易应用于程序设计中。这些特别的程序设计特性违背了纯粹的谓词逻辑，故被称为**扩展逻辑特性**（extralogical feature），包括：输入/输出，截断（改变搜索空间）和断言/取消（不用任何逻辑证明地修改真值）。

在应用归结之前，wff 必须是一个范式（normal form）或标准形式。范式有三种主要类型：**合取范式**（conjunctive normal form）、子句以及 Horn 子句子集。范式的基本思想是用一种只使用∧，∨或可能包括~的标准形式来表达 wff。然后将归结方法用于那些所有其他连接词和量词都已删除的 wff 范式上。由于归结是在析取式对上进行的操作，并产生新的析取式，进而简化 wff，因此归结前必须先转换成范式。

下面是一个合取范式形式的 wff，合取范式是一些**文字**（literal）析取式的合取。

$$(P_1 \vee P_2 \vee \ldots) \wedge (Q_1 \vee Q_2 \vee \ldots) \wedge \ldots (Z_1 \vee Z_2 \vee \ldots)$$

其中每一个项如 P_i，必须是文字，表示它们不包含诸如条件符、充要条件符、或量词等逻辑连接词。一个文字是一个原子公式或原子公式的非。例如以下的 wff：

$$(A \vee B) \wedge (\sim B \vee C)$$

就是一个合取范式。括号中的项：

$$A \vee B \text{ 和 } \sim B \vee C$$

是一个子句。

以后将会看到，任何包含命题逻辑作为特例的谓词逻辑 wff 都可以写成子句。**全子句形式**（clausal form）可以表达任何谓词逻辑公式，但它对于人来说可能是不自然的或不可读的。PROLOG 的语法是 Horn 子句子集，这使得 PROLOG 在实现定理机器证明时比标准的谓词逻辑表示法或全子句的形式更容易和更有效。正如在第 1 章提到的，PROLOG 只允许一个头。一个全子句形式的表达式通常被写成一种称为 Kowalski 子句的特殊形式：

$$A_1, A_2, \ldots A_N \rightarrow B_1, B_2, \ldots B_M$$

上式解释为如果所有子目标 A_1，A_2，…，A_N 是真的，那么 B_1，B_2，…，B_M 中的一个或更多个也是真的。注意在这种表示法中箭头的方向有时是相反的。将这个子句写成标准的谓词表示法就是：

$$A_1 \wedge A_2 \ldots A_N \rightarrow B_1 \vee B_2 \ldots B_M$$

使用等式

$$p \rightarrow q \equiv \sim p \vee q$$

可以将其表示为下面的文字**析取式**（disjunctive form）：

$$\begin{aligned} A_1 \wedge A_2 \ldots A_N &\rightarrow B_1 \vee B_2 \ldots B_M \\ &\equiv \sim(A_1 \wedge A_2 \ldots A_N) \vee (B_1 \vee B_2 \ldots B_M) \\ &\equiv \sim A_1 \vee \sim A_2 \ldots \sim A_N \vee B_1 \vee B_2 \ldots B_M \end{aligned}$$

这里使用了德·摩根定律：

$$\sim(p \wedge q) \equiv \sim p \vee \sim q$$

来简化最后一个表达式。

正如在第 1 章中讨论的，PROLOG 使用一种受限的子句形式，即 Horn 子句，这种子句只允许一

个头：

$A_1, A_2, \ldots A_N \rightarrow B$

用 PROLOG 语法可以写为：

```
B :- A1,A2,... AN
```

直接证明定理存在一个问题，即只用系统的推理规则和公理进行推理会很困难。推导一个定理可能要花很长时间，或者我们根本不够聪明而无法推导。证明一个定理为真，常常使用经典的**反证法**（reductio ad absurdum）或矛盾法。用这个方法我们试图证明 wff 的否定是一个定理。如果得出矛盾的结果，则原来的 wff 是一个定理。

归结的基本目标是从两个称为**根子句**（parent clause）的子句推导出一个新的子句，即**消解式**（resolvent）。消解式比它的根有较少的项。通过归结过程的不断进行，最终会得出矛盾，或者因为不再有进展而终止。在下面的辩论中给出了一个简单的归结例子。

```
A ∨ B
A ∨ ~B
∴ A
```

要想知道是如何得出结论的，可以将前提写成：

```
(A ∨ B) ∧ (A ∨ ~B)
```

利用分配公理：

```
p ∨ (q ∧ r) ≡ (p ∨ q) ∧ (p ∨ r)
```

可得：

```
(A ∨ B) ∧ (A ∨ ~B) ≡ A ∨ (B ∧ ~B) ≡ A
```

这里最后一步的推导是因为（B∧～B）总是为假。它是由排中律（the Law of the Excluded Middle）得出的，即某事不能既真又假。在第 5 章讨论的模糊逻辑中，我们将会看到排中律并不成立。另一种书写排中律的方法是使用表示空、无或假的项 nil 或 null。例如，null 指针在 C 语言中表示不指向任何地方，于是排中律可以表示为（B∧～B）≡nil。

上述归结例子中表示了怎样将根子句（A∨B）和（A∨～B）简化为消解式 A。表 3.13 总结了子句表示法中一些基本的根子句和它们的消解式，这里分隔子句的逗号表示∧。

表 3.13 子句和消解式

根 子 句	消 解 式	意 义
p→q, p	q	假言推理
或～p∨q, p		
p→q, q→r	p→r	链规则或假设三段论
或～p∨q, ～q∨r	或～p∨r	
～p∨q, p∨q	q	合并
～p∨～q, p∨q	～p∨p	真（重言式）
	或～q∨q	
～p, p	nil	假（矛盾）

3.11 归结系统与演绎

给定 wffs A_1，A_2，…A_N 和它们的逻辑推论或定理 C，我们知道：

$A_1 \wedge A_2 \ldots A_N \vdash C$

可以等价地表示为：

$(1)\ A_1 \wedge A_2 \ldots A_N \rightarrow C \equiv \sim(A_1 \wedge A_2 \ldots A_N) \vee C$

$\equiv \sim A_1 \vee \sim A_2 \ldots \sim A_N \vee C$

考虑如下的否定式：

$$\sim[A_1 \wedge A_2 \ldots A_N \rightarrow C]$$

由于

$$p \rightarrow q \equiv \sim p \vee q$$

于是上式变为：

$$\sim[A_1 \wedge A_2 \ldots A_N \rightarrow C] \equiv \sim[\sim(A_1 \wedge A_2 \ldots A_N) \vee C]$$

根据德·摩根定律：

$$\sim(p \vee q) \equiv \sim p \wedge \sim q$$

上式又变为：

$$(2)\ \sim[A_1 \wedge A_2 \ldots A_N \rightarrow C] \equiv [\sim\sim(A_1 \wedge A_2 \ldots A_N) \wedge \sim C] \equiv A_1 \wedge A_2 \ldots A_N \wedge \sim C$$

如果（1）式有效，则它的否定式一定无效。换句话说，如果（1）是重言式则（2）必定是矛盾式。公式（1）和（2）说明了两种证明公式C是定理的等价的方法。公式（1）证明定理的方法是通过检查它是否在所有情形下都为真。而公式（2）证明定理的方法是通过说明（2）会导致矛盾。

正如前面小节提到的，通过说明它的否定会导致矛盾来证明一个定理的方法叫反证法。这种证明方法的主要部分就是**反驳**（refutation）。反驳某事就是证明某事是假的。归结是一个合理的推理规则，它也是**反驳完备的**（refutation complete）的，因为如果在子句集合里有矛盾，那么最终就会得到空子句。本质上，这意味着如果存在矛盾，**归结反驳**（resolution refutation）就会在有限的步骤内停止。虽然归结反驳不能告诉我们如何产生定理，但它将明确地告诉我们一个wff是否为定理。

作为用归结反驳证明的一个简单例子，考虑以下的辩论：

$$A \rightarrow B$$
$$B \rightarrow C$$
$$\underline{C \rightarrow D}$$
$$\therefore A \rightarrow D$$

要用归结反驳法证明结论A→D是一个定理，首先利用等式：

$$p \rightarrow q \equiv \sim p \vee q$$

将其转换成析取式，于是，

$$A \rightarrow D \equiv \sim A \vee D$$

而它的否定式是：

$$\sim(\sim A \vee D) \equiv A \wedge \sim D$$

前提的析取式和结论的否定式的合取一起形成了适于归结反驳的合取范式。

$$(\sim A \vee B) \wedge (\sim B \vee C) \wedge (\sim C \vee D) \wedge A \wedge \sim D$$

归结方法现在就可以用于前提和结论。图3.18给出了一个用**归结反驳树**（resolution refutation tree）来表达归结反驳法的例子，这里位于同一层的子句就会被归结。最后归结得到的根是空的，它可根据表3.13中关于~p，p的最后一行得出，因此原来的结论A→D是一个定理。

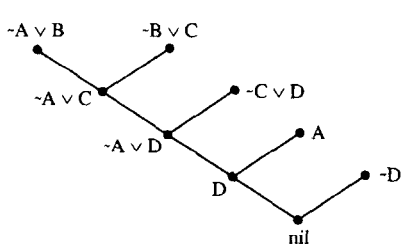


图3.18 归结反驳树

3.12 浅推理和因果推理

归结系统和产生式规则系统是用于定理证明的两个应用较广的范例。虽然大多数的人会从数学的角度去思考一条定理，但是一条定理实际上是一个有效逻辑辩论的结论。现在让我们考虑一个使用推理链的专家系统。通常，较长的链代表了有较多的因果知识。而浅推理一般使用单一规则或少量的推理。除推理链的长度以外，规则中所包含的知识的质量亦是区分**浅推理**（shallow reasoning）和**深推理**（deep reasoning）的一个主要因素。有时也称浅知识为**经验知识**（experiential knowledge），它是基于经验的知识。

一条推理链的结论是一条定理，因为它可以通过该推理链得到证明。如前述例子所示：

$A \rightarrow B,\ B \rightarrow C,\ C \rightarrow D \vdash A \rightarrow D$

使用推理链得到结论的专家系统实际上是在使用定理。这个结论十分重要，否则就无法用专家系统来进行因果推理，而只能用无链的单一规则来进行浅推理。

为了更好地区别浅推理和深推理，让我们来看一些规则。作为第一个例子，思考以下规则，其中圆括号中的数字仅用作标识：

(1) IF 一部车有
好的电池
好的火花塞
汽油
好的轮胎
THEN 这部车子能跑

这是一条适用于专家系统的完美的规则。

如第 1 章所述，专家系统的一个重要特征是解释机。基于规则的专家系统能方便地对其推理做出解释。在本例中，若用户问为什么车能跑，专家系统会列出其条件中的各元素来回答：

好的电池
好的火花塞
汽油
好的轮胎

这是一种最基本的解释机类型，因为系统只需列出规则中的各项条件即可。更复杂的解释机可以是列出前面所用的各规则并以此推出正在使用的规则。另一种解释机是允许用户提出“如果……会怎样?”的问题，以探索其他推理路径。

上述规则也是浅推理的一个例子。也就是说，由于很少或没有推理链，因此在浅推理中很少或根本无须理解因果关系。上述规则实质上是启发式的，其所有知识都包含在规则中。当所有条件均满足，规则就被激活，而不是因为专家系统明确了各条件的功能后才被激活。在浅推理中，很少甚至没有由某一规则推至另一规则的**因果链**（causal chain）。在最简单的情形，因和果都被包含在与其他规则毫无关联的一条规则中。如果像我们在第 1 章中讨论的那样，将规则认为是知识块，那么浅推理就像一个简单的反射一样，在块与块之间无任何联系。

相对于因果推理而言，浅推理的优点是易于编程。易于编程意味着只需较短的开发时间，程序小，速度快，并且只需较少的开发经费。

框架和语义网络是适用于因果或深推理的两种模型。“深”一词常用作因果推理的同义词，表示了对问题的更深层次的理解。然而，深层次的理解意味着不仅要理解过程发生的因果链，还要从更抽象的层次去理解这一过程。

我们可以通过定义如下的规则来为我们的规则增加简单的因果推理：

(2) IF 如果电池是好的

THEN 有电
(3) IF 有电
并且火花塞是好的
THEN 可以打火
(4) IF 可以打火
并且有汽油
THEN 发动机能工作
(5) IF 发动机能工作
并且轮胎是好的
THEN 汽车能跑

注意，由于对汽车每一部件的工作都用一条规则进行了准确说明，所以具有因果推理的解释机可以很好地解释汽车的各个部件能做什么。这样一个因果系统也更容易构造一个诊断系统来决定一个坏的部件将会造成怎样的影响。因果推理可应用在一个执行速度，存储容量，开发经费均受限的系统中，对其操作做任意的改进。

因果推理可用于构造一个各方面都和真实事物行为相一致的真实系统**模型**（model）。这样一个模型可用于模拟探索“如果……会怎样”式询问的假设推理。然而，因果模型并不总是必需的。例如作为钻孔液顾问的 MUD 系统。因钻孔液与泥浆（mud）很相似故称之为 MUD，它可以冷却和润滑钻头，是勘探的重要用品。MUD 系统诊断出钻孔液问题并建议解决方法。

由于钻探工程师并不能观察到地底深处所发生的事情之间的因果联系，相反，他只能观测到表面的征兆，不能观测到对诊断产生潜在影响的中间事件，因而因果系统并不是那么有用。

在医学方面，情况就大不相同了。医生可通过大范围的诊断测试来核实中间事件。例如，若病人抱怨不舒服，医生可帮他检查是否发烧；如有发烧，则有可能受了感染，因而需进行血液检查；若血液检查表明病人感染了破伤风，医生会检查病人最近有否被生锈物体割伤。与之相反，若钻孔液含盐，钻孔工程师就会猜测探头可能经过了盐层。然而，由于并不能进入洞中，而地质测试又十分昂贵且未必可靠，因此并没有直接方法可以检验这一猜测。由于钻孔工程师不能像医生那样检测中间假设，因而他们也不能采用医生那样的方式进行问题诊断。MUD 正反映了这一点。

在 MUD 中不使用因果推理的另一个原因是诊断的可能性和征兆的个数都是有限的。大部分在 MUD 中使用的相关测试均按常规步骤建立并预先存入。如果系统知道所有的相关测试和诊断途径，那么工程师采用交互询问的方式来寻求一条诊断途径就毫无优势可言了。如果某一可被证实的中间假设会导致许多可能的诊断途径，那么因果知识就具有优势，因为工程师可与系统一起工作以便修正可能的搜索途径。

由于对因果推理的要求不断增长，因而有必要将一些规则合并为一个浅推理规则。反驳归结法可以用于证明一条简单规则是否的确是多条规则的结论。该简单规则就是归结所要证明的定理。

作为一个例子，假设我们要证明规则（1）是规则（2）～（5）的逻辑结论。利用以下的命题定义，我们可将上述规则写成：

B = 电池是好的　　C = 汽车能跑
E = 有电　　F = 火花塞能打火
G = 有汽油　　R = 发动机能工作
S = 火花塞是好的　　T = 有好的轮胎

(1) $B \wedge S \wedge G \wedge T \rightarrow C$
(2) $B \rightarrow E$
(3) $E \wedge S \rightarrow F$
(4) $F \wedge G \rightarrow R$
(5) $R \wedge T \rightarrow C$

使用归结反驳的第一步是否定结论，即否定目标规则：

(1') ~(B ∧ S ∧ G ∧ T → C) = ~[~(B ∧ S ∧ G ∧ T) ∨ C]
= ~[~B ∨ ~S ∨ ~G ∨ ~T ∨ C]

把其他每一条规则利用：

p → q ≡~p ∨ q and ~(p ∧ q) ≡ ~p ∨ ~q

两个等式化为析取式，从而得到以下规则（2）～（5）的新形式：

(2') ~B ∨ E
(3') ~(E ∧ S) ∨ F = ~E ∨ ~S ∨ F
(4') ~(F ∧ G) ∨ R = ~F ∨ ~G ∨ R
(5') ~(R ∧ T) ∨ C = ~R ∨ ~T ∨ C

正如我们在前面小节所述的，一种表示（1′）～（5′）式的连续归结的简便方法是使用归结反驳树，如图 3.19 所示，从树的头部开始，用结点表示子句，它可被归结为其下的归结项。例如：

~B ∨ E and ~E ∨ ~S ∨ F

经归结后可推出：

~B ∨ ~S ∨ F

它再与

~F ∨ ~G ∨ R

一起归结可推出：

~B ∨ ~S ∨ ~G ∨ R

等等。为了画图简便，省略了最后归结式的一项一项的归结。

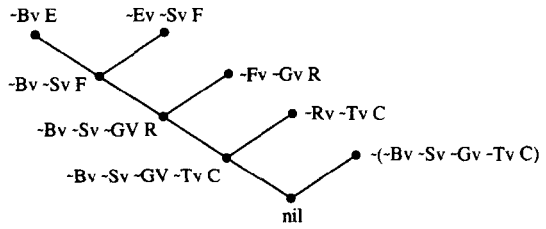


图 3.19 汽车例子的归结反驳树

由于树的根为空，这个归结是一个矛盾。根据反驳，由于此结论的否定导致了矛盾，因而原结论

B ∧ S ∧ G ∧ T → C

是一条定理。所以（1）确实可由（2）～（5）逻辑推出。

3.13 归结与一阶谓词逻辑

归结法也可与一阶谓词逻辑一起使用。事实上，这是 PROLOG 的主要推理机制。但在归结之前，必须把 wff 表示成子句的形式。考察以下的例子：

有些程序员讨厌所有错误
程序员都不会憎恶成功
∴错误都不是成功

定义以下谓词：

P(x) =x是程序员
F(x) =x是错误
S(x) =x是成功
H(x,y) = x憎恶y

前提以及结论的否定可表示为：

```
(1) (∃ x) [P(x) ∧ (∀ y) (F(y) → H(x,y))]
(2) (∀ x) [P(x) → (∀ y) (S(y) → ~H(x,y))]
(3) ~(∀ y) (F(y) → ~S(y))
```

这里，为了准备归结，已将结论表示为否定的形式。

转化为子句形式

以下9个步骤是一个将一阶谓词的wff转化为子句形式的算法。我们以上例中的wff（1）式为例来说明这一过程。

1. 利用等式

```
p → q ≡ ~p ∨ q
```

消去条件符“→”，wff变为：

```
(∃ x) [P(x) ∧ (∀ y) (~F(y) ∨ H(x,y))]
```

2. 在所有可能的地方，将否定符号消去或将否定的辖域减小到一个原子公式。使用附录A中的等式，如：

```
~~p ≡ p
~(p ∧ q) ≡ ~p ∨ ~q
~(∃ x) P(x) ≡ (∀ x) ~P(x)
~(∀ x) P(x) ≡ (∃ x) ~P(x)
```

3. 将wff中的变量标准化，使得每一量词的约束变量（哑变量）有惟一的名称。注意量词的变量名是哑元。即，

```
(∀ x) P(x) ≡ (∀ y) P(y) ≡ (∀ z) P(z)
```

因而

```
(∃ x) ~P(x) ∨ (∀ x) P(x)
```

的标准形式是

```
(∃ x) ~P(x) ∨ (∀ y) P(y)
```

4. 用**Skolem函数**（Skolem function）将所有的存在量词消去，Skolem函数是以挪威哲学家Thoralf Skolem的名字命名的。考虑

```
(∃ x) L(x)
```

其中L（x）为一谓词，当x<0时为真。这个wff可用

```
L(a)
```

代替，其中a是一个能使L（a）为真的常数，如－1，称a为Skolem常数，它是Skolem函数的一个特例。对于存在量词前有全称量词的情形，比如：

```
(∀ x) (∃ y) L(x,y)
```

当整数x小于整数y时，L（x，y）为真。该wff表示对于任意整数x总有整数y大于x。注意，该式并未指出对于给定的x应怎样计算y。假设存在产生y>x的函数f（x），那么以上的wff可Skolem化为：

```
(∀ x) L(x,f(x))
```

在全称量词辖域内的存在量词变量的Skolem函数是一个其左边所有全称量词的函数。例如，

```
(∃ u) (∀ v) (∀ w) (∃ x) (∀ y) (∃ z) P(u,v,w,x,y,z)
```

可Skolem化为：

```
(∀ v) (∀ w) (∀ y) P(a,v,w,f(v,w),y,g(v,w,y))
```

其中a是某个常数，第二个Skolem函数g必须与第一个函数f不同。于是例中的wff可化为：

```
P(a) ∧ (∀ y) (~F(y) ∨ H(a,y))
```

5. 将 wff 化为量词序列 Q 后跟有**母式**（matrix）M 的**前束式**（prenex form）。通常，量词可以是∀或∃。不过在本例中，由于步骤 4 已将所有的存在量词消去，因而 Q 只能是∀。同样，因为每一∀都有其哑元变量，所有的∀都可以移到 wff 的左边，所以每一∀的辖域均为整个 wff。

本例可化为：

```
(∀ y) [P(a) ∧ (~F(y) ∨ H(a,y)]
```

其中母式是方括号中部分。

6. 将母式化为合取范式，它是子句的合取式，每个子句是析取式。本例已经为合取范式。其中一个子句是 P（a），另一个是（－F（y）∨H（a，y））。如有必要的话，在将母式化为合取范式时，可使用以下的分配律：

```
p ∨ (q ∧ r) ≡ (p ∨ q) ∧ (p ∨ r)
```

7. 由于此时 wff 中的所有变量都是约束的，全称量词不必要，故可以消去。wff 就为母式。本例的 wff 化为：

```
P(a) ∧ (~F(y) ∨ H(a,y))
```

8. 消去符号∧，将 wff 写成子句的集合。本例就变为：

```
{ P(a), ~F(y) ∨ H(a,y) }
```

一般将其写为无大括号的子句形式：

```
P(a)
~F(y) ∨ H(a,y)
```

9. 如有必要的话，重命名子句中的变量，使得一个变量名只出现在一个子句中。比如以下的子句：

```
P(x) ∧ Q(x) ∨ L(x,y)
~P(x) ∨ Q(y)
~Q(z) ∨ L(z,y)
```

可被重命名为：

```
P(x1) ∧ Q(x1) ∨ L(x1,y1)
~P(x2) ∨ Q(y2)
~Q(z) ∨ L(z,y3)
```

如果我们按上述步骤对例中的第二个前提和否定结论进行变换，最终可得以下子句：

```
(1a) P(a)
(1b) ~F(y) ∨ H(a,y)
(2a) ~P(x) ∨ ~S(y) ∨ ~H(x,y)
(3a) F(b)
(3b) S(b)
```

序号表示了原来前提和否定结论的序号。即，前提（1）和否定结论（3）被变换为分别带有（a），（b）后缀的两个子句而前提（2）变换为单个子句（2a）。

合一与规则

将 wffs 转化为子句的形式以后，常常需要为变量寻找合适的**代换实例**（substitution instance）。比如子句

```
~F(y) ∨ H(a,y)
F(b)
```

不能对谓词 F 作归结，除非找到了 F 中的参量匹配。这个寻找合适的代换从而实现参量匹配的过程就称为**合一**（unification）。

合一是区别专家系统与简单判定树的特征之一。没有合一，规则的条件元素只能匹配常数，这意味着必须为每一可能存在的事实写一条专门的规则。例如，假定希望在探测器指示有烟雾时发出火警警报，如果有 N 个探测器，那么就需要为每一探测器设置一个如下的规则：

IF 探测器1指示有烟雾 THEN 发出火警警报1
IF 探测器2指示有烟雾 THEN 发出火警警报2
…
IF 探测器N指示有烟雾 THEN 发出火警警报N

然而，利用合一，用一个称为“? N”的变元作为探测器标志，就只需如下一条规则即可：

IF 探测器? N指示有烟雾，
THEN 发出火警警报? N

若两个互补的文字是合一的，则可以通过归结消去。对于前两个子句，用b代换y，得

```
~F(b) ∨ H(a,b)
F(b)
```

谓词F因合一可消去，从而得到H（a，b）。

用与变量不同的项同时对变量进行替换称为代换，这些项可以是常数、变量或函数。这种对变量的代换可用集合表示为：

$\{ t_1/v_1, t_2/v_2 \ldots t_N/v_N \}$

如果θ是这样一个集合而A是一参量，那么Aθ定义为A的代换实例，例如，若

```
θ = { a/x, f(y)/y, x/z }
A = P(x) ∨ Q(y) ∨ R(z)
```

则

```
Aθ = P(a) ∨ Q(f(y)) ∨ R(x)
```

注意代换是同时进行的，因而我们得到的是R（x）而不是R（a）。

假定有C_1，C_2两个子句，定义为：

```
C1 = ~P(x)
C2 = P(f(x))
```

对P的一种可能的合一为：

```
C1' = C1 { f(x)/x } = ~P(f(x))
```

对P的另一种可能的合一是用常数a进行两个代换：

```
C2'' = C2 { f(a)/x } = ~P(f(a))
C2'  = C2 { a/x }    = P(f(a))
```

注意P（f（x））比P（f（a））更一般，因为用任意常数代换x就可得到P（f（x））的无穷个实例，像C_1这样的子句称为**最一般子句**（most general clause）。

通常，称代换θ为集合$\{A_1, A_2, \cdots, A_N\}$的合一代换（unifier），当且仅当$A_1\theta = A_2\theta = \cdots A_N\theta$成立。存在合一代换的集合称为**可合一的**（unifiable），如果其他所有的合一代换都是某合一代换的实例，则称该合一代换为**最一般合一代换**（most general unifier，简称mgu）。这可以形式化地表述为：θ是最一般合一代换，当且仅当对集合的任一合一代换α，存在一代换β使得

$\alpha = \theta\beta$

其中θβ是一个复合代换，即先用θ进行代换，再用β进行代换。**合一算法**（unification algorithm）是一种为有限的可合一的参量集合寻找最一般合一代换的方法。

对于我们所举的例子中的子句（1a）到（3b），代换和合一的结果可用图3.20的归结反驳树来表示。由于根结点为空，结论的否定为假，因而结论“错误都不是成功”是正确的。

目前所使用的例子都很简单，因而可以直接归结，尽管对于人来说显得相当乏味。然而在其他很多的情况下，归结的过程可能是一条死路，因而必须回溯以尝试归结其他的子句。虽然归结极为有用而且是PROLOG的基础，但对某些问题它可能不那么有效。归结法的一个问题是它内部没有有效的搜索策略，因而程序员必须提供一些启发，如PROLOG的截断语句，以实现有效的搜索。

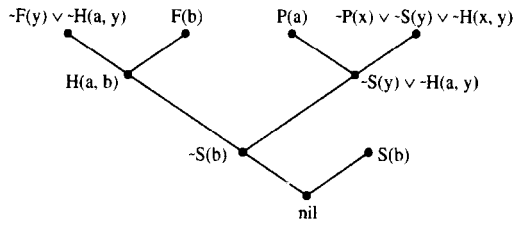


图 3.20 证明错误都不是成功的归结反驳树

目前已提出了一些改进的归结法，如**单项优先**（unit preference）、**输入归结**（input resolution）、**线性归结**（linear resolution）和**支撑集**（set of support）等。归结的主要优点在于它是一种在许多情况下均可满足需要的简单而有效的方法。这使得建立一个像 PROLOG 的机械系统比建立一个试图实现许多不同推理规则的系统要简单得多。归结的另一个好处在于如果成功的话，归结通过显示导致空的步骤而自动提供了证明。

3.14 正向链和反向链

一组将问题和其解答联系起来的多重推理称为一条**链**（chain）。一条由问题开始搜索并得到其解答的链称为正向链。另一描述正向链的方法是：由事实推出基于事实的结论。一条由假设回推到支持该假设的事实的链称为反向链。反向链的另一种表述是通过满足某个目标的子目标来完成该目标。由此可见，用于描述正向链和反向链的术语由所讨论的问题决定。

链可以很方便地用推理来表示。例如，假设有以下的假言推理类型的规则：

```
p → q
p
∴ q
```

它形成了一条推理链，比如：

```
elephant(x) → mammal(x)
mammal(x) → animal(x)
```

上述规则可用于由 Clyde 是大象推出 Clyde 是动物的正向因果推理链中。推理链如图 3.21 所示，注意这个图也可以表示反向链。

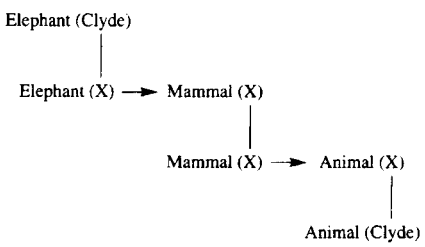


图 3.21 因果正向链

在图 3.21 中，因果链用一系列的竖线表示，这些竖线将前一规则的后件与后一规则的前件联接起来。竖线也表示了变量到事实的合一。例如，在使用关于大象的规则前，必须先把谓词“elephant (x)”中的变量 x 与事实“elephant (Clyde)”合一。事实上，因果链是一个蕴含式与合一的序列，如图 3.22 所示。

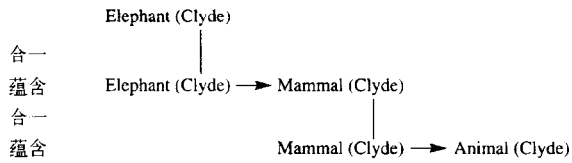


图 3.22 显式因果链

反向链是一个相反的过程。假定我们想要证明“动物 (Clyde)”这一假设。反向链的主要问题是要找到一条将证据和假设连接起来的链。在反向链中，“大象 (Clyde)”这一事实被称为**证据**（evidence），证据是用于支持假设的，正如在法庭上证据用于证明被告有罪一样。

一个正向链和反向链的简单例子，假设你正在开车，忽然看到一部闪着警灯的警车。使用正向链，你也许会推断警察想要你或其他人停下来。即，上述的事实支持两个可能的结论。若警车恰好在你身后停下来或警察确你挥手，更进一步的推论是警察找你的可能性大于其他人。以此为有效假设，你会使用反向链来推理这是为什么。

一些可能的中间假设是：因为乱丢杂物、超速、设备故障和开着一部偷来的车。于是你就会检查是否有支持这些中间假设的证据：是因为你扔出窗外的啤酒瓶吗？是在时速限制为30英里的地方以100英里的时速行驶吗？是破碎的车尾灯或是执照牌表明你开的车是偷来的？在此情况下，每一个证据支持一个中间假设，因而它们都是成立的。任意或所有的这些中间假设都是证明警察找你这一假设的可能理由。

为了更具体地了解正向链和反向链，将它们表示成一条穿过问题空间的道路，其中在反向链中的中间状态表示中间假设，在正向链中的中间状态表示中间结论。表3.14中总结了一些正向链和反向链的共同特点。注意表中的特点只作为一个指导。在正向链系统中进行判断和在反向链系统中进行规划当然是可能的，特别是在反向链中由于系统可以很方便地解释究竟要达到什么目的，因而解释是很容易实现的。在正向链中，解释就不那么容易实现了，因为在子目标被发现前并不能明确地知道子目标。

表3.14 正向链和反向链的一些特征

正向链	反向链
规划，监视，控制	诊断
从当前到未来	从当前到过去
前件到后件	后件到前件
数据驱动，自底向上推理	目标驱动，自顶向下推理
向前推理以找到由事实可推出的解决方案	向后推理以找到支持假设的事实
便于广度优先搜索	便于深度优先搜索
前件决定搜索	后件决定搜索
不便于解释	便于解释

图3.23说明了在基于规则的系统中正向链的基本概念。规则由满足前件或左部（left-hand-side，简称LHS）的事实触发。例如，要激活规则R_1必须满足事实B和C。然而，由于只有C存在，因而R_1未被激活。规则R_2由事实C和D激活。由于这两者均存在，因而R_2产生出中间事实H。其他要满足的规则有R_3、R_4、R_7、R_8和R_9。规则R_8和R_9的执行便产生了正向链的结论。这些结论可以是其他事实、输出等。

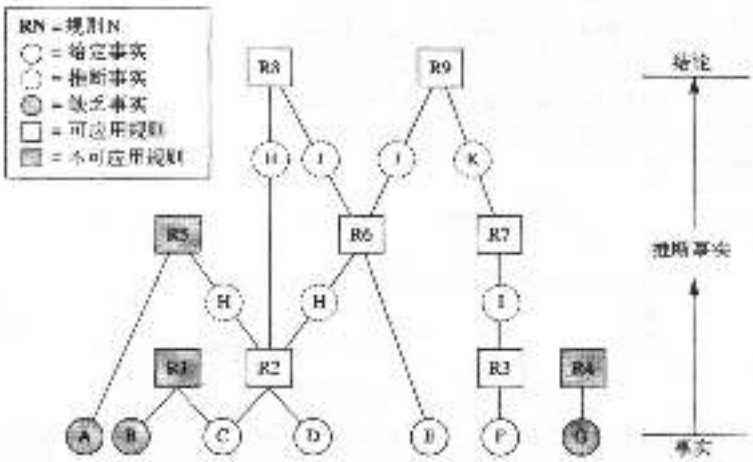


图3.23 正向链

正向链又称为**自底向上推理**（bottom-up reasoning）。因为它从低层的证据、事实出发，由推理得到顶层的基于事实的结论。专家系统中的自底向上推理，类似于第 1 章所讨论的自底向上传统程序设计。事实是基于知识系统中的基本单元，因为它不可以再细分为任何有意义的更小单元。例如，事实“duck”作为名词和动词有确定的意思。可是，如果进一步细分的话，结果是字母 d、u、c、k，它们都没有特定的意思。在传统的程序中，有意义的基本单元是数据。

按照习惯，由较低层部分组成的较高层部分放在上面。因此，从较高层部分，比如假设开始，推理得到支持这些假设的较低层事实，就称为**自顶向下推理**（top-down reasoning），或反向链。图 3.24 说明了反向链的概念。为了证明或否定假设 H，至少要证明中间假设 H_1，H_2，H_3 中的一个。注意到将这个图画成一个与—或树的形式，它说明了像 H_2 这些结点，位于其下层的所有假设都必须支持 H_2。对于其他结点，例如顶层的假设 H，只需有一个较低层次的假设的支持。在反向链中，系统为了证明或否定一个假设，通常会从用户那里引征证据。这与正向链中已预先知道所有相关的事实相比，是不同的。

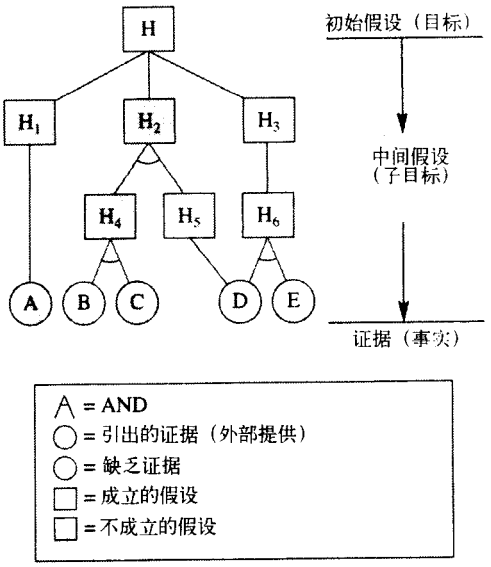


图 3.24 反向链

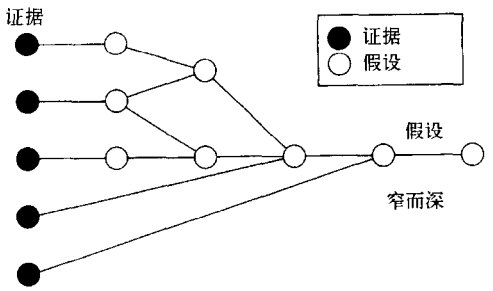
如果对照图 3.24，我们就会发现，决策网的组织结构非常适于反向链。顶层的假设是各种不同种类的草莓，例如花莓、黑莓、酒莓和红莓。支持这些假设的证据位于较低层次。很容易写出区别这些草莓的规则。例如：

```
IF 叶子是单叶 THEN 是花莓
```

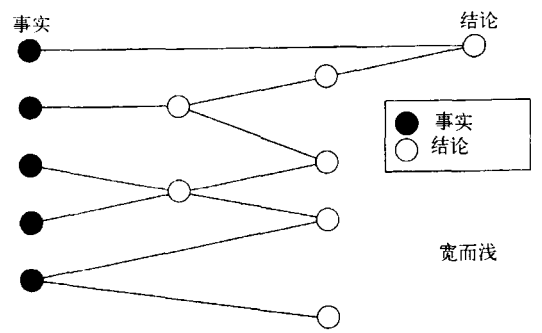
引征证据的一个重要手段是提出正确的问题。正确的问题有助于提高得出正确回答的效率。这方面的一个明确要求是，专家系统应该只提出与其试图证明的假设有关的问题。如果系统提出成百上千的问题，那么就必须付出相应多的时间和金钱以便得到解答问题的证据。另外，收集某一类型的证据，例如医学检验的结果，也可能会令病人感觉不舒服，甚至有危险（事实上，很难想像会有舒服的医学检验）。

理想上专家系统应该容许用户自愿提出一些系统没有问及的证据。这可以加快反向链的进程，并使系统对用户而言更方便。自愿提出的证据可能会让系统跳过因果链中的某些连接，或者寻求一个完全不同的新方法。这种做法的缺点是，由于系统可能不是按照原来的链接关系一步一步地进行推理，因此专家系统的程序设计会更复杂。

图 3.25 给出了一个很好的正向链和反向链的应用例子。为简单起见，这些图表示成树而不是通常的网络。当树宽且不太深的时候，应用正向链是比较好的，这是因为正向链便于实现宽度优先搜索。也就是说，正向链适合于一层一层地搜索结论。与之相反，反向链便于实现深度优先搜索。一棵适合深度优先搜索的树是窄而深的。



(a) 反向链的一个很好的应用



(b) 正向链的一个很好的应用

图 3.25 正向链和反向链

注意，规则的结构决定了搜索解的方法。也就是说，什么规则被激活依赖于这种规则所要匹配的模式。LHS 上的模式决定了规则是否可以被事实激活。RHS 上的动作决定了被断定和删除的事实，并因此而影响到其他规则。除了使用的是假设而不是规则之外，在反向链中也有类似的情况。当然，一个中间假设也可认为是一条匹配其后件而不匹配其前件的规则。

作为一个非常简单的例子，让我们考虑下面简单的 IF…THEN 型规则。

```
IF A THEN B
IF B THEN C
IF C THEN D
```

如果给定事实 A，并且推理机是用事实来匹配前件，那么中间事实 B 和 C 将被断定并得到结论 D。这个过程相当于正向链。反过来，如果事实 D（实际上是假设）被断定并且推理机是用事实来匹配后件，那么结果将相当于反向链。在诸如 PROLOG 这样的反向链系统中，其反向链机制包括了一系列的功能，例如自动回溯，以便实现反向链。

通过重新设计规则，反向链可以用正向链来实现，反之亦然。例如，前面的正向链规则可以被重写为：

```
IF D THEN C
IF C THEN B
IF B THEN A
```

为了满足假设 D，现在 C 和 B 被看作是必须满足的子目标或中间假设。证据 A 是标志产生子目

标结束的事实。如果存在事实 A，那么在这个反向推理链中，D 可证实并被认为是真。如果不存在 A，那么假设 D 便无法证实而被认为是假。

该方法的困难之一是效率。反向链系统便于深度优先搜索，而正向链系统便于宽度优先搜索。尽管你可以在正向链系统中写一个反向链的应用程序，反之亦然，但在搜索结论的过程中系统的效率将会不高。第二个困难是概念上的困难。从专家那里得到的知识必须修正以适应推理机要求。例如，正向链推理机匹配规则的前件，而反向链推理机匹配规则的后件。也就是说，如果专家的知识本身符合反向链，为了把它放到一个正向链中，就必须完全改变其结构，反之亦然。

3.15 其他推理方法

其他的许多推理方法也经常用于专家系统中，尽管这些方法并非像演绎法那么通用，但它们还是非常有用的。

类比（Analogy）

除了演绎和归纳，另外一种很有用的推理方法是类比。类比推理的基本思想就是试图把旧情形作为新情形的指导。自然界中许多生物都善于在它们的生活中使用类比法。这一点很重要，因为在现实世界中每时每刻都有很多新的情形出现，与其每次都孤立地去处理一个新的情形，还不如把新情形和所熟悉的情形联系起来。类比推理同归纳是有联系的。归纳是在同样的情形下从具体到一般的推理，而类比却是试图从不同的情形进行推理。类比不能像演绎那样进行形式化的证明，但类比法作为一种启发式推理工具，有时可发挥较大的作用并且是法律论证及医疗诊断等推理的主要方法。

作为类比推理的一个例子，我们看一下疾病的诊断。当你身体不适去看医生的时候，医生会询问你信息并记录下症状，如果你的症状与其他人患 X 病的症状一模一样或者极其相似的话，医生就会通过类比推理推断你患有 X 症。类比是简单的推理。

注意，这种诊断不属于演绎推理，因为你同别人存在个体差异。仅仅因为另一个患有同样病的人显示出某些症状，并不能说你就会有那些症状。相反，医生会假设你的症状使你与具有同样症状且病况已知的人相类似。初步的诊断只是一个假设，它可被医学检验证实或推翻。但首先提出初始假设是十分重要的，因为这样可以极大地缩少要考察的可能性的个数。若没有这种初始假设，那么就只好对每一种可能性都去检查，这样成本又高，又花时间。

下面举一个应用类比法推理的例子。假设有两个人在玩一个叫做“15”的游戏。他们轮流从数字 1~9 中取数，同一个数不能使用两次，首先使所取数之和等于 15 的那方算赢。这个游戏乍一看似乎很花脑筋，但使用类比法可以很容易地解决。

考虑下面的一字棋棋盘，每个格中有一个数：

6	1	8
7	5	3
2	9	4

这是一个**魔方**（magic square），因为其中每一行、每一列、每条对角线上的和都是常量 15。该魔方可与“15”游戏进行类比，在玩“15”游戏时如果参照这个魔方，并将其致胜的策略用于“15”游戏中，那么就很容易玩了。

这个特殊的魔方又称为**3 阶标准方阵**（standard square of order 3）。阶数是由方阵的行数或列数决定的。只有一种 3 阶方阵，其他的魔方可以通过对折和旋转标准方阵得到。另外，只要在每个格中加上相同的数，仍可得到一个魔方。知道了这一点后，我们又可以推出“18”游戏的致胜策略。其中，待选数是从下面集合选出的：

```
{2, 3, 4, 5, 6, 7, 8, 9, 10}
```

类似地，在“21”游戏中，待选数的集合是：

```
{3, 4, 5, 6, 7, 8, 9, 10, 11}
```

我们现在可以归纳出“15+3N”游戏的致胜策略，其中N是自然数1，2，3，…。这是通过把它们与一个一字棋棋盘类比，而它又与具有以下值的魔方类比：

```
{1+N, 2+N, 3+N, 4+N, 5+N, 6+N, 7+N, 8+N, 9+N}
```

类比“3阶方阵”用于走三步的游戏，通过归纳，我们可以找到更高阶的魔方用于走更多步的游戏。例如，如下的4阶魔方：

16	3	2	13
5	10	11	8
9	6	7	12
4	15	14	1

通过对照一字棋，可以助你赢得四步“34”游戏。与3阶标准方阵只有一种不同，四阶标准方阵共有880种。这使得我们可以玩更多的游戏。

类比推理是常识推理的重要组成部分，这对计算机（或者对小孩子）来讲是非常困难的。类比的另一个应用是用于学习。

生成与验证

另一种推理方法是人工智能中的一种经典方法——**生成与验证**（Generate-and-Test），这种方法首先生成一种可能解，然后进行验证，看所提出的解是否满足所有条件。若满足，则退出，否则再生成一个新解，再去验证，如此下去。这种方法在建于1965年的第一个专家系统DENDRAL中被应用，用以帮助识别一个有机分子的构造。数据从质谱仪所提供的未知样本中获取，并被输入到DENDRAL中，以生成所有可能产生该未知光谱图的分子结构；DENDRAL接下来对那些最有可能的结构，通过模拟出它们的质谱图，并与原始数据比较，来进行验证。另外一个使用生成与验证的程序是AM（人工数学家 Artificial Mathematician），它用来推出新的数学概念。

为了减少可能解的巨大数目，生成与验证常常与规划程序一起使用以限制生成的可能解，这种改进后的方法叫**“规划—生成—验证”**（plan-generate-test），并在很多系统中应用以提高效率。例如，当病人被确诊后，医疗诊断专家系统MYCIN就具有规划治疗处方的功能。**规划**（plan）实际上就是基于已有证据的支持，找出那些联结着“问题”和“解”（目标）的规则或推理链。实现规划的最有效方法是同时从事实正向推理和从目标反向推理。

MYCIN的规划程序首先将一些对病人来讲作用明显的药列成具有优先级的清单。为了减少药之间不必要的相互作用，即使病人患有几种不同的疾病，最好开给病人的药尽可能少些。生成程序从规划程序开列的优先级清单中生成一些子清单，如果可能的话，子清单只有一两种药就够了。然后，在做出一个决定给病人开药之前，这些子清单将被测试对疾病的效果、病人的过敏史以及其他一些因素等等。

生成与验证也可看作是规则的基本推理模式。如果规则的条件满足，它就生成一些动作，比如新事实。然后推理机就测试这些事实，看它们是否满足知识库中规则的条件，那些被满足的规则被放入议程中，继而具有最高优先级的规则再生成它的动作，然后再测试，如此下去。这样，生成与验证就形成了一条可能会得出正确解的推理链。

溯因

溯因（abduction）推理是一种在诊断系统中常用的方法。溯因法的图解有点像假言推理。但实际上两者相差甚远，如表 3.15 所示。

表 3.15 溯因和假言推理的比较

溯　因	假言推理
$p \rightarrow q$	$p \rightarrow q$
q	p
$\therefore p$	$\therefore q$

溯因是我们在第 3.6 节中谈到的逆谬论的另一种说法。尽管溯因不是一个有效的演绎辩论，但它是一个很有用的推理方法，且已被用于专家系统中。与同样不是有效逻辑辩论的类比一样，溯因可作为推理中一个有用的启发式规则。也就是说，当演绎法在我们的推理中不适用时，溯因法就可能会显得有用了，但这并不保证说一定能行。类比、生成与验证、溯因都不是演绎的，也都不保证一定能行。从正确的假设，这些方法不一定能得出正确的结论。但不管怎样，这些方法都能够通过生成合理的假设，来帮助缩小搜索的范围。这些合理的假设还可被用于演绎。

溯因有时候是指根据观察到的现象推出最佳的解释。作为一个例子，考虑如下规则：

IF x 是大象，THEN x 是动物
IF x 是动物，THEN x 是哺乳动物

如果我们知道 Clyde 是一只哺乳动物的话，我们能否得出 Clyde 是一头大象呢？

回答这个问题，要看是在现实世界中还是在专家系统中来进行讨论。在现实世界中我们没有任何确信度做出这样的结论。Clyde 可以是一只狗、一只猫、一头牛以及除大象以外的其他任何哺乳动物。事实上，有如此多的动物，如果没有关于 Clyde 的任何信息，那么，Clyde 为大象的概率会非常低。

然而在一个只有前述规则的专家系统中，通过溯因我们可以百分之百地说，如果 Clyde 是一只哺乳动物的话，那么 Clyde 是一头大象。这个推理基于一个**封闭世界假设**（closed-world-assumption），在该假设中，我们假定在专家系统的封闭世界之外不存在任何东西。凡是在封闭世界不能证明的事物就是假的，在封闭世界假设下，所有的可能性都是已知的。因为专家系统中只有上述两条规则，并且只有大象才能是哺乳动物，所以如果 Clyde 是哺乳动物的话，那么它百分之百是一头大象了。

假设我们加入以下第三条规则：

IF x 是狗 THEN x 是动物

我们仍然可以在专家系统的封闭世界假设下去考虑问题，然而此时得出 Clyde 是大象的确信度就没有百分之百。我们只能说 Clyde 要么是大象要么是狗。

如果要确定是两者中的哪一个，就必须要有更多的信息。例如，有另外一条规则：

IF x 是狗 THEN x 会吠

并且有证据 Clyde 会吠，所有规则经修改后如下：

(1) IF x 是动物，THEN x 是哺乳动物
(2) IF x 会吠，THEN x 是动物
(3) IF x 是狗，THEN x 会吠
(4) IF x 是大象，THEN x 是动物

现在我们就可以用（1）、（2）、（3）组成一条溯因推理的反向链，并说明 Clyde 一定是一只狗。

这里的溯因反向链同习惯上的反向链意义是不同的，反向链意味着我们试图找出证据来证明假设。反向链可以用来证明 Clyde 是一只哺乳动物。当然，在我们的小型系统中已没有其他的可能性。然而，在其他分类问题中，可能会加入爬虫、鸟类等动物。

如果已知 Clyde 是一只哺乳动物，那么可以用溯因法来确定 Clyde 是一头大象还是一只狗。如果已

知 Clyde 是一头大象，而我们想知道它是否是哺乳动物，那么可以用正向推理。可见，选用什么推理方法，取决于你想推出什么。因为正向链就是演绎，所以只有其结论才保证是有效的。表 3.16 概括了三种推理技术的目的。

表 3.16 正向链、反向链、溯因法的目的

推理技术	开始	目的
正向链	事实	基于事实的结论
反向链	未确定的结论	支持结论的事实
溯因法	已确定的结论	可能引致结论的事实

许多人工智能和专家系统在解决诊断性问题时采用了基于框架的溯因法。在这些系统中，知识库中包含一些疾病与症状之间的**因果联系**（causal association）。在推理时，使用生成与验证法来生成假设疾病。

非单调推理

一般来说，若在一个逻辑系统中增加一条公理，则在系统中应能推导出更多的定理来，这是因为有更多的公理可供推理。这种增加了公理后，系统中定理便随着增加的性质称为单调性（monotonicity），而相应的系统，比如演绎逻辑就称为**单调系统**（monotonic system）。

然而，假如新引入的公理与原有的公理部分或完全矛盾的话，问题就会出现了。在这种情形下，原来已证的定理可能就不成立了。因此，在**非单调系统**（nonmonotonic system）中，公理增加并不意味着定理会随之增加。

非单调性的概念在专家系统中有着非常重要的应用。随着一些新事实的产生——类似于新证明的定理——单调专家系统就会不断建立新的事实。如果有一个或多个事实变为假，由于单调系统不能对公理和定理的真值改变进行处理，一个大的问题就会出现。作为一个非常简单的例子，假设有一个断定时间的事实，随着时间一秒一秒地变化，原有的事实可能不再成立，单调系统是不能处理这种情况的。另外一个例子，假设有一个事实是飞机识别系统发现目标是敌机。后来，又有新的证据表明那个目标原来是友方的。在单调系统中，初始目标是敌机是不能改变的，而非单调系统则允许撤销事实。

作为另一个应用，假设你准备写一个专家系统的解释机。它可以使用户返回到先前的某一个推理点，探索“如果……会怎样”问题的其他推理路径。那么在这一个推理点之后所做出的推理就必须从系统中撤销。除了事实外，基于非单调性，后来引入的规则也应该从系统中去掉，并必须被移回知识库中。在像 OPS5 这样的系统中，还会产生更复杂的问题，因为 OPS5 会在运行期间，在规则的右部自动创建新的规则。基于非单调性，在推理点之后所生成的任何规则也都必须去掉。这样记录下所有的推理会消耗大量的内存，也大大降低了系统的速度。

为了支持非单调性，有必要为每个事实或规则附上一个依据以解释确信它的理由。每当进行一次非单调判定时，系统就检查每个事实或规则的依据是否仍然成立，并尽可能地恢复依据成立的已去掉规则和已撤销事实。

为事实提供依据的问题最早是在**帧问题**（frame problem）中提出来的，这里的帧与第 2 章讨论的框架（frame）是两个不同的概念。帧问题是一个形象的词汇，它是根据问题——在电影的“帧”中如何标识哪些东西已改变或没改变而命名的。放电影的过程其实就是连续地将一幅幅静态的画面放到屏幕上，每幅静态的画面叫做“帧”。当我们每秒中能放 24 或更多的帧时，人的眼睛就分辨不出帧之间的变化，从而就得到了电影效果。在人工智能中讨论的“帧问题”是要识别一段时间内环境发生的变化。作为一个例子，考虑猴子与香蕉问题，假设猴子只有站到一只红色的箱子上才能够得着香蕉，因此它要做的动作就是“把红色的箱子推到香蕉下面”。现在帧问题就是我们怎么知道在这个“推”动作之后箱子还是红色的呢？推箱子不应该使环境发生改变。但是，其他动作如“把箱子涂成蓝色”将改

变环境。在一些专家系统工具中把“环境”称为**问题世界**（world），它由一组相关的事实组成。一个专家系统可以同时记录下多个问题世界的假设推理。维护一个系统正确性的问题称为**“正确性维护”**（truth maintenance）。正确性维护，也称为**“基于假设的正确性维护”**（assumption-based truth maintenance），它对于撤销无依据事实以保证问题世界的纯粹性是必不可少的。

作为非单调性推理的一个简单例子，让我们来看一个经典的 Tweety 鸟问题（Tweety the bird）。在缺乏其他信息的情况下，我们假设既然 Tweety 是一只鸟，那么 Tweety 会飞。这是**默认推理**（default reasoning）的一个例子，它类似于框架槽中的默认值。默认推理可以看作是一条关于规则推理的规则，即**元规则**（metarule），比如：

```
IF X 不确定,并且
   没有证据反驳 X
THEN 暂且认为 Y 成立
```

我们将在下一节详细讨论元规则。

对于我们的 Tweety 鸟问题，元规则具有更明确的形式：

```
X 为规则"所有鸟都会飞"以及事实"Tweety 是一只鸟"
Y 为结论"Tweety 会飞"
```

这可以表示为知识库中的一条规则，即

```
IF X 是一只鸟,THEN X 会飞
```

同时在内存工作区中有这样一条事实：

```
Tweety 是一只鸟
```

这条事实与已有的规则合起来可以推出 Tweety 会飞。

现在问题来了。假设在内存工作区中加入了一个事实说 Tweety 是一只企鹅。我们知道企鹅是不能飞的，因此推理 Tweety 会飞是不正确的。当然，在系统的知识库中必须有一条规则来描述该知识，或者系统忽略掉这一事实。

为了维护系统的正确性，那些不正确的推理都应该删掉。然而如果有其他推理基于这些错误推理的话，这还不够。也就是说，其他规则有可能把这些错误的推理作为证据进行了另一些推理。这是“正确性维护”的一个问题。推理 Tweety 会飞是基于默认推理的一个**合情推理**（plausible inference，意思是“不是不可能”），这将在第 4 章中进一步讨论。

一种允许有非单调推理的方法是定义一个句子算符 M，M 的非形式化定义为“是相容的”。例如：

```
(∀ x) [Bird(x) ∧ M(Can_fly(x))→ Can_fly(x)]
```

可以陈述为：“对任意 x，若 x 是一只鸟，且这与鸟会飞是相容的，则 x 会飞”。一个更加通俗的说法是“大多数鸟会飞”。术语“是相容的”意味着与其他知识没有矛盾。然而，这种解释已遭到批评，因为它实际上是说只有那些被推出不能飞的鸟才被认为是不会飞的鸟。这是一个**自觉推理**（autoepistemic reasoning），其字面意思是根据你自己的知识进行推理。默认推理和自觉推理都可以用于**常识推理**（commonsense reasoning）。常识推理对人类来说十分简单，但对计算机则非常困难。

自觉推理是指根据不同于通常知识的自有知识作出推理。一般来讲，人们可以在这方面做得很好，因为人们知道自己的知识有限。例如，假设一个完全陌生的人来到你面前自称是你的配偶，你只要不是患了失忆症的话你会立刻知道这是假的，因为你没有关于陌生人的任何知识。自觉推理的一般元规则是：

```
IF 我没有关于 X 的知识
THEN X 是假的
```

注意，自觉推理是如何依赖于封闭世界假设的。所有未知的事实都认为是假的。在自觉推理中，

封闭世界就是你所具有的全部知识。

自觉推理和默认推理都是非单调的。但是两者不同，默认推理之所以非单调是因为它是可撤销的(defeasible)，术语“可撤销的”意思是所有的推理都是暂时的，当增加新的信息后这些推理可能要被撤销。然而纯粹的自觉推理是不可撤销的，因为封闭世界假设宣称所有的正确知识都是已知的。譬如说，由于一个结了婚的人肯定知道他的配偶是谁（除非想忘记），因而当一个完全陌生的人跑来说是他的配偶时，他是不会接受的。由于人们知道他们的记忆并不是百分之百的准确，因此他们并不依赖纯粹的自觉推理。当然，计算机不会存在这样的问题。

自觉推理之所以非单调是因为自觉语句的意义是**上下文相关的**（context-sensitive），术语上下文有关意思是语境不同，意义也将不同。作为一个上下文相关的简单例子，考虑单词“read”在以下两个句子中的发音：

```
I have read the book
I will read the book
```

这里“read”一词的发音是上下文有关的。

现在让我们来看一个系统，它包括下面两条公理：

```
(∀ x) [Bird(x) ∧ M(Can_fly(x))→ Can_fly(x)]
Bird(Tweety)
```

在这个逻辑系统中，Can _ fly（Tweety）是通过对 Tweety 和蕴含式中的 X 进行合一而得出的一条定理。

现在假定增加一条新的公理：Tweety 不会飞，则与先前推得的定理矛盾。

```
~Can_fly(Tweety)
```

由于现在 M（Can _ fly（Tweety））与新的公理不相容，所以 M 算符的运算结果就要改变。在这三条公理的新语境下，M 算符对 Can _ fly（Tweety）的结果不是 TRUE，因为它与新公理冲突。在新语境下，M 算法的返回值是 FALSE，于是式中的合取项也为 FALSE。因此就推不出 Can _ fly（Tweety）这条定理，也就没有冲突了。

通过规则实现上述推理的一种方式是：

```
IF x是一只鸟 且 x是一只典型的鸟
THEN x会飞
IF x是一只鸟 且 x是一只非典型的鸟
THEN x不会飞

Tweety是一只鸟
Tweety是一只非典型的鸟
```

注意这个系统不会得到无效的结论 Can _ fly（Tweety），而且完全杜绝了错误规则的激活。这比起使用一条规则和一条特殊公理~Can _ fly（Tweety）来说，是一个更加有效的正确性维护方法。现在我们得到了一个更一般的系统，这个系统能更容易地处理其他像鸵鸟一样不能飞的鸟，而不需要不断增加新的推理，这正是系统所追求的目标。

3.16 元知识

元—DENDRAL 程序使用归纳法来推导新的化学结构规则。它是从人类专家中获取分子结构规则以克服知识瓶颈问题的一个尝试。元—DENDRAL 程序在重新发现已知规则并推出新规则方面非常成功。

作为一个例子，以下的元规则取自 MYCIN 的知识获取程序 TEIRESIAS，MYCIN 是一个诊断血液感染和脑膜炎的专家系统。

```
元规则 2
IF
```

病人是感染体，并且
有些规则在它们的前提中提到了假单细胞菌属，并且
有些规则在它们的前提中提到了克雷白杆菌属
THEN
有证据表明(0.4)前者应先于后者使用

(在规则动作部分的数字 0.4 表示一种确定程度，这将在后面的章节讨论)

TEIRESIAS 从专家那里交互地获取知识。若 MYCIN 的诊断不正确，那么 TEIRESIAS 将带领专家沿不正确的推理链回溯至专家认为是不正确推理的起始点。在沿推理链回溯的同时，TEIRESIAS 和专家一起交互作用，以修改不正确的规则或获得新的规则。

有关新规则的知识并不是立即就存入 MYCIN 中。相反，TEIRESIAS 先要检查新的规则是否与类似的规则兼容。例如，若有一条新的规则描述人体如何受到感染，而其他已接受的规则中有一条件要素指明了进入体内的入口，则新的规则同样要有这一条件要素。若新的规则中并未表明进入的入口，那么 TEIRESIAS 将就此差异对用户提出质疑。TEIRESIAS 中有一个相似规则的**规则模型**（rule model），并试图使新规则适合其规则模型。换句话说，规则模型是 TEIRESIAS 关于其自身知识的知识。一个在人们生活中会出现的类似情形是，你去汽车商那儿买一部新车，而汽车商努力想要将一部有三个轮子的车卖给你。

TEIRESIAS 的元知识有两种类型。前面讲述过的元规则 2 是一个说明如何使用规则的控制策略。与之相反，元知识的规则模型则决定新规则是否具备了可进入知识库的恰当形式。在基于规则的专家系统中，将确定新规则的形式是否正确这一工作称为规则的**验证**（verification）。将确定一正确推理链是否导出正确答案这一工作称为**证实**（validation）。证实与验证密不可分，因此通常用缩写词 V&V 指代两者。在软件工程中关于上述术语的更通俗定义为：

验证："我在以正确的方法构造产品吗？"
证实："我在构造正确的产品吗？"

V&V 将在以后的章节中更为详细地讨论。

3.17 隐马尔可夫模型

作为元知识的一个现代例子，考虑机器人的路径规划。如果机器人不具备一个全球定位系统(GPS)，或者不够精确，例如通常商业系统只能确定在 3 米内的位置，那么必须使用其他的方法。一个好的方法是使用**马尔可夫决策过程**（Markov decision process，MDP）（Kaelbling 98）。也有其他方法使用经典的 A^* 算法、Kalman 过滤法和其他技术（Lakemeyer 03）。

在现实世界中总是存在着不确定性，而纯逻辑在存在不确定性时不大适用。在状态、参数和规划需求只存在部分信息或具有隐藏信息的情况下，MDP 更具有现实性。这种处理过程不仅仅适于物理路径规划，而且还适于其他规划，例如不完整环境下的石油勘探、运输后勤以及传感器可能出现故障和问题的工厂流程控制。当进行核电站过程控制时，这种过程控制显得尤为重要。

机器人探索另一个星球表面，例如火星，就是只有部分信息出现的最好例子。如果目的是到达某块岩石，但有部分路径不明时，机器人必须在约束条件内，例如能源供应限制，或者当走入深坑无法回到正确路径等限制下找到最优方法。

一个 MDP 可以定义为元组｛状态，动作，转换，回报｝。更形式地我们可以写作 MDP≡｛S，A，T，R｝，其中，

S 是环境状态集合；
A 是动作集合；

T：S x A≡Π（s）；其中Π称为**状态转换函数**（state-transition function），决定某个动作后的下一个状态。"x" 是笛卡儿积符号。Π是所有可能状态和动作的集合。通常有一些并不可行或不必要，但

是笛卡儿操作不会筛选出最优结果。为了决定哪些行为可以得到较佳的状态，例如，机器人不能停留在死机状态，这就需要给出回报。

R：S x A→R（S）；其中R是**回报函数**（reward function），提供立即回报给**代理**（agent）以便采取某个动作。代理一词用来指代替其他实体做出动作的实体。因此，机器人是在其他行星上的人类代理。不同的状态会给出不同的期望回报。在寻找到达目标的最优路径中，最优的一种定义就是使所有期望回报的总和最大。立即回报可以马上得到，但是回报总和必须由代理是否成功到达目标决定。例如，如果机器人向右转会找到一条清晰的路径。不幸的是这条清晰的路径不会到达目标岩石。这有点像前面讨论的验证，而这里的证实意味着机器人已经到达正确的岩石。在下一章，我们将看到更多的实用函数形式的回报例子以及用贝叶斯定理处理不确定性的例子。

在下面情况下隐马尔可夫模型（HMM）非常有用，（1）机器人不能肯定自己所处的位置，一个不确定的状态，或者（2）走哪一条路，一个不确定的参数。现在已有基于HMM的软件如HTK工具包（http://htk.eng.cam.ac.uk/）。这个软件已经成功用于语音识别、语音合成、字符识别和DNA定序。一旦通过基因定序确定了生物结构，就可以合成或者修改以识别和治疗疾病。其他的软件，例如HMMER是一个免费的HMM软件，用于蛋白质序列分析（http://hmmer.wustl.edu/）。

广受欢迎的Matlab有一个HMM工具包，支持不同类型的科学推理，特别适用于信号分析（http://www.ai.mit.edu/~murphyk/Software/HMM/hmm.html）。经典的例子是决定音素，这样当用户对着麦克风说话生成一个声学信号给计算机分析时就能决定其语音单词。尽管多种技术，如ANS已经可以处理分段的或发音缓慢的语音，但是不进行训练就对任何人发出的连续的语音进行识别（独立的说话人语音识别）仍然是一大难题。

路径规划也是许多视频游戏的基本问题，例如一个角色需要从一个地方到另一个地方但是有墙或障碍物阻碍。这时候通常使用A*算法。

3.18 小结

本章讨论了专家系统中通常使用的推理方法。推理在专家系统中尤为重要，因为它是专家系统解决问题的技术。另外还讨论了树、图、格在知识表示中的应用。并且举例说明了这些结构在推理中的优点。

演绎逻辑的讨论从简单的三段论逻辑开始。接着讨论了命题逻辑和一阶谓词逻辑。真值表及推理规则被描述为证明定理和语句的方法。此外还提到了逻辑系统的一些特性，如完备性、合理性和可判定性等。

本章还讨论了在命题逻辑和一阶谓词逻辑中证明定理的归结方法。举例说明了把一个合式公式转化成子句形式的9个步骤。在将wff转化为子句形式的内容中讨论了斯柯伦化、前束范式和合一等。

本章还讨论了推理的另一个有力的方法——类比法。尽管由于实现上的困难，它并没有被广泛地应用于专家系统，但类比法经常为人们所使用，在专家系统的设计中应予以考虑。此外，在MYCIN中讨论了生成与验证以及它的应用例子。在TEIRESAS中对元知识的应用以及它与专家系统的验证和证实之间的关系也进行了讨论。

习题

3.1 写一个可自学习的判定树程序。使用图3.3的动物知识对它进行教学。

3.2 写一个程序，可以自动把存储在二叉判定树里面的知识翻译成IF…THEN类型的规则。用习题3.1中的动物判定树来测试它。

3.3 画一个包括了图3.4草莓知识的语义网络。

3.4 画一个状态图用来表示经典的农民、狐狸、山羊、白菜问题的解决办法。在该问题中，有一个农民，一只狐狸，一只山羊和一棵白菜在河的一边。只能用一艘船来把他们运到河对岸。船一次只

可以运他们中的两样（只有农民可以划船）。如果农民不在，而狐狸和山羊被留在一起，狐狸会吃掉山羊。而如果山羊和白菜单独留在一起，山羊会吃掉白菜。

3.5 画一个状态图解决下面的结构化旅行问题。

(a) 有 3 种支付方法：现金、支票和记账

(b) 旅游者的兴趣：阳光、雪

(c) 根据旅游者的兴趣和钱有 4 个可能的目的地

(d) 有 3 种交通工具

写出 IF…THEN 规则，它们根据旅游者的兴趣和钱来给出目的地的建议。挑选实际的目的地并查明从你所在的地方到目的地所需的花费。

3.6 判断下列辩论是有效的还是无效的。

(a) $p \rightarrow q, \sim q \rightarrow r, r; \therefore p$

(b) $\sim p \vee q, p \rightarrow (r \wedge s), s \rightarrow q; \therefore q \vee r$

(c) $p \rightarrow (q \rightarrow r), q; \therefore p \rightarrow r$

3.7 使用文氏图的判定过程，判断下列三段论是否有效。

(a) AEE-4

(b) AOO-1

(c) OAO-3

(d) AAI-1

(e) OAI-2

3.8 证明下列各题是谬论还是推理规则，并举例说明。

(a) 复杂合并两难推理（Complex Constructive Dilemma）

$p \rightarrow q$

$r \rightarrow s$

$\underline{p \vee r}$

$\therefore q \vee s$

(b) 复杂分离两难推理（Complex Destructive Dilemma）

$p \rightarrow q$

$r \rightarrow s$

$\underline{\sim q \vee \sim s}$

$\therefore \sim p \vee \sim r$

(c) 简单分离两难推理（Simple Destructive Dilemma）

$p \rightarrow q$

$p \rightarrow r$

$\underline{\sim q \vee \sim r}$

$\therefore \sim p$

(d) 取反（Inverse）推论

$p \rightarrow q$

$\underline{\sim p}$

$\therefore \sim q$

3.9 从下列前提中推出结论。选自 Lewis Carroll 的《Alice's Adventures in Wonderland》。

(a) 这个房间中所有标有日期的信都是用蓝色纸写的。

(b) 除了用第三人称写的之外，没有一封信是用黑色墨水写的。

(c) 我没有把我可以读的信归档。

(d) 没有一封写在单独一页纸上的信是不标日期的。
(e) 所有没有划过的信，都是用黑墨水写的。
(f) 所有 Brown 写的信都以 Dear Sir 开头。
(g) 所有用蓝色纸写的信都归档了。
(h) 没有一封写在多于一页纸上的信，被划过。
(i) 没有一封以 Dear Sir 开头的信，用第三人称写。
提示：使用逆否定律并定义：
A=以 Dear Sir 开头　B=被划过　C=标日期的
D=已归档的　E=用黑墨水写　F=用第三人称写
G=我可以读的信　H=写在蓝色纸上的信　I=写在单页纸上的信
J=Brown 写的信

3.10 用谓词逻辑给出下列三段论的一个形式证明。
没有一个软件是有质量保证的
所有程序是软件
∴没有一个程序是有质量保证的

3.11 利用下列定义写出带量词的一阶谓词逻辑公式。
P（x）=x 是一个程序员
S（x）=x 是聪明的
L（x，y）=x 喜欢 y
(a) 所有程序员都是聪明的
(b) 有些程序员是聪明的
(c) 没有程序员是聪明的
(d) 有些人不是程序员
(e) 不是每个人都是程序员
(f) 每个人都不是程序员
(g) 每个人都是程序员
(h) 有些程序员不聪明
(i) 有一些程序员
(j) 每个人都喜欢某人
提示：利用附录 B

3.12 考虑下列谓词逻辑辩论：
马是一种动物
所以，一匹马的头是一只动物的头。给出下列定义：

H(x,y)=x 是 y 的头
A(x)=x 是一种动物
S(x)=x 是一匹马

前提和结论是：

$(\forall x)(S(x)\rightarrow A(x))$
$(\forall x)\{[(\exists y)(S(y)\wedge H(x,y))]\rightarrow[(\exists z)(A(z)\wedge H(x,z))]\}$

用归结反驳法证明结论。写出结论的所有 9 个子句转换步骤。

3.13 (a) 根据你的银行或储蓄信息以及贷款标准获得汽车贷款。写一个规则的反向链系统以判定一个申请者是否能获得汽车贷款。尽可能写得详细。
(b) 根据房屋贷款标准，对你的汽车贷款程序作出修改，使它能像解决汽车贷款那样解决房子

贷款问题。

(c) 根据商业贷款标准，对汽车贷款程序作出修改以解决商业贷款问题。

3.14 写一个因果规则的产生式系统来模拟汽车里面汽化器的工作。你可以从你汽车的维修指南里获得信息。尽可能写得详细。

3.15 用你的汽车维修指南，给定汽车症状，写一个可以诊断和修复汽车电路问题的产生式系统。

3.16 如果一个结论是错的，而你想判别由此推出的事实，你会使用溯因法还是其他推理方法？试解释之。

3.17 使用图 3.4 中的部分判定树，写出识别草莓的 IF…THEN 规则。

3.18 一个父亲和他的两个儿子待在河的左岸，希望过到河的右岸。父亲重 200 磅，每个儿子重 100 磅，只有一只可以容纳 200 磅的船提供。至少要有一个人划船。画出判定树，指明所有可能的过河方法。指出可以让他们全都成功过河的路径。假设下述操作在你的树中标志为：

SL_一个孩子从左岸到右岸去

SR_一个孩子从右岸到左岸去

BL_两个孩子从左岸到右岸

BR_两个孩子从右岸到左岸

PL_父亲从左到右过河

PR_父亲从右到左过河

3.19 (a) 画出图 3.11 汽车判定树的“与－异或”逻辑图。注意要用标准的与、或和非运算来实现异或。

(b) 写一个 IF…THEN 规则的正向链，用来决定可根据那一个假设，例如：卖或者修，进行推理。

3.20 以下陈述采用了哪种推理（如果采用了推理）？

(a) “我坚持说在 Iraq 和 Saddam 和 al Qaeda 之间存在联系的原因是在 Iraq 和 al Qaeda 之间存在联系”

(b) “如果一个高级法院审判的公正性被如此藐视是合理的，那么国家所处的情况比我想像中糟糕得多”。

参考文献

(Giarratano 93). Joseph Giarratano, *CLIPS User's Guide*, NASA, Version 6.2 of CLIPS, 1993.

(Kaelbling 98). Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra, "Planning and Acting in Partially Observable Stochastic Domains," *Artificial Intelligence* 101 pp. 99-134, 1998.

(Klir 98). George J. Klir and Mark J. Weirman, *Uncertainty-Based Information, Physica-Verlag*, 1998.

(Lakemeyer 03). ed. by Gerhard Lakemeyer and Bernhard Nebel, *Exploring Artificial Intelligence in the New Millennium,* Morgan Kaufmann Publishers, 2003.

(Lakoff 00). George Lakoff and Rafael E. Nunez, *Where Mathematics Comes From*, Basic Books, 2000.

(Merritt 04). Dennis Merrit, "Building a Custom Rule Engine with PROLOG," *Dr. Dobb's Journal*, pp. 40-45, March 2004.

(Russell 03). Stuart J. Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach,* 2nd Edition, 2002.

第4章　不确定性推理

4.1　概述

本章将讨论使用**概率理论**（probability theory）和**模糊逻辑**（fuzzy logic）进行**不确定性**（uncertainty）推理的各种方法。由于专家系统的主要功能之一就是能像人类一样处理不确定性，所以该课题在专家系统中非常重要。如果已知的算法或判定树令人满意，就不需要专家系统了。比起必须得到所有事实才能得到结论的判定树，处理不确定性是专家系统的主要优势。概率理论是某些不确定性理论的基础，因此本章将涵盖那些在专家系统中与概率化不确定性以及模糊逻辑相关的内容。

在下一章中将介绍处理不确定性的其他理论，并更详细地从近似推理的形式化角度来学习模糊逻辑。尽管只用一种不确定性理论（如经典概率理论）来处理问题就比较方便，但是每种理论都有其优缺点。这一点类似于一个程序员必须面对的在平常的计算机程序中决定采用什么算法和数据结构。通过理解不确定性的各种处理方法的优缺点，可以构建最适于某个专门领域的专家系统。有些专家系统工具把对不确定性的处理嵌入到语言中，如 CLIPS 有两个定制的处理模糊逻辑的版本，加拿大国家研究委员会的 FuzzyClips 以及 Togai InfraLogic。前言中也讨论了一些其他版本。

在使用这样一个定制工具前，必须确保这是处理不确定性的正确方法。也许，你所需要的仅是一些你可以自己编程的特殊规则，或者只需定义一个不确定性函数而不是使用整个特制工具。另一个可能性是你想要模型化不止一类不确定性，但是没有一种工具可以处理所有的可能情况。附录 G 展示了许多处理概率的软件工具和其他资源。其中有一些完整的工具，也有一些是特定的类别，例如在 C++ 中，帮助你修改和重新编译 CLIPS 使之更方便添加不确定性或其他你所喜欢的特性。这再次验证了 CLIPS 的一个巨大优势，作为开放资源 CLIPS 可以由用户根据自己需要进行定制而不是要用户去适应工具。在 CLIPS 的官方网站上列出了使用 Java 等其他语言的 CLIPS 版本。

4.2　不确定性

不确定性可以理解为缺少足够的信息来作出判断。不确定性是一个问题，因为它妨碍我们作出最好的决定，甚至导致作出一个坏的决定。在医学上，不确定性会使得医生没有发现最好的疗法或采用不正确的疗法。在商业上，不确定性可能意味着经济上的损失而不是获利。

为此，人们提出了许多理论来处理不确定性。其中包括经典概率，贝叶斯概率（Bayesian probability）（Castillo 97），在经典集合论基础上的 Hartley 理论，在概率论基础上的香农理论，Dempster-Shafer 理论，马尔可夫模型和 Zadeh 的**模糊理论**（fuzzy theory）。特别是，香农理论和模糊理论在电信方面得到了普遍的证实并且已经被应用到生物学、心理学、音乐和物理学等多种领域。模糊逻辑也用于很多消费应用中，从不是像 19 世纪用电子时间控制而是按衣物的干净程度判断来清洗衣物的洗衣机，到产生美丽照片的照相机。模糊逻辑，也称为近似推理，将在下一章中详细讨论。

所有生物都是处理不确定性的专家，否则就不能存在于这个现实的世界。人类尤其擅长处理有关交通、天气、工作、学校等方面的不确定性事物。过一会儿，读者将会看到，我们将全都变成如何在各种交通状况下驾驶、处理寒冷、选择容易的课程等方面的专家，有些人会成为选择任何事情的最简单处理方法的专家。处理不确定性需要不确定性推理并需要很多常识。常识的唯一问题是并不一定要“好的判断力”，而只是在正常的情况下作出正常的处理。但有时正常的方法并不是最好的方法，这就是不确定性推理之所以重要的原因。例如，如果你曾经玩过投币机并且赢了一些钱，然后开始输钱，常识会让你觉得如果你玩的时间足够长，最终你会赢。但不幸的是，常识不会告诉你，在你再次赢钱之前你可能会用完你的钱，信用卡也会超支，汽车会被卖掉，结婚戒指会被当掉。

在第3章描述的演绎推理是**精确推理**（exact reasoning），因为它处理的是精确事实和由此得出的精确结论。回忆一下在每个演绎辩论中，结论必须由前提推出。前提为真，结论必为真，反之，前提为假，则结论必为假。

但是，归纳辩论不像演绎辩论一样保证结论的正确性。归纳辩论的前提为结论提供支持而不是保证。例如，给出一个整数序列，1，2，3，你可以使用归纳去假设下一个数字是4。但是，如果你被告知下一个数字是5呢？那么，1，2，3可能是从著名的斐波那契数列中取出的序列，其中，每个数字是前两个数字的和，其序列是0，1，1，2，3，5。在一个归纳辩论中，支持结论的前提越多，结论为真的可能性就越大。

但是，关于归纳支持结论的前提越多结论就越真的解释也存在问题。假设你是右派乌鸦，听到一左派乌鸦宣称所有乌鸦是黑的。于是你去考察乌鸦，一段时间后发现所看到的1 000 000只乌鸦都是黑的。这时某些右派乌鸦可能会放弃，但既然这个理论是左派乌鸦提出的，你可能认为“只是因为我没有看到一只非黑色的乌鸦，但并不代表它不存在，也许只是隐藏在某个地方”。这也许是正确的。考虑“所有乌鸦是黑的”，在逻辑上等价于“所有不是黑色的东西不是乌鸦”。

例如，一个红苹果当然不是黑乌鸦。通过逐步提高支持结论的前提的数量，例如非黑色的东西，来提高结论的可靠性。所以你看到的红苹果越多，越相信所有的乌鸦是黑色的。这个愚蠢的故事称为**乌鸦诡论**（Raven Paradox），因为它演示了归纳如何同直觉抵触。

现在这可能成为灾难，因为有很多的红苹果，意味着左派乌鸦说所有乌鸦是黑色的是正确的。但是我们永远不放弃。考虑还有很多黑色的黑莓。所以如果你坚持计算不是黑乌鸦的黑色的东西，那么这些可以减弱前提。如果你到一个食品仓库，开始计算黑莓的数量，你会发现有很多黑莓，其数量可以超过红色苹果。那么既然有比红苹果更多的黑莓，那么左派人士是错的，一定有不是黑色的乌鸦，只不过你还没找到而已。

计算红苹果或任何非黑色的东西来支持所有乌鸦是黑色的论点将导致错误的结论。处理乌鸦诡论有很多种方法，但最好的方法应该是贝叶斯理论，本章稍后将讨论这个理论。

当包括不确定的事实时，精确推理就不再适用，尽管人们可能会误以为也像前面讨论的乌鸦和赌博问题一样适用。不确定性增加了可能的输出结果，因此也许会变得不但困难而且可能找不出最优解。更糟糕的是，不确定性还可能导致错误的推理，而使我们花费所有时间在统计黑莓数量上。

遗憾的是，确定最优结论并不是件容易的事。人们已提出多种方法来处理不确定性并帮助选择最优结论。但是在处理不确定性问题时，我们也许应该满足于好的解而不是最优解。另一方面，一个现实时间内可获得的99%好的解也许比一个需花费百万年时间才能得到的最优解要好得多。根据应用而选择最合适的方法是专家系统设计者的责任。虽然有些专家系统工具含有不确定性推理机制，但是，它们在允许应用其他方法方面一般不灵活。有一个谚语，如果你拥有的唯一工具是铁锤，那么所有的东西看起来都像铁钉。选择一个只在一个场合中有效的工具，就像选择了一个铁锤。

虽然有很多能用精确推理处理的专家系统应用，但更多的是需要**不精确推理**（inexact reasoning），这是因为事实和知识本身不一定是精确的。一个很好的例子是医疗应用。为了获取最大利益，想要通过最少的测试检验得到合适的治疗方案，因为测试检验增加花费。在专家系统中，可能存在不确定的事实、规则，或两者都有。处理不确定性的经典成功专家系统例子有用于医疗诊断的MYCIN和用于矿产探测的PROSPECTOR。

这些系统在文献中经常被引述是因为它们有完善的文档，而且可以显示专家系统能像人类专家一样思考甚至比人类专家处理得更好。

在MYCIN和PROSPECTOR系统中，甚至可以在证明结论所必需的所有证据还不知道的情形下得出结论。虽然可以通过做更多的测试来得到一个更可靠的结论，但做更多的测试有一个时间和开支的问题。这些时间和开支的限制在医疗中是特别重要的。推迟治疗而做更多测试会增加相当大的费用，在这期间，病人还可能会死亡。在矿产探测的例子中，增加测试的开支也是一个很重要的因素。在你

有95%的成功率时开始钻探可能要比花费几十万美元把成功率提高到98%时再钻探更合算。

4.3 误差种类

许多不同种类的**误差**（error）都可以引起不确定性。有关不确定性的不同理论都尝试解决其中的部分或全部来提供更可靠的结论。图4.1列出了一个误差的简单分类表。

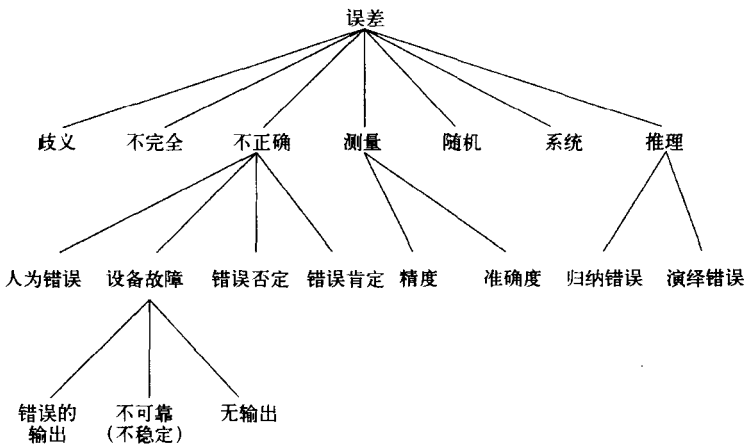


图4.1 误差的种类

这个图严格来说应该画成一个网格，因为在许多不同类型的误差之间是存在联系的。例如，人为的误差就可以有歧义、度量、推理等等的误差。表4.1给出了这些误差的一些例子。

表4.1 常见误差类型的例子

例 子	误 差	原 因
将阀门关上	歧义	哪个阀门?
调节阀门1	不完全	什么方式?
将阀门1关上	不正确	正确是打开
阀门是固定的	错误肯定	阀门不是固定的
阀门不是固定的	错误否定	阀门是固定的
将阀门1调节到5	不精确	正确是5.4
将阀门值1调节到5.4	不准确	正确是9.2
将阀门1调节到5.4或6或0	不可靠	设备错误
阀门1的设置为5.4或5.5或5.1	随机错误	统计上的错误
阀门1的设置为7.5	系统误差	刻度错
阀门1没有固定，因为它以前从未固定过	无效归纳	阀门是固定的
输出是正常的，所以阀门1状况良好	无效演绎	阀门处在开放位置上

第一种列举出来的误差是**歧义**（ambiguity），即对某事物的解释多于一种。第二种类型是**不完全**（incompleteness），即**遗漏**了某些信息。第三种误差是**不正确**（incorrectness），即信息是错误的。引起不正确的可能原因是人为的错误，例如偶然错读刻度或数据、撒谎或错误信息、以及设备失灵等。

一个**假设**（hypothesis）是一个需要检验的假定。**零假设**（null hypothesis）是指最初的假定，如"阀门是固定的"。不正确信息的一种类型是**错误肯定**（false positive），意思是对一个不真假设的接受。类似地，**错误否定**（false negative）是指对一个真假设的拒绝。因此，如果阀门确实不是固定的，则接

受它是固定的就是错误肯定。在统计学上称之为**第一类误差**（Type Ⅰ error）。类似地，如果阀门确实是固定的，则拒绝接受阀门是固定的假设就是错误否定或叫**第二类误差**（Type Ⅱ error）。

表4.1中接下来的两种误差是**测量误差**（errors of measurement）。它们是**精度**（precision）误差和**准确度**（accuracy）误差。虽然这些术语有时被同义使用，但它们是不同的。考虑两把尺子，一把的刻度是毫米，另一把的刻度是厘米。显然，毫米尺子比厘米尺子有更高的精度。但是，假设毫米尺子的刻度在制造时有错。在这种情形下，毫米尺子是不准确的，它的测量也是不可信的，除非知道一个正确的校正因子。这样，准确是相对于真实而言，而精确则相对于真实的程度而言。对这两把尺子来说，毫米尺子的精度是厘米尺子的10倍，但是如果不准确则会导致严重问题。

另外一种类型的误差是**不可靠**（unreliability）。如果测量设备提供的事实是不可靠的，那么这些数据就是**不稳定的**（erratic）。一个不稳定的读数不是一个常数而是波动的。它有时正确，有时不正确。

一个波动的读数可能是由所研究系统的随机本质引起的，如放射性原子的衰变。因为引起衰变的原因是一种量子现象，所以衰变的速率是在一个平均数附近随机波动的。在一个平均数附近的随机波动构成了一个**随机误差**（random error），并且引起该平均值的不确定性。另外引起随机误差的可能原因是布朗运动、由热效应引起的电子噪音等等。不过，引起不稳定的原因也可能是连接不紧。

系统误差（systematic error）不是由随机性而是由某些偏差引起的。例如，对一把尺子的错误校准会使得它的刻度比正常的小，这样就造成了其读数比正常高的一个系统误差。

4.4 误差与归纳

接下来的一种误差："因为阀门以前从不固定，所以是不固定的"是**无效归纳**（invalid induction）。与**归纳**（induction）过程相对的是演绎。习惯上认为演绎是由一般到特殊，例如：

所有人都会死
苏格拉底是一个人
∴苏格拉底会死

是推出特殊的情形苏格拉底会死

归纳则是由一般到特殊，例如，

我的碟片从未跌碎过
∵我的碟片永不会跌碎

这里倒三角符号∵代表**归纳的所以**（inductive therefore），对应于三角符号∴，**代表演绎的所以**（deductive therefore）。

虽然你还会从一些书籍中看到演绎是从一般到特殊而归纳是从特殊到一般的论述。但当代逻辑认为，当前提为真时，演绎确保结论的正确性。而归纳辩论仅意味着前提越强，结论越可能为真。

考虑下面辩论：

我的两个宠物是Smokey和Boots
Smokey有尾巴
Boots有尾巴
∴我的所有宠物都有尾巴

这是一个有效演绎辩论的例子，其中，特定的前提推出了一般的结论。相反的，原来的教导是说演绎从一般到特殊，并且仍然可以在一些旧的辞典中找到。这种辩论也称为**合理的**（sound），因为它不但有效而且基于真的前提。这里我们说真，是指语义上有效，也就是在现实世界中是真实的，我的宠物都有尾巴。

现在考虑一下辩论：

我的两个宠物是Smokey和Boots

Smokey有象鼻
Boots有象鼻
∴我的所有宠物都有象鼻

这也是一个有效的辩论但不再是合理的，因为Smokey和Boots不是大象，不应该有象鼻。

除了数学上的归纳，归纳辩论从来不能证明是正确的。归纳辩论只能提供一些结论正确的可信度，在一个归纳辩论中，我们可能并不具有太大的可信度。下面是一个辩论：

火灾报警响起
∴一场火灾发生了

一个更强的辩论是：

火灾报警响起
我闻到烟雾气味
∴一场火灾发生了

虽然这是一个强辩论，但是它不能作为火灾发生的证明。烟雾可能是由在烤架上的汉堡包产生且火灾报警也可能是偶然响起。下面才是一个火灾发生的证明：

火灾报警响起
我闻到烟雾气味
我的衣服着火了
∴一场火灾发生了

注意这是一个演绎因为你能看到火焰。尽管是特殊的前提条件，而不是普通情况。

专家系统可包括演绎和归纳规则，其中归纳规则是启发性的。归纳法也被用于规则的自动生成，这一点将在后面的知识获取章节中讨论。

除了不确定事实外，如果专家系统的规则是基于启发式的，那么它们也可能存在不确定性。启发式规则常被称为拇指规则，因为它是基于经验的（例如，如果你要学习正确使用铁锤钉钉子的重要规则，首先锤一下你的拇指。这称为基于“拇指规则”的学习。这当然比坐着听一个3小时的讲座昏昏欲睡的学习效果好多了）。

有时候经验是一个好的向导。但是，它不可能百分之百地适用于各种情形，这是因为经验是启发性的。例如，你可能认为每个跳下五十层楼的人都会死去。但是，人可以跳下五十层的楼而不死——如果他带着降落伞的话。

人类专家的一个重要特征是能熟练地在不确定性下推理。即使在非常不确定性的情形下，专家们通常都能作出很好的判断，否则他们就不会再被叫做专家了。

另一个特征是如果发现一些初始事实是错误的，专家们会很容易地修改他们的观点。这就是第3章讨论的非单调推理。设计一个专家系统可回溯其推理过程非常困难，因为所有的中间事实都要保存起来并且需要保留规则激活的记录。人类专家似乎不需要太多的努力就可以修改他们的推理。

除了归纳上的错误，也存在一些演绎上的错误或谬论，如在第3章所讨论的。一个在第3章讨论过的谬误模式例子是：

$p \rightarrow q$
q
$\therefore p$

例如，

如果阀门状况良好，
　　那么输出是正常的
输出是正常的
∴阀门状况良好

是一个错误辩论。阀门可能处于一个开放位置，所以输出是正常的。但是，如果必须关闭阀门，问题就会显现出来。如果阀门需要快速关闭，情况会更严重，例如一个核反应堆需要紧急关闭的情况。

与那些我们已讨论过的误差相比，归纳和演绎误差是推理误差。这些类型的误差导致了规则的不正确形成。

一般，人们处理不确定信息似乎并不得心应手。即使专家也不能避免犯错误，尤其是在不确定性情况下。当专家知识一定要在规则中定量表示时，这可能是知识获取的一个主要问题。不一致、不精确和其他所有可能的不确定性误差都会暴露出来。此时，专家们会修正他们的知识，但这会推迟专家系统的完成。

4.5 经典概率

一个用于人工智能问题求解的古老但仍然很重要的工具是**概率**（probability）（Grinstead 97）。概率是一个处理不确定性的量化方法，它起源于17世纪，当时一些法国的赌徒向一些顶尖数学家，如Pascal、Fermat等求助。那时，赌博很流行，而且由于其中涉及大量的金钱，赌徒们都希望找到一种方法来帮助计算赢的胜算。事实上，经典的赌徒破产问题是就是赌徒们根据经验事实使用概率理论的一个证明：如果你玩的时间足够长，那匹马最终会赢的（Ross 02）（不幸的是正是这些以为可以通过学习概率得到成功的赌徒输的最多）。

经典概率（classical probability）理论首先是由Pascal和Fermat在1654年提出来的。自那时开始，人们做了大量工作，几个新的分支也得到了发展。概率在自然科学、工程学、商业、经济和实际上其他每个领域的应用都显示出来。

经典概率的定义

经典概率又叫做先验概率，因为它处理的是理想的游戏或系统。如在第2章所讨论的，术语先验意思是“超前的”，也就是说，不考虑真实的世界。

也就是说，当一个真实骰子的一面经过大量投掷而损耗后，会显示对某一特定的数存在偏差。同样，由于制造的关系，一个真实的骰子会显示它更倾向于更大的数，因为更大数的面有更多的小孔。这个偏差是通过对一个真实骰子的1 000 000次投掷数据的分析而得出来的。每20 000次投掷就会换一个新骰子，以避免由于磨损所造成的偏差。从表4.2中可以看到各个数的次数的比例。

表4.2 1 000 000次投掷骰子的结果

数	1	2	3	4	5	6
比例	0.155	0.159	0 .164	0.169	0.174	0.179

在一个理想系统里，所有数出现的机会都是相等的，这样使得分析起来更加容易。

经典概率的基本公式定义为概率：

$$P \approx \frac{W}{N}$$

其中W是获胜的数目，N是等可能性的**事件**（event）的数目，事件即一个实验或**试验**（trial）可能得出的结果。例如，如果你掷一次骰子，那么这次试验就有6种可能的事件。该骰子会在顶面是1、2、3、4、5、6其中之一停下来。在经典概率下，这6个事件假定是等可能的，所以得到顶面是1的概率P（1）为

$$P(1) = \frac{1}{6}$$

类似地，

$$P(2) = \frac{1}{6}$$

等等。

失败的概率 Q 是：

$$Q = \frac{N - W}{N} = 1 - P$$

因为概率是在游戏开始之前就计算出来，所以 P 的基本公式是一个先验定义。当在讨论概率时术语先验是指“超前的”或“在事件之前”。先验概率假定所有可能事件都是已知的且有相等的机会发生。

例如，在投掷骰子的例子中，每个面上的可能点数是 1、2、3、4、5 和 6。所以，如果骰子是**公平的**（fair）——不是灌铅骰子，则每个面出现的可能性都是相等的。类似地，在一副公平的纸牌中，所有 52 张牌都是知道的，并且在公平的洗牌后，每张牌被抽出的可能性都是相等的。骰子的每一面的可能性都是 1/6，而每张牌的可能性都是 1/52。

当重复试验而得出同一个结果，这样的系统就是**确定的**（deterministic）。如果不是确定的，就称为**不确定的**（nondeterministic）。但是，不确定和随机并不一样。**随机**（random）是指可能有好的或不好的含义。例如，一个随机事件像在拉斯维加斯掷骰子，结果可能让你成为百万富翁或者乞丐。相反的，一个系统被定义为不确定的只意味着对于相同的输入，可能有多于一种的方法得到多于一种的结果。

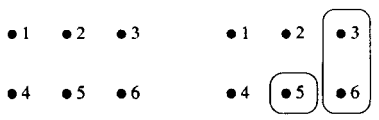
对于一个不确定的有限状态机，当输入一个数字时，可以得到状态 1 或 2。如果只有数字被输入，最终机器会识别出整数。如果带有小数点或带有指数符号的实数被输入，机器最终会辨认出这些内容并结束于一个不同的终态。通常，一个不确定的有限状态机所具有的状态比确定的少，而且可以被转换成一个确定的有限状态机。

人们在使用搜索引擎时已经熟悉了不确定性。输入相同的内容，你会发现最先出现的内容各不相同（除非它是一个赞助商的内容，永远出现在最开始）。例如，你在 google 中输入“giarratano expert systems”进行搜索，记录下最先出现的结果。然后再次搜索“expert systems giarratano”，你会发现两者的结果并不相同。

另一个例子是头痛时的处理方法。有很多方式治疗头痛，你选择哪一种取决于你当时的心情和条件限制。在有些专家系统工具如 CLIPS 中，不确定用来防止激发规则时带有固定倾向。当有多个规则的模式符合条件，而且没有明确的优先说明哪一条规则被激发，推理机将从中任意选出一条规则加以执行。这就防止了某条规则因为固定的倾向被激发，例如倾向于执行第一条规则。类似的，当所有的知识以 IF…THEN 语句形式输入到一个传统的编程语言中，那么程序会不断地按照它们输入的次序进行尝试。事实上，这也是专家系统不采用传统编程语言编写的原因之一：避免太多确定性。专家系统推理机能够像人类一样操作，当具备相同优先级的规则满足执行条件时，并不总是做出相同的选择，例如：我应该吃阿司匹林还是 Bloody Mary 治疗头痛？

样本空间

一次试验的结果称做一个**样本点**（sample point），而所有可能的样本点的集合定义了一个**样本空间**（sample space）。例如，如果投掷一个骰子，任一个样本点 1、2、3、4、5 或 6 都可能出现。图 4.2 (a) 显示了投掷一个骰子的样本空间。该样本空间是集合 {1，2，3，4，5，6}。



(a) 一个骰子的样本空间　(b) 一个简单事件 {5}
和一个复合事件 {3，6}

图 4.2　样本空间和事件

一个事件是样本空间的一个子集。例如，当骰子出现是 1 时，事件 {1} 发生。一个**简单事件**（simple event）只有一个元素，一个**复合事件**（compound event）则有多于一个的元素，如图 4.2（b）所示。

决定样本空间的一个图形方法是构造一棵**事件树**（event tree）。作为一个简单的例子，假设有两台计算机，它们或者工作，W，或者不工作，D。图 4.3 显示的是一棵事件树，表 4.3 显示的是形成样本空间的一个表。注意，复合事件以 {计算机 1，计算机 2} 的顺序列出。样本空间是 {WW，WD，DW，DD}。

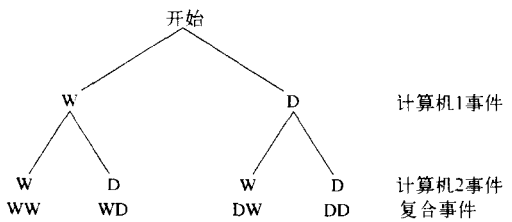


图 4.3 复合事件的事件树

表 4.3 二元事件的样本空间

		计算机 2	
		W	D
计算机 1	W	WW	WD
	D	DW	DD

因为只包含两种可能，所以这种类型的事件树是一棵二叉树。也就是说，计算机或者工作或者不工作。很多其他的例子都有这种类型的事件树。例如，投掷硬币也会得出一棵二叉树，这是由于每次投掷只会有两种可能情形。

概率和统计都是在其他理论中有广泛应用的领域。统计研究有关**种群**（populations）数据的收集和分析，种群即是被抽取样本的一个集合。统计的一个目标是从抽取样本种群中作出其父种群的推断。统计推断的一个典型例子是通过电话投票来推断某一个候选人在已注册的选民中的得票的比例。

概率理论的一个应用是推断投票者的样本是否真正代表所有的投票者，或者是可能偏向某一个党派，虽然这个例子与真实对象如人有关。其他的一些种群如投掷硬币的可能数目，是假定的，每次硬币的投掷都是所有硬币投掷的假定种群的一个样本。

归纳和演绎是有关种群推理的基础，如图 4.4 所示。给定一个已知种群，演绎可以让我们作出有关未知样本的推断。相应地，给定一个已知样本，归纳可以让我们作出有关未知种群的推断。

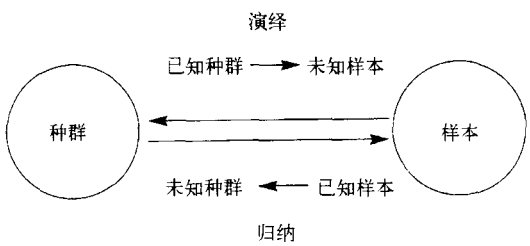


图 4.4 关于种群和样本的演绎和归纳推理

概率理论

概率的形式理论由以下 3 条公理组成：

公理 *1*： $0 \leq P(E) \leq 1$

这条公理规定了概率的取值范围是 0 到 1 之间的实数，不能为负数。一个**必然事件**（certain event）的概率指定为 1，**不可能事件**（impossible event）的概率指定为 0。

公理 *2*： $\sum_{i} P(E_i) = 1$

该公理阐述了所有互不影响，即**互斥事件**（mutually exclusive event）的概率和为 1。互斥事件没有公共样本点。例如，一台计算机不能同时工作正常和工作不正常。

该公理的一个推论是：

$$P(E) + P(E') = 1$$

其中 E′是事件 E 的补。这一推论意味着一个事件发生的概率加上不发生的概率等于 1。即事件的发生与不发生是互斥的，且覆盖整个样本空间。

公理3： $P(E_1 \cup E_2) = P(E_1) + P(E_2)$

其中 E_1 和 E_2 是互斥事件。该公理意味着：如果 E_1 和 E_2 不能同时发生（互斥事件），那么它们其中之一发生的概率是它们的概率的和。

根据这些公理，可推导出定理来计算其他情形下的概率，如我们将在后面看到的相容事件。虽然这些公理是概率论的基础，但注意到它们均未规定基本概率 P（E），因为事件的基本概率是由诸如先验概率等的方法计算出来的。

这些公理使概率有了一个合理的理论基础。它们也称为**客观概率理论**（objective theory of probability）。这些公理是由 Kolmogorov 提出的。而使用条件概率公理的另一套等价理论则由 Renyi 创建。

4.6 经验主观概率

与具有等可能性的理想游戏不同，经典概率不能回答像你的磁盘驱动器明天会坏的概率或你的伴侣的期望寿命等问题。

与先验概率相反，**经验概率**（experimental probability）用频率分布的极限定义了一个事件的概率 P（E）：

$$P(E) = \lim_{N \to \infty} \frac{f(E)}{N}$$

其中 f（E）是在 N 个总观察结果中，事件 E 出现的频率。这种概率叫做事后概率，意即“在事件之后”。事后概率也称作**后验概率**（posterior probability）。其旨在从大量的试验中测出事件发生的频率，并以此推导出经验概率。

例如，要决定你的驱动器要坏的经验概率，你可以调查一下其他用这种驱动器的人。假定调查结果如表 4.4 所示。

表 4.4 驱动器损坏的假定时间

已损坏的总百分比	已用小时数
10	100
25	250
50	500
75	750
99	1000

如果你的驱动器已用过 750 小时，则有 75%的可能明天就坏。注意 75%的数字是由归纳而不是由演绎得到。与理想游戏不一样，你的驱动器与其他人的并不完全相同。在材料使用、质量控制、环境条件及使用等方面的不同都会影响驱动器及其寿命。当然生产简单的设备如骰子或纸牌比复杂的设备如驱动器具有更高的宽松度。

人寿保险公司的死亡表，以年龄、性别的函数方式表示一个人的死亡概率，从这可得出另一类型的经验概率。由于每个人不同，因而从表中计算死亡率也是归纳，对房屋保险亦然。除非你的房子被烧了大片而被纳入个别经验概率计算，否则就得按相似房屋类型的经验概率来计算。

另一种概率是**主观概率**（subjective probability）。若问你：在 2020 年，使用超导引擎的汽车费用为 $10000 的概率是多少？由于没有这些汽车的费用数据，因此也无从推断 $10000 的费用是否合理。

主观概率处理不可重复且无历史资料推断的事件，如在一个新位置钻油井。专家的主观概率总比乱猜一通好，而且往往很准确（否则专家就不再是专家了）。

与其说主观概率是基于公理或经验度量的概率，不如说它实际上是表达信念或意见的概率。信念和意见在专家系统中扮演着重要角色，这一点我们将在本章看到。表 4.5 总结了几种不同的概率：

表 4.5 概率类型

名 称	公 式	特 征
先验概率	$P(E)=\frac{W}{N}$	可重复事件
（经典的，理论上的，数学上的，均匀的，等概率的，等可能的）	其中 W 是事件 E 在 N 次总可能结果中发生的次数	相同的可能结果已知精确的数学形式不以实验为基础所有可能事件和结果都已知
后验概率（实验的，经验的，科学规律的，相对频率，统计的）	$P(E)=\lim_{N\to\infty}\frac{f(E)}{N}$ 其中，f（E）是事件 E 在 N 次总结果中发生的频率 f	基于实验的可重复事件用有限次实验来近似未知精确的数学形式
$P(E)\approx\frac{f(E)}{N}$		
主观概率（个人的）	参看第 4.12 节	不可重复事件未知精确的数学形式不可能有相对频率基于专家的观点、经验、判断、信念

4.7 复合概率

复合事件的概率可由其样本空间计算。作为一个简单的例子，考虑掷出骰子为双数且能被 3 整除的概率，用文氏图在骰子的样本空间表示集合：

$A=\{2,4,6\}\quad B=\{3,6\}$

如图 4.5 所示。

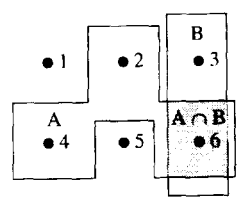


图 4.5 掷出骰子为双数且能被 3 整除的复合概率

$A\cap B=\{6\}$

因而该事件的概率为：

$$P(A\cap B)=\frac{n(A\cap B)}{n(S)}=\frac{1}{6}$$

其中 n 为集合元素个数，S 为样本空间。

彼此互不影响的事件，称为**独立事件**（independent event）。两个独立事件 A 与 B 同时发生的概率为它们的概率之积，事件 A 与 B 称为**两两独立**（pairwise independent）。

$$P(A\cap B)=P(A)\ P(B)$$

当且仅当上式成立时，两个事件称为**随机独立事件**（stochastically independent event）。其中术语 stochastic 源于希腊文，意思是“猜测”，它常用作可能、随机、机会等的同义词。所以，与确定性实验有非随机结果相反，一个 stochastic 实验会有随机的结果。

对三个事件，你可能会猜测其独立条件为

$$P(A\cap B\cap C)=P(A)\ P(B)\ P(C)$$

遗憾的是，生活和概率并非如此简单。N 个事件**相互独立**（mutual independence），需要满足 2^N 个等式。这种要求可在下列等式中归结出，其中 * 表示事件及其补集均需满足该等式：

$$P(A^*_1\cap A^*_2\cap\ldots\cap A^*_N)=P(A^*_1)\ P(A^*_2)\ \ldots\ P(A^*_N)$$

对 3 个事件而言，上述独立性的等式可变为需要满足下列所有等式。

$$\begin{aligned}
&P(A\cap B\cap C)=P(A)\ P(B)\ P(C)\\
&P(A\cap B\cap C')=P(A)\ P(B)\ P(C')\\
&P(A\cap B'\cap C)=P(A)\ P(B')\ P(C)\\
&P(A\cap B'\cap C')=P(A)\ P(B')\ P(C')\\
&P(A'\cap B\cap C)=P(A')\ P(B)\ P(C)\\
&P(A'\cap B\cap C')=P(A')\ P(B)\ P(C')\\
&P(A'\cap B'\cap C)=P(A')\ P(B')\ P(C)\\
&P(A'\cap B'\cap C')=P(A')\ P(B')\ P(C')
\end{aligned}$$

任意两个事件相互独立，并不能保证所有事件是相互独立的，这可在习题 4.6 中得到验证。

就拿刚才的骰子来说，点数为偶数与能被 3 整除的事件相互影响，并非随机实验。但一个骰子的点数为偶数和另一骰子的点数能被 3 整除，这两个事件的概率却是独立的。

$$P(A \cup B) = P(A)\ P(B) = \frac{3}{6} \times \frac{2}{6} = \frac{1}{6}$$

现在让我们来考虑一下事件的并的情形，P（A∪B)。定义 n 为一个函数，返回一个集合的元素个数。假如我们把 A 的元素个数与 B 的元素个数相加之后再除以样本空间的元素总数 n（S)，

$$(1)\ P(A \cup B) = \frac{n(A) + n(B)}{n(S)} = P(A) + P(B)$$

那么当集合有重叠的话，结果将会很大。如果你观察图 4.5，应用这个公式可以得到

$$P(A \cup B) = \frac{3 + 2}{6} = \frac{5}{6}$$

但你很容易发现，并集里面只有 4 个元素，因为

$$A \cup B = \{ 2, 3, 4, 6 \}$$

问题出在当 n（A）和 n（B）相加时把同时属于 A 和 B 的交集 {6} 计算了两次。

正确的公式仅仅需要减去额外的交集概率

$$(2)\ P(A \cup B) = P(A) + P(B) - P(A \cap B)$$

结果减少为：

$$P(A \cup B) = \frac{3}{6} + \frac{2}{6} - \frac{1}{6} = \frac{4}{6} = \frac{2}{3}$$

当集合不相交即没有任何公共元素时公式（1）成立。它是公式（2）的一个特例。公式（2）称为**加法律**（Additive Law)。运用前面讨论过的 3 个概率公理，加法律可作为一个定理被推出。对于 3 个事件，加法律是：

$$P(A \cup B \cup C) = P(A) + P(B) + P(C) - P(A \cap B) - P(A \cap C) - P(B \cap C) + P(A \cap B \cap C)$$

其他定律如非 A 非 B 和 AB 的异或也可以从公理中推导出来。这些定律对于复合概率的作用在于：（1）可以不用去试验概率的每一种可能组合；（2）不用计算大样本空间的独立元素。

4.8 条件概率

不是互相排斥的事件可相互影响。一个事件的发生可能导致我们去更改另一个将发生事件的概率。

乘法律

在事件 B 发生的条件下，事件 A 发生的概率称为**条件概率**（conditional probability)，用 P（A|B）表示。条件概率定义为：

$$P(A \mid B) = \frac{P(A \cap B)}{P(B)} \quad \text{for } P(B) \neq 0$$

概率 P（B）是在任何信息被知道之前的先验概率。当和条件概率一起使用时，先验概率有时也称为**非条件概率**（unconditional probability）或**绝对概率**（absolute probability)。

条件概率的定义可以作出直观解释，如果你观察图 4.6 的有 8 个事件的样本空间的例子。从图 4.6，概率可用事件数 n（A）或 n（B）和样本空间的总事件数 n（S）的比率来计算。即：

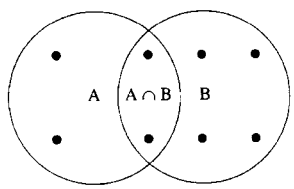


图 4.6 两个相交事件的样本空间

$$P(A) = \frac{n(A)}{n(S)} = \frac{4}{8}$$

$$P(B) = \frac{n(B)}{n(S)} = \frac{6}{8}$$

如果我们知道事件B已经发生，减少的样本空间仅仅为B：

$$n(S) = 6$$

既然B已发生，则仅需考虑A中与B一起的事件

$$P(A \mid B) = \frac{n(A \cap B)}{n(B)} = \frac{2}{6}$$

要用概率的形式来表达结果，只需上式中的分子分母同时除以n (s)：

$$P(A \mid B) = \frac{\frac{n(A \cap B)}{n(S)}}{\frac{n(B)}{n(S)}}$$

$$= \frac{P(A \cap B)}{P(B)} \quad \text{for } P(B) \neq 0$$

于是两个事件的**乘法律**（Multiplicative Law）就被定义为：

$$P(A \cap B) = P(A \mid B)\, P(B)$$

等价于：

$$P(A \cap B) = P(B \mid A)\, P(A)$$

三个事件的乘法律是：

$$P(A \cap B \cap C) = P(A \mid B \cap C)\, P(B \mid C)\, P(C)$$

进一步，**一般的乘法律**（Generalized Multiplicative Law）为：

$$P(A_1 \cap A_2 \cap \ldots \cap A_N) = P(A_1 \mid A_2 \cap \ldots \cap A_N) \cdot P(A_2 \mid A_3 \cap \ldots \cap A_N) \cdot \ldots P(A_{N-1} \mid A_N)\, P(A_N)$$

作为概率的一个例子，表4.6给出了使用X品牌的磁盘驱动器在一年内损坏的假设概率。

表4.6 磁盘驱动器在一年内损坏的假设概率

	X 品 牌	非X品牌X′	行 总 和
损坏C	0.6	0.1	0.7
不损坏C′	0.2	0.1	0.3
列总和	0.8	0.2	1.0

表中间部分的概率0.6、0.1、0.2和0.1称为**内在概率**（interior probabilities），表示事件的交集。行和列的和显示总数，称为**边际概率**（marginal probabilities），因为它们位于表的边缘。

表4.7和表4.8更加详细地体现了集合和概率的含义。图4.7给出了相交样本空间的文氏图。从表4.8我们可以看出，行和列的概率总和均为1。

表4.7 集合解释

	X	X′	行总和
C	$C \cap X$	$C \cap X'$	$C = (C \cap X) \cup (C \cap X')$
C′	$C' \cap X$	$C' \cap X'$	$C' = (C' \cap X) \cup (C' \cap X')$
列总和	$X = (C' \cap X) \cup (C \cap X)$	$X' = (C' \cap X') \cup (C \cap X')$	样本空间

表 4.8 两个集合的概率解释

	X	X′	行总和
C	P (C∩X)	P (C∩X′)	P (C)
C′	P (C′∩X)	P (C′∩X′)	P (C′)
列总和	P (X)	P (X′)	1. 0

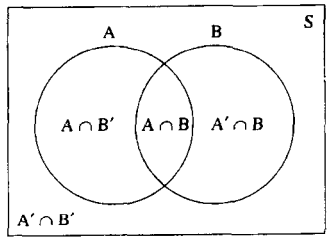


图 4.7 两个集合的样本空间解释

使用表 4.8，所有事件的概率都可以计算出来。一些概率如下：

(1) 使用 X 品牌和非 X 品牌（样本空间）损坏的概率

$$P(C) = 0.7$$

(2) 对于样本空间，不损坏的概率

$$P(C') = 0.3$$

(3) 使用 X 品牌的概率

$$P(X) = 0.8$$

(4) 不使用 X 品牌的概率

$$P(X') = 0.2$$

(5) 使用 X 品牌，且损坏的概率

$$P(C \cap X) = 0.6$$

(6) 在使用 X 品牌的条件下，损坏的概率

$$P(C \mid X) = \frac{P(C \cap X)}{P(X)} = \frac{0.6}{0.8} = 0.75$$

(7) 在没有使用 X 品牌的条件下，损坏的概率

$$P(C \mid X') = \frac{P(C \cap X')}{P(X')} = \frac{0.1}{0.2} = 0.50$$

当你读概率 (5) 和 (6) 的陈述时，好像意思相近，但是 (5) 仅仅是两个事件的交集而 (6) 是一个条件概率。交集 (5) 的意思是：

如果随机抽取一个磁盘驱动器，那么有 0.6 的可能是 X 品牌并且已经损坏。

也就是说，我们只是从磁盘驱动器的集中抽取样本。驱动器有一些是 X 品牌且已损坏 (0.6)；一些不是 X 品牌并且已损坏 (0.1)；一些是 X 品牌并且没有损坏 (0.2)；一些不是 X 品牌并且没有损坏 (0.1)。

相反，条件概率 (6) 的意义是非常不同的：

如果一个 X 品牌的驱动器被抽取，那么有 0.75 的可能它是已经损坏的。

注意在条件概率中，我们是挑选了那些我们感兴趣的项目 (X 品牌)，并把它们看作一个新的样本空间。

如果下列等式中有任何一个成立，则事件 A 和 B 是**独立的** (independent)。

$$P(A \mid B) = P(A) \text{ or}$$
$$P(B \mid A) = P(B) \text{ or}$$
$$P(A \cap B) = P(A)\ P(B)$$

如果这些等式有任何一个成立，则其他的也同样成立。

贝叶斯定理

条件概率 P (A|B) 说明了当事件 B 发生时事件 A 的概率。相反的问题就是去寻找**逆概率** (inverse

probability)，即当后面的事件发生时，前面的事件发生的概率。这类概率经常发生，如根据表现出来的症状进行医学或设备诊断时，问题就是要寻找最可能的起因。解决这类问题的方法是**贝叶斯定理**(Bayes' Theorem)，有时也称为贝叶斯公式、贝叶斯准则、贝叶斯定律，它是以 18 世纪英国牧师、数学家汤姆斯·贝叶斯的名字命名的。

作为贝叶斯定理的一个例子，让我们看一看它是如何应用在磁盘驱动器损坏这个问题上的。从条件概率 (6) 可知，X 品牌的驱动器有 75% 的可能性在一年内损坏，根据 (7)，非 X 品牌的驱动器在一年内损坏的概率是 50%。相反的问题是，假设你有一个驱动器并且不知其品牌，那么如果它损坏的话，是 X 品牌或非 X 品牌的概率是多少呢?

由于计算机厂家甚少生产自己的驱动器，所以你不知道驱动器的品牌这类事情是经常发生的。许多驱动器是从原设备供应厂商 (OEM) 买来然后重新包装以自己的标签出售。根据 OEM 提供的最低价格，驱动器可能年年不同而只有计算机厂商的标签是一样的。

已知一个驱动器损坏，是 X 品牌的概率可以用条件概率和结果 (1)、(5) 得出：

$$P(X \mid C) = \frac{P(C \cap X)}{P(C)} = \frac{0.6}{0.7} = \frac{6}{7}$$

也可以通过对分子用乘法律及 (1)、(3)、(6) 得出：

$$P(X \mid C) = \frac{P(C|X)P(X)}{P(C)} = \frac{(0.75)(0.8)}{0.7} = \frac{0.6}{0.7} = \frac{6}{7}$$

概率 P (X|C) 是逆概率或后验概率，它表示了如果驱动器已损坏，是 X 品牌的概率。图 4.8 显示了驱动器损坏问题的判定树，方形表示**行为** (act) 或判定，圆形表示事件。先验概率是那些在判断损坏的试验之前决定的。后验或逆概率是那些在试验完成之后决定的。后验概率允许我们去修正先验概率以获得更精确的结果。

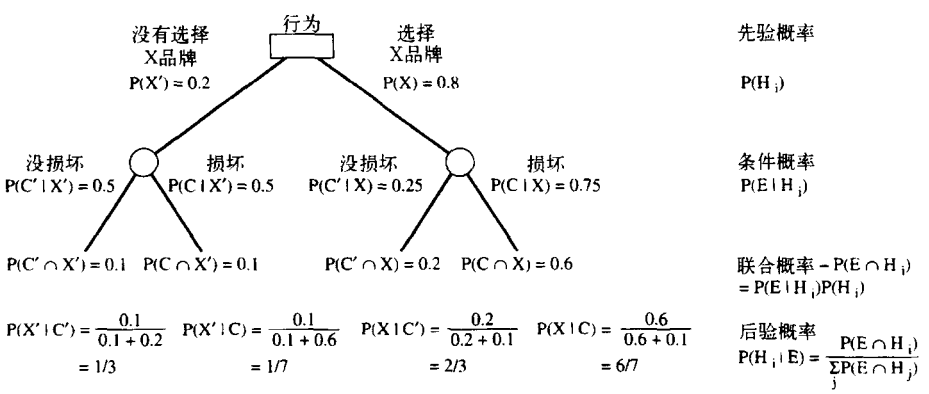


图 4.8 磁盘损坏的判定树

贝叶斯定理的一般形式可以用一组事件 E 和假设 H 以下列形式之一表示：

$$P(H_i \mid E) = \frac{P(E \cap H_i)}{\sum_j P(E \cap H_j)}$$

$$= \frac{P(E \mid H_i)\,P(H_i)}{\sum_j P(E \mid H_j)\,P(H_j)}$$

$$= \frac{P(E \mid H_i)\,P(H_i)}{P(E)}$$

4.9 假设推理与反向归纳

贝叶斯定理经常在商业和社会科学的判定树分析中使用。**贝叶斯决策过程**（Bayesian decision making）也应用于 PROSPECTOR 专家系统以确定矿产探测的最佳位置。作为发现价值 $ 100 000 000 的钼的第一个专家系统，PROSPECTOR 久负盛名。

作为在不确定环境下贝叶斯决策过程的一个例子，让我们看一看石油探测问题。首先，PROSPECTOR 必须决定发现石油的可能性，如果没有迹象表明是否有石油，PROSPECTOR 将选定石油的主观先验概率：

$$P(O) = P(O') = 0.5$$

在没有任何迹象的情况下，把概率平均分配给可能的结论称为**绝望**（in desperation）做法。术语绝望并不意味着 PROSPECTOR 处于绝望状况。它是一个技术术语，表示无偏见地先验分配概率。设 PROSPECTOR 相信有比 50-50 更高的可能性找到石油。假设如下：

$$P(O) = 0.6 \quad P(O') = 0.4$$

一个在石油和矿产探测中非常重要的工具是**地震测量**（seismic survey）。这种技术使用炸药或机器产生声音振动并在地球中传播。声波被不同位置的扩音器接收。通过观察振动的到达时间和声波格式的失真，就可能知道地质结构和含有石油、矿产的可能性。遗憾的是地震测试并不是百分之百准确。声波可能被某种类型的地质结构影响，导致测试报告说有石油，但实际上没有（错误肯定）。同样的，测试可能报告石油不在某处而事实上该处有石油（错误否定）。假设地震测试的过去历史已给定了下列的条件概率，条件概率中“+”是一个肯定结论，“-”是一个否定结论。注意这些是条件概率，因为起因（有油或没油）肯定已在结果（测试结果）之前发生。后验概率将从结果（测试结果）返回起因（有油或没油）。一般的，一个条件概率在时间上是正向的而后验概率在时间上是反向的。

$P(+ \mid O) = 0.8$　　$P(- \mid O) = 0.2$(错误否定)

$P(+ \mid O') = 0.1$(错误肯定)　　$P(- \mid O') = 0.9$

使用先验和条件概率，我们可以构造出如图 4.9 所示的初始概率树。图中同时还列出了联合概率，它可通过先验和条件概率计算得到。

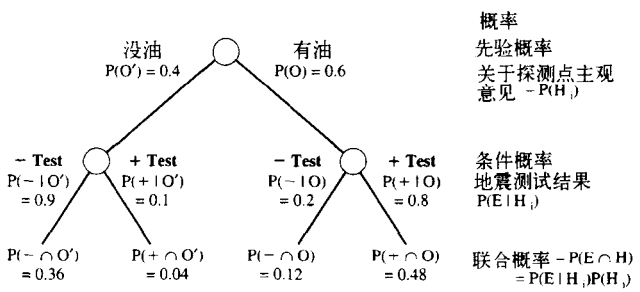


图 4.9　石油探测的初始概率树

然后，可用加法律来计算肯定和否定的总概率。

$$P(+) = P(+ \cap O) + P(+ \cap O') = 0.48 + 0.04 = 0.52$$
$$P(-) = P(- \cap O) + P(- \cap O') = 0.12 + 0.36 = 0.48$$

P（+）和 P（-）不是条件概率，现在可用它来计算该探测点的后验概率，如图 4.10 所示。例如，P（O′ | -）是在该点的基于否定测试的没油的后验概率。接着，联合概率也可计算出来。注意图 4.9 和图 4.10 的联合概率是一样的。在得到初始概率估计（或猜测）之后的试验信息，如地震测试结果后，要获得好的结果，概率的修正就非常必要了。

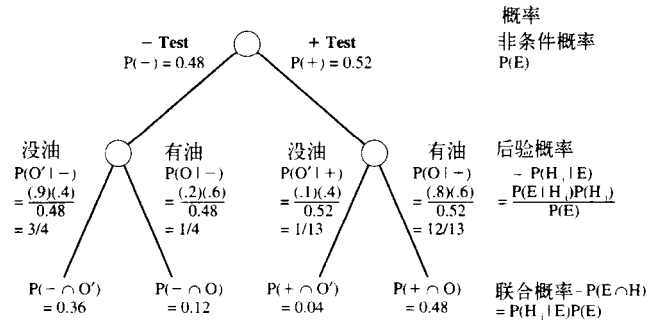


图 4.10 石油探测的修正概率树

图 4.11 显示的是使用了图 4.10 中数据的初始贝叶斯判定树。如果赚钱的话，树中最底的**收益**（payoff）是正的，反之是负的。假设的金额见表 4.9。

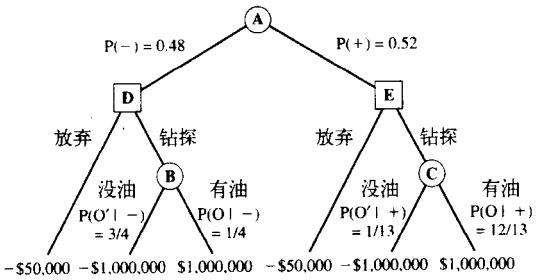


表 4.9 石油探测问题的收益表

如果成功，石油的收益	$ 1 000 000
钻探费用	- $ 200 000
地震测试	- $ 50 000

图 4.11 石油探测的初始贝叶斯判定树

因此，如果发现石油，收益是 $ 1 000 000 - $ 200 000 - $ 50 000 = $ 750 000，然而如果地震测试结果给出 - $ 50 000 的收益，则应放弃该决策。而没有找到石油的地震测试和钻孔花费是 - $ 200 000 - $ 50 000 = - $ 250 000。

为了使 PROSPECTOR 做出最好的决策，必须在事件结点 A 处计算**期望收益**（expected payoff）。期望收益是 PROSPECTOR 沿着最好的行动路线能够得到的收益的总数。在开始结点 A 计算期望收益，我们必须从叶子结点往回推。在概率理论中，这个过程称为**反向归纳**（backward induction）。也就是说，为了得到期望收益或达到我们的目的，我们必须往回推导，去寻找引导我们达到目的的原因。

一个事件结点的期望收益是每个收益乘以导致该收益的概率后的总和。

结点 C 的期望收益：

$ 673 077 = ($ 750 000) * (12/13) - ($ 250 000) * (1/13)

结点 B 的期望收益：

$ 0 = ($ 750 000) * (1/4) - ($ 250 000) * (3/4)

在行为结点 E，我们必须在期望收益 - $ 50 000（放弃）或者 $ 846 153（钻探）之间做出决策。显然 $ 846 153 大于 - $ 50 000，我们认为这是较好的决策并把它写在结点 E 上面。放弃的路径将用 = 来**删除**（prune）或**中止**（poisoned）以表示该路径将不再继续，如图 4.12 所示。同样的，在行为结点 D，在收益 - $ 500 000 和 - $ 50 000 中最好的选择是 - $ 50 000，因此它被写在 D 上。

最后，开始处的期望收益将被计算出来，如下：

结点 A 的期望收益

$ 350 000 = ($ 673 077) * (0.52) - ($ 0) * (0.48)

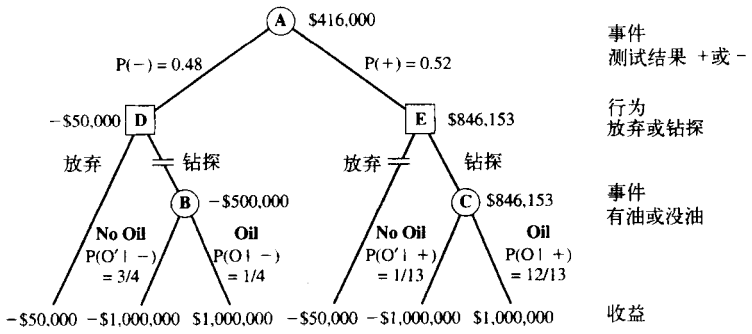


图 4.12 使用反向归纳的完整的石油探测贝叶斯判定树

该值被写在结点 A 上，在结点 A 没有路径能被删除，因为我们已经决定去做地震测试。如果要在地震测试上做决策，则判定树将继续回推到结点 A 之前。注意在反向归纳中，我们及时回推是为了使将来的行动是最优的。

判定树是假设推理或“如果……怎么样”情形的一个例子。通过探究行为的任一路径，我们可以删除不能得到最优收益的路径。某些类型的专家系统工具和贝叶斯软件就设计了假设推理和终止机制。关于更多的细节，可参看附录 G 中列出的软件资源。

特别是，贝叶斯工具既可免费也可从商业资源上得到，这些工具使得我们可以很容易地以图形方式根据不同的“如果……怎么样”情形来构建贝叶斯和概率树。电子报表如 Microsoft Excel 中有很多详尽的方法来处理不确定性，并且除了 Excel 自带的经典概率和统计函数外，你还可以从网上找到大量的宏。电子报表方便的图像功能也使得复杂数据可视化变得简单。

图 4.12 的判定树给出了 PROSPECTOR 的最优策略。如果地震测试是肯定结果，则该位置可以钻探。反之，则应该放弃。虽然这是贝叶斯决策过程的一个非常简单的例子，但是它确实说明了包括处理不确定情形的推理类型。在更复杂的事件中如决定是否做地震测试，判定树可能会非常庞大。

4.10 时间推理与马尔可夫链

依赖于时间的事件推理称为**时间推理**（temporal reasoning），它对人来说非常容易。但是，却难以把**时间事件**（temporal event）形式化以便计算机来进行推理。然而能对时间事件如航空交通控制进行推理的专家系统是非常有用的。在医学上，人们已经开发出了能进行时间推理的专家系统，如对帮助病人呼吸的人工呼吸机进行通风管理的 VM 系统，医治青光眼的 CASNET 系统和心脏病病人的数字化治疗指导系统。

除了 VM，上面提到的其他医学系统相对于必须**实时**（real time）操作的航空控制系统而言，在时间推理方面容易一些。由于推理机的设计和大量处理过程的需要，大多数专家系统不能实时操作。建立一个能实时进行大量时间推理以探究多种假设的专家系统非常困难。基于不同公理人们已提出了不同的时序逻辑。不同的理论针对不同的问题。时间是否有开始和最后时刻？时间是连续还是离散的？是否只有一个过去而有许多可能的将来？

根据如何回答这些问题，可以提出许多不同的逻辑。时序逻辑在传统程序上也非常有用，如在并行程序的合成与同步过程中。

另一个处理时间推理的方法是概率。我们可以认为时间的推移就是系统从一个状态发展到另一个状态。这个系统可以是任何可能的东西如股票、选票、天气、商业、疾病、设备、遗传学等。如果它是概率化的，则该系统通过一系列状态的发展称为**随机过程**（stochastic process）。

用一个转换矩阵的形式来表示一个随机过程是非常方便的。对于有两个状态 S_1 和 S_2 的简单事件，转换矩阵为：

$$\text{现在}\begin{matrix}S_1\\S_2\end{matrix}\begin{bmatrix}P_{11} & P_{12}\\ P_{21} & P_{22}\end{bmatrix}\quad\begin{matrix}\text{将来}\\ S_1 \quad S_2\end{matrix}$$

其中 P_{mn} 是指从状态 m 转换到状态 n 的概率。

作为一个例子，假设所有现在用 X 品牌驱动器的人中有 10% 的人会在需要的时候购买另一个 X 品牌的驱动器。同样，假设现在没有使用 X 品牌驱动器的人中有 60% 的人会在需要一个新驱动器时购买 X 品牌的驱动器（X 品牌驱动器的唯一优点是它的广告）。经过一段长的时间之后，有多少人会使用 X 品牌的驱动器呢？

转换矩阵 T 为：

$$T = \begin{matrix}X\\X'\end{matrix}\begin{bmatrix}0.1 & 0.9\\ 0.6 & 0.4\end{bmatrix}\quad\begin{matrix}X & X'\end{matrix}$$

其中每个行之和必须为 1。各元素非负且和为 1 的向量称为**概率向量**（probability vector）。T 的每一行就是一个概率向量。解释转换矩阵的一个方法是用如图 4.13 所示的图表。注意状态 X 到自身的概率是 0.1，状态 X′到自身的概率是 0.4，从 X 到 X′的概率为 0.9，从 X′到 X 的概率为 0.6。

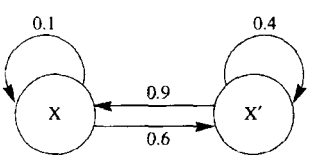
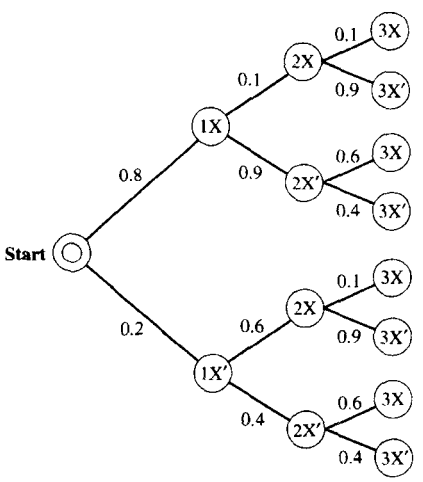
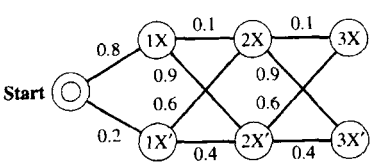


图 4.13 转换矩阵的状态图解释

假设初始有 80% 的人使用 X 品牌。图 4.14（a）显示了部分状态转换的概率树，其中状态用状态号和驱动器品牌标出。注意该树是如何生成的，如有 10 个转换，则树将有 $2^{10}=1024$ 个分支。另一种画树的方法是采用格，如图 4.14（b）所示。格表示法的好处在于它不需要用许多链来连接状态。



(a) 状态的树图



(b) 状态的格图

图 4.14 状态随时间变化的树和格图

在确定状态下系统的概率可以用一个称作**状态矩阵**（state matrix）的行矩阵来表示。

$$S = [P_1 \quad P_2 \quad \cdots \quad P_N]$$

其中 $P_1 + P_2 + \cdots + P_N = 1$；

开始时，八成的人拥有 X 品牌的商品，状态矩阵为：

$$S_1 = [0.8 \quad 0.2]$$

随着时间的推移，这个数字会根据人们购买哪个品牌而改变。

为了计算在状态 2 时拥有 X 品牌和不拥有 X 品牌的人数，我们只需要依据矩阵乘法规则用状态矩阵乘上转换矩阵就可以了：

$$S_2 = S_1 \ T$$

得到

$$S_2 = [0.8 \quad 0.2] \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix}$$

$$= [(0.8)(0.1) + (0.2)(0.6) \quad (0.8)(0.9) + (0.2)(0.4)]$$

$$= [0.2 \quad 0.8]$$

用转换矩阵乘以第二个状态可以得到：

$$S_3 = S_2 \ T$$

$$= [0.2 \quad 0.8] \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix}$$

$$S_3 = [0.5 \quad 0.5]$$

用转换矩阵乘以第三个状态可以得到：

$$S_4 = S_3 \ T$$

$$= [0.5 \quad 0.5] \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix}$$

$$S_4 = [0.35 \quad 0.65]$$

以下的状态是：

$$S_5 = [0.425 \quad 0.575]$$

$$S_6 = [0.3875 \quad 0.6125]$$

$$S_7 = [0.40625 \quad 0.59375]$$

$$S_8 = [0.396875 \quad 0.602125]$$

注意到这些状态收敛于：

$$[0.4 \quad 0.6]$$

这称为一个**稳定状态矩阵**（steady-state matrix）。当系统处于稳定状态时称它是平衡的，因为它不再改变。有趣的是稳定状态并不依赖于初始状态。如果使用其他任何初始概率向量，会得到同样的稳定状态值。

一个概率向量 S 是一个对于转换矩阵 T 的稳定状态矩阵，如果

$$(1) \quad S = S \ T$$

如果 T 是一个**正规转换矩阵**（regular transition matrix），即它只有正的元素，则存在一个唯一的稳定状态 S。转换矩阵的元素是正的这个事实表明，无论初始状态如何，在某些时候，达到任何状态都是可能的。也就是说，每个状态都是有可能达到的。

一个**马尔可夫链过程**（Markov chain process）定义为具有以下的特征：

(1) 有限个可能的状态。

(2) 过程在一个时刻有且仅有一种状态。

(3) 过程随着时间从一个状态向另外一个状态变化或**逐步**（steps）趋近。

(4) 一个变化发生的概率仅依赖于当前的状态。

例如，假设给定一个有限状态集合 {A, B, C, D, E, F, G, H, I}，如果在H后面过程的状态是I，那么条件概率：

$$P(I \mid H) = P(I \mid H \cap G \cap F \cap E \cap D \cap C \cap B \cap A)$$

注意图4.14（b）的格图非常类似于一条链。

磁盘驱动器的例子就是一个马尔可夫链过程，稳定状态矩阵可应用等式（1）求得。假设某个任意向量S由X和Y组成，应用等式（1）计算如下：

$$[X \quad Y] \begin{bmatrix} 0.1 & 0.9 \\ 0.6 & 0.4 \end{bmatrix} = [X \quad Y]$$

左边相乘并令它的元素和右边对应的元素相等，则得到：

$$0.1X + 0.6Y = X$$
$$0.9X + 0.4Y = Y$$

这是一个依赖方程。用Y表示X有：

$$X = \frac{0.6}{0.9} Y = \frac{2}{3} Y$$

为了最终求出X和Y，我们将应用概率之和为1的事实，也就是说

$$X + Y = 1$$

则得

$$X = 1 - Y = \frac{2}{3} Y$$

解得

$$X = \frac{2}{5} \quad Y = \frac{3}{5}$$

于是稳定状态矩阵是

$$[0.4 \quad 0.6]$$

这就是我们试验数据的收敛值。

4.11 信任几率

到目前为止，我们已经讨论了作为理想系统中可重复事件测度的概率。然而，人类擅长于计算许多不可重复事件的概率，例如医疗上的诊断和矿产的探测，因为每个病人和矿产位置都是不同的。为了在这样的领域开发专家系统，我们必须扩大事件的范围以便处理命题，即一个为真或假的陈述。例如，一个可能的事件是：

"病人浑身长满红斑点"

而命题是：

"病人出麻疹"

设A是一个命题，条件概率

$$P(A \mid B)$$

在一般意义上是一个不必要的概率，如果事件和命题不可重复或者没有一个数学上的依据的话。相反地，P（A|B）可被解释为当B成立时A为真的**可信度**（degree of belief）。

如果P（A|B）=1，那么我们相信A当然是真的；如果P（A|B）=0。那么我们相信，A显然是假的，而其他值0<P（A|B）<1则表示我们不能完全确定A是真还是假。从统计学的角度来说，**假**

设（hypothesis）就是其真假依据某些**证据**（evidence）还不能确定的命题。于是，使用条件概率来表示**似然性**（likelihood），如 P（H|E），就是表示在某种证据 E 的基础上，假设 H 的似然性。

虽然 P（H|E）有条件概率的形式，它实际上意味着某些不同的意思——似然性和可信度。概率适用于重复事件，而似然性适用于表示非重复事件中信任的程度。由于专家系统是人类专家的模拟，P（H|E）一般是在有某些证据 E 的情况下，专家对某种假设为真的可信度。当然，若事件是重复的，那么，P（H|E）就仅仅是概率而已。

如果我们同意 P（H|E）意味着似然性或可信度，那么，50%或 95%这样的数字又意味什么呢？例如，假设你有 95%的信心你的汽车下次能启动。一种表达这种似然性的方式，是用打赌中的**几率**（odds）。在某事件 C 的前提下，A 相对于 B 的几率是：

$$\text{odds} = \frac{P(A \mid C)}{P(B \mid C)}$$

若 B=A′，则有：

$$\text{odds} = \frac{P(A \mid C)}{P(A' \mid C)} = \frac{P(A \mid C)}{1 - P(A \mid C)}$$

定义

$$P = P(A \mid C)$$

则有：

$$\text{odds} = \frac{P}{1-P} \quad 及 \quad P = \frac{\text{odds}}{1 + \text{odds}}$$

用赌博中几率的话来说，我们可把 P 解释为赢的可能性，而 1－P 是输的可能性。于是：

$$\text{odds} = \frac{\text{赢}}{\text{输}}$$

已知几率的话就可计算概率或似然性，反之亦然。

于是，P=95%的似然性相当于：

$$\text{odds} = \frac{0.95}{1-0.95} = 19 \ : \ 1$$

即你相信汽车会启动。也就是说，你应该**不在乎**（indifferent）以 19:1 的几率打赌汽车会启动。如果有人出 1 美元赌车不会发动，而它又真的不动的话，你将要赔 19 美元。无论什么时候，当一个可信度用概率表示时，你都能用打赌中等价的几率来解释它。换句话说，你应该不在乎现实环境中是用可信度还是打赌中等价的几率。

概率通常与演绎问题一起使用，即在相同的假设下，一系列不同事件 E_i 均可能发生的问题。例如，一个骰子掷一个偶数，有三种可能的事件：

P(2|偶数)
P(4|偶数)
P(6|偶数)

在概率上，我们一般感兴趣于 P（E_i|H），其中 E_i 是根据一个共同假设得出的可能事件。在统计和归纳推理中，我们知道已经发生的事件但想找出能导致 E 的假设的似然性，即 P（E|H_i）。概率本质上是正向链或演绎的，而似然性则是反向链和归纳的。虽然我们对概率和似然性使用同样的符号 P（X|Y）表示，但应用却不同。通常，我们是指一种假设下的似然性或一个事件的概率。

基于可信度已发展起来的一个理论是**个人概率**（personal probability）论。在个人概率论中，**状态**（state）是可能的假设，而**后果**（consequence）是基于信念的行动的结果。

4.12 充分性与必然性

贝叶斯定理是：

$$(1)\ P(H \mid E) = \frac{P(E \mid H)P(H)}{P(E)}$$

对于H的否定，则为

$$(2)\ P(H' \mid E) = \frac{P(E \mid H')P(H')}{P(E)}$$

用（1）除以（2）得到：

$$(3)\ \frac{P(H \mid E)}{P(H' \mid E)} = \frac{P(E \mid H)P(H)}{P(E \mid H')P(H')}$$

定义H的先验几率为：

$$O(H) = \frac{P(H)}{P(H')}$$

后验几率为：

$$O(H \mid E) = \frac{P(H \mid E)}{P(H' \mid E)}$$

最后定义**似然率**（likelihood ratio）为：

$$(4)\ LS = \frac{P(E \mid H)}{P(E \mid H')}$$

于是（3）变成：

$$(5)\ O(H \mid E) = LS\ O(H)$$

等式（5）是Bayer定理的**几率似然性形式**（odds-likelihood form）。这种Bayes定理的形式比等式（1）使用起来更为方便。

因子LS也称为**充分似然性**（likelihood of sufficiency），因为若LS=∞，则证据E对于得出H为真是逻辑充分的。如果E对H逻辑充分，则P（H|E）=1且P（H|E′）=0。等式（5）可用来求解LS：

$$(6)\ LS = \frac{O(H \mid E)}{O(H)} = \frac{\frac{P(H \mid E)}{P(H' \mid E)}}{\frac{P(H)}{P(H')}}$$

此时，P（H）/P（H′）是某个常数C，于是等式（6）化成：

$$LS = \frac{\frac{1}{0}}{C} = \infty$$

等式（4）还表明在这种情况下，E对于H是充分的。表4.10列出了LS其他值的意义。

类似于LS，可定义**必然似然性**（likelihood of necessity）LN：

$$(7)\ LN = \frac{O(H \mid E')}{O(H)} = \frac{P(E' \mid H)}{P(E' \mid H')} = \frac{\frac{P(H \mid E')}{P(H' \mid E')}}{\frac{P(H)}{P(H')}}$$

$$(8)\ O(H \mid E') = LN\ O(H)$$

如果LN=0，那么P（H|E′）=0。这表明当E′为真时H必为假。也就是如果E不存在则H为假，即E对H来说是必然的。

表4.11列出了LN，E和H之间的关系。注意，除了“证据”被“无证据”所替代外，它和表4.10是完全一样的。

似然值LS和LN必须由专家提供，以便计算后验几率。等式（5）和（8）非常易于人们理解。LS因子表明当证据存在时，先验几率的改变，LN因子则表明当证据不存在时，先验几率的改变。这种形式使得专家更容易指定LS和LN因子。

表 4.10 似然率、假设和证据之间的关系

LS	对假设的影响
0	当 E 为真时 H 为假 或 E′对 H 是必然的
小 (0<LS≪1)	E 对得出 H 是不利的
1	E 对 H 无影响
大 (1≪LS)	E 对得出 H 是有利的
∞	E 对 H 是逻辑充分的 或观测到 E 意味着 H 必为真

表 4.11 必然似然性、假设和证据之间的关系

LN	对假设的影响
0	当 E 不存在时 H 为假 或 E 对 H 是必然的
小 (0<LN≪1)	E 不存在对得出 H 是不利的
1	E 不存在对 H 无影响
大 (1≪LN)	E 不存在对 H 是有利的
∞	E 不存在对 H 是逻辑充分的

作为一个例子，在 PROSPECTOR 专家系统里有一个规则具体说明了特定矿藏的证据是如何支持假设的：

IF 有石英硫矿带
THEN 必有钾带顺迁

这个特别的中间假设为其他的假设提供了支持，而这些假设导致了铜矿存在的顶层假设。对于这条规则来说，LS 和 LN 的值是：

```
LS = 300
LN = 0.2
```

这意味着观测到石英硫矿带非常有用，而若不能观测到硫矿带则没有什么意义。如果 LN≪1，那么缺乏硫矿带将强烈表明假设是错误的。一个例子是以下这条规则：

IF 有玻璃褐铁矿
THEN 有最佳的矿产结构

其中

```
LS = 1000000
LN = 0.01
```

4.13 推论链中的不确定性

不确定性可能出现在规则或规则所使用的证据中，或同时出现在两者之中。在本节，你将会看到现实世界中对具有不确定性的某些问题以及概率是如何给出解答的。

专家的不一致性

从前面小节的等式 (4) 可知：

如果 LS>1 则 P(E|H′)< P(E|H)

两边同时用 1 来减，得到一个反向的不等式：

$$1 - P(E \mid H') > 1 - P(E \mid H)$$

由于 P (E′ |H) =1−P (E|H)，且 P (E′|H′) =1−P (E|H′)，所以 (7) 变成：

$$LN = \frac{1 - P(E \mid H)}{1 - P(E \mid H')} < 1$$

对值 LN 和 LS 的规定可归结为如下几种情形：

情形 1：LS>1 且 LN<1

从等式 (4) 和 (7) 可得其他情形：

情形 2：LS<1 且 LN>1

情形3：LS=LN=1

虽然这三种情形在数学意义上严格限制了LS和LN的取值范围，然而这些情形并非总能在现实世界中有用。对于用作矿产探测的专家系统PROSPECTOR来说，专家们指定LS>1且LN=1并不少见。这不是以上三种情形中的一种，因为专家认为对证据的观察很重要，而缺少证据则并不重要。

上面的情形表明这种基于贝叶斯概率理论的似然性理论对矿产探测问题来说是不完全的。也就是说，贝叶斯似然性理论只是能处理LS>1且LN=1情形理论的一种近似。对于专家意见符合前三种情形的领域来说，贝叶斯似然性理论是适用的。

不确定性证据

在现实世界中，除了死亡和税收以外，我们很少能绝对确定某个事物。虽然到目前为止，我们已集中讨论了不确定性假设，但是更为一般和现实的情形是不确定性假设以及不确定性证据。

一般地，假定**全证据**（complete evidence）E的可信度依赖于**部分证据**（partial evidence）e，表示为：

P(E|e)

全证据就是所有证据，即所有可能的证据和假设，它们组成E。部分证据e是我们所知的E的一部分。如果我们知道所有证据，则E=e且

P(E|e)=P(E)

其中P（E）是证据E的先验似然性。而似然性P（E|e）则是已知全证据E中不完整的知识e后，我们对E的信任。

例如，假设你的邻居因在他们的厨房底下发石油而变得富了起来，你可能会考虑下面假设和证据：

E = 我将富起来

H = 我的厨房底下有石油

用规则表达就是：

IF 我的厨房底下有石油

THEN 我将富起来　　P(E|H)=1

开始时，你并不确切知道是否有石油在你厨房底下。确实的证据是钻一个测试井，但这很费钱。于是你考虑如下支持H的部分证据e：

- 邻居已经因此富起来了。
- 经常有某些黑色物质从厨房的火炉边渗出来（直到现在，你的配偶还在抱怨你的烹饪和清洁）。
- 一个陌生人上门愿意出20 000 000美元买下你的房子，说他喜欢那种环境。

基于这些部分证据，你可能认为你厨房底下有石油的似然性相当高，P（E|e）=0.98。

在概率或似然性类型的推理链中，如H依赖于E，而E基于某些部分证据e，则P（H|e）是H依赖于e的似然性。根据条件概率的定义有：

$$P(H \mid e) = \frac{P(H \cap e)}{P(e)}$$

图4.15说明了H是如何由E和e构成的。于是上式变成：

$$P(H \mid e) = \frac{P(H \cap E \cap e) + P(H \cap \neg E \cap e)}{P(e)}$$

应用条件概率的定律有：

$$P(H \mid e) = \frac{P(H \cap E \mid e)P(e) + P(H \cap E' \mid e)P(e)}{P(e)}$$

$$P(H \mid e) = P(H \cap E \mid e) + P(H \cap \neg E \mid e)$$

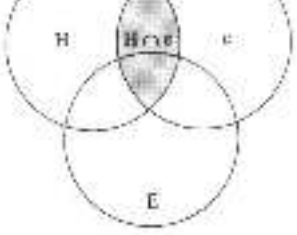


图4.15 H和e的交集

(1) 式也可用下面的一种形式来表达。因为，

$$P(H \cap E \mid e) = \frac{P(H \cap E \cap e)}{P(e)}$$

$$(2)\quad P(H \cap E \mid e) = \frac{P(H \mid E \cap e)\ P(E \cap e)}{P(e)}$$

又因为

$$P(E \mid e) = \frac{P(E \cap e)}{P(e)}$$

所以 (2) 化成

$$P(H \cap E \mid e) = P(H \mid E \cap e)\ P(E \mid e)$$

同样，

$$P(H \cap E' \mid e) = P(H \mid E' \cap e)\ P(E' \mid e)$$

于是，(1) 化成：

$$(3)\quad P(H \mid e) = P(H \mid E \cap e)\ P(E \mid e) + P(H \mid E' \cap e)\ P(E' \mid e)$$

通常，我们并不知道概率 P (H|E∩e) 和 P (H|E′∩e)，但是，如果我们假设这些都能用 P (H|E) 和 P (H|E′) 趋近，则 (3) 化简为：

$$(4)\quad P(H \mid e) = P(H \mid E)\ P(E \mid e) + P(H \mid E')\ P(E' \mid e)$$

等式 (4) 实质上是 P (H|e) 关于 P (E|e) 的一个线性插值表达式。端点是：

(ⅰ) E 为真，则 P(H|e) = P(H| E)；

(ⅱ) E 为假，则 P(H|e) = P(H| E′)。

当 P (E| e) 等于先验概率 P (E) 时，等式 (4) 的问题就显现出来。如果系统服从纯粹的贝叶斯概率，则

$$(5)\quad P(H \mid e) = P(H \mid E)\ P(E) + P(H \mid E')\ P(E')$$

$$(6)\quad P(H \mid e) = P(H)$$

哪一个正确？

然而，在现实世界中，经验表明专家们给出的主观概率几乎可以肯定是不一致的。例如，如果专家使用不一致情形 LS>1 且 LN=1，则

$$O(H \mid E') = LN\ O(H) = O(H)$$

由于

$$O = \frac{P}{1 - P}$$

所以

$$(7)\quad P(H \mid E') = P(H)$$

在 (5) 中应用 (7)，有：

$$P(H \mid e) = P(H \mid E)\ P(E) + P(H)\ P(E')$$

$$= P(H \mid E)\ P(E) + P(H)\ (1 - P(E))$$

$$(8)\quad P(H \mid e) = P(H) + P(H \mid E)\ P(E) - P(H)\ P(E)$$

此时，

$$O(H \mid E) = LS\ O(H)$$

如果 LS=1，则

```
O(H | E) = O(H)
P(H | E) = P(H)
```

由于专家已经说明 LS>1，所以 P (H|E) >P (H)，于是在等式 (8) 中，项

```
P(H | E) P(E) - P(H) P(E)
```

>0。其中，0 是当 LS=1 时的下界。于是由 (8)，当 LS>1 且 LN=1 时，

```
P(H | e) > P(H)
```

这与当 P (E|e) =P (E) 时，P (H | e) 应和 P (H) 相等的事实矛盾。由于 P (H|e) >P (H)，概率比它应有的要大，并且有可能在把它作为某一规则的推论应用到推理链中的另一规则时被进一步放大。

修正不确定性

修正上述问题的一个方法是，假设 P (H|e) 是一个分段线形函数。这是一个在 PROSPECTOR 中应用得很好但不是基于传统概率理论的特别假设。这个线形函数为了简化运算而选定。

该函数通过图 4.16 中的三点。P (H | e) 的计算使用以下线性插值式：

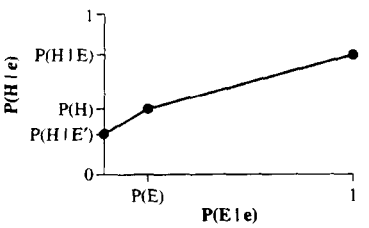


图 4.16 PROSPECTOR 中部分证据的分段线性插值函数

$$P(H \mid e) = \begin{cases} P(H \mid E') + \dfrac{P(H) - P(H \mid E')}{P(E)} P(E \mid e) \\ \qquad \text{for } 0 \le P(E \mid e) < P(E) \\ P(H) + \dfrac{P(H \mid E) - P(H)}{1 - P(E)} [P(E \mid e) - P(E)] \\ \qquad \text{for } P(E) \le P(E \mid e) \le 1 \end{cases}$$

使用该公式，先前讨论的 LS>1 且 LN=1 的不一致情形也能很好地符合。若 P (E | e) <P (E)，则 P (H|e) 的值不变，若 P (E|e) ≥P (E)，则 P (H| e) 的值增加。选择该分段函数而不选一条直线是为了保证当 P (E |e) =P (E) 时，P (H|e) =P (H)。

4.14 证据组合

最简单的规则是如下形式：

```
IF E THEN H
```

其中 E 是单个已知证据，由它我们可得出 H 为真。遗憾的是，不是所有的规则都如此简单，因此有必要对不确定性作补充说明。

不确定性证据的分类

如果规则具有不确定性，便会出现许多复杂的情形。根据我们处理的是可重复事件还是主观可能性，这些不确定性可用概率或者似然性来表达。为了简便，我们使用概率一词来表达这种不确定性。

根据证据是确定的还是不确定的、是**简单证据** (simple evidence) 还是**复合证据** (compound evidence)，可把各种不同的情形分类。简单证据由单个证据组成，例如，

```
IF 变速器坏
THEN 修车
```

而复杂证据则由多个证据组成，它们通常用 AND 连结，例如，

```
IF 变速器坏 AND
   发动机坏
```

THEN 卖车

或者形式地表示为：

IF E_1 and E_2 then H

我们可以为这个规则指定一个概率，例如 $P(H|E_1 \cap E_2) = 0.80$，它意味着基于这些证据，我们有 80% 的把握确信车子应被卖掉。

对证据的进一步精炼是确定其概率。例如，变速器坏了的概率依赖于下面两种征兆：

(a) 变速器漏油

(b) 不能倒车

根据这些观测可得出变速器证据的概率。例如，基于征兆 (a) 与 (b)，可断定

$P(E|e) = 0.95$

其中 E = 变速器坏

e = 上述征兆(a)与(b)

现在，汇总各种不同情形，以便详尽地探讨概率。

情形 1：由一个已知证据得出 H

这是最简单的情形，规则具有形式：

IF E THEN H

在没有证据前，H 的先验概率为 P (H)。当知道证据后，H 的概率根据贝叶斯定理变为：

$$P(H \mid E) = \frac{P(H \cap E)}{P(E)} = \frac{P(H \cap E)}{P(E \cap H) + P(E \cap H')}$$

$$P(H \mid E) = \frac{P(H \mid E)P(H)}{P(E \mid H)P(H) + P(E \mid H')P(H')}$$

其中 P (E) 为 E 的观测概率。

情形 2：由两个已知证据得出 H

这种情形比情形 1 复杂，其规则形式为：

IF E_1 and E_2 then H

当观测到 E_1 与 E_2 之后，H 的概率由先验概率 P (H) 变为：

$$P(H \mid E_1 \cap E_2) = \frac{P(H \cap E_1 \cap E_2)}{P(E_1 \cap E_2)}$$

$$= \frac{P(H \cap E_1 \cap E_2)}{P(E_1 \cap E_2 \cap H) + P(E_1 \cap E_2 \cap H')}$$

(1) $P(H \mid E_1 \cap E_2)$

$$= \frac{P(E_1 \cap E_2 \mid H)\ P(H)}{P(E_1 \cap E_2 \mid H)\ P(H) + P(E_1 \cap E_2 \mid H')\ P(H')}$$

除非作出简化假设，否则上式将不能再化简。如果假定证据 E_1 与 E_2 彼此条件独立，则：

$P(E_1 \cap E_2 \mid H) = P(E_1 \mid H)\ P(E_2 \mid H)$
$P(E_1 \cap E_2 \mid H') = P(E_1 \mid H')\ P(E_2 \mid H')$

于是（参看习题 4.12 的条件独立问题）：

(2) $P(H \mid E_1 \cap E_2)$

$$= \frac{P(E_1 \mid H)\ P(E_2 \mid H)}{P(E_1 \mid H)\ P(E_2 \mid H) + O(H')\ P(E_1 \mid H')P(E_2 \mid H')}$$

式 (2) 是式 (1) 的简化，其概率都具有单一的概率形式，而不存在诸如 $P(E_1 \cap E_2|H')$ 的联合概率。这种条件独立假定会出现一些问题，稍后我们将讨论。

式（2）虽已简化，但仍须知道先验概率 P（E_1）与 P（E_2）。指定先验概率对专家来说往往很困难，因为他们不用这种方式推理。比如，某人去看医生，医生不能因为感冒是常见病，发病率高而假设那人患感冒的先验概率。

然而指定先验概率的主要问题是难以确定其似然性。对于诸如掷骰子之类的可重复事件，当然很容易通过经验或理论研究得到其概率。但当处理寻找矿藏之类的事件时，便不可能准确地确定其似然性。比如，你的房子下有大金矿的先验似然性是多少？是 0.0000001，0.000 00001，还是其他数值？这并不是先验概率不能确定的唯一情形。

情形 3：由 N 个不确定性证据得出 H

这是证据以及依赖于证据的规则中具有不确定性的一般情形。随着证据数目的增加，将不可能确定所有复合和先验概率（或似然性）。对 N 个证据的一般情形，人们使用了不同的近似方法来计算。

通过模糊逻辑组合证据

证据的合取

设有规则

```
IF E THEN H
```

其中 E 为证据的合取，即

```
IF E1 AND E2 AND ... EN THEN H
```

要使前件为真，所有的 E_i 都必须以某个概率为真。在一般情况下，每个证据都基于部分证据 e，证据的概率为：

$$P(E \mid e) = P(E_1 \cap E_2 \cap \ldots E_N \mid e) = \frac{P(E_1 \cap E_2 \cap \ldots E_N \cap e)}{P(e)}$$

若证据 E_i 全部条件独立，则联合概率变成单独概率的积，这可由一般的乘法律推得。以两个证据为例：

$$P(E_1 \cap E_2 \mid e) = \frac{P(E_1 \cap E_2 \cap e)}{P(e)} = \frac{P(E_2 \mid E_1 \cap e)\ P(E_1 \mid e)\ P(e)}{P(e)}$$

使用独立性假设，

$$P(E_1 \mid e) = P(E_2 \mid E_1 \cap e)$$

由于 E_1 与 E_2 无关，得

$$P(E_1 \cap E_1 \mid e) = P(E_2 \mid e)\ P(E_1 \mid e)$$

一般，

$$P(E_1 \cap E_2 \cap \ldots E_N \mid e) = \prod_{i=1}^{N} P(E_i \mid e)$$

虽然理论上该式是正确的，但应用到现实问题上却有两个困难。首先，在现实世界中单独概率 P（E_i|e）往往并不独立。其次，各因子的乘积通常也比 P（E|e）小很多。

该问题的一个近似解法是用**模糊逻辑**（fuzzy logic）来计算 P（E|e）：

```
P(E | e) = min [P(Ei | e)]
```

其中 **min 函数**取各个 P（E_i|e）中最小值。在 PROSPECTOR 系统中，上式取得了令人满意的效果。而且一旦 P（E|e）确定，便可用分段线性公式计算 P（H|e）了。

模糊逻辑公式的主要问题是：除了 P（E_i|e）的最小值之外，概率 P（E|e）与其他的 P（E_i|e）无关。这意味着只要最小值不变，即使其他所有的 P（E_i|e）增大，也不能使 P（E|e）发生变化。即其

他 $P(E_i|e)$ 通过推理链所传播的变化，均被最小 $P(E_i|e)$ 所堵住。而该式的好处，则是计算简便。

证据的析取

若规则是证据的析取：

IF E_1 OR E_2 OR ... E_N THEN H

则当假设每个 E 独立之后，有（参看习题 4.13）

$$P(E \mid e) = 1 - \prod_{i=1}^{N} [1 - P(E_i \mid e)]$$

该式的问题是计算出的概率太高。在 PROSPECTOR 中基于模糊逻辑改为：

$$P(E \mid e) = \max [P(E_i \mid e)]$$

其中 max 函数取 $P(E_i|e)$ 中最大值。

证据的逻辑组合

若规则前件是证据的逻辑组合，则可用模糊逻辑与否定法则来计算。例如

IF E_1 AND (E_2 OR E_3') THEN H

则

$$E = E_1 \text{ AND } (E_2 \text{ OR } E_3')$$
$$E = \min \{P(E_1 \mid e), \max [P(E_2 \mid e), 1 - P(E_3 \mid e)]\}$$

虽然这些模糊逻辑公式已在许多系统上成功使用，但也可为组合证据定义其他函数。比如，下面就是代替析取 max 函数的另一种定义：

$$P(E_1 \cup E_2 \mid H) = \min [1, P(E_1 \mid H) + P(E_2 \mid H)]$$

有效似然性

一般情况下，具有不确定证据与不一致先验概率的复杂规则可能会得出一个特殊假设。假定条件独立性成立，且所有对 H 起作用的证据 E_i 都为真，则有：

$$O(H \mid E_1 \cap E_2 \cap \ldots E_N) = \left[\prod_{i=1}^{N} LS_i\right] O(H)$$

其中

$$LSi = \frac{P(E_i \mid H)}{P(E_i \mid H')}$$

如果所有对 H 起作用的证据都为假，则有另一相似式子：

$$O(H \mid E_1' \cap E_2' \cap \ldots E_N') = \left[\prod_{i=1}^{N} LN_i\right] O(H)$$

其中

$$LNi = \frac{P(E_i' \mid H)}{P(E_i' \mid H')}$$

在具有不一致先验概率与不确定证据的一般情况下，**有效似然率**（effective likelihood ratio）LE 定义为

$$LE_i = \frac{O(H \mid e_i)}{O(H)}$$

其中 e_i 为第 i 个对 H 起作用的部分证据。基于不确定与不一致证据，类似前面情况可修改 H 为：

对于一个使用不确定证据与不一致先验概率的专家系统来说，上式可采用以下方式使用：

$$O(H \mid e_1 \cap e_2 \cap \ldots e_N) = \left[\prod_{i=1}^{N} LE_i\right] O(H)$$

(a) 存储每条规则的先验几率以及每个对规则起作用的 LE_i。

(b) 每当 $P(E_i|e_i)$ 被更新时，计算新的 LE_i 及后验几率。

条件独立性的困难

尽管条件独立性假设对简化贝叶斯定理有用，但却存在一些问题。在专家系统建造的初始阶段，该假设是有用的，因为此时系统的一般行为要比正确的数值结果重要许多。因此，在开始，知识工程师可能更感兴趣于建立正确的推理链而不是正确的数值结果。也即，在某些系统中将会出现，中间假设 10 由证据 23 与 34 激活，假设 10 与证据 8 又可激活假设 52 与 96，假设 15 不能由证据 23 与 24 激活，等等。

正如本节等式（2）所示的那样，在条件独立性下，计算 $P(H \mid E_1 \cap E_2)$ 只需 5 个概率，它们是：

$$P(E_1 \mid H),\ P(E_2 \mid H),\ P(E_1 \mid H'),\ P(E_2 \mid H'),\ P(H)$$

而计算 $O(H')$ 所需的 $P(H')$ 则不需单独列出。这是因为，

$$P(H') = 1 - P(H)$$

在习题 4.12（b）中给出了另一个计算 $P(H|E_1 \cap E_2)$ 的公式，它涉及另外 4 个概率：

$$P(H \mid E_1),\ P(H \mid E_2),\ P(H' \mid E_1),\ P(H' \mid E_2)$$

因此总共有 9 个概率可用于计算 $P(H|E_1 \cap E_2)$，但是实际上只须指定其中 5 个，给出 5 个后剩下的 4 个可由其导出。

这种对独立概率个数的要求在专家系统开发后期将成为一个实际的问题。一旦推理链功能正确了，知识工程师必须确信能给出正确的数值输出。而由于条件独立性假设，知识工程师不能完全自由地调整 9 个概率值，以使其输出期望结果。

条件独立性假设决定了联合概率 $P(E_1 \cap E_2)$，这是因为，

$$P(E_1 \cap E_2) =$$

$$P(E_1)P(E_2)\left[\frac{P(H \mid E_1)P(H \mid E_2)}{P(H)} + \frac{P(H' \mid E_1)P(H' \mid E_2)}{P(H')}\right]$$

这意味着先验概率 $P(E_1)$、$P(E_2)$、$P(H)$ 已限定了联合概率 $P(E_1 \cap E_2)$，这与人类专家分别得知 $P(E_1)$，$P(E_2)$，$P(E_1 \cap E_2)$ 相矛盾。因此，条件独立性限制了人类专家所有知识的应用。

尽管条件独立性假设看起来是合理的，但它毕竟依赖于专家系统所模拟的现实情况，因而不是在所有情形下都成立。此前已有理论宣称条件独立性往往是错误的，另一理论则宣称条件独立性暗含严格独立的意思，而这就证明证据与假设的更新无关！不过，这一论断已被另一理论所推翻。

另一应用主观贝叶斯概率的方法显示了条件独立性假设可被削弱，以致不再是严格的独立。

4.15 推理网

到目前为止，你所见的正向和反向推理链例子都很小，只包含几条规则。但在现实世界问题中，支持一个假设或得出一个结论所需的推理数目却非常大，而且这些推理中的许多或全部都是在不确定性证据与规则下进行的。概率推理与贝叶斯定理已成功地运用在这类现实系统中。

对专家系统来说，推理网是依赖于知识分类学的一个好的结构。分类学是一种分类方法，它常常应用于诸如地理、生物之类的自然科学中，在第 2 章的不同种类黑莓问题分类图中，你已经见到了一个简单的分类法。

基于两个目的，分类学是有用的。在把对象分类并显示它们与其他对象之间的联系时，分类学有助于组织知识。一些重要特性如继承性可以通过分类而弄得很清楚。

分类学有用的另一个原因是它能指导证明一个假设的搜索，如“该位置有铜矿”的假设。由于地

球表面有矿藏的直接证据很少，所以在矿产探测这样的领域，分类学对证明或否定假设的帮助是非常重要的。在决定投资时间、金钱开始钻探之前，地质学家必须尽力搜集证据以支持其假设。

PROSPECTOR

使用概率推理的经典专家系统是 PROSPECTOR，它用来辅助采矿学家决定一个位置是否具有某种矿产的良好结构。其基本思想是把地质学家的专业实用知识编码成不同的矿产**模型**（model）。矿产模型是支持某种矿产在一位置的一组证据与假设。除了辅助识别矿产外，PROSPECTOR 还可建议最佳采矿点。随着更多的模型被创建，PROSPECTOR 的能力也随之增强。

每个模型的数据均以**推理网**（inference net）的形式组织。图 4.17 总结了曾讨论过的各种类型的网，且在图 4.17（d）中显示了一个很简单的推理网。推理网中的结点可表示证据，它支持其他表示假设，如存在矿产之类的结点。PROSPECTOR 有 22 种矿产模型，表 4.12 中列出了几种。

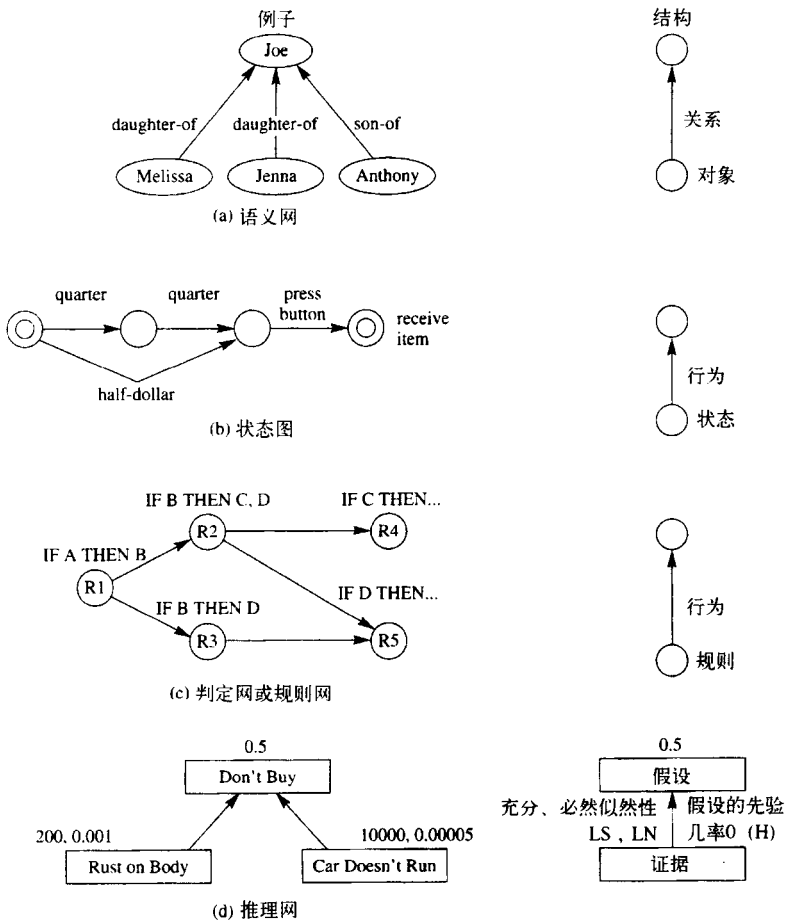


图 4.17 一些网络类型

表 4.12 某些 PROSPERCTOR 模型

模 型 名	模 型 描 述	结 点 数	可询问结点数	规 则 数
PCD	斑岩铜	200	97	135
RFU	前滚铀	185	147	169
SPB	砂石基质铅	35	21	32

在表4.12中，结点数是模型中所有结点的数目，**可询问结点**（askable nodes）是指那些询问用户以获取观测证据的结点，规则数是概率推理规则的数目。可见，模型中有很多不确定性，而概率在支持假设中起着重要作用。

推理网

PROSPECTOR中的每个模型均编码成一个证据与假设之间的联系或者关系网络，因此推理网就是一种语义网。观测事实，如由地层探测所获得的岩石结构等，组成了支持中间假设的证据，而多组中间假设又用来支持**顶层假设**（top-level hypothesis），即我们想要证明的。如果不是很有必要区分证据与假设的话，我们以术语**断言**（assertion）统称这两者。图4.18显示了斑岩铜模型中顶层假设的一小部分推理网。

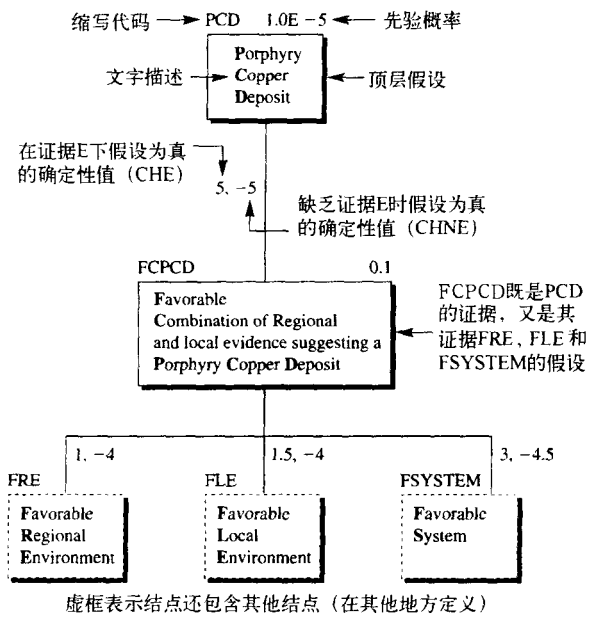


图4.18 使用确定性因子表示的PROSPECTOR顶层斑岩铜假设

经验表明，专家们难以指定先验概率或者似然率，因此，PROSPECTOR中使用了**确定性因子**（certainty factors）CHE与CHNE。在诊断血液疾病的MYCIN系统中也有类似情形，医生不习惯指定概率，于是就使用了确实性因子。与MYCIN一样，PROSPECTOR中确实性因子从-5到+5分为11个等级，-5表示“肯定不”，而+5表示“肯定是”。

由于PROSPECTOR采用了模糊逻辑与确实性因子来表示证据，所以它不是纯概率系统。在第5章，我们将详细讨论确实性因子与模糊逻辑。

图4.19更详细地显示了图4.18中的FRE结点。

该详细图中，每个结点上均有两个用逗号隔开之数，分别为似然率LS与必然性测度LN，比如左下方结点RCIB的LS，LN值为20，1。每个结点的缩写名代表其描述，如RCIB代表（该地带含角炼岩）the Region Contain Instrcitive Bseclas。在每个结点上还有一单个数值，它是先验概率，如对RCIB，该值为0.001。

推理关系

在推理网中，证据支持或否定假设用箭头表示。例如RCS、RCAD、RCIB、RCVP与SMIRA均支

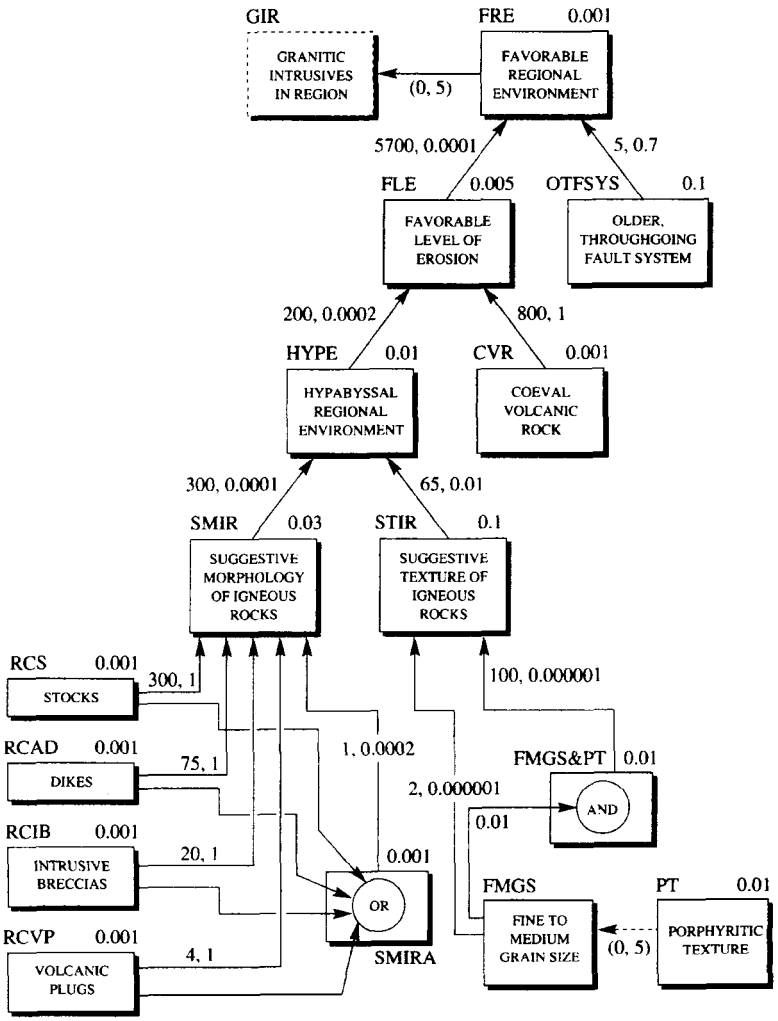


图 4.19 PROSPECTOR 中斑岩铜推理网的一小部分

持或否定中间假设 SMIR，而中间假设 SMIR 则是假设 HYPE 的证据，HYPE 又是 FLE 的证据，如此下去。

地质学家在模型定义中要求以两种一般方式来组合证据之间的关系：

- **逻辑组合**（logical combinations），如 AND 和 OR。如上节所说，在这种结点上可用模糊逻辑计算结果。
- **加权组合**（weighted combinations），用似然率 LS 与必然率 LN。当证据为真时，后验几率可通过

$$O(H \mid E) = LS\ O(H)$$

计算，当 E 为假时，可通过

$$O(H \mid E') = LN\ O(H)$$

计算。当 E 不能确定时，如前面小节所讨论的，可用线性插值法计算 P（H | E)。

术语**加权组合**（weighted combination）源于多个证据对假设起作用时的一般情形。如上节所述，

$$O(H \mid E_1 \cap E_2 \cap \ldots E_N) = \left[\prod_{i=1}^{N} LS_i\right] O(H)$$

取对数得：

$$\log O(H \mid E_1 \cap E_2 \cap \ldots \cap E_n) = \log O(H) + \sum_{i=1}^{n} \log LS_i$$

这可解释为每个 LS_i 通过 log LS 对假设“投票”，每个 log LS 都是影响假设的一个证据。

由于具有加权组合的规则形式为：

IF E_1 AND E_2 AND ... E_n THEN H

所以，PROSPECTOR 有时被称为基于规则的系统。但虽然如此，PROSPECTOR 却不如一个真正基于规则的产生式系统灵活。其局限之一是缺乏约束变量的完整机制。PROSPECTOR 是一个针对用户设计的系统，它强调效率与地质应用控制，而不注重产生式系统的一般性。

加权组合也是**合情关系**（plausible relations）的一个例子。术语合情指有部分证据用来证实可信性。PROSPECTOR 就是一个用合情推理来支持或否定假设的系统例子。PROSPECTOR 的合情推理基于贝叶斯概率，其 LS、LN 值由人类专家给出。

合情与其他不同的可信度如图 4.20 中的模糊图形所示。这些术语的一般意义在表 4.13 中说明。

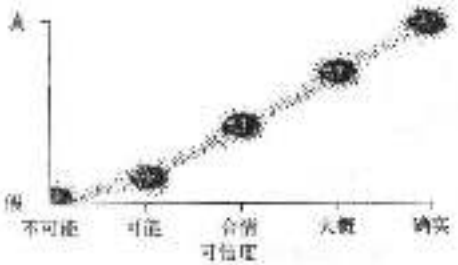


图 4.20 用于描述证据的一些术语的相关含义

表 4.13 某些用于事件的术语

术语	与假设相关的证据
不可能	断然否定
可能	不全然否定
合情	有些证据存在
大概	有证据表明
确实	完全肯定

图 4.20 的图形故意画得模糊，就是为了描述出这些术语的模糊面貌，以及从一个转化到另一个时的模糊性。注意图中我们对假设的信任是如何从不可能变到确实的。**确实信任**（certain belief）意味着真，而**不可能信任**（impossible belief）意味着假，没有一种不确定性能同时包含确实与不可能信任，它们相当于逻辑真与逻辑假。术语 certain evidence 有时含糊使用，它可逻辑真或假。也就是说，certain evidence 不具有不确定性，这意味 certain evidence 既可确实信任（逻辑真）也可不可能信任（逻辑假）。

可能信任（possible belief）意味着无论可能性如何小，也不能就此否定假设。例如，在月球表面的科学分析之前，可说月球是由青乳酪组成的，虽然可能性非常小，但还是可能的。由于没有明确证明，因此可能性仍然存在。

合情信任（plausible belief）意味着有点可能。术语合情常常用于法律场合表示合理而缺乏确证。因此，即使在科学研究之前，说月球是由青乳酪组成也是不合情的。

大概信任（probable belief）意味着，存在某些证据支持假设，但不足以完全证明这个假设。

在没有矿场证据的情况下，比如说，没有 RCIB，那么图 4.19 中矿产点的关系仅仅是合情的。随着证据的积累，这种合情关系可能变成大概，随后变成确实。

大概信任（probable belief）意味着，存在某些证据支持假设，但不足以完全证明这个假设。例如，如果你玩骰子一直在赢，但当你的朋友提出使用他们的“幸运”骰子时，你突然开始输起来，你可能会大概信任他们开始转运了。

在没有矿场证据的情况下，比如说，没有 RCIB，那么图 4.19 中矿产点的关系仅仅是可能的。随着证据的积累，这种可能关系将会变成合情，接着大概，最后如果样本确定了假设，则变为确实关系。当和其他人玩骰子时，如果你输了你赢的所有钱，这是合情的，你的运气不好。如果你输了所有的钱以及你的赌本，那么可以确认你运气不好。但不管你抛多少次骰子，甚至一百万次，都只是统计上的

一个机会，因为只有抛无数次骰子才能达到100%的概率。“0%的可能性”和“100%的可能性”实际上是自相矛盾的。0%或100%都是确实的，对于确实的东西不存在可能性，因为在现实世界中，确实就是信念。如果有人说：“我看到和别人一样的证据，但仍然确信有不是黑色的乌鸦”，他们是在表达一种信念，真正的信念不能被事实改变。

当你建立一个专家系统，尝试把专家的知识以规则和事实形式表达的时候，这是一个需要重点考虑的问题。当你和专家面谈获取知识时，你可能听到一些似乎是知识但实际上是信念的内容。在把这些内容放进专家系统时，你必须十分小心，因为它们可能导致错误结论。另一方面，假如专家雇用你就是基于这一点的话，如果系统不能如预期所料，他们会不高兴。

另一个关于可能性的自相矛盾是你听天气预报说下雨的可能性是50%。一个50%的可能性意味着完全不知道。要么下雨要么不下。你不需要概率理论或者天气预报员预测下雨还是不下。事实上，下雨不下雨是一个常识。同样地，可能在下着瓢泼大雨的时候你听到天气预报员说有100%的可能会下雨。这不是概率，这是确实。

在推理网中除了决策结点与它们的可能行为之间的关系外，还有一个值得注意的特性是**前后关系**(contexts)，它会阻断信息的传播，直到适当时候。使用前后关系可启动或中止推理网的一部分直到确知某个部分是否存在、缺乏或未知。前后关系的一个目的在于避免在所需的证据建立之前，系统询问使用者关于某些证据的问题。这一点非常重要，如果人们被问到一些似乎无关的问题，他们会觉得很苦恼。任何一个系统的目标都应该是获取最少的信息得到一个有效的或至少是可接受的结论。例如，如果你去看医生，第一个前后关系的内容是：“你有医疗保险吗?”从时间和金钱考虑，问更多不是必需的信息是昂贵的，例如“你有什么问题？你不舒服么？你有挑选好的太平间么?”

推理网中的第三种关系是**前后关系**（contexts），它会阻断信息的传播，直到适当时候。使用前后关系可启动或中止推理网的一部分直到确知某个部分是否存在或未知。前后关系的一个目的在于避免在所需的证据建立之前，系统询问使用者关于某些证据的问题。

前后关系的基本思想是控制系统追踪断言的顺序。前后关系说明了在使用一个断言前必须证明其所需的条件。前后关系用其下带有确定性范围的虚线箭头表示，如FMGS和PT。PT结点被阻断除非有确定性在0到+5之间的证据表明有小到中等颗粒大小的斑岩石组织。斑岩石是一种火山岩，其组织或外观是由嵌有小水晶的岩石组成，这些岩石称为母体。火山岩是由地球深处的熔石即岩浆凝固而成的。某些延伸到地面上的火山岩，即插角砾岩，是矿物构成物如斑岩铜的证据。

因此，斑岩铜是由嵌有小铜水晶的岩石母体组成的。斑岩铜是铜最普遍的存在形式。除非至少有小到中等颗粒大小的小水晶存在，否则询问斑岩组织是没有必要的，因此专家系统应该足够聪明不问这个问题。问没有必要问题的专家系统是无效率的且很快就会使用户苦恼。

对应于确定性因子为0，或者不超过+5的正数，其中+5表示证据“确实存在”。前后关系中确定性范围0到+5的意思是PT结点不会被追问，除非用户指明缺乏证据。

所有这3种方法组合或者容许的证据，都可是结点之间的实际关系。它们指明了一个断言在概率上的变化是如何影响其他断言的。

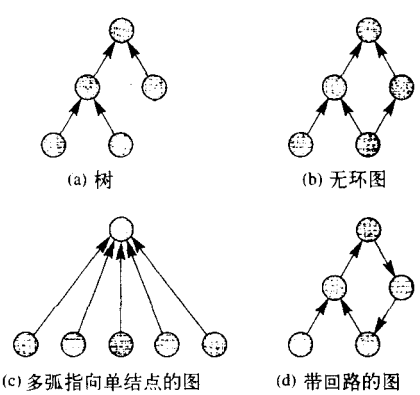


图4.21 图的一些类型

推理网的结构

形式地，一个推理网可以定义为一个有向无环图，其中结点表示断言，弧表示不确定性测度如LS和LN。图4.21 (a)，(b)和(c)就是合法的推理网结构，因为它们都没有回路。注意，在推理树中箭头指向假设，而在数据结构树中箭头却是从根发出。正如第3章所述那样，在无

环图中，沿着箭头是无法返回出发结点的。在图4.21（d）中有一个包含4个结点的环。禁止有环的原因是在建立假设时防止循环推理。在基于规则的系统中，一个合法的例外是通过两个相互触发的规则建立一个循环，该循环直到满足某个终止条件为止。

图4.21（e）是推理网的一种不期望类型。问题在于当有许多证据对一个假设起作用时，在证据之间很容易产生不必要的相互影响。一般说来，最好把这种类型转化为一种带有中间假设的更象树型的结构。

PROSPECTOR推理网也是一个**分段语义网**（partitioned semantic net）。网的每一段都形成一个有意义的单元。分段语义网是由Hendrix提出的，以便允许功能强大的谓词演算，如量化、蕴含、否定、析取、合取、回忆。下第2章的普通语义网，它实际上是为通过联系来表达描述性的知识而设计的。这种结构适合于因果关系的知识，但却难以表达逻辑关系。

分段语义网的基本思想是将抽象空间（spaces）中的结点集和弧集分类。其中的抽象空间定义了关系的范围。一个空间就好比是结构化语言中的模块或软件包的范围。例如，图4.18中的FRE结点就可认为是图4.19所示结构的一个空间。

语义网的能力主要体现在蕴涵语句。实际上，Hendrix在他的博士论文中首先设计它们用来表示自然语言。语句就好像组成推理网结点的证据和假设命题一样。作为分段语义网中语句的一个简单例子，考虑语句“有一台带彩显的计算机”。注意，这是一个存在句，因为它含有量词“有”。

图4.22用三个空间显示了表示这一语句的分段语义网。空间-1包含一些关于计算机的一般有关概念。空间-2包含我们所讨论的特定计算机，它被标识为计算机-1。空间-3包含特定关系COMPONENTS-OF-1，它以特定彩显-1，即我们所谈到的彩显，作为其对象。每条标有“元素”的弧表示一个状态是另一状态的元素。而标有“实体”的弧表示其所指特定计算机是带有特定彩显的实体。

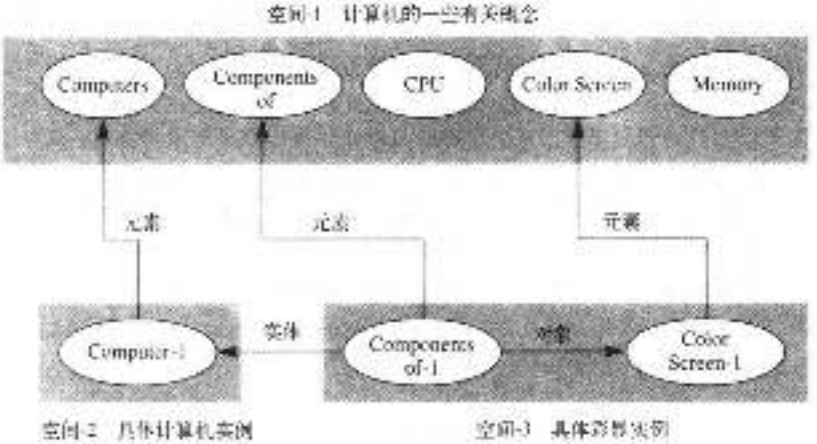


图4.22 一个关于计算机的简单分段语义网

分段语义网的优点可通过图4.23来说明。图4.23显示的是PROSPECTOR中关于结点STIR的一条规则，结点STIR使用了证据FMGS。该规则可描述为“如果有实体E-1，其基数值为ABUNDANT（丰富）且由INTRUSIVE ROCKS（插角砾岩）组成，那么STIR（去明石火山岩组织）”。注意，图表已经被简化了，因为没有包含前件中还必须有小到中等颗粒大小的条件。

注意，在图4.23中，一个分段语义网组成了规则的前件，另一个组成了后件。虽然该规则的后件比较简单，但其他规则会有更复杂的结构。分段语义网的优点在于系统可以推断出没有直接相连的结点之间的关系。这样系统的知识就比浅知识深了。

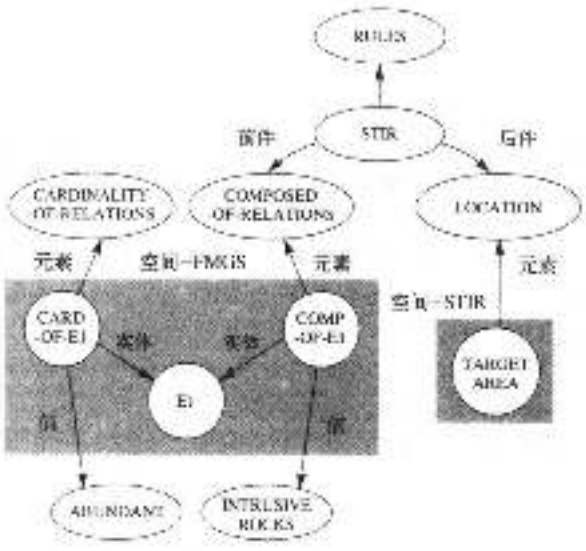


图 4.23 PROSPECTOR 分段语义网中使用了证据 FMGS 的结点 STIR 的简化规则

4.16 概率的传播

推理网，比如 PROSPECTOR 具有**静态知识结构**（static knowledge structure）。也就是说，结点和它们之间的联系是固定的，以便保持知识结构中结点之间的关系。这与基于规则的系统不同，其模式与事实匹配的规则被放入议程中，进行冲突归结，然后执行具有最高优先级的规则。因此，基于规则的系统是一个**动态知识结构**（dynamic knowledge structure），因为不像推理网结点之间那样，在规则之间没有固定的联系。

尽管推理网的结构是静态的，但是与每个假设结点关联的概率会随着证据的获取而改变。事实上，这种概率的改变是推理网中唯一改变的东西。推理网的基本特征就是随着证据的积累，概率从先验概率变为后验概率。这种概率的改变向着最终支持或否定顶层假设，如斑岩铜的存在——PCDA——的方向移动。

基于某些证据，让我们来看看 PCDA 中概率传播的几个阶段。这可作为你所见过的许多公式的一个清晰例子。参看图 4.19，我们将从证据结点 Region Contains Intrusive Breccias（该地区含有丰富的插角砾岩），简记为 RCIB 开始，跟踪概率的传播。

如果用户说，确实存在丰富的插角砾岩，那么

$$P(E_i \mid e) = P(SCIB \mid e) = 1$$

从图 4.19 可以看出，SMIR 的其他证据的 LN 值都为 1。也就是说，如果用户对 stocks（岩株），dikes（岩脉），和 volcanic plugs（火山栓）等证据没有什么补充的话，那么这些对 SMIR 没有什么作用，因此在概率的传播中可以被忽略。

所有这些证据可以被忽略的原因源于第 4.14 节讨论的复合证据公式：

$$O(H \mid e) = \left[\prod_{i=1}^{n} L_i\right] O(H)$$

其中，对所有已知为真的证据，$L_i = LS_i$；对所有已知为假的证据，$L_i = LN_i$。如果没有已知证据，则 $P(E_i|e)$ 还原为证据的先验概率 $P(E_i)$，同样 $P(H|E_i)$ 还原为先验概率 $P(H)$。于是

$$L_i = \frac{O(H \mid e_i)}{O(H)} = \frac{O(H)}{O(H)} = 1$$

且这些 L_i 对 $O(H|E)$ 没有什么作用。例如，当只有 RCIB 已知是真时，SMIR 的几率为：

$$O(H \mid E) = LS_{RCVP}\, LS_{RCAD}\, LS_{RCS}\, LS_{SMIRA}\, LS_{RCIB}$$

$$= 1 \times 1 \times 1 \times 1 \times 20$$

$$O(SMIR \mid RCIB) = 20$$

SMIR 证据的另一个有趣特性是标记为 SMIRA 的 OR 结点。注意 SMIR 的所有证据也连向 SMIRA。如果有任何对 SMIR 起作用的证据，如 RCS，RCAD，RCIB 或 RCVP，那么 SMIRA 中的或结点对 SMIR 不起任何作用，因为 SMIRA 的 LS 是 1。然而，如果没有 RCS，RCAD，RCIB 和 RCVP 这些证据，那么 SMIRA 中值为 0.0002 的 LN 值会使得 SMIR 的概率实质上为零。从根本上这意味着，虽然对 SMIR 来说，缺乏一个或多个证据并不重要（LS=1），但是对推断出 SMIR，缺乏所有证据是很重要的。

OR 结点、AND 结点、LS 值和 LN 值使得模型设计者可以灵活地利用证据来影响假设。然而，在某些复杂的情况下，为适应证据要求而添加的许多 OR 和 AND 结点会使推理网变得晦涩，以致让人难以理解。这些特殊结点的增加也需要对模型进行更多的测试。

从图 4.20 知，对插角砾岩 RCIB，有 LS=20，P（SMIR）=0.03。

由于

$$odds = O = \frac{P}{1-P}$$

于是 SMIR 的先验几率为：

$$O(SMIR) = \frac{0.03}{1-0.03} = 0.0309$$

如果证据 E 是确实的，则后验几率为：

$$O(H \mid E) = LS\ O(H)$$
$$O(SMIR \mid RCIB) = 20 \times 0.0309 = 0.618$$

且后验概率可由基本几率公式来计算。

$$P = \frac{O}{1+O}$$

$$P(H \mid E) = \frac{O(H|E)}{1+O(H|E)}$$

$$P(H \mid E) = \frac{0.618}{1+0.618} = 0.382$$

因此，在证据 RCIB 确实的条件下，SMIR 的后验概率为：

$$P(SMIR \mid RCIB) = 0.382$$

而 HYPE 的先验几率为：

$$O(HYPE) = \frac{0.01}{1-0.01} = 0.0101$$

此时，你可能会用

$$O(H \mid E) = LS\ O(H)$$

$$O(HYPE \mid SMIR) = LS_{SMIR}\ O(SMIR)$$

$$= 300 \times \frac{0.0101}{1+0.0101} = 3.00$$

作为 HYPE 在 SMIR 下的几率，但这是错误的。只有当证据 E 确实时，公式

$$O(H \mid E) = LS\ O(H)$$

才是正确的。该公式表示了在证据是确实的假定下，假设概率的改变。然而，SMIR 并未确实。事实上，在证据插角砾岩确实的条件下，SMIR 的概率是：

$$P(SMIR \mid RCIB) = 0.382$$

这意味着只有38.2%的似然性SMIR为真。

我们真正需要的是基于不确定性证据SMIR的假设HYPE的概率，即P（HYPE|RCIB)。计算这个概率的一种方法是用在第4.14节讨论的P（H|e）公式，即

$$P(H \mid e) = \begin{cases} P(H \mid E') + \dfrac{P(H) - P(H \mid E')}{P(E)} P(E \mid e') \\ \qquad \text{for } 0 \le P(E \mid e) < P(E) \\ P(H) + \dfrac{P(H \mid E) - P(H)}{1 - P(E)} [P(E \mid e) - P(E)] \\ \qquad \text{for } P(E) \le P(E \mid e) \le 1 \end{cases}$$

其中，SMIR证据的不确定性为：

$$P(E \mid e) = P(SMIR \mid RCIB) = 0.382$$

而我们所需的概率为：

$$P(H \mid e) = P(HYPE \mid SMIR)$$

其中我们所知的是：

$$P(H \mid E) = \frac{O(H \mid E)}{1 + O(H \mid E)} = \frac{3.00}{1 + 3.00} = 0.75$$

从图4.19可知先验几率是：

$$P(H) = P(HYPE) = 0.01$$

$$P(E) = P(SMIR) = 0.03$$

由于P（E|e）>P（E)，用公式P（E）≤P（E|e）≤1来计算P（H|E)，如下：

$$P(H \mid e) = 0.01 + \frac{(0.75 - 0.01)(0.382 - 0.03)}{1 - 0.03}$$

$$P(HYPE \mid RCIB) = 0.279$$

这就是HYPE的后验概率。

概率的这种传播构成了推理网。现在概率P（HYPE|RCIB）就是FLE的不确定性证据。利用换成了FLE值的同样公式可以算出后验概率P（FLE|RCIB)。

$$P(E) = P(HYPE) = 0.01$$

$$P(E \mid e) = P(HYPE \mid RCIB) = 0.279$$

$$P(H) = P(FLE) = 0.005$$

$$O(H \mid E) = O(FLE \mid HYPE) = LS_{HYPE}\, O(FLE)$$

$$= 200 \times 0.005 = 1$$

$$P(H \mid E) = P(FLE \mid HYPE)$$

由于P=odds/（1+odds)，

$$P(H \mid E) = O(FLE|HYPE)/(1 + O(FLE|HYPE))$$

$$= 1/(1 + 1)$$

$$= 0.5$$

$$P(H \mid e) = P(FLE \mid RCIB)$$

$$= 0.005 + \frac{(0.5 - 0.005)(0.279 - 0.01)}{1 - 0.01}$$

$$P(FLE \mid RCIB) = 0.140$$

4.17 小结

在这一章，我们首先讨论了不确定性推理的基本概念，以及由不确定性导致的误差的可能类型。然后回顾了经典概率理论的基本内容。讨论了经典概率理论和其他理论，如经验概率和主观概率的差别。也讨论了组合概率的方法和贝叶斯定理。本章还描述了信任与概率的关系，以及似然性。

分析了经典的专家系统 PROSPECTOR 以说明这些概率概念是如何应用于实际系统的。同时，以 PROSPECTOR 系统为例介绍了推理网和分段语义网。

尽管经典概率论在理想系统中运用良好，但它并不总是适用于现实系统，如 PROSPECTOR 矿产探测。虽然最初是用概率来处理不确定性，但直到引入信任因子和模糊逻辑到概率理论中，专家系统才正确预测出已知模型并发现了某些矿产。随后 PROSPECTOR 被用于未知特征的探测，并成功预测出价值 100 000 000 美元的钼矿。

注意，在探测其他类型矿产时需要用到不同的地质特征模型、不同的确定性因子以及模糊逻辑。这样做并没有不对的地方，因为这只不过是反映了现实世界中不同的矿产形成方式不同，所以探测它们的证据和理论也就不同。因此，不同领域的专家知识非常重要。擅长发现钼的专家不必擅长发现石油。

另一个重要的事情是你必须使你的专家系统适应现实世界，而不是试图让现实世界符合你的专家系统，否则会导致失败。所有的理论最终都是基于不需证明的公理，例如欧几里德的两条平行线永不相交的公理。尽管这一点在欧几里德平面中是正确的，但是在球行空间或者其他非平面空间中并不成立。不确定性理论也基于公理。我们需要引入其他因素如信任因子和模糊逻辑仅仅是因为我们不了解百万年前矿物质是如何发展，以及地球上那个位置的特殊化学结构，所以不能确定适用哪一个公理。

我们同时讨论了术语不可能、可能、合情、大概、确实的含义。理解这些术语对与专家沟通时从信任中分离出概率非常重要。你需要特别注意一些理所当然的内容，因为这些可能是由常识伪装而成。例如，常识告诉我们如果一匹马是活的，那么它跑完比赛的条件概率是 100%，P（Finish | Live）= 100%。注意，我们不是说它会赢而只是说它会跑完比赛。常识告诉我们要赢得比赛，竞争者必须活着。这是理所当然的，对么？所以你就因此把赌注压在这上？

正如前面所提到的，政治逻辑与其他任何类型的逻辑都不同。2000 年 10 月，密苏里州州长 Mel Carnahan 在竞选旅行时死于飞机失事。他的对手，Sen.John Ashcroft（R-Mo），仍然在 11 月份竞选失败。投票人选出了一个死去的参议员，并由 Carnahan 的遗孀担任这个职位（http: //www.specialed-news.com/washwatch/washnews/election111000.html）。这是一个很好的例子，死亡在今天也不能成为常识推理中的确定因素。

习题

4.1 (a) 只用概率的三个公理，证明概率的加法律。

$$P(A \cup B) = P(A) + (B) - P(A \cap B)$$

（提示：任意事件 X ∪ Y 可以写成 X ∪（X′∩Y）的形式，其中 X 和 X′ ∩Y 不相交。同样，B 也可以写成两个不相交集合的并）

(b) 已知两台计算机，它们或者正常工作或者不正常，根据概率的加法律，问至少有一台计算机正常工作的概率是多少？

4.2 已知可能重叠的两事件 A 和 B，根据概率公理，从集合的角度求以下概率：

(a) 非 A 非 B

(b) 或 A 或 B，但不同时发生（异或）

4.3 有三个箱子装着一些好的和一些坏的元件，如下：

箱子	好的	坏的
1	8	2
2	3	1
3	2	2

经过一段很长的时间后，有20%的元件从箱子1拿走，30%的元件从箱子2拿走；50%的元件从箱子3拿走。

(a) 画一棵概率树

(b) 如果拿走的是一个坏的元件，问它来自每个箱子的概率是多少？画一个显示结果及先验概率、条件概率、联合概率和后验概率的表。

4.4 某人打算购买一个磁盘驱动器，准备在三个品牌中挑选。下面这张表列出了他的偏好和在一年内驱动器损坏的概率（偏好是价格、速度、大小和可靠性的一个函数）：

品牌	选择的概率	损坏的概率
X	0.3	0.1
Y	0.5	0.3
Z	0.2	0.6

(a) 画一个显示结果、先验概率、条件概率、联合概率和后验概率的表。

(b) 在一年内，每种驱动器损坏的概率是多少？

(c) 在一年内，任一种驱动器损坏的概率是多少？

(d) 已知一年内已经损坏，问是X品牌X，Y，Z的概率各是多少？

4.5 筛选测试是为大群人检查某种疾病的一种低成本方法。一种花费更多但更准确的测试显示，所有人群中1%有该疾病。而筛选测试显示，那些确实有该疾病的人中90%测试有疾病（测试肯定），而那些确实没有该疾病的人中20%测试有疾病（错误肯定）。

(a) 画一个显示结果、先验概率、条件概率、联合概率和后验概率的表。

(b) 有百分之几的人，测试肯定但实际没有该疾病？（错误肯定）

(c) 有百分之几的人，测试否定但实际有该疾病？（错误否定）

4.6 证明两两独立不一定意味着相互独立。定义以下投掷两个骰子的事件：

A=第一个骰子是偶数

B=第二个骰子是偶数

C=和是偶数

(a) 画出两个骰子的样本空间。画出A，B，$A\cap B$。

(b) 写出$A\cap B$，C，$A\cap C$，和$B\cap C$的元素。

(c) P（C）为多少？

(d) 证明两两独立：

$$P(A \cap B) = P(A)P(B)$$
$$P(A \cap C) = P(A)P(C)$$
$$P(B \cap C) = P(B)P(C)$$

(e) 证明

$$P(A \cap B \cap C) = P(A \cap B) \neq P(A \cap B)P(C)$$

这说明两两独立并非意味着相互独立。

4.7 对不相交的两集合A和B

(a) 用A与B的概率来表示P（$A|B'$）是多少？（不是用A，B'）

(b) P（$A'|B$）的值为多少？

(c) P（$A|B$）的值为多少？

(d) 用A与B的概率来表示P（A′|B′）是多少？（不是用A′，B′）

(e) 计算

(i) $P(A|B) + P(A'|B)$

(ii) $P(A|B') + P(A'|B')$

4.8 (a) 证明

$$P(A \cap B \cap C) = P(A|B \cap C)P(B|C)P(C)$$

(b) 证明

$$P(A \cap B|C) = P(A|B \cap C)P(B|C)$$

4.9 一个磁盘驱动器可能会由于毛病 F_1 或 F_2 发生故障，F_1，F_2 不会同时发生。可能的症状有：

A= {写错误，读错误}

B= {读错误}

而且 F_2 发生的可能性是 F_1 的3倍。对这种类型的驱动器有

$P(A|F_1) = 0.4 \quad P(B|F_1) = 0.6$

$P(A|F_2) = 0.2 \quad P(B|F_2) = 0.8$

一个磁盘驱动器有毛病 F_2 的概率是多少？有毛病 F_2 的概率是多少？

4.10 给定一个每年更换磁盘驱动器品牌的转换矩阵：

		下一年 X	X	Z
今年	X	0.5	0.5	0
	Y	0.25	0.5	0.25
	X	0	0.5	0.5

(a) 假设开始有50%的人用X驱动器，25%的人用Y驱动器，25%的人用Z驱动器，问一年、两年和三年后使用各驱动器的百分比分别为多少？

(b) 求稳定状态矩阵。

4.11 (a) 给定N个随机选取的人，没有两人有相同生日的概率是多少？

(b) 对30个人，该概率是多少？

4.12 在 E_1 和 E_2 条件独立下，对一个含有合取证据的规则，

IF E_1 AND E_2 THEN H 证明

(a) $$P(H|E_1 \cap E_2) = \frac{P(E_1|H)P(E_2|H)}{P(E_1|H)P(E_2|H') + O(H')P(E_1|H')P(E_2|H')}$$

(b) $$P(H|E_1 \cap E_2) = \frac{P(H|E_1)P(H|E_2)}{P(H|E_1)P(H|E_2) + O(H)P(H'|E_1)P(H'|E_2)}$$

4.13 给定一个含有析取证据的规则：

IF E_1 OR E_2 OR ... E_N THEN H

基于概率理论，并且假定证据是条件独立的，证明

$$P(E \mid e) = 1 - \prod_{i=1}^{N}(1 - P(E_i \mid e))$$

其中E是证据，e是E的相关观测。

4.14 给定以下证据作为规则：

IF E THEN H

的前件。写出P（E|e）的模糊逻辑表达式。

(a) E= E_1 OR (E_2 AND E'_3)

(b) E = (E_1 AND E_2)OR(E_3 AND E_4)

(c) E = (E'_1 AND E'_2) OR E_3

(d) E = E'_1 AND (E'_2 OR E_3)

(e) E = E'_1 OR (E_2 AND E_3)

4.15 对斑岩铜的推理网，假定已知 RCS 为真，RCAD 为假。那么此时 FRE 的概率是多少？

4.16 考虑以下问题，应用语义归纳确定数列 2，8，8，…接下来的 3 个数是什么？在语义归纳中需要一个线索来找出数列次序（提示：考虑“交换”）

参考文献

Note: Many other references to probability, statistics and Bayesian software are given in Appendix G Software Resources.

(Castillo 97). Ed. by Enrique Castillo *et al., Expert Systems and Probabilistic Network Models*, Springer-Verlag New York, Inc., 1997.

(Gates 00). Bill Gates, *Business @ the Speed of Thought: Succeeding in the Digital Economy*, Warner Business Books, 2000. An excerpt from the book about Bayesian logic in Microsoft products is at: (http://www.microsoft.com/billgates/speedofthought/ additional/badnews.asp).

(Grinstead 97). Charles M. Grinstead and J. Laurie Snell, *Introduction to Probability*, American Mathematical Society, 2nd edition, 1997. NOTE: Besides including software, the book is also available GNU free from: (http://www.dartmouth.edu/~chance/teaching_aids/books_articles/probability_book/book-5-17-03.pdf)

(Jackson 99). Peter Jackson, *Introduction to Expert Systems*, Addison-Wesley, 3rd Edition, 1999.

(Neapolitan 03). Richard E. Neapolitan, *Learning Bayesian Networks*, Prentice-Hall, 2003.

(Pearl 00). Judea Pearl, *Causality: Models, Reasoning, and Inference*, Cambridge University Press, 2000.

(Ross 02). Sheldon M. Ross, *Introduction to Probability Models, Eighth Edition*, pp. 188-191, Academic Press, 2002.

注意：关于概率、统计和贝叶斯软件的其他参考资料列在附录 G 软件资源中。

第5章　不精确推理

5.1　概述

本章继续讨论从第4章开始的不确定性推理。在第4章中不确定性推理的主要范例是概率推理和贝叶斯定理，本章我们将讨论其他几种处理不确定性的方法。特别地，模糊逻辑已经有许多成功的应用，例如图像处理和控制，基于模糊规则的机器学习分类、聚类以及函数逼近（Ibrahim 03）（Chen 01）.

CLIPS有两个版本用来开发具有很多模糊推理的专家系统。一个是NRC的fuzzyCLIPS（http：//ai.iit.nrc.ca/IR_public/fuzzy/）。NRC也有为JAVA平台准备的工具包FuzzyJToolkit和FuzzyJess（基于JAVA的CLIPS版本）。另一个是Togai InfraLogic的模糊CLIPS，（http：//www.ortech-engr.com/fuzzy/togai.html）提供了很多到其他模糊资源的连接。模糊专家系统已经得到广泛应用，你可以很容易地从网站和书籍中得到信息（Siller 04）。

正如第4章所述，概率原本是为研究理想状况下游戏胜算而发展起来的，概率中的试验可被无限次重复。事实上，概率论被数学家称作是处理**可重复不确定性**（reproducible uncertainty）的一种理论。听起来像是一个矛盾修饰法，因为，如果一个事物是不确定的，为什么还可以重复？术语可重复意味着在统计意义上，大数量级地经过多次试验，结果都是平均的。

除了第4章所述的主观概率理论已成功应用到PROSPECTOR上外，还有其他一些理论也成功运用在许多其他应用上。这些理论主要用于处理有关信任的问题，而不是有关概率的经典解释——频率方面的问题。所有这些理论都是关于**不精确推理**（inexact reasoning）的。在不精确推理中，规则前件、结论、甚至规则本身在某种程度上都是不确定的。

5.2　不确定性与规则

本节将概述规则以及不确定性，随后的小节将全面介绍处理规则中和智能系统中不确定性的具体方法。智能系统这个词在今天已经有广泛的应用，严格地讲，是任何使用人工智能的系统。在实际上，经常用来描述可以操作不确定信息的系统。

规则中不确定性的来源

图5.1所示的是一个基于规则的系统中不确定性的高层视图。这些不确定性可能来源于单个规则、冲突归结和规则后件间的不相容。知识工程的目标就是尽可能地减少或消除这些不确定性。

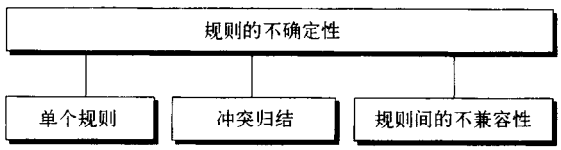


图5.1　基于规则的专家系统的主要不确定性

减少单个规则的不确定性属于规则验证的一部分。如第3.15节所述，验证与系统构成单元的正确性有关，对一个基于规则的系统来说，构成单元就是规则。

单个规则是正确的并不意味着系统将给出正确的答案。因为规则间的不兼容性，推理链可能不正确，因此有必要进行证实。对一个基于规则的系统来说，证实工作的一个部分就是减少推理链中的不确定性。验证可看作是减少局部不确定性，而证实是减少整个专家系统的不确定性。比如一个土木工程学上的粗略分析，验证是从好的材料及施工角度来问桥建好没有、而证实是问这桥是否能处理所需

要的交通负载问题，最重要的，桥是否建在合适的位置？验证与证实对于保证高质量的专家系统来说都是必需的，这将在随后的第6章中讨论。

图5.1中单个规则的不确定性可被更详细地扩展，其顶层视图如图5.2所示。除了包括如第4.3节中所描述的，规则创建时可能发生的错误之外，还有与似然性值的指定有关的不确定性。对概率推理而言，正如第4章中所述的，这些不确定性与充分似然性LS及必然似然性LN的值有关。由于LS和LN的值基于人们的估计，因此存在不确定性。此外，也有规则后件似然性的不确定性。对概率推理来说，确定的证据记作P（H | E），不确定的证据记作P（H | e）。

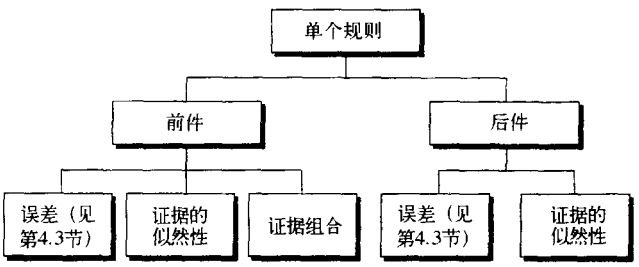


图5.2 单个规则的不确定性

另一个不确定性的来源是证据组合。证据应按哪种方式组合，是

E_1 AND E_2 AND E_3

或是 E_1 AND E_2 OR E_3

或是 E_1 AND NOT E_2 OR E_3

还是其他使用AND，OR，NOT等连接符的任意一种可能的逻辑组合？

缺乏理论基础

正如第4章所述，在对概率论中模糊逻辑等公式的介绍中提出了一个问题，专家系统没有一个基于经典概率论的合理的理论基础。所以，它只是一个适用于受限情形下的“特别”方法，“特别”方法的危险性在于没有一个完备的理论去指导应用，在不适宜条件下使用时，也没有出错警告。

你已见过的“特别”方法的另一个例子是推理网中LS和LN的使用。理论上，一个有N个结点的推理网是一个事件空间有N个可能事件的概率系统，因此，有2^N种可能概率。实际上，在现实世界中，这些概率只有很少是已知的。作为罗列所有可能情况的替代，我们用LS和LN值来标识弧，这样大大减少了网络的复杂度。

然而，LS与LN的简化使用也有不利的影响，这就是不能从理论上来保证，网中每种假设的条件概率之和为1。如果只是想获得网中所有假设的相对级别，那么，这并不重要。例如，当一个对斑岩铜感兴趣的用户得知该结点表明有铜的似然性非常大，即使它不是1，用户也可能会相当满意。不过绝对概率也很重要，即使有铜的假设排在第1，但如果发现铜的概率非常小，比如0.00000001，那么是否也值得钻探呢？

规则间相互影响

不确定性的另一个来源是冲突归结的不确定性。如果知识工程师指定了规则的**显式优先级**（explicit priority），那么将有一个潜在的误差来源，因为优先级未必是最优或正确的。而如果规则有**隐式优先级**（implicit priority），则又会发生冲突归结。

不确定性的一个主要来源是由规则间的相互作用而引起的。冲突归结是其中一部分。规则间的相互作用依赖于冲突归结和**规则的兼容性**（compatibility of rules），如图5.3所示，与规则的兼容性有关

的不确定性来自 5 个主要因素。

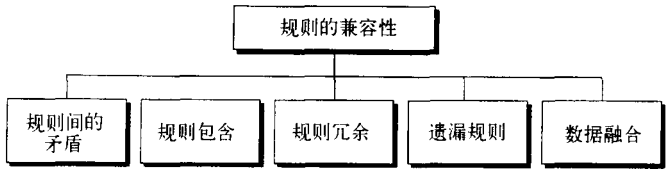


图 5.3 与规则的兼容性有关的不确定性

不确定性的一个原因是**规则间的潜在矛盾**（potential contradiction of rules）。规则将产生矛盾的结果，如果前件没有适当说明的话，就可能会产生这种情况。举一个非常简单的例子，假设知识库中有两条规则：

(1) IF 有火灾 THEN 泼水

(2) IF 有火灾 THEN 不要泼水

规则（1）对普通火灾是适用的，比如木头着火，规则 2 对油脂类火灾是适用的，问题在于前件没有明确给出火灾的类型。如果存在“有火灾”的事实，那么两条规则都将执行，并产生矛盾结果，于是产生了不确定性。

不确定性的第二个来源是**规则的包含**（subsumption of rules），如果一个规则的前件是另一个的子集，那么它被另一个规则包含。例如，假设两条规则有同样结论：

(3) IF E_1 THEN H

(4) IF E_1 AND E_2 THEN H

如果仅 E_1 存在，那么不会有问题，因为只有规则（3）被激活。然而，如果 E_1 与 E_2 同时存在，那么两条规则都会被激活，它们中间必然要发生冲突归结。

冲突归结

对火灾问题的优先级，在冲突归结中存在不确定性，该不确定性依赖于一些因素，如图 5.4 所示。

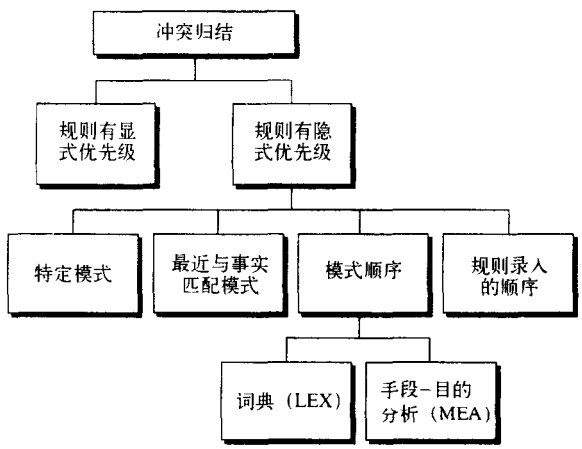


图 5.4 与冲突归结有关的不确定性

第一个因素是所使用的外壳和工具。在使用 Rete 算法的专家系统工具中，越特定的规则具有越高的优先级。这一点的意义在于：当多条规则可能被引发执行时，信息越多越能增加我们的信心。

例如，如果有人说“我觉得热”，有一条规则是：IF 热 THEN 服用阿司匹林。另一方面，有一条

规则是：IF 热 AND 在沙滩上穿着外套 THEN 脱下外套。如果有更多的信息可以判断这个人在沙滩穿着外套，则这个规则更特定。通常，符合规则左侧模式个数多的规则会比符合模式少的规则得到更多的信任。CLIPS 规则的**特定性**（specificity）依赖于模式的数目以及每个模式的内部复杂度。例如模式

（ball ellipsoidal）

比另一模式

（ball）更加特定。

因为一个椭圆形的球可能是橄榄球，而球可能是网球、乒乓球、篮球、棒球或其他任何类型的球。

对于前面的两条规则，规则（4）更特定，因为它的前件有两个模式，这样它有较高的隐式优先级。在 CLIPS 系统中，规则（4）首先被执行。

然而，在 CLIPS 中，还有其他一些纠纷。除了特定性外，还必须考虑**事实的最近性**（recency of facts）。每当一个事实输入到工作存储区中时，它将获得一个唯一的**时间标识**（timetag），表明它是什么时候进入的。这样规则

(5) IF E_3 THEN H

将比规则（3）有较高级别的优先级，如果事实 E_3 在 E_1 后输入的话。

在一条规则中模式的次序也会有很大关系。OPS5 允许两种不同的**控制策略**（control strategy），叫做**词典**（LEX，lexicographic）和**手段目的分析**（MEA，means-ends analysis），这些控制策略决定了推理机如何去解释规则模式，当输入一条策略命令给专家系统解释器时，就选定了一种控制策略。在词典（LEX）方式下，模式次序没有关系，只是对效率可能有影响。这样在词典方式下，下面规则在本质上是相同的：

IF E_1 AND E_2 THEN H

IF E_2 AND E_1 THEN H

手段目的分析是另一种策略方式，它被用于第 1 章中所讨论的 Newell-Simon 通用问题求解器。它的基本思想是不断减少开始状态与成功状态之间的差异。在 OPS5 的 MEA 方式下，第一个模式非常重要，因为它控制着模式匹配过程。MEA 的目的是帮助系统持续执行某个任务，以免被最近输入工作存储区的事实干扰。MEA 根据第一个，即 IF 之后的模式来决定优先级。如果基于第一个模式某条规则优先于其他规则，那么它将被选定。这种优先可以依据特定模式或者事实的最近性。如果优先的第一个模式有多个，那么 MEA 将根据它们后面的模式来决定优先级。如果没有优先规则，可任选一条规则执行。

规则输入进专家系统的次序也可能是产生冲突归结的因素之一。如果推理机不能确定规则的优先级，那么将任选一个执行。然而，除非推理机的设计者非常仔细，否则就某种意义上来说，选择次序已经确定，因为它依赖于规则的进入次序。由于推理机设计者使用栈或队列去存储议程中的规则，因此这种情况是可能发生的。对具有优先级的规则进行任意选择，实际上应该是从栈或队列中随机选择。然而，设计者可能只是简单地从栈顶或队列的下一位置中取出规则。这种选择方法虽然容易但却给系统带来一个已知的人为机制，因为选择并不是任意的。

包含与不确定性

如果存在与规则有关的似然性，那么包含问题将更加不确定。如

(6) IF E_1 THEN H with LS_1

(7) IF E_1 AND E_2 THEN H with LS_2

例如，考虑以下规则：

IF 启动马达不工作

THEN 检查电池 with LS_1 = 5

IF 启动马达不工作 AND 灯不亮
THEN 检查电池 with $LS_2 = 10$

这里 LS_2 大于 LS_1 值，因为有更多的证据支持结论。

由第 4 章可知，后验几率

$$(8)\ O(H \mid E) = \left[\prod_{i=1}^{N} LS_i\right] O(H)$$

意味着，如果 E_1 与 E_2 都存在的话，那么积为 $5 \times 10 = 50$。

然而，(8) 实际上是对那些证据组合是独立的规则而言的。(6)、(7) 的前件并不是真正独立的，因为它们共享 E_1 证据。这样，由于单个 E_1 是 E_1 AND E_2 的一个特例，所以 E_1 AND E_2 包含 E_1。因此，由于前件条件不独立，使得似然率 LS_1 和 LS_2 的积非常大。一个解决方法是用 LS_2 / LS_1 去替换 LS_2，以便当 E_1 和 E_2 都存在时，可得到正确的积 LS_2。在一个大型的知识库中，手工发现包含是非常困难的。

不确定性的第 3 个原因是具有同样后件和证据的**冗余规则**（redundant rules）。一般，这种情况由知识工程师偶然输入或由于修改一条规则时模式的删除而引发。例如：

IF E_1 AND E_2 THEN H

IF E_2 AND E_1 AND E_3 THEN H

如果 E_3 被删除的话，由于前件条件相同，上面的规则将是冗余的。决定那条冗余的规则被删除并不简单。如果稀有的模式被首先列出的话，一个冗余的规则会大大提高系统效率。如果第一个模式不匹配，推理机不必去检查第二个模式是否匹配。此外在专家系统外壳，如 OPS5 中还会产生一个复杂的问题，因为其规则激活的次序依赖于用户所选择的控制策略。因此如包含一样，冗余规则也将给出一个很大的似然率积。

不确定性的第 4 个原因是遗漏规则，这发生在人类专家忘记或者没有注意的情况下，比如规则

IF E_4 THEN H

如果证据 E_4 被忽略，那么 H 将不能得出。推理网，如 PROSPELTR 的一个优点是清楚明了，这使得确定一个假设不可能或者很难达到非常容易。如果遗漏规则，推理网将会表明需要这些规则。

不确定性的第 5 个原因是由于**数据融合**（data fusion），这一术语指与从不同类型信息中融合数据有关的不确定性。例如，在进行诊断时，医士可能会考虑不同来源，如体检、化验、病史、社会经济环境、精神情感状况、家庭、工作问题等方面的证据，这些都是不同类型的证据，必须融合它们以支持最终假设。融合这么多不同来源的证据要比融合某一领域，比如地质方面的证据要困难的多。

同样地，一个商业决策也依赖于商品市场、经济条件、外贸、接收意图、公司政策、个人问题、工会和其他许多因素，如同医学问题一样，指定所有这些因素的似然率并选择一个合理的组合函数是相当困难的。

5.3 确定性因子

另一个处理不确定性问题的方法是用**确定性因子**（certainty factors），这一方法最初是为 MYCIN 专家系统设计的。

贝叶斯方法的困难

如同由 PROSPECTOR 提出的地质问题一样，医疗诊断问题也常常具有不确定性。主要的不同是关于矿物的地质假设数目是有限的，因为自然界中只有 92 种天然元素。但是，由于有更多的微生物，因而可能的疾病假设也更多。

虽然贝叶斯定理在医学上有用，但它的准确使用依赖于知道多少种可能。例如，给定确定的症状，可用贝叶斯定理来确定某一特定疾病的概率

$$P(D_i \mid E) = \frac{P(E \mid D_i)\,P(D_i)}{P(E)} = \frac{P(E \mid D_i)\,P(D_i)}{\sum_j P(E \mid D_j)\,P(D_j)}$$

这里 j 是对所有疾病求和，且

D_i 是第 i 种病，

E 是证据，

$P(D_i)$ 是在已知任何证据之前病人得这种病的先验概率，

$P(E|D_i)$ 是在已知患有 D_i 疾病的情况下，病人呈现 E 证据的条件概率。

对一般的人群来说，要确定所有这些概率一致的、完全的值常常是不可能的。

实际上，证据趋向于一点一点的积累，这种积累需一定时间且花费昂贵，尤其是在需要医学检验时。时间、花费、检验时潜在的危险这些因素常常使得检验的项目被限制在一个好的诊断所需的最少数目。

贝叶斯定理表达证据逐步增加积累的一个便利形式如下（参见习题 5.1）：

$$P(D_i \mid E) = \frac{P(E_2 \mid D_i \cap E_1)\,P(D_i \mid E_1)}{\sum_j P(E_2 \mid D_j \cap E_1)\,P(D_j \mid E_1)}$$

这里 E_2 是已存在证据 E_1 上的新增证据，于是得到新证据

$$E = E_1 \cap E_2$$

虽然该公式是准确的，但所有这些概率一般并不知道。而且，随着证据的积累，要求更多的概率，情况会越来越糟。

信任与不信任（Belief and Disbelief）

除了贝叶斯定理要求知道所有的条件概率问题外，另一个与医学专家一起出现的主要问题是信任与不信任的关系问题。乍看起来这似乎是个小问题，因为显然不信任仅仅是信任的相反情况。但事实上，概率理论是如下描述的：

$$P(H) + P(H') = 1$$

于是，

$$P(H) = 1 - P(H')$$

对依赖于证据 E 的后验假设，有

$$(1)\ P(H \mid E) = 1 - P(H' \mid E)$$

然而，当 MYCIN 工程师访问医学专家时，他们发现医生非常反对把他们的知识转化成式（1）的形式。

例如，考虑如下的一个 MYCIN 规则：

IF 1) 生物体的染色呈革兰氏阳性，并且
2) 生物体的形态是球形，并且
3) 生物体生长构造是链状
THEN 有证据表明(0.7)这种生物是链球菌

简单地说，这个规则是指：如果一种细菌生物体在革兰氏染色中呈现颜色，并且生物体是呈球状链式排列，则有 70% 的似然性确定它是一种链球菌。用后验概率可写为：

$$(2)\ P(H \mid E_1 \cap E_2 \cap E_3) = 0.7$$

其中 E_i 对应着前件的 3 个模式。

MYCIN 知识工程师发现，尽管专家同意式（2），但他们不乐意且拒绝同意下面的概率结果：

$$(3)\ P(H' \mid E_1 \cap E_2 \cap E_3) = 1 - 0.7 = 0.3$$

专家们不愿接受式（3）说明了这些数字如0.7和0.3是信任的似然性而不是概率。

为了充分说明信任和不信任的不一致问题，考虑下面的例子，假设这是你拿到学位所需的最后一门课，假定你的平均分数（GPA）不是很好，你需要在这门课中得“A”以提高你的GPA，下面这个公式表达了你要毕业的信任似然性：

(4) P(毕业|这门课A) = 0.70

注意到该似然性不是100%，这是因为你所修的课程和分数要由学校作最后审查。下面的一系列原因仍会阻止你毕业：

1. 课程目录的改变使得你所修的课程中不是所有的都是学位要求必修课。
2. 你忘记了修一门必修课。
3. 不承认副修课程。
4. 不承认你所学的某些选修课程。
5. 你欠学费或图书馆的罚款忘了交。
6. 你的平均分比你想像的还要低，即使得了“A”也拉不上来。
7. “他们”企图为难你。

假设你同意（4）式的结果，（或者你自己选一个似然性值）则由（1）式

(5) P(不能毕业|这门课得A) = 0.3

尽管从概率的角度看，（5）式是正确的，但直觉似乎觉得它是错的。如果你真的很努力，且在这门课中得了“A”，而你却有30%的可能毕不了业，仅这一点就不对。如同那些相信

$$P(H \mid E_1 \cap E_2 \cap E_3) = 0.70$$

但不相信概率结果

$$P(H' \mid E_1 \cap E_2 \cap E_3) = 0.30$$

的医学专家一样，（5）式令你难以接受。

根本问题在于，尽管P（H|E）暗示着E和H存在一种因果关系，但E和H′可能没有因果关系。可是，式

$$P(H \mid E) = 1 - P(H' \mid E)$$

却暗示着，如果E和H有因果关系，则E和H′也有因果关系。

概率论上的这些问题使得Shortliffe去寻求找表达不确定性的另外途径。他在MYCIN中所使用的方法是基于从Carnap的证实论中得来的确定性因子。Carnap把概率分为两种类型。

一种类型是与重复事件出现频率有关的普通概率，第二类叫做**认知概率**（epistemic probability）或**确认度**（degree of confirmation），因为它是基于某些证据去证实假设。第二类概率是信任似然度的另一个例子。

信任与不信任的测度

在MYCIN中，确认度最初定义为确定性因子，它是信任与不信任二者的差：

CF(H,E) = MB(H,E) - MD(H,E)

其中，

CF是由证据E得到假设H的确定性因子。

MB是由证据E得到假设H的**信任增加测度**（measure of increased belief）。

MD是由证据E得到假设H的**不信任增加测度**（measure of increased disbelief）。

确定性因子是把信任和不信任组合成单个数字的一种方式。

把信任与不信任测度组合成单个数字有两个好处。第一，确定性因子可以用来把假设按重要性排

序。例如，如果一个病人表现出好几种病都有可能的某些症状，那么具有最高 CF 值的那种疾病将被首先安排检验。

用概率来定义信任与不信任的测度为：

$$MB(H,E) = \begin{cases} 1 & \text{如果 } P(H)=1 \\ \dfrac{\max[P(H \mid E), P(H)] - P(H)}{\max[1,0] - P(H)} & \text{否则} \end{cases}$$

$$MD(H,E) = \begin{cases} 1 & \text{如果 } P(H)=0 \\ \dfrac{\min[P(H \mid E), P(H)] - P(H)}{\min[1,0] - P(H)} & \text{否则} \end{cases}$$

此式中 max［1，0］总是取1，min［1，0］总是取0，之所以这样写是为了体现 MB 和 MD 间的一种形式对称性。MB 和 MD 公式之间的不同之处只是把 max 替换为 min，反之亦然。

根据这些定义，表 5.1 列出了它们的一些特征：

确定性因子 CF 是指基于某些证据的假设的纯信任。正的 CF 意味着证据支持假设，这是因为 MB>MD。CF = 1 意味着证据一定可以推出假设。CF=0则有两种可能性，第一，CF=MB - MD=0 可能意味 MB 和 MD 都为0，即是说，没有证据；第二，MB=MD≠0，即是说，信任被不信任所抵消。不幸的是，由不信任所取消的强信任并不是简单忽略，而是一种混淆状态。例如，更糟糕的情况是在一个路口转向哪一边，或者你车上的乘客说着右边却指向左边。

CF 为负意味着证据有利于否定假设，这是因为 MB<MD。换言之，有更多的理由使我们不信任这个假设多于信任它。例如，CF= - 70%表明，不信任比信任多70%，而 CF= 70%表明信任比不信任多70%。注意，有了确定性因子后，MB 和 MD 的各自取值可不限定，重要的是 MB 和 MD 的差值。例如：

$$CF = 0.70 = 0.70 - 0$$
$$= 0.80 - 0.10$$

等等。

表 5.1 MB，MD 和 CF 的一些特征

特　征	值
	0≤MB≤1
范围	0≤MD≤1
	-1≤CF≤1
确定的真假设	MB = 1
P(H|E)= 1	MD = 0
	CF = 1
确定的假假设	MB = 0
P(H′|E)= 1	MD = 1
	CF = -1
缺乏证据	MB = 0
P(H|E)= P(H)	MD = 0
	CF = 0

确定性因子允许一个专家在无须设定一个值给不信任的情况下去表达一个信任。正如习题 5.2 所示，

$$CF(H,E) + CF(H',E) = 0$$

意味着如果证据是以某个 CF（H|E）值去证实假设的话，它并不像概率论中所说的是以 1 - CF（H|E）的确认度去否定假设，即是说

$$CF(H,E) + CF(H',E) \neq 1$$

事实上，CF(H|E)+ CF（H′| E)= 0 意味着证据以相同的数量值支持一个假设和不支持这个假设的否定。因此，二者之和总是为0。

对学生毕业的例子来说，如果给定课程 A，

$$CF(H,E) = 0.70 \qquad CF(H',E) = -0.70$$

意思是：

(6) 如果我这门课得了 A，我就有 70％的把握我会毕业。

(7) 如果我这门课得了 A,我就有 -70% 的把握我毕不了业。

注意到可以出现 -70%。这是因为确定性因子是定义在区间

$$-1 \leqslant CF(H,E) \leqslant +1$$

上，其中 0 表示无证据。因此，确定性因子的值大于 0 则支持假设，小于 0 则支持假设的否定。如果使用确定性因子去分析的话，语句 (6) 和 (7) 是等价的，这相当于“是=不是不”。

以上的 CF 取值可能会引发下面的疑问：

如果证据是支持假设的话，

你对得了 A 将有助于你毕业,有多少程度的相信?

或者

你对得了 A 将有助于你毕业,有多少程度的不相信?

对每个问题回答 70%，将使 $CF(H|E) = 0.70$ 和 $CF(H'|E) = -0.70$。在 MYCIN 中，不是用百分比去询问确定性因子，而是要求专家用 1～10 的数来表示，其中 10 表示肯定。用户可以用表示不知道的 UNK 来回答，这相当于 CF = 0。

确定性因子的计算

尽管 CF 的原始定义是：

$$CF = MB - MD$$

但这个定义有一些困难之处。因一反面的证据会支配很多正面证据。如：对可能产生 MB=0.999 的 10 个正面证据和 MD=0.799 的一个反面证据，会得出：

$$CF = 0.999 - 0.799 = 0.200$$

在 MYCIN 中，一个规则前件的 CF 值必须>0.2，才能认为该前件为真并激活这条规则。在 CF 理论中，0.2 这个**阈值**（threshold value）不是作为一个基本公理，而是作为一个“特别”方法来减少所激活的对假设仅仅弱支持的规则数目。如果没有这个阈值，许多 CF 值很小甚至没有值的规则都将被激活，这将大大降低系统的效率。

1977 年，MYCIN 中的 CF 定义改为：

$$CF = \frac{MB - MD}{1 - \min(MB, MD)}$$

以削弱一个反面证据对多个正面证据的影响。基于这个定义，若 MB=0.999，MD=0.799，则

$$CF = \frac{0.999 - 0.799}{1 - \min(0.999, 0.799)} = \frac{0.200}{1 - 0.799} = 0.995$$

这个结果与由原始定义得到的 $0.999 - 0.799 = 0.200$ 有很大的不同，原先由于 CF 不大于 0.2 而不能激活规则，现在 CF 为 0.995 将激活规则。

MYCIN 计算规则前件中组合证据 CF 的方法如表 5.2 所示。注意，这些方法与基于模糊逻辑的 PROSPECTOR 规则中所使用的方法是一样的。

例如，给定一个组合证据的逻辑表达式：

$$E = (E_1 \text{ AND } E_2 \text{ AND } E_3) \text{ OR } (E_4 \text{ AND NOT } E_5)$$

证据 E 的确定性可如下计算：

$$E = \max[\min(E_1, E_2, E_3), \min(E_4, -E_5)]$$

给定值，

$$E_1 = 0.9 \quad E_2 = 0.8 \quad E_3 = 0.3$$
$$E_4 = -0.5 \quad E_5 = -0.4$$

结果为：

表 5.2 MYCIN 中计算组合前件证据的确定性的基本表达式规则

证据,E	前件确定性
E_1 AND E_2	$\min[CF(H, E_1), CF(H, E_2)]$
E_1 OR E_2	$\max[CF(H, E_1), CF(H, E_2)]$
NOT E	$-CF(H,E)$

```
E = max[min(0.9,0.8,0.3),min(-0.5,-(-0.4)]
  = max[0.3,-0.5]
  = 0.3
```

规则

```
IF E THEN H
```

的 CF 基本公式可表示为：

```
(8) CF(H,e) = CF(E,e) CF(H,E)
```

其中，

CF（E，e）是证据 E 基于不确定证据 e 而构成规则前件的确定性因子。

CF（H，E）是假定证据已知确定时，即当 CF（E，e）= 1 时，假设的确定性因子。

CF（H，e）是基于不确定证据 e 的假设的确定性因子。

因此，如果前件里所有证据都是已知确定的，由于 CF（E，e）= 1，则假设的确定性因子公式为：

```
CF(H,e) = CF(H,E)
```

作为这些确定性因子的一个例子，考虑前面已讨论过的链球菌规则：

```
IF  1)生物体的染色呈革兰氏阳性,并且
    2)生物体的形态是球形,并且
    3)生物体生长构造是链状
    THEN  有证据表明(0.7)这种生物是链球菌
```

这里，在确定证据支持下假设的确定性因子是：

$$CF(H,E) = CF(H,E_1 \cap E_2 \cap E_3) = 0.7$$

这个因子也称为**衰减因子**（attenuation factor）。

衰减因子是基于这样的假设，所有证据 E_1，E_2，E_3 都是已知确定的。即是说，

$$CF(E_1,e) = CF(E_2,e) = CF(E_3,e) = 1$$

其中 e 是观测证据，由它可得出结论 E_i 是确定的。这些 CF 值类似于 PROSPECTOR 中证据的条件概率 P（E|e）。衰减因子表达了在给定确定证据下，假设的确信度。

与 PROSPECTOR 中一样，如果不是所有证据都是已知确定的，则情况将变得复杂。在 PROSPECTOR 中，对于不确定证据是用插值公式 P（H|e）。在 MYCIN 中，由于 CF（H，$E_1 \cap E_2 \cap E_3$）= 0.7 对于不确定证据已不再有效，因此必须用公式（8）去决定 CF 的值。

例如，假设

$$CF(E_1,e) = 0.5$$
$$CF(E_2,e) = 0.6$$
$$CF(E_3,e) = 0.3$$

则

$$\begin{aligned} CF(E,e) &= CF(E_1 \cap E_2 \cap E_3,e) \\ &= \min[CF(E_1,e),CF(E_2,e),CF(E_3,e)] \\ &= \min[0.5,0.6,0.3] \\ &= 0.3 \end{aligned}$$

又因为前件条件的 CF（E，e）> 0.2，所以前件被认为是正确的。因此规则被激活，结论的确定性因子为：

$$\begin{aligned} CF(H,e) &= CF(E,e)\ CF(H,E) \\ &= 0.3 \times 0.7 \\ &= 0.21 \end{aligned}$$

假设有另外一个规则也得出相同的假设，但有不同的确定性因子。得出相同假设的规则的确定性因子可通过如下定义的确定性因子的**组合函数**（combining funciton）来计算：

$$(9)\quad CF_{COMBINE}(CF_1,CF_2,)=\begin{cases}CF_1+CF_2\ (1-CF_1) & \text{both}>0\\ \dfrac{CF_1+CF_2}{1-\min(|CF_1|,|CF_2|)} & \text{one}<0\\ CF_1+CF_2\ (1+CF_1) & \text{both}<0\end{cases}$$

其中 $CF_{COMBINE}$中各公式的使用取决于各确定性因子的正负，多于两个确定性因子的组合函数要迭代应用，即是说，先计算出两个 CF 值的 $CF_{COMBINE}$，然后再用这个 $CF_{COMBINE}$和第三个 CF 值代入（9）式计算，依此类推。图 5.5 总结了基于不确定证据和得出相同假设的两规则的确定性因子的计算。注意，这不是一棵与或树，因为 $CF_{COMBINE}$同与或无关。

在我们所举的例子里，如果另外一个得出链球菌的规则具有确定性因子 $CF_2=0.5$，则利用（9）中第一个式子可得组合确定性因子：

$$CF_{COMBINE}(0.21,0.5)=0.21+0.5\ (1-0.21)=0.605$$

假设第三个规则也有相同的结论，但 $CF_3=-0.4$，则利用（9）中第二个式子可得出：

$$CF_{COMBINE}(0.605,-0.4)=\frac{0.605-0.4}{1-\min(|0.605|,|-0.4|)}$$

$$=\frac{0.205}{1-0.4}=0.34$$

$CF_{COMBINE}$公式具有证据的可交换性，即：

$$CF_{COMBINE}(X,Y)=CF_{COMBINE}(Y,X)$$

因此证据的次序并不影响结果。

在 MYCIN 中，并不是在每个假设中分别存放 MB 和 MD 的值，而是保存每个假设的当前 $CF_{COMBINE}$值，当需要时，再将它和新的证据进行组合。

$$CF_{COMBINE}(CF_1,CF_2)=\begin{cases}CF_1+CF_2\ (1-CF_1) & \cdots\cdots CF_1,CF_2\text{ 都大于0}\\ \dfrac{CF_1+CF_2}{1-\min(|CF_1|,|CF_2|)} & \cdots\cdots CF_1,CF_2\text{ 其中之一小于0}\\ CF_1+CF_2\ (1+CF_1) & \cdots\cdots CF_1,CF_2\text{ 都小于0}\end{cases}$$

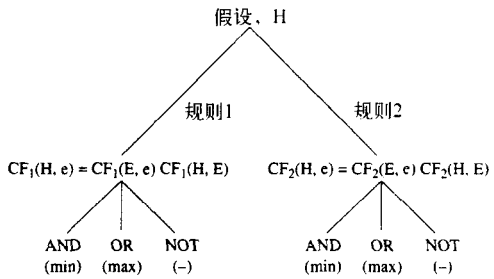


图 5.5 基于不确定证据的有相同假设的两个规则的 CF

确定性因子的困难

尽管 MYCIN 在诊断中是成功的，但确定性因子的理论基础仍有困难。虽然确定性因子部分基于概率理论和证实理论，但 CF 也有部分是“特别”的。CF 的主要优点是：通过简单的计算，不确定性可以在系统中传播。CF 也容易理解，并且将信任与不信任清楚地区分开来。

然而 CF 也有问题存在。一个问题是，CF 值可能与条件概率得出的值相驳。例如，若

$$P(H_1)=0.8\quad P(H_2)=0.2$$
$$P(H_1\mid E)=0.9\quad P(H_2\mid E)=0.8$$

则

$$CF(H_1,E) = 0.5 \text{ and } CF(H_2,E) = 0.75$$

因为 CF 的一个目的就是像诊断一样，把假设归类，故若有一种疾病有很高的条件概率 P（H|E）但却有很低的确定性因子 CF（H|E），则会产生矛盾。

CF 的第二个主要问题是，通常

$$P(H \mid e) \neq P(H \mid i)\ P(i \mid e)$$

其中 i 是基于证据 e 的某些中间假设，然而在一个推断链中的两条规则的确定性因子却是作为独立概率计算的：

$$CF(H,e) = CF(H,i)\ CF(i,e)$$

上式只有在下面的特殊情况下才成立，即具有属性 H 的统计种群包含在具有属性 i 的统计种群中，且它们又包含在具有属性 e 的统计种群中。尽管存在这些问题，MYCIN 仍然是成功的，这可能是由于推理链短且假设简单的缘故。如果把确定性因子应用到其他不具有短推理链和简单假设的领域，就有可能出现问题了。事实上，Adams 证明了确定性因子理论实际上是标准概率理论的一个近似。

5.4 Dempster-Shafer 理论

在这一节里，我们将讨论一个称之为 **Dempster-Shafer**（或 Shafer-Dempster）**理论**（theory）的不精确推理方法，它基于最早由 Dempster 所做的工作。Dempster 曾试图用一个概率范围而不是单个的概率值去模拟不确定性。Shafer 在 1976 年出版的《Mathematical Theory of Evidence》一书中延拓并改进了 Dempster 的工作。一个进一步的扩展称之为**证据推理**（evidential reasoning），它用来处理那些不确定、不精确、间或不准确的信息。Dempster-Shafer 理论有良好的理论基础。确定性因子可以看作是 Dempster-Shafer 理论的一个特例，Dempster-Shafer 理论给了确定性因子一个理论性的基础而不是“特别”基础（Gardenfors 04）。Dempster-Shafer 理论也被用于智能数据库中，被数据挖掘用来做模式抽取（Bertino 01）。

识别框架

Dempster-Shafer 理论假设了一个不变的两两互斥的完备元素集合，称作**环境**（environment），用希腊字母 Θ 表示。

$$\Theta = \{\theta_1,\theta_2,\cdots\theta_N\}$$

环境是集合论中所述的论域的另一种说法。即是说，环境是一个我们感兴趣的对象集合。例如，环境的一些例子：

Θ = {客机,轰炸机,战斗机}

Θ = {红,绿,蓝,橙,黄}

Θ = {谷仓,草,人,牛,车}

注意，这里的元素是两两互斥的。例如，客机不是轰炸机或战斗机，红不是绿，草不是牛等等。如果宇宙间所有可能的元素都包括在一个集合中，就说这个集合是完备的。为了讨论的简单起见，假设 Θ 是一个有限集。但是，对 Dempster-Shafer 环境中元素是连续变量，如时间、距离、速度等方面的工作，人们也已经做了。

一个考虑 Θ 的方式是从问题与答案的角度。假定

Θ = {客机,轰炸机,战斗机}

问题是，“哪些是军用飞机?”，答案是 Θ 的一个子集：

$\{\theta_2,\theta_3\}$ = {轰炸机、战斗机}

同样地，问题“哪些是民用飞机?”，答案是集合：

$\{\theta_1\}$ = {客机}

该子集称为**单子集**（singleton set），因为它只有一个元素。

Θ的每一个子集都可解释为一个问题的可能答案。由于元素是互斥的，环境是完备的，因此只有一个子集是问题的正确答案。当然，并不是所有的问题都是有意义或值得回答的。不过，重要的一点是，环境的子集在所讨论的论域中全都是可能的有效答案。每个子集都可以看作是一个隐含的命题，如

正确答案是$\{\theta_1,\theta_2,\theta_3\}$

正确答案是$\{\theta_1,\theta_2\}$

等对所有的子集都适用。在这样的语句里“正确的答案是”是隐含在子集中的。

飞机环境的所有可能子集如图 5.6 所示，线条表示子集间的关系，A、B 和 F 分别是客机、轰炸机和战斗机的缩写。图画成一个层次格，Θ在顶部，空子集 Ø= {} 在底部。通常，空子集并不明确标出，因为它总是对应着答案假。又因为 Ø 不含元素，当与环境是完备的假设相矛盾时，就选择 Ø 作为一个答案，即

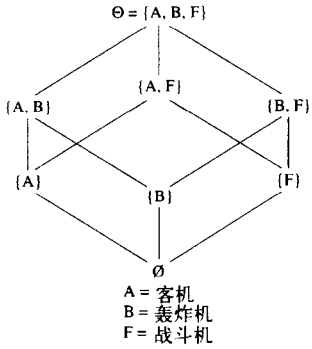


图 5.6 飞机环境的所有子集

“正确的答案是没有元素”

注意，图是一个格而不是一个树，这是因为子集结点可以有多于一个的父结点。又因为画的时候是从大子集到小子集画出，故格是分层的。例如：从 Θ 到 Ø 的一条路径表示了连接父子结点的子集的层次关系，如

$\emptyset \subset \{A\} \subset \{A,B\} \subset \{A,B,F\}$

正如第 2.10 节讨论的，两个集合 X 和 Y 的关系，比如

$X \subseteq Y$

表示 X 中所有的元素都是 Y 中的元素，写得更形式一点就是：

$X \subseteq Y = \{x \mid x \in X \rightarrow x \in Y\}$

这说明如果 x 是集合 X 的一个元素，则蕴含着 x 也是集合 Y 的一个元素。如果 $X \subseteq Y$，但 $X \neq Y$，则至少有一个元素属于 Y 但不属于 X，X 称为 Y 的真子集，记作

$X \subset Y$

当一个环境的元素可以解释为可能答案，但只有一个答案正确时，这个环境称为**识别框架**（frame of discernment）。术语“识别”的意思是说有可能从一个问题的所有可能答案中分辨出一个正确答案。如果这个答案不在框架里，则必须扩大这个框架以容纳增加的知识元素 θ_{N+1}，θ_{N+2}，等等。一个正确的答案要求集合是完备的，且子集是不相交的。

一个含有 N 个元素的集合有 2^N 个子集，包括它本身，这些子集定义了幂集（参看习题 2-11），记作 P（Θ）。因此，对飞机环境，

$\mathcal{P}(\Theta) = \{\emptyset,\{A\},\{B\},\{F\},\{A,B\},\{A,F\},\{B,F\},\{A,B,F\}\}$

环境的幂集中的元素可作为识别框架中可能问题的所有答案，这意味着 P（Θ）的元素与 Θ 的子集间存在一一对应关系。

Mass 函数与未知

在贝叶斯理论中，后验概率随证据的获取而改变。同样地，在 Dempster-Shafer 理论中，证据的信任也可以变化。在 Dempster-Shafer 理论中，习惯于考虑证据的信任度，这就好像考虑物体的

质量（mass）一样，即是说，证据的质量支持信任。把**证据的测度**（evidence measure）记为 m，类似于质量的总量。mass 的另一个说法是**基本概率分配**（basic probability assignment，bpa），或者有时简称为**基本分配**（basic assignment），它出自对概率密度和质量的公式分析。然而由于与概率论之间的潜在的混淆性，我们不用这些术语，而简单地称为 mass。之所以类比为对象的质量，是因为这样可把信任看成一个可以移动、分离、合并的量来考虑。这有助于我们把对象考虑成是由黏土组成的，于是其各部分可以移去也可以重新组合。

Dempster-Shafer 理论和概率论之间的基本区别在于对**未知**（ignorance）的处理。正如第 4 章所讨论的，即使在未知时，概率论也必须按等量概率分配。例如：如果没有先验知识，你会假设各种可能性的概率

$$P = \frac{1}{N}$$

其中 N 是可能性的个数，如第 4 章所述，P 的这种分配是在绝望时用的，用一个比较形象的术语称之为**无差别原理**（principle of indifference）。

当只有两种可能性时，应用无差别原理会发生极端情况。例如：有油和无油，记为 H 和 H′，像这种情况，即使无任何知识，由概率论

$$P(H) + P(H') = 1$$

都有 P = 50%，即是说：由于不允许未知，只要不支持就必定是否定。

如果不加思索地这样应用，将会导致荒谬的结果。例如：在你的房子里要么有油要么没油，根据无差别原理，若你完全没有其他知识，则在你的房子里有 50% 的概率是有油的。试想一下，50% 的机会有油是多么诱人，这提供了一个可以比任何合法投资更好的快速致富机会，既然有 50% 的机会有油，你还不赶快拿出所有的积蓄，租一套钻探设备，在你的厨房中开始钻探？

根据同样的推理，应用无差别原理和概率论，有 50% 的机会有：

钻石
和海盗一样多的财富
毛大衣
新鲜奶酪
你的下一次家庭作业

和其他任何你能想出的你房子内的东西（事实上，你可以这样就发财——上电视并告诉每个人只要花 9.95 美元买你的关于概率论的书就能致富。这也是不作清洁的最好理由。如果不知道那儿是什么，它就可以是任何东西，并同样有 50% 的可能性通过发现值钱的东西而变得富有）。

即使不使用无差别原理，规定：

$$P(H) + P(H') = 1$$

也使得必须为反面假设分配一个概率，即使没有任何证据支持它。正如在第 5.3 节中讨论的那样，对像医学知识那样有很多种信任的情况，这不是一个好的假定。然而，概率论所要求的就是这样的证据，不是支持假设就是否定假设。

Dempster-Shafer 理论并不强迫为未知或者反面假设分配一个信任。相反，你只需为那些你想分配信任的环境中的子集分配一个 mass。没有分配信任的特殊子集被认为是**无信任的**（no belief or nonbelief），它们只是仅仅与环境 Θ 有关。否定一个假设的信任是不信任，而不是无信任。

例如，假设一个识别友机或敌机的传感器（Identification Friend or Foe，IFF）从飞机雷达收发器上没有获得任何回应。IFF 是一个向飞机发射无线电信息的无线电收发器，如果该飞机是友机，它的收发器就会发回自己的身份代码作为回应。没有作出回应的飞机因违背约定而被认为是敌机。一个飞机未对 IFF 作出回应可能由于各种原因，如

- IFF 失灵

- 飞机的收发器失灵
- 飞机上没有 IFF
- IFF 信号阻塞
- 飞机被命令不要回复无线电信号

假设 IFF 未能得到回应表明有 0.7 的证据信任说明目标飞机是敌机。这里，敌机仅指轰炸机和战斗机。于是，只须给子集 {B，F} 分配 mass。

$$m_1(\{B,F\}) = 0.7$$

这里 m_1 指第一种 IFF 传感器的证据。

余下的信任留给环境，Θ，作为无信任。

$$m_1(\Theta) = 1 - 0.7 = 0.3$$

环境的幂集中每一个 mass 值大于 0 的集合称作一个**焦元**（focal element）。使用术语焦元是因为 m（x）>0的集合 X 是幂集中的一个元素且它聚焦或集中了有用的证据。

Dempster-Shafer 理论与概率论有一主要的区别，概率论假设

P(敌机) = 0.7

P(非敌机) = 1 - 0.7 = 0.3

在概率论中，如果敌机的信任是 0.7，则敌机的不信任必须是 0.3。而在 Dempster-Shafer 理论中，0.3 被认为是该环境的无信任 m（Θ），它表示有 0.3 的既不信任也不不信任。我们相信目标是敌机的度是 0.7，但保留 0.3 的判断给不信任和附加信任它是敌机。给环境 Θ 分配 0.3 并不是给 Θ 的子集分配任何值，意识到这一点非常重要，即使这些子集包含有敌机的子集 {B，F}、{B} 和 {F}。

回顾一下上一节的学生例子，

m{得了 A 并且毕业} = 0.7

并不自动意味

m{得了 A 但毕不了业} = 0.3

除非两者的值都是分配的。

如表 5.3 所示，mass 一般认为比概率有更大的自由度。

每个 mass 可形式地表示为从幂集中的元素映射到区间 0～1 之间的实数的一个函数。这仅意味着子集的信任可以取 0～1 之间的任何值。该映射可形式地表示为：

$$m: \mathcal{P}(\Theta) \rightarrow [0,1]$$

习惯上，空集的 mass 通常被定义为 0：

$$m(\varnothing) = 0$$

且幂集中所有子集 X 的 mass 和为 1。

$$\sum_{X \in \mathcal{P}(\Theta)} m(X) = 1$$

例如，在飞机环境中

$$\sum_{X \in \mathcal{P}(\Theta)} m(X) = m(\{B,F\}) + m(\theta) = 0.7 + 0.3 = 1$$

表 5.3 Dempster-Shafer 理论的 mass 与概率论的比较

Dempster-Shafer 理论	概率论
m(Θ)不必一定等于 1	$\sum_i P_i = 1$
如果 X⊆Y，不必 m(X)≤m(Y)	P(X)≤P(Y)
m(X)和 m(X′)之间不必有关系	P(X) + P(X′) = 1

组合证据

现在，让我们来看一个可附加证据的情形。我们将把所有的证据组合起来以获得一个更好的证据信任估计，为了说明这一点，让我们首先看一个例子，它是证据组合一般公式的一个特例。

假设第二种感应器识别目标为轰炸机的证据信任为0.9。此时，从不同感应器所得的mass如下：

$m_1(\{B,F\}) = 0.7 \quad m_1(\Theta) = 0.3$

$m_2(\{B\}) = 0.9 \quad m_2(\Theta) = 0.1$

这里 m_1、m_2 指第一种和第二种类型的感应器。

用以下特殊形式的**Dempster组合规则**（Dempster's Rule of Combination）可将这些证据组合并得到**组合mass**（combined mass）：

$$m_1 \oplus m_2(Z) = \sum_{X \cap Y = Z} m_1(X)\, m_2(Y)$$

这里求和是对所有交集 $X \cap Y = Z$ 的元素，⊕运算符称作**正交和**（orthogonal sum，direct sum），它定义为右边交集合的mass的积的和。Dempster规则组合mass以将最初的可能有冲突的证据表示为一个**一致**（consensus）的新mass，新的mass之所以是一致的是因为它试图通过仅包括交集中的mass以获得一致而不是不一致。交集表示证据中的公共元素。很重要的一点是该规则应被用于组合那些错误独立的证据，但这并不等同于独立地收集证据。

表5.4将飞机环境的mass值及交集合的积列于一张表中。每个交集后都列出其mass积的值。

表5.4 确认证据

	$m_2(\{B\}) = 0.9$	$m_2(\Theta) = 0.1$
$m_1(\{B,F\}) = 0.7$	{B} 0.63	{B,F} 0.07
$m_1(\Theta) = 0.3$	{B} 0.27	Θ 0.03

表中的各项用行和列的mass值交叉相乘而得出，如下所示，这里 T_{ij} 表示表中第 i 行第 j 列元素：

$T_{11}(\{B\}) = m_1(\{B,F\})\, m_2(\{B\}) = (0.7)(0.9) = 0.63$

$T_{21}(\{B\}) = m_1(\Theta)\, m_2(\{B\}) = (0.3)(0.9) = 0.27$

$T_{12}(\{B,F\}) = m_1(\{B,F\})\, m_2(\Theta) = (0.7)(0.1) = 0.07$

$T_{22}(\Theta) = m_1(\Theta)\, m_2(\Theta) = (0.3)(0.1) = 0.03.$

一旦各个mass值被算出，正如以上所示，然后根据Dempster规则，把公共交集的积相加：

$m_3(\{B\}) = m_1 \oplus m_2(\{B\}) = 0.63 + 0.27 = 0.90$ 轰炸机

$m_3(\{B,F\}) = m_1 \oplus m_2(\{B,F\}) = 0.07$ 轰炸机或战斗机

$m_3(\Theta) = m_1 \oplus m_2(\Theta) = 0.03$ 无信任

这里 $m_3(\{B\})$ 表示目标是轰炸机并且只是轰炸机的信任。但是，$m_3(\{B,F\})$ 和 $m_3(\Theta)$ 还含有附加的信息，由于它们的集合中也含有轰炸机，其正交和对目标是轰炸机的信任可能有作用，这一点似乎是合理的，于是它们的和 $0.07+0.03=0.1$ 可以被加到轰炸机的信任0.9上，作为可能是轰炸机的最大信任——合情信任。这里用证据的一个**信任范围**（range of belief）来代替单个信任值。信任范围从最小值0.9开始，0.9是已知的可能为轰炸机的信任，直到可能为轰炸机的最大合情信任 $0.9+0.1=1$。从最小值0.9到最大值1，真正的信任是0.9～1的某个数。

在证据推理中，证据被归纳成一个**证据区间**（evidential interval）。**下限**（lower bound）在证据推理中被称作**支持度**（support，Spt），在DEMPSTER-SHAFER理论中被称为Bel，**上限**（upper bound）被称作**合情度**（plausibility，Pls）。本例中，证据区间是[0.9，1]，下限是0.9，上限是1。支持度是证据的最低信任度，合情度是我们愿意给的最大信任度，一般而言，Bel和Pls的范围是 $0 \leqslant Bel \leqslant Pls \leqslant 1$。在Dempster-Shafer理论中，下限和上限有时被称为低或高概率。表5.5列举了一些常用的证据区间。

支持或**信任函数**（belief function），Bel，是一个集合和它所有的子集的信任总和。Bel也是支持一个集合的全部mass，它是用mass来定义的。

$$Bel(X) = \sum_{Y \subseteq X} m(Y)$$

例如，在飞机环境中，对第一种感应器，

$$Bel_1(\{B,F\}) = m_1(\{B,F\}) + m_1(\{B\}) + m_1(\{F\})$$

$$= 0.7 + 0 + 0 = 0.7$$

Bel 函数有时被称作信任测度，或简称为信任。然而，信任函数与 mass 非常不同，mass 是分配给单个集合的证据信任。例如，假设你拥有一辆福特汽车并听说警方正在寻找一辆银行抢劫犯用作逃跑工具的福特车，听说警方在寻找一辆福特车和听说警方在寻找你的福特车，这两者之间有很大区别。Mass 是一个集合而不是其子集的信任，但信任函数却是应用在集合和它所有的子集上。Bel 是信任总和，因此它比局部信任 mass 更具有全局性。由于 mass 和 Bel 之间的相互关系，Dempster-Shafer 理论也被称为信任函数理论。一般而言，Dempster 规则可解释为组合信任函数的一种方法。mass 和信任函数具有如下关系：

表 5.5 一些常用的证据区间

证据区间	含义
[1,1]	完全真
[0,0]	完全假
[0,1]	完全未知
[Bel,1]其中 0< Bel<1	倾向于支持
[0,Pls]其中 0< Pls<1	倾向于否定
[Bel,Pls]其中 0<Bel≤Pls<1	既倾向于支持又倾向于否定

$$m(X) = \sum_{Y \subseteq X} (-1)^{|X-Y|} Bel(Y)$$

其中，|X－Y|是集合

$$X - Y = \{x \mid x \in X \text{ and } x \notin Y\}$$

的**基数**（cardinality），即|X－Y|是集合 X－Y 中的元素个数。

由于信任函数是用 mass 来定义的，两个信任函数的组合也可用集合和其所有子集的 mass 的正交和来表示。例如，

$$Bel_1 \oplus Bel_2(\{B\}) = m_1 \oplus m_2(\{B\}) + m_1 \oplus m_2(\emptyset)$$

$$= 0.90 + 0 = 0.90$$

通常，空集的 mass 不写出来，因为它一般被定义为零。轰炸机－战斗机集合 {B，F} 的信任总和，要比上面有更多的子集。

$$Bel_1 \oplus Bel_2(\{B,F\})$$

$$= m_1 \oplus m_2(\{B,F\}) + m_1 \oplus m_2(\{B\}) + m_1 \oplus m_2(\{F\})$$

$$= 0.07 + 0.90 + 0 = 0.97$$

{B} 和 {F} 的项包括进来是因为它们是（{B，F}）的子集。从图 5.6 可知，（{B，F}）有子集 {B} 和 {F}，由于 {F} 没有给出 mass，于是 m（{F}）= 0，且它对总和无贡献。事实上，m（{F}）和其他值为 0 的 mass 根本没有列入表 5.4，因为任何与它们交叉相乘的结果都为 0。如果除了空集为 0 外，mass 被分配给 {A，B，F} 的每一个子集，则表 5.4 共会有 $(2^3-1)(2^3-1) = 7\times7 = 49$ 个项。

基于所有证据的 Θ 的组合信任函数如下：

$$Bel_1 \oplus Bel_2(\Theta) = m_1 \oplus m_2(\Theta) + m_1 \oplus m_2(\{B,F\}) + m_1 \oplus m_2(\{B\})$$

$$= 0.03 + 0.07 + 0.90 = 1$$

事实上，在各种情况下，Bel（Θ）= 1，因为所有 mass 的和必须总为 1。证据组合只是将 mass 重新分配给不同的子集。

集合 S 的证据区间，EI（S），可以用信任来定义：

$$EI(S) = [Bel(S), 1 - Bel(S')]$$

如果 S = {B}，那么 S′ = {A，F}，且

$$Bel(\{A,F\}) = m_1 \oplus m_2(\{A,F\}) + m_1 \oplus m_2(\{A\}) + m_1 \oplus m_2(\{F\})$$

$$= 0 + 0 + 0 = 0$$

由于它们都不是焦元，且对非焦元，mass为0。于是，{B} 的证据区间如下：

$$EI(\{B\}) = [0.90, 1 - 0]$$

$$= [0.90, 1]$$

类似的，如果 S = {B, F}，那么 S′ = {A}，且

$$Bel(\{A\}) = 0$$

这是因为 {A} 不是焦元。同样地，

$$Bel(\{B,F\}) = Bel_1 \oplus Bel_2(\{B,F\}) = 0.97$$

$$EI(\{B,F\}) = [0.97, 1 - 0] = [0.97, 1]$$

$$EI(\{A\}) = [0, 1]$$

这里，证据区间 [0, 1] 表示我们对 {A} 完全不知。

证据区间[信任总和,合情度]可以被表示为：

[支持证据,支持证据 + 未知]

概率论中，这个区间只是一个点，

[支持证据,支持证据]

这是因为不允许未知，即证据不是支持就是否定。例如，“如果手套不合适，你就必须放弃。”

合情度可定义为证据不否定 X 的程度：

$$Pls(X) = 1 - Bel(X') = 1 - \sum_{Y \subseteq X} m(X')$$

合情信任，Pls，是将信任延伸到一个绝对最大值，在该情况下未分配的信任 m (Θ) 也有可能对信任有贡献。这里 m (Θ) 可能是轰炸机、战斗机或客机。在合情的假定下，m (Θ) 被假设为对子集中的某一个有贡献。由于 {B} 是Θ的一个子集，m_1 (Θ) 的信任值 0.3 分配给轰炸机也是合情的。回忆第 4.15 节，一个合情信任要比可能信任略强一些，但并不需要有强有力的证据支持。另一点是Θ并不是能延伸信任给任一集合 X 的惟一集合，任何与 X 相交的集合，其补集都有此能力。

怀疑度 (dubiety, Dbt) 或**疑惑度** (doubt) 表示 X 不被相信或被否定的程度。**未知度** (ignorance, Igr) 表示集合支持 X 或 X′的程度，定义如下：

$$Dbt(X) = Bel(X') = 1 - Pls(X)$$

$$Igr(X) = Pls(X) - Bel(X)$$

信任的标准化

假设第三种感应器此时报告了一个客机的冲突证据：

$$m_3(\{A\}) = 0.95 \quad m(\Theta) = 0.05$$

表 5.6 列出了如何计算交叉积。

表 5.6 组合附加证据 m_3

	$m_1 \oplus m_2(\{B\})$ 0.90	$m_1 \oplus m_2(\{B,F\})$ 0.07	$m_1 \oplus m_2(\Theta)$ 0.03
$m_3(\{A\}) = 0.95$	Ø 0	Ø 0	{A} 0.0285
$m_3(\Theta) = 0.05$	{B} 0.045	{B,F} 0.0035	Θ 0.0015

之所以存在空集 Ø 是因为 {A} 和 {B} 无公共元素，{A} 和 {B, F} 也是同样，所以他们的叉乘是 0 而不是 0.855 或 0.0665。但是我们很快会看到这些定值在标准化中的作用，行与列的叉乘如下：

$m_2 \oplus m_2 \oplus m_3(\{A\}) = 0.0285$
$m_1 \oplus m_2 \oplus m_3(\{B\}) = 0.045$
$m_2 \oplus m_2 \oplus m_3(\{B,F\}) = 0.0035$
$m_1 \oplus m_2 \oplus m_3(\Theta) = 0.0015$
$m_2 \oplus m_2 \oplus m_3(\emptyset) = 0$ （根据空集的定义）

注意在本例中，所有 mass 的和小于 1：

$$\sum m_2 \oplus m_2 \oplus m_3(X) = 0.0285 + 0.045 + 0.0035 + 0.0015 = 0.0785$$

这里求和是对所有焦元。然而，和必须为 1，因为组合证据 $m_1 \oplus m_2 \oplus m_3$ 是有效的 mass，因此对所有焦元求和必须为 1，而事实上结果小于 1，这说明存在问题。

可通过对焦元的**标准化**（normalization）来解决该问题。即将各焦元除以

$1 - k$

这里，对任何集合 X 和 Y，κ 定义为：

$$\kappa = \sum_{X \cap Y = \emptyset} m_2(X)\, m_2(Y)$$

在我们的例子中，

$\kappa = 0.855 + 0.0665 = 0.9215$

因此

$1 - \kappa = 1 - 0.9215 = 0.0785$

对 $m_1 \oplus m_2 \oplus m_3$ 中每个焦元除以 $1 - \kappa$，得到标准化值：

$m_1 \oplus m_2 \oplus m_3\{A\} = 0.363$
$m_1 \oplus m_2 \oplus m_3\{B\} = 0.573$
$m_1 \oplus m_2 \oplus m_3\{B,F\} = 0.045$
$m_1 \oplus m_2 \oplus m_3(\Theta) = 0.019$

此时，{B} 的标准化信任总和为：

$Bel(\{B\}) = m_1 \oplus m_2(\{B\}) = 0.573$

注意到，正如我们所预料的一样，{A} 的一个证据较大地减低了 {B} 的信任度，从 0.90 到 0.573，大约减少了一半。

$$\begin{aligned} Bel(\{B\}') &= Bel(\{A,F\}) \\ &= m_2 \oplus m_2 \oplus m_3(\{A,F\}) + \\ &\quad m_2 \oplus m_2 \oplus m_3(\{A\}) + \\ &\quad m_1 \oplus m_2 \oplus m_3(\{F\}) \\ &= 0 + 0.363 + 0 = 0.363 \end{aligned}$$

因此此时的证据区间是：

$$\begin{aligned} EI(\{B\}) &= [Bel(\{B\}), 1 - Bel(\{B\}')] \\ &= [0.573, 1 - 0.363] \\ &= [0.573, 0.637] \end{aligned}$$

注意，由于 {A} 的冲突证据，{B} 的支持度和合情度已显著减少了。

Dempster 组合规则的一般形式为：

$$m_2 \oplus m_2(Z) = \frac{\sum_{X \cap Y = z} m_1(X)\, m_2(Y)}{1 - \kappa}$$

这里，为了方便，再次定义 κ。如果 $\kappa = 1$，则定义为无正交和。

$$\kappa = \sum_{X \cap Y = \emptyset} m_2(X)\, m_2(Y)$$

κ 表示**证据冲突**（evidential conflict）的额度。$\kappa = 0$ 说明完全兼容，1 说明完全冲突，值 $0 < \kappa < 1$ 表

示部分兼容。

移动 mass 和集合

对移动 mass 的类比有助于理解支持度和合情度。主要的概念如下：

- 支持度是分配给集合和其所有子集的 mass。
- 某一集合的 mass 能自由进入它的子集。
- 集合中的 mass 不能进入它的超集。
- 将 mass 从一个集合移入其子集，仅对子集的合情度有贡献，对它本身的支持度没有贡献。
- 环境 Θ 中的 mass 可移入任何一个子集，因为 Θ 在所有子集外。

在图 5.7（a）中，所有 mass 都假定在集合 X 内，因此

```
m(X) = 1
```

这意味着 X 的支持度为 1。X 的合情度也为 1，因为所有的 mass 都在 X 内，没有超集能对 mass 有贡献。于是

```
EI(X) = [1,1]
```

如果 m（X）= 0.5，则 EI（X）= [0.5，0.5]，一般，如果 m（X）= a，这里 a 是任意常数，则 EI（X）= [a，a]。

在图 5.7（b）中，假定 m（X）=0.6，m（Y）=0.4，这也是它们的支持度。X 的合情度为 0.6，因为 Y 的 mass 不能移给 X。但是，X 的 mass 可移入 Y，因为 Y 是 X 的一个子集，因此 Y 的合情度是 0.4+0.6=1。于是，X 和 Y 的证据区间如下：

```
EI({X}) = [0.6,0.6]
EI({Y}) = [0.4,1]
```

图 5.7（c）和（d）用于习题 5.4。

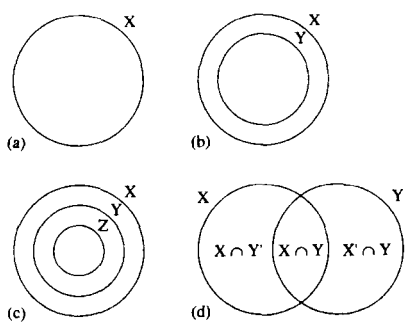


图 5.7 用集合来说明支持度和合情度

Dempster-Shafer 理论的困难

Dempster-Shafer 理论的一个困难是标准化，它会导致结果与我们预期的相矛盾。标准化之所以会造成问题是因为它忽略了所考虑的对象不存在时的信任。

Zadeh 引用的一个例子是关于两个医生，A 和 B，对一个病人病情的信任问题，病人病情的信任值如下：

m_A(脑膜炎) = 0.99

m_A(脑瘤) = 0.01

m_B(脑震荡) = 0.99

m_B(脑瘤) = 0.01

注意，两个医生都认为脑瘤的可能性很低，0.01，但对于主要问题的看法却很不一致。用 Dempster 组合规则可得脑瘤的组合信任为 1。该结果完全出乎意料且与我们的直觉相悖，因为两个医生都同意脑瘤是很不可能的。但无论其他概率是多少，都会得到同样的结果，脑瘤的组合信任为 1。

5.5 近似推理

本节我们将要讨论基于**模糊逻辑**（fuzzy logic）的不确定性理论，该理论主要关注定量和推理，这些定量和推理使用了自然语言且其中含有许多意义模糊的词，如高、热、危险、一点、很多等。人们经常使用模糊逻辑“IF … THEN”规则，例如“如果我的闹钟坏了，那么我可以多睡一会”。但我们都知道，多睡一会是非常不精确的描述，对每个人可能都不一样。模糊规则现在在计算机中有广泛应

用，无论是内嵌到一个工具中还是加到一个基本工具（如 CLIPS）中。

模糊逻辑是传统（布尔）逻辑的超集，被扩展用于处理部分真概念；即介于“完全正确”和“完全错误”之间的真值。模糊逻辑并不像听起来的那样，它并不是一种朦胧的、混浊的、含糊的思维方式；事实上恰恰相反。当事情非常复杂，难以理解时，事情就开始不确定。事情越复杂，就越不精确或模糊化。模糊逻辑提供了一个精确的方法去处理不确定性，而这种不确定性产生于人类行为的复杂性。

这个概念的最早描述是在 Zadeh 1965 年的论文中，它为 20 年后出现的模糊计算机系统和芯片提供了基本理论基础。他的理论引发了数学、工程和科学的全新分支。**软计算**（soft computing）的提出意味着计算并不是基于传统的二值逻辑（Roger 96）。软计算包括模糊逻辑、神经元网络和概率性推理。今天，术语概率性（probabilistic）在人工智能中不仅仅意味着经典概率理论，还包括了贝叶斯信任网，进化计算包括 DNA 计算、混沌理论和量子计算。

该理论不断扩展并被应用到许多领域，如空间物体的自动影像追踪（Giarratano 91）。模糊逻辑也和神经网络结合于许多应用中（Liu 04）。许多摄像机和照相机都使用了模糊逻辑。表 5.7 列出了模糊逻辑的一些主要应用。对于任何一个指定的应用，只需使用一个搜索引擎，你就可以得到大量信息。

模糊集合和自然语言

传统的表示一个对象属于一个集合是采用**特征函数**（characteristic function）的形式，有时也称为**识别函数**（discrimination function）。如果某对象是一集合的元素，那么它的特征函数值为 1，如果不是集合的元素，则它的特征函数值为 0，该定义可被概括成如下特征函数：

$$\mu_A(x) = \begin{cases} 1 & \text{if } x \text{ is an element of set } A \\ 0 & \text{if } x \text{ is not an element of set } A \end{cases}$$

此处 x 是某一论域 X 的元素（Ross 04）。

特征函数也可用函数映射的方式来定义（参见第 1.10 节的函数式程序设计）：

$$\mu_A(x) : X \rightarrow \{0,1\}$$

这说明特征函数将一个论域 X 映射到一个由 0 和 1 组成的集合。这个定义简单地表达了一个经典的概念，一个物体或者属于一个集合，或者不属于这一集合。这种集合称为**分明集**（crisp set），以区别于模糊集合。这种观点来源于亚里士多德的**二价**（bivalent）或**二值逻辑**（two-valued logic），它认为仅有真或假两种可能。

传统或经典逻辑总是把信息分类到二值模式，例如黑/白、真/假、是/否或所有/无一，而模糊逻辑把注意力放在“额外的中间”，尝试计算这样一个“灰色地带”，即构成人类日常生活中多数推理的部分真和部分假情形。它的建立基于一种假设，任何事物包括一个可变化的度——不管是否为真，年龄、美丽、财富、颜色、种族或任何其他事物都是由动态的人类行为和感知影响的。

表 5.7 模糊理论的一些应用

控制算法
医疗诊断
决策
经济
工程
环境
文学
运筹
模式识别
心理学
可靠性
安全性
自然科学

二值逻辑的问题在于，我们生活在一个模拟而非数字的世界中。在现实世界里，事物通常不是非此即彼的。仅仅是为了便于计算机组织才使用了基于二值逻辑的数字逻辑。计算机模拟理论的发展，如人工神经系统和模糊理论更精确地反映了现实世界。

在模糊集合中，一个对象可以部分属于一个集合。其隶属于模糊集合的程度可通过一个一般化的特征函数来度量。该函数称为**隶属**（membership）或**兼容函数**（compatibility function）。定义为：

$$\mu_A(x) : X \rightarrow [0,1]$$

虽然这个定义看起来与特征函数的定义极为相似，但实际上却非常不同。特征函数把 X 中的所有元素

映射为两个精确的元素之一：0 或 1。而隶属函数却把 X 映射到**闭区间**（interval）0 到 1，即 [0，1] 上的一个实数，

$$0 \leq \mu_A \leq 1$$

其中，0 表示不隶属，1 表示完全隶属于该集合。隶属函数的某个值，如 0.5，称为**隶属度**（grade of membership）。

虽然初次谈及一个元素部分属于一个集合有些奇怪，但事实上它比经典的二元集合要更自然。尽管许多人希望这样，但现实世界并非只是是非、黑白、对错、开关等。正如有许多不同深浅的灰，而不仅仅是黑白一样，现实世界也有很多不同程度的含义。只有债务和计算机代码才是精确的。

使用隶属函数，我们可以描述现实世界的情况。举一个简单的例子，如考虑多云的天气。使用分明集描述需要一个关于多云的判别定义。多云是指有一些云、许多云、布满云、局部布满云或者其他定义吗？同样，下雨，是指降水量为 1″、2″、3″或者某个降水量？

模糊集合和概念在自然语言中被普遍地使用。如：

"约翰是*高*的"
"天气是*热*的"
"音量调*高*一些"
"如果生面团*太硬*了，加*多点*水。"

斜体部分的内容是模糊集和量。所有这些模糊集和量都可以用模糊理论来表示和处理。特别是，曾在第 2.16 节中作为谓词逻辑的主要局限提出的大多（most）量词，可以用模糊逻辑解决。马上你就可以看到这点。

在自然语言里，术语含糊（vague）和模糊（fuzzy）这两个词有时可以通用。然而在模糊理论中，这两个词有着明显的差别。一个**模糊命题**（fuzzy proposition）是指包含有如"高"这样的词语，高是模糊集合 TALL 的标识符。与经典的命题如"约翰高五尺"，表示一个或真或假的陈述不同，模糊命题的真实有程度之分。例如，模糊命题"约翰高"，可能为真，也可能是某种程度的真：有一点真、几分真、相当真、很真等等。一个模糊真值称为**模糊限定词**（fuzzy qualifier），它可以作为一个模糊集合使用，也可以用于改变一个模糊集合。与分明命题不允许量词不同，模糊命题可以有**模糊量词**（fuzzy quantifier），如大多、很多、通常等等，和经典情况一样，这些量词在语句和命题之间并无分别。

术语含糊是用于不完全信息的情况下。如"约翰在某处"，如果没有足够的信息以供判断，则它就是含糊的。一个模糊命题，如"他高"，如果我们不知道他指的是谁，也是含糊的。含糊也有程度之分，一个命题，如"约翰高"比"他高"要不含糊些，但如果我们不知道是指哪个约翰，则依然是含糊的。

自然语言中使用了许多模糊词汇，如表 5.8 所示。这些词的含义可以用模糊集合来定义，这一点下文将很快涉及。复合的术语，如表 5.9 所示，也可以用模糊理论来定义和处理。

诚然，难以想像一个对象仅是部分属于某个集合，另一方法是把隶属函数作为表示对象具有某种属性的程度。这种属性程度的概念可用与隶属函数含义相同的兼容函数表示。术语兼容意味着一个对象是如何地符合某些属性，实际上也很适用于描述模糊集合。然而隶属函数是书面上最常用的，因此，我们也将采用。当考虑模糊集合时，你会发现把模糊元素作为一个如第 2 章所述的对象-属性-值三元组是大有帮助的。对分明集而言，只有对象－属性，这是因为假定值要么为 1，要么为 0，即在分明集中，元素或者属于集合，或者不属于集合。对模糊集合而言，值可为 0 到 1 之间的任意数。

为了说明模糊集合的概念，再考虑先前的例子：

"约翰高"

如果他是一个成年人，图 5.8 给出了一个可能的隶属函数。任何一个 7 英尺以上高度的人都被认为其隶属函数为 1.0。任何一个低于 5 英尺的人都被认为不属于"高"模糊集合，所以其隶属函数是 0，在 5 英尺和 7 英尺之间，隶属函数随高度单调增加。注意"非常"这个词如何增加了一个不同的隶属函数，得到的平均高度的曲线会是怎样。

表 5.8 一些自然语言中的模糊术语

高
热
低
中等
高度
很
不
一点
几个
少许
许多
更多
大多
大约
近乎
左派

表 5.9 复合模糊自然语言术语

差不多低
近乎一样低
不低
不很低
一样低
中等高
比低略高
低于中等
最高
自由左派
激进自由左派

这个特定的隶属函数仅是许多可能的函数之一。对普通人、篮球运动员、骑师等不同人，隶属函数有很大不同。例如，一个 5 英尺高的骑师，在某种程度上可以认为是高的，尽管图 5.8 中一个 5 英尺高的人的隶属函数为 0。

依赖于应用，隶属函数可以由一个人或一群人的观点而建立。在专家系统中，隶属函数由模型化了的专家意见建立。虽然对高的看法似乎不能在专家系统中模型化，但许多其他观点却可以。例如一笔贷款的风险、不明飞机的敌意、产品质量，应聘者对一项工作的适合程度等等。注意，像这样的一些观点不是简单的是或不是。虽然也可设置一个阈值来判断是或不是，然而真正的问题在于这个分明阈值的有效性。例如，是否一个人应该被拒绝给予抵押贷款因为他的收入为 $29 999.99 而阈值为 $30 000.00?

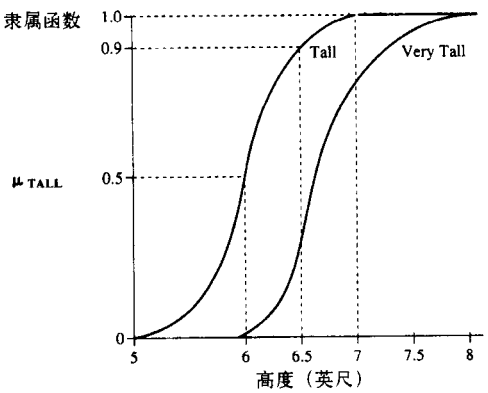


图 5.8 模糊集合“高”的一个隶属函数

直觉上，对一群人的隶属函数可以用选票的方法来考虑。假如一群路人被问及他们认为称得上高的最小身高。可能没有一个人会认为低于 5 英尺是高，同样的，可能每个人都认为 7 英尺或以上是高。在 5 英尺和 7 英尺之间，会认为某一身高为高的人数比率，与图 5-8 曲线所示极为相似。随着身高从 5 英尺增加到 6 英尺，会认为该身高为高的人越来越多。对于这个特定的隶属函数，身高的**交叉点**（crossover point)为 6 英尺。交叉点是 $\mu=0.5$ 的点。用我们的话说，就是 50% 的人会认为一个 6 英尺以上的人是高的。在 6.5 英尺处，同意的人达 90%。7 英尺以上，每个人都同意。所以隶属函数在 1 处是水平的。

尽管这个例子是以一群人的观点选票的方式给出，但隶属函数事实上并不是概率分布，认识这一

点很重要。正如我们在第4章讨论过的，概率用于对相同或同一对象的重复观察，虽然这群人中每个人，当再次问及同一问题时，可能给出同样的答案，但这些答案具有似然性，因为它们反映的是一个人的信任。

S-**函数**（S-function）是常常作为模糊集合中隶属函数的一个数学函数。其定义如下：

$$S(x;\ \alpha,\beta,\gamma) = \begin{cases} 0 & \text{for } x \leqslant \alpha \\ 2\left(\dfrac{x-\alpha}{\gamma-\alpha}\right)^2 & \text{for } \alpha \leqslant x \leqslant \beta \\ 1-2\left(\dfrac{x-\gamma}{\gamma-\alpha}\right)^2 & \text{for } \beta \leqslant x \leqslant \gamma \\ 1 & \text{for } x \geqslant \gamma \end{cases}$$

S-函数的图形如图5.9所示：

在定义模糊函数如“高”时，S-函数是一个有用的工具。它不是保持一个定义隶属函数的数据表，而是简洁地用一个公式来表达。在这个定义中，a，β，γ是参数，它们可调整以适合所要求的隶属数据。依赖于所给定的隶属数据，我们可给出某些a，β，γ值精确地适合这些数据，或只是近似地适合。当然隶属函数可以简单地定义为S-函数，而不必参照任何表格值。其他函数，如三角函数，也可以根据应用需要来定义。

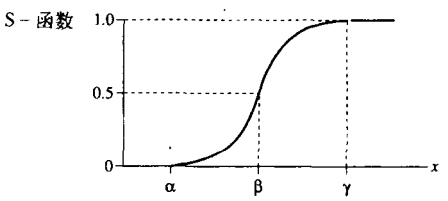


图5.9 S-函数

S函数在x≤α时，值为0，在x≥γ时，值为1，在这两种情况下，其曲线都是水平的。在α和γ之间，函数是x的二次函数。从图5.9可见，β参数对应着交叉点0.5，且其值为（α+γ）/2。对图5.8“高”隶属函数，S函数如下：

$$(1)\quad S(x;\ 5,\ 6,\ 7) = \begin{cases} 0 & \text{for } x \leqslant 5 \\ 2\left(\dfrac{x-5}{7-5}\right)^2 = \dfrac{(x-5)^2}{2} & \text{for } 5 \leqslant x \leqslant 6 \\ 1-2\left(\dfrac{x-7}{7-5}\right)^2 = 1-\dfrac{(x-7)^2}{2} & \text{for } 6 \leqslant x \leqslant 7 \\ 1 & \text{for } x \geqslant 7 \end{cases}$$

模糊命题“X近似于γ”的隶属函数如图5.10所示。该隶属函数可以表达所有近似于某个特定值γ的数，如“X近似于6”，这里，X可定义为｛5.9，6，6.1｝。隶属函数可表示为

$$\mu_{CLOSE}(x) = \frac{1}{1+\left(\dfrac{x-\gamma}{\beta}\right)^2}$$

交叉点为：

$$x = \gamma \pm \beta$$

如图5.10所示，在交叉点，参数β的值为曲线**宽度的一半**（half-width）。β较大的值对应较宽的曲线，较小的值对应较窄的曲线。较大的β意味着数值必须更接近于γ才能有一个特别大的隶属值。注意在这种定义中，隶属函数只有在无穷远处才趋近于0。

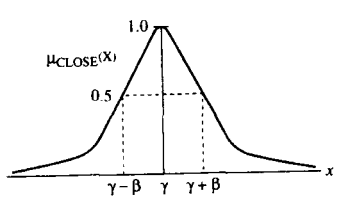


图5.10 模糊命题“X近似于γ”的一个隶属函数

下述函数也给出了一个类似的曲线并且在特定点处为0：

$$\Pi(x;\ \beta,\ \gamma) = \begin{cases} S(x;\ \gamma-\beta,\ \gamma-\beta/2,\ \gamma) & \text{for } x \le \gamma \\ 1 - S(x;\ \gamma,\ \gamma+\beta/2,\ \gamma+\beta) & \text{for } x \ge \gamma \end{cases}$$

该 Π-**函数**（Π-function）的图形如图 5.11 所示。注意，此时 β 参数在交叉点是**带宽**（bandwidth）或**整个宽度**（total width）。在点

$$x = \gamma \pm \beta$$

处，Π-函数值为 0，交叉点位于

$$x = \gamma \pm \frac{\beta}{2}$$

处。

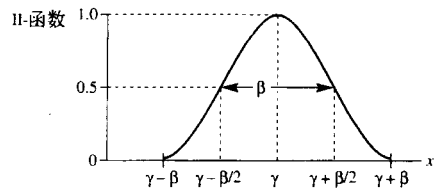


图 5.11 Π-函数

除了连续函数外，隶属函数也可以是一个有限的元素集合。例如，高度的论域可定义为：

$$U = \{5, 5.5, 6, 6.5, 7, 7.5, 8\}$$

“高”的一个有限模糊子集可定义为：

$$TALL = \{0/5, 0.125/5.5, 0.5/6, 0.875/6.5, 1/7, 1/7.5, 1/8\}$$

在这个模糊集合中，符号“/”用来分隔隶属度和相应高度值。注意“/”并不意味通常模糊集合表示法中的除。模糊集中 $\mu(x) > 0$ 的元素组成模糊集的**支撑**（support）集。“高”的支撑集是除了 0/5 外的所有元素。

N 个元素的有限模糊子集用标准的模糊表示法表示为模糊单子 μ_i/x_i 的并，此处的“+”符号是布尔连接符并。

$$(2)\quad F = \mu_1/x_1 + \mu_2/x_2 + \ldots \mu_N/x_N$$

$$F = \sum_{i=1}^{N} \mu_i/x_i$$

$$F = \bigcup_{i=1}^{N} \mu_i/x_i$$

在有些文章中“/”符号并不写出，于是 F 也可写为：

$$(3)\quad F = \mu_1 x_1 + \mu_2 x_2 + \ldots \mu_n x_N$$

$$F = \sum_{i=1}^{N} \mu_i x_i$$

$$F = \bigcup_{i=1}^{N} \mu_i x_i$$

两个等式（2）（3）都表示了 N 个元素的有限模糊子集。等式（3）可用于紧凑格式的模糊集合。然而当涉及数据时，等式（3）的格式难以明白。如

$$F = 0.127 + 0.385$$

没有“/”分隔符，难以判断隶属度是 0.1，0.3 还是 0.12，0.38 或别的值，这就是为什么当涉及数字时，

$$F = 0.1/27 + 0.38/5$$

的表示法要更好一些。

模糊集合的支撑集 F 是一个论域 X 的子集，定义为：

$$\text{support}(F) = \{x \mid x \in X \text{ and } \mu_F(x) > 0\}$$

支撑集通常简写为 supp，如

$$\text{supp}(F)$$

支撑集的优点在于模糊集合 F 可以被写为：

$$F = \{ \mu_F(x)/x \mid x \in supp(F) \}$$

这意味着仅是那些隶属函数大于 0 的模糊元素属于 F。所以“高”可以被写为除了 0/5 元素外的集合，因为这一元素不属于支撑集。

$$TALL = \{0.125/5.5, 0.5/6, 0.875/6.5, 1/7, 1/7.5, 1/8\}$$

虽然这仅仅减少了一个元素。但它对于含有许多 0 隶属度元素的模糊集而言，作用是明显的。从另一方面来说，你可能对那些元素含有 0 隶属度感兴趣。

与支撑集相关的概念是 **α 截集**（α-cut）。集合的 α 截集是论域的一个非模糊集合，它的元素具有大于或等于某一值 α 的隶属函数。

$$F_\alpha = \{ x \mid \mu_F(x) \geq \alpha \} \quad \text{for } 0 < \alpha \leq 1$$

“高”的一些 α 截集：

$$TALL_{0.1} = \{ 5.5, 6, 6.5, 7, 7.5, 8 \}$$

$$TALL_{0.5} = \{ 6, 6.5, 7, 7.5, 8 \}$$

$$TALL_{0.8} = \{ 6.5, 7, 7.5, 8 \}$$

$$TALL_1 = \{ 7, 7.5, 8 \}$$

注意一个集合的 α 截集是支撑集的子集。α 的值可以任意选择，但通常选取那些可得到所需论域子集的值。

另一个模糊集合常用的术语是**顶点**（height），它是元素的最大隶属度。对“高”集合而言，最大隶属度为 1。如果模糊集合的某个元素达到最大的可能隶属度，则称该集合是**标准化的**（normalized）。通常隶属度定义在闭区间 [0, 1] 上，所以最大的可能隶属度为 1。但是，隶属度也可以定义在别的区间上，所以最大隶属度并不必然为 1。

论域上的一个任意的**连续**（continuum）模糊子集通常用积分形式表示。术语连续是针对实数集。积分表示模糊集合单子 μ（x）/x 的并。例如，我们可以定义：

$$TALL = \int_X \mu_{TALL}(x)/x$$

$$= \int_5^8 \mu_{TALL}(x)/x$$

$$= \int_5^6 \frac{(x-5)^2}{2}/x + \int_6^7 \left[1 - \frac{(x-7)^2}{2}\right]/x + \int_7^8 1/x$$

其中，使用了 S 函数作为隶属函数。在这个公式中，分隔积分的符号“+”代表并，是布尔逻辑表示，而不是算术加。

有许多不同类型的模糊集合。论域 X 上的基本**类型 1 模糊子集**（type 1 fuzzy subset），定义如下：

$$\mu_F: X \rightarrow [0,1]$$

即类型 1 模糊子集简单地定义为其隶属函数是实闭区间 0 到 1 上的一个数值。例如，

$$TALL = 0.125/5.5 + 0.5/6 + 0.875/6.5 + 1/7 + 1/7.5 + 1/8$$

是一个类型 1 的集合，因为其隶属度都是 [0, 1] 上的实数。同样式（1）中

$$\mu_{TALL}(x) = S(x; 5, 6, 7)$$

也是一个类型 1 模糊子集。

一般，**类型 N 模糊子集**（type N fuzzy subset）定义为从论域到类型 N-1 模糊子集上关于 μ_F 的一个映射。例如，一个**类型 2 的模糊子集**（type 2 fuzzy subset）可以用类型 1 子集来定义。对“高”，一个类型 2 的模糊集合可以是：

$\mu_{TALL}(5)$ = LESS THAN AVERAGE(低于平均值)

$\mu_{TALL}(6)$ = AVERAGE(平均)

$\mu_{TALL}(7)$ = GREATER THAN AVERAGE(高于平均值)

这里，低于平均值，平均，高于平均值都是类型 1 的模糊子集。它们可定义为如下的模糊子集：

$\mu_{\text{LESS THAN AVERAGE}}(x)$ = 1 − S(x; 4.5, 5, 5.5)
$\mu_{\text{AVERAGE}}(x)$ = $\prod$ (x; 1, 5.5)
$\mu_{\text{GREATER THAN AVERAGE}}(x)$ = S(x; 5.5, 6, 6.5)

模糊集合的操作

普通分明集是一个隶属函数为 {0, 1} 的模糊集合的特例。当模糊性为 0，模糊集合就是分明集时，所有模糊集合的定义、证明和理论都应与其相容。这样，模糊集合理论才能较分明集理论有更为广泛的应用，因而也就可以处理与主观看法有关的情形。模糊集合的基本思想是通过一模糊元素集合来描述模糊的真实世界中的概念，比如“高”，而无须一个极端的二元阈值。以下归纳了论域 X 中的某些模糊集合操作。

- 集合相等（set equality）

A = B

$\mu_A(x) = \mu_B(x)$ 对所有 $x \in X$

- 集合补（set complement）

A′

$\mu_{A'}(x) = 1 - \mu_A(x)$ 对所有 $x \in X$

图 5.12 例示了一个集合的模糊补集。这个定义的依据是由 Bellman 和 Giertz 给出的。

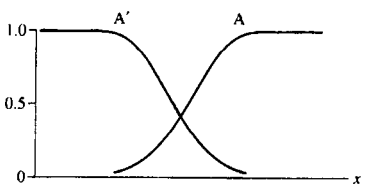


图 5.12 模糊补集

- 集合包含（set containment）

$A \subseteq B$

模糊集合 A 包含于 B 或是 B 的子集，当且仅当

$\mu_A(x) \leqslant \mu_B(x)$ 对所有 $x \in X$

模糊集 A 是 B 的**真子集**（proper subset）当 A 是 B 的子集且不相等。即：

至少有一个 $x \in X$ 使得 $\mu_A(x) \leqslant \mu_B(x)$ 且 $\mu_A(x) < \mu_B(x)$

- 集合并（set union）

$A \cup B$

$\mu_{A \cup B}(x) = \vee(\mu_A(x), \mu_B(x))$ 对所有 $x \in X$

其中，**连接运算符**（join operator）$\vee$ 表示参量中最大的。

- 集合交（set intersection）

$A \cap B$

$\mu_{A \cap B}(x) = \wedge(\mu_A(x), \mu_B(x))$ 对所有 $x \in X$

其中，**相交运算符**（meet operator）$\wedge$ 表示参量中最小的。

在第4.13节讨论模糊逻辑在PROSPECTOR的组合前件中的应用时，你已遇到过并和交运算。Bellman给出了使用最大、最小函数求解并和交的理由。

有了这些定义，可知分明集的一些定律如交换律、结合律等同样适合于模糊集合，但排中律和矛盾律除外。于是，对模糊集合A，可能有：

$$A \cup A' \neq \mathscr{U} \text{ and } A \cap A' \neq \emptyset$$

其中∅表示空集，$\mathscr{U}$表示论域。由于模糊集可没有定义域，因此，如果μ(x)定义在闭区间[0，1]上，且$\mu_{A'}(x) = 1 - \mu_A(x)$，则对交集的唯一约束是：

$$A \cap A' = \min(\mu_A(x), \mu_{A'}(x)) \leq 0.5$$

同样，对并集的唯一约束是：

$$A \cup A' = \max(\mu_A(x), \mu_{A'}(x)) \geq 0.5$$

由于模糊集合A和A′可重叠，所以它们不是完全覆盖论域的。

如果排中律和矛盾律被定义为适合模糊集合，那么幂等律和分配律将不再成立。对幂等律，即是：

$$A \cup A \neq A \qquad A \cap A \neq A$$

由于集合的模糊性，所以排中律和矛盾律不成立的观点似乎更为合理，因此幂等律是成立的。附录C中列出有用的集合属性。

- 集合积（set product）

$$A\ B$$

$$\mu_{AB}(x) = \mu_A(x)\mu_B(x)$$

- 集合的幂（power of a set）

$$A^N$$

$$\mu_{A^N}(x) = (\mu_A(x))^N$$

- 概率和（probabilistic sum）

$$A \dagger B$$

$$\mu_{A \dagger B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x)\mu_B(x)$$

$$= 1 - (1 - \mu_A(x))(1 - \mu_B(x))$$

其中，“+”和“-”是普通算术运算符。

- 有界和（bounded sum）或凸并（bold union）

$$A \oplus B$$

$$\mu_{A \oplus B}(x) = \wedge(1, (\mu_A(x) + \mu_B(x)))$$

其中，“∧”是求最小值函数，“+”是算术运算符。

- 有界积（bounded product）或凸交（bold intersection）

$$A \odot B$$

$$\mu_{A \odot B}(x) = \vee(0, (\mu_A(x) + \mu_B(x) - 1))$$

其中，“∨”是求最大值函数，“+”是算术运算符。有界和与积不满足幂等律、分配律和吸收律，但满足交换律、结合律、德·摩根律，A⊕∪=∪，A⊙∅=∅，以及排中律、矛盾律（见附录C的总结）。

- 有界差（bounded difference）

$$A\ |-|\ B$$

$$\mu_{A\ -|\ B}(x) = \vee(0, (\mu_A(x) - \mu_B(x)))$$

其中，分隔μ_A和μ_B的“-”是算术差运算符，A|-|B代表那些A中有而B中无的元素，补集可以用论域和有界差表示为：

$$A' = \mathscr{U}\ |-|\ A$$

• 集中（concentration）

CON(A)

$$\mu_{CON(A)}(x) = (\mu_A(x))^2$$

CON 运算通过减小那些隶属度较小的元素的隶属度来集中模糊元素。图 5.13 例示了集中运算。它和后面的 DIL、NORM 以及 INT 运算在通常的集合运算中都没有相应的运算。集中运算符可被粗略地理解为语言上的 Very，即是说，对模糊集 F，有

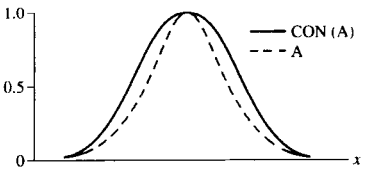


图 5.13 模糊集合的集中

$$\text{Very } F = F^2$$

例如，“高”集合

$$TALL = 0.125/5 + 0.5/6 + 0.875/6.5 + 1/7 + 1/7.5 + 1/8$$

那么

$$\text{Very TALL} = 0.016/5 + 0.25/6 + 0.76/6.5 + 1/7 + 1/7.5 + 1/8$$

注意，除了隶属度为 1 的元素外，其他隶属度是如何减小的，其直接影响就是，使 Very TALL 模糊集合包含更小的隶属度。

• 扩张（dilation）

DIL(A)

$$\mu_{DIL(A)}(x) = (\mu_A(x))^{0.5}$$

DIL 运算通过增大那些隶属度较小元素的隶属度来扩张模糊元素。图 5.14 例示了 DIL 运算。注意，由于选择了 2 次方和 0.5 次方，它是集中运算的逆运算。

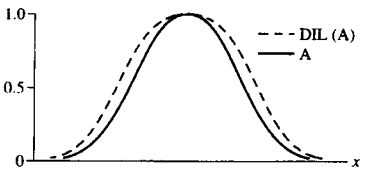


图 5.14 模糊集合的扩张

$$A = DIL(CON(A)) = CON(DIL(A))$$

扩张运算符可粗略地类似语言上的修饰词 More Or Less。因此对任何模糊集合 F，有

$$\text{More Or Less } F = F^{0.5} = DIL(F)$$

• 强化（intensification）

INT(A)

$$\mu_{INT(A)}(x) = \begin{cases} 2(\mu_A(x))^2 & \text{for } 0 \le \mu_A(x) \le 0.5 \\ 1 - 2(1 - \mu_A(x))^2 & \text{for } 0.5 \le \mu_A(x) \le 1 \end{cases}$$

INT 运算好比一幅图的对比度增加。如图 5.15 所示，强化运算增大了那些交叉点以内的隶属度并减小了那些交叉点以外的隶属度。作为电子学上的一个分析，考虑交点作为定义信号的带宽，强化运算放大了带宽内的信号并减小了带宽外的“噪音”。因此，强化运算提高了交叉点内外隶属度的对比。

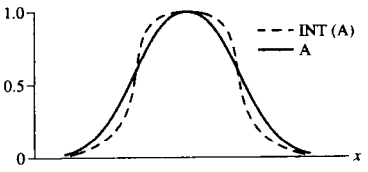


图 5.15 模糊集合的强化

• 标准化（normalization）

NORM(A)

$$\mu_{NORM(A)}(x) = \mu_A(x)/\max\{\mu_A(x)\}$$

其中，max（）返回所有元素 x 的隶属度的最大值。如果最大值＜1，那么所有的隶属度都会增加，如果 max = 1，那么隶属度不变。

模糊关系

关系（relation）也是描述模糊集合的一个重要概念。关系的直观意义是指元素之间的某些联系。

下面是关系的一些例子：

Bob 和 Ellis 是*朋友*。
洛杉矶和纽约*很远*。
1、2、3 和 4 比 100*小得多*。
1、2 和 3 是*小的*数。
苹果和桔子是一种*圆形水果*。

其中斜体字为模糊词。

分明集 N 的**笛卡儿积**（Cartesian product）定义为这样的分明集，它的每个元素都是一个有序 N 元组（x_1，x_2，x_3，$\cdots x_N$），其中每个 x_i 均是分明集 X_i 的一个元素。对于两个集合 A 和 B，有

$$A \times B = \{(a,b) \mid a \in A \text{ 且 } b \in B\}$$

定义

A = {巧克力,草莓}
B = {馅饼,牛奶,糖果}

那么笛卡儿积

A×B = {(巧克力,馅饼),(巧克力,牛奶),(巧克力,糖果),
(草莓,馅饼),(草莓,牛奶),(草莓,糖果)}

注意，当 A 和 B 有不同元素时，一般 B×A 不等于 A×B，即是说，一般 (a，b) ≠ (b，a)。称 A×B 定义了一个**二元变量**（binary variable）(a，b)。

关系 R 是笛卡儿积的一个子集。例如，关系 R= “包含馅饼”可定义为 A×B 的子集：

R = {(巧克力,馅饼),(草莓,馅饼)}

而关系 R= “包含甜品”可定义为：

R = {(巧克力,馅饼),(巧克力,糖果),(草莓,馅饼),(草莓,糖果)}

关系有时也称为**映射**（mapping），因为它联系了一个域和另一个域的元素。在 A×B 中，关系是一个 A→B 的映射，这里“→”表示映射。A 中的巧克力映射或联系 B 中的馅饼和糖果，对草莓也是同样。

如果 X 和 Y 是论域，那么

$$R = \{ \mu_R(x,y)/(x,y) \mid (x,y) \subseteq X \times Y\}$$

是 X×Y 上的一个模糊关系。

模糊关系（fuzzy relation）实际上是笛卡儿积论域上的一个模糊子集。当处理模糊图像时，模糊集的另一个定义很有用。

N 个集合的模糊关系定义为包括隶属度的分明关系的一个扩展，即

$$R = \{\mu_R(x_1, x_2, \ldots x_N)/(x_1, x_2, \ldots x_N) \mid x_i \in X_i, i = 1, .. N)\}$$

它将隶属度与每个 N－元组联系起来。对于馅饼例子的二元关系，另一个定义为：

R = {0.9/(巧克力、馅饼),0.2/(草莓、馅饼)}

这个模糊关系表明了一个人喜欢巧克力馅饼更甚于草莓馅饼。

表示一个关系的方便方法是通过矩阵，对于“包含甜品”关系

$$M_R = \begin{array}{c} \\ \text{巧克力} \\ \text{草莓} \end{array} \begin{array}{c} \begin{array}{cc} \text{馅饼} & \text{糖果} \end{array} \\ \begin{bmatrix} 0.9 & 0.7 \\ 0.2 & 0.1 \end{bmatrix} \end{array}$$

这里 M_R 是表示模糊关系的矩阵。注意，对于分明集，M_R 由只包含 0 和 1 表示你完全喜欢和不喜欢有序组（调味品，甜品）（这大概就是现实世界是真正模糊而不是分明的最好证明）。

关系**合成**（composition）是一个关系应用于另一个关系的直接结果，如两个二元关系 P 和 Q，它们的关系合成是二元关系 R，

$$R(A,C) = Q(A,B) \circ P(B,C)$$

这里

R(A,C)是 A,C 之间的关系，

Q(A,B)是 A、B 间的关系，

P(B,C)是 B、C 间的关系，

A、B、C 是集合，

"o"是**合成运算符**（composition operator），关系 R 就是首先应用关系 P，然后再应用关系 Q。用隶属度表示即是：

$$R = \{\mu_R(a,c)/(a,c) \mid a \in A,\ c \in C\}$$

这里 μ_R 定义如下：

$$\mu_R(a,c) = \bigvee_{b \in B} [\mu_Q(a,b) \wedge \mu_P(b,c)]$$

$$= \max_{b \in B} [\min (\mu_Q(a,b), \mu_P(b,c))]$$

这种合成关系通常用**最大-最小矩阵积**（max-min matrix product）或简单地称为**最大-最小**（max-min）来定义，即把矩阵乘法运算中的加法和乘法换为最大和最小函数。例如，定义

$$Q = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \quad P = \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.0 & 0.4 \end{bmatrix}$$

则合成关系 R 为

$$R = Q \circ P = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix} \circ \begin{bmatrix} 0.1 & 0.3 & 0.5 \\ 0.2 & 0.0 & 0.4 \end{bmatrix}$$

$$R = Q \circ P = \begin{bmatrix} \max(0.1,0.2) & \max(0.1,0.0) & \max(0.1,0.2) \\ \max(0.1,0.2) & \max(0.3,0.0) & \max(0.3,0.4) \end{bmatrix}$$

$$= \begin{bmatrix} 0.2 & 0.1 & 0.2 \\ 0.2 & 0.3 & 0.4 \end{bmatrix}$$

其他一些常用的关系运算符定义还有**最大积**（max-product）和**关系连接**（relational join）。

模糊关系对于近似推理有着重要作用，这一点将在后面看到。**投影**（projection）关系是另一个有用的模糊集合概念，基本上，投影可消去指定的元素。在给出形式定义前，让我们先看一下表 5.10 的简单例子。

表 5.10 给出了前面关系 R 的投影，为方便起见，这里行和列给出了标记符 x_i 和 y_i。注意，标记为第一投影的列中包含了每行中的最大隶属度。同样，标记为第二投影的行中包含了每列中的最大隶属度。右下方标记为总投影处的值 0.4 是整个关系的最大隶属度。

表 5.10 关系及其投影

	y_1	y_2	y_3	第一投影	
x_1	0.2	0.1	0.2	0.2	
x_2	0.2	0.3	0.4	0.4	
第二投影	0.2	0.3	0.4	0.4	总投影

关系 R 可用 x_i 和 y_i 表示成下式：

$$R = 0.2/x_1,y_1 + 0.1/x_1,y_2 + 0.2/x_1,y_3 + 0.2/x_2,y_1 + 0.3/x_2,y_2 + 0.4/x_2,y_3$$

第一投影用 R_1 表示，它可通过去掉笛卡儿对 x_i，y_j 中除第一个外的所有元素而获得，这里，最左的元素定义为第一个元素。

$$R_1 = 0.2/x_1 + 0.1/x_1 + 0.2/x_1 + 0.2/x_2 + 0.3/x_2 + 0.4/x_2$$

R_1 等式在投影后可更进一步化简，因为"+"代表并运算符，因此只有最大的模糊元素被保留下来。

$R_1 = 0.2/x_1 + 0.4/x_2$

同样，第二投影仅包含每一笛卡儿对的第2点 y_j。

$R_2 = 0.2/y_1 + 0.1/y_2 + 0.2/y_3 + 0.2/y_1 + 0.3/y_2 + 0.4/y_3$

该式通过并运算可简化为下式：

$R_2 = 0.2/y_1 + 0.3/y_2 + 0.4/y_3$

一般情况下，包含N个笛卡儿点的关系的投影，将去掉除那些要投影的点以外的所有成分。例如，如果关系有N点，那么 R_{136} 将删去除第1、第3和第6点以外的所有点。

对一个在论域 $X \times Y$ 上的关系，

$R = \{\mu_R(x, y)/(x, y)\}$ 对所有 $(x,y) \in X \times Y$

第一投影定义为：

$proj(R;X) = R_1$

其中，

$$R_1 = \{\max_y \mu_R(x,y)/x \mid (x,y) \in X \times Y\}$$

且对 y 求最大值。同样，

$proj(R;Y) = R_2$

其中

$$R_2 = \{\max_x \mu_R(x,y)/y \mid (x,y) \in X \times Y\}$$

这里，是对 x 求最大值。

投影关系的**圆柱扩展**（cylindrical extension）定义为与投影相容的最大模糊关系。圆柱投影有点类似于投影，因为它将投影值（最大）扩展到对所有其他元素，例如

$proj(R;X) = R_1 = 0.2/x_1 + 0.4/x_2$

于是

$$R_1 = 0.2/x_1,y_1 + 0.2/x_1,y_2 + 0.2/x_1,y_3 + 0.4/x_2,y_1 + 0.4/x_2,y_2 + 0.4/x_2,y_3$$

$$R_2 = 0.2/y_1,x_1 + 0.2/y_1,x_2 + 0.3/y_2,x_1 + 0.3/y_2,x_1 + 0.3/y_2,x_2 + 0.4/y_3,x_1 + 0.4/y_3,x_2$$

也即，代换第二个变量，变为 $0.2/x_1$，{所有变量} $+0.4|x_2$，{所有变量}。由于投影给出了 $\max\mu$，所以圆柱扩展是与投影相容的最大关系。

对于前面的例子，

$$\overline{R_1} = \begin{bmatrix} 0.2 & 0.2 & 0.2 \\ 0.4 & 0.4 & 0.4 \end{bmatrix}$$

$$\overline{R_2} = \begin{bmatrix} 0.2 & 0.3 & 0.4 \\ 0.2 & 0.3 & 0.4 \end{bmatrix}$$

这里投影上面的横线表示是投影的圆柱扩展。

合成可用投影和圆柱扩展来定义。对于一个定义在论域 $\mathcal{U}_1 \times \mathcal{U}_2$ 上的二元关系R以及定义在 $\mathcal{U}_2 \times \mathcal{U}_3$ 上的S，其合成为：

$R \circ S = proj(\overline{R} \cap \overline{S}; U_1 \times U_3)$

语言变量

模糊集合的一个重要应用是**计算语言学**（computational linguistics），目的是对自然语言的语句进行

计算，就像对逻辑语句进行逻辑运算一样。模糊集合和**语言变量**（linguistic variable）可用于量化自然语言的含义，因而可用来处理指定了值的语言变量，这些指定值表示了自然语言或人工语言中的单词、短语或句子。表 5.11 列举了一些语言变量和可赋予它们的典型值。

虽然可为语言变量定义一个值，如红色，但它们本质上是非常主观的。例如，眼睛可分辨的红色对应一个频率范围，而不仅仅只是一个频率。另外，如蓝绿色，它到底是蓝色还是绿色？

语言变量常常用于启发式规则。但是，正如表 5.12 前两条规则所示，这些变量可能是隐含的。

表 5.11 典型值

语言变量	典型值
高度	矮小的、短的、一般的、高的、巨大的
数量	几乎无、几个、少数、许多
生命历程	婴儿、初学走路的孩子、小孩、青少年、成人
颜色	红色、蓝色、绿色、黄色、橙色
亮度	微暗的、弱的、正常的、明亮的、强烈的
甜品	馅饼、蛋糕、冰激凌、阿拉斯加面包

表 5.12 一些启发式规则，其中某些含有隐含语言变量

如果声音太低，那么调大音量
如果太热，那么加点凉水
如果压力太高，那么打开安全阀
如果利率上升，那么买债券
如果利率下降，那么买股票

头两条规则含有的隐含语言变量是图像质量和水温。

某些语言变量如图像质量可以是二阶模糊集合。例如，图像质量的值可以是：

颜色，色度，亮度，噪音，浓度

这里面的每一个值都可以是一个语言变量，且其值又是一个模糊集合。因此，可为语言变量安排一个层次，它对应着模糊集合的阶，最终直到一阶模糊集合，如“高”和“亮”，它们定义为闭区间 [0，1] 上的一个映射，从而语言变量的值就成为一个数字范围。

语言变量 L 的**项集**（term-set），T（L），是一个其可能的值集，例如：

T(馅饼) = 巧克力 + 苹果 + 草莓 + 胡桃

T（馅饼）中的每一个名字都是一个模糊集的名称。它们可能是其他子集的并集，例如：

巧克力 = 半甜巧克力 + 牛奶巧克力 + 荷兰巧克力 + 黑巧克力 + ……

巧克力模糊集合的另一个定义可包含**限定词**（hedge），用以修改集合的意义。例如，一种巧克力的模糊集可定义为：

巧克力 = Very 巧克力 + Very Very 巧克力 + More Or Less 巧克力 +
Slightly 巧克力 + Plus 巧克力 + Not Very 巧克力 + ……

标准限定词可用某些模糊集合运算符和模糊集合 F 来定义，如表 5.13 所示。

表 5.13 一些语言限定词和算子

限 定 词	算 子 定 义
Very F	$CON(F) = F^2$
More Or Less F	$DIL(F) = F^{0.5}$
Plus F	$F^{1.25}$
Not F	$1 - F$
Not Very F	$1 - CON(F)$
Slightly F	INT[NORM (PLUS F And NOT (VERY F))]

正如上面的 Not Very F 限定词，其他一些复合限定词也可通过组合运算符来形成。注意，在 Slightly 限定词中的“And”是模糊集运算符交，∩，它的运算对象是 Plus F 和 Not (Very F)。

语言变量 Appetite（胃口）的层次如图 5.16 所示。假定 LIGTH（轻）和 HEAVY（重）模糊集合是 S-函数，而 MODERATE（中等）集合为Π-函数。

注意 LIGHT 与 MODERATE 甚至 LIGHT 与 HEAVY 之间有重合，在经典的分明集中是不会有重合的，因为这些集合都不会相交，即 LIGHT 不可能是 MODERATE 或 HEAVY。然而，模糊集合间通常无明显界限（除非定义）。

限定集合用在模糊集合边界内的虚线显示。限定词 Very 作为语言变量的一个修饰词使用，从而得到模糊集合 Very LIGHT，Very MODERATE 和 Very HEAVY。

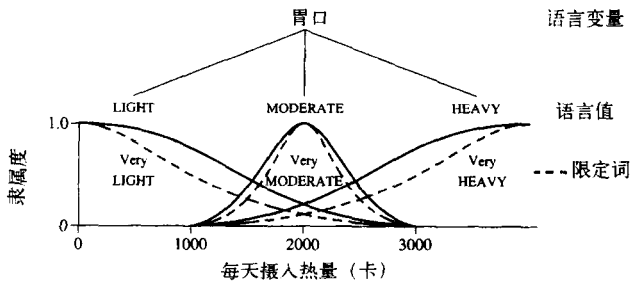


图 5.16 语言变量“胃口”及其值

一个语言变量必须要有合法的语法和语义，这可通过模糊集合或规则来说明。**语法规则**（syntactic rule）定义了 T（L）中的一个合式表达式。例如，项集

$$T(Age) = \{OLD, \text{Very OLD}, \text{Very Very OLD}, \ldots\}$$

可由下面的规则产生，

$$T^{i+1} = \{OLD\} \cup \{\text{Very } T^{i}\}$$

例如，

$T^0 = \varnothing$(空集)

$T^1 = \{OLD\}$

$T^2 = \{OLD, \text{Very OLD}\}$

$T^3 = \{OLD, \text{Very OLD}, \text{Very Very OLD}\}$

T（L）中的**语义规则**（semantic rule）通过模糊集合定义了 L 中项 L_i 的**含义**（meaning）。例如，Very OLD 的语义规则可定义为：

$$\text{Very OLD} = \mu^2_{OLD}$$

这里，隶属函数可定义为如下的 S－函数：

$$\mu_{OLD}(X) = S(x;\ 60,\ 70,\ 80)$$

基本项（primary term）是这样的一些项，如 YOUNG（年轻）、OLD（年老）、CHOCOLATE（巧克力）、STRAWBERRY（草莓）等，其含义必须先于限定词定义，限定词修改了基本术语的意义并得到一个项集的另外项，如 Very YOUNG、Very OLD、Very CHOCOLATE、Slightly CHOCOLATE 等，限定模糊集合的含义可由适当的算子给出。例如：

$$\mu_{\text{Very CHOCOLATE}} = \mu^2_{\text{CHOCOLATE}}$$

$$\mu_{\text{Not CHOCOLATE}} = 1 - \mu_{\text{CHOCOLATE}}$$

$$\mu_{\text{More Or Less CHOCOLATE}} = \mu^{0.5}_{\text{CHOCOLATE}}$$

$$\mu_{\text{Not Very CHOCOLATE}} = 1 - \mu_{\text{Very CHOCOLATE}}$$

正如传统的语法可定义为 BNF 范式一样（参见第 2.2 小节），模糊语法（fuzzy grammar）也可如

此，实际上，第 2.2 节中的语法使用了“heavy”这个修饰词：

〈形容词〉→ heavy

从而生成一个**模糊产生式**（fuzzy production）：

```
an eater was the heavy man
```

在上述产生式中，heavy 是模糊集合 man 的限定词。也许你不认为 man 是一个模糊集，可是，boy 是怎么成为 man 的呢？在某些文化观念中，12 岁或经过宗教仪式后，boy 就成为 man，一些国家认为 18 岁或 21 岁后 boy 才成为 man。对于 17～19 岁的人，报刊倾向于将被控犯了罪的人定义为 man，而做了一些值得赞赏的事的人定义为 youth。军队中的男性有时被认为是 boy，有时被认为是 man，尤其是在政治演说中。

模糊语法可通过在 BNF 中增加非终止符详细说明，例如：

```
〈范围短语〉::=〈被限定的基本项〉TO
              〈被限定的基本项〉
〈被限定的基本项〉::= 〈限定词〉〈基本项〉|〈基本项〉
〈限定词〉::= Very |More Or Less |A Little
〈基本项〉::= SHORT|MEDIUM|TALL
```

由上，可产生以下语句：

```
Very SHORT TO Very TALL
A Little TALL TO Very SHORT
More Or Less MEDIUM TO TALL
```

语言变量概念的一个新奇应用是由东京工学院 Sugeno 制造的模糊车。该车使用一个基于模糊逻辑的控制系统，可在矩形的轨道上自动操作。它可停泊在一个指定地方，且可从其他案例中学习。在控制车移动的规则中使用了语言变量。此外，还有其他许多种类的模糊逻辑控制系统，它们可控制设备及工业生产过程，如生产水泥的干燥炉。

扩张原理

扩张原理（extension principle）是模糊理论中非常重要的概念，它描述了如何从一个给定的分明函数域扩展到包含模糊集。运用此原理，任何一个数学、自然科学、工程、商业等领域的普通或分明函数都可以扩展到模糊集上，它使模糊集合可应用到所有领域。

设 f 是一个普通的，从论域 X 映射到 Y 的函数，如果 F 是 X 的一个如下模糊子集，

$$F = \int_X \mu_F(x)/x$$

那么扩张原理定义了映射函数 $f(x)$ 下模糊集合 F 的映像：

$$f(F) = \int_X \mu_F(x)/f(x)$$

例如，设 $f(x)$ 为一个求平方的分明函数：

$$f(x) = x^2$$

扩张原理描述了如何实现一个模糊集合的平方函数：

$$f(F) = \int_X \mu_F(x)/f(x) = \int_X \mu_F(x)/x^2$$

例如，定义论域 X，Y 为实闭区间 [0，1000]，模糊集合

```
F =0.3/15 +0.8/20 + 1/30
```

则扩张原理定义 f（F）为：

$$f(F) \approx \int_x \mu_F(x)/f(x)$$

$$= 0.3/f(15) + 0.8/f(20) + 1/f(30)$$

$$f(F) = 0.3/225 + 0.8/400 + 1/900$$

模糊逻辑

正如经典逻辑组成了传统专家系统的基础一样，模糊逻辑组成了模糊专家系统的基础，除了能处理不确定性外，模糊专家系统还能模拟**常识推理**（commonsense reasoning），这是传统专家系统很难做到的。但是常识推理的问题是大量本体信息会被我们认为是理所当然的。

经典逻辑的根本缺陷在于二值——真、假的严格限制。正如我们在第 2 章和第 3 章中所讨论的，这有利也有弊。主要优点在于基于二值逻辑的系统易于演绎建模，从而使得推理是精确的。而主要缺点在于现实世界中很少东西是真正二值的。现实世界是模拟而不是数字世界。

从亚里士多德时代起就已知道二值逻辑的缺陷。虽然亚里士多德首先建立了演绎推理规则和排中律，但他认识到关于未来的命题在其发生之前不是实际真或假的。

许多不同的多值逻辑理论已经建立起来，包括 Lukasiewicz、Bochvar、Kleene、Heyting、Reichenbach 等逻辑，其中常用的是那些基于三值 TRUE、FALSE、UNKNOWN 的逻辑。这些**三价**（trivalent）或**三值逻辑**（three-valued logic）常常用 1，0，1/2 来分别表示 TRUE、FALSE、UNKNOWN。

几种关于 N 值的一般逻辑理论也已发展起来，这里 N 为一个大于等于 2 的任意整数。Lukasiewicz 在 20 世纪 30 年代发展了第一个 N 值逻辑。在 N 值逻辑中，真值集 T_N 假定在闭区间 [0，1] 中等分，即

$$T_N = \left\{\frac{i}{N-1}\right\} \quad for\ 0 \le i < N$$

例如，

$$T_2 = \{0, 1\} \qquad T_3 = \{0, 1/2, 1\}$$

N 值逻辑，这里 N≥2 的某些 **Lukasiewicz 逻辑运算符**（Lukasiewicz logic operator）的定义见表 5.14。正如习题 5.13 所示，当 N=2 时，它们退化成标准逻辑。注意，－、min、max 运算符与模糊逻辑中的一样。

表 5.14 Lukasiewicz N-值逻辑的基本运算符

x'	$= 1 - x$
$x \wedge y$	$= \min(x, y)$
$x \vee y$	$= \max(x, y)$
$x \rightarrow y$	$= \min(1, 1 + y - x)$

N-值 Lukasiewicz 逻辑或 **L-逻辑**（L-logic），记作 L_N，这里 N 是其真值的个数。L_2 就是经典的二值逻辑，当 $N = \infty$，L_∞ 定义了**无穷值逻辑**（infinite-valued logic），其真值集 T_∞ 定义在有理数上，无穷值逻辑也可定义在连续的实数集上。术语无穷值逻辑常常指真值定义在实数集 [0，1] 上的逻辑，并记为 L_1。

但是，L_1 并不等同于 N = 1 的**单值逻辑**（unary logic）。由于 L-逻辑只定义在 N≥2 上，所以单值逻辑根本就不是 L-逻辑。L_1 中的 1 实际上是 $\aleph_1$（读作 aleph 1）的缩写，它是实基数。$\aleph_1$ 不是一个有限数，而是一个**超限数**（transfinite number）。该理论是 Cantor 为无限数计算而首先引进的。Cantor 还定义了不同的无穷阶。例如，最小的超限数为 $\aleph_0$，它是自然数的基数。$\aleph_1$ 的无穷阶比 $\aleph_0$ 高，因为对每一个自然数，都有无穷多个实数与之对应。

模糊逻辑可看作多值逻辑的扩展。但是，模糊逻辑的目的和应用不同，因为模糊逻辑是**近似推理**（approximate reasoning），而不是精确的多值推理。本质上，近似或**模糊推理**（fuzzy reasoning）是在一组可能不精确的前件下推出的一个可能不精确的结论。人们非常熟悉近似推理，因为它是现实生活中最常用的推理，也是许多启发式规则的基础。下面是一些近似推理的启发式规则例子：

如果电视声音太小
则增加一点儿音量

如果电视声音太大引起邻居抱怨
则将音量调小一点儿

如果交通拥堵
则开始大量开辟车道

如果你由于吃香蕉片、馅饼、冰激凌、蛋糕而变得太胖
则减少吃香蕉的数量

近似推理既不精确，也不像纯猜测那样完全不精确。就像模糊逻辑与近似推理、二值逻辑与精确推理的关系一样，近似推理与自然语言推理的关系特别密切。关于精确推理的一个例子是演绎与定理证明，见第 3 章中的讨论。

有许多种不同种类的模糊集理论、模糊逻辑及近似推理。这里所讨论的模糊逻辑基于 Zadeh 的近似推理理论，该理论所使用的模糊逻辑则基于 Lukasiewicz 的 L_1 逻辑。在这种模糊逻辑中，真值是可以最终用模糊集合来表示的语言变量。

基于表 5.14 中 Lukasiewicz 逻辑运算符的模糊逻辑运算符定义在表 5.15 中给出。x（A）是［0，1］中的一个数值真值，表示命题“x is A”的真值，它也可解释为隶属度 μ_A（x）。

作为模糊逻辑运算符的一个例子，设模糊集合 TRUE 定义如下：

```
TRUE = 0.1/0.1 + 0.3/0.5 + 1/0.8
```

使用表 5.15 中的运算符，则

```
FALSE = 1 - TRUE
      = (1 - 0.1)/0.1 + (1 - 0.3)/0.5 + (1 - 1)/0.8
      = 0.9/0.1 + 0.7/0.5
```

对限定词 Very 使用 CON 运算符有：

```
Very TRUE = 0.01/0.1 + 0.09/0.5 + 1/0.8
Very FALSE = 0.81/0.1 + 0.49/0.5
```

表 5.15 一些模糊逻辑运算符

x(A′)	= x(NOT A)	$=1-\mu_A(x)$
x(A) ∧ x(B)	= x(A AND B)	$=\min(\mu_A(x),\mu_B(x))$
x(A) ∨ x(B)	= x(A OR B)	$=\max(\mu_A(x),\mu_B(x))$
x(A)→x(B)	= x((A→B)	$=x((\sim A)\vee B)=\max[(1-\mu_A(x)),\mu_B(x)]$

模糊规则

作为模糊逻辑运算符的一个简单例子，考虑模式识别问题。模式可以是需要进行质量控制的被检物体，如生产部件、需分拣的新鲜水果等（Harris 00）。其他一些重要的模式识别问题还有医学图像、矿物中地震数据、石油勘探等。

表 5.16 给出了一些假设数据，表示对应某些图像的模糊集合导弹、战斗机、客机的隶属度。图像可能是从远处的系统中生成的，由于目标移动、方向、噪音等因素，而具有不确定性。

每幅图像的模糊集并表示了目标确认的总不确定性。图 5.17 显示了表 5.16 中 10 幅图像的模糊集并。当然，在现实环境中可能还有更多其他可能的图像，而不仅仅是这 10 幅依赖于系统分辨率及目标

距离的图像。此外，除了系统硬件的不确定性，导弹、战斗机、客机的初始模糊集也具有不确定性，隶属度是基于典型导弹、战斗机、客机的知识而主观指定的。在现实环境中，对每一个这样的初始集都还有许多其他的类型。

表 5.16 图像的隶属度

图像	隶属度		
	导弹	战斗机	客机
1	1.0	0.0	0.0
2	0.9	0.0	0.1
3	0.4	0.3	0.2
4	0.2	0.3	0.5
5	0.1	0.2	0.7
6	0.1	0.6	0.4
7	0.0	0.7	0.2
8	0.0	0.0	1.0
9	0.0	0.8	0.2
10	0.0	1.0	0.0

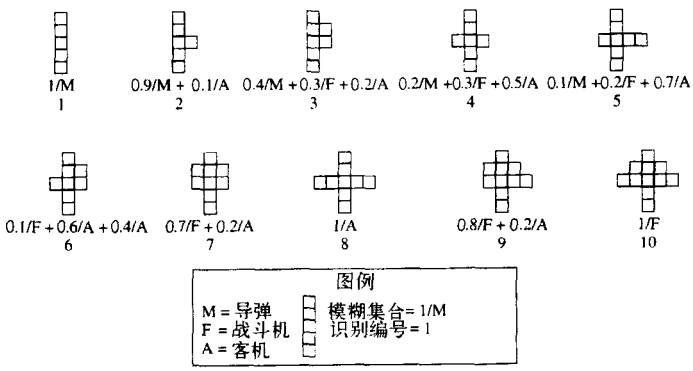


图 5.17 飞机识别的模糊集合

图 5.17 中的模糊集并可认为表示了如下形式的规则：

```
IF E THEN H
```

这里，E 为所观察到的图像，H 是模糊集并。例如：

```
IF IMAGE4 THEN TARGET (0.2/M + 0.3/F + 0.5/A)
```

其中，括号中的表达式是目标的模糊集并。该规则也可表示为：

```
IF IMAGE4 THEN TARGET4
```

其中，

```
TARGET4 = 0.2/M + 0.3/F + 0.5/A
```

假设还有时间继续另一次观察，并设图像 6 被观测到。相应的规则为：

```
IF IMAGE6 THEN TARGET6
```

其中，

```
TARGET6 = 0.1/M + 0.6/F + 0.4/A
```

则可获得目标的总模糊集：

```
TARGET = TARGET4 + TARGET6
```

“+”表示取并集，于是，

```
TARGET = 0.2/M + 0.3/F + 0.5/A + 0.1/M + 0.6/F + 0.4/A

TARGET = 0.2/M + 0.6/F + 0.5/A
```

这里，对每一个元素只取最大隶属度值。

如果把隶属度值最大的元素当作最有可能的目标，则目标最可能是战斗机，这是因为其隶属度值最大为 0.6。但是，若客机的隶属度值也为 0.6，那么目标为战斗机或客机的可能性是同样的。

一般给定 N 次观察及规则：

```
IF E1 THEN H1
IF E2 THEN H2
      .
      .
      .
IF EN THEN HN
```

这里，所有 H_i 都与一个共同的假设 H 有关，于是 H 的隶属度可由这些假设的并来决定，即：

$$\mu_H = \max(\mu_{H1}, \mu_{H2}, \ldots \mu_{HN})$$

注意，这个结果不同于确定性因子和 Dempster-Shafer 理论。假设 H 的 μ_H 称为 H 的**真值**（truth value）。

下面这个假定是合理的，即假设的真值不可能超过其证据的真值。体现在规则上，即是后件的真值不可能超过前件的真值。于是：

$$\mu_H = \max(\mu_{H_1}, \mu_{H_2}, \ldots \mu_{H_N})$$

$$= \max[\min(\mu_{E_1}), \min(\mu_{E_2}), \ldots \min(\mu_{E_N}))$$

其中，每个 E_i 可能会是一个模糊表达式。例如，E_1 可能定义为：

$$E_1 = E_A \text{ AND } (E_B \text{ OR NOT } E_C)$$

并且，可用模糊逻辑运算符来计算这些表达式。即是：

$$\mu_{E_1} = \min(\mu_{E_A}, \max(\mu_{E_B}, 1 - \mu_{E_C}))$$

前件的组合隶属度称为**前件真值**（truth value of the antecedent）。这类似于 PROSPECTOR 规则前件中的部分证据。事实上，回想一下 PROSPECTOR 中前件证据是使用了一种“特别”的模糊逻辑方法来组合的。现在，你可看到这种组合的根据就是模糊理论的推理合成规则。

最大－最小合成

上面关于 H 的等式就是模糊逻辑中的**最大－最小推理合成规则**（max-min compositional rule of inference）。在每个规则只有两个证据项的简单情形下，

$$\text{IF } E_{11} \text{ AND } E_{12} \text{ THEN } H_1$$
$$\text{IF } E_{21} \text{ AND } E_{22} \text{ THEN } H_2$$
$$\vdots$$
$$\text{IF } E_{N1} \text{ AND } E_{N2} \text{ THEN } H_N$$

则最大-最小推理合成规则为：

$$\mu_H = \max[\min(\mu_{E_{11}}, \mu_{E_{12}}), \min(\mu_{E_{21}}, \mu_{E_{22}}), \ldots \min(\mu_{E_{N1}}, \mu_{E_{N2}})]$$

类似地可扩展到具有更多的证据 E_{i3}、E_{i4} 等。

作为推理合成规则的另一个例子，来看一下它是如何应用于关系上的。定义一个模糊关系 R（x，y）= APPROXIMATELY EQUAL（大约相等），这是一个关于人的体重，范围在 120～160 磅之间的二元关系，如表 5.17 所示。

表 5.17 人的体重上的大约相等关系

	Y				
x	120	130	140	150	160
120	1.0	0.7	0.4	0.2	0.0
130	0.7	1.0	0.6	0.5	0.2
140	0.4	0.6	1.0	0.8	0.5
150	0.2	0.5	0.8	1.0	0.8
160	0.0	0.2	0.5	0.8	1.0

该表是这样构成的，隶属度基于这两个值与平均值之间的差按照 0.075% 递减。比如，如果 x 和 y 的值为 150、130，则平均值为 140。x、y 与平均值的差的绝对值与平均值的比为 10/140 = 7.1%，此值乘以常数因子 −0.075%得出 − 0.5。于是 150，130 对应的最后隶属度为 1 − 0.5 = 0.5。另一个容易计算的简单定义是隶属度按照每相差 10 递减 0.3，但是这个定义对较小的体重如 10 和 20 会得到不合理的结果，因为此时它们大约相等的隶属度为 0.7。

注意，关系 R（x，y）作为一个**模糊约束**（fuzzy restriction）是如何作用在具有非零隶属度 R（x，y）的两个值 x，y 上的。模糊关系是一**种弹性约束**（elastic constraint），它允许隶属度有一定范围，而不像分明关系那样要求有严格的约束。事实上，大约相等并不能用非模糊逻辑来定义。分明关系的严格约束要求值要么确切相等，要么确切不等。即 x 要么确切等于 y，要么不等于 y。

作为模糊约束的一个例子，考虑如下命题 p：

$$p = X \text{ is } F$$

其中 F 是一些模糊集合，它作为一个模糊约束作用在语言变量 X 上。下面是具有如上形式的一些模糊命题例子：

约翰是高的
sue 是超过 21 岁的
混凝土是太厚的
Target 是友好的
X 是一个近似于 100 的数
馅饼是水果的

上面最后一个例子，模糊集可定义为：

水果 = 1/苹果 + 1/橘子

它表明水果的组成类型。注意，在模糊理论中你可用任何有意义的方式增加苹果和橘子的隶属度值。

现在让我们来定义一个模糊约束 R_1（x）。例如，模糊集合 HEAVY（重）可被定义为：

$R_1(x)$ = HEAVY = 0.6/140 + 0.8/150 + 1/160

推理合成规则定义为作用于 y 上的一个模糊限制：

$R_3(y) = R_1(x) \circ R_2(x,y)$

这里合成运算符 o 是最大-最小运算：

$$\max_x \min(\mu_1(x), \mu_2(x,y))$$

R_3（y）也可解释为是求解关于

$R_1(x)$
$R_2(x,y)$

的关系等式而得到的一个解答。即是说，给定 x 的模糊约束，x 和 y 的模糊约束，可推出 y 的模糊约束。这样的推导将涉及**模糊约束演算**（calculus of fuzzy restriction），模糊约束演算是近似推理的基础。

运用这些定义，R_3（y）可计算如下：

$R_3(y) = R_1(x) \circ R_2(x,y)$

$$R_3(y) = [0.0 \quad 0.0 \quad 0.6 \quad 0.8 \quad 1.0] \circ \begin{bmatrix} 1.0 & 0.7 & 0.4 & 0.2 & 0.0 \\ 0.7 & 1.0 & 0.6 & 0.5 & 0.2 \\ 0.4 & 0.6 & 1.0 & 0.8 & 0.5 \\ 0.2 & 0.5 & 0.8 & 1.0 & 0.8 \\ 0.0 & 0.2 & 0.5 & 0.8 & 1.0 \end{bmatrix}$$

这里 R_1（x）代表一个行向量。R_3（y）的非零元素计算如下：

$R_3(120) = \max \min[(0.6,0.4),(0.8,0.2)]$

$= \max(0.4,0.2) = 0.4$

$R_3(130) = \max \min[(0.6,0.6),(0.8,0.5),(1,0.2)]$

$= \max(0.6,0.5,0.2) = 0.6$

$R_3(140) = \max \min[(0.6,1),(0.8,0.8),(1,0.5)]$

$= \max(0.6,0.8,0.5) = 0.8$

$R_3(150) = \max \min[(0.6,0.8),(0.8,1),(1,0.8)]$

$= \max(0.6,0.8,0.8) = 0.8$

$R_3(160) = \max \min[(0.6,0.5),(0.8,0.8),(1,1)]$

$= \max(0.5,0.8,1) = 1$

于是，如果 R_1（x）是 HEAVY，则

$R_3(y) = 0.4/120 + 0.6/130 + 0.8/140 + 0.8/150 + 1/160$

可以用语言粗略的描述为：R_3（y）是 MORE OR LESS HEAVY（差不多重）。

关系 R_3（y）仅仅只是一个语言上的近似，这是因为把 DIL 运算 $\mu^{0.5}$ 作用到 HEAVY 上，实际得到

$DIL(HEAVY) = 0.8/140 + 0.9/150 + 1/160$

此时，关于 120 和 130 的项被丢失，但是，隶属度 $\mu \geqslant 0.8$ 的元素仍然表示得很好，这证明了 MORE OR LESS HEAVY 只是一个粗略近似。因此，max-min 合成推理表示了模糊语言关系：

MORE OR LESS HEAVY = HEAVY ∘ APPROXIMATELY EQUAL

注意，这些关系依赖于模糊集合的定义及集合的语言标记。既然它依赖于模糊集合、关系、标记的定义，因此从绝对的意义上来说，它是不真的。但是，一旦这些基本定义建立起来，模糊理论就为处理这些表达提供了一种形式的、一致的机制。这一点非常重要，因为语言的处理和项的含义便因此而建立在合理的理论基础上，而不是依赖于“特别”的方法或个人的直觉理解。

最大值和瞬时方法

选择具有最大隶属度的元素来决定规则后件真值的方法叫**最大值方法**（maximum method）。另一种方法称作**瞬时方法**（moments method），它类似于计算物体惯性的第一时刻的方法来指定规则后件的真值。瞬时方法的基本思想是考虑所有规则的后件而不只是考虑值最大的那个后件。正如前面所提到的，如果多条规则的后件都有相同的最大值，则用最大值方法就很容易产生歧义。

作为瞬时方法的一个简单例子，考虑下面配制混凝土的模糊产生式规则集合。

R1：IF 混合太湿
　　THEN 增加沙和石子
R2：IF 混合正好
　　THEN 保持原状
R3：IF 混合太干
　　THEN 减少沙和石子

混凝土是用水泥、水、沙和石子按一定比例混合而成的。**混合量**（mix）是一个用来决定所需应用最佳比例的测试量。选择比例的原则一般是根据所要求混凝土的强度来定。然而，由于当地使用的原料、石子大小、环境条件和其他因素的不同，选择的标准也会有所不同。在开始建造价值 10 000 000 美元的楼房前做一个混合量测试是一个好的主意。

决定混合量正确或合适的一个常用方法是**滑落测试**（slump test）。把测试的混凝土放入一个圆锥体，然后移开圆锥体，移开后混凝土所滑落的距离就表明了原料的情况。为一般的厚板和梁设计的混凝土，其最小和最大的滑落距离分别为 4 和 8 英寸。图 5.18 的模糊集合显示了混凝土混合量产生式规则的模糊前件的一个可能定义。这些集合也可以用一张表或用 S-函数和 Π-函数来定义。

模糊规则后件中的模糊集合的定义如图 5.19 所示。在前面所说的前件下，这些能提供一个合适混合量的模糊后件之间会有重叠。注意，对于沙和石子的变化在 -20% 与 20% 之间的限制是任意定义的。和模糊集合前件一样，后件也可用 S-函数和 Π-函数来定义。

作为这些模糊产生式规则是如何工作的例子，假定混凝土的滑落距离是 6 英寸。从图 5.18 可看出，每条规则的隶属度或前件真值如下：

$\mu_{\text{TOO STIFF}}(6) = 0$
$\mu_{\text{WORKABLE}}(6) = 1$
$\mu_{\text{TOO WET}}(6) = 0$

因此只有一条规则 R_2 的前件被满足。这条规则激活后所产生的模糊后件是保持原状。

应用推理合成规则有：

$\mu_{\text{LEAVE ALONE}} = \max[\min(\mu_{\text{WORKABLE}}(6))] = \max[\min(1)] = 1$

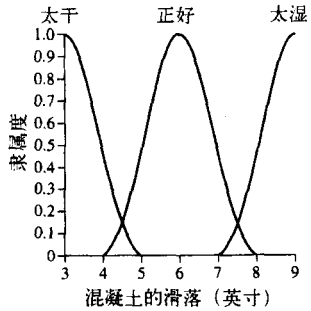


图 5.18 混凝土混合处理控制的模糊产生式规则前件

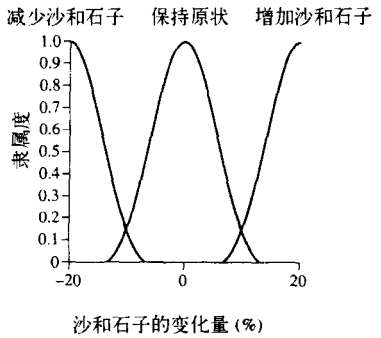


图 5.19 混凝土混合处理控制的模糊产生式规则后件

且从图 5.19 可看出，这可转化为沙和石子的变化量为 0%。

现在假设滑落距离为 4.8 英寸。从图 5.18 可得出：

$\mu_{\text{TOO STIFF}}(4.8) = 0.05$
$\mu_{\text{WORKABLE}}(4.8) = 0.2$
$\mu_{\text{TOO WET}}(4.8) = 0$

规则 R_1 和 R_2 的前件被部分满足，这与 PROSPECTOR 中概率规则前件的部分证据满足相类似。由于混合量太干或正好具有非零真值，因此规则 R_1 和 R_3 都会被激活并触发。注意，在一个模糊产生式系统中，除非具有一个前件真值的阈值，否则每一个具有非零前件真值的规则都会被触发。设置一个阈值是有必要的，它可阻止很多具有较低真值的规则被激活、触发而导致的无效操作。回想在 MYCIN 系统中，为了提高专家系统效率，一条规则被激活的最小确定性必须>0.2。同样也可在模糊集合的前件真值上定义一个类似的阈值。

对于滑落距离为 4.8 英寸，有两条规则被激活。应用最大-最小组合规则可得：

$$\mu_{\text{DECREASE SAND AND COARSE AGGREGATE}} = \max\ [\min(\mu_{\text{TOO STIFF}}(4.8))] = 0.05$$

$$\mu_{\text{LEAVE ALONE}} = \max\ [\min(\mu_{\text{WORKABLE}}(4.8))] = 0.2$$

可见，对单个前件项，最大-最小函数可不需要。

既然现在有两条规则有非零后件，则我们必须进行控制选择。最大值的方法将会简单地挑选具有最大隶属度的规则。在这个例子中，保留原状会被选择，因为它的隶属度 0.2 比另一规则的 0.05 要大。

瞬时方法从根本上算出规则模糊后件的**重心**（center of gravity）。术语重心来源于物理学，它表示这样一个点，如果物体的所有质量集中于这点，那么在外力的作用下，其效果是同样的。重心也叫第一瞬时惯性（first moment of inertia），其定义 I 是：

$$I = \frac{\int m(x)x\ d(x)}{\int m(x)\ d(x)}$$

其中积分符号表示普通的积分。

图 5.20 显示了两条规则 R_2 和 R_3 的模糊集合。注意模糊集合在其前件真值处被截断。这反映了推理组合规则，截断是因为，从直觉上，后件真值不可能超过前件真值。

后件的瞬时值计算如下：

$$I = \frac{\int \mu(x)\ x\ d(x)}{\int \mu(x)\ d(x)} \qquad \text{对连续元素}$$

或

$$I = \frac{\sum_i \mu_i x}{\sum_i \mu_i} \quad \text{对离散元素}$$

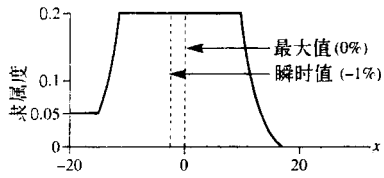


图 5.20 混凝土处理控制模糊规则的最大值和瞬时值方法

从图 5.20 可看出其值大约为 - 1%。虽然这与从最大值方法获得的 0% 非常接近，但其差异对于那些有重叠定义的模糊集合来说意义非常重要。最大值和瞬时方法都已用于对航空进行控制的模糊控制器中。在这个控制器中，最大值方法算出模糊后件的全部最大值的算术平均值。因此，即使存在多个元素有相同的最大值，仍可计算出一个明确的控制值。

除了最大值和瞬时方法外，其他方法也已用于解决**逆模糊化问题**（defuzzification problem），即把隶属度转化为一个明确控制值或一个描述控制变量的语言近似值。但是，仅通过一个简单数字或一个语言短语来描述一个模糊集合是很困难的。

可能性和概率

在模糊理论中，术语**可能性**（possibility）具有特别的含义。基本上，可能性指允许值。例如，假设有一个关于在其点数和论域$\mathscr{U}$上掷两个骰子的命题被定义如下：

p = X是一个在$\mathscr{U}$中的整数

$\mathscr{U}$ = {2,3,4,5,6,7,8,9,10,11,12}

在模糊术语学中，对任何整数 i，

Poss{x = i} = 1 (2≤i≤12)

Poss{x = i} = 0 否则

这里，Poss {x = i} 是“X 假定为 i 的可能性”的简写。骰子的值为 2～12 中任何一个值的可能性与值 i 的概率是不同的。换句话说，就是**可能性分布**（possibility distribution）与**概率分布**（probability distribution）是不一样的。骰子的概率分布是指**随机变量**（random variable）X，即两个骰子的和，被期望出现的频率。例如，7 由于有 1+6，2+5，3+4，因此 7 的出现概率为：

$$\frac{2\times3}{36} = \frac{1}{6}$$

而 2 的出现概率为：

$$\frac{1}{36}$$

与此相反，就公平骰子而言，对于全部从 2 到 12 的整数，可能性分布是个为 1 的常数。命题 p 称为引发一个可能性分布，Π_X。即给定一个模糊命题 p，它基于模糊集合 F 和语言变量 X，

p = X is F

当用这种方式表达时，该命题称为具有**典型形式**（canonical form）。术语典型意思是标准形式。模糊集合 F 是一个与普通逻辑的谓词不同的**模糊谓词**（fuzzy predicate），F 也可是一个模糊关系。由 p 引发的可能性分布等于 F 且可以用下面的**可能性分配方程**（possibility assignment equation）：

Π_x = F

来定义。这个方程的意思是，对于论域$\mathscr{U}$中的所有 x，都有：

Poss (X = x) = $\mu_F(x)$ $x \in \mathscr{U}$

例如，给定命题，

p = John is tall

可定义一个具有值 John 的语言变量 Height（高度），则典型形式：

```
X is F
```

可以用变量 Height 通过

```
Height(John) is TALL
```

的形式来表示，于是，

$$\text{Poss}\{\text{Height(John)} = x\} = \mu_{\text{TALL}}(x)$$

命题 P 可被写成如下的可能性分布：

$$\text{John is tall} \rightarrow \Pi_{\text{Height(John)}} = \text{TALL}$$

其中箭头符号表示“转换为”，Height 是一个语言变量，TALL 是一个模糊集合。注意 John 不是一个语言变量。

虽然一个模糊集合可以被指定为一个可能性分布，如 $\Pi_X = F$，但这两者实际上是不一样的。作为说明它们区别的一个例子，思考下面定义的模糊集合：

```
ROLL(1) = 1/3 + 1/4
```

这个集合的意思是，在骰子的一次具体投掷 Roll 1 中，有一个骰子为 3，另一个为 4。与之相反，可能性分布：

$$\Pi_{\text{ROLL(1)}} = 1/3 + 1/4$$

是指投掷结果为 3 或 4。这里的“或”是异或，表示我们对投掷的不确定性。有可能为 3，也有可能为 4。在模糊集合中，可以确定骰子的值是 3 和 4。而可能性分布是指不管是 3 还是 4 出现，骰子都是公平的。模糊集合告诉了我们一次投掷后所显示的值。

作为另外一个例子，思考命题“Hans 以 X 个鸡蛋作为早餐”，这里 X 是指论域 X = {1，2，…8} 中的任何一个值。

表 5.18 Hans 吃 X 个鸡蛋的难易程度和概率分布

X	1	2	3	4	5	6	7	8
$\Pi_{\text{ATE(Hans)}}(X)$	1.0	1.0	1.0	1.0	0.8	0.6	0.4	0.2
P(X)	0.1	0.8	0.1	0.0	0.0	0.0	0.0	0.0

可能性分布 $\Pi_{\text{ATE(Hans)}}(X)$ 解释了 Hans 吃 X 个鸡蛋的难易程度。概率分布 P（X）是通过对 Hans 吃早餐进行一年的调查而总结出来的。

有一点很重要，即可能性是非统计的，而概率是统计的。例如，虽然 Hans 能吃 8 个蛋，但调查结果表明，在观察的这段时期内他从未吃过多于 3 个鸡蛋的早餐。从这个意义上说，可能性是指能力或容量。可能性高并不意味着概率高，即是在可能性和概率之间没有关联。

对普通概率的模糊扩展就是**模糊概率**（fuzzy probability）。它描述了一种不能精确知道的概率。模糊概率的一些例子是**模糊限定词**（fuzzy qualifier），如非常可能、未必可能、不是非常可能等等。含有模糊概率的一个模糊命题例子是：

电池是坏的是非常可能的

转换规则

模糊概率纳入模糊逻辑，称为基于 Lukasiewicz 的 L_1 逻辑的 FL。FL 的一个主要部分是一组**转换规则**（translation rule），这些规则说明了如何从基本命题得到修饰或合成规则。

转换规则可以分为四类：

- 类 I：**修饰规则**（modification rule），如

  X 非常大

John 比 Mike 高得多

- 类Ⅱ：**合成规则**（composition rule），如

  条件合成（conditional composition）

  如果 X 是高的，则 Y 是矮的

  合取合成（conjunctive composition）

  X 是高的且 Y 是矮的

  析取合成（disjunctive composition）

  X 是高的或 Y 是矮的

  条件合取合成（conditional and conjunctive composition）

  如果 X 是高的，则 Y 是矮的，否则 Y 是更矮的

- 类Ⅲ：**量化规则**（quantification rule），如

  大多数甜品是很好的

  太多营养食物是致胖的

- 类Ⅳ：**限定规则**（qualification rule），如

  真值限定（truth qualification）

  巧克力馅饼是美味的、是非常真实的

  概率限定（probability qualification）

  巧克力馅饼服务迅速是非常可能的

  可能性限定（possibility qualification）

  对你而言巧克力馅饼是坏的是不可能的

限定规则是那些与模糊概率有关的规则。量化规则用来处理像 Most（大多数）那样的模糊量词，这些量词不能用经典的论域和存在量词来定义。

类Ⅰ规则的转换可表示为：

$$X \text{ is } F \rightarrow \Pi_X = F$$

和转换命题

$$X \text{ is } mF \rightarrow \Pi_X = F^+$$

这里，m 是一个修饰词，如 Not、Very、More Or Less 等等。F^+ 表示 F 被 m 修饰。m 和 F^+ 的一些默认定义如表 5.19 所示，它们和以前所讨论的语言限定词一样。也可采用其他的对 m 和 F^+ 的定义。

作为一个例子，定义 TALL，

$$TALL = 0.2/5 + 0.6/6 + 1/7$$

则转换如下：

$$\text{John is not tall} \rightarrow 0.8/5 + 0.4/6 + 0/7$$
$$\text{John is very tall} \rightarrow 0.04/5 + 0.36/6 + 1/7$$
$$\text{John is more or less tall} \rightarrow 0.45/5 + 0.77/6 + 1/7$$

变量 X 不一定是一元变量。一般，X 可以是一个二元或 N 元关系。例如，命题“Y and Z areF”包括在“X is F”中。定义 CLOSE 为一个 $\mathscr{U} \times \mathscr{U}$ 上的模糊关系，

$$CLOSE = 1/(1,1) + 0.5/(1,2) + 0.5/(2,1)$$

于是，

表 5.19 某些类Ⅰ规则的转换参数值

m	F^+
Not	$F' = \int [1 - \mu_F(x)]/x$
Very	$F^2 = \int \mu_F{}^2(x)/x$
More Or Less	$\sqrt{F} = \int \sqrt{\mu_F(x)}/x$

注：所有积分都是对整个论域。

X and Y are close → $\Pi_{(X,Y)}$ = CLOSE

X and Y are very close → $\Pi_{(X,Y)}$
= $CLOSE^2$
= 1/(1,1) +0.25/(1,2) +0.25/(2,1)

一个类Ⅱ规则的例子如下：

IF X is F then Y is G → $\Pi_{(X,Y)} = \overline{F}' \oplus \overline{G}$

这里，$\overline{F}$ 和 $\overline{G}$ 是F和G在论域上的圆柱扩展。

$\overline{F}$ = F × V $\overline{G}$ = U × G

⊕是有界和。对于这个规则，

$\mu_{\overline{F}' \oplus \overline{G}}(x,y) = 1 \wedge [1 - \mu_F(x) + \mu_G(y)]$

∧是最小函数。这个定义与 L_1 逻辑中的蕴含是一致的，但其他的定义可能会不一致。作为一个类Ⅱ规则的例子，定义

U = V = {4, 5, 6, 7}

F = TALL =0.2/5 +0.6/6 + 1/7

G = SHORT = 1/4 +0.2/5

IF X is tall then Y is short → $\Pi_{(X,Y)}$

= 1/(5,4) + 1/(5,5) + 1/(6,4) +
0.6/(6,5) + 1/(7,4) +0.2(7,5)

这里，元素如（5，4）的隶属度是如下计算的：

$\mu_{\overline{F}' \oplus \overline{G}}(5,4) = 1 \wedge [1 - 0.2 + 1] = 1 \wedge [1.8] = 1$

模糊专家系统中的不确定性

当模糊概率用于专家系统中时，将会与普通的概率推理有所不同。思考下面这个典型的模糊规则：

IF X is F then Y is G（概率是 ß）

这可以写成一个条件概率的形式：

P(Y is G | X is F) = ß

一个使用经典概率论的传统专家系统会假设：

P(Y is not G | X is F) = 1 − ß

然而，在模糊理论中，如果F是一个模糊集，那么这个式子就不再成立了。正确的模糊结果将会弱些：

P(Y is not G | X is F) + P(Y is G | X is F) ≥ 1

因为这仅设置了概率的一个下限，这个下限可能是模糊的。一般说来，对于模糊系统：

P(H|E)不一定等于 1 − P(H′ | E)

在模糊专家系统中，可能存在3个方面的模糊：

(1) 规则的前件和/或后件，如：

If X is F then Y is G
If X is F then Y is G with CF = α

这里，CF是确定性因子，α是一个如0.5之类的数值。

(2) 前件与匹配前件模式的事实之间只是部分匹配。在非模糊专家系统中，规则不会被激活除非模式与事实精确匹配。但是，在模糊专家系统中，任何事情都只是一个程度的问题。除非设置了一个

阈值，否则所有规则在某种程度上都可以被激活。

(3) 模糊量词如 most 和模糊限定词如 Very Likely（非常可能）、Quite True（十分正确）、Definitely Possible（完全可能）等等。

命题中通常含有明确和/或不明确的模糊量词。作为一个例子，思考下面**倾向**（disposition）

d = desserts are Wonderful(甜饼是美味的)

术语“倾向”表示一个通常正确的命题，它的典型形式是：

Usually(X is R)

这里，Usually 是一个隐含的模糊量词，R 是一个**约束关系**（constraining relation），它规定了约束变量 X 取值的范围。许多人们知道的启发式规则都是“倾向”。实际上，**常识知识**（commonsense knowledge）本质上就是对现实世界中“倾向”的总括。

“倾向”可以转换成明确的命题形式：

```
p = Usually desserts are wonderful
p = Most desserts are wonderful
```

它可以表示成下面这个启发式规则：

```
r = If x is a dessert
    then it is likely that x is wonderful
```

下面是一些模糊系统中的推理规则：

- 传递原理（entailment principle）

$$\begin{array}{l} X \text{ is } F \\ \underline{F \subset G} \\ X \text{ is } G \end{array}$$

- 倾向传递（dispositional entailment）：它是把 Always（总是）变成 Usually（常常）的一种限定情形

$$\begin{array}{l} \text{Usually } (X \text{ is } F) \\ \underline{F \subset G} \\ \text{Usually } (X \text{ is } G) \end{array}$$

- 合成规则（compositional rule）

$$\begin{array}{l} X \text{ is } F \\ \underline{(X,Y) \text{ is } R} \\ Y \text{ is } F \circ R \end{array}$$

这里，R 是作用在二元变量（X，Y）上的一个二元关系，且

$$\mu_{F \circ R}(y) = \sup_x [\mu_F(x) \wedge \mu_R(x,y)]$$

Sup 是 supremum 的缩写，意思是上**确界**（least upper bound）。通常，上确界与最大值函数是一样的。不同之处在于，当没有最大值时，如小于 0 的实数区间，此时没有一个小于 0 的最大实数，但上确界却取值为 0。

- 广义假言推理（generalized modus ponens）

$$\begin{array}{l} X \text{ is } F \\ \underline{Y \text{ is } G \text{ if } X \text{ is } H} \\ Y \text{ is } F \circ (H' \oplus G) \end{array}$$

这里 H′是 H 的模糊补集，有界和的定义是：

$$\mu_{H' \oplus G}(x,y) = 1 \wedge [1 - \mu_H(x) + \mu_G(y)]$$

广义假言推理不需要前件“X is H”与前提“X is F”是一样的。注意，这与经典逻辑完全不同，经典逻辑需要它们精确匹配。广义假言推理实际上是推理合成规则的一个特例。传统的专家系统把假言推理作为基本规则，而模糊专家系统则把推理合成规则作为基本规则。

专家系统使用近似推理采用了两种不同的方法，一种是真值约束方法，另一种是合成推理方法。在 Whalen 所总结的 11 个模糊专家系统中，几乎都是使用合成推理方法。由于很难提前预测哪一种方法最有效，通常使用经验模型来考察哪一个最适合当前数据。这并不像听起来那么糟，比如在统计理论中，首先使用线性回归，因为它是最简单的，此外人们也假设人口呈高斯分布。如果不起作用，则使用更复杂的模型直到其中的误差是可接受的。

5.6 不确定性的现状

如果你已经读过以上 5 章的每一个单词并推导了每一条公式，求解了每一个习题，那么你要么是对知识有强烈的渴求，要么是有失眠症。不管哪种情况，阅读这本书的后续部分都将解决你的困惑。最大的问题在于，走过了所有不同的树木和森林，我们能看到任何山脉么？哪里才是正确建造专家系统的根本所在？

在树木和森林的远处有两座山，其中一座很高很清晰，这是“逻辑之山”，专家系统必须建立在其上。一个专家系统必须像人类一样给定有效的前提可得到有效的结论。给定（1）规则书写正确，（2）推理机推出结论的事实为真，专家系统必须能够得到有效的结论。注意我们没有要求事实在现实世界中是合理的。如果专家的推理不合理或者事实无效，我们并不期望推理机可以得到合理的结论。设计专家系统仅是用来模拟某个有限知识领域的人类专家，人们并不总是理性的推理。不过至少在给出有效事实时专家系统必须得到有效结论。

第二座看起来很高的山是“不确定性之山”。奇特的是无论你走得多近，它的影像始终不清晰。事实上当你研究一个特定理论时，每当出现一个新变化，在不确定性的分形之山上也就多了一个新变化。我们能做的最恰当处理是用专家知识模型化它，或者尝试用不确定性的多种方法，让不同的技术与之一决雌雄。这种渐进的方法是基于经典的**黑板架构**（blackboard architecture），这个架构中不同的代理同时从不同角度考察一个问题。所有的代理把它们的疑问碎片放在一块黑板上进行考察，希望不同的碎片可以构成一个整体。一个监督程序将考察所有的碎片然后试图拼出完整的图案。在不确定性领域，如果有一个确定的理论，那就不存在不确定性。

今天模糊逻辑和贝叶斯定理最常用来处理不确定性。然而也有很多其他理论应用于这个领域。选择一个能提供广泛选择范围的软件是很重要的。通常，商业软件提供最多的选择但是最贵。在许多情况下，可以得到学术研究用版本或者试用版本。

随着功能强大的台式电脑的普遍应用以及提供成千上万计算机网格计算的链接的存在，使你可以在短时间内在大型数据集上试用不同的模型。这种大型混合计算的问题在于，你必须很仔细，因为可以使用很多参数以使它 100% 符合你的测试数据。一个 100% 的符合和 0% 的符合一样糟糕（也可能是很好，在于你如何看它），因为这意味着其中有些错误。从统计学我们知道由于数据噪音以及有限的样本大小，所以应该存在着随机变动。

例如，给出 N 个数据点，从头到尾我们总是可以画出 N－1 条线。这样一个模型对任何两点间的增添都适用，但是对数据进行外推就不适合了。我们在专家系统中追求的是像人类一样能从已知预测和外推新情况的能力。

经典贝叶斯理论、**贝叶斯凸集理论**（convex set Bayesian）、Dempster-Shafer 、Kyburg 以及模糊逻辑中的可能性理论是目前为数不多的其软件工具广泛可得的几种方法。试用不同的模型可能比分析更耗费你的时间。在贝叶斯理论的凸集方法中，信任不是经典贝叶斯方法中的一个函数，而是通过一组**凸函数**（convex function）集来刻划，这意味着任何函数都能由另外两个函数的线性组合来表示。

自从 Dempster-Shafer 理论最早作为经典概率理论的扩展而引入，就受到了很多的批驳。例如，Kyburg 在论文中宣称 Dempster-Shafer 理论不是经典概率论的扩展，实际上只是一种别的方法。Kyburg 认为 Dempster-Shafer 理论包含在经典概率论中，Dempster-Shafer 区间也包含在贝叶斯理论的凸集方法中。Dempster-Shafer 理论在处理趋近于 0 的信任时似乎也遇到了困难，信任为 0 和信任很小时所得的

结果有很大的不同。另一个问题是，随着诊断问题可能答案的增加，Dempster-Shafer 理论的计算呈指数增长。

虽然 Gordon 与 Shortliffe 的近似值方法避免了指数增长，但它在处理有明显冲突的证据时，得出的结果不理想。另一种不是近似值的方法在处理对立证据时得出了较好的结果。许多别的论文趋向于使用对 Dempster-Shafer 理论的不同扩展来克服指数增长问题。

所有这些工作的主要意义在于再次检查了概率论的基础是否正确，并且增加了对处理不确性的方法的兴趣。此外，一系列综合了模糊逻辑和 ANS 的混合方法也被开发出来并且取得了成功（Mendel 00）。一个重要的事情就是选择不确定性模型类似于在传统程序语言中选择数据结构，例如，数组、链表、树、队列、堆栈等等。请选择最适合问题的模型。

5.7 模糊逻辑的一些商业应用

从照相机到洗衣机，模糊逻辑的商业应用无处不在。模糊逻辑资源的链接列在附录 G 中。

- **水电站**的水闸门自动控制
  （Tokyo Electric Power）
- **机器人**简化控制
  （Hirota，Fuji Electric，Toshiba ，Omron）
- 体育转播的**摄影瞄准**
  （Omron）
- **股票交易评估**的专家代理
  （Yamaichi，Hitachi）
- **空调系统**中的温度跳动预防
  （Mitsubishi，Sharp）
- **汽车引擎**的稳定有效控制
  （Nissan）
- **机动车**巡航控制
  （Nissan，Subaru）
- **工业控制应用**系统的效率提高和功能优化
  （Aptronix，Omron，Meiden，Sha，Micom，Mitsubishi，Nisshin-Denki，Oku-Electronics）
- **半导体生产**中的晶片分档器定位
  （Canon）
- **公交车运行时间表**的优化
  （Toshiba，Nippon-System，Keihan-Express）
- **文档**管理系统
  （Mitsubishi Electric.）
- 地震早期预报的**预测系统**
  （Inst. Of Seismology Bureau of Metrology，Japan）
- **医疗技术**：癌症诊断
  （Kawasaki Medical School）
- 模糊逻辑和**神经网络**结合
  （Matsushita）
- **掌上电脑**中的手写体识别
  （Sony）
- 摄像机的动画补偿

（Canon，Minolta）

- **真空吸尘器**地表情况和脏污程度识别的自动马达控制
  （Matsushita）
- **摄像机**的背光控制
  （Sanyo）
- **摄像机**防震补偿
  （Matsushita）
- **洗衣机**单键控制
  （Matsushita，Hitatchi）
- 手写体、物体、声音**识别**
  （CSK，Hitachi，Hosai Univ.，Ricoh）
- **直升机**飞行辅助
  （Sugeno）
- **法律诉讼程序**模拟
  （Meihi Gakuin Univ，Nagoy Univ.）
- 工业过程的**软件设计**
  （Aptronix，Harima，Ishikawajima-OC Engeneering）
- **钢铁制造**的机器速度和温度控制
  （Kawasaki Steel，New-Nippon Steel，NKK）
- 提高驾驶舒适度、停定精度和节省能源的**地铁系统控制**
  （Hitachi）
- 节省**机动车**燃料消耗
  （NOK，Nippon Denki Tools）
- **电梯**控制中的灵敏度和效率提高
  （Fujitec，Hitachi，Toshiba）
- 提高**核反应**的安全性
  （Hitachi，Bernard，Nuclear Fuel Div.）

5.8 小结

本章讨论了不确定性的非概率理论。确定性因子、Dempster-Shafer 理论和模糊理论都是处理专家系统中不确定性的方法。确定性因子易于实现，并且已成功地用于如 MYCIN 这样推理链较短的专家系统中，但确定性因子只是一种“特别”的理论，它一般不适于长推理链的情况。

Dempster-Shafer 理论基础严密且专门针对专家系统。

模糊理论是已系统化的关于不确定性的最通用理论。由于扩张原理而得到广泛应用。自从 Zadeh 1965 年发表了第一篇经典论文之后，模糊理论已被用于许多领域。附录 G 列出了许多链接。

习题

5.1 在初始证据 E_1 之上给出证据 E_2，证明：

$$P(D_i \mid E_1 \cap E_2) = \frac{P(E_2 \mid D_i \cap E_1)P(D_i \mid E_1)}{\sum_j P(E_2 \mid D_j \cap E_1)P(D_j \mid E_1)}$$

5.2 证明：

$$CF(H,E) + CF(H',E) = 0$$

5.3 给定规则：

IF E_1 AND E_2 AND E_3
THEN H (CF_1)

IF E_4 OR E_5
THEN H (CF_2)

这里，

$CF_1(E_1,e)=1$　　$CF_1(E_2,e)=0.5$　　$CF_1(E_3,e)=0.3$

$CF_2(E_4,e)=0.7$　　$CF_2(E_5,e)=0.2$

$CF_1(H,E)=0.5$　　$CF_2(H,E)=0.9$

(a) 画一棵树说明这些规则如何推出 H。

(b) 计算确定性因子 $CF_1(H,e)$及 $CF_2(H,e)$。

(c) 计算 $CFCOMBINE[CF_1(H,e),CF_2(H,e)]$。

5.4 (a) 在图 5.7 (c) 中，设：

m(X) = 0.2
m(Y) = 0.3
m(Z) = 0.5

运用 Dempster-Shafer 理论找出 X、Y 及 Z 的证据区间。

(b) 在图 5.7 (d) 中，设：

m(X) = 0.4
m(Y) = 0.6

找出下面式子的证据区间：

X
$X \cap Y$
Y
$X \cap Y'$
$X' \cap Y$

5.5 给定规则：

规则 1：IF E THEN H
规则 2：IF E THEN H′

并设：

$\Theta=\{H,H'\}$
对规则 1，$m_1(H)=0.5$　$m_1(\Theta)=0.5$
对规则 2，$m_2(H')=0.3$　$m_2(\Theta)=0.7$

(a) 写出 Dempster-Shafer 表显示证据组合并计算组合信任函数。

(b) 计算合情度

(c) 计算证据区间

(d) 计算怀疑度

(e) 计算未知度

5.6 基于不同类型传感器的报告，下表给出了飞机环境中客机 (H)、轰炸机 (B)、战斗机 (F) 的信任度。

(a) 计算初始信任函数、合情度、证据区间、怀疑度、未知度。

(b) 经过证据组合后，计算上述同样参数。

焦元	传感器 1 (m_1)	传感器 2 (m_2)
Q	0.15	0.2
A, B	0.3	0.1
A, F	0.1	0.05
B, F	0.1	0.1
A	0.05	0.3
B	0.2	0.05
F	0.1	0.2

5.7 一个警察拦截了一个超速驾驶者。基于警察雷达探测器和驾驶者速度计的误差，信任函数如下：

(a) 计算每个人的信任函数、合情度、证据区间、怀疑度、未知度。

(b) 经过证据组合后，再计算上述参数。

(c) 基于这些参数，解释你为什么相信驾驶者超速或没有超速。

警 察	驾 驶 者
$m_1(57)=0.3$	$m_2(56)=0.2$
$m_1(56)=0.5$	$m_2(55)=0.6$
$m_1(55)=0.2$	$m_2(54)=0.2$

5.8 给定模糊集，

$A = 0.1/1 + 0.2/2 + 0.3/3 \quad B = 0.2/1 + 0.3/2 + 0.4/3$

计算/解释下面各题：

(a) 这两个集合相等吗？解释之。

(b) 集合补

(c) 集合并

(d) 集合交

(e) 每个集合满足排中率吗？解释之。

(f) 集合积

(g) 每个集合的二次幂

(h) 概率和

(i) 有界和

(j) 有界积

(k) 有界差

(l) 集中

(m) 扩张

(n) 强化

(o) 标准化

5.9 给定模糊集合

$$Q = \begin{bmatrix} 0.2 & 0.3 \\ 0.4 & 1.0 \end{bmatrix} \quad \text{定义在 } U_1 \times U_2 \text{ 上}$$

$$P = \begin{bmatrix} 0.1 & 0.5 & 0.3 \\ 0.2 & 0.0 & 0.4 \end{bmatrix} \quad \text{定义在 } U_2 \times U_3 \text{ 上}$$

(a) 计算每个集合的第一、第二及全投影。

(b) 计算每个集合的圆柱扩展。

(c) 证明：

$$Q \circ P = \text{proj}(\bar{Q} \cap \bar{P};\ U_1 \times U_3)$$

5.10 考虑语言变量“人”作为一个三阶模糊集合。

(a) 定义人为3个二阶模糊集合。

(b) 用3个一阶模糊集合来定义每个二阶模糊集合。

(c) 用S-函数和/或Π-函数定义这3个一阶模糊集合。

5.11 (a) 为语言变量“不确定性”定义5个语言值。

(b) 为这些值画出适当的函数，并解释之。

(c) 画出下面的模糊集合

Not TRUE(不真)

More Or Less TRUE (几乎真)

Sort Of TRUE(有点真)

Pretty TRUE (很真)

Rather TRUE (相当真)

TRUE (真)

设 TRUE 是一个 S-函数。TRUE 的极限是什么？解释之。

5.12 (a) 为语言变量“水温”定义至少 6 个值。

(b) 在一个图上画出每个模糊集合值的适当函数。

(c) 基于冰点 (FREEZING) 给出 3 个受限的模糊集合函数。

5.13 (a) 通过 N=2 和 N=3 的真值表来证明表 5.14 中的基本逻辑运算符。

(b) 从 x，y 的绝对值角度推导 x↔y。

5.14 给定数值真值，

```
x(A) = 0.2/0.1 + 0.6/0.5 + 1/0.9
x(B) = 0.1/0.1 + 0.3/0.5 + 1/0.9
```

计算下面的模糊逻辑真值：

(a) NOT A

(b) A AND B

(c) A OR B

(d) A→B

(e) B→A

5.15 使用限定基本项和范围短语定义模糊语法以生成下面规则：

```
IF PRESSURE IS HIGH (压力高)
THEN TURN VALVE LOWER (调低阀门)

IF PRESSURE IS VERY HIGH
THEN TURN VALVE MUCH LOWER

IF PRESSURE IS VERY VERY HIGH
THEN TURN VALVE MUCH MUCH LOWER

IF PRESSURE IS LOW TO MEDIUM (压力是低到中等)
THEN TURN VALVE HIGHER
```

设 (1) PRESSURE 的基本项只有 LOW 和 HIGH，(2) 范围短语 TO 只出现在规则前件中，(3) 结论中 VALVE 的基本项只有 LOWER 和 HIGHER。

参考文献

Note that many software resources and online material on fuzzy logic is shown in Appendix G Software Resources.

(Bertino 01). Elisa Bertino *et. al., Intelligent Database Systems,* ACM Press, 2001.

(Chen 01). Guanrong Chen and Trung Tat Pham, Introduction to Fuzzy Sets, Fuzzy Logic, and Fuzzy Control Systems, CRC Press, 2001.

(Constantin 95). Von Altrock Constantin, *Fuzzy Logic and Neuro Fuzzy Applications Explained,* Prentice- Hall, 1995.

(Cornelius 98). *Fuzzy Logic and Expert Systems Applications* ed. by Cornelius T. Leondes Academic Press, 1998.

(Kerre 00). Da Ruan and Etienne E. Kerre, *Fuzzy If-Then Rules in Computational Intelligence*, Kluwer Academic Publishers, 2000.

(Da Ruan 97). Intelligent Hybrid Systems: Fuzzy Logic, Neural Networks, and Genetic Algorithms, Kluwer Academic Publishers, 1997.

(Gardenfors 04). Peter Gardenfors, *Belief Revision,* Cambridge Tracts in Theoretical Computer Science, 2004. (Dempster 67). A. P. Dempster, "Upper and Lower Probabilities Induced by Multivalued Mappings," *Annals of Math. Stat.*, 38, pp. 325-329, 1967.

(Giarratano 91). Joseph Giarratano, *et al.*, "Fuzzy Logic Control for Camera Tracking System," Fifth Annual Workshop on Space Operations Applications and Research (SOAR '91), pp. 94-99, 1991.

(Harris 00). John Harris, *An Introduction to Fuzzy Logic Applications*, Kluwer Academic Publishers, 2000.

(Ibrahim 03). Ahmed M. Ibrahim, *Fuzzy Logic for Embedded Systems Applications*, book with CDROM with lots of references, pub. by Newnes, 2003.

(Kosko 96). Bart Kosko, *Fuzzy Engineering,* Prentice Hall, 1996 .

(Liu 04). Puyin Liu, *Fuzzy Neural Network Theory and Application (Machine Perception and Artificial Intelligence*, World Scientific Publishing Company, 2004.

(Mendel 00). Uncertain Rule-Based Fuzzy Logic Systems: Introduction and New Directions, Prentice-Hall, 2000.

(Nguyen 99). Hung T. Nguyen and Elbert A. Walker *A First Course in Fuzzy Logic*, CRC Press, 1999.

(Roger 96). Jyh-Shing Roger Jang, Chuen-Tsai Sun, Eiji Mizutani, *Neuro-Fuzzy and Soft Computing: A Computational Approach to Learning and Machine Intelligence*, Prentice-Hall, 1996.

(Ross 04). Timothy J. Ross, *Fuzzy Logic with Engineering Applications*, John Wiley & Sons, 2004.

(Siller 04). William Siler and James J. Buckley , *Fuzzy Expert Systems: Theory and Applications*, Wiley-Interscience 2004.

第6章　专家系统设计

6.1　概述

在前面几章，我们已讨论了专家系统和其他一些智能决策系统的一般概念和理论。许多论文尝试总结智能系统的特性。每种类型都有很多优缺点（Begley 03）。本章就实际专家系统的设计提供一些一般的指导，这些设计是针对现实世界应用，而不是研究模型的。本章将描述**软件工程方法学**（software engineering methodology）以便在最短的时间内开发出高质、高效、低成本的专家系统。

许多书籍、论文和软件工具都致力于专家系统设计，涵盖了各个方面的细节问题。所以这一章不可能让你马上成为一个专家。但是通过理解专家系统设计的方法原理，你可以更好地利用那些复杂的工具和方法。事实上，专家系统设计是一个更广泛的领域，称为**知识管理**（Knowledge Management，KM）的一部分，知识管理研究知识的组织（Becerra-Fernandez 03）。

KM和信息管理（Information Management IM）有关，信息管理又与信息处理（Information Processing，IP）有关，信息处理又与信息系统（Information System，IS）有关，而信息系统与信息技术（Information Technology，IT）又有关。KM主要为不同类型用户考虑所有不同类型的资源，如办公室、网页、常见问题、桌面帮助（人工和自动）、电子邮件、传真、电话、产品、程序、开发者、管理者、终端用户和广告等。

大公司具有自己详细的方法、文档、书籍和软件，并由他们自行开发、管理、维护和售卖，因为这是一个盈利领域（Conway 02）。特别是智能工具被开发后可以极大地减少人员开支，甚至比全球外包更便宜。

如果你想想一个普通人花在使用计算机上的时间，则我们需要用计算机来管理由计算机创建的文档。专家系统之所以能广泛用于商业上，就是因为它对不断迅猛增长的网上信息和知识来说是必不可少的。

6.2　选择合适的问题

有很多方法和资源提供给你来建立一个基于知识的系统或专家系统，也有更多的方法会让你浪费时间和金钱。不过，**智慧第四法则**（Fourth Law of Wisdom）告诉我们在拟定一个行程之前应该明确目的地。

在建立专家系统前，必须选择合适的问题，就像第1.6小节里讨论的那样。做专家系统和做任何其他软件项目一样，在决定其所需的人力、资源、时间时，都要作一个全盘考虑。这个全盘考虑就是传统程序设计的项目管理，这一点在专家系统设计里必须形成制度。图6.1显示了专家系统开发的一个高层管理模式，其三个一般阶段的具体细节在它们的下面列出，这些具体细节必须采取问题-答案的方式进行讨论以作为专家系统设计的指导方针。

选择合适的范例

为什么要建立专家系统

对任何一个项目来说，这可能是最重要的问题。最理想的答案是公司领导的决定。假如不是这种情形，第1章第2节中所描述的专家系统的一般优点就无从谈起。最重要的是，记住只有管理人员才有权决定技术人员想要建造的系统。如果你决定用你的业余时间建造专家系统，以向公司证明你的正确，随之，你意识到这是个很成功的产品，然后你决定辞职并自创公司。那么，这就牵涉到你所签的**知识产权协议**（Intellectual Property Agreement）。绝大多数IPA说明，无论你做什么，不管与你的工作

是否有关，你一年 24×7×52 小时都属于公司。特别是，这个问题必须由物主或股东来决定是否提供资金开发。在开始项目前，对问题、专家、用户等都必须有一个清楚的认识。

注意，我们这里使用的是“专家”，而不是“专家们”。正如拥有多个伴侣会有很多麻烦，多个专家也会产生矛盾。即使是总统看病，一次也只看一个医生，这并不是花费的问题。试图在一个系统中模型化多个专家的知识是一个糟糕的主意。但是，使用黑板架构把多个系统中的多个专家的知识统一到一个主要观点并模型化却是一个好想法。这里使用了模糊术语“好”。如果 10 个医生中有 9 个认为病人会死，病人只会选择那个能提供生的机会的医生。如何排列多个专家的意见是一件很困难的事，甚至比为单一专家建立单一系统更难。所以我们在本章坚持单一系统单一专家的方法，而把多专家多系统作为一个课程项目。

收益

收益是什么

这个问题和第一个问题有联系，然而却更实际，它要求在人力、资源、时间和金钱上的投资都必须有特别的回报。收益可体现在金钱、效率和第 1 章中所描述的专家系统优点上。此外，很重要的一点是必须记住，如果没有用户，就没有收益。由于专家系统是一门新的技术，较之传统程序设计来说，回答这个问题更困难，也更具风险。

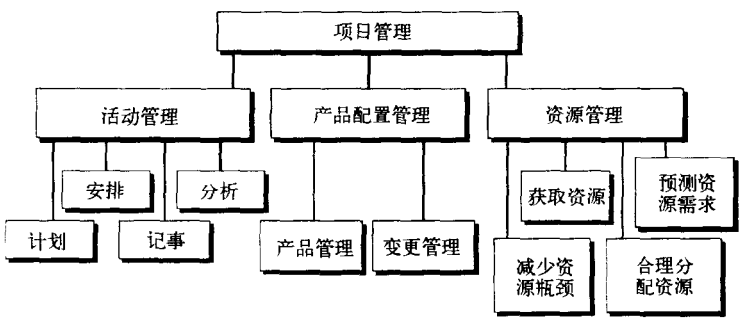


图 6.1 项目管理任务

工具

建立系统时哪些工具是可用的

目前有许多不同优缺点的专家系统工具可供选择，附录 G 总结了这些工具的一些特性。然而，由于软件的迅速发展，这些仅仅只能提供一个参考。一般地，每两年每种工具都会有许多功能增强，而每五年则会有一个大的版本更新。当然，修正旧版本的漏洞比以新的名字新的版本号发行会花费更多的金钱。如果你对这一点有疑问，可以数数过去十年内你使用的同一个操作系统一共有多少个新的名字。

选择专家系统工具应该遵循**智慧第三法则**（Third Law of Wisdom）：如果你迷惑了，问别人。查看已使用这软件建立的关于其应用的网页。试找出不仅仅是成功的，还有失败的应用。尽管很难得到严格的统计数字，但是多数估计认为成功的软件只有 20%～30%。如果你需要和一个忙碌的专家合作并且不了解其专业术语的时候，这个数字还将下降。

这也就引发了**智慧第二法则**（Second Law of Wisdom）：如果你要寻求某人的帮助，最好先懂得他的语言。尽管有很多人类和计算机翻译，但学习一个知识领域中的术语还是很困难。

然而，仅仅学习术语或记住单词的字典定义并不能建立语义关系网，这是基于规则的专家系统的基础。尽管每条规则都表达了一个知识块，在知识库中规则之间还同时存在强的和弱的连接。如果某

条规则的激发必然导致另一条规则也被激发，那么它们存在**强耦合**（strong-coupling），或者说，是推理链中一个从有效事实到有效结论的强连接。如果一个规则的激发导致多条规则可以被推理机激发，那么是**弱耦合**（weak-coupling）。这意味着推理链并不强；由于多条路径存在，我们不知道应该走哪个方向。

可潜在激发的规则越多，意味着推理链还达不到强耦合。但这并不是坏事。如果每条规则都与另一规则强耦合，就不需要推理机和专家系统了，因为过程化程序设计语言能更好地处理。过程化语言的执行是连续的，一个语句接一个语句，除非有条件控制语句如 IF，否则按照输入顺序执行。如果有一个算法能有效地解决问题而不需要人工智能方案。理想地，专家系统是强弱耦合规则的均衡混合，就如同人类会通过演绎、归纳、概率和其他方法来得到结论。

考虑下面某个知识领域词语的**语义表**（semantic list）。称之为语义表是因为词与词之间没有图连接，而是一个词自然地连接着上一个词（注意，你年纪越大，就越能理解）。看你需要多少时间理解：过期电影、过期通知、过期罚款、拖欠账户、冻结账户，收债代理、电话通知、电子邮件巨量增加、信用败坏、低信用、无信用、“我们可以为你提供贷款”、欠账、应付税收、附加税利息、代理费用、抵押、州长拍卖、说客、竞选捐献、挪用公款、州长道歉、公司道歉、巨额赔偿、买下录像出租公司。

语义表通常和其他符号相关联。符号学研究符号及其含义。大家都熟悉的一个例子是小时学“A、B、C”这首歌，因为其中旋律和歌词具有符号关联，所以当小孩们唱“A、B、C”时，我们仍能记起曲调。在著名的 TV 叮当节目广告中广泛地应用符号关联。某些叮当制作人善于制造出能让你的脑海不停地回旋这些旋律的叮当乐曲。

当你拜访一个专家的时候，同时记录内容是有必要的。通过阅读这些记录，你可以发现这些词语之间的语义关系，这才是重要的。知识术语并不是孤立的单词；它们构成了语义簇，继而又连接形成专家脑海中的整个知识语义网。大脑是相关联符号记忆的一个很好例子，其中一个词，味觉、触觉、视觉或听觉可以激发所有其他记忆。

面谈是导出知识的传统方法，这方面已做了很多工作（Novak 98）。事实上，面谈的一个技巧就是通过画出图表来确定概念和关联。可视化的形象表示有利于专家和开发者同时看到关联和意义，并就所表达的内容达成一致。已有表达整个本体的软件工具。回忆第 1 章中，本体是某个领域的完整形式描述。

有一个非常成功的工具 Protégé，这是一个本体和知识库编辑工具。它是斯坦福大学免费提供的复杂工具，在其网站上有关于创建第一个本体以及其他许多主题的良好技术支持。Protégé 和 TixClips，作用一个 CLIPS 的集成开发环境一起使用。当建立有成千条规则的大系统时，本体是必不可少的。

成本

要花费多少成本

这就如同询问离婚的成本。这种花费不是一次性的，它会随时间推移而不断增加。建立一个专家系统的成本取决于人力、资源和时间。同时，正如其他软件和硬件一样，你需要计算维护费用。这引导我们使用**智慧第一法则**（First Law of Wisdom）：如果你买了一辆车想不担心油费，最好拥有自己的油田。以及**智慧第零法则**（Zero Law of Wisdom）：最好是用别人的汽车和汽油。当专家不在的时候很难维护一个专家系统。

由于同行之间的竞争，引入另一个专家并不能起作用。相对而言，建立一个知识来源于人们、文档和其他大众化来源的基于知识的系统要容易得多。很多公司都做了这项工作，因为他们发现这些系统除了一些小问题外，可以 90% 地解决问题。这比建立一个电话服务中心便宜得多。

运行一个专家系统工具除了软件和硬件以外，还得考虑培训的成本。如果你的雇员没有什么经验的话，就要花费大量成本进行培训。任何一个软件的专业培训通常要每人每星期 2500 美元。当然，这一点已不再像过去一样成为不利因素。现在公司可以通过提供产品的在线或课堂培训、认证考试、认

证考试辅导，培养认证考试教师、书籍、研究会、专题讨论会和上门服务而获取利润。今天，把一个容易使用、功能强大、没有漏洞的产品投入市场注定会失败。

6.3 开发专家系统的步骤

系统是如何开发的

从大的方面来讲，专家系统的开发依赖于所提供的资源。然而和别的项目一样，开发也依赖于组织和管理。一个好的参考见（Wentworth 94），联邦高速公路管理局的具有超过 300 条规则的专家系统就是基于此开发的。如果你已经熟悉一个基本的项目计划工具例如 Microsoft Project，那么坚持用这个工具。这就更容易向管理层解释项目进度和项目拖延原因。

项目管理

基本的项目管理技巧和软件工具需要包括下列部分：

活动管理

- 计划
  - 确定活动内容
  - 规定活动的优先次序
  - 资源需求
  - 阶段目标
  - 时间
  - 责任
- 日程安排
  - 规定开始和结束时间
  - 解决同等优先权任务之间的冲突
- 记事
  - 监督项目性能
- 分析
  - 分析计划、日程和已记事的活动

产品配置管理

- 产品管理
  - 管理不同版本的产品
- 变更管理
  - 管理变更建议及其影响评估
  - 指定人员进行改变
  - 安装新版本

资源管理

- 预测资源需求
- 获取资源
  - 明确专家和知识基础
  - 为专家的便利而组织时间表
- 规定合理使用资源的责任
- 提供紧要资源以减少瓶颈问题

特别地，在资源获取中要让你的时间表去适应专家，而不是相反。除非你付给专家全职报酬，他们有其他的工作，时间非常宝贵。

图 6.2 从系统所经步骤的角度给出了开发一个系统所需的理想高层活动。

从理想角度，在解决所有漏洞之前产品不能交付给终端用户。这一点特别适用军用产品，因为取消一个已经启动的核导弹并不容易。

与政府合约相反，商业界的规则很不一样，因为如果项目延期，就没法保证会有稳定的资金供给和额外的预算。今天商业届遵循的范例是每季度花费大量费用在产品宣传上直到产品已投入市场几年后。这有两个目的。首先，想使用产品的私人开发者愿意付费给发布前版本，即使产品还有漏洞。这

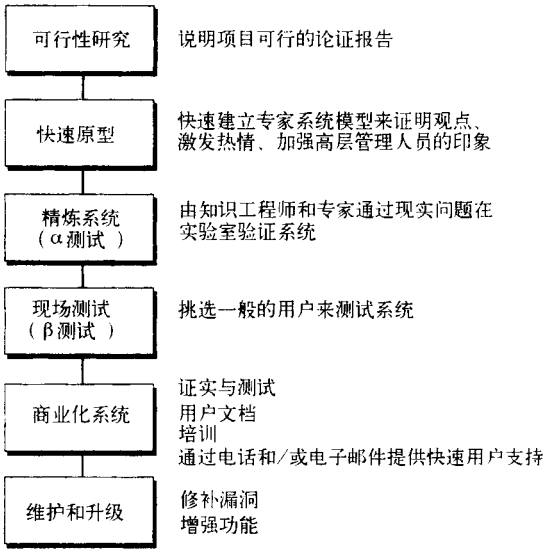


图 6.2 开发专家系统的一般步骤

就提供了一个持续的收入来源，就好像产品已经开发好了，避免了公司需要连续几年负担所有的开发费用。另一个好处是你不需要做太多的内部测试。

只需要确信在每次产品运行崩溃时，询问用户是否发送一个自动的漏洞报告。当用户点击确定时，漏洞信息和判断哪里出错的诊断信息会自动送出。公司的自动邮件系统可以存放漏洞报告到数据库中。标准的数据挖掘工具就可以找出模式提醒各个开发队伍负责相应模块的更正。可以设置不同的激发层次，只有当漏洞数量达到激发层次要求时，相关的信息才会送到开发队伍。有些团队的工作比其他的更加重要，因此不用设置一样的激发层次。根据漏洞的数量和严重程度分配资源，以便在下一季度发布版本中修复这些问题。需要为发布前的版本建立一个专门网站，提供获取新的季度版本的预定费用。

第二，也是最重要的，证券交易委员会（Securities and Exchange Commission，SEC）要求各公司定期向股东发布季度报告。即使你仍是私人公司还没有发布公共股票，这也可为你的网站吸引风险投资，增加你的原始公共积累。

发行问题

系统如何发行

由于依赖于专家系统的配置数量，所以**发行问题**（delivery problem）是开发中的一个主要考虑因素，发行问题必须在开发的最早阶段进行考虑。在 20 世纪，用于开发的系统比把产品部署到最终用户的系统耗费更大。现在由于低价机器的存在，发行问题的这个弊端已经解决。

但是，另一个问题是你的产品应该能在多种硬件设备上运行。这一点依赖于 Java 最初的流行口号“一次编写，任何地方使用”。口号和保证只能用于“砂盒”，配置在不同的硬件仍然是一个问题，除非使用解释器并在每一种不同类型的硬件上安装虚拟机。自从机器代码解释器在 20 世纪 70 年代随着 Pascal 引入后就可以成功应用于任何微型计算机，使得这样的方法可行。而 Java 采用了同样的技术。

尽管 CLIPS 并不产生机器代码，它只产生标准 C 版本的完整专家系统。这样就可以使用标准的 C 语言编译器为特定的硬件平台编译，为特定平台产生可执行的代码，而不需要“CLIPS 虚拟机”。最初设计 CLIPS 的一个目标是使它可以很方便地为任何可能产生的新硬件提供一个专家系统。如果这还不够，其他版本的 CLIPS 例如前言中提到的 Jess 已经开发出来。Jess 是用 Java 开发的，尽管它并不支持多重继承，但它具备 Java 的所有优点，可以很容易地配置在不同硬件上。

很多情况下专家系统必须和其他现存程序集成在一起，应考虑到如何通讯和协调专家系统与这些程序的输入/输出。此外，把专家系统作为传统程序语言的一个过程来调用也具有必要，而系统必须能支持这一点。CLIPS 可被**主语言**（host language）例如 C++程序调用，完成功能，然后把控制交还给主程序。这样一个混合系统也是 CLIPS 设计考虑的主要因素之一，因为一个专家系统的效率随着使用规则和模式匹配的增加会快速下降。尽管 Rete 算法很优秀，专家系统仍不能像编译的代码一样快。

尽管你可以使用 CLIPS 模拟一个计算器，电子报表或甚至一个文字处理器，但是它们的性能不会比用第三代语言如 C、C++、C#编写的程序好。例如，你可以从标准数据库如 Oracle 或 MySql 中调用 CLIPS 创造一个智能的数据挖掘工具，然后把建议返回给数据库用户。在 Oracle 中，游标用来使其他应用程序可以使用数据然后把控制交还给 Oracle。CLIPS 遵循这一规则，既是一个好的宿主，也是一个好的从客。另一个可选方法是使用和更改 CLIPS 源代码，把它融合到你的应用中，然后把结果编译成新的应用发布。

维护和升级

如何维护和升级系统

专家系统的维护和升级较之传统程序来说更为开放，因为专家系统不是基于算法，它的性能依赖于知识，当新知识被获取而旧知识被修改时，系统的性能就提高了。

但是存在一个问题，基于算法的传统程序不像专家系统一样会有规则冲突。规则冲突并不可怕。事实上这是推理机被开发的原因。规则冲突只是意味着有多于一条规则符合规则左边模式，可以被激发。人们考虑问题一直是这种方式。

例如，考虑**悖论游戏**（Game Paradox）。如果你坐在沙发上看电视节目，一手拿着啤酒一手拿着薯条，你会把什么放到嘴里？一条规则说“如果有食品则吃”，另一条规则说“如果有啤酒则饮”。由于既有食品又有啤酒，导致了规则冲突。

幸运的是，**智慧负一法则**（Minus One Law of Wisdom）说：如果你没有一个特定的目的地，就随便走，因为你永不知道什么是迷路。悖论游戏的答案就是喝泡着薯条的啤酒然后吃一喝。

作为一个商业系统，专家系统发行后的改进工作同样关系重大，商业系统的开发者希望它能取得经济成功，这意味着听取用户的需求并了解用户希望并乐意为什么样的改进而付款。在商业世界，一次又一次地证明了人们会为软件更新而不顾一切地付款，因为渴望漏洞更少。

自然，当新的特性增加时不会发生这种情况。事实上，发布包含成千上万个漏洞的产品是现实可行的。这实际是与盗版行为作斗争，因为合法用户可以到网站上下载补丁，直到获得服务包，然后有更多的补丁直到获得另一个服务包，一直到最后，公司发布新的产品或者因操纵股票价格而面临 SEC 的惩罚。就如所有这些软件一样，专家系统永远不会完成，它只会变得越来越好。

6.4 开发过程中的误区

专家系统开发的潜在主要错误可根据它们最可能在哪个阶段发生而进行分类，如图 6.3 所示。这些错误包括：

- **专家知识错误**（expertise error）。专家是专家系统的**专家知识来源**（expertise source）。如果专家的知识是错误的，则会影响到系统的整个过程。正如第 1 章所说，建造专家系统的一个好处就是当处理专家知识时，可对错误知识进行潜在检测。专家并不意味着永远不会犯错。他们只是在他们的专家领域中比普通人较少犯错。比如说你可以为别人进行脑外科手术，但是除非你是一个受过训练的神经外科医生，否则你的病人不可能生还。另一方面，我们也从电视广告中听到，如果你被一个专家伤害，会有很多律师愿帮你获得赔偿。

对那些人们生命和财产具有风险的**紧要**（mission-critical）任务，必须建立一个正规程序来验证专家的知识。在 NASA 里使用的一个成功方法是**飞行技术小组**（Flight Technique Panel），它会对开发过

程中所使用的问题的求解和分析技术进行定期检查。该小组由系统用户、独立的领域专家、系统开发人员、管理人员等所有可能对系统开发具有影响的人来组成。

这种方法的优点是将专家知识的详细检查限定在开发的初始阶段，此时错误最容易校正。错误发现得越晚，校正的代价就越大。如果专家知识不被较早校正，将难以通过最终的系统证实测试，系统的证实测试将证明系统是否满足要求，特别是求解的正确性和完备性。

大公司总是使用小组的形式。有时他们称之为性能检查小组或一个更中性的词。在 20 世纪 70 年代到 80 年代，在软件开发中通过允许开发者相互察看代码来构成各种检查小组。不幸的是，一旦管理者开始用“每天多少行正确代码”、“每 100 行代码多少错误”等来衡量程序员的工作表现，这很快退化成一个糟糕的管理监察手段。尽管管理者并不了解成对检查过程的细节，而只是相信程序员们应该是诚实的，但是当受升迁降职影响时，有些组队并不那么“公正”。只有当从一个恰当的角度看待时，小组才是一个有用的工具，如果涉及到信息的可信度问题，“公正”因素应该被考虑在内。

另一个有用的帮助是**焦点小组**（focus group）。焦点小组对发现什么东西会有好的销售量非常有帮助。在开始新产品的开发之前，大公司通常雇请人们讨论有关主题，这样他们会听到没有偏见的观点，了解产品是否真有市场需求，需要有什么样的特性。采用焦点小组的缺点在于额外的时间和金钱花费。但是这种花费可被更有效的开发过程抵销。如果产品根本没有需求，焦点小组可以为公司节省成百万的投入。

把市场决策交给系统开发人员不是一个明智的做法，除非开发者正针对和他们具有相同**文化**（culture）的人开发产品。这里文化是技术意义上的，指**企业文化**（corporate culture）或最终用户文化，是用于把人分成不同类型的共同价值观。自从 20 世纪 80 年代公司试图定义他们与众不同的地方，这个概念非常的流行。但是在全球化的今天，越来越少用了。

例如，最成功的视频游戏是由整天玩游戏的游戏迷开发出来的。因此，这些开发者真正了解市场，具有游戏文化的公司也会雇佣开发技术不高但是非常喜欢游戏的开发者。这就阐释了**智慧负二法则**（Minus Two Law of Wisdom）：如果你要开始一个旅程，你最好喜欢旅游。在 20 世纪 90 年代，公司花费大量时间和金钱去定义他们的公司文化、TQM 以及其他“潮流趋势”。这些仍在继续，公司必须适应时代的变化。

但是一旦公司走向大众，就很难保持一个与众不同的企业文化，因为必须交付较好的季度报告给股票市场。因此一个初始建立计算机公司的总裁可能会被一个软饮料公司的总裁取代（注意：对于那些对使用负数智慧法则感兴趣的人来说，哲学工程的这种应用是有效的）。

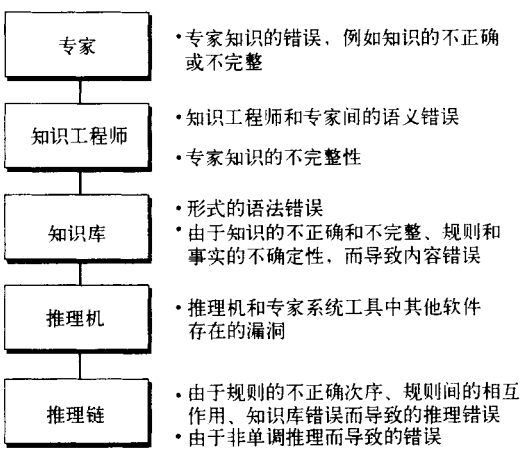


图 6.3 专家系统的主要错误及一些起因

• **语义错误**（Semantic Error）。由于交流不当而产生的错误。例如一个非常简单的例子，一个专家说“你能用水灭火”，而知识工程师理解成“所有的火都能被水灭”。当知识工程师误解专家的回答、专家误解知识工程师的问题、或两者兼有时就会发生语义错误。

误解产生的原因之一是因为符号语言关联触发了误解。例如，假设一个新的工人被雇佣，他的办公室在你的隔壁，每天当他来工作的时候，你觉得有一种讨厌的气味。终于，你遇到他，并为他所说的话震惊：“It’s good to beat you”，而他实际上是说“It’s good to meet you”（办公室保安摄像记录）。尽管人们在听或看，我们的其他感官也同时工作着，如果你把某些不愉快的事与某人关联，这种感觉将渗透到交流中。

这一点在和专家面谈时非常重要。应该有一个专门的面谈室，具有中性色调的地毯和墙壁，空气新鲜没有香气，舒适但不至于昏昏欲睡的沙发，并且安静。专家将比较放松并开始把专门的面谈室和分享领域知识联系起来。最糟糕的面谈地点是你的办公室，专家在别人的“地盘”会觉得不舒服。但在专家的办公室面谈也不好，因为不停地有电话、电邮和“只需一二分钟”的来客打断，以及在面谈结束后还有其他工作等着的提醒。自然他们会想尽快完成面谈以便返回到“正常工作”中。

另一个在面谈中比较普遍发生的错误是由于认知错觉（Pohl 04）。一个错觉是你知道不存在，但看起来似乎真的存在，就如沙漠中的海市蜃楼。人们很难驱逐错觉，特别是和感知相关时。如果你看过经典的 Muller-Lyer 箭头图，你知道对两种箭头线的长度是一样的，但你的眼睛认为它们不一样长（如果你对此有疑问，你可以用尺子量量看）。



这种错觉产生的原因是你的视觉系统与你的思考、认知分离。你的大脑在黑板结构模型上进行操作，其中专门的神经集簇进行着不同的活动。可把这些集簇看作大脑中的与普通软件代理相对应的硬件代理。视觉系统作为一个重要的生存反应机制，毫无疑问地比高级认知部分更早被激发。

认知错觉随时都会出现而且很难避免。例如，加州的洛杉矶和内华达州的里诺哪一个更西边？多数人会立即说洛杉矶，因为它靠近太平洋而里诺在内华达州的内陆。这就是一个常识推理的绝佳例子。不幸地是，这是错误的。里诺确实比洛杉矶更西边一点，你可通过地球仪或在网上查到。另一个例子，罗马和纽约哪一个更北边？我们看了很多电影表现意大利的阳光和纽约的雪，所以常识告诉我们纽约更北边。同样这也是错的。纽约比罗马更南边。这就是认知错觉，即使你知道这一点，你也会发现很难相信，就像上面的箭头图。

这样的认知错觉是很难发现的，特别是在和专家面谈的时候。他是一个领域的专家并不意味着他也是别的领域的专家。作为人，他们和其他人一样容易受认知错觉的影响。危险在于你可能认为他们说的任何事情，甚至他们专业领域以外的内容，都是正确的（当然，对你的老板而言，这一点是肯定的，你应每天都这么说，特别是在发薪水的那天）。

• 语法错误（Syntax Error）。这些是简单的错误，当规则或事实的输入形式不正确时就会发生。专家系统工具应标出该类错误并给出适当提示信息。还有其他一些错误是在知识库建立过程中，由于在早期阶段没能检查出的知识源错误而产生的。

• 推理机错误（Inference Engine Error）。和软件的其他部分一样，推理机也会有漏洞。在一个专家系统工具被发布时，所有的一般漏洞都应被修改。然而还有一些漏洞仅仅出现在少数情况下，例如当 4 月 1 日一个议程中有 159 条规则时。有些漏洞可能会非常隐蔽，它们仅仅出现在特定的模式匹配操作中。

一般，推理机漏洞会出现在模式匹配、冲突归结和动作执行中。如果这些漏洞是偶尔的，将很难检测到。所以，如果专家系统工具是应用在紧要任务中，则必须决定如何来证实工具。通常专家系统

工具的每个新版本都会有一系列测试，包括 CLIPS。测试是自动的，以便给出无偏的结果。但一般不必担心，因为是管理层由于外部的压力而坚持投放市场的。

检测工具错误的最简单方法是询问其他用户和工具销售商的传统方法。销售商会很乐意提供一个顾客、漏洞报告、错误改正、以及工具使用了多久的清单。用户群是一个优秀的信息来源。

• **推理链错误**（Inference Chain Error）。这些错误可能由错误的知识、语义错误、推理机漏洞、规则优先级错误和规则间无计划的相互作用等引起，更复杂的推理链错误是由于规则和证据的不确定性、这些不确定性在推理链中的传播、非单调性等引起。

选择处理不确定性的方法并不能自动解决所有有关该方法的问题，例如：在选择简单的贝叶斯推理前，你必须检查条件独立性是否成立。

• **未知界限错误**（Limits of Ignorance Error）。对于所有开发阶段来说，指明系统未知界限是一个共同的问题。正如第 1 章所提到的，人类专家知道他们的知识限度，且在边界区域其性能会逐渐降低。人类专家应坦率地承认在这些边界他们的结论是很不确定的。然而，除非专家系统专门编程来承认这些不确定性，否则，即使推理链和证据很不充分，专家系统也会给出答案。更糟糕地是，人们会相信这些答案是可靠的。

6.5 软件工程与专家系统

在前面章节，我们讨论了使用专家系统范例时的一般问题，现在让我们从实际建立系统的知识工程师的技术角度来纵览专家系统的发展阶段。

自从专家系统走出研究阶段，就已推出达到传统软件标准的高质量软件的现实需求。开发符合商业、工业和政府标准的高质量软件的合适方法是软件工程。

在产品开发中遵循一个好的标准是很重要的，否则，产品质量可能会不好。现在，专家系统应被当成如文字处理、工资程序、电脑游戏等一样的一个产品。

然而，专家系统和典型的消费系统，如文字处理和视频游戏的**使命**（mission）有很大的不同，术语“使命”是指一个项目或一个技术的全部目的，专家系统技术一般有一个重要的使命，就是提供高水平的专家建议给人类生命财产可能处于危险的情况，这就是前面小节所提到的紧要应用。

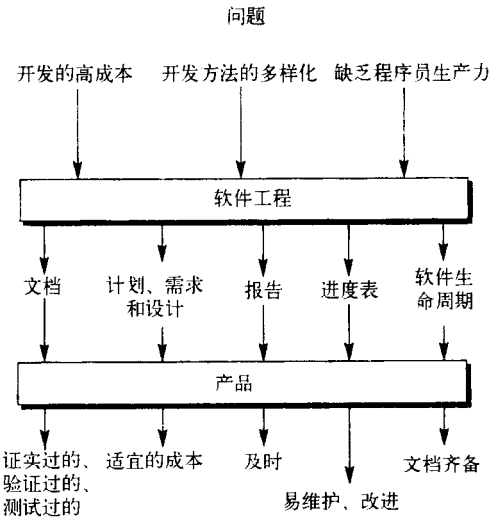


图 6.4 软件工程方法

紧要使命应用与文字处理和视频游戏等轻松应用不同，它们只是为了提高效率和提供娱乐，没有谁的生活会取决于一个文字处理或一个视频游戏软件中的漏洞（或者至少不必取决于）。

专家系统属于高性能系统，它必须是高质量的，否则，将容易出错。软件工程提供了建立高质量软件的方法，如图 6.4 所示。

在一般意义上，很难描绘**质量**（quality）这个词，因为它对不同人有不同的意思，定义质量的一个方法是以某种尺度决定一个对象所必需的属性，词语“对象”在这里表示任何软件或硬件，例如一个软件产品，属性和它们的值称为**计量**（metrics），因为它们用来衡量一个对象。例如，磁盘驱动器的可靠性是它的一个质量计量，该属性的一个测试方法是磁盘驱动器**平均出错时间**（mean time between failures，MTBF），一个可靠驱动器的 MTBF 可能为 50 000 小时，而一个不可靠驱动器的 MTBF 可能为 10 000 小时。在 20 世纪，人们在工作中和家里可能只有几个小时使用电脑。现在习惯于

让电脑 24×7×52 小时开着，使用的时间变多了。当然你可以设置你的操作系统关闭驱动器如同关闭显示器。但是反复打开和关闭这些设备可能引发其他问题。

重要的不仅仅是时间，而是书写时的错误率。假设一个磁盘驱动器每写十亿次 byte 就会有一个错误。在 20 世纪 90 年代普通的磁盘驱动器是 20 Mbyte，意味着可写 50 次磁盘不会有写错误。现在磁盘驱动是 200 Gbyte 的数量级，每十亿次有一次错误意味着如果错误率相同，会有 200 个错误在你的硬盘中。

当写入一个压缩数据如 .jpg 时出现错误，那可能不会引发问题。但如果错误出现在一行代码中，一个单一字节的错误可能会使一个程序语句由 X=A 变成 X=B，这将导致潜在的非常坏的后果。现代驱动器有一个内置感应器持续监视磁盘状态，使用一种称为自检、分析和报告的技术 SMART（Self-Monitoring，Analysis，and Reporting Technology）。程序可以存取 SMART，例如在 Norton 系统工具中运行磁盘诊断工具时。这样就可以提前预告可能存在的硬件错误，不过你必须在数据丢失前通过备份和更换磁盘来预防。

SMART 是一个预测磁盘情况的工具，但是没有十全十美的东西，如果涉及到被其他磁性媒体破坏时则 SMART 不能处理。在生产和操作过程中硬盘表面的磁性覆盖物受损越微小，越容易在写磁盘时出现错误，从而导致软件错误。硬盘磁表面上读写的数据越多，驱动器运行时间越长，部件就越容易磨损。驱动器越新运转越快，因为旋转的速度更快，磁性表面也就越完整没有缺点。

在 20 世纪 90 年代，标准是 3600 RPM，然后是 5400 RPM，现在是 7200 RPM，并已升到 10 000 RPM。作为类比，假设你的汽车轮胎是 2 英尺直径的，圆周大概是 6 英尺，如果以 3600 RPM 转动，则每小时 250 英里，如果是 10 000 RPM，则每小时 700 英里，如果你的硬盘转得像汽车轮胎一样快的话，就超过了音速！过去，计算机系统的最薄弱之处就是运转元件，由于它们产生的摩擦和热量。

随着 64 位处理器的出现又引发了新的弊端，因为它比 32 位处理器发热量更大。这是一个严重的问题，新的主板必须在微处理器上方设置温度传感器，驱热风扇。假设 6 个风扇为处理器降温，并且温度过高时软件会自动关闭系统。我们仍需要把磁盘驱动器和内存放在远离 CPU 的地方。

表 6.1 给出了一些可衡量专家系统质量的计量。这些计量仅作为一个指导，因为一个特定的专家系统可能比这些多或少，然而，重要的是必须有一个计量表，以便用来描述质量。

由于计量间可能会有冲突，所以计量表使你更容易确定它们的优先级。例如，加强专家系统测试以确保它的正确性将会提高成本。决定一个测试什么时候结束，通常是一个复杂的事情，它涉及到进度表、成本和需求，最理想的是这三方面的要求都得到满足。实际上，我们可能会决定某些方面比其他方面重要，而不是满足所有方面。

表 6.1 专家系统的某些质量计量

软件质量计量
正确的输入得到正确的输出
正确的输入得到完整的输出
同样的输入得到一致的输出
可靠，不会因为漏洞而崩溃
易用，用户界面友好
易维护
易扩展
确认能满足用户的需求
测试证明其正确和完整
适宜的成本
可重用代码以供其他应用使用
易移植到其他软/硬件环境
对其他软件提供良好的接口
代码的可读性
准确性
精确性
在知识边界的能力降低缓慢
嵌入其他语言的能力
证实过的知识库
解释机制

6.6 专家系统生命周期

软件工程的一个关键方法是**生命周期**（life cycle），软件的生命周期是指从软件原始概念开始到它退出使用为止的一段时期。与开发和维护分开不同，生命周期概念提供的是所有阶段的连续性，在生命周期的早期就计划好维护和发展会减少以后阶段的成本。

维护成本

对于成功的传统软件，每隔几年都有修改和扩展，维护成本很容易就达到最初开发成本的 10 倍。

虽然这似乎很多，正如我们在前面已说过的，维护需要很多钱，但是**智慧负三法则**（Minus Three Law of Wisdom）说：如果你不喜欢旅行，就喜欢你现在所在之处。因此，如果你一直在修改你的产品漏洞，那么不要换一个全新的名字，可在旧名字上作变化，但不要命名为："Doors 漏洞修复第 10 版"，而要称为："Doors 2010"。

专家系统更多地依赖于知识而不是人类专家。但也有例外，如对一些专业人员，比如医生和飞行员的智能训练。专家系统作为训练系统非常成功，它可以减少专家花费在教一些介绍性知识上的时间，不过，如果花费足够的时间和资源在专家系统上，它也可真正教好一个学生做某事。

随着技术的变化，就没有很大的必要再保留一个退休专家的知识。专家系统需要很多维护是因为它是基于大量的启发式和经验知识，专家系统在不确定性情况下做大量的推理，这使它更易有高的维护和发展成本。随着专家系统中建立和输入本体的自动工具的出现，专家系统的维护变得更容易，但是系统也变得越来越大，就像所有软件一样，这一直是一场赛跑。

瀑布模型（Waterfall Model）

传统软件的许多不同生命周期模型已经开发出来。为传统程序员所熟悉的，经典的瀑布模型是最早的生命周期模型，如图 6.5 所示。

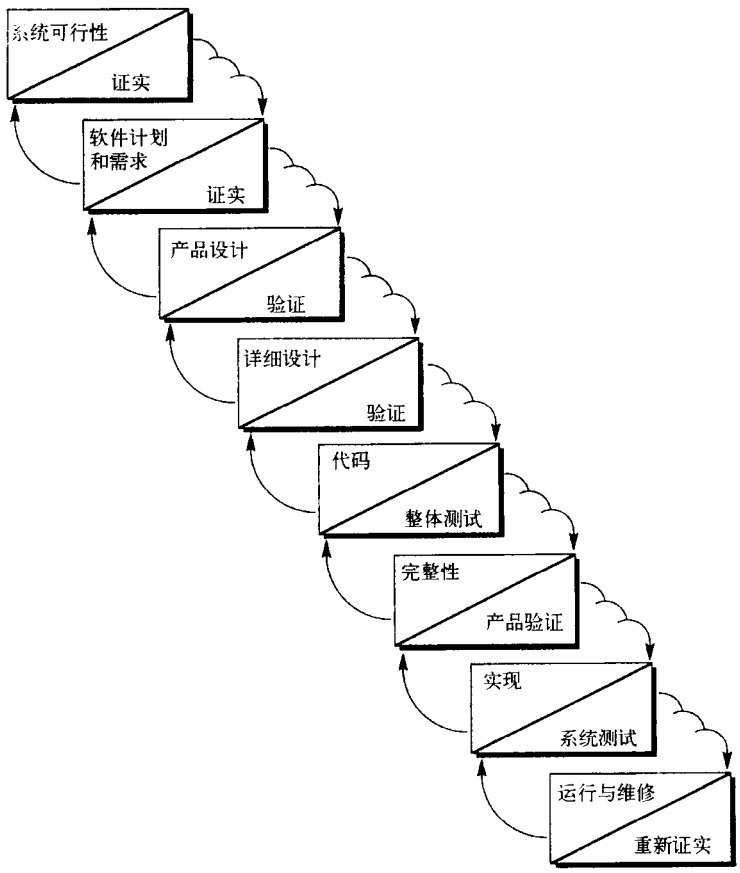


图 6.5 软件生命周期的瀑布模型

这个模型为传统程序员所熟悉。在瀑布模型中，每一个阶段都以验证和证实（verification and validation，V&V）活动结束，以便将该阶段的问题减小到最少。注意向前和向后箭头在某个时刻只处于一个阶段，这表示重复开发只在两个相邻的阶段，从而减少了在几个阶段重复开发所需的高成本。返回阶段的花费不是简单的线性函数，而是以指数增长。

生命周期也称为**过程模型**（process model），因为它涉及软件开发的两个基本问题：

(1) 下一步该做什么？

(2) 下一步完成需多久？

过程模型实际上是一个**元方法论**（metamethodology），因为它决定一般软件方法的顺序和时间。一般软件开发方法（或方法论）：

描述实现一个阶段的特定方法，这些阶段有：

- 计划
- 需求
- 知识获取
- 测试

阶段产品描述：

- 文档
- 代码
- 图表

代码-修改模型

许多过程模型被用于软件的开发，最早的“模型”是声名狼藉的**代码-修改模型**（Code-and-Fix Model），在这种模型中，写完一些代码，若发现有错误，再进行修改。很多程序设计新手在传统程序设计和专家系统中都常常采用这种方法。

到1970年，这种模型的缺陷已经十分明显，于是，开发了瀑布模型以提供一个系统的方法论，这特别适合于大型项目的开发。然而，瀑布模型有一个难处，因为它假定一个阶段所需的所有信息都是已知的。实际上，往往直到一个原型建立起来以后才有可能写出完全的需求。这导致了**做两次**（do-it-twice）的概念：建立原型、确定需求、然后再建立最终系统。

增量模型（Incremental Model）

增量瀑布模型（incremental waterfall model）是瀑布模型和标准自顶向下方法的改进。增量开发的基本思想是通过逐步增加功能来开发软件。这种模型已成功运用在传统的大型软件上。增量模型对专家系统的开发也非常有用，通过规则的增加，系统能力从助手级到同事级，最后到专家级。这样，在一个专家系统中，**大增量**（major increment）是从助手到同事及从同事到专家。

小增量（minor increment）是在每个层次中由于大的改进而使专家知识有所增加，而**微增量**（microincrement）则是由于增加或精炼个别规则而使专家知识略有改进。

这种增量模型的最大好处在于功能的增加较之瀑布模型的每个阶段产品来说更易测试、验证和证实。且相对于最后的整个测试，每个新增功能更易被立即测试、验证和证实。这样就降低了系统的混合修正成本。本质上，增量模型类似于一个波及到整个开发过程的连续快速原型。比起“做两次”方法中仅仅是为了决定需求的初始阶段快速原型，这种模型的原型是系统。

螺旋模型

增量模型的一个具体体现是传统**螺旋模型**（Spiral Model）的改进，如图6.6所示。螺旋的每一个环路都对系统增加一些功能。标有“交付系统”

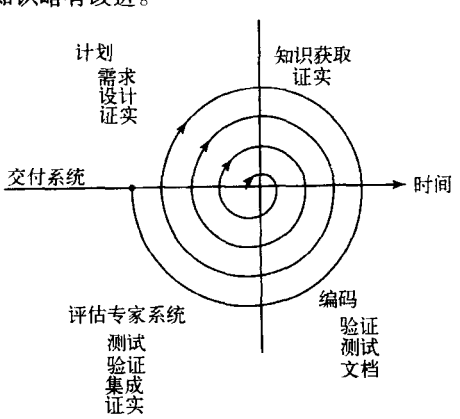


图6.6 专家系统开发的螺旋模型

的结束点实际上并不是螺旋的终结。相反，新的螺旋是以系统的维护与发展开始的。该螺旋可以进一步优化以更清晰地说明知识获取、编码、评估和计划的整个阶段。

6.7 详细生命周期模型

一个成功地用于许多专家系统项目的生命周期模型是**线性模型**（Linear Model），如图 6.7 所示，这个模型包括从计划到系统评估的许多阶段，它对系统开发的描述一直到功能评估这种程度上，之后，它的生命周期重复着：从计划到系统评估，直到系统交付正常使用。然后，生命周期再进入随后的系统维护与发展。虽然没有明确地示出，但验证与证实与这些阶段是并行进行的。与仅仅是修改某些漏洞不同，沿着开发阶段的同样次序对专家系统的质量进行维护是十分重要的。跳过这些阶段，即使只是修改一个小漏洞，亦会损害整个系统。

生命周期可以认为是螺旋模型的一条环路。每个阶段都包含**任务**（tasks）。当然并非所有任务对某个阶段都是必要的，特别是当系统进入维护与发展时。这些任务是整个生命周期，从初始构想一直到软件退出中所有任务的一个合成。它们依赖于具体的应用，且只能作为一个指导，而不是每个阶段所必须完成的绝对要求。

我们将详细探讨该生命周期模型，以说明对一个大型高质量专家系统来说，有很多因素都必须考虑。对那些不打算作为一般用途的小型研究原型来说，并非所有任务或每个阶段都是必需的。不过，又有多少设计作为个人或研究用途的软件获得许可，进入了一般用途呢？

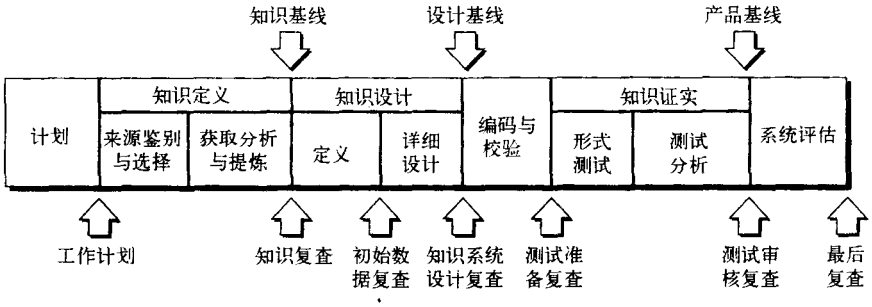


图 6.7 专家系统开发生命周期的线性模型

计划

计划阶段的目的在于产生一个专家系统开发的正式**工作计划**（work plan），该工作计划是用来指导和评估开发的一套文档。表 6.2 列出了该阶段的任务。

表 6.2 计划阶段的任务

任 务	目 的
可行性评价	决定系统是否值得建立，如果值得，是否要用到专家系统技术
资源管理	估计所需的人力、时间、软硬件。如何获取和管理所需资源
任务指派	具体说明各任务和在阶段中的顺序
时间表	指定各阶段中任务的开始和完成日期
初始功能设计	通过指明系统高层功能来定义系统必须完成什么，这个任务说明了系统的目的
高层需求	从高层的角度描述系统功能该如何实现

可行性分析是生命周期中最重要的任务。它必须回答这个项目是否值得做以及相应的选用专家系统是否合适。这两个问题的答案决定了项目是否应用专家系统方法进行。可行性分析涉及许多因素，正如第 6.1 节所讨论的那样，这些因素包括选择一个合适的专家系统问题领域、成本、收益等等。

知识定义

知识定义阶段（knowledge definition stage）的目的是定义专家系统的知识需求，包括下面两个主要任务：

- 知识来源鉴别与选择
- 知识获取、分析与提炼

每个主要任务由其他任务组成。表 6.3 描述了知识来源鉴别与选择的任务。

获取、分析与提炼任务在表 6.4 中描述。

表 6.3 知识来源鉴别与选择任务

任 务	目 的
来源鉴别	知识来源于谁，来源于什么，不考虑可获取性
来源重要性	以开发的重要性次序排列知识来源
来源可获取性	以可获取性排列知识来源，Web、书籍和其他文献一般比专家更容易获取
来源选择	根据重要性和可获取性选择来源

表 6.4 知识获取、分析与提炼任务

任 务	目 的
获取策略	详细说明知识如何获取的方法，如访问专家、参阅文档、规则归纳，搜集积累等等
知识成分鉴别	从来源中挑选出在生命周期的该次反复中有用的特别知识
知识分类系统	对知识分类和组织以帮助开发者验证和理解。尽可能使用层次分组法
详细功能设计	详细说明系统的功能。这是从更为技术的层次，而初始功能设计是从管理的层次
初始控制流程	描述专家系统执行的一般阶段，这些阶段对应于那些通过激活/不激活来控制执行流程的逻辑规则
初始用户手册	从用户角度描述系统，这是系统必须有但常常被忽略的一个部分。应尽早有用户参与以便得到反馈信息，这是至关重要的。如果他们不使用系统，那么所做一切都是徒然的
需求说明	准确定义系统意图。专家系统将根据这些需求来进行证实
知识基线	规定系统的知识基线。此时任何改变都必须有一个正式的改变请求。此时所得到的高层知识可用于下一阶段的知识设计

知识获取任务、知识分析任务和知识提取任务的主要目的是产生和验证系统所需要的知识，从知识被确定基线开始，它应是正确的和合适的以便进行下阶段的知识设计。除了常用的访问专家方法外，许多软件工具也可用来实现自动知识获取，这些工具由软件工具商提供。通过知识工程师从专家那里提取知识的知识获取方法需要很多技巧，在（Grzymala-Busse 91）中有更多讨论。

知识设计

知识设计阶段（knowledge design stage）的目的是产生专家系统的详细设计。该阶段包括两个主要任务：

- 知识定义
- 详细设计

表 6.5 描述了知识定义的任务。

表 6.5 知识定义任务

任 务	目 的
知识表示	说明如何表示知识，例如规则、框架或逻辑。这依赖于专家系统工具支持哪种表示
详细控制结构	说明三种一般控制结构：(1)如果系统嵌入到过程代码中，它将怎样被调用；(2)系统执行中有关规则组的控制；(3)规则的元级控制结构

（续）

任　务	目　的
内部事实结构	以一致方式说明内部事实结构，以帮助理解和优化设计
初始用户界面	说明初始用户界面，从用户处获得有关界面的反馈
初始测试计划	说明代码将如何测试。定义测试数据、测试装置以及如何分析测试结果

表 6.5 中描述的内部事实结构将在关于 CLIPS 的章节中更详细地讨论。明确事实结构的基本思想是便于优化设计。例如，像“10”这样的事实本身意义并不明确。“10”代表什么呢？如果在事实中含有附加信息，如“价格 10”或者更进一步，“黄金价格 10”，那么黄金价格就是有意义的。注意事实的这种形式是传统的对象－属性－值形式，因此很方便人们阅读与理解。CLIPS 通过规则的自定义模板结构以及对象机制支持这种形式。

在某些专家系统语言中，域可能具有**强类型**（strong typing）以致只允许特定值。如果某一规则想指定一个不允许值，则推理机将标识这是一个错误。知识的**详细设计阶段**（detailed design stage）如表 6.6 所示。

详细设计阶段的成果是原始设计文档，依据它才能开始编码。这些原始设计文档需经历知识系统设计复查，这是代码生成前的最后一次检查。

表 6.6　知识的详细设计任务

任　务	目　的
设计结构	说明知识在知识库中是如何逻辑组织的，并说明知识库中有什么
实现策略	说明系统如何实现
详细用户界面	说明从初始用户界面设计中收到用户反馈后的详细用户界面
设计说明书与报告	设计文档
详细测试计划	精确说明代码将如何测试和验证

编码与校验

表 6.7 描述了**编码与校验阶段**（code and checkout stage），它从实际的代码实现开始。

表 6.7　编码与校验任务

任　务	目　的
编码	代码实现
测试	使用测试数据、测试装置和测试分析程序测试代码
源代码列表	产生注释过的文档化源代码
用户手册	产生工作用户手册以便专家和用户能得供反馈
安装/操作指南	为用户提供系统的安装/操作文档
系统描述文档	专家系统功能、局限及存在问题的文档

这个阶段结束于**测试准备复查**（test readiness review），测试准备复查决定专家系统是否已为下一阶段的知识验证做好了准备。

知识验证

知识验证阶段（knowledge verification stage）的目的是决定系统的正确性、完备性和一致性。该阶段分为两个主要任务：

- 形式测试
- 测试分析

表6.8描述了知识验证阶段的**形式测试任务**（formal test task）。

测试分析任务（test analysis task）如表6.9所示。测试分析希望解决下面主要问题：

- 不正确答案
- 不完全答案
- 不一致答案

并决定问题是出自规则、推理链、不确定性还是这三个因素的组合。如果问题不是出在专家系统，那么得分析专家系统工具软件的漏洞，这是最后的步骤。CLIPS和其他工具的不同处在于它提供所有源代码以便于你分析漏洞。

表6.8 知识验证阶段的形式测试任务

任务	目的
测试程序	实现形式的测试程序
测试报告	测试结果文档

表6.9 测试分析任务

任务	目的
结果评估	分析测试结果
建议	建议与测试结论文档

系统评估

正如表6.10所描述那样，系统开发生命周期的最后一个阶段是**系统评估阶段**（system evaluation stage）。该阶段的目的是总结从改进和完善建议中我们学到了什么。

表6.10 系统评估阶段任务

任务	目的
结果评估	总结测试与验证结果
建议	建议对系统的修改
证实	证实系统正确实现了用户需求
中期或最终报告	如果系统已完成，发布最终报告。如果未完成，发布中期报告。申请更多经费

由于专家系统的建立是一个反复过程，因此随着新知识的增加，系统评估阶段所形成的报告，常常作为描述系统新增功能的中期报告。但是，系统新功能必须通过本身验证，而且是作为系统原有知识的一部分。也就是说，必须在系统所有知识合取的情况下实施系统验证，而不仅仅是对新知识。此外，每当这一阶段，专家系统都应进行证实而不是等到最后一次反复。目前，有关知识库的自动验证系统也正在研究中。

6.8 小结

在本章中，我们讨论了建造专家系统的软件工程方法。目前，专家系统技术已用于解决现实世界问题包括防御系统等，因此，它必须是一个高质量的产品。

在设计一个专家系统时必须考虑许多因素，如选择问题、成本和收益等。要建立一个成功系统，管理和技术方面都应进行考虑（Gonzalez 93）。

软件工程中一个非常有用的概念是生命周期。生命周期把软件开发从开始构思到退出视为一系列的阶段。然而随着从去除漏洞，修复/增强中赚取更多钱的现代商业范围的出现，好像从不使软件退休更有利。坚实地沿着生命周期，就有可能建立高质量的软件。本章讨论了专家系统的几个不同生命周期模型，并且详细介绍了其中一个。

RuleXML，Rule Markup Initiative 和 RuleML 致力于使规则极其更具可读性（http://www.ruleml.org）。

习题

6.1 思考一个简单的基于知识的汽车问题诊断系统。描述该系统线性模型的每一阶段（不必写出每一

个任务)。假定有很多人参加这个项目，考虑他们之间的协作。并解释你所用的假设。

6.2 写一份关于 KM 的一个商业自动工具的报告。从附录 G 中的链接或网页上找到测试版本下载。列出能发现的所有问题以及其花费。

6.3 试述线性模型对于由许多人开发一个大型项目和一个人开发一个小型项目的不同或区别。

参考文献

(Becerra-Fernandez 04). Irma Becerra-Fernandez, Avelino Gonzalez, and Rajiv Sabherwal, *Knowledge Management,* Prentice-Hall, 2003.

(Begley 03). Good explanation of developing knowledge acquisition in expert systems and summaries of decision making systems in convenient tables, "Adding Intelligence to Medical Devices An overview of decision support and expert system technology in the medical device industry." Ralph J. Begley, Mark Riege, John Rosenblum, and Daniel Tseng: (http://www.devicelink.com/mddi/archive/00/03/014.html).

(Chandler 01). Daniel Chandler, *Semiotics: The Basics*, Rutledge Press, 2001. Note, a shorter online version is at: (http://www.aber.ac.uk/media/Documents/S4B/).

(Conway 02). Susan Conway and Char Sligar, *Unlocking Knowledge Assets: Solutions from Microsoft*, Microsoft Press, 2002.

(Gonzalez 93). Avelino J. Gonzalez and Douglas D. Dankel, *The Engineering of Knowledge-based Systems: Theory and Practice*, Prentice-Hall, 1993. Excellent supplement to this book. Similar coverage with more details, and even includes an older version DOS version of CLIPS discussed in an appendix. It is recommended you use the latest version of CLIPS with this book. Lots of good problems from easy to hard.

(Grzymala-Busse 91). Jerzy W. Grzymala-Busse, *Managing Uncertainty in Expert Systems*, Kluwer Academic Publishers, 1991.

(Malhotra 01). Y. Malhotra, "Expert Systems for Knowledge Management: crossing the chasm between information processing and sense making," *Expert Systems with Applications* journal, (20) 2002 pp. 7–16, available online at: (http://www.brint.org/expertsystems.pdf).

(Pohl 04). Rudiger Pohl, Cognitive Illusions: A Handbook On Fallacies And Biases In Thinking, Judgement And Memory, pub. by Taylor & Francis, 2004.

(Tiwana 03). Book and CDROM with many case studies, lots of software resources, and many links. Amrit Tiwana, *Knowledge Management Toolkit : Orchestrating IT, Strategy, and Knowledge Platforms*, 2nd Edition, Prentice-Hall, 2003.

(Wentworth 94). Good online book based on Federal Highway Administration experience with "PAMEX: Expert System for Maintenance Management of Flexible Pavements," with 327 rules, by James A. Wentworth, Rodger Knaus, and Hamid Aougab, *Verification, Validation and Evaluation of Expert Systems*, (http://www.tfhrc.gov/advanc/vve/cover.htm).

第7章 CLIPS介绍

7.1 概述

本章开始介绍一些实际的概念，这些概念对于建立一个专家系统是必要的。从第1章到第6章我们介绍了专家系统的背景、历史、定义、术语、概念、工具和应用。简而言之，它让我们明白了什么是专家系统和它能做些什么。这种概念和算法的理论框架对建立一个专家系统是必不可少的。然而，对于建立专家系统来说，还有许多实际方面必须通过动手实验来学习。建立一个专家系统很像用一种程序语言来编写程序。知道一个算法的工作原理并不等于会编一个程序来实现这个算法。同样，掌握了专家系统的知识并不等于会建立专家系统。所以，使用专家系统工具的实际经验对学习专家系统是非常重要的。

在本书的后面部分，用专家系统语言CLIPS来展示各种概念。由CLIPS支持的编程范例有3种：基于规则的、面向对象的和面向过程的。第7、8、9章包括了基于规则的程序开发，第10章涵盖了过程化程序设计，第11章包含面向对象程序设计。最后，第12章讨论几个范例问题，以演示前面章节介绍的特性。第7～12章重点讨论了作者眼中最有用的CLIPS特性和会被使用到的内容。尽管这些章节和相关附录可以作为CLIPS编程语言的参考指南，但这些内容首要的是教会你如何写一个CLIPS程序，它们并不是CLIPS所提供的所有特性的详尽描述文档。随书附带的CD－ROM光盘包含了电子格式的指南："CLIPS基本编程指南（CLIPS Basic Programming Guide）"是CLIPS编程的完整参考。当你已经懂得如何编写CLIPS程序但是需要某个特性的详细信息时，它将是特别有用的参考手册。

本章描述CLIPS中具有的基于规则的专家系统的基本组成（如第1章所讨论的）。这些基本组成是：

1）**事实表**（fact list）：包含推理所需的数据

2）**知识库**（knowledge base）：包含所有规则

3）**推理机**（inference engine）：对运行进行总体控制

CLIPS的这3个组成部分将是本章的重点。第一部分——事实，会详细介绍。将要讨论的内容有：添加、删除、修改、复制、显示和跟踪事实。之后，将解释CLIPS程序的规则是怎样与事实相互作用使程序执行的，其中包含模式匹配中变量和通配符的使用。最后，第7.23节将演示多条规则的相互作用。

7.2 CLIPS

CLIPS是一种多范例编程语言，它支持基于规则的、面向对象的和面向过程的编程。基于规则的CLIPS编程语言的推理和表示能力与OPS5相似，但功能更强。在语法方面，CLIPS规则与Eclipse、CLIPS/R2和Jess语言的规则极为相似。CLIPS仅支持正向链规则，而不支持反向链规则。

CLIPS中面向对象的编程能力也就是指CLIPS面向对象语言（CLIPS Object-Oriented Language，COOL），它除加入了许多新的思想之外，还结合了其他面向对象语言的特征，如公共Lisp对象系统（CLOS）和SmallTalk。CLIPS中面向过程的编程语言的特征类似于C、Ada和Pascal语言，语法上类似于LISP。

CLIPS是"C语言集成产生式系统（C Language Integrated Production System）"的首字母缩略词。它是美国航空航天局/约翰逊太空中心（NASA/Johnson Space Center）用C语言设计的。设计目的是可移植性高、成本低和易于与外部系统集成。然而，CLIPS的部分首字母缩略词的含义可能会引起误解。最初，CLIPS仅支持基于规则的编程（所以称之为"产生式系统"）。CLIPS 5.0版本则引进了面向过

程和面向对象的编程。由于CLIPS有一个版本是完全用Ada开发的，所以，缩略词中的“C语言”几个词所代表的含义也会引起误解。本书附带的光盘包含有可在DOS、Windows 2000/XP和MacOS环境运行的CLIPS 6.2版本、电子文档和CLIPS的C源代码。

由于CLIPS具有可移植性，所以，它已安装在多种类型的计算机上，从PC机到CRAY超级计算机都有。本章和以后几章的大多数例子都可以在安装了CLIPS的任何计算机上运行。建议用户学习一些CLIPS所在计算机上用的操作系统的知识。例如，在不同的操作系统中，规定文件的方法常常是有所不同的。依所用的机器或操作系统不同，命令可能也不同，这一点今后还会提到。

7.3 记号

这一章和后面几章将使用相同的记号来描述将要学习到的各种命令和结构的语法。这类记号由3种不同类型的待输入的文本组成。

第一类记号是符号和字符，它们要按所显示出来的那样准确输入。这类记号包括没有被字符对< >、[ ]或{ }括住的任何东西。例如，下面的语法描述：

```
(example)
```

这个语法描述表示（example）应按它所显示的那样输入。确切地说，应首先输入字符“(”，然后是字符“e”、“x”、“a”、“m”、“p”、“l”、“e”，最后输入字符“)”。

方括号[ ]表示括号中的内容是可选的。例如，语法描述：

```
(example [1])
```

表示括号里的1是可选的。因此，下面的输入

```
(example)
```

与上面的语法一致，就像下面的输入与之一致一样。

```
(example 1)
```

尖括号< >表示由括号中的内容规定的一类值要被替换掉。例如，下面的语法描述：

```
<integer>
```

表示要用一个实际的整数值代替它。以前面的例子为例，语法描述：

```
(example <integer>)
```

可替换成：

```
(example 1)
```

或

```
(example 5)
```

或

```
(example -20)
```

或更多的输入，只要包含字符“(example”，后接一个整数，最后是字符“)”就行。我们要注意语句中所示的空格也应该输入，这是很重要的。

另一个记号是语句描述后面的“*”。它表示语句可以用规定的值替换0次或更多次。在每一个值之后应该输入空格。例如，语法描述：

```
<integer>*
```

可以被替换成

```
1
```

或

```
1 2
```

或

```
1 2 3
```

或者任何数目的整数，或者根本没有数据。

描述后面跟一个“+”号的语句表示，该描述所规定的一个或多个值应该被用来代替这语句描述。注意，对于这种记号，语句

```
<integer>+
```

等价于语句

```
<integer> <integer>*
```

一条竖线“|”表示在由此竖线分开的一个或多个选项中选择一项。例如，语句

```
all | none | some
```

可以替换成

```
all
```

或

```
none
```

或

```
some
```

7.4 字段

随着知识库的构建，CLIPS必须从键盘和文件中读取输入，以便执行命令并调入程序。当CLIPS从键盘或文件中读入字符时，它把字符组合起来形成**标记**（token）。标记代表多组字符，对CLIPS有特殊意义。有些标记，如左右括号仅由一个字符组成。

这组标记就是我们所知的**字段**（field），它具有特别重要的意义。CLIPS有8种字段，也称为CLIPS原始数据类型：**浮点型**（float）、**整型**（integer）、**符号型**（symbol）、**字符串型**（string）、**外部地址**（external address）、**事实地址**（fact address）、**实例名**（instance name）和**实例地址**（instance address）。

头两种字段，浮点型和整型，叫作**数字字段**（numeric field），或简单地叫作**数字**（number）。数字字段由三部分组成：符号、值和指数。符号和指数是可选的。符号为 + 或 - 。值包括一个或多个数字，也可包括小数点。指数由字母e或E、后跟一个可选的 + 或 - 、之后是一个或多个数字组成。任何由一个可选的符号、之后跟着的仅仅是数字组成的任何数字都作为**整型**（integer）来存储。所有其他的数字都作为**浮点型**（float）来存储。

下面是CLIPS的合法浮点数的例子：

```
1.5
1.0
0.7
9e+1
3.5e10
```

下面是CLIPS的合法整型数的例子：

```
1
+3
-1
65
```

符号（symbol）是一种字段，它以一个可打印的ASCII字符开头、后接零个或多个字符。符号以一个**分界符**（delimiter）为结尾。分界符包括任何非打印ASCII字符（包括空格、制表符tab、回车和换行）、"(双引号）字符、（（左括号）字符、）（右括号）字符、；（分号）字符、&（即and）字符、|（竖线)字符、~字符和<（小于）字符。符号中不能包括分界符（<小于号除外，它可以是符号的第一个字符。同样，? 和 $? 字符串不能出现在一个符号的开头，因为它们被用来表示变量（见第7.19节）。另外，一个不完全遵循数字字段格式的字符序列被当作是一个符号。

下面是合法符号的例子：

```
Space-Station
February
fire
activate_sprinkler_system
notify-fire-department
shut-down-electrical-junction-387
!?#$^*
345B
346-95-6156
```

注意，下划线（即_）和连字符（即-）是怎样把符号联结成一个字段的。

CLIPS会保持标记（token）中的大小写字母。因为CLIPS区别大小写字母，所以它被称为**大小写敏感的**（case-sensitive）。例如，下面的符号，CLIPS认为是不同的：

```
case-sensitive
Case-Sensitive
CASE-SENSITIVE
```

下一种字段类型是**字符串**（string）。字符串必须以双引号开始和结束，双引号也是字段的一部分。双引号间可以有零个或多个任意字符，包括通常用作CLIPS分界符的那些字符。下面是合法的CLIPS字符串：

```
"Activate the sprinkler system."
"Shut down electrical junction 387."
"!?#$^"
"John Q. Public"
```

在CLIPS中，空格通常用作分界符，用来将字段（如符号）和其他标记隔开。标记间多余的空格将会被摒弃。然而，字符串里面的空格会被保留。例如，CLIPS会认为下面的字符串是4种不同的字符串：

```
"spaces"
"spaces "
" spaces"
" spaces "
```

如果去掉包围的双引号，则CLIPS会认为每一行包含相同的词，因为那些未用作分界符的空格会被忽略。

因为双引号用作字符串的分界符，所以，不能直接在一个字符串里输入双引号。例如，

```
""three-tokens""
```

会被CLIPS解释成以下3个不同的标记：

```
""
three-tokens
""
```

这是因为双引号起分界符的作用。

要在一个字符串里包含双引号，可以使用反斜杠 \ 。例如：

```
"\"single-token\""
```

在CLIPS里被解释为字符串

```
""single-token""
```

因为反斜杠阻止后续的双引号用作分界符，所以仅有一个字段产生。反斜杠本身也可以通过使用连续两个反斜杠而出现在字符串里。如：

```
"\\single-token\\"
```

在CLIPS里被解释为字符串字段

```
"\single-token\"
```

下一个字段**外部地址**（external address），对于探究CLIPS的基于规则的编程能力的重要性有限。外部地址代表由**用户自定义函数**（user-defined function）返回的外部数据结构的地址（用户自定义函数是一个用诸如C或Ada语言编写、与CLIPS连接以添加附加功能的函数）。由于外部地址的值不能由

形成标记的字符序列规定，且 CLIPS 基本原版本不包含有返回外部地址的函数，所以，在 CLIPS 的这种版本里不可能创建这种类型的字段。

剩下的 3 种字段是**事实地址、实例地址和实例名**。事实是 CLIPS 提供的复杂数据表示，我们将简要讨论。事实地址用来指向一个特定的事实。就像外部地址一样，事实地址不能由形成标记的字符序列规定。然而，规则可以获取事实地址作为模式匹配过程的一部分。我们将在第 7.21 节讨论其实现语法。

实例是 CLIPS 提供的其他复杂数据表示，将会在第 11 章详细讨论。实例可以通过实例地址或实例名来引用。一个实例地址就像事实地址一样，但是指向一个实例而不是事实。实例也可以通过名字引用。实例名是一种以左右方括号括起来的特定类型符号。例如，[pump-1] 是一个实例名。

零个或多个字段组合在一起被称为**多字段值**（multifield value）。复合字段值通常是通过调用一个函数（以后几章将会介绍）或规定字段的一系列值而被创建。当打印时，一个多字段值会被左右括号括住。例如，长度为零的多字段会打印成如下形式：

```
()
```

而包含符号 this 和 that 的多字段，会打印成如下形式：

```
(this that)
```

7.5 进入和退出 CLIPS

通过输入适当的运行命令到装有 CLIPS 的计算机上，即可进入 CLIPS。CLIPS 的提示符显示如下：

```
CLIPS>
```

至此，可以直接给 CLIPS 输入命令。这种模式叫**顶层**（top level）。

退出 CLIPS 的一般做法是输入 exit 命令。命令语法是：

```
(exit)
```

注意，exit 是被一对匹配的括号括住的。许多基于规则的语言起源于 LISP，而 LISP 使用括号作为分界符。既然 CLIPS 是基于最初使用 LISP 机而发展起来的语言，所以它保留了这些分界符。不带括号的 exit 与带括号的 exit 的意义是非常不同的。带括号的 exit 表明这 exit 是将被执行的命令，而不仅仅是个符号 exit。后面我们将会看到括号对于命令来说是作为重要的分界符起作用的。

现在要紧的是，记住每一条 CLIPS 命令必须有数目相等的左右括号，并且要互相匹配。

输入匹配的括号后，执行 CLIPS 命令的最后一步是按回车键。回车键也可以在任何标记输入之前或之后按。例如，在输入“ex”后、输入“it”之前按回车，会产生两个标记：一个是符号 ex 标记，另一个是符号 it 标记。

现以下面的命令序列作为样例，演示会话过程：进入 CLIPS、求一个固定字段值、求一个函数值、然后用 exit 命令退出。示出的例子是在使用 MS-DOS 的 IBM PC 机上、CLIPS 可执行文件存储在 A 驱动器的磁盘上、当前驱动器也为 A 的情况下演示的。CLIPS 可执行文件的名称假定为 CLIPS DOS。MS-DOS 或 CLIPS 显示的输出以正常字体显示。所有需由用户键入的输入以黑体显示。回车键为↵。记住，CLIPS 是大小写敏感的，所以，要完全按所显示的大小写键入。

```
A:\>CLIPSDOS↵
         CLIPS (V6.22 06/15/04)
CLIPS> exit↵
exit
CLIPS> (+ 3 4)↵
7
CLIPS> (exit)↵
A:\>
```

当处于顶层时，CLIPS 会接收来自用户的输入，并试图求值输入以作出适当的反应。输入字段本身会被认为是常量，对一个常量求值的结果是常量本身。因此，当键入符号 exit 和一个回车键时，CLIPS 求值这个输入，并显示符号 exit 作为结果。一个由括号括住的符号被看作一个命令或一个函数调用。因此，输入（+ 3 4）是对执行加法的 + 函数的调用。此函数调用的返回值是 7（返回值将在

第8章更详细地讨论)。最后，输入（exit）将调用 exit 命令，于是退出 CLIPS。函数（function）和命令（command）两词可以互换使用。不过在本书，函数指有返回值，而命令没有返回值或一般在顶层提示符下执行。

7.6 事实

为了解决一个问题，CLIPS 程序必须有数据或信息，以便推理。在 CLIPS 中，一个信息块（chunk）被称为**事实**（fact)。事实由**关系名**（relation name，一个符号字段)、后跟零个或多个**槽**（slot，也是符号字段）以及它们的相关值组成。以下是一个事实的例子：

```
(person (name "John Q. Public")
        (age 23)
        (eye-color blue)
        (hair-color black))
```

整个事实以及每个槽由一对左右括号限定。符号 person 是事实的关系名。此事实包括 4 个槽：name、age、eye-color 和 hair-color。槽 name 的值是“John. Q. Public”，槽 age 的值是 23，槽 eye-color 的值是 blue，槽 hair-color 的值是 black。注意，槽的顺序是无关紧要的。事实：

```
(person (hair-color black)
        (name "John Q. Public")
        (eye-color blue)
        (age 23))
```

与前一个 person 事实在 CLIPS 中是等同的。

自定义模板结构

事实被创建之前，必须告知 CLIPS 一个给定关系名的合法槽的列表。共享相同的关系名和包含共同的信息的几组事实可以利用**自定义模板**（deftemplate）结构来描述。通过加入程序员的知识到 CLIPS 环境中，**结构**（construct）形成了 CLIPS 程序的核心，它与函数和命令不同。自定义模板结构类似于 Pascal 语言中的记录结构。自定义模板结构的一般格式是：

```
(deftemplate <relation-name> [<optional-comment>]
   <slot-definition>*)
```

＜slot-definition＞的语法描述定义为：

```
(slot <slot-name>) | (multislot <slot-name>)
```

使用这种语法，事实 person 可描述成下面自定义模板：

```
(deftemplate person "An example deftemplate"
   (slot name)
   (slot age)
   (slot eye-color)
   (slot hair-color))
```

多字段槽（Multifield Slot）

一个事实的槽在它们相应的自定义模板中用关键词 slot（槽）指定后，就只能包含一个值（这些值被称为单字段槽)。但我们经常希望可以在一个给定的槽中输入 0 个或多个字段。这可以通过在它们相应的自定义模板中用关键词 multislot（多槽）指定事实的槽（这些值被称为多字段槽）来实现。例如，在自定义模板 person 中的槽 name，将此人的名字作为单个字符串值来存储。如果槽 name 用关键词 multislot 来定义，那么，它可以存储任意多个字段。因此，如果 name 是单字段槽，那么，事实：

```
(person (name John Q. Public)
        (age 23)
        (eye-color blue)
        (hair-color brown))
```

将是不合法的。但如果 name 是多字段槽，则它是合法的。一个自定义模板可由单字段槽和多字段槽的任意组合形成。

有序事实

有关系名且有一相应自定义模板的事实称为**自定义模板事实**（deftemplate fact）。有关系名而没有相应自定义模板的事实称为**有序事实**（ordered fact）。有序事实有一个隐含的多字段槽，以存储关系名下的所有值。事实上，每当 CLIPS 遇到一个有序事实时，它就会为该事实自动生成一个**隐式自定义模板**（implied deftemplate，与**显式自定义模板**（explicit deftemplate）相反，它使用自定义模板结构创建）。既然一个有序事实仅有一个槽，所以，定义一个事实时，可以不要槽名。例如，可用下面事实来表示一列数字：

```
(number-list 7 9 3 4 20)
```

本质上，它与定义下面的自定义模板是等同的：

```
(deftemplate number-list (multislot values))
```

然后，定义事实如下：

```
(number-list (values 7 9 3 4 20))
```

一般地，我们应该尽可能使用自定义模板事实，因为槽名使事实更具可读性和更易于操作。在两种情况下，有序事实是很有用的。第一，仅由一个关系名组成的事实用作标记是有用的，而且，不论自定义模板是否已定义，其外观是完全相同的。例如，有序事实：

```
(all-orders-processed)
```

可用作标记以标明所有 orders 何时已被处理。

第二，当事实仅包含一个槽，这个槽名通常是关系名的同义词。例如，事实：

```
(time 8:45)
(food-groups meat dairy bread
             fruits-and-vegetables)
```

与下列事实具有同样的含义：

```
(time (value 8:45))
(food-groups (values meat dairy bread
                     fruits-and-vegetables))
```

图 7.1 描述了本节介绍的术语间的关系。

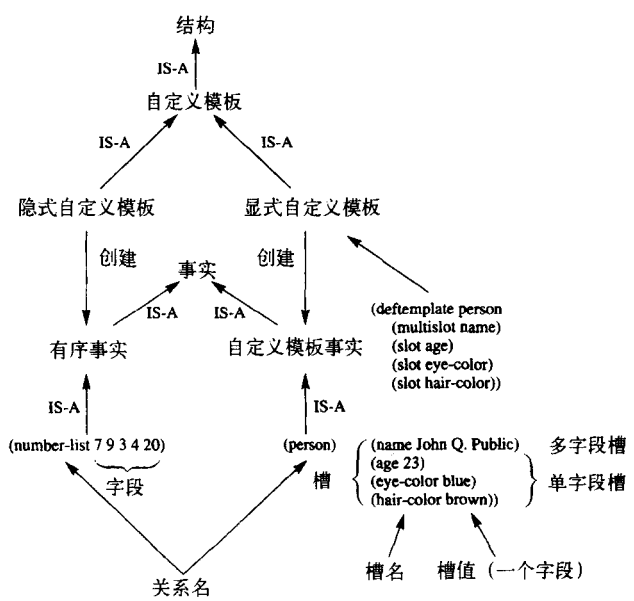


图 7.1 自定义模板总览

7.7 增加和删除事实

CLIPS把所有已知的事实组都存储在事实表中。我们可以从事实表中增加或删除表示信息的事实。可以用assert命令把新的事实增加到事实表中。assert命令的语法为：

```
(assert <fact>+)
```

作为一个例子，我们用person自定义模板来把一些人描述成事实。通过使用下面的命令，可以把关于John Q. Public的信息增加到事实表中：

```
CLIPS>
(deftemplate person
   (slot name)
   (slot age)
   (slot eye-color)
   (slot hair-color))↵
CLIPS>
(assert (person (name "John Q. Public")
                (age 23)
                (eye-color blue)
                (hair-color black)))↵
<Fact-0>
CLIPS>
```

注意，assert命令返回一个值：<Fact-0>。返回值的使用将会在第8章讨论。

命令facts可以用来显示事实表中的事实。facts命令的基本语法是：

```
(facts)
```

例如：

```
CLIPS> (facts)↵
f-0     (person (name "John Q. Public")
                (age 23)
                (eye-color blue)
                (hair-color black))
For a total of 1 fact.
CLIPS>
```

“f-0”是CLIPS指定给该事实的**事实标识符**（fact identifier）。每一个插入到事实表中的事实都被指定一个唯一的事实标识符。事实标识符以字母f开头，后接一个称为**事实索引**（fact index）的整数。事实索引0也在前一个assert命令的返回值<Fact-0>中显示。注意，正如本书所说的，在facts命令的输出中添加附加的空格，例如在槽（age 23）和槽（eye-color blue）之间，是为了提高可读性。通常，CLIPS仅在槽之间留一个空格，并且将输出按从最左端到最右端的顺序自动换行。在本书的其他例子中，只要增加的空格不会引起混乱，就会偶尔在CLIPS的输出中增加一些空格以提高可读性。

通常，CLIPS不接受一个事实的重复输入（尽管这种情况可以改变，正如第12.2节所示的）。因此，试图输入第二个具有相同槽值的John Q. Public事实到事实表是不会产生结果的。当然，其他不会与现有事实重复的事实可以很容易地增加到事实表中。例如：

```
CLIPS>
(assert (person (name "Jane Q. Public")
                (age 36)
                (eye-color green)
                (hair-color red)))↵
<Fact-1>
CLIPS> (facts)↵
f-0     (person (name "John Q. Public")
                (age 23)
                (eye-color blue)
                (hair-color black))
f-1     (person (name "Jane Q. Public")
                (age 36)
                (eye-color green)
                (hair-color red))
For a total of 2 facts.
CLIPS>
```

正如 assert 命令的语法所表明的，一个 assert 命令可以声明多个事实。例如，命令：

```
(assert (person (name "John Q. Public")
                (age 23)
                (eye-color blue)
                (hair-color black))
        (person (name "Jane Q. Public")
                (age 36)
                (eye-color green)
                (hair-color red)))
```

将增加两个事实到该事实表中。

认识到事实表的标识符不必连续是很重要的。正如我们可以把事实增加到事实表一样，同样也可以删除事实。当事实被删除后，被删除事实的标识符就会从表中消失。所以，当 CLIPS 程序执行时，事实标识符可能并不会严格按序排列。

因为一个事实表里可包含大量的事实，所以，能够查看事实表的一部分常常是有益的。要这样做，只需在使用 facts 命令时使用一些附加选项就行。facts 命令的完整语法是：

```
(facts [<start> [<end> [<maximum>]]])
```

这里，<start>、<end>和<maximum>都取正整数。注意 facts 命令允许选用 0～3 个参数。如果没有指定参数，则会显示所有事实。如果规定了<start>参数，则事实索引号≥<start>的所有事实会显示出来。如果同时规定<start>和<end>参数，则事实索引号≥<start>且≤<end>的所有事实会被显示。最后，如果<maximum>参数连同<start>和<end>参数一起被规定，则最多<maximum>个事实被显示。

就像事实可以被增加到事实表中一样，它们同样也可以被删除。从事实表中删除事实叫作**撤销**(retraction)，可以使用 retract 命令完成。retract 命令的语法是：

```
(retract <fact-index>+)
```

retract 命令的选项可以包括 1 个或多个待撤销事实的事实索引号。例如，用下面的命令可以删除事实表中的 John Q. Public 事实：

```
(retract 0)
```

同样，命令

```
(retract 1)
```

将撤销 Jane Q. Public 事实。

试图撤销一个不存在的事实将会产生以下错误信息（这里［PRNTUTIL1］是在《*CLIPS Reference Manual*》中查询错误信息的一个关键词）：

```
[PRNTUTIL1] Unable to find fact <fact-identifier>.
```

例如，

```
CLIPS> (retract 1)↵
CLIPS> (retract 1)↵
[PRNTUTIL1] Unable to find fact f-1.
CLIPS>
```

可以使用单个 retract 命令同时删除多个事实。例如，命令

```
(retract 0 1)
```

可撤销事实 f-0 和 f-1。

7.8 修改和复制事实

我们可用 modify 命令来修改自定义模板事实的槽值。modify 命令的语法为：

```
(modify <fact-index> <slot-modifier>+)
```

这里，<slot-modifier>为：

```
(<slot-name> <slot-value>)
```

例如，如果 John Q. Public 刚过完生日，则可用 modify 命令把他的年龄从 23 改为 24。

```
CLIPS> (modify 0 (age 24))↵
<Fact-2>
CLIPS> (facts)↵
f-2     (person (name "John Q. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
For a total of 1 fact.
CLIPS>
```

修改命令通过撤销原事实、然后用修改过的指定槽值声明一个新事实来完成操作。因为这一点，CLIPS会为一个被修改的事实产生一个新的事实索引。

duplicate命令的工作情况与此相同，只不过它并不撤销原事实。譬如，如果John（约翰）的长期失踪的孪生兄弟Jack（杰克）被找到了，则可以通过复制约翰的事实并改变name（姓名）槽把Jack添加到事实表中。

```
CLIPS> (duplicate 2 (name "Jack S. Public"))↵
<Fact-3>
CLIPS> (facts)↵
f-2     (person (name "John Q. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
f-3     (person (name "Jack S. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
For a total of 2 facts.
CLIPS>
```

modify（修改）命令和duplicate（复制）命令不能用于有序事实。

7.9 监视命令

watch命令对于调试程序是很有用的。监视事实的作用将在本节论述，剩下的监视项目会在本章和后续几章中讨论。这一命令的语法格式如下：

```
(watch <watch-item>)
```

其中，<watch-item>是facts（事实）、rules（规则）、activations（激活）、statistics（统计）、complications（编译）、focus（焦点）、deffunction、global、generic-function、method、instance、slot、message、message-handler等符号之一或all。

这些项目可以以任何组合被监视，以提供正确的调试信息。监视（watch）命令可多次使用来监视CLIPS执行的多个特征。“all”一词可用于选定全部监视特征。默认状态下，当CLIPS首次启动时，则编译受到监视，而剩下的监视项目则不被监视。

如果事实正受到监视，则CLIPS会自动打印一则消息，表明一旦事实被声明或被撤销，事实表就会完成一次更新。下面的命令说明了这一调试命令的用法：

```
CLIPS> (facts 3 3)↵
f-3     (person (name "Jack S. Public")
                (age 24)
                (eye-color blue)
                (hair-color black))
For a total of 1 fact.
CLIPS> (watch facts)↵
CLIPS> (modify 3 (age 25))↵
<== f-3       (person (name "Jack S. Public")
              (age 24)
              (eye-color blue)
              (hair-color black))
==> f-4       (person (name "Jack S. Public")
              (age 25)
              (eye-color blue)
              (hair-color black))
<Fact-4>
CLIPS>
```

字符序列 <== 表示该事实正在被撤销。字符序列 ==>> 表示事实正在被声明。

通过在 watch 命令后指定一个或多个自定义模板名可以监视指定的事实。例如，(watch facts person) 将显示 person 事实的信息。

使用对应的 unwatch 命令可以关闭 watch 命令的作用。unwatch 命令的语法格式是：

```
(unwatch <watch-item>)
```

7.10 自定义事实结构

能够自动声明一组事实，而不用在顶层键入相同的声明信息，这是很方便的。这特别适合于那些在程序运行前已知是正确的事实（即初始知识）的情况。运行测试实例去调试程序是另一种情况，在此情况下，能自动地声明一组事实也是很有用的。表示初始知识的一组事实可使用自定义事实(deffacts)结构来定义。例如，下面自定义事实语句提供了我们已经遇到过的一些人的初始信息：

```
(deffacts people "Some people we know"
   (person  (name "John Q. Public") (age 24)
            (eye-color blue) (hair-color black))
   (person  (name "Jack S. Public") (age 24)
            (eye-color blue) (hair-color black))
   (person  (name "Jane Q. Public") (age 36)
            (eye-color green) (hair-color red)))
```

自定义事实的一般格式是：

```
(deffacts <deffacts name> [<optional comment>]
   <facts>*)
```

自定义事实关键词后是本结构必需的自定义事实名。任何有效的符号都可用作自定义事实名。在本例中，选取的自定义结构名为 people。此名字后面的双引号中是可选的注释。与规则的注释一样，在 CLIPS 调入之后，该注释和自定义事实一起被一直保留。名字或注释后是事实，它将被此自定义事实语句声明到事实表中。

使用 CLIPS 的 reset 命令可以声明自定义事实语句中的事实。reset 命令会删除事实表中的所有事实，然后根据现有的自定义事实语句声明事实。reset 命令的语法是：

```
(reset)
```

假设自定义事实 people 已经输入（在自定义模板 person 之后输入），下面的对话展示了 reset 命令如何将事实添加到事实表中。

```
CLIPS> (unwatch facts)↵
CLIPS> (reset)↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (person  (name "John Q. Public")
                 (age 24)
                 (eye-color blue)
                 (hair-color black))
f-2     (person  (name "Jack S. Public")
                 (age 24)
                 (eye-color blue)
                 (hair-color black))
f-3     (person  (name "Jane Q. Public")
                 (age 36)
                 (eye-color green)
                 (hair-color red))
For a total of 4 facts.
CLIPS>
```

此输出显示了自定义事实语句中的事实，并显示了由 reset 命令产生的一个新事实，称之为**初始事实**（initial-fact）。一旦启动，CLIPS 自动定义以下两个结构：

```
(deftemplate initial-fact)

(deffacts initial-fact
   (initial-fact))
```

因此，即使你没有定义任何的自定义事实，reset 命令也会声明此事实（initial-fact）。initial-fact（初始事实）的事实标识符总是 f-0。initial-fact 的作用在于启动一个程序的执行（这将在下一节中论述）。

7.11 规则的组成

为了完成有用的工作，专家系统必须定出规则以及事实。由于事实的声明和撤销已论述过了，因此，现在可以论述规则的工作原理。

规则可以直接键入 CLIPS 或从编辑器创建的规则文件中调入（从文件中调入结构将在第 7.17 节中论述）。除了一些最小的程序外，你可能想要使用 CLIPS 提供的一个集成编辑器。有关编辑器的信息（对 Windows 2000/XP 和 MacOS 可执行文件）可在“接口指南（Interfaces Guide）”中找到，它以电子格式存于随书附送的光盘中。关于 EMACS 编辑器（它可用于如 Unix 等环境中）的信息在“基本编程指南（Basic Programming Guide）”论述。集成编辑器可以使你在程序开发过程中有选择地重新定义结构，这是非常有用的。例如，如果在顶层提示符下输入了一个结构，你有了该结构的版式，则你仍须重新键入整个结构。但如果你先在编辑器中输入该结构，那么，你在此编辑器中敲几个键就可以更正该版式或重定义该结构。开始时，示出的例子将是一些规则，它们在顶层直接输入到 CLIPS 中。

作为例子，让我们思考事实和规则的类型，它们可用于监视和响应一定范围内可能的紧急事件。这些紧急事件的例子可能是火灾或洪水。在一个工厂的监视专家系统中，一个可能规则的伪代码表示如下：

```
IF the emergency is a fire
THEN the response is to activate
     the sprinkler system
```

在将伪代码转换成规则之前，规则涉及的事实类型的自定义模板必须先进行定义。一个紧急事件（emergency）可由下面的自定义模板来表示：

```
(deftemplate emergency (slot type))
```

其中，emergency 事实的 type 字段可包括 fire（火灾）、flood（水灾）和 power outage（停电）之类的符号。类似地，其响应可用下面的自定义模板进行描述：

```
(deftemplate response (slot action))
```

其中，response 事实的 action 字段表示待采取的响应。

此规则用 CLIPS 的语法表示如下。此规则可在 CLIPS 提示符后键入，但你必须先输入 emergency 和 response 的自定义模板之后才能这样做。然而，在输入任何这类结构之前，在顶层提示符下键入命令（clear）并回车，这会清除前一节建立的自定义模板和自定义事实。clear（清除）命令将在第 7.13 节详细解释。

```
(defrule fire-emergency "An example rule"
   (emergency (type fire))
   =>
   (assert (response
             (action activate-sprinkler-system))))
```

如果此规则如上所示那样输入正确，则 CLIPS 提示符会重新出现。否则，出现错误信息，最可能的提示错误是关键词拼写出错或括号放错了位置。

下面是同一条规则，但它对规则的各个部分加入了相应的注释。注释以分号开始，到回车符处结束。注释会被 CLIPS 省略，这将在第 7.18 节讨论。

```
; Rule header
(defrule fire-emergency "An example rule"
   ; Patterns
   (emergency (type fire))
   ; THEN arrow
   =>
   ; Actions
   (assert (response
             (action activate-sprinkler-system))))
```

规则的一般格式是：

```
(defrule <rule name> [<comment>]
   <patterns>* ; Left-Hand Side (LHS) of the rule
   =>
   <actions>*) ; Right-Hand Side (RHS) of the rule
```

整条规则必须用括号括起，规则中的每一个模式（pattern）和行为（action）也必须用括号括起。一条规则可能有多个模式和行为。如果模式和行为被嵌套，则将它们括住的括号必须适当地配对。在 fire-emergency 规则中，只有一个模式和一个行为。

规则的开头包括 3 个部分。它必须以关键词 defrule 开头，接着是规则名。规则名可以是 CLIPS 中用的任何合法符号。如果输入的规则名与一个已存在的规则同名，那么，新的规则将会取代旧的那条规则。在本规则中，规则名是 fire-emergency。接下来是可选的注释字符串。对本规则，注释是“An example rule”（样例规则）。注释一般用于描述规则的目的或程序员想要的其他信息。规则名后的注释与以分号开头的注释不同，前者不会被忽略，它会与规则的其余部分一起被显示出来（用 ppdefrule 命令，该命令将在第 7.13 节中介绍）。

在规则头之后，是零个或多个条件元素（conditional element，CE）。最简单的条件元素是**模式条件元素**（pattern CE）或简称为**模式**（pattern）。每一个模式由一个或多个约束构成，其目的是匹配自定义模板事实中的字段。在 fire-emergency 规则中，模式是（emergency（type fire））。type 字段的约束表示这一规则只满足在 type 字段中包含符号 fire 的 emergency 事实。CLIPS 试图使规则的模式与事实表中的事实相匹配。如果规则的所有模式与事实匹配，则规则就被**激活**（activated）并放入**议程**（agenda）——已被激活的规则集合。在议程中可能没有也可能有多条规则。

规则中模式后的符号 =＞ 叫作**箭头**（arrow）。它由一个＝和一个＞构成。箭头是 IF-THEN 规则中 THEN 部分开始的标记。箭头前的规则部分称之为**左部**（LHS），之后的部分称之为**右部**（RHS）。

如果一条规则没有模式，那么，将添加一特殊模式（initial-fact）作为该规则的模式。既然 initial-fact 自定义事实是自动定义的，那么，自动声明 initial-fact 事实后一旦执行 reset 命令，那么，就会激活在规则的 LHS 侧没有模式的任何规则。因此，当执行 reset 命令后，任何无 LHS 模式的规则都会被置于议程之中。

规则的最后一部分是行为列表，当此规则**触发**（fire）时这些行为就会被执行。有些规则可能没有行为，这样做没有特定的作用，但允许这样做。在我们的例子中，行为是声明事实（response（action activate-sprinkler-system））。触发的意思是 CLIPS 执行议程中某条规则的行为。当议程中没有规则时，CLIPS 正常地停止执行。当议程中有多条规则时，CLIPS 会自动决定哪条是要触发的规则。CLIPS 按从低到高的优先级对议程中的规则排序，并触发优先级最高的那条规则。规则的优先级是一种整数属性，叫作**优先级**（salience）。优先级将在第 9 章作更详细的论述。

7.12 议程与执行

可以用 run 命令使 CLIPS 程序运行。run 命令的语法结构是：

```
(run [<limit>])
```

其中，可选的参数＜limit＞是要被触发的规则的最大数目。如果＜limit＞没输入或＜limit＞等于－1，则规则会被触发，直到议程中无规则剩下为止。否则，触发＜limit＞规定的规则个数后，规则的执行就会停止。

当运行 CLIPS 程序时，议程中优先级最高的规则会被触发。如果这时议程中只有一条规则，显然它将触发。由于 fire-emergency 规则的条件元素与事实（emergency（type fire））匹配，因此，当此程序运行时，fire-emergency 规则被触发。

每当规则的全部模式与事实相匹配，规则即变为激活状态。无论事实是否在规则定义之前或定义之后被声明，模式匹配过程总是一直进行并不断更新。

因为规则需要事实来执行，在 CLIPS 中 reset 命令是启动或重新启动专家系统的主要方法。一般地，用 reset 命令声明的事实与一条或多条规则的模式相匹配，并把这些规则的激活状态置于议程之中。发出 run 命令则开始执行该程序。

议程显示

使用 agenda 命令可以显示议程中的规则清单。agenda 命令的语法结构是：

```
(agenda)
```

如果议程中无激活存在，则在发出 agenda 命令后 CLIPS 提示符会重新出现。如果事实索引号为 1 的（emergency（type fire））事实激活了（fire-emergency）规则，则 agenda 命令会产生以下输出：

```
CLIPS> (reset)↵
CLIPS> (assert (emergency (type fire)))↵
<Fact-1>
CLIPS> (agenda)↵
0      fire-emergency: f-1
For a total of 1 activation.
CLIPS>
```

0 表示议程中规则的优先级，其后是规则名和与该规则的模式匹配的事实标识符。在本例中，只有一个事实标识符，为 $f-1$。

规则和反射

由于议程中有 fire-emergency 规则，因此，run 命令将使得该规则触发。（response（action activate-sprinkler-system））事实将作为规则的行为被加入到事实表中，像下面所输出的那样：

```
CLIPS> (run)↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (emergency (type fire))
f-2     (response
           (action activate-sprinkler-system))
For a total of 3 facts.
CLIPS>
```

在这里出现了一个有趣的问题：若再次发出 run 命令会产生什么呢？由于有一条规则，并有一个满足该规则的事实，因此，该规则应该再次触发。然而，若试图使用 run 命令，则此命令不会产生任何结果。检查该议程，会确认是因为该议程中已没有任何规则，所以没有规则被触发。

由于 CLIPS 的设计方式，因此该规则没有再次触发。CLIPS 中的规则具有一种叫作**反射**（refraction）的特性，它是指对于一个特定的事实集合，规则不会触发一次以上。如果没有反射，则专家系统会一直陷于无用的循环之中。也就是说，一旦规则被触发，它就会一次又一次地被同一事实不断触发。在现实世界中，引发触发的刺激作用最终会消失。例如：火灾最终会被洒水系统扑灭或自己熄灭。然而，在计算机世界中，一旦事实进入事实表，它就会一直保留，直到被明确删除为止。

如果有必要，可以通过撤销（emergency（type fire））事实并重新声明它来使此规则再次触发。基本上，CLIPS 会记住使规则触发的事实标识符。而且，若事实标识符的组合完全相同，则 CLIPS 不会再激活该规则。相同的几组事实标识符必须既在顺序上又在事实索引上相匹配。本章后面的例子将会显示一个事实如何以多种方法与模式匹配。在本例中，对一条规则来说，有相同事实标识符集合的几个激活可以置入此议程中，每一个对应着不同的匹配。

另外，可使用 refresh 命令使该规则再次触发。refresh 命令把对于某一规则已经触发的全部激活重新放回到该议程中（条件是，触发此激活的事实仍然还在事实表中）。refresh（更新）命令的语法结构是：

```
(refresh <rule-name>)
```

以下命令演示 refresh（更新）命令怎样重新激活 fire-emergency 规则：

```
CLIPS> (agenda)↵
CLIPS> (refresh fire-emergency)↵
CLIPS> (agenda)↵
0      fire-emergency: f-1
For a total of 1 activation.
CLIPS>
```

监视激活、规则和统计数据

如果激活正在被监视，则每当一个激活被加入到议程或从议程中清除时，CLIPS 就会自动打印一个消息。对事实而言，字符序列 <== 表示一个激活正在从议程中移走，而字符序列 ==> 表示一个激活正在被加入到议程中。在初始字符序列之后，打印出此激活，且有相同的由 agenda 命令使用的格式。下面的命令序列阐明了正在被监视的事实：

```
CLIPS> (reset)↵
CLIPS> (watch activations)↵
CLIPS> (assert (emergency (type fire)))↵
==> Activation 0       fire-emergency: f-1
<Fact-1>
CLIPS> (agenda)↵
0      fire-emergency: f-1
For a total of 1 activation.
CLIPS> (retract 1)↵
<== Activation 0       fire-emergency: f-1
CLIPS> (agenda)↵
CLIPS>
```

如果规则正处于被监视状态，则一旦触发某个规则，CLIPS 就会打印一则消息。以下命令序列说明了处于被监视状态下的激活和规则：

```
CLIPS> (reset)↵
CLIPS> (watch rules)↵
CLIPS> (assert (emergency (type fire)))↵
==> Activation 0       fire-emergency: f-1
<Fact-1>
CLIPS> (run)↵
FIRE    1 fire-emergency: f-1
CLIPS> (agenda)↵
CLIPS>
```

符号 FIRE 后的数字表示在发出 run 命令后已有多少条规则被触发。举个例子，如果另一条规则要在 fire-emergency 规则后触发，则此规则前会显示“FIRE 2”字样。在触发顺序打印出来之后，该规则名也被打印，规则名的后面是与该规则模式匹配的事实索引。注意，当触发一条规则（因此会从此议程中删除掉）时，监视激活不会显示消息。

通过在监视命令后指定一个或多个规则名，可以监视特定的规则和行动，例如，(watch activations fire-emergency) 将显示 fire-emergency 规则的行为信息。

如果统计数据处于被监视状态下，则 CLIPS 会在运行结束后打印出如下式样的信息性消息：

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch statistics)↵
CLIPS> (assert (emergency (type fire)))↵
<Fact-1>
CLIPS> (run)↵
1 rules fired    Run time is 0.02 seconds
50.0 rules per second
3 mean number of facts (3 maximum)
1 mean number of instances (1 maximum)
1 mean number of activations (1 maximum)
CLIPS> (unwatch statistics)↵
CLIPS>
```

若统计数据被监视，则在 run 命令后会显示 6 个统计量。被触发的规则总数、触发规则所需的时间（以秒为单位）和每秒平均触发规则的数目（第一个统计量除以第二个统计量）会显示出来。另外，

在每一次执行循环中，CLIPS都会保持有关事实数、激活数和实例数的统计数据。事实的平均数等于在每条规则触发后在事实表上的事实总数除以已触发的规则数。括号内 maximum 一词之前的数字表示对任一触发规则其事实表中包含的最大事实数。同样，激活的平均数和最大数两个统计数据分别表示每条规则触发的平均激活数和任一触发规则在议程中的最大激活数。实例的平均数和最大数表示的是与 COOL 有关的信息。

7.13 结构处理命令

显示特定结构的成员清单

list-defrules 命令用于显示由 CLIPS 维护的当前规则清单。类似地，list-deftemplates 命令和 list-deffacts 命令分别用于显示当前的自定义模板列表和当前的自定义事实表。这些命令的语法结构是：

```
(list-defrules)

(list-deftemplates)

(list-deffacts)
```

例如，

```
CLIPS> (list-defrules)↵
fire-emergency
For a total of 1 rule.
CLIPS> (list-deftemplates)↵
initial-fact
emergency
response
For a total of 3 deftemplates.
CLIPS> (list-deffacts)↵
initial-fact
For a total of 1 deffacts.
CLIPS>
```

显示指定结构成员的文本描述

ppdefrule（pretty print defrule，漂亮打印自定义规则）、ppdeftemplate（pretty print deftemplate，漂亮打印自定义模板）和 ppdeffacts（pretty print deffacts，漂亮打印自定义事实）命令分别用于显示自定义规则、自定义模板和自定义事实的文本描述。这些命令的语法结构如下：

```
(ppdefrule <defrule-name>)

(ppdeftemplate <deftemplate-name>)

(ppdeffacts <deffacts-name>)
```

每一个命令有一个参数，用于规定待显示的自定义规则、自定义模板和自定义事实的名称。当它们被显示时，CLIPS会在不同的行显示不同的结构部分以增加可读性。举例如下：

```
CLIPS> (ppdefrule fire-emergency)↵
(defrule MAIN::fire-emergency "An example rule"
   (emergency (type fire))
   =>
   (assert (response
             (action activate-sprinkler-system))))
CLIPS> (ppdeftemplate response)↵
(deftemplate MAIN::response
   (slot action))
CLIPS> (ppdeffacts initial-fact)↵
CLIPS> (deffacts start-fact (start-fact))↵
CLIPS> (ppdeffacts start-fact)↵
(deffacts MAIN::start-fact
   (start-fact))
CLIPS>
```

在每一个结构名之前的符号 MAIN:: 表示已将结构置于该模块（module）中。模块提供了分割知识库的方法，这将在第 9 章更详细地讨论。注意，initial-fact 自定义事实没有文本描述（因为它是由 CLIPS 自动生成的）。但是，输入的 start-fact 自定义事实则有文本描述。

删除指定的结构成员

undefrule、undeftemplate 和 undeffacts 命令分别用于删除自定义规则、自定义模板和自定义事实。这些命令的语法结构是：

```
(undefrule <defrule-name>)

(undeftemplate <deftemplate-name>)

(undeffacts <deffacts-name>)
```

每一条命令有一个参数，规定待删除的自定义规则、自定义模板和自定义事实的名称。例如：

```
CLIPS> (undeffacts start-fact)↵
CLIPS> (list-deffacts)↵
initial-fact
For a total of 1 deffacts.
CLIPS> (undefrule fire-emergency)↵
CLIPS> (list-defrules)↵
CLIPS>
```

注意，initial-facts 自定义事实和 initial-facts 自定义模板像其他用户自定义结构一样可以被删除。如果执行 reset 命令，则 initial-facts 事实不会被添加到此事实表中。

如果符号 * 作为任何结构删除命令的一个参数，则会删除所有相应类型的结构。例如：（undefrule *）会删除所有的自定义规则结构。符号 * 也可以与 retract（撤销）命令一起使用以删除所有事实。

被其他结构参照的结构，只有在参考它的结构被删除后才能被删除。如以下对话所示，在 initial-fact 自定义事实、（initial-fact）事实和 example 自定义规则被删除前，initial-fact 自定义模板是不能被删除的（使用默认的 initial-fact 模式）：

```
CLIPS> (defrule example =>)↵
CLIPS> (undeftemplate initial-fact)↵
[PRNTUTIL4] Unable to delete deftemplate
initial-fact.
CLIPS> (undeffacts initial-fact)↵
CLIPS> (undeftemplate initial-fact)↵
[PRNTUTIL4] Unable to delete deftemplate
initial-fact.
CLIPS> (undefrule example)↵
CLIPS> (undeftemplate initial-fact)↵
[PRNTUTIL4] Unable to delete deftemplate
initial-fact.
CLIPS> (retract *)↵
CLIPS> (undeftemplate initial-fact)↵
CLIPS>
```

清除 CLIPS 环境中的所有结构

clear 命令可用于删除 CLIPS 环境中的所有信息。此命令会删除当前包含在 CLIPS 中的所有结构和事实表中的所有事实。clear 命令的语法结构如下：

```
(clear)
```

在清除 CLIPS 环境之后，clear 命令会把 initial-facts 自定义事实添加到 CLIPS 环境中：

```
CLIPS> (list-deffacts)↵
CLIPS> (list-deftemplates)↵
emergency
response
start-fact
For a total of 3 deftemplates.
CLIPS> (clear)↵
```

```
CLIPS> (list-deffacts)↵
initial-fact
For a total of 1 deffacts.
CLIPS> (list-deftemplates)↵
initial-fact
For a total of 1 deftemplate.
CLIPS>
```

7.14 打印输出命令

除了在规则的RHS声明事实之外，RHS也可以通过使用printout命令打印输出信息。printout（打印输出）命令的格式是：

```
(printout <logical-name> <print-items>*)
```

其中，<logical-name>表示printout（打印输出）命令的输出目标，<print-items>* 是由该printout命令打印的零个或多个项目。

下面的规则演示了printout（打印输出）命令的用法：

```
(defrule fire-emergency
   (emergency (type fire))
   =>
   (printout t "Activate the sprinkler system"
               crlf))
```

非常重要的一点是要在printout命令后包括字母t，因为此参数t指明了输出的目标。这个目标也叫作**逻辑名**（logical name）。在本例中，逻辑名t告诉CLIPS把输出结果发送给计算机的**标准输出设备**（standard output device），标准输出设备通常是终端。但是，这种设备可以重新定义，因此，标准输出设备可以是其他东西，如modem（调制解调器）或打印机。逻辑名的概念将在第8.6节详细介绍。

逻辑名后的变量是由printout（打印输出）命令打印的项目。字符串

```
"Activate the sprinkler system"
```

将被除去双引号打印在终端上。crlf被 *printout* 命令特殊处理。它会强制回车/换行，这样，对在不同行上的输出进行格式化，可以改善输出显示。

7.15 使用复合规则

到现在为止，提到的都是只由一条规则组成的最简单的程序。然而，仅由一条规则组成的专家系统并没有多少应用价值。实际的专家系统可能由成百上千条规则组成。除了fire-emergency规则外，监视工厂的专家系统还包括对应于其他一些紧急情况的规则，如发大水。扩展的一组规则如下所示：

```
(defrule fire-emergency
   (emergency (type fire))
   =>
   (printout t "Activate the sprinkler system"
               crlf))

(defrule flood-emergency
   (emergency (type flood))
   =>
   (printout t "Shut down electrical equipment"
               crlf))
```

这些规则一旦输入到CLIPS中，声明（emergency (type fire)）事实、然后发出run命令，将会产生Activate the sprinkler system的输出结果。声明（emergency (type flood)）事实并发出run命令，将会产生Shut down the electrical equipment的输出结果。

在规则中捕获真实世界

如果只有火灾和水灾两种紧急情况需处理，那么，上述规则就足够了。然而，真实世界并非那么

简单。例如，火灾并非都能用水来扑灭。有些火灾需用化学灭火剂。如果火灾产生毒气或引发爆炸，该怎么办呢？除了现场消防员之外，还需要通知不在现场的消防部门吗？大楼的哪层发生大火影响重大？启动二楼的喷水装置可能会在一楼和二楼引起水患。两层楼的设备供电可能必须被切断。若大火发生在首层，则可能只需关掉首层的设备供电。如果大楼被水浸，则有没有可以关闭的防水门防止水的侵害呢？如果检测到工厂遭抢劫又该怎么办？如果所有这些情况都包括在规则中，那么，是否所有的可能性已经包括了呢？

很遗憾，答案是否定的。在真实世界中，事物并非总是运作得完美无缺。在专家系统中捕捉所有相关的知识是很困难的。最好的情况是，能够识别最主要的紧急情况，并提供规则让专家系统知道何时不去捕捉紧急情况。

多模式的规则

大多数真实世界的启发式应用都过于复杂，以至不能仅用一种简单模式来表达成规则。例如，对不同类型的火灾启动喷水装置，可能不仅是错误的，甚至还是危险的。涉及普通易燃物如纸、木、布的火灾（A类火灾）可以用水或水剂灭火器扑灭。涉及易燃液体、油脂和类似材料的火灾（B类火灾），必须使用不同的方法来扑灭，如二氧化碳灭火器。多模式的规则可用于表示这些情况。例如：

```
(deftemplate extinguisher-system
   (slot type)
   (slot status))

(defrule class-A-fire-emergency
   (emergency (type class-A-fire))
   (extinguisher-system (type water-sprinkler)
                        (status off))
   =>
   (printout t "Activate water sprinkler" crlf))

(defrule class-B-fire-emergency
   (emergency (type class-B-fire))
   (extinguisher-system (type carbon-dioxide)
                (status off))
   =>
   (printout t "Use carbon dioxide extinguisher"
               crlf))
```

两套规则都有两种模式。第一种模式确定发生了大火，并确定大火是A类还是B类。第二种模式确定是否已经打开合适的灭火装置。可添加更多的规则来关闭灭火装置（例如，喷水装置已启动，而发生的是B类火灾，则关闭喷水装置；或者，如果火灾已被扑灭，则关闭灭火装置）。还可用更多的规则来确定是否是一种特定的燃烧材料形成了A类或B类火灾。

规则中可包括任意多种模式。要认识到的重要一点是，只有所有模式与事实匹配，该规则才被置于议程中。这种限制叫作**与条件元素**（and conditional element）。因为所有规则的模式都隐含地包含在一个and条件元素中，因此，如果只有一个模式得到匹配，则该规则不会触发。在一条规则的LHS得到匹配，且该规则置于议程中之前，所有事实必须存在。

7.16 设置断点命令

CLIPS有一个调试命令，叫作set-break命令。该命令允许指定的一组规则中任何一个被触发之前，执行被暂停。一个在触发前被暂停执行的规则称为**断点**（breakpoint）。set-break命令的格式如下：

```
(set-break <rule-name>)
```

其中，<rule-name>（规则名）是被设置断点的规则名。例如，思考下面的规则（注意有序事实的使用，如第7.6节所述）：

```
(defrule first
   =>
   (assert (fire second)))

(defrule second
   (fire second)
   =>
   (assert (fire third)))

(defrule third
   (fire third)
   =>)
```

下列命令行显示没有设置断点时规则的执行情况：

```
CLIPS> (watch rules)↵
CLIPS> (reset)↵
CLIPS> (run)↵
FIRE    1 first: f-0
FIRE    2 second: f-1
FIRE    3 third: f-2
CLIPS>
```

当发出 run 命令后，所有 3 条规则都相继触发。下列命令行演示了使用 set-break 命令暂停执行的方法：

```
CLIPS> (set-break second)↵
CLIPS> (set-break third)↵
CLIPS> (reset)↵
CLIPS> (run)↵
FIRE    1 first: f-0
Breaking on rule second
CLIPS> (run)↵
FIRE    1 second: f-1
Breaking on rule third
CLIPS> (run)↵
FIRE    1 third: f-2
CLIPS>
```

在这个例子中，规则 2 和规则 3 被允许触发之前，执行暂停下来。注意，在断点停止执行前，run 命令必须触发至少一条规则。例如，规则 2（second）终止执行后，再输入 run 命令时，此规则不会再次终止执行。

show-breaks 命令用于列出所有断点。其格式是：

```
(show-breaks)
```

remove-break 命令用于删除断点。其格式是：

```
(remove-break [<rule-name>])
```

如果有＜rule-name＞参数，则只有该规则中的断点才被删除。否则，将删除所有断点。

7.17 调入和保存结构

从文件中调入结构

使用 load 命令可以把用文本编辑器产生的结构文件调入到 CLIPS 中。load 命令的格式如下：

```
(load <file-name>)
```

其中，＜file-name＞是包含待调入文件名的字符串或符号。

假设 emergency 规则和自定义模板保存在一台 IBM PC 机驱动器 B 上文件名为 fire.clp 的文件中，下列命令将把此结构调入到 CLIPS 中：

```
(load "B:fire.clp")
```

当然，文件名规范会因机器的不同而不同，这个例子应该仅作为指导。由于反斜杠字符在某些操

作系统中用作路径分隔符，因此，在调入时可能会出现问题。因为 CLIPS 把反斜杠解释为转义字符，所以，在字符串中要用两个反斜杠才能产生一个反斜杠字符。例如，普通的路径名可能写作：

```
B:\usr\clips\fire.clp
```

为保留反斜杠字符，路径名必须按如下命令形式写出：

```
(load "B:\\usr\\clips\\fire.clp")
```

结构并非都要保存在一个文件中。它们可以保存在多个文件中，并使用几个 load 命令调入。如果调入文件时未出现错误，则 load 命令会返回符号 TRUE（第 8 章将详细讨论返回值）。否则，返回符号 FALSE。

监视编译

当编译被监视时（这是默认情况），对于用 load 命令调入的每个结构都会打印出一则包含该结构名的消息。例如，假设 CLIPS 刚被启动，并输入下列命令：

```
CLIPS> (load "fire.clp")↵
Defining deftemplate: emergency
Defining deftemplate: response
Defining defrule: fire-emergency +j
TRUE
CLIPS>
```

这些消息表明，有两个自定义模板被调入（即 emergency 和 response），接着调入的是 fire-emergency 规则。跟在 Defining defrule 消息后的字符串 + j 是 CLIPS 关于被编译规则的内部结构的信息。此信息对于调整程序是很有用的，这将在论述效率的第 9 章中讨论。

如果编译不受监视，则 CLIPS 会为每个被调入的结构打印出一个字符：* 指自定义规则，% 指自定义模板，$ 指自定义事实。例如：

```
CLIPS> (clear)↵
CLIPS> (unwatch compilations)↵
CLIPS> (load fire.clp)↵
%%*
TRUE
CLIPS>
```

将结构保存到文件中

CLIPS 也提供了一个与 load 命令相对的命令。save 命令允许存于 CLIPS 中的一组结构被保存到磁盘文件中。save 命令的格式是：

```
(save <file-name>)
```

例如，以下命令将把 fire 结构保存到 B 盘上的 fire.clp 文件中：

```
(save "B:fire.clp")
```

Save 命令将 CLIPS 中的所有结构保存到指定的文件中。它不能把指定的结构保存到一个文件中。正常情况下，如果编辑器用来创建并修改结构，则不必使用 save 命令，因为你在使用编辑器时这些结构会被保存。然而，有时在 CLIPS 的提示符下直接输入结构然后将其保存在某个文件中，是很方便的。

7.18 注释结构

把一些注释添加到 CLIPS 程序中是一个好想法。有时，结构难以理解，此时注释可用来向读者解释这些结构将做些什么。注释还用于编写好的程序文档，这对于长程序是非常有用的。

CLIPS 中注释是任何以分号开头、以回车结尾的文本。下面是在 fire 程序中使用注释的例子：

```
;***********************************
;*                                 *
;* Programmer: G. D. Riley         *
;*                                 *
;* Title: The Fire Program         *
;*                                 *
;* Date: 05/17/04                  *
;*                                 *
;***********************************
; Deftemplates
 (deftemplate emergency "template #1"
  (slot type))       ; What type of emergency

 (deftemplate response  "template #2"
  (slot type))       ; How to respond
; The purpose of this rule is to activate
; the sprinkler system if there is a fire

 (defrule fire-emergency "An example rule" ; IF
  ; There is a fire emergency
  (emergency (type fire))
=>                                        ; THEN
  ; Activate the sprinkler system
 (assert (response
              (action activate-sprinkler-system))))
```

把这些结构调入 CLIPS，然后，应用漂亮打印法打印它们时会发现，每一个以分号开头的注释在 CLIPS 中都被删掉了。只有在结构名后被引号括起的注释才被保留下来。

7.19 变量

正如其他编程语言，CLIPS 有一些**变量**（variable）可用来保存变量值。CLIPS 中的变量在语法上总是由一个问号后接一个标识字段名组成。变量名遵循标识符的语法，但必须以一个字符开头。一种好的程序设计风格，其变量应该取有意义的名称。变量的例子如下：

```
?speed
?sensor
?value
?noun
?color
```

问号和标识字段名间没有空格。以后会讨论到，问号本身是有其作用的。在规则的 LHS 上可使用变量保存槽值，这些槽值可与规则的 LHS 上的其他值相比较，或者可以在规则的 RHS 上访问这些槽值。术语**被约束**（bound）和**约束**（bind）用来说明将某个值赋给某个变量。

变量通常用于约束规则 LHS 上的变量，然后，在规则的 RHS 上调用该值。例如：

```
CLIPS> (clear)↵
CLIPS>
(deftemplate person
   (slot name)
   (slot eyes)
   (slot hair))↵
CLIPS>
(defrule find-blue-eyes
   (person (name ?name) (eyes blue))
   =>
   (printout t ?name " has blue eyes." crlf))↵
CLIPS>
(deffacts people
   (person (name Jane)
           (eyes blue) (hair red))
   (person (name Joe)
           (eyes green) (hair brown))
   (person (name Jack)
           (eyes blue) (hair black))
   (person (name Jeff)
           (eyes green) (hair brown)))↵
CLIPS> (reset)↵
CLIPS> (run)↵
```

```
Jack has blue eyes.
Jane has blue eyes.
CLIPS>
```

Jane 和 Jack 都是蓝眼睛，所以规则 find-blue-eyes 激活了两次，描述 Jane 和 Jack 的事实各一次。规则被激活时，它便检测激活它的事实的 names 槽值，并把该值用于打印输出语句。

如果在规则的 RHS 上引用某变量，而在规则的 LHS 上没有约束它，则 CLIPS 将打印以下错误信息（假设未约束的变量为？x）：

```
[PRCCODE3] Undefined variable x referenced in
           RHS of defrule.
```

7.20 变量的复合用法

在规则 LHS 的多个地方使用的同名变量，有一个重要且有用的属性。一个变量首次约束于某一值时，该变量在此规则内便保持该值。其他同名的变量必须约束为这第一个变量的值。

除了可以编写一条规则只查找蓝眼睛的人之外，还可以声明一个事实，指定所要查找的特定的眼睛颜色。例如，

```
CLIPS> (undefrule *)↵
CLIPS> (deftemplate find (slot eyes))↵
CLIPS>
(defrule find-eyes
   (find (eyes ?eyes))
   (person (name ?name) (eyes ?eyes))
   =>
   (printout t ?name " has " ?eyes " eyes."
      crlf))↵
CLIPS>
```

自定义模板 find 说明了眼睛颜色，这是由 eyes 槽值规定的。规则 find-eyes 从 find 自定义模板的 eyes 槽值中获取该值，然后查找 eyes 槽值与？eyes 变量约束值相同的所有 person 事实。下面说明其工作原理：

```
CLIPS> (reset)↵
CLIPS> (assert (find (eyes blue)))↵
<Fact-5>
CLIPS> (run)↵
Jack has blue eyes.
Jane has blue eyes.
CLIPS> (assert (find (eyes green)))↵
<Fact-6>
CLIPS> (run)↵
Jeff has green eyes.
Joe has green eyes.
CLIPS> (assert (find (eyes purple)))↵
<Fact-7>
CLIPS> (run)
CLIPS>
```

注意，声明事实（find (eyes purple)）时，不会激活任何规则，因为在 eyes 槽值中没有值为 purple 的 person 事实。

7.21 事实地址

事实的撤销、修改和复制是很普遍的操作，一般在规则的 RHS 上完成，而不是在顶层完成。不过，我们也展示了在顶层提示下，利用事实指针对事实执行这些操作的过程。在规则的 RHS 上处理一个事实前，必须以某种方式规定与某特定模式匹配的事实。为此，使用**模式约束**（pattern binding）操作符“<-”在规则的 LHS 将某个变量约束到匹配某个模式的**事实地址**（fact address）上。该变量约束后，就可以在事实指针处和撤销、修改或复制命令一起使用。例如，以下说明如何在规则的 RHS 更新自定义模板槽值。

```
CLIPS> (clear)↵
CLIPS>
(deftemplate person
   (slot name)
   (slot address))↵
CLIPS>
(deftemplate moved
   (slot name)
   (slot address))↵
CLIPS>
(defrule process-moved-information
   ?f1 <- (moved (name ?name)
                 (address ?address))
   ?f2 <- (person (name ?name))
   =>
   (retract ?f1)
   (modify ?f2 (address ?address)))↵
CLIPS>
(deffacts example
   (person (name "John Hill")
           (address "25 Mulberry Lane"))
   (moved  (name "John Hill")
           (address "37 Cherry Lane")))↵
CLIPS> (reset)↵
CLIPS> (watch rules)↵
CLIPS> (watch facts)↵
CLIPS> (run)↵
FIRE    1 process-moved-information: f-2,f-1
<== f-2    (moved (name "John Hill")
                  (address "37 Cherry Lane"))
<== f-1    (person (name "John Hill")
                   (address "25 Mulberry Lane"))
==> f-3    (person (name "John Hill")
                   (address "37 Cherry Lane"))
CLIPS>
```

本例使用了两个自定义模板。自定义模板 person 用于存储关于某个人的信息，在本例中就是人的姓名和地址。也可存储其他信息，如年龄或眼睛颜色。自定义模板 moved 用于说明某人的地址已改变。其新地址在 address 槽中给出。

规则 process-moved-information 的第一个模式确定地址改变是否需要处理，第二个模式查找事实 person 中有待改变的地址信息。事实 moved 的事实地址约束于变量? f1 中，从而，处理完此地址变化后，即可撤销该事实。事实 person 的事实地址约束于变量? f2 中，之后，该变量可在规则的 RHS 用来修改 address 槽的槽值。

注意，可以使用事实中具有约束值的变量，也可以使用具有事实地址约束值的变量。另外，在变量? address 获取值的事实被撤销后，该变量的值还可用于 RHS 上。

在规则 process-moved-information 的 RHS 上撤销事实 moved 对规则的正确运行是非常重要的。注意，删去命令 retract 后，会出现什么情况：

```
CLIPS>
(defrule process-moved-information
   (moved (name ?name) (address ?address))
   ?f2 <- (person (name ?name))
   =>
   (modify ?f2 (address ?address)))↵
CLIPS> (unwatch facts)↵
CLIPS> (unwatch rules)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (watch rules)↵
CLIPS> (watch activations)↵
CLIPS> (run 3)↵
FIRE    1 process-moved-information: f-2,f-1
<== f-1    (person (name "John Hill")
                   (address "25 Mulberry Lane"))
==> f-3    (person (name "John Hill")
                   (address "37 Cherry Lane"))
==> Activation 0 process-moved-information: f-2,f-3
```

```
FIRE    2 process-moved-information: f-2,f-3
<== f-3    (person (name "John Hill")
                   (address "37 Cherry Lane"))
==> f-4    (person (name "John Hill")
                   (address "37 Cherry Lane"))
==> Activation 0  process-moved-information: f-2,f-4
FIRE  3 process-moved-information: f-2,f-4
<== f-4    (person (name "John Hill")
                   (address "37 Cherry Lane"))
==> f-5    (person (name "John Hill")
                   (address "37 Cherry Lane"))
==> Activation 0  process-moved-information: f-2,f-5
CLIPS>
```

由于在修改约束给? f2 的事实后声明了一个新的 person 事实，因此，程序进入死循环，从而重复激活 process-moved-information 规则。记住，一个 modify 命令可作为一个 retract 命令和一个 assert 命令对待。因为只要求 run 命令激活三条规则，因此，这就指明了程序运行结束的时间。如果 run 命令没有限制条件，程序将无限循环下去。在运行 CLIPS 的计算机上，终止这种循环的惟一办法是用 Control－C 中断 CLIPS，或者用另一个合适的中断命令。

7.22 单字段通配符

有时检测一个槽内字段的存在性但不实际赋值给一个变量是很有用的，尤其是对多字段槽。例如，我们想打印出姓某个姓的所有人的社会保险号。下面显示用来描述每一个人的自定义模板、某些预定义的人的自定义事实和打印姓某个姓的所有人的社会保险号的规则：

```
(deftemplate person
   (multislot name)
   (slot social-security-number))

(deffacts some-people
   (person (name John Q. Public)
           (social-security-number 483-98-9083))
   (person (name Jack R. Public)
           (social-security-number 483-98-9084)))

(defrule print-social-security-numbers
   (print-ss-numbers-for ?last-name)
   (person (name ?first-name ?middle-name
                 ?last-name)
         (social-security-number ?ss-number))
=>
(printout t ?ss-number crlf))
```

规则 print-social-security-mumbers 把此人的第一个名字和中间名分别赋值给变量? first-name 和? middle-name。但这些变量不能被规则 RHS 上的操作、LHS 上的其他模式或槽引用。当需要一个字段而不关心其值时，可以使用**单字段通配符**（single-field wildcard），而不必使用变量。单字段通配符用一个问号表示。利用单字段通配符后，规则 print-social-security-numbers 可以重写如下：

```
(defrule print-social-security-numbers
   (print-ss-numbers-for ?last-name)
   (person (name ? ? ?last-name)
           (social-security-number ?ss-number))
   =>
   (printout t ?ss-number crlf))
```

注意，person 事实的槽 name 必须包括待激活规则 print-social-security-numbers 的三个字段。例如，事实

```
(person (name Joe Public)
        (social-security-number 345-89-3039))
```

就不满足规则 print-social-security-numbers 的要求。

注意，问号和变量符号名之间没有空格。模式

```
(person (name ?first ?last))
```

要求 name 槽有两个字段，但模式

```
(person (name ?first ? last))
```

要求 name 槽有三个字段，最后一个字段一定是符号 last。

当模式中没有说明单字段槽时，CLIPS 会自动为该模式的槽添加一个单字段通配符。例如，模式

```
(person (name ?first ?last))
```

被转换成：

```
(person (name ?first ?last)
        (social-security-number ?))
```

7.23 块世界

为了演示变量的约束，我们将建立一个程序使块在一个简单的块世界里移动。这种程序和传统的块世界中的程序类似，程序中的知识范围只限于块（Firebaugh 88）。这是一个规划的范例，它可应用于通过机器手控制部件的自动制造中。

在块世界里惟一有趣的东西就是块。单一的块可以堆在另一个块上。一个复杂的块世界程序的目的是通过最少的移动次数使块达到所要求的目标结构。就这个例子来说，需要作一些简化限制。第一个限制是只允许有一个最初目标并且这个目标只能是把某个块移到另一个块上。在这个限制下，决定最优移动来达到目标并不很重要。如果目标是把块 x 移到块 y 上，就把块 x 上和块 y 上的所有块移到底层（floor）上，然后把块 x 移到块 y 上即可。

第二个限制是，任何目标还没有被达到。即是说，如果块 x 已经在块 y 的上面，则此目标就不能是把块 x 移动到块 y 上面。这是一个相当容易检测的情况。然而，这种情况的合适的检测语法要在第 8 章才介绍。

开始解决这个问题时，建立一个用来检测程序的块结构是很有用的。图 7.2 示出了将要用到的块结构。在这个结构中有两个堆栈。第一个堆栈是块 A 在块 B 上面，B 在块 C 上面。第二个堆栈是块 D 在块 E 上面，块 E 在块 F 上面。

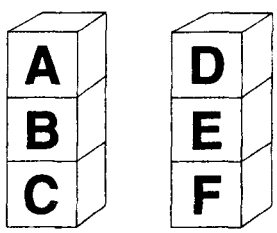


图 7.2 块世界的初始配置

为了决定究竟哪种规则解决这个问题更有效，就要按部就班地完成一个块世界目标。把块 C 移到块 E 上面要采取什么步骤？最简单的解决办法是直接把块 C 移到块 E 上面。但这个规则只有在块 C 和块 E 上都没有块时才能用。这个规则的伪码是：

```
RULE MOVE-DIRECTLY
IF   The goal is to move block ?upper on top of
       block ?lower and
     block ?upper is the top block in its stack and
     block ?lower is the top block in its stack,
THEN Move block ?upper on top of block ?lower.
```

规则 move-directly 在这个例子中不能用，因为块 A 和块 B 在块 C 上，块 D 在块 E 上。为了允许规则 move-directly 把块 C 移到块 E 上，块 A、块 B 和块 D 必须被移到底层。因为这是使它们不妨碍移动的最简单的办法。简单块世界不要求块重新堆起来，只需达到一个简单的初始目标，所以不必把移开的块重新堆上。这个规则可以用两个伪码程序表示：一个规则是将待移动的块上面的块移走，另一个规则是将被堆砌的块上面的块移走。

```
RULE CLEAR-UPPER-BLOCK
IF   The goal is to move block ?x and
     block ?x is not the top block in its stack and
     block ?above is on top of block ?x,
THEN The goal is to move block ?above to the floor

RULE CLEAR-LOWER-BLOCK
IF   The goal is to move another block on top of
```

```
      block ?x and
    block ?x is not the top block in its stack and
    block ?above is on top of block ?x,
THEN The goal is to move block ?above to the floor
```

规则 clear-upper-block 将块 C 上的块移走。它首先决定块 B 需要移到底部。为了把块 B 移到底层，同样的规则会决定块 A 需要移到底部。类似地，规则 clear-lower-block 会决定块 D 要移到底层以便把其他块移到块 E 上。

现在有了一些子目标就是把块 A、块 B 和块 D 移到底层。块 A 和块 D 可以直接移到底层。如果写得适当，规则 move-directly 可以将块移到底层，或移到其他块的上面。因为底层实际上并不是块，所以对底层的处理方法可以是不同的。下面的伪码规则是把块移到底层的特例：

```
RULE MOVE-TO-FLOOR
IF   The goal is to move block ?upper on top of
       the floor and
     block ?upper is the top block in its stack,
THEN Move block ?upper on top of the floor.
```

规则 move-to-floor 现在可以把块 A 和块 D 移到底层了。一旦块 A 移到底层，规则 move-to-floor 就可以把块 B 移到底层了。由于块 A、块 B 和块 D 在底层，所以，块 C 和块 E 现在就在堆栈的顶层，就可以用规则 move-directly 把块 C 移到块 E 上。

现在这个规则已经用伪码写出来了，规则用到的事实应该是确定的。由于没有原型，需要的事实的类型不能完全确定。在这个例子中，伪码规则指出了几种将会用到的事实类型。举例来说，哪些块在其他块上这一信息就是很重要的。这一信息可以用以下自定义模板来描述：

```
(deftemplate on-top-of
   (slot upper)
   (slot lower))
```

而且，这一模板描述的事实是：

```
(on-top-of (upper A) (lower B))
(on-top-of (upper B) (lower C))
(on-top-of (upper D) (lower E))
(on-top-of (upper E) (lower F))
```

因为知道哪些块在堆栈的顶部和底部也是很重要的，所以，包括以下事实也很有用：

```
(on-top-of (upper nothing) (lower A))
(on-top-of (upper C) (lower floor))
(on-top-of (upper nothing) (lower D))
(on-top-of (upper F) (lower floor))
```

单词 nothing 和 floor 在这些事实中有特殊含义。事实 (on-top-of (upper nothing) (lower A)) 和 (on-top-of (upper nothing) (lower D)) 表示块 A 和块 D 在堆栈顶。类似地，事实 (on-top-of (upper C) (lower floor)) 和 (on-top-of (upper F) (lower floor)) 表示块 C 和块 F 在堆栈底部。包含这些事实并不一定能确定堆栈的栈顶和栈底是哪些块。如果规则写错了，则单词 nothing 和 floor 可能会被误认为是块的名字。指出块的名字的事实可能也是有作用的。下面的事实用了隐式模板 block，它们可以用来辨别块和特殊单词 nothing 和 floor。

```
(block A)
(block B)
(block C)
(block D)
(block E)
(block F)
```

最后，需要一个事实来描述正在被处理的移动块的目标。这些目标可以用自定义模板来描述：

```
(deftemplate goal (slot move) (slot on-top-of))
```

使用这个模板的初始目标是：

```
(goal (move C) (on-top-of E))
```

事实和模板定义后，初始块结构可以用以下自定义事实来描述：

```
(deffacts initial-state
   (block A)
   (block B)
   (block C)
   (block D)
   (block E)
   (block F)
   (on-top-of (upper nothing) (lower A))
   (on-top-of (upper A) (lower B))
   (on-top-of (upper B) (lower C))
   (on-top-of (upper C) (lower floor))
   (on-top-of (upper nothing) (lower D))
   (on-top-of (upper D) (lower E))
   (on-top-of (upper E) (lower F))
   (on-top-of (upper F) (lower floor))
   (goal (move C) (on-top-of E)))
```

规则 move-directly 可写作：

```
(defrule move-directly
   ?goal <- (goal (move ?block1)
                  (on-top-of ?block2))
   (block ?block1)
   (block ?block2)
   (on-top-of (upper nothing) (lower ?block1))
   ?stack-1 <- (on-top-of (upper ?block1)
                          (lower ?block3))
   ?stack-2 <- (on-top-of (upper nothing)
                          (lower ?block2))
   =>
   (retract ?goal ?stack-1 ?stack-2)
   (assert (on-top-of (upper ?block1)
                      (lower ?block2))
           (on-top-of (upper nothing)
                      (lower ?block3)))
   (printout t ?block1 " moved on top of " ?block2
             "." crlf))
```

前3个模式确定的目标是把一个块移到另一个的顶上。其中，模式2和模式3保证本规则不会将块移到底层。第4和第6个模式检查这些块是否处于栈的顶部。第5和第6个模式匹配所需的信息，更新堆栈，将移动块从堆栈移走，或移入堆栈。这个规则的操作是更新两个堆栈的栈信息，并打印一则消息。在已移走的块下面的块，至此已在栈的顶部，而那个被移走的块也已在另一个栈的顶部。

规则 move-to-floor 可实现如下：

```
(defrule move-to-floor
   ?goal <- (goal (move ?block1)
   (on-top-of floor))
   (block ?block1)
   (on-top-of (upper nothing) (lower ?block1))
   ?stack <- (on-top-of (upper ?block1)
                        (lower ?block2))
   =>
   (retract ?goal ?stack)
   (assert (on-top-of (upper ?block1)
                      (lower floor))
           (on-top-of (upper nothing)
                      (lower ?block2)))
   (printout t ?block1 " moved on top of floor."
             crlf))
```

本规则和 move-directly 规则类似，但不必更新底层的一些信息，因为它不属于块。

规则 clear-upper-block 实现如下：

```
(defrule clear-upper-block
   (goal (move ?block1))
   (block ?block1)
   (on-top-of (upper ?block2) (lower ?block1))
```

```
   (block ?block2)
   =>
   (assert (goal (move ?block2)
                 (on-top-of floor))))
```

规则 clear-lower-block 实现如下：

```
(defrule clear-lower-block
   (goal (on-top-of ?block1))
   (block ?block1)
   (on-top-of (upper ?block2) (lower ?block1))
   (block ?block2)
   =>
   (assert (goal (move ?block2)
                 (on-top-of floor))))
```

有了规则 move-directly、move-to-floor、clear-upper-block 和 clear-lower-block 和自定义模板 goal、on-top-of 以及自定义事实 initial-state，这个程序至此就完整了。以下输出结果展示了运行此块世界程序的一个范例。

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (run)↵
A moved on top of floor.
B moved on top of floor.
D moved on top of floor.
C moved on top of E.
CLIPS>
```

首先把块 A 和块 B 移到底层，以清空块 C 的顶部；然后，把块 D 移到底层，以清空块 E 的顶部。最后，便可以把块 C 移到块 D 的顶部，从而完成了初始目标。

这个范例说明了如何使用渐进的方法来建立一个程序。首先，把伪码规则用类似于英语的文本写出来。其次，用伪码规则来决定所需事实的类型。设计出描述这些事实的自定义模板，并用这些模板把初始知识编成程序代码。最后，以自定义模板作为翻译指南，将伪码规则转换成 CLIPS 规则。

一个专家系统的开发通常要比本例有更多的原型构造和反复设计。也并非总是可以决定出最好的方法来表示建立一个专家系统所需的事实和规则类型。然而，当需要进行大量的原型构造和反复过程时，遵循一个一致的方法对于开发专家系统是有帮助的。

7.24 多字段通配符和变量

多字段通配符

多字段通配符和变量可用来匹配模式中的 0 个或多个字段。多字段通配符用一个美元符号后跟一个问号，即符号“$?”来表示，可表示出现 0 个或多个字段。注意，普通的变量和通配符只精确匹配一个字段。这是一个小区别，但很有意义。为了说明多字段通配符的使用，让我们回顾在第 7.22 节中描述的 print-social-security-numbers 规则：

```
(defrule print-social-security-numbers
   (print-ss-numbers-for ?last-name)
   (person (name ? ? ?last-name)
           (social-security-number ?ss-number))
   =>
   (printout t ?ss-number crlf))
```

这条规则只与恰好有 3 个字段的姓名槽匹配。于是事实

```
(person (name Joe Public)
        (social-security-number 345-89-3039))
```

将无法与规则匹配。然而，如果把两个单字段的通配符换成单个多字段通配符（如下所示），则模式 person 就能与至少包含一个字段的任何姓名槽匹配，并把其最后的字段作为指定姓名。

```
(defrule print-social-security-numbers
   (print-ss-numbers-for ?last-name)
   (person (name $? ?last-name)
           (social-security-number ?ss-number))
   =>
   (printout t ?ss-number crlf))
```

类似地，当一个模式中的多字段槽没有指定时，CLIPS会自动为该模式的槽增加一个多字段通配符检查。例如，模式：

```
(person (social-security-number ?ss-number))
```

被转换成：

```
(person (name $?)
        (social-security-number ?ss-number))
```

多字段变量

正如单字段变量前加“?”，多字段变量前则加“$?”。下列指令说明如何打印一特定人的所有子女的姓名：

```
(deftemplate person
   (multislot name)
   (multislot children))

(deffacts some-people
   (person (name John Q. Public)
           (children Jane Paul Mary))
   (person (name Jack R. Public)
           (children Rick)))

(defrule print-children
   (print-children $?name)
   (person (name $?name)
           (children $?children))
   =>
   (printout t ?name " has children " ?children
               crlf))
```

规则 print-children 的第一个模式把将要打印出其子女的那个人的姓名约束给变量 $? name。第二个模式把事实 person 与包含在变量 $? name 中的指定姓名相匹配，然后，将此人的子女列表约束给变量 $? childern。该值随后在规则的 RHS 中打印出来。

注意，在规则的 RHS 中引用一个多字段变量时，无须在变量名中包含符号 $。符号 $ 只用于 LHS 中来表明能约束给该变量的零个或多个字段。

以下对话显示了 print-children 规则如何运行：

```
CLIPS> (reset)↵
CLIPS> (assert (print-children John Q. Public))↵
<Fact-3>
CLIPS> (run)↵
(John Q. Public) has children (Jane Paul Mary)
CLIPS>
```

注意，约束给变量? name 和? children 的多字段值在打印时，可用括号括起。

在单个槽中可以使用多个多字段变量。例如，假定我们想找出所有这些人，他们有一个孩子，孩子有特定的姓名。以下规则可完成这一任务：

```
(defrule find-child
   (find-child ?child)
   (person (name $?name)
           (children $?before ?child $?after))
   =>
   (printout t ?name " has child " ?child crlf)
   (printout t "Other children are "
               ?before " " ?after crlf))
```

总的来说，如果我们只对变量? child 的值感兴趣，那么就可用多字段通配符来替代变量? before 和? after（同时删去与这两个变量相关联的打印输出语句）。以下对话显示了 find-child 规则如何运行：

```
CLIPS> (reset)↵
CLIPS> (assert (find-child Paul))↵
<Fact-3>
CLIPS> (run)↵
(John Q. Public) has child Paul
Other children are (Jane) (Mary)
CLIPS> (assert (find-child Rick))
<Fact-4>
CLIPS> (run)↵
(Jack R. Public) has child Rick
Other children are () ()
CLIPS> (assert (find-child Bill))↵
<Fact-5>
CLIPS> (run)↵
CLIPS>
```

当 Paul 约束给变量? child 时，变量? before 和? after 分别约束到（Jane）和（Mary）上。同样，当 Rick 约束给变量? child 时，变量? before 和? after 均约束到（ ）上，多字段当然也包括零字段。

以多种方式进行模式匹配

到目前为止，我们已处理了单一事实以单一方式与模式匹配的情形。通过使用多字段通配符，可以多种方式进行模式匹配。假定输入这样一个事实，某人以他本人的姓名为他所有的子女命名，参看下列：

```
CLIPS> (reset)↵
CLIPS> (assert (person (name Joe Fiveman)
                       (children Joe Joe Joe)))↵
<Fact-3>
CLIPS> (assert (find-child Joe))↵
<Fact-4>
CLIPS> (agenda)↵
0      find-child: f-4,f-3
0      find-child: f-4,f-3
0      find-child: f-4,f-3
For a total of 3 activations.
CLIPS> (run)↵
(Joe Fiveman) has child Joe
Other children are () (Joe Joe)
(Joe Fiveman) has child Joe
Other children are (Joe) (Joe)
(Joe Fiveman) has child Joe
Other children are (Joe Joe) ()
CLIPS>
```

正如本规则激活情况所显示的，有 3 种不同的方式使变量? child、? before 和? after 约束到事实f-3 上。第一种情形，变量? before 约束到 (),? child 约束到 Joe，? after 约束到（Joe Joe）。第二种情形,? before 约束到（Joe),? child 约束到 Joe,? after 约束到（Joe）上。第三种情形下,? before 约束到 (Joe Joe),? child 约束到 Joe,? after 约束到（）上。

堆栈的实现

堆栈（stack）是一种有序事实结构，其元素能被添加和删除，这些操作在“栈顶”进行。新的元素可以压入堆栈（添加）或最后添加的元素从栈顶弹出（删除）。在堆栈中，最早添加的元素最后被移走，最后添加的元素最早被移走。

堆栈的一个很好的类比是餐厅的盘子，新盘加在盘叠的顶端（压入），最后加到顶端的盘最先被移走（弹出）。

使用多字段变量能相对容易地实现堆栈的压入和弹出操作。首先，使用包含一组元素的有序事实 stack。以下规则是一个值压入事实 stack 中的操作：

```
(defrule push-value
   ?push-value <- (push-value ?value)
   ?stack <- (stack $?rest)
   =>
   (retract ?push-value ?stack)
   (assert (stack ?value $?rest))
   (printout t "Pushing value " ?value crlf))
```

实现出栈操作需两条规则：一条用于空栈，另一条用于非空栈。

```
(defrule pop-value-valid
   ?pop-value <- (pop-value)
   ?stack <- (stack ?value $?rest)
   =>
   (retract ?pop-value ?stack)
   (assert (stack $?rest))
   (printout t "Popping value " ?value crlf))

(defrule pop-value-invalid
   ?pop-value <- (pop-value)
   (stack)
   =>
   (retract ?pop-value)
   (printout t "Popping from empty stack" crlf))
```

很容易将这些规则转换成对已命名的堆栈进行的压入和弹出操作。例如，下列模式：

```
?push-value <- (push-value ?value)
?stack <- (stack $?rest)
```

可替换为：

```
?push-value <- (push-value ?name ?value)
?stack <- (stack ?name $?rest)
```

此处，? name 代表栈名。

块世界问题的重新探讨

使用多字段通配符和变量，可使块世界问题的解决方式大大简化。每个栈可由一个单一事实表示，如下所示。移动块的操作类似于入栈/出栈操作。

```
(deffacts initial-state
   (stack A B C)
   (stack D E F)
   (goal (move C) (on-top-of E))
   (stack))
```

设置空栈事实 stack 是为了防止以后可能出现的情况。比如，堆栈中只有一个块时，把它移到另一块的顶部以后将成为空的堆栈。

使用多字段变量的块世界程序规则如下：

```
(defrule move-directly
   ?goal <- (goal (move ?block1)
                  (on-top-of ?block2))
   ?stack-1 <- (stack ?block1 $?rest1)
   ?stack-2 <- (stack ?block2 $?rest2)
   =>
   (retract ?goal ?stack-1 ?stack-2)
   (assert (stack $?rest1))
   (assert (stack ?block1 ?block2 $?rest2))
   (printout t ?block1 " moved on top of "
             ?block2 "." crlf))

(defrule move-to-floor
   ?goal <- (goal (move ?block1) (on-top-of floor))
   ?stack-1 <- (stack ?block1 $?rest)
   =>
```

```
   (retract ?goal ?stack-1)
   (assert (stack ?block1))
   (assert (stack $?rest))
   (printout t ?block1 " moved on top of floor."
               crlf))

(defrule clear-upper-block
   (goal (move ?block1))
   (stack ?top $? ?block1 $?)
   =>
   (assert (goal (move ?top) (on-top-of floor))))

(defrule clear-lower-block
   (goal (on-top-of ?block1))
   (stack ?top $? ?block1 $?)
   =>
   (assert (goal (move ?top) (on-top-of floor))))
```

7.25 小结

本章介绍了CLIPS的基本组成。事实是CLIPS系统中的第一个组成部分，它由字段组成。字段可以是符号、字符串、整数或者是浮点数。事实的第一个字段一般用于显示存储在该事实中的信息类型，称为关系名（relation name）。自定义模板结构用来将槽名指定给事实中特定字段，这些字段以一个指定的关系名开头。自定义事实结构用来将事实指定为初始知识。

规则是CLIPS系统的第二个组成部分。一条规则可分为LHS和RHS两部分。规则中的LHS可被看作是IF部分，RHS可被看作是THEN部分。规则有多个模式和行为。

CLIPS的第三个组成部分是推理机。其模式与事实相匹配的规则会产生一个激活，该激活被置于议程中。反射会使规则不会频繁地被旧事实激活。

本章还介绍了变量的概念。变量用来从事实获取信息，并在进行规则LHS模式匹配时约束槽值。变量可存储规则LHS中模式的事实地址，使与模式相联系的事实在规则的RHS上可以被撤销。如果匹配的字段可以任意，且它的值在规则的LHS或RHS上不需要时，则单字段通配符可以代替变量。多字段变量和通配符允许匹配模式中的多个字段。

习题

7.1 用自定义事实语句把下列句子转换成事实。对于每组相关的事实，定义一个自定义模板，以描述一种更一般的关系。

The father of John is Tom.（约翰的父亲是汤姆）
The mother of John is Susan.（约翰的母亲是苏姗）
The parents of John are Tom and Susan.（约翰的父母是汤姆和苏姗）
Tom is a father.（汤姆是父亲）
Susan is a mother.（苏姗是母亲）
John is a son.（约翰是儿子）
Tom is a male.（汤姆是男的）
Susan is a female.（苏姗是女的）
John is a male.（约翰是男的）

7.2 为包含一个集合信息的事实定义一个自定义模板。包括该集合的名称或描述信息、集合中的元素列表和它是否为另一集合的子集。用你的自定义模板格式把下列集合表示为事实：

```
A = { 1, 2, 3 }
B = { 1, 2, 3, red, green }
C = { red, green, yellow, blue }
```

7.3 一个稀疏数组包含相对少的非零元素。把它表示成链接表或树会更有效。一个稀疏数组怎样用事

实来表示？描述表示该数组的事实所用的自定义模板。与在过程语言中使用数组数据结构相比，用事实表示数组的潜在缺点是什么？

7.4 把图2.4（a）表示航线的一般网络转换成一系列以自定义事实语句表述的事实。用一个自定义模板描述这些事实。

7.5 把图2.4（b）表示一个家庭的语义网转变成一系列以自定义事实语句表述的事实。用几个自定义模板描述所产生的事实。

7.6 把图2.5中的语义网转换成一系列以自定义事实语句表述的事实。用几个自定义模板描述这些事实。例如，IS-A和AKO连接是关系名，每一个连接应有自己的自定义模板。

7.7 把图3.3中表示动物分类信息的二叉判定树转换成一系列用自定义事实语句表述的事实。说明怎样才能表示出结点间的连接。叶子与树的其他结点需要不同的表示吗？

7.8 利用AKO和IS-A自定义模板，将习题2.1中的语义网转换成自定义事实语句。

7.9 植物正常生长需要多种不同类型的营养素。化肥提供的3种最重要的营养素是氮、磷和钾。这3种元素中缺少任何一种都会引起各种不同的症状。把下面的直观推断转换成规则，以决定是否缺乏某种营养素。假设此植物正常时是绿色的。

发育不良的植物可能缺氮。
暗黄色的植物可能缺氮。
叶边红褐色的植物可能缺磷。
根发育不良的植物可能缺磷。
茎干细长的植物可能缺磷。
紫色的植物可能缺磷。
成熟期推迟的植物可能缺磷。
叶边枯萎的植物可能缺钾。
茎干脆弱的植物可能缺钾。
种子或果实枯萎的植物可能缺钾。

用自定义模板描述规则中使用的事实。程序的输入应该通过把症状声明为事实来形成。在终端打印的输出应该指出缺乏哪种营养素。设计一种方法，避免缺乏一种营养素的多个症状产生的多个打印输出。用以下输入来测试你的程序。

植物的根发育不良。
植物颜色为紫色。

7.10 根据主要的燃烧材料，可把火灾归类。把以下信息转换成规则，以决定火灾的类型：

A类火灾包括如纸、木和布等普通易燃物。
B类火灾包括易燃液体(如石油和石油气)、油脂和类似的物质。
C类火灾包括使用电力的电器。
D类火灾包括易燃的金属，如镁、钠和钾。

用来灭火的灭火器类型取决于火的类型。把下面的信息转换成规则：

A类火灾应该使用吸热或阻燃型灭火器灭火，如水或水类液体和无水化学品。
B类火灾应该通过隔绝空气、抑制易燃蒸汽的释放或终止易燃物的连锁反应来灭火。灭火器包括无水化学品、二氧化碳、泡沫和含溴三氟甲烷。
C类火灾应该使用防短路的非导电媒质灭火。如果可能的话，应切断电源。灭火器包括无水化学品、二氧化碳和含溴三氟甲烷。
D类火灾应该使用焖熄法和不与燃烧金属发生反应的吸热化学品灭火。这些化学品包括：三甲氧硼化物和涂有石墨的焦碳。

描述规则中所用的事实。程序应该通过声明燃烧材料的类型作为事实来得到输入。输出结果应

该显示可以使用何种灭火器以及应该采取的其他措施，如切断电源。演示你的程序，对于一种材料的每种火灾类型均能正常处理。

7.11 低纬度云是那些高度等于或低于6000英尺高的云，包括层云和层积云。中纬度云是那些高度在6000～20000英尺的云，包括高层云、高积云和乱层云。高纬度云是那些高于20000英尺的云，包括卷云、卷层云和卷积云。积云和积雨云可以从低到高纬度之间分布。层云、高层云、卷层云、积云和积雨云呈大圆堆状。层云、高层云、乱层云和卷层云像平滑绵延的被单。卷云有纤细的外观，像一簇簇头发。乱层云和积雨云是含雨云，呈黑灰色。编写一个程序辨别云的类型。程序的输入应该是描述云特征的事实。程序应输出已辨别出来的云的类型。

7.12 星体被分成不同的颜色群，这些不同的颜色群叫光谱类型。这些类型的范围从蓝色类的“O型”星，到黄色类的“G型”星，再到红色类的“M型”星。星体的光谱类型和它的温度有关：“0”类星的温度超过37000°F；“B”类星的温度从17001°F到37000°F；“A”类星的温度从12501°F到17000°F；“F”类星的温度从10301°F到12500°F；“G”类星的温度从8001°F到10300°F；“K”类星的温度从5501°F到8000°F；“M”类星的温度等于或低于5500°F。星体还可根据其光度（magnitude）分类，光度是星体亮度的一种度量。光度越低，星体越亮。假设光度的取值为-7到15之间。下表列出了一些常见的最亮的星体和它们的光谱类型、光度及与地球的距离(以光年为单位)。编写一个程序，以表示星体光谱类型和光度的两个事实为输入。结果应以这种顺序输出以下信息：有指定光谱类别的所有星体，有指定光度的所有星体，最后是，光谱类型和光度以及与地球的距离（以光年为单位）都匹配的所有星。

星　体	光谱类别	光　度	距　离
天狼星	A	1	8.8
老人星	F	-3	98
大角星	K	0	36
织女星	A	1	26
五车二(御夫星座之一等星)	G	-1	46
参宿七星	B	-7	880
南河三星	F	3	11
猎户星座中的一等星	M	-5	490
牵牛星	A	2	16
毕宿五(金牛座中的一等星)	K	-1	68
角宿星	B	-3	300
心大星	M	-4	250
双子座星	K	1	35
天津四(天鹅座第一亮星)	A	-7	1630

7.13 下表列出了普通宝石的特征，包括硬度（对外部压力的抵抗能力，以Mohs刻度度量）、密度(单位体积的重量，以g/cm^2为单位）和颜色。根据给定的表示宝石硬度、密度和颜色的3个事实，写出所需的规则以判断该种宝石是否是金绿宝石。

宝　石	硬　度	密　度	颜　色
钻石	10	3.52	黄色,褐色,绿色,蓝色,白色,无色
刚玉	9	4	红色,粉红色,黄色,褐色,绿色,蓝色,紫罗兰,黑色,白色,无色
金绿宝石	8.5	3.72	黄色,褐色,绿色
尖晶石	8	3.6	红色,粉红色,黄色,褐色,绿色,蓝色,紫罗兰,白色,无色

（续）

宝石	硬度	密度	颜色
黄玉	8	3.52－3.56	红色,粉红色,黄色,褐色,蓝色,紫罗兰,白色,无色
绿柱石	7.5－8.0	2.7	红色,粉红色,黄色,褐色,绿色,蓝色,白色,无色
锆石	6－7.5	4.7	黄色,褐色,绿色,紫罗兰,白色,无色
石英	7	2.65	红色,粉红色,绿色,蓝色,紫罗兰,白色,黑色,无色
电气石	7	3.1	红色,粉红色,黄色,褐色,绿色,蓝色,白色,黑色,无色
橄榄石	6.5－7	3.3	黄色,褐色,绿色
硬玉	6.5－7	3.3	红色,粉红色,黄色,褐色,绿色,蓝色,紫罗兰,白色,黑色,无色
蛋白石	5.5－6.5	2－2.2	红色,粉红色,黄色,褐色,白色,黑色,无色
软玉	5－6	2.9－3.4	绿色,白色,黑色,无色
绿松石	5－6	2.7	蓝色

7.14 写一个CLIPS程序，以帮助选择种植适当的灌木。下表列出了几种灌木，并显示每种灌木是否具有某些特征，包括耐寒力、抗阴暗的能力、耐旱能力、抗湿土能力、抗酸土能力、适应城市居住的能力（抵抗严重污染的能力）、在容器中生长的能力；灌木是否容易养护、是否生长迅速。一个黑点表示此灌木具有该特征。程序的输入应该是事实，以表示灌木必须具有的所需要的特征，程序的输出结果应列出具有每一种指定特征的植物。

灌 木	寒冷	阴暗	干旱	湿土	酸土	城市	容器(中生长)	易于养护	生长
法国八仙花		•				•	•		•
夹竹桃						•	•	•	•
北方月桂树	•	•	•	•		•		•	•
金银花						•	•	•	•
栀子		•			•		•		
欧洲刺柏	•		•		•	•		•	
甜椒灌木	•	•		•	•			•	
鞑靼山茱萸	•	•		•		•		•	
日本 aucuba		•	•				•	•	
沼泽杜鹃花		•		•	•		•		

7.15 在某堆栈中，第一个压入栈的值是最后一个弹出的值；而最后一个压入的值是第一个弹出的值。某个队列以相反方式工作——第一个添加的值是第一个移出队列的值，而最后一个加入队列的值是最后一个移出队列的值。写出添加和移出队列的规则。假设只存在一个队列。

7.16 编写一条或更多条规则，生成一个base事实的全部排列情况，并且打印。例如，事实

```
(base-fact red green blue)
```

应该产生的输出结果为：

```
permutation is (red green blue)
permutation is (red blue green)
permutation is (green red blue)
permutation is (green blue red)
permutation is (blue red green)
permutation is (blue green red)
```

7.17 给出一个形为

```
(input <value))
```

的事实，写出规则，能使一个有限状态机从目前状态转换到下一个状态。状态机和它的弧要求表示为事实。状态机进入下一个状态时，输入的事实应该被撤销。根据图 3.5 和图 3.6，在有限状态机上测试你编写的规则和事实。

参考文献

(Firebaugh 88). Morris W. Firebaugh, *Artificial Intelligence: A Knowledge-Based Approach*, Boyd & Fraser Publishing Co., pp. 224–226, 1988.

第8章　高级模式匹配

8.1　概述

第7章所述的规则阐述了模式与事实的简单模式匹配问题。本章将介绍几个概念，以提供对事实匹配和控制的强大功能。第一个概念是字段约束。然后介绍CLIPS中函数的使用，包括一些基本算术和I/O操作函数。并将讨论控制规则执行的几种技巧。我们将使用多组规则来建立比上一章功能更强大、结构更复杂的程序。一些有关输入、值的比较和产生循环的技术将被演示，另外，也包括应用规则来规定控制知识的技术。除了介绍模式CE之外，还会介绍一些其他类型的条件元素。这些条件元素允许单独的一条规则去执行几条规则的功能，并且提供了虚事实匹配的能力。

8.2　字段约束

非（NOT）字段约束

除了具有文字常量和变量约束的基本模式匹配能力之外，CLIPS还有许多功能很强的模式匹配符。这些新增加的模式匹配能力，将通过对确定人群的头发和眼睛颜色问题的重新考虑来进行介绍说明。举例来说，我们要找出所有头发不是褐色的人。其中一种解决方法是为每一种头发颜色写规则。例如，以下规则用于找出有黑头发的人群（第7.19节中的自定义模板person将用于本节的例子）。

```
(defrule black-hair-is-not-brown-hair
   (person (name ?name) (hair black))
   =>
   (printout t ?name " does not have brown hair"
               crlf))
```

另一条规则可找出有金色头发的人群。

```
(defrule blonde-hair-is-not-brown-hair
   (person (name ?name) (hair blonde))
   =>
   (printout t ?name " does not have brown hair"
           crlf))
```

此外，还可写出其他的规则来找出有红头发的人群。运用这种方式写出规则的问题来检查条件为头发不是褐色。上面的规则试图以一种迂回的方式检查这一条件，即这些规则选出所有非褐色的头发颜色，同时要为每一种颜色写规则。如果能规定所有的颜色，那么这种方法就行得通。在本例中，假定头发可以是任何颜色（甚至是紫色或绿色）也许会更简单。

解决这一问题的一种方法是使用**字段约束**（field constraint）来限制一个字段可能在LHS中的值。其中一种被称作**连接约束**（connective constraint），命名的原因在于它被用于连接变量和约束）。有3种连接约束，第一种叫**非约束**（not constraint），用波浪号~来表示。非约束作用于其后的约束或变量上。如果其后的约束同一个字段相匹配，则非约束成功；否则，非约束失败。本质上，非约束对其后的约束结果进行否定。

使用非约束可使找出无褐色头发人群的规则大大简化：

```
(defrule person-without-brown-hair
   (person (name ?name) (hair ~brown))
   =>
   (printout t ?name " does not have brown hair"
               crlf))
```

通过使用非约束，该规则与许多需指明每种可能头发颜色的规则具有同样的效果。

或（OR）字段约束

第二个连接约束是**或约束**（or constraint），用一竖杠符号 | 表示。或约束允许1个或多个可能的值与一个模式的字段相匹配。

例如，下面这条规则使用了或约束来找出全部有黑色或褐色头发的人：

```
(defrule black-or-brown-hair
   (person (name ?name) (hair brown | black))
   =>
   (printout t ?name " has dark hair" crlf))
```

声明事实（person (name Joe) (eye blue) (hair brown)) and (person (name Mark) (eye brown) (hair black)) 会为议程中的每一事实激发这一规则。

与（AND）字段约束

第三种类型的连接约束是**与约束**（and constraint）。它不同于在第7.15节中讲过的与条件元素。与约束用符号 & 来表示。与约束一般与其他约束联合应用才有意义，否则，没有多大实用价值。

与约束的一个使用场合是在需要给变量的约束实例以附加限制时。例如，假定一条规则被这样的person事实激发，即一个人的头发为褐色或黑色，通过使用或（OR）约束可容易地表达寻找该事实的模式，如上例所示。然而，如何识别头发的颜色值呢？解决方法是通过使用与约束，把一个变量约束到匹配的颜色上，然后打印出该变量：

```
(defrule black-or-brown-hair
   (person (name ?name) (hair ?color&brown|black))
   =>
   (printout t ?name " has " ?color " hair" crlf))
```

变量？color将约束到与black | brown限制条件匹配的任一种颜色上。

与约束和非约束合用同样有效。例如，下述规则在一个人的头发既非黑色又非褐色的情况下激发：

```
(defrule black-or-brown-hair
   (person (name ?name)
           (hair ?color&~brown&~black))
   =>
   (printout t ?name " has " ?color " hair" crlf))
```

字段约束组合

字段约束与变量和其他文字值混合使用，可以提供极强的模式匹配能力。比如，需要这样一个规则以确定是否有满足以下条件的两个人：第一个人有蓝眼睛或绿眼睛，但没有黑头发；第二个人眼睛颜色与第一个人不同，头发颜色与第一个人一样，或者是红色的。则可用以下规则匹配这些约束：

```
(defrule complex-eye-hair-match
   (person (name ?name1)
           (eyes ?eyes1&blue|green)
           (hair ?hair1&~black))
   (person (name ?name2&~?name1)
           (eyes ?eyes2&~?eyes1)
           (hair ?hair2&red|?hair1))
  =>
   (printout t ?name1 " has " ?eyes1 " eyes and "
               ?hair1 " hair" crlf)
   (printout t ?name2 " has " ?eyes2 " eyes and "
               ?hair2 " hair" crlf))
```

本例值得深入研究。当眼睛为蓝色或绿色这一事实匹配时，第一个模式的eyes（眼睛）槽中的限制条件？eyes1&blue | green将第一个人的眼睛颜色约束给变量？eyes。当头发不是黑色这一事实匹配时，第一个模式的hair（头发）槽中的限制条件？hair1 &～black将约束给变量？hair1。

第二个模式的 name（姓名）槽中的限制条件？name2&～？name1 执行一项很有用的操作。当变量？name2 与变量？name1 的值不相等时，把事实 person 的 name 槽值约束给变量？name2。如果姓名有惟一的标识符（例如，在事实库中不存在两个相同的人名），就能保证两个模式不会匹配于相同的事实。这不可能发生于这两个模式，因为第二个人的眼睛颜色一定不同于第一个人的。然而，这是一种有用的技术，它有许多应用。第 12.2 节将说明如何使用事实地址来确保事实相异，从而实现这种技术。这样就能使规则在人们姓名相同而头发和眼睛相异的情形下仍能正常运作。

第二个模式的 eyes 槽中的限制条件？eyes2&～？eyes1 如同前面的限制条件一样执行相同的检查。如果头发的颜色是红色或与变量？hair1 的值一样，则第二个模式的 hair（头发）槽中的最后一个限制条件？hair2 &red | ？hair1 将把第二个人的头发的颜色值约束给变量？hairs2。须指出，如果变量被用作或字段约束中的一部分时，该变量必须已被约束。

只有当这些变量是某字段中的第一个条件且分别出现，或者是用与连接约束将它们与其他条件连接成一体时，这些变量才必须已被约束。例如，以下语句：

```
(defrule bad-variable-use
   (person (name ?name) (hair red|?hair))
 =>
   (printout t ?name " has " ?hair " hair " crlf))
```

会产生一个错误结果，因为？hair 没有被约束。

对组合限制条件的最后一点说明是，有些组合限制条件是不起实际作用的。例如，用与约束来连接字符常量（例如 black&blue）会引起该条件总不能被满足，除非两个字符常量是相同的。同样，用 or 约束来连接相反的字符常量（例如～black|～blue）会使该条件总是成立。

8.3 函数和表达式

基本数学函数

CLIPS 除了可以处理用符号表示的事实之外，还可以进行计算。请记住，像 CLIPS 这类专家系统语言不是专为数据处理而设计的。虽然 CLIPS 的数学功能非常强大，但主要是用于处理应用程序中有关推理的数据操作。其他计算机语言如 FORTRAN 更适合于不需要或只需要很少推理的数据处理。CLIPS 提供了最基本的算术运算符，见表 8.1（附录 E 列出了 CLIPS 提供的一系列其他数学函数。随书附送的光盘中的 Basic Programming Guide 提供了更详尽的 CLIPS 函数）。

表 8.1 CLIPS 基本算术运算符

算术运算符	意　义
+	加法
-	减法
*	乘法
/	除法

CLIPS 中的数学表达式是按 LISP 风格表示的。在 LISP 和 CLIPS 中数学表达式必须写成**前缀形式**（prefix form），如习惯写法 2+3 要写成（+ 2 3）。我们平时写的数学表达式叫**中缀形式**（infix form），即算术运算符放在两个**操作数**（operand）或**参数**（argument）之间。但在 CLIPS 中是采用前缀形式，即运算符必须放在操作数的前面，数学表达式用符号括起。

把中缀形式的数学表达式转换为前缀形式较为容易。例如，假设要检验两个分数是否成正比例，习惯用中缀形式可写成：

```
(y2 - y1) / (x2 - x1)  >  0
```

注意大于号>是 CLIPS 中一个用于判断前一个数是否大于后一个数的函数（ > 函数的描述见附录 E 及 Basic Programming Guide）。为了将此式写成前缀形式，可把分子认为是（Y），分母认为是（X）。这样，上述表达式可写成

```
(Y) / (X)  >  0
```

除法的前缀形式是：

```
(/ (Y) (X))
```

因为前缀形式中运算符在变量的前面，除法结果必须检验以确定它是否大于0。所以。前缀形式是这样的：

```
(> (/ (Y) (X)) 0)
```

在中缀形式中，Y＝y2－y1。由于建立的是前缀表达式，故需使用前缀形式。所以，Y代替的是（－y2 y1），而X代替的是（－x2 x1）。用（Y）和（X）的前缀形式来表示，最后会得到这样一个表达式：

```
(> (/ (- y2 y1) (- x2 x1)) 0)
```

在CLIPS中求数学表达式（以及其他表达式）最简单的方法是在顶层提示符状态下求此表达式的值。例如，在CLIPS提示符后输入（＋ 2 2）会得到以下结果：

```
CLIPS> (+ 2 2)↵
4
CLIPS>
```

计算机会显示出正确结果：4。一般来说，任何待求值的CLIPS表达式都可以在顶层提示符状态下输入。大部分函数，如加法运算，都会有一个**返回值**（return value）。这个返回值可能是一个整数、一个浮点数、一个符号、一个字符串或者甚至是一个多字段值。其他函数，如facts命令和agenda命令，就没有返回值。这种没有返回值的函数一般都有所谓的**副作用**（side effect）。facts命令的副作用就是列出事实表中的事实。

其他数学函数也可以在提示符状态下运行。以下一些例子展示了其他表达式的计算过程：

```
CLIPS> (+ 2 3)↵
5
CLIPS> (- 2 3)↵
-1
CLIPS> (* 2 3)↵
6
CLIPS> (/ 2 3)↵
0.66666667
CLIPS>
```

注意，除法的结果可能在最后一个数位出现舍入误差。此结果在不同机器上可能有所不同。

如果加、减和乘法运算中的所有操作数都是整数，则其结果也是整数。只要其中有一个是浮点数，则得出的结果也是浮点数。除法函数的第一个操作数总是自动转化为浮点数，因此其结果也为浮点数。例如：

```
CLIPS> (+ 2 3.0)↵
5.0
CLIPS> (+ 2.0 3)↵
5.0
CLIPS> (+ 2 3)↵
5
CLIPS>
```

可变参数运算

用前缀形式来表达不固定的操作数个数相当简单。许多CLIPS函数允许操作数的个数不固定。在加、减和乘法运算表达式中，参与运算的操作数可多于两个。对于2个以上的操作数，其运算顺序是一样的。下面的例子展示3个操作数参与运算的情况。运算顺序是从左到右。

```
CLIPS> (+ 2 3 4)↵
9
CLIPS> (- 2 3 4)↵
-5
CLIPS> (* 2 3 4)↵
24
CLIPS> (/ 2 3 4)↵
0.16666667
CLIPS>
```

再次提醒注意的是，除法的结果的最后一位数可能会随计算机的不同而不同。

优先级和嵌套表达式

CLIPS 和 LISP 的一个重要特点是它们没有内置的数学运算符的优先级。在其他计算机语言里，乘法和除法的优先级高于加法和减法运算，即计算机会优先做高优先级的运算。在 CLIPS 和 LISP 中，所有的运算都是从左到右依次进行，优先级只由括号确定。

混合运算也可以用前缀形式运行。例如，假定要求下面中缀形式的表达式：

```
2 + 3 * 4
```

通常是 3 先乘以 4，再加上 2，得出结果。但在 CLIPS 中，优先级必须明显地表示出来，即通过如下形式的表达式进行计算：

```
CLIPS> (+ 2 (* 3 4))↵
14
CLIPS>
```

在此准则下，最里面括号内的运算首先执行，即 3 先乘以 4，所得的结果再加上 2。如果所需的运算是（2+3）*4，即先执行加法，则表达式将是：

```
CLIPS> (* (+ 2 3) 4)↵
20
CLIPS>
```

一般来说，表达式可以自由地嵌套到其他表达式中，因此，一个表达式可以嵌套在 assert 命令中。例如，

```
CLIPS> (clear)↵
CLIPS> (assert (answer (+ 2 2)))↵
<Fact-0>
CLIPS> (facts)↵
f-0    (answer 4)
For a total of 1 fact.
CLIPS>
```

而且，因为函数的名称也是符号，所以使用它们就像使用其他符号一样。例如，将其作为 fact 事实中的字段：

```
CLIPS> (clear)↵
CLIPS> (assert (expression 2 + 3 * 4))↵
<Fact-0>
CLIPS> (facts)↵
f-0    (expression 2 + 3 * 4)
For a total of 1 fact.
CLIPS>
```

然而，在 CLIPS 中括号是定界符，因此，不可能像使用其他符号那样使用它们。为了在事实中使用它们，或将它们作为函数的操作数，就需要用引号把它们括起来，使之成为一个字符串。

8.4 使用规则求和

作为一个使用函数进行运算的简单例子，思考求一组矩形面积的和。矩形的长和宽可以用以下自定义模板规定：

```
(deftemplate rectangle (slot height) (slot width))
```

矩形面积之和可以使用一个有序事实规定，如：

```
(sum 20)
```

一个自定义事实语句包含以下简单信息：

```
(deffacts initial-information
   (rectangle (height 10) (width 6))
   (rectangle (height 7) (width 5))
   (rectangle (height 6) (width 8))
   (rectangle (height 2) (width 5))
   (sum 0))
```

最初试图产生求矩形面积之和的规则可能是：

```
(defrule sum-rectangles
   (rectangle (height ?height) (width ?width))
   ?sum <- (sum ?total)
   =>
   (retract ?sum)
   (assert (sum (+ ?total (* ?height ?width)))))
```

但这条规则会不停地循环运行。先撤销 sum 事实，然后再次声明此事实将通过单个 rectangle 事实产生一次循环。一个解决办法是在将各个矩形面积加到 sum 事实之后，才撤销 rectangle 事实。这防止了该规则用不同的 sum 事实激活同一个 rectangle 事实。如果要保持 rectangle 事实，则要用别的方法。对于每个 rectangle 事实，可创建一个临时事实，它包含了将要加到和上去的矩形区域。这个临时事实可以撤销，从而防止了无限循环。以下是修改后的程序：

```
(defrule sum-rectangles
   (rectangle (height ?height) (width ?width))
   =>
   (assert (add-to-sum (* ?height ?width))))

(defrule sum-areas
   ?sum <- (sum ?total)
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (assert (sum (+ ?total ?area))))
```

以下的运行结果显示了这两个规则如何相互配合计算出矩形面积的总和：

```
CLIPS> (unwatch all)↵
CLIPS> (watch rules)↵
CLIPS> (watch facts)↵
CLIPS> (watch activations)↵
CLIPS> (reset)↵
==> f-0     (initial-fact)
==> f-1     (rectangle (height 10) (width 6))
==> Activation 0       sum-rectangles: f-1
==> f-2     (rectangle (height 7) (width 5))
==> Activation 0       sum-rectangles: f-2
==> f-3     (rectangle (height 6) (width 8))
==> Activation 0       sum-rectangles: f-3
==> f-4     (rectangle (height 2) (width 5))
==> Activation 0       sum-rectangles: f-4
==> f-5     (sum 0)
CLIPS> (run)↵
FIRE    1 sum-rectangles: f-4
==> f-6     (add-to-sum 10)
==> Activation 0         sum-areas: f-5,f-6
FIRE    2 sum-areas: f-5,f-6
<== f-5     (sum 0)
<== f-6     (add-to-sum 10)
==> f-7     (sum 10)
FIRE    3 sum-rectangles: f-3
==> f-8     (add-to-sum 48)
==> Activation 0   sum-areas: f-7,f-8
FIRE    4 sum-areas: f-7,f-8
<== f-7     (sum 10)
<== f-8     (add-to-sum 48)
==> f-9     (sum 58)
FIRE    5 sum-rectangles: f-2
==> f-10    (add-to-sum 35)
==> Activation 0   sum-areas: f-9,f-10
FIRE    6 sum-areas: f-9,f-10
<== f-9     (sum 58)
<== f-10    (add-to-sum 35)
==> f-11    (sum 93)
FIRE    7 sum-rectangles: f-1
==> f-12    (add-to-sum 60)
==> Activation 0   sum-areas: f-11,f-12
FIRE    8 sum-areas: f-11,f-12
```

```
<== f-11     (sum 93)
<== f-12     (add-to-sum 60)
==> f-13     (sum 153)
CLIPS> (unwatch all)↵
CLIPS>
```

当 reset 命令声明 rectangle 事实后，sum-rectangles 规则被激活 4 次。每次激活 sum-rectangles 规则，就声明一事实，激活 sum-areas 规则。sum-areas 规则将矩形面积加到中间和上，并删除 add-to-sum 事实。因为 sum-rectangles 规则与 sum 事实不匹配，所以，声明一个新的 sum 事实时，不会再次激活 sum-rectangles 规则。

8.5 BIND 函数

当函数产生副作用时，用一个临时变量来存放结果以防止重复计算是一个好方法。bind 函数可以用于将一个变量的值约束为表达式的值。其语法结构是：

```
(bind <variable> <value>)
```

被约束的变量＜variable＞使用单字段变量语法结构。新的值＜value＞应该是一个求单字段或多字段值的表达式。例如，sum-areas 规则可以打印出总和以及加到总和上的各个矩形的面积。

```
(defrule sum-areas
   ?sum <- (sum ?total)
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (printout t "Adding " ?area " to " ?total crlf)
   (printout t "New sum is " (+ ?total ?area) crlf)
   (assert (sum (+ ?total ?area))))
```

注意，表达式（+ ? total ? area）在规则的 RHS 中用了两次。用一个 bind 函数来代替这两个计算式可以省去不必要的计算。用 bind 函数改写后的规则是：

```
(defrule sum-areas
   ?sum <- (sum ?total)
   ?new-area <- (add-to-sum ?area)
   =>
   (retract ?sum ?new-area)
   (printout t "Adding " ?area " to " ?total crlf)
   (bind ?new-total (+ ?total ?area))
   (printout t "New sum is " ?new-total crlf)
   (assert (sum ?new-total)))
```

bind 函数除了可以在规则的 RHS 上创建新的变量之外，还可以用来重新约束规则 LHS 上的变量。

8.6 I/O 函数

Read 函数

专家系统通常需要计算机用户输入信息。CLIPS 允许使用 read 函数从键盘读入信息。Read 函数在基本语法中不需要自变量。以下的例子说明如何用 read 函数来输入数据：

```
CLIPS> (clear)↵
CLIPS>
(defrule get-first-name
   =>
   (printout t "What is your first name? ")
   (bind ?response (read))
   (assert (user's-name ?response)))
CLIPS> (reset)↵
CLIPS> (run)↵
What is your first name? Gary↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (user's-name Gary)
For a total of 2 facts.
CLIPS>
```

注意，每次读入的事实都要以回车确认，而且每次只能读入一个字段。在第一个字段后到回车之间输入的所有附加字符都被丢弃。例如，如果 get-first-name 规则试图读入下列输入的姓和名：

```
Gary Riley↵
```

的话，则只有第一个字符 Gary 会被读入。为读入所有输入，必须用双引号将其括住。当然，输入一旦用双引号括起，就成了一个文字字段了。Gary 和 Riley 字段不容易分开访问。

Read 函数允许输入除符号、字符串、整数和小数之外的字段，如允许输入括号。以下命令行展示了这种情况：

```
CLIPS> (read)↵
(↵
"("
CLIPS>
```

除了键盘输入和终端输出之外，CLIPS 还有文件读写功能。在读或写文件之前，必须先用 open 函数打开该文件。能同时打开的文件数目取决于计算机硬件和操作系统。

OPEN 函数的语法结构是：

```
(open <file-name> <file-ID> [<file-access>])
```

例如，

```
(open "input.dat" data "r")
```

open 的第一个参数＜file-name＞表示文件名字符串。该例子中，文件名是 input.dat。文件名还可以包括其路径信息（此文件所在的目录）。各个操作系统的路径格式都有所不同，因此用户要熟悉自己计算机的操作系统。

第二个参数＜file-ID＞是 CLIPS 与文件联系的**逻辑名**（logical name）。逻辑名是一个全局名称，因此，CLIPS 可以在提示符状态下或语句里访问该文件。虽然逻辑名可以取与文件名相同的名字，但为了避免混淆，最好另取一个名字。在例子里，data 就是 input.dat 的逻辑名。当然，也可以取其他有意义的名字，如 input 或 file-data。

运用逻辑名的一个好处是，可以在不改变程序的情况下将文件名替换。因为文件名只用在 open 函数中，后来由它的逻辑名引用，所以，当要读另一个文件时，只要改变 open 函数即可。

第三个参数＜file-access＞是一个表示文件存取的4种方式之一的字符串。表8.2列出了这4种存取方式。

表 8.2 文件存取方式

方 式	操 作
"r"	只读
"w"	只写
"r+"	读或写
"a"	只添加

如果选项＜file-access＞默认，则系统将存取方式默认为“r”。有一些存取方式可能在某些操作系统上不通用。大部分的操作系统支持“只读（输入数据）”和“只写（输出数据）”这两种方式，其余两种方式可能无效。

请记住，文件打开的结果可能随机器而不同。例如，IBM PC 机上的 MS-DOS 或 Unix 操作系统不支持多文件版本，一个已被打开的“只写”文件可能会覆盖当前已存在的文件。相反，VAX 上的 VMS 里，当文件已存在且用“只写”方式打开时，会另外创建一个新的文件版本。

open 函数是一种谓词函数（在第 8.8 小节中描述）。如果文件成功打开，函数返回 TRUE，否则返回 FALSE。返回结果可用作错误检测。例如，如果用户提供的文件名不存在而试图在“只读”方式下打开文件，open 函数会返回 FALSE。打开文件的准则可以对此进行判断并作出相应的操作提示用户输入另一个文件名。

Close 函数

一旦文件存取操作不再需要之后，应该将文件关闭。如果文件不关闭，写入的信息就不能保存在文件里。而且，文件打开时间越长，由于断电或其他故障导致信息不能被保存的机会就越大。

close 函数的一般格式是：

```
(close [<file-ID>])
```

可选项＜file-ID＞表示要关闭文件的逻辑名。如果＜file-ID＞项默认，则全部打开的文件都会被关闭。例如，

```
(close data)
```

表示关闭 CLIPS 中逻辑名为 data 的文件。语句

```
(close input)
(close output)
```

则分别关闭逻辑名为 input 和 output 的文件。注意，不同的文件要用不同的 close 语句分别关闭。

使用文件时要记住，已打开的文件最后都要用 close 函数关闭，这一操作非常重要。特别地，如果不发出命令进行这一操作，则写入文件的信息可能会丢失。CLIPS 不会提示用户关闭一个已打开的文件。系统只有唯一的一个安全措施，就是当发出 exit（退出）命令时，所有打开的文件都会被关闭。

对文件进行读写操作

前面的例子里，所有的输入都是通过键盘读入，所有的输出都是在终端送出。逻辑名的使用实现了事实通过其他方式输入和输出。在第 7 章里讲过的 printout 函数就是运用逻辑名 t 将事实输出到屏幕上的。还可以在 printout 函数中使用其他逻辑名将数据输出到除屏幕以外的其他设备。

当把逻辑名用于输出功能时，逻辑名 t 会将输出写到标准输出设备，一般是指终端。同样，当把逻辑名用于输入功能时，逻辑名 t 会从**标准输入设备**（standard input device）读入数据，标准输入设备一般是指键盘。

下面的例子说明了使用逻辑名写入文件的方法：

```
CLIPS> (open "example.dat" example "w")↵
TRUE
CLIPS> (printout example "green" crlf)↵
CLIPS> (printout example 7 crlf)↵
CLIPS> (close example)↵
TRUE
CLIPS>
```

首先，用只读方式打开文件 example.dat，这样，数值可以写入该文件。Open 函数把文件 example.dat 与逻辑名 example 联系起来。printout 函数的第一个参数是逻辑名 example，用此逻辑名将值 green 和 7 写入文件 example.dat。值写入文件后，通过 close 命令关闭该文件。

既然这些值已被写入文件中，故可以使用只读方式打开此文件，并使用 read 函数访问这些值。read 函数的一般格式是：

```
(read [<logical-name>])
```

如果没有给出参数，read 函数默认从标准输入设备 t 读入。上述 read 函数的例子利用了这个默认的逻辑名称。以下的例子说明 read 函数使用逻辑名从文件 example.dat 获取这些值：

```
CLIPS> (open "example.dat" example "r")↵
TRUE
CLIPS> (read example)↵
green
CLIPS> (read example)↵
7
CLIPS> (read example)↵
EOF
CLIPS> (close example)↵
TRUE
CLIPS>
```

首先，打开文件 example.dat，但此次用只读方式打开。注意，第二次调用 open 函数时，无须使用 r 选项来打开文件 example.dat，因为 r 是默认值。打开文件后，read 函数用于从相关的逻辑名为 example 的文件中获取值 green 和 7。注意，第三次调用 read 函数返回符号 EOF。当输入函数试图读取文件

末尾之后的数据时，CLIPS 的输入函数会返回这个值。通过检查 read（或其他输入）函数的返回值，可以确定文件何时已没有要读出的数据了。

Format 函数

CLIPS 程序经常使用格式输出，例如，排列表格数据。虽然可用 printout 函数，但有一个专用于格式化的函数，format 函数，它提供众多格式化风格。Format 函数的语法是：

```
(format  <logical-name> <control-string>
         <parameters>*)
```

format 函数分为若干部分，<logical-name>是输出目的地的逻辑名。默认的标准输出设备用逻辑名 t 规定。之后是**控制字符串**（control string），它必须置于双引号之内。控制字符串由**格式符**（format flag）组成，说明 format 函数的参数的打印方式。控制字符串后是参数表。控制字符串的格式符个数决定格式化参数的个数。参数可以是常量或表达式。format 函数返回值是格式化字符串。如果 format 命令使用 nil 为逻辑名，则不会产生打印输出（打印到终端或文件），但仍然返回格式字符串。

以下是一个 format 函数的例子，它建立一个格式化字符串，包含一个人名（预留 15 个空格）和年龄。注意例子中的人名和年龄是如何按列对齐的。format 函数用于显示数据列。

```
CLIPS> (format nil "Name: %-15s Age: %3d"
                   "Bob Green" 35)↵
"Name: Bob Green       Age: 35"
CLIPS> (format nil "Name: %-15s Age: %3d"
          "Ralph Heiden" 32)↵
"Name: Ralph Heiden    Age: 32"
CLIPS>
```

格式符总是以一个“%”符号开始。普通字符串如“Name:”也可以置于控制字符串中，并在输出中打印。若干格式符并不对参数格式化。例如，“%n”用于在输出中打印一个回车/换行符，就像在 printout 命令中使用标识符 crlf 一样。

在这个例子中，格式符“%-15s”的作用是在 15 个字符长的一列中打印名字。符号“-”说明输出左对齐，符号“s”说明要打印一个字符串或标识符。格式符“%3d”说明数字作为整数右对齐打印在 3 字符长的列中。若数值 5.25 作为格式符的参数，则打印 5，因为整数格式中不允许小数部分。

注意当 format 函数在规则的 RHS 中使用时，其返回值通常被忽略。在这些情况下，逻辑名将与一个文件或表示输出到屏幕的 t 相联系。格式符的一般格式是：

```
%-M.Nx
```

其中“-”是可选的，表示**左对齐**（left justify），默认是**右对齐**（right justify）。当预留字符数多于实际打印字符数时，则出现对齐问题。若是此情况，则左对齐使数值在左边打印，右边为空格符。右对齐使数值在右边打印，左边为空格符。

M 规定列字段的宽度。至少输出 M 个字符。除非 M 以 0 开始（此时使用 0），否则，一般用空格填充到 M 列。如果超出 M 列，format 函数将按需要扩大宽度。

N 是可选说明符，说明小数点后打印的数字位数。默认是浮点数的小数点后 6 位数字。

x 说明格式显示。表 8.3 给出了格式显示说明。

表 8.3 格式显示说明

字 符	意 义
d	整数
f	浮点数
e	指数（10 的乘方格式）
g	通用（数值型）；以最短格式显示
o	八进制；无符号数值（N 说明符不适用）
x	十六进制；无符号数值（N 说明符不适用）
s	字符串；字符串用引号引出（引号除外）
n	回车/换行符
%	“%”字符本身

Readline 函数

Readline 函数用于读入整行输入信息。语法是：

```
(readline [<logical-name>])
```

与 read 函数相同，logicalname（逻辑名）是可选的。若没有提供逻辑名或使用逻辑名 t，则将从标准输入设备读入输入信息。Readline 函数把下一行与逻辑名相联的输入信息源直到（包括）回车符的整行输入作为一个返回字符串。若到达文件末尾，则 readline 返回标识符 EOF，只有当逻辑名与一个文件相联时才出现此情况。以下例子说明 readline 函数的用法：

```
CLIPS> (clear)↵
CLIPS>
(defrule get-name
   =>
   (printout t "What is your name? ")
   (bind ?response (readline))
   (assert (user's-name ?response)))
CLIPS> (reset)↵
CLIPS> (run)↵
What is your name? Gary Riley↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (user's-name "Gary Riley")
For a total of 2 facts.
CLIPS>
```

例子中，名字 Gary Riley 存储在 user’s-name 事实中单独作为一个字段。因为它是单独作为一个字段存储，所以，不可能用模式变量从字段中直接获取姓氏和名字。使用 explode $ 函数，它接受一个单独的字符串且把它转化为一个多字段值，我们可以把 readline 函数返回的字符串转化为一个多字段值，这些字段会声明为 user’s-name 事实中的一连串字段。以下例子说明 readline 函数与 explode $ 函数联合使用的情况：

```
CLIPS>
(defrule get-name
   =>
   (printout t "What is your name? ")
   (bind ?response (explode$ (readline)))
   (assert (user's-name ?response)))↵
CLIPS> (reset)↵
CLIPS> (run)↵
What is your name? Gary Riley
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (user's-name Gary Riley)
For a total of 2 facts.
CLIPS>$
```

附录 E 包含了用于建立和操作字符串和多字段值的其他函数列表。本书配套光盘中的《*Basic Programming Guide*》提供了 CLIPS 中可用的所有函数的详细用法。

8.7 棍子游戏

在接下来的几个小节中，将用一个简单的 2 人玩“棍子”游戏来说明各种控制技术，它们是在基于规则的语言中实现的。这个游戏的目的是避免去拿一堆棍子中的最后一支，即谁拿了最后的棍子谁就输了。每个玩家每次可以取走 1、2 或 3 根棍子。要获胜的策略就是注意到当轮到你取的时候，仍剩有 2、3 或 4 支棍子，你就可以让对手输了。最后手里留有 5 根棍子的现象就输了。所以，你必须总要在你取过之后，桌上保持 5 加 4 的整数倍根棍子，才能够让你的对手最后面对 5 根棍子。即是说，在你取完之后，要保持桌上的棍子数为 5、9、13 等等。如果由你先取，但你取完之后剩下的棍子数不是前面说的那些数，则你不会赢，除非你的对手失误。但如果你每次取完都保证了棍子是那么多，那么你会赢。

在游戏进行之前，程序必须确定一些信息。首先，由于是人机对战，所以必须决定由谁先走。另外，要确定这堆棍子的起始数目。此信息可放于自定义事实结构中。然而，要程序的对手从键盘输入是非常简单的。下面的例子显示了 read 函数如何用来输入数据：

```
(deffacts initial-phase
   (phase choose-player))

(defrule player-select
   (phase choose-player)
   =>
   (printout t "Who moves first (Computer: c "
          "Human: h)? ")
   (assert (player-select (read))))

(defrule good-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&c | h)
   =>
   (retract ?phase ?choice)
   (assert (player-move ?player)))
```

当一个特定的事实在事实表中的时候，两条规则使用模式（phase choose-player）来显示它们的适用性。这就叫**控制模式**（control pattern），因为它专门用于控制规则何时可用。**控制事实**（control fact）用来激发控制模式。因为这些规则的控制模式包括文字字段，所以，控制事实必须精确匹配该模式。在这种情况下，用来激发这些规则的控制事实必须是事实（phase choose-player）。这种控制事实对于出错控制是很有用的，比如，当从 read 函数接收到的输入不与字母 c 或 h 匹配的时候，其中 c 代表“计算机”、h 代表“人”。这时，player-select 规则会通过撤销和再声明该控制事实而被重新激发。

下面的输出显示了 player-select 与 good-player-choice 规则如何运作来决定谁先走：

```
CLIPS> (unwatch all)↵
CLIPS> (watch facts)↵
CLIPS> (reset)↵
==> f-0      (initial-fact)
==> f-1      (phase choose-player)
CLIPS> (run)↵
Who moves first (Computer: c Human: h)? c↵
==> f-2      (player-select c)
<== f-1      (phase choose-player)
<== f-2      (player-select c)
==> f-3      (player-move c)
CLIPS>
```

这些规则在正确输入如 c 或 h 情况下会正常工作。但如果输入一个错误的响应，则也不会执行出错检查。有很多情况是重复显示输入请求来改正输入错误。这样做的一种办法示于下例中——对输入请求循环编程。下面的规则与 player-select 和 good-player-choice 规则一起提供了出错检查功能。

```
(defrule bad-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&~c&~h)
   =>
   (retract ?phase ?choice)
   (assert (phase choose-player))
   (printout t "Choose c or h." crlf))
```

再一次注意控制模式（phase choose-player）的用处。它为输入循环提供了基本控制，同时也防止了这组规则在程序执行的其他阶段被激活。

注意 player-select 与 bad-player-choice 这两条规则，看它们如何一起工作，每一条规则提供的事实是用来激活其他规则所必需的。如果对游戏者问题的响应不正确，则 bad-player-choice 规则将撤销控制事实（phase choose-player），然后再声明它，从而使得 player-select 规则被再激活。

8.8 谓词函数

谓词函数（predicate function）被定义为任何可返回符号 TRUE 或 FALSE 的函数。事实上，当处理谓词逻辑时，CLIPS 把不是符号 FALSE 的任何值视为符号 TRUE。一个谓词函数也被认为有一个布尔型返回值。谓词函数既可以**预定义**（predefined）、也可以由**用户定义**（user-defined）。预定义函数是已经由 CLIPS 系统提供了的那些函数。用户定义函数，或称**外部函数**（external function），与预定义函

数不同，它是用户用C或其他编程语言编写并与CLIPS连接而成的。附录E中包括了由CLIPS所提供的谓词函数的列表，它们用来执行布尔逻辑操作、值的比较及特定类型的测试。这些函数的一部分示于下面的对话中：

```
CLIPS> (and (> 4 3) (> 4 5))↵
FALSE
CLIPS> (or (> 4 3) (> 4 5))↵
TRUE
CLIPS> (> 4 3)↵
TRUE
CLIPS> (< 6 2)↵
FALSE
CLIPS> (integerp 3)↵
TRUE
CLIPS> (integerp 3.5)↵
FALSE
CLIPS>
```

8.9 测试条件元素

在许多例子中，重复进行一个计算或重复其他的信息处理是很有用的。处理这些问题常用的办法是设置一个循环。在前面的例子中设置了循环，用来重复用户的响应，直到用户对问题作出了正确的响应为止。然而，在许多情况下，根据对任意表达式计算的结果，循环要自动地终止。

测试条件元素（test conditional element）为计算规则LHS上的表达式提供了有力的方法。测试条件元素（test CE）不是与事实表中的事实进行模式匹配，而是计算表达式。表达式的最外层函数必须为谓词函数。如果表达式计算结果是不为FALSE的任何值，则测试条件元素（test CE）满足，否则，不满足。规则只会在所有的测试条件元素与其他模式都满足时才被激发。测试条件元素语法为：

```
(test <predicate-function>)
```

例如，假设轮到人来操作了。如果只剩下一根棍子，那么人就输了。如果不止一根棍子，那么就该问此人想拿走多少棍子。问游戏者这个问题的规则就应检查是否有不止一根棍子余下。谓词函数“>”可用来表示这个要求，正如下面的测试CE所示，其中？size表示堆中还剩多少根棍子：

```
(test (> ?size 1))
```

一旦游戏者确定了他要拿走的棍子数，就要检查所作的响应，确保这个数是合法的。棍子数首先必须是整数，并且在大于等于1到小于等于3的范围内。游戏者取走的棍子数不能大于棍子堆中的棍子数并且必须取最后一根（如果只余1根）。谓词函数“and”可表达所有这些要求，正如下列测试CE所显示的，其中？choice表示取走的棍子数，？size表示堆中剩余的棍子数：

```
(test (and (integerp ?choice)
           (>= ?choice 1)
           (<= ?choice 3)
           (< ?choice ?size)))
```

同样地，下面的数字是不合法的：非整数、比1小或比3大的数、大于或等于堆中剩余的棍子数。我们可用谓词函数or来描述，如下例所示，其中？choice表示取走的棍子数，？size是堆中剩余的棍子数：

```
(test (or (not (integerp ?choice))
          (< ?choice 1)
          (> ?choice 3)
          (>= ?choice ?size)))
```

下面的规则使用前面的测试CE来检查游戏者输入的数字是否合法。规则中使用事实pile-size来存放堆中剩余棍子数的信息：

```
(defrule get-human-move
   (player-move h)
   (pile-size ?size)
   ; Human player only has a choice when there is
   ; more than one stick remaining in the pile
   (test (> ?size 1))
   =>
```

```
   (printout t
          "How many sticks do you wish to take?")
   (assert (human-takes (read))))

(defrule good-human-move
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (and (integerp ?choice)
              (>= ?choice 1)
              (<= ?choice 3)
              (< ?choice ?size)))
=>
   (retract ?whose-turn ?number-taken)
   (printout t "Human made a valid move" crlf))

(defrule bad-human-move
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (or (not (integerp ?choice))
             (< ?choice 1)
             (> ?choice 3)
             (>= ?choice ?size)))
=>
  (printout t "Human made an invalid move" crlf)
   (retract ?whose-turn ?number-taken)
  (assert (player-move h)))
```

当要取走的棍子数为合法的数字输入时，该程序将结束。控制事实可用来重新激发规则以允许不合法响应时重新输入。Bad-human-move 规则可再次声明控制事实（player-move h），以求再次激活 get-human-move 规则。于是，游戏者可再次输入他要从堆中取走的棍子数。

8.10 谓词字段约束

谓词字段约束（predicate field constraint）即符号“:”，对于直接在模式中执行谓词测试是很有用的。虽然在有些情况下，如在第 9 章中将要讨论的，用谓词字段约束比用测试 CE 更加有效，但是，在许多方面，谓词字段约束类似于在模式之后直接执行一次测试 CE。谓词字段约束的用法就像文字字段约束一样，它可以单独存在于一个字段中，或者作为一个更复杂的字段的一个部分，各部分通过“~”、“&”和“|”等连接字段约束连接起来。谓词字段约束后面总是跟着一个求值函数。在与测试 CE 一起使用时，这个函数必须是一个谓词函数。

例如，前一节的 get-human-move 规则使用了下面两个模式来检查堆中是否有一根以上的棍子：

```
(pile-size ?size)
(test (> ?size 1))
```

这两个模式可由下面一个单独模式代替：

```
(pile-size ?size&:(> ?size 1))
```

谓词字段约束可单独使用，然而，这种情况有实用价值的很少。一般地，一个变量被约束，然后使用谓词字段约束进行测试。当理解一个模式时，把它的谓词字段约束的意思理解成使得（such that）是方便的，例如以前出现的字段约束：

```
?size&:(> ?size 1)
```

可被理解为“约束 ? size 使得 ? size 大于 1”。

谓词字段约束的一个应用是对数据进行出错检查。例如，下面的规则在把数据项加入到总数之前检查它是否为数字型：

```
(defrule add-sum
   (data-item ?value&:(numberp ?value))
   ?old-total <- (total ?total)
   =>
   (retract ?old-total)
   (assert (total (+ ?total ?value))))
```

第一个模式的第二个字段可被理解成“约束 ? value 使得 ? value 是一个数字”。下面两条规则执行同样类型的出错检查，同时，它们也演示了谓词字段约束和连接字段约束“~”、“|”一起使用的情况：

```
(defrule find-data-type-1
   (data ?item&:(stringp ?item)|:(symbolp ?item))
   =>
   (printout t ?item " is a string or symbol "
               crlf))

(defrule find-data-type-2
   (data ?item&~:(integerp ?item))
   =>
   (printout t ?item " is not an integer " crlf))
```

find-data-type-1 规则通过使用 stringp 和 symbolp 谓词函数来检查一个数据项是一个字符串还是一个符号。find-data-type-2 规则先使用 integerp 函数、然后再用连接约束“~”否定此值以检查某数据项不是一个整数。

8.11 返回值字段约束

返回值字段约束（return value field constraint）即“=”，允许函数的返回值用于一个模式内的比较。返回值字段约束与谓词字段约束一样，可以和“~”、“&”或“|”等连接字段约束一起使用。像谓词字段约束一样，返回值字段约束之后也必须跟一个函数。然而，此函数不一定是谓词函数。唯一的要求是函数必须有单一的字段返回值。下面的规则展示了返回值约束如何用于棍子程序中，以决定计算机从棍子堆中取走多少棍子：

```
(deftemplate take-sticks
   (slot how-many)
   (slot for-remainder))

(deffacts take-sticks-information
   (take-sticks (how-many 1) (for-remainder 1))
   (take-sticks (how-many 1) (for-remainder 2))
   (take-sticks (how-many 2) (for-remainder 3))
   (take-sticks (how-many 3) (for-remainder 0)))

(defrule computer-move
   ?whose-turn <- (player-move c)
   ?pile <- (pile-size ?size)
   (test (> ?size 1))
   (take-sticks (how-many ?number)
                (for-remainder =(mod ?size 4)))
   =>
   (retract ?whose-turn ?pile)
   (assert (pile-size (- ?size ?number)))
   (assert (player-move h)))
```

computer-move 规则决定了轮到计算机操作时，它所从堆中取出棍子的合适的数目。第一个模式 CE 确保只有轮到计算机操作时，这条规则方可使用。第二和第三个 CE 是判断堆中剩余的棍子数是否大于 1。如果仅剩一根，那么计算机将被迫取走这根，输掉游戏。最后一个模式与 take-sticks-information 自定义事实一起决定从堆中取走的合适棍子数。mod（为 modulus 的缩略形式）函数返回一个整数，这个整数是第一个变量除以第二个所剩的余数。当理解某个模式时，把返回值字段约束当作等于（equal to）的是有用的。例如，字段约束：

```
=(mod ?size 4)
```

可读作“此字段等于 ? size 模 4 的值”。

计算机总是取走尽可能多的棍子，使当棍子总数被 4 除时，剩余的棍子数为 1。当计算机开始取的时候，如果这个余数为 1，计算机就输了，除非它的对手有失误。在这种情况下，它就会每次只取一根棍子来拖延游戏时间（希望对手失误）。在所有其他情况下，计算机都可以取走适当数量的棍子让对手输。

为了演示返回值字段约束工作的细节，让我们考虑一个具体的例子。假设计算机取的时候，堆中有7根棍子，并轮到计算机取棍子。computer-move规则的头三个CE将被满足，只剩下最后一个CE。函数表达式（mod ? size 4）用数值7来代替? size，于是式子变为（mod 7 4），它的值就为3。只有一个take-sticks事实，它的for-remainder的槽值为3。此事实的how-many槽值为2，因此，计算机会取走2根棍子，使堆中剩下5根棍子。当人在下一次移动后，计算机会取胜。

注意，“=”字段约束只可用于模式CE内，它不同于“=”谓词函数。“=”谓词函数可用于“:字段约束”、“测试CE”、“规则的RHS”或顶层对话提示内。另外，和谓词字段约束一样，通过把返回值约束与“~”、“&”和“|”连接字段约束一起使用，可产生更复杂的字段约束。

8.12 棍子游戏程序

所有需要完成棍子程序的基本技术都已讨论过了。棍子程序的完整清单在Stick.clp文件中，此文件可在本书附送的光盘中找到。在调入该程序之后运行它的情况如下：

```
CLIPS> (reset)↵
CLIPS> (run)↵
Who moves first (Computer: c Human: h)? c↵
How many sticks in the pile? 15↵
Computer takes 2 stick(s).
13 stick(s) left in the pile.
How many sticks do you wish to take? 3↵
10 stick(s) left in the pile.
Computer takes 1 stick(s).
9 stick(s) left in the pile.
How many sticks do you wish to take? 2↵
7 stick(s) left in the pile.
Computer takes 2 stick(s).
5 stick(s) left in the pile.
How many sticks do you wish to take? 1↵
4 stick(s) left in the pile.
Computer takes 3 stick(s).
1 stick(s) left in the pile.
You must take the last stick!
You lose!
CLIPS>
```

8.13 OR条件元素

至此，所有示出的规则都有一个**隐式的and条件元素**（implicit and conditional element）在模式之间。即，除非所有模式都正确，否则，规则不会被激发。CLIPS也提供了下面功能，它可在LHS上规定一个**显式的and条件元素**（explicit and conditional element）和一个**显式的or条件元素**（explicit or conditional element）。

作为一个or CE的例子，下面的规则用在工厂的监视系统中（如第7章讨论的），这里先没有使用or CE。之后，你会明白如何用or CE将它们改写。

```
(defrule shut-off-electricity-1
   (emergency (type flood))
   =>
   (printout t "Shut off the electricity" crlf))
(defrule shut-off-electricity-2
   (extinguisher-system (type water-sprinkler)
                        (status on))
   =>
   (printout t "Shut off the electricity" crlf))
```

更简洁地，不需写两条独立的规则，使用or CE可将这两条规则合并到下面的规则中：

```
(defrule shut-off-electricity
  (or (emergency (type flood))
      (extinguisher-system (type water-sprinkler)
                           (status on)))
  =>
  (printout t "Shut off the electricity" crlf))
```

这是一条使用 or CE 的规则，它与前面两条规则等价。声明两个事实与这一规则的模式相匹配将使规则激发两次，即每个事实激发一次。

除 or CE 之外，还可以使用其他的 CE，对于规则的整个 LHS 来说，它们将是隐式的 and CE 中一部分。例如，下面的规则：

```
(defrule shut-off-electricity
   (electrical-power (status on))
   (or (emergency (type flood))
       (extinguisher-system (type water-sprinkler)
                            (status on)))
   =>
   (printout t "Shut off the electricity" crlf))
```

等价于下面两条规则：

```
(defrule shut-off-electricity-1
   (electrical-power (status on))
   (emergency (type flood))
   =>
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-2
   (electrical-power (status on))
   (extinguisher-system (type water-sprinkler)
                        (status on))
   =>
   (printout t "Shut off the electricity" crlf))
```

由于一个 or CE 产生等价的多条规则，因此，一条规则可能不止一次被那些包含于 or CE 的模式激活。所以，自然会问如何防止多次激发的问题。例如，对于多个事实多次打印信息 Shut off the electricity（关闭电源）是不必要的。毕竟，电源只要关一次就可以了，不用考虑原因有多少次。

阻止其他规则触发的最合适办法可能是改变规则来更新事实表，显示电源已被关闭。被修改的规则如下所示（注意，我们假设在写这些规则时已经有一个机灵的工厂操作者，他的任务就是观察监视系统的输出，然后采取行动。在现实世界中，专家系统可能直接控制各种系统的开和关，同时工厂操作者有一个机制告诉监视系统各种行动已被采取）。

```
(defrule shut-off-electricity
   ?power <- (electrical-power (status on))
   (or (emergency (type flood))
       (extinguisher-system (type water-sprinkler)
                            (status on)))
=>
   (modify ?power (status off)))
   (printout t "Shut off the electricity" crlf))
```

现在，规则仅被触发一次，因为当规则触发时，包括电源各种信息的事实就被修改。这会阻止由其他模式引起的规则的其他激活。同样，如果有必要从事实表中删除支持关闭电源的理由，那么规则将被改写成下面的形式：

```
(defrule shut-off-electricity
   ?power <- (electrical-power (status on))
   (or ?reason <- (emergency (type flood))
       ?reason <- (extinguisher-system
                (type water-sprinkler)
          (status on)))
   =>
   (retract ?reason)
   (modify ?power (status off))
   (printout t "Shut off the electricity" crlf))
```

注意，在 or CE 中的所有模式 CE 都被约束给同一个变量 ? reason。首先，这可能引起一个错误，因为同样的变量一般并不会分配给一条普通规则的不同模式。但使用 or CE 的规则就不同了。因为一个 or CE 会产生多条规则，上面的规则等价于下面两条规则：

```
(defrule shut-off-electricity-1
   ?power <- (electrical-power (status on))
   ?reason <- (emergency (type flood))
   =>
   (retract ?reason)
   (modify ?power (status off))
   (printout t "Shut off the electricity" crlf))

(defrule shut-off-electricity-2
   ?power <- (electrical-power (status on))
   ?reason <- (extinguisher-system
                (type water-sprinkler)
                (status on))
=>
  (retract ?reason)
   (modify ?power (status off))
    (printout t "Shut off the electricity" crlf))
```

通过观察这两条规则，我们可以看到，有必要取一个相同的变量名以便在 RHS 上匹配（retract ? power ? reason）操作。

8.14 AND 条件元素

And CE 的概念与 or CE 相反。与几条 CE 中的任一条满足就可以激发规则不同，and CE 要求所有的 CE 都要满足。CLIPS 在一条规则的 LHS 中自动设置一个隐式的 and CE。例如，规则：

```
(defrule shut-off-electricity
   ?power <- (electrical-power (status on))
   (emergency (type flood))
   =>
   (modify ?power (status off))
   (printout t "Shut off the electricity" crlf))
```

也可用显式的 and CE 改写为：

```
(defrule shut-off-electricity
   (and ?power <- (electrical-power (status on))
        (emergency(type flood)))
   =>
   (modify ?power (status off))
   (printout t "Shut off the electricity" crlf))
```

当然，对整个 LHS 用显式的 and CE 写是没有好处的。提供 and CE 是为了能与其他 CE 共同使用，以建立更复杂的模式。如下面的例子，它可与 or CE 一起使用，以要求多个条件组为真：

```
(defrule use-carbon-dioxide-extinguisher
   ?system <- (extinguisher-system
                (type carbon-dioxide)
                (status off))
   (or (emergency (type class-B-fire))
       (and (emergency (type class-C-fire))
            (electrical-power (status off))))
   =>
   (modify ?system (status on))
   (printout t "Use carbon dioxide extinguisher"
               crlf))
```

如果有 B 类（如油或油脂燃烧）或 C 类（涉及电力设备）火警，而且电源已被关闭，则将激活此规则。在效果上，总是用二氧化碳灭火器来扑灭 B 类火灾；但对于 C 类火警，当关掉电源不能熄灭此火时，才用它来熄灭 C 类火灾。use-carbon-dioxide-extinguisher 规则与下面两条规则等价：

```
(defrule use-carbon-dioxide-extinguisher-1
   ?system <- (extinguisher-system
                (type carbon-dioxide)
                (status off))
   (emergency (type class-B-fire))
   =>
   (modify ?system (status on))
   (printout t "Use carbon dioxide extinguisher"
```

```
                crlf))

(defrule use-carbon-dioxide-extinguisher-2
   ?system <- (extinguisher-system
              (type carbon-dioxide)
              (status off))
   (emergency (type class-C-fire))
   (electrical-power (status off))
   =>
   (modify ?system (status on))
   (printout t "Use carbon dioxide extinguisher"
               crlf))
```

8.15 NOT 条件元素

有时候，能够根据事实表中缺少某一特定事实来激活规则是有作用的。CLIPS 利用 **not 条件元素**（not conditional element），来说明一条规则的 LHS 中缺少某个事实。作为一个简单的例子，监视专家系统可能有以下两条规则用以报告它的状态：

```
IF the monitoring status is to be reported and
   there is an emergency being handled
THEN report the type of the emergency

IF the monitoring status is to be reported and
   there is no emergency being handled
THEN report that no emergency is being handled
```

Not CE 能如下方便地应用到上述简单的规则：

```
(defrule report-emergency
   (report-status)
   (emergency (type ?type))
   =>
   (printout t "Handling " ?type " emergency"
               crlf))

(defrule no-emergency
   (report-status)
   (not (emergency))
   =>
   (printout t "No emergency being handled" crlf))
```

注意这两条规则是互相排斥的。也就是说，它们不能同时存在于一个议程中，这是因为每一条规则中的第 2 个模式不能同时被满足。

变量也能用于否定模式，以产生一些有趣的效果。思考以下在一组表示数字的事实中找出最大数的规则：

```
(defrule largest-number
   (number ?x)
   (not (number ?y&:(> ?y ?x)))
   =>
   (printout t "Largest number is " ?x crlf))
```

第一个模式将约束所有的 number 事实，但第二个模式将不允许该规则起作用，除非对于? x 为最大值的事实。

注意，约束于一个 not CE 内的变量只在 not CE 的范围内保留其值。例如，规则：

```
(defrule no-emergency
   (report-status)
   (not (emergency (type ?type)))
   =>
   (printout t "No emergency of type " ?type crlf))
```

就会产生一个错误，因为变量? type 用于规则的 RHS 中、但它只能约束于该规则的 LHS 中的 not CE 中。

变量的范围同样适用于规则的 LHS。例如，以下的规则确定某日不是一个认识的人的生日：

```
(defrule no-birthdays-on-specific-date
   (check-for-no-birthdays (date ?date))
   (not (person (birthday ?date)))
   =>
   (printout t "No birthdays on " ?date crlf))
```

如果前面的两个 CE 交换位置，如下所示：

```
(defrule no-birthdays-on-specific-date
   (not (person (birthday ?date)))
   (check-for-no-birthdays (date ?date))
   =>
   (printout t "No birthdays on " ?date crlf))
```

则该规则将不再正确工作。即使有任一 person 事实，第一个 CE 也不被满足。约束于第一个 CE 中的变量? date 的值将不影响第二个 CE 中的变量? date 的允许值。这与原始的 no-birthdays-on-specific-date 规则不同，原规则中第一个 CE 中的变量? date 的值限制了在第二个 CE 中去查找特定日子是生日的 person 事实的范围。不像其他模式 CE，一条规则的 LHS 中的 not CE 的不同排列次序会影响该规则的激活。

Not CE 能与其他 CE 一起使用。例如，如下的规则确定不存在两个人的生日是同一天：

```
(defrule no-identical-birthdays
   (not (and (person (name ?name)
                     (birthday ?date))
             (person (name ~?name)
                     (birthday ?date))))
   =>
   (printout t
      "No two people have the same birthday"
      crlf)
```

因为 not CE 能够包含最多 1 个 CE，所以，and CE 可用于 not CE 内。注意，变量? name 和? date 被重复用于 not CE 中去正确地限制对有相同生日的两个人的搜索。

基于以下为 CLIPS 使用的算法，(initial-fact) 模式被添加到任何 and CE（隐式的或显式的）的开始处，此处的第一个 CE 将是 not CE 或者一个 test CE。所以，规则：

```
(defrule no-emergencies
   (not (emergency))
   =>
   (printout t "No emergencies" crlf))
```

被转换为以下规则：

```
(defrule no-emergencies
   (initial-fact)
   (not (emergency))
=>
   (printout t "No emergencies" crlf))
```

理解这种转换在检查由 matches 命令产生的输出中是有用的。同时应该注意，not CE 的事实索引在规则的部分匹配或激活中不会被显示出来。所以，“f-5 ,, f-3” 激活表示，第一个 CE 和具有事实索引 5 的事实匹配；第二个 CE 是一个不和任何事实匹配的 not CE，因此得到满足；第三个 CE 和具有事实索引 3 的事实匹配。

8.16 EXISTS 条件元素

Exists 条件元素（exists conditional element）只需至少存在一个事实与某模式匹配，而不管实际匹配该模式的事实的总数。这样，以某类事实中的一个事实是否存在为根据，就允许一条规则中的一个部分匹配或激活产生出来。例如，假定无论何时出现紧急事件就打印一则消息以指示工厂操作员保持警惕。这一规则可以如下形式写出：

```
(deftemplate emergency (slot type))

(defrule operator-alert-for-emergency
   (emergency)
   =>
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

注意，当多个 emergency 事实被声明时，将出现什么情况——给操作员的消息被多次重复打印：

```
CLIPS> (reset)↵
CLIPS> (assert (emergency (type fire)))↵
<Fact-1>
CLIPS> (assert (emergency (type flood)))↵
<Fact-2>
CLIPS> (run)↵
Emergency: Operator Alert
Emergency: Operator Alert
CLIPS>
```

operator-alert-for-emergency 规则可以用以下方法修改，以避免这一规则在附加的紧急情况下被重新激发：

```
(defrule operator-alert-for-emergency
   (emergency)
   (not (operator-alert))
   =>
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

(not (operator-alert)) CE 防止该规则被重新激发；然而，这种改变规则的方法是假设如果有一个操作员警报，则这个警报就是由 operator-alert-for-emergency（操作人员警惕紧急情况）规则产生的。可能有这种情况：操作人员在一个紧急事件训练中被报警，就像以下这一规则所表示的那样：

```
(defrule operator-alert-for-drill
   (operator-drill)
   (not (operator-alert))
   =>
   (printout t "Drill: Operator Alert" crlf)
   (assert (operator-alert)))
```

注意这种情况：如果这个训练报警规则首先触发，则由于 operator-alert 事实被声明，而使紧急报警规则不能触发。为了修正这一问题，operator-alert 事实可以被修改，使之能存储引起报警的原因。但是，通过增加规则间的控制依赖性，过度使用控制事实来防止规则触发会增加规则的复杂性、降低规则的可维护性。

幸好，这个 operator-alert-for-emergency 规则能够被修改以使用 exists CE，这将可以去掉使用控制事实的必要。不管有多少事实符合 exists CE 中的 CE，exists CE 都将只产生一个部分匹配。所以，如果一个 exists CE 是一条规则的第 N 个 CE，而且前 N-1 个 CE 已经产生 M 个部分匹配，则这第 N 个 CE 所能产生的最大部分匹配数为 M。使用 exists CE，该 operator-alert-for-emergency 规则可重写为：

```
(defrule operator-alert-for-emergency
   (exists (emergency))
   =>
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

这一规则将会产生一个激活，所以，该操作员警报只被打印一次，就像以下对话显示的那样：

```
CLIPS> (reset)↵
CLIPS> (assert (emergency (type fire)))↵
<Fact-1>
CLIPS> (assert (emergency (type flood)))↵
<Fact-2>
CLIPS> (agenda)↵
0      operator-alert-for-emergency: f-0,
For a total of 1 activation.
CLIPS> (run)↵
Emergency: Operator Alert
CLIPS>
```

Exists CE 通过使用 and CE 和 not CE 的组合来实现。exists CE 中的 CE 是封装在一个 and CE、然后再封装在二个 not CE 中的。所以，通过一个 and CE 包围整个 LHS，然后替换 exists CE，从而把该 operator-alert-for-emergency 规则改变成如下形式：

```
(defrule operator-alert-for-emergency
   (and (not (not (and (emergency)))))
   =>
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

因为包围着该规则的 LHS 的 and CE 将 not CE 作为第一个 CE，故（initial-fact）模式 CE 被添加到开头。在 emergency 模式中进行这一增加并去掉无关的 and CE 将产生以下规则：

```
(defrule operator-alert-for-emergency
   (and (initial-fact)
        (not (not (emergency))))
   =>
   (printout t "Emergency: Operator Alert" crlf)
   (assert (operator-alert)))
```

在规则开始处的（initial-fact）模式解释了 f-0 事实索引，该索引是在前面的对话中发出 agenda 命令时为此规则作显示的。如果没有 emergency 事实，则最内层的 not CE 将被满足。如果这个 CE 被满足，则最外层的 not CE 将不被满足，且这一规则也不会被激活。相反，如果有 emergency 事实，则最内层的 not CE 将不被满足。既然这个 CE 不被满足，则最外层的 not CE 就会被满足，且此规则将会被激发。

8.17 FORALL 条件元素

Forall 条件元素（forall conditional element）用于每次出现另一个 CE 时一组 CE 被满足的情况。例如，假设在工业区有一系列位置（如建筑物）正在着火，我们想知道是否每一座已着火的建筑都已疏散，是否有一支消防队已去救火。forall CE 能用于检查这些情况。emergency 事实将被修改成如下形式，包含紧急情况的位置和紧急事件的类型。其他两个自定义模板将被用来显示消防队的位置和某一建筑内的人是否已经疏散。all-fires-being-handled 规则使用这些自定义模板去决定是否已满足适当的条件。

```
(deftemplate emergency
   (slot type)
   (slot location))

(deftemplate fire-squad
   (slot name)
   (slot location))

(deftemplate evacuated
   (slot building))

(defrule all-fires-being-handled
   (forall (emergency (type fire)
                      (location ?where))
           (fire-squad (location ?where))
           (evacuated (building ?where)))
=>
  (printout t
            "All buildings that are on fire " crlf
            "have been evacuated and" crlf
            "have firefighters on location" crlf))
```

每一个符合（emergency（type fire）（location ? where））模式的事实，也是满足（fire-squad（location ? where）模式和（evacuated（building ? where））模式的事实。当自定义模板和自定义规则开始载入，并且发出了 reset 命令时，只要没有火警出现，该规则就会被满足。

```
CLIPS> (watch activations)↵
CLIPS> (reset)↵
==> Activation 0       all-fires-being-handled:
  f-0,
CLIPS>
```

一旦 emergency 事实被声明，这一规则就将失效，直到适当的 fire-squad 和 evacuated 事实被声明为止。

```
CLIPS>
(assert (emergency (type fire)
                   (location building-11)))↵
<== Activation 0   all-fires-being-handled: f-0,
<Fact-1>
CLIPS>
(assert (evacuated (building building-11)))↵
<Fact-2>
CLIPS>
(assert (fire-squad (name A)
                    (location building-11)))↵
==> Activation 0   all-fires-being-handled: f-0,
<Fact-3>
CLIPS>
(assert (fire-squad (name B)
                    (location building-1)))↵
<Fact-4>
CLIPS>
(assert (emergency (type fire)
                   (location building-1)))↵
<== Activation 0   all-fires-being-handled: f-0,
<Fact-5>
CLIPS> (assert (evacuated (building building-1)))↵
==> Activation 0       all-fires-being-handled:
  f-0,
<Fact-6>
CLIPS>
```

如果建筑物 1 的 fire-squad 事实被除去，则该规则失效。删去该建筑物的 emergency 事实将重新使该规则有效。

```
CLIPS> (retract 4)↵
<== Activation 0   all-fires-being-handled: f-0,
CLIPS> (retract 5)↵
==> Activation 0   all-fires-being-handled: f-0,
CLIPS> (run)↵
All buildings that are on fire
have been evacuated and
have firefighters on location
CLIPS>
```

forall CE 一般的格式如下：

```
(forall <first-CE>
        <remaining-CEs>+)
```

为了使 forall CE 得到满足，每一个匹配＜first-CE＞的事实也必须匹配所有＜remaining-CEs〉事实。forall CE 的一般格式可用 and 和 not CE 的组合改写成以下的格式：

```
(not (and <first-CE>
          (not (and <remaining-CEs>+))))
```

8.18 LOGICAL 条件元素

Logical 条件元素（logical conditional element）允许你规定，某事实的存在依赖于另一事实或另一组事实的存在。logical CE 是 CLIPS 为正确性维护所提供的一个工具。作为一个例子，思考以下规则，当一场火灾正在放出有毒气体时，该规则指示消防员需要佩戴氧气面具：

```
(defrule noxious-fumes-present
   (emergency (type fire))
   (noxious-fumes-present)
   =>
   (assert (use-oxygen-masks)))
```

如同以下对话所展示的，无论何时出现这样一场火灾，上面的规则将声明 use-oxygen-masks 事实：

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (assert (emergency (type fire))
               (noxious-fumes-present)))↵
==> f-1    (emergency (type fire))
==> f-2    (noxious-fumes-present)
<Fact-2>
CLIPS> (run)↵
==> f-3    (use-oxygen-masks)
CLIPS>
```

当火被扑灭后、且不再有毒气体放出时，会发生什么事情呢？就像下一个对话显示的那样，撤销 emergency 事实或撤销 noxious-fumes-present 事实并不会影响 use-oxygen-masks 事实。

```
CLIPS> (retract 1 2)↵
<== f-1      (emergency (type fire))
<== f-2      (noxious-fumes-present)
CLIPS> (facts)↵
f-0     (initial-fact)
f-3     (use-oxygen-masks)
For a total of 2 facts.
CLIPS>
```

CLIPS 提供了一个正确性维护机制去建立事实间的依赖关系，这一机制就是 logical CE。如下所示修改 noxious-fumes-present 规则，允许在与规则 LHS 模式匹配的事实和规则 RHS 上声明的事实之间建立一种依赖关系：

```
(defrule noxious-fumes-present
   (logical (emergency (type fire))
            (noxious-fumes-present))
   =>
   (assert (use-oxygen-masks)))
```

当 noxious-fumes-present 规则执行时，在与包含在规则 LHS 中 logical CE 内的模式匹配的事实和规则 RHS 上声明的事实之间就建立起一个链接。对于这一规则，如果 emergency 事实或 noxious-fumes-present 事实被撤销，那么，use-oxygen-masks 规则将同样被撤销，如同以下对话所示的那样：

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (assert (emergency (type fire))
               (noxious-fumes-present)))↵
==> f-1      (emergency (type fire))
==> f-2      (noxious-fumes-present)
<Fact-2>
CLIPS> (run)↵
==> f-3      (use-oxygen-masks)
CLIPS> (retract 1)↵
<== f-1      (emergency (type fire))
<== f-3      (use-oxygen-masks)
CLIPS>
```

use-oxygen-masks 事实接受来自 emergency 事实和 noxious-fumes-present 事实的**逻辑支持**（logical support）。emergency 事实和 noxious-fumes-present 事实向 use-oxygen-masks 事实提供逻辑支持。use-oxygen-masks 事实依赖于 emergency 事实和 noxious-fumes-present 事实。noxious-fumes-present 事实和 emergency 事实是 use-oxygen-masks 事实的依据。

logical CE 不必包含在一条规则 LHS 上的所有模式中。然而，如果使用 logical CE，它就必须包含一条规则中的 LHS 的第一个 CE，并且，在由 logical CE 包含的 CE 之间不能有空隙（gap）。例如，logical CE 不能被安排为规则的第二和第四个 CE，甚至不能被安排为规则的第一个、第三个 CE，因为这样做会留出空隙。对 logical CE 的这一限制是它的潜在实现的原因。同样可以通过在 logical CE 中使用 not CE 来使事实依赖于其他事实的不存在性。在 logical CE 中，可以使用 exists CE、forall CE 或其他

CE 的组合形成的更为复杂的条件。logical CE 在其他方面的情况与 and CE 一样，但不能在多组事实之间建立依赖关系。

为了修改 noxious-fumes-present 规则，使 use-oxygen-masks 事实仅依赖于 noxious-fumes-present 事实，则需要重新排列这些模式如下：

```
(defrule noxious-fumes-present
   (logical (noxious-fumes-present))
   (emergency (type fire))
   =>
   (assert (use-oxygen-masks)))
```

有了上述修改后的规则，则当 emergency 事实撤销后，use-oxygen-masks 事实将不会自动地被撤销（这是一个更安全的方法，因为即使火灾熄灭后仍有可能存在有毒气体）。

正常地，声明一个已存在于事实表中的事实并无什么影响。但是，一个逻辑上依赖于多个源推导出来的事实不会自动撤销，除非所有源的逻辑支持都被除去。例如，假设增加另一条规则，指示当正在使用气体灭火器时应该戴上氧气面具：

```
(defrule gas-extinguishers-in-use
   (logical (gas-extinguishers-in-use))
   (emergency (type fire))
   =>
   (assert (use-oxygen-masks)))
```

运行该系统将引起两个不同的规则根据不同的原因去声明同一事实。

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS> (watch rules)↵
CLIPS> (assert (emergency (type fire))
               (noxious-fumes-present)
               (gas-extinguishers-in-use))↵
==> f-1      (emergency (type fire))
==> f-2      (noxious-fumes-present)
==> f-3      (gas-extinguishers-in-use)
<Fact-3>
CLIPS> (run)↵
FIRE    1 gas-extinguishers-in-use: f-3,f-1
==> f-4       (use-oxygen-masks)
FIRE  2 noxious-fumes-present: f-1,f-2
CLIPS>
```

撤销 noxious-fumes-present 事实不足以使 use-oxygen-masks 事实自动撤销，因为对将要使用的氧气面罩有另一个逻辑支持。在撤销 use-oxygen-masks 事实之前，还必须撤销 gas-extinguishers-in-use 事实。

```
CLIPS> (retract 2)↵
<== f-2      (noxious-fumes-present)
CLIPS> (retract 3)↵
<== f-3      (gas-extinguishers-in-use)
<== f-4      (use-oxygen-masks)
CLIPS>
```

在顶层提示下、或从一个在 LHS 中没有任何逻辑 CE 的规则 RHS 上被声明的事实会无条件得到支持。无条件受支持的事实将永不会因为另一个事实的撤销而自动撤销。一旦接收到无条件支持，对某个事实的以前所有逻辑支持都会被丢弃。

为了观察与事实相联系的依赖关系，CLIPS 提供了两条命令。命令格式如下：

```
(dependents <fact-index-or-address>)

(dependencies <fact-index-or-address>)
```

对最后一个例子，在 noxious-fumes-present 和 gas-extinguishers-in-use 两个事实被撤销之前，这些命令将产生下面的输出：

```
CLIPS> (facts)↵
f-0     (initial-fact)
```

```
f-1     (emergency (type fire))
f-2     (noxious-fumes-present)
f-3     (gas-extinguishers-in-use)
f-4     (use-oxygen-masks)
For a total of 5 facts.
CLIPS> (dependents 1)↵
None
CLIPS> (dependents 2)↵
f-4
CLIPS> (dependents 3)↵
f-4
CLIPS> (dependents 4)↵
None
CLIPS> (dependencies 1)↵
None
CLIPS> (dependencies 2)↵
None
CLIPS> (dependencies 3)↵
None
CLIPS> (dependencies 4)↵
f-2
f-3
CLIPS>
```

8.19 小结

本章介绍了字段约束的概念。字段约束允许对一个特定的字段使用多个约束组合或否定。not 字段约束用于阻止匹配某些特定值。and 字段约束用于确保一连串的匹配条件都为真。or 字段约束用于确保一连串匹配条件中至少有一个为真。

函数用在 CLIPS 顶层命令循环中，或用在规则的 LHS 或 RHS 上。许多函数，例如某些算术函数，可以具有数量不固定的选项。函数调用可以嵌套在其他函数调用中。bind 命令允许变量在规则的 RHS 上约束。

CLIPS 提供了若干个 I/O 函数。open 和 close 函数用于打开和关闭文件。打开的文件与一个逻辑名相联系。逻辑名可用于大多数函数中，以实现多种物理设备上的输入和输出。printout 和 read 函数使用逻辑名。printout 函数可以输出到终端和文件。read 函数可以从键盘和文件中输入。format 和 readline 函数同样可使用逻辑名。format 函数允许对输出外观有更多的控制。readline 函数可以用于读入整行数据。explode $ 函数用于把一个字符串转换成一个多字段值。

本章还介绍了执行控制流的若干概念。read 函数用来展示如何使用被撤销然后又被声明的控制事实来建立一个对输入的简单控制循环。Test CE 和谓词函数一起可用于规则的左部以提供更强大的模式匹配能力。此外，test CE 可用来维护控制循环。谓词字段约束允许谓词测试直接置于模式中。相等字段约束用来比较字段和函数返回值。Sticks 程序展示了这些控制技巧。

除了 test CE 以外，还有其他的 CE。or CE 用来将几条规则表达为单一的一条规则。not CE 允许某事实不在事实表上时仍能进行模式匹配。and CE 将 CE 形成组，并与 or 和 not 一起使用，它可以任意嵌套以表达需要符合某条规则的复杂条件。exists CE 用来确定满足一个 CE 或一组 CE 的事实中至少有一组事实是否存在。forall CE 用来确定某个 CE 集是否满足另一 CE 的每一次出现。logical CE 提供了正确性维护机制，可以使事实的存在依赖于其他事实的存在。

习题

8.1 假设下列事实的自定义模板描述一棵家族树，

```
(deftemplate father-of (slot father) (slot child))
(deftemplate mother-of (slot mother) (slot child))
(deftemplate male (slot person))
(deftemplate female (slot person))
(deftemplate wife-of (slot wife) (slot husband))
(deftemplate husband-of (slot husband) (slot wife))
```

写出能推出下列关系的规则。描述解决问题所用的自定义模板。

(a) Uncle，aunt

(b) Cousin

(c) Grandparent

(d) Grandfather，grandmother

(e) Sister，brother

(f) Ancestor

8.2 一家工厂有 10 个传感器，编号从 1 到 10。每个传感器有“良好”或“不良”两种状态。建立一个描述传感器的自定义模板，写出一条或更多的规则，使在三个或更多的传感器处于不良状态时打印警告信息。在下列情况下测试你的规则：3 号和 5 号传感器不良；2、8 和 9 号不良；1、3、5 和 10 号不良。如何防止警告信息显示多次呢?

8.3 基于为习题 3.5 所设计的 IF…THEN 规则建立一个 CLIPS 程序。程序要求询问旅行者的费用支付方法及旅游兴趣，要求在这两个输入信息的基础上输出潜在旅程。

8.4 假如有使用下列自定义模板来描述形状的一系列事实：

```
(deftemplate square
   (slot id) (slot side-length))
(deftemplate rectangle
   (slot id) (slot width) (slot height))
(deftemplate circle
   (slot id) (slot radius))
```

编写一条或更多条规则，计算下列各项的和：

(a) 图形的面积；

(b) 图形的周长。

用下列自定义事实测试规则的输出结果：

```
(deffacts test-8-8
   (square (id A) (side-length 3))
   (square (id B) (side-length 5))
   (rectangle (id C) (width 5) (height 7))
   (circle (id D) (radius 2))
   (circle (id E) (radius 6)))
```

8.5 已知使用下列自定义模板的某人的姓名、眼睛颜色、头发颜色和国籍：

```
(deftemplate person (slot name)
                    (slot eye-color)
                    (slot hair-color)
                    (slot nationality))
```

写出一条规则找出：

(a) 任何有蓝色或绿色眼睛、棕色头发的法国人。

(b) 任何不是蓝眼睛、不是黑头发且头发和眼睛颜色不同的人。

(c) 两个人；第一个是眼睛为棕色或蓝色、头发不是金色的德国人；第二个是眼睛为绿色、头发颜色与第一个相同且国籍不限的人。如果第一个人的头发是棕色，则第二个人的眼睛可以是棕色。

8.6 把下列中缀表达式转换成前缀表达式：

(a) `(3 + 4) * (5 + 6) + 7`

(b) `(5 * (5 + 6 + 7)) - ((3 * (4 / 9) + 2) / 8)`

(c) `6 - 9 * 8 / 3 + 4 - (8 - 2 - 3) * 6 / 7`

8.7 以下是关于一个棒球队的信息。Andy 不喜欢做接球手。Ed 的姐姐作为第二垒手。中场外野手比右场外野手高。Harry 和第三垒手住在同一座楼里。Paul 和 Allen 打牌每人各赢投球手 20 美元。

Ed和外野手们在空闲时打扑克。投球手妻子是第三垒手的姐姐。除了Allen、Harry和Andy外，投捕手（包括投球手和接球手）和内野手都比Sam矮。Paul、Andy和游击手每人在赛马场输了50美元。Paul、Harry、Bill和接球手在桌球上输给了第二垒手。Sam正在办离婚。接球手和第三垒手每人有两个孩子。Ed、Paul、Jerry、右场外野手和中场外野手都是单身汉，其他人都结了婚。游击手、第三垒手和Bill赌拳各输了100美元。有一个外野手不是Mike就是Andy。Jerry比Bill高，Mike比Bill矮。他们三个都比第三垒手重。Sam、接球手和第三垒手是左撇子。Ed、Sam和游击手一起读高中。试编写一个CLIPS程序确定他们各自的打球位置。

8.8 把图3.3中示出的判定树转换成一系列CLIPS规则。用下列自定义模板创建与事实匹配的模式：

```
(deftemplate question
   (slot query-string)
   (slot answer))
```

例如，若由树的根结点代表的问题答案是“否定”，则说明该信息的事实应该是：

```
(question (query-string "Is it very big?")
          (answer no))
```

在规则的RHS上应用printout和read函数回答图中提出的问题，以用户的响应来确定question事实。

8.9 写出一个CLIPS程序，不用算术函数实现两个二进制数加法。用以下自定义模板来表示二进制数。

```
(deftemplate binary-#
   (multislot name)
   (multislot digits))
```

给出两个已命名二进制数相加的事实，要求程序创建一个新的命名二进制数存储该和数。例如，事实：

```
(binary-# (name A) (digits 1 0 1 1 1))
(binary-# (name B) (digits 1 1 1 0))
(add-binary-#s (name-1 A) (name-2 B))
```

要求使以下事实：

```
(binary-# (name ( A + B )) (digits 1 0 0 1 0 1))
```

添加到事实表中。

8.10 写出一个CLIPS程序，提示输入需要输血的病人血型和献血者的血型。要求程序根据血型决定输血是否可以进行。O型血只能输入O型血；A型血可以输入A型或O型血；B型血可以输入B型或O型血；AB型血可以输入AB型、A型、B型或O型血。

8.11 写出一个CLIPS程序，提供关于如何烹调特定的牛肉的信息、或提供能以特定方式烹调的牛肉的信息。要求程序首先提示选择牛肉还是选择烹调的方式，然后再提示相应的选择。运用以下指导原则决定适宜的肉或烹调方法：臀部烤肉（rump roast）应该蒸炖或烘烤；腰排（sirloin steak）应该烤熟、用盘烤或用盘炸；T-骨牛排应该烤熟、用盘炸或蒸炖；肋骨应该烘烤；背部肉应该烘烤、烤熟、用盘炸或蒸炖；侧排应该蒸炖；圆排应该蒸炖。

8.12 修改习题7.14中的程序，使输入由用户回答的一系列关于灌木的必要特点的问题而决定。程序的输出应该与原来习题中的相同。

8.13 细菌可依据不同的特征分类。这些特征包括它们的基本形状（球状、棒状、螺状或丝状）、实验室的染色实验结果（正、负或无）和它们的生存是否需要氧气（好氧或厌氧）。编写一个程序，根据下面表中的信息识别细菌类别。此程序要询问用户关于细菌形状、染色实验结果和需氧否的信息。用户可指定任何输入信息为未知。程序的输出结果应该能够根据用户所提供的信息给出所有可能的细菌类型。

类　型	形　状	染色实验	需氧性
Actinomycete	棒状或丝状	正	好氧
Coccoid	球状	正	好氧或厌氧
Coryneform	棒状	正	好氧
Endospore-forming	棒状	正或负	好氧或厌氧
Enteric	棒状	负	好氧
Gliding	棒状	负	好氧
Mycobacterium	球状	否	好氧
Mycoplasma	球状	否	好氧
Pseudomonad	棒状	负	好氧
Rickettsia	球状或棒状	负	好氧
Sheathed	丝状	负	好氧
Spirillum	螺旋状	负	好氧
Spirochete	螺旋状	负	厌氧
Vibrio	棒状	负	好氧

8.14 尖端电子学制造出一种名为 Thingamabob 2000 的设备。这种设备有 5 种不同的模型，分别由底座区分。每个底座提供一些对可选小装置的支架，它可产生一定量的动力。下表总结了底座的各种属性：

底　座	提供的支架数	提供的动力	价格（$）
C100	1	4	2000.00
C200	2	5	2500.00
C300	3	7	3000.00
C400	2	8	3000.00
C500	4	9	3500.00

每个可以安装在底座上的小装置需要一定量的动力运行。下表总结了小装置的各种属性：

小装置	所用动力	价格（$）
Zaptron	2	100.00
Yatmizer	6	800.00
Phenerator	1	300.00
Malcifier	3	200.00
Zeta-shield	4	150.00
Warnosynchronizer	2	50.00
Dynoseparator	3	400.00

已知挑选出的底座和小装置的输入事实，编写一个程序，生成已用的小装置数的事实、各小装置所需的总动力以及所选底座和全部小装置的总价格。

8.15 根据习题 7.13 给出的宝石表格，编写区分下列宝石的规则：钻石、刚玉、金绿宝石、尖晶石、石英和电气石。要求包含向用户查询宝石的硬度、密度和颜色的规则，并提示用户一定要以表格所列宝石的颜色之一作为回答。

8.16 在 Sticks 程序中加入一些规则，使它能向使用者询问是否在游戏结束之后再玩一次游戏。

8.17 修改 Sticks 程序，除了能使人与计算机进行游戏对战外，还可以使得两人能相互对战。

8.18 用 and 和 or CE 重新把下列规则写成一条规则：

```
(defrule rule-1
   (fact-a)
   (fact-d)
   =>)

(defrule rule-2
   (fact-b)
   (fact-c)
   (fact-e)
   (fact-f)
   =>)

(defrule rule-3
   (fact-a)
   (fact-e)
   (fact-f)
   =>)

(defrule rule-4
   (fact-b)
   (fact-c)
   (fact-d)
   =>)
```

8.19 用 and 和 or CE 为图 3.10 的与或树编写一个程序，并对所有分枝进行测试。

8.20 判断变量 x 是否为下列每一条规则正确引用，解释你的答案。

(a)
```
(defrule example-1
      (not (fact ?x))
      (test (> ?x 4))
      =>)
```

(b)
```
(defrule example-2
      (not (fact ?x&:(> ?x 4)))
      =>)
```

(c)
```
(defrule example-3
      (not (fact ?x))
      (fact ?y&:(> ?y ?x)))
      =>)
```

(d)
```
(defrule example-4
      (not (fact ?x))
      (fact ?x&:(> ?x 4)))
      =>)
```

8.21 重写第 7.23 节的“块世界”程序，使得它能重新排列各个块，使得从原始的堆栈状态排列为任何目标状态。例如，如果块的原始状态为：

```
(stack A B C)
(stack D E F)
```

其中一个可能的目标状态为：

```
(stack D C B)
(stack A)
(stack F E)
```

8.22 编写一个 CLIPS 程序，使其能询问用户各颜色值，然后，打印所有具有含指定颜色的国旗的所有国家列表。下表列出了许多国家的国旗颜色：

国　别	国旗颜色
United States 美国	Red, white, and blue
Belgium 比利时	Black, yellow, and red
Poland 波兰	White and red
Monaco 摩纳哥	White and red
Sweden 瑞典	Yellow and blue
Panama 巴拿马	Red, white, and blue

（续）

国　别	国旗颜色
Jamaica 牙买加	Black, yellow, and green
Colombia 哥伦比亚	Yellow, blue, and red
Italy 意大利	Green, white, and red
Ireland 爱尔兰	Green, white, and orange
Greece 希腊	Blue and white
Botswana 博茨瓦纳	Blue, white, and black

注：Red 红，Orange 橙，Yellow 黄，Green 绿，Blue 蓝，Black 黑，White 白

8.23 给定下面自定义模板描述一个集合（set）：

```
(deftemplate set
   (multislot name)
   (multislot members))
```

试写出一条或多条规则，

(a) 求出两个指定集合的并，已知这两个集合的事实由下列自定义模板给出：

```
(deftemplate union
   (multislot set-1-name)
   (multislot set-2-name))
```

(b) 求出两个指定集合的交，已知这两个集合的事实由下列自定义模板给出：

```
(deftemplate intersection
   (multislot set-1-name)
   (multislot set-2-name))
```

注意，计算并和交时，不允许出现重复的元素。(a) 和 (b) 的最后结果应该是一个新的集合事实，它包含这两个指定集合的并或交，而且，操作完成后，应该撤销这两个并和交事实。

8.24 写出一组规则，按语态和型将三段论归类。例如，三段论：

```
No M is P
Some M is not S
∴ Some S is P
```

属于 EOI-3 类型。规则的输入应该是一个事实，描述大小前提和结论。输出是一条语态和型的打印语句。

8.25 编写一个程序，读取含有人名和年龄的数据文件，并创建一个按年龄升序排列的新文件。该程序应该提示输入文件和输出文件。例如，输入文件

```
Linda A. Martin 43
Phyllis Sebesta 40
Robert Delwood 38
Jack Kennedy 39
Glen Steele 37
```

应该创建的输出文件是：

```
Glen Steele 37
Robert Delwood 38
Jack Kennedy 39
Phyllis Sebesta 40
Linda A. Martin 43
```

8.26 编写一个程序，利用点数法计算桥牌手中 13 张牌的点数。A 为 4 点、K 为 3 点、Q 为 2 点、J 为 1 点。缺一种花色（某一花色一张牌都没有）为 3 点、某花色只有单张为 2 点、某花色只有双张为 1 点。

8.27 编写一个程序，指出某人吞入毒物后要采取什么措施。此程序知道下列毒物：酸（如去锈剂、碘酒），碱（如氨水、漂白剂），以及石油产品（如汽油、松节油）。其他所有毒物归为 other 类。

中毒时，应呼叫医师或中毒控制中心。对于酸、碱和其他类型的毒物（但不是石油产品），应该让病人喝水或牛奶之类的液体以稀释毒药。对于其他毒物，要驱呕。但对于酸、碱或石油产品不能驱呕。如果病人神志不清或惊厥，则不要喝水类液体，也不要驱呕。

8.28 编写一个程序，计算给定两点形成的直线斜率。程序应该检查确保所提供的两点包含的是数字，且一点不能规定两次。垂线的斜率为无穷大。

8.29 不等边三角形的三边不相等。等腰三角形有两条边相等。等边三角形的三条边相等。编写程序，当给定形成三角形的三个点时，判断该三角形的类型。程序应该考虑可能的舍入误差（假如两边长之差小于0.00001，则认为这两条边相等）。用下列三角形测试你的程序。

(a) 给定3点：(0, 0)，(2, 4) 和 (6, 0)。

(b) 给定3点：(1, 2)，(4, 5) 和 (7, 2)。

(c) 给定3点：(0, 0)，(3, 5.196152) 和 (6, 0)。

8.30 编写一个程序求解 Hanoi 塔问题：将一个测标处的一组环移到另一个测标处，移动过程中不能将大环压在小环之上。允许你使用3个测标，环的数量应该作为程序的输入。初始状态时，所有的环在第一个测标处，这些环从下到上按从大到小的顺序堆放。初始目标是将所有的环从第一个测标移至第三个测标。

8.31 编写程序确定使下列加密算法正确的字母值。每一个 H、O、C、U、S、P、R、E 和 T 惟一对应于0～9中的一个数字。且：

```
  HOCUS
+ POCUS
= PRESTO
```

8.32 编写一个程序，对投资互动基金给出意见。程序的输出应该指出投资的比例：有固定收入的存款，指主要投资于债券和优先股的资金；和股票资金，指高风险高可能收益的资金。根据投资者对各种问题的回答，通过对投资者愿意承担的风险大小记分来确定投资比例。如果投资者不超过29岁，则分数加4；若在30～39之间，则加3；若在40～49之间，则加2；若在50～59之间，则加1；若在60岁和60岁以上，则加0。若投资者还有0～9年才退休，则加0；还有10～14年，则加1；还有15～19年，则加2；还有20～24年，则加3；还有25年或25年以上，则加4。若投资者愿意经历的损失只有5%或更少，则分数加0；若愿意在6%～10%之间，则加1；若愿意在11%～15%之间，则加2；若愿意在16%或更多，则加3。如果投资者对投资很有经验、对股票市场很了解，则分数加4；若经验和了解程度一般，则加2；若没有这方面的经验和知识，则加0。假如投资者对可能的高回报愿意承担高风险，则分数加4；若只愿意承担部分风险，则加2；若愿意承担很小的风险，则加0。如果投资者认为按他/她的目前收入和财产将达到他/她的退休目标，则分数加4；若可能会达到目标，则加2；若不大可能达到目标，则加0。若最后的分数为20分以上，则100%的投资都应该在股票资金上；若为16～20分，则80%在股票上，20%在有固定收入的存款上；若为11～15分，则60%在股票上，40%在有固定收入的存款上；若为6～10分，则40%在股票上，60%在有固定收入的存款上；若为0～5分，则20%在股票上，80%在有固定收入的存款上。

8.33 修改习题7.12中的程序，使关于星（stars）的信息用事实表示。程序的输出应该有相同的顺序：先列出全部有指定光谱级的星星，再是全部有指定光度的星星，最后是既有指定的光谱级又有指定的光度以及指定的从地球到这些星星的光年数距离的星星。

第9章　模块化设计、执行控制和规则效率

9.1　概述

本章主要介绍一些CLIPS特性，这些特性有利于专家系统的开发和维护。自定义模板属性允许其槽值的增强值约束。当装入一条规则时，自定义模板约束能检测出阻止规则的LHS不匹配的语义错误。本章利用优先级，一种确定规则优先次序的方法和代表控制知识流程的事实，展示控制CLIPS程序运行的技术。除此之外，本章也探讨了自定义模块（defmodule）结构，这种自定义模块结构允许知识库分块，以提供一个更明确的能控制系统执行的方法。本章还介绍了许多提高基于规则的专家系统效率的技巧，这里专家系统使用的是Rete模式匹配算法。在解释Rete算法之前，将讨论需要一个有效的模式匹配算法的原因。同样，也将讨论一些更加有效地编写规则的技巧。

9.2　自定义模板属性

当要定义自定义模板槽时，CLIPS提供了许多种能被规定的槽属性，这些属性通过提供强大的类型功能和约束检查，达到协助开发和维护专家系统的目的。可以定义槽中的允许类型和值。对于数字值，数的允许范围可以规定。多槽（Multislots）也可以规定它们所能包含字段的最小和最大值。最后，当assert命令中没有槽的具体说明时，默认属性可以为其提供默认槽值。

类型属性

类型属性（type attribute）定义了槽中的数据类型。说明类型属性的一般格式为：（type ＜type-specification＞），其中＜type-specification＞可以是？VARIABLE，或者是一个或几个符号：SYMBOL、STRING、LEXEME、INTEGER、FLOAT、NUMBER、INSTANCE-NAME、INSTANCE-ADDRESS、INSTANCE、FACT-ADDRESS、EXTERNAL-ADDRESS。如果使用？VARIABLE，则该槽允许使用任何数据类型（这也是所有槽的默认属性）。如果使用一种或几种符号类型说明，则该槽就限于这几种类型之一。类型说明LEXEME代表SYMBOL和STRING这两种类型。同样，类型说明NUMBER代表INTEGER和FLOAT这两种类型说明。类型说明INSTANCE代表INSTANCE-NAME和INSTANCE-ADDRESS这两种类型。

下面的自定义模板person中，限制存于name槽的值为字符，并限制age槽的值为整数：

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER)))
```

一旦已经定义自定义模板，CLIPS就会自动对任何槽的属性实施限制。例如，假定将符号four赋值给age槽，而不是用4去赋值，就会出现以下错误：

```
CLIPS> (assert (person (name Fred Smith)
                       (age four)))↵

[CSTRNCHK1] A literal slot value found in the
assert command does not match the allowed types
for slot age.
CLIPS>
```

CLIPS也会检查约束于规则LHS和RHS中的变量的一致性。例如，让我们假定有一条规则：不论他（或她）何时生日，都要更新他的age槽。指出一个人已过生日的控制事实的自定义模板如下：

```
(deftemplate had-a-birthday
   (slot name (type STRING)))
```

在以上所举的两个自定义模板中的name槽的数据类型是不一致的。试图直接比较这两个name槽会产

生错误，如下所示：

```
CLIPS>
(defrule update-birthday
   ?f1 <- (had-a-birthday (name ?name))
   ?f2 <- (person (name ?name) (age ?age))
   =>
   (retract ?f1)
   (modify ?f2 (age (+ ?age 1))))↵

[RULECSTR1] Variable ?name in CE #2 slot name has
constraint conflicts which make the pattern
unmatchable.

ERROR:
(defrule MAIN::update-birthday
   ?f1 <- (had-a-birthday (name ?name))
   ?f2 <- (person (name ?name) (age ?age))
   =>
   (retract ?f1)
   (modify ?f2 (age (+ ?age 1))))
CLIPS>
```

事实 had-a-birthday 的 name 槽必须是字符串，而事实 person 的 name 槽必须是一个字符。不可能用变量? name 来同时满足这两种不同类型的约束条件，因而，规则的 LHS 永不会满足。

静态和动态约束检查

CLIPS 提供两级约束检查。第一级是**静态约束检查**（static constraint checking），它是通过 CLIPS 解释一个表达或结构的意义来默认执行的。这种检查通过使用类型属性在上面的违反约束的例子中得到说明。静态约束检查可以通过调用函数 set-static-constraint-checking 并将符号 FALSE 传递给它来使之失效。相反，使用符号 TRUE 调用该函数会激活静态约束检查。该函数的返回值是前一个状态值（如果前一次关闭了检查功能，则其值为符号 FALSE；否则，为符号 TRUE）。你可以知道静态约束检查的当前状态，方法是调用 get-static-constraint-checking 函数（当静态约束检查在起作用时，它返回符号 TRUE；否则，返回符号 FALSE）。

在分析阶段，不是总能知道所有的约束错误。举个例子：在下面的 create-person 规则中，变量? age 和? name 可约束到一些非法值上：

```
(defrule create-person
   =>
   (printout t "What is your name? ")
   (bind ?name (explode$ (readline)))
   (printout t "What is your age? ")
   (bind ?age (read))
   (assert (person (name ?name) (age ?age))))
```

readline 函数用于输入一个人的完整姓名，它是字符型的。然后 explode $ 函数把它变成一个可放在 name 槽中的多字段值。read 函数用于输入一个人的年龄，该值存放于 age 槽中。对于这两个待输入值，可能会接受到无效的值。例如：符号 four 可能输入为某人的年龄，如下显示：

```
CLIPS> (reset)↵
CLIPS> (run)↵
What is your name? Fred Smith↵
What is your age? four↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (person (name Fred Smith) (age four))
For a total of 2 facts.
CLIPS>
```

请注意，关于 Fred Smith 的同一个事实 person，此前引起了约束冲突而没有添加到事实表中去，现在却添加到事实表中去了。这是因为 CLIPS 的第二级约束检查——**动态约束检查**（dynamic constraint checking），在默认状态是关闭的。动态约束性检查是当事实确实被声明时才对该事实执行检查

的，因而，能检查出在语法检查阶段所不能检查出来的错误。

动态约束检查可以通过函数 set-dynamic-constraint-checking 来激活或使它失去作用，也可以用功能函数 get-dynamic-constraint-checking function 来确定动态约束检查的当前状态。在已激活动态约束检查的情况下，以下说明如何解决约束冲突。在下面的例子中，Fred Smith 的事实 person 仍然被添加到事实表中，但检测到了约束冲突，并终止了规则的执行。

```
CLIPS> (set-dynamic-constraint-checking TRUE)↵
FALSE
CLIPS> (reset)↵
CLIPS> (run)↵
What is your name? Fred Smith↵
What is your age? four↵

[CSTRNCHK1] Slot value (Fred Smith) found in fact
f-1 does not match the allowed types for slot age.
[PRCCODE4] Execution halted during the actions of
defrule create-person.
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (person (name Fred Smith) (age four))
For a total of 2 facts.
CLIPS>
```

允许值属性（The Allowed Value Attributes）

CLIPS除了利用类型属性来限制能被允许的类型之外，还允许你为特定的类型规定一系列的允许值。例如，如果一个 gender 槽被添加到自定义模板 person 中，那么，该槽允许的符号就被限制为 male 或者 female：

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER))
   (slot gender (type SYMBOL)
                (allowed-symbols male female)))
```

CLIPS 总共提供了 8 种不同的允许值属性：allowed-symbols、allowed-strings、allowed-lexemes、allowed-integers、allowed-floats、allowed-numbers，allowed-instance-names 和 allowed-values。每一种属性后面应紧跟着？VARIABLE（这意味着对于一个已被确定的数据类型任何值都是合法的），或者紧跟着带 allowed-前缀的一系列类型值。例如，属性 allowed-lexemes 后应紧跟着？VARIABLE 或者紧跟着一系列符号或字符串。槽的允许值默认属性为（allowed-values ？VARIABLE）。

注意，允许值属性并没有限制槽的允许类型。例如，（allowed-symbols male female）并不会限制槽 gender 的类型为一个符号。它仅仅是指出，当槽 gender 的值为一个符号时，它只能是以下两个取值之一：或者是 male，或者是 female。如果删除属性（type SYMBOL），则任何字符串、整数或者浮点数都是槽 gender 的合法取值。

允许值属性可用于完全将槽的允许值集合限制在一个规定的列表中。例如，按如下方式改变自定义模板 person 将有效地限制槽 gender 的允许类型只为符号：

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER))
   (slot gender (allowed-values male female)))
```

范围属性

范围属性（range attribute）允许对槽的最小值和最大值进行说明。其一般格式为（range ＜lower-limit＞ ，＜upper-limit＞），其中，＜lower-limit＞和＜upper-limit＞的取值可以是？VARIABLE 或者是一个数值。＜lower-limit＞项指槽所允许的最小值，＜upper-limit＞项指槽所允许的最大值。？VARIABLE 指出没有最小或最大值的情况（取决于？VARIABLE 是放在＜lower-limit＞还是放在＜upper-lim-

it>的位置)。例如，自定义模板 person 中槽 age 可以修改为不允许负数置于 age 槽中：

```
(deftemplate person
   (multislot name (type SYMBOL))
   (slot age (type INTEGER) (range 0 ?VARIABLE)))
```

如果我们愿意假定没有人能活得超过 125 岁，则范围属性可以改为（range 0 125)。正如允许值属性一样，范围属性并没有限制槽值的类型为数字型。它仅仅是指出：当槽值是数字时，槽的允许数值只限制在指定的范围。槽的范围属性的默认值为（range ? VARIABLE ? VARIABLE)。

基数性属性

基数性属性（cardinality attribute）允许对存于一个多槽（multislot）中的值规定最小值和最大值。其一般格式为（cardinality <lower-limit> <upper-limit>)，其中，<lower-limit>和<upper-limit>可以是? VARIABLE，也可以是一个正整数。<lower-limit>指出该槽所允许的最小值。<upper-limit>指出该槽所允许的最大值。? VARIABLE 表示没有最小值或没有最大值（这取决于? VARIABLE 是放在<lower-limit>还是放在<upper-limit>位置)。对于一个多槽，其默认属性为（cardinality ? VARIABLE ? VARIABLE)。下面的自定义模板可用于代表公司的排球队，排球队必须有 6 个队员，还可以有两个替补队员。请注意，类型、允许值和范围属性可用于这个多槽中的每一个值。

```
(deftemplate volleyball-team
   (slot name (type STRING))
   (multislot players (type STRING)
                      (cardinality 6 6))
   (multislot alternates (type STRING)
                         (cardinality 0 2)))
```

默认属性

在前面几章中，已声明的每一个自定义模板事实中，对于每个槽总有一个明确的值。如果在 assert 命令中没有明确规定取值，则自动有一个规定的值存于槽中，这通常是非常方便的。**默认属性**（default attribute）就是这样一个规定的默认值。其一般格式为（default <default-specification>)。其中，<default-specification>可以是? DERIVE、? NONE、一个表达式（对于一个单字段槽)、0 或者是更多的表达式（对于一个多字段槽)。

如果用? DERIVE 来规定默认属性，就会得到一个能满足所有槽属性的值。如果没有对槽的默认属性进行规定，它就会假定其默认属性为（default ? DERIVE)。对于一个单字段槽而言，这意味着将选出一个值来满足槽的类型、范围和允许值属性。对于一个多字段槽而言，产生的默认值将是一串相同的值，这些值是该槽的最小允许基值（默认是 0)。如果有一个或多个值被包含在一个多字段槽的默认值中，那么，每一个值都会满足该槽的类型、范围和允许值属性。以下是一个例子：

```
CLIPS> (clear)↵
CLIPS>
(deftemplate example
   (slot a)
   (slot b (type INTEGER))
   (slot c (allowed-values red green blue))
   (multislot d)
   (multislot e (cardinality 2 2)
                (type FLOAT)
                (range 3.5 10.0)))↵
CLIPS> (assert (example))↵
<Fact-0>
CLIPS> (facts)↵
f-0     (example (a nil)
                 (b 0)
                 (c red)
                 (d)
                 (e 3.5 3.5))
For a total of 1 fact.
CLIPS>
```

CILPS仅仅保证槽的导出（derived）默认属性满足槽的约束属性。也就是说，你的程序不能依赖于置于槽中的特定导出值（例如，上例中槽 a 中的符号 nil 和槽 b 中的整数 0）。如果你的程序依靠于一个特定的默认值，那么，你应该使用一个有默认属性的表达式（很快就会解释默认属性）。

如果在默认属性中规定的是? NONE，则当事实被声明时，必须要给该槽提供一个值。也就是说，没有默认值。例如：

```
CLIPS> (clear)↵
CLIPS>
(deftemplate example
   (slot a)
   (slot b (default ?NONE)))↵
CLIPS> (assert (example))↵
[TMPLTRHS1] Slot b requires a value because of its
(default ?NONE) attribute.
CLIPS> (assert (example (b 1)))↵
<Fact-0>
CLIPS> (facts)↵
f-0     (example (a nil) (b 1))
For a total of 1 fact.
CLIPS>
```

如果一个或多个表达式使用默认属性，则在对该槽作语法检查时，会求出表达式的值。而且，若在 assert 命令中没有规定槽值，则其值会存于该槽中。对于一个单字段槽而言，其默认属性必须包含一个确切的表达式。如果在一个多字段槽的默认属性中没有规定表达式，则一个长度为 0 的多字段会被用于该默认值。否则，所有表达式的返回值就会成为一组形成一个多字段值。以下是使用带默认属性的表达式的例子：

```
CLIPS> (clear)↵
CLIPS>
(deftemplate example
  (slot a (default 3))
  (slot b (default (+ 3 4)))
  (multislot c (default a b c))
  (multislot d (default (+ 1 2) (+ 3 4))))↵
CLIPS> (assert (example))↵
<Fact-0>
CLIPS> (facts)↵
f-0     (example (a 3) (b 7) (c a b c) (d 3 7))
For a total of 1 fact.
CLIPS>
```

默认动态属性

当使用默认属性时，在对槽定义作语法检查时，槽的默认值就被确定了。当然，也可以在声明将要使用默认值的事实时，产生此默认值。这就需要用到默认-动态属性（default-dynamic attribute）。当利用默认-动态属性的槽值没有在 assert 命令中规定时，就会求出在默认-动态属性中规定的表达式的值，并将其用于该槽值。

举个例子来说，让我们考虑这样一个问题：在经过一定的时间后删除事实的问题。首先，我们需要某种方法知道事实是什么时候声明的。CLIPS 中提供的 time 函数可用于标记事实创建的时间，还会返回经过的秒数，这是一个依赖于系统的参考时间。但就其本身而言，时间函数返回的时间值并没有什么意义。它仅仅是在和时间函数返回的其他值作比较时才有用。下面举个自定义模板的例子。它包含一个 creation-time 槽和一个 value 槽，分别用于保存事实创建的时间和与该事实相联系的值。

```
(deftemplate data
   (slot creation-time (default-dynamic (time)))
   (slot value))
```

当每一个 data 事实被创立、而没有规定 creation-time 槽时，time 函数就会被调用、其时间值会被保存于 creation-time 槽中。

```
CLIPS> (watch facts)↵
CLIPS> (assert (data (value 3)))↵
==> f-0     (data (creation-time 12002.45)
                  (value 3))
<Fact-0>
CLIPS> (assert (data (value b)))↵
==> f-1     (data (creation-time 12010.25)
                  (value b))
<Fact-1>
CLIPS> (assert (data (value c)))↵
==> f-2     (data (creation-time 12018.65)
                  (value c))
<Fact-2>
CLIPS>
```

假定事实 creation-time 已经被声明，并被其他规则更新以包含当前系统时间，以下规则将撤销还不到 1 分钟之前才被声明的事实 data：

```
(defrule retract-data-facts-after-one-minute
   ?f <- (data (creation-time ?t1))
   (current-time ?t2)
   (test (> (- ?t2 ?t1) 60))
   =>
   (retract ?f))
```

注意，将规则 retract-data-facts-after-one-minute 改变为下列形式不会产生相同的结果。

```
(defrule retract-data-facts-after-one-minute
   ?f <- (data (creation-time ?t1))
   (test (> (- (time) ?t1) 60))
   =>
   (retract ?f))
```

只在第一模式与 data 事实匹配时，在 test CE 中的 time 函数才会被检查。由于返回值和 creation-time 槽的值大致有相同的时间，因此，该规则不会得到满足。CLIPS 不会连续重新检查 test CE 以决定它们是否得到一个不同的值；只有当前面的 CE 发生变化时，它们才会被检查。这就是为什么在规则的原始版本中 current-time 事实必须更新，test CE 才会被周期性地重新检查的原因。

冲突槽属性

CLIPS 不允许你规定槽的冲突属性。例如，槽的默认值必须满足槽的类型、allowed-……、范围和基数性属性。如果规定 allowed-…… 属性，则与属性相关联的类型必须满足槽的类型属性。allowed-numbers、allowed-integers 和 allowed-floats 属性可以不与范围属性一起使用。

9.3 优先级

至此，控制事实一直用来间接控制程序的执行。CLIPS 提供 2 种具体的技术来控制规则的执行：优先级（salience）和模块（modules）。用模块来控制规则的执行将在本章稍后讨论。关键词优先级的使用允许规则的优先级被明确地规定。通常，议程像堆栈一样起作用。也就是说，在议程中最近的激活，将会最先触发。不管规则何时被增加，优先级都允许更为重要的规则停留在议程的顶部。在议程中，较低的优先级规则被推到更高的优先级规则之下。

优先级可以设定为一个数字值，范围从最极小的 - 10 000 到最大的 10 000。如果程序员没有在程序中明确指定规则的优先级，则 CLIPS 假定其优先级值为 0。注意，优先级值 0 是介于最大和最小优先级值之间的。优先级为 0 并不意味着该规则没有优先级，而是意味着它有一个中间的优先级。一个最近激活的规则被置于议程中优先级相同或更低的所有规则之前、优先级更高的所有规则之后。

优先级的应用之一是迫使规则以连续的方式触发。考虑下列规则集，规则中没有说明优先级：

```
(defrule fire-first
   (priority first)
   =>
```

```
   (printout t "Print first" crlf))

(defrule fire-second
   (priority second)
   =>
   (printout t "Print second" crlf))
(defrule fire-third
   (priority third)
   =>
   (printout t "Print third" crlf))
```

规则触发的顺序取决于满足规则 LHS 中 CE 的事实被声明的顺序。例如，如果输入这些规则，则下列命令将产生下面所显示的输出：

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (assert (priority first))↵
<Fact-1>
CLIPS> (assert (priority second))↵
<Fact-2>
CLIPS> (assert (priority third))↵
<Fact-3>
CLIPS> (run)↵
Print third
Print second
Print first
CLIPS>
```

注意输出语句的顺序。Print third 先被打印，然后是 Print second，最后是 Print first。第一个事实 (priority first) 激活规则 fire-first。当第二个事实被声明时，它激活栈中置于规则 fire-first 之上的规则 fire-second。最后，第三个事实被声明，它激活的规则 fire-third 位于栈中的规则 fire-second 之上。

在 CLIPS 中，被不同模式激活的有相同优先级的规则按事实的堆栈顺序来区分优先级次序。议程中规则的触发顺序是自堆栈的顶部开始往下进行。因此，由于 fire-third 规则在栈的顶部，所以，它被首先触发；其次是规则 fire-second；最后是规则 fire-first。如果声明事实的顺序颠倒过来，那么，规则触发的顺序也将颠倒过来。这可从下列输出中看到：

```
CLIPS> (reset)↵
CLIPS> (assert (priority third))↵
<Fact-1>
CLIPS> (assert (priority second))↵
<Fact-2>
CLIPS> (assert (priority first))↵
<Fact-3>
CLIPS> (run)↵
Print first
Print second
Print third
CLIPS>
```

重要的一点是，如果二条或更多的具有相同优先级的规则被同一个事实激活，那么，将无法保证规则以何种顺序置于议程中。

优先级能用来迫使规则按顺序 fire-first、fire-second、fire-third 触发，而不管激活的事实被声明的顺序。这可以通过说明优先级的值来完成：

```
(defrule fire-first
   (declare (salience 30))
   (priority first)
   =>
   (printout t "Print first" crlf))

(defrule fire-second
   (declare (salience 20))
   (priority second)
   =>
```

```
   (printout t "Print second" crlf))

(defrule fire-third
   (declare (salience 10))
   (priority third)
   =>
   (printout t "Print third" crlf))
```

不管事实被声明的优先级顺序如何，该议程总是按相同的顺序排列。在声明优先级事实后执行agenda命令将产生下列输出：

```
CLIPS> (reset)↵
CLIPS> (assert (priority second)
               (priority first)
               (priority third))↵
<Fact-3>
CLIPS> (agenda)↵
30     fire-first: f-2
20     fire-second: f-1
10     fire-third: f-3
For a total of 3 activations.
CLIPS>
```

注意，优先级值是如何重新排列议程中规则的优先级的。当运行程序时，规则触发的顺序将总是：fire-first、fire-second、fire-third。

9.4 阶段和控制事实

基于规则的专家系统的最精确概念是，每当它们可适用时，规则按机率执行。然而，大多数专家系统有某种程序特点。例如，Sticks程序，它有不同的规则可适用，只是取决于它是人类的意图还是计算机的愿望。此程序的控制，由表明轮到顺序的事实进行处理。这些控制事实允许有关程序的控制结构的信息，嵌入到领域知识的规则中。这有一个缺点：关于规则的控制知识与关于如何玩游戏的知识相互混淆起来。对于Sticks程序来说这不是主要的缺点，因为此程序很小。但是，对于那些有成百或成千条规则的程序来说，混淆领域知识和控制知识会使程序的开发和维护成为主要问题。

例如，考虑一个系统（如电子设备）的故障检测、隔离和恢复的问题。故障检测是指识别电子设备工作不正常的过程。隔离是指找出使设备产生故障的设备元件的过程。恢复是指决定排除故障（如果可能的话）必须采取的步骤的过程。一般地，对这种类型的问题，专家系统必须有一些规则确定是否出现了故障、有一些规则隔离故障的原因、还要有一些规则决定如何从故障中恢复。然后，继续循环。图9.1给出了这种类型系统的控制流的例子。

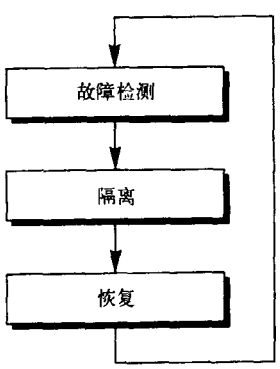


图9.1 故障检测、隔离和恢复问题的不同阶段

在此系统中，实现控制流至少有4种方法。前3种应用优先级的方法将在这里讨论。第4种应用模块的方法将在本章稍后讨论。

第一种实现控制流的方法是直接将控制知识嵌入规则中。例如，(故障）检测规则包括指示隔离阶段应在何时开始的规则。对每个组的规则将给予一个模式，以指示在哪一个阶段该规则可以被应用。此项技术有两个缺点：第一个缺点已提到过，控制知识嵌入到领域知识规则中，难以理解；第二个缺点是，一个阶段完成的时间不是都易于决定。一般地，只有所有其他的规则已触发，才可写出一个合用的规则。

第二种方法是应用优先级来组织规则，如图9.2所示。这种方法也有两个缺点：第一，控制知识仍被嵌入到使用优先级的规则中；第二，这种方法并不保证正确的执行顺序。检测规则总是先于隔离规则触发。但是，当隔离规则开始触发时，它们可能会引起检测规则被激活，并由于检测规则有更高的优先级而立即触发。

在控制执行流的过程中，第三种也是更好的一种方法是，将控制知识与领域知识分离，如图 9.3 所示。用这种方式，每个规则被给予一个控制模式，指示它的适用阶段。然后，写出控制规则以便在不同时期转换控制，如下所示：

```
(defrule detection-to-isolation
   (declare (salience -10))
   ?phase <- (phase detection)
   =>
   (retract ?phase)
   (assert (phase isolation)))

(defrule isolation-to-recovery
   (declare (salience -10))
   ?phase <- (phase isolation)
   =>
   (retract ?phase)
   (assert (phase recovery)))

(defrule recovery-to-detection
   (declare (salience -10))
   ?phase <- (phase recovery)
   =>
   (retract ?phase)
   (assert (phase detection)))
```

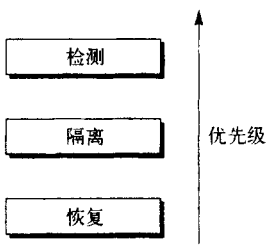


图 9.2 优先级在不同阶段的分配

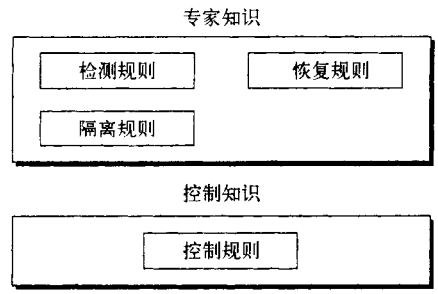


图 9.3 专家知识和控制知识的分离

适用于特定阶段的每一条规则都被给予一个控制模式，此模式用于确认适当的控制事实已出现。例如，恢复规则可能会是这样：

```
(defrule find-fault-location-and-recovery
   (phase recovery)
   (recovery-solution switch-device
                      ?replacement on)
   =>
   (printout t "Switch device" ?replacement "on"
               crlf))
```

优先级层次（salience hierarchy）用于对专家系统的优先级值进行描述。优先级层次中的每一级都和一组特定的规则相对应，这组规则的成员都有相同的优先级。如果检测、隔离和恢复 3 条规则都被给予一个为 0 的默认优先级，那么，优先级层次可示于图 9.4。注意到，当事实（phase detection）在事实表中时，规则 detection-to-isolation 将在议程中。既然它有比检测规则更低的优先级，因此，在所有的检测规则有机会触发之前，它将不会触发。下列输出给出了之前的 3 个控制规则的运行范例：

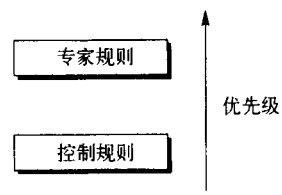


图 9.4 使用专家规则和控制规则的优先级层次

```
CLIPS> (reset)↵
CLIPS> (assert (phase detection))↵
<Fact-1>
CLIPS> (watch rules)↵
CLIPS> (run 10)↵
```

```
FIRE    1 detection-to-isolation: f-1
FIRE    2 isolation-to-recovery: f-2
FIRE    3 recovery-to-detection: f-3
FIRE    4 detection-to-isolation: f-4
FIRE    5 isolation-to-recovery: f-5
FIRE    6 recovery-to-detection: f-6
FIRE    7 detection-to-isolation: f-7
FIRE    8 isolation-to-recovery: f-8
FIRE    9 recovery-to-detection: f-9
FIRE   10 detection-to-isolation: f-10
CLIPS>
```

注意，控制规则仅仅按顺序逐个触发，因为没有领域知识规则在任何阶段被应用。如果有，则领域知识规则将在适当的阶段应用于已激活的规则。

利用自定义事实结构和一条规则，可以将前面的控制规则更一般地写成：

```
(deffacts control-information
   (phase detection)
   (phase-after detection isolation)
   (phase-after isolation recovery)
   (phase-after recovery detection))
(defrule change-phase
   (declare (salience -10))
   ?phase <- (phase ?current-phase)
   (phase-after ?current-phase ?next-phase)
   =>
   (retract ?phase)
   (assert (phase ?next-phase)))
```

或者，利用一系列循环执行的阶段将它们写为：

```
(deffacts control-information
   (phase detection)
   (phase-sequence isolation recovery detection))

(defrule change-phase
   (declare (salience -10))
   ?phase <- (phase ?current-phase)
   ?list <- (phase-sequence ?next-phase
                            $?other-phases)
   =>
   (retract ?phase ?list)
   (assert (phase ?next-phase))
   (assert (phase-sequence ?other-phases
                           ?next-phase)))
```

附加层次能很容易地添加到优先级层次中去。图9.5有2个附加层次。约束规则表示检测非法或多余状态的规则，这些状态在专家系统中可能会出现。例如，一个给人安排各种任务的专家系统可能会得到一份违反约束的计划表。约束规则不是允许较低的优先级规则继续在计划表中起作用，而是立即删除该计划表中的冲突。另一例子是，用户可能输入一系列合法值作为对相应的一整套问题的响应，而此合法值会产生某个非法值。约束规则可用来解决这些冲突。

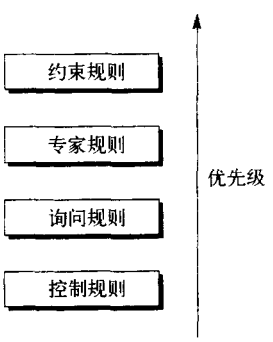


图9.5 4层优先级层次

在图9.5中反映的询问规则代表了那些询问用户一些特定问题以协助专家系统作出回答的规则。这些规则比专家规则的优先级低，因为询问用户可由专家规则解决的问题是不合适的。这样，只有当专家规则不能得到更多信息时才使用询问规则。

9.5 优先级属性的误用

优先级属性虽然是一个强有力的控制执行工具，但它容易被滥用。特别是那些基于规则的程序设计初学者，因为优先级属性能让它们明确地控制执行过程。这更像它们

常用的过程程序设计，其中的语句按顺序执行。

过多地使用优先级属性会导致质量低劣的程序。基于规则程序的主要优点是程序员不必担心控制执行的过程。一个设计良好的基于规则的程序有一个自然的执行模式，它允许推理机指导规则以最佳方法触发。

优先级属性应主要作为确定规则触发顺序的机制。这意味着，一条存于议程中的规则通常最后才触发。当模式能用来描述选择的条件时，优先级属性就不应作为从一组规则中选择一条规则的方法，也不应作为“快速解决方法（quick fix）”来使规则按合适的顺序触发。

通常，任何在规则中使用的优先级值都应对应着专家系统优先级层次中的一层。认为优先级值的范围为 -10 000～10 000 是容易引起误解的。编写专家系统程序极少需要 7 个以上的优先级值；大部分编写良好的专家系统需要的优先级值不会超过 3 个或 4 个。对于一个大型专家系统，强烈推荐程序员最好利用模块控制执行的流程（下一节讨论），并且优先级值不超过 2 个或 3 个。

下面是怎样避免使用优先级的例子。该例示出一套简单的规则，该规则建议在一字棋游戏中标记哪些方格。这些规则按它们被使用的顺序列出。

```
IF a winning square is open, THEN take it.
IF a blocking square is open, THEN take it.
IF a square is open, THEN take it.
```

如果事实 choose-move 表明应该选移动，且事实 open-square 表明一个成功（winning）、堵塞（blocking）、中间（middle）、拐角（corner）或边界（side）方格是否开放，则下面规则将选择合适的移动：

```
(defrule pick-to-win
   (declare (salience 10))
   ?phase <- (choose-move)
   (open-square win)
   =>
   (retract ?phase)
   (assert (move-to win)))

(defrule pick-to-block
   (declare (salience 5))
   ?phase <- (choose-move)
   (open-square block)
   =>
   (retract ?phase)
   (assert (move-to block)))

(defrule pick-any
   ?phase <- (choose-move)
   (open-square ?any&corner|middle|side)
   =>
   (retract ?phase)
   (assert (move-to ?any)))
```

注意，如果超过一种方格可选用，则全部 3 条规则都将放在议程中。当规则 pick-to-win、pick-to-block 或 pick-any 三者之一被触发时，撤销控制事实将移走议程中的其他规则。这三条规则就是这样紧密联系的。如果不整体理解而想单独理解每条规则的意图是不可能的。这违反了基于规则编程的基本观点：一条规则应尽可能多地代表一种表达完全的启发式方法。在这种情况下，优先级属性用来表达一些规则间的固有关系，而这种关系能利用这些规则中的附加模式加以明确描述。不使用优先级属性，这些规则可改写为：

```
(defrule pick-to-win
   ?phase <- (choose-move)
   (open-square win)
   =>
   (retract ?phase)
   (assert (move-to win)))

(defrule pick-to-block
   ?phase <- (choose-move)
```

```
   (open-square block)
   (not (open-square win))
   =>
   (retract ?phase)
   (assert (move-to block)))

(defrule pick-any
   ?phase <- (choose-move)
   (open-square ?any&corner|middle|side)
   (not (open-square win))
   (not (open-square block))
   =>
   (retract ?phase)
   (assert (move-to ?any)))
```

把附加模式添加到规则上明确说明了规则可用的条件。规则间紧密的相互作用被切断，从而允许这些规则有机会生效。重写这些规则也可证明原来的启发式方法可以表达得更清楚。

```
IF a winning square is open, THEN take it.
IF a blocking square is open, and
   a winning square is not open, THEN take it.
IF a corner, middle, or side square is open, and
   a winning square is not open, and
   a blocking square is not open, THEN take it.
```

9.6 自定义模块结构

直到现在，所有的自定义规则、自定义模板和自定义事实都只包含在单个工作空间中。CLIPS用**自定义模块**（defmodule）结构通过定义不同的模块来划分知识库。这种结构的基本语法是：

```
(defmodule <module-name> [<comment>])
```

CLIPS定义的模块默认名叫作MAIN模块。在前面的例子中，我们已看到MAIN模块名伴随着漂亮的结构打印描述而出现。例如，

```
CLIPS> (clear)↵
CLIPS> (deftemplate sensor (slot name))↵
CLIPS> (ppdeftemplate sensor)↵
(deftemplate MAIN::sensor
   (slot name))
CLIPS>
```

在名称MAIN::sensor中的符号“::”叫做**模块分隔符**（module separator）。模块分隔符的右边是结构的名称。分隔符的左边是包含于结构中的模块名。因为所有至此被定义的结构全部默认地放在MAIN模块中，所以，MAIN模块名和漂亮的结构打印描述一起出现。

既然我们知道了自定义模块的语法格式，我们就能定义新的模块。将前面的故障检测系统作为例子，我们将定义对应于DETECTION（检测）、ISOLATION（隔离）和RECOVERY（恢复）三个阶段的模块。

```
CLIPS> (defmodule DETECTION)↵
CLIPS> (defmodule ISOLATION)↵
CLIPS> (defmodule RECOVERY)↵
```

一旦我们定义了除MAIN模块之外更多的模块，则“新结构应放到哪个模块中”的问题也就出现了。默认地，CLIPS把新定义的结构放到当前模块中。当CLIPS开始启动或被清除时，当前模块自动置为MAIN模块。这样，在前面几个例子中，因为只有一个模块且它是当前模块，所以，所有定义的结构都放到了MAIN模块中。

无论什么时候定义一个新模块，CLIPS都把该新建的模块置为当前模块。因为上面例子中最后定义的是RECOVERY模块，因此，它就是当前模块，新定义的规则将置于其中。

```
CLIPS> (defrule example1 =>)↵
CLIPS> (ppdefrule example1)↵
(defrule RECOVERY::example1
   =>)
CLIPS>
```

结构所在的模块可在结构名中详细说明。在结构名中，首先规定模块，接着是模块分隔符，最后是结构名。

```
CLIPS> (defrule ISOLATION::example2 =>)↵
CLIPS> (ppdefrule example2)↵
(defrule ISOLATION::example2
   =>)
CLIPS>
```

当在结构名中指定了模块名时，当前模块则被改变。通过函数 get-current-module 可以确定当前模块。这个函数不需要参数，返回当前模块名。函数 set-current-module 是用来改变当前模块的。它需要一个参数：新的当前模块名。它返回前一次的当前模块名。

```
CLIPS> (get-current-module)↵
ISOLATION
CLIPS> (set-current-module DETECTION)↵
ISOLATION
CLIPS>
```

命令中指定模块

默认地，大部分 CLIPS 的结构命令只对包含于当前模块的结构有效。例如，如果当前模块是 DETECTION，那么，命令 list-defrules 将不产生输出，因为该模块没有包含规则：

```
CLIPS> (list-defrules)↵
CLIPS>
```

如果想看包含在 ISOLATION 模块中的 defrules，我们可以设置当前模块为 ISOLATION，然后执行另一 list-defrules 命令。

```
CLIPS> (set-current-module ISOLATION)↵
DETECTION
CLIPS> (list-defrules)↵
example2
For a total of 1 defrule.
CLIPS>
```

或者，list-defrules 命令接受一个模块名作为一个可选参数，该参数说明列出哪个模块的规则：

```
CLIPS> (list-defrules RECOVERY)↵
example1
For a total of 1 defrule.
CLIPS>
```

如果符号 * 作为 list-defrules 的参数，那么将列出所有模块的规则。模块名先于规则列出。

```
CLIPS> (list-defrules *)↵
MAIN:
DETECTION:
ISOLATION:
   example2
RECOVERY:
   example1
For a total of 2 defrules.
CLIPS>
```

List-deftemplates 和 list-deffacts 的功能与 list-defrules 命令相似。Show-breaks 命令显示指定模块中的断点。这些函数的语法结构为：

```
(list-defrules [<module-name>])
(list-deftemplates [<module-name>])
(list-deffacts [<module-name>])
(show-breaks [<module-name>])
```

对指定结构的操作允许说明一个模块名。例如，如果没有指明模块名，ppdefrule 只搜索当前模块：

```
CLIPS> (ppdefrule example2)↵
(defrule ISOLATION::example2
   =>)
CLIPS> (ppdefrule example1)↵
[PRNTUTIL1] Unable to find defrule example1.
CLIPS>
```

Example2 规则能以漂亮打印格式显示因为它包含在 ISOLATION 模块里面，但是 example1 规则不在 ISOLATION 里面，所以 ppdefrule 命令找不到它。

可以在结构名之前，模块分隔符之后设置所要找的模块名。例如：

```
CLIPS> (ppdefrule RECOVERY::example1)↵
(defrule RECOVERY::example1
   =>)
CLIPS>
```

在不同模块中可能存在同名的结构。在结构名前使用模块说明可以在命令中将两者区分开来：

```
CLIPS> (defrule DETECTION::example1 =>)↵
CLIPS> (list-defrules *)↵
MAIN:
DETECTION:
   example1
ISOLATION:
   example2
RECOVERY:
   example1
For a total of 3 defrules.
CLIPS> (ppdefrule RECOVERY::example1)↵
(defrule RECOVERY::example1
   =>)
CLIPS> (ppdefrule DETECTION::example1)↵
(defrule DETECTION::example1
   =>)
CLIPS>
```

下列命令允许模块说明为结构名的一部分：ppdefrule，undefrule，ppdeftemplate，undeftemplate，ppdeffacts，undeffacts，matches，refresh，remove-break，set-break。

9.7 输入、输出事实

你已经学会了通过把结构置于不同模块中来划分结构的方法。事实本身也可以这样划分。被声明的事实会自动与定义的对应自定义模板联系上。例如：

```
CLIPS>
(deftemplate DETECTION::fault
   (slot component))↵
CLIPS> (assert (fault (component A)))↵
<Fact-0>
CLIPS> (facts)↵
f-0     (fault (component A))
For a total of 1 fact.
CLIPS>
(deftemplate ISOLATION::possible-failure
   (slot component))↵
CLIPS> (assert (possible-failure (component B)))↵
<Fact-1>
CLIPS> (facts)↵
f-1     (possible-failure (component B))
For a total of 1 fact.
CLIPS> (set-current-module DETECTION)↵
ISOLATION
CLIPS> (facts)↵
f-0     (fault (component A))
For a total of 1 fact.
CLIPS>
```

注意，在 ISOLATION 模块中，列出的惟一事实是 possible-failure 事实，它对应的自定义模板包含在此 ISOLATION 模块中。在 DETECTION 模块中的 fault 事实也同样如此。

facts 命令与 list-defrules 及其类似命令一样，能接受一个模块名作为可选参数。facts 命令的语法结构为：

```
(facts [<module-name>]
       [<start> [<end> [<maximum>]]])
```

和 list-defrules 命令一样，指明一个模块名只会列出包含于该模块中的事实。如果用符号 * 来代替模块名，则会列出所有事实：

```
CLIPS> (facts DETECTION)↵
f-0     (fault (component A))
For a total of 1 fact.
CLIPS> (facts ISOLATION)↵
f-1     (possible-failure (component B))
For a total of 1 fact.
CLIPS> (facts RECOVERY)↵
CLIPS> (facts *)↵
f-0     (fault (component A))
f-1     (possible-failure (component B))
For a total of 2 facts.
CLIPS>
```

自定义模板结构（和所有使用该自定义模板的事实）不像自定义规则和自定义事实结构，它可以与其他模块共享。事实被包含其自定义模板的模块所“拥有”，但该模块能**输出**（export）与事实相关联的自定义模板，这样，使得该事实和所有使用该自定义模板的事实都可以为其他模块可见（visible）。但仅输出自定义模板使某个事实为另一个模块可见是不够的。为了能使用在另一个模块中定义的自定义模板，一个模块还必须**输入**（import）该自定义模板的定义。

输出自定义模板的模块必须在自定义模块中使用输出属性。输出属性必须使用下面格式之一：

```
(export ?ALL)
(export ?NONE)
(export deftemplate ?ALL)
(export deftemplate ?NONE)
(export deftemplate <deftemplate-name>+)
```

第一种格式从一个模块中输出所有可输出的结构。在本书目前为止所讨论的结构中，只有自定义模板是可以输出的。在 CLIPS 中某些其他例程和面向对象程序结构也可以输出，其输出方式在第 10 章和第 11 章中讨论。第二种格式表示没有结构输出，这是自定义模块的默认情况。第三种格式表明，模块中的所有自定义模板都输出。对于本书中讨论的结构来说，这与第一种格式是相同的。类似地，第四种格式表明，没有自定义模板结构输出。第二和第四种格式主要用来使通过模块输出的结构能够明确地加以说明。最后，第五种格式提供了通过模块输出的自定义模板的一个指定列表。输出属性能够在模块定义中多次使用，以对不同种类的输出结构进行规定。但在我们所讨论的范围内，自定义模板是唯一的可输出结构，因此，没有必要使用多个输出属性语句。

输入属性也有以下 5 种可能的格式：

```
(import <module-name> ?ALL)
(import <module-name> ?NONE)
(import <module-name> deftemplate ?ALL)
(import <module-name> deftemplate ?NONE)
(import <module-name> deftemplate
                      <deftemplate-name>+)
```

以上格式除了指定的结构被输入之外，其他含义与对应的输出格式相同。此外，从其中输入结构的模块必须加以说明。与输出属性一样，自定义模块也可以有多种输入属性。

一种结构，在输入列表中说明它之前，必须定义之；但在输出列表中说明结构之前，则不必定义（在输出列表中说明结构的目的是将结构置于模块中，此模块必须要定义。所以，实际上在输出结构的模块定义之前是不可能定义结构的）。由于这种限制，两种模块互相从对方输入是不可能的（例如：如果模块 A 从模块 B 输入，则模块 B 不可能从模块 A 中输入）。

为了说明输入和输出事实的含义，我们可以假设模块 RECOVERY 从模块 DETECTION 中输入 fault 自定义模板、从模块 ISOLATION 输入 possible-failure 自定义模板。和其他结构不同，自定义模块一旦被定义后则不可以被重定义。因此，为了改变其输入和输出属性，一个清除命令必须要先发出。不过有一种情况是不受此限制的：即预定义的 MAIN 模块可以重新定义一次以包括不同的输入和输出

属性（MAIN模块的默认值是不输入不输出任何东西）。注意，MAIN模块的默认定义并不输出自定义模板initial-fact。回忆第8章，我们知道这种（initial-fact）模式在某种情况下（例如，当第一个CE是一个not CE时）被加入到规则的LHS中。如果这样一条规则置于不从MAIN模块中输入initial-fact自定义模板的模块中，则不可能激活此规则。而且注意，你在定义规则的时候不会收到错误信息，因为这种（initial-fact）模式会导致在当前模块中建立一个隐式initial-fact自定义模板。

至于DETECTION、ISOLATION和RECOVERY模块及其自定义模板（都应该在clear命令后输入）的全新定义如下：

```
(defmodule DETECTION
   (export deftemplate fault))

(deftemplate DETECTION::fault
   (slot component))
(defmodule ISOLATION
   (export deftemplate possible-failure))

(deftemplate ISOLATION::possible-failure
   (slot component))

(defmodule RECOVERY
   (import DETECTION deftemplate fault)
   (import ISOLATION deftemplate possible-
   failure))
```

有了这些定义，就可以声明在DETECTION和RECOVERY模块中的fault事实以及声明在ISOLATION和RECOVERY中的possible-failure事实：

```
CLIPS>
 (deffacts DETECTION::start
 (fault (component A)))↵
CLIPS>
 (deffacts ISOLATION::start
 (possible-failure (component B)))↵
CLIPS>
 (deffacts RECOVERY::start
 (fault (component C))
 (possible-failure (component D)))↵
CLIPS> (reset)↵
CLIPS> (facts DETECTION)↵
f-0     (fault (component A))
f-2     (fault (component C))
For a total of 2 facts.
CLIPS> (facts ISOLATION)↵
f-1     (possible-failure (component B))
f-3     (possible-failure (component D))
For a total of 2 facts.
CLIPS> (facts RECOVERY)↵
f-0     (fault (component A))
f-1     (possible-failure (component B))
f-2     (fault (component C))
f-3     (possible-failure (component D))
For a total of 4 facts.
CLIPS>
```

注意，模块DETECTION和RECOVERY均可以看到由它们之中任意一个声明的fault事实。同样也适用于由模块ISOLATION和RECOVERY声明的possible-failure事实。

9.8 模块与执行控制

除了控制模块能输入和输出哪些自定义模板外，自定义模块结构还能用来控制规则的执行。每个在CLIPS中定义过的模块都有自己的议程，而不只是一个总体议程中的一部分。通过选择模块议程来执行规则可以进行执行控制。例如，以下自定义规则可全部由上例中声明的fault和possible-failure事实激活：

```
(defrule DETECTION::rule-1
   (fault (component A | C))
   =>)

(defrule ISOLATION::rule-2
   (possible-failure (component B | D))
   =>)

(defrule RECOVERY::rule-3
   (fault (component A | C))
   (possible-failure (component B | D))
   =>)
```

如果 agenda 命令是在这些规则调入后发出的，则由于定义的最后规则是放入 RECOVERY 模块而显示该模块的议程：

```
CLIPS> (get-current-module)↵
RECOVERY
CLIPS> (agenda)↵
0      rule-3: f-0,f-3
0      rule-3: f-0,f-1
0      rule-3: f-2,f-3
0      rule-3: f-2,f-1
For a total of 4 activations.
CLIPS>
```

与 list-defrules 和 facts 命令一样，agenda 命令也接受一个可选参数，该参数指示列出哪个模块的议程：

```
CLIPS> (agenda DETECTION)↵
0      rule-1: f-2
0      rule-1: f-0
For a total of 2 activations.
CLIPS> (agenda ISOLATION)↵
0      rule-2: f-3
0      rule-2: f-1
For a total of 2 activations.
CLIPS> (agenda RECOVERY)↵
0      rule-3: f-0,f-3
0      rule-3: f-0,f-1
0      rule-3: f-2,f-3
0      rule-3: f-2,f-1
For a total of 4 activations.
CLIPS> (agenda *)↵
MAIN:
DETECTION:
   0      rule-1: f-2
   0      rule-1: f-0
ISOLATION:
   0      rule-2: f-3
   0      rule-2: f-1
RECOVERY:
   0      rule-3: f-0,f-3
   0      rule-3: f-0,f-1
   0      rule-3: f-2,f-3
   0      rule-3: f-2,f-1
For a total of 8 activations.
CLIPS>
```

Focus 命令

此时，有 3 个不同议程中的规则，那么，当发出一个 run 命令时会发生什么情况呢?

```
CLIPS> (unwatch all)↵
CLIPS> (watch rules)↵
CLIPS> (run)↵
CLIPS>
```

没有规则触发！除当前模块，即 CLIPS 使用该模块决定新结构加在何处、哪些结构可以使用或哪些结构会受命令影响外，CLIPS 还维护**当前焦点**（current focus），从而决定在执行过程中 run 命令使用

哪个议程。reset 和 clear 命令自动将当前焦点设为 MAIN 模块。当前焦点在当前模块改变时保持不变。因此，在本例中，当发出 run 命令后，与 MAIN 模块相联系的议程被用来选择规则以便执行。但因该议程为空，所以不会触发任何规则。

focus 命令可用来改变当前焦点。其语法如下：

```
(focus <module-name>+)
```

在这个简单的例子中只规定一个模块名，当前焦点被设置为指定的模块。通过将当前焦点设为模块 DETECTION，然后发出 run 命令，则在模块 DETECTION 议程中的规则将被触发：

```
CLIPS> (focus DETECTION)↵
TRUE
CLIPS> (run)↵
FIRE    1 rule-1: f-2
FIRE    2 rule-1: f-0
CLIPS>
```

使用 focus 命令不仅会改变当前焦点，而且会唤起当前焦点的前一个值。实际上，当前焦点就是称为焦点栈的堆栈数据结构的顶点值。无论何时用 focus 命令改变当前焦点，实际上就是将新的焦点推至焦点栈的顶部，从而取代原焦点。随着规则的执行，当前焦点的议程变空时，当前焦点将从焦点栈中弹出（移出），同时另一模块成为当前焦点。然后，规则将按新的当前焦点的议程执行，直至另一新模块成为焦点或已无规则遗留在当前焦点的议程中。规则会一直继续执行，直至在焦点栈中无模块留下或发出 halt 命令为止。

继续讨论本例，先把 ISOLATION 模块作为焦点、再将 RECOVERY 模块作为焦点将触发所有在 RECOVERY 模块的议程中的规则，紧接着又会触发 ISOLATION 模块的议程中的规则。list-focus-stack 命令可用来显示焦点栈中的模块。

```
CLIPS> (focus ISOLATION)↵
TRUE
CLIPS> (focus RECOVERY)↵
TRUE
CLIPS> (list-focus-stack)↵
RECOVERY
ISOLATION
CLIPS> (run)↵
FIRE    1 rule-3: f-1,f-4
FIRE    2 rule-3: f-1,f-2
FIRE    3 rule-3: f-3,f-4
FIRE    4 rule-3: f-3,f-2
FIRE    5 rule-2: f-4
FIRE    6 rule-2: f-2
CLIPS> (list-focus-stack)↵
CLIPS>
```

用两个 focus 命令来压入 ISOLATION 和 RECOVERY 模块将导致 RECOVERY 规则在 ISOLATION 规则之前执行。然而，当多个模块在单个 focus 命令中指定时，这些模块将从右至左推入焦点栈中。例如：

```
CLIPS> (focus ISOLATION RECOVERY)↵
TRUE
CLIPS> (list-focus-stack)↵
ISOLATION
RECOVERY
CLIPS> (focus ISOLATION)↵
TRUE
CLIPS> (list-focus-stack)↵
ISOLATION
RECOVERY
CLIPS> (focus RECOVERY)↵
TRUE
CLIPS> (list-focus-stack)↵
RECOVERY
ISOLATION
RECOVERY
CLIPS>
```

注意，同一个模块可以不只一次出现在焦点栈中，但是，压入一个已经是当前焦点的模块于栈中将不起作用。

控制和检查焦点栈

CLIPS提供了几个命令和函数处理当前焦点和焦点栈。clear-focus-stack命令将所有模块从焦点栈中移出。而get-focus命令则返回当前焦点的模块名，或者，当焦点栈为空时返回符号FALSE。pop-focus函数删除焦点栈中的当前焦点（且返回模块名，或者当焦点栈为空时返回符号FALSE）。get-focus-stack函数则返回包含在焦点栈中模块的多字段值。

```
CLIPS> (get-focus-stack)↵
(RECOVERY ISOLATION RECOVERY)
CLIPS> (get-focus)↵
RECOVERY
CLIPS> (pop-focus)↵
RECOVERY
CLIPS> (clear-focus-stack)↵
CLIPS> (get-focus-stack)↵
()
CLIPS> (get-focus)↵
FALSE
CLIPS> (pop-focus)↵
FALSE
CLIPS>
```

watch命令可以用关键字focus作为参数，以此来监视焦点栈的变化：

```
CLIPS> (watch focus)↵
CLIPS> (focus DETECTION ISOLATION RECOVERY)↵
==> Focus RECOVERY
==> Focus ISOLATION from RECOVERY
==> Focus DETECTION from ISOLATION
TRUE
CLIPS> (run)↵
<== Focus DETECTION to ISOLATION
<== Focus ISOLATION to RECOVERY
<== Focus RECOVERY
CLIPS>
```

在run命令已发出且焦点栈为空的情况下，MAIN模块会自动推入焦点栈。这种特性主要是为了在程序结束后无模块剩余在焦点栈中时，为添加新的激活而提供方便措施。例如：

```
CLIPS> (clear)↵
CLIPS> (watch focus)↵
CLIPS> (watch rules)↵
CLIPS> (defrule example-1 =>)↵
CLIPS> (reset)
<== Focus MAIN
==> Focus MAIN
CLIPS> (run)↵
FIRE    1 example-1: f-0
<== Focus MAIN
CLIPS> (defrule example-2 =>)↵
CLIPS> (agenda)↵
0      example-2: f-0
For a total of 1 activation.
CLIPS> (list-focus-stack)↵
CLIPS>
```

规则example-2在议程中，但无模块于焦点栈中。发出一个run命令将MAIN模块推入焦点栈，因而规则example-2无论如何都可以触发。

```
CLIPS> (run)↵
==> Focus MAIN
FIRE    1 example-2: f-0
<== Focus MAIN
CLIPS>
```

Return 命令

正如在第 9.4 节中讨论的那样，用控制事实来表示阶段以控制执行流程有一个缺点，就是在某个特定状态下不可能触发某些激活（activation），只有退出该阶段，然后再回到该阶段执行议程中剩余的激活。这种情况的发生是由于一旦表示阶段的控制事实被撤销，则该阶段的前面所有激活将从议程中移出。当控制事实随后被重新声明时，此阶段所有前面的激活也将被重新激活，而不仅仅是重新激活那些以前未触发的激活（当然，这是假定只声明或只撤销该控制事实，而不是任何其他事实）。

如果模块用来控制执行流程，则可能提前（即，在模块议程为空之前）终止执行特定模块议程中的激活。return 命令可用来立刻中止规则 RHS 的执行，并将当前焦点从焦点栈中移出（因此，将执行控制返回到焦点栈中的下一个模块）。当 return 命令在规则 RHS 中运用时，就不应该传递参数给它（在 CLIPS 提供的例程结构中也可以使用 return 命令）。下面的例子可用来解释返回命令的使用：

```
CLIPS> (clear)↵
CLIPS>
(defmodule MAIN
   (export deftemplate initial-fact))↵
CLIPS>
(defmodule DETECTION
   (import MAIN deftemplate initial-fact))↵
CLIPS>
(defrule MAIN::start
   =>
   (focus DETECTION))↵
CLIPS>
(defrule DETECTION::example-1
=>
  (return)
  (printout t "No printout!" crlf))↵
CLIPS>
(defrule DETECTION::example-2
=>
  (return)
  (printout t "No printout!" crlf))↵
CLIPS> (watch rules)↵
CLIPS> (watch focus)↵
CLIPS> (reset)↵
<== Focus MAIN
==> Focus MAIN
CLIPS> (run)↵
FIRE    1 start: f-0
==> Focus DETECTION from MAIN
FIRE    2 example-1: f-0
<== Focus DETECTION to MAIN
<== Focus MAIN
CLIPS>
```

本例有两点值得注意。首先，为了用默认的 initial-fact 模式激活 DETECTION 模块中的规则，自定义模板 initial-fact 必须从 MAIN 模块输出并由 DETECTION 模块输入。其次，注意 return 命令将立刻挂起规则的 RHS 的执行。printout 命令（在 return 命令后）将不会在 example-1 规则中执行（或，不会在 example-2 规则中执行，因为没有机会触发）。注意，return 函数的功能与 pop-focus 命令类似，但不完全相同，后者从焦点栈中删除当前焦点，但允许规则的 RHS 操作继续执行。若在本例中用后者代替 return 命令，则当执行规则 example-1 的操作时将显示字符串“No printout!”。

自动焦点特性

除了可以用 focus 命令明确聚焦到模块之外，当某个模块中特定的规则被激活时，自动聚焦到该模块也是可能的。默认状态下，被激活规则的模块是不会自动被聚焦的。但使用自动聚焦（auto-focus）特性可以改变这种情况。与优先级属性一起，自动聚焦特性在 declare 语句中规定。规定的关键词 auto-focus 之后是 TRUE（使该特性有效）或 FALSE（使该特性无效）。没有必要为了让模块中的某些规则

使用自动聚焦功能，而使模块中的所有规则都有允许的自动聚焦特性。同样，如果规则中使用 declare 语句，也没有必要既宣布（declare）优先级属性又宣布自动聚焦特性。下面的例子说明了自动聚焦特性的使用：

```
CLIPS> (clear)↵
CLIPS>
(defmodule MAIN
   (export deftemplate initial-fact))↵
CLIPS>
(defmodule DETECTION
   (import MAIN deftemplate initial-fact))↵
CLIPS>
(defrule DETECTION::example
   (declare (auto-focus TRUE))
   =>)↵
CLIPS> (watch focus)↵
CLIPS> (reset)↵
<== Focus MAIN
==> Focus MAIN
==> Focus DETECTION from MAIN
CLIPS>
```

当发出 reset 命令时，MAIN 模块就会因焦点栈已空而自动聚焦。例中 example 规则被 initial-fact 事实的声明激活。由于对此规则自动聚焦特性已经有效，所以，DETECTION 模块会自动地被推入焦点栈中。自动聚焦特性对探测约束冲突的规则特别有用。因为该约束规则的模块会立即成为当前焦点，因此，当冲突出现时采取行动是可能的，而不需要检测冲突的明确阶段。

替换阶段和控制事实

通过使用自定义模块、focus 和 return 命令，以及自动聚焦属性，就可以用更加明确的机制取代阶段和控制事实的使用，从而控制规则执行的流程。在第 9.4 节里描述过的控制执行的结构可以用以下的结构来代替：

```
(defmodule DETECTION)
(defmodule ISOLATION)
(defmodule RECOVERY)

(deffacts MAIN::control-information
   (phase-sequence DETECTION ISOLATION RECOVERY))

(defrule MAIN::change-phase
   ?list <- (phase-sequence ?next-phase
                            $?other-phases)
   =>
   (focus ?next-phase)
   (retract ?list)
   (assert (phase-sequence ?other-phases
                           ?next-phase)))
```

执行的控制可以从模块 DETECTION 到模块 ISOLATION 再到模块 RECOVERY 进行，然后再从头重新开始循环。在允许下一个模块的规则触发以前（除非发出了 return 命令或由于自动聚焦特性而使一个新的模块被聚焦），每个模块中的所有规则都会触发。

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (watch rules)↵
CLIPS> (watch focus)↵
CLIPS> (run 5)↵
FIRE    1 change-phase: f-1
==> Focus DETECTION from MAIN
<== Focus DETECTION to MAIN
FIRE    2 change-phase: f-2
==> Focus ISOLATION from MAIN
<== Focus ISOLATION to MAIN
FIRE    3 change-phase: f-3
```

```
==> Focus RECOVERY from MAIN
<== Focus RECOVERY to MAIN
FIRE    4 change-phase: f-4
==> Focus DETECTION from MAIN
<== Focus DETECTION to MAIN
FIRE    5 change-phase: f-5
==> Focus ISOLATION from MAIN
<== Focus ISOLATION to MAIN
CLIPS>
```

9.9 Rete模式匹配算法

基于规则的的语言，如CLIPS、Jess、Eclipse和OPS5，使用了一种非常有效的算法，将事实和规则中的模式相竞争，以确定哪些规则满足了它们的条件。这种算法称为**Rete模式匹配算法**（Rete Pattern-Matching Algorithm）（Forgy 79，Forgy 85，Brownston 85）。编写有效的CLIPS规则不需要理解Rete算法。然而，理解CLIPS及其他基于规则的语言中使用的基本算法，使人们更易于明白为什么用这种方法写规则会比另一种方法更有效。

要明白为什么Rete算法是有效的，可研究一下通常将事实和规则进行匹配的问题，并接着考察不那么有效的其他算法，这是有帮助的。图9.6示出了Rete算法所提出的问题。

如果匹配过程只需一次，那么这个问题的解决方法就简单易懂了。推理机可以检查每条规则并搜索一组事实来决定规则的模式是否已满足。如果是的话，则将此规则记入议程中。图9.7示出了这一方法。

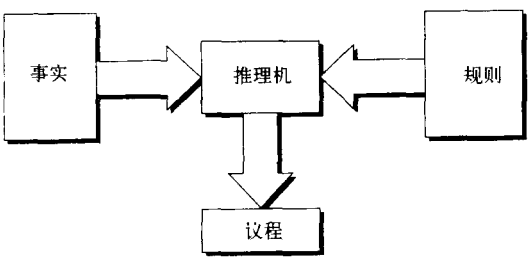


图9.6 模式匹配：规则和事实

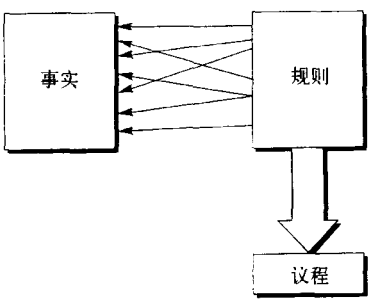


图9.7 搜索事实的规则

然而，在基于规则的语言中，匹配过程不断重复进行。通常，事实表在每次执行中都会被修改，添加新的事实到事实表或删除旧的事实。这些改变可令先前不满足条件的模式得到满足，反之亦然。匹配问题因此成了不断进行的过程。在每次循环中，随着事实的添加和删除，必须对已满足条件的规则集合进行维护和更新。

在每次循环后，令推理机检查每条规则以指导对事实的搜索，从而为解决这个问题提供了简单直接的技巧。这种方法最主要的缺点是速度太慢。大多数基于规则的专家系统都显示了这种特征：**时间冗余性**（temporal redundancy）。一般地，一条规则的运行仅会改变事实表中的少数事实。即：专家系统中的事实随时间改变很慢。在每次循环中，仅添加、删除了很少一部分事实，所以，事实表中的变化一般只影响很少部分的规则。因此，令规则推动对所需事实的搜索，需要大量不必要的计算。这是因为，在当前循环中，大多数规则所找到的事实很可能与上一次循环所找到的相同。图9.8示出了这种低效的方法。阴影部分代表了对事实表所做的改变。正如图9.9所显示的，在不断循环中，通过记住哪些是已经匹配好的，然后只计算那些刚添加或删除事实所引起的必要变化，从而可以避免不必要的计算。规则是不变的，而事实是变化的，所以应是事实寻找相应的规则，而不是反过来。

Rete模式匹配算法正是利用了基于规则的专家系统所具有的时间冗余性。它的实现是通过存储不断循环中匹配过程的状态，并且只重新计算在事实表中发生了变化、又反映到本次状态中的变化来完

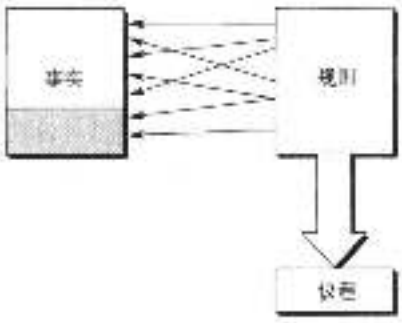


图 9.8 当规则搜索事实时，产生不必要的计算

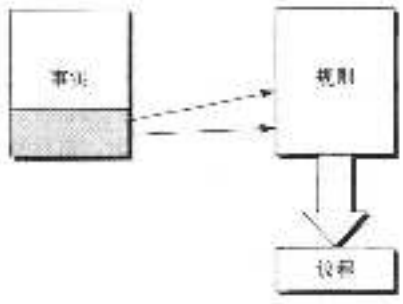


图 9.9 事实搜索规则

或的。也就是说，如果在一次执行周期中，一组模式找到 3 个所需事实中的 2 个，那么，在下一周期中，就无须对已经找到的这两个事实进行检查——只有第 3 个事实才是需要关注的。仅当添加或删除事实时，匹配过程的状态才被更新。如果添加、删除事实的数量与事实和模式的总数相比很小，那么匹配过程会进行的很快。最糟的情况是，如果所有的事实都改变了，那么，所有的事实将与所有的模式进行竞争。

如果仅是事实表进行更新，那么每一条规则必须记住已与之匹配的事实。也就是说，如果一个新的事实与规则的第三个模式相匹配，那么，头两个模式的匹配信息必须存在以完成匹配过程。这种状态信息指出了与某一规则中前面的模式相匹配的事实，称之为**部分匹配**（partial match）。规则的部分匹配是满足规则模式的任何一组事实，它以规则的第一个模式为开始，以任一模式（包括最后一个模式）为结束。因此，一条有三个模式的规则对第一个模式、第一和第二个模式、第一、第二和第三个模式都有部分匹配。一条规则的所有模式的部分匹配包是一个激活。另一种存储的状态信息称为**模式匹配**（pattern match）。当一个事实满足了任一规则中的单个模式而不需考虑在其他模式中可能会限制匹配过程的变量时，则出现的就是模式匹配。

Rete 模式匹配算法的主要缺点是内存使用量大。将所有的事实与所有的模式进行简单地比较不需要使用内存。但是，存储使用了模式匹配和部分匹配的系统的状态会消耗大量的内存。总的来说，为了速度而牺牲内存是值得的，然而要记住，一条设计不好的规则不仅运行速度慢，而且会耗掉大量内存。

通过利用规则中**结构相似性**（structural similarity）的优点，Rete 算法同样也会提高基于规则的系统的效率。结构相似性是指许多规则通常包含了相似的模式或模式群。利用这一特性，Rete 算法通过将公共部分放在一起来提高效率，因为公共部分不必计算一次以上。

9.10 模式网络

事实与规则的匹配问题可分为二步。首先，当添加或删除事实时，必须决定哪些模式是已匹配的。其次，必须对跨模式的变量约束进行比较，以决定模式群的部分匹配。

决定哪些事实已与哪些模式匹配的过程是在**模式网络**（pattern network）中进行的。为简单起见，我们将模式匹配限于自定义模板事实的单字段槽。所有不涉及与其他模式中的约束变量相比较的匹配均可在模式网络中进行。模式网络的结构像一棵树，所有模式的第一槽值约束是与树根相连的结点。所有模式的第二槽值约束又是与此结点相连的另一些结点，依此类推。一个模式中的最后一些槽值约束是该树的叶子。因为每一结点只接收它上层结点的信息，所以，模式网络中的结点称为**单输入结点**（one-input node）。模式网络中的结点有时也称为**模式结点**（pattern node）。叶结点也称为**终端结点**（terminal node）。每一个模式结点包含一个说明，用来决定一个事实的槽值是否与一个模式的槽值约束相匹配。例如，因为只有一个槽值约束，所以，模式：

```
(data (x 27))
```

只需一个结点来表示。第一个结点的匹配说明将是：

```
The slot x value is equal to the constant 27
```

实际上，还须检查是否为事实取了一个合适的对应于模式的关系名（例如，你不希望仅因有相同的槽名字将 foobar 事实与 data 模式相匹配）。为了执行这一测试，CLIPS 为每一个自定义模板设有独立的模式网络，这样，在建立事实时，但在模式匹配进行之前，就进行关系名的检查。

为匹配单个槽，匹配说明包括了所有的信息。一些测试可同时进行。例如，模式：

```
(data (x ~red&~green))
```

会产生以下 x 槽的匹配说明：

```
The slot x value is not equal to the constant red
             and is not equal to the constant green
```

除非变量在模式中使用一次以上，否则，模式网络通常是不检查变量约束的，例如，模式：

```
(data (x ?x) (y ?y) (z ?x))
```

不会为 x 槽或 y 槽生成匹配说明，因为槽值变量的第一个约束对模式是否与事实相匹配没有影响。然而，z 槽一定与 x 槽有相同的值，所以，z 槽的匹配说明是：

```
The z slot value is equal to the x slot value
```

在模式网络中可以检查那些全部位于模式中的含有变量的描述。例如，模式

```
(data (x ?x&:(> ?x ?y)))
```

不会为 x 槽生成匹配说明，因为变量？y 不在模式中（当然，假设变量？y 在前一模式中已被定义），然而，x 槽在模式：

```
(data (x ?x&:(> ?x 4)))
```

中将有匹配说明：

```
The x slot value is greater than the constant 4
```

因为在描述中找到的唯一变量？x 同样在该模式之内。

正如前面所述，模式网络是按层次分布的，顶部是与模式第一槽值约束相对应的模式结点。当声明一个事实时，会检查模式网络中的第一槽值约束的模式结点。任何满足了匹配说明的模式结点会直接激活它以下的模式结点。这一过程将持续下去直到模式网络的终端结点。模式网络的终端结点表示一个模式的结尾和一次成功的模式匹配。每一终端结点有一个 alpha 或 right memory，alpha memory 包含与终端结点有关、已与模式匹配的所有事实的集合。换句话说，alpha memory 存储了某一特定模式的模式匹配集。

通过共享模式间的公共模式结点，模式网络利用了结构相似性的优点。因为模式结点是按层次存储的，所以，如果两个模式的头 N 个槽值约束的匹配说明相同，则它们共享头 N 个模式结点，例如，模式：

```
(data (x red) (y green))
(data (x red) (y blue))
```

可以共享 x 槽的公共模式结点。注意：匹配说明必须相同，而模式中的槽值约束不必相同。例如，模式：

```
(data (x ?x) (y ?x))
(data (x ?y) (y ?y))
```

可以共享 y 槽的模式结点，尽管这两个模式中的变量不一样。x 槽并不形成匹配说明，所以，为了共享而忽略这些 x 槽。然而，将槽的顺序变为：

```
(data (x ?x) (y ?x))
(data (y ?y) (x ?y))
```

会使模式不能共享结点，因为第一模式会产生 y 槽的匹配说明，而第二模式会产生 x 槽的匹配说明。

图 9.10 示出了由下面规则生成的模式网络。

```
(defrule Rete-rule-1
   (match (a red))
   (data (x ?x) (y ?x))
   =>)

(defrule Rete-rule-2
   (match (a ?x) (b red))
   (data (x ~green) (y ?x))
   (data (x ?x) (y ?x))
   =>)
```

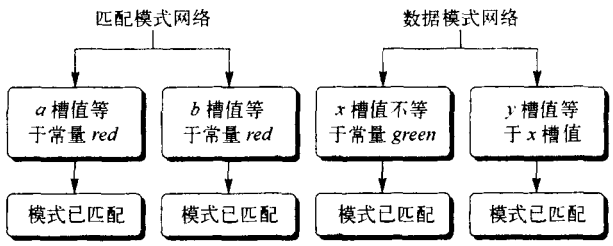


图 9.10 两个规则的模式网络

9.11 连接网络

一旦确定哪些模式与事实匹配，则必须要进行跨模式的变量约束比较以保证多个模式中所用的变量有一致的值。这种比较是在**连接网络**（join network）中实现的。模式网络的每一个终端结点在连接网络中作为一个**连接**（join），即**双输入结点**（two-input node）的一个输入。每一个连接包括一个匹配说明，该匹配说明是为与终端结点相联系的 alpha 内存的匹配和已经与前面模式匹配了的部分匹配集而设置的。前面模式的部分匹配保存在连接的 beta 或 left memory 中。一条有 N 个模式的规则将有 N－1 个连接（但是 CLIPS 实际上用 N 个连接来简化 Rete 算法，因此，每个连接允许只有一个模式输入）。

第一个连接和最先的两个模式比较，剩下的连接将其余的模式与前面连接的部分匹配相比较。例如，假如在前面的例子中使用规则 Rete-rule-2：

```
(defrule Rete-rule-2
   (match (a ?x) (b red))
(data (x ~green) (y ?x))
(data (x ?x) (y ?x))
=>)
```

第一个连接包含如下的匹配说明：

```
The a slot value of the fact
    bound to the first pattern is equal to
the y slot value of the fact
    bound to the second pattern.
```

第二个连接将收到来自第一个连接的一组部分匹配作为输入，它将包含如下的匹配说明：

```
The x slot value of the fact
    bound to the third pattern is equal to
the y slot value of the fact
    bound to the second pattern.
```

注意，第三模式中的变量？x 的值可与第一模式中的变量？x 的值进行比较，而不是与第二模式中的变量？x 的值进行比较。？x 在第三模式中的第二次出现可以在模式网络中被检查到，因此，它不用在连接网络中被检查。规则 Rete-rule-2 的模式和连接网络如图 9.11 所示。

连接网络通过共享规则间的连接来利用结构的相似性。从第一个模式开始，连接网络中的连接可被两条规则共享，前提是这两条规则有相同的模式和两个以上模式的连接比较。例如，这些规则：

```
(defrule sharing-1
   (match (a ?x) (b red))
   (data (x ~green) (y ?x))
   (data (x ?x) (y ?x))
   (other (q ?z))
   =>)

(defrule sharing-2
   (match (a ?y) (b red))
   (data (x ~green) (y ?y))
   (data (x ?y) (y ?y))
   (other (q ?y))
   =>)
```

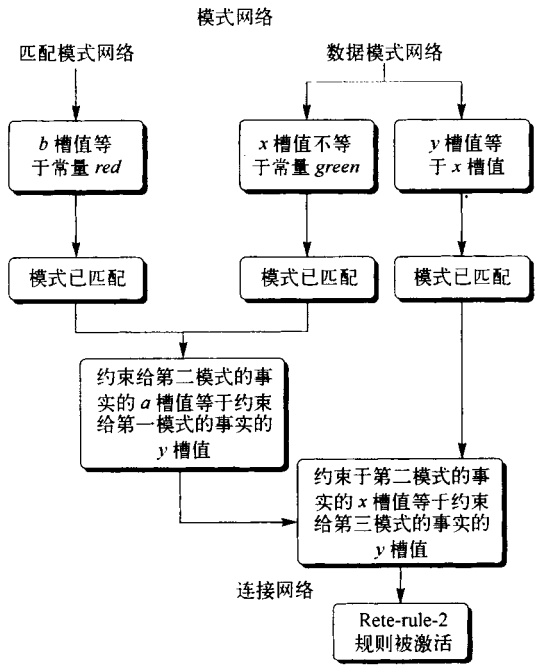


图 9.11 规则 Rete-rule-2 的模式网络和连接网络

能够共享它们在模式网络中的所有模式并共享在连接网络中为前三个模式所生成的连接。在连接网络中第四个模式的连接不能被共享，因为两条规则的匹配说明不相同。规则 sharing-1 的第四模式的匹配说明不需要进行任何比较，因为变量? z 没有在其他模式里使用。然而，规则 sharign-2 的第四模式的匹配说明一定要把变量? y 和其他模式中所使用的? y 进行比较，以保证变量约束的一致性。再次注意，利用结构相似性特点时，变量名不需要相同。最重要的是匹配说明一定要相同。

如果发出 watch compilations 命令，CLIPS 将提供有关共享连接的有用信息。例如，以下命令阐明了如何显示有关共享的信息。假定规则 sharing-1 和 sharing-2 在文件 rules.clp 中。

```
CLIPS> (watch compilations)↵
CLIPS> (load "rules.clp")↵
Defining defrule: sharing-1 +j+j+j+j
Defining defrule: sharing-2 =j=j=j+j
TRUE
CLIPS>
```

输出中的 +j 表示增加一个连接，而 =j 表示一个连接被共享。因此，当加上规则 sharing-1 时，就生成 4 个新的连接，当加上规则 sharing-2 时，它将与 sharing-1 共享头 3 个连接，同时为它最后的模式生成一个新的连接。注意，与通常需要的 3 个连接不同，CLIPS 用 4 个连接来表达规则。由于实现原因，每个连接只有一个模式更方便，所以，CLIPS 在前两个模式中不用一个连接而用两个。

9.12 模式顺序的重要性

为了提高速度和节省内存，规则中的模式要按正确排列，刚使用基于规则的语言的程序员往往会误解这一点。因为 Rete 算法网络保存了从一个循环到下一个循环的状态，所以，确保规则不产生大量的部分匹配是很重要的。例如，考虑以下简单的程序段：

```
(deffacts information
   (find-match a c e g)
   (item a)
   (item b)
   (item c)
   (item d)
   (item e)
   (item f)
   (item g))

(defrule match-1
   (find-match ?x ?y ?z ?w)
   (item ?x)
   (item ?y)
   (item ?z)
   (item ?w)
   =>
   (assert (found-match ?x ?y ?z ?w)))
```

这段程序重置（reset）很快，在 reset 命令前发一个 watch facts 命令将证实这一点。现在考虑以下程序：

```
(deffacts information
   (find-match a c e g)
   (item a)
   (item b)
   (item c)
   (item d)
   (item e)
   (item f)
   (item g))

(defrule match-2
   (item ?x)
   (item ?y)
   (item ?z)
   (item ?w)
   (find-match ?x ?y ?z ?w)
   =>
   (assert (found-match ?x ?y ?z ?w)))
```

在这段程序中，一个后面跟着重置命令的 watch facts 命令将演示一段很慢的重置时间。当重置执行时，自定义事实结构中的头几个事实很快被声明，而随后的事实则需要越来越长的时间。规则 match-1 和 match-2 各自有相同的模式，但 match-2 重置时间更长，实际上，在一些电脑中，规则 match-2 会让 CLIPS 内存不足。通过向 information 自定义事实增加事实，速度的差别会更大（事实上，为了显示明显差别，在一些电脑中必须向 information 自定义事实加入额外的事实）。

计算规则 match-1 的匹配

计算规则 match-1 的模式匹配和部分匹配能提供有用的信息，在列出模式匹配和部分匹配时，使用事实标识，而不用整个事实。而且部分匹配将被括在括号中，事实标识如下：

```
f-1 (find-match a c e g)
f-2 (item a)
f-3 (item b)
f-4 (item c)
f-5 (item d)
f-6 (item e)
f-7 (item f)
f-8 (item g)
```

规则 match-1 包含以下模式匹配：

```
Pattern 1: f-1
Pattern 2: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 3: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 4: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 5: f-2, f-3, f-4, f-5, f-6, f-7, f-8
```

规则 match-1 包含以下部分匹配：

```
Pattern  1:   [f-1]
Patterns 1-2: [f-1,f-2]
Patterns  1-3: [f-1,f-2,f-4]
Patterns  1-4: [f-1,f-2,f-4,f-6]
Patterns  1-5: [f-1,f-2,f-4,f-6,f-8]
```

规则 match-1 总共包含有 29 个模式匹配和 5 个部分匹配。

计算规则 match-2 的匹配

规则 match-2 包含以下模式匹配：

```
Pattern 1: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 2: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 3: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 4: f-2, f-3, f-4, f-5, f-6, f-7, f-8
Pattern 5: f-1
```

规则 match-1 和 match-2 含有相同数量的模式匹配，现在仅考虑模式 1 的部分匹配：

```
[f-2],[f-3],[f-4],[f-5],[f-6],[f-7],[f-8]
```

模式 1 有 7 个部分匹配，这并不奇怪，因为模式匹配和部分匹配在第一个模式中是一样的。然而在模式 1 和 2 中的部分匹配相当多。

```
[f-2,f-2],[f-2,f-3],[f-2,f-4],[f-2,f-5],
          [f-2,f-6],[f-2,f-7],[f-2,f-8],
[f-3,f-2],[f-3,f-3],[f-3,f-4],[f-3,f-5],
          [f-3,f-6],[f-3,f-7],[f-3,f-8],
[f-4,f-2],[f-4,f-3],[f-4,f-4],[f-4,f-5],
          [f-4,f-6],[f-4,f-7],[f-4,f-8],
[f-5,f-2],[f-5,f-3],[f-5,f-4],[f-5,f-5],
          [f-5,f-6],[f-5,f-7],[f-5,f-8],
[f-6,f-2],[f-6,f-3],[f-6,f-4],[f-6,f-5],
          [f-6,f-6],[f-6,f-7],[f-6,f-8],
[f-7,f-2],[f-7,f-3],[f-7,f-4],[f-7,f-5],
          [f-7,f-6],[f-7,f-7],[f-7,f-8],
[f-8,f-2],[f-8,f-3],[f-8,f-4],[f-8,f-5],
          [f-8,f-6],[f-8,f-7],[f-8,f-8]
```

总共有 49 个部分匹配在模式 1 和 2 中（模式 1 中的 7 个乘上模式 2 中的 7 个模式匹配），部分匹配很快会受到空间的限制，因为从模式 1 到模式 3 将有 343 个部分匹配，从模式 1 到模式 4 会有 2401 个部分匹配。而对于规则 match-1，从模式 1 到 5 仅有一个部分匹配存在。

```
[f-2,f-4,f-6,f-8,f-1]
```

注意，尽管对于规则 match-1 和 match-2，其模式匹配及其激活数目都一样多，但规则 match-1 仅有 5 个部分匹配，而 match-2 有 2801 个。随着事实的增多，差别会继续增大。事实（item h）不会为规则 match-1 增加新的部分匹配，但会给 match-2 增加 1880 个。这个例子表明，部分匹配的产生数量会大大影响程序的性能，一个高效的规则集应该不但能尽量减少部分匹配的生成，而且能减少旧部分匹配的删除。实际上，应该尝试尽量减少从一个循环到下一个循环的系统状态变化。减少状态变化的专门技术将在本章稍后讨论。

Matches 命令

CLIPS 有一个调试命令 matches，它能显示规则中的模式匹配、部分匹配及规则的激活。对于查找

生成大量部分匹配的规则和调试那些已匹配好所有模式但却未能激活的规则，这一命令很有用。matches 命令的语法格式是：

```
(matches <rule-name>)
```

matches 命令的参数是其所要显示的规则名，以下的对话说明了 matches 命令的输出情况：

```
CLIPS> (clear)↵
CLIPS>
(defrule match-3
   (find-match ?x ?y)
   (item ?x)
   (item ?y)
   =>
   (assert (found-match ?x ?y)))↵
CLIPS>
(assert (find-match a b)
        (find-match c d)
        (find-match e f)
        (item a)
        (item b)
        (item c)
        (item f))↵
<Fact-6>
CLIPS> (facts)↵
f-0     (find-match a b)
f-1     (find-match c d)
f-2     (find-match e f)
f-3     (item a)
f-4     (item b)
f-5     (item c)
f-6     (item f)
For a total of 7 facts.
CLIPS> (matches match-3)↵
Matches for Pattern 1
f-0
f-1
f-2
Matches for Pattern 2
f-3
f-4
f-5
f-6
Matches for Pattern 3
f-3
f-4
f-5
f-6
Partial matches for CEs 1 - 2
f-1,f-5
f-0,f-3
Partial matches for CEs 1 - 3
f-0,f-3,f-4
Activations
f-0,f-3,f-4
CLIPS>
```

规则 match-3 的第一个模式有 3 个模式匹配，每个 find-match 事实一个。类似地，第二、三个模式都有 4 个模式匹配，每个 item 事实一个。头两个模式包含 2 个部分匹配：一个与事实（find-match c d）和（item c）对应，另一个与事实（find-match c d）和（item c）对应。由于事实（item e）不存在，因此没有与事实（find-match e f）对应的部分匹配。对全部 3 个模式仅有一个部分匹配存在：对应着事实（fact-match a b）、（item a）和（item b）的部分匹配。这个部分匹配还有一个激活，一旦 match-3 规则触发了这个激活，就不会被 matches 命令显示出来。

监视状态变化

matches 命令提供了一种有效方法检查规则中的部分匹配。另一种监视部分匹配的方法是将它们作

为规则的部分激活来考虑。如果规则 match-1 被看作是若干个独立分开的规则，每个都计算部分匹配，那么当部分匹配生成时，我们就可以用 watch activations 命令来监视部分匹配。规则 match-1 能分成以下几个规则：

```
(defrule m1-pm-1
         "Partial matches for pattern 1"
   (find-match ?x ?y ?z ?w)
   =>)

(defrule m1-pm-1-to-2
         "Partial matches for patterns 1 and 2"
   (find-match ?x ?y ?z ?w)
   (item ?x)
   =>)

(defrule m1-pm-1-to-3
         "Partial matches for patterns 1 to 3"
   (find-match ?x ?y ?z ?w)
   (item ?x)
   (item ?y)
   =>)

(defrule m1-pm-1-to-4
         "Partial matches for patterns 1 to 4"
   (find-match ?x ?y ?z ?w)
   (item ?x)
   (item ?y)
   (item ?z)
   =>)

(defrule match-1 "Activations for the match rule"
   (find-match ?x ?y ?z ?w)
   (item ?x)
   (item ?y)
   (item ?z)
   (item ?w)
   =>
   (assert (found-match ?x ?y ?z ?w)))
```

如果调入以上规则和 information 自定义事实，则当部分匹配生成时，我们可以用以下命令对话来监视到它们：

```
CLIPS> (watch activations)↵
CLIPS> (watch facts)↵
CLIPS> (reset)↵
==> f-0     (initial-fact)
==> f-1     (find-match a c e g)
==> Activation 0     m1-pm-1: f-1
==> f-2     (item a)
==> Activation 0     m1-pm-1-to-2: f-1,f-2
==> f-3     (item b)
==> f-4     (item c)
==> Activation 0     m1-pm-1-to-3: f-1,f-2,f-4
==> f-5     (item d)
==> f-6     (item e)
==> Activation 0     m1-pm-1-to-4: f-1,f-2,f-4,f-6
==> f-7     (item f)
==> f-8     (item g)
==> Activation 0     match-1: f-1,f-2,f-4,f-6,f-8
CLIPS>
```

如果对规则 match-2 使用同样技术，将产生数以百计的部分激活。从效率的观点看，现在显然不能把规则的 LHS 看成是事物的全部。每个规则的 LHS 应被视为若干条独立的规则，各规则为整体生成一系列部分匹配。编写高效的规则不但要求限制规则中激活的总数，而且要求限制组成规则的 LHS 的各个独立的子规则的部分匹配的数量。

9.13 排列模式以求高效

当我们确定模式的次序以限制产生的部分匹配数量时，应遵循几个基本原则。要确定最佳次序是很困难的，因为这些原则间会产生冲突。总的来说，这些原则是为了避免基于规则的系统在总体上出现低效率。调试好一个专家系统需要反复对模式进行重新排序，以确定哪些改变能令系统运行更快。通常，尝试完全不同的途径会比模式微调带来更好的效果。

最特定的模式优先（Most Specific Patterns Go First）

最特定的模式应放在规则的 LHS 前面。特定模式通常只含有最少数量的事实表中的匹配事实，而含有最多数量的限制其他模式的变量约束。在规则 match-1、match-2 中模式（match ? x ? y ? z ? w）是最特定的，因为它限制了为规则的其他 4 个模式生成部分匹配的事实。

对应多变事实的模式最后（Patterns Matching Volatile Facts Go Last）

与经常在事实表中增加或删除的事实相对应的模式应该置于规则的 LHS 后面。这将尽可能少地引起部分匹配的变化。但必须注意，与多变事实对应的模式往往是规则中最特定的模式。这给最大效率地安排模式次序带来困难。例如，通常将控制事实作为开始模式是有利的。如果控制事实不出现，就没有部分匹配生成。然而，如果控制事实被定义和撤销太过频繁，不断重复计算大量的部分匹配，那么，把控制事实放在规则最后反而会更高效些。

对应事实最少的模式优先（Patterns Matching the Fewest Facts Go First）

把对应事实很少的模式放到规则中靠前的位置将减少部分匹配的生成数量。再说明一次，利用这一原则可能会与其他原则发生冲突。一个对应极少事实的匹配模式不一定是最特定模式，又或者模式匹配的事实可能是多变的。

9.14 多字段变量与效率

多字段通配符和多字段变量有强大的模式匹配能力。然而，若使用不当，它们也会降低效率。使用多字段通配符和变量时应遵循两个规则。第一，必要时才用。第二，即使使用，也应注意它们在模式的单个槽中的数量。以下规则显示出多字段通配符和变量很有用，但代价也很高：

```
(defrule produce-twoplets
   (list (items $?b $?m $?e))
   =>
   (assert (front ?b))
   (assert (middle ?m))
   (assert (back ?e)))
```

给定一个类似于（list (items a 4 z 2)）的事实，这个规则将生成一个描述表的前、中、后部的事实。可变部分的长度在 0 到表长度之间。这个规则很容易说明多字段变量的使用；然而，这是个代价极高的模式匹配操作。表 9.1 示出了所有被尝试的匹配，也显示了多字段通配符和变量可以完成模式匹配过程中的大量工作。一般而言，若有 N 个字段包含在 item 事实中，将会有 $(N^2+3N+2)/2$ 个匹配在 produce-twoplets 规则中出现。

表 9.1 3 个多字段变量的匹配尝试

匹配尝试	匹配 $? b 的字段	匹配 $? m 的字段	匹配 $? e 的字段
1			a 4 z 2
2		a	4 z 2
3		a 4	z 2

（续）

匹配尝试	匹配$?b的字段	匹配$?m的字段	匹配$?e的字段
4		a 4 z	2
5		a 4 z 2	
6	a		4 z 2
7	a	4	z 2
8	a	4 z	2
9	a	4 z 2	
10	a 4		z 2
11	a 4	z	2
12	a 4	z 2	
13	a 4 z		2
14	a 4 z	2	
15	a 4 z 2		

9.15 测试条件元素与效率

规则中的测试条件元素应尽量靠近规则的顶部。例如，以下的规则尝试寻找3个不同的点：

```
(defrule three-distinct-points
   ?point-1 <- (point (x ?x1) (y ?y1))
   ?point-2 <- (point (x ?x2) (y ?y2))
   ?point-3 <- (point (x ?x3) (y ?y3))
   (test (and (neq ?point-1 ?point-2)
              (neq ?point-2 ?point-3)
              (neq ?point-1 ?point-3)))
   =>
   (assert (distinct-points (x1 ?x1) (y1 ?y1)
                            (x2 ?x2) (y2 ?y2)
                            (x3 ?x3) (y3 ?y3))))
```

测试条件元素确定事实地址？point-1与？point-2不相同，因此，它可被立即置于第二个模式之后。将测试CE放于此处将减少部分匹配的生成。

```
(defrule three-distinct-points
   ?point-1 <- (point (x ?x1) (y ?y1))
   ?point-2 <- (point (x ?x2) (y ?y2))
   (test (neq ?point-1 ?point-2))
   ?point-3 <- (point (x ?x3) (y ?y3))
   (test (and (neq ?point-2 ?point-3)
              (neq ?point-1 ?point-3)))
=>
(assert (distinct-points (x1 ?x1) (y1 ?y1)
                         (x2 ?x2) (y2 ?y2)
                         (x3 ?x3) (y3 ?y3))))
```

当部分匹配在连接网络中生成时，规则LHS上的测试CE总会被求值。如果某些条件满足，在模式匹配过程中，使用了谓词或相等字段约束的表达式可能会被求值。模式网络里，在模式匹配过程中对表达式进行求值会带来更高效率。如果表达式引用的所有变量可以在包含它的模式中找到的话，使用了谓词或返回值字段限制的表达式将在模式匹配过程中被求值。

因为处于测试CE中，以下规则中的表达式将在部分匹配生成时被求值：

```
(defrule points-share-common-x-or-y-value
   (point (x ?x1) (y ?y1))
   (point (x ?x2) (y ?y2))
   (test (or (= ?x1 ?x2) (= ?y1 ?y2)))
   =>
   (assert (common-x-or-y-value
              (x1 ?x1) (y1 ?y1)
              (x2 ?x2) (y2 ?y2))))
```

在模式内放置表达式不会造成在模式匹配时求值，因为变量? x1 和? y1 并未被包含在第二个模式中。

```
(defrule points-share-common-x-or-y-value
   (point (x ?x1) (y ?y1))
   (point (x ?x2) (y ?y2&:(or (= ?x1 ?x2)
                              (= ?y1 ?y2))))
   =>
   (assert (common-x-or-y-value
             (x1 ?x1) (y1 ?y1)
             (x2 ?x2) (y2 ?y2))))
```

同样地，当部分匹配生成时，以下规则中的表达式会被求值，因为该表达式包含在一个测试 CE 中：

```
(defrule point-not-on-x-y-diagonals ""
   (point (x ?x1) (y ?y1))
   (test (and (<> ?x1 ?y1) (<> ?x1 (- 0 ?y1))))
   =>
   (assert (non-diagonal-point (x ?x1) (y ?y1))))
```

然而这一次，在模式中放置表达式将允许它在模式匹配时被求值，因为这两个变量? x1 和? y1 均被包含在表达式的模式中：

```
(defrule point-not-on-x-y-diagonals
   (point (x ?x1)
          (y ?y1&:(and (<> ?x1 ?y1)
                       (<> ?x1 (- 0 ?y1)))))
   =>
   (assert (non-diagonal-point (x ?x1) (y ?y1))))
```

9.16 内置的模式匹配约束

内置的模式匹配约束常常比等价的必须被求值的表达式更有效率。例如，当使用模式匹配约束可以得到同样的结果时，下面的规则

```
(defrule primary-color
   (color ?x&:(or (eq ?x red)
                  (eq ?x green)
                  (eq ?x blue)))
   =>
   (assert (primary-color ?x)))
```

就不应使用，而应使用如下规则：

```
(defrule primary-color
   (color ?x&red|green|blue)
   =>
   (assert (primary-color ?x)))
```

9.17 通用规则与专用规则

较多专用规则是否比较少的通用规则更有效并不总是那么明显。专用规则趋向于将大部分模式网络中的模式匹配过程分离开来，减少连接网络的工作量。而通用规则通常提供更多的模式网络和连接网络内的共享机会。单个规则应该比大群专用规则更易于维护。虽然如此，写通用规则却要十分小心。因为它们要完成多个规则的工作，因此写一个低效的通用规则比写一个低效的专用规则容易。为说明两种技术间的差异，思考以下自定义模板和四个规则，它们可更新包含一个能向东南西北移动的物体的网格坐标的事实。位置事实包含了该物体的 x 和 y 坐标。向北移动将增大 y 坐标值，向东移动将增大 x 坐标值。

```
(deftemplate location (slot x) (slot y))

(defrule move-north
   (move north)
   ?old-location <- (location (y ?old-y))
   =>
   (modify ?old-location (y (+ ?old-y 1))))

(defrule move-south
```

```
   (move south)
   ?old-location <- (location (y ?old-y))
   =>
   (modify ?old-location (y (- ?old-y 1))))

(defrule move-east
   (move east)
   ?old-location <- (location (x ?old-x))
   =>
   (modify ?old-location (x (+ ?old-x 1))))

(defrule move-west
   (move west)
   ?old-location <- (location (x ?old-x))
   =>
   (modify ?old-location (x (- ?old-x 1))))
```

以上4个规则可以由一个更通用的规则、一个附加的自定义模板和一个自定义事实所取代：

```
(deftemplate direction
   (slot which-way)
   (slot delta-x)
   (slot delta-y))

(deffacts direction-information
   (direction (which-way north)
              (delta-x 0) (delta-y 1))
   (direction (which-way south)
              (delta-x 0) (delta-y -1))
   (direction (which-way east)
              (delta-x 1) (delta-y 0))
   (direction (which-way west)
              (delta-x -1) (delta-y 0)))

(defrule move-direction
   (move ?dir)
   (direction (which-way ?dir)
              (delta-x ?dx)
              (delta-y ?dy))
?old-location <- (location (x ?old-x)
                           (y ?old-y))
=>
(modify ?old-location (x (+ ?old-x ?dx))
                      (y (+ ?old-y ?dy))))
```

变量? dx和? dy分别是delta x（Δx）和delta y（Δy）的值，它们要加到旧位置的x和y值上去，以得到新位置的x和y坐标值。

这个新规则在产生部分匹配以决定将delta x和delta y的值加到当前位置上时要做大量工作。然而，它提供了一个抽象层，令增加更多方向变得简单起来。向东北、东南、西北、西南方向移动，需要增加4个新的专用规则。而通用规则只需在自定义事实结构中加4个新的事实，如下所示：

```
(deffacts direction-information
   (direction (which-way north)
              (delta-x 0) (delta-y 1))
   (direction (which-way south)
              (delta-x 0) (delta-y -1))
   (direction (which-way east)
              (delta-x 1) (delta-y 0))
   (direction (which-way west)
              (delta-x -1) (delta-y 0))
   (direction (which-way northeast)
              (delta-x 1) (delta-y 1))
   (direction (which-way southeast)
              (delta-x 1) (delta-y -1))
   (direction (which-way northwest)
              (delta-x -1) (delta-y 1))
   (direction (which-way southwest)
              (delta-x -1) (delta-y -1)))
```

9.18 简单规则与复杂规则

基于规则的语言可以简洁而又优雅地表达许多问题。虽然 CLIPS 不是专门用来解决计算问题的，但它很容易用来寻找一组数字中的最大值。以下规则与相关联的自定义事实将声明一组样本数字作为规则寻找最大数字的测试数据：

```
(deffacts max-num
   (loop-max 100))
(defrule loop-assert
   (loop-max ?n)
   =>
   (bind ?i 1)
   (while (<= ?i ?n) do
      (assert (number ?i))
      (bind ?i (+ ?i 1))))
```

寻找最大数字的最简单的方法已在第 8 章阐述过了。如下所示，我们可以用一条规则来寻找最大数字：

```
(defrule largest-number
   (number ?number1)
   (not (number ?number2&:(> ?number2 ?number1)))
   =>
   (printout t "Largest number is " ?number1 crlf))
```

这条规则虽然很简单，但不是寻找最大数的最快方法。如果 N 代表与 number 有关的事实的数目，则第一个和第二个模式将做 N 次模式匹配。即使最初的两个模式只有一次部分匹配，也要进行 N^2 次比较来找到此部分匹配。将事实 loop-max 的值增加到 200、300、400 等等，可发现运行该问题的时间与 N^2 成比例。

这种类型的比较是极其低效率的，因为每增加一个数字，它都要和其他所有的数字进行比较来确定它是否是最大的。例如，代表数字 2 至 100 的事实已经定义了，现在定义代表数字 1 的事实，则要进行 199 次比较来确定 1 是否是最大的数字。该事实（number 1）符合第一个模式，因而它要和符合第二个模式的 99 个事实进行比较来确定它是否最大。第一次比较就会失败（因为从 2 到 100 的任何一个数字都比 1 大），但它还是要和余下的数字进行比较。同样地，该事实（number 1）符合第二个模式，因而它将和符合第一个模式的 100 个事实进行比较（第一个模式此在也包括该事实（number 1））。

提高程序运行速度的关键是防止不必要的比较发生。要做到这一点，可以使用一个附加的事实来记住最大的数字，将事实 number 与它进行比较。下面的规则说明如何做到这一点：

```
(defrule try-number
   (number ?n)
   =>
   (assert (try-number ?n)))

(defrule largest-unknown
   ?attempt <- (try-number ?n)
   (not (largest ?))
   =>
   (retract ?attempt)
   (assert (largest ?n)))

(defrule largest-smaller
   ?old-largest <- (largest ?n1)
   ?attempt <- (try-number ?n2&:(> ?n2 ?n1))
   =>
   (retract ?old-largest ?attempt)
   (assert (largest ?n2)))

(defrule largest-bigger
   (largest ?n1)
   ?attempt <- (try-number ?n2&:(<= ?n2 ?n1))
   =>
```

```
   (retract ?attempt))

(defrule print-largest
   (declare (salience -1))
   (largest ?number)
   =>
   (printout t "Largest number is " ?number crlf))
```

当 max-loop 的值为 100、200、300、400 等等时，运行这个程序的时间与 N 成比例。更有趣的是第一个程序只激发两条规则，而第二个程序将激发大约 2N 条规则。第二组的规则说明一条规则不仅要尽量限制它拥有的部分匹配的数量，而且要尽量限制用于决定部分匹配所需的比较次数。就像第一条规则所展示的那样，如果第一个和第二个模式都有 N 个匹配，则该规则会做 N^2 次比较来确定最初两个模式的部分匹配。即使没有产生部分匹配，计算时间也大约等于产生了 N^2 次部分匹配的时间。第二组规则限制了比较的次数使之等于 N。既然一次只有一个 largest（最大数字）事实和一个 try-number（尝试数字）事实存在，因此，在任何时候，对于 largest-unknown、largest-smaller 和 largest-bigger 三条规则，也只存在一个部分匹配。N 个 try-number 事实的产生会使计算时间限制在产生 N 次部分匹配的时间范围内。第一眼看去，似乎有可能生成一个以上的 try-number 事实（因此导致有 N^2 次部分匹配）。但是回忆第 9.3 节的内容就会发现，议程工作就像一个堆栈——这意味着通过 loop-assert 规则放在议程中的所有的 try-number（尝试数字）的激活都将在激活 largest-unknown、largest-smaller 和 largest-bigger 规则后触发。因为这些规则总是删除当前的 try-number（尝试数字）事实，所以这种类型的事实决不会超过一个。

这个例子论证了两个重要的概念。第一，用基于规则的语言解决问题的最简单的方法不一定是最好的。第二，通过运用临时事实来存储数据，通常可减少比较的次数。在这个问题中，largest 事实用来存储今后比较的值，使得比较时不必查找所有的 number 事实。

9.19 小结

本章介绍了各种 CLIPS 特性，以利于开发健壮的专家系统。自定义模板属性允许实施类型约束和值约束，从而可以防止书写和语义上的错误。约束检查可以静态地（当表达式或结构被定义时）或动态地（当表达式被求值时）执行。type 属性约束槽的合法类型，allowed value 属性限制某个槽的合法值于某个规定的列表中。range 属性限制数值于某个特定的范围内，cardinality 属性限制存储于一个多字段槽中的最小或最大字段值。另外两个自定义模板属性，即 default 和 default-dynamic 属性，并不约束槽值，但允许指定自定义模板槽的初始值。

优先级提供了一个针对更为复杂的控制结构的机制。它可赋予规则优先级，这样，已被激活且拥有最高优先级的规则能被首先触发。优先级可与控制事实结合使用，以便将专家知识和控制知识区分开来。

自定义模块结构允许将一个知识库分区。通过明确描述可从哪些自定义模板输入、哪些自定义模板可输出到其他模块中去，一个模块可以控制哪些事实可为它所见。使用 focus 命令，程序的执行可以受控，但不必使用优先级属性，而是通过把规则分为不同的组，并将这些组放到不同的模块中去。

本章阐明了有效地将事实和规则进行竞争的重要性。Rete 算法在这种匹配过程中非常有效，因为它利用了基于规则的专家系统所展示的时间冗余性和结构相似性。

在规则网络中，规则被转换为数据结构。这个网络包括了一个模式网络和一个连接网络。模式网络将事实和模式进行竞争，连接网络保证跨模式的变量约束是一致的。模式的排序会对规则的性能产生重要的影响。通常，最特定的和匹配最少事实的模式应该首先放在规则的 LHS 侧，而匹配多变事实的模式应该放在规则的 LHS 侧的后面。命令 matches 用来显示规则的模式匹配、部分匹配和激活。

还有其他几种提高规则效率的技巧，包括正确使用多字段变量、定位测试模式和使用内置的模式匹配约束。此外，在编写与通用规则和专用规则、简单规则和复杂规则有关的规则时，也可以进行效率的平衡。

习题

9.1 修改第 8 章中的 Stick 程序，将控制规则与玩游戏的规则分离。使用优先级属性给控制规则赋予较低的优先级。

9.2 添加一条规则到第 7.23 节的“块世界程序”中，如果某个移动目标已经满足这个规则，则删除该移动目标。

9.3 写出实现判定程序的规则，以确定某三段论是否有效。用习题 8.24 来检验你的这个关于三段论的程序。

9.4 编写程序，确定一个数的质数因子。例如，15 的质数因子是 3 和 5。

9.5 已知德州各城市间的距离如下，求解旅行售货者问题（第 1.13 节）。写一个程序来寻找访问所有城市的最短路径。程序的输入应该是出发的城市和要访问的所有城市列表。用 Houston 作为出发城市，测试你的程序，找出最短路径。

	Houston	Dallas	Austin	Abilene	Waco
Houston	—	241	162	351	183
Dallas	241	—	202	186	97
Austin	162	202	—	216	106
Abilene	351	186	216	—	186
Waco	183	97	106	186	—

9.6 给出以下信息，写一个程序，询问可见到的云的种类和风向，然后，给出会不会下雨的预报。积云意味着天晴，但如果风向是从东北到南的话，它们就会变成乱层云。卷积云且风向是东北到南的话，则本日内有雨。如果风向是北到西，则会出现阴天。如果风向是东北到南，则层积云会变成积雨云。层云意味着下毛毛雨。如果风向是从东北到南，就会发生长时间降雨。如果风向从西南到北，乱层云只会带来短时间降雨。如果风向是东北到南，则意味着长时间的降雨。如果积雨云在午前可见，则预示着阵雨。如果风向是东北到南，则卷层云预示 15 到 24 小时之内会下雨。如果高层云且风向是东北到南，则本日内会下雨，否则会是阴天。高层积云且风向东北到南预示在 15 至 20 小时内会降雨。

9.7 编写一个程序，把摩尔斯（Morse）码信息转换成一系列等价的字母。首先请看范例，范例展示了程序的输入和输出（其中 * 和 - 分别代表点和划，字符/用来分隔摩尔斯码）。

```
Enter a message (<CR> to end): * * * / - - - /
   * * *↵
The message is S O S
Enter a message (<CR> to end): ↵
CLIPS>
```

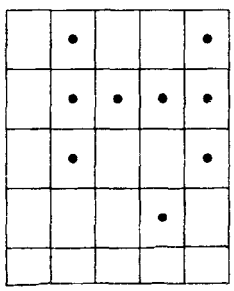
代码和等价字符如下表：

A	·–	H	····	O	–––	V	···–
B	–···	I	··	P	·––·	W	·––
C	–·–·	J	·–––	Q	––·–	X	–··–
D	–··	K	–·–	R	·–·	Y	–·––
E	·	L	·–··	S	···	Z	––··
F	··–·	M	––	T	–		
G	––·	N	–·	U	··–		

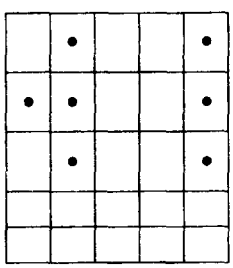
9.8 写一个程序，使给出一个由数字和单位组成的表达式时，能把其中的单位转化成基本单位（如米、秒、千克、便士和安培）。下面的例子说明该程序所要求的输入和输出。

```
Enter an expression (<CR> to end): 30 meters /
 minute↵
Conversion is 0.5 m / s
Enter an expression (<CR> to end): ↵
CLIPS>
```

9.9 写一个程序解决 Life 问题（一个常见的模仿细胞自动机的程序）。假设有一个二维细胞阵列，其中的细胞或者是死的或者是活的，下一代细胞的值取决以下规则：任何与两个或三个其他活细胞相邻的活细胞都能够继续生存；与少于两个或多于三个其他活细胞相邻的活细胞都将会死亡；与三个其他活细胞相邻的死细胞会复活。例如，如果第一代细胞是一个 5x5 的阵列，其中的活细胞用点表示如下：



那么下一代细胞的情况会是这样：



用上图所示的初始配置，计算后面四代的情况。

9.10 四边形是有 4 条边的图形。有两对相连等长边的四边形是筝形（kite）。有一对平行边的四边形是梯形。两对边都平行的四边形是平行四边形。四条边等长的四边形是菱形。四个角都是直角的四边形是矩形。四条边等长且四个角都是直角的四边形是正方形。注意，菱形是筝形的一种，也是平行四边形的一种；平行四边形是梯型的一种；矩形是平行四边形的一种；正方形是矩形的一种，也是菱形的一种。

写一个程序，给出一个四边形的四个顶点，判定它是何种四边形。该程序应能处理舍入误差（如果两条边之间的差小于 0.00001，就认为它们是相等的）。用如下四边形测试你的程序：

(a) 4 个点分别是：(0, 0)、(2, 4)、(6, 0) 和 (3, 2)；

(b) 4 个点分别是：(0, 3)、(2, 5)、(4, 3) 和 (2, 0)；

(c) 4 个点分别是：(0, 0)、(3, 2)、(4, 2) 和 (9, 0)；

(d) 4 个点分别是：(0, 0)、(1, 3)、(5, 3) 和 (4, 0)；

(e) 4 个点分别是：(0, 0)、(3, 5.196152)、(9, 5.196152) 和 (6, 0)；

(f) 4 个点分别是：(0, 0)、(0, 4)、(2, 4) 和 (2, 0)；

(g) 4 个点分别是：(0, 2)、(4, 6)、(6, 4) 和 (2, 0)；

(h) 4 个点分别是：(0, 2)、(2, 4)、(4, 2) 和 (2, 0)。

提示：若边 1 包含点 (x1, y1) 和 (x2, y2)，边 2 包含点 (x3, y3) 和 (x4, y4)，那么，边 1 平行于边 2 的条件是：(x2-x1) * (y4-y3) 等于 (x4-x3) * (y2-y1)。

9.11 写一个 CLIPS 程序，判定一个简单句子的语法是否正确。句子使用如下的 BNF。

```
<sentence> ::= <verb> <direct-object>
                    [<indirect-object>]
<direct-object> ::= [<determiner>] <adjective>* <noun>
<indirect-object> ::= <preposition> <direct-object>
<determiner> ::= a | an | the
<adjective> ::= red | shiny | heavy
<noun> ::= ball | wrench | gun | pliers
<preposition> ::= with | in | at
<verb> ::= get | throw | shoot
```

例如：

```
Enter a sentence (<CR> to end): shoot the red
shiny gun at the pliers↵
OK
Enter a sentence (<CR> to end): gun shoot↵
I don't understand.
Enter a sentence (<CR> to end):
CLIPS>
```

9.12 修改习题 8.12 中的程序，使得只有符合所有要求的灌木才能列出，否则打印信息表示没有符合条件的灌木。

9.13 已知有若干发电机提供电力和有若干台设备使用电力，写一个程序使设备与发电机相连，使所用的发电机的台数最少、且使每一台发电机未被利用的电力最少。例如，由 4 台发电机分别供应 5、6、7 和 10 瓦的电力，有 4 台设备分别消耗 4、5、6 和 7 瓦的电力，则把 5 瓦的设备连至 5 瓦的发电机上，7 瓦的设备连至 7 瓦的发电机上，4 瓦与 6 瓦的设备均连至 10 瓦的发电机上，这样使所用的发电机数量最少，且每一台发电机未被利用的电力最小。程序的输入和输出可以是一系列事实。用上列数据测试你的程序；另外，也用与上相同的发电机，但用耗电为 1、3、4、5 和 9 瓦的设备作为数据测试你的程序。

9.14 写一个程序，使其具有类似计算机操作系统的功能，并确定内存中适合的位置以加载应用程序，这些应用程序要求分配给它们固定数量的内存。程序的输入可以是类似于下面的一系列事实：

```
(launch (application word-processor)
(memory-needed 2))
(launch (application spreadsheet)
(memory-needed 6))
(launch (application game) (memory-needed 1))
(terminate (application word-processor))
(launch (application game) (memory-needed 1))
(terminate (application spreadsheet))
(terminate (application game))
```

假设计算机有 8 Mbyte 内存，每个应用程序使用的内存的兆数为整数。如果内存中恰好符合应用程序需要的空间，就应使用该空间。例如，有两个可用内存块分别为 6 Mbyte 和 4 Mbyte，启动某个应用程序需要使用 4 Mbyte 空间，那么，该应用程序就应被置于此 4 Mbyte 的内存块而非 6 Mbyte 的内存块中。当处理启动或结束命令时，要求打印出一条信息。如果没有足够的内存空间运行某个程序，也要求打印出这种信息。通过声明上面的事实，并在声明完每一个事实后发出 run 命令来测试你的程序。运行结果应类似如下形式：

```
Application word-processor memory location is 1 to 2.
Application spreadsheet memory location is 3 to 8.
Unable to launch application game.
```

```
Terminating word-processor.
Application game memory location is 1 to 1.
Terminating spreadsheet.
Terminating game.
```

9.15 开发出一个文字菜单接口，它自包含在一个模块中，且可以适应于其他应用程序的重用。菜单项可以被表示为事实。菜单项必须支持两种类型的行为：一是可以退出程序的执行，即如果该项类型被选择，程序应停止执行；另一个为声明一个事实并聚焦于一个特定的模块，对该类型行为，菜单项事实应包含能规定焦点模块的槽，还包含当该项被选择时所声明的事实值。例如，在下面对话中，选择 Option A 的结果是聚焦于模块 A 并声明事实（menu-select (value option-a)），其中 A 和 option-a 是在 menu-item 事实中被规定的值。模块 A 包含一条与 menu-select 事实匹配的规则，而致使 Executing Option A 的消息被打印出来。

```
CLIPS>(run)
Select one of the following options(选择下列选项中的一个):
   1—Option A(选项 A)
   2—Option B(选项 B)
   9—Quit Program(退出程序)

   Your choice(你的选择): 1

   Executing Option A

   Select one of the following options(选择下列选项中的一个):
   1—Option A(选项 A)
   2—Option B(选项 B)
   9—Quit Program(退出程序)

   Your choice(你的选择): 9
   CLIPS>
```

9.16 修改习题 8.33 中的程序，使不同模块能分别打印所有有指定光谱级的星星、所有有指定光度的星星、以及所有同时符合以上两个条件并满足地球到这些星星的光年数距离的星星的信息。

9.17 修改习题 8.15 中的程序，以便包括在习题 7.13 的宝石表中的剩余宝石。对于其硬度或密度只有一个数值的宝石，修改这些规则，使得任何指定值在 0.01 之内的值都可以接受。增加规则，使得当硬度不在 1～10 之间、密度不在 1～6 之间时，检测这些情况并重新提示用户。

9.18 下表列出某高中所有课程、教师姓名和上课时间。输入一个事实表示准备要上的课和愿意的教师以及希望有课或无课的时间，写一个程序来建议适当的教师及上课时间。为了决定“最佳”教师及上课时间，要给每一个候选方案评分。规则为：开始分数为 0；愿意的教师和时间各加一分；不愿意的教师和时间各减一分；没有表示愿意与否的教师和空余时间则分数不变。这样，有最高得分的教师和时间则为“最佳”。

课　程	教　师	上课时间
代数	Jones	1，2，3
代数	Smith	3，4，5，6
美国历史	Vale	5
美国历史	Hill	1，2
艺术	Jenkins	1，3，5
生物	Dolby	1，2，5
化学	Dolby	3，6
化学	Vinson	6

（续）

课　程	教　师	上课时间
法语	Blake	2，4
地质	Vinson	1
几何	Jones	5，6
几何	Smith	1
德语	Blake	5
文学	Henning	2，3，4，5，6
文学	Davis	1，2，3，4，5
音乐	Jenkins	2，4
体育	Mack	1，2，3，4，5
体育	King	1，2，3，4，6
体育	Simpson	2，3，4，5，6
物理	Vinson	2，3，5
西班牙语	Blake	1，3
德州历史	Vale	2，3，4
德州历史	Hill	5，6
世界历史	Vale	2，3，4
世界历史	Hill	4

9.19 修改习题8.14中的程序，以便打印配置的总价格。如果选择的小装置比支架数还多，或者，小装置所需的功率比底座提供的还要多，则都打印一则警告信息。

9.20 对于下列各模式的槽，列出所产生的模式结点说明：

(a) `(blip (altitude 100))`

(b) `(blip (altitude ?x&:(> ?x 100)))`

(c) `(stop-light (color ~red))`

(d) `(balloon (color blue|white))`

9.21 画出下列模式的模式网络：

```
(data (x red) (y ?y) (z ?y))
(data (x ~red))
(item (b ?y) (c ?x&:(> ?x ?y)))
(item (a red) (b blue|yellow))
```

9.22 画出下列规则的模式网络和连接网络。列出在每个结点处待计算的表达式：

```
(defrule rule1
  (phase (name testing))
  (data (x ?x) (y ?y))
  (data (x ?y) (y ?x&:(> ?x ?y)))
  =>)

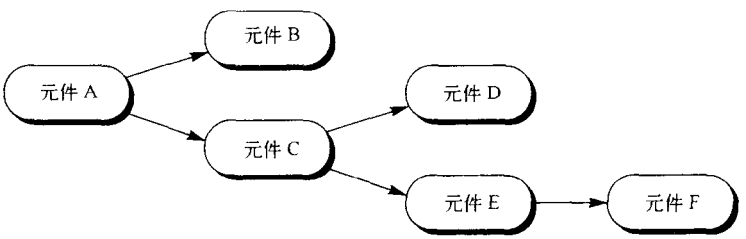
(defrule rule2
  (phase (name testing))
  (data (x ?x) (y ?y))
  (data (x ?y&~red) (y ?x&~green))
  =>)
```

9.23 改写下列规则，使之执行更有效率：

```
(defrule bad-rule
  (items (x ?x1) (y ?y1) (z ?z1))
  (items (x ?x2) (y ?y2) (z ?z2))
  (items (x ?x3) (y ?y3) (z ?z3))
```

```
(test (and (or (eq ?x3 green)
               (eq ?x3 red))
           (eq ?z2 ?y3)
           (> ?y1 ?x1)
           (< ?z1 ?x1)
           (neq ?z3 ?x1)))
=>)
```

9.24 下图示出了一台假想的设备故障网络图。其中的箭头指出故障传播的方向。例如，若元件 A 出故障，则元件 B 和 C 也将出现故障。



下列规则将在网络中传播故障：

```
(defrule propagate-device-A-fault
 (fault device-A)
=>
 (assert (fault device-B))
 (assert (fault device-C)))

(defrule propagate-device-C-fault
 (fault device-C)
=>
 (assert (fault device-D))
 (assert (fault device-E)))

(defrule propagate-device-E-fault
 (fault device-E)
=>
 (assert (fault device-F)))
```

将这 3 条规则改写为一条通用规则和一个自定义事实结构，使之传播故障的方式与上述 3 条规则一样。解释在此故障网络中添加一个新元件时必须对这些方法进行的修改。

参考文献

(Brownston 85). Lee Brownston et al., *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*, Addison-Wesley, 1985.

(Forgy 85). Charles L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, 19, pp. 17–37, 1985.

(Forgy 79). Charles L. Forgy, "On the Efficient Implementation of Production Systems," Ph.D. thesis, Carnegie-Mellon University, 1979.

第 10 章　过程化程序设计

10.1　概述

本章将介绍 CLIPS 提供的类似 C、Ada 和 Pascal 语言的过程化程序设计规范。有时，使用过程化程序执行某些操作比使用基于规则的程序设计更有用（也更有效）。本章首先介绍包括一些允许循环和条件判断操作的过程化函数。自定义函数结构可以用来定义新的函数，这些函数可以被规则或其他函数调用。自定义全局变量结构允许对全局变量进行定义，与在一个规则或函数中定义的局部变量不同，这些变量总是维持它们的值。自定义类属和自定义方法结构允许对类属函数进行定义，类属函数类似于函数，除了其执行依赖于所传递的参数个数和类型外。最后，本章介绍一些有用的实用函数。

10.2　过程化函数

CLIPS 提供了一些函数以控制执行流程。While、if、switch、loop-for-count、progn $ 和 break 函数提供了与现代高级程序语言，如 Ada、Pascal 和 C 类似的功能和控制结构。另外，halt 函数允许规则的执行在其 RHS 上停止。

CLIPS 被设计成一种高效的基于规则的语言。应该审慎使用这些过程化函数。在规则的 RHS 编写冗长的过程程序会令使用基于规则的语言变得毫无意义。通常，在规则的 RHS，这些函数应被用于实现简单的测试和循环控制。应避免在规则的 RHS 出现复杂的这些函数的嵌套结构。

If 函数

If 函数的语法格式是：

```
(if <predicate-expression>
   then <expression>+
   [else <expression>+])
```

其中＜predicate-expression＞是一个独立的表达式（如谓词函数或变量），关键字 if 和 then 后的＜expression＞＋是一个或多个表达式，这些表达式将在＜predicate-expression＞返回值的基础上被求值。注意 else 子句是可选项。

当 if 函数执行时，由＜predicate-expression＞描述的条件将首先被求值，以决定是否执行 then 子句或 else 子句。如果条件不为 FALSE，则执行 then 子句。如果条件为 FALSE，则执行 else 子句。如果没有 else 子句，则在 FALSE 的条件下不执行任何操作。一旦 if 函数执行完毕，且规则 RHS 上有行为，则 CLIPS 将继续执行下一个行为。

不用其他规则来执行测试，而用 if 函数来检查规则 RHS 的值是很有用的。例如，以下规则可用来决定一段程序是否应继续执行：

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? ")
   (bind ?answer (read))
   (if (or (eq ?answer y) (eq ?answer yes)
      then (assert (phase continue))
      else (assert (phase halt))))
```

注意，if 函数将用户的 yes 或 no 回答转变成一个事实，以指示下一步的操作类型。在这种情况下，操作或者继续，或者停止。

If 函数的返回值是函数的 then 或 else 部分中最后一个表达式的值。如果＜predicate-expression＞的值为 FALSE 且没有 then 部分，则返回 FALSE。

While 函数

While 函数的语法格式是：

```
(while <predicate-expression> [do]
   <expression>*)
```

其中＜predicate-expression＞是一个独立表达式(如谓词函数或变量)，而在可选关键字 do 后的＜expression＞＊是 0 个或更多的待求值的表达式，它们求值与否依赖于＜predicate-expression＞的返回值。这些表达式构成一个**循环体** (body)。

在执行循环体前，由＜predicate-expression＞描述的 while 函数部分将首先被求值。如果＜predicate-expression＞值不为 FALSE，则执行循环体中的表达式。如果＜predicate-expression＞的值为 FALSE，则执行 while 函数体后的语句，如果有的话。每次执行循环体前都要检查 while 函数的条件以决定是否再次执行。

While 函数与 if 函数可以同时使用以实现在规则的 RHS 进行输入错误检查。以下对规则 continue-check 的修改利用了 while 函数来继续执行循环，直到得到适当的答案为止：

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? ")
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no))
      do
      (printout t "Continue? ")
      (bind ?answer (read)))
   (if (eq ?answer yes)
      then (assert (phase continue))
      else (assert (phase halt))))
```

switch 函数

Switch 函数的语法格式是：

```
(switch <test-expression>
    <case-statement>*
    [<default-statement>])
```

其中＜case-statement＞定义为：

```
(case <comparison-expression> then <expression>*)
```

＜default-statement＞定义为：

```
(default <expression>*)
```

在 then 和 default 关键字后面的＜expression＞＊是一个或多个表达式，它的值依赖于＜comparison-expression＞的返回值与＜test-expression＞的返回值的匹配关系。注意可选的 default 操作必须在所有的 case 后面。

当执行 switch 函数时，首先计算＜test-expression＞的值。然后按照出现的顺序，计算每个＜comparison-expression＞的值，如果两者相同，则其 then 关键字后的语句被执行，接着 switch 函数结束。如果没有找到匹配值，并且定义了＜default-statement＞，则执行＜default-statement＞。

以下代码使用 switch 函数把一个符号名映射到一个数学函数中：

```
(defrule perform-operation
   (operation ?type ?arg1 ?arg2)
   =>
   (switch ?type
      (case times then
         (printout t ?arg1 " times " ?arg2
                           " is " (* ?arg1 ?arg2)
                                  crlf))
      (case plus then
         (printout t ?arg1 " plus " ?arg2
                           " is " (+ ?arg1 ?arg2)
                                  crlf))
      (case minus then
         (printout t ?arg1 " minus " ?arg2
                           " is " (- ?arg1 ?arg2)
                                  crlf))
      (case divided-by then
         (printout t ?arg1 " divided by " ?arg2
                           " is " (/ ?arg1 ?arg2)
                                  crlf))))
```

例如：

```
CLIPS> (assert (operation plus 3 4))↵
<Fact-1>
CLIPS> (run)↵
3 plus 4 is 7
CLIPS>
```

符号 plus 在操作中激发了 switch 函数中的 plus 情况，所以结果打印出 3 加 4 的和。

Perform-operation 规则可以重写成 4 个独立规则而不需要使用 switch 函数。

```
(defrule perform-operation-times
   (operation times ?arg1 ?arg2)
   =>
   (printout t ?arg1 " times " ?arg2
                     " is " (* ?arg1 ?arg2) crlf))

(defrule perform-operation-plus
   (operation plus ?arg1 ?arg2)
   =>
   (printout t ?arg1 " plus " ?arg2
                     " is " (+ ?arg1 ?arg2) crlf))

(defrule perform-operation-minus
   (operation minus ?arg1 ?arg2)
   =>
   (printout t ?arg1 " minus " ?arg2
                     " is " (- ?arg1 ?arg2) crlf))

(defrule perform-operation-divided-by
   (operation divided-by ?arg1 ?arg2)
   =>
   (printout t ?arg1 " divided by " ?arg2
                     " is " (/ ?arg1 ?arg2) crlf))
```

Loop-For-Count 函数

Loop-For-Count 函数的语法格式是：

```
(loop-for-count <range-spec> [do] <expression>*)
```

其中，<range-spec>定义为：

```
<end-index> |
(<loop-variable> <end-index>) |
(<loop-variable> <start-index> <end-index>)
```

<end-index>和<start-index>是返回整数的表达式。如果只指定了<end-index>，那么函数体<expression>*执行该数值次循环。如果指定了<loop-variable>和<end-index>，那么函数体执行该

数值次循环，而且当前循环，其范围从 1 到＜end-index＞，将会在每次循环中存放在＜loop-variable＞中。如果＜start-index＞也指定了，那么循环将从＜start-index＞开始而不是从 1 开始，循环的次数将是＜end-index＞与＜start-index＞的差加 1。如果＜start-index＞大于＜end-index＞，则循环不执行。

以下是使用＜range-spec＞三个变量和调用 loop-for-count 的例子：

```
CLIPS>
(loop-for-count (?cnt1 2 4) do
   (loop-for-count (?cnt2 3) do
     (printout t ?cnt1 " ")
     (loop-for-count 3 do
        (printout t "."))
        (printout t " " ?cnt2 crlf))).↵
2 ... 1
2 ... 2
2 ... 3
3 ... 1
3 ... 2
3 ... 3
4 ... 1
4 ... 2
4 ... 3
FALSE
CLIPS>
```

一个＜loop-variable＞将不影响所有在 loop-for-count 表达式外定义的同名变量。例如：

```
CLIPS>
(defrule masking-example
   =>
   (bind ?x 4)
   (loop-for-count (?x 2) do
      (printout t "inside ?x is " ?x crlf))
     (printout t "outside ?x is " ?x crlf))
CLIPS> (reset)↵
CLIPS> (run)↵
inside ?x is 1
inside ?x is 2
outside ?x is 4
CLIPS>
```

Progn $ 函数

Progn $ 函数的语法格式是：

```
(progn$ <list-spec> <expression>*)
```

其中，＜list-spec＞定义为：

```
<multifield-expression> |
(<list-variable> <multifield-expression>)
```

如果＜list-spec＞指定为＜multifield-expression＞，那么对计算＜multifield-expression＞所得到的多字段值结果中的每个字段，函数体＜expression＞＊执行一次。如果指定＜list-variable＞和＜multifield-expression＞，则允许通过引用变量获取当前迭代的字段值。同时还通过在＜list-variable＞后附加-index 创建一个特殊的变量，该变量存放当前迭代的索引。函数的返回值是对＜multifield-expression＞最后一个字段执行＜expression＞＊的结果。正如调用 loop-for-count 一样，对 progn $ 的调用可以嵌套，为 progn $ 表达式创建的变量将不影响在 progn $ 表达式外部定义的同名变量。以下是使用两种不同＜list-spec＞的例子：

```
CLIPS>
(progn$ (create$ 1 2 3)
   (printout t . crlf)).↵
.
.
.
```

```
CLIPS>
(progn$ (?v (create$ a b c))
   (printout t ?v-index " " ?v crlf))↵
1 a
2 b
3 c
CLIPS>
```

Break 函数

Break 函数的语法格式是：

```
(break)
```

break 函数中止并结束 while、loop-for-count 和 progn $ 函数的执行。它通常用于当满足某种特殊情形时循环提早结束。例如，以下对话显示的 print-list 规则打印出表的最初 5 个成员，如果还有其他成员则打印……

```
CLIPS>
(defrule print-list
   (print-list $?list)
   =>
   (progn$ (?v ?list)
      (if (<= ?v-index 5)
          then
          (printout t ?v " ")
          else
          (printout t "...")
          (break)))
   (printout t crlf))↵
CLIPS> (reset)↵
CLIPS> (assert (print-list a b c d e))↵
<Fact-1>
CLIPS> (run)↵
a b c d e
CLIPS> (assert (print-list a b c d e f g h))↵
<Fact-2>
CLIPS> (run)↵
a b c d e ...
CLIPS>
```

Halt 函数

Halt 函数可用在规则的 RHS 以停止执行议程中的规则，它不需要任何参数。一旦调用，被激活规则 RHS 的进一步操作将都被中止，控制立即返回顶层。当调用 halt 函数时，议程将包含任何余下的被激活规则。

例如，continue-check 规则可以将以下操作：

```
(assert (phase halt))
```

换成操作：

```
(halt)
```

该 halt 操作会中止规则的执行。

当用户打算稍后用 run 命令重新开始执行操作时，用 halt 函数来中止操作特别有用。思考以下对 continue-check 规则的修改：

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? ")
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no)
      do
```

```
    (printout t "Continue? ")
    (bind ?answer (read)))
  (assert (phase continue))
  (if (neq ?answer yes)
   then (halt)))
```

注意，无论用户对“是否继续”的回答如何，该规则都将声明事实（phase continue）。声明此事实将把合适的规则放到议程中以继续执行操作。如果对“是否继续”的回答不是 yes，执行操作会被 halt 函数中止。然后，用户就可以检查这些规则和事实，之后发出 run 命令在中止的地方重新开始执行。

10.3 自定义函数结构

CLIPS 允许你像在其他过程化语言中一样定义新的函数。对于规则来说，这将有助于减少 LHS 和 RHS 中的重复表达式。新的函数使用**自定义函数**（deffunction）结构来定义。一个自定义函数的一般格式如下：

```
(deffunction <deffunction-name> [<optional-comment>]
   (<regular-parameter>* [<wildcard-parameter>])
   <expression>*)
```

其中，＜regular-parameter＞是一个单字段变量，而＜wildcard-parameter＞是一个多字段变量。自定义函数的名字＜deffunction-name＞必须是不同的，且不能与 CLIPS 中的预先已定义函数重名。自定义函数体由＜expression＞＊表示，是一系列表达式，类似于规则的 RHS，在自定义函数被调用时将依次被执行。用户定义的自定义函数的执行与 CLIPS 提供的预先已定义函数类似。在你可以调用一个 CLIPS 系统提供的预先已定义函数的地方，你都可以调用自定义函数。与预先已定义函数不同的是，自定义函数可以被删除，而且可以使用 watch 命令跟踪它们的执行。

＜regular-parameter＞和＜wildcard-parameter＞允许你指定当自定义函数被调用时传入的参数。这与规则中你在 LHS 中约束变量，在 RHS 中作为参数使用有点类似。正如在规则 RHS 中一样，你可以使用 bind 命令在自定义函数的内部创建局部变量。

正如一些预先已定义函数具有返回值一样，一个自定义函数也可以有返回值。该返回值就是自定义函数内部最后一个表达式的执行结果。

作为一个例子，我们将编写一个使用勾股定理计算直角三角形斜边的长度的自定义函数。如果 a 和 b 是直角边，c 是斜边，则定理表示为：

$$a^2 + b^2 = c^2$$

通过改变公式的形式得到斜边长度：

$$c = \sqrt{a^2 + b^2}$$

把这个公式转化成自定义函数得到：

```
(deffunction hypotenuse-length (?a ?b)
   (** (+ (* ?a ?a) (* ?b ?b)) 0.5))
```

这个函数的两个参数是？a 和？b，用来传递直角三角形的两条直角边。＊＊函数和它的第二个参数 0.5 用来计算开平方根，因为（＊＊ ＜numeric-expression＞ ＜numeric-expression＞）的值是第一个参数的第二个参数次幂。一旦斜边长度函数被定义，它可以在命令行中被调用：

```
CLIPS> (hypotenuse-length 3 4)↵
5.0
CLIPS>
```

因为直角边长度为 3 和 4，所以得到斜边的正确长度为 5。

既然自定义函数可以使用局部变量，对斜边的计算可以采用更可读的形式：

```
(deffunction hypotenuse-length (?a ?b)
   (bind ?temp (+ (* ?a ?a) (* ?b ?b)))
   (** ?temp 0.5))
```

Return 函数

除了可以中止规则 RHS 的执行，return 函数还可以中止当前执行的自定义函数。其在自定义函数中使用的语法如下：

```
(return [<expression>])
```

如果指定了＜expression＞，则其计算结果作为自定义函数的返回值。斜边长度函数可以使用return函数以下面任何一种方法来用返回斜边长度：

```
(deffunction hypotenuse-length (?a ?b)
   (bind ?temp (+ (* ?a ?a) (* ?b ?b)))
   (return (** ?temp 0.5)))
```

或者：

```
(deffunction hypotenuse-length (?a ?b)
(bind ?temp (+ (* ?a ?a) (* ?b ?b)))
(bind ?c (** ?temp 0.5))
(return ?c))
```

不过，既然最后一个表达式的计算结果将作为自定义函数的返回值，而且在斜边长度自定义函数中的各个表达式被顺序执行，那么就没有必要写出明确的返回语句。return 函数主要用于当符合某种条件而终止自定义函数的执行或者当返回值是由某个不处于函数最后的表达式计算而来的场合。例如，以下的自定义函数判断给定的数是否是质数：

```
(deffunction primep (?num)
   (loop-for-count (?i 2 (- ?num 1))
      (if (= ?num (* (div ?num ?i) ?i))
         then
         (return FALSE)))
   (return TRUE))
```

当?num 参数是一个质数时，primep 自定义函数返回 TRUE，否则返回 FALSE。如果一个数的因子只有 1 和它本身，那么它是一个质数。因此如果它能被其他数整除，它就不是质数。primep 自定义函数使用 loop-for-count 函数枚举从 2 到小于本身的所有数来检查?num 能否满足质数条件。如果这些数中有任何一个能整除?num，则函数结束并返回 FALSE，因为它不是质数。Div 函数用来决定?num 的质数条件是否满足。Div 函数执行整数除法返回一个整数。因此，(div 5 2) 返回 2 而不是 2.5，2.5 是 (/ 5 2) 的返回值。如果表达式 (* (div ?num ?i) ?i) 返回最初的?num 值，那么?num 能被?i 整除，?num 不是质数。如果 loop-for-count 函数中枚举的所有值?i 都不能整除?num，那么?num 是质数，自定义函数的返回值为 TRUE。

重写棍子程序

回忆第 8.7 节自定义事实和 3 条规则决定是计算机还是人先走：

```
(deffacts initial-phase
   (phase choose-player))

(defrule player-select
   (phase choose-player)
   =>
(printout t "Who moves first (Computer: c "
                  "Human: h)? ")
      (assert (player-select (read))))

(defrule good-player-choice
   ?phase <- (phase choose-player)
   ?choice <- (player-select ?player&c | h)
   =>
   (retract ?phase ?choice)
   (assert (player-move ?player)))
```

```
(defrule bad-player-choice
?phase <- (phase choose-player)
?choice <- (player-select ?player&~c&~h)
=>
(retract ?phase ?choice)
(assert (phase choose-player))
(printout t "Choose c or h." crlf))
```

因为通常都需要验证输入，我们首先考察一个能够免除 3 条规则这种需要的函数：

```
(deffunction check-input (?question ?values) ;; Line 1
   (printout t ?question " " ?values " ")    ;; Line 2
   (bind ?answer (read))     ;; Line 3
   (while (not (member$ ?answer ?values))    ;; Line 4
      (printout t ?question " " ?values " ") ;; Line 5
      (bind ?answer (read)))                 ;; Line 6
   (return ?answer))                       ;; Line 7
```

从 Line 1 开始定义 check-input 自定义函数，它接受两个参数：?question 是随后显示给用户的问题，?value是对应此问题的可接受的数据列表。Line 2 打印出问题和可接受的数据列表。用户对问题的回答由 Line 3 的 read 函数获取。Line 4 开始执行 while 循环直到用户反馈一个可接受的回答。如果 Member $ 函数的第一个参数是第二个多字段值参数中的一个，则返回 TRUE。如果用户反馈的回答? answer 不是可接受值表? value 中的一个，则 while 循环体执行。Line 5 和 Line 6 构成 while 循环体的内容，重复询问用户。一旦用户反馈一个有效回答，while 循环终止并执行 Line 7，返回用户提供的有效回答。

接下来我们将演示如何使用 check-input 自定义函数。Create $ 函数用于从单字段参数创建一个多字段值，以便作为第二个参数传递给自定义函数：

```
CLIPS>
(check-input "Who moves first, Computer or Human?"
             (create$ c h))↵
Who moves first, Computer or Human? (c h) x↵
Who moves first, Computer or Human? (c h) computer↵
Who moves first, Computer or Human? (c h) c↵
c
CLIPS>
```

使用 check-input 函数，player-select 规则可以重写如下内容，而不再需要 good-player-choice 和 bad-player-choice 规则：

```
(defrule player-select
   ?f <- (phase choose-player)
   =>
   (retract ?f)
   (bind ?player
      (check-input
      "Who moves first, Computer or Human?"
      (create$ c h)))
  (assert (player-move ?player)))
```

递归

自定义函数可以在函数体中调用任何其他自定义函数，包括自己本身。例如，一个正整数 n 的阶乘定义为：

$$\text{factorial (n)} = \begin{cases} n * factorial\ (n-1) & \text{当 } n>1 \\ 1 & \text{当 } n=0 \end{cases}$$

注意到要计算 n 的阶乘必须计算 $n-1$ 的阶乘。所以 3 的阶乘（写作 3!）等于 3×2!，也就是 3×2×1!，等于 3×2×1 也就是 6。以下的自定义函数将计算一个整数的阶乘：

```
(deffunction factorial (?n)
   (if (>= ?n 1)
      then (* ?n (factorial (- ?n 1)))
      else 1))
```

如上所示，(factorial 3) 将会按照3×(factorial 2)计算，也就是3×2×(factorial 1)，最后实际上是计算3×2×1，等于6。这些可以通过直接调用阶乘自定义函数证实：

```
CLIPS> (factorial 3)↵
6
CLIPS> (factorial 2)↵
2
CLIPS> (factorial 1)↵
1
CLIPS>
```

向前声明

有时候自定义函数循环引用另一个函数。例如，自定义函数 A 调用自定义函数B，而 B 调用自定义函数C，C 调用自定义函数A，如下：

```
(deffunction A (?n)
   (if (<= ?n 0)
      then 1
      else (+ 2 (B (- ?n 1)))))

(deffunction B (?n)
   (if (<= ?n 0)
      then 1
      else (* 2 (C (- ?n 1)))))

(deffunction C (?n)
   (if (<= ?n 0)
      then 1
      else (- 2 (A (- ?n 1)))))
```

下面用更清晰的方式来表示这 3 个函数计算的值：

$$A(n)=\begin{cases}1 & n\leqslant 0\\ 2+B(n-1) & n>0\end{cases}$$

$$B(n)=\begin{cases}1 & n\leqslant 0\\ 2\times C(n-1) & n>0\end{cases}$$

$$C(n)=\begin{cases}1 & n\leqslant 0\\ 2-A(n-1) & n>0\end{cases}$$

因为自定义函数必须先被定义后被引用，这就造成一个问题。这些函数互相依赖，不能编译其中的任何一个：

```
CLIPS>
(deffunction A (?n)
   (if (<= ?n 0)
      then 1
      else (+ 2 (B (- ?n 1)))))↵
[EXPRNPSR3] Missing function declaration for B.

ERROR:
(deffunction MAIN::A
   (?n)
   (if (<= ?n 0)
      then
      1
      else
      (+ 2 (B
CLIPS>
```

解决这个问题的方法是为其中某些函数作向前声明。在一个向前声明中给出函数名和参数但是不给出函数体。函数被声明后就可以被引用，但由于还没有函数体所以它不能引用其他函数。向前声明随后将被替换成包含函数体的版本。例如：

```
CLIPS> (deffunction B (?n))↵
CLIPS>
(deffunction A (?n)
   (if (<= ?n 0)
      then 1
      else (+ 2 (B (- ?n 1)))))↵
CLIPS> (deffunction C (?n))↵
CLIPS>
(deffunction B (?n)
   (if (<= ?n 0)
      then 1
      else (* 2 (C (- ?n 1)))))↵
CLIPS>
(deffunction C (?n)
   (if (<= ?n 0)
      then 1
      else (- 2 (A (- ?n 1)))))↵
CLIPS>
```

在这个例子中，在自定义函数要求声明之前插入了向前声明。通常，如果你把规则存放在一个文本文件中调入，你可能会把所有向前声明一起放在文件的头部以便提高代码的可读性和可维护性，但是最主要的是只需保证在自定义函数要求的定义之前声明就可以了。

监视自定义函数

当使用 watch 命令监视自定义函数时，将打印出自定义函数开始和结束时的信息性消息。例如：

```
CLIPS> (watch deffunctions)↵
CLIPS> (factorial 2)↵
DFN >> factorial ED:1 (2)
DFN >> factorial ED:2 (1)
DFN >> factorial ED:3 (0)
DFN << factorial ED:3 (0)
DFN << factorial ED:2 (1)
DFN << factorial ED:1 (2)
2
CLIPS>
```

开始的 DFN 表明是与自定义函数有关。符号>>表示进入一个自定义函数，<<表示退出一个自定义函数。接下来的符号是进入或退出的自定义函数的名字，在上面的例子中是自定义函数 factorial。符号 ED 是 Evaluation Depth（求值深度）的缩写，后面跟着一个冒号和当前的求值深度，一个整数。求值深度表示自定义函数调用的嵌套信息。它从零开始，每次进入一个自定义函数则加一，每次退出一个自定义函数则减一。最后的信息表示传入自定义函数的实际参数。

在这个例子中，factorial 自定义函数最初调用时的参数为 2，调用的求值深度为 1。factorial 自定义函数再次调用以计算 1 的阶乘，此时调用求值深度为 2，参数为 1。接着调用计算 0 的阶乘，调用求值深度为 3，参数为 0。由于 factorial 自定义函数不需要再次递归以计算 0 的阶乘，因此函数不需要再次被调用，嵌套退出。每次退出都使求值深度减一，直到最初的自定义函数调用退出，返回数值 2。

通过在 watch 自定义命令后面加上函数的名字可以监视特定的自定义函数，例如：

```
CLIPS> (unwatch deffunctions)↵
CLIPS> (watch deffunctions C)↵
CLIPS> (A 2)↵
DFN >> C ED:3 (0)
DFN << C ED:3 (0)
4
CLIPS> (watch deffunctions A B)↵
CLIPS> (A 2)↵
DFN >> A ED:1 (2)
DFN >> B ED:2 (1)
DFN >> C ED:3 (0)
DFN << C ED:3 (0)
DFN << B ED:2 (1)
DFN << A ED:1 (2)
4
CLIPS>
```

注意，即使某个自定义函数并没有被监视，其求值深度仍然会被计算。

通配参数

如果自定义函数参数列表中的所有参数都是单字段变量，那么在参数和自定义函数被调用时传入的值之间存在一个一对一的对应关系。也就是说，如果有 3 个参数，那么自定义函数被调用时必须有 3 个值传给函数。

如果自定义函数的最后一个参数是多字段变量，我们称之为通配参数，那么自定义函数接受的参数可以比参数列表中列举的更多。如果参数列表中有 M 个参数，在自定义函数调用中有 N 个实际参数，那么前 M－1 个实际参数对应着参数列表的前 M－1 个参数。其他的从 M 到 N 个实际参数作为一个多字段值对应于参数列表中的第 M 个参数。

作为一个例子，下面重写 check-input 函数使它的最后一个参数? values 成为通配参数：

```
(deffunction check-input (?question $?values)
   (printout t ?question " " ?values " ")
   (bind ?answer (read))
   (while (not (member$ ?answer ?values))
      (printout t ?question " " ?values " ")
      (bind ?answer (read)))
   (return ?answer))
```

最后一个参数被改写为通配参数，因此不再需要使用 create $ 函数来创建多字段值传给自定义函数：

```
CLIPS>
(check-input "Who moves first, Computer or Human?"
c h)↵
Who moves first, Computer or Human? (c h) computer↵
Who moves first, Computer or Human? (c h) human↵
Who moves first, Computer or Human? (c h) h↵
h
CLIPS>
```

在本例中，当调用 check-input 时，字符串 Who moves first, Computer or Human? 被约束给参数? question，余下的参数 c 和 h 转化成多字段值存放到通配参数 $? values 中。

自定义函数命令

ppdeffunction（pretty print deffunction，漂亮打印自定义函数）命令用来显示自定义函数的文本描述。undeffunction 命令用来删除一个自定义函数。list-deffunctions 命令用来显示 CLIPS 中定义的自定义函数列表。get-deffunction-list 函数返回一个包含自定义函数列表的多字段值。这些命令的语法如下：

```
(ppdeffunction <deffunction-name>)

(undeffunction <deffunction-name>)

(list-deffunctions [<module-name>])

(get-deffunction-list [<module-name>])
```

以下例子演示如何使用这些函数：

```
CLIPS> (list-deffunctions)↵
hypotenuse-length
primep
check-input
factorial
A
B
C
For a total of 7 deffunctions.
CLIPS> (get-deffunction-list)↵
(hypotenuse-length primep check-input factorial
A B C)
CLIPS> (undeffunction primep)↵
```

```
CLIPS> (get-deffunction-list)↵
(hypotenuse-length check-input factorial A B C)
CLIPS> (ppdeffunction factorial)↵
(deffunction MAIN::factorial
   (?n)
   (if (>= ?n 1)
      then
      (* ?n (factorial (- ?n 1)))
      else
      1))
CLIPS>
```

当存在引用这个函数的其他自定义函数或者结构时，无法删除这个自定义函数。在这种情况下，只有删除所有引用或者执行 clear 命令才能删除这个自定义函数。例如：

```
CLIPS> (undeffunction A)↵
[PRNTUTIL4] Unable to delete deffunction A.
CLIPS> (deffunction C (?n))↵
CLIPS> (undeffunction A)↵
CLIPS> (undeffunction C)↵
[PRNTUTIL4] Unable to delete deffunction C.
CLIPS> (clear)↵
CLIPS>
```

用户定义函数

除了自定义函数，CLIPS 还提供一个方法调用用 C 语言编写的函数。使用这种方式定义的函数称为**用户定义函数**（user-defined function）。这个名字可以追溯到早期 CLIPS 还未提供自定义函数结构，加入新的函数的惟一方法是用 C 语言编写函数。所以尽管自定义函数也是由用户定义的，也可以称为用户定义函数，但我们只把用 C 语言编写的函数称为用户定义函数。

本书配套光盘中附带的“高级编程指南（Advanced Programming Guide）”讲述了如何创建和添加一个用户定义函数到 CLIPS 中。一般只有两种情况下需要用用户定义函数而不使用自定义函数。第一种情况是你想集成到 CLIPS 中的代码已用 C 语言编写好了。重新编写大量代码没有必要，所以多数情况下在 CLIPS 中就采用用户定义函数的方式调用这些代码。第二种情况是为了提高速度，特别是对大量程序代码而言，执行一个编译成机器代码的 C 函数比在 CLIPS 提供的解释环境下执行一个自定义函数快得多。幸运的是，对大多数任务，在 CLIPS 中可以直接定义自定义函数的便利性弥补了运行解释代码的速度减慢问题。把用户定义函数添加到 CLIPS 中需要创建一个新的 CLIPS 可执行程序。如果你对 C 编译器和编程技术并不熟悉，这可能非常困难。在“高级编程指南”中，除了用户定义函数的例子，你还可以通过阅读 CLIPS 的源代码找到许多用户定义函数，这些都包含在光盘中。所有在第 7 节到第 12 节讨论的函数和命令都可以用用户定义函数的方法加入到 CLIPS 中。

10.4 自定义全局变量结构

CLIPS 允许定义在结构外仍保持其值的变量。这些变量称为**全局变量**（global variable）。直到现在，所有的变量都在结构内使用，称为**局部变量**（local variable）。这些变量局限在使用它们的结构内。例如，思考以下两条规则中的变量？x：

```
(defrule example-1
   (data-1 ?x)
   =>
   (printout t "?x = " ?x crlf))

(defrule example-2
   (data-2 ?x)
   =>
   (printout t "?x = " ?x crlf))
```

？x 在规则 example-1 中的值并不影响？ x 在规则 example-2 中的值。如果事实（data-1 3）被断言，？x在规则 example-1 中被赋予 3。这并不会限制规则 example-2 中的模式只能以？ x 的值为 3 来匹配事实

data-2。而且，断言更多的事实（data-1 4）不会限制在规则 example-1 的两次激活中，? x 的值必须保持一致。基本上每个部分匹配或者规则的激活都有自己的局部变量。这一点同样适用于自定义函数的调用。

全局变量使用**自定义全局变量**（defglobal）结构定义。自定义全局变量结构的一般格式为：

```
(defglobal [<defmodule-name>] <global-assignment>*)
```

其中，<global-assignment>定义为：

```
<global-variable> = <expression>
```

而<global-variable>定义为：

```
?*<symbol>*
```

<defmodule-name>是全局变量所在的模块。如果没有指定，则指当前模块。全局变量的名字以 * 括住，你可以很容易分辨出? x 是一个局部变量，而? * x * 是一个全局变量。当一个自定义全局变量结构被定义时，每个自定义全局变量的初始值由<expression>的计算结果决定，该结果将赋给自定义全局变量。例如：

```
CLIPS>
(defglobal ?*x* = 3
           ?*y* = (+ ?*x* 1))↵
CLIPS> ?*x*↵
3
CLIPS> ?*y*↵
4
CLIPS>
```

注意全局变量? * x * 在? * y * 之前定义，所以可用来决定? * y * 的初始值。此外，由于全局变量在结构外保持它们的值，所以在命令行中输入全局变量名就可以获得其值。除了命令行，在任何可以使用<expression>的地方都可以引用全局变量，这意味着你可以在自定义函数体中或者规则 RHS 中使用它们。但是你不能把它们作为自定义函数的参数，而且只有当包含于函数调用当中时，才可以在规则 LHS 中使用它们。例如，以下结构都是不合法的：

```
(deffunction illegal-1 (?*x* ?y)
   (+ ?*x* y))

(defrule illegal-2
   (data-1 ?*x*)
   =>)

(defrule illegal-3
   (data-1 ?x&~?*x*)
   =>)
```

自定义全局变量命令

自定义全局变量的值可以用 bind 命令改变，只需在原来是局部变量的地方换成全局变量。ppdefglobal（pretty print defglobal，漂亮打印自定义全局变量）命令用来显示自定义全局变量的文本描述。undefglobal 命令用来删除一个自定义全局变量。list-defglobals 命令用来显示在 CLIPS 中定义的自定义全局变量列表。show-defglobals 命令用来显示 CLIPS 中定义的自定义全局变量的名字和值。get-defglobal-list 函数返回一个包含自定义全局变量列表的多字段值。这些命令的语法如下：

```
(ppdefglobal <defglobal-name>)

(undefglobal <defglobal-name>)

(list-defglobals [<module-name>])

(show-defglobals [<module-name>])

(get-defglobal-list [<module-name>])
```

其中，ppdefglobal 和 undefglobal 命令的＜defglobal-name＞必须使用没有 * 括住的全局变量名字（例如，应该用 x 而不是 * x * ）。下面的例子使用了这些函数：

```
CLIPS> (ppdefglobal y)↵
(defglobal MAIN ?*y* = (+ ?*x* 1))
CLIPS> (list-defglobals)↵
x
y
For a total of 2 defglobals.
CLIPS> (get-defglobal-list)↵
(x y)
CLIPS> (show-defglobals)↵
?*x* = 3
?*y* = 4
CLIPS> (bind ?*y* 5)↵
5
CLIPS> (show-defglobals)↵
?*x* = 3
?*y* = 5
CLIPS> (undefglobal y)↵
CLIPS> (list-defglobals)↵
x
For a total of 1 defglobal.
CLIPS>
```

自定义全局变量重置

每当执行 reset 命令，或者执行 bind 命令但不提供一个新值时，自定义全局变量的值恢复到初始值。例如：

```
CLIPS> (bind ?*x* some-value)↵
some-value
CLIPS> ?*x*↵
some-value
CLIPS> (reset)↵
CLIPS> ?*x*↵
3
CLIPS> (bind ?*x* another-value)↵
another-value
CLIPS> ?*x*↵
another-value
CLIPS> (bind ?*x*)↵
3
CLIPS> ?*x*↵
3
CLIPS>
```

通过调用 set-reset-globals 函数设置 FALSE，可以取消全局变量的重置。设置 TRUE 将又恢复重置。当取消重置时，reset 命令不会把全局变量的值恢复为初始值，例如：

```
CLIPS> (bind ?*x* 5)↵
5
CLIPS> (set-reset-globals FALSE)↵
TRUE
CLIPS> (reset)↵
CLIPS> ?*x*↵
5
CLIPS> (set-reset-globals TRUE)↵
FALSE
CLIPS> (reset)↵
CLIPS> ?*x*↵
3
CLIPS>
```

监视自定义全局变量

使用 watch 命令监视自定义全局变量时，每当其值发生改变则会打印出信息性消息。例如：

```
CLIPS> (watch globals)↵
CLIPS> (bind ?*x* 6)↵
:== ?*x* ==> 6 <== 3
6
CLIPS> (bind ?*x* 7)↵
:== ?*x* ==> 7 <== 6
7
CLIPS> (reset)↵
:== ?*x* ==> 3 <== 7
CLIPS> (unwatch globals)↵
CLIPS> (bind ?*x* 8)↵
8
CLIPS>
```

信息性消息前面的：==符号表明出现了一个对全局变量的赋值。随后是被修改的自定义全局变量的名字，紧接着==>和新的值，而<==后面是原来的值。

自定义全局变量和模式匹配

自定义全局变量可以在规则 LHS 的表达式中使用，但对自定义全局变量的修改不会激发模式匹配。例如，思考下面自定义全局变量和自定义规则：

```
(defglobal ?*z* = 4)

(defrule global-example
   (data ?z&:(> ?z ?*z*))
   =>)
```

考虑 data 事实断言后与 global-example 规则的单模式匹配，结果会怎样。

```
CLIPS> (reset)↵
CLIPS> ?*z*↵
4
CLIPS> (assert (data 5) (data 6))↵
<Fact-2>
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (data 5)
f-2     (data 6)
For a total of 3 facts.
CLIPS> (agenda)↵
0      global-example: f-1
0      global-example: f-2
For a total of 2 activations.
CLIPS>
```

当事实（data 5）被断言，局部变量?z 的值约束为 5，该值与全局变量?*z* 的值 4 做比较。因为 5 大于 4，global-example 规则由事实 factf-1 激发。类似的，6 比 4 大，global-example 规则由事实 factf-2 激发。此时，模式匹配完成，改变?*z* 的值不会导致模式匹配重新进行：

```
CLIPS> (bind ?*z* 5)↵
5
CLIPS> (agenda)↵
0      global-example: f-1
0      global-example: f-2
For a total of 2 activations.
CLIPS>
```

把?*z* 的值改为 5，也不会删除由事实 factf-1 引发的行为，即使此时匹配 global-example 规则的事实 data 的值 5 不再大于全局变量?*z* 的值。但如果撤销事实并重新断言，则会激发模式匹配过程，并像断言新的事实一样使用全局变量的新值。

```
CLIPS> (retract 1)↵
CLIPS> (assert (data 5))↵
<Fact-3>
CLIPS> (agenda)↵
0      global-example: f-2
For a total of 1 activation.
```

```
CLIPS> (assert (data 7))↵
<Fact-4>
CLIPS> (agenda)↵
0      global-example: f-2
0      global-example: f-4
For a total of 2 activations.
CLIPS>
```

当事实（data 5），即 f-1 撤销并重新断言时，该事实不再满足 global-example 规则，所以不会产生任何行为。断言事实（data 7）将引发一个行为，因为? z 的值为 7，大于全局变量? *z* 的值 5。

自定义全局变量的使用

自定义全局变量通常用作规则中的常量或者仅在规则 RHS 中传入信息，而不激发模式匹配。作为常量时，自定义全局变量可以使程序更加易懂。考虑下面规则：

```
(defrule plant-advisory
   (temperature ?value Fahrenheit)
   (test (<= ?value 32))
   =>
   (printout t "It's freezing." crlf)
   (printout t "Bring your plants inside." crlf))
```

当然，大多数人知道结冰的温度是华氏 32°（或者摄氏 0°），所以规则 plant-advisory 中的常量 32 是显而易见的。但并不是所有的常量都是常识。通过把一个全局变量赋值为 32，我们可以在 plant-advisory 规则中使用一个有意义的符号使其更加易懂：

```
(defglobal ?*water-freezing-point-Fahrenheit* = 32)

(defrule plant-advisory
   (temperature ?value Fahrenheit)
   (test (<= ?value
   ?*water-freezing-point-Fahrenheit*))
   =>
   (printout t "It's freezing." crlf)
   (printout t "Bring your plants inside." crlf))
```

自定义全局变量不激发模式匹配的一个用途是在调试中。假设你想打印出 watch 命令所不能提供的额外信息。有一种方法是在规则中添加以下的 printout 或者 format 命令：

```
(defrule debug-example
   (data ?x)
   =>
   (printout t "Debug-example ?x = " ?x crlf))
```

这时就出现一个问题，有时你并不想看到这些信息，那么当你不想看到的时候你必须删除掉或者把它们转化成注释。一个便利的解决方法是把输出调试信息的逻辑名字存放在一个自定义全局变量中。如果使用 nil 作为 printout 或者 format 的逻辑参数名，将不会产生任何输出（尽管 format 命令仍然返回格式化字符串作为返回值）。你可以利用这一点取消调试信息。以下演示了如何使用自定义全局变量修改规则 debug-example：

```
(defglobal ?*debug-print* = nil)

(defrule debug-example
   (data ?x)
   =>
   (printout                          ?*debug-print*
   "Debug-example ?x = " ?x crlf))
```

默认地，调试信息将被送到逻辑名字 nil 中，而不会打印在屏幕上。如果你要在屏幕上看到这些信息，只需像下面代码显示的一样，把? * debug-print * 全局变量的值赋为 *t*：

```
CLIPS> (reset)↵
CLIPS> (assert (data a) (data b) (data c))↵
<Fact-3>
CLIPS> (agenda)↵
```

```
0      debug-example: f-1
0      debug-example: f-2
0      debug-example: f-3
For a total of 3 activations.
CLIPS> (run 1)↵
CLIPS> (bind ?*debug-print* t)↵
t
CLIPS> (run 2)↵
Debug-example ?x = b
Debug-example ?x = c
CLIPS>
```

注意，我们在一个 debug-example 被激活之后才使用调试信息。如果我们把调试信息存放在事实中，我们将做不到上面这一点。这种情况下 Debug-example 规则如下：

```
(defrule debug-example
   (debug-print ?debug-print)
   (data ?x)
   =>
   (printout ?debug-print "Debug-example ?x = " ?x
   crlf))
```

撤销 debug-print 事实然后用新值重新断言，将不可避免地重新激发 debug-example，这正是我们不愿看到的。

10.5 自定义类属和自定义方法结构

除了使用自定义函数结构定义函数以外，CLIPS 还提供了类属函数。一个**类属函数**（generic function）本质上是一组相关的函数（称为方法）共用一个名字。事实上，一个包含多于一个方法的类属函数称为加载（overloaded）（因为相同的名字指向不止一个方法）。在组中的每一个方法都有自己的标记：所要求的参数个数和类型。当调用一个类属函数时，会考察参数内容，其标记和之相匹配的方法就是被执行的函数。这个过程称为**类属指派**（generic dispatch）。**自定义类属**（defgeneric）结构用来定义被一组方法共用的公共类属函数名。自定义类属的一般格式为：

```
(defgeneric <defgeneric-name> [<optional-comment>])
```

仅当作为一个向前声明时，你才需要明确定义一个自定义类属，此时，你需要在真正定义一个自定义类属的方法之前引用一个类属函数。如果在自定义类属结构声明之前定义了类属函数的方法，那么会自动产生一个自定义类属。

使用**自定义方法**（defmethod）结构为一个已定义的类属函数指定方法。自定义方法的一般格式为：

```
(defmethod <defgeneric-name> [<index>]
    [<optional-comment>]
    (<regular-parameter-restriction>*
    [<wildcard-parameter-restriction>])
<expression>*)
```

定义自定义方法不需要名字。对组成类属函数的所有方法都使用同一个名字＜defgeneric-name＞。但是每一个特定的方法，都赋予了独一无二的整数作为索引，可以用来引用这个方法。可通过定义方法时指定＜index＞值把一个特定的索引赋给一个方法。如果没有指定＜index＞值，CLIPS 将自动产生一个。对于给定的类属函数，只有一个方法对应特定的参数标识（由＜regular-parameter-restriction＞和＜wildcard-parameter-restriction＞提供的值）。如果为一个类属函数定义了与其他方法有同样参数标识的方法，并且没有指定＜index＞值，那么已存在的方法会被这个新的方法所覆盖，新方法仍使用原来的索引。相反地，如果指定了＜index＞值，那么仅当＜index＞值与具有同样参数标识的原有方法相同时，原有的方法才会被新方法覆盖。

每一个＜regular-parameter-restriction＞可以有两种形式。它可以是一个单字段变量（像在自定义函数中一样），或者是下面形式：

```
(<single-field-variable> <type>* [<query>])
```

第二种形式被圆括号括住，单字段变量后面跟着 0 个或多个类型，然后是一个可选查询。<type>可以是任何合法的类名。详细的内容将在第 11 章介绍，目前的例子将局限于已用作类型属性，但也是类名的值：SYMBOL、STRING、LEXEME、INTEGER、FLOAT、NUMBER、INSTANCE-NAME、INSTANCE-ADDRESS、INSTANCE、FACT-ADDRESS 和 EXTERNAL-ADDRESS。如果指定了最后的<query>值，则必须是一个全局变量或者函数调用。

除了用多字段变量取代了单字段变量外，<wildcard-parameter-restriction>和<regular-parameter-restriction>类似。所以它的两种形式一是多字段变量，二是下面形式：

```
(<multifield-variable> <type>* [<query>])
```

除了<multifield-variable>取代<single-field-variable>外，<type>和<query>值的限制与<regular-parameter-restriction>中一样。当调用一个方法时，为<wildcard-parameter-restriction>指定的多字段变量的作用类似自定义函数的通配参数。方法中超出参数个数的所有剩余参数集中成一个多字段值传递给通配参数变量。

自定义方法的最后一部分<expression>*是方法体。就如同自定义规则的 RHS 或者自定义函数的函数体，它是一系列表达式，在方法被调用时顺序执行。

修订 Check-Input 自定义函数

我们重新回顾第 10.3 节的 check-input 自定义函数。原来的定义是：

```
(deffunction check-input (?question $?values)
   (printout t ?question " " ?values " ")
   (bind ?answer (read))
   (while (not (member$ ?answer ?values))
       (printout t ?question " " ?values " ")
       (bind ?answer (read)))
   (return ?answer))
```

只需要替换结构的名字，这个自定义函数就可以直接转化成自定义方法：

```
(defmethod check-input (?question $?values)
   (printout t ?question " " ?values " ")
   (bind ?answer (read))
   (while (not (member$ ?answer ?values))
    (printout t ?question " " ?values " ")
    (bind ?answer (read)))
(return ?answer))
```

注意不能用同名类属函数替换自定义函数，反之亦然，所以为了定义 check-input 自定义方法，必须删除 check-input 自定义函数。刚才定义的 check-input 自定义方法将起到和原来自定义函数一样的作用。如果不定义更多方法，就没有理由使用类属函数而不用原来的自定义函数。所以下面我们检查棍子程序另一个部分。回忆第 8.9 节，使用了 3 个规则来决定人类玩家移除的棍子数目：

```
(defrule get-human-move
   (player-move h)
   (pile-size ?size)
   (test (> ?size 1))
   =>
   (printout t
   "How many sticks do you wish to take? ")
   (assert (human-takes (read))))

(defrule good-human-move
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (and (integerp ?choice)
              (>= ?choice 1)
              (<= ?choice 3)
              (< ?choice ?size)))
```

```
   =>
   (retract ?whose-turn ?number-taken)
   (printout t "Human made a valid move" crlf))

(defrule bad-human-move
   ?whose-turn <- (player-move h)
   (pile-size ?size)
   ?number-taken <- (human-takes ?choice)
   (test (or (not (integerp ?choice))
             (< ?choice 1)
             (> ?choice 3)
             (>= ?choice ?size)))
   =>
   (printout t "Human made an invalid move" crlf)
   (retract ?whose-turn ?number-taken)
   (assert (player-move h)))
```

由于可以移走的棍子数目不总是1、2、3，因此不能采用下面的调用来询问用户要移走的棍子数目：

```
(check-input " How many sticks do you wish to take?"
 1 2 3)
```

相反的，棍子数目需要动态的计算，就如同以下对 get-human-move 规则的更新：

```
(defrule get-human-move
   (player-move h)
   (pile-size ?size)
   (test (> ?size 1))
   =>
   (bind ?responses (create$))
   (bind ?upper-choice (min (- ?size 1) 3))
   (loop-for-count (?i ?upper-choice)
      (bind ?responses (create$ ?responses ?i)))
      (bind ?answer
      (check-input "How many sticks do you wish to
      take?"
      ?responses))
   (assert (human-takes ?answer)))
```

get-human-move 规则的前4行动态创建了一个多字段值，该值包含了用户可选择的可能值。可选择的值只能是1、2或者3，但它必须比已有的棍子数目小。计算允许的回答是很不方便的，因为当提问题后，所有的可能都被显示。如果 check-input 是询问用户一个1到100之间的数，就会有100个值显示。添加一个额外的方法可以解决这个问题。下面的方法使用类型限制在调用 check-input 后为指定的两个整数采取专门的行为：

```
(defmethod check-input ((?question STRING)
                        (?value1 INTEGER)
                        (?value2 INTEGER))
   (printout t ?question " (" ?value1 "-"
     ?value2 ") ")
   (bind ?answer (read))
   (while (or (not (integerp ?answer))
        (< ?answer ?value1)
        (> ?answer ?value2))
      (printout t ?question " (" ?value1 "-"
        ?value2 ") ")
        (bind ?answer (read)))
   (return ?answer))
```

这个方法和初始的 ask-user 方法有一些共同的代码，但是也有明显的不同，最重要的是参数列表。每一个参数用括号括住，同时包括其类型。为变量? quesiton 指定的类型是 STRING，? value1 是 INTEGER，? value2 是 INTEGER。因为这些限制，这个特定的方法只有在这3个参数都提供给 ask-user 并且第一个是 STRING、第二个是 INTEGER、第三个是 INTEGER 时才会被激发。

方法中的另一个改变是在问题的后面将打印出允许的整数范围，而不是打印出每一个整数。方法的最后一个改变是在 while 循环中用3个测试取代 member $ 测试以保证回答是一个整数并且在允许的范围内。当两个方法都定义好后，下面简单的例子展示了这些方法如何恰当地使用：

```
CLIPS> (check-input "Pick a number" 1 3)↵
Pick a number (1-3) a↵
Pick a number (1-3) 34↵
Pick a number (1-3) 3↵
3
CLIPS> (check-input "Pick a number" 1 2 3)↵
Pick a number (1 2 3) a↵
Pick a number (1 2 3) 34↵
Pick a number (1 2 3) 3↵
3
CLIPS>
```

注意，激发的第一个方法打印出有效范围（1－3），而第二个方法打印出有效值列表（1 2 3）。

方法优先级

当调用一个类属函数时，CLIPS只执行其中一个方法。决定执行哪一个方法的过程称为**类属指派**(generic dispatch)。在前面的例子中，当用一个整数范围而不是允许值列表调用时，check-input类属函数提供了不同的行为。对前面的例子，第二个表达式是：

```
(check-input "Pick a number" 1 2 3)
```

为方法定义的两个标识如下：

```
(?question $?values)
(?question STRING) (?value1 INTEGER)
(?value2 INTEGER)
```

很明显，第一种方法标识可以用来计算这个表达式，因为它需要一个或多个参数，但第二个方法标识不能用来计算该表达式，因为它要求3个参数，但却提供了4个。另一方面，如果我们检查前面例子的第一个表达式：

```
(check-input "Pick a number" 1 3)
```

用它与定义的方法标识比较，很明显两个方法标识都可以处理这个表达式。同样的，第一个方法标识只需要1个或多个任意类型的参数。而第二个方法标识明确需要3个参数：第一个是STRING，第二个是INTEGER，第三个是INTEGER。由于该表达式包含了正确的参数个数和类型，这个方法也可以被使用。那么如果不止一个方法可被使用，CLIPS如何决定使用哪个方法呢?

这个问题的答案是，CLIPS为特定方法定义了一个优先级次序。如果不止一个方法可以使用，则具有较高优先级的将会执行。preview-generic命令用来显示对应某组参数可使用方法的优先级次序。这个命令的语法如下：

```
(preview-generic <defgeneric-name> <expression>*)
```

其中<defgeneric-name>参数是类属函数的名字，而<expression>*是被传递给类属函数的0个或者多个参数。当这个函数执行时不会执行类属函数。所有可使用方法将被检查然后按优先级次序列表。例如：

```
CLIPS>
(preview-generic check-input "Pick a number" 1 3)↵
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #1 () $?
CLIPS>
```

输出验证了我们已经按照经验决定的：整数范围检查的check-input方法比使用通配参数的check-input值检查方法有较高的优先级。方法#2是整数范围检查方法，首先被列出来，这意味着它具有最高的优先级。类属函数名后是方法索引。然后列出每个参数可允许的类型，用括号括住。在自定义方法参数列表中的变量名没有什么作用，所以没有必要列出它们。下一个列出的方法是#1。因为第一个参数没有类型限制，所以显示一对空括号()。余下的参数是通配参数。如果通配参数没有类型限制，那么preview-generic将显示符号$？来表达它。

给定类属函数中的两个方法，CLIPS使用以下步骤来决定哪一个方法有更高的优先级：

1．比较每个方法的最左未检查参数限制。如果只有一个方法有剩余参数，转到步骤6。如果都没

有剩余参数，转到步骤 7。否则，执行步骤 2。

2. 如果一个参数是普通参数，另一个是通配参数，则具有普通参数的方法优先。否则，执行步骤 3。

3. 如果一个参数有类型限制另一个参数没有，则有类型限制的方法优先。否则执行步骤 4。

4. 比较两个参数的最左未检查类型限制。如果一个参数的类型限制比另一个更特定，那么有更特定类型限制的方法优先。例如，类型限制 INTEGER 比 NUMBER 更特定。如果没有一个类型限制更特定（例如，INTEGER 不比 LEXEME 更特定），而且都还有其他类型限制，那么重复步骤 4。如果一个参数有其他类型限制而另一个没有，那么没有额外类型限制的方法优先。否则，执行步骤 5。

5. 如果一个参数有查询限制而另一个没有，那么具有查询限制的方法优先。否则转到步骤 1，比较下一对参数。

6. 如果方法剩余的下一个参数是普通参数，这个方法优先。否则，另一个方法优先。

7. 首先被定义的方法优先。

图 10.1 显示了从步骤 1 到 7 的等价流程图。步骤 1 从图左上角的方框开始。图 10.2 显示了从步骤 3 到 4，决定哪一个参数有更特定参数限制的等价流程图。

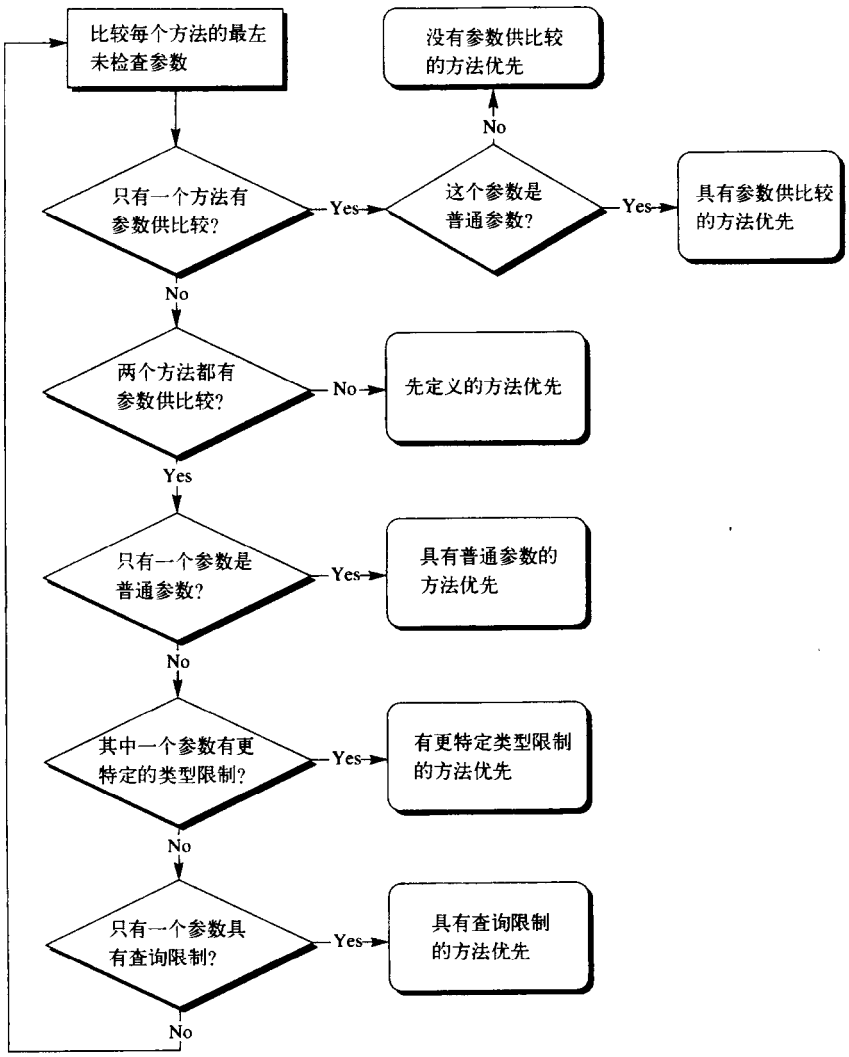


图 10.1 方法优先级判定

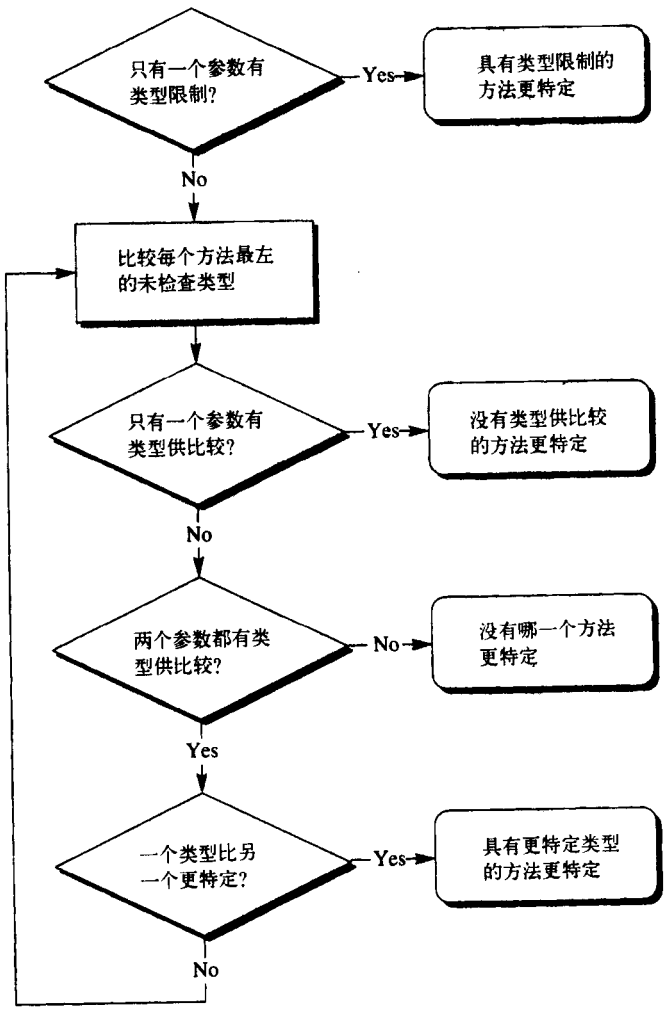


图 10.2 方法类型特定性判定

使用这些步骤我们可以决定为什么类属函数 check-input 中方法 #2 比方法 #1 有更高的优先级。从步骤 1 开始，比较两个方法的最左未检查参数。方法 #1 的参数限制是()而方法 #2 的参数限制是(STRING)。由于两个方法都有参数，执行步骤 2。两个参数都是普通参数，执行步骤 3。只有方法 #2 有类型限制，方法 #2 比方法 #1 有较高优先级。

由于 check-input 方法 #2 只允许我们输入整型数据，我们创建另一个方法以便允许输入指定范围的浮点类型数据：

```
(defmethod check-input ((?question STRING)
                        (?value1 NUMBER)
                        (?value2 NUMBER))
   (printout t ?question " (" (float ?value1)
                 "-" (float ?value2) ") ")
   (bind ?answer (read))
   (while (or (not (numberp ?answer))
              (< ?answer ?value1)
              (> ?answer ?value2))
      (printout t ?question " (" (float ?value1)
                    "-" (float ?value2)
                    ") ")
      (bind ?answer (read)))
   (return ?answer))
```

这个方法和先前的整数范围检查方法类似，除了把参数限制 INTEGER 替换成 NUMBER，它强制在显示有效范围时，任何整型数据都以浮点数形式显示，使用 numberp 而不是 integerp 来检查无效的类型。增加这个方法后，以下例子演示了两个范围检查方法都能正确执行：

```
CLIPS> (check-input "Pick a number" 1 3.5)↵
Pick a number (1.0-3.5) 6↵
Pick a number (1.0-3.5) 2.7↵
2.7
CLIPS> (check-input "Pick a number" 1 3)↵
Pick a number (1-3) 2.7↵
Pick a number (1-3) 2↵
2
CLIPS>
```

preview-generic 命令显示了对于第二个 check-input 类属调用，3 个方法都能适用：

```
CLIPS>
(preview-generic check-input "Pick a number" 1 3)↵
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #3 (STRING) (NUMBER) (NUMBER)
check-input #1 () $?
CLIPS>
```

具有两个 INTEGER 限制的方法比具有两个 NUMBER 限制的方法优先级高。使用优先级判定步骤我们可以明白其中的道理。从步骤 1 开始，比较最左的未检查参数。方法＃2 和方法＃3 都是(STRING)。由于两个方法都有参数，执行步骤 2。两个参数都是普通参数，执行步骤 3。都有类型限制，执行步骤 4。类型限制相同，执行步骤 5。没有参数有查询限制，重新执行步骤 1。方法＃2 的最左未检查参数限制是（INTERGER）而方法＃3 是（NUMBER)。由于都有参数，执行步骤 2。都是普通参数，执行步骤 3。都有类型限制，执行步骤 4。由于方法＃2 的 INTEGER 类型限制比方法＃3 的 NUMBER 类型限制更特定，因此方法＃2 优先。

下面考虑优先级的另一个例子。在这个例子中，我们定义了一个方法，除了用 INTEGER 和 FLOAT 取代原来的 NUMBER 类型限制外，其他和最后一个方法基本上一样：

```
(defmethod check-input ((?question STRING)
                        (?value1 INTEGER FLOAT)
                        (?value2 INTEGER FLOAT))
   (printout t ?question " (" (float ?value1)
               "-" (float ?value2) ") ")
   (bind ?answer (read))
   (while (or (not (numberp ?answer))
              (< ?answer ?value1)
              (> ?answer ?value2))
      (printout t ?question " (" (float ?value1)
                  "-" (float ?value2) ") ")
      (bind ?answer (read)))
   (return ?answer))
```

如果我们监视 preview-generic 调用的结果，会发现新的方法＃4 出现在方法＃2 和＃3 之间：

```
CLIPS>
(preview-generic check-input "Pick a number" 1 3)↵
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #4 (STRING) (INTEGER FLOAT) (INTEGER FLOAT)
check-input #3 (STRING) (NUMBER) (NUMBER)
check-input #1 () $?
CLIPS>
```

这是步骤 4 作用在方法第二个参数上的结果。比较方法＃3 和方法＃4，方法＃4 的第一个类型限制 INTEGER 比方法＃3 的第一个类型限制 NUMBER 更特定，所以方法＃4 有更高的优先级。比较方法＃2 和方法＃4，方法＃2 的第一个类型限制 INTEGER 和方法＃4 的第一个类型限制相同，所以执行第二个类型限制的比较。由于方法＃2 没有第二个类型限制，而方法＃4 有，所以方法＃2 有更高的优先级。

查询限制

除了对参数添加类型限制外，也可以对参数添加查询限制。**查询限制**（query restriction）是一个自定义全局变量引用或函数调用，当一个类属函数被调用时决定方法是否适用。如果查询限制的结果是FALSE，那么方法不可用。对一个参数的查询限制只有在这个参数的类型限制满足了以后才会执行。一个具有多个参数的方法拥有多个查询限制，只有每一个限制都满足了这个方法才可以使用。

下面看一个例子演示查询限制的用处。如果我们调用 check-input 类属函数，但是除了查询字符串外不提供任何其他参数，我们将得到以下结果：

```
CLIPS> (check-input "Pick a number")↵
Pick a number () 3
Pick a number () 2
Pick a number () 5
Pick a number () 0
 .
 .
 .
```

可适用于这种情况的是方法＃1，使用了通配参数。由于在查询字符串参数? question 后没有提供其他参数，通配参数 $? value 约束为一个长度为 0 的多字段值。方法使用的 member $ 测试将总是返回 FALSE，因为用户提供的值不可能包含在空多字段值中，所以方法将永远不能返回。我们可以通过为通配参数添加一个查询限制来解决这个问题：

```
(defmethod check-input
(?question ($?values (= (length$ ?values) 0)))
   (return FALSE))
```

在这个方法中，存放在通配参数 $? values 中的值通过调用 length $ 函数，其返回值用 = 函数与 0 作比较。这就确保了对空多字段值时方法将适用。注意允许在查询限制中使用方法参数（在这个例子中是 ? value）。方法所做的就是返回符号 FALSE，因为用户不可能提供合法的值。有了这个方法，先前的类属调用不再导致无限循环：

```
CLIPS> (check-input "Pick a number")↵
FALSE
CLIPS>
```

用这样的查询字符串参数调用 preview-generic 显示，在所有参数都相同的情况下，方法＃5 的查询限制使它比另一个适用方法＃1 有更高的优先级：

```
CLIPS>
(preview-generic check-input "Pick a number")↵
check-input #5 () ($? <qry>)
check-input #1 () $?
CLIPS>
```

再看另外一种情况，调用 check-input 类属函数出现无限循环：

```
CLIPS> (check-input "Pick a number" 3 1)↵
Pick a number (3-1) 3
Pick a number (3-1) 2
Pick a number (3-1) 5
Pick a number (3-1) 0
 .
 .
 .
```

在这种情况下，可适用的是方法＃2，它具有一个查询串和两个整数参数。由于不存在整数大于或等于 3，并且小于或等于 1，用户永远不能提供满足类属函数的值。我们可以通过创建另外一个带有查询限制的方法，该查询限制检测上限是否小于下限，从而解决这个问题：

```
(defmethod check-input ((?question STRING)
              (?value1 INTEGER)
              (?value2 INTEGER
                   (< ?value2 ?value1)))
   (return FALSE))
```

当发现范围倒置时，这个方法仅仅返回符号 FALSE。我们可以更进一步，对范围值进行交换。完成这项工作的一种方式是复制代码但是在适当的位置交换变量：

```
(defmethod check-input ((?question STRING)
                        (?value1 INTEGER)
                        (?value2 INTEGER
                           (< ?value2 ?value1)))
(printout t ?question " (" ?value2 "-"
                    ?value1 ") ")
(bind ?answer (read))
(while (or (not (integerp ?answer))
           (< ?answer ?value2)
           (> ?answer ?value1))
   (printout t ?question " (" ?value2 "-"
                       ?value1 ") ")
   (bind ?answer (read)))
(return ?answer))
```

这个方法虽然可行，但是重写了太多不必要代码。更好的方法是以倒置后的参数重新调用类属函数：

```
(defmethod check-input
((?question STRING) (?value1 INTEGER)
   (?value2 INTEGER (< ?value2 ?value1)))
   (check-input ?question ?value2 ?value1))
```

在这个方法里面，当参数次序不对时，不会出现无限循环：

```
CLIPS> (check-input "Pick a number" 3 1)↵
Pick a number (1-3) 3
3
CLIPS>
```

为了完整性，我们增加一个方法处理其他用 NUMBER 类型指明范围的方法：

```
(defmethod check-input
((?question STRING) (?value1 NUMBER)
   (?value2 NUMBER (< ?value2 ?value1)))
   (check-input ?question ?value2 ?value1))
```

以倒置范围调用 preview-generic 将会显示出合理的可用方法：

```
CLIPS>
(preview-generic check-input "Pick a number" 3 1)↵
check-input #6 (STRING) (INTEGER) (INTEGER <qry>)
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #4 (STRING) (INTEGER FLOAT)
          (INTEGER FLOAT)
check-input #7 (STRING) (NUMBER) (NUMBER <qry>)
check-input #3 (STRING) (NUMBER) (NUMBER)
check-input #1 () $?
CLIPS>
```

监视类属函数和方法

当使用 watch 命令监视类属函数时，会在函数开始和结束时打印信息性消息。同样当使用 watch 命令监视类属方法时，会在方法开始和结束时打印信息性消息。例如：

```
CLIPS> (watch methods)↵
CLIPS> (watch generic-functions)↵
CLIPS> (check-input "Pick a number" 3 1)↵
GNC >> check-input ED:1 ("Pick a number" 3 1)
MTH >> check-input:#6 ED:1 ("Pick a number" 3 1)
GNC >> check-input ED:2 ("Pick a number" 1 3)
MTH >> check-input:#2 ED:2 ("Pick a number" 1 3)
Pick a number (1-3) 2↵
MTH << check-input:#2 ED:2 ("Pick a number" 1 3)
GNC << check-input ED:2 ("Pick a number" 1 3)
MTH << check-input:#6 ED:1 ("Pick a number" 3 1)
GNC << check-input ED:1 ("Pick a number" 3 1)
2
CLIPS>
```

信息性消息开头的 GNC 表示和类属函数相关。符号>>表示激发了一个类属函数而符号<<表示对类属函数的激发已经结束。下一个符号是类属函数的名字，在本例中是 check-input 类属函数。符号 ED 是 Evaluation Depth（求值深度）的缩写，后边跟着一个冒号和当前的求值深度，一个整数。求值深度表示类属函数和自定义函数调用的嵌套信息。它从零开始，每次进入一个自定义函数或者类属函数则加一。每次退出一个自定义函数或者类属函数则减一。最后的信息表示类属函数的实际参数。以 MTH 开头的是方法信息，包括了和类属函数本质上一样的内容。主要不同是在类属函数名后增加了方法索引，表示正在执行的特定方法。

在本例中，你可以看到两个方法完成执行。方法 #6 首先进入，因为结束范围参数 1 比开始范围参数 3 小。交换参数后，Check-input 类属函数再次被调用。由于参数次序正确，方法 #2 得以执行。用户输入值 2，是在允许的范围内，两个方法都得到返回。一般监视方法比监视类属函数更有用，因为你想知道正在执行的是哪一个方法，不过可以同时都看。

自定义方法命令

有几个命令可以操作自定义方法。ppdefmethod（pretty print defmethod，漂亮打印自定义方法）命令用来显示自定义方法的文本描述。undefmethod 命令用来删除自定义方法。list-defmethods 命令用来显示 CLIPS 中定义的自定义方法列表。get-defmethod-list 函数返回一个包含自定义方法列表的多字段值。这些命令的语法如下：

```
(ppdefmethod <defgeneric-name> <index>)

(undefmethod <defgeneric-name> <index>)

(list-defmethods [<defgeneric-name>])

(get-defmethod-list [<defgeneric-name>])
```

ppdefmethod 和 undefmethod 命令与其他结构的命令的不同之处在于除了自定义类属名外，还需指定索引。而 list-defmethods 和 get-defmethod-list 函数没有可选的模块名参数。代之的，可选参数是类属函数名。如果指定了名字，命令只对这个类属函数的方法起作用；否则，将应用于所有类属函数的所有方法。以下例子演示了如何使用这些命令：

```
CLIPS> (ppdefmethod check-input 5)↵
(defmethod MAIN::check-input
   (?question ($?values (= (length$ ?values) 0)))
   (return FALSE))
CLIPS> (undefmethod check-input 4)↵
CLIPS> (list-defmethods)↵
check-input #6 (STRING) (INTEGER) (INTEGER <qry>)
check-input #2 (STRING) (INTEGER) (INTEGER)
check-input #7 (STRING) (NUMBER) (NUMBER <qry>)
check-input #3 (STRING) (NUMBER) (NUMBER)
check-input #5 () ($? <qry>)
check-input #1 () $?
For a total of 6 methods.
CLIPS> (get-defmethod-list check-input)↵
(check-input 6 check-input 2 check-input 7
 check-input 3 check-input 5 check-input 1)
CLIPS>
```

注意用 list-defmethods 命令列出的方法列表以优先级顺序排列。get-defmethod-list 函数的返回值也是同样，它由类属函数名和方法索引对组成。

自定义类属命令

有几个命令可以操作自定义类属。ppdefgeneric（pretty print defgeneric，漂亮打印自定义类属）命令用来显示自定义类属的文本描述。undefgeneric 命令用来删除自定义类属。list-defgeneric 命令用来显示 CLIPS 中定义的自定义类属列表。get-defgeneric-list 函数返回一个包含自定义类属列表的多字段值。

这些命令的语法如下：

```
(ppdefgeneric <defgeneric-name>)

(undefgeneric <defgeneric-name>)

(list-defgenerics [<module-name>])

(get-defgeneric-list [<defgeneric-name>])
```

undefgeneric 命令不仅删除指定的自定义类属，还删除与这个类属相关联的所有方法，如下面命令所示：

```
CLIPS> (list-defgenerics)↵
check-input
For a total of 1 defgeneric.
CLIPS> (undefgeneric check-input)↵
CLIPS> (list-defgenerics)↵
CLIPS> (list-defmethods)↵
CLIPS>
```

加载函数和命令

自定义函数和类属函数不能共用一个相同的名字，但是类属函数可以加载用户定义函数。假设你想为 NUMBER 外的其他数据类型提供类似的操作，正常地，CLIPS 不允许你这样做：

```
CLIPS> (+ 3 4)↵
7
CLIPS> (+ "red" "blue")↵
[ARGACCES5] Function + expected argument #1 to be
of type integer or float
CLIPS> (+ (create$ a b c) (create$ d e f))↵
[ARGACCES5] Function + expected argument #1 to be
of type integer or float
CLIPS>
```

尝试把两个字符串或两个多字段值用 + 相加会产生错误。如果我们要实现把字符串连接起来以及把多字段值组合的加法运算，则可以通过定义方法使用 str-cat 和 create $ 函数来完成：

```
(defmethod + ((?x LEXEME) (?y LEXEME))
   (str-cat ?x ?y))

(defmethod + ((?x MULTIFIELD) (?y MULTIFIELD))
   (create$ ?x ?y))
```

这些方法一旦定义，我们就可以相加这些新的数据类型而不会产生错误：

```
CLIPS> (+ "red" "blue")↵
"redblue"
CLIPS> (+ (create$ a b c) (create$ d e f))↵
(a b c d e f)
CLIPS>
```

list-defmethods 命令可以列出已定义的两个新方法以及 CLIPS 中定义的原始 + 函数，它用 SYS1 方法索引指明：

```
CLIPS> (list-defmethods +)↵
+ #SYS1 (NUMBER) (NUMBER) ($? NUMBER)
+ #2 (LEXEME) (LEXEME)
+ #3 (MULTIFIELD) (MULTIFIELD)
For a total of 3 methods.
CLIPS>
```

注意，当创建了方法标识后，CLIPS 可以决定系统定义的 + 函数的数据类型。在这个例子中，原 + 函数要求两个或更多数值参数。

如果你想加载一个用户定义函数，必须在定义任何引用此用户定义函数的结构之前加载它。你可以通过明确定义一个自定义类属或在函数引用之前定义一个方法来隐含地定义一个自定义类属来做到

这一点。在用户定义函数加载前进行的结构引用，不会使用类属指派机制，将总是调用用户定义函数。在用户定义函数加载后进行的结构引用，将会使用类属指派机制。

10.6 过程化结构和自定义模块

和自定义模板结构类似，自定义全局变量、自定义函数和自定义类属结构可以用模块输入和输出。在第 9 章中讨论的与自定义模板有关的 4 种可能的输入输出语句，同样可以应用在这些过程化结构上：

```
(export ?ALL)
(export ?NONE)
(import <module-name> ?ALL)
(import <module-name> ?NONE)
```

第一种格式将从一个模块中输出所有可输出的结构；第二种格式表示没有结构输出；第三种格式输入指定模块的所有可输出结构；第四种格式表示在指定模块中没有输出结构输入。

自定义全局变量、自定义函数和自定义类属结构都有相应的输入/输出语句可以规定输入/输出所有、没有或指定的一组结构：

```
(export defglobal ?ALL)
(export defglobal ?NONE)
(export defglobal <defglobal-name>+)

(export deffunction ?ALL)
(export deffunction ?NONE)
(export deffunction <deffunction-name>+)

(export defgeneric ?ALL)
(export defgeneric ?NONE)
(export defgeneric <defgeneric-name>+)

(import <module-name> defglobal ?ALL)
(import <module-name> defglobal ?NONE)
(import <module-name> defglobal
                      <defglobal-name>+)

(import <module-name> deffunction ?ALL)
(import <module-name> deffunction ?NONE)
(import <module-name> deffunction
                      <deffunction-name>+)

(import <module-name> defgeneric ?ALL)
(import <module-name> defgeneric ?NONE)
(import <module-name> defgeneric
                      <defgeneric-name>+)
```

当输入/输出特定自定义全局变量时，不能包含自定义全局变量名的开始和结束符号 * 。例如，使用 water-freezing-point-Fahrenheit，而不是 * water-freezing-point-Fahrenheit * 。输入自定义类属结构到一个模块同时也输入其所有方法。不能输入或输出某个特定的方法。一个没有从别的模块输入自定义全局变量、自定义函数或自定义类属结构的模块可以用相同的名字创建该结构。引用一个没有正确输入或输出的结构会产生错误，例如：

```
CLIPS> (clear)↵
CLIPS>
(defmodule MAIN
   (export deffunction function-1)
   (export defglobal global-1)
   (export defgeneric generic-1))↵
CLIPS>
(deffunction MAIN::function-1 (?x)
   (+ 1 ?x))↵
CLIPS>
(deffunction MAIN::function-2 (?x)
   (+ 2 ?x))↵
CLIPS>
```

```
(defglobal MAIN ?*global-1* = 1
  ?*global-2* = 2)↵
CLIPS>
(defmodule EXAMPLE
   (import MAIN deffunction ?ALL)
   (import MAIN defglobal global-1))↵
CLIPS>
(defglobal EXAMPLE ?*global-2* = 3)↵
CLIPS>
```

EXAMPLE 模块通过关键字? ALL 从 MAIN 模块隐式地输入自定义全局变量 global-1。通过指定名字显式地从 MAIN 模块输入自定义函数 function-1。输入输出语句的结果可以通过存取每个模块的结构看到：

```
CLIPS> (get-current-module)↵
EXAMPLE
CLIPS> (function-1 1)↵
2
CLIPS> (function-2 1)↵
[EXPRNPSR3]  Missing  function  declaration  for
function-2.
CLIPS> ?*global-1*↵
1
CLIPS> ?*global-2*↵
3
CLIPS> (set-current-module MAIN)↵
EXAMPLE
CLIPS> (function-1 1)↵
2
CLIPS> (function-2 1)↵
3
CLIPS> ?*global-1*↵
1
CLIPS> ?*global-2*↵
2
CLIPS>
```

10.7 有用的命令和函数

调入和保存事实

一个 CLIPS 程序的运行速度可通过减少事实列表中的事实数来提高。减少事实数的一个方法是，当需要用到这些事实时才将它们装进 CLIPS 中。例如，一个用来诊断汽车故障的程序也许首先要询问汽车制造商和汽车型号，然后才将该车的特定信息装到程序中。CLIPS 提供了函数 load-facts 和 save-facts，它们允许事实从文件中读出或保存到文件中。这两个函数的语法是：

```
(load-facts <file-name>)

(save-facts <file-name>
            [<save-scope> <deftemplate-names>*])
```

其中<save-scope>定义为：

```
visible | local
```

函数 load-facts 将装入一组存于<file-name>指定的文件中的事实。文件中的事实应该符合标准格式，即，或者是一个有序事实，或者是一个自定义模板事实。例如，如果文件 facts.dat 包含：

```
(data 34)
(data 89)
(data 64)
(data 34)
```

则命令

```
(load-facts "facts.dat")
```

将调入该文件所包含的事实。

函数 save-facts 用于将事实列表中的事实保存到＜file-name＞指定的文件中。这些事实的存储格式由 load-facts 函数规定。如果没有说明＜save-scope ＞或它指定为 local，则只有那些对应着当前模块中已定义的自定义模板事实才能够保存到此文件中。如果＜save-scope＞指定为 visible，则所有对应于当前模块可见的自定义模板事实被保存到文件中。如果指定了＜save-scope＞，则也可以指定一个或以上的自定义模板名。在这种情况下，只有和特定自定义模板对应的事实才被保存（但自定义模板名仍须满足 local 或 visible 说明）。

如果能够成功打开，然后调入或保存该事实文件，则 load-facts 和 save-facts 返回 TRUE；否则，返回 FALSE。当 load-facts 命令执行时，程序员必须确保在事实文件中的自定义模板事实所对应的自定义模板对于当前模块来说是可见的。

System 命令

System 命令允许从 CLIPS 中调用操作系统命令。命令的语法为：

```
(system <expression>+)
```

例如，下面的规则可以显示出使用 Unix 操作系统的机器上指定目录的目录表：

```
(defrule list-directory
(list-directory ?directory)
   =>
   (system "ls " ?directory))
```

对于本例，system 命令的第一个参数“ls”是 UNIX 系统的列目命令。注意格式中字符后有一个空格。在允许操作系统处理该命令之前，system 命令只是将所有的参数一起作为一个字符串。操作系统所需的任何空格都必须作为 system（系统）命令调用的一部分包容进去。

对于不同的操作系统，system 命令的作用可以不同。并非所有的操作系统都提供实现 system 命令的功能，所以，你不能指望此命令能在 CLIPS 中运用。而且 system 命令并不返回任何值，所以，在执行一个操作系统命令之后，不可能直接将值返回给 CLIPS。

Batch 命令

batch（批处理）命令允许命令和响应直接从文件中读入，这些命令和响应一般是在顶层提示符下输入的。batch 命令的语法如下：

```
(batch <file-name>)
```

例如，设想下面的对话中的命令和响应必须输入，以运用 CLIPS 程序（记住：粗体字表明你要输入的内容）。

```
CLIPS> (load "rules1.clp")↵
**************
CLIPS> (load "rules2.clp")↵
******************
CLIPS> (load "rules3.clp")↵
***********
CLIPS> (reset)↵
CLIPS> (run)↵
How many iterations? 10↵
Starting value? 1↵
End value? 20↵
Completed
CLIPS>
```

需要运行该程序的命令和响应可存入一个文件中，如下所示：

```
(load "rules1.clp")↵
(load "rules2.clp")↵
(load "rules3.clp")↵
(reset)↵
```

```
(run)↵
10↵
1↵
20↵
```

如果这个包含命令和响应的文件取名为 commands.bat，则下面的对话将表明如何使用 batch 命令：

```
CLIPS> (batch "commands.bat")↵
CLIPS> (load "rules1.clp")↵
**************
CLIPS> (load "rules2.clp")↵
******************
CLIPS> (load "rules3.clp")↵
**********
CLIPS> (reset)↵
CLIPS> (run)↵
How many iterations? 10↵
Starting value? 1↵
End value? 20↵
Completed
CLIPS>
```

一旦所有的命令和响应均从此批文件中读取完，则在顶层提示符下的键盘交互将回到正常状态。

当在支持命令行参数的操作系统（如 Unix）中运行时，CLIPS 能在启动时自动地执行批处理文件中的命令。假定敲入“clips”就能执行 CLIPS 的可执行命令，那么，在启动时执行一个批处理文件的语法如下：

```
clips -f <file-name>
```

一旦 CLIPS 开始运行，则使用-f 选项就等价于输入了命令（batch <file-name>）。对 batch 命令的调用可以嵌套。

Dribble-on 和 Dribble-off 命令

dribble-on 命令可用来把所有输出到终端的记录或从键盘来的所有输入保存起来。其语法如下：

```
(dribble-on <file-name>)
```

一旦执行 dribble-on 命令，则送到终端的所有输出和从键盘键入的所有输入都将会送到由<file-name>指定的文件中，就像送到终端一样。

dribble-on 命令的作用可以用 dribble-off 命令取消，其语法是：

```
(dribble-off)
```

产生随机数

random 函数产生一个随机整数。它的语法如下：

```
(random [<start-expression> <end-expression]])
```

其中，如果指定了<start-expression>和<end-expression>，则必须为整数，表示所返回随机整数受限的整数范围。例如，下面的 roll-die 自定义函数使用 random 函数产生抛六面骰子的随机数，其值在 1 到 6 之间：

```
(deffunction roll-die ()
   (random 1 6))
```

多次调用 roll-die 自定义函数会返回不同的值：

```
CLIPS> (roll-die)↵
6
CLIPS> (roll-die)↵
1
CLIPS> (roll-die)↵
```

```
4
CLIPS> (roll-die)↵
1
CLIPS>
```

seed函数可以用来生成随机数种子，使得下一次通过相同的种子得到相同的随机数。seed函数的语法如下：

```
(seed <integer-expression>)
```

其中，<integer-expression>是种子值。seed命令的作用是重复产生相同的随机数，这可从下面的一组命令中看到：

```
CLIPS> (seed 30)↵
CLIPS> (roll-die)↵
4
CLIPS> (roll-die)↵
5
CLIPS> (roll-die)↵
3
CLIPS> (seed 30)↵
CLIPS> (roll-die)↵
4
CLIPS> (roll-die)↵
5
CLIPS> (roll-die)↵
3
CLIPS>
```

转化字符串为字段

string-to-field函数用来把一个字符串字段转化成一个字段。它的语法如下：

```
(string-to-field <string-expression>)
```

其中，<string-expression>是要解析和转化的字符串字段。例如：

```
CLIPS> (string-to-field "7")↵
7
CLIPS> (string-to-field " 3.4 2.1  3")↵
3.4
CLIPS>
```

第一个string-to-field调用的例子把字符串字段7.3转化成整数字段值7。第二个例子所传入的字符串参数有多个字段，第一个字段被获取，返回浮点数字段3.4，字符串中的其他字段被抛弃。注意字符串中的额外空格不会影响返回值。基本上，调用string-to-field等同于调用read函数，其中的字符串参数代表用户从键盘输入的内容。

查找符号

apropos命令用来显示CLIPS中定义的所有具有指定子串的符号。它的语法如下：

```
(apropos <symbol-or-string-expression>)
```

其中，<symbol-or-string-expression>是待查找子串。当需要显示具有共同子串的函数和命令列表或者你只记得函数、命令和结构的部分而不是整个符号时，apropos命令特别有用。例如，假设你要列出所有包含符号deftemplate的函数和命令：

```
CLIPS> (apropos deftemplate)↵
get-deftemplate-list
deftemplate
list-deftemplates
undeftemplate
ppdeftemplate
deftemplate-module
CLIPS>
```

排序字段表

sort 函数应用于字段表上。其语法如下：

```
(sort <comparison-function-name> <expression>*)
```

其中，<comparison-function-name>用来决定在排序过程中两个字段之间是否需要交换的函数、自定义函数或者类属函数的名。其余由<expression>*表示的参数，可以是单字段或多字段值。它们被拼接成单个多字段值，然后进行排序。函数的返回值是一个排好序的多字段值。下面例子对一个整数表排序：

```
CLIPS> (sort > 4 3 5 7 2 7)↵
(2 3 4 5 7 7)
CLIPS>
```

sort 函数不仅仅限于对数值排序。例如，考虑下面自定义函数：

```
(deffunction string> (?a ?b)
   (> (str-compare ?a ?b) 0))
```

sort 函数中的任何比较函数接受两个参数，然后在第一个参数在排序表中应排在第二个参数的后面时返回 TRUE，否则返回 FALSE。当第一个参数词典排序比第二个参数大的时候，str-compare 函数将返回一个正数。所以通过用>函数比较返回值和 0 的关系，string>自定义函数可以用来对符号或字符串表进行词典排序。例如：

```
CLIPS> (sort string> ax aa bk mn ft m)↵
(aa ax bk ft m mn)
CLIPS>
```

Str-compare 函数认为所有的大写字母都比小写字母小。所以如果你对大小写混合字母排序，结果可能和预期的不同：

```
CLIPS> (sort string> a C b A B c)↵
(A B C a b c)
CLIPS>
```

从输出内容可以看到，所有的大写字母都排在小写字母前面。string>函数可以修改成把大小写字母放在一起：

```
(deffunction string> (?a ?b)
   (bind ?rv (str-compare (lowcase ?a)
   (lowcase ?b)))
   (if (= ?rv 0)
       then
       (> (str-compare ?a ?b) 0)
       else
       (> ?rv 0)))
```

字符串首先被转化成小写字母，然后进行比较。只有当字符串相等时才考虑大小写。这得到了以下显示的预期结果：

```
CLIPS> (sort string> a C b A B c)↵
(A a B b C c)
CLIPS>
```

使用 sort 函数要注意的最后一点是只有当两个值需要交换时比较函数才应返回 TRUE。考虑一下调用：

```
CLIPS> (sort <> 3 4 5 6)↵
...
```

在这个例子中，sort 函数调用<>函数来决定字段是否需要交换。由于各个字段互不相同，<>函数总是返回 TRUE，排序永远不会完成。显然这是你应该避免的情况。

10.8 小结

if、while、switch、loop-for-count、progn $ 和 break 函数可以用在自定义函数、类属函数或规则的

RHS中进行流程控制。在规则RHS中过多使用这些函数是不好的编程风格。Halt函数可以用来终止规则的执行。

自定义函数允许你像在其他过程化语言中一样，定义新的函数，但不需要使用C编译器来重新编译CLIPS源代码。另外，CLIPS也提供了机制可以把C语言写的函数集成到CLIPS中，在“高级编程指南”中有详细介绍，但这需要使用C语言编译器来生成一个新的CLIPS可执行代码。

自定义全局变量结构允许你定义变量，称为全局变量，它们在结构外仍能保持其值。使用自定义类属和自定义方法结构实现的类属函数通过允许以一个相同的名字访问多个过程化方法，而提供比自定义函数更强大和灵活的功能。当一个类属函数被调用时，类属指派机制检查传给函数的参数类型，并计算相关查询限制来决定应该激发的适当方法。

本章介绍了几个实用命令。save-facts和load-facts函数用来保存事实到一个文件或从文件中调入事实。system命令允许从CLIPS中调用操作系统命令。batch命令允许把一系列命令和响应存储在文件中而取代普通的键盘输入。dribble-on和dribble-off命令允许终端输出的记录保存在文件中。random函数用来产生一个随机整数。seed函数用来生成随机数种子。string-to-field函数用来解析字符串中的字段。apropos命令用来显示所有包含指定子串的符号。sort函数用来排序字段表。

习题

10.1 把下面规则重写成一个或多个规则，不使用if和while函数。通过比较你的规则和下面规则的输出和最后事实表，验证你的规则实现了相同的功能。

```
(defrule continue-check
   ?phase <- (phase check-continue)
   =>
   (retract ?phase)
   (printout t "Continue? ")
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no)) do
      (printout t "Continue? ")
      (bind ?answer (read)))
   (if (eq ?answer yes)
      then (assert (phase continue))
      else (assert (phase halt))))
```

10.2 给出一个$N \times N$的棋盘，其中N是整数。写一个程序把N个皇后放在棋盘上，使得没有皇后可以抓住另外一个皇后。提示：使用4行4列开发程序。这是能够找到解决方案的最小数目(除了1×1棋盘外)

10.3 修改习题9.12的程序，使得如果没有灌木符合所有要求，则列出符合最多要求的灌木，并打印出符合的要求数。

10.4 使用模块修改习题9.17的程序。如果对指定的颜色、硬度和密度没有匹配的宝石，程序应能指出这一点。在辨别出一种宝石后，程序应提供用户辨别其他宝石的机会。

10.5 组合习题9.14和9.15的程序，使得有一个文本菜单窗口可以启动一个应用程序、结束一个应用程序和退出程序。当选中启动条目时，提示用户输入应用程序名以及所需要的内存。当选择结束条目时，提示用户输入结束应用程序名。

10.6 修改习题9.15的程序，使得可以支持子菜单。例如：

```
CLIPS> (run)

Select one of the following options:

 1 - Option A
 2 - Option B
 3 - Submenu 1
```

```
 9 - Quit Program
Your choice: 3

Select one of the following options:

 1 - Option C
 2 - Option D
 9 - Previous Menu

Your choice: 9

Select one of the following options:

 1 - Option A
 2 - Option B
 3 - Submenu 1
 9 - Quit Program

Your choice: 9
CLIPS>
```

10.7 修改习题9.18的程序，以便为一个学生安排6门课程。程序应最大化6门课程的全部得分（见习题9.18)。程序输入是代表安排的课程的6个事实，输出是按1～6上课时间排列的课程表。用下面课程表测试你的程序。

Course	Instructors Preferred	Periods Preferred	Instructors Not Preferred	Periods Not Preferred
Texas History	Hill	2, 5	-	1, 3
Algebra	Smith	1, 2	Jones	6
Physical Education	-	-	Mack, King	1
Chemistry	Dolby	5, 6	-	-
Literature	-	3, 4	-	1, 6
German	-	-	-	-

10.8 扑克选手发了5张牌。以这5张牌作为输入事实，写一个程序打印出选手手中牌的类型：同花大顺、同花顺、四张相同、三个和一对、同花、顺子、三张相同的牌、两对、一对、什么都不是。

10.9 写一个程序通过把所有常量移到等号右边、把所有变量移到等号左边和约减相同项来简化代数等式。例如：

```
2 x + y + 5 + 3 y - 2 z - 8 = 3 z - 4 y + 4
```

可以简化为：

```
2 x + 8 y - 5 z = 7
```

由于=符号在模式中有特别的意义，你可通过隐含=符号的方式来表达等式。例如：

```
(equation (LHS 2 x + y + 5 + 3 y - 2 z - 8)
          (RHS 3 z - 4 y + 4))
```

10.10 组合习题9.19和习题10.6，以便为配置程序创建一个菜单驱动窗口。主菜单选项有选择底座、增加小装置、删除小装置、打印配置花费。子菜单进行底座的选择以及小装置的增删。选择一个底盘或者增删一个小装置后，如果所提供的小装置、支架数目以及底座提供的动力之间有冲突，则打印警告信息。然后把控制返回主菜单。选择打印配置菜单选项则列出所选底座和小装置各自的价格以及加在一起的总价。

10.11 使用第2.2节的算法写一个自定义函数，把字符串 137179766832525156430015 转化成 GOLD 438+。

提示：使用子串和 string-to-field 函数把字符串中的数字抽取出来，然后用带 %c 标志的 format 函数把数字转换成字符。

10.12 写一个自定义函数计算两个多字段值的并和交。注意当计算并和交时，重复的字段不应该出现在返回值中。

10.13 写一个自定义函数计算抛掷一个六面骰子某一面的经验概率（如第 4.6 节所述）。自定义函数的参数为骰子面数，返回值为经验概率。

10.14 写一个自定义函数找出从 1 到指定整数间的所有质数，以多字段值的形式返回这些质数。

10.15 写一个自定义函数计算一个字符串在另一个字符串中出现的次数。

10.16 写一个自定义函数一行一行的比较两个文件，把区别打印到一个指定的逻辑名。

10.17 写一个自定义函数，接受 0 个或多个参数，返回包含这些参数并以逆序存放的多字段值。

10.18 写一个自定义函数不采用递归计算整数 N 的阶乘。

10.19 写一个自定义函数把一个包含 0 和 1 的二进制字符串转换成十进制数。

10.20 不使用 if 和 switch 函数，写一组方法提供度量单位英寸、英尺和码的互相转换。例如，(convert 3 feet inches) 返回 36。写另一组方法提供度量单位厘米、米和千米的转换。

10.21 使用习题 10.20 的方法，加载 + 函数，允许两个度量单位相加。返回结果的单位为 + 方法第一个参数的单位。例如，(+ 3 feet 12 inches) 返回 4。不需要另外提供英式度量和米之间转换的方法。

10.22 加载-函数，使之去掉两个多字段值的重复字段。例如，(- (create $ a b cd) (create $ b d f)) 返回多字段值 (a c)。

10.23 为第 5.5 节的 S-函数的 4 种情况写一个方法。

10.24 给定习题 2.11 的幂集定义，写一个自定义函数打印由多字段值表达的幂集中包含的每一个集合。

10.25 Jack 的汽车里程数 N 在 200 000 和 300 000 之间。N 的十进制表示中只有一个 0。N 可平方数。N 的每个十进制位数的平方和也可平方数。写一个或多个自定义函数计算 N 的值。

第 11 章　类、实例和消息处理程序

11.1　概述

除了事实，实例（或对象）也是 CLIPS 提供的另一种数据表示。实例从类创建，类使用 CLIPS 面向对象语言（COOL）定义。就像事实结构使用自定义模板说明一样，实例结构使用自定义类（defclass）说明。使用实例和类比使用事实和自定义模板有几个优势。首先是继承。一个自定义类可以从一个或多个其他类中继承信息。这就允许更加模块化的数据定义。第二，通过使用消息处理程序，对象可以带有过程信息。第三，对象的模式匹配比事实的模式匹配更加灵活。对象模式利用继承，可以在属于多个类的槽中进行模式匹配，可以避免未说明槽的改变再激发模式，可以提供基于槽的真值维护。

11.2　自定义类结构

在创建实例之前，CLIPS 必须得到关于给定类的有效槽列表。**自定义类**（defclass）结构用来完成这个工作。它的最基本形式和自定义模板很相似：

```
(defclass <class-name> [<optional-comment>]
   (is-a <superclass-name>)
   <slot-definition>*)
```

其中＜superclass-name＞是一个类，新定义的类将要从它那里继承信息。系统类 USER 是所有用户定义类要最终继承的类。一个用户定义类既可继承另一个用户定义类也可继承 USER 类。＜slot-definition＞的语法描述如下：

```
(slot <slot-name> <slot-attribute>*) |
(multislot <slot-name> <slot-attribute>*)
```

使用这个语法，person 实例可以用下面的自定义类描述如下：

```
(defclass PERSON "Person defclass"
   (is-a USER)
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color))
```

注意和第 7.6 节的例子 person 自定义模板不同，使用了槽名 full-name 而不是 name。对于对象模式匹配，name 是具有特别意义的保留符号，在后面将会讨论。

下面的自定义模板槽属性可以用来定义自定义类的槽：type、range、cardinality、allowed-symbols、allowed-strings、allowed-lexemes、allowed-integers、allowed-floats、allowed-numbers、allowed-values、allowed-instance-names、default 和 default-dynamic。例如：

```
(defclass PERSON "Person defclass"
   (is-a USER)
   (slot full-name
      (type STRING))
   (slot age
      (type INTEGER)
      (range 0 120))
   (slot eye-color
      (type SYMBOL)
      (allowed-values brown blue green)
      (default brown))
   (slot hair-color
      (type SYMBOL)
      (allowed-values black brown red blonde)
      (default brown)))
```

自定义类的槽属性也称为槽侧面（slot facet）。

11.3 创建实例

使用 make-instance 命令可以创建实例。其语法如下：

```
(make-instance [<instance-name-expression>]
               of <class-name-expression>
               <slot-override>*)
```

其中，<slot-override>为：

```
(<slot-name-expression> <expression>)
```

例如，以下演示如何使用 person 自定义类创建一些实例：

```
CLIPS>
(make-instance [John] of PERSON
   (full-name "John Q. Public")
   (age 24)
   (eye-color blue)
   (hair-color black))↵
[John]
CLIPS> (make-instance of PERSON)↵
[gen1]
CLIPS> (make-instance Jack of PERSON)↵
[Jack]
CLIPS> (instances)↵
[John] of PERSON
[gen1] of PERSON
[Jack] of PERSON
For a total of 3 instances.
CLIPS>
```

创建了三个名为［John］、［gen1］和［Jack］的实例。如果在创建实例时没有提供实例名，系统将为你提供一个（如本例中的［gen1］）。对于 make-instance 来说，是否用中括号［］并没有区别，如用［John］和 Jack 分别调用 make instance 所显示。除了显示的是实例列表外，例子中的 instances 命令和 facts 命令类似。实例不会显示槽值，但会在下一节演示如何显示槽值。instances 命令的完整语法如下：

```
(instances [<module-name> [<class-name> [inherit]]])
```

和事实及其对应的自定义模板类似，实例属于对应自定义类的模块（参见第 11.16 节）。如果指定了可选的模块名参数，则只列出指定模块的实例。如果用*表示模块名，则列出所有实例。如果同时指定了可选类名，则只列出属于这个类的实例。最后，如果还指定了可选的 inherit 关键字，则所有属于指定类名的子类的实例也会被列出（参见第 11.6 节）。

11.4 系统定义消息处理程序

除了数据，类还附带有过程信息。这种过程称为消息处理程序。除了用户定义的消息处理程序外，还将自动为每个类产生一些系统定义的消息处理程序。这些消息处理程序通过 send 命令被唤醒以用于实例。send 命令的语法如下：

```
(send <object-expression>
<message-name-expression> <expression>*)
```

例如，print 消息显示一个实例的槽信息：

```
CLIPS> (send [John] print)↵
[John] of PERSON
(full-name "John Q. Public")
(age 24)
(eye-color blue)
(hair-color black)
CLIPS>
```

对于自定义类中定义的每一个槽，CLIPS 自动定义一个 get-和 put-槽消息处理程序来获取和设置槽

值。实际的消息处理程序名由后面加上槽名组成。例如对 PERSON 自定义类中的槽：full-name、age、eye-color 和 hair-color 将自动创建 8 个消息处理程序：get-full-name、put-full-name、get-age、put-age、get-eye-color、put-eye-color、get-hair-color 和 put-hair-color。get-消息处理程序没有参数，返回槽值。例如：

```
CLIPS> (send [John] get-full-name)↵
"John Q. Public"
CLIPS> (send [John] get-age)↵
24
CLIPS>
```

put-消息处理程序有零个或多个参数。如果没有提供参数，则槽值为初始默认值。如果提供了一个或多个参数，则槽值为这些参数值。试图把多个值放入一个单字段槽将导致错误。put-消息处理程序的返回值是槽的新值。例如：

```
CLIPS> (send [Jack] get-age)↵
nil
CLIPS> (send [Jack] put-age 22)↵
22
CLIPS> (send [Jack] get-age)↵
22
CLIPS> (send [Jack] put-age)↵
nil
CLIPS> (send [Jack] get-age)↵
nil
CLIPS>
```

watch 命令可以监视与实例有关的项目，其中一个是槽。如果槽处于监视中，那么当实例槽值发生变化时，将打印出一个信息性消息。可以使用 unwatch 命令取消对槽的监视：

```
CLIPS> (watch slots)↵
CLIPS> (send [Jack] put-age 24)↵
::= local slot age in instance Jack <- 24
24
CLIPS> (unwatch slots)↵
CLIPS> (send [Jack] put-age 22)↵
22
CLIPS>
```

另一个预先已定义的消息处理程序是 delete。你可能不相信，delete 消息处理程序用来删除一个实例。成功删除时返回 TRUE，否则返回 FALSE：

```
CLIPS> (instances)↵
[John] of PERSON
[gen1] of PERSON
[Jack] of PERSON
For a total of 3 instances.
CLIPS> (send [gen1] delete)↵
TRUE
CLIPS> (instances)↵
[John] of PERSON
[Jack] of PERSON
For a total of 2 instances.
CLIPS>
```

另一个监视项目是实例。如果实例处于监视中，当实例被创建或删除时，CLIPS 自动打印一个消息。与修改一个事实的槽值不同，修改一个实例的槽值不会删除旧实例并以新值创建一个新实例，所以要看实例槽值的变化必须使用槽监视。下面的命令对话演示如何使用实例监视命令：

```
CLIPS> (watch instances)↵
CLIPS> (make-instance Jill of PERSON)↵
⇒ instance [Jill] of PERSON
[Jill]
CLIPS> (send [Jill] put-age 22)↵
22
CLIPS> (send [Jill] delete)↵
⇐ instance [Jill] of PERSON
TRUE
CLIPS> (unwatch instances)↵
CLIPS>
```

字符<=表示一个实例被删除，而字符=>表示一个实例被创建。

11.5 自定义实例结构

与自定义事实结构对应的是**自定义实例**（definstance）结构。当执行一个重置命令时，所有的实例都会被发送一个 delete 消息，接着在自定义实例结构中的所有实例都将会被创建。自定义实例的一般形式为：

```
(definstances <definstances name> [active]
   [<optional comment>]
   <instance-definition>*)
```

其中，<instance-definition>为：

```
([<instance-name-expression>] of
   <class-name-expression>
   <slot-override>*)
```

在自定义实例结构中的可选 active 关键字用来表明在创建实例重置槽值时应引发模式匹配。默认情况下，在所有槽值重置完成之前，不会为自定义实例进行模式匹配。下面是一个自定义实例的例子：

```
(definstances people
   (Jack of PERSON (full-name "Jack Q. Public")
              (age 23))
   (of PERSON (full-name "John Doe")
              (hair-color black)))
```

有几个命令可以操作自定义实例。list-definstances 命令用来显示当前 CLIPS 所维护的自定义实例列表。ppdefinstances（pretty print definstances，漂亮打印自定义实例）命令用来显示自定义实例的文本描述。undefinstances 命令用来删除一个自定义实例。get-definstances-list 函数返回一个包含自定义实例列表的多字段值。这些命令的语法如下：

```
(list-definstances [<module-name>])

(ppdefinstances <definstances-name>)

(undefinstances <definstances-name>)

(get-definstances-list [<module-name>])
```

11.6 类与继承

使用 COOL 的一个好处是类可以从其他类继承信息，这就允许信息共享。思考如果有一个表达个人信息的自定义模板，需要做些什么：

```
(deftemplate person "Person deftemplate"
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color))
```

为了表示一个公司雇员和大学学生的其他有关信息，必须要做些什么？一个方法是扩展个人自定义模板以包括其他信息：

```
(deftemplate person "Person deftemplate"
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color)
   (slot job-position)
   (slot employer)
   (slot salary)
   (slot university)
   (slot major)
   (slot GPA))
```

在这个自定义模板中只有 4 个槽会用于所有人：full-name、age、eye-color 和 hair-color。而 job-position、employer 和 salary 槽只适用于雇员。university、major 和 GPA 槽只适用于学生。为了增加更多的信息，就必须增加更多的槽到 person 自定义模板，但其中很多槽并不适用于所有人。

另一个方法是为雇员和学生分别创建一个独立的自定义模板：

```
(deftemplate employee "Employee deftemplate"
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color)
   (slot job-position)
   (slot employer)
   (slot salary))

(deftemplate student "Student deftemplate"
   (slot full-name)
   (slot age)
   (slot eye-color)
   (slot hair-color)
   (slot university)
   (slot major)
   (slot GPA))
```

通过这种方法，每一个自定义模板都只有必要的信息，但是必须复制几个槽。如果需要改变这些重复槽的某个属性，就必须在多个地方修改以保持一致性。同样地，如果要写一个寻找有蓝色眼睛的人的规则，必须使用两个模式而不是一个（如果还包括 person 事实，则要 3 个）：

```
(defrule find-blue-eyes
   (or (employee (full-name ?name)
                 (eye-color blue))
       (student (full-name ?name)
                 (eye-color blue)))
   =>
   (printout t ?full-name "has blue eyes." crlf))
```

类允许在多个类中共享信息而不用重复信息或把不必要的信息包括进去。现在返回到原来的 PERSON 自定义类：

```
(defclass PERSON "Person defclass"
   (is-a USER)
   (slot full-name)
(slot age)
(slot eye-color)
(slot hair-color))
```

为了定义新类以扩展 PERSON 类，使用 PERSON 类作为新类的 is-a 属性。例如：

```
(defclass EMPLOYEE "Employee defclass"
   (is-a PERSON)
   (slot job-position)
   (slot employer)
   (slot salary))

(defclass STUDENT "Student defclass"
   (is-a PERSON)
   (slot university)
   (slot major)
   (slot GPA))
```

EMPLOYEE 和 STUDENT 类都继承了 PERSON 类的属性。下面的对话演示了为这 3 个类创建实例的过程：

```
CLIPS> (make-instance [John] of PERSON)↵
[John]
CLIPS> (make-instance [Jack] of EMPLOYEE)↵
[Jack]
CLIPS> (make-instance [Jill] of STUDENT)↵
[Jill]
CLIPS> (send [John] print)↵
```

```
[John] of PERSON
(full-name nil)
(age nil)
(eye-color nil)
(hair-color nil)
CLIPS> (send [Jack] print)↵
[Jack] of EMPLOYEE
(full-name nil)
(age nil)
(eye-color nil)
(hair-color nil)
(job-position nil)
(employer nil)
(salary nil)
CLIPS> (send [Jill] print)↵
[Jill] of STUDENT
(full-name nil)
(age nil)
(eye-color nil)
(hair-color nil)
(university nil)
(major nil)
(GPA nil)
CLIPS>
```

注意每个实例都包含仅适用于本类的槽。对于一个类来说，可以重定义一个它的父类已经定义了的槽，下一节将讨论这个内容。

一个直接或间接继承其他类的类是一个被继承的类的**子类**（subclass）。被继承的类称为**父类**（superclass）。PERSON、EMPLOYEE 和 STUDENT 类都是 USER 的子类。EMPLOYEE 和 STUDENT 是 PERSON 的子类。USER 是 PERSON、EMPLOYEE 和 STUDENT 的父类。PERSON 是 EMPLOYEE 和 STUDENT 的父类。一个单继承类组织是其中每个类只有一个直接父类。一个多继承类组织是其中的类可以有多于一个的直接父类。COOL 支持多继承，但我们的例子都限制在单继承直到第 11.15 节更详细地介绍多继承。以下是一个使用多继承的类例子（一个有工作的学生）：

```
(defclass WORKING-STUDENT
"Working Student defclass"
(is-a STUDENT EMPLOYEE))
```

化解槽定义冲突

默认情况下，如果一个子类重新定义一个槽，则这个类的实例专门使用新定义的槽属性。例如，下面的类中：

```
(defclass A
   (is-a USER)
   (slot x (default 3))
   (slot y)
   (slot z (default 4)))

(defclass B
   (is-a A)
   (slot x)
   (slot y (default 5))
   (slot z (default 6)))
```

创建一个 A 和 B 类的实例得到以下结果：

```
CLIPS> (make-instance [a] of A)↵
[a]
CLIPS> (make-instance [b] of B)↵
[b]
CLIPS> (send [a] print)↵
[a] of A
(x 3)
(y nil)
(z 4)
```

```
CLIPS> (send [b] print)↵
[b] of B
(x nil)
(y 5)
(z 6)
CLIPS>
```

注意实例 b 的槽 x 有一个默认值 nil 而不是 3。这是因为类 B 的槽 x 默认值空覆盖了类 A 的槽 x 默认值 3。source 槽属性可以用来允许槽属性从父类中继承。如果这个属性设置为 exclusive，这是默认设置，那么槽属性由定义这个槽的最先定义的类决定。在一个单继承组织中，就是具有最少父类的类。如果 source 槽设置为 composite，那么如果最先定义的类没有明确定义槽属性，则将从下一个最先明确定义的类中得到属性。例如，如果前面 A 和 B 自定义类重定义如下：

```
(defclass A
   (is-a USER)
   (slot x (default 3))
   (slot y)
   (slot z (default 4)))

(defclass B
   (is-a A)
   (slot x (source composite))
   (slot y (default 5))
   (slot z (default 6)))
```

然后创建类 A 和 B 实例得到以下结果：

```
CLIPS> (make-instance [a] of A)↵
[a]
CLIPS> (make-instance [b] of B)↵
[b]
CLIPS> (send [a] print)↵
[a] of A
(x 3)
(y nil)
(z 4)
CLIPS> (send [b] print)↵
[b] of B
(x 3)
(y 5)
(z 6)
CLIPS>
```

注意类 B 的槽 x 定义为 source 属性设置为 composite，它可继承类 A 的默认属性，因此实例 b 中槽 x 的默认结果值为 3。

也可以通过 propagation 槽属性来取消槽的继承。如果这个属性设置为 inherit，这是默认设置，那么槽将被子类继承。如果这个属性设置为 no-inherit，那么槽不会被子类继承。例如，定义类 A 和 B 如下：

```
(defclass A
   (is-a USER)
   (slot x (propagation no-inherit))
   (slot y))

(defclass B
   (is-a A)
   (slot z))
```

然后创建类 A 和 B 的实例得到以下结果：

```
CLIPS> (make-instance [a] of A)↵
[a]
CLIPS> (make-instance [b] of B)↵
[b]
CLIPS> (send [a] print)↵
[a] of A
(x nil)
```

```
(y nil)
CLIPS> (send [b] print)↵
[b] of B
(y nil)
(z nil)
CLIPS>
```

类 B 的实例 b 从类 A 中继承了槽 y，但没有继承槽 x，因为 x 的 propagation 属性为 no-inherit。

抽象和具体类

可以定义只能用于继承的类。这样的类称为**抽象**（abstract）类。不能从抽象类来创建实例。默认情况下，类是**具体**（concrete）的。类属性 role 用来规定一个类是抽象的还是具体的。role 类属性必须在 is-a 类属性后、槽定义之前定义。例如：

```
(defclass ANIMAL
   (is-a USER)
   (role abstract))

(defclass MAMMAL
   (is-a ANIMAL)
   (role abstract))

(defclass CAT
   (is-a MAMMAL)
   (role concrete))

(defclass DOG
   (is-a MAMMAL)
   (role concrete))
```

类 ANIMAL 和 MAMMAL 是抽象的。类 CAT 和 DOG 是具体的。role 属性是可以继承的，所以可不必声明 MAMMAL 是抽象的，因为它继承了 ANIMAL 的这个属性；必须声明 CAT 和 DOG 类是具体的，否则它们将是抽象的。尝试从抽象类创建实例会导致出错消息：

```
CLIPS> (make-instance [animal-1] of ANIMAL)↵
[INSMNGR3] Cannot create instances of abstract
class ANIMAL.
CLIPS> (make-instance [cat-1] of CAT)↵
[cat-1]
CLIPS>
```

除非可以使代码更加容易维护和重用，否则没有必要总是声明一个类为抽象类。如果你创建类的目的不是让别人使用它创建实例，你就不会希望有人用它来创建实例。假使有人用它创建了实例，那么以后将无法在删除它的同时保持代码完整性。

在现在的例子中，无论 ANIMAL 和 MAMMAL 类是否为抽象的都可以成立。如果要创建一个动物园的详细目录，那么这些类都很可能为抽象的。不存在抽象的动物或哺乳动物，也没有任何动物或哺乳动物不是某一更具体类的成员。但是，如果我们试图识别一个动物，可以想像，我们可能会从 ANIMAL 或 MAMMAL 类来创建实例，例如，说明如何识别动物。

自定义类命令

有几个命令用来操作自定义类。list-defclasses 命令用来显示 CLIPS 当前维护的自定义类列表。它的语法如下：

```
(list-defclasses [<module-name>])
```

从这个函数得到的输出是：

```
CLIPS> (list-defclasses)↵
FLOAT
INTEGER
```

```
SYMBOL
STRING
MULTIFIELD
EXTERNAL-ADDRESS
FACT-ADDRESS
INSTANCE-ADDRESS
INSTANCE-NAME
OBJECT
PRIMITIVE
NUMBER
LEXEME
ADDRESS
INSTANCE
USER
INITIAL-OBJECT
PERSON
EMPLOYEE
STUDENT
For a total of 20 defclasses.
CLIPS>
```

有很多预先已定义的基本类：OBJECT、PRIMITIVE、NUMBER、LEXEME、FLOAT、INTEGER、SYMBOL、STRING、MULTIFIELD、ADDRESS、INSTANCE、EXTERNAL-ADDRESS、FACT-ADDRESS、INSTANCE-ADDRESS 和 INSTANCE-NAME。你不能用这些类来创建其他的类。基本类与第 10 章讨论的类属函数特别有用。剩下的预先已定义类是 USER 和 INITIAL-OBJECT。USER 是创建新类的基础。INITIAL-OBJECT 是 USER 的子类，用来创建初始对象实例（参见第 11.7 节）。我们创建的类（PERSON，EMPLOYEE 和 STUDENT）在上述表的最末。图 11.1 显示了这些预先已定义类的层次关系。

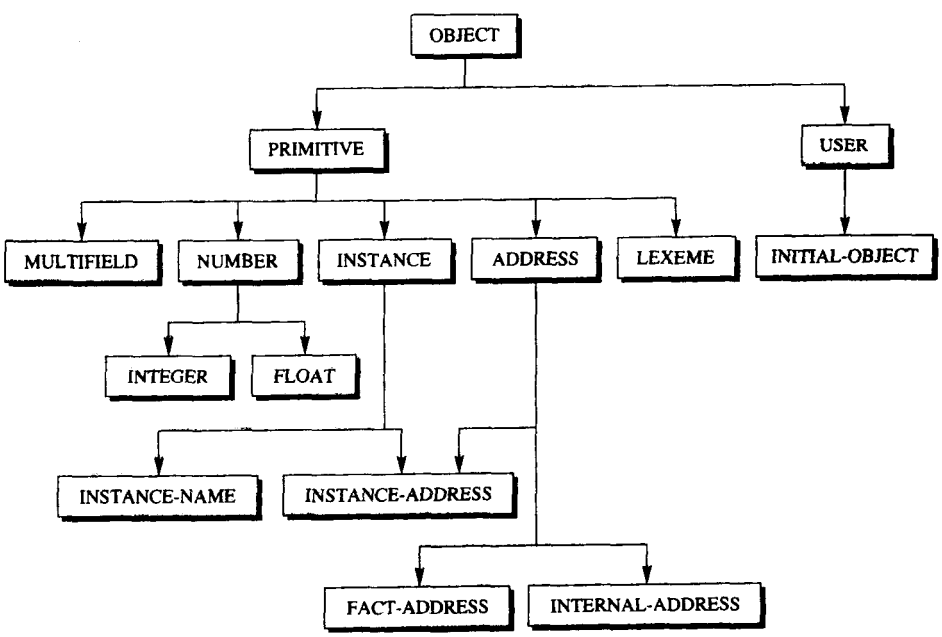


图 11.1 预先已定义类一览

browse-classes 命令用来显示一个类和它的子类的继承关系。它的语法如下：

```
(browse-classes [<class-name>])
```

如果没有指定类名，则显示根类 OBJECT 的继承关系。例如下面的命令显示出图 11.1 中信息，并

显示了 PERSON 类及其子类作为例子：

```
CLIPS> (browse-classes)↵
OBJECT
  PRIMITIVE
    NUMBER
      INTEGER
      FLOAT
    LEXEME
      SYMBOL
      STRING
    MULTIFIELD
    ADDRESS
      EXTERNAL-ADDRESS
      FACT-ADDRESS
      INSTANCE-ADDRESS *
    INSTANCE
      INSTANCE-ADDRESS *
      INSTANCE-NAME
  USER
    INITIAL-OBJECT
    PERSON
      EMPLOYEE
      STUDENT
CLIPS>
```

缩行用来表示一个类是前面第一个较少空格缩行类的子类。例如，NUMBER、LEXEME、MULTIFIELD、ADDRESS 和 INSTANCE 都是 PRIMITIVE 的子类。类后面的星号表示它是多个类的直接子类。例如，INSTANCE-ADDRESS 是 ADDRESS 和 INSTANCE 的直接子类。

在 browse-classes 命令中指定类将显示指定类及其子类的继承关系：

```
CLIPS> (browse-classes PERSON)↵
PERSON
  EMPLOYEE
  STUDENT
CLIPS>
```

ppdefclass（pretty print defclass，漂亮打印自定义类）命令用来显示自定义类的文本描述。undefclass 命令用来删除一个自定义类。这些命令的语法为：

```
(ppdefclass <defclass-name>)

(undefclass <defclass-name>)
```

当存在类实例时无法删除这个类。在删除类之前，属于这个类或子类的实例都必须删除。例如：

```
CLIPS> (undefclass STUDENT)↵
[PRNTUTIL4] Unable to delete defclass STUDENT.
CLIPS> (send [Jill] delete)↵
TRUE
CLIPS> (undefclass STUDENT)↵
CLIPS>
```

11.7 对象模式匹配

对象模式提供了自定义模板或有序事实模式所不具备的几个能力。首先，一个单一对象模式可以匹配几个类的实例。其次，在对象模式中没有说明的槽值的改变不会再激发这个模式所属的规则。第三，在逻辑条件元素的对象模式里没有说明的槽值更改不会导致相关规则的逻辑支持失效。对象模式的一般形式为：

```
(object <attribute-constraint>*)
```

其中，<attribute-constraint>为：

```
(is-a <constraint>) |
(name <constraint>) |
(<slot-name> <constraint>*)
```

<constraint>和用于自定义模板模式中的模式槽约束是一样的。我们将简短地介绍 is-a 和 name 关键字。首先从一个简单的对象模式匹配例子开始：

```
CLIPS> (clear)↵
CLIPS>
(defclass 1D-POINT
   (is-a USER)
   (slot x))↵
CLIPS>
(defrule Example-1
   (object (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (make-instance p1 of 1D-POINT (x 3))↵
[p1]
CLIPS> (agenda)↵
0      Example-1: [p1]
For a total of 1 activation.
CLIPS> (run)↵
Value of x slot is 3
CLIPS>
```

首先，定义类 1D-POINT 具有单个属性 x。然后定义自定义规则 Example-1。注意这个规则只在槽 x 上匹配，并没有提到 1D-POINT 类。当规则被创建后，CLIPS 自动地判定 1D-POINT 类可以匹配这个模式。在创建了 1D-POINT 类的实例以后，Example-1 规则被适当地激发，运行程序正确地输出实例的 x 槽值。

现在看看如果我们定义另一个包含 x 槽的类会发生什么：

```
CLIPS>
(defclass 2D-POINT
   (is-a USER)
   (slot x)
   (slot y))↵
CLIPS> (make-instance p2 of 2D-POINT (x 4) (y 2))↵
[p2]
CLIPS> (agenda)↵
CLIPS>
```

可能有些意外，Example-1 规则不会被实例［p2］激发。这是因为一个类能否匹配一个对象模式是在规则定义时决定的。在规则定义之后创建的类实例将不会匹配规则里的对象模式。如果 Example-1 规则被重新定义，将看到我们原来预期的行为：

```
CLIPS>
(defrule Example-1
   (object (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (agenda)↵
0      Example-1: [p2]
0      Example-1: [p1]
For a total of 2 activations.
CLIPS> (run)↵
Value of x slot is 4
Value of x slot is 3
CLIPS>
```

注意 Example-1 规则现在被来自不同类但都包含 x 槽的［p1］和［p2］实例激发。

对象模式匹配与继承

对象模式也可以在继承槽上匹配：

```
CLIPS> (clear)↵
CLIPS>
(defclass 1D-POINT
   (is-a USER)
   (slot x))↵
CLIPS>
```

```
(defclass 2D-POINT
   (is-a 1D-POINT)
   (slot y))↵
CLIPS>
(defclass 3D-POINT
   (is-a 2D-POINT)
   (slot z))↵
CLIPS>
(defrule Example-2
   (object (y ?y))
   =>
   (printout t "Value of y slot is " ?y crlf))↵
CLIPS> (make-instance p1 of 1D-POINT (x 3))↵
[p1]
CLIPS> (make-instance p2 of 2D-POINT (x 1) (y 2))↵
[p2]
CLIPS> (make-instance p3 of 3D-POINT (x 2) (y 4) (z 3))↵
[p3]
CLIPS> (agenda)↵
0      Example-2: [p3]
0      Example-2: [p2]
For a total of 2 activations.
CLIPS> (run)↵
Value of y slot is 4
Value of y slot is 2
CLIPS>
```

注意 Example-2 规则被实例 [p2] 和 [p3] 激发，而不是 [p1]。实例 [p2] 是类 2D-POINT 的一个实例，这个类定义了一个被规则引用的 y 槽。实例 [p3] 是类 3D-POINT 的一个成员，这个类从类 2D-POINT 继承了 y 槽。而类 1D-POINT 的实例 [p1] 只有一个 x 槽，所以不能匹配规则 Example-2 的对象模式。

is-a 和 name 关键字

在对象模式中使用时，is-a 关键字作为槽名有特殊的含义。它限制实例模式匹配只针对那些满足 is-a 约束的类。例如：

```
CLIPS>
(defrule Example-3
   (object (is-a 2D-POINT) (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (agenda)↵
0      Example-3: [p3]
0      Example-3: [p2]
For a total of 2 activations.
CLIPS> (run)↵
Value of x slot is 2
Value of x slot is 1
CLIPS>
```

注意只有实例 [p2] 和 [p3] 激活规则 Example-3，即使实例 [p1] 也有 x 槽属性。这是因为 [p1] 不是类 2D-POINT 的一个实例也不是继承自它的类的实例。实例 [p3] 满足对象模式，因为它通过继承是 2D-POINT 类的一个成员。通过明确地禁止 3D-POINT 类可以阻止实例 [p3] 匹配对象模式：

```
CLIPS>
(defrule example-4
   (object (is-a 2D-POINT&~3D-POINT) (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (agenda)↵
0      example-4: [p2]
For a total of 1 activation.
CLIPS> (run)↵
Value of x slot is 1
CLIPS>
```

任何被对象模式引用的类必须是已经定义了的；否则会产生错误：

```
CLIPS>
(defrule example-5
   (object (is-a 4D-POINT) (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
[OBJRTBLD5] Undefined class in object pattern.

ERROR:
(defrule MAIN::example-5
   (object (is-a 4D-POINT)
CLIPS>
```

被对象模式引用的槽也是如此。必须至少有一个类包含对象模式中引用的所有槽属性，否则也会产生错误：

```
CLIPS>
(defrule example-6
   (object (w ?w))
   =>
   (printout t "Value of w slot is " ?w crlf))↵
[OBJRTBLD2] No objects of existing classes can
satisfy w restriction in object pattern.
ERROR:
(defrule MAIN::example-6
   (object (w
CLIPS>
```

既然没有一个现存的类（1D-POINT、2D-POINT 和 3D-POINT）具有 w 槽属性，规则 Example-6 中的对象模式不能满足，因此产生错误。

name 关键字用来匹配指定的实例。例如：

```
CLIPS>
(defrule example-7
   (object (name [p1] | [p3]) (x ?x))
   =>
   (printout t "Value of x slot is " ?x crlf))↵
CLIPS> (agenda)↵
0      example-7: [p3]
0      example-7: [p1]
For a total of 2 activations.
CLIPS> (run)↵
Value of x slot is 2
Value of x slot is 3
CLIPS>
```

规则 Example-7 使用 name 关键字限制可以匹配对象模式的实例为 [p1] 和 [p3]。实例 [p2] 具有 x 槽值，但不会匹配这个模式，因为它的名字不满足 name 限制。由于在对象模式中的特别含义，is-a和 name 关键字不能在自定义类的定义中作为槽名。

激发对象模式

对象模式与事实模式的一个重要不同在于只有那些明确匹配某个槽的对象模式才会在实例槽值改变时受到影响。为了说明这一点，我们重新看看第 8.10 节的 sum-rectangles 规则：

```
(defrule sum-rectangles
   (rectangle (height ?height) (width ?width))
   ?sum <- (sum ?total)
   =>
   (retract ?sum)
   (assert (sum (+ ?total (* ?height ?width)))))
```

这个规则的问题在于任何修改 sum 事实的尝试都会引发规则重新激发并应用于刚刚处理过的 rectangle 事实，从而导致一个无限循环。把 sum 事实从一个有序事实转换成自定义模板事实，然后使用 modify 命令并不能解决这个问题。第 8.4 节中的解决方案使用多规则来计算总和。一个更好的解决方法是恰

当地利用对象模式。下面是尝试用对象重写第 8.4 节 sum-rectangles 的代码：

```
(defclass RECTANGLE
   (is-a USER)
   (slot height)
   (slot width))

(defclass SUM
   (is-a USER)
   (slot total))

(definstances initial-information
   (of RECTANGLE (height 10) (width 6))
   (of RECTANGLE (height 7) (width 5))
   (of RECTANGLE (height 6) (width 8))
   (of RECTANGLE (height 2) (width 5))
   ([sum] of SUM (total 0)))

(defrule sum-rectangles
(object (is-a RECTANGLE)
     (height ?height) (width ?width))
?sum <- (object (is-a SUM) (total ?total))
 =>
(send ?sum put-total
  (+ ?total (* ?height ?width))))
```

关于新的 sum-rectangles 规则有几点需要注意的地方。首先，如事实模式一样可以使用<－模式约束操作符约束匹配模式的实例给一个变量。在这个例子中，匹配 SUM 对象模式的实例地址约束给 ? sum 变量。这个变量可以作为参数传给 send 函数以发送消息给实例。运行代码证明与事实模式有同样的问题：无限循环。这是因为对规则 RHS 中 total 槽值的修改所使用的 put-total 消息由于匹配 total 槽，而再次激发了 SUM 对象模式。

我们并不需要在规则 LHS 中匹配 total 槽，因为约束给 ? total 的值将不在任何 LHS 模式中被再次引用。通过在规则 RHS 传递消息获取值将得到下面规则：

```
(defrule sum-rectangles
   (object (is-a RECTANGLE)
        (height ?height) (width ?width))
   ?sum <- (object (is-a SUM))
=>
(bind ?total (send ?sum get-total))
(send ?sum put-total
    (+ ?total (* ?height ?width))))
```

这个规则版本不会像初始版本一样无限循环。如果知道 SUM 类只有一个实例，可进一步简化这个规则：

```
(defrule sum-rectangles
   (object (is-a RECTANGLE)
        (height ?height) (width ?width))
   =>
   (bind ?total (send [sum] get-total))
   (send [sum] put-total
     (+ ?total (* ?height ?width))))
```

我们可简单通过在规则 RHS 使用名字引用实例，而不是通过模式匹配获取存放在? sum 变量中的实例地址。

模式匹配属性

进行模式匹配时，可以通过设置 pattern-match 属性令槽或者类不参与进来。如果属性设置为 reactive，这是默认值，那么指定的槽或类将激发规则 LHS 的模式匹配。如果属性设置为 non-reactive，那么指定的槽或类将不会激发规则 LHS 的模式匹配。例如，如果使用 pattern-match 属性重定义 SUM 类，则原来的 sum-rectangles 规则可改成如下：

```
(defclass SUM
   (is-a USER)
   (slot total (pattern-match non-reactive)))

(defrule sum-rectangles
   (object (is-a RECTANGLE)
         (height ?height) (width ?width))
   ?sum <- (object (is-a SUM) (total ?total))
   =>
   (send ?sum put-total
     (+ ?total (* ?height ?width))))
```

当定义 sum-rectangles 规则时将出现下面错误信息：

```
[OBJRTBLD2] No objects of existing classes can
satisfy total restriction in object pattern.
```

基本上，既然第二个模式的 is-a 条件限制了可能的类为 SUM，而这个类没有 total 槽可用于模式匹配，那就不可能匹配这个模式，因此规则出错。多个类可具有相同的槽名，其中有些为 reactive，有些为 non-reactive。当规则定义时就确定了哪些类可以进行匹配，那些在模式引用的槽中有 non-reactive 的类将不予考虑。

也可以在类一级使用 pattern-match 属性。例如：

```
(defclass SUM
   (is-a USER)
   (pattern-match non-reactive)
   (slot total))
```

pattern-match 类属性必须在 is-a 和 role 类属性之后、槽定义之前说明。如果一个类说明为 non-reactive，则类的实例将不会匹配任何模式（无论每个槽的 pattern-match 属性设置如何），但如果 pattern-match 类属性重新定义为 reactive，则子类的实例可以匹配模式。由于类的 pattern-match 属性并不明确地影响槽 pattern-match 属性，一个类可以从 non-reactive 类继承 reactive 槽。

对象模式与逻辑条件元素

正如事实和事实模式一样，对象模式和实例创建可以和逻辑条件元素结合使用。如果在逻辑条件元素中槽没有被一个对象模式引用，那么实例槽值的改变不会影响事实或实例的逻辑支持。例如，思考下面使用自定义模板模式的结构：

```
(deftemplate emergency
   (slot type)
   (slot location))

(deftemplate response-team
   (slot name)
   (slot location))

(defrule create-response-team
   (logical (emergency (location ?location)))
   =>
   (assert (response-team (name first-response)
                          (location ?location))))
```

如果调入结构并断言一个 emergency 事实，create-response-team 规则将在运行程序时创建一个 response-ream事实：

```
CLIPS> (reset)↵
CLIPS> (watch facts)↵
CLIPS>
(assert (emergency (type unknown)
                   (location building-1))↵
==> f-1     (emergency (type unknown)
                       (location building-1))
<Fact-1>
CLIPS> (run)↵
==> f-2     (response-team (name first-response)
                           (location building-1))
CLIPS>
```

修改 emergency 事实导致 response-team 事实被撤销，即使 create-response-team 规则并没有存取type 槽：

```
CLIPS> (modify 1 (type fire))↵
<== f-1     (emergency (type unknown)
               (location building-1))
<== f-2     (response-team (name first-response)
                           (location building-1))
==> f-3     (emergency (type fire)
                       (location building-1))
<Fact-3>
CLIPS>
```

create-response-team 规则需要重新激发以再创建 response-team 事实：

```
CLIPS> (run)↵
==> f-4     (response-team (name first-response)
                           (location building-1))
CLIPS>
```

以下是使用自定义类和对象的同一个程序。注意我们删掉了 response-team 自定义模板中的 name 槽，因为这在对象模式中有特殊含义。而在 RESPONSE-TEAM 自定义模板 name 槽的地方仅是给出了 response-team 实例的名：

```
(defclass EMERGENCY
   (is-a USER)
   (slot type)
   (slot location))
(defclass RESPONSE-TEAM
   (is-a USER)
   (slot location))

(defrule create-response-team
   (logical (object (is-a EMERGENCY)
                    (location ?location)))
   =>
   (make-instance first-response of RESPONSE-TEAM
      (location ?location)))
```

结合这些新的结构，我们可以看到最初的行为和使用事实类似：

```
CLIPS> (reset)↵
CLIPS> (watch instances)↵
CLIPS>
(make-instance e1 of EMERGENCY
(type unknown) (location building-1))↵
==> instance [e1] of EMERGENCY
[e1]
CLIPS> (send [e1] print)↵
[e1] of EMERGENCY
(type unknown)
(location building-1)
CLIPS> (run)↵
==> instance [first-response] of RESPONSE-TEAM
CLIPS>
```

在创建［e1］实例后运行程序建立了［first-response］实例。这些实例和事实例子中创建的 emergency 以及 response-team 自定义模板类似。

把［e1］实例的类型从 unknown 改变为 fire 不会导致删除或创建另一个［first-response］实例：

```
CLIPS> (send [e1] put-type fire)↵
fire
CLIPS> (send [e1] print)↵
[e1] of EMERGENCY
(type fire)
(location building-1)
CLIPS>
```

由于［first-response］实例并没有被删除，create-response-team 规则再次执行将重新创建它，在这个例子

中，对象模式和实例的使用比使用事实和事实模式更加有效。

由于 location 槽在 create-response-team 规则对象模式中有明确的匹配，改变这个槽值会产生类似使用事实的行为：

```
CLIPS> (send [e1] put-location building-2)↵
<== instance [first-response] of RESPONSE-TEAM
building-2
CLIPS> (run)↵
==> instance [first-response] of RESPONSE-TEAM
CLIPS>
```

初始对象模式

正如第 7.11 和第 8.15 节讨论的，在某些环境下 CLIPS 将增加规则 LHS 的 initial-fact 模式。当在一个规则中使用对象模型时，CLIPS 在某些环境下会使用 initial-object 模式。在对象中与第 7.10 节讨论的 initial-fact 自定义模板和 initial-fact 自定义事实相对应的是 INITIAL-OBJECT 自定义类和 initial-object 自定义实例：

```
(defclass INITIAL-OBJECT
   (is-a USER))

(definstances initial-object
   (initial-object of INITIAL-OBJECT))
```

如果在规则的某个位置需加入一个 initial-fact/initial-object 模式，那么如果插入点之前的模式是事实模式，则插入一个 initial-fact 模式。否则如果插入点之前的模式是对象模式，那么则插入一个 initial-object 模式。如果插入点之前没有模式，那么插入点之后的模式将用来决定被插入的模式类型。下面的格式用于 initial-object 模式：

```
(object (is-a INITIAL-OBJECT)
    (name [initial-object]))
```

使用模式加入的这些新规则，自定义规则：

```
(defrule no-emergencies
   (not (object (is-a EMERGENCY)))
   =>
   (printout t "No emergencies" crlf))
```

将被转化为：

```
(defrule no-emergencies
   (object (is-a INITIAL-OBJECT)
           (name [initial-object]))
(not (object (is-a EMERGENCY)))
=>
(printout t "No emergencies" crlf))
```

尽管在 not 条件元素之前没有模式，但 not 条件元素之后的模式是一个对象模式。

11.8 用户定义消息处理程序

除了 COOL 自动为每个类所定义的 print、delete、put-和 get-系统定义消息处理程序外，你还可以定义自己的消息处理程序。defmessage-handler 结构用于完成此功能。这个结构的一般语法是：

```
(defmessage-handler <class-name> <message-name>
                    [<handler-type>]
                    [<optional-comment>]
   (<regular-parameter>*
[<wildcard-parameter>])
   <expression>*)
```

其中＜regular-parameter＞是一个单字段变量，＜wildcard-parameter＞是一个多字段值，而＜handler-type＞是符号 around、before、primary 或 after 中的一个。默认情况下，消息处理程序是一个 primary 消

息处理程序。around、before 和 after 类型处理程序的意义将在第 11.10 节讨论。每一个类有自己的消息处理程序集合，所以没有必要区别<message-name>和其他类使用的消息处理程序名。<regular-parameter>和<wildcard-parameter>是在调用时要传给消息处理程序的参数。它们的作用等同于自定义函数中的参数。消息处理程序体，表示为<expression>*，也和自定义函数体一样。可以使用 bind 函数来约束局部变量，消息处理程序体最后表达式的计算结果作为消息处理程序的返回值：

```
(defclass RECTANGLE
   (is-a USER)
   (slot height)
   (slot width))

(defclass CIRCLE
   (is-a USER)
   (slot radius))

(defmessage-handler RECTANGLE compute-area
   ()
   (* (send ?self get-height)
      (send ?self get-width)))
(defmessage-handler CIRCLE compute-area
   ()
   (* (pi)
      (send ?self get-radius)
      (send ?self get-radius)))

(definstances figures
   (rectangle-1 of RECTANGLE (height 2) (width 4))
   (circle-1 of CIRCLE (radius 3)))
```

这个例子定义了两个类，RECTANGLE 和 CIRCLE，每一个都有适当的槽。消息处理程序 compute-area 附加给每个类。这个消息处理程序的目的是计算每个对象的面积。对 RECTANGLE 类，其实例面积是实例的高乘以宽。对 CIRCLE 类，其实例面积是 pi 函数返回的 π 值乘以实例半径的平方。注意变量? self 在两个消息处理程序中都有使用。这是自动为每个消息处理程序定义的一个特殊变量。当消息处理程序被调用，? self 变量的值赋为消息将要发往的实例的地址。可以使用这个变量来发送消息给实例，正如我们在本例中利用它获取 height，width 和 radius 槽的值。

一旦消息处理程序被定义，可以发送一个 compute-area 消息给 RECTANGEL 和 CIRCLE 类实例以便计算实例的面积：

```
CLIPS> (reset)↵
CLIPS> (send [circle-1] compute-area)↵
28.2743338823081
CLIPS> (send [rectangle-1] compute-area)↵
8
CLIPS>
```

注意我们发送给每个实例相同的消息，但每个返回它们不同的面积。这种对相同消息不同实例能以自身的方式的响应，称为**多态性**（polymorphism）。

槽速记引用

每次获取或设置槽值都必须向实例发送消息常常很不方便，因此可使用速记机制来存取约束给? self 变量的实例槽。不必使用表达式：

```
(send ?self get-<slot-name>)
```

而只使用：

```
?self:<slot-name>
```

就可以获取槽值。例如，compute-area 消息处理程序可以重定义为如下：

```
(defmessage-handler RECTANGLE compute-area
   ()
   (* ?self:height ?self:width))

(defmessage-handler CIRCLE compute-area
   ()
   (* (pi) ?self:radius ?self:radius))
```

可使用类似的机制来设置槽值。不必使用表达式：

```
(send ?self put-<slot-name> <expression>*)
```

而使用表达式：

```
(bind ?self:slot-name <expression>*)
```

这些可替换机制既可传递消息也可直接控制槽。如果你定义类中 get-和 put-消息处理程序为 after、before 或 around 类型消息处理程序，这将有某些特殊意义（在第 11.10 节讨论）。

默认情况下，类的消息处理程序只可以对直接由类定义的槽使用速记引用（即不是继承的槽）。例如，给出下面两个自定义类：

```
(defclass A
   (is-a USER)
   (slot x))

(defclass B
   (is-a A)
   (slot y))
```

下面对话展示了使用速记引用只可在类 B 的消息处理程序中引用槽 y 而不能引用槽 x：

```
CLIPS>
(defmessage-handler B bmh1 ()
   (* 2 ?self:x))↵
[MSGFUN6] Private slot x of class A cannot be
accessed directly by handlers attached to class B

[PRCCODE3] Undefined variable self:x referenced in
message-handler.
ERROR:
(defmessage-handler MAIN::B bmh1
   ()
   (* 2 ?self:x)
   )
CLIPS>
(defmessage-handler B bmh2 ()
   (* 2 ?self:y))↵
CLIPS>
```

bmh1 消息处理程序引用了？self：x，这是不允许的。因为 x 槽从类 A 中继承。bmh2 消息处理程序可以引用？self：y，因为 y 槽由类 B 定义。COOL 支持**对象封装**（encapsulation），这意味着可以隐藏一个类的实现细节，存取类只限定于使用定义好的接口：为类而定义的消息处理程序。因为 get-x 消息处理程序是获取类 A 实例的 x 槽值的接口，所以这个接口必须由 bmh1 消息处理程序使用。经过这些更改，bmh1 消息处理程序可以正确地定义如下：

```
CLIPS>
(defmessage-handler B bmh1 ()
   (* 2 (send ?self get-x)))↵
CLIPS>
```

使用 visibility 槽属性可以取消对槽的封装。如果这个属性设置为 private，这是默认设置，则槽只能被其定义类中的消息处理程序直接存取。如果这个属性设置为 public，则槽可以被其任何定义类的子类或者父类直接存取。在前面的例子中，定义类 A 如下：

```
(defclass A
   (is-a USER)
   (slot x (visibility public)))
```

将允许你定义消息处理程序 bmh1 如下：

```
(defmessage-handler B bmh1 ()
   (* 2 ?self:x))
```

监视消息和消息处理程序

如果消息和消息处理程序被 watch 命令监视，每当消息处理程序开始或结束运行时会打印出一个信息性消息。例如：

```
CLIPS> (watch messages)↵
CLIPS> (watch message-handlers)↵
CLIPS> (send [circle-1] compute-area)↵
MSG >> compute-area ED:1 (<Instance-circle-1>)
HND >> compute-area primary in class CIRCLE
       ED:1 (<Instance-circle-1>)
HND << compute-area primary in class CIRCLE
       ED:1 (<Instance-circle-1>)
MSG << compute-area ED:1 (<Instance-circle-1>)
28.2743338823081
CLIPS> (send [rectangle-1] compute-area)↵
MSG >> compute-area ED:1 (<Instance-rectangle-1>)
HND >> compute-area primary in class RECTANGLE
       ED:1 (<Instance-rectangle-1>)
HND << compute-area primary in class RECTANGLE
       ED:1 (<Instance-rectangle-1>)
MSG << compute-area ED:1 (<Instance-rectangle-1>)
8
CLIPS>
```

当一个消息被发送给实例以及这个消息的执行完成时，监视消息会打印出调试信息。当一个指定的消息处理程序开始和结束执行时，监视消息处理程序会打印出调试信息。监视消息处理程序可提供监视消息的所有信息，还加上其他一些信息。在前面的对话中，监视消息打印的信息以 MSG 开头，监视消息处理程序打印的信息以 HND 开头。符号≫表示消息/消息处理程序开始，而符号≪表示消息/消息处理程序完成。随后跟着消息或消息处理程序名。对消息处理程序此时还有额外的信息被打印：处理程序类型（before、after、primary 或 around），接着是与执行消息处理程序相关联的类。一个消息可以引起不同类或类型的多个消息处理程序执行，该信息告诉实际执行的是哪一个。最后打印出的信息是表示求值深度的符号 ED，后面跟着传给消息处理程序的参数。和自定义函数一样，求值深度表示自定义函数和消息处理程序调用的嵌套信息。它从零开始，每次进入一个自定义函数/消息处理程序则增加一。每次退出一个自定义函数/消息处理程序则减一。在参数列表中，列出的第一个参数总是变量?self 的值，消息处理程序的显式参数列在后面。

通过指明类的名字可以监视这个类的所有消息处理程序；要监视指定的消息处理程序可以指明其所在的类名和消息处理程序名；通过指明类、消息处理程序名以及类型（before、after、primary 或 around）可以监视指定类型的消息处理程序：

```
CLIPS> (unwatch all)↵
CLIPS> (watch message-handlers CIRCLE)↵
CLIPS> (watch message-handlers RECTANGLE
       compute-area)↵
CLIPS> (watch message-handlers RECTANGLE
       get-height primary)↵
CLIPS>
```

第一个 watch 命令将监视 CIRCLE 类的所有用户和系统定义消息处理程序。第二个 watch 命令将监视 RECTANGLE 类中 primary 类型的 compute-area 消息处理程序（也会监视 after、before 和 around 处理程序，如果定义了的话）。第三个 watch 命令将仅监视 RECTANGLE 类中 primary 类型的 get-height 系统定义消息处理程序。

自定义消息处理程序命令

有几个命令可以操作自定义消息处理程序。list-defmessage-handlers 命令用来显示当前 CLIPS 维护的自定义消息处理程序列表。它的语法是：

```
(list-defmessage-handlers
  [<defclass-name> [inherit]])
```

其中 inherit 关键字表明列出继承的消息处理程序。例如：

```
CLIPS> (list-defmessage-handlers RECTANGLE)↵
get-height primary in class RECTANGLE
put-height primary in class RECTANGLE
get-width primary in class RECTANGLE
put-width primary in class RECTANGLE
compute-area primary in class RECTANGLE
For a total of 5 message-handlers.
CLIPS>
```

ppdefmessage-handlers（pretty print defmessage-handler，漂亮打印自定义消息处理程序）命令用来显示自定义消息处理程序的文本描述。undefmessage-handler 命令用来删除一个自定义消息处理程序。get-defmessage-handler-list 函数返回一个包含指定类的自定义消息处理程序列表的多字段值。这些命令的语法如下：

```
(ppdefmessage-handler <defclass-name>
                      <handler-name>
                      [<handler-type>])

(undefmessage-handler <defclass-name>
                      <handler-name>
                      [<handler-type>])
(get-defmessage-handler-list <defclass-name>
                             [inherit])
```

其中＜handler-type＞是符号 around、before、primary 或 after 中的一个。例如：

```
CLIPS>
 (ppdefmessage-handler RECTANGLE compute-area)↵
(defmessage-handler MAIN::RECTANGLE compute-area
   ()
   (* (send ?self get-height) (send ?self
                               get-width)))
CLIPS>
 (undefmessage-handler RECTANGLE get-height)↵
[MSGPSR3] System message-handlers may not be
modified.
CLIPS> (undefmessage-handler RECTANGLE
       compute-area before)↵
[MSGFUN8] Unable to delete message-handler(s) from
   class RECTANGLE.
CLIPS> (undefmessage-handler RECTANGLE
       compute-area primary)↵
CLIPS> (get-defmessage-handler-list CIRCLE)↵
(CIRCLE get-radius primary
CIRCLE put-radius primary
CIRCLE compute-area primary)
CLIPS>
```

注意你不能够删除系统定义的消息处理程序。RECTANGLE 类没有 before 类型的 compute-area 消息处理程序，所以不能删除。它具有一个 primary 类型的 compute-area 消息处理程序，即使不在命令中指明 primary 也可以删除，因为这是删除的默认类型。get-defmessage-handler-list 函数为每个消息处理程序返回 3 个值：处理程序所属的类（如果指明了 inherit 关键字，则只是传给函数的类名不同）、消息处理程序名和类型。

11.9 槽存取和处理程序创建

通过 access 和 create-accessor 槽属性可以控制对槽的存取。access 属性直接限制槽允许的存取类型。

如果这个属性设置为 read-write，这是默认值，则类的处理程序可以用槽速记命令:? self: <slot-name>直接读写槽值。如果这个属性设置为 read-only，则使用槽速记命令可以获取槽值，但不能用 bind 函数来改变值。把一个值存入槽中的惟一方法是使用默认属性。设置 access 属性为 initialize-only 和 read-only 类似，除了在创建实例时可以设置槽值（例如 make-instance）。

create-accessor 属性用来控制类槽的 get-和 put-处理程序的自动创建。如果这个属性设置为 read-write，这是默认值，则会创建 get-和 put-处理程序。类似的，如果设置为 read，则只生成 get-处理程序，如果设置为 write，则只生成 put-处理程序。如果设置为? NONE，则不创建 get-和 put-处理程序。

access 和 create-accessor 属性的某些组合会产生错误。例如，对某个槽设置 access 为 read-only 而 create-accessor 为 read-write。很明显你不能对一个只允许读的槽进行写操作。

下面考察一个例子演示这些属性。假设我们从顾客得到一个订单，需要算出总价格。我们为每个价格编上独特的编号 ID，一旦确定就不再更改这个编号。订单的总价格根据订单中项目的价格加上销售税来计算，而且除了类的处理程序，我们不希望用别的方式来计算或设置这个总价格。计算订单类的代码如下：

```
(defclass ORDER
    (is-a USER)
    (slot ID (access initialize-only)
          (default-dynamic (gensym)))
(slot total-price (create-accessor read)
      (default 0.0))
    (slot order-price (default 0.0))
    (slot sales-tax (default 0.0)))

(defmessage-handler ORDER compute-total-price ()
    (bind ?self:total-price
       (* ?self:order-price
          (+ 1 ?self:sales-tax))))
```

ID 槽只能在定义实例时指定。如果不定义其值，它将通过调用 gensym 函数自动产生。total-price 槽的 create-accessor 属性设置为 read，意味着将生成一个 get-total-price 处理程序，但不生成 put-total-price 处理程序。最后，定义了 compute-total-price 处理程序，将通过加上适当的税计算订单的总价格。例如，如果订单价格是 10.00，销售税是 0.05，则总价格是 10.50。以下代码演示了对槽设置的限制：

```
CLIPS>
(make-instance order1 of ORDER
   (ID #001)
   (order-price 10.00)
   (sales-tax 0.05))↵
[order1]
CLIPS> (send [order1] put-ID #002)↵
[MSGFUN3] ID slot in [order1] of ORDER: write
access denied.
[PRCCODE4] Execution halted during the actions of
message-handler put-ID primary in class ORDER
FALSE
CLIPS>
```

注意当创建 order1 实例时可以设置 ID 槽，但之后再使用 put-ID 处理程序就不行。类似的，total-price 可以使用 get-total-price 处理程序获取，但不能使用 put-total-price 处理程序设置：

```
CLIPS> (send [order1] get-total-price)↵
0.0
CLIPS> (send [order1] put-total-price 10.50)↵
[MSGFUN1] No applicable primary message-handlers
found for put-total-price.
FALSE
CLIPS>
```

为了设置正确的价格必须调用 compute-total-price 处理程序：

```
CLIPS> (send [order1] compute-total-price)↵
10.5
CLIPS> (send [order1] get-total-price)↵
10.5
CLIPS>
```

可以人为地定义 get-total-price 处理程序以便计算总价的正确值时，不用显式地调用 compute-total-price 处理程序。当使用 create-accessor 属性定义类的 get-处理程序时会创建一个下面形式的消息处理程序：

```
(defmessage-handler <class> get-<slot-name>
            primary ()
?self:<slot-name>)
```

类似的，put-处理程序使用以下格式：

```
(defmessage-handler <class> put-<slot-name>
            primary (?value)
(bind ?self:<slot-name> ?value))
```

在这两种情况下，<class>是自定义类名，<slot-name>是类中的槽名。如果 ORDER 类中的 total-price 的 create-accessor 属性被修改为? NONE，则 get-total-price 处理程序可以定义为在调用时计算价格：

```
(defclass ORDER
    (is-a USER)
    (slot ID (access initialize-only)
             (default-dynamic (gensym)))
    (slot total-price (create-accessor ?NONE)
                      (default 0.0))
    (slot order-price (default 0.0))
    (slot sales-tax (default 0.0)))

(defmessage-handler ORDER get-total-price ()
    (send ?self compute-total-price))

(defmessage-handler ORDER get-total-price ()
    (send ?self compute-total-price))
```

创建 ORDER 实例然后查询总价格会返回一个最新值：

```
CLIPS>
(make-instance order2 of ORDER
   (ID #002)
   (order-price 20.00)
   (sales-tax 0.05))↵
[order2]
CLIPS> (send [order2] get-total-price)↵
21.0
CLIPS>
```

11.10 BEFORE、AFTER 和 AROUND 消息处理程序

类的功能并不总是符合要求，你也无法修改使之符合需要。因为别的代码可能依靠这些类而维护它的功能，或者你对代码不够熟悉而无法修改它们。在这样的情况下，你可定义一个新类继承父类中你所需要的功能，然后修改这个类使之具有你想要的新功能。下面重新检查 ORDER 例子的代码，看如何创建一个新类使得总价格信息总是最新的：

```
(defclass ORDER
    (is-a USER)
    (slot ID (access initialize-only))
    (slot total-price (create-accessor read)
                      (default 0.0))
    (slot order-price (default 0.0))
    (slot sales-tax (default 0.0)))

(defmessage-handler ORDER compute-total-price ()
    (bind ?self:total-price
       (* ?self:order-price
         (+ 1 ?self:sales-tax))))
```

接下来需要定义一个新类，在这个类中实现特殊的功能，这个类命名为 MY-ORDER：

```
(defclass MY-ORDER
   (is-a ORDER))
```

可以采用的一种方法是完全重定义 get-total-price 消息处理程序以组合新旧功能：

```
(defmessage-handler MY-ORDER get-total-price ()
   (send ?self compute-total-price)
   ?self:total-price)
```

这种情况下，我们在返回 total-price 槽值之前调用 compute-total-price 消息处理程序。这样做有几个问题。首先，它不能运行。total-price 槽是 private 的，所以我们不能在 ORDER 类外使用 ?self 引用来存取它。也不能用（send ?self get-total-price）代替 ?self：total-price 引用，因为这会调用 MY-ORDER 处理程序而不是 ORDER 处理程序。

其次，这个方法造成代码重复。本例中是一个很小很简单的代码：?self：total-price，但情况可能不一定这样。程序重复的代码越少，则初始 ORDER 处理程序改变时 MY-ORDER 处理程序不继承新功能的可能性就越大。因此，这样重定义处理程序的方法不可行。

可以轻松地修改处理程序获得预期的效果：

```
(defmessage-handler MY-ORDER get-total-price ()
   (send ?self compute-total-price)
   (call-next-handler))
```

call-next-handler 函数将调用下一个消息处理程序，当前的消息处理程序会被重置。没有必要提供任何参数。当前处理程序使用的参数会传递给它。如果在发送 MY-ORDER 实例一个 get-total-price 消息时监视消息处理程序，可以看到 MY-ORDER 和 ORDER 都调用了 get-total-price 消息处理程序：

```
CLIPS>
(make-instance order3 of MY-ORDER
   (ID #003)
   (order-price 10.00)
   (sales-tax 0.05))↵
CLIPS> (watch message-handlers)↵
CLIPS> (send [order3] get-total-price)↵
HND >> get-total-price primary in class MY-ORDER
       ED:1 (<Instance-order3>)
HND >> compute-total-price primary in class ORDER
       ED:2 (<Instance-order3>)
HND << compute-total-price primary in class ORDER
       ED:2 (<Instance-order3>)
HND >> get-total-price primary in class ORDER
       ED:1 (<Instance-order3>)
HND << get-total-price primary in class ORDER
       ED:1 (<Instance-order3>)
HND << get-total-price primary in class MY-ORDER
       ED:1 (<Instance-order3>)
10.5
CLIPS>
```

Before 和 after 处理程序

除了 primary 类型消息处理程序，这是没有指定处理程序类型时的默认值，CLIPS 也提供了 before、after 和 around 处理程序类型。处理程序类型在处理程序名后声明，每一个类可以有每种类型的处理程序。before 消息处理程序类型指明在 primary 消息处理程序前执行的处理程序。下面是一个 MY-ORDER 类中 before 消息处理程序的例子：

```
(defmessage-handler MY-ORDER get-total-price
                    before ()
(send ?self compute-total-price))
```

不像前面 primary 类型的 get-total-price 消息处理程序，在这个处理程序里我们只发送 compute-total-price 消息，而且不调用 call-next-handler 函数。不用调用这个函数是因为 MY-ORDER 的 before 处理程序将首先被执行，然后 ORDER 的 primary 处理程序将被调用。如果删除先前定义的 MY-ORDER 中的 primary 处理程序，可以通过监视消息处理程序看到：

```
CLIPS>
(make-instance order4 of MY-ORDER
   (ID #004)
   (order-price 10.00)
   (sales-tax 0.05))↵
[order4]
CLIPS> (watch message-handlers)↵
CLIPS> (send [order4] get-total-price)↵
HND >> get-total-price before in class MY-ORDER
       ED:1 (<Instance-order4>)
HND >> compute-total-price primary in class ORDER
       ED:2 (<Instance-order4>)
HND << compute-total-price primary in class ORDER
       ED:2 (<Instance-order4>)
HND << get-total-price before in class MY-ORDER
       ED:1 (<Instance-order4>)
HND >> get-total-price primary in class ORDER
       ED:1 (<Instance-order4>)
HND << get-total-price primary in class ORDER
       ED:1 (<Instance-order4>)
10.5
CLIPS>
```

MY-ORDER 的 before 类型 get-total-price 消息处理程序首先被调用，它调用了 ORDER 的 primary 类型 compute-total-price 消息处理程序，然后这些处理程序都退出。最后，ORDER 的 primary 类型 get-total-price 消息处理程序被调用，这将返回最新的 total-price 槽值。

这是在 get-total-price 处理程序被调用前计算正确的 total-price 槽值，我们还可以在修改了 sales-tax 或 order-price 槽值之后计算 total-price。我们将通过定义下列的 after 消息处理程序来完成这项工作：

```
(defmessage-handler MY-ORDER put-sales-tax
                    after (?value)
   (if (numberp (send ?self get-order-price))
      then
      (send ?self compute-total-price)))

(defmessage-handler MY-ORDER put-order-price
                    after (?value)
   (if (numberp (send ?self get-sales-tax))
      then
      (send ?self compute-total-price)))
```

这些处理程序将在 primary 类型的 put-sales-tax 和 put-order-price 处理程序调用之后调用 compute-total-price 处理程序。在每个处理程序中，我们必须检查用于 compute-total-price 消息处理程序的其他槽值已正确定义为数，因为在整个实例创建过程中，这些槽不会设置为默认值 0.0。如果删除先前的 before 类型 get-total-price 处理程序，可以通过监视消息处理程序看到：

```
CLIPS>
(make-instance order5 of MY-ORDER
   (ID #005)
   (order-price 10.00)
   (sales-tax 0.05))↵
[order5]
CLIPS> (send [order5] get-total-price)↵
10.5
CLIPS> (watch message-handlers)↵
CLIPS> (send [order5] put-order-price 20.00)↵
HND >> put-order-price primary in class ORDER
       ED:1 (<Instance-order5> 20.0)
HND << put-order-price primary in class ORDER
       ED:1 (<Instance-order5> 20.0)
HND >> put-order-price after in class MY-ORDER
       ED:1 (<Instance-order5> 20.0)
HND >> get-sales-tax primary in class ORDER
       ED:2 (<Instance-order5>)
HND << get-sales-tax primary in class ORDER
       ED:2 (<Instance-order5>)
HND >> compute-total-price primary in class ORDER
       ED:2 (<Instance-order5>)
```

```
HND << compute-total-price primary in class ORDER
       ED:2 (<Instance-order5>)
HND << put-order-price after in class MY-ORDER
       ED:1 (<Instance-order5> 20.0)
20.0
CLIPS> (send [order5] get-total-price)↵
HND >> get-total-price primary in class ORDER
       ED:1 (<Instance-order5>)
HND << get-total-price primary in class ORDER
       ED:1 (<Instance-order5>)
21.0
CLIPS>
```

在创建了实例后，可以看到 total-price 槽值为最新值。此时我们没有监视消息处理程序，因为在实例创建过程中有一些我们不感兴趣的消息发出。如果调用 put-order-price 改变 order-price 槽值，可以看到 ORDER 的 primary 类型 put-order-price 消息处理程序首先被调用然后退出。接下来 MY-ORDER 的 after 类型 put-order-price 消息处理程序被调用，这将调用 ORDER 的 primary 类型 get-sales-tax 消息处理程序以确定 sales-tax 槽设置为恰当的值。因为 sales-tax 槽有有效值，ORDER 的 primary 类型 compute-total-price 处理程序被调用来计算正确的价格，然后 primary 类型的 compute-total-price 和 after 类型的 put-order-price 处理程序都退出。发送 get-total-price 消息随后返回正确的值。

Around 处理程序

在 after 消息处理程序中检查 order-price 和 sales-tax 槽具有数字类型值是件麻烦的事情，我们寻找不同的方法来完成这项工作。首先，从 after 消息处理程序中去掉这些值的检查：

```
(defmessage-handler MY-ORDER put-sales-tax
                    after (?value)
   (send ?self compute-total-price))

(defmessage-handler MY-ORDER put-order-price
                    after (?value)
   (send ?self compute-total-price))
```

COOL 提供的第 4 个也是最后一个消息处理程序类型是 around 消息处理程序。这个消息处理程序类型也称为 wrapper，因为它围绕在其他消息处理程序周围，可以在它们之前或之后执行。around 消息处理程序是一种融合了 before 和 after 处理程序，同时允许你在消息处理过程中中止消息的处理程序。下面是一个 compute-total-price 消息的 around 处理程序：

```
(defmessage-handler MY-ORDER compute-total-price
                    around ()
   (if (or (not (numberp
                (send ?self get-order-price)))
           (not (numberp
                (send ?self get-sales-tax))))
     then
     (return))
   (call-next-handler))
```

首先，around 处理程序检查 order-price 和 sales-tax 槽是否为数。如果不是，则处理程序返回不做任何操作。此时，compute-total-price 的 before、primary 或 after 处理程序都不会被调用。另一方面，如果 order-price 和 sales-tax 槽都为数，call-next-handler 将被激发，其他的消息处理程序将被调用。在这种情况下，只有 ORDER 的 primary 类型 compute-total-price 消息处理程序被调用。下面的对话演示了这一过程。因为输出内容太多，在消息处理程序各部分之间插入了解释：

```
CLIPS> (watch message-handlers)↵
CLIPS>
(make-instance order5 of MY-ORDER
   (ID #005)
   (order-price 10.00)
   (sales-tax 0.05))↵
HND >> create primary in class USER
```

```
        ED:1 (<Instance-order5>)
HND << create primary in class USER
        ED:1 (<Instance-order5>)
HND >> put-ID primary in class ORDER
        ED:1 (<Instance-order5> #005)
HND << put-ID primary in class ORDER
        ED:1 (<Instance-order5> #005)
```

使用 make-instance 创建了 order5 实例。在创建实例时，激发了另一个系统定义的消息处理程序 create。一旦新的实例被创建，make-instance 调用中指定的值就会放在实例槽中。首先调用 put-ID 消息处理程序把 #005 放入 ID 槽：

```
HND >> put-order-price primary in class ORDER
        ED:1 (<Instance-order5> 10.0)
HND << put-order-price primary in class ORDER
        ED:1 (<Instance-order5> 10.0)
HND >> put-order-price after in class MY-ORDER
        ED:1 (<Instance-order5> 10.0)
HND >> compute-total-price around in class MY-ORDER
        ED:2 (<Instance-order5>)
HND >> get-order-price primary in class ORDER
        ED:3 (<Instance-order5>)
HND << get-order-price primary in class ORDER
        ED:3 (<Instance-order5>)
HND >> get-sales-tax primary in class ORDER
        ED:3 (<Instance-order5>)
HND << get-sales-tax primary in class ORDER
        ED:3 (<Instance-order5>)
HND >> get-order-price primary in class ORDER
        ED:3 (<Instance-order5>)
HND << get-order-price primary in class ORDER
        ED:3 (<Instance-order5>)
HND << compute-total-price around in class MY-ORDER
        ED:2 (<Instance-order5>)
HND << put-order-price after in class MY-ORDER
        ED:1 (<Instance-order5> 10.0)
```

接着，ORDER 的 primary 类型 put-order-price 消息处理程序被调用以存储 10.0 在 order-price 槽中。一旦这个处理程序完成，MY-ORDER 的 after 类型 put-order-price 处理程序将被调用。这个处理程序发送 compute-total-price 消息，这将激发 MY-ORDER 的 around 类型 compute-total price 处理程序。在 around 处理程序中，ORDER 的 primary 类型 get-order-price 处理程序被调用以判断 order-price 是否为数（由于被设置为 10.0，是数）。接下来，ORDER 的 primary 类型 get-sales-tax 处理程序被调用以判断 sales-tax 是否为数。结果为否，因为这个槽还没有在 make-instance 中得到值。所以 around 处理程序重新调用 ORDER 的 primary 类型 get-order-price 处理程序，并返回这个值。在这种情况下，around 类型的 computer-order-price 处理程序没有调用 primary 处理程序就中止了。最后，还剩下 MY-ORDER 的 after 类型 put-order-price 处理程序：

```
HND >> put-sales-tax primary in class ORDER
        ED:1 (<Instance-order5> 0.05)
HND << put-sales-tax primary in class ORDER
        ED:1 (<Instance-order5> 0.05)
HND >> put-sales-tax after in class MY-ORDER
        ED:1 (<Instance-order5> 0.05)
HND >> compute-total-price around in class MY-ORDER
        ED:2 (<Instance-order5>)
HND >> get-order-price primary in class ORDER
        ED:3 (<Instance-order5>)
HND << get-order-price primary in class ORDER
        ED:3 (<Instance-order5>)
HND >> get-sales-tax primary in class ORDER
        ED:3 (<Instance-order5>)
HND << get-sales-tax primary in class ORDER
        ED:3 (<Instance-order5>)
HND >> compute-total-price primary in class ORDER
        ED:2 (<Instance-order5>)
```

```
HND << compute-total-price primary in class ORDER
       ED:2 (<Instance-order5>)
HND << compute-total-price around in class MY-ORDER
       ED:2 (<Instance-order5>)
HND << put-sales-tax after in class MY-ORDER
       ED:1 (<Instance-order5> 0.05)
```

接下来，ORDER 的 primary 类型 put-sales-tax 消息处理程序被调用以存储 0.05 到 sales-tax 槽中。一旦这个处理程序完成，MY-ORDER 的 after 类型 put-sales-tax 处理程序被调用。这个处理程序发送 compute-total-price 消息，这将激发 MY-ORDER 的 primary 类型 compute-total-price 处理程序。在 around 处理程序中，ORDER 的 primary 类型 get-order-price 处理程序被调用以判断 order-price 槽是否为数。是，因此接下来 ORDER 的 primary 类型 get-sales-tax 处理程序被调用以判断 sales-tax 是否为数（由于被设置为 0.05，是数）。around 处理程序调用 call-next-handler 函数激发 ORDER 的 primary 类型 compute-total-price 处理程序，这将计算 total-price 槽的最新值。最后，ORDER 的 primary 类型 compute-total-price 处理程序结束，随后 MY-ORDER 的 around 类型 compute-total-price 处理程序结束，再接着，MY-ORDER 的 after 类型 put-sales-tax 处理程序结束：

```
HND >> init primary in class USER
       ED:1 (<Instance-order5>)
HND << init primary in class USER
       ED:1 (<Instance-order5>)
[order5]
CLIPS>
```

最后，init 消息被送到新创建的实例，返回实例名。init 消息处理程序是另一个系统定义的消息处理程序。在实例被创建、槽值被初始化之后调用。

重置消息处理程序参数

通过使用 override-next-handler 命令可以重置传递给消息处理程序的参数。这个命令的语法如下：

```
(override-next-handler <expression>*)
```

其中，<expression>是发送给下一个消息处理程序的替换参数。作为使用这个命令的例子，思考如何在外币中处理订单。再次假设由于各种原因不能直接修改 ORDER 类。如果 ORDER 实例中的 order-price 值是美元，可以有特殊逻辑建立在价格为美元的基础上。例如，海运费用可能在不同订单中相差 100 美元以上。

处理不同货币可以采用的一种方法是定义一个 ORDER 类的继承类，能够自动把在外币和美元之间转换。下面的 FOREIGN-ORDER 自定义类和相关联的处理程序实现了这个方法：

```
(defclass FOREIGN-ORDER
    (is-a ORDER)
    (slot exchange-rate (default 1.0)))

(defmessage-handler FOREIGN-ORDER get-order-price
                    around ()
    (* ?self:exchange-rate (call-next-handler)))

(defmessage-handler FOREIGN-ORDER put-order-price
                    around (?value)
(override-next-handler
    (/ ?value ?self:exchange-rate)))
```

FOREIGN-ORDER 类有一个 exchange-rate 槽用来表示外币和美元之间的汇率兑换值。例如，如果兑换值是 2，那么 10 美元将等于 20 个外币单位。对于实际应用，我们会同时提供一个槽存放外币单位（例如欧元），但在这个例子中没有必要。

FOREIGN-ORDER 类的 around 类型 get-order-price 处理程序通过 call-next-handler 命令激发 ORDER 类的 primary 类型 get-order-price 处理程序，返回值乘以 exchange-rate 槽值，然后返回这个结果值。FOREIGN-ORDER 类的 around 类型 put-order-price 处理程序使用 override-next-handler 激发 ORDER 类的

primary 类型 put-order-price 处理程序，但它传送的不是这个外币值，而是把这个值除以兑换值，然后传送该值，即美元值。

通过监视消息处理程序，可以监视 FOREIGN-CURRENCY 类的行为。首先，创建这个类的一个实例：

```
CLIPS> (unwatch all)↵
CLIPS>
(make-instance order6 of FOREIGN-ORDER
   (ID #006)
   (exchange-rate 2)
   (sales-tax .10))↵
[order6]
CLIPS>
```

接下来，我们把 order-price 槽的值改为外币 20.00

```
CLIPS> (watch message-handlers)↵
CLIPS> (send [order6] put-order-price 20.00)↵
HND >> put-order-price around in class FOREIGN-
ORDER
       ED:1 (<Instance-order6> 20.0)
HND >> put-order-price primary in class ORDER
       ED:1 (<Instance-order6> 10.0)
HND << put-order-price primary in class ORDER
       ED:1 (<Instance-order6> 10.0)
HND << put-order-price around in class FOREIGN-
ORDER
       ED:1 (<Instance-order6> 20.0)
10.0
CLIPS>
```

首先 FOREIGN-ORDER 的 around 类型 put-order-price 处理程序被以值 20 激发。接着，ORDER 的 primary 类型 put-order-price 处理程序被以值 10 激发，该值是 override-next-handler 调用的结果。然后美元值被传回作为 send 命令的结果。

获取 order-price 槽类似：

```
CLIPS> (send [order6] get-order-price)↵
HND >> get-order-price around in class
     FOREIGN-ORDER
     ED:1 (<Instance-order6>)
HND >> get-order-price primary in class ORDER
     ED:1 (<Instance-order6>)
HND << get-order-price primary in class ORDER
     ED:1 (<Instance-order6>)
HND << get-order-price around in class
     FOREIGN-ORDER
     ED:1 (<Instance-order6>)
20.0
CLIPS>
```

首先，FOREIGN-ORDER 类的 around 类型 get-order-price 处理程序被激发，然后使用 call-next-handler 调用 ORDER 的 primary 类型 get-order-price 处理程序。primary 处理程序返回美元值 10 乘以 around 处理程序的兑换值，最后返回外币单位值 20。

发送一个 print 消息给 FOREIGN-CURRENCY 实例显示货币值以美元而不是外币值存放：

```
CLIPS> (unwatch all)↵
CLIPS> (send [order6] print)↵
[order6] of FOREIGN-ORDER
(ID #006)
(total-price 0.0)
(order-price 10.0)
(sales-tax 0.1)
(exchange-rate 2)
CLIPS>
```

处理程序执行顺序

通过给 MY-ORDER 类附加处理程序，我们使用了 4 种不同的技术来修改 ORDER 类的功能：重置

primary 处理程序、定义 before 处理程序、定义 after 处理程序以及定义 after 和 around 处理程序。一个明显的问题是：哪一个最好？在这个例子中，使用 before 处理程序可能是最好的解决方案。它用最少的代码，并且比重置 primary 处理程序（使用了第二少数目的代码）有一个重要的优势：除非出现错误或者 around 处理程序终止消息，否则属于一个类或其父类的所有继承 before 和 after 消息处理程序都会被调用。这意味着一个类可以使用 before 和 after 处理程序而不用重置 primary 处理程序来略微改变父类的行为。子类不能阻止一个 before 和 after 处理程序的执行，除非终止消息而阻止所有 before、after 和 primary 处理程序的执行。如果你想通过重定义一个新类并重置 primary 处理程序来改变一个已有类的行为，那么 primary 处理程序同时也可以被子类重置。除非重置类调用 call-next-handler 函数，否则 primary 处理程序不能执行。这并不是说你不可重置 primary 处理程序，但是如果有一些特别的功能不想继承的类重置它，那么绝对应考虑把这个功能放在 before 或者 after 处理程序。

假定有多个被一个类继承的 before、after 和 around 处理程序，那么各种消息处理程序执行的顺序就显得很重要。当一个消息被 send 命令送到实例，所有可用的消息处理程序都会按以下步骤判定：

1. 如果存在没有调用过的 around 处理程序，那么，激发最特定的一个；否则，执行步骤 2。如果激发的 around 处理程序调用了 call-next-handler 或 override-next-handler，那么重复本步骤；否则执行步骤 6。

2. 如果存在没有调用过的 before 处理程序，那么，激发最特定的一个，等待完成，然后重复本步骤；否则，执行步骤 3。忽略 before 处理程序返回值，因为没法直接把这个值传给其他处理程序。

3. 如果存在没有调用过的 primary 处理程序，那么，激发最特定的一个；否则，执行步骤 4。如果激发的 primary 处理程序调用了 call-next-handler 或 override-next-handler，那么重复本步骤；否则执行步骤 4。除非至少有一个 primary 处理程序可用，否则你无法把消息传递给实例。

4. 允许每一个 primary 处理程序完成并返回，然后从已调用 primary 处理程序列表中删除这些处理程序。如果一个 primary 处理程序再次调用 call-next-handler 或 override-next-handler，那么返回步骤 3。一旦所有 primary 处理程序完成，记住最后执行的 primary 处理程序返回值，执行步骤 5。

5. 如果存在没有调用过的 after 处理程序，那么，激发最不特定的一个，等待它完成，然后重复本步骤；否则，执行步骤 6。忽略 after 处理程序的返回值，因为没法直接把这个值传给其他处理程序。

6. 如果无 around 处理程序可调用，那么执行步骤 7。否则，允许每一个 around 处理程序完成并返回，然后从已调用 around 处理程序列表中删除这些处理程序。如果一个 around 处理程序再次调用 call-next-handler 或 override-next-handler，那么返回步骤 1。一旦所有 around 处理程序完成，记住最后执行的 around 处理程序返回值，执行步骤 7。

7. 最后的 send 命令返回值是步骤 6 中最特定的 around 处理程序的返回值。如果没有 around 处理程序，那么采用步骤 4 的最特定的 primary 处理程序的返回值。call-next-handler 或 override-next-handler 函数的返回值是下一个 around 或 primary 处理程序调用最后的行为。一个处理程序可以忽略这个值或者用它作为返回值。

图 11.2 显示了从步骤 1 到步骤 7 的等价流程图。步骤 1 从图左上角的方框开始。图 11.3 显示了从步骤 3 到步骤 4 执行与发送给实例的消息相关联的可用 primary 处理程序的等价流程图。在没有可用的 around、before 或 after 处理程序（也就是，只有 primary 处理程序）情况下，图 11.2 流程图简化为图 11.3 的流程图。

下面结构用来演示处理程序执行顺序：

```
(defclass A
   (is-a USER))
(defmessage-handler A msg1 primary ()
 (return msg1~A))
(defmessage-handler A msg1 before ())
(defmessage-handler A msg1 after ())
(defmessage-handler A msg1 around ()
```

```
   (call-next-handler))

(defclass B
   (is-a A))

(defmessage-handler B msg1 primary ()
 (return msg1-B))

(defmessage-handler B msg1 before ())

(defmessage-handler B msg1 after ())

(defmessage-handler B msg1 around ()
   (call-next-handler))
```

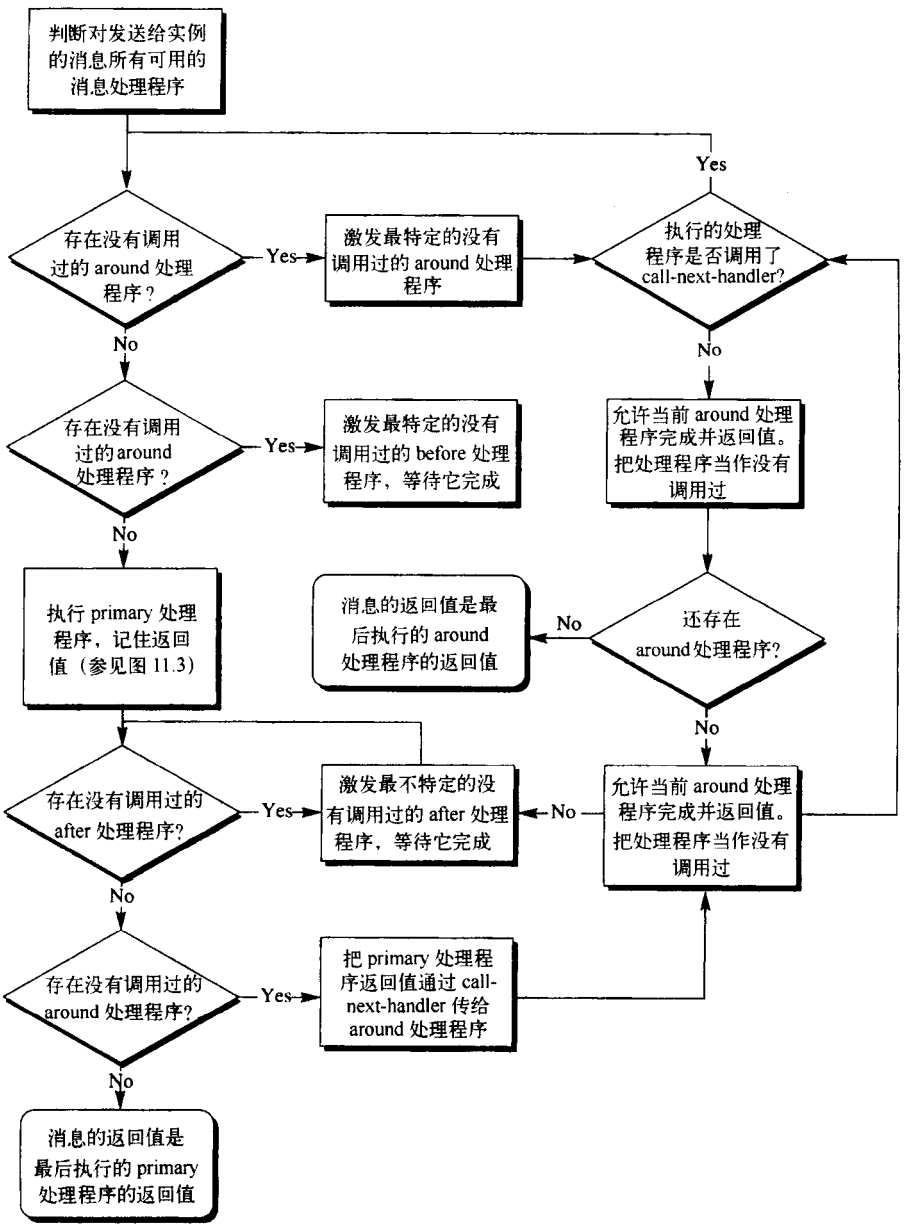


图 11.2 判定处理程序执行顺序

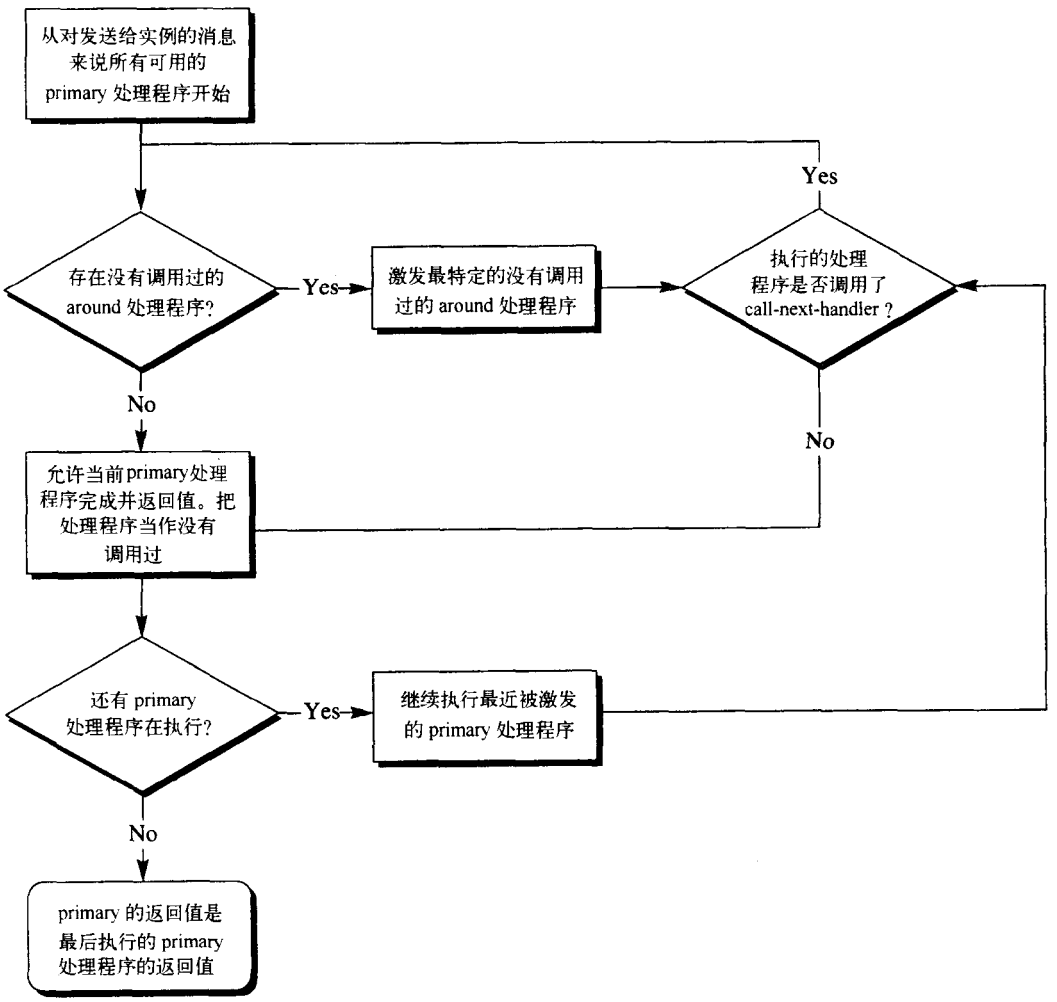


图 11.3 判定 primary 处理程序执行顺序

类 B 是类 A 的一个子类。每一个类都有自己的 primary、before、after 和 around 处理程序。首先，我们创建这两个类的实例：

```
CLIPS> (make-instance [a1] of A)↵
[a1]
CLIPS> (make-instance [b1] of B)↵
[b1]
CLIPS>
```

现在监视消息处理程序，把 msg1 消息发送给实例 a1。

```
CLIPS> (watch message-handlers)↵
CLIPS> (send [a1] msg1)↵
HND >> msg1 around in class A
       ED:1 (<Instance-a1>)
HND >> msg1 before in class A
       ED:1 (<Instance-a1>)
HND << msg1 before in class A
       ED:1 (<Instance-a1>)
HND >> msg1 primary in class A
       ED:1 (<Instance-a1>)
HND << msg1 primary in class A
       ED:1 (<Instance-a1>)
```

```
HND >> msg1 after in class A
       ED:1 (<Instance-a1>)
HND << msg1 after in class A
       ED:1 (<Instance-a1>)
HND << msg1 around in class A
       ED:1 (<Instance-a1>)
msg1-A
CLIPS>
```

对这个消息有 4 个可用的消息处理程序：类 A 的 msg1 primary、before、after 和 around 处理程序。从步骤 1 开始，类 A 的 msg1 around 处理程序执行。这个处理程序调用 call-next-handler 函数，所以重复步骤 1。由于没有剩下的未调用 around 处理程序，执行步骤 2。类 A 的 msg1 before 处理程序执行并返回。由于没有其他的未调用 before 处理程序，执行步骤 3。类 A 的 msg1 primary 消息处理程序执行。它没有调用 call-next-handler，所以执行步骤 4。类 A 的 msg1 primary 处理程序完成并返回符号 msg1-A。由于所有的 primary 处理程序已经执行完毕，执行步骤 5。类 A 的 msg1 after 处理程序执行并返回。由于没有其他的未调用 after 处理程序，执行步骤 6。类 A 的 msg1 around 处理程序执行完成。这个处理程序的最后行为是调用 call-next-handler 函数，所以返回值是下一个被激发的 around 或 primary 处理程序的返回值。在本例中，是具备返回值 msg1-A 的类 A 的 msg1 primary 处理程序，所以这也是 around 处理程序的返回值。由于所有的 around 处理程序都已经完成，执行步骤 7。最后 send 命令的返回值是类 A 的 msg1 around 处理程序返回的 msg1-A.

下面我们把 msg1 消息发送给实例 b1：

```
CLIPS> (send [b1] msg1)↵
HND >> msg1 around in class B
       ED:1 (<Instance-b1>)
HND >> msg1 around in class A
       ED:1 (<Instance-b1>)
HND >> msg1 before in class B
       ED:1 (<Instance-b1>)
HND << msg1 before in class B
       ED:1 (<Instance-b1>)
HND >> msg1 before in class A
       ED:1 (<Instance-b1>)
HND << msg1 before in class A
       ED:1 (<Instance-b1>)
HND >> msg1 primary in class B
       ED:1 (<Instance-b1>)
HND << msg1 primary in class B
       ED:1 (<Instance-b1>)
HND >> msg1 after in class A
       ED:1 (<Instance-b1>)
HND << msg1 after in class A
       ED:1 (<Instance-b1>)
HND >> msg1 after in class B
       ED:1 (<Instance-b1>)
HND << msg1 after in class B
       ED:1 (<Instance-b1>)
HND << msg1 around in class A
       ED:1 (<Instance-b1>)
HND << msg1 around in class B
       ED:1 (<Instance-b1>)
msg1-B
CLIPS>
```

对这个消息有 8 个可用的消息处理程序：类 A 以及类 B 的 msg1 primary、before、after 和 around 处理程序。从步骤 1 开始，类的 B msg1 around 处理程序执行，因为它是最特定的 around 处理程序。这个处理程序调用 call-next-handler 函数，所以重复步骤 1。下一个最特定的 around 处理程序是类 A 的 msg1 around 处理程序。这个处理程序也调用了 call-next-handler 函数，所以重复步骤 1。由于没有剩下的未调用 around 处理程序，执行步骤 2。类 B 的 msg1 before 处理程序是最特定的 before 处理程序，于是首先执行并完成。下一个最特定的 around 处理程序是类 A 的 msg1 before 处理程序，于是执行并返回。由于没有其他的未调用 before 处理程序，执行步骤 3。类 B 的 msg1 primary 消息处理程序执行，因为它是最

特定的 primary 处理程序。由于它没有调用 call-next-handler，所以没有执行类 A 的 msg1 primary 处理程序，执行步骤 4。类 B 的 msg1 before 处理程序完成并返回符号 msg1-B。所有的 primary 处理程序已经执行完毕，因此执行步骤 5。与 before 处理程序的顺序相反，类 A 的 msg1 after 处理程序首先执行并返回，因为它是最不特定的 after 处理程序。接着类 B 的 msg1 after 处理程序执行并返回。由于没有其他的未调用 after 处理程序，执行步骤 6。类 A 的 msg1 around 处理程序执行完成。这个处理程序的最后行为是调用 call-next-handler 函数，所以返回值是下一个被激发的 around 或 primary 处理程序的返回值。在本例中，是具有返回值 msg1-B 的类 B 的 msg1 primary 处理程序，所以这也是 around 处理程序的返回值。此时，类 B 的 msg1 around 处理程序执行完成，它的最后行为也是调用 call-next-handler 函数，因此，它的返回值就是类 A 的 msg1 around 处理程序的返回值 msg1-B。由于所有的 around 处理程序都已经完成，执行步骤 7。最后 send 命令的返回值是类 B 的 msg1 around 处理程序返回的 msg1-B.

preview-send 命令用来显示对发送给指定类的实例的消息可用的处理程序。它的语法是：

```
(preview-send <defclass-name> <message-name>)
```

例如，取代监视消息处理程序命令并发送 msg1 消息给实例 b1，可执行下面命令：

```
CLIPS> (preview-send B msg1)↵
>> msg1 around in class B
| >> msg1 around in class A
| | >> msg1 before in class B
| | << msg1 before in class B
| | >> msg1 before in class A
| | << msg1 before in class A
| | >> msg1 primary in class B
| | | >> msg1 primary in class A
| | | << msg1 primary in class A
| | << msg1 primary in class B
| | >> msg1 after in class A
| | << msg1 after in class A
| | >> msg1 after in class B
| | << msg1 after in class B
| << msg1 around in class A
<< msg1 around in class B
CLIPS>
```

所有的可用处理程序按照在每个 around 和 primary 处理程序调用 call-next-handler 的情况下被调用的顺序列出。缩行层次表示了处理程序调用的嵌套程度。与监视消息处理程序类似，≫和≪符号表示处理程序开始和结束执行。竖线|符号串连接着处理程序执行的开始和结束部分，该处理程序必须调用 call-next-handlers 以允许父类处理程序的执行。如果你要看对于给定的消息和实例类的可用处理程序列表，调用 preview-send 比监视消息处理程序然后发送给实例一个消息更加方便。第一种方法显示哪些可调用，而第二种方法显示实际的调用。

11.11 实例创建、初始化和删除消息处理程序

正如前面提到的，有几个预先已定义消息处理程序可以从 USER 类继承，包括 create、init 和 delete 消息处理程序。create 消息处理程序在实例创建后、赋给槽默认值或者槽重置之前调用。init 消息处理程序在重置所有未设置为默认值的剩余槽值后被调用。delete 消息处理程序既可以显式地调用以删除一个实例，也可以在调用 make-instance 并指明一个已存在的实例名后自动调用。一般，当定义类时，不要重置 USER 类中已定义的 primary 消息处理程序，但是定义响应这些处理程序的 before 和 after 处理程序将非常有用。

首先，我们看看什么情况下定义一个 after 类型 init 处理程序会非常有用。考虑下面自定义类：

```
(defclass PERSON
   (is-a USER)
   (slot first-name (type STRING)
   (access initialize-only))
   (slot middle-name (type STRING)
```

```
   (access initialize-only))
   (slot last-name (type STRING)
   (access initialize-only))
   (slot full-name (type STRING)
   (access initialize-only)))
```

我们已经在定义 PERSON 自定义类时规定了一个人的名字在一个 PERSON 实例创建时定义，而且之后不能改变的限制。Full-name 槽是 first-name、middle-name 和 last-name 槽的连接，中间用空格隔开。既然 full-name 槽只可以在初始化时指定，我们必须连同其他的 name 一起给它提供一个适当的值：

```
CLIPS>
(make-instance [p1] of PERSON
   (first-name "John")
   (middle-name "Quincy")
   (last-name "Public")
   (full-name "John Quincy Public"))↵
[p1]
CLIPS>
```

我们可以提供一个 after 类型的 init 消息处理程序自动地从其他槽值构成全名，而不需要自己定义 full-name 槽的值：

```
(defmessage-handler PERSON init after ()
   (bind ?self:full-name
     (str-cat ?self:first-name " "
              ?self:middle-name " "
              ?self:last-name)))
```

通过监视 init 消息处理程序，可以看到 PERSON 的 after 类型 init 消息处理程序在 USER 的 init 消息处理程序之后激发，存储在 full-name 槽中的值是预期的值：

```
CLIPS> (watch message-handlers USER init)↵
CLIPS> (watch message-handlers PERSON init)↵
CLIPS>
(make-instance [p2] of PERSON
   (first-name "Jane")
   (middle-name "Paula")
   (last-name "Public"))↵
HND >> init primary in class USER
       ED:1 (<Instance-p2>)
HND << init primary in class USER
       ED:1 (<Instance-p2>)
HND >> init after in class PERSON
       ED:1 (<Instance-p2>)
HND << init after in class PERSON
       ED:1 (<Instance-p2>)
[p2]
CLIPS> (send [p2] get-full-name)↵
"Jane Paula Public"
CLIPS>
```

注意，定义一个 before 类型 init 处理程序通常没有多大意义，因为所有的实例槽值都还没有被初始化。

Storage 属性

作为使用 after 类型 create 和 before 类型 delete 处理程序的一个例子，我们将使用 storage 槽属性。如果这个属性设置为 local，这是默认值，则创建的每个实例具有自己的存放槽值的空间。如果 storage 属性设置为 shared，则一个类的所有实例共享存放槽值的空间。如果某个实例的槽值被修改了，则所有实例的槽值都被修改。例如，下面的类有一个使用了共享空间的 count 槽：

```
(defclass INSTANCE-COUNTER
   (is-a USER)
   (slot count (storage shared) (default 1)))
```

如果我们创建两个实例然后改变其中一个的 count 槽值，那么其他实例的 count 槽值也被改变，如下命令序列所示：

```
CLIPS> (make-instance i1 of INSTANCE-COUNTER)↵
[i1]
CLIPS> (make-instance i2 of INSTANCE-COUNTER)↵
[i2]
CLIPS> (send [i1] get-count)↵
1
CLIPS> (send [i2] get-count)↵
1
CLIPS> (send [i1] put-count 2)↵
2
CLIPS> (send [i1] get-count)↵
2
CLIPS> (send [i2] get-count)↵
2
CLIPS>
```

有了这个功能，可以定义 after 类型 create 和 before 类型 delete 消息处理程序，每次创建这个类的实例时增加 count 槽值，每次删除这个类的实例时减少 count 槽值：

```
(defmessage-handler INSTANCE-COUNTER create
                    after ()
   (if (integerp ?self:count)
      then
      (send ?self put-count (+ ?self:count 1))))

(defmessage-handler INSTANCE-COUNTER delete
                    before ()
   (bind ?self:count (- ?self:count 1)))
```

有必要解释一下 after 类型 create 消息处理程序。在应用任何默认值之前 create 消息将被发送给实例。在第一个 INSTANCE-COUNTER 实例创建的情形中，当 create 消息处理程序被激发时，count 槽值将被置为 nil。试图把这个值加一将导致错误，因为 nil 不是数。after 类型 create 消息处理程序的 if 函数检查这种情况以避免它。由于一个共享槽的默认值在第一个实例创建时应用，我们把 count 槽的默认值设为 1，所以第一个实例自动得到正确值。对于随后的 after 类型 create 处理程序的激发，count 槽的值为整数，所以 if 函数的 integerp 调用成功，count 槽值将增加。

下面的命令序列演示了创建两个新的 INSTANCE-COUNTER 实例将正确地增加 count 槽值到 4，删除其中的一个实例将把 count 槽值减为 3：

```
CLIPS> (make-instance i3 of INSTANCE-COUNTER)↵
[i3]
CLIPS> (make-instance i4 of INSTANCE-COUNTER)↵
[i4]
CLIPS> (send [i1] get-count)↵
4
CLIPS> (send [i4] delete)↵
TRUE
CLIPS> (send [i1] get-count)↵
3
CLIPS>
```

11.12 修改和复制实例

CLIPS 提供了几个实例命令，它们提供了和事实的修改、复制命令类似的功能。这些命令的语法如下：

```
(modify-instance
  <instance-expression> <slot-overrides>*)
(message-modify-instance
  <instance-expression> <slot-overrides>*)
(active-modify-instance
  <instance-expression> <slot-overrides>*)
(active-message-modify-instance
```

```
  <instance-expression> <slot-overrides>*)
(duplicate-instance <instance-expression>
                    [to <instance-name-expression>]
                    <slot-overrides>*)
(message-duplicate-instance <instance-expression>
                    [to <instance-name-expression>]
                    <slot-overrides>*)
(active-duplicate-instance <instance-expression>
                    [to <instance-name-expression>]
                    <slot-overrides>*)
(active-message-duplicate-instance
                    <instance-expression>
                    [to <instance-name-expression>]
                    <slot-overrides>*)
```

其中，<instance-expression>是将被修改或复制的实例，<slot-overrides>是修改槽列表，对于复制命令，<instance-name-expression>是可选的复制出的实例的新名字。

最基本的实例修改命令是 modify-instance 命令，下面是使用它的一个例子：

```
CLIPS> (unwatch all)↵
CLIPS> (clear)↵
CLIPS>
(defclass PERSON
   (is-a USER)
   (slot first-name)
   (slot last-name))
CLIPS>
(make-instance [p1] of PERSON
   (first-name "Jeff")
   (last-name "Public"))↵
[p1]
CLIPS> (watch messages)↵
CLIPS> (watch slots)↵
CLIPS>
(modify-instance [p1] (first-name "Jack")
                      (last-name "Private"))↵
MSG >> direct-modify ED:1 (<Instance-p1>
                           <Pointer-0x0062b200>)
::= local slot first-name in instance p1 <- "Jack"
::= local slot last-name in instance p1
<- "Private"
MSG << direct-modify ED:1 (<Instance-p1>
                           <Pointer-0x0062b200>)
TRUE
CLIPS> (unwatch all)↵
CLIPS> (send [p1] print)↵
[p1] of PERSON
(first-name "Jack")
(last-name "Private")
CLIPS>
```

注意 direct-modify 消息被发送给实例［p1］，但 put-first-name 和 put-last-name 消息没有发送。direct-modify 消息处理程序是另一个自动创建的系统定义消息处理程序。当使用 modify-instance 命令时，槽值直接被 direct-modify 消息处理程序改变而不需要激发消息传递。其结果是和槽相关的 primary、after、before 和 around 类型 put-处理程序都不会被激发。

message-modify-instance 命令和 modify-instance 命令有相同的参数和语法，但它使用消息传递来改变槽值。例如：

```
CLIPS> (watch messages)↵
CLIPS> (message-modify-instance [p1]
                                (first-name "Jeff")
                                (last-name "Public"))↵
MSG >> message-modify ED:1 (<Instance-p1>
                            <Pointer-0x0062b1c0>)
MSG >> put-first-name ED:2 (<Instance-p1> "Jeff")
MSG << put-first-name ED:2 (<Instance-p1> "Jeff")
MSG >> put-last-name ED:2 (<Instance-p1> "Public")
```

```
MSG << put-last-name ED:2 (<Instance-p1> "Public")
MSG << message-modify ED:1 (<Instance-p1>
                            <Pointer-0x0062b1c0>)
TRUE
CLIPS>
```

在这种情况下，因为使用了 message-modify-instance，所以 put-first-name 和 put-last-name 消息处理程序用来改变槽值。使用了另一个系统定义消息处理程序 message-modify 而不是 direct-modify 来修改实例。

最后，两个另外的实例修改命令对对象模式匹配进行控制：active-modify-instance 和 active-message-modify-instance。参数和语法与其他的修改命令相同。使用 modify-instance 和 message-modify-instance 命令，对象匹配直到所有的槽值改变完成后才进行。使用 active-modify-instance 和 active-message-modify-instance 命令，对象匹配在每个槽值改变后进行。典型地，你会希望在实例的所有槽值修改完后再激发对象模式匹配，因为这将提高性能，但是 active 修改命令可以用在需要交替进行的情况下。

还提供了 4 种配对的复制实例命令：duplicate-instance、message-duplicate-instance、active-duplicate-instance 和 active-message-modify-instance。这些命令和对应的修改命令有类似的行为和参数；但是，修改实例是参数传送，而复制实例是创建实例，并应用槽重置。可以提供可选的复制实例名；否则将自动生成。这些命令的返回值是复制的实例。下面例子演示了如何使用其中的两个复制命令：

```
CLIPS> (duplicate-instance [p1]
         (first-name "Jack"))↵
MSG >> direct-duplicate ED:1 (<Instance-p1> [gen2]
                              <Pointer-0x006b8840>)
MSG << direct-duplicate ED:1 (<Instance-p1> [gen2]
                              <Pointer-0x006b8840>)
[gen2]
CLIPS> (message-duplicate-instance [p1] to [p3]
         (first-name "Jill"))↵
MSG >> message-duplicate ED:1 (<Instance-p1> [p3]
                              <Pointer-0x006b8840>)
MSG >> create ED:2 (<Instance-p3>)682
MSG << create ED:2 (<Instance-p3>)
MSG >> put-first-name ED:2 (<Instance-p3> "Jill")
MSG << put-first-name ED:2 (<Instance-p3> "Jill")
MSG >> put-last-name ED:2 (<Instance-p3> "Public")
MSG << put-last-name ED:2 (<Instance-p3> "Public")
MSG >> init ED:2 (<Instance-p3>)
MSG << init ED:2 (<Instance-p3>)
MSG << message-duplicate ED:1 (<Instance-p1> [p3]
                              <Pointer-0x006b8840>)
[p3]
CLIPS> (instances)↵
[p1] of PERSON
[gen2] of PERSON
[p3] of PERSON
For a total of 3 instances.
CLIPS>
```

在 duplicate-instance 例子中，由于没有指定实例名，使用了生成名 [gen2]。在 message-duplicate 例子中，指定名字 [p3] 作为复制实例名。两个其他的系统定义消息处理程序 direct-duplicate 和 message-duplicate 根据调用不同被分别发送消息。在本例中，是调用 message-duplicate 和 active-message-duplicate，则 create、init 和 put-消息将会被发送以复制和重置新创建实例的槽值。

最后，可以为系统定义消息处理程序 direct-modify、message-modify、direct-duplicate 和 message-duplicate 定义 around、before 和 after 消息处理程序。但是，由于槽重置以 EXTERNAL-ADRESS 的形式传送，你只能用外部 C 语言来解码这个参数，通过定义处理程序还不足以实现。

11.13 类和类属函数

类属函数的参数限制并不仅限于 CLIPS 提供的预先已定义类型。用户定义类也可作为参数限制。例如，下面的类可以用来表示一个复数：

```
(defclass COMPLEX
   (is-a USER)
   (slot real)
   (slot imaginary))
```

COMPLEX 类表示了以下格式的复数：

```
a + bi
```

其中，a 和 b 是实数，i 是 −1 的平方根。在 COMPLEX 类中，a 的值存储在 real 槽，b 的值存储在 imaginary 槽。复数的加法很简单：

```
(a + bi) + (c + di) = (a + c) + (b + d)i
```

下面方法加载 + 函数以实现两个复数的相加：

```
(defmethod + ((?c1 COMPLEX) (?c2 COMPLEX))
   (make-instance of COMPLEX
      (real (+ (send ?c1 get-real)
               (send ?c2 get-real)))
      (imaginary (+ (send ?c1 get-imaginary)
                    (send ?c2 get-imaginary)))))
```

这个方法所做的就是首先把复数的实部相加，然后把虚部相加，并存放在新创建的 COMPLEX 实例中。创建 COMPLEX 类的实例并把它们相加显示这个方法正确运行：

```
CLIPS> (make-instance c1 of COMPLEX
       (real 3) (imaginary 4))↵
[c1]
CLIPS> (make-instance c2 of COMPLEX
       (real 5) (imaginary 6))↵
[c2]
CLIPS> (+ [c1] [c2])↵
[gen1]
CLIPS> (send [gen321] print)↵
[gen1] of COMPLEX
(real 8)
(imaginary 10)
CLIPS>
```

11.14 实例集合查询函数

除了对象的模式匹配，也可以直接查询 COOL 得到满足指定条件集的实例集合。为了演示这些函数，使用下面结构表达一个家族树：

```
(defclass PERSON
   (is-a USER)
   (slot full-name)
   (slot gender)
   (multislot children))

(defclass FEMALE
   (is-a PERSON)
   (slot gender (access read-only)
                (storage shared)
                (default female)))

(defclass MALE
   (is-a PERSON)
   (slot gender (access read-only)
                (storage shared)
                (default male)))

(definstances people
   ([p1] of MALE (full-name "John Smith")
         (children [p5]))
   ([p2] of FEMALE (full-name "Jan Smith")
         (children [p5]))
   ([p3] of MALE (full-name "Bob Jones")
         (children [p6]))
```

```
([p4] of FEMALE (full-name "Pam Jones")
        (children [p6]))
([p5] of MALE (full-name "Frank Smith")
        (children [p7]))
([p6] of FEMALE (full-name "Sue Jones")
        (children [p7]))
([p7] of MALE (full-name "Dave Smith")))
```

判定是否满足查询要求

最简单的查询函数是 any-instancep。它的语法为：

```
(any-instancep <instance-set-template> <query>)
```

其中< instance-set-template>是要被搜索的类的说明，<query>是一个布尔表达式，表示那些类的匹配实例必须满足的条件。如果发现一个实例集合满足查询，any-instancep 函数返回符号 TRUE；否则，返回 false。

< instance-set-template>的语法是：

```
(<instance-set-member-template>+)
```

其中<instance-set-member-template>是：

```
(<single-field-variable> <class-name-expression>+)
```

<single-field-variable>是实例约束给的变量，一次或多次出现的<class-name-expression>是包含被分别约束给<single-field-variable>的实例的类。

使用 any-instancep 的最简单测试是判断一个类是否存在实例：

```
CLIPS> (any-instancep ((?p PERSON)) TRUE)↵
FALSE
CLIPS> (reset)↵
CLIPS> (any-instancep ((?p PERSON)) TRUE)↵
TRUE
CLIPS>
```

在这个例子中，PERSON 类不存在任何实例，直到执行 reset 命令，创建了自定义实例 pepole 的实例。在查询中使用的 TRUE 意味着任何赋给 ?p 的值都满足查询。

如果只创建了 MALE 和 FEMALE 实例，可以通过指明这两个类判断是否有 PERSON 类实例存在：

```
CLIPS> (any-instancep ((?p MALE FEMALE)) TRUE)↵
TRUE
CLIPS>
```

可以使用下面 3 个查询判断是否存在男人：

```
CLIPS> (any-instancep ((?p MALE)) TRUE)↵
TRUE
CLIPS>
(any-instancep ((?p PERSON))
               (eq (send ?p get-gender) male))↵
TRUE
CLIPS>
(any-instancep ((?p PERSON)) (eq ?p:gender male))↵
TRUE
CLIPS>
```

在第一个例子中，我们找类 MALE 的实例，这些实例的 gender 槽包括 male。在第二个例子中，我们发送 get-gender 消息给指派给变量 ?p 的 PERSON 实例，使用 eq 函数判断其值是否为 male。在第三个例子中，演示了槽速记表达可以与实例集合模板中约束的变量一起使用。

也可以在模板中指定超过一个实例集合成员：

```
CLIPS> (any-instancep ((?f FEMALE) (?p PERSON))
                      (member$ ?p ?f:children))↵
TRUE
CLIPS>
```

这个例子测试是否存在母亲。对于每一个 FEMALE 类和 PERSON 类的实例组合，应用查询来判断 PERSON ?p 是否是 FEMALE ?f 的一个孩子。

判定满足查询的实例

find-instance 和 find-all-instances 查询函数的语法与 any-instancep 函数类似：

```
(find-instance <instance-set-template> <query>)
(find-all-instances
  <instance-set-template> <query>)
```

find-instance 查询函数返回的是包含满足查询的第一个实例集合的多字段值，而不是 TRUE 或 FALSE。如果没有实例集合满足查询，多字段值为空。类似的，find-all-instances 函数返回包含满足查询的所有实例集合的多字段值。例如：

```
CLIPS> (find-instance ((?p MALE)) TRUE)↵
([p1])
CLIPS> (find-all-instances ((?p MALE)) TRUE)↵
([p1] [p3] [p5] [p7])
CLIPS>
(find-all-instances ((?p MALE))
                    (eq ?p:gender female))↵
()
CLIPS>
```

在模板中指定了超过一个集合成员的情况下，你必须编程才能分组返回值。例如：

```
CLIPS>
(find-all-instances ((?f FEMALE) (?p PERSON))
                    (member$ ?p ?f:children))↵
([p2] [p5] [p4] [p6] [p6] [p7])
CLIPS>
```

实例 [p2] 和 [p5] 属于第一个实例集合，[p4] 和 [p6] 属于第二个实例集合，[p6] 和 [p7] 属于第三个实例集合。下面自定义函数演示了如何编程把这些实例集成员分组：

```
(deffunction print-mother-message (?query-result)
  (bind ?iterations
     (div (length$ ?query-result) 2))
(loop-for-count (?i ?iterations) do
   (bind ?mother (nth$
      (- (* 2 ?i) 1) ?query-result))
   (bind ?child (nth$ (* 2 ?i) ?query-result))
   (printout t (send ?mother get-full-name)
           " is the mother of "
           (send ?child get-full-name)
           "." crlf)))
```

查询返回值的长度除以每个成员集合的实例数得到在成员集合数上的迭代次数。loop-for-count 函数与 nth $ 函数结合来抽取成员集合中的每个实例值。结合前面 find-all-instances 调用得到的返回值，使用 print-mother-message 得到以下结果：

```
CLIPS>
(print-mother-message
   (find-all-instances ((?f FEMALE) (?p PERSON))
                       (member$ ?p ?f:children)))↵
Jan Smith is the mother of Frank Smith.
Pam Jones is the mother of Sue Jones.
Sue Jones is the mother of Dave Smith.
FALSE
CLIPS>
```

对满足查询的实例采取行为

do-for-instance、do-for-all-instances 和 delayed-do-for-all-instances 查询函数允许你在满足查询的实例集合上采取行为。它们的语法如下：

```
(do-for-instance
  <instance-set-template> <query> <expression>*)
(do-for-all-instances
  <instance-set-template> <query> <expression>*)
(delayed-do-for-all-instances
  <instance-set-template> <query> <expression>*)
```

do-for-instance 函数在满足查询的第一个实例集合上采取指定的行为。例如：

```
CLIPS>
(do-for-instance ((?f FEMALE) (?p PERSON))
                 (member$ ?p ?f:children)
   (printout t ?f:full-name " is the mother of "
               ?p:full-name "." crlf))↵
Jan Smith is the mother of Frank Smith.
CLIPS>
```

do-for-all-instance 函数在满足查询的所有实例集合采取指定的行为。例如：

```
CLIPS>
(do-for-all-instances ((?f FEMALE) (?p PERSON))
                      (member$ ?p ?f:children)
   (printout t ?f:full-name " is the mother of "
               ?p:full-name "." crlf))↵
Jan Smith is the mother of Frank Smith.
Pam Jones is the mother of Sue Jones.
Sue Jones is the mother of Dave Smith.
CLIPS>
```

注意使用 do-for-all-instances 是一个简单的获得在前面的例子中的相同功能的方法。前面的例子使用 find-all-instance 函数和 print-mother-message 自定义函数。

delayed-do-for-all-instances 函数和 do-for-all-instances 函数类似，除了它首先找出所有满足查询的实例集合，然后对每个实例集合执行行为。相反的，do-for-all-instance 函数对每个实例集合应用查询，如果查询满足则执行行为。只有在执行的行为改变了一个还没被处理的实例集合的查询结果时，这两个函数才会得到不同的结果。例如，改变一个被查询引用的实例槽值可以导致两个函数得到不同的结果。

当没有实例集合满足查询时，所有允许行为的查询函数的返回值是符号 FALSE。否则，其返回值是在最后一个满足查询的实例集合上执行最后一个行为的结果。此外，允许行为的查询函数可以使用 break 或 return 函数提前终止。

11.15 多继承

目前为止，所有的自定义类例子中，都只有单个类作为 is-a 属性。这些都是**单继承**（single inheritance）例子。在 is-a 属性中也可以指定超过一个类。当出现这种情况时，称为**多继承**（multiple inheritance）。在指定的类没有共同的槽或消息处理程序的情况下，多继承基本上等于在定义类的父类中使用单继承。例如，下面的例子中，类 D 通过多继承直接继承 x、y 和 z 槽：

```
CLIPS> (clear)↵
CLIPS>
(defclass A
   (is-a USER)
   (slot x))↵
CLIPS>
(defclass B
   (is-a USER)
   (slot y))↵
CLIPS>
(defclass C
   (is-a USER)
   (slot z))↵
CLIPS>
(defclass D
   (is-a C B A))↵
```

```
CLIPS> (make-instance [d1] of D (x 3) (y 4) (z 5))↵
[d1]
CLIPS> (send [d1] print)↵
[d1] of D
(x 3)
(y 4)
(z 5)
CLIPS>
```

在下面的例子中，类B和C作为中间类允许类D通过单继承间接地继承槽x和y：

```
CLIPS> (clear)↵
CLIPS>
(defclass A
   (is-a USER)
   (slot x))↵
CLIPS>
(defclass B
   (is-a A)
   (slot y))↵
CLIPS>
(defclass C
   (is-a B)
   (slot z))↵
CLIPS>
(defclass D
   (is-a C))↵
CLIPS> (make-instance [d1] of D (x 3) (y 4) (z 5))↵
[d1]
CLIPS> (send [d1] print)↵
[d1] of D
(x 3)
(y 4)
(z 5)
CLIPS>
```

注意，虽然在两个例子中类D继承了相同的槽值，但类B和C不同。由于在第二个例子中重定义了类B和C，通过使用单继承而不是多继承可以得到类D相同的最终结果。典型的多继承是在不能改变预先已存在类的情况下使用的。

多继承冲突

大多数多继承的实际例子中类所继承的父类没有共享槽和消息处理程序（不是从USER类继承）。在这种情况下，在父类之间没有需要化解的冲突，在类中定义的与父类冲突的槽和消息处理程序可以进行重置，就像在单继承中一样。

看下面一个简单的例子，其中使用了多继承，父类之间的槽和消息处理程序定义有冲突：

```
CLIPS> (clear)↵
CLIPS>
(defclass A
   (is-a USER)
   (slot x (default 3))
   (slot y (default 4)))↵
CLIPS>
(defmessage-handler A compute ()
   (* ?self:y 10))↵
CLIPS>
(defclass B
   (is-a USER)
   (slot y (default 1))
   (slot z (default 5)))↵
CLIPS>
(defmessage-handler B compute ()
   (+ ?self:y 3))↵
CLIPS> (defclass C (is-a A B))↵
CLIPS> (defclass D (is-a B A))↵
CLIPS>
```

在这个例子中，类C和D直接继承类A和B。因为类A和B对y槽和compute消息处理程序的定义有冲突，接下来看看如何解决这个冲突：

```
CLIPS> (make-instance [c1] of C)↵
[c1]
CLIPS> (make-instance [d1] of D)↵
[d1]
CLIPS> (send [c1] print)↵
[c1] of C
(z 5)
(x 3)
(y 4)
CLIPS> (send [d1] print)↵
[d1] of D
(x 3)
(y 1)
(z 5)
CLIPS>
```

对实例［c1］和［c2］显示的结果槽值有两点应该注意。首先是槽打印的顺序，其次是y槽的值。槽打印的顺序有一点差别，显示出父类在is-a属性中的次序会影响类定义结果。而这两个实例最明显的差别是y槽的值在［c1］中是4而在［d1］中是1。由于类C首先继承类A，它使用了类A中y槽的默认值而不是类B的默认值。类似的，由于类D首先继承B，它使用类B中y槽的默认值而不是类A的默认值。相似的行为也会发生在compute消息处理程序中：

```
CLIPS> (send [c1] put-y 3)↵
3
CLIPS> (send [d1] put-y 3)↵
3
CLIPS> (send [c1] compute)↵
30
CLIPS> (send [d1] compute)↵
6
CLIPS>
```

尽管两个实例的y槽都赋给值3，实例［c1］的compute消息处理程序计算结果是30（3×10），而［d1］是6（3+3）。同样地，这是因为［c1］使用类A的compute消息处理程序，而［d1］使用类B的compute消息处理程序。

在is-a属性中指定的类不是相同的用户定义父类的简单情况下，类出现的顺序决定了多个同名槽或消息处理程序的优先级。在这个例子中，类C由于先指定A再指定B，所以类A定义优先级高于类B。在类D中，由于先指定B再指定A，所以类B定义优先级高于类A。

更复杂的多继承情况留给读者自己探索。本书配套光盘中的“Basic Programming Guide”包括了各种情况下多继承冲突如何化解的完整说明及例子。一般而言，如果使用多继承创建类，在is-a属性中类出现的顺序影响了被定义的类的行为，则你需考虑你是否把问题复杂化了。

保存和恢复槽值

下面看一个实用的多继承例子：定义一个RESTORABLE类，与其他类一起来保存和恢复一个实例的槽值。为了实现这个类，我们需要定义另一个用来存放槽值的类：

```
(defclass SAVED-SLOT
   (is-a USER)
   (slot slot-name)
   (multislot slot-value))
```

SAVED-SLOT类具有slot-name和slot-value槽。slot-name槽用来存放保存的槽的名字。slot-value槽用来存放名字对应的槽的值。它被定义为多字段槽，因为我们既需要存放单字段槽也需要存放多字段槽的值。

RESTORABLE类定义如下：

```
(defclass RESTORABLE
   (is-a USER)
   (multislot saved-slots))
```

saved-slots 多字段槽将存放指向 SAVED-SLOT 实例的零个或多个引用，它们用来代表实例的槽值。

save 消息处理程序用来保存 RESTORABLE 实例的槽值：

```
(defmessage-handler RESTORABLE save ()
   ; Delete existing saved slots
   (progn$ (?si ?self:saved-slots)
      (send ?si delete))
   (bind ?self:saved-slots)
   ; Determine the list of slots
   (bind ?class (class ?self))
   (bind ?slots
     (delete-member$ (class-slots ?class inherit)
                     saved-slots))
   ; Create an empty list
   (bind ?list (create$))
   ; Iterate over each slot
   (progn$ (?slot ?slots)
      (bind ?value (send ?self
                   (sym-cat get- ?slot)))
      (bind ?ins (make-instance of SAVED-SLOT
                   (slot-name ?slot)
                   (slot-value ?value)))
      (bind ?list (create$ ?list ?ins)))
   ; Store the saved slots
   (bind ?self:saved-slots ?list))
```

save 消息处理程序的第一个行为是删除所有在 saved-slots 槽中存有引用的 SAVED-SLOT 实例。saved-slots 槽然后被约束为空多字段。

接着，求出将要保存的槽列表。调用 class 函数决定实例的类名，然后类名与关键字 inherit 一起传给class-slots 函数以获取包含与类相关联的所有槽名的多字段列表。最后，通过调用 delete-member $ 函数从列表中删除槽名 saved-slots。我们并不想在这个槽中保存值，因为它将用来存放所有其他槽的值。

然后，创建一个空表用来包含 SAVED-SLOT 实例。使用 progn $ 函数在要保存的实例的每个槽上迭代。首先，槽值被获取。槽名附加到 get-符号上以创建合适的消息，然后发送给实例以获取槽值。一旦槽值被获取，就创建一个 SAVED-SLOT 实例包含槽名与值，对这个实例的引用被加到要保存的槽列表中。一旦所有的槽都处理了，SAVED-SLOT 实例引用的列表就存放到 saved-slots 槽中。

与刚定义的 save 消息处理程序一起，下面 restore 消息处理程序用来恢复一个实例的槽值：

```
(defmessage-handler RESTORABLE restore ()
   (progn$ (?si ?self:saved-slots)
      (bind ?name (send ?si get-slot-name))
      (bind ?value (send ?si get-slot-value))
      (send ?self (sym-cat put- ?name) ?value)))
```

restore 消息处理程序在所有存放在 saved-slots 槽中的 SAVED-SLOT 实例上迭代。首先从 SAVED-SLOT 实例获取每个保存的槽的名字和值。槽名附加到 put-符号上以创建合适的消息，然后发送给实例以设置槽值。这个消息接着发送给实例以恢复槽值为保存值。

现在已定义了 RESTORABLE 类，我们可以检查它如何与已有的类一起用 save/restore 功能创建新的类。使用和前面例子类似的 PERSON 类：

```
(defclass PERSON
   (is-a USER)
   (slot full-name)
   (slot gender)
   (multislot children))
```

RESTORABLE-PERSON 类将继承 PERSON 和 RESTORABLE 类：

```
(defclass RESTORABLE-PERSON
   (is-a RESTORABLE PERSON))
```

为了看到 save/restore 如何工作，我们需要首先创建 RESTORABLE-INSTANCE 类的实例：

```
CLIPS> (reset)↵
CLIPS>
(make-instance [p1] of RESTORABLE-PERSON
   (full-name "Sue Jones")
   (gender female)
   (children Bob Jan))↵
[p1]
CLIPS> (send [p1] print)↵
[p1] of RESTORABLE-PERSON
(full-name "Sue Jones")
(gender female)
(children Bob Jan)
(saved-slots)
CLIPS>
```

发送一个 save 消息给实例［p1］将保存槽的现有值：

```
CLIPS> (send [p1] save)↵
([gen1] [gen2] [gen3])
CLIPS> (send [p1] print)↵
[p1] of RESTORABLE-PERSON
(full-name "Sue Jones")
(gender female)
(children Bob Jan)
(saved-slots [gen1] [gen2] [gen3])
CLIPS>
```

实例［gen1］、［gen2］和［gen3］是创建的 SAVED-SLOT 类的实例，用来保存［p1］的槽值。可以检查这些实例看看被保存的单个槽值：

```
CLIPS> (send [gen1] print)↵
[gen1] of SAVED-SLOT
(slot-name full-name)
(slot-value "Sue Jones")
CLIPS> (send [gen2] print)↵
[gen2] of SAVED-SLOT
(slot-name gender)
(slot-value female)
CLIPS> (send [gen3] print)↵
[gen3] of SAVED-SLOT
(slot-name children)
(slot-value Bob Jan)
CLIPS>
```

下面更改实例［p1］中的某些槽值：

```
CLIPS> (send [p1] put-full-name "Sue Smith")↵
"Sue Smith"
CLIPS> (send [p1] put-children Bob Jan Paul)↵
(Bob Jan Paul)
CLIPS> (send [p1] print)↵
[p1] of RESTORABLE-PERSON
(full-name "Sue Smith")
(gender female)
(children Bob Jan Paul)
(saved-slots [gen1] [gen2] [gen3])
CLIPS>
```

当发送给实例一个 restore 消息时，会恢复原来的槽值：

```
CLIPS> (send [p1] restore)↵
(Bob Jan)
CLIPS> (send [p1] print)↵
[p1] of RESTORABLE-PERSON
(full-name "Sue Jones")
(gender female)
(children Bob Jan)
(saved-slots [gen1] [gen2] [gen3])
CLIPS>
```

11.16 自定义类和自定义模块

和其他结构类似，自定义类结构可以在模块间输入和输出。前面讨论的输入和输出语句可以输入或输出所有结构，同样也可应用于自定义类。另外，使用下面任一语句可以显式地指定输入或输出哪些类：

```
(export defclass ?ALL)
(export defclass ?NONE)
(export defclass <deffunction-name>+)

(import <module-name> defclass ?ALL)
(import <module-name> defclass ?NONE)
(import <module-name> defclass <defclass-name>+)
```

如果一个类被输入或输出，那么所有与其相关联的自定义消息处理程序结构也会被输入或输出。不能输入或输出某个指定消息处理程序。一个没有从别的模块输入自定义类结构的模块可以用相同的名字创建该类。这个准则的例外情况是预先已定义系统类，例如 USER 类，对所有模块都可见，不能进行重定义。

正如与事实相关联的模块就是与事实相关联的自定义模板的定义模块，与实例相关联的模块是与实例相关联的自定义类的定义模块。不过，每一个模块都有自己的“名字空间”（namespace）用来保持实例名字唯一。这意味着一个模块不能有两个同名实例，但不同的模块可以有同名实例。例如：

```
CLIPS> (defmodule A)↵
CLIPS>
(defclass A::ACLASS (is-a USER)
  (export defclass ?ALL))↵
CLIPS> (make-instance [same] of ACLASS)↵
[same]
CLIPS> (send [same] print)↵
[same] of A::ACLASS
CLIPS> (defmodule B)↵
CLIPS>
(defclass B::BCLASS (is-a USER)
  (export defclass ?ALL))↵
CLIPS> (make-instance [same] of BCLASS)↵
[same]
CLIPS> (send [same] print)↵
[same] of B::BCLASS
CLIPS> (set-current-module A)↵
B
CLIPS> (send [same] print)↵
[same] of A::ACLASS
CLIPS>
```

注意，在模块 B 中定义实例 [same] 没有导致模块 A 中的实例 [same] 被删除，而如果在模块 A 中有同名实例创建则会引起删除操作。

正如下面例子所演示的，一个输出模块的实例名字空间对输入模块并不是自动可见的：

```
CLIPS> (defmodule C (import A defclass ?ALL)
                    (import B defclass ?ALL))↵
CLIPS> (send [same] print)↵
[MSGPASS2] No such instance same in function send.
FALSE
CLIPS>
```

在这种情况下，尽管模块 A 和 B 都有一个 [same] 实例，但模块 C 仅仅从自己的实例名字空间中找寻指定的实例，所以没有找到。在另一个模块中引用一个实例的方法是把模块名和模块分隔符作为实例名的一部分。例如：

```
CLIPS> (send [A::same] print)↵
[same] of A::ACLASS
CLIPS> (send [B::same] print)↵
[same] of B::BCLASS
CLIPS>
```

在这种情况下，由于模块作为实例名的一部分，CLIPS就知道使用哪一个名字空间来找指定名字的实例。单独使用模块分隔符也可通过实例名确定实例：

```
CLIPS> (send [::same] print)↵
[same] of B::BCLASS
CLIPS>
```

在这种情况下，CLIPS首先搜索当前模块，然后检查每个输入了自定义类的模块。一般，在已确定实例在当前模块或者一个引入的模块中时，你可能想只使用模块分隔符。在这个例子中，实例［B::same］首先被找到，但如果输入的顺序改变，那么［A:: same］将首先被找到。

引用在其他模块中的实例看起来可能有点复杂，但事实上你只需了解它是如何工作的。典型地，程序使用实例地址而不是实例名来引用实例。当使用实例地址时，就不会像使用实例名一样不明确哪一个实例正在被引用。实例名一般用于你从命令行发送一个消息给实例，通常这个名字是唯一的，因此，如果当前模块输入了与这个实例相关联的自定义类，你可仅使用模块分隔符。

11.17 调入和保存实例

和事实类似，有几个命令用于保存实例到文件和从文件中调入实例。这些命令是save-instances、bsave-instances、load-instances、restore-instances和bload-instances。这些命令的语法如下：

```
(save-instances <file-name>
   [<save-scope> [[inherit] <class-names>+]])

(bsave-instances <file-name>
   [<save-scope> [[inherit] <class-names>+]])

(load-instances <file-name>)

(restore-instances <file-name>)

(bload-instances <file-name>)
```

其中＜save-scope＞定义为：

```
visible | local
```

load-instances命令从＜file-name＞指定的文件中调入一组实例。在文件中的实例必须遵循声明实例的自定义实例结构的基本格式。例如，如果文件“instances.dat”包含：

```
(Jack of PERSON (full-name "Jack Q. Public")
                (age 23))
(of PERSON (full-name "John Doe")
                (hair-color black))
```

那么命令：

```
(load-instances "instances.dat")
```

将调入这个文件中的实例。调用load-instances等同于使用一系列的make-instance调用。restore-instances命令和load-instances命令类似；但是，当对装入的实例进行删除、初始化或设置槽值时，它不采用消息传递的方式。这些命令的返回值都是调入的实例数，或者当命令无法存取实例文件时返回－1。

save-instance命令用来保存实例到＜file-name＞指定的文件中。这些实例以load-instances和restore-instances命令要求的格式存储。如果没有指定＜save-scope＞，或者指定为local，那么只有当前模块所定义的自定义类相应的实例会被保存到文件中。如果＜saved-scope＞指定为visible，那么所有在当前模块中可见的自定义类相应的实例都会被保存到文件中。如果指明了＜saved-scope＞，还指明了一个或多个类名。在这种情况下，只有指定的自定义类相应的实例会被保存（但是，自定义类名必须仍满足local或visible说明）。如果指定了inherit关键字，那么满足local或visible说明的指定类的子类实例也会被保存。这个命令的返回值是被保存的实例数。

除了使用二进制格式而不是文本格式存储实例外，bsave-instances和bload-instances命令与save-in-

stances 和 load-instances 命令类似。因此，你只能对使用 bsave-instances 命令创建的文件采用 bload-instances 命令。使用这些命令的好处在于，对于大量的实例，调入二进制格式会比文本格式快得多。缺点是二进制文件通常不能从一个平台移植到另一个平台。

11.18 小结

这一章介绍了 CLIPS 面向对象语言（COOL）。实例（或对象）是 CLIPS 提供的另一种数据表示。实例的属性用自定义类结构说明。过程化代码，称为消息处理程序，可以使用自定义消息处理程序结构来关联类。继承允许类使用与另一个类相关联的槽和消息处理程序。COOL 支持单继承和多继承。一个从别的类继承的类称为子类。被继承的类称为这个类的父类。子类从父类继承属性和消息处理程序，但当两个类有重复定义，子类会重置父类的定义。使用 role 属性，可以创建只能用于继承的抽象类。这些类无法创建实例。相反的，具体类可以继承也可以创建实例。COOL 预先已定义了一些基本类。大多数用户创建类继承系统类 USER。自定义实例结构允许当发出 reset 命令时创建指定的实例集合。

除了自定义模板提供的槽属性，自定义类还支持几个另外的槽属性。source 属性允许槽属性从父类继承。propagation 属性可以取消槽的继承。pattern-match 属性用来说明一个槽或类能否参与到模式匹配中。visibility 属性允许说明一个槽能否被子类的消息处理程序直接存取。access 属性直接限制槽允许的存取类型。create-accessor 属性用来控制类槽的 get-和 put-处理程序的自动创建。storage 属性允许说明一个槽值是否被类的所有实例共享，还是每个实例有自己的值。

有几个预先已定义系统消息处理程序用来创建、初始化、打印和删除实例。此外，可以创建用户定义消息处理程序。通过 send 命令发送消息名和相关参数给实例来激发消息处理程序。消息处理程序可以是 4 种类型之一：primary、around、before 或 after。primary 处理程序是响应消息的主要处理程序。before 和 after 处理程序分别被在 primary 处理程序之前和之后激发。around 处理程序也称为 wrapper 处理程序，因为它围绕在 before、after 和 primary 处理程序的周围，可以在他们之前或之后执行。around 处理程序必须显式地激发其他类型处理程序。primary 消息处理程序重置或者掩盖从父类继承相同消息的 primary 消息处理程序，不过，也可以激发被掩盖的处理程序。但父类的 around、before 和 after 消息处理程序不会被子类定义掩盖。

对象模式匹配提供了事实模式匹配所不具备的几个功能。首先，一个单一对象模式可以匹配几个类的实例。其次，在对象模式中没有说明的槽值的改变不会再激发这个模式所属的规则。第三，在逻辑条件元素的对象模式里没有说明的槽值更改不会导致相关规则的逻辑证实失效。

COOL 提供了几个实例集合查询函数，可以直接在实例集合上查询满足指定条件集的实例。这些函数中有几个还允许对查询结果采取行为。除了 CLIPS 提供的预先已定义类型外，类属函数的参数限制还可以使用用户自定义类。save-instances、bsave-instances、load-instances、restore-instances 和 bload-instances 函数用来把实例保存到文件，或从文件中调入实例。

习题

11.1 修改习题 10.3 中的程序，使之具有解释能力。在打印最优匹配的灌木之后，程序应该提示用户决定他/她是否想要关于该灌木的解释说明。若按了 return 键，则程序终止执行。若输入了某灌木的名称，则程序列出所满足的要求、不满足的要求和其他可以满足的要求。一旦打印了此解释，就再次提示用户决定他/她是否需要另一个解释机会。

11.2 修改习题 10.7 的程序使得可以安排几个学生的课程表。程序的输入从文件读入。你可以自己决定输入数据的格式；但是，它应该包含每个学生的姓名、要安排的课程、学生偏爱的教师和上课时段。程序的输出写到文件中，输出的文件包括每个学生的姓名以及他的课程表。

11.3 修改习题 10.6 的程序使得可以动态重配置菜单。例如，选择子菜单 1 的菜单选项 1 将在子菜单

2 中显示两个菜单选项，但选择子菜单 1 的菜单选项 2 将在子菜单 2 中显示 4 个菜单选项。

11.4 对下面的自定义类编写 get-side 处理程序的 before 处理程序，当 side 槽的值为默认值 unspecified 时，提示用户输入槽值：

```
(defclass SQUARE
   (is-a USER)
   (slot side (default unspecified)))
```

11.5 创建一个 ARRAY 自定义类和相关的消息处理程序以表达多维数组。提供获取和设置数组值的消息处理程序。提供可以指定数组元素默认值的功能，以及在用 make-instance 创建实例时指定初始值的功能。最后，提供显示数组内容的功能。一维数组显示为一行数据，二维数组显示为行和列。其他多位数组按其下标顺序显示为一行一组数值。

11.6 使用习题 11.15 的 ARRAY 自定义类和消息处理程序，加载 * 函数，提供两个二维数组相乘的方法。使用其他方法来检查条件错误，如两个二维数组的行和列数目不适合相乘。

11.7 创建一个 LINKED-LIST 自定义类和相关的消息处理程序，以便创建链接表。自定义类的定义应该允许其他类继承它以获取链接表功能。提供获取链表中的下一个和前一个对象、插入对象到链表中、删除链表中的一个对象、打印链表的消息处理程序。写一个程序演示如何使用一个自定义类继承 LINKED-LIST 类。

11.8 创建一个 ITERATOR 自定义类和相关的消息处理程序，可以枚举一个多字段值的字段。被枚举的值可以读取，也可以设置为 make-instance 调用的一部分以创建这个类的一个实例，但除了这个类的消息处理程序外，不能用其他方法来存取类的槽值。提供 first 消息处理程序初始化枚举表并返回枚举表的第一个值。提供 Next 和 previous 消息处理程序分别获取枚举表的下一个和前一个值。

11.9 创建 MEASUREMENT 自定义类存储一个长度的单位和数量。使用习题 10.20 和 10.21 中的方法，创建一个新方法，把两个 MEASUREMENT 实例相加，返回新的存放总和的实例。

11.10 创建 STACK 自定义类和消息处理程序以支持堆栈的压入和弹出操作。

11.11 创建 SHUFFLER 自定义类，实现 shuffle 消息处理程序，可以随机重排存储在 SHUFFLER 实例中的值表。

11.12 创建 CARD 自定义类表示扑克牌。创建 DECK 自定义类并初始化为包含 52 张扑克牌。使用习题 10.11 的 SHUFFLER 自定义类，提供 shuffle 消息处理程序来洗牌。

11.13 使用方法重置 - 、* 和 / 函数，实现 COMPLEX 类实例的减法、乘法和除法。

11.14 写一个自定义函数统计一个字符串中每个字母出现的次数，并打印统计结果。

11.15 创建 DIRECTORY 自定义类，存放姓名和电话号码。提供在目录中增加和删除表项，以及按照姓名或号码搜索，并打印所有匹配表项的消息处理程序。

第 12 章　专家系统设计实例

12.1　概述

本章提供了几个 CLIPS 程序的例子。第一个例子演示了怎样利用 CLIPS 表示不确定性。接下来的两个例子演示了怎样通过 CLIPS 来仿真其他知识表示范例：其一是关于判定树的表示，另一是有关反向链规则的表示。第 4 个也是最后一个例子是建立监视一组传感器件的简单专家系统框架。

12.2　确定性因子

CLIPS 内部没有处理不确定性的能力。然而，通过在事实和规则中放置一些处理不确定性的信息，从而使 CLIPS 具有对不确定性的处理能力是可能的。例如：MYCIN 中的不确定性机制将会通过 CLIPS 仿真。本节将演示下面的 MYCIN 规则（Firebaugh 88）是如何用 CLIPS 改写的：

```
IF
  The stain of the organism is gramneg(生物体的染色呈革兰氏阳性) and
  The morphology of the organism is rod(生物体的形态是棍状) and
  The patient is a compromised host(病人是感染体)
THEN
  There is suggestive evidence (0.6) that the
    identity of the organism is pseudomonas
  (有证据表明(0.6)这种生物是假单胞细菌属)
```

MYCIN 使用对象－属性－值（OAV）三元组来代表实际信息。这些 OAV 三元组在 CLIPS 中通过使用下列的自定义模板结构来表达（这种结构将被置于它自身的模块中来创建一种可重用的软件模块）：

```
(defmodule OAV (export deftemplate oav))

(deftemplate OAV::oav
   (multislot object (type SYMBOL))
   (multislot attribute (type SYMBOL))
   (multislot value))
```

使用这种自定义模板，前面 MYCIN 规则的 IF 部分所需要的一些事实将是：

```
(oav (object organism)
     (attribute stain)
     (value gramneg))

(oav (object organism)
     (attribute morphology)
     (value rod))

(oav (object patient)
     (attribute is a)
     (value compromised host))
```

MYCIN 也将每个事实与代表事实中可信度的确定性因子（CF）联系起来。这个确定性因子的变化范围为－1 到 1；－1 意味着该事实已知是假的；0 意味着不知道任何关于这个事实的信息（彻底的不确定性）；1 意味着该事实已知是真的。

因为 CLIPS 不能自动处理确定性因子（CF），所以，此信息也必须被维护。为此，每个事实上的一个附加槽将用来表示确定性因子。每个事实的 oav 自定义模板现在成为：

```
(deftemplate OAV::oav
   (multislot object (type SYMBOL))
   (multislot attribute (type SYMBOL))
   (multislot value)
   (slot CF (type FLOAT) (range -1.0 +1.0)))
```

例子中的事实可以是：

```
(oav (object organism)
     (attribute stain)
     (value gramneg)
     (CF 0.3))
(oav (object organism)
     (attribute morphology)
     (value rod)
     (CF 0.7))

(oav (object patient)
     (attribute is a)
     (value compromised host)
     (CF 0.8))
```

为了使 oav 事实工作正常，必须对 CLIPS 进行更进一步的修改。MYCIN 允许相同的 OAV 三元组由不同的规则推导出来。产生的这些 OAV 三元组可以组合生成单个 OAV 三元组，这个三元组组合了这两项的确定性因子。现在的 OAV 自定义模板只有当两个相等的 OAV 三元组有不同的确定因子时，才允许它们被声明（通常 CLIPS 是不允许两条重复的事实被声明的）。为使具有相同确定因子的相等 OAV 三元组被声明，可使用 set-fact-duplication 命令使阻止相等的事实不会被声明的 CLIPS 行为失效。命令：

```
(set-fact-duplication TRUE)
```

将使其行为无效。同样，命令

```
(set-fact-duplication FALSE)
```

将阻止相同的事实不会被声明。

前已述及，MYCIN 系统将会组合两条相同的 OAV 三元组使之成为一条单独的有组合的确定性的 OAV 三元组。如果两个事实中的确定性因子（表示为 CF_1 和 CF_2）都大于或等于 0，则 MYCIN 系统将使用下面的公式来计算新的确定因子：

$$\text{New Certainty} = (CF_1 + CF_2) - (CF_1 * CF_2)$$

例如，假定下列事实在事实表中：

```
(oav (object organism)
     (attribute morphology)
     (value rod)
     (CF 0.7))

(oav (object organism)
     (attribute morphology)
     (value rod)
     (CF 0.5))
```

设 $CF_1 = 0.7$ 和 $CF_2 = 0.5$，则对于这两个事实的组合的新确定因子可计算如下：

```
New Certainty = (0.7 + 0.5) - (0.7 * 0.5)
              = 1.2 - 0.35
              = 0.85
```

并且，代替原来两个事实的新事实可表示为：

```
(oav (object organism)
     (attribute morphology)
     (value rod)
     (CF 0.85))
```

既然 CLIPS 不能自动地处理事实的确定因子，故可以推断出，它也不能自动组合由不同规则推出的两个 OAV 三元组。OAV 三元组的组合可以很容易地通过一条规则来处理，该规则就是在事实表上搜索待组合的相同 OAV 三元组。下面的方法和规则演示了当两个 OAV 三元组的确定因子大于或等于 0 时，是怎样实现它们的组合的：

```
(defmethod OAV::combine-certainties
   ((?C1 NUMBER (> ?C1 0))(?C2 NUMBER (> ?C2 0)))
    (- (+ ?C1 ?C2)(* ?C1 ?C2)))

(defrule OAV::combine-certainties
   (declare (auto-focus TRUE))
   ?fact1 <- (oav (object $?o)
                  (attribute $?a)
                  (value $?v)
                  (CF ?C1))
   ?fact2 <- (oav (object $?o)
                  (attribute $?a)
                  (value $?v)
                  (CF ?C2))
   (test (neq ?fact1 ?fact2))
   =>
   (retract ?fact1)
   (modify ?fact2
(CF (combine-certainties ?C1 ?C2))))
```

请注意，在 test CE 中事实标识符？fact1 和？fact2 被相互比较。这样做是为了保证，对前两个模式使用完全相同的事实时，该规则并不匹配。函数 eq 和 neq 能比较事实的地址。同时要注意，规则中自动焦点属性一直是有效的。这就保证了为这两个三元组所满足的其他规则被允许触发之前，两个 OAV 三元组已组合了。

实现确定因子的下一步是将两个确定性因子连接起来，一个是符合规则 LHS 的事实的确定因子，另一个是被规则的 RHS 声明的事实的确定因子。在 MYCIN 中，与规则的 LHS 相联系的确定因子是由下列公式推导出来的：

```
CF(P1 or P2) = max { CF(P1), CF(P2) }
CF(P1 and P2) = min { CF(P1), CF(P2) }
CF(not P) = - CF(P)
```

其中，P、P_1 和 P_2 代表 LHS 的模式。另外，如果 LHS 的确定因子小于 0.2，则此规则被认为是不适用的，因此将不会被触发。

规则 RHS 则被声明的事实的确定性因子可这样求得：将规则的 LHS 的确定性因子乘以声明的确定性因子。下面的 CLIPS 规则是对本节开始介绍的 MYCIN 规则的翻译，它演示了 LHS 及 RHS 的确定性因子的计算过程。这条规则被放在 IDENTIFY 模块中，它从 OAV 模块中输入 oav 自定义模板。

```
defmodule IDENTIFY (import OAV deftemplate oav))

(defrule IDENTIFY::MYCIN-to-CLIPS-translation
   (oav (object organism)
        (attribute stain)
        (value gramneg)
        (CF ?C1))
   (oav (object organism)
        (attribute morphology)
        (value rod)
        (CF ?C2))
   (oav (object patient)
        (attribute is a)
        (value compromised host)
        (CF ?C3))
   (test (> (min ?C1 ?C2 ?C3) 0.2))
   =>
   (bind ?C4 (* (min ?C1 ?C2 ?C3) 0.6))
   (assert (oav (object organism)
                (attribute identity)
                (value pseudomonas)
                (CF ?C4))))
```

最后一步要求完成 MYCIN 中确定性因子的仿真。一条使用 combine-certainties 规则的 combine-certainties方法只处理确定性因子都为正的一种情况，其余组合情况由下面的公式完成：

$$\text{New Certainty} = (CF_1 + CF_2) + (CF_1 * CF_2)$$

若 $CF_1 \leqslant 0$ and $CF_2 \leqslant 0$

$$\text{New Certainty} = \frac{CF_1 + CF_2}{1 - \min\{|CF_1|, +|CF_2|\}}$$

若 $-1 < CF1 * CF2 < 0$

12.3 判定树

回忆第 3 章，判定树提供了一个十分有用的范例，该范例解决了某些类别的分类问题。判定树的解决方法是通过一系列问题和判定来调整搜索区域以减少可能的解决方案而得到的。适合用判定树解决的问题可由其特征描述，该特征值从一组预先决定的可能的解决方法中提供一个答案给这个问题。例如，一个分类学问题可能需要从一系列已知的宝石中鉴定一种宝石，或者，诊断问题需要从一组治疗方法中选择一种可能的治疗方法或从一系列可能的原因中选择一种失败的原因。因为答案必须预先设定，所以，通常来说，判定树在调度、计划或综合问题中不是很有效，因为这些问题除了在解决方法中选择外，还必须提出解决方法。

记住，一棵判定树是由结点和分枝组成的。结点代表树中的位置。当从顶端移到底端时，分枝就将父结点连到子结点；当从底端移到顶端时，它又将子结点连到父结点。树顶端没有父结点的结点称为根结点。注意，在判定树中除了根结点外，每个结点只有一个父结点。没有子结点的结点称为树叶。

判定树的叶结点代表能从树派生出来的所有可能的解决方案。这些结点称为答案结点（answer nodes），树上所有其他结点则被称为判定结点（decision nodes）。每个判定结点代表一个问题或判定。当回答问题或作判定时，它决定选取一个合适的判定分枝继续下去。在简单的判定树中，问题可以是“yes”或“no”的问题。例如：“该动物是暖血动物吗?”如果回答是“yes”，则结点的左分枝代表继续的路径；如果回答是“no”，则结点的右分枝代表继续的路径。通常，如果选择过程总是只产生一个单独的分枝的话，判定结点可使用任何准则来选择接下去的分枝。因此，判定结点可以选择一条分枝，它可对应于一组值或一个值范围、一系列情况或对应于一些从判定结点状态映射到分枝的功能。复杂的判定结点甚至可能允许回溯或概率推理。

为了说明判定树的操作，思考采用试探的方法选择合适的酒以供就餐使用：

```
IF 主菜是牛羊肉
THEN 选择红酒

IF 主菜是家禽并且是火鸡
THEN 选择红酒

IF 主菜是家禽并且不是火鸡
THEN 选择白酒

IF 主菜是鱼
THEN 选择白酒
```

关于酒的探索法的二叉判定树表示已示于图 12.1 中。此判定结点假定，每个问题的答案只能用“是”或“否”来回答。在主菜既不是牛羊肉，也不是家禽和鱼的情况下，一个默认的结果结点将“最好的颜色不知道”这个答案添加到试探的集合中。

遍历树得到答案结点的过程是十分简单的。推理过程以设置判定树的当前位置为根结点开始。若当前位置为一判定结点，则必须以某种方式回答与该判定结点相联系的问题（一般是由人来询问判定树）。如果答案为“是”，则当前位置将被设置为同当前位置的“是”分支（或左分支）相联系的子结点。

如果答案是“否”，则当前位置将被设置为同当前位置的“否”（或右）分支相联系的子结点。如

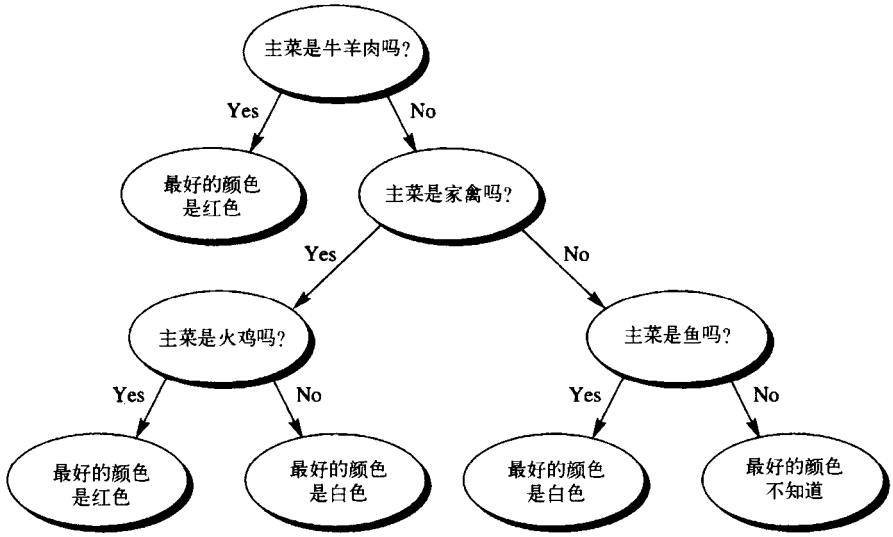


图 12.1 二叉判定树

果在任意点，答案结点成为当前结点，则此答案结点的值就是询问该判定树所得的答案。否则，处理判定结点的过程将重复进行，直至到达一个答案结点。关于这个算法的伪代码如下：

```
procedure Solve_Binary_Tree
   Set the current location in the tree
      to the root node.
   while the current location is a decision node do
      Ask the question at the current node.
      if the reply to the question is yes
         Set the current node to the yes branch.
      else
         Set the current node to the no branch.
      end if
   end do
   Return the answer at the current node.
end procedure
```

多叉判定树

二叉判定树很难表示一个有一组响应或一系列情况的判定。关于酒的二叉判定树给它的低效性提供了一个很好的例子。在主菜是鱼的情况下，在决定最好的颜色是白色之前，必须做 3 个决定："主菜是否为牛羊肉?""主菜是否为家禽?""主菜是否为鱼?"这 3 个问题必须询问。一个使判定结点简洁表达的更直接的问题是"主菜是什么?"一个能处理这个问题的判定决点必须有多个分枝，以便给出一系列可能的决定（在这里答案有牛羊肉、家禽、鱼和其他）。图 12.2 显示了图 12.1 修改后的判定树，简单修改 Solve _ Binary _ Tree 算法即允许实现多分枝：

```
procedure Solve_Tree
   Set the current tree location to the root node.
   while the current location is a decision node do
      Ask the question at the current node until
         an answer in the set of valid choices
         for this node has been provided.
      Set the current node to the child node of
         the branch associated with the choice
         selected.
   end do
   Return the answer at the current node.
end procedure
```

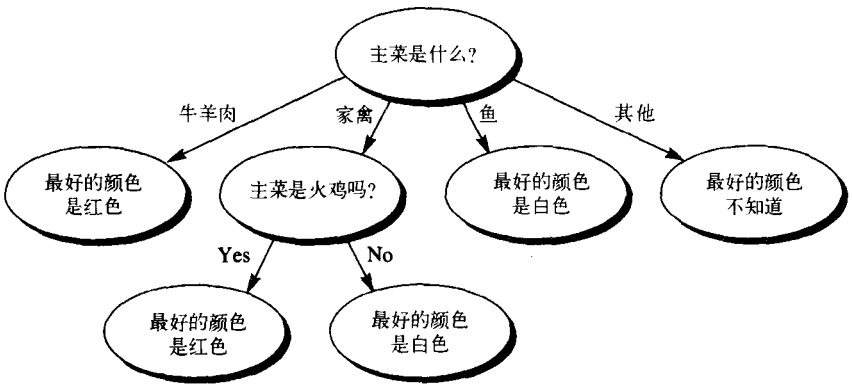


图 12.2 多叉判定树

判定学习树

有时，一棵判定树通过学习在其中添加新的知识是非常有用的，像通常用到的动物识别判定树例子就是这样。一旦判定树已得到答案，它就问答案是否正确。如果正确，则不须再做什么了。然而，如果答案不正确，那么，判定树将被修改以得出正确的答案。包含一个问题的判定结点将代替答案结点，所包含的问题不同于原来这个结点上的旧答案和没有被猜中的答案。图 12.3 显示了一棵根据特征将动物分类的判定树。这棵判定树有点过于简单（它仅知道 3 种动物的情况），并且它需要学习。

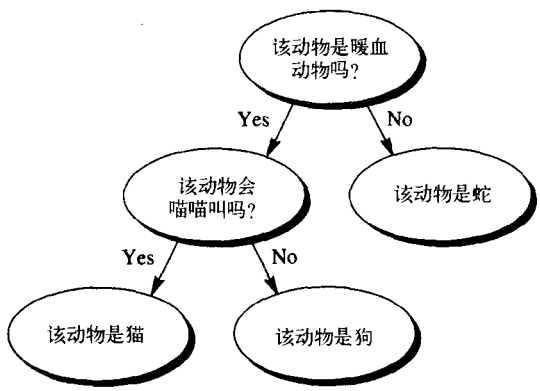


图 12.3 动物识别判定树

一棵树的猜测性对话可能按如下过程进行：

```
Is the animal warm-blooded? (yes or no) yes↵
Does the animal purr? (yes or no) no↵
I guess it is a dog
Am I correct? (yes or no) no↵
What is the animal? bird↵
What question when answered yes will distinguish
   a bird from a dog? Does the animal fly?↵
Now I can guess bird
Try again? (yes or no) no↵
```

这一对话可以一直继续下去，于是判定树可学习到越来越多的信息。图 12.4 显示了上述对话完成后判定树的表示情况。用这种方式学习的一个缺点是，在猜测适当的动物中，判定树最后不会很有层次或很有效率。一棵有效的判定树从根结点到答案结点的所有路径应具有基本相同的分枝数。

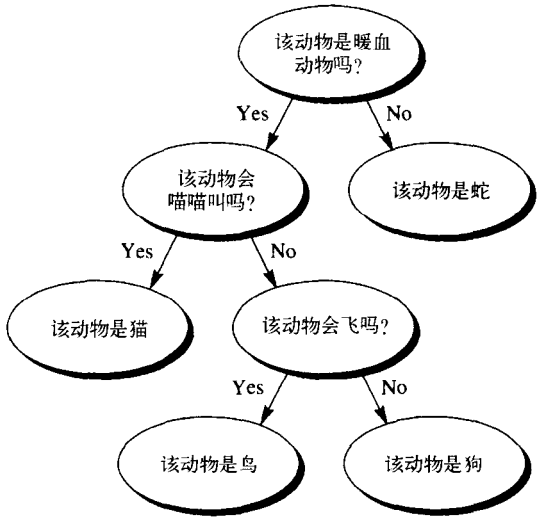


图 12.4 学习了鸟之后的动物识别判定树

将算法 Solve _ Tree 修改成具有学习能力的伪码程序是：

```
procedure Solve_Tree_and_Learn
   Set the current location in the tree
      to the root node
   while the current location is a decision node do
      Ask the question at the current node.
      if the reply to the question is yes
         Set the current node to the yes branch.
      else
         Set the current node to the no branch.
      end if
   end do
   Ask if the answer at the current node is
   correct.
   if the answer is correct
      Return the correct answer.
   else
      Determine the correct answer.
      Determine a question which when answered yes
         will distinguish the answer at the current
         node from the correct answer.
      Replace the answer node with a decision node
          that has as its no branch the current
          answer node and as its yes branch an
          answer node with the correct answer.
          The decision node's question should be
          the question which distinguishes the
          two answer nodes.
   end if
end procedure
```

一个基于规则的判定树程序

确定如何用 CLIPS 实现一棵会学习的判定树的第一步是决定应该怎样表达知识。因为判定树需要学习，所以，用事实而不用规则来表示知识也许是值得的，因为事实在学习过程中通过添加和删除来更新判定树的操作更容易。我们可以通过使用基于规则的方法实现 Solve _ Tree _ and _ Learn 算法，从而将一组 CLIPS 规则应用于判定树的遍历。

在判定树中，每个结点将表示为一个事实。下面的自定义模板将既用来表示答案结点又用来表示判定结点：

```
(deftemplate node
   (slot name)
   (slot type)
   (slot question)
   (slot yes-node)
   (slot no-node)
   (slot answer))
```

其中，name 槽是对应于此结点的唯一名字，type 槽表示结点的类别，它包含值——answer（答案）或 decision（判定）。question 槽、yes-node 槽和 no-node 槽都只能用于判定结点。question 槽表示遍历某个问题结点时所问的问题。yes-node 槽表示对问题作出肯定回答后将历经的结点。no-node 槽是对问题作否定回答后所要历经的结点。answer 槽只能用于答案结点，它是遍历一个答案结点时对判定树的答案。

由于此动物分类程序需要学习，所以，在此程序下一次被运行前有必要将从这次运行时学习到的信息存储下来。因为判定树将被设计为一个事实收集的结构，因此使用 load-facts 命令将它们（信息）存储于文件中、在程序开始运行时用 load-facts 命令声明它们并在程序运行结束时用 save-facts 命令保存它们，这样做将是很有用的。在这个程序中，事实被存入一个名叫 animal.dat 的文件中。如果图 12.3 用于初始的判定树，则文件 animal.dat 就应该包含下面的文本。注意，根结点已被标记了，其余每个结点也有一个唯一的名字。同时要注意的是，某些槽（例如，decision 槽、答案结点的 no-node 槽和 yes-node 槽）都未被指定，因为这些槽将被设为默认值（在我们的程序中并不关心这些槽中放置了什么值）。

```
(node (name root) (type decision)
      (question "Is the animal warm-blooded?")
      (yes-node node1) (no-node node2))
(node (name node1) (type decision)
      (question "Does the animal purr?")
      (yes-node node3) (no-node node4))
(node (name node2) (type answer) (answer snake))
(node (name node3) (type answer) (answer cat))
(node (name node4) (type answer) (answer dog))
```

现在，要写遍历判定树的规则。下面的规则将对学习判定树程序进行初始化：

```
(defrule initialize
   (not (node (name root)))
   =>
   (load-facts "animal.dat")
   (assert (current-node root)))
```

当根结点不在事实表中时，将执行这个 initialize 规则。这个初始化规则所进行的操作是将判定树表示调入到事实表中，并声明一个事实指示当前感兴趣的结点是根结点。

下面的规则将询问与一个判定结点相关联的问题，然后声明一个包含对该问题的答案的事实：

```
(deffunction ask-yes-or-no (?question)
   (printout t ?question " (yes or no) ")
   (bind ?answer (read))
   (while (and (neq ?answer yes) (neq ?answer no))
      (printout t ?question " (yes or no) ")
      (bind ?answer (read)))
   (return ?answer))

(defrule ask-decision-node-question
   ?node <- (current-node ?name)
   (node (name ?name)
         (type decision)
         (question ?question))
   (not (answer ?))
   =>
   (assert (answer (ask-yes-or-no ?question))))
```

只有在当前结点为判定结点时，第二模式才与当前结点相匹配。第三模式检测该问题是否还未得到回答。在规则 RHS 中，ask-yes-or-no 自定义函数重复询问答案直到得到一个肯定或否定的回答，然

后执行以下两条规则之一。

```
(defrule proceed-to-yes-branch
   ?node <- (current-node ?name)
   (node (name ?name)
         (type decision)
         (yes-node ?yes-branch))
   ?answer <- (answer yes)
   =>
   (retract ?node ?answer)
   (assert (current-node ?yes-branch)))

(defrule proceed-to-no-branch
   ?node <- (current-node ?name)
   (node (name ?name)
         (type decision)
         (no-node ?no-branch))
   ?answer <- (answer no)
   =>
   (retract ?node ?answer)
   (assert (current-node ?no-branch)))
```

其中，current-node 事实将被撤销，接着，用一个依赖于问题答案的新声明更新该事实。answer 事实也将被撤销，以便再次激活 ask-decision-node-question 规则。

下一条规则询问答案结点是否已作出正确的猜测。它所进行的操作与 ask-decision-node-question 规则类似。

```
(defrule ask-if-answer-node-is-correct
   ?node <- (current-node ?name)
   (node (name ?name) (type answer)
         (answer ?value))
   (not (answer ?))
   =>
   (printout t "I guess it is a " ?value crlf)
   (assert
(answer (ask-yes-or-no "Am I correct?"))))
```

如果答案既非肯定也非否定，则 bad-answer 规则将激活另一个 ask-if-answer-node-is-correct 规则。如果回答是肯定的或否定的，则将执行以下两条规则之一。如果答案结点的值得到验证，则声明 ask-try-again 事实以要求该用户继续。如果答案是错误的，则开始学习并声明 replace-answer-node 事实，以表明此结点名要被替换掉。不论哪一种情况都会撤销 current-node 事实和 answer 事实。

```
(defrule answer-node-guess-is-correct
   ?node <- (current-node ?name)
   (node (name ?name) (type answer))
   ?answer <- (answer yes)
   =>
   (assert (ask-try-again))
   (retract ?node ?answer))

(defrule answer-node-guess-is-incorrect
   ?node <- (current-node ?name)
   (node (name ?name) (type answer))
   ?answer <- (answer no)
   =>
   (assert (replace-answer-node ?name))
   (retract ?node ?answer))
```

接下来的 3 条规则用来判定用户是否想要继续进行。ask-try-again 规则提出“是否重试?”问题。一次重试后，如果又得出一个既不是肯定又不是否定的回答，则将再次执行 bad-answer 规则。如果回答是肯定的，那么 one-more-time 规则将会把 current-node 事实恢复为根结点而使猜测过程重新开始。如果回答是否定的，那么，使用 save-facts 命令将代表判定树的事实保存到 animal.dat 文件中。

```
(defrule ask-try-again
   (ask-try-again)
   (not (answer ?))
   =>
   (assert (answer (ask-yes-or-no "Try again?"))))

(defrule one-more-time
   ?phase <- (ask-try-again)
   ?answer <- (answer yes)
   =>
   (retract ?phase ?answer)
   (assert (current-node root)))

(defrule no-more
   ?phase <- (ask-try-again)
   ?answer <- (answer no)
   =>
   (retract ?phase ?answer)
   (save-facts "animal.dat" local node))
```

最后，如果答案是错误的，那么，以下的规则将添加一个允许判定树去学习的新判定结点。

```
(defrule replace-answer-node
   ?phase <- (replace-answer-node ?name)
   ?data <- (node (name ?name)
                  (type answer)
                  (answer ?value))
   =>
   (retract ?phase)
   ; Determine what the guess should have been
   (printout t "What is the animal? ")
   (bind ?new-animal (read))
   ; Get the question for the guess
   (printout t "What question when answered yes ")
   (printout t "will distinguish " crlf "   a ")
   (printout t ?new-animal " from a " ?value "? ")
   (bind ?question (readline))
   (printout t "Now I can guess " ?new-animal crlf)
   ; Create the new learned nodes
   (bind ?newnode1 (gensym*))
   (bind ?newnode2 (gensym*))
   (modify ?data (type decision)
                 (question ?question)
                 (yes-node ?newnode1)
                 (no-node ?newnode2))
   (assert (node (name ?newnode1)
                 (type answer)
                 (answer ?new-animal)))
   (assert (node (name ?newnode1)
                 (type answer)
                 (answer ?value)))
   ; Determine if the player wants to try again
   (assert (ask-try-again)))
```

Replace-answer 结点规则要求对这个动物的正确进行判断，并询问一个问题，以便将该动物区别于已由判定树判断为正确答案的那种动物。旧的答案结点将被一个判定结点代替，同时产生两个答案结点作为刚学习的问题的回答。gensym* 函数（每次调用时产生一个唯一的标识）可用来为两个新生成的答案结点取名。然后，ask-try-again 事实被声明，以判定该程序是否须再运行一次。

判定树程序的逐步跟踪

判定树程序的特性可通过执行过程而了解。假设判定树规则已被调入，包含初始化判定树事实的文件 animal.dat 也已创立，下面的对话显示了执行 reset 命令后的系统状态：

```
CLIPS> (watch facts)↵
CLIPS> (watch rules)↵
CLIPS> (reset)↵
==> f-0     (initial-fact)
CLIPS> (agenda)↵
0      initialize: f-0,
For a total of 1 activation.
CLIPS>
```

initialize 规则因 root node 事实的缺失而被激活。判定树中允许执行初始化规则。注意，下列各步骤中的某些输出缩行是为了提高可读性。

```
CLIPS> (run 1)↵
FIRE    1 initialize: f-0,
==> f-1     (node (name root) (type decision)
          (question "Is the animal warm-blooded?")
             (yes-node node1) (no-node node2)
             (answer nil))
==> f-2     (node (name node1) (type decision)
             (question "Does the animal purr?")
             (yes-node node3) (no-node node4)
             (answer nil))
==> f-3     (node (name node2) (type answer)
             (question nil) (yes-node nil)
             (no-node nil) (answer snake))
==> f-4     (node (name node3) (type answer)
             (question nil) (yes-node nil)
             (no-node nil) (answer cat))
==> f-5     (node (name node4) (type answer)
             (question nil) (yes-node nil)
             (no-node nil) (answer dog))
==> f-6     (current-node root)
CLIPS> (agenda)↵
0      ask-decision-node-question: f-6,f-1,
For a total of 1 activation.
CLIPS>
```

initialize 规则使用 load-facts 函数将事实调入到判定树中。current-node 事实被设置为 root node 事实。因为根结点是判定结点，所以，ask-decision-node-question 规则被激活。该规则和相关的 proceed-to-yes-branch 规则执行会产生下列对话：

```
CLIPS> (run 2)↵
FIRE    1 ask-decision-node-question: f-6,f-1,
Is the animal warm-blooded? (yes or no) yes↵
==> f-7     (answer yes)
FIRE    2 proceed-to-yes-branch: f-6,f-1,f-7
<== f-6     (current-node root)
<== f-7     (answer yes)
==> f-8     (current-node node1)
CLIPS> (agenda)↵
0      ask-decision-node-question: f-8,f-2,
For a total of 1 activation.
CLIPS>
```

与根判定结点相关联的问题是“该动物是暖血动物吗?”当问题的回答是肯定时，则 proceed-to-yes-branch 规则使判定结点的左结点（即 node1）成为当前结点。由于 node1 也是判定结点，所以 ask-decision-node-question 规则被再次激活。其后的两条规则再次执行将产生下面的对话：

```
CLIPS> (run 2)↵
FIRE    1 ask-decision-node-question: f-8,f-2,
Does the animal purr? (yes or no) no↵
==> f-9     (answer no)
FIRE    2 proceed-to-no-branch: f-8,f-2,f-9
<== f-8     (current-node node1)
<== f-9     (answer no)
==> f-10    (current-node node4)
CLIPS> (agenda)↵
0      ask-if-answer-node-is-correct: f-10,f-5,
For a total of 1 activation.
CLIPS>
```

与node1判定结点相关联的问题是，“该动物会喵喵吗?”因为问题的回答是否定的，所以，proceed-to-no-branch规则将使判定结点的右结点（即node 4）成为当前结点。由于node 4结点是一个答案结点，所以，将激活ask-if-answer-node-is-correct规则。该规则和下一条规则执行将产生以下的对话：

```
CLIPS> (run 2)↵
FIRE    1 ask-if-answer-node-is-correct: f-10,f-5,
I guess it is a dog
Am I correct? (yes or no) no↵
==> f-11    (answer no)
FIRE    2 answer-node-guess-is-incorrect: f-10,f-5,
                                          f-11
==> f-12    (replace-answer-node node4)
<== f-10    (current-node node4)
<== f-11    (answer no)
CLIPS> (agenda)↵
0      replace-answer-node: f-12,f-5
For a total of 1 activation.
CLIPS>
```

与这个答案结点相关联的猜测是狗。由于该猜测错误，故replace-answer-node规则被激活以决定正确答案。允许执行这条规则将产生以下的对话：

```
CLIPS> (run 1)↵
FIRE    1 replace-answer-node: f-12,f-5
<== f-12    (replace-answer-node node4)
What is the animal? bird↵
What question when answered yes will distinguish
   a bird from a dog? Does the animal fly?↵
Now I can guess bird
<== f-5     (node (name node4) (type answer)
                  (question nil)
                  (yes-node nil) (no-node nil)
                  (answer dog))
==> f-13    (node (name node4) (type decision)
                  (question
                  "Does the animal fly?")
                  (yes-node gen1) (no-node gen2)
                  (answer dog))
==> f-14    (node (name gen1) (type answer)
                  (question nil)
                  (yes-node nil) (no-node nil)
                  (answer bird))
==> f-15    (node (name gen2) (type answer)
                  (question nil)
                  (yes-node nil) (no-node nil)
                  (answer dog))
==> f-16    (ask-try-again)
CLIPS> (agenda)↵
0      ask-try-again: f-16,
For a total of 1 activation.
CLIPS>
```

首先，控制事实（replace-answer-node node4）被撤销，接着，通过对能产生正确推理的问题的回答作出正确推理。不正确的答案结点被修改为问题结点，然后，从新的问题结点产生两个答案结点。最后，声明ask-try-again（询问是否重试）事实来决定是否需要做另一次识别。允许ask-try-again规则、然后又允许no-more规则执行将产生以下的对话：

```
CLIPS> (run 2)↵
FIRE    1 ask-try-again: f-16,
Try again? (yes or no) no↵
==> f-17    (answer no)
FIRE    2 no-more: f-16,f-17
<== f-16    (ask-try-again)
<== f-17    (answer no)
CLIPS> (agenda)↵
CLIPS>
```

通过 ask-try-again 规则询问用户是否需要再做一次鉴别。由于回答为否定（no），所以 no-more 规则将把判定树保存回 animal.dat 文件中。完成此对话后，animal.dat 文件的最终形式如下：

```
(node (name root) (type decision)
      (question "Is the animal warm-blooded?")
      (yes-node node1) (no-node node2)
      (answer nil))
(node (name node1) (type decision)
      (question "Does the animal purr?")
      (yes-node node3) (no-node node4)
      (answer nil))
(node (name node2) (type answer)
      (question nil)
      (yes-node nil) (no-node nil) (answer snake))
(node (name node3) (type answer)
      (question nil)
      (yes-node nil) (no-node nil) (answer cat))
(node (name node4) (type decision)
      (question "Does the animal fly?")
      (yes-node gen1) (no-node gen2) (answer dog))
(node (name gen1) (type answer)
      (question nil)
      (yes-node nil) (no-node nil) (answer bird))
(node (name gen2) (type answer)
      (question nil)
      (yes-node nil) (no-node nil) (answer dog))
```

node 4 结点已被指向两个新的答案结点的一个判定结点所替代。而且，在保存这些事实时，自动赋予某些自定义模板槽的默认值 nil 现在也得到明确说明。

12.4 反向链

CLIPS 并不直接将反向链作为其推理机的一部分来实现。但是，反向链可用 CLIPS 正向链规则来模拟。本节将论述如何在 CLIPS 中建立简单的反向链系统。应该注意的是，CLIPS 是作为正向链接语言设计的；如果一个反向链方法更适合于解决某个问题，那么，就应该采用能在其推理机中直接实现反向链的语言，如 PROLOG。

建立的 CLIPS 反向链系统将有以下性能和限制：

- 事实可表示为属性－值对。
- 反向链以一个单一的初始目标属性的声明而开始。
- 只有属性值与一个特定值相等才被作为一个规则前件的测试条件。
- 规则前件的行为只能是指定单一属性的值。
- 如果使用规则不能决定一个目标属性的值，那么，反向链系统将要求提供该属性的值。不能赋给属性一个未知值。
- 一个属性只能有唯一的值。系统不支持关于不同规则中的不同属性的假设推理。
- 不能表示不确定性。

一个反向链的算法

在写一个反向链推理机并尝试用 CLIPS 的基于规则的方法之前，我们将思考一个程序算法。以下伪码程序用来决定一个目标属性值，它使用了具有前面所讨论性能和限制的反向链方法：

```
procedure Solve_Goal(goal)
   goal: the current goal to be solved
   if value of the goal attribute is known
      Return the value of the goal attribute.
   end if

   for each rule whose consequent is the goal
   attribute do
```

```
      call Attempt_Rule with the rule
      if Attempt_Rule succeeds then
         Assign the goal attribute the value
         indicated by the consequent of the rule.
         Return the value of the goal attribute.
      end if
   end do

   Ask the user for the value of the goal
   attribute.
   Set the goal attribute to the value supplied
   by the user.
   Return the value of the goal attribute.

end procedure
```

目标属性是作为参数传给 Solve _ Goal 程序的。这个程序将决定该目标属性的值，并返回该值。Solve _ Goal 程序首先检测这个目标属性值是否已知。该值可能已被某条规则的后件所指定，或者，已由反向链系统的用户提供。若该值确实已知，则返回该值。

如果该属性值未知，则 Solve _ Goal 程序将试着找出一条把赋值该属性值作为其后件的规则来决定这个属性值。Solve _ Goal 程序将逐一尝试每一个把赋值给目标属性作为其后件的规则，直至其中之一成功为止。每一条有所需属性的规则都要给 Attempt _ Rule 程序（将很快详细讨论）去尝试。如果被尝试的规则的前件满足，那么就成功了；否则，该规则就失败。如果规则成功，则这条规则后件中的属性值就被赋值给目标属性，并通过 Solve _ Goal 程序返回该值。若该规则不成功，那么，将尝试下一个把赋值给目标属性作为其后件的规则。

如果所有规则都不成功，就必须询问用户去决定此目标属性的值。由用户提供的值将被 Solve _ Goal 程序返回。

Attempt _ Rule 程序用于判断是否满足规则的前件。若此前件被满足，则该规则的后件可用来赋值目标属性值。该程序的伪代码如下：

```
procedure Attempt_Rule(rule)
   rule: rule to be attempted to solve goal
   for each condition in the antecedent
   of the rule do
      call Solve_Goal with condition attribute
      if the value returned by solve_goal is not
      equal to the value required by the condition
      then
         Return unsuccessful.
      end if
   end for

   Return successful

end procedure
```

Attempt _ Rule 程序从规则的第一条件处开始，并在尝试该规则随后的条件之前先检验该条件。为了确定是否满足一个条件，Attempt _ Rule 程序必须知道在条件中被测试的属性值。为了决定该值，则递归调用 Solve _ Goal 程序。如果由 Solve _ Goal 程序返回的值不等于条件所需的值，则 Attempt _ Rule 程序将中止，并返回值：unsuccessful（不成功）（记住，在条件中只测试相等）。否则，将继续检测此规则的下一个条件。若规则的所有条件都满足，则 Attempt-Rule 程序将返回值：successful（成功）。

CLIPS 反向链规则的表达

又一次遇到此类问题，解决它的第一步仍然是确定应该如何表达知识。由于 CLIPS 不能自动执行反向链，因此，将反向链规则表达为事实是有用的，这样，前件和后件就可以由作为反向链推理机的规则检验。表达反向链规则的自定义模板如下所示。它将存储在自定义模块 BC 中（当学完反向链推理机所需的所有自定义模板后，将在本小节末尾定义此模块 BC）。

```
(deftemplate BC::rule
   (multislot if)
   (multislot then))
```

if 和 then 槽分别存储每条规则的前件和后件。每个前件或者包含单个形如：

```
<attribute> is <value>
```

的属性－值对（attribute-value pair），或者包含一系列由符号 and 连接的这种属性－值对。而每条规则的后件只允许包含单个属性－值对。

思考图 12.2 中的判定树，作为使用这种格式来表达规则的例子。可以使用前面描述的 AV 对很容易地将这种树转换为规则。转换后的规则的伪码是：

```
IF main-course is red-meat
THEN best-color is red

IF main-course is poultry and
   meal-is-turkey is yes
THEN best-color is red

IF main-course is poultry and
   meal-is-turkey is no
THEN best-color is white

IF main-course is fish
THEN best-color is white
```

规则所用的属性是 main-course、meal-is-turkey 和 best-color。main-course 属性对应于由判定树问题“主菜是什么?”决定的回答。meal-is-turkey 属性对应于由问题“主菜是火鸡吗?”决定的回答。注意，确定最好的颜色未知的判定树的分枝没有规则表达，因为我们的反向链系统的限制之一是，未知值不能被表达。如果主菜不能得到任何答案，像这种情况，将会询问用户 best-color 属性的值。

下列自定义事实示出了如何使用反向链规则的格式表达酒的规则。由于这些 rule 事实不是反向链推理机的固有部分，因此，它们被置于 MAIN 模块中（回忆 MAIN 模块从所有其他模块输入，因此，rule 自定义模板对于此 MAIN 模块是可见的）。

```
(deffacts MAIN::wine-rules
   (rule (if main-course is red-meat)
         (then best-color is red))

   (rule (if main-course is fish)
         (then best-color is white))

   (rule (if main-course is poultry and
             meal-is-turkey is yes)
         (then best-color is red))

   (rule (if main-course is poultry and
             meal-is-turkey is no)
         (then best-color is white)))
```

当处理这些反向链规则时，这种表达提供了很高的灵活性。例如，如果确定属性 main-course 有 poultry 值，则事实：

```
(rule (if main-course is red-meat)
      (then best-color is red))
```

和

```
(rule (if main-course is fish)
      (then best-color is white))
```

可以从事实表中删除，表示这些规则不适用，并且事实：

```
(rule (if main-course is poultry and
          meal-is-turkey is yes)
      (then best-color is red))
```

和

```
(rule (if main-course is poultry and
          meal-is-turkey is no)
      (then best-color is white))
```

可分别修改为下面事实：

```
(rule (if meal-is-turkey is yes)
      (then best-color is red))
```

和

```
(rule (if meal-is-turkey is no)
      (then best-color is white))
```

这表明，这两个规则的第一条件已经被满足。

随着反向链的进行，将产生一些子目标以确定属性值。这需要事实来表达关于目标属性的信息。有序事实将用来表达目标属性，其格式是：

```
(deftemplate BC::goal
   (slot attribute))
```

一开始，目标值是 best-color。这可以用自定义事实表达为：

```
(deffacts MAIN::initial-goal
   (goal (attribute best-color)))
```

当属性值确定后，就需要存储它们，这可用下面的自定义模板完成：

```
(deftemplate BC::attribute
   (slot name)
   (slot value))
```

至此，所有自定义模板都已提出，故可以开始定义 BC 模块了。记住，当你调入一个结构文件时，包含其他结构的自定义模块必须在定义此模块中的结构之前被定义。

```
(defmodule BC
   (export deftemplate rule goal attribute))
```

CLIPS 反向链推理机

反向链推理机可以用两组规则实现。第一组将产生属性的目标，并在这些值不能被规则确定时要求用户提供属性值。第二组规则将执行更新操作。更新操作包括：条件已满足时修改规则并删除目标。第一组规则如下：

```
(defrule BC::attempt-rule
   (goal (attribute ?g-name))
   (rule (if ?a-name $?)
         (then ?g-name $?))
   (not (attribute (name ?a-name)))
   (not (goal (attribute ?a-name)))
   =>
   (assert (goal (attribute ?a-name))))

(defrule BC::ask-attribute-value
   ?goal <- (goal (attribute ?g-name))
   (not (attribute (name ?g-name)))
   (not (rule (then ?g-name $?)))
   =>
   (retract ?goal)
   (printout t "What is the value of "
               ?g-name "? ")
   (assert (attribute (name ?g-name)
                      (value (read)))))
```

attempt-rule 规则查找那些能为目标属性提供属性值的规则。第一模式匹配 goal（目标）事实。第二模式查找那些将值指定给目标属性的所有规则。第三模式检查以确认此目标属性的值是否还没有被

确定。第四模式确认已没有需确定属性值的目标了。对于每一条被找到的规则，attempt-rule 规则的 RHS 将声明一个目标以决定由规则的第一条件测试过的属性值。

ask-attribute-value 规则非常类似于 attempt-rule 规则。前面两种模式是相同的。第三种模式检查是否没有剩余的规则可用来确定目标属性值。在本例中，要求用户提供属性的值。代表该属性值的事实被声明，且此属性的 goal 事实被撤销。

下面 4 条规则用于更新反向链规则和表示为事实的目标。分配给这些规则的优先级为 100，以便在任何企图想产生新的目标或要求用户提供属性值之前，都可以允许更新。

```
(defrule BC::goal-satisfied
   (declare (salience 100))
   ?goal <- (goal (attribute ?g-name))
   (attribute (name ?g-name))
   =>
   (retract ?goal))

(defrule BC::rule-satisfied
   (declare (salience 100))
   (goal (attribute ?g-name))
   (attribute (name ?a-name)
              (value ?a-value))
   ?rule <- (rule (if ?a-name is ?a-value)
                  (then ?g-name is ?g-value))
   =>
   (retract ?rule)
   (assert (attribute (name ?g-name)
                      (value ?g-value))))

(defrule BC::remove-rule-no-match
   (declare (salience 100))
   (goal (attribute ?g-name))
   (attribute (name ?a-name) (value ?a-value))
   ?rule <- (rule (if ?a-name is ~?a-value)
                  (then ?g-name is ?g-value))
   =>
   (retract ?rule))

(defrule BC::modify-rule-match
   (declare (salience 100))
   (goal (attribute ?g-name))
   (attribute (name ?a-name) (value ?a-value))
   ?rule <- (rule (if ?a-name is ?a-value and
                      $?rest-if)
                  (then ?g-name is ?g-value))
=>
(retract ?rule)
(modify ?rule (if $?rest-if)))
```

goal-satisfied 规则删除任何属性值已确定的目标。

rule-satisfied 规则查找任何有单个剩余条件的规则。如果某个属性存在且满足此剩余条件、有一个目标确定该属性的值，那么，该规则后件的属性值就添加到事实表上。

remove-rule-no-match 规则查找满足这些条件的规则：其前件能为目标属性提供属性值且包含一个或多个条件，其中的第一个条件与指定给某个属性的值相冲突。如果是这样，则该规则因不再适用而从事实表中删除。

modify-rule-match 规则查找满足这些条件的规则：其前件能为目标属性提供属性值且包含一个或多个条件，其中的第一个条件由指定给某个属性的值所满足。如果找到这样一个规则，则第一个条件从该规则中删除从而留下剩余的必须被测试的条件。

至此，所有的反向链规则都提供了，需要启动反向链过程的所有操作是聚焦于 BC 模块。这可以通过添加下列规则到 MAIN 模块上来完成：

```
(defrule MAIN::start-BC
   =>
   (focus BC))
```

反向链系统的逐步跟踪

使用规则实现的 CLIPS 反向链推理机的性能可以通过监视其执行情况来了解。假定 wine - rules 和 initial - goal 自定义事实与反向链推理机规则一起调入，那么，使用 reset 命令后系统的状态如下（再次声明，某些输出的缩行是为了提高可读性）：

```
CLIPS> (unwatch all)↵
CLIPS> (reset)↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-1     (goal (attribute best-color))
f-2     (rule (if main-course is red-meat)
              (then best-color is red))
f-3     (rule (if main-course is fish)
              (then best-color is white))
f-4     (rule (if main-course is poultry and
                  meal-is-turkey is yes)
              (then best-color is red))
f-5     (rule (if main-course is poultry and
                  meal-is-turkey is no)
              (then best-color is white))
For a total of 6 facts.
CLIPS> (agenda)↵
0      start-BC: f-0
For a total of 1 activation.
CLIPS>
```

Start-BC 规则只聚焦于 BC 模块。此规则一旦触发，BC 模块将成为当前焦点。

```
CLIPS> (run 1)↵
CLIPS> (agenda)↵
0      attempt-rule: f-1,f-5,,
0      attempt-rule: f-1,f-4,,
0      attempt-rule: f-1,f-3,,
0      attempt-rule: f-1,f-2,,
For a total of 4 activations.
CLIPS>
```

注意，此议程包含 attempt-rule 规则的 4 个激活。启动目标就是确定由 f-1 事实规定的 best-color 属性的值。由于每个规则事实 f-2、f-3、f-4 和 f-5 的后件都将一个值分配给了 best-color 属性，因此，这些规则每一条都必须尝试看是否满足 best-color 属性目标。

执行的下一步是触发 attempt-rule 规则的第一个激活。在触发此规则之前，要激活 watch 规则和事实：

```
CLIPS> (watch rules)↵
CLIPS> (watch facts)↵
CLIPS> (run 1)↵
FIRE    1 attempt-rule: f-1,f-5,,
==> f-6     (goal (attribute main-course))
CLIPS> (agenda)↵
0      ask-attribute-value: f-6,,
For a total of 1 activation.
CLIPS>
```

attempt-rule 规则由事实 f-5 触发，它代表的反向链规则如下：

```
IF main-course is poultry and
   meal-is-turkey is no
THEN best-color is white
```

在此规则可用来指定 best-color 属性的值之前，必须满足其前件中的 CE。第一条件需要属性 main-course 的值。既然此属性未知，为此要建立一个目标，此目标由事实 f-6 代表。由于没有规则指定 main-course 属性的值，因此激活 ask-attribute-value 规则的值。

随着执行的继续，ask-attribute-value 规则触发以确定 main-course 属性的值。

```
CLIPS> (run 1)↵
FIRE    1 ask-attribute-value: f-6,,
<== f-6     (goal (attribute main-course))
What is the value of main-course? poultry↵
==> f-7     (attribute (name main-course)
                       (value poultry))
CLIPS> (agenda)↵
100     remove-rule-no-match: f-1,f-7,f-3
100     remove-rule-no-match: f-1,f-7,f-2
100     modify-rule-match: f-1,f-7,f-5
100     modify-rule-match: f-1,f-7,f-4
For a total of 4 activations.
CLIPS>
```

由于要求用户提供 main-course 属性的值，因此，删除对应于属性 f-6 的目标。由用户提供的值作为 attribute 事实 f-7 被声明。对此事实的声明使得 4 个新的激活置于议程中。由事实 f-4 和 f-5 代表的规则都把要求 main-course 属性是 poultry 作为其第一条件。因此，两个事实引起激活 modify-rule-match 规则。由事实 f-2 和 f-3 代表的规则都把要求 main-course 属性不是 poultry 而是其他东西作为第一条件。因此，这些事实都不再适用了。激活规则 remove-rule-no-match 使得上述两个事实被删除。

允许两个 remove-rule-no-match 激活触发将产生下列输出：

```
CLIPS> (run 2)↵
FIRE    1 remove-rule-no-match: f-1,f-7,f-3
<== f-3     (rule (if main-course is fish)
                  (then best-color is white))
FIRE    2 remove-rule-no-match: f-1,f-7,f-2
<== f-2     (rule (if main-course is red-meat)
                  (then best-color is red))
CLIPS> (agenda)↵
100     modify-rule-match: f-1,f-7,f-5
100     modify-rule-match: f-1,f-7,f-4
For a total of 2 activations.
CLIPS>
```

事实 f-2 和 f-3 从事实表上删除，表示由这些事实代表的规则不能再用了。当这些事实被删除时，attempt-rule 激活也从议程中被删除掉。

转向执行两个 modify-rule-match 激活会产生下列输出：

```
CLIPS> (run 2)↵
FIRE    1 modify-rule-match: f-1,f-7,f-5
<== f-5     (rule (if main-course is poultry and
                      meal-is-turkey is no)
                  (then best-color is white))
==> f-8     (rule (if meal-is-turkey is no)
                  (then best-color is white))
FIRE    2 modify-rule-match: f-7,f-4
<== f-4     (rule (if main-course is poultry and
                      meal-is-turkey is yes)
                  (then best-color is red))
==> f-9     (rule (if meal-is-turkey is yes)
                  (then best-color is red))
CLIPS> (agenda)↵
0       attempt-rule: f-1,f-9,,
0       attempt-rule: f-1,f-8,,
For a total of 2 activations.
CLIPS>
```

Modify-rule-match-rule 规则的第一次触发是基于事实 f-5，它代表下列反向链规则：

```
IF main-course is poultry and
   meal-is-turkey is no
THEN best-color is white
```

modify-match-rule 规则的操作会将此反向链规则修改为：

```
IF meal-is-turkey is no
THEN best-color is white
```

代表已修改的规则的新事实是 f-8。此新事实代表第一条件满足后剩下的初始规则的条件，并引起

此反向链规则的 attempt-rule 规则再一次激活。此新的激活将声明一个新的目标来决定 meal-is-turkey 的值，以使得该规则的后件可用来指定 best-color 属性的值。

第二次触发 modify-rule-match 规则与第一次触发类似。代表该规则的事实：

```
IF main-course is poultry and
   meal-is-turkey is red
THEN best-color is red
```

被修改为下列规则：

```
IF meal-is-turkey is yes
THEN best-color is red
```

它是由事实 f-9 代表的。与此类似，当代表该规则的事实撤销后，这个新的事实会激活 attempt-rule 规则，以取代失去的激活。

允许第一个 attempt-rule 激活触发将产生下列输出：

```
CLIPS> (run 1)↵
FIRE    1 attempt-rule: f-1,f-9,,
==> f-10    (goal (attribute meal-is-turkey))
CLIPS> (agenda)↵
0      ask-attribute-value: f-10,,
For a total of 1 activation.
CLIPS>
```

事实 f-10 被声明，它代表一个目标以决定 meal-is-turkey 属性的值。既然没有规则指定该属性的值，故激活 ask-attribute-value 规则以决定该值。

允许 ask-attribute-value 规则触发会产生下列输出：

```
CLIPS> (run 1)↵
FIRE    1 ask-attribute-value: f-10,,
<== f-10    (goal (attribute meal-is-turkey))
What is the value of meal-is-turkey? yes↵
==> f-11    (attribute (name meal-is-turkey)
                       (value yes))
CLIPS> (agenda)↵
100    rule-satisfied: f-1,f-11,f-9
100    remove-rule-no-match: f-1,f-11,f-8
For a total of 2 activations.
CLIPS>
```

由于触发了此规则，因此，代表属性 meal-is-turkey 之值的 attribute 事实被声明。此外，决定此属性值的 goal 事实被删除。新的 attribute 事实产生两个激活。第一个激活是针对于 remove-rule-no-match 规则的。因为事实 f-8 的第一条件与此新属性的值不一致，且该事实代表的规则不再可用，因此，该事实有必要被删除。第二个激活是针对于 rule-satisfied 规则的。由于事实 f-8 的剩余条件被新的 attribute 事实满足，因此，该事实的后件可用。

剩余的规则触发后完成反向链过程。

```
CLIPS> (run)↵
FIRE    1 rule-satisfied: f-1,f-11,f-9
<== f-9     (rule (if meal-is-turkey is yes)
                  (then best-color is red))
==> f-12    (attribute (name best-color)
                       (value red))
FIRE    2 goal-satisfied: f-1,f-12
<== f-1     (goal (attribute best-color))
CLIPS> (agenda)↵
CLIPS> (facts)↵
f-0     (initial-fact)
f-7     (attribute (name main-course)
                   (value poultry))
f-8     (rule (if meal-is-turkey is no)
              (then best-color is white))
f-11    (attribute (name meal-is-turkey)
                   (value yes))
f-12    (attribute (name best-color) (value red))
For a total of 5 facts.
CLIPS>
```

作为由事实 f-9 代表的规则的部分后件，触发 rule-satisfied 规则以指定 best-color 属性的值。由此规则声明的 attribute 事实满足在事实表中剩下的目标事实。规则 goal-satisfied 被激活，然后触发以删除剩下的 goal 事实。

agenda 命令显示，没有剩下要触发的规则。facts 命令显示已被赋值的属性。f-12 事实表明，初始目标属性 best-color 被赋的值是 red。

12.5 监视问题

本节提出 CLIPS 程序的逐步开发以作为简单问题的解决方法。开发步骤包括对问题的初始描述，对问题性质所作的假定以及表示问题知识的初始定义，最后是逐步建立规则解决问题。

问题描述

本节待解决的问题是一个简单的监视问题。监视问题由于其数据驱动本性，故适于使用基于规则的正向链语言。一般地，程序每循环一周，就读取一组输入或传感器的值。推理一直进行，直到得到所有可能的能从输入数据推出的结论为止。这与数据驱动方法，即推理过程是从数据到该数据推出的结论是一致的。

对于本例，待实现的监视类型在特点上是一般的。假设某处理厂有几台设备要监视。一些设备的工作要依赖于其他设备。每台设备有一个以上的传感器与之相连，以提供数字读数指出设备的工作状态。每个传感器有低警戒线（LGL）、低危险线（LRL）、高警戒线（HGL）和高危险线（HRL）量值。在低和高警戒线之间的读数认为是正常的。在高警戒线之上但在高危险线之下，或者，在低警戒线之下但在低危险线之上认为是可接受的，尽管这表明设备将很快会不正常。在高危险先之上、或低危险线之下的读数认为设备不正常，应该关机。在警戒区的任何设备都应该发出警告信息。此外，在警戒区工作过久的任何设备也应该关机。对于给定的传感器值，表 12.1 总结了需要采取的动作。

表 12.1 对应于传感器值所采取的动作

传感器值	动作
小于或等于低危险线	关机
大于低危险线、小于或等于低警戒线	发出警告或关机
大于低警戒线、小于高警戒线	无
大于或等于高警戒线、小于高危险线	发出警告或关机
大于或等于高危险线	关机

监视程序应该能够读取传感器的数据、计算传感器的读数并根据对传感器的计算值和趋势发出警告或关机。监视程序的输出样板列示如下：

```
Cycle 20 - Sensor 4 in high guard line
Cycle 25 - Sensor 4 in high red line
   Shutting down device 4
Cycle 32 - Sensor 3 in low guard line
Cycle 38 - Sensor 1 in high guard line for 6 cycles
   Shutting down device 1
```

对于本例，图 12.5 示出了待监视的设备和传感器之间的连接情况，表 12.2 列出了每个传感器的属性。

表 12.2 传感器属性

传感器	低危险线	低警戒线	高警戒线	高危险线
S1	60	70	120	130
S2	20	40	160	180

（续）

传感器	低危险线	低警戒线	高警戒线	高危险线
S3	60	70	120	130
S4	60	70	120	130
S5	65	70	120	125
S6	110	115	125	130

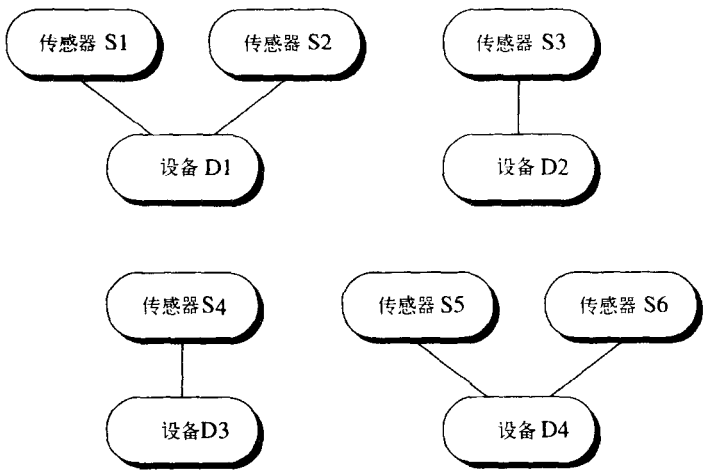


图 12.5 监视系统中的设备和传感器

这个问题的部分解决方法包括利用问题的一般描述以确定在实现此解决方法之前必须解决的特定细节。一般地，这个过程包括不断咨询对此问题领域有广博知识的专家，但也不必详细说明打算由专家系统捕捉的这个任务。要开发出该问题的原型以指出在问题说明中丢失的细节。这些细节是通过咨询专家确定的。而且要开发另一个原型，以更进一步地展示问题说明中丢失的细节。最后，通过这种反复的编程开发方法完全捕捉问题的详细说明。

此问题的许多规范细节仍悬而未决：怎样表示传感器和设备的信息？如何设计通用或专用的事实或规则？如何访问传感器数据？在某个传感器的相关设备关机之前，该传感器应在警戒区多长时间？当监视器检测到设备不正常时，要采取什么措施？

开始必需的细节

建立专家系统遇到的一个主要问题是，问题往往说明得不明确。其思想是模仿专家，但除了专家之外没有哪个人有足够的知识来说明其细节。一般地，系统的期望性能是已知的，但如何达到这种性能的方式却是未知的。专家在表达准确的要采取的步骤以得出解决方法上可能有困难。由于本质上支持迭代开发技术，因此能相对容易地建立解决定义不好问题的专家系统。然而，这并不是说，专家系统能解决以前一直未得到解决的问题，也不是说能解决没得到理解的问题。

对于本例的监视问题，在建立专家系统之前，必须说明几个问题。首先，应该说明期望的专家系统性能，包括初始信息和系统将要产生的信息。这并非意味着此说明不能改变。程序的开发可能显示出，问题的范围应该缩小或扩大，而且这会影响到初始输入和最终输出。必须作一些假定，还必须确定有关专家系统应该怎样执行其任务的初始细节。而且，这种判定问题并不是永久不变的。专家通常可以展示他们是怎样解决问题的，但不容易使他们所采用的规则公式化。在描述他们所采用的规则时，专家们可能漏掉了对他们来说是显而易见的细节，或者，忘记了例外情况。

监视问题要求作出许多初始的决定。其一是实现过程。解决方法是要专门针对于此问题说明的精

确细节，还是要有足够的普遍性以便于升级或修改？对于每一种受监视的设备，可以写出特定的规则，也可以写出监视所有设备的通用规则。对于此问题，写出通用规则可能更合适，因为，各个模型化了的设备和传感器没有各自独特的特点。这种通用性应该允许易于增添更多的设备和传感器。

有关系统的控制流程细节也易于忽略。对于此问题，将使用简单的监视循环。每个监视循环包含3个阶段。第一阶段是读取传感器的值，第二阶段是分析这些值，第三阶段是采取适当的措施。

还必须假定要怎样访问传感器的数据。直接读取传感器的值吗？需要处理传感器数据的仿真吗？需要时总是可以访问传感器数据吗？传感器数据可靠还是易受误差的影响？在正常原型情况下，应该约见专家确定这些信息，但对于此问题的目的，将作些假设填写有关细节。

所有这些问题指出了在定义不好的问题中可能会遇到的纰漏。在作一系列假定时，应该注意与此问题说明有关的疑问和可能的不一致性。在迭代开发程序时，这些假定列表应该是讨论的焦点，必须有专家确保问题说明与专家解决此问题的观点相符。下面是本监视问题开始的假定列表：

- 传感器数据总是可靠的，需要时总是可以读取其数据
- 传感器值可直接读取，也支持模拟的传感器值
- 已经关机的机器，其相应的传感器值将不受监视
- 假定可以实施由专家系统规定的动作（假设我们有警报操作员且设备控制可由程序直接处理）
- 此问题将分成3个阶段：读取传感器值、分析传感器值、采取适当的行动，如关闭一台设备

作为对问题说明细节的补充，问题实现的细节也必须确定。包括如何表示有用的信息、控制流以及测试专家系统。

知识定义

我们再次以确定如何表达知识来作为解决问题的开始。此问题的一个好的开始是对图12.5和表12.2中的知识编码。下面的自定义模板将用来描述这些设备：

```
(defmodule MAIN (export ?ALL))

(deftemplate MAIN::device
   (slot name (type SYMBOL))
   (slot status (allowed-values on off)))
```

其中，name槽是设备名，status槽表示该设备的开关状态。利用图12.5，并假定所有设备初始状态是开（on），则可以用下列自定义事实描述设备的初始状态：

```
(deffacts MAIN::device-information
(device (name D1) (status on))
(device (name D2) (status on))
(device (name D3) (status on))
(device (name D4) (status on)))
```

图12.5也表示了哪些传感器与哪些设备有关。下面的自定义模板将用来表示这种关系：

```
(deftemplate MAIN::sensor
(slot name (type SYMBOL))
(slot device (type SYMBOL))
(slot raw-value (type SYMBOL NUMBER)
                (allowed-symbols none)
                (default none))
(slot state (allowed-values low-red-line
                            low-guard-line
                            normal
                            high-red-line
                            high-guard-line)
            (default normal))
(slot low-red-line (type NUMBER))
(slot low-guard-line (type NUMBER))
(slot high-guard-line (type NUMBER))
(slot high-red-line (type NUMBER)))
```

其中，name 槽是传感器名，device 槽是与传感器相连的设备名。raw-value 槽是处理之前直接从传感器读得的数值。state 槽指示传感器的当前状态（如，正常、低警戒线、高危险线等）。用于包含表 12.2 中描述信息的槽有：low-red-line、low-guard-line、expected-average-value、high-guard-line 和 high-red-line。下列自定义事实可用来描述图 12.5 中的传感器：

```
(deffacts MAIN::sensor-information
  (sensor (name S1) (device D1)
          (low-red-line 60) (low-guard-line 70)
          (high-guard-line 120)
          (high-red-line 130))
  (sensor (name S2) (device D1)
          (low-red-line 20) (low-guard-line 40)
          (high-guard-line 160)
          (high-red-line 180))
  (sensor (name S3) (device D2)
          (low-red-line 60) (low-guard-line 70)
          (high-guard-line 120)
          (high-red-line 130))
  (sensor (name S4) (device D3)
          (low-red-line 60) (low-guard-line 70)
          (high-guard-line 120)
          (high-red-line 130))
  (sensor (name S5) (device D4)
          (low-red-line 65) (low-guard-line 70)
          (high-guard-line 120)
          (high-red-line 125))
  (sensor (name S6) (device D4)
          (low-red-line 110) (low-guard-line 115)
          (high-guard-line 125)
          (high-red-line 130)))
```

由于监视系统是循环的，因此，需要一个事实来代表当前循环。第一个循环可以在 1 的时候开始，接着，每一个新的循环就增加 1。此信息的有序事实格式如下：

```
(cycle <number>)
```

其中，<number>是当前循环的取值。此外，由于传感器的值可能从多个来源得到（如一个来自于仿真，一个来自于实际值），所以，设置一个事实指示传感器数据的来源是有用的。此事实可用下列格式表示：

```
(data-source <source>)
```

其中，<source>是表示读取数据的来源的实例名称。<source>实例是将被简短描述的 DATA-SOURCE 类中的一员。可能的来源包括：传感器、仿真器、文本文件、一组事实或用户输入。

包含此初始信息的自定义事实是：

```
(deffacts MAIN::cycle-start
(data-source [user])
(cycle 0))
```

注意，用户可为传感器提供数据。对于本例，从键盘输入数据比从文件读取数据更方便。

执行的控制

问题假定语句表明，监视过程有 3 个特定阶段。第一个阶段是读取用户提供的、仿真的或传感器实际的数据。第二个阶段是将警戒线和危险线条件与传感器联系起来，并确定发展趋势。一旦建立了趋势，监视系统则发出适当的警告，关掉运行不良的设备并重新启动可以回到界线上来的设备。在采取适当的措施后，读取新的传感器值将开始下一个新循环。

利用类似于第 9 章描述的技术，可以处理此监视专家系统的控制问题。建立 3 个独立的模块：INPUT、TRENDS 和 WARNINGS。每一次循环，cycle 事实的值被更新，并以适当的顺序聚焦于正在进行的模块。下列规则将执行这些操作：

```
(defrule MAIN::Begin-Next-Cycle
   ?f <- (cycle ?current-cycle)
   (exists (device (status on)))
   =>
   (retract ?f)
   (assert (cycle (+ ?current-cycle 1)))
   (focus INPUT TRENDS WARNINGS))
```

只要有设备开着，Begin-Next-Cycle 规则就重新激活它自己，所以，下面的规则保证在没有设备开着的时候系统终止执行。

```
(defrule MAIN::End-Cycles
   (not (device (status on)))
   =>
   (printout t "All devices are off" crlf)
   (printout t "Halting monitoring system" crlf)
   (halt))
```

读取传感器的原始值

建立专家系统的下一步是从传感器读入其值。读取这些值的逻辑点是当前焦点正好是 INPUT 模块时。为了纠错和测试，能从几个来源读取传感器的值是很方便的。本节说明如何直接从传感器、自定义事实、文件或用户读取传感器的值。前面描述的 data-source 事实可用来指示此专家系统中数据的来源。标为“raw”数据的传感器值可直接存于 sensor 事实的 raw-value 槽中。对此原始值的分析和处理将在分析阶段执行。

为了以后方便地增加输入源，我们把所有的输入源用一个自定义类 DATA-SOURCE 来表示，如下：

```
(defclass INPUT::DATA-SOURCE
   (is-a USER))

(defmessage-handler INPUT::DATA-SOURCE
get-data (?name)
   (printout t "Input value for sensor "
   ?name ": ")
   (read))

(defmessage-handler INPUT::DATA-SOURCE
next-cycle (?cycle))

(definstances INPUT::user-data-source
   ([user] of DATA-SOURCE))
```

DATA-SOURCE 自定义类有两个消息处理程序：get-data 和 next-cycle。get-data 消息处理程序用来取回由？name 参数指定的传感器的原始值。默认地，该处理程序简单地向用户询问该值。next-cycle 消息处理程序用来处理在开始一个新的循环时，数据源所需要的所有过程。默认地，该处理程序不做任何事情。user-data-source 自定义实例定义 [user] 实例，如果是数据源，则向用户询问传感器值。

在实际工作环境中，这些值可能不是从用户而是直接从传感器获得，这要求外部函数做这项工作。让我们假定，get-sensor-value 函数将返回当前传感器的值，用一个参数代表需要该值的传感器标识号（为调用用 C、Ada、FORTRAN 或其他编程语言写的函数，需要重新编译此 CLIPS 源代码）。通过创建 DATA-SOURCE 的一个子类并指向它的一个实例，传感器值可以不用从用户而直接从传感器获得：

```
(defclass INPUT::SENSOR-DATA-SOURCE
   (is-a DATA-SOURCE))

(defmessage-handler INPUT::SENSOR-DATA-SOURCE
            get-data (?name)
(get-sensor-value ?name))

(definstances INPUT::sensor-data-source
   ([sensor] of SENSOR-DATA-SOURCE))
```

注意，唯一明显的改变就是重置了 get-data 处理程序，我们把它称为 get-sensor-value 函数。

为了测试或避免用户进行大量的输入，从一个“脚本”读取数据，而不是实际读取传感器的值或询问用户关于传感器的每一个值，这可能是更合人意的。完成此任务的一项技术是，将数据存于规则能够访问的事实中。下面的自定义类描述了用于存储数据值的实例：

```
(defclass INPUT::SENSOR-DATA
   (is-a USER)
   (multislot data))
```

其中，data 槽是该传感器实际数据值列表。

利用此自定义类，包含测试值的自定义实例可有如下的形式：

```
(definstances INPUT::sensor-instance-data-values
  ([S1-DATA-SOURCE] of SENSOR-DATA
    (data 100 100 110 110 115 120))
  ([S2-DATA-SOURCE] of SENSOR-DATA
    (data 110 120 125 130 130 135))
  ([S3-DATA-SOURCE] of SENSOR-DATA
    (data 100 120 125 130 130 125))
  ([S4-DATA-SOURCE] of SENSOR-DATA
    (data 120 120 120 125 130 135))
  ([S5-DATA-SOURCE] of SENSOR-DATA
    (data 110 120 125 130 135 135))
  ([S6-DATA-SOURCE] of SENSOR-DATA
    (data 115 120 125 135 130 135)))
```

注意包含数据的传感器的名字作为实例名的一部分，例如实例［S1-DATA-SOURCE］包含了传感器 S1 中的数据。

INSTANCE-DATA-SOURCE 类用于访问 SENSOR-DATA 实例中的传感器数据值：

```
(defclass INPUT::INSTANCE-DATA-SOURCE
   (is-a DATA-SOURCE))

(defmessage-handler INPUT::INSTANCE-DATA-SOURCE
            get-data (?name)
   ;; Locate the SENSOR-DATA instance.
   (bind ?sensor-data
      (instance-name
         (sym-cat ?name -DATA-SOURCE)))
   (if (not (instance-existp ?sensor-data))
      then (return nil))
   ;; Verify there are remaining data values.
   (bind ?data (send ?sensor-data get-data))
   (if (= (length$ ?data) 0)
      then
      (return nil))
   ;; Remove the first value in the list and
   ;; return it.
   (send ?sensor-data put-data (rest$ ?data))
   (nth$ 1 ?data))

(definstances INPUT::instance-data-source
   ([instance] of INSTANCE-DATA-SOURCE))
```

INSTANCE-DATA-SOURCE 的 get-data 消息处理程序比前面两个 get-data 的处理程序要稍微复杂。首先，它在传感器名后附加-DATA-SOURCE 以得到 SENSOR-DATA 实例名并确认它是否存在。然后它检查是否还有原始数据存在。如果存在，它将从表中去掉第一个值并返回它，在任何情况下遇到错误将返回 nil。

要讨论的最后一个输入技术涉及从一个数据文件读取信息。这比前面一些例子更复杂，因为必须处理几个问题。开始，必须打开文件，数据必须从文件中按序读出。前面讨论的输入技术与读取数据的顺序无关，可以以任何顺序访问传感器的新数据值。也不必知道还要读取多少传感器的值且有必要阻止在规则中对这种信息进行硬编码（hardcoding）。作一个假定将增加复杂性，假定传感器数据值可以不规定，即从前一次循环中获取传感器的原始数据值。文件数据可以用以下自定义类实现：

```
(defclass INPUT::FILE-DATA-SOURCE
   (is-a DATA-SOURCE)
   (slot file-logical-name (default FALSE))
   (multislot sensor)
   (multislot value))
```

file-logical-name 槽用来存放打开数据文件的逻辑名。既然数据从文件中获得，传感器的名字将存放在 sensor 槽中，原始数据将存放在 value 槽的相应位置。

必须处理的第一个问题是一开始打开数据文件。处理程序 get-file 用来询问用户文件名然后打开它：

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
           get-file ()
   (bind ?logical-name (gensym*))
   (while TRUE
     (printout t
     "What is the name of the data file? ")
     (bind ?file-name (readline))
     (if (open ?file-name ?logical-name "r")
        then
        (bind ?self:file-logical-name
        ?logical-name)
        (return))))
```

首先，通过调用 gensym* 函数为打开的文件创建了一个独一无二的逻辑名。在这个例子中只有一个文件被使用，所以可以硬编码一个特有名字放在处理程序中。但是很多情况下你要从不同的文件中读取数据，所以最好使用自动产生的逻辑名，以提高可扩展性。While 循环会不断地询问用户是否打开数据文件。如果文件成功打开，file-logical-name 槽的值将为产生的文件名，随后处理程序退出。否则只要 open 函数不能成功打开文件，就会一直询问用户要打开的文件名。

下一个决定涉及存储传感器数据的文件格式。只规定那些自上一个循环以来改变的传感器值是符合人意的。因此，对于一个给定的循环，要读取多少传感器的值是未知的。要读取的传感器值的顺序也不能事先确定。这些假定描述了一种数据格式，传感器名称被保存在要读取的原始传感器值的旁边。由于要读取的数据值的数目未知，因此，一个循环的数据结束值将由关键词 end-of-cycle 指示。数据格式的形式如下：

```
S1 100
S2 110
S3 100
S4 120
S5 110
S6 115
end-of-cycle
S2 120
S3 120
S5 120
S6 120
end-of-cycle
S1 110
S2 125
S3 125
S5 125
S6 125
end-of-cycle
...
```

put-sensor-value 消息处理程序将被用来存储从文件读取到 FILE-DATA-SOURCE 实例的数据值：

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
           put-sensor-value (?sensor ?value)
   (bind ?position (member$ ?sensor ?self:sensor))
   (if ?position
      then
      (bind ?self:value
        (replace$ ?self:value ?position
             ?position ?value))
      else
      (bind ?self:sensor ?self:sensor ?sensor)
   (bind ?self:value ?self:value ?value)))
```

首先，消息处理程序在 sensor 槽值中寻找传感器名。如果在 sensor 槽中，它就用新的值取代 value 槽中相应的值。否则，在包含 sensor 和 value 槽中的值的末尾加入新的传感器名和值。因为值只能被取代或者添加，如果在某个循环中没有指定传感器的值，当查询数据时仍使用原先的值。

当文件中没有传感器值剩下时，消息处理程序 close-data-source 用来关闭文件：

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
            close-data-source ()
   (close ?self:file-logical-name)
   (bind ?self:sensor (create$))
   (bind ?self:value (create$)))
```

函数 close 用来关闭 file-logical-name 槽中指定的文件。sensor 和 value 槽也置为空，这样就不能再获取到传感器值。

与前面的子类 DATA-SOURCE 不同，FILE-DATA-SOURCE 将重置 next-cycle 消息处理程序以从文件中获取下一个循环的传感器值：

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
            next-cycle (?cycle)
   (if (not ?self:file-logical-name)
      then (send ?self get-file))
   (bind ?name (read ?self:file-logical-name))
   (if (eq ?name EOF)
      then
      (send ?self close-data-source)
      (return))
   (while (and (neq ?name end-of-cycle)
               (neq ?name EOF))
      (bind ?raw-value
         (read ?self:file-logical-name))
      (if (eq ?raw-value EOF)
         then
         (send ?self close-data-source)
         (return))
      (send ?self put-sensor-value
         ?name ?raw-value)
      (bind ?name (read ?self:file-logical-name))
      (if (eq ?name EOF)
         then
         (send ?self close-data-source)
         (return))))
```

首先，如果 file-logical-name 槽仍然维持默认值 FALSE，则 get-file 消息会被送到实例以获取文件名并打开文件。该处理程序随后从数据文件中读取所有数据值直到遇到关键字 end-of-cycle。如果到达文件结束点，则 close-data-source 消息会被送到实例并关闭文件，删去所有存在的传感器值。否则，数据值被读取到实例中，以便随后通过 put-sensor-value 消息获取。

消息处理程序 get-data 必须被重置：

```
(defmessage-handler INPUT::FILE-DATA-SOURCE
            get-data (?name)
   (bind ?position (member$ ?name ?self:sensor))
(if ?position
   then
   (nth$ ?position ?self:value)
   else
   (return nil)))
```

它在 sensor 槽中搜索指定传感器，当存在时返回 value 槽中相应的值。最后，还必须加上一个包含 FILE-DATA-SOURCE 类中实例的自定义实例：

```
(definstances INPUT::file-data-source
   ([file] of FILE-DATA-SOURCE))
```

通过 4 种不同的数据源共享同一组消息处理程序，我们可以使用两个规则和一个自定义事实读取其中任何一个数据：

```
(deffacts local-cycle
   (local-cycle 0))

(defrule INPUT::next-cycle
(cycle ?cycle)
  ?f <- (local-cycle ~?cycle)
  (data-source ?source)
  (object (is-a DATA-SOURCE) (name ?source))
=>
  (send ?source next-cycle ?cycle)
  (retract ?f)
  (assert (local-cycle ?cycle)))

(defrule INPUT::Get-Sensor-Value-From-Data-Source
(cycle ?cycle)
(local-cycle ?cycle)
(data-source ?source)
(object (is-a DATA-SOURCE) (name ?source))
  ?s <- (sensor (name ?name)
                (raw-value none)
                (device ?device))
  (device (name ?device) (status on))
=>
  (bind ?raw-value (send ?source get-data ?name))
  (if (not (numberp ?raw-value))
      then
      (printout t "No data for sensor "
                  ?name crlf)
      (printout  t  "Halting  monitoring  system"
           crlf)
(halt)
else
(modify ?s (raw-value ?raw-value))))
```

local-cycle 事实用来指示在当前监视循环中是否已处理了 next-cycle 消息。规则 next-cycle 检查 cycle 事实指定的循环是否与 local-cycle 事实指定的不一样。如果这种情况存在，则匹配规则 LHS 的 DATA-SOURCE 实例被送到 next-cycle 消息中，而 local-cycle 事实更新为和 cycle 事实一样的值。对 DATA-SOURCE、SENSOR-DATA-SOURCE 和 INSTANCE-DATA-SOURCE 实例，next-cycle 消息处理程序不做任何事情。对 FILE-DATA-SOURCE 实例，next-cycle 消息处理程序将读取本次循环中的所有数据到实例中。

一旦 cycle 和 local-cycle 事实匹配，规则 Get-Sensor-Value-From-Data-Source 用来获取与运行设备相连的每个传感器的原始数据。Get-data 消息送到与规则 LHS 匹配的 DATA-SOURCE 实例中以获取指定传感器的原始数据。如果返回一个非数字值，监视系统会停止，因为非数字值表示没有更多的数据。否则，原始传感器值存放在 sensor 事实的 raw-value 槽中。

检测趋势

下一个是趋势阶段，此阶段确定传感器的当前状态，计算可能产生的趋势。传感器的当前状态（正常、低或高警戒线、低或高危险线）必须根据在输入阶段声明的原始传感器值来确定。传感器的当前状态存于 sensor 自定义模板的 slot 槽中。下面的规则确定传感器是否处于正常状态：

```
(defrule TRENDS::Normal-State
  ?s <- (sensor (raw-value ?raw-value&~none)
          (low-guard-line ?lgl)
          (high-guard-line ?hgl))
  (test (and (> ?raw-value ?lgl)
          (< ?raw-value ?hgl)))
  =>
  (modify ?s (state normal) (raw-value none)))
```

第一模式和接下来的测试 CE 找出任何处于正常状态的传感器。在 sensor 模式中的 raw-value 槽与值 none 相比较，保证不会在规则的测试 CE 中将符号与数值相比较。需要测试传感器是否处于正常状态的范围是低警戒线和高警戒线之间。测试 CE 将此规则限用于原始传感器值处于正常状态范围的情

况。此规则的唯一操作是声明推导出来的传感器状态。raw-value 槽也被设置为 none，以便在下一个输入阶段读取该槽值。

还需要 4 条或更多规则分析传感器剩下的 4 种可能状态。这些规则与对应的 Normal-state 规则类似，区别是从 sensor 事实得到的值和用来确定当前状态的测试 CE。所有 5 条状态规则可用规则的 RHS 侧的 if 表达式写成一条规则，以确定传感器的适当状态。然而，这种类型的编码打破了数据驱动系统的目的。此外，用一条难处理的规则将使进一步修改动作、条件或可能状态的子集变得更加复杂。剩下的 4 条状态分析规则如下：

```
(defrule TRENDS::High-Guard-Line-State
  ?s <- (sensor (raw-value ?raw-value&~none)
          (high-guard-line ?hgl)
          (high-red-line ?hrl))
  (test (and (>= ?raw-value ?hgl)
          (< ?raw-value ?hrl)))
  =>
  (modify ?s (state high-guard-line)
          (raw-value none)))

(defrule TRENDS::High-Red-Line-State
  ?s <- (sensor (raw-value ?raw-value&~none)
           (high-red-line ?hrl))
  (test (>= ?raw-value ?hrl))
  =>
  (modify ?s (state high-red-line)
          (raw-value none)))

(defrule TRENDS::Low-Guard-Line-State
  ?s <- (sensor (raw-value ?raw-value&~none)
           (low-guard-line ?lgl)
           (low-red-line ?lrl))
  (test (and (> ?raw-value ?lrl)
           (<= ?raw-value ?lgl)))
  =>
  (modify ?s (state low-guard-line)
           (raw-value none)))

(defrule TRENDS::Low-Red-Line-State
  ?s <- (sensor (raw-value ?raw-value&~none)
           (low-red-line ?lrl))
  (test (<= ?raw-value ?lrl))
  =>
  (modify ?s (state low-red-line)
           (raw-value none)))
```

上述 5 条规则确定当前循环的传感器状态。由于检测传感器的趋势是 TRENDS 模块的目标之一，因此，这将需要维护有关传感器过去状态的信息。下面的自定义模板将用于存储这类信息。由于 TRENDS 和 WARNINGS 模块都要使用此自定义模板，因此，将它放于 MAIN 模块中。

```
(deftemplate MAIN::sensor-trend
   (slot name)
   (slot state (default normal))
   (slot start (default 0))
   (slot end (default 0))
   (slot shutdown-duration (default 3)))
```

name 槽是传感器的名称，state 槽对应于传感器的最当前状态，start 槽是传感器处于当前状态的第一个循环，end 槽是当前循环，shutdown-duration 槽是传感器在其相关设备必须被关机之前处于警戒线范围内的时间。

更新趋势信息的规则将依赖于每个传感器的事实表中的 sensor-trend 事实。因此，传感器的初始趋势将被定义于一个自定义事实结构中。

```
(deffacts MAIN::start-trends
(sensor-trend (name S1) (shutdown-duration 3))
(sensor-trend (name S2) (shutdown-duration 5))
(sensor-trend (name S3) (shutdown-duration 4))
(sensor-trend (name S4) (shutdown-duration 4))
(sensor-trend (name S5) (shutdown-duration 4))
(sensor-trend (name S6) (shutdown-duration 2)))
```

有了这种信息，就可以定义两条规则来监视传感器的趋势。其中一条规则监视上一次循环以来仍没有变化的趋势，另一条则监视上一次循环以来发生了变化的趋势：

```
(defrule TRENDS::State-Has-Not-Changed
  (cycle ?time)
  ?trend <- (sensor-trend
(name ?sensor) (state ?state)
         (end ?end-cycle&~?time))
  (sensor (name ?sensor) (state ?state)
          (raw-value none))
  =>
  (modify ?trend (end ?time)))

(defrule TRENDS::State-Has-Changed
  (cycle ?time)
  ?trend <- (sensor-trend
(name ?sensor) (state ?state)
               (end ?end-cycle&~?time))
  (sensor (name ?sensor)
          (state ?new-state&~?state)
          (raw-value none))
  =>
  (modify ?trend (start ?time)
            (end ?time)
            (state ?new-state)))
```

两条规则的第一模式都建立循环。下一模式则找出前面循环的 sensor-trend 事实。end 槽上的约束确保这些规则不会进入死循环。对于 State-Has-Not-Changed 规则，下一模式检查前面循环的状态是否与当前循环相同。State-Has-Changed 规则中的约束? new-state&~? state 则进行相反的检查，确保其状态自上一次循环以来已发生变化。检查 raw-value 槽是否等于 none 可以防止在确定传感器的当前状态之前不会确定趋势。在两个规则中，结束（end）循环时间得到更新。如果状态已改变，则状态值和启动循环时间也必须更新。

发出警告

循环的最后阶段是警告阶段。此阶段必须处理 3 种操作：进入危险区的传感器必须关闭相关的设备；处于警戒区达到规定循环次数的传感器也要关闭相关的设备；处于警戒区且没有关闭相关设备的传感器要求发出警告信息。

下面的规则是关闭已进入危险区的传感器：

```
(defrule WARNINGS::Shutdown-In-Red-Region
  (cycle ?time)
  (sensor-trend
     (name ?sensor)
     (state ?state&high-red-line | low-red-line))
  (sensor (name ?sensor) (device ?device))
  ?on <- (device (name ?device) (status on))
  =>
  (printout t "Cycle " ?time " - ")
  (printout t "Sensor " ?sensor " in "
?state crlf)
  (printout t "   Shutting down device "
?device crlf)
  (modify ?on (status off)))
```

可使用 sensor-trend 事实检查传感器的状态。如果传感器处于危险区，则关掉相关的设备。相关设备的传感器只要有一个进入危险区，则立即关掉这些相关设备，因此，它主要不是用于检查传感器处于这种状态有多久。

Shutdown-In-Guard-Region 规则与上一条规则类似，但该传感器必须已进入警戒区一段时间（由 sensor-trend 事实的 shutdown-duration 槽规定）才能关闭。传感器处于特定状态的时间可以从 sensor-trend 事实的 end 槽减去 start 槽得到。

```
(defrule WARNINGS::Shutdown-In-Guard-Region
  (cycle ?time)
  (sensor-trend
      (name ?sensor)
      (state ?state&high-guard-line |
             low-guard-line)
      (shutdown-duration ?length)
      (start ?start) (end ?end))
  (test (>= (+ (- ?end ?start) 1)  ?length))
  (sensor (name ?sensor) (device ?device))
  ?on <- (device (name ?device) (status on))
  =>
  (printout t "Cycle " ?time " - ")
  (printout t "Sensor " ?sensor " in " ?state " ")
  (printout t "for " ?length " cycles " crlf)
  (printout t "   Shutting down device " ?device
              crlf)
  (modify ?on (status off)))
```

在 sensor-trend 和相关的测试 CE 中增加 shutdown-duration 槽是此规则和 Shutdown-In-Red-Region 规则之间的唯一主要区别。这两个模式确定传感器是否处于警戒区的时间太长，足以关掉与该传感器相关的设备。

监视系统的最后一条规则是发出处于警戒区传感器的警告信息，但它处于此区域的时间不够长，不足以关掉相关的设备：

```
(defrule WARNINGS::Sensor-In-Guard-Region
  (cycle ?time)
  (sensor-trend
      (name ?sensor)
      (state ?state&high-guard-line |
             low-guard-line)
      (shutdown-duration ?length)
    (start ?start) (end ?end))
(test (< (+ (- ?end ?start) 1) ?length))
=>
(printout t "Cycle " ?time " - ")
(printout t "Sensor " ?sensor " in "
           ?state crlf))
```

此规则是作为 Shutdown-In-Guard-Region 规则的补充。测试（test）CE 已被修改以检测该传感器处于警戒区的时间小于需要关闭其相关设备所需的循环次数。既然该传感器的相关设备不必被关闭，因此，没有包含确定相关设备的模式在内。

这最后一条规则完成了一个很简单的监视系统的基本工作。附加的规则可以包括特定的应该受监视的情况，或者，提供处理复杂监视情况并描述传感器/设备关系的通用模型。

12.6 小结

本章展示了用 CLIPS 表示 MYCIN-样式的确定性因子的技术。对象－属性－值（OAV）三元组用事实来表示。每个事实中有一个附加槽表示该事实的确定性因子。通过使用约束在规则 LHS 侧的确定性值，可利用规则来计算在规则 RHS 侧新近被声明事实的确定性值。然后，可使用一条规则将出现两次的 OAV 三元组组合为一次出现，它有一个根据原始的两个三元组的确定性因子求得的新确定性因子。

判定树也可以用 CLIPS 的正向链系统表示。遍历判定树有几种算法，包括：二叉树、多叉树和学习式多叉树。学习式多叉树的算法可以用 CLIPS 来实现。

CLIPS 也可模仿反向链推理策略。反向链推理机可以利用 CLIPS 规则建立，它建立的基础中有一部分是依赖于程序语言中实现反向链的算法。反向链规则可以表示为事实，并受 CLIPS 中反向链推理规则的作用。逐步跟踪一个简单的反向链对话展示了该范例用 CLIPS 来表示的原理。

本章的最后一个例子是一个简单的监视专家系统。监视问题的初始描述可用来作为编写专家系统

的开始。系统中作了一些假定，并随着越来越多的规则添加到此专家系统，也相应地添加了一些细节。监视系统的执行可分为3个阶段。输入阶段是收集传感器的原始数据值。书中展示了几种不同的提供数据值的方法。趋势阶段是分析传感器的原始数据值，并检查发展趋势。最后是警告阶段，此阶段发出警告信息，并根据趋势阶段的分析结果执行适当的操作。

习题

12.1 对于下列两种情况，写出CLIPS规则来组合如第12.2节所示的MYCIN确定性因子：

New Certainty = $(CF_1 + CF_2) + (CF_1 * CF_2)$

若 $CF_1 \leqslant 0$ 且 $CF_2 \leqslant 0$

New Certainty = $\dfrac{CF_1 + CF_2}{1 - \min\{|CF_1|, |CF_2|\}}$

若 $-1 < CF_1 * CF_2 < 0$

12.2 利用第12.2节中展示的几种技术，说明经典概率如何合并到CLIPS中。列出在规则中使用经典概率的可能的优缺点。

12.3 利用程序语言，如LISP、C或PASCAL，实现第12.3节中描述的Solve_Tree_and_Learn算法。利用动物识别的例子测试你的实现情况。

12.4 利用程序语言，如LISP、C或PASCAL，实现第12.4节中描述的Solve_Goal和Attempt_Rule算法。利用选择酒的例子测试你的实现情况。

12.5 修改DATA-SOURCE的get-data规则，使它只允许接收用户输入的数字。仅当在输入期间输入了词halt，本规则才产生执行终止。

12.6 生成DATA-SOURCE自定义类的子类，以允许用户使用回车符表示应保留每个传感器的前一次值。例如，若传感器1和2的前一次值分别为100和120，则下列响应：

```
What is the value for sensor 1? ↵
What is the value for sensor 2? 130↵
```

应将传感器1的原始数据值设置为100、传感器2的原始数据值设置为130。若传感器的前一次值未知，则应该作什么假定、采取什么措施？

12.7 修改程序使得已被关闭的设备能继续受传感器的监视。如果该设备的所有传感器回到正常状态，则应该重新将设备打开。

12.8 即使另一条规则在议程中，Sensor-In-Guard-Region规则也会发出一则关于某传感器的警告信息，此后关闭它相关的设备。如果WARNINGS模块的另一条规则会关闭它相关的设备，那么，应该对此规则作怎样的修改来防止它发出警告信息？

12.9 增加一些规则，打印在警告阶段的信息，指出某传感器已处于正常状态至少n个循环，数字n在事实中规定。只在每第n个循环打印。

12.10 怎样修改第12.4节中的反向链规则，以允许正向链？

12.11 修改习题11.1中编写的程序，使得代表灌木的事实可利用load-facts函数从一个文件载入。在提供解释后，应该询问用户是否需要再次选择灌木。如果是，则应该重新运行该程序。

12.12 利用习题11.3编写的程序，修改习题10.10中编写的程序，使得仅有还没有被选择的小装置列在Add Gizmo子菜单上、使已被选择的小装置列在Remove Gizmo子菜单上。而主菜单上，仅当有未被选择的小装置时，才显示Add Gizmo菜单选项；仅当有已被选择的小装置时，才显示Remove Gizmo菜单选项。

12.13 利用习题11.3编写的程序，修改习题10.4中编写的程序，以便使用菜单驱动式界面。应给用户提供独立的菜单选项，以说明待识别的宝石的颜色、硬度和密度。使用包含有效颜色的子菜单进行颜色说明。应提供菜单选项，列出满足当前规定条件的宝石。例如，若用户只规定了宝

石的颜色为黑色，则此菜单选项只列出黑色的宝石。还应该提供另外两个菜单选项，一个列出规定条件的当前值，另一个将所有条件复位到一个未规定的状态。

参考文献

(Firebaugh 88). Morris W. Firebaugh, *Artificial Intelligence: A Knowledge-Based Approach*, Boyd & Fraser Publishing, p. 309, 1988.

附录 A　一些有用的等式

$$\sim\sim p \equiv dp$$

$$p \rightarrow q \equiv \sim p \vee q \equiv \sim q \rightarrow \sim p$$

$$\sim(p \wedge q) \equiv \sim p \vee \sim q$$
$$\sim(p \vee q) \equiv \sim p \wedge \sim q$$

$$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$$
$$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$$

$$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$$
$$(p \vee q) \vee r \equiv p \vee (q \vee r)$$

$$p \wedge q \equiv q \wedge p$$
$$p \vee q \equiv q \vee p$$

$$\sim(\forall x)\ P(x) \equiv (\exists x)\ \sim P(x)$$
$$\sim(\exists x)\ P(x) \equiv (\forall x)\ \sim P(x)$$

$$(\forall x)\ P(x) \wedge (\forall x)\ Q(x) \equiv (\forall x)\ (P(x) \wedge Q(x))$$
$$(\exists x)\ P(x) \vee (\exists x)\ Q(x) \equiv (\exists x)\ (P(x) \vee Q(x))$$

注意：$\forall$不能分配给$\vee$运算，$\exists$不能分配给$\wedge$运算，所以，在下面的两个等式中，要引入一个哑变量z。

$$\begin{aligned}(\forall x)\ P(x) \vee (\forall x)\ Q(x) &\equiv (\forall x)\ P(x) \vee (\forall z)\ Q(z) \\ &\equiv (\forall x)\ (\forall z)\ (P(x) \vee Q(z))\end{aligned}$$

$$\begin{aligned}(\exists x)\ P(x) \wedge (\exists x)\ Q(x) &\equiv (\exists x)\ P(x) \wedge (\exists z)\ Q(z) \\ &\equiv (\exists x)\ (\exists z)\ (P(x) \wedge Q(z))\end{aligned}$$

附录 B　一些基本量词公式及其含义

公　式	含　义
$(\forall x)(P(x) \rightarrow Q(x))$	对所有 x，P 成立则 Q 成立
$(\forall x)(P(x) \rightarrow \sim Q(x))$	对所有 x，P 成立则 Q 不成立
$(\exists x)(P(x) \wedge Q(x))$	对所有 x，P，Q 都成立
$(\exists x)(P(x) \wedge \sim Q(x))$	对所有 x，P 成立但 Q 不成立
$(\forall x)P(x)$	对所有 x，P 成立
$(\exists x)P(x)$	对某些 x，P 成立
$\sim(\forall x)P(x)$	不是所有 x 都使 P 成立
$(\forall x) \sim P(x)$	对所有 x，P 不成立
$(\forall x)(\exists y)P(x, y)$	对所有 x，都有某一 y 使 P 成立
$(\exists x) \sim P(x)$	有些 x 使 P 不成立

附录C　一些集合性质

交换率	$A \cup B = B \cup A$ $A \cap B = B \cap A$
结合率	$A \cup (B \cup C) = (A \cup B) \cup C$ $A \cap (B \cap C) = (A \cap B) \cap C$
幂等率	$A \cup A = A$ $A \cap A = A$
分配率	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
排中率	$A \cup A' = U$
矛盾率	$A \cap A' = \varnothing$
同一率	$A \cup \varnothing = A$ $A \cap U = A$
吸收率	$A \cup (A \cap B) = A$ $A \cap (A \cup B) = A$
德·摩根定律	$(A \cap B)' = A' \cup B'$ $(A \cup B)' = A' \cap B'$
自反率	$(A')' = A$
等值式	$(A' \cup B) \cap (A \cup B') = (A' \cap B') \cup (A \cap B)$
对称差	$(A' \cap B) \cup (A \cap B') = (A' \cup B') \cap (A \cup B)$

附录 D　CLIPS 支持信息

本书附带的光盘含有 CLIPS 6.22 在 MS-DOS、Windows 2000/XP，以及 MacOS X 上的可执行程序；此外，还有 CLIPS 源程序代码、《CLIPS 参考手册》（CLIPS Reference Manual）三卷、《CLIPS 用户指南》（CLIPS User's Guide）。参考手册第一卷，《基本编程指南》（Basic Programming Guide），主要包括 CLIPS 语法定义及用法例子。第二卷，《高级编程指南》（The Advanced Programming Guide），包括定制 CLIPS 的细节，在 CLIPS 中增加新的函数、嵌入 CLIPS，以及其他一些高级特性介绍。第三卷，《用户界面指南》（The Interfaces Guide），包括 CLIPS 与环境有关的界面信息。可以通过在 X Windows 界面下编译光盘中的源代码来获得 CLIPS 的一个 X Windows 版本。

CLIPS 的漏洞修补、更新以及其他一些相关信息可以在 CLIPS 的主页 http://www.ghg.net/clips/CLIPS.html 上获得。关于 CLIPS 的问题可以发送电子邮件到下面的地址：clipsYYYY@ghg.net，其中 YYYY 是当前的年份（例如，2004）。Usenet 用户可以在 comp.ai.shells 新闻组上查找信息或提交问题。

CLIPS 开发者论坛在 http://www.cpbinc.com/clips，这是一个基于线程的消息公告栏，提供了研究讨论、开发和实现 CLIPS 专家系统以及相关技术的站点。

CLIPS 用户还可以通过电子会议来交流信息，与会者可以 E-mail 的形式来提交问题、看法、答复、评论等。所有的与会者都可通过 E-mail 来收到会议讨论信息，要想注册成为与会者，只需向 clips-request@discomsys.com 发一个 E-mail 包含单词“subscribe”。共中，subject 栏可以不填，但“Reply:”，“Reply to:”，或“From:”栏必须填写。注册后你会收到一个 E-mail 告诉你怎样参加会议。

附录E　CLIPS命令与函数概要

议程

```
(agenda [<module-name>])
```

列出指定模块议程中的活动（如果没有指定＜module-name＞，则为当前模块）。

```
(clear-focus-stack)
```

从焦点栈中返回所有模块名。

```
(focus <module-name>+)
```

将一个或多个模块按照它们在参数表中的反序压入焦点栈中。

```
(get-focus)
```

返回当前焦点栈中的模块名。

```
(get-focus-stack)
```

将焦点栈中所有模块名作为一个多字段值返回。

```
(get-salience-evaluation)
```

返回当前优先级计算方式。

```
(get-strategy)
```

返回当前冲突的解决策略。

```
(halt)
```

终止规则执行。

```
(list-focus-stack)
```

列出焦点栈中所有的模块名。

```
(pop-focus)
```

返回当前焦点的模块名并从焦点栈中删除当前焦点。

```
(refresh-agenda [<module-name>])
```

对指定模块（如果没有指定＜module-name＞，则为当前模块）议程中的规则重新计算优先级。

```
(run [<integer-expression>])
```

开始执行当前焦点中的规则。如果指定了＜integer-expression＞，则只执行该序号的规则。否则，直到议程为空才停止执行。

```
(set-salience-evaluation <behavior>)
<behavior> ::= when-defined | when-activated |
               every-cycle
```

设置优先级计算方式。

```
(set-strategy <strategy>)
<strategy> ::= depth | breadth | simplicity |
               complexity | lex | mea | random
```

设置当前冲突归结策略。

基本数学函数

(abs <numeric-expression>)

返回唯一参数的绝对值。

(div <numeric-expression> <numeric-expression>+)

返回第一个参数除以其余参数的值，运算满足整数除法规则。

(float <numeric-expression>)

将唯一参数转换为浮点数类型并返回。

(integer <numeric-expression>)

将唯一参数转换为整数类型并返回。

(max <numeric-expression> <numeric-expression>+)

返回最大参数的值。

(min <numeric-expression> <numeric-expression>+)

返回最小参数的值。

(+ <numeric-expression> <numeric-expression>+)

返回所有参数的和。

(- <numeric-expression> <numeric-expression>+)

返回第一个参数减去其余所有参数的差。

(* <numeric-expression> <numeric-expression>+)

返回所有参数的乘积。

(/ <numeric-expression> <numeric-expression>+)

返回第一个参数除以其余所有参数的值。

转换函数

(deg-grad <numeric-expression>)

将参数从度数值转换为梯度值并返回。

(deg-rad <numeric-expression>)

将参数从度数值转换为弧度值并返回。

(exp <numeric-expression>)

返回 e 的参数次幂。

(grad-deg <numeric-expression>)

将参数从梯度值转换为度数值并返回。

(log <numeric-expression>)

返回参数以 e 为底的对数值。

(log10 <numeric-expression>)

返回参数以 10 为底的对数值。

(mod <numeric-expression> <numeric-expression>)

返回第一个参数除以第二个参数的余数值。

(pi)

返回 π 值。

```
(rad-deg <numeric-expression>)
```

将参数从弧度值转换为度数值并返回。

```
(round <numeric-expression>)
```

返回参数最接近的整数值。

```
(sqrt <numeric-expression>)
```

返回参数的平方根。

```
(** <numeric-expression> <numeric-expression>)
```

返回第一个参数的第二个参数次幂。

调试命令

```
(dribble-off)
```

停止对跟踪文件的输出，该跟踪文件由函数 dribble-on 打开。若跟踪文件成功关闭，则返回 TRUE，否则返回 FALSE。

```
(dribble-on <file-name>)
```

将屏幕输出重定位到跟踪文件＜file-name＞中。如果跟踪文件打开成功，则返回 TRUE，否则返回 FALSE。

```
(list-watch-items)
```

列出监视项的当前状态。

```
(unwatch <watch-item>)
```

当某种 CLIPS 操作发生时，关闭显示信息。

```
(watch <watch-item>)
<watch-item> ::= activations | all | compilations |
                 deffunctions | facts | focus |
                 generic-functions | globals |
                 instances | messages |
                 message-handlers | methods |
                 rules | slots | statistics
```

当某种 CLIPS 操作发生时，激活显示信息。

自定义类

```
(browse-classes [<class-name>])
```

列出指定的类及其子类的继承关系。如果没有指定类，＜class-name＞默认为 OBJECT。

```
(class-abstractp <class-name>)
```

如果指定的类为抽象类，返回 TRUE，否则 FALSE。

```
(class-existp <class-name>)
```

如果指定的类已定义，返回 TRUE，否则 FALSE。

```
(class-reactivep <class-name>)
```

如果指定的类为反应类，返回 TRUE，否则 FALSE。

```
(class-slots <class-name> [inherit])
```

以多字段值形式返回类中定义的显式槽名。如果使用了可选的 inherit 关键字，将包括继承槽。

```
(class-subclasses <class-name> [inherit])
```

以多字段值形式返回一个类的直接父类。如果使用了可选的 inherit 关键字，将包括间接子类。

```
(class-superclasses <class-name> [inherit])
```

以多字段值形式返回一个类的直接父类。如果使用了可选的 inherit 关键字，将包括间接父类。

```
(defclass-module [<class-name>])
```

返回指定自定义类的定义所在的模块。

```
(describe-classes [<class-name>])
```

提供一个类的详细描述。

```
(get-class-defaults-mode)
```

返回定义类时的当前默认模式。

```
(get-defclass-list [<module-name>])
```

返回一个包含指定模块（当没有指定<module-name>时，为当前模块）中的类列表的多字段值。

```
(list-defclasses [<module-name>])
```

列出指定模块（当没有指定<module-name>时，为当前模块）中的类列表。

```
(ppdefclass <class-name>)
```

显示指定自定义实例的文本。

```
(set-class-defaults-mode convenience | conservation)
```

设置定义类时的默认模式。

```
(slot-allowed-values <class-name> <slot-name>)
```

以多字段值形式返回槽的允许值。

```
(slot-cardinality <class-name> <slot-name>)
```

以多字段值形式返回一个多槽所允许的最小和最大字段数。

```
(slot-default-value <class-name> <slot-name>)
```

返回与槽相关关联的默认值。

```
(slot-direct-accessp <class-name> <slot-name>)
```

如果指定的槽可以直接存取则返回 TRUE，否则返回 FALSE。

```
(slot-existp <class-name> <slot-name> [inherit])
```

如果指定的槽在类中，则返回 TRUE，否则返回 FALSE。如果使用了可选的 inherit 关键字，则槽可以是继承的。

```
(slot-facets <class-name> <slot-name>)
```

以多字段值形式返回指定槽的侧面值。

```
(slot-initablep <class-name> <slot-name>)
```

如果指定的槽可以初始化，返回 TRUE，否则返回 FALSE。

```
(slot-publicp <class-name> <slot-name>)
```

如果指定的槽是公用的，返回 TRUE，否则返回 FALSE。

```
(slot-range <class-name> <slot-name>)
```

以多字段值形式返回槽所允许的最小和最大值。

```
(slot-sources <class-name> <slot-name>)
```

以多字段值形式返回为指定类的指定槽提供了侧面的类名。

```
(slot-types <class-name> <slot-name>)
```

以多字段值形式返回指定类的指定槽所允许的初始类型。

```
(slot-writeablep <class-name> <slot-name>)
```

如果指定槽是可写的，返回 TRUE，否则返回 FALSE。

```
(subclassp <class1-name> <class2-name>)
```

如果第一个类是第二个类的子类，返回 TRUE。

```
(superclassp <class1-name> <class2-name>)
```

如果第一个类是第二个类的父类，返回 TRUE。

```
(undefclass <class-name>)
```

删除指定的类。

自定义事实

```
(deffacts-module <deffacts-name>)
```

返回指定自定义事实的定义所在的模块。

```
(get-deffacts-list [<module-name>])
```

返回指定模块（如果没有指定＜module-name＞，则为当前模块）的所有自定义事实列表。

```
(list-deffacts [<module-name>])
```

列出指定模块（如果没有指定＜module-name＞，则为当前模块）的自定义事实。

```
(ppdeffacts <deffacts-name>)
```

显示指定自定义事实的文本。

```
(undeffacts <deffacts-name>)
```

删除指定的自定义事实。

自定义函数

```
(deffunction-module <deffunction-name>)
```

返回指定自定义函数的定义所在的模块。

```
(get-deffunction-list [<module-name>])
```

返回一个包含指定模块（如果没有指定＜module-name＞，则为当前模块）中的自定义函数列表的多字段值。

```
(list-deffunctions [<module-name>])
```

列出指定模块（如果没有指定＜module-name＞，则为当前模块）中的自定义函数。

```
(ppdeffunction <deffunction-name>)
```

显示指定自定义函数的文本。

```
(undeffunction <deffunction-name>)
```

删除指定的自定义函数。

自定义类属

```
(defgeneric-module <defgeneric-name>)
```

返回指定自定义类属的定义所在的模块。

```
(get-defgeneric-list [<module-name>])
```

返回包含指定模块（如果没有指定<module-name>，则为当前模块）中的自定义类属列表的多字段值。

```
(list-defgenerics [<module-name>])
```

列出指定模块（如果没有指定<module-name>，则为当前模块）的自定义类属。

```
(ppdefgeneric <defgeneric-name>)
```

显示指定自定义类属的文本。

```
(preview-generic <generic-function-name> <expression>*)
```

以降序形式列出对于特定类属函数调用可用的方法，但不执行。

```
(type <expression>)
```

返回参数的类型（或类）名。

```
(undefgeneric <defgeneric-name>)
```

删除指定的自定义类属及其所有方法。

自定义全局变量

```
(defglobal-module <defglobal-name>)
```

返回指定自定义全局变量的定义所在的模块。

```
(get-defglobal-list [<module-name>])
```

返回包含指定模块（如果没有指定<module-name>，则为当前模块）中的自定义全局变量列表的多字段值。

```
(get-reset-globals)
```

返回重置全局变量开关的当前值（TRUE 或 FALSE）。

```
(list-defglobals [<module-name>])
```

列出指定模块（如果没有指定<module-name>，则为当前模块）的自定义全局变量。

```
(ppdefglobal <defglobal-name>)
```

显示指定自定义全局变量的文本。

```
(set-reset-globals <boolean-expression>)
```

设置重置全局变量开关。如果开关打开（默认为 TRUE），则执行 reset 命令时，全局变量重置为初始值。

```
(show-defglobals [<module-name>])
```

列出指定模块（如果没有指定<module-name>，则为当前模块）中的自定义全局变量及其当前值。

```
(undefglobal <defglobal-name>)
```

删除指定的自定义全局变量。

自定义实例

```
(definstances-module <definstances-name>)
```

返回指定自定义实例的定义所在的模块。

```
(get-definstances-list [<module-name>])
```

返回包含指定模块（如果没有指定<module-name>，则为当前模块）中的自定义实例列表的多字段值。

```
(list-definstances [<module-name>])
```

列出指定模块（如果没有指定＜module-name＞，则为当前模块）中的自定义实例。

```
(ppdefinstances <definstances-name>)
```

显示指定自定义实例的文本。

```
(undefinstances <definstances-name>)
```

删除指定的自定义实例。

自定义消息处理程序

```
(call-next-handler)
```

当从消息处理程序内部调用时，调用被正在执行的消息处理程序重载或覆盖的下一个消息处理程序。

```
(get-defmessage-handler-list <class-name> [inherit])
```

返回包含指定类的类名、处理程序名和处理程序类型三元组的多字段值。如果指定了 inherit，则被继承的消息处理程序也包含在多字段值内。

```
(list-defmessage-handlers [<class-name> [inherit]])
```

列出当前模块中指定类的自定义消息处理程序。如果没有指定类，当前模块中所有类的自定义消息处理程序都被列出。如果指定了 inherit，则被继承的消息处理程序也将显示。

```
(message-handler-existp <class-name><handler-name>
                        [<handler-type>])
<handler-type> ::= around | before |
                   primary | after
```

如果指定类的指定消息处理程序是直接定义的（不是继承的），则返回 TRUE，否则返回 FALSE。

```
(next-handlerp)
```

如果有另一个消息处理程序可执行，则返回 TRUE，否则返回 FALSE。

```
(override-next-handler <expression>*)
```

调用下一个被覆盖的处理程序，允许改变参数。

```
(ppdefmessage-handler <class-name> <handler-name>
                      [<handler-type>])
<handler-type> ::= around | before |
                   primary | after
```

显示指定自定义消息处理程序的文本。如果没有指定＜handler-type＞，则为 primary。

```
(preview-send <class-name> <message-name>)
```

显示对某个送到指定类的实例消息可用的处理程序的列表。

```
(undefmessage-handler <class-name> <handler-name>
                      [<handler-type>])
<handler-type> ::= around | before |
                   primary | after
```

删除指定的自定义消息处理程序。如果没有指定＜handler-type＞，则为 primary。

自定义方法

```
(call-next-method)
```

调用下一个被覆盖的方法。

```
(call-specific-method <defgeneric-name> <method-index>
                      <expression>*)
```

调用一个忽略方法优先级类属函数的特定方法。

```
(get-defmethod-list [<defgeneric-name>])
```

返回包含当前模块中自定义类属名和方法索引所组成的对的多字段值。如果指定了自定义类属名，则只有属于这个自定义类属的方法会被包括在返回值中。

```
(get-method-restrictions <defgeneric-name>
                         <method-index>)
```

返回包含指定方法限制信息的多字段值。

```
(list-defmethods [<defgeneric-name>])
```

升序列出当前模块中的所有自定义方法。如果指定了自定义类属名，则只列出属于这个自定义类属的方法。

```
(next-methodp)
```

如果在一个类属函数的方法中调用，如果有一个方法被当前方法覆盖，函数 next-methodp 返回 TRUE；否则返回 FALSE。

```
(override-next-method <expression>*)
```

调用下一个被覆盖的方法，允许提供新的参数。

```
(ppdefmethod <defgeneric-name> <index>)
```

显示与指定类属函数和方法索引相关联的自定义方法的文本。

```
(preview-generic <generic-function-name>
                 <expression>*)
```

降序列出对某特定类属函数调用可用的方法，但不执行。

```
(undefmethod <defgeneric-name> <index>)
```

删除与指定类属函数和方法索引相关联的自定义方法。

自定义模块

```
(get-current-module)
```

返回当前模块。

```
(get-defmodule-list)
```

返回所有自定义模块列表。

```
(list-defmodules)
```

列出 CLIPS 环境中所有自定义模块。

```
(ppdefmodule <defmodule-name>)
```

显示指定自定义模块的文本。

```
(set-current-module <defmodule-name>)
```

设置当前模块为指定模块名，并返回前一个当前模块。

自定义规则

```
(defrule-module <defrule-name>)
```

返回指定自定义规则的定义所在的模块。

```
(get-defrule-list [<module-name>])
```

返回指定模块（如果没有指定＜module-name＞，则为当前模块）中的所有自定义规则列表。

```
(get-incremental-reset)
```

返回增量重置开关的当前值。

```
(list-defrules [<module-name>])
```

列出指定模块中的自定义规则（如果没有指定＜module-name＞，则为当前模块）。

```
(matches <defrule-name>)
```

显示与指定规则模式相匹配的事实及部分匹配列表。

```
(ppdefrule <defrule-name>)
```

显示指定自定义规则的文本。

```
(refresh <defrule-name>)
```

刷新指定自定义规则。规则中那些已触发但仍然有效的活动被放入议程中。

```
(remove-break [<defrule-name>])
```

删除指定规则的断点，如果没有指定规则，则删除所有的断点。

```
(set-break <defrule-name>)
```

为指定规则设置断点。这将导致在该规则触发前停止规则执行。

```
(set-incremental-reset <boolean-expression>)
```

设置增量重置开关。

```
(show-breaks [<module-name>])
```

显示指定模块中设置了断点的规则（如果没有指定＜module-name＞，则为当前模块）。

```
(undefrule <defrule-name>)
```

删除指定的自定义规则。

自定义模板

```
(deftemplate-module <deftemplate-name>)
```

返回指定自定义模板的定义所在的模块。

```
(get-deftemplate-list [<module-name>])
```

返回指定模块（如果没有指定＜module-name＞，则为当前模块）中的所有自定义模板列表。

```
(list-deftemplates [<module-name>])
```

列出指定模块中的自定义模板（如果没有指定＜module-name＞，则为当前模块）。

```
(ppdeftemplate <deftemplate-name>)
```

显示指定自定义模板的文本。

```
(undeftemplate <deftemplate-name>)
```

删除指定的自定义模板。

环境命令

```
(apropos <lexeme>)
```

显示 CLIPS 环境中当前已定义了的包含指定子串＜lexeme＞的所有符号。

```
(batch <file-name>)
```

用文件＜file-name＞的内容替换标准输入，从而允许 CLIPS 交互式命令进行批处理。若执行成功

返回 TRUE，否则返回 FALSE。

```
(batch* <file-name>)
```

执行文件中的命令，与 batch 命令的不同在于执行了指定文件中的所有命令后才返回而不是替换标准输入。

```
(bload <file-name>)
```

以二进制方式调入文件。若调用成功返回 TRUE，否则返回 FALSE。

```
(bsave <file-name>)
```

以二进制方式保存到文件。

```
(clear)
```

清除 CLIPS 环境中的所有结构。

```
(exit)
```

退出 CLIPS 环境。

```
(get-auto-float-dividend)
```

返回自动浮点除法开关的当前值。

```
(get-dynamic-constraint-checking)
```

返回动态约束检查开关的当前值。

```
(get-static-constraint-checking)
```

返回静态约束检查开关的当前值。

```
(load <file-name>)
```

在 CLIPS 环境中调入文件＜file-name＞中的结构，若成功返回 TRUE，否则返回 FALSE。

```
(load* <file-name>)
```

在 CLIPS 环境中调入由＜file-name＞指定的文件中的结构并显示有用的消息，若成功返回 TRUE，否则返回 FALSE。

```
(options)
```

列出 CLIPS 编译器标志的设置。

```
(reset)
```

重新设置 CLIPS 环境。

```
(save <file-name>)
```

保存 CLIPS 环境中的所有结构到由＜file-name＞指定的文件中。

```
(set-auto-float-dividend <boolean-expression>)
```

若＜boolean-expression＞为 FALSE，则关闭自动浮点除法开关，否则激活自动浮点除法开关。然后返回旧的自动浮点除法开关值。

```
(set-dynamic-constraint-checking <boolean-expression>)
```

若＜boolean-expression＞为 FALSE，则关闭动态约束检查，否则激活动态约束检查。然后返回旧的约束检查值。

```
(set-static-constraint-checking <boolean-expression>)
```

若＜boolean-expression＞为 FALSE，则关闭静态约束检查，否则激活静态约束检查。然后返回旧的约束检查值。

```
(system <lexeme-expression>*)
```

把所有参数拼接成一个串，然后把它当作一个命令传送给操作系统执行。

事实

```
(assert <RHS-pattern>)
```

增加一个或多个事实到事实表中，并返回最后增加事实的地址。

```
(assert-string <string-expression>)
```

将串转换成事实并增加到事实表中，返回新增事实的地址。

```
(dependencies <fact-index-or-fact-address>)
```

列出所有指定事实可从中得到逻辑支持的部分匹配。

```
(dependents <fact-index-or-fact-address>)
```

列出所有可得到指定事实逻辑支持的事实。

```
(duplicate <fact-index-or-fact-address> <RHS-slot>*)
```

声明一个改变了一个或多个槽值的自定义模板事实的复制拷贝。

```
(facts [<module-name>]
       [<start-integer-expression>
          [<end-integer-expression>
             [<max-integer-expression>]]])
```

显示事实表中的所有事实。如果指定了<module-name>，则显示对指定模块可见的事实，否则，显示对当前模块可见的事实。事实表中索引号在<start-integer-expression>之前或<end-integer-expression>之后的事实不显示。如果指定了<max-integer-expression>，则大于该值的事实不显示。

```
(fact-existp <fact-address-expression>)
```

如果事实索引或事实地址参数指定的事实存在，则返回 TRUE，否则返回 FALSE。

```
(fact-index <fact-address-expression>)
```

返回和事实地址相关联的事实索引。

```
(fact-relation <fact-address-expression>)
```

返回和事实相关联的自定义模板（关系）名。

```
(fact-slot-names <fact-address-expression>)
```

返回和事实相关联的槽名。

```
(fact-slot-value <fact-address-expression>
                 <slot-name>)
```

返回指定事实的指定槽值。

```
(get-fact-duplication)
```

返回事实复制开关的当前值。

```
(get-fact-list [<module-name>])
```

返回指定模块（如果没有指定<module-name>，则为当前模块）的所有事实列表。

```
(load-facts <file-name>)
```

调入文件<file-name>中当前模块的事实。若成功返回 TRUE，否则返回 FALSE。

```
(modify <fact-index-or-fact-address> <RHS-slot>*)
```

改变自定义模板事实中的一个或多个槽值。

```
(retract <fact-index-or-fact-address>+)
```

从事实表中删除一个或多个事实。

```
(save-facts <file-name>
            [visible | local <deftemplate-names>*])
```

保存指定的事实到文件<file-name>中，若成功返回 TRUE，否则返回 FALSE。

```
(set-fact-duplication <boolean-expression>)
```

如果<boolean-expression>为 FALSE，则允许相同的事实增加到事实表中，否则不允许添加重复的事实到事实表中。函数返回事实复制开关的旧值。

实例

```
(any-instancep <instance-set-template> <query>)
<instance-set-template> ::=
                              (<instance-set-member-template>+)
<instance-set-member-template ::=
         (<single-field-variable>    <class-name-expression>+)
<query> ::= <boolean-expression>
```

如果有实例集满足指定的查询则返回 TRUE，否则 FALSE。

```
(class <object-expression>)
```

返回参数所表示的类名。

```
(delayed-do-for-all-instances <instance-set-template>
                              <query> <expression>*)
```

首先决定所有满足指定查询的实例集合，然后为前面每个实例集合计算指定表达式。

```
(delete-instance)
```

在消息处理程序内部调用时删除活动实例。

```
(direct-slot-delete$ <mv-slot-name>
                     <range-begin> <range-end>)
```

删除消息处理程序内活动实例的多字段槽值中指定范围的字段。

```
(direct-slot-insert$ <mv-slot-name> <index>
                     <expression>+)
```

插入一个或多个值到消息处理程序内活动实例的多字段槽值中。

```
(direct-slot-replace$ <mv-slot-name> <range-begin>
                      <range-end> <expression>+)
```

替换消息处理程序内活动实例的多字段槽值中指定范围的字段。

```
(do-for-instance <instance-set-template> <query>
                 <expression>*)
```

为满足指定查询的第一个实例集合计算指定表达式。

```
(do-for-all-instances <instance-set-template> <query>
                      <expression>*)
```

为满足指定查询的所有实例集合计算指定表达式。

```
(dynamic-get <slot-name-expression>)
```

返回活动实例的指定槽值。

```
(dynamic-put <slot-name-expression> <expression>*)
```

设置指定活动实例槽的值。

```
(find-instance <instance-set-template> <query>)
```

返回包含满足指定查询的第一个实例集合的多字段值。

```
(find-all-instances <instance-set-template> <query>)
```

返回包含满足指定查询的所有实例集合的多字段值。

```
(init-slots)
```

实现初始消息处理程序附加到 USER 类。这个函数只有在定义了初始消息处理程序使得与 USER 附加的另一个消息处理程序不会被调用时才直接调用。

```
(instance-address <instance-expression>)
```

把实例转化为实例地址。

```
(instance-addressp <expression>)
```

如果参数是实例地址，则返回 TRUE，否则，返回 FALSE。

```
(instance-existp <instance-expression>)
```

如果指定实例存在，则返回 TRUE，否则，返回 FALSE。

```
(instance-name <instance-expression>)
```

把实例地址转化为实例名。

```
(instance-namep <expression>)
```

如果参数是实例名，则返回 TRUE，否则，返回 FALSE。

```
(instance-name-to-symbol <instance-name-expression>)
```

把实例名转化为符号。

```
(instancep <expression>)
```

如果参数是实例名或实例地址，则返回 TRUE，否则，返回 FALSE。

```
(instances [<module-name> [<class-name> [inherit]]])
```

如果参数是实例名或实例地址，则返回 TRUE，否则，返回 FALSE。

```
(load-instances <file-name>)
```

从指定文件中调入实例。

```
(make-instance [<instance-name-expression>]
    of <class-name-expression> <slot-override>*)
<slot-override> ::= (<slot-name-expression>
                     <expression>)
```

用指定槽值创建并初始化实例。

```
(ppinstance )
```

在消息处理程序体中调用时，打印出活动实例的槽。

```
(restore-instances <file-name>)
```

从指定文件中调入实例。

```
(save-instances <file-name>)
```

把实例保存到指定文件。

```
(send <object-expression> <message-name-expression>
      <expression>*)
```

用指定参数发送消息给指定对象。

```
(slot-delete$ <instance-expression> <mv-slot-name>
              <range-begin> <range-end>)
```

在多字段槽值中删除指定范围的字段。

```
(slot-insert$ <instance-expression> <mv-slot-name>
              <index> <expression>+)
```

在多字段槽值中插入一个或多个值。

```
(slot-replace$ <instance-expression> <mv-slot-name>
               <range-begin> <range-end>
               <expression>+)
```

在多字段槽值中替换指定范围的字段。

```
(symbol-to-instance-name <symbol-expression>)
```

把符号转化为实例名。

```
(unmake-instance <instance-expression> | *)
```

通过发送删除消息来删除指定实例（如果指定 *，则为所有实例）。

I/O 函数

```
(close [<logical-name>])
```

关闭与逻辑名＜logical-name＞相关联的文件（如果没有指定，则指所有文件），如果成功返回 TRUE，否则返回 FALSE。

```
(format <logical-name> <string-expression>
        <expression>*)
```

对＜expression＞*求值并按照＜string-expression＞规定的格式输出到具有逻辑名＜logical-name＞的文件中。参见第 8. 12 节的格式化标志细节。

```
(open <file-name> <logical-name> [<mode>])
```

按照＜mode＞所指明的方式（r，w，r+或 a）打开文件＜file-name＞，并赋予一个逻辑名 ＜logical-name＞，若成功返回 TRUE，否则返回 FALSE。

```
(printout <logical-name> <expression>*)
```

对＜expression＞*求值并不加格式地输出到具有逻辑名＜logical-name＞的文件中。

```
(read [<logical-name>])
```

从具有逻辑名＜logical-name＞的文件中读取一个字段（如果无参数的话则从标准输入设备中输入），若成功则返回读取字段，若无输入则返回 EOF。

```
(readline [<logical-name>])
```

从具有逻辑名＜logical-name＞的文件中读取一整行（如果无参数的话则从标准输入设备中输入），若成功则返回读取串，若无输入则返回 EOF。

```
(remove <file-name>)
```

删除文件＜file-name＞。

```
(rename <old-file-name> <new-file-name>)
```

重命名文件＜old-file-name＞ 为＜new-file-name＞。

内存函数

```
(conserve-mem  on | off)
```

打开或关闭保存、漂亮打印命令对信息存储区的使用。

```
(mem-used)
```

返回 CLIPS 从操作系统所请求到的内存字节数。

```
(mem-requests)
```

返回 CLIPS 向操作系统请求内存的次数。

```
(release-mem)
```

把 CLIPS 占有的空余内存还给操作系统。返回值为归还的内存数。

杂项

```
(funcall <function-name-expression> <expression>*)
```

从参数构造一个函数调用，然后评估函数调用。

```
(gensym)
```

返回一个具有形式 genX 的序列符号，其中 X 为一个整数。

```
(gensym*)
```

返回一个具有形式 genX 的序列符号，其中 X 为一个整数。与 gensym 不同，gensym* 产生一个 CLIPS 环境中当前不存在的符号。

```
(get-function-restrictions <function-name-expression>)
```

返回与 CLIPS 或用户自定义函数关联的限制字符串。

```
(length <lexeme-or-multifield-expression>)
```

返回一个多字段值的字段整数个数，或者一个字符串或符号的长度。

```
(random [<start-integer-expression>
         <end-integer-expression>])
```

返回一个随机整数（如有参数，则其值介于指定的起始和结尾整数值之间）。

```
(seed <integer-expression>)
```

为随机函数 random 设置种子值。

```
(setgen <integer-expression>)
```

为函数 gensym 和 gensym* 设置序列索引。

```
(sort <comparison-function-name> <expression>*)
```

对＜expression＞* 指定的字段列表排序，使用 comparison 函数决定是否两个字段需要交换。

```
(time)
```

返回一个浮点数值表示从系统参考时间起所经过的秒数。

```
(timer <expression>*)
```

返回求值一串表达式所花费的秒数。

多字段

```
(create$ <expression>*)
```

把 0 个或多个表达式拼接在一起形成一个多字段值。

```
(delete$ <multifield-expression>
         <begin-integer-expression>
         <end-integer-expression>)
```

从＜multifield-expression＞中删去 ＜begin-integer-expression＞到＜end-integer-expression＞）之间的所有字段并返回所得结果。

```
(delete-member$ <multifield-expression> <expression>+)
```

删除多字段值中的指定值，并返回修改后的多字段值。

```
(explode$ <string-expression>)
```

根据包含于串中的字段创建一个多字段值并返回。

```
(first$ <multifield-expression>)
```

返回<multifield-expression>的第一个字段。

```
(implode$ <multifield-expression>)
```

返回一个包含多字段值中字段的串。

```
(insert$ <multifield-expression>
         <integer-expression>
         <single-or-multifield-expression>+)
```

在<multifield-expression>的第<integer-expression>个值之前插入所有的<single-or-multifield-expression>。

```
(length$ <multifield-expression>)
```

以多字段值形式返回中字段的个数。

```
(member$ <single-field-expression>
         <multifield-expression>)
```

返回第一个参数在第二个参数中的位置，如果第一个参数中不包含在第二个参数中，则返回FALSE。

```
(nth$ <integer-expression> <multifield-expression>)
```

返回<multifield-expression>中的第<integer-expression> 个字段。

```
(replace$ <multifield-expression>
          <begin-integer-expression>
          <end-integer-expression>
          <single-or-multifield-expression>+)
```

把<multifield-expression>中从<begin-integer-expression>到<end-integer-expression>之间的字段替换为<single-or-multifield-expression>并返回结果。

```
(replace-member$ <multifield-expression>
                 <substitute-expression>
                 <search -expression>+)
```

替换包含在多字段值内的指定值并返回修改后的多字段值。

```
(rest$ <multifield-expression>)
```

返回一个去掉<multifield-expression>中第一个字段后的多字段值。

```
(subseq$ <multifield-expression>
         <begin-integer-expression>
         <end-integer-expression>)
```

从<multifield-expression>中抽取<begin-integer-expression>到<end-integer-expression>）之间的字段形成一个多字段值并返回。

```
(subsetp <expression>)
```

如果第一个参数是第二个参数的一个子集，则返回 TRUE，否则返回 FALSE。

谓词函数

```
(and <expression>+)
```

如果每个参数均为真，返回 TRUE，否则返回 FALSE。

```
(eq <expression> <expression>+)
```

如果第一个参数与其余参数的类型和值都相等，则返回 TRUE，否则返回 FALSE。

```
(evenp <expression>)
```

如果<expression>之值为偶数，则返回 TRUE，否则返回 FALSE。

```
(floatp <expression>)
```

如果<expression>之值为浮点数，则返回 TRUE，否则返回 FALSE。

```
(integerp <expression>)
```

如果<expression>之值为整数，则返回 TRUE，否则返回 FALSE。

```
(lexemep <expression>)
```

如果<expression>为串或符号，则返回 TRUE，否则返回 FALSE。

```
(multifieldp <expression>)
```

如果<expression>是一个多字段值，则返回 TRUE，否则返回 FALSE。

```
(neq <expression> <expression>+)
```

如果第一个参数与其余参数的类型和值都不相等，则返回 TRUE，否则返回 FALSE。

```
(not <expression>)
```

如果<expression>之值为 FALSE，则返回 TRUE，否则返回 FALSE。

```
(numberp <expression>)
```

如果<expression>之值为浮点数或整数，则返回 TRUE，否则返回 FALSE。

```
(oddp <expression>)
```

如果<expression>之值为奇数，则返回 TRUE，否则返回 FALSE。

```
(or <expression>+)
```

如果任一参数为 TRUE，返回 TRUE，否则返回 FALSE。

```
(pointerp <expression>)
```

如果是外部地址则返回 TRUE，否则返回 FALSE。

```
(stringp <expression>)
```

如果<expression>为串，则返回 TRUE，否则返回 FALSE。

```
(symbolp <expression>)
```

如果<expression>为符号，则返回 TRUE，否则返回 FALSE。

```
(= <numeric-expression> <numeric-expression>+)
```

如果第一个参数与其余参数的数值相等，则返回 TRUE，否则返回 FALSE。

```
(<> <numeric-expression> <numeric-expression>+)
```

如果第一个参数与其余参数的数值不相等，则返回 TRUE，否则返回 FALSE。

```
(> <numeric-expression> <numeric-expression>+)
```

如果所有参数满足：第 n－1 个参数大于第 n 个参数，则返回 TRUE，否则返回 FALSE。

```
(>= <numeric-expression> <numeric-expression>+)
```

如果所有参数满足：第 n－1 个参数大于或等于第 n 个参数，则返回 TRUE，否则返回 FALSE。

```
(< <numeric-expression> <numeric-expression>+)
```

如果所有参数满足：第 n－1 个参数小于第 n 个参数，则返回 TRUE，否则返回 FALSE。

```
(<= <numeric-expression> <numeric-expression>+)
```

如果所有参数满足：第 n－1 个参数小于或等于第 n 个参数，则返回 TRUE，否则返回 FALSE。

过程

```
(bind <variable> <value>)
```

约束变量为指定值。

```
(break)
```

终止直接包含它的 while，loop-for-count，progn $，do-for-instance，do-for-all-instances 或 delayed-do-for-all-instances 函数的执行。

```
(if <predicate-expression> then <expression>+
                          [else <expression>+])
```

如果＜predicate-expression＞为 FALSE，则对 else 后面的表达式求值，否则对 then 后面的表达式求值。

```
(loop-for-count <range-spec> [do] <expression>*)
<range-spec> ::= <end-index> |
                 (<loop-variable> <end-index>) |
                 (<loop-variable> <start-index>
                                  <end-index>)
```

对＜expression＞*求值＜range-spec＞指定的次数。如果＜start-index＞没有值，则默认为 1。如果＜start-index＞大于＜end-index＞，那么循环体不会被执行。如果指定了＜loop-variable＞，则存放当前所迭代的次数。

```
(progn <expression>*)
```

求值所有参数，返回最后参数的求值结果。

```
(progn$ <list-spec> <expression>*)
<list-spec> ::= <multifield-expression> |
                (<list-variable>
                 <multifield-expression>)
```

为每个包含在＜list-spec＞中的字段计算＜expression＞*。如果指定了＜list-variable＞，则存放当前迭代的字段值。而且，当前迭代的整数索引值存放在变量＜list-variable＞-index 中。

```
(return [<expression>])
```

终止正在执行的自定义函数、类属函数方法、消息处理程序或自定义规则的 RHS。如果没有参数，则没有返回值。但如果包含参数，则其求值结果将作为自定义函数、方法、消息处理程序的返回值。

```
(switch <test-expression> <case-statement>*
                          [<default-statement>])

<case-statement> ::=
   (case <comparison-expression> then <expression>*)
<default-statement> ::= (default <expression>*)
```

计算＜test-expression＞并按次序与每个 case 的＜comparison-expression＞进行比较。如果匹配，求值这个 case 的＜expression＞*并返回。如果都不匹配，并且有＜default-statement＞，则求值＜default-statement＞下的＜expression＞*并返回。

```
(while <predicate-expression> [do] <expression>*)
```

不停对＜expression＞*求值直到＜predicate-expression＞为 FALSE。

profile 函数

```
(get-profile-percent-threshold )
```

返回 profile 百分比阈值的当前值。

```
(profile constructs | user-functions | off)
```

开启/关闭结构和用户函数的 profile。

```
(profile-info)
```

显示当前从结构或用户函数收集到的 profile 信息。

```
(profile-reset)
```

重置当前从结构或用户函数收集到的 profile 信息。

```
(set-profile-percent-threshold
            <number-in-range-of-0-to-100>)
```

设置由 profile-info 命令显示的用于执行结构或用户函数的最小时间分值。

序列扩展

```
(expand$ <multifield-expression>*)
```

当在函数调用内部使用时，扩展其参数作为函数的独立参数。$ 操作符是 expand $ 函数调用的简写形式。

```
(get-sequence-operator-recognition)
```

返回序列操作符识别开关的当前值。

```
(aet-sequence-operator-recognition
             <boolean-expression>)
```

设置序列操作符识别开关。

字符串

```
(build <string-or-symbol-expression>)
```

求值一个串就好像它是在命令行中输入的一样。只允许求值结构。

```
(check-syntax <string-expression>)
```

对一个结构或函数调用的文本表示进行语法和语义错误检查。如果没有错误或警告则返回 FALSE。

```
(eval <string-or-symbol-expression>)
```

求值一个串就好像它是在命令行中输入的一样。只允许求值函数。

```
(lowcase <string-or-symbol-expression>)
```

将参数中的所有大写字母换成小写字母。

```
(str-cat <expression>*)
```

将所有参数拼接成一个串。

```
(str-compare <string-or-symbol-expression>
             <string-or-symbol-expression>)
```

如果两个参数相等，则返回 0，如第一个参数的词典序比第二个参数大则返回正整数，如小则返回负整数。

```
(str-index <lexeme-expression> <lexeme-expression>)
```

如果第一个参数是第二个参数的一个子串，则返回其在第二个参数中的位置，否则返回 FALSE。

```
(str-length <string-or-symbol-expression>)
```

返回串的长度。

```
(string-to-field <string-or-symbol-expression>)
```

转化字符串为字段。

```
(sub-string <begin-integer-expression>
            <end-integer-expression>
            <string-expression>)
```

返回＜string-expression＞中从＜begin-integer-expression＞到＜end-integer-expression＞之间的子串。

```
(sym-cat <expression>*)
```

把所有参数拼接成一个符号。

```
(upcase <string-or-symbol-expression>)
```

将参数中所有的小写字母换成大写字母。

文本处理

```
(fetch  <file-name>)
```

把文件调入到内部查询表。

```
(print-region <logical-name> <lookup-file>
              <topic-field>*)
```

在特定的已经调入到查询表的文件中查找指定的条目，并打印条目内容到指定逻辑名的文件中。

```
(toss <file-name>)
```

从内部查询表中卸载指定的文件。

三角函数

```
(acos <numeric-expression>)
```

返回参数的反余弦值（弧度）。

```
(acosh <numeric-expression>)
```

返回参数的双曲反余弦值（弧度）。

```
(acot <numeric-expression>)
```

返回参数的反余切值（弧度）。

```
(acoth <numeric-expression>)
```

返回参数的双曲反余切值（弧度）。

```
(acsc <numeric-expression>)
```

返回参数的反余割值（弧度）。

```
(acsch <numeric-expression>)
```

返回参数的双曲反余割值（弧度）。

```
(asec <numeric-expression>)
```

返回参数的反正割值（弧度）。

```
(asech <numeric-expression>)
```

返回参数的双曲反正割值（弧度）。

```
(asin <numeric-expression>)
```

返回参数的反正弦值（弧度）。

```
(asinh <numeric-expression>)
```

返回参数的双曲反正弦值（弧度）。

```
(atan <numeric-expression>)
```

返回参数的反正切值（弧度）。

```
(atanh <numeric-expression>)
```

返回参数的双曲反正切值（弧度）。

```
(cos <numeric-expression>)
```

返回参数的余弦值（弧度）。

```
(cosh <numeric-expression>)
```

返回参数的双曲余弦值（弧度）。

```
(cot <numeric-expression>)
```

返回参数的余切值（弧度）。

```
(coth <numeric-expression>)
```

返回参数的双曲余切值（弧度）。

```
(csc <numeric-expression>)
```

返回参数的余割值（弧度）。

```
(csch <numeric-expression>)
```

返回参数的双曲余割值（弧度）。

```
(sec <numeric-expression>)
```

返回参数的正割值（弧度）。

```
(sech <numeric-expression>)
```

返回参数的双曲正割值（弧度）。

```
(sin <numeric-expression>)
```

返回参数的正弦值（弧度）。

```
(sinh <numeric-expression>)
```

返回参数的双曲正弦值（弧度）。

```
(tan <numeric-expression>)
```

返回参数的正切值（弧度）。

```
(tanh <numeric-expression>)
```

返回参数的双曲正切值（弧度）。

附录 F　CLIPS BNF 范式

CLIPS 程序

```
<CLIPS-program> ::= <construct>*

<construct>     ::= <deffacts-construct> |
                    <deftemplate-construct> |
                    <defglobal-construct> |
                    <defrule-construct> |
                    <deffunction-construct> |
                    <defgeneric-construct> |
                    <defmethod-construct> |
                    <defclass-construct> |
                    <definstance-construct> |
                    <defmessage-handler-construct> |
                    <defmodule-construct>
```

自定义事实结构

```
<deffacts-construct> ::= (deffacts <name> [<comment>]
                           <RHS-pattern>*)
```

自定义模板结构

```
<deftemplate-construct>  ::= (deftemplate <name>
                               [<comment>]
                               <slot-definition>*)

<slot-definition>
   ::= <single-slot-definition> |
       <multislot-definition>

<single-slot-definition> ::= (slot <name>
                                 <slot-attribute>*)

<multislot-definition>   ::= (multislot <name>
                                 <template-attribute>*)

<template-attribute>     ::= <default-attribute> |
                             <constraint-attribute>
<default-attribute>
   ::= (default ?DERIVE | ?NONE | <expression>*) |
       (default-dynamic <expression>*)
```

事实规范

```
<RHS-pattern>            ::= <ordered-RHS-pattern> |
                             <template-RHS-pattern>

<ordered-RHS-pattern>    ::= (<symbol> <RHS-field>+)

<template-RHS-pattern>   ::= (<deftemplate-name>
                              <RHS-slot>*)

<RHS-slot>               ::= <single-field-RHS-slot> |
                             <multifield-RHS-slot>

<single-field-RHS-slot>  ::= (<slot-name> <RHS-field>)

<multifield-RHS-slot>    ::= (<slot-name> <RHS-field>*)

<RHS-field>              ::= <variable> |
                             <constant> |
                             <function-call>
```

自定义规则结构

```
<defrule-construct>   ::= (defrule <name> [<comment>]
                             [<declaration>]
                             <conditional-element>*
                             =>
                             <expression>*)

<declaration>         ::= (declare <rule-property>+)

<rule-property>
   ::= (salience <integer-expression>) |
       (auto-focus <boolean-symbol>)

<boolean-symbol>      ::= TRUE | FALSE

<conditional-element> ::= <pattern-CE> |
                          <assigned-pattern-CE> |
                          <not-CE> |
                          <and-CE> |
                          <or-CE> |
                          <logical-CE> |
                          <test-CE> |
                          <exists-CE> |
                          <forall-CE>

<pattern-CE>          ::= <ordered-pattern-CE> |
                          <template-pattern-CE> |
                          <object-pattern-CE)

<assigned-pattern-CE>
   ::= <single-field-variable> <- <pattern-CE>

<not-CE>          ::= (not <conditional-element>)

<and-CE>          ::= (and <conditional-element>+)
<or-CE>           ::= (or <conditional-element>+)

<logical-CE>      ::= (logical <conditional-element>+)

<test-CE>         ::= (test <function-call>)

<exists-CE>       ::= (exists <conditional-element>+)

<forall-CE>       ::= (forall <conditional-element>
                              <conditional-element>+)

<ordered-pattern-CE>  ::= (<symbol> <constraint>+)

<template-pattern-CE>
   ::= (<deftemplate-name <LHS-slot>*)

<object-pattern-CE>
   ::= (object <attribute-constraint>*)

<attribute-constraint>  ::= (is-a <constraint>) |
                            (name <constraint>) |
                            (<slot-name> <constraint>*)

<LHS-slot>              ::= <single-field-LHS-slot> |
                            <multifield-LHS-slot>

<single-field-LHS-slot> ::= (<slot-name> <constraint>)

<multifield-LHS-slot> ::= (<slot-name> <constraint>*)

<constraint>       ::= ? | $? | <connected-constraint>

<connected-constraint>
   ::= <single-constraint> |
       <single-constraint> & <connected-constraint> |
       <single-constraint> | <connected-constraint>

<single-constraint>   ::= <term> | ~<term>

<term>                ::= <constant> |
                          <single-field-variable> |
                          <multifield-variable> |
                          :<function-call> |
                          =<function-call>
```

自定义全局变量结构

```
<defglobal-construct>
   ::= (defglobal [<defmodule-name>]
         <global-assignment>*)

<global-assignment>
   ::= <global-variable> = <expression>

<global-variable>       ::= ?*<symbol>*
```

自定义函数结构

```
<deffunction-construct>
   ::= (deffunction <name> [<comment>]
         (<regular-parameter>* [<wildcard-parameter>])
         <expression>*)
<regular-parameter>  ::= <single-field-variable>

<wildcard-parameter> ::= <multifield-variable>
```

自定义类属结构

```
<defgeneric-construct>
   ::= (defgeneric <name> [<comment>])
```

自定义方法结构

```
<defmethod-construct>
   ::= (defmethod <name> [<index>] [<comment>]
         (<parameter-restriction>*
          [<wildcard-parameter-restriction>])
         <expression>*)

<parameter-restriction>
   ::=  <single-field-variable> |
        (<single-field-variable> <type>* [<query>])

<wildcard-parameter-restriction>
   ::= <multifield-variable> |
       (<multifield-variable> <type>* [<query>])

<type>         ::= <class-name>

<query>        ::= <global-variable> | <function-call>
```

自定义类结构

```
<defclass-construct> ::= (defclass <name> [<comment>]
                           (is-a <superclass-name>+)
                           [<role>]
                           [<pattern-match-role>]
                           <slot>*
                           <handler-documentation>*)
<role>  ::= (role concrete | abstract)

<pattern-match-role>
   ::= (pattern-match reactive | non-reactive)

<slot>  ::= (slot <name> <facet>*) |
            (single-slot <name> <facet>*) |
            (multislot <name> <facet>*)

<facet> ::=  <default-facet> | <storage-facet> |
             <access-facet> | <propagation-facet> |
             <source-facet> | <pattern-match-facet> |
             <visibility-facet> |
             <create-accessor-facet> |
```

```
                <override-message-facet> |
                <constraint-attribute>

<default-facet>
   ::= (default ?DERIVE | ?NONE | <expression>*) |
       (default-dynamic <expression>*)


<storage-facet> ::= (storage local | shared)

<access-facet>  ::= (access read-write |
                            read-only |
                            initialize-only)

<propagation-facet>
   ::= (propagation inherit | no-inherit)

<source-facet> ::= (source exclusive | composite)

<pattern-match-facet>
   ::= (pattern-match reactive | non-reactive)

<visibility-facet> ::= (visibility private | public)

<create-accessor-facet>
   ::= (create-accessor ?NONE | read |
                        write | read-write)

<override-message-facet>
   ::= (override-message ?DEFAULT | <message-name>)

<handler-documentation>
   ::= (message-handler <name> [<handler-type>])

<handler-type> ::= primary | around | before | after
```

自定义消息处理程序结构

```
<defmessage-handler-construct>
   ::= (defmessage-handler <class-name>
          <message-name> [<handler-type>] [<comment>]
          (<parameter>* [<wildcard-parameter>])
          <action>*)

<handler-type>    ::= around | before | primary | after

<parameter>       ::= <single-field-variable>

<wildcard-parameter> ::= <multifield-variable>
```

自定义实例结构

```
<definstances-construct> ::= (definstances <name>
                                [active] [<comment>]
                                <instance-template>*)

<instance-template>    ::= (<instance-definition>)

<instance-definition>
   ::= <instance-name-expression> of
          <class-name-expression> <slot-override>*

<slot-override>
   ::= (<slot-name-expression> <expression>*)
```

自定义模板结构

```
<defmodule-construct>
   ::= (defmodule <name> [<comment>])
          <port-specification>*)

<port-specification>
   ::= (export <port-item>) |
       (import <module-name> <port-item>)
```

```
<port-item>     ::= ?ALL |
                    ?NONE |
                    <port-construct> ?ALL |
                    <port-construct> ?NONE |
                    <port-construct> <construct-name>+

<port-construct> ::= deftemplate | defclass |
                     defglobal | deffunction |
                     defgeneric
```

约束属性

```
<constraint-attribute>
                    ::= <type-attribute> |
                        <allowed-constant-attribute> |
                        <range-attribute> |
                        <cardinality-attribute>

<type-attribute>    ::= (type <type-specification>)

<type-specification> ::= <allowed-type>+ | ?VARIABLE

<allowed-type>       ::= SYMBOL | STRING | LEXEME |
                         INTEGER | FLOAT | NUMBER |
                         INSTANCE-NAME |
                         INSTANCE-ADDRESS |
                         INSTANCE | EXTERNAL-ADDRESS |
                         FACT-ADDRESS

<allowed-constant-attribute>
   ::= (allowed-symbols <symbol-list>) |
       (allowed-strings <string-list>) |
       (allowed-lexemes <lexeme-list> |
       (allowed-integers <integer-list>) |
       (allowed-floats <float-list>) |
       (allowed-numbers <number-list>) |
       (allowed-instance-names <instance-list>) |
       (allowed-values <value-list>)

<symbol-list>      ::= <symbol>+ | ?VARIABLE

<string-list>      ::= <string>+ | ?VARIABLE

<lexeme-list>      ::= <lexeme>+ | ?VARIABLE

<integer-list>     ::= <integer>+ | ?VARIABLE

<float-list>       ::= <float>+ | ?VARIABLE

<number-list>      ::= <number>+ | ?VARIABLE
<instance-name-list> ::= <instance-name>+ | ?VARIABLE

<value-list>       ::= <constant>+ | ?VARIABLE

<range-attribute> ::= (range <range-specification>
                             <range-specification>)

<range-specification> ::= <number> | ?VARIABLE

<cardinality-attribute>
   ::= (cardinality <cardinality-specification>
                    <cardinality-specification>)

<cardinality-specification> ::= <integer> | ?VARIABLE
```

变量与表达式

```
<single-field-variable> ::= ?<variable-symbol>

<multifield-variable>   ::= $?<variable-symbol>

<global-variable>       ::= ?*<symbol>*

<variable>              ::= <single-field-variable> |
                            <multifield-variable> |
                            <global-variable>
```

```
<function-call>
   ::= (<function-name> <expression>*) |
       (<special-function-name>
        <special-function-arguments>)

<special-function-name>
   ::= The name of a function such as assert or
       make-instance that does not parse its
       arguments in the default manner that standard
       functions use.

<special-function-arguments>
   ::= Argument parsing for special functions varies.
       Refer to the documentation of each special
       function for its valid syntax. For example,
       in the function call (assert (value 3)), the
       argument (value 3) is not a call to the value
       function with an argument of 3, it is the
       fact to be asserted.

<expression>           ::= <constant> |
                           <variable> |
                           <function-call>
```

数据类型

<symbol> ::= 第 7.4 节中所规定的合法符号

<string> ::= 第 7.4 节中所规定的合法字符串

<float> ::= 第 7.4 节中所规定的合法浮点数

<integer> ::= 第 7.4 节中所规定的合法整数

<instance-name> ::= 第 7.4 节中所规定的合法实例名

<number> ::= <float>|<integer>

<lexeme> ::= <symbol>|<string>

<constant> ::= <number>|<lexeme>

<comment> ::= <string>

<variable-symbol> ::= 一个以字母开头的 <symbol>

<function-name> ::= <symbol>

<name> ::= <symbol>

<...-name> ::= 一个<symbol>，其中省略号表示这个符号代表什么，例如，<deftemplate-name>是一个表示自定义模板的名字符号。

附录G 软件资源

本附录的主要目的是为本书每一章讨论的论题提供当前的网站资源。通过布置学生调查软件并准备课堂报告和演示，这些资源也能用于课堂上以丰富教材的内容。传统的课程教学中使用教材，学生也可以通过自己学习这些知识来获得额外收获。

许多商业软件产品可以下载试用版本并带有例子演示产品的功能。最好让学生构造原始例子然后比较结果。例如，比较第1章中讨论的旅行售货者问题的求解效率，使用数据结构课程中的标准贪心算法、人工神经网络（并决定哪种类型最好）、遗传算法、蚁群进化算法和其他附录中所列出的方法。

特别的，比较人工智能算法与标准算法在问题规模从几个城市扩展到几百个城市将会非常有趣。把例子和真实世界联系起来，例如西南航空公司的航线，使得这种比较更有意义。可以很容易地从网上下载西南航空公司的所有航线城市及其经纬度作为数据集。

工作

除了许多一般的工作搜索引擎，有很多职业网站专门为具有人工智能和专家系统计算机背景的人们提供在防卫、医疗、商业、工业和视频游戏，这是今天计算机的5个最大增长领域的工作机会。这些工作用到了本书中所讨论的很多主题。下载各种演示软件和各章的资源例子可以得到更多的经验。

- 数据分析者（第1～12章）
- 系统设计者（第1～12章）
- 开发者（第1～12章）
- 知识领域专家（你学习本书不一定能成为知识领域专家，但你可以成为一个知识工程师，特别地，本书第6章给出了面试知识领域工程师的一些实践知识，它们也是知识工程师的基本任务）
- 搜索技术（第1、2章）
- 基于规则的系统（有限状态机，决策树，产生式系统）（第1～3、第6～12章）
- 博弈理论与博弈树（有限状态机，决策树的另一种说法，第1章）
- 人工生命与群技术（第1章）
- 规划技术（第2章）
- 模糊逻辑（第4、5章）
- 人工神经网络、遗传算法、信念网（第1、4章）

以下是一些提供工作机会的链接。在在线杂志例如PCAI. com或下面列表中其他杂志所登出的公司主页上还可以找到其他的机会。杂志期刊通常会在最后列出工作，并都可以从网上得到。学校图书馆具有这些杂志期刊的电子版本，你可以免费地从网上获取。工作机会经常发布在下面列出的新闻列表中。如果可能，可以进行一个有趣的项目，开发一个人工智能Agent，自动搜索发布的工作机会，向用户返回报告。

需要提醒你：如果你发布或送出你的简历，小心不要加入任何私人信息，例如社会安全保障号码等。这些都可能导致身份被盗窃。这是越来越严重的问题。事实上，无论工作机会多么诱人，都不要把可能用于身份盗窃的私人信息放在电子邮件中。

如果你没有符合所有的条件也不用担心。**智慧第五法则**表明如果你是惟一一个能够完成旅程的人，那么你就是赢家。这意味着你只需要符合最主要的要求，并愿意接受公司能提供的最高额薪水。当一个公司需要具有10年20种不同语言工作经验的人，他们不一定能负担得起这些人所要求的薪水，那么你可能就是最好的选择。找工作就像找伴侣。你马上或稍后能发现你娶的最好的人的缺点。

工作链接

http://www.aaai.org/Magazine/Jobs
http://www.mary-margaret.com/
http://www.blizzard.com/jobopp/
http://www.aktor-kt.com/jobs.htm
http://www.cdacindia.com/html/aai/aaidx.asp
http://www.lplizard.com/lounge/jobs_programmer.htm
http://corporate.infogrames.com/corp_hrmain.php?action=jobdetails&jobID=218&locationID=7
http://www.insomniacgames.com/html/about/jobs.html
http://www.alifemedical.com/careers.html
http://www.hirehealth.com/ci/servlet/com.ci.jobseeker.JobDetails;jsessionid=79FDF0992C80EB299B6B5778CBB6EB1D?JOB_ID=48179
http://www.shrinershq.org/cgi-bin/classifieds/classifieds.cgi
http://www.genesciences.com/DNAjobsNews/12June04.htm
http://www.business.com/search/rslt_default.asp?query=medical+field&bdct=&bdcf=&vt=all&type=jobs&search=Next+Search
http://www.aegiss.com/html/jobs.html
http://www.gamedev.net/directory/careers/
http://www.capcom.com/jobs/job.xpml?jobid=400016
http://www.gamerecruiter.com/
http://www.3drealms.com/gethired.html

在线百科全书

在线百科全书罗列了术语的简单解释，其中有一些还附加了额外的资源链接和例子。建议学生应该从本书内容中找出这些术语的详细解释和例子。

Wikipedia：一个极佳的大型内容开放的百科全书，具有多个语言的版本，提供了许多术语的清楚解释，并附有更进一步阅读的链接。http://en.wikipedia.org/wiki/Main_Page

Webopedia. 一个很好的通用在线计算机科学百科全书：http://www.webopedia.com/

Platonic Realms. 一个有利于学生的在线数学百科全书并带有其他资源。高度推荐：http://www.mathacademy.com/pr/

数学百科，来自 Wolfram，Mathematica 的制作者：http://mathworld.wolfram.com/

因特网哲学百科。很好的在线哲学和逻辑百科，主要是第 2～5 章的参考资料：http://www.iep.utm.edu/

哲学和逻辑的一般定义和解释。见在线百科：http://plato.stanford.edu/

令人感兴趣的在线演讲。包括了许多著名的演讲者通过微软多个大学研究实验室发表的各种主题的演讲：http://murl.microsoft.com/ContentMap.asp

Google. 对于失眠的人来说，最好阅读 google 宣称的最丰富的逻辑资源：http://www.uni-bonn.de/logic/world.html

新闻组

新闻组有很多发布内容，常见问题、资源、会议以及关于人工智能、专家系统和其他上千个主题的新闻。新闻组和杂志也会列出一些特殊领域的工作机会，例如专家系统和人工智能。能够寻找专家系统工作的网页是：http://mailgate.supereva.it/comp/comp.ai.shells/，在它的父新闻组可以找到人工智能的工作机会：http://mailgate.supereva.it/comp/comp.ai/。如果你想在其中注册，你需要懂得更多内容。

通常关于专家系统，例如 CLIPS 的问题如果发表在 comp.ai.shells 新闻组或者其他列在附录 D 中的 CLIPS 官方网站中可以得到回答。有 3 种方法去联络人工智能新闻组。第一种方法是从你的网络提供商得到你的新闻组服务器地址。在 IE 浏览器中你可以选择工具菜单，然后是邮件和新闻子菜单，然后选底部的阅读邮件子菜单。就会打开一个新的窗口显示一个链接：建立新闻组账户，然后随网络连接向导进行设置。

如果你的时间宝贵，可以到 google 的网站 http://www.google.com，点击你通过关键词所搜索到的红色新闻组链接，将会得到一系列常见的新闻组类型。点击 comp. 链接，你可能对其中一些新闻组感兴趣：

comp.job.* （1 组）

comp.jobs.* （6 组）

comp.jobsoffered

其中星号表示有更多的子组。其他特定语言的新闻组也会发布工作机会。注意小心保护可能引起身份盗窃的个人信息。一旦你在 comp. 集合中，你可以通过点击 comp.ai 浏览所有的人工智能新闻组。

第三种方法，输入：http://groups.google.com/groups? hl = en&lr = &ie = UTF - 8&group = comp.ai 到浏览窗口的地址栏中，你将得到：

comp.ai	comp.ai.jair.announce
comp.ai.alife	comp.ai.jair.papers
comp.ai.doc-analysis.misc	comp.ai.nat-lang
comp.ai.analysis.ocr	comp.ai.neural-nets
comp.ai.edu	comp.ai.nlang-know-rep
comp.ai.fuzzy	comp.ai.philospohy
comp.ai.games	comp.ai.shells
comp.ai.gentic	comp.ai.vision

注意，jair 是 Journal of AI Research，一个在线期刊。新闻组的好处在于你可以发布内容，回复问题。还有其他新闻组发布来自政府、州和某些城市的工作机会。输入关键词 job 可以找到这些内容。

开始某个特定主题研究，例如人工智能、神经网络、遗传算法等的一个最佳方法除了通览许多资源外，还可以是阅读常见问题列表（FAQs）。FAQ 是新手最好的起点，它由资深者所写，并且完全免费。尽管商业产品经常被提及，但它们以具体对象为讨论方式，不具备普遍性。你可以注册每一个看起来有用的新闻组，阅读它们的 FAQs。也可以采用更有效的方法，成千上万个新闻组的 FAQs 都被登记到一个可搜索页面 http://www.faqs.org/faqs/，可以通过输入关键词或按目录或者字母次序查询。而且，人工智能及其子目录的 FAQs 可以很方便地从 http://www.faqs.org/faqs/ai-faq/获得。在 http://www.faqs.org/faqs/by-newsgroup/comp 有大量关于计算机的 FAQs。

期刊和在线杂志

期刊和杂志都有电子版和文字版，提供最近的新闻和某一主题的文章，是寻找工作机会、会议主题、工作组、会议的好资源。期刊和杂志最主要的区别在于，期刊靠订户支持运作，而杂志通过广告获得主要经费。期刊为了强调其独立报道，长期以来采用这种策略。期刊文章要求被多个评阅人审阅，所以真正的期刊文章都会被审阅。

在杂志上很少看到作者在其所在公司发广告的杂志上发表文章。另一方面，期刊的文章很少出自期刊编者或成员。你需要特别注意带有以下类似脚注的文章："本研究部分得到 XYZ 的支持"，因为作者为了继续得到基金支持，他们会较详尽地介绍工作而不是空洞的介绍。XYZ 也会通过发表的文章和资助者得到的基金数量展示他们的品牌。

会议文章需要较仔细的阅读，因为这可能是作者为获取旅游经费，或者是学生为获得毕业学位或致谢 XYZ 而正在进行的工作。因此，本书不引用任何来自会议的文章。

AI Case-Based Reasoning Journal：http://www.ai-cbr.org/theindex.htm

BusinessWeek Online：http://www.businessweek.com. 尽管不是期刊，这个在线杂志提供了很好的搜索引擎做人工智能等关键词的搜索，经常提供关于商业、政府和军方的高技术新闻，因为这三者都花费了大量的财力。BusinessWeek 提供了计算机在商业应用的最新资讯。在这里你可以发现沃尔玛（Wal-Mart）正在使用数据挖掘技术应用于超过 3000 个仓库的数据并为每个商品提供销售指引，以及人工智能的广泛应用：http://www.businessweek.com/bw50/content/mar2003/a3826072.htm

American Association for Artificial Intelligence: http://www.aaai.org/
Expert Systems International Journal: http://www.blackwellpublishing. com/journal.asp?ref
Expert Systems Journal: http://www.aaai.org/AITopics/html/expert.html

Fuzzy Optimization and Decision Making：一个关于不确定性下模拟和计算的期刊，http://www.kluweronline.com/issn/1568-4539

IEEE Intelligent Systems Journal: http://www.computer.org/intelligent/

IEEE Transactions on Fuzzy Systems: http://www.ieee.org/portal/index.jsp?pageID=corp_level1&path=pubs/transactions&file=tfs.xml&xsl=generic.xsl

IEEE Transactions on Neural Networks: http://www.ieee.org/portal/index.jsp?pageID=corp_level1&path=pubs/transactions&file=tnn.xml&xsl=generic.xsl

Journal of Artificial Intelligence Research: http://www.cs.washington.edu/ research/jair/home.html

Neural Computation by the MIT Press: http://mitpress.mit.edu/catalog/ item/default.asp?sid=B395B969-A0A5-A566-AF7B2EDCE494&ttype=4&tid=31

Neural Computing&Applications by Springer-Verlag。输入 http://springerlink.metapress.com. 在这个页面点击 Browse Publication，然后跳到 N，点击 Neural Computing and Applications。

North American Fuzzy Information Processing Society。点击他们的在线资源链接可以得到很多资源：http://morden.csee.usf.edu/Nafipsf/

PCAI.com。一个优秀的提供很好搜索引擎的在线杂志。PCAI 创办于 20 世纪 80 年代，包括了大量的人工智能信息资源，链接到软件、文章和其他资源信息组，例如：C++、黑板技术、Client/Server、Dylan、创造思维、数据挖掘、Forth、分布计算、专家系统、LISP、模糊逻辑、一般 AI 站点、Logo、遗传算法 、OPS、智能 Agent、智能应用、Prolog、因特网、逻辑程序设计、Scheme、机器学习、模拟和仿真、Smalltalk、多媒体、自然语言处理、神经网络、面向对象开发、视觉、特征、识别、机器人、语音识别、虚拟现实，http://PCAI.com

Generation5.com。一个很全面的人工智能网站，提供人工智能各个方面的新闻和信息，包括游戏，以及很多下列人工智能一般领域的链接：航天与军事、Agent、人工智能、人工生命、生物测定学、约束满足规划、创造、分形、混沌、复杂性、非线性动力学、博弈、遗传算法、手写识别、家庭自动化、商业、图像分析/识别、基于知识的系统、LISP、自然语言处理、神经网络、神经科学、模式识别、个性化、哲学、Prolog、机器人、Furby、LEGO 头脑风暴、Scheme（编程语言）、语音识别与合成、VoiceXML：http://www.generation5.org/

SourceForget.Net。一个软件和其他资源的主要来源站点。作为最大的开源软件网站，具有超过 90 000个项目，包括并超过了本书讨论的所有主题：http://sourceforget.net/

AI@HOME。一个很好的人工智能在线网站，发布所有神经网络、进化算法，元学习和分布式计算的开源代码。元学习即用学习算法来创建新的学习算法，http://www.aiathome.com/

Dr Dobbs Portal AI Links。一个搜集来自 Dr.Dobbs 期刊——一个关许多编程语言的杂志——的大量文章、导读、链接和软件的网站：

http://www.ddj.com/documents/s=7730/ddj0212ai/0212ai001.html#egyptian_cu

Adaptive Behavior。一个关于动物和自治人工系统例如机器人的自适应行为研究的国际期刊。其主题包括感知与运动控制、学习、进化、动作选择与行为序列、刺激与反应、环境特征、群体与社会行为、定位、通信和信号：

http://www.isab.org/journal/

Artificial Life。一个人工智能研究以及商业应用的网站。除了电影工业每年产生 200 亿美元的利润，视频游戏产生的利润更多，人工智能还为军方提供很多应用。这个网站研究通过计算机、机器、分子和其他可选媒体来从科学、工程、哲学和社会等方面合成生命行为，研究生命的起源、自组织、生长与发展、进化与生态演变、动物与机器行为、社会组织、文化发展。人工生命是视频游戏、电影如 Lord of the Rings 系列、I，Robot 等的一个重要课题。人工生命形式要学会战斗，其中一些战斗场面涉及成千上万个人工智能生命形式。

对于传统计算机动画和人类电影导演来说，与成千上万个计算机动画角色合作是太过庞大的工作任务。所以人工智能生命形式被开发来学习战斗技巧以及复杂的战斗策略以便导演可以选择最有趣的内容。这种方法是传统的导演指导个别演员和动画内容的改革。在 Star Wars 2：Attack of the Clones 电

影中，电影制作者使用奖品来吸引群众，但现在，庞大的群众场景可使用人工生命形式（令人担忧的是电影 Matrix 可能会真正成为现实，除了电影里真实的人陷入虚假环境外。现在我们知道，没有哪个人是不可缺少的）。

http://www-mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=41

Computational Linguistics。讨论自然语言处理系统的设计和分析。人工智能、认知科学、语音、语言处理和表现的心理学：

http://www-mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=10

Computer-Mediated Communication Magazine。一个在线杂志提供最新的新闻，包括信息技术、知识管理、电子商务、人、事件、技术、公共政治、文化、实践、研究以及与网上人们交流与交互相关的应用。是当前事件和技术的很好汇总：http://www.december.com/cmc/mag/meta/

Decision Support Systems and Electronic Commerce。一个实现和评价决策支持系统在下面一些领域的技术期刊：人工智能、认知科学、计算机支持的协同工作、数据库管理、决策理论、经济、语言学、管理科学、数学建模、操作管理心理学、用户接口管理系统以及其他领域。http://www.elsevier.com/wps/find/journaldescription.cws_home/505540/description#description

Electronic Journals and Periodicals in Psychology。尽管不是期刊，但有一个很好的列表提供所有与心理相关的电子期刊、会议议程和其他定期内容的链接。一个最大的特点是使用电子邮件提醒与你感兴趣内容相关的新信息已经送出。人工智能领域从心理学吸取了很多精髓，千百万年来，人们就是使用其中某些好的（或坏的）技术来解决问题。如果你想深入研究人工智能，你需要学习心理学、生物学、神经科学、哲学和其他领域的知识来开拓你的视野。http://psych.hanover.edu/Krantz/journal.html

AI Events。尽管不是一个期刊，但它是一个非常有用的搜索数据库，可以查找近期人工智能会议、工作组、暑期培训以及类似事件。是一个规划你的暑假的好帮手。http://www.drc.ntu.edu.sg/users/mgeorg/enter.epl

Evolutionary Computation。一个从客观抽象到意识的计算系统期刊，侧重于进化算法（EA）、遗传算法、进化策略、进化编程、遗传编程、分类系统和其他从生物系统发展出来的自然计算技术。http://www-mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=25

International Journal of Human-Computer Studies/Knowledge Acquisition. A；为第 6 章提供了特别有用的资源。包括了广阔的主题：http://repgrid.com/IJHCS/

- intelligent user interfaces
- natural language interaction
- human factors of multimedia systems
- human and social factors of virtual reality
- human and social factors of the World Wide Web
- human and social factors of software engineering
- computer-supported collaborative work
- speech interaction
- graphic interaction
- knowledge acquisition
- knowledge-based systems
- hypertext and hypermedia
- user modeling
- empirical studies of user behavior
- the psychology of programming
- systems theory and foundations of human-computer interaction
- user interface management systems
- information and decision-support systems
- requirements engineering
- innovative designs and applications of interactive systems

Journal of Artificial Intelligence Research。一个很全面的关于人工智能所有主题的在线期刊。http://www-mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=12

Journal of Cognitive Neuroscience。讨论大脑行为交互、功能和大脑底层事件、神经科学、神经心理学、认知心理学、神经生物学、语言学、计算机科学和哲学。http://www-mitpress.mit.edu/catalog/item/default.asp?ttype=4&tid=12

Journal of Constructivist Psychology。构成主义也是教育以及人类学习的热门分支。http://www.tandf.co.uk/journals/titles/10720537.asp

Journal of Mind and Behavior. 一个试图联系意识与行为的理论期刊：http://kramer.ume.maine.edu/~

jmb/welcome.html

Noetica: A Cognitive Science Forum，一个在线的可参考期刊，包括认知科学、机器学习和许多与人工智能相关的主题：http://www.drc.ntu.edu.sg/users/mgeorg/enter.epl

PRESENCE: Teleoperators and Virtual Environments：一个吸引人的期刊，包括在遥控手术和军事方面的重要应用，例如使用无人驾驶航空工具（UAV）进行无人侦察。UAV 通常被军方使用，因为他们比有人驾驶飞机更加便宜、难以被发现、可以飞行更长时间。最重要的是，一个操作员可以控制多架 UAV 而不用承受一个有人驾驶飞机被打下的压力，如发生在 1962 年的著名的 Francis Gary Powers 事件。

预期在 2010 年，智能 UAV，即 X-45 无人驾驶战斗机（UCAV）将可以投入军事使用。http://www.fas.org/man/dod-101/sys/ac/ucav.htm。它是一个不需要人操作的可飞行到预定目的地的轰炸机/监视机。花费上亿美元研究的智能武器系统正在开发中，你可以看一些电视，如 History 频道中的 Mail Call，Discovery 频道中的 Tactics to Practice，G4TechTV 中的 Future Fighting Machines，从中可以知道这些钱是如何花在基于人工智能的军方系统中。

真正令人迷惑的是这些 X-45 看起来类似在电影 Terminator 系列中由 AI 程序空间的罪犯所控制的人工智能飞机（在电影中，人工智能部队试图从人类手中夺取世界）。同样的主题也出现在 2004 年的电影 I，Robot 中。再往前可追溯到 20 世纪 70 年代经典的电影 Colossus：The Forbin Project，其中美国和俄国的超级计算机联合起来从毁灭的世界中拯救人类。

Public E-print Archive。由 Stevan Harnad 发表的认知心理学论文包括了很多令人感兴趣的主题，例如归纳、认知、机器学习、机器人和其他内容。是可读性很高的文章：http://www.ecs.soton.ac.uk/~harnad/genpub.html

PSYCHE。一个关于意识研究的跨学科期刊，提供大量有趣的文章 http://psyche.cs.monash.edu.au/

Shufflebrain。探索大脑这个物理对象如何存储意识以及很多如下面的有趣问题。http://www.indiana.edu/~pietsch/home.html

- Shuffle Brain——一个受损的大脑能否记忆？
- The Beast's IQ——智能的隐藏面
- Hologramic Mind——什么是真正存在的？以什么方式？
- Microminds——细菌是否会思考？会伤心？会快乐？
- 人类大脑思考？还是人类思考？
- Brain Swapping——洗脑
- Hololgic——一个关于自然全息像的 Asa Zook 对话
- Split Human Brain——自身分裂体
- Musical Brain——温和的一面
- Optics of Memory——一个非常好的工程

The Journal of Computer-Mediated Communication。这个站点不仅仅包括人工智能，还涵盖了许多可能导致人工智能新发展的领域，因此使阅读变得更加有趣：http://www.ascusc.org/jcmc/

The Journal of Mind and Behavior。主要是给心理学者，但也提供有趣的人工智能新研究领域的线索：http://kramer.ume.maine.edu/~jmb/welcome.html

第 1 章　人工智能资源

DARPA. the Defense Advanced Research Projects Agency of the Department of Defense（国防部高级研究项目组）。人工智能研究的一个主要基金来源。而且，DARPA 同时为 ARPANET 的启动提供了资金，ARPANET 引发了 20 世纪 90 年代的商业因特网。DARPA 宣称在 1990 年的海湾战争中，人工智能在“沙漠风暴”行动的后勤规划中的应用已经成倍地回馈了他们自 20 世纪 50 年代以来投资于人工智能研究的资金。http://www.au.af.mil/au/aul/school/acsc/ai02.htm

American Association for Artificial Intelligence（美国人工智能学会）。一个很好的学习了解人工智能和专家系统的起点。http://www.aaai.org/AITopics/html/welcome.html。更多的专家系统论文和资源可以参考链接：http://www.aaai.org/AITopics/html/expert.html#good

Out of Control：《***The New Biology of Machines***》，Kevin Kelly 著，这是一本引人入胜的书，你可以在线阅读了解人工智能是如何影响社会和经济。值得一读：http://www.kk.org/outofcontrol/

Metaxiotis K. and Psarras, J. "**Expert Systems in Business: Applications and future directions for the operations researcher**," *Industrial Management & Data Systems*, Vol. 103, No. 5, pp. 361-368, 2003.

Hugh McKellar, "**Artificial intelligence: Past and future**," KMWorld Magazine, Vol.12, Issue 4, 2004。概述了人工智能和世界前 5 个人工智能市场领域：专家系统、信念网、决策支持系统、神经网络和 Agent 的发展，这些领域到 2007 年市场将达到 210 亿美元。这个免费在线杂志专注于报导知识管理的最新趋势。是非常值得一读的概述文章。http://www.kmworld.com/publications/magazine/index.cfm?action=readarticle&article_id=1504&publication_id=1

Alexander Nareyek, "**AI in Computer Games**." ACMQueue.com. ACM Queue Vol.1, No.10, February 2004。较好地讨论了有限状态机、决策树以及其他人工智能技术在游戏中的实际应用。人工智能应用：http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=117&page=1

AI on the Web. 大量收集了超过 850 个人工智能所有领域的链接，来源于 Stuart Russell http://www.cs.berkeley.edu/~russell/ai.html。这些和其他资源来自于 Stuart Russell 和 Peter Norvig 所著的 Artificial Intelligence: A Modern Approach（第二版），2002 年。这是一本具有很多其他支撑材料的人工智能入门书，高度推荐。http://aima.cs.berkeley.edu/

浏览和搜索 Google 的人工智能链接：http://directory.google.com/Top/Computers/Artificial_Intelligence/

浏览和搜索 Yahoo 的人工智能链接：http://dir.yahoo.com/Science/Computer_Science/Artificial_Intelligence/

The AI Center：http://www.ai-center.com/sitemap.html。一个很好的人工智能学习起点。如果你点击链接 http://www.ai-center.com/links/将得到很多资源。

Dr. Mark Humphrey's Homepage。具有吸引人的阅读材料，你从中可判断你的祖先是否具有王室血统！当然引用其页面的其他原因在于它可以作为教学的补充链接。人工智能和许多其他资源都罗列在其网页上。http://www.compapp.dcu.ie/%7Ehumphrys/index.html

AI Links。Humphrey 的网页，其庞大的列表覆盖了下面常见主题，每一个都有大量链接：http://www.compapp.dcu.ie/%7Ehumphrys/ai.links.html

- 思维结构（包括机器人的自适应行为）
- 学习（增强学习）（神经网络）
- 进化（计算进化）（自然进化）

Teaching Links。Humphrey 提供的庞大在线人工智能资源。如果你从未学过人工智能课程或者正在搜寻某些新见解，这些内容实在值得一看：http://www.compapp.dcu.ie/%7Ehumphrys/teaching.html

而且这个网站还提供了很好的世界搜索引擎集合：http://www.compapp.dcu.ie/%7Ehumphrys/computers.internet.html

他的王室血统搜索很好用。例如，你可以看到 Walt Disney 的家族树追溯到英格兰王室 Henry I Beauclerc（公元 1070 至 1135 年）。http://members.aol.com/dwidad/disney.html#wd。现在你知道为什么城堡是 Disney 乐园的一个标志。

由 Planet 9 Studios 创建的虚拟环境包括了超过 **40** 个高精度的 **3D** 城市。这些数据集可以有多种用途，包括自治机器人在城市街道间穿行的最后测试。如果你曾经在电影 Matrix 中看过 3D 城市，这对

你来说将非常熟悉。你可从 http://www.planet9.com/demos_dwnlds.html 下载演示 WMV 文件。

Online Robotics and Cameras archive: http://ford.ieor.berkeley.edu/ir/

由 Stuart Reynolds 收集的超过 350 个人工智能文章的链接：http://ford.ieor.berkeley.edu/ir/

Reinforcement Learning*: *An Introduction。由 Richard S.Sutton 和 Andrew G.Barto 所著的在线书籍，同时 MIT 出版社在 1998 年印刷出版。探讨软件 Agent 和机器人如何学习对环境做出反应。讨论许多技术，包括马尔柯夫决策过程、规划与学习以及某些复杂案例学习。值得一看：http://www-anw.cs.umass.edu/%7Erich/book/the-book.html

Complexity Papers Online。一个大型的多主题论文集合，包括人工智能、复杂理论、混沌和数百个其他主题。包括初始人工智能创立者的经典论文：http://www.calresco.org/papers.htm

EvoWeb。整个欧洲关于进化计算所有方面的网页。包括遗传算法、机器人、智能系统，还有很多软件文章、书籍和链接。在这个站点可以看到最新新闻和技术。欧盟正通过政府资助基金帮助小型企业创建网页，以向全球市场提供它们的产品和服务。EvoWeb 的创建显示了他们运用高级人工智能技术，例如进化计算的决心。http://evonet.lri.fr/

其他主要专注于人工智能特定领域的欧洲站点也列在上述网页中。

EvoBIO。生物信息网页，专注于基于进化计算的算法，解决在分子生物学、基因学、遗传学等方面的重要问题。

EvoELEC。进化电子学方面的欧洲网页。包括了数字和模拟进化硬件、生物计算、进化机器、进化电子学等。

EvoGP。遗传编程网页。针对一些困难的设计问题、模式识别、控制问题，以及金融数据挖掘、信号和图像处理、生物信息、工程设计、音乐和艺术等应用问题。

EvoIASP。图像分析和信号处理的欧洲网页。数百篇进化算法（EAs）在图像分析和信号处理方面的应用文章表明，进化算法是自动设计和系统优化的有效工具。

EvoROB。进化机器人的欧洲网页。专注于进化算法应用于自动设计和自主机器人。它采用自底向上方法进行机器人学习，通过与环境交互而不是由人来进行综合控制。

EvoSTIM。调度和时间表的欧洲网页。进化算法被证明为非常成功。

Tom Lloyd "**When swarm intelligence beats brainpower**"。关于使用复杂工具例如群体智能来求解非常复杂的问题，例如旅行售货者问题，当问题规模达到一定大小时，不存在有效的算法。随着城市数目的增加，即使使用网格计算也无法避免速度变得缓慢，因为对于 N 个城市的可能路线有（N-1）!个。群体智能使用类似蚁群算法的技术提供 N 较大时一个可接受时间内的较好解决方案。这篇文章是商业购买群体智能技术解决复杂问题的一个好的概述。http://money.telegraph.co.uk/money/main.jhtml?_xml=%2Fmoney%2F2001%2F06%2F06%2Fcnantz06.xml

SwarmWiki。一个主要的提供信息、软件和许多处理群体智能及其人工智能应用的站点。http://wiki.swarm.org/wiki/Main_Page

好的在线演讲。包括了许多著名的演讲者通过微软多个大学研究实验室发表的各种主题的演讲：http://murl.microsoft.com/ContentMap.asp

从 **PROLOG** 角度进行人工智能和专家系统教学的幻灯片系列在线资源。由 Alison Cawsey 著，涵盖了本书理论部分的大多数主题。即使有新的语言例如 Python，PROLOG 和 Lisp 仍然是人工智能教科书使用的基本语言。http://www.cee.hw.ac.uk/~alison/ai3/

大量收集许多人工智能主题的幻灯片，由 Aaron Sloman 所做，值得一看：http://www.cs.bham.ac.uk/%7Eaxs/misc/talks/

神经网络不仅用于研究，也用于实际应用，为信用卡联合服务组织（CUSO）在检测信用卡欺诈上节省了上百万美元。PSCU 金融服务中心，国家最大的 CUSO 宣称在 2002 年超过 74 000 个信用卡欺诈案中，它使用神经网络挽回了 1280 万美元损失，占挽回损失的 99.9%。http://www.cuna.org/news-

now/archive/list.php? date=041003

PSCU金融服务中心使用的工具是 Falcon neural net tool，这个工具实时检测信用卡欺诈，http://ww.pscufs.com/falcon.htm。如果你曾经在登记处等待了较长的时间，可能是神经网络正在检查你的可疑活动。对于不同寻常的大额消费，交易可能被拒绝，你必须致电信用卡公司确认是你本人在进行消费。

国家信用卡联合委员会要求他们的客户使用神经网络技术作为减少欺诈和降低开销的方法。至于其他的商业应用，BusinessWeek 在线站点中有很好的搜索引擎可以很容易地搜索到最近的人工智能应用：http://www.businessweek.com/bw50/content/mar2003/a3826072.htm

CorMac Technologies 具有非常全面的成功商业人工智能工具和例子，包括：

- 使用神经网络诊断癌症
- 使用归纳规则抽取来诊断癌症
- 从线粒体 DNA 中进行归纳规则抽取
- 从线粒体 DNA 中进行聚类识别

其站点在 http://www.cormactech.com。一个人工神经网络软件的试用版本在 http://cormactech.com/neunet/index.html，其他产品可以从他们的主页中得到。许多完整的例子和数据集在 http://cormactech.com/neunet/download.html#DATA

很多有趣的人工智能演示和项目：http://www.cs.wisc.edu/~dyer/cs540/demos.html

人工生命链接：http://www.alcyone.com/max/links/alife.html

Simple online explanation，称为“普通人的模糊逻辑”：http://www.fuzzy-logic.com/

模糊在线书籍和模糊专家系统 FLOPS：http://members.aol.com/wsiler/

遗传编程的优点：http://murl.microsoft.com/LectureDetails.asp? 785

Genetics and Evolutionary Algorithm Archive。遗传算法和进化算法及其应用的巨大资源。http://www.aic.nrl.navy.mil/galist/

GAUL™-the Genetic Algorithm Utility Library。一个进化计算开源代码的顶尖图书馆：http://sourceforge.net/forum/forum.php? forum_id=386667

PMSI 有一个好的站点，具有大量向导和工具，可以免费下载例子：http://www.pmsi.fr/homegb.htm。遗传算法的下面应用在：http://ww.pmsi.fr/gafxmpa.htm

- 食物：机器人学习寻找食物
- GAFUNC：各种优化问题的函数
- TSPSA：模拟退火神经网络求解旅行售货者问题

神经网络的演示在：http://www.pmsi.fr/sxcxmpa.htm

- Parabola：神经网络的实际应用
- FUNCTION：演示神经网络学习建立函数模型的能力
- OCR：一个简单的视觉特征识别例子
- VEHICLE.EXE：车辆向导、行为建模和模型倒置，车辆或机器人试图从某个任意起点找到到达目的地的路径。

使用遗传算法和模拟退火的神经网络的进化发展，你可以从：http://www.pmsi.fr/grnxmpa.htm 下载到很多有趣的例子。

归纳决策树发展，使用经典的 ID3 和 C4.5 方法自动建立决策树：http://www.pmsi.fr/padxmpa.htm

蚁群进化软件，来自 Marco Dorigo 和 Thomas Stützle 所著的 Ant Colony Optimization，MIT 出版社，2004。一个应用是解决重要的旅行售货者问题，这个软件在：
http://iridia.ulb.ac.be/~mdorigo/ACO/aco-code/public-software.html

计算机科学历史。关于计算机科学多年来的发展的许多链接，有许多关于重要历史人物，如阿兰·

图灵的主题，http：//www.eingang.org/Lecture/toc.html

图灵测试网页。关于阿兰·图灵一生和其重要贡献的一个介绍网站，包括图灵测试的在线参考、背景阅读材料、由 Dr. Hugh Loebner 在 1991 年创建的 10 万美元 Loebner 奖，奖给第一个通过非限制图灵测试的计算机程序的作者。你可以通过某些程序，看看人们在哲学或思维、智能机器人和其他主题所做的有趣的工作。许多内容可用于课堂演示和进一步的工作：http：//cogsci.ucsd.edu/%7Easaygin/tt/ttest.html#people

阿兰·图灵的主页。阿兰·图灵是谁？可以从这里知道。以时间为线索，提供了图灵工作的一个完整文献目录，在线传记，文件、图片、专业论文等：http：//www.turing.org.uk/turing/

图灵测试不是骗术。图灵测试作为有效的科学手段可以分辨计算机和人类，而不是人们误以为的游戏。从另一方面来说，当人们通过电视游戏例如 Jeopardy，或幸存者游戏获得百万美元，难道也仅仅只是一个游戏？这就好像说投身职业篮球只是一个游戏，但某些球员每年可得到 10 000 000 美元！http：//www.ecs.soton.ac.uk/%7Eharnad/Papers/Harnad/harnad92.turing.html

2003 年的 Loebner 奖获得者。尽管 2003 年 Jabberwock 并没有完全通过非限制的图灵测试，Jabberwock 还是因为其产品获得了 2 000 美元。尽管不是 10 万美元，但是你有多少软件产品可以给你带来 2 000美元？http：//www.surrey.ac.uk/dwrc/loebner/

Botspot。用户可以在这个网页尝试很多流行的 Chatterbots。Chatterbots 是一种能像人一样进行会话的程序。有一些是真正为图灵测试设计的，而其他则是为各种各样的娱乐而设计。他们越来越完善，但仍不足以赢得 Loebner 奖。说不定你的老板会使用其中一个来回答你关于项目作业的问题。其最终的目的是取代在电话中心工作的人们。这将比任何外来工便宜，因为你至少得付薪水给他们。http：//www.botspot.com

The LISA project。是基于 Lisp 的智能软件 Agent 的开发平台。它是在 Common Lisp Object System（CLOS）上实现的一个产生式系统，受 CLIPS 和 Jess 的影响较深。智能 Agent 在今天可以用于多种场合。http：//lisa.sourceforge.net/

New Directions Magazine 有很多关于量子计算机和其他主题的文章和链接。都是科普文章，是一个开始学习的好地方。主页可以看到有关科学和计算的最新发展信息。同时还有一个工作栏目，不过绝大多数工作是在欧洲，这是因为 New Scientist 在英国出版。http：//www.newscientist.com/hottopics/quantum/

Quantum Information Science。澳大利亚 Innsbruck 的短期课程。有大量量子计算、量子密码和其他主题的演示报告。某些报告是我所见过的最好的，值得一看。http：//wtec.org/qis/Awschalom.PDF

人工智能接口标准委员会。这是全球游戏开发协会的下属单位。他们的目标是“……促进基本人工智能功能接口、代码重用和供应，把程序员从底层人工智能开发中解放出来进行更复杂的人工智能资源工作。如果你想加入游戏行业，或者为公司建立人工智能开发标准，而不是在每次新的项目需要人工智能技术时才“重新启动车轮”，那么这是一个很好的站点：http：//www.igda.org/ai/

网格计算由于背景的原因，还没能使用网络或者因特网的很多计算机能力。它被用来模拟大量的虚拟网格计算机，处理能力只受到带宽和计算机数量限制；理论上世界上的每一台计算机都可以应用其中。网格计算正在被越来越多的厂家例如 IBM 用于实际应用 http：//www-1.ibm.com/grid/。

新的网格服务提供商不需要真正的超级计算机就能提供超级计算能力。事实上，网格可能比任何单独的超级计算机的处理能力都强，这取决于具体任务如何分派给所有相联的机器。特殊目的的网格，例如 BioGrid 正用于生物和医药研究中，因为当在所有机器上运行时软件共享使得分析更加有效，虽然这并不一定必要。网格也可以用于探索由于计算机能力有限以前没有进行实践的新的人工智能技术和算法。以下是一些参考内容和链接：

Madhu Chetty and Rajkumar Buyya. “**Weaving Computational Grids: How Analogous are they with Electrical Grids.**” *Computing in Science & Engineering*, pp. 61-71, August 2002.

Sergio Rajsbaum. “**Distributed Computing Research Issues in Grid Computing,**” *ACM SIGACT News Distributed Computing Column 8*, 50-70, July 2002.

更多的网格计算链接：

http://www.gridcomputing.com
http://www.globus.org/
http://www.gpds.org/
http://www.ncbiogrid.org/
http://www.biogrid.jp
http://www.biogrid.icm.edu.pl
http://www.ncbi.nih.gov/BLAST/
http://www.nbirn.net
http://www.eu-datagrid.org -
http://www.bioinformaticsworld.info/feature3b.html
http://www-1.ibm.com/grid
http://www.sbml.org
http://biocomp.ece.utk.edu
http://www.grid.org
http://gridcafe.web.cern.ch

专家系统应用

专家系统应用，论文、软件和公司。来自 PCAI 在线杂志的大量列表：http://www.pcai.com/web/ai_info/expert_systems.html

医疗专家系统。大量列表：http://www.computer.privateweb.at/judith/name_3.htm

人工智能和专家系统。大量链接：http://www.dmaier.net/teaching/cis386/links.htm

农业专家系统。大量链接：http://potato.claes.sci.eg/Home/wes.htm

CLIPS 概述幻灯片，来自 Peter Jackson，《*Introduction to Expert System*》一书的作者，这是一本介绍 CLIPS 和其他专家系统的书：http://www.geocities.com/jacksonpe/clips/clips.htm

Judith Lamont，**"Innovative Applications Make Government More Responsive"**，KMWorld Magazine，Vol.12，Issue 6。较好地介绍了不同的政府机构如何使用专家系统来更好地回答公众问题。这是一个具有很多关于知识管理和智能系统在商业和政府应用方面有趣文章的杂志，值得一看。可以通过搜索引擎或发布列表浏览文章：http://www.kmworld.com/publications/magazine/index.cfm?action=readarticle&Article_ID=1541&Publication_ID=93

在线医疗诊断专家系统。如果你营养不够，试一试他们的免费诊断：http://easydiagnosis.com/

U.S.Department of Safety & Labor（美国安全劳动部）提供了许多有关联邦雇佣法、工作场所法、权利和义务的专家系统。http://www.dol.govelaws/see_adv.asp?Subset=ID>0

OSHA 政府专家系统。协助承诺处理的电子工具和产品：http://www.osha.gov/dts/osta/oshasoft/eTools。

OSHA 应急计划专家系统。设计判断公司是否需要应急计划，如果需要，则帮助创建一个方案：http://www.osha-slc.gov/SLTC/etools/evacuation/experts.html

自然科学基金－环境监控与测量顾问：http://www.emma-expertsystem.com/

美国地理概貌－天鹅管理决策支持系统：http://swan.msu.montana.edu/cygnet/

自然资源与环境部的牛奶场专家系统。澳大利亚—第七"在线顾问"：http://www.nre.vic.gov.au/web/root/Domino/Target10/T10Frame.nsf

BusinessLaw.gov 具有大量的商业和专家系统信息 http://www.businesslaw.gov/tools/business-wizards.htm

Shyster。一个基于案例的法律专家系统，由 James Popple 开发。设计、实现、操作和测试细节都在他的书中给出：《*A Pragmatic Legal Expert System*》，运用了 Legal Philosophy 丛书，Dartmouth，Aldershot，1996。Shyster 在：http://cs.anu.edu.au/software/shyster/

Knowledge acquisition and expert system shell Acquire® 。一个软件的试用版本，可以从其站点获得：http://www.aiinc.ca/ 还有很多例子在 http://www.aiinc.ca/demos/index.html：

- The Whale Watcher
- The Graduate Admissions Screening System
- The Spa Advisor
- The Stock Demo
- Petroleum Advisor for the Geochemical and Environmental Sciences
- The Job Coach
- Douglas-Fir Cone and Seed Insects System

XpertRule Knowldege Builder，以及一个数据挖掘工具，名为 XperRule Miner，可以从 Attar 软件有限公司获得。使用他们的工具创建的专家系统列表可以从 http：//www.attar.com/deploy/demos.htm 以及他们的附属公司了解到：

- Expenses web demo
- Savings and Investments web demo
- PC support web demo
- Pension web demo
- Direct PC Sales
- PC support web demo

Attar 软件有限公司的其他一些应用产品 http：//www.intellicrafters.com/cases.htm

- **Rockwell Aerospace and NASA**。由 Rockwell 国际公司为 NASA 开发的污染控制建议专家系统。
- **Channel 4 TVResource Optimization using XpertRule®** ，使用遗传算法安排连续商业广告间隔。
- **Tokyo Nissan**。使用 XpertRule® 的智能“汽车选择系统”。
- **Australian Taxation Office**。税收专家系统。
- **Work and Income New Zealand**。3000 名员工使用 XpertRule® 专家计算器处理有关工作胜任、津贴、收益等问题。
- **Hosokawa MicronData**。挖掘粉末与颗粒处理。
- **GE Capital Global Consumer Finance**。金融方面的数据挖掘。
- **The Gas Research and Technology Centre**。减少天燃气开采花费的数据挖掘。
- **Department of Industry and Fisheries, Tasmania**。Tasmanian 政府机构服务于农民的专家系统。
- **Department of Industry and Fisheries, Tasmania**。针对森林公司和承包商的有威胁动物顾问系统。
- **Misselbrook and Weston stores**。基于知识的系统，检测商店内部欺诈。
- **Hibernian（Ireland）**。数据挖掘与基于知识的系统，实现企业过程重组，由 Hibernian Life&Pensions（Ireland）开发。
- **United Distillers**。使用 XpertRule® 进行资源优化，优化苏格兰 United Distillers 酒厂的 whisky 生产以便产出高质量的混合物。
- **ICI and Carlsberg Tetley**。ICI's Thornton Power Station and the Carlsberg Tetley brewery 使用数据挖掘减少能耗。
- **Elf-Atochem North America**。Produced Rilsan® Advisor，一个为他们的技术销售人员和市场人员提供向导的专家系统。
- **The Leeds Building Society**。Leeds Building Society（现在是 HBOS plc 的一部分）使用数据挖掘来发现抵押拖欠。
- **Swedish Marines**。瑞典海军健康专家系统。
- **Heureka**。瑞典工业开发委员会的产品评估专家系统。
- **VAT in Sweden**. 一个瑞典 VAT（增值税）的税务建议和培训系统。
- **Meiji Mutual Life Insurance**。基于知识的系统，用于 Meiji Mutual Life Insurance 公司的保险计划选择。
- **Ebara Manufacturing**。用于一个日本公司调整 3000 个风力水力汞以满足顾客需求。
- **Traversum AB**。数据挖掘和基于知识的系统，提供欧洲股票和参股建议。
- **Sun Direct Insurance**。基于知识的系统，用于家庭和汽车保险咨询。

LPA（Logic Programming Associates）具有基于 PROLOG 增强版本（称为 PROLOG++）的软件工具。另一个称为 Flint 的工具被设计用于 Windows 环境下的模糊逻辑应用和 PROLOG。同时还有免费的 PROLOG 版本。有趣的演示在：http：//www.lpa.co.uk/pws_dem.htm

- Moon Phase
- Choose Meal
- Salesman
- Eliza
- Chat-80
- Network
- Expert
- E-Forms
- Insurance

Exsys. Inc。具有用商业专家系统工具开发出的大量现代专家系统列表。一个演示版本可以从 http: //www.exsys.com/case2.html 下载。以下列出了用他们的专家系统工具开发出来的大量应用，网站的软件具有版权，需经允许才能复制。

金融服务 - 欺诈检测 - 保险

- Commercial Loan Approval Predictor and Fund Selector
- Detecting Insider Trading
- Expert Credit Analysis & Analytical Report Support
- System Prevents $Millions in Costs Due to Compromised International Assignments
- EXPERTAX Handles Complex Tax and Legislative Auditing and Reporting
- Private Online Pension "Consultant" Helps with Financial Planning Decisions
- Expert Assistance and Database Analysis for Examiners Available Over a Network
- Online Business Structure Recommendations from SBA
- Bad Check Legal Assistant
- Web-based system provides financial services to Navajo Nation

法律 - 法庭程序 - 法律执行

- Electronic Arrest Warrant and Bad Check Legal Assistance for Judges
- System Handles Many Requirements of Loans
- BusinessLaw.gov Provides Online Legal Business Structure Advisor
- Detecting Insider Trading on Stock Exchange
- Public School Online Advisor Helps Select Appropriate Disciplinary Actions
- Handling Legal Issues of Environmental Compliance
- System Provides Quick Determination for Claimants
- Tax and Legislative Auditing & Reporting
- Making Relevant Sense of Fire Code Standards
- The Expert on Security Classification Guidance
- Over 20 Advisory Systems (and Growing) from Dept. of Labor
- System Provides Federal and State Accounting and Reporting Capabilities

农业 - 森林 - 地球科学

- Best Seed Selection for Best Yield and Profit
- Cross Breeding System Increases Margins by up to 50%
- Forest Inventory - Quality and Quantity
- Lynx Population Management System
- Irrigation and Pest Management
- Planning and Design of Agroforestry Systems
- Tree Selection Application

交通 - 船运 - 高速公路

- Rockwell's Aircraft Systems Material and Design Expertise
- Railroad, Navy and Air Force Engine Component Failure Prediction
- American Association of State Highway and Transportation Officials System
- Improved Nautical Chart Cartography
- Highway Construction Equipment Selection
- Better Weather Prediction for Safer Seas and Skies
- Federal Highway Administration and Transportation Research Council System
- System Identifies Problems with Overseas Assignments

电子 - 通讯 - 因特网/内部网

- System Runs on Same Hardware as Machine Vision Programmer
- Expertise for Rockwell's Autonetics Sensors and Aircraft Systems
- Analysis of Spectrum Analyzer Data Saves Millions
- Pacific Gas & Electric Field Assistant Helps Service Revenue Meters
- EPRI Prevents Power Outage Due to Bearing Failures
- GE Identification of Common Metals
- HP's Recommendations for Warehouse Implementations
- CIM Cell Re-Configuration in Minutes, Not a Full Day
- Rotating Equipment Vibration Advisor
- Pacific Bells' Monitoring, Prediction, and Repair Network Assistance
- AUDEX Provides Electrophysiological Expertise
- U.S. Postal Service Electronic Performance Support Systems
- Voice Driven Diagnostic System

销售与市场 - 在线产品选择 - 发布

- Online Product Configuration Drives HP's E-Business Strategy
- Sales Support System Provides Pricing, Quotations and Reports
- Multimillion$ Loan Approval System Saves Study Time and Costs
- Profiling System Helps Businesses Successfully Penetrate Foreign Markets
- Cost, Labor & Productivity Analysis and Estimating
- Credit Analysis Advisor and Report System
- Selecting the Right Equipment, Then the Right Model
- Power Generation Unit Commitment Advisor Customized for Each Client

化学－自然资源－挖掘

- Compliance for Asbestos Contamination
- Texas Eastman Connects Over 400 Systems with External Programs
- System Identifies Impurities from Analysis of Spectrum Analyzer Data
- Know What to Do When Working with Fuel
- GE Identification of Common Metals
- Vibration Advisor Uses Patterns and Symptoms in Diagnosis
- Nestle's Real-Time Process Control
- EXNUT Reduces Fungicide Use in Farm Management Operations

医药－诊断－保健组织

- Respiratory & Anesthesia Monitoring
- Cedars-Sinai Information Management for Lung Cancer Patients
- System Determines Adjustment/Stress Issues Before Overseas Assignments
- Experts Agree with Pediatric Auditory Diagnostics System 100%
- Keeping People Safe From the Hazards of Asbestos
- In-Vitro Fertilization Cycle Stimulation
- Urodynamic Diagnosis Rates Diagnosis in Order of Probability
- Hematology Support with Voice Driven System for Accurate Diagnosis

建筑－设备选择－费用评估

- Advisors Help Contractors Interpret Complex Compliance Information
- Labor Cost Diagnostics for Steel Construction and Welding Selection
- Major Commercial Construction Loan Approval Predictor
- System Provides Front-End Analysis for Construction Simulation
- Equipment Selection for Highway Construction
- Fast Fire Code Interpretation for Architects, Engineers and Designers
- Control Panel Layout Design Aid
- In-the-Field Procedural Assistant
- Confined Spaces Work Place Advice
- Economic, Crew and Maintenance Optimization for Scheduling
- Construction Specialists Systems Help Project Supervisors at Field Sites
- Work Zone Interactive Video Trainer & Advisor

计算机硬件与软件

- Complex Configuration of Computer Integrated Manufactured Cells
- Financial Analysis Support Techniques (FAST)
- Customized Diagnostic System
- Eastman Implements Knowledge Automation Systems Enterprise Wide
- EASE System for Dept. of Labor Running on LAN
- Langton Clarke's EXPERTAX in Use Since 1986
- HP's Interactive Hardware and Network Configuration
- A Blend of Knowledge Automation Systems and 3-D Design
- Nestle's Real-Time Process Control with IBM Cooperative Effort
- Systems Monitor Pacific Bell's Front-End Computers
- EXSYS Streamlines ANVIL's Tapes That Control Selection Process
- U.S. Postal Service Electronic Performance Support Systems
- Multimedia Integration Aids User in Analyzing and Diagnosing Problems

培训－维修－故障修理

- System Keeps Compliance Support Up to Date for Businesses and Consultants
- Troubleshooting System Prevents Power Outages
- Tutorial Tool for Inexperienced Personnel
- 13 Years Running - Component Failure Prediction System Saves $Millions
- Profiling System Works in Tandem with Cross-Cultural Training
- Interactive System Helps Train New Auditors
- Troubleshooting Problems in Complex Equipment
- System Brings Less Experienced Case Workers Quickly Up To Speed
- System Assists New Judges
- No Electricity? No Problem - Laptop Systems Provide Support in the Field

- Training Tool Explains Recommendations and Brings in External Graphics
- Classification System Makes Determination in Minutes
- OSHA Helps Businesses Help Themselves
- NetHELP Provides 24-Hour Network Repair Assistance
- Diagnostics System Helps in Training Programs
- Portable Self-Paced Training Covers Operations Processing and Repairs
- System Chooses Testing by Considering all Relevant Factors
- Interactive Video/Trainer/Advisor

研究与开发－识别

- Land Management System Assists Research Scientists
- Agronomy Systems Combine "Expert Intuition" and Hard Data
- Effective Cross Breeding Strategies via Genetic, Environmental Management
- System Runs on Same Hardware as Machine Vision Programmer
- Stratified Line Plot and Point Sampling Used in Forest Inventory System
- GE's System Identifies Metals in Non-Laboratory Setting
- System for Invitro Program Speeds-Up Critical Decision Making
- Improved Design Process Without Expensive Iterative Analyses
- Diagnose and Locate Problems in Electrical, Mechanical or Fluid Systems
- System Provides Relevant Staging, Prognostic and Therapeutic Information
- Improved Accuracy for Charting and Geodetic Service
- EXSYS Customized Interface Key to Real-Time System Success
- Years of Experience Codified into Classification System
- Turn Here When You Have to Work In Confined Spaces
- Pacific Bell Monitoring, Repair and Prediction Systems
- National Research Lab Systems Provide New Concepts
- Diagnostic System Challenge Results in 100% Domain Expert Agreement
- Scheduling System Combines Traditional Numeric Methods with Heuristic Rules
- Rubber Research Institute Fields Systems in the "Field"
- Sandia National Laboratory System Greatly Reduces Information Gathering Time
- National Oceanic and Atmospheric Administration Prediction System
- Support System in Latest Portable PC and Digital Technology
- Voice Driven Interactive Diagnostic System Frees Eyes and Hands for Other Work

能源－工具－油气

- Unit Commitment Advisor for Power Generation Scheduling
- Power Plant Outage Prevention and Maintenance Procedures
- Invisibly Embedded System Handles Failure Prevention with Extreme Accuracy
- Advisor for Fuel Delivery Systems
- Nuclear Power Plant Automates Emergency Contingency Plan
- Problem-Solving and Maintenance System Distributed to over 400 Client Installations
- Pacific Gas & Electric Personnel Assistance
- U.S. Department of Energy Classification Automation
- Work Place Confined Spaces Compliance Systems
- National Lab Numerically Controlled Machine Tool Selection

工程－计划－调度

- Human-Factors Engineering Reduces Costly Operator Errors
- Material and Process Design Expertise From the "Get-Go"
- Complex Configuration of Computer Integrated Manufactured Cells
- Construction Simulation & Analysis
- Customized Diagnostics for Machine Vision System
- Eastman Distributes Top Engineering Expertise Enterprise
- EPRI System Prevents Bearing Failures
- Highway Engineering Support for Earth-Moving Projects
- Oil Analysis Saves $Millions By Predicting Component Failure
- Fast and Specific Standards Interpretation for Engineers
- GE Identifies Common Metals
- Mechanical Equipment Diagnosis in Electrical, Mechanical or Fluid Systems
- Real-Time Analysis Increases Power and Enables Multi-Tasking
- Interactive Online and Downloadable Systems Help with Confined Spaces Permits
- Monitoring, Repair and Prediction Handled Through Network System
- Power Generation Scheduling Incorporating Numeric and Qualitative Aspects
- Numerically Controlled Machine Tool Selection
- Total Productivity Support for U.S. Postal Service Mail Sorter
- Major Oil Company Reduces Cost with Weld Procedure and Test Selection Systems
- Traffic Control and Management Strategies for Construction

安全－质量控制－人力资源

- Asbestos Advisor System Honored in Vice President's "Best Practices" Award
- Disaster Manager's Emergency Management System
- Interactive Corporate Family Support for Overseas Assignments
- Career Goal Advisor Assists Supervisors with Development Planning
- Unemployment Eligibility Determination with Data Access and Automated Reports
- Tremendous Time Savings, Nothing Overlooked and Site-Specific Report
- Better Human-Factors Design Provides Safer Plant Operation
- Case Worker Advisor Provides Equitable, Effective Welfare Administration
- In-the-Field System Provides Quick Indexing and Reference to Safety Precautions
- Nestle Pension Fund Advisor Distributed to all Personnel Departments
- Public School Disciplinary Action Advisor
- Nuclear Weapons Security Classification
- Thousands Access System Which Provides Safer, Healthier Work Place
- Automated Student Advising System Provides Individualized Assistance
- Advisors Help Farm Operations Management
- Fast Storm Forecasting in Limited Time Frame
- Work Zone Planning, Safety and Design Aid

规划/环境/政策保护

- Interactive Regulatory Compliance Assistance
- Environmental Compliance Support System
- Public School Disciplinary Action Advisor
- Counter Measures for Nuclear Power Station Emergencies
- System Handles More than 400 Questions a Week
- Wildlife Population Protection and Trapping Regulations System
- System Detects Situations and Concerns in Overseas Relocations
- Online Help for Choosing a Legal Business Structure
- National Fire Code and Environmental Permit Advisor
- System Facilitates Transition of Welfare Program to Navajo Nation
- Organizational Policy on Development Goals
- Incorporating Environmental and Management Constraints
- Increasing Quality and Uniformity of Legal Procedures
- EPRI Reduces Forced Plant Outages with Maintenance Procedures System
- In the Field Procedural Assistant
- Consistency & Reduced Subjectivity for Marine Service Organization
- Pension Fund Advisor Conforms with Standards
- System Interprets over 100 Classification Guides
- Environmental Conditions Factor into Tree Selection
- Severe Thunderstorm Prediction
- Major Oil Company Reduces Costs with Selection of Qualification Tests

制造－过程控制－预防

- Increased Performance, Productivity, Optimization and Overall Quality
- Nestle Foods Install Real-Time Applications
- Set of Systems Handle Monitoring, Repair and Prediction
- Bidding, Control and Productivity Analysis for Total Cost Evaluation
- Customized Port for Diagnostics for Machine Vision System
- Eastman Distributes "Know-How" Enterprise-Wide
- Fast Prediction Using External Calls and Complex Mathematical Calculations
- Complex Product Configuration of Computer Integrated Manufactured Cells
- GE Identifies Common Metals
- Optimized Control Panel Layout Reduces Costly Operator Errors
- Linear Programming Interface Optimizes Management Modules
- Numerical and Qualitative Rules Solve Scheduling Problems
- System Automates Appropriate Tool Selection
- Multimedia Electronic Repair Support Systems
- Voice Driven Diagnostics and Frees Hands for Other Work
- Procedure Selection System with Time Estimates and Material Requirements
- Selecting Appropriate Control and Management Strategies

政府－军事－航空

- OSHA Compliance Advisor Helps Thousands
- SBA Legal Business Structure Recommendation Wizard
- System Identifies Potential Problems Before Overseas Assignments
- Federal Money Approval System for Major Construction
- Assistance for State Magistrate Court Judges
- Los Alamos National Laboratory Support System
- Public School Disciplinary Action Advisor
- Emergency Support System for Security Bureau
- American Association of State Highway and Transportation Equipment Selector

- Claimants Unemployment Compensation Determination
- Oil Analysis Saves U.S. Air Force over $100 Million
- U.S. Dept. of Commerce - 40% Higher Accuracy for U.S. Nautical Charts
- U.S. Dept. of Energy Security Classification
- Case Worker Advisor
- U.S. Dept. of Labor Saves Time, Resources and Taxpayer Money
- U.S. Dept. of Agriculture Planning Modules - Higher Yield/Less Water & Pesticides
- Sandia National Laboratories Machine Tool Selection
- Alaska Department of Fish and Game System
- National Oceanic and Atmospheric Admin. Severe Thunderstorm Prediction
- U.S. Postal Service Total Organization Productivity Support
- Federal Highway Administration Work Zone Safety Manager

第 2 章

Ulf Nilsson and Jan Mahuszynski, Logic Programming and PROLOG，第二版，2000。一个可以免费下载的在线书籍。**PROLOG** 是专门为反向链设计的语言，它与 LISP 是仍在使用的经典人工智能语言。http: //www.ida.liu.se/~ ulfni/lpp/。许多额外的教学资源可以从 Nilsson 的站点下载。http: //www.ida.liu.se/~ulfni/teaching.shtml。Nilsson 的书提供了对知识表示、逻辑程序设计和例子的比第 2、3 章内容更详细的介绍。

特殊的语言例如 **KQML** 已经被开发出来用于知识表示和查询：http: //www.cs.umbc.edu/kqml/

Dave Hannay 的站点允许你尝试有限状态机和其他不同类型的编译器去识别教学材料中的符号。http: //scoter3.union.edu/~hannayd/csc140/simulators/

Doug Lenat 关于常识和人工智能在线演讲：

http: //murl.microsoft.com/LectureDetails.asp? 1032。他因开发 OpenCyc，Cyc 技术的开源版本（http: //www.cyc.com/cyc/technology/whatiscyc）而闻名，Cyc 是世界上最大最完整的通用知识库和常识推理引擎。他同时因其在本书中提到的 Automated Mathematician（AM）和 Eurisko 中的自动数学发现的开创性工作而闻名。

形式方法。有许多逻辑资源的主要站点：

http: //archive.comlab.ox.ac.uk/comp/formal-methods.html

通过归纳学习规则的软件。Windows 平台的 VisiRex 2.0，包括试用版本，可从 CorMac Technologies, Inc 公司获取。还可获取其他两种方法，经典 backprop 和神经网络 SFAM。软件还带有很好的例子：http: //cormactech.com/visirex/faq.html

数据挖掘资源和软件

Data mining success stories，来自 Complexica® 公司，提供数据挖掘工具和服务。http: //internet.cybermesa.com/~rfrye/complexica/dm _ em.htm

归纳机器学习的经典软件是 **Ross Quinlan** 的 **ID3**，后来增强到 **C4.5**。连同其他论文和幻灯片可以从他的个人主页获取：http: //www.cse.unsw.edu.au/~quinlan/。他最近的数据挖掘工具 See3 和 C5 提供了比 C4.5 更强的功能，可以从其公司购得：http: //www.rulequest.com/

大量的程序集。关于数据挖掘、决策树构造、使用神经网络分析数据、统计、模糊技术：http: //www.the-data-mine.com/bin/view/Software/WebIndex

大量的数据库、领域理论和数据生成器集合，可用于测试机器学习算法 http: //www.ics.uci.edu/~mlearn/MLRepository.html

大量用于机器学习算法的C++类库。可以开发数据挖掘和可视化工具：http: //www.sgi.com/tech/mlc/。因为这些代码用C++开发，你可以很容易地为这些任务编写 CLIPS 函数，然后重新编译 CLIPS 以优化性能。在本书附带的光盘中提供了 CLIPS 的完整 C++ 代码。

各种领域的大数据集集合。可用于测试数据挖掘工具。当你开发自己的 CLIPS 数据挖掘版本时会发现这些数据非常有用：http: //kdd.ics.uci.edu/

alphaWorks：一个用于数据挖掘和其他很多应用的产品。http://www.alphaworks.ibm.com/Home/

创建超链接语义网的优秀软件工具。免费而且功能强大。可以用于以可视化的形式表达知识专家知识。一个好的知识获取工具，知识获取在第6章中有讨论。附带全世界用户提供的很多例子。如果你注册到邮件列表中，将收到大量西班牙语的电子邮件：http://cmap.ihmc.us/

Tap是一个知识库，用于辅助可机读的语义网的构造（http://www.w3.org/2001/sw/），升级到当前万维网WWW。可以从以下网址得到更多链接：

http://www.iturls.com/English/TechHotspot/TH_SemanticWeb.asp

目前人们只能使用初级布尔运算对web进行关键词搜索。尽管web站点正在转化成专门领域的XML语言，如MusicXML、RuleXML、MathXML等，它们是机器可读的。但在语义上下文中，TAP将对现实世界和当前事件有更多了解。

TAP是一个浅的但广泛的知识库，包括广泛范围内流行对象的基本词典和分类学信息，例如音乐、电影、作者、体育、汽车，公司、家具、玩具、婴儿用品，处所、消费电子和健康等。可以下载下来测试不同类型的专家系统。它基本上是浅的本体，也就是说它是Cyc的补充而不是取代Cyc，因为Cyc是关于常识现象的深的本体，Cyc不包括当前事件，如：谁是Yo-Yo Ma？学生的研究项目可以基于TAP：http://tap.stanford.edu/tap/tapkb.html

知识表示系统中的传统家族，针对只要求有限的表达能力，但需快速回答问题能力的应用。语义网络具有的大多数特征传统家族也都具有。它允许用户表达其他知识表示系统和面向对象程序设计语言里的描述、概念、角色、个体、规则、框架等。其中最令人感兴趣的是概念将被自动分类，而对象将自动地继承。

传统家族可以发现信息中所存在的不一致性（这也是为什么它可以在电话推销员来电时回答电话，或者删除告诉你如何通过帮助一个外币存款来获得1千万美元收入的垃圾邮件）。

传统家族有3个成员。他们是（1）LISP版本，用于研究，（2）C版本和（3）用C++编写的Neo-Classic版本。传统家族被用于PROSE和QUESTAR中来配置产品，已经配置了价值超过40亿美元的AT&T和Lucent产品。更多细节可参看http://www.bell-labs.com/project/classic/。

Rule Markup Initiative，RuleML，是一个依据标准的正向反向链方式进行规则推理的项目，因此对不同专家系统都机器可读。另一个相关的项目是RuleXML语言，人们正努力使它们之间兼容。RuleML的站点：http://www.ruleml.org/

为规则创建一种单一的统一格式，而不是使用不同的编码风格，将有利于不同专家系统之间的知识交换。目前没有针对不同专家系统交换知识的标准方法，因为每一个专家系统工具使用自己的语法规则。这是一个大的项目，具有潜在的巨大研究回报。基本上来说，它类似重用软件而不是每次开始一个新专家系统的构建。最能实际应用的是CLIPS中的对象，因为对象是良定义的，它可以在不同的CLIPS专家系统间交换。尽管Jess用面向对象的语言Java开发，但2004的Jess不支持规则中的对象，例如COOL。Jess最大的好处在于能嵌入宿主语言Java中。

也正因为对象包含了数据、方法和良定义接口，它就更容易建立大型的专家系统。建立面向对象的混合规则系统是巨大的挑战。自从1986年发布以来，CLIPS的每一个版本都在持续修订和严格调试。但是还有很多充满诱惑的特性可以加入，例如反向链、模糊逻辑、贝叶斯等。为了保持其简单和健壮，我们一直限制着这些诱惑，以便人们可在一个稳定的软件基础上去建立系统。这样的处理方法很有效，你可以看到很多的后续版本提供各种各样的新功能。

逻辑软件资源

很多人工智能技术正被用于自动化推理过程，利用推理人们可处理逻辑。事实上，这是人工智能面对的最早挑战，重复数学家的符号推理过程而不是数字计算机器。虽然千百年来，机器已经用于提

供成倍于人类的力量，或者简单的设备例如算盘用于算术，但现在历史上第一次，人类思维的智能面临着证明符号数学定理的挑战。

谬论

在专家系统领域，在与一个领域知识的专家交流中，谬论研究很重要。有时甚至专家也会出现逻辑谬论，或者你误解专家所说而产生了一个谬论。你需要在把专家知识放入数据库前证实其正确性。比第 6 章更完整的谬论列表提供在此附录中。

好的演绎逻辑谬论的描述和例子：http://webpages.shepherd.edu/maustin/rhetoric/deductiv.htm

归纳谬论：http://webpages.shepherd.edu/maustin/rhetoric/inductiv.htm

其他逻辑谬论的列表：http://webpages.shepherd.edu/maustin/rhetoric/fallacies.htm

Critical Thinking on the Web 有很多逻辑、在线向导以及很全面的谬论列表：http://www.austhink.org/critical/

逻辑软件

以下所选软件可以从因特网中获得，覆盖了有关逻辑的广阔主题，可以用于检查你的逻辑问题的答案，观察自动推理等。这些关于逻辑和其他参考的教育软件的信息获得了 Hans van Ditmarsch，hans@cs.otago.ac.nz 的许可，请从其主页获取最新版本：http://www.cs.otago.ac.nz/staffpriv/hans/logic-courseware.html

Akka，http://turing.wins.uva.nl/~lhendrik/AkkaStart.html
功能：可证明几种逻辑公式，验证模型中的公式，推导和编辑 Kripke 模型以及动态逻辑模型
平台：web
开发者：Lex Hendriks，ILLC，Amsterdam 大学，荷兰
email：lhendrik@illc.uva.nl

Alfie，http://www.cs.chalmers.se/~sydow/alfie/index.html
功能：命题逻辑的自然演绎
平台：web
开发者：Björn von Sydow，计算机科学系，Chalmers 大学，Göteborg，瑞典
email：sydow@cs.chalmers.se
注释：ASCII 接口

Aristotle，http://www.utexas.edu/courses/plato/aristotle.html
功能：形式化英语句子为符号逻辑
平台：Windows95/98/NT
开发者：
email：marcow@cs.utexas.edu，Robert C. Koons：rkoons@mail.utexas.edu

Athena Software，http://www.athenasoft.org/
功能：层次结构化辩论及其支持，采用树形图形化界面；对否定式也有类似的工具
平台：Windows
开发者：Bertil Rolf，Blekinge 工学院，瑞典，以及其他人
email：info@athenasoft.org，bertil.rolf@bth.se
书籍：各种在线文档，PPT 文件等
注释：也可用于中学，教师教学的支持资料，2003 年加入列表

Bertie3，http://137.99.26.4/~wwwphil/SOFTWARE.HTML
功能：命题逻辑和谓词逻辑的自然演绎

平台：DOS，Windows
开发者：Austen Clark，Connecticut 大学，美国
email：austen.clark@uconn.edu
书籍：Merrie Bergmann，Jim Moor，and Jack Nelson，The Logic Book，2nd edition. McGraw Hill，1992
注释：可作为 GNU Public Domain License 软件获取

Bertrand，http://www.uwosh.edu/faculty_staff/herzberg/Bertrand.html
功能：场景式的谓词逻辑公式有效性测试
平台：Apple
开发者：Larry A. Herzberg，Wisconsin - Oshkosh 大学，美国
email：herzberg@vaxa.cis.uwosh.edu
注释：2003 年更新

blobLogic，http://users.ox.ac.uk/~univ0675/blob/
功能：命题逻辑和谓词逻辑的语义场景
平台：web（需要下载 Shockwave），Mac，PC
开发者：Corin Howitt，Oxford 大学，英国 .
email：corin.howitt@philosophy.ox.ac.uk
书籍：无
注释：在线的证明演示；保存/上传到服务器；交互向导；模态逻辑的版本随后发布；2003 年加入列表

blogic，http://www.umich.edu/~velleman/logic/
功能：布尔搜索，逻辑电路，真值表，带可能世界图的模态逻辑的语义，量化
平台：web
开发者：David Velleman，Michigan 大学，美国
email：velleman@umich.edu
书籍：这是一本交互式教材
注释：2003 年加入列表

Boole，http://www-csli.stanford.edu/LPL/
功能：真值表
平台：Windows，Apple
开发者：John Etchemendy，Stanford 大学，美国，Jon Barwise，Indiana 大学，美国
email：Dave Barker-Plummer，dbp@csli.stanford.edu，或用户支持，LPLbugs@csli.stanford.edu
书籍：John Etchemendy&Jon Barwise，*Language*，*Proof* and Logic. CSLI Publications，2000. 注意：这个出版社在其网站上发布了大量关于逻辑的书，http://www-csli.stanford.edu/
注释：这本书带有软件：Fitch，Boole，and Tarski's World。一个可补充任何逻辑教材的广泛使用的一阶逻辑软件。以可视化图形世界表示逻辑命题。

Expression Evaluator，http://www.cc.utah.edu/~nahaj/logic/evaluate/
功能：语义（解释）评估命题和谓词逻辑公式
平台：web
开发者：John Halleck，Utah 大学，美国
email：John.Halleck@utah.edu
书籍：无
注释：链接到其他网页资源（模态逻辑）

Fitch，http://www-csli.stanford.edu/LPL

功能：谓词逻辑的自然演绎
平台：Windows，Apple
开发者：John Etchemendy，Stanford 大学，美国；Jon Barwise，Indiana 大学，美国
email：Dave Barker-Plummer，dbp@csli.stanford.edu，或用户支持 LPLbugs@csli.stanford.edu
书籍：John Etchemendy&Jon Barwise，*Language*，*Proof* and *Logic*.CSLI Publications，2000.
注释：这本书带有软件：Fitch，Boole，and Tarski's World.

Gateway to logic，http://logik.phl.univie.ac.at/~chris/formular-uk.html
功能：谓词逻辑的自然演绎（Lemmon- 和 Fitch-类型），真值表……
平台：web
开发者：Christian Gottschall（Vienna 大学，奥地利）
email：gottschall@gmx.de
书籍：无
注释：具有英文和德文

Hexagon，http://www.science.uva.nl/projects/opencollege/cognitie/hexagon/
功能：认识逻辑中的公共更新
平台：web
开发者：Jan Jaspars，自由激进派逻辑学者，Amsterdam 大学，荷兰
email：jaspars@science.uva.nl
书籍：无
注释：荷兰语（Amsterdam 大学的额外课程软件）

Hyperproof，http://csli-www.stanford.edu/hp/
功能：谓词逻辑的自然演绎，可视化推理
平台：Apple
开发者：John Etchemendy，Stanford 大学，美国；Jon Barwise，Indiana 大学，美国
email：dbp@csli.stanford.edu（Dave Barker-Plummer）
书籍：Hyperproof，John Etchemendy and Jon Barwise，CSLI publications，1994
注释：

Inference Engine，http://blue.butler.edu/~sglennan/InferenceEngine.html
功能：
平台：Mac，PC
开发者：Stuart Glennan
email：sglennan@butler.edu
书籍：Joseph Bessie and Stuart Glennan，《*Elements of Deductive Inference*》
http://blue.butler.edu/~sglennan/Elements.html，Wadsworth 2000
注释：

Interactive Logic Programs，http://www.thoralf.uwaterloo.ca/htdocs/LOGIC/st_ilp.html
功能：命题逻辑的真值表，词语统一
平台：web
开发者：Stanley N.Burris，Waterloo 大学，加拿大
email：snburris@thoralf.uwaterloo.ca
书籍：Logic for Mathematics and Computer Science，Prentice-Hall，1998
注释：2003 年加入列表

Jape，http://www.jape.org.uk/

功能：经典谓词逻辑的自然演绎和序列证明；以及各种其他逻辑和形式系统；以及用户定义逻辑
平台：MacOSX，Unix，Linux，Solaris
开发者：Bernard Sufrin，Oxford 大学，英国；Richard Bornat，QMW，伦敦，英国
email：Bernard.Sufrin@comlab.ox.ac.uk，richard@dcs.qmw.ac.uk
书籍：待出版
注释：Jape 是“just another proof editor”的缩写。2003 年更新

JOJ-logics：The Propositional Proof Generator，
http://pgs.twi.tudelft.nl/~tonino/teaching/JOJ-logics/JOJ-Logics.html
功能：真值表和命题逻辑的自然演绎
平台：web
开发者：Jonne Zutt and Joost Broekens，Delft 大学，荷兰
email：j.zutt @ twi.tudelft.nl，d.j.broekens @ twi.tudelft.nl 或导师 Hans Tonino：J.F.M.Tonino @ its.tudelft.nl
注释：M.Sc. 学生项目，无证明编辑

LICS web tutor，http://www.cis.ksu.edu/~huth/lics/tutor/
功能：关于各种逻辑主题的 MC 问题和答案，包括模态
平台：web
开发者：Michael Huth and Marc Ryan，计算机学院，Birmingham 大学，英国
email：M.Huth@doc.ic.ac.uk，M.D.Ryan@cs.bham.ac.uk
书籍：《*Logic in Computer Science*》，2nd Edition，Cambridge University Press，2004
注释：

Logic Animations，http://turing.wins.uva.nl/~jaspars/animations/
功能：命题、谓词、动态、模态逻辑的语义计算
平台：web
开发者：Jan Jaspars，自由激进派逻辑学者，Amsterdam 大学，荷兰
email：jaspars@science.uva.nl
书籍：无
注释：基本上是荷兰语

Logic Cafe，http://www.oakland.edu/phil/cafe/
功能：真值表，谓词逻辑辩论，MC 问题
平台：web（Linux，Mac public domain versions）
开发者：John Halpin，Oakland 大学，美国
email：halpin@oakland.edu
书籍：在线
注释：在线的逻辑课本，附带基于 web 的练习，public domain 版

Logic Daemon，http://logic.tamu.edu/
功能：谓词逻辑的自然演绎
平台：web
开发者：Colin Allen，Texas A&M 大学，美国
email：colin-allen@tamu.edu
书籍：Colin Allen and Michael Hand，*Logic Primer*（2nd ed.），MIT Press，2001.
注释：ASCII 接口

Logic for Fun，http://csl.anu.edu.au/~jks/puzzlesite/

功能：表达一阶逻辑中的疑问和其他问题
平台：web
开发者：John Slaney，澳大利亚国立大学
email：John.Slaney@anu.edu.au
Logic Toolbox，http://philosophy.lander.edu/~jsaetti/Welcome.html
功能：三段论，真值表，命题逻辑的自然演绎
平台：web，windows
开发者：John Saetti
email：john.saetti@gcccd.net
LogicCoach Ⅲ and Ⅳ，http://academic.csuohio.edu/polen/
功能：非常全面的软件，协助网站中列出的各种不同逻辑书籍的逻辑教学。包括了辩论图、定义、谬论、类比、Mill 方法、概率、范畴论、标准化、三段论、省略三段论、复合三段论、Venn 图、符号逻辑、转化：英语为符号、转化：符号到英语、逻辑表格、真值表、间接真值表、量化、同一、证明、树
平台：Windows，Apple
开发者：Nelson Pole，Cleveland 州立大学，Ohio，美国
email：n.pole@csuohio.edu
书籍：Patrick Hurley，*A Concise Introduction to Logic*，8th edition，Wadsworth，2003
注释：提供教学指导包，2003 年更新
Logics Workbench，http://www.lwb.unibe.ch/
功能：命题逻辑中的证明和计算（最小化，直觉主义，经典的，模态，非单调）
平台：web，Apple，Linux，Solaris
开发者：Gerhard Jäger（项目负责人），Peter Balsiger，Alain Heuerding，Stefan Schwendimann，Bern 大学，瑞士
email：lwb@iam.unibe.ch
书籍：扩展的在线手册
注释：无证明编辑
LogicWorks，http://www.pdcnet.org/logicwo.html
功能：真值表，辩论分析（通过例子），谬论（通过例子）
平台：PC，Apple（2000 年版）
开发者：Rob R.Brady，Stetson 大学，美国
email：（订购者）order@pdcnet.org
注释：由 Philosophy Documentation Center 发布，http://www.pdcnet.org。提供教学指导包
MacLogic，http://www-theory.dcs.st-and.ac.uk/~rd/logic/soft.html
功能：谓词逻辑（最小化，直觉主义，经典的），自然演绎，序列演算
平台：Apple
开发者：Roy Dyckhoff，St Andrews 大学，苏格兰，英国
email：rd@dcs.st-andrews.ac.uk
书籍：可选的，G.Forbes，*Modern Logic*，Oxford University Press，1993
注释：没有更新到当前版本的 MacOS
New Pandora，http://www.doc.ic.ac.uk/~kb/NewPandora.html
功能：谓词逻辑的自然演绎
平台：web

开发者：Krysia Broda，计算机系，Imperial 学院，伦敦，英国
email：kb@doc.ic.ac.uk
注释：基于“Pandora”
Paul Tomassi，Logic，http://www.oxford-virtual.com/Philosophy/Tomassi/
功能：命题和谓词逻辑
平台：web
开发者：Oxford Virtual Technology
书籍：Paul Tomassi，Logic，Routledge，1999
注释：提供配合课本的网站指导
Pier，http://gentzen.math.hc.keio.ac.jp/
功能：交互式证明编辑器，自然演绎
平台：web
开发者：Masaru Shirahata，Keio 大学，日本
email：sirahata@math.hc.keio.ac.jp
注释：编写成一个 Java Applet 和应用程序
Plato，http://www.utexas.edu/courses/plato/
功能：命题证明和谓词演算
平台：PC，Mac
开发者：
email：marcow@cs.utexas.edu，Robert C.Koons：rkoons@mail.utexas.edu
书籍：Robert C. Koons，A Logical Toolbox，2000
注释：
Power of Logic，http://www.poweroflogic.com/
功能：各种逻辑主题，证明检查
平台：web
开发者：
email：webmaster@poweroflogic.com
书籍：C.Stephen Layman，The Power of Logic (2nd ed.)，McGraw Hill.
注释：
Program to learn Natural Deduction in Gentzen-Kleene’s style，http://193.51.78.161/dnfn/deductioneng.html
功能：谓词逻辑的序列式自然演绎；正规形式计算
平台：DOS
开发者：Patrice Bailhache，Nantes 大学，法国
email：patrice.bailhache@humana.univ-nantes.fr
注释：由 Nantes（法国）哲学学生使用
Reason! Able，http://www.goreason.com
功能：可视化的辩论表达
平台：Windows 95 及其高版本
开发者：Tim van Gelder，Melbourne 大学，澳大利亚
email：info@goreason.com
注释：教学资源
Socrates，http://www.utexas.edu/courses/socrates/

功能：语义场景
平台：PC
开发者：
email：marcow@cs.utexas.edu，Robert C. Koons：rkoons@mail.utexas.edu
书籍：Robert C. Koons，A Logical Toolbox，2000
注释
Tarski's World，http://csli-www.stanford.edu/hp/
功能：谓词逻辑语义解释
平台：Windows，Apple
开发者：John Etchemendy，Stanford 大学，美国，Jon Barwise，Indiana 大学，美国
email：dbp@csli.stanford.edu（Dave Barker-Plummer）
书籍：The Language of First-order-Logic（3rd edition），John Etchemendy and Jon Barwise，CSLI publications，1994（以及：Tarski "Lite"，作者同上；Language，Proof and Logic）
注释：著名例子，包括 Hintikka 游戏（语义场景）
TPS and ETPS，http://gtps.math.cmu.edu/tps.html
功能：一阶逻辑或高阶逻辑（类型理论）的交互定理证明
平台：Unix，Windows，web
开发者：Peter B. Andrews，Carnegie Mellon 大学；以及其他人
email：andrews@cmu.edu
书籍：Peter B. Andrews，An Introduction to Mathematical Logic and Type Theory：To Truth Through Proof，2nd. Edition，Kluwer Academic Publishers，2002
注释：还可参看"ETPS：A System to Help Students Write Formal Proofs"，在 http://gtps.math.cmu.edu/tps-papers.html，2003 年加入列表
Twootie，http://137.99.26.4/~wwwphil/SOFTWARE.HTML
功能：计算命题逻辑和谓词逻辑的真值
平台：DOS，Windows
开发者：Austen Clark，Connecticut 大学，美国
email：austen.clark@uconn.edu
书籍：Merrie Bergmann，Jim Moor，and Jack Nelson，《*The Logic Book*》，2nd edition. McGraw-Hill，1992. Richard Jeffrey，Formal Logic：《*Its Scope and Limits*》，3rd edition，McGraw-Hill，1991
注释：作为 GNU Public Domain License 软件，可以获取

定理证明

更复杂的软件，称为定理证明器，自动推理或者机器推理用于逻辑方面的高级工作。

机器推理主页 http://www-formal.stanford.edu/clt/ARS/ars-db.html

World Wide Web Virtual Library：Formal Methods。收集了大量逻辑工具和语言验证器：http://vl.fmnet.info/

形式方法教育资源 http://www.cs.indiana.edu/formal-methods-education/

Logik Software für Unterrichtszwecke（德语）：
http://www.phil-fak.uni-duesseldorf.de/logik/software.html

现存机器推理系统数据库。一个巨大的资源描述和链接：
http://www-formal.stanford.edu/clt/ARS/systems.html

逻辑程序与教学辅助，定理证明器和语言。

http: //home.clara.net/ghrow/subjects/logic _ software.html

Newsletter on Philosophy and Computers:
http: //www.apa.udel.edu/apa/publications/newsletters/computers.html

A landmark Turing Test，在2000年春天展示，是图灵预测50年内计算机能够通过他的测试的50周年纪念。虽然人们能很快判断出计算机，但有趣的是也有人会误认另一些人是计算机。因此，计算机取代人类还需要一定的时间。http: //www.apa.udel.edu/apa/publications/newsletters/v99n2/computers/chair.asp

Linear Logic Prover，Naoyuki Tamura的基于web的线性逻辑证明器，包括许多其他逻辑证明器的参考、程序（包括PROLOG），以及软件列表：http: //bach.cs.kobe-u.ac.jp/llprover/

逻辑教育项目和研究

Association of Symbolic Logic: Committee on Logic Education。大量逻辑软件和逻辑教学资源的链接。http: //www.math.ufl.edu/~jal/asl/logic _ education.html

Taller de Didáctica de la Lógica。（西班牙语）墨西哥的逻辑教学网页。http: //www.filosoficas.unam.mx/~Tdl/TDL.htm

Aracne（西班牙语）。西班牙和拉丁美洲的逻辑资源。http: //aracne.usal.es/

David Gries的网站。http: //www.cs.cornell.edu/gries/，有很多逻辑特别是新的计算逻辑的资源。同时包括他的一本书的资料，A Logical Approach to Discrete Math，Springer-Verlag，1993。http: //www.cs.cornell.edu/gries/Logic/intro.html

List to discuss logic education。由美国Bucknell大学维护，要注册到列表中，可发送消息给listserv@bucknell.edu，要注册到消息组中，可发消息给logic-l@bucknell.edu

Carnegie Mellon的**Causal and Statistical reasoning**课程，提供因果推理的在线教学资源。可从他们的Causality Lab免费下载，其例子教导学生在接受任何事物之前要进行论证。在今天这一点显得尤为重要，很多人轻信电视上公布的一切，但仅一年以后这些内容就被证明是虚假的。

这也导致了专家系统中真值维护（**Truth Maintenance**）这个重要主题。CLIPS带有真值维护功能，如果一个事实被证明为不正确，所有后继被断言、修改或删除的事实或规则将被撤销，系统恢复到错误事实被断言前的状态。这使得CLIPS的结论是有效的，这一点比很多人都做得好。http: //www.phil.cmu.edu/projects/csr/

The Self-Paced Logic Project，提供和开发软件，为大型的自学式逻辑课程创建和管理一个逻辑问题的测试库。这个问题测试库由程序自动创建，达到每学期不同的目的。即使课程内容没有包括，作者也对他认为逻辑学学生应该掌握的内容给出了一个非常全面的学习指导。主题列表十分全面，可以使学生不会重复问一个问题："我到底需要学习什么才能通过测试?" http: //www.sp.uconn.edu/~py102vc/selfpace.htm

第3章

Probability Web。一个全面的关于概率的资源集合，覆盖了：要点、书籍、中心、组和社团、会议、工作、期刊，数学概要、杂集、新闻组和自动邮件系统、人群、概率主题区、出版商和书店、赞助商、软件、教学资源、最新消息等内容：http: //www.mathcs.carleton.edu/probweb/probweb.html

Chance：辅助讲授Chance一个计量文学课程的资料。Chance课程的目的是使学生能更好地阅读使用了概率和统计的新闻事件。一本很好的免费GNU概率和统计书籍可以从Grinstead和Snell列在下面的站点得到。计算机程序以及单数号习题的答案也可从Chance站点获取。网站中其他的主题包括Chance新闻、Chance课程、视频和语音、教学辅助、最新消息等。相关链接：http: //www.dartmouth.edu/~chance/

概率计算机项目：http://www.wku.edu/~neal/probability/prob.html

许多概率链接，包括要点、自动邮件系统、新闻组、人群、工作，期刊、软件、书籍、会议、出版社和杂项：http://www.maths.uq.oz.au/~pkp/probweb/probweb.html

概率和统计软件：http://forum.swarthmore.edu/probstat/probstat.software.html

公共软件和在线概率与统计出版商，因特网项目：

http://forum.swarthmore.edu/probstat/probstat.projects.html

许多不确定性下决策的软件工具，由 Pittsburgh 大学的决策系统实验室评审，http://www.sis.pitt.edu/~dsl/software.htm

第 4 章 统计、概率和数据挖掘

大量软件资源，关于概率、统计和贝叶斯。某些来自 DOS 环境，但仍能运行。http://archives.math.utk.edu/software/msdos/probability/.html

大量数据集，可以测试数据挖掘软件。http://lib.stat.cmu.edu/datasets

许多与统计书籍相关的数据集。有一些设计用来与 Excel 一起使用，包括马尔柯夫模型。http://www.duxbury.com/cgi-brookscole/course_products_bc.pl?fid=M67&discipline_number=17

各种各样的概率计算器，http://softsia.com/re.php?kw=Probability+Distribution+Calculator

很多有趣的视频，你可以下载关于概率、统计和现实生活问题应用的内容。http://www.dartmouth.edu/~chance/ChanceLecture/download.html

巨大的数学资源，软件、文档、向导、包括幻灯片的教学辅助资料、书籍和相关软件列表。http://bayes.stat.washington.edu/almond/belief.html

各种数学软件的链接列表，http://www.math.fsu.edu/Virtual/index.php?f=21

统计方面国际资源列表，http://gsociology.icaap.org/methods/statontheweb.html

非常全面的各种统计资源链接列表，http://gsociology.icaap.org/methods/statontheweb.html

免费统计软件的集合，http://members.aol.com/johnp71/javasta2.html

马尔柯夫链，http://www.saliu.com/Markov_chains.html

贝叶斯资源

(Korb 04) .Kevin B.Korb and Ann E.Nicholson, **Bayesian Artificial Intelligence**, CRC Press, 2004。这本书包括了不同贝叶斯工具的非常深入的比较，这些工具在一个在线附录中：http://www.csse.monash.edu.au/bai/book/appendix_b.pdf

贝叶斯网络主要信息资源链接可以从 Kevin Patrick Murphy 的主页获得。特别的，他关于向导和软件的链接包括指向许多其他资源、贝叶斯产品比较、软件的链接。http://www.ai.mit.edu/~murphyk/

Microsoft 的免费贝叶斯分析工具（**MSBNx**），贝叶斯网络编辑器和工具箱，可以创建、评估、求值贝叶斯网络。MSBNx 用于办公辅助，他们的技术支持包括故障排除、垃圾过滤：http://research.microsoft.com/adapt/MSBNx/

Bayes Net Toolbox v5 for MATLAB。Cambridge, MA: MIT 计算机科学和人工智能实验室：http://www.ai.mit.edu/~murphyk/Software/BNT/bnt.html

Bayesian Knowledge Discoverer。可以学习贝叶斯信念网的软件。这个商业软件有免费的学习版本：http://kmi.open.ac.uk/projects/bkd/

贝叶斯网络和人工智能的 **Java** 工具：http://bndev.sourceforge.net/

数据挖掘和其他问题的复杂贝叶斯工具，包括免费的试用版本：http://www.bayesia.com/

信念网络操作软件：关于图形化信念函数模型，以及相关的模式例如贝叶斯网络、影响图和概率

图形模型的许多资源：http://archives.math.utk.edu/

贝叶斯和依赖网软件：大量的贝叶斯软件和资源：http://www.kdnuggets.com/software/bayesian.html

大量的贝叶斯和马尔柯夫软件及资源：http://www.cs.toronto.edu/~radford/fbm.software.html

贝叶斯网络和影响图。概率、贝叶斯理论和很多软件的优秀网站：http://www.ia.uned.es/~fjdiez/bayes/

构造贝叶斯信念和影响网络的商业工具：http://norsys.com。优秀的贝叶斯网络在线向导，带有自动诊断、预测、金融风险管理、业务分配、保险、生态系统建模、传感器合成等方面的例子：http://norsys.com/tutorials/netica/nt_toc_A.htm

(Das 99)．Balaram Das，**"Representing Uncertainties Using Bayesian Networks."** 非常详细的贝叶斯网络用于现实战争场景中不确定性情况下的建模和推理等紧急任务应用的报告。比书中的 Prospector 例子复杂的多：http://www.dsto.edfence.gov.au/corporate/reports/DSTO-TR-0918.pdf

第 5 章

Glenn Shafer 的 **Dempster-Shafer** 理论主页。许多资源，例如关于他的工作的文章：http://www.glennshafer.com/。特别的，有一个该理论的简短而明确的解释：http://www.glennshafer.com/assets/downloads/article48.pdf

信任函数和模式识别的免费 **MatLab** 软件。不同模型例如 Dempter-Shafer、神经网络和模糊逻辑的软件：http://www.hds.utc.fr/~tdenoeux/software.htm

Dave Marshall 的好的在线教学资源，包括概率、贝叶斯理论、信任模型、确定性因子、Dempster-Shafer、贝叶斯网络和模糊逻辑：http://www.cs.cf.ac.uk/Dave/AI2/node84.html

Allan Ramsay 的好的在线教学资源。参看链接了解为什么谓词逻辑不适于处理现实世界。涵盖模糊、Dempster-Shafer 以及其他。http://www.ccl.umist.ac.uk/teaching/material/5005/

对象分类的 **Dempster-Shafer** 方法：不是浅显读物，而是 Dempster-Shafter 理论如何应用于导弹防御。http://www.stormingmedia.us/28/2812/A281293.html

什么是 **Dempster-Shafer** 模型？该方法的在线介绍：http://iridia.ulb.ac.be/~psmets/WhatIsDS.pdf

国际模糊系统协会主页。点击他们的链接可以找到模糊逻辑的许多资源：http://www.pa.info.mie-u.ac.jp/~furu/ifsa/

Magdeburg 大学有很多神经模糊软件和资源：http://fuzzy.cs. uni-magdeburg.de/software.html

- 神经模糊控制软件
- 神经模糊数据分析软件（NEFCLASS 现在有 JAVA 版）
- 神经模糊函数近似软件
- 模糊聚类软件
- 教学软件：多层感知训练演示
- 教学软件：学习向量量化演示
- 教学软件：自组织图训练演示
- 教学软件：作为关联记忆的 Hopfield 网络

Christian Borgelt，来自 **Magdeburg** 大学，收集了广泛的免费 **GNU** 软件：http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html。所有的程序都是免费的 GNU 软件，用下面的话发布："所有程序都是免费软件，你可以重新发布他们或者遵循 GNU General Public License 或 GNU Lesser (Library) General Public License 作为免费软件的基础来修改他们（根据程序来决定选用哪一个 license)。"

以下是来自他的页面的列表：

名　称	语　言	描　述
Apriori	C	关联规则归纳/频繁项目集挖掘
Eclat	C	频繁项目集挖掘
MoSS/MoFa	Java	分子子结构挖掘器
MPR	C/Java	多元多项式回归
Dtree	C	决策和回归树归纳
DTView	Java	决策和回归树可视化
Bayes	C	朴素和完全贝叶斯分类器归纳
BCView	C	贝叶斯分类器可视化
NPoss	C	朴素概率分类器归纳
INeS	C	网络结构归纳（图形模型）
MLP	C	多层感知
LVQ	C	学习向量量化
Cluster	C	模糊和概率聚类表工具
Matrix	C	矩阵工具
CHull	C	凸体构造
MLPDemo	C	多层感知演示
LVQDemo	C	学习向量量化演示
SOMDemo	C	自组织图演示
HopfDemo	C	Hopfield 网络演示
Hamster	C	程序设计竞赛环境
Bridgit	C	简单的两人游戏

数据生成器

- BAYES PACKAGE 包括程序 bcdb，可用于产生一个样本的随机数据库，符合朴素或完全贝叶斯分类器所描述的概率分布。
- INES PACKAGE 包括程序 gendb，可用于产生一个样本的随机数据库，符合贝叶斯网络（属性具有有限个域）所描述的概率分布。

模糊逻辑资源

许多模糊逻辑资源的好的页面链接。来自 Ortec Engineering，一个制造模糊逻辑硬件的公司：http://www.ortech-engr.com/fuzzy/reservoir.html#sharks

Fuzzy Logic Laboratorium 每年都有模糊逻辑和数学的主题讨论会：http://www.flll.uni-linz.ac.at/navigation/main_navigation/frame_research.html

Fuzzy Sets & Systems：一个由 Elsevier Science Inc 公司出版的期刊：http://www.elsevier.com/wps/find/journaldescription.cws_home/505545/description

模糊逻辑的 **CLIPS.NRC** 版本：http://ai.iit.nrc.ca/IR_public/fuzzy/fuzzyClips/fuzzyCLIPSIndex.html

James Mathews，"**An Introduction to Fuzzy Logic**"。模糊逻辑的简短指导和如何用 C++ 实现模糊逻辑的例子程序和文档：http://www.generaion5.org/content/1999/fuzzyintro.asp

第 6 章

微软有一个知识管理的巨大分类目录，在：http://www.microsoft.com/windows/catalog

一个非常复杂的工具，**KnowledgeBase.net** 4.0，试图管理所有的 KM。关于 IT 界当前概念的有趣读物：http://www.knowledgebase.net/

个人构造心理学。一个流行的处理人们作为个人科学家概念的理论，不断地生成关于世界的假设和理论。这已经用作通过个人存储网格进行知识获取的基础。许多资源和链接在：http://repgrid.com/pcp/

信息和知识管理社群。具有许多会议、论文、案例分析和软件资源：http://www.ikms.org.sg/resources/index.html

知识管理的大的国际门户，很容易搜索到大量文档，http://www.kmtool.net/

Brint Institute: The Knowledge Creating Company。有许多知识管理各种主题的优秀论文，可以通过其非常好的摘要去阅读那些最感兴趣的文章，http://www.kmbook.com/

组织信息管理课程：模型和平台。很多案例分析和 KM 类型的资源，来自 Chun Wei Choo：http://choo.fis.utoronto.ca/FIS/Courses/LIS2102/LIS2102.slides.html

Soumeya L. Achour, et. al., "A UMLS-based Knowledge Acquisition Tool for Rule-based Clinical Decision Support System Development"：
http://www.pubmedcentral.gov/articlerender.fcgi? tool=pmcentrez&artid=130080

Knowledge acquisition and expert system shell Acquire® 。虽然在专家系统的应用中介绍过这个工具，但这里再次提到它是因为它具有知识获取能力。如果一个工具不是作为人类知识工程师的通用目的，那么它可能善于某种领域的知识构造，例如决策树，它可以通过询问问题和规则来自动构造。这个软件可以从其网站获得试用版：http://www.aiinc.ca

编写规则时要仔细倾听领域专家，避免写出谬论规则。另一方面，如果是针对法律专家系统，则完全可以，因为法律辩论不需要一定是有效的，这只看哪一方的律师更好。谬论辩论的主要问题是人们愿意相信它们。最好的谬论听起来比真理更容易接受。以下都使用了善说服的辩论而成功地使人们相信你是正确的而他们是错误的，即使事实上他们是正确的而你是错误的。正如谬论，修辞是一种能说服人的辩论技巧，结婚之前学会非常有用。以下是谬论的部分列表，注意：即使你不是一个律师或者已婚者，这些都是你说服你的老师你的成绩应是 A，或者说服你的老板你应升职的好的技巧。
http://www.iep.utm.edu/f/fallacies.htm

- Abusive Ad Hominem
- Accent
- Accident
- Ad Baculum
- Ad Consequentiam
- Ad Crumenum
- Ad Hoc Rescue
- Ad Hominem
- Ad Ignorantiam
- Ad Misericordiam
- Ad Novitatem
- Ad Numerum
- Ad Populum
- Ad Verecundiam
- Affirming the Consequent
- Amphiboly
- Anecdotal Evidence
- Anthropomorphism
- Appeal to Authority
- Appeal to Emotions
- Appeal to Force
- Appeal to Ignorance
- Appeal to the Masses
- Appeal to Money
- Appeal to the People
- Appeal to Pity
- Appeal to Consequence
- Argument from Outrage
- Argument from Popularity
- Argumentum Ad
- Avoiding the Issue
- Avoiding the Question
- Bald Man
- Bandwagon
- Begging the Question
- Biased Sample
- Biased Statistics
- Bifurcation
- Digression
- Distraction
- Division
- Domino
- Double Standard
- Either/Or
- Equivocation
- Etymological
- Every and All
- Excluded Middle
- False Analogy
- False Cause
- False Dichotomy
- False Dilemma
- Far-Fetched Hypothesis
- Faulty Comparison
- Formal
- Four Terms
- Gambler's
- Genetic
- Group Think
- Guilt by Association
- Hasty Conclusion
- Hasty Generalization
- Heap
- Hooded Man
- Ignoratio Elenchi
- Ignoring a Common Cause
- Incomplete Evidence
- Inconsistency
- Intensional
- Invalid Inference
- Irrelevant Conclusion
- Irrelevant Reason
- Is-Ought
- Jumping to Conclusions
- Line-Drawing
- Loaded Language
- One-Sidedness
- Outrage, Argument from
- Oversimplification
- Past Practice
- Pathetic
- Perfectionist
- Petitio Principii
- Poisoning the Well
- Popularity, Argument from
- Post Hoc
- Prejudicial Language
- Questionable Analogy
- Questionable Cause
- Questionable Premise
- Quibbling
- Quoting out of Context
- Rationalization
- Red Herring
- Refutation by Caricature
- Regression
- Reversing Causation
- Scapegoating
- Scare Tactic
- Scope
- Secundum Quid
- Self-Fulfilling Prophecy
- Slanting
- Slippery Slope
- Small Sample
- Smear Tactic
- Smokescreen
- Sorites
- Special Pleading
- Specificity
- Stacking the Deck
- Stereotyping
- Straw Man
- Style Over Substance

- Black-or-White
- Circular Reasoning
- Circumstantial Ad Hominem
- Clouding the Issue
- Common Belief
- Common Cause
- Common Practice
- Complex Question
- Composition
- Consensus Gentium
- Consequence
- Converse Accident
- Cover-up
- Cum Hoc, Ergo Propter Hoc
- Definist
- Denying the Antecedent
- Logical
- Lying
- Many Questions
- Misconditionalization
- Misleading Vividness
- Misrepresentation
- Missing the Point
- Modal
- Monte Carlo
- Name Calling
- Naturalistic
- Neglecting a Common Cause
- No Middle Ground
- No True Scotsman
- Non Causa Pro Causa
- Non Sequitur
- Subjectivist
- Superstitious Thinking
- Suppressed Evidence
- Sweeping Generalization
- Syllogistic
- Tokenism
- Traditional Wisdom
- Tu Quoque
- Two Wrongs Make a Right
- Undistributed Middle
- Unfalsifiability
- Unrepresentative Sample
- Untestability
- Weak Analogy
- Willed ignorance
- Wishful Thinking

第 7 章

CLIPS 已经被用于全世界数百个大学，用来讲授专家系统。使用搜索引擎是寻找如何组织典型课程内容和教学资源的最好方法。以下是少量示例网站：

http://www.ghgcorp.com/clips/CLIPS.html，CLIPS 的主页。链向很多资源。

http://www.cs.unc.edu/Admin/Courses/descriptions/275.html

http://www.cs.wpi.edu/~dcb/courses/CS538/