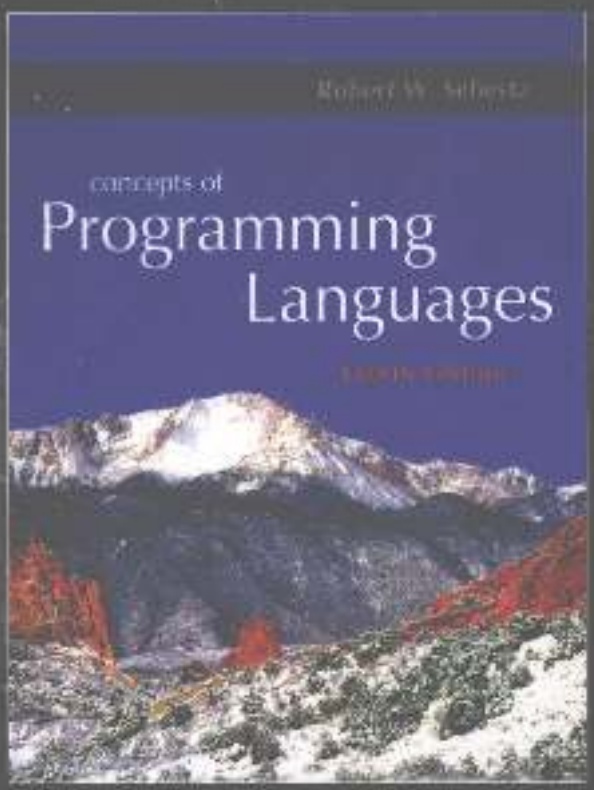
计 算 机 科 学 丛 书

原书第8版

程序设计语言原理

(美) Robert W. Sebesta 著 张勤 王方矩 译
科罗拉多大学科罗拉多斯普林斯分校



Concepts of Programming Languages

Eighth Edition

机械工业出版社
China Machine Press

程序设计语言原理（原书第8版）

本书的主旨是为读者提供对现有的和将来的程序设计语言进行客观评价所需要的方法和思路，增强读者学习新语言的能力并理解语言的实现。本书从学习程序设计语言的原因、常用程序设计语言的演化史、评估程序设计语言结构的标准、以及这些语言基本的实现方法开始讲起，通过不同层于特定语言种类地分析语言结构的设计问题，检测设计选择，以及比较设计可选方案来讲述程序设计语言基本原理。本书并非讲授如何使用一门语言，而是讨论语言的结构、特性及其在各种情景中的设计和实现以及如何根据给定的任务选择合适的语言。

本书特点及新增内容：

- 把程序设计语言Python和Ruby融入相关章节。
- 修改了关于操作语义的内容。
- 新增有关支持Java 5.0和C# 2005泛型类的内容。
- 涵盖了当代语言（包括C#、Java、JavaScript、Perl、PHP、Python和Ruby等）有趣而重要的特性。
- 收录了James Gosling、Larry Wall、Alan Cooper、Bjarne Stroustrup等人的访谈。
- 以Prolog语言为例，剖析了逻辑程序设计语言。
- 讨论了包括Scheme和ML在内的函数式程序设计语言。
- 将面向对象和非面向对象的命令式程序设计语言结合起来讨论。
- 提供了产生现有语言的特定设计选择的历史背景。

作者简介

Robert W. Sebesta 宾夕法尼亚州立大学获得计算机科学博士，拥有30多年的教授计算机科学课程的经验，目前担任科罗拉多大学科罗拉多斯普林斯分校计算机科学系的副教授，ACM和IEEE计算机学会的会员，主要研究方向是设计和评估程序设计语言、编译器设计以及软件测试方法和工具。

www.PearsonEd.com

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

ISBN 978-7-111-23951-2

9 787111 239512

定价：75.00元

计 算 机 丛 书

原书第8版

TP311.1
XB52

程序设计语言原理

(美) Robert W. Sebesta 著 张勇 王方矩 译
特罗拉多大学特罗拉多斯普林斯分校



Concepts of Programming Languages

TP311.1
XB52

机械工业出版社
China Machine Press

本书从为什么学习程序设计语言入手，深入细致地讲解了命令式语言的主要结构及其设计与实现，内容涉及变量、数据类型、表达式和赋值语句、控制语句、子程序、数据抽象机制、支持面向对象程序设计（继承和动态方法绑定）、并发和异常处理等方面，最后两章介绍了三数式程序设计语言和谓辅程序设计语言。

本书内容丰富，原标逻辑，被美国和加拿大多所高等院校采用作为教材。本书既可用做高等院校计算机及相关专业本科生程序设计语言课程的教材和参考书，也可供程序设计人员参考。

Simplified Chinese edition copyright © 2008 by Pearson Education Asia Limited and China Machine Press.

Original English language title: *Concepts of Programming Languages, Eighth Edition* (ISBN 0-321-49362-1) by Robert W. Sebesta, Copyright © 2008.

All rights reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Addison-Wesley.

本书封面贴有Pearson Education（培生教育出版集团）激光防伪标签，无标签者不得销售。

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：01-2007-3086

图书在版编目（CIP）数据

程序设计语言原理（原书第8版）/（美）塞巴斯塔（Sebesta, R. W.）著；张勇，王芳矩译。—北京：机械工业出版社，2008.6

书名原文：Concepts of Programming Languages，Eighth Edition

（计算机科学丛书）

ISBN 978-7-111-23951-2

Ⅰ.程… Ⅱ.①塞… ②张… ③王… Ⅲ.程序语言 高等学校 教材 Ⅳ.TP312

中国版本图书馆CIP数据核字（2008）第053981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：王茂辉

山西新华印业有限公司新华印刷分公司印刷 · 新华书店北京发行所发行

2008年6月第1版第1次印刷

184mm × 260mm · 33.25印张

标准书号：ISBN 978-7-111-23951-2

定价：75.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了绝对性的优势，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，不仅奠定了研究的范畴，还奠基了学术的源泉，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、翻译国外优秀教材上。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、研究及收藏。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，有的还专程为丛书的中译本作序。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，引进全美流行的教学辅导书“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、有的历经二十年而不衰、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

本书是一本在美国、加拿大得到广泛使用的大学教材，适用于计算机科学或计算机工程专业本科二、三年级开设的程序设计语言课程。本书已升级至第8版，多次再版的事实足以说明其受欢迎的程度。一本书的市场价值往往能够反映出它在技术上的价值，本书就是如此。

计算机科学的各个方面都离不开程序设计语言，计算机工作者一生中必然会接触好几种语言：当一种新的语言问世并被广泛接受时，你需要学习这种语言以更新技能；当接手一个新项目时，你必须为这个项目选择一种最合适的实现语言，甚至你可能会为它专门设计并实现一种新的语言。本书并不教授如何使用一种语言，而是讨论程序设计语言的结构与特性、这些结构与特性在不同语言中的设计与实现以及这些结构与特性带给语言的优点与缺点。掌握了这些知识，会让读者在学习新的语言时有一种“似曾相识”的感觉，并很快掌握该语言中的许多特性，当读者需要选择一种语言时，可以根据各种语言的适用性及它们的优缺点做到知“语”善用；当其需要构造一种新语言时，也会知道应该从何处着手来选择出最优的设计实现方案。另外，本书所提供的关于语言“内部”结构的设计与实现的知识，正是大多数介绍语言使用方法的书籍所欠缺的。这种知识能够让读者将一种语言的优点充分地发挥出来，并且避免该语言本身的缺点可能带来的种种问题。

本书的主要内容可分为四部分。第一部分包括第1章和第2章，第二部分包括第3章和第4章，第三部分从第5章到第14章，第四部分包括第15章和第16章。

在第一部分中，第1章介绍一些预备性的基础知识。第2章是对程序设计语言的历史进行有趣而引人入胜的回顾，这一章介绍了一种在第二次世界大战期间由德国开发的功能十分齐全、但鲜为人知的语言——Plankalkul语言，还介绍了Fortran语言的巨大成功及其深远的影响，还包括ALGOL、COBOL、Pascal、Ada、C、C++、Java、C#、JavaScript、PHP、Python和Ruby等语言，所有重要的语言几乎都涉及了。本章为程序设计语言描绘了一个完整的家谱，在这里，各种语言的来龙去脉变成了一幅清晰的画卷。

在第二部分中，第3章是关于语法和语义的描述，语法和语义是语言中的两大要素，这一章讨论对程序设计语言的语法和语义进行形式描述的方法。第4章简明地介绍了程序设计语言的编译原理，这一章是专门为不开设编译技术课程的学校而编写的。然而实际上，即使你打算去啃一本编译技术的大部头书，或者是准备学习一门编译课程，这一章也很值得读一读，它会让你事先获得编译技术的概貌，而不至于一头扎入大量的技术细节当中，导致“不识庐山真面目”。

第三部分集中了本书精华的部分。这一部分详细地剖析语言的各个组成部分设计的实现。程序设计语言的主要组成部分包括变量、类型、表达式、赋值语句、控制结构、子程序、并发、异常处理等，对于语言的每一个组成部分，本书首先讨论其必要性，接着分析设计中必须解决的问题，然后列出了各种不同的解决方法，并评价了各种方法的优缺点，讨论了一些重要语言中具体的设计实现。在读完这一部分之后，一门语言将不再是一个黑盒子，而像是在硬件高手面前打开了盖子的一台计算机，各个部分的特征、优劣以及对整体性能的影响都变得清清楚楚，这些与语言有关的知识对于编写可靠高效的程序将是极有帮助的。例如，如果一位程序员了解他所使用的语言中调用子程序的代价，他就能根据对程序速度和程序模块化的特定要求来决定

是否使用过归子程序。同样，如果一位程序员熟悉他所使用的语言中的变量引用环境，他的程序中由于变量引用错误而产生的“疑难病症”就会少很多。有时，这些知识还能帮助你判断出编译器内部的错误，使你从盲目地排刷那些无事程序的思维中解放出来。

第四部分介绍两种“另类”的语言：以LISP为代表的函数式语言以及逻辑程序设计语言Prolog，它们都是人工智能语言，而此前所讨论的语言都属于命令式语言。命令式语言的程序向机器发出一条条可执行的命令语句。然而，在函数式和逻辑程序设计语言中，程序设计却遵循一种完全不同的思维方式，连算法设计都大相径庭。在这些语言里，没有我们熟悉的、必不可少的赋值语句、循环语句等，使用这些语言时，程序设计不再能够先画框图，然后将每一个框图翻译成一条或几条命令语句，函数式程序设计通过函数调用来解决问题，逻辑程序设计则是通过逻辑推理来解决问题。在这两章中，作者举了几个很好的例子，让你很容易转换思维，接受这两种不同的方式，另外还给出了十分巧妙的算法来解决传统的问题，它们会帮助你拓宽解决问题的思路，也许在下一个项目中，这两种程序设计的知识能够使你在解决问题时另辟蹊径，获取出人意料的成功。

本书还收录了与一批著名的计算机科学家进行的有趣而生动的访谈，其中还有C++、Java、PHP、Perl等语言的作者，他们在程序设计语言上的卓越贡献对计算机事业的发展产生了巨大的影响。从这些访谈之中，我们可以了解到这些科学家们丰富的成长经历、独特的思维方式、对问题的深刻见解，以及他们对未来的科学展望。与这些科学家们进行的这些访谈，向读者们打开了另一扇奇妙的窗子，其形式生动活泼、妙趣横生，希望这些内容能够激励读者，尤其是年轻的读者，走上一条成功的道路。

这是一本通用面很广的书，既可用作大学程序设计语言课程的教材，也可用作自学语言的读物，毕业多年的经验丰富的计算机工作者也可以读一读来更新知识。

译 者

2008年3月

前　言

第8版的变化

本书的目的，总体结构以及写作方式与前面的7个版本保持了一致。其主要目的是介绍当代程序设计语言的主要结构，并为读者提供对已有的程序设计语言和未来的程序设计语言进行客观评估所必需的工具；还有一个目的是通过提供对程序设计语言的深入讨论，以及通过表述一种描述语法的形式化方法，为读者学习编译器的设计做准备。

本书是在第7版的基础加进两种改变演化而来的。首先，本书采用一些较新语言的素材取代了较老的程序设计语言内容，从而保持内容上的新颖。有关Python和Ruby独特的控制结构的讨论加入到第8章。第9章中将函盖Ruby块和迭代。在Python和Ruby中对抽象数据类型支持以及在Java 5.0和C# 2005中对泛型类支持的讨论被在第11章。第12章加入了支持面对对象程序设计（OOP）的Ruby语言的概述。其次，为了增强本书表述的清晰程度，书中的大部分章节都进行了细微的修正。

全书概貌

本书通过讨论各种语言结构的设计问题，讨论最常用语言中这些结构的设计选择，并客观比较各种设计选择，来描述程序设计语言的基本概念。

要对程序设计语言进行认真研究，需要讨论以下相关课题，其中包括描述程序设计语言的语法和语义的形式方法，该课题被概括进本书的第3章。本书还考虑了各种语言结构的实现技术：第4章讨论词法分析和语法分析，第10章讨论了程序链接的实现。其他一些语言结构的实现问题，也在本书的各个不同部分进行讨论。

下面简略说明第8版包括的内容。

章节概述

第1章从学习程序设计语言的目的开始，讨论用于评估程序设计语言及语言结构的标准，以及影响语言设计的主要因素，设计中常用到的权衡方法及其基本的实现方法。

第2章概述书中讨论的大部分重要语言的演化过程。虽然没有完整地描述其一种语言，但对于每一种语言的起源、目的和它的贡献都进行了分析。这种历史回顾十分有价值：它为人们理解当代语言设计的实践和理论基础提供了必要的背景，也为进一步学习语言的设计以及评估提供了动力。除此之外，因为书中的其他章节都不依赖第2章，因而可以将这一章作为完全独立的部分来阅读。

第3章讨论描述程序设计语言语法的主要形式方法，即巴科斯-诺尔范式（BNF）。接着是关于属性文法的描述，该文法描述了语言的静态语义及语法。然后讲解语义描述这一有难度的主题。这里包括对三种最常用语义描述方法的简略介绍，即操作语义、公理语义和指称语义。

第4章介绍词法分析和语法分析。这一章是为那些没有设置编译器设计课程的学校准备的。与第2章相似，这一章也是独立的，可以不依赖书中的其他章节独立学习。

第5章到第14章详细地描述命令式语言主要结构的设计问题。对其中每一种结构，作者列举

和评价了几种范例语言的设计选择。具体讲，第5章包括变量的多种特性，第6章包括数据类型；而第7章则解释表达式及赋值语句，第8章描述控制语句，第9章和第10章讨论子程序及其实现问题。第11章研究数据抽象的机制，第12章提供了关于支持面向对象程序设计语言特征（继承和动态方法绑定）的深入讨论，第13章讨论并发程序单元，第14章介绍异常处理以及事件处理。

最后两章（第15章和第16章）描述两种截然不同的程序设计范型：函数式程序设计及逻辑程序设计。第15章介绍Scheme语言，包括它的一些基本功能、特殊形式、函数形式，并给出用Scheme语言编写的简单函数示例。接下来，简要地介绍ML和Haskell语言，以说明不同形式的函数式语言。第16章介绍逻辑程序设计以及逻辑程序设计语言：Prolog。

写给教师

在位于科罗拉多州斯普林的科罗拉多大学的初级程序设计语言课堂上，我们是这样来使用本书的：我们通常会详细地讲授第1章与第3章。由于第2章没有很难的技术内容，我们只花费很少的课时来讲解，而且如我们在前面提到的，后面所有章节的内容都不依赖于第2章，所以这一章的内容完全可以自学完成。此外，我们单独设置编译器设计课程，所以第4章也不在本课程里讲授。

对于具有丰富的C++、Java和C#语言程序设计经验的学生，第5章到第9章是相对容易的，第10章到第14章则有些挑战性，因而需要进行比较详细地讲授。

对大多数的低年级学生而言，第15章和第16章是全新的。理想情况下，应该对那些学习这两章内容的学生提供 Scheme 和 Prolog的语言处理器。书中提供了充足的资料，指导学生写出简单的程序。

本科生的课程中可以不讲解最后两章的全部内容，但在研究生的课程中则可以讨论这两章的所有内容，此时可以跳过前面几章关于命令式语言的内容。

教辅资料

本书的所有读者都可以从网址www.aw.com/cssupport得到下列的教辅资料：

- 一套教学幻灯片，本书每一章都能获取到相应的PowerPoint®幻灯片。
- PowerPoint®幻灯片包含有本书所有的图片。

为了便于增强课堂教学的效果，配合课程中动手实验，帮助远程教学中学习的学生，我们在网站www.aw.com/sebesta放置了以下资源与大家共享：

- 提供几种语言的小型学习手册（大约100页的辅导材料）。我们希望借助这些手册，给予学生足够的信息来完成许多章节里有关各种语言的练习，使学生们知道如何在其他的语言中编写程序。作者提供的相关网站下包括C++、C、Java以及Smalltalk语言的手册。
- 实行自我评分。学生们可以通过完成多项选择以及填空练习来检测自己对于学完章节的理解程度。

在此声明：练习题的答案只提供给教师（位于www.aw-bc.com/irc的教师资源中心）。请采用本书作为教材的教师按书后的教学支持说明表中提供的联系方式联络Addison-Wesley公司北京办事处索取相关教辅资料或发送E-mail（computing@aw.com以获取更多的信息）。

可用的语言处理器

书中讨论的一些程序设计语言的处理器，以及一些关于程序设计语言的信息，能够通过下列网址获得：

C、C++、Fortran和Ada	gcc.gnu.org
C#	microsoft.com
Java	java.sun.com
Haskell	haskell.org
Scheme	www.plt-scheme.org/software/drscheme
Perl	www.perl.com
Python	www.python.org
Ruby	www.ruby-lang.org/en/

实际上，几乎所有浏览器都包含了Java Script，几乎所有Web服务器都包含了PHP。相关网站也包含了这些信息。

致谢

许多审阅者对本书提出了宝贵的建议，按字母顺序排列如下：

I-ping Chu，德宝大学

Amer Diwan，科罗拉多大学

Stephen Edwards，弗吉尼亚科技大学

Nigel Gwee，路易斯安那州立大学

K.N. King，佐治亚州立大学

Donald Kraft，路易斯安那州立大学

Simon H. Lin，加利福尼亚州立大学Northridge分校

Mark Llewellyn，中佛罗里达大学

Bruce R. Maxim，密执安大学迪尔伯恩分校

Gloria Melara，加利福尼亚州立大学Northridge分校

Frank J. Mitropoulos，诺瓦东南大学

Euripides Montagne，中佛罗里达大学

Bob Neufeld，科奇托州立大学

Amar Raheja，加利福尼亚科技大学

Hossein Saiedian，堪萨斯大学

Neelam Soundarajan，俄亥俄州立大学

Paul Tymann，罗彻斯特理工学院

Cristian Videira Lopes，加利福尼亚大学Irvine分校

Salih Yurttas，得克萨斯州A & M大学

还有许多其他人也对本书前几个版本提出过宝贵意见，我也向他们致以深深的谢意：Vicki Allan，Henry Bauer，Carter Bays，Manuel E. Bermudez，Peter Brouwer，Margaret Burnett，Paosheng Chang，Liang Cheng，John Crenshaw，Charles Dana，Barbara Ann Griem，Mary Lou Haag，John V. Harrison，Eileen Head，Ralph C. Hilzer，Eric Ioanis，Leon Johnban，Hikyoo Koh，Jiang B. Liu，Meilin Lu，Jon Mauney，Robert McCoard，Dennis L. Mumaugh，Michael G. Murphy，Andrew Oldroyd，Young Park，Rebecca Parsons，Steve J. Phelps，Jeffery Popyack，Raghvinder Sangwan，Steven Rapkin，Hamilton Richard，Tom Sager，

Joseph Schell、Sibylle Schupp、Mary Louise Soffa、Neelam Soundarajan、Ryan Stansifer、Steve Stevenson、Virginia Teller、Yang Wang、John M. Weiss、Franck Xia和Salih Yurnas。

编辑Matt Goldstein、编辑助理Maurene Goo和Addison-Wesley出版社的高级生产主管Marilyn Lloyd以及在WestWords的Tammy King和Bethann Thompson，我也感谢他们对本书快速出版所做的努力。

作者简介

Robert Sebesta是科罗拉多大学科罗拉多斯普林分校计算机科学系的荣誉副教授。Sebesta教授获得了博尔德、科罗拉多大学的应用数学学士学位和宾夕法尼亚州立大学计算机科学硕士学位和博士学位。他拥有超过36年的教授计算机科学课程的经验，主要研究方向是程序设计语言设计与评估、编译器设计以及软件测试方法和工具。

Robert W. Sebesta

目　录

出版者的话
专家指导委员会
译者序
前言

第1章　基本概念 ……1
1.1　学习程序设计语言原理的意义 ……1
1.2　程序设计应用领域 ……3
1.3　语言评价标准 ……4
1.4　影响语言设计的因素 ……13
1.5　语言分类 ……15
1.6　语言设计中的权衡 ……16
1.7　实现方法 ……17
1.8　程序设计环境 ……21
小结*文献注释*复习题*练习题 ……22
第2章　主要程序设计语言的发展 ……24
2.1　Zuse的Plankalkül语言 ……24
2.2　最小硬件的程序设计：伪代码 ……26
2.3　IBM 704计算机与Fortran ……28
2.4　函数式程序设计语言：LISP ……32
2.5　迈向成熟的第一步：ALGOL 60 ……36
2.6　商务应用计算机化：COBOL ……40
2.7　分时操作的开始：BASIC ……44
2.8　用途广泛的语言：PL/I ……47
2.9　两种早期的动态语言：APL和SNOBOL ……50
2.10　数据抽象的开始：SIMULA 67 ……51
2.11　正交性语言的设计：ALGOL 68 ……52
2.12　早期ALGOL系列语言的后代产品 ……53
2.13　基于逻辑的程序设计：Prolog ……55
2.14　历史上规模最大的语言设计：Ada ……59
2.15　面向对象的程序设计：Smalltalk ……62
2.16　结合命令式与面向对象的特性：C++ ……64
2.17　一种基于命令式的面向对象语言：Java ……66
2.18　脚本语言：JavaScript、PHP、Python和Ruby ……69
2.19　一种基于C的新世纪语言：C# ……72
2.20　标志与程序设计混合式语言 ……74
小结*文献注释*复习题*练习题*程序设计练习题 ……76
第3章　描述语法和语义 ……79
3.1　概述 ……79
3.2　描述语法的普遍问题 ……80
3.3　描述语法的形式方法 ……81
3.4　属性文法 ……91
3.5　描述程序的意义：动态语义 ……95
小结*文献注释*复习题*练习题 ……108
第4章　词法分析和语法分析 ……112
4.1　概述 ……112
4.2　词法分析 ……113
4.3　语法分析问题 ……117
4.4　递归下降语法分析 ……119
4.5　自底向上语法分析 ……126
小结*复习题*练习题*程序设计练习题 ……131
第5章　名字、绑定、类型检测和作用域 ……135
5.1　概述 ……135
5.2　名字 ……135
5.3　变量 ……137
5.4　绑定的概念 ……139
5.5　类型检测 ……146
5.6　强类型化 ……147
5.7　类型等价 ……148
5.8　作用域 ……151
5.9　作用域与生存期 ……157

5.10 引用环境 ……157
5.11 命名常量 ……159
小结*复习题*练习题*程序设计练习题 ……160
第6章 数据类型 ……166
6.1 概述 ……166
6.2 基本数据类型 ……167
6.3 字符型类型 ……169
6.4 用户定义的序数类型 ……173
6.5 数组类型 ……176
6.6 关联数组 ……186
6.7 记录类型 ……189
6.8 联合类型 ……192
6.9 指针类型与引用类型 ……195
小结*文献注释*复习题*练习题*程序设计练习题 ……204
第7章 表达式与赋值语句 ……208
7.1 概述 ……208
7.2 算术表达式 ……208
7.3 重载操作符 ……214
7.4 类型转换 ……216
7.5 关系表达式和布尔表达式 ……219
7.6 短路求值 ……221
7.7 赋值语句 ……222
7.8 混合模式赋值 ……225
小结*复习题*练习题*程序设计练习题 ……226
第8章 语句层次的控制结构 ……229
8.1 概述 ……229
8.2 选择语句 ……230
8.3 循环语句 ……238
8.4 无条件分支 ……248
8.5 守卫的命令 ……249
8.6 结论 ……252
小结*复习题*练习题*程序设计练习题 ……253
第9章 子程序 ……256
9.1 概述 ……256
9.2 子程序的基本原理 ……256
9.3 子程序的设计问题 ……262
9.4 局部引用环境 ……263
9.5 参数传递方法 ……265
9.6 子程序名作为参数 ……280
9.7 重载子程序 ……281
9.8 通用子程序 ……282
9.9 函数的设计问题 ……287
9.10 用户定义的重载操作符 ……288
9.11 协同程序 ……288
小结*复习题*练习题*程序设计练习题 ……290
第10章 实现子程序 ……294
10.1 调用与返回的一般语义 ……294
10.2 实现“简单”子程序 ……294
10.3 实现具有栈动态局部变量的子程序 ……296
10.4 嵌套子程序 ……306
10.5 块 ……305
10.6 实现动态作用域 ……306
小结*复习题*练习题 ……309
第11章 抽象数据类型和封装结构 ……313
11.1 抽象概念 ……313
11.2 数据抽象介绍 ……314
11.3 抽象数据类型的设计问题 ……315
11.4 语言示例 ……318
11.5 参数化的抽象数据类型 ……328
11.6 封装结构 ……331
11.7 命名封装 ……333
小结*复习题*练习题*程序设计练习题 ……336
第12章 支持面向对象的程序设计 ……339
12.1 概述 ……339
12.2 面向对象程序设计 ……339
12.3 面向对象语言的设计问题 ……341
12.4 Smalltalk对面向对象程序设计的支持 ……344
12.5 C++对面向对象程序设计的支持 ……346
12.6 Java对面向对象程序设计的支持 ……354
12.7 C#对面向对象程序设计的支持 ……357
12.8 Ada 95对面向对象程序设计的支持 ……358
12.9 Ruby对面向对象程序设计的支持 ……361

12.10 JavaScript的对象模型 ……364
12.11 面向对象结构的实现 ……366
小结*复习题*练习题*程序设计练习题 ……368
第13章 并发 ……372
13.1 概述 ……372
13.2 子程序层次并发的介绍 ……374
13.3 信号量 ……377
13.4 管程 ……380
13.5 消息传递 ……382
13.6 Ada对并发的支持 ……382
13.7 Java线程 ……391
13.8 C#线程 ……397
13.9 语句层次的并发 ……398
小结*文献注释*复习题*练习题*程序设计练习题 ……400
第14章 异常处理和事件处理 ……403
14.1 异常处理概述 ……403
14.2 Ada中的异常处理 ……407
14.3 C++中的异常处理 ……412
14.4 Java中的异常处理 ……416
14.5 事件处理概述 ……423
14.6 Java的事件处理 ……424
小结*文献注释*复习题*练习题*程序设计练习题 ……427
第15章 函数式程序设计语言 ……431
15.1 概述 ……431
15.2 数学函数 ……431
15.3 函数式程序设计语言的基础 ……433
15.4 第一种函数式程序设计语言：LISP ……434
15.5 Scheme概述 ……436
15.6 COMMON LISP ……448
15.7 ML ……449
15.8 Haskell ……451
15.9 函数式语言的应用 ……454
15.10 函数式语言和命令式语言的比较 ……455
小结*文献注释*复习题*练习题*程序设计练习题 ……456
第16章 逻辑程序设计语言 ……459
16.1 概述 ……459
16.2 谓词演算的简短介绍 ……459
16.3 谓词演算与定理证明 ……462
16.4 逻辑程序设计概述 ……463
16.5 Prolog的基础 ……464
16.6 Prolog的基本元素 ……465
16.7 Prolog的缺陷 ……475
16.8 逻辑程序设计的应用 ……480
小结*文献注释*复习题*练习题*程序设计练习题 ……481
参考文献 ……483
索引 ……493

第1章 基本概念

我们在深入学习程序设计语言原理之前，需要考虑几个基本的概念。首先我们将讲解，为什么计算机科学专业的学生以及专业软件开发人员需要学习语言的设计和评估的一般原理。这种讨论对于那些认为只要有一、两种程序设计语言的工作经验就足够了的计算机科学人员来说，是十分有价值的。接下来，我们将简略描述程序设计的主要范畴，然后，由于本书将评价各种程序设计语言的构成和特性，我们将会列出用于判别程序设计语言优劣的标准。紧接着，我们将讨论两个影响程序设计语言的重要因素，即计算机体系结构以及程序设计方法学。之后，我们将给出程序设计语言的不同分类。再接着，我们将阐述在语言设计中必须考虑的几种主要权衡取舍方法。

由于本书也叙述程序设计语言的实现，因而我们在这一章中概括了最常用的程序设计语言实现的方法。最后，我们将简略地描述几个程序设计环境的例子并且讨论这些环境因素对于软件产品的影响。

1.1 学习程序设计语言原理的缘由

学生们会很自然地问，他们如何能够从程序设计语言原理的学习中得益，毕竟在计算机科学领域里，还有其他大量的题材值得花费时间认真学习。下面，是我们认为计算机科学人员通过学习程序设计语言原理能够获得的种种益处：

- **增进表达思想的能力。**人们普遍认为，我们思维的深度受我们用来进行思想交流的语言以及这种语言的表达能力的影响。对自然语言而言，那些只掌握有限语言的人，其思维的复杂程度，特别是其抽象思维的程度，必然受到局限。换言之，人们对不能口头或笔头描述的事物结构必定很难将其概念化。

程序员在开发软件的过程中也同样受制于这个法则。用来开发软件的程序设计语言制约着程序员能够应用的控制结构、数据结构和数据抽象的类型，因而也制约着他们所能够设计的算法。了解多种程序设计语言的特性，能够使程序员在软件开发的过程中减少这种受限性。程序员可以通过学习新的程序设计语言的结构来拓宽软件开发的思路。

有人或许会提出不同意见，学习其他程序设计语言所具备的功能，并不能帮助一个正用某种不具备这些功能的程序设计语言工作的编程人员。然而，这种论点事实上不成立。因为一些程序设计语言的结构，常常可以使用其他不直接支持这些结构的语言来构造。例如，一个学会了Perl（Wall *et al.*, 2000）语言中关联数组结构及使用的C程序员，就可能用C设计出模拟关联数组的结构来。换句话说，学习程序设计语言的原理，将培养评价语言特性的能力，并将鼓励程序人员运用这些语言特性。

- **增强选择适用语言的能力。**许多专业软件开发人员只受过有限的正规计算机科学教育，往往是通过自学，或者通过公司或单位的培训课程。这样的培训往往只教会他们一种或者是两种与当前工作项目直接相关的语言。另外的许多程序人员是在很早以前受过正规的训练。他们当时学习的语言已经不再使用，而许多当今在程序设计语言中已经广泛使用的特性那时还没有出现。这样就导致许多程序员当需要为新的项目选择程序设计语言

时，他们宁可继续使用自己最为熟悉的那种语言，尽管这种语言已经很不适宜于新的项目。如果这些程序员熟悉许多种语言和语言结构，他们便能够针对实际问题更好地选择更适当的语言。

许多的语言特性可以在其他不具备这些特性的语言中构造出来的事实，丝毫不能削弱设计具有一系列最佳特性语言的重要性。一种已经嵌入某种语言的特性，总是优于对该特性的仿制，仿制的特性往往不够优雅、方便，而且在一种并不支持该特性的语言中也不够安全。

- **增强学习新语言的能力。**计算机程序设计仍旧是一个相对年轻的学科，其设计方法学、软件开发工具，以及程序设计语言，都还处于不断演变的状态之中。这使得软件开发成为一种激动人心的专业，但这同时也意味着不断学习成为关键所在。学习一种新的程序设计语言可能是一个漫长而艰难的过程。这对于那些只习惯于一种或者两种语言，并且从未认真钻研过程序设计语言基本原理的人来说尤其如此。一旦透彻地掌握了程序设计语言的基本概念，就不难理解这些基本原理是如何地融入我们所学习的语言的设计之中，例如，理解面向对象程序设计原理的程序人员，较之从未运用这些原理的程序人员，能够更为轻松地学习Java语言（Arnold *et al.*，2000）。

人们在使用自然语言的时候也会发生同样的现象，你对母语的语法理解得越好，你会发现
3 学习第二种语言越容易。更为甚之，学习第二种语言还能够帮助你更好地理解母语。

TIOBE程序设计社区发布了一个指示程序设计语言相对流行程度的排行榜（http://www.tiobe.com/tiobe_index/index.htm）。例如，在2007年1月的排行榜上，Java、C和C++是三种最常使用的语言。然而，数十种其他语言同时也应用得相当广泛。排行榜数据也说明了程序设计语言使用的分布情况总是在改变。正在使用的语言列表的长度和统计的动态变化暗示了每个软件开发人员必须频繁地学习不同的语言。

最后，对于进行实际工作的程序设计人员来说，关键是透彻理解程序设计语言的术语以及基本概念。这样，就能阅读和理解程序设计语言手册，以及语言和编译器的说明书和其他文件。它们是选择和学习一门语言的信息源。

- **更好地理解实现的重要性。**在学习程序设计语言原理时，影响这些原理的实现方法是十分有趣也是十分必要的课题。常常，人们对程序设计语言实现方法的理解，会使得他们容易理解一种语言为什么要这样设计，而这种理解能够进一步催生更加明智地、按照设计目的来运用语言的能力。通过理解程序设计语言组成结构的种种选择，并且明了这些选择所拥有的结果，我们就能成长为更加优秀的程序设计人员。

往往只有懂得有关的程序设计语言实现细节的程序员才可能发现和改正某些类型的程序错误。理解语言实现问题的另一个好处是，它会使我们明了计算机是怎样执行各种程序设计语言结构的。在某些情况下，有关实现问题的知识能够为选择相对高效的一种结构编写程序提供线索。例如，不懂得递归调用是如何实现的程序员，常常不知道递归算法比等价的迭代算法要慢得多。

因为本书只教授一些实现问题的知识，所以上述两段内容也适用于对学习编译器设计的解释。

- **更好地使用已知的语言。**当代许多语言都是庞大的和复杂的，因此程序员熟悉和使用所采用语言的所有特性是很少见的。通过学习程序设计语言原理，程序员能够学习到他们已采用语言的以前未知和未使用的部分特性，然后开始使用那些特性。
- **促进整个计算科学的发展。**最后，从计算科学的整体上来看，可以证明学习程序设计语
4 言原理的必要性。虽然我们通常可以确定某种程序设计语言得以广泛应用的原因，但许多人在同领中都认为，最广泛流行的语言在当时并不总是最好的语言。在某些情况下，

种语言被广泛使用，部分原因是由于选择语言的决策人员对程序设计语言的一般原理不够熟悉。

例如，许多人认为，20世纪60年代初期，如果ALGOL 60（Backus *et al.*，1962）取代了Fortran（INCITS/ISO/IEC，1997）的话，情形会好得多，因为ALGOL 60语言更优雅，它的控制语句也比Fortran的好得多，之所以没能如此，是由于当时许多程序员以及软件开发的管理人员没有清晰理解ALGOL 60的设计原则。他们感觉ALGOL 60的说明书太难读（的确如此），又更难懂。他们不能够欣赏它的模块结构，递归结构以及出色的控制语句，因而不能体会到ALGOL 60超越Fortran的好处。

当然，如我们将在第2章里介绍的，还有其他许多的原因也使得ALGOL 60没能为人们所接受。然而可以肯定的是，当时的开发人员没有领悟到这种语言的种种优点，这起了决定性的作用。

一般而言，如果选择语言的决策人员具有全面的知识，会选择好的语言取代差的语言。

1.2 程序设计应用领域

今天计算机已经大量地应用于现代社会的各个领域，从控制核能发电厂到提供移动电话中的电子游戏。由于计算机应用领域的千差万别，人们开发了用于不同目的的程序设计语言。在这一节中，我们将简略地讨论几个计算机应用领域，以及与其相关的程序设计语言。

1.2.1 科学应用

20世纪40年代出现的第一台数字计算机曾经用于科学应用，并且事实上是为科学应用而发明的。科学计算的特点是，数据结构简单，但具有大量浮点数的算术运算。其中最常用的数据结构是数组和矩阵，而最常用的控制结构是计数循环和选择。早期为科学计算而发明的高级语言就是为了满足这种需要而设计的。那个时候高级语言的对手是汇编语言，共而高效率是其设计的基本要点。第一种科学应用语言是Fortran。ALGOL 60及其绝大多数的后代语言，虽然也是为其他相关的用途而设计的，但主要是以科学计算为主要目的。对于一些以计算效率为基本要点的科学应用而言，如在20世纪50年代和60年代较为普遍的那些科学应用，后来的程序设计语言没有哪一种在效率方面明显优于Fortran。

1.2.2 商务应用

计算机在商务上的应用始于20世纪50年代。人们为此开发了专门用途的计算机，随之以特定的程序设计语言。第一种成功的高级商务语言是初版出现于1960年的COBOL（ISO/IEC，2002）。至今这种语言仍然是这一应用中最为广泛使用的程序设计语言。商务语言的特征有：有为产生详尽报表的机制，能以精确的方式描述与存储十进制数以及字符数据，以及能够进行十进制算术运算的能力。

商务应用语言中，除了COBOL的发展与演化以外，其他开发十分有限。因而本书仅仅包括了对于COBOL语言结构的有限讨论。

1.2.3 人工智能

人工智能（AI）是计算机应用的一个广泛领域，它的特征是使用符号运算而非数值运算。符号运算指的是，我们处理的是由名字而非数字组成的符号。对于符号运算，采用数据链表结构比

用数组结构更为方便。这种类型的程序设计，有时需要比其他程序设计领域拥有更大的灵活性。例如在一些人工智能的应用中，能够在程序运行的过程中创立并执行程序段的能力将带来方便。

第一种为人工智能应用而开发并被广泛运用的程序设计语言是产生于1959年的函数式语言LISP（McCarthy et al., 1965）。大多数1990年以前的人工智能应用程序，都是用LISP语言或者与其相近的一种程序设计语言编写的。然而在20世纪70年代的初期，出现了实施于一些人工智能应用的另外一种方法，即使用Prolog（Clocksin and Mellish, 2003）语言的逻辑程序设计。更后来，人们使用诸如C语言之类的科学语言来编写一些人工智能的应用。本书将于第15章和第16章分别介绍LISP的一种方言Scheme（Dybbig, 2003）以及Prolog语言。

1.2.4 系统程序设计

计算机系统的操作系统，以及所有在计算机系统中支持程序设计的工具，被统称为系统软件。系统软件几乎总在运行着，因此必须要有很高的执行效率。因而，用于这个领域的语言必须提供快速执行的能力。更进一步讲，这种语言必须要有低层次的语言特性，这样才能编写连接外部设备的软件界面。

在20世纪60年代与70年代，一些计算机制造厂商，如IBM、Digital和Burroughs（现在的UNISYS），为他们自己机器上的系统软件开发了专用的面向机器的高级语言。用于IBM大型机
6 的语言是PL/S，它是PL/I的一种方言；用于Digital机器的语言是BLISS，它是一种仅仅高于汇编语言的程序设计语言；用于Burroughs机器的语言是一种扩展的ALGOL语言。

UNIX操作系统几乎全部是用C（ISO, 1999）来编写的，这使得UNIX系统很容易移植到不同的机器上。C的一些特征使得它非常适合于系统程序设计：它是低级语言，它有很高的执行效率，而且它没有给用户增加许多安全限制的额外负担，系统程序员往往是优秀的程序人员，他们常常认为不需要这种限制。然而也有一些非系统程序员认为，用C来编写大规模的、重要性程度高的系统软件危险性太大。

1.2.5 万维网语言

世界万维网是由一系列兼容语言来支持的，包括从并不是程序设计语言的标志语言，如XHTML，到通用目的的程序设计语言，如Java。由于对于动态万维网内容的普遍需要，一些计算功能常常被包括进表示内容的技术之中。这种技术可以通过将程序设计代码嵌入XHTML文件来实现。被嵌入的代码经常是一种脚本语言，如JavaScript，或者PHP。反过来，XHTML文件也可以要求运行万维网服务器上的一个单独程序来提供动态内容，而这种服务器上的程序则完全可以使用任何程序设计语言来编写。也有一些类似标志语言的语言，这些被扩展的语言包括了一些控制文档处理的结构。关于这些，我们将在1.5节和第2章中进行讨论。

1.3 语言评估标准

如前所述，本书的目的是详细探讨程序设计语言中各种结构及功能中所包含的基本原理。我们也要评价语言特性，尤其要将注意力集中于这些特性对于软件的开发过程（包括软件的维护过程）的影响。要做到这一点，需要有一系列的评估标准。但这样的一系列标准必定有争议，任何两位计算机科学家对某些给定语言特性的价值都不可能完全达成一致。尽管具有如此大的争议，但是绝大部分计算机科学工作者赞同下面小节所讨论的这些标准的确是重要的。

7 表1-1中列举了影响4种最重要标准中的3种的语言特性，而关于这些标准的本身，将在接下

来的几个小节里进行讨论。[⊖]注意，只有最重要的特性在表中列出来了，并在随后的小节中将对其进行讨论。根据个人判断，也能决定特性的重要性，实际上，所有表中位置都能用"填充。

注意这里的一些标准，如可读性，其范围广泛并具有一定程度的模糊性。而另外的一些则是特定的语言结构，如异常处理。尽管在这里的讨论可能会让人理解为所有的这些标准都是同等重要的，但这并不是我们的初衷，而且事实上也显然并非如此。

表1-1 语言的评估标准和影响这些评估标准的语言特征

语言特征	标准		
	可读性	可写性	可靠性
简单性	*	*	*
正交性	*	*	*
控制结构	*	*	*
数据类型与数据结构	*	*	*
语法设计	*	*	*
支持抽象		*	*
表达性		*	*
类型检测			*
异常处理			*
别名机制			*

1.3.1 可读性

判断一种程序设计语言优劣的一个最重要的标准，是用它所编写的程序好读、好懂的程度。在1970年之前，人们普遍认为软件开发就是编写代码。当时考虑程序设计语言的主要特征是效率与机器的可读性，而语言中结构的设计更多地是从机器的角度，而较少考虑计算机用户的因素。然而自20世纪70年代提出了软件生命周期概念（Booch，1987）以后，编程阶段为较为次要的角色，而软件维护则被认为是软件生命周期中的主要部分，尤其是从成本的角度而言。由于软件维护的难易在很大程度上取决于程序的可读性，所以可读性就成了衡量程序以及程序设计语言质量的重要标志。这曾是程序设计语言发展中的一个重要转折点，程序设计语言的关注点明显地转移，从面向机器转向了面向人类(用户)。

必须将可读性放置于问题领域中进行考虑。例如，如果描述一种计算问题的程序是用某种语言编写的，而这种语言并非是为进行这种类型的计算而设计的，这个程序可能就会不自然，也很复杂，因而就会特别难读。

下面我们将会阐述影响程序设计语言的可读性的特征。

1.3.1.1 整体简单性

一种程序设计语言的整体简单性极大地影响着它的可读性。一种具有大量基本结构的语言较只有少量基本结构的语言要难学得多。当程序员们必须使用一种"大"的语言时，他们往往趋向于只学习这种语言的一部分内容，即该语言的一个子集，而忽略它的其他特性。人们的这种学习模式，有时成了语言设计者们将大量的语言结构加进语言的借口，然而这是不正确的。当程序的编写者学会的语言子集与阅读者所熟悉的语言子集不相同时，可读性的问题就出现了。

程序设计语言的第二种复杂特征是其特性的多样化，即某一种特定的操作存在着多种实现

⊖ 第四个重要标准是代价，它没有在表中列出，因为它有几个与这个标准关联度较低，并且与影响它们的特性高关联度地依赖。

方式。例如在Java中，用户可以用四种不同方法来实现一个简单整数变量的增值：

```
count = count + 1
count += 1
count++
++count
```

尽管后两条语句之间，以及后两条语句与前两条语句之间，在某些情形之下存在细微的差异，但当作为独立表达式使用时，这四条语句的意义都是一样的。我们将在第7章中讨论这些差异。

第二种可能出现的问题是运算符重载，即单个运算符被赋予多种运算意义。虽然这是一种有用的特性，但是如果我们允许程序的编写人员给运算符赋予他们自己的运算意义，而他们又不能谨慎所为的话，就会降低程序的可读性。例如，人们显然同意将加法运算符"+"重载，同时来做整数与浮点数的加法。事实上这样的运算符重载，通过减少程序设计语言中运算符的数量简化了语言。但是，假设某编程人员将加号置于两个一维数组的操作数之间，并定义加号的意义为计算这两个一维数组所有元素之和。由于这样的用法与通常的矢量加法完全不同，无论程序的编写者还是阅读者都会对上述加法程序产生不解。另一个更为极端的程序混淆示例是，某用户置加号于两个矢量操作数之间，并定义该加号的意义为计算这两列矢量相应第一个元素之差。运算符重载的问题还将在第7章里作进一步讨论。

过分追求语言的简单性也不妥。例如，下一节就要讲到的汇编语言，该语言中多数语句的形式及语义都是简单性的典范。然而，这种极端的简单性导致了汇编语言较差的可读性。由于汇编语言缺乏较为复杂的控制语句，它的程序结构的清晰性就欠缺。由于汇编语言的语句简单，编写类似功能的程序时，往往需要使用比高级语言多得多的语句。一些不具备足够的控制结构及数据结构的高级语言，也会具有同样的问题。

9

1.3.1.2 正交性

程序设计语言的正交性指的是，使用该语言中一组相对少量的基本结构，经过相对少的结合步骤，可以构成该语言的控制结构与数据结构。而且，它的基本结构的任何组合都是合法的和有意义的。例如，我们来考虑数据类型，假设一种语言具有四种基本数据类型（即整数、浮点数、双精度数和字符），它还具有两个类型操作符（即数组和指针），如果这两个类型操作符能够作用于自身以及这些种基本数据类型，大量的数据结构就能够由此而被定义出来。

正交语言特性的意义是独立于它在程序中出现的上下文的。（正交这个名词来自于正交向量的数学概念，正交向量是相互独立的。）语言的正交性来自于它的基本结构的对称关系。指针应该能够指向任何类型的变量或者数据结构。缺乏正交性会导致语言规则的异常。例如，如果不容许指针指向数组类型，许多可能性就被消除。

下面，我们将通过比较用于IBM大型机和VAX超级小型机系列的汇编语言的一个方面，说明作为语言设计的原则，应该如何使用正交性。让我们考虑一种简单情形：将两个置于存储器或寄存器的32位整数数值相加，并用相加之和来替代其中的一个数值。IBM大型机有两种指令来进行这种运算，它们的形式是

```
A  Reg1, memory_cell
AR Reg1, Reg2
```

这里的Reg1和Reg2代表两个寄存器。这两个指令的语义分别为

```
Reg1 ← contents(Reg1) + contents(memory_cell)
Reg1 ← contents(Reg1) + contents(Reg2)
```

而两个32位整数值的VAX加法指令为

```
ADD  operand_1, operand_2
```

该指令的语义为

```
operand_2 ← contents(operand_1) + contents(operand_2)
```

在这种情形中，两个操作数（operand）中任何一个都可以是寄存器或者存储单元。 10

VAX指令的设计是正交的，它的一条指令可以将寄存器或者存储单元作为其两个操作数，有两种方式可以来说明这两个操作数，而且它们能以任意方式组合。IBM的设计不是正交的，在四种可能性中，只有其中两种操作数的组合是合法的，并且这两种合法组合需要两条不同的指令，即A和AR。IBM的设计具有更多的限制性，所以它的可写性较差。例如，你无法将两个数值相加，并将其和存到一个存储单元上去；进而，由于IBM的设计的这种限制，并额外需要一条指令，使其较难掌握。

语言的正交性与语言的简单性是紧密相关的：程序设计语言的正交性设计得越好，该语言规则中的异常情况就会越少。较少的异常意味着设计中的较高程度的规范性，因而这样的语言较容易被人们学习、阅读和理解。任何学习过较多英语的人，一定领教过了它频繁的规则异常（例如，i总是在e前，除非在c后，等等）。

作为高级语言缺乏正交性的例子，我们考虑C语言中下面的一些规则以及一些异常。尽管C具有数组和记录（struct）这两种结构化的数据类型，然而记录可以从函数中返回，而数组则不能。一种结构的成员可以是任意数据类型，但不能是void或者是同一种类型的结构。一个数组的元素可以是任意数据类型，但不能是void或者函数。参数是按值传递的，但数组参数却是按地址传递的（因为在C程序中出现的不带下标的数组名称会被解释为该数组的第一个元素的地址）。

作为环境依赖性的一个例子，考虑这样一条C表达式

```
a + b
```

这种表达式通常意味着从存储空间取出a和b的数值，并将它们相加。然而，如果a碰巧是一个指针，那么这就会影响到b的数值。例如，如果a指向一个有四个字节长的浮点数值，那么b的数值就必须在与a相加之前先被扩大——在这里是被4相乘。因而a的类型影响到对b数值的处理。b所在的环境影响着它本身的意义。

过分地追求的正交性也会产生问题。最为正交的程序语言或许是ALGOL 68（van Wijngaarden *et al.*, 1969），ALGOL 68语言中的每一种结构都具有类型，并且这些类型都不存在任何限制。此外，绝大多数结构都产生数值。这种结合的自由导致了这种语言的极为复杂的程序结构。例如，条件语句可以与变量声明或其他语句一起出现在赋值语句的左面，只要它的结果是一个地址。这种极端的正交性会导致不必要的复杂性。更进一步，由于此类语言需要大量的基本结构，高度的正交性导致过量的各种结合，即使这些结合都是简单的，它们绝对的大数量也会导致语言的复杂性。 11

因而语言的简单性至少部分是由于相对少量的基本结构所产生的结合，以及限量地运用正交性原理。

部分人认为，函数式语言是简单性与正交性的有机结合。函数式语言，如LISP，主要是通过将函数应用到指定参数上来进行运算的。与之相反，在C、C++及Java这类命令式语言中，运算通常以变量及赋值语句为特征。函数式语言提供了最大可能的整体简单性，因为它能使用单个结构（即函数的调用）来完成任何运算。而多个函数调用又能以简单的方式相结合。函数式语言如此地简单漂亮，这就是为什么它吸引了一些程序设计语言的研究人员，将这种语言作为

首选来替代复杂的非函数式语言，如C++语言。然而其他的因素阻碍了函数式语言的更广泛的应用，如效率等。

1.3.1.3 控制语句

20世纪50年代和60年代的一批程序设计语言由于缺乏控制语句，导致了很差的可读性。20世纪70年代针对这种缺陷兴起了结构化程序设计的变革。尤其是人们已经普遍地意识到，滥用goto语句会严重地降低程序的可读性。能够从头顺序读到结尾的程序，比需要读者从一条语句跳跃到另一条语句来跟踪程序的执行顺序要好读得多。例如下面用C编写的嵌套循环：

```
while (incr < 20) {
   while (sum <= 100) {
      sum += incr;
   }
   incr++;
}
```

如果C如同Fortran的早期版本一样，不具有这种while语句，这段代码将会写成下面这样的形式：

```
loop1:
  if (incr >= 20) go to out;
loop2:
 if (sum > 100) go to next;
  sum += incr;
  go to loop2;
next:
  incr++;
  go to loop1;
out:
```


20世纪70年代初期BASIC及Fortran的版本缺乏严格采制使用goto语句的控制语句，因而很难用它们写出可读性高的程序。然而自20世纪60年代后期所设计的大多数程序设计语言都包括了足够的控制语句，这极大地减少了使用goto语句的必要（如果还没有完全消除它的使用的话），所以现代语言中的控制语句的设计就不像过去那样对可读性有很大的影响了。

当然，使用任何语言都有可能编写出结构性能差的程序，而在一种语言中包括goto语句尤其容易诱导人们编写这类差的程序。因而，语言中适度的控制语句能够允许人们编写结构优良、可读性高的程序。但仅仅是这类控制语句的存在并不能保证就能编写出可读的程序来。

1.3.1.4 数据类型与数据结构

在程序设计语言中给出定义数据类型与数据结构的合理机制，是语言可读性的又一种重要辅助。例如，假设在某种语言中不存在布尔数据类型，数值数据类型就被用来作为标志。在这种语言中，我们可能会有如下的赋值语句：

```
timeOut = 1
```

这条语句的意义是不清楚的，而在具有布尔数据类型的语言中，我们可以有赋值语句

```
timeOut = true
```

这条语句的意义就完全清楚了。同样地，记录数据类型比使用一组相似数组来记录雇员档案，其可读性更好；在没有记录数据类型的语言中，就必须运用这些相似数组来记录这种信息，其中的每一个数组记录雇员档案中的一列数据。例如在Fortran 95中，可以将一组雇员档案的记录存储在下面的数组之中：

```
Character (Len = 30):: Name (100)
Integer :: Age (100), Employee_Number (100)
Real :: Salary (100)
```

13

在这四个数组之中，具有相同下标的数组元素记录着某一个雇员的资料。

1.3.1.5 语法设计

一种语言的组成元素的语法或形式对程序的可读性有着极为重要的影响。下面的三个例子说明语法设计的选择是如何影响可读性的：

- 标识符形式。将标识符的长度限制得很短会降低语言的可读性。如果像在Fortran 77中那样，规定标识符的长度最多不能超过六个字符，那么常常很难赋于变量以有意义的名字。一个更为极端的例子是美国国家标准协会（ANSI）关于BASIC语言的原始规定（ANSI, 1978b），它限制标识符只能包含单个字母，后面只能跟随一个数字。其他有关标识符形式的设计问题将在第5章里进行讨论。
- 特殊字。语言中特殊字的形式（例如，while、class和for）将极大地影响程序的外观，及与此相关联的程序的可读性。其中特别重要的是构造复合语句或语句组合的方式，尤其是在控制结构中构造复合语句的方式。一些语言使用配对的特殊字或者特殊符号来构造复合语句。C及其后继语言使用括号来说明复合语句。所有这些语言都有相类似的麻烦：它们的复合语句总是以同样的方式终止，当出现end或者}时，人们很难判断，究竟是哪一个语句组合应该被结束。Fortran 95和Ada语言在这个问题上则较为清晰，它们对不同类型的复合语句采用不同的终止语法，例如，Ada用end if来终止选择结构，而用end loop来终止循环结构。这仅是语言的简单性与可读性之间存在矛盾的一个例子：为实现语言的简单性，则定义较少的特殊保留词，类似在C++中；为实现较大程度的可读性，则使用较多的保留词，类似在Ada中的那样。

  另外一个重要的问题是，能否使用一种语言的特殊字作为程序中的变量名。如果容许，这样编写的程序很容易引起混淆。例如在Fortran 95中，特殊字如Do和End，就是合法的变量名。所以，当这些词在程序中出现时，它们可能（也可能）没有包含特殊的含义。
- 形式与意义。设计程序语言的语句时，使其字面形式至少部分地表明它们的功用。这对语言的可读性有着明显的助益。语义或者说意义应该紧跟语法或形式。但有时两种语言在结构形式上相同或者相似，而当将其放置于不同的位置时，就会具有不同的语义，这就违反了上述原则。例如在C语言中，保留字static的语义就取决于它所出现的位置，当它被用于一个函数中的变量的定义时，意味着该变量是在编译时产生的。当它被用于处于所有函数之外的变量的定义时，则意味着该变量只在它所定义的文件之中，而不能被输出到这个文件以外。

14

人们对UNIX（Raymond, 2004）系统的shell指令的不满之一，是它们的形式并非总是提示它们的功能。例如对于UNIX指令grep功能的猜认，仅仅只能通过事先对于它们的了解，或者取决于人们的聪明才智，以及对于UNIX编辑器ed的熟悉程度。这个指令的字面形式对UNIX的初学者来说毫无帮助。（在UNIX编辑器ed中，指令 /regular_expression/用来搜索与正则表达式相匹配的子字符串。在这个指令的前面放上g，它就变成了一个全局指令，它说明现在所搜索的范围是正在编辑的整个文件。如果在这个指令的后面再加上p，则说明打印该文件中所有包含匹配子字符串的行。所以显而易见地，可以将指令g/regular_expression/p缩写为grep。它的意义是：打印文件中所有包含与正则表达式相匹配的子字符串的行。）

1.3.2 可写性

程序设计语言的可写性是在给定的应用领域内为该语言产生程序的难易程度的一种度量。大多数影响可读性的语言特征也可以影响可写性。这是因为在编写程序的过程中，编程人员要不断地阅读已经编写了的程序部分。

如同可读性的情形一样，可写性只能在程序设计语言所针对的问题领域之内来考虑。如果在一个特定的应用领域内比较两种语言的可写性，一种语言是专为这种应用而设计的，而另一种语言则不是，这样的比较是不合理的。例如，COBOL与Fortran的可写性在产生处理二维数组的程序时是截然不同的，Fortran对于这类应用是理想的。在涉及产生复杂形式的财务报告时，这两种语言的可写性也十分不同，而COBOL正是为这类问题所设计的。

在下面的几个小节里，我们将阐述影响程序设计语言可写性的几个最重要的特性。

1.3.2.1 简单性与正交性

如果一种程序设计语言中具有大量的不同结构，那么一些编程人员可能只熟悉它们的一部分。这种情形会导致一些语言特性被误用，而另外的一些则被忽略。那些被忽略的特性也许比被采用的特性能够写出更加漂亮、更加高效或二者兼备的程序来。更有甚者，就如Hoare（1973）所指出，人们可能会意外地采用一些他们所不理解的特性，产生出莫名其妙的结果。所以，一组较少量的基本结构以及一套相互一致的组合规则（即正交性），比仅仅具有大量的基本结构要优越得多。这样，程序人员在学会了一套简单的基本结构之后，就能针对复杂问题设计出解决办法。

另一方面，过分的正交性也有损于可写性。当基本结构的任意结合几乎都是合法时，程序
15 中的错误就很难被检测出来。这也会导致编译器不能够发现代码中的错误。

1.3.2.2 支持抽象

简而言之，抽象指的是以合法省略许多细节的方式，来定义并且使用复杂结构或复杂运算的能力。在当代程序设计语言的设计之中，抽象是一个关键性的概念。这反映了抽象在现代程序设计方法学中所起的主角作用。一种程序设计语言所允许的抽象程度，以及这种抽象表现形式的自然程度，对语言的可写性是非常重要的。程序设计语言可以支持两种不同类型的抽象，即过程抽象与数据抽象。

过程抽象的一个简单例子，是采用子程序来实现在程序中反复运用的排序算法。如果没有子程序，这段排序代码将重复出现于程序中所有需要的地方，这使得程序冗长，而且编写的过程也麻烦得多。尤其重要的是，如果不采用子程序，程序里将会乱糟糟地充满了应该写在子程序中的排序算法的细节，这将使程序的流程和总态图很不清楚。

作为数据抽象的一个例子，考虑一种将整数数值存储于其节点的二叉树结构。在一种类似Forman 77的、不支持指针、并且不支持运用准的动态存储管理的语言之中，这种二叉树结构通常被实现为三个并行的整数数组，其中的两个整数被用来定义子节点的下标。而在C++ 和Java语言中，就能够使用树节点的抽象来实现这种树结构，而其中每个节点的形式为具有两个指针（或引用）以及一个整数的简单的类。后一种表示方法在表达形式上的自然性，使得运用这类语言来编写具有二叉树结构的程序比用Fortran 77编写要容易得多。这仅仅是因为这种程序设计语言解决问题的范畴更接近实际问题的领域。

对抽象是否全面支持显然是影响语言可写性的重要因素。

1.3.2.3 表达性

语言的表达性可以指语言中几种不同的特征，在一种类似APL(Gilman & Rose,1984)的语言之中，它指的是这些语言所具有的一些功能很强的运算符，这些运算符仅允许使用很小的程序完成极大量的运算。更一般地，表达性是指一种程序设计语言具有相对方便、非繁琐的方式来

说明运算。例如，C中的表达式count++比表达式count=count+1更为方便和短小精炼。还有，Ada中的布尔运算符and then是一种说明布尔表达式短路求值的方便方式。在Java语言中包括进for语句，使得在编写计数循环时比使用while语句更为容易，尽管使用后者也可以达到同样的目的。所有的这些都增进了程序设计语言的可写性。 16

1.3.3 可靠性

如果一个程序在任何条件下的运行都能够达到它的说明标准，我们称这个程序是可靠的。下面我们将阐述在特定语言中，对程序可靠性有极大影响的一些语言特性。

1.3.3.1 类型检测

类型检测就是用编译器或在程序的运行过程中测试给定程序中的类型错误。类型检测是语言可靠性中的一个重要因素。编译时的类型检测要比运行时的类型检测更为理想。这是因为运行时的类型检测是高代价的。程序中的错误发现得越早，改正错误的代价就越低。Java的设计要求在编译时对几乎所有的变量及所有的表达式都进行类型检测。这种方法实际上消除了Java程序运行时的类型错误。我们将在第5章和第6章深入讨论类型以及类型检测。

一个因为没有类型检测而导致无数程序错误的典型例子，是在早期的C语言（Kernighan and Ritchie，1978）中使用的子程序参数，它不在编译时，甚至也不在运行时实施类型检测。C不检测函数调用语句中的实参类型，以确定该实参的类型是否与函数定义中的形参类型相一致。一个在函数调用中被作为实参的int类型变量可以被传递给一个期待以float类型作为形参的函数；之后甚至无论是编译器或者运行系统都不能够发现这种不一致性。例如，表示整数23的位字位串与表示浮点数23的字位串是根本不相关的。如果将一个整数23传递给一个期待浮点数参数的函数，必然地，在该函数中任何对这个参数的使用都会产生毫无意义的结果。更有甚之，这类问题常常很难被发现[⊖]。目前的C语言版本通过要求对所有的参数都进行类型检测，从而消除了这类问题。有关子程序以及参数传递的技术将在第9章中讨论。

1.3.3.2 异常处理

一个程序如果具有中断运行时错误（以及这个程序所发现的其他非正常情况）并改正错误然后继续运行的这种能力，将显然有助于提高程序的可靠性。这种程序设计语言的机制就称为异常处理。Ada、C++和Java语言都具有用于异常处理的庞大机制，而这种机制在许多广泛应用的程序设计语言中（包括C和Fortran），实际上并不存在。异常处理的问题将在第14章中讨论。 17

1.3.3.3 使用别名

使用别名的非严格定义是，用两个或多个名字来访问同一个存储单元。现在人们已经普遍地意识到，使用别名是程序设计语言中的一个危险特性。大多数的程序设计语言允许一定程度的别名使用，例如在大多数语言中都有可能将两个指针指向同一个变量。程序人员必须时刻记住，在这样的程序中，如果这两个指针中的一个所指向的数值改变了，将引起另外一个指针所引用的数值的改变。在第5章和第9章将要讲到，有一些类型的别名使用能够被一种语言的设计所禁止。

一些语言靠使用别名来克服其数据抽象机制的低效率。而另一些语言则严格禁止使用别名以提高语言的可靠性。

⊖ 针对这个问题，以及其他一些类似问题，UNIX系统包含了一种名为lint的实用程序。这个程序将检查C程序中的这类问题。

1.3.3.4 可读性与可写性

可读性与可写性都会影响到可靠性。如果编写程序的语言不具有表达所需算法的自然方式，就必然会采用非自然方式。非自然的方法不能保证在所有可能的情况下都是正确的。一个程序越容易编写，则其正确的可能性就越大。

在软件生命周期中的编写与维护阶段，程序的可读性都会影响其可靠性。难于阅读的程序也难于编写和修改。

1.3.4 代价

程序设计语言的最终总代价是这种语言中各种特征的一个函数。

第一是训练程序员使用这种语言所具有的代价。这种代价是语言的简单性与正交性，以及程序人员以往经验的一个函数。尽管功能更强大的程序设计语言不一定就更难掌握，但通常的情形却往往是这样的。

第二是使用这种语言来编写程序所具有的代价。这种代价是程序设计语言可写性的一个函数，它部分地取决于这种语言的设计目的与特定应用问题的接近程度。设计和实现高级语言的初始目标就是降低软件开发的代价。

在优良的程序开发环境中，一种语言的程序人员培训代价以及程序编写代价都能够极大地降低。关于程序设计的环境问题，我们将在1.8节讨论。

第三是在这种语言中编译程序的代价。早期Ada语言应用的一个主要障碍是当时的第一代Ada编译器让人望而却步的高运行代价。随着更好的Ada编译器的出现，这个问题已经得以解决。

第四是程序运行的代价。一种语言的设计方式极大地影响着用这种语言所编写的程序的运行代价，无论该语言编译器的质量如何，要求进行大量运行时类型检测的语言必将阻碍代码的高速执行。尽管在早期的语言设计中，程序的执行效率是最优先考虑的因素，然而现在却认为不是那么重要了。

在被编译的代码上，其编译代价与运算速度之间存在着一种简单的权衡。一系列被称为优化的技术可以使得编译器减小它所产生的代码的规模，以及/或者提高这些代码的执行速度。如果在编译过程中做或者仅仅做很少的优化，编译过程就会比花费很大努力来产生优化编码的情形要快得多。而在这些编译过程所做的额外努力将导致代码执行的高效率。两种情形应该如何选择，取决于编译器将要运用的环境。对于学习初级程序设计的学生，则不需要或者只需要很少的优化，因为学生们使用大量的时间来编译程序，而只使用很少的执行时间（他们的程序很小，并且只需要一次正确的执行）。而在生产环境中，编译过的程序会需要多次执行，这必然值得付出额外的代价来优化代码了。

第五是一种语言实现系统的代价。人们之所以很快就接受Java语言的原因之一，是该语言免费的编译器/解释器系统，在Java语言设计第一次公布之后的短时期内，很快便能免费下载它的编译器/解释器系统。如果一种语言的实现系统价格昂贵，或者只限于运行在价格昂贵的硬件设备之上，这种语言被广泛应用的机会就要小得多。例如，第一代Ada语言编译器的高昂代价就曾经在早期阻止了Ada成为广泛应用的语言。

第六是可靠性差的代价。如果软件在一个关键系统中出错，如在一个核电站或在一台医用X光机上出错，这种代价可能会非常高。在非关键系统中出错，如果考虑到有可能失去今后的生意，或者因软件系统而造成官司，其代价也可能会非常昂贵。

最后要考虑的一点是程序维护的代价。程序维护包括程序的改错，以及为增加新功能而进行的修改。这种软件维护的代价取决于语言的许多特征，但主要是语言的可读性。因为维护工

作常常由非软件原始编写人员担任，可读性差的程序必将使维护工作极为艰难。

软件维护的重要性无论怎么强调都不会过其实。据估计，对于使用期相对长的大型软件系统，软件维护的代价可以是软件开发代价的2~4倍（Sommerville，2005）。

在影响程序设计语言代价的所有因素之中，三种因素最为重要：即程序开发、软件维护及可靠性。由于这三种代价又都是可写性与可读性的函数，由此可得出结论，可写性与可读性就成为评估程序设计语言的两种最重要的标准。

当然，其他的一些标准也可以用来评估程序设计语言。其中的一个例子就是可移植性，即一个程序能够从一种实现转移到另一种实现的难易程度。可移植性又极受语言的标准化程度的影响。某些程序设计语言，例如BASIC，就根本没有实现标准化。用这些语言编写的程序就很难从一种实现转移到另一种实现。程序设计语言的标准化是一个耗时又艰难的过程。某个委员会从1989年开始起草C++语言的一种标准版本，而这个版本直到1998年才最终被批准。

语言的通用性（在广泛范围内可应用的程度）及定义良好性（程序设计语言官方定义文档的完整性及准确程度）是另外两种评估语言的标准。

绝大多数的评估标准，尤其是可读性、可写性及可靠性，就不是精确定义的，也不可以精确测量。然而它们是很有用的概念，它们提供了有关程序设计语言的设计及其评估方面的极有价值的观念。

程序设计语言评估标准的最后一个要点：语言设计的标准随着不同人员的不同角度而有着不同的侧重。语言的实现人员主要关心的是实现这种语言的构造及其特性所可能遭遇的困难。语言的使用人员则首先担心的是可写性，其次是可读性。语言的设计人员则喜欢强调语言的优美，以及吸引人们广泛应用语言的能力。但所有这些特征有时又是相互矛盾的。

1.4 影响语言设计的因素

除了在1.3节中阐述的那些因素以外，还有一些因素会影响到程序设计语言的基本设计。其中最为重要的是计算机体系结构及程序设计方法学。

1.4.1 计算机体系结构

计算机的基本体系结构对语言的设计有着深远的影响。在过去50年中，绝大多数广为应用的语言都是依据一种普遍流行的计算机结构来设计的。这种结构被称为冯·诺依曼（von Neumann）体系结构，这个名字取自它的初创者之一，John von Neumann（发音为"von Noyman"）。这种类型的语言被称为命令式语言。在冯·诺依曼计算机中，数据与程序都被存储于同一个存储器。执行指令的中央处理器（CPU）与存储器是分开的。因而，指令与数据必须从存储器传输到CPU。而CPU运行的结果又必须传回存储器。几乎所有自20世纪40年代以来生产的数字计算机，都是基于冯·诺依曼体系结构的。图1-1显示了冯·诺依曼计算机的整体结构。

由于冯·诺依曼体系结构，命令式语言的核心特性有：模拟存储单元的变量、基于传输操作的赋值语句，以及迭代形式的循环运算——这是一种在冯·诺依曼体系结构上实现重复操作的最高效方式。表达式中的操作数从存储器传向CPU，而表达式的运算结果又被传回到由赋值语句左端所代表的存储单元上。由于运算指令顺序地存储于相邻的存储单元，并且一小段代码的重复执行只需要一个简单的分支指令，所以在冯·诺依曼计算机上迭代式循环是高速的。这种高效率不鼓励使用递归式循环，尽管递归式循环常常更为自然。

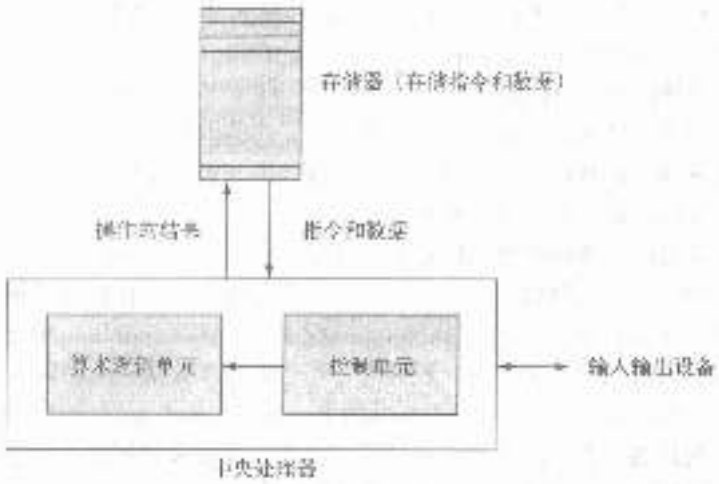


图1-1 冯·诺依曼计算机体系结构

在冯·诺依曼体系结构计算机上，机器码程序的执行发生于被称为取指-执行周期的过程中。如在1.4.1节所述，程序存储于存储器内，但在CPU上执行。必须将每一条要执行的指令从存储器转移到中央处理器。下一条要执行的指令的地址则保存在一个被称为程序计数器的寄存器之中。取指-执行周期可以简单地由下面的算法来描述：

```
初始化程序计数器
永远地重复
        取出程序计数器指向的指令
        增加程序计数器以指向下一条指令
        解码指令
        执行指令
结束重复
```

算法中的“解码指令”步骤意味着检查指令，以确定这条指令所说明的动作。当遇到一条停止指令时，程序的执行即被终止。尽管在实际的计算机上极少执行停止指令。控制则正好相反，是从操作系统送至用户程序使之执行，而当用户程序的执行完毕之后，再传回操作系统。如果在一个计算机系统中，同时有多个用户程序存在于存储器中，这样的过程就要复杂得多。

21 如前所述，函数式（或称作用式）语言进行计算的主要方式是将函数作用于给定参数之上。在函数式语言中的程序设计可以没有命令式语言所必需的那种变量，可以没有赋值语句，也可以没有循环。尽管许多计算机科学工作者为函数式语言的极大优越性做出了详细的说明，如Scheme语言，然而在一种能够允许函数式语言程序高效率运行的、非冯·诺依曼体系结构的计算机出现之前，函数式语言是不可能替代命令式语言的。在为这种现实惋惜不止的人中，最具雄辩力的莫过于Fortran语言早期版本的主要设计者John Backus（Backus，1978）。

在过去25年间产生的并行体系结构计算机，具有加快函数式程序的执行速度的希望。然而迄今为止，这种函数式程序的执行速度仍然不足以与命令式程序的执行速度相比。事实上，尽管存在着使用并行体系结构来执行函数式程序的好的方式，但大多数并行机仍被用于命令式程序，特别是用于那些使用Fortran的方言所编写的程序。

尽管事实上，命令式程序设计语言模拟的是机器的系统结构，而并不是基于语言的使用人员的能力及倾向，但仍然有一些人认为，使用命令式程序设计语言在某种程度上比使用函数式

程序设计语言更为自然。

1.4.2 程序设计方法学

从20世纪60年代末期至70年代初期，由于结构化程序设计运动的兴起，人们开始深入分析软件开发的过程及程序设计语言的设计问题。

当时开展这种研究的一个重要原因是，随着硬件成本的降低及软件费用的增加，计算机的主要代价从硬件转向了软件。当时程序开发人员生产效率的提高相对很少。加之，计算机被用来解决规模越来越大、越来越复杂的问题。这些计算问题不再如20世纪60年代初期那样，仅仅是解方程组以模拟卫星的轨迹；人们已经开始在为巨大和复杂的任务编写程序，如控制庞大的炼油设备，提供全球性的航空订票系统。

作为20世纪70年代的研究成果产生的新的软件开发方法学被称为自顶向下设计、逐步求精。当时，发现程序设计语言的主要缺点是不完全的类型检测，以及不适宜的控制语句（需要运用 22
太多的goto语句）。

在20世纪70年代的后期，开始了从面向过程转移到面向数据的程序设计方法学。面向数据的方法着重于数据的设计。这种方法的注意力集中于运用抽象的数据类型来解决问题。

要将数据抽象有效地应用于软件系统设计，其实现语言必须能够支持这种数据抽象。SIMULA 67 （Birtwistle et al., 1973）是第一种对数据抽象提供有限支持的程序设计语言，尽管这个语言的本身事实上并没有因此而得到普及。直到20世纪70年代初期，人们才普遍地认识到运用数据抽象的优越性。当面自20世纪70年代末期以来所设计的大部分程序语言都支持数据抽象。数据抽象的问题将在第11章中详细讨论。

在面向数据的软件开发的演进中，最后的一步是始于20世纪80年代初期的面向对象的程序设计，面向对象的程序设计方法学开始于数据抽象。这种方法将数据处理与数据对象封装在一起，并控制对数据的访问，并添加了继承与动态方法绑定。继承是一个功能强大的概念，它大大地提高了重用现有软件的潜能，从而提供了大幅提高软件开发效率的可能性。这是使得面向对象的程序设计语言广为普及的一个重要因素。动态（运行时）方法绑定使得对于继承的运用变得更为灵活。

面向对象的程序设计是伴随着一种支持这种概念的程序设计语言Smalltalk（Goldberg and Robson，1989）而发展起来的。尽管Smalltalk语言从来没有像其他程序设计语言那样被广泛普及地应用，但它所采取的支持面向对象程序设计的方法，如今已成为最为广泛流行的命令式语言的组成部分，这里包括Ada 95（ARM，1995）、Java和C++。面向对象的原理也进入了CLOS（Bobrow et al.，1988）语言中的函数式程序设计，并且还进入了Prolog++（Moss，1994）语言中的逻辑式程序设计。关于支持面向对象程序设计的语言，我们将在第12章中详细讨论。

在某种意义上，面向过程的程序设计是面向数据程序设计的反面。尽管当今面向数据的方法在软件开发中占据统领地位，然而面向过程的方法也没有被抛弃。相反，人们近年来进行了大量关于面向过程的程序设计方法的研究，尤其是在并发性方面的研究，这些研究工作引发了 23
基于产生与控制并发程序单元的语言工具的需求。Ada，Java和C#都包括了这项功能。并发性问题将在第13章里详细讨论。

1.5 语言分类

程序设计语言通常可以被分为四类：命令式语言、函数式语言、逻辑语言，以及面向对象的语言。我们已经讨论了命令式语言和函数式语言的特点，我们也简述了最流行的面向对象语

言是如何从命令式语言中生成。尽管面向对象的软件开发流程与通常用于命令式语言中的面向过程的流程有很大不同，但使一种命令式语言也能够支持面向对象的程序设计所需做的扩展工作并不巨大。例如，C语言与Java语言中的表达式、赋值语句以及控制语句几乎都是相同的。（但另一方面，Java中的数据、子程序和语义与C却大不相同。）类似的语句能用于支持面向对象程序设计的函数式语言。

另一类语言，可视化语言，形成了命令式语言的一个子类。可视化语言中最为流行的是Visual BASIC（Schneider, 1999）。Visual BASIC现在已经由Visual BASIC.NET（Deitel, et al., 2002）所替代。这类语言（或者是这类语言的实现）包括了以"拖放"方式产生代码段的功能。这类语言曾经被称为第四代程序设计语言，当然，这个名称现在又已经被弃用了。可视化语言的一种代表性特性是提供了生成程序的图形用户界面的简单方式。例如在Visual BASIC中，单击某个键便能产生显示表单控件的代码，例如一个按钮或一个文本框。现在，所有的.NET语言都实现了这项功能。

逻辑程序设计语言是基于规则的语言的范例。在命令式语言中，必须对一个算法能以详尽说明，并且其中还必须包括执行这些指令或语句的顺序。而在一种基于规则的语言中，规则的说明并不需要一定的顺序，但语言的实现系统则必须选择一种执行顺序以获得预期的结果。这种软件开发的方式与其他三种类型的语言所使用的方式是根本不同的，因而它所需要的是一种完全不同类型的语言。我们将在第16章中讨论最常用的逻辑程序设计语言Prolog，以及关于逻辑程序设计的问题。

近年出现了一种新类型的语言，即标记与程序设计混合语言。标记语言，包括最广为应用的XHTML标记语言，并非程序设计语言。这种语言仅被用来说明信息在Web文档中的布局。然而，一些程序设计的功能已经融汇进XHTML及XML语言的某些扩展形式中。我们所指的这些扩展形式包括Java Server Pages Standard Tag Library（JSTL）和eXtensible Stylesheet Language Transformations（XSLT）。这两种语言将在第2章中简略地介绍。

在过去的40多年里，出现了一系列特殊用途语言。这些语言从生成商务报告的Report Program Generator（RPG）到用作指示可编程机器的工具的Automatically Programmed Tools（APT），再到基于系统模拟的General Purpose Simulation System（GPSS）。主要是为这些语言应用领域狭窄，很难将它们与其他语言进行比较，因此本书将不会讨论特殊用途语言。

1.6 语言设计中的权衡

在1.3节中所阐述的关于程序设计语言的评估标准给语言的设计提供了一个框架，但遗憾的是，这个框架又是自相矛盾的。Hoare（1973）在他关于语言设计的论文中深刻的指出："存在着这么多可靠、但又矛盾的标准，以至于发它们之间的相互协调与满足成为了一项主要的工程任务。"

两个相互矛盾的标准是可靠性与执行代价。例如，Java语言的定义要求：必须对所有数组元素的引用进行检测，以保证所有下标都在合法的范围之内。这个步骤给包含大量数组元素引用的Java程序增加了很大的执行代价。C语言不要求进行下标范围的检测，所以C程序的执行速度比语义上相同的Java程序要快得多。当然Java程序则更为可靠。Java语言的设计人员以程序执行效率为代价换取了可靠性。

直接导致设计权衡的、相互矛盾的标准的另一个例子是APL语言。APL语言具有一些强功能的强大、用于数组操作数的运算符。因为这些运算符的数目很大，APL必须引入大量的新符号来表达这些运算符。另外，多个APL运算符可以同时用于同一冗长、复杂的表达式中。

作为这种表达性高的一种结果是，对于那些具有多个数组运算的应用，APL语言的可写性很高。的确，只需要一个十分短小紧凑的程序就能进行大量的计算。然而它的另一个结果则是，APL程序的可读性极差。紧凑精练的表达式具有一定程度上的数学形式美，然而却让非程序编写者的其他人员难于理解。著名的作者Daniel McCracken曾经写到，他花费了五个小时来阅读和理解一个仅有四行的APL程序（McCracken，1970）。APL语言的设计者以可读性为代价换取了可写性。

可写性与可靠性之间的矛盾是语言设计中的一对普遍矛盾。C++中的指针可以以各种不同的方式来操作，这导致了C++中数据寻址的高度灵活性。因为指针所具有的潜在可靠性问题，Java没有包括这样的方式。

矛盾存在于语言设计（以及语言评估）的标准之中，这样的例子举不胜举。有一些还干脆妙，另一些则较为明显。然而很明显，设计一种程序设计语言时，在对语言的结构及特性进行选择的工作中，包含着一系列的妥协与权衡。

1.7 实现方法

如1.4.1节所述，计算机的两个主要组成部分是它内部的存储器及处理器。内部存储器被用来存储程序和数据，处理器则是一组电路，用来实现一系列的基本运算或机器指令，如进行算术运算和逻辑运算的指令。在大多数计算机中，有一些指令通常被称为宏指令，实际上这些指令是通过定义于更低层次的指令（称为微指令）来实现的。因为微指令从不在软件中显示，因此我们在这里不对它们作进一步的讨论。

计算机的机器语言是一套指令。在没有其他支持软件的情况下，机器语言是大多数硬件计算机能够“理解”的唯一语言。理论上，也可以这样来设计和建造一台计算机，即可以使用一种特殊的高级语言作为它的机器语言。但这样建造的计算机会十分复杂且非常昂贵。此外也会极不灵活，因为很难通过其他的高级语言来使用它（尽管并非不可能）。计算机设计中较现实的选择是，在其硬件上实现能够提供最普遍需要的基本操作的较低层次的语言，而要求其系统软件生成使用其他高层次语言编写程序的接口。

一种语言的实现系统并不是一台计算机上的唯一软件。它还需要一个称为操作系统的大程序集，这个程序集提供高于机器语言层次的基本操作。这些基本操作包括系统资源的管理、输入和输出操作、文件管理系统、文字以及/或者程序编辑器，还包括其他各种普遍需要的功能。因为语言的实现系统需要许多操作系统工具，所以实现系统是与操作系统接口，而不是（用机器语言）直接与处理器打交道。

操作系统和语言实现系统被分层放置于计算机的机器语言接口之上。可以将这些层次设想为虚拟计算机，这个计算机在高层次上给用户提供使用接口。例如，一个操作系统和一个C程序编译器就是一个虚拟的C计算机。借助于其他编译器，一个机器能变成其他类型的虚拟计算机。绝大多数的计算机系统都提供几种不同类型的虚拟计算机。用户程序在这个虚拟计算机的顶端层次上形成另一个层次。图1-2显示了这种计算机的分层概念。

在20世纪50年代后期创造的第一种高级程序设计语言的实现系统，当时属于最复杂的软件系统。在20世纪60年代，人们进行了大量深入的研究工作，以理解构造高级语言实现系统的过程，并将这一过程形式化。当时这些工作中最成功的部分是在语法分析领域，这主要是因为这一部分的实现过程是已经理解了的自动机理论和形式语言理论的部分应用。

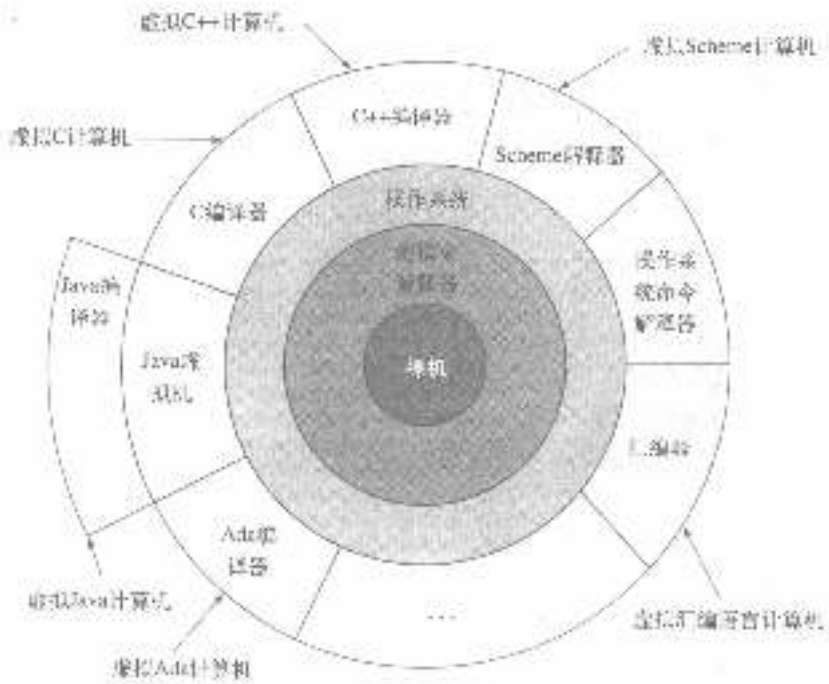


图1-2 一台典型的计算机系统所提供的虚拟计算机的分层接口

1.7.1 编译

实现程序设计语言的方法可以是三种一般方法中的任何一种。一个极端方面是，可以将程序翻译成能够在计算机上直接运行的机器语言，这种方法被称为编译器实现。这种方法的优越性是，一旦完成翻译过程，程序执行速度非常快。大多数程序设计语言（如C，COBOL和Ada）的实际实现都是借助于编译器的。

27

被编译器翻译的语言称为源语言。编译的过程以及程序的执行跨越了几个阶段，图1-3显示了其中最重要的几个阶段。

词法分析器将源程序中的字符集合起来组成词法单元。程序中的词法单元有识别符，特殊字、运算符和标点符号。词法分析器将忽略源程序中的注释部分，因为这些部分对于编译器没有使用价值。

语法分析器从词法分析器中取出词法单元，并使用这些词法单元构造一种被称为语法分析树（parse tree）的层次结构。这种语法分析树代表了程序中的语法结构。在许多情况下，并没有真正构成语法分析树的实际结构，而只是直接产生和利用了这些建立语法分析树所必需的信息。词法单元和语法分析树将在第3章中作进一步的讨论，而词法分析和语法分析则将在第4章中进行讨论。

中间代码生成器产生一个在不同语言中的程序，这种程序介于源程序和编译器最后输出的机器语言程序之间[⊖]。中间语言看起来往往很像汇编语言，而事实上有时真的就是汇编语言。在其他一些情况下，中间代码处于略微高于汇编语言的层次上。语义分析器是中间代码生成器的一个组成部分。语义分析器将检测在语法分析过程中难以发现的错误，如类型错误。

⊖ 注意，程序与代码两个词经常被替换地使用。



图1-3 编译过程

优化通过使程序更加精炼或快速或两者兼有之，达到改进程序（通常是在程序的中间代码版本之上）的目的。优化常常是编译的一个可选部分。事实上，一些编译器不能实行任何重要的优化。这类编译器适用于程序的编译速度远比翻译后程序的执行速度更为重要的情形。这种情形的一个例子是初学程序人员的计算实验室。大多数商务和工业生产的情形中，程序的执行速度远比其编译速度更为重要。因而常常需要优化。又由于许多类型的优化难于在机器语言上实现，因而绝大多数的优化是在中间代码上完成的。

代码生成器将程序优化后的中间代码版本翻译为相应的机器语言程序。

符号表被用作编译过程的数据库。符号表的主要内容是程序中用户定义的名字的类型和属性信息。这些信息由词法分析器和语法分析器放置于符号表中，以供语义分析器和代码生成器使用。

如前所述，尽管由编译器产生的机器语言能够直接在硬件上运行，但同时，这些机器语言几乎必须与一些其他代码一起运行。大多数的用户程序还需要来自操作系统的程序，其中最常用的是用于输入和输出的程序。用户程序需要这些程序时，编译器就产生对于所需要的系统程序的调用。在编译器所产生的机器语言程序被执行之前，必须找到所需要的操作系统程序，并将它们与用户程序连接起来。这是通过将系统程序的入口地址放置到用户程序中对其调用的位置，而将用户程序与系统程序连接起来。这样连接起来的用户程序和系统程序，有时被统称为**装载模块**或**可执行镜像**。这种收集系统程序并将它们与用户程序相连接的过程被称为**链接与装载**，或有时仅称为**链接**。这样的过程是由被称为**链接器**的系统程序来完成的。

除了系统程序以外，用户程序还必须时常与已编译过的、放置于程序库中的其他用户程序链接。这样链接器的任务就不仅仅是将一个给定的程序与系统程序相链接，它还要将这个程序

与其他的用户程序相链接。

一台计算机的存储器与它的处理器之间的链接速度通常决定着这台计算机的速度，因为执行指令的速度往往比将指令传送到处理器的速度更快。这一问题被称为冯·诺依曼瓶颈；它是冯·诺依曼体系结构计算机速度的主要限制因素。解决冯·诺依曼瓶颈是并行计算机研究和发展的一个主要动机。

1.7.2 单纯解释

单纯解释在实现方法中正好是与编译相反的另一个极端。使用单纯解释的方法，程序不需要经过任何翻译过程，而是由另一个被称作解释器的程序来解释执行。解释器的作用就如同一个机器的软件模拟。它的取指-执行周期的对象是高级语言中的程序语句而非机器指令。这种软件模拟显然给程序设计语言提供了一个虚拟机器。

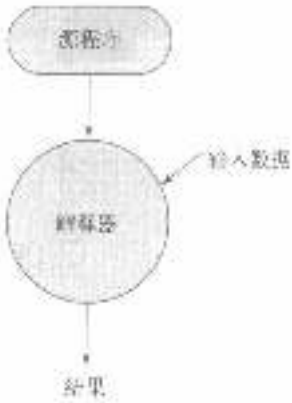


图1-4 单纯解释

单纯解释技术的优越性是，它能够允许容易地实现在许多源程序层次上的调试操作。这是因为运行时的所有出错消息能够指向出错的源程序中的单元。例如当发现一个数组的下标越界时，出错消息能够很容易地指出源程序里错误所在的行以及数组的名称。但另一方面，这个方法存在着严重的缺陷，程序执行的速度要比编译过的系统速度慢10～100倍。其速度缓慢的主要原因是对高级语言中语句的解码过程。这种解码过程远比机器语言指令的解码过程复杂得多（尽管使用高级语言的语句可能会比实能相同的机器码的指令数目少）。更有甚者，一条语句无论被执行多少次，对它的每一次执行都必须解码，因而语句的解码（而非处理器与存储器之间的连接）是单纯解释方法的瓶颈。

29～30

单纯解释的另一个缺点是它常常需要较多的存储空间。除了源程序之外，符号表在解释过程中也必须出现。此外，源程序的存储形式是为了方便存取和修改而设计的，而不是为了占用最小的存储空间。

尽管20世纪60年代的一些简单的早期语言（如APL、SNOBOL以及LISP语言）是单纯解释性的语言，但到了20世纪80年代，这种方式已经很少在高级语言上使用。然而近年来，单纯解释在万维网的一些脚本语言上又大量使用，例如在如今广为应用的JavaScript和PHP上。图1-4所示为单纯解释的过程。

1.7.3 混合实现系统

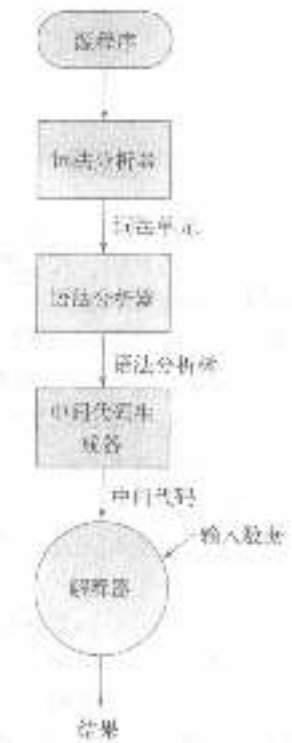


图1-5 混合实现系统

一些程序设计语言的实现系统介于编译器与单纯解释器之间。这类系统将高级语言的程序翻译成一种专为方便解释而设计的中间语言。因为只需要对源程序语言中的语句解码一次，所以这种方法比单纯解释要快。这样的语言实现系统就被称为混合实现系统。

图1-5所示是一种用于混合实现系统中的过程。这个系统仅仅是解释中间语言代码，而不是将中间代码翻译成机器码。

Perl语言通过一个混合实现系统实现。Perl的程序被部分地编译，以便于在解释执行之前发

或错误，并且使得解释器简单化。

Java语言的最初实现都是采用混合方法。其中被称为字节码的中间形式给任何装有字节码解释器以及与之相关的运行时系统的机器提供了可移植性。所有这样的机器都被统称为 Java虚拟机。而现在有些系统将Java的字节码翻译成机器码，以便达到较快的执行速度。 31

即时（Just-in-Time，JIT）实现系统最初将程序翻译成为一种中间语言。然后在执行的过程中，当中间语言的方法被调用时，这种实现系统再将中间语言的方法编译成为机器代码。现在JIT实现系统被广泛用于Java程序。所有的.NET语言也都是用JIT系统实现的。

有些时候，实现程序可以给同一种语言同时提供编译和解释两种实现方法。在这些情况下，解释器被用来开发和调试程序。当达到无错状态（相对而言）之后，再编译程序，以使于提高执行的速度。

1.7.4 预处理器 32

预处理器是在一个程序被编译之前对这个程序进行处理的程序机制。预处理器的指令被嵌入程序中。归根到底，预处理器是一个宏指令扩展器。预处理器的指令通常被用来说明程序所应该包括的、来自于其他文件的代码。例如下面这条 C 语言预处理器指令用于将myLib.c中的内容复制到程序中#include所在的位置：

```
#include mylib.c
```

另外一些预处理器指令被用来定义表达式中的符号。例如我们可以使用下面的指令来确定两个给定表达式之中数值最大的一个：

```
#define max(A, B) ((A) > (B) ? (A) : (B))
```

例如表达式

```
x = max(2 * y, z / 1.73);
```

通过预处理器就可以被扩展为：

```
x = ((2 * y) > (z / 1.73) ? (2 * y) : (z / 1.73));
```

请注意，这就是一个表达式的副作用会引起麻烦的例子。例如，如果这两条属于宏指令max的表达式之一具有如z++之类的副作用，就会引起问题。因为这两条表达式中的一个参数被计算了两次，经由宏指令扩展所产生的代码导致z增值两次。

1.8 程序设计环境

程序设计环境是指一系列在软件开发过程中所使用的工具。这套工具可能只包括了一个文件系统、一个文本编辑器、一个链接器以及一个编译器，或者，它也可能包括经由一个统一的用户界面来调用的一系列集成工具。后面的这种情形极大地提高了软件开发和维护的效率。因而一种程序设计语言的特征不是一个系统的软件开发能力的重要衡量标准。我们现在简略地描述几种程序设计的环境。

UNIX系统是一种比较老的程序设计环境，它首次发布于20世纪70年代中期，是围绕一种可移植的多道程序设计操作系统建造的。UNIX 系统给各种语言的软件开发与软件维护提供了广泛的强有力的支持工具。过去 UNIX 系统所缺乏的最重要特性是这些支持工具的统一界面。正是这种欠缺，使得人们较难掌握和运用UNIX 系统。然而现在的 UNIX 系统常常是通过运行于 UNIX 系统顶层的一个图形用户界面(GUI)来使用。UNIX 图形用户界面包括Solaris通用桌面环境（Solaris Common Desktop Environment，CDE），GNOME以及 KDE。这些图形用户界面使得 33

UNIX系统的界面形式就与Windows以及Macintosh系统的用户界面相类似。

Borland JBuilder是一种程序设计环境，它是供用于Java开发的一个集编译器、编辑器，调试器和文件系统于一体的组合，而这四个组成部分的调用都是经过一个图形界面。JBuilder是一个创建Java软件的复杂而又功能供全的系统。

软件开发环境发展中的最新进展以微软的Visual Studio.NET为代表。这是一组大量而精致的软件开发工具，全部通过视窗系统界面来使用。可以运用这个系统在.NET系列的五种语言中挑选任意一种来开发软件。这五种语言是：C#、Visual BASIC .NET、JScript（微软公司的JavaScript版本）、J#（微软公司的Java版本）、或者受控的C++。

小结

学习程序设计语言具有几个重要的理由：增强编写程序时运用不同语言结构的能力，使我们能够更为明智地为软件项目选择程序设计语言，以及更轻松地学习新的程序设计语言。

计算机已被广泛用于解决各个领域中的各种问题。对于一种特定程序设计语言的设计和使用，很大地取决于它所应用的领域。

评估程序设计语言最重要的标准是可读性、可写性、可靠性和总体代价，这些将成为我们审视和判断本书讨论的各种语言特性的基础。

影响语言设计的主要因素是计算机体系结构和软件设计方法学。

设计一种程序设计语言主要是一种工程技巧，之中包括语言的各种特性、结构与功能上的一系列权衡。

实现程序设计语言的主要方法有编译、单纯解释及混合实现。

程序设计的环境已经成为软件开发系统的重要组成部分，而其中程序设计语言仅仅只是一个部分。

复习题

1. 为什么具有一些语言设计知识背景对编程人员会有帮助，即使他或她可能实际上从不设计程序设计语言？
2. 为什么程序设计语言特征的知识会使整个计算机界受益？
3. 在过去45年中，哪一种程序设计语言是科学计算领域里最主要的应用语言？
4. 在过去45年中，哪一种程序设计语言是商务应用领域里最主要的应用语言？
5. 在过去45年中，哪一种程序设计语言是人工智能领域里最主要的应用语言？
6. UNIX系统是用哪一种语言编写的？
7. 一种程序设计语言具有太多的特性有什么缺点？
8. 为什么用户定义的操作符重载会使书写程序的可读性？
9. 能否举出一个在C设计中缺乏正交性的例子？
10. 哪一种程序设计语言将正交性作为一个主要的设计准则？
11. 哪一种从不控制语句被用来在缺乏控制语句的语言中实施比较复杂的控制语句？
12. 程序设计语言中的哪一种构造提供了过程抽象？
13. 一条程序是可靠的，意味着什么？
14. 为什么子程序参数的类型检测很重要？
15. 什么是别名使用？
16. 什么是异常处理？
17. 为什么可读性对于可写性很重要？
18. 特定语言编译器的代价与这种语言的设计有什么关系？
19. 在过去的45年中，影响程序设计语言设计的最重要因素是什么？
20. 其语言结构取决于运行着计算机体系结构的是什么类型的程序设计语言？给出类型名。
21. 20世纪70年代软件开发研究的结果，发现了哪两种程序设计语言的缺陷？

22. 面向对象的程序设计语言有哪三个基本特性？
23. 支持面向对象程序设计语言三个基本特性的第一种语言是哪种语言？ 35
24. 给出一个例子，说明两个语言设计标准是相互矛盾的。
25. 实现一种程序设计语言有哪三种一般方法？
26. 哪一种方法能产生较快的程序执行，是编译器、还是单纯解释器？
27. 符号表在编译器中起什么作用？
28. 链接器做什么工作？
29. 当一语句结束的重要性是什么？
30. 用单纯解释器来实现一种语言的优点是什么？

练习题

1. 你相信人们的想象思维的能力受人们的语言能力的影响吗？给出理由来支持你的观点。
2. 就你所知道的程序设计语言中，哪些语言特性的合理性使你无法理解？
3. 如果要支持所有的程序设计领域，仅使用一种语言，你的论据是什么？
4. 如果要反对所有的程序设计领域，仅使用一种语言，你的论据是什么？
5. 除了本章中讨论过的准则之外，命名并解释另外一种能够用来判断程序设计语言的准则。
6. 依你之见，常用程序设计语言中的哪种语句对可读性是最有危害的？
7. Java使用右花号来标志所有复合语句的结束。你支持或者反对这种设计的论点是什么？
8. 许多语言(用户定义的名字中区别大小写字母。你支持或者反对这种设计决策的论点是什么？
9. 解释程序设计语言代价的不同方面。
10. 为什么即使硬件已经相对的便宜，也要编写高效率的程序？给出你的论点。
11. 就你所知道的一些程序设计语言，描述效率与安全之间的设计权衡。
12. 在你的观点中，一种完美的程序设计语言应该包括哪些主要的特性？ 36
13. 你所学习的第一种高级语言是由一个单纯解释器、一个混合实现系统，还是由一个编译器来实现的？(如果没有研究过，你必然不会知道这些。)
14. 就一些你曾经使用过的程序设计环境，描述它们优点与缺点。
15. 说明简单变量的类型声明语句如何影响程序设计语言的可读性？设想一些语言不需要它们。
16. 运用本章所描述的可写标准给你所知道的一些程序设计语言写评估。
17. 一些程序设计语言，例如Pascal，用分号来分隔语句，而Java则用分号来结束语句。依你所见，哪一种用法最自然，并且不会造成语法错误？给出理由。
18. 有些现代语言允许两种类型的注释：一种在注释行的两端使用花括号(多行注释)，另外一种只在注释行的开头使用同等标志(单行注释)。参照我们的标准，分别讨论这两种设计选择的优点和缺点。 37

第2章 主要程序设计语言的发展

本章将叙述一系列程序设计语言的发展过程，还要探究每一种语言的设计环境，并且重点介绍各种语言的贡献与开发动机。本章并不包括对这些程序设计语言的总体描述，相反，我们只讨论每种语言引入的一些新特性。我们尤其感兴趣的是那些对后来的程序设计语言或对计算机科学领域有极大影响的特性。

本章将不深入讨论任何语言特性或者概念；我们将这样的讨论留到后面的章节中。对语言特性非形式化的简略解释就足以帮助我们理解这些语言的发展过程。

本章将讨论许多读者都不熟悉的不同的语言和语言原理。而且，在后续的章节中将详细讲解相关主题，读者在本章遇到的悬而未决的问题只有在本书的后面章节中才能解决。

在这里选择那些语言进行讨论是基于作者的观点，许多读者可能会不高兴他们所喜欢的一种或多种语言未被包括其中。然而，为了保持合理的篇幅，我们必须舍弃一些被人们推崇的语言，这种选择是基于我们对每种语言在程序设计语言的发展及其对整个计算机世界的重要性的估价。我们还将简略地讨论一些在本书的后面几章将要引用的其他语言。

本章按如下方式组织：按照年代的顺序一般性地讨论语言的最初版本。然而，语言的后续版本也和最初版本一起突出，后不放在后面章节中。例如，Fortran 2003在Fortran I (1956) 一节中讨论。当然，在某些情况下，与一些语言联系不那么紧密的语言放在它们自己的小节中介绍。

本章包括14个完整的示例程序，每个程序使用一种不同的程序设计语言。但本章并不对这些程序进行描述，仅仅是举例说明程序在这些语言中的外观。除了LISP、COBOL和Smalltalk语言的程序以外（关于LISP程序的例子，将在第15章讨论），任何熟悉常用命令式语言的读者都应该能够阅读和理解本章程序中的大部分代码。这一章中的Fortran、ALGOL 60、PL/I、BASIC、Pascal、C、Perl、Ada、Java、JavaScript以及C#程序，都用于解决同一个问题，请注意，上面绝大多数现代语言都支持动态数组，但因为示例问题的简单性，我们没有在这些示例程序中运用动态数组。另外，在Fortran 95的程序中，我们没有使用那些可以用以避免循环的特性，部分原因是为了保持程序的简单、可读性，而另外的原因仅仅是为了介绍这种语言中的基本循环结构。

图2-1是本章所讨论的高级语言的一个概貌。

2.1 Zuse的Plankalkül语言

这一章讨论的第一种程序设计语言在几个方面都极不寻常。首先，它从来没有被实现过；其次，尽管它于1945年开发，但关于它的描述直到1972年才发表。由于极少有人熟悉这种语言，所以直到这种语言开发完成15年之后，它的一些功能才出现于其他程序设计语言之中。

2.1.1 历史背景

1936年至1945年间，德国科学家Konrad Zuse（发音为"Tsoo-zuh"）用电子机械继电器制造了一系列复杂的计算机。到了1945年初，战争几乎毁坏了他所有的最新计算机模型，只有Z4模型幸存下来，他因此搬到了一个称为Hinterstein的遥远的巴伐利亚村庄，而他的研究小组成

员也都各有前景。

Zuse独自工作着，他若干开发一种用来表达计算的语言。这是开始于1943年，作为他的博士论文计划的一个项目。他将这种语言命名为 Plankalkül，意为“程序设计学”。在一份很长的、落款日期为1945年但直到1972年才发表的手稿中（Zuse，1972），Zuse 定义了Plankalkül语言，并且使用这种语言为各种问题编写了算法。

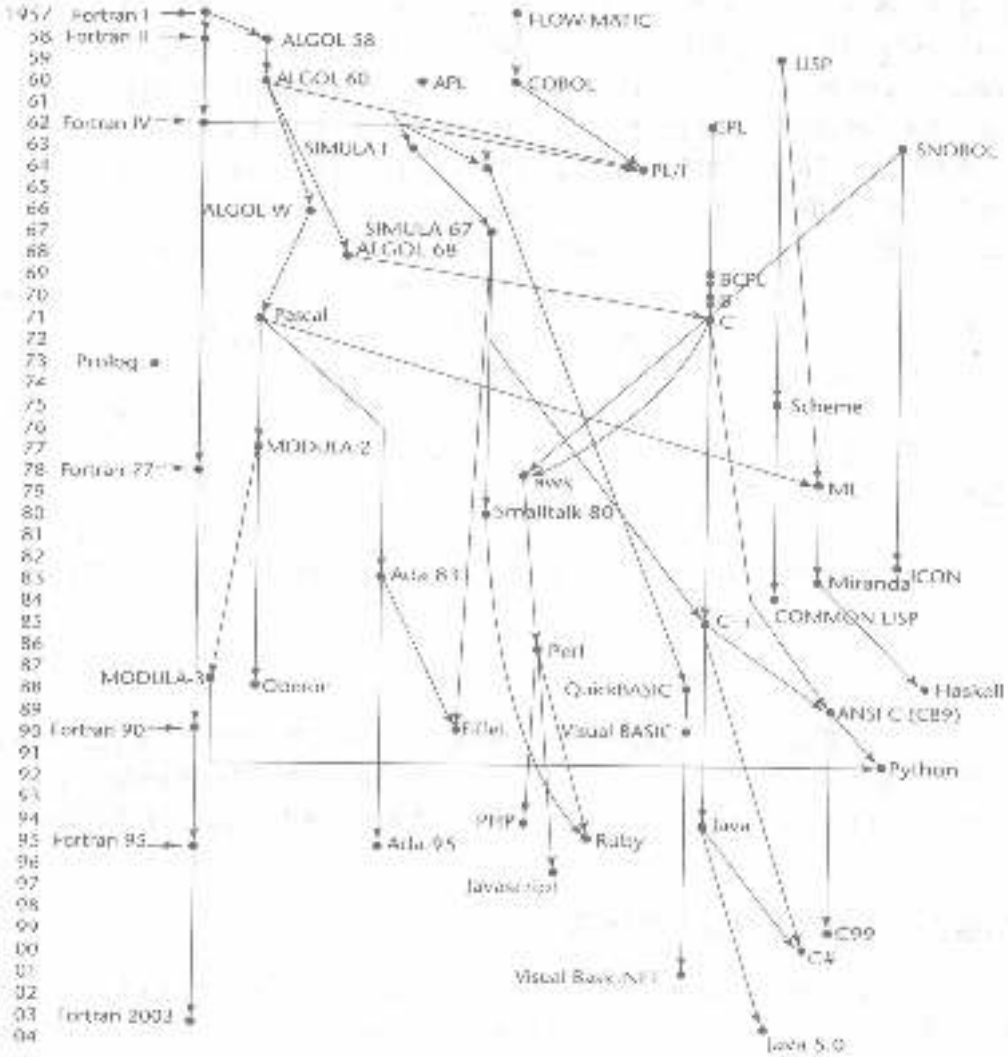


图2-1 常用高级语言的概貌

2.1.2 语言概述

Plankalkül 语言值人兆完整，并在数据结构方面具有一些最先进的特性。Plankalkül中最简单的数据类型是单个字位类型。从字位类型构造出整数与浮点数值类型，浮点类型使用了双补的标记法以及“隐藏字位”方案，现在人们使用这种方案来避免存储一个数值的标准化小数部分的最高字位。

除了这些常用的标量类型，Plankalkül语言还包括了数组和记录。这种记录又可以包括嵌套

的记录。

尽管这种语言没有显性goto语句，但它确实包括了一个类似于Ada语言的for语句的迭代语句，它还有一个带有上标的Fin命令。上标的值指示跳出特定数目的嵌套迭代，或者跳到一个新迭代周期的开始位置。Plankalkül语言还包括了一条选择语句，但它不允许使用else子句。

Zuse的程序中最有趣的特性之一是用数学表达式来表示程序变量之间的关系。这些表达式声明，在代码执行中它们出现的地方什么条件为真。这非常类似于今天在 Eiffel 程序设计语言（Meyer，1992）和公理语义中使用的断言。Eiffel 程序设计语言以及公理语义将在第3章讨论。

Zuse的手稿包含了远比在1945年之前编写的任何程序都更复杂的程序。这些程序包括数值数组排序程序、测试图连接性的程序、执行整数和浮点运算（包括计算平方根）的程序，并且还有对含有六个不同优先级的括号和运算符的逻辑公式进行语法分析的程序。也许最令人惊异的是他长达49页的国际象棋算法，尽管他并不是国际象棋游戏的专家。

如果在20世纪50年代初，有一位计算机科学家发现了 Zuse 关于 Plankalkül语言的描述，能够防碍这位科学家依照该语言的定义来实现这种语言的唯一因素，只能是 Zuse 所使用的各种标记法。Plankalkül语言中的每一条语句由2~3行代码组成。第一行代码最像目前同类语言使用的语句，第二行代码是可选择的，其中包含第一行所引用的数组的下标值。一个值得注意的有趣事实是，19世纪中叶的Charles Babbage在为他的分析引擎（Analytical Engine）编写的程序中使用了相同的方法来表示下标。在每一条Plankalkül语句的最后一行，包含了在第一行中所用到的变量类型的名字。当人们第一次看到这种标记法时会相当吃惊。

42

下面的赋值语句示例介绍了这种标记方法。这里表示，将表达式A[4]+1赋值给A[5]。其中的行标号V是下标的数值，而行标号S是数据类型。这个例子中的1.n表示一个n字位的整数。

```
    | A + 1 => A
V   | 4       5
S   | 1.n     1.n
```

我们现在只能猜想，如果Zuse的工作早在1945年或者即使是在1950年就广泛地为人们知晓，那么程序设计语言该会向什么方向发展。再猜想一下，假若他是工作于和平的环境，有其他科学家的帮助，而不是处于1945年的德国——在那种与世隔绝的环境中，那么他工作的影响方又是怎样的不同。

2.2 最小硬件的程序设计：伪代码

首先，注意这里使用到的伪代码一词与现在一般意义上的不一样。我们称本节中讨论的语言为伪代码，是因为它们在开发和使用时（20世纪40年代末到20世纪50年代初）被命名为伪代码。然而，在当代，它们已不再是伪代码。

20世纪40年代后期至50年代初期的计算机，远没有今天的计算机这么好用，除了速度慢、不可靠、价格昂贵和极其有限的存储空间外，那个时代的计算机因为缺少支持软件，而使编程十分困难。

当时没有高级程序设计语言，甚至没有汇编语言，因而程序设计都是使用机器代码来完成，这项工作真是既麻烦又错误百出。其中的一个问题是要使用数值代码来说明指令。例如，用数值编码14来表示ADD指令，而不是使用蕴涵意义的文字命名，哪怕仅仅是使用单个字母也好。这使得程序非常难于阅读。另一个更为严重的问题是要使用绝对地址，这使得程序的修改非常困难。例如，假设我们有一个存储于存储器中的机器语言程序，这个程序中的许多指令都指向

该程序中的其他位置，通常表示引用数据或者说明分支指令的目标。在程序中的任何位置（除了在程序的结尾）插入一条指令，就会改变指向该插入地址之后的所有指令的正确性。因为为了给新的指令让出位置，那些指令的地址都必须增加。要正确地增加这些地址，必须找到并修改所有指向这些地址的、在新增指令之后的那些指令。同样的问题也发生在删除一条指令的时候。当然在这种情况下，机器语言通常可以使用一条“无操作”指令来替代被删除的指令，以避免这种类型的问题。 43

这些是所有机器语言的典型问题，它们是促使人们发明汇编器以及汇编语言的主要动力。除此以外，当时大部分的程序设计问题是需要有浮点数的算术运算，以及一些使于数组使用的下标索引方法。然而这些功能都没有出现20世纪40年代末至50年代初期的计算机体系结构之中。这些缺陷自然地导致了较高级语言的发展。

2.2.1 短代码

第一种被称为短代码（Short Code）的新语言是由John Mauchly于1949年为 BINAC 计算机开发的。后来短代码被移植到一台 UNIVAC I 计算机上。在此后多年，这种语言一直是这类机器上程序设计的主要手段。尽管，因为关于原始短代码的完整描述从来没有被发表过，人们对其所知非常有限，然而一本 UNIVAC I 版本的程序设计手册却被保存了下来（Remington-Rand, 1952）。我们可以比较有把握地推断，这个版本与原始短代码的版本非常相似。

UNIVAC I 存储器的字具有72个字位。这72个字位被组合成为12个六字位的字节。短代码是由编好码的、将被求值的数学表达式组成的。这些代码是字节对（byte-pair）的数值形式，并且大多数的方程适宜被包含进一个字内。下面是这样一些代码的形式：

```
01   -      06   abs value      1n   (n+2)nd power
02   )      07   +              2n   (n+2)nd root
03   =      08   pause          4n   if <= n
04   /      09   (              5n   print and tab
```

变量（或者是存储空间的地址）用字节对代码来命名。存放常量的地址也是以这种方式表示。例如，X0和Y0可以是变量，语句

```
X0 = SQRT(ABS(Y0))
```

可以编码成为一个字，如 00 X0 03 20 06 Y0。起始代码 00 用作字的填充码。有趣的是，这种语言没有乘法代码；与在代数中一样，对于乘法的表示是将两个操作数相邻放置。 44

短代码不能译成机器码，相反，它由单纯解释器来实现。当时人们将这种过程称为自动程序设计。它显然简化了程序设计的过程，但却是以程序的执行时间作为代价的，短代码的解释过程大约比机器代码的运行要慢 50 倍。

2.2.2 快速编码

在其他地方，开发解释系统是为了使其能包括浮点数的操作，从而扩展机器语言。John Backus为IBM 701计算机开发的快速编码（Speedcoding）就是这种类型系统的一个例子（Backus, 1954）。这个快速编码解释器有效地将701机器转换成为一个具有虚拟的三地址浮点数的计算器。这个系统包括在浮点数据上进行四种算术运算的虚拟指令，以及进行平方根、正弦、正切、指数和对数等运算的虚拟指令。条件和无条件分支以及输入/输出的转换，也是该虚拟体系结构的组成部分。这个系统的局限性可以用一种情形来说明，这个系统在装载了解释器之后，

剩余的可用存储空间就只有700个字，并且运行一条ADD指令就要耗费4.2毫秒。另一方面，快速编码包括了给地址寄存器自动增值的新颖机制，而直到1962年，UNIVAC 1107计算机开发之后，相类似的机制才在硬件中出现。因为快速编码的这种自动增值的特性，矩阵乘法的运算可以由12条快速编码的指令来完成。Backus曾经宣称，在机器码中要花两个星期来编程的这类问题，使用快速编码可以在几个小时之内完成。

2.2.3 UNIVAC "编译" 系统

1951年至1953年之间，在UNIVAC公司，一个由Grace Hopper领导的小组开发了一系列的"编译"系统，被命名为A-0，A-1和A-2。这些系统采用了如同将宏指令扩展成汇编语言一样的方式，将伪代码扩展成机器码子程序，作为这些"编译器"源代码的伪代码仍是相当原始的，但采用这种代码使得源程序短了很多，所以相对于机器码来说，这已经是一个巨大的进步。Wilkes（1952）也独立地提出了一个相类似的过程。

2.2.4 相关的工作

大约就在同一时期，用以减轻程序设计工作难度的其他方法也被相继开发。剑桥大学的David J. Wheeler开发了一种方法，使用可以重新定位地址的块结构来部分地解决绝对地址的问题（Wheeler，1950）。后来，Maurice V. Wilkes（也是剑桥大学的）扩展了这种思路来设计一种汇编程序，这种程序可以组合选择的子程序，并分配存储空间（Wilkes等人，1951，1957）。这的确是一个重要的具有里程碑性质的进步。

45

我们应该提到，汇编语言是于20世纪50年代的初期发展起来的，这种语言与上述的伪代码相当不同。然而汇编语言对于高级语言的设计基本上没有产生什么影响。

2.3 IBM 704计算机与Fortran

1954年IBM 704计算机的问世，毫无疑问是计算机界有始以来最重大的进步之一。最主要的原因是由于这种计算机的功能促进了Fortran语言的发展。也许有人会争辩，假若没有IBM的704计算机和Fortran语言，不久也会有其他的公司开发出类似的计算机并产生与其相关的高级语言。然而，IBM却是第一个具有远见与资源而从事这项开发的公司。

2.3.1 历史背景

为什么从20世纪40年代末一直到50年代的中期，人们在这么长时期内都容忍了解释系统呢？主要的原因之一是当时的计算机上缺乏浮点数运算的硬件，因而必须使用软件来模拟所有浮点数运算，这是一个极为耗时的过程。当为中央处理器耗费很多的时间用于软件的浮点数处理过程，以至于解释过程以及检索模拟的额外开销就显得相对无关紧要了。只要是浮点数的运算必须由软件来完成，解释过程就成为可以接受的代价。然而当时有许多程序人员从不使用解释系统，而宁可采用手工编码的机器语言（或汇编语言）。在硬件上兼有索引检索与浮点数运算指令的IBM 704系统的诞生，至少为科学计算领域预示了解释器时代的结束。在硬件上包含浮点数运算指令，也消除了解释器带来的开销问题。

Fortran通常被誉为第一种编译式高级语言，但究竟谁应该获得实现第一种这类语言的荣誉却向无定论。Knuth和Pardo（1977）将这个殊荣给予Alick E.Glennie，因为他在Manchester Mark II计算机上构造了自动编码（Autocode）编译器。Glennie是在位于英国的Halstead兵营的皇家军备研究所（Royal Armaments Research Establishment）开发的这种编译器。这个编译器于

1952年9月之前已经投入使用。然而根据John Backus的观点（Wexelblat，1981，p.26），他认为Glennie的自动编码编译器是在很低的层次上，并且是面向机器的，因而不应该认为它是一个编译系统，Backus则将这个殊荣给予麻省理工学院（MIT）的Laning和Zierler。

Laning和Zierler的系统（Laning and Zierler，1954）是实现了的第一种代数翻译系统。这里代数的意思是指它能够翻译算术表达式，能够进行数学函数的调用，并且还包括了数组。这个系统首先以实验样机的形式于1952年夏天在麻省理工学院（MIT）的旋风（Whirlwind）计算机上实现，并于1953年5月之前在同一机器上完成了一种更为实用的实现形式。这个翻译器在程序中的每一个计算公式或每一条计算表达式产生一个子程序的调用。这种源语言很容易阅读，它包括的唯一的机器指令是分支指令。虽然这项工作超前于Fortran，但是却从来没有跨出过麻省理工学院。 46

尽管有这些较早期的工作，Fortran仍旧是第一种为人们广泛接受的编译式高级程序设计语言。下面的几节将按年代记述这一重要的发展。

2.3.2 设计过程

早在704系统于1954年5月被推出以前，Fortran语言的计划就已经开始了。IBM的John Backus和他的工作小组于1954年11月之前发表了题目为"FORTRAN：IBM的数学公式翻译系统(The IBM Mathematical FORmula TRANslating System：FORTRAN)"的报告（IBM，1954）。这份文件描述了被称为Fortran 0的Fortran语言实现之前的第一个版本。它大胆地声明Fortran将提供与手工编码程序一样的高效率，以及与解释式伪代码系统一样容易的程序设计。过于乐观的一面是，这个文件宣称Fortran将会消除代码中的错误，并能够免除程序调试的过程。在此前提下，Fortran的第一个编译器几乎没有包括语法错误的检测。

当时开发Fortran的环境有下列特点：(1) 计算机仍然是小型、速度慢且相对不可靠；(2) 计算机的主要应用领域是科学计算；(3) 不存在计算机编程的高效率方法；(4) 因为计算机的代价比编程人员的代价高，因而第一代Fortran编译器的主要目标是高速度的目标代码。Fortran语言早期版本的特征直接与当时的语言设计环境相关。

2.3.3 Fortran I概况

Fortran 0是在实现期间被修改的，这个修改过程从1955年1月开始，一直持续到1957年4月Fortran编译器被推出为止。这个被实现了的语言称为Fortran I。在1956年10月出版的第一本关于Fortran语言的*Programmer's Reference Manual*（IBM，1956）对Fortran I进行了描述。Fortran I包括了输入/输出格式化、长达六个字符的变量名（在Fortran 0中只包括有两个字符的变量名），用户定义的子程序（尽管当时的子程序是不能分别编译的），IF选择语句以及DO循环语句。

Fortran I中所有的控制语句都是以704机器指令为基础的。现在不太清楚，究竟是704机器的设计人员支配了Fortran I控制语句的设计，还是704机器的设计人员接受了Fortran I语言的设计人员的建议，从而构造这些机器指令。

Fortran I语言中没有数据类型的说明语句，名字由I、J、K、L、M和N字母开始的变量被暗示为整数类型，而由其他所有字母开始的则被暗示为浮点数类型。这种字母选择的约定是基于当时的使用习惯，即整数主要用来作为下标，而科学人员通常使用I、J和K作为下标。Fortran语言的设计人员出手更为大方，比这种通常的用法又多给了二个字母。 47

Fortran语言开发小组在语言的设计中提出的最大明宣称是，由Fortran编译器产生的机器码

与手工编码产生的代码有大致相同的效率[9]。在Fortran语言实际推出之前，这个宣称使得潜在的用户产生很大疑虑，并且大大降低了人们对Fortran原有的兴趣。然而出乎人们意料的是，Fortran开发小组在语言的效率上几乎达到了它声称的目标。建造第一台编译器的18个人年工作中的大部分，是用于编译器的优化上；而这项工作的结果是惊人的高效率。

1958年4月所做的调查结果显示了Fortran语言的初步成功。粗略来说，当时704机型的一半程序是用Fortran语言编写的，尽管仅在一年之前，大部分程序设计人员还持着极端怀疑的态度。

2.3.4 Fortran II概况

1958年的春天发布了Fortran II编译器。这个系统对Fortran I编译系统中的许多错误都进行了修改，并且给Fortran语言增加了一些重要的特性。其中最重要的特性是子程序的独立编译的功能。没有独立编译，在一个程序中进行任何改动，都需要重新编译整个程序。Fortran I缺乏子程序独立编译功能，704机器不可靠，两者结合在一起，使得程序的长度受到很大限制：程序最多只能有300～400行（Wexelblat，1981，p. 68）。较长的程序很少能够在死机之前完成编译。Fortran II允许将预先编译好的子程序机器码包括进程序，这极大地缩短了编译的过程。

2.3.5 Fortran IV，77，90，95和2003

Fortran III从来没有得到广泛应用，而Fortran IV成了当时最广泛使用的一种程序设计语言。它的发展时期是从1960年到1962年，并在1966年被标准化成为Fortran 66（ANSI，1966）。Fortran 66这个名称很少被使用。在许多方面，Fortran IV是Fortran II的扩展，其中最重要的扩展 [48]
包括变量类型的显式声明，If逻辑结构，以及将子程序作为参数传递给其他子程序的能力。

Fortran 77后来又替代了Fortran IV，并在1978年成为了新的标准（ANSI，1978a）。Fortran 77保持了Fortran IV的大部分特性，并且新增了字符串的处理，逻辑循环控制语句，以及If与可选择的Else所构成的一个子句。

Fortran 90（ANSI，1992）与Fortran 77有着巨大的不同，最大的变动是加入了动态数组，记录，指针，多选择语句和模块。而且，Fortran 90子程序也能被递归调用。

Fortran 90的定义中包括了一种新概念，即需要从早期Fortran版本中删掉一些语言特性。尽管Fortran 90包含了Fortran 77的所有特性，但有两个表列出的特性是专门留待未来Fortran语言的新版本删除的。废弃特性表列出了可能在Fortran 90的下一个版本中删除的特性。

Fortran 90包括了两种简单的语法改进，这些改进改变了程序与文字描述语言的外观。首先，去掉了对于代码固定形式的要求，即要求在特殊的字符位置编写语句的特定部分。例如，语句的标记只能出现在前面的五个位置上，而语句只能从第七个位置开始。这种死板的代码形式是设计用于穿孔卡的。第二种改变是语言名称的正式拼法由FORTRAN改变为Fortran。这种改变是当Fortran程序中传统协定的改变向改变，即关键字以及标识符全部使用大写字母。现在的协定是，只有关键字及标识符的第一个字母使用大写。

Fortran 95（INCITS/ISO/IEC，1997）是进一步发展该语言的结果。但是与前一版相比，只作了很少的改动。为了改进Fortran程序的并行化，引入了一种新的迭代结构Forall。

Fortran语言的最新版本Fortran2003（Metcalf *et al.*，2004）引入了带参数的派生类型，它支持面向对象程序设计，过程指针和与C语言的互操作性。

---

[9] 事实上，Fortran语言一经问世，自由们的编译器产生高效码的水平也与手工编码的相当或更高一些；不然，用户将不会接受这种语言。

2.3.6 评估

最初的Fortran语言设计小组认为，语言设计仅仅是设计翻译器这个关键性工作的必需前奏，他们甚至从来都没有想过，Fortran语言会应用于非IBM制造的计算机上。确实，只是因为IBM 704的后代机器，IBM 709，在推出704机器上的Fortran编译器之前就宣告诞生，他们才被迫不得不考虑为其他的IBM机器建造Fortran编译器。Fortran语言对各种计算机使用的影响，连同后来所有的程序设计语言都得益于Fortran语言的事实，的确让人感叹，它当初的设计人员的目标是多么具有远见。

Fortran I以及在Fortran 90之前的所有后代语言具有的一个能够允许高度优化的编译器特性是，在运行之前，将所有变量的类型及其存储位置都固定下来，而在运行期间不再分配新的变量或者存储空间，这是以牺牲灵活性的代价来换取简单性和高效率。但是，这种方式排除了递归式子程序的可能性，并且使得难以实现动态生长或改变形状的数据结构。当然，在Fortran初期版本的发展时期，程序的开发主要是用于数值计算，它们与近代的软件项目相比十分简单，因而当时的这种特性还不算巨大。

总而言之，Fortran语言的成功，显著而且是永远地改变了计算机的使用方式，这种说法决不言过其实。当然，这在很大程度上是由于它是第一种广泛应用的高级语言。比较后来的程序设计语言原理，以及后来开发的程序设计语言，人们必须接受的事实是，Fortran语言的早期版本在许多方面都存在着缺陷，这正如同不能跨越时代，将1910年的T型福特汽车与2005年的福特野马型相比一样。另外，尽管Fortran语言也有不足，但是，投入Fortran软件的巨大投资是使它始终立于最广泛应用的高级语言之列的主要原因之一。

ALGOL 60语言的设计者之一Alan Perlis，在1978年曾就Fortran说道，“Fortran语言是计算世界的混合语言。它是普通人使用的语言。这样说是褒义的，而不是贬义的。因为它已经成为充满活力的商务世界里最顽强的一颗草，所以它已经生存下来了，并且还将生存下去。”（Wexelblat，1981，p.161）

下面是Fortran 95程序的一个例子：

```
! Fortran 95 Example program
!  Input:  An integer, List_Len, where List_Len is less
!          than 100, followed by List_Len-integer values
!  Output: The number of input values that are greater
!          than the average of all input values
Implicit none
Integer Dimension(99):: Int_List
Integer :: List_Len, Counter, Sum, Average, Result
Result= 0
Sum = 0
Read *, List_Len
If ((List_Len > 0) .AND. (List_Len < 100)) Then
! Read input data into an array and compute its sum
   Do Counter = 1, List_Len
      Read *, Int_List(Counter)
      Sum = Sum + Int_List(Counter)
   End Do
! Compute the average
   Average = Sum / List_Len
! Count the values that are greater than the average
   Do Counter = 1, List_Len
      If (Int_List(Counter) > Average) Then
         Result = Result + 1
```

```
        End If
      End Do
! Print the result
      Print *, 'Number of values > Average is:', Result
Else
      Print *, 'Error - list length value is not legal'
End If
End Program Example
```

2.4 函数式程序设计语言：LISP

第一种函数式程序设计语言是为了提供表数据处理的语言特性而发明的。后来人们对函数式程序设计语言的需要超出了它最初在人工智能（AI）领域的应用。

2.4.1 人工智能与链表数据处理的开始

从20世纪50年代中期开始，人们在许多不同的领域对人工智能产生了兴趣。一些兴趣来自语言学领域，一些来自心理学领域，而另一些则来自数学领域。语言学家们关心的是自然语言的处理；心理学家们感兴趣的则是模拟人类大脑信息的存储与检索、以及其他一些大脑的基本过程；数学家们感兴趣的是某些特定智能过程的机器化，例如定理的证明。所有这些方面的研究达成了一致的结论：必须开发一些方法，使得计算机能处理链表中的符号数据。而当时几乎所有的计算都是数组的数值运算。

表处理的概念是由Allen Newell、J. C. Shaw和Herbert Simon提出的，这个概念第一次发表是在他们的一篇经典的论文中，该论文描述了最早的人工智能程序之一、名为"Logical Theorist"[○]的程序，并且还描述了一种可以实现这个程序的程序设计语言（Newell and Simon, 1956）。这个被命名为IPL-I（Information Processing Language I，信息处理语言 I）的语言从来没有被实现过。它的下个版本被命名为IPL-II，在兰德（Rand）公司的Johnniac计算机上被实现了。IPL 语言的开发工作一直持续到1960 年，在这一年，发表了关于IPL-V语言的描述（Newell and Tonge, 1960）。然而IPL语言的低层次妨碍了它的广泛使用。IPL 语言实际上是为一种假想的计算机而开发的汇编语言，实现于一个包括了表处理指令的解释器上。因为它第一次的实际实现是在臺无名气的 Johnniac 机器上，这也成为妨碍 IPL 语言流行的另一个原因。

IPL语言所作的贡献是它们关于表结构的设计，以及它们所示范的表处理方法的可行性与实用性。

51

IBM从20世纪50年代中期开始对人工智能感兴趣，并且选择了定理证明作为其示范领域。当时，Fortran语言初项目仍然在进行，Fortran I编译器的高昂代价使得IBM认为，应该将他们的表处理功能附加到Fortran语言上，而不是以一种新的语言形式出现。因而IBM设计并实现了Fortran表处理语言（Fortran List Processing Language，FLPL），并将其作为Fortran语言的一种扩展形式。FLPL被用来建造一个平面几何学定理证明器，这在当时被认为是机械定理证明领域里一个最容易的区域。

2.4.2 LISP的设计过程

麻省理工学院的John McCarthy于1958年在IBM信息研究部门得到一个夏天的工作位置。那

○ Logical Theorist发现了命题演算的一些定理证明。

个更大的目标是研究符号运算，并且为进行这种类型的运算开发一组语言需求。作为先行例子的问题范例，他选择了代数表达式的差分。他从这项研究里导出了一系列的语言需求，其中包括数学函数的控制流程方法：递归与条件表达式。而在当时，唯一的高级语言Fortran I中没有这些方法。

从符号差分的研究中得出的另一个需求是动态地分配链表结构的需要，以及将废弃的表结构以某种隐式方式解除分配。McCarthy所要求的仅仅是不允许一些显性解除分配语句搅乱他漂亮的差分算法。

因为IPL语言不支持递归运算、条件表达式、动态存储空间的分配或动态隐性解除分配，McCarthy当时十分清楚：必须要发明一种新的语言。

当McCarthy于1958年秋天回到麻省理工学院时，他和Marvin Minsky用从电子研究实验室得到的研究基金组成了麻省理工学院人工智能项目。在这个项目中所做的第一个重要工作就是开发一个表处理系统。他们首先实现了McCarthy所提议的、被称为“听取建议者”（Advice Taker）的程序。[1]这项应用成为开发表处理语言LISP的动力。LISP语言的第一个版本有时候被称为纯LISP语言，当为它是一种纯粹的函数式语言。下面的小节将描述纯LISP语言的发展过程。

2.4.3 语言概述

2.4.3.1 数据结构

纯LISP语言只有两种数据结构：原子与表。原子是具有标识符形式的符号，或者是数值的文字常数。在链表结构中存储符号信息的概念是自然的，而且这种概念已经被应用于IPL II中。这种结构允许在任何位置进行插入与删除操作，这两种操作被认为是表处理的必要部分。然而，当LISP语言的开发最终完成以后，LISP程序却几乎不需要这两种操作。

52

表结构的说明方式是将其元素包括在括号之内。简单表的元素仅限于原子，具有下面的形式：

```
(A B C D)
```

嵌套表结构也用括号来说明。例如，下面的表结构

```
(A (B C) D (E (F G)))
```

是由四个元素组成。第一个元素是原子A，第二个元素是子表（B C），第三个元素是原子D，第四个元素是子表（E (F G)），而它又以子表（F G）作为它的第二个元素。

在计算机内部，表通常被存储为单向链表结构，在这种结构中，每个节点具有两个指针，并且每个节点代表一个表元素。一个原子节点的第一指针指向这个原子的某种表示，例如，这个原子的符号或者数值，或者指向一个子表的指针。一个子表元素节点的第一指针指向这个子表元素的第一个节点。在以上两种情况下，一个节点的第二指针都指向该表的下一个元素。表是由一个指向它的第一个元素的指针来引用的。

图2-2描述了上面例子中两个表的内部表示。请注意，表的元素是水平显示的。表的最后一个元素没有后继元素，因此它的链接是NIL。在图2-2中，在元素的位置上使用一条斜线来表示NIL。子表也使用同样的结构表示。

---

(1) 听取建议者（Advice Taker）程序，使用一种形式语言编写的句子来表示信息，并且使用一种逻辑引用过程来确定步骤。

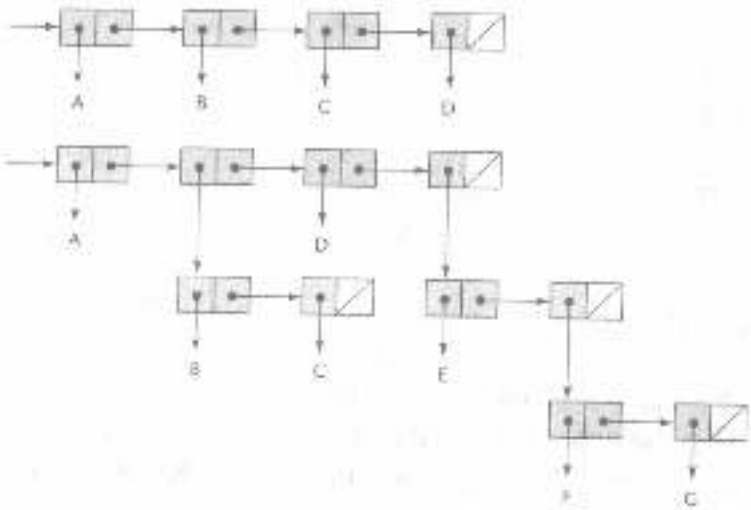


图2-2 两个LISP表的内部表示

2.4.3.2 函数式程序设计的过程

LISP语言是作为函数式程序设计语言而设计的，函数式程序中的所有运算都是通过在自变量上应用函数来完成。在命令式语言程序中所具有的大量的赋值语句及变量，在函数式语言的程序中都是不必要的。此外，迭代过程可以被声明为递归函数的调用，这样变得循环也不需要了。这些函数式程序设计的基本概念，使得它与命令式语言程序设计显著不同。

2.4.3.3 LISP的语法

LISP 语言与命令式语言有着很大的不同，首先因为它是一种函数式程序设计语言，其次因
53 为 LISP 的程序看起来与 Java或者 C++ 程序有很大不同。例如，Java 的语法是一种英文与代数的复杂混合，而LISP 的语法则是一种简单性的典范。LISP的程序代码与数据具有完全相同的形式：即被括号包括的表。让我们再一次考虑表

```
(A B C D)
```

当你将它解释成数据时，它是一个四个元素的表。而当你将它视作代码时，它是一个名称为A的函数作用于三个参数：B、C和D。

2.4.4 评估

LISP语言完全垄断人工智能应用领域长达四分之一世纪之久。许多导致LISP语言具有极低效率名声的因素已经被除去。现在所实现的多种系统都是编译式的，产生的代码比在解释器上运行源程序要快得多。除了在人工智能领域中的成功之外，LISP是函数式程序设计领域内的先驱，事实已经证明。这个领域是程序设计语言研究方面一个充满活力的区域。如在第1章中所述，许多程序设计语言的研究人员相信，在软件并发中，函数式程序设计是比命令式语言优越得多的工具。

下面是一个LISP程序的例子：

```
;  LISP Example function
;  The following code defines a LISP predicate function
;  that  takes two lists as arguments and returns True
54 ;  if the two lists are equal, and NIL (false) otherwise
   (DEFUN equal_lists (lis1 lis2)
```

```
    (COND
      ((ATOM lis1) (EQ lis1 lis2))
      ((ATOM lis2) NIL)
      ((equal_lists (CAR lis1) (CAR lis2))
                (equal_lists (CDR lis1) (CDR lis2)))
      (T NIL)
    )
)
```

2.4.5 LISP的两种后代语言

LISP 语言的两种方言，COMMON LISP和Scheme，现在被广泛地应用。我们将在下面的几节简略地进行讨论。

2.4.5.1 Scheme

Scheme方言于20世纪70年代中期出现于麻省理工学院（Sussman and Steele，1975）。它的特征为小巧、独特地使用静态作用域（将在第5章中讨论），以及将函数处理为第一类实体。作为第一类实体，Scheme中的函数可以是表达式的值和表中的元素，它们可以被赋予变量，可以被作为参数传递，或者作为函数调用的返回值。早期的 LISP 版本并没有提供所有的这些功能，也并没有使用静态作用域。

作为具有简单语法与语义的小型语言，Scheme 方言很适合教育应用领域，例如用于函数式程序设计课程，以及程序设计一般性介绍课程。如前面提到的，我们将在第15章描述 Scheme 方言的细节。

2.4.5.2 COMMON LISP

在20世纪70年代至80年代早期，人们开发并且运用了LISP的许多不同的方言。这导致了我们熟悉的移植性问题。为了解决这个问题，人们开发了COMMON LISP（Graham，1996）。COMMON LISP方言的创建意图是为了将开发于20世纪80年代初期的几种LISP方言（包括Scheme）的特性结合进一种语言之中。身为这样一种混合物的COMMON LISP成为一种大而复杂的语言。然而它的基础还是纯 LISP 语言，因此它的语法、基本功能以及基本的本质都来自于纯 LISP 语言。

认识到动态作用域所能提供的灵活性以及静态作用域的简单性，COMMON LISP 兼有这两种作用域。而变量的默认作用域为静态作用域，但是通过声明变量为 special，该变量的作用域就可以变成是动态的。

COMMON LISP方言具有大量的数据类型与数据结构，其中包括记录、数组、复数以及字符串。它还具有包的形式用来将函数和数据的集合模块化，以提供访问控制。COMMON LISP方言将在第15章中进一步描述。 55

2.4.6 相关语言

元语言（MetaLanguage，ML）（Ullman，1998）最初由Robin Milner于20世纪80年代，在爱丁堡大学为一个名为“用于可计算函数的逻辑”（Logic for Computable Functions，LCF）的程序验证系统设计的元语言（Milner et al.，1990）。元语言基本上是一种函数式语言，但是它也支持命令式程序设计。不同于LISP以及Scheme方言，元语言中每一个变量的类型以及表达式的类型都可以在编译时决定。类型与变量的实体相关联，而不是与变量的名字相关联。表达式的类型是从表达式所处环境来推断的，这些我们将在第7章里讨论。

不同于LISP和Scheme方言，元语言不使用源于Lambda表达式的，运用括号的函数式语法，

相反，元语言的语法仿效了命令式语言的语法，如Java和C++。

Miranda语言是由David Turner于20世纪80年代初期在英国坎特伯雷的肯特大学开发的（Turner，1986），Miranda部分地基于ML、SASL和KRC语言。而Haskell语言（Hudak and Fasel，1992）则在很大程度上是基于Miranda语言的。类似于Miranda语言，Haskell是一种纯函数式语言，不具有变量与赋值语句。Haskell的另一个十分独特的特征是使用惰性求值，即仅当需要表达式的值时才进行计算。这种特性导致了这种语言中一些出人意料的功能。

元语言和Haskell语言都将在第15章中简略地讨论。

2.5 迈向成熟的第一步：ALGOL 60

ALGOL 60对后来的程序设计语言有着巨大的影响，因此它是语言历史回顾中的重要核心。

2.5.1 历史背景

ALGOL 60的出现源自人们企图设计一种科学应用领域的通用语言。在1954年年底之前，Laning和Zierler的代数系统已经经过了一年多的运行，并且有关Fortran语言的第一个报告已经发表了，到了1957年，Fortran语言已成为现实，而且其他的一些高级程序设计语言也正在开发之中。这些语言中最有名的是由Carnegie Tech的Alan Perlis设计的IT语言，以及两种为UNIVAC
56 计算机设计的语言，即MATH-MATIC语言和UNICODE语言。语言的繁多使得用户之间难以交流。此外，所有这些新语言都是围绕着单一的计算机体系结构开发的，一些用于UNIVAC计算机，而另一些则用于IBM的700号系列机器。针对这种语言种类激增的情况，美国的一些主要计算机用户组织，包括IBM科学计算用户团体（the IBM scientific user group，SHARE）和UNIVAC科学计算交换（UNIVAC Scientific Exchange，USE，一个大规模的UNIVAC科学计算用户组织），于1957年5月10日向美国计算机协会（ACM）递交了一份请求信，要求组成一个委员会来研究并推荐一种通用的程序设计语言。虽然Fortran是可能的候选语言，但由于它当时是归IBM所独有，因此不能成为这样一种通用语言。

在此前的1955年，应用数学与力学协会（德文缩写为GAMM）也已经成立一个委员会，为各种类型的计算机设计一种通用的、独立于机器的算法语言。这种对于新语言的渴求，部分是出于欧洲人对IBM占支配地位的惧怕。然而，在1957年年底，几种高级语言相继在美国诞生，这才使得这个GAMM的下委员会认为创建通用语言的工作也应该包括美国人，而该委员会向美国计算机协会（ACM）呈递了一封邀请信。1958年4月，GAMM的Fritz Bauer将正式提议送交ACM之后，这两个组织正式同意联合开发一个语言设计项目。

2.5.2 早期设计过程

GAMM和ACM决定，每个协会派四个成员参加第一次联合设计语言的会议。这个会议于1958年5月27日至6月1日在苏黎世举行，一开始它就为新的语言设立了下列目标：

- 这种语言的语法应该尽可能地接近数学的标准记法，并且，由这种语言编写的程序在没有解释的情况下应该是可读的。
- 它应该可以被用作出版物中描述计算过程的语言。
- 由这种新语言编写的程序应该能够被计算机翻译成机器语言。

这里的第一个目标说明这种新语言将要用于科学领域的程序设计，这是当时计算机的主要应用领域。第二个目标对当时的计算机界是全新的要求。最后一个目标则对任何程序设计语言

显然都是必要的。

由于人们观点的不同，苏黎世会议有可能产生重大的结果，也有可能引出永无休止的争论。实际上这两种情况都存在。会议的本身包括了无数的妥协，既是在与会的个人之间，也是在大西洋两岸之间。在有些情况下，这些妥协有时计较的是对于全球的影响，而不是一些重要的问题。是使用逗号（欧洲方式）还是使用句号（美国方式）来表示小数点的问题，就是一个例子。 [57]

2.5.3 ALGOL 58概况

在苏黎世会议上设计的语言被命名为国际算法语言（International Algorithmic Language，IAL）。在设计期间被提议称为ALGOL语言，意为算法语言，但是因为这个名称没有反映出联合委员会的国际性而遭到了拒绝。然而在第二年，它的名称又被改成了ALGOL，这个语言后来以ALGOL 58闻名于世。

在许多方面，ALGOL 58是Fortran的一个后代语言，这是很明显的。它将Fortran的许多语言特性通用化了，并增加了一些新的结构和新的概念。这些通用化，一部分与不允许将语言绑于任何特定机器的目的有关，而另一些则是企图使这种语言更灵活、功能更强大。在这些努力之下，一种罕见的简单与完美的组合出现了。

ALGOL 58形式化了数据类型的概念，虽然仅是非浮点数的变量需要有显性的类型声明。ALGOL 58增加了复合语句的概念，这个概念被几乎所有后续程序设计语言所吸收。一些Fortran语言的特性在被通用化之后包括进了ALGOL 58之中：标识符可以具有任意的长度，这与Fortran语言的最多六个字符的限制相反；数组可以具有任意的维数，这不同于Fortran语言所规定的最多只能有三个维数；数组的下界可以由程序人员来定义。然而在Fortran语言中，数组下界被隐性地规定为1；允许嵌套选择语句，而在Fortran语言中则不能。

ALGOL 58以相当不寻常的方式获得了它的赋值操作符。Zuse曾经在Plankalkül语言中使用

表达式 => 变量

的形式作为赋值语句。尽管Plankalkül语言还没有发布，但是ALGOL 58委员会的一些欧洲成员熟悉这种语言。委员会考虑过Plankalkül语言的这种赋值语句的形式，但是因为有关字符限制性的争论[⊖]，大于号被改成了冒号。然后多半是由于美国人的坚持，整个语句被改为下面的形式：

变量 := 表达式

欧洲人则更喜欢与此相反的表达形式。然而，那就正好是Fortran的颠倒形式。 [58]

2.5.4 ALGOL 58报告的接受

1958年12月发表的关于ALGOL 58语言的报告（Perlis and Samelson，1958）得到了极其热烈的反响。在美国，更多地是将这种新语言视为一种程序设计语言设计思想的集合，而不仅仅是一种通用的标准语言。实际上，ALGOL 58语言报告的本意并不是提供一种完成了的产品，而只是一份供国际讨论的初步文案。然而，三个主要从事语言设计与实现工作的组织已经将这个报告作为他们工作的基础。在密歇根大学（the University of Michigan）诞生了MAD语言（Arden *et al.*，1961）；美国海军的电子科学集团（The U.S. Naval Electronics Group）实现了NELIAC语言（Huskey *et al.*，1963）；系统开发公司（System Development Corporation）设计并实现了

⊖ 那时的卡片穿孔还不包括大于号。

JOVIAL语言（Shaw，1963）。JOVIAL是“国际代数语言的Jules版本”（Jules'Own Version of the International Algebraic Language）名称的缩写，它代表了这种基于ALGOL 58并唯一得到广泛应用的语言（Jules是JOVIAL的设计人员之一，他的全名为Jules I. Schwartz），JOVIAL之所以被广泛使用，是因为它曾经在长达四分之一个世纪中被用作美国空军官方的科学计算语言。

美国计算机界的其余部分对待新语言就不是那么友好。起初，IBM及其主要科学计算用户组织SHARE似乎都欢迎ALGOL 58。IBM在ALGOL 58的报告发表之后不久，即开始了这种语言的实现工作，SHARE也成立了一个称为SHARE IAL的子委员会来研究ALGOL 58。该子委员会接下来建议ACM应该实行ALGOL 58语言的标准化，并建议IBM为它的所有700号系列计算机都实现这种语言。然而这种热情是短暂的。在1959年春季，IBM和SHARE都通过了Fortran体验期，在新语言的开发及第一代编译器实用方面，以及在培训及规劝用户使用新语言方面，他们体验到了开始一种新语言所具有的所有痛苦与代价。到1959年中期，IBM和SHARE显示了他们对于Fortran的既定兴趣，并决定在IBM 700系列机器上保留Fortran，以作为他们的科学计算语言，并因此最终放弃了ALGOL 58。

2.5.5 ALGOL 60的设计过程

在1959年，关于ALGOL 58的辩论在欧洲和美国激烈地展开，大量有关修改和增加功能的提议在欧洲的*ALGOL Bulletin*以及*Communications of the ACM*上发表。在1959年中，最重要的事件之一是苏黎世委员会在“信息处理国际会议”上发表的工作报告。在这里，Backus介绍了他用来描述程序设计语言语法的新标记法，后来这种方法以巴科斯-诺尔范式（Backus-Naur form，BNF）命名而闻名于世。关于巴科斯-诺尔范式，我们将在第3章详细介绍。

1960年1月，第二次ALGOL会议在巴黎举行。这次会议的议题是要讨论80项已经正式提交
59 的提案。丹麦的Peter Naur尽管不是苏黎世委员会的成员，但他十分积极地参与了ALGOL的开发。也正是他创建并且出版了*ALGOL Bulletin*。Naur花费了大量的时间来研究Backus介绍巴科斯-诺尔范式的文章，并决定应该正式使用巴科斯-诺尔范式来描述1960年ALGOL会议的结果。在对巴科斯-诺尔范式进行了些微改动之后，他用巴科斯-诺尔范式写出了一份关于所提议的新语言的描述，并在会议开始时散发给与会的1960小组成员。

2.5.6 ALGOL 60语言概述

1960年的会议虽然仅持续了六天，但是对ALGOL 58进行了重大的修改。其中最重要的新发展有下列各项：

- 引入了块结构的概念。这将允许程序人员通过引入新的数据环境或作用域，进行程序部分的局部化。
- 允许使用两种不同的方式进行子程序参数的传递：按值传递与按名传递。
- 允许递归过程。ALGOL 58关于这一问题的描述不是十分清楚。请注意，虽然递归运算对于命令式语言还是崭新的概念，但LISP在1959年就已经提供了递归功能。
- 允许栈动态数组。栈动态数组的下标范围由变量来指定，所以数组的大小是在数组分配存储空间时就被确定的，这发生于执行到数组声明之时。我们将在第6章中详细描述栈动态数组。

在会议上，人们还提出了一些可能对语言的成功或失败产生巨大影响的语言特性，然而这些提议都遭到了否决，其中最重要的包括格式化输入和输出语句。之所以未被采纳，是因为人

们认为这些语句过于依赖机器。

ALGOL 60的报告于1960年5月发表（Naur，1960），许多疑义之处仍然保留在语言的描述之中，第二次会议已经预定于1962年4月在罗马召开，以讨论遗留问题。而在第二次会议上，会员们只是处理了一些问题，没有被允许增加语言的内容。这次会议的结果发表在题为“ALGOL 60算法语言的修改报告”之中（Backus *et al.*，1962）。

2.5.7 ALGOL 60的评估

在某些方面，ALGOL 60是一个巨大的成功，而在另一些方面，它却是一个让人灰心的失败。它的成功在于，它几乎立刻就成为计算机界出版刊物中，交流算法的唯一可以接受的形式化方法，并且在之后的20多年间，它始终是发表算法的唯一语言。自1960年以来所设计的每一种命令式语言，都或多或少得益于ALGOL 60。大部分的语言在事实上都是直接或间接的ALGOL 60的后代语言，例如PL/I，SIMULA 67、ALGOL 68、C、Pascal、Ada、C++以及Java语言。

ALGOL 58/ALGOL 60语言的设计工作包括了一系列的“第一”。这是第一次一个国际组织尝试设计一种程序设计语言，它是被设计的第一种独立于机器的语言，它也是第一种形式地描述语法的语言，巴科斯-诺尔范式的形式化方法的成功应用，开拓了计算机科学中的一些重要领域：形式语言、语法分析理论，以及基于巴科斯-诺尔范式的编译器设计。最后一点，ALGOL 60语言的结构影响了计算机的体系结构，其中最令人瞩目的例子，是一种扩展的ALGOL 60，被用作一系列大型计算机的系统语言，如Burroughs的B5000，B6000和B7000型机器，这些机器设计了硬件栈结构，以便于高效率地实现语言中的块结构及递归子程序。

然而就像硬币有正反两面一样，这个故事还有着另外的一面，ALGOL 60从来没有在美国得到广泛的应用，甚至就是在欧洲，即使这种语言在欧洲远比在美国受欢迎，它也从来没有成为过主要的语言。有许多原因使得它不被人们接受。原因之一，是ALGOL 60的一些语言特性过分灵活，使得人们难于理解且难于实现。说明这个问题的最好的一个实例是以按名传递方法传递参数给子程序，我们将在第9章解释这种方法。实现ALGOL 60的困难则可由Rutishauser于1967年所做的报告来证明，这就是，几乎不存在一种实现包括了完整的ALGOL 60语言（Rutishauser，1967，p. 8）。

在语言中缺乏输入和输出语句，也是ALGOL 60不能够被接受的另一个主要原因。依赖于实现的输入/输出，使得ALGOL 60的程序很难被移植到其他计算机上。

与ALGOL 60相关联的，对于计算机科学最重要的贡献之一是巴科斯-诺尔范式，但这同时也是使得ALGOL 60不被接纳的原因。虽然现在巴科斯-诺尔范式被认为是一种简单和完美相结合的语法描述方法，但对于1960年的世界，它却是十分奇怪和复杂的。

最后，尽管还有许多其他问题导致ALGOL 60不能被广泛应用，但用户们对于Fortran语言的固守以及缺乏IBM的支持，或许才是其中最重要的原因。

从ALGOL 60的语言描述总是伴随着歧义性和模糊性的意义上来说，ALGOL 60语言的工作从来就没有真正地完成（Knuth，1967）。

下面是ALGOL 60程序的一个例子：

```
comment ALGOL 60 Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all the input values  ;
begin
```

```
    integer array intlist [1:99];
    integer listlen, counter, sum, average, result;
    sum := 0;
    result := 0;
    readint (listlen);
    if (listlen > 0) ∧ (listlen < 100) then
      begin
  comment Read input into an array and compute the average;
      for counter := 1 step 1 until listlen do
        begin
        readint (intlist[counter]);
        sum := sum + intlist[counter]
        end;
  comment Compute the average;
      average := sum / listlen;
  comment Count the input values that are > average;
      for counter := 1 step 1 until listlen do
        if intlist[counter] > average
          then result := result + 1;
  comment Print result;
      printstring("The number of values > average is:");
      printint (result)
      end
    else
      printstring ("Error-input list length is not legal");
  end
```

2.6 商务记录计算机化：COBOL

COBOL的故事在某种意义上正好与ALGOL 60的故事相反。尽管它的应用远远超出了其他任何一种程序设计语言，但COBOL对后续语言的设计工作几乎没有什么影响，仅仅对于PL/I语言是一个例外。直至今日，它可能仍然是最为广泛应用的语言，[⊖]只是这一事实很难使用任何方式给以确认。COBOL语言对后来的语言没有影响力的最重要原因，可能是自COBOL语言出现以后，就没有人再试图设计一种新的商务应用语言。这正是由于COBOL语言的功能极好地满足了其应用领域的需要。另一个原因是在过去的20年间，商务计算的大量增长主要出现于小型商务之中，而小型商务很少开发计算机软件。代之，他们所使用的大部分软件是为各种一般商务目的而购买的现成产品。

62

2.6.1 历史背景

COBOL的起源在某种意义上与ALGOL 60有些相似，即，它们都是由一个委员会在相对短的会议期间所设计出来的语言。在1959年，当时的商务计算状况类似于几年以前的科学计算状况，即当Fortran语言还在设计之中时的状态。一种编译式商务应用语言FLOW-MATIC已经于1957年被实现，但这种语言属于制造商UNIVAC，并且它是专门为这个公司的计算机而设计的。另一种是美国空军使用的语言AIMACO，但这种语言只是FLOW-MATIC的一种变种。IBM曾经为商务应用设计了一种程序设计语言，称为“商务翻译”语言（COMmercial TRANslator，COMTRAN），但在当时还并没有被实现。其他的一些语言设计项目也只是还列于计划之中。

⊖ 20世纪90年代的后期，在一个有关Y2K问题的研究中，人们估计，单是在曼哈顿22平方英里的范围之内，大约一共有8亿行的COBOL程序在使用之中。

2.6.2 FLOW-MATIC

FLOW-MATIC语言的起源值得我们对它作简短讨论，因为这种语言是COBOL语言主要的前辈语言。1953年12月，在 Remington-Rand UNIVAC公司工作的Grace Hopper，写了一份实质上是预言性的提议报告。这份报告建议，“数学程序应该用数学的标记方法来编写，数据处理的程序应该用英文语句来编写”（Wexelblat，1981，p. 16）。但不幸的是，在1953年，非编程人员不能相信可以让计算机理解英文。直到1955年，另一份相似的提议才有了获得UNIVAC公司管理层资助的一些希望，就是到这个时候，还是需要依靠一个样机系统进行最后的说服。当时这个说服过程包括了编译与运行一个小程序，首先是使用英文关键字，然后再使用法文关键字，最后又使用德文关键字。这个演示对UNIVAC的管理层印象深刻，是使得他们最终接受Hopper提议的主要因素。

2.6.3 COBOL的设计过程

由美国国防部资助，以商务应用的通用语言为主题的第一次正式会议，于1959年5月28至29日在五角大楼举行（正好是在ALGOL语言的苏黎世会议一年之后）。会议一致认为通用商务语言，当时被命名为CBL（Common Business Language），应该具有下面这些一般特性。这种语言应该尽可能多地使用英语。尽管有个别人争辩，要采用更加数学化的标记方法，但大部分人还是赞成应该尽可能多地使用英语。这种语言必须要易于使用，甚至应该不惜以牺牲一些语言功能为代价，以便加宽可以运用计算机程序的群众基础。这种语言除了容易使用之外，当时人们还相信，一种使用英语的语言会让管理人员都能够阅读程序。最后，在语言的设计上，不应 63
该过分受限于语言实现问题。

在这个会议上最为关切的问题之一是，应该尽快采取行动来创建这种通用语言，因为许多创建新商务语言的准备工作都已经完成了。除了当时已有的语言之外，RCA和Sylvania都正在创建他们自己的商务应用语言。很显然，创建这种通用语言的过程拖得越久，这种语言将更难成为广泛应用的语言。基于此，会议确定应该尽快地研究当时已有的语言。针对这项任务，成立了短期委员会（Short Range Committee）。

早期人们决定，将语言的语句分为两个种类，即数据描述类和可执行操作类，并将分属于这两个种类的语句分别放置于程序的不同部位。短期委员会的一个重要议题是关于是否包括下标。委员会的许多成员认为，下标对于从事数据处理的人员太过复杂，因为这些人不习惯处理数学标记。类似的争论也围绕着是否应该在语言中包括算术表达式。短期委员会的最后报告于1959年12月完成，这个报告中描述了后来命名为COBOL 60的语言。

COBOL 60 的语言说明于1960年4月由政府印刷办公室（the Government Printing Office）出版（Department of Defense，1960），这个版本被称为“初期版本”。它的两个修订版本分别于1961年和1962年出版（Department of Defense，1961，1962）。COBOL语言于1968年由美国国家标准协会（ANSI）实现了标准化。而后来的COBOL 60语言的三个版本分别于1974年，1985年和2002年由美国国家标准协会实现标准化。这种语言一直持续发展到今天。

2.6.4 评估

COBOL语言初创了许多崭新的概念，许多概念后来出现在其他语言之中。例如，COBOL 60的DEPEND动词是第一种宏指令的高级语言结构。更为重要的是，最早出现于Plankalkül语言中的层次数据结构首次被实现于COBOL语言之中。在这之后所设计的大多数命令式语言中，都

包含了这种层次数据结构。COBOL也是允许名称真正具有含义的第一种语言，因为它允许长名字（长达30个字符）以及连字符（-）。

总体来说，数据分区是COBOL语言设计中的优势，然而它的过程分区则相对较弱。每一个变量都在数据分区中被详细地定义，其中包括小数的位数以及隐含的小数点位置。COBOL语言对文件记录的描述也同样详细，例如它甚至定义了由打印机输出的线条，这使得COBOL成为理想的财务报告打印语言。也许COBOL语言早期的过程分区中最重大的弱点在于它不具有函数。
64 早于1974年标准化之前的COBOL版本也不允许带参数的子程序。

我们对COBOL语言的最后评价：它是第一种由美国国防部（DoD）规定要求使用的程序设计语言。因为COBOL语言不是专门为美国国防部而设计的，所以在它的初期并发之后，国防部的这个规定随之而来。如果没有这种规定，COBOL语言尽管具有优点，也许还是不会生存下来。COBOL语言早期编译器的性能很差，这使得它的使用代价实在太高。当然，人们最终改进了编译器的设计，加之后来计算机的速度也更快更便宜，并且有了大得多的存储空间。所有这些因素加在一起，使得COBOL语言无论是在国防部之内还是之外，都取得了巨大的成功。COBOL语言的出现，导致了商务处理的电子自动化，无论从哪方面，这都毫无疑问地是一项意义重大的改革。

下面是COBOL程序的一个例子。这个程序读入一个名为BAL-FWD-FILE的文件，这个文件包含多种货物的库存信息，每种货物的记录包括当前存货的数量（BAL-ON-HAND），以及这种货物的临界记录点（BAL-REORDER-POINT）。当现存货物的数量降低到这个临界记录点时，就必须往订这种货物。这个程序产生一个必须再次订货的货物清单，这个货物清单的文件名为REORDER-LISTING。

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PRODUCE-REORDER-LISTING.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. DEC-VAX.
OBJECT-COMPUTER. DEC-VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT BAL-FWD-FILE    ASSIGN TO READER.
    SELECT REORDER-LISTING  ASSIGN TO LOCAL-PRINTER.

DATA DIVISION.
FILE SECTION.
FD  BAL-FWD-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS.

01  BAL-FWD-CARD.
    02 BAL-ITEM-NO          PICTURE IS 9(5).
    02 BAL-ITEM-DESC        PICTURE IS X(20).
    02 FILLER               PICTURE IS X(5).
    02 BAL-UNIT-PRICE       PICTURE IS 999V99.
    02 BAL-REORDER-POINT    PICTURE IS 9(5).
65  02 BAL-ON-HAND          PICTURE IS 9(5).
    02 BAL-ON-ORDER         PICTURE IS 9(5).
    02 FILLER               PICTURE IS X(30).

FD  REORDER-LISTING
    LABEL RECORDS ARE STANDARD
```

```
    RECORD CONTAINS 132 CHARACTERS.

01  REORDER-LINE.
    02 RL-ITEM-NO          PICTURE IS Z(5).
    02 FILLER              PICTURE IS X(5).
    02 RL-ITEM-DESC        PICTURE IS X(20).
    02 FILLER              PICTURE IS X(5).
    02 RL-UNIT-PRICE       PICTURE IS ZZZ.99.
    02 FILLER              PICTURE IS X(5).
    02 RL-AVAILABLE-STOCK  PICTURE IS Z(5).
    02 FILLER              PICTURE IS X(5).
    02 RL-REORDER-POINT    PICTURE IS Z(5).
    02 FILLER              PICTURE IS X(71).

WORKING-STORAGE SECTION.
01  SWITCHES.
    02 CARD-EOF-SWITCH     PICTURE IS X.
01  WORK-FIELDS.
    02 AVAILABLE-STOCK     PICTURE IS 9(5).

PROCEDURE DIVISION.
000-PRODUCE-REORDER-LISTING.
    OPEN INPUT BAL-FWD-FILE.
    OPEN OUTPUT REORDER-LISTING.
    MOVE "N" TO CARD-EOF-SWITCH.
    PERFORM 100-PRODUCE-REORDER-LINE
        UNTIL CARD-EOF-SWITCH IS EQUAL TO "Y".
    CLOSE BAL-FWD-FILE.
    CLOSE REORDER-LISTING.
    STOP RUN.

100-PRODUCE-REORDER-LINE.
    PERFORM 110-READ-INVENTORY-RECORD.
    IF CARD-EOF-SWITCH IS NOT EQUAL TO "Y"
        PERFORM 120-CALCULATE-AVAILABLE-STOCK
        IF AVAILABLE-STOCK IS LESS THAN BAL-REORDER-POINT
            PERFORM 130-PRINT-REORDER-LINE.

110-READ-INVENTORY-RECORD.
    READ BAL-FWD-FILE RECORD
        AT END
            MOVE "Y" TO CARD-EOF-SWITCH.

120-CALCULATE-AVAILABLE-STOCK.
    ADD BAL-ON-HAND BAL-ON-ORDER
        GIVING AVAILABLE-STOCK.

130-PRINT-REORDER-LINE.
    MOVE SPACE              TO REORDER-LINE.
    MOVE BAL-ITEM-NO        TO RL-ITEM-NO.
    MOVE BAL-ITEM-DESC      TO RL-ITEM-DESC.
    MOVE BAL-UNIT-PRICE     TO RL-UNIT-PRICE.
    MOVE AVAILABLE-STOCK    TO RL-AVAILABLE-STOCK.
    MOVE BAL-REORDER-POINT TO RL-REORDER-POINT.
    WRITE REORDER-LINE.
```

2.7 分时操作的开始：BASIC

BASIC语言（Mather and Waite，1971）是另一种广泛使用的语言，但是却没有得到应有的尊重。像COBOL语言一样，它在很大程度上被计算机科学家所忽略。也像COBOL语言一样，它的早期版本并不完美，且仅仅包括了一组很有限的控制语句。

20世纪70年代后期至80年代初期，BASIC语言在微型计算机上非常流行。这直接源于BASIC语言的两个主要特征：便于初学者掌握，尤其是那些非科学研究人员，而且它的小型方言可以被实现在只具有很小存储空间的计算机上。当微型计算机的功能得到增强并实现了其他各种语言之后，BASIC语言的使用几乎消失。但随着Visual Basic语言（Microsoft，1991）于20世纪90年代初期的出现，BASIC语言的应用又复苏了。

2.7.1 设计过程

BASIC（Beginner's All-purpose Symbolic Instruction Code，意为初学者万用符号指令代码）语言是由John Kemeny与Thomas Kurtz两位数学家在新罕布什尔的Dartmouth学院（即现在的Dartmouth大学）设计的。Kemeny和Kurtz曾经于20世纪60年代初期积极参与了为Fortran及ALGOL 60的多种方言创建编译器的工作。当年他们学院的理科学生能够在学习期间完全没有困难地学习并且使用这些语言。然而Dartmouth主要是一所文科学院，其中的理工科学生人数大约只占学生总数的25%。在1963年的春天，他们决定特别为文科学生设计一种新的语言。这种新语言将使用计算机终端访问计算机。这个系统的目标为：

1. 它必须让非理科学生容易学习与使用。
2. 它必须让人愉快且友好。
3. 它必须方便学生的家庭作业。
4. 它必须允许自由的私人的访问。
5. 它必须视用户的时间比计算机时间更为重要。

67

最后的这一个目标的确是一种革命性观念。至少部分是缘于人们相信计算机的价格将随着时间变得越来越便宜。事实上的确是如此。

第二、第三和第四个目标相结合，导致了BASIC语言的分时操作。在20世纪60年代的初期，只能通过同时允许多个用户分别使用终端访问计算机，这些目标才能够实现。

在1963年的夏天，Kemeny开始为BASIC语言的第一个版本建造编译器。他使用远程终端访问一台GE225型计算机。设计和编写BASIC的操作系统的工作始于1963年的秋天。1964年5月1日早上4点，使用分时操作的BASIC语言的第一个程序被键入终端并且开始运行。到6月份，系统上终端机的数目增长到11台，而到秋天之前又增加到了20台。

2.7.2 语言概述

BASIC语言的最初版本非常小，而且十分奇怪地，它不是交互式的，即不具有从终端输入数据的方法。从键入程序，编译程序，然后到运行程序，所有的这一切都是以一种所谓"批处理"方式（batch-oriented）来进行。这个最初的版本只有14种不同的语句类型，并且仅仅具有一种浮点数据类型。因为当时人们相信，很少会有用户需要区别整数与浮点数这两种不同的数据类型，BASIC语言将这些类型笼统地称为"数字"，总而言之，它是一种相当容易学习但局限性很大的语言。

2.7.3 评估

早期BASIC最重要的方面是，它是第一种使用远程终端访问计算机的广为应用的语

言。[6] 当时终端机才刚刚开始出现，而在此之前的大多数程序，都是经过穿孔卡片或者纸带输入计算机。

BASIC语言中的大部分设计都来自于Fortran语言，也受到ALGOL 60语言语法的些微影响。后来，BASIC语言的各个方面都在发展，却很少或者根本就没有这种语言标准化方面的工作。美国国家标准协会发行了“最基本BASIC语言的标准”（Minimal BASIC standard）（ANSI，1978b），仅仅代表了BASIC语言很少的最基本特性，而在事实上，早期BASIC版本与这个基本标准非常相似。

令人吃惊的是，数字设备公司（Digital Equipment Corporation）于20世纪70年代中期使用BASIC语言的一种名为BASIC-PLUS的加强版本，编写了用于PDP-11型微型计算机的最大操作系统RSTS中的重要部分。

BASIC语言在许多方面遭到了批评，其一是因为用它编写的程序结构很差。根据第1章讨论的评估标准，特别是可读性和可靠性，BASIC语言的确非常糟糕。十分明显的是，BASIC语言的早期版本不是设计用来（也不应该）编写任何大型的重要程序。后来的版本才比较适合这样的工作。 68

20世纪90年代Visual BASIC（VB）的出现复活了BASIC语言。VB在各种领域广泛使用，因为它能够为构建图形用户接口（GUI）提供一种简单的方式，也由此被命名为Visual BASIC。Visual Basic.NET（简称为VB.NET）是微软公司的.NET系列语言中的一种。尽管这种语言与Visual Basic语言非常不同，但很可能就在今后的几年中，Visual Basic.NET将替代较老的 Visual Basic（简称为VB语言）。VB与VB.NET语言之间的最大差别可能在于VB.NET语言全面支持面向对象的程序设计。VB语言原来的用户很可能不会学习VB.NET语言，而是转移到另一种语言如C#（参见第2.19小节），尤其是因为所有的.NET语言都可以访问到VB语言的图形用户界面（GUI）。

下面是BASIC程序的一个例子：

```
REM  BASIC Example Program
REM  Input:  An integer, listlen, where listlen is less
REM          than 100, followed by listlen-integer values
REM  Output: The number of input values that are greater
REM          than the average of all input values
  DIM intlist(99)
  result = 0
  sum = 0
  INPUT listlen
  IF listlen > 0 AND listlen < 100 THEN
REM  Read input into an array and compute the sum
    FOR counter = 1 TO listlen
      INPUT intlist(counter)
      sum = sum + intlist(counter)
    NEXT counter
REM  Compute the average
    average = sum / listlen
REM  Count the number of input values that are > average
    FOR counter = 1 TO listlen
      IF intlist(counter) > average
        THEN result = result + 1
    NEXT counter
REM  Print the result
    PRINT "The number of values that are > average is:";
```
69

[6] LISP最为同时终端，但在20世纪60年代早期没有再到广泛使用。

```
            result
    ELSE
      PRINT "Error-input list length is not legal"
    END IF
  END
```

访谈 用户设计与语言设计



ALAN COOPER

Alan Cooper是畅销书*About Face: The Essentials of User Interface Design*的作者，在设计Visual Basic这种宣称最注重用户界面设计的语言中也不乏他的大手笔。对于他而言，这一切都来自于技术人性化的观点。

背景信息

问：你是如何开始现在所从事的这一切的？

答：我曾经从高中退学，后来获取了加州社区大专的程序设计专业学位。我的第一份工作是在位于旧金山的American President Lines公司（美国最老的海洋运输公司之一）作程序员。除了成立几家那几个工作几个月以外，我基本上都是一个"个体户"。

问：你现在的工作是什么？

答：是从事技术人性化工作的Cooper公司的创始人，并在公司任首席设计师。

问：你最喜爱的工作是什么？

答：界面设计方面的咨询工作。

问：你在语言设计以及用户界面设计领域非常有名。在关于设计语言或设计软件或者设计其他任何东西方面，你有什么样的想法？

答：在软件世界里基本上都是一样的，即了解你的用户。

关于早期的视窗（WINDOWS）系统

问：早在20世纪80年代你就开始使用视窗操作系统，并曾经谈到过你被视窗的特点所吸引：它对于图形用户界面的支持，程序间的动态链接，这些使得你创造了让它们配置自身的工具。最终，你是如何改变了视窗的某些部分？

答：微软在视窗系统中包括进实际意义的多任务支持，给我十分深刻的印象。这里包括了动态地址重新分配以及进程间通信。MSDOS.exe是最初几次视窗发行时的壳程序。这是一个很糟糕的程序，我当时相信还可以对这个程序作巨大的改进，我就进行了这项工作。当时我用业余时间开始编写比视窗操作的壳程序更好的一个壳程序。我将这个程序称为"三脚架"（Tripod）。微软公司原来的壳程序MSDOS.exe曾经是视窗初期成功的一个主要绊脚石。而"三脚架"通过更容易使用和配置来全面解决存在的问题。

问：哪个时刻是你灵感来临的时刻？

答：是1987年的后期，当我正与一个公司的顾客面谈时，关于"三脚架"设计的关键性领悟突然跳入了我的脑海。当时这个IS的经理向我解释，他需要为他的各类用户建立并发行一个范围很广的壳解决方案。我意识到这里的谜底是，根本就不存在一个十全十美的壳。每一个用户都要有能自己个性化的、适合自己需要以及技能水平的壳。那刻，我构思了壳设计问题的解决方案：它是壳结构的一个集合；它是一个工具，而每一个用户都能够根据他或她的应用课题以及训练程度的独特组合，使用这个工具来构造符合需求的壳。

问：为什么个性化壳的思想如此吸引人？

答：不是由我来告诉用户什么是理想的壳，他们可以自己设计个性化的理想的壳。使用这种量体裁衣的壳结构，一个程序员可以创造出一个功能强大并且适用广泛，当然也具有某种危险性的壳来；然而，一个IT经理则可以为一个文秘职员专门创建仅包含适合该职员需要的特定应用工具的壳。

问：你是怎么从编写壳程序到与微软公司合作的？

答："三脚架"与"红宝石"是同一个程序。在我与比尔·盖次签署了合同之后，我将样机的名称从"三脚架"改为"红宝石"。然后，我以样机应有的使用方式来运用"红宝石"样机：一个用于构造发布级质量代码的先行模型，这就是我当时所完成的工作。微软公司取得"红宝石"的发布版本，并将其与QuickBASIC揉合起来就产生了VB。这中间所有的原始革新都包含在"三脚架"/"红宝石"里面。

"红宝石"作为VISUAL BASIC的孵化器

问：让我们重新谈一谈你对早期的视窗系统及其动态链接库（DLL）特性的兴趣。

答：DLL并不是独立的，它只是操作系统中的一项工具。这个工具允许程序人员构造代码对象，并且能够在运行时实现链接，而不是在编译时链接。正是这种特性允许我创造了VB语言中可动态扩展的部分，这一部分就连作为第三者的实方也可以施加控制。

"红宝石"产品促成了软件设计中的许多重大进步，其中两项更是空前成功。如我曾经提到的，视窗系统的动态链接功能当时时刻在激励着我，但是具有一些工具和知道如何运用它们却是完全不同的两件事情。有了"红宝石"，我最终找到了两种动态链接的实际使用，初期的"红宝石"程序就包括了这两种使用。首先，这种语言是可安装的，并且能够被动态地扩展，第二，新发明的模块能够动态地加入其中。

问：你的"红宝石"是不是最早具有动态链接库并且被链接到一个可视化前端的？

答：就我所知，是的。

问：请用一个简单例子说明，这些能够让一个程序员对他或她的程序做些什么？

答：从第三方购买一个控件，例如一个网格控件，将这个控件安装到他或她的计算机上，并使得这个网格控件看上去像是该语言的一个组成部分，包括可视化程序设计前端。

问：为什么人们称你为"Visual Basic之父"？

答："红宝石"伴随着一种小型语言，这种语言适合执行壳程序所需要的十来条或者极为简单的指令。然而，这种语言被作为DLL链来实现，而任意数目的DLL都可以在运行时装载。计算机内部的语法分析器可以识别一个动词，并将这个动词沿着DLL的链传递，直到链上某个环节表示知道如何处理为止。如果这个动词通过了整个DLL链，则存在着一个语法错误。从我与微软公司的早期讨论，我们双方都赞同扩展这种语言的主意，也可能，甚至会用一种"真实"的语言来完全替代它。C是经常被提及的候选语言，然而微软最终利用这种动态接口拆卸下我们的小型壳语言，并用QuickBasic语言整体取代了这种壳语言。这种语言与可视化前端的新型结合是静态与固定的。而且，尽管正是原始的动态接口才使得这种结合成为可能，但原始部分在这个过程中被遗弃。

关于新思想的最后几句话

问：在程序设计以及程序设计工具领域里，也包括语言与环境，哪一个项目让你最感兴趣？

答：我感兴趣的是创造程序设计工具，这些工具主要设计用来帮助用户而不是程序人员。

问：让你谨记在心的重要法则，著名语录或者设计思想是什么？

答：桥梁是由钢铁工人建造的，而不是由工程师建造的。

同样地，软件是由程序人员编写的，而不是由工程师编写的。

71

2.8 用途广泛的语言：PL/I

PL/I 代表了第一次大规模的尝试，人们尝试设计一种适用于应用范围广泛的语言。所有在此之前以及大多数在此之后设计的语言，都将重心集中于某一特殊的应用领域，例如，科学应用、人工智能或者商务应用。

2.8.1 历史背景

与Fortran语言一样，PL/I是作为IBM的一项产品被开发的。早在20世纪60年代的初期，工

业界中的计算机用户就被分为两个相互独立而又相当不同的阵营。以IBM的观点，从事科学计算的程序人员可以使用IBM的7090大型计算机或者1620小型计算机。正是这一部分人员，极大地使用浮点数据类型和数组。他们使用的主要语言是Fortran，也使用一些汇编语言。他们有着自己的用户团体SHARE，而且他们与工作于商务应用领域的人员极少联系。

对于商务应用，人们使用IBM的7080大型计算机或者1401小型计算机。他们需要十进制及字符串数据类型，也需要精细和高效的输入及输出设施。他们使用COBOL语言。尽管在1963年的初期，关于PL/I语言的故事才刚刚开始，而那时从汇编语言到COBOL的转换还远没有完成，他们却已经使用了COBOL。这种类型的用户也有自己的用户团体GUIDE，并且他们也很少与科学计算类型的用户联系。

在1963年初，IBM的规划人员感觉到这种情况在开始变化，这两个互相远离的计算机用户团体正在向彼此靠近的方向移动。IBM的人员认为这必定会产生问题。当时，科学家们已经开始采集大型文件数据来进行处理，这些数据需要更复杂更高效的输入输出设施。而商务应用的人们开始应用回归分析来建立信息管理系统，其中需要使用浮点数据和数组。这开始显示，很快人们会需要两台不同的计算机以及不同的技术人员来进行计算装载工作，以便支持两种非常不同的程序设计语言。[8]

这种感觉十分自然地导致人们想要设计一种通用计算机的想法，这种计算机应兼有浮点数运算以及十进制算术运算的能力，因而能够同时实施科学应用以及商务应用。IBM System/360系列计算机的原理就由此而诞生。随之而来的想法便是，需要一种能够兼用于科学以及商务应用的程序设计语言。作为锦上添花，这种语言还增添了系统程序设计和表处理的语言特性。因此，
72 这样的新语言被计划用来替代Fortran、COBOL、LISP，以及汇编语言的系统应用。

2.8.2 设计过程

语言的设计工作开始于1963年10月，当时IBM和SHARE成立了"SHARE Fortran项目的高级语言开发委员会"(the Advanced Language Development Committee of the SHARE Fortran Project)。这个新委员会的成员很快就头，并成立了一个被称为3×3委员会的子委员会，之所以这样命名，是因为它的三个成员来自于IBM，而另外三个成员来自于SHARE。3×3委员会每两一个星期开会三天或四天，来设计新的语言。

就像COBOL的短期委员会一样，他们要在相当短的时间内完成初步的语言设计。显然在20世纪60年代初期盛行的观念是，不论这种语言设计的工作量如何，都可以在三个月之内完成。PL/I语言的第一个版本（当时被命名为Fortran VI）原定于委员会成立之后不到三个月的时间即12月前完成。委员会成功地申请了延期，将完成时间推移到了1月份，进而又再次推移到了1964年2月的下旬。

在最初的设计构想中，新语言将是Fortran IV的一种扩展，以便维持语言的兼容性。但很快，这个目标连同名字Fortran VI一起被弃用。到1965年之前，这种语言一直被称为NPL，源于英文"新程序设计语言"(New Programming Language)的首字母组合。NPL语言的第一次发布报告于1964年3月在SHARE会议上公布。接着在4月份，发表了NPL语言的比较完整的描述，而它的能够被实际实现的版本于1964年12月由英国的IBM Hursley实验室的编译器工作组发表(IBM，1964)。该工作组专门进行这种语言的实现工作。NPL语言的名字在1965年被改为PL/I，以免与英国国家物理实验室（National Physical Laboratory，NPL）的名字相混淆。假如PL/I的编译器

[8] 同时，大型计算机设备需要专门的硬件和专门的系统软件维护人员。

是仁英国之外开发的，它的名字可能会保持为NPL。

2.8.3 语言概述

描述PL/I语言最为贴切的一句话也许是，它包括了下列这些语言中当时被认为最优秀的部分：ALGOL 60的“递归与块结构”，Fortran IV的“通过全局数据来分开编译与通信”，以及COBOL 60的“数据结构、输入/输出以及生成报告的设施”，还包括一系列相当多的新结构，并将所有这些部分结合在一起。因为PL/I现在几乎是已经死去的语言，所以我们不会哪怕只以简略的方式来讨论这种语言的所有特性，或者它的最具争议的结构。我们只是简要地概括这种语言对于程序设计语言知识库的一些贡献。

PL/I是第一种具有下列设施的程序设计语言：

- 允许程序产生并发执行的子程序。这虽然是一个好主意，但是这种功能在PL/I中的开发却很差。
- 可以发现并处理23种不同种类的异常或运行时错误。
- 允许子程序递归，但为了使子非递归子程序高效率地链接，也可以停止递归功能。
- 指针被作为一种数据类型。
- 可以单独引用数组的横向部分。例如，可以将矩阵的第三行作为一个矢量来引用。

2.8.4 评估

任何对于PL/I语言的评估，必须从认识这项设计工作的动荡雄心开始。现在来看，当时认为这么多的结构可以成功地结合起来显然是过于天真。但在回顾这段历史时，我们必须认识到，当时人们还没有多少语言设计的经验。总而言之，PL/I语言设计的前提，是要包括进任何有用的、并可以实现的语言结构。但没有充分意识到，当将这么多的语言特性绑在一起时，将会出现什么样的行为。Edsger Dijkstra在他的图灵奖讲座（Turing Award Lecture）（Dijkstra，1972）中，对PL/I语言的复杂性做出了最为激烈的批评：“我绝对无法预见，当程序设计语言——请注意，这是我们的基本工具——已经超出了我们的智力控制范围时，我们如何能够仍然幸福地，将不断增长的程序置于我们的智力掌握之下。”

除了由于过于庞大的规模所引起的语言复杂性问题之外，PL/I还有许多现在被认为是设计得很差的结构，其中有指针结构、异常处理与并发性；虽然我们必须指出，这里所列举的每一种结构从来没有在前面的语言中出现过。

就语言的应用方面而言，PL/I至少应该被看作是部分成功的。它于20世纪70年代被大量地运用于商务和科学应用中。这期间，它的几种子语言也被作为教学语言广泛运用，如PL/C语言（Cornell，1977）和PL/CS语言（Conway and Constable，1976）。

下面是PL/I程序的一个例子：

```
/* PL/I PROGRAM EXAMPLE
 INPUT:  AN INTEGER, LISTLEN, WHERE LISTLEN IS LESS THAN
         100, FOLLOWED BY LISTLEN-INTEGER VALUES
 OUTPUT: THE NUMBER OF INPUT VALUES THAT ARE GREATER THAN
         THE AVERAGE OF ALL INPUT VALUES    */
PLIEX: PROCEDURE OPTIONS (MAIN);
  DECLARE INTLIST (1:99) FIXED,
  DECLARE (LISTLEN, COUNTER, SUM, AVERAGE, RESULT) FIXED;
  SUM = 0;
```

```
    RESULT = 0;
    GET LIST (LISTLEN);
    IF (LISTLEN > 0) & (LISTLEN < 101) THEN
      DO;
/* READ INPUT DATA INTO AN ARRAY AND COMPUTE THE SUM */
      DO COUNTER = 1 TO LISTLEN;
        GET LIST (INTLIST (COUNTER));
        SUM = SUM + INTLIST (COUNTER);
      END;
/* COMPUTE THE AVERAGE */
      AVERAGE = SUM / LISTLEN;
/* COUNT THE NUMBER OF VALUES THAT ARE > AVERAGE */
      DO COUNTER = 1 TO LISTLEN;
        IF INTLIST (COUNTER) > AVERAGE THEN
          RESULT = RESULT + 1;
      END;
/* PRINT RESULT */
      PUT SKIP LIST ('THE NUMBER OF VALUES > AVERAGE IS:');
      PUT LIST (RESULT);
      END;
    ELSE
      PUT SKIP LIST ('ERROR-INPUT LIST LENGTH IS ILLEGAL');
    END PLIEX;
```

2.9 两种早期的动态语言：APL和SNOBOL

本节的结构将不同于本章中的其他各小节，因为在这里讨论的语言与其他的语言非常不同。APL与SNOBOL都没有基于任何已经存在的语言，而且这两种语言对以后的主流语言都没有多少影响。[5] 我们将在本书的后面介绍APL的一些有趣的语言特性。

在语言的外观和语言的目的这两方面，APL与SNOBOL非常不同。但是它们却共同享有两种基本特征：动态类型化和动态存储空间分配。这两种语言中的变量基本上不具有类型。当赋值给一个变量时，该变量就获得一个类型，这时它即承接所赋数值的类型。仅仅当一个变量被赋值时，才将存储空间分配给这个变量。因为在此之前无法知道这个变量所需要的存储空间大小。

2.9.1 APL的起源与特征

APL（Brown et al., 1988）是由IBM的Kenneth E. Iverson于1960年的前后设计的。最初的计划并不是将它设计成为一种可实现的程序设计语言，而是想要设计成一种用来描述计算机体系结构的工具。关于APL的首次描述，发表在一本名为*A Programming Language*（Iverson, 1962）的书中。它正是因为这本书而得名。APL的第一次实现是于20世纪60年代中期在IBM开发的。

APL具有大量功能强大的操作符，这给语言的实现人员带来了一个问题。使用APL的第一种方法是通过IBM的打印终端。这些打印终端具有一些特殊的印刷球，来提供这种语言所要求的特殊字符集合。APL语言具有这么多操作符的一个原因，是它允许像操纵一个单位一样来操纵数组。例如，可以通过一个操作符来完成任意矩阵的转置运算。尽管这些庞大的操作符系列提供了极高的语言表达力，但APL的程序难于阅读。这使得有些人认为，最好是用APL来进行"一次性"的程序设计。尽管使用它编写程序很快，但由于这些程序十分难以维护，因而使用之

[5] 然而，它们对非主流语言有影响（J语言是基于APL，ICON语言是关于SNOBOL，AWK语言部分基于SNOBOL）。

后应该丢弃。

APL已经生存了40年，尽管它的应用不广泛，但今天仍然还在使用之中。此外，自APL产生以来，它没有经过大的修改。

2.9.2 SNOBOL的起源与特征

SNOBOL（发音为"snowball"）(Griswold et al., 1971）语言是于20世纪60年代初期由贝尔实验室的三个工作人员所设计的。他们是D.J.Farber, R.E.Griswold和F.P. Polensky（Farber et al., 1964）。它是专门为文本处理而设计的语言。SNOBOL语言的核心是一系列用于字符串模式匹配的功能强大的操作符。SNOBOL的早期应用之一是编写文本编辑器。因为SNOBOL的动态特征，使得它比其他一些语言的速度要慢。现在已经不再使用这种类型的程序。然而SNOBOL仍然是具有生命力且得到支持的语言，它被用于许多不同的应用领域，完成各种各样的文本处理任务。

2.10 数据抽象的开始：SIMULA 67

虽然SIMULA 67从未得到广泛应用，并对当时的计算界以及程序人员影响甚微，但是它所引入的一些概念，却使得它成为一种具有重要历史意义的语言。

2.10.1 设计过程

1962年至1964年间，两个挪威人Kristen Nygaard 和 Ole Johan Dahl，在挪威计算中心（NCC）共同开发了SIMULA I语言。他们的主要兴趣是使用计算机进行模拟，但也进行操作系统方面的研究。SIMULA I完全是为系统模拟而设计的，于1964年下半年首先在一台 UNIVAC 1107 计算机上实现。

就在 SIMULA I 语言实现完成之后，Nygaard和Dahl很快开始了语言的扩展工作：增加一些 [76]
新的语言特性以及修改一些已有的语言结构，以便SIMULA I能够适用于一般应用。这项工作的结果导致了SIMULA 67语言的诞生。SIMULA 67的设计于1967年3月首次被公开发表（Dahl and Nygaard，1967）。尽管我们所感兴趣的一些SIMULA 67语言特性也出现于SIMULA I之中，但我们将仅讨论SIMULA 67语言。

2.10.2 语言概述

SIMULA 67是ALGOL 60语言的一种扩展，它采纳了ALGOL 60中的两种结构：块结构与控制语句结构。ALGOL 60（以及当时的其他各种语言）用于模拟时的主要缺陷是子程序的设计。模拟要求子程序能够从以前停止的位置重新开始运行。具有这种控制结构的子程序都被称为协同程序，因为在它们的调用程序与被调用子程序之间存在一种彼此平等的关系，而不是通常在命令式语言中所有的主从层次关系。

为了在 SIMULA 67中提供对协同程序的支持，人们开发了类（class）结构。这是一项十分重要的发展，因为我们的数据抽象概念就是由此开始的。类的基本概念是将数据结构以及操纵这种数据结构的程序包装在一起。此外，类的定义只是一个用于数据结构的模板，它与类的实例是不同的，因此程序能够产生并且使用某一个类中任意数目的实例。类的实例能够包括局部数据，类的实例也能够包括实例产生时执行的代码，这些代码可以将一些类实例的数据结构初始化。

对于类以及类实例的比较完整的讨论，将在第11章中给出。有趣的是直到1972年，当 Hoare

(1972) 认识到这种数据抽象与类的关联时，才将数据抽象的重要概念发展和归结到类的结构之中。

2.11 正交性语言的设计：ALGOL 68

ALGOL 68是语言设计中几种新思想的起源，其中的一些后来被其他语言采纳。这就是在这本书中包括 ALGOL 68 的原因，尽管它从来没有在欧洲或美国得到广泛地应用。

2.11.1 设计过程

当ALGOL语言的修改报告于1962年发表时，ALGOL语言系列的开发工作并没有结束，又过了6年之后，ALGOL语言的下一个设计版本被发表。最后产生的语言ALGOL 68 (van Wijngaarden et al., 1969) 与先前的 ALGOL 语言有着显著不同。

77

ALGOL 68最具意义的革新是它的主要设计准则之一：正交性。回顾我们在第1章中关于语言正交性的讨论。正交性的运用导致了ALGOL 68语言的一些新颖特性，我们将在下面的小节中描述其中的一个特性。

2.11.2 语言概述

ALGOL 68中正交性的一个重要成果是它所包含的用户定义的数据类型。早期的语言，如 Fortran，只包含了少量的基本数据结构，而PL/I语言包括了大量的数据结构，这使得学习与实现这种语言极为困难，并且它显然不能提供一种适用于所有应用问题的数据结构。

ALGOL 68中数据结构的设计方式是，在语言中只提供少量的基本类型和基本结构，而允许用户将这些基本类型与基本结构相结合来产生大量的不同结构。这种在语言中提供用户定义数据类型的方法，在相当大的程度上几乎被所有自此以后所设计的主要命令式语言所继承。用户定义的数据类型很有价值，它们可以让用户设计非常接近某些特定问题的合适的数据抽象。关于数据类型的全面讨论将放在第6章中。

另一个在数据结构领域里的第一是ALGOL 68 首次引入了动态数组，我们在第5章称这种动态数组为隐式堆动态。动态数组是指该数组的声明不规定其下标范围的界限。对一个动态数组赋值才导致为该数组分配所需要的存储空间。在 ALGOL 68 中，动态数组称为 flex 数组。

2.11.3 评估

ALGOL 68包括了在此之前从未被程序设计语言使用过的大量特性。它对于正交性的运用毫无疑问是革命性的，尽管其些人可能认为过分了。

然而 ALGOL 68重复了ALGOL 60的一个错误，这也是这种语言仅获得有限接受的一个重要因素。这就是，用来描述这种语言的是一种优雅与简洁但却鲜为人知的元语言。在人们能够阅读描述 ALGOL 68的文件 (van Wijngaarden et al., 1969) 之前，必须学习这种被称为 van Wijngaarden文法的新的元语言。更为糟糕的是，ALGOL 68的设计人员还生造了一系列的术语用来解释它的文法和语言。例如，将“关键字”(keyword) 称为“指示符”(indicant)，将“子字符串提取”(substring extraction) 称为“修剪”(trimming)，将“过程执行的处理”(process of procedure execution) 称为“非过程化强制”(coercion of deproceduring)，而这种强制还可以是“柔和的”(meek)，“固定的”(firm) 或者别的什么。

人们会很自然地将ALGOL 68的设计与PL/I的设计进行对比。ALGOL 68靠运用正交性原理取得了可写性：只用极少量的基本概念加上无限制地使用几种结合机制，而 PL/I 靠包括进大量

固定的结构来达到可写性，ALGOL 68延续了ALGOL 60语言优雅的简单性，而PL/I 语言仅仅是将几种语言的特性简单地堆积在一起来达到它的目标。当然我们也必须记住，PL/I 的目的是要为广泛类型的应用问题提供一种统一工具；与此正好相反，ALGOL 68 所针对的只是一种类型的问题：科学应用。

78

PL/I获得了远远高于ALGOL 68的接纳程度，其中很大一部分的原因归结于IBM公司的促进工作，以及人们对于ALGOL 68语言在理解与实现上所存在的问题。这两种语言的实现都是一个困难问题，但PL/I获得了IBM公司的资源来促进它的编译器，而ALGOL 68则没有得到这样的赞助者。

2.12 早期ALGOL系列语言的后代产品

所有命令式语言，包括如C++和Java之类的命令式/面向对象语言，它们的一些设计成果都应该归功于ALGOL 60以及/或者ALGOL 68语言。本节将讨论这些语言早期的一些后代语言。

2.12.1 为简单性而设计的语言：Pascal

2.12.1.1 历史背景

Niklaus Wirth（Wirth发音为"Virt"）是国际信息处理联合会（International Federation of Information Processing，IFIP）2.1工作组的一个成员，这个工作组成立于20世纪60 年代的中期，其目的是为了继续 ALGOL 语言的开发，Wirth与C.A.R. Hoare于1965年8月为继续这方面工作，向工作组提交了略显保守的、意欲增加和修改ALGOL 60的提案（Wirth和Hoare，1966）。然而工作组内的多数成员因为这种改进在ALGOL 60原有的基础上进步太小而否决了这个提案，代之以开发了一个更为复杂的语言版本提案，并最后演变成了ALGOL 68。Wirth连同其他几个工作组的成员，基于ALGOL 68语言的本身，以及用来描述ALGOL 68的元语言的复杂性，认为不应该公布关于ALGOL 68的报告。他们的这种立场后来被证明不无道理，因为计算机界的确发现，ALGOL 68语言的文件以及语言的本身正接受考验。

Wirth和Hoare的ALGOL 60的修订版本被命名为ALGOL-W，ALGOL-W后来在斯坦福大学被实现，并主要被用来作为一种教学工具，而且它只是用于很少的几所大学里。ALGOL-W 语言的主要贡献是参数的按值-结果传递方法，以及多样选择的case语句。按值-结果传递方法是ALGOL 60中按名传递方法的一种替代方法。这两种方法都将在第9章里讨论。case语句则将在第8章里讨论。

Wirth的下一个主要设计工作再一次地基于ALGOL 60语言。这一次是他最成功的工作：Pascal语言。[⊖]早期发表的Pascal语言的定义公布于1971年（Wirth，1971）。这个版本在实现的过程中被稍作修改，并由Wirth进行了描述（Wirth，1973）。一些常常被归于 Pascal 语言的特性事实上来自于更早期的语言。例如，用户定义的数据类型是在 ALGOL 68 中被引入的，case语句来自于ALGOL-W，而Pascal中的记录数据类型与COBOL 和 PL/I 语言中的类似。

79

2.12.1.2 评估

Pascal 语言的最大影响是在程序设计的教学方面。在1970年，大多数的计算机、工程以及科学学科的学生使用 Fortran语言作为程序设计的入门。尽管当时也有些大学使用 PL/I，基于PL/I的其他语言，以及 ALGOL-W，到20世纪70年代的中期，Pascal 已经成为教学用途中最为广泛应用的语言。这种结果也许不能预测，但却十分自然，因为 Pascal 实际上是专门为程序设

⊖ Pascal语言是按名字Blaise Pascal而命名的，Blaise Pascal是法国17世纪的一位哲学家和数学家，他曾经在1642年发明了第一台机械加法机器，还有其他的为一些发明。

计的教学而设计的。直到20世纪90年代的末期，Pascal才不再是大专院校中最普遍使用的程序设计教学语言。

因为Pascal语言被设计为一种教学使用语言，所以缺乏多种类型的应用所必须的许多基本特性。其中最具说服力的一个例子便是，不能够使用Pascal来编写一个将可变长度数组作为参数的子程序。另一个例子是，Pascal缺乏分别编译的能力。这些特性的缺乏，自然导致了许多非标准的方言的产生，如Turbo Pascal。

Pascal在程序设计的教学以及其他应用中得到广泛使用的主要原因，是这种语言所具有的简单性与表达性之间的卓越结合。尽管在Pascal语言中也存在着一些不安全的因素，但它仍然是一种相对安全的语言，尤其是与Fortran及C语言相比。到了20世纪90年代中期，无论是在工业界或者大专院校，Pascal的应用都减少了，这主要是因为Modula-2、Ada和C++语言的崛起，它们都包含有Pascal不具备的特性。

下面是Pascal程序的一个例子：

```
{Pascal Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all input values }
program pasex (input, output);
  type intlisttype = array [1..99] of integer;
  var
    intlist : intlisttype;
    listlen, counter, sum, average, result : integer;
  begin
  result := 0;
  sum := 0;
  readln (listlen);
  if ((listlen > 0) and (listlen < 100)) then
    begin
{ Read input into an array and compute the sum }
    for counter := 1 to listlen do
      begin
      readln (intlist[counter]);
      sum := sum + intlist[counter]
      end;
{ Compute the average }
    average := sum / listlen;
{ Count the number of input values that are > average }
    for counter := 1 to listlen do
      if (intlist[counter] > average) then
        result := result + 1;
{ Print the result }
    writeln ('The number of values > average is:',
             result)
    end { of the then clause of if (( listlen > 0 ... }
  else
    writeln ('Error-input list length is not legal')
end.
```

80

2.12.2 可移植的系统语言：C

与Pascal语言类似，C也几乎没有创新的语言特性，然而它的广泛应用延续了一个相当长的时期。尽管C原来是专门为系统程序设计而设计的语言，但它十分适合于广阔范围内的各种

应用。

2.12.2.1 历史背景

C的前驱语言包括CPL、BCPL、B和ALGOL 68。CPL是于20世纪60年代初期由剑桥大学开发的。BCPL则是于1967年由Martin Richards开发出来的一种简单系统程序设计语言(Richards，1969)。

20世纪60年代末期，由Ken Thompson在贝尔实验室首次进行了C在UNIX操作系统上的工作。C的第一个版本是用汇编语言写成。而在UNIX系统上实现的第一种高级语言是B语言，它是一种基于BCPL的语言，B语言于1970年由Thompson设计并实现。

BCPL与B都不是一种类型化的语言，这在高级语言中是一个异类，虽然这两种语言的层次都要比Java这类语言要低很多。无类型语言意味着其中所有的数据都被认为是机器字，这种情形尽管极为简单，却导致了许多复杂性与安全性问题。例如，如何在表达式中说明是浮点数而不是整数的算术运算问题。在BCPL的一种实现中，是在进行浮点数运算的操作数前面放置一个句号，所有前面没有句号的操作数都被认为是整数。这种表达形式的另一种替代形式是必须为浮点数操作使用不同的符号。

这种问题连同其他的一些问题一起导致了一种基于B语言的新类型化语言的发展。这就是最初被称为NB语言然后被命名为C的语言。这种语言是于1972年由Dennis Ritchie在贝尔实验室设计并实现的(Kernighan and Ritchie，1978)。C受到了ALGOL 68的影响，有些是经由BCPL而受影响，而另一些则是直接来自于ALGOL 68的影响。这些影响通过它的for语句、它的switch语句、它的赋值操作符以及指针处理而可见一斑。

在最初的十多年中，有关C语言的唯一"标准"是Kernighan和Ritchie所编写的书(Kernighan and Ritchie，1978)。[6] C在这一段时期经历了缓慢的演变，不同的实现人员给C语言增加了不同的特性。1989年，美国国家标准协会(ANSI)产生了C的正式描述版本(ANSI，1989)，其中包括了已经由实现人员结合进来的许多语言特性。这个标准版本于1999年再次被更新(ISO，1999)。这个新版本包含语言的一个重大改变。C的1989年版本在很长的时期内都被称为ANSI C，我们现在应该称它为C89，而将C的1999年版本称为C99。

81

2.12.2.2 评估

C具有足够的控制语句以及数据结构化的设施，这允许它能够被用于各种应用领域。C也具有丰富的操作符，这给语言提供了极高的表达性。

人们对于C语言既喜爱又不喜爱的一个最重要的原因，是它缺乏完整的类型检测。例如在C99以前的版本中，函数参数可以不经过类型检测。喜爱C语言的人珍视它的灵活性，而不喜爱C的人则感觉它太不安全。C语言受欢迎的程度在20世纪80年代急剧增加。其中的一个主要原因是，C语言的编译器是使用广泛的UNIX操作系统的一部分。这种编译器嵌入UNIX系统，为不同类型计算机的程序人员提供了基本免费和性能优异的编译器。

下面是C程序的一个例子：

```
/* C Example Program
 Input:  An integer, listlen, where listlen is less than
         100, followed by listlen-integer values
 Output: The number of input values that are greater than
         the average of all input values */
```

[6] 这种语言有时也被称为"K&R C"。

```
void main (){
  int intlist[99], listlen, counter, sum, average, result;
  sum = 0;
82
  result = 0;
  scanf("%d", &listlen);
  if ((listlen > 0) && (listlen < 100)) {
/* Read input into an array and compute the sum */
    for (counter = 0; counter < listlen; counter++) {
      scanf("%d", &intlist[counter]);
      sum += intlist[counter];
     }
/* Compute the average */
    average = sum / listlen;
/* Count the input values that are > average */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Print result */
    printf("Number of values > average is:%d\n", result);
   }
  else
    printf("Error-input list length is not legal\n");
}
```

2.12.3 一种（或多或少）相关的语言：Perl

我们在本节简略地讨论Perl语言的起源与特征。就理想情况而言，Perl语言肯定不完全适合于这一节——它与ALGOL语言的联系仅仅是通过C，而且即使是通过C语言，也仅仅是在语法以及基本控制语句方面。然而，Perl语言也不适合于这一章中的任何其他小节，尽管它的重要性还不足以让它单独占用一个小节，但它是不应该被忽略的。

2.12.3.1 历史背景

脚本语言在过去的25年间发展了起来。早期的脚本语言被用来在一个执行文件中放置一列被称为脚本（script）的指令，第一种脚本语言命名为sh（意义为shell），开始于一小组指令，这些指令被翻译为对系统中功能性子程序的调用，这些功能包括文件的管理以及简单文件的筛选。在这一组指令的基础之上又增加了变量、控制流语句、函数以及其他的各种功能，结果是形成了一套完整的程序设计语言。这类语言中功能最强大并且最著名的是ksh语言（Bolsky and Korn, 1995），它由David Korn在贝尔实验室开发产生。

另一种脚本语言是awk语言，它由Al Aho, Brian Kernighan以及Peter Weinberger在贝尔实验室开发完成（Aho et al., 1988）。awk语言在开始时是一种报告生成语言，后来成为了一种更为通用的语言。tcl语言是一种可扩展的脚本语言，由John Ousterhout在加利福尼亚大学伯克利分校
85 开发（Ousterhout, 1994）。tcl语言现在已经与一种提供建造GUI方法的tk语言相结合。

由Larry Wall所开发的Perl语言（Wall et al., 2000），其最初形式是sh语言和awk语言的结合。Perl语言自创始以来有了极为重大的发展，现在已经成为一种有强大功能但还有些许原始的程序设计语言，尽管人们仍然通常称Perl语言为脚本语言，但它实际上更类似于一种典型的命令式语言，当为在这种语言被执行之前，它经常被编译成为一种中间语言的形式。此外，这种语言有着所有可能的各种结构，这使得它能够被应用于广泛多样的计算问题。

2.12.3.2 语言的特征性质

Perl语言具有许多有趣的特性，我们仅在本章中提出并在本书的剩余章节里讨论其中少数几个特性。

Perl中的变量是被静态类型化、并且被隐式地声明。它的变量具有三个不同的命名空间，分别由变量名中的第一个字符来标记。所有的数量变量名始于"$"符号，所有的数组名始于"@"符号，以及所有的散列名（下面将简要介绍散列）始于百分号"%"。这种规定使得Perl程序中的变量名比在其他程序设计语言中的变量名可读性更好。

Perl语言还包括了大量的隐性变量。一些隐性变量被用来存储Perl中的参数，例如，特殊形式的换行字母或者用在实现中的字母。隐性变量通常被用作内建函数中的默认参数，以及某些操作符中的默认操作数。这些隐性变量都具有不同（尽管是有点神秘）的名字，形如"$1"和"@_"。隐性变量的名字也类似于用户定义的变量名，同样使用三个名字空间，所以"$1"是一个标量。

Perl数组的两种特征使得它们不同于一般命令式语言中的数组。首先，它们具有动态长度，这意味着它们在执行期间可以根据需要伸长或缩短。第二，数组可以是稀疏的，这意味着数组的元素之间可以存在空隙。这些空隙在储存空间以及在用于数组的循环语句中都并不占有位置，foreach循环将越过空缺的元素进行。

Perl语言包括了关联数组，这种关联数组被称为散列。散列数据结构是通过字符串进行索引，是一些隐性控制的散列表。Perl系统提供散列函数，在需要时增加散列结构的大小。

2.12.3.3 评估

Perl是一种功能强大但又有某种程度的危险性的语言。它的标量类型既可储存字符串又可储存数值两种类型，而通常，数值是以双精度浮点数的形式来储存的。取决于数值所在的环境，有时候数值可能会被强制转换成为字符串，或者是反过来，即字符串会被强制转换为数值。如果一个字符串被用于数值环境中，并且这个字符串不能够被转换成数值，则会赋予0值，而且不会给用户送出任何警告或出错信息，这将导致编译器或运行时系统都不能发现初错误。数组索引不能够被检测，因为没有数组下标的范围。对于空缺元素的引用将返回undef，这在数值环境里被翻译为0值。也而在数组元素的访问中也不会发现错误。 84

Perl语言的最初使用是作为UNIX系统中用于文字文件处理工具的功能语言。无论是在当时还是现在，它一直都被用作UNIX系统的管理工具。万维网出现以后，Perl作为用于万维网的通关接口语言得到了广泛的应用，尽管Perl的使用正在减少，现在Perl更是用于各种应用的一种通用语言，它的应用包括了计算生物学和人工智能。

下面是Perl程序的一个例子：

```
# Perl Example Program
# Input:  An integer, $listlen, where $listlen is less
#         than 100, followed by $listlen-integer values.
# Output: The number of input values that are greater than
#         the average of all input values.
($sum, $result) = (0, 0);
$listlen = <STDIN>;
if ($listlen > 0) && ($listlen < 100)) {
# Read input into an array and compute the sum
  for ($counter = 0; $counter < $listlen; $counter++) {
    $intlist[$counter] = <STDIN>;
  } # end of for (counter ...
# Compute the average
  $average = $sum / $listlen;
# Count the input values that are > average
  foreach $num ($intlist) {
    if ($num > $average) { $result++; }
  } # end of foreach $num ...
```

```
# Print result
  print "Number of values > average is: $result \n";
} #- end of if (($listlen ...
else {
  print "Error--input list length is not legal \n";
}
```

2.13 基于逻辑的程序设计：Prolog

简言之，逻辑程序设计就是运用形式逻辑标记，就计算过程与计算机进行通讯。谓词演算是目前逻辑程序设计语言中所使用的标记方法。

逻辑程序设计语言中的程序设计是非过程的。这类语言的程序并不严格说明怎样计算以求得结果，相反，它只是描述这种必要的形式和（或）结果的特征。为了让程序设计语言具有这样一种功能，需要有精确的方式给计算机提供相关的信息以及求取所需结果的推理方式。谓词
85 演算提供了人机通讯的基本逻辑形式，以及命名为归结（resolution）的证明方法；这种由Robinson（Robinson，1965）首先开发的证明方法提供了这种推理技术。

2.13.1 设计过程

在20世纪70年代的初期，Aix-Marseille 大学人工智能小组的 Alain Colmerauer 和 Phillippe Roussel，与爱丁堡大学人工智能系的Robert Kowalski 一起，进行了关于 Prolog 语言的基本设计。Prolog的主要组成部分是一种说明谓词演算命题的方法，以及一种归结受限形式的实现。谓词演算与归结都将在第16章里描述。第一个Prolog语言解释器于1972年在 Marseille 开发完成。这个实现了的Prolog版本由Roussel给予描述（Roussel，1975）。Prolog 的名称来自“逻辑程序设计”（programming logic）中两个英文单词的各自前二个字母。

2.13.2 语言概述

Prolog程序由一系列语句组成。Prolog语言只具有很少的语句类型，但这些语句是复杂的。

Prolog语言最通常的用法是作为一种智能数据库，这种应用为我们讨论Prolog 语言提供了一个简单的框架。

一个Prolog程序数据库包括两种类型的语句：事实与规则。事实语句的例子有

```
mother(joanne, jake).
father(vern, joanne).
```

这里说明的是，joanne是jake的母亲，vern是joanne的父亲。

而规则语句的一个例子是

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

这里说明的是，对于具有特定值的变量X，Y和Z，如果这些条件为真：X是Y的parent，并且Y是Z的parent，那么就能够推断出：X是Z的grandparent。

我们能够使用目标语句与Prolog数据库进行交互式查询，这样的一个例子是

86

```
father(bob, darcie).
```

这一条语句意思是询问：bob是不是darcie的父亲。当传送这样的一个查询或者目标给Prolog系统时，这个系统将运用归结过程来试图确定这条语句的真实性。如果这个系统能够导出目标为真的结论，将会显示“true”，否则，将会显示“false”。

2.13.3 评估

在20世纪80年代，只有少数计算机科学家相信，逻辑程序设计方法提供了使我们能够摆脱命令式语言复杂性的最大希望，也是最有希望能够在产生所需要的大量可靠软件中避免无数问题。然而到目前为止，有两个主要因素阻碍了逻辑程序设计方法的广泛应用。首先，也像其他一些非命令式方法一样，与对应的命令式程序相比，用逻辑语言编写的程序被证明是极其低效的。其次，它只是在很少并且是相对小的应用领域中显示出是一种有效方法，这些领域为某种类型的数据库管理系统，以及某些人工智能领域。

Prolog语言的一种方言Prolog++(Moss，1994)支持面向对象的程序设计。关于逻辑程序设计和Prolog语言，将在第16章中给予详细描述。

2.14 历史上规模最大的语言设计：Ada

Ada语言的设计是有史以来涉及范围最广、耗资最大的工作。接下来的段落简要介绍Ada的发展过程。

2.14.1 历史背景

Ada是为美国国防部（DoD）开发的，所以在决定语言的形式时，其计算环境状态是指令式的。在1974年以前，美国国防部内超过半数的计算机应用系统是嵌入式系统。嵌入式系统是这样的系统：在这种系统中，计算机的硬件被嵌入到它所控制或者服务的设备之中。当时软件费用的快速增长主要源于系统复杂性的增加。美国国防部当时使用着450种以上不同种类的程序设计语言用于它的各种项目，但其没有对其中任何一种语言实行过标准化。每一个国防承包商都可以为一份合同定义一种新的不同语言。[6] 由于语言的大量繁殖，应用软件极少重复使用。此外，没有产生开发软件的工具（因为它们通常是附属于语言的），尽管使用着大量语言，但没有一种语言适宜于嵌入式系统的应用。由于这些原因，美国陆、海、空三军于1974年分别独立地提出了开发一种嵌入式系统高级语言的计划。 87

2.14.2 设计过程

国防研究与工程部主任Malcolm Currie注意到了这种广泛要求，于1975年1月成立了高级语言工作组（High-Order Language Working Group，HOLWG）。这个工作组最初由英国空军的Lt. Col. William Whitaker领导。HOLWG中有来自所有军种、各个部门的代表，以及英国、法国和西德的联络官员。它的初始章程为：

- 确认新的美国国防部高级语言的需求。
- 评估现存语言以决定是否存在一种可行的候选语言。
- 推荐采用或者实现一组程序设计语言的最小集合。

1975年4月，高级语言工作组产生了一份称为“草人”（Strawman）的需求文件用于描述新语言（Department of Defense，1975a）。这份文件被分发到各个军事部门、联邦政府单位、有选择的工业及大专院校代表单位，以及对此感兴趣的欧洲团体。

在“草人”文件之后，1975年8月产生了“木人”（Woodenman）文件（Department of Defense，1975b），于1976年1月产生了“铁人”（Tinman）文件（Department of Defense，1976）

6. 这种状况主要是由于用于嵌入式系统的汇编语言的广泛流行，同时绝大多数嵌入式系统都使用特殊的处理器，

于1977年1月产生了“铁人”（Ironman）文件（Department of Defense, 1977），最终于1978年6月产生了“钢人”（Steelman）文件（Department of Defense, 1978）。

在乏味的过程之后，最后提交的语言提案缩小到四家，而且他们都是基于Pascal语言的。1979年5月最终决出Cii Honeywell/Bull的语言设计为优胜者。有趣的是，最后优胜者是参与竞争的四个商家中唯一的外国公司。这家法国的Cii Honeywell/ Bull设计组由Jean Ichbiah领导。

1979年春天，海军军需司令部的Jack Cooper为新的语言推荐了“Ada”名字，后来这个名字被采纳。值得纪念的是，Lovelace 伯爵夫人、数学家、诗人拜伦的女儿Augusta Ada Byron（1815－1851年），她被普遍认为是世界上的第一位程序人员，她曾经和Charles Babbage一起，在她的机械计算机、微分分析引擎上工作过，为许多数值处理编写过程序。

Ada设计本身及其说明由美国计算机协会（ACM）发表于*SIGPLAN Notices*（ACM, 1979）之上，并发行到超过一万的读者手中。1979年10月，在波士顿举行了语言的公开测试与评估会
88 议，与会代表来自美国和欧洲的100多个组织。到11月份，已经收到了来自15个不同国家的500多份关于语言的报告。其中大部分报告都建议进行小的修改，而不是作重大改动，也没有对于这种语言的绝对否定。在这些语言报告的基础上，下一个版本的需求说明“石人”（Stoneman）文件（Department of Defense, 1980a）于1980年2月发表。

Ada语言设计的校订版本于1980年7月完成，这个版本被国人接受并被命名为MIL-STD 1815，也即标准的“Ada语言参考手册”（Ada Language Reference Manual）。选择数字“1815”，是因为它是Augusta Ada Byron的出生之年。“Ada语言参考手册”的另一个校订版本公布于1982年7月。在1983年，美国国家标准协会对Ada语言实行了标准化。这个“最后”的官方版本由Goos和Hartmanis描述（Goos and Hartmanis, 1983）。在此之后，Ada语言的设计至少被冻结了5年。

2.14.3 语言概述

本节我们简略地描述Ada语言的四个主要特性。

Ada语言中的包结构提供了封装数据对象、数据类型的说明以及处理过程的手段。如我们将在第11章里描述的，这为在程序设计之中使用数据抽象提供了支持。

Ada语言包括了大量异常处理的机制，这允许程序人员能够在出现任何一种异常或运行时错误时取得控制。关于异常处理将在第14章中讨论。

Ada中的程序单元可以是通用性的。例如，可以用Ada语言来编写一个排序过程，而不需要说明将要用于排序的数据的类型。但在使用之前，必须根据特定的排序数据将通用过程实例化。实例化可由一条语句来完成，这条语句指示编译器按照给定的数据类型产生排序过程的一个版本。这种通用程序单元增加了可重复使用程序单元的范围，而不是由程序人员来重新产生程序。关于通用性将在第9章和第11章中进行讨论。

通过使用会合（rendezvous）机制，Ada语言还提供了特殊程序单元的并发执行，称之为“任务”（task）。会合是任务间一种进行通讯和同步的方法的名称。关于并发将在第13章中讨论。

2.14.4 评估

Ada语言设计中值得讨论的最重要方面也许是下面几项：

- 由于语言的设计是竞争性的，因而对参与人员没有限制。
- Ada语言包含了20世纪70年代后期软件工程中以及语言设计中的绝大部分概念。尽管人们
89 可以质疑用来包括这些语言特性的实际方法以及在一种语言中包括如此众多语言特性的

可取性，但大多数人还是肯定这些语言特性的价值。

- Ada语言编译器的开发是一项困难的工作，虽然许多人最初没有意识到这一点，直到1985年，Ada语言的设计工作已经完成约4年之后，真正可以实际使用的 Ada 编译器才开始出现。

在Ada出现的最初几年，对它最严厉的批评是过于庞大和复杂。尤其是Hoare曾经声明，Ada语言不应该被用于可靠性至为关键的任何应用上（Hoare，1981）。但不幸，Ada恰恰是专门为这类应用设计的。另一部分人则与此相反，称赞Ada语言为当时语言设计的典范。而在事实上，就连Hoare最终也改变了对Ada语言的严厉观点。

下面是Ada程序的一个例子：

```
-- Ada Example Program
-- Input:  An integer, List_Len, where List_Len is less
--         than 100, followed by List_Len-integer values
-- Output: The number of input values that are greater
--         than the average of all input values
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Ada_Ex is
  type Int_List_Type is array (1..99) of Integer;
  Int_List : Int_List_Type;
  List_Len, Sum, Average, Result : Integer;
begin
  Result:= 0;
  Sum := 0;
  Get (List_Len);
  if (List_Len > 0) and (List_Len < 100) then
-- Read input data into an array and compute the sum
    for Counter := 1 .. List_Len loop
      Get (Int_List(Counter));
      Sum := Sum + Int_List(Counter);
    end loop;
-- Compute the average
    Average := Sum / List_Len;
-- Count the number of values that are > average
    for Counter := 1 .. List_Len loop
      if Int_List(Counter) > Average then
        Result:= Result+ 1;
      end if;
    end loop;
-- Print result
    Put ("The number of values > average is:");
    Put (Result);
    New_Line;
  else
    Put_Line ("Error-input list length is not legal");
  end if;
end Ada_Ex;
```

90

2.14.5 Ada 95

Ada语言的新版本被命名为 Ada 95，具有的两个最重要的新语言特性将在下面的段落简略描述。在本书的剩余部分，当有必要区分新老两个版本时，我们将用Ada 83来指Ada的初始版

本，用Ada 95（它的真实名字）来指后来的新版本。在讨论这两种版本所共有的语言特性时，我们将简单地采用Ada这个名称。Ada 95的标准语言被定义于ARM之中（ARM，1995）。

Ada 83的类型派生机制被扩展以允许从父类型派生的类型上再增加新的成分。这就提供了继承，继承是面向对象程序设计语言中的关键成分。子程序定义与子程序调用之间的动态绑定由子程序的发送来完成，这种发送基于类范围的类型中派生类型的标志值。这种特性就提供了多态。多态是面向对象程序设计的另一种主要特性。Ada 95的这些特性将在第12章中讨论。

Ada 83语言中的会合机制仅仅提供了并发进程中共享数据的一种不方便和低效率的手段。当时已有必要引入新的方法来控制对共享数据的访问。对此，Ada 95中的被保护对象（protected object）提供了一种很有吸引力的替代方法。在这种方法中，将共享数据封装于一种语法结构中，由这种结构控制所有对数据的访问，无论是来自会合机制还是子程序调用。关于Ada 95中并发和共享数据的新特性，将在第13章中讨论。

人们普遍相信：因为美国国防部不再需要在军用软件系统中使用Ada 95，所以Ada 95应用的广泛性受到了影响。当然，其他一些因素也妨碍了这种语言的进一步广泛应用。种种因素之中最为重要的，是C++在面向对象程序设计领域的广泛接受。事实上，C++语言在Ada 95被公布以前就已经很受欢迎了。

2.15 面向对象的程序设计：Smalltalk

Smalltalk是第一种完全支持面向对象程序设计的程序设计语言。因此，对这种程序设计语言发展过程的任何讨论都是重要的。

91

2.15.1 设计过程

促成开发Smalltalk语言的概念始于20世纪60年代后期Alan Kay在犹他大学的博士论文工作（Kay，1969）。Kay以惊人的远见预见到了功能强大的桌面计算机在未来的广泛使用。请记住，第一台微型计算机系统直到20世纪70年代中期才被推出，况且它们与Kay想像中的机器还有很大的距离。Kay所预见的机器每秒执行超过一百万条指令，并具有几兆字节的内存。以工作站为形式的这种机器直到20世纪80年代初期才广泛普及。

Kay相信桌面计算机会由非程序人员使用，因而需要功能很强的人机接口。20世纪60年代后期的计算机大部分是批处理方式的（batch-oriented），为专业程序员和科学工作者大量使用。为了让非程序人员使用，Kay预测计算机必须具有很高程度的交互性，并且使用复杂的图形用户界面。一些关于图形的概念来自Seymour Papert系统中的LOGO图形经验，这些图形被用来帮助孩子们使用计算机（Papert，1980）。

Kay最初想像了一个称为Dynabook的系统，这是一个普通的信息处理器。这个系统部分地基于他曾经帮助设计的Flex语言。Flex主要是立于SIMULA 67语言的基础之上。Dynabook系统基于典型的桌面图形，桌面上有许多纸张，一些纸张被部分地盖着。最上面的一张通常是注意力焦点之所在，而另外的一些纸张则暂时处于注意力之外。Dynabook系统将使用屏幕视窗的画面来模拟这种桌面场景。用户将通过使用键盘或者他或她的手指接触屏幕这两种方法与系统进行交互。在Dynabook系统的初步设计为Kay赢得了博士学位之后，他的下一个目标便是要亲眼看见这种系统的实现。

Kay找到Xerox Palo Alto研究中心（简称Xerox PARC），提出了关于Dynabook的想法。结果他在那里被雇用，并接下来组织诞生了"Xerox学习研究小组"（Learning Research Group at Xerox）。这个研究小组接受的第一项任务是设计一种语言来支持Kay的程序设计范型，并将这种语言实现

于当时最好的个人计算机上。这项工作产生了一个"过渡"的Dynabook系统，它由Xerox Alto的硬件以及被称为Smalltalk-72的软件组成，它们的结合形成了一个进一步开发研究的工具。许多的研究项目都曾经在这个系统上进行，包括教孩子们进行程序设计的一些实验。进一步开发实验的工作同时也在进行，结果产生了以Smalltalk-80为最终版本的Smalltalk系列语言。Smalltalk-80正是本书将要讨论的一种版本。随着这种语言的成长，运行这种语言的硬件的功能也同时在增长。到了1980年，这种语言以及Xerox的硬件几乎已经与Alan Kay最初的设想相一致。 92

2.15.2 语言概述

在Smalltalk语言的世界里，除了对象之外别无他物，对象包括了从简单的整数常数到大而复杂的软件系统。在Smalltalk中的所有计算都由相同一致的技术来完成：发送一条消息给一个对象来调用它的一个方法。而对一条消息的回复是一个对象，这个对象返回所要求的信息，或仅仅通知消息的发送者，它所要求的处理已经完成。消息与子程序调用之间的根本差别在于：消息被发送到一个数据对象，特别是为该对象定义的方法。被调用方法执行，通常修改消息发送到的对象的数据；一次子程序调用是一条发送给子程序代码的消息。通常子程序处理的数据作为一个参数来发送。[9]

在Smalltalk中，对象的抽象是类（class），Smalltalk中的类非常类似于SIMULA 67中的类。可以建立类的实例，并且实例在被创建之后就成为程序中的对象。Smalltalk语法不像任何其他程序设计语言，主要是因为它采用消息机制，而又是使用算术和逻辑表达式或者传统的控制语句。关于Smalltalk控制结构的一个示例将在下一个小节中给出。

2.15.3 评估

Smalltalk在促进两个企业的计算，图形用户接口和面对对象程序设计方面做出了巨大的贡献。现在在软件系统的用户界面方面占统治地位的视窗系统来自于Smalltalk语言。当今最重要的软件设计方法学和程序设计语言都是面向对象的。虽然面向对象语言的最初思想来自于SIMULA 67，但正是Smalltalk语言让这些思想趋向成熟。显然，Smalltalk语言对于计算世界的影响是深刻的，并且将是久远的。

下面是Smalltalk语言中类定义的一个例子：

```
"Smalltalk Example Program"
"The following is a class definition, instantiations of
which can draw equilateral polygons of any number of sides"
class name                      Polygon
superclass                      Object
instance variable names         ourPen
numSides
sideLength                                                    93
"Class methods"
  "Create an instance"
  new
      ^ super new getPen

  "Get a pen for drawing polygons"
  getPen
```

[9] 很显然，方法调用也能传递数据给被调用方法处理。

```
    ourPen <- Pen new defaultNib: 2

"Instance methods"
"Draw a polygon"
draw
    numSides timesRepeat: [ourPen go: sideLength;
                           turn: 360 // numSides]

"Set length of sides"
length: len
    sideLength <- len

"Set number of sides"
sides: num
    numSides <- num
```

The ancestry of Smalltalk is shown in Figure 2.12.

2.16 结合命令式与面向对象的特性：C++

关于C语言的由来曾在2.12节中讨论过，Smalltalk的由来在2.15节中讨论过。C++语言是在C的基础上建立起语言机制，用以支持大部分由Smalltalk所开创的语言功能。C++是对C进行一系列的修改发展而来，它改进了原有的命令式语言特性，并且增加了支持面向对象程序设计的语言结构。

2.16.1 设计过程

从C语向C++的第一步是1980年由Bjarne Stroustrup在贝尔实验室完成的。这次修改包括了增加函数参数的类型检测以及转换，意义尤其重大的是，增加了与SIMULA 67和Smalltalk中的类相似的类。增加部分还包括了派生类、对被继承部分的公用与私有访问的控制、构造函数与析构函数方法以及友元类。1981年还增加了内联函数、默认参数以及赋值运算符的重载。这样产生的语言被称为是“具有类的C语言”，并在Stroustrup的文章中给予了描述（Stroustrup，1983）。

理解这种“具有类的C语言”的目标，将是有帮助的。当时的主要目标是要提供一种能够像SIMULA 67那样运用类与继承来组织程序的语言。它的第二个重要目标是类的加入不应该使它的性能受到影响，即它的性能不应该比C差。例如，没有考虑对修改数组下标范围的检测，因为这样可能会导致相对于C的较大的性能差异。第三个目标是将“具有类的C语言”用于所有能够使用C的应用之中，实际上，C的任何特性都没有被删除，甚至包括那些已经被认为是不安全的特性。

94

到1984年之前，这种语言又得到了扩展，包括了一些虚拟方法以提供方法调用与特殊方法定义之间的动态绑定，方法名与运算符重载以及引用类型。该语言的这个版本被称为C++，它由Stroustrup给予了描述（Stroustrup，1984）。

在1985年，C++的第一次可应用的实现问世了，这是一个被命名为Cfront的系统。这个系统将C++程序翻译成C程序。Cfront系统的这个版本与它所实现的C++版本一起被命名为“Release 1.0”。Release 1.0由Stroustrup描述（Stroustrup，1986）。

从1985年到1989年之间，在很大程度上是基于用户对首次公布的C++实现版本的反应。

C++语言继续发展，产生了被命名为“Release 2.0”的第二个版本。这个版本的Cfront的实现于1989年6月推出。C++ Release 2.0中增加的最重要特性是支持多重继承（具有超过一个父类的类）与抽象类，以及其他一些方面的增强。关于抽象类将在第12章里描述。

C++ Release 3.0于1989年至1990年期间继续演进。这个版本增加了提供参数化类型的模板以及异常处理。于1998年标准化的C++的当前版本由ISO描述（ISO，1998）。

在2002年，微软公司公布了它的.NET计算平台，其中包括C++语言的一种新版本，称为“受控的C++语言”（Managed C++），或者称为MC++。MC++扩展了C++语言，用以提供对于.NET框架功能的访问。这种扩展了的语言还包括了性质（property）、委托（delegate）、接口以及一种用于垃圾收集对象的引用类型。第11章讨论性质，第2.19节引入委托。因为.NET不支持多重继承，MC++也不支持。

2.16.2 语言概述

C++语言包括了两种定义类型的结构：类及struct，而这两种结构之间并没有什么差别。实际上，包括方法定义的struct已经很少使用。C++支持多重继承。在C++中，通常称方法为成员函数（member functions）。

因为C++语言既包括了函数又包括了方法，所以这种语言同时支持过程式程序设计和面向对象的程序设计。

C++中的运算符可以重载，这意味着用户可以在针对已有用户定义类型的运算符上创建新的运算符。C++中的方法也可以重载，这意味着用户可以使用同一个名字来定义多个方法，只要它们的参数数目或者参数类型不相同。

C++中的动态绑定由虚拟方法来提供。这些方法在一组通过继承相关联的类中使用重载方法来定义依赖于类型的操作。一个指向类A的对象的指针也可以指向以类A为前辈类的对象。当 95
这个指针指向一个重载虚拟方法时，可以动态地选择当前类型的方法。

方法与类都可以被模板化，这意味着它们可以被参数化。例如，可以将一个方法写成模板化方法，以允许它能够具有多个参数类型的不同版本。类也具有同样的灵活性。

C++所包括的异常处理与Ada语言有着重大的不同，其中之一是不能处理硬件可检测的异常。Ada以及C++语言的异常处理结构将在第14章中讨论。

2.16.3 评估

C++很快就成为一种非常受欢迎的语言，而且这种形势一直保持着。使得它广为应用的一个因素是它价廉物美的编译器。C++受欢迎的另一个因素是它几乎完全与C向后兼容（这意味着，只要作少许改变，就可以将C程序作为C++程序来编译），而且在大多数实现中，可以将C++的代码与C的代码相链接——当面许多C程序人员掌握C++相对容易。最后一点，当C++语言刚刚出现时，面向对象程序设计也开始受到人们的广泛关注，而C++是当时唯一适合于大型商业软件项目的语言。

至于C++语言的缺点，因为它是一种十分庞大而复杂的语言。虽然它也有着与PL/I语言类似的缺点。它继承了C语言中的大部分不安全因素，这使得C++没有Ada和Java之类的语言安全。

我们将在第12章里对C++语言面向对象的语言特性进行更详细的描述。

2.16.4 一种相关语言：Eiffel

Eiffel是另一种命令式与面向对象的语言特性相混合的语言（Meyer，1992）。Eiffel是由

Bertrand Meyer一个人设计的。Bertrand Meyer是住在美国加州的法国人。这种语言包括了支持抽象数据类型、继承以及动态绑定的语言特性，因而它能够全面地支持面向对象的程序设计。也许Eiffel语言中最显著的语言特性是使用断言来强制实施子程序及其调用之间的"合约"。这种设计思想最初产生于Plankalkül语言，但被大多数后来设计的语言所忽略。我们会自然地将Eiffel与C++进行比较。Eiffel比C++短小而简单，但几乎具有与C++等同的表达性和可写性。之所以C++的应用被为广泛，而Eiffel的应用十分有限，其中原因不难确定。C++最然是许多软件开发组织转向面向对象程序设计的最容易的途径，因为在大多数情况下，它们的开发人员已经掌握了C。Eiffel就没有享有这样容易被采纳的路径。另外，在C++开始传播的前几年，
96 Cfront 系统随手可得而且十分便宜，而在Eiffel的最初几年，Eiffel的编译器就不那么容易得到并且较为昂贵。C++享有声望极高的贝尔实验室的支持，而Eiffel只是由Bertrand Meyer个人与他的相对小型的软件公司——交互式软件工程公司（Interactive Software Engineering）来支持。

2.16.5 另一种相关语言：Delphi

Delphi是一种混合式语言。类似于C++，它的产生是通过增加面向对象的支持以及一些其他特性到现有的命令式语言之上。Delphi是从Pascal演变而来。Delphi与C++之间的许多差别来自它们各自的前辈语言，以及导致产生它们的不同程序设计文化环境。因为C是一种功能强大但具有不安全性的语言，C++也有着同样的特征，至少它在如下方面是不安全的：数组的下标检测、指针的算术运算以及数值类型的强制转换。同样地，因为Pascal语言比C安全，Delphi也比C++更为安全。并且Delphi也没有C++语言那么复杂。例如，Delphi语言不允许用户定义的运算符重载，不允许通用子程序，也不允许参数化的类；而这些都是C++的组成部分。

Delphi像C++一样，给开发人员提供了图形用户界面（GUI），并提供了给由Delphi语言所编写的应用产生图形用户界面的简单方式。Delphi语言的设计者Anders Hejlsberg先前曾经开发了Turbo Pascal系统。他所开发的Delphi语言以及Turbo Pascal系统都由Borland公司销售与发行。Hejlsberg也是C#语言的主要设计人员。

2.17 一种基于命令式的面向对象语言：Java

Java语言的设计人员开始从C++中删除了大量的结构，修改了一些结构，又增加了另外一些结构。这样所产生的语言，具有C++语言的强大功能与灵活性，同时又更为小巧、简单和安全。

2.17.1 设计过程

Java也像许多程序设计语言一样，是针对某种应用而设计的，而当时显然没有现成又满足需要的语言；然而Java语言实际上却是针对一系列的应用而设计的，其中第一类应用是在家用电子装置上进行嵌入式程序设计，例如吐司炉、微波炉和交互式电视系统。看起来，似乎一台微波炉软件的可靠性不至于成为重要因素。如果一台微波炉的软件发生故障似乎不大可能对任何人造成致命的威胁，也不至于引起重大的法律纠纷。但是，如果某种型号上的软件有错误，而在一百万个这种型号的产品已经被制造并售出之后才发现，这时召收回这些产品必将带来极
97 其重大的损失。因此，在消费性电子产品的软件中，可靠性仍然是一个重要特征。

在1990年，Sun公司（Sun Microsystems）已经确信，他们考虑的两种程序设计语言C#和C++，

都不满足Sun公司在家用电子装置中开发软件的需要。尽管C++非常地精练，但它不提供面向对象程序设计的支持，而他们认为这是必需的语言特征。C++虽然支持面向对象的程序设计，但他们判定这种语言过于庞大和复杂。其中部分原因是因为C++也支持面向过程的程序设计。他们还相信，C与C++语言都不能提供所要求的可靠程度。Java语言的设计由这样的基本信念所指导：它应该具有比C++语言所能够提供的更好的简单性和可靠性。

尽管推动Java设计的原始动力是家用电子产品，但是早期配备了Java程序的这类产品从没有被销售出来，从1993年开始，因为出现了大量新的图形式浏览器，万维网开始被广泛使用，这时人们才发现Java是网络程序设计的一种有用工具。尤其是，相对小巧的Java程序Java applet输出效果能显示在Web文档中，所以在20世纪90年代中期很快就流行了起来。在万维网被公开使用的最初几年，它成为Java语言最常见的应用领域。

Java设计小组由James Gosling领导，他先前曾经设计过UNIX系统的emacs编辑器与NeWS视窗系统。

2.17.2 语言概述

如前所述，Java是以C++为基础的，但又特别设计得更加小巧、简单和可靠。Java兼有类与基本类型，这一点如同C++中的一样。Java的数组是预定义的类的实例，而C++中的数组则不是，尽管有许多C++的使用人员为数组建立起封装类，以便增加下标范围检测这样的特性。而这样的语言特性在Java中是隐含的。

Java语言没有指针，但它的引用类型提供了指针的一些功能。这些引用被用来指向类的实例，而事实上，这是引用类的实例的唯一方法。所有对象都在堆上被分配。尽管指针与引用表面上似乎极其相像，但它们之间在语义上却有一些重大不同。指针指向存储单元的地址，但引用指向对象。这使得对引用进行任何算术运算都毫无意义，从而避免了一些容易出错的做法。区分指针的值与指针所指向的值之间的差别，在大多数语言中是程序人员的责任，这时常常必须将指针显式间接引用（dereference）。然而在必要的时候，引用总是隐式间接引用，因而引用的行为更接近普通的标量变量。

Java具有一个基本布尔类型（boolean），主要用于控制语句中的控制表达式（如在if和while语句中）。与C和C++语言不同，Java里的算术表达式不能够用来作为控制表达式。

Java与更早、支持面向对象程序设计的语言（包括C++）之间的另一个重要区别，是在Java中不能够编写独立的子程序。Java中所有子程序都是定义于类之中的方法。此外，任何对于方法的调用只能通过类或者对象。这种方式的结果就是，Java只能够支持面向对象的程序设计，而C++则能够支持面向过程以及面向对象的程序设计。

C++与Java之间的另外一个重要区别，是C++在它的类定义中直接支持多重继承。有些人认为多重继承带来过多的复杂性与混乱，因而不值得采用。Java仅仅支持类的单继承，但通过使用Java语言中的接口结构，能够获得一些多重继承的优点。

C++的结构中没有被Java语言复制的是struct和union。

Java语言通过它的同步（synchronize）修饰符包括了一种相对简单的并发控制形式。这种并发控制能够出现在它的方法与块结构中。这给方法与块结构附加了一把锁，而这种锁保证了互斥访问或互斥执行不至于发生。在Java语言中产生并发进程相对容易，并发进程在Java语言中被称为线程。

Java隐式解除对对象的存储空间分配，这通常被称为垃圾收集。有了垃圾收集，程序员就不必显式地删除不再需要的对象。没有垃圾收集的语言所编写的程序经常出现所谓内存泄漏的

问题。这个问题指的是，被分配了的存储单元从来没有被解除分配，最终将耗尽所有的存储空间。

不同于C与C++语言，Java中的赋值类型强制转换（强式类型转换）仅仅允许类型转换的"宽化"（即从一种较"小"的类型转换为一种较"大"的类型），因而可以通过赋值运算符完成从int到float的强制类型转换，但不能完成从float到int的强制类型转换。

2.17.3 评估

Java的设计人员在剪裁C++中多余的以及/或者不安全的特性方面做了很出色的工作。例如，从C++中的赋值强制转换里删掉一半，这显然是迈向更高可靠性的一步。数组下标范围的索引检测也使得语言更为安全。而增加并发功能则更扩大了语言的应用范围。增加下列功能也达到了同样的功效，如小应用程序（applet）类库、图形用户界面、数据库的访问以及网络化。

Java的可移植性，至少是以中间代码的形式，常常被人们归功于Java的语言设计，其实却不然。任何语言都可以被翻译成中间代码的形式，并且能够"运行"于任何适于这种中间形式的虚拟机器平台上。这种可移植性的代价便是解释的代价，传统上解释的代价大约比机器码运行的代价高一个数量级。Java解释器的最初版本被称为Java虚拟机（Java Virtual Machine，JVM），它至少比功能类似的编译的C程序慢十倍。然而，现在的Java程序在执行之前被翻译成为机器代码，使用即时编译器（Just In Time，JIT），这使得Java程序的效率已经可以与编译式语言（如C++）的程序不相上下。

99

Java语言应用的增长速度远超过任何其他的程序设计语言。最初是因为它在动态万维网程序设计方面的价值。另一个原因则是由于Java的编译器/解释器系统是免费的，而且从万维网上获取十分容易。而Java语言迅速升至显著位置的最主要原因就在于程序人员喜欢它的设计。总有一些开发人员认为，C++语言过于庞大与复杂，不便于使用，还不够安全。Java给他们提供了一种替代方案，它具有C++的大部分功能，但更为简单与安全。今天，Java语言已经被广泛地用于各种各样的应用领域。

最近的Java语言版本增加了一些重要内容。这个版本出现于2004年，最初被称为Java 1.5，后来又被称为Java 5.0。新增的内容包括一种枚举类、一些适用结构，以及一种新的循环结构。

下面是Java程序的一个例子：

```
// Java Example Program
//   Input: An integer, listlen, where listlen is less
//          than 100, followed by length-integer values
// Output: The number of input data that are greater than
//          the average of all input values
import java.io.*;
class IntSort {
public static void main(String args[]) throws IOException {
  DataInputStream in = new DataInputStream(System.in);
  int listlen,
      counter,
      sum = 0,
      average,
      result = 0;
  int[] intlist = new int[99];
  listlen = Integer.parseInt(in.readLine());
  if ((listlen > 0) && (listlen < 100)) {
/* Read input into an array and compute the sum  */
    for (counter = 0; counter < listlen; counter++) {
```

```
        intlist[counter] =
              Integer.valueOf(in.readLine()).intValue();
        sum += intlist[counter];
      }
/* Compute the average */
      average = sum / listlen;
/* Count the input values that are > average */
      for (counter = 0; counter < listlen; counter++)
        if (intlist[counter] > average) result++;
/* Print result */
        System.out.println(
            "\nNumber of values > average is:" + result);
      }  //** end of then clause of if ((listlen > 0) ...
      else System.out.println(
              "Error-input list length is not legal\n");
  }  //** end of method main
}  //** end of class IntSort
```

2.18 脚本语言：JavaScript、PHP、Python和Ruby

万维网的爆炸性发展发生于20世纪90年代的中期，第一个图形浏览器出现之后。由于HTML文件的本身完全是静态的，因而与HTML文件相关的计算需要很快就成为了关键。通用网关接口使得在服务器终端上的计算成为可能。通用网关接口（Common Gateway Interface，CGI）允许HTML文件要求执行服务器上的程序，又将这种计算的结果以HTML文件的形式返回给浏览器。而Java的小应用程序的出现，则使得浏览器上的计算成为了现实。现在，这两种方式都逐步被一些更新的技术所代替，主要是被脚本语言所代替。本节将简略地讨论两种最流行的脚本语言：JavaScript语言，它是居于HTML文件内、在客户机上运行的脚本语言；PHP语言，它是居于HTML文件内、在服务器上运行的脚本语言。本节还将简略地讨论另外两种脚本语言——Python语言和Ruby语言。Python和Ruby不是仅仅为了万维网应用而创建的，尽管它通常用于适用网关接口的程序设计。

100

请注意，Perl通常被认为是一种脚本语言，但事实上Perl更像C语言，而不像典型的脚本语言。Perl语言在2.12.3节里讨论。

2.18.1 JavaScript的起源及特征

原名为LiveScript的JavaScript（Flanagan，1998）在Netscape开发产生。1995年下半年，LiveScript语言成为Netscape与Sun公司的联合事业，并被改名为JavaScript。通过增加许多新的特性及功能，JavaScript语言从版本1.0到1.5经历了巨大的演变。JavaScript语言的标准化于20世纪90年代后期由欧洲计算机制造业协会（European Computer Manufacturers Association，ECMA）开发，作为文件ECMA-262。这种标准化也获得了国际标准化组织（ISO）的批准，为文件ISO-16262。微软公司的JavaScript版本被命名为JScript。

尽管可以将JavaScript解释器嵌入到许多不同的应用之中，但最经常的用法是嵌入Web浏览器。JavaScript代码嵌入HTML文档，在文档显示时解释执行。这种语言在万维网中的主要应用是确保表单输入数据的正确性，以及HTML文件的动态产生。

101

虽然它的名称为JavaScript，但它与Java语言的关联仅仅是它们使用相类似的语法。Java是强类型化语言，而JavaScript中的类型则是动态的（参见第5章）。JavaScript的字符串及数组都具有动态长度。正是因为如此，JavaScript语言不检测数组下标的有效性，而Java语言则要求检测。

Java语言全面地支持面向对象的程序设计，JavaScript则既不支持继承，也不支持方法调用与方法之间的动态绑定。

JavaScript语言的一个最重要应用，是动态地产生与修改HTML文件。JavaScript定义一个对象层次结构，这个结构与文档对象模型（Document Object Model）所定义的HTML文件的层次模型相吻合。所有对于HTML文档元素的访问都必须经过这些对象，这提供了对于HTML文档元素实施动态控制的基础。

下面是一段JavaScript脚本的例子，因为嵌入在HTML文件之中，它的外观十分奇特。这个程序可以在任何装载了JavaScript解释器最新版本的万维网浏览器上运行。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.1 //EN"
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11-strict.dtd">
<!-- example.html
     Input: An integer, listLen, where listLen is less
            than 100, followed by listLen-numeric values
     Output: The number of input values that are greater
            than the average of all input values
     -->
<html xmlns = "http://www.w3.org/1999/xhtml">
<head><title> Example </title>
</head>
<body>
<script type = "text/javascript">
<!--
var intList = new Array(99);
var listLen, counter, sum = 0, result = 0;

listLen = prompt (
        "Please type the length of the input list", "");
if ((listLen > 0) && (listLen < 100)) {
// Get the input and compute its sum
   for (counter = 0; counter < listLen; counter++) {
      intList[counter] = prompt (
                    "Please type the next number", "");
      sum += parseInt(intList[counter]);
   }
// Compute the average
   average = sum / listLen;
// Count the input values that are > average
   for (counter = 0; counter < listLen; counter++)
      if (intList[counter] > average) result++;
// Display the results
   document.write("Number of values > average is: ",
                  result, "<br />");
} else
   document.write(
       "Error - input list length is not legal <br />");
// -->
</script>
</body>
</html>
```

2.18.2 PHP的起源及特征

PHP语言（Converse和Park，2000）是1994年由Apache小组的一个成员Rasmus Lerdorf所开发的。最初的动机是想给Lerdorf的个人网站提供一种工具，以便帮助追踪网站的访问者。在1995年，Lerdorf开发了一个称为“个人网页工具”（Personal Home Page Tools）的软件包，这个软件包成为了首次发布的PHP语言版本。起初，名称PHP是英文Personal Home Page（即个人网页）的首字母，但后来这个语言的用户群体开始称之为PHP（超级文本预处理器，Hypertext Preprocessor），结果使得原来的名称反而被人遗忘了。现在，PHP语言被作为开放源代码（open-source）产品来开发、发布以及被支持，PHP处理器是大多数万维网服务器的内在成分。

PHP是专门为万维网应用而设计的，是嵌入HTML文件的服务器端的脚本语言。当万维网浏览器请求一份嵌入了PHP代码的HTML文件时，万维网服务器则将文件中的PHP代码进行解释。通常，PHP代码所产生的输出为HTML文件的代码，而这种输出将替代HTML文件中的PHP代码。因而，万维网浏览器端从来就不会看见PHP代码。

在语法的表现，在字符串和数组的动态特征，以及使用动态类型化等方面，PHP语言与JavaScript都十分类似。PHP语言的数组结合了JavaScript的数组以及Perl语言的相关数组。

早先版本的PHP不支持面向对象程序设计，但是在第二个发布版就支持这种特性了。然而，PHP仍然不支持抽象类、接口、析构或对类成员的访问控制。

PHP允许HTML表单数据的简单访问，因而PHP的表单处理十分容易。PHP还提供了对于多种不同数据库管理系统的支持，这使得它成为创建万维网数据库访问程序的优秀语言。 105

2.18.3 Python的起源及特征

相对而言，Python（Lutz&Ascher，2004）是一种更新的、面向对象的解释式脚本语言。它的最初版本是由Guido van Rossum于20世纪90年代初在荷兰的Stichting Mathematisch Centrum完成设计的，而现在的开发工作是由“Python软件基金协会”（Python Software Foundation）承担。Python被用于与Perl语言同类型的应用领域：系统的行政管理、适用网关接口的程序设计，以及其他一些较小型的任务。Python是一种开放源代码系统，并且存在于多数一般计算平台上。用于微软视窗系统的、工业标准化的Python语言版本可以从网址www.activestate.com/Products/ActivePython获得。Python在其他一些计算平台上的实现版可从网址www.python.org获得。这个网站还有关于Python语言的极为丰富的信息。

Python的语法没有直接以任何一种常用的语言作为基础。它是类型检测的，然而是动态类型化的语言。Python语言使用三种类型的数据结构代替了数组：表、被称为元组（tuple）的不变表，以及被称为字典（dictionary）的散列。Python具有一组表方法，如append、insert、remove以及sort。Python还有一组用于字典的方法，如keys、values、copy以及has_key。Python语言也支持原来属于Haskell语言的表领悟（list comprehension）。关于表领悟，我们将在第15.8节讨论。

Python是面向对象的语言，它包括了Perl语言的模式匹配功能，还具有异常处理功能。垃圾收集在对象不再使用时回收它们。

Python语言对于适用网关接口的支持，尤其是对于表单处理的支持，是由cgi模块来提供的。支持cookie、连网与数据库访问的模块也是Python系统中的部分。

一个更有趣的Python语言特性是任何用户都可以很容易地扩展这种语言。可以使用任何编译式语言来编写支持语言扩展的模块。在这些扩展中可以包括函数、变量，以及一些对象类型。

这些扩展被实现为Python语言解释器的附加部分。

2.18.4 Ruby的起源及特征

Ruby（Thomas *et al.*, 2005）是Yukihiro Matsumoto（aka Matz）在20世纪90年代初设计并于1996年发布的一种语言。然后它经历了一段连续的发展过程。Ruby诞生的动机是它的设计者对Perl和Python的不满意。尽管Perl和Python都支持面向对象程序设计，但是它们都不是完全的面向对象程序设计语言。至少它们都包含基本类型（非对象）和支持函数。

像Smalltalk一样，Ruby的基本特性是完全的面向对象程序设计。每个数值都是对象，所有操作都必须通过方法调用来完成。Ruby中的操作符只是用来完成相应操作指定方法调用的句法
104 机制。由于它们都是方法，所以它们都能被重定义。所有预定义和用户定义的类都能子类化。

在Ruby中，类和对象都是动态的，这意味着它们能动态加入方法。同时，这也意味着在每次执行期间，类和对象都有不同的方法集。这此，同一个类的不同实例呈现不同的行为。方法，数据和常量都能包含在类的定义中。

Ruby的语法与Eiffel和Ada是有关联的。不需要声明变量，因为使用了动态类型化机制。变量名指定了其作用域：以字母开头的变量具有局部作用域；以@开头的变量是实例变量（instance variable）；以$开头的变量具有全局作用域。许多Perl特性都出现在Ruby中，包含隐含变量，如$_。

因为Ruby的易用性，任何用户都能扩展和/或修改Ruby。Ruby具有很强的文化趣味，因为它是第一种由日本人设计并获得广泛应用的程序设计语言。

2.19 一种基于C的新世纪语言：C#

在2000年，微软公司在宣布它的新开发平台.NET[⊖]的同时[⊜]也推出了C#语言。在2002年1月，微软发布了这两种新产品的版本。

2.19.1 设计过程

C#是基于C和Java语言的，但也包括了一些Delphi以及Visual BASIC的思想。C#语言的主要设计者是Anders Hejlsberg，他也曾经设计过Turbo Pascal和Delphi语言，这正是C#部分地继承Delphi语言的缘由。

C#语言的目标是要为基于组件（component-based）的软件开发提供一种语言，尤其是为在.NET框架里进行的并发提供语言。在.NET环境中，来自各种不同语言的组件可以很容易地结合起来形成系统。所有.NET的语言，包括C#，Visual Basic.NET，MC++，J#.NET以及JScript.NET，都使用“通用类型系统”（Common Type System，CTS）。这种通用类型系统提供了一个通用的类库。这五种.NET语言中的所有类型都继承自同一个根，即System.Object，可以将适合通用类型系统规范的编译器产生的对象结合进软件系统。所有五种.NET的语言都被编译成为同一种中间形式，即“中间语言”（Intermediate Language，IL）。[⊜]然而与Java语言不同的是，这个系统从来不对“中间语言”进行解释。在“中间语言”被执行之前，即时编译器
105 （Just-In-Time）将“中间语言”翻译成为机器代码。

⊖ 关于.NET开发系统，曾经在第1章里简略地讨论过。

⊜ 最初，IL称为MSIL（Microsoft Intermediate Language），但是很明显，许多人认为它的名字太长了。

2.19.2 语言概述

许多人相信，Java语言比C++优越的一个重要之处，在于Java语言删除了C++中的一些语言特性，例如，C++支持多重继承、指针、structs、enum类型、运算符重载以及goto语句，这些都没有包括在Java语言中。而C#语言的设计人员显然不同意这种将语言特性成批删除的方式，他们将上面列举的语言特性（除了多重继承以外）重新搬回了新的语言。

然而在某些情形下，C#语言的设计人员在C#版本中改进了原来C++的语言特性。例如，C#中的enum类型因为不能够隐式转换成整数类型，从而比在C++中的enum类型更为安全。这使得C#中的enum类型具有类型安全性。C#中的struct类型也经历了重大改变，使得这种类型成为了一种真正有用的结构；而C++中的这种类型并没有实用目的。在C#中struct是一种轻型类，它不支持继承或子类。然而C#中的struct可以实现接口，还可以具有构造函数；并且它们是数值类型的，这意味着它们是在运行时的栈上分配和直接访问，而不是通过引用访问。C#语言中的所有基本类型都被实现为struct。

C#语言试图改进C、C++以及Java语言中使用的switch语句。在这些语言中，由于可选择代码段的终端不存在隐式分支，从而引起了大量的程序错误。在C#中，每个非空的case段必须用一个非条件分支语言来结束。因而，如果你想控制执行流程，从一个case段到下一个case段，就必须使用一条goto语句来分支。

C#语言包括了函数指针，它们同样具有从C++的变量指针继承的不安全性。C#包括了一种新的类型——代表，它是面向对象的，也是类型安全的方法引用。代表被用于实现事件句柄以及回调。[2]在Java中实现的回调具有接口，而在C++中的回调则使用方法指针。

在C#中，方法可以具有不同数目的参数，但这些参数必须是同一种类型。这通过在前面放置保留字params的数组类型的形式参数来给予说明。

C++与Java语言都使用两套不同的类型系统，一套用于基本类型，另外一套用于对象。这除了造成混乱之外，还导致经常需要在这两套系统之间进行数值转换。例如，将基本类型的值放置到一个储存了对象的组合之中。C#通过隐式的boxing以及unboxing操作提供了在这两套类型系统间半隐式的数值转换，关于这些，我们将在第12章予以讨论。[3]

其他的C#语言特性还有长方数组，大多数的程序设计语言都不支持这个特性；还有一条foreach语句，它是数组以及集合对象的迭代器。在Perl、PHP以及Java 5.0里，都可以找到一条类似的foreach语句。另外，C#语言还包括了属性（property），属性是公有数据成员的一种替代。将属性声明为具有get与set两种方法的数据成员，当对相关的数据成员进行引用或者赋值时，则隐式地调用属性。 106

2.19.3 评估

C#语言的初衷是要成为优越于C++和Java的通用程序设计语言，虽然对于它的某些特性是否在倒退可能具有争议，但C#语言显然包含了超过其前辈语言的语言结构。C#所企望的基本应用是作为.NET环境中的一种主要语言，要说C#语言将吸引极大量的用户，现在还为时过早。然而，在微软公司销售.NET所做出的巨大努力的前提之下，C#成为广泛应用的语言有着极为优越的机会。此外，它的一些语言特性肯定会被将来的程序设计语言所采纳。

---

[2] 当一个对象调用另外一个对象的一种方法，而这个方法完成了它的任务而需要通知那个对象时，此时就将调用程序再调用回来，这种调用就称为回调。

[3] Java 5.0加入了这个特性。

下面是C#程序的一个例子：

```
// C# Example Program
// Input:  An integer, listlen, where listlen is less than
//         100, followed by listlen-integer values.
// Output: The number of input values that are greater
//         than the average of all input values.
using System;
public class Ch2example {
  static void Main() {
    int[] intList;
    int listlen,
        counter,
        sum = 0,
        average,
        result = 0;
    intList = new int[99];
    listlen = Int32.Parse(Console.readLine());
    if ((listlen > 0) && (listlen < 100)) {
// Read input into an array and compute the sum
      for (counter = 0; counter < listlen; counter++) {
        intList[counter] = Int32.Parse(Console.readLine());
        sum += intList[counter];
      } //- end of for (counter ...
// Compute the average
      average = sum / listlen;
// Count the input values that are > average
      foreach (int num in intList)
        if (num > average) result++;
// Print result
      Console.WriteLine(
        "Number of values > average is:" + result);
    } //- end of if ((listlen ...
    else
      Console.WriteLine(
        "Error--input list length is not legal");
  } //- end of method Main
} //- end of class Ch2example
```

107

2.20 标志与程序设计混合式语言

标志（markup）与程序设计混合式语言是一种标志语言，但是它的一些元素又能够说明某些程序设计的行为，如控制程序流程以及运算。下面的小节将介绍两种这样的混合式语言，XSLT与JSP语言。

2.20.1 XSLT

可扩展标志语言（eXtensible Markup Language，XML）是一种元标志语言。这种语言被用来定义标志语言，而由XML派生的标志语言则用来定义数据文件，这些数据文件被称为XML文件。尽管XML文件也可以供人阅读，但主要用于计算机处理。有时候，这种处理只是将文件转换成为一种能够高效率显示或打印的形式。但更多的时候，是将文件转换成为万维网浏览器可以显示的HTML文件，而在其他一些时候，则是处理文件中的数据，就像处理其他形式的数据文件一样。

一种专门为显示转换XML文件的方式，是使用另一种标志语言——可扩展样式表语言转换（eXtensible Stylesheet Language Transformations，XSLT）（www.w3.org/TR/XSLT）。XSLT的转换可以声明类似程序设计的操作，因而，XSLT是一种标志与程序设计混合式语言。在20世纪90年代的后期，世界万维网协会（W3C）给予了XSLT语言的定义。

XSLT处理器是一个程序。它接受XML数据文件和XSLT文件（它也是一种XML文件的形式）作为输入。在这种处理过程中，它使用在XSLT文件里所声明的转换将XML数据文件转换成为另外一种XML文件。[1] XSLT文件通过定义模板来声明这种转换。这些模板是一些数据的模式，而XSLT语言的处理器能够在XML输入文件中找到这些数据模式。与XSLT文件中的每一个模板相关联的是它的转换指令，这些指令说明在将这些匹配的数据放入输出文件之前，如何将这些数据进行转换。因而，这些模板（及其相关联的处理过程）具有子程序的行为，当XSLT处理器发现XML文件中的数据与一个模式相匹配，就会“执行”这个所谓的“子程序”。 108

XSLT语言也具有低层次的程序设计结构。例如，它包括了一种循环结构，这个结构允许选择XML文件中重复的部分。它还具有排序的过程。可以使用一些XSLT的标记来声明这些低层次的结构，例如<for-each>。

2.20.2 JSP

Java服务器页面标准标记库（Java Server Pages Standard Tag Library，JSTL）的核心部分是另一种标志与程序设计混合式语言，尽管这种语言的形式以及目标与XSLT语言十分不同。在讨论JSTL语言之前，有必要介绍Servlet以及Java服务器页面（Java Server Pages）的思想。**servlet**是一种内居于并且运行于万维网服务器的Java类。servlet的执行源于万维网浏览器所显示的标志文件发出请求，而servlet的输出以HTML文件的形式返回到发出运行请求的浏览器上。运行于万维网服务器进程中的一个程序称为**Servlet容器**（Servlet container），它控制着Servlet的执行。Servlet通常被用于表格的处理以及数据库的访问。

JSP语言是一组设计技术，被用于支持动态的万维网文件，以及提供万维网文件处理的其他需求。

JSP文件通常是HTML文件与Java程序的混合。当浏览器请求一份JSP文件时，内居于服务器系统的JSP处理程序将这份文件转换成为一个Servlet。而将文件中所嵌入的Java代码复制到Servlet上。纯HTML则被原封不动地复制到Java输出语句之中，如下面的段落将要讨论的，JSP文件中的JSTL标志也被处理。JSP处理器所产生的Servlet由Servlet容器来运行。

JSTL语言定义了一组XML行动元素，它们控制服务器上的JSP文件的处理。这些元素的形式与HTML或XML中的元素相同。一种最常用的JSTL控制行动元素是if。它说明了作为属性的布尔表达式。只有当布尔表达式的值为真时，if元素的内容（在开始标志<if>与结尾标志</if>之间的文字）则是放入输出文件中的标志代码。这个if元素与C/C++语言中的#if预处理器相关联，JSP的容器处理JSP文件的JSTL标志部分，与C/C++语言中的预处理器处理C/C++程序的方式相类似。预处理器的命令是指示预处理器以说明如何从输入文件来构造输出文件。同样，JSTL的控制行动元素是对JSP处理器发出指令，以说明如何从输入的XML文件来构造XML输出文件。 109

if元素的一种通常用法是验证浏览器上用户所传送的表单数据的有效性。JSP处理器能够

[1] XSLT处理器的输出文件也可以是HTML文件的形式，或者一般文件的形式。

获取表单数据，并且能够使用if元素进行测试，以保证数据的合理性。如果这些数据不合理，if元素将为用户在输出文件中插入出错信息。

JSTL语言有choose、when和otherwise元素，用于多选择控制。JSTL语言还包括一个forEach元素用于集合循环，集合通常是来自客户的表单值。forEach元素可以包括begin、end和step属性来控制循环。

小结

我们探讨了许多种最重要的程序设计语言的发展，以及它们的开发环境。读者们应该已经从这一章里获得对于当前语言设计问题的正确观点。我们希望所有的这些能够为深入讨论当代程序设计语言的重要特性打下基础。

文献注释

也许，关于程序设计语言发展历史资料的最重要来源是由Richard Wexelblat所编辑的*History of Programming Languages*（Wexelblat，1981）一书，这本书包括了13种重要程序设计语言的开发背景与开发环境，而且都是语言设计人员自己叙述的故事。一本与此相类似的著作来源于第二次"历史性"的会议，这次出版了*ACM SIGPLAN Notices*（ACM，1993a）的一个特集。在这一著作中，对13种程序设计语言的历史及演化进行了讨论。

Early Development of Programming Languages（Knuth和Pardo，1977）一文是*Encyclopedia of Computer Science and Technology*中的一部分，这篇85页的文章是一个优秀作品，它详细叙述了直到Fortran语言为止的所有程序设计语言的发展历史。这篇文章还包括了一些程序示例，用以对大部分语言中的特性进行示范。

另外一本极有意义的书是*Programming Languages: History and Fundamentals*，由Jean Sammet所著（Sammet，1969）。这本785页的著作记载了从20世纪50年代到60年代的80种程序设计语言的各种细节。Sammet后来又出版了这本书的几个更新版本，其中的一个版本是*Roster of Programming Languages for 1974-75*（Sammet，1976）。

110

复习题

1. Plankalkül是哪一年设计的？又是在哪一年发布的？
2. Plankalkül中包括了哪两种常用的数据结构？
3. 20世纪50年代初期的伪代码是怎样实现的？
4. 快速码的发明是为了克服计算机硬件在20世纪50年代初期存在的两个重大缺点。这两个缺点是什么？
5. 为什么在20世纪50年代的初期，人们可以接受程序解释的慢速度？
6. 在IBM 704计算机中首先出现的什么硬件功能强烈地影响了程序设计语言的发展？请解释为什么？
7. Fortran语言的设计项目是哪一年开始的？
8. 在Fortran的设计时期，计算机的主要应用领域是什么？
9. Fortran I中的所有控制流语句的来源是什么？
10. 在Fortran I中增加了哪些最重要特性，从而产生了Fortran II？
11. 在Fortran IV中增加了哪些控制流语句，从而产生了Fortran 77？
12. Fortran的哪种版本最先具有任意类型的动态变量？
13. Fortran的哪种版本最先具有字符串处理？
14. 为什么20世纪50年代的语言学家对人工智能感兴趣？
15. LISP是在哪里被开发的？又是由谁开发的？

16. Scheme与COMMON LISP在什么方面是相互对立的?
17. LISP的哪种方言被用于一些大学的程序设计概论课程?
18. 哪两个专业组织一起设计了ALGOL 60?
19. ALGOL的哪一个版本中出现了块结构?
20. ALGOL 60所缺乏的哪种语言元素，导致它失去了被广泛应用的机会?
21. 哪一种语言是被设计用来描述ALGOL 60的语法的?
22. COBOL是基于哪种语言创建的?
23. COBOL的设计过程开始于哪一年?
24. 在COBOL中出现的哪一种数据结构源于Plankalkül?
25. 哪一个组织对COBOL早期成功的贡献最大（就应用的广泛程度而言）? 111
26. BASIC的一个版本是以哪一个用户团体为目标的?
27. 为什么BASIC是20世纪80年代初期的一种重要的程序设计语言?
28. PL/I是为了代替哪两种语言而设计的?
29. PL/I是为了哪种新型计算机系列而设计的?
30. SIMULA 67中的哪些语言特性现在是面向对象程序设计语言的重要组成部分?
31. 哪一种数据结构的改革是由ALGOL 68引入，但是常常被叫成Pascal?
32. 哪一种设计标准在ALGOL 68中被大量地运用?
33. 哪一种语言引入了case语句?
34. C语言中的哪些运算符模仿了ALGOL 68中的类似运算符?
35. C语言中的哪两种特征导致了C没有Pascal安全?
36. 什么是非过程的语言?
37. Prolog数据库所具有的两种语句是什么?
38. Ada语言主要是为哪一种应用领域而设计的?
39. Ada中的并发程序单元被称为什么?
40. Ada中的哪一种结构提供了对抽象数据类型的支持?
41. 是什么构成了Smalltalk的世界?
42. 哪三种概念是面向对象程序设计的基础?
43. 为什么C++包括了C语言中已知的不安全特性?
44. Ada语言和COBOL语言有什么共同之处?
45. Java的第一种应用是什么?
46. Java语言的什么特征在JavaScript中有最明显的表现?
47. PHP以及JavaScript中的类型系统与Java语言中的类型系统有什么不同?
48. 哪一种数组结构在C#中有，而在C、C++或者Java语言中都没有?
49. C#中包括了Delphi的类的什么特性?
50. C#新采用的switch语句，就原来在C语言中的switch语句的哪种缺陷进行了改进?
51. XSLT处理器的输入是什么?
52. XSLT处理器的输出是什么?
53. JSTL中的什么元素是与工程序相关联的?
54. JSP处理器将JSP文件转换成什么?
55. Servlet程序在哪里运行? 112

练习题

1. 如果当时Fortran的设计人员熟悉Plankalkül的话，你认为Plankalkül中的哪些语言特性会对Fortran 0产生最大的影响?

2. Backus的701快速编码系统有什么功能？请将这些功能与当代可编程手持计算器的功能进行比较。
3. 写出一小段关于Grace Hopper和她的助手们设计的A-0、A-1和A-2系统的内容。
4. 作为一个研究项目，将Fortran 0语言的实现与Turing和Zuse系统的机制进行比较。
5. 依你之见，ALGOL设计委员会的最初三个目标中的哪一个是未实现？
6. 关于LISP程序中最常见的语法错误，做出一个有根据的猜测。
7. LISP开始是作为一种纯函数式语言，但逐渐接受了越来越多的命令式特性。为什么？
8. 请用你的观点详细描述三条最重要的理由，说明为什么ALGOL 60没有成为一种较为广泛应用的语言。
9. 依你之见，为什么当Fortran以及ALGOL语言不允许长标识符时，COBOL语言却允许？
10. 列举IBM开发PL/I语言的主要动机。
11. IBM开发PL/I的主要动机是必要的？参考自1964年以来计算机和语言发展的历史。
12. 使用你自己的语言，描述程序设计语言设计中的正交性概念。
13. 使得PL/I比ALGOL 68的应用更为广泛的主要原因是什么？
14. 支持和反对元类型语言的两种不同的论点是什么？
15. 除了Prolog语言之外，还有其他的逻辑程序设计语言吗？
16. 对于使用过于复杂的语言太危险这种论点，你的看法是什么？我们是否应该因此保持所有的语言精炼与简单？
17. 你认为由委员会进行语言设计是一种好的方法吗？请给出理由来支持你的观点。
18. 语言在不断地演化，对于程序设计语言的修改，你认为应该给出什么样的适当限制？请将你的答案与Fortran的演化过程进行比较。 113
19. 建立一种表格来标记所有主要的程序设计语言的发展，同时记录它们引用时出现，首先出现在哪种语言之中，以及语言开发人员的身份。
20. 关于微软的J++和C#设计与开发的Java设计，微软与Sun公司之间有过一些公开的交流。请从它们各自的网站上阅读一些有关的文献，并写出你对于这两个公司关于"委托"之不同观点的分析。
21. 请简略地给出一种标志与程序设计混合式语言的一般概述。

程序设计练习题

1. 为了理解程序设计语言中的记录的作用，用C语言编写一个小程序，并使用数组结构来存储学生信息，包括名字、年龄、GPA（float型）和年级（string型，例如，"freshman"等）。然后再用同样的语言但不使用结构来编写这个程序。
2. 为了理解程序设计语言中递归的作用，编写一个程序来实现快速排序。先使用递归实现，然后再使用非递归实现。
3. 为了理解计数循环的作用，使用计数循环结构来编写一个程序以实现矩阵乘法。然后仅使用逻辑循环来编写相同功能的程序，如while循环。 114

第3章 描述语法和语义

本章包括下列内容。首先，我们将给出语法以及语义这两个术语的定义，然后详细地讨论一种最常用的描述语法的方法，即上下文无关文法（也称巴科斯-诺尔范式，Backus-Naur Form），这里讨论的内容包含语言的派生、语法分析树、歧义性、描述操作符过程和组合性以及扩展巴科斯-诺尔范式。然后讨论用于描述程序设计语言的语法和静态语义的属性文法。最后介绍描述语义的三种形式方法/操作语义、公理语义和指称语义。由于语义描述方法所固有的复杂性，所以我们只能进行些简单的讨论。仅仅这三种语义描述方法中的任意一种，就能很容易地写成一本完整的书（而且确实有几位作者已经这样做了）。

3.1 概述

为程序设计语言提供一种精炼易懂的描述是一项重要的工作，却是保证语言成功所必需的工作。ALGOL 60和ALGOL 68语言是首先采用精炼的形式描述的语言；然而，由于这两种语言部分地采用了一种新标记方法的描述，从而使得人们难于理解。另外，这两种语言受欢迎的程度都不高。另一方面，一些语言具有许多略微不同的方言，这是由于语言定义比较简单，且定义不规范也不严谨。

在描述一种语言时，目标是要使各种各样的人都能够理解这种语言描述。这些人包括最初的评估者、实现者和产品用户。大多数的新程序设计语言在完成设计之前，都要提交给潜在的用户（经常在编用语言设计人员的组织里）推敲一段时期。这个反馈周期的成功与否，在很大程度上取决于语言描述的清晰程度。

显然，程序设计语言的实现人员一定要能够确定如何形成语言的表达式、语句和程序单元，以及它们被执行时所预期的效果。实现人员工作的难度部分地取决于语言描述的完整程度与精确程度。

最后，语言的使用人员必须能够通过查阅语言参考手册来确定如何编写软件系统。教科书和培训课程固然能够帮助学习语言，然而通常，语言手册是一种语言唯一的权威性信息来源。

学习程序设计语言也像学习自然语言一样，可以分为由词语法与语义两个部分。程序设计语言的语法是它的表达式、语句及程序单元的形式，而它的语义则是这些表达式、语句和程序单元的含义。举例来说，Java语言的while语句的语法是

```
while (<布尔_表达式>) <语句>
```

这种语句形式的语义是，如果布尔表达式的当前值为真，执行嵌入的语句。否则，执行while结构之后的语句。在这之后，控制又隐式地返回布尔表达式，以重复这个过程。

尽管为了讨论方便，常常将语法与语义分开来，但它们实质上是紧密相关的。在一种设计良好的程序设计语言中，语义应该直接紧跟语法；也就是说，语句形式应该能够清晰地提示出这条语句所要完成的任务。

描述语法要比描述语义容易，部分是因为已经存在着一种用于语法描述的精炼而通用的标记法，然而到目前为止，仍然没有为语义描述开发出类似的方法。

3.2 描述语法的普遍问题

无论是自然语言（如英语）还是人造语言（如Java），都是某种字符集中字符串的集合。语言的这种串就被称为句子或者语句。一种语言的语法规则说明，语言字符集合中的哪种字符串是属于该语言内的合法语句。例如，英文就有一套庞大而复杂的规则，用以说明英语句子的语法。与英文比较起来，即使是现编最大最复杂的程序设计语言，在语法上也都极为简单。

为了简单起见，程序设计语言的语法的形式描述通常不包括对于最低层次语法单元的描述。这些语法小单元被称为词素（lexeme）。词素的描述由词法说明给出，它与语言的语法描述是分开的。一种程序设计语言的词素包括它的字面值、操作符和特殊字，以及其他。我们也可以认为程序是一些词素的串，而不是字符的串。

语素分成组，例如变量名、方法和类等。在程序设计语言表中组称为标识符（identifier）。每一组用一个名字或标记表示。因此，一种语言的标记（token）是这种语言词素中的一个类别。例如，一个标识符是一个标记，它可以具有词素或者实例，如sum和total。在某些情况下，一个标记只能具有唯一词素。例如，算术操作符"+"的标记可以被命名为"plus op"，即"加法运算符"，它只有唯一可能的词素。考虑下面的Java语句示例：

117

```
index = 2 * count + 17;
```

这条语句的词素与标记分别是：

词 素	标 记	词 素	标 记
index	标识符	count	标识符
=	等于符号	+	加法运算符
2	整数字面值	17	整数字面值
*	乘法运算符	;	分号

本章中包括的语言描述的例子都非常简单，而且大部分都包括了词素描述。

3.2.1 语言识别器

通常，可以使用两种不同的方式来形式地定义语言：识别和生成（尽管这两个术语本身并不能给试图学习甚至使用程序设计语言的人员提供一种具有实际意义的定义）。假设我们有一种语言L，这种语言使用字母字符集Σ。为了使用识别的方法来形式地定义语言L，我们需要构造一个称为识别装置的机制，R。这种识别装置能够读入字母集Σ中的字符串。需要将R设计成为可以指示给定的输入字符串是否来自语言L。在实际效果上，R或者是接受，或者是拒绝这一条给定的字符串。这样的装置就如同过滤器一样，将正确与错误的句子区分开来。如果向R输入集合Σ中的任意字符串，R仅接受那些属于L中的字符串，那么R就是L的描述。出于实用的目的，大多数有用的语言是无限的，似乎这种识别过程会是冗长而低效的。然而在实际上，并不会使用识别装置来枚举语言中的所有句子。

编译器中的语法分析部分对于编译器所翻译的语言而言就是一个识别器。在充当这个角色时，识别器并不需要测试所有可能的字符串，以决定每一个串是否属于该语言。相反，它只需要决定所给的程序是否在语言之中。实际上，语法分析程序所要决定的，是给定的程序在语法上正确与否。关于语法分析程序（也称为语法分析器）将在第4章中讨论。

3.2.2 语言生成器

语言生成器是能用来产生语言中句子的一种装置。我们可以想象，这个生成器装有一个按

钮，每按一下，就产生一个语言句子。在按生成器的按钮时，会产生哪个特定句子并不可以预知。因而生成器似乎只是一种用途有限的装置，就像语言的描述器一样。然而较之识别器，人们还是更喜欢某些形式的生成器，因为它们更容易被人们阅读和理解。相比之下，编译器的语法检测部分（一种语言识别器）对程序人员而言就不是一种有用的语言描述，因为它只能被用于试探性模式（trial-and-error mode）。例如，使用一个编译器来确定某一特定语句语法的正确性时，程序人员只能输入一个试探性版本，以检测它能否被编译器所接受。而在使用生成器时，人们则通常可以通过将某一特定语句与生成器中的结构进行比较，就可以确定这条语句的语法是否正确。 118

同一种语言的形式生成器和识别装置有着紧密的联系。这是计算机科学中的基本发现之一，正是这项发现，产生了许多现在称为形式语言以及编译器设计的理论。我们将在下一节再返回来讨论生成器与识别器的关系。

3.3 描述语法的形式方法

本节将讨论形式语言的产生机制，这种通常被称为文法的机制，是用来描述程序设计语言的语法的。

3.3.1 巴科斯-诺尔范式及上下文无关文法

从20世纪50年代中期至后期，John Backus和Noam Chomsky分别于独立研究工作中发明了相同的标记法。这种标记法从此成为最广泛应用的程序设计语言语法描述的形式方法。

3.3.1.1 上下文无关文法

在20世纪50年代中期，著名语言学家Chomsky描述了用来定义四个类型的语言的四种生成装置或文法（Chomsky, 1956, 1959），其中的两类文法成为描述程序设计语言语法的有用工具，分别为上下文无关文法和正则文法。正则文法描述程序设计语言的标记（token）的形式。而除了极个别部分，上下文无关文法则可以描述整个程序设计语言的语法。因为Chomsky是一位语言学家，所以他主要感兴趣的是自然语言的理论性质。当时他对用于与计算机交流的人造语言没有兴趣。因而一直到后来，他的工作才被应用于程序设计语言。

3.3.1.2 巴科斯-诺尔范式的起源

在Chomsky的语言文法之后不久，美国计算机协会和应用数学与力学协会（ACM-GAMM）开始设计ALGOL 58语言。ACM-GAMM协会中的一位重要成员John Backus，在1959年的一个国际会议上发表了一篇描述ALGOL 58的论文，这篇论文具有里程碑式的意义（Backus, 1959），在其中，Backus介绍了一种用来说明程序设计语言语法的新的形式标记法。后来，Peter Naur又对这种新的标记法进行了少量的修改，用以描述ALGOL 60语言（Naur, 1960），修改过的语法描述方法就成为闻名的巴科斯-诺尔范式（Backus-Naur Form），或者能称为BNF。 119

BNF是一种非常自然的语法描述标记法。事实上，一种与BNF相类似的标记法曾经在公元前几百年就被旷尼尼（Panini）用来描述梵语（Sanskrit）语法（Ingerman, 1967）。

尽管当时计算机的使用人员对于在ALGOL 60语言报告中采用BNF还不能够接受，但是BNF很快就成为（而且至今仍然是）精炼描述程序设计语言语法的最普遍方法。

十分惊人的是，BNF与Chomsky的上下文无关语言的生成机制（称为上下文无关文法）几乎完全相同。在本章的余下部分，我们将简单地称上下文无关文法为文法，此外，术语“BNF”和“文法”将交替使用。

3.3.1.3 基本原理

元语言是一种用来描述另外一种语言的语言，而BNF就是程序设计语言的一种元语言。

BNF使用抽象语法结构。例如，可以将一条简单的Java赋值语句抽象地表示为<赋值语句>。（通常使用尖括号来限定抽象名称。）<赋值语句>的实际定义可以为

```
<赋值语句> → <变量> = <表达式>
```

箭头左边的符号称为“左手边”（left-hand side, LHS），它是正在定义的抽象；箭头右边的内容是LHS的定义，它被称为“右手边”（right-hand side, RHS）。RHS部分由标记、词素以及指向其他抽象的引用所组成。（实际上，标记也是抽象。）全部都在一起，这一条定义被称为一个规则，或者称为一个产生式。在刚刚给出的规则示例中，显然要在<赋值语句>定义具有意义之前，先定义抽象<变量>与<表达式>。

这一条特定规则说明，抽象<赋值语句>被定义成为抽象<变量>的一个实例，后面跟随着词素=，再后面接着抽象<表达式>的一个实例。下面是一个句子示例，这条句子的语法结构由上面的规则描述：

```
total = subtotal1 + subtotal2
```

在BNF的描述或文法中，抽象通常被称为非终结符（nonterminal symbol），或简称为非终结（nonterminal），而规则中的词素和标记则被称为终结符（terminal symbol），或者简称为终结（terminal）。一个BNF描述，或一种文法，仅仅是一个规则的集合。

非终结符可以具有两个或多个不同的定义，用以表示语言中的两种或多种可能的语法形式。
120 可以将多重定义写成一条规则，使用符号“|”将不同的定义分别开来。符号“|”的意义为逻辑或（OR）。例如，可以使用两条规则来描述Ada语言中的if语句：

```
<if_语句> → if <逻辑_表达式> then <语句>
<if_语句> → if <逻辑_表达式> then <语句> else <语句>
```

或者，仅仅使用一条规则

```
<if_语句> → if <逻辑_表达式> then <语句>
          | if <逻辑_表达式> then <语句> else <语句>
```

BNF虽然简单，但它具有充分的表达能力来描述几乎所有的程序设计语言的语法，尤其是，它能够描述相似结构的链表、不同结构出现的顺序、任意深度的嵌套结构，甚至隐含操作符优先级以及操作符结合性。

3.3.1.4 表描述

数学中使用省略符号（…）来书写长度可变的表，1, 2, …，就是一个例子。BNF中没有包括省略符号。因而需要使用其他的替代方法来描述程序设计语言中语法元素的表（例如，出现在数据声明语句里的标识符表）。最常用的替代方法是递归。如果一条规则的LHS出现在它的RHS之中，我们称这条规则是递归的。下面这条规则用以说明如何运用递归进行表的描述：

```
<标识符_表> → 标识符
            | 标识符, <标识符_表>
```

这条规则定义<标识符_表>为单个标记（标识符）或者标识符，然后紧跟逗号，后面再跟另一个<标识符_表>的实例。在本章的余下部分还会在许多文法例子里运用递归进行表的描述。

3.3.1.5 文法与派生

文法是定义语言的生成装置。语言的句子通过应用一系列语法规则生成，这个过程起始于

文法中的一个特殊的非终结符，称为起始符（start symbol）。句子的生成则被称为派生。在一个完整语言的文法中，起始符代表了一个完整的程序，通常被命名为<程序>。我们用例3.1显示的简单文法来说明派生。

例3.1 一个简单语言的文法

```
<程序> → begin <语句_表> end
<语句_表> → <语句>
          | <语句>; <语句表>
<语句> → <变量> = <表达式>
<变量> → A | B | C
<表达式> → <变量> + <变量>
          | <变量> - <变量>
          | <变量>
```

例3.1中的语言只具有一种语句形式，即赋值语句。它的程序包括特殊字begin，后面跟随一列由分号隔开的语句，再接着特殊字end。它的表达式为单个变量，或者是被“+”或“-”操作符分开的两个变量。这个语言中的变量名只有A，B和C。

下面是这个语言的一个派生程序：

```
<程序> => begin <语句_表> end
       => begin <语句>; <语句_表> end
       => begin <变量> = <表达式>; <语句_表> end
       => begin A = <表达式>; <语句_表> end
       => begin A = <变量> + <变量>; <语句_表> end
       => begin A = B + <变量>; <语句_表> end
       => begin A = B + C; <语句_表> end
       => begin A = B - C; <语句> end
       => begin A = B - C; <变量> = <表达式> end
       => begin A = B + C; B = <表达式> end
       => begin A = B + C; B = <变量> end
       => begin A = B + C; B = C end
```

在这个<程序>例子里，这个派生出程序的所有派生一样，始于起始符。符号“=>”读为“派生”。每一个后续的串都派生自它前面的串，通过使用非终结符的一个定义值来代替前面串中的非终结符，从而完成了这种派生。派生进程中的每一个串，包括<程序>，都被称为句子范式。

在上面的派生进程中，被替换的非终结符总是前面句子范式中最左面的那个非终结符。使用以上顺序进行替换的派生被称为最左派生。派生将一直持续到在句子范式里再没有非终结符为止。最后的句子范式中仅仅包含了终结符或者词素，这就是被生成的句子。

除了最左派生以外，也可以有最右派生，或者既非最左又非最右的派生。派生的顺序对于由一种文法生成的语言并没有影响。

通过选择规则中不同的RHS部分，并以此来替换派生中的非终结符，能够生成语言中不同的句子。穷尽所有选择的组合，就能够产生完整的语言。像大多数其他语言一样，这样的语言是无限的，因此不能够期望在有限的时间内生成语言中的所有句子。

例3.2是另一种类型的例子，它给出了典型程序设计语言的一部分文法。

例3.2中的文法描述了几种赋值语句，这些语句的右边是具有乘法和加法操作符以及括号的算术表达式。例如，语句

```
A = B * (A + C)
```

例3.2 简单赋值语句的文法

```
<赋值语句> → <标识符> = <表达式>
<标识符> → A | B | C
<表达式> → <标识符> + <表达式>
         | <标识符> * <表达式>
         | ( <表达式> )
         | <标识符>
```

是由最左派生生成的：

```
<赋值语句> => <标识符> = <表达式>
           => A = <表达式>
           => A = <标识符> * <表达式>
           => A = B * <表达式>
           => A = B * ( <表达式> )
           => A = B * ( <标识符> + <表达式> )
           => A = B * ( A + <表达式> )
           => A = B * ( A + <标识符> )
           => A = B * ( A + C )
```

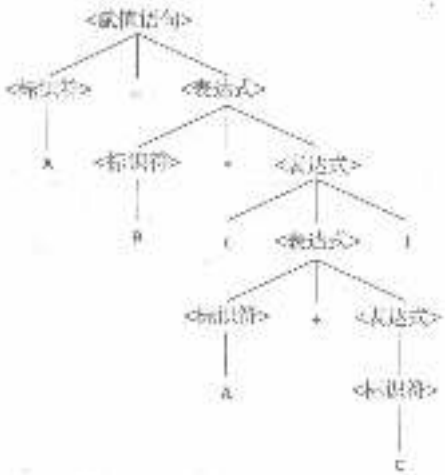


图3-1 简单语句A = B * (A + C)的一棵语法分析树

3.3.1.6 语法分析树

最吸引人的一种语法特征，是在这种语法中定义的对语言句子层次语法结构的自然描述，这种层次结构就被称为语法分析树。例如，图3-1中的语法分析树显示了在前面派生的赋值语句的结构。

123

语法分析树的每一个内部节点都由一个非终结符来标记，而它的每一个叶节点都由一个终结符来标记。语法分析树的每一棵子树都描述语句中抽象的一个实例。

3.3.1.7 歧义性

如果由一条文法生成的句子具有两种或者多种不同的语法分析树，就称这条文法是歧义的。考虑例3.3中显示的文法，它是例3.2中文法的一个变体。

例3.3 简单赋值语句的一条歧义性文法

```
<赋值语句> → <标识符> = <表达式>
<标识符> → A | B | C
<表达式> → <表达式> + <表达式>
         | <表达式> * <表达式>
         | ( <表达式> )
         | <标识符>
```

例3.3中的文法是歧义的，因为句子

```
A = B * C + A
```

具有如图3-2所示的两棵不同的语法分析树。这说明在语法结构上，例3.3中的文法比例3.2中的文法稍欠严谨，导致了歧义性。例3.3中的文法不是仅仅允许表达式的语法分析树从右边生长，而是允许它既从左边又从右边生长。

121

语言结构上的语法歧义性是一个问题，这为编译器通常将语法形式作为这些结构的语义的基础。尤其是，编译器是按照语句的语法分析树来决定应该为这类语句产生什么样的代码。如

某一种语言结构具有多个语法分析树，那么就不能够唯一地确定这种结构的语义。关于这个问题，将在下面小节的两个特别的例子中进行讨论。

有时，语法有一些其他特征来决定其是否是歧义的，[9]这很有用。这些特征包括：1）是否语法产生一个句子并带有超过一个最左派生；2）是否语法产生一个句子并带有超过一个最右派生。

一些语法分析的算法可以基于歧义的文法。当这样一个语法分析器遇到一种歧义的结构时，它将使用设计人员所提供的非文法信息来构造正确的语法分析树。在许多情形下，歧义语法能重写成非歧义的，但是仍会产生需要的语言。

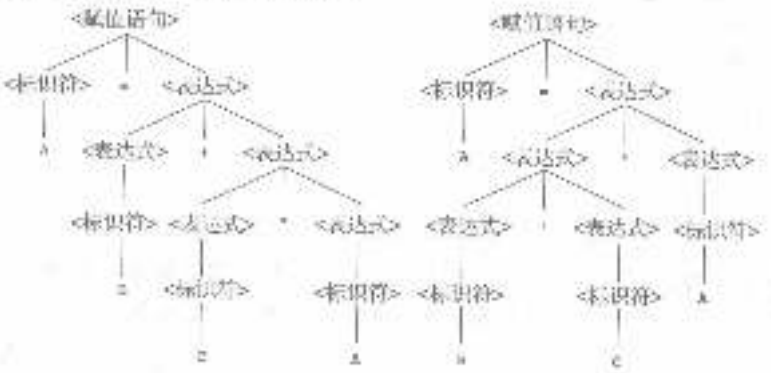


图3-2 同一条语句A = B + C * A的两棵不同的分析树

3.3.1.8 操作符优先级

当一个表达式包含两个不同的操作符时，例如，x+y*z，一个明显的语义问题是两个操作符的评估顺序（是加后乘，还是乘后再加）。这个语义问题能通过赋予操作符不同的优先级来解决。例如，如*能被赋予比+更高的优先级（由语言设计人员来实现），那么将先做乘法，而不管在表达式中两个操作符出现的顺序。

如前所述，文法能够描述一定的语法结构，而通过这种结构的语法分析树可以确定语法结构的部分语义。特别地，如果一个算术表达式中的操作符生成于分析树的较低层上（因此该操作符必须先被求值），这就表明，它比在分析树的较高层上生成的操作符有优先级。例如，图3.2中的第一棵语法分析树中的乘法操作符产生于分析树的较低层，这表示它比表达式中的加法操作符具有优先级；然而在第二棵语法分析树上却恰恰相反。显而易见，这两棵分析树所给出的关于优先级的信息正好相互矛盾。

请注意，尽管例3.2中的文法不是歧义的，但是这个例子中的操作符优先级却不是通常所应有的顺序。在这个文法中，一棵具有多操作符句子的语法分析树，无论涉及了哪些操作符，表达式中最右边的操作符总是占据语法分析树上最低的位置，而其他操作符随着它在表达式中位置的从右到左，它在语法分析树上的位置也逐渐移往高处。例如，在表达式A+B*C中，*在树中最低的位置，表明最先处理它。然而，在表达式A*B+C中，+在树中最低的位置，表明最先处理+运算。

125

无论这些操作符在表达式中出现的次序如何，这里正在讨论的简单表达式的文法都能够写成非歧义的，也能够对+和*操作符指定一种前后一致的优先级。只要对具有不同优先级的操作符的操作数使用分开的非终结符，就能指定正确的顺序。这需要额外的非终结符和一些新规则。

[9] 正式，数学上不可能对任意文法（arbitrary grammar）是歧义的。

相对在+和*操作符又对操作数使用<表达式>，我们可以使用3个非终结符来表示操作数，它们能允许文法强制不同的操作符对应语义树的不同层次。如果<表达式>是表达式的根符号，通过让<表达式>只直接产生+操作符使用新的非终结符<项>作为+的右操作数，强制使+位于语义树的顶端。接着，使用<项>作为左操作数和新的非终结符<因子>作为右操作数，来定义<项>产生*操作符。现在，*总是位于语义树的低层中，简单地因为在每次派生中它比+更远离开始符。例3.4中的文法就是这样一种文法。

例3.4 表达式的非歧义性文法

```
<赋值语句> → <标识符> = <表达式>
<标识符> → A | B | C
<表达式> → <表达式> + <项>
          | <项>
<项> → <项> * <因子>
       | <因子>
<因子> → ( <表达式> )
       | <标识符>
```

126

例3.4中的文法与例3.2和例3.3中的文法都产生同一种语言。但不同的是，这条文法是非歧义的，而且指定了通常的乘法与加法操作符的优先级顺序。下面运用例3.4中的文法派生句子 A = B + C * A：

```
<赋值语句> => <标识符> = <表达式>
           => A = <表达式>
           => A = <表达式> + <项>
           => A = <项> + <项>
           => A = <因子> + <项>
           => A = <标识符> + <项>
           => A = B + <项>
           => A = B + <项> * <因子>
           => A = B + <因子> * <因子>
           => A = B + <标识符> * <因子>
           => A = B + C * <因子>
           => A = B + C * <标识符>
           => A = B + C * A
```

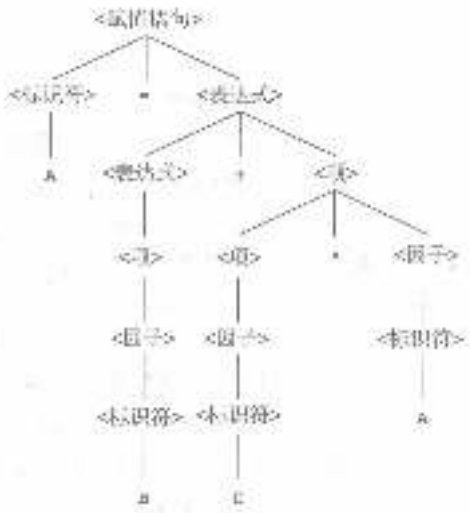


图3-3 语句A = B + C * A运用非歧义文法的唯一语法分析树

图3-3显示了运用例3.4中的文法生成这一条句子的唯一语法分析树。

语法分析树与派生之间的联系非常紧密，很容易从它们两者中的任意一个构造出另外的一个。非歧义的文法的每一个派生只有唯一的语法分析树，尽管这棵树能够被不同的派生所表示。例如，句子A = B + C * A在如下所示的派生就不同于在前面给出的同一条语句的派生。这里是一个最右派生，而在前面给出的是最左派生。然而这两种派生都由同一棵语法分析树来表示。

```
<赋值语句> => <标识符> = <表达式>
           => <标识符> = <表达式> + <项>
           => <标识符> = <表达式> + <项> * <因子>
           => <标识符> = <表达式> + <项> * <标识符>
           => <标识符> = <表达式> + <项> * A
           => <标识符> = <表达式> + <因子> * A
```

```
=> <标识符> = <表达式> + <标识符> * A
=> <标识符> = <表达式> + C * A
=> <标识符> = <项> + C * A
=> <标识符> = <因子> + C * A
=> <标识符> = <标识符> + C * A
=> <标识符> = B + C * A
=> A = B + C * A
```

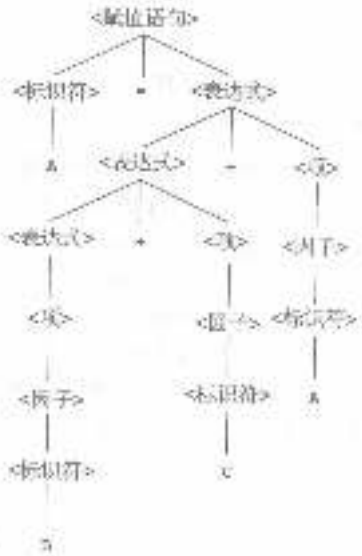


图3-4 语句A = B + C * A的语法分析树，突出加法结合性

127

3.3.1.9 操作符的结合性

当一个表达式包括同一优先级的两个操作符（如*和/），例如A/B*C时，需要一个语义规则来指定优先级。[⊖]这个规则叫结合性（associativity）。正如优先级的例子，表达式的文法可能恰当地隐含了操作符的结合性。考虑下面的一个赋值语句的例子：

```
A = B + C + A
```

图3-4显示了由例3.4中的文法所定义的这条句子的语法分析树。

图3-4中的这棵语法分析树显示出左边的加法操作符比右边的加法操作符层次低。对于必须左结合的加法，这是正确的顺序：左结合加法是典型的。但在大多数计算机的情形中，加法的结合性是任意的。在数学中的加法是结合的，这意味着左结合与右结合的运算顺序等价，即 (A + B) + C = A + (B + C)。计算机中浮点数加法不是结合的。例如，假设要为浮点数值保存七位数字的精确度。考虑11个数字的相加问题，其中的一个数为10^7，而其他的10个数为1。如果将每一个小数值的数（即1）加到大数值的数字上，一次加一个，这对数值没有影响，因为小数值的数仅仅出现在大数值数的第8位数上。然而，如果先将10个小数值的数加在一起，然后再将这个结果加入大数值的数，这个需要保存七位数字精确度的数值结果就成了$1.000\ 000\ 1 \times 10^7$。至于减法与除法，则无论在数学中还是在计算机方面都不是结合的。所以，正确的结合性对于包含减法或除法的表达式尤其至关紧要。

128

当有一条文法规则，它的LHS也出现在它的RHS的起始位置，这条规则就被称为左递归的。左递归说明左结合性。例如，例3.4中的文法规则的左递归导致了加法与乘法这两种操作都为左结合的。不幸的是，左递归不允许使用一些重要的语法分析算法。当使用这样的算法时，必须修改文法以删除左递归。而且，这不允许文法精确指定确定的操作是左结合的。幸运的是，左结合性能够由编译程序指定，尽管文法中没有指定。

在大多数提供乘幂运算的语言之中，乘幂操作符是右结合的，可以使用右递归来说明右结合。如果一条文法规则的LHS出现在RHS的最右端，则该条文法规则是右递归的。例如，规则

129

```
<因子> → <幂> ** <因子>
       | <幂>
<幂>  → ( <表达式> )
       | <标识符>
```

可以被用来描述右结合的操作符的乘幂。

⊖ 在一个表达式中重复出现同一个操作符会有相同问题，例如，A/B/C。

3.3.1.10 if-then-else的非歧义性文法

在3.3.1.3节里，曾给出一种关于特殊形式的if-then-else语句的BNF规则，这里再次给出如下：

```
<if_语句> → if <逻辑_表达式> then <语句>
          | if <逻辑_表达式> then <语句> else <语句>
```

如果我们也有<语句>→<if_语句>，那么这条文法就是歧义的。能够最简单地说明这种歧义性的一个句型为：

```
if <逻辑_表达式> then if <逻辑_表达式> then <语句> else <语句>
```

图3.5中的两棵语法分析树显示了这个句型的歧义性。考虑这个构造的下面的例子：

```
if (done == true)
  then if (denom == 0)
    then quotient = 0;
    else quotient = num / denom;
```

问题是，在图3-5中上面的语义树用作翻译的主要部分，当done不为真时，else子句将执行，但这可能不是构造作者的原本意图。我们将在第8章里研究与这种else结合性问题有关的一些实际问题。

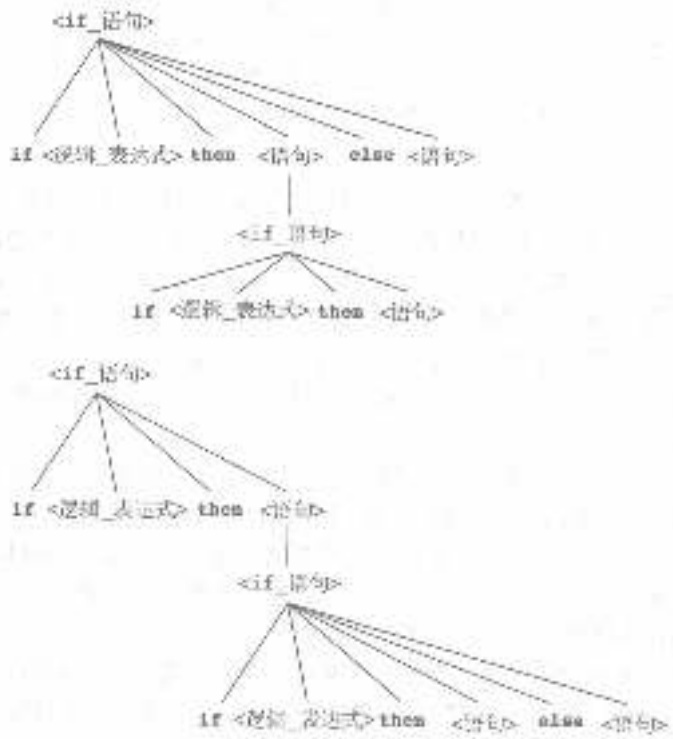


图3-5 同一句型的两棵不同的语法分析树

我们现在将开发描述这种if语句的一种非歧义性文法。大多数语言关于if结构的规则是，当有else子句出现时，else与前面最接近的没有被匹配的then相匹配。因此，在一个then以及与之匹配的else之间，不能有一个没有else的if语句。因而针对这种情形，就必须区别已经被匹配了的语句和没有被匹配的语句。这里没有被匹配的语句只能是else的数目少于if的数目，而所有其他语句都已经被匹配好了。在上述文法中存在的问题是，它以等同的语法重

要性来对待所有的语句；也就是说，似乎所有的语句全都被匹配好了。

要反映出不同种类的语句，必须使用不同的抽象或不同的非终结符。下面列出的是基于这种思想的一个非歧义文法： 130

```
<语句> → <匹配_语句> | <未匹配_语句>
<匹配_语句> → if <逻辑_表达式> then <匹配_语句> else <匹配_语句>
              任何非 if 语句
<未匹配_语句> → if <逻辑_表达式> then <语句>
              | if <逻辑_表达式> then <匹配_语句> else <未匹配_语句>
```

运用这条文法，下面的句形就只存在唯一可能的语法分析树：

```
if <逻辑_表达式> then if <逻辑_表达式> then <语句> else <语句>
```

3.3.2 扩展的BNF

因为BNF在某些方面存在着些微不方便之处，因此人们从几个方面对它进行了扩展。大多数扩展的BNF版本被称为“扩展的BNF”，或者简称为“EBNF”——尽管这些扩展的版本不尽完全相同。这些扩展性工作并不提高BNF的描述能力，它们仅仅增强了BNF的可读性与可写性。 131

通常，EBNF的各种不同版本中包括三项扩展。第一项扩展是说明RHS中的可选择部分，并用方括号将这些部分括起来。例如，可以将C语言的一条选择语句描述为

```
<选择> → if ( <表达式> ) <语句> [ else <语句> ]
```

如果不使用方括号，这条语句的语法描述就会需要两条规则：

```
<选择> → if ( <表达式> ) <语句>
         if ( <表达式> ) <语句> else <语句>
```

第二项扩展，是在RHS中使用花括号，表明可以无限制地重复花括号包含的部分，或者就是完全没有。这项扩展允许仅使用一条规则就可以构造链表，而不是使用递归和两条规则。例如，可以使用下列规则来描述由逗号分隔的标识符表：

```
<标识符_表> → <标识符> { , <标识符> }
```

这里通过一种隐式重复替代了递归：可以将包含在花括号之内的部分重复任意次数。

第三项常用的扩展是处理多重选择的选项。当必须从一组元素中选择其中的单个元素时，就在圆括号之内放入候选项，并用OR操作符的记号“|”将这些项分开。例如

```
<项> → <项> (* | / | % ) <因子>
```

如果没有这项扩展，就需要三条BNF规则来描述上述的结构。在EBNF扩展中使用的方括号、花括号和圆括号是元符号；这意味着它们只是标记的工具，而不是用以帮助描述的、语法实体中的终结符。有时，在所描述的语言中这些元符号也被用作终结符，表示终结符的实例时，可以为其添加下划线或者使用引号。

BNF的规则

```
< 表达式 > → < 表达式 > + < 项 >
```

清楚地说明了（事实上是强制性地）加法操作符“+”是左结合的，然而EBNF的这个版本

```
< 表达式 > → < 项 > { + < 项 > }
```

却没有隐含结合性的方向。在一种基于EBNF表达式文法的语法分析器中，关于这个问题的 132
解决是通过设计语法分析过程来强制施行正确的结合性。我们将在第4章进一步讨论这个问题。

例3.5 表达式文法的BNF与EBNF版本

```
BNF 版本: < 表达式 > → < 表达式 > + < 项 >
                      < 表达式 > - < 项 >
                      < 项 >
          < 项 >   → < 项 > * < 因子 >
                   | < 项 > / < 因子 >
                   | < 因子 >
          < 因子 > → < 表达式 > ** < 因子 >
                   | < 表达式 >
          < 表达式 > → (< 表达式 >)
                     | <标识符>
EBNF 版本: < 表达式 > → < 项 > { (+ | -) < 项 > }
          < 项 >    → < 因子 > { (* | /) < 因子 > }
          < 因子 >  → < 表达式 > [ ** < 表达式 > ]
          < 表达式 > → (< 表达式 >)
                     | <标识符>
```

某些EBNF的版本允许将一个数值上标附于花括号的右半，用以说明所包括部分可重复次数的上限。此外，一些版本还使用加号（+）上标来标明是一次或多次重复。例如，

```
< 复合语句 > → begin < 语句 > { ; < 语句 > } end
```

和

```
< 复合语句 > → begin { < 语句 > ; } end
```

为等价的。

近年来，BNF和EBNF的版本中出现了好些变异，其中包括：

- 使用冒号“:”来代替箭头“→”，并且将RHS放置于下一行。
- 不使用竖线符“|”来分开可能替代的RHS，而是将这些RHS放置于不同的行。
- 在表示可选择的项目时，不使用方括号，而是使用下标“*opt*”，例如，

```
<函数_声明符> → <简单名称> (<形式参数_表opt>)
```

- 使用“one of”（其中之一）来表示选择，而不是在一组括号包括的元素中使用竖线符“|”。例如，

133 赋值_操作符 → one of = *= /= %= += -= <<= >>= &= ^= |=

EBNF有一种标准，ISO/IEC 14977：1996（ISO/IEC, 1996），但是很少被使用。这个标准在规则中使用等号“=”来代替箭头“→”，使用分号“;”来终止每一个RHS，并且在所有终止符上都要求有引号，而且还指定了一系列的其他标志规则。

3.3.3 文法与识别器

在本章的前面我们提出，在特定语言的生成装置与识别装置之间有着紧密的联系。事实上，给定一个上下文无关文法，就可以使用算法方式来构造由这条文法产生的语言识别器。许多构造识别器的软件系统已经被开发出来了。这样的系统可以很快产生一种新语言编译器的语法分析部分，因而相当有价值。最早的语法分析器的生成器之一，是被称为yacc的系统（意为“另一种编译器的编译器”）（Johnson, 1975）。现在已经有了很多这样的系统。

3.4 属性文法

属性文法是一种用来描述更多程序设计语言结构的机制。而这些结构是上下文无关文法所不能描述的。属性文法是上下文无关文法的一种扩展，这种扩展能够方便地描述某些语言规则，例如类型兼容性，在我们正式定义属性文法的形式之前，必须澄清静态语义的概念。

3.4.1 静态语义

程序设计语言结构有一些特征是BNF难以描述的，还有一些特征则是BNF不能够描述的。BNF不能够说明的语言规则的一个例子是类型兼容性规则。例如在Java语言中，不能够把浮点数值赋给一个整数类型的变量，尽管反方向的赋值是合法的。虽然这种限制性能够在BNF中给予说明，但是需要额外增加一些非终结符以及规则，如果Java语言中的所有类型规则都必须使用BNF来说明，这种文法就会过于庞大，以致根本无法使用。这是因为文法的大小决定着语法分析器的大小。

BNF不能描述的语言规则的一个例子是这样一条通用规则：在引用变量之前必须先声明该变量。可以证明，这条规则不能够使用BNF来说明。

列举在这个问题之中的语言规则类型被称为静态语义规则。一种语言的静态语义仅仅于运行时与程序意义间接相关；相反，它与程序的合法形式（语法而非语义）有关。一种语言的多种静态语义规则说明类型的限制。之所以被命名为静态语义，是因为检测这些规范所必需的分析能够在编译时进行。

由于使用BNF描述静态语义存在一些问题，所以人们发明出了各种功能更为强大的机制，其中的一种机制便是属性文法，它是由Knuth（Knuth, 1968a）设计的，用来描述程序的语法以及程序的静态语义。

历史注释

属性文法已经被用于各种广泛的应用之中。它们被用于提供程序设计语言语法以及静态语义的完整描述（Watt, 1979）。它们也被用于形式地定义一种可以被人到编译生成系统中的语言（Farrow, 1982）。它们还被作为许多语法程序的编辑系统的基础（Teitelbaum和Reps, 1981；Fischer et al., 1984）。此外，属性文法还已经被应用于自然语言处理系统中（Correa, 1992）。

[135]

属性文法是用来描述及检测程序静态语义规则正确性的一种形式方法。尽管在编译器的设计中，并不是总是形式地使用这种文法，但至少每一个编译器都非形式化地采用了属性文法的基本概念（参阅Aho et al., 1986, 第5章）。

动态语义将在3.5节中讨论。

3.4.2 基本概念

属性文法是一些增加了属性、属性计算函数以及谓词函数的上下文无关文法。与文法符号相关联的属性，就它能够被赋值的意义而言，与变量相类似。属性计算函数有时也被称为语义函数，与文法规则相关联。通常是使用它们来说明如何计算属性值。谓词函数用于说明语言的一些语法规则以及静态语义规则，与文法规则相关联。

在我们形式地定义了属性文法并讨论了一个例子之后，这些概念将会更为清晰。

3.4.3 属性文法定义

属性文法是一种具有下列附加特性的文法：

- 与每一个文法符号X相关联的是一个属性集合A(X)。集合A(X)由两个互不相交的集合S(X)和I(X)组成。这两个集合分别被称为合成属性和继承属性。合成属性用于在语法分析树中往上传递语义信息，而继承属性则用于穿过语法分析树往下传递语义信息。
- 与每一条文法规则相关联的是一个语义函数集合，以及一个有可能为空的谓词函数集合。这些函数作用于与符号文法规则相关的属性之上。对于一条规则$X_0 \to X_1 \cdots X_n$，其中X_i的合成属性由形式为$S(X_i) = f(A(X_1), \cdots, A(X_n))$的语义函数来计算。因此语法分析树上一个节点的合成属性值仅仅取决于这个节点的子节点属性的值。符号X_i，$1 \le i \le n$（上述规则中的元素）的继承属性由形式为$I(X_i) = f(A(X_0), \cdots, A(X_{i-1}))$的语义函数来计算。因而一个分析树节点的继承属性值取决于这个节点的父节点以及兄弟节点的属性值。注意，为了避免迂回，继承属性通常被限制为函数形式$I(X_i) = f(A(X_0), \cdots, A(X_{i-1}))$。这样就防止了继承属性依赖于它的自身，或者依赖于语法分析树上其右边节点的属性。
- 谓词函数具有布尔表达式的形式，它是属性集合$\{A(X_0), \cdots, A(X_n)\}$和一套文字属性值的并集。属性文法所允许的唯一派生是当与每一个非终结符相关的每一条谓词都为真时所产生的派生。为假的谓词函数值表示违反了语言的语法规则或者语言的静态语义规则。

135

一个属性文法的语法分析树是基于属性文法所包含BNF的语法分析树，其中每一个节点都附上了有可能为空的属性值集合，如果已经计算了一棵语法分析树中所有的属性值，这棵树就被称为**完全属性的**。在编译器构造了一棵完整非属性语法分析树之后，通常可以方便地认为它的所有属性值都已经被计算了，虽然实际上并不总是这样。

3.4.4 内在属性

内在属性是语法分析树叶节点的合成属性，其值决定于分析树之外。例如，程序中变量实例的类型可以是来自储存变量名与类型的符号表，而这个符号表的内容又由前面的声明语句来决定。假设在初始时构造了一棵非属性语法分析树，但又要求有属性值，此时具有值的属性就只能是叶节点的内在属性。只要给出了一棵语法分析树上的内在属性值，运用语义函数就能够计算出其他所有的属性值。

3.4.5 属性文法的例子

如何使用属性文法来描述静态语义，要说明这个问题，一个非常简单的例子是下面用来描述一条语法规则的属性文法录。这条被描述的语法规则为：在Ada过程中，end后面的名字必须与过程名相匹配。<过程名>的字符串属性，标记为<过程名>.字符串，这是词法分析器所发现的实际字符串。请注意，当同一个非终结符多次出现于属性文法的一条语法规则中时，就要使用方括号附以下标来加以区别。这种下标以及方括号都不是被描述语言的部分。

136

```
语法规则：<过程_定义> → procedure <过程_名>[1]
                          <过程_体> end <过程_名>[2];
谓词：<过程_名>[1].字符串 == <过程_名>[2].字符串
```

这个例子中的谓词规则说明，子程序首部的<过程_名>非终结符的名字串属性必须与子程序尾部的<过程_名>非终结符的名字串属性相匹配。

下面，我们来考虑一个属性文法的较大例子，这个例子介绍如何使用属性文法来检查一条简单赋值语句的类型规则。下面是这条赋值语句的语法与语义：变量名只有A、B和C。赋值语句右边可以是一个变量或者一个变量与另一个变量相加的表达式形式。变量可以为这两种类型中的一种，整数或实数。当有两个变量都位于赋值语句的右面时，它们不必是相同的类型。当

操作数的类型不相同时，表达式的类型总是实数。当操作数的类型相同时，表达式的类型就是操作数的类型。赋值语句左面的类型必须与右面的类型相匹配，所以尽管右面的操作数可以为混合类型，但是只有当LHS的类型与RHS求值结果的类型相同，这一条赋值语句才成立。下面的属性文法就说明这些语义规则。

我们的属性文法例子的语法部分是

```
<赋值语句> → <变量> = <表达式>
<表达式> → <变量> + <变量>
         | <变量>
<变量> → A | B | C
```

在这个属性文法例子中，非终结符的属性由下面的段落给予描述：

- 实际_类型——与非终结符<变量>和<表达式>相关联的一种合成属性。这种合成属性用来储存变量或表达式的实际类型，即为整型或实型。对于变量，实际类型是内在的。而对于表达式，它取决于非终结符<表达式>的子节点的实际类型。
- 期望_类型——与非终结符<表达式>相关联的一种继承属性。这种继承属性被用来储存表达式所期望的类型，或者是整型或者是实型，由赋值语句左边的变量类型所决定。 137

完整的属性文法在下面的例3.6中给出。

例3.6 简单赋值语句的一种属性文法

```
1. 语法规则：<赋值_语句> → <变量> = <表达式>
   语义规则：<表达式>.期望_类型 ← <变量>.实际_类型
2. 语法规则：<表达式> → <变量>[2] + <变量>[3]
   语义规则：<表达式>.实际_类型 ←
                    if ( <变量>[2].实际_类型 = 整型 ) and
                            ( <变量>[3].实际_类型 = 整型 )
                       then 整型
                    else 实型
                    end if
   语句：<表达式>.实际_类型 = = <表达式>.期望_类型
3. 语法规则：<表达式> → <变量>
   语义规则：<表达式>.实际_类型 ← <变量>.实际_类型
   语句：<表达式>.实际_类型 = = <表达式>.期望_类型
4. 语法规则：<变量> → A | B | C
   语义规则：<变量>.实际_类型 ← 查表( <变量>.字符串 )
*表函数用于左括号表中查询一个给定为变量名，并返回这个变量的类型
```

图3-6显示了一个语法分析树的例子，它是由例3.6中的文法产生的句子A = A + B的语法分析树。与上面文法一样，将包括数字的方括号附加在分析树中重复节点的标号之后，这样它们就能够无歧义地被引用。 138

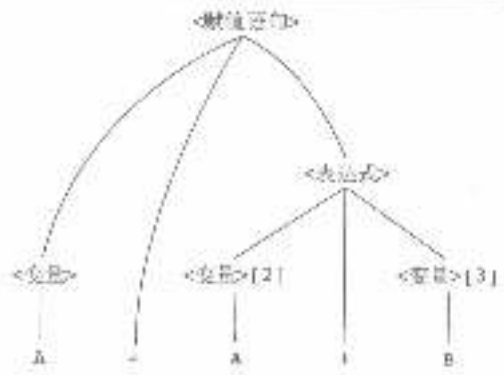


图3-6 A = A + B的一种语法分析树

3.4.6 计算属性值

现在让我们考虑使用属性装饰语法分析树的过程。如果所有属性都是被继承的，这个过程可以完全按照自上而下的顺序，即从根到叶来进行。

但是，如果所有属性都是被合成的，这个过程可以完全按照自底而上的顺序，即从叶到根来进行。因为我们的文法具有两种属性：合成属性与继承属性，所以求值的过程就不能仅按照任何一个单一方向。下面所列举的是一个属性求值的过程，使用的求值顺序是计算这些属性值可能采用的顺序。

1. <变量>.实际_类型 ← 查表(A) (规则 4)
2. <表达式>.期望_类型 ← <变量>.实际_类型 (规则 1)
3. <变量>[2].实际_类型 ← 查表(A) (规则 4)
   <变量>[3].实际_类型 ← 查表(B) (规则 4)
4. <表达式>.实际_类型 ← 算术结果类型 (规则 2)
5. <表达式>.期望_类型 == <表达式>.实际_类型为真 或 为假 (规则 2)

图3-7中的树显示了图3-6例子中的属性值流。其中的实线代表语法分析树，虚线表示树中的属性流。

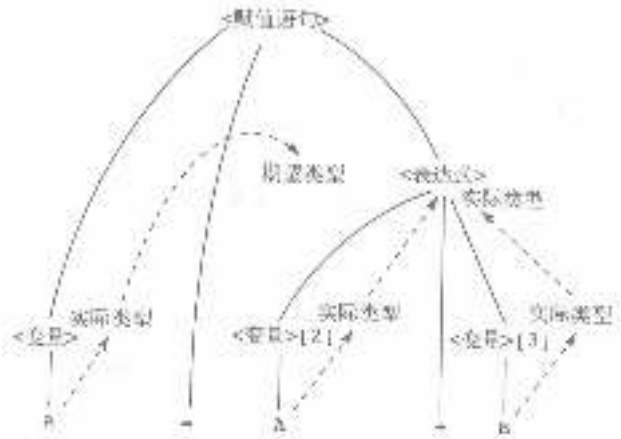


图3-7 树中的属性流

图3-8中的树显示节点上的最终属性值。在这个例子中，A被定义为实型，B为整型。

在使用属性文法的一般情况下，确定属性求值的顺序是一个复杂的问题，通常需要构造一个依赖图，用以显示所有属性之间的依赖关系。 139

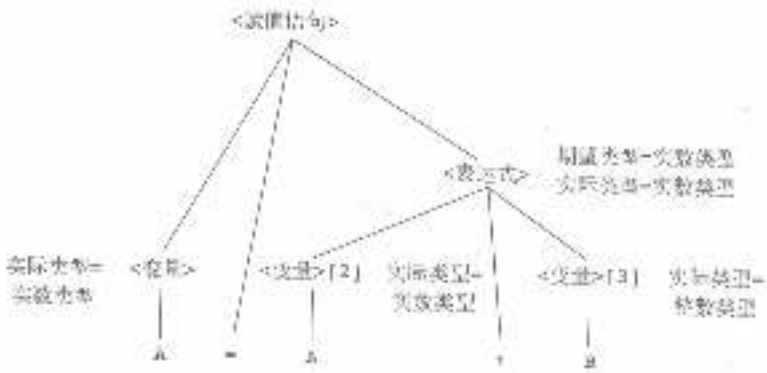


图3-8 一棵完全属性化的语法分析树

3.4.7 评估

检测语言的静态语义规则是所有编译器的一个重要部分，即使一个编写编译器的人员从来没有听说过属性文法，他或她也终将需要使用属性文法的基本概念来设计编译器中检测静态语义规则的部分。

在使用属性文法来描述当代程序设计语言中所有的语法以及静态语义时，主要的困难是文法本身的庞大与复杂性。一种完整的程序设计语言所需要的大量属性和语义规则，使得这种文法难写又难读。此外，从一棵大的语法分析树求取属性值的代价很高。另一方面，不十分形式化的属性文法是编译器编写人员普遍使用的功能强大的工具，这部分人员对产生编译器的过程的兴趣大大超过了对于形式主义的兴趣。

3.5 描述程序的意义：动态语义

我们现在转到描述表达式、语句以及程序单元的动态语义或意义。这是一项困难的工作。鉴于执行标记法的功能及自然特征，描述语法是一件相对简单的事情。另一方面，人们还没有发明出一种被普遍接受的动态语义标记法。在本节中，我们将简略地描述几种已经开发出来的方法。而在本节的余下部分，当我们使用术语“语义”的时候，指的就是动态语义，我们将使用“静态语义”来表示静态语义。

人们之所以关注描述语义，可能有几种不同的理由。首先，程序人员显然需要精确地知道语言语句的用途。他们通常通过阅读语言手册中的英语解释来发现语句的用途。但这种解释常常不够严密也不完整。典型地，编译器的编写人员也是通过英语描述来确定语言的意义。之所以使用这些非形式的描述，是由于形式化的语义描述中所存在的复杂性。一个研究目标显然是，让程序开发人员以及编译器的编写人员都能够使用一种语义形式。 140

在第15章将要描述的一种函数式程序设计语言Scheme方言就是仅有的少数几种语言的定义包括了一种形式语义描述的程序设计语言之一。然而Scheme方言所使用的方法将不在本章描述，因为本章的注意力放在命令式语言的方法上。

3.5.1 操作语义

操作语义的思想是，通过把语句或语言翻译成一种更容易理解的语言，从而描述语句或程序的意义。

操作语义的使用有不同的层次。在高级层次，关注的是完整程序的最终执行结果，有时称之为自然操作语义。在低级层次，通过检查语句的翻译版本，操作语义能用来决定单一语句的精确含义，有时称之为结构操作语义。

3.5.1.1 基本过程

创建一种语言的操作语义描述的第一步是设计一种合适的中间语言，该语言的最基本特征是清晰明确，该中间语言的每一步构造都有明白的、非歧义的意义。这种语言是中级语言，因为机器语言太低级以致于很难理解，而高级语言也显然是不合适的。如果语义描述能用作自然操作语义，那么虚拟机（解释器）肯定能构建中间语言，虚拟机能用来执行单个语句、代码段或整个程序。如果语义描述用于结构操作语义，虚拟机就是不必要的。

操作语义的基本过程其实很常见。事实上，这种概念经常被用于程序设计教科书以及程序设计语言参考手册之中。例如，C语言中for结构的语义就可以使用非常简单的指令进行描述，如

```
        C语句                          操作语义
for (expr1; expr2; expr3) {        expr1;
  ...                          loop: if expr2 == 0 goto out
}                                  ...
                                   expr3;
                                   goto loop
                                 out: ...
```

这种描述的人类读者就是一台虚拟计算机，并且我们假定这台虚拟计算机能够正确地“执行”定义中的指令，并且能够识别“执行”之后产生的效果。

141

作为使用低层次语言描述操作语义的一个例子，考虑下列语句。这一组语句足以描述一种典型程序设计语言中的简单控制语句。

```
ident = var
ident = ident + 1
ident = ident - 1
goto label
if var relop var goto label
```

在这些语句中，relop是关系操作符集合{ =, <>, >, <, >=, <= }中的任意一个。ident是标识符，var可以是标识符也可以是常量。所有的这些语句都很简单，因而很容易被理解和实现。

对上述二条赋值语句进行一般化处理，就能够描述更为一般的算术表达式和赋值语句。新的语句为

```
ident = var bin_op var
ident = un_op var
```

这里的bin_op是一个二元算术操作符，而un_op是一个一元操作符。当然，多样的算术数据类型以及自动类型转换会增加这种一般化的复杂程度。如果将上面的语句再增加几条相对简单的指令，就能够描述数组、记录、指针以及子程序的语义。

在第8章，我们将运用操作语义来描述各种控制语句。

3.5.1.2 评估

形式操作语义的首要也是最重要的应用，是用于描述PL/I语言的语义（Wegner, 1972）。那台特殊的抽象机器，连同PL/I语言的翻译规则一起，被称为维也纳定义语言（Vienna Definition Language, VDL）。这个名称源于IBM设计这种定义语言所在的城市名。

操作语义为语言的使用人员以及语言的实现人员提供了一种描述语义的有效方法。其前提是要能够保持这种描述的简单化及非形式化。但PL/I语言的VDL描述过于复杂，以致实际上不能够达到这个目的。

操作语义依赖于较低层次的程序设计语言，而不是数学。借助一种低层次程序设计语言的语句来描述某一种程序设计语言的语句，这种方式可能导致证明，即一些概念会间接地定义于它们的自身。在下面两个小节里将要描述的方法，从基于逻辑与数学而非机器的意义上来说，要更加形式化。

142

3.5.2 公理语义

公理语义的定义是与一种证明程序正确性的方法同时开发的。当能够构造这种正确性证明时，正确性证明表明程序能够按照说明进行运算。在证明中，程序中每一条语句的前后都分别放置了一条给程序中变量设置限制的逻辑表达式。正是这些逻辑表达式（而不是抽象机

器的总状态，如操作语义的情形）被用来描述语句的意义。用于指定限制的标记法——即公理语义的语言，就是谓词演算。尽管简单布尔表达式通常可以表达这种限制，但有时却不够充分。

当公理语义用于形式化指定语句的语义时，其含义由语句对断言（关于该语句所处理的数据）的影响来定义。

3.5.2.1 断言

公理语义以数理逻辑为基础时，这些逻辑表达式就被称为谓词或者断言（assertion）。放置于一条程序语句之前的断言，说明程序运行到该处时对变量的限制。而紧跟在一条语句后面的断言，则说明在这条语句运行之后对于这些变量（以及其他可能的变量）的新限制。这两种断言分别被称为这条语句的前置条件和后置条件。对于两条邻接语句而言，第一条的后置条件是第二条的前置条件。开发给定程序的公理描述或证明，则要求在程序中的每一条语句都同时具有前置条件和后置条件。

在下面的几个小节里，我们将运用这种观点来检测断言：即，从给定的后置条件能够求解前置条件；当然也可以从相反的方向进行检测。我们假设所有变量都为整型。作为一个简单的例子，考虑下面的赋值语句及其后置条件：

```
sum = 2 * x + 1 {sum > 1}
```

为了便于与程序语句相区别，将前置条件与后置条件断言都包括在花括号之中。这条语句的一个可能的前置条件是{ x > 10 }。

在公理语义中，一种特定语句的含义由计算该种语句前置条件的过程并紧带其后置条件来定义。实际上，就逻辑表达式而言，此过程精确地指定了执行语句的效果。

在接下来的小节中，我们关注语句和程序的正确性证明，这是公理语义的常见运用。公理语义的更多的通用概念会以逻辑表达式的形式来精确叙述语句和程序的意义。程序验证是语言公理描述的一种运用。 143

3.5.2.2 最弱前置条件

最弱前置条件是限制最少的前置条件，它将保证相关联的后置条件的有效性。例如，在上述的赋值语句及其后置条件中，{x > 10}，{x > 50}和{x > 1000}都是有效的前置条件。此时，所有前置条件中最弱的一个是{x > 0}。

如果对于某种语言中的每一条语句，都可以从最通用的后置条件来出它的最弱前置条件，那么，用于计算这些前置条件的过程就为那种语言的语义提供了简要的描述。而且，正确性证明能用那种语言编写的程序来构造。程序证明从使用所需要的程序执行结果开始，作为程序最后一条语句的后置条件。这个后置条件连同最后一条语句一道，被用来计算最后一条语句的最弱前置条件。然后这个前置条件又被用来作为倒数第二条语句的后置条件。这个过程一直继续到程序首部为止。而程序第一条语句的前置条件就表示了能够使程序计算出所需结果的条件。如果程序的输入说明隐含了这个条件，这个程序就已经证明是正确的。

推理规则是一种以其他断言值为基础推断出一个断言真值的方法。推理规则的一般形式如下：

```
S1, S2, ..., Sn
----------------
       S
```

上式说明，如果S1，S2，…，Sn为真，那么S的真值就能推出。推理规则的上面部分叫做前件（antecedent），下面部分叫做后件（consequent）。

公理是一条假定为真的逻辑语句。然而，公理是一条没有前件的推理规则。

对于某些程序语句，由语句及其后置条件来推出最弱前置条件是简单的，而且能够使用一条公理来说明。然而在大多数情况下，只能通过推理规则来求得最弱前置条件。

在给定程序设计语言中运用公理语义，无论是用于正确性证明或者形式语义的说明，对于语言中每一类型的语句，都必须定义公理或推理规则。在下面的几个小节中，我们将给出赋值语句的一条公理，以及对于语句系列、选择语句和逻辑先测试循环的一些推理规则。请注意，
144 我们假设算术表达式以及布尔表达式都没有副作用。

3.5.2.3 赋值语句

赋值语句的前置条件和后置条件共同精确定义赋值语句的含义。反之，为了定义赋值语句的含义，我们需要能计算它的前置条件。

假设x = E为一般赋值语句，Q为它的后置条件。这条语句的前置条件P由下列公理来定义：

$$P = Q_{x \to E}$$

它的意义为，将Q中所有x的实例都替换成E，就可以由Q计算出P值。例如，如果我们有如下的赋值语句以及后置条件

```
a = b / 2 - 1 {a < 10}
```

最弱前置条件由b / 2 - 1替换后置条件{a < 10}来计算，如下所示：

```
b / 2 - 1 < 10
b < 22
```

因此，这条赋值语句及其后置条件的最弱前置条件为{b < 22}。请注意，只有在没有副作用的情况下赋值公理才为真。如果一条赋值语句改变了一些变量值而不是赋值号左边的值，则这一条赋值语句具有副作用。

说明给定句型的公理语义的常用标记方法是

{P} S {Q}

这里，P是前置条件，Q是后置条件，S是句型。上面的赋值语句的标记是

$$\{Q_{x \to E}\}\ x = E\ \{Q\}$$

作为计算赋值语句前置条件的另外一个例子，考虑下面的赋值语句及其后置条件：

```
x = 2 * y - 3 {x > 25}
```

它的前置条件计算如下：

```
2 * y - 3 > 25
y > 14
```

145 因此{y > 14}是这一条赋值语句及其后置条件的最弱前置条件。

注意，赋值语句的左边出现在这条语句右边，并不影响最弱前置条件的计算过程。例如，对于

```
x = x + y - 3 {x > 10}
```

其最弱前置条件是

```
x + y - 3 > 10
y > 13 - x
```

在讨论开始时，我们曾经说明公理语义是为证明程序正确性而开发的。在这个前提之下，读者现在自然会问：如何能够使用赋值语句的公理来证明正确性。我们给出的回答是：可以认为具有前置条件和后置条件的赋值语句是一条定理。如果将赋值公理作用于后置条件以及这条

赋值语句之上时，产生了所给定的前置条件，这条定理便得到了证明。例如，考虑逻辑命题

```
{x > 3} x = x - 3 {x > 0}
```

实施赋值公理于

```
x = x - 3 {x > 0}
```

会产生{x > 3}，这正是所给定的前置条件，因而证明了上面的逻辑命题是正确的。

下面，再考虑逻辑命题

```
{x > 5} x = x - 3 {x > 0}
```

在这种情况下，所给定的前置条件{x > 5}不同于公理所产生的断言。然而{ x > 5 } -> { x > 3 }显然成立，为了将它纳入证明之中，我们需要名为后果规则的推理规则。

后果规则的形式为

$$\frac{\{P\}\ S\ \{Q\},\ P'\Rightarrow P,\ Q\Rightarrow Q'}{\{P'\}S\{Q'\}}$$

这里，符号 => 的意义是“蕴涵”，并且S可以为任意的程序语句，可以使用下面的文字来表述这一条规则：如果逻辑命题{P}S{Q}为真，断言P'蕴涵断言P，断言Q蕴涵断言Q'，那么就能推断出{P'}S{Q'}。换言之，后果规则说明的是，后置条件总是能够被变弱，而前置条件总是能够被增强。这在程序证明当中是相当有用的。例如，它能够允许我们完成上面最后一个例子中逻辑命题的证明。如果我们让P为{x > 3}，Q和Q'为{x > 0}，P'为{x > 5}，那么我们就有 146

$$\frac{\{x > 3\}\ x = x - 3\ \{x > 0\},\ \{x > 5\}\Rightarrow\{x > 3\},\ \{x > 0\}\Rightarrow\{x > 0\}}{\{x > 5\}\ x = x - 3\ \{x > 0\}}$$

前件中的第一项({ x > 3 } x = x - 3 { x > 0 })由赋值公理证明。第二和第三项是明显的。因而，通过使用后果规则，后件为真。

3.5.2.4 序列

公理不能够描述语句序列的最弱前置条件，因为前置条件取决于语句序列中是哪种特殊类型的语句。在这种情况下，就只能够使用一条推理规则来描述前置条件。假设S1和S2是相邻的两条程序语句，如果S1和S2分别具有下面的前置条件与后置条件

```
{P1} S1 {P2}
{P2} S2 {P3}
```

对于这样一个两条语句的序列，其推理规则是

$$\frac{\{P1\}\ S1\ \{P2\},\ \{P2\}\ S2\ \{P3\}}{\{P1\}\ S1;\ S2\ \{P3\}}$$

其中，对于上述例子，{ P1 } S1; S2 { P3 }描述了序列S1; S2的公理语义。这条推理规则说明，要想求得序列的前置条件，必须先计算第二条语句的前置条件，然后将新计算出的断言用作第一条语句的后置条件，然后可以使用它来作为第一条语句以及整个序列的前置条件。如果S1和S2分别是赋值语句

```
x1 = E1
```

和

```
x2 = E2
```

那么我们就有

$\{P3_{x1\to E1}\}\ x2 = E2\ \{P3\}$
$\{(P3_{x2\to E2})_{x1\to E1}\}\ x1 = E1\ \{P3_{x1\to E1}\}$

147 因而，后置条件为P3的序列x1 = E1; x2 = E2的最弱前置条件是{($P3_{x1\to E1}$)$_{x2\to E2}$}。例如，考虑下面的序列及其后置条件：

```
y = 3 * x + 1;
x = y + 3;
{x < 10}
```

上面最后一条赋值语句的前置条件为

```
y < 7
```

然后，这个条件被用作第一条语句的后置条件。现在我们可以来计算第一条语句的前置条件：

```
3 * x + 1 < 7
x < 2
```

3.5.2.5 选择

我们接下来考虑选择语句的推理规则。选择语句的一般形式为

```
if B then S1 else S2
```

我们仅考虑包括else子句的选择。其推理规则为

$$\frac{\{B \text{ and } P\}\ S1\ \{Q\},\ \{(\text{not } B) \text{ and } P\}\ S2\ \{Q\}}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S1\ \texttt{else}\ S2\ \{Q\}}$$

这条规则说明必须证明选择语句的两种情形，即当布尔表达式的值为真以及为假的两种情形。位于横线上方的第一条逻辑语句代表then子句，第二条逻辑语句代表else子句。根据推理规则，我们需要一个能够用于这两条子句（即then子句和else子句）的前置条件P。

考虑下面运用选择推理规则计算前置条件的例子。例子中的选择语句是

```
if (x > 0)
  y = y - 1
else y = y + 1
```

假设这条选择语句的后置条件Q是{ y > 0 }，我们就能够使用公理来对then子句赋值

148 y = y - 1 {y > 0}

这就产生了{ y - 1 > 0 }或{ y > 1 }。现在我们对else子句运用相同的公理

```
y = y + 1 {y > 0}
```

这就产生了前置条件{ y + 1 > 0 }或者{ y > -1 }。因为{ y > 1 } -> { y > -1 }，后果规则允许我们使用{ y > 1 }作为整个选择语句的前置条件。

3.5.2.6 逻辑先测试循环

命令式程序设计语言的另一种基本结构是逻辑先测试循环，或while循环。计算while循环的最弱前置条件本质上比计算语句序列的最弱前置条件更为困难，原因不是在所有情况下都能够预先确定循环中的重复次数。在预先知道重复次数的情况下，可以将循环作为序列来处理。

计算循环最弱前置条件的问题与求证有关所有正整数的定理的问题相类似，后者通常是运用归纳法来证明，因而同样的归纳方法也能够运用于循环。归纳法中的主要步骤是要找出一种归纳假设。在while循环的公理语义中的相应步骤则是找出一种被称为循环不变式的断言，这个步骤是计算循环最弱前置条件的关键。

计算一个while循环前置条件的推理规则是

$$\frac{\{I \text{ and } B\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\ \{I \text{ and } (\text{not } B)\}}$$

这里的I是循环不变式。这个推理看起来简单，其实却不然，事情的复杂性在于要找到一条合适的循环不变式。

一个while循环的公理描述可以写为

```
{P} while B do S end {Q}
```

循环不变式必须满足许多要求才能够有用。首先，这个while循环的最弱前置条件必须能够保证循环不变式为真。而反过来，循环不变式则必须保证后置条件在循环终止时为真。这些限制将我们从推理规则过渡到公理描述。在循环执行的期间，循环不变式的真值不能受控制循环的布尔表达式的求值以及循环体内语句的影响。这就是之所以称"不变式"的由来。

while循环的另一个复杂因素是循环终止的问题。如果Q是在循环终止时即刻成立的后置条件，那么前置条件P则保证循环终结时Q能够成立，并且能够保证循环的确终止。 149

一个while结构的完整的公理描述需要所有下列的各项都为真，其中I是循环不变式：

```
P => I
{I and B} S {I}
(I and (not B)) => Q
循环终止
```

要找到循环不变式，我们可以运用一种类似于数学归纳法中决定归纳假设时使用的方法。这种方法如下所述：计算一些情形中的关系，希望能够从中发现适用于一般情形的模式。这时候，将产生最弱前置条件的过程处理为一个函数wp会很有帮助。还需

```
wp ( 语句, 后置条件 ) = 前置条件
```

为了找到I，我们使用循环后置条件Q，从零次开始对循环体反复运算几次，用以计算循环的前置条件。如果循环体包含了单条赋值语句，赋值语句的公理可以用来计算这种情形。考虑一个循环的例子：

```
while y <> x do y = y + 1 end {y = x}
```

请记住，这里的等号被用于两种不同的目的：在断言中，它意味着数学里的相等；在断言外，它意味着赋值操作符。

对于零次循环，其最弱前置条件显然是

```
{y = x}
```

对于一次循环，其最弱前置条件是

```
wp(y = y + 1, {y = x}) = {y + 1 = x}, or {y = x - 1}
```

对于两次循环，它是

```
wp(y = y + 1, {y = x - 1}) = {y + 1 = x - 1}, or {y = x - 2}
```

对于三次循环，它是

```
wp(y = y + 1, {y = x - 2}) = {y + 1 = x - 2}, or {y = x - 3}
```

现在已经很清楚，{ y < x }在一次或多次循环的情况下，都足以保证后置条件成立。结合零次循环情形中的{ y = x }，我们得出：能够使用{ y <= x }作为循环不变式。while 150
语句的前置条件则可以由循环不变式确定。事实上，I就可以用作前置条件P。

对于例子中的循环，我们必须确保我们的选择满足I的四个标准。第一，因为P = I，所以P => I。第二个要求是，

{I and B} S {I}

必须为真。

在我们的例子中，我们有

```
{y <= x and y <> x} y = y + 1 {y <= x}
```

运用赋值公理到下面的公式上

```
y = y + 1 {y <= x}
```

我们得到{y + 1 <= x}，它等价于{y < x}，并且蕴涵在{y <= x and y <> x}之中。因而，上面的关系已经得到证明。

下面，我们必须有

{I and (not B)} => Q

在我们的例子中，我们有

```
{(y <= x) and not (y <> x)} => {y = x}
{(y <= x) and (y = x)} => {y = x}
{y = x} => {y = x}
```

因此，上述的表达式显然为真。再下一步，必须考虑循环的终止。在这个例子中，我们的问题是循环

```
{y <= x} while y <> x do y = y + 1 end {y = x}
```

是否终止？回忆我们曾经假定x和y为整数变量，这样就很容易看出这个循环的确会终止。前置条件保证y初始时不大于x。循环体的每一次重复都增加y的值，直到y与x相等。无论y的初始值比x小多少，它最终将和x相等。到时候，循环将会结束。因为我们选择的I满足所有的四项标准，因而它是以充当循环不变式以及循环前置条件。

上面用来计算循环不变式的过程并不一定总是能够产生最弱前置条件的断言（尽管在上面的例子里是如此）。

作为使用数学归纳法求取循环不变式的另一个例子，考虑下面的循环语句：

151

```
while s > 1 do s = s / 2 end {s = 1}
```

与前面一样，我们使用赋值公理来试图找到这个循环的循环不变式以及前置条件。对于零次循环，最弱前置条件是{ s = 1 }。对于一次循环，它是

```
wp(s = s / 2, {s = 1}) = {s / 2 = 1}, or {s = 2}
```

对于二次循环，它是

```
wp(s = s / 2, {s = 2}) = {s / 2 = 2}, or {s = 4}
```

对于二次循环，它是

```
wp(s = s / 2, {s = 4}) = {s / 2 = 4}, or {s = 8}
```

从以上这几种情形中，我们就能够清楚地看出，这个循环的循环不变式为

{ s 是 2 的一个非负次幂}

再一次，计算出来的I可以用作P，而且I也通过了上述四项要求。与我们在前面例子中所求取的循环前置条件不同的是，这次计算的I显然不是最弱前置条件。考虑使用前置条件{ s > 1 }，逻辑命题

```
{s > 1} while s > 1 do s = s / 2 end {s = 1}
```

就能够很容易被证明，而且它的前置条件的范围比在前面计算的前置条件的范围大得多。正如这一过程所显示的，对于任何为正的a值，不仅是2的幂，这个循环及其前置条件都能够得到满足。运用后来规则我们得知，采用比最弱前置条件更强的前置条件都不会使得证明无效。

求取循环不变式并非总是这么容易。了解一些不变式的性质会有所帮助。首先，循环不变式是削弱了的循环后置条件版本，同时又是循环的前置条件。因此I必须足够的弱，以便满足循环开始执行之前的条件，但是结合了循环终止条件之后，它又必须足够的强，以便足以强制后置条件为真。

因为求证循环终止很困难，所以这条要求常常被忽略。如果循环的终止能够被证明，就称循环的公理描述具有完全正确性。如果其他的条件都能够满足，但是不能够保证循环的终止，这时就称循环的公理描述具有部分正确性。

在更复杂的循环中，即使只是为了部分正确性，也需要颇具技巧才能求得合适的循环不变式。因为计算while循环的前置条件依赖于找到一个循环不变式，因而使用公理语义来求证具有while循环的程序正确性，会是十分困难的。

3.5.2.7 程序证明

本节提供对两个简单程序的验证。第一个正确性证明的例子是一个很短的程序，仅仅包含三条赋值语句序列，用于交换两个变量的值。 152

```
{x = A AND y = B}
t = x;
x = y;
y = t;
{x = B AND y = A}
```

因为这一条程序完全由赋值语句序列组成，从而可以使用赋值公理以及序列语句的推理规则来证明程序的正确性。第一个步骤，是对最后一条语句以及整个程序的后置条件运用赋值公理。这样就产生了前置条件

```
{x = B AND t = A}
```

下一步，我们用这个新的前置条件作为中间语句的后置条件，并计算中间语句的前置条件，即为

```
{y = B AND t = A}
```

再下一个步骤，我们使用这个新的断言作为第一条语句的后置条件，并且再次运用赋值公理。这样就产生了

```
{y = B AND x = A}
```

除了AND操作符的操作数的顺序之外，这与程序的前置条件完全相同。由于AND是一个对称操作符，我们的证明已经完成。

下面的例子是一个计算阶乘函数的虚拟码程序的正确性证明。

```
{n >= 0}
count = n;
fact = 1;
while count <> 0 do
    fact = fact * count;
    count = count - 1;
end
{fact = n!}
```

前面所描述的求取循环不变式的方法对于这个例子中的循环无效。在这里需要一些技巧，还需要对这段代码进行粗略的研究。这个循环以大数相乘先运算的顺序来计算阶乘，即当n大于1时，先计算（n - 1）* n，因而不变式中的一部分可以是

153
```
fact = (count + 1) * (count + 2) * . . . * (n - 1) * n
```

但我们还必须保证count总是非负的。我们可以将这个条件与上面的部分相加，从而得到

```
I = (fact = (count + 1) * . . . * n) AND (count >= 0)
```

接下来，必须检查I是否符合不变式的要求。我们再一次将I作为P来使用，因此P显然也就蕴涵了I。下面的一个问题是

```
{I and B} S {I}
```

I and B

为

```
((fact = (count + 1) * . . . * n) AND (count >= 0)) AND
   (count <> 0)
```

约减为

```
(fact = (count + 1) * . . . * n) AND (count > 0)
```

在这种情形中，我们必须用后置条件的不变式来计算循环体的前置条件。对于

```
{P} count = count - 1 {I}
```

我们计算P为

```
((fact = count * (count + 1) * . . . * n)  AND
   (count >= 1))
```

使用上述公式作为循环体中第一条赋值语句的后置条件，

```
{P} fact = fact * count {(fact = count * (count + 1)
                          * . . . * n) AND (count >= 1)}
```

在这种情况下P为

```
((fact = (count + 1) * . . . * n) AND (count >= 1))
```

显然，I and B蕴涵着P。由此后果规则，

```
{I AND B} S {I}
```

为真。I的最后一个测试是

```
I AND (NOT B) => Q
```

在我们的例子中，这是

154
```
((fact = (count + 1) * . . . * n) AND (count >= 0)) AND
   (count = 0)) => fact = n!
```

这显然为真，因为当count = 0时，第一部分正好是阶乘的定义。所以我们选择的I满足循环不变式的要求。现在我们能够使用从while语句里得到的P（相同于I）作为程序中第二条赋值语句的后置条件

```
{P} fact = 1 {(fact = (count + 1) * . . . * n) AND
   (count >= 0)}
```

由此而产生P

```
(1 = (count + 1) * . . . * n) AND (count >= 0))
```

将此用作下列代码中第一条赋值语句的后置条件

```
{P} count = n {(1 = (count + 1) * . . . * n) AND
    (count >= 0)}
```

产生P为

```
((n + 1) * . . . * n = 1) AND (n >= 0))
```

AND操作符左边的操作数为真（因为1 = 1），并且其右边的操作数恰恰是整个程序段的前置条件{n >= 0}，因此这个程序被证明是正确的。

3.5.2.8 评估

在运用公理方法定义一种完整程序设计语言的语义时，对于语言中每一种语句类型，都必须给出公理或推理规则的定义。事实证明，对程序设计语言中某些语句定义公理与推理规则是一项艰难的工作，对于这个问题的一种显而易见的解决办法就是在设计语言时就考虑公理。这样，这种语言仅仅包括能够为之书写公理或推理规则的语句。然而不幸的是，在公理语义学科的现状下，这样的一种语言会相当短小和简陋。

公理语义是程序正确性证明研究中的一种有力工具，在程序开发及其后的阶段，它为对程序进行推理的工作提供了一个完美的构架。然而，仅仅就语言使用人员或编译器的编写人员将这种方法用于描述程序设计语言的意义方面，它的用途却非常有限。

3.5.3 指称语义

指称语义是众所周知最严格的描述程序意义的方法。这种方法坚实地基于递归函数的理论，对于使用指称语义来描述程序设计语言语义的完整讨论将会很长也很复杂。因而我们在这里介绍的内容只是使读者了解指称语义是如何工作的。

135

指称语义的基本概念是要为每一个语言实体定义一个数学对象以及一个映射函数，这种映射函数将语言实体的实例映射到数学对象的实例上。因为数学对象是严格定义的，它们就代表了相对应的程序实体的准确意义。这种思想是基于这样的事实：我们已有操纵数学对象的严格方法，但还没有类似的严格方式来操纵程序设计语言中的结构。运用指称语义的困难之处在于构造数学对象以及映射函数。这种方法之所以被称为“指称”语义方法，是因为其中的每一个数学对象指称了与其对应的语法实体的含义。

历史注释

人们进行了大量的工作，用以研究使用指称语义的描述来自动产生编译器的可能性（Jones, 1980; Milos et al., 1984; Bodwin et al., 1982）。这些努力已经证明了这种方法的可行性，但这方面的工作还没有达到能够产生实用编译器的程度。

3.5.3.1 两个简单例子

我们使用一种非常简单的语言结构，即二进制数，来介绍指称方法。下面的文法规则用来描述二进制数的语法：

```
<二进制数> → 0
           | 1
           | <二进制数> 0
           | <二进制数> 1
```

图3.9显示了二进制数110的语法分析树。



图3.9 二进制数110的语法分析树

156

为了使用指称语义及上述例子中的文法规则来描述二进制数的意义，我们将实际意义与每一条规则联系起来，这些规则都以单个终结符作为它的RHS。在这个情形中的数学对象是简单的十进制数。因此，这个例子中的二进制数的意义将是与它相等的十进制数。

在我们的例子中，有意义的对象必须与前面两条文法规则相关联。后面的两条文法规则在某种意义上是计算规则，因为它们将一个与对象相关联的终结符与一个可以代表某种结构的非终结符结合起来。假设有一种在语法分析树上往上进行的运算，其右边的非终结符已经被赋予了意义，那么语法规则会需要一个函数来计算LHS的意义。这个LHS的意义必须能够代表这条规则中整个RHS的意义。

假设对象语义值的范围为N，它是一个非负的十进制整数数值的集合。我们正是希望将这样的对象与二进制数相关联。如在前面的文法规则中所描述的，一个命名为M_{bin}的语义函数将语法对象映射到N中的对象上。函数M_{bin}的定义如下：

$$M_{bin}(\text{'0'}) = 0$$
$$M_{bin}(\text{'1'}) = 1$$
$$M_{bin}(\langle \text{二进制数} \rangle \text{'0'}) = 2 * M_{bin}(\langle \text{二进制数} \rangle)$$
$$M_{bin}(\langle \text{二进制数} \rangle \text{'1'}) = 2 * M_{bin}(\langle \text{二进制数} \rangle) + 1$$



图3-10 具有110的指称对象的语法分析树

注意，我们用单引号来包括语法中的数字，以表示它们与数学数字的区别。这两种数字之间的关系类似于ASCII码数字与数学数字的关系。当程序读入一个表示为字符串的数字时，程序在将它作为数字使用之前，必须将它转换成数学数字。

可以将意义或者标志对象（在这个例子中是二进制数）附加到上面的语法分析树的节点上，产生图3-10中的树。这就是语法指导的语义，语法实体被映射到具有具体意义的数学对象之上。

因为我们将来还会用到，所以下面给出一个描述语法十进制字面常量意义的例子。

```
<十进制数> → '0' |'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
           | <十进制数> ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9')
```

对于这些语法规则的指称映射是

$$M_{dec}('0') = 0, M_{dec}('1') = 1, M_{dec}('2') = 2, \cdots, M_{dec}('9') = 9$$
$$M_{dec}(\langle \text{十进制数} \rangle \text{'0'}) = 10 * M_{dec}(\langle \text{十进制数} \rangle)$$
$$M_{dec}(\langle \text{十进制数} \rangle \text{'1'}) = 10 * M_{dec}(\langle \text{十进制数} \rangle) + 1$$
$$\cdots$$
157 $$M_{dec}(\langle \text{十进制数} \rangle \text{'9'}) = 10 * M_{dec}(\langle \text{十进制数} \rangle) + 9$$

在下面的几节里，我们将给出一些简单结构的指称语义。这里所做的最重要的简单假定是假设结构的语法以及静态语义都是正确的。除此以外，我们规定只包括两种标量类型，即整型与布尔型。

3.5.3.2 程序的状态

可以用一台虚拟计算机上的状态变化来定义程序的指称语义。操作语义是以这种方式来定义的，指称语义几乎也可以这样。然而在作进一步的简化以后，可以仅仅使用程序中所有变量的值来进行定义。操作语义与指称语义之间的关键区别在于，操作语义中的状态变化由用某种程序设计语言编写的代码算法来定义，而指称语义中的状态变化则由严格的数学函数来定义。

假设一个程序的状态s由下面一组有顺序的对来代表：

$\langle i_1, v_1\rangle, \langle i_2, v_2\rangle, \ldots, \langle i_n, v_n\rangle$

i是一个变量名，而与之相关的v是这个变量的当前值。这些v中的任意一个都可以具有特殊值undef，它指示与这个v相关联的变量目前无定义。假设VARMAP为一个具有两个参数的函数，其中的一个参数是变量名，另一个参数是程序状态。VARMAP(i_j, s)的值为v_j（在s状态下与i_j相配对的值）。大多数用于程序和程序结构的语义映射函数都在不同的状态之间进行映射。这些状态变化就被用来定义程序以及程序结构的意义。有一些语言结构（如表达式）是被映射到数值而不是状态。

3.5.3.3 表达式

对于大多数程序设计语言，表达式是基本的结构。我们在此假设表达式没有副作用。另外， 158
我们只处理非常简单的表达式：它们仅仅包括 + 和 * 操作符，并且一个表达式最多只能够有一个操作符，在这里仅有的操作数是标量变量和整数字面常量，没有括号；而且表达式的值是整数。下面是这些表达式的 BNF 描述：

```
< 表达式 > → < 十进制数 > | < 变量 > | < 二元表达式 >
< 二元表达式 > → < 左边表达式 > < 操作符 > < 右边表达式 >
< 左边表达式 > → < 十进制数 > | < 变量 >
< 右边表达式 > → < 十进制数 > | < 变量 >
< 操作符 > → + | *
```

在表达式中我们所要考虑的唯一错误是变量具有无定义的值。其他的错误当然也可能出现，但是大部分的错误与机器相关。设Z为一个整数集合，并且设error为错误值。那么Z∪{error}就是表达式计算结果值的集合。

对于给定表达式E和状态s，我们可以给出所需要的映射函数如下。为了区别数学函数的定义与程序设计语言中的赋值语句，我们使用符号Δ = 来定义数学函数。下面定义中的蕴涵符号 => 将一个操作数的形式和与它相关的case(或者switch) 相连接。点的标记 "." 用来表示一个结点的子结点。例如，< 二元表达式 >.< 左边表达式 >表示的是< 二元表达式 >的左子结点。

```
Me( < 表达式 >, s ) Δ=
case < 表达式 > of
  < 十进制数 > => M_num( < 十进制数 >, s )
  < 变量 >    => if VARMAP( < 变量 >, s ) == undef
                 then error
                 else VARMAP( < 变量 >, s )
  < 二元表达式 > =>
      if ( Me( < 二元表达式 >.< 左边表达式 >, s ) == undef OR
           Me( < 二元表达式 >.< 右边表达式 >, s ) == undef )
        then error
      else if ( < 二元表达式 >.<操作符> == '+' )then
              Me( < 二元表达式 >.< 左边表达式 >, s ) +
                                  Me( < 二元表达式 >.< 右边表达式 >, s )
        else Me( < 二元表达式 >.< 左边表达式 >, s ) *
                                Me( < 二元表达式 >.< 右边表达式 >, s )
```

3.5.3.4 赋值语句

赋值语句是一个表达式的求值，可以将赋值语句左边变量的值设定为表达式的值。在这种 159
情况下，意义函数在状态之间进行映射。这个函数可以被描述如下：

M_e(x - E, s) ≙ if M_e(E, s) == error
then error
else s' = {<i_1', v_1'>, <i_2', v_2'>, ..., <i_n', v_n'>}, where
for j = 1, 2, ..., n
if i_j == x
then v_j' = M_e(E, s)
else v_j' = VARMAP(i_j, s)

请注意，在上面倒数第三行的比较，i_j == x，是名字的比较而非数值的比较。

3.5.3.5 逻辑先测试循环

一个简单逻辑循环的指称语义简单得让人迷惑。为了方便讨论，我们假设存在另外两个已有的映射函数M_{sl}和M_{bl}，它们分别将语句列表映射到状态，并将布尔表达式映射到布尔值（或者是error）之上。这个函数为

M_l(**while** B **do** L, s) ≙ if M_b(B, s) == undef
then error
else if M_b(B, s) == false
then s
else if M_{sl}(L, s) == error
then error
else M_l(**while** B **do** L, M_{sl}(L, s))

这个循环的意义仅仅是，如果没有出现错误，循环中的语句被执行规定的次数之后程序中的变量所取的值。在实质上，已经将这个循环从迭代转换成为递归，这里递归控制的数学定义取自其他的递归状态映射函数。较之迭代，递归更容易从数学上严格地描述。到此为止的一个重大发现是，上面的这个定义就像实际程序中的循环一样，可能因为非终止性而不进行任何计算。

3.5.3.6 评估

正如在上面讨论中所使用的那些对象与函数一样，我们也可以为程序设计语言的其他语法实体来定义对象与函数。当对给定语言定义了一套完整的系统时，就能够使用这个系统来决定这种语言中完整程序的意义。这给运用高度严格的方式进行程序设计的思维提供了框架。

指称语义能够辅助语言的设计。例如，如果一条语句的指称语义描述既困难又复杂，这就
160 告诉了设计人员，未来的用户们也会难于理解这样的语句，因而应该转而使用其他可能的设计。

由于指称描述的复杂性，它们对语言的使用人员几乎没有用处。但在另一方面，它们提供了一种精确描述语言的优异方法。

虽然通常将指称语义的使用归功于Scott和Strachey两人（Scott and Strachey，1971），但是关于语言描述的一般标志方式，却一直能够追溯到19世纪（Frege，1892）。

小结

巴科斯-诺尔范式和与上下文无关文法是等价的元语言，它们能够适于理想地描述程序设计语言语法。这不仅是由于它们独特的简明描述方法，它们所具备的与分析操作相关联的语法分析树提供了语法结构的图形表示。此外，它们还与识别它们所产生的语言的装置自然相关，这使得构造这些语言编译器的语法分析器相对容易。

属性文法是一种描述形式，它能够描述语言的语法以及静态语义。属性文法是与上下文无关文法的一种扩展。一个属性文法包括一个文法、一组属性、一组属性计算函数以及一组描述静态语义的谓词。

主要有三种描述语义的方法：操作、公理与指称。操作语义是以语言结构在一部理想机器上的效应来描述语言意义的方法。基于形式逻辑的公理语义被设计为来证程序正确性的一种工具。指称语义则使用

数学对象来代表语言结构的意义。语言实体通过语义函数被等换成为数学对象。

文献注释

关于使用与上下文无关文法以及BNF的语法描述，在Cleaveland和Uzgalis的书中进行了透彻的讨论（Cleaveland and Uzgalis, 1976）。

公理语义的研究始于Floyd（1967），并由Hoare给予更进一步的发展（Hoare, 1969）。Hoare和Wirth运用这种方法描述了Pascal语言中的大部分语义（Hoare and Wirth, 1973）。还没有完成描述的部分涉及了函数的副作用以及goto语句。这被公认为是最难描述的。

Dijkstra描述（和推荐）了在程序的开发中运用前置条件与后置条件的技术（Dijkstra, 1976）。后来，Gries又对此进行了更为详细的讨论（Gries, 1981）。

关于指称语义较好的介绍，能够在Gordon（1979）和Stoy（1977）的文章中找到。关于本章所讨论的三种语义描述方法的介绍，也能够在Marcotty *et al.*（1976）的文章中读到。关于本章许多内容的另一个较好的参考书为（Pagan, 1981）。本章中指称语义函数的形式与Meyer（1990）论文中的相类似。

复习题

1. 给出语法和语义的定义。
2. 语言描述是为什么人设计的？
3. 描述一般语言生成器的运作。
4. 描述一般语言识别器的运作。
5. 语句与句型之间有什么区别？
6. 定义一条左递归文法规则。
7. 在大多数的EBNF之中，哪三个扩展是最常用的？
8. 区别静态语义和动态语义之差别。
9. 谓词在属性文法中起什么作用？
10. 合成属性和继承属性之间有什么不同？
11. 对于给定的属性文法树，属性求值的顺序是怎样决定的？
12. 属性文法的主要应用是什么？
13. 使用软件纯解释器来作为操作语义，存在着什么样的问题？
14. 请解释，给定语句的前置条件与后置条件在公理语义中的意义是什么。
15. 描述使用公理语义证明给定程序正确性的方式。
16. 描述指称语义的基本概念。
17. 操作语义与指称语义在基本方式上有什么不同？

练习题

1. 语言描述的两种数学模型是生成与识别。请分别描述它们是怎样定义程序设计语言的语法的。
2. 写出下列各项的EBNF描述：
   a. Java语言类定义的一条头语句。
   b. Java语言的方法调用语句。
   c. C语言的一条`switch`语句。
   d. C语言的一条`union`定义。
   e. C语言的`float`字面常量。
3. 改写例3.4中的BNF，使得 + 的优先级高于 * ，并且使得 + 为右结合的。
4. 改写例3.4中的BNF，使其包括Java中的 ++ 以及 -- 一元操作符。

5. 为Java的表达式写出BNF的描述，其中包括这样三个操作符：&&，和！，还包括关系表达式。

6. 运用例3.2中的文法为下面每一条语句构造语法分析树和最左派生：

```
a.A = A * (B + (C * A))
b.B = C * (A * C + B)
c.A = A * (B + (C))
```

7. 运用例3.4中的文法，为下面每一条语句构造语法分析树和最左派生：

```
a.A = ( A + B ) * C
b.A = B + C + A
c.A = A * (B + C)
d.A = B * (C * (A + B))
```

8. 证明下面的文法是歧义性的：

<S> → <A>

<A> → <A> + <A> | <标识符>

<标识符> → a | b | c

163 9. 修改例3.4中的文法，增加一个一元减法操作符，并让它的优先级高于+或*操作符。

10. 用英语描述下列文法所定义的语言：

<S> → <A><B><C>

<A> → a <A> | a

<B> → b <B> | b

<C> → c <C> | c

11. 考虑下列文法：

<S> → <A>a<B>b

<A> → <A>b | b

<B> → a <B> | a

下面的哪些句子属于这些文法所产生的语言？

a. baab

b. bbbab

c. bbaaaaa

d. baaab

12. 考虑下列文法：

<S> → a <S> c <B> | <A> | b

<A> → c <A> c

<B> → d | <A>

下面的哪些句子属于这些文法所产生的语言？

a. abcd

b. accebd

c. accchcc

d. acd

e. accc

13. 为一种由字符串构成的语言编写文法，这个字符串包括字母a的n次复制，后面接着字母b的相同次数的复制，n> 0。例如，字符串ab、aaaabbbb和aaaaaaaaabbbbbbbbb是属于这种语言，但是字符串a、abb、ba和aaabbb则不在这种语言之中。

14. 为句子aabb和aaaabbbb画出由练习题13中的文法所派生的语法分析树。

15. 将例3.1中的BNF转换为EBNF。

16. 将例3.3中的BNF转换为EBNF。

17. 将下面的EBNF转换为BNF：

S → A { bA }

A → a [b] A

18. 运用3.5.1.1节中的虚拟机指令，给出下面结构的操作语义定义： 164

a. Java语言的**do-while**

b. Ada语言的**for**

c. Fortran语言如下形式的Do语句：Do N k = start, end, step

d. Pascal语言的**if-then-else**

e. C语言的**for**

f. C语言的**switch**

19. 对于下面列出的第一类赋值语句及其后置条件，计算最弱前置条件：

```
a. a = 2 * (b - 1) - 1 {a > 0}
b. b = (c + 10) / 3 {b > 6}
c. a = a + 2 * b - 1 {a > 1}
d. x = 2 * y + x - 1 {x > 11}
```

20. 对于下面列出的每一系列赋值语句及其后置条件，计算最弱前置条件：

```
a. a = 2 * b + 1;
   b = a - 3
   {b < 0}
b. a = 3 * (2 * b + a);
   b = 2 * a - 1
   {b > 5}
```

21. 为下列的各条语句写出指称语义的状态函数：

a. Ada语言的**for**

b. Java语言的**do-while**

c. Java语言的布尔表达式

d. Java语言的**for**

e. C语言的**switch**

22. 内在属性与非内在合成属性之间的差别是什么？

23. 写出一个属性文法，它的BNF基础是3.4.5节中的例3.6，而它的语言规则如下：在表达式中不能够混合数据类型，但是不要求赋值语句中赋值操作符的两边为相同类型。

24. 写出一个属性文法，它的BNF基础是例3.2，它的类型规则与第3.4.5小节中赋值语句的例子相同。 165

25. 证明下面程序的正确性：

```
{n > 0}
count = n;
sum = 0;
while count <> 0 do
  sum = sum + count;
  count = count - 1;
end
{sum = 1 + 2 + ... + n}
```

166

第4章　词法分析和语法分析

要认真研究编译器的设计，至少需要一个学期的时间来集中学习，包括设计和实现一种实际的小程序设计语言的编译器。这样一门课程的第一部分就是词法和语法分析。语法分析器是编译器的核心，因为其他的一些重要组件，包括语义分析器和中间代码产生器都是由语法分析器的动作所驱动的。

有些读者可能会疑惑：为什么包含编译器所有部分的一章会放入讲述程序设计语言的书中。在本书中包含对词法分析和语法分析的讨论至少有两个原因：首先，语法分析直接基于第3章中讨论过的文法，因此把它们作为文法的一个应用进行讨论就是理所当然的。其次，词法分析器和语法分析器在编译器设计中占有重要的地位。许多应用程序，包括程序列表格式化器、计算程序复杂度的程序以及分析和响应配置文件内容的程序，它们需要做的就是词法分析和语法分析。因此，对于软件开发人员来说，词法分析和语法分析是很重要的主题，即使他们不需要编写一个编译器。而且，一些计算机科学专业不再要求学生完成编译器设计课程，这会让学生缺乏词法分析和语法分析的知识。在上述情况下，可以将本章纳入程序设计语言课程。如果设置了相关的编译器设计课程，则可以跳过本章。

本章从词法分析的介绍开始，包括了一个简单的例子；然后讨论语法分析的一般问题，包括语法分析的两种主要方式以及语法分析的复杂性；接着我们介绍自顶向下语法分析器的递归下降实现技术，包括两个递归下降语法分析器的例子。最后一节讨论自底向上语法分析和LR语法分析算法。这一节还包括了一个小型LR语法分析表的例子，以及使用LR语法分析过程对字符串的语法进行分析。

4.1　概述

我们曾经在第1章里介绍了实现程序设计语言的三种不同方式：编译，单纯解释以及混合实现。编译是使用被称为编译器的一种翻译程序，将用高级语言编写的程序翻译成机器码。一般采用编译技术来实现适合大型应用的程序设计语言，这些应用通常由类似C++或COBOL等语言编写。单纯解释系统不采用翻译程序，而是由软件解释器来对程序的原有形式进行解释。单纯解释通常被用于较小型的系统，这种系统对执行效率的要求并不高，例如，在HTML文档中嵌入由JavaScript编写的脚本。混合实现系统将高级语言编写的程序翻译成为中间形式，然后对这种中间形式进行解释。如今，这种类型的系统得到了比以往任何时候都更为广泛的应用，主要应归功于Java以及Perl语言被广泛接受。从传统意义考虑，混合系统将导致比编译系统慢得多的程序执行速度。然而近年来，Just-In-Time（JIT）编译器的使用开始盛行，尤其在Java程序上的运用。JIT编译器将中间代码翻译成机器码，当第一次调用一个方法时，JIT编译器将对这个方法进行翻译。从实际效果上看，JIT编译器将一个混合系统转换成为一个延迟的编译系统。

167
168

语法分析器（Syntax analyzer或parser）几乎总是基于一种对程序语法的形式描述。最普遍应用的语法描述形式是曾经在第3章里介绍过的上下文无关文法，即BNF。相对于某些非形式的语法描述，使用BNF至少具有三个吸引人的优点：首先，BNF程序语法的描述，无论是对于人还是对于使用它们的软件系统，都十分清晰与精确；其次，能够将BNF的描述作为语法分析器

的基础；最后，因为有清晰的模块化结构，基于BNF来实现的系统相对容易维护。

几乎所有编译器都将分析语法的任务分成两个分离的部分，即词法分析和语法分析。虽然这样的术语容易让人混淆。简言之，词法分析处理小规模的语言结构，如名字和数字的字面常量。语法分析处理大规模的结构，如表达式、语句以及程序单元。我们将在第4.2节介绍词法分析，在4.3节、4.4节和4.5节讨论语法分析。

下面叙述一些理由，用以说明为什么要将词法分析从语法分析中分离开来。

1. 简单性。词法分析需要的技术比语法分析较简单。因而如果将词法分析从语法分析中分离出来，它的分析过程就可能比较简单。另外，将词法分析中的低层次细节从中移出来之后，语法分析器就更为小巧而简洁。

2. 效率。优化词法分析器会获得相当的回报，因为词法分析占用总编译时的相当一个部分；然而优化语法分析器则没有效果。分离这两种分析器有助于选择性地进行优化。

3. 可移植性。因为词法分析器读入输入的程序文件，并已通常包括了输入文件的缓冲，它在某种程度上依赖于平台。然而，语法分析器则可以是独立于平台的。将软件系统中依赖于机器的部分分离开来，总是较为明智的举动。

4.2 词法分析

词法分析器实质上是一个模式匹配器。模式匹配器总是试图从一个给定字符串中找到与一个给定的字符模式相匹配的子串。模式匹配是计算科学中的传统部分。最早期模式匹配的一个应用是文本编辑器，如在UNIX早期版本中引入的ed行编辑器。从那个时候开始，模式匹配就进入了程序设计语言，例如，它存在于Perl和JavaScript语言中。在Java，C++以及C#的标准类库中也存在。 169

可以将词法分析器作为语法分析器的前端。在技术上，词法分析是语法分析的一个组成部分。词法分析器是在程序结构的最低层进行语法分析。一段输入的程序对于编译器而言就像是单个的字符串。词法分析器将字符输入逻辑的组合，并根据这些组合的结构赋以内部编码。这种字符组合称为词素（lexeme），而这种组合种类的内部编码称为标记（token）。通过将输入的字符串与字符串模式匹配的方式来进行词素的识别。尽管在编码中，通常使用整数值作为标记，为了词法分析器以及语法分析器的可读性，通常是使用命名常量来进行引用。

考虑下面一条赋值语句的例子：

```
result = oldsum - value / 100;
```

下面分别是这条语句的标记与词素：

标记	词素
标识符	result
赋值_操作符	=
标识符	oldsum
减法_操作符	-
标识符	value
除法_操作符	/
整数_常量	100
分号	;

词法分析器从给定的输入字符串中提取词素，并产生与之对应的标记。在早期的编译器中，词法分析器通常处理整个程序文件，从而产生标记与词素的文件。然而现在大部分的词法分析器都是语法分析器的子程序，它们从输入中产生下一个词素，还产生与词素相关联的标记代码。

并将这些代码返回给调用程序——即语法分析器。语法分析器唯一能够看到的输入程序就是自词法分析器产生的输出，一次一个词素。

词法分析的过程包括了跳越注释以及词素之外的空格，因为它们与程序的意义无关。此外，词法分析器将用户定义名字的词素插入符号表，以供编译器在后面阶段使用。最后，词法分析器将找出标记里的语法错误，例如有错误的浮点字面常量，并向用户报告这种错误。

170

构造一个词法分析器有三种方式：

1. 使用与正则表达式[⊖]有关的一种描述性语言，写出语言标记模式的形式描述，并使用软件工具自动产生词法分析器。有许多这种用途的工具，其中最古老且最容易获得的是lex，它通常是UNIX系统的一部分。

2. 设计一个描述语言标记模式的状态转换图，并编写出实现这种状态转换图的程序。

3. 设计一个描述语言标记模式的状态转换图，并手工建造一个这种状态图的表格驱动式实现。

状态转换图，或简单地称为状态图，是一种有向图。在状态图的节点上标有状态的名字，而在状态图的弧线上则标有引起这种状态转换的输入字符，弧线也可以包括在实行这种转换时词法分析器所必须采取的动作。

用于词法分析器的这种状态图被表示成称为有限自动机的数学机器的类。可以设计有限自动机来识别一类被称为正则语言的语言。正则表达式和正则文法是正则语言的生成装置。程序设计语言的标记就是一种正则语言，而词法分析器则是一台有限自动机。

我们现在用一个状态图及实现这个状态图的代码来介绍词法分析器的构造。状态图可以仅包括每一个标记的状态及其状态的转换，然而这种方式将产生一个极为庞大和复杂的图形，因此必须想办法简化。

假设，我们需要一个词法分析器，它仅仅识别程序名、保留字以及整数字面常量。在这个例子中，名字由包括大写字母、小写字母以及数字的字符串组成，但是必须从字母开始。这些名字没有长度限制。我们首先观察到，可以使用52个不同的字符（英文的大、小写字母）来开始一个名字，这就要求转换图的初始状态具有52个转换。然而，词法分析器的兴趣仅仅是确定这是个名字，并不关心这是一个什么样的特定名字，因此，我们为这52个英文字母定义一个字符类LETTER，在任何时候，只对名字的第一个字母实施单个转换。

171

然后我们观察到，名字以及保留字具有相类似的模式。虽然可以建立一个状态图来识别一种程序设计语言中的每个特定的保留字，但这样会产生一个很大的状态图。如果让词法分析器使用相同的模式来识别名字和保留字，然后再通过查询保留字表来决定哪些名字为保留字，这样就会简单快捷得多。使用这种方式把保留字作为标记种类的例外。

简化转换图的另外一种可能方式是使用整数字面常量的标记。整数字面常量的词素可以使用十个不同字符来作为第一个字符，这样就需要十个状态图初始态的转换。因为词法分析器并不关心是哪一个数字，所以如果我们也为数字定义一个字符类DIGIT，并对这个字符类中的任意字符施行单个转换，将其转换到整数字面常量的状态，我们就能构造出一个更为紧凑的状态图。

大多数程序设计语言允许程序名在首个字母后紧跟数字。为了从名字第一个字符后的节点开始转换，我们使用了在LETTER或DIGIT上转换来继续收集名字的字符。

下面，我们来定义一些在词法分析器内完成一般任务的功能子程序。首先，我们需要一个名为getChar的子程序。当被调用时，getChar从输入程序中获取下一个输入字符，并将它

⊖ 这些正则表达式是现在许多程序设计语言模式匹配的基础部分（直接地或间接一个类态）。

放入全局变量nextChar之中，这可能需要读进输入的下一行或者整个输入缓冲区。getChar还必须决定输入字符的字符类，并将它放入全局变量charClass之中。对于我们的简单范例语言，我们有一个字母的字符类，还有一个数字的字符类。由词法分析器构造的词素可能被实现为一个字符串或者一个数组，并将其命名为lexeme。

我们再用一个名为addChar的子程序来实现这样的过程：将nextChar内的字符放入lexeme中。这个子程序必须被显式调用，因为程序中还包括了一些不需要放入lexeme中的字符，即词素间的空白字符。当调用词法分析器时，假如输入的下一个字符是下一个词素的第一个字符，它是很方便的。为此，使用getNonBlank函数来跳过空格符。

最后，我们需要用一个名为lookup的子程序来确定lexeme中的当前内容究竟是一个保留字还是一个名字。如果词素不是一个保留字，lookup子程序将返回零值。否则将返回保留字的标记代码。在这里，假定名字的标记代码为零。标记代码是由编译器设计人员随意分配给标记的数字。

图4-1中的状态图描述了我们的标记模式。它包括了状态图上每一种转换所需要的动作。

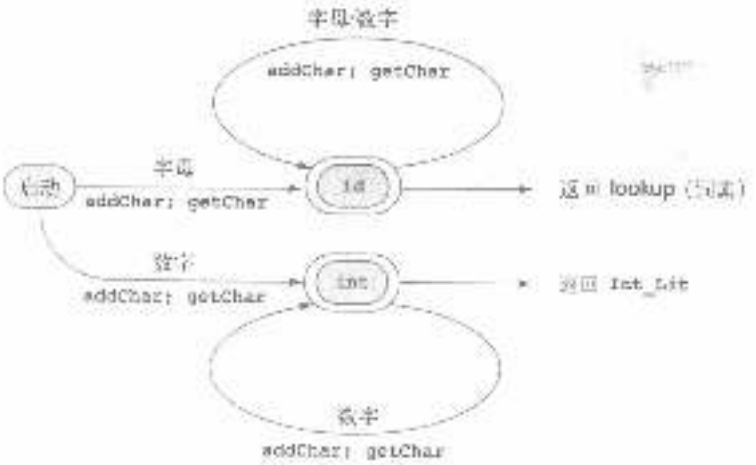


图4-1 识别名字、保留字以及整数字面常量的状态图

使用程序来实现这个状态图相对容易。下面的C函数即是图4-1状态图词法分析器一个例子：

```
/* Global variables  */
      int charClass;
      char lexeme [100];
      char nextChar;
      int lexLen;
      int LETTER = 0;
      int DIGIT = 1;
      int UNKNOWN = -1;

/* addChar - a function to add nextChar to lexeme */

      void addChar() {
        if(lexLen <= 99){
          lexeme[lexLen++] = nextChar;
        else printf("Error - lexeme is too long \n");
      }

/* getChar - a function to get the next character of input
```

172
,
173

```
        and determine its character class */

    void getChar() {
      /* do whatever is required to get the next
         character from input and put it in nextChar */
      if(isalpha(nextChar))
        charClass = LETTER;
        else if (isdigit(nextChar))
              charClass = DIGIT;
            else charClass = UNKNOWN;
    }

/* getNonBlank - calls getChar until it returns
                 a non-whitespace character */

    void getNonBlank() {
      while(isspace(nextChar))
        getChar();
    }

/* lex - a simple lexical analyzer */

    int lex() {
      lexLen = 0;
      static int first = 1;

/* If it is the first call to lex, initialize by calling
getChar */

      if(first) {
        getChar();
        first = 0;
      }
      getNonBlank();
      switch (charClass) {

/* Parse identifiers and reserved words */

        case LETTER:
          addChar();
          getChar();
          while (charClass == LETTER ||
                 charClass == DIGIT) {
            addChar();
            getChar();
          }
          return lookup(lexeme);
          break;

/* Parse integer literals */

        case DIGIT:
          addChar();
          getChar();
          while (charClass == DIGIT) {
            addChar();
            getChar();
```

```
            }
            return INT_LIT;
            break;
        } /* End of switch */
    } /* End of function lex */
```

这段代码表明词法分析器的相对简单性。当然我们在这里省略了完成许多必要工作并包括了许多细节的功能函数。此外，我们在这里仅仅处理了一个很小的简单标记集合。

词法分析器通常负责符号表的初始创建工作，这种符号表也就是编译器的名字数据库。符号表中储存了有关用户定义名字以及这些名字的属性的信息。例如，如果一个名字是一个变量，那么变量类型就是名字的一个属性，所以变量类型将被存入符号表之中。通常，名字由词法分析器放入符号表中，而名字的属性则由编译器的其他部分在词法分析之后放入符号表中。 174

4.3 语法分析问题

本节讨论一般语法分析问题，并将介绍两种主要的语法分析算法，即自顶向下算法和自底向上算法，还将包括语法分析过程的复杂性问题。

4.3.1 语法分析介绍

用于程序设计语言的语法分析器为程序构造语法分析树，在某些情况下，只是隐式地构造语法分析树。这意味着也许仅仅是产生了树的遍历，但是在所有的情况下，建立语法分析树所必需的信息都在语法分析的过程产生。无论是分析树还是派生，都包括了语言处理器所需要的所有有关语法的信息。

语法分析有两个不同的目的。首先，语法分析必须检查输入的程序，以确定它在语法上的正确性。一旦发现错误，分析器必须产生诊断信息，并且恢复编译。在这里，恢复编译意味着分析器必须回复到正常状态，并且继续分析输入程序。这是十分必要的，只有这样，编译器才能够在一次输入程序的分析过程中尽可能多地发现错误。如果没有做好，从错误中恢复可能会产生更多的错误，或者至少是更多的出错信息。语法分析的第二个目的是对于语法正确的输入程序产生一棵完整的语法分析树，或者至少追踪整个语法分析树的结构。这棵语法分析树（或者是这种追踪）将被作为翻译的基础。 175

根据语法分析器建立语法分析树的方向可以将语法分析器进行分类。语法分析器有两大类，一类是自顶向下，这类分析树是从树根往下建造到树叶；另一类是自底向上，这类分析树是从树叶往上建造到树根。

在这一章中，为了讨论的清晰性，我们将使用一小组符号约定来表示文法符号以及字符串。对于形式语言，这种符号约定如下所示：

1. 终结符——小写字母表的最前面几个字母（a,b,…）。
2. 非终结符——大写字母表的最前面几个字母（A,B,…）。
3. 终结或者非终结——大写字母表的最后面几个字母（W,X,Y,Z）。
4. 终结字符串——小写字母表的最后面几个字母（w,x,y,z）。
5. 混合字符串（终结以及/或者非终结）——小写的希腊字母（α,β,δ,γ）。

对于程序设计语言而言，终结符是语言的小型语法构造或标记。程序设计语言的非终结符通常为由尖括号包括起来的隐含名字或缩写。例如<while_语句>、<表达式>和<函数定义>。一种语言的语句（对于程序设计语言，也即程序）是终结的字符串。描述文法规则RHS的混合

字符串被用于语法分析的算法之中。

4.3.2 自顶向下语法分析器

自顶向下语法分析器按前序遍历或构造一棵语法分析树，它对应于一个最左派生。一棵语法分析树的前序遍历开始于树根。每一个节点都先于它后面的树枝被访问。对于同一个节点的树枝的访问，遵循自左向右的顺序。

根据派生，可以将一个自顶向下的语法分析器描述为：当给定的句型是一个最左派生的部分时，语法分析器的任务是要在那个最左派生之中找到下一个句型。一个最左派生的一般形式为xAα，按照我们的记号约定，x是一个终结符串，A是一个非终结，α则是一个混合字符串。因为x只包含了终结，A就是这个句型里的最左非终结。这面必须将A扩展以便在最左派生中得到下一个句型。要确定下一个句型，需要选择一条正确的文法规则，这个规则必须以A作为它的LHS。例如，如果当前的句型为

136 xAα

并且有A-规则：A→bB，A→cBb，以及A→a。自顶向下语法分析器必须在这三条A-规则中进行选择，从而获得下一个句型。这个句型的形式可能会是：xbBα、xcBbα或者xaα。这就是自顶向下语法分析器的分析决策问题。

不同的自顶向下算法运用不同的信息进行语法分析的决策。通过比较输入的下一个标记和能由那些规则的RHS产生的开始符号，最常用的自顶向下语法分析器在当前句型中为最左非终结符选择正确的RHS。无论哪个有字符串右边末尾标记的RHS，它产生的都是正确的。因此，在xAα句型中，语法分析器将根据字符串x中最后一个词素后面的任意标记来决定使用哪一条A-规则，以得到下一个句型。在上面的例子中，A-规则的3个RHS都用不同的终结符开始。语法分析器很容易选择基于输入下一个标记的正确的RHS（这个例子中，必须为a、b或c）。通常，选择正确的RHS并不是如此简单的，因为在当前句型中的一些最左非终结符的RHS可能会以一个非终结符开始。

最常用的一些自顶向下语法分析的算法，相互间是密切关联的。递归下降语法分析器是一种直接基于语言语法的BNF描述的语法分析器的编码版本。对于递归下降分析方法的最常用替代方法，是使用一种语法分析表而不是编码来实现BNF规则。这两种被称为LL算法的方法具有同等的功能，这意味着它们都处理相同的一个文法子集。LL中的第一个字母L说明输入扫描是从左到右；而第二个字母L说明所产生的是最左派生。在第4.4节将介绍如何运用递归下降方法来实现一个LL语法分析器。

4.3.3 自底向上语法分析器

自底向上语法分析器是由叶节点开始向树根发展，从而构造一棵语法分析树。自底向上的语法分析器产生最右派生的逆向。根据派生，可以将一个自底向上的语法分析器描述为：当给定一个右句型[⊖]α，语法分析器必须确定α中的那一个子串是在一条文法规则的右边（RHS），而当将这个子串约减到规则的LHS时，就能生成最右派生中先前的句型。例如，自底向上语法分析器的第一个步骤是决定最初给定的句子中哪个子串是一条规则的RHS，并且要被归约到与之相应的LHS中，以便得到派生过程里的倒数第二个句型。要找到正确的RHS进行归约是一个复

⊖ 右句型是一种出现在最右派生的句型。

杂的过程，原因是，一个右句型可能包含多个来自被分析语言的文法中的RHS。正确的RHS被称为句柄（handle）。

考虑下面的文法和派生：

177

S→aAc

A→aA | b

S→aAc⇒aaAc⇒aabc

句子aabc的自底向上语法分析器带着句子开始，并且必须找到其中的句柄。在这个例子中，这是个容易的任务，因为字符串只包含一个RHS:b。当语法分析器用它自己的LHS（A）取代b时，它在派生中得到第二个最后的句型aaAc。在一般情况下，正如前面所说的，找寻句柄是比较困难的，因为一个句型可能包含有许多不同的RHS。

自底向上语法分析器通过审查可能句柄的一边或两边的符号来寻找一个给定右句型的句柄。位于可能句柄右边的符号通常是输入中的，但还没有被分析的标记。

最常用的自底向上语法分析的算法在LR家族中。这里的L说明是从左到右的输入扫描，而R则说明所产生的是最右派生。

4.3.4 语法分析的复杂性

任何用于歧义文法的语法分析算法都十分复杂与低效。事实上，这些算法的复杂性是O (n^3)，这意味着这类算法所需要的时间与被分析字符串长度的三次方成正比。之所以需要相当长的时间，是因为这些算法必须经常进行备份，并且必须重复分析句子的某些部分。当语法分析器在分析过程中犯了错误时，就必须实施重复分析。当将一棵语法分析树(或它的除速)拆散并重建时，实施备份也是十分必要的。O (n^3)的算法因为太慢，常常不适于实际的应用。例如，进行编译器的语法分析，在这种情况下，计算机科学家通常会寻找比较快的算法，尽管这些算法并不一定具有普遍适用性。这就意味着需要牺牲普遍适用性而换取高的效率。就语法分析而言，人们常常发现，快速算法只能够适用于分析所有可能文法中的一个子集。当然，只要这个子集包括了描述程序设计语言的文法，这样的算法就是可以接受的。（实际上，正如在第3章的讨论，任何文法都不足以描述大多数程序设计语言的所有语法。）

178

所有用于编译器的语法分析器算法都具有O (n)的计算复杂性，这意味着它们所耗费的时间与被分析字符串的长度成线性关系。这较之复杂性为O (n^3)的算法效率要高得多。

4.4 递归下降语法分析

本节介绍自顶向下语法分析器的实现过程，即递归下降。

4.4.1 递归下降语法分析过程

之所以命名为递归下降语法分析器，因为它是由一系列子程序组成，这些子程序多数为递归程序，并且在它们产生语法分析树时都按自顶向下的顺序。这种递归反映了程序设计语言的性质，它们包括了多种不同的递归结构。例如，语句常常被嵌套于其他的语句之中。此外还有，必须正确地匹配表达式中的括号。所有这些结构的语法都可以由递归文法规则自然地描述。

EBNF对于递归下降语法分析器是十分理想的。第3章曾讲过，EBNF主要的一种扩展是运用花括号和方括号。花括号说明所包括的部分可以出现零次或者多次，而方括号则说明所包括

的部分可以出现零次或者一次。注意，这两种括号中所包括的符号都是可以选择的。例如：

```
< if_语句 > → if < 逻辑_表达式 > < 语句 > [else < 语句 >]
< 标识符_表 > → 标识符 { , 标识符 }
```

在上面的第一条规则中，if语句的else子句是可选择的，而在第二条规则中，<标识符_表>是一个标识符，后面跟着一个逗号和一个标识符的零次或多次的重复。

对于文法中的每一个非终结，递归下降语法分析器都有一个对应的子程序。与一个特定非终结相关的子程序具有如下责任：当给出一个输入字符串时，它都能够追踪出一棵语法分析树，树根就是这个非终结，树叶则与输入的字符串相匹配。在效果上，一个递归下降语法分析子程序就是由相关非终结产生的这种语言（一组字符串）的一个语法分析器。

考虑下面的简单算术表达式的EBNF描述：

```
< 表达式 > → < 项 > { ( + | - ) < 项 > }
< 项 > → < 因子 > { ( * | / ) < 因子 > }
< 因子 > → 标识符 | ( < 表达式 > )
```

回忆在第3章的描述，一个算术表达式的EBNF文法（例如上面的这个）并不强制任何结合性规则。因而，当使用这样的文法作为编译器的基础时，我们必须小心地确保，这种通常由语法分析驱动的代码生成过程生成的代码与语言的结合规则相一致。使用递归下降的语法分析时可以很容易地做到这一点。 179

我们用下面的递归下降函数expr作为一个例子，在这个例子里，词法分析器是一个被命名为lex的函数。lex函数获取下一个词素，并且将词素的标记代码放入全局变量nextToken。标记代码被定义为命名常量，例如，PLUS_CODE是加号标记代码的一个命名常量。

只有一个RHS的规则的递归下降子程序相对简单些。将RHS中的每一个终结符都与变量nextToken进行比较，如果它们不匹配，就是一个语法错误；如果它们相匹配，则调用词法分析器以便再获取下一个输入标记。对于每一个非终结，就调用对应于这个非终结的语法分析子程序。

对于上面文法例子中的第一条规则，用C语言编写的递归下降子程序如下：

```
/* Function expr
   Parses strings in the language generated by the rule:
   <expr> -> <term> {(+ | -) <term>}
   */
void expr() {

/* Parse the first term */

    term();

/* As long as the next token is + or -, call lex to get
the next token, and parse the next term */

  while (nextToken == PLUS_CODE ||
         nextToken == MINUS_CODE){
    lex();
    term();
  }
}
```

编写递归下降子程序有一个协定，即每一个子程序都在nextToken中放入下一个输入的标

记。因而，当开始一个语法分析函数时，假定nextToken具有还没有被语法分析过程使用的输入标记中最左面的一个标记。

expr函数所分析的语言部分包括：被加号或减号分开的一项或多项。这就是由非终结<表达式>生成的语言。因而它首先调用对项（<项>）进行语法分析的函数，然后它就继续调用这个函数，直到找到了PLUS_CODE或MINUS_CODE标记（通过调用lex来进行传递）。这个递归下降函数比大多数情况都简单，因为它的规则只包含一项 RHS。此外，它不包括任何检测语法错误或进行恢复的代码，因为不存在与这条文法规则相关的任何可检测错误。 189

当一个非终结的规则具有多项RHS时，这个非终结的递归下降语法分析子程序首先从确定究竟应对哪个RHS进行语法分析开始。在编译器的构造时期，需要检测每个RHS，以确定哪一组终结符可以出现在所产生句子的开头部分。通过将这一组终结符与下一个输入的标记相匹配，语法分析器就能够选择出正确的 RHS。

<项>的语法分析子程序与<表达式>的语法分析子程序相类似：

```
/* Function term
   Parses strings in the language generated by the rule:
   <term> -> <factor> {(* | /) <factor>}
   */
void term() {

/* Parse the first factor */

    factor();

/* As long as the next token is * or /, call lex to get
   the next token, and parse the next factor */

  while (nextToken == AST_CODE ||
         nextToken == SLASH_CODE){
    lex();
    factor();
  }
}
```

我们的算术表达式文法中用来分析非终结<因子>的函数必须在两个 RHS 之中做出选择。这个函数还包括了错误检测。<因子>的函数中，在发现一个语法错误时所采取的行动仅仅是调用error函数。而当真正的语法分析器发现错误时，必须产生诊断信息。此外，语法分析器必须能够从错误中恢复，以便继续进行语法分析过程。

```
/* Function factor
   Parses strings in the language generated by the rule:
       <factor> -> id  |  (<expr>)
   */

void factor() {

/* Determine which RHS */

  if (nextToken == ID_CODE)                                    181
/* Get the next token */

    lex();
```

```
/* If the RHS is (<expr>), call lex to pass over the left
   parenthesis, call expr, and check for the right
   parenthesis */

  else if (nextToken == LEFT_PAREN_CODE) {
    lex();
    expr();
    if (nextToken == RIGHT_PAREN_CODE)
      lex();
    else
      error();
  }  /* End of else if (nextToken == ... */

/* It was neither an id nor a left parenthesis */

  else error();

}
```

要跟踪语法分析，可以在每一个语法分析子程序的头部与尾部都加上代码。另外，在递归里也可以加上每次对lex函数的调用，其中包括被返回的标记。例如，在expr函数的头部可以加上：

```
printf("Enter <expr> \n");
```

而在该函数的尾部可以加上：

```
printf("Exit <expr> \n");
```

下面是使用语法分析函数expr，term，factor以及lex对a + b进行语法分析的追踪。请注意，这个语法分析开始于对lex函数以及起始符子程序的调用；在这个情形中，起始符子程序即为expr。

```
Call lex  /* returns a */
Enter <expr>
Enter <term>
Enter <factor>
Call lex  /* returns + */
Exit <factor>
Exit <term>
Call lex  /* returns b */
Enter <term>
Enter <factor>
Call lex  /* returns end-of-input */
Exit <factor>
Exit <term>
Exit <expr>
```

182

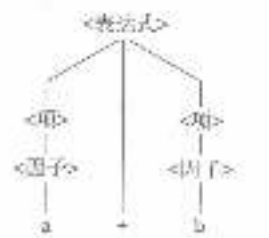


图4-2 a + b的语法分析树

图4-2显示了由这个语法分析器所追踪产生的语法分析树。

下面是对Java中if语句的语法描述：

```
<if_语句> if (<布尔表达式>) <语句> [else <语句>]
```

这条规则的递归下降子程序如下所示：

```
/* Function ifstmt
    Parses strings in the language generated by the rule:
    <ifstmt>->if (<boolexpr>) <statement>
```

```
                [else <statement>]
    */
void ifstmt() {
/* Be sure the first token is 'if' */
  if (nextToken != IF_CODE)
    error();
  else {
/* Call lex to get to the next token */
    lex();
/* Check for the left parenthesis */
    if (nextToken != LEFT_PAREN_CODE)
      error();
    else {
/* Call boolexpr to parse the Boolean expression */
      boolexpr();
/* Check for the right parenthesis */
      if (nextToken != RIGHT_PAREN_CODE)
        error();
      else {
/* Call statement to parse the then clause */
        statement();
/* If an else is next, parse the else clause */
        if (nextToken == ELSE_CODE) {
/* Call lex to get over the else */
          lex();
          statement();
        } /* end of if (nextToken == ELSE_CODE ... */
      } /* end of else of if (nextToken != RIGHT ... */
    } /* end of else of if (nextToken != LEFT ... */
  } /* end of else of if (nextToken != IF_CODE ... */
} /* end of ifstmt */
```

183

所有这些例子的目的是想让读者们相信，对于任何一种语言，只要具有合适的文法，就很容易编写递归下降语法分析器。在下面的小节里，我们就来讨论那些能够让我们构建递归下降语法分析的文法特征。

4.4.2 LL文法类

在选择递归下降作为编译器或其他程序分析工具的语法分析策略之前，我们必须考虑到这种方式在有关文法限制方面的局限性。本节将讨论这些有关的限制以及可能的解决办法。

有一种简单的文法特征会给LL语法分析器带来实质性的问题，它就是左递归。例如，考虑下面的规则：

$A \rightarrow A + B$

A的递归下降语法分析器子程序会立刻调用自身来对RHS中的第一个符号进行语法分析。激活这个语法分析子程序将导致它即刻再次调用自身，一次又一次，如此反复。显然，这将是无休止的。

在规则$A \rightarrow A+B$中的左递归称为**直接左递归**，因为它出现在同一条规则里。直接左递归可以通过下面过程的文法来消除：

对于每个非终结符，A，

1. 将A-规则分组为$A \rightarrow A\alpha_1 | \cdots A\alpha_n | \beta_1 | \cdots | \beta_r$

这里，β都不以A开始。

2. 用下面内容取代最初的A-规则

$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_n A'$

$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_m A' | \varepsilon$

注意，ε表示空字符串。让ε作为RHS的规则称为消除规则。当为在派生中使用它可以从句型中有效地消除它的LHS。

考虑下面例子的文法和上面过程的运用：

E→E + T | T

T→T * F | F

F→(E) | id

对于E-规则，有α_1=+T和β=T，因此，用以下内容替代E-规则

E→TE'

E'→+TE' | ε

对于T-规则，有α_1=*F和β=F，因此，用以下内容替代T-规则

T→FT'

T'→*FT' | ε

因为在F-规则中没有左递归，它们保持相同，所以完整的替代文法是

E→TE'

E'→+TE' | ε

T→FT'

T'→*FT' | ε

F→(E) | id

该文法产生与最初文法相同的语言，但是它不是左递归的。

正如在4.1.1节中使用BNF编写的表达式文法，此文法不指定操作符的左结合性。然而，设计基于该文法的代码生成是相对容易的，以致于加法和乘法操作符将具有左结合性。

间接左递归也具有与直接左递归相同的问题。例如，假设我们有

A → B a A

B → A b

这些规则的一个递归下降语法分析器将使得A的子程序立即调用B的子程序，而B的子程序又再立即调用A的子程序。所以，这里的问题与直接左递归的问题是相同的。左递归的问题不仅仅局限于以递归下降方式来建造的自顶向下语法分析器，它是所有自顶向下语法分析算法的一个共同问题。幸运的是，左递归对于自底向上的语法分析算法不存在问题。

184
-
185

有一种修改给定文法的算法可以用来排除间接的左递归（Aho *et al.*, 1986）。然而我们不会在这里进行介绍。当为一种程序设计语言编写文法时，必须小心避免采用任何直接和间接的左递归。

左递归还不是我们所不能接受的唯一的自顶向下语法分析的文法特性。另一个这样的特性为，是否语法分析器总能根据下一个输入的标记来选择正确的RHS。对于非左递归的文法，有一种相对简单的测试，它能够测出语法分析器能否以这样的方式来选择正确的RHS。这种测试称为**成对不相交测试**（pairwise disjointness test），这种测试要求能够根据一种文法给定的非终结符的RHS计算出一个被称为FIRST的集合。FIRST集合的定义如下：

FIRST(α) = {a | α =>* aβ} (If α =>* ε, ε is in FIRST(α))

这里的"=>*"意味着0或多个派生步骤。

在Aho等人的文献（Aho *et al.*, 1986）中能够发现对于任何混合字符串α计算FIRST的算法，单就我们的目的而言，常常能够通过检测文法的自身来计算FIRST。

成对不相交测试为：

对于具有多个RHS的文法中的每一个非终结A，

对于每一对规则$A \rightarrow \alpha_i$和$A \rightarrow \alpha_j$，$FIRST(\alpha_i) \cap FIRST(\alpha_j) = \phi$必须为真。

（即$FIRST(\alpha_i)$与$FIRST(\alpha_j)$两集合的交集必须为空。）

换言之，如果非终结A具有多个RHS，在每个RHS的派生中产生的第一个终结符对于这个RHS都必须是唯一的。考虑下面的规则：

A → aB | bAb | Bb

B → cB | d

对于这些规则的三个RHS，三个FIRST的集合分别为{ a }、{ b }和{ c }，这些集合显然是不相交的。因此，这些规则通过了成对不相交测试。这意味着，就递归下降语法分析器而言，对于非终结A进行语法分析的子程序可以通过观察在输入中这个非终结所产生的第一个终结符（标记），来选择处理哪一个RHS。现在考虑下面的规则：

A → aB | BAb

B → aB | b

这两条规则中的两个RHS的FIRST集合分别为{ a }和{ a }，它们显然是相交的。因而，这些规则不能通过成对不相交测试。就语法分析器而言，A的子程序就不能仅仅通过检测下一个输入符号来确定对哪个RHS进行语法分析。当为a时如果它是a的话，可以选择两个RHS中的任意一个。如果一个或多个RHS开始于非终结，这个问题自然会变得更复杂。

在许多情况下，可以对没有通过成对不相交测试的文法进行修改，使得它在修改后能够通过这个测试。例如，考虑规则

< 变量 > → 标识符 | 标识符 [< 表达式 >]

这里的规则说明，一个<变量>可以是一个标识符，也可以是一个标识符后面跟随一个在方括号内的表达式（下标）。这些规则显然不能通过成对不相交测试，因为这两个RHS起始于相同的终结符，即标识符。经过一个被称为**提取左因子**（left factoring）的过程，这个问题能够得到缓解。

现在我们非正式地讨论一下提取左因子问题。考虑上面的<变量>规则，其中两个RHS都由标识符开始，在两个RHS中跟随标识符后面的部分为ε（空字符串）和[<表达式>]。可以将这两条规则替换为

< 变量 > → 标识符 < 新的 >

这里 < 新的 > 被定义为

< 新的 > → ε | [< 表达式 >]

不难看出，这两条规则一起产生与开始时的两条规则相同的语言。然而，这两条规则通过了成对不相交测试。

如果要将这条文法作为递归下降语法分析器的基础，我们还可以提出避免提取左因子的一种替代方法，使用一种EBNF的扩展方法，可以产生非常类似于提取左因子的方案用于解决上面的问题。考虑上面的第一条<变量>规则。通过将下标放置于方括号之中，下标就可以成为可选择的，如

186

<变量> → 标识符 [| <表达式> ']

在这条规则中，外层方括号是元语言符号，说明括号内的内容是可选择的。内层方括号是被描述的程序设计语言的终结符。这里的要点是，我们使用一条规则代替了两条规则，这条规则产生的是相同的语言，并且通过了成对不相交测试。

提取左因子的形式算法能在Aho等人的文献（Aho *et al.*，1986）中找到。提取左因子不能够解决文法中所有的成对不相交问题。在某些情况下，必须采用其他的方式来重新编写文法规则，才能消除这个问题。

187

4.5 自底向上语法分析

本节介绍自底向上语法分析的一般过程，并且描述LR语法分析算法。

4.5.1 自底向上语法分析器的语法分析问题

考虑下面的文法：这个文法产生的算术表达式包括了加法和乘法操作符，括号以及操作数id。

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

请注意，这个文法与第4.4节的例子中的文法产生同样的算术表达式。它们的差别在于，这个文法是左递归的，它能够被自底向上的语法分析器所接受。也请注意，自底向上的语法分析器通常不包括元符号，例如那些用来说明BNF扩展的元符号。下面的最右派生给出了这种文法：

$$
\begin{aligned}
E &\Rightarrow \underline{E} + \underline{T} \\
&\Rightarrow E + \underline{T} * \underline{F} \\
&\Rightarrow E + T * \underline{id} \\
&\Rightarrow E + \underline{F} * id \\
&\Rightarrow E + \underline{id} * id \\
&\Rightarrow \underline{T} + id * id \\
&\Rightarrow \underline{F} + id * id \\
&\Rightarrow \underline{id} + id * id
\end{aligned}
$$

派生的句型中有下划线的部分都是RHS，将RHS重写为与之对应的LHS，以便获得前一个句型。自底向上的语法分析过程产生最右派生的逆过程。因而在上面的派生中，一个自底向上的语法分析器开始于最后句型（即输入句子），从那里开始产生一系列的句型，直到只剩下起始符为止。在这个文法里起始符为E。在每一个步骤中，自底向上语法分析器的任务是要找到句型中的某个RHS（句柄），而又必须将这个RHS重写，以便获得下一个（前一个）句型。前面讲过，一种右句型可以包括多个RHS。例如，右句型

188

$E + T * id$

包括了三个RHS，即E + T、T和id。在它们中间，只有一个RHS是句柄。例如，如果选择这个句型中的E + T进行重写，所产生的句型会是E * id，但是E * id对于给定的文法是不合法的右句型。

右句型的句柄是唯一的。自底向上语法分析器的任务是要找到任何给定右句型的句柄，它能够通过与之相关的文法来产生。句柄的形式定义为：

定义：β是右句型$\gamma = \alpha\beta w$的句柄，当且仅当$S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$。

在这个定义中，$\Rightarrow_{rm}$说明一个最右派生的步骤，$\Rightarrow^*_{rm}$则说明零个或多个最右派生的步骤。尽管句柄的定义在数学形式上很精确，然而这对于如何在给定的右句型中确定句柄却并没有帮

助。我们在下面给出与句柄相关的句型的子字符串的定义，目的在于提供一些对句柄的直觉。

定义：β是右句型γ的一个短语，当且仅当$S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow^+ \alpha_1\beta\alpha_2$

在这个定义中，$\Rightarrow^+$意味着一个或多个派生步骤。

定义：β是右句型γ的一个简单短语，当且仅当$S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1\beta\alpha_2$

如果仔细比较这两种定义，你会发现它们的差别只是在最后一项派生的说明。有关短语的定义使用了一个或多个步骤，而有关简单短语的定义却严格地仅使用一个步骤。

短语和简单短语的定义看起来似乎与句柄的定义一样缺乏实用价值，然而事实上却并非如此。让我们来考虑短语是怎样与语法分析树相关联的。短语是字符串，它包括一个子树中的所有叶节点，而这棵子树的根是整个语法分析树的一个内部节点。简单短语是以作为根的非终结节点只经过单个派生步骤产生的短语。就语法分析树而言，短语可以派生自位于语法分析树的单层或多层上的非终结，但一条简单短语就只可以派生于语法分析树的单层之上。考虑图4-3所显示的语法分析树。

189

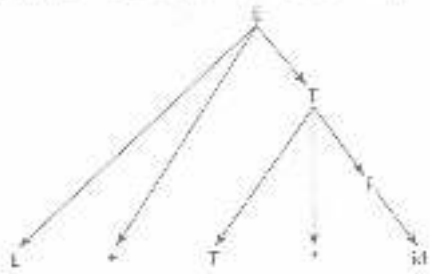


图4-3 E + T * id的语法分析树

图4-3中语法分析树的树叶包含句型E + T * id。因为这棵树具有三个内部节点，因此就有三条短语。每一个内部节点都是一棵子树的根，而一棵子树的所有树叶都是短语。整个语法分析树的根E产生的是完整句型E + T * id，它同时也是一条短语。内部节点T产生的是树叶T * id，它也是一条短语。最后的一个内部节点F产生短语id。所以句型E + T * id的三条短语分别为E + T * id、T * id和id。请注意，短语在分析文法中不必是RHS。

简单短语是短语的一个子集。在上面的例子中，id是唯一的简单短语，而简单短语总是文法中的RHS。

我们讨论短语和简单短语的理由是：任意最右句型的句柄是最左的简单短语。假设我们已经有了一个右句型的文法并且可以画出语法分析树，我们现在就有了一种高度直观的方法来找到这个右句型的句柄。对于语法分析器，以这种方式来寻找句柄当然是不切实际的（如果已经有了分析树，为什么还需要语法分析器？）。这里的唯一目的只是为了给读者提供一些直觉，以便理解相对于分析树而言什么是句柄。我们认为这种方法比试图用句型来找到句柄要更容易一些。

我们现在可以从语法分析树的角度来考虑自底向上的语法分析。尽管语法分析的目的是要产生语法分析树，如果给出整个句子的语法分析树，你就会很容易找到句柄。它是句子中第一个需要重新改写以使产生前一个句型的串。在改写之后，可以将句柄从语法分析树上修剪下来，并重复这一过程，一直进行到分析树的根，就能构成完整的最右派生。

4.5.2 移进-归约算法

自底向上的语法分析器常常也被称为**移进-归约算法**，因为移进（shift）与归约（reduce）是这种算法中最常用的两种运算。每个自底向上语法分析器不可或缺的部分是栈。移进运算将下一个输入标记移到语法分析器的栈上。归约运算则将语法分析器栈顶的RHS替换为与之对应的LHS。一个任意类型的语法分析器是一个**下推自动机**（pushdown automaton，PDA）。人们并不需要熟悉下PDA才能够理解一个自底向上的语法分析器是怎样工作的，当然熟悉PDA会有助于这种理解。PDA是一台非常简单的数学机器，它从左到右扫描符号串。之所以命名为下推自动机，是因为它使用下推栈作为它的存储器。可以使用PDA作为语言的识别器。当给定特定字符集合中的一个符号串，专门为识别目的而设计的PDA就可以确定这个符号串是否是某一种语

言中的句子。在这一过程中，PDA可以产生构造句子语法分析树所需要的信息。

可以使用PDA来检测输入字符串，从左到右，一次一个字符。这种输入处理的方式就好像这个字符串是储存于方一个栈上，因为PDA只能看见输入中的最左符号。

必须注意的是，递归下降语法分析器也是一个PDA。这种分析器的栈就是运行时系统的栈，
190 它记录了对应于文法中的非终结的子程序的调用（还有一些其他的信息）。

4.5.3 LR语法分析器

人们设计了许多不同种类的自底向上语法分析的算法，其中的大部分是一个被称为LR的过程的变种。LR语法分析器使用相对小的程序以及一个语法分析表。最初的LR算法由Donald Knuth设计（Knuth，1965）。这种算法有时被称为规范LR。这种算法在发布后的几年间都没有获得应用，因为产生语法分析表需要大量的计算机时间以及存储空间。后来，几种规范LR表构造过程的变体又被开发了出来（DeRemer, 1971; DeRemer & Pennello, 1982）。这些变体具有如下两种性质：(1) 产生语法分析表只需要比标准LR算法更少的计算机资源，(2) 能够将它们用于比规范LR算法更小型的文法上。

LR语法分析器有许多优点：

1. 所有的程序设计语言都能构造它们。

2. 在从左到右的扫描中，它们能够尽可能早地发现语法错误。

3. LR类文法是可由LL语法分析器进行语法分析的文法类型的超集（例如，许多左递归文法都是LR，但它们都不是LL）。

LR语法分析的唯一缺点是很难用手工方式产生给定文法的语法分析表。然而，这并不是一个严重的缺点，因为有许多程序以文法为输入，然后产生语法分析表。这将在本节后面讨论。

在LR语法分析算法出现之前，曾经有过一些语法分析的算法，这些算法通过检测可能为句柄的句型字符串的左右两边，来寻找右句型的句柄。Knuth的见解是，你可以有效地检测这个可能句柄的左边，一直到语法分析栈的底端为止，以便确定它到底是不是句柄。所有与语法分析过程相关的语法分析栈的信息可以由一个单个状态来代表，这个状态位于栈的顶端。换言之，Knuth发现就语法分析过程而言，不论输入字符串的长度如何，句型的长度如何，或者是语法分析栈的深度如何，只存在少数不同的情形，而每一种情形都可以由一个状态来代表，并将它储存于栈中。一个状态对应于栈中一个文法符号。栈的顶端总是一个状态符，它代表到目前为止的整个语法分析史的信息。我们将使用带有下标的大写字母S来表示语法分析器的状态。

191 图4-4显示一个LR语法分析器的结构。

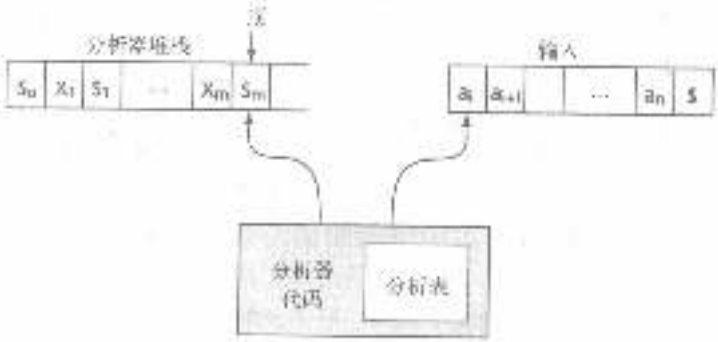


图4-4 LR语法分析器的结构

LR语法分析器的语法分析栈的内容具有下面的形式：

$S_0X_1S_1X_2\cdots X_mS_m$（顶端）

这里S为状态符号，X为文法符号。LR语法分析器结构是一对字符串（栈，输入），其具体形式为：

$(S_0X_1S_1X_2S_2\cdots X_mS_m, a_ia_{i+1}\cdots a_nS)$

注意，在输入字符串的右端有一个“S”符号，这个符号在语法分析器设定初值的期间就已经放置在那里，它被用于使语法分析器能够正常地终止。使用这种语法分析器的结构，我们就能够形式地定义基于语法分析表的LR语法分析器的过程。

一个LR语法分析表具有两个分别命名为ACTION和GOTO的部分。语法分析表的ACTION部分说明这个语法分析器所施行的大部分工作。它以状态符作为行的标号，以文法中的终结符作为列的标号。当给出当前语法分析器的状态，该状态由在语法分析栈顶的状态符代表，并给出下一个输入的符号（标记），语法分析表就会指示出语法分析器所应该执行的任务。语法分析器的两种主要动作就是移进与归约。它或者将下一个输入符号移进到语法分析栈上，或者已经具有了在栈顶的句柄，语法分析器则运用一条RHS与句柄相同的规则，将句柄归约为这条规则的LHS。语法分析器还可能具有另外两种行动，其一为接受，即语法分析器已经成功地完成了输入的语法分析；其二则为报告错误，即语法分析器发现了一个语法错误。

LR语法分析表的GOTO部分的行以状态符为标号，而语法分析表中这一部分的列则以非终结为标号。LR语法分析表的GOTO部分的作用是，在归约完成之后哪个状态符应该被推到语法分析栈上，这也就意味着已经将句柄从语法分析栈上移开，并且已经将新的非终结推到了语法分析栈上。当将句柄以及相关的状态符从栈顶移走之后，栈顶就有一个状态符。以这个状态符为行标号，就可以在表的GOTO部分找到要进入栈的状态符。而这个状态符所在列的标号就是用于这次归约的规则中的LHS。

考虑下面算术表达式的传统文法：

1. E → E + T
2. E → T
3. T → T * F
4. T → F
5. F → (E)
6. F → id

这个文法规则附有编号，以提供在语法分析表中引用它们的简单方式。

图4-5显示了这个文法的LR语法分析表。请注意，在这里我们使用了动作的缩写，并且使用数字来表示状态。R4意味着使用第4个规则进行归约；S6意味着将输入的下一个符号移进到栈上，并将第6个状态推到栈上。ACTION表中的空位置表示语法错误。在一个完整的语法分析器中，这里就应该出现对错误处理程序的调用。

使用把文法作为输入的软件工具可以很容易地构建LR语法分析表，例如yacc[○]（Johnson，1975），尽管LR语法分析表能手工产生，但是对于一种现实程序设计语言的文法来说，这个任务是漫长的、乏味的和漏洞百出的。在实际的编译器中，LR语法分析表总是通过软件工具来产生。

LR语法分析器的初始格局为

○ yacc是“另一种编译器的编译器”（yet another compiler-compiler）之英文缩写。

$(S_0, a \cdots a_n\$)$

语法分析器动作的正规定义如下：

1. 如果ACTION$[S_m, a_i]$ = 移进S，则下一个格局为

$(S_0X_1S_1X_2S_2\cdots X_mS_maS, a_{i+1}\cdots a_n\$)$

移进过程十分简单：将下一个输入符号连同一个状态符号一起推到栈上，这个状态符在ACTION表中是作为移进部分的说明。

例如，假定这个格局是$(S_0ES_1, +id\cdots\$)$，ACTION表指定S6作为它在[1, +]位置的动作，这产生了格局$(S_0E_1+S_6, id\cdots\$)$。

2. 如果ACTION$[S_m, a_i]$ = 用$A \to \beta$以及 S = GOTO$[S_{m-r}, A]$，这里r = β的长度，则下一个格局为

$(S_0X_1S_1X_2S_2\cdots X_{m-r}S_{m-r}AS, a_ia_{i+1}\cdots a_n\$)$

这是一个更为复杂的动作。为了归约，必须将句柄从栈里移走，因为栈里的每一个文法符号都对应一个状态符，从栈里移走的符号数目是句柄中符号数目的两倍。在移走句柄及其相关的状态符之后，规则中的LHS被推入栈中。最后，以移走句柄及其相关的状态符后所暴露出来的符号为行的标号，并以归约中使用的规则里LHS的非终结为列的标号，来查询GOTO表。因而生新的格局中，栈顶的符号来自表里的GOTO部分。而且最顶端的文法符号是用于归约的规则中的LHS。

例如，假定这个格局是$(S_0idS_5, +id\cdots\$)$，ACTION表在[5, +]位置上指定R6。这表示使用了规则6归约（F→id）。这个规则的RHS有1的长度，因此两个符号必须从栈中弹出，S_5就出现在栈的顶部。因此我们就看到了GOTO表的0行F列（因为F是用在归约规则中的LHS）。在GOTO表的那个位置，我们发现3，因此在把F压入栈后再压入S_3。

3. 如果ACTION$[S_m, a_i]$ = 接受，则语法分析已经完成，并且没有发现错误。

4. 如果ACTION$[S_m, a_i]$ = 错误，语法分析器此时就会调用一个错误处理程序。

192
-
194

尽管在在许多基于LR概念的语法分析算法，但它们之间的差别仅仅是语法分析表的构造。所有的LR语法分析器都使用相同的语法分析算法。

也许，熟悉LR语法分析过程的最好方法是使用实际的例子。初始时，语法分析栈中只有单个符号0，代表语法分析器的0状态。而输入则包括将一个底部标记附于右端结尾处的输入串。在我们的这个情形中采标是"$"符号。语法分析器动作的每一个步骤都由栈顶的符号（图4-4中最右端的符号）以及下一个输入标记（图4-4中最左端的标记）来指导。正确的动作来自语法分析表中ACTION部分相对应的单元。而语法分析表中GOTO部分则在归约动作之后才采用。前面讲过，表的GOTO部分是用来决定在一个归约动作之后，应该将那个状态符放置于语法分析栈上。

下面是运用LR语法分析算法以及图4-5中的语法分析表来跟踪于符号id + id * id的语法分析。

栈	输入	动作
0	id + id * id$	移进5
0id5	+ id * id$	归约6（使用GOTO[0, F]）
0F3	+ id * id$	归约4（使用GOTO[0, T]）
0T2	+ id * id$	归约2（使用GOTO[0, E]）
0E1	+ id * id$	移进6
0E1+6	id * id$	移进5
0E1+6id5	* id$	归约6（使用GOTO[6, F]）
0E1+6F3	* id$	归约4（使用GOTO[6, T]）

（续）

栈	输入	动作
0E1*6T9	*id$	移进7
0E1*6T9*7	id$	移进5
0E1*6T9*7id5	$	归约6（使用GOTO[7，F]）
0E1*6T9*7F10	$	归约3（使用GOTO[6，T]）
0E1*6T9	$	归约1（使用GOTO[0，E]）
0E1	$	接受

	动作						Goto		
状态	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

图4-5 一个算术表达式文法的LR语法分析表

Aho等人的文献（Aho *et al.*，1986）描述了从给定文法产生LR语法分析表的一些算法。这些算法并不是很复杂，但它超出了程序设计语言类书籍的讲述范围。正如前面提到的，有许多可用来产生LR语法分析表的软件系统。

小结

无论使用什么样的实现方式，语法分析是所有语言实现中的共有部分。语法分析通常是基于被实现语言的形式语法描述。最常用的语法描述机制是上下文无关文法，也称为BNF。语法分析的任务通常被分为两个部分，即词法分析部分以及语法分析部分。将词法分析分离出来有几个重要理由，它们是简单性、高效率以及可移植性。

195

词法分析器就是一个模式匹配器。它从程序中将称为词素的部分分离出来。词素的类别有整数字面量或名字。这些种类也被称为标记。对于每一个标记分配一个数值码。词法分析器在识别词素的同时也产生这些数值码。标准文法分析器有三种不同的方式：其一，使用软件工具为表驱动分析器来产生一个表格；其二，手工经过这样的表格；其三，编写代码来产生被实现语言标记的一个状态图描述。如果是使用字符类来决定状态的转换，而不是使用来自每个状态节点的可能的字符来决定状态的转换，那么标记的状态图就可能相应地小些。另外，使用表格查询来识别保留字，可以简化状态图。

语法分析器具有两个目的：即，在给定的程序中发现语法错误，以及产生语法分析树，或者是，对于给定的程序产生构造语法分析树所必需的信息。语法分析器可以是自顶向下的（这意味着它们构造或在

派生，接自底向下的顺序产生语法分析树）或者自底向上的（在此情形中它们构成最右派生的逆向，按自底向上的顺序产生语法分析树）。用于所有非及文生文法的语法分析器，复杂性为$O(n^3)$。然而，为程序设计语言实现语法分析器并且用于处理非及文生文法的子类的语法分析器，就具有复杂性$O(n)$。

递归下降语法分析器是LL语法分析器。它的实现是通过直接从源语言文法来编写代码。EBNF作为递归下降语法分析器的基础是十分恰当的。递归下降语法分析器中，对于文法中的每一个非终结都有一个对应的子程序。如果原始的一条文法规则只有单个RHS。那么这条规则的代码就很简单。对RHS进行的处理是从左至右，对于每一个非终结，这种代码都将调用与这个非终结相关联的子程序。每个子程序又将对非终结产生的内容进行语法分析。对于每一个终结符，这种代码将这个终结符与下一个输入标记进行比较，如果它们匹配，这段代码将只消用语法分析器来得到下一个标记；如果它们不相匹配，子程序将报告一个语法错误。如果一条规则具有多个RHS，该子程序必须首先确定应该对哪个RHS进行语法分析，它必须能够基于下一个输入的标记来做出这项决定。

有两个不同的文法特征妨碍构造基于文法的递归下降语法分析器。其中之一就是左递归。从文法中消除直接左递归的过程是相对简单的。有一种算法能够从文法中删除直接和间接的左递归，然而我们不会在书中讨论这种算法。另一个则是进行成对不相交测试，这种测试能够判断一个语法分析器的子程序能否基于下一个输入标记来确定应该对哪个RHS进行语法分析。运用提取左因子的方法能够对一些文法进行修改，以使得这些文法能够通过成对不相交测试。

自底向上语法分析器的语法分析问题是要找到当前句型的子串，必须将这个子串归约到与之相关的LHS，以使在最右派生中获得下一个（先前的）句型。这种子串被称为句型的句柄。能够使用语法分析树来描述句柄的定义。自底向上语法分析器是一种移进-归约算法，因为这种语法分析器在大多数情况下，或者是将下一个输入的词素移进到语法分析栈上，或者是归约栈上找到的句柄。

移进-归约语法分析器的LR家族是程序设计语言中最普遍应用的自底向上语法分析的方式，这为LR家族中的语法分析器比其他方式更具优越性。LR语法分析器使用语法分析栈，这种语法分析栈包括文法符号和状态符号，以保持语法分析器的状态。在语法分析栈顶的符号总是一个状态符，它代表了语法分析栈与语法分析过程相关的所有信息。LR语法分析器使用两种语法分析表，即ACTION表和GOTO表。ACTION表说明了给出了语法分析栈顶的状态符以及下一个输入标记之后，语法分析器所应该执行的任务。GOTO表则被用来决定产生归约完成之后应该将哪个状态符放置于语法分析栈上。

复习题

1. 语法分析器以文法为基础的一个理由是什么？
2. 给出词法分析与语法分析分离开来的一个理由。
3. 给出词素与标记的定义。
4. 词法分析器的主要任务是什么？
5. 简洁描述建造一个词法分析器的三种方法。
6. 什么是状态转换图？
7. 为什么词法分析器状态图的转换使用字符类而不使用每个字符？
8. 语法分析有哪两个不同的目标？
9. 描述自顶向下和自底向上两种语法分析器之间的差别。
10. 描述自顶向下语法分析器的语法分析问题。
11. 描述自底向上语法分析器的语法分析问题。
12. 解释编译器使用的语法分析算法为什么只能作用于所有文法的一个子集。
13. 为什么标记的实现使用命名常量而不使用数字？
14. 描述对一条具有单个RHS的语法规则，如何编写递归下降语法分析子程序。
15. 文法的哪两个特征阻碍了这种文法被直接应用于递归下降语法分析器的基础？请给出解释。
16. 什么是给定文法和句型的FIRST集合？

17. 描述成对不相交测试。
18. 什么是提取左因子？
19. 什么是句型的短语？
20. 什么是句型的简单短语？
21. 什么是句型的句柄？
22. 自顶向下以及自底向上这两种语法分析器是基于什么数学机器？
23. 描述LR语法分析器的三大优点。
24. 在开发LR语法分析技术中，Knuth的贡献是什么？
25. 描述LR语法分析器中ACTION表的目的。
26. 描述LR语法分析器中GOTO表的目的。
27. 对于LR语法分析器，左递归是一个问题吗？

练习题

1. 对下面的文法规则进行成对不相交测试。
   a. A → aB | b | cBB
   b. B → aB | bA | aBb
   c. C → aaA | b | caB
2. 对下面的文法规则进行成对不相交测试。
   a. S → aSb | bAA
   b. A → b{aB} | a
   c. B → aB | a
3. 使用第4.4.1节中的递归下降语法分析器分析字符串 a + b * c，并且写出对于语法分析的跟踪。
4. 使用第4.4.1节中的递归下降语法分析器分析字符串 a * (b + c)，并且写出对于语法分析的跟踪。
5. 使用下面给出的文法，给每一个右句型画出一棵语法分析树，并且表示出短语、简单短语以及句柄。
   S → aAb | bBA A → ab | aAB B → aB | b
   a. aaAbb
   b. bBab
   c. aaAbBb
6. 使用下面给出的文法，给每一个右句型画出一棵语法分析树，并且表示出短语、简单短语以及句柄。
   S → AbB | bAc A → Ab | aBB B → Ac | cBb | c
   a. aAccbbc
   b. AbcaBccb
   c. baBcBbbc
7. 对于串 id * (id + id)，使用第4.5.3节中的文法和语法分析表显示完整的语法分析，包括语法分析栈的内容、输入字符串以及动作。
8. 对于串 (id + id) * id，使用第4.5.3节中的文法和语法分析表显示完整的语法分析，包括语法分析栈的内容、输入字符串以及动作。
9. 写出一条EBNF规则来描述Java或C++语言中的while语句，并且使用Java或C++写出这条规则的递归下降子程序。
10. 写出一条EBNF规则来描述Java或C++语言中的for语句，并且使用Java或C++写出这条规则的递归下降子程序。

程序设计练习题

1. 设计一个状态图来识别基于C的程序设计语言的一种注释形式。这种注释开始于"/*"，结束于"*/"。

2. 设计一个状态图来识别你所喜欢的程序设计语言中的浮点文字常量。
3. 编写并测试程序设计练习题1中状态图的实现代码。
4. 编写并测试程序设计练习题2中状态图的实现代码。
5. 使用Java语言来改写4.2节中的词法分析器（原来是用C编写的）。
6. 对于能够通过练习题1中测试的那些规则编写一个递归下降语法分析子程序，来对这些规则产生的语言进行语法分析。假设，你有一个被命名为lex的词法分析器以及一个被命名为error的错误处理子程序，当检测到语法错误时，则调用错误处理子程序。
7. 对于能够通过练习题2中测试的那些规则编写一个递归下降语法分析子程序，来对这些规则产生的语言进行语法分析。假设，你有一个被命名为lex的词法分析器以及一个被命名为error的错误处理子程序，当检测到语法错误时，则调用错误处理子程序。
8. 实现并上机测试在4.5.3节中给出的LR语法分析算法。

第5章 名字、绑定、类型检测和作用域

本章介绍变量的基本语义问题。我们将首先讨论这些题目中最基本的课题，即程序设计语言中名字及特殊字的性质。然后讨论变量的属性，包括变量类型、变量地址和变量值。别名问题也包括在讨论之中。接着再介绍绑定和绑定时间的重要概念，由于对变量属性所具有的可能不同的绑定时间，从而定义出了四种不同的变量类型。在描述了这四种变量类型之后，是对于类型检测、强类型化以及类型兼容规则的深入研究。两种不同名字的作用域规则，即静态与动态的作用域，将与语句引用环境的概念一起讨论。最后将讨论命名常量以及变量的初始化。

5.1 概述

在不同的程度上，命令式程序设计语言是冯·诺依曼计算机体系结构的抽象。计算机体系结构中的两个主要组成部分为：存储器——存储指令与数据；处理器——提供修改存储器内容的一些操作。变量是机器存储单元在语言中的抽象。在某些情况下，这种抽象的特征非常接近存储单元的特征；整数变量就是一个例子，整数变量通常完全由单个或多个硬件存储字节来代表。而在其他的情况下，这种抽象又与硬件存储单元相去甚远，例如一个三维数组的情形，则要求一个软件映射函数来支持这种抽象。

变量可以由一组性质或属性来概括，在这些属性中最重要的就是类型，它是程序设计语言的一项基本概念。一种语言的数据类型设计需要考虑多种因素（数据类型将在第6章中讨论），其中最重要的因素是变量的作用域以及生存期。与这两个因素相关联的是变量的类型检测以及变量的初始化问题。类型兼容是语言数据类型设计中的另一个重要部分。

所有这些概念对于理解命令式语言十分必要。

在本书的余下部分，我们时常会以引述一种语言的方式来引述该语言的家族，例如，当提及Fortran时，我们指的是Fortran语言的所有版本，这同样也适用于引述Ada时的情形，当引述C语言时，包括C的原始版本以及C89和C99版本。我们用“**基于C的语言**”来指C、C++、Java以及C#语言。[1] 如果我们特别地引述一种语言的某个特定版本时，这是因为就我们所讨论的课题而言，它与该语言家族的其他成员不同。

201
~
202

5.2 名字

在开始关于变量的讨论之前，我们必须讨论变量的一个基本属性——名字。名字不只是用于变量，它具有更广泛的用途。名字也与标号、子程序、形参以及其他程序结构相关联。标识符常常与名字一词互换使用。

5.2.1 设计问题

下面是关于名字的主要设计问题：

- 名字是否大小写敏感？
- 特殊字究竟是保留字还是关键字？

[1] 我们试图包括脚本语言JavaScript和PHP作为基于C的语言，但是与它们的相关性，它们显得难了点。

这些问题将在下面的两节中讨论，这两节也包括了一些设计选择的例子。

5.2.2 名字形式

名字是用来标识程序中某些实体的字符串。

历史注释

早年的程序设计语言使用单个字母的名字，这种记法十分自然，因为早期的程序设计主要是数学方式的，而数学家们长期习惯于在形式化标记中对未知数使用单个字母名字。

Fortran I打破了使用单个字母名字的传统，它允许名字使用多至6个字符，而Fortran 77仍然将名字限制于6个字符。

Fortran95允许在一个名字中有多达31字符的名字。C89对于它的内部名字没有长度的限制，但名字中只有前31个字符具有意义。C99类似于C89，但它的名字中前63个字符具有意义。在C89中，外部名字（那些定义于函数之外，必须使用语言链接器来处理的名字）被限制为6个字符。而在C99中，这种限制增加到了31个字符。名字在Java、C#以及Ada中都没有长度限制，并且这些语言的名字中所有字符都是具有意义的。然而，Ada语言的实现允许施加一种长度限制，但这种限制是不少于200个字符。显然，这种限制并不令人讨厌。C++不指明名字的长度限制，但有些时候，语言的实现人员会施加某种限制，这样做是为了使编译期间存储标识符的符号表不至于太大，并且还可以简化符号表的维护工作。

在大多数程序设计语言中，名字具有相同的形式：一个字母后跟一个包含字母、数字以及下划线（_）的字符串。在20世纪70年代及80年代，包括下划线的名字使用得很广泛，但这种用法现在已经很少了。在基于C的语言中，下划线已经在很大程度上被“骆驼”形式的标记所取代。所谓“骆驼”形式，就是在一个多单词的名字中，第二个单词的第一个字母用大写，例如myStack。[⊖]注意，在名字中使用下划线和混合大小写是程序设计风格问题，不是语言设计问题。

203

历史注释

在Fortran90之前的版本中，只有大写字母可以用于名字，这是一种不必要的限制，这种限制源于当时的卡片打孔机只有大写字母。像Fortran 90一样，Fortran 77的多种实现都允许小写字母，然而在编译前同义的小写字母都翻译成大写字母表供编译器内部使用。

在Fortran 90之前的各种Fortran版本中，名字的中间可以嵌入空格，而这些空格将会被忽略。例如，下面的这两个名字就是等价的：

```
Sum Of Salaries
SumOfSalaries
```

在许多语言中，尤其是在基于C的语言中，名字中的大写字母与小写字母是有区别的，也就是说，在这些语言中的名字是大小写敏感的。例如，在C++中，下面的这三个名字是不同的：rose，ROSE，Rose。在一些人看来，这严重地损失了语言的可读性，因为看上去十分相像的名字实际上却表示不同的实体。从这种意义上来说，大小写敏感违反了语言构造的基本设计原则，这个原则要求，看似一样的事物应该具有相同的意义。对于变量名不区分大小写的程序设计语言来说，虽然Rose和rose看起来很接近，但事实上一点联系也没有。

显然，并不是每个人都认为名字大小写敏感是一件坏事。在C语言中，大小写敏感的问题通过只在名字中使用小写字母而得以避免；然而在Java和C#中，这个问题就不能够避免，因为许多预定义的名字包括了大写字母和小写字母。例如，Java中将一个字符串转换成整数值的方法是parseInt，如果将其拼写为ParseInt或者parseint，就不能够被识别。这是可写性的

⊖ 之所以被称为“骆驼”，是因为用这种形式写出的字符串在中间嵌入了大写字母，看起来很像是驼峰的样。

问题，而不是可读性的问题，因为要记住独特的拼字方法会使正确编写程序更困难。这是由编译器强制的，是对部分语言的设计人员的某种严律性。

5.2.3 特殊字

程序设计语言中的特殊字用于对所要进行的操作命名，从而使得程序的可读性更好。特殊字也用来分离程序中的语法实体。大多数语言将特殊字归为保留字，但在某些语言中它们仅仅是关键字。

关键字是程序设计语言中的一种字，它只在特定的上下文里是特殊的。Fortran语言中的特殊字就是关键字，在Fortran中，当字Real出现于语句的开头，并且后面跟随一个名字时，就认为它是一个关键字，它示意这条语句是一条声明语句。但如果在Real的后面跟随一个赋值操作符，则认为它是一个变量名。下面举例来说明这两种不同的用法： 204

```
Real Apple
Real = 3.4
```

Fortran编译器以及Fortran程序的读者必须通过上下文来识别这种名字与特殊字之间的差别。

保留字是程序设计语言中的特殊字，它不能用作名字。作为语言设计的选择，保留字比关键字优越，因为重新定义关键字的功能会导致混淆，例如，在Fortran中，可以有下面的语句：

```
Integer Real
Real Integer
```

这里声明程序变量Real为整数（Integer）类型，而变量Integer为实数（Real）类型。[5] 除了这两条声明语句的奇异外表之外，Integer和Real作为变量名出现在程序中其他部分，会给程序读者以误导。

保留字有个潜在的问题：如果语言包含了大量的保留字，用户很难确定名字不是保留的，关于这个问题，最好的例子是COBOL。它有300个保留字。不幸的是，程序员最常用到的一些名字出现在保留字列表中，例如，LENGTH、BOTTOM、DESTINATION和COUNT。

一些语言包括了预定义的名字。在某种意义上，这种名字位于保留字与用户定义名字之间。它们既具有预定义的含义，又能够被用户重新定义。例如，Ada中内置数据类型的名字（如Integer和Float）都是预定义的。这些名字不是被保留的，任何Ada程序都能够将其重新定义。

在大多数语言中，在其他程序单元中，如Ada和Java语言中的包以及C和C++语言中的库，定义的名字只对某一个程序可见。它们是预定义的名字，但必须在显式导入之后才可见。而一旦导入之后，这些名字就不能再被定义。

5.3 变量

程序的变量是计算机的存储单元或对计算机一系列存储单元的抽象。程序人员常常认为变量是存储地址的名字，但是变量比仅是名字具有更重大的意义。从机器语言到汇编语言的发展，在很大程度上是用名字替代绝对数字的存储地址的发展，这使得程序可读性更好，因而也就更容易编写和维护，因为将名字转换成实际地址的翻译器也选择这些地址，所以这一步回避了绝对地址的问题。 205

可以使用6种属性来刻画一个变量：名字、地址、数值、类型、生存期、作用域。尽管这对于一种看似简单的概念似乎太过复杂，但正是这6种属性提供了一种最清晰的方式来解释变量的

[5] 当然，任何专业的程序人员名编写出这样的程序，就应当担心自己的饭碗不保。

各个方面。

关于变量属性的讨论将会导致对许多相关重要概念的研究，包括别名、绑定、绑定时间、声明、类型检测、强类型化、作用域规则以及引用环境。

关于变量的名字、地址、类型以及数值，将在下面的几节讨论。关于生存期与作用域属性，将在5.4.3节和5.5节分别讨论。

5.3.1 名字

变量名是程序中最常见的名字。我们已经在第5.2节中讨论过。大多数的变量具有名字。没有名字的变量将在5.4.3.3节中讨论。

5.3.2 地址

变量的地址是与这个变量相关联的存储地址。然而这种关联关系却不是看上去那么简单。在许多语言中，程序中的相同名字可以在不同的时间，与不同的地址相关联。例如，如果子程序有一个局部变量（当调用子程序时，它从运行时栈分配而来），不同的调用可能会导致变量有不同的地址。从某种意义上说，它们是同一变量的不同实例。

变量与地址相关联的过程将在第5.4.3节中进一步讨论。有关子程序以及子程序激活的一个实现模型将在第10章中讨论。

右对，也将变量地址称为变量的左值（l-value），这是因为变量通常位于赋值语句的左边。

历史注释

Fortran语言中的Equivalence语句是专门为构造别名而设计的。当时设计Equivalence语句的动机之一是为了节省存储空间。由于Fortran现在已经有了动态存储，再加上现在的存储空间已经相对丰富，当前已没有任何理由使用Equivalence语句。结果，Fortran 90中就除了Equivalence语句。

多个变量可以具有同一个地址。当用多个变量名来访问单个存储地址时，这些变量名就称为别名。别名使用有损于可读性，因为它允许通过给一个变量赋值来改变另一个变量的值。例如，如果变量total和sum为别名，total的任何变化就会改变sum，反之亦然。程序的读者必须时刻记住，total和sum是同一个存储单元的不同名字。因为在一个程序中可以具有数个别名，所以实际使用时非常困难。别名使用也使得程序的验证更为困难。

206

在程序中可以用多种不同的方式产生别名。在C和C++语言中产生别名的常用方式是使用联合（union）类型。关于联合，我们将在第6章进行深入讨论。

指向同一存储地址的两个指针变量也是别名。对于引用变量也有同样的情形。这种类型的别名使用不过是指针与引用特性的一种副作用。设定一个C++指针来指向一个命名变量后，当指针被间接引用时，这个指针以及变量名都是别名。

在许多语言中，通过子程序的参数也可以产生别名使用。关于这种类型的别名，我们将在第9章中讨论。

变量与地址相关联的时间对于理解程序设计语言是非常重要的。关于这个题目，将在第5.4.3节讨论。

5.3.3 类型

变量的类型决定变量可以存储的值的范围，并决定为这种类型的值所定义的操作集合。例如Java中的int类型说明从−2 147 483 648 到2 147 483 647的取值范围，以及一组包括加法、减

法、乘法、除法以及取模的算术运算。

5.3.4 值

一个变量的值是与这个变量相关联的存储单元的内容。我们可以很方便地将计算机的存储器设想成抽象的单元，而非物理的单元。大多数现代的计算机存储单元或单个地址单元，是以字节为单位的，一个字节具有八个位长度。对于大多数的程序变量，这样的大小还是太小，因而我们定义一种抽象的存储单元，它具有相关变量所需要的大小。例如，当然在某种特定语言的特定实现中，浮点数值可能占据四个物理字节，但我们可以设想，一个浮点数值只占有一个抽象存储单元。我们也可以认为，每一个简单非结构类型的值只占有一个抽象单元。自此往后，我们使用术语存储单元时，指的是抽象存储单元。

变量的值有时被称为变量的右值（r-value），因为变量被用于赋值语句的右边时，则会要求这种变量值。要取得右值必须首先确定左值。获得这种确定性并非总是简单的。例如，在5.8节将要讨论的作用域规则会使情况非常复杂。

5.4 绑定的概念

在一般的意义上，绑定是一种关联，如存在于属性与实体之间的关联，或者操作与符号之间的关联。绑定发生的时间称为绑定时间。绑定和绑定时间是程序设计语言语义中的重要概念。绑定可以发生在语言设计时，语言实现时、编译时、载入时、连接时或者运行时。例如，星号（*） 207
通常在语言设计时与乘法操作相绑定。一种数据类型，如C语言中的int，是在语言实现时与可能的取值范围相绑定的。Java程序中的变量是在编译时被绑定于某种特定的数据类型的。而当程序被载入存储器时，一个变量与一个存储单元相绑定。但在某些情况下，这种类型的绑定直到运行时才发生，如在Java方法中声明的变量。一个对库子程序的调用是在链接时与该子程序相绑定的。

考虑下面的C赋值语句：

```
count = count + 5;
```

对于这条赋值语句中的部分，其绑定与绑定时间如下所示：

- count的类型：绑定于编译时。
- count的可能取值集合：绑定于编译器设计时。
- 操作符+的意义：绑定于编译过程中，当操作数的类型被确定之后。
- 字面常量5的内部表示：绑定于编译器设计时。
- count的值：绑定于这条语句被执行时。

彻底地理解程序实体属性的绑定时间是理解程序设计语言语义的前提。例如，要理解一个子程序的作用，必须先理解子程序调用中的实参是怎样与子程序定义中的形参相绑定的。要确定一个变量的当前值，就需要知道这个变量是在什么时候与存储单元绑定的。

5.4.1 属性与变量绑定

如果一种绑定的第一次出现是在运行时之前，并且在整个程序的执行过程中保持不变，我们称这种绑定为静态的。如果一种绑定的第一次出现是在运行时期间，或者这种绑定可以在程序执行中被改变，我们称这种绑定为动态的。在虚拟存储器环境中，变量与存储单元的物理绑定是十分复杂的，因为存储单元所在的地址空间的页或段在程序的执行期间可以被多次地移进与移出存储器。在某种意义上，这些变量是被重复地绑定与释放。然而这些绑定是由计算机硬

件来维持的，无论是程序还是用户都感觉不到这些变化。这种硬件的绑定不是我们关心的话题。
208 我们最需要了解的是区分静态绑定与动态绑定。

5.4.2 类型绑定

在程序中引用一个变量之前，变量必须被绑定到一种数据类型之上。这种绑定具有的两个重要方面是指定类型的方式以及绑定发生的时间。可以通过一些显式或者隐式的声明来静态指定变量的类型。

5.4.2.1 静态类型绑定

显式声明是程序中的一条说明语句，它列出一批变量名并指明这些变量的特定类型。隐式声明则是通过默认协定的方法而不是声明语句将变量与类型相关联。在隐式声明的情况下，变量名在程序中的第一次出现即构成了它的隐式声明。显式声明与隐式声明都产生对类型的静态绑定。

自20世纪60年代中期以来设计的大多数程序设计语言，要求对所有的变量都予以显式声明(Perl、JavaScript和ML是这些语言中的三个例外)。一些于20世纪60年代末期开始设计的广泛应用的语言，特别是Fortran、PL/I和BASIC语言中都具有了隐式声明。例如，在Fortran中，一个出现于程序中但没有被显式声明的标识符，是根据下面的协定来隐式声明的：如果一个标识符以字母I、J、K、L、M或者N开始，或这些字母的小写字母它便被隐式地声明为Integer类型；否则，它就被声明为Real类型。

尽管隐式声明给程序人员带来了些许方便，但它却有损于程序的可靠性，因为它有碍于在编译过程中发现键入错误以及程序人员的编程错误。Fortran的程序人员不经意漏掉声明的变量会被自动地给予默认类型和一些未曾预料的属性，这会导致出现难以诊断的微妙错误。现在许多Fortran程序人员在他们的程序中包括了声明：implicit none。这种声明指示编译器不要隐式地声明任何变量，这面避免了由漏掉声明的变量可能带来的问题。

要避免隐式声明带来的问题，还可以通过要求特殊类型名字以某些特殊的字符开始的方式。例如，在Perl语言中，任何以$开始的名字都是一个标量，它能够存放一个字符串或者数字值；如果是以@开始的名字，它就是一个数组；名字以%开始的是一个散列结构。以这种方式来对不同类型的变量产生不同的名字空间，在这里，名字@apple与名字%apple是无关的，因为它们来自各自不同的名字空间。此外，程序的读者只要读到变量名，就必定知道了它的类型，请注意，这种规定与Fortran中的不同，因为Fortran语言包括了隐式及显式两种声明，因而从变量名
209 的拼法不一定能够确定一个变量的类型。

5.4.2.3节将讨论另一种隐式类型绑定——类型引用。

C和C++都有数据声明和定义。声明用来说明类型以及其他属性，并不引起存储空间的分配，定义则说明属性，并引起存储空间的分配。C程序对于某一给定的名字可以有任意数目相互兼容的声明，但是只有一个定义。C语言中变量声明的一个目的是为在函数外部定义供在函数内部使用的变量提供类型信息。这种思想也被用于C和C++的函数中，函数原型声明函数名字以及接口，但是没有包括函数的代码，而函数的定义则完整地定义了所有这些内容。

5.4.2.2 动态类型绑定

在使用动态类型绑定时，变量的类型不是由声明语句来说明的，也不能够通过名字的拼法来确定，而是在赋值语句给变量赋值时，变量才与类型相绑定。在执行赋值语句时，被赋值的变量与赋值语句右边的表达式的值的类型相绑定。

具有动态类型绑定的语言与那些只有静态类型绑定的语言有很大不同。变量与类型动态绑

定的主要优越性在于给程序设计提供了极大的灵活性。例如，在一种使用动态类型绑定的语言中，可以将一个处理一组数值数据的程序写成一个通用程序，这意味着它能够处理任何数值类型的数据。无论输入什么类型的数据都可以被它接受，因为当将输入数据赋给变量时，用来存储这些数据的变量能够与正确的类型相绑定。反之，如果只是使用静态类型绑定的话，在不知道数据的类型之前，是不可能编写一个Java程序来处理这组数据的。

在JavaScript以及PHP中，变量与类型的绑定是动态的。例如，一个JavaScript脚本可以包括下面的语句：

```
list = [10.2, 3.5];
```

不论这个命名为list的变量先前是什么类型，这项赋值将导致list成为一个长度为2的一维数值元素数组。如果赋值语句为

```
list = 47;
```

在赋值之后，list就成为一个标量变量。

然而动态类型绑定有两个缺点。首先，它使程序变得较不可靠。因为相对于具有静态类型绑定语言的编译器而言，前者的编译器发现错误的功能较弱。动态类型绑定允许将任何变量赋以任何类型的值，在赋值语句右边的错误类型就不能够被发现；反之，它会将左边的类型改为错误的类型。例如，假设在一个JavaScript程序中，i和x为当前存储的标量数量值，而y当前存储一个数组。再进一步假设这个程序需要赋值语句 210

```
i = x;
```

但是由于键入错误，变成了赋值语句

```
i = y;
```

在JavaScript或任何一种动态类型绑定语言中，解释器都不会发现这条语句中的错误，仅仅是将i改为一个数组而已，但在后来使用i时，因为期待i为一个标量，否则不可能得到正确的结果。在具有静态类型绑定的语言中，例如Java，编译器会发现这个错误，因而程序不会被执行。

请注意，这种缺点也不同程度地出现于一些使用静态类型绑定的语言中。如Fortran、C以及C++，在许多情况下，这些语言自动将赋值语句中RHS的类型转换成LHS的类型。

也许，动态类型绑定最大的缺点是它的代价。实现动态属性绑定的代价是相当可观的，特别是在运行时，类型检测必须在运行时进行。此外，每一个变量必须有一个相关的运行时描述符来维持当前类型，变量值的存储空间必须是大小可变的，因为不同类型的数值需要不同大小的存储空间。

最后，具有变量的动态类型绑定特性的语言，通常用单纯的解释器而不是编译器来实现，因为在编译时，计算机指令的操作数类型必须为已知。然而，如果A和B的类型在编译时是未知的，那么编译器不能为表达式A+B构建机器指令。而单纯的解释器的执行比等价的机器码至少要多花费十倍的时间。当然，如果一种语言是使用单纯的解释器来实现的，它用于动态类型绑定的时间被隐藏在总的解释时间中，因而在这种环境下没有显出那么高的代价。另一方面，使用静态类型绑定的语言很少通过单纯的解释器来实现，因为将这种语言编写的程序翻译成高效率的机器码版本十分容易。

访谈



脚本语言以及其他灵活解决方案的例子

RASMUS LERDORF

Rasmus Lerdorf于1968年出生在格林兰海岸线的Disko岛。在获得工程学位之后，Rasmus Lerdorf曾经就职于几个咨询公司。后来为了追踪上网阅读他个人简历的访问者，Rasmus Lerdorf开发了第一种PHP软件。现在他是开源运动的倡导者，同时就职于美国加利福尼亚州Sunnyvale的Yahoo公司。

有关背景

问：你早期有哪些有关计算机的经验？

答：1976年，我父亲和我一起使用从美国circa预订来的一套零件装配了一个称为"Pong"的游戏。我还记得大约是在1978年，我得到了德州仪器公司的一个"说话与拼音"设备。这个设备装载了有史以来的第一个单芯片语音合成器。1983年左右，我有了自己的第一台计算机。这是一台Commodore Vic20型计算机，具有5K的内存和1MHZ的6502 CPU。我当时耗费了大量的时间来键入从杂志上摘录的程序。在高中时，我曾经玩过Commodore的宠物（PET）游戏机，还有运行于QNX操作系统上的UNISYS 80186游戏盒。QNX是到目前为止我所见过的最"棒"的操作系统。它对我在后来得到微软的MS-DOS游戏盒起到了些很降临的作用。我认为自己早期的经验使得我更倾向于UNIX以及类似UNIX的操作系统。

问：你最喜欢过去的哪一份工作？

答：我非常喜欢在丹麦的时光。当时的公司在加利MV设立了办公室，因而是他们最早向我介绍了硅谷。我最终于1993年搬到了那里。我也很喜欢曾经在多伦多大学的工作。当时帮助他们建立了一个拨号系统。有关PHP工作的大部分内容就是在这段工作期间创建的。

关于脚本语言的工作

问：你关于脚本语言的定义是什么？

答：一种能传统程序设计技术解决特殊类型问题中的繁琐与复杂性的高级语言。

问：当你制作你的个人网页时，想要追踪上你个人网站的访问者，后来为多伦多大学的系统工作时，又想追踪上网站的大学学生，当时你考虑使用哪些可能的工具？

答：我不记得曾经使用过任何当时现有的解决办法。在当时你通常仅能匆读服务器上的原始访问"Log"文件。当然，当时也有一些多半是用Perl语言编写的"Log"文件的分析工具用来给出某些方面的总结。但当时我所要的是，当每一次访问者阅读我的简历时，我能够获得一个电子邮件，并且想要知道这个访问者来自哪里。当时没有工具专门进行这些工作。

问：是什么促成你做出自己花时间开发解决办法的决定，而不是使用当时现有的工具？

答：当时最主要的替代工具是Perl语言。尽管这些年来人们将一些东西归功于我，但我实际上并不擅长Perl语言。我喜欢它，并曾经大量地使用过它，但当时我的需求十分简单。我当时在一个与人共享的服务器上运行着我的个人网页，我没有大量的存储空间或CPU时间，为每一次的查询来回调用Perl的公共网关接口太耗费资源。我所需要的只是一个嵌入到我的Web服务器的简单语法分析器，而使用Perl程序进行这种嵌入太过复杂，也太过庞大，况且我当时从事的工作并不需要Perl语言的任何功能。因而我编写了一个小型、简单并且当时我认为是容易嵌入的语法分析器。当然它从此往后开始变大，一旦你开始给它增加任何可编的逻辑流程，它就不知不觉地滑向了完整的程序设计语言。但我的初衷绝对不是编写一个完整的语言。

问：PHP语言今天能够提供哪些其他脚本语言（Perl、Tcl、Python、Ruby）所不能提供的功能？

答：PHP语言是特别针对网络问题的。你能够访问到的关于PHP语言的所有一切都是针对网络的，因此，如果你试图解决网络上的问题，这是绝对决定于你怎样应用PHP语言。这不像其他较通用的语言那么清楚：你首先选择语言，然后再围绕特定的语言研究出最好的应用方式。

问：关于PHP语言的前景，在代码方面的下一个步骤是什么？或者说，你想在下一步应该增加或者应该完善的功能是什么？

答：我们需要大量高质量的PHP语言的代码以及语言的扩展，因为在我们以前建立PHP语言的一些时并没有质量与完善的尺度。

关于灵活解决办法

问：这里是你在以前的访谈中所谈到的："我肯定是赞成私密商对于困难问题给予灵活解决办法的。"是什么样的思路以及什么样的行动，导致了这种灵活解决的办法？

答：解决问题的办法的关键是从不同的角度来思考。有时候，正规训练以及传统观念可能会限制和束缚人们的想象力。你还要有经历多次失败的准备。

一个问题之所以艰巨，是因为你还没有找到解决问题的方法。这有点像你在纸上看到的拼字游戏。你读着这些乱七八糟的字母，它们不具有任何意义。你试了又试，想将这些字母正确地组合起来，但百思不得其解。然后有人能够告诉了你一个单词，一瞬那间，一切就变得如此地显而易见。现在当你再来看这些字母时，那个单词分明就在那里，你不明白为什么当时你就看不见。这就是当我看见别人对一个问题的灵活解决办法时，我所拥有的感觉。

问：你最喜欢的灵活解决办法是什么？

答：这个世界充满了灵活的解决办法，纸别针、尼龙搭扣、圆珠笔。但我猜你所问的是PHP语言中的办法。我认为其中的一个灵活解决办法是将HTML中的get、post，以及cookie数据直接联系到PHP中的变量上。这似乎是一件十分显然的事情，但在当时却没有人这么做，这使得PHP成为解决网络问题的非常便捷的方式。虽然这一特征可能被非正确地使用，而且从它开始起就受到了一些批评，但我仍然坚持并且赞赏这样一种灵活的方式。

211
-
213

5.4.2.3 类型推理

ML是一种既支持函数式程序设计又支持命令式程序设计的程序设计语言（Ullman，1998）。ML采用了一种有趣的类型推理机制，运用这种机制能够决定大部分表达式的类型，而不需要程序人员来指明其中的变量类型。

在通过ML函数研究类型推理之前，我们来看一下ML函数的一般语法：

```
fun 函数名 (形参) = 表达式;
```

表达式的值通过函数返回。[6]

现在我们讨论类型推理。考虑以下ML函数声明

```
fun circumf(r) = 3.14159 * r * r;
```

这指定了一个自变量浮点数（在ML语言中为real）并且产生浮点数结果的函数circumf。函数的类型从表达式中常量的类型推理出来。同样地，下面的函数

```
fun times10(x) = 10 * x;
```

其中的自变量以及函数的值被推理为整数类型。

考虑下列ML函数：

```
fun square(x) = x * x;
```

ML从函数定义中的"*"操作符确定出参数以及函数返回值的类型。因为"*"是一个算术操作符，因而可以设想参数以及函数值为数值类型。在ML语言中的默认类型为int。由此可以推理出参数以及square函数的返回值类型都为int。

如果使用一个浮点数来调用函数square，例如

```
square(2.75);
```

[6] 表达式可以是一个用分号分隔、用圆括号括起的表达式列表。这个例子的结果简单地是一个表达式。

就会产生错误，因为ML语言不能够将real类型的值转变为int类型。如果我们想让square函数接受real类型的参数，可以将square函数改写为

```
fun square(x) : real = x * x;
```

因为ML语言不允许重载的函数，因而这个函数不能够与前面square函数的int版本共存。

从函数值为real类型的事实足以推断出其中参数也为real类型。下面的这些定义也都具有合法性：

```
fun square(x : real) = x * x;
fun square(x) = (x : real) * x;
fun square(x) = x * (x : real);
```

214 纯函数式语言 Miranda 和 Haskell 中也运用了类型推理。

5.4.3 存储绑定与生存期

命令式程序设计语言的一种根本特征在很大程度上取决于这种语言中变量存储绑定的设计方式。因而清晰地理解变量的存储绑定就变得十分重要。

变量所绑定的存储单元必须取自一个或有存储单元的"池"。这个过程就称为存储空间分配。存储空间解除分配则是将已经与变量解除绑定的存储单元重新放回这个池中的过程。

变量的生存期是该变量被绑定于其一特定存储地址的时间。因此变量的生存期开始于将变量绑定到一个特定存储单元的时刻，而结束于该变量从这个存储单元上解除绑定之时。为了分析变量的存储绑定，可以较方便地根据标量（非结构性的）变量的生存期将它们分成五种类型，分别被称为静态变量、栈动态变量、显性堆动态变量以及隐性堆动态变量。下面的几节将讨论这四种类型的意义，以及它们的目的和优缺点。

5.4.3.1 静态变量

静态变量是在程序运行开始之前就被绑定到存储单元之上，并直到程序运行结束之前能够始终保持绑定在相同的存储单元之上的变量。静态地绑定于存储空间的变量对于程序设计具有几种应用价值。显然，全局可存取的变量常常用于程序执行的整个过程，因此必须在程序运行的整个期间将它们绑定于相同的存储空间。一些时候，让在子程序中声明的变量成为历史敏感的较为方便，也就是说，让变量在于程序的分别调用执行期间保持它们的值。这是静态绑定于存储空间的变量的一个特征。

静态变量的另一个优点是高效率。所有静态变量都可以直接寻址；[6]其值类型的变量则常常需要间接寻址，而间接寻址的存取速度比较慢。此外，静态变量没有在运行时进行分配与解除分配所需要的额外代价。

静态绑定于存储空间的一个缺点则是灵活性较差。尤其是一种语言只具有静态存储绑定的变量，这种语言不能够支持递归子程序。另一个缺点是在这些变量之间不能够共享存储空间。例如，假设一个程序有两个子程序，而这两个子程序都需要大型数组，并且假设从不同时调用
215 这两个子程序，如果这些数组为静态的，这两个子程序的数组就不能够共享存储空间。

C和C++允许程序人员在函数内变量的定义中包括static修饰符，使得所定义的变量成为静态的，请注意，当static修饰符出现于C++，Java以及C#的类定义中的变量声明时，与所声明变量的生存期无关，在这种情况下，它仅仅意味着所声明变量是一个类变量，而不是一个实例变量。有时，在第一次实例化类时创建类变量。同一个保留字的多种用途可能令人产生混淆，

[6] 在某些实现中，静态变量或许不需存储寻址，访问它们的开销与访问栈动态的变量差不多。

尤其是对语言的初学人员而言。

5.4.3.2 栈动态变量

栈动态变量是这样一些变量：当确立它们的声明语句时即产生了存储绑定，但它们的类型是静态绑定的。这种声明的确立指的是由这种声明所指示的存储空间的分配及绑定的过程，这个过程发生在程序执行到声明所依附的代码之时。因此这种确立发生于运行时。例如，在一个Java方法的头部声明的变量确立于调用这个方法之时。而在完成了这个方法执行的时候，由这些声明定义的变量被解除分配。

栈动态变量正如它们的名字所示，其存储空间被分配自运行时的栈。

有些语言（如C#和Java）允许在语句可以出现的任何地方进行变量声明。在这些语言的一些实现中，所有在函数或方法中声明的栈动态变量（不包括那些在嵌套块中声明的变量）可以在该函数或方法开始执行时绑定到存储空间，即使其中某些变量不是在一开始时就声明。在这种情况下，变量在声明处可见，但在函数或方法开始执行时进行存储空间的绑定（如果声明中指定了初始化，则还包括初始化）。可以在变量成为可见之前进行变量的存储空间绑定，这一事实对语言的语义没有影响。

至少在大部分情况下，能够被实际运用的递归子程序需要有某些形式的动态局部存储，这样，递归子程序的每一个活动副本都具有自己的局部变量版本，而通过使用栈动态变量，这些要求被十分方便地得到了满足。甚至在没有递归的情况下，子程序的栈动态局部存储也有着优越性，因为子程序中的所有局部变量都可以共享相同的存储空间。但与静态变量相比较，它所具有的缺点是，运行时的分配以及解除分配所需要的额外开销，由于间接寻址而导致的较慢访问，以及子程序不能够成为历史敏感的。栈动态变量的分配及解除分配并不需要耗费大量的时间，因为所有在子程序头部声明的栈动态变量都同时被旅行分配与解除分配，而不是分别操作的。 216

Fortran 95允许实现人员局部地使用栈动态变量，但需要包括这样一条语句：

```
Save list
```

的语句。如果在一个子程序中放置了Save语句，它允许程序人员将该子程序中的部分或者全部的变量（即list中的那些变量）指明为静态的。

在Java、C++以及C#中，在方法中定义的变量默认为是栈动态的；而在Ada中，所有定义于子程序中的非堆变量都是栈动态的。

除了存储之外的所有属性都是被静态地绑定于栈动态标量变量上。对于一些结构化类型则不是如此，我们将在第6章讨论这一点。对于栈动态变量的分配以及解除分配过程的实现，将在第10章中进行讨论。

5.4.3.3 显式堆动态变量

显式堆动态变量是由程序人员指定的显式运行时指令来进行分配与解除分配的无名（抽象）存储单元。这些从堆上被分配和解除分配的变量只能通过指针变量或者引用变量来引用。堆是一组由于使用的不规则性而在组织上高度松散的存储单元。一个显式堆动态变量具有两个与其关联的变量：其中的一个变量是指针或引用变量，只有通过这个变量才可以访问堆动态变量，另一个变量则是堆动态变量自身。建立这个指针或引用变量的方式与建立其他任何标量变量的方式一样，通过一个操作符（例如在Ada和C++中）或一个专为此目的而提供的系统子程序的调用（例如在C中）来产生一个显式堆动态变量。

在C++中，名为new的分配操作符使用一个类型名作为它的操作数。当执行这个分配操作符时，即产生一个操作数类型的显式堆动态变量，并且返回一个指向该堆动态变量的指针。由

于显式堆动态变量是在编译时与类型相绑定。而这种绑定是静态的。然而在产生这种变量时，它就与存储空间相绑定，这种绑定发生于运行时。

除了用于产生显式堆动态变量的子程序或操作符以外，一些语言还包括用于显式删除它们的子程序或操作符。

作为显式堆动态变量的一个例子，考虑下面的一段C++程序：

```
int *intnode;        // 定义一个指针
...
intnode = new int; // 建立堆动态变量
...
delete intnode;    // 将intnode指向的堆动态变量解除分配
```

217

在这个例子中，通过new操作符创建一个int类型的显式堆动态变量。然后可以通过指针intnode来引用这个变量。接下来，通过delete操作符将该变量解除分配。C++语言要求有显式解除分配操作符delete，原因是这种语言不使用隐式存储空间的回收，例如垃圾收集。

在Java语言中，除了基本标量之外的所有数据都是对象。Java中的对象是显式堆动态变量，并且是通过引用变量来访问。Java语言不具有任何显式的方式来删除显式堆动态变量；反之，它使用的是隐式垃圾收集。

C#语言既有堆动态对象也有栈动态对象，这两者都是隐式地解除分配。除此以外，C#还支持C++风格的指针。这种指针被用来引用堆、栈甚至是静态的变量以及对象。这些指针与C++中的指针一样具有危险性，而且向这些指针引用的堆对象不是隐式地解除分配。C#语言之所以包括这些指针，是希望C#语言的程序组件能够与C以及C++语言的程序组件一起工作。为了表示不鼓励使用指针，任何定义了指针的方法都必须包括保留字unsafe（不安全）。

显式堆动态变量常用于动态结构。如链表以及树结构，这些结构需要在运行期间生长或收缩。通过使用指针或者引用，以及使用显式堆动态变量，能够很方便地构造出这类结构。

显式堆动态变量的缺点是难于正确地使用指针变量与引用变量，还有这种变量的引用、变量的分配以及变量的解除分配所具有的代价，还有存储管理在实现上的复杂性。实质上，这就是经时而复杂的堆管理问题。关于显式堆动态变量的实现方法，将在第6章进行深入讨论。

5.4.3.4 隐式堆动态变量

隐式堆动态变量只有当它们被赋值时才被绑定到堆存储空间。实际上，当它们每次被赋值时，其所有的属性都被绑定。在某种意义上，它们仅仅是适应于任何被请求的用途的名字。例如，考虑下面的JavaScript赋值语句：

```
highs=[74, 84, 86, 90, 71];
```

不管变量highs是否是预先用在程序中，还是专具他用途，现在它都是具有5个数字值的数组。

这种变量的优点是它们具有高度的灵活性，允许编写极为通用的程序，而缺点则是在运行时

218 维护所有动态属性的额外开销。需要在各项属性中包括数组下标的类型和范围。另一个缺点是如我们在5.4.2.2节中讨论的，编译器会遗漏某些错误的检查。此外，隐式堆动态变量也存在与显示堆动态变量一样的存储管理问题。JavaScript中隐式堆动态变量的例子曾在5.4.2.2节中给出。

5.5 类型检测

为了便于讨论类型检测，我们将操作数和操作符的概念一般化，使得它们也包括子程序以及赋值语句。我们将认为子程序也是操作符，而子程序中的参数即为操作数。我们将认为赋值运算符是二元操作符，而它的目标变量以及它的表达式即为操作数。

类型检测是保证一个操作符的所有操作数都具有相应兼容类型的措施。兼容类型是对操作符而言为合法的类型，或者在语言规定的允许下，能够被编译器（或解释器）产生的代码隐式地转换成为合法类型。这种自动类型的转换被称为强制转换。例如在Java语言中，如果将一个int类型的变量与一个float类型的变量相加，该int变量的值就被强制转换成为float数值并加上一个小数点。

类型错误是将操作符作用于具有不适当类型的操作数。例如在C的初期版本中，如果将一个int数值传递给一个期望float数值的函数，就会产生一个类型错误（因为这种语言的编译器不检测参数的类型）。

如果在一种语言中，所有变量类型的绑定都是静态的，那么几乎总是能够静态地进行类型检测。动态类型的绑定要求在运行时进行类型检测，这种类型的检测被称为动态类型检测。

一些语言，如JavaScript和PHP，因为它们具有动态类型绑定，所以只允许动态类型检测，在编译时发现错误比在运行时发现错误优越得多，因为一般而言，改正错误越早代价就越低。静态检测的缺点是缺少灵活性，可以利用的捷径与技巧很少。现在，这种技术一般不被看好。

当一种语言允许存储单元在执行期间的不同时间存储不同的类型值时，类型检测就变得十分复杂。这种情形出现于使用Ada语言的变体记录，Fortran中的Equivalence，以及C和C++中的union。如果在这些情形中要完成类型检测，就必须执行动态类型检测，并且要求运行时系统保持存储单元里当前值的类型。所以，即使语言中的所有变量都与类型静态地绑定，如在C++语言中那样，但也并不是通过静态类型检测就能够发现所有的类型错误。

5.6 强类型化

强类型化是语言设计中的新思想之一，这种新思想在20世纪70年代里所倡结构化程序设计革命中成为主流。强类型化被广泛认为是一种具有很高价值的概念，但是它常常没有严谨的定义，有时甚至在完全没有定义的情况下用于计算机的文献之中。 219

只要某个程序设计语言总能够发现其程序中的类型错误，我们就定义这种程序设计语言为强类型的。这就要求能够在编译时或运行时确定所有操作数的类型。强类型化的重要性在于它能够发现所有因为变量的误用而导致的类型错误。一种强类型的语言也允许在运行时检测能够存储多个类型值的变量中不正确类型值的使用。

Fortran 95不是强类型的，因为在不同类型的变量之间使用Equivalence，就允许了一种类型的变量可以引用不同类型的值。当一个Equivalence的变量被引用或者被赋值时，没有系统能够检测其值的类型。事实上，如果对于Equivalence变量进行类型检测，将会失去Equivalence变量的大部分用途。

Ada几乎是强类型的，它仅仅几乎是强类型的，因为Ada允许程序人员通过特别要求来中止对于某种特殊类型转换进行类型检测，这违反了Ada中类型检测的规则。只有当调用适用函数指令Unchecked_Conversion时，才能够暂时地中止类型检测。可以使用这个函数对任何一对子类型施行实例化。[⊖]其中之一，它取其参数类型的一个值，并返回这个参数当前值的位串。在这里，实际转换并没有发生，它只是把取一种类型的变量值，并将这个变量值作为不同的类型来使用。这种方法对于用户定义的存储空间的分配和解除分配操作十分有用。在这些操作中，将地址作为整数来处理，但又必须将地址作为指针来使用。因为在Unchecked_Conversion

⊖ 通常，这两个子类型必须具有相同的长度。然而在一种实现中，可以为不同长度的子类型提供通用的Unchecked_Conversion指令，并且为支持进行不同的实现而修改规则。

中不进行检测，程序人员有责任确保这种转换所获得的值是有意义的。

C和C++都不是强类型的语言，因为这两种语言都包括了union类型，但是都不对这种类型实施类型检测。

ML是强类型的，尽管一些函数参数的类型在编译时可能是未知的。

虽然Java以及C#是基于C++的语言，但它们与Ada语言为同样意义上的强类型。语言中的类型能够被显式地转换，这可能会导致类型错误。但是，它们却能够发现任何隐式类型错误。

一种语言的强制转换规则在数值类型检测上具有重要影响。例如在 Java 中，表达式是强类型的。然而，一个算术操作符具有一个浮点操作数和一个整数操作数是合法的。这个整数操作数的值被强制转换成为浮点数，从而产生一个浮点运算。这正是程序人员通常想要的。然而，强制转换也会导致语言失去检测错误的能力，而检测错误正是我们需要强类型的原因之一。例如，假设一个程序具有int类型的两个变量a与b，以及一个float类型的变量d。现在，如果一个程序人员试图键入a + b，但却错误地键入了a + d，编译器就不能够检测出这个错误。a的
220 值就会被强制转换成为float类型。因此强类型化的价值会因为强制转换而有所减弱。一些语言，如Fortran、C以及C++，具有大量的强制转换，因而这些语言较之仅具有较少量强制转换的语言，如Ada，可靠程度要差得多。Java以及C#只具有C++的赋值类型强制转换数量的一半，所以Java与C#发现错误的能力比C++要强，但是仍然不能够具有Ada语言那么高的效率。关于强制转换的问题，还将在第7章中详细地研究。

5.7 类型等价

类型兼容的思想成型于类型检测被引入时。兼容规则指定每个操作符可接受的操作数的类型，并且随之指定了该语言可能的错误类型。[1]把这种规则称为兼容的原因是，为了使操作符接受操作数，在某些情况下，操作数的类型能被编译器或运行时系统隐式转换。

历史注释

Pascal语言的初始定义（Wirth，1971）没有清楚说明什么时候应该使用名字类型等价或结构类型等价。这对语言的可移植性被为不利，因为在一种实现中是正确的程序，在另外一种实现中可能不合法。ISO标准化的Pascal（ISO，1982）清楚地说明了语言的类型等价规则。这个规则既非完全取决于名字，也非完全取决于结构，在大多数情况下使用结构等价，而名字等价则被用于形参以及其他少数的一些情形。

类型兼容规则对于预定义标量类型是简单的和固定的。然而，在结构类型（如数组和记录）以及用户定义类型的情况下，这些规则更复杂。这些类型的强制性是很少的，因此这里的问题不是类型兼容，而是类型等价。也就是说，假如在一个表达式中，一种类型的操作数被另一种类型的操作数替代而无强制性，那么这两种类型是等价的。类型等价是类型兼容的一种严格形式——没有强制性的兼容。这里的主要问题是如何定义类型等价。

语言中的类型兼容规则设计是十分重要的，因为它影响着数据类型的设计，以及类型值操作的设计。这里讨论的类型，很少有预定义操作。也许两个变量具有兼容类型的最重要结果，是它们之中的任意一个可以将值赋给另一个。

有两种不同的类型等价方法：名字等价以及结构等价。**名字类型等价**意味着：两个变量具有等价类型，仅当它们
221 或者被定义于同一个声明之中，或者被定义于使用相同类型名的声明之中。**结构类型等价**意味着：两个变量具有等价的类型，如果它们的类型具有完全相同的结构。这两种方法还有一些变体，大部分语言使用的是这两种方法的结合形式。

[1] 在子程序调用中的实参和子程序定义的形参之间，也存在有类型兼容问题。第9章将讨论这个问题。

名字类型等价容易实现，但有着高度的局限性。从严格意义上，一个整数子范围类型的变量不会与一个整数类型的变量等价。例如，假设Ada使用了严格的名字类型等价，考虑下面的声明：

```
type Indextype is 1..100;
count : Integer;
index : Indextype;
```

变量count与变量index不等价，因而不能够将count的值赋给index，反向赋值也不可能。

当一个结构类型通过参数在一些子程序中传递时，就产生了另一种名字等价问题。这样的类型只能够被全局定义一次，子程序不能够局部地声明形参的类型。这正是Pascal最初版本中的情形。

注意，为了实用名字类型等价，所有类型必须有名字。大多数语言允许用户定义原本匿名的类型，为了让一种语言使用名字类型等价，编译器必须隐式指定这些类型的内部名字。

结构类型等价比名字类型等价更为灵活，但也更难以实现。在名字类型等价中，只需要比较两个类型名字以确定是否等价。然而在结构类型等价中，两种类型的整个结构都必须进行比较，而这种比较并非总是简单的（考虑一种数据结构引用自己的类型的情形，例如一个链表）。还会产生其他问题。例如，如果两个记录（或者两个struct）类型具有相同的结构，但域名不同，它们是否等价？如果两个Ada程序中的一维数组类型具有相同的元素类型，但下标范围分别为0，…，10和1，…，11，它们是否等价？如果两个枚举类型具有同样数目的组成部分，但字面常量的拼法不同，它们又是否等价？

结构类型等价的另一个难题是无法区分具有相同结构的类型，例如，考虑下面的类Pascal声明：

```
type celsius = float;
     fahrenheit = float;
```

这两种类型的变量类型在结构类型等价下被认为是等价的，它们被允许在表达式中相混合，然而这肯定不是我们期望的。一般而言，具有不同名字的类型可能是不同种类问题值的抽象，而不应该将它们视为等同。

Ada使用名字类型等价，但是提供了两种类型结构，即子类型与派生类型，这两种类型避免了上面所述的问题。派生类型是一种新类型，它基于某种以前定义的类型，但又与那个作为 222
基础的类型并不等价，虽然它也可能与之具有同一种结构。派生类型继承它们父类型中的所有性质。考虑下面的例子：

```
type celsius is new Float;
type fahrenheit is new Float;
```

这两种派生类型的变量类型是不等价的，尽管它们的结构相同。此外，这两种派生类型的变量都不与任何其他浮点类型等价。但这个规则不适宜于字面常量。一个字面常量（例如3.0）具有通用实数类型，并与任何浮点类型等价。派生类型在继承了父类型的所有操作之时，也继承了其父类型上的范围限制。

Ada中的子类型是一种已有类型的范围受限版本，子类型与其父类型相等价。例如，考虑下面的声明：

```
subtype Small_type is Integer range 0..99;
```

Small_type类型与Integer类型是等价的。

对于Ada语言中非限制性数组类型的变量，我们使用结构等价。例如，考虑下面的类型声明以及有关两个对象的声明：

```
type Vector is array (Integer range <>) of Integer;
Vector_1: Vector (1..10);
Vector_2: Vector (11..20);
```

这两个对象的类型是等价的，虽然它们分别具有不同的名字和不同的下标范围。因为对于非限制性数组类型的对象，我们使用结构类型等价而不是名字类型等价，又因为这两种类型都具有十个元素，并且这些元素的类型都为整数，因而这两个对象是等价的。

对于限制匿名类型来说，Ada使用了一种名字类型等价的、高度受限的形式。考虑下面Ada语言中有关限制匿名类型的一些声明：

```
A : array (1..10) of Integer;
```

在这种情形中，A具有匿名但唯一的、由编译器指定而程序不可知的类型。如果我们也有

223

```
B : array (1..10) of Integer;
```

A和B都会有匿名但不相同也不等价的类型，尽管它们结构上相同。下面的多重声明：

```
C, D : array (1..10) of Integer;
```

产生两个匿名的类型，一个是C的类型，另一个是D的类型，它们是不等价的。实际上可以将这个声明处理成下面的两个声明：

```
C : array (1..10) of Integer;
D : array (1..10) of Integer;
```

注意，名字类型等价的Ada形式比本节开始定义的名字类型等价更严格。假如我们将它们改写为：

```
type List_10 is array (1..10) of Integer;
C, D : List_10;
```

那么现在，C和D的类型是等价的。

除了匿名的数组之外，Ada中的所有类型必须都要指定，这也是其名字类型等价性性能良好的原因之一。

就语言而言，Ada中的类型等价规则比具有许多类型之间的强制转换的语言规则更为重要。例如在Java中，加法操作符的两个操作数实质上可以为语言中数值类型的任意组合。可以将其中的一个操作数强制转换成另一个操作数的类型；但是在Ada语言中，则不具有算术操作符的操作数的强制转换。

C使用了名字类型等价和结构类型等价。每一个struct及union的声明都产生一个与所有其他类型不等价的新类型。因此，名字类型等价用在结构、枚举和联合类型中。其他非标量类型使用结构类型等价，假如数组类型有相同类型的成分，则它们是等价的。当然，如果数组类型是恒定大小，那么它对于每个恒定大小的其他数组或没有恒定小的其他数组都是等价的。请注意，在C和C++语言中的typedef并不引入新的类型，它只是为已有的类型定义一个新的名称。因而，任何使用typedef定义的类型都等价于其父类型。C为结构、枚举和联合使用名字类型等价的一个例外是，如果在不同的源件中定义两个结构、枚举或联合，那么此时使用结构类型等价，在允许不同文件中定义的结构、枚举和联合等价的名字类型等价规则中，这是一个漏洞。

C++语言同C语言一样，希望使用名字等价性。

在不允许用户定义类型，也不允许用户命名类型的语言，如Fortran及COBOL语言中，显然不能够使用名字等价性。

面向对象的语言（如Java和C++）带来了另一种类型兼容问题，这个问题就是对象的兼容以及它与继承层次的关系。这个问题将在第12章中讨论。 224

表达式中的类型兼容将在第7章中讨论，子程序参数的类型兼容则将在第9章中讨论。

5.8 作用域

作用域是理解变量的最重要因素之一。程序变量的作用域是语句的一个范围，在这个范围之内变量为可见的。如果一个变量在一条语句中可以被引用，这个变量即在这条语句中为可见的。

一种语言的作用域规则决定着如何将一个名字的特定出现与一个变量相关联，尤其是，作用域规则决定在当前执行的子程序或程序块之外所声明的变量引用怎样与这种变量声明以及变量属性相关联（关于程序块将在第5.8.2节里讨论）。全面地掌握语言中的这些规则，对我们运用这种语言编写或阅读程序的能力十分关键。

如在第5.4.3.2节所定义的，如果一个变量声明于一个子程序单元或程序块之内，那么它就是这个子程序单元或程序块内的局部变量。程序单元或程序块的非局部变量是指不在这个程序单元或程序块中声明，但又对其可见的变量。

关于类、包以及名字空间的作用域问题将在第11章中讨论。

5.8.1 静态作用域

ALGOL 60引入了将名字绑定于非局部变量的方法，这种方法就称为**静态作用域**，它被后来的大多数命令式语言采用，同时也被许多非命令式语言采用。之所以称为静态作用域，是因为可以静态地决定变量的作用域，即在执行之前决定变量的作用域。这就允许人工程序阅读器决定程序中每个变量的类型。

有两类静态作用域语言：一类是，语言中的子程序可以被嵌套，并产生嵌套的静态作用域；另一类是，语言中的子程序不可以被嵌套。在后一类语言中，还是由子程序来产生静态作用域，但是嵌套的作用域只能够由嵌套的类定义以及嵌套的块产生（参见第5.8.2节）。

Ada、JavaScript和PHP允许嵌套的子程序，然而基于C的语言却不允许。

在这一章中，我们对于静态作用域的讨论将集中在那些允许嵌套的子程序的语言。因而从现在开始我们假设所有作用域都与程序单元相关联。在这一章中我们还假设在所讨论的语言中，作用域是访问非局部变量的唯一方法。这个假设并非对所有语言成立，甚至对于所有使用静态作用域的语言，这个假设都不成立。这种假设仅仅是为了简化这里的讨论。 225

假设一个程序是用一种静态作用域的语言来编写的，当程序的读者发现对一个变量的引用时，为了确定该变量的属性，就必须找到该变量的声明语句。在具有嵌套子程序的静态作用域的语言中，这个过程可以这样进行：假设对于变量X的引用发生于子程序Sub1中。首先通过搜索子程序Sub1中的声明来寻找关于X的正确声明。如果在子程序Sub1中没有发现这种声明，搜索将继续在声明子程序Sub1的子程序声明中进行，该子程序被称为Sub1的**静态父辈**。如果在那里还没有找到变量X的声明，搜索将继续到下一个更大的包含单位（声明子程序Sub1的父辈的程序单元），并依此类推，直到发现变量X的声明为止，或者是直到已经搜索了最大程序单元中的声明，但没有成功发现为止。在这种情况下，就发现了一个没有声明的变量的错误。子程序Sub1的静态父辈和静态父辈的静态父辈，由此往上直到主程序，这些程序单元被称为Sub1的**静态祖先**。请注意，我们将在第10章讨论的静态作用域的实现技术比刚才描述的过程具有更

高的效率。

考虑下面的Ada程序：

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin  -- Sub1开始
    ...
    end;  -- Sub1结束
  procedure Sub2 is
    begin  -- Sub2开始
    ...X...
    end;  -- Sub2结束
  begin  -- Big开始
  ...
  end;  -- Big结束
```

在静态作用域下，对Sub1中变量X的引用也就是对程序Big中声明的变量X的引用。因为对X的搜索开始于引用发生的程序Sub1，然而没有在那里发现X的声明。因此继续搜索Sub1的静态父程Big，终于在那里找到了X的声明。因为x没有在Sub2的静态祖先中，所以Sub1中声明的x能被忽略。

由于出现了曾经在5.2.3节讨论过的预定义名字，在某种程度上使得这个过程更加复杂。在有些时候，预定义名字像关键字一样可以由用户重新定义。因此，只有当用户的程序不包含重新定义时，才能够使用预定义名字，而在其他时候，预定义名字可以是被保留的，这意味着对给定名字的意义的搜寻开始于预定义名字表，甚至是在局部作用域的声明被检测之前。 226

在使用静态作用域的语言中，不论这种语言是否允许嵌套的子程序，一些变量的声明都可以对其他代码段隐藏起来。例如，考虑下面的C++框架方法：

```
void sub() {
  int count;
  ...
  while ( ... ) {
    int count;
    count++;
    ...
  }
  ...
}
```

在while循环中对count的引用是对这个循环中局部count的引用。在这种情况下，函数sub所声明的count对于while循环中的代码段是隐藏的。一般而言，一个变量声明有效地隐藏了在较大的封闭作用域中任何同名变量的声明。需要注意的是，这种代码在C和C++中是合法的，然而在Java及C#中是不合法的。Java和C#的设计人员相信嵌套块中名字的重用是潜在错误的，以致于不能够允许这样使用。

在Ada中，可以使用选择引用来访问祖先作用域中的隐藏变量，这种选择引用必须包括祖先作用域的名字。例如前面的Big程序，通过引用Big.X就可以在Sub1中访问在Big中声明的X。

虽然基于C的语言不允许将子程序嵌套于其他子程序的定义中，然而这些语言却具有全局变量。这些变量声明于任何子程序定义之外，局部变量可以隐藏这些全局变量，就如同在Ada语言中那样。在C++中，可以通过使用作用域操作符（::）来访问隐藏的全局变量。例如，如果

x是一个全局变量，它通过一个局部命名的x而隐藏于一个子程序中，但经由::x就可以引用这个全局变量。

5.8.2 块

许多语言允许在可执行代码中定义新的静态作用域。这个由ALGOL 60引入的十分有用的概念允许一段代码拥有自己的局部变量，而这些局部变量的作用域就是这段代码。这样的变量为典型的栈动态的，因而当执行这个代码段时，就进行变量的存储空间分配，而当这个代码段退出执行时，则进行存储空间的解除分配。我们将这段代码称为块。 227

在Ada语言中，使用declare子句进行块的说明，如下所示：

```
...
declare Temp : Integer;
  begin
  Temp := First;
  First := Second;
  Second := Temp;
  end;
...
```

请注意，如果一条Ada复合语句（由begin和end界定）没有包含声明，则不用包括declare子句。**块结构的语言**这一名称就源自于块。

基于C的语言允许任何复合语句（由配对的花括号界定的语句系列）具有声明，并由此定义新的作用域，这样的复合语句就是块。例如，如果list为一个整数数组，我们可以编写程序：

```
if (list[i] < list[j]) {
  int temp;
  temp = list[i];
  list[i] = list[j];
  list[j] = temp;
}
```

处理这些块产生的作用域，完全可以像处理由于程序产生的作用域一样。对声明不在块中的变量的引用，可以通过从小到大的顺序搜索它的包含作用域，从而找到变量声明。

C++允许变量定义出现于函数中的任何位置。当变量定义出现的位置不在函数的首部也不在一个块中时，该变量的作用域就是从这条定义语句的位置直到函数末尾。但注意在C中，所有在函数中的数据声明如果不是在块中，就必须是在函数的首部。

C++、Java以及C#中的for语句允许在它们的初始化表达式中放置变量定义。在C++的早期版本中，这种变量的作用域是从变量定义的位置到它最小包含块的末尾，然而在C++的标准版本中，仅将作用域限制于for结构之内，这同Java以及C#语言中的情形一样。考虑下面的框架方法： 228

```
void fun() {
  ...
  for (int count = 0; count < 10; ++count) {
    ...
  }
  ...
}
```

在早期版的C++中，count作用域是从for语句到方法结束的地方。在后续版中，像Java和

C#一样，count的作用域是从for语句到其结束的地方。

C++，Java以及C#中的类对待它们的实例变量的方式与对待定义于它们的方法之中的变量的方式不同。一个定义于方法中的变量的作用域开始于它的定义位置。然而，不论一个实例或类变量是在类中的什么位置定义的，它的作用域是整个类。

在面向对象的语言中，类和方法的定义会产生嵌套静态作用域。这些将在第12章中讨论。在Ruby中，变量名的形式表示了它们的作用域，因为Ruby是一种完全的面向对象语言，而且它的作用域与面向对象构建关联。这也将在第12章中讨论。

5.6.3 静态作用域的评估

静态作用域提供了一种非局部访问的方法。在许多情况下，这是一种相当好的方法，但也存在问题。考虑图5-1所示的框架程序。在这个例子中，假设所有作用域都由主程序及过程的定义产生。

这个程序包含主程序main的一个总作用域。它有两个过程A和B，它们在main中定义自己的作用域。在A的内部是过程C和D的作用域，而在B的内部是过程E的作用域。我们假设，必要的数据以及过程访问决定这个程序的结构。所要求的过程访问为：main能够调用A和B，A能够调用C和D，B能够调用A和E。

229

可以将这个程序的结构方便地表示为一棵树，其中的每个节点代表一个过程和一个作用域。图5-2显示了图5-1中程序的树结构。这个程序结构看上去可能像是一种非常自然的程序组织，它清楚地反映了设计的需要。然而，图5-3给出了这个系统可能的过程调用。该图显示，可能的调用会比要求的调用多。

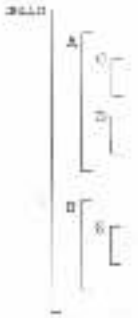


图5-1 一个程序的结构

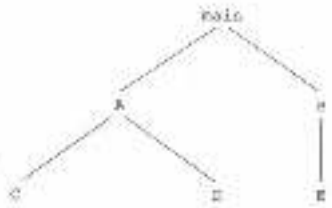


图5-2 图5-1中程序的树结构

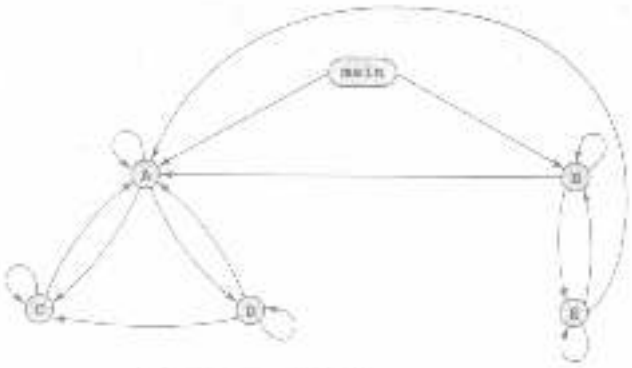


图5-3 图5-1中程序的可能调用图

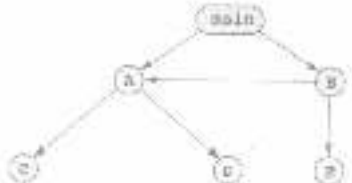


图5-4 图5-1中程序所需要的调用图

图5-4显示了上面例子中的程序所需要的调用。图5-3和图5-4之间的差别说明，在这个特定应用之中有许多可能的调用是不需要的。

程序人员有可能错误地调用一个不该被调用的子程序，而这个调用不会被编译器检查为错误。这使得错误的发现被推迟到运行时，从而提高了改正错误的代价。因此，应该将过程的访问限制在必需的范围之内。

太多的数据存取会造成问题。例如，所有在主程序中声明的变量不论是否需要，对所有过程都是可见的，并且无法避免。

为了说明静态作用域的另一种问题，考虑下面的情况。假设在开发与测试了上面的程序之后，需要对它的说明进行修改。特别是，假设现在过程E必须能够存取D的作用域中的一些变量。提供这种存取的一种方式就是将E移到D的作用域之内。但是这样做之后E就不可能够访问B的作用域。而这种访问多半是需要的（不然为什么原来将E放在那里？）。另一种解决的方式是将在D中定义而被E需求的变量移入main。这样就能够允许所有过程对这些变量进行存取，这会超过需要，并由此产生错误存取的可能。例如，可能将在一个过程中错误拼写的标识符当作是对一个包含作用域中标识符的引用，而不是将其检查为错误。此外，假设变量x被移到main中，而且D和E都需要x，但是假设在A里也有一个命名为x的变量。这就使得正确的x对于它原来的所有者D隐藏了起来。将x的声明移到main中的最后一个问题是，变量声明远离变量的使用将有损于可读性。

与静态作用域相关的变量可见性问题也出现于子程序的访问之中。在图5-2的树中，假设由于改变了一些说明，过程E需要调用过程D。这只能够通过移动D，将D直接嵌套于main中来完成。假设A或C也需要D，那么D也将会失去对在A中定义的变量的存取。如果是重复地使用这种解决方法，会导致在程序的开始有一长列低层次的功能过程。

因此要绕开静态作用域的限制，会导致程序设计的结果与初期设想产生很大差异，甚至对于程序中没有改变的部分也会如此。设计人员因此使用比所需要的多得多的全局变量。所有的过程都使用全局变量，而不是较深层次的嵌套，而且最终都嵌套在同一个层次，即主程序中。[⊖]此外，最后的设计可能是臃肿和不自然的，不能反映作为设计基础的概念设计。静态作用域所存在的上述这些问题以及其他缺点，在（Clarke，Wileden and Wolf，1980）中进行了详细讨论。解决静态作用域问题的一个办法是将要在第11章讨论的一种封装结构，现在这种结构已经被包含在许多新型的语言中。

230

231

5.8.4 动态作用域

在APL，SNOBOL4和LISP的早期版本中，变量的作用域是动态的。Perl和COMMON LISP也允许变量具有动态作用域，尽管这两种语言也使用静态作用域。动态作用域基于子程序的调用序列，而不是基于作用域相互之间的空间关系。因此这种作用域只能够在运行时得以确定。

让我们再次考虑第5.8.1节中的Big过程，我们将它再次显示如下：

```
procedure Big is
  X : Integer;
  procedure Sub1 is
    X : Integer;
    begin  -- Sub1开始
```

⊖ 看起来像C程序员的风格，不是吗？

```
  ...
  end;  -- Sub1结束
procedure Sub2 is
  begin  -- Sub2开始
  ...X...
  end;  -- Sub2结束
begin  -- Big开始
...
end;  -- Big结束
```

假设动态作用域的规则适用于非局部引用。在Sub1中引用标识符X的意义为动态的，即它不能够在编译时确定。子程序调用的顺序将决定对这两个X的声明中任何一个X的引用。

一种能够在运行时确定X正确意义的方法是从局部声明开始搜索。这也是静态作用域的开始方式，然而除此以外，这两种技术之间再也没有相似之处。当局部声明的搜索不成功时，搜索将进行到其动态父辈（即调用程序）的声明。如果在那里也没有发现X的声明，搜索又就继续到这个程序的动态父辈，依此类推，直到找到X的声明为止。如果在任何一个动态祖先中都没有发现X的声明，它就是一个运行时错误。

考虑上面例子中对Sub1的两种不同的调用序列。第一种情形，Big调用Sub2，Sub2再调用Sub1。在这种情况下，搜索过程从局部程序Sub1到它的调用者Sub2，而在Sub2中找到了X的声明。所以，在这种情况下在Sub1中引用的X是在Sub2中声明的X。第二种情形，如果Big
232 直接调用Sub1，在这种情况下，Sub1的动态父辈是Big，因而所引用的是在Big中声明的X。

请注意，如果我们使用的是静态作用域，则不论是上面讨论的那一种调用序列，对于Sub1中X的引用都是引用Big中的X。

Perl语言中的动态作用域是非同寻常的——虽然它在语义上是传统的动态作用域(参见程序设计练习题1)，但事实上却与在本节讨论的不完全一样。

5.8.5 动态作用域的评估

动态作用域对于程序设计的影响极为深刻。程序语句可见的非局部变量的正确属性不能够静态地确定。此外，这样的变量并非总是一样的。包含非局部变量引用的子程序中的语句可以在子程序的不同执行期间引用不同的非局部变量。而几种程序设计的问题都是直接由动态作用域产生的。

首先，从子程序的执行开始到执行结束的期间，该子程序中的局部变量对于任何其他正在执行的子程序都是可见的，不论这些子程序的接近程度如何，还没有任何办法可以保护局部变量避免这种可存取性。子程序总是在先前被调用的程序的环境中执行，只要这些程序的执行还没有结束；所以，动态作用域导致比静态作用域更低的程序可靠性。

动态作用域的第二个问题是不能进行非局部变量引用的静态类型检测。这个问题起源于不能静态地决定非局部变量引用的声明。

动态作用域也使得程序的可读性降低，因为必须知道了程序的调用顺序，才能确定非局部变量引用的意义。这对于人类读者而言实际上是不可能的。

最后，在动态作用域的语言中，对非局部变量的存取远比使用静态作用域时对非局部变量的存取所耗时间长得多，我们将在第10章就其原因给出解释。

但动态作用域并不是没有优点。在某些情形下，从一个子程序传递到另一个子程序的参数，仅仅是在调用程序中定义的变量。然而在动态作用域语言中，并不需要传递这些参数，因为它们对被调用的子程序是总式可见的。

不难理解为什么动态作用域不如静态作用域使用广泛。使用静态作用域语言编写的程序比使用动态作用域语言编写的等同程序更易读，更可靠，而且运行速度更快。正是出于这样的原因，在LISP最新方言中，用静态作用域取代了动态作用域。关于静态作用域及动态作用域的实现方法，将在第10章讨论。 [233]

5.9 作用域与生存期

有时，变量的作用域与生存期似乎是相关的。例如，考虑一个在不包含方法调用的Java方法中声明的变量。这个变量的作用域是从它的声明到这个方法的末尾。它的生存期则开始于方法被进入之时，终止于方法执行完毕之时。虽然变量的作用域与生存期显然不是同一件事情：静态作用域是文本的或空间的概念，而生存期是时间的概念，但至少它们在这个情形下似乎是相关的。

作用域与生存期之间这种表面上的关系在其他情况下并不存在。例如在C和C++中，一个在函数中由修饰符static声明的变量被静态地绑定于函数的作用域，同时也被静态地绑定于存储空间。所以这个变量的作用域是静态的、并且对这个函数为局部的，但它的生存期却扩展到了整个程序的执行期间。

当涉及到子程序调用时，作用域和生存期已是无关的。考虑下面这些C++函数：

```
void printheader() {
  ...
 }  /* printheader结束 */
void compute() {
  int sum;
  ...
  printheader();
 }  /* compute结束 */
```

变量sum的作用域完全包含在函数compute之中。尽管printheader在函数compute的运行中可执行，但sum的作用域却没有扩展到printheader函数体中。然而，sum的生存期则被扩展到了printheader执行的整个期间。在printheader被调用之前，无论将变量sum绑定于什么存储空间，这种绑定在printheader的执行中间以及执行之后都将继续。

5.10 引用环境

一条语句的引用环境是这条语句中所有可见变量的集合。在静态作用域的语言中，一条语句的引用环境是在它的局部作用域所声明的变量和在它的祖先作用域所声明的所有可见变量的集合。在这种语言中编译一条语句时，需要有关这条语句的引用环境。这样就能够产生代码和数据结构，以便在运行时对来自其他作用域的变量进行引用。我们将在第10章讨论如何在静态作用域语言以及动态作用域语言中实现非局部变量引用的技术。 [234]

在Ada语言中，可以通过过程定义来产生作用域。一条语句的引用环境包括局部变量，再加上该要这条语句的程序中所声明的所有变量（不包括被较近的过程声明所覆盖的非局部作用域中的变量）。每一条过程定义产生一个新的作用域以及由此而来的新环境。现在考虑下面的Ada框架程序：

```
procedure Example is
  A, B : Integer;
  ...
```

```
procedure Sub1 is
  X, Y : Integer;
  begin  -- Sub1开始
  ...  <----------------1
  end;  -- Sub1结束
procedure Sub2 is
  X : Integer;
  ...
  procedure Sub3 is
    X : Integer;
    begin  -- Sub3开始
    ...  <--------------2
    end;  -- Sub3结束
  begin  -- Sub2开始
  ...  <----------------3
  end;  -- Sub2结束
begin  -- Example开始
...  <------------------4
end.  -- Example结束
```

标记了的程序中不同点的引用环境如下：

点	引用环境
1	Sub1中的X和Y，Example中的A和B
2	Sub3中的X（Sub2中的X是隐藏的），Example中的A和B
3	Sub2中的X，Example中的A和B

现在来考虑这个框架程序里的变量声明。首先注意，虽然Sub1的作用域层次高于Sub3的作用域层次（它的嵌套比较浅），但是Sub1的作用域却不是Sub3的作用域的静态祖先，因而Sub3不能存取在Sub1中声明的变量。这种现象的原因是，在Sub1中声明的变量是被动态的，因此在Sub1没有被执行之前，这些变量还没有与存储空间绑定。又因为Sub1没有被执行之前Sub3可能已经被执行，所以Sub3就不能存取Sub1中的变量，这些变量不一定在Sub3的执行
235 期间就与存储空间绑定了。

如果已经开始一个子程序的执行，并且该执行还没有终止，就称这个子程序是活跃的。在动态作用域的语言中，一条语句的引用环境是局部声明的变量，加上当前活跃的其他子程序中的所有变量。一些活跃子程序中的变量可能对引用环境为隐藏的，最近的子程序活动可以具有变量声明，它隐藏了前一个子程序活动中具有的同名变量。

考虑下面的程序例子。假设其中仅有的函数调用为：main调用sub2，sub2再调用sub1。

```
void sub1() {
  int a, b;
  ...  <----------------1
 }  /* sub1 结束 */
void sub2() {
  int b, c;
  ...  <----------------2
  sub1;
 }  /* sub2 结束 */
void main() {
  int c, d;
  ...  <----------------3
  sub2();
 }  /* main 结束 */
```

已标记的程序中不同点的引用环境如下：

点	引用环境
1	sub1中的a和b，sub2中的c，main中的d（main中的c以及sub2中的b是隐藏的）
2	sub2中的b和c，main中的d（main中的c是隐藏的）
3	main中的c和d

5.11 命名常量

命名常量是这样一种变量，它只与值绑定一次。命名常量的用处在于它有助于增进程序的可读性与可靠性。例如，通过使用名字pi而不是使用常量3.14159，可读性可以得到改进。

使用命名常量的另一个优点是将程序参数化，例如一个程序处理固定数目的数据值100。这样的程序通常在许多位置使用常量100，来声明数组下标范围或用于循环控制限制。考虑下面的Java框架程序段： 236

```
void example() {
  int[] intList = new int[100];
  String[] strList = new String[100];
  ...
  for (index = 0; index < 100; index++) {
    ...
  }
  ...
  for (index = 0; index < 100; index++) {
    ...
  }
  ...
  average = sum / 100;
  ...
}
```

当必须修改这个程序以处理不同数目的数据时，就必须找到并修改所有出现数字100的位置。在处理一个大程序时，这可能会很麻烦而且很容易出错。一种较容易并且可靠的方法是使用一个命名常量，如下面的程序的用法：

```
void example() {
  final int len = 100;
  int[] intList = new int[len];
  String[] strList = new String[len];
  ...
  for (index = 0; index < len; index++) {
    ...
  }
  ...
  for (index = 0; index < len; index++) {
    ...
  }
  ...
  average = sum / len;
  ...
}
```

现在，不论这个数字在程序中被使用多少次，当必须进行改变时，只需要修改一行代码（变量len）。这是抽象所具有的优点的另一个例子。名字len是数组元素的数目以及循环重复

237 数目的抽象。这个例子说明命名常量如何有助于程序的修改。

Fortran 95只允许用常量表达式作为其命名常量的值。这些常量表达式可以包含先前定义的命名常量、常量值以及操作符。在Fortran 95中对于常量和常量表达式实施这种限制的原因，是因为它将值与命名常量静态地绑定。在使用值的静态绑定的语言中，有时也称命名常量为说明常量（manifest constant）。

Ada和C++允许将值动态地绑定于命名常量。这样就能够允许含有变量的表达式对声明中的命名常量赋值。例如这样一条C++语句：

```
const int result = 2 * width + 1;
```

将result声明为整数类型的命名常量，它的值被设为表达式2 * width+1的值。当result被分配并与值绑定时，变量width的值必须为可见的。

Java也允许将值动态地绑定于命名常量。在Java中，使用保留字final来定义命名常量（如前面的例子所示）。可以在声明语句中或者在后面的赋值语句中，将初始值赋给命名常量，可以使用任何表达式来指明被赋的值。

C#有两种命名常量：用const定义的命名常量，以及用readonly定义的命名常量。const-命名常量是式，静态地与值绑定。也就是说，它们在编译时与值绑定。这意味着这些值只能够通过字面量或其他const成员来指定。而readonly-命名常量则是动态地与值绑定，它们可以在声明中或者在静态构造函数中赋值。[9]因此，如果程序需要一个常量值对象，该对象值在程序的每一次使用中都是相同的，那么将会用到const常数。然而，如果程序需要一个常量值对象，该对象值只在创建对象时确定，并且随着程序的不同执行情况而改变，那么此时将用到readonly常量。

Ada允许枚举类型与结构类型的命名常量，这些将在第6章进行讨论。

关于将值与命名常量绑定的讨论，自然地导致了变量初始化的主题，因为将值绑定于一个命名常量的过程与变量的初始化过程相同，只是前者为永久性的。

在许多情况下，如果变量在开始执行具有变量声明的程序或子程序之前，就已经具有了值，那将是较为方便的。在变量与存储空间绑定时，这个变量与值的绑定就称为变量的初始化。如果变量被静态地与存储空间绑定，而且绑定与初始化都发生在运行时之前，在这些情况下，初始值只能够被声明为字面常量，或者是仅包含命名常量的非字面常量操作数的表达式。如果对
238 存储空间的绑定是动态的，则初始化过程也会是动态的，并且初始值可以为任意表达式。

在大多数语言中，初始化是在产生变量的声明中指定的。例如在C++中，我们可能有：

```
int sum = 0;
int* ptrSum = &sum;
char name[] = "George Washington Carver";
```

小结

大、小写敏感性以及名字与特殊字（保留字或关键字）的关系，是有关名字的设计问题。

变量可以由六种属性来刻画：名字、地址、值、类型、生存期、作用域。

别名为绑定于同一个存储单元的两个或多个名字。别名被认为有损语言的可靠性，然而又很难从一种语言中完全消除。

绑定是属性与程序实体的关联。关于属性与实体的绑定时间的知识，对于理解程序设计语言的语义

[9] C#静态构造函数运行在实例化发生之前的一些非关键时刻。

十分灵活。绑定可以是静态的或者动态的。显式声明或者隐式声明变量与类型的静态绑定的方式，一般而言，动态绑定可以允许较大的灵活性，但却是以可读性、效率以及可靠性作为代价的。

通过考虑标量变量的生存期，可以将它们划分为四类：静态类、栈动态类、显式堆动态类、隐式堆动态类。

强类型化的概念是要求能够发现所有的类型错误。强类型化的优点是增加可靠性。

语言的类型等价规则决定在语言的结构化类型中什么操作是合法的。名字类型等价和结构类型等价是定义类型等价的两种基本方法。

静态作用域是ALGOL 60及其大部分后继语言的一个主要特性。它为允许了程序中非局部变量的可见性提供了一种有效方法。动态作用域比静态作用域提供了更多灵活性，但却是以可读性、可靠性以及效率作为代价。

一条语句的引用环境是所有可以在该语句为可见的变量的集合。

命名常量就是与值仅绑定一次的变量。

复习题

1. 名字的设计问题是什么？
2. 大、小写敏感的名字具有什么潜在危险？ 159
3. 保留字比关键字好在哪些方面？
4. 什么是别名？
5. 哪一类C++的引用变量总是别名？
6. 什么是变量的左值？什么是变量的右值？
7. 定义绑定与绑定时间。
8. 在语言的设计以及实现之后，在一个程序中哪几个时间可以发生绑定？
9. 定义静态绑定与动态绑定。
10. 隐式声明有哪些优点与缺点？
11. 动态类型绑定有哪些优点与缺点？
12. 定义静态、栈动态、显式堆动态和隐式堆动态的变量，它们的优点与缺点分别是什么？
13. 定义强制转换、类型错误、类型兼容和显类型化。
14. 定义名字类型等价和结构类型等价，什么是它们之间相对的优点？
15. Ada派生类型与Ada子类型之间有什么不同？
16. 定义生存期、作用域、静态作用域和动态作用域。
17. 一个静态作用域的程序中关于非局部变量的引用是如何与它的定义相关联的？
18. 什么是静态作用域中的营救问题？
19. 什么是语句的引用环境？
20. 什么是子程序的静态祖先？什么是子程序的动态祖先？
21. 什么是程序块？
22. 动态作用域的优点与缺点是什么？
23. 命名常量的优点是什么？

练习题

1. 从下列标识符形式中，确定每一种具有最好的可读性，并给出理由。

```
SumOfSales
sum_of_sales
SUMOFSALES
```

2. 一些程序设计语言是无类型的。一种没有类型的语言具有哪些明显的优点和缺点？

3. Fortran中Equivalence语句的一种通常用法如下：可以将一种包含大量数值的数组用作子程序的参数。这种数组包含了许多相互无关的不同变量，而不只是相同变量的副本的集合。之所以将它表示为数组，是为了能够减少在参数传递中所需要名字的数目。在子程序中使用较长的Equivalence语句对各种不同的数组元素产生有意义的名字作为别名，以增加子程序代码的可读性。你认为这是否是一个好主意？有什么不同的方法可以替代别名？

240

4. 用你所知道的一种语言来编写一条包含一个算术操作符的简单赋值语句。对于这条语句中的每一种成分，列出执行语句时用来确定语义所需要的各种不同的绑定。并且对其中的每一种绑定，指出语言中的绑定时间。

5. 动态类型绑定与隐式非动态变量紧密关联。请解释这种关系。

6. 描述在子程序中历史敏感变量具有实用意义的一种情形。

7. 查阅Gehani在文献（Gehani，1983）中所给出的强类型的定义，并将它与本章给出的定义相比较，它们有哪些不同？

8. 考虑下面的Ada框架程序：

```
procedure Main is
  X : Integer;
  procedure Sub3;  -- 这是Sub3的声明，它定得Sub1调用Sub3
  procedure Sub1 is
    X : Integer;
    procedure Sub2 is
      begin  -- Sub2开始
      ...
      end;  -- Sub2结束
   begin  -- Sub1开始
   ...
   end;  -- Sub1结束
 procedure Sub3 is
   begin  -- Sub3开始
   ...
   end;  -- Sub3结束
 begin  -- Main开始
 ...
 end;  -- Main结束
```

假设这个程序的执行遵循下面的顺序：

```
Main 调用 Sub1
Sub1 调用 Sub2
Sub2 调用 Sub3
```

241

a. 假设是静态作用域，其中哪一个X的声明对于下列X的引用是正确的：

i. Sub1

ii. Sub2

iii. Sub3

b. 假设是动态作用域，重复a中的问题。

9. 假设下面的Ada程序已经使用静态作用域规则进行过编译和运行。在过程Sub1中将输出什么样的X值？而在动态作用域规则之下，在过程Sub1中又将输出什么样的X值？

```
procedure Main is
  X : Integer;
  procedure Sub1 is
```

```
    begin  -- Sub1开始
    Put(X);
    end;  -- Sub1结束
  procedure Sub2 is
    X : Integer;
    begin  -- Sub2开始
    X := 10;
    Sub1
    end;  -- Sub2结束
  begin  -- Main开始
  X := 5;
  Sub2
  end;  -- Main结束
```

10. 考虑下面的程序：

```
procedure Main is
  X, Y, Z : Integer;
  procedure Sub1 is
    A, Y, Z : Integer;
    procedure Sub2 is
     A, B, X : Integer;
     begin  -- Sub2开始
     ...
     end;  -- Sub2结束
    begin  -- Sub1开始
    ...
    end;  -- Sub1结束
  procedure Sub3 is
    A, X, W : Integer;
    begin  -- Sub3开始
    ...
    end;  -- Sub3结束
  begin  -- Main开始
  ...
  end;  -- Main结束
```

242

列出所有在Sub1、Sub2和Sub3的程序体中可见的变量，并同时说明这些变量的程序单元，假设使用的是静态作用域。

11. 考虑下面的程序：

```
procedure Main is
  X, Y, Z : Integer;
  procedure Sub1 is
    A, Y, Z : Integer;
    begin  -- Sub1开始
    ...
    end;  -- Sub1结束
  procedure Sub2 is
    A, X, W : Integer;
    procedure Sub3 is
      A, B, Z : Integer;
      begin  -- Sub3开始
      ...
      end;  -- Sub3结束
    begin  -- Sub2开始
    ...
    end;  -- Sub2结束
```

```
begin  -- main开始
...
end;  -- main结束
```

列出所有在sub1、sub2和sub3的程序体中可见的变量，连同声明这些变量的程序单元，假设使用的是静态作用域。

12. 考虑下面的C程序：

```
void fun(void) {
  int a, b, c; /* 定义1 */
  ...
  while (...) {
    int b, c, d; /* 定义 2 */
    ... <--------------1
    while (...) {
     int c, d, e; /* 定义3 */
     ... <--------------2
     }
    ... <--------------3
   }
  ... <---------------4
 }
```

243

对函数中的每一个标记点，请列出每个可见变量，同时也列出定义这个变量的定义语句的数目。

13. 考虑下面的骨架C程序：

```
void fun1(void);  /* 样式 */
void fun2(void);  /* 样式 */
void fun3(void);  /* 样式 */
void main() {
  int a, b, c;
  ...
 }
void fun1(void) {
  int b, c, d;
  ...
 }
void fun2(void) {
  int c, d, e;
  ...
 }
void fun3(void) {
  int d, e, f;
  ...
 }
```

给定下面的调用序列，并假设使用的是动态作用域，在最后一个被调用的函数的执行期间，哪些变量是可见的？对于每一个可见变量，请给出定义这些变量的函数的名称。

a. main调用fun1，fun1调用fun2，fun2调用fun3。

b. main调用fun1，fun1调用fun3。

c. main调用fun2，fun2调用fun3，fun3调用fun1。

d. main调用fun3，fun3调用fun1。

e. main调用fun1，fun1调用fun3，fun3调用fun2。

f. main调用fun3，fun3调用fun2，fun2调用fun1。

14. 考虑下面的程序：

```
procedure Main is
  X, Y, Z : Integer;
  procedure Sub1 is
    A, Y, Z : Integer;
    begin  -- Sub1开始
    ...
    end;  -- Sub1结束
  procedure Sub2 is
    A, B, Z : Integer;
    begin  -- Sub2开始
    ...
    end;  -- Sub2结束
  procedure Sub3 is
    A, X, W : Integer;
    begin  -- Sub3开始
    ...
    end;  -- Sub3结束
  begin  -- Main开始
  ...
  end;  -- Main结束
```

244

给定下面的调用序列，并假设使用的是动态作用域，在执行最后一个活跃子程序期间，哪些变量是可见的？对于每个可见变量，请给出声明这些变量的程序单元的名称。

a. main调用sub1，sub1调用sub2，sub2调用sub3。

b. main调用sub1，sub1调用sub3。

c. main调用sub2，sub2调用sub3，sub3调用sub1。

d. main调用sub3，sub3调用sub1。

e. main调用sub1，sub1调用sub3，sub3调用sub2。

f. main调用sub3，sub3调用sub2，sub2调用sub1。

程序设计练习题

1. Perl语言允许静态作用域以及一种动态作用域。编写一个使用这两种作用域的Perl程序，并清晰地显示这两种作用域在效果上的不同。清楚地解释本章所描述的动态作用域与Perl语言中所实现的动态作用域之间的不同。
2. 编写一个COMMON LISP程序，它能够清楚地显示静态作用域与动态作用域之间的区别。
3. 编写一个具有三层嵌套子程序的JavaScript脚本，其中每一个子程序可以引用所有在包含这个子程序本身的子程序中定义的变量。
4. 编写一个C函数，它包含了下列语句系列：

```
x = 21;
int x;
x = 42;
```

245

运行这个程序并解释其结果。再用C++和Java重新编写同样的程序，并比较它们的结果。

5. 用C++、Java以及C#编写测试程序，来确定在一条for语句中声明的变量的作用域。尤其是所编写的代码必须能够确定这样的变量在for语句的循环体之后是否可见。
6. 用C++或者C编写二个函数，第一个静态地声明一个大数组，第二个在栈上声明同样的一个数组，第二个则从堆中动态同样的数组。每个函数至少被调用100 000次，记录下它们各自所需要的运行时间。解释你的结果。
7. 选择一种语言编写程序，使该语言在使用名字等价和结构等价时显现不同的行为。
8. 在C++中而不是在Java中，哪种类型的A和B是合法的简单赋值语句A=B？
9. 在Java中而不是在Ada中，哪种类型的A和B是合法的简单赋值语句A=B？

246

第6章 数据类型

这一章首先将介绍数据类型的概念，以及常用基本数据类型的特征，然后讨论枚举和子范围类型的设计，接着将学习结构化数据类型，特别是数组、记录和联合。最后对指针和引用进行深入的学习。

我们将阐述每一种数据类型的设计问题，并且将解释在一些重要语言的设计中设计人员所做出的选择，并对这些设计做出评估。

数据类型的实现方法有时极大地影响着它们的设计，因此各种数据类型的实现问题，尤其是数组的实现问题，是本章的另一个重要部分。

6.1 概述

数据类型定义一组数据值，以及在这些数值上预定义的一组操作。计算机程序通过操纵数据来产生结果。决定计算机程序执行任务难易程度的一个重要因素，是可提供的正在使用的数据类型与真实世界问题空间的匹配程度。因此，一种语言能够支持适当多样化的数据类型与结构就成为关键。

数据类型化的现代概念是从过去50年间发展起来的。在最早期的语言中，必须用语言来支持少量基本数据结构，以模拟所有问题空间的数据结构，例如在Fortran 90之前，通常使用数组来模拟链表及二叉树结构。

COBOL的数据结构通过允许程序人员指明小数的精度，并适当给信息记录提供一种结构化的数据类型，跨出了脱离Fortran I模式的第一步。PL/I语言更是将精度说明的功能扩展到整数与浮点类型。这种功能从此被引进了Ada和Fortran语言中。PL/I的设计人员包括了多种数据类型，目的在于让语言能够支持较大范围的应用。ALGOL 68则引入了一种更好的方式，就是仅提供少数的基本类型以及少量灵活的结构定义操作符，而允许程序人员为每一种需求设计一种数据结构。显然，这是数据类型设计的发展过程中最重要的进步之一。用户定义的类型通过使用具有意义的类型名字增进了可读性，它们允许对特殊用途的变量进行类型检测，这在过去是不可能的。用户定义的类型也有助于可修改性：程序人员只需要改变类型声明语句，就能够改变程序中一类变量的类型。

从用户定义类型这个概念往前一步，我们就到达了抽象数据类型的阶段。这种抽象数据类型能够在Ada 83中进行模拟，并且它是大部分后续语言的组成部分。抽象数据类型的基本思想是，类型的接口（它对用户可见）与类型的表示和在类型的值上的操作集相分离（它对用户不可见）。高级程序设计语言提供的所有类型都是抽象数据类型。关于用户定义的抽象数据类型，将在第11章中进行详细讨论。

247
↓
248

尽管关联数组变得流行起来，但两种最常用的结构化（非标量的）数据类型是数组和记录。这两种数据类型以及少数的一些其他数据类型由类型操作符（或构造器）来指明。类型操作符被用来形成类型表达式。例如，C语言中的类型操作符是方括号、圆括号及星号，它们分别用来指明数组、函数以及指针。

使用描述符来描述变量，无论是从逻辑角度还是从实际角度都是很方便的。描述符是一个

变量的一组属性集合。在实现中，描述符是存储变量属性的存储单位集合。如果所有的这些属性都是静态的，那么只有在编译时才需要描述符，它们通常是由编译器构造的，是符号表的一部分，并且用于编译期间。然而对于动态属性，部分或者全部的描述符则必须在执行期间维护。在这种情况下，描述符被运行时系统使用。在所有的情况下，描述符都被用于类型检测以及建造分配与解除分配操作的代码。

对象（object）一词常常与一个变量的值以及这个变量所占的空间相关联。然而在本书中，我们特别保留对象这一词，仅将它用于用户定义的抽象数据类型的实例，而不再将它用于预定义类型的变量值。在面向对象的语言中，每一个类中的每一个实例，不论它是预定义的还用户定义的，都被称为对象。关于对象，我们将在第11章和第12章中详细讨论。

在下面的几节里，我们将讨论所有常用的数据类型。还将对其中的大部分数据类型说明类型设计中的特殊问题。我们将对所有的数据类型都给出一个或多个设计范例的描述。有一个设计问题是所有数据类型中最基本的问题：即，对于这种类型的变量提供了什么样的操作，以及怎样说明这些操作？

6.2 基本数据类型

不用其他类型来定义的数据类型被称为**基本数据类型**。几乎所有程序设计语言都提供一组基本数据类型。某些基本类型仅仅是硬件的反映，例如整数类型，而另一些基本类型只是在实现上需要一点儿硬件的支持。

语言的基本数据类型之间一个或多个类型构造器被用来提供结构化类型。

6.2.1 数值类型

许多早期的程序设计语言仅仅具有数值基本类型。数值类型在现代语言所支持的类型中仍然扮演着重要角色。

6.2.1.1 整数

249

最常用的基本数值数据类型是整数。现在许多计算机都支持几种不同大小的整数。整数的大小及其他功能在一些程序设计语言中得到支持。例如，Java支持四种有符号的整数类型：byte, short, int以及long。一些语言，例如C++和C#，包括了无符号的整数类型，它们只不过是不具有正负号的整数值类型，通常用于二进制数据中。

在计算机中，有符号的整数值由位串来表示，最左边的位通常用来表示正负符号。整数类型由硬件直接支持。一个硬件不直接支持的整型例子是Python的长整型，该类型的值的长度没有限制。长整型值能指定为字面量，正如

```
243725839182756201923L
```

对于产生值太大以致于不能用int类型表示的整数算术操作可以使用长整型值来存储。

可以用符号-数值（sign-magnitude）记法来存储一个负整数，其中的符号位用来指示负数，剩余的位用来表示这个数的绝对值。然而，符号数值记法本身并不介入计算机的算术运算。现在大多数的计算机使用一种被称为二进制补码的记法来存储负整数。它对于加法和减法都极为方便。在二进制补码记法中，负整数的表示是通过取正整数的逻辑反再加"1"构成的。但有些 250
计算机仍然使用二进制反码记法。在二进制反码记法中，负整数被存储为其绝对值的逻辑补码。二进制反码记法的缺点是，它具有零的两种表示方式。关于整数表示的细节，可以查阅任何汇编语言程序设计的有关书籍。

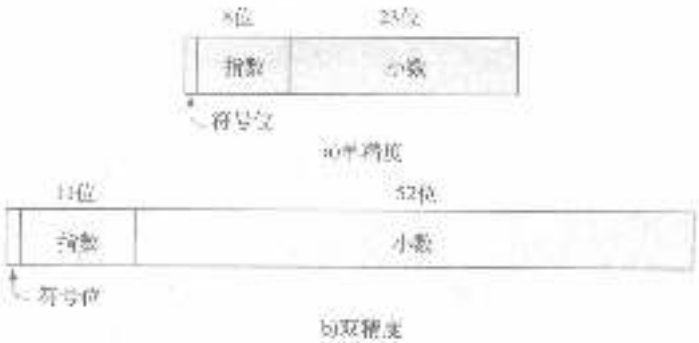
6.2.1.2 浮点数

浮点数据类型模拟实数，但对于大多数实数而言，浮点数的表达只是一种近似。例如，基本数π或e（自然对数的底）都不能用浮点数记法准确地表示。当然，这两个数都不能在任何有限的空间内被精确地表示出来。在大多数计算机上，浮点数被存储为二进制数，这使得问题更加糟糕。例如，甚至连十进制数的0.1都不能够用有限的二进制数来表示。[⊖]浮点类型的另一个问题是在算术运算之后会丢失准确性。有关浮点数记法问题的更多信息，请参考任何有关数值分析的书籍。

浮点值被表示为小数和指数，这是借鉴科学标记法的一种形式。早期的计算机使用了多种不同的浮点值表示方式。然而，最新的机器则使用IEEE浮点标准754的格式。语言的实现人员采用硬件支持的表示形式。大多数的语言通常包括了被称为Float（浮点数）和Double（双精度浮点数）的两种浮点类型。Float类型具有标准的大小，通常被存储于四个字节之中。Double类型专门用于需要较长小数部分的情形。双精度变量通常占用两倍于浮点数变量所占有的存储空间，并且提供至少两倍于浮点数变量的小数位。

精度与范围定义了能够被浮点类型表示的值的集合。精度是一个值的小数部分的准确性，它以位的数目来衡量。范围则是小数范围和指数范围的组合。在这里，指数范围更为重要。

图6-1显示IEEE浮点标准754中单精度表示和双精度表示的格式（IEEE，1985）。关于IEEE格式的细节可从（Tanenbaum，2005）中读到。



351

图6-1 IEEE浮点格式

6.2.1.3 复数

一些程序设计语言支持复数数据类型，例如，Fortran和Python。复数值用浮点值的有序对来表示。在Python中，复数字面量的虚数部分用后接j或J来指定。例如，

```
(7+3j)
```

支持复数类型的语言包含了在复数值上的算术操作。

6.2.1.4 小数

大多数为支持商务系统应用设计的较大型计算机具有支持小数数据类型的硬件。小数数据类型存储固定数目的小数位，而小数点处于数值中的一个固定位置上。这些类型是商务数据处理中的主要数据类型，并因此成为COBOL语言中的关键成分。C#也具有一种小数数据类型。

小数类型具有的优点是，它至少能够在有限范围以内精确存储小数数值。而浮点类型却不能够做到。例如，数字0.1（十进制）在十进制中能够精确表示，但是在浮点类型中却不能，正

⊖ 十进制的0.1等于二进制的0.0001100110011…。

如在6.2.1.2节中我们所看到的。小数类型的缺点是，因为不允许有指数，小数数值的范围受限，并且它们在存储器中的表示十分浪费。

小数类型的存储与字符串的存储非常类似，都是使用二进制码来表示十进制数字。这样的表示方式被称为**二进制编码的十进制数**（binary coded decimal，BCD）。在某些情况下，每一个数字被存储于一个字节中，而在另一些情况下，每两个数字被放进一个字节。无论哪一种方式，它们都比二进制表示占据的存储空间更多，它们至少需要4个位进行一个十进制数字编码。因此，存储一个6位数字的十进制数就需要24个位的存储空间，而在二进制中存储相同的数只需要20个位。[1]小数数值上的操作是在具有这种功能的机器的硬件上进行的；不然的话就使用软件来模拟这些操作。

6.2.2 布尔类型

布尔类型也许是所有类型中最简单的类型。这种类型值的范围只有两个元素，一个为真，一个为假。布尔类型由ALGOL 60引入，并被引进大部分20世纪60年代以后设计的通用语言中，有一个例外则是广为流行的C89语言，在这种语言中能够使用数值表达式作为条件。在这样的表达式中，所有非零值的操作数都被认为是真，而具有零值的操作数则被认为是假，虽然C 99以及C++语言具有布尔类型，但它们还是允许使用数值表达式代替布尔类型的使用。但这在Java和C#中是不允许的。

在程序中，布尔类型常被用来表示转换（switch）或标志（flag）。尽管其他类型，如整数，也可以用作同样的目的，但使用布尔类型能够获得更好的可读性。

可以由单个位来表示布尔值，但因为在许多机器上难以有效地存取存储器中的单个位，所以常常将布尔值储存于存储器中最小但能够高效寻址的单位中，是在一个字节中。 252

6.2.3 字符类型

字符数据是以数值编码的形式存储于计算机中。最普遍使用的数值编码是8位的ASCII码（American Standard Code for Information Interchange），这种编码使用0～127的数值来对128个不同字符进行编码。ISO 8859-1是另一种8位字符编码，但它允许256个不同的字符。Ada 95使用ISO 8859-1码。

因为商务的全球化，以及计算机间全球性交流的需要，ASCII的字符集很快就不够用了。人们因而开发了一种称为Unicode码的16位字符集。Unicode包括了世界上大多数自然语言中的字符。例如，Unicode包括了用于塞尔维亚语的Cyrillic字母，以及泰国人使用的数字。Unicode中的前128个字符与ASCII码中相同。Java是第一种使用Unicode字符集的广泛应用的语言。此后，Unicode也被用于JavaScript、Python、Perl以及C#中。

为了提供处理单个字符代码的方法，许多程序设计语言都包括了字符基本类型。然而，Python支持单字符，并把该字符作为长度为1的字符串。

6.3 字符串类型

字符串类型是这样一种类型：这种类型的值由字符序列组成。字符串常量用来标记输出，因为各种数据的输入与输出常常是以字符串的形式来完成的。当然，字符串也是所有进行字符处理程序中的一个基本类型。

[1] 当然，除非程序需要维护大量的六十进制位，不然区别是很微小的。

6.3.1 设计问题

字符串类型所特有的两个最重要的设计问题是：

- 字符串应该仅仅是一种特殊种类的字符数组，还是一种基本类型？
- 字符串应该具有静态长度还是动态长度？

6.3.2 字符串及其操作

一般的字符串操作包括赋值、连接、子字符串引用、比较和模式匹配。

子串引用是对给定字符串中的一个子串的引用。子串引用将在学习数组时再进行讨论。数组中的子串引用称为片（slice）。

通常情况下，由为具有不同长度操作数赋值和比较的可能，因此字符串的赋值和比较操作是复杂的。例如，当把一个长字符串赋给一个短字符串时将发生什么，反之，又会怎么样？通常，根据实际情况作出相应简单合理的选择，尽管用户很难记住它们。

模式匹配是另一种基本的字符字符串操作。在一些语言中，模式匹配直接受到支持，在另一些语言中，它通过函数或类库得到支持。

如果不将字符串定义为基本类型，字符串数据则通常会被储存于字符数组中，并会像数组
253 数据一样被引用。这是C以及C++采用的方法。

C和C++使用char数组来存储字符串，并由一个头文件为string.h的标准程序库来提供一组字符串操作。大多数字符串的使用以及大部分程序库函数都遵循由特殊字符null终止字符串的协定，null用零值表示。这是保持字符串变量长度的一种替代方法。当程序库的操作作用于一个串时，它将一直进行到null字符出现为止。构造串的库函数常常提供了这种null字符。由编译器构造的字符串常量也具有null字符。例如下面的声明：

```
char str[] = "apples";
```

在这个例子中，str是一个char指针，被设置用来指向字符串apples0，这里的0就是null字符。

在C和C++中，用于字符串的最常用库函数有strcpy，它被用来进行串的复制；strcat，它的作用是将给定的串连接到另一个串上；strcmp，它的作用是按字符来比较两个给定的串（按照字符编码的次序）；strlen，它返回给定串中的字符数目，但不包括null字符。大多数
254 串处理函数的参数及返回值是一些指向char数组的char指针。参数也可以是字符串字面常量。

C标准库中的串操作函数在C++语言中也可以使用，但它们本身是不安全的，并已经造成了无数的安全问题。这是由于这种移动串数据的库函数不管制溢出范围的问题。例如，对于库函数strcpy的调用：

```
strcpy(src, dest);
```

如果dest的长度为20，而src的长度为50，strcpy将会覆盖dest后面30个字符（通常在运行时栈）的区域，而这个区域通常是在运行栈中。这里的问题是strcpy不知道dest的长度，因而不能够保证它不会覆盖dest后面的存储空间。C库中的几个其他字符串函数也存在同样的问题，包括C风格字符串。C++也通过与Java相似的标准类来支持字符串。因为C字符串库的不安全性，C++的程序人员应该使用标准库中的string类，而不是使用char数组以及C中的字符串库。

Fortran 95将串作为基本类型来处理，并且还提供了赋值、关系操作符、连接以及子串引用操作。

在Java中通过String类和StringBuffer类，Java支持字符串为基本类型。String类的值为常量字符串，而StringBuffer的值是可变的，并且较类似于字符数组。StringBuffer类中的变量允许具有下标。C#和Ruby中的字符串类与分类似于Java中的字符串类。

Python也把字符串作为基本类型，并且有子字符串引用、连接、索引访问单个字符以及查找和替代方法等操作。字符串也有字符成员操作。因此，尽管Python字符串是基本类型，但是它们执行的字符和子字符串引用更像是字符数组。然而，Python字符串是不可变的，这与Java的String类对象很相似。

历史注释

SNOBOL 4是第一种支持高级的支持模式匹配的语言。

Perl、JavaScript、Ruby以及PHP中包括了内置的模式匹配操作。在这些语言中，模式匹配表达式都或多或少地基于数学的正则表达式。事实上，常常就称它们为正则表达式。它们从早期 UNIX行编辑器ed演变成为UNIX shell语言的一部分，最终变成了目前所具有的复杂形式。如果要解释模式匹配表达式，至少需要完整的一本书（Friedl，1997）。本节只是想通过两个相对简单的例子来提供关于这些表达式风格的一个概貌。

考虑下面的模式表达式：

```
/[A-Za-z][A-Za-z\d]*/
```

这种模式匹配了（或描述了）程序设计语言中的典型名字形式。方括号包起的是字符类。第一字符类说明了所有字母；第二类则说明了所有的字母以及数字（数字使用缩写 \d 来说明）。如果仅仅是说明了第二类，我们不能阻止名字从数字开始。紧跟第二个类的加号操作符说明必定存在一个或多个这种类的实例。这样，与整个模式相匹配的字符串开始于一个字母，后面跟随一个或者多个字母或数字。

下面再考虑另一个模式表达式：

```
/\d+\.?\d*|\.\d+/
```

这个模式匹配数值字面常量。这里的“\.”说明一个字面意义的小数点。[⊖]问号说明它所跟随的事物出现零次或一次。垂直线“|”分开这个模式的两种可能选择。第一种选择匹配的串是一个或者多个数字，可能跟随着一个小数点，再跟随零个或者多个数字；第二行选择匹配的串开始于一个小数点，小数点之后跟随一个或者多个数字。

C++、Java、Python和C#的类库包括了模式匹配的功能。

6.3.3 串长度选择

关于字符串值的长度有几种设计选择。第一种选择是这种长度可以为静态的，并可以在声明语句里进行说明，我们称这样的串为静态长度串。这正是一种设计选择，如Python的字符串，Java的String类中的不可变对象，C++标准类库中类似的类、Ruby的内建String类，以及可用于C#的.NET类库。

第二种选择是允许串具有可变长度，且将其上限固定为由变量定义所设定的最大值。这样的例子有C以及C++中C风格的串。这种串被称为有限动态长度串。这行由变量能够存储零与最大值之间任意数目的字符。回忆C语言中的串，它们使用一个特别的字符来作为串里字符的结尾，而不是保持串的长度。

⊖ 句点常必须加以转义，主要是因为在正则表达式中句点有特殊的含义。

第三种选择是允许串具有可变的长度，而并不限制长度的最大值，如JavaScript和Perl中的情形。这种类型的串被称为动态长度串。这种选择具有动态存储空间分配以及动态解除分配的额外代价，但能够提供最大限度的灵活性。

Ada 95支持所有的这三种选择。Standard包中的String类型提供静态长度串，Ada.Strings.Bounded包中的Bounded_String类型支持有限动态长度串，而Ada.Strings.Unbounded包中的Unbounded_String类型则支持动态长度串。

6.3.4 评估

串类型对一种语言的可写性十分重要。把串作为数组来处理，可能比将它作为基本类型来处理更为麻烦。把串作为一种基本类型加到语言中，就语言或者编译器的复杂性而言，所增加的气价并不高。另面，人们乐意为一些当代语言省略基本串类型来进行维护。当然，实施串处理的标准库可以常外没有将串包括进基本类型的缺陷。

256

字符串操作，例如简单的模式匹配及连接，是非常必要的，应该将它包括进来，以便对串类型的值进行操作。尽管动态长度串显然最灵活，但是必须在实现时所需的额外开销与其所带来的灵活性之间进行权衡。

6.3.5 字符串类型的实现

字符串类型可以由硬件直接支持，但在大多数情况下，是使用软件来实现串的存储、检索以及处理。当字符串类型被表示为字符数组时，这种语言常常不提供什么操作。

只在编译期间需要的静态字符串类型，其描述符具有三个域。每一个描述符的第一个域都是类型的名字。在静态字符串的情况下，第二个域是类型的长度（以字符计）。第二个域则是第一个字符的地址。图6-2显示了这种描述符。有限动态串需要一个运行时描述符来存储固定的最大长度以及当前长度，如图6-3所示。动态长度串则要求一个较简单的运行时描述符，因为它只需要储存当前的长度。虽然我们将描述符描绘成为独立的块，但在大多数情况下它们被存放于符号表中。



257

图6-2 静态字符串的编译时描述符



图6-3 有限动态串的运行时描述符

C以及C++中的有限动态串不需要运行时描述符，因为这种串的结尾由null字符标记。它们也不需要限制最大长度，因为在这些语言中数组引用不具有下标值范围的检测。

静态长度串以及有限动态长度串不需要特别的动态存储分配。对于有限动态长度串，当串变量与存储空间绑定时，就已经为最大长度分配了充分的存储空间。其面只需要进行一次分配的过程。

动态长度串则需要较复杂的存储管理。串的长度以及与长度关联的串所绑定的存储空间必须动态地变大和变小。

关于动态长度串所需的动态存储分配问题，有两种可能的解决方式。第一种方式是可以将

串储存于链表中，这样当串变长时，就可以从堆上获取所需要的新单位。这种方法的缺点是链表中的链接占据了大量的存储空间，以及施行串操作必然带来的复杂性。

第二种方法是存储字符串作为指向堆中分配的单个字符的指针数组。这种方法仍然使用了额外的内存，但是字符串处理方法比链表方法更快。

第三种替代方案是在相邻的存储单位中储存完整的串，而当串变长时这种方法产生的问题是：怎样为串变量继续分配那些与已经存在的存储单位相邻的存储空间呢？常常，这样的存储空间已经用于其他用途，不能使用。新的替代方式是在存储器中找一个能存储完整串的新区域，并将旧的串中部分移到这个新的区域中，接下来，再将旧的存储单位进行解除分配。后面的这种方式是现在典型的使用方法。6.9.9节将讨论处理可变大小存储段的分配与解除分配的一般问题。

尽管链表方法需要更多的存储空间，但相关的分配与解除分配过程却很简单。然而，一些字符串操作因需要追查指针而变慢。另一方面，为完整字符串使用相邻内存单元会加速字符串操作，而且所需的存储空间也显著减少，但分配与解除分配过程变慢。

6.4 用户定义的序数类型

序数类型是这样一种类型：这种类型的可能值的范围能够很容易地与一组正整数相关联。例如，在Java中的基本序数类型有integer（整数）、char（字符）以及boolean（布尔）类型。程序设计语言支持两种用户定义的序数类型：枚举类型和子范围类型。

6.4.1 枚举类型

枚举类型是这样一种类型：这种类型的所有可能值都是在类型定义中枚举的命名常量。枚举类型提供了定义以及组合命名常量的一种方式，而这种命名常量就被称为**枚举常量**。下面C# 258
语言的例子是一种典型的枚举类型的定义方式：

```
enum days {Mon, Tue, Wed, Thu, Fri, Sat, Sun};
```

典型地，枚举常量被隐式地赋予整数值0，1，…，但也可以在类型定义中被显式地赋予任何整数字面值。

枚举类型所特有的主要设计问题是：

- 是否允许枚举常量出现在一种以上的类型定义中？如果允许，在程序中当这个常量出现时，如何对它们的类型进行检测？
- 可以将枚举值强制转换为整数吗？
- 可以将任何其他类型强制转换为枚举类型吗？

所有这些设计问题都与类型检测有关。如果可以将一个枚举类型强制转换为一个整数类型，我们对于其合法操作范围以及取值范围就几乎没有控制。如果可以将一个int类型的值强制转换为一个枚举类型，就可以将任何整数值赋予枚举类型的变量，而不论这个数值是否表示一个枚举常量。

6.4.1.1 设计

在没有枚举类型的语言中，程序人员通常是使用整数值来模拟枚举类型。假若我们需要在C程序中表示颜色，C没有一种枚举类型，我们可以使用0来表示蓝色，使用1来表示红色等。可以如下所示来定义这些值：

```
int red = 0, blue = 1;
```

现在，在这条程序里我们就可以使用RED和BLUE，就像它们是颜色类型一样。这种方式所

带来的问题是，因为我们并没有定义颜色类型，当使用RED和BLUE时不能够进行类型检测。例如，将RED和BLUE相加是合法的操作。虽然我们并不想有这样的运算，RED和BLUE还可能被与任何算术操作符以及任何类型的操作数结合在一起，而这些结合行为极少有用。另外，因为RED和BLUE实质上仅仅是变量，所以它们可能会被赋予任何整数值，从而弄丢掉它们与颜色之间的关联。关于最后面的这个问题，可以通过将RED和BLUE定义为命名常量而得以避免。

C和Pascal是首批拥有枚举类型的广泛应用的语言，C++则包含了C中的枚举类型。在C++中，我们可以有

259
```
enum colors {red, blue, green, yellow, black};
colors myColor = blue, yourColor = red;
```

类型colors使用了枚举常量的默认内部值，0，1，…，虽然也可以对枚举常量赋予任何其他的整数值(或者甚至是任何常量值的表达式)。当将枚举常量放置于整数的上下文中时，枚举值就被强制转换成为int类型，这样就允许将它们用于任何数值表达式中。例如，如果myColor的当前值是blue，表达式

```
myColor++
```

将会赋予myColor以green值。

C++也允许将枚举常量赋给任何数值类型的变量，虽然这常常会是一个错误，但是在C++中，任何其他类型的值都不允许强制转换成枚举常量。例如

```
myColor = 4;
```

是非法的。如果将这条赋值语句的右边显式地转换成colors类型，这条语句就会是合法的。这种做法防止了一些潜在错误的发生。

C++中的枚举常量仅能够出现在同一引用环境里的一种枚举类型中。

在Ada中的枚举字面常量被允许出现在同一引用环境里的多个声明中。这样的字面常量被称为**重载字面常量**。解决重载问题的规则，即确定这样的字面常量在某次出现时的类型，是必须能够从它所出现的上下文来确定其类型。例如，如果将一个重载字面常量与一个枚举变量进行比较，这个字面常量的类型就将被确定为那个变量的类型。在某些情况下，程序人员必须为某一次出现的重载字面常量标明一些类型说明。

因为不能够将Ada中的枚举字面常量和枚举变量强制转换为整数，所以Ada中的枚举类型操作的范围以及枚举类型的值的范围都是受限，这使得编译器能够检查出出许多程序人员的错误。

在2004年，Java 5.0版本中加入了一种枚举类型，在Java语言中的所有枚举类型都隐式地为预定义类Enum的子类。而且为枚举类型是类，它们就可以具有对象数据域，构造函数以及方法。在语法上，Java中枚举类型定义的表现十分类似于C#中的枚举类型定义，除了Java中的枚举类型可以包括域、构造函数以及方法之外。一个枚举类可能具有的值仅仅就是枚举类所有可能的对象。所有的枚举类型都继承toString以及一些其他方法。使用静态方法values可以获取一个枚举类型的对象数组。而使用方法ordinal可以获得一个枚举变量的内部数值的值。不可以将任何其他类型的表达式赋给枚举变量，而且，不可以将枚举变量强制转换为任何其他的类型。

C#中的枚举类型与C++中的枚举类型很类似。但是在C#中不能够将枚举类型强制转换为整
260 数。相比，可以将枚举类型的操作限制在具有意义的那些操作上。而且可以将值的范围限制在特定枚举类型的值之中。

有趣的是，最近没有一种脚本语言包含枚举类型，其中包括Perl、JavaScript、PHP、Python和Ruby。甚至在枚举类型加入Java以前，Java已经有10年的历史了。

6.4.1.2 评估

枚举类型在可读性和可靠性两方面都提供了优越性。可读性的提高是由于一种非常直接的方式：即，可以容易地辨认命名的值，但不容易辨认编码的值。

在可靠性方面，Ada，C#以及Java 5.0中的枚举类型提供了两种好处。首先，没有任何一种算术操作对枚举类型是合法的，这样就防止如将一周中的星期一与星期二相加的类似运算。其次，不能够给枚举类型赋以其定义范围之外的任何值。如果前面的colors枚举类型具有10个枚举常量，并使用0，…，9作为它的内部值的话，那么就不可以将任何大于9的数值赋给colors类型变量。

因为C对待枚举变量就像对待整数变量一样，因而C不具有上面所述的这两种优点。

C++比C语言稍稍进步，可以将数值赋给枚举类型的变量，但仅当可以将这些数值转换成被赋值变量的类型时。赋给枚举类型变量的数值将被检测，用以确定这些数值是否在这种枚举类型的内部值范围之内。然而不幸的是，如果用户采用了一个很广泛的值范围，这种检测就将非常的低效。例如，

```
enum colors {red = 1, blue = 1000, green = 1000000}
```

在这个例子中，任何一个赋给colors类型变量的数值都将被检测，只是为了确定它是否在1，…，1 000 000的范围之内。

Ada，C#以及Java 5.0中的枚举类型，较之C++中的枚举类型优越得多，因为这些语言不强制转换枚举类型变量为整数类型。

6.4.2 子范围类型

子范围类型是一个序数类型的连续子系列。例如，12，…，14是整数的子范围。子范围类型由Pascal引进，并被包括在Ada中。没有关于子范围类型的特殊的设计问题。

6.4.2.1 Ada中的设计

261

在Ada语言中，子范围被包括在一类称为子类型（subtype）的类型中。正如我们曾经在第5章所陈述的，子类型完全不是新的类型，它们只是一个新名称，用于那些已有类型的可能受限版本。例如，考虑下面的这些声明：

```
type Days is (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
subtype Weekdays is Days range Mon..Fri;
subtype Index is Integer range 1..100;
```

在这些例子中，已有类型上所具有的范围是这些类型可能值的范围。所有为父类型定义的操作也被定义给子类型，除了在说明范围之外的赋值。例如，在下面的语句中：

```
Day1 : Days;
Day2 : Weekdays;
...
Day2 := Day1;
```

如果Day1的值不是Sat或者Sun，这种赋值就是合法的。

编译器必须对每一个子范围类型变量的赋值都产生范围检测代码。尽管类型的检测是在编译时进行，而子范围类型的检测则在运行时进行。

用户定义的序数类型的最普遍应用之一是数组索引。关于这些我们将在第6.5节中进行讨论。数组索引也能够被用作循环变量。事实上，序数类型的子范围是说明Ada中for循环变量范围的

唯一方法。

请注意，子范围类型与在第5章里讨论过的Ada的派生类型非常不同。例如，考虑下面的类型声明：

```
type Derived_Small_Int is new Integer range 1..100;
subtype Subrange_Small_Int is Integer range 1..100;
```

Derived_Small_Int和Subrange_Small_Int这两种类型的变量有相同的合法值的范围和都继承了Integer的操作。然而，类型Derived_Small_Int的变量与任何Integer类型都不兼容。而类型Subrange_Small_Int的变量与Integer类型的变量及常量以及任何Integer的子类型都兼容。

6.4.2.2 评估

子范围类型使得读者们清楚了子类型的变量只能储存一定范围内的值。这样提高了可读性。可靠性也由于子范围类型的使用而得到了增强，因为如果将声明范围以外的值赋给一个子范围
262 变量，这将被编译器（在所赋的值为字面常量值的情形下）或者运行时系统（在所赋的值为变量或表达式的情形下）发现为一个错误。奇怪的是在当代语言中，只有Ada 95具有子范围类型。

6.4.3 实现用户定义的序数类型

如前所述，枚举类型通常被实现为整数。但如果不对枚举类型的值的范围以及操作施加限制的话，这将不会有助于提高可靠性。

子范围类型的实现方式与其父类型的实现完全相同，唯一的差别是，编译器必须范式地对子范围变量或表达式的每一次赋值进行范围检测。虽然这样增加了代码的规模以及执行的时间，但通常认为这种代价是值得付出的。除此之外，一个良好的优化编译器可以优化掉一些这样的检测。

6.5 数组类型

数组是同种类的数据元素的聚集，其中每一单个元素的标识是由这个元素在聚集中所占据的位置相对于第一个元素的位置来确认。在程序中对一个数组元素的引用，常常包括了一个或者多个非常量的下标。这种引用需要额外的运行时的计算来确定被引用的存储空间位置。数组的单个数据元素具有已经定义的类型，或者是基本类型或者是其他类型。大部分计算机程序需要模拟数值集合，集合中的数值都是相同类型的，并且必须使用相同的处理方式。因而对于数组所存在的普遍需求是显而易见的。

6.5.1 设计问题

数组所特有的主要设计问题如下：

- 什么类型用于下标是合法的？
- 应该对元素引用中的下标表达式进行范围检测吗？
- 什么时候下标范围被绑定？
- 什么时候对数组进行存储空间的分配？
- 应该允许参差不齐的或者整齐长方形的多维数组吗？或者，这两种数组都应该被允许？
- 当数组被分配了存储空间时，能否实施数组的初始化？
- 如果允许片的话，应该允许什么类型的片？

在下面的几个小节里，我们将讨论在几种最常用的程序设计语言中进行数组设计选择的例子。 263

6.5.2 数组与下标

数组中特定元素的引用是通过一种两个层次的语法机制进行的，其中的第一个部分是数组名，第二个部分则是一个可能为动态的选择器，它包括了一项或多项的下标或索引。如果在一个引用中，所有下标都为常量，这个选择器是静态的；否则选择器就是动态的。可以认为选择操作是从数组名与下标值到数组中元素的一种映射。事实上，有时将数组称为有限映射，这种映射可以用符号表示成为

数组_名（下标_值_表）→ 元素

数组引用的语法十分通用：数组名后面跟随一系列下标，使用圆括号或方括号将下标括起来。如果一种语言中的多维数组是数组中的数组，则每一个下标都会出现在自己的括号中。使用圆括号存在的一个问题是，圆括号也常被用来包括子程序的调用参数；这会使得数组引用与函数调用在形式上完全一样。例如，考虑下面的Ada赋值语句：

```
Sum := Sum + B(I);
```

因为圆括号被用于子程序参数，又被用于Ada中数组的下标，程序的阅读者以及编译器都必须通过其他信息来确定这条赋值语句中的B(I)究竟是函数调用还是对数组中元素的引用？这给读者带来一些不必要的麻烦。

Ada语言的设计人员特别选用圆括号来包括下标，这样表达式中的数组引用与函数调用之间就一致起来，虽然这具有潜在的可读性问题。他们之所以做出这种选择，是基于这样的事实：即无论数组元素的引用还是函数的调用，在实质上都是一种映射。数组元素的引用将下标映射到这个数组中的某一特定元素之上；函数的调用将实参映射到函数定义之上，并且最终映射到函数值上。

基于C的语言使用方括号来界定数组下标。

在一个数组类型中涉及了两种不同的类型：元素的类型和下标的类型。下标类型通常为整数的子范围，但是Ada允许使用其他一些类型作为下标，如布尔类型、字符类型以及枚举类型。例如，Ada有

历史注释

Fortran 90之前的Fortran语言以及PL/I语言以前的设计人员选择圆括号来包括数组下标，因为在当时没有比它更合适的字符可用。那时的穿孔卡不具有方括号字符。 264

```
type Week_Day_Type is (Monday, Tuesday, Wednesday, Thurs-
day, Friday);
type Sales is array (Week_Day_Type) of Float;
```

Ada的for循环能为它的计数器使用任何序数类型变量，我们将在第8章看到这种情况。这使得可以方便地处理带有序数下标的数组。

早期的程序设计语言不要求下标范围的隐式检测。下标的范围错误在程序中常见，因而范围检测的要求是决定语言可靠性的重要因素。在当代语言中，C、C++、Perl和Fortran不要求下标范围的检测，但是Java、ML以及C#则要求施行下标范围的检测。默认情况下，Ada语言自动检测所有的下标范围，但程序人员也可以取消这项功能。

因为在Perl中，数组元素总是标量，标量名用美元标记（$）开始（对数组元素的引用使用美元标记，而不是名字中的其他标记），所以由于所有数组名都以标记（@）开始，下标显得比较独特。例如，在数组@list中，用$list[1]来引用第二个元素。

在下标值是基于数组末尾的偏移量时，Perl也能用一个负下标引用一个数组元素。例如，如果数组@list有5个元素，下标是0..4，$list[-2]引用下标为3的元素。在Perl中引用一个不存在的元素产生undef，但是不报告错误。

6.5.3 下标绑定及数组类别

下标的类型以及数组变量的绑定通常都为静态的，但下标值范围的绑定有时为动态的。

在一些语言中，下标范围下界的绑定为隐式的。例如，在基于C的语言中，总将所有下标范围的下界绑定为零；在Fortran 95中，则将它默认为1，而在其他一些语言中，就必须完全由程序人员来说明下标的范围。

一共有五种类型的数组。这种归类是基于下标范围的绑定，存储空间的绑定和分配的存储器位置来定义的。类别名表示这4类的设计选择。在前面4个类别中，一旦绑定了下标范围，并且分配了存储空间，它们在变量的整个生命周期将保持固定，注意，当下标范围是固定的，数组不会改变其大小。

静态数组的下标范围是静态地被绑定的，它的存储空间也被静态分配（完成于运行时之前）。静态数组的优点是高效率：不需要动态地进行分配或者解除分配。

固定栈动态数组的下标范围是动态地被绑定的，但是它的存储空间分配则完成于执行期间。当确立它的声明语句之时，固定栈动态数组相对于静态数组的优越性是空间效率。一个子程序中的大数组可以与另一个子程序中的大数组共享同一空间，只要这两个子程序不在同时活跃。

栈动态数组的下标范围是动态绑定的，它的存储空间的分配也是动态的（完成于运行时）。然而，一旦它的下标范围被绑定，它的存储空间也被分配，它们将在变量的生存期里保持不变。栈动态数组相对于静态数组以及固定栈动态数组的优越性是它的灵活性，直到将要使用它时才需要知道这种数组的大小。

固定堆动态数组类似于固定栈动态数组，它的下标范围是动态绑定的，并且它与存储空间的绑定也是动态的，但这两种绑定在分配存储空间之后就成为固定不变的。与固定栈动态数组不同的是：固定堆动态数组的绑定是在执行时用户程序发出请求时进行的，此外，它的存储空间是在堆中分配，而不是从栈中分配。

堆动态数组的下标范围的绑定以及存储空间的分配都是动态的，而且在数组的生存期内可以进行任意次数的改变。堆动态数组相对于其他类型数组的优越性是它的灵活性：在程序执行期间，可以根据空间变化的需要来改变数组的大小，可以变大或者变小。下文给出了这五种类别数组的例子。

在C以及C++的函数中，声明时带有**static**修饰符的数组是静态的。

在C以及C++的函数中，声明时不具有**static**修饰符的数组是固定栈动态数组的范例。

Ada中的数组可以为栈动态的，如下面的句子所示：

```
Get(List_Len);
declare
  List : array (1..List_Len) of Integer;
  begin
  ...
  end;
```

在这个例子中，用户输入数组List所需要元素的个数，当程序执行到达declare块时，就动态地进行存储空间的分配。而当执行到达程序块的末尾时，数组List则被解除分配。

C和C++语言也提供固定堆动态数组。标准库函数malloc和free（它们是一般的堆分配和解除分配操作）能够用于普通的C数组。C++则使用new和delete操作符来管理堆存储空间。数组被处理为指向存储单位集合的指针，而且这些指针可以被索引，第6.9.5节将讨论这些内容。

Fortran 95也支持固定堆动态数组。

在Java中，所有数组都是固定堆动态数组。这些数组一经建立，它们将保持同样的下标范围及存储空间。C#也提供固定堆动态数组。 266

C#还包括了第二种数组类ArrayList，以提供堆动态数组。这种类中的对象在建立时不具有任何元素，例如

```
ArrayList intList = new ArrayList();
```

通过使用方法Add将元素加入到对象中，例如

```
ArrayList.Add(nextOne);
```

Java包含了与C#的ArrayList相似的结构，除了不支持下标之后，其他都相同（必须通过get和set方法来访问元素）。

通过使用push（在数组的尾部放置一个或多个新元素）和unshift（在数组的开始处放置一个或多个新元素），或通过赋给数组一个超过其最当前下标的值，Perl数组能够增长。通过赋值空列表()，数组能缩小为0个元素。数组的大小等于最大下标加1。

历史注释

Fortran I将数组下标数目限制为3，因为在设计的当时，主要关心的是执行效率。Fortran I的设计人员使用IBM 704机的指令有序创建了存取三维数组元素的快速方法。Fortran IV是第一种实现于IBM 7094机器之上，具备七维数组引用的语言。这使得Fortran IV的设计人员允许数组的下标多达七维。大多数的当代语言都已经没有了这种限制。

像Perl一样，JavaScript允许使用push和unshift方法让数组扩展和设置为空列表使其缩小。然而，它不支持下标。

JavaScript数组是松散的，这意味着下标值不需要连续。例如，假如我们有称为list的数组，它有10个元素，下标为0…9，[O]考虑赋值语句

```
list[50]=42;
```

现在，列表有11个元素，长度为51。具有下标11…49的元素没有定义，因此不需要存储空间。在JavaScript数组中，对不存在元素的引用将产生undefined。

在Python和Ruby中的数组仅能够通过添加元素或连接其他数组方法。Ruby支持负下标，但是Python不支持。Python和Ruby都能删除元素或数组片。在Python中，对不存在的元素的引用导致运行时错误，因此在Ruby中相似的引用将产生nil，并不报告错误。

6.5.4 异构数组

异构数组是指具有不同类型元素的数组。Perl、Python、JavaScript和Ruby都支持这类数组。在所有这些语言中，数组都是在堆上动态分配的。 267

在Perl中，数组元素能够是包含数字、字符串和引用的标量类型的任意混合。JavaScript是一种动态类型语言，任何数组元素都能是任何类型。Python和Ruby数组是对任何类型的对象的引用。

O 一些实现能够容易地与1000…1009。

当我们在本章后面部分讨论数组时，异构数组也会谈及到。

6.5.5 数组初始化

一些语言提供了在分配数组的存储空间之后设定数组初始值的方法。在Fortran 95中，可以通过在数组声明中给数组赋予一组数值，从而实施数组的初始化。当数组为一维时，这组数值就是一列使用括号与斜线包围的字面常量。例如，我们可以有

```
Integer, Dimension (3) :: List = (/0, 5, 5/)
```

C、C++、Java以及C#也允许施行数组的初始化，但使用了一种新的样式。如下面的C的声明

```
int list [] = {4, 5, 7, 83};
```

即由编译器设定了数组的长度。这种方法十分方便，但却具有代价。它有实际上消除由系统来发现程序人员的某类错误的可能性，比如说，错误地漏掉了数组中的一个数值。

正如在6.3.2节讨论的，C和C++语言中的字符串被实现成char数组，可以将这些数组的初值设为串常量，例如

```
char name [] = "freddie";
```

数组name具有八个元素，因为所有串都由空字符null（零）结束，空字符是系统为字符常量隐式提供的。

在C和C++中的字符串数组也能够使用字符字面常量来设定初值。在这种情况下，数组是指向字符的一个指针，例如，

```
char *names [] = {"Bob", "Jake", "Darcie"};
```

这个例子说明了C和C++中字符字面常量的性质。在前面的例子中，曾经使用一个字符串字面常量来为char数组name赋以初值，在这里将字面常量处理为char数组。但是在后面的这一
268 个例子（names）中，则将字面常量处理为指向字符的指针，因而这个数组是指向字符的指针的数组。例如names[0]是指向文字字符数组中的字母"B"的指针，在这个数组里包含了字符"B"、"o"、"b"以及空字符null。

在Java中使用类似的语法来对对象String的引用数组元素定义并实施初始化，例如，

```
String[] names = ["Bob", "Jake", "Darcie"];
```

Ada语言提供了两种在声明语句中设定数组初值的机制：1.通过将它们按将要被存储的顺序排列；2.通过使用 => 操作符（这个操作符在Ada中被称为arrow）将它们直接赋给一个索引地址。例如下面的例子：

```
List : array (1..5) of Integer := (1, 3, 5, 7, 9);
Bunch : array (1..5) of Integer := (1 => 17, 3 => 34,
                                    others => 0);
```

在上面的第一条语句中，数组List中的所有元素都被赋予了初值。这些数值出现的次序就是它们在数组中的元素位置。在第二条语句中，使用了直接赋值方法来给第一和第三个数组元素设定初值，而使用others子句给其余元素设定初值。这种由括号包括的数值集合称为聚集值。

6.5.6 数组操作

数组操作是将数组作为一个单位来进行的操作，最常用的数组操作是赋值、连接、比较（相等和不相等）和片（在6.5.8节中讨论）。

除了通过Java、C++和C#的方法，基于C的语言不提供任何数组操作。Perl支持数组赋值，但是不支持比较。

Ada允许数组的赋值，包括那些右边为聚集值而非数组名称的赋值。Ada还提供了由符号&说明的连接，这种连接被定义于两个一维数组之间，以及一个一维数组与一个标量之间。几乎 269
在Ada中的所有类型都具有内建的等于关系操作符和不等于关系操作符。

Python数组的元素是对对象的引用。Python提供了数组赋值操作，尽管它只是引用的改变。Python也有数组连接（+）和元素成员（in）。它包括两个不同的比较操作符，一个操作符决定两个变量是否引用同一个对象（is），另一个操作符比较在引用对象中的所有对应的对象，而不管它们嵌套得有多深，如等于（==）。

与Python一样，Ruby数组的元素是对对象的引用。同样，当Ruby在两个数组中使用==操作符时，只有在两个数组有相同的长度，并且对应的元素都相等时，结果才为真。Ruby的数组能用Array方法连接。

Fortran 95包括了一些被称为元素的（elemental）数组操作，它们是在数组元素对之间进行的操作。例如，在两个数组之间的加法操作符（+）将产生一个数组，其元素为这两个数组元素对之和。可以为任何大小或形状的数组将赋值操作符、算术操作符、关系操作符以及逻辑操作符重载为数组操作符。Fortran 95还包括了进行矩阵乘法、矩阵转置以及矢量点积的内部函数或库函数。

数组及其操作是APL语言的核心。APL是有史以来功能最强的数组处理语言。然而由于它较为晦涩和缺乏对后继语言的影响力，我们只在这里对它的数组操作做简略的介绍。

在APL语言中，两种基本的算术操作都是为矢量（一维数组）和矩阵以及标量操作数而定义的。例如，

```
A + B
```

是有效的表达式。在这里A和B可以是标量变量，矢量或者矩阵。

APL包括了一组矢量和矩阵的一元操作符，我们将其中的一些列举如下（在这里，V为矢量，而M为矩阵）：

```
⍋V  对V的元素
⌽M  相当M矩阵
⊖M  相当M的行
⍉M  M的转置（行变成列，或者反过来）
⌹M  M的逆
```

APL还包括了几种特殊操作符，它们使用其他操作符作为操作数。这类操作符的一个例子就是内积操作符，它用点号（.）表示。这个操作符的两个操作数都是二元操作符，例如，

```
+.×
```

是一个新的操作符，它作用于两个变量，矢量或者矩阵。它首先将两个参数的对应元素相乘，然后取得其结果的和。例如，如果A和B都是矢量，

```
A × B
```

则是A和B的数学内积（即，A和B中对应元素的乘积的矢量），语句

```
A +.× B
```

则是A和B的内积之和。如果A和B为矩阵，这个表达式就说明了A和B的矩阵乘积。

APL中的特殊操作符实际上是函数的形式，我们将在第15章给以描述。 270

6.5.7 长方形数组与参差不齐数组

长方形数组是一个多维数组，数组中的所有行都具有同样数目的元素，数组中的所有列也都具有同样数目的元素。长方形数组准确地模拟了表格。

参差不齐数组是，其中行的长度不必相同。例如，一个参差不齐矩阵可能包括了3行，其中的一行有5个元素，另一行有7个元素，再另一行有12个元素。列的长度也不必相同。对于更高维数的数组也是如此。因而，如果一个数组具有第三维（层）的话，其中的每一层都可以具有不同数目的元素。如果将多维数组构造为数组之数组，我们就可以得到参差不齐数组。例如可以将一个矩阵构造为数组的数组。

C、C++以及Java支持参差不齐数组，而并不支持长方形数组。在这些语言中，对于一个多维数组中的元素的引用使用方括号来指示，每一对方括号分别对应于数组的一维。例如，

```
myArray[3][7]
```

Fortran, Ada以及C#支持长方形数组。（C#也支持参差不齐数组。）在这些语言中，将所有对数组元素的引用的下标表达式都放置于一对方括号中。例如，

```
myArray[3, 7]
```

6.5.8 片

数组的片是数组中的一个子结构。例如，如果A为一个矩阵，A的第一行就是一个可能的片，同样，A的最后一行以及第一列也可能是片。重要的是要知道，片并不是一个新的数据类型，它只是将数组的一部分作为一个单位来引用的一种机制。如果一种语言中的数组不能够被处理为一个单位，这种语言就不能够处理片。

考虑下面Fortran 95中的声明：

```
Integer, Dimension (10) :: Vector
Integer, Dimension (3, 3) :: Mat
Integer, Dimension (3, 3, 4) :: Cube
```

前面讲过，Fortran中数组下标的下限是1。Vector (3:6)是一个四个元素的数组，它包含Vector中第三到第六个元素；Mat (:,2)表示的是Mat的第二列；Mat (3, :)表示的是Mat的第三行。所有的这些引用都可以作为一维数组来使用。所有对数组中片的引用都可以处
271 理成为就像它们是剩余维数的数组一样。因而可以将一个片引用，例如Cube (:, :, 2)，合法地赋给Mat。片也可以出现在赋值语句的左边。例如，可以将一个一维数组赋给一个矩阵的片。图6-4显示了Mat和Cube的几个片。

Fortran 95可以说明更复杂的片。例如，Vector(2:10:2)是包括了Vector中第二、第四、第六、第八和第十个元素的五个元素的数组。片也可以具有一个现有数组元素的不规则的排列。例如，Vector ((/3, 2, 1, 8/))是一个具有Vector中第三、第二、第一和第八个元素的数组。

Perl支持两种形式的片：列表式的特定的下标或范围式的下标。例如，

```
@list[1..5] = @list2[3, 5, 7, 9, 13];
```

注意，因为片是数组（不是标量），所以片引用使用数组名，而不是标量名。Python也支持
272 简单的和复杂的数组片。例如，list[1:20:2]引用从下标1开始的每一个其他的列表元素；list[10:]引用所有下标大于10的元素（包括下标等于10的元素）；list[:5]引用所有下标小于5的元素（不包括下标等于5的元素）。

Ruby支持两种片，一种片的元素由开始下标和结束下标（不包括结束下标值）指定，另一种是，第一个给定的下标指定第一个元素，第二个指定片中元素的个数。

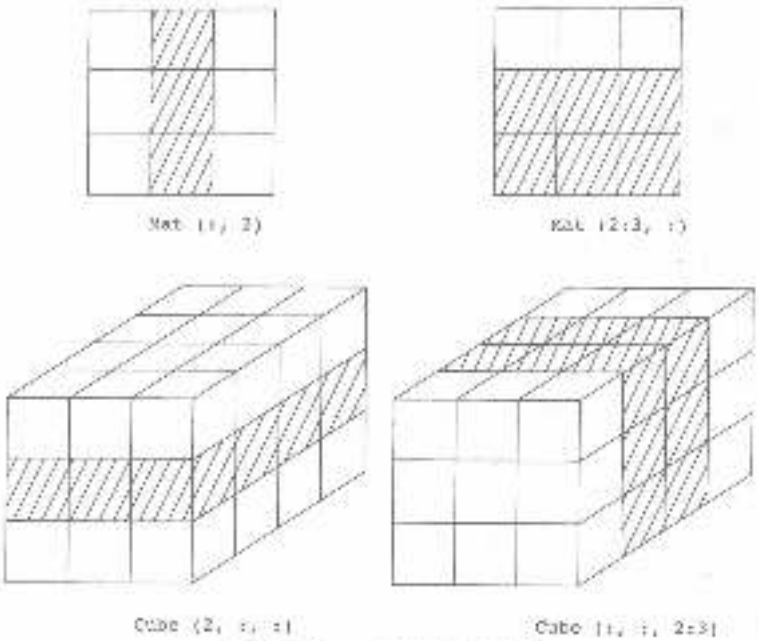


图6-9 Fortran 95中片的例子

在Ada中只允许有高度受限的片：即那些由一维数组的连续元素组成的片。例如，如果List是一个具有下标范围（1，…，100）的数组List(5..10)即是List的一个片，它包括了下标从5到10的6个List元素。如在6.3.2节的讨论，一个String类型的片被称为子串引用（substring reference）。

6.5.9 评估

事实上，所有的语言都包括了数组。数组结构简单，并且已经被很好地开发。自从Fortran I引入数组以来，数组所获得的主要的进步是将所有的序数类型包括进来作为可能的下标类型，当然还有动态数组。尽管数组非常基本和重要，涉及它们的设计争议却很少。

6.5.10 数组类型的实现

数组的实现比简单类型的实现（如整数）需要更多的编译时工作量。进行数组元素存取的代码必须产生于编译时，还必须在运行时执行这种代码，以产生元素的地址。对于下面这样的引用将要访问的地址，没有预先计算的方法

```
list[k]
```

一个一维数组是一串相邻的存储单位。假设定义数组list具有下界为1的下标范围。list的存取函数通常具有这样的形式

```
地址(list [k]) = 地址(list [1]) + (k - 1) * 元素的大小
```

可以将它简化为

```
地址(list [k]) = (地址(list [1]) - 元素的大小) + k * 元素的大小
```

这里，加法中的第一个操作数是这个存取函数的常量部分，而第二个操作数则是其变量部分。

如果将元素类型静态地绑定，并且数组与存储空间也是静态地绑定，那么就能够在运行时之前计算常量部分的值，只将加法和乘法操作留在运行时进行。如果数组的基址，也即起始地址，是直到运行时才获知，则必须在数组被分配存储空间之时进行减法。

对具有任意下限的数组实施存取的函数，它的一般形式为

273 地址(list[k]) = 地址(list[下限]) - (k - 下限) * 元素的大小

图6-5显示一个一维数组的编译时描述符所可能具有的形式。描述符包括了构造存取函数所必需的信息。如果不进行运行时下标范围的检测，并且所有的属性都为静态，那么在执行期间只需要存取函数，而不需要描述符。如果实施运行时下标范围的检测，那么就可能需要将这些下标范围储存于一个运行时的描述符中。如果某一数组类型的下标范围为静态的，那么就可以将这种范围与进行检测的代码结合起来，从而消除对运行时描述符的需要。如果描述符中有任何一项被动态地绑定，那么在运行时就必须维护描述符中的这些项。

数组
元素类型
下标类型
下标下限
下标上限
地址



图6-5 一维数组的编译时描述符

多维数组的实现比一维数组的实现更复杂，虽然到多维数的扩展相对简单。硬件存储器是线性的，即它通常为字节的简单序列。因而具有二维或多维数据类型的值就必须被映射到一维的存储器上。有两种常用的方法可以将多维数组映射到一维：即按行存放和按列存放。在按行存放（row major order）的方法中，第一个下标为下标下限值的数组元素被首先储存，接着储存的是第一个下标为第二个值的元素，依此类推。如果这个数组是一个矩阵，则可以按行存储。例如，如果某个矩阵具有数值

3 4 7

6 2 5

1 3 8

按照按行存放的方法可以将它储存为

3，4，7，6，2，5，1，3，8

在按列存放的方法中，最后一个下标为下标下限值的数组元素被首先储存，紧接着储存的是最后一个下标为第二个数值的元素，依此类推。如果这个数组是一个矩阵，则可以按列存储。

274 如果上述的矩阵例子是以按列存放的方法存储，在存储器中它将具有下面的次序：

3，6，1，4，2，3，7，5，8

在Fortran中使用按列存放方法，其他的语言使用的是按行存放方法。

了解多维数组的储存次序，有时候十分关键，例如在C程序中，当使用指针来处理数组的情形。在所有情况下，如果对矩阵元素的存取是以这些元素的储存顺序来进行，则存取速度将会比较快速，因为这将会产生更多的内存局部性。[⊖]

我们将讨论使用在真正的具有多维数组的语言的存取函数。[⊖]一个多维数组的存取函数将数组的基址和一组索引值映射到由这些索引值指定的元素在存储器中的地址。一个以按行存放方法存储的二维数组的存取函数可以由下面的方法来构造。一般而言，一个元素的地址是结构的基址加元素的大小再乘以这个元素之前元素的数目。对于一个按行存放的矩阵，一个元素之前的元素数目是这个元素的行数乘以行的大小，再加上位于这个元素左边的元素数目。这些在

⊖ 更好的内存局部性意味着需要更少的缓冲来重新装填数据。

⊖ 某十C的语言支持多维的数组是数组的数组，而不是真正的多维数组。

图6-6中给予了说明。在这个例子中，我们使用了简化假设，即将下标下界都设为1。

为了取得实际的地址值，指定元素之前的元素数目必须乘以元素的大小。现在可以将存取函数写成

```
位置(a[i, j]) = a[1, 1] 的地址 +
                ((((第 i 行上方的行数) *(行大小))
                  + (第 j 列左面的元素数目)) *
                    元素大小)
```

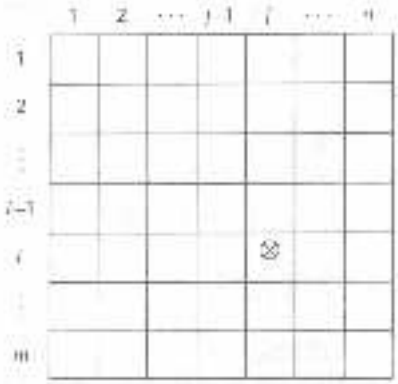


图6-6 矩阵中元素[i, j]的位置

因为在第i行上方的行数是(i－1)，而且在第j列左面的元素数目是(j－1)，因而我们有

```
位置(a[i, j]) = a[1, 1] 的地址 + (((i - 1) * n)
+ (j - 1)) *
                元素大小)
```

其中n是每行元素的数目。可以将这个公式重新组织为下面的形式：

```
位置(a[i, j]) = a[1, 1] 之地址 - ((n + 1) * 元素大小) +
                ((i * n + j) * 元素大小)
```

在这里，前面的两项为常量部分，最后一项则是变量部分。

对任意下限进行一般化处理，产生了下面的存取函数：

```
位置(a[i, j]) = a[行_下限, 列_下限] 之地址 +
                ((i - 行_下限) * n) + (j - 列_下限)) * 元素大小)
```

其中，“行_下限”为行的下界，“列_下限”为列的下限。又可以将它重新组织为下面的形式：

```
位置(a[i,j]) = a[行_下限, 列_下限] 的地址 -
               (((行_下限 * n) +列_下限) * 元素大小) +
               (((i * n) + j) * 元素大小))
```

其中，前面的两项为常量部分，最后一项是变量部分。可以相对容易地将这个公式一般化到任意的维数。

对于一个数组的每一维，它的存取函数需要有一个加法指令和一个乘法指令。因此，要访问一个具有几个下标的数组中元素，具有相当高的代价。图6-7显示了一个多维数组编译时的描述符。



图6-7 一个多维数组的编译时描述符

片将存储映射函数的复杂性再提高到了另一个层次。为了说明这一点，考虑一个具有一个矩阵和一个数组的程序，并将矩阵的一列赋给数组：

```
Integer, Dimension (10, 5) :: Mat
Integer, Dimension (10) :: List
...
List = Mat (1:3, 3)
```

假设映射是按行存放，并且元素的大小为1，矩阵Mat的存储映射函数为

```
位置(Mat [i, j]) = Mat [1, 1] 的地址 + ((i - 1) * 5 + (j - 1)) * 1
```

```
= (Mat [1, 1] 的地址 - 6) + ((5 * i) + j)
```

片引用Mat [1:3,3]的存储映射函数为

```
位置(Mat [i, 3]) = Mat [1, 1] 的地址 + ((i - 1) * 5) + (3 - 1)) * i
= (Mat [1, 1] 的地址 - 3) - (5 * i)
```

注意，这个映射与任何其他的一维数组存取函数具有完全相同的形式，只是由于其基本数组是二维的，而有不同形式的常量部分。

通过让i来取Mat中第一维下标范围(1:3)中的值，就能够找到从Mat中赋给List的元素。

6.6 关联数组

关联数组是一些数据元素的无序集合，通过使用同样数目的、被称为关键码（key）的值对这些数据元素进行索引。在非关联数组中，就不需要储存索引（因为数据元素所具有的规律性），
280 然而在相关数组中，必须将用户定义的关键码储存在这个结构中。因而相关数组中的每一个元素实际上就是由一个关键码以及一个值组成的一对实体。我们使用Perl语言中的相关数组设计来介绍这种数据结构。Python和Ruby以及Java、C++和C#的标准类库都直接支持相关数组。

相关数组所特有的唯一的设计问题是对数组中元素的引用形式。

6.6.1 结构与操作

在Perl语言中，常常将相关数组称为散列（hash），因为在实现中是通过散列函数来存储和检索相关数组中的元素的。Perl中不同散列的名字空间是不同的，每一个散列变量都必须以一个百分号（%）开始，可以使用赋值语句将散列设为字面常量值，如下面的例子：

```
%salaries = ("Gary" => 75000, "Perry" => 57000,
             "Mary" => 55750, "Cedric" => 47850);
```

Perl使用一种独特的标记方法来引用单个元素的值，将关键码放置于花括号之内，并使用标量变量的名字来替代散列名，而这个替代的名字除了第一个字符以外都是相同的。正如Perl数组的例子，尽管散列不是标量，但是散列元素的值部分是标量，所以对散列元素值的引用使用标量名，标量变量开始于$符号，例如，

```
$salaries{"Perry"} = 58850;
```

通过使用同样的句型可以添加一个新的元素。还可以使用delete操作符从散列中删除一个元素，例如，

```
delete $salaries{"Gary"};
```

通过将空的字面常量赋给散列，可以使得整个散列变为空，例如，

```
%salaries = ();
```

Perl中散列的大小是动态的：当增加一个新的元素时散列即会长大，当删除掉一个元素时散列会缩小，当赋以散列空的字面常量时可以将它清空。exists操作符将返回一个真值或一个假值，取决于其操作数关键码是否是散列中的一个元素，例如，

```
if (exists $salaries{"Shelly"}) . . .
```

281 当将操作符keys运用于一个散列时，它将返回散列的一个关键码数组。而操作符values将返回散列的值的数组，操作符each将遍历散列的每一个元素对。

PHP语言中的数组既是正常数组又是相关数组。可以将这些数组处理为任意的一种，PHP语言提供了对元素进行索引式存取和散列式存取的函数。该语言中的数组可以同时包含由简单数值索引所提供的元素，以及由字符串散列关键码所提供的元素。

如果要求进行元素搜寻，散列比数组优越得多，因为用来存取散列元素的隐式散列操作具有非常高的效率。而且，当数据是成对存储时，散列是完美的，如职工名字和他们的薪水。另一方面，如果必须处理一个表列中的每一个元素，则使用数组会有更高的效率。

访谈



开放源代码运动以及工作经历

RASMUS LERDORF

Rasmus Lerdorf在获取了工程学士学位以后，先后从事了几项咨询工作。后来，为了让除那些上网阅读他个人简历的人员，他创建了PHP语言的最初版本。现在，他是开放源代码运动的倡导者，同时也受雇于美国加州Sunnyvale的Yahoo公司。

问：关于开放源代码界与商业软件界，你有一些什么想法？考虑一下你为用于数据库与网页连接的PHP语言的解决办法，以及那些商业软件界的软件：ASP、Cold Fusion，等等。对于由开放源代码方式所产生的技术与商界所倡导技术，你从其中的一种方式里将会得到什么？从另一种方式里将不会得到什么？

答：无论你使用的是哪一种商业化的技术，你总会不得不担心，你将完全依存于这个特定的商务公司。你没有多少自主权来决定将哪一些特性放入语言的下一个版本。甚至有时也不能够确定还会不会有语言的一个版本。这些公司总是在试图将你完全绑定在他们的技术之上，剥夺你改换技术的灵活性，或者是与另一种技术相结合的容易程度。当然商业产品至少有一个明显的好处，就是你可以买到有保证的技术支持，这样，当你遇到问题时，有人在电话的那头等着你。

在开放源代码的世界里，至少就PHP语言，重点是放在集体之上。我个人并没有写出PHP语言，而是有几百人编写了它。我们形成了一个大集体，我们中的每一人都共同致力于同一个问题。我们创造了一种帮助我们解决这个问题的工具，并且我们都有对于这种工具的主人公意识。这就是为什么你在开放源代码世界中，常常会看到人们对于某种技术问题的热情投入，这种投入在商业软件界是非常罕见的。人们加入到这个集体中十分容易，并且在PHP语言开发方面给予了哪怕是一分微小的帮助，参与这些工作的人们将获得对于这种技术的较为深刻的感受，他们将满怀信心地认为，这种技术不会在他们这里消亡，将会帮助他们解决他们当前的问题，很有可能还会帮助他们解决将来的一些问题。通过这个集体，他们能够获得所需要的技术支持。

设想一种情景，你走进商店去购买一个盒装的软件，希望它的技术能够解决你的大部分问题，再设想另一种情景，你与一千多位有着同样问题的好朋友坐在一起，比较各自的工作笔记和各自的工具。前者的办法在某些时候可能是一种较好的解决办法，但是那个庞大的团体所围绕着开放源代码的项目不惜代价地建设自身，却是一种多么吸引人的方式！商务公司也试图围绕着他们的办法来建立一些这样的团体，有时候他们也非常成功；但终究，这里的气氛与一个健康的开放源代码的团体是完全不相同的，你简直不能够将这两者进行比较。当开放源代码团体所建设的工具成熟以后，这些工具就开始与任何已有的商业化工具相竞争，或者是超过它们。对于PHP语言，我们实际上已经超过了ASP以及Cold Fusion。在最初的阶段，我们并没有真正去追赶任何特定的商业化工具，我们仅仅是试图找到解决问题的办法。

问：你能否让我们分享你的一些想法，关于在1995年你独自为你的个人网页开发解决办法，以及你让新增的分散团队来接受你的贡献起步，进而将这些想法实现为被成百万的网站使用的工具。

答：当时，我是在从重复编写C语言的通用网关接口（CGI）的工作中解脱出来。我将一些通用的功能程序收集到一个C程序库中，并为这个库建立一个简单的语法分析器，以便我可以在我的HTML文件中

放上一些特别的标记，来启动我用C编写的各种子程序，这种方式在当时似乎是一种显然的解决办法，因而在早期我最主要的要求是开发的速度以及动态网页的使用，还有就是获取我所需要的一种简单框架：用C来编写业务逻辑，再从HTML文件中访问调用这些逻辑程序。

在任何人开始注意之前，我很可能已经在初始代码上耗费了大约18个月的时间。当时大约是1994年，这远早于任何围绕着开放源代码或免费软件开发的争议开始之前。我个人还是认为当时进入的门槛较坏在低。当时还没有多少针对网络问题的免费工具。但我还是必须将PHP开发到一定的程度以引起人们的关注，让人们觉得PHP可以解决他们很多的问题，而且很容易使用，也许只需要稍微扩展就能够解决他们的问题，他们不需要完全从零开始。这就是任何一个成功的开放源代码项目在其进程中的某一个时刻所必须跨越的门槛。使用其他人的解决办法，再解它，并将它实施于你自己的问题之上。这是你在时间与精力上的一项大投资，况且你还不能够肯定这种方法是否适用。在一个开放源代码项目被证明是适用之前，跨越这个门槛是十分艰巨的一步，如果你询问任何曾经成功开创过开放源代码项目的人士，你多半会发现，在他们生命中的这一个阶段，他们曾经为此完全贡献出了他们业余时间的每分每秒。这正是我在1994年到1995年间，为了PHP语言所经历的真实的生活状态。

问：你还记得你第一次接受到的来自其他人对于语言的扩展吗？当时的需求是什么？

答：我曾经接受到几次改错，但在早期第一次接受到的大的贡献是为连接Sybase和Oracle数据库所作的扩展。我曾经编写了连接数据库的代码，用以连接一个来自澳大利亚的名称为mSQL的免费数据库。我当时没有使用Sybase或Oracle数据库，但是其他地方的一些人却需要与这些数据库进行交互，因而他们编写了代码，并将这些代码贡献了出来。

问：能否与我们分享一下，当你最初公布PHP时，它是什么样子？以及它能够进行什么样的工作？

答：当时它只是一组联系松散的CGI程序，用于解决个人网页上的一般问题。它具有一个点击计数器，这个计数器同时也显示最后一个浏览网页的人的IP地址，以及机器在网络上的名称。它还能将点击记录到一个SQL数据库中，还包括产生有关网页访问报告的工具。因而早期的注意力是放在这些工具之上，而并没有放在编写这些工具的语言与框架之上。

问：开放源代码运动有些什么贡献？对比过去，今天的PHP语言又是什么样子？

答：开放源代码运动的集体对PHP语言贡献了所有一切。没有这个集体，就不会有PHP语言。PHP已经成长为一种一般用途的网络脚本语言，这种语言可以对你可以想像的任何对象进行交流，并且已经被用于30%的网络服务器上。它现在的支持范围从最小的家庭网页（可能每个月只有两次点击）到世界上最繁忙的网站。

在IT商务中的经历及工作岁月

问：你曾经有过各种职业，从大公司到网站，再到个人的咨询工作：你怎样决定下一步应该做什么？

答：我期望有意思且具有挑战的项目，也希望能够与相处愉快、融洽的人一道工作。但有时候也只好碰碰运气，我当年去巴西，最初仅是因为我在加拿大的Calgary工作时正是冬天，那里实在是太冷了。我刚好看了一些Mickey Rourke在Rio拍摄的电影，就开始想自己干嘛要待在这冰冻极地，而巴西肯定也有计算机，肯定也需要程序人员。

问：你是怎么决定在什么时候选择辞职？

答：当工作不再有意思时，我就会辞职。如果我每天早上醒来都很怕出去工作，或者我每天坐在那里都时刻在看钟，想着是否已经到了应该回家的时间，这时，我就知道我应该辞职了。总会有那么一段时间，你会不想去工作而想去别的地方；但是如果这种感觉是持续的，那么你的短暂的生命是不值得耗费在沮丧与无聊之中的。如果你甚至发现自己在计算每一秒钟挣多少钱，或者是计算在你去厕所来回时，你的公司付了你多少钱，那么就该辞职！

问：你在选择下一个任务时，有没有一些标准或者规则？

答：没有。我经常是跳来跳去，但我从来没有任何一些所谓预先的计划，要在一个公司待X个月或者一年。常常是工作的本身来决定我应该待多久。我从事某一些工作，纯粹是为了解决某一个特定的问题。在解决了这个问题之后，他们多半希望我继续待在那里，帮助他们维护和支持这种解决办法，但到那时，

挑战性的工作已经没有了，我又需要新的东西。当然，现在我有一个小baby在家里，这可能将将改变了我作选择的优先级别。也许，一个没有压力同时又具有挑战性的管理工作是一个好主意。我会在两年以后再告诉你。

276 ↓ 278

6.6.2 实现相关数组

为了快速查询，在Perl的相关数组实现中施行了优化。当由于数组的增长而带来需求时，它也提供了相对快的重新组织数组的功能。虽然相关数组在初始时只使用很小部分的散列值，对每一个散列项都计算了一个32字位的散列值，并将这个散列值与散列项储存在一起。但当一个相关数组必须被扩展，超过它的初始大小时，这并不需要改变散列函数，而只需使用更多的字位散列值。当发生这种情形时，仅需要挪动一半数目的散列项。因而，尽管相关数组的扩展是有代价的，但是它并不是你所想象的那么昂贵。

PHP中数组的元素被储存于一种链表形式中，其中的节点顺序就是产生这些节点的顺序。

通过current以及next函数，这个链支持数组元素的遍历访问。散列函数则被用于通过关键码存取数组中的所有元素。

6.7 记录类型

记录是一些不同类型的数据元素的聚集，其中的单个元素是通过名字来标识。

在程序中，常常需要模拟不同类型的数据集合。例如，一个关于大学生的信息可能包括姓名、学号、平均分数，等等。为这样的一种集合设计的数据类型可能会使用字符串来存放姓名，使用整数来存放学号，使用浮点数来存放平均分数，等等。记录就正是为了满足这一需求而设计的。

自20世纪60年代早期COBOL语言引入记录以来，记录类型已经成为所有最流行的程序设计语言中的组成部分，除了Pre-90以前的Fortran语言版本。

282

在C，C++以及C#语言中，是通过使用struct数据结构来支持记录。在C++中，结构只是类的少许变体。在C#中，结构也与类相似，但又具有显著的差别。C#中的结构正好与类中的对象相反，它们是一些在栈上分配的值类型，而类中的对象则是一些在堆上分配的引用类型。C++以及C#中的结构通常被用作封包结构，而不是被用作数据结构。关于这些结构功能的进一步讨论请见第11章。

Python和Ruby的记录能用散列来实现，散列可以是数组元素。

在下面的几节，我们将描述如何声明或定义记录，怎样引用记录中的域，以及一些常用的记录操作。

记录所特有的设计问题是：

- 域的引用的语法形式是什么？
- 在记录中允许省略引用吗？

6.7.1 记录的定义

记录与数组之间的根本不同在于：数组中元素具有同构性，而记录中元素可能具有异构性。这个区别的结果之一是，记录中的元素或域通常不是由下标来引用。反之，域是由标识符来命名，并使用这些标识符进行域的引用。记录与数组之间的一种更为重要的差别是，某些语言中的记录允许包括联合，我们将在第6.8节中讨论这个问题。

记录声明的COBOL形式是COBOL程序数据段中的部分，下面给出一个例子：

```
01  EMPLOYEE-RECORD.
    02  EMPLOYEE-NAME.
        05  FIRST    PICTURE IS X(20).
        05  MIDDLE   PICTURE IS X(10).
        05  LAST     PICTURE IS X(20).
    02  HOURLY-RATE PICTURE IS 99V99.
```

记录EMPLOYEE-RECORD包括记录EMPLOYEE-NAME和域HOURLY-RATE，每一行记录声明开头的数字01、02和05为层号，它们的相对值指示了记录的层次结构。如果一个行的后面跟随了一个较高层号的行，这个行的本身就是一个记录，PICTURE子句表示域的存储地址格式，X(20)表明20个字母数字字符，而99V99表明4个十进制的数字，小数点就位于中间。

Ada对于记录使用了不同的语法，它不使用COBOL语言的层号，它通过将一个记录声明嵌套于另一个记录声明，来说明记录的结构。Ada中的记录不能够是匿名的，它们必须为命名类型。考虑下面的Ada声明：

283

```
type Employee_Name_Type is record
   First : String (1..20);
   Middle : String (1..10);
   Last : String (1..20);
end record;
type Employee_Record_Type is record
   Employee_Name: Employee_Name_Type;
   Hourly_Rate: Float;
end record;
Employee_Record: Employee_Record_Type;
```

Fortran 95的记录声明要求将任何嵌套记录先定义为类型。那么在上面的雇员记录范例中，就需要先定义雇员名字记录，然后，雇员记录仅仅将它命名为雇员记录的第一个域的类型。

在Java中，可以将记录定义为数据类，而将嵌套的记录定义为嵌套的类。这种类中的数据成员则被用作记录中的域。

6.7.2 记录域引用

对记录中各个域的引用，在语法上可用几种不同的方法来说明，其中的两种方法要命名所需要的域和包含它的记录。COBOL中的域引用具有下面的形式：

```
域名 OF 记录名_1 OF ... OF 记录名_n
```

其间的第一个记录名是包含了这个域的最小或最里面的记录，在序列中的下一个记录名是包含了前面的记录的记录，并依此类推。例如，在前面COBOL记录例子中的MIDDLE域就可以采用下面的形式来引用：

```
MIDDLE OF EMPLOYEE-NAME OF EMPLOYEE-RECORD
```

大多数的其他语言采用点标记方式来进行域的引用，其中被引用的部分由点（句号）连接起来，而在点标记法中名字的排列顺序则正好与COBOL中的引用顺序相反，它们首先使用最大包含的记录名，最后才使用域名。例如，下面是对于上述Ada记录例子中的Middle域的引用：

```
Employee_Record.Employee_Name.Middle
```

C和C++使用同样的语法执行对其结构成员的引用。Fortran 95中的域引用也具有这种形式，只是它使用的是百分号（%）而不是点（句号）。

284

对一个记录域的完全限定引用，是指引用中必须出现所有的中间记录的名称，从最大包含的记录直到特定的域。上面例子中的COBOL和Ada的域引用都是完全限定的。作为对完全限定引用的一种替代，COBOL语言还允许对记录中域的省略引用。在省略引用中必须要有域名，但却可以省略任何部分或者全部的包含记录名，只要所产生的引用在引用环境中是非歧义的。例如，FIRST、FIRST OF EMPLOYEE-NAME以及FIRST OF EMPLOYEE-RECORD，在上面声明的COBOL的记录中是对于雇员名的省略引用。虽然省略引用给程序人员行来了方便，但是它们要求编译器具有精细的数据结构以及处理过程，以便正确地识别所引用的域。它们在某种程度上也会有损于可读性。

6.7.3 记录操作

赋值是一种常用的记录操作，在大部分情况下，等号两边的类型必须相同。Ada允许对记录进行等于和不等于两种比较。另外，Ada中的记录还可以使用字面常量的聚集来设置初值。

COBOL提供了MOVE CORRESPONDING语句来移动记录。这条语句将给定的源记录中的域复制到目的记录中，但是只有当这个目的记录具有一个相同名字的域时，才能够进行这种复制。在数据处理应用中，这常常是一种十分有用的操作，可以将其中的输入记录在经过一些修改之后移到输出文件中。因为输入记录常常具有许多的域，这些域与输出记录中的域同名以及同目的，但却不一定具有相同的次序，MOVE CORRESPONDING操作能够节省许多语句。例如，考虑下面的COBOL结构：

```
01  INPUT-RECORD.
    02  NAME.
        05  LAST          PICTURE IS X(20).
        05  MIDDLE        PICTURE IS X(15).
        05  FIRST         PICTURE IS X(20).
    02 EMPLOYEE-NUMBER    PICTURE IS 9(10).
    02 HOURS-WORKED       PICTURE IS 99.
01  OUTPUT-RECORD.
    02  NAME.
        05  FIRST         PICTURE IS X(20).
        05  MIDDLE        PICTURE IS X(15).
        05  LAST          PICTURE IS X(20).
    02  EMPLOYEE-NUMBER   PICTURE IS 9(10).
    02  GROSS-PAY         PICTURE IS 999V99.
    02  NET-PAY           PICTURE IS 999V99.
```

语句

```
MOVE CORRESPONDING INPUT-RECORD TO OUTPUT-RECORD.
```

将域FIRST、MIDDLE、LAST以及EMPLOYEE-NUMBER复制到输出记录中。

6.7.4 评估

在程序设计语言中，记录通常是十分有价值的数据类型。记录类型的设计是简单的，使用也很安全。记录在可读性方面唯一不清晰之处，是在COBOL语言中允许的省略引用。

记录与数组是密切相关的结构形式，因而将记录与数组进行一些比较是十分有意义的事情。如果所有的数据值都具有相同的类型，并且以相同的方式处理时，就采用数组。当整个结构是一系列的同构元素时，数据处理就很容易完成。动态下标寻址方法很好地支持了这种处理过程。

285

而如果这组数据值为不同类型，并且对不同域要使用不同的方式来处理时，就采用记录。另外，记录的域通常也不需要按某种特定的顺序来处理。域名类似于字面常量、常量或者下标，因为域名是静态的，所以它们对域提供了非常高效的存取，也可以使用动态下标来进行记录域的存取，但那样将不允许类型的检测，并且速度也会比较慢。

记录与数组为满足两种不同但又相关的数据结构的有关应用提供了周全及高效率的方法。

6.7.5 记录类型的实现

记录的域被储存于相邻的存储单位中。但是因为域的大小不一定相同，所以数组使用的存取方法就不适用于记录。相对于记录开始位置的偏移地址与每一个域相关联，因此域的存取完全是使用这些偏移地址来进行的。记录的编译时描述符具有图6-8中所示的一般形式。记录的运行时描述符则是不必要的。



图6-8 记录的编译时描述符

286

6.8 联合类型

联合是一种类型，在程序的执行期间，这种类型可能在不同的时刻储存不同类型的值。作为联合类型应用的一个例子，我们来考虑编译器的一个常量表。这个表被用来储存从正在编译的程序里发现的常量。每一个表项的域存放一个常量值。假设对于正在编译的某种语言，它的常量的类型为整型、浮点型和布尔型，就表的管理而言，如只在相同的位置上，即表中的一个域里，可以储存这三种类型中的任何一种类型的值，将会是很方便的。这样，可以使用同样的方法对所有的常量值寻址。在某种意义上，这样一种位置的类型是它所能够储存的三种值的类型的联合。

6.8.1 设计问题

在第5章曾经讨论过联合类型的类型检测问题，引出了一个重要的设计问题。另一个基本问题是怎样从语法上表示联合。在某些情形下，联合被限制为记录结构的组成部分，但是在另一些情形下，它们却不是。因而联合类型所特有的基本设计问题如下：

- 对联合类型应该要求类型检测吗？请注意，任何对联合类型的类型检测都必须是动态的。
- 应该将联合类型嵌入到记录中吗？

6.8.2 判别的联合与自由联合

Fortran、C和C++语言都提供了联合结构，但是都没有被语言支持的类型检测。在Fortran中，是使用Equivalence语句来说明联合；在C和C++中，则是使用union结构。因为允许

287 程序人员在使用联合时有不进行类型检测的自由，所以将这些语言中的联合称为自由联合。例如，考虑下面的C联合：

```
union flexType {
   int intEl;
   float floatEl;
union flexType el1;
```

```
float x ;
...
ell.intEl = 27;
x = ell.floatEl;
```

这最后的赋值不是类型检查的，因为系统不确定ell当前值的当前类型，所以它把27的位字符串表示赋给float变量x。当然，这是没有意义的。

对于联合的类型检测，要求每一个联合结构包括一个类型指示符，这种指示符被称为标志（tag）或者判别式（discriminant），一个具有判别式的联合就被称为判别的联合（discriminated union）。第一种提供了判别联合的语言是ALGOL 68。Ada语言现在也支持判别的联合。

6.8.3 Ada联合类型

Ada在前辈语言Pascal的基础上对判别的联合做出的设计是允许用户指明一个变体记录类型的变量，这种变体记录类型将仅仅储存变体中的一个可能值。用户可以通过这种方式来告诉系统，类型检测在什么时候可以为静态的。这种限制了的变量就被称为受限变体变量。

处理一个受限变体变量的标志十分类似于命名常量。在Ada语言中的非受限变体变量允许变体的值在执行期间改变类型。然而要改变变体的类型，就只能够通过赋予包括判别式的整个记录。这样就杜绝了记录的不一致性，因为如果新赋值记录是常量数据的聚集，那么就可以静态地检测标志值以及变体类型的一致性。[⊖]如果所赋的值是一个变量，它的一致性在赋值时就得到了保证，因而这个变量被赋的新值肯定是一致的。

下面的例子显示了一个Ada变体记录：

```
type Shape is (Circle, Triangle, Rectangle);
type Colors is (Red, Green, Blue);
type Figure (Form : Shape) is
  record
    Filled : Boolean;
    Color : Colors;
    case Form is
      when Circle =>
        Diameter : Float;
      when Triangle =>
        Left_Side : Integer;
        Right_Side : Integer;
        Angle : Float;
      when Rectangle =>
        Side_1 : Integer;
        Side_2 : Integer;
    end case;
  end record;
```

288

图6-9显示了这个变体记录的结构。下面的两条语句声明类型Figure的变量：

```
Figure_1 : Figure;
Figure_2 : Figure(Form => Triangle);
```

Figure_1被声明为不具有初始值的非受限变体变量。它的类型改变可以通过赋以包括判别式的整个记录而得以实现，如在下面所示的：

⊖ 此处是一致性是指：如果标签显示当前的联合类型是整型，那么联合体的类型就为整型。

```
Figure_1 := (Filled => True,
             Color => Blue,
             Form => Rectangle,
             Side_1 => 12,
             Side_2 => 3);
```

289

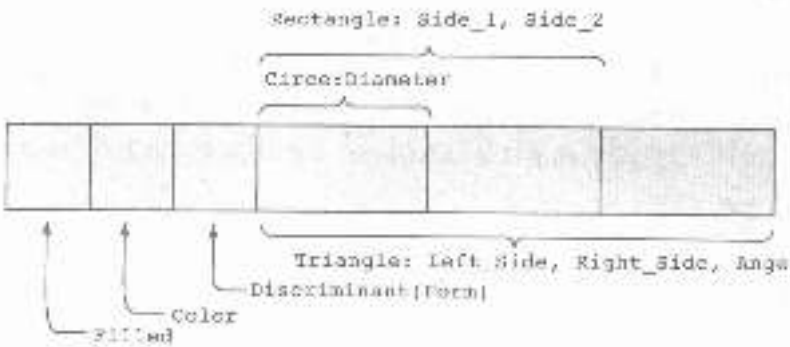


图6-9 三种形状变量的判别式联合（假设所有变量都是同样大小的）

这一条赋值语句的右边是一个数据的聚集。

向变量Figure_2的声明则将其限制成为一个三角形，并且还不能被改变为另外的变体。

这种判别联合的形式是十分安全的，因为它总是允许类型检测；虽然必须动态地检测受限变体中域的引用。例如，假设有下面的语句

```
if(Figure_1.Diameter > 3.0) ...
```

运行时系统将需要检测Figure_1以确定其Form标签是否为Circle。如果不是，引用Diameter将产生一个类型错误。

290

6.8.4 评估

在许多语言中，联合是潜在的非安全结构。这就是Fortran、C以及C++不是强类型语言的原因之一；因为这些语言不允许对联合的引用进行类型检测。但在另一方面，联合也可以被安全地应用，就如同在Ada中那样。在绝大多数的语言中，在使用联合时务必十分小心。

Java和C#都没有包括联合，这可能反映出对于程序设计语言中的安全问题所增加的关注。

6.8.5 联合类型的实现

实现判别的联合可以仅通过对每一个可能的变体都使用相同的地址。给最大变体分配以充足的存储空间。至于Ada语言中的受限变体的情形，因为不存在大小的变化，因而可以使用准确大小的存储空间。判别的联合的标志与变体一起被储存在一种与记录相类似的结构中。

在编译时，必须将每一个变体的完整描述储存起来。完成这项任务的方式是为描述符中的标志项建立一个case表格，在这个表格中，每一种变体都有一个项，这个项指向特定变体的描述符。为了说明这种安排，请考虑下面的Ada例子：

```
type Node (Tag : Boolean) is
 record
 case Tag is
     when True => Count : Integer;
     when False => Sum : Float;
 end case;
end record;
```

这种类型的描述符可以具有图6-10中所示的形式。

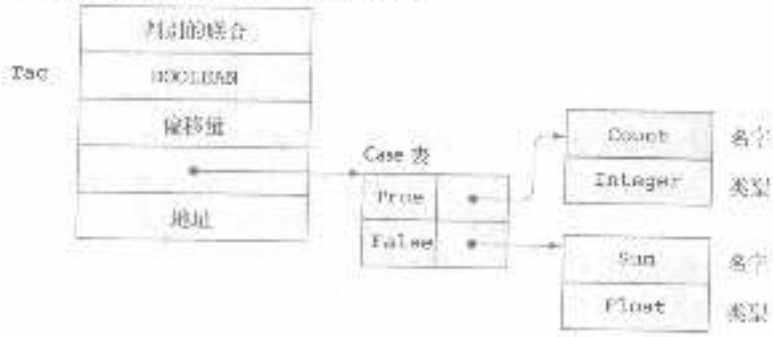


图6-10 判别的联合的编译时描述符

6.9 指针类型与引用类型

指针类型是这样一种类型：这种类型的变量的值范围包括了存储器地址以及特殊值nil（空值）。nil并不是一个有效地址，仅使用它来表示当前的指针不能够指向任何存储单位。

指针是为两种不同的用途而设计的。首先，指针提供某种间接寻址的功能。这种功能被频繁地使用于汇编程序设计语言中。其次，指针提供了管理动态存储空间的一种方法。指针可以被用来访问动态分配的存储区域中的某一个位置。这种动态分配的存储区域通常被称为堆（heap）。

其存储空间是从堆中动态分配的变量被称为堆动态变量（heap dynamic variable）。通常没有与它们相关联的标识符。因而这种变量的引用只能通过指针或引用类型的变量来进行。没有名字的变量被称为匿名变量（anonymous variable）。正是在后面的这种指针的应用领域里产生了最重要的设计问题。

指针不像数组与记录，它们不是结构化的类型。虽然在指针的定义中使用了一个类型操作符（如，在C和C++中的*，以及Ada中的access），此外，指针也不同于标量变量，因为指针经常被用来引用其他的一些变量，而不是用来储存某类数据。变量的这两个种类被分别称为引用类型以及值类型。

指针的这两类应用都为语言增加了可写性。例如，假设我们必须在一种类似于Fortran 77的没有指针的语言中实现一种二叉树之类的动态结构，这就要求程序人员提供并且维护一组树节点变量，人们多半会将这样一组变量实现为平行数组。另外，因为在Fortran 77中缺乏动态的存储空间，程序人员还必须猜测所需节点的最大数目。显然，这是一种极为别扭与麻烦的处理二叉树的方法。

将要在第6.9.7节讨论的引用变量与指针紧密相关。

6.9.1 设计问题

指针所特有的，主要的设计问题如下：

- 什么是指针变量的作用域与生存期？
- 什么是堆动态变量的生存期？
- 是将指针限制在它们所能够指向的值的类型之上吗？
- 是将指针应用于动态存储空间的管理，还是应用于间接寻址？或者用于这两者？

291

- 程序语言是应该支持指针类型，还是应该支持引用类型？或者应该支持这两者？

6.9.2 指针操作

提供指针类型的语言通常包括了这些两种指针的基本操作：即赋值与间接引用。第一种操作是将指针变量的值设置于某个有用的地址。如果只是使用指针变量来管理动态存储空间，则无论通过操作符还是通过内置于程序所构成的空间分配机制都能够设置指针变量的初始值。如果是将指针用于非堆动态变量的间接寻址，那么就必须通过一个显式操作符或者一个内置于程序来获取一个变量地址，然后再将这个地址赋给指针变量。

出现在表达式中的指针变量可以用两种不同的方式来解释。首先，可以将它解释为，它是变量所绑定的存储单位中的内容的引用，而这种内容仅仅是一个地址。这恰恰是在表达式中的非指针变量所应该获得的解释，只是在那种情形下，存储单位的内容不是一个地址。其次，也可以将一个指针变量解释为，它是一个存储单位中的值的引用，而这个存储单位的地址储存于变量所绑定的单位中。在这种情形下，将指针解释成为一种间接引用。前面的那种情形是通常的指针引用；后面的情形则是指针间接引用（dereferencing）的结果。间接引用经过一个间接的层次来进行引用，它是指针的第二种基本操作。

指针的间接引用可以是显式的或者隐式的。在Fortran 95中它是隐式的，但是在一些当代语言中，它只是出现在显式说明时。在C++中，使用星号（*）作为前缀一元操作符来显式地说明间接引用。考虑下面间接引用的例子：如果ptr是一个具有值为7080的指针变量，其地址为3080的存储单位，具有值206。那么赋值语句

```
j = *ptr
```

将j的值设为206。这个过程被显示在图6-11中。

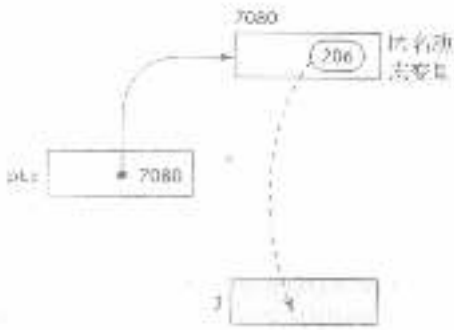


图6-11 赋值操作j = *ptr

当指针指向记录时，用于引用这些记录的域的语法根据语言的不同而不同。在C和C++中，可以使用两种指针到记录的方式来引用这个记录中的域。如果一个指针变量p指向一个记录，该记录具有一个命名为age的域，就可以使用(*p).age来引用这个域。当操作符 -> 被用于所指向的记录的指针和那条记录的域之间时，它就结合了间接引用以及域的引用。例如，表达式 p -> age与(*p).age是等价的。在Ada中可以使用p.age，这为指针的这种用法是隐式间接引用。

为本的管理而提供指针的语言必须包括一种显式分配操作。有时是通过一个子程序来进行分配的说明，如C中的malloc。在支持面向对象程序设计的语言中，常常是使用new操作符来说明堆对象的分配。C++语言不提供显式解除分配，它使用delete作为解除分配操作符。

6.9.3 指针的问题

第一种包括了指针变量的高级程序设计语言是PL/I。在PL/I语言中，指针可以被用来引用堆动态变量以及其他程序变量。PL/I中的指针具有高度灵活性，但是这些指针的使用可能导致几种程序错误。PL/I中指针存在的问题也出现在一些后继语言的指针中。一些最新的语言，如Java，已经使用引用类型完全取代了指针，连同隐式解除分配一起，从而消除了指针所带来的

主要问题，引用类型只是带有限制操作的指针，第6.9.7节将讨论引用类型。

6.9.3.1 悬挂指针

悬挂指针，或悬挂引用，是一个包含了已解除分配的堆动态变量地址的指针。我们可以给出几个理由来说明悬挂指针是危险的。首先，悬挂指针指向的位置可能已经被重新分配给一个新的堆动态变量，如果这个新变量与老变量不是同一种类型，对悬挂指针使用的类型检测将会是无效的。即使这个新动态变量与老动态变量具有相同的类型，它的新值将与悬指针所间接引用的值没有关系。此外，如果是使用悬挂指针来改变堆动态变量，新堆动态变量的值就会被破坏。最后，这个位置很有可能当前暂时被存储管理系统所使用，可能是作为一个可用存储块链上的一个指针，因而允许这个位置的值的改变，这将导致存储管理的失败。 293

下面的操作序列将在许多语言中产生一个悬挂指针：

1. 设指针p1指向一个新的堆动态变量。
2. 给指针p2赋以p1的值。
3. 将p1所指向的堆动态变量显式地回收（设p1指向空null），但是这种操作并不改变指针p2。p2现在成为了一个悬挂指针。如果回收操作不改变p1，p1和p2都将是悬挂的。

例如在C++中，我们可能有

```
int * arrayPtr1;
int * arrayPtr2 = new int[100];
arrayPtr1 = arrayPtr2;
delete [] arrayPtr2;
// 其中arrayPtr1是悬挂的，因为arrayPtr1所指向的堆现在存储的已经被回收了。
```

6.9.3.2 丢失的堆动态变量

丢失的堆动态变量是一个已经分配的堆动态变量，用户程序不可以再对它进行访问。常常称这种变量为垃圾，因为对于它们的原始目的，它们不再有用，并且它们也不可以为了某种新的用途通过程序给以重新分配。最常见的丢失的堆动态变量是由下面的操作序列所产生的：

1. 设置指针p1指向一个新创建的堆动态变量。
2. 在后来，又设置p1指向另一个新创建的堆动态变量。 294

现在，第一个堆动态变量就是不可以访问的，或者称它为丢失的。有时候，将它称为存储器泄漏，存储器泄漏是一个问题，无论语言使用的是隐式还是显式的解除分配方法。

在下面的几个小节中，我们将研究语言的设计人员是怎样处理悬挂指针以及丢失的堆动态变量的问题。

6.9.4 Ada语言中的指针

Ada中的指针常常被称为access类型，通过Ada语言的设计，悬挂指针问题得以缓解。作为一种实现方式，堆动态变量可以在它的指针类型作用域的终端被隐式地解除分配，从而大量地减少对显式解除分配的需要。因为堆动态变量只可以通过一种类型的变量来存取，当到达了这种类型声明的作用域的终端时，指针就不可以继续指向动态变量。这样就减少了悬挂指针的问题，因为不恰当地实现显式解除分配，是产生悬挂指针的主要原因。然而不幸的是，Ada语言还是包括了一个显式解除分配符号：Unchecked_Deallocation。这个名字的本身就意味着并不鼓励它的使用，或者，至少是警告用户它可能具有的潜在问题。

丢失的堆动态变量的问题并没有通过Ada的指针设计而得以消除。

6.9.5 C和C++中的指针

C和C++中指针的用法很像汇编语言中地址的使用。这意味着它们具有极大的灵活性，但而在使用时必须极其小心。这些语言的设计没有针对悬挂指针或丢失的堆动态变量问题提供解决办法。然而在C和C++中可以进行指针算术，这使得它们的指针比其他程序设计语言中的指针更为有趣。

历史注释

Pascal包括了一种显式撤销分配的操作符dispose。由于dispose会引起悬挂指针问题，当程序中出现了dispose时，一些Pascal的实现者十分明智地选择，虽然这种方式有效地防止了悬挂指针的问题，但在同时，它却不允许重复使用那些程序已经不再需要的堆存储空间。换言之说，Pascal语言是作为一种教学语言来设计的，而不是作为一种工业化的工具。

不像Ada中的指针只能指向堆，在C和C++中的指针几乎可以指向任何变量，无论这些变量被分配了什么位置。事实上，它们可以指向存储空间的任何位置。而不论在这个位置上有没有变量的存在。这也是这些指针的危险性所在。

在C和C++中，*表示间接引用操作，而&表示产生变量地址的操作符。例如，在下面的代码中

```
int *ptr;
int count, init;
...
ptr = &init;
count = *ptr;
```

295

对变量ptr的赋值，将ptr设为init的地址。第二条对count的赋值语句间接引用ptr以产生init处的值，然后又将这个值赋给count，因而前面两条赋值语句的效应是将init的值赋给count。请注意，一个指针的声明定义了它的域类型。

注意，这里的两条赋值语句等价于下面的这一条作用在count上的赋值语句：

```
count = init;
```

可以给指针赋以任何正确的域类型变量的地址值，或者，可以给它们赋以用来表示空值nil的常量零。

也可以在一些受限形式中进行指针算术。例如，如果ptr是一个指针变量，它被声明为指向某个数据类型的某个对象，那么

```
ptr + index
```

是一个合法的表达式。这种表达式的语义如下面所描述。这里，并不是简单地将index的值加到ptr之上，相反，它首先将index的值根据ptr所指向的存储单位的大小（以存储单位计）按比例放大。例如，如果ptr指向的是放置一个具有四个存储单位大小的类型的存储单位，那么就将index乘以4，并且将这个乘法的结果加到ptr上。这种地址算术的主要目的是数组的管理。下面，我们仅仅讨论一维数组的情形。

在C和C++中，所有数组都使用零作为下标范围的下限，而且没有下标的数组名表示的总是这个数组中第一个元素的地址。事实上，可以完全像处理指针那样来处理没有下标的数组名，除了它本身是一个常量，并因此不能够对它赋值以外。考虑下面的声明：

```
int list [10];
int *ptr;
```

考虑赋值语句

```
ptr = list;
```

这条语句可以理解为将list [0]的地址赋给ptr。因为没有下标的数组名解释为这个数组的基址。对于这一条赋值语句，我们能得出下面的结论：

*(ptr + 1) 与 list [1] 等价。

*(ptr - index) 与 list [index] 等价。 296

*ptr [index] 与 list [index] 等价。

这些语句清楚地表明，指针操作包括了与下标操作相同的按比例缩放。此外，可以将指向数组的指针像数组名一样进行索引。

在C和C++中的指针可以指向函数。这种特征被用来将函数作为参数传递给其他的函数。第9章中将要讨论到，指针也被用于参数传递。

C和C++包括了void *类型的指针，这意味着它们可以指向任何类型的值。它们实际上是一些通用指针。然而，因为不可以间接引用void *指针，所以它们不存在类型检测的问题。void *指针的一种常见应用是作为运用于存储空间的函数的参数。例如，假设我们想要用一个函数将一个数据字节的序列从存储空间里的一处移到另一处。如果这个函数可以传递两个任意类型的指针，则将是最一般的。如果这个函数中相应的形参为void *类型，就会是合法的。然后这个函数可以将形参转换为char *类型并完成操作，而不论传递的是什么类型的指针作为实参。

6.9.6 引用类型

引用类型变量类似于指针，除了一个重要而且基本的不同之处：指针指示内存中的地址，而引用指示内存中的对象或值。因此，尽管在地址上做算术运算是很自然而然的，但对引用做算术运算则是不合适的。

C++语言包括了一种特殊的指针类型，这种类型主要用于函数定义中的形参。C++的引用类型变量是一个总是被隐式间接引用的、具有固定值的指针。因为C++引用类型变量是一个常量，所以在它的定义中它必须以某个变量的地址来做初始化，并且在初始化后，引用类型变量不能再被设置来引用任何其他变量，隐式间接引用阻止了对引用变量的地址值的赋值。

引用类型变量通过在定义中将&符号放置于它们的名字之前来说明。例如：

```
int result = 0;
int &ref_result = result;
...
ref_result = 100;
```

在这一段代码中，result和ref_result互为别名。

当引用类型在函数定义中被用作形参时，它在调用函数与被调用函数之间提供了双向交流。因为C++的参数是按值传递的，所以对非指针的参数类型这是不可能的。将指针作为参数传递也实现了双向交流，但由于指针的形参要求显式间接引用，这就导致了较差的代码可读性和安全性。在被调用函数中执行引用参数的引用，完全与引用其他的参数相同。当一个实参所对应的形参是引用类型时，调用函数不需要说明这个实参与众有什么不同，编译器会传递地址而非值给引用参数。 297

Java中的引用变量是C++中引用变量形式的扩展，它实用引用变量完全地替代了指针。为了追求比C++更好的安全性，Java的设计人员完全去掉了C和C++式的指针。不像C++引用变量，为了能引用不同的类实例，可以给Java中的引用变量赋值；也就是说，它们不是常数。所有Java中的类实例都通过引用变量来引用。这事实上也是Java中引用变量的唯一用途。关于这些

问题还将在第12章中进一步讨论。

在下面的代码中，String是一个标准的Java类：

```
String str1;
. . .
str1 = "This is a Java literal string";
```

这段代码将str1定义为对String类的实例或对象的一个引用，但将其初始设为空。后面的赋值语句设str1引用String类中的对象："This is a Java literal string"。

Java的类实例是隐式地被解除分配的（没有显式解除分配的操作符），因而不会有悬挂引用。

C#既包括了Java中的引用，又具有C++中的指针。但是C#极不鼓励使用指针。事实上，任何使用指针的方法都必须包括unsafe修饰符。请注意，尽管被引用指向的对象是被隐式解除分配的，但被指针指向的对象却不是这样。之所以在C#中包括进指针，主要是为了能够允许C#程序与C以及C++的程序相兼容。

在纯面向对象语言Smalltalk、Python和Ruby中的所有变量都是引用，它们总是隐式地间接引用，而且，这些变量的直接值不能够访问到。

6.9.7 评估

关于悬挂指针以及垃圾的问题，已经进行了较多的讨论。堆管理的问题将在第6.9.9.3节进行讨论。

人们曾经将指针与goto语句进行比较。goto语句扩展了下一条可以执行的语句的范围，
298 而指针变量则拓广了一个变量所能够引用的存储单位的范围。大概对于指针最强烈的谴责莫过于Hoare（1973）所声称的："将指针引入高级语言中，是我们已经无法复原的一大退步。"

另一方面，在一些程序设计应用程序中，指针是很基本的，例如，在编写设备驱动时，指针是必需的，因为在那里会访问到具体的、绝对的地址。

Java以及C#中的引用提供了指针的某种程度的灵活性及能力，但不具有指针的危险性。程序人员是否愿意牺牲C和C++中指针的完全功能来换取引用的较强安全性能，还有待观察。C#程序在指针方面的扩展就是对这个问题的一个衡量。

6.9.8 指针类型和引用类型的实现

在大多数语言中，指针被用于堆管理。Java和C#中的引用以及Smalltalk、Python和Ruby中的变量也同样应用于堆管理，因而我们应该等同地对待指针与引用。下面，我们首先简略地描述指针和引用是怎样表示的，然后，再讨论两种可能的解决悬挂指针问题的办法，最后，我们将描述堆管理技术中的一些主要问题。

6.9.8.1 指针与引用的表示

在大多数较大型计算机中，指针和引用是储存于存储单位中的单个值。然而，大多数的微型计算机是基于英特尔微处理器的。它们中的地址分为两个部分：即段和偏移量。因此在这些系统中将指针和引用实现为一对16个字位的单位，一个单位对应于段，另一个单位对应于偏移量。

6.9.8.2 解决悬挂指针问题的办法

针对悬挂指针的问题已经有了几种提议的解决方法。其中有墓碑（tombstone）（Lomet，1975）方法。在墓碑方法中，每一个堆动态变量都包括了一个被称为墓碑的特殊单位，墓碑自

身就是指向堆动态变量的一个指针。实际的指针变量只是指向墓碑，而不是指向堆动态变量。当将一个堆动态变量解除分配时，墓碑则还被保持着，只是被设为空（null），这就指示堆动态变量已经不存在。这种方式防止了指针再指向一个被解除分配了的变量。对于指向空值的墓碑指针的任何引用都被检测为错误。

无论就时间方面还是空间方面而言，墓碑方法的代价都是十分高的。因为从来不会将墓碑解除分配，因而它们所占据的存储空间就不会被重新使用。每一次经过墓碑来对堆动态变量进行存取，都要求有更多的间接层次，而这在大多数的计算机上就需要额外的机器周期。显然，流行语言的设计人员并不认为所增加的安全性能够值得付出这些额外的代价。因为没有一种广泛应用的语言使用了墓碑。

墓碑的一种替代方法是用于UW-Pascal（Fischer and LeBlanc，1977，1980）实现中的锁—钥匙方法（locks-and-keys approach）。在这种编译器中，指针值被表示为一些顺序的对（钥匙，地址），在这里钥匙是一个整数值。堆动态变量被表示为变量的存储空间再加上一个存储整数锁值的头单位。当堆动态变量被分配存储空间时，锁值被创建并被放置在堆动态变量的锁单位中，同时也被放置在对new的调用中所指明的指针的钥匙单位中。对于间接引用指针的每一次存取，都将进行指针的钥匙值与堆动态变量中的锁值之间的比较，如果它们相匹配，这种存取就是合法的；不然，就会将这种存取处理为运行时错误。任何指针值对其他指针的拷贝都必须复制这种钥匙值。因此，任意数目的指针都可以引用同一个堆动态变量。当使用dispose将一个堆动态变量解除分配时，就将这个变量的锁值清理成为一个不合法的锁值。此时，如果间接引用一个在dispose说明以外的指针时，虽然指针的地址值并没有被改变，但是它的钥匙值将不再与这个锁值相匹配，因此对于它的存取将不被允许。

当然，解决悬挂指针问题的最好办法是取消程序人员将堆动态变量解除分配的权力。如果程序不能够显式地将堆动态变量解除分配，就不会有悬挂指针。要达到这一目的，运行时系统就必须将它们不再需要的堆动态变量隐式地解除分配。LISP的系统就是这样实施的。Java以及C#也对它们的引用变量施行这样的方式。前面讲过，C#中的指针是不具有隐式解除分配的。

6.9.8.3 堆管理

堆管理可以是一种非常复杂的运行时过程。我们分别从两种情形来研究这个过程：其一，所有的堆存储空间都以同样大小的单位进行分配与解除分配；其二，以大小可变的段进行分配与解除分配。请注意，我们仅仅讨论隐式的解除分配方式。我们的讨论将会比较简略，不可能包罗万象，因为透彻地分析这些过程以及与这些过程相关的问题，更像是一个实现问题，而不是语言设计问题。

单一大小的单位 最简单的情形是所有被分配以及解除分配的单位都是同样大小的。如果每个单位都已经包含了一个指针，这时的情形就被进一步地简化了。这正是许多LISP实现中的情形。在那里，动态存储空间分配的问题第一次以大规模出现。所有LISP的程序以及大部分LISP中的数据，都由连接为链表的单位构成。

在按同一大小的单位进行分配的堆中，通过单位中的指针将所有可以使用的单位连接起来，从而形成一个可用空间的链表。当需要时，存储空间的分配就仅从链表上取得所需数目的单位。但解除分配是一个远为复杂的过程。因为多个指针可以同时指向一个堆动态变量，这使得难以确定程序是否已经不再需要这个变量。仅仅因为有一个指针已经不再指向这个单位，显然不足以使这个单位成为垃圾；其他的几个指针可能仍旧指向这个单位。

在LISP中，程序中的几种最常见操作会产生一些程序不再能够存取因而应该被解除分配（放回可用空间链表的后部）的存储单位。LISP的基本设计目标之一就是要确保不由程序人员

回收那些不再使用的单位。这个任务要由运行时系统来执行。这就给 LISP 的实现人员留下了一个基本设计问题：应该在什么时候进行解除分配的工作？

有一些不同的方法用于垃圾收集。两种最常用的传统技术在某种意义上是反向的过程。它们是**引用计数器法**和**标记清除法**。在引用计数器方法中，回收是递增的，只要出现了不能被访问的单位就回收；而在标记清除方法中，回收仅仅发生于可用空间的链表为空时。有时，我们将这两种方法分别称为**积极方法**（eager approach）和**懒惰方法**（lazy approach）。这两种方法衍生出了许多变化。然而，在本书中，我们只讨论基本过程。

存储空间回收的引用计数器方法是通过在每一个单位中保持一个计数器来达到它的目标，在这个计数器中储存了当前指向这个单位的指针数目。嵌入式的引用计数器的赋值操作包括了零值的检测。当每次有一个指针与这个单位分离时，计数器的值就减1。如果引用计数器达到了零值，这意味着已经没有一个程序指针还在指向这个单位，这个单位因此成为了垃圾，可以被回收到可用空间的链表之上。

引用计数器的方法存在三个问题。第一，如果存储单位相对较小，计数器所需要的空间就会占据很大比例。第二，为了维持计数器的值，显然需要一些执行时间。每一次改变指针的值，这个指针曾经指向的单位就必须将它的计数器递减，而指针当前指向的单位就必须将它的计数器递增。在一种类似LISP的语言中，其中的每一个动作几乎都包括了指针的变更，这可以耗费掉总执行时间的很大部分。当然，如果指针的改变不是太频繁，这显然不成为一个问题。通过一种称为**延迟引用计数**的方法可以清除一些引用计数器的无效率，延迟引用计数能为一些指针避免引用计数器。第二，当一组单位被连接成环状时，情况就趋于复杂化。这里的问题是，在环状链表中每一个单位的引用计数器值至少为1，从而阻止了这个单位的收集，不能将它放回可用空间的链表之上。关于这个问题的一种解决办法，在Friedman和Wise (1979)的文章中可以读到。

引用计数器方法的优点是它在本质上是递增的，它的动作随着应用程序插入，因此，在应用程序执行中，它不会引起较大的延时。

最初的垃圾收集的标记清除过程执行如下：运行时系统按照要求分配存储单位，并且在需要时将指针从这些单位上脱离下来，且并不牵涉到存储空间的回收（允许垃圾的累积）。直到系统已经将所有可用单位分配完毕。此时，标记清除过程开始收集堆中的所有垃圾。为了帮助这种标记清除过程，每一个堆单位都具有一个额外的指示器字位或域，以供收集算法使用。

这种标记清除过程包括了二个不同的阶段。首先，将堆中所有单位的指示器都设置成表示这些单位为垃圾。当然，这种假设只对一部分单位是正确的。收集过程的第二个阶段（标记阶段）是最困难的，这时程序中的每一个指针将追踪到堆中，并且将所有可以达到的单位都标记成非垃圾。之后执行第三个阶段（清除阶段），将堆中所有没有特别标记为正在使用的单位返回给可用空间的链表。

为了说明用于标记正使用单位的算法风格，我们提供下面这种标记算法的简单版本。假设所有的动态变量或者堆单位都包括了一个信息部分。一个被命名为tag的标记部分，以及两个被分别命名为llink和rlink的指针。这些单位被用来建立一种有向图，图中最多只具有来自任一节点的两条边。这种标记算法遍历图中所有的生成树，标记出所有被找到的单位。像其他遍历一样，标记算法使用递归。

```
for every pointer r do
    mark(r)

void mark(void * ptr) {
```

```
    if (ptr != 0)
      if (*ptr.marker is not marked) {
        set *ptr.marker
        mark(*ptr.llink)
        mark(*ptr.rlink)
      }
}
```

图6-12显示了将这个过程作用于一个给定图的例子。这种简单标记算法要求大量的存储空间（用栈空间来支持递归）。Schorr和Waite开发了一种不需要额外栈空间的标记程序（Schorr and Waite，1967）。他们的方法是在追踪出链接结构的时候，将指针反转，然后在到达链表的尾端时，处理过程可以跟随指针在结构中反向进行。

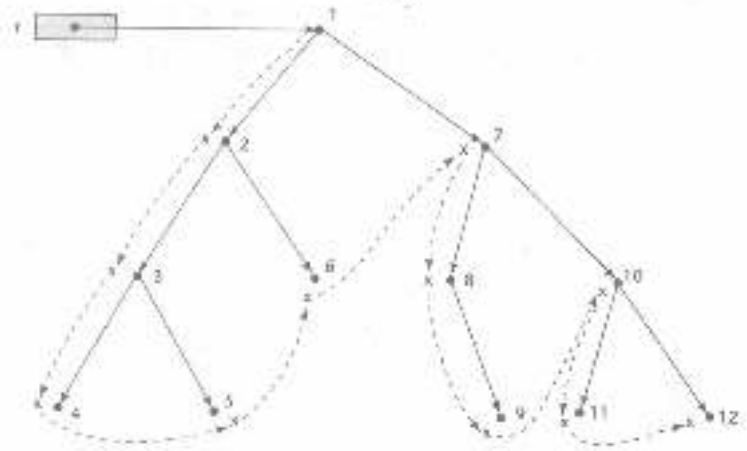


图6-12 标记算法行为的一个例子

使用标记清除的最初版的最大问题是，它工作得太不频繁了——只有在程序用完或即将用完所有的堆存储空间时工作。在这种情形下，标记清除会花费大量的时间，因为大部分单元必须在使用时跟踪和标记。这会在应用程序运行时产生一个显著的延时。而且，这个过程可能只会生成很少的放置在可用空间列表的单元，这个问题慢慢得到了改善。例如，在回收存储器的数量上，在内存耗尽之前，增量标记清除垃圾收集能更频繁、更有效地工作。

这个过程每一轮的时间明显减少了，因此减少了应用程序执行时的延时，另一种可供替代 302
的方法是在不同时间部分执行标记清除过程，而不是全部内存。这样，它能取得和增量标记清除法同样的效果。

标记清除方法的标记算法以及引用计数器方法所需要的过程，都可以通过使用Suzuki描述的指针旋转和滑行操作而更具有效率（Suzuki，1982）。

可变大小单位　管理一个可以分配可变大小单位的堆时，不仅具有单一大小单位分配的一切困难，还具有一些额外的问题。但大多数程序设计语言都会需要可变大小的单位，取决于所用方法的不同，管理可变大小单位中的额外问题也就不一样。如果使用的是垃圾收集方法，就会出现下述的额外问题：

- 将堆中所有单位的指示器设定初值以指示这些单位为垃圾，这是很困难的过程。因为这些单位都具有不同的大小，对它们进行扫描就是一个问题。对于这个问题的一种解决办法是要求将每一个单位中的第一个域用来表示单位大小，接下来就能够完成扫描，尽管 303
这种做法与单一大小单位的同类型操作相比会需要稍大的空间和稍长的时间。

- 这种标记过程不是简单的。如果没有预先定义的位置来存放指向单位的指针，怎么能够从指针进行链的跟踪？完全不包含指针的单位也是一个问题。将系统指针附于每一个单位是一种解决办法，但是必须将这些指针与用户定义的指针保持平行，这会在程序运行的代价上再加上空间和执行时间的额外开销。
- 维护可用空间的链表是另一种额外开销。在开始时，这种链表只有一个单位，它包括了所有可用的空间。对空间的不断需求减少了这种块的大小。回收的单位被增加到链表上。不久之后，这种链表变成了一长列各种不同大小的段（或块）。这就减慢了分配的速度，因为分配需要搜索这个链表，以找到足够大的段。最终，这个链表可能包括了大量非常小的块。面对于大部分的需求它们都不足够大。此时，可能需要将相邻的块合并成为较大的块。另一种方法是将链表按块的大小排序。这样可以缩短找到足够大的块的时间。但无论上述两种方法中的哪一种，链表的维护都成为额外的开销。

如果使用引用计数器，前两个问题就能够避免。但可用空间链表的维护问题仍然会存在。

小结

一种语言的数据类型很大程度上决定了这种语言的风格，以及这种语言的应用。语言的数据类型与语言的控制结构一起，形成了一种语言的核心部分。

大多数命令式语言的基本数据类型包括数值类型、字符类型和布尔类型。数值类型常常由硬件直接支持。

用户定义的枚举类型和子范围类型很方便应用，又增加了程序的可读性与可靠性。

数组是大多数程序设计语言的部分。在一个存取函数中给出了数组元素的引用与这个元素的地址之间的关系。所以存取函数是一种映射的实现。数组可以为静态，如在C++中的数组，这种数组的定义中包括了static说明符；也可以为固定栈动态的，如C中的函数（它不具有static说明符）；还可以为栈动态的，如在Ada中的块；或者固定非动态的，如Java的对象；还可以是堆动态的，如在Perl中的数组。大多数的语言只允许步量的对整个数组的操作。

异构数组是指其元素可以是不同类型的数组，Perl、JavaScript、Python和Ruby都支持异构数组。

记录现在已经被包括进大多数语言中。记录的域通过多种方式来说明。在COBOL中，可以引用域而并不需提及所有包含这个域的记录。但这样实现起来十分繁乱，并且还会影响可读性。在Java中，记录由类结构支持。

联合，就是可以在不同的时间存储不同类型值的地址。判别的联合包括了一个记录当前类型值的标志。自由联合是一种没有标志的联合。大多数具有联合的语言，除了Ada语言以外，其联合设计并不安全。

集合常常是很方便的，它的实现也相对容易。然而，通常需要使用集合的那些应用可以毫无困难地由其他数据类型来完成。

指针可增进灵活性，并控制动态存储管理。指针具有一些内在的危险性，难以避免悬挂指针，还会出现存储泄漏。

引用类型，如Java中的引用类型，可以提供堆管理，但不具有指针的危险性。

实现一种数据类型的难易程度，对这种类型能否被包括进语言极其影响。枚举类型、子范围类型和记录类型都相对地容易实现。数组也很单纯，尽管当数组具有多个下标时，存取数组元素有很高的代价。存取函数需要对每一个下标都进行一次加法和乘法运算。

如果不考虑堆管理，指针的实现也相对容易。如果所有单位都是单一大小的，堆管理会比较容易，但可变大小单位的分配及解除分配则较为复杂。

文献注释

有关于数据类型的设计、使用以及实现方面，有着极为丰富的计算机科学文献。Hoare在（Dahl et al.，

1972）中给出了最早期结构化类型的系统定义。Cleaveland（1986）给出了几种更广泛的数据类型的一般性讨论。

Fischer和LeBlanc（1988）讨论了对Pascal数据类型可能具有的非安全性实现运行时检测。大多数关于编译器设计的书，如Fischer和LeBlanc的专著（1988），以及Aho等人的专著（1986），都描述了数据类型的实现方法。Pratt和Zelkowitz所著的程序设计语言教程（2001），以及Scott所著的教程（2000）也都给出了数据类型的实现方法。关于废弃单元问题的详细讨论，可以在Tenenbaum et al.（1990）中找到。垃圾收集方法由Schorr和Waite（1967）以及Deutsch和Bobrow（1976）开发。关于垃圾收集算法的全面讨论，可以在 305
Cohen（1981）以及Wilson（2005）文献中找到。

复习题

1. 什么是描述符？
2. 小数数据类型具有什么优点和缺点？
3. 字符串类型的设计问题是什么？
4. 描述串长度的三种选择。
5. 定义序数类型、枚举类型和子范围类型。
6. 用户定义的枚举类型有什么优点？
7. C#中用户定义的枚举类型在哪些方面比C++中的枚举类型可靠性更高？
8. 数组有哪些设计问题？
9. 定义静态数组、固定栈动态数组、栈动态数组、显式堆动态数组以及堆动态数组，每一种数组各有什么优点？
10. 什么是异构数组？
11. 当在Perl中引用一个不存在的数组元素时会发生什么？
12. JavaScript怎样支持松散数组？
13. 什么语言支持负下标？
14. 什么语言支持带有步长的数组片？
15. 在Ada中的哪些数组初始化特性在其他一般命令式语言中不存在？
16. 什么是聚集常量？
17. 什么样的数组操作是专门为Algol中的一些数组而提供的？
18. Fortran 95中的片与Ada中的片有什么不同？
19. 定义按行存放和按列存放。
20. 数组的存取函数是什么？
21. Java中的数组描述符要求的项是什么？必须在什么时候将它们绑定（是在编译时，还是在运行时）？
22. COBOL中记录的层号的目的是什么？
23. 对于记录中的域，定义完全限定引用及省略引用。
24. 定义联合、自由联合以及判别的联合。
25. 关于联合的设计问题是什么？
26. 通常总是对Ada中的联合进行类型检测吗？
27. 关于指针类型有哪些设计问题？
28. 指针具有的两个基本用途是什么？
29. 为什么大多数语言中的指针都被限制了只能够指向一种类型的变量？
30. 什么是C++的引用类型？它有哪些基本应用？
31. 为什么将C++中的引用变量用于形参时比使用指针更好？ 306
32. Java以及C#中的引用类型变量具有什么超越其他语言中的指针的优点？

33. 描述垃圾收集的懒惰方法和积极方法。
34. 为什么对Java及C#中的引用执行算术运算没有意义？

练习题

1. 支持与反对在存储空间将布尔值表示为单个位的论点各是什么？
2. 为什么小数值浪费存储空间？
3. VAX小型计算机使用一种不同于IEEE标准的浮点数格式。这种格式是什么？为什么VAX计算机的设计人员选择了这种格式？关于VAX浮点数的表示可以参考文献（Sebesta，1991）。
4. 从安全性和实现代价的角度，对悬挂以及锁-钥匙两种避免悬挂指针的方法进行比较。
5. 指针的隐式间接引用存在什么缺点？这些缺点仅仅在某些特定情形下才产生吗？例如，考虑Ada中一个指向记录的指针的隐式间接引用。当使用这个指针来引用一个记录的域时。
6. 解释子类型与派生类型之间的所有差别。
7. C与C++中关于->操作符的使用有哪些主要的辩解？
8. 列举C++中的枚举类型与Java中的枚举类型之间的所有不同。
9. C和C++中的联合与这些语言中的记录是分开的，而不像在Ada中那样，这两者是结合在一起的。这两种不同的设计选择分别具有什么优点和缺点？
10. 可以使用按行存放的方法来储存多维数组，如在C++中那样，也可以使用按列存放的方法，如在Fortran一那样。请为以上这两种存放方法开发三维数组的存取函数。
11. 在Burroughs扩展的ALGOL语言中，将矩阵储存为指针的一维数组，放置于矩阵的行。因此能够将它们像数值的一维数组一样来处理。这种方案具有什么样的优点及缺点？
12. 分析并且写出C中的malloc和free函数与C++中的new和delete操作符的比较。将安全性作为比较中的主要考虑因素。 307
13. 分析并且写出使用C++中的指针，以及使用Java中的引用变量，或引用堆动态变量之间的比较。将安全性以及方便性作为比较中的主要考虑因素。
14. Java的设计人员没有将C++的指针包括进来，针对这种设计决定写出一段关于这种决策的得失的简短讨论。
15. 当比较C++中所要求的显式堆存储空间的回收时，支持与反对Java中隐式堆存储空间回收的论点各是什么？
16. 支持在C#语言中包括枚举类型的论点是什么？尽管在Java语言早期的一些版本中，并没有包括枚举类型。
17. 你期待或建议C#语言中的指针使用保持在什么样的程度？当并不是绝对需要指针时，应该经常使用它们吗？
18. 产生两列关于矩阵的应用，其中的一列要求参差的矩阵，另一列则要求长方形矩阵。现在请给出理由来说明，在程序设计语言中，是否只应该包括参差的矩阵，或者只应该包括长方形矩阵，或者应该包括这两种类型的矩阵？
19. 比较C++，Java以及C#中的类库进行异处理的功能。它们有什么差别？

程序设计练习题

1. 设计一组简单的测试程序，以确定你使用的一个C编译器的类型兼容规则，并且将你的发现写入你的报告中。
2. 确定你使用的一些C编译器是否实现了free函数。
3. 使用某种语言编写一个矩阵乘法程序，要求这种语言进行下标范围的检测，并且你还可以从它的编译器获得汇编语言或者机器语言的版本。确定进行这种下标范围检测所必需的指令数目，并且将这个数目与矩阵乘法程序中的总指令数进行比较。
4. 如果你曾经使用过一个编译器，在其中用户可以说明是否需要进行下标范围的检测。编写一个进行大量矩阵的存取并且记录执行时间的程序。在具有下标范围检测与没有这种检测的情形下运行这个程序，

比较各自的执行时间。 308

5. 用C++语言编写一个简单程序，来检测枚举类型的安全性。至少要包括10个针对枚举类型的操作，以确定什么样的错误或愚蠢行为会是合法的。然后再编写一个执行同样任务的C#程序，并运行这个C#程序，以确定有多少错误或愚蠢行为是合法的。比较你的结果。
6. 使用C++或C#语言来编写一个程序，这个程序要包括两种不同的枚举类型，还包括大量的枚举类型操作。再编写一个仅使用整数变量的同样程序。比较它们的可读性，并预测这两种程序在可靠性方面的差别。
7. 编写一个C程序，这个程序将仅通过下标来大量引用二维数组中的元素，再编写第二个程序，这个程序将进行同样的操作，然而是通过使用指针以及用于储存映射函数的指针的运算来进行数组的引用。这两个程序哪一个更可靠？为什么？
8. 编写一个使用散列并在散列上施行大量操作的Perl程序。例如，这种散列可以储存人的姓名与年龄。可以使用一个随机数生成器来产生3个字符的名字及年龄，并将这些新的名字及年龄加入到散列中。当产生了一个重复的名字时，它只会导致对散列的存取，但不会增加新的元素。重新编写这个程序，但这一次不使用散列。比较这两个程序的执行效率，比较这两种程序的难易程度，以及它们的可读性。 309

第7章　表达式与赋值语句

正如本章标题所示，这一章的重点是关于表达式和赋值语句。我们将首先讨论决定表达式中操作符的计算顺序的语义规则。紧接着，将讨论当函数具有副作用时操作数求值顺序中的潜在问题。然后再讨论预定义和用户定义的两种重载操作符，以及这两种操作符对程序中表达式的影响。接下来将讨论与评估混合模式的表达式。这种讨论导致了宽化及窄化类型转换的定义，包括使用隐式与显式两种方式。然后还将讨论关系表达式和布尔表达式，其中包括短路求值的思想。最后将讨论的是赋值语句，包括从它的最简单形式到它的所有变体。还包括赋值语句作为表达式以及混合模式的赋值。

这一章仅限于讨论命令式语言中的表达式和赋值语句。关于函数式语言和逻辑语言中的表达式说明与求值问题，将分别在第15章和第16章进行讨论。

关于字符串的模式匹配表达式，曾经在第6章作为字符串的一部分讨论过，因而本章不再讲述它们。

7.1　概述

在程序设计语言中，表达式是说明计算的基本方式。理解正在使用语言的表达式的语法及语义对于程序员十分关键。我们曾经在第3章早讲述过一种形式描述表达式语法的方法（BNF）。在这一章里，我们的注意力将集中在由表达式的求值方式所决定的表达式语义，也即表达式的意义是什么。

要理解表达式的求值，就必须熟悉操作符和操作数运算的顺序。表达式操作符的运算顺序服从语言的结合性以及优先级规则。虽然表达式的值经常依赖于这种顺序，但语言的设计人员常常并不说明表达式中操作数的求值顺序。这就允许语言的实现人员进行顺序的选择，从而导致一些程序可能在不同的实现中产生不同的结果。表达式语义上的其他问题还包括类型的错误匹配、强制转换以及短路求值。

命令式程序设计语言的实质是赋值语句占据主导地位。赋值语句的目的是改变变量的值。因而所有命令式语言的共同部分就是在程序的执行期间改变变量的值的概念（非命令式语言有时会包括一系列不同种类的变量，如函数式语言中的函数参数）。

简单的赋值语句指明一条将被计算的表达式和一个放置表达式计算结果的目标位置。如本章将要讲到的，还存在许多基于这种基本形式的变化形式。

7.2　算术表达式

与数学、科学和工程表达式相类似的算术表达式的自动求值是第一种高级程序设计语言的主要目标之一。程序设计语言中算术表达式的大部分特征都来自数学中的传统形式。其中包括了操作符、操作数、括号以及函数调用。操作符可以是一元的（unary），这意味着这些操作符只具有一个操作数，也可以是二元的（binary），即它们具有两个操作数。第7.2.1.4节中将要讲到，基于C的语言还包括了一个三元（ternary）操作符，它则具有三个操作数。

在大多数命令式程序设计语言中的二元操作符是中缀的，这意味着这些二元操作符出现在它

间的操作数之间。在Perl语言是一个例外，它的一些操作符是前缀的，这意味着这些操作符位于它们的操作数之前。

算术表达式的目的是要说明一种算术运算。这种运算的实现必将产生两个动作：通常是从存储器中获取操作数，并在这些操作数上执行这个算术运算。在下面的几节中，我们将研究命令式语言中算术表达式的一般设计细节。

下面是关于算术表达式的主要设计问题，所有这些内容都会在本节讨论：

- 什么是操作符的优先级规则？
- 什么是操作符的结合性规则？
- 什么是操作数的求值顺序？
- 对操作数求值运算的副作用进行限制吗？
- 这些语言允许用户定义的操作符重载吗？
- 在表达式中允许什么样的混合模式？

7.2.1 操作符求值顺序

我们首先来研究指定操作符求值顺序的一些语言规则。

7.2.1.1 优先级

表达式的值至少部分取决于表达式中操作符的求值顺序。考虑下面的表达式：

```
a + b * c
```

设想变量a、b和c分别具有值3、4和5。如果是从左到右计算（先加法，后乘法），其结果是35，但如果是从右到左计算，其结果则是23。 313

除了仅仅是从左到右或者从右到左的计算顺序，数学家们还开发了这样一种概念，即按照计算优先级别的层次来放置操作符，并且表达式的计算顺序将部分按照这种层次差别。例如在数学中，认为乘法比加法具有更高的优先级别，也许由于它有更高的复杂度。如果在上面表达式的例子中，我们参照这种传统的话，则会首先计算乘法。

表达式计算的操作符优先级规则定义了对不同优先级的操作符进行计算的顺序。从语言设计人员的观点来看，表达式的操作符优先级规则是以操作符的优先级别的层次为基础的。在一般的命令式语言中，操作符的优先级规则几乎都是相同的，因为它们都是基于数学中的优先级规则。在这些语言中，乘幂具有最高的优先级（当语言提供乘幂操作时），后面接着是同在一个层次上的乘法和除法，再接着是同一个层次上的二元加法与二元减法。

许多语言还包括了加法与减法的一元版本。一元加法也被称为恒等操作符，因为它通常并不具有相关的操作，因此对其操作数不产生影响。Ellis和Stroustrup在论述C++时，将一元加法称为是一个历史性的意外，并将它明白无误地标记为是无用的（Ellis and Stroustrup,1990, p.56）。在Java和C#语言中，当一元加法的操作数为short或byte类型时，就会使这些操作数被隐式地转换为 int类型。因而这种一元加法实际上还是有影响的。当然，一元减法总是改变其操作数的符号。在Java和C#中，一元减法也会引起short和byte操作数被隐式地转换为int类型。

在所有的常用命令式语言中，一元减法操作符可以出现在表达式的开头，或者表达式中的任意位置上，但需要对它加上括号以避免它与另一个操作符相毗邻。例如，

```
A * (- B) * C
```

是合法的，但是

```
A * - B * C
```

314 则通常不合法。

接着，考虑下面的表达式：

```
-A / B
-A * B
-A ** B
```

在前两个例子中，一元减操作符和二元操作符的相对优先级是不相关的——两个操作符的求值顺序对表达式的值没有影响。然而，在最后一个例子中，它却产生了麻烦。在常用程序设计语言中，只有Fortran、Ruby、Visual Basic和Ada有乘幂操作符。在4种语言中，乘幂有比一元减更高的优先级，因此

```
-A ** B
```

等价于

```
-(A ** B)
```

在有种情况下，一元操作符的优先级是迷惑的。Ada中一元减的优先级比mod低，因此表达式

```
-17 mod 5
```

等价于

```
- (17 mod 5)
```

计算结果是-2，而不是3（如果一元减的优先级高于mod）。基于C的语言也会得到这个结果。少数几种常用程序设计语言中的算术操作符的优先级列举如下：

	Ruby	基于C的语言	*Ada*
最高优先级	**	后缀 ++，--	**，abs
	*，/	前缀 ++，--，一元 +，-	*，/，mod，rem
	所有 +，-	*，/，%	一元 +，-
最低优先级	二元 +，-		二元 +，-

**是乘幂操作符。C中的%操作符与Ada中的rem操作符十分类似：它具有两个整数操作数，并产生第一个整数被第二个整数相除以后的余数。[⊖]当两个操作数都为正数时，Ada中的mod操作符与rem操作符完全一样。但是当有一个或者两个操作数为负数时，这两个操作符就不相同。关于C中的++和--操作符，将在第7.7.4节进行描述。Ada中的abs操作符是一个一元操作符，它产生其操作数的绝对值。

APL在语言中是个例外，因为如同下面小节将要介绍的，它只具有单个级别的优先级。

315 优先级只是操作符运算顺序规则中的一个部分，结合性规则也影响着操作符的运算顺序。

7.2.1.2 结合性

考虑下面的表达式：

```
a - b + c - d
```

如果表达式中的加法操作符与减法操作符具有同样的优先级，优先级规则对于这个表达式中操作符的运算顺序就不起任何作用。

当一个表达式包含了两个相邻[⊖]出现、具有同样优先级的操作符时，先计算哪个操作符是由语言的结合性规则来决定。一个操作符可以具有左结合性或右结合性，它们分别表示，先计

⊖ 在C99之前的C版本中，%操作符在某些情况下是依赖于实现的，因为无法从其他地方获得依赖于实现的行为。

⊖ 如果一对操作符之间由另一个操作数分隔，这对操作符也可以是"相邻"的。

算最左边出现的操作符，或者先计算最右边出现的操作符。

常用的命令式语言中的结合性是从左到右的，只有乘幂操作符（当提供乘幂操作时）是从右到左结合的。在下面的Java表达式

```
a - b + c
```

中，先计算左边的操作符。在Fortran和Ruby中的乘幂是右结合的，因而在表达式

```
A ** B ** C
```

中，先计算右边的操作符。

Ada中的乘幂是非结合的，这意味表达式

```
A ** B ** C
```

在Ada中是不合法的。必须给这种表达式加上括号，以明确表示求值顺序，就象下面的表达式

```
(A ** B) ** C
```

或者

```
A ** (B ** C)
```

在Visual Basic中的乘幂操作符“^”是左结合的。

下面给出一些最常用的命令式语言的结合性规则：

语言	结合性规则
Ruby	左：*，/，+，-
	右：**
基于C的语言	左：*，/，%，二元 +，二元 -
	右：++，--，一元 -，一元 +
Ada	左：除了**之外的所有操作符
	非结合的：**

如在7.2.1节中曾经说明的，在APL中，所有的操作符都具有相同级别的优先级。因而在APL表达式中，操作符的运算顺序完全由结合性规则来决定。在这些规则中，所有的操作符都是从右到左结合的。例如，在表达式

```
A×B+C
```

中，先计算加法操作符，然后才计算乘法操作符（×）是APL中的乘法操作符。如果A是3，B是4，C是5，那么这条APL表达式的值是27。

对于常用命令式语言，许多编译器都利用这样一种事实，即一些算术操作符是数学结合的，这意味着，结合性规则对于仅包括算术操作符的表达式的值没有影响。例如，加法是数学结合的，因此在数学上，表达式

```
A + B - C
```

的值并不取决于操作符的运算顺序。如果数学结合操作的浮点操作也是结合的话，编译器就可以利用这个事实进行一些简单优化。尤其是，如果允许编译器对操作符的运算重新排序，它也许可以为表达式运算产生更快速的代码。事实上，编译器的确会进行这类优化。

然而不幸的是，在计算机中，浮点表示法以及浮点算术操作都只是数学的近似（因为大小限制），数学乘幂符是结合的，这并不一定就意味着与其对应的浮点操作也是结合的。事实上，

316

只有当所有的操作数以及中间结果都可以使用浮点标记准确地表示时，这样的过程才严格地为结合的。例如，在有些非正常情形下，计算机上的整数加法都不是结合的。如，假设有一个程序必须计算下面的表达式

```
A + B - C + D
```

在这里，A和C为非常大的正数，而B和D为绝对值非常大的负数。在这种情况下，将B加入A不会引起溢出，但是将C加入A则会引起。同样地，将C加入B不会引起溢出，但是将D加入B则会引起。因为计算机算术的限制，在这种情况下的加法是非结合的。因此，如果编译器将这些加法操作重新排序，就会影响表达式的值。当然，如果假设变量的近似值是已知的，程序人员就可以避免这个问题。程序人员只需要给表达式加括号，以便确保只进行安全顺序的计算。然而，这些情形有时可能以非常微妙的方式出现，那么程序人员多半就不会注意到这种顺序的依赖性。

317

7.2.1.3 括号

程序人员可以通过在表达式中放置括号来改变优先级规则以及结合性规则。表达式中加括号的部分比相邻的没有加括号的部分具有更高的优先级。例如，尽管乘法比加法优先，但在表达式

```
(A + B) * C
```

中，将先进行加法运算。这在数学上是十分自然的，在这个表达式中，乘法操作符的第一个操作数，必须在括号中的子表达式进行加法运算之后才能够被求值。第7.2.1.2节的表达式指定为

```
(A + B) + (C + D)
```

以避免溢出。

允许在算术表达式中使用括号的语言可以省却所有的优先级规则，只需要将所有操作符从左到右或者从右到左地结合。程序人员会使用括号指明所需要的计算顺序。这种方法十分简单，因为无论是程序的编写人员还是阅读人员，都不需要记住任何优先级规则或结合性规则。这种方案的缺点是表达式的书写比较麻烦，也会严重地降低代码的可读性。然而这正是APL语言的设计人员Ken Iverson做出的选择。

7.2.1.4 Ruby表达式

回顾一下，Ruby是一种完全面向对象的语言，这意味着包括字面常量在内的每个数据值都是对象。Ruby支持很多包括基于C的语言的算术和逻辑运算。在表达式方面使Ruby和基于C的语言区分开来的内容是，所有算术、关系和赋值运算符，以及数组索引、移位和位逻辑运算符都是用方法实现的。例如，表达式a+b是一个对a引用对象的+方法的调用，并传递b引用的对象作为参数。

318

一个把运算符实现为方法的有趣的结果是它们都被应用程序重写。然而，这些运算符能重新定义。为预定义类型重新定义运算符是没有用的，而正在第7.3节所描述的，为用户定义类型定义预定义运算符才是有用的。这在某些语言中可以通过运算符重载来完成。

7.2.1.5 条件表达式

我们现在来看看一元操作符?:，它被包含在基于C的语言中。这个操作符被用来建立条件表达式。

有时候我们使用if-then-else语句来执行一种条件表达式语句。例如下面的语句：

```
if (count == 0)
  average = 0;
else
  average = sum / count;
```

在基于C的语言中，可以方便地在赋值语句中使用条件表达式来说明上面的这种运算。这条语句将具有下面的形式：

```
表达式_1 ? 表达式_2 : 表达式_3
```

在这里，“表达式_1”被解释为布尔表达式。如果“表达式_1”的结果为真，则整个表达式的值就是“表达式_2”的值，否则，它就是“表达式_3”的值。例如，上面的if-then-else语句可以通过在下面的赋值语句中使用条件表达式来实现：

```
average = (count == 0) ? 0 : sum / count;
```

在效果上，问号标志着then子句的开始，冒号则标志着else子句的开始。这两个子句都是强制性的。注意，这里将?:作为一个三元操作符用于条件表达式。

可以将条件表达式用于程序中任何可以使用其他表达式的位置（在基于C的语言中）。包括基于C的语言、Perl、JavaScript和Ruby提供了条件表达式。

7.2.2 操作数求值顺序

操作数的求值顺序是一种通常较少被讨论的表达式设计特征。表达式中变量的求值是通过从存储器中获取变量的值来进行的。有时也以同样的方式进行常量的求值。在其他情况下，常量可能是机器语言指令的一部分，并不需要从存储器取得。如果一个操作数是一个被加上括号 319
的表达式，那么在括号中包括的所有操作符必须在其值可以被作为操作数使用之前就完成计算。

如果一个操作符的两个操作数都没有副作用，那么操作数的求值顺序是无所谓的。因此，只有当操作数的求值确实存在副作用时才会产生有趣的情形。

7.2.2.1 副作用

当函数改变它的一个参数或者一个全局变量时，就会产生函数的副作用（或函数副作用）。（全局变量是被声明于函数之外，但可以在函数中访问的变量。）

考虑下面的表达式

```
a + fun (a)
```

如果fun在a变更时没有副作用，那么操作数a和fun(a)的求值顺序不会影响表达式的值。然而，如果fun在a改变时具有副作用，则表达式的值会有影响。考虑下面的情形：fun返回10，并且改变其参数值为20。假设我们有

```
a = 10;
b = a + fun (a);
```

那么，如果在表达式的运算过程中先获取a的值10，则表达式的值为15。但如果是先计算第二个操作数，那么第一个操作数的值是20，而表达式的值则是25。

下面的C程序演示了当函数改变表达式中的全局变量时所产生的副作用问题：

```
int a = 5;
int fun1() {
  a = 17;
  return 3;
} /* of fun1 */
void main() {
```

历史注释

Fortran 77的设计人员设计了第二种对于操作数求值顺序问题的解决方案。Fortran 77的定义声明：只要当函数不改变表达式中其他操作数的值时，包括该个函数调用的表达式才是合法的。然而不幸的是，编译器并不容易确定地确定一个函数对于函数以外的变量所具有的影响，尤其是当存在由Common提供的全局变量，以及由Equivalence提供的别名使用时。这种情形正好说明：语言定义仅仅说出了某种结构的合法条件，然后这种结构在程序中的合法性完全由程序人员来保障。

```
    a = a + fun1();
} /* of main */
```

a在fun2中的计算值取决于表达式a + fun1()中操作数的求值顺序。a的值将是8（如果先求值a）或者20（如先求值函数调用）。

注意，数学中的函数没有副作用，因为数学没有变量的概念。它对于纯函数式程序设计语言同样是正确的。在数学运算和纯函数式程序设计语言中的函数比命令式语言的函数更容易处理和理解，因为它们的内容与它们的含义是不相关的。

320 对于操作数求值顺序的问题，有两种解决办法。第一种办法是语言的设计人员可以通过禁止函数的副作用从而限制函数计算影响表达式的值。避免这个问题的第二种办法是，在语言的定义中说明将以某种特定顺序对表达式中的操作数求值，并要求实现人员能够保证这种顺序。

完全禁止函数副作用是十分困难的，况且全面地禁止将会取消程序人员的某些灵活性。考虑C和C++中的情形，这两种语言都是仅具有函数的语言，这就意味着子程序将返回一个值。为了限制双向参数的副作用，但却仍然能够提供返回多个数值的子程序，就需要一种与其他命令式语言中的过程类似的新型子程序类型。还必须禁止从函数中访问全局变量。然而，当效率十分关键时，允许对全局变量的访问来避免参数传递是提高执行速度的一种重要方法。例如，在编译器中，对符号表这种数据的全局访问是极为普通的。

具有严格的求值顺序所产生的问题是，编译器使用的一些代码优化技术将会对操作数求值进行重新排序。当涉及函数调用时，受到保证的顺序就不能允许这些优化。遗憾，正如在实际语言设计中证实的那样，没有一种十全十美的解决办法。

Java语言的定义保证大致将操作数以从左到右的顺序求值，从而避免了本节所讨论的问题。

7.2.2.2 引用透明和副作用

引用透明的概念与函数副作用相关联并受其影响。如果程序中某两个有相同值的表达式能在该程序的任何地方被彼此替换，而不影响程序的动作，那么该程序就具有引用透明性。引用透明函数的值全部取决于它的参数。[6] 下面的例子说明了引用透明和函数副作用的结合：

```
result1 = (fun(a) + b) / (fun(a) - c);
321 temp = fun(a);
result2 = (temp + b) / (temp - c);
```

如果函数fun没有副作用，result1和result2就是相等的，因为赋值给它们的表达式是等价的。然而，假如fun有让b或c加1的副作用，result1将不等于result2。因此，副作用违反了程序的引用透明。

引用透明程序有一些优点。最重要的优点是引用透明的程序的语义比非引用透明的程序的语义更容易理解，成为引用透明使函数以更易于理解的方式等价于数学函数。

因为用纯函数式语言编写的程序没有变量，所以它们都是引用透明的。纯函数式语言的函数不能有存储于局部变量中的状态。如果这样一个函数使用一个从函数外传来的值，那么该值必须是常量，因为没有变量。这样，函数值依赖于它的参数值和可能的一个或多个全局常量。

第15章将深入讨论引用透明。

7.3 重载操作符

我们常常将算术操作符用于多种目的。例如，在命令式程序设计语言中，“+”常用于任意

[6] 也就是，函数值不依赖于求值参数的顺序。

数值类型的操作数的加法。某些语言，如Java，也使用它进行串的连接。一个操作符的多种用途被称为操作符重载。人们通常认为可以接受操作符的重载，只要它没有影响语言的可读性及可靠性。

要说明这种重载可能具有的危险性，我们考虑在C语言中&符号的使用。当被用作二元操作符时，它说明按位的逻辑与（AND）操作；然而当被用作一元操作符时，它的意义则完全不同。当被用作一个以变量为操作数的一元操作符时，它的表达式的值即是这个变量的地址。在这种情况下&符号称为地址操作符。例如，执行下面的表达式

```
x = & y;
```

就导致将y的地址放置于x之中。&符号的多种用法具有两个问题：第一，使用同一个符号 322
进行两种完全不相关的操作有损于可读性；第二，因为输入错误而漏掉了按位逻辑与操作中的第一个操作数，而编译器不会检测出来，因为它将&符号解释成为地址操作符。这种错误可能很难被诊断出来。

基本上，所有的程序设计语言都具有虽然不那么严重但却十分类似的问题。这个问题通常来源于减法操作符的重载。编译器此时将分不清这个操作符究竟应该是一元还是二元的。所以同样地，当操作符应该为二元但却遗漏掉第一个操作数时，编译器不能够发现这个错误。当然，一元与二元，这两种操作的意义至少还是紧密相关的，因而还不至于严重地影响可读性。

使用不同的操作符符号，不但可以增加可读性，时常还会使于一般的操作。除法操作符就是这样的一个例子。考虑计算一组整数的浮点平均数值的问题。通常，整数之和被计算为整数，假设将求和的结果存入变量sum，并将值的个数存入count中。现在，如果将要计算这个浮点平均数值，并将它放到浮点变量avg之中，在C++中可以将这种计算说明为

```
avg = sum / count;
```

但是这项赋值在大多数情况下会产生一个错误结果。因为这个除法操作符的两个操作数都是整数类型，整数除法的求值结果被舍位为整数，那么，尽管它的最终结果（avg）为浮点类型，但从这个赋值语句得到的数值却不能够具有小数部分。除法的整数结果在整数除法的舍位之后才被转换为浮点数。

解决这个问题的一种方法是包括两个不同的除法操作符，一个操作符用于整数除法，另一个操作符则用于浮点数除法。Pascal就是使用这种解决办法，语言中的div说明整数除法，而/符号则说明浮点除法。这面可以使用下面的赋值来获得两个整数sum和count相除的正确浮点数商：

```
avg := sum / count
```

在这里，avg为浮点类型。这里的两个操作数都会被隐式地转换为浮点数，从而进行浮点数的除法操作。这类隐式的转换操作将在后面的第7.4.1节中进行讨论。

JavaScript中没有整数算术运算，因而避免了这种问题。

PHP则使用另一种解决办法。在PHP中，如果两个整数相除的商不为整数的话，产生的结果就是一个浮点数。

除了JavaScript以及PHP之外，其他语言就必须对整数操作数进行显式的类型转换。这种转换将在第7.4.2节中讨论。

一些支持抽象数据类型（见第11章）的语言，例如Ada、C++、Fortran 95以及C#，允许程序人员进一步将操作符重载。例如，假设一个用户想要将 * 定义在一个标量整数与一个整数数组之间，令其意义为，数组的每一个元素都与这个标量整数相乘。这可以通过编写一个完成这

种新的操作名为 * 的函数子程序来定义操作符。当使用一个重载操作符时，编译器将会基于操
323 作数的类型来选择正确的意义，就像使用语言定义的重载操作符一样。例如，如果这种 * 的新定义是被定义在一个C#程序中，每当这种 * 操作符的出现是以一个简单整数作为其左操作数，并以一个整数数组作为其右操作数时，C#的编译器就会对 * 符号使用新的定义。

如果谨慎地使用用户定义的操作符重载，会有助于提高可读性。例如，如果 + 和 * 操作符对矩阵抽象数据类型重载，并且A、B、C和D为这种类型的变量，那么就可以使用

```
A * B + C * D
```

来代替

```
MatrixAdd ( MatrixMult(A,B), MatrixMult(C,D))
```

但在另一方面，用户定义的操作符重载则可能有损可读性。可以肯定的是，没有什么能阻止用户将 "+" 定义为乘法。此外，当读者在程序中看见一个 * 操作符，首先必须找出两个操作数的类型，以及操作符本身的定义，以便确定它的实际意义。任何或者所有的相关定义都可能是在其他文件中。

C++具有少数不能被重载的操作符，这些操作符是类或结构成员操作符（.），以及作用域操作符（::）。有趣的是，操作符重载是没有被复制到Java中的C++语言特性之中，然而却出现在C#之中。

关于用户定义的操作符重载，将在第9章中进行讨论。

7.4 类型转换

类型转换可以是窄化的转换也可以是宽化的转换。**窄化转换**将值转换成另一种类型，但这种类型并非可以存储所有原有类型的值，那值是近似的值。例如，在Java中将一个double的值转换成一个float的值（double的范围比float的范围大很多）。**宽化转换**也是将值转换成另一种类型，但是这种类型至少能够包括所有原有类型的值的近似值。例如，在Java中将一个int的值转换成一个float的值。宽化转换几乎总是安全的转换的值的大小保持不变。而窄化转换则不然——有时转换的值的大小会在处理过程中改变。例如，如果将浮点数1.3E25转换成
324 一个整数，其结果将与原数值相去甚远。

对于基本数值类型来说，宽化和窄化转换的问题是相对简单的。例如，在Java中，下面内容是基本数值类型的宽化转换：

```
byte to short, int, long, float, or double
short to int, long, float, or double
char to int, long, float, or double
int to long, float or double
long to float or double
float to double
```

窄化转换是：

```
short to byte or char
char to byte or short
int to byte, short, or char
long to byte, short, char, or int
float to byte, short, char, int, or long
double to byte, short, char, int, long, or float
```

虽然宽化转换通常是安全的，但却有可能降低准确度。在许多语言的实现中，尽管将数到

浮点数的转换为宽化转换，但可能会丢失某些准确度。例如，在某些实现中，将整数存储于32位，允许至少9位十进制数字的精确度。但在许多情况下，也将浮点数值存储于32位，却只具有大约7个小数数字的精确度（因为需要字位来存放乘幂）。因此，整数到浮点数的宽化就可能导致损失了两位数字的精确度。

当然，非基本类型的强制转换是更复杂的。第5章讨论了数组和记录类型的赋值兼容性的复杂性。这也存在着方法的参数类型和返回类型允许全重写超类中的方法的问题——只有当类型相同时，或一些其他情况下，与子类作为子类型的概念相同，第12章将讨论这个问题。

类型的转换可以是显式的也可以是隐式的。下面的两个小节将讨论这两种类型转换。

7.4.1 表达式中的强制转换

算术表达式设计中的决策之一，是决定一个操作符是否能够允许不同类型的操作数。如果允许则被称为**混合模式表达式**，包含这种表达式的语言必须为隐式的操作数类型转换定义一种被称为强制转换的规则，因为通常，计算机并没有取不同类型操作数的二元操作。第5章里我们曾经定义过强制转换，它是一种由编译器启动的隐式类型转换。我们将由程序人员明确要求的类型转换称为显式转换或类型转换，而不是强制转换。

> **历史注释**
>
> 太多强制转换会造成危险并为此付出代价，下面给出一个较为规则的例子，考虑PL/I为了实现表达式中的灵活性而做出的努力。在PL/I语言中，可以将一个字符串变量与表达式中的一个数组结合。在运行时，系统对这个字符串进行扫描以便找到数值。如果这个数组恰恰包含了一个小数点，将设这个数值为浮点数类型，将另一个操作数强制转换为浮点数，并且产生最后结果的操作也是浮点的。这种强制转换的策略是高代价的，因为必须在运行时进行类型检测和类型转换。这也消除了在表达式中发现程序人员错误的可能性，因为二元操作符可以将一个任何类型的操作数与另一个任何类型的操作数相结合。

虽然可以将某些操作符重载，但我们假设一个计算机系统，在它的硬件或某个层次的软件模拟中，对每一种操作数类型和语言中所定义的每一个操作符都具有一种相应的操作。[①]对于使用静态类型绑定的语言中的一种重载操作符，编译器是基于操作数的类型来选择正确的操作类型。当操作符的两个操作数属于不同的类型，但是这种情况在这种语言中为合法时，编译器必须选择其中的一个操作数进行强制转换，并为这种强制转换提供代码。在下面的讨论中，我们研究在一些常见语言中有关强制转换的设计选择。

语言的设计人员在算术表达式的强制转换问题上没有达成一致。那些反对广泛范围的强制转换的人员，担心强制转换会引起可靠性问题，因为这类转换将抵消类型检测的优势；而那些愿意包括所有强制转换的人员，则更担心严格的限制会导致丧失灵活性。这里的问题是，程序人员是否应该关注这种类型的错误，或者编译器是否应该检测这种类型的错误。

作为对这个问题的一种简单解释，考虑下面的Java代码：

```
int a;
float b, c, d;
...
d = b * a;
```

假设，这个乘法操作符的第二个操作数应该为c，但是由于键入错误，它变成了a。因为在

① 这种假设对许多语言都是非真的。我们将在本节后面的讨论中给出关于这点的一个例子。

Java中，混合模式表达式是合法的，编译器就不能够检测出这个错误。结果，编译器只是插入代码来强制转换int型操作数a的值为float。如果在Java中，混合模式表达式为不合法的，编译器就能够发现这个键入错误引起的类型错误。

由于允许混合模式表达式会减少发现错误的可能性，所以Ada只允许表达式中少量的混合类型操作数。它不允许将表达式中的整数与浮点操作数相混合。然而还是有一种例外：乘幂操作符**可以使用一个浮点数或整数类型来作为它的第一个操作数。使用一个整数类型来作为它的第二个操作数。Ada也允许少数其他种类的操作数类型相互混合，通常这些操作数的类型是与子范围类型有关的类型。如果是使用Ada语言来编写上面的Java代码，就成为下面的样子：

```
A : Integer;
B, C, D : Float;
...
C := B * A;
```

326

Ada编译器将会认为上面的表达式是错误的，原因是对于"*"操作符而言，Float操作数与Integer操作数是不能够混合的。

但在大多数其他常见语言中，对混合模式算术表达式不存在限制。

基于C的语言中，有比int类型小的整数类型。在Java语言中，这些类型是byte和short。实际上当有任何操作符作用于它们时，所有这些类型的操作数都被强制转换为int类型，所以虽然可以将数据存储于这些类型的变量中，但只有将这些数据强制转换到一个较大类型之后才能够处理这些数据。例如，考虑下面的Java代码：

```
byte a, b, c;
...
a = b + c;
```

b和c的值被强制转换为int，然后才进行int的加法。最后将其和转换成byte并放入a中。考虑到当代计算机的大容量内存，所以很难想起使byte和short的类型，除非在存储大量数据的情况下。

7.4.2 显式类型转换

大多数的语言都提供进行显式转换的功能，包括窄化转换和宽化转换两种类型。在某些情况下，当显式窄化转换导致了被转换对象的值发生重大变化时，则产生警告信息。

在基于C的语言中，将显式类型转换称为类型转换（cast）。在表达式被转换之前，将期望的类型放入括号内，如下所示：

```
(int) angle
```

使用括号将类型名称括起来的原因之一，是因为第一种允许这种转换的语言C具有几个两个字的类型名称，如long int。

Ada提供具有函数调用语法的显式转换操作，例如，

```
Float(Sum)
```

回顾一下（第5章），Ada也有通用类型转换函数Unchecked_Conversion，它不改变值的表示——只改变其类型。

7.4.3 表达式中的错误

在表达式运算中可能会出现一些错误。如果语言要求进行类型检测，静态或动态，就不会产生操作数类型的错误。我们已经讨论了由于表达式中操作数的强制转换所产生的错误。另一种类型错误是由于计算机算术中的限制，以及算术本身固有的限制。最常见的一个错误是不能够将操作结果表示在必须用来存储它的存储单位中。取决于操作结果的大小，结果太大被称为上溢（overflow），结果太小则被称为下溢（underflow）。至于在算术上的一种限制，是不允许对零实施除法。当然，在数学上遇到禁止的事实，并不妨碍程序企图实施。

327

浮点数的上溢与下溢以及零除法是运行时错误的例子，有时也称为异常。关于允许程序发现并处理异常的语言机制，将在第14章中进行讨论。

7.5 关系表达式和布尔表达式

除了算术表达式之外，程序设计语言还具有关系表达式和布尔表达式。

7.5.1 关系表达式

关系操作符是将两个操作数的值相比较的操作符。关系表达式具有两个操作数及一个关系操作符，关系表达式的值是布尔值，绝非语言中不具有布尔类型。关系操作符也常常为多种类型重载。确定一个关系表达式是真还是假的操作完全取决于操作数的类型。它可能相当简单，就像整数操作数的情形，也可能十分复杂，就像字符串操作数的情形。典型情况下，可以用于关系操作符的操作数类型只有数值类型、字符串类型以及序数类型。

历史注释

Fortran I的设计人员使用了英文缩写，因为在设计Fortran I时，符号>和<没有出现在穿孔卡中。

在一些常用语言中，现有的关系操作符的语法如下：

操作	Ada	基于C的语言	Fortran 95
等于	=	==	.EQ. 或 ==
不等	/=	!=	.NE. 或 <>
大于	>	>	.GT. 或 >
小于	<	<	.LT. 或 <
大于等于	>=	>=	.GE. 或 >=
小于等于	<=	<=	.LE. 或 >=

328

JavaScript和PHP语言还具有另外两个关系操作符，=== 和 !==。这两个操作符分别与 == 和!= 操作符相类似，但是却不允许进行操作数类型的强制转换。例如，下面的表达式

```
"7" == 7
```

在JavaScript里为真，因为当关系操作符的操作数是字符串及数值时，字符串则被强制转换成为数值。然而

```
"7" === 7
```

却为假，原因是这个操作符不允许实施操作数的强制转换。

Ruby使用==作为使用强制转换的等于关系操作符，使用eql?作为不带强制转换的等于（这需要操作数的类型和值是相等的）。Ruby只在case语句的when从句中使用===，这将在第8章里讨论。

关系操作符总是比算术操作符具有较低的优先级。所以在下面的表达式中：

```
a + 1 > 2 * b
```

就是先计算其中的算术表达式。

7.5.2 布尔表达式

布尔表达式由布尔变量、布尔常量、关系表达式以及布尔操作符构成。这些操作符通常包括了用来进行与（AND）、或（OR）和非（NOT）操作的操作符，有时还包括了异或（exclusive OR）和等价操作符。布尔操作符通常只取布尔类型的操作数（布尔变量、布尔字面常量或者关系表达式），而产生的是布尔值。

在布尔代数中，OR和AND操作符有着相同的优先级。为了与此相一致，Ada的OR和AND操作符也具有相同的优先级。然而在基于C的语言中，AND操作符的优先级则高于OR。也许，这是将AND与乘法、OR与加法这无意中地关联起来的结果，而这种关联自然导致了AND的优先级高于OR。

329

因为算术表达式可以成为关系表达式的操作数，又因为关系表达式可以成为布尔表达式的操作数，当而必须将这三类操作符放置于它们彼此相对的优先级层次中。

在基于C的语言中，算术、关系和布尔操作符的优先级为：

最高级别	后缀 --, ++
	一元 +, -, 前缀 ++, --, !
	*, /, %
	二元 +, -
	<, >, <=, >=
	==, !=
	&&
最低级别	||

C99之前的C语言版本在流行的命令式语言中是一个例外，这些版本中不具有布尔类型。因此也不具有布尔值，代之，它们使用数值来代表布尔值，在需要布尔操作数的位置上放置数值变量或常量，并将零值认为是假，而将所有非零值认为是真，这种表达式的计算结果是整数，如果为假，具有值0；如果为真，则具有值1。如果这样一个表达式只有一个关系操作符，没有布尔操作数，那么它的形式是常见的。例如

```
x > 17
```

含有布尔操作符的表达式有不常见的表现形式，如下：

```
ptr && count
```

其中，ptr是指针，而count是int型的变量。该表达式的求值结果是一个整数，当为假时值为0，当为真时值为1。在C99以及C++中，也可以使用算术表达式来作为布尔表达式。

这种C语言设计的一个奇怪结果是使得表达式

```
a > b > c
```

成为合法的。因为C的关系操作符为左结合的，所以先计算最左边的关系操作符，它产生0或1。然后接着将这个结果与变量c相比较。这里就不再存在b与c之间的比较。

一些语言，包括Perl和Ruby，提供了两套二元逻辑操作符，为AND提供了&&和and，以及为OR提供了or和||。一个在&&和and（||和or）之间的不同之处是拼写版具有更低的优先级。而且，and和or只有相同的优先级，但是&&有比||更高的优先级。

在基于C的语言中，逗号非算术操作符存在，总共有超过40个操作符，以及至少具有14个不同层次的优先级别。这显然正好证明了这些语言中操作符集之丰富，以及它们的表达式可能具有的相当程度的复杂性。

正如我们曾经在第6章里陈述过的，可读性要求语言应该包括布尔类型，而不是在布尔表达式中使用数值类型。这种数值类型的代替使用可能会丢失了应被发现的错误。因为任何数值表达式，不论是有意还是无意，都是布尔操作符的一个合法操作数。在其他的命令式语言中，使用任何非布尔表达式来作为布尔操作符的操作数，都被检测为错误。 330

7.6 短路求值

表达式的短路求值是指在没有计算完表达式中所有的操作数或操作符之前，确定表达式的结果。例如，下面算术表达式的值

```
(13 * a) * (b / 13 - 1)
```

如果a为0的话，就将与(b / 13 - 1)的值不相关，因为对于任意的x，0 * x = 0。所以当a为0时，就没有必要计算(b / 13 - 1)，或者进行第二项的乘法。然而在运行期间，却并不容易发现这种算术表达式中的捷径，因而从来也没有被采用。

布尔表达式的值

```
(a >= 0) && (b < 10)
```

如果(a < 0)，整个表达式的值与第二个关系表达式无关，因为对于x的任意值，表达式(FALSE && (b<10))都为FALSE。所以当a < 0时，就没有必要对b、常量10、第二个关系表达式或者&&操作进行求值。不像算术表达式的情形，在执行期间很容易发现这种捷径。

为了解释布尔表达式非短路求值中的一个潜在问题，假设Java不使用短路求值。现在，再假设我们使用while语句来编写查询表的循环。下面是用于这种查询的一段简单的Java代码，假定list具有listlen个元素，它是将要被搜寻的数组，而key是所搜寻的值。

```
index = 0;
while ((index < listlen) && (list[index] != key))
  index = index + 1;
```

如果这种求值不是短路的，则无论第一个关系表达式的值是什么，都需要计算while语句中布尔表达式中的两个关系表达式。因此如果key不在list中，程序将会终止于下标出界的错误。具有index == listlen的循环执行会导致对list [listlen]的引用，从而引起检索错误，因为list被声明为以listlen - 1为下标值的上界。

如果一种语言提供布尔表达式的短路求值并被使用，这将不会成为一个问题。在前面的例子中，短路求值方案将会计算AND操作符的第一个操作数，如果第一个操作数为假，则会越过它的第二个操作数。 321

一种提供布尔表达式的短路求值和在表达式中有副作用的语言允许精巧错误发生。假设将短路求值运用于一个表达式，并且，这个表达式中包含副作用的部分没有被计算，那么副作用就会出现在当整个表达式的计算完成之时。如果这个副作用决定了程序的正确性，短路求值就可能造成一种严重的错误。例如，考虑C的表达式

```
(a > b) || (b++ / 3)
```

在这个表达式中，只有a <= b时才会改变b值（在第二个算术表达式中）。如果程序人员

假设执行期间每一次计算表达式时，都会改变b（并且程序的正确性取决于这种改变），那么这个程序注定将失败。

Ada通过使用两个字的操作符and then以及or else，来允许编程人员说明布尔操作符AND和OR的短路求值。例如，我们再次假设List被声明为具有下标范围1..Listlen，这样，Ada的代码为

```
Index := 1;
while (Index <= Listlen) and then (List (Index) /= Key)
  loop
  Index := Index + 1;
  end loop;
```

当Key不在List之中，并且Index变成比Listlen还大时，将不会引起错误。

在基于C的语言中，通常的AND和OR操作符分别标记为&&和||，它们是短路的。然而，这些语言也具有按位操作符AND和OR，分别标记为&和|，可以将它们用于布尔数值的操作数，但却不是短路的。当然，如果所有的操作数都被限制为1（为真）或者为0（为假）时，这些按位操作符将与通常的布尔操作符等价。

Ruby，Perl和Python的所有逻辑操作符都是短路求值的。

在Ada语言中，包括短路操作符和一般操作符，显然是最佳的设计方案，因为它给程序人员提供了对任何一种或所有的布尔表达式选择短路求值的灵活性。

7.7 赋值语句

332 正如前面所说，赋值语句是命令式语言的核心结构之一，它为用户可以动态地改变值到变量的绑定提供了一种机制。在下面的这一节里，我们将讨论赋值语句的最简单形式，而在余后的几节中，将描述多种替代方案。

7.7.1 简单赋值

简单赋值语句的一般语法为

<目标_变量> <赋值_操作符> <表达式>

几乎目前所有的程序设计语言都使用等号作为赋值操作符，它们用不同于等号的符号作为等于关系操作符，以避免与它们的赋值操作符相混淆。

ALGOL 60开创了使用 := 作为赋值操作符的先例，Ada语言也沿用了这种选择。

在基于C的语言中，将赋值操作符作为二元操作符来处理，并因此能够将它嵌入到表达式中。我们将在第7.7.5节讨论这种操作符。

在一种语言中怎样使用赋值，有关的设计选择十分不同。在一些语言中，如Fortran和Ada，只能将赋值作为单独语句出现，并且将它的目标变量限制为单个变量。然而还有许多不同的方案。

7.7.2 条件目标

C++允许赋值语句的条件目标。例如，

```
flag ? count1 : count2 = 0;
```

它等价于

```
if (flag)
  count1 = 0;
else
  count2 = 0;
```

7.7.3 复合赋值操作符

复合赋值操作符是一种说明普遍需要的赋值形式的简写方法。运用这项技术可以缩写赋值形式，目标变量也可以作为第一个操作数出现在右边的表达式中，例如，

```
a = a + b
```

由ALGOL 68引入的复合赋值操作符后来被C采纳，但形式稍有不同，这种形式也成为其他基于C的语言以及Perl、JavaScript、Python和Ruby中的部分。这些赋值操作符的语法是所要求的二元操作符与 = 操作符的连接。例如，

```
sum += value;
```

它等价于

```
sum = sum + value;
```

基于C的语言中的大多数二元操作符都有复合赋值操作符的版本。

7.7.4 一元赋值操作符

基于C的语言包括了两种特殊的一元算术操作符，实际上它们是缩写的赋值语句。这类操作符将递增和递减操作与赋值相结合。++操作符为递增，--操作符为递减，可以将它们用于表达式中，或者形成单个操作符的赋值语句。它们可以作为前缀操作符出现，这意味着将它们放置于操作数的前面，也可以作为后缀操作符出现，这意味着将它们紧跟在操作数的后面。赋值语句

```
sum = ++ count;
```

将count的值增加1，然后赋给sum。也可以将它表示为

```
count = count + 1;
sum = count;
```

如果将这个操作符作为后缀操作符，如下

```
sum = count ++;
```

则首先将count的值赋给sum，然后才将count递增。其效果与下面这两条语句相同

```
sum = count;
count = count + 1;
```

使用一元递增操作符形成一条完整赋值语句的一个例子为

```
count ++;
```

它仅仅是count的递增。看起来这似乎不像是赋值，但它确实是赋值，并等价于语句

```
count = count + 1;
```

历史注释

首先实现了C语言的PDP-11型计算机具有自动递增和自动递减的寻址模式，它们是将C的递增和递减操作符用作数组下标时的硬件版本。在这里，我们也许会猜测，这些C操作符的设计是基于PDP-11型机器的体系结构的。然而，这种猜测是错误的，因为这些C的操作符继承自B语言，而B语言的设计在第一台PDP-11出现之前就已经完成。

当将两个一元操作符运用于同一个操作数时，其结合性是从右到左的。例如，

```
- count ++
```

首先将count递增，然后才改变它的正负号，因此它是

```
- (count ++)
```

而不是

```
(- count) ++
```

7.7.5 赋值作为表达式

在基于C的语言，Perl和JavaScript中，赋值语句生成一个与赋给目标的值相同的结果。因此可以将它用作表达式，或者将它用作其他表达式中的操作数。在这种设计中，对待赋值操作符就像对待其他任何二元操作符一样，只是这个操作符具有改变左操作数的副作用。例如，在C中经常将语句写为

```
while ((ch = getchar()) != EOF) { ... }
```

在这条语句中，来自标准输入文件（通常为键盘）中的下一个字符通过getchar获得，并将它赋给变量ch。其结果（或者是被赋的值）然后与常量EOF进行比较。如果ch不等于EOF，就执行复合语句{...}。注意，我们必须将赋值语句括起来，在支持赋值作为表达式的语言里，因为在这些语言中赋值操作符的优先级低于关系操作符的优先级。如果没有括号，新的字符会首先与EOF进行比较，然后再将这种比较结果，0或者1，赋给ch。

允许赋值语句成为表达式中的操作数，这个特性所具有的缺点是导致了另一类表达式副作用。这类副作用使得表达式难读又难理解。一个具有任何种类的副作用的表达式都有这种缺点。
335 这样的表达式看起来只是一串具有奇怪执行顺序的指令，很难被解读。但在数学上它是一种值的标记。例如，表达式

```
a = b + (c - d / b++) - 1
```

所标记的指令为

```
赋b给temp
赋b + 1给b
赋d / temp给c
赋b - c给temp
赋temp - 1给a
```

请注意，把赋值操作符当作任何其他二元操作符的处理允许多目标赋值的效果，如

```
sum = count = 0;
```

这个式子先将count赋以零值，然后再将count的值赋给sum。

在C的设计中，赋值操作会进行错误检查，这常常会导致程序错误，尤其是如果我们键入的是

```
if (x = y) ...
```

而不是

```
if (x == y) ...
```

这是一个相当容易产生的错误，然而却不能够被编译器检查出来。编译器仅仅是测试赋给

x的值，而不是测试这个关系表达式（在这个例子中，x的值是执行这条语句时y的值）。实际上这是三项设计决策导致的结果：允许赋值具有一般二元操作符的行为，像使用布尔操作数一样使用算术表达式，以及使用两个非常相似的操作符，=和==，来表达完全不同的意义。这是C及C++的程序缺乏安全性的另一个例子。注意，Java和C#在它们的if语句中只允许boolean表达式，从而不会出现这样的问题。

7.7.6 列表赋值

包括Perl和Ruby等一些最近的程序设计语言提供了多目标，多源赋值语句。例如，在Perl中编写如下语句：

```
($first, $second, $third) = (20, 40, 60);
```

它的语义是，把20赋值给$first，把40赋值给$second，以及把60赋值给$third。如果两个变量的值是相互交换的，可以用一条简单的赋值语句完成，正如： 336

```
($first, $second) = ($second, $first);
```

这就正确地完成了$first和$second的值交换，而没有使用临时变量（至少有一个由程序创建和管理的变量）。

在Perl中，如果右边的值的个数多于左边变量的个数，那么将忽略多余的值。如果左边变量的个数多于右边值的个数，那么多余的变量设置为undef。如果左边包括一个数组名，那么该数组得到所有右边剩下的值。如果在左边有任何变量跟在数组名之后，那么它们都将设置为undef。

Ruby列表赋值的最简单形式的语法与Perl的相类似，除了左边和右边没有插入插弧内之外。Ruby也包括了一些这里没有讨论的、更加详尽的列表赋值形式。

7.8 混合模式赋值

我们曾经在第7.4.1节讨论了混合模式表达式。通常，赋值语句也是混合模式的。它的设计问题是：表达式的类型必须与被赋值的变量的类型相同吗？或者，能够将强制类型转换用于某些类型不匹配的情形吗？

Fortran、C、C++和Perl对于混合模式赋值使用强制转换规则，与那些在混合模式表达式中使用的规则十分类似：也就是说，许多可能类型的混合是合法的，强制转换可以无限制地使用。[1] Ada不允许混合模式赋值。

通过一种与C和C++明显不同的方式，Java和C#只有在所需的强制转换为宽化转换时，才允许混合模式赋值。[2] 因此，可以将一个int值赋给一个float变量，反之却不然。禁止可能的另一半混合模式赋值，是Java和C#采用的一种简单有效的方法。相对于C和C++而言，这种方法提高了Java和C#的可靠性。

在所有允许混合模式赋值的语言中，强制转换仅发生在右边的表达式被计算之后。一种替代方案是：在计算之前，将所有的右边操作数都强制为目标类型。例如，考虑下面的代码：

```
int a, b;
```

---

[1] 注意在Python和Ruby中，类型与关系相关联，而不是变量，因此在这些语言中，没有混合模式赋值等操作。

[2] 并不是完全正确的：如果由编译器默认赋值类型为int的整数字面量赋值给char、byte或short变量，而且字面常量在变量类型的范围之内，那么在窄化转换中把int值强制转换成变量的类型。这种窄化转换不会引起错误。

```
float c;
...
c = a / b;
```

337

因为c是浮点数，因而可以在除法操作之前，将a和b的值强制转换为float，这样较之延迟强制转换的情形，将会产生不同的c值（例如，如果a为2，b为3）。

小结

表达式由常量、变量、括号、函数调用和操作符组成。赋值语句包括目标变量、赋值操作符以及表达式。

表达式的语义在很大程度上由操作符的求值顺序决定。一种语言表达式中的操作符的结合性和优先级规则决定在这些表达式中操作符的求值顺序。如果可能具有函数的副作用，操作数的求值顺序则十分重要。类型转换可以是变化的或者窄化的。某些窄化转换将产生错误的数值。隐式类型转换（或强制转换）在表达式中十分常见，但它抵消了类型检测发现错误的好处，从而导致可靠性降低。

赋值语句以多种形式出现，包括条件目标、多目标以及赋值操作符。

复习题

1. 定义操作符优先级和操作符结合性。
2. 定义函数的副作用。
3. 什么是强制转换？
4. 什么是条件表达式？
5. 什么是重载的操作符？
6. 定义窄化转换和变化转换。
7. 什么是混合模式表达式？
8. 操作数的求值顺序与函数的副作用有什么相互影响？
9. 什么是短路求值？
10. 指出一种总是进行布尔表达式上的短路求值的语言。指出一种从不进行短路求值的语言。指出一种允许程序员选择是否进行短路求值的语言。
11. C是怎样支持关系表达式和布尔表达式的？
12. 复合赋值操作符的目的是什么？
13. C中一元算术操作符的结合性是什么？

338

14. 将赋值操作符处理为算术操作符时，会有哪种可能的缺点？
15. 哪种语言包含列表赋值？
16. 在Ada中允许什么样的混合模式赋值？
17. 在Java中允许什么样的混合模式赋值？

练习题

1. 你在什么时候可能希望编译器忽略表达式中的不同类型？
2. 表述你对允许混合模式算术表达式的看法，分别给出支持以及反对它的论点。
3. 你认为在你喜爱的语言中取消重载操作符会有益吗？为什么会有益处？或者为什么没有益处？
4. 取消所有操作符的优先级规则，并要求用括号表示表达式中所需的优先级。这是不是一个好主意？并给出理由。
5. 应该将C的赋值操作（例如 +=）引入其他的语言吗？为什么？请给出理由。
6. 应该将C的单一操作数赋值形式（例如 --count）引入其他的语言吗？为什么？请给出理由。

7. 描述一种程序设计语言中的加法操作符不可交换的情形。
8. 描述一种程序设计语言中的加法操作符不可结合的情形。
9. 假设存在下列的表达式结合性与优先级规则：

优先级：	最高	*, /, not
		+, -, &, mod
		-(一元的)
		=, /=, <, <=, >=, >
		and
	最低	or, xor
结合性：	从左到右	

通过将所有的子表达式加上括号，并在其右括号的上方加上指示顺序的上标，以指出表达式的求值顺序。例如，对表达式

```
a + b * c + d
```

可以将它的求值顺序表示为

$((a + (b * c)^1)^2 + d)^3$

请指出下列表达式的求值顺序：

a. a * b - 1 + c
b. a * (b - 1) / c mod d
c. (a - b) / c & (d * e / a - 3)
d. -a or c = d and e
e. a > b xor c or d <= 17
f. -a - b

10. 显示练习题9中表达式的求值顺序，假设不存在优先级规则，并且所有的操作符都是从右到左结合的。
11. 为练习题9中定义的表达式写出一个优先级规则以及结合性规则的BNF描述，假设，仅有的操作数名称分别为a，b，c，d和e。
12. 使用练习题11中的文法，为练习题9中的表达式画出语法分析树。
13. 定义函数fun为

```
int fun(int *k) {
  *k += 4;
  return 3 * (*k) - 1;
}
```

假设在下面的程序中使用fun函数：

```
void main() {
  int i = 10, j = 10, sum1, sum2;
  sum1 = (i / 2) + fun(&i);
  sum2 = fun(&j) + (j / 2);
}
```

sum1与sum2的值分别是什么？

a. 如果表达式中的操作数是以从左到右的顺序来求值的？
b. 如果表达式中的操作数是以从右到左的顺序来求值的？

14. 你反对（或者支持）APL中操作符优先级规则的主要论点是什么？
15. 为你选择的某种语言编写一组操作符，使用这些操作符可以消除所有的操作符重载。
16. 根据你所知道的两种语言，当一个被转换的值失去了它的用途时，确定是不是所使用的窄化算式类型转换提供了错误的信息。

339

340 17. 应该允许C或C++的优化编译器改变布尔表达式中的子表达式的顺序吗？为什么？请给出理由。

18. 针对Ada语言重新练习题17中的问题。

19. 考虑下面的C程序：

```
int fun(int *i) {
  *i += 5;
  return 4;
 }
void main() {
  int x = 3;
  x = x + fun(&x);
 }
```

在main中的赋值语句之后，x的值是什么？假设

a. 操作数是以从左到右的顺序来求值的。

b. 操作数是以从右到左的顺序来求值的。

20. 为什么Java语言指定其表达式中所有操作数的求值都按照从左到右的顺序？

程序设计练习题

1. 在某种支持C语言的系统上，运行练习题13中给出的代码，以确定sum2与sum2的值。解释你的结果。
2. 使用C++，Java以及C#语言重新编写程序练习题1中的程序，运行这些程序，并且比较它们的结果。
3. 使用你所喜爱的一种语言来编写一个测试程序。这个程序将确定并且给出这种语言中的算术操作符以及布尔操作符的优先级和结合性。
4. 编写一个Ada程序来说明mod与rem之间的差别。包括使用各种正的及负的操作数。
5. 编写一个Java程序以揭示Java中操作数的求值顺序，条件是其中的一个操作数是方法调用。
6. 使用C++重复程序练习题5。
7. 使用C#重复程序练习题5。
8. 编写一个C++、Java或者C#程序，说明使用表达式作为一个方法中的实参时的求值顺序。
9. 编写一个C程序，包括下面的语句：

341
```
int a, b;
a = 10;
b = a + fun();
printf("With the function call on the right");
printf("b is: %d\n", b);
a = 10;
b = fun() + a;
printf("With the function call on the left");
printf("b is: %d\n", b);
```

并且将fun定义为给a加10。解释你的结果。

342 10. 编写一个C#程序以确定C#是否使用了Java的操作数求值顺序规则。

第8章　语句层次的控制结构

关于程序中的控制流程或执行序列，可以从几个层次上进行研究。我们在第7章中曾经讨论了表达式中的控制流程，它受操作符的结合性以及优先级规则的支配。位于最高层次的是程序单元之间的控制流程，我们将在第9章和第13章里进行讨论。在这两个极端层次之间，是最为重要的语句间的控制流程问题，这就是本章的主题。

我们将从命令式程序设计语言的控制语句的发展概况开始；接着再对选择结构进行透彻的研究，包括单向、双向以及多路选择结构；随后讨论程序设计语言中开发和应用的多种循环结构；然后，我们将简要地讨论具有争议的无条件转移语句；最后描述守卫的命令的控制结构。

8.1　概述

在命令式语言程序中的运算是通过对表达式求值以及将值赋给变量来完成的。然而，只有极少的程序完全是由赋值语句组成的。至少需要两种额外的语言机制，才能够使程序中的计算足够灵活且有效率：一种是在可能的控制流程（语句执行的）路径中进行选择的方法，另一种是引起某些语句顺序重复执行的方法。我们称提供这类功能的语句为控制语句。

第一种成功的程序设计语言Fortran中的控制语句实际上是由IBM 704机器的设计人员设计的。这种语言完全直接与机器语言的指令相关联。因而它们的功能更像是来自指令的设计，而不是来自语言的设计。当时编写程序的困难鲜为人知。结果在20世纪50年代后期，人们完全地接受了Fortran中的控制语句。如果按照现在的标准，这些控制语句根本不能够被使用。

从20世纪60年代中期至70年代中期的这10年间，在控制语句的问题上集中了大量的研究与讨论。这些工作所得出的主要结论之一是，虽然单个控制语句（可选择的goto语句）在最小的限度上已经足够使用，但是设计一种不包括goto语句的语言，也仅仅需要少数几条不同的控制语句。事实已经证明，所有能够由流程图表述的算法都能够使用两种控制语句在程序设计语言中编码：一种是在两条控制流程路径之间的选择语句，第二种是逻辑控制的循环语句（Böhm和Jacopini，1966）。这里的一个重要结论是，无条件转移语句是多余的，它可能很方便，但却不是绝对必要的。这种事实结合使用无条件转移（即goto）中的问题，导致了大量关于goto语句的争论，我们将在8.4.1节中讨论这个问题。

程序人员对于可写性与可读性的关注远胜过对控制语句的理论研究结果。所有广泛应用的语言都包含了多于两种最基本控制语句的控制语句数目。因为大量的控制语句提高了可写性。例如，并不要求对所有的循环都使用while语句，而可以使用for语句来建立循环，而这种循环在可以自然地用计数器来控制时将会相对容易。限制语言中的控制语句数目的主要原因是语言的可读性，因为大量的语句形式将使得程序的阅读人员需要学习较大型的语言。很少有人真正学习过一种大型语言的全部内容，人们通常仅学习使用语言的一个子集，而这个子集又常常与程序人员使用的子集不尽相同。另外，控制语句太少会导致人们使用较低层次的语句，如goto语句，这将降低程序的可读性。

人们曾经就能够提供所需功能的最佳控制语句的集合，以及语言所需要的可写性问题进行过广泛的讨论。实质上，这就是这样一个问题：如果以一种语言的简单性、规模以及可读性作

为代价，究竟可以将这种语言扩展到什么样以增加语言的可写性。

控制结构就是一条控制语句加上受这条语句控制执行的一组语句。

所有的选择语句以及循环控制语句都存在这样一个设计问题：控制结构应该具有多个入口吗？所有的选择语句以及循环语句都控制着代码段的执行，问题是，是否总是从代码段的第一条语句开始代码的执行。现在人们普遍相信，多个入口并没有增加多少控制结构的灵活性，反而由于所增加的复杂程度降低了语言的可读性。请注意，只有在包括了goto语句以及语句标号的语言中，多个入口才成为可能。

在这里，也许读者会问：为什么我们没有把控制结构的多个出口列为一个设计问题。原因是，所有的程序设计语言都允许从控制结构中形成一些形式的多个出口。合理的解释如下：如果控制结构的所有出口都被限制到对结构之后的第一个语句的传送控制（假如控制结构没有显式的出口，控制将流动），那么对可读性没有害处，也没有危险。然而，如果出口有去限制的目标，并能引发控制传送到包含控制结构的程序单元的任何地方，那么它对可读性的损害与在程序中到处使用goto语句是相同的。有goto语句的语言允许它出现在包括在控制结构里的任何
345 地方。然而，问题是包含goto语句，而不是控制结构的多个出口是否受允许。

8.2 选择语句

选择语句在程序中提供在两个或者多个执行路径之间进行选择的方法。这种语句是所有程序设计语言中基本且必要的部分，正如Böhm和Jacopini曾经证明过的。选择语句被归纳为两种普通的类型：双向选择与多向选择。双向选择将在第8.2.1节中讨论，多向选择将在第8.2.2节中讨论。

8.2.1 双向选择语句

虽然当代命令式语言中的双向选择语句十分相似，但是它们有一些变动。一个双向选择器的一般形式为

```
if 控制表达式
then 子句
else 子句
```

8.2.1.1 设计问题

可以将双向选择器的有关设计问题总结如下：

历史注释

Fortran包括了一个名为"算术If"的三向选择器，这个选择器使用一条算术表达式来进行控制。取决于该条控制表达式的值究竟是负数、零值还是正数，它将导致控制流向三条不同标记的语句中的一条。这种结构被列在Fortran 95中将被废弃的特性表之中。

- 控制选择的表达式的形式及其类型是什么？
- 怎样说明then和else子句？
- 应该怎样说明嵌套选择器的意义？

8.2.1.2 控制表达式

基于C的语言不使用保留字then（或一些其他语法标记）来引出then子句，而是将控制表达式放在括号中。在使用保留字then（或可替代的标记）的情形中，就不再需要括号了，Ada语言中常常将括号省略掉。

C89不具有布尔数据类型。因而使用算术表达式作为控制表达式。在C99和C++语言中也是这样。然而在这些语言

中，可以同时使用算术表达式和布尔表达式作为控制表达式，但在其他的当代语言中，如Ada，Java以及C#，就只能使用布尔表达式作为控制表达式。

8.2.1.3 子句形式

在大多数的当代语言中，then子句以及else子句可以是简单语句也可以是复合语句。但 346
Perl语言是一个例外，Perl中所有的then和else子句都必须是复合语句，哪怕代码仅仅包括了一条语句。在基于C的语言以及在Perl，JavaScript和PHP中，是使用大括号构成复合语句，这作为then子句和else子句的主要部分。在Fortran 95、Ada、Python和Ruby中，then子句和else子句是语句序列。完整的选择结构在Fortran 95、Ada和Ruby中用保留字终止了。[⊖]

Python使用缩进来指定复合语句。例如，

```
if x > y :
  x = y
  print "case 1"
```

在复合语句中，所有语句都有相同的缩进量。[⊜]注意，Python使用冒号来引入then子句，而不仅仅是then。

子句形式的变化表示改变选择器意义的指定。下一小节将讨论这部分内容。

8.2.1.4 嵌套选择器

回顾第3章的双向选择结构的文法的语法歧义性问题。文法如下：

```
<if_stmt> -> if <logic_expr> then <stmt>
           | if <logic_expr> then <stmt> else <stmt>
```

问题是，当选择结构嵌套在选择结构的then子句时，是否应该关联一个else子句是不明确的。这个问题反映在选择语句的语义中。考虑下面类似Java的代码：

```
if (sum == 0)
  if (count == 0)
    result = 0;
else
    result = 1;
```

根据else子句是与第一个then子句还是与第二个then子句相匹配，可以有两种不同的方式来解释这种结构。注意，程序的缩进似乎指示了，else子句是属于第一个then子句。然而，除了Python之外，程序缩进对大部分当代语言的语义并不产生影响，因此编译器将忽略这些缩进。

在这个例子中，问题的关键是：else子句跟随在两个then子句之后，而没有另一个else子句帮助说明它与其中的哪一个then子句相匹配，并且也没有语法指示器来说明else子句与其中的哪一个then子句相匹配。在Java中，也如同在许多其他的命令式语言中一样，语言的静态语义说明else子句总是与最靠近的没有配对的then子句相匹配。一条规则在这里被用来起到消除歧义的作用。因而在上面的例子中，这一条else子句将是第二条then子句的匹配语句。使用一条规则而非一个语法实体，它所具有的缺点是，尽管程序人员可能希望else子句作为第一条then子句的匹配语句，并且编译器也将发现这种语法结构是正确的，但它的语义却不然。为了在Java中强制这种替代语义，需要一种不同的语法形式，在这种形式中，内层的if被放入一个复合语句中，例如

347

⊖ 实际上，在Ada和Fortran中有两个保留字：end if (Ada)或end if (Fortran)。

⊜ 复合语句中的语句必须有与if相同的缩进。

历史注释

ALGOL 60的设计人员选择使用语法而不是规则来进行else子句和then子句的连接，尤其是，不允许将if语句直接嵌套于then子句之中，如果必须将一条if语句嵌套于一条then子句的话，则必须将if语句放置在一条复合语句之内，例如后面的Java例子。

这两种设计之间的不同点于Java成为允许人们编写最简的选择器，看上去，这种选择器似乎将else子句与第一条then子句相匹配，但实际上却不是；然而在ALGOL 60中，这种形式在语法上是不合法的，因此不会允许出现Java中的悬挂问题。

```
if (sum == 0) {
  if (count == 0)
    result = 0;
}
else
  result = 1;
```

C、C++和C#具有与Java一样的选择语句嵌套问题。Perl要求所有的then子句都与else子句复合，由此避免了所有的这些问题。在Perl中可以将前面的代码写为

```
if (sum == 0) {
  if (count == 0) {
    result = 0;
  }
} else {
  result = 1;
}
```

如果需要另外一种语义的话，这段代码就可以成为

```
if (sum == 0) {
  if (count == 0) {
    result = 0;
  }
  else {
    result = 1;
  }
}
```

另一个用以避免嵌套选择语句中的问题的方法是使用不同的方式来形成复合语句。考虑Java中if语句的语法结构。then子句跟随着中心表达式，else子句则由保留字else引入。当then子句为单条语句，并且有else子句时，尽管并不需要在语句的结尾作标记，但事实上，保留字else标记了then子句的结束，当then子句为复合语句时，它由一个右花括号结束。然而，如果if语句中的最后一个子句，无论是then子句还是else子句，不是复合语句的话，就没有语法实体来标记整个选择结构的结束。用于这种目的的特殊字将解决嵌套选择器的语义问题，并且增加这种结构的可读性。这就是Fortran 95、Ada和Ruby中对于选择结构的设计。例如，考虑下面的Ruby结构：

```
if a > b then
  sum = sum + a
  acount = acount + 1
else
  sum = sum + b
  bcount = bcount + 1
end
```

这个选择结构的设计比基于C的语言中的选择结构更为规则，因为不论在then子句和else子句里的语句数目是多少，它们的形式都是相同的（这对Perl也是正确的）。回顾在Ruby中，这些子句包含的是语句序列，而不是复合语句。对于在第8.2.1.4节（else子句与嵌套if相匹配）开始时选择器例子的第一种解释，可以编写成为下面的Ruby程序：

```
if sum == 0 then
  if count == 0 then
```

```
    result = 0
  else
    result = 1
  end
end
```

因为保留字end if关闭了嵌套的if，显然，else子句与内层的then子句相匹配。

对于在第8.2.1.4节中选择器例子的第二种解释，else子句与外部if相匹配，也可以编写成下面的Ruby程序： [349]

```
if sum == 0 then
  if count == 0 then
    result = 0
  end
else
  result = 1
end
```

下面用Python编写的结构与上面的Ruby结构在语义上是等价的：

```
if sum == 0 :
  if count == 0 :
    result = 0
else:
  result = 1
```

如果else:缩进到嵌套if同一列的开始处，else子句将与内部if相匹配。

8.2.2 多向选择结构

多向选择结构允许在任意数目的语句或语句组中进行一种选择。因而它是选择器的一种一般化形式。事实上，可以由一个多向选择器来构造一个双向选择器。

在程序中，常常需要在多于两条控制路径中进行选择。虽然多向选择器也可以使用双向选择器和goto语句来构造，但这样产生的结构却很不方便，并且难以编写与阅读，还不大可靠。因此很显然，我们需要有一种特殊的结构。

8.2.2.1 设计问题

多向选择器的一些设计问题与某些双向选择器的问题相类似。例如其中的一个问题是选择器基于的表达式的类型问题。在这种情况下可能值的范围很大，部分原因是可以选择的数目比较大。一个双向选择器需要表达式能够具备两种可能的值。另一个问题是，当选择结构被执行时，是否可以选择单条语句、复合语句或者语句序列。再接下来的问题是，当执行选择结构时，是否可以执行单个可选择段。这对于双向选择器不是一个问题，因为所有的设计都只允许执行期间在一条控制路径上只有一条子句。正如我们将要看到的，对于多向选择器问题的这种解决办法，在可靠性与灵活性之间取得了一种平衡。另一个问题是case值的形式。最后的一个问题是，如果选择器表达式的计算产生了一个数值，而这个数值不是选择段所具有的，这将会导致什么样的结果（这个数值没有被选择段所表示）。这里可能的选择仅仅是，不允许这种情况出现，当这种情况确实出现时，程序结构就不执行任何事情。 [350]

下面是关于这些设计问题的一个总结：

- 控制选择过程的表达式的形式与类型是什么？
- 怎样来说明可选择段？
- 应该将穿过结构的执行流程限制于只能包括单个的可选择段吗？

- 怎样指定case值?
- 如果出现了任何未被选择器的表达式所表示的数值，应该怎样处理?

8.2.2.2 多向选择器示例

C中的多向选择器构造switch也是C++、Java和JavaScript语言中的一部分，它是一种相对原始的设计。它的一般形式为

```
switch (表达式) {
    case 常量_表达式_1: 语句_1;
    ...
    case 常量_表达式_n: 语句_n;
    [default: 语句_n+1]
}
```

这里的控制表达式和常量表达式都为某种整数类型。而可选择语句则可以是语句序列、复合语句或者块。可选择的default段用于控制表达式产生的那些不代表任何可选择段的值。如果控制表达式的值没有代表任何可选择段，而且当前没有默认段，这个结构将不做任何事情。

switch语句允许多个入口，并且在代码段结尾不提供隐式分支。这样就允许控制流程在一次执行中穿过多个可选择的代码段。考虑下面的例子

```
switch (index) {
  case 1:
  case 3: odd += 1;
          sumodd += index;
  case 2:
  case 4: even += 1;
          sumeven += index;
  default: printf("Error in switch, index = %d\n", index);
}
```

[251]

在每一次执行这段代码时，都会打印出错信息。类似地，每一次在执行常量1和3位置的代码时，常量2和4处的代码也被执行。为了能够逻辑地分离这些代码段，就必须包括一种显式分支。break语句实质上就是受限制的goto语句，通常被用于退出switch结构。下面的switch构造使用了break语句来限制在每一次执行中仅执行单个可选择段：

```
switch (index) {
  case 1:
  case 3: odd += 1;
          sumodd += index;
          break;
  case 2:
  case 4: even += 1;
          sumeven += index;
          break;
  default: printf("Error in switch, index = %d\n", index);
}
```

偶然允许控制流程经过一个可选择代码段到达另一个可选择代码段，是较为方便的。这显然就是为什么在switch构造中不存在隐式分支的缘由。当遗漏一个选择段中的break语句，从而导致控制不正确地流到下一段时，这种设计就会出现可靠性问题。C的switch语句的设计人员选择降低某种程度的可靠性以换取灵活性。然而研究表明，人们很少使用将控制流程从一个可选择段过渡到另一个可选择段的这种功能。C语言中的switch语句基于的是ALGOL 68中的多向选择语句，但后者并没有可选择段的隐式分支。C#中的switch语句与其他基于C的语

言中的switch语句不同，它的静态语义规则不允许隐式地执行多个可选择段。按照这条规则，每一个可选择段都必须结束于一个显式的无条件分支语句，或者是一条break语句，它将控制转移出switch构造，或者是一条goto语句，它将控制转移到一个可选择段（或者任意位置）。例如

```
switch (算符) {
   case -1:
      负数 ++;
      break;
   case 0:
      零值 ++;
      goto case 1;
   case 1:
      下数 ++;
   default:
      Console.WriteLine("Error in switch \n");
}
```

352

请注意，Console.WriteLine是C#语言中用来显示字符串的方法。

Ada中的case语句是出现于1966年的ALGOL W语言中的多向选择器语句的后裔。这种结构的一般形式为

```
case表达式 is
   when 选择列表 => 语句序列;
   ...
   when 选择列表 => 语句序列;
   [when others => 语句序列;]
end case;
```

这里的表达式是序数类型的（即整数、布尔、字符或枚举类型），而when others子句是可选择的。

Ada中的case语句选择列表通常是单个的字面常量，但它们也可以是子范围，如10 .. 15，它们还可以使用出符号"|"分开的OR操作符。例如，下面的形式也可以作为选择列表出现：10 | 15 | 20。when others子句被用于未表示的值。Ada要求选择列表被穷举，这个要求稍微提高了可靠性，因为它不允许不经意地忽略一个或者多个选择值。大多数的Ada中的case语句都包括了when others子句，用以保证选择列表是穷举的。因为要求选择列表必须是穷举的，因而从来就不会存在应该如何处理控制表达式具有未被表示的值的问题。

在选择列表中的值必须是互斥的，这就是说，一个常量不能够出现在多个选择列表之中。另外，字面常量（也可以是命名常量）必须具有与表达式相同的类型。

Ada语言的case语句的语义如下所述：首先对表达式求值，然后将所求出的值与选择列表中的字面常量进行比较。如果找到了匹配，就将控制转移到与匹配的常量相关联的语句。当完成了语句的执行时，又将控制转移到case构造之后的第一条语句。因此，Ada语言中的case语句就比基于C的语言中的switch语句更可靠，因为这些switch语句在可选择程序段之后没有隐式的出口。

PHP中的switch语句使用了C的switch语句的语法，但却具有不同的语义。其中的case的值可以为PHP语言中的任何数量类型——字符串、整数或者双精度值。在C中，如果在选择段的尾部没有break，那程序将进入下一段继续执行。 353

Ruby有两种形式的多向选择结构，两种结构都称为case表达式，也都产生最后表达式求

值的值。有一种Ruby的case表达式在语义上是与嵌套if语句的列表是相似的：

```
case
when 布尔_表达式 then 表达式
...
when 布尔_表达式 then 表达式
[else 表达式]
end
```

这个case表达式的语义是自顶向下地一次求值一个布尔表达式。case表达式的值是布尔表达式为真的第一个表达式的值。在这个结构中，else表示真，else子句是可选的。例如，[1]

```
leap = case
       when year % 400 == 0 then true
       when year % 100 == 0 then false
       else year % 4 == 0
       end
```

如果year是闰年，case表达式求值为真。

其他的case表达式形式（更像switch）如下：

```
case 表达式
when 值 then
-语句顺序
when 值 then
- 语句顺序
[else
-语句顺序]
end
```

case值与case表达式比较，一次一个，自顶向下，直到找到一个匹配项。通过使用为所有内建类中定义的===关系操作符来作比较。如果case是一个范围，如（1...100），===定义为内部测试，case表达式的值在给定范围时为真，如果case值是类名，case值是一个case表达式类的对象或其子类的一个对象，===定义为产生真，如果case值是正则表达式，===定义为简单的模式匹配。

354 考虑下面例子：

```
century = case year
          when (1700..1799) then "Eighteenth"
          when (1800..1899) then "Nineteenth"
          when (1900..1999) then "Twentieth"
          else "other"
          end
```

Perl和Python都没有多向选择结构。

8.2.2.3 使用if的多向选择

在许多情况下，switch或case构造（Ruby的case例外）并不适合于多向选择。例如，当选择必须基于布尔表达式，而不是基于一些序数类型时，就能够使用嵌套双向选择器来模拟

[1] 该例来自于Thomas et. al (2005)。

多向选择器。为了改善深层嵌套双向选择器的可读性较差的问题，一些语言，如Perl和Ada，特别为这种用法实施了扩展。在这些扩展中允许丢弃掉一些特殊字。具体而言，如使用单个特殊字来代替else-if词组，并且抛弃了嵌套的if语句上的特殊终结字。然后将嵌套的选择器称为elsif子句。考虑下面的Python选择器构造（注意，在Python中逻辑else if拼写为elif）：

```
if count < 10 :
  bag1 = True
elif count < 100  :
  bag2 = True
elif count < 1000 :
  bag3 = True
```

它与下面的结构等价：

```
if count < 10 :
  bag1 = True
else :
  if count < 100 :
    bag2 = True
  else :
    if count < 1000 :
      bag3 = True
    else :
      bag4 = True
```

else-if版本是这两种形式中可读性较好的一种。注意，这个例子不容易用switch语句来模拟，因为其中每一条可选择语句中的选择都是以布尔表达式为基础的。因此，else-if构造不是switch的重复形式。事实上，在当代语言中没有一种多向选择器比if-then-elsif更为一般化。下面给出一个具有else-if子句的一般选择器语句的操作语义描述，其中的B代表逻辑表达式，而S则代表语句： 355

```
  if E1 goto 1
  if E2 goto 2
  ...
1: S1
  goto out
2: S2
  goto out
...
out ...
```

从这种描述中，我们可以看到多向选择结构与else-if构造之间的不同：在一种多选择构造中，所有的B都是制在将单个表达式的值与其他一些值的一些比较中。

没有包括else-if构造的语言也可以使用同样的控制结构，只是键入量稍微多一些。

上面Python例子的if-then-else-if构造能写成下面Ruby的case语句：

```
case
when count < 10 then bag1 = True
when count < 100 then bag2 = True
when count < 1000 then bag3 = True
end
```

else-if构造基于的是条件表达式的常用数学构造。我们将在第15章讨论的函数式程序设

计语言将经常使用条件表达式作为语言的基本控制构造之一。

8.3 循环语句

循环语句是导致一条语句或一组语句被执行零次、一次或多次的一种语句。Plankalkül以后的每一种程序设计语言都包括了某些重复代码段的执行方法。循环是计算机的基本功能，如果
356 不能循环的话，就会需要程序人员说明程序序列中每一个动作，可用的程序将会极为庞大和不灵活，并且将耗费大量的时间来编写，需要大量的存储空间来存储程序。

语句的重复执行，在一种函数式语言中通常是由递归而不是循环构造来完成的。关于函数式语言中的递归，将在第15章中讨论。

程序设计语言中的第一种循环构造与数组直接关联。它起源于这样一种事实，在计算机时代的最早期，绝大多数的计算都是数值性的，并经常使用循环来处理数组中的数据。

人们已经开发了几种类型的循环控制语句。这种分类是基于设计人员怎样来回答下面两个基本设计问题而定义的：

- 怎样来控制循环？
- 循环的控制机制应该出现在什么位置？

用于循环控制的主要可能类型有逻辑、计数或者两者的结合。对于控制机制位置的主要选择，有在循环的顶部，或者在循环的底部。这里的所谓顶部与底部是逻辑上的标志，而非物理上的标志。问题的实质不是控制机制的物理位置，而是这个机制究竟是在循环体的执行之前还是之后执行并影响控制。我们将在第8.3.3节讨论第二种选择，即能够允许用户决定应该将控制机制放置在什么位置上。

循环结构的体是循环语句控制执行的语句系列。我们使用术语先测试来指循环完成的测试发生在循环体执行之前，而术语后测试指这种测试发生于循环体执行之后。循环语句加上相关的循环体就构成循环构造。

除了基本的循环语句以外，我们还讨论另一种形式，它自成一类，即，用户定义的循环控制。

8.3.1 计数器控制的循环

计数循环控制语句具有一种维持计数数值的变量，称为循环变量。这种语句也包括了一些方式来说明循环变量的初值与终值，以及步长，步长是相邻的两个循环变量值之差。循环的说明包括初值、终值以及步长，这些被统称为循环参数。

尽管逻辑控制的循环比计数控制的循环更为一般，但逻辑控制的循环并不是必然比计数控制的循环应用更普遍，因为计数控制的循环较为复杂，所以它们的设计要求就更高。

计数控制的循环通常由机器指令支持。然而不幸的是，机器的体系结构常常比设计这个结
357 构时所流行的程序设计方法寿命更长。例如，VAX 计算机有一条非常便于后测试的计数控制循环实现的指令，这条指令是在 VAX的设计之时（20世纪70年代的中期）Fortran语言所具有的。但是在VAX计算机被广泛使用时，Fortran已经不再具有这种计数控制循环。

当然，语言构造比机器的体系结构寿命更长，却也是句的事实。例如，本书作者就知道，当代机器已经不再具有实现Fortran算术IF语句的三向分支指令，虽然这条语句最初包括进Fortran语言是因为在半个世纪以前，IBM 704机器曾经具有三向分支的指令。

8.3.1.1 设计问题

循环的计数控制语句存在许多设计问题。循环变量以及循环参数的性质引起了大量的设计

问题。循环变量的类型与循环参数的类型显然应该是相同的，或者至少应该是兼容的，然而究竟应该允许什么样的类型呢？一种明显的选择就是整数，但是枚举、字符以及浮点类型如何呢？另一个问题是，就作用域而言，循环变量是否是一种普通变量，或者它是否应该具有某种特殊的作用域。与作用域问题相关的是在循环终止以后循环变量的值的问题，允许用户在循环中改变循环变量或循环参数会导致代码难以理解，因而，另一个问题则是，这种改变是否值得，因为它虽然可能增加一些灵活性，但也因此增加了复杂性。一种类似的问题是对循环参数求值的次数及这种参数求值所需特定的时间：如果只是对这些参数求值一次的话，这将导致简单而不灵活的循环结构。

下面是对这些设计问题的一个总结：

- 循环变量的类型和作用域是什么？
- 循环变量在循环终止时具有什么样的值？
- 循环变量或者循环参数在循环中被改变应该是合法的吗？如果是，这种改变会影响循环控制吗？
- 循环参数应该只能够被求值一次，还是应该在循环每一次重复时都被求值一次？

8.3.1.2 Fortran 95的Do语句

Fortran 95具有两种不同的计数循环语句，这两种语句都使用了关键字Do。其中的一种语句的一般形式为：

```
Do 标号 变量 = 初值，终值 [, 步长 ]
```

这里的标号为循环体中最后一条语句的标号。如果没有说明步长，取它的默认值1。这里的循环变量必须为Integer类型；循环参数允许为表达式，并且可以具有正或负的值。循环参数在Do语句开始执行时被求值，而将这个值用来计算循环计数，这个计数然后将达到循环要求执行的次数。是循环计数控制着循环，而不是循环参数。即使可以在循环中将循环参数合法地改变，这些改变也不会影响到循环控制。循环计数是不允许用户代码存取的一种内部变量。

历史注释

Fortran 1包括了一种Do计数循环控制语句，在Fortran II、Fortran IV以及Fortran 66中，也同样地保留了这种语句。这种语句的原来特殊为它是后测试的，这将得它有别于程序设计语言中的所有其他计数循环语句。（实际上在Fortran 66的说明中，并没有指出它的Do语句必须是后测试的，但在大多数的Fortran 66实现中，确实是将它实现为后测试的形式。）

358

只有通过Do语句才能够进入Do结构，因而这种语句为单入口的结构。一旦Do语句终止，而不论它是怎样终止的，循环变量都具有了最后被赋予的值。因而循环变量的用途独立于引起循环终止的方法。考虑构造

```
    Do 10 Index = 1, 10
      ...
10 Continue
```

在循环正常终止后，Index的值为11。

下面给出一种关于Fortran 95的Do语句的操作语义描述：[⊖]

```
初值 = 初值_表达式
终值 = 终值_表达式
步长值 = 步长_表达式
do_变量 = 初值
```

⊖ 注意，如果initial value为0，terminal value为实现的最大合法整数值，该描述失败，因为这些值在计算iteration-count时会引起整数溢出。

```
循环_计数 =
  max ( int ( ( 终值 - 初值 + 步长值)/步长值 ), 0 )
loop:
  if 循环_计数 <= 0 goto out
  [ 循环体 ]
  do_变量 = do_变量 + 步长值
  循环_计数 = 循环_计数 - 1
  goto loop
out: ...
```

Fortran 95还包括了Do语句的第二种形式：

```
[名称:] Do 变量 = 初值, 终值[,步长]
  ...
End Do [名称]
```

359

这种Do语句使用一个特别的结束特殊字（或短语）End Do替代了一条标号语句。下面是一个第二种形式的Do结构框架的例子：

```
Do Count = 1, 10
  ...
End Do
```

8.3.1.3 Ada的for语句

Ada的for语句具有以下的形式：

```
for 变量 in [reverse] 离散范围 loop
  ...
end loop;
```

离散范围是整数类型或枚举类型的子范围，如1…10或者Monday…Friday。当保留字reverse出现时，就表示将离散范围内的值以相反的次序赋给循环变量。请注意，Ada的for语句比Fortran的Do语句简单。因为Ada中for语句的步长总是1（或者是离散范围中的下一个元素）。

Ada的for语句中最有意义的新特性是循环变量的作用域，也就是循环的范围。变量在for语句中被隐式地声明，并在循环终止之后被隐式地解除声明。例如，在下面的代码中：

```
Count : Float := 1.35;
for Count in 1..10 loop
  Sum := Sum + Count;
end loop;
```

Float类型的变量Count不受for循环的影响。直到循环终止之前，变量Count仍然是Float类型的，并且具有数值1.35。另外，该Float类型的变量Count在循环中被循环计数变量Count所遮掩。后者被隐式地声明为离散范围的类型，即Integer。

不能够在循环体中对Ada循环变量赋值，但可以在循环体中改变用来说明离散范围的变量。但因为对这种范围仅求值一次，因而这种改变不会影响循环控制。将Ada的for循环体分支是不合法的。下面是Ada的for循环的一种操作语义描述：

```
    [ 定义 for_变量 : 变量类型属于离散范围 ; ]
    [ 计算离散范围 ]
loop:
    if [在离散范围内没有剩余的元素 ] goto out
    for_变量 := [ 离散范围中的下一个元素 ]
```

360

```
    [ 循环体 ]
    goto  loop
out:
    [删除 for_变量的定义 ]
```

因为循环变量的作用域就是循环体，在循环结束之后循环变量将无定义，因而在那时，这些变量的值就已经无关了。

8.3.1.4 基于C的语言的for语句

C的for语句的一般形式为

```
for (表达式_1; 表达式_2; 表达式_3 )
    循环体
```

在这里，循环体可以是单个语句、复合语句或者null语句。

因为这些语句是在C中产生结果，因此可以认为是表达式。而在for语句中的表达式常常是一些语句。第一个表达式用于设定初值，并只被计算一次；第二个表达式是循环控制，并且在每一次执行循环体之前被进行计算。正如通常在C中的情形一样，零值意味着假，所有的非零值意味着真。因此，如果第二个表达式的值为零，则会终止for语句；否则执行循环语句。在C99中，这些表达式还可以是布尔类型，C99中的布尔类型仅仅储存0和1。在for语句中的最后一个表达式于每一次执行循环体之后被执行。它常常用来给循环计数器增值。C的for语句的操作语义描述如下。因为这些C的表达式同时也是语句，我们在此处表示表达式的求值就如同表示语句一样。

```
    表达式_1
loop:
    if 表达式_2 = 0 goto out
    [ 循环体 ]
    表达式_3
    goto  loop
out: ...
```

下面是一个C的for结构框架：

```
for (count = 1; count <= 10; count++)
  ...
}
```

361

C的for语句中所有表达式都是可选的。如果缺少第二个表达式，则认为该表达式为真，因此一条没有第二个表达式的for语句可能是一个无穷循环。如果是缺少第一或者第三个表达式，就没有任何假设。例如，如果缺少了第一个表达式，这仅仅意味着没有设定初值。

注意，C中的for语句不需要进行计数。下一小节中将给出演示，可以很容易地模拟计数以及逻辑循环结构。

C的for语句的设计选择如下：没有显式的循环变量或者循环参数。可以在循环体中改变所有的相关变量。可以按照上面描述的顺序来计算表达式。分支进入C的for循环体是合法的，虽然这会引起很大的混乱。

C中的for语句比Fortran和Ada中的计数循环语句更灵活，因为它的每一个表达式都可以包含多条语句，这就允许了多种任意类型的循环变量。当for语句的单个表达式中包括了多条语句时，使用逗号将这些语句分开。C中的所有语句都具有值，这种形式的多条语句也不例外，这个多条语句的值就是其中最后一条语句的值。

考虑下面的for语句：

```
for (count1 = 0, count2 = 1.0;
```

```
    count1 <= 10 && count2 <= 100.0;
    sum = ++count1 + count2, count2 *= 2.5;
```

这条语句的操作语义描述为：

```
    count1 = 0
    count2 = 1.0
loop:
    if count1 > 10 goto out
    if count2 > 100.0 goto out
    count1 = count1 + 1
    sum = count1 +count2
    count2 = count2*2.5
    goto loop
    out:...
```

上面例子中的C的for语句不需要并且也不存在一个循环体。所需要的行动恰巧是for语句自身的一部分，而不是在它的循环体中。其中的第一个和第三个表达式为多条语句。在这两个表达式中，对整个表达式都进行计算，但是所产生的值没有被用于循环控制。

在两个方面上，C99和C++中的for语句不同于早期C版本中的for语句。首先，除了可以使用算术表达式之外，C99和C++中的for语句还可以将布尔表达式用于循环控制。其次，在这些for语句的第一个表达式中，还可以包含变量的定义，例如，

362

```
for (int count = 0; count < len; count++) { ... }
```

在for语句中所定义的变量作用域是，从变量定义开始，一直到循环体的结尾。

Java和C#的for语句与C++的for语句类似，但它的循环控制表达式被仅限制为boolean类型。

在所有基于C的语言中，每一次的循环都将对循环参数进行求值。另外，还可以在循环体中改变循环参数表达式中的变量。因而，这些基于C的语言中的循环要复杂得多，并且较之Fortran和Ada中的计数循环，可靠性程度也较低。

8.3.1.5 Python的for语句

Python的for的一般形式

```
for 循环_变量 in 对象:
-循环体
else:
-else子句
```

循环变量用有一个范围的对象中的值来赋值。当循环正常终止时，就执行else子句。

考虑下面的例子：

```
for count in [2, 4, 6]:
  print count
```

产生

```
2
4
6
```

Python中大多数简单的计数循环都使用range函数。range接收1个、2个或3个参数。下面例子说明了range的动作：

```
range(5) returns [0, 1, 2, 3, 4]
range(2, 7) returns [2, 3, 4, 5, 6]
range(0, 8, 2) returns [0, 2, 4, 6]
```

注意，range仅给定的参数范围内从来不返回最高值。例如，

```
for count in range(5, 11, 2):
  print count
```

363

产生

```
5
7
9
```

8.3.2 逻辑控制的循环

在许多情况下，需要重复地执行语句系列，但这种重复的控制是基于布尔表达式的，而不是基于计数器的。在这些情形中，使用逻辑控制循环十分方便。事实上，逻辑控制的循环比计数控制的循环更加一般化。任何一个计数循环都可以使用逻辑循环来建造，但反之却不然。另外前面曾经讨论过，只有选择结构和逻辑循环才是表达任何流程图控制结构的关键。

8.3.2.1 设计问题

因为逻辑控制的循环远比计数控制的循环简单，所以它只具有少数的设计问题。

- 控制应该是先测试还是后测试的？
- 逻辑控制的循环仅仅是计数循环的特殊形式，还是一种完全不同的语句？

8.3.2.2 示例

基于C的语言包括了先测试和后测试这两种逻辑控制循环。这些逻辑控制循环并不是这些语言中的计数控制循环语句的特殊形式。先测试和后测试逻辑控制的循环具有下面的形式：

```
while (控制_表达式)
  循环体
```

以及

```
do
  循环体
while (控制_表达式)
```

这两种语句形式都可以通过下面的C++代码段来说明：

```
sum = 0;
indat = Int32.Parse(Console.ReadLine());
while (indat >= 0) {
  sum += indat;
  indat = Int32.Parse(Console.ReadLine());
}
```

366

```
value = Int32.Parse(Console.ReadLine());
do {
  value /= 10;
  digits ++;
} while (value > 0);
```

注意，这些例子中的所有变量都是整数类型。Console对象的ReadLine方法是从键盘输入一行文本。Int32.Parse是从它的字符串参数中查找出数字，并将数字转换成int类型，然后返回。

在先测试版本（以while开始的版本）中，只要表达式的计算结果为真，就执行语句。在

C、C++以及Java中的后测试版本（以do开始的版本）中，在表达式的计算结果为假之前，都将执行循环体。do与while语句之间的唯一真正不同，是do语句总是使得循环体至少被执行一次。在这两种情形中，它们的语句都可以是复合的。下面给出这两种语句的操作语义描述：

```
while

loop:
 if控制_表达式 为假goto out
 [循环体]
 goto loop
out: ...

do-while

loop:
 [循环体]
 if控制_表达式 为真goto loop
```

在C和C++语言中分支进入while和do循环体都是合法的。C89版本使用一种算术表达式来实施控制，而在C99以及C++中，控制表达式可以是算术的，也可以是布尔代数的。

Java中的while和do语句与C和C++中的类似，只是Java中的控制表达式必须为boolean类型，而且由于Java中没有goto语句，所以进入这两种循环体都只能通过循环体的开头，而不能通过其他任何位置。

Fortran 95既没有先测试逻辑循环，也没有后测试逻辑循环。Ada具有一种先测试逻辑循环，但是没有逻辑循环的后测试版本。

361 Perl和Ruby具有两种先测试逻辑循环：while和until，这种until循环十分类似于while循环，仅仅是使用控制表达式的反值。Perl也有两种后测试循环，它在do块使用while和until作为语句修饰符。

后测试循环并不常用，而且这种测试还可能存在某种程度的危险性，因为程序人员有时会忘记循环体总是至少被执行一次。将后测试控制放置在循环体之后，它在这个位置具有语义上的影响，这种语法设计通过使逻辑更加清晰，可以帮助避免这类问题。

8.3.3 用户定位的循环控制机制

在某些情形下，由程序人员来选择除循环的顶端或底部之外的循环控制位置较为方便。因而一些语言提供了这种功能。用户定位的循环控制语法机制可以是相当简单的，因而它的设计也不会很复杂。也许最需要关注的问题是，是否可以退出单层循环或者退出几层嵌套的循环。这种机制的设计问题如下：

- 条件机制应该是出口的整体部分吗？
- 应该只允许退出一个循环体，还是也可以退出包含它的循环？

C、C++、Python、Ruby和C#具有无条件、无标号的退出语句（break语句）；Java、Perl以及C#则具有无条件、有标号的退出语句（Java以及C#中的break语句，Perl中的last语句）。

下面是Java中嵌套循环的一个例子，在这个例子中有从嵌套的内层循环到外层循环的退出。

```
outerLoop:
  for (row = 0; row < numRows; row++)
    for (col = 0; col < numCols; col++) {
      sum += mat[row][col];
      if (sum > 1000.0)
```

```
        break outerLoop;
    }
```

C、C++和Python包括了一条没有标号的控制语句continue，这条语句将控制转移到最小的封闭循环的控制机制。这不是一种退出，而只是一种方式，它在当前的循环中跳越其余的循环语句，并没有终止循环结构。例如，考虑下面的代码：

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) continue;
  sum += value;
}
```

368

在这里，负值使得后面的赋值语句被跳越过去，并将控制转移到循环顶部的条件语句。另一方面，在下面的代码中：

```
while (sum < 1000) {
  getnext(value);
  if (value < 0) break;
  sum += value;
}
```

在这里，负值则终止了这个循环。

Java和Perl都具有类似continue的语句，但是它们包括了语句标号，用以说明应该继续的是哪一个循环。

last和break语句都提供退出循环的多个出口，这在某种程度上妨碍了可读性。然而需要终止循环的非异常情况十分常见，这就支持了这种构造。此外，可读性并没有受到严重的危害，因为所有这些循环退出的目标都是循环（或者一个封闭的循环）之后的第一条语句，而不是程序中的任意位置。最后，选择使用多个break来退出多层循环在可读性上是很差的。

用户定位循环出口的设计动机十分简单：通过一种高度限制的分支语句，来满足对goto语句的需求。goto语句的目标可以是程序中的许多位置，既可以在goto语句的上方，也可以在它的下方。然而，用户定位的循环出口的目标必须是在出口的下方，并且只能紧跟在复合语句的结尾。

访谈



第一部分：语言学及Perl语言的诞生

LARRY WALL

Larry Wall教育许多领域于一他涉足书籍出版、语言出版、软件出版，以及儿童教育（孩有四个孩子）。他曾在Seattle Pacific University（西雅图太平洋大学）、Berkeley（伯克利）和UCLA（加州大学洛杉矶分校）学习过。他还曾经在Unisys公司、Jet Propulsion Laboratories，以及Seagate公司工作过。给他带来最大名气的是语言出版（"但是却只有最少的金钱回报" Larry补充道）。Larry是Perl脚本语言的作者。

专业背景介绍

问：你从事过的最激动的工作是什么？

答：在JPL（Jet Propulsion Laboratories）的工作很有意思。现在那里又做系统管理，又进行系统开发。系统管理员的工作特别好，因为在90%的时间里无所事事，只有10%的时间是去忙碌。所以我有足够的时间用于Perl的开发。

问：你最奇怪的工作是什么？

答：这个问题很难回答。我的大多数工作都很奇怪。让我想一想……我曾经给一个叫“疯马”的野营当过向导。我曾经给“王安”公司的一个系统用BASIC编写过商用软件。这个系统只能够通过“菜单”来控制。我曾经在MusiComedy Northwest做过几年的首席小提琴。也许最有趣的一件事情是录音。因为只有我一个人能够拉小提琴，他们就给我录音八次或者是十次，这样，我就像是整个小提琴声部一样。

问：你当前的工作是什么？

答：我现在的工作是设计Perl语言的第6个版本。不幸的是，目前谁也不给这项工作付一分钱。当然，也没有人为前面的五个版本正式付过我任何工资。这一次，如果有人愿意是非正式地付我一点钱，那就好了。也许当你读到这篇访谈时，已经有人雇用我到好莱坞当作家或者演员了。

语言学与计算机语言

问：你学的是语言学，还在读书时，你考虑的是什么样的专业道路？

答：我和我的妻子曾经打算去非洲作现场语言学家，曾经有一段时间是打算作与Wycliffe圣经有关的翻译工作。直到我发现了自己的食物过敏症（后来才有的）才放弃了这一条路。我至今不知道，我是否会成为一个很好的现场语言学家——也许我待在现场之外，编写Perl语言对语言学做出的贡献更大一些。不容置疑的是，我始终热爱语言学，在过去的几年中，我一直在自学日语，只是为了不让这方面的神经细胞僵化或死掉，或者是自然进化之类。

问：你当初对自然语言的兴趣怎么会转向了程序设计语言？

答：我应该承认，在我写出Perl语言之前，我的语言学与我的计算机科学几乎是相互不搭界的。在计算机方面，我曾经编写过几个编译器，从来也没有反写过自己；为什么大多数的计算机语言看起来都是那么的不自然？在语言学方面，我曾经使用LISP编写过几个自然语言的程序，但我一般从未考虑过，使用计算机来处理自然语言。（任何使用过Babblefish语言翻译器的人都会发现这一点。）但我是直到开始编写Perl语言的工作之后，才开始意识到：之所以自然语言让人感到自然，是因为其中能产生多原则使之然，那么，我们可以将这中间的一些原则教给计算机，而并不会让计算机产生烦恼。

Perl语言的诞生

问：在您的写Perl语言的那些日子，你在Unisys公司做些什么工作？

答：我当时是系统管理员加系统程序员，为美国国家保密局(NSA，National security Agency)的一个机密项目工作。如果我告诉你关于这个项目的事情，你可能会杀死我也不一定，那件事看来很糟糕……。反正，我当时将自己关在一个被称之为“笼屋”的房间里，整天搜索一些对外界根本就不存在的计算机。

问：你是怎样产生了有关新语言的想法的？

答：我们当时试图通过一个低速的加密网络，来进行一种全国性的配置管理。这样就产生了大量的文本数据，但其中有用的信息很少。这点最为Perl语言是设计用来对那些分散着、装有文本的文件进行检索，找出其中的有用信息，并且打印出报告。Perl的两个正式的全名都反映了这个目的。它们是：“Practical Extraction and Report Language”（实用提炼与报告语言）和“Pathologically Eclectic Rubbish Lister”（有毛病的垃圾收集者）。

问：有哪些商业的需求在当时还没有得到满足？

答：实际上，是对于灵活性的要求。我们当时使用UNIX系统就是为了灵活性，然而UNIX系统所提供的用于shell的脚本程序设计工具却远非灵活，不能够像我们所希望的那样，在文本文件之中游刃。UNIX的awk程序设计语言虽然是迈向正确方向的一步，但却并不符合我的要求。我觉得自己可以做得更好。UNIX的工具在它的适用方面真的已经非常地好。但对于它的不适用方面，几乎根本就是无用的东西。问题在于……UNIX的toolbox是想成为一种可扩展的语言。但却没有达到实际上的成功。我想，如果我能够将UNIX中的好东西放进一种真正的语言，这些东西都很灵巧，你想要什么就可以得到什么，那么人们会发现这是很有用的。这就是产生Perl语言的原因。

问：你首先喜欢的是哪一种，是设计一种语言还是设计一种程序？究竟是哪一件导致了另外的一件？

答：我首先回答第一个问题。我从来不认为，在写人可以还没有使用过语言就能够设计一种语言，

我们会自然地自下而上地学习语言，开始会模仿语言中的各种小的部分。只有到后来，才能够归纳出语言中将语言强加在一起的一些原则。而计算机科学家们喜欢认为他们可以自上而下地进行设计。但这种方式只有在你已经知道你所寻求的答案时才会获得成功。一般而言，设计一种解决某些特殊问题的语言并不是一件十分有趣的事情，至少对我来说是这样的。

因此，我必须首先学习程序。我不认为你可以将它视为“一见钟情”，它更像是“一见钟情加上一见钟恨”。真正喜爱程序设计的人不会去设计一种新的语言。他们没有设计语言的动力，只有当你发现你所使用的语言让你沮丧时，你才会急到要设计一种新的语言。所以，如果你想以计算机语言设计作为你的职业，最好要准备好去过一种让你会不断感到沮丧的生活。这就是与这个领域相关的一种事实。如果你发现你已经满足于你的新语言，你就是一个妄自尊大的白痴，你也就不能成为一个好的语言设计者。当然，你也可以像我这样地妄自尊大，但同时又适当地感到些沮丧，这样，你还是有些希望的。

362 | 365

8.3.4 基于数据结构的循环

需要在这里考虑的还有另一种循环结构，即依赖于数据结构的循环。与通过计数器或者布尔表达式来控制的循环不同，这种循环是由数据结构中元素的数目来控制的。Perl、JavaScript、PHP、Java以及C#都具有这种类型的语句。

一般的基于数据的循环语句使用一种用户定义的数据结构和一种用户定义的函数，来遍历结构中的所有元素。这种函数被称为迭代器（iterator）。每一次循环开始时，都将调用迭代器，并在每次调用迭代器时，都按某种特殊顺序从特定数据结构中返回一个元素。例如，假设一个程序有一棵二叉树，其中每一个树节点中的数据都必须按某种特定顺序来处理。针对这种树的一个用户定义的循环语句不断地将循环变量设为指向树中的节点，一次循环一个节点。最初执行这种用户定义的循环语句时，需要对迭代器发出特殊调用，以便获取树中的第一个元素。迭代器必须时刻记住它最后访问的是哪一个节点，这样它就能够遍访所有节点，而不会对任何一个节点进行重复访问；所以迭代器必须是历史敏感的。当迭代器不再能够找到元素时，用户定义的循环语句即被终止。 369

基于C的语言中的for语句，因为具有极大的灵活性，所以可以使用它来模拟用户定义的循环语句。再次假设所要处理的是一棵二叉树的节点。如果一个名为root的变量指向二叉树的根，并且如果traverse是一个函数，这个函数将它的参数设置为指向树中指定顺序中的下一个元素，就可以使用下面的形式：

```
for (ptr = root; ptr == null; ptr = traverse(ptr)) {
  ...
}
```

这条语句中的traverse为迭代器。

在PHP中，使用一些预定义的迭代器来对PHP中特有数组进行迭代访问。将current指针指向循环中上一次访问过的元素，而next迭代器则将current指针移动，将它指向数组中的下一个元素。prev迭代器再将current指针移动到指向数组中的上一个元素。通过使用reset操作符，可以将current指针设置为（或者重新设置为）指向数组中的第一个元素。下面的代码将打印出数字数组$list中的所有元素：

```
reset $list;
print ("First number: " . current($list) . "<br />");
while ($current_value = next($list))
  print ("Next number: " . $current_value . "<br />");
```

用户定义的循环语句在面向对象程序设计语言中比在早期软件开发的范例中更为重要，原因是，现在的用户经常为数据结构构造一些抽象数据类型。在这种情况下，数据抽象的作者就必须提供用户定义的循环语句及其迭代器，因为用户并不知道这种类型对象的表示方式。

在C++中，用户定义类型的迭代器或者类经常被实现为类的友元函数，或者是一些单独的迭代器类。

在Java中，一个由用户定义的、实现Collection接口的集合中的元素可以通过实现Iterator接口来进行迭代式访问。Iterator接口具有三种基本的方法：next，hasNext以及remove。next方法是实际上的迭代器，当被迭代式访问的集合中不再有元素时，next方法将抛出一个NoSuchElementException异常，而hasNext方法则通常是在调用next方法之前被调用，如果在集合中至少还有一个元素，它将返回true。

在Java 5.0版本中加入了一条增强的for语句，这条语句通过在实现了Iterable接口的集合中的对象的值或者数组中的值，从而简化了迭代过程。例如，如果我们有一个ArrayList[⊖]
370 集合（名为myList的字符串数组）下面的这条语句将对myList中的所有元素进行迭代，并且每一次都将一个元素放到myElement中：

```
for (String myElement : myList) { ... }
```

这条新语句被称为“foreach”语句，虽然这条语句的保留字仍然是for。

C#中的foreach语句在数组以及其他集合中的元素上进行迭代，例如：

```
String[] strList = {"Bob", "Carol", "Ted", "Beelzebub"};
...
foreach (String name in strList)
  Console.WriteLine("Name: {0}", name);
```

在上面的代码里，在Console.WriteLine的参数中标记{0}表示，在将打印的字符串中的什么位置上放置第一个命名变量的值，即例子中name的值。

Ruby为其每个预定义容器类提供了迭代器，因为它们与块（实际上是Ruby的子程序）相关联，所以第9章将讨论它们。

8.4 无条件分支

无条件分支语句将执行控制转移到所说明的程序位置上。在20世纪60年代后期，语言设计中争论最热烈的问题是无条件分支是否应该成为任何高级语言的组成部分？如果是，又是否应该限制它的使用？

无条件分支，或者goto语句，是控制程序语句执行流程的最具功效的语句。然而，如果不小心地使用goto语句，则会导致一些问题。goto语句具有强大的功能以及极佳的灵活性（使用goto语句，再加上一个选择器，便能够构造所有其他的控制结构），但是这种非常的功能也使得它的使用十分危险。如果没有在使用上施加语言设计上或者程序设计上的标准来加以限制的话，goto语句可以使程序无法阅读，结果导致程序高度不可靠，并且极难维护。

这些问题直接来自goto语句的功能，即强迫任何程序语句跟随执行顺序中的其他语句，而

⊖ ArrayList是一个预定义的集合，这个集合实际上是一个动态的任意类型对象的数组；也即，它是对任意类型对象的引用的集合，而且它实现了这种可迭代的接口。

不论该语句在文本顺序中是否应该在第一条语句之前或者之后来执行。当语句的执行顺序几乎与语句出现的顺序相同时，可读性为最好，在我们的情形中，语句出现的顺序意味着自顶往下，这是我们习惯的顺序。这样就给goto语句设以限制，使得它们只能够在程序中由上往下来转移控制，以便在部分程度上减轻这种问题。goto语句允许在针对错误或非寻常情况作出反应时，围绕代码段来转移控制，但是不允许用于建立任何类型的循环。 371

历史注释

尽管有一些富有创见的人士很早就提议过，但还是Edsger Dijkstra在给计算机界广为阅读的材料里首先阐述了goto语句的危险性。他在信中说理道，"goto语句的形式太原始；它给制造程序混乱提供了太多机会"（Dijkstra，1968a）。在Dijkstra对于goto语句的观点发表之后的前几年，很多人公开争论是否应该禁止或者至少限制goto语句。在一些并不支持完全废除goto语句的人中，便有Donald Knuth，他争辩道，有时候goto语句的效率超过了它所可读性的损害（Knuth，1974）。

少数语言的设计没有包括goto语句，例如Java、Python和Ruby，然而，当前流行的大多数语言中都包括了goto语句。Kernighan和Ritchie（Kernighan and Ritchie，1978）称goto语句可能被无限地滥用，但它仍然被包括进Ritchie的语言C中。在一些删除了goto语句的语言里，提供了其他的一些控制语句，通常典型地是以循环退出的形式来代替goto语句。

较新的语言C#包括了goto语句，虽然作为C#语言基础的Java并没有包括goto语句。C#中的goto语句的一种合法使用是将它用于switch语句中，我们曾经在第8.2.2.2节中讨论过。

所有那些曾在8.4.3节中讨论过的循环退出语句，实际上都是经过装饰的goto语句。然而它们都是被严格限制的goto语句，并对可读性不产生危害。甚至还可以说，它们在事实上改进了可读性，因为如果不使用这些语句，会产生非常难于理解、缠绕不清且不自然的代码。

8.5 守卫的命令

Dijkstra提议了一些新颖且形式十分不同的选择结构和循环结构（Dijkstra，1975）。他的动机是要提供一种可以支持程序设计方法学的控制语句，使用这种方法学来确保开发期间的正确性，而不是依赖于对已完成程序的验证与测试来确保程序的正确性。Dijkstra对这种方法学进行了描述（Dijkstra，1976）。另一个开发守卫命令的动机是并发程序中有时需要的非决定论，第13章将其进行讨论。另一个动机是具有守卫命令可能引起的逐步增加的清晰性。简单地讲，守卫命令构造中的选择构造的可选段能被独立地认为构造的任何其他部分，这对常用程序设计语言的选择构造是不正确的。

我们在这一章里将概述守卫的命令，因为它们是为后来在CSP（Hoare，1978）和Ada这两种语言中的并发程序设计所开发的两种语言学机制的基础。关于Ada中的并发，将在第13章中讨论。它们也被用在Haskell中进行函数的定义，我们将在第15章中讨论这一点。 372

Dijkstra的选择构造具有以下形式：

```
if < 布尔表达式 > -> < 语句 >
[] < 布尔表达式 > -> < 语句 >
[] ---
[] < 布尔表达式 > -> < 语句 >
fi
```

终止保留字fi是起始保留字if的反向拼音。这种形式的终止保留字来自ALGOL 68。使用被称为fatbars的小模块来分离守卫的子句，并允许这些子句成为语句序列。在选择构造中的每

一行，都由布尔表达式（守卫）和语句或称为守卫命令的语句序列组成。

这种选择构造具有多向选择的外观，然而它的语义却完全不同。在执行期间，每一次执行到这种构造时，都要计算所有的这些布尔表达式。如果有一个以上的表达式为真，就要非确定地选择执行对应的语句之一。一种实现方法是总是选择与第一条为真的布尔表达式相关联的语句。因此，程序的正确性会取决于所选择的这一条语句（在这些为真的布尔表达式相关联的语句中）。如果所有的表达式都不为真，就会产生引起程序终止的运行时错误。这就迫使程序人员考虑并列出所有的可能性，如同Ada语言中的case语句那样。考虑下面的例子：

```
if i = 0 -> sum := sum + i
[] i > j -> sum := sum + j
[] j > i -> sum := sum + i
fi
```

如果i = 0且j > i，这种构造将在第一条与第三条赋值语句之间非确定地选择。如果i
373 等于j并且不为零，则出现运行时错误，因为所有的这些条件都不为真。

这种构造可能是允许程序人员说明执行次序的一种漂亮方式，然而在某些情况下却是不恰当的。例如，为了找出两个数字之中较大的一个，我们可以使用

```
if x >= y -> max := x
[] y >= x -> max := y
fi
```

这里进行的是计算所需要结果，但却并没有过分地说明怎样来计算。尤其是，如果当x与y相等时，我们将x与y中的哪一个赋给max都没有关系。这是由语句的非确定语义提供的一种抽象形式。

现在，考虑用传统程序设计语言选择器编码的同一过程：

```
if (x >= y)
  max = x;
else
  max = y;
```

这也能编码为：

```
if (x > y)
  max = x;
else
  max = y;
```

这两种构造并没有实际的差异，当x等于y时，第一种赋值x给max，在同样情形下，第二种赋值y给max，对这两种构造的选择复杂化了其代码和正确性证明的形式分析。这是一个为什么由Dijkstra开发守卫命令的原因。

要对守卫的命令的语义做出精确描述是十分困难的。虽然流程图并不是程序设计的理想工具，但有时对于语义的描述还会有些帮助。图8-1是一种描述Dijkstra所使用的选择器语句的流程图，注意，这种流程图稍欠精确。这也反映出在获取守卫的命令的语义方面存在的困难。

由Dijkstra提议的循环结构具有这种形式：

```
do < 布尔表达式 > -> < 语句 >
[] < 布尔表达式 > -> < 语句 >
[] ...
[] < 布尔表达式 > -> < 语句 >
od
```



图8-1 Dijkstra的选择语句所使用方式的流程图

这种构造的语义为，在每一次的重复中计算所有的布尔表达式。如果有一个以上的布尔表达式为真，其中相关联语句之一就被非确定地选择执行，此后，将再一次地计算所有的布尔表达式。当所有的布尔表达式都为假时，循环终止。 374

考虑下面的问题。4个变量q1、q2、q3和q4的值经过了重新安排，使得q1≤q2≤q3≤q4，没有守卫命令，一个直接的解决方案是，把4个值放入数组，对该数组排序，然后从数组中赋值回标量变量q1、q2、q3和q4。该解决方案不是困难的，它需要一些好的代码，特别是必须包含排序过程。现在，考虑下面的代码，它使守卫命令来解决同样的问题，但是采用了一种更精致和优雅的方法。⊖

```
do q1 > q2 -> temp := q1; q1 := q2; q2 := temp;
[] q2 > q3 -> temp := q2; q2 := q3; q3 := temp;
[] q3 > q4 -> temp := q3; q3 := q4; q4 := temp;
od
```

375

图8-2显示了描述Dijkstra的循环语句所使用方式的一种流程图。请再次注意，这种构造的控制流程的语义不能够完全由流程图来描述。

Dijkstra的守卫命令的构造是有趣的，部分原因是因为它们介绍了语句的语法和语义对于程序验证具有什么样的影响力，以及反过来的影响。当使用goto语句时，程序验证实际上是不可能的。当仅仅使用逻辑循环和选择时，或者，仅使用守卫的命令时，程序验证就被极大地简化。守卫的命令的公理语义已被很方便地说明（Gries，1981）。然而很明显，与对应的传统的确定方法相比，守卫命令在实现方面极大地增加了复杂性。 376

⊖ 这段代码以稍微不同的形式出现在文献中（Dijkstra，1975）。

8.6 结论

我们已经描述并讨论了多种语句层次上的控制结构，现在应该是进行简短的评估的时候了。

首先，我们有了理论上的结论，即对于表达计算绝对必要的只有顺序、选择及先测试逻辑循环（Böhm and Jacopini，1966），这个结论已经被希望全面禁止无条件分支的人们广泛接受与应用。当然，即使没有从理论上找到理由，goto语句也已经因为太多的实际问题而遭受谴责；一种对于goto语句的合法要求——从循环中提前退出，已经可以通过高度限制的分支语句（如break）而得到满足。

对于Böhm和Jacopini的结论的一种明显的误用，是反对任何除选择与先测试循环之外的控制结构的论点。没有任何一种广泛应用的语言采用了这种观点，并且，因为它在可写性与可读性上的负面影响，我们怀疑任何语言将不会采用这种观点。仅使用选择以及先测试循环来编写的程序，一般会在结构上不太自然，也更加复杂，因此写和读都比较困难。例如，C#中的多向选择结构极大地增进了C#的可写性，而没有明显的副作用，另一个例子则是许多语言中的计数循环结构，尤其是当这种语句的形式十分简单时，如Ada中的情形。

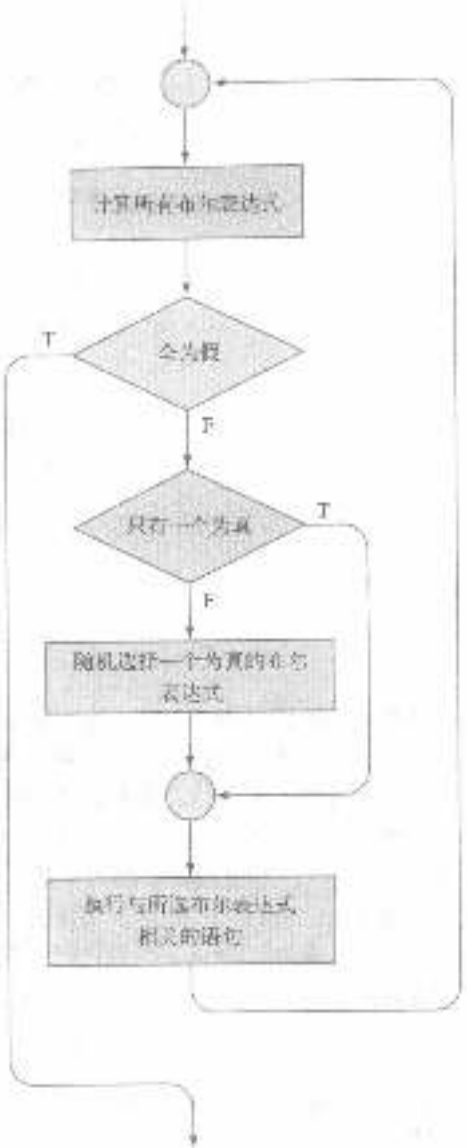


图8-2 Dijkstra的循环语句所使用方式的流程图

是否能将已被提议的许多其他控制结构的功能也包括在语言中还不太清楚（Ledgard and Marcotty，1975）。这个问题在很大程度上取决于更基本的问题，即是否必须尽可能地减小语言的规模。Wirth（1975）和Hoare（1973）都十分坚决地支持语言设计中的简单性，对于控制结构，简单性即意味着在一种语言中应该只有少量的控制语句，并且这些语句都应该是十分简单的。

人们设计了丰富多样的语句层次的控制结构，这反映了语言设计人员的不同观念。经过所有这些设计、讨论以及评估之后，对于究竟应该将哪些控制语句包括在语言中仍然没有一致的看法。当然，大多数当代语言确实具有类似的控制语句，但它们在语法和语义的细节上仍然具有不同。此外，是否应该在语言中包括goto语句，各种语言仍然不一致，C++和C#是包括了，但是Java则没有。

最后一点，函数式程序设计语言和逻辑程序设计语言中的控制结构都与在这一章中描述的控制结构十分不同。关于上述两种语言的机制，将分别在第15章和第16章中进行深入讨论。

小结

命令式语言的控制语句具有几种类型：选择、多向选择、循环以及无条件分支。

从于C的语言中的switch语句是多向选择语句的代表。C#中的switch语句不允许在执行了所选择的语句之后，接着执行下一条可选择的语句。这种方式就避免了C中的switch语句的可靠性问题。

从Fortran中的计数Do语句开始，人们为高级语言设计了许多不同的for循环语句。就复杂性而言，Ada的for语句很简单，它十分优雅地实现了最常用的计数循环形式。C的for语句是最具有灵活性的循环构造，虽然它的灵活性导致了一些可靠性问题。

基于C的语言都具有退出循环语句，这些语句取代了最常用的goto语句。

基于数据的迭代器是处理数据结构（如链表、数列及树结构）的循环构造。基于C的语言中的for语句允许用户为用户定义的数据结构建造迭代器。Perl和C#中的foreach语句是为标准数据结构预定义的迭代器。在当代面向对象的语言中，使用标准接口来说明集合的迭代器。而标准接口集合则由设计人员来完成。

无条件分支，或goto语句，是大多数命令式语言的组成部分。它所具有的问题已经被广泛地讨论并引发争论。当前达成的一致是应该将它保留在大多数的语言中。但对于它所具有的危险性，应该通过程序设计的规定将其减到最小程度。

Dijkstra的守卫的命令是具有正面理论特征的另一种控制结构。尽管它们还没有被任何一种语言实用为控制结构，但它的部分语义已经出现在CSP和Ada的并发机制中，还出现在Haskell的函数定义中。

复习题

1. 控制结构的定义是什么？
2. 块的定义是什么？
3. 选择结构的设计问题是什么？
4. 关于复杂语句的Python设计，什么是不常用的？
5. 对于双向选择器的嵌套问题，有哪些常见的解决办法？
6. 多向选择语句的设计问题是什么？
7. C的多向选择语句有什么是非同寻常的？在这种设计中有什么样的设计权衡？
8. 解释为什么C#中的switch语句比Java中的switch语句更安全？
9. 计数控制的循环语句有哪些设计问题？
10. 什么是先测试循环语句？什么是后测试循环语句？
11. C++中的for语句与Java中的for语句之间有什么不同？
12. 逻辑控制循环语句的设计问题是什么？
13. 开发用户定位的循环控制语句有哪些主要理由？
14. C#中的break语句比C中的break语句有哪些优越性？
15. C++中的break语句与Java中的break语句有什么不同？
16. 什么是用户定义的循环控制？
17. 哪种常用程序设计语言的设计部分地借鉴了Dijkstra的守卫的命令？

练习题

1. 描述二种给要计数循环与逻辑循环相结合的构造的情形。
2. 研究（Liskov et al., 1984）文献中CLU的迭代器的特性，并确定它的优点及缺点。
3. 将Ada中的控制语句系列与C#中的控制语句系列进行比较，决定它们中哪一个更好，并说明为什么。
4. 在复合语句中使用独特的终止保留字的利与弊各是什么？
5. 对于Python控制构造中指定复杂语句的缩进的使用，争论的利弊是什么？
6. 分析对控制语句使用终止保留字，潜在的可读性问题是什么。这些终止保留字是它对应的起始保留字

的反向拼法，如ALGOL 68中的保留字case与esac。例如，考虑常见打字输入错误，如将两个相邻的字符颠转。

7. 使用科学引证索引来查找一篇曾经引用过（Knuth，1974）的文章。阅读这篇文章并阅读Knuth的论文，写一篇文章来概述关于goto语句争论的两个方面。
8. 在一篇关于goto语句的争论的文章中，Knuth（1974）建议一种能允许多个出口的循环控制构造。阅读这篇文章并写出这种构造的操作语义描述。
9. 支持与反对在Java的控制语句中仅使用布尔表达式的论点各是什么（与在C和C++语言中允许算术表达式相反）？
10. 在Ada中，case构造的选择列表必须是穷举的，从而在控制表达式中将不会出现没有被表示的值。但在C++中，可以通过default选择器在运行时截住没有被表示的值。如果没有default选择器的话，一个没有被表示的值便可能会导致整个结构被跳越。这两种设计的优点与缺点各是什么（Ada与C++）？
11. 解释Java中的for语句具有什么优点与缺点？并比较Ada中的for语句。

程序设计练习题

1. 在指定的语言中，使用循环结构来重新编写下面的伪码程序段：

```
  k = (j + 13) / 27
loop:
  if k > 10 then goto out
  k = k + 1
  i = 3 * k - 1
  goto loop
out: ...
```

a. Fortran 95

b. Ada

c. C，C++，Java或者C#

d. Python

e. Ruby

假设所有的变量都为整数类型。讨论对于这段代码，哪一种语言具有最佳的可写性，最佳的可读性，或者两者的结合是最佳的。

2. 重新做程序设计练习题1，这次所有的变量和常量都为浮点数类型，并且将语句：

```
k = k + 1
```

改写为

```
k = k + 1.2
```

3. 使用下面列举的语言中的多向选择语句重新的写下面的代码段：

```
if ((k == 1) || (k == 2)) j = 2 * k - 1
if ((k == 3) || (k == 5)) j = 3 * k + 1
if (k == 4)  j = 4 * k - 2
if ((k == 6) || (k == 7) || (k == 8))  j = k - 2
```

a. Fortran 95（必须要穷举）

b. Ada

c. C，C++，Java或者C#

d. Python

e. Ruby

假设所有的变量都为整数类型。讨论对于这段代码，使用这些语言的相对优点。

4. 考虑下面的C程序段，重新编写这段程序，但不使用goto或者break语句。

```
j = -3;
for (i = 0; i < 3; i++) {
  switch (j + 2) {
    case 3:
    case 2: j--; break;
    case 0: j += 2; break;
    default: j = 0;
  }
  if (j > 0) break;
  j = 3 - i
}
```

5. Rubin在一封给CACM的编辑的信中（Rubin，1987），使用了下面的代码段来作为证据，试图证明某些具有goto语句的代码的可读性比没有goto语句的同等代码要好。这段代码从名为x的n×n的整数矩阵里找到它的第一个具有零值的行。

```
for (i = 1; i <= n; i++) {
  for (j = 1; j <= n; j++)
    if (x[i][j] != 0)
      goto reject;
  println ('First all-zero row is:', i);
  break;
reject:
 }
```

使用下面的语言重新编写这段代码，但不使用goto语句：C、C++、Java、C#或者Ada。

将你的代码的可读性与上面这段代码的可读性进行比较。

6. 考虑下面的程序设计问题：必须使用三个整数变量的值——first，second以及third来替代三个变量——max，mid以及min，并且不使用数组或用户定义的或预定义的子程序，而要使每个值具有明显的意义。针对这个问题，编写出两种解决方案，一种使用嵌套的选择结构，但另一种则不使用这种结构。 381
比较这两种方法的复杂性，以及这两种方法的可靠性。

7. 使用Ada语言编写下面的Java中的for结构：

```
int i, j, n = 100;
for (i = 0, j = 17; i < n; i++, j--)
  sum += i * j + 3;
```

8. 使用C语言中的if以及goto语句，重新编写程序设计练习题4中的C程序段。

9. 使用Java语言中的switch构造，重新编写程序设计练习题4中的C程序段。 382

第9章 子 程 序

子程序是程序的基本构件，并因此是程序设计语言设计的最重要概念之一。我们现在来研究关于子程序的设计，包括参数传递的方法、局部引用环境、重载子程序、通用子程序，还包括别名使用的问题，以及与子程序相关的一些副作用问题。我们也包括了对于提供对称的程序单位控制的协同程序的一些简短讨论。

关于子程序的实现方法，将在第10章讨论。

9.1 概述

一种程序设计语言可以包括两种基本的抽象设施：过程抽象与数据抽象。在高级程序设计语言的早期历史阶段，仅仅承认并包括了过程抽象。过程抽象是所有程序设计语言中的主要概念。然而从20世纪80年代开始，许多人相信数据抽象也同等重要。关于数据抽象，我们将在第11章里详细讨论。

第一台可编程的计算机是设计于20世纪40年代的Babbage的分析引擎（Analytical Engine），它具有在程序中的一些不同位置上重复使用一系列指令卡片的能力。而现代的程序设计语言是将这样的语句系列编写成为子程序。这种重复使用导致了在几个不同方面的节省，从存储空间到程序的编码时间。这样的重复使用也是一种抽象，因为它使用一条放置于程序之中的子程序调用语句代替了子程序的计算细节。不是在程序中解释如何进行计算，而只需要通过一条调用语句就启动了这种解释（即子程序中的语句系列），从而达到从各种细节中抽象出来的效果。通过展示程序的逻辑结构，而隐藏其底层的细节，这样就提高了程序的可读性。

面向对象语言中的方法与本章中讨论的子程序十分接近。方法与子程序存在不同的主要方面是对于它们的调用，以及它们与类及对象相关联的方式。虽然我们将在第12章讨论这些方法的特征，但是关于这些方法与子程序所具有的一些特性，如参数和局部变量，还是由本章来阐述。

9.2 子程序的基本原理

9.2.1 子程序的一般特征

本章讨论的所有子程序，除了在第9.11节里描述的协同程序以外，都具有下面的特征：

- 每一个子程序都只有一个入口。
- 在被调用子程序的执行期间，调用程序单位被停止执行。这意味着在任何给定时刻，只有一个子程序在执行。
- 当子程序的执行结束时，总是将控制返回到调用程序。

与上面的特征不同的子程序是协同程序（见第9.11节）和并发程序单元（第13章）。

虽然Fortran子程序可以具有多个入口，但这种特殊类型的入口相对地并不重要，因为它不提供任何不同的基本功能。因而在这一章里，我们将忽略Fortran子程序中的多个入口的可能性。

9.2.2 基本定义

子程序定义描述的是子程序的接口以及子程序的抽象行为。子程序调用是显式地要求执行子程序，如果在调用一个子程序之后，它被开始执行，但还没有完成这种执行，我们称这个子程序为活跃的。关于两种基本类型的子程序——过程与函数，将在第9.2.4节中给予定义并进行讨论。

定义中的第一个部分是**子程序首部**，它具有几个目的。第一，它说明下面的语法单位是关于某个特殊类型子程序的定义[○]。子程序的类型通常由特殊字来指定。第二，首部给子程序提供一个名称。第三，首部可以通过可选的方式来说明一列参数。

考虑下面的首部例子：

```
Subroutine Adder(parameters)
```

这是一个名为Adder的Fortran子程序的首部。在Ada中，这个子程序的首部将会是

```
procedure Adder(parameters)
```

Python子程序的首部有下面的形式：

```
def adder(parameters):
```

在除了Fortran和Ada的语言中，子程序的首部不会出现特殊字。这些语言只有一种子程序，即函数（也许还有方法），而函数的首部是由上下文来识别，而不是通过特殊字来识别的。例如，在C中

```
void adder(parameters)
```

这是名为adder的函数的首部，这里的void指示这个函数将不会返回任何数值。

因为带有复合语句，Python函数体的语句必须采用缩进风格。其结尾由未缩进的第一个语句指明（第一个语句接在函数定义之后）。使Python函数区别于其他常用程序设计语言的一个特征是，函数def语句是可执行的，当执行def语句时，它把给定的名字赋值给给定的函数体，直到执行完函数的def语句才能调用该函数。考虑下面框架的例子：

```
if ...
  def fun(...):
    ...
else
  def fun(...):
    ...
```

如果执行上述选择结构的then子句，则调用其子句中的fun函数，而不是else子句中的；反之，如果选择else子句，则调用else子句中的fun函数，而不是then子句中的。

与其他程序设计语言的子程序相比，Ruby函数存在着一些有趣的不同之处。Ruby方法通常在类定义时定义，但也可以在类定义外定义，此时，它们被认为是根对象object的方法。调用这样的方法不需要指对象接收器。就像是C或C++的函数一样。如果Ruby方法的return语句后没有跟表达式，则返回nil；如果后面跟了表达式，则返回该表达式的值；如果后跟多个表达式，则返回所有表达式值的数组。如果调用Ruby方法时不带接收器，则假定self。如果类中没有该名称的方法，则搜索封闭类，一直到Object（如果需要）。

子程序的**参数描绘**（有时也称为签名）是它所具有的形参的数目、次序以及类型。一个子

385

○ 一些程序设计语言包括了两种类型的子程序：即过程与函数。

程序的协议是这个子程序的参数描述加上它的返回类型，如果它是函数的话。在一些语言中，子程序具有类型，这些子程序的类型就通过子程序协议来定义。

子程序也可以具有声明和定义。这些声明和定义与C中变量的声明和定义很相似：可以使用这些声明来提供类型信息，但是并不能够被用来定义变量。在C中使用extern修饰词的变量
386 声明用来指定在函数中使用的变量在其他地方定义（不在使用的地方定义）。子程序的声明提供了程序的协议，但并不包括了程序体。不允许向前引用子程序是必要的，实施变量与子程序的静态类型检测都需要这些声明。对于子程序，则必须检测参数的类型。在C和C++程序中，子程序声明非常普遍，它们被称为原型（prototype）。

在大多数其他语言（除了C和C++）中，子程序不需要声明，因为这些语言并不要方法的定义必须实施于方法被调用以前。

9.2.3 参数

子程序通常描述计算。有两种方法让子程序能够获取它所要处理的数据：通过对非局部变量的直接访问（在别处被声明，但在子程序中可见），或者通过参数传递。经参数传递的数据是通过对子程序为局部的名称来存取的。参数传递比对非局部变量的直接访问更灵活。实质上，对将要处理的数据采用参数存取的子程序是一种所谓参数化的计算。这种计算能够在经参数接受的任何数据之上进行（假设这些参数的类型正是子程序期待的参数）。如果数据是通过非局部变量来存取的，可能在不同数据上进行计算的唯一方式是在子程序的调用之间将新的值赋给非局部变量。对于非局部变量的大量访问，将导致可靠性程度的降低。变量在需要对它们进行存取的子程序内可见，这是必要的；但它们常常在不需要被存取的地方也成为可见的。关于这个问题，我们曾经在第5章中讨论过。

虽然方法也通过非局部的引用和参数来存取外部的数据，但方法所处理的主要数据是实施方法调用的对象。然而，当一个方法确实存取非局部的数据时，所产生的可靠性问题就与非方法的子程序中一样。另外在一种面向对象的语言中，方法对于类变量（是与类相关联的变量，而非与对象相关联的变量）的存取将涉及到非局部数据的概念，应该尽可能地避免。在这种情况下，以及在C的程序进行非局部数据的存取的情况下，方法将可能具有改变一些非自身的参数或者一些非局部数据的副作用。这些改变将使得方法的语义复杂化，导致可靠程度的降低。

在某些情况下，如果能够将计算而不是数据作为子程序的参数来传送，将十分方便。此时，就可以将实现这种计算的子程序名作为参数来使用。关于参数的这种形式将在第9.6节中讨论。关于数据参数，则将在第9.5节中讨论。

子程序首部中的参数被称为形参，有时将形参认为是虚变量，因为它们不是平常意义上的变量；在大多数情况下，只有当子程序被调用时，它们才与存储空间相绑定，并且这种绑定通
387 常经过了一些其他的程序变量。

子程序的调用语句必须包括子程序的名称，以及一组将与子程序中的形参相绑定的参数，这些参数被称为实参。必须将实参与形参相区别，因为这两种参数在形式上有着不同的限制，它们的使用自然也相当不同。

几乎在所有的程序设计语言中，实参与形参之间的对应——或者是实参对形参的绑定——是简单地按位置进行的：第一个实参与第一个形参相绑定，依此类推。这样的一些参数被称为位置参数。只要参数表相对短，使用这种方法将实参与形参关联起来是一种十分有效且很安全的方法。

然而当参数表很长时，程序人员容易在表中实参的次序上犯错误。对这个问题的一种解决

办法是提供一些关键字参数。这种方法是：将一个与实参相绑定的形参的名称与这个实参在一起说明。关键字参数的优越性是它们能够以任何顺序出现于实参表中。Python中的过程就可以使用这种方法来调用，如

```
sumer(length = my_length,
      list = my_array,
      sum = my_sum)
```

在这里，Sumer的定义中具有形参Length，List以及Sum。

使用关键字参数的缺点是了程序的使用人员必须知道形参的名字。

除了关键字参数以外，Ada，Fortran 95和Python还允许位置参数，并且可以将这两种参数混合在一个调用之中，如

```
sumer(my_length,
      sum = my_sum,
      list = my_array)
```

这种形式的唯一限制是，在一个关键字参数出现在列表中以后，必须将所有的其余参数关键字化。之所以这是有必要的，因为在关键字参数出现之后，参数位置的顺序就可能不再遵循原来的定义。

在Python、Ruby、C++、Fortran 95、Ada以及PHP中，形参可以具有默认值。如果没有将实参传递给子程序首部中的形参，就可以使用形参的默认值。考虑下面的Python函数的首部：

```
def compute_pay(income, exemptions = 1, tax_rate)
```

参数Exemptions可以不出现在对Compute_Pay的调用中，但当它确实未出现时，就使用整数值1。在一个Python的调用中，对于没有出现的实参不使用逗号，因为这种逗号的唯一价值是指示下一个参数的位置，而这种指示在这种情况下就不必要了，因为在没有出现的实参之后的所有实参都必须被关键字化。例如，考虑下面的调用：

```
pay = compute_pay(20000.0, tax_rate = 0.15)
```

在不支持关键字参数的C++中，对于默认参数的规则必然是不同的。默认参数必须出现在最后面，因为参数与位置相关联。一旦一个默认参数在调用中被省略掉，所有其余的形参就都必须具有默认的值。可以将函数Compute_Pay的C++函数首部编写为：

```
float compute_pay(float income, float tax_rate,
                  int exemptions = 1)
```

注意，这里的参数已经被重新安排，以便将具有默认值的参数排在最后面。一个对C++中的compute_pay函数的调用例子为

```
pay = compute_pay(20000.0, 0.15);
```

在形参不具有默认值的大多数语言中，一个调用中的实参数目必须与子程序定义的首部中的形参数目相匹配，然而，在C、C++、Perl以及JavaScript中，则没有这项要求。当一个调用中的实参数目少于函数定义中的形参数目时，程序人员就有责任保证这些参数总在位置上相对应，并且保证子程序的执行是有意义的。

虽然这种允许参数数目不同的设计显然容易引起错误，但有时也较为方便。例如，C的printf函数就可以打印任何数目的项（数据值以及/或者字面量字符串）。

C#允许它的方法接受不同数目的参数，只要这些参数都具有相同的类型。C#中的方法使用params修饰符来说明形参。方法的调用程序可以传送一个数组或者一组表达式给方法，这些

388

数组或表达式的值由编译器放置在一个数组中，并且传递给被调用的方法。例如，考虑下面的方法：

```
public void DisplayList(params int[] list) {
   foreach (int next in list) {
      Console.WriteLine("Next value {0}", next);
   }
}
```

389

如果将DisplayList定义为类Myclass中的方法，而且我们有下面的声明：

```
Myclass myObject = new Myclass;
int[] myList = new int[6] {2, 4, 6, 8, 10, 12};
```

那么就可以使用下面的这两种方式来调用DisplayList：

```
myObject.DisplayList(myList);
myObject.DisplayList(2, 4, 3 * x - 1, 13);
```

Ruby支持复杂而又高度灵活的实参，初始化参数是表达式，表达式的值对象能被传递给相应的形参。初始化参数后能接一列关键的=>值对，它们放置在一个匿名的散列中，而且把对该散列的引用传递给下一个形参。这些用来替代Ruby不支持的关键字参数。散列项后能接一个由*开头的单一参数，这个参数称为**数组形参**。当调用方法时，数组形参设置为引用一个新的Array对象。所有余下的实参都被赋值给新的Array对象的元素。如果对应数组形参的实参是一个数组，那么它必须以一个*开始，而且它必须是最后的实参。[1]因此，Ruby支持与C#相同形式的可变数量的参数。因为Ruby数组能存储不同的类型，因此它对传递给数组的实参有相同类型没有需要。

下面例子简要勾勒出函数的定义，而对其的调用说明了Ruby的参数结构：

```
list = [2, 4, 6, 8]
def tester(p1, p2, p3, *p4)
  ...
end
...
tester('first', mon => 72, tue => 68, wed => 59, *list)
```

在tester内部，其形参值是：

```
p1 is 'first'
p2 is {mon => 72, tue => 68, wed => 59}
p3 is 2
p4 is [4, 6, 8]
```

390

Python支持与Ruby相似的参数。初始化形参在大多数常用语言中都是相似的。它们之后能紧跟一个常数数组（在Python中称为元组），该数组由以*开头的形参指定。调用子程序时变成数组的参数接收所有超出对应的初始化参数范围的非关键字实参。最后，由两个*开头的形参指定的最后的形参变成一个散列（在Python中称为字典）。与该参数对应的实参是关键的=值对，该值对放置在散列形参中。考虑下面梗要的例子函数和对其的调用：

```
def fun1(p1, p2, *p3, **p4):
  ...
```

[1] 并不完全正确，因为数组形参后能接一个以&开始的方法说函数。

```
...
fun1(2, 4, 6, 8, mon=68, tue=72, wed=77)
```

fun1的形参有下面的值：

```
p1 is 2
p2 is 4
p3 is [6, 8]
p4 is {'mon': 68, 'tue': 72, 'wed': 77}
```

9.2.4 Ruby块

在大多数其他程序设计语言中，处理数组或其他结构的数据通过迭代带有循环的数据结构并处理循环中的每个数据元素来完成。回想一下，Ruby包括其数据结构的迭代器方法。例如，数组结构有能用于处理任何数组的迭代器方法each。在Ruby中，它通过在对迭代器的调用上指定一块代码来完成。由花括号或**do-end**对界定的语句序列的代码块仅能出现在接下来的方法调用里。而且，它必须在同一行开始，作为最少的调用的最后部分。块能有在垂直条之间指定的形参。传递给被调用子程序的块是其本身，称为带有**yield**语句，该语句包含跟着实参的**yield**保留字。**yield**不能比有形参的块包括更多的实参。从块返回的值（给调用它的**yield**）是块中最后的表达式或值出来的值。

迭代器通常用来处理一个存在的数据结构中的数据。然后，当在迭代器方法中计算处理的数据时，它们也能使用到。考虑下面方法的简单例子和两个对其的调用，它们都包含了一个块 391

```
# A method to compute and yield Fibonacci numbers up to a
# limit
def fibonacci(last)
  first, second = 1, 1
  while first <= last
    yield first
    first, second = second, first + second
  end
end

# Call fibonacci with a block to display the numbers
puts "Fibonacci numbers less than 100 are:"
fibonacci(100) {|num| print num, " "}
puts  # Output a newline

# Call it again to sum the numbers and display the sum
sum = 0
fibonacci(100) {|num| sum += num}
puts "Sum of the Fibonacci numbers less than 100 is: #{sum}"
```

注意，在方法中，并列赋值的使用与Perl中很相似。也注意，在对puts的参数字符串中的注释#{...}用来指定把闭合表达式的值转换为一个字符串，并插入到字符字符串中。这段代码的输出如下：

```
Fibonacci numbers less than 100 are:
1 1 2 3 5 8 13 21 34 55 89
Sum of the Fibonacci numbers less than 100 is: 232
```

块是封闭的，这意味着它们保持它们定义时位置的环境，包括局部变量和当前对象，而不管它们在哪里调用。

9.2.5 过程与函数

存在着两种不同类型的子程序：过程与函数，可以将这两者都看作是语言的扩充方式。过程是定义参数化计算的语句系列，通过单个的调用语句来启动这些计算。过程实际上定义了新的语句，例如，因为Ada中不存在一条排序语句，用户就可以独立一个过程来对数据数组进行排序，并且用对这个过程的调用来代替缺乏的排序语句。在Ada中，将过程称为过程（procedures），
392 但在Fortran中则将过程称为子程序（subroutines）。

通过使用两种方法，过程可以在调用程序中产生结果。第一种，如果存在一些不是形参的变量，但这些变量在过程和调用程序单位中都是可见的，过程就可以改变这些变量。第二种，如果子程序具有允许将数据转移到调用程序的形参，这些形参也可以被改变。

函数在结构上模仿了过程，但在语义上却模拟自数学函数。如果一个函数是一个真正的（函数）模式，就不会产生副作用，也就是说，函数不会修改它的参数，也不会修改在函数之外定义的任何变量。事实上，程序中的许多函数都有副作用。

函数被调用时包括了它们在表达式中的名称，还同所需要的实参，而将函数执行产生的值返回给调用代码，这在效果上替代了调用本身。例如，表达式f(x)的值是当实用参数x调用f时所产生的任何值。一个不产生任何副作用的函数，它所返回的值就是它的唯一效果。

函数定义新的用户定义的操作符。例如，如果一种语言没有指数操作符，则可以编写一个函数来计算它的一个参数以另一个参数为指数的乘幂，并返回这个值。在C++中这个函数的首部就可以是

```
float power(float base, float exp)
```

它可以通过下面的语句来调用：

```
result = 3.4 * power(10.0, x)
```

C++的标准程序库已经包括了一个名为pow的类似函数。将这个函数与Perl中的相同操作进行比较，在Perl中，乘幂是语言中内置的操作：

391
```
result = 3.4 * 10.0 ** x
```

在Ada、Python、Ruby、C++和C#中，允许用户通过定义新的函数来使得操作符重载。在这些语言中，用户可以定义指数操作符，而使用起来就很像Perl中内置的乘幂操作符。关于用户定义的操作符重载，将在第9.10节中讨论。

一些程序设计语言，例如Fortran和Ada语言，同时提供了函数和过程。基于C的语言则只具有函数。然而，这些语言中函数的行为却与过程相类似。也可以定义这些函数不返回任何值，只要将它们的返回类型定义为void。因为在这些语言中的表达式也可以被用作语句，所以对void类型的函数的单独调用是合法的。例如，考虑下面的函数首部及其调用：

```
void sort(int list[], int listLen);
...
sort(scores, 100);
```

Java、C++以及C#中的方法与C中的函数相类似。

9.3 子程序的设计问题

子程序是程序设计语言中的复杂结构，并且随之而来的是在它们的设计中涉及广大量的问

题。其中的一个明显问题就是对参数传递方法的选择。各种语言所使用的各种不同方法反映了在这个问题上不同的意见。另一个与此紧密相关的问题是，是否应该按照与实参相对应的形参的类型来进行实参类型的类型检测。

子程序局部环境的性质在某种程度上表现了这个子程序的性质。这里最重要的问题是，局部变量是被静态地还是动态地实施分配。

下面的一个问题是，是否允许子程序定义的嵌套。还有一个问题是，是否可以将子程序名作为参数来传递。如果允许将子程序名作为参数来传递，而且这种语言同时允许子程序的嵌套，那么对于一个作为参数来传递的子程序，就又存在着一个正确引用环境的问题。

最后的一个问题是子程序是否可以重载或者通用化。重载的子程序是在同一种引用环境中，与另一个子程序同名的子程序。通用子程序则是指该子程序的计算可以通过不同的调用在不同的数据类型上进行。

下面是对这些子程序的设计问题的一个一般性总结。另一些特别与函数相关联的问题，将在第9.9节中进行讨论。

- 局部变量是静态地还是动态地被绑定？
- 子程序的定义可以出现在其他子程序的定义之中吗？
- 可以使用什么样的参数传递方法？
- 应该按照形参的类型来对实参类型进行检测吗？
- 如果可以将子程序作为参数传递，并且子程序可以被嵌套，这个被传递的子程序的引用环境将是什么？
- 子程序可以重载吗？
- 子程序可以通用化吗？

384

关于这些问题以及范例设计，将在下一节中讨论。

9.4 局部引用环境

本节讨论在子程序中定义的变量相关的问题，还简要涵盖了嵌套子程序定义的问题。

9.4.1 局部变量

子程序可以定义它们自己的变量，由此而定义局部引用环境。定义于子程序内部的变量被称为局部变量，因为这些变量的作用域通常就界定于定义它们的子程序之中。

在第5章中的术语里，局部变量可以为静态的，也可以为栈动态的。如果局部变量是栈动态的，当子程序开始执行之时，这些变量就与存储空间相绑定，并在执行终止之时解除这种绑定。栈动态的局部变量具有很多优点，其中的主要优点是它们为子程序提供了灵活性。递归子程序具有栈动态的局部变量。这是十分关键的。栈动态局部变量的另一个优点是，在活跃子程序中的局部变量的存储空间可以与所有在非活跃子程序中的局部变量共享。当然，现在这个优点已经不像当年计算机只有很小的存储空间时那么重要了。

栈动态局部变量的主要缺点如下：首先，对于子程序的每一次调用，这种变量所需要的存储空间分配、初始化（当必要时）以及变量解除分配，都有时间上的代价。其次，对于栈动态局部变量的存取必须是间接的；而对于静态变量的存取则可以是直接的。[8]这种

[8] 在一些实现中，静态变量也可以间接访问到，这样就消除这个缺点。

间接性之所以必要，是因为只有在执行期间才能确定一个局部变量在栈中的位置（见第10章）。在大多数计算机上，间接寻址比直接寻址要慢。最后一点，具有栈动态局部变量的子程序不是历史敏感的，也就是说，它们不能够在调用之间保持局部变量的数据值。能够编写历史敏感的子程序有时是很方便的。需要历史敏感的子程序的一个例子是一个产生伪随机数的子程序。对这种子程序的每一次调用都将基于这个子程序所产生的最后一个伪随机数，再由此计算新的伪随机数，因此必须在一个静态局部变量里储存这个最后的随机数。协同程序以及用于迭代循环结构的子程序（曾经在第8章讨论过）是需要历史敏感的子程序的另一些例子。

较之栈动态局部变量，静态局部变量所具有的主要优点是它们的高效率——因为没有间接性，它们通常可以被很快地存取。此外，它们的分配与解除分配也没有运行时的额外代价。当然还有一点，它们允许历史敏感的子程序。静态局部变量的最大缺点是不具有支持
395 递归运算的能力。另外，静态局部变量的存储空间不能够与其他非活跃子程序中的局部变量共享。

在大多数的当代语言中，都将子程序中的局部变量默认为栈动态的。在C和C++的函数中，局部变量一般都是栈动态的，除了那些被特别声明为static（静态）的变量除外。例如，在下面的C（或者C++）函数中，变量sum是静态的，而变量count则是栈动态的。

```
int adder(int list[], int listlen) {
  static int sum = 0;
  int count;
  for (count = 0; count < listlen; count ++)
    sum += list [count];
  return sum;
}
```

Ada中的子程序仅具有栈动态局部变量，C++，Java以及C#中的方法也仅仅具有栈动态局部变量。

如在第5章中的讨论，Fortran 95的实现人员可以将局部变量选择为静态的还是栈动态的。实际上，因为Fortran 90之前的Fortran版本不允许递归运算，所以没有理由一定要将局部变量选择为栈动态的。一般人们认为，为了节省存储空间而损失效率是不值得的。不论在哪种实现中，Fortran 95的用户都可以通过将变量名列于一条Save语句中，从而将一个或多个局部变量强制为静态的。

在Fortran 95中，可以显式地将一个子程序说明为递归的，而且在这种情形下，这个子程序的局部变量则是栈动态的。这种说明递归调用的子程序的思想最初来自于PL/I语言。显式说明的目的是为了允许使用一种更高效的方式来实现非递归子程序。下面是一个递归子程序框架的例子：

```
Recursive Subroutine Sub ()
  Integer :: Count
  Save, Real :: Sum
  ...
End Subroutine Sub
```

在这个子程序中，因为将这个子程序定义为是递归的，因而变量Count是栈动态的，变量Sum是静态的，因为它被标记了Save。

在Python中，用在方法定义中的仅有的声明是全局的。在方法中任何声明为全局的变量必须是在方法外面定义的变量。在方法外面定义的变量不需要在方法内声明它为全局就能引用它。如果在一个方法中对全局变量赋值，该变量隐式声明为局部的，赋值操作也会影响全局。所有Python方法的局部变量都是栈动态的。

396

9.4.2 嵌套子程序

嵌套子程序的思想起源于Algol 60，其动机是能创建一个逻辑和作用域层次。如果一个子程序只在另一个子程序内需要，为什么不把它放在那里，并以程序的剩余部分隐藏它？因为静态作用域常常用在允许子程序嵌套的语言中，这在闭合子程序中也为访问非局部变量提供了一种高度结构化的方法。回顾一下，这里引入的问题在第5章中的讨论。长期以来，只有允许嵌套子程序的语言是从Algol 60衍生而来，包括Algol 68、Pascal和Ada。许多其他语言，包括所有C的直接衍生语言，都不允许子程序嵌套。近来，一些新语言又开始允许它，其中有JavaScript、Python和Ruby。

9.5 参数传递方法

参数传递方法是将参数传送到或者取自被调用子程序的方式。首先，我们将注意力集中于参数传递方法的主要语义模式上，然后讨论语言的设计人员为这些语义模式所开发的各种实现模式；接下来再审视各种命令式语言的设计选择，并且讨论用于实现这些实现模式的实际方法；最后我们将考虑语言设计人员在选择这些方法时所面临的一种设计考虑。

9.5.1 参数传递的语义模式

可以使用三种不同的语义模式来刻画形参：(1) 它们能够接受来自与其相对应的实参的数据；(2) 它们能够将数据传递给这些实参；(3) 它们能够施行这两种任务。我们将这三种语义模式分别称为**输入型**、**输出型**以及**输入输出型**。例如，考虑一个子程序，该子程序取int类型的两个数组来作为参数，它们分别是list1和list2。这个子程序必须将list1与list2相加，并且将返回的计算结果作为list2的新版本——即替换了原来的list2。另外，该子程序还必须使用这两个已给的数组（进行某种计算）产生出一个新的数组，并且返回这个新的数组。对于这个子程序，数组list1应该为输入型的，因为它将不会被子程序改变，而数组list2则必须是输入输出型的，因为子程序将需要这个数组的初始值，而且还必须给它返回它的新值。这里的第三个数组就应该是输出型的，因为它不具有初始值，而且还必须将计算的新值返回给调用程序。

397

关于在参数的传递中怎样进行数据的传输，有两种概念模式：将实际的值复制（到调用程序，到被调用程序，或者到这两者），或者通过一条存取途径来传输，最普通的方法是，存取途径就是一个简单的指针或者一个简单的引用。图9-1说明当使用值的复制方式时，参数传递的三种语义模式。

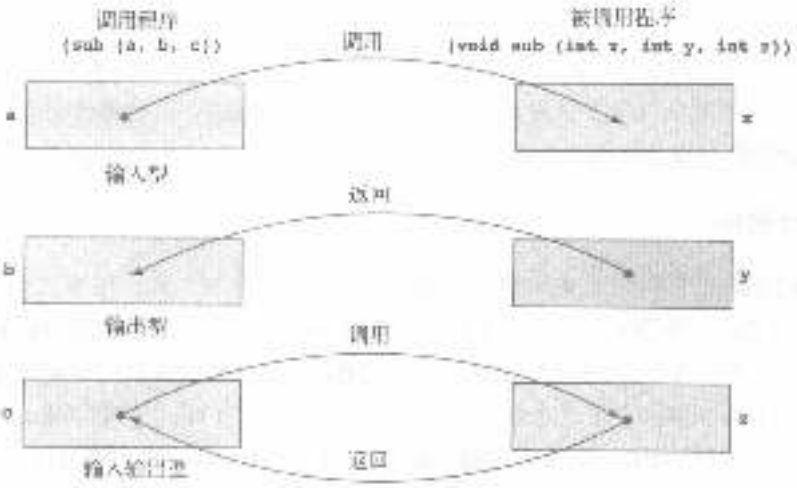


图9-1 当使用数值物理移动的方式时，参数传递的三种语义模式

访谈

398~399



第二部分：脚本语言的一般性与Perl语言的特殊性

LARRY WALL

Larry Wall经历丰富，他涉足于书籍出版、语言出版、软件出版以及儿童教育（他有四个孩子）。他曾在Seattle Pacific University（西雅图太平洋大学）、Berkeley（伯克利）以及UCLA（加州大学洛杉矶分校）学习过。他还曾经在Unisys公司、Jet Propulsion Laboratories以及Seagate公司工作过。给他带来最大名气的是语言出版（"却只有最少的金钱回报"Larry补充道）：Larry是Perl脚本语言的作者。

脚本语言的一些特点

问：关于脚本语言的定义是什么？

答：人们经常在争论脚本语言与程序设计语言之间的差别。如果你问我这个问题，我会告诉你：脚本是给演员的，而程序是给观众的。我的意思是：脚本是正在进过作者、导演和演员而逐渐成形的东西，而程序则是相对固定的一系列预先决定的事件。

在脚本与程序之间并没有截然的差别。脚本也可以演变为程序。如果使用我们的比喻：当你正在编写一个程序，它就像是一个脚本，而当你完成了一个脚本，它就更像是一个程序了。

有一些语言能够让你比较容易即兴地写出一段东西：应该将这些语言归为脚本语言。而另外的一些语言，则能够让你比较容易地对数据结构进行精确而又高效的处理。当然，你必须首先说明很多东西。应该将这些语言归为程序设计语言。人们通常将Perl归类于脚本语言，但这只是一种过于简单化的分类，或者称为云复杂化的分类。

问：你认为读者应该怎样理解脚本语言？

答：最重要的事情是：当人们常常打算将一个样机程序扔掉时，他们事实上并没有这么做，他们本来只是打算使用这个代码一次，却因为它"足够好"而将它留下来，导致这些代码在软件生产中使用的时间比当初的期望长得多。

关于Perl语言的更多方面以及Perl的发展历程

问：Perl与其他脚本语言有什么区别？

答：我认为一种好的脚本语言应该能够允许你将你的脚本进化为"真正"的程序。Perl就能够提供这个方面的全面支持。换言之，Perl是一种能够随着你的需求而不断"长大"的语言。

脚本就像是通往高速公路的加速道。大多数的语言只是试图提供加速道而没有高速公路；或者是只提供高速公路而没有加速道。Perl则提供这两者。这就是为什么Perl的口号是："条条道路通罗马"。Perl并不给你强加任何特定的方式。有一些问题最好是使用管道和模式匹配技术来解决；而其他的一些问题则最好是使用函数式程序设计、事件驱动的程序设计或者面向对象的程序设计来解决。

与此相反，其他的一些语言企图将你套在一种特定的思维方式中，或者是使得与外部的交互因难重重。Perl并不是想要成为一种完美的语言，它只是要成为一种有用的、能够方便地与其他语言和系统合作的语言。自然语言不是乌托邦，它们不禁固思想，计算机语言也不应该禁固人的思想。

问：当你创建Perl语言时，你想像过它能够成为今天的这个样子吗？是什么使得Perl成为网络上有用的工具？市场是非常重要的。在你看来，工业界和计算机界做了些什么，使得Perl腾飞起来？

答：是的。我知道Perl会很大，但没有料到它会这么大。最初，我发现在文本处理方面，Perl会将awk和sed淘汰；在与任意的接口连接方面，会令shell程序设计成为过时的东西。一旦我将所有的系统管理操作增加进去，我相信大多数的UNIX系统管理人员将会选择Perl作为程序设计语言。这是不足为怪的。

真正使我吃惊的是，万维网以及很大部分的万维网程序会使用Perl来编写。HTTP是一种基于文本的协议，万维网服务器需要某种粘合代码来进行文本与末端数据库之间的翻译转换。Perl正好是在正确的时刻出现在正确的位置，具有正确功能的语言。

这恰恰印证了另一个事实：一个好的工具被用于它的创造者都没有想象到的地方。然而这并不完全仅仅是一种意外。一个优秀的创造人员会将优良的功能加入到一种产品中。当机会出现时，这些功能就派上了用场。

问：你最喜欢Perl语言中的哪个部分？

答：我最喜欢的是亲手修改我所最不喜欢的东西。认真地说，我最喜欢的是保留在Perl 6中的部分。Perl总是十分有利于演化，以及在Perl环境中能够与其他程序及语言很好地合作。在这些方面，Perl只能会继续变得更好。Perl在快速样机设计以及文本处理方面将继续保持优秀。它还将具有更好的模式匹配功能（所有的这些，都是假设我们能够完成Perl 6的话……）。

问：作为结束语，如果你不是在进行你目前所做的这一切，你会怎样打发你的时间？

答：很可能会让我周围的人疯狂……

9.5.2 参数传递的实现模式

语言的设计人员开发了各种模式，来指导如何实现三种基本的参数传递模式。在下面的几节里，我们将讨论其中的几种模式，并且评估它们各自的相对长处与弱点。

9.5.2.1 按值传递

当参数是按值传递时，实参的值将被用来为与其相对应的形参设定初值，然后这个形参的行为就像是子程序中的局部变量，并由此实现了输入型的语义。

通常将按值传递实现为数据的复制，因为采用这种方法通常具有较高的存取效率。也可以通过传输一条通往调用程序中实参的值的存取途径来实现，但这种方式要求数值是在一个受写入保护的单位（即只读单位）中，强制写入保护并不总是一件简单的事情。例如，设想一个接受被传递参数的子程序，而它又将这个参数传递给另一个子程序。这就正是使用复制传递的另一个理由。我们将在9.5.4节中看到，C++提供了一种方便而有效的方法，对于以存取路径方式传递的按值传递的参数进行强制写入保护。

按值传递的优点是对于标量来说，它是快速的，链接开销小和存储时间。

使用数据复制方式的按值传递方法的主要缺点是需要给形参以额外的存储空间，或者是在调用程序中，或者是在调用程序以及被调用子程序以外的某个区域中。除此之外，还必须将实参复制到与之相对应的形参的存储区域中。如果一个参数很大的话（例如包含很多元素的数组），

这些存储空间以及这种复制操作可能就具有很高的代价。

9.5.2.2 按结果传递

按结果传递是用于输出型参数的一种实现模式。当一个参数被按结果传递时，并没有将值传递到子程序。相对应的形参的行为就如同一个局部变量，但是在将控制返回到调用程序之前，形参值被传递给调用程序的实参；显然，这个实参必须是一个变量。（如果这个实参是一个字面常量或者一个表达式，调用程序怎么能够引用计算结果呢？）

按结果传递方法有按值传递的优点和缺点，再加上一些额外的缺点。如果这些值如同平常那样是经复制返回（而不是经存取路径），按结果传递也会需要额外的存储空间和复制操作。就如同按值传递一样，通过传输存取途径来实现按结果传递，同样是十分困难的，这也如同在按值传递时的情形一样；返而这也就导致通常将按结果传递实现为数据复制。在这种情况下，要确保被调用的子程序不能够使用实参的初始值。

按结果传递模式中的一个额外的问题是可能会出现实参冲突，如下面的这个调用：

```
sub(p1, p1);
```

假设在sub中，这两个形参具有不同的名字，那么显然可以给这两个形参赋以不同的值，而无论将其中的哪一个最后复制给它们对应的实参，都将成为p1的值，返而实参复制的次序就决定了它们的值。例如，考虑下面的C#方法，它指定了在其形参上带有out指示符的按结果传递方法。[⊖]

401

```
void Fixer(out int x, out int y) {
  x = 17;
  y = 35;
}
...
f.Fixer(out a, out a);
```

如果在Fixer执行的末尾，形参x首先赋值给其对应的实参，在调用函数中实参a的值将是35。如果首先赋值y，调用函数中实参a的值为17。

因为次序有时依赖于具体语言的实现，不同的实现能产生不同的结果。

第9.5.2.4节将要讨论到，当使用其他的参数传递方法，在调用一个过程时使用两个相同的实参也可能引起不同类型的问题。

使用按结果传递的方式可能出现的另一个问题是实现人员可以选择在两个不同时刻对实参的地址求值：即在调用时，或在返回时。例如，考虑下面C#方法和下面的代码：

```
void DoIt(out int x, int index){
  x = 17;
  index = 42;
}
...
sub = 21;
f.DoIt(list[sub], sub);
```

list[sub]的地址在方法开始和结束之间变化。实现者必须在调用时或返回时选择决定的返回值地址的时间。如果在方法入口处计算地址，值17将返回给list[21]；如果正好在返回前计算，17将返回给list[42]。这使程序在子程序开始时为出模式参数选择求值地址和结束时选择求值地址的实现之间是不可移植的。

⊖ out指示符也必须在对应的实参上指定。

9.5.2.3 按值与结果传递

按值与结果传递是对输入输出型参数的一种实现模式，在这种模式中，实际的值被复制。它在效果上是按值传递和按结果传递的结合。使用实参值来为其对应的形参设定初值，此后，这个形参即具有局部变量的行为。事实上，按值与结果传递中的形参必须具有与被调用子程序相关联的局部存储空间。在子程序终止时，形参值被传递回实参。 402

有时，按值与结果传递被称为按复制传递，因为实参在子程序的入口被复制给形参，然后在子程序终止时又被复制回来。

按值与结果传递具有与按值传递以及按结果传递这两者共同的缺点：需要给参数提供额外的存储空间以及数值复制的时间。它也与按结果传递具有共同的实参赋值次序问题。

按值与结果传递的优点是与按引用传递相关的，因此它们将在第9.5.2节中讨论。

9.5.2.4 按引用传递

按引用传递是对输入输出型参数的第二种实现模式。按引用传递的方式不像按值与结果传递那样，将数据值来回地复制，按引用传递的方式是给被调用子程序传递一条存取途径，通常就是一个地址。这给被调用子程序提供了对实参存储单位的存取途径，因此允许被调用子程序从调用程序单位中存取实参。在实际效果上，被调用子程序就共享了这个实参。

按引用传递方式的优点是，这种传递过程的本身在时间与空间两方面都具有高效率。不需要复份的存储空间，也不需要任何复制的过程。

然而，按引用传递方法也具有几种缺点。第一，对形参的存取速度将较按值传递的方式慢，因为需要额外层次的间接寻址。[○]第二，如果只要求对被调用子程序的单向传递，可能会在实参上产生一些不易察觉但是错误的改变。

按引用传递的另一个严重问题是可能产生别名使用。因为按引用传递使得被调用子程序可以使用存取途径，并由此扩展它们对非局部变量的访问，所以别名使用应该是在预料之中。当按引用传递参数时，有几种产生别名的方式。在其他环境中，这些别名使用的问题是相同的：它对可读性和可靠性是有害的。它也使程序验证变得极端困难。

现在我们讨论一些按引用传递参数能创建别名的方法。首先，在实参之间可能产生一些冲突。考虑一个C++中的函数，它具有两个将按引用传递的参数（参见9.5.3节），如

```
void fun(int &first, int &second)
```

如果对fun的调用恰巧两次都传递了同一个变量，如

```
fun(total, total)
```

那么，在fun中的first与second将会是别名。

其次，在数组元素之间的冲突也会引起别名。例如，设想函数fun是通过由变量下标说明的两个数组元素被调用，如

```
fun(list[i], list[j])
```

如果这两个参数是按引用传递的，并且i恰好就等于j，那么，first和second又一次成为了别名。

最后，如果一个子程序的两个形参，一个为数组元素，而另一个为整个数组，而这两个参数都是按引用传递的，那么下面这样的一个调用：

```
fun1(list[i], list)
```

○ 这将在第9.5.3节中深入讨论。

就可以导致函数fun1中的别名使用。因为fun1可以通过它的第二个参数来访问list中的所有元素，通过它的第一个参数来访问一个单一元素。

运用按引用传递的参数还有另一种引起别名使用的方式，即形参与可见的非局部变量之间的冲突。例如，考虑下面的C代码：

```
int * global;
void main() {
   ...
   sub(global);
   ...
}
void sub(int * param) {
   ...
}
```

403

在sub中，param和global则为别名。

如果是使用按值与结果传递，而不是按引用传递的话，所有这些可能的别名使用问题都不会存在。然而，当没有别名使用问题时，有时又会出现如9.5.2.3节中讨论的其他一些问题。

9.5.2.5 按名传递

按名传递是一种输入输出型参数的传递方法，它不对应于单个的实现模式。当参数是按名传递时，对于子程序中的所有情形，实参实际上都以文本形式替代了与它相对应的形参。这与迄今为止讨论过的方法相当不同。在前面的情况下，形参在子程序调用时被绑定于实际的值或者地址。而一个按名传递的形参，是在子程序调用时被绑定于一种存取方法，而这个形参对于值或者对于地址的实际绑定，则被推迟到对形参赋值或者形参被引用以后。

404

因为按名传递方法不是任何广泛应用的语言中的部分，所以我们不会进一步对它进行讨论。然而，这种方法却被实用于Algol编语言的编译时，以及C++和Ada中的通用子程序的通用参数，9.8节将要讨论这些内容。

9.5.3 实现参数传递的方法

我们现在来讨论如何实际地实现参数传递的各种实现模式。

在大多数当代语言中，参数间的交流是通过运行时栈而发生的。对于运行时栈进行初始化和维护，都是由管理程序执行的系统程序，即运行时系统来执行的。正如我们将在第10章讨论的，在子程序的控制链接过程以及子程序的参数传递过程中，都极大量地使用了运行时栈。在下面的讨论中，我们假定所有的参数传递过程都使用运行时栈。

按值传递的参数，它们的值被复制到栈中的位置。栈中的这个位置接着就成为相对应的形参的存储空间。实现按结果传递参数方式正好与按值传递参数方式相反。赋给按结果传递中的实参的值将被放置到栈里，在被调用子程序结束以前，调用程序都可以查询这些实参的值。按值与结果传递的参数可以根据它们的语义直接实现，它们是按值传递方式与按结果传递方式的结合。由调用程序来实施参数栈空间的初始化，然后，它们的使用就如同被调用子程序中的局部变量。

按引用传递的参数也许是实现起来最简单的一种方式。不论实参的类型如何，仅仅必须将实参的地址放入栈空间。对于字面常量，是将字面常量地址放入栈空间。而如果是表达式，编译器则必须在控制转移到被调用子程序之前，产生出表达式求值的代码，而将存放这个代码求值结果的存储单位的地址放入栈中。编译器还必须确保被调用的子程序不会改变参数，而无论

这些参数是字面常量还是表达式。存取被调用子程序中的形参是通过栈中的地址来间接寻址。图9-2显示了使用运行时栈来实现的按值传递、按结果传递、按值与结果传递以及按引用传递参数的方式。从主程序main中调用了程序sub，调用形式为sub(w, x, y, z)。其中的参数w是按值传递的，x是按结果传递的，y是按值与结果传递的，而z是按引用传递的。

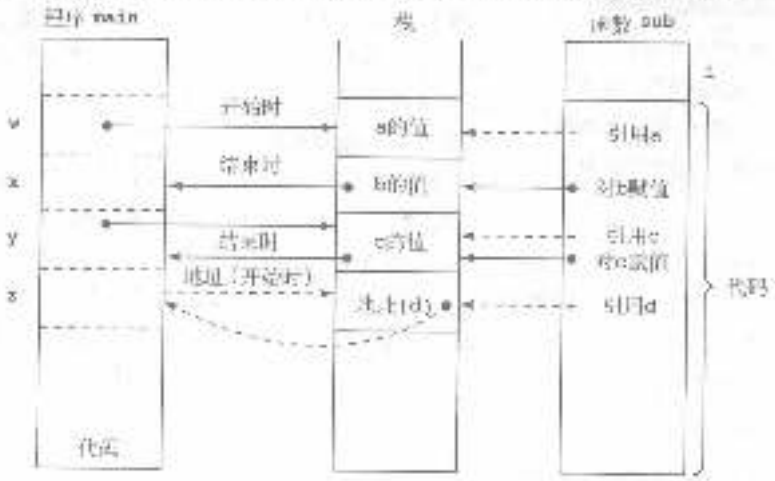


图9-2 几种常见参数传递方法的一种可能的实现

如果是使用按引用传递以及按值与结果传递参数的方式，但在实现这些方式时没有特别小心，将产生一个微妙但却致命的错误。假设，一个程序包括了两个对常量10的引用，第一个是作为对子程序进行调用中的实参。如果进一步地考虑，子程序错误地改变了与10相对应的形参的值，将10改变成5。正如编译器通常所做的那样，程序的编译器可能会在编译时为数值10建立单个的存储位置，并且将这个位置用于程序中所有对数值10的引用。但是在子程序返回之后，所有后续的对于数值10的引用实际上都成为了对数值5的引用。如果允许这种情形发生的话，就会产生一个很难诊断的程序设计问题。这种情形在Fortran IV的许多实现之中的确发生过。 405

9.5.4 主要语言中的参数传递方法

C使用指针来作为参数，进而满足了按值传递以及按引用传递（输入输出型）的语义。将指针的值传给被调用函数，但并不返回任何数值。然而，因为所传递的是对于调用函数数据的存取途径，所以被调用函数就可以改变调用函数中的数据。C语言是从ALGOL 68复制了这种按值传递的方法。在C与C++中，形参可以为指向常量的指针类型，它所对应的实参不必是常量。因为在这种情况下会将这些实参强制为常量。这样，指针参数被允许通过使用单向按值传递的语义来提供按引用传递的效率，在被调用函数中参数的写保护是强式的。

历史注释

ALGOL 60引入了按名传递的方式，它也将按值传递作为可选择的方式。主要由于在实现方面的困难，参数的按名传递没有被后来进行的ALGOL 60的修订后继语言所继承，除了SIMULA67之外。

第6章中讨论过，C++包括了一个特殊指针类型，称为引用类型，它常常被用于参数。引用参数是在函数中或者在方法中被隐式间接引用，它们的语义则是按引用传递。C++也允许将引用参数定义为常量。例如，我们可以有 406

```
void fun(const int &p1, int p2, int &p3) { ... }
```

在这里，p1是按引用传递的，但是不能够在函数fun中被改变，参数p2是按值传递的，而p3是按引用传递的，在函数fun中，p1以及p3都不需要被显式间接引用。

历史注释

ALGOL W（Wirth and Hoare，1966）引入参数的按值与结果传递方法，来替代低效率的按名传递方式，以及存在问题的按引用传递方式。

常量参数与输入型参数并不完全一样，十分显然的是，常量参数实现了输入型参数。然而，在除了Ada之外的所有常用的命令式语言中，可以在子程序中给输入型参数赋值，虽然这些改变从来没有反映在与其对应的实参值之上，常量参数则不能够被赋值。

如同在C和C++中一样，Java中的所有参数都是按值传递的。然而，因为对象只能够通过引用变量来存取，对象参数实际上只能够按引用传递。同样，因为引用变量不能够直接指向标量变量，加上Java中不存在指针，因而在Java中，标量不能够按引用传递（一个包含标量的对象却能够按引用传递）。

Ada的设计人员定义了参数传递的三种语义模式版本：输入型、输出型以及输入输出型。这三种模式分别用保留字in、out和in out来表示，其中的in为默认方法。例如，考虑下面的Ada子程序的首部：

```
procedure Adder(A : in out Integer;
                B : in Integer;
                C : out Float)
```

在Ada中被声明为out模式的形参可以被赋值，但是却不可以引用。in模式的参数则可以引用，但是不可以被赋值。自然，在Ada中in out模式的参数既可以被引用又可以被赋值。

在Ada 95中，所有标量都是通过复制来传递，而所有的结构参数则是通过引用来传递。

Fortran 95十分类似于Ada，通过使用它的Intent属性，可以将它的形参声明为输入、输出或者输入输出型。例如，考虑下面Fortran 95子程序的首部：

407

```
Subroutine Adder(A, B, C)
   Integer, Intent(Inout) :: A
   Integer, Intent(In) :: B
   Integer, Intent(Out) :: C
```

这个子程序中参数的语义模式与前面的Ada子程序中相同。

C#中默认的参数传递方法是按值传递，可以通过特殊的说明来使用按引用传递：需要在形参以及对应的实参前面都放置说明符ref。例如，考虑下面C#中的方法和调用框架：

```
void sumer(ref int oldSum, int newOne) { ... }
...
sumer(ref sum, newValue);
```

传递给sumer的第一个参数是按引用传递的，而第二个参数则是按值传递的。

C#还支持输出型的参数，这是一些按引用传递的不需要初始值的参数。这类参数是用out修饰符在形参表中说明。

C#中的方法可以获取不同数目的参数，只要这些参数具有相同的类型。在方法中只定义一个参数，即一个具有修饰符params的数组。例如，

```
void SumInts(params int [] intValues) { ... }
```

这个方法可以被一个数组或者一列整数表达式来调用。例如，

```
int [] myIntArray = new int[6] {2, 4, 6, 8, 10, 12};
```

```
sum1 = SumInts(myIntArray);
sum2 = SumInts(10, i, 17, k);
```

PHP中的参数传递类似于C#中的参数传递，但PHP可以将形参或者实参的传递说明为按引用传递，这种说明是通过在两个参数中的一种或者两种之前放置&符号。

Perl语言采用了传递参数的一种原始形式。它将所有（所有一切！）的实参都隐式地放置在一个名为@_的预定义数组中，这个数组可以装载任何类型的数据。子程序从这个数组中获取实参的值（或者地址）。这个数组最特殊的方面是它所具有的神奇特性，即它的元素事实上就是实参的别名。因而，如果被调用的子程序改变了@_中的一个元素，这种改变将反映到调用中的实参之上。在这里，我们假设这个子程序调用具有相应的实参（实参的数目与形参的数目并不一定相同），而且这个实参是一个变量。

Python和Ruby的参数传递方式称为按赋值传递。由于所有的数据值都是对象，所以每个变量都是对对象的引用。在按赋值传递中，实参值赋值给形参。因此，因为所有实参值都是引用，所以按赋值传递实际上是按引用传递。然而，这只会在某些情况下产生按引用传递参数传递语义。例如，许多对象基本上是不变的。在纯面向对象的语言中，改变赋值语句中变量值的过程，如下

```
x = x + 1
```

并不改变x引用的对象。而采用这种方式：它在x引用的对象上加1，从而创建一个新的对象（具有x+1的值），然后改变x以引用新的对象。因此，当把对标量对象的引用传递给子程序时，引用的对象不能在适当的地方改变，因为引用是按值传递的，所以尽管形参在子程序中改变，这个改变对调用者的实参没有影响。

现在，假设对数组的引用作为参数传递。如果把对应的形参赋值给新的数组对象，那么它对调用者没有影响。然而，如果形参用于赋值给数组元素，如下

```
list[3] = 47
```

实参就改变了。因此，虽然改变形参的引用对调用者没有影响，但是改变作为参数传递的数组元素却会对调用者产生影响。

9.5.5 参数类型检测

现在人们广泛地接受了这样一种观念，即软件可靠性要求进行实参的类型检测，以便与对应的形参类型相一致。如果没有这种类型检测，小的打字错误就可能导致难以诊断的程序错误，因为编译器或者运行时系统不能够发现这种错误。例如，在下面的函数调用中：

```
result = sub1(1)
```

实参是一个整数常量，如果sub1的形参为一个浮点数类型，而又不进行参数的类型检测的话，就不能够发现错误。尽管整数1与浮点数1具有相同的数值，但它们的表示却非常不同。当sub1期望的是浮点数值，而实际给予的是一个整数实参时，sub1就不能产生正确的结果。

早期的程序设计语言，如Fortran 77以及C的原始版本，不要求参数的类型检测，大多数后来的语言都要求进行参数的类型检测。然而，更近代的语言，如Perl、JavaScript以及PHP，反而又没有要求。

C和C++在参数类型检测的问题上需要进行一些特殊的讨论。早期的C无论是对参数的数目还是类型都不进行检测。在C89中，函数的形参可以用两种方式来定义。第一种方式与早期的C中的定义相同，也就是将参数名列在括号之内，而参数的类型声明跟随在后。如

```
double sin(x)
  double x;
  { ... }
```

使用这种方法避免了类型检测，因而如下的调用：

```
double value;
int count;
...
value = sin(count);
```

是合法的，尽管它并不总正确的。

另一种方法是被称为原型（prototype）的方法，这种方法将形参类型也包括进参数表，如

```
double sin(double x)
 { ... }
```

使用与上面相同的函数调用来调用sin的这个版本，即

```
value = sin(count);
```

这也是合法的。对照形参类型（double），对实参的类型（int）进行检测。虽然它们并不匹配，但是可以将int类型强制转换为double类型（宽化强制转换），从此完成类型的转换。如果这种转换不可能（例如，如果实参是一个数组），或者参数的数目是错误的，则会检查出一个语法错误。因而在C89中，用户可以选择是否需要进行类型的检测。

在C99以及C++中，所有的函数所具有的形参都必须在它的原型形式中，然而，通过使用省略号来替代参数表中的最后部分，就可以避免对某些参数的类型检测，如下面所示：

410

```
int printf(const char* format_string, ...);
```

一个对于printf的调用至少必须包括一个参数，这个参数是一个指向常量字符串的指针。除此以外，任何东西（或者什么也没有）都是合法的。printf用来确定是否存在额外参数的方式是通过字符串参数中所出现的特殊符号，例如，一个整数输出的格式代码为%d。它作为字符串部分出现，如下面所示

```
printf("The sum is %d\n", sum);
```

这里的符号%通知函数printf后面还有一个参数。

当基本类型能按引用传递时（如在C#中），有一个更有趣的关于实参向形参强制转换的问题，假设对一个方法的引用传递float值给double型形参。如果该参数按值传递，那么float值会强制转换为double值，而没有任何问题。特定的强制转换是非常有用的，因为它允许库提供两种版本的子程序（float型和double型）。然而，假设参数是按引用传递的。当double型实参的值返回给调用者的float型实参时，该值将发生溢出。为了避免这种问题的产生，C#需要精确匹配对应形参的ref实参的类型（不允许强制转换）。

Python和Ruby没有参数类型检查，原因是在这些语言中的类型化是一个不同的概念，对象有类型，但变量却没有，因此形参是没有类型的。类型检查参数的思想在这里不适用。

9.5.6 多维数组作为参数

我们曾经在第6章用较长篇幅讨论过将多维数组元素引用中的下标值映射到存储空间地址的存储映射函数。在一些语言中，如C和C++，当将一个多维数组作为参数来传递给一个子程序时，编译器在只看见了程序的文本时（而不是看见调用程序时）必须能够为这个数组建立映射函数。

之所以会如此，是因为子程序与调用它们的程序可以被分开编译。考虑在C中将一个矩阵传递给一个函数的问题，在C中，多维数组实际上是数组的数组，并且是以按行存放的方法存储。下面是一个按行存放矩阵的映射函数，这个矩阵下限的下标都为0，并且元素的大小是1：

```
address(mat[i, j]) = address(mat[0,0]) + i * number_of_columns + j
```

411

注意，这个映射函数需要矩阵的列数，但不需要行数。因而在C和C++中，当将一个矩阵作为参数来传递时，形参必须将列数包括在它的第二对括号中。这在下面的C程序框架中进行了说明：

```
void fun(int matrix[][10]) {
  ... }
void main() {
  int mat[5][10];
  ...
  fun(mat);
  ...
}
```

将矩阵作为参数来传递的问题是：不允许编程人员编写一个函数，它可以接受不同列数的矩阵；而对于每一个具有不同列数的矩阵，都需要编写一个新的函数。这在实际上是不允许编写可以有效地重复使用的灵活函数来用于处理多维数组。在C和C++中，因为包括了指针算术，因而存在一种方式来绕过这种问题：可以将矩阵作为指针来传递，并将矩阵的实际维数作为参数包括进来。然后当每一次引用矩阵中的一个元素时，这个函数可以使用指针算术来计算用户编写的存储映射函数。例如，考虑下面的函数原型：

```
void fun(float *mat_ptr,
         int num_rows, int num_cols);
```

可以使用下面的语句将变量x的值赋给fun中的参数矩阵的元素[row][col]：

```
*(mat_ptr + (row * num_cols) +
    col) = x;
```

虽然能够完成这种计算，但显然很难以阅读，并且因为所具有的复杂性，十分容易产生错误。可以通过使用一条宏指令定义存储映射函数，以减轻阅读方面的困难，如

```
#define mat_ptr(r,c)
         (*mat_ptr + ((r) * (num_cols) + (c)))
```

使用这一条定义，上面的赋值语句就可以写成为

```
mat_ptr(row,col) = x;
```

其他的语言以不同方式来处理多维数组传递的问题。Ada的编译器能够在编译子程序时，确定所有作为参数使用的数组的定义维长。在Ada中，不受限制的数组类型可以为形参。不受限制的数组类型是在数组类型的定义中没有给出下标范围。但在不受限制数组类型的变量定义中则必须包括下标范围。传递一个不受限制数组的子程序代码，可以获得与这个参数相关的实参的下标范围信息。例如，考虑下面的定义：

历史注释

Ada 83语言的定义说明，标量（以结构的）参数是按复制传递的。也就是输入型和输入输出型的参数是局部变量，它们通过复制对应的实参值而被初始化。当子程序终止时，输出型或者输入输出型的简单参数值被复制回相对应的实参之上。当然在着多次复制时，复制的次序不是由语言的定义来指定的。输出型和输入输出型的参数来优先在控制移动到被调用子程序之间发生的。例如，假设一个输出型的实参具有下面的形式：

历史注释

Ext[index]

在调用时将计算这个参数的地址值。如果list的"index"给巧在被调用子程序中为可见的，并且这个子程序对它还进行了修改，然而这个参数地址将不会受到影响。

当形参是数组或者记录时，Ada 83的实现人员可以在按值与结果传递与按引用传递之间进行选择。如果Ada 83的设计人员没有说明传递结构化参数的实现方法，将可能产生一种微妙的问题。这个问题就是，这两种实现方法可能会导致某些程序产生不同的程序结果。之所以会出现这种不同的结果，是因为按引用传递的方法提供了对调用程序中一个存储单元的访问，但如果实参为全局可见的，就提供对这个存储单元的另一种访问，这样就产生了别名。如果是使用按值与结果传递来代替按引用传递，这种对实参的双重访问则是不可能的。

另一个问题如下所述：假如子程序非正常地结束（由异常引起），在按值与结果传递实现中的实参将不会被改变，而在按引用传递的实现中，则可能甚至在错误出现之前就已经改变了相对应的实参，再一次地说明，这两种实现方法之间可能存在着不同。

如果一个Ada 83程序根据实现输入输出方法的不同将产生不同的结果，这个程序就被称为有误的(erroneous)。编译器不能够检测出这种错误。通常是当用户将程序从一种实现转移到另一种实现，并且说到它将产生不同结果时，才能够发现这种错误。对于这种问题，Ada 83设计的哲学是，编程人员必须确保能够防止别名的出现；如果他们创建了别名，他们必须与这种潜在的问题进行"斗争"。

412

```
type Mat_Type is array (Integer range <>,
                        Integer range <>) of
                          Float;
Mat_1 : Mat_Type(1..100, 1..20);
```

一个函数将返回Mat_Type类型的数组中元素的总和，这个函数的形式如下：

```
function Sumer(Mat : in Mat_Type) return Float
is
  Sum : Float := 0.0;
  begin
  for Row in Mat'range(1) loop
    for Col in Mat'range(2) loop
      Sum := Sum + Mat(Row, Col);
    end loop;  -- 结束列的循环 --
  end loop;  -- 结束行的循环 --
  return Sum;
  end Sumer;
```

这里的属性range返回给定下标的实参数组的下标范围，因此不论参数的下标范围大小如何，总是能够完成这项任务。

在Fortran中，这种问题是以下面的方式来解决的。必须在函数首部之后声明数组类型的形参。对于一维数组，声明中的下标是无关紧要的；但是对于多维数组，这种声明中的下标能够使编译器产生存储映射函数。考虑下面示例的Fortran子程序的框架：

```
Subroutine Sub(Matrix, Rows, Cols, Result)
  Integer, Intent(In) :: Rows, Cols
  Real, Dimension(Rows, Cols), Intent(In) :: Matrix
  Real, Intent(In) :: Result
  ...
End Subroutine Sub
```

只要实参Rows具有被传递矩阵的定义中行数的值，这个子程序就能够工作。如果在目前，被传递数组中并没有将所定义的范围都装满有用的数据，那么，可以将所定义的数组下标范围以及已经填充部分的下标范围都传递给子程序。然后，这种定义的范围被用于数组的局部声明中，而已经填充部分的下标范围则被用来控制数组元素引用的计算。例如，考虑下面的Fortran子程序：

```
Subroutine Matsum(Matrix, Rows, Cols, Filled_Rows,
     Filled_Cols, Sum)
  Real, Dimension(Rows, Cols), Intent(In) :: Matrix
  Integer, Intent(In) :: Rows, Cols, Filled_Rows,
                         Filled_Cols
  Real, Intent(Out) :: Sum
  Integer :: Row_Index, Col_Index
  Sum = 0.0
  Do Row_Index = 1, Filled_Rows
```

```
    In Col_Index = 1, Filled_Cols
      Sum = Sum + Matrix(Row_Index, Col_Index)
    End In
  End In
  End Subroutine Matsum
```

413

Java和C#使用一种与Ada类似的技术，将多维数组作为参数来传递。在Java和C#中数组是对象，对象都是一维的。但是这些对象的元素也可以是数组。每一个数组都继承了一个命名常量（在Java中为length，在C#中为Length）；当创建数组对象时，将命名常量length或Length设置为数组的维长。矩阵的形参与两对空的方括号同时出现，如同下面Java中的方法所示，这个方法与前面的Ada示例函数Sumer具有相同的功能：

```
float sumer(float mat[][]) {
  float sum = 0.0f;
  for (int row = 0; row < mat.length; row++) {
    for (int col = 0; col < mat[row].length; col++) {
      sum += mat[row][col];
    } //** for (int row ...
  } //** for (int col ...
  return sum;
}
```

414

当为每一个数组都具有自己的长度值，所以，矩阵中的行可以具有不同的长度。

9.5.7 设计考虑

在选择参数传递的方法时，应该考虑涉及的两个重要方面：效率以及是否需要进行单向还是双向的数据传送。

当代软件工程的原则明确指出，应该将通过子程序代码来存取了程序之外数据的可能性减到最小。基于这种原则，只要是不需要通过参数将数据返回调用程序时，就应该使用输入型的参数，而当不需要将数据传递到被调用子程序，但子程序必须返回数据到调用程序时，就应该使用输出型的参数。最后，只有当数据必须在调用子程序与被调用子程序之间双向传递时，才使用输入输出型参数。

一种实际原因引起与上述原则的冲突。有时，进行单向的参数传输时，传递一条存取路径是合适的。例如，要将一个大的数组传递给一个并不修改数组的子程序时，选用单向方法可能更合适一些。如果是按值传递的话，需要将整个数组传送到子程序的局部存储区域，这在时间与空间两个方面都是高代价的。因为这个原因，对大的数组通常是按引用传递。这恰恰是为什么在Ada 83语言定义中允许实现人员为结构化参数在两种方法之间进行选择的原因，C++中的常量引用参数提供了另一种解决方案。还有一种方法允许用户在这些方法中进行选择。

关于函数的参数传递方法的选择，则与另一个设计问题相关：即函数的副作用。这个问题将在9.9节中进行讨论。

9.5.8 参数传递的例子

考虑下面的C函数：

```
void swap1(int a, int b) {
  int temp = a;
  a = b;
  b = temp;
}
```

假设使用下面的语句来调用这个函数：

```
swap1(c, d);
```

415 前面讲过C是使用按值传递的，可以使用下面的伪码来描述swap1的行为：

```
a = c          - 将入第一个参数的值
b = d          - 将入第二个参数的值
temp = a
a = b
b = temp
```

虽然a被后具有d的值，b被后具有c的值，但是因为没有数据返回到调用程序，所以c和d的值没有发生变化。

我们可以修改C的交换函数，以使能够使用指针参数达到按引用传递的效果：

```
void swap2(int *a, int *b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}
```

可以使用下面的语句来调用swap2：

```
swap2(&c, &d);
```

可以以下面的形式来描述swap2的行为：

```
a = &c  - 将入第一个参数的地址
b = &d  - 将入第二个参数的地址
temp = *a
*a = *b
*b = temp
```

在这种情况下交换操作是成功的，事实上，值c与d相互交换了。

还可以使用C++中的引用参数来编写swap2，如下所示：

```
void swap2(int &a, int &b) {
  int temp = a;
  a = b;
  b = temp;
}
```

这种简单的交换操作在Java中却是不可能的，因为Java中既没有指针，也没有C++中的这种引用。Java中的引用变量只能够指向对象，而不能够指向标量值。

按值与结果传递的语义与按引用传递的相同，除非是涉及别名使用的情形，前面说过。在
416 Ada中对输入输出型的标量参数使用的是按值与结果传递。为了更深入地探究按值与结果传递的方式，考虑下面的函数swap3，我们假设这个函数使用按值与结果传递的参数。这个函数是使用一种类似于Ada语言的语法来编写的。

```
procedure swap3(a : in out Integer, b : in out Integer) is
  temp : Integer;
  begin
  temp := a;
  a := b;
  b := temp;
  end swap3;
```

假设可以使用下面的语句来调用swap3：

```
swap3(c, d);
```

在这种调用之下，swap3的行为是：

```
addr_c = &c          — 移入第一个参数的地址
addr_d = &d          — 移入第二个参数的地址
a = *addr_c          — 移入第一个参数的值
b = *addr_d          — 移入第二个参数的值
temp = a
a = b
b = temp
*addr_c = a          — 移出第一个参数的值
*addr_d = b          — 移出第二个参数的值
```

因而，这个交换子程序的操作也是正确的。下面来考虑这个调用

```
swap3(i, list[i]);
```

在这种情况下，它的行为是：

```
addr_i = &i                — 移入第一个参数的地址
addr_listi = &list[i]      — 移入第二个参数的地址
a = *addr_i                — 移入第一个参数的值
b = *addr_listi            — 移入第二个参数的值
temp = a
a = b
b = temp
*addr_i = a                — 移出第一个参数的值
*addr_listi = b            — 移出第二个参数的值
```

其次，这个子程序的操作也是正确的，因为在这种情形下对返回参数值的地址的计算进行于函数调用之时，而不是返回时。如果是在返回时才计算实参的地址，就会产生错误的结果。 417

最后，我们必须研究当使用按值与结果传递以及按引用传递时，如果涉及了别名使用，将发生什么情况。考虑下面用类似C的语法编写的程序框架：

```
int i = 3;  /* i 是一个全局变量 */
void fun(int a, int b) {
  i = b;
}
void main() {
  int list[10];
  list[i] = 5;
  fun(i, list[i]);
}
```

如果在函数fun中使用按引用传递，则i与a是别名。如果是使用按值与结果传递，i与a则不是别名。假设是按值与结果传递，那么fun的行为是： 418

```
addr_i = &i                — 移入第一个参数的地址
addr_listi = &list[i]      — 移入第二个参数的地址
a = *addr_i                — 移入第一个参数的值
b = *addr_listi            — 移入第二个参数的值
i = b                      — 将i设置为b
*addr_i = a                — 移出第一个参数的值
*addr_listi = b            — 移出第二个参数的值
```

在这种情况下，在函数fun中对全局变量i赋值，将会把它的值从3变成5。但是从第一个形参复制回来时（上面行为说明中的倒数第二行），又将它设置回3。这里的一个重要发现就是，如果是使用按引用传递，结果导致所复制回来的值并不是语义中的部分，并且i仍将保持为5。另外也请注意，因为是在函数fun开始时计算第二个参数的地址，全局变量i的任何改变都不会影响到在程序的尾端用于返回list[i]值的地址。

9.6 子程序名作为参数

如果可以将子程序的名称作为参数传递给其他子程序的话，就可以极为方便地处理程序设计中产生的许多情形。其中一种最常见的情形出现于，当子程序必须进行一些数学函数的抽样时。例如，一个数值积分子程序在估算一个函数剖形下面的面积时，它通过对函数在许多不同的点抽取样值来进行这项工作。所编写的这个子程序应该可以被用于任何给出的函数，而不应该对每一个需要积分的函数都重新编写程序。因而十分自然地，应该可以将需要进行积分计算的程序函数名称作为参数传递给积分子程序。

虽然这种思想十分自然，而且看上去也相当简单，但是其中如何工作的细节却使人困惑。如果只是需要传递子程序的代码，则可以通过传递单个指针来实现；然而，却有两种复杂的情形。

首先，作为参数传递的那个子程序的启动参数有类型检测的问题。不能够将C和C++中的函数作为参数来传递，但是指向函数的指针却可以，一个指向函数的指针的类型就是这个函数的协议。因为协议之中包括了所有的参数类型，完全可以对这些参数进行类型检测。Fortran 95具有一种机制来对作为参数传递的子程序提供参数类型的检测，而且必须对这些参数进行检测。Ada不允许将子程序作为参数来传递。Ada所提供的通用功能代替了将子程序作为参数传递的功能，关于这些，我们将在9.8节中进行讨论。

第二个带有子程序作为参数的应用只出现在允许嵌套子程序的语言中，这个问题即是，执行被传递的子程序时，应该使用什么样的引用环境。这里存在以下三种选择：

1. 启动被传递子程序的调用语句的环境（浅绑定）。
2. 被传递子程序的定义环境（深绑定）。
3. 将子程序作为实参传递的调用语句的环境（特别绑定）。

历史注释

Pascal中的原始定义（Jensen and Wirth，1974）允许将为参数传递的子程序不包括参数类型的信息，如果默认编译是可能的（在初始Pascal中没有包括独立编译），编译则甚至无法不检测参数的数目正确与否。在没有独立编译的情况下，检测参数的一致性是可能的，但如果一个相当复杂的工作，因而通常并不进行。Fortran 77中也有着同样的问题，但是因为Fortran 77从来也不检测参数类型的一致性，因而这并不是一个额外的问题。

下面用JavaScript语法编写的示例程序对这些选择进行了解释。

```
function sub1() {
  var x;
  function sub2() {
    alert(x);  // 对x的值产生一个对话框
    };
  function sub3() {
    var x;
    x = 3;
    sub4(sub2);
    };
  function sub4(subx) {
    var x;
    x = 4;
    subx();
    };
  x = 1;
```

```
sub3());
}
```

考虑当在sub4中调用sub2时sub2的执行情况。对于浅绑定，sub2的执行引用环境就是sub4的执行引用环境。因而将在sub2中对x的引用绑定于sub4中的局部变量x，而且程序的输出为x=4。对于深绑定，sub2的执行引用环境就是sub1的引用环境，因而将在sub2中对x的引用绑定于sub1中的局部变量x，而且程序的输出为x=1。对于特别绑定，这种绑定是将sub2中对于x的引用绑定到sub3中局部变量x之上，并且其输出为x=3。

在一些情况下，一个声明了程序的子程序也可以将被声明的子程序作为参数来传递。在这种情况下，深绑定与特别绑定是相同的。特别绑定从来没有被使用过，因为人们可能会猜测，作为参数的程序所出现的环境与被传递的子程序并没有自然的联系。

浅绑定不适合于具有嵌套子程序的静态作用域语言。例如，假设程序Sender将程序Sent作为参数传递给程序Receiver。这里的问题是，Receiver可能不在Sent的静态环境之中，这使得Sent存取Receiver中的变量非常不自然。另一方面，对于任何一种静态作用域语言中的子程序，包括被作为参数传递的子程序，通过子程序定义中的词法位置来确定它的引用环境是完全正常的。所以这些语言使用深绑定更合乎逻辑，一些动态作用域的语言则使用浅绑定。

9.7 重载子程序

重载的操作符是具有多种意义的操作符。重载操作符的其一个特定实例由它的操作数类型决定。例如，如果在一个Java程序中的*操作符具有两个浮点操作数，这个操作符所说明的就是浮点数乘法。但是如果同样的*操作符具有两个整数操作数，则说明的是整数乘法。

重载子程序是与另一个在相同引用环境中的子程序具有相同名称的子程序。每一个重载子程序的版本必须只具有一个协议；也就是说，它必须与子程序的其他版本在参数的数目，参数的顺序或者参数的类型上不相同；如果它是一个函数的话，则是返回的类型不相同。通过实参表来决定（或者，如果子程序是函数的话，则由返回值的类型来决定）重载了程序调用的意义。尽管它是不必要的，但是重载子程序通常实现同样的过程。

C++，Java，Ada以及C#都包括了预定义的重载子程序。例如，在C++，Java和C#中的类都具有重载的结构。这为重载子程序的每一个版本各具有一种独特的参数特征，所以编译器就可以通过这些不同类型的参数来区分对不同版本的调用。不幸的是，它没有那么简单。受允许的参数强制转换大大地增加了消除歧义过程的复杂度。简单地说，问题是，如果没有一种方法的参数列表与方法调用中实参的数目和类型相匹配，但是两个或更多方法有通过强制转换匹配的参数列表，编译器调用哪种方法呢？对于致力于解决这一问题的语言设计人员来说，他（她）必须决定如何将强制转换分级，以致于编译器能选择方法以最好地与调用相匹配。这将会成为一个可怕的、繁重的任务。为了弄明白该情形的复杂性，读者可以参考在C++中使用的方法调用非歧义性规则（Stroustrup，1997）。 420

在Ada中，是通过一个重载函数返回的类型来区分调用的是哪一个版本。因此，两个重载函数可以具有同样的参数特征，只是它们返回的类型不同。因为Ada不允许混合模式表达式，所以函数调用的上下文就能够说明从函数返回的类型。例如，如果一个Ada程序具有两个名称都为Fun的函数，对这两个函数都输入一个Integer类型的参数，然而，一个函数返回一个Integer值，而另一个则返回一个Float值，下面的调用将是合法的：

```
A, B : Integer;
...
```

```
A := 3 + Fun(7);
```

在这段代码中，将对函数Fun的调用绑定于返回Integer值的版本，因为如果选择返回Float值的版本，将会产生类型错误。

因为C++，Java以及C#都允许混合模式的表达式，当而不能够用返回类型来区分重载函数（或者重载方法），也不能够根据调用的上下文来确定返回的类型。例如，如果一个C++程序具有两个名称都为fun的函数，对这两个函数都输入一个int类型的参数。然而，一个函数返回一个int值，而另一个则返回一个float值，这种程序将不会被编译，因为编译器不能够确定应该使用fun的哪一个版本。

在Ada，Java，C++以及C#中，还允许用户使用同样的名字来编写子程序的多个版本。而且，在C++，Java和C#中，最常见的用户定义的重载方法是构造函数。

具有默认参数的重载子程序可能会导致具有歧义的子程序调用。例如，考虑下面的C++ 421
代码：

```
void fun(float b = 0.0);
void fun();
...
fun();
```

这个调用是歧义的，它会引起编译错误。

9.8 通用子程序

软件的复用是软件生产力增长的一个重要促进因素。增加软件复用性的一种方式是减少产生那些在不同数据类型上实现相同算法的不同子程序。例如，一个编程人员不必给四个只是元素类型不同的数组编写四个不同的排序子程序。

多态子程序在输入不同类型的参数时，将启动不同的处理。重载子程序提供一种特殊类型的多态，称为**特别多态**。重载子程序的行为不需要相似。

Python和Ruby的方法提供了一种更通用的多态性。前面提到过，这些语言的变量没有类型，因此形参没有类型。然而，只要定义了方法内使用在形参上的操作符，该方法在任何实参类型情况下都能正常使用。

参数多态性由一个带有通用参数的子程序提供，这种通用参数被用于描述了程序参数类型的类型表达式之中。为这种类型的子程序的不同实现给定不同的通用参数，将会产生带有不同类型参数的子程序。子程序的参数定义都表现为相同的行为。参数多态子程序还常称为**通用子程序**。Ada和C++都提供一种编译时参数多态性。Java 5.0也支持参数多态性，但是这种支持方式与Ada和C++中的支持方式是十分不同的。

9.8.1 Ada中的通用子程序

Ada通过一个结构来提供参数多态性，这个结构支持构造程序单位的多个版本，以接受不同数据类型的参数。子程序的不同版本是由编译器根据来自用户程序的请求进行实例化或者构造的。因为所有这些子程序版本全都具有相同的名字，这就造成了一种错觉，以为单个子程序 422
可以在不同的调用之中处理不同类型的数据。因为这种程序单位在性质上是通用的，所以有时将它们称为**通用单位**。

同样的机制也可以被用来允许子程序的不同执行来调用一个通用子程序的不同实例。这在提供作为参数传递的子程序的功能方面是十分有用的。

下面的例子介绍了一个具有三个通用参数的程序：它允许将一个通用数组作为参数传递给了程序。这是一个交换排序的程序，被设计用来处理任何具有基本数值类型元素的数组，而且这个数组使用任何序数类型的下标：

```
generic
  type Index_Type is (<>);
  type Element_Type is private;
  type Vector is array (Integer range <>) of
       Element_Type;
  procedure Generic_Sort(List : in out Vector);
  procedure Generic_Sort(List : in out Vector) is
    Temp : Element_Type;
    begin
    for Top in List'First..Index_Type'Pred(List'Last) loop
      for Bottom in Index_Type'Succ(Top)..List'Last loop
        if List(Top) > List(Bottom) then
          Temp := List(Top);
          List(Top) := List(Bottom);
          List(Bottom) := Temp;
          end if;
      end loop; - for Bottom ...
    end loop; - for Top ...
  end Generic_Sort;
```

如果你对Ada不是很熟悉，这个通用程序中的某些部分可能显得十分奇怪。然而，我们并不需要了解这个过程的所有语法细节，它并不重要。这里的数组类型以及数组元素的类型是这个程序中的两个通用参数。数组被声明为具有任意范围内的任意类型的下标（即，可以被合法地作为下标的任何类型）。

这个通用的排序仅仅是一个过程的模板，编译器并不需要为它产生代码，并且它对于程序没有影响，除非它为某一种类型进行了实例化。这种实例化是用类似下面的语句来完成的：

```
procedure Integer_Sort is new Generic_Sort(
                                  Index_Type => Integer;
                                  Element_Type => Integer;
                                  Vector => Int_Array);
```

编译器对这条语句的反应是建立Generic_Sort的一个名为Integer_Sort的版本，这个版本所排序的数组类型为Int_Array，而这个数组的元素类型为Integer，数组下标的类型也为Integer。

在这样写成的Generic_Sort中，假设操作符 ">" 是对于将要被排序的数组元素而定义的。通过在Generic_Sort的通用参数中包括一个比较函数，就能够提高Generic_Sort的通用性。

前面说过，Ada不允许将子程序作为参数传递给其他子程序，但Ada可使用通用形式的子程序来提供这种功能。在类似Fortran的语言中，将子程序作为参数来传递，这样对一个子程序的某一特殊调用就可以使用被特殊传递的子程序来计算结果。在Ada中，通过允许用户将一个通用子程序进行任意次数的实例化，每一次都产生一个可以使用的不同子程序来获得相同的结果。例如，考虑下面的Ada范用程序：

```
generic
  with function Fun(X : Float) return Float;
  procedure Integrate (Lowerbd : in Float;
                       Upperbd : in Float;
```

```
                        Result : out Float);
  procedure Integrate (Lowerbd : in Float;
                       Upperbd : in Float;
                       Result : out Float) is
    Funval : Float;
    begin
    ...
    Funval := Fun(Lowerbd);
    ...
    end Integrate;
```

我们可以使用下面的语句，为用户定义的函数Fun1对上面的程序施行实例化。

```
procedure Integrate_Fun1 is new Integrate(Fun => Fun1);
```

现在，Integrate_Fun1是对函数Fun1进行积分的程序。

9.8.2 C++中的通用函数

C++中的通用函数具有模板函数的描述性名称。一个模板函数定义的一般形式为：

```
模板 <模板参数>
—这是一个可能包括了模板参数的函数定义
```

模板参数（必须至少有一个）具有下面形式中的一种：

```
类 标识符
类型名称 标识符
```

这里使用类的形式作为类型名称，而类型名称形式被用于传递数值给模板函数。例如，传递模板函数中的数组的维长整数值有时是十分方便的。

一个模板能够把另一个模板作为参数。实际上，模板类经常定义用户定义通用类型。但我们不在这里考虑这种选择。[1]

作为一个例子，考虑下面的模板函数：

```
template <class Type>
Type max(Type first, Type second) {
  return first > second ? first : second;
}
```

这里的Type是一个参数，它说明函数将要操作的数据类型。可以将这个模板函数对任何由操作符 > 定义的类型进行实例化。例如，如果将这个模板函数以int来实施参数实例化，它将成为

```
int max(int first, int second) {
  return first > second ? first : second;
}
```

虽然可以将这一过程定义成为一个宏指令，但是，如果这些参数是具有副作用的表达式，这种宏指令有时候就不能够正确地操作。例如，假设将宏指令定义为

```
#define max(a, b) ((a) > (b)) ? (a) : (b)
```

这一条宏指令是适用的，就它可以工作于任意数值类型而言。然而，如果它的调用参数具

[1] 模板将在第11章中讨论。

有副作用的话，宏指令就不总是能够正确地工作。例如，

```
max(x++, y)
```

这将会产生

```
((x++) > (y) ? (x++) : (y))
```

只要是x的值大于y的值，x的值将会增加两倍。

当调用一个C++中的模板函数时，或者是使用&操作符来获取这个模板函数的地址时，模板函数就被隐式地实例化。例如，上面定义的模板函数将会被下面的代码段实例化两次，一次是为int类型的参数，另一次是为char类型的参数：

```
int a, b, c;
char d, e, f;
...
c = max(a, b);
f = max(d, e);
```

下面是曾经在9.8.1节中给出的通用排序子程序的C++版本。它在某种程度上与以前的版本有所不同，因为C++中的数组下标被限制为整数，而下标的下限被固定为零。

```
template <class Type>
void generic_sort(Type list[], int len) {
  int top, bottom;
  Type temp;
  for (top = 0; top < len - 2; top++)
    for (bottom = top + 1; bottom < len - 1; bottom++)
      if (list[top] > list[bottom]) {
        temp = list[top];
        list[top] = list[bottom];
        list[bottom] = temp;
      } //** list[top] ... 结束
} //** end of generi 结束
```

这个模板函数实例化的一个例子为：

```
float flt_list[100];
...
generic_sort(flt_list, 100);
```

Ada的通用子程序和C++的模板函数就子程序而言是一对难兄难弟。它们不同于常见的一般子程序。一般，子程序中的形参类型与调用中的实参类型被动态地绑定。因此只需要代码的单个拷贝。然而在Ada和C++的方式中，必须在编译时为每一个不同的类型产生一个拷贝，而子程序的调用与子程序之间的绑定是静态的。

9.8.3 Java 5.0中的通用方法

在Java 5.0中，增加了对于通用类型以及通用方法的支持。Java 5.0中通用类的名称由尖括号限定的一个或更多类型变量名来指定。例如，

```
GenType<T>
```

这里，T是类型变量，通用类型的更多内容将在第11章中讨论。

Java中的通用方法与Ada以及C++中的通用子程序在几个重要方面不同。第一，通用参数必须是类，而不能够是基本类型。因为有这种要求，就不允许一个通用方法模仿我们在Ada和

C++中的那些例子，在那些例子中，数组部件的类型既是通用类型的，同时又可以是基本类型的。Java中的数组部件的类型（与容器相反）则不能够是通用的。第二，尽管Java通用方法的实例化能够进行多次，但是只有一个代码拷贝产生了。一个通用方法的内部版本称为raw方法，在类对象Object上进行操作。当返回一个通用方法的通用值时，编译器将插入一个类型转换，以使将其转换成为合适的类型。第三，在Java中，限制能够指定在传递通用方法作为通用参数的类的范围内。这样的限制称为界限（bound）。

作为通用Java 5.0方法的例子，考虑下列的方法定义框架。

```
public static <T> T DoIt(T[] list) {
  ...
}
```

它定义了一个带有通用类型元素数组的方法DoIt。通用类型名是T，而且它必须是一个数组。下面是调用DoIt的例子：

```
DoIt<int>(myList);
```

现在，考虑下面形成的doIt。它在其通用参数上有界限：

```
public static <T extends Comparable> T doIt(T[] list) {
  ...
}
```

这定义了一个方法，该方法带有通用数组参数，其元素是实现Comparable接口的类。那是通用参数的限制或界限。保留字extends似乎暗示通用类子类化接下来的类。然而在这里，extends有不同的含义。表达式<T extends BoundingType>指定T应该是绑定类型的"子类型"。无比在这里，extends表示通用类（或接口）扩展绑定类（如果绑定的是类）或实现绑定接口（如果绑定的是接口）。绑定保证任何通用实现的元素都能与Comparable方法CompareTo相比较。

如果一个通用方法对于它的通用类型存在着两种或者多种的限制，则将这些限制放入extends子句，并通过符号&将这些限制分开。另外，通用方法可以具有多个通用参数。

Java 5.0支持通配（wildcard）类型。例如，Collection<?>就是collection类的一个通配类型。可以将这种类型用作任何类部件的任何collection类型。例如，考虑下面的通用方法：

```
void printCollection(Collection<?> c) {
  for (Object e: c) {
     System.out.println(e);
  }
}
```

这个方法将打印出任何Collection类中的元素，而不论它的部件的类是什么。在使用通配类型的对象时必须小心。例如，因为这种类型的某一个特殊对象的部件具有一种类型，就不能够将其他类型的对象再放入这个集合中，例如，

```
Collection<?> c = new ArrayList<String>();
```

如果使用add方法将String类型以外的类型放入这个集合中，就是不合法的。

就像对于非通配类型一样，也可以对通配类型施加一些限制。我们将这种受限制通配类型称为有界的通配类型。例如，考虑下面方法中的首部：

```
public void drawAll(ArrayList<? extends Shape> shapes)
```

这里的通用类型是一种通配类型，它是Shape类的一个子类。可以使用这个方法来画出类型为Shape类的子类的任何对象。

9.8.4 C# 2005的通用方法

C# 2005的通用方法与Java 5.0相似，除了不支持通配类型之外。一个C# 2005通用方法的独特特性是，如果编译器能推测未指定类型，则可以忽略调用中的实参类型。例如，考虑下面的类定义框架：

```
class MyClass {
  public static T DoIt<T>(T p1) {
    ...
  }
}
```

如果编译器能从调用时的实参推测通用类型，那么不需要指定通用参数就能调用方法DoIt。例如，下面调用都是合法的：

```
int myInt = MyClass.DoIt(17);  // Calls DoIt<int>
string myStr = MyClass.DoIt('apples');  // Calls
DoIt<string>
```

9.9 函数的设计问题

下面的两个设计问题是函数所特有的：

- 允许函数具有副作用吗？
- 函数可以返回什么类型的值？

9.9.1 函数的副作用

如在第5章中曾经描述过的，因为在表达式中调用的函数所具有的副作用问题。传递给函数的参数应该总是输入型的。事实上有一些语言就是这样来要求的，例如，Ada的函数就只能够具有输入型的形参。这种要求有效地阻止了函数通过它的参数或者通过参数的别名使用以及全局变量的别名使用而引起的副作用。然而在大多数其他的语言中，函数也可以有按值传递或者按引用传递的参数，这样就允许了产生副作用和别名使用的函数。

9.9.2 返回值的类型

大多数的命令式程序设计语言限制它们的函数所能够返回的类型。C语言允许它的函数返回任何类型，除了数组和函数以外，而且可以通过指针类型的返回值来处理这两种不被允许的类型。C++也像C一样，但是C++还允许从它的函数返回用户定义的类型或者类。Ada、Python和Ruby语言与其他的当代命令式语言不一样，它的函数可以返回任何类型的值。然而，因为在Ada中的函数不是类型，因而函数不能够返回函数。当然，函数可以返回指向函数的指针。

在一些程序设计语言之中，子程序是第一等级的对象。这意味着可以类似于处理数据对象一样来处理它们。例如，在JavaScript中，可以将函数作为参数来传递，以及作为参数从函数中返回。在Python和Ruby中，方法是能被处理为其他任何对象的对象。许多函数式语言也采用了这样的方式（参见第15章）。

Java以及C#都不能够有函数。尽管其中的方法与函数相类似，在这两种语言中，方法可以

返回任何类型或者任何类。然而方法不是类型，因而不能够将方法返回。

9.10 用户定义的重载操作符

在Ada和C++程序中，用户可以将操作符重载。作为一个例子，考虑下面的Ada函数，这个函数将乘法操作符（*）重载，以计算两个数组的点积。两个数组的点积是这两个数组中对应元素对的乘积之总和。假设，定义Vector_Type为具有Integer元素的数组类型：

```
function "*"(A, B : in Vector_Type) return Integer is
  Sum : Integer := 0;
  begin
  for Index in A'range loop
    Sum := Sum + A(Index) * B(Index);
  end loop;  -- for Index ...
  return Sum;
  end "*";
```

正如在这个函数的定义中说明的，每当乘法操作符（*）出现于类型为Vector_Type的两个操作数之间时，就计算点积。还可以进一步将乘法操作符重载任意次数，只要定义的函数中具有唯一的协议。

也可以用C++来编写上面的点积函数。这种函数的一种原型可以为：

```
int operator *(const vector &a, const vector &b, int len);
```

读者自然会提出一个问题：将操作符重载多少次为好，或者说，会不会重载太多次数了？这里的答案是，这在很大程度上取决于你的风格。反对将操作符太多次重载主要考虑的是可读性。在很多情况下，调用一个函数来执行一种操作，相对于使用一个经常被用于其他操作数的操作符，可读性常常更好，甚至是对于点积的情形。可能都会容易忘记在程序中发现的一条简单赋值语句都涉及了什么，如在程序中发现，

430

```
c = a * b
```

会很容易地假设：a、b和c都是简单的数量标量。

另外的一种考虑，是使用不同的开发人员创建的模块来建造软件系统的过程。如果这些不同的开发人员以不同的方式将同一个操作符重载的话，显然，在将这些重载操作符放入系统之前，必须消除掉它们之间的差别。

历史注释

现在很难确定对多单位控制模式概念的实际起源。最早发表的协同程序的应用之一，是句法分析领域（Conway，1963）。第一种包括了协同程序功能的高级程序设计语言是SIMULA 67，前面说过，SIMULA语言的最初目的是系统模拟，这常常需要进行独立过程的模拟。正是这种需求点为了在SIMULA 67中开发协同程序的动机。其他支持协同程序的语言有BLISS（Wulf et al.，1971），INTERLISP（Teitelman，1975）以及Modula-2（Wirth，1985）。

9.11 协同程序

协同程序是一种特殊类型的子程序。不像在传统程序中的调用子程序与被调用子程序之间存在的主-从关系，调用与被调用的协同程序之间是基于更平等的关系。事实上，通常称协同程序的控制机制为对称单位控制模式（symmetric unit control model）。

协同程序具有多个自身控制的入口，也具有保存活动之间状态的机制，这意味着，协同程序必须是历史敏感的，并且因此而具有静态局部变量。一个协同程序的非首次执行通常不是从程序开头的位置开始。正因为这种事实，协同程序的激活被称为重新启动（resume），而不是调用。

例如，考虑下面的协同程序框架：

```
sub co1(){
  ...
  resume co2();
  ...
  resume co3();
  ...
}
```

第一次co1启动执行，它从第一条语句开始，执行向下，启动co2，并把控制传递给co2。第二次co1启动执行，它在调用co2后第一条语句处执行。当第二次启动执行co1时，它在启动执行co3后的第一条语句处开始执行。

协同程序中保持了一个子程序的常见特征：在任何给定的时刻，只有一个协同程序在执行。

正如下面例子中说明的那样，然而协同程序并不是一直执行到结束，经常只是部分地执行，然后将控制转移到另一个协同程序。当执行重新开始时，协同程序在用于转移控制的语句之后重新启动执行。这样的执行顺序与多处理器操作系统的工作方式有关。虽然这里可能只有一个处理器，但这种系统中的所有执行程序似乎都在共享这个处理器，而并发地运行。在协同程序里，有时将这种情形称为准并发（quasi-concurrency）。

十分典型的是，在应用中是由一个称为主单位的程序单位来创建协同程序。这个主程序单位并不是一个协同程序。被创建之后，协同程序将执行它们的初始化代码，然后将控制返回给主单位。当构造出了一个协同程序家族的所有成员之后，主程序将重新启动这些协同程序中的一个，然后协同程序家族的成员以某种顺序相互地重新启动直到完成任务，如果任务可以完成的话。如果一个协同程序的执行达到了它的代码段的末端，控制将返回到创建它的主单位。这就是当需要的时候结束一组协同程序的执行机制。在某些程序中，每当计算机开始运行时，协同程序也同时启动运行。 431

一组协同程序在一起解决问题的一个例子是模拟一种纸牌游戏。假设这种游戏有四个牌手来玩，他们都使用相同的策略。可以用一个主程序单位产生出一个协同程序家族来模拟这种游戏，每一个协同程序初始化以得到一手牌，然后主程序可以重新启动其中的一个牌手协同程序来开始模拟。这个牌手玩过了自己的一轮之后，又可以重新启动下一个牌手协同程序，依此类推，直到游戏结束。

可以使用同样形式的重新启动语句，来开始或者重新开始一个协同程序的执行。 432

假设程序单位A和B为协同程序。图9-3显示了涉及A和B的执行顺序的两种方式。

在图9-3a中，通过主单位来启动协同程序A的执行。经过了一段执行之后，A再启动B。当图9-3a中的协同程序B将控制返回到协同程序A时，语义为A从它上次执行结束之处再继续执行，尤其是，它的局部变量具有上次执行时留下来的值。图9-3b显示了协同程序A和B的另一种执行顺序。在这种情形下，B被主单位启动。

不同于图9-3中显示的模式，协同程序常常具有一个循环，用以包含它的重新启动语句。图9-4显示了这种情况下的执行顺序。在这种情形下，A由主单位来启动。在它的主循环的内部，A重新启动B。而B反过来又在它的主循环中重新启动A。

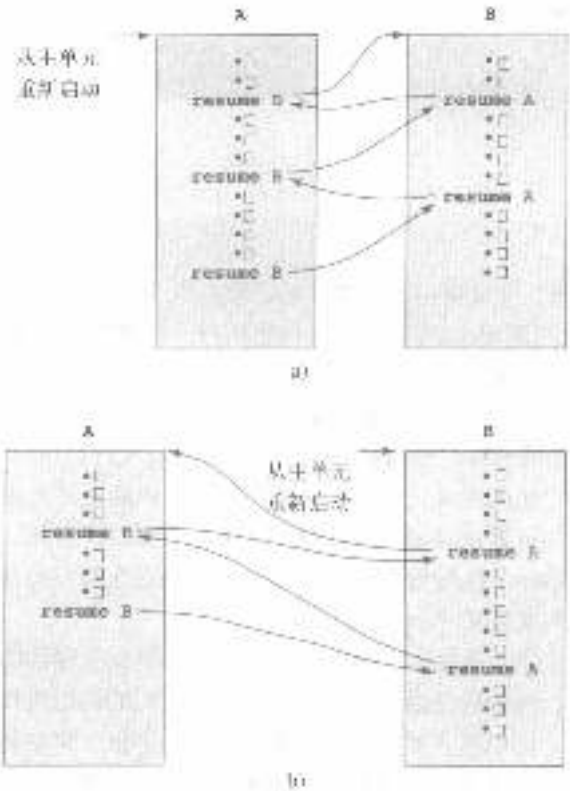


图9.3 两个不具有循环的协同程序可能的执行控制顺序

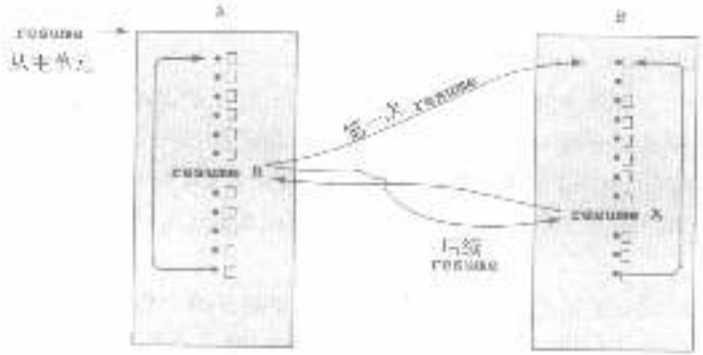


图9.4 两个具有循环的协同程序的执行顺序

小结

在程序设计语言中，过程抽象由子程序表示。子程序的定义描述子程序代表的行为，而子程序的调用则激活这种行为。

形参是一些参数名称，子程序使用这些名称来引用那些在子程序调用之中的实参。Python和Ruby的数组和散列形参用来支持参数的变量数。Ruby允许方法调用时加入块，块可以带参数。在被调用的方法里，调用它们时带有yield语句。

子程序可以是函数或者过程，前者模拟数学函数，并且被用于定义新的操作；后者则被用于定义新的语句。

子程序中的局部变量可以是栈动态的，以支持递归，也可以是静态的，给局部变量提供高效率以及

历史敏感性。 433

参数传递共有三种基本语义模式：输入型、输出型以及输入输出型，它还有多种实现方式。

当使用按引用传递的参数时，可能在两个或者多个参数之间，以及在一个参数与一个可存取的非局部变量之间，产生别名使用。

基于程序的名称的参数提供了一种十分必要的机制，但有时却难以理解。当运行一个被作为参数来传递的子程序时，难以区分其引用环境。

Ada、C++、C# Ruby和Python允许子程序以及操作符的重载。子程序可以是重载的，只要子程序的各种版本中的参数类型或者所返回的值不具有歧义性，可以使用函数的定义来构造操作符的额外意义。

通过使用参数的多态，Ada、C++和Java 5.0中的子程序可以为通用的。这样就可以将所需要的数据对象的类型传递给编译器，然后编译器就能够构造所需要类型的单位。Java 5.0中的方法也可以是通用的，然而是以一种更加受限的、也更具有潜在效率的方式。

协同程序是一种具有多个入口的特殊子程序，可以使用这种程序来提供子程序的交错执行。

复习题

1. 子程序的三个共同特征是什么？
2. 一个子程序是活跃的意味着什么？
3. 什么是参数描绘？什么是子程序的协议？
4. 什么是形参？什么是实参？
5. 关键字参数的优点与缺点各是什么？
6. 子程序有哪些设计问题？
7. 动态局部变量的优点与缺点各是什么？
8. 参数传递有哪三种语义模式？
9. 什么是传递的模式和传递的概念模式？参数传递的按值传递、按结果传递、按值与结果传递以及按引用传递，这些方式各自具有什么优点与缺点？
10. 在按引用传递参数时，别名可以什么样的方式出现？
11. 当所处理的实参类型与它相对应的形参类型不一致时，早期的C与C89版本在处理方式上有什么不同？
12. Ada的策略允许实现人员决定，哪些参数使用按引用传递，哪些参数使用按值与结果传递。这种方式存在什么问题？ 434
13. 对于参数传递的方法，有哪两种基本的设计考虑？
14. 当将子程序名作为参数时，会产生哪两种问题？
15. 定义作为参数传递的子程序，其引用环境的浅绑定以及深绑定。
16. 什么是重载的子程序？
17. 什么是参数的多态？
18. C++的模板函数的实例化是由什么引起的？
19. 通用参数对于Java 5.0中的通用方法与C++中的方法，在哪些基本方式上不同？
20. 如果Java 5.0中的一个方法返回一种通用类型，它实际上返回的是对象的什么类型？
21. 如果调用了Java 5.0中的一个通用方法，编译器将产生这个方法的几个版本？
22. 重载的设计问题是什么？
23. 协同程序与传统的子程序在哪些方面不同？

练习题

1. 对于在用户程序中为现存的操作符增加额外的定义，如在Ada和C++中那样，支持和反对这种方法的论点是什么？你相信这种用户定义的操作符的重载好还是不好？请给出理由。
2. 在大多数的Fortran IV实现中，参数是按引用传递的，仅传递存取途径。陈述这种设计选择的优点与

缺点。

3. 给出你的论点，来支持Ada 83的设计人员允许实现人员在实现in out型（输入输出型）参数时于通过复制或引用之间所做出的选择决定。

4. 假如你愿意编写一个子程序，这个子程序在新的输出页上打印一个标题，还同一个开始启动时为1并在每次活动时增加1的页码。如果不使用参数，也没有对非局部变量的引用，你能不能够在Java中完成这项任务？也能够在C#中完成这项任务吗？

5. 考虑使用C中语法编写的下面的程序：

```
void swap(int a, int b) {
  int temp;
  temp = a;
  a = b;
  b = temp;
}
void main() {
  int value = 2, list[5] = {1, 3, 5, 7, 9};
  swap(value, list[0]);
  swap(list[0], list[1]);
  swap(value, list[value]);
}
```

435

对于下面每一种参数传递方法，在对swap三次调用中的每次调用之后，变量value和变量list的值各是什么？

a. 按值传递

b. 按引用传递

c. 按值与结果传递

6. 给出反对在子程序中同时提供静态和动态局部变量的论据。

7. 考虑使用C中语法编写的下面的程序：

```
void fun (int first, int second) {
  first += first;
  second += second;
}
void main() {
  int list[2] = {1, 3};
  fun(list[0], list[1]);
}
```

对于下面每一种参数传递的方法，在执行程序之后，数组list的值将是什么？

a. 按值传递

b. 按引用传递

c. 按值与结果传递

8. 给出反对在C的设计中仅提供函数子程序的论据。

436

9. 从一本Fortran教科书上学习语句函数的语法和语义。论证在Fortran语言中存在语句函数的正确性。

10. 学习在C++和Ada语言中用户定义操作符重载的方法，并运用我们评估语言的标准，写出比较这两种方法的报告。

11. C#语言支持输出型参数。然而，Java和C++却不支持。请解释这种差别。

12. 研究以使用按名传递的参数而闻名的Jensen设备，请简略地描写这种设备是什么，以及如何使用这种设备。

程序设计练习题

1. 编写一个Fortran程序，这个程序将确定你所涉及的一种Fortran编译器究竟是将局部变量实现为静态的，还是实现为栈动态的。提示：检测这个问题的最容易方式是让你的程序测试一个子程序的历史敏感性。
2. 使用你所知道的一种语言编写一个程序，这个程序将确定按引用传递一个大数组所需要的时间，与按值传递同样一个数组所需要时间之比。在机器可能的范围之内，建立一个尽可能大的数组，并且实现你的这种应用。尽可能多次地传递这个数组，以便获取传递操作的精确时间。
3. 编写一个C#或者Ada程序，这个程序将确定在什么时候计算一个输出型参数的地址(是在调用时，还是在子程序执行结束之时)。
4. 编写一个Perl程序，这个程序将按引用传递的方式将一个字面常量传递给一个子程序；而这个子程序将会试图改变这个参数。在Perl语言总体设计思想的前提下，解释这种结果。
5. 使用C#语言重复程序设计练习题4。
6. 使用一种语言来编写一个程序，在这种语言的子程序中的局部变量既可以是静态的，又可以是栈动态的。在子程序中产生6个大型矩阵（至少是100×100的）。其中的三个为静态的，另外的三个为栈动态的。使用范围为1到100的随机数字将两个静态的矩阵以及两个栈动态的矩阵填满。子程序中的代码必须在静态矩阵上进行大量次数的矩阵乘法，并且将计算过程计时。然后再在栈动态矩阵上重复进行同样的计算。比较并且解释这两种计算的结果。
7. 编写一个C#程序，在程序中包括被重复多次调用的两种方法。这两种方法都传递一个大数组；其中的一种方法是按值传递，另外的一种则按引用传递。比较调用这两个方法所需要的时间，并且解释这种差别。一定要足够多次地调用这两种方法，以便能够说明它们在所需时间上的差别。
8. 编写一个Ada程序，这个程序将确定，如果一个函数被通过指针传递给另一个函数，调用这个函数是否合法。
9. 编写一个程序，使用无论哪种你喜欢的语言的语法，在参数传递时使用按引用传递或按值和结果传递来产生不同的行为。
10. 编写一个Ada通用函数，这个函数取一个具有通用元素的数组作为参数，并且以同样类型的标量来作为数组中的元素。这些数组元素的类型以及这些标量是通用参数。数组的下标是正整数。这个函数必须在给定的数组之中搜索给定的标量，并且返回这个标量在数组中的下标值。如果这个标量没有在数组中，函数就必须返回-1，将函数实例化为Integer以及Float两种类型，并且对这两种实例化情形进行测试。 437
11. 编写一个C++通用函数，这个函数取一个具有通用元素的数组作为参数，并且以同样类型的标量来作为数组中的元素。这些数组元素的类型以及这些标量是通用参数。这个函数必须在给定的数组中搜索给定的标量，并且返回这个标量在数组中的下标值。如果这个标量没有在数组中，函数就必须返回-1。对于int和float这两种类型，测试这个函数。
12. 设计一个子程序和调用代码，其一个或多个参数是按引用传递和按值和结果传递的，以产生不同的结果。 438

第10章　实现子程序

本章的目的是研究实现子程序的方法。这些讨论将给读者提供子程序链接如何工作的详细情况，以及为什么ALGOL 60在20世纪60年代早期对毫无警惕的编译器编写人员曾经是一种挑战。我们将从最简单的子程序开始：具有静态的局部变量、不能嵌套的子程序；然后提升到具有栈动态变量的较复杂子程序；最后再到具有栈动态变量以及非局部作用域的嵌套子程序。在具有嵌套子程序的语言中，实现子程序的额外困难是由需要引入，支持非局部变量的存取机制而引起的。

我们将详细地讨论，在静态作用域的语言中存取非局部变量的静态连接方法。而对实现块的技术将给予简略描述。我们还将讨论，在一种动态作用域的语言中，存取非局部变量的几种方法。

10.1至10.5节讨论的全是在静态作用域的语言中实现子程序；而10.6节讨论的是在动态作用域的语言中实现子程序。

10.1　调用与返回的一般语义

我们将子程序的调用及返回操作统称为这种语言的子程序链接。子程序的任何实现方法都必须以实现语言的子程序连接的语义为基础。

在一种典型的语言中，与子程序调用相关的动作有很多。这种调用过程必须包括无论使用何种参数传递的方法的实现。如果局部变量是非静态的，这种调用必须对在被调用子程序中声明的局部变量进行存储空间的分配，还必须将这些变量与所分配的存储空间相绑定。它必须保留调用程序单位的执行状态，执行状态是任何需要重新启动调用程序单元的状态。它包括寄存器值，CPU状态位和环境指针（EP）。EP用来在子程序执行期间存取参数和局部变量，第10.3.2节将对它作更深入的讨论。调用过程必须安排将控制转移到子程序的代码，并且在子程序执行结束后必须确保能够将控制返回到正确的位置。最后，如果这种语言支持嵌套子程序，调用过程还必须产生某种机制，以提供对被调用子程序为可见的、非局部变量的访问。

子程序返回所必需的动作十分复杂。如果这种子程序具有输出型参数，并且是通过复制来实现的，那么返回过程的第一个动作就是将形参的局部值转移到相关联的实参上。下一步，它必须将局部变量使用的存储空间解除分配，并且恢复调用程序的执行状态。如果这种语言支持嵌套子程序的话，还必须采取一些行动，将非局部引用的机制返回到调用之前的状态。最后，必须将控制返回到调用程序。

439
↓
440

10.2　实现“简单”子程序

我们从实现简单子程序的任务开始。这里的“简单”指的是，子程序不能够是嵌套的，并且所有的局部变量都是静态的。早期的Fortran语言就具有这类子程序。对一个“简单”子程序调用的语义要求有下面这些动作：

1. 保留程序单位当前的执行状态。

2. 传递参数。

3. 将返回地址传送给被调用的程序。

4. 将控制转移给被调用的程序。

从简单子程序返回的语义要求有下面的动作：

1. 如果存在按值与结果传递的参数，或者是输出型的参数，将这些参数的当前值转移到对应实参上。

2. 如果子程序是一个函数，将函数值转移到调用程序可以存取的位置。

3. 恢复调用程序的执行状态。

4. 将控制转回调用程序。

调用与返回动作要求有空间来存储以下信息：

- 关于调用程序的状态信息。
- 参数。
- 返回的地址。
- 函数子程序的函数值。

所有这些连同局部变量以及子程序的代码一起，形成了一套子程序执行及在执行后返回控制给调用程序所需要的全部信息。

一个简单的子程序包括两个公开的部分：子程序的实际代码（这是不变部分），以及局部变量和上面所列举的数据（这个部分在子程序的执行中是可以改变的）。而在简单子程序的情形下，这两个部分的大小都是不会改变的。



图10-1 简单子程序的活动记录

子程序中的非代码部分的格式（或布局）被称为**活动记录**。因为这一部分仅仅描述子程序的活动期间或执行期间的有关数据。活动记录的形式是静态的。一个活动记录实例是活动记录的一个具体示例，它是活动记录形式的一组数据。

因为仅有简单子程序的语言不支持递归，所以在某一个时刻，给定的子程序只能够有一个活跃版本。因此，只可能存在一个子程序活动记录的单个实例。图10-1显示了活动记录的一种可能格式。在这个图中，我们省略了被保留的调用程序的执行状态，因为这种状态十分简单，并且与我们的讨论没有关联。

因为简单子程序的活动记录实例具有固定的大小，因此它可以被静态地分配存储空间。事实上，可以将这个活动记录实例附在子程序的代码部分中。

图10-2显示一个程序，它包括了一个主程序以及三个子程序：A、B和C。虽然在图10-2中，所有的代码段与活动记录实例是分开的，但在某些情况下，可以将这些活动记录实例附于与它们相关联的代码上。

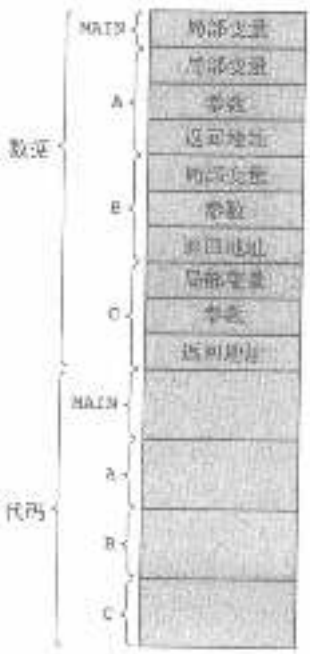


图10-2 一个具有简单子程序的程序代码及活动记录

441

442

图10-2中显示的完整的程序构造并不是全部由编译器来完成。事实上，因为有独立编译，这四个程序单位——MAIN、A、B以及C——可能是在不同的日子，甚至是在不同的年份被编译的。在编译每

一个程序单位时，它的机器码连同一张对外部子程序的引用都被写入一个文件中。图10-2中显示的可执行程序通过操作系统的连接器被连接起来（有时，也将连接器称为装载器（loader）、装载与连接器（loader/ linker）或者连接编辑器（link editor））。当调用连接器来连接一个主程序时，它的第一项任务是要找到在程序中引用的被翻译了的子程序文件，连同它们的活动记录实例，并且将这些载入存储器。然后，连接器必须将所有对主程序中子程序调用的目标地址设置为这些子程序的入口地址。连接器还必须对所有装载子程序中的子程序调用以及所有的库子程序调用进行同样的工作。在前面的例子中，是由这种连接器来调用主程序MAIN，连接器还必须找到A，B和C的机器码程序，连同它们的活动记录实例，并且将它们与MAIN的代码一起载入存储器。然后，连接器还必须设置所有对A，B和C调用的目标地址，以及所有在A，B，C和MAIN中对库子程序调用的目标地址。

10.3 实现具有栈动态局部变量的子程序

现在来研究如何在具有栈动态局部变量的语言中实现子程序的连接，并且再次将注意力集中在调用与返回操作上。

栈动态局部变量的一个最重要的优点是支持递归。因而，具有栈动态局部变量的语言也支持递归。

我们将被这类子程序所需要的额外复杂性问题延后到10.4节讨论。

10.3.1 更复杂的活动记录

443

在具有栈动态局部变量的语言中的子程序连接比在简单的程序的子程序连接更为复杂，主要有如下理由：

- 编译器必须为局部变量的隐式存储空间分配和解除分配产生代码。
- 递归增加了一个子程序同时有多个活动的可能性。这意味着在同一时刻可能有一个子程序的多个实例（未完成的执行），其中的一个调用是子程序的外部调用，另外的一个或多个调用则是递归调用。因此，递归要求有活动记录的多个实例，每一个实例对应于子程序的一个活动，而这些实例可以同时存在。每一个活动都需要有自己的形参的副本，以及被动态分配了空间的局部变量，连同返回的地址。

在大多数语言中，一个给定子程序的活动记录的格式可以在编译时知道，在许多情况下，活动记录的大小也是已知的。因为所有的局部数据都是固定大小的。但在有些语言中，例如Ada语言，就不是这样的情形。在这些语言中，一个局部数组的大小可能取决于某一个实参的数值。在这种情况下，活动记录的格式虽然是静态的，但它的大小却可能是动态的。在具有栈动态局部变量的语言中，就必须动态地产生活动记录的实例。图10-3显示了这样一种语言的典型的活动记录。



图10-3 一种具有栈动态局部变量的语言的典型活动记录

因为调用程序将返回地址，动态连接以及参数都放置于活动记录实例中，所以这些项必须首先出现。

返回地址通常包括了一个指向指令的指针，该指令后接着在调用程序单元的代码段中的调用。动态连接是一个指向调出程序活动记录实例顶端的指针，在静态作用域的语言中，当完成了程序的执行之后，使用这种连接来删除当前的活动记录实例。栈顶被设置为前一个动态连接的值。之所以要求动态连接，是因为在某些情况下，一个子程序从栈里分配到的空间会超出活

动记录的范围。例如，这个子程序的机器语言版本需要临时的空间，就可能在那里进行分配。因此，虽然已经知道活动记录的大小，但不能够将这种大小简单地从栈顶的指针上减掉，从而删除活动记录。活动记录中的实参是调用程序所提供的值或地址。

局部数量变量被绑定于一个活动记录实例范围之内的存储空间。结构化的局部变量则有时是在别处被分配。只有这些变量的描述符以及指向它们的存储区域的指针是活动记录中的一部分。局部变量在被调用的子程序中被分配，并且可能在被调用的子程序中进行初始化，因而它们是在最后出现。

考虑下面的C函数框架：

```
void sub(float total, int part) {
  int list[5];
  float sum;
  ...
}
```



图10-4 函数sub的活动记录

图10-4显示了函数sub的活动记录。

启动一个子程序，需要动态地产生这个子程序的活动记录实例。如前所述，活动记录的格式在编译时被固定，而活动记录的大小可能取决于调用。因为调用与返回的语义说明应该首先执行最后被调用的子程序，由此在栈上产生这些活动记录的实例就十分合理。栈是运行时系统的一部分，因此我们将它称为运行时栈（run-time stack），但通常我们仅称它为栈。子程序的每一种活动，不论是递归还是非递归活动，都将在栈上产生一个活动记录的新实例。这样就提供了所需要的参数、局部变量以及返回地址的分离的副本。

更多需要做的是控制子程序的执行。开始时，印指向基址或主程序活动记录实例的首地址。随后，运行时系统必须确保它总是指向当前执行程序单元的活动记录实例的基地址，当调用子程序时，当前EP和其他执行状态信息一起存储在新活动记录实例中。然后把EP设置为指向新活动记录的基址。根据了程序的返回信息，从完成执行操作的子程序的活动记录实例中恢复EP。

注意，当前使用的EP不是存储在运行时栈中，只有保存形式存储在活动记录实例中。因为这种保存形式与其他执行状态信息存储在一起，因此存储EP没有在运行时栈的图中示出。

第9章中曾经谈到，一个子程序从被调用的时刻开始到执行完成之时，这段期间是活动的（active）。当子程序不再活动时，它的局部作用域也就不再存在，它的引用环境也不再具有意义。因而到了这个时刻，就可以删除掉这个子程序的活动记录实例。

10.3.2 一个不具有递归的例子

考虑下面的C程序框架：

```
void fun1(float r) {
  int s, t;
  ...      <-------------1
  fun2(s);
  ...
}

void fun2(int x) {
  int y;
  ...      <-------------2
  fun3(y);
```

```
  ...
}

void fun3(int q) {
  ...      <-------------3
}

void main() {
  float p;
  ...
  fun1(p);
  ...
}
```

在这个程序中，函数调用的顺序为

main调用fun1
fun1调用fun2
fun2调用fun3

图10-5显示了在被标记为1、2和3的位置处栈中内容。

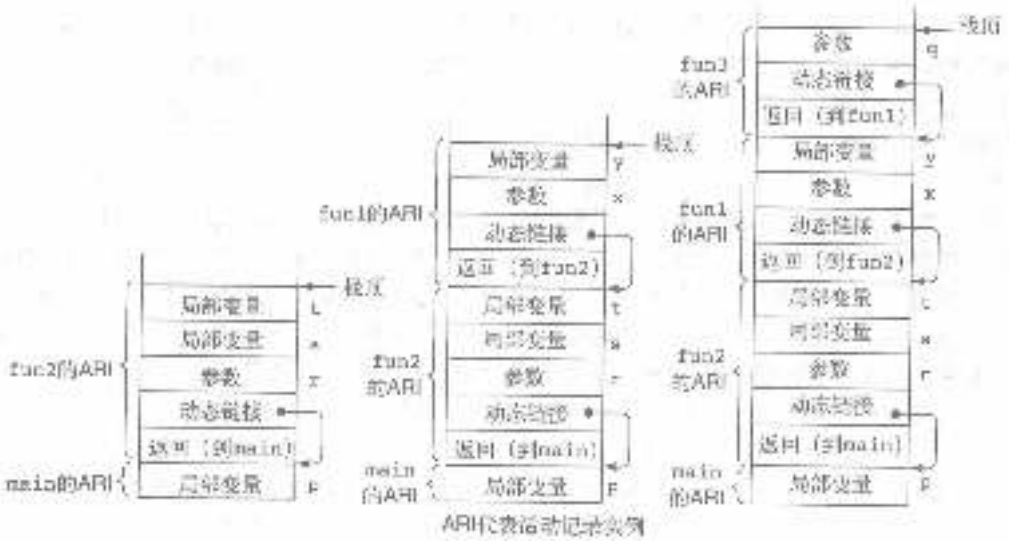


图10-5 程序中3个位置的栈内容

在标记为1的位置，在栈上只有程序main和函数fun1的活动记录实例。当fun1调用fun2时，栈上产生了fun2的活动记录的一个实例。当fun2调用fun3时，栈上又产生了fun3的活动记录的一个实例。当fun3的执行结束时，它的活动记录实例则被从栈上删除，此时还将使用动态连接来重新设置栈顶的指针。当函数fun1和fun2结束时，也将发生类似的过程。当在main中对fun1的调用返回之后，栈上只有main的活动记录实例。注意，有些实现对主程序实际上并不使用图10-5中所示的栈上的活动记录实例。然而它也可以这样做，这样可以使程序的实现和我们的讨论都简单化。在这个例子以及本章所有其他例子中，我们假设栈从比较低的地址往比较高的地址增长，尽管在一个特别实现中，栈可能会反向增长。

446

在给定的时刻出现在栈中的一组动态连接被称为动态链或者调用链。它代表执行是怎样到达当前位置的一个动态的经历。这个当前位置总是在活动记录实例位于栈顶的程序代码中。可以在代码中将对局部变量的引用表示为从局部作用域中的活动记录开始的偏移。这种偏移被称

为局部偏移（local offset）。

通过使用与活动记录相关的子程序中所声明变量的次序、类型以及大小，可以在编译时确定一个变量在活动记录中的局部偏移。为了简化这种讨论，我们假设所有的变量都在活动记录中占据一个位置。在一个子程序中声明的第一个变量在活动记录中被分配的位置是从记录底部往上第三个位置（前两个位置为返回地址和动态连接）再加上参数的数目。所声明的第二个局部变量在活动记录中的位置继续往栈顶最近一个位置，依此类推。例如，再次考虑前面的程序例子。在fun1中y的局部偏移是3，t是4的局部偏移；类似地，在fun2中y的局部偏移是3。为了得到任何局部变量的地址，变量的局部偏移加入到EP中。 477

10.3.3 递归

考虑下面的C程序例子，这个程序使用递归来计算阶乘函数：

```
int factorial(int n) {
    <-------------1
  if (n <= 1)
    return 1;
  else return (n * factorial(n - 1));
    <-------------2
}
void main() {
  int value;
  value = factorial(3);
    <-------------3
}
```



图10-6 函数factorial的活动记录

图10-6显示了函数factorial的活动记录的格式。注意，它有一个用于函数返回值的额外项。图10-7显示三次执行到达函数factorial中位置1时栈中的内容。每一个都显示了这个函数的

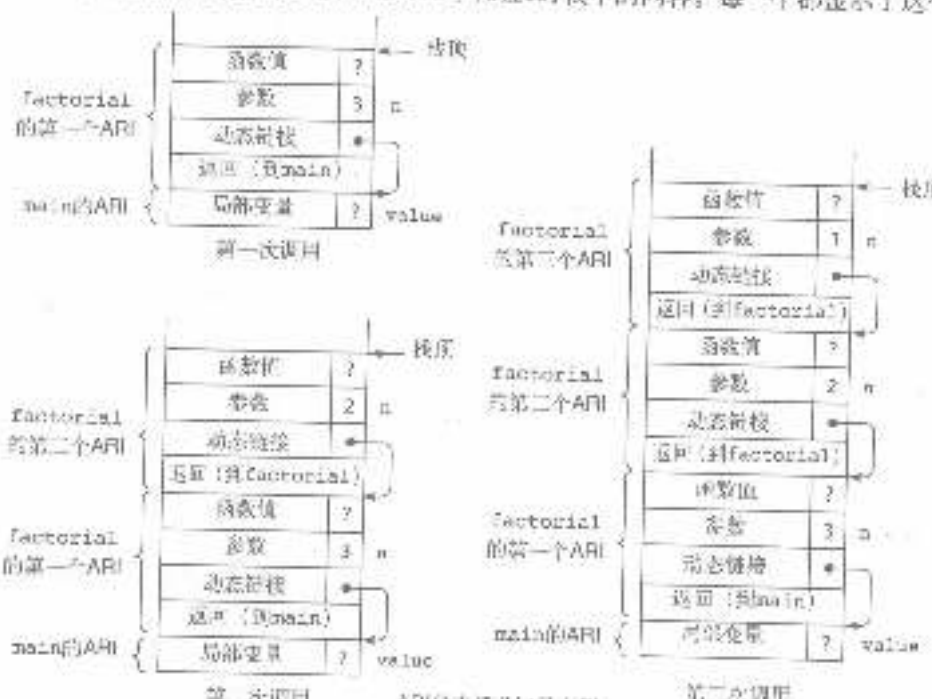


图10-7 函数factorial中位置1的栈中内容

义一次活动，而它的函数值并没有定义。第一个活动记录实例具有返回给调用函数main的一个地
448 址。另外的两个各具有返回给函数自身的一个地址，这些地址是用于递归调用的。

图10-8显示三次执行到达函数factorial中位置2时栈中的内容，位置2指的是在执行了return之后，但是在活动记录被从栈上删除之前的这一时刻。前面讲过，函数的代码将参数n的当前值乘以函数的递归调用所返回的值。从factorial中第一次返回的值为1。这次活动的活动记录实例以值1作为它的参数n的版本。这次乘法的结果1再被返回给factorial的第二次活动，以便与它的参数n的值2相乘。这次的返回值2返回给factorial的第一次活动，以使与它的参数
449 n的值3相乘。产生最后函数值6，然后将这个值返回给main中对factorial的第一次调用。

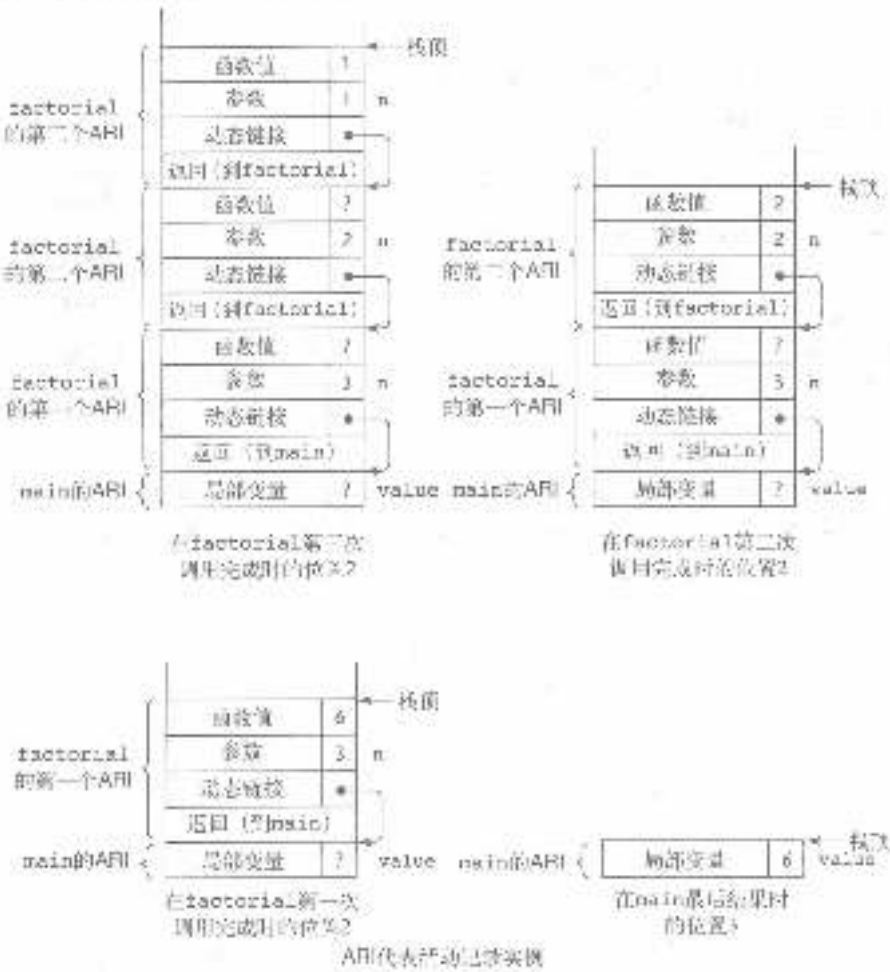


图10-8 main和factorial执行返回前栈中内容

10.4 嵌套子程序

450 一些不基于C的静态作用域程序设计语言使用栈动态局部变量，并且允许子程序嵌套。这些语言有Fortran 95、Ada以及JavaScript。本节将审视在这种环境中最常用的实现子程序的方式。

10.4.1 基础

在具有嵌套子程序的静态作用域语言中，需要一种具有两个步骤的存取过程来引用非局部

的变量。所有能够被非局部访问的变量都已经存放在已有的活动记录实例中，也就是位于栈中的某一位置。存取过程的第一个步骤是在栈中找到变量被分配其中的那个活动记录实例。第二个步骤是使用这个变量的局部偏移（在活动记录实例内）进行存取。

找到正确的活动记录实例，是这两个步骤中较有趣也是较困难的一步。首先要注意到，在一个给定的子程序中，只有在静态前辈的作用域中声明的变量才是可见的，才能够被存取。除此而外，当通过一个嵌套子程序引用这些变量时，所有这些静态前辈们的活动记录实例总是被存放于栈之上。这由静态作用域语言的静态语义规则保证：即，一个子程序是可以被调用的，仅当这个子程序的所有静态前辈程序单位都是活跃的。如果某个特别的静态前辈是非活跃的，它的局部变量将不会被绑定到存储空间，因而无法对这些变量进行存取。

非局部引用的语义规定：当在一个包含作用域中进行搜寻时，从最深内的嵌套子程序开始向外查找，这样找到的第一个声明就是正确说明。因而为了支持非局部引用，必须有可能在栈中找到对应于静态前辈们的所有活动记录实例。这就导致了下面所述的这种方法。

在10.5节之前，我们将不讨论块的问题，因而在本节余下部分，所有的作用域都是由子程序定义的。因为在基于C的语言中函数是不能被嵌套的（只能够通过块产生这些语言的静态作用域），因而在这一节的讨论中将不包括这些语言。

10.4.2 静态链

在一种允许嵌套子程序的语言中，实现静态作用域的最常用方式是静态链。使用这种方式时，将一个被称为静态连接的新指针附加到活动记录中。静态链接有时也被称为静态作用域指针，它指向静态父层活动的活动记录实例的底层。静态连接被用来存取非局部变量。典型地，静态连接在活动记录中出现在参数所在的位置下面。静态连接添加到活动记录中以后，要求局部偏移比在静态连接没有加入时要有所不同。在活动记录中位于参数之前的位置就有了二个元素，而不是象以前只有两个元素，它们现在是：返回地址，静态连接以及动态连接。 451

静态链是找中活动记录实例的静态连接的一个链。在执行过程P的期间，过程P的活动记录实例的静态连接将指向P的静态父层程序单位的活动记录实例。如果过程P的祖父辈的活动记录实例存在，静态父层程序单位的活动记录实例的静态连接将转而指向这个实例。因而，静态链以静态父层优先的顺序连接了执行子程序的所有静态前辈。显然，能够使用这种链在静态作用域语言中实现非局部变量的存取。

使用静态链找出正确的非局部变量活动记录实例是相对容易的。当引用一个非局部变量时，可以通过搜索静态链来寻找包含这个变量的活动记录实例。这种搜索将一直进行到发现一个静态前辈的活动记录实例包含了这个变量为止。然而，在实际上可以比这还容易。因为在编译时 454
就已经知道了作用域的嵌套，所以编译器不但可以确定一个引用是否为非局部的，还可以确定到达包含这个非局部变量的活动记录实例所需的静态链长度。

设静态深度（static_depth）为一个与静态作用域相关的整数，它表示一个作用域从最外层作用域开始所嵌套的深度。Ada主程序具有的静态深度为0。如果过程A是定义于主程序中的唯一过程，则A的静态深度为1。如果过程A又包含了嵌套过程B的定义，那么B的静态深度为2。

在对变量x的非局部引用中，达到正确的活动记录实例所需要的静态链长度，正好是包含x引用的过程的静态深度与包含x声明的过程的静态深度之差。这个差被称为引用的嵌套深度（nesting_depth）或者链偏移（chain_offset），可以将实际的引用表示为一个整数的有序对（链偏移，局部偏移）。在这里，链偏移是到正确的活动记录实例的连接数目（局部偏移将在第10.3.2节中描述）。例如，考虑下面的程序框架：

```
procedure A is
  procedure B is
    procedure C is
      ...
    end;  -- of C
    ...
  end;  -- of B
  ...
end;  -- of A
```

A，B和C的静态深度分别为0、1和2，如果过程C引用一个在A中声明的变量，这个引用的链偏移将会是2（C的静态深度减去A的静态深度）。如果过程C引用一个在B中声明的变量，这个引用的链偏移将会是1，可以使用同样的机制来处理对局部变量的引用，这种引用的链偏移为零。

Main_2用于说明非局部存取的完整过程，考虑下面的Ada程序框架：

```
procedure Main_2 is
  X : Integer;
  procedure Bigsub is
    A, B, C : Integer;
    procedure Sub1 is
      A, D : Integer;
      begin  -- Sub1 开始
      A := B + C;   <--------------1
      ...
    end;  -- Sub1 结束
    procedure Sub2(X : Integer) is
      B, E : Integer;
      procedure Sub3 is
        C, D : Integer;
        begin  -- Sub3 开始
        ...
        Sub1;
        ...
        E := B + A;  <--------------2
        end;  -- Sub3 结束
      begin  -- Sub2 开始
      ...
      Sub3;
      ...
      A := D + E;  <--------------3
      end;  -- Sub2 结束
    begin  -- Bigsub 开始
    ...
    Sub2(7);
    ...
    end;  -- Bigsub 结束
455 begin  -- Main_2 开始
  ...
  Bigsub;
  ...
  end;  -- Main_2 结束
```

在这里，过程调用顺序是

```
Main_2 调用 Bigsub
Bigsub 调用 Sub2
Sub2 调用 Sub3
```

```
Sub3 调用 Sub1
```

图10-9显示了当执行首次到达位置1时，栈中的情形。

在程序Sub1中的位置1，引用的是局部变量A，而不是Bigsub中的非局部变量A。这个对A的引用具有链偏移/局部偏移对（0，3）。对B的引用是对来自Bigsub中的非局部变量B的引用。这个引用可以由偏移对（1，4）来表示。它的局部偏移为4，因为一个为3的偏移将会是第一个局部变量的偏移（Bigsub不具有参数）。请注意，如果使用动态连接简单地搜索一个具有变量B的声明的活动记录实例，将会找到在Sub2中声明的变量B，这是不正确的。如果对偏移对（1，4）用于这个动态链，就会使用Sub3中的变量B。然而，静态连接指向Bigsub的活动记录，这个活动记录中具有B的正确版本。此时，Sub2中的变量B并没有在引用环境中，而且是不能够被（正确地）存取的。在位置1对C的引用是对在Bigsub中定义的C的引用，这由偏移对（1，5）表示。

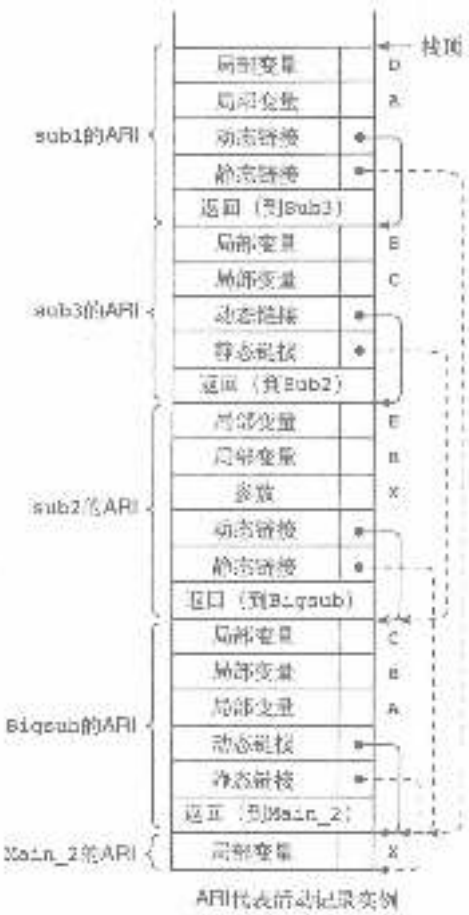


图10-9 程序Main_2中位置1的栈内容

在Sub1执行完毕之后，Sub1的活动记录实例被从栈中删除，控制也将返回到Sub3，在Sub3中的位置2对Sub3中变量E的引用是局部的，并将使用偏移对（0，4）来存取。对变量B的引用是对在Sub2中声明的B的引用，因为Sub2是包含了这种声明的最接近的静态前辈。这通过使用偏移对（1，4）来存取，其中的局部偏移为4，是因为B是在Sub1中声明的第一个变量，并且Sub2具有一个参数。对变量A的引用是对在Bigsub中声明的A的引用，因为Sub3及其静态父辈Sub2都没有包含变量A的声明。这通过使用偏移对（2，3）进行引用。

在Sub3执行完毕以后，Sub3的活动记录实例被从栈中删除，只留下了Main_2、Bigsub以及Sub2的活动记录实例。在Sub2中的位置3对变量A的引用是对Bigsub中A的引用，在活动的子程序中只有它包含了变量A的声明。这通过使用偏移对（1，3）进行存取。在这个位置上没有可见的作用域包含了变量D的声明，因而这个对D的引用是一个静态语义错误。当编译器在企图计算链偏移与局部偏移对时，能够检查出这个错误。对变量E的引用即是对在Sub2中E的引用，可以使用偏移对（0，5）进行存取。

总之，在位置1、2和3对变量A的引用，可由下面这些项表示：

- (0,3)(局部)
- (2,3)(两层以外)
- (1,3)(一层以外)

此刻有理由提出疑问，在程序执行期间如何维护静态链？如果这种维护过于复杂的话，静态链所具有的简单性和高效率就变得不那么重要了。在这里我们假设，没有实现是子程序名的

参数。

457 对于程序的每一次调用和返回都必须修改静态链，返回部分到是相当简单：当子程序结束时，将它的活动记录实例从栈上删除，删除后的新的栈顶活动记录实例是调用刚结束的子程序单位的活动记录实例。因为来自这个活动记录实例的静态链从来就没有被改变，所以就如同在对其他子程序的调用之前一样，这个静态链可以一如既往正确地工作，因而并不需要采取其他的行动。

在子程序调用之时所需要的行动则较为复杂。虽然可以在编译时十分容易地确定正确的父辈作用域，但必须在调用时找到最近的父辈作用域的活动记录实例。可以通过搜索动态链上的活动记录实例，直到发现第一个父辈的作用域为止，从而完成这项任务。然而，如果处理过程的声明及引用完全像处理变量的声明及引用，就可以避免这种搜寻。如果编译器遇上了一个子程序的调用，在多项任务中，它将确定这个被调用子程序的声明所在的子程序，而这个子程序必定是调用了程序的一个静态前辈。然后，编译器将计算嵌套深度，或者计算在调用子程序与声明被调用了程序的子程序之间包含作用域的数目。这种信息将被存储起来，在执行期间可以通过子程序调用来访问。在调用时，被调用子程序的活动记录实例的静态连接是通过将调用子程序的静态链往下移动来确定的，往下移动的连接数目等于在编译时计算的嵌套深度。

再次考虑程序Main_2，以及在图10.9中所示的栈中的情形。当在Sub3中调用Sub1时，编译器确定了Sub3（调用过程）的嵌套深度为过程Bigsub内两个层次，Bigsub是声明被调用过程Sub1的过程。当执行Sub3中对Sub1的调用时，这种信息被用来为Sub1建立活动记录实例的静态连接，将这个静态连接设定为指向一个活动记录实例，这个实例是来自调用过程的活动记录实例的静态链中第二个静态连接所指向的实例。在这个例子中，调用过程是Sub3，其静态连接指向它的父辈过程（Sub2）的活动记录实例。Sub2的活动记录实例的静态连接指向Bigsub的活动记录实例。因此，将Sub1的新的活动记录实例的静态连接设为指向Bigsub的活动记录实例。

这种方法适用于所有的子程序的连接，涉及参数为子程序名的情况除外。

458 关于使用静态链存取非局部变量的一种批评是，引用作用域在静态父辈作用域之外的变量，其气价十分昂贵。这种静态链从引用到声明，必须一个包含作用域一个连接地环环紧跟。另一种批评是，在编写对于执行时间要求很高的程序时，程序人员很难估计非局部引用的代价。这为每个引用的代价取决于在引用与声明作用域之间的嵌套深度，使得这一问题更复杂的情况是，代码的后续修改可能会改变嵌套深度，并由此改变某些引用的时间。这可以发生在改变了的代码中，也可能发生在远离这种改变的代码中。

现在已经发展了一些静态链的替代方法，最著名的一种方法是使用称为显示（display）的辅助数据结构。然而，没有发现一种可替代方法优于静态链方法，它仍然是最广泛使用的方法。这里并不讨论它们。

访谈



保持简单性

NIKLAUS WIRTH

Niklaus Wirth在苏黎世的瑞士联邦技术学院开始了他的工程生涯，后来又到加拿大学习；此后，于1963年在美国的Berkeley大学获得博士学位。他曾经是斯坦福大学和匹兹堡大学的计算机科学的助教授。苏黎世的瑞士联邦技术学院的计算机科学教授，加州Xerox PARC公司的访问员，Wirth教授曾经获得IEEE的计算机先驱者奖。

"因为开发了一系列创新的计算机语言：EULER，ALGOL-W，PASCAL和MODULA"，他获得了ACM的图灵奖。

关于设计与解决问题

问：Pascal清晰一致的结构设立了一个新的标准。Pascal中的什么东西导致了重大的改进？

答：主要是Pascal表示了程序设计中最基本的元素，并让它们以一种自由和通用的方式相结合，就语句和数据方面而言，Pascal是一种结构式语言。

问：你曾经花费大量时间，思考硬件和软件和谐工作问题的解决方案：Lilith计算机与Oberon和Modula。在较大型的硬件构架中有关程序设计工具的设计方面，是什么使你在解决问题时将两者放在一起考虑？

答：如果计算机体系结构和软件能够很好地构造，并且能够一起和谐地工作，那么这两者都将变得更加简单和经济。最重要的是，它们将变得更容易理解。

问：在一篇关于你的采访中，我读到了下面的文字："一个好的设计者必须依靠他的经验，他的准确的逻辑思维和他的精致。像变戏法那样是不行的。"怎样区别变戏法与最好的解决方法？怎么能够将前者从后者中除去？

答：我想大多数情况下是通过经验。有能力进行这种区分的老师将是很有帮助的。

问：在另一次对你的采访中，你提到了今天的设计状况以及设计的动力："用户的愿望比需求更为重要。相对于可靠性和透明的程序，人们更容易买那些具有很酷特点的东西，哪怕这种东西很少使用"。如果事实恰恰相反，你能够想象今天一般用户的操作系统或万维网的界面会是什么样子？在哪些方面会更好一些？

答：客户在描述一个软件系统时，常常并不准确地知道他们到底需要的是什么。因此他们的"愿望"也许并不真正反映了他们的需求。当仔细审阅时将发现，他们对一大堆东西的说明十分多余，至少不重要。如果能够将力量集中在关键的部分，而少搞那些花哨的东西，就能够设计出比较简单、更经济、比较容易理解和操作的健壮系统。

问：你表达过这样的意见，就是科学技术能够完成的事情常常被夸大了，例如，人工智能可以被用来制造能够"思想"的机器。你能不能解释一下，为什么相信这种技术的夸大，会将科学家引向一条一无所获的道路？

答：对于现代IT中很多东西的炒作，还不仅仅是人工智能，这些还没有太多地误导科学家，被误导的是消费者和投资者。支持具有一定风险的长期性研究是一回事，声明言过其实的产品又是另外一回事。

最新的发展以及工具

问：回想在过去的十年间，语言特征中的哪些进展对于开发更好的语言作出了最大的贡献？

答：程序和数据的结构、与断言相关的概念以及循环不变式，模块化设计以及模块的分开编译，数据类型的层次（在面向对象程序设计中，被称为具有继承的子类）。

问：在学生们比较和学习这数式语言、逻辑语言、过程式语言以及面向对象程序设计语言时，应该在学习和工作中怎样考虑过程式的程序设计？过程式程序设计已经成为一种历史风格了吗？什么样的设计将很快成为一种称心的风格？

答：计算机科学专业的学生应该知道各种程序设计方式：过程式的、函数式的、逻辑的以及面向对象的，这是十分重要的。然而明显的是，过程式的风格仍然最接近运行程序的计算机。计算机的特征是具有可以被分别修改的存储器单位，它们对应于程序设计语言中的变量。面向对象的风格就是基于过程式风格的。虽然在面向对象的语言中，将过程称为"方法"，将调用一个过程称为"传送一个消息"；但面向对象只是过程式的一个变种，并非真正本质上的不同。我个人认为，函数式程序设计以及逻辑程序设计是"称心的风格"的程序设计，而过程式的程序设计不是。

452
1
453

10.5 块

第5章里曾经谈到，有几种语言，包括基于C的语言，都为变量提供了用户指定的局部作用

域，它们被称为块（block）。作为块的一个例子，考虑下面一段C的代码段：

```
{ int temp;
  temp = list[upper];
  list[upper] = list[lower];
  list[lower] = temp;
}
```

在基于C的语言中，块为一段代码，这段代码开始于一条或几条数据定义，并且这些定义包括在花括号中。上面所示的块中变量temp的生存期从控制进入块时开始，到控制退出块时终止。使用这样一种局部变量的优越性在于它不会干扰任何在程序中其他位置声明的同名变量。

可以使用我们所描述的、用于实现嵌套子程序的静态链方法来实现块。可以将块处理为总是从程序中的同一位置调用的无参数子程序。因而，每一个块都具有一个活动记录。每次执行这个块时，就会产生它的活动记录的一个实例。

也可以使用不同的、略为简单然而更有效率的方式来实现块。在程序执行中的任何时刻，块变量所需要的最大存储空间可以被静态地确定，因为块的进入与退出是严格按照文本顺序的。可以将块的这些存储空间分配在活动记录中局部变量的后面，也可以静态地计算所有块变量的偏移，因而块变量就可以完全像局部变量那样被直接寻址。

例如，考虑下面的程序框架：

```
void main() {
  int x, y, z;
  while ( ... ) {
    int a, b, c;
    ...
    while ( ... ) {
      int d, e;
      ...
    }
  }
  while ( ... ) {
    int f, g;
    ...
  }
  ...
}
```

459

对于这个程序，可以使用图10-10中的静态存储空间格局。注意f和g占据了相同的存储空间位置，a和b也占据了相同的存储空间位置，因为当a和b的块被退出时（在f和g被分配之前），a和b被从栈中弹出。

460

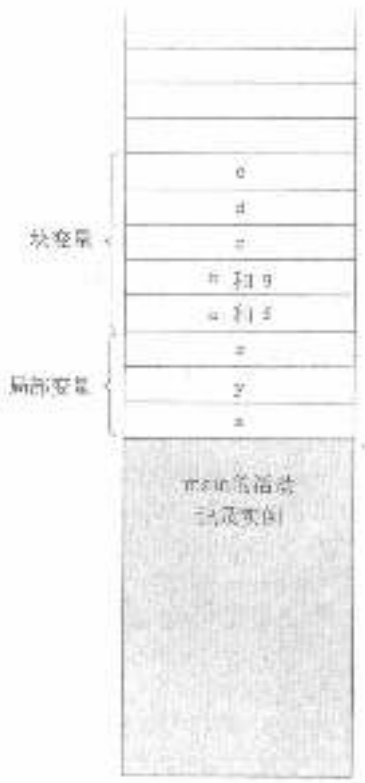


图10-10 当没有将块处理成为无参数的过程时，块变量的存储状态

10.6 实现动态作用域

在一种动态作用域语言中，至少存在着实现非局部引用的两种不同的方式：即深访问和浅访问。请注意，深访问和浅访问的概念与深邦定和浅邦定的概念并没有关联。在这里，邦定与访问之间的基本不同是深绑定和浅邦定将导致不同的语义，而深访问和浅访问则并不导致语义的不同。

10.6.1 课访问

如果在一种动态作用域的语言中，局部变量是栈动态的，并且它们是活动记录中的一部分，那么对非局部变量的引用可以通过搜索具有当前活动的子程序的活动记录实例来得以解决，搜索将开始于最近激活的子程序。这种概念类似于，在一种使用嵌套子程序的静态作用域的语言中访问非局部变量；只是所跟踪的是动态链而不是静态链。动态链将所有子程序活动记录实例连接在一起，但是以这些子程序活动先后的反向次序进行连接。因此，动态链恰恰就是在一种动态作用域的语言中引用非局部变量所需要的技术。这种方法被称为课访问，因为这种访问可能需要在栈中深入地搜索。

考虑下面的程序例子：

```
void sub3() {
  int x, z;
  x = u + v;
  ...
}

void sub2() {
  int w, x;
  ...
}

void sub1() {
  int v, w;
  ...
}

void main() {
  int v, u;
  ...
}
```

这个程序的语法从表面上看像是一种基于C的语言的程序，但这并不意味着它是任何特定语言的程序。假设有下面的程序调用序列：

```
main 调用 sub1
sub1 调用 sub1
sub1 调用 sub2
sub2 调用 sub3
```

图10-11显示了在这个调用序列之后函数sub3的执行期间栈中的情形。请注意，这种活动记录实例没有静态连接；在一种动态作用域的语言中，静态连接没有用途。

考虑对函数sub3中变量x，u和v的引用。对x的引用是在sub3的活动记录实例中找到的，对u的引用是通过搜寻栈上所有的活动记录实例才最终找到的，因为现有的、唯一具有这个名称的变量是在main中。这种搜寻涉及跟踪四个动态连接，以及检查十个变量名称。对v的引用是在函数sub1的最近的（动态链上最接近之处）活动记录实例中



图10-11 一个动态作用域程序的栈中的内容

找到的。

在动态作用域语言中的非局部引用的深访问方法，与在静态作用域语言中的静态链方法之间，存在着两个重大的差别。首先，在动态作用域的语言中，无法在编译时确定必须搜索的链的长度。然而，却必须搜索这个链中的每一个活动记录实例，直到找到了被引用变量的第一个实例为止。这就是为什么动态作用域语言的执行速度通常比静态作用域语言的执行速度更慢的原因。其次，为了搜索，在活动记录中必须储存变量名，而在静态作用域语言的实现中则只要求保存值。（静态作用域不要求变量名，因为所有的变量都由链偏移与局部偏移对表示。）

10.6.2 浅访问

浅访问是一种替代的实现方法，而不是一种替代语义。如前所述，深访问的语义与浅访问的语义是相同的。在浅访问方法中，并不将在子程序中声明的变量储存在这个子程序的活动记录中。因为在动态作用域的方法里，任何一个特定名称的变量在任一给定的时刻最多只能有一个可见版本，因而可以采取一种非常不同的方式。浅访问的一种变化方式是，整个程序中的每一个变量名都有一个分离的栈。每当一个子程序在激活初期的声明中产生一个特定名称的新变量时，便在栈顶给这个变量分配一个装载它的名字的单位。对这个名字的每一个引用，即是对与这个名字相关联的、被置于栈顶的变量的引用。这是由于栈顶的变量是最后才创建的，当一个子程序终止时，它的局部变量的生存期也就结束了，因而这些变量名的栈也即被弹出。这种方法允许非常快速的变量引用，但是在子程序进入与退出时，栈的维护却十分昂贵。

图10-12显示的是上面的第7程序的变量栈，这个程序处于与图10-11所示的同样的情形。

实现浅访问的另一种选择方式是使用一个中央表格，在表格中列出程序中每一个不同变量名的位置。将一个被称为活跃的位与每一个表项一起来维护，这个活跃的位被用来指示变量名是否具有当前绑定，或者具有任何变量的关联。然后，就可以将对任何变量的任何访问表示为表格中的偏移。这种偏移是静态的，因而这种访问可以十分快捷。SNOBOL的实现使用了中央表格的实现技术。



（表单元中的名字指变量声明的程序单元）

图10-12 一种使用浅访问来实现动态作用域的方法

维护中央表格很简单。一个子程序调用将要求将它所有的局部变量合乎逻辑地放置于中央表格中。如果在中央表格中要存放的新变量的位置本来就已经是活跃的，即这个位置包含了一个生存期还没有结束的变量（由活跃位指示），则这个值必须在新变量的生存期内被保存在某个其他地方，每当有一个变量开始它的生存期，都必须在中央表格中设置它的活跃位。

463

中央表格的设计有几种不同的形式，也有几种不同的方式来储存临时被替换的值。一种变化形式是使用一个“垃圾”的栈来保存所有要被储存的东西。因为子程序的调用与返回以及由此局部变量的生存期都是嵌套的，这种表格形式就十分适用。

第二种变化形式实现起来也许最简洁，代价也最低。这种方法使用一种单位的中央表格，仅储存每一个变量名的当前版本。将被替换的变量储存于产生这个替换变量的子程序的活动记录中。这是一种栈机制，因为已经有一个栈存在，新部分的开销因此十分有限。

对于非局部变量，在浅访问与深访问之间的选择取决于子程序调用以及非局部引用的相对频率。深访问方法提供快速的子程序连接，但是对非局部变量的引用，特别是对远程非局部变量（就调用链上的距离而言）的引用，代价较高。浅访问的方法提供了快得多的非局部引用，尤其是对远程非局部变量，但是就子程序连接方面而言，则代价较高。

小结

子程序连接的语义要求许多运行实现的动作。在"简单"子程序的情况下，这些动作是相对简单的。在具有栈动态局部变量以及嵌套子程序的语言中，子程序的连接则较为复杂。

在具有栈动态局部变量以及嵌套子程序的语言中，子程序具有两个组成部分：即静态的实际代码以及栈动态的活动记录。活动记录实例包括形参和局部变量，以及其他一些数据。

静态链是在具有嵌套子程序的静态作用域的语言中实现非局部变量存取的主要方法。

在一种动态作用域语言中，可以通过使用动态链或者中央变量表格的方法来实现对非局部变量的存取。动态链提供了较慢的访问，但却提供了快速的调用与返回。中央表格方法提供了快速的访问，但却提供了较慢的调用与返回。

复习题

1. 为什么实现具有栈动态局部变量的子程序比实现简单子程序更为困难？有哪两条理由？
2. 活动记录与活动记录实例之间有什么差别？
3. 为什么将实间地址、动态连接以及参数放置在活动记录的底部？
4. 在一种具有栈动态局部变量和嵌套子程序的静态作用域语言中，定位一个非局部变量需要哪两个步骤？
5. 定义静态链、静态深度、嵌套深度和链偏移。
6. 静态链方法具有哪两种潜在的问题？
7. 解释当使用显示时，怎样找到一个对非局部变量的引用。
8. 在静态链方法中怎样表示变量的引用？
9. 解释实现块的两种方法。
10. 描述实现动态作用域的深访问方法。
11. 描述实现动态作用域的浅访问方法。
12. 在动态作用域语言中，进行非局部存取的深访问方法，与静态作用域语言中的静态链方法之间，有哪两点不同？
13. 就调用以及非局部存取这两个方面，比较深访问方法与浅访问方法的效率。

练习题

1. 显示当执行到达下面程序框架中的位置1时的栈，连同所有的活动记录实例，并且包括静态链与动态链。假设Bigsub位于层次1。

```
procedure Bigsub is
  procedure A is
    procedure D is
      begin  -- D 开始
      ...  <-------------1
      end;  -- D 结束
    procedure C is
      begin  -- C 开始
      ...
      B;
      ...
```

464

```
    end;  -- C 结束
  begin  -- A 开始
  ...
  C;
  ...
  end;  -- A 结束
begin  -- Bigsub 开始
...
A;
...
end;  -- Bigsub 结束
```

2. 显示当执行到达下面程序框架中的位置1时的栈，遍历所有的活动记录实例，并且包括静态链与动态链。假设Bigsub位于层次1。

```
procedure Bigsub is
  MySum : Float;
  procedure A is
    X : Integer;
    procedure B(Sum : Float) is
      Y, Z : Float;
      begin  -- B 开始
      ...
      C(Z)
      ...
      end;  -- B 结束
    begin  -- A 开始
    ...
    B(X);
    ...
    end;  -- A 结束
  procedure C(Plums : Float) is
    begin  -- C 开始
    ...<-------------1
    end;  -- C 结束
  L : Float;
  begin  -- Bigsub开始
  ...
  A;
  ...
  end;  -- Bigsub 结束
```

465

466

3. 显示当执行到达下面程序框架中的位置1时的栈，遍历所有的活动记录实例，并且包括静态链与动态链。假设Bigsub位于层次1。

```
procedure Bigsub is
  procedure A(Flag : Boolean) is
    procedure B is
      ...
      A(false);
      end;  -- B 结束
    begin  -- A 开始
    if flag
      then B;
      else C;
    ...
    end;  -- A 结束
  procedure C is
```

```
    procedure D is
      ... <-----------------1
      end;  -- D 结束
    ...
    D;
    end;  -- C 结束
  begin  -- Bigsub 开始
  ...
  A(true);
  ...
  end;  -- Bigsub 结束
```

这个程序执行到达D的调用序列为：

```
Bigsub 调用 A
A 调用 B
B 调用 A
A 调用 C
C 调用 D
```

4. 显示当执行到达下面程序框架中的位置1时的栈，连同所有的活动记录实例，包括动态链。这个程序使用深访问方法实现动态作用域。

```
void fun1() {
  float e;
  ...
}

void fun2() {
  int b, c;
  ...
}

void fun3() {
  float d;
  ...<-----------------1
}
void main() {
  char e, f, g;
...
}
```

这个程序的执行到达fun3的调用序列为：

```
main 调用fun2
fun2 调用 fun1
fun1 调用 fun1
fun1 调用 fun3
```

5. 假设练习题4中的程序是使用浅访问来实现的，每个变量名用一个栈。显示当执行到fun3时这些栈的情形。假设是通过练习题4中的调用序列到达这个执行状态的。
6. 虽然每次激活Java方法中的局部变量时，都将这些变量动态地进行分配，但在什么样的情况下，某一次激活的局部变量能可以保持前一次被激活的值？
7. 在本章中曾经指出，当使用动态链在一种动态作用域的语言中存取非局部变量时，必须将变量名和变量值一同保存到活动记录中。如果实际开发中将其付诸实现，则每一次对非局部变量的访问都会要求一系列高代价的、对名字的字符串比较。假设计另一种较快速的字符串比较方法。

8. Pascal允许goto语句具有非局部的目标。如果是实用静态链存取非局部变量，怎样处理这种类型的语句？提示：考虑寻找新近激活过程的静态父辈的正确活动记录实例（参见第10.4.2节）。
9. 可以通过在每个活动记录实例中使用两个静态连接，来实现扩展静态链的方法，其中的第二个静态连接指向静态链父辈的活动记录实例。这种扩展会对子程序连接和非局部引用所需时间产生什么影响？
10. 设计一个框架程序和调用序列，当静态链接和动态链接指向运行时栈中的不同的活动记录实例，该序列产生活动记录实例。

468

第11章　抽象数据类型和封装结构

在这一章里，我们研究程序设计语言中支持数据抽象的结构。在过去的45年中，在程序设计方法学和程序设计语言设计的新思想中，数据抽象是最深刻的思想之一。

我们将从在程序设计和在程序设计语言中的一般抽象概念开始讨论。然后定义数据抽象并且使用一个例子来介绍这种定义。接着将描述在Ada、C++、Java、C#和Ruby中对于数据抽象的支持。并且将给出在Ada、C++、Java、C#和Ruby语言中实现同一数据抽象的例子，以说明在支持数据抽象方面语言设计机制上的相似性及差别。接着将讨论Ada、C++、Java 5.0和C# 2005语言中建立有参数的抽象数据类型的功能。

支持抽象数据类型的结构是一些封装，它们将数据以及在这种类型的对象上的操作封装在一起。大型程序中的结构需要包括多种类型的封装。我们将在这一章中讨论这种类型的封装，以及与这种封装相关联的名称空间。

11.1　抽象概念

抽象是对于实体的一种观念或者实体的一种表示，它仅仅包括这个实体的最重要的属性。在一般情况下，抽象允许人们将实体的实例组织成为一些组合，在这些组合中不需要考虑它们的共同属性。例如，假如我们定义鸟是带有下面属性的动物：两条腿、两个翅膀、一条尾巴和飞行能力。然后如果我们说乌鸦是鸟，那么对乌鸦的描述不需要包括这些属性。对于鸽（robin）、麻雀和胆小的黄腹啄木鸟也是这样。描述鸟的特定种类的这些共同属性能被抽象出来。而在这些组合里，只需要考虑单个元素间有差别的属性，这就大大简化了组中元素。如果有必要了解更多的细节时，就必须考虑这些实体的较低抽象层次上的表示。

在程序设计语言的世界里，抽象是对抗程序设计复杂性的一种武器，其目的是简化程序设计的过程。因为抽象允许程序人员将注意力集中于基本属性而忽略次要的属性，因而它是一种有效的武器。

当代程序设计语言中的两类基本抽象是过程抽象和数据抽象。

过程抽象的概念是程序设计语言设计中最老的概念之一。甚至Plankalkül也曾经支持过程抽象。所有的子程序都是过程抽象，因为它们提供了一种方式，来让一个程序说明所要完成的某些过程，而不是提供如何完成的细节（至少是在调用程序之中）。例如，当一个程序需要将某种类型的数值数据对象数组进行排序时，它通常是使用一个子程序来进行这种排序过程，在程序里需要排序的位置上，通常使用如下语句

```
sortInt(list, listLen)
```

这个调用是实际排序过程的一种抽象，而且排序的算法并不在这里说明。这种调用独立于被调用子程序中的算法实现。

在子程序sortInt的情形中，唯一重要的一些属性是：将要被排序的数组名称，数组元素的类型，数组的维长，以及对于sortInt的调用将产生数组排序的事实。在sortInt中所实现的特殊算法是一种对于用户并不重要的属性。用户只需要知道排序子程序的名称以及协议，以便能够使用这个子程序。

数据抽象必将随着过程抽象的发展而发展，因为每一种数据抽象中的一个不可分割的中心部分，都是被定义为过程抽象的操作。

11.2 数据抽象介绍

从语法的角度而言，抽象数据类型是一个封装，它仅仅包括一种特定数据类型的数据表示，以及一些给这种类型提供操作的子程序。通过访问控制，可以将类型的一些不必要的细节对这个封装以外的、使用这种类型的单位隐藏起来。使用抽象数据类型的程序单位可以声明这种类型的变量，即使这种类型的实际表示对于它们是隐藏的。抽象数据类型的实例称为对象。

数据抽象的动机之一与过程抽象的动机相类似，它是用来对抗复杂性的一种武器；是使管理大型以及复杂程序比较容易的一种方法。数据抽象的其他动机和数据抽象的优越性，将在本节下半部进行讨论。

将在第12章中描述的面向对象程序设计是在软件发展中使用数据抽象所产生的结果，而且数据抽象也是面向对象程序设计中最重要的组成部分之一。

11.2.1 浮点数作为抽象数据类型

至少就内建的类型而言，抽象数据类型的概念并不是近期才发展起来的。所有的内建数据类型，甚至是在Fortran I中的那些内建类型，都是抽象数据类型；虽然人们很少这样来称呼它们。例如，考虑一个浮点数据类型。大多数语言至少包括一种这样的类型，以提供创建浮点数数据变量的方法，而且这些语言还提供一组算术操作来操纵这种类型的对象。

在高级语言中的浮点类型采用了数据抽象中的一个关键概念：信息隐藏。浮点存储单位中数据值的实际格式对于用户是隐藏的。唯一可以使用的一些操作是由语言所提供的操作。除了那些可以用内建操作来构造的操作之外，不允许用户在这种类型的数据上创建新的操作。用户不能直接操纵浮点对象的实际表示部分，因为这种表示是隐藏的。这是允许程序在某种特定语言的各种实现之间进行移植的特性，即使这些实现可能使用的是浮点值的不同表示。例如，在IEEE 754标准浮点表示方法于20世纪80年代出现之前，不同的计算机体系结构使用几种不同的浮点表示方法。然而，这些变体并没有妨碍使用浮点类型的程序在各种不同的体系结构中移植。

11.2.2 用户定义的抽象数据类型

一种用户定义的抽象数据类型应该提供与语言所定义的类型相同的特征。例如浮点类型就具有：(1) 一种类型定义，允许程序单位来声明这种类型的变量，但隐藏这种类型对象的表示。
471 (2) 一组操作，用于操纵这种类型的对象。

我们现在来形式地定义处于用户定义类型环境中的抽象数据类型。一种抽象数据类型是满足下面两个条件的数据类型：

- 将提供类型接口的这种类型的声明和在这种类型对象上的操作协议包含在一个语法单位之中。类型的接口并不依赖于这个对象的表示或这些操作的实现。可以将这个类型及其操作实现于声明所在的同一个语法单位里，也可以实现在另外给出的一个语法单位中。另外，还允许其他的程序单位创建所定义类型的变量。
- 这种类型对象的表示，对使用这种类型的程序单位是隐藏的，这样，在这些对象上所可能的直接操作就是类型定义中所提供的那些操作。

这种将类型的声明及其操作包装在一个语法单位中的主要优点，是提供了将程序组织成为一些可以分别编译的逻辑单位方法。将类型及其操作实现在另外一个语法单位中的优点，是这

种做法能够将说明与实现保持分离。使用某一种特殊抽象数据类型的一些程序单位，被称为这种类型的客户（client）。这些程序单位需要类型的说明可见，但却不被允许看见这种类型的实现。如果将声明和类型及其操作定义都放置于同一个语法单位之中，就必须有某种特殊的方式将指定定义单元的部分对于客户隐藏起来。 472

类型的接口独立于对象的表示或独立于操作实现的好处是：它允许可以随时改变这些表示，而不需要对类型的客户进行修改。

信息隐藏的好处是提高可靠程度。客户不能够有意或无意地直接操纵对象的内部表示，这样就增强了对象的整体性。只有通过所提供的操作才能够改变对象。

11.2.3 示例

试想构造一个栈的抽象数据类型，它具有以下这些抽象操作：

create(stack)	创建并可能初始化一个栈对象
destroy(stack)	解除分配栈的存储空间
empty(stack)	一个谓词（或布尔）函数，如果所说明的栈是空的，它的返回值为真，否则，它的返回值为假
push(stack, element)	将被说明的元素推入所说明的栈中
pop(stack)	将顶端的元素移出所说明的栈
top(stack)	从所说明的栈中返回顶端元素的副本

注意，一些抽象数据类型的实现并不要求创建与解除这两种操作。例如，定义一个变量为一种抽象数据类型，可能就隐式地创建了这个变量的数据结构，并且还隐式地将它初始化。这种变量的存储空间可能会在该变量的作用域终结时隐式地解除分配。

这种栈类型的一个客户可以具有下面的代码序列：

```
...
create(stk1);
push(stk1, color1);
push(stk2, color2);
if(! empty(stk1))
  temp = top(stk1);
...
```

假设栈抽象的初始实现使用了一种邻接表示（一种在数组中实现栈的表示方法）。但后来，因为邻接表示所存在的存储器的管理问题，又将邻接表示改变为链表示。因为使用了数据抽象，这种改变可以在定义栈类型的代码上进行，而不要求栈抽象的任何客户进行改变。尤其是，不需要改变上例中的代码序列，当然，任何一种在操作协议上的改变都将要求在客户中进行改变。 473

11.3 抽象数据类型的设计问题

在一种语言中定义抽象数据类型的机制，必须提供一种语法单位，这个语法单位包含了抽象操作的类型定义以及子程序定义。必须使类型名称和子程序的首部对于这种抽象的客户为可见的。这样就允许了客户来声明这种抽象类型的变量，并且可以操纵这些变量的值。虽然类型名称必须具有外部可见性，但却必须将类型定义隐藏起来。

除了那些类型定义中所提供的操作外，应该给抽象数据类型的对象提供少数的、通用的内建操作（如果有的话）。实际上，很少有适用广泛的抽象数据类型的操作，仅有的这些操作包括赋值和进行等于与不等于的比较。如果这种语言不允许用户将赋值操作重载，就必须将赋值设

置为内建的。在某些情形下，应该预定义等于与不等于比较的操作，但在其他的情形下则不然。例如，如果将这种类型实现为指针，等于则可能意味着指针的相等，但是用户可能希望的是，它应该意味着指针所指向的结构相等。

某些操作可能为大多数的抽象数据类型所需要，但是因为这些操作不是通用的，所以必须由类型的设计人员来提供。在这些操作中包括：迭代器、存取器、构造器以及析构器。迭代器曾经在第8章讨论过。存取器提供了对于隐藏于客户而不能够直接存取的数据的一种存取形式。构造器用于为新创建对象的某些部分设定初值。析构器则用来回收被抽象数据类型对象的部分所使用过的准存储空间。

如前所述，一个抽象数据类型的封装定义了单数据类型及其操作。许多当代语言，包括C++、Java以及C#，都直接支持抽象数据类型。一种替代的方法是提供一种可以定义任何数目的实体的。更一般化的封装结构，对于其中的任何实体，都可以有选择地说明为在封装单位以外可见。这就是 Ada 中所使用的方式。这些封装不是抽象数据类型，而是抽象数据类型的一般化。这样，就可以使用它们来定义抽象数据类型。虽然我们在这一节中讨论了Ada的封装结构，但我们将它处理成一种用于单数据类型的小型封装。关于通用的封装是11.6节的讨论课题。

抽象数据类型的首要设计问题是，是否可以将它们参数化。例如，如果这种语言支持有参数的抽象数据类型，人们就可以设计一种可以存储任何标量类型元素的队的抽象数据类型。

关于有参数的抽象数据类型，将在11.5节中讨论。另外的一个设计问题是，提供什么样的存取控制和怎样来说明这种控制。

474

访谈



C++语言：诞生、广泛被人们接受和遭受到的批评

BJARNE STROUSTRUP

Bjarne Stroustrup是C++的设计者和最早的实现者，也是《C++程序设计语言》和《C++的设计与发展》的作者。他的研究兴趣包括分布式系统模拟、设计、程序设计以及程序设计语言。Stroustrup原来是美国得克萨斯州A&M大学工学院的计算机科学教授。他和极多与了ANSI/ISO的C++标准化。在A&M大学工作了二十多年之后，他仍然保持着与AT&T实验室的联系，作为信息与软件系统研究实验室的成员来从事科学研究。他是ACM的高级成员，AT&T贝尔实验室的高级成员，AT&T的高级成员。在1993年，Stroustrup荣获了ACM的Grace Murray Hopper奖。嘉奖他早年对C++程序设计语言的奠基工作。Stroustrup博士在先前基础上的继续努力，使得C++成为计算机历史上最有影响力的程序设计语言。

1. 你与计算机的简略历史

问：你在20世纪80年代初加入贝尔实验室之前，曾在做些什么，在什么地方？

答：在贝尔实验室，我在分布式系统的一般领域从事研究工作。我是1979年加入贝尔的。在那之前，我在剑桥大学完成博士学位。

问：你马上就开始了在“带类的C”方面的工作（“带类的C”后来成了C++）吗？

答：在开始“带类的C”之前，在开发“带类的C”和C++的过程中，我在与分布式系统有关的几个项目中工作。例如，我试图找到一种方法将UNIX内核分布到几台计算机上。还有就是帮助很多项目建立模拟器。

问：是不是由于对数学的兴趣使你进入了这个职业？

答：我本科修的专业是“数学与计算机科学”。读硕士时研究数学。那时，我请误地认为计算（computing）就是某一种应用数学。我学习了两年数学后还发现自己是一个很糟糕的数学家，但还是从中学到了很多东西。在我学习时，从来就没有见过计算机。我所喜欢的是程序设计，而不是那些更偏向于数

学的领域。

Ⅱ 剖析一门成功的语言

我喜欢从后往前：先列出一些我认为使得C++成功的因素，再听听你的看法。这种顺序就是“开放源代码”，非专利，以及ANSI/ISO的标准化。

ANSI/ISO的标准化很重要。当时存在着几种独立开发和发展的C++实现。如果没有一种标准来让它们遵守，如果没有一个标准的过程来协调C++的发展，C++方言中将会爆发出一场混乱。

在有“开放源代码”的实现同时又有商业化的实现，这也很重要。另外对于很多用户十分关键的是，C++的标准将保护他们不受语言实现者的操纵。

ISO标准化过程是公开和民主的。C++的委员会很少在少于50个人时开会。典型的是，每一次会议，与会代表都来自八个以上的国家。它不仅仅是一个贵家的论坛。

C++是系统程序设计的理想工具（在C++诞生时，系统程序设计是开发代码市场中最大的一个部分）。

的确，C++是任何系统程序设计项目的有力竞争者。它也是嵌入式系统程序设计的有效工具。嵌入式系统是当前发展最快的一个领域。C++的另一个领域是高性能的数值/工程/科学的程序设计。

C++的面向对象的特征和它对于类或库的引入，使得程序设计具有更高的效率和透明性。

C++是一种多风格的程序设计语言，即它支持多种基本的程序设计风格（包括面向对象程序设计）以及这些风格的结合。当使用得当时，将导致比使用任何单独的程序设计风格更为简洁、灵活和高效率的程序库。一个例子就是关于C++的标准程序库和算法，这基本上是一个通用的程序设计框架。当这种框架与（面向对象的）类层次结构一起使用时，结果就导致了类型安全性、高效率和灵活性的卓越结合。

C++在AT&T的开发环境中的孵化。

AT&T贝尔实验室所提供的环境对C++的开发十分的关键。实验室是挑战性问题的一个异常丰富的来源，并且是实用研究的有力支持者。C++与C诞生于同一研究实验室，并得益于同一知识传统、经验，以及同一批杰出的研究人员。自始至终，AT&T都支持C++的标准化。然而，与很多当代语言不同的是，C++不是推出倒海的市场攻势的受惠者，因为这些本来就不是实验室的工作方式。

我有没有什么最重要的遗漏？肯定会有。

现在，让我再来谈谈对于C++的批评，以及还想听听你的看法：C++庞大且笨重。它的“hello world”程序比C中的大十倍。

C++肯定不是一种小型语言。但没有什么当代语言是小型语言。如果一种语言很小的话，你就会需要有巨大的程序库才行。通常，你得依赖于协定和扩展。对于那些有着不可避免的复杂性的关键部分，我宁可把它们包括在语言之中而不是藏在系统中的别的什么地方。在语言中，它们是可见的，能够被学习并且能够被有效地标准化。对于大多数的目的，我并不认为C++是笨重的。在我的机器上，C++的“hello world”程序并不比C中的更大；在你的机器上也不应该更大。事实上，在我的机器上的“hello world”C++程序的目标代码比C中的还小。为什么一种语言的程序会比另一种语言的小，这并不是语言本身的原因，这完全是实现者如何组织程序库的问题。如果一种语言的程序会比另外一种语言大很多，就应该向这种语言的实现人员报告这个问题。

批评者说，较之C，使用C++编写程序比较困难，甚至你也承认过这一点，说如果用C来对比C++的话，是“自找麻烦——自己射自己的脚”。

是的，我的确说过：“用C你容易射到自己的脚，然而使用C++的话就没那么容易了；但是如果你真的射到了，你会将整条腿都炸飞的”。这里我说的是关于C++，但却在不同程度上适用于所有功能强大的语言，当你保护人们使之不至遇到简单的危险时，他们却会碰到新的和不那么明显的问题。一个避免了简单问题的人，很可能就会遇到不那么简单的问题。一个具有保护能力的环境的缺点，可能是发现困难时太晚的问题；而且一旦发现了问题，却很难进行修复。况且，罕见的问题比常见的问题更难以发现，而因是你不容易怀疑它。

C++对今天的嵌入式系统是合适的，但对今天的互联网软件却是不合适的。C++适合于今天的嵌入式系统。

C++非适合于开发广泛地应用于今天的“万维网软件”，例如，看看我的C++Application网页，你会注意到一些万维网服务的主要供应者，例如Amazon、Adobe、Google、Quicken和Microsoft，都在关键之处依赖于C++。计算机游戏也是大量应用C++的领域。

478
-
479

我有没有遗漏什么其他的重要的批评？当然也有。

11.4 语言示例

抽象数据类型概念的原始形态是在SIMULA 67语言之中，虽然这种语言仅仅部分地支持抽象数据类型。在这一节中我们将描述Ada、C++、Java、C#和Ruby语言中对于数据抽象的支持。

11.4.1 Ada中的抽象数据类型

Ada提供一种能够用来定义单抽象数据类型的封装结构，包括隐藏其表示的功能。Ada是第一种提供对于抽象数据类型全面支持的语言。

11.4.1.1 封装

Ada 中的封装结构，被称为包（package）。包可以有两个部分，其中的每一个部分也被称为包：其一为说明包（specification package），它提供封装的接口。另外一个为体包（body package），提供在说明中所命名的大多数实体的实现，如果不是关联的说明包中全部的实体实现的话。并不是所有的包都具有体包部分（仅仅包含类型与常量的包，不具有或者是不需要有体包）。

一个说明包和与其相关联的体包，共享同一个名字。在一个包的首部中，保留字body将这个包标识为一个体包。说明包与体包，可以被分开编译，但必须首先编译说明包。

11.4.1.2 信息隐藏

定义一种数据类型的Ada包的设计人员，可以选择是使得这种类型对于客户完全可见，还是仅仅提供接口信息。当然，如果没有将类型的表示隐藏的话，那么这种定义的类型就不是一种抽象数据类型。有两种方式将说明包中的类型表示对于客户隐藏起来。一种方式是通过提供说明包的两个段来施行：一个段中的实体对于客户是完全可见的，而另外一个段中的内容则对客户隐藏起来。对于一种抽象数据类型，一种缩写的声明将出现在说明中的可见部分，这个声明仅仅提供类型名称和这种类型表示被隐藏的事实。类型表示出现在说明中的部分被称为私有部分，它由保留字private引入。private子句总是位于说明的末尾。

将表示隐藏起来的第二种方式，是将抽象数据类型定义为指针，并且在体包中提供指向结构的定义。将所有这些内容都对于客户隐藏起来。

下面是前面这种方式，将类型表示对于客户隐藏起来的一个例子。假设将在一个包中定义一个命名为Node_Type的抽象数据类型。Node_Type将被声明于说明包中的可见部分，但没有表示的细节，如下列语句：

475

```
type Node_Type is private;
```

将被声明为私有的类型称为私有类型。私有数据类型具有内建的赋值以及等于与不等于的比较操作。任何其他操作，都必须在定义这种类型的说明包中声明。

注意，在下面例子中的private子句中，对于Node_Type的声明是重复的，但是这一次，具有完整的类型定义。

```
package Linked_List_Type is
```

```
type Node_Type is private;
...
private
  type Node_Type;
  type Ptr is access Node_Type;
  type Node_Type is
    record
    Info : Integer;
    Link : Ptr;
    end record;
end Linked_List_Type;
```

注意在这个包中的private子句里，既有Node_Type的声明，又有Node_Type的定义，因为在Ptr的定义中对于Node_Type的引用，这个引用又前置于Node_Type的定义，因而这个声明是必要的。因为是被定义于private子句之中，对于Linked_List_Type的客户，Info和Link都是不可见的。

如只在一个包中没有实体需要隐藏，就没有必要包括说明中的私有部分。当然，这样的一种包就不能够定义一种抽象数据类型。

类型表示之所以会出现在说明包中，其原因是与编译问题紧密相关的。客户只能看见说明包（而不是体包），但编译器必须能够在编译客户时，给输出类型的对象分配存储空间。此外，只有当抽象数据类型的说明包存在并且已经被编译了时，客户才是可编译的。也就是，编译者必须能够确定来自说明包的对象的大小。所以类型的表示对于编译器必须为可见的，但对于客户代码则是不可见的。这恰恰是说明包中的private子句所说明的情形。

在上述例子中，说明包提供部分实现细节（关于数据的定义），而实现的其他细节（关于操作的定义）却在体包之中。这确实在某种程度上带来了麻烦。如果说明包仅仅提供接口，而体包则提供实现的所有细节，就会清晰得多。通过将抽象数据类型设计成指针的方式，将会消除这种问题，如下面的定义：

```
package Linked_List_Type is
  type Node is private;
  function Create_Node() return Node;
  private
    type Node_Record;
    type Node is access Node_Record;
end Linked_list_Type;
```

现在就可以将所有的实现细节都在体包中给出，如下所示：

```
package body Linked_List_Type is
  type Node_Record is
    record
      Info : Integer;
      Link : Ptr;
    end record;
    ...
end Linked_List_Type;
```

这种稍微清晰的版本包存在着几个问题。第一，处理指针有一些内在的困难。第二，在新的抽象数据类型的一些对象之间进行的比较，将会成为指针之间的比较，这种比较不会产生所期望的结果，因为比较的是指针，而不是指针所指向的那些对象。将抽象数据类型定义为指针的另外一个问题是类型不能够控制这种类型的对象的分配与解除分配。例如，一个客户可以产生一个指向对象的指针（通过一个变量的声明），而并没有产生一个对象。

私有类型的一种替代类型是一种更受限制的形式：**受限私有类型**。同非指针型的私有类型一样，受限私有类型在说明包的私有段中给予描述。这里唯一的语法差别是，将受限私有类型在说明包中的可见部分声明为limited private。一种被声明为受限私有类型的对象，不具有内建的操作。当预定义的赋值操作和比较操作没有意义或是没有用时，这样的一种类型就很有用。例如，对于栈很少使用赋值和比较操作。但是如果需要赋值和相等性的比较操作但又不能够使用内建的版本时，就必须由说明包来提供这些操作。这是当抽象数据类型为指针时，避
477 免比较问题的一种方式。赋值操作必须是一种正常的过程形式，而等于和不等于操作符，则可以通过重载新类型的那些操作符来提供。

11.4.1.3 示例

下面是一个栈抽象数据类型的说明包：

```
package Stack_Pack is
-- 可见部分，或者是公有部分
  type Stack_Type is limited private;
  Max_Size : constant := 100;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
                 Element : in Integer);
  procedure Pop(Stk : in out Stack_Type);
  function Top(Stk : in Stack_Type) return Integer;
-- 这一部分对客户隐藏
  private
    type List_Type is array (1..Max_Size) of Integer;
    type Stack_Type is
      record
      List : List_Type;
      Topsub : Integer range 0..Max_Size := 0;
      end record;
  end Stack_Pack;
```

注意，这里没有包括创建或者解除的操作，因为它们不是必需的。

Stack_Pack的体包是：

```
with Ada.Text_IO; use Ada.Text_IO;
package body Stack_Pack is
  function Empty(Stk: in Stack_Type) return Boolean is
    begin
    return Stk.Topsub = 0;
    end Empty;

  procedure Push(Stk : in out Stack_Type;
      Element : in Integer) is
    begin
    if Stk.Topsub >= Max_Size then
      Put_Line('ERROR - Stack overflow');
    else
      Stk.Topsub := Stk.Topsub + 1;
      Stk.List(Topsub) := Element;
    end if;
  end Push;
480 procedure Pop(Stk : in out Stack_Type) is
  begin
  if Stk.Topsub = 0
    then Put_Line('ERROR - Stack underflow');
```

```
    else Stk.Topsub := Stk.Topsub - 1;
  end if;
  end Pop;

function Top(Stk : in Stack_Type) return Integer is
  begin
  if Stk.Topsub = 0
    then Put_Line("ERROR - Stack is empty");
    else return Stk.List(Stk.Topsub);
  end if;
  end Top;
end Stack_Pack;
```

这个体包的第一行代码包括了两条语句：一条with语句和一条use语句。这条with子句使得定义在外部包中名字成为可见的，在这个例子中是Ada.Text_IO，它提供文本输入与输出的函数。这条use子句消除了对于命名包中的实体引用，进行显式限定的需要。对于外部封装的存取以及名字限定的问题，将在11.6节中进一步讨论。

体包中的子程序定义的首部，必须与相关的说明包中子程序的首部相匹配。这个说明包承诺，这些子程序将会被定义在相关的体包之中。

下面的过程Use_Stacks是包Stack_Pack的一个客户。我们用它来说明怎样使用这种包。

```
with Stack_Pack;
use Stack_Pack;
procedure Use_Stacks is
  Topone : Integer;
  Stack : Stack_Type;    -- 声明一个Stack_Type的对象
  begin
  Push(Stack, 42);
  Push(Stack, 17);
  Topone := Top(Stack);
  Pop(Stack);
  ...
  end Use_Stacks;
```

对于大多数当代语言来说，因为它们的标准类库包含了对栈的支持，所以使用栈是不明智的。但是，栈提供了一个简单的例子，以允许比较本节讨论的语言。 [481]

11.4.2 C++中的抽象数据类型

C++是通过将一些特性加入C语言中而创造的。所增加的第一种重要特性，是支持面向对象程序设计。因为面向对象程序设计的主要成分之一是抽象数据类型，所以显然，C++就必须支持它们。

Ada提供能够用来实现抽象数据类型的包，C++提供两种相互类似的结构：类（class）和结构（struct），可更为直接地支持抽象数据类型。当仅仅涉及数据时，C++中的结构被最为普遍地使用，我们将不在这里做进一步的讨论。

C++中的类是类型，而如前所述的Ada中的包则是更为一般化的封装，它可以定义任意数目的抽象数据类型。当一个程序单位获得了对于一个Ada包的可见性时，便可以直接通过名字来访问这个包中的任何公有（public）实体。而当一个C++的程序单位声明了某个类的一个实例时，也可以访问这个类中的任何公有实体，但仅仅能够通过这个类中的实例来进行。这是一种

提供抽象数据类型的更为清晰与直接的方式。

11.4.2.1 封装

在C++的类中所定义的数据，被称为**数据成员**（data member），而在一个类中所定义的函数就被称为**成员函数**（member function）。数据成员和成员函数出现在两种类型，类和实例中。类成员与类相关联。实例成员与类的实例相关联。本章只讨论类的实例成员。一个类中的所有实例，共享一套成员函数，但每一个实例都有它自己的一套类的数据成员。类的实例可以是栈动态的，或者是堆动态的。如果是栈动态的，可以通过值变量来引用。如果是堆动态的话，则通过指针来引用。栈动态的类的实例，总是通过对象声明的确立而创建的。此外，当到达了声明的作用域末尾时，这种类实例的生存期便也终结。堆动态类对象可以通过new操作符创建，通过delete操作符销毁。栈动态和堆动态类都能有引用堆动态数据的指针数据成员。因此尽管类实例是栈动态的，但是它能包括引用堆动态数据的数据成员。

一个类的成员函数，可以通过两种不同的方式来定义：完整的定义可以出现在类中或仅仅出现在它的首部。当一个成员函数的首部和函数体都出现在类的定义中时，这个成员函数就是隐式内联的。这意味着它的代码被放置在调用程序之中，而不需要一般的调用与返回的连接过程。如果只有成员函数的首部出现在类的定义中，它的完整定义就将出现在类的外部，并且是被分别编译的。允许合理内联成员函数将为实时应用程序节省链接时间，运行时效率是最重要的。内联成员函数的缺点是聚集了类定义界面，从而减少了可读性。

11.4.2.2 信息隐藏

C++中的类可以包含隐藏的和可见的两种实体（指它们对类的客户是可见还是隐藏的）。将隐藏的实体放置在private子句中，而将可见的实体或者是公有实体写进一个public子句里。因而public子句描述的是与类中对象的接口。还有第三种类型的可见性，即被保护的（protected）。关于这种可见性，将在第12章中讨论继承时再进行讨论。

C++允许用户在类的定义中包括构造器函数，这种函数被用来给新创建对象中的数据成员设定初值。构造器也可以给新对象的指针引用的堆动态数据成员分配空间。当创建一个类的对象时，构造器被隐式地调用。一个构造器的名字与它施行初始化的对象所属的类的名字相同。构造器可以被重载，然而，一个类中的每一个构造器都必须具有唯一的参数剖绘。

C++中的类还可以包括一个被称为析构器的函数，当这个类的实例生存期终止时，则隐式地调用析构器。如前所述，栈动态类的实例可以包含堆动态的数据成员。这种实例的析构器函数，包括了一个用于堆动态成员的delete操作符，来对于这些成员施行堆空间的解除分配。析构器也常常被用作程序调试的一种辅助工具。在这种情况下，在对象的数据成员被解除分配之前，析构器仅仅是显示或打印这个对象的一些数据成员或全部数据成员的值。一个析构器的名字也是类的名字，只在前面放置"~"号。

构造器和析构器都没有返回类型，也都不使用return语句。它们都可以被显式地调用。

11.4.2.3 示例

再一次，我们的C++中抽象数据类型的例子也是一个栈：

```
#include <iostream.h>
class stack {
  private:  //** 这些成员，仅仅关于其他的成员以及友元类为可见的（参见11.6.4节）
    int *stackPtr;
    int maxLen;
    int topPtr;
  public:   //** 这些成员对于客户为可见的
```

```
    stack() {  //** 一个构造器
      stackPtr = new int [100];
      maxLen = 99;
      topPtr = -1;
    }
    ~stack() {delete [] stackPtr;};  //** 一个析构器
    void push(int number) {
      if (topPtr == maxLen)
        cerr << "Error in push--stack is full\n";
      else stackPtr[++topPtr] = number;
    }
    void pop() {
      if (topPtr == -1)
        cerr << "Error in pop--stack is empty\n";
      else topPtr--;
    }
    int top() {return (stackPtr[topPtr]);}
    int empty() {return (topPtr == -1);}
}
```

我们将只讨论这个类定义的少数几个方面，因为并不需要了解这段代码的所有细节。类stack具有三个数据成员：stackPtr，maxLen以及topPtr，所有这三个成员都是私有的。它还具有四个公有的成员函数：push，pop，top和empty。还有一个构造器和一个析构器。这个构造器使用分配操作符new，来从堆上分配一个具有100个int类型元素的数组。它还设定maxLen和topPtr的初值。这个析构器函数的目的，是当stack类对象的生存期终止时，对用于实现这个栈的数组的存储空间，进行解除分配，这个数组是由构造器来实施分配。因为这个类的定义包括了成员函数体，它们都是隐式内联的。

一个使用stack抽象数据类型的示例程序如下：

```
void main() {
  int topOne;
  stack stk;  //** 产生stack类的一个实例
  stk.push(42);
  stk.push(17);
  topOne = stk.top();
  stk.pop();
  ...
}
```

11.4.2.4 评估

C++通过它的类结构来支持抽象数据类型。这种表达能力与Ada中通过包的表达能力相类似。 481
这两种语言都提供了封装和抽象数据类型信息隐藏的有效机制。它们之间的主要差别在于类是类型，而Ada中的包是更一般化的封装。此外，如将在第12章里所要讨论的，类的设计超出了数据抽象的功能。

11.4.3 Java中的抽象数据类型

Java对于抽象数据类型的支持与C++中的十分类似。然而又有一些重要的不同。在Java中所有用户定义的数据类型，都是类（Java中不具有结构），而且所有的对象都是从堆上分配，并通过引用变量来存取。Java与C++在对抽象数据类型的支持上，另外一个不同之处是：Java的方法必须在类中完整定义。方法体必须与相应的方法头一起出现。因而，Java中的一个抽象数据类

型总是同时被声明于也被定义于同一个语法单位之中。通过规定抽象数据类型的定义为私有的，而将这些定义对于客户隐藏起来。

Java的类定义中没有私有及公有子句。在Java中，可以将存取修饰符附加到方法以及变量的定义之上。

下面是用于我们的栈示例的Java类定义：

```
import java.io.*;
class StackClass {
  private int [] stackRef;
  private int maxLen,
              topIndex;
  public StackClass() {  // 一个构造器
    stackRef = new int [100];
    maxLen = 99;
    topIndex = -1;
  }
  public void push(int number) {
    if (topIndex == maxLen)
      System.out.println("Error in push-stack is full");
    else stackRef[++topIndex] = number;
  }
  public void pop() {
    if (topIndex == -1)
      System.out.println("Error in pop-stack is empty");
    else --topIndex;
  }
  public int top() {return (stackRef[topIndex]);}
  public boolean empty() {return (topIndex == -1);}
}
```

485

下面是一个使用StackClass类的例子：

```
public class TstStack {
  public static void main(String[] args) {
    StackClass myStack = new StackClass();
    myStack.push(42);
    myStack.push(29);
    System.out.println("29 is: " + myStack.top());
    myStack.pop();
    System.out.println("42 is: " + myStack.top());
    myStack.pop();
    myStack.pop();  // 产生一个出错信息
  }
}
```

一个明显的区别是，在Java版本中不存在析构器，它由Java中的隐式废料收集来代替。

我们的示例没有介绍很多关于C++与Java之间在支持抽象数据类型方面的重要的差别。然而，正如将在11.6节中讨论的，在C++的多类型封装与Java的多类型封装之间，存在着较大的差别。此外，当考虑到面向对象程序设计的其他方面时，如将在第12章中进行的，在C++中的类与Java中的类之间还存在着更大的差别。

11.4.4 C#中的抽象数据类型

C#语言是基于C++以及Java语言的，它还包括了一些新的结构。

在C#中使用存取修饰符private、public和protected[1]的方式，与在Java中使用存取修饰符的方式完全一样。然而，C#中还包括了两种额外的修饰符：internal和protected internal，关于internal修饰符，将在讨论一般化封装的11.6节中给予描述。

与Java中的一样，C#中的类实例都是堆动态的。给实例数据提供初始值的默认构造器，可以为所有的类所使用。这些构造器提供典型的初始值，如int类型中的0，boolean类型中的false等。用户可以为他或她所设计的类，构造一个构造器。这样一个构造器，可以给所设计的类中的一些或所有的实例数据赋初始值。任何在用户定义的构造器中没有被赋初值的实例变量，将由默认构造器赋以初始值。

286

因为C#对于它的大多数堆对象，都使用垃圾收集，因而基本上不使用析构器。

虽然抽象数据类型的基本原则规定，对象的数据成员应该对客户隐藏，但许多的实际情形导致客户必须存取这些数据成员。通用的解决办法是提供一些存取器方法：获取器和设置器，这样就允许间接地存取所谓的隐蔽数据。尽管客户可以通过获取器和设置器来存取，较之仅仅是将数据定义为公有的从而提供直接地存取方式，这种间接方式更为优越。说明存取器的方式较为优越的理由，主要有如下几点：

1. 通过定义获取器方法而不定义设置器方法，能够提供只读的存取。

2. 在设置器中可以设置一些限制。例如，如果需要将数据值限制在某一个特定的范围中，设置器方法可以强制实行。

3. 如果获取器和设置器为唯一的存取方式，就有可能改变数据成员的实际实现，而不会影响到客户。

C#从Delphi语言继承了属性，作为不需要显式的调用实现获取器和设置器的一种方式。属性提供了对于私有的实例数据的隐式存取。例如，考虑以下简单类和客户的代码：

```
public class Weather {
  public int DegreeDays {   //** DegreeDays是一个属性
    get {
      return degreeDays;
    }
    set {
      if(value < 0 || value > 30)
        Console.WriteLine(
            "Value is out of range: {0}", value);
      else
        degreeDays = value;
    }
  }
  private int degreeDays;
  ...
}
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

[1] 第12章将讨论protected存取修饰符。

在类Weather中定义了属性DegreeDays，这个属性提供了一个获取器方法以及一个设置器方法，用于存取私有的数据成员degreeDays。在类定义后的客户代码中，对于degreeDays的处理方式，就像它是一个公有的成员变量一样。虽然只有通过属性才能够存取degreeDays，注意在设置方法中隐式变量value的使用。这正是使用属性中新的值的机制。

如曾在11.4.2节中所提到的，C++中包括了类与结构这两种近乎相同的结构。它们之间的唯一差别是用于类的默认存取修饰符是private，而用于结构的默认存取修饰符是public。C#中也具有结构（struct）。但这些结构与在C++中的十分不同。在C#中的结构，在某种程度上是轻量级的类。这些结构可以具有构造函数、属性、方法以及数据领域，并且还可以实现接口，然而却不支持继承。在C#中的结构与类之间另外的一种根本区别是结构为数值类型而不是引用类型。结构被分配于运行时的栈上，而不是被分配于堆上。如果将这些结构作为参数来传递，它们是按值传递的。C#中的所有数值类型，包括它的所有基本类型，实际上都是结构。虽然这样看起来十分奇怪，通过使用new操作符来创建结构对象，就与创建类对象的方式一样。

在C#中所使用的结构，主要用于实现相对小型的类型，这些类型不是用于继承的基本类型。当类型的对象是在栈上而非堆上分配时，使用这些结构也较为方便。

11.4.5 Ruby的抽象数据类型

Ruby对它的类提供了完整的抽象数据类型支持。Ruby类与C++和Java类非常相似。

在Ruby里，通过使用开头带class保留字的复合语句来定义类。局部变量名有其他程序设计语言中变量名的形式。实例变量名以标记@开始。类有以两个标记@@开头作为名称的类变量（它们与类关联，而不是实例）。实例方法有与Ruby函数同样的语法。它们以保留字def开头，以end结尾。类方法与实例方法不同之处是实例方法名由类方法名前加分隔符组成。例如，在Stack类中，类方法名将以Stack开头。Ruby的构造函数命名为initialize。它们可以通过定义多个具有不同数量的参数的原型来重载。

类成员能标记为私有或公有（默认下为公有）。[6]Ruby的私有和公有访问权限与Java完全相同。所有的数据成员必须指定为私有以支持信息隐藏。

Ruby的类是动态的，这意味着成员能够动态添加。简单地包括指定新成员的附加类定义就可以完成动态添加。当然，方法也可能从类中删除。这可以通过提供另一个类定义，即把要删除的方法作为参数发送给方法remove_method来完成。Ruby的动态类是语言设计人员把可读性（和可靠性）换为灵活性作为语言取舍的另一个例子。允许类的动态改变虽而易见地增加了语言的灵活性，但却减小了其可读性。为了决定当前类的定义方式，我们必须发现程序中所有的定义并通盘考虑它们。

下面例子是用Ruby语言编写的栈：

```
# Stack.rb - 定义并实现一个最大长度为100，由数组实现的栈
#

class StackClass

# 构造器

  def initialize
    @stackRef = Array.new
```

[6] Ruby也支持保护访问模式，这将在第12章中讨论。

```
    @maxLen = 100
    @topIndex = -1
  end

# push方法

  def push(number)
    if @topIndex == @maxLen
      puts "Error in push - stack is full"
    else
      @topIndex = @topIndex + 1
      @stackRef[@topIndex] = number
    end
  end

# pop方法

  def pop
    if @topIndex == -1
      puts "Error in pop - stack is empty"
    else
      @topIndex = @topIndex - 1
    end
  end

# top方法

  def top
    @stackRef[@topIndex]
  end
#empty方法
  def empty
    @topIndex == -1
  end
end  # of Stack class

#StackClass的测试代码

myStack = StackClass.new
myStack.push(42)
myStack.push(29)
puts "Top element is (should be 29): #{myStack.top}"
myStack.pop
puts "Top element is (should be 42): #{myStack.top}"
myStack.pop

#下面的pop应该产生一个
#错误指出一栈为空
```

回忆一下，注释#{变量}转换变量值为字符串，然后把它插入到字符串出现的地方。该类定义了能存储任何类型对象的栈结构。前面讲过，Ruby中的一切都是对象，数组实际上是对对象引用的数组。显而易见地，这使上面的栈比Ada、C++和Java例子中出现的栈更灵活。而且，通过简单地把要求的最大长度传递给构造器，此类的对象能具有任何给定的最大长度。当然，因为Ruby数组的长度能动态改变，所以类能改变实现栈对象的任何长度，除非机器存储器容量的限制。

11.5 有参数的抽象数据类型

如果能将抽象数据类型参数化常常是很方便的。例如，我们应该能够设计一个栈抽象数据类型，它可以存储任何数量类型的元素；而不应该被要求为每一个不同的数量类型，都编写一个单独的栈抽象。在下面的两节中，我们将讨论 Ada、C++、Java 5.0和C# 2005中构造有参数的抽象数据类型的能力。

489 - 490

11.5.1 Ada

关于Ada中的通用过程，也曾经在第9章中讨论过。包也可以是通用的，因而我们也可以构造通用的或是有参数的抽象数据类型。

曾在11.4.1节中所显示的，Ada的栈抽象数据类型示例，受到两种限制：(1) 这种类型的栈只能够存储整数类型的元素；(2) 这种栈最多只能具有100个元素。通过使用一个通用包，就能够消除这两种限制，从而栈可以被实例化为其他的元素类型以及任何期望的大小。(这是一种通用的实例化；这种实例化与一个类创建对象的实例化十分不同。) 下面的说明包描述一个具有这些特性的通用栈抽象数据结构的接口：

```
generic
  Max_Size : Positive;  --  一种记录栈的大小的通用参数
  type Element_Type is private;  --  一种用于元素类型的通用参数
package Generic_Stack is
-- 可见的实体，或者是公有的接口
  type Stack_Type is limited private;
  function Empty(Stk : in Stack_Type) return Boolean;
  procedure Push(Stk : in out Stack_Type;
                 Element : in Element_Type);
  procedure Pop(Stk : in out Stack_Type);
  function Top(Stk : in Stack_Type) return Element_Type;
-- 隐藏的部分
private
  type List_Type is array (1..Max_Size) of Element_Type;
  type Stack_Type is
    record
    List : List_Type;
    Topsub : Integer range 0..Max_Size := 0;
    end record;
  end Generic_Stack;
```

Generic_Stack的体包与在前一节中的Stack_Pack的体包相同，除了在Push和Top 中形参Element的类型是Element_Type而不是Integer以外。

下面这条语句将Generic_Stack实例化为一个具有100 个元素的Integer类型的栈：

491

```
package Integer_Stack is new Generic_Stack(100, Integer);
```

人们也可以构造一个长度为500元素的Float类型的栈，如

```
package Float_Stack is new Generic_Stack(500, Float);
```

这些实例化在编译时构造了Generic_Stack的两种不同的源代码版本。

11.5.2 C++

C++也支持有参数的或是通用的抽象数据类型。为了使在11.4.2节中的C++栈类的示例在栈

的大小方面通用化，只需要改变其中的构造函数。如下所示：

```
stack(int size) {
  stkPtr = new int [size];
  maxLen = size - 1;
  top = -1;
}
```

现在，一个栈对象的声明就可以是：

```
stack stk(150);
```

stack的类定义可以包括两个构造函数，从而用户可以使用默认大小的栈，或者是说明别的大小。

我们可以通过将这个类构造成为模板类，而使栈元素的类型成为通用的。那么，这个元素类型就可以是一个模板参数。一个栈类型的模板类的定义为：

```
#include <iostream.h>
template <class Type>  // Type是模板参数
class stack {
  private:
    Type *stackPtr;
    int maxLen;
    int topPtr;
  public:
// 100个元素的构造器
    stack() {
      stackPtr = new Type [100];
      maxLen = 99;
      topPtr = -1;
    }
// 给定元素数目的构造器
    stack(int size) {
      stackPtr = new Type [size];
      maxLen = size - 1;
      topPtr = -1;
    }
    ~stack() {delete stackPtr;};  // 一个析构器
    void push(Type number) {
      if (topPtr == maxLen)
        cout << "Error in push-stack is full\n";
      else stackPtr[++ topPtr] = number;
    }
    void pop() {
      if (topPtr == -1)
        cout << "Error in pop-stack is empty\n";
      else topPtr --;
    }
    Type top() {return (stackPtr[topPtr]);}
    int empty() {return (topPtr == -1);}
}
```

如同在Ada中一样，在编译时将C++中的模板类实例化。其差别在于C++中的实例化是隐式的：当创建一个对象时，如果它需要的一个模板类的版本，而且前这个版本还不存在，即会产生这个版本的一个新实例。

11.5.3 Java 5.0

Java 5.0支持一种有参数的抽象数据类型形式，其通用参数必须是类。第9章曾简要讨论过它。

最常用的通用类型是集合（collection）类型，如LinkedList和ArrayList。它们都是在通用支持出现以前的Java类库。集合类型存储Object类对象，因此它们能存储任何对象（但不是基本类型）。集合类型总是能存储多种类型（只要它们是类）。但是，它也存在3个问题：第一，在每次从集合中删除对象时，必须进行正确的类型转换。第二，在把元素添加到集合时没有错误检查机制。这意味着，一旦创建了集合，则任何类的对象都能添加到集合中。第二，集合类型不能存储基本类型。因此，为了在ArrayList中存储int值，必须首先把值放入Integer类
493 对象中。例如，考虑下面的代码：

```
ArrayList myArray = new ArrayList();//* Create an ArrayList
myArray.add(0, new Integer(47));    //* Create an Element
Integer myInt = (Integer)myArray.get(0); //* Get first object
```

在Java 5.0中，最常使用的集合类（List、ArrayList和Queue）变成了通用类。这些类通过调用类构造器的new并将其传递给指向括号内的通用参数来完成实例化。例如，ArrayList类能实例化为下面语句的存储Integer对象：

```
ArrayList <Integer> myArray = new ArrayList <Integer>();
```

该类克服了Java 5.0之前的集合的两个问题。只有Integer对象能放入myArray集合中。而且，它不需要在集合中删除对象时进行类型转换。然而，实例化存储基本类型的通用集合仍然是不可能的。

第9章讲过Java 5.0支持通配类。例如，Collection<?>是可提供给所有集合类的通配类。它允许编写能接收任何集合类型作为其参数的方法。因为集合自己是通用的，所以Collection<?>类在某种意义上是通用的通用类。

使用通配类型的对象必须小心。例如，因为这种类型的特定对象的组成部分具有类型，所以其他类型的对象不能添加到该集合中。例如，考虑：

```
Collection<?> c = new ArrayList<String>();
```

使用add方法把一些方法等放入集合将是非法的，除非其类型为String。

用户能定义Java 5.0的通用类。这是一个渐进的过程，这种类的表现像预定义通用类的行为。

11.5.4 C# 2005

正如Java的例子，C#的第一版定义集合类为存储任何类的对象。ArrayList、Stack和Queue这些类有与Java 5.0之前版本相同的问题。

通用类添加到了C# 2005中。5个预定义通用集合为Array、List、Stack、Queue和Dictionary（Dictionary类实现散列）。也正如Java 5.0一样，这些类消除了在集合中允许混合类型和把对象从集合中删除需要进行类型转换的问题。

正如Java 5.0，用户能在C# 2005中定义通用类。一个用户定义C#通用集合的功能是它们都能定义为允许索引其元素（通过下标访问）。尽管索引通常都是整数，但是也可以使用字符串替
494 代它。

Java 5.0提供了通配类的功能，而C# 2005却没有提供。

11.6 封装结构

本章前几个章节所讨论的抽象数据类型，是一些小型的封装。[2]这一节将描述大型程序所需要的多类型的封装。

11.6.1 介绍

当一个程序的规模超过数千行时，就会出现两种实际问题。从程序人员的角度来看，如果仅仅将一个程序视为一些子程序的集合，或者是一些抽象数据类型定义的集合，那么这个程序就没有适当的程序组织层次以保持它的可管理性。大型程序的第二种实际问题，是重新编译问题。在小程序的情形下，每一次修改之后重新编译整个程序的代价并不高，然而对于大型的程序，重新编译的代价就是巨大的。这面显然就有必要找到方法来避免重新编译那些没有改变的程序部分。对于这两种问题的一种明显的解决办法是将程序组织成为一系列逻辑上相关联的代码与数据的组合，可以单独地编译其中的每一个组合，而不需要编译程序的其余部分。一个封装，就是这样的一个组合。

通常将封装放置在程序库内，并且提供给编写这些封装之外的其他程序重复使用。人们编写超过几千行的程序已有40余年的历史，因而提供封装的技术也已经发展了许多年。

11.6.2 嵌套的子程序

在允许嵌套的子程序的语言中，可以通过将嵌套的子程序的定义，放置在使用这些嵌套的子程序的、逻辑上更大范围的子程序之中。Ada、Fortran 95、Python和Ruby就是这样实行的。然而，正如在第5章曾经讨论过的，这种使用静态作用域的组织程序的方式，远非理想。因而，即使在允许嵌套的子程序的语言中，也并没有将它们作为主要的组织封装结构。 405

11.6.3 C中的封装

C并不提供对于抽象数据类型的有力支持，虽然在C中可以模拟抽象数据类型和多类型的封装。

在C中，可以将相关的函数和数据的定义，放置在一个可以独立编译的文件中。这种作用相当于程序库的文件具有对于其实体的一种实现。将这个包括了数据、类型和函数声明的文件接口，放置在另外一个单独的、称为头文件的文件之中。通过在头文件中，将类型表示定义为指向一些结构类型的指针，就可以将类型表示隐藏起来，关于这些结构类型的完整定义，将仅仅出现在实现文件之中。这样的一种方式，具有与在Ada的包中使用指针作为抽象数据类型时同样的缺点，即指针内在固有的问题，以及指针赋值与指针比较所可能带来的混淆。

以源程序形式存在的头文件以及实现文件的编译版本，将被配备给客户。当使用这种程序库时，通过使用一种预处理说明：#include，就将这个头文件包含进了客户代码之中，并且，在客户代码中对于函数和数据的引用，都将经过类型检测。预处理说明#include也证明了客户程序依赖于程序库实现文件的事实。这种方式有效地将一个封装的说明与实现分离开来。

这种类型的封装虽然有效，但也产生了一些安全问题。例如，一个用户可以不使用#include，而仅仅是将头文件中的定义裁剪并复制到客户的程序中去。这样也会行得通，因为#include无非就是将它的操作数文件中的内容，复制到出现了#include的文件之中。然

[2] 在Ada中，可以将包的封装用于单类型，也可以用于多类型。

而，这样会出现两种问题。首先，丢失了客户程序依赖于程序库（及其头文件）的证明。其次，程序库的编写人员可能会修改头文件和实现文件，客户使用的就会是新的实现文件（即使并不知道它们已经被修改了）。再加上用户复制到客户程序中的旧的头文件。例如，如果客户仍旧使用的是在旧的头文件中的变量定义：x为int类型，尽管实现代码已经与新的头文件被重新编译过了，而在新的头文件中，变量x被定义为float类型。从而，实现代码将x编译为float，而客户代码将x编译为int。连接程序将不会检测出这个错误。

所以，用户有责任来保证头文件和实现文件都是最近的版本。使用make功能就能够做到这一点。

11.6.4 C++中的封装

C++中的封装类似于C中的封装，头文件被用来提供对于封装中资源的接口。事实上，由于
496 C++中模板以及分别编译所带来的复杂的交互作用，C++中模板程序库的头文件，通常包含了源程序的完整定义，而不仅仅是一些数据的声明和子程序的协议，其中部分的原因，是由于C++中的程序使用了C语言的连接器。

因为其有类似又不具有一般的封装结构，这样所导致的一种语言设计问题是，当单个对象与对象操作相关联时，并不总是那么自然。例如，假设我们有一个用于矩阵的抽象数据类型，以及一个用于矢量的抽象数据类型，而需要进行一个矩阵与一个矢量间的乘法操作。这个乘法操作的代码，必须能够存取矢量类以及矩阵类的数据成员，但这两个类都不是这段代码的自然归属。如之不论是选择了它们中的哪一个，在存取另外一个类的成员时，都会出现问题。在C++中，则可以通过允许非成员函数成为一个类的“友元”（friend）来处理这种类型的情况。友元函数对于在其中将它声明为友元的类，能够来存取这个类的私有实体。对于矢量与矩阵间的乘法操作，C++中的一种解决办法，是将这个操作定义在这两个类——矢量类以及矩阵类的外部，但是将这个操作定义为这两个类的友元。下面的代码框架介绍了这种现象：

```
class Matrix;  //** 一个类声明
class Vector {
  friend Vector multiply(const Matrix&, const Vector&);
  ...
};
class Matrix {  //** 类定义
  friend Vector multiply(const Matrix&, const Vector&);
  ...
};
//** 使用矢量及矩阵对象的函数
Vector multiply(const Matrix& ml, const Vector& vl) {
  ...
}
```

除了函数外，还可以将整个类定义为一个的友元，那么，这个类的所有私有成员对于友元类的成员都是可见的。

11.6.5 Ada中的包

Ada中的说明包，可以在它的公有与私有段中，包括任意数目的数据和子程序的声明。因而，它们也就可以包括任意数目的抽象数据类型的接口，以及任何其他程序资源。所以，包是一种多类型的封装结构。

Ada中的包可以被分别编译，如果首先编译的是说明包，那么包中的说明与体这两个部分也是分别编译的。一个使用任意数目外部的包的完整程序，也可以被分别编译，只要是编译了所有被使用的包的说明。这些被使用的包的体，则可以在客户程序被编译之后，再进行编译。 497

考虑在11.6.4小节中所描述的矢量与矩阵类型的情形，并且考虑需要一种方法来存取这两种类型中的私有部分；在C++中是通过友元函数来处理的。在Ada中，则可以将矢量和矩阵都同时定义在一个单个的Ada包中，从而避免使用友元函数的需要。

11.6.6 C#中的汇编

C#中包括了一种比类更大的封装结构。在这种情形下，这种结构是为所有的.NET程序设计语言所使用，即汇编。一个汇编是一个文件或多个文件的一种组合。这些文件是作为单个的动态连接程序库或是一个可执行文件（.EXE），出现于应用程序之中。每一个文件定义一个可以分别开发的模块。一个动态链接程序库（DLL），是一组类或方法；在执行时如果有需要，可以将这些类或方法与执行程序单独连接。因而，虽然一个程序可以存取一个DLL中的所有资源，但只有实际使用的部分将被装载连接到程序之上。自从视窗出现以来，这些DLL就是视窗程序设计环境中的部分。作为其资源的对象代码的添加部分，一个.NET的汇编包括一个清单，这个清单列出所包括的每一个类的类型定义。汇编中其他资源的定义，还有这个汇编中所有汇编引用的表列以及一个汇编的版本号码。

在.NET的世界里，汇编是软件采用的基本单位。汇编可以是私有的，此时就只能够将它们用于某一种应用；或者是公有的，这时就意味着可以将它们用于任何应用。

如前所述，C#还具有一个存取修饰符，`internal`。一个类的`internal`成员在它所出现的汇编中对于所有的类都是可见的。

因为一个汇编是一个自我包括的可执行单位，它只能具有一个入口。

11.7 命名封装

我们认为，封装是用于逻辑上关联的软件资源——特别是用于抽象数据类型的一些语法容器。封装的目的是提供将程序组织成一些便于编译的逻辑单位的一种方式。这个步骤允许程序的部分在孤立的修改之后，被重新进行编译。还有另外的一种用来构造大型程序的封装，即命名封装。

一个大型程序可能由许多人员来编写，他们在某种程度上独立的工作，也许甚至是在不同的地点位置。这就要求程序的逻辑单位是独立的，但又能够一起工作。这也就产生了命名的问题：这些独立工作的开发人员，怎么能够为他们的变量、方法和类来创建名字，而不至于意外 498
地使用一些已经被开发同一个软件系统不同部分的程序人员使用了的名字。

最早存在着此类命名问题的是程序库。在过去的两个年代，大型的软件系统越来越依赖于支持软件的程序库。几乎所有使用当代程序设计语言所编写的软件，除了使用应用特定的程序库外，都要求使用大型而又复杂的标准程序库。多程序库的广泛使用，使得管理名字的新机制成为必然。例如，当一个程序人员给一个现存的程序库或新的程序库增加一些新名字时，他或她肯定不能够使用一个在客户的应用程序中或在其他的程序库中，已经定义了的名字。如果没有一些语言学方面的帮助，这简直是不可能的事，因为程序库的编写人员不可能知道一个客户程序使用了什么名字，或者是客户可能使用的其他的程序库定义了什么名字。

命名封装定义一些名字作用域帮助避免这种名字冲突。每一个程序库可以创建它自己的命名封装，以便防止自己的名字与其他程序库中定义的名字相冲突，或与客户代码中所定义的名

字相冲突。为了同样的目的，一个软件系统中的各个部分也可以创建一个命名封装。

基于命名封装是非连续的意义。命名封装是一种逻辑封装，可以将几种不同组合中的代码，放置于同一个名称空间，尽管这些代码的存储空间是不同的。在下面小节的讨论中，我们将简略地描述在C++、Java、Ada和Ruby语言中，关于命名封装的使用。

11.7.1 C++中的名称空间

C++包括一种称为名称空间的说明，以帮助程序来管理全局名称空间的问题。当在名称空间以外使用这些程序中的名字时，人们可以将每一个程序库放置在它自己的名称空间中，并使用名称空间中的名字将程序中的这些名字限定。例如，假设有一个抽象数据类型的头文件，是为实现栈的。如果担心其他的程序库文件可能会定义一个已经在这个栈抽象数据类型中使用了的名字，就可以将定义栈的文件，放置在它自己的名称空间之中。这可以通过将这个栈的所有声明都放置在一个名称空间的块中来完成，如

```
namespace MyStack {
  // 栈的声明
}
```

499

栈抽象数据类型的实现文件就可以使用在头文件中使用作用域决议操作符——::声明的这些名字。如：

```
MyStack::topPtr
```

这个实现文件也可以出现在一个名称空间的块说明中，这种说明与在头文件中所使用的说明一模一样，这就使得所有在头文件中所声明的名字，成为直接可见的。这种方法的确简单，然而可读性较差，因为这并没有明显地说明，在实现文件中声明某一特定名字的位置。

客户的代码可以通过三种方式对于一个程序库的头文件中名称空间的名字进行存取。一种方式，是使用名称空间的这个名字来限定程序库中的这些名字。例如，一种对于变量topPtr的引用，可以显示为：

```
MyStack::topPtr
```

这正是实现代码引用这个变量的方式。

另外的两种方式，是运用using指示的方式。可以使用这种指示来限定来自于某个名字空间的单个的名字，如使用语句：

```
using MyStack::topPtr;
```

这将使得topPtr成为可见的；但是，并没有使得MyStack名称空间的其他名字可见。

当然，还可以使用using指示来限定一个名称空间中的所有名字，如下列语句：

```
using namespace MyStack;
```

包括了这种指示语句的代码，就可以直接地存取在这个名称空间中定义的名字，如这一条语句：

```
p = topPtr;
```

必须知道，名称空间是C++中的一种复杂的特性，我们在这里所描述的只是其中的简单部分。

C#所包括的名称空间十分类似于C++中的名称空间。

11.7.2 Java中的包

Java中包括了一种命名封装的结构：包。包可以包括一种以上的类定义，并且在一个包中的类中间，彼此都是部分的友元类。在这里，“部分的”意味着在包的一个类中所定义的实体，或者是公有的或被保护的（参见第12章），或者是不具有任何存取说明符，这些实体对于包中的所有其他的类，都是可见的。一个包只能具有一种公有类的定义。

将不具有存取说明符的实体称为具有包作用域，因为它们在整个包中都是可见的。因而， 500
在Java中很少需要显式地声明友元，并且它没有包括C++中的友元函数或友元类。

使用一种包的声明，可以将一个文件中的资源定义说明在一个特定的包中，如下所示：

```
package myStack;
```

这种包的声明必须出现在文件的第一行。在每一个文件中没有包声明的资源，都被隐式地放置在同一个没有被命名的包中。

一个包的客户可以通过使用完全限定的名字，来引用在包中定义的名字。例如，如果在myStack包中定义了一个变量topPtr，myStack包的一个客户可以引用这个变量，如，myStack.topPtr，因为当包被嵌套时，这种引用方法很快就成为障碍，Java提供了这种import声明，它允许缩短包中名字的引用。例如，假设客户包括了下面的语句：

```
import myStack.*;
```

现在，就可以仅仅通过使用名字来引用变量topPtr，以及定义在包myStack中的其他名字。如果只是从这个包中存取一个名字，就可以在这条import声明中写出这个特定的名字，如：

```
import myStack.topPtr;
```

注意，Java中的import声明只是一种省略机制，使用import声明并不能够获取或隐藏的外部资源。事实上，在Java中，如果可以通过编译器或类装载器（装载器使用的是包名称，以及CLASSPATH环境中的变量）找到的话，那么任何资源都不是隐式隐藏的。

Java中的import声明证明了在import声明中出现包的名称的包的一些依赖关系。如果没有使用import声明，这些依赖关系是不明显的。

11.7.3 Ada中的包

通常包括了程序库的Ada的包，是在层次结构上定义的，这种层次与文件的层次相对应。例如，如果将包subPack定义为包Pack的子包，subPack的代码文件将会出现在存储Pack包的目录的子目录中。Java中的标准类库，也被定义在一个包的层次结构中，并且还被存储在一个相对应的目录层次之中。 501

正如我们第11.4.1小节中所讨论过的，包也定义名称空间。对于一个来自于程序单位的包，是通过with子句来获得可见性的，例如，下列语句：

```
with Ada.Text_IO;
```

就可以使包Ada.Text_IO中的资源和名称空间能够被使用。对于在Ada.Text_IO的名称空间所定义的名字的存取，则必须被限定。例如，存取来自于Ada.Text_IO中的Put过程，必须是下面的形式：

```
Ada.Text_IO.Put
```

为了不限定就能够存取Ada.Text_IO中的名字，就必须使用use子句，如

```
use Ada.Text_IO;
```

通过这条子句，就能够存取来自于Ada.Text_IO中的Put过程。Ada中的use子句与Java中的import声明紧密关联。

11.7.4 Ruby模组

Ruby类作为命名空间封装，其他支持面向对象程序设计的语言的类也如此。Ruby还有一个额外的命名封装，称为模组（module）。典型地，模组定义方法和常量的集合。因此，模组对于封装有关联的方法和常量是很方便的，这些方法名和常量名在相互隔离的命名空间里，以使其不与使用该模组的程序中的其他名称冲突。模组与类不同，因为它们不能实例化或子类化，也不能定义变量。在模组中定义的方法包括在它们名称中的模组名。例如，考虑下面的模组定义框架：

```
module MyStuff
  PI = 3.1415926535
  def MyStuff.mymeth1(p1)
  ...
  end
  def MyStuff.mymeth2(p2)
  ...
  end
end
```

303 假定Mystuff模组存储在它自己的文件中，一个想使用Mystuff方法和常量的程序必须先获得访问模组的权限。这可以通过require方法来完成，它接收文件名的字符串字面常量作为参数。然后，可以通过模组名访问模组的常量和方法。考虑下面使用上述模组Mystuff的代码，它存储在命令为myStuffMod的文件中：

```
require 'myStuffMod'
...
MyStuff.mymeth1(x)
...
```

有关嵌套包的讨论详见第12章。

小结

抽象数据类型的概念和它们在程序设计中的应用，是工程级程序设计发展中的一个里程碑。尽管这种概念相对简单，但是直到语言被设计出能支持它时，它的使用才成为方便和安全的。

抽象数据类型的两个主要特性是：将数据对象及其相关的操作包装在一起，和信息隐蔽。一种语言可以直接地支持抽象数据类型，或者是使用更为一般的封装来模拟抽象数据类型。

Ada提供了可以用来模拟抽象数据类型的封装——包，包通常具有两个部分：一个说明，给客户提供接口，以及一个体，它提供抽象数据类型的实现。数据类型的表示，可以出现在说明包中，然而通过将这些表示放置在包的私有部分中，而将它们对于客户隐藏起来。在说明包的公有部分，将抽象类型的本身定义成为私有的。私有的类型具有一些内置的操作，用于赋值以及等于和不等于的比较。

C++是通过类来提供数据抽象。类是一些类型，类的实例可以是静态的或动态的。一个成员函数（或方法）的完整定义，可以出现在类中，或者是仅仅将协议放置在类中，而将定义放置在另外一个可以单分别编译的文件之中。C++中的类具有三种子句，每一种都将一个存取控制符放置在前面：私有的、公有的和被保护的。构造函数和析构函数都可以在类定义中给出。对于堆分配的对象，必须使用delete显式地删除分配。

Java中的数据抽象与C++中的相类似，只是所有Java中的对象都是从堆中分配，并且是通过引用变量来存取的。此外，Java中的所有对象都被放料收集。Java中的存取修饰符没有被附加在子句里，而是出现在单独的声明或定义中。

C#使用类与结构来支持抽象数据类型，其中的结构是数值类型的并且不支持继承。除此之外，C#中的类与Java中的相类似。 503

Ruby支持抽象数据类型。Ruby的类与其他大部分语言的类都不相同，因为它们是动态的——在执行时能添加、删除或改变成员。

Ada、C++、Java 5.0和C# 2005都允许它们的抽象数据类型被参数化；Ada是通过它的通用包，C++是通过它的模板类而Java 5.0和C# 2005通过它的集合类。

为了支持大型程序结构，一些当代程序设计语言包括了多类型的封装结构，这种结构可以包含一组逻辑相关的类型。封包还可以提供对于实体的存取控制。封装给程序人员提供了一种组织程序的方法，这种方法也有助于重新编译。

C++、C#、Java、Ada和Ruby都提供命名封装。对于Java和Ada，这种封装是命名的包，而对于C++和C#，它们是名称空间。部分是由于包的可用性，Java不具有友元函数或友元类。在Ada中，是将包用作命名封装。

复习题

1. 定义抽象数据类型。
2. 抽象数据类型定义中的两个部分有什么优点？
3. 对于一种支持抽象数据类型的语言，语言设计上的要求是什么？
4. 什么是抽象数据类型的语言设计问题？
5. 解释在Ada的包中是怎样提供信息隐藏的。
6. 在Ada中，private与limited private的类型之间有什么区别？
7. 在Ada的说明包中有什么？在体包中又有什么？
8. 在C++中的类与Ada中的包之间，有哪些根本上的差别？
9. 一个C++中的成员函数的定义，可以出现在哪些不同的位置？
10. C++中构造函数的目的是什么？
11. Java中的所有方法都是在什么地方定义的？
12. C++中的类对象是在哪里创建的？
13. Java中的类对象是在哪里创建的？
14. Java中为什么没有析构函数？ 504
15. 什么是友元函数？什么是友元类？
16. Java没有友元函数或友元类的一个理由是什么？
17. Java中为什么没有析构函数？
18. 描述C#中的结构与它的类之间，有哪些根本上的差别？
19. 在C#中，一个结构对象是怎样创建的？
20. 请解释，将存取器用于私有类型，比将这种类型变为公有类型要优越的三个理由。
21. C++中的结构与C#中的结构之间，有什么差别？
22. 为什么Java语言不象Ada语言那样，需要一条use子句？
23. 所有Ruby构造器的名称是什么？
24. Ruby的类与C++和Java的类有什么根本的不同？
25. 怎样创建Ada中通用类的实例？
26. 怎样创建C++中模板类的实例？

27. 描述在大型程序的结构中所出现的两种问题，正是这两种问题导致了封装结构的开发。
28. 使用C来定义抽象数据类型会出现什么问题？
29. C++中的名称空间是什么？它的目的是什么？
30. 描述with子句和use子句的目的。
31. 什么是一个Java的包？它的目的是什么？
32. 描述一个.NET的组编。
33. 在Ruby模组中出现了什么元素？

练习题

1. 一些软件工程师相信，所有的输入实体，都应该由输出程序单位的名字来限定。你赞同吗？给出理由来支持你的答案。
2. 假设有人设计了一个栈抽象数据类型，其中的函数top所返回的，是一个访问途径或指针，而不是返回栈顶元素的副本。这不是一个真正的数据抽象。为什么？举例说明。
3. 分析Java的包和C++的名称空间之间的相似性及差别。
4. 将抽象数据类型设计成为一种指针，具有哪些优越性？
5. 为什么在Ada的说明包中，必须给予非指针抽象数据类型的结构？
505 6. 相对于C++，在Java中通过废料收集避免了什么样的危险？
7. 相对于在C++和Java中编写存取器方法，讨论C#中的属性的优越性。
8. 解释C中的封装方式的危险性。
9. 为什么在C++中没有消除练习题8中所讨论的问题。
10. 解释为什么命名封装对于开发大型程序是重要的。
11. 描述一个客户程序从C++的名称空间引用一个名字的三种方式。
12. C#中标准程序库的名称空间System对于C#的程序并非隐式可用的。你认为这是一个好主意吗？给出理由来支持你的答案。

程序设计练习题

1. 设计本章前面示例的Fortran抽象栈类型，将一个具有多人口的单个子程序，用于类型定义与操作的定义。
2. 就可靠性以及灵活性方面，将程序设计练习题1中Fortran的实现与在这一章中的Ada的实现相比较。
3. 在你所知道的一种语言中，设计一个矩阵抽象的抽象数据类型，包括用于加法、减法和矩阵乘法的操作。
4. 在你所知道的一种语言中，设计一个队抽象数据类型，包括用于入队、出队和判空的操作。
5. 修改在本章中出现的，用于抽象栈类型的C++类，改为使用链表来表示，并使用在本章所出现的相同的代码来进行测试。
6. 编写一个复数的抽象数据类型，包括用于这些运算的操作：加法、减法、乘法、除法、抽出复数每一个部分，以及以两个浮点常量、两个变量或两个表达式来构造一个复数。使用Ada、C++、Java、C#或Ruby语言。
7. 编写一个队的抽象数据类型，队中的元素存储10个字符的名字。这个队的元素必须动态地从堆中分配。队的操作包括：入队、出队和判空。使用Ada、C++、Java、C#或Ruby语言。
506 8. 编写一个队的抽象数据类型，队中的元素可以是任何基本类型。使用C++或Ada语言。

第12章　支持面向对象的程序设计

这一章将从介绍面向对象程序设计开始，接着对于继承与动态绑定中的主要设计问题进行讨论。然后，我们将讨论Smalltalk、C++、Java、C#、Ada 95和Ruby中对于面向对象程序设计的支持。之后我们将简略地描述JavaScript的对象模型。在本章的结尾，是面向对象程序设计语言中的方法调用与方法间动态绑定的实现。

12.1　概述

现在支持面向对象的程序设计语言已经稳固地进入了主流。从COBOL到LISP（实际上包括了在这两种语言之间的所有语言），都出现了一些支持面向对象程序设计的方言。C++和Ada 95除了支持面向对象程序设计以外，还支持面向过程，以及面向数据的程序设计。CLOS是LISP的面向对象的版本（Bobrow et al., 1988），也支持函数式程序设计。一些较新的支持面向对象程序设计的语言，并不支持其他程序设计的范型，但仍然采用基本的命令式结构，并仍然具有命令式语言的外观。这样的语言有Java和C#。Ruby有点不好分类：它是全部数据都是对象的纯面向对象语言，但是它也是能用于过程的混合语言。最后，还有一种纯面向对象的语言，这种语言很典型地非传统：它就是Smalltalk语言。Smalltalk是第一种全面支持面向对象程序设计的语言。支持面向对象程序设计的细节，不同的语言间各有不同，而这正是本章的一个重要课题。

本章与第11章十分紧密地联系在一起，实际上可以说，本章就是第11章的继续。这种关系充分反映了这样的事实：即面向对象程序设计，是抽象数据类型的抽象原理的一种应用。尤其是当将面向对象程序设计中一组类似抽象数据类型的共性抽出来，形成一种新的类型时。这组类型的成员，都继承了来自这种新类型的共同部分。这种特征就是继承(inheritance)。继承是面向对象程序设计以及支持这种设计的语言的核心。

12.2　面向对象程序设计

12.2.1　介绍

面向对象程序设计概念的根源可以追溯到SIMULA 67，但是这个概念的全面开发，是直到Smalltalk演化促成了Smalltalk 80（当然是在1980年）的产生之后才进行的。实际上，有些人认为Smalltalk是唯一的纯面向对象程序设计语言。一种面向对象语言必须提供对于三个关键的语言特性的支持。这些语言特性是：抽象数据类型、继承以及方法调用与方法间的动态绑定。我们已经在第11章详细讨论过抽象数据类型，所以本章的重点是继承和动态绑定。

507
508

12.2.2　继承

程序人员长期以来的一种压力，就是要提高生产率。随着计算机硬件成本的持续下降，这种压力变得越来越大。到了20世纪80年代的中期和后期，许多软件开发者清楚地认识到，在计算机领域里提高生产率的最好机会就是软件的重复使用。具有封装及访问控制的抽象数据类型，显然是软件复用的候选者。几乎在所有的情形下，抽象数据类型复用中的问题，都是已有类型的特性和功能不适合于新的使用。这就至少需要对老的类型稍微进行一些修改；然而这种修改

工作可能会很困难，因为这要求修改人员了解部分（如果不是全部的话）已有的代码。此外，在许多情况下，这种修改还需要更新所有的客户程序。

面向数据程序设计的第二个问题，是所有抽象数据类型的定义都是独立的，并都在同一个层次上。这种设计常常使得不可能来构造一个适合问题空间的程序。在许多情况下，所隐含的问题具有不同的种类但彼此相关的对象：有的如同兄弟（彼此相类似），也有的如同父子（具有上下代的关系）。

继承提供了一种解决抽象数据类型的复用所面临的修改问题，以及程序组织问题的办法。如果一种新的抽象数据类型能够继承一些已有类型的数据和功能，并且能够允许修改一些实体及增加一些新的实体，那就不需要改变被复用的抽象数据类型，从而极大地有助于软件的复用。程序人员可以从已有的抽象数据类型出发，设计一个修改了的后代类型，以满足新问题的要求。继承还为相关类的层次关系的定义，提供了一个框架。这种层次关系可以反映出，在问题空间中这些类的上下代关系。

面向对象语言中的抽象数据类型，采用SIMULA 67中的术语，通常称之为**类**（class）。对应于抽象数据类型的实例，即类的实例，被称为**对象**（object）。通过对另外一个类的继承来定义的类，被称为**派生类**（derived class）或**子类**（subclass）。派生新类的类，是这个新类的**父类**（parent class）或**超类**（superclass）。将定义于类的对象上的操作子程序，称为**方法**（method）。而对于方法的调用则通常称为**消息**（message），将一个对象的方法的完整集合，称为是这个对象的**消息协议**（message protocol）或者**消息接口**（message interface）。在一个面向对象程序中的计算，是通过从一个对象送往另外一些对象（在某些情况下是类）的消息来说明的。

509

在最简单的情况下，一个派生的类继承其父类的所有实体（变量及方法）。因为在父类实体上的访问控制，情况就变得十分复杂。例如，当我们在第11章看到抽象数据类型的定义时，其中的一些实体被设为公有的，而另外的一些实体则被设为私有的。这些访问控制允许程序的设计人员对类的客户隐匿部分的抽象数据类型。派生类是另外一种客户，可能会允许，也可能会阻止来自这些派生类的访问，考虑到这种可能性，从而在一些面向对象语言中包括了第三种类型的访问控制，通常称为**受保护的**（protected）：使用它来允许来自派生类的访问，但阻止来自其他类的访问。关于是否一个子类包括了其父类的所有实体，我们将在12.3.2小节中详细地讨论这个问题。

除了从父类继承实体外，派生类还可以增加新的实体，以及修改所继承的这些方法。修改后的方法与原有的方法具有相同的名字，通常也具有相同的协议，新的方法被称为覆盖（override）了继承的版本，因而称这种方法为**覆盖方法**（overridden method）。覆盖方法最通常的目的，是为派生类的对象提供一种操作，但这种操作对父类的对象却是不恰当的。

一个类可以具有两种类型的方法及两种类型的变量。最常用的方法和变量，被分别称为**实例方法**（instance method）和**实例变量**（instance variable）。类的每一个对象，都拥有一套自己的实例变量来存储对象的状态。同一个类的两个对象之间的唯一不同，只是它们的实例变量的状态不同。实例方法只能被施操作于类的对象之上。**类变量**（Class variable）属于类，但并不属于类的对象，因而一个类只有唯一的一个拷贝。**类方法**（Class method）可以完成在类上的操作，也可以完成这个类的对象上的操作。

如果一个新的类是单个父类的子类，那么这个派生的过程就被称**单继承**（single inheritance）。如果一个类具有多个父类，那么这个派生的过程就被称**多继承**（multiple inheritance）。当许多类都是通过单继承相关联时，就能够将这些类的彼此关系显示在一棵派生树中；多继承中类的相互关系则能够显示在派生图之中。

继承是作为增加复用可能性的一种手段，然而它的缺点则是产生了在继承层次结构中的类之间的相互依赖性。这种结果正好与抽象数据类型的优点相反，抽象数据类型的优点是数据类型间的相互独立性。当然，并不是所有的抽象数据类型都必须是完全独立的。但一般而言，抽象数据类型的独立性是其最具有效率的一个正面特征。然而，要增加抽象数据类型的复用性，而不产生它们之间的相互依赖性，则是非常困难的。在很多情况下，类的依赖关系自然地反映出了问题空间中的依赖关系。 510

12.2.3 动态绑定

面向对象程序设计语言的第三个特征，是消息对于方法定义的动态绑定所提供的一种多态性。考虑下面的情况：存在着一个基类A，A中所定义的一个方法对于A的对象施行一个操作，将B定义为A的一个子类，B的对象也需要一个类似于A中的这个方法。由于B的对象与A的对象略有不同，因而B所需的方法与A的方法也就不同。因而这个子类将覆盖继承的方法。如果一个A和B的客户，具有对于A的对象的一个引用或指针，这个引用或指针也将指向B的对象，形成所谓的**多态引用**（polymorphic reference）或多态指针。如果定义于A和B这两个类中的一个方法是通过多态引用来调用的，运行时系统在执行时就必须决定，究竟是调用A中的方法还是调用B中的方法。这可以通过先确定指针或引用当前所引用的究竟是哪一个类的对象来决定。在12.5.3小节中，我们将给出一个C++中的动态绑定的例子。

动态绑定的一个目的，就是为了使得软件系统的开发和维护更加容易。例如，假设一个类定义表示鸟的对象，它的子类定义为特定的鸟类。更进一步，假设每个子类重新定义一个从显示它的特定鸟类的基类中继承的方法，这些方法能通过引用或指向鸟的基类的指针来调用。因此，如果客户端代码创建一个引用具体鸟类子类对象的基类指针数组，并且需要显示数组中引用的每一种鸟，那么每一种鸟的显示方法将通过数组中的基类指针调用相同的调用（在一个循环体内）。维护这样的好处是添加新的子类（对系统来说是一种新的鸟）将不需要改变调用显示方法的代码。

在许多情况下，继承的层次结构的设计将产生一个或多个类，这些类在这种层次结构中占据了较高的位置，以至于这些类的实例化都不再有意义。例如，假设程序定义类building和一组特定类型的子类，如类French_Gothic。如果在类building中具有被实现的方法draw，或许已经没有意义了。但是由于building的所有后代类都应该具有这种实现的方法，这种方法的协议（但不是方法体）就将被包含在building类中。这种方法常常被称为**抽象方法**（abstract method），在C++中也称为**纯虚拟方法**（pure virtual method）。此外，任何至少包括了一个抽象方法的类都称为**抽象类**（abstract class），在C++中是称为**抽象基类**（abstract base class）。这样的类不能够被实例化，因为不是它的所有方法都具有方法体。抽象类的任何被实例化的子类，都必须是供所有继承的抽象方法的实现。 511

12.3 面向对象语言的设计问题

当设计用以支持继承与动态绑定的程序设计语言的特性时，有几个问题必须考虑，我们将被认为是其中最重要的一些问题，在这一节里进行讨论。

12.3.1 纯对象模型

完全采用计算的对象模式的语言的设计人员设计了一种对象的系统，这种系统吸收了所有的其他的类型概念。在此系统中的每一件事物，从最小的整数到完整的软件系统都是对象。这

行这种选择的优点是语言的优雅性及纯粹一致性，以及它的方便使用。而它的主要缺点是，甚至连简单操作都必须经过消息传递的过程来完成，这使得这种方法常常会比在命令式模型中的类似操作要慢得多。而命令式模型是用机器指令来实现简单操作的。例如在Smalltalk中将数字7加到一个名为x的变量之上，是通过将对象7作为参数传送给对象x的+方法来实现的。在最纯的面向对象的计算模型中，所有的类型都是类。预定义的类与用户定义的类之间不存在差别。事实上，所有的类都被同样地对待，而且所有的计算都是经过消息传递来完成的。

纯对象模型的一种替代方法（这常见于增加了支持对面向对象程序设计的命令式语言中），是保持完全的命令式类型化模型，而仅仅是再加上对象模型。这样的结果即是产生了一种大型的语言，这种语言的类型结构，将使得除了个别专业用户以外的大多数人十分困惑。

纯对象模型的另外一种替代方法，是使用一种命令式风格的类型结构来作为原始标量类型，但是实现所有结构化类型作为对象。这种选择能够提供在原始值上的快速操作，而这种操作的速度能够与人们在命令式模型中所期望的速度相媲美。但是，这种替代方法导致了语言的复杂化。在这里不变的是，非对象值必须与对象相混合，从而为非对象类型产生了对于所谓的包装类（wrapper class）的需要，这样就可以将一些通常需要的操作发送给包含非对象值的对象。在12.6.1小节中，我们将会在Java中讨论一个这样的例子。

12.3.2 子类是子类型吗

这里的问题相对简单："is a"（是一个）关系是否在派生类与其父类之间成立？从纯语义的观点来看，如果一个派生的类"是一个"父类，那么这个派生类的对象必须暴露所有由父类对象暴露的成员，在一种不那么抽象的层次上，is a关系将保证一个派生类类型的变量，可以出现在其父类类型的变量为合法的任何位置上，而不会导致任何类型错误。而且，派生类对象必须在行为上与父类对象一样。

512

Ada中的子类型就是这种简单继承形式的例子。例如，

```
subtype Small_Int is Integer range -100..100;
```

Small_Int类型的变量具有Integer变量所拥有的所有操作，但是只能够保存Integer的所有可能值的一个子集。此外，可以将每一个Small_Int变量用于任何可以使用Integer变量的位置。这就意味着，Small_Int变量在某种程度上就是Integer变量。

有很多种方式使子类与其基类或父类不同。例如，子类能有更多的方法，能有更少的方法，一些参数的类型在一种或多种方法中可能是不同的，一些方法的返回类型可以不同，或者一种体或多种方法体也可以是不同的。大多数程序设计语言严格限制子类这分基类的方法。在大多数情形下，语言规则限制其子类成为其父类的子类型。

正如前面讲述的，如果一个派生类与其父类具有 is-a 关系，就称这个派生类为子类型（subtype）。能够确保一个子类是子类型的特征是：前这里的兼容指的是，这种覆盖方法可以替代被覆盖方法，而不会引起类型错误。那意味着每种覆盖方法必须具有与被覆盖方法相同数目的参数，并且参数类型和返回类型必须与父类兼容。如果具有同样的参数数目，以及同样的参数类型与返回类型，当然就能够保证这种兼容性。稍微不严格的限制也是可能的，这取决于语言的类型兼容性规则。

我们的子类型的定义明白地说明了，父类中的所有实体都必须被继承到子类之中。因此，为子类型的派生过程必须要求继承父类的公有实体作为子类的公有实体。

子类型关系和继承关系看起来几乎完全是相同的。然而，这种推测却远非正确。在12.5.2小

节中，我们将使用一个C++的例子来解释这种不正确的假设。

12.3.3 类型检测与多态

在12.2节中，曾经将面向对象领域中的多态定义为使用一个多态的指针或一种多态的引用来进行访问的方法，这个方法的名字在类的层次结构中被覆盖，而正是这种层次结构定义了指针或是引用所指向的对象。多态的变量具有父类的类型，并且父类至少定义了被派生类覆盖的一个方法的协议。多态的变量可以引用父类及其派生类的对象，因而并不总是能够静态地确定多态变量所指向的对象的类。经过多态变量所传送的消息与方法的绑定必须为动态的。这里的问题是，什么时候对于这种绑定来施行类型检测。 513

这个问题十分重要，因为它与程序设计语言的基本性质相关联。因为动态类型检查占用了执行时间和延缓了类型错误检测，所以静态进行类型检测是更好的选择。要求静态类型检查给多态消息与方法间的关系强行加上了严格的限制。

一种强类型的语言在消息与方法之间必须进行两种类型检测：消息的参数类型必须对照方法的形参来进行检测。方法返回的类型必须对照消息所期待的类型进行检测。如果这些类型必须完全匹配，那么一种覆盖方法必须与被覆盖方法具有同样数目的参数以及同样的参数类型，并且返回的也是同样的类型。这条规则的一种较宽的规定可以允许在实参与形参之间以及返回的类型与消息所期待的类型之间，存在赋值兼容性。

静态类型检测的一种明显的替代方案，是推迟类型检测直到使用多态变量来调用一个方法之时。

12.3.4 单继承与多继承

另外一个简单的问题是：语言除了允许单继承外也允许多继承吗？然而，这个问题的答案也许不是那么简单，多继承的目的是要允许一个新的类继承两个或者多个类。

多继承有时非常有用，但为什么语言设计者不包括进多继承呢？有两个原因：即复杂性与效率。所增加的复杂性可由几个问题来说明。首先请注意，如果一个类具有两个互不相关的父类，而两者所定义的名字都没有在对方之中，这当然没有问题。然而，假设一个名称为C的子类继承类A与类B，而且如果类A与类B都包括了一个名称为display的可继承方法。如果C需要引用display的这两个版本，怎么能够做到？当两个父类同时定义具有完全相同名称的方法而且这两个方法中的一个或两个必须在其子类中被覆盖时，这种歧义性问题就会变得更加复杂。

另外的一个问题是，如果A和B都源于共同的父类Z，而且A和B都是C的父类。我们就称这种情形为菱形继承（diamond或shard inheritance）。在这种情况下，A和B都具有Z的可继承变量。假设Z有一个称为sum的可继承变量，现在的问题是，C究竟是应该继承sum的这两个版本，还是其中的一个。如果是其中之一，又应该是哪一个？在一些程序设计的情况下，仅仅需要继承其中的一个，而在其他的一些情况下，又需要二者都被继承。在12.10节中包括了实现这几种情形的简略讨论。图12-1显示了菱形继承。 514



图12.1 菱形继承的一个例子

关于效率方面的问题，可能更多的是感觉上的而非实际上的。例如，在C++中，至少是在一些机器的体系结构上，支持多继承就只需要对每一个动态绑定方法的调用附加上一种额外的数组访问和操作（Stroustrup，1994，p.270）。即使程序没有使用多继承的话，也必须这样做；但这仅仅是很小的额外代价。

多继承的使用容易导致程序组织的复杂化。许多试图使用多继承的人都发现，设计多继承中的多父类是十分困难的。维护使用多继承的系统可能是一个更为严重的问题，因为多继承导致了复杂的类与类之间的依赖关系。人们仍旧不十分清楚的是，多继承所具有的优点，与在设计及维护中增加的额外工作量相比是否真正值得。

12.3.5 对象的分配与解除分配

对象的分配与解除分配涉及两个设计问题。第一个问题，是在那里进行对象的分配。如果这些对象的行为类似于抽象数据类型，那么也许能够在任何位置进行分配。这意味着它们可以从运行时的栈中来分配，也可以使用一个操作符或函数，如new，将它们显式地创建在堆中。如果它们全部都是从堆上分配，则将具有通过指针或引用变量来创建或访问的统一方法的优越性。这种设计将简化对于对象的赋值操作，使得在任何情况下只须要改变指针或引用的值。也可以允许隐式间接地进行对象的引用，从而简化访问的语法。

如果对象是栈动态的，就会有一个与子类型有关的问题。如果B类是A类的子类，并且B是A的子类型，那么可以将B类型的一个对象赋给A类型的一个变量。例如，如果b1是B类型的变量，
515 并且a1是A类型的变量，那么，

```
a1 = b1;
```

是合法的语句。如果a1和b1是对于堆动态对象的引用，则什么问题也没有——这仅仅是一个简单的指针赋值。然而如果a1和b1是栈动态的，那么a1和b1就是值变量，这样就必须将对象的值复制到目标对象的空间中去。如果B给它从A所继承的部分增加了一个数据域，那么a1在栈中将没有足够的空间来存放所有的b1，当赋值时，b1中的多余数据将被截掉。这对于书写和使用这段代码的人来说，这可能是难以理解的。

第二个问题涉及对象从堆中分配的情形。这里的问题是，解除分配到底应该是隐式的还是显式的，或是这两者兼而有之。如果是隐式地解除分配，就需要某种隐式存储空间回收的方法。如果分配可以是显式的，就引起了是否会产生悬挂指针或悬挂引用的问题。

12.3.6 动态绑定与静态绑定

如我们曾经讨论过的，在继承的层次结构中的消息对方法的动态绑定，是面向对象程序设计的根本部分。这里的问题是，是否所有的消息与方法间的绑定都是动态的。一种替代的方案，是允许用户来说明哪种绑定应该为动态的，哪种又应该为静态的。静态绑定的好处，是比动态绑定的速度更快。所以如果并不需要动态绑定的话，我们就没有必要一定要付出高的代价。

12.3.7 嵌套类

进行嵌套类定义的一个主要原因是信息隐藏。如果仅有一个类需要一个新类，那么没有理由定义一个能被其他类看见的新类。在这种情况下，这个新类可以嵌套在使用它的类中。在某些情况下，新类可以嵌套在子程序中，而不是直接嵌套在另一个类中。

在其中嵌套了新类的类可作嵌套类。与类嵌套有关的最明显的设计问题与可见性有关。一个主要问题是：嵌套类中的哪些内容对嵌嵌套类是可见的？另一个主要问题正好与此相反：被嵌套类中的哪些内容对嵌套类是可见的？

12.4 Smalltalk对面向对象程序设计的支持

很多人认为Smalltalk是权威的面向对象程序设计的语言。它是第一种包括了对于面向对象

程序设计全面支持的语言。因而很自然地，我们将从Smalltalk开始，来描述支持面向对象程序设计的语言。 516

12.4.1 一般特征

Smalltalk的程序完全由对象组成。因而无物不是对象。事实上每一个事物，从简单的，如整数常量2，到复杂的文件处理系统，都是对象。对象被等同地对待。它们都具有局部存储能力、内在处理能力、与其他对象通信的能力以及从祖先那里继承方法与实例变量的可能性。

消息可以使用引用对象的变量来作为参数。对于消息的回复具有对象的形式，并且被用来返回被请求的信息或者是确定被请求的服务已完成。

Smalltalk中的所有对象都从堆中分配，并且通过引用变量来引用——这是一种隐式间接的引用。不存在显式解除分配的语句或操作，所有的解除分配都是隐式的，并且使用废料收集程序来进行存储空间的回收。

不同于C++和Ada 95之类的混合式语言，Smalltalk只是为单一的软件开发技术——即面向对象的方法而设计的。此外，Smalltalk没有采用任何命令式语言的外表。这种语言的目的之纯粹，反映在语言的简单优雅以及语言设计的一致性上。

12.4.2 类型检测与多态

在Smalltalk中，是通过这样的操作来进行消息与方法的动态绑定的：送往一个对象的消息，将引起这个对象所属的类来搜索对应的方法。如果没有找到的话，搜索将继续在这个类的超类之中进行，并依此类推，一直进行到不具有超类的系统类Object为止。Object是类的派生树的根，而在这个派生树中的每一个类都是一个节点。如果在这条链中没有发现任何方法，则会产生一个错误。重要的是必须记住，这些方法的搜索是动态的，并且发生于发送消息之时。在任何情况下Smalltalk中的消息都不会与方法静态地绑定。

Smalltalk中唯一的类型检测也是动态的，并且在检测中仅有的类型错误，发生在发送消息给一个没有被匹配的方法的对象之时，而无论这个方法是局部的还是继承的。这个概念与其他大多数语言中的类型检测不同。Smalltalk中的类型检测具有确保消息与某个方法相匹配的简单目的。

Smalltalk中的变量是无类型的，任何名字都能够被绑定到任何对象之上。这样做的直接结果是Smalltalk支持动态的多态。只要变量的类型是一致的，究竟是什么类型都是无所谓的，从而Smalltalk的代码都是通用的，变量上的操作（方法或操作符）的含义由变量当前所绑定的类来决定。

这种讨论的要点是，只要在一个表达式中所引用的对象具有这个表达式的消息的方法，对象的类型就是无所谓的。这意味着这些代码都与特定的类型无关。 517

12.4.3 继承

一个Smalltalk的子类将从它的超类继承所有的实例变量、实例方法和类方法。子类也可以拥有自己的实例变量，但必须具有与祖先类中的变量不同的变量名。最后一点，子类可以定义新的方法，也可以重新定义已经存在于祖先类中的方法。如果子类具有一个方法，这个方法的名称及协议与一个祖先类的相同，那么这个子类方法就将祖先类的方法隐藏起来。对于这个被隐藏的方法的访问，是通过具有伪变量super前缀的消息来提供。这个前缀使得方法的搜索是从超类开始，而不是从局部开始。

因为在父类中的实体不能够对于子类隐藏起来，因而所有子类都是子类型。每个子类对象和其父类对象之间保持着is-a关系。

Smalltalk支持单继承，但不允许多继承。

12.4.4 Smalltalk的评估

虽然Smalltalk的系统很大，但它的核心是一种小型语言。这种语言的语法也非常简单与规则。它本身就是一个极好的例子，说明如果一种语言是围绕一个简单而有效的概念来建造的，一种小型语言就能够提供强大的功能。在Smalltalk的情形中，这种语言的概念是，所有的程序设计都能通过使用继承、对象和消息传递所建造类的层次结构来完成。

比较传统编译的命令式语言的程序，等同Smalltalk的程序要慢得多。虽然从理论的角度来省视，能够在消息传递的模型中提供数组的索引以及循环是十分有趣的，然而效率是评估程序设计语言的重要因素。因此在大部分讨论中，效率显然是Smalltalk在实际应用中的一个问题。

Smalltalk的动态绑定允许直到运行时才检查类型错误。可以编写并且编译一个程序，这个程序中的消息被发送给一个并不存在的方法。较之一种静态类型的语言，这将引起比后面开发中的多得多的错误修正。

总而言之，Smalltalk的设计自始至终都偏重于语言的优雅性，以及严格遵从面向对象的程序设计，而常常并不从实际的角度来考虑问题，尤其是在执行效率方面。最为明显的是在非对象不使用以及无类型变量等方面。

Smalltalk的用户界面对于计算有着重要的影响：视窗的整合使用，鼠标指向机制，以及弹出式或下拉式菜单，都是首先出现于Smalltalk中，主宰了当代的软件系统。

518 Smalltalk语言最重大的影响是先进的面向对象的程序设计，这已经成为了最广泛使用的程序设计以及编码方法学。

12.5 C++对面向对象程序设计的支持

我们在第2章里曾经描述了C++是怎样从C和SIMULA 67中演变而来的，C++的设计目的，是支持面向对象程序设计。我们也曾经在第11章中讨论过，C++中的类对于抽象数据类型的支持。在这一节里我们将要研究的是C++对于面向对象程序设计的一些其他基本要素的支持。C++中的类、继承和动态绑定的整个细节十分庞大与复杂。本书仅仅讨论这些题目中最为重要的一些问题，特别是那些与曾经在12.3节中讨论过的设计问题直接相关的问题。

C++仍然是最广泛使用的、最流行的面向对象程序设计语言。因而自然成为了其他语言相比较的对象。由于以上两个原因，我们对于C++的介绍比在这一小节中包括的其他语言更为详细。

12.5.1 一般特征

C++的一个主要设计考虑是与C语言向后兼容，因而C++保留了C语言的类型系统，并且还加进了类。因此，C++既有传统命令式语言的类型，又有面向对象语言的类结构。它既支持方法，又支持不从属于任何类的函数。这使得C++成为了一种混合式语言，它既支持过程式程序设计，也支持面向对象程序设计。

C++的对象可以是静态的、栈动态的和堆动态的，因为C++没有隐式存储空间的回收，需要使用delete操作符对于堆动态的对象显式地解除分配。

C++中的所有的类至少包括了一个构造器方法，用于设定新对象的数据成员的初值。当产生一个对象时，构造器方法就被隐式地调用。如果任何一个数据成员是指向堆分配的数据的指针，则由构造器来进行这种分配。如果在类的定义中没有包含构造器，编译器就将提供一个简单的构造器。这个默认的构造器将调用父类的构造器，如果存在着父类的话（参见12.5.2节）。

许多类的定义包括一个析构器方法。当类的一个对象停止存在时析构器方法被隐式地调用。析构器被用来删除数据成员引用的堆分配的内存。通常为了调试的目的，也可以使用析构器方法来记录这个对象停止之前部分的或全部的状态。

12.5.2 继承

一个C++的类可以派生于一个已有的类，这个已有的类就成为这个C++类的父类或基类。与Smalltalk不同的是，C++中的一个类可以是独立的，它不具有超类。 519

定义于一个类的定义中的数据，被称为这个类的数据成员；而定义于一个类的定义中的函数，则被称为这个类的成员函数（在其他语言中的成员函数通常称为方法）。基类的部分或全部的数据成员以及成员函数，都可能是被派生类所继承的，它们也能够增加新的数据成员和成员函数，并且能够修改继承而来的成员。

在第11章里曾经讨论过，类的成员可以是私有的、受保护的或者是公有的。私有的成员只能够接受成员函数以及这个类的友元的访问。函数和类都可以被声明为一个类的友元，并因此被许可对于这个类的私有成员进行访问。任何函数都可以访问公有的成员。除了对派生类以外，受保护的成员像私有成员一样，对于这类成员的访问将在下面给予描述。派生类可以修改它们所继承的成员的可访问性。派生类的语法形式为：

```
class 派生类名称: 访问模式 基类名称
{数据成员和成员函数的声明};
```

"访问模式"可以是public或private（不要混淆带有公有成员和私有成员的公有派生和私有派生）。[⊖]在公有的派生类中，基类的公有的成员与受保护的成员，也分别是公有的和受保护的成员。在私有的派生类中，基类的公有的成员与受保护的成员都是私有的。因此在类的层次结构中，一个私有的派生类切断了所有后续类对于任何祖先类的成员的访问，受保护的成员或许可以也或许不可以接受后续子类的访问。一个基类的私有成员是被派生类所继承的，但它们对于这个派生类的成员是不可见的，因此在那里就没有用处。考虑下面的例子：

```
class base_class {
  private:
    int a;
    float x;
  protected:
    int b;
    float y;
  public:
    int c;
    float z;
};

class subclass_1 : public base_class { ... };
class subclass_2 : private base_class { ... }; 522
```

⊖ 也可以是protected，但这里不讨论这个选项。

在subclass_1中，b和y为受保护的，而c和z为公有的。在subclass_2中，b，y，c和z是私有的。Subclass_2的派生类的成员不能够访问base_class中的任何成员。Base_class中的数据成员a和x在subclass_1或subclass_2中都是不能够访问的。

注意，私有派生子类不能为子类型。例如，如果基类有一个公有数据成员，那么在私有派生下，数据成员在子类中将是私有的。然而，如果用子类对象替代基类对象，对于类对象数据成员的访问将是不合法的。这将破坏is-a关系。

父类的成员对于私有派生类的实例都不是隐式可见的。如果任何成员要成为可见的，就必须从派生类里重新输出，虽然这个派生是私有的，这种重新输出事实上避免了成员的藏匿。例如，考虑下面的类定义：

```
class subclass_3 : private base_class {
  base_class :: c;
  ...
}
```

现在，subclass_3的实例就能够访问c。对于c而言，这个派生好像已经成为了公有的一样。在这个类定义中的双冒号（::）是一个作用域归结操作符。它说明定义后面所跟随的实体的类。

下面段落的例子说明私有派生的目的和使用。

考虑下面C++继承的例子：在此例中定义了一个普通的链表类，然后又使用这个类来定义两个有用的子类：

```
class single_linked_list {
  private:
    class node {
      public:
        node *link;
        int contents;
    };
    node *head;
  public:
    single_linked_list() {head = 0};
    void insert_at_head(int);
    void insert_at_tail(int);
    int remove_at_head();
    int empty();
};
```

523

被嵌套的类node定义了一个链表的单位来包括一个整数，以及一个指向一个单位的指针。类node是位于private子句中，这使得node对于其他的类为不可见的。然而，node的成员们又是公有的，这使得这些成员对于嵌套node的single_linked_list类是可见的。如果这些成员是私有的，node则会需要将嵌套它的single_linked_list类声明成为一个friend，以使得这些成员在single_linked_list中为可见的。请注意，被嵌套的类对于嵌套它们的类的成员并没有特殊的访问权力。对被嵌套的类在嵌套它们的类中只有static的成员是可见的。[1]

嵌套类single_linked_list只有一个数据成员——即作为这个表的首部的一个指针。single_linked_list还包含了一个构造函数，这个函数只是将head设置为空指针值。四个

[1] 也可以将一个类定义在另一个类的方法之中。这种类的作用域规则与普通定义在类体中的类的作用域规则相同，甚至也与在这个方法中所声明的局部变量的作用域规则相同。

成员函数允许将节点插入到链表对象的任意一端，并允许将节点从链表的一端删除，以及允许测试链表为空。

下面的定义提供了栈及队的类，它们都基于single_linked_list类：

```
class stack : public single_linked_list {
  public:
    stack() {}
    void push(int value) {
      single_linked_list :: insert_at_head(value);
    }
    int pop() {
      return single_linked_list :: remove_at_head();
    }
};
class queue : public single_linked_list {
  public:
    queue() {}
    void enqueue(int value) {
      single_linked_list :: insert_at_tail(value);
    }
    int dequeue() {
      single_linked_list :: remove_at_head();
    }
};
```

注意，stack和queue子类的对象都可以访问定义于基类single_linked_list（因为它是一个公有的派生）中的empty函数。这两个子类都定义了不做任何事情的构造函数。当创建了类的一个对象时，将隐式地调用这个子类中适用的构造器。然后再调用基类中的任何适用的构造器。因而在我们的例子中，当创建一个stack类型的对象时，将调用不从事任何事情的stack的构造器。然后再调用single_linked_list中的构造器，用以进行必要的初始化工作。 524

类stack与类queue一样都存在着相同的严重问题：这两者的对象都能访问父类single_linked_list的所有公有的成员，因而，一个stack对象可以访问insert_at_tail，并因此来破坏它的栈完整性。同样地，一个queue对象也可以访问insert_at_head。之所以允许这些不必要的访问，是因为stack和queue都是 single_linked_list的子类型。公有派生使用在子类继承基类的全部接口。可以替换的方式是在子类只继承基类的实现时允许派生。可以使用private的派生而不是用public的派生来编写这两个派生类，从而使得它们不再是其父类的子类型，[1]这样，这两者也都会需要重新输出empty，因为empty将会对于这两个派生类的实例隐藏起来。这种情况说明需要私有派生。我们将分别命名为stack_2与queue_2的栈与队类型的新的定义显示如下：

```
class stack_2 : private single_linked_list {
  public:
    stack_2() {}
    void push(int value) {
      single_linked_list :: insert_at_head(value);
    }
    int pop() {
```

[1] 因为父类的公有成员在客户的申明，所以它们将不是子类型（不在子类的客户域中，它们的成员是私有的）。

```
        return single_linked_list :: remove_at_head();
      }
      single_linked_list:: empty;
};
class queue_2 : private single_linked_list {
  public:
    queue_2() {}
    void enqueue(int value) {
      single_linked_list :: insert_at_tail(value);
    }
    int dequeue() {
      single_linked_list :: remove_at_head();
    }
    single_linked_list:: empty;
};
```

525

这两种栈和队的版本说明了子类型与非子类型的派生类型之间的差别。由于可以将栈和队实现为链表，链表就是这两者的一般化形式。因此很自然地，我们可以通过继承一个链表类来定义栈和队的类。然而，栈和队的类都不是链表类的子类型。相反过来，链表类却是栈和队类的子类型。

为什么需要友元，其中的一个理由是有时必须编写这样的子程序，它能够访问两个不同的类的成员。例如，假设一个程序使用一个类作为向量，而使用另外的一个类作为矩阵，并且需要一个子程序来将这两个类的对象相乘。在C++中，是将这个乘法函数定义为这两个类的友元，就可以进行所需运算。

C++通过允许命名多个类作为一个新类的父类，从而提供了多继承的选择。例如，

```
class A { ... };
class B { ... };
class C : public A, public B { ... };
```

类C继承类A与类B的所有成员。如果A和B碰巧都包括了具有相同名字的成员，通过使用作用域归结操作符，这些成员就能够被类C的对象毫无歧义地引用。在第12.10节中，我们将讨论在C++中实现多继承的一些问题。

C++中的覆盖方法必须具有与被覆盖方法完全一致的参数形式。如果在这种参数形式中有存在着任何不同，那么在子类中的这个方法就会被认为是一个新的方法，与在祖先类中的具有相同名字的方法无关。覆盖方法的返回类型必须与被覆盖方法的一致，或者必须是被覆盖方法返回类型的公有派生类型。

12.5.3 动态绑定

到目前为止，我们所定义的所有成员函数都是静态绑定的，也就是说，将每一个成员函数的调用都静态地绑定于一个函数定义。可以通过一个值变量来操纵一个C++中的对象，而不是通过一个指针或通过一个引用（这样的对象就是栈动态的）。在这种情况下，对象的类型是已知的静态的，所以就不需要动态的绑定。另外的一个方面，我们能够使用一个指针或是一个具有基类类型的引用变量，来指向任何由基类派生的类的对象，使得它成为一个多态变量。私有派生子类不是子类型，指向基类的指针不能用来引用在不是子类型的子类中的方法。

C++不允许值变量（非指针或引用）为多态的，当使用这种多态变量来调用一个定义于派生类中的函数时，必须将这个调用动态地绑定于正确的函数定义之上。对于需要动态绑定的成员函数，必须将保留字virtual放置于这些函数的首部之前，来将这些成员函数声明为虚拟函

数；而这种声明只能够出现于类体中。

考虑有一个名为shape的基类以及一组用于各种形状（如圆形，矩形等）的派生类的情形。如果需要显示这些形状，则对于每一个子类或者是对于每一种形状，显示成员函数draw都必须是唯一的。draw的这些版本都必须被定义为虚拟函数。当使用一个指向这些派生类的基类的指针来调用draw时，必须将这个调用动态地绑定于正确派生类的成员函数上。下面给出刚才描述过的例子中的定义： 526

```
public class shape {
  public:
    virtual void draw() = 0;
  ...
}
public class circle : public shape {
  public:
    void draw() { ... }
  ...
}
public class rectangle : public shape {
  public:
    void draw() { ... }
  ...
}
public class square : public rectangle {
  public:
    void draw() { ... }
  ...
}
```

在给出了这些定义之后，就有下面的静态绑定与动态绑定两类调用的例子：

```
square* sq = new square;
rectangle* rect = new rectangle;
shape* ptr_shape;
ptr_shape = sq;          // 设ptr_shape指向square对象
ptr_shape->draw();       // 动态绑定到square类中的draw方法
rect->draw();            // 静态绑定到rectangle类中的draw方法
```

527

注意，将上面基类shape的定义中的draw函数设置为零。使用这一特定语法来指示其成员函数是一个纯虚拟函数（pure virtual function），这就意味着这种函数没有函数体，并且不能够被调用。如果它们调用函数，那么这种函数必须被重新定义于派生类之中。纯虚拟函数的目的是提供函数的接口，但并不给出函数的任何实现。当基类中一般成员函数将不使用时，通常定义纯虚拟函数，第12.2.3节将讨论这种情形。

任何包括了纯虚拟函数的类，都是一种**抽象类**（abstract class）。实例在抽象类是不合法的。在严格意义上，抽象类只用于表示类型特征。C++提供抽象类来模拟这些真正的抽象类型。如果一个抽象类的子类没有被重新定义为其父类的一个纯虚拟函数，这个函数仍旧是子类中的纯虚拟函数，而且子类也是抽象类。

抽象类与继承，成为支持软件开发的强有力的技术。这些技术允许按层次来定义类型，以便相关的类型可以是定义其共同抽象特征的抽象类型的子类。

动态绑定，允许在任何draw的版本被编写之前就编写调用draw的代码，甚至是任何draw的版本被编写之前，新的派生类可以在几年之后再附加进来，而不需要对使用这种动态绑定成

员的代码做出任何改动。这正是面向对象语言的一种非常有用的特性。

栈动态对象的引用赋值与堆动态对象的指针赋值不同。例如，考虑下面的代码，它使用了与上例中相同的类层次：

```
square sq;          // 在栈上分配square对象
rectangle rect;     //在栈上分配rectangle对象
rect = sq;          //从square对象复制数据成员值
rect.draw();        //从rectangle对象调用draw方法
```

528

在赋值语句rect=sq中，虽然sq引用对象的成员数据将被赋值给rect引用对象的数据成员，但是rect仍将引用rectangle对象。因此，通过rect引用对象调用draw将是rectangle类。如果rect和sq是指向堆动态对象的指针，那么同样的赋值将是指针赋值，它使rect指向square对象，通过rect调用draw将动态绑定到square对象的draw方法。

12.5.4 评估

人们会自然地将C++的面向对象的特性与Smalltalk中的同样的特性进行比较。就访问控制而言，C++的继承比Smalltalk中的更为复杂。通过使用类定义中的访问控制，以及派生的访问控制，加上友元函数和友元类，C++的程序人员具有对于类成员进行访问的高度细微的控制。此外，C++还提供了多继承，而Smalltalk只允许单继承；尽管人们对于多继承的真实价值还存在着一些争论。

在C++中，程序人员能够指定是使用静态绑定还是使用动态绑定。因为静态绑定比较快速，在不需要动态绑定的情况下静态绑定具有优越性。此外，甚至C++中的动态绑定较Smalltalk中的动态绑定，也还是更为快速的。在C++中，将一个虚拟成员函数调用绑定于一个函数的定义有着固定的代价，而不论出现在继承层次结构中的这个函数定义相距有多远。与静态绑定的调用相比，虚拟函数的调用只需要五次额外的存储空间的引用（Stroustrup，1988）。然而，在Smalltalk中，消息总是动态地绑定于方法，并且正确的方法在继承阶层结构中相距越远，绑定所耗费的时间就越长。由用户来选择静态绑定与动态绑定的缺点，是必须将这种选择决定包括进最初设计之中，但往往在后面可能又会需要修改这个决定。

C++的静态类型检测与Smalltalk的相比，是一种主要的优越性；在Smalltalk中，所有类型检测都是动态的。Smalltalk的程序可以与发送到不存在方法的消息一起编译，直到程序被执行时才能够发现这个错误。C++编译器则能够发现这种错误。编译器发现的错误比在测试时发现的错误的改正代价要低。

Smalltalk基本上是无类型的，这意味着所有代码实际上是通用的。这提供了极大的灵活性，但牺牲了静态类型检测。C++通过其模板设施提供了通用的类（如在第11章中描述的），这种类保持了静态类型检测的优越性。

Smalltalk的主要优点在于语言的优雅与简单，这归功于其单一的设计思想。这种语言纯粹而且毫无保留地贡献于面向对象的范型，完全没有因为要固守用户基础而被迫妥协的必要。然而C++是一种庞大而复杂的语言，除了支持面向对象程序设计以及包括C的用户基础之外，不具有单一的设计思想来作为基础。它的最重要的目标之一，是在提供面向对象程序设计的优越性的同时，保留C语言的效率与风格。有些人认为这种语言的一些特性并不总是相互适用的，并且它的复杂性多半也是不必要的。

根据Chambers and Ungar（1991），Smalltalk运行了一套小型的C风格的基准测试程序，它只具有优化的C的10%的速度。C++程序只需要比等同的C程序稍微多一点的时间（Stroustrup，

1988)。在Smalltalk和C++之间效率上有很大差别的前提下，C++在商业上的使用远远超出Smalltalk就毫不奇怪了。当然，造成这种局面还有其他一些因素，但是效率显然是支持C++的一个重要原因。 529

访谈



关于程序设计范型与更优良的程序设计

BJARNE STROUSTRUP

Bjarne Stroustrup是C++的设计者和最早的实现者，也是《C++程序设计语言》和《C++的设计与发展》两书的作者。他的研究兴趣包括分布式系统、模拟、设计、程序设计和程序设计语言。Stroustrup博士是美国Texas A&M大学工学院的计算机科学教授。他积极参与了ANSI/ISO的C++标准化的工作。在A&M大学工作了二十多年后，他仍然保持着与AT&T实验室的联系，作为信息与软件系统研究实验室的成员来进行科学研究。他是ACM的高级成员、AT&T贝尔实验室的高级成员，以及AT&T实验室的高级成员。1993年，"由于他早在对于C++程序设计语言的贡献上"，在此基础之上，Stroustrup博士继续的努力使得C++已经成为计算机历史上最有影响力的程序设计语言"，Stroustrup获得了Grace Murray Hopper奖。

1. 程序设计范型

问：请谈谈你对于面向对象程序设计范型的看法，它的优点和缺点。

答：首先让我谈谈我的"面向对象程序设计范型"的意思是什么。许多人认为"面向对象"就是"好"的同义词。如果是这样的话，那还需要其他的范型干什么。面向对象的关键是使用类的层次，通过大致相同的虚拟函数以提供多态行为。面向对象的重要一点是避免对层次中数据的直接访问。数据访问只能通过良好设计的函数接口。

除了它众所周知的优点外，面向对象程序设计也有缺点。特别是，并非每一种概念都适合于类的层次。并且，比较其他的程序设计范型，支持面向对象程序设计的机制会显著地增加开销。对于许多简单的抽象，不依赖于层次与运行时绑定的类，提供一种更加简单和高效的方式。另外，在不需要运行时决策的地方，依赖于（编译时）参数多态的通用程序设计是更好和更高效的方式。

问：C++是面向对象的还是其他什么的？

答：C++支持几种程序设计范型——包括面向对象程序设计、通用程序设计和过程程序设计。这些程序设计范型的结合定义了多范型的程序设计，以支持多种程序设计风格（范型）以及那些风格的结合。

问：你能够给出一个多范型程序设计的"迷你"例子吗？

答：考虑经典的"形状的集合"例子的一个变种（此例最早使用在第一个支持面向对象程序设计的语言：Simula67之中）。

```
void draw_all(const vector<Shape*>& vs)
{
    for (int i = 0; i<vs.size(); ++i)
        vs[i]->draw();
}
```

在这里，我将通用包含器vector与多态类型Shape一起使用。vector提供静态的类型安全性和最优的运行时性能。Shape提供在不重新编译的情况下处理一个形状（Shape派生类的对象）的能力。

我们可以很容易地将它一般化为任何满足C++标准库要求的包含器。

```
template<class C>
        void draw_all(const C& c)
{
    typedef typename C::
        const_iterator CI;
    for (CI p = c.begin();
```

```
        p!=c.end(); ++p)
        (*p)->draw();
}
```

使用循环器能够使我们将draw_all()应用于不支持下标的包含器之上，如标准库的链表：

```
vector<Shape*> vs;
list<Shape*> ls;
// . . .
draw_all(vs);
draw_all(ls);
```

我们甚至还可以将它进一步一般化，以便能够处理任何由一对循环器定义的元素序列：

```
template<class Iterator> void
draw_all(Iterator b, Iterator e)
{
    for_each(b,e,mem_fun(Shape::Draw));
}
```

为了简化实现，我使用了标准库算法for_each。

我们可以对标准库链表和数组来调用这个最后版本的draw_all()。

```
list<Shape*> ls;
Shape* as[100];
// . . .
draw_all(ls.begin(),ls.end());
draw_all(as,as+100);
```

11. 为一项工作选择"正确"的语言

问：在多种不同程序设计范型中的知识背景有用吗？或者说，投入时间去进一步熟悉这门对象语言更好呢，还是学习其他范型的语言更好？

答：对任何希望被认为是软件领域中的专业人员的人来说，会用多种语言和多种程序设计范型是很关键的。当前，C++是最好的多范型语言，也是学习各种形式的程序设计的好语言。但是，仅仅知道C++是不够的。只知道一种程序设计范型语言是不好的。这有点像色盲或只会说一种语言的人。你很难知道你遗漏或忽略了什么东西。好的程序设计灵感来自于已学到的，并且能够综合多种程序设计风格，而且知道怎样将这些风格使用于不同的语言之上。

520
|
521

此外，我认为任何并且极不足道的程序的程序设计是具有坚实而广泛教育背景的专业人员的工作，而不是那些只经过初步而又缺乏的"培训"能人的工作。

12.6 Java对面向对象程序设计的支持

Java中的类、继承和方法的设计与C++中的相类似，所以我们在这一节里仅仅将注意力集中于Java与C++的不同之处。

12.6.1 一般特征

如同在C++中一样，Java并不仅仅使用对象。然而在Java中，只有原始数量类型（布尔类型、字符类型及数值类型）的值不是对象。Java中的枚举和数组是对象。效率是使得Java具有非对象实体的理由。然而，如在12.3.1节中所讨论过的，存在着的两种类型系统导致了一些不方便的情形。在Java中一种情形是预定义的包含器的类，例如ArrayList只能包含对象。因而如果你想要将一个原始类型的值放入ArrayList的对象，就必须首先将这个值放置于一个对象之中，在Java 5.0之前的版本中，这可以通过为这个原始类型创建一个新的包装类的对象来完成。这样

的一个类具有这个原始类型的实例变量以及一个构造器，它接受原始类型的值作为参数，并将这个值赋给它的实例变量。例如，如果要将10放入变量myArray所引用的ArrayList对象，我们就可以使用下面的语句：

```
myArray.add(new Integer(10));
```

这里add是ArrayList的一个插入新元素的方法，Integer是int的包装类。当将一个值从myArray移出并赋给一个int变量时，这个值的类型必须转换回int类型。

在Java5.0中，这种工作变得容易，当把一个原始类型的值放在对象的环境中时，先对这个值进行隐式的强制转换，如作为参数传送到ArrayList的add方法。由强制转换基本值为基本值类型包装类的对象。例如，把int值或变量放入对象的环境中会引起创建int基本类型值的Integer对象。这种强制转换称为包装（boxing）。例如在Java5.0中，下面的语句是合法的：

```
myArray.add(10);
```

520

编译器完成从int到Integer的包装。

同样地，当一个值从myArray移出并赋给一个int变量时，它的类型被隐式地转换为int类型。

虽然可以将C++中的类定义为不在父类，在Java中的类却不可以这样来定义。Java中的所有类都必须是根类Object的子类，或者是Object的后代类的子类。设置单个根类的原因之一是因为有一些操作是普遍需要的。这些操作中的一种就是一个比较对象是否相等的方法。

所有的Java对象都是显式堆动态的，大多数是用new操作符来分配，但是却没有显式解除分配操作符。废料收集被用于存储空间的回收，如同其他许多语言特征一样。虽然废料收集可以避免一些严重的问题，如虚悬的指针等，但也会引起其他的一些问题。其中的一个问题是废料收集者只回收对象所占用的存储空间，而不进行其他的一些事情，例如一个对象没有访问所存储空间而是访问别的资源，如文件或共享资源上的锁，废料收集器就不会回收这些东西。为了处理这些情况，Java中包括了与C++中的析构函数相关的一个特殊方法，finalize。

当废料收集器要收集对象占用的存储空间时，finalize方法则被隐式地调用。finalize方法的问题是它所运行的时间是不可强制甚至是不可预测的，finalize方法的一种替代是定义一个回收方法，这样做的唯一问题是对象的所有客户必须知道有这么一个方法，并且必须能够记住调用它。

12.6.2 继承

在Java中，一个方法可以被定义为final，这意味着这个方法不能够被覆盖于任何后裔类之中。当使用保留字final来对一个类的定义进行说明时，这意味着这个类不能够是任何子类的父类。

Java仅仅是直接支持单继承，但是Java包括了一种被称为接口的虚拟类，它提供了一种多继承的版本，接口的定义与类的定义相似，只是接口仅仅能够包括命名常数以及方法声明（而不是定义）。在不能包含构造器或非抽象方法。因而正如接口的名字所指示的那样，它只是定义类的说明。（回忆C++的抽象类可以具有实例变量，并且可以完全定义除了一个方法之外的所有方法。）类不继承接口，它实现它。实际上，类能够实现任意数量的接口。为了实现接口，类必须实现所有规格出现在接口定义中的方法。

接口能用来模拟多继承，带有替代第二个父类的接口的类能从类中派生并实现接口。有时，这称为混合（mix-in）继承。由为接口的常量和方法与从超类继承的方法和数据混合在一起。

531

任何在子类中定义的新数据和方法也一样。

接口的另一个有趣的功能是它们提供了另一种多态，这是因为接口能处理类型。例如，方法能指定是接口的形参。这种形参能接受任何接口实现的类的实参，这使方法具有多态性。

非参数变量也能声明为接口的类型。这种变量能引用任何接口实现的类的任何对象。

当一个类从两个父类派生，并且都用同样的名称和协议定义了公有方法时，一个多继承的问题将会出现。因为实现接口的类必须提供接口指定的所有方法的定义，所以此问题可用接口来解决。如果父类和接口都包括带有同样名称和协议的方法，那么子类必须重新实现该方法，而且，在多继承出现的命令冲突在单继承和接口中不会出现。

接口并不是多继承的替代，因为多继承提供代码重用，而接口却不能提供。这是它们间重大的不同处，因为代码重用是继承最大的优点。

作为接口的一个示例，我们考虑标准Java类Arrays的sort方法，任何使用这个方法的类都必须提供一个方法的实现来比较被排序的元素。Comparable接口对这个用来比较元素的方法提供协议，这个方法称为CompareTo。Comparable接口的代码如下：

```
public interface Comparable {
   public int compareTo(Object b);
}
```

如果调用CompareTo方法的对象是在参数对象之前，这个方法将返回一个负整数。如果相等的话，则返回零值，如果调用CompareTo方法的对象是在参数对象之后，这个方法将返回一个正整数。实现Comparable接口的类，可以对于任何数组的内容进行排序，只要实现的CompareTo方法提供合适的值。

第14章描述Java中如何使用接口来进行事件处理。

12.6.3 动态绑定

在C++中必须将一个方法定义为虚拟的，以便允许动态绑定。在Java中，所有的方法调用都是动态绑定的，除了这个被调用的方法已经被定义为final以外；而且在这种情况下这个方法就不能够被覆盖。并且所有的绑定都是静态的，如果一个方法是static或private的，也将使用静态绑定。static和private的方法不能够被覆盖。

532

12.6.4 被嵌套的类

Java具有好几种被嵌套的类。这些类的优点之一，是除了嵌套它们的类而外，对于包中的其他的类，它们是不可见的。直接定义在另一个类中的非静态类，有一个指向嵌套类的隐式的指针。这个指针允许被嵌套的类的方法访问嵌套类中的所有成员。如果一个被嵌套的类是静态的，则不具有这个指针，因而就不能访问嵌套类中的成员。因此Java中的静态的被嵌套的类十分类似于C++中被嵌套的类。

被嵌套的类也可以是匿名的。匿名的类具有复杂的语法，如果一个类只在一个地方使用，这就是一种练的方法。

也可以将一个局部被嵌套的类定义在嵌套类的方法之中。局部被嵌套的类从来不使用访问说明符（如private或public）来进行定义。总是将它们的作用域限制在它们的嵌套类之中。一个局部被嵌套的类中的方法，可以访问定义在其嵌套类中的变量以及定义在其方法中的final变量。只有定义局部被嵌套的类的方法才可以访问其内部的成员。

12.6.5 评估

Java的支持面向对象程序设计的语言设计与C++中的相类似，然而Java与面向对象的原则紧密地保持着一致。由于Java中不具有函数，所以Java不支持过程程序设计。此外，Java不允许没有父类的类，它还使用动态绑定作为一种“正常”的方式来将方法的调用绑定于方法定义。比较C++中的访问控制之复杂情形（从派生控制到友元函数），Java中对于类定义内容的访问控制则是十分简单的。最后，Java使用接口提供了一种简单的形式以支持多继承，而C++则包括了完整然而复杂的形式来支持多继承。

12.7 C#对面向对象程序设计的支持

C#对于面向对象程序设计的支持与Java中的相类似。

12.7.1 一般特征

正如在第11章中所讨论的，C#包括了类和结构（struct），C#中的类与Java中的类十分相似，其中的struct为能力稍差的浅动态结构。

12.7.2 继承

C#使用C++的文法来定义类，例如，

```
public class NewClass : ParentClass { ... }
```

从父类继承的方法，可以在派生的子类中被替代，这需要在子类中的替代方法前面加上new，对于一般的访问，这个带有new的方法将父类中相同名称的方法隐藏起来。但只需要在方法名称上加以前缀base，就仍然可以访问父类中的方法，例如，

```
base.Draw();
```

与Java相同，C#支持接口。

12.7.3 动态绑定

在C#中，要将方法调用动态地绑定于方法，必须将基方法（即原定义的方法）和它在派生类中的对应方法都给以标记，如同在C++中的那样，必须将基方法标上virtual，为了避免事故性的覆盖，必须将派生类中的对应方法都标上override，override标记将清楚地指示出，这是所继承方法的一个新的版本。例如，下面是12.5.3节中的C++ Shape类的C#版本：

```
public class Shape {
  public virtual void Draw() { ... }
  ...
}
public class Circle : Shape {
  public override void Draw() { ... }
  ...
}
public class Rectangle : Shape {
  public override void Draw() { ... }
  ...
}
public class Square : Rectangle {
```

```
    public override void Draw() { ... }
    ...
}
```

534 C#中包含类似于C++中的抽象方法，但是使用不同的文法来说明。例如，下面是一个C#的抽象方法：

```
abstract public void Draw();
```

包含至少一个抽象方法的类是抽象类，每个抽象类必须使用abstract来标记。抽象类不能被实例化。将要实例化的抽象类的任何子类都必须实现其继承的所有抽象方法。

如同在Java中的那样，所有C#的类都从根类Object派生而来，根类Object定义了一组方法，包括ToString、Finalize和Equals，所有C#的类型都继承这些方法。

12.7.4 嵌套类

在C#中，一个类可以直接嵌套在另一个类之中，被嵌套类的行为与Java中的静态嵌套类的行为相类似。（后者的行为又与C++中的嵌套类的行为相类似。）C#不支持类似于Java中的非静态嵌套类。

12.7.5 评估

因为C#是近年设计的基于C的语言，人们应该能够想像到，C#的设计人员从前面语言的设计中学习了很多东西，他们可以继续过去成功的经验，而去除一些前面语言中的问题。这样的一种结果即是，附加上了Java中的少量问题，在对于面向对象程序设计的支持上，C#与Java的差别相对较小。

12.8 Ada 95对面向对象程序设计的支持

Ada 95基于Ada 83，并在Ada 83的基础上进行了一些重要的扩展。这一节将简略地探讨被设计来支持面向对象程序设计的这些扩展。正如在第11章里曾经讨论过的，Ada 83已经包括了构造抽象数据类型的结构，余下的必要特性就是那些支持继承与动态绑定的特性。设计这些特性的目的包括对于Ada 83的类型与包的结构所需要的尽可能小的改变，以及保持尽可能多的静态类型检测。

12.8.1 一般特征

Ada 95中的类是一种被称为标志类型（tagged type）的新类型，它们可以是记录类型也可以是私有类型。它们被定义在包之中，这就允许它们被分别编译。之所以将它们命名为标志类型，是因为标志类型的每一个对象都隐式地包括了一个指示其类型的、由系统来维护的标志。

535 定义标志类型上的操作的子程序，出现在与类型声明相同的声明表中。考虑下面的这个例子：

```
package Person_Pkg is
  type Person is tagged private;
  procedure Display(P : in Person);
  private
    type Person is tagged
      record
        Name : String(1..30);
        Address : String(1..30);
        Age : Integer;
```

```
        end record;
end Person_Pkg;
```

这个包定义了类型Person，这种类型本身可以作为类来使用，也可以用来作为其他派生类的父类。

不像在C++中的那样，Ada 95中不具有构造器或析构器子程序的隐式调用，尽管也可以编写这样的子程序，但是这些子程序必须由程序人员来显式地调用。

12.8.2 继承

Ada 83仅仅支持派生类型与子类型中的具有受限形式的继承。在这两种受限形式的继承中，可以在已有类型的基础之上定义一种新的类型，但是唯一被允许的修改只是限制新的类型值的范围。

这不是面向对象程序设计所需要的完全的继承。Ada 95支持了面向对象程序设计。

Ada 95中的派生类是以标志类型为基础的。通过包括进一种记录定义，可以将新的实体增加到被继承的实体之中。考虑下面这个例子：

```
with Person_Pkg; use Person_Pkg;
package Student_Pkg is
  type Student is new Person with
    record
      Grade_Point_Average : Float;
      Grade_Level : Integer;
    end record;
  procedure Display(St : in Student);
end Student_Pkg;
```

在此例中，派生类型Student被定义为具有其父类Person的实体以及新的实体Grade_Point_Average和Grade_Level。这个例子也重新定义了过程Display，这个新的类被定义于一个分离的包中，这样就允许了对于它所进行的修改将不会引起包含其父类型定义的包的重新编译。

这种继承机制没有办法阻止父类的实体被包括在派生类之中。其后果是，派生类只能够扩充父类，并因此成为子类型。然而下面将要简略讨论的子类包，能够被用来定义不是子类型的子类。 536

假设我们具有下面的定义：

```
P1 : Person;
S1 : Student;
Fred : Person:= ("Fred", "321 Mulberry Lane", 35);
Freddie : Student:= ("Freddie", "725 Main St.", 20, 1.25,
3);
```

因为Student是Person的子类型，赋值语句

```
P1 := Freddie;
```

应该是合法的；事实上它也是合法的。Freddie的两个实体Grade_Point_Average和Grade_Level在必需的强制转换中被省略。

现在一个明显的问题是，相反方向的赋值是否也是合法的；也就是说，我们能够将一个Person赋给一个Student吗？在Ada 95中，只要所赋的值包括了子类的实体，就是合法的。在我们的例子中，下面的形式是合法的：

```
S1 := (Fred, 3.65, 2);
```

要派生一个不包括所有父类的实体的类，就需要使用子库包。子库包是一个使用父库包的名字作为其名字的前缀的包。也可以将子库包用于C++中友元定义的位置。例如，如果必须编写一个能够访问两个不同类的成员的子程序，父包就可以定义其中的一个类，而子包则可以定义另外的一个类。然后一个子包中的子程序就能够访问这两个类的成员。

Ada 95不提供多继承。虽然通用类和多继承是两个相差甚远的概念，但使用通用类也可以达到多继承的效果。然而这个方法却不如C++中的那么优雅。关于这一点，我们将不在这里进行讨论。

12.8.3 动态绑定

Ada 95在标志类型中提供了过程调用与函数定义的静态绑定与动态绑定，通过使用类范围(classwide)类型来强制进行动态绑定，这种类型代表了一个类层次中的所有类型，这个类层次的根是某一类型，每一个标志类型都隐含地有一个类范围类型，对于标志类型T，这种类范围类型被使用T'class来说明，如果T是一个标志类型，那么一个T'class的变量的类型可能是T也可能是T的任何派生类型。

537

在这里我们再一次来考虑12.8.2节中定义的Person和Student两个类。假设我们有一个变量Pcw，其类型为Person'class。这个变量有时引用一个Person对象，有时引用一个Student对象。进一步地假设我们想要显示Pcw所引用的对象，而不论它是Person对象还是Student对象。这就要求必须将对于Display的调用动态地绑定到正确的Display版本上。我们可以使用一个新的过程来接受一个Person类型的参数，并将这个参数传送给Display。下面就是这样的一个过程：

```
procedure Display_Any_Person(P: in Person) is
  begin
  Display(P);
  end Display_Any_Person;
```

这个过程可以被以下两个调用来调用：

```
with Person_Pkg; use Person_Pkg;
with Student_Pkg; use Student_Pkg;
P : Person;
S : Student;
Pcw : Person'class;
...
Pcw := P;
Display_Any_Person(Pcw);  -- call the Display in Person
Pcw := S;
Display_Any_Person(Pcw);  -- call the Display in Student
```

Ada 95也支持多态指针，这些多态指针被定义为具有类范围类型，例如

```
type Any_Person_Ptr is access Person'class;
```

在Ada 95中，通过将保留字abstract包括进类型定义和子程序定义中，从而可以定义纯抽象基类型。此外，子程序定义不能够具有体。考虑下面的例子：

```
package Base_Pkg is
  type T is abstract tagged null record;
  procedure Do_It (A : T) is abstract;
end Base_Pkg;
```

12.8.4 子程序包

可以将一个程序包直接嵌套在其他的程序包中，这时就称之为子程序包（child package）。这种设计的问题是，如果一个程序包有着很大数目的子程序包，而且这些子程序包都很大，这 538
个嵌套程序包就会变得很大，以致不能够成为一个有效的编译单位。这个问题的解决方案比较简单：允许子程序包成为单独的单位，并且可以被分开编译。子程序包的名字就是嵌套程序包的名字，再加上子程序包自己的名字，中间使用句号分隔开。例如，如果嵌套程序包的名字是Binary_Tree，子程序包自己的名字是Traversals，那么子程序包的全名就是Binary_Tree.Traversals。

子程序包可以是公有的（默认）或私用的。一个公有的子程序包的位置，是在嵌套程序包的说明包之声明部分的尾部。因此，在嵌套程序包的说明包中所声明的所有实体，对子程序包都是可见的。然而，任何出现在嵌套程序包的体中的声明，对于子程序包都是不可见的。如果有多个子程序包，它们之间没有逻辑上的顺序。所以，一个子程序包不能够看见另一个子程序包的声明，除非它包含有一个带有其他子程序包名字的with子句。

将保留字private放在保留字package之前，就可以将一个子程序包声明为私用的。一个私用子程序包的逻辑位置是在嵌套程序包的说明包的声明部分的首部。对于嵌套程序包的体，一个私用子程序包的声明是不可见的，除非嵌套程序包含有带有该子程序包名字的with子句。

12.8.5 评估

Ada对面向对象程序设计提供了完全的支持。尽管其他面向对象程序设计语言的用户会觉得Ada的支持是弱的。虽然程序包可以用来创建抽象数据类型，但它们实际上是更一般化的封包构造。除非使用子程序库包，否则无法限制继承。在这种情况下，所有的子类都是子类型。相比C++、Java和C#，这种形式的访问限制是有限的。

C++显然提供了一种比Ada 95的更好的多继承形式。然而，使用子库单位来控制对父类实体的访问似乎是比C++中的友元函数和类更为清晰的解决办法。例如，如果当定义一个类时不知道是否需要友元，而当发现有这种需要时，将必须改变和重新编译这个类的定义。在Ada 95中，可以定义新的子包里的新类而不打扰父包，后出是对子程序包来说。每一个定义在父程序包中的名字都是可见的。

C++中包括用来对对象进行初始化的构造器和析构器。这是十分优良的设计。而Ada 95就没有包括这种功能。

这两种语言之间的另外一种差别，是C++的根类的设计人员必须决定，某一特定成员函数到底应该是静态绑定还是动态绑定。如果是选择静态绑定，然而系统后来又需要改变为动态绑定的话，则必须改变根类。在Ada 95中，这种设计决定不必与根类的设计联系在一起。每一个 539
调用的自身可以说明究竟会静态绑定还是动态绑定，而不论根类是如何设计的。

一种更为细微的差别是，将基于C的语言中的动态绑定仅仅限制于指针和(或)对象的引用，而并非对象的自身，Ada 95中就没有这种限制。总而在这种情况下Ada 95更为正交化。

12.9 Ruby对面向对象程序设计的支持

正如前面讲述的，与Smalltalk类似，Ruby是一种纯面向对象程序设计语言。它虚拟化所有一切为对象，并且通过消息传递来完成所有计算。尽管程序有使用中缀操作符的表达式，还有与Java语言相似的表达式外观，实际上这些表达式通过消息传递来求值。当计算a+b时，它发

送消息+给a引用的对象，并传递引用给对象b。

12.9.1 一般特征

回忆第11章，Ruby类定义与C++和Java等语言不同，因为它们都是可执行的。这允许它们在执行期间保持开放性，允许程序通过简单提供包含新成员的类的二次定义来多次添加类。在执行期间，类的当前定义是已经执行的类的所有定义的联合。方法定义也是可执行的，这使程序在执行期间通过简单地把两种定义放入选择结构的then和else分句里来选择一种方法定义。

Ruby的所有变量都是对对象的引用并且都是无类型的。所有实例变量名都以标记@开头。Ruby与其他常用程序设计语言的明显区别是，Ruby的访问控制在访问数据和访问方法上不同。
543 所有实例数据默认下都是私有访问的，并且不能改变。如果需要对实例变量进行外部访问，必须定义访问方法。例如，考虑下面的类定义框架：

```
class MyClass

# 构造器

  def initialize
    @one = 1
    @two = 2
  end

# 获取@one

  def one
    @one
  end

# 设置@one

  def one=(my_one)
    @one = my_one
  end

end  # MyClass类
```

设置方法名后连接=标记说明它的变量是可赋值的。因此，所有设置方法都有等号连接它们的名称。one获取方法体说明当没有返回语句时Ruby设计为让方法返回上个表达式求值的结果。这个例子返回@one的值。

因为需要频繁使用获取和设置方法，Ruby提供了两者的快捷方式。如果一个类有两个实例变量的获取方法@one和@two，那么这些获取方法能通过类中单一语句来指定

```
attr_reader :one, :two
```

attr_reader实际上是使用:one和:two作为实参的函数调用。在变量前加冒号：表示变量名被使用，而不是取消其引用对象的引用。

类似地创建设置方法的函数称为attr_writer。该函数有与attr_reader相同的参数列表。

创建获取方法和设置方法的函数都是指定的。因为它们为类的对象提供了协议（在Ruby中称为属性），因此类的属性是对类对象的数据接口（公有数据）。

因为对Ruby方法的访问控制是动态的，所以访问法例只能在执行时检测到。默认的方法访问是公有的，但是它也能是保护的或私有的。指定访问控制有两种方式，它们都使用带有同样

名称的函数作为访问控制（private、protected和public）。一种方式是调用没有参数的合适的函数，这重设了随后在类中定义的方法的默认访问权限。例如，

```
class MyClass
  def meth1
  ...
  end
  ...
private
  def meth7
  ...
  end
  ...
protected
  def meth11
  ...
  end
  ...
end  # of class MyClass
```

541

可替换的方法是调用把具体方法名作为参数的访问控制函数。例如，下面的类定义在语义上是与前面等价的：

```
class MyClass
  def meth1
  ...
  end
  ...
  def meth7
  ...
  end
  ...
  def meth11
  ...
  end
  ...
  private :meth7, ...
  protected :meth11, ..
  end  # of class MyClass
```

类变量通过在它们名称前加两个标记（@@）来指定。它们对类和其实例是私有的。私有性不能改变。与全局变量和实例变量不同，类变量在使用前必须初始化。

12.9.2 继承

Ruby定义子类采用小于号（<），而不是在C++使用的冒号。例如，

```
class MySubClass < BaseClass
```

542

Ruby方法访问控制的一个不同之处在于通过简单调用访问控制函数能在子类中改变它们。这意味着基类的两个子类都能被定义，以致于一个子类的对象能访问基类定义的方法，但是另一个子类的对象却不能。这也允许改变基类中公有访问方法的访问为子类的私有访问方法。这种子类明显不能为子类型。

回忆第11章讨论过的Ruby模组。它们定义了一种经常用于定义库数据的命令封装。然而，也许模组最有趣的方面是能从类直接访问它们的函数。在类中访问模组用include语句来指定。

例如

```
include Math
```

包含模组的效果是使类获得了指向模组的指针，并且有效地继承了模组中定义的函数。实际上，当类包含了模组时，模组变成了类的一个代理超类。这种模组称为混入（mixin），因为它的函数混入了类中定义的方法。混入提供一种在任何需要它的类中包含模组功能的方法。当然，类仍然有一个正常的、能继承成员的超类。因此，混入提供了多继承的优点，而且不会出现如果模组在它们函数中不需要模组名时可能出现的命名冲突。

12.9.3 动态绑定

Ruby对动态绑定的支持与Smalltalk相同。变量不能类型化。它们都是对任何类的对象的引用。因此，所有变量都是多态的。所有方法调用绑定都是动态的。

12.9.4 评估

因为Ruby是最纯的面向对象程序设计语言，所以它对面向对象程序设计的支持是彻底的。然而，它对类成员的访问控制比C++更弱。它也不支持多继承。最后，尽管混入与接口紧密相关联，但是Ruby不支持抽象类或接口。

12.10 JavaScript的对象模型

虽然JavaScript中不具有类，也不支持继承与动态绑定，但它使用一个松散地基于C++和
515 Java的对象模型。JavaScript的设计因此成为使用语言支持对象的传统概念的一种有趣的替代。

12.10.1 一般特征

JavaScript原本是由Netscape设计来用于Web服务器的程序设计的脚本语言。它自此而发展并演化成为地位显著的语言，它补充HTML文档，用来提供这些文档的计算能力。JavaScript也用来定义动态HTML文档。并且，在表格数据送往Web服务器之前，JavaScript用来对之进行验证。正是出于这一点，JavaScript才成为了一种流行的语言。

尽管它的名字的第一个部分是Java，但JavaScript与Java却没有共同点。然而，它的名字的后面部分才的确是真的——它是一种脚本语言。JavaScript 代码的片段可以散布于HTML文档中的不同位置。当浏览器在文档中遇到JavaScript代码时，代码即刻被解释。

JavaScript与Java的相似仅仅在于它们使用类似的语法，但它们有着许多根本的差别。Java是一种静态类型的面向对象语言；而JavaScript是动态类型的并且不是面向对象的。JavaScript中没有类，它的对象既是对象又是对象类型，因为不具有类，JavaScript就不能够支持基于类的继承，而没有基于类的继承，JavaScript就不能够支持多态。虽然JavaScript有对象，但是它们与面向对象语言中的对象非常不同。

如同大多数支持面向对象程序设计语言一样，JavaScript具有两类变量，引用对象的变量和直接存储原始类型值的变量。变量可以被声明，但声明不强制类型。当每一次给一个变量赋值时，都能够改变它的类型。而新的类型就是所赋的值的类型。可以使用任何变量来存储任何原始类型的值或者是来引用任何对象。

12.10.2 JavaScript对象

JavaScript中的对象具有一系列性质，这些性质对应于Java与C++中类的成员。每一个性质

或者是一个数据的性质，或者是一个方法的性质。

JavaScript的对象无论是内部还是外部都很像Perl中的散列（见第6章）。每一个对象都是一系性质一值的对。其中的性质为名字，而值则是数据的值或是对于对象的引用，其中的一些是方法。所有的函数与方法都是对象，并且都是通过变量来引用的。函数与方法之间的唯一差别是，方法是通过对象的性质来引用的。

如同Perl中的散列，一个JavaScript对象的性质的系列是动态的，即在任何时刻，性质都可以被增加或被删除。这使得它们非常不同于C++和Java这样的语言中的对象。尽管在任何正式的意义上对象并不具有类型，每一个对象都以它的性质系列为特征。JavaScript包括一个typeof操作符，它对任何对象都返回字符串"object"。 544

12.10.3 对象的创建与修改

new表达式常常被用来创建对象，对象的创建必须包括对构造器方法的调用。被new表达式所调用的构造器产生刻画新对象的性质。在面向对象语言中，new操作符产生一个特定对象，这意味着一个具有类型及一组特殊成员的对象。在这种语言中的构造器将成员初始化，但并不创建这些成员。在JavaScript中new操作符创建一个空的对象，即不具有性质的对象。构造器创建这些性质，并将这些性质初始化。

下面的语句使用预定义的Object对象的构造器来创建一个对象，但它并不产生性质：

```
var my_object = new Object();
```

现在变量my_object引用这个新的对象。下面进一步讨论构造器。

一个对象的属性是使用点标记法来访问的，其中的第一个字是对象名称，第二个字则是属性名称。属性实际上并非变量。它们是进入散列的关键字，正如Perl的散列关键字一样，它们自身并不具有值。它们被用来与对象的变量一起访问属性的值，这为属性不是变量。它们从来不需要被声明。

如上所述，在解释期间的任何时刻，可以从一个对象中增加或者删除属性。通过给一个属性赋值，从而来创建这个对象的属性。考虑下面的例子：

```
var my_car = new Object();  // 产生一个空对象
my_car.make = "Ford";  // 产生并且初始化make
my_car.model = "Contour SVT";  // 产生并且初始化model
```

这段代码创建了一个具有两个属性：make与model的新对象my_car。因为对象可以被改变，所以我们能够创建一个为my_car的属性的新对象，这个对象自身又有着自己的一些属性。例如，

```
my_car.engine = new Object();
my_car.engine.config = "V6";
my_car.engine.hp = 200;
```

如果企图访问一个并不存在的对象的属性，则使用undefined的值。使用delete指令能够删除属性，例如，

```
delete my_car.model;
```

JavaScript的构造器是一些特殊的方法，它们为新的对象创建属性，并设定这些属性的初值。正如前面讲述的，每一个new表达式必须包括一个对于构造器的调用，构造器实际上由new操作符来调用的，在new表达式中的new操作符置于构造器名称之前。

构造器虽然必须能够引用它所操作的对象。JavaScript为此目的而具有名为this的预定义的引用变量。当调用构造器时，this是对于新创建的对象的引用。this变量被用来创建对象的属性，并且设定这些属性的初值。例如，考虑下面的构造器：

```
function car(new_make, new_model, new_year) {
    this.make = new_make;
    this.model = new_model;
    this.year = new_year;
}
```

这个构造器也可以作如下使用：

543
```
my_car = new car("Ford", "Contour SVT", "2000");
```

如果要将一个方法包含到对象中，给它设定初值的方式就与给数据性质设定初值的方式相同。例如，假设我们为显示属性值的car对象创建了一个方法，命名为display_car。通过给构造器增加下面的这行指令，display_car方法被加入到new所创建对象以及构造器的调用之中：

```
this.display = display_car;
```

JavaScript中唯一的与面向对象程序设计语言中的类相关的概念是对象种类的概念。一个对象种类是通过使用同一个构造器来创建的一组对象。所有这样的对象都具有一套相同的属性和方法，至少是在这些对象产生的初期。然而，在脚本中却没有方便的办法来确定两个对象是否具有一套相同的属性与方法。

12.10.4 评估

就语言的设计目标而言，JavaScript是一种十分有效的脚本语言。如果是被设计来开发大规
546 模的软件系统，它的对象模型则是完全不够用的。由于没有类的封装功能，在JavaScript中不能够有效地组织大型的程序。没有继承，复用将会十分困难。比此，虽然JavaScript中的对象模型是有趣的，但它不能够满足先进面向对象程序设计语言发展的需要。

12.11 面向对象结构的实现

在支持面向对象程序设计的语言中，至少其中有两个部分包含了语言的实现人员所感兴趣的问题，即实例变量的存储结构以及消息对于方法的动态绑定。在这一节中，我们将简略地探讨这两个问题。

12.11.1 存储实例数据

在C++中，将类定义为是对于C中的记录结构（即struct）的一些扩展。它们之间的这种相似性提示我们，可以将记录结构作为类实例变量的存储结构。我们称这种形式的结构为类实例记录（class instance record，CIR）。CIR的结构是静态的，当比它在编译时被建造，并且被用来作为产生类实例的模板。每个类都有它自己的CIR。当派生类时，子类的CIR是父类的拷贝，新的实例变量项添加到末尾。

因为CIR的结构是静态的，可以使用从CIR实例开端的常量偏移来对所有实例变量进行访问，就如同在记录中的那样。这种功能使得这些访问具有与在记录的域中访问一样的高效率。

12.11.2 消息对方法的动态绑定

静态绑定的类中的方法并不需要是在这个类的CIR中涉及。然而，动态绑定的方法必须在

CIR结构中具有入口，这种入口可以仅仅是指向方法的代码的一个指针，必须将它设立于创建对象之时。然后对于方法的调用就可以通过这个CIR中的指针连接到相对应的代码之上。这种技术的缺点是，每一个实例都需要存储指向可能从这个实例调用的所有动态绑定方法的指针。

注意，可以从类的一个实例来调用的动态绑定方法对于这个类的所有实例都是相同的。因此，只应该将这些方法的表存储一次，因而CIR只需要一个指针来指向这个表，使得能够找到被调用的方法。有时将这个表的存储结构称为虚拟方法表格（virtual method table，简称为vtable）。可以使用VMT中的偏移来表示方法的调用。一个祖先类的多态变量总是能够引用正确类型对象的CIR，从而确保了能够得到动态绑定方法的正确版本。考虑下面的Java示例，其 547
中的所有方法都被假设为动态绑定的：

```
public class A {
  public int a, b;
  public void draw() { ... }
  public int area() { ... }
}
public class B extends A {
  public int c, d;
  public void draw() { ... }
  public void sift() { ... }
}
```

图12-2显示了类A与类B的CIR以及它们的vtables。注意，在B的vtable中area方法的方法指针指向A的area方法的代码。因为B没有覆盖A的area方法。所以如果B的客户端调用area，area方法是从A继承的。另一方面，在B的Vtable中draw和sift的指针指向B的draw和sift，B覆盖draw方法并添加sift的定义。 548

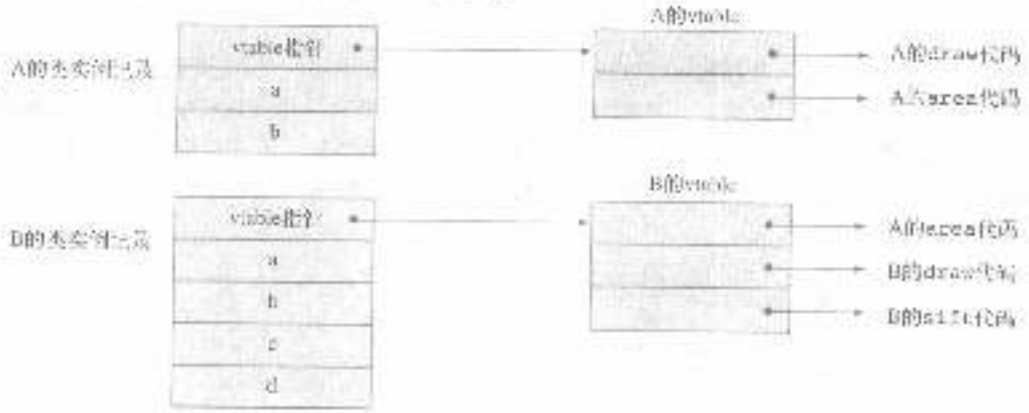


图12-2 一个具有继承的CIR的例子

假设一个被命名为C的类产生类A与类B的对象，并具有两个被分别命名为aobj和bobj的引用。进一步地假设，在类C中存在着赋值：

```
aObj = bObj;
```

现在又存在一个调用：

```
aObj.draw();
```

而且前面的赋值语句仅仅是将bObj的指针复制到aObj上，可以调用A的draw方法。然而却是不正确的。这个例子说明了，将引用bObj赋给aObj是有副作用的，它将aObj的vtable

指针改为指向B类的vtable指针。

多继承将动态绑定的实现复杂化，考虑下面的二个C++类的定义：

```
class A {
  public:
    int a;
    virtual void fun() { ... }
    virtual void init() { ... }
class B {
  public:
    int b;
    virtual void sum() { ... }
};
class C : public A, public B {
  public:
    int c;
    virtual void fun() { ... }
    virtual void dud() { ... }
};
```

类C从类A那里继承了变量a以及方法init，类C重新定义了方法fun，虽然类C中的fun及其父类A中的fun通过一个（类型A的）多态变量都有可能是可见的。类C从类B那里继承了变量b以及方法sum，类C定义了自己的变量c，并且还定义了一个非继承的方法dud，类C的一个CIR必须包括类A的数据，类B的数据、类C的数据以及访问所有方法的一些方式。在单继承的情况下，CIR 包含一个指向vtable 的指针，vtable存有所有可见方法的代码的地址。然而，在多继承的情况下，事情就没那么简单。CIR中包括至少两个不同的影像，每个影像存放一个父类的信息，而两个影像之一又存放子类C的影像。与单继承的情况相同，在存放父类信息的影像中存放子类的影像。

349

必须有两个vtable：一个存放A和C的影像，另一个存放B的影像。在这种情况下，C的CIR的第一部分可以是C和A的影像部分，其起始处是一个vtable的指针，指向C的方法和从A继承的方法，并包括从A继承的数据。在C的CIR中紧接着的是B的影像部分，其起始处是一个vtable的指针，指向B的虚拟方法，再接着是从B继承的数据和C中定义的数据。图12-3显示了类C的CIR。

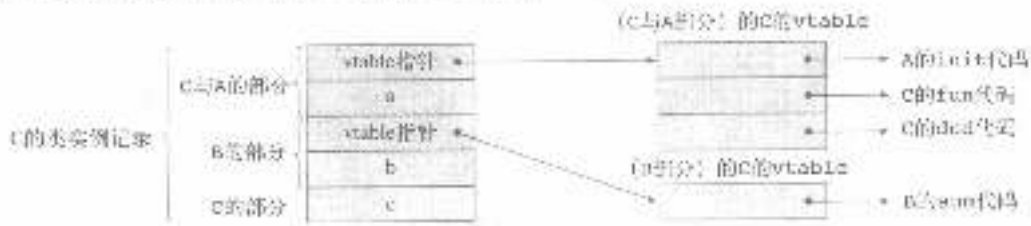


图12-3 一个具有多父类的子类CIR的示例

小结

面向对象程序设计包括了三个基本概念：抽象数据类型、继承和动态绑定。面向对象程序设计语言通过类、方法、对象和消息传递，来支持面向对象程序设计范式。

在这一章中关于面向对象程序设计语言的讨论，围绕着七个设计问题：纯对象、子类与子类型、类型检测及对象的多态、单继承与多继承、动态绑定、对象的显式与隐式解除引用以及嵌套类。

Smalltalk是一种纯面向对象语言——其中的每一个事物都是一个对象。所有计算都通过消息传递来完

成。在 Smalltalk中的所有了类都是子类型。所有的类型检测以及消息对于方法的绑定都是动态的，而且所有的继承都是单继承。Smalltalk 支持显式解除分配的操作。

C++提供了对于数据抽象、继承、消息和方法的选择性的动态绑定，以及 C 的所有传统特性的支持。这意味着它具有两个不同的类型系统，尽管Smalltalk的动态绑定，给程序设计提供了比混合式语言C++更多的灵活性，但Smalltalk的效率却要差得多。C++ 提供多继承以及对象的显式解除分配。C++包括了多种对于类中实体的访问控制，其中的一些可以使类子类成为子类型。构造器与析构器方法都可以包括在类中，这两种方法都被隐式地调用。

Java不是像C++那样的混合式语言，它是专门用来支持面向对象程序设计的。然而与C++一样，Java具有原始数量的类型与类。所有对象都从堆来分配，并通过引用变量而被访问。Java没有显式的对象解除分配的操作。Java中的唯一的子程序就是方法。只能通过对象或类来调用这些方法。尽管通过接口的使用，多继承也是可能的，但Java只是直接支持单继承。除了不能够覆盖的方法以外，所有消息对于方法的绑定都是动态的。除了类以外，Java还包括了包来作为第二种封装结构。

Ada 95通过标志类型提供了对于面向对象程序设计的支持，标志类型可以使用继承。通过使用类范围类型，动态绑定成为可能。派生类型扩展到了父类型，这非常类型是被定义于子程包之中，而在后面这一类情形之下，可以从派生类型中将父类型的实体以除掉。在子程包以外，所有子类都是子类型。

基于C++和Java的C#支持面向对象的程序设计。对象可以从类或结构来显行实例化。结构对象是栈动态的，并且不支持继承。通过在父类成员的名称前包括base，派生类中的方法就可以定义父类的这些方法。可以被覆盖的方法必须标记上virtual，以及覆盖方法即必须标记上override，所有的类，还有所有的基本类型，都派生自Object。

Ruby是一种纯本语言，其所有数据都是对象。正如Smalltalk，所有对象都是堆分配的，所有变量都是对对象的无类型的引用。所有构造器命名为initialize。所有实例数据都是私有的，但可以有容易包括获取和设定方法。已经提供的访问方法的所有实例变量形成对类的公有接口。这种实例方法称为属性。Ruby类是动态的，这意味着它们是可执行的并在任何时期是可改变的。Ruby只支持单继承，其子类不必为子类型。

JavaScript虽然不是一种面向对象的语言，但它包括了对象概念上的一种有趣的变体。方法调用和子方法的动态绑定，可以使用类实例记录和方法地址的查找表来实现。这种方式还可以被扩展，以便包括对于多继承的支持。

复习题

1. 描述面向对象语言三个特性。
2. 类变量与实例变量之间的差别是什么？
3. 什么是覆盖方法？
4. 描述一种动态绑定是极为重要的情形。
5. 什么是虚拟方法？
6. 简略描述本章讨论的面向对象语言的六个设计问题。
7. 什么是方法的消息协议？
8. 为什么Smalltalk的类能够响应消息？
9. 解释Smalltalk的消息是怎样绑定于方法的，这种绑定何时发生？
10. Smalltalk中进行什么样的类型检测？何时进行？
11. Smalltalk支持什么类型的继承，单继承还是多继承？
12. Smalltalk对计算有哪两种重要的影响？
13. 所有Smalltalk的变量实质上都是一个类型。这个类型是什么？

14. C++中的对象可以在哪里被分配？
15. C++中堆分配的对象是怎样被解除分配的？
16. 所有C++中的子类都是子类型吗？
17. 在什么情况下一个C++的方法调用可以静态地绑定于一个方法？
18. 允许设计人员来说明哪一个方法能够被静态地绑定有什么缺点？
19. 解释在C++中关于private的两种用法之间的差别。
20. 什么是C++中的friend系数？
21. Java的类型系统与C++的有什么不同？
22. Java的对象可以在哪里被分配？
23. 怎样将Java的对象解除分配？
24. 所有Java的子类都是子类型吗？
25. 在什么情况下一个Java的方法调用可以静态地绑定于一个方法？
26. C#中的覆盖方法怎样在语法上不同于C++中的覆盖方法？
27. 在C#中，当一个子类覆盖了从父类继承的一个方法时，怎样从子类中调用这个方法在父类中的版本？
28. 所有 Ada 95 的子类都是子类型吗？
29. 如何说明 Ada 95 中对一个子程序的调用是动态地绑定到一个子程序的定义？这个决定是在何时作出的？
30. Ruby如何实现基本类型，如整型和浮点型数据？
31. Ruby类如何定义获取方法？
32. Ruby支持实例变量的什么访问控制？
33. Ruby支持方法的什么访问控制？
34. 所有Ruby子类都是子类型吗？
35. Ruby支持多继承吗？
36. 在哪些基本的方面，JavaScript 对象不同于 Java 对象？
37. JavaScript 构造器和 Java 构造器之间的主要不同是什么？

552

练习题

1. 比较C++和Java中的动态绑定。
2. 比较C++和Java中的类实体的访问控制。
3. 比较C++和Ada 95中的类实体的访问控制。
4. 比较C++中的多重继承和Java中由接口所提供的多重继承。
5. 比较GJ（generic Java）和C++中的通用功能。
6. 在什么程序设计情况下，多重继承比单继承好得多？
7. 抗象数据类型的哪两个问题被继承所改善？
8. 描述一个子类型可以对其父类作哪些改变？
9. 解释继承的一个缺点。
10. 解释一种语言中所有的值都是对象的优点和缺点。
11. 当一个子类与其父类有is-a关系时，这意味着什么？
12. 一个覆盖方法的参数应与被覆盖方法的参数多紧密地匹配？
13. 明显地，Java的设计者认为由静态方法绑定所来提高效率是不值得的。谈谈支持和反对Java的这个设计的理由。
14. Java的所有对象都由一个共同的祖先，这样做的理由是什么？

15. Java中的finalize子句的目的是什么？
16. 如果Java中既允许栈动态对象，又允许堆动态对象，会得到什么好处？又会有什么缺点？
17. 从程序设计方便性的角度，比较Ada 95和C++提供多态的方式。 553
18. 研究并解释为什么C#不包括非静态嵌套类。
19. 你能为一个抽象类定义实用变量吗？这样的变量有什么用？
20. 比较Java和Ruby中的实例变量的访问控制。
21. 比较Java和Ruby中的实例变量的类型错误检测。

程序设计练习题

1. 用Java改写12.5.2小节中的single linked list, stack 2和queue 2类，并在可读性和程序设计容易性方面与这些类的C++版本进行比较。
2. 用Ada 95重做第1题。
3. 用Ruby重做第1题。
4. 设计和实现这类A定义的程序，A有子类B，B有子类C。A类必须实现一个方法，该方法在B和C中被覆盖。编写一个测试类，以实例化A，B和C，以及包括3个对方法的调用。一个调用必须静态绑定到A方法，一个调用必须动态绑定到B方法，一个方法动态绑定到C方法。所有的方法调用必须使用指向类A的指针。 554

第13章 并 发

本章首先将描述子程序（或单位）层次和语句层次上的各种并发，也将简略地描述一些常见种类的多处理器计算机的体系结构。然后，我们将较详细地讨论程序单位层次上的并发。我们先描述一些在讨论单位层次上的并发之前所必须理解的基本概念，包括竞争同步和合作同步。接下来，我们将描述为并发提供语言支持的设计问题，然后我们将详细地讨论，语言中支持并发的三种主要的方法：信号量、管程和消息传递。我们也将给出程序示例。使用一个虚拟代码示例程序来示范信号量的使用方法；使用一个用并发Pascal所编写的示例程序来说明管程的使用；使用一个Ada的程序来作为消息传递的例子。Ada支持并发的一些语言特性也将被详细描述。这些特性包括任务、受保护的对象（在效果上就是管程）和异步消息传递。然后我们将讨论，Java和C#在程序单位层次上对于并发的支持。本章的最后一节讨论语句层次上的并发，包括高性能的Fortran对于并发的支持。

13.1 概述

软件执行时的并发表现为4个层次：指令层次（同时运行两条或更多的机器指令）、语句层次（同时运行两条或更多的语句）、单位层次（同时运行两个或更多的子程序单位）和程序层次（同时运行两个或更多的程序）。因为指令层次和程序层次的并发不涉及语言的问题，本章将不进行讨论。我们将讨论子程序层次上和语句层次上的并发，并将重点放在子程序层次上的并发。

程序单位上的并发执行，可以进行在一些物理上分开的处理器上，也可以在共享单处理器的情况下逻辑地进行。这似乎是一个简单的概念，但是它对于程序设计语言的设计人员，却是一个重大的挑战。

并发控制方法增加了程序设计的灵活性。它们本来被发明来解决操作系统方面的问题，但后来被用到多种其他程序设计应用中。例如，许多被设计来模拟实际的物理系统的软件系统，许多这些实际的物理系统，是由多个并行的子系统所组成的。对于这些应用，传统的子程序控制的受限形式是不适当的。

语句层次的并发与程序单位层次的并发有很大不同。从语言设计者的角度来看，语句层次的并发主要是说明数据应该如何分布到多个存储器中，以及哪些语句应该被并行地执行。

555
|
556

本章的意图，是要讨论并发与语言设计问题最相关的方面，而不是对于并发中的所有问题作出一项权威性的研究。那样的研究对于一本程序设计语言的书是不适合的。

13.1.1 多处理器体系结构

很多不同计算机体系结构有一个以上的处理器，并且能够支持某些形式的并发执行。在开始讨论各种程序和语句的并发执行之前，我们先简略地描述多处理器的体系结构。

第一批具有多处理器的计算机是一个适用的处理器，加上一个或多个仅用于输入和输出操作的处理器（外设处理器）。这样的体系结构允许那些在20世纪50年代后期出现的计算机，在执行一个程序的同时，并行地执行其他程序的输入或输出。因为这种并发不需要语言的支持，我们将不再在本书中进行考虑。

到20世纪60年代早期，就有了具有多个完整处理器的机器。这些处理器通过操作系统的作业调度程序来使用，操作系统的作业调度程序将批量作业队列中那些分开的作业分配到不同的处理器。具有这种结构的系统支持了程序层次的并发。

在20世纪60年代中期，出现了新的多处理器计算机。多处理器计算机的处理器是完全相同的。它们还可以分别执行一个指令流中的不同指令。例如，有些机器具有两个或多个浮点数的乘法器。还有些机器具有两个或多个完整的浮点数算术单元。需要由为这些机器所编写的编译器来决定哪些指令可以被并行地执行，以便对这些指令进行相应的调度。具有这种结构的系统支持指令层次的并发。

现在有许多不同类型的多处理器计算机。下面我们将描述其中最常用的两种。

在不同的数据上同时运行相同指令的多处理器计算机被称为“单指令多数据”（SIMD）体系结构的计算机。在一部SIMD结构的计算机中，每个处理器都有自己的局部存储器。由一个处理器来控制其他处理器的操作。因为除了控制器之外的所有处理器，都在同一时刻执行相同的指令，然而在软件中并不需要同步。也许最广泛使用的SIMD机器是一类被称为**矢量处理器**的计算机。这种计算机具有多组寄存器，来存储矢量操作的操作数，而相同的指令同时在这些操作数上运行。从这种体系结构获益最大的，是科学计算程序，科学计算通常就是多处理器计算机的设计目标。

具有独立地操作能力，并且这些操作能够被同步的多处理器计算机称为**多指令多数据**（MIMD）的计算机。在一部MIMD计算机中，每个处理器都执行自己的指令流。MIMD计算机具有两种不同的配置结构，即分布式系统与共享存储系统。在分布式的MIMD机器中，每一个处理器都有自己的存储器。一台分布式MIMD机器能够被装入一个盒子之中，或者是可以被分布在一个大的区域之中。共享存储的MIMD机器则显然必须提供一些同步方法来避免在储访问间的冲突。甚至分布式的MIMD机器也需要同步，以使同时操作于同一程序之上。MIMD计算机比SIMD计算机更加昂贵也更加通用；显然MIMD计算机都支持单位层次的并发。 557

13.1.2 并发的种类

存在着两种并发单位的控制。最普通的一种是假设存在着多个处理器可供使用，并且来自相同程序的几个程序单位同时地运行，这就是**物理并发**（physical concurrency）。一种稍微放宽的并发概念是允许程序员和应用软件假设：存在着多个处理器提供真实的并发；然而事实上，程序是在单处理器上被分时地执行。这种形式被称为**逻辑并发**（logical concurrency）。这种情形类似于多道程序设计的计算机系统向不同用户提供同时执行时，所给人的错觉。从程序人员以及语言设计人员的观点来看，逻辑并发与物理并发是相同的。将逻辑并发映射到执行操作的硬件上去将是语言实现人员的工作。逻辑和物理的并发都允许将并发的概念作为程序设计方法学来使用。在本章后面的几节里，我们将讨论应用物理并发和逻辑并发。

一种有用的可视化程序执行流程的方法是，设想在程序源文本的语句上放置一条线。到达某个特定执行点的所有语句都被表示该执行的线所覆盖。跟随经过源程序的线就可以追踪经过程序可执行版本的执行流程。当然，除了那些最简单的程序以外，几乎所有程序中的这种线都是非常复杂的。程序中的**控制线**（thread of control）就是控制经过程序时到达程序中之点的序列。

具有协同（参见第9章）的程序有时被称为准并发。这种程序有一条控制线，在物理并发执行的程序中有多条控制线。每一个处理器能够执行其中的一条控制线。虽然逻辑并发程序的执行实际上只有一条控制线，但是对于这样的程序进行设计和分析时，可以设想它们具有多条控制线。当多条控制线的程序运行于单处理器的机器上时，它的控制线被映射到一条线上。在这

558 种情况下，它就变成了虚拟多控制线的程序。

语句层次的并发是一个相对简单的概念，可以将包括数组元素操作语句的循环拆开，以便处理操作能够被分配到多个处理器上。例如，如果一个循环重复运行500次，这个循环包括了一条语句，一次操作500个数组元素中的一个，可以将这个循环拆开并分配到10个不同的处理器上同时处理，从而一个处理器将处理50个数组元素。

13.1.3 学习并发的动机

学习并发的第一个理由，是因为并发提供了将一些问题的程序解决办法概念化的一种方法。如同递归是解决某一类问题的自然方法一样，许多问题可以很自然地使用并发来解决。许多程序被编写来模拟实际实体和行为。在许多情况下，被模拟的系统包括了多于一个的实体，而且这些实体都同时在进行自己的活动；例如，在一个控制区域中正在飞行的飞机、一个通信网络中的中继站和一个制造设备中的不同机器。为了使用软件来准确地模拟这样的系统，软件就必须要能够处理并发。

讨论并发的第二个理由，是因为当前已经广泛地使用了多处理器的计算机，需要有能够有效利用这种硬件功能的软件。又为语句层次和单位层次并发的重要性，因而必须开发是供这两种并发的设施，并且必须将这种设施包括进当代程序设计语言。

13.2 子程序层次并发的介绍

在我们讨论对于并发的语言支持之前，我们需要介绍关于并发的一些基础概念，以及能够使并发变得十分有用的一些要求。然后，我们再讨论支持并发的语言的语言设计问题。

13.2.1 基本概念

任务（task）是一个可以与同一程序的其他单位一起被并发执行的程序单位（类似于子程序）。程序中的每一个任务提供一条控制线。有时也将任务称为进程（process）。

任务具有的三个特征能够用来与子程序相区别。首先，任务可以被隐式地启动，而子程序则必须被显式地调用。其次，当一个程序单位调用一个任务时，它在继续自己的工作之前并不需要等待任务执行的完成。最后，当完成一个任务的执行时，控制可能会也可能不会返回到启
559 动任务执行的程序单位。

可以将任务分为两种：重任务（heavyweight task）与轻任务（lightweight task）。简而言之，一个重任务在自己的地址空间中执行，而所有轻任务都在同一地址空间中执行。实现轻任务比重任务要容易。

任务可以通过共享非局部变量、消息传递或是参数，来与其他的任务进行通讯。如果一个任务不以任何方式与程序中的其他任务施行通讯，或者是它不影响任何其他任务的执行，我们就称这个任务是不相关的（disjoint）。因为多个任务常常一起工作来产生一些模拟或者是解决一些问题，因而它们是相关的，并且它们必须使用某种通讯的方式来同步彼此的执行或者共享数据，或者是这两种方式皆而有之。

同步（synchronization）是一种控制任务执行顺序的机制。当一些任务共享数据时，需要有两种类型的同步：即合作同步与竞争同步。当任务A在继续它的执行之前，如果它必须等待任务B完成某种特定活动，在任务A与任务B之间就需要合作同步（cooperation synchronization）。当这两个任务都需要不可能同时使用的某种资源时，这两个任务之间需要竞争同步（competition synchronization），特别是，如果任务A要访问共享数据单位x，而任务B正在处理x，

则任务A必须等待任务B完成对于x的处理，不论这是什么样的处理。因此在合作同步中，任务可能需要等待完成某一种处理，它们操作的正确性取决于这种处理；然而在竞争同步中，任务可能需要等待其他的任务完成在共享数据上正在进行的任何处理。

可以使用被称为“生产者-消费者问题”的例子来说明一个简单形式的合作同步。这个问题起源于操作系统的开发，其中一个程序单位产生某一种数据值或资源，而另外的一个程序单位将使用它。通常，所产生的数据值被生产的单位放在一个存储缓冲区，然后再被消费的单位从缓冲区中取走。这种在缓冲区里存放和取走的操作必须是同步的。如果缓冲区是空的，消费单位不被允许从缓冲区取走数据。如果缓冲区是满的，生产单位也不被允许把缓冲区存放数据。这就是被称为合作同步的问题，因为要想正确地使用缓冲区，共享数据的使用者们就必须合作。

竞争同步防止两个任务在同一时间同时存取一个共享的数据结构；这种情况可能会破坏共享数据的正确性。要提供竞争同步，就必须要保证共享数据的互斥访问。

为了澄清竞争问题，让我们来考虑下列情节：假设任务A必须把2加到共享整数变量 TOTAL 上，TOTAL的初值是3。此外，任务B要将TOTAL的值乘以2。这两个任务都分别进行下面3步的过程： 560

1）从TOTAL取得数值；

2）在数值上进行算术操作；

3）然后再将新的值放回TOTAL。

如果没有竞争同步，它们的操作可能会产生三个不同的值。如果在任务B开始之前，任务A已经完成了它的操作，TOTAL的值将会是8。我们假设这是正确的结果。但是如果A和B都是在另一方将它的新值放回原处之前取得的TOTAL的原值，结果将会是不正确的。如果A首先将它的新值放回原处，TOTAL的值将会是6，图13-1中显示了这种情形。如果B首先将它的新值放回原处，TOTAL的值将会是4。有时将导致这种问题的这种情形，称为竞争条件，因为两个或多个任务相互竞争使用它们共享的资源，而且这个问题的行为取决于哪一个任务首先获得了这个资源。现在，竞争同步的重要性应该是很清楚的了。

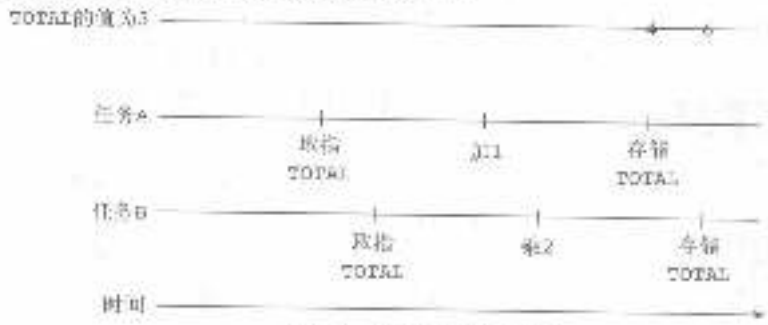


图13-1 竞争同步的举例

对于共享资源提供互斥访问的一般方法，是将资源看成任务可以拥有的某种物体，并且在任何时刻只允许一个任务来拥有它。为了得到一个共享资源，任务必须首先请求它。当一个任务使用完了它所拥有的共享资源时，就必须放弃对于资源的持有，以使其他的任务可以来使用。

提供对于共享资源互斥访问的三种方法是：将在13.3节中讨论的信号量，将在13.4节中讨论的管程和将在13.5节中讨论的消息传递。

同步机制必须能够延迟任务的执行。同步将一个执行次序强加到任务之上。这种次序由延

迟来实现，要理解在任务的生存期间发生了什么，我们必须考虑任务执行是怎样被控制的。不论一部机器具有单个还是多个处理器，总会存在任务多于处理器的可能性。**调度程序**（scheduler）管理任务间对于处理器的共享。如果从来就没有任何中断，并且所有任务都有相同的优先级，调度程序可以只给每个任务很短的一个时间片，例如 0.1 秒。当轮到一个任务执行时，调度程序让它在一个处理器上仅运行一段给定长度的时间。当然，会存在很多使情况复杂化的事件，例如，为了同步而延迟以及为了等待输入输出操作而延迟。任务可能处于几种不同的状态之中。这些状态是：

1）新生状态：一个任务刚被创建，但还没有开始执行时，这个任务处于新生状态。

2）可运行状态、或就绪状态：一个可运行任务，是准备好运行但当前还没有运行的任务。
561 或者是因为调度程序还没有给处理器以时间，或者它先前已经运行过了，但被下面第四段所描述的一个方式所阻止。将可运行任务存储于一个通常被称为**任务就绪队列**（task ready queue）的队中。

3）运行状态：一个运行的任务，是一个当前正在执行的任务，也就是说，这个任务具有一个处理器，并正在执行这个任务的代码。

4）阻塞状态：一个被阻塞的任务已经被运行，但任务的执行被某一事件所打断，最通常的是输入或输出操作的事件。因为输入和输出操作比程序的执行慢得多，一个任务开始了一个输入或输出操作后，当它在等候输入或输出操作完成时，它会被阻止使用处理器。除了这些类型的阻塞而外，一些语言还给用户程序提供一些操作，用以说明某一个任务应该被阻塞。

5）死亡状态：死亡的任务就不再活跃，在完成了一个任务的执行之后，或者是一个任务被程序显式地撤销之后，这个任务就成为死亡的任务。

任务执行中的一个重要的问题：是当前正在运行的任务已经变成阻塞状态时，或者是已经用完它的时间片时，怎样来选择一个就绪任务移入运行的状态？有几种可供选择的不同算法，其中的一些是基于可说明的优先级。选择算法是在调度程序中实现。

活性的概念，与任务的当前执行及共享资源的使用相关联。在顺序执行的程序的环境中，如果一个程序继续运行，并最终导致执行的最后完成，这个程序就具有**活性**（liveness）的特征。在更加通用的术语里，活性则意味着：如果一个事件（例如程序的完成）是应该发生的，它终将发生。这就是说，执行不断地推进着。在并发和使用共享资源的环境中，任务的活性可能会停止存在，这意味着程序不能够继续运行，并因此会永不终止。

历史注释

PL/I是第一种包括并发任务的程序设计语言。它允许并发地执行用户程序中的调用程序和被调用的子程序。然而，为这些并发执行而实现的同步机制，却并不充分。这种同步机制仅仅包括了被称为事件的二元信号量，以及任务完成执行时对于信号量进行检测的功能。

ALGOL 68允许复合语句层次的并发，并且包括了一个名为sema的信号量数据类型。

例如，假设任务A和任务B都需要共享的资源X和Y以完成它们的工作。进一步再假设，任务A已持有X，并且任务B已持有Y。经过一段执行之后，A需要资源Y以继续它的执行，
562 因此它请求Y，但必须等到B释放Y。同时，任务B请求X，但也必须等待A释放X。这个任务都不放弃它们所持有的资源，结果这两者都失去了活性，程序执行将永远不能正常地完成。这种类型的失去活性，被称为**死锁**（deadlock），死锁对于程序的可靠性是一种严重的威胁，因此在语言和程序设计中需要认真考虑避免死锁的发生。

我们现在已经准备进行关于提供并发单位控制的语言机制的讨论。

13.2.2 为并发而设计的语言

许多语言被设计来支持并发，从20世纪60年代中期的PL/I开始，包括当代语言Ada 95、Java、C# Python和Ruby。

13.2.3 设计问题

支持并发的语言的最重要的设计问题，已经在前面进行了讨论：竞争同步与合作同步。除此之外，还有一些次重要的设计问题，例如，怎样提供任务的调度，另外，怎样以及何时开始任务的执行和结束任务执行的问题。还有，怎样以及何时创建任务的问题。

请记住，我们不打算对并发进行全面的讨论，我们仅仅讨论那些与支持并发有关的最重要的设计问题。

在以下几节里，我们将讨论并发设计问题的三种答案：信号量、管程和消息传递。

13.3 信号量

信号量是一种能够用来提供任务同步的简单机制。我们将在下面的段落中描述信号量，并讨论如何能够将它们用于这个目的。 363

13.3.1 介绍

在对于共享数据结构的互斥访问提供竞争同步的工作中，Edsger Dijkstra 在 1965 年设计了信号量（Dijkstra，1968b）。信号量也能够被用来提供合作同步。

信号量（semaphore）是一种数据结构，它由一个整数和一个存储任务描述符的队所组成。**任务描述符**（task descriptor）也是一种数据结构，它存储有关任务执行状态的相关信息。信号量的概念是：为了提供对于一种数据结构的有限制的访问，而将守卫放置于存取这个数据结构的代码附近。守卫（guard）是一种语言装置，它只能允许被守卫的代码运行于某种被说明的条件之下。守卫能够被用来保证任何时候只有一个任务存取一个共享的数据结构。信号量是守卫的实现。守卫机制中的一种不可缺少的技术是要确保所有对被守卫的代码的访问企图，终将如愿以偿。这种技术将暂时不能够被允许的访问请求，存储在任务描述符的队之中。之后，将允许这些请求离开，并执行被守卫的代码。这就是之所以一个信号量是由一个计数器，和一个任务描述符的队所组成的原因。

提供给信号量的仅有的两种操作，最早由Dijkstra命名为P和V，它们分别来自于两个荷兰文词 passeren（通过）和 vrygeren（释放）（Andrews and Schneider，1983）。我们在下面的讨论中将分别称这两种操作为："等待"与"释放"。

下面小节将用最常见的应用来描述信号量提供守卫的过程。

13.3.2 合作同步

在本章的许多地方，我们将使用一个共享缓冲区的示例，来说明几种不同的提供合作同步和竞争同步的方法。为了进行合作同步，这个缓冲区必须具有一个方法，来记录缓冲区中空位的个数以及被填充位置的个数。信号量的计数器能够部分地用于这种目的。能够使用一个信号量变量emptyspots存储共享缓冲区中空位置的数目，而能够使用另外的一个信号量变量fullspots来存储缓冲区中被填充位置的数目。而使用这两个信号量的任务描述符队，来存储被信号量的延迟操作所阻塞的任务。

我们将缓冲区的例子设计为一种抽象数据类型：所有的数据都经过子程序DEPOSIT进入缓冲区，并经过子程序FETCH离开缓冲区。这样，子程序DEPOSIT只需要检测信号量emptyspots，就能够知道是否存在着空位置。如果存在着至少一个空位，子程序DEPOSIT就将继续执行。子
564 程序DEPOSIT必须包括，将emptyspots的计数器减值的功能。如果缓冲区已经占满了，DEPOSIT的调用程序就必须在emptyspots的任务描述符队之中等候，直到可以获得一个空位置。当子程序DEPOSIT完成操作时，它将增加fullspots信号量计数器的值，指示缓冲区中增加了一个被填充的位置。

子程序FETCH与子程序DEPOSIT具有着相反的执行序列。子程序FETCH检查fullspots信号量，以确定缓冲区中是否至少包含了一个项。如果是的话，这个项将被删除，并且信号量emptyspots的计数器的值将增加1。如果缓冲区为空的话，将调用进程放置于fullspots的任务描述符的队中等候，直到在缓冲区中出现一个项为止。当子程序FETCH执行完成时，它必须增加emptyspots的计数器的值。

对于信号量类型的操作通常不是直接的，而是通过等待与释放子程序来进行。因此，上面所描述的子程序DEPOSIT的部分操作实际上是通过调用等待与释放子程序来完成的。请注意，等待与释放子程序必须可以访问任务就绪队。

使用等待子程序来测试给定信号量变量的计数器。如果这个计数器的值大于零，调用程序就能够进行操作。在这种情况下，信号量变量计数器的值被减1以指示少了一项。如果这个计数器的值为零，就必须将调用程序放置于信号量变量的等候队之中，并且必须将处理器传递给其他的就绪任务。

一个任务将使用释放操作，来允许某个其他的任务得到一个被信号量变量的计数器所计数的事物。如果这个信号量变量的队为空就意味着没有任务在等候，释放操作则会增加计数器的值（指示多了一个可用的被控制的事物），如果有一个或多个任务在等候，释放操作将它们中的一个从信号量的等候队中移到就绪队之中。

下面是对于等待操作与释放操作的简略描述：

```
wait(aSemaphore)
if aSemaphore 的计数器 > 0 then
    aSemaphore 的计数器减1
else
    将调用程序放置于aSemaphore的队中
    尝试将控制传递给一个就绪任务
    (如果就绪任务队为空，死锁发生)
end if

release(aSemaphore)
if aSemaphore 至少为空(没有任务在等待) then
    aSemaphore 的计数器加1
else
    将调用任务放至于就绪任务队中
    将控制传给aSemaphore的队中的一个任务
565 end
```

我们现在可以给出一个示例程序。这个程序为一个共享缓冲区实现合作同步。在此例中，共享缓冲区存储整数值，并被构造成一个逻辑的环形结构。这个共享缓冲区被设计来支持多生产者与多消费者的任务。

下面的虚拟码显示了这种生产者任务与消费者任务的定义。使用了两个信号量来防止缓冲区的下溢与上溢，并因此提供合作同步。假设缓冲区的长度为BUFLEN，并且已经存在着实际

操纵缓冲区的子程序FETCH和DEPOSIT。对于信号量计数器的访问使用点来标记。例如，如果fullspots是一个信号量，它的计数器则由fullspots.count来引用。

```
semaphore fullspots, emptyspots;
fullspots.count = 0;
emptyspots.count = BUFLEN;
task producer;
  loop
  --产生VALUE--
  wait(emptyspots);      {等待一个空位}
  DEPOSIT(VALUE);
  release(fullspots);    {增加占缓冲的位置}
  end loop;
end producer;

task consumer;
  loop
  wait(fullspots);       {确保它不是为空}
  FETCH(VALUE);
  release(emptyspots);   {增加空单元位置}
  --消费VALUE--
  end loop
end consumer;
```

如果共享缓冲区当前为空，信号量fullspots将使得consumer任务进入队之中，以等待缓冲区的项。如果共享缓冲区当前已满，信号量emptyspots将使得producer任务进入队之中，以等待缓冲区的空位置。

13.3.3 竞争同步

我们上面的缓冲区的例子不提供竞争同步，对于这种结构的访问，可以使用一个附加的信号量来控制。这个附加的信号量并不需要进行任何计数，它仅仅使用计数器来指示缓冲区是否当前正在被使用。如果这个信号量的计数器为1，wait语句允许访问缓冲区；数值1表示共享缓冲区现在没有被访问。如果这个信号量的计数器为0，则表示缓冲区当前正被访问，此时将任务放入信号量的队之中。请注意，必须将信号量计数器的初值设为1，也即必须将信号量的队初始化为空。 366

如果一个信号量只需要一个二元计数器，如我们用来提供竞争同步的信号量那样，就称这个信号量为二元信号量（binary semaphore）。

下面的示例代码说明，怎样使用信号量为当前被访问的共享缓冲区提供竞争同步与合作同步。使用信号量access来确保对于缓冲区的互斥访问。再次请注意，可能存在着多生产者以及多消费者。

```
semaphore access, fullspots, emptyspots;
access.count = 1;
fullspots.count = 0;
emptyspots.count = BUFLEN;

task producer;
  loop
  --产生VALUE--
  wait(emptyspots);      {等待一个空位}
  wait(access);          {等待访问}
  DEPOSIT(VALUE);
```

```
    release(access);        {放弃访问}
    release(fullspots);     {增加占据了的位置}
    end loop;
  end producer;

  task consumer;
    loop
    wait(fullspots);        {确保它不是为空}
    wait(access);           {等待访问}
    FETCH(VALUE);
    release(access);        {放弃访问}
    release(emptyspots);    {增加为空的位置}
    -- 消费VALUE --
    end loop
  end consumer;
```

如果只是粗略地看一下这个例子，可能会相信它有问题。设想一个任务正等候consumer中的wait(access)的调用，而另外一个任务从共享缓冲区取走了最后的一个值。但幸运的是，这是不可能发生的。因为wait(fullspots)通过将fullspots的计数器减1，为调用它的任务在缓冲区中保留了一个值。

567

到目前为止，我们还没有讨论信号量的一个关键方面。回忆我们在前面对于竞争同步问题的描述：不能够重载共享数据上的操作。如果前面的操作仍然在进行，而后面的操作却已经开始，它们所共享的数据可能就会不正确。一个信号量本身就是一个共享数据对象，因而信号量的操作也会受到同样问题的影响。所以最根本的问题是，信号量的操作是不能够被中断的。许多计算机都具有为信号量操作而明确设计的非中断指令。如果没有这种指令的话，使用信号量来提供竞争同步将产生严重的问题，而且还没有简单的解决方法。

13.3.4 评估

使用信号量提供合作同步，产生非安全的程序设计环境。不存在静态的方式来检测信号量使用的正确性，这种使用的正确性完全依赖于信号量所出现的程序的语义。在缓冲区的示例中，如果producer任务不包括wait(emptyspots)语句，将会造成缓冲区的上溢。如果consumer任务不包括wait(fullspots)语句，则会造成缓冲区的下溢。如果遗漏任何释放操作，都会造成死锁。这些情形都是合作同步的失败。

使用信号量提供合作同步，产生可靠性的问题；使用信号量来提供竞争同步时，也会出现同样的问题。如果在任意一个任务中没有包括wait(access)语句的话，将引起对于缓冲区的非安全的访问。而如果在任意一个任务中没有包括release(access)语句，将会引起死锁。这些情形都是竞争同步的失败。Brinch Hansen注意到了使用信号量的危险性，他曾经写道："对于一位从不犯错误的理想程序员，信号量是一种优雅的同步工具"（Brinch Hansen，1973）。然而，这样的程序人员却极少有。

13.4 管程

解决并发环境中信号量问题的一种办法，是将共享数据结构与对于这些结构的操作封装在一起，并且将这些数据结构的表示藏匿起来；也就是说，将共享数据结构定义为抽象数据类型。这种解决方法能够将同步的责任转移到运行时系统，以便提供不具有信号量的竞争同步。

13.4.1 介绍

在制定数据抽象的概念时，参与这项工作的人员将相同的概念运用于并发程序设计环境的

共享数据，从而生产管程。根据Brinch Hansen（Brinch Hansen，1977，p. xvi），Edsger Dijkstra 568
在1971年曾经做出的提议：应该将共享数据上的所有同步操作包括进一个程序单位之中。Brinch Hansen在操作系统的环境中使得这项概念形式化（Brinch Hansen，1973）。第二年，Hoare将这种结构命名为管程（Hoare，1974）。

第一种提供管程的程序设计语言是并发Pascal（Brinch Hansen，1975），Modula（Wirth，1977），CSP/k（Holt et al.，1978）和Mesa（Mitchell et al.，1979）也提供了管程。在当代语言中，Ada，Java和C#支持管程；关于这些，我们将在本章以下部分进行讨论。

13.4.2 竞争同步

管程最重要的特性之一，是共享数据居于管程之中而不是居于任何客户单位之中。这样，程序人员就不需要通过使用信号量或者是其他的机制，来同步共享数据的互斥访问，因为所有的访问都发生在管程中，所以能够将管程实现为任何时刻只允许一个访问的结构，从而保证同步访问。当对于管程的调用繁忙时，会将一些调用隐性地放入队之中。

13.4.3 合作同步

虽然互斥访问之共享数据是通过管程的内部，然而一些进程之间的合作仍然是程序人员的责任。特别是，程序人员必须保证一个共享缓冲区不会出现下溢或上溢。不同的语言提供不同的方式，来进行合作同步的程序设计，所有的这些方式都与信号量有关。

图13-2中显示一个包含了四个进程和一个管程的程序，管程提供对于一个并发共享缓冲区的同步访问。此图中，管程的接口显示在插入和删除框中（为了插入和删除数据）。

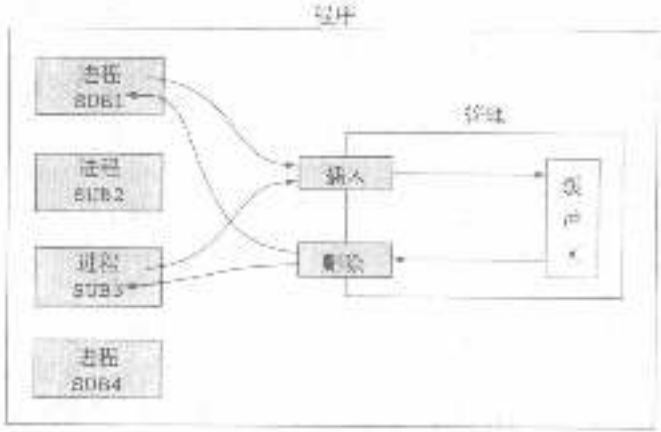


图13-2 一个使用管程来控制共享缓冲区访问的程序

13.4.4 评估

就提供竞争同步而言，管程是一种比信号量更好的方法，这主要是因为信号量方法中所存在的一些问题，如曾经在13.3节中讨论过的。在使用管程时，合作同步仍然是存在的问题；关于这一点，在下面讨论了Ada和Java的管程实现后，将会变得更加清晰。

在并发控制的表示方面，信号量和管程一样功能强大——可以使用信号量来实现管程，也可以使用管程来实现信号量。

Ada提供了两种方式来实现管程。Ada 83包括了一种可以用来实现管程的通用任务模型，

569 Ada 95则增加了一种更为简洁及高效的方式来构造管程，称为受保护的对象（protected object）。这两种方式都是以消息传递为基础模型来支持并发的。消息传递模型允许并发的程序单位为分布式的，然而管程却不允许分布式的并发的程序单位。关于消息传递将在下面的小节中讨论，关于Ada语言对于消息传递的支持，则将在13.6节中讨论。

13.5 消息传递

这一小节介绍消息传递的基本概念。13.6节将详细描述Ada对消息传递方法的支持。

13.5.1 介绍

最早尝试提供并发任务之间的消息传递能力的语言设计是Brinch Hansen（1978）和 Hoare（1978）的工作。消息传递的先驱开发人员还开发了一种技术，处理当一个给定任务同时收到来自其他任务的多个请求时，应该如何处置的问题。当时决定需要某种非确定性的方式，以便在多个请求中公平地选择一个首先执行。存在着关于这种公平性的各种不同的定义，然而在大体
570 上这种公平性的意义是：所有请求者都具有与给定任务通信的平等机会（假设每一个请求者都有同样的优先级别）。Dijkstra（1975）首创了用于语句层次上来控制的非确定性结构被称为守卫的命令（guarded command）。（守卫的命令曾经在第8章中进行过讨论。）守卫的命令是控制消息传递的设计的结构基础。

13.5.2 同步消息传递的概念

消息传递可以是同步的，也可以是异步的。在13.6.8小节中将描述 Ada 95 的异步消息传递，在这里，我们描述同步消息传递。同步消息传递的基本概念是：任务通常是很忙碌的，忙碌时不希望被其他的单位所中断。假如任务A和任务B都是在执行之中，而且任务A需要传递一个消息给任务B。显然，如果B很忙碌，它就不愿意让其他的任务来中断自己的执行。因为这将打乱B当前的工作。此外，消息的到来常常使得接收者去进行相关的处理，在其他处理还没有完成时，这通常是不明智的。一种替代方案是提供一种语言机制，以允许一个任务在它准备好接收消息时告诉其他的任务。这个方法像一个主管告诉他的秘书让打来的电话都进行等候，直到另外一个活动（也许是一个重要会议）结束以后。此时，主管告诉秘书他现在愿意接听那些在等候的电话。

一个任务应该能够在某一个时刻暂时停止它的执行，或者是因为它无事可干，或者是它需要来自另外一个单位的数据才能够继续执行。这正像一个在等待重要电话的人。在某些情况下，没有事情可做，只能坐等。在这种情形下，如果任务A需要发送一个消息给任务B，并且任务B愿意接收，消息就能够被传递过来，这种实际的传输被称为会合（rendezvous）。注意，会合只能够发生在发送者和接收者都希望它发生的时候。

如在下面的几节里所描述的，任务的合作同步与竞争同步都能够方便地使用消息传递的模型来进行处理。

13.6 Ada对并发的支持

这一小节描述Ada对于并发的支持。Ada 83 仅仅支持同步消息传递。在 Ada 95中增加了对于异步消息传递的支持。

13.6.1 基本概念

Ada中对于任务的设计部分地基于Brinch Hansen和Hoare的工作，其中的消息传递是这种设计的基础，并且使用非确定性在竞争传递消息的任务中来进行选择。 571

完整的Ada任务模型十分复杂，因而下面仅仅是对于这种任务类型的有限的讨论。我们在这里的重点将放在Ada版本中的同步消息传递机制上。

Ada任务可能比管程更加活跃。管程是被动的实体，它为存储的共享数据提供管理服务。然而，只有当服务被请求时管程才提供这些服务。当使用Ada的任务来管理共享数据时，可以认为这些任务是与所管理的资源在一起的管理者。任务具有几种管理机制，一些是确定性的，一些是非确定性的。这允许了任务在对于资源访问的竞争请求中来进行选择。

Ada任务的形式，与Ada中的包的形式相类似。任务也具有两个部分，即说明部分和体部分。两者都具有相同的名字。任务的接口就是它的入口点，这些接口即是能够接收来自其他任务的消息的位置。可以十分自然地将入口点列在任务的说明部分中。因为会合涉及数据的交换，消息可以具有参数；因此，任务的入口点也必须允许参数，这些参数被描述在说明部分。在外观上，任务的说明与包中的抽象数据类型的说明非常类似。

考虑下面代码中的Ada任务说明的示例，这个任务包括一个被命名为Entry_1的入口点，它具有一个输入型的参数：

```
task Task_Example is
  entry Entry_1(Item : in Integer);
end Task_Example;
```

任务体中必须包括入口点的某种语法形式，而入口点则必须与任务说明中的entry子句相对应。在Ada中，这些是由accept子句来说明，这种说明由保留字accept所引出，一个accept子句是一个语句系列，它起始于保留字accept、终止于匹配的保留字end。accept子句本身相对简单，但是这些子句所嵌入的其他结构就可能相当地复杂。简单的accept子句有着下面的形式：

```
accept 入口点名称(形参) do
  ...
end 入口点名称;
```

accept的名字与相关任务的说明部分中的一条entry子句中的名字相匹配。可选参数提供调用任务与被调用任务之间的数据通讯方式。在do与end之间的语句则定义了将在会合期间发生的操作，这些语句一起被称为accept子句体。在实际的会合期间，发送者任务被悬挂起来。 572

在任何时刻，当一个accept子句接收到一条由于某种原因而没有准备接收的消息时，发送者任务就必须被悬挂起来，直到接收者任务已经准备好了来接收这条消息时为止。当然，入口点也必须记住这些没有被接收的消息的发送者任务。为了这个目的，任务中的每一个accept子句都有一个与它相关联的队，这个队记录下不成功地试图与这个任务通讯的其他的任务。

下面是在前面给出了说明的任务的任务体框架：

```
task body Task_Example is
  begin
  loop
    accept Entry_1(Item : in Integer) do
      ...
    end Entry_1;
  end loop;
  end Task_Example;
```

这个任务体中的accept子句是任务说明中被命名为Entry_1的entry的实现。如果在任何其他的任务发送一条消息给Entry_1之前，Task_Example就开始了执行，并且达到了accept接受子句的Entry_1时，Task_Example将被悬挂起来。如果当Task_Example被悬挂在accept子句之时，另外的一个任务发送一条消息给Entry_1，这时将发生一个会合，并且将执行这一条accept子句的体。此后，因为体中的循环，执行将再一次达到accept子句。此时，如果没有其他的任务再发送消息给Entry_1，执行将再一次被暂停，以等待下一条消息的到来。

在这个简单的示例中，会合能够以两种基本的方式发生。第一种，接收者任务，Task_Example可能正在等待另外一个任务发送消息给入口Entry_1。当消息被发送时会合就发生。这就是上面所描述的情形。第二种，当另外一个任务试图发送一条消息给相同的入口时，接收者任务可能正忙于一个会合，或忙碌于与会合无关的其他的一些处理。在这种情况下，发送者就被悬挂起来，直到接收者可以在一个会合中接收那一条消息为止。如果接收者任务正忙于一个会合时，送来了几条消息，则发送者都被放入队之中进行等待。

图13-3中用时间线图形来说明这两种会合。



图13-3 与Task_Example会合的两种方式

一个任务并不一定需要有入口点，我们将没有入口点的任务称为**施动者任务**（actor task），因为这种类型的任务并不需要等待会合，然后才进行它们的工作。施动者任务能够通过给其他的任务发送消息来与其他的任务相会合。与施动者任务不同的另外一种任务，则可以具有accept子句，但是在accept子句的外面没有其他的代码，所以它仅仅是对于其他的任务作出反应。我们将这样的任务称为**服务者任务**（server task）。

一个给另外一个任务发送一条消息的Ada任务，必须知道接受消息的任务中入口点的名称。然而反过来则不成立：一个任务的入口点并不需要知道将发送消息给它的那些任务的名称。这种不对称性与CSP（Communicating Sequential Processes）（Hoare，1978）语言的设计正好相反。CSP也使用消息传递的并发模型。然而在CSP中，任务只接收来自被明确命名的任务的消息。这种方法的缺点是不能够建造通用的任务库。

573

图13-4显示描述一种常见的会合的图形式方法，在这个会合中，任务A发送一条消息给任务B。

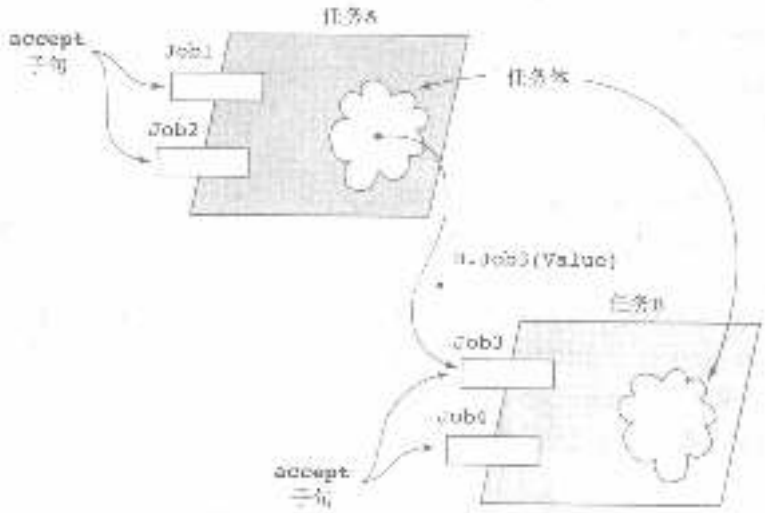


图13-4 任务A发送一条消息给任务B的会合的图形表示

Ada中的任务是具有类型的，并且可以是匿名的或者是命名的。一个具有命名类型的Ada任务，可以通过使用new操作符被动态地创建，并且可以通过使用指针来引用这个任务。例如，考虑下面的任务

```
task type Buffer is
  entry Deposit(Value : in Integer);
  entry Fetch(Value : out Integer);
end;
type Buf_Ptr is access Buffer;
...
Buf : Buf_Ptr;
Buf := new Buffer;
```

574

将任务声明于包、子程序或块中的声明部分，静态创建的任务与带有声明部分的代码语句同时开始执行。例如，一个在主程序中声明的任务与主程序的代码体中的第一条语句被同时地开始执行。使用new操作符创建的任务则被立刻开始执行。任务的终止是一个复杂的过程，我们将在本小节的稍后部分进行讨论。

任务可能具有任意多个入口，任务中相关的accept子句所出现的次序将决定接收消息的次序。如果一个任务具有多个入口点，而且这些入口可以按照任意的次序来接收消息，那么这个任务将使用一条select语句来包括这些入口。例如，假设把银行柜员活动的任务模型化，柜员必须为银行内的走入站台（walk-up station）的顾客服务，还要为免停窗口（drive-up window）的顾客服务。下面的柜员任务框架列出了一个select结构：

```
task body Teller is
  loop
    select
      accept Drive_Up(formal parameters) do
        ...
      end Drive_Up;
      ...
    or
```

575

```
        accept Walk_Up(formal parameters) do
          ...
        end Walk_Up;
        ...
      end select;
    end loop;
  end Teller;
```

这个任务中包括了两条accept子句，Walk_Up和Drive_Up，每一条accept子句都具有一个相关的队。当执行这个任务时，其中包含的select语句的行动是检查与这两条accept子句相关联的队。如果其中的一个队为空，但在另外的一个队中至少包含了一条在等候的消息（顾客），与这条在等候的消息相关联的accept子句就将与发送所收到的第一条消息的任务进行会合。如果这两个与accept子句相关的队都是为空的话，select语句将继续等候，直到入口之一被调用为止。如果与这两条accept子句相关联的队都是非空的话，两条accept子句中的一条将被非确定性地选择，来与它的一个发送者进行会合。循环将迫使select语句永远地重复执行。

accept子句的end语句标志着对于accept子句中的形参赋值的代码或引用的代码之结束。在accept子句与下一条or语句之间，或者在accept子句与下一条end select语句之间（当这一条accept子句在select中是最后一条accept子句时）如果存在代码段，这段代码就被称为扩展的accept子句。扩展的accept子句只有在相关的（它所紧跟的）accept子句被执行之后才能够被执行。扩展的accept子句之执行不是会合中的部分，因而扩展的accept子句能够与调用任务并行地执行。发送者在会合期间被悬挂起来，但是当执行到达accept子句的尾部时，发送者任务又被重新地启动（放回到就绪队之中）。如果一条accept子句没有形参，它就不需要do-end语句，可以仅仅由扩展的accept子句来组成accept子句。这样的accept子句就只能够用于同步。第13.6.3节将说明Buf_Task任务中的扩展accept子句。

13.6.2 合作同步

每一条accept子句都可以具有一个附属的守卫，以用于延迟会合。这种守卫的形式为一种when子句，例如：

```
when not Full(Buffer) =>
  accept Deposit(New_Value) do
```

一条与when子句连接在一起的accept子句，或者是开放的或者是关闭的。如果when子句 576
的布尔表达式为真，这时的accept子句就被称为开放的（open）；如果布尔表达式为假的话，这时的accept子句就被称为关闭的（closed）。一条没有守卫的accept子句总是开放的，可以将一条开放的accept子句用于会合；而一条关闭的accept子句则不能会合。

假设在一条select语句中包含了几条守卫的accept子句。通常是将这行select子句放置于无限循环之中。这种循环将使得select子句被重复执行，而在每一次的重复中都对每一条when子句进行求值。每一次重复都构造出一串开放accept子句。如果在这些开放的子句中，正好只有唯一的一条子句具有一个非空的队，就从这个队中取出一条消息来进行会合。如果在多条开放的子句中都具有非空的队，则非确定性地选择一个队，并从这个队中取出一条消息来进行会合。如果所有开放子句的队都是为空，任务就将等待下一条消息到达，然后再进行会合。如果已经执行了一条select语句，但所有的accept子句都是关闭的，就会产生一个运行时的异常或错误。如果总是使得when子句为真，或者总在select语句中增加一条else子句，就能

够避免出现异常的可能性。else子句能够包括除了accept子句以外的任何语句序列。

select子句也可以只有一条特殊的语句terminate语句，只有当terminate语句为开放的，并且没有其他的accept子句为开放的情况下，才能够选择terminate语句。而当选择了terminate子句时，就意味着任务已经完成了工作，但还没有被终止。我们将在本节稍后部分讨论任务的终结。

13.6.3 竞争同步

到目前为止，我们描述了提供合作同步以及任务之间通讯的语言特性。下面，我们将讨论在Ada中如何来保证对于共享数据结构的互斥访问。

如果需要由一个任务来控制存取一个数据结构，那么通过在任务中声明这个数据结构，就能够获得互斥访问。因为在任一给定的时刻，任务中只能够存在一条活跃的accept子句，因而，任务执行的语义通常就保证了对于结构的互斥访问。这种情形的唯一的一种例外，发生于当任务被嵌套于过程之中，或者是被嵌套于其他的任务之中时。例如，如果有一个定义了共享数据结构的任务，这个任务中又嵌套了另外一个任务，被嵌套的这个任务也可以访问共享数据结构，这将破坏数据的完整性。因此，控制对于共享数据访问的任务中，就不应该再定义其他的任务。

下面的例子，是给缓冲区提供管理的一个Ada任务。这个缓冲区与我们在13.3节中讨论的缓冲区的例子非常相似，在那个缓冲区中是使用信号量来控制同步。

```
task Buf_Task is
  entry Deposit(Item : in Integer);
  entry Fetch(Item : out Integer);
end Buf_Task;

task body Buf_Task is
  Bufsize : constant Integer := 100;
  Buf     : array (1..Bufsize) of Integer;
  Filled : Integer range 0..Bufsize := 0;
  Next_In,
  Next_Out : Integer range 1..Bufsize := 1;
  begin
  loop
    select
      when Filled < Bufsize =>
        accept Deposit(Item : in Integer) do
          Buf(Next_In) := Item;
        end Deposit;
        Next_In := (Next_In mod Bufsize) + 1;
        Filled := Filled + 1;
    or
      when Filled > 0 =>
        accept Fetch(Item : out Integer) do
          Item := Buf(Next_Out);
        end Fetch;
        Next_Out := (Next_Out mod Bufsize) + 1;
        Filled := Filled - 1;
      end select;
    end loop;
end Buf_Task;
```

这个例子中的两条accept子句都是扩展的。这两条扩展的accept子句可以与调用它们的

任务并发地执行。

使用Buf_Task的生产者任务和消费者任务，可以具有下面的形式：

```
task Producer;
task Consumer;
task body Producer is
  New_Value : Integer;
  begin
    loop
      -- 产生 New_Value --
      Buf_Task.Deposit(New_Value);
    end loop;
  end Producer;

task body Consumer is
  Stored_Value : Integer;
  begin
    loop
      Buf_Task.Fetch(Stored_Value);
      -- 消费 Stored_Value --
    end loop;
  end Consumer;
```

578

13.6.4 任务终结

我们现在来讨论任务终结的问题。首先必须要定义任务的完成。如果控制达到任务体代码的结尾，任务就完成了。当提出了异常然而没有相应的异常处理程序时，也称为任务完成了。（关于Ada中的异常处理将在第14章描述。）如果一个任务没有创建任何其他的被称为依靠者的任务，当这个任务的执行完成时任务即终结。如果一个任务产生了其他的依靠者任务，当这个任务的代码执行完成时，并且当它的所有依靠者任务都终结时，这个任务才终结。一个任务可以通过在一条开放的terminate子句等候结束它的执行。在这种情况下，只有当一个任务的主人（创建它的块、子程序或任务）已经完成，并且所有依靠于这个主人的任务或者已经完成，或者正在等候一个开放的terminate子句时，这个任务才终结。在这种情况下，所有的这些任务同时终结。一个块或一个子程序，需要等到它的所有依靠者任务都终结时，才能够退出。

13.6.5 优先级

可以给命名的类型以及匿名的类型赋予优先级。这是通过pragma语句来完成：

```
pragma Priority (表达式);
```

这里表达式的值说明任务的相对优先级，或者是任务所属类型定义的相对优先级。优先级的可能值范围取决于具体的实现。最高优先级可以用priority类型的最后一个属性来说明：priority类型被定义于System中（System是一个预定义的包）。例如，下面的表达式说明任何实现中的最高优先级：

579

```
pragma Priority (System.Priority'Last);
```

当目前执行的任务被阻塞，或用完分配时间，或完成执行而让予其他任务优先级时，决定从准备好的队列中选择任务的任务表使用这些。而且，如比当前正在执行的任务优先级更高的

任务进入准备好的任务队列，那么它抢占正在执行的低优先级任务而执行自己（如果它刚才已经在执行则重新启动执行）。被抢占的任务离开处理器并将其放置到准备好的任务列表中。

13.6.6 二元信号量

如果要控制对于一种数据结构的访问，并且这个数据结构没有被封装在一个任务中，那么就必须使用其他的方法来提供互斥访问。一种方法，是建立一个二元信号量任务，将这个任务与要引用该数据结构的任务一起来使用。这样的一个二元信号量任务可以被定义如下：

这个任务的目的是要保证Wait与Release操作以交替的方式来进行。

```
task Binary_Semaphore is
  entry Wait;
  entry Release;
end Binary_Semaphore;

task body Binary_Semaphore is
  begin
  loop
    accept Wait;
    accept Release;
  end loop;
  end Binary_Semaphore;
```

当仅仅是为了同步的目的来传递Ada中的消息，而并非是传递数据时。这种Binary_Semaphore任务显示了简化的可能性特别需要注意的是，这种不需要体的accept子句的简单形式。

使用Binary_Semaphore任务来提供对一个共享数据的互斥访问，可以如我们曾经在13.3节中的例子程序中所使用的信号量一样来进行。当然，信号量的这种用法也具有曾经在本章中讨论过的所有潜在问题。

像信号量的情形一样，也可以使用Ada中的任务功能来模拟管程，任务完全能够象管程一样来提供隐式的互斥访问。因而，Ada的任务模型同时支持信号量和管程。

13.6.7 受保护的对象

正如我们已经看到的，通过将共享数据包含于任务中，并且只允许通过任务入口的访问来控制对于共享数据的访问，任务的入口隐式地提供竞争同步。然而，使用这种方法的一个的问题是，它很难实现高效率的会合机制。Ada 95中的受保护对象是另外一种提供竞争同步的可能的方法，而且这种方法并不需要涉及会合。

受保护的对象不是任务，它更像在13.4小节中描述的管程。受保护的对象能够通过受保护的子程序或任务中语句构造上与accept子句相似的入口来进行访问。[1]受保护对象的入口与任务的入口相类似。受保护的子程序可以是保护的过程或受保护的函数，前者给受保护对象的数据提供互斥的读写访问，而后者给这些数据提供并发的只读访问。在受保护的过程体中，包含了受保护的程序单位的当前实例，被定义为变量，而在受保护的函数体中，包含受保护的程序单位的当前实例则被定义为常量。这种常量允许并发的只读访问。

当一个或多个任务使用同一个受保护的对象时，专门为这个受保护对象的入口调用提供了同步通讯。这种入口调用所提供的访问，类似于对于包含在任务中的数据的访问。

580

[1] 受保护对象的入口使用保留字entry，而不是在任务体中使用的accept。

在前一节中使用任务来解决的缓冲区问题，也可以通过使用受保护的对象来解决，后一种方法还更为简便。注意，这个例子不包括受保护的子程序。

```
protected Buffer is
  entry Deposit(Item : in Integer);
  entry Fetch(Item : out Integer);
private
  Bufsize : constant Integer := 100;
  Buf     : array (1..Bufsize) of Integer;
  Filled : Integer range 0..Bufsize := 0;
  Next_In,
  Next_Out : Integer range 1..Bufsize := 1;
  end Buffer;

protected body Buffer is
  entry Deposit(Item : in Integer) when Filled < Bufsize is
    begin
    Buf(Next_In) := Item;
    Next_In := (Next_In mod Bufsize) + 1;
    Filled := Filled + 1;
    end Deposit;
  entry Fetch(Item : out Integer) when Filled > 0 is
    begin
    Item := Buf(Next_Out);
    Next_Out := (Next_Out mod Bufsize) + 1;
    Filled := Filled - 1;
    end Fetch;
  end Buffer;
```

581

13.6.8 异步消息传递

到目前为止，在这一节中所描述的会合机制都是严格同步的，并且在实际通过会合来进行通讯前，发送者和接收者都必须就绪。

一个任务可以具有一条特殊的select子句，被称为异步select子句。这种子句能够对来自其他任务的消息立刻作出反应。它可以具有两种不同触发机制中的一种：入口调用或是一条delay语句。在异步select子句中，除了触发部分外，还具有可中止的部分；可中止的部分可以包含任意的Ada语句序列。异步select子句的语义是只能够执行这种子句中两个部分之一。如果触发事件发生（或者是接收到一个entry调用，或者delay定时器终止），将执行触发部分；否则，则执行可中止部分。下面异步select子句的两个示例，来自Ada 95参考手册（ARM，1995）。在第一段代码中，可中止子句被重复执行（因为是在循环中），直至接收到一个调用，Terminal.Wait_For_Interrupt。在第二段代码中，可中止子句所调用的函数至少运行五秒钟。如果到时候它还没有结束，select子句将退出。

```
-- 用于命令解释器的主命令循环
loop
  select
    Terminal.Wait_For_Interrupt;
    Put_Line("Interrupted");
  then abort
    -- 一旦终止介入这部分将被删除
    Put_Line("-> ");
    Get_Line(Command, Last);
    Process_Command(Command (1..Last));
```

```
    end select;
  end loop;

  -- 时间表格的计算
  select
    delay 5.0;
    Put_Line("Calculation does not converge");
  then abort
    -- 计算应该在五秒钟内完成，如果五秒钟还没有完成计算，即报设发散
    Horribly_Complicated_Recursive_Function(X, Y);
  end select;
```

13.6.9 评估

使用一般消息传递开发模型来构造管理，就像使用Ada的程序包来支持抽象数据一样——这两者较之所要解决的特殊问题，都是通用得多的工具。受保护的对象是提供同步共享数据的一种优良方法。

582

在不具有独立存储器的分布式处理器的情况下，如果要在管程和消息传递之间进行选择，以便挑选出在并发环境中实现共享数据的一种方式，就仅仅是根据人们各自的喜好。然而，在Ada中使用受保护的对象来支持对共享数据的访问则显然是一种更佳的方式。这种方法不仅在于代码简单，它的效率也要高很多。

对于分布式系统，消息传递是一种较好的开发模型，因为它自然地支持在分离处理器上并行运行分离程序的概念。

13.7 Java线程

Java中的并发单位是一种被称为run的方法。run方法的代码可以与其他对象的相同方法以及main方法被并发地执行。执行run方法的进程被称为线程（thread）。Java的线程是轻任务；这意味着这些线程都运行于同一个地址空间中。这就与Ada中的任务不同：Ada的任务是重任务，它们都分别运行于自己的地址空间之中。一个十分重要的结果就是，Java的线程比Ada中的任务所需要的代价小得多。

存在着可以被用来定义具有run方法的类对象的两种方式。其中的一种方式是定义预定义类Thread的一个子类，并重新编写这个子类的run方法。然而，如果这个新的子类具有一个必定的自然父类，那么将它定义为Thread的子类显然是不行的。在这种情况下，我们将定义一个类来继承它的自然父类，并且实现接口Runnable。接口Runnable提供对于run方法协议的支持。正如我们将在13.7.4小节中看到的，使用这种方式仍然需要一个Thread对象。

正如我们将在13.7.3小节中讨论的，能够使用Java的线程来实现管程。

13.7.1 Thread类

Thread类不是任何其他类的自然父类。Thread类对它的子类提供某些服务，但却没有任何自然的方式与这些子类的计算目的相关联。然而，Thread类是程序人员唯一可以用来构造并发Java程序的类。正如在前面曾经提到的，在13.7.4小节里，我们将简要地讨论关于Runnable接口的使用。

Thread类的核心是被称为run和start的两个方法。Run方法总是被Thread类的子类所覆盖。Run方法中的代码描述线程的行为。Thread类的start方法通过调用它的run方法将它

583 的线程启动为一个并发单位。[1]start方法的调用不同于通常的一般调用，因为控制即刻就返回到了调用程序，调用程序将接着继续它的执行，与新启动的run方法的执行相并行。

下面是Thread一个子类的框架和一段代码，这段代码产生这个子类的一个对象并在这个新的线程中开始run方法的执行：

```
class MyThread extends Thread {
  public void run() { ... }
}
...
Thread myTh = new MyThread();
myTh.start();
```

当开始执行一个Java应用程序（而不是一个小应用程序applet）时，将创建一条新的线程（main方法将被执行于这一条新线程中），并且调用main方法。小应用也都运行于它们的线程之中。因此，所有的Java程序都被执行于线程之中。

当一个程序具有多条线程时，必须有在一个调度程序来决定哪一条或是哪一些线程将在任何给定时刻运行。在大多数情况下只有一个处理器，因而在某一个时刻只有一条线程在实际运行。因为不同的实现（升阳，微软等）不一定会以完全相同的方式来调度线程，所以很难精确地描述Java的调度程序是如何工作的。然而典型地，只要所有的线程都有相同的优先级，调度程序将按照轮流的方式，给予每一个可运行线程同样长的时间片段。

Thread类提供了一些方法来控制Thread对象的执行。不具有参数的Yield方法，是正在运行的线程自动放弃处理器的请求。这一条线程被立刻放置于任务就绪队之中，成为可运行线程。然后，调度程序将从任务就绪队中选择出最高优先级的线程。如果其他可运行线程的优先级，都比刚才放弃处理器的线程低，这一条线程就成为下一个得到处理器的线程。

sleep方法只具有一个整数参数，这个参数是sleep方法的调用程序希望阻塞线程运行的毫秒数。在所说明的毫秒时间后，线程将被放置于任务就绪队中。因为无法确切知道究竟一个线程在被执行以前将在任务就绪队中等待多久，sleep的参数页面就是线程执行前的最少等待时间。sleep方法能够抛出InterruptedException异常，必须在调用sleep的方法中进行处理这个异常。关于异常将在第14章详细描述。

使用join方法来强迫延迟一个方法的执行，直到另外一条线程的run方法执行完毕为止。当一个方法的执行必须等待另外一条线程的工作完毕后才能继续时，就可以使用join方法来延
584 迟前面这个方法的执行。例如，我们可以有下面的run方法：

```
public void run() {
  ...
  Thread myTh = new Thread();
  myTh.start();
  // 进行这一条线程的一些的计算
  myTh.join();  // 等待myTh线完成
  // 进行这一条线程的剩余部分的计算
}
```

join方法将调用它的线程设置在被阻塞的状态，只有当join方法所属的线程完成运行时，这种状态才能够终结。如果join方法所属的线程恰好也被阻塞住了，就有产生死锁的可能。为了避免死锁的发生，可以使用一个参数来调用join方法。这个参数是一种毫秒时间限制，它说

[1] 直接地调用run方法并非总是可行的，其中的原因，是run方法时常所需要先初始化，被包含于start方法之中。

明调用线程可以等待多久，以便被调用的线程能够完成。例如：

```
myTh.join(2000);
```

这就使得调用线程可以等待2秒钟，以便myTh得以完成。如果时间已经过了2秒钟，但myTh还没有完成执行，调用线程将被放回就绪队，这意味着一旦被调度，这一条线程将被继续执行下去。

Java早期的版本曾经包括了另外的二种Thread方法，即stop，suspend和resume方法。因为安全方面的问题，Java反对使用这二种方法。stop方法时常被一种简单方法所覆盖，这种简单方法设置它的引用变量为null，从而撤销线程。

run方法完成执行的一种正常方式，是达到代码的结束。然而，在许多情况下，线程在运行时会被要求停止。这就存在着一个问题，即线程怎样决定究竟是应该继续执行，还是应该停止执行。interrupt方法就是一种与停止的线程进行通讯的方式。这种方法并不直接停止线程的执行，它仅仅是传送一条消息。这条消息将设定线程中的一个字位，而线程将对于这个字位进行检测。用于这种检测的是谓词方法isInterrupted。这并不是一种完善的解决方法。原因是当调用interrupt方法时，将要被中断的线程可能正处于睡眠或等待的状态。这样它就不会去检测自己是否已经被中断了。为了对付这些情况，interrupt方法将抛出一个InterruptedException异常来将线程（从睡眠或等待的状态中）唤醒。这样，线程就可以周期地检测自己是否已经被中断了和如果被中断了，它是否可以终止。由于当中断发生时，如果线程是处于睡眠或等待的状态，它会被中断所唤醒，因而线程不会错过中断。实际上，关于interrupt方法的行动及其使用还存在着很多的细节，但我们将不在这里进行讨论（Arnold, *et al.*, 2006）。 [585]

13.7.2 优先级

线程的优先级不必都是相同的。一个线程的默认优先级，与创建它的线程的优先级相同，如果 main产生了一条线程，这条线程的默认优先级是常量NORM_PRIORITY，其值通常是5。Thread还定义了两个其他的优先级常量，MAX_PRIORITY和MIN_PRIORITY，其值通常分别为10和1。[⊖]可以使用setPriority方法来改变线程的优先级，新的优先级可能是任何一个预定义的常量，或者是MIN_PRIORITY和MAX_PRIORITY之间的任何其他的数字。getPriority方法返回线程的当前优先级。

当一些线程具有不同的优先级时，调度程序的行为由这些优先级来控制。当正在执行的线程被阻止或删除，或者是时间片段被使用完毕时，调度程序将从任务就绪队中选择具有最高优先级的线程。一旦有了运行机会，只有当较高优先级的线程不在任务就绪队中时，较低优先级的线程才能够被执行。

13.7.3 竞争同步

在Java中实现竞争同步的办法，是说明访问共享数据的一个方法之运行，完成于另外一个相同对象上的方法的执行之前。换言之，我们能够说明一旦开始执行一个特定的方法，在任何其他的方法开始在相同对象上的运行之前，这个方法将完成它的执行。这种方法相当于在对象上放置了一把锁，从而阻止其他同步的方法在同一个对象上的执行。可以通过在方法的定义中

⊖ 不同系统具有不同的优先级的数目，所以在一些实现上的优先级数目可能会小于或大于缺省值10。

附加synchronized修饰符来说明，如下面类定义的框架中的：

```
class ManageBuf {
  private int [100] buf;
  ...
  public synchronized void deposit(int item) { ... }
  public synchronized int fetch() { ... }
  ...
}
```

定义在ManageBuf中的两个方法都被定义为synchronized的，这就能够阻止这两个方法
586 在不同对象上执行时的相互干扰，即使它们是由不同的线所调用的。

如果一个对象的所有方法都是同步的，它实际上就是一个管程。请注意，一个对象可能具有一个或多个同步的方法，也可能会具有一个或多个非同步的方法。

有时，在一个方法中对于共享数据进行处理的语句数目，大大少于其他的语句数目。这时，最好是仅仅对于存取共享数据的代码段或改变共享数据的代码段施行同步，而不是对于整个方法施行同步。这可以通过使用所谓的同步语句（synchronized statement）来说明，同步语句的一般形式为：

```
synchronized (表达式)
  语句
```

这里的表达式是对于一个对象的求值，而语句则可以是一条语句，也可以是一组复合语句。在这一条语句或复合语句的执行期间，将这个对象锁住。从而这条语句或复合语句的执行，就完全像在同步的方法体中的执行一样。

一个具有同步方法的对象必须有一个相关的队。当一个方法对于这个对象进行操作时，其他试图对于这个对象进行操作的方法，则被保存在这个队中。当一个同步方法完成了一个对象上的执行，如果这个队不为空，在等候队中的一个方法将被放入任务就绪队中去。

13.7.4 合作同步

Java中的合作同步，由Object类中定义的wait，notify和notifyAll方法来完成。Object类是所有Java中的类的根类。除了Object以外的所有的类，都继承这些方法。每一个对象都有一个等待的队列，其中包含调用了这个对象中的wait方法的所有的线。使用notify方法来告诉一条正在等待的线，它所等待的事件已经发生了。究竟哪一条线是由notify方法来唤醒，是不能确定的，因为Java的虚拟机器（JVM）从等待的列中挑选出一条线往往是随机的。正是由于这种原因，还由于在等待的线都是各自等待着不同的条件，所以我们经常是使用notifyAll方法，而不是notify方法。notifyAll方法唤醒这个对象的等待列中所有的线，在它们调用了wait方法后即开始执行。

wait，notify和notifyAll方法都只能从同步的方法中来调用，因为这些方法使用了由这种同步方法放置在对象上的锁。而对于wait的调用通常是放置在while循环之中，循环的控制由方法所等待的条件来实施。由于是使用notifyAll方法，其中有一些线可能自上一次测试
587 后已经将条件改为了false。

wait方法能够抛出InterruptedException，这种异常是Exception的一个后裔（关于Java中的异常处理，我们将在第14章中讨论）。因此，任何调用wait的代码都必须捕捉InterruptedException异常。假设我们所等待的条件是theCondition，传统的使用wait的方式就是：

```
try {
  while (!theCondition)
    wait();
  -- 在条件theCondition为真之后，进行所需要的任何事情
}
catch(InterruptedException myProblem) { ... }
```

下面的程序实现存储int值的一个环形队。这个程序说明如何来进行合作同步与竞争同步。

```
// Queue
// This class implements a circular queue for storing int
// values. It includes a constructor for allocating and
// initializing the queue to a specified size. It has
// synchronized methods for inserting values into and
// removing values from the queue.

class Queue {
  private int [] que;
  private int nextIn,
              nextOut,
              filled,
              queSize;

  public Queue(int size) {
    que = new int [size];
    filled = 0;
    nextIn = 1;
    nextOut = 1;
    queSize = size;
  }  //** end of Queue constructor

  public synchronized void deposit (int item) {
    try {
      while (filled == queSize)
        wait();
      que [nextIn] = item;
      nextIn = (nextIn % queSize) + 1;
      filled++;
      notifyAll();
    }  //** end of try clause
    catch(InterruptedException e) {}
  }  //** end of deposit method

  public synchronized int fetch() {
    int item = 0;
    try {
      while (filled == 0)
        wait();
      item = que [nextOut];
      nextOut = (nextOut % queSize) + 1;
      filled--;
      notifyAll();
    }  //** end of try clause
    catch(InterruptedException e) {}
    return item;
  }  //** end of fetch method
}  //** end of Queue class
```

588

请注意，这里的异常处理程序（catch）并不做任何事情。

可以将能够使用Queue类的生产者和消费者对象的类定义如下：

```
class Producer extends Thread {
  private Queue buffer;
  public Producer(Queue que) {
    buffer = que;
  }
  public void run() {
    int new_item;
    while (true) {
      //-- 产生一个 new_item
      buffer.deposit(new_item);
    }
  }
}

class Consumer extends Thread {
  private Queue buffer;
  public Consumer(Queue que) {
    buffer = que;
  }
  public void run() {
    int stored_item;
    while (true) {
      buffer.fetch(stored_item);
      //-- 消费 stored_item
    }
  }
}
```

589

下面的代码创建一个Queue对象，还产生一个Producer对象和一个Consumer对象，Producer对象与Consumer对象都依附于Queue对象；这段代码同时也开始执行Producer与Consumer对象。

```
Queue buff1 = new Queue(100);
Producer producer1 = new Producer(buff1);
Consumer consumer1 = new Consumer(buff1);
producer1.start();
consumer1.start();
```

我们可以将Producer和Consumer中的一个或是这两个对象，都定义成为Runnable接口的实现，而不是定义为Thread的子类。这里唯一改变了的是第一行，现在这一行代码将成为：

```
class Producer implements Runnable {
```

为了创建并且运行这个类对象，仍然需要创建一个连接到这个对象的Thread对象。这在下面的代码中给予了示范：

```
Producer producer1 = new Producer(buff1);
Thread producerThread = new Thread(producer1);
producerThread.start();
```

13.7.5 评估

Java对于并发的支持相对简单，然而卓有成效。因为Ada中的任务是单线程，从而容易将Ada的任务分布到不同的处理器上，尤其是在不同地点的不同计算机中，具有不同存储器的不

同处理器的情形。不可能将Java中的轻线程用于这种类型的系统。

13.8 C#线程

虽然C#中的线程是松散地基于Java的线程，但二者之间却有着重大的不同。下面是关于C#中的线程的简略概貌。

13.8.1 基本线程操作

任何C#的方法都可以被执行于自己的线程中。这不像在Java中，只有run方法可以执行于自己的线程中，一个C#中线程是通过产生一个Thread对象从而产生的。在产生Thread对象时，必须给Thread构造器传送一个预定义的代表类ThreadStart[1]的实例；而在产生ThreadStart类的实例时，还必须给Thread构造器传送实现线程的行动。例如，我们可以有： 590

```
public void MyRun1() {...}
...
Thread myThread = new Thread(new ThreadStart(MyRun1));
```

如同在Java中的那样，在产生了一条线程时，并不开始这条线程的并发执行。必须通过一个方法来请求执行；在这里的情形就是通过Start方法。例如：

```
myThread.Start();
```

也如同在Java中的那样，可以让一条线程等待另外的一条线程完成了执行后，再继续这条线程的执行，这也是通过使用一个具有类似名称的方法Join。

也可以使用Sleep方法将一条线程悬挂一定的时间。Sleep方法是Thread类中的一个公有的静态方法。Sleep方法的参数是为整数的毫秒数。与在Java中的Sleep方法不同的是，C#中的Sleep方法并不抛出任何异常，因而不需要将它在try程序块中调用。

还可以使用Abort方法来终止一条线程。然而在实际上，Abort方法并不将一条线程处死，它仅仅抛出线程可以捕捉的ThreadAbortException异常。当一条线捕捉到了这个异常后，这条线程通常将解除自身被分配了的空间，然后（到代码的结尾处）终结。

13.8.2 线程同步

存在着三种不同的方式用来同步C#中的线程：Interlock类、lock语句和Monitor类。每一种机制都设计来满足不同的需要。Interlock类使用于当需要进行同步的操作仅仅是整数的增减时。使用Interlock类的两个方法Increment和Decrement来进行这些操作。这两种方法将接受一个整数引用来作为参数。例如，如果要在一条线程中增加共享整数counter的值，我们可以使用：

```
Interlocked.Increment(ref counter);
```

lock语句被用来在线程的代码中标记一个关键段，lock语句的文法如下所示：

```
lock (表达式) {
   // 关键段
}
```

591

这里的表达式看起来像是lock的参数，但实际上它通常是对于线程所属对象的引用，即this。

[1] 一个C#中的代表(delegate)，是一个函数指针的面向对象的版本，在这里的情形，这个函数指针实际上指向了我们想要在新线程中执行的方法。

Monitor类具有着4个方法：Enter、Wait、Pulse和Exit。这些方法可以用来提供更为复杂的线程同步。Enter方法将接受一个对象的引用来作为参数，它标志这个对象的线程同步的开始。Wait方法将一条线程的执行悬挂起来，并且指示.NET的Common Language Runtime（CLR），希望这一条线程能够在下次有机会时恢复执行。Pulse方法也接受一个对象的引用来作为参数，它通知那些在等待的线程它们现在已经有机会再次运行。Pulse方法类似于Java中的notifyAll。等待线程的运行的顺序，是按照这些线程调用Wait方法的次序来进行。Exit方法将结束线程的关键段。

13.8.3 评估

C#中的线程，是从先驱语言Java中的线程改进而来。任何方法都可以执行于自己的线程之中。C#中线程的终结较为简洁（调用方法Abort比把线程指针设置为空更优雅）。而在C#中线程执行的同步则更为复杂，因为C#有一些不同的机制来满足不同的应用。与Java中的线程一样，C#中的线程是轻线程，因而它们不能够像Ada中的任务那么通用。

13.9 语句层次的并发

在这一节里，我们将简略描述支持语句层次并发的语言设计，从语言设计的角度来看，这种设计的目的是提供一种机制，程序人员能够使用这种机制告诉编译器，怎样将程序映射到多处理器体系结构上。[5]

在这一节中，我们将仅仅讨论一组支持语句层次并发的语言结构。此外，我们将描述这些结构在SIMD体系结构的机器中的用途（参见13.1.1小节），虽然这些语言结构是被设计来用于各种不同的体系结构。

592

我们所讨论的语言结构的问题是要将处理器之间的通信，以及处理器与其他处理器的存储器之间的通信减到最少。这里的假设是，处理器对于自己的存储器的数据存取，比对其他处理器的存储器的数据存取要快。设计优良的编译器能够进行许多这样的优化，但是如果程序人员能够给编译器提供可能的有关并发的信息，效果将会好得多。

13.9.1 高性能Fortran

高性能Fortran（HPF）（ACM，1993b）是对于Fortran 90所进行的一系列扩展，它的目的在于允许程序人员给编译器提供信息，从而帮助优化多处理器计算机上的程序执行。HPF包括了新的说明语句以及内建的子程序。在这一节里我们仅讨论这些新的语句。

HPF的主要说明语句被用来说明处理器的数目，这些处理器的存储器中的数据分布，以及数据之间就存储器位置而言的对准方式。HPF的说明语句以一种特殊注释语句的形式出现于Fortran程序中。每一条说明语句都由前缀!HPF$来引出。这里的!是Fortran 90用来开始注释语句的字符。这种前缀使得这些说明语句对于Fortran 90编译器为不可见的，然而却容易被HPF的编译器识别。

PROCESSORS说明语句具有这样的形式：

```
!HPF$ PROCESSORS procs (n)
```

使用这条语句来给编译器说明，这个程序产生的代码所能够使用的处理器的数目。这一条

[5] 虽然ALGOL 68中包括了信号量类型以便处理语句层次的并发，但我们将不在这里讨论关于信号量和并发性的应用。

信息连同其他的说明一起，告诉编译器如何将数据分配到与处理器相关联的存储器上。

DISTRIBUTE语句说明将要分配什么数据，以及将要使用的分配类型。它所具有的形式是：

```
!HPF$ DISTRIBUTE (类型) ONTO procs :: 标识符_表列
```

在这条语句中，“类型”可以是BLOCK或CYCLIC。而“标识符_表列”是一些将要被分配的数组变量名。将一个被说明按照BLOCK方式分配的变量，分成为n个大小相等的组，其中的每一组由连续的数组元素所组成；将这些元素平均地分配给所有处理器的存储器。例如，如果一个有500个元素的称为LIST的数组被按BLOCK方式分配给五个处理器，LIST的前100个元素将会被存储在第一个处理器的存储器中，第二个100个元素存储在第二个处理器的存储器中，并依此类推。CYCLIC分配则将数组中的单个元素循环地存储到那些处理器的存储器中。例如，如果LIST是按照CYCLIC方式被分配到五个处理器中，那么LIST的第一个元素将被存储到第一个处理器的存储器中，第二个元素则到第二个处理器的存储器中，依此类推。 593

ALIGN语句的形式是：

```
ALIGN array1_element WITH array2_element
```

使用ALIGN语句来将一个数组的分配与另外一个数组的分配相关联。例如，

```
ALIGN list1(index) WITH list2(index+1)
```

说明对于所有的index值，list1中下标为index的元素与list2中下标为index+1的元素，将被存储于同一个处理器的存储器中。对于ALIGN语句中的两个数组的引用，将会一同出现在程序中的某一条语句中。将它们放置于相同的存储器（这也就意味着相同的处理器）中，能够确保对于它们的引用将尽可能地接近。

考虑下面的示例代码段：

```
      REAL list_1 (1000), list_2 (1000)
      INTEGER list_3 (500), list_4 (501)
!HPF$ PROCESSORS proc (10)
!HPF$ DISTRIBUTE (BLOCK) ONTO procs :: list_1, list_2
!HPF$ ALIGN list_3 (index) WITH list_4 (index+1)
      ...
      list_1 (index) = list_2 (index)
      list_3 (index) = list_4 (index+1)
```

在每次执行上面的赋值语句时，会将两个被引用的数组元素存储在同一个处理器的存储器中。

HPF的说明语句实际上只是为编译器提供信息，而编译器则或许会，也或许不会使用这种信息来优化它所产生的代码。编译器实际上会怎么做，取决于编译器的复杂程度以及目标机器的体系结构。

FORALL语句，说明一组可以被并发执行的语句。例如，

```
FORALL (index = 1:1000) list_1 (index) = list_2 (index)
```

说明将数组list_1的元素的值赋给list_2中的相对应的元素。在概念上，它说明在所有的1 000个赋值操作发生之前，能够首先对于赋值语句的右边求值。这就允许了并发地执行所有的赋值语句。HPF的FORALL语句也被包含在Fortran 95之中。

我们仅仅简略地讨论了HPF的部分功能。然而，这些讨论给读者提供了充分的信息，来理解用于具有大量数目处理器的计算机的程序设计语言之扩展的思想。 594

小结

能够在指令、语句或子程序的层次上并发地执行。当使用多处理器来实际地执行并发单位时，我们将这种并发称为物理并发。如果将并发单位运行于单个的处理器之上，我们仍称这种并发为逻辑并发。我们可以将所有的并发的基本概念模型，都称为逻辑并发。

可以将大多数的多处理器计算机分为两类：SIMD或MIMD类。MIMD计算机可以是分布式的。

支持子程序层次的并发的语言必须提供两种主要的功能：对于共享数据结构的互斥访问（竞争同步），以及任务间的合作（合作同步）。

任务可能存在不同的状态：新生、就绪、运行、阻塞以及死亡状态。

信号量是一种数据结构，它由一个整数以及一个任务描述队列组成。能够使用信号量来提供并发任务之间的竞争同步和合作同步。信号量很容易被不正确地使用，这会产生编译器、连接器、或运行时系统所不能够发现的错误。

管程是一种数据抽象，它提供一种自然的方式以允许任务间对于共享数据的互斥访问。对几种程序设计语言都支持管程，如Ada、Java和C#，具有管程的语言还必须提供某种形式的信号量以进行合作同步。

Ada中基于消息传递的模型，为并发提供了复杂而有效率的结构。Ada中的任务是重任务。任务间通过会合机制来相互通讯，这种机制即为同步的消息传递。会合是一个任务接受来自另外一个任务的消息的一种行为。Ada包括简单与复杂的方法来控制任务间会合的发生。

Ada 95包括了附加的功能来支持并发，其中主要的功能包括受保护的对象和异步消息传递。Ada 95以两种方式来支持管程：任务和受保护的对象。

Java使用一种相当简单而有效的方式来提供线程的并发单位。任何继承Thread或实现 Runnable的类都能够覆盖一个被继承的方法，称为run方法，并且能够使得这个方法的代码与其他这样的方法以及main方法并发地执行。将访问共享数据的方法定义为同步的，就能够实现竞争同步。即使是很小的代码段也能够同步。通过使用wait、notify和notifyAll方法来实现合作同步。Thread类还提供sleep、yield、join和interrupt方法。

595

C#中对并发的支持基于Java的模式，然而却更为复杂。任何方法都可以运行于线程之中。C#使用Interlock类和Monitor类以及lock语句来支持一种线程同步。

为性能Fortran中包括了一些语句，用以说明如何将数据分配到与多处理器连接的存储器的单位中。还包括了一些语句来说明能够被并发执行的语句系列。

文献注释

Andrews and Schneider（1983）、Holt et al.（1978）和Ben-Ari（1982）详细讨论了并发中的一般课题。各种概念的开发及管程在并发Pascal中的实现，Brinch Hansen（1977）对其进行了描述。

Hoare（1978）和Brinch Hansen（1978）讨论了并发单位控制的消息传递模型的早期开发。Ichbiah et al.（1979）深入讨论了Ada中的任务控制的发展。Ada 95在ARM（1995）中进行了详细的描述。关于高性能Fortran则在ACM（1993）中被描述。

复习题

1. 程序中并发的一种可能的层次是什么？
2. 描述一个SIMD计算机的逻辑体系结构。
3. 描述一个MIMD计算机的逻辑体系结构。
4. SIMD计算机能够最好地支持什么层次的程序并发？
5. MIMD计算机能够最好地支持什么层次的程序并发？
6. 物理并发与逻辑并发之间的差别是什么？
7. 一个程序中的控制线是什么？

8. 定义任务、不相交任务、同步、竞争同步与合作同步、活性、竞争条件以及死锁。
9. 哪种任务不需要任何类型的同步？
10. 描述任务可能会存在的五种不同的状态。
11. 任务池结构的目的是什么？
12. 支持并发的语言的有什么设计问题？
13. 描述信号量的等待和释放操作的行为。 596
14. 什么是二元信号量？什么是计数信号量？
15. 使用信号量提供同步的主要问题是什么？
16. 管程相对于信号量具有什么优点？
17. 定义会合、accept子句、entry子句、通动者任务、服务者任务、扩展的accept子句、开放的accept子句、关闭的accept子句以及完成的任务。
18. 通过管程的并发与通过消息传递的并发，哪一种更为通用？
19. Ada中的任务是静态地还是动态地创建的？
20. 扩展的accept子句的目的是什么？
21. Ada中的任务怎样来提供合作同步？
22. 相对于提供对于共享数据对象访问的任务而言，Ada 95 中的受保护的对象的优点是什么？
23. 描述Ada 95中的异步select子句。
24. 哪种情况的Java程序单位能够与应用程序中的 main方法并发地运行？
25. Java 中的sleep方法有什么用途？
26. Java 中的yield方法有什么用途？
27. Java 中的join方法有什么用途？
28. Java 中的interrupt方法有什么用途？
29. Java中两个能够实现同步的结构是什么？
30. 描述在Java中用来支持合作同步的三个方法的行为。
31. 什么种类的Java中的对象是管程？
32. 解释为什么Java中包括了Runnable接口。
33. 什么种类的方法可以运行于C#的线程之中？
34. C# 中的Sleep方法相对于Java中的sleep方法有什么差别？
35. C# 中的Abort方法究竟有什么用途？
36. Interlock类的目的是什么？
37. C# 中的lock语句有什么用途？
38. 新性能 Fortran中的说明语句的目标是什么？
39. 新性能 Fortran中的FORALL语句的目的是什么？ 597

练习题

1. 请解释在支持协同程序、但不支持并发的程序设计环境中，为什么竞争同步不是一个问题。
2. 当发现死锁时，一个系统所能采取的最佳行动是什么？
3. 忙于等待，是一个任务等待一件给定的事件时不断地测试那件事件发生的一种方法。这样一种方式的主要问题是什么？
4. 在13.3节的生产者-消费者的示例中，假设我们不正确地在消费者过程中使用了 wait(access)代替了release (access)。当执行这个系统时，这个错误将导致什么结果？
5. 假设一本关于使用 Intel Pentium 处理器的计算机的汇编语言程序设计的书，确定这本书提供了什么指令来支持信号量的构造。

6. 假如两个任务A和B必须使用共享变量Buf_Size。任务A将Buf_Size增加2，任务B则从它减去1。假设这种算术操作具有三个步骤：取得当前值，进行计算和放回新的值。在缺乏竞争同步的情况下，可能的事件序列是什么？操作的结果会是什么？假设Buf_Size的初始值为6。
7. 比较Java和Ada中的竞争同步机制。
8. 比较Java与Ada中的合作同步机制。
9. 如果一个管程过程调用同一个管程中的另一个过程，会发生什么情况？
10. 解释这两种情况下合作同步的相对安全性：使用信号量与使用Ada任务中的when子句。

程序设计练习题

1. 编写一个Ada任务来实现通用的信号量。
2. 编写一个Ada任务来管理一个共享缓冲区，就像在我们例子中的那样；但必须使用程序设计练习题1中的信号量任务。
3. 在Ada中定义信号量，并且使用这些信号量来为示例中的共享缓冲区提供合作同步与竞争同步。
4. 使用Java来编写程序设计练习题3中的问题。
5. 使用C#来编写这一章中的共享缓冲区的例子。
6. 可以将读者—作者的问题陈述如下：任意数目的任务可以并发地从一个共享存储地址读出，然而当一个任务必须写入这个共享存储地址时，这个任务必须具有唯一的存取。
7. 使用Ada来编写程序设计练习题6中的问题。
8. 使用C#来编写程序设计练习题6中的问题。

第14章　异常处理和事件处理

这一章讨论程序设计语言对于许多当代程序中两个相关部分的支持，这两个部分就是异常处理和事件处理。异常和事件发生的时间都是不可以确定的，而且这两者都最好地处理特殊的语言结构和过程。其中有些概念（例如传播）对异常处理和事件处理几乎是相同的。

首先我们描述异常处理的基本概念，包括可以被硬件和软件检测的异常、异常处理程序和异常的产生。然后介绍和讨论异常处理的设计问题，包括异常和异常处理程序的绑定、继续、默认的异常处理程序和异常的禁用。接下来将描述和评估三种程序设计语言的异常处理设施，包括Ada、C++和Java。

本章后面的部分是事件处理。我们将首先介绍事件处理的基本概念，然后讨论对于Java的GUI成分中事件处理的方法。

14.1　异常处理概述

大多数的计算机硬件系统，具有检测某些运行时错误情况的能力，例如浮点数上溢。在许多早期的程序设计语言所使用的设计和实现的方式，使得用户程序就不能发现也不能处理这样的错误。在这些早期语言中，发生一种错误仅仅是引起程序终止，并且将控制转移到操作系统。操作系统对于运行时错误的典型反应，是显示一条诊断信息，这条信息可能含义清楚而有用，或者是十分模糊的。在显示了出错信息后，系统就将此程序终止。

然而，在输入和输出操作情况下，有时候情况是不相同的。例如Fortran中的Read语句，就能够检测输入错误以及“文件结尾”的情况，而这两种问题都能够被硬件输入设备发现。在这两种情况下，Read语句都能够说明用户程序中用来处理这些情况的一些语句标号。当然我们不应该总是将“文件结尾”的情况认为是一种错误，它通常只不过是一种信号，表明某种类型的处理完成了，而另外一种新的类型的处理即将开始。尽管“文件结尾”明显不同于错误事件，如输入过程失败，Fortran使用同样的机制来处理这两种情形。考虑下面Fortran的Read语句：

```
Read(Unit=5, Fmt=1000, Err=100, End=999) Weight
```

这里的Err子句说明，如果在读操作中发生错误，将控制转移到标号为100的语句。End子句说明，如果在读操作中遇到“文件结尾”，则将控制转移到标号为999的语句。因此Fortran使用了简单的分支来处理输入错误与文件结尾这两种情况。

有一种类型的严重错误就不能被硬件发现，但是可以由编译器所产生的代码发现。例如，硬件几乎不可能发现数组下标范围的错误。但是这些错误却能够导致在后面程序执行中才会注意到的致命错误。

数组下标范围错误的检测有时是语言的设计所要求的。例如，Java编译器就要求产生代码来检测每一个下标表达式的正确性。（但是如果编译器可以在编译时确定，数组下标的表达式不会具有越界的值时，Java编译器便不会产生这样的代码，例如，下标是一个字面常量的情形。）在C中，由于认为检测带来的好处与检测所付出的代价相比并不值得，所以不进行下标范围的

⊖　在20世纪70年代，的确有一些计算机的硬件中能够发现数组下标范围的错误。

检测。而在其他语言的编译器中，如果程序需要或者是在运行编译器的命令之中，可以选择是否进行下标范围的检测。

许多当代语言的设计人员，已经在语言中包括了某些机制，能够使程序按照一种标准的方式，对于某些运行时错误和其他程序所检测到的异常事件做出反应。当硬件或系统软件检查出某种事件的发生时，还能够通知程序以使能够做出必要的反应。所有这些机制都被统称为异常处理。

也许使一些语言不包括异常处理的最重要的原因，是这种处理给语言所增加的复杂性。

14.1.1 基本概念

我们将硬件能够发现的错误，如读盘错误，以及硬件能够发现的非寻常情况，如文件结束，都认为是异常。再进一步，我们将异常的概念扩展到包括任何可以由软件发现的错误或任何非寻常的情况。因此，我们将**异常**（exception）定义为：不论存在错误与否，任何可能由硬件或软件发现的并需要特殊处理的非寻常事件。

异常检测所要求的特殊处理被称为**异常处理**（exception handling）。通过一种被称为异常处理程序（exception handler）的代码单位，来进行这种处理。当发生与异常相关的事件时，异常就被提出（raised）。在一些最近的基于C的语言中，将异常出现称为抛出的（thrown），而不是提出的。[⊖] 异常处理程序，通常根据异常类型的不同而不同。当检测进行到文件的结尾时，几乎总是需要这种程序做出某种特殊的行为。但显然，这种行为不属于数组下标范围错误的异常。有的时候，异常处理程序做出的唯一行动，可能就是产生出错信息以及命令程序终止。

603

在有些情形下，程序人员可能希望暂时忽略某种由硬件检测出来的异常，例如，被零除的情况。这可以通过禁用异常处理来达到。被禁用的异常处理设施可以在稍后再启用。

即使在语言中没有专用的异常处理设施，也可以处理由用户所定义的、并且软件可以检测的异常。如果在一个程序单位里发现了这种异常，通常是由这个单位的调用程序来处理。一种可能的设计是，发送一个额外的参数来作为状态变量，被调用的子程序根据计算的正确性和正常性给状态变量赋值。当状态变量从被调用的单位返回时，调用程序立刻测试这个变量，如果状态变量值指示发生了异常，可能是位于调用单位中的异常处理程序就行动起来。许多C语言的库函数使用这种方式的一种变体：返回值被用来作为出错指示器。

另外一种可能的方案是传递一个标号参数给子程序。当然，这只能是在允许将标号作为参数使用的语言中。如果异常发生，传递的标号允许被调用单位返回到调用程序中不同的地点。与第一种方案一样，异常处理程序通常是调用单位代码中的一段。这是Fortran中标号参数的一种常见用法。

第二种可能的方案是将一个异常处理程序写成单独的子程序，并且可以将这个子程序作为参数传递给被调用的单位。在这种情况下，异常处理子程序由调用程序来提供。当异常出现时，被调用单位直接调用异常处理子程序。这种方法中的问题是：不论需要与否，每一次的子程序调用，都必须发送异常处理子程序。此外，为了要处理一些不同类型的异常，还必须传递几个异常处理子程序，从而增加了代码的复杂性。

如果希望在发现异常的单位中处理异常，可以将异常处理程序编写成这个单位中的代码段。

在语言中包括异常处理有着明显的优点。首先，如果没有异常处理的话，程序中所需要的

⊖ C++是基于C的语言中第一种包括了异常处理设施的。之所以使用术语"抛出"，而不是使用"提出"，是因为在标准C程序库中包括了一个名字为raise（提出）的函数。

检测错误情况的代码会将整个程序拖乱。例如，假设一个子程序中的表达式包含了10种对于矩阵mat中元素的引用，并且其中任何一种引用都可能产生下标越界的错误。此外，还假设这种语言不要求下标范围的检测。如果没有内建的异常处理的话，每一个上述的引用操作的前面，都必须有一段代码来检测可能的下标越界错误。例如，考虑下面的对于mat中一个元素的引用，矩阵mat具有10行20列：

```
if (row >= 0 && row < 10 && col >= 0 && col < 20)
  sum += mat[row][col];
else
  System.out.println("Index range error on mat, row = " +
                     row + " col = " + col);
```

604

语言中内建的异常处理使得编译器能够在每一个矩阵元素的引用之前插入这种检测，从而极大地缩短和简化了源程序。

语言支持异常处理的另外一个优点是异常传播。异常传播允许在一个程序单位中出现的异常，由这个程序单位的动态或静态祖先中的某一个其他程序单位来处理。这就允许了多个不同的程序单位使用一个异常处理程序。这种复用能够大大减少程序开发的费用、程序的规模和程序的复杂性。

一种支持异常处理的语言，就驱使用户去考虑在程序执行期间所有可能发生的事件，以及如何来处理这些事件。这种方法远比不考虑各种可能性、仅仅是希望问题不会出现要好得多。例如，Ada要求一种多向选择结构，以便对于控制表达式所有可能的值采取不同的行动。

最后，异常处理可以大大简化程序中对于那些不是错误、但是非寻常情形的处理。如果没有异常处理，这种处理将使程序变得错综复杂。

14.1.2 设计问题

现在我们来讨论，当异常处理是程序设计语言的组成部分时，异常处理系统的一些设计问题。这种系统应该允许内建的，以及用户定义的异常和异常处理程序。注意，预定义的异常是隐式地提出的，而用户定义的异常则是显式地由用户代码提出的。考虑下面的子程序框架，其中包括了隐式出现的异常的异常处理机制：

```
void example() {
  ...
  average = sum / total;
  ...
  return;
/* 异常处理器 */
  when zero_divide {
    average = 0;
    printf("Error-divisor (total) is zero\n");
  }
  ...
}
```

605

这个函数能够在截获被零除的异常后，将控制转移到适当的处理程序，然后执行处理程序。

异常处理的第一个重要的设计问题，是怎样将发生的异常与异常处理程序相绑定。这个问题发生在两个不同的层次上。在程序单位层次上的问题是，怎样将出现在一个程序单位中不同地点相同的异常，与不同的异常处理程序相绑定。例如在上面子程序的示例中，就编写了一个异常处理程序（程序中所显示的那段代码），来处理在一条特定语句中所出现的被零除的异常。

历史注释

PL/I (ANSI, 1976) 开创了允许用户程序直接参与异常处理的概念。这种语言允许用户针对语言定义的一系列的异常，编写异常处理程序。此外，PL/I还引入了用户定义的异常的概念，允许程序创建可由软件检测的异常。这些异常使用了与内建异常相同的机制。

自从PL/I被设计以来，人们进行了大量的工作来设计异常处理的替代方法。特别是，CLU (Liskov et al., 1984)，Mesa (Mitchell et al., 1979)，Ada，COMMON LISP (Steele, 1984)，ML (Milner et al., 1990)，C++，Modula-3 (Cardelli et al., 1989)，Eiffel，Java和C#，都包括了异常处理的设施。

但假设函数中具有几个带有除法操作符的其他表达式。对于这些操作符，这个异常处理程序或许就不适用了。因此，应该可以将在某些特定的语句中提出的那些异常，绑定到某些特定的异常处理程序之上，即使相同的异常可能会被许多不同的语句提出。

在较高的层次上，绑定问题可能出现于提出异常的程序单位中不存在局部异常处理程序。在这种情况下设计人员必须决定，是否将异常传播到某个其他的单位上去。如果要传播，又应该传播到那一个单位？怎样来进行异常的传播以及应该传播多远，所有这些对于异常处理程序的可写性具有极大的影响。例如，如果处理器必须是局部的，那么就得编写许多的处理程序，这会使得程序的写与读都变得很复杂。另一方面，如果进行异常的传播，一个处理程序可能要处理多个程序单位中所提出的相同的异常。这样就可能要求处理器比期望中的更为通用。

在异常处理程序执行之后，可以将控制转移到程序中、处理程序代码之外的某个位置，或者仅仅是终止程序的执行。我们称这个问题为处理器执行后的继续控制问题，或简称为继续（continuation）。终止程序的执行显然是最为简单的选择，并且在存在着许多错误异常的情况下，这是最佳的选择。然而在其他的情况，特别是在那些非灾难但非错误事件的情况下，继续执行则是最佳的选择。这种设计称为重新开始（resumption）。此时，需要选择某些规则来决定从那里开始继续执行。被继续执行的可能是出现异常的语句，也可能是在出现异常的语句之后的语句，或者可能是某一个其他的程序单位。回到出现异常的语句可能是一个好的选择，但是在错误异常的情况下，如果处理程序能够修正导致异常被提出的数值或操作，这种方法才是有用的。否则只能是再次地提出异常。对于错误异常所必需的修正，通常非常困难，即使可能修正的话，也可能不是一种可靠的方法。它仅仅允许程序除去问题的症状，而不消除根源。

606

图14-1说明异常与异常处理程序的绑定以及继续。

当包含异常处理时，子程序能通过两种方法终止其执行：执行完成或遇到异常。在一些情形下，不管子程序是怎样终止执行的，完成一些计算是必须的。指定这种计算的功能称为析构（finalization）。是否支持结束化明显是异常处理的设计问题。

另一个设计问题是：如果允许用户来定义异常，怎样来说明异常？通常的答案是将异常的声明，放置在这些异常可能出现的程序单位的声明部分，被声明异常的作用域，通常是包含这个声明的程序单位的作用域。

在语言提供预定义的异常的情形下，随之而来的还有好几个其他的设计问题。例如，语言的运行时系统是否应该为这些内建异常，提供默认处理器？或者，用户是否应该为所有的异常都编写异常处理程序？另外的一个问题是，内建异常是否能够被用户程序显式地提出。如果用户能在可以通过软件检测的情况下，使用预定义的异常处理的程序，那将是很方便的。

607

另一个问题是：是否应该将可以被硬件检测出来的错误视为可以被用户程序处理的异常。如果不应该的话，那么显然所有异常都是可以通过软件来检测的。这里一个相关的问题是：是

台应该有任何预定义的异常。预定义的异常，由硬件或系统软件隐式地提出。

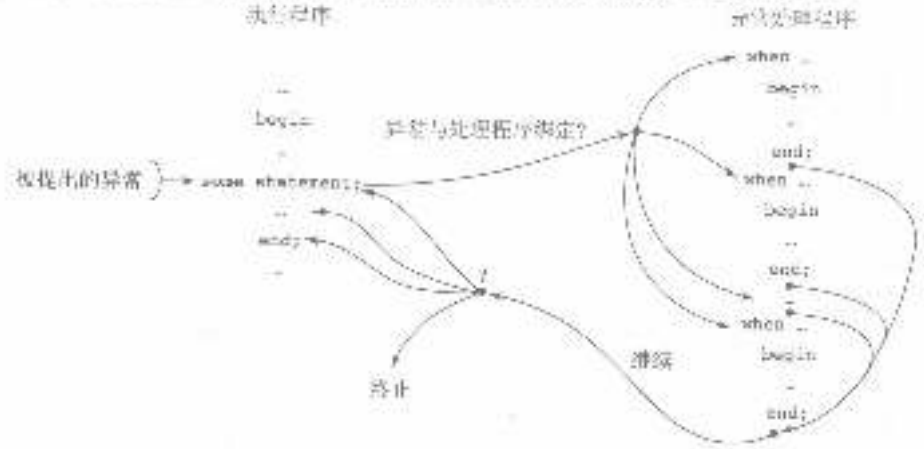


图14-1 异常处理的控制流程

最后的一个问题是：是否可以暂时或永远地禁用预定义的或用户定义的异常。在某种程度上，这是一个哲学问题，尤其是在预定义错误的情况下。例如，假设一种语言具有一种预定义的异常，当下标范围错误出现时则这个异常被提出。许多人相信，下标范围错误总是应该被检测的，因而对下标范围错误的检测不应该被禁用。但另外一些人则争辩道，检测下标范围错误的代价太高，如果相信代码没有什么错误的话，就没有必要检测下标范围的错误。

可以将异常处理的设计问题总结如下：

- 怎样以及在何处来说明异常处理程序？它们的作用域是什么？
- 怎样将异常出现和异常处理程序相绑定？
- 提供了某种析构形式吗？
- 在完成执行异常处理程序后，如果要继续执行程序，应该从哪里开始继续执行？（这是继续的问题。）
- 异常信息能传递给处理程序吗？
- 怎样来说明用户定义的异常？
- 如果有预定义的异常，而一些程序又没有自己的异常处理程序，应该给这些程序提供默认的异常处理程序吗？
- 预定义的异常可以被显式地提出吗？
- 应该像能够被处理的异常一样来对待可以被硬件发现的错误吗？
- 存在任何预定义的异常吗？
- 预定义的异常可能被禁用吗？

现在我们已经准备就绪，来讨论三种当代程序设计语言的异常处理设施。

14.2 Ada中的异常处理

Ada中的异常处理是构造更为可靠软件系统的一种有力工具。它包括了两种较早期的语言PL/I和CLU中，设计较好的异常处理部分。

14.2.1 异常处理程序

通常，Ada中的异常处理程序对于提出异常的代码来说是局部的（尽管它们能传播到其他

608 程序单元)。因为异常处理程序与代码具有同样的引用环境，所以异常处理程序不需要也不允许参数。因此，如果在一个与引起异常的单元不同的单元中处理异常，那么没有关于异常的信息能传递到处理函数。[9]

异常处理程序有下面一般的形式，这里给出EBNF形式：

```
when 异常选择 {| 异常选择 } => 语句序列
```

这里的花括号是元符号，表示所包含的内容可能不会出现，也可能任意次地重复出现。异常选择所具有的形式为：

```
异常名 | others
```

异常名代表异常处理程序所要处理的某个或某些特定的异常。语句序列是异常处理程序的程序体。保留字others说明这个异常处理程序，将处理任何没有在其他局部异常处理程序中被提及名称的异常。

能够将异常处理程序包含在块、子程序、程序包或任务体中。不论它们是出现于块中，还是出现在程序单位中，都将异常处理程序集中在exception子句里，并且必须将这种子句放置在块或单位的尾部。例如，exception子句通常的形式为：

```
begin
-- 块、或单位体 --
exception
   when 计算名_1 =>
      -- 第一个异常处理程序 --
   when 异常名_2 =>
      -- 第二个异常处理程序 --
      -- 其他的异常处理程序 --
end;
```

任何能够合法出现于块中或单位中的语句，也能够合法地出现在异常处理程序中。

14.2.2 异常与异常处理程序的绑定

当提出异常的程序块或程序单位包含了一个异常处理程序时，异常与处理程序的绑定为静
609 态的。如果当提出异常的程序块或程序单位，没有包含处理这种异常的处理程序时，这个异常就被传播到某个其他的块或单位中去。异常的传播方式取决于出现异常的程序实体。

当异常在一个过程中被提出时，不论是在声明语句确立时，还是在程序体执行时，如果这个过程没有处理这种异常的处理程序，异常就被隐式地在调用的位置传播到调用程序单位去。这种策略反映了一种设计思想，即异常传播应该是经由控制途径（即动态祖先）的回溯，而不是经由静态祖先。

如果异常所传播到的调用单位也没有这种异常的处理程序，就再一次将这个异常传播到这个单位的调用单位。如果必要的话，这种过程将一直继续到主程序。如果异常被传播到主程序，而在主程序也找不到相应的处理程序，程序就被终止。

在异常处理中，可以认为 Ada 的程序块是无参数的过程。当执行控制达到程序块的第一条语句时，这个块由父辈过程所“调用”。当一个异常在一个程序块中被提出，不论是在块的声明语句，或者是在块的可执行语句中，而且这个程序块中不具有这种异常的处理器时，便将异常传播到下一个比较大的包含作用域中，即“调用”这个程序块的代码之中。异常传播的地点，

[9] 它并不是完全正确。处理程序获取异常名，该异常的简短描述和引发异常的大致位置是可能的。

是在提出它的程序块的结尾处，也即它将“返回”的地点。

当异常在一个程序包的体中被提出，并且这个包体中不具有处理这种异常的处理程序时，便将这个异常传播到含有这个包的声明单位中的声明部分。如果这个包碰巧是一个分别编译的库程序单位，程序就将终止。

如果异常是在任务体的最外一层被提出，而且这个任务包含了一个处理这种异常的处理程序，这个处理程序就被执行，并且任务被标记为完成。如果这个任务没有包含一个处理这种异常的处理程序，这个任务就简单地被标记为完成。但异常并不被传播。任务的控制机制过于复杂，因而无法合理并简单地回答一个未处理的异常是否应该被传播。

异常也可能发生在子程序、块、包和任务的声明段的确立期间。例如，假设调用一个函数来对其声明语句中的一个变量设定初值，如：

```
procedure River is
  Current_Flow : Float := Get_Flow;
  ...
  begin
  ...
 end River;
```

假设Get_Flow是一个无参数的函数。如果Get_Flow提出一个异常，并将这个异常传播给它的调用程序，这个异常就在这个声明语句中被提出。声明确立期间的存储空间分配，也可能提出异常。当异常的提出发生在子程序、块、包和任务的声明确立期间，也完全可以象在代码中提出的异常一样这样些异常传播。在任务的情形，任务被标记为完成但并没有更进一步的异常处理；内置异常Tasking_Error将在任务启动之处被提出。 610

14.2.3 继续

在Ada中，提出一个异常的程序块或程序单位，连同所有传播这个异常但又不能处理它的程序单位一起，总是被终止。在处理异常之后，控制从来不会隐式地返回到提出异常的块或程序单位。控制只是在exception子句之后继续，而exception子句又总是位于块或程序单位的末部。这使得控制立即就返回到较高的层次上去。

在异常处理程序的执行完成以后，当决定执行应该从哪里开始继续时，Ada的设计小组没有什么选择，这是由于Ada的需求说明（Department of Defense，1980a），清楚地规定了提出异常的程序单位不能够被继续或重新启动。然而在块的情况下，在一条语句提出异常并在异常被处理之后，这条语句可以被重新试用。例如，假设提出异常的语句，以及处理这种异常的处理程序都在同一个程序块中，而且这个程序块本身被包含在一个循环里。下面的代码段示例，说明了这样的一个结构：这个代码段从键入获得所期望范围中的四个整数值：

```
...
type Age_Type is range 0..125;
type Age_List_Type is array (1..4) of Age_Type;
package Age_IO is new Integer_IO (Age_Type);
use Age_IO;
Age_List : Age_List_Type;
...
begin
for Age_Count in 1..4 loop
  loop  -- 当异常出现时用于重复的循环
 Except_Blk:
```

```
        begin  -- 实现异常处理的复合语句
        Put_Line("Enter an integer in the range 0..125");
        Get(Age_List(Age_Count));
        exit;
        exception
          when Data_Error =>  -- 输入的字符串不是数字
        Put_Line("Illegal numeric value");
        Put_Line("Please try again");
          when Constraint_Error =>  -- 输入值 < 0 或者 > 125
        Put_Line("Input number is out of range");
        Put_Line("Please try again");
        end Except_Blk;
      end loop;  -- 当有异常时，结束无限循环，以使重复输入
    end loop;  -- 结束1..5循环中的 Age_Count
    ---
```

在接收到一个有效的输入数字前，控制被停留在包含这个块的内循环中。

14.2.4 其他设计选择

在默认包Standard中定义了四种异常：

```
Constraint_Error
Program_Error
Storage_Error
Tasking_Error
```

其中的每一种实际上是异常的一个种类。例如，当数组下标越界时，当一个具有范围限制的数值变量出现范围错误时，当引用一个记录域而这个域不在一个判别联合中时，以及在许多其他的情况下，异常Constraint_Error会被抛出。

除了在Standard中定义的异常之外，其他预定义的包还定义了其他的异常，例如，Ada.Text_IO定义了End_Error异常。

用户定义的异常使用下面的声明形式来定义：

```
异常名表列: exception
```

这样定义的异常与预定义异常被完全一样来对待，除了这种异常必须被显式地抛出之外。有默认的处理程序用于预定义的异常，这些程序都导致程序终止。

异常由raise语句来显性地提出，raise语句的一般形式为

```
raise [异常_名]
```

能够出现一条不命名异常raise语句的唯一位置，是在一个异常处理程序之中。此时，它抛出引起处理器执行的同一个异常。它将具有依前面描述的异常传播规则，来传播异常的实际效果。在一个异常处理程序中的raise语句很有用处，尤其当人们希望在异常被提出的位置打印一条出错信息，然而却是在另外的一个位置来处理这个异常时。

Ada中的pragma是一条对于编译器的指令。通过使用Suppress pragma指令可以禁用某些Ada程序中内处异常组成部分的特定的运行时检测。这条指令的简单形式为：

```
pragma Suppress (检测_名)
```

这里的"检测_名"是一种特定异常检测的名字。在本章的后面，还将给出一些异常检测的示例。

Suppress pragma指令只能够出现在声明段。当它出现时，就可能在相关的块或程序单元中暂停所说明的检测。显式提出的异常不受Suppress的影响。虽然并不存在强制性的要求，但大多数Ada的编译器都实现了Suppress pragma指令。

能够被禁用的检测有示例如下：Index_Check和Range_Check，它们说明的是Ada程序中的两种常用检测：Index_Check指的是数组下标范围的检测，Range_Check指的是检测赋给子类型变量的值的范围。如果Index_Check或Range_Check被违反的话，就会提出Constraint_Error异常。Division_Check和Overflow_Check是与Numeric_ Error异常有关的可禁用的检测。下面的pragma指令禁用数组下标范围的检测：

```
pragma Suppress(Index_Check);
```

还可以使用Suppress将所说明的检测，进一步地限制到特定的对象、类型、子类型和程序单位上。

14.2.5 一个示例

下面的示例程序，介绍Ada异常处理程序中，一些简单而通常的用法。这个程序使用一个计数器数组来计算并打印输入分数的分布。输入是一列分数，用负数来终止输入。因为分数是自然数（Natural类型），所以会产生Constraint_Error异常。一共有10个类别的分数等级（0～9，10～19，…，90～100）。使用输入分数来计算它们在计数器数组中的下标：一个计数器数组中的元素对应于一个分数等级类别。无效的输入分数，被认为是计数器数组的下标出错。又因为所有的分数等级类别都具有10个可能的分数值，然而最高的等级类别却有11个分数值：（90，91，…，100），所以100分在分数分布的计算中，是一个特殊的分数（A的分数比B的分数、或C的分数多的事实，说明老师的慷慨）。同样也是使用处理无效输入分数的异常处理程序，来处理100分的情形。

613

```
--分数的分布
--输入：一列表示分数的自然数，后跟
--      一个负数
--输出：分数分布情况，分数等级
--      (0~9,10~19,...,90~100)
with Ada.Text_IO, Ada.Integer_Text_IO;
use Ada.Text_IO, Ada.Integer_Text_IO;
procedure Grade_Distribution is
  Frog: array (1..10) of Integer := (others => 0);
  New_Grade : Natural;
  Index,
  Limit_1,
  Limit_2 : Integer;
  begin
  Grade_Loop:
    loop
    begin  -- 下标范围处理块
    Get(New_Grade);
    exception
      when Constraint_Error =>  -- for negative input
        exit Grade_Loop;
   end;  -- end of negative input block
    Index := New_Grade / 10 + 1;
      begin  -- A block for the subscript range handler
```

```
            Freq(Index) := Freq(Index) + 1;
            exception
              when Constraint_Error =>  -- for index range errors
                if New_Grade = 100 then
                  Freq(10) := Freq(10) + 1;
                else
                    Put("ERROR -- new grade: ");
                    Put(New_Grade);
                    Put(" is out of range");
                    New_Line;
                end if;
            end;  -- 结束
          end loop;
        -- Produce output
            Put("Limits  Frequency");
            New_Line; New_Line;
            for Index in 0..9 loop
              Limit_1 := 10 * Index;
              Limit_2 := Limit_1 + 9;
              if Index = 9 then
                Limit_2 := 100;
              end if;
614           Put(Limit_1);
            Put(Limit_2);
            Put(Freq(Index + 1));
            New_Line;
          end loop;  --结束循环Index in 0..9 ...
        end Grade_Distribution;
```

注意，处理无效输入分数的代码是在自己的局部程序块中。这允许了异常处理之后程序的继续，如同前面所描述的键入数值的例子一样。负数输入的处理函数也在它自己的块中，这样的原因是限制处理负数输入引起Constraint_Error异常的处理函数的作用域。

14.2.6 评估

如对其他的一些语言的结构一样，至少是在设计时期（20世纪70年代后期至20世纪80年代初期），Ada中异常处理的设计代表了参与人员的一致意见。相对于PL/I中的异常处理，Ada的设计显然有了重大的进步（参见14.1.2小节中的历史注释部分）。在相当长的时期内，Ada曾经是唯一广泛使用的包含了异常处理的语言。现在，异常处理已经成为大多数新近定义的语言中的一部分，曾经有段时间，Ada是包含异常处理的广泛使用的语言。

Ada的异常处理有一些问题，其中一个问题是允许异常传播到异常不可见的外部作用域的传播模型。发现传播异常的引发位置并不总是可能的。

另一个问题是对任务异常处理的缺乏。例如，引发异常的任务不能处理它，只是简单地关闭了任务。

最后，当Ada 95加入对面向对象程序设计的支持时，它的异常处理没有扩展来处理新结构。例如，当在块中创建和使用类的几个对象，它们中的一个传播一个异常时，找到那一个对象引发了异常是可能的。

Romanovsky和Sander (2001)讨论了Ada的异常处理问题。

14.3 C++中的异常处理

ANSI C++标准化委员会于1990年接受了C++的异常处理，此后，C++的异常处理被C++的许

多种实现所采纳。它的设计部分地基于CLU、Ada和ML语言中的异常处理的设计。C++中的异常处理与Ada中的异常处理之间的主要区别是C++中没有预定义的异常（除了它的标准程序库中的之外）。其次，C++中的异常只能是用户定义或库定义的，并且这些异常都是经显式提出的。 615

14.3.1 异常处理程序

在14.2节里我们曾经了解到，Ada使用程序单位或块来说明异常处理程序的作用域。C++则使用一种特别的结构，这种结构由保留字try引出。一个try结构包括一个称为try子句的复合语句以及一列异常处理程序。这些复合语句定义了所跟随的异常处理程序的作用域。这种构造的一般的形式是

```
try {
//** 提出一个异常的代码
}
catch (形参) {
//^^ 一个异常处理程序的体
}
...
catch (形参) {
//** 一个异常处理程序的体
}
```

其中的每一个catch函数都是一个异常处理程序。一个catch函数可能只具有一个形参，这个形参与C++的函数定义中的相类似，但它也可能仅仅是一种省略形参。具有一个省略形参的处理程序，是一种捕捉所有异常的处理程序；如果对于提出的异常没有找到合适处理程序，这种处理程序就被启动，就像在函数原型中的一样，这个形参也可以仅仅是一个类型说明符，如float。在这种情况下，形参的唯一目的，就是使得这个异常处理程序能够唯一地被标识。当关于异常的信息被传递给处理程序时，这个形参包括了一个用于标识目的的变量名。因为这种参数的类可以是任何用户所定义的类，从而参数可以包括任何所需数目数据成员。关于异常与异常处理程序的绑定，将在14.3.2小节中进行讨论。

在C++的异常处理程序中，能够包括C++的任何代码。

14.3.2 异常与异常处理程序的绑定

C++中的异常只能够通过throw语句被显式地提出；throw语句的EBNF的一般形式为：

```
throw [表达式];
```

这里的方括号是元符号，用来说明表达式是可选择的。没有操作数的throw语句，只能出现在处理程序中。当throw语句出现在处理程序中时，异常被重新提出；之后，这个异常将在别处被处理。这种效果与Ada中的完全一样。

throw语句中表达式的类型被用来选择特定的处理程序，当然，被选择的处理程序必须要有匹配的形参类型。在这里匹配的含义如下：如果一个处理程序的形参类型为T、const T、T&（对于类型T的对象的引用）或者const T&，这个处理程序与一个表达式的类型为T的throw语句相匹配。当T是一个类时，如果一个处理程序的参数类型为T，或者是为任何一个T的祖先类，它们也相匹配。throw语句中的表达式与形参的匹配还有一些更加复杂的情形，但是我们将不在这里进行描述。 616

一个try结构中提出的异常将即刻终止执行这个结构中的代码，然后从紧跟在try结构后面的第一个处理程序开始，搜索一个匹配的处理程序。这个搜索匹配的过程，又将继续到后面的另外一个处理程序，直到发现一个匹配的处理程序为止。这意味着，如果在找到完全匹配之

能存在着其他的匹配，就可能不会使用这个完全匹配的处理程序。因此，特定异常的处理函数被放置在列表的顶部，后面接着更通用的处理函数。最后的处理函数通常带有省略（...）的形参，以匹配任何异常。这保证可以捕获所有的异常。

如果一条try子句提出一个异常，然而却没有匹配的异常处理程序与之关联。这时，这个异常就被传播。如果这条try子句被嵌套在另外一条try子句中，这个异常就将被传播到与外层的try子句相关联的处理程序。如果所有的外层的try子句，都不具有相匹配的处理程序，这个异常就将被传播到提出它的函数之调用程序。但如果这个调用程序不具有与之匹配的处理程序，此时就使用默认处理器。关于默认处理器将在14.3.4小节中进行讨论。

14.3.3 继续

在一个处理程序完成执行后，控制将被转移到跟随在try结构之后的第一条语句（同结构中紧接在最后一个处理程序之后的那一条语句）。一个处理程序能够使用一条不具有表达式的throw语句来提出异常，在这种情况下，异常将被传播。

14.3.4 其他设计选择

就14.1.2节中所总结的设计问题而言，C++中的异常处理是十分简单的。它仅仅具有用户定义的异常，并且这些异常还没有被说明（虽然可以将它们声明为新的类）。C++中有一种默认的处理程序，称为unexpected。它的唯一功能就是终止程序的执行。这个处理程序捕捉所有的没有被程序所捕捉的异常。这种异常处理程序可以由一个用户定义的异常处理程序来代替。用来替代的用户定义的处理程序，必须是一个返回void的函数，并且不具有参数。这个替代函数
617 是通过赋予名字set_terminate来设立。C++中的异常不能够被禁用。

C++中的函数能够列出它所可以提出的异常类型（也即throw表达式的类型），只需要在函数的首部附上保留字throw，后面跟随用括号括起来的类型的列。例如：

```
int fun() throw (int, char *) { ... }
```

它说明函数fun可以提出然而却不能处理的类型为int和char *的异常，但不会提出这些类型以外的异常。throw子句的作用是为函数的用户指定该函数可能引发的异常。throw子句在函数和它的调用者之间建立了规则，它保证函数不引发其他异常。如果函数抛出一个列表中没有的异常，程序将终止。注意，编译器忽略throw子句。

如果throw子句的类型是类，那么函数能引发列表类派生的任何异常。如果函数的首部有条throw子句，但它所提出的异常没有在throw子句中被列出，也不是由所列出的类所派生出来的，那么，默认处理器就会被调用。请注意，这种错误不可能在编译时被检测出来。如果throw子句的类型表列为空，这意味着这个函数将不会产生任何异常。但如果函数的首部没有throw的说明的话，这个函数就能够引发任何异常。这个异常类型表列并不是函数的类型中的一部分。

如果一个函数覆盖了有throw子句的函数，那么具有比被覆盖函数更多异常的覆盖函数不能有throw子句。

当一个异常终止了一个try结构时，在这个结构中执行的代码于异常发生之前所分配的所有栈动态和堆动态的变量都要被解除分配。从而，异常处理程序将无法存取这些变量。

虽然在C++中没有规定义的异常，它的标准程序库定义了并且可以抛出异常；例如，可以由程序库的包含类来抛出异常out_of_range，以及可以由数学库的函数来抛出异常overflow_error。

14.3.5 示例

下面的例子与本章前面的14.2.5小节中的Ada程序的例子，具有相同的意图而且使用的是相同的异常处理的方式。这个程序通过使用一个计数器数组，来计算并且显示输入分数之分布。这个数组具有10个不同分数类别。分数的值被用来作为计数器的下标。非法的输入的分数，通过增加选择计数器来检测无效的下标，从而得以发现。 618

```
//分数的统计
//输入：一列表示分数的整数值，后跟
//      一个负数
//输出：分数分布情况，分数范围
//    （0-9,10-19,...,90-100）
#include <iostream.h>
void main() {   //* Any exception can be raised
  int new_grade,
      index,
      limit_1,
      limit_2,
      freq[10] = {0,0,0,0,0,0,0,0,0,0};
// The exception definition to deal with the end of data
class NegativeInputException {
  public:
   NegativeInputException() {  //* Constructor
    cout << "End of input data reached" << endl;
  }  //** end of constructor
}  //** end of NegativeInputException class
  try {
    while (1) {
      cout << "Please input a grade" << endl;
      if ((cin >> new_grade) < 0)  //* Terminating condition
        throw new NegativeInputException();
      index = new_grade / 10;
      {try {
        if (index > 9)
          throw new_grade;
        freq[index]++;
        }  //* end of inner try compound
      catch(int grade) {  //* Handler for index errors
        if (grade == 100)
          freq[9]++;
        else
          cout << "Error -- new grade: " << grade
               << " is out of range" << endl;
        }  //* end of catch(int grade)
      }  //* end of the block for the inner try-catch pair
    }  //* end of while (1)
  }  //* end of outer try block
 catch(NegativeInputException e) {  //** Handler for
                                    //** negative input
   cout << "Limits   Frequency" << endl;
   for (index = 0; index < 10; index++) {
     limit_1 = 10 * index;
     limit_2 = limit_1 + 9;
     if (index == 9)
       limit_2 = 100;
```

```
619         cout << limit_1 << limit_2 << freq[index] << endl;
          } //* end of for (index == 9)
        } //* end of catch (short int)
      } //* end of main
```

我们使用这个程序是为了说明C++中的异常处理机制。然而，这个例子中处理异常的两种方式，都可以通过使用别的更好的方法来进行。使用cin表达式来控制while循环，可以更容易处理"文件结尾"的情况。此外在C++中，下标范围异常通常是由重载下标操作来处理的，然而这种方法还可以是出异常，而不是像在我们的示例中的那样，使用选择结构来进行下标操作的直接检测。

14.3.6 评估

C++的异常处理机制在一些方面与 Ada 的异常处理机制相类似：异常和异常处理程序的绑定都是静态的，并且都将所不能够处理的异常传播给函数的调用程序。然而在其他的一些方面，C++中的设计则是相当不同的：它不具有内建的、能够被硬件检测的并且能够由用户来处理的异常，以及不能够给异常命名。异常通过参数类型与处理程序相连接，形参又可以不存在。处理程序中的形参类型决定调用处理程序的条件，但是又与处理程序所抛出的异常的性质毫不相干。因而，使用异常的预定义类型并不能够改进可读性，如果能够在一种有意义层次结构中使用有意义的名字来定义异常类，情况将会好得多。异常参数提供了一种传递异常信息给异常处理函数的方法。

14.4 Java中的异常处理

在第13章中，曾经在Java的示例程序里包括了异常处理的应用，然而当时并没有进行任何解释。在这一节里，我们将详细描述 Java 的异常处理功能。

Java中的异常处理是以C++的异常处理为基础的，但它的设计与面向对象语言的风格更为接近。此外，Java中还包括了一组由Java的虚拟机器所隐式抛出的预定义异常。

14.4.1 异常类

Java中的所有异常都是对象，这些异常的类都是Throwable类的后裔。Java系统中包括了两种系统定义的异常类，Error和Exception，这两个类都是Throwable的子类。Error类及其后裔类，都与 Java虚拟机器所抛出的错误相关；例如，已经没有了堆存储器空间。这些异常从来不会由用户程序来抛出，因而也就不应该在用户程序之中处理，Exception具有两个系
620 统定义的直接后裔：RuntimeException和IOException。正如这两个名字所指示的，当在输入或输出操作中发生错误时，IOException被抛出；所有的输入与输出操作，都被定义为包java.io中定义的各种类方法。

还有预定义的RuntimeException的后裔类，在大部分情况下，当由用户程序引起错误时，RuntimeException则被Java虚拟机器抛出。例如，在java.util中所定义的ArrayIndexOutOfBoundsException，就是一个经常被抛出的异常，这个异常是RuntimeException的后裔。另外一个经常被抛出的RuntimeException的后裔异常是NullPointerException。

用户程序可以定义自己的异常类，Java中的约定为凡是用户定义的异常都是Exception的子类。

14.4.2 异常处理程序

Java的异常处理程序，与C++中的异常处理程序有着相同的形式，只是其中的每一个

catch都必须具有参数，而且这些参数的类型必须是预定义类Throwable的子类。

除了在14.4.6小节中所描述的finally子句外，Java中的try结构的语法，与C++中的完全相同。

14.4.3 异常与异常处理程序的绑定

抛出一个异常十分简单，只要将异常类的一个实例作为throw语句的操作数来给出。例如，假设我们定义一个被命名为MyException的异常，如下：

```
class MyException extends Exception {
  public MyException() {}
  public MyException(String 消息) {
    super (消息);
  }
}
```

能够使用下面的语句来抛出这个异常：

```
throw new MyException();
```

也可以将throw语句与创建throw的异常实例分开来进行，例如，

```
MyException myExceptionObject = new MyException();
...
throw myExceptionObject;
```

在我们的新的类中所包含的两个构造器，其中的一个不具有参数，而另外的一个则具有一个String对象的参数，这个参数被送给超类的Exception，并且由Exception来显示这个参数。因而我们的新异常可以通过下面的语句来抛出：

621

```
throw new MyException ("一条锁明错误出现的地址的相情");
```

在Java中，异常与异常处理程序的绑定没有C++中的那么复杂。如果一个异常是在try结构的复合语句之中被抛出的话，这个异常就与紧接在这个try子句后面的第一个异常处理程序（catch函数）相绑定。这个函数的参数与被抛出的对象具有相同的类或与被抛出的对象的祖先相同的类。如果发现了一个相匹配的处理程序，throw就与之绑定，并且执行这个处理程序。

可以在一个处理程序的尾部，包括不具有操作数的throw语句，这样就可以进行异常的处理，并在处理之后重新抛出异常。新抛出的异常不会在原来抛出它的同一条try子句中处理，因此循环将不是一个问题。重新抛出异常，通常是当需要一些局部的行为，然而异常又是由一个包含的try子句、或调用函数来处理时。处理程序中的一条throw语句可以抛出一个异常，而不是将控制转移给处理程序的这种异常。

为了确保try子句中所抛出的异常总能在一个方法中来处理，可以编写一个特殊的处理程序匹配从Exception中派生出来的所有的异常，这个处理程序的参数类型就是Exception，如，

```
catch (Exception genericObject) {
  ...
}
```

因为一个类总是与自身或任何祖先类相匹配，从Exception中派生的任何类都与Exception相匹配。当然，总是将这样的异常处理程序放置于处理程序序列的末尾，否则它将阻止使用，在同一个try结构中排在这个处理程序后面的处理程序。因为匹配处理程序的搜寻是顺序的，并且这种搜寻总是结束于发现了一个匹配处理程序之时。

14.4.4 其他设计选择

在执行一个程序时，Java 的运行时系统将程序中每一个对象的类的名称存储起来。可以使用getClass方法来得到一个存储类名称的对象，而类名称的本身则能够通过使用 getName方法来获得。因此我们就能够检索到来自throw语句的、引起处理器执行的实参的类的名称。对于上面的处理程序，可以这样来获得类名称：

624
```
genericObject.getClass().getName()
```

参数对象由构造器创建，可以通过下面的方式来获得与参数对象相关的消息：

```
genericObject.getMessage()
```

此外，在用户定义的异常情况下，所抛出的对象可以包括对于处理程序可能有用的任意数目的数据成员。

Java中的throws子句的外表以及它在程序中的放置，与C++中的throw说明的相类似。然而Java的throws子句的语义却完全不同于C++中的throw说明。

在一个Java方法的throws子句中出现的一个异常类的名称，说明这个方法可以抛出这个异常类或它的任何后裔异常类。例如，当一个方法说明它能够抛出IOException时，这意味着这个方法能够抛出IOException对象或者它的任何后裔类的对象，如EOFException 的对象，并且它不处理它抛出的异常。

类异常Error与RuntimeException以及它们的后裔，被称为**不检测的异常** (unchecked exception)。所有其他的异常都被称为**要检测的异常** (checked exception)。编译器从不关心不检测的异常。然而，编译器却必须保证能够被方法所抛出的要检测的异常，要么是在throws子句中出现，要么是在方法中被处理。Error和RuntimeException 类异常及其后裔不被检测的原因，是因为任何方法都可以抛出这些异常。可以有一个程序来捕捉不检测的异常，然而并不要求这样做。

正如C++的例子，在一个方法的throws子句中所声明的异常，不能够多于它所覆盖的方法中的throws子句中所声明的异常；当然，它所声明的异常可以少于它所覆盖的方法所声明的异常。一个方法能够抛出其throws子句中所列出的任何异常，连同这个异常的后裔类。

如果一个方法不直接抛出某种特定异常，但是调用另外一个可以抛出这个异常的方法，这个方法就必须将这个异常列入它的throws子句中。这就是在下面的示例中使用readLine方法的buildDict方法，必须在首部的throws子句中说明IOException的原因。

一个不包含throws子句的方法不能传播任何要检测的异常。在C++中，没有throws子句的函数抛出任何异常。

当一个方法调用另外一个方法，而后者将一个要检测的异常列在其throws子句中，那么这个方法就具有三种可能的方式来处理这个异常：第一，它能够捕捉这个异常并进行处理；第二，它能够捕捉这个异常并且抛出一个列于自己的throws子句中的异常；第三，它可以在自己的throws子句中声明这个异常，但并不进行处理。这实际上就是将异常传播到一个包含它的try子句中，如果这种try子句存在，或者，在不存在包含它的try子句的情况下，将异常传
625 播到调用它的方法中去。

Java中没有默认的异常处理程序，因而不可能禁用异常，Java中的继续与C++中的一样一样。

访谈



Java的诞生

JAMES GOSLING

James Gosling是Sun公司的资深研究员和副总裁，Java程序设计语言的领导人，也是计算机界最有名的程序员之一。他是1996年软件开发程序设计优秀奖的获奖者。Gosling开发了NeWS, Sun公司的网络可扩展视窗系统。他曾经是卡内基·梅隆大学的Andrew项目的主要成员。Gosling在卡内基·梅隆大学获得了计算机科学博士学位。

个人简史

问：你是怎样涉足到计算机领域的？

答：在我十四岁那年，一个朋友的父亲带我去参观Calgary大学的计算机设施，我就被吸引住了。可以说，第一眼我就喜欢上了计算机。然后我开始自学程序设计。读高中时在Calgary大学物理系得到一份工作。此后，从那里开始就象滚雪球一样滚到现在。

问：你的第一份计算机方面的工作是什么？

答：在Calgary大学物理系为ISIS-II卫星编写软件。

问：你最喜欢的工作是什么？

答：我的第一份工作很特殊，但我也很喜欢我现在的工作，在Sun公司的研究实验室作研究员。我正式的职位是"副总裁和资深研究员"。

JAVA：一种程序设计语言的诞生

问：许多语言的诞生都是创始人目标的副产品。我听说Java和"绿色项目"也是这样的。你能告诉我们这个故事吗？

答：我们一个小组当时正研究有可能对Sun公司产生影响的未来发展趋势。我们很快就将注意力集中在与网络相关的非计算机设备（如移动电话、电视、控制系统等）中的数码系统的使用上。我们开始建造了一个样机设备来学习这个领域。我们在使用的基本程序设计工具中遇到不少问题。我在那个项目中的工作最后变成解决工具的问题。最后的成果就是Java。

问：为什么要创造一种语言？

答：因为别的语言都不行。第一个样机是用C++写的。在那以前我们曾考虑过并尝试了许多其他的语言。它们在二进制码的层次上太依赖于CPU体系结构，并缺少考虑安全性。不用说，它们还有一些可靠性问题。

问：你把Simula说成是Java的先驱者。为什么是Simula？

答：它是最早的面向对象的语言。在过去的几年中，我与Simula用得很多。相对C和C++，它是单继承，并有垃圾的存储器模型。

问：哪些特征使得Java与那些受欢迎语言不一样？

答：效率、可靠性、安全性和可移植性。

问：这些特征和当前硬件软件的发展趋势对Java的成功起了多少作用？

答：很多。

JAVA：命名

问：为什么你原先的项目称为"绿色项目"？

答：没有什么好的理由。Apple有一个"橙色项目"，人们用颜色来命名该项目。我们项目的名字来自于我们工作的办公室盖房的门的颜色。那是一扇绿门。Java原来的名称为"橡树"。当在我决定这个名字时，我情正打着窗外，那时窗外长着一棵橡树，但是原来的那个名字与别人的商标有着种种冲突。律师们

就要我们找出一个没有问题的名字。

问：你已经厌倦了Java这个名字吗？

答：没有。

问：如果你今天能给Java取另外一个名字，你会给它什么名字？

答：这个问题太难了，挑选名字很难。

JAVA：过去、现在和将来

问：如果你能够回到过去并改变Java的两个特征，或者是重新来创建它们，那会是怎样的两个特征？

答：轻对象和switch语句。

问：你曾经思考过要进行这样的工作吗？

答：是的。

问：回头看看你们当年在“绿色项目”中所企图创建的东西，今天这样的设备还有吗？当年你们所走的路线正确吗？

答：是的。Palm Pilot和摩登、精巧的手机，正是我们当年的方向。

问：如果今天安排你去做类似于“绿色项目”的工作，你会将你的精力集中在什么样的用户功能上？或者说，如果互联网和Java是20世纪90年代初期的“伟大的”工具，下一个“伟大的”工具是什么？

答：下一个“伟大的”工具没有变——仍然是互联网。我们至今甚至还没有开始发掘它。

问：往前挑选15年，那时的程序设计语言会提供哪些它们现在所没有提供的功能？

答：推理和验证。

622 / 623

问：如果你没有从事现在的工作，你会做些什么？

答：任何涉及创建东西的事情。

14.4.5 一个示例

下面是Java的类，它具有与在14.3.5节中的C++程序同样的功能：

```
//分数的分布
//输入：一列表示分数的整数值，后跟一个负数
//输出：分数分布情况，分数等级(0-9,10-19,...,90-100)
import java.io.*;

// 处理数据属性的异常定义
class NegativeInputException extends Exception {
  public NegativeInputException() {
    System.out.println("End of input data reached");
  }  //** 结构体结束
}  //** NegativeInputException 类结束

class GradeDist {
  int newGrade,
      index,
       limit_1,
       limit_2;
  int [] freq = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

void buildDist() throws IOException {
  DataInputStream in = new DataInputStream(System.in);
  try {
    while (true) {
```

```
        System.out.println("Please input a grade");
        newGrade = Integer.parseInt(in.readLine());
        if (newGrade < 0)
          throw new NegativeInputException();
        index = newGrade / 10;
        try {
          freq[index]++;
        }  //** 为内 try 语句结束
        catch(ArrayIndexOutOfBoundsException) {
          if (newGrade == 100)
            freq [9]++;
          else
            System.out.println("Error - new grade: " +
                        newGrade + " is out of range");
        }  //** catch (ArrayIndex... 结束
      }    //** while (true) ... 结束                                    626
    }      //** 外部 try 语句结束
    catch(NegativeInputException) {
      System.out.println ("\nLimits    Frequency\n");
      for (index = 0; index < 10; index++) {
        limit_1 = 10 * index;
        limit_2 = limit_1 + 9;
        if (index == 9)
          limit_2 = 100;
        System.out.println("" + limit_1 + " - " +
                        limit_2 + "       " + freq [index]);
      }  //** for (index = 0; ... 结束
    }    //** catch (NegativeInputException ... 结束
  }      //** method buildHist 结束
```

在这个程序中定义了因为负输入而发生的异常NegativeInputException。当这个类的一个对象被抛出时，它的构造器显示出一条消息。它的处理程序产生方法的输出。ArrayIndexOutOfBoundsException是预定义的，并由Java虚拟机器抛出。在这两种情况下，处理程序都不在它的参数中包括对象的名称，因为在这两种情形下，对于任何目的，名称都是没有用的。注意，所有的处理器都是获取对象来作为参数，然而这些参数常常是没有什么用途的。

14.4.6 finally子句

在有些情况下必须执行一个过程，而不论一条try子句是否抛出了一个异常，也不论是否在一个方法中捕获了这个被抛出的异常。关于这种情形的一个例子，就是必须要关闭一个文件的情形。另外一个例子就是，如果一个方法占有某种外部资源，而不论这个方法的执行是怎样结束的，都必须释放这些资源。finally子句就是为这些类型的需要而设计的。finally子句被放置在try结构之后的处理程序列表的末尾。一般而言，try结构与finally子句的位置关系如下所示：

```
try {
  ...
}
catch (...) {
  ...
}
... //** 更多的处理程序
```

```
finally {
  ...
}
```

637

这种结构的语义为：如果try子句没有抛出一个异常，在执行try结构后的语句之前就执行finally子句；如果try子句抛出了一个异常，并且这个异常被try结构中的一个处理程序所捕获，在处理程序完成执行后，就执行finally子句；如果try子句抛出一个异常，但是这个异常没有被try结构中的任何处理程序所捕获，在异常被传播之前，执行finally子句。

一个没有异常处理程序的try结构，后面可以跟随一个finally子句。当然，这只有在复合语句中具有break、continue或return语句时才有意义。在这些情况下，它的目的与它和异常处理在一起使用时是相同的。例如，我们可以有下面的代码：

```
try {
  for (index = 0; index < 100; index++) {
    ...
    if (... ) {
      return;
    } //** 结束if语句
    ...
  } //** 结束for语句
}   //** 结束try子句
finally {
  ...
} //** 结束try结构
```

不论这里的控制是由return终止的还是正常结束的，都将执行finally子句。

14.4.7 断言

在第2章讨论Plankalkül语言时，我们曾经提到过，这种语言包含了断言。在Java 1.4版本中也增加了断言。然而在这个版本中的默认情况下，断言是停用的。[⊖]要使用断言，必须在执行程序时，使用enableassertions（或ea）选择来启用它们，例如：

```
java -enableassertions MyProgram
```

两种可能的assert语句的形式为：

```
assert 条件;
assert 条件:表达式;
```

628

如果是使用第一种形式，当执行达到assert语句时，条件就被检测。如果条件为真，什么也不进行；如果条件为假，就抛出AssertionError的异常。如果是使用第二种形式，系统的行为将是一样的，除了会将表达式的值作为字符串传递给AssertionError的构造程序，并成为程序除错而输出以外。

assert语句被用于防错性的程序设计。在一个程序中可以使用多条assert语句，以便确保程序的计算能够沿着正确轨道来产生正确的结果。很多程序即使在使用不具有断言的语言时，也在程序中加入这种检测以帮助除错。在程序经过了充分的检测后，就将这些检测删除。assert语句的作用与这些检测是一样的。assert语句所具有的优点是可以将它们停用，而不需要将它们语句一个个地从程序中删除。这样不但节省了删除这道工序，而且还可以在子程序的维护过程中再使用这些语句。

⊖ Java 5.0默认启动断言。

14.4.8 评估

Java中的异常处理机制以C++的版本为基础，并且进行了一些改进。

首先，一个C++的程序能够抛出任何定义于程序中的或者是由系统所定义的类型，而在Java中，只能够抛出Throwable及其后裔类的对象。这样就将能够被抛出的对象与所有其他存在于程序中的对象（及非对象）区别开来。如果一个异常只是引起一个int数值被抛出，那有什么意义？

其次，一个没有包括throws子句的C++程序单位能够抛出任何异常，但并不通知程序的读者。在Java中，一个没有包括throws子句的方法，不能够抛出任何它所不能够处理的要检测的异常。因此，Java的方法的读者能够从方法的首部知道它可以抛出哪些它所不能够处理的异常，虽然C++编译器忽略throws子句，但是Java编译器使方法能抛出的所有异常都列在throws子句中。

再次，所增加的finally子句在某些情形下能够提供极大的方便。它允许采取某些行动，而不论一条复合语句是如何终止的。

最后，Java的运行时系统隐式地抛出多种异常，例如，数组下标越界和存取空的引用变量，这些都能够被任何用户程序来处理。C++程序却只能够处理它所显式抛出的异常（或由它使用的库类抛出）。

Java的异常处理的功能对比Ada中的异常处理，它们基本上是相当的。Java方法中的throws子句有助于可读性，然而Ada则没有与这种子句相对应的特性。就允许程序处理系统检测的异常的方面而言，比较Ada与C++中的异常处理，Java肯定更加接近于Ada。 629

C#中所包括的异常处理的结构十分类似于Java中的，除了C#不具有throws子句之外。

14.5 事件处理概述

事件处理与异常处理相类似。在这两者中都是当某个事件（异常或者事件）发生时，处理器就被隐式地调用。所不同的是，异常是由用户程序显式地产生，或者是由硬件或软件解释器隐式地产生。而事件则是由外部行动来产生，例如，用户通过图形接口与程序的互动。本小节将介绍事件处理的基本概念。事件处理远没有异常处理那么复杂。

在传统的（非事件驱动的）程序设计中，代码本身就说明代码被执行的顺序，尽管执行顺序通常还受到程序输入数据的影响。在事件驱动的程序设计中，程序的许多部分是在完全不可预料的时刻被执行的。这些程序部分的执行，是由用户与正在执行的程序的互动所激发的。

本章所讨论的事件处理，都与图形用户接口有关。因此，大多数的事件都是通过图形对象（常常称为*widget*）的用户的互动而产生的。最常见的*widget*的例子就是按钮。对于用户通过图形接口所产生互动的反应是事件处理最常见的一种形式。

事件（event）就是通知某个特定的事情已经发生，例如，鼠标在按钮上的点击。严格来说，至少是在这里所讨论的事件处理中，一个事件是一个对象。它由运行时系统对于用户行动作出的反应来产生。

事件处理器（event handler）是对于事件作出响应时所执行的一段代码。事件处理器使得程序能够对于用户行动作出反应。

虽然事件驱动的程序设计在图形用户界面（GUI）出现之前很久就已经被使用了，但是只有在用于广泛流行的图形界面时，它才成为了一种广泛使用的程序设计方法学。例如，考虑显示给万维网浏览器用户的图形界面。现在许多给万维网浏览器用户显示的文档都是动态的。这

样的文档可以给用户显示订购表单。用户通过点击按钮来选择货物，在响应按钮的点击时所需的内部计算就由一个事件处理器来进行。这个事件处理器将对点击事件作出反应。

事件处理器的另外的一常见的应用是在表单更改或在表单被提交给服务器时，检测表单元素中的简单错误及遗漏。使用浏览器中的事件处理器来检测表单数据的有效性，可以节约将数据送往服务器的时间，此后，服务器中内层的程序或脚本将在数据被处理之前进行正确性的检
630 测。浏览器中的事件驱动的程序设计常常使用客户端的脚本语言，如JavaScript。

14.6 Java的事件处理

Java支持两种不同的方法，将交互显示呈现给用户或者应用程序，或者小应用程序applet。这两者都使用相同的类来定义GUI的成分，以及使用相同的事件处理器来提供交互。虽然我们在这里仅仅讨论applet，这一小节的内容也同样适用于应用程序。

Java最早期的版本提供了一种比较原始的方式，来支持GUI的成分。在Java的1.2版本中，加入了一组新的GUI成分，统称为Swing。

14.6.1 Java Swing的GUI成分

在javax.swing中定义的Swing包包括了一组成分。由于我们在这里的兴趣是事件处理，而不是GUI的成分，我们将仅仅讨论两种widgets——文字方框以及单选按钮。

一个文字方框是JTextField类的对象。最简单的JTextField构造器只接受一个参数，即按字符来计算的方框长度，例如：

```
JTextField name = new JTextField(32);
```

JTextField构造器也可以接受一个文字串作为可选的第一个参数。如果存在这个文字串参数，它将显示作为文本框的最初内容。

单选按钮是放置在按钮组中的一种特殊的按钮，一个按钮组是ButtonGroup类的对象，它的构造器不接受任何参数。在一个单选按钮组中，一次只能按一个按钮，如果任何一个按钮被按下，先前按下的按钮就被隐式地放开。JRadioButton构造器被用来产生单选按钮，这个构造器接受两个参数：标号以及按钮的初始状态（true或false，按下或未按下）。在产生一个单选按钮后，使用按钮组对象的add方法，将这个按钮放入到按钮组中去。考虑下面的例子：

```
ButtonGroup payment = new ButtonGroup();
JRadioButton box1 = new JRadioButton("Visa", true);
JRadioButton box2 = new JRadioButton("Master Charge",
                                     false);
JRadioButton box3 = new JRadioButton("Discover", false);
payment.add(box1);
payment.add(box2);
payment.add(box3);
```

一个applet的显示实际上是一个多层次结构的框架(FRAME)。我们只是对于这些层次之一
631 感兴趣，即内容板面。内容板面是放置applet的输出的地方。用户程序则不直接将任何东西放在内容板面中；相反，它们是将图形对象放在一个板面中，然后再将这个板面加入到内容板面上。在应用程序的情况下，则是先产生一个框架，然后再将构造出的板面加入到这个框架的内容板面上。

通过使用getContentPane方法将一个内容板面创建成为Container类的对象，如下面的语句

```
Container contentPane = getContentPane();
```

可以将预定义的图形对象，如GUI的成分，直接加入到applet中所产生的板面上，然后再加入到applet的内容板面上。下面的语句产生一个板面对象。在我们后面的讨论中将会使用到它。

```
JPanel myPanel = new JPanel();
```

在使用构造器产生了GUI成分后，必须再使用add方法将这些成份放入板面之上：

```
myPanel.add(button1);
```

14.6.2 Java事件模型

用户与GUI成分的交互产生可以由事件处理器来捕捉的事件。事件处理器提供所需要的计算，这些GUI成分被认为是事件产生器。它们都产生事件。在Java中，将事件处理器称为事件监听器（event listener）。事件监听器通过注册事件监听器与事件产生器相连接。在本小节后面我们将会读到，一个实现监听器接口类的方法，使用这个方法来完成监听器的注册。包含GUI成分的板面对象，可以是这些GUI成分的事件监听器，当发生一个事件时，仅仅会通知那些与这个事件注册了的事件监听器。

事件产生器通过传送消息给事件监听器（调用监听器的方法）来通知事件监听器所发生的事件。接收消息的监听器方法将实现事件处理器。使用一个接口来使事件处理器方法与标准协议相一致。一个接口体现标准的方法协议，但是并不提供这些方法的实现。强迫事件产生器为一个类的子类，从而将从这个类继承一个标准协议，这样就可以说明这个协议，然而，JApplet类已经有了一个超类。在Java中，一个类只能具有一个父类。因此，协议只能来自一个接口，除非一个类是依它所实现的接口中的所有方法的定义，它才能被范例化。

一个需要实现监听器的类必须实现这些监听器的接口。在Java中有许多的事件类和监听器接口。 652

一个事件类是ItemEvent，它与选择复选框、单选按钮或列表项的事件相关联。ItemListener接口指定了itemStateChanged方法，它处理ItemEvent事件。因此，为了提供单选按钮动作触发的动作，必须实现的接口ItemListener需要定义方法itemStateChanged。

如前所述，GUI成分与事件监听器的连接，是通过一个实现监听器接口的类方法来完成的。例如，因为ActionEvent是用户在按钮上的行动所产生的事件对象的类名，addActionListener方法就被用来将一个监听器注册到按钮上。可以在一个面板中实现在applet的面板中产生的按钮事件的监听器。因此如果面板Mypanel实现ActionEvent按钮事件处理器，button1就是这个面板中的按钮，我们可以用下面的语句来注册监听器：

```
button1.addItemListener(this);
```

每一个事件处理器方法接受一个事件参数，这个参数提供关于事件的信息。使用事件类的方法，如getState，用来访问这条信息。例如，当通过一个单选按钮来调用getState时，根据按钮是开或关的状态（按下或未按下），getState将返回true或false。

所有与事件有关的类都被包括在java.awt.event包中。所以，任何使用事件的applet通常都会输入这个包。

下面是一个名称为Radio8的示例applet。我们用它来说明如何使用事件和事件处理器来显示applet中的动态内容。这个applet生成一些单选按钮来控制一个文字域内容的字体。对于各种字体中的每一种，它都将产生一个Font对象。每一个Font对象都具有一个单选按钮用于选择相关的字体。这个applet然后会产生一个文字串。用户将使用按钮来控制字体。这里的事件处理

器是itemStateChanged。在接到ItemEvent对象的关于按钮状态变化的通知以后，事件处理器将确定被按下的是哪一个按钮，然后它将设定相对应的字体。

```
/* RadioB的示例applet如何使用事件
   和事件处理器来显示applet中
   的动态内容
   */
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;

public class RadioB extends JApplet implements
                                        ItemListener {

// 将大多数变量定义为受变量。
// 从而使发生和事件处理器对其可见

    private Container contentPane = getContentPane();
    private JTextField text;
    private Font plainFont, boldFont, italicFont,
                 boldItalicFont;
    private JRadioButton plain, bold, italic, boldItalic;
    private ButtonGroup radioButtons = new ButtonGroup();
    private JPanel myPanel = new JPanel();

// 在开始处构建init方法

    public void init() {

// 设置面板的背景色

        myPanel.setBackground(Color.cyan);

// 创建字体

        plainFont = new Font("Serif", Font.PLAIN, 16);
        boldFont = new Font("Serif", Font.BOLD, 16);
        italicFont = new Font("Serif", Font.ITALIC, 16);
        boldItalicFont = new Font("Serif", Font.BOLD +
                                  Font.ITALIC, 16);

// 创建测试文字串，设置其字体。
// 并将其添加到面板上

        text = new JTextField(
             "In what font style should I appear?", 30);
        myPanel.add(text);
        text.setFont(plainFont);
// 创建字体并连接钮片
// 添加到面板上

        plain = new JRadioButton("Plain", true);
        bold = new JRadioButton("Bold");
        italic = new JRadioButton("Italic");
        boldItalic = new JRadioButton("Bold Italic");
```

634

```
        radioButtons.add(plain);
        radioButtons.add(bold);
        radioButtons.add(italic);
        radioButtons.add(boldItalic);

// 记录事件处理器将对myPanel的处理

        plain.addItemListener(this);
        bold.addItemListener(this);
        italic.addItemListener(this);
        boldItalic.addItemListener(this);

// 在面板上放置按钮

        myPanel.add(plain);
        myPanel.add(bold);
        myPanel.add(italic);
        myPanel.add(boldItalic);

// 为applet添加面板到其内容窗格

        contentPane.add(myPanel);

    } // 结束 init()

// handler 事件

    public void itemStateChanged (ItemEvent e) {

// 判定当前的活动按钮并
// 设置其字体

        if (plain.isSelected())
           text.setFont(plainFont);
        else if (bold.isSelected())
           text.setFont(boldFont);
        else if (italic.isSelected())
           text.setFont(italicFont);
        else if (boldItalic.isSelected())
           text.setFont(boldItalicFont);

    } // 结束 itemStateChanged

} // 结束 RadioB applet
```

635

图14-2是RadioB applet所产生的屏幕结果显示。

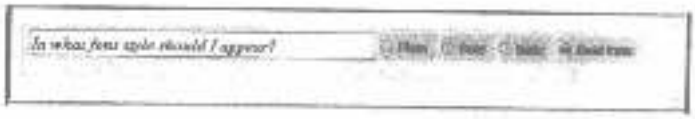


图14-2 RadioB applet的输出

小结

自20世纪70年代中期以来设计的许多现代语言都包含了异常处理的设施。然而时至今日已经有好

几乎广泛应用的语言包括了异常处理。

Ada提供了完善的异常处理设施以及一组小而全的内建的异常。异常处理程序位于程序实体之上。虽然在没有局部处理程序可用的条件下，会隐式或显式地将异常传播给其他的程序实体。

C++中不包括预定义的异常（除了定义在标准程序库中的那些异常而外）。C++中的异常为派生类型的对象。预定义的类的对象或用户定义的类的对象。C++中不包括内建的异常。异常与异常处理程序的绑定，是通过将throw语句中表达式的类型与处理程序形参的类型相匹配。所有的处理程序都有着相同的名字，即catch。一个C++方法的throw子句列举这个方法所抛出的异常的类型。

Java中的异常是对象，它的祖先必须可以回溯到Throwable类的一个后裔类。Java具有两个类型的异常，即要检测的异常与不检测的异常。要检测的异常是用户程序及编译器所关心的异常。不检测的异常能够发生在任何位置，并且常常被用户程序所忽略。

Java中一个方法的throws子句列举它所抛出但并不处理的要检测的异常。throws子句还必须包括它所调用的方法可能抛出的并传播到它的调用程序的异常。

Java的finally子句提供了一种机制来保证一些代码的执行，而不论一个try结构中的复合语句的执行是怎样结束的。

Java现在还包括了assert语句，用以进行防错性的程序设计。

一个事件通知某一体需要处理的事件的发生。事件常常是在用户与程序交互时通常是通过图形用户接口时所产生的。Java中的事件处理器被称为监听器。如果一个事件发生时需要通知它的监听器，就必须将这个监听器注册到这个事件之上。Java中最常用的监听器是actionPerformed和itemStateChanged，这两个监听器的特定由相关的接口来指定。

636

文献注释

由Goodenough所著的（Goodenough，1975）文献是讨论异常处理但又没有局限于某种特定语言的最好的论文之一。MacLaren（1977）描述了PL/I中的异常处理的设计问题。Liskov和Snyder（1979）清楚地描述了CLU的异常处理的设计。ARM（1995）描述了Ada语言的异常处理设施。Romanovsky和Sander对其进行了批判性的评价。Stroustrup（1997）描述了C++中的异常处理。Campione et al.（2001）描述了关于Java的异常处理。

复习题

1. 定义异常、异常处理程序、抛出异常、禁用异常、继续以及内建异常。
2. 异常处理的设计问题是什么？
3. 异常与异常处理程序相绑定意味着什么？
4. 在Ada中的异常的可能框架是什么？
5. Ada中没有被处理的异常将被传播到哪里，如果这个异常是在一个子程序中被抛出的？是在一个块中被提出的？是在一个包体中被提出的？是在一个任务中被提出的？
6. 在Ada中，一个异常被处理之后，执行将从何处继续？
7. 在Ada中，怎样显式地提出一个异常？
8. 在Ada中，怎样来定义一个用户定义的异常？
9. 在Ada中，怎样禁用一个异常？
10. C++中所有的异常处理器的名称是什么？
11. 在C++中，怎样显式地提出一个异常？
12. 在C++中，怎样将异常与异常处理程序相绑定？
13. 在C++中，怎样编写一个能够处理任何异常的异常处理程序？
14. 在C++中，当一个异常处理程序的执行完成后，执行控制将被转移到哪里？

15. C++包括了内建的异常吗？
16. 所有Java的异常类的基类是什么？
17. 在Java中，大多数的用户定义的异常类的父类是什么？
18. 如何在Java中编写一个能够处理任何异常的异常处理程序？ 637
19. C++中的throw说明与Java中的throws子句之间有什么区别？
20. 在Java中，要检测的异常与不检测的异常之间有什么差别？
21. 你能够禁用一个Java中的异常吗？
22. Java中的finally子句的目的是什么？

练习题

1. 什么是运行时错误或状态？如果这种运行时错误或状态存在的话，Pascal程序能够检测及处理吗？
2. 从PL/I或Ada程序设计的教科书中查找关于它们的内建异常的相对应的部分。从完整性与灵活性的两个方面，对于这两者做出比较性的评估。
3. 根据（ARM，1995）来确定如何处理在会合期间发生的异常。
4. 根据一本COBOL的教科书来确定，在COBOL程序中怎样进行异常处理。
5. 在不具有异常处理设施的语言中，通常是让大多数子程序包括一个“出错”参数，以这个参数的某个值来表示“OK”，而以某个其他的值来表示“过程中有错”。相对于这种方法，一种语言中的异常处理设施，如Ada语言中的，具有什么优越性？
6. 在不具有异常处理设施的语言中，我们可以将一个错误处理过程，作为参数传送给每一个可能检测出需要处理的错误的过程。这种方法有什么缺点？
7. 比较在练习题5和练习题6中所给出的方法，你认为哪一种方法更好，为什么？
8. 比较C++中的和Ada中的异常处理设施。依你之见，哪一种设计更为灵活？哪一种能够写出更为可靠的程序？
9. 考虑下面的Java程序框架：

```
class Big {
  int i;
  float f;
void fun1() throw int {
  ...
  try {
    ...
    throw i;                                    638
    ...
    throw f;
    ...
  }
  catch(float) { ... }
    ...
}
class Small {
  int j;
  float g;
  void fun2() throws float {
    ...
    try {
    ...
       try {
         Big.fun1();
         ...
```

```
        throw 3;
        ...
        throw 9;
        ...
      }
    catch(int) { ... }
    ...
    }
  catch(float) { ... }
}
```

生这只条throw语句中的每一条，在什么位置处理异常？请注意，fun1被类Small中的fun2所调用。

10. 写出C++和Java中的异常处理功能的详细比较报告。

11. 概述有关于继续的终止和重新开始模型的参数。

程序设计练习题

1. 编写一段Ada代码，实现新尝试对于过程Tape_Read的调用，过程Tape_Read从磁带机读入数据，并且可以提出Tape_Read_Error异常。

2. 假设你正在编写一个Ada过程，存在着三种不同方案来完成这个过程的要求，编写出这个过程的一个框架，使得当第一种方案提出任何异常时，就试行第二个方案，而在第二个方案提出任何异常时，就执行第三个方案。假设这三种方案已经分别被命名为ALT1、ALT2以及ALT3。

639

3. 编写一个Ada程序，从键盘输入一组从-100到100的整数值，并且计算所输入值的平方和。这个程序必须使用异常处理，从而保证输入值是在范围之内并都是合法的整数，当所计算的平方和大于标准的Integer变量所能够存储的值时，处理这个错误，并且检测文件结尾的状态以及使用这种状态来产生结果输出，在平方和溢出的情况下，显示出错信息并终止程序。

4. 为练习题3中的说明编写一个C++程序。

640

5. 为练习题3中的说明编写一个Java程序。

第15章　函数式程序设计语言

本章介绍函数式程序设计以及一些为用函数式程序设计来进行软件开发而设计的语言。因为这些语言以数学函数为基础，所以我们首先复习这些函数的基本概念。接着，介绍函数式程序设计语言的概念以及第一种函数语言LISP，它的链表数据结构和基于Lambda标记的函数语法。接下来的较长的一节用来介绍Scheme，包括它的一些原始函数，特殊的形式，函数的形式和用Scheme写的简单函数的一些示例。然后我们对COMMON LISP、ML和Haskell作简单介绍。接下来的一节描述函数程序设计语言的一些应用。最后，我们将函数式语言和命令式语言进行简短比较。

15.1　概述

本书的前14章主要是关于命令式程序设计语言和面向对象的程序设计语言的。除了Smalltalk外，我们所讨论的面向对象程序设计语言都类似于命令式语言。

命令式语言之间的高度类似的部分来自于它们共同的设计基础之一：冯·诺依曼体系结构，如同我们曾经在第1章中讨论的，我们可以整体地将命令式语言视为在Fortran I的基本模式上的改进与发展。所有的命令式语言都被设计来高效率地使用冯·诺依曼体系结构的计算机。虽然命令式风格的程序设计已经被大多数的程序人员所接受，但许多人认为它对于冯·诺依曼体系结构的高度依赖性，是对软件开发过程的不必要的限制。

除了命令式的以外，还存在其他的语言设计的基础。其中的一些更着眼于特殊的程序设计范型或者是方法学，而不是在某一种特定的计算机体系结构上的高效率执行。然而迄今为止，只有很少一部分的程序是采用非命令式语言来编写的。

以数学函数为基础的函数式程序设计，是最重要的非命令式语言风格的设计基础之一。这种风格的程序设计为函数式程序设计语言所支持。

LISP起初是一种纯粹的函数式语言，但很快就获得了一些重要的命令式的特性，以增进它的执行效率。它仍然是最重要的函数式语言，至少它是唯一的一种得到广泛使用的函数式语言。Scheme是LISP的一个很小型的静态作用域的方言。COMMON LISP是20世纪80年代初期的几种LISP方言的一种混合产品。ML和Haskell是强类型的函数式语言，它们具有比LISP和Scheme更加传统的语法。

1977年的图灵奖颁发给了John Backus，以奖励他在Fortran语言开发中的贡献。每一个获奖人员都在颁奖仪式上进行一个演讲，这篇演讲后来发表在ACM的Communications杂志上。在他的演讲中，Backus认为函数式语言比命令式的程序设计语言更好。原因是使用函数式语言所编写的程序可读性更好也更可靠，并且正确性可能也更好。他的论据是由于函数式程序中的表达式以及函数不存在副作用，它们的语义也与上下文无关，因而在开发中以及开发后，纯函数式程序更容易被人们所理解。

641
}
642

在这篇演讲中，Backus提出了一种纯函数式语言FP（functional programming），用来作为讨论他的论点的框架。后来，至少在获得广泛使用这一点上，这种语言没有成功。但是他的观点推动了对于函数式语言的讨论和研究。一些有名的计算机科学家都试图推进这样的概念，那就

是函数式语言比传统的命令式语言更好。然而，结果似乎并不理想。

本章的目的之一是通过描述Scheme语言中的核心部分来介绍函数式程序设计。虽然我们有意识地省略了Scheme语言中的一些命令式的特征，但我们仍然有充足的材料使读者能够编写出一些简单而又有趣的程序。缺乏实际编程经验的人会感觉到自己很难对函数式程序设计有彻底地理解，所以我们极力建议大家动起手来。

15.2 数学函数

数学函数是一个集合中的成员到另一个集合中成员的映射，前一个集合被称为定义域集（domain set），后一个集合被称为值域集（range set）。一个函数定义显式或隐式地说明定义域集、值域集和映射。在某些情况下，将映射描述为一个表达式或一个表格。通常将函数作用于定义域集（作为函数的参数）中的特定的元素之上。请注意，一个定义域集可能是多个集合的叉积（它反映有多个参数）。一个函数产生或是返回一个值域集中的元素。

数学函数的基本特性之一，是由递归和条件表示式所控制的。它们的映射表达式的求值次序，而不是命令式程序设计语言中常用的顺序以及循环的重复。

因为它们没有副作用，所以数学函数另外的一个重要的特性是，当给定一组相同的自变量
643 时，它们总是定义相同的值。[⊖]程序设计语言中的副作用，与模拟存储单元的变量相关联。

在数学上没有什么东西可以模拟存储单元，命令式语言函数中的局部变量保存函数的状态。在数学上，没有函数状态这一概念。

一个数学函数定义一个值，而不是说明从内存中的值生产另一个值的运算序列。这里没有命令式语言中的变量，因此不具有任何副作用。

15.2.1 简单函数

通常将函数定义写为一个函数名，后面跟随圆括号中的一串参数，再跟随着映射表达式。例如：

cube(x) ≡ x*x*x，这里的 x 是一个实数

在这个定义中，定义域集和值域集都是实数，符号≡意为“定义为”。参数x能够表示定义域集中的任何成员，但是在函数表达式的求值期间，则表示的是一个特定的元素。这是数学函数的参数与命令式语言中的变量之间的不同。

将函数的应用说明为一个函数名与定义域集中的一个特定的元素的对。将定义域集的元素代入函数映射表达式的参数，来进行表达式的求值，将获得值域集中的元素。举例来说，cube(2.0)产生值8.0。在这里，一个值得注意的重要问题是：在求值期间，一个函数映射所包含的参数必须是绑定的；一个绑定的参数是一个特定数值的名字。一个参数的每一次出现都被绑定于定义域集中的一个数值之上，并且在求值期间被考虑成为一个常量。

早期函数的理论研究将定义函数与给函数取名的任务区别开来。由Alonzo Church所设计的Lambda标记（Church，1941）提供了一种定义无名函数的方法。Lambda表达式说明一个函数中的参数以及函数的映射。Lambda表达式就是无名函数的本身。例如，考虑函数：

λ(x)x×x×x

如前所述，在求值以前，一个参数表示定义域集中的任何成员，但是在求值的期间，它将

⊖ 注意，数学函数定义值，而程序设计语言函数产生值。

绑定于一个特定的成员。当一个Lambda表达式对一个给定的参数求值时，我们就称之为：表达式被应用到那个参数之上。这种应用机制对于任何函数求值都是相同的。在下面的例子中，Lambda表达式的应用，就应该表示为：

(λ(x)x*x*x)(2)

所产生的结果是数值8。

像其他的函数定义一样，Lambda 表达式可以具有多个参数。 644

15.2.2 函数形式

一个高阶函数（或函数形式）是一个可以取函数为参数或产生一个函数为结果或者两者皆有的函数。一种常见的函数形式是函数复合（function composition），它有两个函数参数，并且产生一个函数，这个结果函数的值是将第一个实参函数应用到第二个函数的结果上而产生的。复合函数被写成一个表达式，使用°来作为操作符，如：

$h \equiv f \circ g$

举例来说，如果

$f(x) \equiv x + 2$

$g(x) \equiv 3 * x$

那么h就被定义成为

$h(x) \equiv f(g(x))$或者$h(x) \equiv (3 * x) + 2$

应用到所有参数（apply-to-all）是一种函数形式，它取一个函数来作为参数。如果是在一组自变量上的应用，“应用到所有参数”就是将函数参数应用到自变量表中的每一个值，而且将结果收集在一个表或一个序列中。符号α被用来表示“应用到所有参数”。考虑下面的示例：

设

$h(x) \equiv x * x$

那么

$\alpha(h, (2, 3, 4))$将产生 (4, 9, 16)

还有许多其他的函数形式，但以上两例说明了它们的特征。

15.3 函数式程序设计语言的基础

函数式程序设计语言的设计目的是尽可能最好地模仿数学函数。这个目的产生了一种根本不同于命令式语言所采用的解决问题的方式。在命令式语言中，对于一个表达式求值，然后将结果存储在一个存储单元中。在程序中，这个单元被表示为变量。对存储单元需求的注意，产生了一种相对低层次的程序设计方法学。在汇编语言的程序中，就经常必须存储表达式的部分求值的结果。例如，计算表达式

$(x + y)/(a - b)$

首先计算 $(x + y)$ 的值。然后，当计算 $(a - b)$ 的值时，必须将前面的值存储起来。在高级语言中，是由编译器来处理表达式求值中间结果的存储。现在，仍然需要存储中间的结果，但是将细节对于程序人员隐藏了起来。 645

纯函数式程序设计语言不使用变量或赋值语句。这使得程序人员不必关心程序所运行的计

算机存储单元。然而，如果没有变量，循环结构是不可能的，因为循环结构是通过变量控制的。因而，不使用循环，就必须使用递归来实现重复。一个程序就是函数定义以及函数应用的说明；一个程序的执行就是对于函数应用的求值。如果没有变量，纯函数式语言程序的执行就不存在在操作语义和指称语义中的状态。当给予同样的参数的时候，函数的执行总是会产生同样的结果。这种特征称为引用透明性（referential transparency）。这种特征使得纯函数式语言的语义比命令式语言的语义（包括命令式特性的函数式语言的语义）简单得多。

函数式语言提供一组原始的函数，一组用来从那些原始函数构造复杂函数的函数形式，一种函数应用的操作，以及某种或某些表示数据的结构。使用这些结构来表示参数以及函数计算的值。一个良好定义的函数式语言只需要很少的原始函数。

虽然函数式语言经常是使用解释器来实现，但是也能够被编译。

命令式语言通常只提供对于函数式程序设计的有限支持。使用一种命令式语言来进行函数式程序设计的最严重的缺点，是在命令式语言中对于函数返回值类型上的限制。在一些语言，如Fortran语言中，仅仅能够返回标量类型的数值。尤其重要的是，命令式语言通常不能够返回一个函数。这种限制局限了所能够提供的函数形式的种类。带有函数的命令式语言的另外一个严重问题是具有函数副作用的可能性正如我们在第7章中看到的，函数式副作用降低了代码可读性和表达式的可靠性。

15.4 第一种函数式程序设计语言：LISP

人们已经开发了多种函数式程序设计语言。一种最古老并且最为广泛使用的语言是LISP。通过LISP语言来学习函数式语言，类似于通过Fortran语言来学习命令式语言：LISP是第一种函数式语言，但是一些人现在相信，虽然它在过去40年来不断发展，但它已经不再代表函数式语言的最新的设计概念。此外，除了它的第一个版本以外，所有的LISP方言都包括命令式语言的特性，例如命令式风格的变量、赋值语句和循环（命令式风格的变量被用来命名存储单元，能够在程序执行期间多次地改变这些变量的值）。尽管有着一些奇怪的形式，LISP的后代语言还是很好地表示了函数式程序设计的基本概念，并因此而具有研究价值。

646

15.4.1 数据类型和结构

在最初的LISP语言中，仅有两种类型的数据对象：原子和链表[⊖]。LISP中类型的含义与命令式语言中类型的含义不同。事实上，最初的LISP是一个无类型的语言。原子要么是以标识符形式存在的符号，要么是数值常量。

回忆在第2章里我们曾经读到，因为将链表认为是表处理中的一个核心部分，所以LISP在早期使用链表作为它的数据结构。然而最终发展的结果，LISP却很少需要插入和删除的链表操作。

LISP中链表的说明是通过将它们的元素放在圆括号内。简单链表的元素被限制为原子，如

```
(A B C D)
```

嵌套的链表结构也使用圆括号来指定，如链表

```
(A (B C) D (E (F G)))
```

是一个具有四个元素的链表。第一个元素是原子A；第二个是子表(B C)；第三个是原子D；第四个是子表(E (F G))，而它的第二个元素是子表(F G)。

⊖ 实际上，链表是一种更为一般的数据结构——点对的一种最通用的形式。我们将不在这里讨论点对，只是参考在15.5.8节中给予回顾的介绍。

在内部，链表通常被存储为单向链表结构，其中的每个节点表示一个元素并具有两个指针。原子节点的第一个指针指向原子的某一种表示，例如，它的符号或数字值。子表元素节点的第一个指针指向子表的第一节点。在这两种情况下，节点的第二个指针都指向链表的下一个元素。一个链表通过指向它第一个元素的指针来引用。

图15-1中显示了我们两个示例链表的内部表示。请注意，其中链表的元素被水平地显示。链表的最后一个元素没有后继，因而它的链接为NIL。使用相同的结构来显示子表。

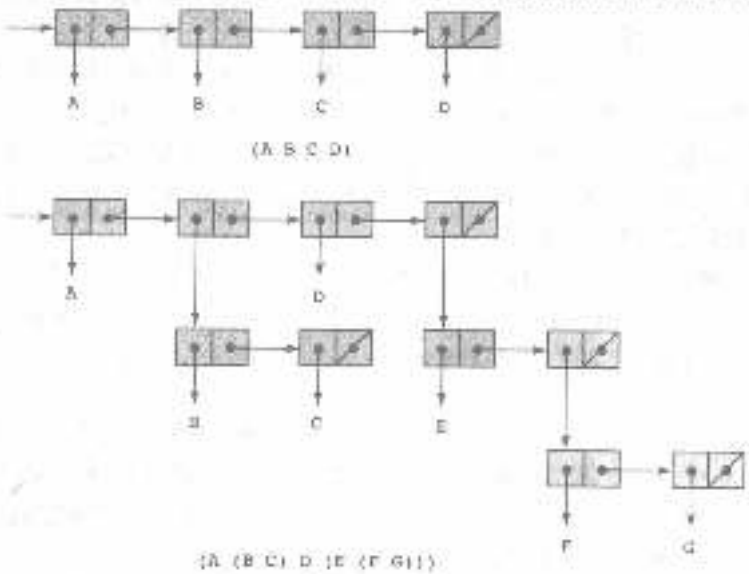


图15-1 两个LISP链表的内部表示

647

15.4.2 第一个LISP解释器

LISP设计者的最初意图是使LISP的程序标记法尽可能地接近Fortran语言的，在必要时再加入一些部分。这种标记被称为M标记，意为元标记。编译器会将使用M标记写的程序翻译为语义上相同的IBM 704机器码程序。

在LISP发展的早期，McCarthy决定写一篇文章来将表处理提升为一种一般符号处理的方式。在其中，McCarthy相信可以使用表处理来研究可计算性；当时通常是使用图灵机来研究可计算性。McCarthy认为符号表的处理是比图灵机更自然的计算模型。研究计算中的通常的一种需求，是必须证明完整类计算模型的可计算性特征，无论这个类使用的是什么计算模型。在图灵机模型的情况下，就可以构造一个模仿任何其他图灵机操作的通用图灵机。从这个概念我们得到了一种思想，就是构造一个通用的LISP函数，来对LISP中的任何其他函数求值。

对于通用LISP函数的第一个要求，是允许使用和表示数据同样的方式来表示函数的一种标记法。在15.4.1节中所描述的加括号的链表标记法，当时就已经被采用来表示LISP中的数据。因此人们决定发明，能够表示为链表的函数定义，以及函数调用的规则，可以将函数调用说明为，一种称为剑桥波兰表达式（Cambridge Polish）的前缀链表形式。如：

648

```
(函数名 参数1, …, 参数n)
```

例如，如果+是一个具有两个数字参数的函数，

```
(+ 5 7)
```

计算值为12。

在15.2.1节中曾经描述的Lambda标记号被选择来说明函数定义。然而，必须将它进行修改，从而允许名字与函数的绑定，以便函数能被其他的函数或自身所引用。这个名字的绑定由一个链表来说明，而这个链表则由一个函数名和包含Lambda表达式的一个链表所组成，如：

(函数名 (LAMBDA (参数1，…，参数n) 表达式))

如果你过去没有接触过函数式程序设计，也许会对一个无名函数感到很奇怪。然而，在函数式程序设计中以及在数学上，无名函数有时也会有用。例如，考虑一个函数，它的行动是要产生一个即刻应用于一个参数链表的函数。这个将被产生的函数就没有具有名字的必要，因为它将只被应用于它被构造的位置。在15.5.10节中将给出这样的一个示例。

采用这种新的标记说明的LISP函数，被称为S-表达式，也即符号表达式。最后，将所有的LISP结构，数据和代码，都称为S-表达式。一个S-表达式，不是一个链表就是一个原子。我们通常会简单地称S-表达式为表达式。

McCarthy成功地开发了一个可以计算任何其他函数的通用函数。这个函数被命名为EVAL，并且它本身就被表示为一种表达式的形式。人工智能项目中的两个人物Stephen B. Russell和Daniel J. Edwards，注意到可以将EVAL的实现用作LISP解释器，他们立即就创建了这样的一种实现（McCarthy et al., 1965）。

这个很快速的、容易的、料想不到的实现有着一些重要的结果。首先，所有早期的LISP系统都复制了EVAL，因此这些系统都是解释性的。其次，原来计划将M-标记的定义用作LISP程序设计的标记，然而却从未被完成或实现，因此S-表达式就成为了LISP中的唯一标记。使用相同的标记来表示数据和代码有重要的结果，其中之一，我们将在15.5.12小节中进行讨论。第三，最初语言设计中的很多东西实际上被冻结了起来，然而却将一些奇怪的特性留在了语言之中，例如，条件表示式的形式和使用一对空括号()来表示空链表和逻辑的假。

早期LISP系统的另外一个显然是意外的特性就是动态作用域的使用。函数在它们的调用者的环境中被求值。在那时，没有什么人知道有关作用域的事，而且对于做出的选择也没有过多的考虑。在1975年以前LISP中的大多数方言都使用动态作用域。而当代的方言或者使用静态作用域，或者允许程序人员在静态作用域和动态作用域之间进行选择。

649

15.5 Scheme概述

在这一节中，我们将描述Scheme的一部分（Dybvig, 2003）。我们选择Scheme是因为它相对简单，并且流行于学院和大学中，另外Scheme的解释器对于许多计算机都是现成可得的。在这一节中所描述的版本是Scheme 4。请注意，本节仅仅介绍Scheme中很小的一个部分，而且我们将不介绍Scheme的命令式特征。

15.5.1 Scheme的起源

Scheme语言是LISP的一支，于20世纪70年代的中期起源于MIT（Sussman and Steele, 1975）。它的特点是规模小、仅使用静态作用域和它是将函数作为第一类实体来对待。作为第一类实体，Scheme中的函数可能是表达式的值、以及链表中的元素，而且能够将它们赋给变量，以及它们可以被作为参数传递。LISP的早期版本不提供所有的这些功能。

作为一种具有简单语法和语义的小型语言，Scheme十分适合于教育应用。例如，函数式程序设计课程以及程序设计概论课程。

注意，在下面一节中的大部分Scheme函数只需要稍作修改，就能够成为LISP函数。

15.5.2 Scheme解释器

Scheme解释器是一种"读一计算一写"的无限循环。它重复地读用户键入的表达式（以链表的形式），解释这个表达式，然后显示所产生的值。表达式由函数EVAL来解释。文字常数的结果就是它们的自身。因此，如果你给解释器键入一个数字，它就仅仅是显示这个数字。调用原始函数的表达式是以下面的方式来进行求值的：首先，对于每一个参数表达式求值，并且无所谓次序。然后将原始函数应用到参数值，而且将所产生的值显示出来。

15.5.3 原始数值函数

在本小节中，将讨论仅仅处理数值原子而不处理符号原子和链表的Scheme原始函数。

Scheme包括基本算术运算的原始函数，它们是+、-、*和/，即加法、减法、乘法和除法。*和+运算能够具有零个或多个参数。如果*运算没有参数，它将返回1；如果+运算没有参数，它将返回0。+运算将它的全部参数相加起来。*运算则将它的所有的参数相乘。/和-运算能够具有两个或更多的参数。在减法的情况下，除了第一个参数外的所有的参数，都被从第一个参数中减去。除法与减法类似。示例如下： 650

```
      表达式          值
42                    42
(* 3 7)               21
(+ 5 7 8)             20
(- 5 6)               -1
(- 15 7 2)             6
(-24 (* 4 3))         12
```

如果参数值不为负，SQRT将返回它的数值参数的平方根。

15.5.4 定义函数

Scheme程序是一个函数定义的集合，因而定义函数是编写哪怕是最简单的程序的先决条件。回忆在15.2.1小节中曾经提到的，Scheme函数形式基于的是Lambda标记法。在Scheme中的一个无名函数实际上包括了LAMBDA这个字，并被称为**lambda表达式**（Lambda expression）。例如：

```
(LAMBDA (x) (* x x))
```

是一个无名函数，它将返回所给定数值参数的平方值。可以像命名函数一样来应用这个函数：即，将它放置在包含实参的链表的前面。例如，我们可以有：

```
((LAMBDA (x) (* x x)) 7)
```

它将得到值49。在这里，x被称为Lambda表达式中的**绑定变量**。当lambda表达式开始求值时，如果将一个变量绑定到一个实参数值，之后这个绑定变量就不会在表达式中被改变。

特殊形式的Scheme函数DEFINE服务于Scheme程序设计的两个基本需求：将一个名字绑定到一个值上，以及将一个名字绑定到一个lambda表达式上。第一种使用可能听起来像是能够使用DEFINE来产生命令式风格的变量。然而，这些名字绑定所产生的是命名常量而不是变量。

我们将DEFINE称为是一种特殊的形式，因为与算术函数等通常的原始函数相比较，它是通过EVAL的一种不同的方式来被解释的。我们将很快就能够了解到这一点。 651

DEFINE的一种最简单的形式，是用来将一个符号绑定到一个表达式的值上。这种形式为：

```
(DEFINE 符号 表达式)
```

例如，

```
(DEFINE pi 3.14159)
(DEFINE two_pi (* 2 pi))
```

如果已经将这两个表达式键入给Scheme解释器，然后再键入pi时，就将显示值3.14159；当键入two_pi时，将显示值6.28318。实际上，计算机显示的值比真正的数值位数少。

Scheme中的名字由字母、数字和除括号之外的特殊字符组成。它们是大小写敏感的，并且不能以数字打头。

DEFINE函数也用于将一个lambda表达式绑定到一个名字之上。在这种情况下，lambda表达式被缩写，以便省略lambda一字。通过这种形式，DEFINE将接受两个链表作为参数。第一个参数就是函数调用的原型，这是一个链表，其中函数名的后面跟随着形参。第二个链表包含名字将要绑定的表达式。这样形式的DEFINE一般为：[⊖]

```
(DEFINE (函数名 参数)
  (表达式)
)
```

下面的示例中对DEFINE的调用是将名字square与跟随它的表达式相绑定。含有一个参数：

```
(DEFINE (square number) (* number number))
```

在解释器计算了这个函数后，它就能够使用。例如：

```
(square 5)
```

将显示值25。

下面将举例说明原始函数与DEFINE特殊形式之间的区别。考虑

652

```
(DEFINE x 10)
```

如果DEFINE是一个原始函数，EVAL在这个表达式上的第一个行动，将是对DEFINE的两个参数求值。如果还没有将x绑定到一个值之上，这将会是一个错误。此外，如果已经定义了x，这也将是一个错误，原因是DEFINE将会试图重新来定义x，而这是非法的。

下面是另外一个简单函数的例子。这个函数将计算，在给定两条边的长度时，直角三角形的斜边的长度。

```
(DEFINE (hypotenuse side1 side2)
    (SQRT(+(square side1)(square side2))))
)
```

注意，这个hypotenuse函数使用了前面所定义的square函数。

15.5.5 输出函数

Scheme包括一些简单的输出函数，例如：

```
(DISPLAY 表达式)
```

和

⊖ 实际上，DEFINE的一般形式为一个包含一个或多个表达式的列表，尽管在大多数情况下只包含一个。出于简单的缘故，这里只包含了一个表达式。

```
(NEWLINE)
```

它们的语义是明显的。然而，大多数Scheme程序的输出，即是解释器的正常输出，所显示的是将EVAL应用于高层函数的结果。

15.5.6 数值谓词函数

谓词函数返回一个布尔值（或者是真，或者是假）。Scheme 包括一组用于数值数据的谓词函数，这些函数中的一部分为：

函数	意义
=	等于
<>	不等于
>	大于
<	小于
>=	大于或者等于
<=	小于或者等于
EVEN?	是一个偶数吗？
ODD?	是一个奇数吗？
ZERO?	是零吗？

注意，所有预定义的谓词函数的名字都是以问号来结束的。在Scheme中的两个布尔值是#T和#F。Scheme中预定义的谓词函数返回空链表()，而不是#F。虽然这二者是等价的。谓词函数返回的任何非空链表都被解释为#T。从可读性方面来考虑，本章中所有谓词函数的例子都将返回#F，而不是()。 653

15.5.7 控制流程

Scheme使用3种不同的控制流结构，一种是基于命令式语言的选择结构的模型，其他两种是基于在数学函数中使用的求值控制。

Scheme中有两种控制结构，一种是双向选择，另一种则是多向选择。这两者都是特殊形式。双向选择器称为IF，它具有三个参数：一个谓词表达式、一个then_表达式以及一个 else_表达式。对于IF的调用的形式为：

```
(IF谓词then_表达式else_表达式)
```

例如，

```
(DEFINE (factorial n)
  (IF (= n 0)
    1
    (* n (factorial (- n 1)))
))
```

请注意，这个函数的形式和上面所给的阶乘的数学定义是紧密相关的。

数学函数定义中的控制流程，与命令式程序设计语言程序中的控制流程相当不同。命令式语言中的函数被定义为一种可能包括几种类型的顺序控制流程的语句集合；然而数学函数不具有多重语句，并且只使用递归和条件表达式来说明计算流程。例如可以使用两个操作，将阶乘函数定义为：

$$f(n)=\begin{cases}1 & \text{当 } n=0\\ n*f(n-1) & \text{当 } n>0\end{cases}$$

数学条件表达式的形式是采用一连串的对，其中每个对都是一个被守护的表达式。每个被守护的表达式由一个守护谓词和一个表达式所组成。这样的一个条件表达式的值就是与为真的谓词相关的表达式的值。对于给定的参数或参数链表，只有其中的一个谓词为真。

基于数学条件表达式的Scheme多向选择器的特殊形式称为COND。COND是一个数学条件表示式的稍微一般化的版本：它允许多个谓词同时为真。因为不同的数学条件表达式具有不同数
654 目的参数，COND不需要有固定数目的实参。COND的每一个参数都是一对表达式，其中的第一个参数是一个谓词。

COND的一般形式是：

```
(COND
  (谓词_1 表达式)
  (谓词_2 表达式)
  ...
  (谓词_n 表达式)
  (ELSE 表达式)
)
```

在一些实现中，ELSE是可省略的。

COND的语义是：对于参数的谓词的求值，一次一个，从第一个开始依照顺序，直到其中一个的值为#T。然后对于在第一个为#T的谓词后面的表达式进行求值，将值返回作为COND的值。如果没有谓词为真却有一个ELSE子句，那么求值表达式并返回结果值。如果没有谓词为真也没有ELSE子句，那么COND值是未指定的，因此所有COND都应该包含一个ELSE子句。

请注意，在一个COND与一个在末尾具有一条"otherwise"子句的多向选择语句，如Ada中的case语句，之间的相似性。

下面，是使用COND的一个简单函数的例子。

```
(DEFINE (比较 x y)
  (COND
    ((> x y) "x 大于y")
    ((< x y) "y 大于x")
    (ELSE    ("x等于y"))
)
```

在下面的章节里，我们还有更多关于使用COND的例子。

第三种Scheme控制机制是递归，它在数学中使用，用于控制重复操作。第15.5.10节中的大部分函数例子都使用递归。

15.5.8 链表函数

基于LISP的语言的，实际上即函数式语言的最通常的应用，是用于链表的处理。本小节将介绍Scheme中的链表处理函数。

我们首先描述的Scheme原始函数，即是工具函数EVAL，它是Scheme函数应用操作的性质所需要的函数。EVAL是Scheme中所有函数求值的基础。不论是原始函数还是其他函数的求值，
655 都调用它来处理Scheme解释器中"读-计算-写"行动的求值部分。当应用于一个原始函数之上时，EVAL首先将对于给定的函数的参数求值。当一个函数调用中的实参的本身就是函数调用时（通常就是这种情况），这个行动是必需的。然而在一些函数调用中，参数是数据元素，它可能是原子也可能是链表，而并非函数引用。当一个参数不是函数引用时，显然就不应该对它进行计算。我们在前面没有涉及这一点，因为不能够错误地将文字常数用作函数名。

假设我们有一个函数，它具有两个参数，一个是原子，另外一个是链表。函数的目的是决定是否所给定的原子是在给定的链表中。原子和链表都是文字常数数据，二者都不应该被计算。为了避免计算一个参数，首先将它作为一个原始函数QUOTE的参数来给出。QUOTE函数将它返回，而不会进行任何改变。下面的示例说明QUOTE函数：

```
(QUOTE A) 返回 A
(QUOTE (A B C)) 返回 (A B C)
```

在本章的余下部分，我们将会使用一种常用的缩写来表示对QUOTE的调用，即在被引述的表达式前面放置单引号（'）。因此，我们将会使用'(A B)来替代(QUOTE(A B))。

用于链表处理的语言必须包括有关组织链表的原始函数。尤其是，它必须能够提供一些操作用于选择链表中的一部分，这在某种意义上即是拆散链表，以及至少具有一种操作来构造链表。在Scheme中有两个原始链表选择器：CAR和CDR（发音是"could-er"）。CAR函数将返回给定的链表中的第一个元素。下面的示例用来说明CAR函数：

```
(CAR '(A B C)) 返回 A
(CAR '((A B) C D) 返回 (A B)
(CAR 'A) 是错误的，因为 A 不是一个链表
(CAR '(A)) 返回 A
(CAR '()) 是错误的
```

CDR函数返回给定的链表中CAR部分被移走之后的剩余部分：

```
(CDR '(A B C)) 返回 (B C)
(CDR '((A B) C D)) 返回 (C D)
(CDR 'A) 是错误的
(CDR '(A)) 返回 ()
(CDR '()) 是错误的
```

下面是CAR和CDR函数名字的由来。这些名字起源于LISP在IBM 704计算机上的第一种实现。704机器的内存字具有两个域，分别被命名为减量（decrement）和地址（address），用于各种不同的操作数寻址。每一个域都可以存储一个机器存储器的地址。704机器还包括两条机器指令，656
分别被命名为CAR（contents of address register）和CDR（contents of decrement register），用于提取相关的域。十分自然地，可以使用内存字的这两个域来存储链表节点上的两个指针，以便一个内存字可以完整地存储一个节点。使用这些方法，704机器中的CAR和CDR指令提供了高效率的链表选择器。它们的名字也就被用于LISP的所有方言中，来作为两个原始函数的名称。

下面是一个简单的函数示例：

```
(DEFINE (second lst) (CAR (CDR lst)))
```

一旦对这个函数求值，它就可以被使用，如：

```
(second '(A B C))
```

该语句返回B。

CONS是一种原始链表构造器。它以它的两个参数来建立一个链表，第一个参数或者是一个原子或者是一个链表，第二个参数通常是一个链表。CONS将它的第一参数插入，以作为它的第二个参数的新的CAR。考虑下面的示例：

```
(CONS 'A '()) 返回 (A)
(CONS 'A '(B C)) 返回 (A B C)
(CONS '() '((A B)) 返回 (() A B)
(CONS '(A B) '(C D)) 返回 ((A B) C D)
```

这些CONS操作的结果显示在图15-2中。请注意，在某种意义上CONS是CAR和CDR的逆。

CAR和CDR把一个链表拆开，然而CONS使用给定的链表部分来构造一个新的链表。CONS的两个参数成为新的链表的CAR和CDR。因此，如果lis是一个链表，那么

```
(CONS (CAR lis) (CDR lis))
```

将返回一个与lis完全相同的链表。

由于在本章中讨论的都是相对简单的问题和程序，读者也许觉得没有必要将CONS用于两个

```
(CONS 'A 'B)
```

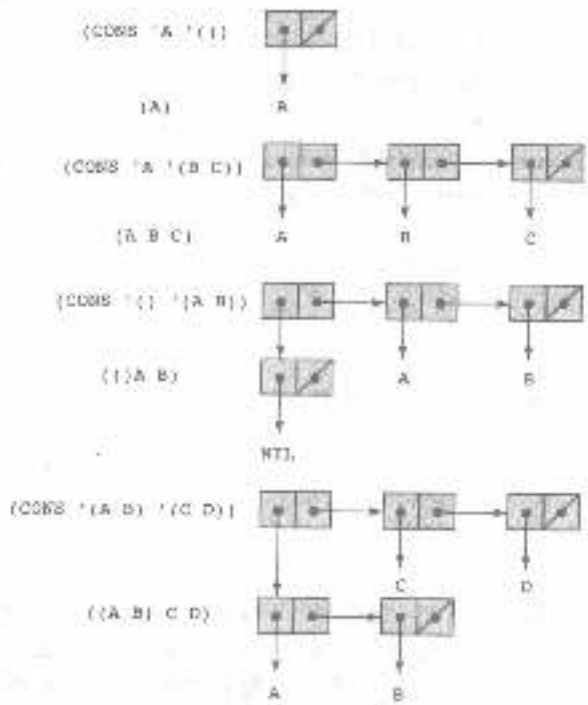


图15-2 几种CONS操作的结果

原子。然而这是合法的，并且常常会意外地发生。将CONS用于两个原子的结果是一个点对。这个名字来自于Scheme中显示结果的方式。例如，考虑下面的调用：

显示其结果是：

657

```
(A . B)
```

这个带点的对表示这个单元具有两个原子，而不是一个原子和一个指针，也不是两个指针。

LIST是一个函数，它从可变数目的参数来构造一个链表。它是缺乏的CONS的一个速记版本。例如：

```
(LIST 'apple 'orange 'grape) 返回 (apple orange grape)
```

15.5.9 处理符号原子和链表的谓词函数

Scheme具有三种用来处理符号原子和链表的重要的谓词函数，即EQ?，NULL?和LIST?。

EQ?函数带有两个符号参数。如果这两个参数都是原子，并且这两个参数相等，它将返回#T，否则，它将返回#F。考虑下面的示例：

```
(EQ? 'A 'A) 返回 #T
```

```
(EQ? 'A 'B) 返回 #F
(EQ? 'A '(A B)) 返回 #F
(EQ? '(A B) '(A B)) 返回 #T，或者是返回 #F
```

658

如最后的这一种情形所显示的，使用EQ?来比较链表的结果取决于实现。一些实现将产生#T，而另外的一些则产生#F。之所以不同的原因在于，通常是将EQ?实现为一种指针的比较（然而两个给定的指针会指向相同的地方吗），而且常常两个完全一样的链表并没有在内存中进行复制。当Scheme中的系统产生一个链表时，它会检查是否已经有这样的一个链表。如果有，新的链表将仅仅是一个指向已经存在的链表的新指针，在这种情况下，EQ?将判断这两个链表相等。然而在某一些情况下，要查出一个相同的链表是否存在，可能是困难的，此时就将创建一个新的链表，在这种现象里EQ?将产生#F。

注意，可以将EQ?作用于符号原子，但是不一定能够将它作用于数值原子。=谓词则能够作用于数值原子，但是不一定能作用于符号原子。正如在上面所讨论过的，也不能够将EQ?可靠地作用于链表参数。

在不知道两个原子是符号的还是数值的情形，如果能够进行相等测试，有时会很方便。为此，Scheme中还有一个不同的谓词EQV?，可以将它作用于数值的和符号的原子。使用EQ?或=，而非EQV?的主要原因，是使用EQ?和=可能比使用EQV?更快速。

如果LIST?谓词函数中唯一的参数是一个链表，函数将返回#T，否则函数将返回#F，正如下面的示例所示：

```
(LIST? '(X Y)) 返回 #T
(LIST? 'X) 返回 #F
(LIST? '()) 返回 #T
```

NULL?函数测试它的参数，以决定是否为空链表。如果是空链表就将返回#T。考虑下面的例子：

```
(NULL? '(A B)) 返回 #F
(NULL? '()) 返回 #T
(NULL? 'A ) 返回 #F
(NULL? '(())) 返回 #F
```

因为参数不是空链表，所以在最后这种情况下返回#F。这个参数是一个包含了空链表元素的链表。

15.5.10 Scheme函数示例

本节介绍Scheme中函数定义的一些例子。这些程序用于解决简单的表处理问题。 659

考虑一个给定的原子在一个给定的简单链表中的成员资格问题，简单链表没有子表。如果将这个函数命名为member，可以将它使用如下：

```
(member 'B '(A B C)) 返回 #T
(member 'B '(A C (B))) 返回 #F
```

从循环的角度来考虑成员资格的问题，只是将给定的原子与给定的链表中的元素进行比较，按某种次序一次比较一个，直到发现了一个匹配或是链表中的元素都被比较过了为止。也可以使用递归函数来完成相类似的过程。这个函数能够将给定的原子与链表的CAR进行比较，如果它们相匹配，将返回#T；如果它们不相匹配，就忽略链表的CAR部分，然后在链表的CDR中继续进行搜索。因此，函数应该以链表的CDR作为链表参数，来调用它的本身，并且返回该递归

调用的结果。如果所找的原子不在链表中，这个函数最终被（自己）调用时的实参将会是一个空链表。这将迫使它返回#F。在这个过程中，递归将会有两种结果：在某次调用时链表为空，并且返回#F，或者是发现了一个匹配，并且返回#T。

总而言之，这个函数必须处理三种情况：一个为空的输入链表，原子和链表的CAR之间的一个匹配，以及原子与链表的CAR之间的不匹配，这将导致递归调用。它们正是COND的三个参数。最后一个是由ELSE所激发的默认的情形。这个完整的函数应该是：

```
(DEFINE (member atm lis)
  (COND
    ((NULL? lis) #F)
    ((EQ? atm (CAR lis)) #T)
    (ELSE (member atm (CDR lis)))
))
```

这种形式是简单Scheme表处理函数的典型。在这种函数中，对于链表中的数据的处理，一次一个元素，CAR可以得到单个的元素，通过在链表的CDR上使用递归而继续这个过程。

值得注意的是，因为应用空链表的CAR是一个错误，所以对于空链表的测试必须是在等于测试之前。

另外的一个示例，考虑决定两个给定的链表是否相等的问题。如果这两个链表是简单链表，问题的解决相对容易，虽然会涉及一些读者可能不熟悉的程序设计技术。下面是一个名称为
660 equalsimp的、用于比较简单链表的谓词函数：

```
(DEFINE (equalsimp lis1 lis2)
  (COND
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((EQ? (CAR lis1) (CAR lis2))
          (equalsimp (CDR lis1) (CDR lis2)))
    (ELSE #F)
))
```

由COND的第一个参数所处理的第一种情况中，第一个链表参数是空链表。如果第一个链表参数本来就为空，这种情况则可能发生在一个外部调用中，或为一个递归调用使用两个参数链表的CDR来作为它的参数，如果第一个链表中的全部元素都被前面的递归调用删除了，第一个链表在这种调用中就可能是空的。当第一个链表为空时，就必须对第二个链表进行检测，以确定它是否也为空。如果正是如此，它们两者就是相等的（或者它们本来就相等，或者所有前面的递归调用之中的CAR是相等的），并且NULL?将正确地返回#T。如果第二个链表不为空，它就将比第一个链表大，并且NULL?应该返回#F。回忆一下，一个谓词函数返回任何非空链表都被解释为#T。

第二种情况是第二个链表是为空，而第一个链表则不是。只有当第一个链表比第二个大时才会发生这种情形。因为第一种情况描述了第一个链表为空的所有可能的实例，当前仅仅必须测试第二个链表。

第三种情况是在两个链表中相对应的元素之间进行相等测试的递归步骤。它通过比较两个非空链表的CAR来完成这个步骤，如果它们是相等的，那么这两个链表直到这一点都是相等的。然后将递归用于两者的CDR，当发现了两个不相等的原子时，这种情况将失败。如果是这样的话，就不再需要继续这一过程，因此选择默认的ELSE，它将返回#F。

注意，equalsimp的期望参数是链表，如果两个参数之一或这两者都是原子，equalsimp

就不能正确地操作。

一般链表的比较问题比简单链表的比较稍微复杂一些：这是因为在比较过程中，必须对子表全面追踪。此时递归特别适用，因为子表的形式与所给定的链表相同。在任何时候，如果两个给定的链表中相对应的元素是链表，则将它们分成为两个部分，即CAR和CDR，而且将递归应用于这两个部分。这正是“分治”方式的用途的一个完美示例。如果两个给定链表中相对应的元素是原子，就使用EQ?来进行比较。

这个完整函数的定义如下：

```
(DEFINE (equal lis1 lis2)
  (COND
    ((NOT (LIST? lis1)) (EQ? lis1 lis2))
    ((NOT (LIST? lis2)) #F)
    ((NULL? lis1) (NULL? lis2))
    ((NULL? lis2) #F)
    ((equal (CAR lis1) (CAR lis2))
           (equal (CDR lis1) (CDR lis2)))
    (ELSE #F)
))
```

661

COND的前两种情况处理的是其中任何一个参数是一个原子，而不是链表的情形。第三和第四种情况则是其中的一个或两个链表为空的情形。这些情况也被用来防止后来试图将CAR应用于空链表。第五个COND的情况是最有趣的一种。这里的谓词是参数为链表的CAR的一个递归调用。如果调用返回#T，那么递归将再一次被用于链表的CDR。这里允许了两个链表包括任意深度的子表。

可以将equal的定义用于任何表达式对，而不仅仅是链表。equal与系统谓词函数EQUAL?是等价的。注意，应该只有在必需的时候才使用EQUAL?（不知道实参的形式），因为它比EQ?和EQV?慢得多。

另外一个普遍需要的链表操作是构造一个包含了两个给定的链表参数中的所有元素的新链表。通常，这是通过实现一个被称为append的Scheme函数来完成的。获取结果是通过重复地使用CONS从而将第一个参数链表中的元素放入到第二个参数链表内。最后，第二个参数链表将成为计算的结果。为了清晰了解append的行为，考虑下面的例子：

```
(append '(A B) '(C D E)) 返回 (A B C D E)
(append '((A B) C) '(D (E F))) 返回 ((A B) C D (E F))
```

append函数的定义为：

```
(DEFINE (append lis1 lis2)
  (COND
    ((NULL? lis1) lis2)
    (ELSE (CONS (CAR lis1) (append (CDR lis1) lis2)))
))
```

第一种COND的情况是用来在第一个参数链表为空时，终止递归过程并返回第二个链表。在第二种CONS情况（ELSE）下，第一个参数链表中的CAR部分与递归调用所返回的结果相连接。这个递归调用将第一个链表中的CDR部分作为第一个参数来传递。

考虑下面一个名为guess的Scheme函数，它使用在本节中所描述的member函数。读者在阅读下面关于它的描述前，应试着去判断它的功能。假设参数是简单的链表。

662

```
(DEFINE (guess lis1 lis2)
```

```
  (COND
    ((NULL? lis1) '())
    ((member (CAR lis1) lis2)
             (CONS (CAR lis1) (guess (CDR lis1) lis2)))
    (ELSE (guess (CDR lis1) lis2))
))
```

guess产生一个包含了它的两个参数链表中的共同元素的简单链表。所以如果将参数链表表示为集合的话，guess计算出的正好是一个表示这两个集合之交的链表。

LET是一个函数，它允许将名字暂时绑定于子表达式的值上。常常将它用于从一些比较复杂的表达式中分解出共同的子表达式，然后能够将这些子表达式的名字用于另外一个表达式的求值中。LET函数的一般形式是：

```
(LET (
(名字_1 表达式_1)
(名字_2 表达式_2)
...
(名字_n 表达式_n))
表达式 (表达式)
)
```

LET的语义是首先对n个表达式求值，将产生的值绑定到与它们相关的名字上，然后再对函数体中的表达式求值。LET的结果是它的函数体中最后一个表达式的值。下面举例来说明LET的用法。例子中的函数将计算二次方程的根，我们假设是实根。[6]

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
                (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
        )
  (LIST (+ minus_b_over_2a root_part_over_2a)
            (- minus_b_over_2a root_part_over_2a))
)
```

663 上面例子使用LIST来创建组成结果的两个值的列表。

LET创建一种新的局部静态作用域，其方式与Ada中的declare的方式大致相同。新的变量可以被创建、使用，然后当到达新的作用域的末尾时被丢弃。LET的命名组成部分十分类似于赋值语句，但是只能够将它们用于新的作用域。此外，还不能够在LET中将它们重新绑定到新值。

实际上，LET是LAMBDA表达式的一种速记形式。下面的这两个表达式是等价的：

```
(LET ((alpha 7))(* 5 alpha))
((LAMBDA (alpha) (* 5 alpha)) 7)
```

在第一个表达式中，通过LET，7与alpha相绑定；在第二个表达式中，通过LAMBDA表达式的参数，7与alpha相绑定。

15.5.11 函数形式

本节将描述两种由Scheme提供的常用数学函数的形式，即复合和“应用到所有参数”的函

[6] Scheme的某些版本将复数包括为数据类型，并且将计算方程的根，而不论它们是实数还是复数。

数形式。我们已经在15.2.2小节里给出了这两种函数形式的数学定义。

15.5.11.1 函数复合

函数复合是最初的LISP所提供的唯一的原始函数形式。所有后来的LISP方言，包括Scheme，也都提供了这种函数形式。函数复合是EVAL工作机制的本质，在使用EVAL时，将所有的非引述的链表都解释成为函数调用，这就要求首先对它们的参数求值。这个过程将被递归地应用到表达式中最小的链表上。这正是函数复合的含义。下面的例子说明了函数复合：

```
(CDR (CDR '(A B C))) 返回 (C)
(CAR (CAR '((A B) B C))) 返回 A
(CDR (CAR '((A B C) D))) 返回 (B C)
(NULL? (CAR '(() B C))) 返回 #T
(CONS (CAR '(A B)) (CDR '(A B))) 返回 (A B)
```

注意，函数名在内层调用中不被引述。这是因为必须对于它们求值，而不是将它们作为字面常量数据来处理。

一些最常用的Scheme函数复合被内建为单一函数。例如，(CAAR x)等价于(CAR(CAR x))(CADR x)等价于(CAR (CDR x))，(CADDAR x)等价于(CAR(CDR(CDR(CAR x))))。在函数名中'C'和'R'之间A和D的任意组合（最多为4个）都是合法的。

15.5.11.2 一种"应用到所有参数"的函数形式 664

在通常的函数式程序设计语言中所提供的最常用的函数形式，是数学的"应用到所有参数"函数形式的变体。最简单的是mapcar，它具有两个参数，其中的一个是函数，另外的一个则是链表，mapcar将给定的函数应用到给定链表中的每个元素上，而且它将返回一个链表，其中的元素就是这些应用的结果。Scheme中的mapcar的定义为：

```
(DEFINE (mapcar fun lis)
  (COND
    ((NULL? lis) '())
    (ELSE (CONS (fun (CAR lis)) (mapcar fun (CDR lis))))
))
```

注意，mapcar的简单形式，它表达了一个复杂的函数形式。这正是Scheme语言的强大表达能力的一种证明。

作为mapcar的使用的一个例子，假设我们想要计算一个链表中所有元素的立方，我们能够通过使用下面的函数来完成：

```
(mapcar (LAMBDA (num) (* num num num)) '(3 4 2 6))
```

这个调用将返回 (27 64 8 216)。

注意，在这个例子中的mapcar的第一参数是一个LAMBDA表达式。当EVAL在计算LAMBDA表达式时，它构造了一个函数，这个函数除了是无名称的之外，与任何预定义的函数都有着相同的形式。在上面的表达式中，这个无名函数立刻被应用到参数链表中的每个元素上，并且将其结果返回到一个链表中。

15.5.12 产生代码的函数

程序和数据具有相同结构的事实可以被发展来构造程序。回忆Scheme的解释器，就是一个称为EVAL的函数，Scheme系统将EVAL应用于每一个键入的表达式。Scheme程序也能够直接地调用EVAL函数。这样，Scheme程序就能够构造其他的程序，并且能够调用EVAL来对它们求值。

这个过程的最简单的例子之一涉及数值导子。大部分的Scheme系统，包括+函数，这个函

数以任意数目的数值原子为参数，并且返回它们之和。例如，(+ 3 7 10 2) 将返回22。

我们的问题是：假设在一个程序中，我们有一个数值原子的链表，并需要求取它们之和。我们不能够直接将+应用到链表上，因为+只取原子参数而不是一个数值原子的链表。当然，我们可以编写一个函数，这个函数将使用递归来穿过整个链表，重复地把链表的CAR加入到它的CDR之和。下面就是这样一个函数：

665

```
(DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (+ (CAR lis) (adder (CDR lis))))
))
```

这个问题的另一种可能的解决办法是编写一个函数，这个函数将使用适当的参数形式对+进行调用。这可以通过使用CONS将原子+插入数值链表中来完成。然后，能够将这个新的链表提交给EVAL求值，如下面的这个函数：

```
(DEFINE (adder lis)
  (COND
    ((NULL? lis) 0)
    (ELSE (EVAL (CONS '+ lis)))
))
```

注意，此处加法函数的名字被引述，以防止EVAL在CONS的求值中对它求值。作为一个例子，考虑下面的调用：

```
(adder '(3 4 6))
```

这个调用将引起adder建立链表

```
(+ 3 4 6)
```

然后将链表提交给EVAL，EVAL调用+并返回结果13。

在所有Scheme的比较早期的版本中，EVAL函数在程序的最外作用域中计算它的表达式。最新版的Scheme（即Scheme 4）要求EVAL具有第二个参数，以说明是在哪一个作用域中进行表达式的计算。为了简单起见，我们在示例中没有包括作用域参数，而且我们在这里也不讨论作用域的名字。

15.6 COMMON LISP

COMMON LISP（Steele，1984）是将几种20世纪80年代初期的LISP方言的特性结合进一种语言的结果，这些方言包括Scheme。作为组合的结果，COMMON LISP是一种大型并且相当复杂的语言。然而，它的基础是最初的LISP语言，因而它的语法，原始函数和基本性质都来自于LISP语言。

设计人员认识到动态作用域所提供的灵活性和静态作用域的简单性，COMMON LISP允许对这两种作用域的使用。默认的变量作用域是静态的，但是如果声明一个变量是"special"，这个变量的作用域就成为动态的。

656

COMMON LISP具有一长列的特性：多种数据类型和结构，包括记录、数组、复数和字符串等，强有力的输入和输出操作，将一系列函数和数据模块化的包，还提供访问控制。

在某种意义上，Scheme和COMMON LISP是相反的。Scheme较为小型化，而且比较简洁，其中的部分原因是由于它仅仅使用静态作用域。COMMON LISP是一种商业语言，并且成功地

成为人工智能应用中的一种广泛使用的语言。另一方面，Scheme常常被用于大学的函数式程序设计课程。因为它相对小型，可以将它用于函数式语言的研究。而使得COMMON LISP成为一种极为大型语言的一个重要设计方针，是为了使它与LISP的一些较早期的方言相兼容。COMMON LISP正是产生于这些方言的。

15.7 ML

ML (Milner et al., 1990) 也像Scheme一样，是一种静态作用域的函数式程序设计语言。然而，在许多的重要方面它又不同于LISP及其方言，包括Scheme。也许其中最重要的区别是ML是强类型的语言，而Scheme实质上是无类型的语言。ML具有类型声明，并使用类型推理（曾经在第5章简略地讨论过）。由于使用了类型推导，常常就不需要变量声明，每一个变量和表达式的类型都能够在编译时间被确定。Scheme和ML之间另外一个重要的区别，是ML所使用的语法更接近于命令式语言的语法，而不是LISP的。例如，ML中的算术表达式使用中缀标记。

ML具有异常处理，还有实现抽象数据类型的模块化设施。我们曾经在第2章里简略地描述过它的发展历史和主要的特性。

在ML中函数声明的一般形式为：

```
fun 函数名(形参) = 函数体表达式;
```

例如，

```
fun square (x : int) = x * x;
```

可以在参数表之后说明返回值类型，例如：

```
fun square(x : int) : int = x * x;
```

当然在这个函数中，不需要显式地说明返回值类型。

下面这一行

```
fun square (x) = x * x;
```

667

定义了一个与前一个函数定义相同的函数。ML假设表达式(x * x)中的操作符的类型为int（x是数值类型）。因而，使用算术操作符的函数不能够是多态的。同样，使用关系操作符（除了=和<>以外）以及布尔操作符的函数也不能够是多态的。然而那些仅仅使用链表的操作，=，<>和元组操作符（用来形成元组，以及施行元素的选择）的函数则可以是多态的。

如果我们已经定义了一个int类型的square函数，现在又需要定义一个real类型的square函数。此时，我们必须使用不同的名字来进行定义。原因是ML不允许用户定义的重载函数。另外，如果我们将square函数定义为real类型，调用它时的实参就不能够是整数的。原因是ML（同Ada一样）不会将整数值强制转换成为real类型。

ML的选择控制流程结构与命令式语言相似。它具有以下形式的条件表达式：

```
if 表达式 then then_表达式else else_表达式
```

必须将其中的第一个表达式计算为布尔值。

在ML中，Scheme的条件表达式可以出现在函数定义层次上。在Scheme中，是使用COND函数来确定给定参数的值，而被确定的值又能够反过来说明COND所返回的值。在ML中，针对给定参数的不同形式，可以定义函数来完成计算。这种特征是为了模仿数学中的条件函数定义的形式及意义。在ML中，定义返回值的那个特定表达式是通过对给定参数进行模式匹配来选择的。例如，如果不使用这种模式匹配的话，可以将计算阶乘的函数写成下面的形式：

```
fun fact(n : int): int = if n = 0 then 1
                         else n * fact(n - 1);
```

而如果使用参数模式匹配，我们就可以具有函数的多重定义，使用OR符号(|)来隔开依赖于参数形式的不同函数定义。例如，如果使用模式匹配，可以将计算阶乘的函数写成下面的形式：

```
fun fact(0) = 1
|   fact(n : int): int = n * fact(n - 1);
```

如果是用实参0来调用fact，就使用其中的第一个定义。如果是以一个非零的int数值来调用fact，就使用其中的第二个定义。

ML具有链表和链表操作，但是它们看起来与Scheme中的不同。链表是使用方括号来说明，链表元素使用逗号来分隔。下面是一个整数链表：

668

```
[5, 7, 9]
```

[]是一个空链表。空链表也可以用nil来说明。

Scheme的CONS函数在ML中是一个二元前置操作符，使用::来表示。例如：

```
3 :: [5, 5, 9]
```

其求值结果为[3, 5, 7, 9]。

一个链表中的元素必须都是同一类型的，因此下面的链表是非法的：

```
[5, 7.3, 9]
```

ML具有对应于Scheme中的CAR和CDR的函数，它们的名称分别为hd和tl，也即head（头）和tail（尾）的缩写。例如：

```
hd [5, 7, 9] is 5
tl [5, 7, 9] is [7, 9]
```

由于可以使用有模式的函数参数，在ML中对于hd和tl函数的使用远不如在Scheme中的频繁。例如在一个形参中，表达式

```
(h :: t)
```

实际上是两个形参，它们即是所给定链表参数的头和尾，而对应的实参是列表。例如，可以使用下面的函数来计算一个给定链表参数中的元素的数目：

```
fun length([]) = 0
|   length(h :: t) = 1 + length(t);
```

作为这些概念的另外一个示例，我们来考虑append函数。这个函数的功能与Scheme中的APPEND函数相同：

```
fun append([], lis2) = lis2
|   append(h :: t, lis2) = h :: append(t, lis2);
```

这个函数的第一个选项处理的是函数调用中的第一个参数为空链表的情况。即使最初的函数调用中的第一个参数不为空，这个选项也用来终止递归。函数的第二个选项将第一个参数链表lis拆为头与尾（CAR与CDR）两个部分，将头部分连接到递归调用的结果之上。递归调用的第一个参数就是尾部分。

在ML中，通过以下形式的值声明语句，将名字绑定到值之上：

669

```
val 所名字 = 表达式;
```

例如，

```
val distance = time * speed;
```

不要认为这个语句与命令式语言中的赋值语句完全一样；实际上它并不是。val语句将一个名字绑定到一个值上，但是在以后，不能够重新将这个名字再绑定到一个新的值。当然，在某种意义上也许能够。但是实际上，如果你的确是使用第二个val语句来将一个名字重新绑定的话，这将在环境中产生一个新的实体，而这个实体与这个名字的早先的版本无关。[3]事实上，在新的绑定之后，旧的环境的入口（用于先前的绑定的）已经不再可见。并且，新绑定的类型并不需要与先前绑定的相同。val语句没有副作用。这种语句只是简单地将一个名字加入到当前的环境中，并且将它绑定到一个值上。就像Scheme中的LET特殊形式一样，val通常用于let表达式中，它的一般形式为：

```
let val 名字 = 表达式_1 in 表达式_2 end
```

例如，

```
let
  val pi = 3.14159
in
  pi * radius * radius
end;
```

在ML中不存在强制类型转换；操作符或赋值的操作数类型必须相同，以避免语法错误。

ML还具有枚举类型、数组以及元组，它们类似于记录。

15.8 Haskell

Haskell语言（Thompson，1996）在下述的这些方面与ML相类似：它使用一种相类似的语法；它是静态作用域的；它也是强类型的，并且还使用一种相同的类型推理方式。Haskell所具有的两个特征又使得它不同于ML。首先，Haskell的函数能是多态的（ML语言的大部分函数都不是）其次，Haskell使用不严格的语义，而ML（和大多数其他程序设计语言）使用严格的语义。本节后面部分将深入讨论不严格语义和多态性。

在本节中的代码使用Haskell的1.4版本来编写。

考虑下面的阶乘函数的定义，这个函数在参数上使用模式匹配。 670

```
fact 0 = 1
fact n = n * fact (n - 1)
```

请注意这个定义与15.7节中ML的阶乘函数定义之间的区别。第一，这里没有使用保留字来引出函数定义（ML中使用fun）。第二，没有使用括号来为形参定界。[4]第三，函数的所有（具有不同形参的）不同定义都有相同的外表。

使用模式匹配，我们就能够定义一个函数来计算第n个斐波纳契数字：

```
fib 0 = 1
fib 1 = 1
fib (n + 2) = fib (n + 1) + fib n
```

可以将守卫加到函数定义的行上，以说明定义可能的应用环境。例如，

```
fact n
  | n == 0 = 1
```

---

[3] 可以将这个环境认为是一种符号表格，这个表格存储名字，而在执行时存储这些名字的绑定的值。

[4] 在ML语言中使用括号实际上也是可选的。

```
  | n > 0 = n * fact(n - 1)
```

这种阶乘的定义比我们在前面讨论过的更为精确，因为它将参数限制在能够工作的范围之内。在这种使用中的模式匹配必然会失败，因为两个值表达式的参数模式都是n。在基于数学表达式之后，这种函数定义的形式称为条件表达式（conditional expression）。

能够将一条otherwise语句作为最后一个条件放置在条件表示式中，它的语义是明显的。例如：

```
sub n
  | n < 10      = 0
  | n > 100     = 2
  | otherwise   = 1
```

注意，第8章讨论过这里的守卫和守卫命令的相似性。

考虑下面的函数定义，它的目的与第15.7节中对应的ML函数是相同的：

671

```
square x = x * x
```

然而在这种情况下，因为Haskell支持多态，所以此函数能接受任何数值类型的参数。

如同在ML中一样，在Haskell中是将链表写在方括号里，如：

```
colors = ["blue", "green", "red", "yellow"]
```

Haskell中包括一组链表操作符，例如，使用++来连接链表；:是CONS的中缀版本；..被用来说明算术系列，例如：

```
5 : [2, 7, 9]产生[5, 2, 7, 9]
[1, 3 .. 11]产生[1, 3, 5, 7, 9, 11]
[1, 3, 5] ++ [2, 4, 6] 产生 [1, 3, 5, 2, 4, 6]
```

请注意，Haskell中的:操作符与ML中的::相同。[⊖]使用：以及模式匹配，我们就可以定义一个简单函数来计算一组给定数字的乘积：

```
product [] = 1
product (x:xs) = x * product xs
```

使用product可以将一个阶乘函数写成比较简单的形式：

```
fact n = product [1..n]
```

Haskell包含了与ML的let和val相似的let结构。例如，我们可以编写：

```
quadratic_root a b c =
    let
     minus_b_over_2a = - b / (2.0 * a)
     root_part_over_2a =
                sqrt(b ^ 2 - 4.0 * a * c) / (2.0 * a)
    in
     [minus_b_over_2a - root_part_over_2a,
      minus_b_over_2a + root_part_over_2a]
```

链表综合提供一种描述表示集合的链表的方法。链表综合的语法与数学上常用的描述集合的语法相同。它的一般的形式为：

[体 | 限定符]

672

例如，

⊖ 十分有趣的是，ML将::用于给名字附加上类型变量，并且将::用于CONS，而Haskell则正好是以相反的方式来使用这两个操作符。

```
[n * n * n | n <- [1..50]]
```

定义了一个链表，这个链表的元素为1～50之间的所有数的立方。将它读作："一个 n×n×n 的链表，其中的n取自从1～50的范围内的数"。在此情况下，限定符以生成器（generator）的形式出现。它产生从1～50的数。在其他情况下，限定符则是以布尔表达式的形式出现，这时将它们称为测试（test）。能够使用这种表示法来描述许多算法，如链表的变换和排序之类。例如，考虑下面的函数，当给定数n时，这个函数将返回一个链表，其中的元素是n的所有因数：

```
factors n = [i | i <- [1..n `div` 2], n `mod` i == 0]
```

factors中的链表综合创建一个数的链表，其中的每一个数都临时地绑定到名字i上，范围是从1～n/2，并满足n mod i = 0这一条件。这的确是一个十分准确而又简短的数的因子的定义。div和mod前后的反引号（向后的撇号）指示这些函数的插入使用。当在函数注释中调用它们时，如div n 2，不使用反引号。

接下来，考虑在下面的快速排序算法的实现中，Haskell的强制转换：

```
sort [] = []
sort (h:t) = sort [b | b <- t, b <= h]
             ++ [h] ++
             sort [b | b <- t, b > h]
```

此程序对小于或等于链表头的链表元素集进行分类和与头元素进行连接，然后对大于链表头的元素集进行分类和连接到前一个结果上。这个定义比命令式语言中编码的相同的算法要简短得多。

如果一种程序设计语言要求完全求值所有的实参，那么此语言是严格的（strict），这使函数值不依赖于参数的求值次序。如果一种语言没有这种严格的要求，那么此语言是不严格的（nonstrict）。不严格的语言比严格的语言有一些不同的优点。首先，不严格的语言通常具有更高的效率，因为它避免一些求值过程。[⊖]其次，不严格的语言具有一些严格的语言不可能具备的、有趣的功能。无限链表就是一例。不严格的语言能使用一种称为懒情求值（lazy evaluation）的求值方式，这意味表达式可以在需要时再求值。 671

在Scheme中函数的参数都在函数调用之前求值。懒情求值意味着只有一个参数值对于函数的求值是必需时，才对这个参数进行求值。所以，如果一个函数具有两个参数，但是在函数的一个特定的执行之中，并不使用第一个参数，那么将不会对为那个参数传递的实参进行求值。此外，如果在函数的一个执行之中，必须被求值的只是实参的一部分，那么，将不会对其余部分求值。最后，实参只被求值一次，而不是每次都必须被求值，尽管同一个实参在一次函数调用中出现多次。

正如前面讲述的，懒情求值允许定义无穷的数据结构。例如，考虑下面的语句：

```
positives = [0..]
evens = [2, 4..]
squares = [n * n | n <- [0..]]
```

显然，没有计算机能够实际地表示出所有这些数的系列，但是如果使用懒情求值，就不会妨碍这些表达式的使用。例如，如果我们想要知道一个特定的数是否为一个完全平方，我们可以就使用一个成员资格函数来查询一个平方表。假如，我们使用一个命名为member的谓词函数来确定一个给定的链表中是否包含了某一个给定的原子，那么我们可以这样来运用：

⊖ 注意，这与一些命令式语言中布尔表达式的短路求值有关。

```
member squares 16
```

这个函数将返回True，直到发现了16，才会对squares定义求值。必须十分小心地编写member函数。尤其如果它是这样定义的话：

```
member [] b = False
member (a:x) b = (a == b) || member x b
```

这个定义中的第二行将第一个参数拆为头与尾部分，如果头与所要查找的元素(b)相匹配，函数将返回True，或者是，如果是使用链表的尾来进行函数的递归调用，返回值为True。

只有当所给定的数是完全平方时，member的这种定义才能够正确地工作。如果不是一个完全平方的话，squares会永远继续产生平方，在链表中寻找给定的数，直至达到了某种内存限制为止。下面的函数将在一个有序链表中执行成员资格测试；当表中测试的数比所要找的数大时，函数将放弃搜寻并且返回False（假）。

674

```
member2 (m:x) n
  | m < n     = member2 x n
  | m == n    = True
  | otherwise = False
```

有时懒情求值提供了一个模块化工具，考虑函数复合的情形。假设程序中有一个对函数f的调用，给f的参数是函数g的返回值。[1]因此，我们有f(g(x))。进一步假设g每次少量地产生大量的数据，则f必须每次少量地处理这些数据。使用懒情求值时，急式执行f和g将会紧紧地同步。函数g只产生让f开始处理的足够的数据。当f需要更多数据时，g重新开始产生数据，而此时f等待。如果f没有得到所有g的输出就终止了，那么g中止其执行，这样避免了无用的计算。g也不需要一个终止函数，它能因为它产生了无限数量的输出，当f终止时，g将被迫终止。因此在懒情求值情形下，g会运行得尽可能少。这个求值过程支持把程序模块化为产生器单元和选择器单元，产生器产生大量可能的结果，而选择器选择正确的子集。

懒情求值并不是没有代价的。如果这样好的表达能力和灵活性是免费的，才肯定会令人惊讶。在这种情况下，代价是一种相当复杂的语义，它将造成慢得多的执行速度。

15.9 函数式语言的应用

在过去45年间，高级程序设计语言的历史中，只有少数的函数式语言得到了广泛的应用，其中最为显著的是LISP语言。

LISP是一种通用且功能强大的语言。在最初的15年中，那些不使用LISP的人们认为它是一种使用代价非常昂贵的陌生语言。的确，在20世纪60年代和20世纪70年代的早期，人们通常认为只有两个种类的语言，一个种类包括LISP语言，而另一个种类包括所有其他的程序设计语言。

LISP是为了人工智能中的符号计算以及表处理应用而开发的。在许多人工智能应用中，LISP以及从它所派生出来的语言仍然是标准的语言。

在人工智能之中，许多领域主要都是通过LISP的使用来发展的。虽然也使用了其他类型的语言，主要是逻辑程序设计语言，但是大部分已有的专家系统都是用 LISP 开发的。LISP 也在知识表示、机器学习、智能训练系统和语音和视觉的建模中占有支配性的地位。

675

LISP在人工智能以外的一些应用也是成功的。例如，Emacs文本编辑程序，用作符号微积

[1] 并称为执行Hughes (1989)。

分的符号数学系统MACSYMA等等，都是使用LISP编写的。LISP机器是一种个人计算机，它的整个系统软件都是使用LISP编写的。LISP还在许多应用领域里被成功地用来开发实验系统。

Scheme广泛用于函数式程序设计的教学。它也在一些大学中用于程序设计入门课程的教学。对ML和Haskell的使用，大致上仅限于研究实验室和大学。

15.10 函数式语言和命令式语言的比较

函数式语言能够具有非常简单的语法结构。LISP就是其中的一个例子。它的链表结构既可以用来组织代码，也可以用于数据。命令式语言的语法要复杂得多。

与命令式语言的语义相比，函数式语言的语义也是简单的。例如，在对3.5.3节中的循环结构进行标志语义的描述时，循环从迭代转换成为了递归。在一种纯粹函数式的语言中，这样的转换就不需要。函数式语言中没有迭代。另外在第3章中，我们曾经假设这一章中命令式结构的标志语义描述不具有表达式的副作用。所有基于C的语言都具有表达式副作用，因而上面的这种限制是不现实的。然而，纯函数式语言的标志语义描述就不需要这样的限制。

函数式程序设计社区的一些人声称，使用函数式程序设计会引起开发效率呈数量级增长。这主要是由于函数式程序的大小只有命令式语言程序大小的10%。解决一些问题时会达到这个比例。而解决有一些可能会达到用命令式语言编写的程序大小的25%左右（Wadler, 1998）。这些因素让函数式程序设计的拥护者声称它的开发效率是使用命令式程序设计语言的4~10倍。然而，单单只用程序的大小并不是好的测量开发效率的因素。当然，并不是每一行源代码都有相同的复杂度，他们也不会花相同的时间在每一行代码上。实际上，由于处理变量的必要性，命令式程序用很多行代码来进行初始化和轻微地改动变量。

676
/
677

执行效率是它们的另一个可比较之处。当解释函数式程序时，它们当然比编译好的命令式程序慢，然而，现在有一些专门为函数式语言设计的编译器，这使函数式程序和命令式程序运行速度的差异不再那么显著。有人可能会说，因为函数式程序明显比对应的命令式程序小，所以它们应该执行得更快。然而，事情常常不是这样的，因为函数式语言的一些语言特征对执行效率有很大的负面影响。考虑函数式程序和命令式程序的相对效率，一般的函数式程序的执行时间是其对应的命令式程序的2倍（Wadler, 1998）。这是合理的估计。这听起来令人感到诧异，对于一个给定的项目，这个因素常常让人们放弃使用函数式语言。然后，两个因素的不同只有在严格要求执行速度的情况下才是重要的。有许多情形下并不要求有很严格的执行速度。例如，考虑许多用命令式语言编写的程序，如用JavaScript和PHP编写的Web软件以及Java小程序，它们都比对应的编译版本慢，但对这些应用程序，执行速度并不是最优先考虑的事。

函数式语言的另一个潜在的优点是可读性好。在许多命令式程序中，对变量细节的处理把程序逻辑弄得难以理解了。考虑一个计算正整数n的立方和的函数。用C语言编写如下：

```
int sum_cubes(int n){
  int sum = 0;
  for(int index = 1; index <= n; index++)
    sum += index * index * index;
  return sum;
}
```

而用Haskell编写如下：

```
sumCubes n = sum (map (^3) [1..n])
```

这种版本简单地指定了3个步骤：

1）构建数列 {1..n}。

2）通过映射计算三次方的函数到数列中的每个数来创建新数列。

3）对新数列求和。

因为缺乏变量和迭代控制的细节，所以这个版本比C版本具有更好的可读性。

命令式语言的并发执行很难设计，也很难使用。例如，考虑Ada的任务模型。它的并发任务的协作由程序设计人员负责。首先，我们通过把函数式程序转化为图来执行它们。然后通过图减化过程来执行这些图。这样能处理许多程序设计人员来指定的并发。图表示自然地为并发执行提供了机会。在这个过程中的协作同步不是程序设计人员考虑的内容。此过程的细节描述超出了本书讲述的范畴。

在一种命令式语言中，程序员必须静态地将程序分成它的并发部分，然后将这些并发部分编写成任务。这可能是一个复杂的过程。函数式语言中的程序能够被执行系统动态地分成并发部分，这使得程序能够极大程度地适应于所运行的硬件。在命令式语言中理解并发程序则要困难得多。

要准确地弄清楚为什么函数式语言没有得到流行不是一件简单的事。很显然，早期实现的低效性是一个因素，并且非常有可能的是，至少有一些当前命令式程序设计人员相信用函数式语言编写的程序运行仍然很慢。此外，大量的程序设计人员是从命令式语言开始的，这使他们觉得函数式程序很奇怪，而且难以理解。对于许多习惯于命令式程序设计的人，切换到函数式程序设计是一件没有吸引力的，潜在上很困难的事。另一方面，从函数式语言开始的那些人从来不会注意到函数式程序的奇异之处。

显而易见地，那些命令式程序设计人员相信，当为他们都自由自在地使用命令式语言进行程序设计，所以命令式程序设计实际上是一种更自然的程序设计方式。一些函数式程序设计人员也是这样看待函数式程序的。当然，没有人发现有一种好的方式来衡量"自然"。最后，函数式程序设计接近于数学，这显然导致了函数式语言的准确和雅致，但事实上，即使许多程序设计人员（特别是那些不擅长数学的人）不那么容易使用函数式程序设计语言。

小结

数学函数是一些命名的或未名的映射，它们仅仅使用条件表示式和递归来控制它们的计算。能够通过函数形式来构造复杂的函数，在其中将函数用作参数、返回值或将函数用作参数又用作返回值。

函数式程序设计语言是基于数学函数的。纯函数式语言不使用变量或赋值语句产生结果，它们使用函数的应用、条件表示式和递归来作为执行的控制，并且使用函数形式来构造复杂的函数。LISP在初始时是纯函数式语言，但是很快便具有了许多命令式语言特性，以追求语言的效率和易用性。

678

LISP始第一个版本产生于人工智能应用中对表处理语言的需要。LISP至今是这个领域里最为广泛应用的语言。

LISP的第一种实现是便然出现的：EVAL的最初版本只是开发者展示编写通用的LISP函数的可能性的。

因为LISP的数据和LISP的程序具有相同的形式，我们可以使用一个程序来检查另外的一个程序。EVAL允许立即执行这样的程序。

Scheme是LISP的一个相对简单的方言，它仅仅使用静态作用域，像LISP一样，Scheme中的主要系统函数包括用于构造或拆散表的函数，用于条件表示式的函数以及用于数字、符号和表的简单的谓词函数。Scheme还包括一些命令式操作，例如，改变给定表中的一个元素。

COMMON LISP是一种基于LISP的大型语言，它被设计来包括20世纪80年代初期的各种LISP方言中的大部分特性。它允许静态作用域和动态作用域两变量，并且包括了许多命令式的特性。

ML是一种静态作用域的强类型函数式程序设计语言，它的语法更类似于命令式语言，而不是类似

于LISP语言。ML包括了类型推理系统、异常处理、数据结构和抽象数据类型。

Haskell与ML相类似，Haskell中的所有表达式都使用一种懒惰方式来求值。这就允许程序来处理无穷链表。Haskell所支持的链表综合提供了一种方便而熟悉的语法来描述集合。

虽然LISP的主要应用领域是人工智能，但是它也成功地应用于许多不同的问题领域。

虽然在有些方面，纯函数式语言优于命令式语言，但是函数式语言在冯·诺依曼机器上执行的低效率阻碍了它们的应用。

文献注释

McCarthy（1960）发布第一种公开发行的LISP版本。从20世纪60年代的中期，到20世纪70年代的后期所广泛使用的版本。McCarthy et al.(1965）和Weissman（1967）对其进行了描述。Steele（1984）对于COMMON LISP进行了描述。Rees和Clinger（1986）讨论了Scheme语言以及它的一些改进和优点。

Dybvig（2003）写了一本获取Scheme程序设计信息的好书。Milner et al.（1990）给出了ML的定义。Ullman（1998）写了一本关于ML的很好的介绍性的教科书。Thompson（1996）介绍了Haskell程序设计。

Henderson（1980）作了关于函数式程序设计的严格讨论。通过图归约实现函数式语言的过程，Peyton Jones（1987）对其进行了详细的讨论。 679

复习题

1. 定义函数形式和引用透明性。
2. 什么数据类型是最初始LISP语言中的一部分？
3. EQ?、EQV?和=之间的区别是什么？
4. 用于Scheme特殊形式DEFINE的求值方法，与用于Scheme原始函数的求值方法之间，有什么不同？
5. DEFINE有哪两种形式？
6. 描述COND的语法和语义。
7. 描述LET的语法和语义。
8. 为什么将命令式特性附加到LISP的大多数方言之中？
9. COMMON LISP与Scheme在哪些方面是相反的？
10. 在Scheme中使用什么样的作用域规则？在COMMON LISP中呢？在ML中呢？在Haskell中呢？
11. 在哪三个方面ML非常不同于Scheme？
12. ML中所用的类型推理是什么？（参见第5章）
13. Haskell中使得它非常不同于Scheme的三种特性是什么？
14. 懒惰求值意味着什么？
15. CONS、LIST和APPEND有什么不同？

练习题

1. 阅读John Backus关于FP的文章（Backus，1978），并且将在本章中讨论过的Scheme的特性与FP的对应特性进行比较。
2. 找出Scheme函数EVAL以及APPLY的定义，并且解释这两种函数的行为。
3. Teitelman和Masinter的文章"The INTERLISP Programming Environment"（*IEEE Computer*，Vol. 14，No. 4，April 1981）中描述到，对于任何语言最现代和最完整的程序设计环境之一是LISP的INTERLISP系统。仔细阅读这篇文章，并且针对使用你的系统来编写LISP程序的难度与使用INTERLISP来编写LISP程序的难度进行比较。（假设你通常并不使用INTERLISP系统。） 680
4. 参考一本关于LISP程序设计的书，确定什么是支持在LISP中包括PROG特性的原因。
5. 一种函数式语言可以使用链表以外的一些数据结构。例如，它可以使用符号序列。如果有这样一种语

言，需要用什么样的原始函数来代替Scheme的原始函数CAR、CDR和CONS？

6. 下面的Scheme函数做些什么？

```
(define (y s lis)
  (cond
    ((null? lis) '() )
    ((equal? s (car lis)) lis)
    (else (y s (cdr lis)))
))
```

7. 下面的Scheme函数做些什么？

```
(define (x lis)
  (cond
    ((null? lis) 0)
    ((not (list? (car lis)))
      (cond
        ((eq? (car lis) nil) (x (cdr lis)))
        (else (+ 1 (x (cdr lis))))))
    (else (+ (x (car lis)) (x (cdr lis))))
))
```

程序设计练习题

1. 编写一个Scheme函数，计算一个球体的体积，球半径是给定的。
2. 编写一个Scheme函数，计算一个给定二次方程式的实数根。如果所计算的根为复数，这个函数必须显示一种指示复数的方法。这个函数必须使用IF函数。而函数的三个参数，就是这个二次方程的三个系数。
3. 重复程序设计练习题2，但是使用COND函数，而不是使用IF函数。
4. 编写一个Scheme函数，这个函数从给定的简单数字链表中返回数字0。

681

5. 编写一个Scheme函数，这个函数将找出给定链表中给定原子的所有顶层实例。
6. 编写一个Scheme函数，这个函数将给定的链表中的最后一个元素移出来。
7. 重复程序设计练习题5，除了其中给定原子或者是一个原子，或者是一个链表之外。
8. 编写一个Scheme函数，它具有三个参数，其中的两个是原子，一个是链表；这个函数将链表中所出现的所有第一个给定的原子都替换成第二个给定的原子，而不论第一个给定的原子在链表中被嵌套的层次有多深。
9. 编写一个Scheme函数来反序排列它的简单链表中的参数。
10. 编写一个Scheme谓词函数，来测试两个给定链表在结构上的相等性。两个被称为结构相等的链表具有相同的链表结构，虽然在这两个链表中的原子可以是不同的。
11. 编写一个Scheme函数，它返回两个表示集合的简单链表参数的并。
12. 编写一个Scheme函数，它具有两个参数，其中的一个是原子，一个是链表。这个函数从链表中将给定原子的所有出现都删除掉，而不论它们出现在链表中的位置多深，并返回删除之后的链表。所返回的链表在删除了原子的位置上，不能够放置任何其他的元素。
13. 编写一个Scheme函数，它取一个链表作为参数，并且删除掉链表顶层的第二个元素，然后返回新的链表。如果所给定的链表没有二个元素，这个函数将返回()。
14. 编写一个Scheme函数，它取一个简单数字链表作为参数，这个函数将链表中的元素按从小到大的顺序重新排列，返回这个重新排序之后的链表。
15. 编写一个Scheme函数，它取一个简单数字链表作为参数，返回这个链表中最大的和最小的数字。
16. 编写一个Scheme函数，它取一个简单链表作为参数；这个函数将给定链表中的元素全部重新组合错位排列，返回这个重新排列之后的链表。

682

第16章　逻辑程序设计语言

这一章的目的是介绍逻辑程序设计的概念以及逻辑程序设计语言，还包括对于Prolog子集的简短描述。谓词演算是逻辑程序设计语言的基础，为此我们将首先介绍谓词演算。接着，我们讨论如何将谓词演算用于自动定理证明的系统。然后是关于逻辑程序设计的一般概述。再接下来，使用了较长的一节来介绍Prolog程序设计语言的基础，其中包括算术运算、表处理和一个追踪工具的用法，这个工具可以帮助调试程序，并且说明Prolog系统的工作机制。最后两节描述Prolog作为逻辑语言的一些问题，以及Prolog语言的一些应用领域。

16.1　概述

第15章讨论了函数式程序设计技术，函数式程序设计与命令式语言一起使用的软件开发方法学有显著的不同。在本章中，我们将描述另外一种不同的程序设计方法学，这种方法使用符号逻辑来表示程序，并且使用逻辑推理过程来产生结果。逻辑程序是说明性的，而不是过程性的；这就意味着仅仅是说明所需要的结果，而不必说明产生这个结果的详细过程。

将某种形式的符号逻辑，用作程序设计语言的程序设计，通常被称为**逻辑程序设计**，而基于符号逻辑的语言，则被称为**逻辑程序设计语言**或**说明性语言**。因为逻辑程序设计语言Prolog是唯一广泛使用的逻辑语言，所以我们在这一章仅描述这一种语言。

逻辑程序设计语言的语法明显地不同于命令式语言和函数式语言的语法。逻辑程序的语义也与命令式语言程序的语义极少相似之处。这些观察会让读者对逻辑程序设计和说明性语言的性质产生一些好奇心。

16.2　谓词演算的简短介绍

在我们讨论逻辑程序设计之前，必须先简略地介绍这种语言的基础，即形式逻辑。这不是我们第一次在这本书中接触形式逻辑；在第3章中，曾经大量使用形式逻辑来描述公理语义。

命题（proposition）可以被认为是一种逻辑陈述，它可能为真也可能为假。命题由对象和对象间的关系所组成。形式逻辑被开发来提供描述命题的一种方法，其目的是为了确定这些形式陈述的命题之真假。

683
↓
684

符号逻辑（symbolic logic）可以被用于形式逻辑中的三种基本需要：表示命题、表达命题之间的关系，以及描述怎样从设定为真的其他命题推理出新的命题。

形式逻辑和数学之间有着紧密的关系。事实上，许多数学问题都能够以逻辑的方式来进行思考。数和集合论的基本公理，是基本命题的集合，这些命题被设定为真。定理，则是从基本命题集合所推断出的一些附加的命题。

用于逻辑程序设计的符号逻辑的特殊形式，被称为**一阶谓词演算**（first-order predicate calculo），我们通常称之为**一阶谓词演算**，虽然这种称法并不十分准确。在本节以下部分，我们将简略地描述谓词演算，我们的目的，是为逻辑程序设计和逻辑程序设计语言Prolog的讨论，提供一种基础。

16.2.1 命题

逻辑程序设计命题中的对象，被表示为简单的项，一个项可以是常量或变量。常量，是表示对象的符号。变量，是在不同时间能表示不同对象的符号，这里的变量，在数学变量与命令式程序设计语言变量之间，更接近于数学变量。

最简单的命题，被称为原子命题（atomic proposition），它由复合项所组成。复合项（compound term），是数学关系的一种元素，数学关系的外观，与数学函数的表示方法相同。回忆曾经在第15章中讨论过的，一个数学函数就是一种映射，可以将它表示为一个表达式或者是元组的表格。复合项就是函数表格定义中的元素。

复合项由两个部分所组成：函数符和一组有序的参数。函数符是命名关系的函数符号。只有一个参数的复合项，是一元组；具有两个参数的则是二元组，等等。例如，我们可以有两个命题：

```
man(jake)
like(bob, steak)
```

这两个命题说明，{jake}是被命名为man（男人）的关系中的一个一元组；而{bob，steak}，是被命名为like（喜欢或者类似）的关系中的一个二元组。如果我们将命题

```
man(fred)
```

加入到上面的那两个命题，那么关系man就具有了两个不同的元素，即{jake}和{fred}。在这些命题里的所有简单项，（man，jake，like，bob和steak）都是常量。注意，这些命题并
685 没有固有的语义。我们希望它们具有什么样的意义，它们就可以具有什么意义。例如，上面第二个例子的意义，可能是bob喜欢（like）牛排（steak），也可以是牛排（steak）喜欢（like）bob，或者是bob在某些方面与牛排（steak）相类似（like）。

命题能够在两种模式中来描述，在第一种模式中，命题被定义为真，而在第二种模式中，命题的真值尚待确定。换句话说，能够将命题陈述为事实或是查询。上面所示例的命题，就可能是事实或查询。

复合命题有两个或多个由逻辑连接符或操作符连接的原子命题，连接的方式与命令式语言中构造复合逻辑表达式的相同。下面是谓词演算逻辑连接符的名字，符号和意义：

名字	符号	示例	意义
非	¬	$\neg a$	非a
合取	∩	$a \cap b$	a与b
析取	∪	$a \cup b$	a或b
等价	≡	$a \equiv b$	a等价于b
蕴涵	⊃	$a \supset b$	a蕴涵b
结论	⊂	$a \subset b$	b结论a

下面是复合命题的例子：

a ∩ b ⊃ c

a ∩ ¬b ⊃ d

操作符¬具有最高的优先级高于，操作符∩，∪和≡，都具有高于⊃和⊂的优先级。因此，上面的第二个例子等价于

$(a \cap (\neg b)) \supset d$

变量可以出现在命题中，但仅当由被称为限定符的特殊符号所引入的条件之下，谓词演算包括了两种限定符，如在下面的描述中所介绍的；在这里，X是一个变量，而P是一个命题：

名字	命题	意义
全称限定符	$\forall XP$	对于所有的X，P都为真，
存在限定符	$\exists X.P$	存在着X的一个值，使得P为真。

在X和P之间的点，只是用来将变量与命题分隔开。例如，考虑下面命题：

$\forall X.(\text{woman}(X) \supset \text{human}(X))$

$\exists X(\text{mother}(\text{mary}, X) \cap \text{man}(X))$

第一个命题的意思是，对于X的任何值，如果X是一个woman（女人），那么X是一个human（人）。第二个命题的意思是，存在着X的一个值，以至于mary是X的mother（母亲），并且X是一个man（男人），换句话说，也即mary有一个儿子。全称限定符和存在限定符的作用域，就是它们所属的原子命题。正如在上面所描述的这两个复合命题中，我们也可以使用圆括号来将这个作用域施行扩展。全称限定符和存在限定符，具有比其他任何操作符都更高的优先级。 686

16.2.2 子句形式

我们之所以讨论谓词演算，是因为它是逻辑程序设计语言的基础。与其他的语言一样，形式最简单的逻辑语言最好，这意味着应该将冗余减少到最低的限度。

到目前为止我们所描述的谓词演算的一个问题，就有着许多不相同的方式，来描述同样含义的命题；也就是说，有很多的冗余。这对于逻辑学家还不是太大的问题，但是如果要将谓词演算用在一个自动化（计算机化的）系统之中，这将会是一个严重的问题。因而我们需要一种谓词的标准形式，以便将事情简单化。一种相对简单的命题形式，子句形式，恰好就是这样的一种标准形式。可以将所有的命题都用子句形式来表达，而不会失去其通用性。一个子句形式的命题，具有下面的通用语法：

$B_1 \cup B_2 \cup \cdots \cup B_n \subset A_1 \cap A_2 \cap \cdots \cap A_n$

这里的A和B是项。这个子句形式的命题意义为：如果所有的A都为真，那么就至少存在着一个B为真。子句形式命题的主要特性是：不需要存在限定符，对于原子命题中的变量使用，全称限定符是隐性的，不需要合取和析取之外的任何操作符。并且，合取和析取只能以通用子句形式中的次序出现：析取出现在左边，合取出现在右边。所有的谓词演算命题，都能够通过算法变换到子句形式。Nilsson（1971）曾经证明了这是能够做到的，并且还存在着一种简单的变换算法。

将子句形式命题的右边称为前件（antecedent），而将左边称为后件（consequent），它是前件为真时的结果。作为子句形式命题的例子，考虑下面的例子：

$\text{likes}(\text{bob}, \text{trout}) \subset \text{likes}(\text{bob}, \text{fish}) \cap \text{fish}(\text{trout})$

$\text{father}(\text{louis}, \text{al}) \cup \text{father}(\text{louis}, \text{violet}) \subset \text{father}(\text{al}, \text{bob}) \cap \text{mother}(\text{violet}, \text{bob})$

$\cap \text{grandfather}(\text{louis}, \text{bob})$

在英文的版本中，上面第一条命题所描述的意思是：如果bob喜欢鱼，并且鳟鱼是鱼，那么bob喜欢鳟鱼。第二条命题描述的意思是：如果al是bob的父亲，violet是bob的母亲，并且 687

louis是bob的祖父，那么，louis如果不是al的父亲，就会是violet的父亲。

16.3 谓词演算与定理证明

谓词演算，提供了一种表达一组命题的方法。一组命题，可以用来确定是否能从它们推理出任何有趣或是有用的事实。这与数学家们的工作完全是类似的。数学家们试图发现是否能够从已知的公理和定理，推断出新的定理来。

在计算机科学的早期（20世纪50年代和60年代的初期），人们对于定理证明过程的自动化，有着极大的兴趣。在自动定理证明中最重要的突破之一，是Syracuse大学的Alan Robinson所发现的归结原理（Robinson，1965）。

归结（resolution）是一条推理规则，它允许从给定的命题中，计算出推理的命题。这样就提供了一种在自动定理证明中，具有潜在应用的方法。归结被设计来应用于子句形式命题。归结的概念介绍如下，假设有下面形式的两个命题：

$P_1 \subset P_2$

$Q_1 \subset Q_2$

它们的意义为，P_2蕴涵P_1，并且Q_2蕴涵Q_1。如果进一步地假设P_1和Q_2相同，我们就可以将P_1和Q_2重新命名为T。那么，我们可以这样来重新编写上面的两个命题：

$T \subset P_2$

$Q_1 \subset T$

现在，因为P_2蕴涵T，并且T蕴涵Q_1，逻辑上很明显：P_2蕴涵Q_1

$Q_1 \subset P_2$

从原有的两个命题，推断出上面这个命题的过程，就是归结。

作为另外的一个例子，考虑下面的两个命题：

older(joanne, jake) ⊂ mother(joanne, jake)

wiser(joanne, jake) ⊂ older(joanne, jake)

688 使用归结，我们就能够从这两个命题构造出下面的命题：

wiser(joanne, jake) ⊂ mother(joanne, jake)

归结构造的机制很简单：将两个命题的左边的项"与"在一起，以产生新命题的左边；将两个命题的右边的项"与"在一起，以产生新命题的右边，然后，删除掉同时出现在新命题两边的项。当命题的一边或两边具有多个项时，归结的过程也完全相同。新命题的左边最初包含了，两个给定命题的左边的所有项；新命题的右边最初包含了，两个给定命题的右边的所有项。然后，删除掉同时出现在新命题两边的项。例如，如果我们有：

father(bob, jake) ∧ mother(bob, jake) ⊂ parent(bob, jake)

grandfather(bob, fred) ⊂ father(bob, jake) ∧ father(jake, fred)

所归结出的命题是：

mother(bob, jake) ∧ grandfather(bob, fred) ⊂ parent(bob, jake) ∧ father(jake, fred)

原来两个命题中的原子命题，除了其中的一个外，都出现在所归结出的命题中。第一个命题中的左边，和第二个命题中的右边的原子命题，父亲（bob，jake），使得归结操作能够进行，然后将被删除。上一个命题的意思为：

如果：bob是jake的父母，而确实，bob不是jake的父亲就是jake的母亲
并且：bob是jake的父亲，jake是fred的父亲，或者bob是fred的祖父
那么：如果bob是jake的父母，并且jake是fred的父亲
那么：要么bob是jake的祖父，或者要么bob是fred的祖父

实际上的归结，要比这些简单例子复杂得多。特别是，在命题中所出现的变量，需要归结的过程为这些变量找到值，以使得匹配过程能够成功。这种为变量确定有用值的过程，就称为合一（unification）。为了进行合一而临时给变量赋值，就称为实例化（instantiation）。

通常，归结过程可能将一个变量实例化为一个值，但如果后来无法完成所必需的匹配，归结过程必须回溯，并将变量实例化为另外一个不同的值。我们将会在 Prolog 的上下文中更多地讨论合一和回溯过程。

归结中一种关键的重要特性，是它能够在一组给定的命题中，发现任何不一致性。这个性质允许了将归结用于证明定理：所用方法是，我们可以将使用谓词演算的定理证明，想像成为一组给定的相关命题，这组命题包括所要证明的定理的非。如果归结发现所要证明的定理的非，与其他的命题不一致，定理就得到证明。这是一种反证法，典型的是，将原有的命题称为假设（hypothese），而将定理的非称为目标（goal）。

在理论上，这是一种有效和有用的过程。然而，归结所需要的时间则可能成为问题。当然当命题的集合为有限的时，归结将会是一个有限的过程，但是在一个大的命题数据库中寻找不一致，所需要的时间可能是很长的。

定理证明是逻辑程序设计的基础。可以将所要进行的计算表示为，一组假设，及使用归结从这些假设推断出的目标，这组假设就是给定的事实与关系。

当将命题用于归结时，只能够使用一种被限制的子句形式，这种形式进一步简化了归结过程。这种特别的命题，就称为霍恩子句（Horn clause）。霍恩子句只能具有两种形式：它们的左边只有一个原子命题，或者左边为空。[○] 有时将一个子句形式命题的左边，称为首（head），具有左边的霍恩子句被称为有首的霍恩子句。有首的霍恩子句被用来描述关系，如：

```
likes(bob, trout) :- likes(bob, fish), fish(trout)
```

通常是使用左边为空的霍恩子句，来陈述事实；我们将这种子句，称为无首的霍恩子句。例如，

```
father(bob, jake)
```

可以使用霍恩子句来描述大部分的命题，然而并不是可以描述所有的命题。

16.4 逻辑程序设计概述

用于逻辑程序设计的语言，被称为说明性语言，这是因为使用这些语言所编写的程序，是由声明组成的，而不是由赋值和控制流程语句组成的。实际上，这些声明就是符号逻辑中的语句，即命题。

逻辑程序设计语言的必要特征之一，是被称为说明语义（declarative semantics）的语义。这种语义的基本概念，即是通过一种简单方法就可以决定每一条语句的意义，而且这并不依赖这一条语句是怎样被用于解决问题的。说明语义比命令式语言的语义简单得多。例如，在一种逻辑程序设计语言中，一个给定命题的意义能从语句的本身准确地确定，而在一种命令

689

690

○ 霍恩子句以Alfred Horn的姓氏命名，Alfred Horn曾经研究过这种形式的子句（Horn，1951）。

式语言中，要确定一条简单赋值语句的语义，我们需要检查局部的声明，需要有语言作用域规则的知识，甚至还可能需要检查其他文件中的程序，以确定这一条赋值语句中变量的类型。如果赋值表达式包含了变量，我们还必须追踪这一条赋值语句以前的程序的执行，以确定那些变量的值。然后，语句所产生的行动结果，还取决于它的运行时上下文。将所有这些与单一语句的简单检查进行比较，由于不需要考虑文本的上下文或执行的顺序，显然，说明语义远比命令式语言的语义简单。因此，说明语义时常被描述为说明性语言相对于命令式语言的优点之一（Hogger，1984，pp.240-241）。

命令式语言和函数式语言中的程序设计，主要是过程的，这意味着程序人员必须知道一个程序要完成什么，并且教计算机怎样来进行所需要的计算。换句话说，是将计算机当作一个服从命令的简单装置。每一种计算都必须具有计算中的每一个细节。一些人相信这是对于计算机进行程序设计的困难的实质。

逻辑程序设计语言中的程序设计是非过程的。这些语言中的程序并不描述究竟如何来计算一种结果，而仅仅是描述这种结果的形式。这里的区别是我们假设计算机系统在某种程度上能够自行决定如何来得到结果。为了给逻辑程序设计语言提供这种能力，我们需要一种简明的方式向计算机提供有关的信息，以及提供计算获得所需结果的推理方法。谓词演算提供了一种与计算机通信的基本形式，而归结则提供了一种推理技术。

一个常常用来说明过程的和非过程的系统之间不同的例子，就是排序。在一种与C++类似的语言中进行排序，我们必须使用C++的程序向一台具有C++编译器的计算机，解释某一种排序算法的所有细节。在这台计算机将C++程序，翻译成为机器码或某种解释性的中间代码后，接受指令产生已经排序了的链表。

在一种非过程的语言中，就只需要描述排序了的链表的特性：它是给定的链表的某一种排列，使得每一对相邻的元素间都满足某一种给定的关系。更加形式的描述则为：假设，将要被排序的链表存放在一个命名为list的数组中，这个数组的下标范围是从1到n。如果是将存放在命名为“old_list”的数组中的元素排序，并将这些排序的元素存放到另外一个命名为“new_list”的分立的数组中，那么，就可以将这种排序的概念表示如下：

691

```
sort(old_list, new_list) ⊂ permute(old_list, new_list) ∩ sorted(new_list)
sorted(list) ⊂ ∀ j such that 1 ≤ j < n, list(j) ≤ list(j+1)
```

这里的permute（交换）old_list是一个谓词，如果它的第二个参数数组，是它的第一个参数数组的一种排列，它将返回真。

从上面的描述中我们可以看到，非过程的语言系统可以产生已经排序的链表。似乎非过程程序设计就像产生一种简明的软件需求说明一样容易。然而事情并没这么简单。仅仅使用归结的逻辑程序，具有机器效率上的严重问题。此外，直到现在人们仍然没有确定，什么是逻辑语言的最好的形式，并且直到现在人们仍然没有在逻辑程序设计语言中开发出，为大型问题建造程序的好方法。

16.5 Prolog的起源

正如在第2章中所描述的，Aix-Marseille大学的Alain Colmerauer和Phillippe Roussel在爱丁堡大学的Robert Kowalski的帮助下，开发了Prolog的基本设计。Colmerauer和Roussel对于自然语言处理感兴趣，而Kowalski则对于自动定理证明感兴趣。这两个大学之间的合作，一直继续到20世纪70年代中期。自此以后，Prolog语言的开发和使用的研究，在这两个地方独

立地进行，结果就产生了Prolog的两种语法不相同的方言。

在爱丁堡和Marseille之外，Prolog的开发，以及其他逻辑程序设计研究的努力，并没有得到太多的注意，直到1981年日本政府发起了一个很大的、称为第五代计算机系统（FGCS）的研究计划（Fuchi，1981；Moto-oka，1981）。这个计划的主要目的之一，是开发智能机器，Prolog被选来作为这项工作的基础。FGCS的公告，唤起了研究人员和美国及一些欧洲国家的政府，对于人工智能和逻辑程序设计的突然的强烈兴趣。

在短短十年以后，FGCS项目就终止了，开始时，人们对于逻辑程序设计和Prolog的潜力，抱有巨大的期望，但最终却极少具有重大意义的发现。这导致了对于Prolog的兴趣与使用的衰减。尽管如此，Prolog仍然不乏自己的应用领域和支持者。 692

16.6 Prolog的基本元素

Prolog现在有着许多不同的方言。可以将这些方言分成几个类型：由Marseille小组所开发的，来自爱丁堡小组的，以及为微型计算机所开发的方言，如由Clark和McCabe（1984）所描述的micro-Prolog，Prolog方言的语法形式，是不相同的。我们不打算描述不同的Prolog方言，或者是这些方言的混合语言的语法，我们只是选择一种特定的广泛可用的方言，这就是在爱丁堡大学开发的方言。有时，将这种语言的形式称为爱丁堡语法。它的第一个实现，是在一台DEC System-10机器上（Warren et al.，1979）。Prolog已经被实现在几乎所有常用的计算机平台上。例如，你可以从Free Software Organization的网站来获取（http://www.gnu.org）。

16.6.1 项

与在其他语言中的程序一样，Prolog的程序由语句的集合所组成。在Prolog中，只有少数几种语句，但是这些语句可能是复杂的。Prolog的所有的语句都由项所构成。

Prolog的项（term），是一个常量、变量或结构。一个常量是一个原子（atom）或整数。原子是Prolog中的符号值。它与LISP中的原子相类似。一个原子则尤为特别，它或者是以小写字母开始的一串字母、数字和下划线（_）；或者是使用单引号所界定的，一串任何可以打印的ASCII的字符。

一个变量是以大写字母开始的一串字母、数字和下划线。变量不是通过声明来与类型相绑定的。将值与类型到变量的绑定，称为实例化。实例化仅仅发生于归结过程中。将一个没有被赋值的变量，称为未实例化的变量。变量的实例化，仅仅持续到它满足了一个完整的目标为止，它涉及一个命题的证明或反证。就语义与使用方面而言，Prolog中的变量仅仅是命令式语言中的变量的“远亲”。

项的最后一个种类，称为结构。结构表示谓词演算的原子命题。它们的一般形式都是相同的：

函数符（参数表）

函数符是任何原子，它被用来标识结构。参数表可能是一组原子、变量或其他的结构。正如下一节中将要仔细讨论的，结构是Prolog中说明事实的方式。也可以将这些结构认为是对象，它们允许使用一些相关的原子来描述事实。在这种意义上，结构也是关系，因为它们描述原子之间的关系。当一个结构的上下文说明它是一个查询（问题）时，结构还是一个谓词。 693

16.6.2 事实语句

我们首先讨论Prolog里用来构造假设或假定信息数据库的语句。从这些语句中，可以推断出新的信息。

Prolog具有两种基本的语句形式，它们分别对应于谓词演算里，无首和有首的霍恩子句。Prolog中无首霍恩子句的最简单形式，是一种单一结构，这种结构被解释成为无条件的断言，即事实。在逻辑上，事实就是被假定为真的命题。

下面的示例，说明一个 Prolog 程序中的事实。注意，Prolog 中的每一条语句都是以句点结尾的。

```
female(shelley).
male(bill).
female(mary).
male(jake).
father(bill, jake).
father(bill, shelley).
mother(mary, jake).
mother(mary, shelley).
```

这些简单结构描述有关jake、shelley、bill和mary的某些事实。例如，第一个结构描述shelley是一位女性。最后的四个结构，使用在函数符原子中所命名的关系，来连接它们的两个参数，例如，可以将第五个命题解释为，bill是jake的父亲。注意，这些 Prolog 命题就像谓词演算中的命题一样，没有固有的语义。它们的语义，就是程序人员赋予它们的语义。例如，命题：

```
father (bill, jake).
```

可以意味着bill和jake有相同的父亲，或者jake是bill的父亲，然而最通常并且也是最直接的意义为，bill是jake的父亲。

16.6.3 规则语句

Prolog中用来构造数据库的、另外一种基本形式的语句，对应于有首的霍恩子句。这种形式，与一个已知的数学定理相关联：如果一组给定的条件被满足的话，就可以从这个已知的定理得出一个结论。右边是前件，即如果（if）部分，左边是后件，即那么（then）部分。如果一条Prolog语句的前件为真，那么语句的后件也必定为真。因为它们都是霍恩子句，Prolog语句的后件是单个的项，而前件则可能是单个的项或合取的项。

604 合取包含被逻辑“与”运算所分开的多个项。在 Prolog中，“与”运算是隐式的。在合取中说明原子命题的结构，由逗点分开；因此我们可以认为，逗点就是“与”运算符。作为合取的一个例子，考虑：

```
female (shelley), child (shelley).
```

Prolog 的有首霍恩子句语句的一般的形式为：

后件_1 : - 前件表达式

可以将它读为：如果前件表达式为真，或通过它的变量实例化使得它为真，则后件为真。例如，

```
ancestor (mary, shelley) :- mother (mary, shelley).
```

描述的是，如果mary是shelley的母亲，则mary是shelley的一位前辈。将有首的霍恩子句称为规则，因为它们描述命题之间的蕴涵规则。

和谓词演算中的子句形式命题一样，Prolog中的语句能够通过使用变量，来将它们的含义一般化。回忆，子句形式中的变量，提供了某种隐式的全称限定符。下列的程序段，示范了Prolog语句中的变量使用：

```
parent(X, Y) :- mother(X, Y).
parent(X, Y) :- father(X, Y).
grandparent(X, Z) :- parent(X, Y) , parent(Y, Z).
sibling(X, Y) :- mother(M, X) , mother(M, Y),
                 father(F, X) , father(F, Y).
```

这些语句给出了变量或通用对象之间的蕴涵规则。这里的通用对象是X、Y、Z、M和F。第一条规则所描述的是，如果有X和Y的实例化导致了mother（X，Y）为真，则对于X和Y的那些相同的实例化，将导致parent（X，Y）为真。

16.6.4 目标语句

到目前为止，我们已经描述了逻辑命题的Prolog语句，这些语句被用来描述已知的事实，以及这些事实之间的逻辑关系，这些语句是定理证明模型的基础。定理是以一种命题的形式出现，我们希望系统地来证明或反证这些命题。在Prolog中，将这些命题称为目标或查询。 695
Prolog目标语句的语法形式，与无首的霍恩子句相同。例如，我们可以有：

```
man (fred).
```

系统对于这个命题的回应，或者是yes，或者是no。回答yes，意味着系统根据给定的事实以及关系的数据库，证明目标为真。回答no，则意味着系统证明了目标为假，或者系统不能够证明它。

合取的命题和有变量的命题，也都是合法的目标。当有变量时，系统不但需要证明目标的真假，还需要识别导致目标为真的变量的实例化。例如，可以使用：

```
father (X, mike).
```

来提出问题。然后系统将会使用合一，来试图找出导致目标为真的X的实例化。

因为目标语句和一些非目标语句，具有着相同的形式（无首的霍恩子句），Prolog的实现必须具有某种方式来区别这两者。交互式的Prolog实现，使用两种模式来进行区别，并且使用不同的交互提示，来指示这两种模式：一种模式是用来输入事实和规则语句，另外一种则用来输入目标。用户在任何时候都可以改变模式。

16.6.5 Prolog的推理过程

这一节讨论Prolog的归结。为了有效率地使用Prolog，程序人员需要精确地知道Prolog系统是怎样处理其他程序的。

将查询称为目标（goal）。当一个目标是一个复合命题时，每一种事实（结构）都被称为子目标（subgoal）。要证明一个目标为真，推理过程则必须在数据库中找到一条推论规则以及一条事实链，这条链将目标连接到数据库中的一个或多个事实之上。例如，如果Q是目标，那么，或者Q必须在数据库中被发现是一个事实，或者这个推理过程必须找到事实P_1，以及命题

的一个序列，$P_2, P_3, \cdots, P_n$，以至于

$P_2 :- P_1$

$P_3 :- P_2$

...

$Q :- P_n$

696 当然，规则右边的复合结构以及规则中的变量，会使得推理过程复杂化。发现那些P的过程，基本上是项的比较或匹配的过程。

因为证明一个子目标的过程，是一个命题匹配的过程，有时将它就称为匹配。在某些情况下，将证明了目标称为满足了目标。

考虑下面的查询：

```
man (bob).
```

这是一条最简单类型的目标语句。归结可以相对容易地确定它是真还是假：将这个目标的模式，与数据库中的事实和规则进行比较。如果数据库中包括了事实：

```
man (bob).
```

那么证明就是十分容易的。然而，如果数据库中包含了下面的事实以及推理规则：

```
father (bob).
man (X):-father(X).
```

Prolog首先需要找到这两条语句，并使用它们来推断目标的真实性。这就需要使用合一，从而将x暂时实例化为bob。

现在考虑下面的目标：

```
man (X).
```

在这种情况下，Prolog必须将目标与数据库中的命题匹配。它所找到的第一个命题，应该与目标具有相同的形式，并且是以任何对象作为参数。这个命题将会使得X被实例化为这个对象的值。然后将X显示为结果。如果没有与目标形式相同的命题，系统就会回答no，以说明目标不能够被满足。

企图将一个给定目标与数据库中的事实相匹配，可以有着两种相反的方式。系统能够从数据库的事实及规则开始，试图来找到导致目标的一个匹配序列。将这种方式称为**自底向上归结**（bottom-up resolution）或**前向链接**（forward chaining）。另外一种方式是从目标开始，试图来找到一个匹配命题序列，这将会被引向数据库中原有事实的某一个集合。将这种方式称为**自顶向下归结**（top-down resolution）或**后向链接**（backward chaining）。大体上，当候选的答案集合是当小的时候，后向链接较适用。当候选答案的数目相当大时，前向链接较适用；在后一种情形下，后向链接将会需要非常大量的匹配，才能够得出答案。Prolog的实现，697 使用后向链接来进行归结；我们推测这是因为它的设计人员相信，对于大多数问题，后向链接比前向链接更加适用。

下面例子说明前向链接和后向链接的不同之处。考虑查询：

```
man (bob).
```

假设数据库包含了：

```
father (bob).
man (X):- father (X).
```

前向链接将会寻找并找到第一个命题。然后通过将X实例化为bob，来实现第一个命题与第二条规则的右边（father（X））的匹配，最后再将第二个命题的左边，与目标进行匹配。后向链接则会首先通过将X实例化为bob，从而使目标与第二个命题的左边（man（X））相匹配。最后的步骤，它将会使第二个命题的右边（现在是father（bob）），与第一个命题相匹配。

当目标具有多种结构时，就像在我们下面的例子中的那样，会出现下面的这种设计问题。这个问题就是：究竟是使用深度优先，还是宽度优先来寻找答案。深度优先搜索在为第一个子目标找到一个完整的命题序列（即证明）之后，再处理另一个子目标。宽度优先的搜索，并行地工作于一个给定目标的所有子目标。Prolog的设计人员，之所以主要选择深度优先的方式，是因为深度优先只需要较少的计算机资源。宽度优先的方式是并行搜索，这种方式使用很多的内存空间。

必须讨论的Prolog归结机制的最后一个特性，就是回溯。当系统处理一个具有多个子目标的目标时，当如果无法证明某个子目标的真值时，系统将会放弃所不能够证明的子目标。然后系统将重新考虑前一个子目标，如果这个子目标存在的话，而且将试图找到其他的解决办法。将这种回到前面的子目标，称为回溯。当停止一个子目标时，系统将回到其起始点，从那里再重新开始搜索新的解决方法。一个子目标变量的不同实例化，可能导致这个子目标的多种解决办法。因为回溯可能必须找到每一个子目标的所有可能的证明，所以回溯将会需要很多的时间和空间。这些子目标的证明，可能没有很好地组织起来，以便尽量地减少找到最终的完整证明的时间，这种情形就使得问题更加糟糕。

考虑下面的这个例子可以加强对于回溯的理解。假设，在一个数据库中有一组事实及规则，并且给予了Prolog下面这个复合目标：

```
male (X), parent (X, shelley).
```

698

这个目标是在问，是否具有X的一个实例化以使得X为一个男性，而且X是shelley的父母。Prolog首先将在数据库中找函数符为male（男性）的第一个事实，然后它将X实例化为所找到事实的参数，例如mike。然后它再试图证明parent（mike，shelley）为真。如果证明失败，它将回溯到它的第一个子目标male（X），并尝试使用X的某一个其他可能的实例化，来满足这个子目标。在找到一个是shelley的父母的男性之前，归结过程可能必须搜索数据库中的每一个“male”。它肯定必须查找所有的“male”，以便证明目标不能够被满足。请注意，如果将我们的例子中的两个子目标的次序颠倒，这个例子目标的处理可能会更有效率。这样，就会首先找到shelley的父母，再试图将它与“male”子目标相匹配。如果在数据库中shelley的父母比男性少（这是一个合理的假设），这样效率就更高。我们将在16.7.1小节里讨论，Prolog系统中用来限制回溯的方法。

在Prolog中数据库搜索总是按从第一条到最后一条语句的方向进行。

下面的两个小节，描述Prolog中的一些示例，以便更进一步地说明归结过程。

16.6.6 简单算术

Prolog支持整数变量和整数算术。最初，算术运算符就是函数符。所以7和变量X的和被写为：

```
+ (7, X)
```

Prolog现在允许一种带有is操作符的更简略的算术语法。这个操作符的右边操作数是一个算术表达式，左边操作数是一个变量。表达式中的所有变量必须已经实例化，但是左边的变

量不能是已实例化的。例如，

```
A is B / 17 + C
```

如果B和C都被实例化，但A还没有，那么，这个子句会将A实例化为右边表达式的值。当这种情况发生时，子句被满足。如果B或C没有被实例化，或者A已经被实例化，上面子句不被满足，并且将不会发生A的实例化。is命题的语义，与命令式语言中赋值语句的语义，非常不同。这种不同可能导致一种有趣的情形。因为is操作符使得它所在的子句，看起来就像赋值语句一样，Prolog语言的初学者也许会写出这样的一条语句：

699

```
Sum is Sum + Number.
```

这条语句在Prolog中是无用的，并且还是不合法的。如果"Sum"没有被实例化，右边对它的引用就是无定义的，因而子句注定会失败。因为在计算is操作符时，左边的操作数不能是已经实例化的，所以如果"Sum"已经被实例化，子句就会失败。因此在任何情况下，都不可能将"Sum"实例化为一个新的值。(如果需要"Sum+Number"的值，就可以将它绑定到某一种新名字上。)

Prolog中没有与命令式语言相同意义的赋值语句。在Prolog被设计来进行的大部分程序设计，并不需要赋值语句。在命令式语言中，赋值语句的实用价值，依赖于程序人员控制嵌有赋值语句的代码的执行控制流的能力。因为在Prolog中，这种类型的控制并非总是可能的，所以这种赋值语句就远没那么有用。

作为在Prolog中使用数值计算的一个简单的例子，考虑下面问题：假如我们知道在一条特定跑道上，一些汽车的平均速度，而且知道这些汽车在跑道上的时间，我们就能够将这些基本信息编码为事实，而可以将速度、时间和距离之间的关系编写为规则，如下面所示：

```
speed(ford, 100).
speed(chevy, 105).
speed(dodge, 95).
speed(volvo, 80).
time(ford, 20).
time(chevy, 21).
time(dodge, 24).
time(volvo, 24).
distance(X, Y) :- speed(X, Speed),
                  time(X, Time),
                  Y is Speed * Time.
```

现在，我们就可以查询某一辆特定汽车旅行的距离，例如，查询：

```
distance (chevy, chevy_Distance).
```

实例"chevy_Distance"的值为2205，距离计算语句右边的前面两个子句，仅仅是将变量"Speed"和"time"、实例化为给定汽车函数符中的对应值，在满足了目标后，Prolog也将显示名字"chevy_Distance"和它的值。

现在从操作的角度，来审视一个Prolog系统是怎样生产结果的，将是很有帮助的。Prolog具有一个被命名为trace的内建结构：在试图满足给定目标的期间，这种内建结构将显示从变量到值的实例化过程中的每一个步骤。Trace被用来理解和调试Prolog的程序。为了更好地了解trace结构，最好先介绍Prolog程序执行的一种不同的模型，它被称为追踪模型。

700

追踪模型用四个事件来描述Prolog的执行：(1) 调用 (call)，发生在开始试图满足一个目标时；(2) 退出 (Exit)，发生在一个目标已经被满足时；(3) 重做 (redo)，发生在回溯引起试图重新满足一个目标时；(4) 失败 (fail)，发生在一个目标失败时。如果将像上面“距离”一样的过程，视为子程序，调用与退出的事件，就能够与命令式语言中子程序的执行模式，直接关联。另外的两个事件，是逻辑程序设计系统所独有的。在下面的追踪示例中，这里的目标不要求重做或失败事件。

下面是对于计算Chevy_Distance值的一个追踪：

```
trace.
distance (chevy, chevy_Distance).

(1) 1 调用: distance(chevy, _0)?
(2) 2 调用: speed(chevy, _5)?
(2) 2 退出: speed(chevy, 105)
(3) 2 调用: time(chevy, _6)?
(3) 2 退出: time(chevy, 21)
(4) 2 调用: _0 is 105*21
(4) 2 退出: 2205 is 105*21
(1) 1 退出: distance(chevy, 2205)

Chevy_Distance = 2205
```

在追踪里由下划线字符 (_) 所开始的符号，是用来存储实例化值的内部变量。第一列指示当前正在试图匹配的子目标。例如在上面的追踪里，具有(3)的第一行，是试图将临时变量 _6 实例化为chevy的一个“time”的值，这里的“time”，是描述“distance”计算语句中右边的第一个项。第二列则指示匹配过程的调用深度。第三列指示当前的动作。

为了举例说明回溯，考虑下面的数据库以及追踪的复合目标示例：

```
likes (jake,chocolate).
likes (jake,apricots).
likes (darcie,licorice).
likes (darcie,apricots).

trace.
likes (jake,X), likes (darcie,X).

(1)  1 调用: likes (jake, _0)?
(1)  1 退出: likes (jake, chocolate)
(2)  1 调用: likes (darcie, chocolate)?
(2)  1 失败: likes (darcie, chocolate)
(1)  1 重做: likes (jake, _0)?
(1)  1 退出: likes (jake, apricots)
(3)  1 调用: likes (darcie, apricots)?
(3)  1 退出: likes (darcie, apricots)

X = apricots
```

我们可以使用图形的方式想像Prolog的计算：将每一个目标视为一个具有四个端口的方格，这四个端口就是调用、失败、退出和重做。控制通过调用端口，按照向前的方向进入一个目标。控制也可以通过重做端口，按照向后的方向进入一个目标。控制能够以两种方式来离开一个目标：如果目标成功，控制将通过退出端口离开；如果目标失败，控制则将通过失败端

701

口离开。图16-1显示了上面示例的一种模型。在这个示例中，控制流程经过每一个子目标两次。第二个子目标在第一次失败后，控制通过重做，返回第一个子目标。

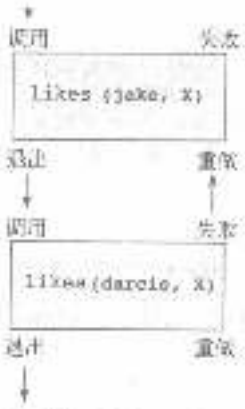


图16-1 用于目标：likes(jake, X), likes(darcie, X)的控制流程模型

16.6.7 链表结构

到目前为止，我们所讨论的唯一的Prolog数据结构，是原子命题，这种结构看起来更像是一个函数调用，而不像一个数据结构。原子命题，也称为结构，实际上是一种记录的形式。Prolog支持的另外一种基本的数据结构，是链表，它与LISP使用的链表结构相类似。链表是
202 任意数目的元素的序列，这些元素可以是原子、原子命题或任何其他的项，也包括其他的链表。

Prolog使用ML和Haskell的语法，来说明链表。使用逗点将元素分隔开，并且使用方括号来给整个链表定界，如：

```
[apple, prune, grape, kumquat]
```

标记法[]，用来表示空链表，Prolog不具有用来构造和拆散链表的显式函数，而是使用一种特别的标记法。[X | Y] 表示一个头为X，尾为Y的链表，这里的头和尾，与LISP中的CAR和CDR相对应，这种方法，也与Haskell和ML中使用的表示法相类似。

一个链表能够与一个简单结构一同被创建，如：

```
new_list ([apple, prune, grape, kumquat]).
```

这说明常量链表[apple, prune, grape, kumquat]，是被命名为new_list（新链表）的关系中的一个新元素（我们刚刚才创造出的一个名字）。这一条语句，并没有将称为new_list的变量与这个链表相绑定，相反，它所从事的，是命题所从事的。如：

```
male(jake)
```

也就是说，它描述[apple,prune,grape,kumquat]是“新链表”的一个新元素。因此，我们可以具有带链表参数的第二个命题，如：

```
new_list([apricot, peach, pear])
```

在查询模式里，我们可以使用：

```
new_list([New_List_Head|New_List_Tail])
```

将“new_list”的元素之一，拆散为头与尾。

如果“new_list”具有上面的两个元素，这个语句将“New_List_Head”实例化为第一个链表元素的头（在前面的例子里是apple），并且将“New_List_Tail”实例化为这个链表的尾（也即[prune, grape, kumquat]）。如果这是复合目标中的一部分，而且回溯强迫对它重新求值，“New_List_Head”和“New_List_Tail”将会分别为apricot和[peach, pear]。因为[apricot, peach, pear]是“new_list”的下一个元素。

这个用来拆散链表的操作符“|”，也可以被用于从给定实例化的头和尾，产生出链表，如
203

```
[Element | List_2]
```

如果将Element 1实例化为pickle，而将List_2被实例化为 [peanut, prune, popcorn]，上面的这个标记法，将会产生链表 [pickle, peanut, prune, popcorn]。

如上面所述，包括了"|"符号的链表标记法是通用的，能够使用它来说明构造一个链表，也能够使用它来说明拆散一个链表，注意，下面的各项是相等的：

```
[apricot, peach, pear | []]
[apricot, peach | [pear]]
[apricot | [peach, pear]]
```

当处理链表时，常常需要某些基本的操作，如在LISP，ML和Haskell中的那些操作。作为Prolog中的这些操作的一个示例，我们来考虑**append**的定义，它与 LISP 中的这种函数相关。在这个例子中，我们将能够看到函数式语言和说明性语言之间的不同和相类似之处。我们并不需要说明Prolog应该如何从给定的链表，来构造一个新链表；我们只需要就给定的链表，来说明新链表的特性。

从外表上看，Prolog中的append定义，非常类似于在第15章中出现的ML中的版本，并且归结中的某种递归，还是以相类似的方式来生产新的链表。在Prolog的情况下，递归由归结过程所引起，并由归结过程来控制。如同在ML和Haskell中的一样，模式匹配过程被用来在两个不同的append过程的定义之间进行选择，这种选择是基于实参的。

在下面代码中，append操作中的前两个参数，是两个将要被结合在一起的链表，第三个参数是结果链表：

```
append([], List, List).
append([Head | List_1], List_2, [Head | List_3]) :-
            append(List_1, List_2, List_3).
```

第一个命题，说明当将空链表结合到任何其他的链表上时，其他的链表就是结果。这条语句对应于ML中的append函数的递归-终结步骤。注意终结命题被放在递归命题前。这是因为Prolog将会从第一个开始，依照顺序来匹配这两个命题（因为它使用的是深度优先）。

第二个命题，说明新的链表的一些特征。它对应于ML函数中的递归步骤。左边的谓词，描述新链表中的第一个元素与第一个给定链表中的第一个元素相同；这两个第一个元素都命名为Head。每当将Head实例化为一个值时，实际上目标中Head的所有出现，都被实例化为同一个值。第二条语句的右边，说明将第二个给定的链表List_2附加到第一个给定链表List_1的尾部，形成结果链表List_3的尾。

可以将append的第二条语句这样来解读：仅当List_3是由List_1附加到List_2所形成时，将链表 [Head | List_1] 附加到任意链表List_2，产生链表[Head |List_3]。在LISP中，这就是：

```
(CONS (CAR FIRST) (APPEND (CDR FIRST) SECOND))
```

在Prolog和LISP这两种语言的版本中，一直到递归产生终结条件时，才会构造出结果链表；此时，第一个链表必定已经成为空表。然后再使用append函数本身，来建造结果链表，从第一个链表所取出的元素，以相反的次序加入到第二个链表中。

为了说明append是怎样工作的，考虑下面的追踪例子：

```
trace.
append([bob, jo], [jake, darcie], X).

(1) 1 调用: append([bob, jo], [jake, darcie], _10)?
(2) 2 调用: append([jo], [jake, darcie], _18)?
```

704

```
(3) 3 调用: append([ ], [jake, darcie], _25)?
(3) 3 退出: append([ ], [jake, darcie], [jake, darcie])
(2) 2 退出: append([jo], [jake, darcie], [jo, jake, darcie])
(1) 1 退出: append([bob, jo], [jake, darcie], [bob, jo, jake, darcie])
family = [bob, jo, jake, darcie]
yes
```

前面的两个调用表示子目标，其中的list_1非空，所以就从第二条语句的右边来产生递归调用。第二条语句的左边，有效地说明了递归调用，或目标中的参数，这样就将第一个链表拆散，一步一个元素。在第一个调用或第一个子目标中，当第一个链表变为空时，第二条语句的右边的当前实例，通过匹配第一条语句而获得成功，这样做的效果，是将空链表值附加到第二个最初的参数表上，并将它作为第二个参数返回。连续的退出表示成功的匹配，其中将从第一个链表取出的元素，附加到结果链表Family上。当从第一个目标退出完成时，过程也就完结，并显示出结果链表。

也可以使用append命题来创建其他的链表操作，如下面的这种操作：我们请读者来确定它的功能。注意，list_op_2具有一个链表来作为它的第一个参数，以及它具有一个变量来作为它的第二个参数；list_op_2的结果，就是第二个参数被实例化的值。

705

```
list_op_2([], []).
list_op_2([Head | Tail], List) :-
list_op_2(Tail, Result), append(Result, [Head], List).
```

正如读者可能已经确定了的，list_op_2导致Prolog系统将它的第二个参数实例化为一个链表，这个链表将具有第一参数的链表中元素，但是次序是相反的。例如，([apple, orange, grape], Q)，将Q实例化为链表[grape, orange, apple]。

这又一次说明了，虽然LISP和Prolog语言的本质是不同的，但仍然能够使用相类似的方法进行相类似的操作。在颠倒操作的情况下，Prolog中的list_op_2，和LISP中的reverse函数，都包括了递归终结条件，以及将链表颠倒的CDR，或者是将尾附加到链表的CAR，或链表的头的基本过程，这两种方法在一起构造结果链表。

下面是这个过程的追踪，现在将它命名为reverse：

```
trace.
reverse([a,b, c],Q).

(1) 1 调用: reverse([a, b, c], _5)?
(2) 2 调用: reverse([b, c], _65636)?
(3) 3 调用: reverse([c], _65646)?
(4) 4 调用: reverse([ ], _65656)?
(4) 4 退出: reverse ([ ], [ ])
(5) 4 调用: append([ ], [c], _65646)?
(5) 4 退出: append([ ], [c], [c])
(3) 3 退出: reverse([c], [c])
(6) 3 调用: append([c], [b], _65636)?
(7) 4 调用: append([ ], [b], _25)?
(7) 4 退出: append([ ], [b], [b])
(6) 3 退出: append([c], [b], [c, b])
(2) 2 退出: reverse([b, c], [c, b])
```

```
(8)  2 调用: append([c, b], [a], _6)?
(9)  3 调用: append([b], [a], _32)?
(10) 4 调用: append([ ], [a], _38)?
(10) 4 退出: append([ ], [a], [a])
(9)  3 返回: append([b], [a], [b, a])
(8)  2 返回: append([c, b], [a], [c, b, a])
(1)  1 返回: reverse([a, b, c], [c, b, a])

Q = [c, b, a]
```

706

假设，我们需要确定所给定的符号是否在给定的链表中。一种直截了当的Prolog的描述是：

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

这里的下划线表示匿名变量，用来表示我们不在乎从合一过程得到什么实例化的变量。如果"Element"是链表的头，不论原先就是、还是经过多次递归第二条语句之后，上面的第一条语句将成功。如果"Element"是在链表的尾，则第二条语句成功。考虑下面的追踪例子：

```
trace.
member (a, [b, c, d]).
(1)  1 调用: member(a, [b, c, d])?
(2)  2 调用: member(a, [c, d])?
(3)  3 调用: member(a, [d])?
(4)  4 调用: member(a, [ ])?
(4)  4 失败: member(a, [ ])
(3)  3 失败: member(a, [d])
(2)  2 失败: member(a, [c, d])
(1)  1 失败: member(a, [b, c, d])
no

member (a, [b, a, c]).
(1)  1 调用: member(a, [b, a, c])?
(2)  2 调用: member(a, [a, c])?
(2)  2 退出: member(a, [a, c])
(1)  1 退出: member(a, [b, a, c])
yes
```

16.7 Prolog的缺陷

虽然Prolog是一种有用的工具，但它既不是一种纯粹的，也不是一种完美的逻辑程序设计语言。这一小节将讨论Prolog中的一些问题。

16.7.1 归结次序控制

因为效率的原因，Prolog允许用户在归结过程中控制模式匹配的次序。在纯逻辑程序设计的环境中，归结过程中所尝试的匹配次序是不确定的，所有的匹配都可能被并行地尝试。然而，因为Prolog总是以相同的次序来进行匹配，匹配并给予数据库的首部以及给定目标的最左边，所以用户能够通过将数据库语句排序，来优化一种特定的应用，从而对效率产生巨大影响。例如，如果用户知道在一个特定的"执行"期间，某一些规则比其他的规则更有可能成

707

功；那么，将这些规则放在数据库的首部，就可以提高程序效率。

程序执行的速度慢，还不是Prolog程序中用户定义次序的唯一的负面结果。另外一个问题，是非常容易写出无限循环语句，从而导致整个程序的失败。例如，考虑下面的递归语句形式：

```
f(X, Y) :- f(Z, X), g(X, Z).
```

因为在 Prolog 中不论语句的目的是什么，总是按照从左往右深度优先的次序来进行计算，上面语句将会引起一个无限循环。作为这种语句的一个例子，考虑：

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

在试图满足第二个命题的右边的第一个子目标时，Prolog将Z实例化，以使得ancestor为真。然后再试图去满足这个新的子目标，系统将立刻回到ancestor的定义，并重复相同的过程，导致无穷尽的递归。

这种特定的问题，与将一个递归下降语法分析器，用于有左递归的文法规则时的问题相同，这个问题曾经在第3章中讨论过。同语法分析中的文法规则的情形一样，只需要颠倒上面命题中右边项的次序，就会将这个问题消除。这种做法的麻烦，是仅仅改变项的次序，应该不至于影响到程序的正确性，无论如何，不需要程序人员关心控制次序，被认为是逻辑程序设计的优点之一。

除了允许用户控制数据库和子目标次序之外，Prolog 还做出了另外的一种迁就，以便提高效率，这就是允许一些对于回溯的显式控制。这是通过使用切口（cut）操作符来进行的，切口操作符由惊叹号（!）说明。切口操作符实际上是一个目标，而不是一个操作符。作为一个目标，它总是立刻成功的，但是却不可能通过回溯来重新满足。因此切口的一个副作用，就是在一个复合目标中，切口左边的子目标也不可能通过回溯来重新满足。例如，在下面目标中：

```
a, b, !, c, d.
```

如果a和b都成功，但c失败，那么整个目标也就失败。如果我们知道，只要c失败，对于a或b的重新满足只是浪费时间而已，那么我们就可以使用这个切口目标，来避免对于a或b的重新满足。

708

切口操作符的目的，就是允许用户告诉系统，不要去试图重新满足那些不可能产生完整证明的子目标，从而提高程序的效率。

作为切口操作符用法的一个例子，考虑来自16.6.7小节的member规则，即：

```
member(Element, [Element | _]).
member(Element, [_ | List]) :- member(Element, List).
```

如果member的链表参数表示一个集合，那么它将只被满足一次（集合不包含重复的元素）。因此，如果member在一个多子目标的目标语句中，被当作子目标使用的话，就可能存在一个问题。这个问题就是，如果member成功，但下一个子目标失败，回溯将会继续一个先前的匹配，企图重新满足member。但因为member的链表参数开始时只有元素的一个副本，虽然可以继续尝试以重新满足member，member却不可能再一次成功，最终导致整个目标的失败。

例如，考虑目标

```
dem_candidate(X) :- member(X, democrats), tests(X).
```

该目标决定一个给定的人是否是民主主义者，是否是某些特定职位的好的候选者。为了发现这个人适合的职位，tests子目标检测给定的民主主义者的许多特征。如果民主主义者

集合没有副本，并且tests子目标检测失败，那么我们不想转到member子目标，因为没有副本，member搜索所有其他民主主义者也会失败。member子目标的再次搜索只会白白浪费计算时间。对于这个问题的解决办法，是给member定义的第一条语句，增加一个右边，而切口操作符就是这个右边中的唯一元素，如：

```
member(Element, [Element | _]) :- !.
```

回溯将不会企图重新来满足member，而是使整个子目标失败。

在Prolog中一种称为产生与测试（generate and test）的程序设计策略中，切口操作特别有用，在这些程序中，目标由子目标所组成，这些子目标产生潜在的解决方法，这些方法被以后的“测试”子目标所测试。被拒绝的子目标需要回溯到“产生者”子目标，它产生新的潜在的解决方法。作为一个产生与测试程序的例子，考虑下面的规则，这个规则出现在Clocksin and Mellish（1997）文献中：

```
divide(N1, N2, Result) :- is_integer(Result),
                          Product1 is Result * N2,
                          Product2 is (Result + 1) * N2,
                          Product1 =< N1, Product2 > N1, !.
```

709

这个程序使用加法和乘法来进行整数除法，因为大多数的Prolog系统提供除法操作符，这个程序实际上并没有用，这里只是用来举例说明简单的产生与测试程序。

只要谓词is_integer的参数，能够被实例化为一个非负的整数，is_integer就成功。如果它的参数没有被实例化，is_integer将它实例化为值0。如果参数已经被实例化为一个整数，is_integer将它实例化为下一个更大的整数值。

因此在divide中，is_integer是产生器子目标。它产生序列0,1,2,…中的元素，每一次is_integer被满足时便产生一个。所有其他的子目标都是测试子目标，它们检测is_integer所生产的值，是否为前两个参数N1和N2的商。使用切口来作为最后一个子目标的目的，很简单：为了避免divide在找到答案之后，又试图再寻找其他的答案。虽然is_integer能够产生一个巨大数目的候选答案，但其中只有一个是答案，这样，切口操作就避免了再产生第二个答案的无意义的企图。

切口操作符的使用，类似于命令式语言中goto的使用（Van Emden，1980）。虽然有时是必需的，但也可能被滥用。的确，有时是使用它来使逻辑程序具有命令式程序设计风格的控制流程。

逻辑程序设计的重要优点之一，是不需说明如何来解决问题，然而干预控制流程的能力直接损害了这个优点，所以在一个Prolog程序中干预控制流程的能力，是一种缺陷。相反，这种程序只需要说明问题的答案。这使得程序更容易编写和阅读。它们不需要说明如何解决问题的细节，尤其不需要说明计算的精确步骤。虽然逻辑程序设计并不需要控制流程，但主要是为了高的效率，Prolog程序还是时常使用它们。

16.7.2 封闭世界假设

Prolog归结的性质，有时会产生误导的结果。就Prolog语言而言，唯一的真值是那些能够从它的数据库得到证明的。Prolog不具有它的数据库之外的世界的知识，对于任何一个查询，如果在数据库中没有充分的信息来证明，就认为是假的。Prolog能够证明一个给定的目标为真，但是它不能够够证明一个给定的目标为假。它仅仅假设，因为它不能够证明一个目标为真，

所以这个目标必定为假。Prolog实质上是一个“真与失败”的系统，而不是一个“真与假”的系统。

实际上，你对于封闭世界的假设并不陌生，它与我们的司法系统有着相同的操作方式。
710 嫌疑犯是无罪的，直到证明了他有罪；嫌疑犯不需要被证明是无罪的。如果审判不能够证明一个人有罪，他即被认为是无罪的。

封闭世界假设的问题与否定问题相关，关于这些将在下一小节中讨论。

16.7.3 否定问题

Prolog存在的另外一个问题，是处理否定时的困难。考虑下面的这个数据库，它具有两个事实和一个关系：

```
parent(bill, jake).
parent(bill, shelley).
sibling(X, Y) :- (parent(X, X), parent(M, Y).
```

现在，假设我们做人查询：

```
sibling (X, Y).
```

Prolog系统将回答：

```
X = jake
Y = jake
```

因此，Prolog系统“认为”jake是他自己的兄妹。这是因为系统开始将M实例化为bill，并将X实例化为jake，以便使得第一个目标，parent (M, X)，为真。然后它再一次从数据库的起始位置开始匹配第二个子目标，parent (M, Y)，并且将M实例化为bill，以及将Y实例化为 jake。因为两个子目标被独立地满足，并且由于这两个匹配都开始于数据库的起始位置，所以就出现了上面的答案。为了避免这种结果，我们必须说明：X是Y的兄妹，仅当他们都有相同的父母时，并且他们不是同一个人。但是，在Prolog中来描述不相等，不是一件十分容易的事，我们将在后面会讨论这一点。最准确的方法，需要为每一对原子增加一条事实，以描述它们的不相同。这种方法当然会导致数据库变得非常庞大，因为通常，否定信息远远多于正面信息。例如对于大多数的人，364天都不是生日，只有一天是生日。

一种简单的替代方案，是在目标中指述X不能够与Y相等，如：

```
sibling (X, Y) :- parent (M, X), parent (M, Y), not(X = Y).
```

在其他的情况下，这种解决方案并不是这么简单。

在上面的情况下，如果归结不能够满足子目标X = Y，Prolog中的not操作符就被满足，因此如果not成功，并不一定就意味着X不等于Y；相反，它仅仅意味着归结不能够从数据库显证
711 明X等于Y。因此，Prolog的not操作符并不等于，逻辑NOT操作符，后者意味着可以证明它的操作数的真实性，是为真的。如果我们碰巧有下面形式的目标，这种非等价性则会产生问题：

```
not(not(some-goal)).
```

等价于

```
Some_goal.
```

如果Prolog中的not操作符，是真正的逻辑NOT操作符，上面目标会等价于某个目标。

然而，在某些情况下它们是不相同的，例如，再一次来考虑member规则：

```
member(Element, [Element | _]) :- !.
member(Element, [_ | List]) :- member(Element, List).
```

为了能够找出给定链表中的一个元素，我们可以使用下面目标：

```
member(X, [mary, fred, barb]).
```

它将使得X实例化为mary，然后再打印出来。但如果我们是使用：

```
not(not(member(X, [mary, fred, barb]))).
```

将会发生下面的事件序列：首先，内层的目标会成功，将X实例化为mary。然后 Prolog 会试图来满足下面这个目标：

```
not(member(X, [mary, fred, barb])).
```

因为member成功，所以这个目标会失败。而当这个目标失败时，将会取消X的实例化，因为，Prolog总是取消所有失败目标中的所有变量的实例化。然后，Prolog会试图满足外层的not目标，这会成功，因为它的参数已经失败。最后，将打印结果，即X。但是X现在并没有被实例化，因此系统将报告这个问题。没有被实例化的变量通常被打印为一串以下划线开头的数字。因此，Prolog的not不等于逻辑NOT，这件事实至少是误导的。

为什么逻辑NOT不可能成为Prolog语言中的一部分？其中的基本缘由来自于霍恩子句的形式：

$$A \text{ :- } B_1 \wedge B_2 \wedge \ldots \wedge B_n$$

712

如果所有的B命题都为真，我们可以得出结论A为真。但是不论任何B或是所有的B，是真或是假，我们都不可能得出结论A为假。从正逻辑出发，我们只能得出正逻辑的结论。因此霍恩子句形式，避免了任何否定的结论。

16.7.4 内在的限制

如在16.4节所描述的，逻辑程序设计的一个基本的目标，是提供非过程的程序设计，也就是使用这种设计程序人员，只需要说明一个程序应该做什么，而不需要说明怎样去做的系统。可以将前面排序的例子重写在这里：

```
sort(old_list, new_list) ⊂ permute(old_list, new_list) ∩ sorted(new_list)
sorted(list) ⊂ ∀j such that 1 ≤ j < n, list(j) ≤ list(j+1)
```

这很容易使用 Prolog来编写。例如，能够将被排序的子目标表示为：

```
sorted ([]).
sorted ([x]).
sorted ([x, y | list]) :- x <= y, sorted ([y | list]).
```

上面的排序程序的问题，是程序并不知道如何排序，它只知道枚举所给定链表的所有排列，直到它碰巧创建了排序的链表，实际上这是一个极慢的过程。

到目前为止还没有人发现一个过程，能够将已经排序的链表的描述转换成为某一种高效率的排序算法。归结能够做出许多有趣的事情，但是却肯定不能够完成这项工作。因此，一个排序链表的Prolog程序，必须详细地说明如何排序，如Inl在命令式语言或函数式语言中的那样。

所有这些问题，都意味着逻辑程序设计应该被抛弃？绝对不是！以逻辑程序设计目前的形式，它就能够处理许多有用的应用问题。此外，它是以一种吸引人的概念作为基础，何况它

的本身就是有趣的。最后一点，人们有可能开发出某种新的推理技术，以允许逻辑程序设计语言系统，高效地处理日益增多的应用问题。

16.8 逻辑程序设计的应用

在这一节中，我们简短地描述逻辑程序设计，特别是Prolog的几种当前的和潜在的重要
713 应用。

16.8.1 关系数据库管理系统

关系数据库管理系统（RDBMS），以表格的形式来存储数据。在这种数据库上的查询，通常是使用SQL(Structured Query Language)描述；与逻辑程序设计一样，SQL也是非过程的，用户不需要描述应该如何取得答案，相反，用户只是描述答案的特征。逻辑程序设计和关系数据库管理系统之间的连接，应该是很明显的。信息的简单表格，就能够使用Prolog结构来描述。而表之间的关系，就能够方便而且容易地使用Prolog的规则描述。检索过程是归结操作中所固有的。Prolog的目标语句，为关系数据库管理系统提供查询。逻辑程序设计十分自然地符合实现关系数据库管理系统的需要。

使用逻辑程序设计实现关系数据库管理系统的优点之一，是只需一种语言。在典型的关系数据库管理系统中，一种数据库语言包括不同语句，来进行数据定义，数据操作和查询；所有这些语句，都被嵌入一种通用程序设计语言中，如COBOL。将这种通用的语言用于数据处理，以及输入输出的功能。所有的这些功能，都可以在一种逻辑程序设计语言中进行。

使用逻辑程序设计实现关系数据库管理系统的另外一个优点，是内建的推理能力。传统的RDBMS不能够从数据库中推断出任何结论，而只能提供存储于数据库中的数据。也即它们只包含事实，而不包含事实规则及推理规则。与传统的关系数据库管理系统比较，使用逻辑程序设计实现关系数据库管理系统的主要缺点，是逻辑程序设计的实现的速度比较慢。使用逻辑推理，比使用平常的命令式程序设计技术的查询方法，需要更长的时间。

16.8.2 专家系统

专家系统，是一种被设计来在一些特定领域中模仿人类专家的计算机系统。这类系统，由一个事实数据库，一个推理过程，一些有关这个领域的启发式过程，以及某种友好的用户接口（它们使得系统像一个内行的人类顾问）所组成。系统的起始知识库由人类专家所提供；专家系统能够在使用过程中学习，从而它们的数据库还必定能够动态地增长。另外，当专家系统决定需要什么样的信息时，它还应该具有包括询问用户之类的获取更多信息的能力。

对于专家系统的设计人员而言，中心问题之一是在处理数据库时不可避免的不一致性和不完备性。逻辑程序设计似乎很适合于处理这些问题，例如，默认推理规则能够容易处理不
714 完备性的问题。

Prolog能够，并且已经被用来构造专家系统，它能够比较容易地实现专家系统的基本要求：使用归结来作为查询处理的基础，使用它的增加事实和规则的能力来提供学习能力，以及使用它的追踪设施来告诉用户一个给定的结果背后的推理。Prolog所缺少的，是在需要时自动询问用户以获得额外信息的能力。

逻辑程序设计在专家系统中一种最著名的应用，是称为APES的专家系统的构造系统。在Sergot（1983）和Hammond（1983）文献中对它进行了描述。APES系统包括一个非常灵活的设施，用于在专家系统构造期间收集来自用户的信息。它还包括了第二个解释器，用来产生

对它的查询的答案所进行的解释。

APECS已经被成功地使用，产生了好几个专家系统，其中的一个，是政府社会福利计划规则的专家系统，另外的一个，是英国国籍法（即英国的国籍规则）的专家系统。

16.8.3 自然语言处理

能够将逻辑程序设计用于某些类型的自然语言处理。特别是，某些计算机软件系统的自然语言的接口，可以方便地使用逻辑程序设计来建造。这些系统包括智能数据库，以及其他的基于知识的智能系统。在描述语言语法的方面，人们发现逻辑程序设计的形式，与上下文无关文法等价。另外，人们发现逻辑程序设计系统中的证明过程，与某些语法分析的策略等价。事实上，可以将后向链接归结，直接用于能够用上下文无关文法描述的句子，来进行语法分析。人们还发现用逻辑程序设计对语言建模时，自然语言的某些种类的语义，能够变得更加清晰。尤其是，在基于逻辑的语义网络的研究中，显示出自然语言中的多种句子，能够使用子句形式来表达（Deliyanni and Kowalski，1979）。在Kowalski（1979）文献中也讨论了基于逻辑的语义网络。

小结

符号逻辑，为逻辑程序设计和逻辑程序设计语言提供了基础。逻辑程序设计的方法，使用一个数据库和一种自动推理的过程，这个数据库包含一组事实，以及一些表述事实之间关系的规则。假定数据库里的事实都视为真，这种自动推理过程将会保持命题正确性。逻辑程序设计的方法，是为自动的定理证明而开发的。

Prolog是最广泛使用的逻辑程序设计语言。逻辑程序设计的起源，是Robinson为逻辑推理而开发的归结规则。Prolog主要是在Marseille大学，由Colmerauer和Roussel所开发的。其间，爱丁堡大学的Kowalski给予了一些帮助。

逻辑程序应该是非过程的，这意味着程序只需要给出答案的特征，而不需要给出取得答案全部过程。

Prolog的语句，是事实、规则或目标。大部分语句是由原子命题的结构和逻辑算子所组成的，虽然也允许了算术表达式。

归结，是Prolog解释器的主要活动，归结过程，主要是在命题中进行模式匹配。归结过程中广泛地使用回溯。当涉及变量时，能够将变量实例化为值，以提供匹配。这种实例化过程，被称为合一。

逻辑程序设计上的实现，存在着许多的问题。为了提高效率，甚至是为了避免无限循环，程序人员有时必须在程序中描述控制流程的信息。如之，还存在着封闭世界假设和否定问题。

逻辑程序设计已经被应用于许多不同的领域，主要是在关系数据库系统、专家系统和自然语言处理的方面。

文献注释

有些书籍述了Prolog语言，Clocksin和Mellish（2003）描述了标准形式的Prolog语言，Clark and McCabe（1984）描述了Prolog在微型计算机上的实现。

Hogger（1991）写了一本关于逻辑程序设计通用范围的优秀书籍，它也是一本关于逻辑程序设计应用的小而美的材料来源。

复习题

1. 在形式逻辑中符号逻辑的二种主要的用途是什么？
2. 一个复合项中的两个部分是什么？

3. 子句形式的命题的一般形式是什么？
4. 给出归结和合一的一般（非严格的）定义。
5. 什么是霍恩子句形式？
6. 说明语义的基本概念是什么？
7. Prolog中的规则三种形式是什么？
716 8. 在Prolog中，事实和规则语句的语法形式及用法是什么？
9. 解释在一个数据库中，将目标与事实相匹配的写和方法。
10. 解释在评论满足多目标时，深度优先与宽度优先之间的差别。
11. 解释在Prolog中回溯是怎样工作的。
12. 解释Prolog语句，X is X+1中的错误是什么。
13. Prolog程序人员在匹配期间能够控制模式匹配次序的两种方法是什么？
14. 解释Prolog中的产生与测试的程序设计策略。
15. 解释Prolog所使用的封闭世界假设。为什么它是一种限制？
16. 解释Prolog中的否定问题。为什么它是一种限制？
17. 解释自动定理证明和Prolog的推理过程之间的联系。
18. 解释过程的语言与非过程的语言之间的差别。
19. 解释Prolog的系统为什么必须执行回溯。
20. 在Prolog中的归结和合一之间的关系是什么？

练习题

1. 比较Ada和Prolog中数据类型化的概念。
2. 描述如何使用一个多处理器的计算机来实现归结，且制定支版本的Prolog能够使用这种方法吗？
3. 使用Prolog来描述你的家谱（仅基于事实）。上溯到你的祖父母，并且包括他们所有的后代。确保包括了所有的亲戚。
4. 编写一组家庭关系规则，包括自祖父母往下两代之间的所有关系。将这些规则加入练习题3中的事实，并且尽可能多地删除掉所需删除的事实。
5. 使用霍恩子句形式的Prolog来编写下面的条件语句：
   a. 如果Fred是Mike的父亲，则Fred是Mike的祖先。
   b. 如果Mike是Joe和Mary的父亲，则Mary是Joe的姐妹（或妹妹）。
   c. 如果Mike是Fred的兄弟，Fred是Mary的父亲，则Mike是Mary的叔叔。
717 6. 解释在两个方面，Scheme和Prolog的表处理功能是相似的。
7. 在什么方面，Scheme和Prolog的表处理功能是不相同的？
8. 写出Prolog与ML语言的比较，其中包括它们之间的两种类似性以及两种差别。
9. 从一本关于Prolog的书中学习，并写出对于一种“出现-检测”问题的描述。为什么Prolog允许这种问题存在于它的实现之中？

程序设计练习题

1. 编写一个Prolog程序，这个程序找出一个数字链表的最大数值。
2. 编写一个Prolog程序，如果两个给定链表参数的交为空时，程序成功。
3. 编写一个Prolog程序，这个程序返回一个包含了两个给定链表中元素的并的链表。
718 4. 编写一个Prolog程序，这个程序返回一个给定链表的最后一个元素。

参考文献

ACM. (1979) "Part A: Preliminary Ada Reference Manual" and "Part B: Rationale for the Design of the Ada Programming Language." SIGPLAN Notices, Vol. 14, No. 6.

ACM. (1993a) History of Programming Language Conference Proceedings. ACM SIGPLAN Notices, Vol. 28, No. 3, March.

ACM. (1993b) "High Performance FORTRAN Language Specification Part 1." FORTRAN Forum, Vol. 12, No. 4.

Aho, A. V., R. Sethi, and J. D. Ullman. (1986) Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA.

Aho, A. V., B. W. Kernighan, and P. J. Weinberger. (1988) The AWK Programming Language. Addison Wesley, Reading, MA.

Andrews, G. R., and F. B. Schneider. (1983) "Concepts and Notations for Concurrent Programming." ACM Computing Surveys, Vol. 15, No. 1, pp. 3–43.

ANSI. (1966) American National Standard Programming Language FORTRAN. American National Standards Institute, New York.

ANSI. (1976) American National Standard Programming Language PL/I. ANSI X3.53-1976. American National Standards Institute, New York.

ANSI. (1978a) American National Standard Programming Language FORTRAN. ANSI X3.9-1978. American National Standards Institute, New York.

ANSI. (1978b) American National Standard Programming Language Minimal BASIC. ANSI X3.60-1978. American National Standards Institute, New York.

ANSI. (1985) American National Standard Programming Language COBOL. ANSI X3.23-1985. American National Standards Institute, New York.

ANSI. (1989) American National Standard Programming Language C. ANSI X3.159-1989. American National Standards Institute, New York.

ANSI. (1992) American National Standard Programming Language FORTRAN 90. ANSI X3.198-1992. American National Standards Institute, New York.

Arden, B. W., B. A. Galler, and R. M. Graham. (1961) "MAD at Michigan." Datamation, Vol. 7, No. 12, pp. 27–28.

ARM. (1995) Ada Reference Manual. ISO/IEC/ANSI 8652:19. Intermetrics, Cambridge, MA.

Arnold, K., J. Gosling, and D. Holmes (2006) The Java (TM) Programming Language, 4e. Addison Wesley, Reading, MA.

Backus, J. (1954) "The IBM 701 Speedcoding System." J. ACM, Vol. 1, pp. 4–6.

Backus, J. (1959) "The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference." Proceedings International Conference on Information Processing. UNESCO, Paris, pp. 125–132.

Backus, J. (1978) "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." Commun. ACM, Vol. 21, No. 8, pp. 613–641.

Backus, J., F. L. Bauer, J. Green, C. Katz, J. McCarthy, P. Naur, A. J. Perlis, H. Rutishauser, K. Samelson, B. Vauquois, J. H. Wegstein, A. van Wijngaarden, and M. Woodger. (1963) "Revised Report on the Algorithmic Language ALGOL 60." Commun. ACM, Vol. 6, No. 1, pp. 1–17.

Balena, F. (2003) Programming Microsoft Visual Basic .NET Version 2003. Microsoft Press, Redmond, WA.

Ben-Ari, M. (1982) Principles of Concurrent Programming. Prentice-Hall, Englewood Cliffs, NJ.

Birtwistle, G. M., O. J. Dahl, B. Myhrhaug, and K. Nygaard. (1973) Simula BEGIN. Van Nostrand Reinhold, New York.

Bobrow, D. G., L. DeMichiel, R. Gabriel, S. Keene, G. Kiczales, and D. Moon. (1988) "Common Lisp Object System Specification X3J13 Document 88-002R." ACM SIGPLAN Notices, Vol. 17, No. 6, pp. 216–229.

Bodwin, J. M., L. Bradley, K. Kanda, D. Litle, and U. F. Pleban. (1982) "Experience with an Experimental Compiler Generator Based on Denotational Semantics." ACM SIGPLAN Notices, Vol. 17, No. 6, pp. 216–229.

Böhm, C., and G. Jacopini. (1966) "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules." Commun. ACM, Vol. 9, No. 5, pp. 366–371.

Bolsky, M., and D. Korn. (1995) The New KornShell Command and Programming Language. Prentice-Hall, Englewood Cliffs, NJ.

Booch, G. (1987) Software Engineering with Ada, 2e. Benjamin/Cummings, Redwood City, CA.

Bradley, J. C. (1989) QuickBASIC and QBASIC Using Modular Structures. W. C. Brown, Dubuque, IA.

Brinch Hansen, P. (1973) Operating System Principles. Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. (1975) "The Programming Language Concurrent Pascal." IEEE Transactions on Software Engineering, Vol. 1, No. 2, pp. 199–207.

Brinch Hansen, P. (1977) The Architecture of Concurrent Programs. Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. (1978) "Distributed Processes: A Concurrent Programming Concept." Commun. ACM, Vol. 21, No. 11, pp. 934–941.

Brown, J. A., S. Pakin, and R. P. Polivka. (1988) APL2 at a Glance. Prentice-Hall, Englewood Cliffs, NJ.

Campione, M., K. Walrath, and A. Huml. (2001) The Java Tutorial, 3e. Addison-Wesley, Reading, MA.

Cardelli, L., J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. (1989) Modula-3 Report (revised). Digital System Research Center, Palo Alto, CA.

Chambers, C., and D. Ungar. (1991) "Making Pure Object-Oriented Languages Practical." SIGPLAN Notices, Vol. 26, No. 7, pp. 1–15.

Chomsky, N. (1956) "Three Models for the Description of Language." IRE Transactions on Information Theory, Vol. 2, No. 3, pp. 113–124.

Chomsky, N. (1959) "On Certain Formal Properties of Grammars." Information and Control, Vol. 2, No. 2, pp. 137–167.

Church, A. (1941) Annals of Mathematics Studies, Volume 6: Calculi of Lambda Conversion. Princeton Univ. Press, Princeton, NJ. Reprinted by Klaus Reprint Corporation, New York, 1966.

Clark, K. L., and F. G. McCabe. (1984) Micro-PROLOG: Programming in Logic. Prentice-Hall, Englewood Cliffs, NJ.

Clarke, T. J. A., J. C. Wileden, and A. L. Wolf. (1980) "Nesting in Ada Is for the Birds." ACM SIGPLAN Notices, Vol. 15, No. 11, pp. 139–145.

Cleaveland, J. C. (1986) An Introduction to Data Types. Addison-Wesley, Reading, MA.

Cleaveland, J. C., and R. C. Uzgalis. (1976) Grammars for Programming Languages: What Every Programmer Should Know About Grammar. American Elsevier, New York.

Clocksin, W. F., and C. S. Mellish. (2003) Programming in Prolog, 5e. Springer-Verlag, New York.

Cohen, J. (1981) "Garbage Collection of Linked Data Structures." ACM Computing Surveys, Vol. 13, No. 3, pp. 341–368.

Converse, T., and J. Park. (2001) PHP 4 Bible. IDG Books, New York.

Conway, M. E. (1963). "Design of a Separable Transition-Diagram Compiler." Commun. ACM, Vol. 6, No. 7, pp. 396–408.

Conway, R., and R. Constable. (1976) "PL/CS—A Disciplined Subset of PL/I." Technical Report TR76293, Department of Computer Science, Cornell University, Ithaca, NY.

Cornell University. (1977) PL/C User's Guide, Release 7.0. Department of Computer Science, Cornell University, Ithaca, NY.

Correa, N. (1992) "Empty Categories, Chain Binding, and Parsing." pp. 83–121, Principle-Based Parsing. Eds. R. C. Berwick, S. P. Abney, and C. Tenny. Kluwer Academic Publishers, Boston.

Dahl, O.-J., E. W. Dijkstra, and C. A. R. Hoare. (1972) Structured Programming. Academic Press, New York.

Dahl, O.-J., and K. Nygaard. (1967) "SIMULA 67 Common Base Proposal." Norwegian Computing Center Document, Oslo.

Deitel, H. M., P. J. Deitel, and T. R. Nieto. (2002) Visual BASIC .Net: How to Program, 2e. Prentice-Hall, Inc., Upper Saddle River, NJ.

Deliyanni, A., and R. A. Kowalski. (1979) "Logic and Semantic Networks." Commun. ACM, Vol. 22, No. 3, pp 184–192.

Department of Defense. (1960) "COBOL, Initial Specifications for a Common Business Oriented Language." U.S. Department of Defense, Washington, D.C.

Department of Defense. (1961) "COBOL—1961, Revised Specifications for a Common Business Oriented Language." U.S. Department of Defense, Washington, D.C.

Department of Defense. (1962) "COBOL—1961 EXTENDED, Extended Specifications for a Common Business Oriented Language." U.S. Department of Defense, Washington, D.C.

Department of Defense. (1975a) "Requirements for High Order Programming Languages, STRAWMAN." July. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1975b) "Requirements for High Order Programming Languages, WOODENMAN." August. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1976) "Requirements for High Order Programming Languages, TINMAN." June. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1977) "Requirements for High Order Programming Languages, IRONMAN." January. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1978) "Requirements for High Order Programming Languages, STEELMAN." June. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1980a) "Requirements for High Order Programming Languages, STONEMAN." February. U.S. Department of Defense, Washington, D.C.

Department of Defense. (1980b) "Requirements for the Programming Environment for the Common High Order Language, STONEMAN." U.S. Department of Defense, Washington, D.C.

DeRemer, F. (1971) "Simple LR(k) Grammars." Commun. ACM, Vol. 14, No. 7, pp. 453–460.

DeRemer, F. and T. Pennello. (1982) "Efficient Computation of LALR(1) Look-Ahead Sets." ACM TOPLAS, Vol. 4, No. 4, pp. 615–649.

Deutsch, L. P., and D. G. Bobrow. (1976) "An Efficient Incremental Automatic Garbage Collector." Commun. ACM, Vol. 11, No. 3, pp. 522–526.

Dijkstra, E. W. (1968a) "Goto Statement Considered Harmful." Commun. ACM, Vol. 11, No. 3, pp. 147–149.

Dijkstra, E. W. (1968b) "Cooperating Sequential Processes." In Programming Languages, F. Genuys (ed.). Academic Press, New York, pp. 43–112.

Dijkstra, E. W. (1972) "The Humble Programmer." Commun. ACM, Vol. 15, No. 10, pp. 859–866.

Dijkstra, E. W. (1975) "Guarded Commands, Nondeterminacy, and Formal Derivation of Programs." Commun. ACM, Vol. 18, No. 8, pp. 453–457.

Dijkstra, E. W. (1976). A Discipline of Programming. Prentice-Hall, Englewood Cliffs, N.J.

Dybvig, R. K. (2003) The Scheme Programming Language, 3e. MIT Press, Boston.

Ellis, M. A., and B. Stroustrup (1990) The Annotated C++ Reference Manual. Addison Wesley, Reading, MA.

Farber, D. J., R. E. Griswold, and I. P. Polonsky. (1964) "SNOBOL, a String Manipulation Language." J. ACM, Vol. 11, No. 1, pp. 21–30.

Farrow, R. (1982) "LINGUIST 86: Yet Another Translator Writing System Based on Attribute Grammars." ACM SIGPLAN Notices, Vol. 17, No. 6, pp. 160–171.

Fischer, C. N., G. F. Johnson, J. Mauney, A. Pal, and D. L. Stock. (1984) "The Poe Language-Based Editor Project." ACM SIGPLAN Notices, Vol. 19, No. 5, pp. 21–29.

Fischer, C. N., and R. J. LeBlanc. (1977) "UW-Pascal Reference Manual." Madison Academic Computing Center, Madison, WI.

Fischer, C.N., and R. J. LeBlanc. (1980) "Implementation of Runtime Diagnostics in Pascal." IEEE Transactions on Software Engineering, SE-6, No. 4, pp. 313–319.

Fischer, C. N., and R. J. LeBlanc. (1991) Crafting a Compiler in C. Benjamin/Cummings, Menlo Park, CA.

Flanagan, D. (2002) JavaScript: The Definitive Guide, 4e. O'Reilly Media, Sebastopol, CA.

Floyd, R. W. (1967) "Assigning Meanings to Programs." Proceedings Symposium Applied Mathematics. Mathematical Aspects of Computer Science Ed. J. T. Schwartz. American Mathematical Society, Providence, RI.

Frege, G. (1892) "Über Sinn und Bedeutung." Zeitschrift für Philosophie und Philosophische Kritik, Vol. 100, pp. 25–50.

Friedl, J. E. F. (2006) Mastering Regular Expressions, 3e. O'Reilly Media, Sebastopol, CA.

Friedman, D. P., and D. S. Wise. (1979) "Reference Counting's Ability to Collect Cycles Is Not Insurmountable." Information Processing Letters, Vol. 8, No. 1, pp. 41–45.

Fuchi, K. (1981) "Aiming for Knowledge Information Processing Systems." Proceedings of the International Conference on Fifth Generation Computing Systems. Japan Information Processing Development Center, Tokyo. Republished (1982) by North Holland Publishing, Amsterdam.

Gehani, N. (1983) Ada: An Advanced Introduction. Prentice-Hall, Englewood Cliffs, N.J.

Gilman, L., and A.J. Rose. (1976) APL: An Interactive Approach, 2e. J. Wiley, New York.

Goldberg, A., and D. Robson. (1983) Smalltalk-80: The Language and Its Implementation. Addison Wesley, Reading, MA.

Goldberg, A., and D. Robson. (1989) Smalltalk-80: The Language. Addison-Wesley, Reading, MA.

Goodenough, J. B. (1975) "Exception Handling: Issues and Proposed Notation." Commun. ACM, Vol. 18, No. 12, pp. 683–696.

Goos, G., and J. Hartmanis (eds.) (1983) The Programming Language Ada Reference Manual. American National Standards Institute. ANSI/MIL-STD-1815A–1983. Lecture Notes in Computer Science 155. Springer-Verlag, New York.

Gordon, M. (1979) The Denotational Description of Programming Languages. An Introduction. Springer-Verlag, Berlin–New York.

Graham, P. (1996) ANSI Common LISP. Prentice-Hall, Englewood Cliffs, N.J.

Gries, D. (1981) The Science of Programming. Springer-Verlag, New York.

Griswold, R. E., and M. T. Griswold. (1983) The ICON Programming Language. Prentice-Hall, Englewood Cliffs, N.J.

Griswold, R. E., F. Poage, and I. P. Polonsky. (1971) The SNOBOL 4 Programming Language, 2e. Prentice-Hall, Englewood Cliffs, N.J.

Hammond, P. (1983) APES: A User Manual. Department of Computing Report 83/9. Imperial College of Science and Technology, London.

Henderson, P. (1980) Functional Programming: Application and Implementation. Prentice-Hall, Englewood Cliffs, N.J.

Hoare, C. A. R. (1969) "An Axiomatic Basis of Computer Programming." Commun. ACM, Vol. 12, No. 10, pp. 576–580.

Hoare, C. A. R. (1972) "Proof of Correctness of Data Representations." Acta Informatica, Vol. 1, pp. 271–281.

Hoare, C. A. R. (1973) "Hints on Programming Language Design." Proceedings ACM SIGACT/SIGPLAN Conference on Principles of Programming Languages. Also published as Technical Report STAN-CS-73-403, Stanford University Computer Science Department.

Hoare, C. A. R. (1974) "Monitors: An Operating System Structuring Concept." Commun. ACM, Vol. 17, No. 10, pp. 549–557.

Hoare, C. A. R. (1978) "Communicating Sequential Processes." Commun. ACM, Vol. 21, No. 8, pp. 666–677.

Hoare, C. A. R. (1981) "The Emperor's Old Clothes." Commun. ACM, Vol. 24, No. 2, pp. 75–83.

Hoare, C. A. R., and N. Wirth. (1973) "An Axiomatic Definition of the Programming Language Pascal." Acta Informatica, Vol. 2, pp. 335–355.

Hogger, C. J. (1984) Introduction to Logic Programming. Academic Press, London.

Hogger, C. J. (1991) Essentials of Logic Programming. Oxford Science Publications, Oxford, England.

Holt, R. C., G. S. Graham, E. D. Lazowska, and M. A. Scott. (1978) Structured Concurrent Programming with Operating Systems Applications. Addison-Wesley, Reading, MA.

Horn, A. (1951) "On Sentences Which Are True of Direct Unions of Algebras." J. Symbolic Logic, Vol. 16, pp. 14–21.

Hudak, P., and J. Fasel. (1992) "A Gentle Introduction to Haskell, ACM SIGPLAN Notices, 27(5), May 1992, pp. T1–T53.

Hughs, (1989) "Why Functional Programming Matters", The Computer Journal, Vol. 32, No. 2 pp. 98–107.

Huskey, H. C., R. Love, and N. Wirth. (1963) "A Syntactic Description of SC NELIAC." Commun. ACM, Vol. 6, No. 7, pp. 367–375.

IBM. (1954) "Preliminary Report, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN." IBM Corporation, New York.

IBM. (1956) "Programmer's Reference Manual, The FORTRAN Automatic Coding System for the IBM 704 EDPM." IBM Corporation, New York.

IBM. (1964) "The New Programming Language." IBM UK Laboratories.

Ichbiah, J. D., J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-Brueckner, and B. A. Wichmann. (1979) "Rationale for the Design of the Ada Programming Language." ACM SIGPLAN Notices, Vol. 14, No. 6, Part B.

IEEE. (1985) "Binary Floating Point Arithmetic." IEEE Standard 754, IEEE, New York.

INCITS/ISO/IEC (1997) 1539-1-1997 Information Technology—Programming Languages—FORTRAN Part 1: Base Language. American National Standards Institute, New York.

Ingerman, P. Z. (1967). "Panini-Backus Form Suggested." Commun. ACM, Vol. 10, No. 3, p. 137.

Intermetrics. (1993) Programming Language Ada, Draft, Version 4.0. Cambridge, MA.

ISO. (1982) Specification for Programming Language Pascal. ISO7185–1982. International Organization for Standardization, Geneva, Switzerland.

ISO/IEC (1996) 14977:1996, Information Technology—Syntactic Metalanguage—Extended BNF. International Organization for Standardization, Geneva, Switzerland.

ISO. (1998) ISO14882-1, ISO/IEC Standard—Information Technology—Programming Language—C++. International Organization for Standardization, Geneva, Switzerland.

ISO. (1999) ISO/IEC 9899:1999, Programming Language C. American National Standards Institute, New York.

ISO/IEC (2002) 1989:2002 Information Technology—Programming Languages—COBOL. American National Standards Institute, New York.

Iverson, K. E. (1962) A Programming Language. John Wiley, New York.

Jensen, K., and N. Wirth. (1974) Pascal User Manual and Report. Springer-Verlag, Berlin.

Johnson, S. C. (1975) "Yacc—Yet Another Compiler Compiler." Computing Science Report 32. AT&T Bell Laboratories, Murray Hill, NJ.

Jones, N. D. (ed.) (1980) Semantic-Directed Compiler Generation. Lecture Notes in Computer Science, Vol. 94. Springer-Verlag, Heidelberg, FRG.

Kay, A. (1969) The Reactive Engine. Ph.D. Thesis, University of Utah, Salt Lake.

Kernighan, B. W., and D. M. Ritchie. (1978) The C Programming Language. Prentice-Hall, Englewood Cliffs, NJ.

Knuth, D. E. (1965) "On the Translation of Languages from Left to Right." Information & Control, Vol. 8, No. 6, pp. 607–639.

Knuth, D. E. (1967) "The Remaining Trouble Spots in ALGOL 60." Commun. ACM, Vol. 10, No. 10, pp. 611–618.

Knuth, D. E. (1968a) "Semantics of Context-Free Languages." Mathematical Systems Theory, Vol. 2, No. 2, pp. 127–146.

Knuth, D. E. (1968b) The Art of Computer Programming, Vol. 1, 2e. Addison-Wesley, Reading, MA.

Knuth, D. E. (1974) "Structured Programming with GOTO Statements." ACM Computing Surveys, Vol. 6, No. 4, pp. 261–301.

Knuth, D. E. (1981) The Art of Computer Programming, Vol. II, 2e. Addison-Wesley, Reading, MA.

Knuth, D. E., and L. T. Pardo. (1977) "Early Development of Programming Languages." In Encyclopedia of Computer Science and Technology, Vol. 7. Dekker, New York, pp. 419–493.

Kowalski, R. A. (1979) Logic for Problem Solving. Artificial Intelligence Series, Vol. 7. Elsevier North-Holland, New York.

Laning, J. H., Jr., and N. Zierler. (1954) "A Program for Translation of Mathematical Equations for Whirlwind I." Engineering memorandum E-364, Instrumentation Laboratory, Massachusetts Institute of Technology, Cambridge, MA.

Ledgard, H. (1984) The American Pascal Standard. Springer-Verlag, New York.

Ledgard, H. F., and M. Marcotty. (1975) "A Genealogy of Control Structures." Commun. ACM, Vol. 18, No. 11, pp. 629–639.

Litchner, R. (2000) Delphi in a Nutshell. O'Reilly Media, Sebastopol, CA.

Liskov, B., R. L. Atkinson, T. Bloom, J. E. B. Moss, C. Schaffert, R. Scheifler, and A. Snyder (1981) "CLU Reference Manual." Springer, New York.

Liskov, B., and A. Snyder (1979) "Exception Handling in CLU." IEEE Transactions on Software Engineering, Vol. SE-5, No. 6, pp. 546–558.

Lomet, D. (1975) "Scheme for Invalidating References to Freed Storage." IBM J. of Research and Development, Vol. 19, pp. 26–35.

Lutz, M., and D. Ascher. (2004) Learning Python, 2e. O'Reilly Media, Sebastopol, CA.

MacLaren, M. D. (1977) "Exception Handling in PL/I." ACM SIGPLAN Notices, Vol. 12, No. 3, pp. 101–104.

Marcotty, M., H. F. Ledgard, and G. V. Bochmann. (1976) "A Sampler of Formal Definitions." ACM Computing Surveys, Vol. 8, No. 2, pp. 191–276.

Mather, D. C., and S. V. Waite (eds.) (1971) BASIC, 6e. University Press of New England, Hanover, NH.

McCarthy, J. (1960) "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I." Commun. ACM, Vol. 3, No. 4, pp. 184–195.

McCarthy, J., P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. Levin. (1965) LISP 1.5 Programmer's Manual, 2e. MIT Press, Cambridge, MA.

McCracken, D. (1970) "Whither APL?" Datamation, Sept. 15, pp. 53–57.

Metcalf, M., J. Reid, and M. Cohen. (2004) Fortran 95/2003 Explained, 3e. Oxford University Press, Oxford, England.

Meyer, B. (1990) Introduction to the Theory of Programming Languages. Prentice-Hall, Englewood Cliffs, NJ.

Meyer, B. (1992) Eiffel: The Language. Prentice-Hall, Englewood Cliffs, NJ.

Microsoft. (1991) Microsoft Visual Basic Language Reference. Document CB20664-3491, Redmond, WA.

Milner, R., M. Tofte, and R. Harper. (1990) The Definition of Standard ML. MIT Press, Cambridge, MA.

Milos, D., U. Pleban, and G. Loegel. (1984) "Direct Implementation of Compiler Specifications." ACM Principles of Programming Languages 1984, pp. 196–202.

Mitchell, J. G., W. Maybury, and R. Sweet. (1979) Mesa Language Manual, Version 5.0, CSL-79-3, Xerox Research Center, Palo Alto, CA.

Moss, C. (1994) Prolog++: The Power of Object-Oriented and Logic Programming. Addison-Wesley, Reading, MA.

Motooka, T. (1981) "Challenge for Knowledge Information Processing Systems." Proceedings of the International Conference on Fifth Generation Computing Systems. Japan Information Processing Development Center, Tokyo. Republished (1982) by North-Holland Publishing, Amsterdam.

Naur, P. (ed.) (1960) "Report on the Algorithmic Language ALGOL 60." Commun. ACM, Vol. 3, No. 5, pp. 299–314.

Newell, A., and H. A. Simon. (1956) "The Logic Theory Machine—A Complex Information Processing System." IRE Transactions on Information Theory, Vol. IT-2, No. 3, pp. 61–79.

Newell, A., and F. M. Tonge. (1960) "An Introduction to Information Processing Language V." Commun. ACM, Vol. 3, No. 4, pp. 205–211.

Nilsson, N. J. (1971) Problem-Solving Methods in Artificial Intelligence. McGraw-Hill, New York.

Ousterhout, J. K. (1994) Tcl and the Tk Toolkit. Addison-Wesley, Reading, MA.

Pagan, F. G. (1981) Formal Specifications of Programming Languages. Prentice-Hall, Englewood Cliffs, NJ.

Papert, S. (1980) MindStorms: Children, Computers and Powerful Ideas. Basic Books, New York.

Perlis, A., and K. Samelson. (1958) "Preliminary Report—International Algebraic Language." Commun. ACM, Vol. 1, No. 12, pp. 8–22.

Peyton Jones, S. L. (1987) The Implementation of Functional Programming Languages. Prentice-Hall, Englewood Cliffs, NJ.

Pratt, T. W. (1984) Programming Languages: Design and Implementation, 2e. Prentice Hall, Englewood Cliffs, NJ.

Pratt, T. W., and M. V. Zelkowitz (2001) Programming Languages: Design and Implementation, 4e. Prentice Hall, Englewood Cliffs, NJ.

Raymond, E. (2004) Art of UNIX Programming. Addison Wesley, Boston.

Rees, J., and W. Clinger. (1986) "Revised Report on the Algorithmic Language Scheme," ACM SIGPLAN Notices, Vol. 21, No. 12, pp. 37–79.

Remington-Rand. (1952) "UNIVAC Short Code." Unpublished collection of dittoed notes. Preface by A. B. Tonik, dated October 25, 1955 (1 p.); Preface by J. R. Logan, undated but apparently from 1952 (1 p.); Preliminary exposition, 1952? (22 pp., where in which pp. 20–22 appear to be a later replacement); Short code supplementary information, topic one (7 pp.); Addenda #1, 2, 3, 4 (9 pp.).

Richards, M. (1969) "BCPL: A Tool for Compiler Writing and Systems Programming." Proc. AFIPS SJCC, Vol. 34, pp. 557–566.

Robinson, J. A. (1965) "A Machine-Oriented Logic Based on the Resolution Principle." Journal of the ACM, Vol. 12, pp. 23–41.

Romanowsky, A. and B. Sandin (2001) "Except for Exception Handling," Ada Letters, Vol. 21, No. 3, September 2001, pp. 19–25.

Roussel, P. (1975) "PROLOG: Manual de Reference et D'Utilisation." Research Report, Artificial Intelligence Group, Univ. of Aix-Marseille, Luming, France.

Rubin, F. (1987) "'GOTO Statement Considered Harmful' considered Harmful" (letter to editor). Commun. ACM, Vol. 30, No. 3, pp. 195–196.

Rutishauser, H. (1967) Description of ALGOL 60. Springer-Verlag, New York.

Sammet, J. E. (1969) Programming Languages: History and Fundamentals. Prentice-Hall, Englewood Cliffs, NJ.

Sammet, J. E. (1976) "Roster of Programming Languages for 1974–75," Commun. ACM, Vol. 19, No. 12, pp. 655–669.

Schneider, G. M. (1999) An Introduction to Programming Using Visual BASIC 6.0. Prentice-Hall, Englewood Cliffs, NJ.

Schorr, H., and W. Waite. (1967) "An Efficient Machine Independent Procedure for Garbage Collection in Various List Structures." Commun. ACM, Vol. 10, No. 8, pp. 501–506.

Scott, D. S., and C. Strachey. (1971) "Towards a Mathematical Semantics for Computer Language." In Proceedings, Symposium on Computers and Automation, J. Fox (ed.). Polytechnic Institute of Brooklyn Press, New York, pp. 19–46.

Scott, M. (2000) Programming Language Pragmatics, Morgan Kaufman, San Francisco, CA.

Sibsita, R. W. (1991) VAX Structured Assembly Language Programming, 2e. Benjamin/Cummings, Redwood City, CA.

Sergot, M. J. (1983) "A Query-the-User Facility for Logic Programming." In Integrated Interactive Computer Systems, P. Degano and E. Sandewall (eds.). North-Holland Publishing, Amsterdam.

Shaw, C. J. (1963) "A Specification of JOVIAL." Commun. ACM, Vol. 6, No. 12, pp. 721–736.

Sommerville, I. (2005) Software Engineering, 7e. Addison-Wesley, Reading, MA.

Steele, G. L., Jr. (1984) Common LISP. Digital Press, Burlington, MA.

Stoy, J. E. (1977) Denotational Semantics: The Scott–Strachey Approach to Programming Language Semantics. MIT Press, Cambridge, MA.

Stroustrup, B. (1983) "Adding Classes to C: An Exercise in Language Evolution." Software—Practice and Experience, Vol. 13, pp. 139–161.

Stroustrup, B. (1984) "Data Abstraction in C." AT&T Bell Laboratories Technical Journal, Vol. 63, No. 8.

Stroustrup, B. (1986) The C++ Programming Language. Addison-Wesley, Reading, MA.

Stroustrup, B. (1988) "What is Object-Oriented Programming?" IEEE Software, May 1988, pp. 10–20.

Stroustrup, B. (1991) The C++ Programming Language, 2e. Addison-Wesley, Reading, MA.

Stroustrup, B. (1994) The Design and Evolution of C++. Addison-Wesley, Reading, MA.

Stroustrup, B. (1997) The C++ Programming Language, 3e. Addison-Wesley, Reading, MA.

Sussman, G. J., and G. L. Steele, Jr. (1975) "Scheme: An Interpreter for Extended Lambda Calculus." MIT AI Memo No. 349 (December, 1975).

Suzuki, N. (1982) "Analysis of Pointer 'Rotation.'" Commun. ACM, Vol. 25, No. 5, pp. 330–335.

Tenenbaum, A. S. (2005) Structured Computer Organization, 5e. Prentice-Hall, Englewood Cliffs, NJ.

Tenenbaum, A. M., Y. Langsam, and M. J. Augenstein. (1990) Data Structures Using C. Prentice Hall, Englewood Cliffs, NJ.

Teitelbaum, T., and T. Reps. (1981) "The Cornell Program Synthesizer: A Syntax Directed Programming Environment." Commun. ACM, Vol. 24, No. 9, pp. 563–573.

Teitelman, W. (1975) INTERLISP Reference Manual. Xerox Palo Alto Research Center, Palo Alto, CA.

Thomas, D., C. Fowler, and A. Hunt. (2005) Ruby: The Pragmatic Programmers Guide, 2e. The Pragmatic Bookshelf, Raleigh, NC.

Thompson, S. (1999) Haskell: The Craft of Functional Programming, 2e. Addison-Wesley, Reading, MA.

Turner, D. (1986) "An Overview of Miranda." ACM SIGPLAN Notices, Vol. 21, No. 12, pp. 158–166.

Ullman, J. D. (1998) Elements of ML Programming. ML97 Edition. Prentice-Hall, Englewood Cliffs, NJ.

van Emden, M.H. (1980) "MicroPlanner for Prolog: A Rejoinder." SIGART Newsletter, No. 72, August, pp. 19–20.

van Wijngaarden, A., B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. (1969) "Report on the Algorithmic Language ALGOL 68." Numerische Mathematik, Vol. 14, No. 2, pp. 79–218.

Wadler, P. (1998) "Why No One Uses Functional Languages." ACM SIGPLAN Notices, Vol. 33, No. 2, February 1998, pp. 25–30.

Wall, L., J. Christiansen, and J. Orwant. (2000) Programming Perl, 3e. O'Reilly & Associates, Sebastopol, CA.

Warren, D. H. D., L. M. Pereira, and F. C. N. Pereira. (1979) "User's Guide to DEC System-10 Prolog." Occasional Paper 15. Department of Artificial Intelligence, Univ. of Edinburgh, Scotland.

Watt, D. A. (1979) "An Extended Attribute Grammar for Pascal." ACM SIGPLAN Notices, Vol. 14, No. 2, pp. 60–74.

Wegner, P. (1972) "The Vienna Definition Language." ACM Computing Surveys, Vol. 4, No. 1, pp. 5–63.

Weissman, C. (1967) LISP 1.5 Primer. Dickenson Press, Belmont, CA.

Wexelblat, R. L. (ed.) (1981) History of Programming Languages. Academic Press, New York.

Wheeler, D. J. (1950) "Programme Organization and Initial Orders for the EDSAC." Proc. R. Soc. London, Ser. A, Vol. 202, pp. 573–589.

Wilkes, M. V. (1952) "Pure and Applied Programming." In Proceedings of the ACM National Conference, Vol. 2. Toronto, pp. 121–124.

Wilkes, M. V., D. J. Wheeler, and S. Gill. (1951) The Preparation of Programs for an Electronic Digital Computer, with Special Reference to the EDSAC and the Use of a Library of Subroutines. Addison-Wesley, Reading, MA.

Wilkes, M. V., D. J. Wheeler, and S. Gill. (1957) The Preparation of Programs for an Electronic Digital Computer, 2e. Addison-Wesley, Reading, MA.

Wilson, P. R. (2005) "Uniprocessor Garbage Collection Techniques." Available at http://www.cs.utexas.edu/users/oops/papers.html#bigsurv.

Wirth, N. (1971) "The Programming Language Pascal." Acta Informatica, Vol. 1, No. 1, pp. 35–63.

Wirth, N. (1973) Systematic Programming: An Introduction. Prentice-Hall, Englewood Cliffs, NJ.

Wirth, N. (1975) "On the Design of Programming Languages." Information Processing 74 (Proceedings of IFIP Congress 74). North Holland, Amsterdam, pp. 386–393.

Wirth, N. (1977) "Modula: A Language for Modular Multi-Programming." Software—Practice and Experience, Vol. 7, pp. 3–35.

Wirth, N. (1985) Programming in Modula-2, 3e. Springer-Verlag, New York.

Wirth, N. (1988) "The Programming Language Oberon." Software—Practice and Experience, Vol. 18, No. 7, pp. 671–690.

Wirth, N., and C. A. R. Hoare. (1966) "A Contribution to the Development of ALGOL." Commun. ACM, Vol. 9, No. 6, pp. 413–431.

Wulf, W. A., D. B. Russell, and A. N. Habermann. (1971) "BLISS: A Language for Systems Programming." Commun. ACM, Vol. 14, No. 12, pp. 780–790.

Zuse, K. (1972) "Der Plankalkül." Manuscript prepared in 1945, published in Berichte der Gesellschaft für Mathematik und Datenverarbeitung, No. 63 (Bonn, 1972); Part 3, 265 pp. English translation of all but pp. 174–196 in No. 106 (Bonn, 1976), pp. 42–244.

索　引

索引中的页码为英文原书的页码，与书中边栏的页码一致。

Symbols（符号）
[] (brackets)（方括号，中括号），132
_(underscore)（下划线），203
{} (braces)（大括号），132

A

Abstract classes（抽象类），511，528
Abstract data types（抽象数据类型），471－473
  design issues（设计问题），474
  in Ada（在Ada中），475－481
  In C#（在C#中），486－488
  in C++（在C++中），482－485
  in Java（在Java中），485－486
  in Ruby（在Ruby中），488－490
Abstraction（抽象），16，22，490－491
  in Backus Naur form（巴科斯－诺尔范式中的），120
Abstract methods（抽象方法），510
accept clause（accept子句），572
Access type in Ada（在Ada中的访问类型），295
ACM (Association for Computing Machinery)（美国计算机协会），57，88
ACTION (LR parsing tables)（ACTION（LR语法分析表）），192
Activation record instance（活动记录实例），441
Activation records（活动记录），441，443－444
Active subprograms（活动子程序），236，446
Actor tasks（演员任务），574
Actual parameters（实参），388
Ada，87－91
  abstract data types（抽象数据类型），475－481
  access types（访问类型），295
  competition synchronization（竞争同步），577－579
  concurrency（support for）（支持并发），573－583
  cooperation synchronization（合作同步），576－577
  design（设计），88－89
  dynamic binding（动态绑定），537－538
  evaluation（评价），89－90
  exception handling（异常处理），608－615
  generic subprograms（通用子程序），422－426
  historical background of（历史背景），87－88
  information hiding（信息隐蔽），475－479
  limited private types（受限私有类型），477
  mod operator（mod操作符），313
  named constants（命名常量），238
  overview of（概述），89
  packages（包），475－481，501－502
  parameterized abstract data types（参数化抽象数据类型），491－493
  pointers（指针），295
  private types（私有类型），476
  rem operator（rem操作符），315
  for statements（for语句），360－361
  subrange type design（子范围类型设计），261－263
  subtypes（子类型），223
    union types（union类型），288－290
    use，481
    with，481
Ada 95，91
  child packages（子包），528－530
  dynamic binding（动态绑定），537－538
  evaluation of support for OOP（支持OOP的评价），539－542
  inheritance（继承），536－537
  object oriented programming (support for)（支持面向对象的程序设计），535－542
  protected objects（受保护对象），580－581
  string length options（字符长度选项），256
Addresses (variables)（变量地址），209－207
Ad hoc binding（特别绑定），419
Ad hoc polymorphism（特别多态），422
Advice Taker（建议采纳器），52
Aggregate values（聚集值），269
Ahc，A1，83
AI (artificial intelligence)（人工智能），6，51－52，625－676
AIMACO（AIMACO语言），63
ALGOL（ALGOL语言），58，58－59
  input（输出），59
ALGOL 60（ALGOL 60语言），4，51，59－60
  Backus-Naur form（巴科斯－诺尔范式），60－61
  design process（设计过程），59－60
  early design process of（早期设计过程），57－58

evaluation（评价），60－61
historical background of（历史背景），56－57
overview of（概述），60
report（报告），60
ALGOL 68 (ALGOL 68语言)，55－79
design（设计），77－78
evaluation（评价），78－79
overview of（概述），78
ALGOL Bulletin（ALGOL公报），59
Algorithms（计算算法）
left factoring（提取左因子），181
LR（LR文法），191－195
parse trees（语法分析树），123－124
parsing（语法分析），175－178，另见Parsing
shift-reduce（移进－归约），190
Aliases（变量的别名），206－207
Aliasing（别名化），18
Allocation（分配），215
of objects（对象的），515－516
Ambiguity (grammars)（文法的歧义性），124－125
American Standard Code for Information Interchange (ASCII)（美国信息标准代码），253
Analysis（分析）
lexical（词法），169－175
syntax（语法），175－195，另见syntax语法
Anonymous variables（匿名变量），291
and then Boolean operator（布尔操符），332
Antecedent（前件），687
APES system（APES系统），715
APL（APL语言），25，75－76
Applications（应用）
business（商务），5
functional programming languages（函数式程序设计语言），675－676
logic programming languages（逻辑程序设计语言），713
scientific（科学的），5
Apply-to-all functional forms（对所有元素应用形式），645，665
APT（自动化程序工具），25
Architecture（体系结构）
computer（计算机），20－22
multiprocessor（多处理器），557－558
von Neumann（冯·诺依曼），20
Arithmetic expressions（算术表达式），313－322
associativity（结合性），316－318
conditional expressions（条件表达式），319
grammars（文法），126－127
in Prolog（在Prolog语言中），699－702
operand evaluation order（操作数求值顺序），319－321
operator evaluation order（操作符求值顺序），315－318
parentheses（）（圆括号），318
in Prolog（在Prolog语言中），699－702
Array formal parameter (Ruby)（Ruby的数组形参），390
Arrays（数组）
associative (hashes)（关联数组），84，280－282
categories（种类），265－267
design issues（设计问题），263
evaluation of（评价），273
flex（flex数组），78
heterogeneous（异构的），267－268
implementation（实现），273－280
indexes（索引，下标），264－265
initialization（初始化），268－269
jagged（参差的），271
multidimensional arrays as parameters（多维数组作为参数），411－415
operations（操作），269－280
rectangular（矩形），271
slices（片），271－273
types（类型），263－280
Artificial intelligence (AI)（人工智能），6，51－52
ASCII (American Standard Code for Information Interchange)（美国标准信息交换码），253
Assemblies（组合），498
Assertions（断言），143－144，628－629
Assignment statements（赋值语句），332－337
compound assignment operators（复合赋值操作符），333－334
conditional targets（条件目标），333
denotational semantics（指称语义），159－160
as expressions（表达式为），335－336
mixed-mode assignment（混合形式赋值），337－338
Plankalkül（Plankalkül语言），43
simple（简单），333
unary assignment operators（一元赋值操作符），334－335
Association for Computing Machinery (ACM)（美国计算机协会），57
Associative arrays（联关数组），280－282
hashes（散列），84，283
implementation（实现），282
operations（操作），281－282
Associativity of operators（结合性操作符），316－318
Asynchronous message passing（异步消息传递），582
Atom（原子），52
Atomic propositions（原子命题），685
Attribute computation functions（属性计算函数），139
Attributes（属性），135
binding to variables（变量绑定），208
computing values（计算值），138－139
inherited（继承），135

intrinsic（内在的），136
synthesized（合成的），135
APT，25
awk，83
Axiomatic semantics（公理语义），143～155
assignment statements（赋值语句），145～146
evaluation of（评估），155
program proofs（程序证明），152～155
selection statements（选择语句），148～149
sequences（顺序，系列），147～148
weakest preconditions（最弱前置条件），144
while loops（while循环），149～152
Axioms（公理），144

B

B，81
Babbage，Charles，88
Backtracking（回溯），698
Backus，John，22，45，46，47，59，119
Backus-Naur form (BNF)（巴科斯-诺尔范式），59，119～120
Backward chaining (Prolog)（Prolog语言的面向目标），697
BASIC，67～69，参见VB设计过程，67～68
evaluation（评估），68～69
overview of（概述），68
BASIC～PLUS（BASIC-PLUS语言），68
Bauer，Fritz，57
BCD (binary code decimal)（二进制编码的十进制），252
BCPL（BCPL语言），81
Bell Laboratories（Bell实验室），76
BINAC computer（BINAC计算机），44
Binary coded decimal (BCD)（二进制编码的十进制），252
Binary operators（二元操作符），313
Binary semaphores（二元信号量），567，580
Binding（绑定），207～211
ad hoc（特别），419
attributes to variables（属性到变量），208
deep（深的），419
dynamic（动态的），208，511
dynamic type（动态类型），210～211
exceptions to handlers（异常处理程序），606
shallow（浅的），419
static（静态的），511
type（类型），209～211
Binding time（绑定期），208
BLISS（BLISS语言），7
Blocks（块），227～229，459～460
Block-structured language（块结构语言），228
BNF (Backus-Naur form)（巴科斯-诺尔范式），59～60，61，119～131
Body packages（体包），475
Borland JBuilder，34
Boolean expressions（布尔表达式），329～331
Boolean types（布尔类型），252
Bottlenecks (von Neumann)（冯·诺依曼瓶颈），30
Bottom-up parsers（自底向上语法分析器），176，177～178，188～195
Bottom-up resolution (Prolog)（自底向上归结），697
Bound variables（绑定变量），651
Boxing（封包），538
Breadth-first searches (Prolog)（Prolog语言的宽度优先搜索），698
break statements（break语句），352
Building lexical analyzers（构造词法分析器），169～175
Burroughs computers（Burroughs的计算机），61
Business applications（商务应用），6
Byron，Augusta Ada，88，89
Byte code（字位码），31
byte type（字节类型），250

C

C，24，81～83
character strings（字符串），254～255
encapsulation in（封装于），496
enumeration types（枚举类型），259～260
pointers（指针），295～297
prototypes（原型，样机），387
typedef（保留字typedef），224
C89，82
C99，82
C#（C#语言），105～108
abstract data types in（抽象数据类型），486～488
access modifiers（存取修饰符），486
assemblies（组合），498
dynamic binding（动态绑定），534～535
evaluation of support for OOP（支持OOP的求值），535
generic methods（通用方法），428～429
inheritance（继承），534
internal（内部的），486
named constants（命名常量），288
nested classes（嵌套的类），535
object-oriented programming (support for)（支持OOP），533～535
properties（属性），487
switch statements（switch语句），106
threads（线程），590～592

C++ (C++语言), 94－97
  abstract classes (抽象类), 528
  abstract data types (抽象数据类型), 482－485
  character strings (字符串), 254－255
  constructors (构造器), 483
  data members (数据成员), 482
  destructors (析构器), 483
  dynamic binding (动态绑定), 526－528
  encapsulation (封装), 496－497
  enumeration types (枚举类型), 260
  evaluation of support for OOP (支持OOP的评价), 529－530
  exception handling (异常处理), 615－620
  generic functions in (通用函数), 424－426
  information hiding (信息隐藏), 485
  inheritance (继承), 519－526
  member functions (成员函数), 482
  named constants (命名常量), 238
  namespaces (命名空间), 499－500
  object-oriented programming support for (支持OOP), 519－530
  overview of (概述), 94－95
  parameterized abstract data types (参数化抽象数据类型), 492－493
  pointers (指针), 295－297
  prototypes (原型), 387
  pure virtual functions (纯虚函数), 528
  typedef (给予字typedef), 224
  using, 500
Calculus (谓词演算), 688－690
Call chains (调用链), 447
Calls (调用)
  semantics for (对于…的语义), 441
  subprograms (子程序), 385
Cambridge Polish, 649
Cambridge University (剑桥大学), 45
Camel notation (camel符号), 205
Canonical LR (Canonical LR文法), 191
Case-sensitivity (大小写敏感性), 204
case statements (case语句), 353－354
Casts (explicit type conversion) (显式类型转换), 327
Categories (种类)
  of arrays (数组的), 265－267
  of concurrency (并发的), 558－559
  of languages (语言的), 24－25
C-based languages (基于C的语言), 202
CBL, 63
C89 Boolean types (C89语言的布尔类型), 252
C99 Boolean types (C99语言的布尔类型), 252
Central processing unit (CPU) (中央处理器), 20－21
CGI (Common Gateway Interface) (通用网关接口), 85, 104
Chain-offset (链偏移), 454
Characters (字符)
  lexical analysis (词法分析), 170
  types (类型), 255
Character string types (字符串类型), 253－258
Checked exceptions (要检测的异常), 625
Child packages (Ada 95) (Ada 95的子包), 538－539
Chomsky, Noam, 119
Chi Hoang-wei/Bnsll (合著者), 88
CIR (class instance record) (类实例记录), 547
Classes (类), 509
  abstract (抽象), 511, 528
  nested (嵌套的), 516, 533, 535
  Thread (线程), 585－586
Class instance record (CIR) (实例记录), 547
Class methods (类方法), 510
Class variables (类变量), 510
Clausal forms (子句形式), 346－347, 687－688
Clauses (子句)
  accept (accept子句), 573
  declare (declare子句), 228
  finally (finally子句), 627－628
  form (形式), 346－347
  Horn (霍恩子句), 690
  others (others子句), 269
  throws (throws子句), 625
Clients (of abstract data types) (抽象数据类型的客户), 472
Closed-world assumptions (Prolog) (Prolog语言的封闭世界假设), 710－711
COBOL (COBOL语言), 62－67
  design process (设计过程), 53－64
  evaluation (评估), 64－65
  historical background of (历史背景), 63
  writability (可写性), 65
Code (代码)
  building (functions) (构造函数), 665－666
  byte (字节码), 97
  Short Code (pseudocodes) (短码的初代码), 44－45
Coercion (强制转换), 219
  in expressions (表达式中的), 325－327
Collection<?> wildcard type (Collection通配类型), 428
Colmerauer, Alain, 86, 692
Column major order (按列存放), 274
COMmerical TRANslator (COMTRAN) (COMTRAN语言), 63

Common Gateway Interface (CGI) (通用网关接口), 85, 104
COMMON LISP (COMMON LISP语言), 55–56, 666–667
Communications of the ACM (ACM通信杂志), 59, 643
Compatibility of types (类型兼容性), 219
Compatible types (兼容类型), 225
Competition synchronization (竞争同步), 560
  Ada (Ada语言), 557–559
  Java threads (Java线程), 586–587
  monitors (管程), 569
  semaphores (信号量), 566–568
Compilation (编译), 27–30
Compleat tasks (完成的任务), 579
Complexity of parsing (语法分析的复杂性), 178
Compound assignment operators (复合赋值运算符), 333–334
Compound terms (复合项), 685
Computer architecture (计算机体系结构), 20–22
COMTRAN (COMTRAN语言), 65
Concurrency (并发),
  Ada (support for) (支持Ada语言), 575–583
  categories of (种类), 558–559
  C# threads (C#线程), 590–592
  design issues (设计问题), 563
  Java threads (Java线程), 583–590
  message passing (消息传递), 570–583
  monitors (管程), 568–570
  motivation for studying (学习动机), 559
  semaphores (信号量), 563–568
  statement level (语句级), 559–592
  subprogram-level (子程序级), 559–562
Conditional expressions (条件表达式), 319
Conditional targets (条件目标), 333
Conjunctions (合取), 695
Canonical LR (Canonical LR文法), 191
Consequent (结论项), 687
Consistency (of tags) (标志一致性的), 288
Constants (常量)
  enumeration (枚举), 259
  manifest (明示), 238
  named (命名常), 236–238
  readonly-named (只读方式命名的), 238
Construined variant variables (受限变体变量), 288
Constructors (构造器), 483
Constructs (构造)
  iterator (迭代), 356–357
  multiple selection (多重选择), 350–356
Context-free grammars (上下文无关文法), 119–131
Continuation (续域), 686
Control expressions (控制表达式), 346
Control flow (控制流)
  exception handling (异常处理), 607
  Scheme (Scheme语言), 654–655
Control statements (控制语句), 12–13, 344
Control structures (控制结构), 345
Conversion (转换)
  explicit type (显式类型转换), 327
  implicit (隐式的), 325–327
  narrowing (窄化), 324
  types (类型), 324–327
  widening (宽化), 324
Cooper, Alan, 70–71
Cooper, Jack, 88
Cooperation synchronization (合作同步), 560
  Ada (Ada语言), 576–577
  Java threads (Java线程), 587–590
  monitors (管程), 569
  semaphores (信号量), 564–566
Coroutines (协同程序), 77, 431–433
  quasi-concurrent (准协同程序), 558
Correctness (正确性)
  partial (部分的), 152
  total (全部), 152
Cost of programming languages (程序设计语言的代价), 18–20
Counter-controlled loops (计数器控制的循环), 357–365
  design issues (设计问题), 358
Counters, reference (计数器, 引用), 303
CPU (central processing unit) (中央处理器), 20–21
Currie, Malcolm, 88
Cut (Prolog), 708

D

Dahl, Ole-Johan, 76
Dangling pointers (悬挂指针), 295–294
Dangling references (悬挂引用), 293
Dartmouth College (Dartmouth学院), 67
Data abstraction (数据抽象), 16, 22, 也参阅抽象数据类型
  design issues (设计问题), 494
Databases (Prolog) (Prolog语言的数据库), 91
Data members (数据成员), 432
Data structures (数据结构), 13
Data types (数据类型), 248
  array types (数组类型), 263–280
  associative arrays (关联数组), 281–282
  Boolean (布尔), 252

character（字符），253
character string types（字符串类型），253－258
complex（复杂的），251
decimal（十进制），252
enumeration（枚举），258－261
floating point（浮点），250－251
integer，250
LISP（LISP语言），647－648
numeric（数值的），249－252
ordinal（有序的），258－263
pointers（指针），291－304
primitives（基本的，原始的），249－253
record types（记录类型），282－286
subrange（子范围），261－263
union types（联合类型），286－290
user-defined ordinal types（用户定义的序数类型），258－263
Deadlock（死锁），563
Deallocation（释除分配），215
objects of（释除分配对象），515－516
Decimal data types（小数数据类型），252
Declarations（声明）
variables（变量），209－210
Declarative languages（陈述性语言），684
Decorating parse trees（修饰语法分析树），138
Declarative semantics（说明语义），600
Declare clauses（Declare子句），228
Deep access（深访问），461－463
Deep binding（深绑定），419
Deferred reference counters（延迟引用计数器），301
Delphi，97
Denotational semantics（指称语义），155－161
assignment statements（赋值语义），159－160
evaluation（评估），160－161
expressions（表达式），158－159
logical pretest loops（测试先测试循环），160
state of programs（程序状态），158
Department of Defense (DoD)（美国国防部），63，87－89
Depth-first searches (Prolog)（深度先行搜索），698
Dereferencing（间接引用），292
Derivations（派生），121－123
leftmost（最左的），122
Derived classes（派生类），509
Derived types（派生类型），222－223
Descriminants（判别式），288
ordinal types（序数类型），259
union types（联合类型），287
Descriptors（描述符），249
Design issues
arrays（数组），264
character strings（字符串），252
counter controlled loops（计数器控制循环），358
functions（函数），429
logically controlled loops（逻辑控制循环），366
multiple selection constructs（多选择结构），350－351
names（名称），203
pointers（指针），292
selection constructs（选择结构），346
two-way selectors（双向选择符），346
union types（联合类型），287
user-defined ordinal types（用户定义的序数类型），259
Design process（设计过程）
Ada，88－89
ALGOL 60，59－60
ALGOL 68，77－78
BASIC，67－68
C#，105
C++，94－95
COBOL，63－64
Fortran，47
Java，97－98
LISP，52
PL/I，73
Prolog，86
SIMULA 67，76－77
Smalltalk，92
Destructors（析构器），483
Diagrams（关系图），171－172
Diamond inheritance（菱形继承），514
Dictionaries（字典），104
Dijkstra，Edsger，74，372，569
Direct left recursion（直接左递归），184
Discriminants（判别式），288
Discriminated unions（判别联合），288
Disjoint (tasks)（不相交任务），560
Displays（显示），459
DLL (dynamic linked library)（动态链接库），484
Documents（文档）
XML（可扩展标记语言），108
XSLT（可扩展语言转换），108－109
DoD (Department of Defense)
Ada，87－89
COBOL，63
Domains (programming)（程序设计定义域），5－7
Do statements (Fortran 95)（Fortran 95的Do语句），358－360
Dot notation (for records)（Dot符号），284
Double precision（双精度），251

Double types（双精度类型），351
Dynabook（笔记本型号），92
Dynamic binding（动态绑定），210，511
  Ada 95，537－538
  C#，534－535
  C++，526－528
  Java，532－533
  methods（方法），516
  object-oriented programming（OOP），516
  type（类型），210－211
Dynamic chains（动态链），447
Dynamic length strings（动态长度字符串），256
Dynamic linked library (DLL)（动态链接库），498
Dynamic links（动态连接），444
Dynamicscope（动态范围），232－233
  evaluation（评估），233
  implementing（实现），461－464
Dynamic semantics（动态语义），140－161
  axiomatic semantics（公理语义），143－155
  denotational semantics（指称语义），155－161
  operational semantics（操作语义），141－142
  Dynamic type binding（动态类型绑定），210－211
Dynamic type checking（动态类型检查），319

E

Eager approach（积极方法），301
EBNF（Extended BNF）（扩展的巴科斯－诺尔范式），131－134
ECMA（European Computer Manufacturers Association）（欧洲计算机制造协会），101
Edinburgh syntax（Edinburgh语法），693
Edwards，D.J.，649
Eiffel，96－97
Elaboration（精立），216
Elemental array operations（元素数组操作），270
Elliptical references（省略引用），285
elsif Clause（else子句），355
Encapsulation（封装），23，475，493－498
  Ada，497－498
  C，496
  C#，498
  C++，496－497
  naming（命名），498－503
  nested subprograms（嵌套子程序），495
Enumeration constants（枚举常量），259
Enumeration types（枚举类型），258－261
enum types（enum类型），259
Environment pointer (EP)（环境指针），440，445
Environments（环境）
  local referencing（局部引用），395－397
  programming（程序设计），33－34
Errors in expressions（表达式中的错误），327－338
European Computer Manufacturers Association (ECMA)（欧洲计算机制造协会），101
Evaluation（计值）
  of Ada，89－90
  of Ada (concurrency)，582－583
  of ALGOL 60，60－61
  of ALGOL 68，78－79
  of arrays（数组），273
  of axiomatic semantics（公理语义），155
  of BASIC，68－69
  of C，82
  of C#，107
  of C# support for OOP（支持OOP的C#），535
  of C# support for threads（支持线程的C#），592
  of C++，95
  of C++ abstract data types（C++抽象数据类型），484－485
  of C++ support for OOP（支持OOP的C++），529－530
  of COBOL，64－65
  of denotational semantics（指称语义），160－161
  of dynamic scoping（动态作用域），233
  of enumeration types（枚举类型），261
  of Fortran，49－50
  of Java，99－100
  of Java support for OOP（支持OOP的Java），533
  of Java threads（Java线程），590
  of languages（语言），7－20
  of LISP，54
  of monitors（管程），569－570
  of operational semantics（操作语义），142
  of Pascal，80
  of Perl，84－85
  of PL/I，74
  of Prolog，87
  of records（记录），286
  of references（引用），298－299
  of semaphores（信号量），568
  of Smalltalk，93
  of Smalltalk support for OOP（支持OOP的Smalltalk），518
  of static scoping（静态作用域），229－231
  of strings（字符串），256－257
  of subrange types（子范围类型），261－263
  of unions（联合），290
Event Handlers（事件处理器），630
Event handling（事件处理），630
  Java，631－636
Event listeners（事件监听器），632

Events, 650, (classes) (事件(类)), 610
Exception handling (异常处理), 602–605
  Ada, 608–615
  C++, 615–620
  design issues (设计问题), 605–608
  Java, 620–630
  reliability (可靠性), 17
Exceptions (异常), 328, 603
Exception handlers (异常处理器), 603
Exclusivity of objects (纯对象模型), 512
Expert systems (Prolog) (Prolog的专家系统), 714–715
Explicit declarations (显式声明), 209–210
Explicit heap-dynamic variables (显式堆动态变量), 217–218
Explicit type conversion (显式类型转换), 327
Expressions (表达式)
  arithmetic (算术的), 313–322
  associativity (结合性), 316–318
  Boolean (布尔), 329–331
  coercion (强制转换), 325–327
  conditional (条件的), 319
  control (控制), 346
  errors in (错误), 327–328
  lambda (lambda表达式), 651
  mixed-mode (混合模式), 325
  operand evaluation order (操作数求值次序), 319–321
  operator evaluation order (操作符求值顺序), 313–318
  parentheses() (圆括号), 318
  regular (正规的, 正则的), 255
  relational (关系的), 328–329
  unambiguous grammars for (无歧义性文法), 126–127
Expressivity (表达式), 16
Extended accept clauses (扩展的accept子句), 576
Extended ALGOL (扩展的ALGOL语言), 7
Extended BNF (EBNF) (扩展的巴科斯-诺尔范式), 131–134
extends (reserved word) (保留字extends), 427
eXtensible HTML (可扩展HTML语言), 参见XHML
eXtensible Markup Language (XML) (可扩展标记语言), 108
eXtensible Stylesheet Language Transformations (XSLT) (可扩展样式表转换), 108–109
Extensions (EBNF) (扩展巴科尔-诺尔范式), 131–134

F

Fact statements (Prolog) (Prolog事实语句), 694
Farber, D. J., 76
Feature multiplicity (特性重复性), 9
Fetch-execute cycles (取行-执行周期), 24
Fields (record) (记录的域), 283
Fifth Generation Computing Systems (FGCS) (第五代计算机系统), 692
Files (header) (文件(头)), 496
Finalization (结束化), 607
finalize method (finalize方法), 531
finally clause (finally子句), 627–628
final method (final方法), 531
Finite automata (有限自动机), 171
Finite mappings (有限映射), 264
First-order predicate calculus (一次谓词演算), 685
Fixed heap-dynamic arrays (固定堆动态数组), 266
Fixed stack-dynamic arrays (固定栈动态数组), 265
Flex (Flex语言), 92
Floating-point data types (浮点数据类型), 250–251
Floating-point hardware (浮点数硬件), 46
float type (float类型), 250
FLOW MATIC (FLOW MATIC语言), 63
FLPL (Fortran List Processing Language) (Fortran表处理语言), 52
Formal parameters (形参), 387
Forms (形式)
  EBNF (Extended BNF) (扩展的巴科斯-诺尔范式), 131–134
  names (名字), 203–204
for statements (for语句)
  Ada, 360–361
  C-based languages (基于C的语言), 361–363
  Python, 363–366
Fortran, 46–51
  evaluation (评价), 49–50
  HPF (High-Performance Fortran) (高性能Fortran语言), 593–594
  List Processing Language (FPLPL) (表处理语言), 52
Fortran 77, 48
Fortran 90, 48–49
Fortran 95, 13, 49
  Do statements (Do语句), 358–360
  named constants (命名常量), 258
Fortran 2003, 49
Fortran II, 48
Fortran IV, 48–49
Fortran List Processing Language (FLPL) (Fortran表处理语言), 52
Forward chaining (Prolog) (Prolog前向链式), 697
Fraction ranges (小数范围), 251
Free unions (自由联合), 287
Fully attributed trees (完全属性树), 136
Fully qualified references (完全限定的引用), 285

Functional forms（函数形式），645，664－665
Functional programming languages（函数式程序设计语言），642－642
  applications（应用），675－676
  COMMON LISP（COMMON LISP语言），666－667
  fundamentals of（基础），645－646
  Haskell（Haskell语言），670－678
  and imperative languages（命令式语言），676－678
  LISP（LISP语言），646－650
  mathematical functions（数学函数），643－645
  ML（ML语言），667－670
  Scheme（Scheme语言），650－666
Function composition（函数复合），645，664－665
Functions（函数），392－393
  building code（构造代码），665－666
  defining Scheme functions（定义Scheme函数），651－652
  design issues（设计问题），429
  list functions in Scheme（Scheme中的表函数），650－651
  mathematical functions in Scheme（Scheme中的数学函数），650－651
  numeric predicate functions in Scheme（Scheme中的数值谓词函数），653－654
  pure virtual（纯虚），528
  return values（返回值），429－430
  side effects（副作用），320，429
Functors（函数符），685

G

GAMM（应用数学与力学协会），57
Garbage（垃圾），294
Garbage collection（垃圾收集），301
Generality（一般性，通用性），20
General Purpose Simulation System (GPSS)（通用目的模拟系统），25
Generate and test (Prolog)（Prolog的产生与测试），709
Generations (of languages)（语言的代），118－119
Generic functions in C++（C++的通用函数），424－426
Generic methods in C# 2005（C#的通用方法），428－429
Generic methods in Java 5.0（Java 5.0的通用方法），422－429
Generic subprograms（通用子程序），422－429
Generic subprograms in Ada（Ada中的通用子程序），422－424
Generic units（通用单元），422
Glennie, Alick E., 46
Goals（目标），690，696
Goal statements (Prolog)（目标语句（Prolog）），695－696
Gosling, James, 98, 622－623
GPSS（通用仿真系统），25
Grammars（文法），120
  ambiguity（二义性），124－125
  associativity of operators（操作符的结合性），128－130
  attribute（属性），134－140
  context-free（上下文无关文法），119－133
  derivations（推导），121－123
  operator precedence（操作符的优先级），125－128
  parse trees（语法分析树），123－124
  recognizers（识别器），118
  unambiguous for expressions（表达式的非二义性），126－127
Graphical user interface. *See* GUI（图形用户接口）
Graphs (state diagrams)（状态图），171
grep command (UNIX)（UNIX系统的grep命令），15
Griswold, R. E., 76
Guarded commands（守卫的命令），372－376
Guards（守卫），564
GUIDE（GUIDE库（包体），72
GUI (graphical user interface)（图形用户接口），95

H

Handles，177，189
Handling（处理）
  events事件，参见事件处理
  exceptions（异常）参见异常处理
Hansen, Per Brinch, 569
Hardware (floating-point)（浮点数硬件），46
Hashes（散列），84，285
Haskell（Haskell语言），56，670－678
Header files（头文件），496
Heap-dynamic arrays（堆动态数组），266
Heap-dynamic variables（堆动态变量），291
Heaps（堆），291
  management（管理），300－304
Heavyweight tasks（重型任务），568
Heijlsberg, Anders, 97, 105
High-Order Language Working Group (HOLWG)（高阶语言工作组），88
High Performance Fortran (HPF)（高性能Fortran语言），593－594
Historical background（历史背景）
  of Ada，87－88
  of Algol 60，56－57
  of C，81－82
  of COBOL，63

of Fortran, 46～47
of Perl, 83～84
of Plankalkül, 42
of P L/I, 72
History-sensitive (历史敏感的), 215, 395
Hoare, C. A. R. ("Tony"), 15, 25, 79
HOLWG (High Order Language Working Group) (高阶语言工作组), 88
Hopper, Grace, 45, 63
Horn clauses (霍恩子句), 690
HPF (High-Performance Fortran), 593～594
Hybrid implementation systems (混合式实现系统), 31～32
Hypotheses (假设), 690

I

IAL (International Algorithmic Language) (国际算法语言), 58
IBM 704, 46, 47, 344
IBM Hursley Laboratory (England) (位于英国的IBM Hursley实验室), 73
IBM mainframe computers (IBM大型计算机), 10, 72
Identifiers (标识符), 14
Identity operators (标识操作符), 314
IEEE floating-point formats (IEEE浮点数格式), 251
IFIP (International Federation of Information Processing) (国际信息处理联合会), 79
Imperative languages (命令式语言), 20
Implementation (实现), 26～33
of array types (数组类型的), 273～280
of associative arrays (相关数组的), 282
of character string types (字符串类型的), 257～258
compilation (编译), 27～30
hybrid (混合的), 31～32
of Java (Java语言的), 32
of object-oriented constructs (面向对象结构的), 547～550
of parameters (参数的), 405～406
of pointers (指针的), 299～304
preprocessors (预处理器), 33
pure interpretation (纯解释), 30～31
of record types (记录类型的), 286
of references (引用的), 299～304
of state diagrams (静态图的), 171～175
of subprograms (子程序的), 440～464, 参照子程序
of union types (联合类型的), 290
of user-defined ordinal types (用户定义序数类型的), 263
Implicit declarations (隐式声明), 209～210
Implicit heap-dynamic variables (隐式堆动态变量), 218～219
import declaration (import声明), 501
#include, 33
Incremental mark-sweep, 303
Indexes (arrays) (数组下标), 264～265
Inference rules (推理规则), 144
Inferencing (推理), 696～699
Infix (binary operators) (Infix二元操作符), 313
Information hiding (信息隐藏)
Ada, 475～477
C++, 483
Information Processing Language 1 (IPL-1) (信息处理语言1), 51
Inheritance (继承)
Ada 95, 536～537
C#, 534
C++, 519～526
Java, 531～532
object-oriented programming (OOP) 509～510
Ruby, 542～543
Smalltalk, 518
Inherited attributes (继承属性), 135
Initialization (初始化), 238～239
of array types (数组类型的), 268～269
In mode semantic model (输入语语义模型), 397
Inout mode semantic model (输入输出类语义模型), 397
Instance data storage (实例数据存储), 547
Instance methods (实例方法), 510
Instance variables (实例变量), 510
Instantiation (实例化), 689, 693
Integers (整数), 250
Interfaces (Java) (Java语言的接口), 531～532
International Algorithmic Language (IAL) (国际算法语言), 58
International Conference on Information Processing (国际信息处理会议), 59
International Federation of Information Processing (IFIP) (国际信息处理联合会), 79
International Standards Organization (ISO) (国际标准组织), 101
Interpreters (解释器)
LISP, 648～650
Scheme, 650
Interrupt (中断), 585
Intrinsic attributes (内在属性), 136
Intrinsic limitations (Prolog) (Prolog语言的内在限制), 713
int type (int类型), 250

IPL（信息处理语言），51
ISO (International Standards Organization)（国际标准组织），101
IT，56
Iteration（迭代）
based on data structures（基于数据结构），369－371
constructs（结构），352
Iterative statements（迭代语句），355－371
Ada，360－361
C based languages（基于C的语言），361－363
Fortran 95，358－360
Python，363－366
Iterators（迭代器），369
Iverson，Kenneth E.，75

J

Jagged arrays（参差的数组），271
Java（Java语言），22，97－101
abstract data types（抽象数据类型），485－486
dynamic binding（动态绑定），532－533
evaluation of support for OOP（支持OOP的评估），532
event handling（事件处理），631－636
event listeners（事件监听器），632
exception handling（异常处理），620－630
generic methods in（通用方法），426－428
GUI components（GUI组件），631－632
implementation（实现），32
inheritance（继承），531－533
interfaces（接口），531－532
named constants（命名常量），258
names（名字），204
nested classes（嵌套类），533
object-oriented programing (support for)（支持OOP），530－533
packages（包），500－501
Swing package（Swing包），631
threads（线程），582－593
JavaScript，101－103
objects（对象），544－545
object model of（对象模型），545－547
Java Server Pages (JSP)（Java服务器网页），109－110
Java Server Pages Standard Tag Library (JSTL)（Java服务器网页标准标记程序库），109
Java Virtual Machine (JVM)（Java虚拟机），99
JIT (Just In Time) compilers（适时编译器），99，169
JOVIAL（Jules同际代数语言），59
JScript，101
JSP (Java Server Pages)（Java服务器网页），109－110
JSTL (Java Server Pages Standard Tag Library)（Java服务器网页标准标记程序库），109－110
Just in Time（适时编译器，参见JIT）
JVM (Java Virtual Machine)（Java虚拟机），99，587

K

Kay，Alan，92
Kemeny，John，67
Kernighan，Brian，82
Keys（关键字），281
Keyword parameters（关键字参数），388
Keywords（关键字），70，204－205
Knuth，Donald，191，379
Kurtz，Thomas，67

L

Lambda expressions（Lambda表达式），644，651
Luning and Ziegler system，46
Languages（语言，参见特殊语言）
Ada，87－91
Ada 95，91
ADMACO，63
ALGOL 58，58－59
ALGOL 60，5，59－68
ALGOL 68，77－79
ALGOL-W，79
APL，75－76
APT，25
awk，83
B，81
BASIC，12，67－69
BASIC-PLUS，68
BCPL，81
BLISS，7
block structured（块结构的），238
C，81－83
C89，82
C99，82
C#，105－106
C++，94－97
categories of（种类），24－25
C-based（基于C的），202
CBL，63
COBOL，6，62－67
COMMON LISP，55，668－669
Delphi，97
Eiffel，96－97
evaluation criteria（评估准则），7－20
evolution，（演化，参见语言的演化）

Flex, 92
FLOW MATIC, 63
Fortran, 46－51
functional programming（函数式程序设计）, 645－646, 参见函数式程序设计语言
genealogy of（家谱）, 41
generators（生成器）, 118－119
GPSS, 25
Haskell, 56, 670－678
IAL (International Algorithmic Language)（国际算法语言）, 58
imperative（命令式的）, 20
Java, 97－101
JavaScript, 101－103
JOVIAL, 59
JScript, 101
JSP, 109－110
ksh, 83
LISP, 51－55
LiveScript, 101
logic programming（逻辑程序设计，参见逻辑程序设计语言）
LOGO, 92
MAD, 59
markup/programming hybrids, 24,（标记/程序设计复合）108－110
MATH MATIC, 56
metalanguages（元语言）, 120
Miranda, 56
ML（元语言）, 56, 667－670
NELIAC, 59
NPL, 73
orthogonality（正交性）, 10－12
parameter-passing methods for（参数传递方法）, 400－405
Pascal, 79－81
Perl, 83－85
PHP, 103
Plankalkül, 40－43
PL/C, 74
PL/I, 72－75
Prolog, 5, 85－87
Prolog++, 87
pseudocodes（虚拟码）, 43－46
Python, 104
readability（可读性）, 8－15
reasons for studying（学习的缘由）, 2－5
recognizers（识别器）, 118
regular（正规的，正则的）, 171
RPG, 25
Ruby, 104－105
Scheme, 6, 55
scripting（脚本化）, 101－105
selection of（选择）, 3
simplicity（简单性）, 9－10
SIMULA 67, 76－77
SIMULAI, 76
SmallTalk, 91－94
SNOBOL, 75－76
sotirce language（源语言）, 28
SQL (Structured Query Language)（结构查询语言）, 714
strongly typed（强类型的）, 219－221
syntax（语法）, 14－15
Tcl, 83
trade-offs（权衡）, 25－26
VDL (Vienna Definition Language)（Vienna定义语言）, 142
visual（可视的）, 24
Visual BASIC, 69
writability（可写性）, 15－16
XSLT, 108－109
Laning and Zierler system（Laning和Zierler系统）, 46, 56
Lazy approach（懒惰方法）, 301
Lazy evaluation（懒惰求值）, 674
LCF (Logic for Computable Functions)（可计算函数逻辑）, 56
Leaf nodes (intrinsic attributes)（内部属性的叶节点）, 136
Learning new languages（学习新语言）, 3－4
Left factoring（提取左因子）, 187
Left-hand side (LHS)（左手边）, 120
Leftmost derivations（最左派生）, 122
Left recursive rules（左递归规则）, 129, 184
Length (string options)（串选择长度）, 256
Lerdorf, Rasmus, 103, 276－278
Level numbers (of records)（记录的层号）, 283
Lexemes（词素）, 117, 170
Lexical analysis（词法分析）, 169－175
Lexical analyzers（词法分析器）, 28, 169－175
LHS (left hand side)（左手边）, 120
Lifetime of variables（变量的生存期）, 215, 234
Lightweight tasks（轻量任务）, 560
Limited dynamic length strings（有限动态长度串）, 256
Limited private types（受限定有类型）, 477

Linkage (subprograms) (子程序的链接), 440
Linkers (链接器), 443
Linking (链接), 30, 442
lint program (UNIX) (UNIX的lint程序), 17
LISP (LISP语言), 6, 51–55
  COMMON LISP, 55–56, 666–667
  data structures (数据结构), 52–53, 647–648
  data types (数据类型), 647–648
  of evaluation (评价的), 54
  interpreters (解释器), 648–650
  overview of (概述), 52–54
  syntax (语法), 53–54
List assignments (链表赋值), 336–337
List comprehensions (链表综合), 672–673
Lists (链表)
  simple (简单), 647
  structures (Prolog) (Prolog语言的链表结构), 702–707
Literals (overloaded) (重载的字面常量), 260
Liveness (活性), 562
LiveScript, 101
LL grammar class (LL文法类), 177, 184–187
Loading (加载), 30, 443
Load modules (加载模块), 30
Local_offset (局部偏移), 447
Local referencing environments (局部引用环境), 395–397
Local variables (局部变量), 395
Locks-and-keys approach (锁与钥匙的方式), 299–300
Logical concurrency (逻辑并发), 558
Logically controlled loops (逻辑控制循环), 366–368
Logical Theorist (逻辑王论家), 51
Logic for Computable Functions (LCF) (可计算函数的逻辑), 56
Logic programming overview (逻辑程序设计概述), 680–692
Logic programming languages (逻辑程序设计语言), 24, 684
  applications (应用), 713–716
  predicate calculus (谓词演算), 684–688
  programming (程序设计), 690–692
  Prolog, 693–713
LOGO (LOGO图形系统), 93
Loop invariants (循环不变式), 149–150
Loop parameters (重环参数), 357
Loops (循环), 352
  axiomatic semantics (公理语义), 149–152
  based on data structures (基于数据结构), 369–371
  counter-controlled (计数器控制的), 357–366
  denotational semantics (指称语义), 160
  logically controlled (逻辑控制), 366–368
  user-located control-al mechanisms (用户定位的循环控制机制), 368–369
Loop variables (重环变量), 357
Lost heap-dynamic variables (丢失的堆动态变量), 294–295
LR parsing algorithms (LR语法分析算法), 191–195
L-value (左值), 206

M

Macros (宏指令), 35
MAD language (MAD语言), 59
Management of heaps (堆管理), 300–305
Manifest constants (说明常量), 338
Mark-sweep (标志-清除法), 301
Markup/programming hybrid languages (标记/程序设计复合语言), 108–110
Matching (Prolog) (Prolog语言的匹配), 697
Mathematical functions (数学函数), 642–645
MATH-MATIC, 56
Matsumoto, Y., 104
Mauchly, John, 44
McCarthy, John, 6, 52, 648
McCracken, Daniel, 25
Member functions (成员函数), 482
Memory leakage (内存泄漏), 99, 295
Message interface (消息接口), 509
Message passing (concurrency) (并发的消息传递), 570–585
Message protocol (消息协议), 509
Messages (消息), 509
Meta Language (ML) (元语言), 56
Metalanguages (元语言), 120
Metasymbols (元符号), 132
Methodologies of software design (软件设计方法学), 22–23
Methods (方法), 509
  abstract (抽象), 510
  class (类), 510
  of describing semantics (描述语义法), 140–161
  of describing syntax (描述法的), 117–134
  dynamic binding (动态绑定), 576
  final (final语句), 521
  finalize (finalize语句), 531
  instance (实例), 510
  overriding (覆写), 510
  parameter-passing (参数传递), 387–405
  prototype (原型, 样机), 410

Meyer, Bertrand, 96
Milner, Robin, 56
MIMD (Multiple-Instruction Multiple-Data)（多指令多数据），557
Minsky, Marvin, 52
MIT（麻省理工学院），52，55
Mixed-mode assignment（混合模式赋值），337－338
Mixed-mode expressions（混合模式表达式），325
ML，56，211，214，667－670
Models, tracing（模型，跟踪），701
Modifiers（修饰符）
  access (C#)（C#的存取），486
  unsafe，298
mod operator (Ada)（取模操作符），315
Modules (Ruby)，502－503
Modules (load)，30
Monitors（管程），568－570
evaluation of（评估），569－570
Multidimensional arrays as parameters（多维数组作为参数），411－415
Multiple inheritance（多继承），510，514－515
Multiple-Instruction Multiple-Data (MIMD)（多指令多数据），557
Multiple selection constructs（多选择结构），350－356
Multiprocessor architectures（多处理器体系结构），557－558
Multithreaded（多线程），558

N

Named constants（命名常量），236－238
Names（名字），203－205
  design issues（设计问题），203
  encapsulation（封装），498－503
  forms（形式），203－204
  patterns（模式），169－170
  variables（变量），306
Namespaces (C++)（C++的命名空间），499－500
Name type equivalence（名字类型等价性），221
Naming encapsulations（命名封装），498－503
  Ada，501－502
  C++，499－500
  Java，500－501
  Ruby，502－503
Narrowing conversion（窄化转换），324
National Physical Laboratory (England)（英国国家物理实验室），73
Natural language processing (Prolog)（Prolog的自然语言处理），715
Natural operational semantics（自然操作语义），141
Naur, Peter, 59－60，119. 参见BNF (Backus-Naur form)
Negation problem (Prolog)（Prolog的否定问题），711－713
NELIAC（NELIAC语言），59
Nested classes（嵌套类），516
  C#，525
  Java，533
Nested subprograms（嵌套子程序），347，495
  implementing（实现），451－459
Nesting-depth（嵌套深度），454
Nesting selectors（嵌套选择符），347－350
Netscape（Netscape搜索引擎），101
Newell, Alan，51
new method（new方法），536
New Programming Language (NPL)（NPL语言），73
Nonstrict（非限制），673
Nonterminal symbols（非终结符），120
Notation（符号）
  dot（点），284
  two complement（两补互补），250
NPL (New Programming Language)（NPL语言），73
Numbers (level)（层次的数目），283
Numeric types（数值调词函数），249－252
Nygaard, Kristen, 76

O

Object-oriented programming (OOP)，93，94－97，508－511
  Ada 95 (support for)（支持Ada 95），535－540
  C# (support for)（支持C#），533
  C++ (support for)（支持C++），519－530
  design issues（设计问题），512－516
  dynamic binding（动态绑定），511
  inheritance（继承），509－510
  JavaScript (object model of)（JavaScript语言的面向对象模型），543－547
  Java (support for)（支持Java），530－535
  Ruby (support for)（支持Ruby），540－543
  Smalltalk (support for)（支持Smalltalk），516－518
Objects（对象），249，471，509
  allocation of（分配），515－516
  deallocation of（回收），515－516
  exclusivity of（互斥），512
  JavaScript，543－547
  protected (Ada 95)（保护），580－581
Operands (evaluation order)（操作数求值顺序），319－322
Operating systems (UNIX)（UNIX操作系统），7
Operational semantics，141－142
  evaluation（操作语义），142

Operator overloading（操作符重数），9
Operator precedence（操作符优先级），125
rules（规则），314
Operators（操作符）
APL，317
associativity（结合性），317～318
binary（二元的），313
C++，102
compound assignment（复合赋值），333～334
evaluation order（求值顺序），313～319
identity（标识），314
infix（中缀），313
overloading（重载），322—324，430～431
parentheses（圆括号），318
precedence（优先级），314～315
relational（关系的），328
rem(Ada)（Ada语言的rem操作符），315
Ruby，318～319
ternary（三元的），313
unary（一元的），313
unary assignment（一元赋值），334～335
user-defined overloaded（用户定义重载），322，324，420
Optimization（优化），19
Ordinal types（序数类型），258～263
Origins of Backus-Naur form（巴科斯-诺尔范式的起源），119～120
Or-else，332
Orthogonality（正交性），10—12
others clause（others子句），269
Ousterhout，John，83
Out mode semantic model（语义模型的输出型），397
Overflow（超流，溢出），328
Overloaded literals（重载文字常量），260
Overloaded subprograms（重载子程序），394，420～422
Overloading operators（重载操作符），322～324，430～431
override（覆盖），534
Overriding methods（覆盖方法），510

P

Packages（包）
Ada，475～486，500～502
Child (Ada 95)（Ada 95语言的子包），538～539
Java，500～501
Swing，631
Pairwise disjointness tests（两两不相交测试），186
Papert，Seymour，92
Parameter passing methods（参数传递方法），400～405
Parameter profiles（参数描绘），386
Parameterized abstract data types，490～495
Ada，491～492
C# 2005，494—495
C++，492～494
Java，493～494
Parameters（参数），387～391
actual（实参），388
formal（形参），387
keyword（关键字），388
multidimensional arrays as（多维数组），411～415
pass-by-copy（按复制传递），403
pass-by-name（按名传递），404～405
pass-by-reference（按引用传递），403～404
pass-by-result（按结果传递），402～403
pass-by-value（按值传递），400～401
pass-by-value result（按值和结果传递），403～404
positional（位置的），388
subprograms（子程序），418～420
type-checking（类型检测），409～411
Parametric polymorphism（参数多态），422
params reserved word (C#)（C#语言中的保留字params），389～390
Parent classes（父类），509
Parentheses ( )（圆括号），318
Parse trees（语法分析树），28，123～124
if-then-else statement unambiguous grammars（if-then-else语句非歧义性文法），131
Parsing（语法分析）
bottom-up parsers（自底向上语法分析器），176，177～178，188—195
complexity of（复杂性），178
LL algorithms（LL算法），177
LR algorithms（LR算法），191—195
overview of（概述），175～176
recursive-descent（递归下降），177，179～184
top-down parsers（自顶向下语法分析器），176～177
traces（追踪），182—183，195
Partial correctness（部分正确性），152
Pascal，79～81
Pascal，Blaise，79
Pass-by-copy parameters（按复制传递参数），403
Pass-by-name parameters（按名传递参数），404～405
Pass-by-reference parameters（按引用传递参数），403～404
Pass-by-result parameters（按结果传递参数），391～392
Pass-by-value parameters（按值传递参数），400～401
Pass-by-value-result parameters（按值和结果传递参数），

403－409
Patterns (lexical analysis)（词法分析的模式），169－170
PDA (pushdown automata)（下推自动机），190
PDP-11 minicomputers（PDP-11型微型计算机），68
Perl，3，83－85
Perlis，Alan，50，56
PHP，102
Phrases（短语），189
Physical concurrency（物理并发），558
Plankalkül，41－43，58，64
PLIC，74
PLICS，74
PL/I，72－75
operational semantics（操作语义），162
PLUS，6
Pointers（指针），291－304
Ada，295
C，295－297
C++，295－297
dangling（悬挂），293－294
design issues（设计问题），292
evaluation（评估），298－299
implementation（实现），299－304
operations（操作），292－295
problems（问题），293－295
Polenky，F. P.，96
Polymorphic references（多态引用），511
Polymorphic subprograms（多态子程序），422
Polymorphism（多态性），422
parametric（参数的），422
type checking and（类型检查），513－514
Portability（可移植性），19
Positional parameters（位置参数），388
Postconditions（后置条件），143
Posttests (loops)（循环的后测试），357
pragma (Ada)，613
Precedence（优先级）
Boolean operators（布尔操作符），329－330
operators（操作符），314－315
Precision (sets)（精度性），251
Preconditions（前置条件），143
weakest（最弱的），144
Predicate calculus（谓词演算），688－690
Predicate functions（谓词函数），135
Preprocessor（预处理器），55
Pretests (loops)（循环的先测试），357
Primitive data types（基本数据类型），249－253
Priorities（优先级）
Ada (concurrency)（并发的Ada语言），579－580
Java threads (concurrency)（并发的Java线程），586
Private types（私有类型），476
Procedures (subprograms)（子程序(过程)），392－393
Process abstraction（过程抽象），470
Processes（进程），559
Processors，vector（处理器，矢量），557
Producer-consumer problem（生产者－消费者问题），560
Program proofs（程序证明），152－155
Program counters（程序计数器），21
Programming（程序设计）
domains（领域），5－7
data oriented（面向数据），23
environments（环境），33－34
object-oriented（面向对象），23，97
procedure oriented（面向过程），23
systems（系统），6
Programming languages, see also 程序设计语言
genealogy of（家谱），41
methodologies（方法学），22－33
reasons for studying（学习的原因），2－5
selection of（选择），3
Programs (See also Applications)（程序，另见应用）
state of（状态），158
Prolog（Prolog语言），6，85－87
closed-world assumptions（封闭世界假设），710－711
deficiencies（缺陷），707－712
expert systems（专家系统），714－715
fact statements（事实语句），694
inferencing（推理），696－699
intrinsic limitations（内在限制），713
natural language processing（自然语言处理），715
negation problem（否定问题），711－713
origins（起源），692
resolution order control（归结次序控制），707－710
terms（术语），693
Prolog++，87
Proof (program)（程序证明），152－155
Properties（属性），487
Propositions（命题），684，685－687
atomic（原子命题），685
protected interval，486
Protected objects (Ada)（Ada的保护对象），580－581
Protocols (subprograms)（子程序协议），386
Prototype methods（原型方法），418
Prototypes（原型），387
Proving theorems（定理证明），688－690
Pseudocode（伪代码，仿代码），43－46

Short Code（短代码），44－45
Spreadsheeting（快速编码），45
Pure interpretation（纯解释），30－31
Pure virtual functions（纯虚函数），528
Pushdown automata (PDA)（下推自动机），190
Python，184

Q

Quasi-concurrency（准并发），431，558
Quasi-concurrent coroutines（准并发协同程序），558
Queries（查询），686

R

Race conditions（竞态条件），561
Raised exceptions（产生的异常），503
Rand Corporation Jonniac computer（Rand公司的Jonniac计算机），51
Ranges（范域），251
RDBMSs (relational database management systems)（关系数据库管理系统），714
Readability（可读性），8－9
　conflicts（冲突），25
　reliability（可靠性），18
Recognizers (grammars)（识别器），134
Recognition (of languages)（语言识别），118
Records（记录）
　activation（启动，激活），441
　CIR (class instance record)（类实例记录），547
　definitions of（定义），283－284
　evaluation of（评估），286
　implementation（实现），356
　operations on（操作），285－286
　references（引用），284－285
Record types（记录类型），282－286
Rectangular arrays（长方数组），271
Recursion, implementing（实现递归），448－450
Recursive-descent parsing（递归下降语法分析），177，179－184
Recursive rules（递归规则），121
Reference variables（引用变量），291
References（引用），297－298
　counters（计数器），301
　elliptical（省略），285
　evaluation of（评估），298－299
　fully qualified（完全限定的），285
　implementation（实现），299－304
　polymorphic（多态的），571
　to record fields（记录域），284－285
　substring（子串），253
Referencing environments（引用环境），234－236
Referential transparency（引用透明性），321－322，646
Regular expressions（正则表达式），255
Regular languages（正则语言），171
Relational database management systems (RDBMSs)（关系数据库管理系统），714
Relational expressions（关系表达式），328－329
Reliability（可靠性），17－18
　aliasing（别名使用），18
　exception handling（异常处理），17
　readability（可读性），18
　type checking（类型检查），17
　writability（可写性），18
rem operator (Ada)（Ada语言的rem操作符），315
Reports（报告）
　ALGOL 58，59
　ALGOL 60，60
Representations of references and pointers（引用和指针的表示），299
Research Laboratory for Electronics（电子实验室），52
Reserved words（保留字），34，205
Resolution（归结），688
Resolution order control (Prolog)（Prolog语言的归结次序控制），707－710
Resumes (coroutines)（重新启动协同程序），431
Resumption（重新启动），606
Returned values (types of)（返回值类型），429－430
Returns (semantics for)（返回语义），441
RHS (right-hand side)（右边），120
Richards, Martin，81
Right-hand side (RHS)（右边），120
Rightmost derivation（最右推生），122
Right recursive rules（右递归规则），129
Ritchie, Dennis，82
Roussel, Philippe，86，692
Row major order（按行存储），274
RPG，25
Ruby，184－185
　abstract data types（抽象数据类型），488－490
　blocks（块），391－392
　dynamic binding（动态绑定），543
　evaluation of support for OOP（支持OOP的评估），543
　inheritance（继承），542－543
　modules（模块），502－503
　support for OOP（支持OOP），540－543
Rule of consequence（后果规则），146
Rules（规则）
　associativity（结合性），316－318
　inference（推理），144

left recursive（左递归），129
operator precedence（操作符优先级），314－315
recursive（递归），129
resolution（归结），688
right recursive（右递归），129
rule of consequence（后果规则），146
statements (Prolog)（Prolog的语句），694－695
Run-time stacks（运行时栈），445
resume（重新开始），431
Russel, S.B., 649
Rutishauser, H., 61
r-value（右值），207

S

Sammet, Jean, 110
Schedulers（调度程序），561
Scheme, 6, 55, 650－666
control flow（控制流），654－655
interpreters（解释器），650
predicate functions（谓词函数），658－659
list functions（链表函数），655－658
Schwartz, Jules I., 59
Scientific applications, 5
Scope (of variables)（变量的作用域），225－233
blocks（块），227－229
dynamic scope（动态作用域），232－233
(and) lifetime（生命期、寿命），234
static scope（静态作用域），225－231
Scripting languages（脚本语言），101－105
Scripts（脚本），83
Selection of languages（语言选择），3
Selection statements（选择语句），346－356
multiple（多），350－356
two way（双向），346－350
Selectors (nesting)（嵌套选择符），347－350
Semantics（语义），116－117
axiomatic（公理），143－155
for calls and returns（关于调用与返回），441
declarative（说明），690
denotational semantics（指称语义），155－161
dynamic（动态的），140－161
operational（操作），141－142
static（静态的），134－135
Semaphores（信号量），563－568
binary（二元），567
evaluation of（评估），568
Sentences（句子），117
Sentential forms（句型），122
Sequences (axiomatic semantics)（公理语义序列），147－148
Server tasks（服务器任务），573
Servlet container（Servlet容器），109
Shallow access（浅访问），463－464
Shallow binding（浅绑定），419
SHARE (computer user group)（SHARE计算机用户组），51，59，32，73
Shared inheritance（共享继承），514
Shaw, J. C., 51
Shift-reduce algorithms（移进－归约算法），190－195
Short-circuit evaluation（短路求值），331－332
Short Code（短代码），44－45
Short Range Committee（短范围委员会），64
Side effects (functions)（函数的副作用），320
Signatures（签名），386
SIMD (Single Instruction Multiple-Data)（单指令多数据），557
Simon, Herbert, 51
Simple assignment statements（简单赋值语句），333
Simple lists（简单链表），647
Simple phrases（简单短语），189
Simplicity（简单性），9－10
orthogonality（正交性），15
SIMULA 67, 23, 76－77, 93
SIMULA1, 76
Single inheritance（单继承），510，514－515
Single Instruction Multiple-Data (SIMD)（单指令多数据），557
Slices（片），353，371－373
Smalltalk, 91－94
evaluation of support for OOP（支持OOP的评估），518
Inheritance（继承），518
object oriented programming (support for)（支持OOP），516－518
polymorphism（多态性），517
type checking（类型检查），517
SNOBOL, 75－76
origins and characteristics of（起源及特性），76
SNOBOL4, 75, 76
Source language（源语言），38
Special variable（特殊变量），55
Special words（特殊字），14，204－205
Specification packages（说明包），475
Speedcoding（快速编码），45
SQL (Structured Query Language)（结构化查询语言），714
Stack dynamic arrays（栈动态数组），266
Stack-dynamic local variables（栈动态局部变量），395

Stack-dynamic variables（栈动态变量），216－217
Start symbols（起始符号），121
State diagrams（状态图），171
Statement-level concurrency（语句层次并发），592－599
Statements（语句）
  assignment（赋值），332－337，另见赋值语句
  assignment (Plankalkül)（赋值），43，58
  break（break语句），352
  case（case语句），79，353－354
  control（控制），12－13，344
  counter controlled loops（计数器控制的循环），357－366
  fact (Prolog)（Prolog的fact语句），694
  foreach (C#)（C#语言的foreach语句），105－107
  foreach (Perl)（Perl语言的foreach语句），84
  goal (Prolog)（Prolog语言的goal语句），690，696
  guarded commands（守卫命令），372－376
  if-then-else（if-then-else语句），219
  iterative（迭代），356－371
  logically controlled loops（逻辑控制循环），366－368
  rules (Prolog)（Prolog语言的规则），694－695
  selection（选择），148－149，346－356
  switch（switch语句），351－353
  switch (C#)（C#语言的switch语句），106，352－353
  unconditional branch（无条件分支），371－372
State of programs（程序状态），158
Static ancestors（静态祖先），226
Static arrays（静态数组），265
Static binding（静态绑定），208
Static chains（静态链），451－459
Static-depth（静态深度），454
Static length strings（静态长度字符串），256
Static links（静态链接），451
Static parents（静态父辈），226
static (reserved words)（保留字static），216
Static scope（静态作用域），225－231
  evaluation（评价），229－231
Static semantics（静态语义），134，135
Static variables（静态变量），215－216
Steelman document (DoD)（美国国防部钢铁人文件），88
Stoneman document (DoD)（美国国防部石人文件），88
Storage bindings（存储器绑定），215
Strawman document (DoD)（美国国防部稻草人文件），88
Strict（严格的），673
Strings（字符串），
  implementation（实现），257－258
  length options（长度选项），256
  operations of（操作），253－256
Strongly typed languages（强类型语言），220
Strong typing（强类型化），219－221
Stroustrup, Bjarne，96，478－479
Structural operational semantics（结构操作语义），141
Structured Query Language (SQL)（结构化查询语言），714
Structures（结构）
  of associative arrays（相关数组），280－282
  of lists (Prolog)（Prolog语言的表），702－707
  Prolog，693
Structure type equivalence（结构类型等价性），221
Subclasses（子类），509
Subgoals（子目标），696
Subprogram linkage（子程序链接），440
Subprograms（子程序）
  active（活动的），236，446
  calls（调用），385
  coroutines（协同程序），431－433
  definitions（定义），385
  design issues（设计问题），384
  displays（显示），459
  dynamic scoping（动态作用域），461－464
  function design issues（函数设计问题），429
  functions（函数），429－430
  fundamentals of（基础），384－394
  generic（通用的），394，422－429
  headers（头），385
  local referencing environments（局部引用环境），395－397
  nested（嵌套的），229，451－459
  overloaded（重载的），394，420－422
  parameter-passing methods（参数传递方法），397－418
  parameters（参数），397－418
  polymorphic（多态的），422
  procedures（过程），392－393
  recursion（递归），448－450
  semantics for calls and returns（关于调用与返回的语义），441
  stack-dynamic local variables（栈动态局部变量），395
  user-defined overloaded operators（用户定义重载操作符），430－431
Subrange types（子范围类型），261－263
Subscripts（下标），264
Substring references（子串引用），253
Subtypes（子类型），512－513
  Ada，262
Subtypes（子类型），235，499
Sun Microsystems（Sun Microsystems公司），98，101
Superclasses（超类），509

Swing package (Java)（Swing包（Java）），631
switch construct（switch结构），351－353
switch (C#)（C#的switch），186,352－353
Symbolic logic（符号逻辑），685
Symbols（符号）：另见符号一节，132
  nonterminal（非终结构），120
  start（开始符），121
  terminal（终结符），120
Synchronization（同步），560
  threads (C#)（C#的线程），591－592
Synchronous message passing（同步消息传递），521
Syntax（语法），118－134
  description of（描述），119－131
  Edinburgh，695
  language evaluation criteria（语言评估准则），14－15
  lexical analysis（语法分析），169－175
  LISP，53－54
  methods of describing（描述方法），119－131
  parsing (problem)（问题的语法分析），175－178，另见语法分析
  recursive descent parsing（递归下降语法分析），179－184
Synthesized attributes（合成属性），135
System Development Corporation（系统开发公司），59
Systems programming（系统程序设计），6－7
Systems software（系统软件），6

T

Tables（表）
  parsing（语法分析），193－194
  symbols（符号），28－29
Tagged types（标记类型），535
Tags（标记），285
  XSLT（可扩展语言转换），129
Task descriptors（任务描述符），564
Task ready queues（任务就绪队列），562
Tasks（任务），559
Tasks (termination of)（任务终止），579
Tcl，80
Terminal symbols（终结符），120
terminate（终结），579
Termination of tasks（任务终止），579
Terms (Prolog)（Prolog语言的项），692
Ternary operators（三元操作符），213
Testing (pairwise disjointness tests)（两两不相交测试），186
Theorems (proving)（定理证明），688－690
Thompson，Ken，81
Threaded class（线程类），583－585
Thread of control（控制线程），558
Threads（线程）
  C#，590－592
  Java，583－590
throws clause（throws子句），625
Human document (DoD)（美国国防部惯人文件），88
TIOBE，3
Tokens（标记），117，171－172
Tombstones（墓碑），299
Top-down parsers（自顶向下语法分析器），176－177
Top-down resolution (Prolog)（Prolog语言的自顶向下归结），697
Total correctness（总体正确性），152
Traces (parse)（语法分析流程），182－185，195
Tracing models（追踪模型），701
Trade offs（折中考虑），25－26
Transition diagrams（转换图形），171－172
Trees（树）
  fully attributed（完全属性化），136
  parse（语法分析），27，123－124，参见语法分析
Tuples（元组），104
Turbo Pascal，80
Turing Award lecture (Dijkstra)（Dijkstra的图灵奖讲座），74
Turing, David，56
Twos complement notation（两补点补符号），250
Two-way selection statements（双向选择语句），346－350
  design issues（设计问题），346
Type checking（类型检测），17，219
  and polymorphism（多态性），513－514
  dynamic（动态的），219
  reliability（可靠性），17
  Smalltalk，517
Type-checking parameters（类型检测参数），394
typedef (C & C++)（C及C++中的保留字typedef），224
Type error（类型错误），219
Type equivalence（类型等价性），221－225
  name（名字），221
  structure（结构），223
Type inference（类型推理），211，214
Types（类型）
  binding（绑定），208－211
  byte（字节，位），250
  compatibility（兼容性），219
  conversions（转换），324－327
  data. *See* Data types（数据，参见数据类型）
  derived（派生），222－223

enumerated（枚举），258－261
equivalence（等价性），221－225
explicit type conversion（显式类型转换），327
inference（推理），211，214
limited private（受限私有），477
private（私有的），476
of returned values（返回值的），429－430
strong typing（强类型化），219－221
tagged（标志的），535
variables（变量），207
widened（加宽类型），428

U

Unambiguous grammars
for expressions（用于表达式的非歧义性文法），126－127
if-then-else statements（if-then-else语句），130－131
Unary assignment operators，334－335
Unary operations（一元运算操作符），313
Unchecked exceptions（非检测异常），665
Unconditional branching（无条件分支），371－372
undef（undef语句），85
Underflow（下溢），328
UNICODE（UNICODE码），56，253
Unification（合一），689
Unions（联合，并），286－290
Ada，288－290
discriminated（判别的），288
evaluation of（评估），290
implementation（实现），290
UNIVAC（公司名），43，56－57，63
UNIVAC 1107，43
UNIVACI（软件系统名），44
UNIVAC Scientific Exchange（USE）（UNIVAC科学交流项目），57
University of Aix-Marseille（Aix Marseille大学），86
University of California at Berkeley（美国加州伯克利大学），83
University of Edinburgh（英国爱丁堡大学），56
University of Kent（Canterbury，England）（位于英国Canterbury的Kent大学），56
University of Michigan（美国密西根大学），59
University of Utah（美国犹他大学），92
UNIX（UNIX操作系统），7，15，33－34
grep command（grep命令），15
lint program（lint程序），17
unsafe modifiers（unsafe修饰符），378
U.S. Naval Electronics Group（美国海军电子组织），59
use，481
User-defined abstract data types（用户定义抽象数据类型），422－423
User-defined ordinal types（用户定义的序数类型），258－263
User defined overloaded operators（用户定义的重载操作符），322－324，430
User-located loop controlled mechanisms（用户定位的循环控制机制），368－369
USE（UNIVAC Scientific Exchange）（UNIVAC科学交流项目），57
using，500

V

Values（值）
aggregate（聚集），269
computing attribute（计算属性），138－139
l-value（左值），206
precision（精度），250－251
returned（types of）（返回类型），429－430
r-value（右值），207
types（类型），291
variables（变量），207
van Rossum，G.，104
Variables（变量），205－207
addresses（地址），206－207
aliases（别名），207
binding to attributes（属性绑定），208
bound（绑定），651
class（类），510
constrained variant（受限变体），288
declarations（声明），209－210
explicit heap dynamic（显式的堆动态），217－218
heap dynamic（堆动态的），217－219
implicit heap dynamic（隐式堆动态的），218－219
initialization（初始化），238－239
instance（实例，元组），510
lifetime of（生存期，寿命），215
local（局部），395
loop（循环），357
lost heap dynamic（丢失的堆动态），294
names（名字），205
scope of（作用域），225－233
stack dynamic（栈动态的），216－217
static（静态的），215－216
type checking（类型检测），219
types（类型），207
values（值），207
visibility of（可见性），225
Variable-size cells（可变大小单元），303－304
VAX（series of superminicomputers）（超级计算机系

别的VAX计算机），10
VDL（Vienna Definition Language）（Vienna定义语言），142
Vector processors（矢量处理器），557
Vienna Definition Language (VDL)（VDL语言），142
Virtual method table (vtable)（虚拟方法表格），547
virtual (reserved word)（保留字virtual），527
Visibility of variables（变量的可见性），225
Visual BASIC，24，69
Visual BASIC NET，24，69，105
von Neumann，John（冯·诺依曼），20
von Neumann architecture（冯·诺依曼体系结构），20－22
von Neumann bottleneck（冯·诺依曼瓶颈），30
Vtable (virtual method table)（虚拟方法表格），547

W

Wall，Larry，84，364－365，398－399
Weakest preconditions（最弱前置条件），344
Web software（Web软件），7
Weinberger，Peter，83
Well-definedness（良好定义性），20
Wexelblat，Richard，63，110
Wheeler，David，45
Whirlwind computer（Whirlwind计算机），40
Widget，630
Wildcard types（通配类型），428
Wilkes，Maurice V.，45
Wirth，Niklaus，79，80，452－453
with，461
Woodenman document (DoD)（美国国防部木人文件），88
World Wide Web Consortium (W3C)（世界万维网联盟），108
Writability（可写性），15
expressivity（表达性），16
reliability（可靠性），18
simplicity（简单性），15

X

Xerox Palo Alto Research Center（Xerox Palo Alto研究中心），92
XHTML (eXtensible HTML)（可扩展HTML语言），24
XML (eXtensible Markup Language)（可扩展标记语言），25，108
XSLT (eXtensible StyleSheet Language Transformations)（可扩展样式转换），108－109

Y

yacc (yet another compiler compiler)（yacc编译器），193
yet another compiler compiler (yacc)（另一个yacc编译器），193
yield，391

Z

Zuse，Konrad，40－42