● 软件工程系列丛书

Ada

程序设计风格与范例

[美] P·希伯特 著
童长忠 黄 江 译
高仲仪 校

北京航空航天大学出版社

Ada 程序设计风格与范例

[美] P·希伯特 等著
童长忠　黄　征　编译
高仲仪　校

北京航空航天大学出版社

内 容 简 介

现代程序设计语言的设计要考虑三个方面：语言本身、环境和设计风格．由于现代软件向大型化、复杂化方向发展，因而适应多人协作开发的程序设计风格上升到举足轻重的地位．Ada 程序设计风格是 Ada 文化的一个重要组成部分．

本书集 Ada 语言程序设计风格的思想、方法、规范和实例于一体，而不死抠任何特定的语言细节．本书由三部分组成．第一部分追溯近十年来现代程序设计语言的重要思想．第二部分系统分析用 Ada 语言编程的五个实例，第三部分给出 Ada 语言编程风格约定．

本书可作为计算机软件、计算机应用和计算机理论专业的大学生、研究生和教师的教学用书和学习参考书，也可供从事计算机工作的工程技术人员和管理人员提高 Ada 语言和其它语言的程序设计风格之用．

Ada 程序设计风格与范例

ADA CHENGXUSHEJI FENGGE YU FANLI

[美] P·希伯特 等著
童长忠 黄 征 编译
高仲仪 校

责任编辑 田 野

*

北京航空航天大学出版社出版
北京航空航天大学软件工程研究所微机输入排版、激光打印
地质出版社胶印厂印刷
新华书店总店科技所发行 各地新华书店经售

*

850×1168 1/32 印张：4.75 字数：103 千字
1990 年 5 月第一版 1990 年 5 月第一次印刷
印数：0001—4000 定价：2.60 元
ISBN 7-81012-191-X/TP·034

编译者的话

现代软件开发一直面临着软件开发费用昂贵、软件质量低下和软件交货经常延期的挑战。Ada 程序设计语言正是美国国防部为解决这些矛盾、支持软件工程化生产而研制的大型通用计算机语言。她从 1975 年 4 月稻草人计划的提出，至 1980 年 7 月 Ada 文本的正式提交，集中了几千名计算机科学家的智慧和数年的辛勤劳动，其整个过程采用了软件工程化生产的思想和方法。可以毫不夸张地说，Ada 语言是第四代计算机具有代表性的语言之一。

Ada 语言主要应用于大型复杂软件系统、实时系统和嵌入式系统，也能支持分布式应用和系统开发，并能在整个软件生存周期中有效地降低费用。由于 Ada 语言在功能与性能设计上的成功，美国国防部确定在全军通力支持 Ada 开发，并且宣布，自 1984 年 2 月起，所有军用软件一律采用 Ada 作为开发语言，否则不予承认。回想起 COBOL 语言在 60 年代受到美国国防部通力支持而蓬勃发展的光辉历史，我们可以乐观地预言，在不久的将来，Ada 将是军用软件的主导语言，将会在世界范围内获得广泛的应用。

我们认为，现代程序设计语言的设计要考虑三个方面：语言本身（即语法、词义和语用）、环境和设计风格。早期的程序设计语言强调语言语法的严谨性、语义的一致性和语用的简洁性。到了七十年代后期，人们普遍认为，一个好的程序设计语言，必须具有强大的环境支持。随着软件开发朝着工程化和工业化的方向发展，单个人的编程技巧已经变得并不十分重要，而适应多人协作开发的设计风格上升到举足轻重的地位。美国国防部在公布 Ada 语言的同时，曾以官方名义出版了三本著作，分别说明了 Ada 程序设计语言的环境、设计风格及其语言本身。《Ada 程序设计风格与范例》(原名为"Studies in Ada Style") 即是其中的一本。

根据北京航空航天大学软件工程研究所几年来从事软件工程实

践的经验，深深感到软件人员的素质是决定软件质量与软件开发效率的主要因素，而这种素质的培养十分需要有一个良好的开端。在上述动机的激励下，我们将《Ada 程序设计风格与范例》一书翻译出版，奉献给广大读者。如果本书能对 Ada 语言在我国的提倡与推广起到促进作用，如果读者能通过本书受到一些程序设计风格方面的启迪，这将使我们感到由衷的欣慰。

《Ada 程序设计风格与范例》由美国卡内基梅隆大学的彼得·希伯特等人撰写，编译后的中文本由三部分组成，第一部分追溯近十年来现代程序设计语言的重要思想，如结构化程序设计、程序验证、抽象技术和规格说明技术等，展示了现代语言（如 Ada 语言）是如何对现代软件开发中的诸多问题作出反应的。第二部分考察了使用 Ada 语言编程的五个问题，说明如何选择设计策略，使问题求解空间尽可能地接近现实境界，而不死抠任何特定的语言特征。第三部分是北京航空航天大学软件工程研究所根据几年来在软件工程实践方面的经验，对 Ada 程序的书写格式作出的约定，它是 Ada 文化的一个重要组成部分，把它收入此书，可使原著趋于完整。本书将 Ada 语言程序设计风格的思想、方法、规范和实例融为一体，而不涉及 Ada 语言的具体细节，可作为提高 Ada 程序设计风格的良好教材，同时对培养用其它语言进行编程的良好程序设计风格也具有重要参考价值。

本书的翻译工作是在北京航空航天大学软件工程研究所周伯生教授的积极倡导下进行的。翻译过程中，Ada 概念方面的译法采用了当前国内的通用译法，个别译法也有杜撰之处。由于译者水平有限，文中定有不妥之处，敬请读者批评指正。北京航空航天大学计算机系高仲仪副教授仔细审校了全书译稿，在此谨致衷心感谢。

童长忠、黄　征

1989 年 12 月

于北京航空航天大学软件工程研究所

软件工程丛书

1. Ada 软件工程　　1986 年 2 月，科学普及出版社
G.Booch 著，麦中凡、梁南元译，周伯生校
2. C 程序设计教程　　1986 年 8 月，科学普及出版社
T.Plum 著，金茂忠、张子让译，周伯生校
3. 软件工程规范选编　　1988 年 3 月，北京航空学院出版社
周伯生编
4. 算法和数据结构手册　　1988 年 5 月，人民邮电出版社
G.H.Gonnet 著，张子让、周晓东译，周伯生校
5. 航空工业部软件工程规范汇编 1988 年 6 月，航空工业部科学技术局
航空工业部软件工程化工作小组编写
6. PASCAL 程序设计基础　　1989 年 8 月，北京航空航天大学出版社
张子让、周伯生编著
7. 面向对象的程序设计——Smalltalk／V 语言及环境
1990 年 5 月，北京航空航天大学出版社
夏晓东、宋杰、刘柏译，高仲仪、金茂忠校
8. Ada 程序设计风格与范例 1990 年 5 月，北京航空航天大学出版社
童长忠、黄征编译，高仲仪校
9. C 程序设计高级教程——UNIX 系统程序库
1990 年 6 月，北京航空航天大学出版社
金茂忠主编
10. 软件工程规范基础　　待出版，北京航空航天大学出版社
周伯生著
11. 软件工程理论、实践和展望　　待出版，北京航空航天大学出版社
C.R.Vick 等编，周伯生等译

目　录

示图清单

第二版导言

自《Ada 程序设计风格与范例》第一版出版以来，Ada 语言在推向 ANSI 标准过程中已作了部分的修改。在出版《Ada 程序设计风格与范例》第二版时，其中某些章节作了修改，使之与最新的 Ada 手册[Department of Defense 82]相一致。我们对 Ada 文本的所有引用都来自该手册。

我们希望将来的 Ada 程序员能从我们的经验中获得益处，并欢迎各界同仁提出批评和建议。

彼得·希伯特
安迪·希思琼
乔纳森·罗松伯格
玛丽·肖
马克·谢尔曼

Pittsburgh, PA.
1982 年 9 月 21 日

第一版导言

现代软件的主要问题是寻求有效的技术和工具来组织和维护大型、复杂程序。在现代程序设计中，控制复杂性的主要原理是抽象，也就是可选择地强调细节。这本专著讨论 Ada 程序设计语言是怎样支持与开发这种抽象技术的方法的。

本书由两部分组成。第一部分追溯近十年来现代程序设计语言的重要思想，展示现代语言，譬如 Ada，是如何对软件开发中的当代问题作出反响的。

第二部分考察使用 Ada 语言编程的五个问题。对于每个问题，都给出一个完整的 Ada 程序。接着讨论 Ada 语言是如何影响各种设计策略的。选择这些问题时尽可能地接近实际，而不是用来说明任何特定的语言特征。

本书的许多材料以前已公开出版过。第一部分的较早版本是由玛丽·肖所著，发表在 1980 年 9 月《软件工程特刊》第 68 卷第 9 期的 IEEE 会议论文集上，题为"抽象因素对现代程序设计语言的冲击"，页号为 1119–1130，经 IEEE 允许，本次重印。原文已作了修改，以满足修订后的 Ada 语法和语义。第二部分的较早版本是由彼得·希伯特、安迪·希思琼、乔纳森·罗松伯格和马克·谢尔曼四人合著，发表在卡内基梅隆大学计算机科学系技术报告 CMU–CS–80–149 上，题目是"Ada 程序设计举例"（现在已脱销）。某些程序为适应分别编译而作了部分修改，并在程序中增加了注释和描述程序性能的说明。

第一部分

抽象因素
对现代程序设计语言的冲击

玛丽·肖

卡内基梅隆大学计算机科学系

1982 年 9 月

第 1 章

抽象因素对现代程序设计语言的冲击

1.1 现代软件存在的问题

现代软件开发存在的主要问题是软件开发、使用和维护的费用太高以及开发出的系统的质量太低。这些问题对于具有较长使用寿命的大型、复杂程序尤其严重，而现代软件的特征之一就是程序的大型化与复杂化。这种程序不论是在开发阶段，还是在软件开始使用后的维护和改进期间，一般都要牵涉到很多程序员。因而，软件的费用和质量要受到管理和软件工程实践双方面因素的影响[Brooks 75, Goldberg 73]。

软件工作者为解决软件的高费用与低质量问题进行过各种尝试，本文考察了贯穿于这段历史的其中一个论题，就是抽象技术及其有关的规格说明和验证技术在现代程序设计语言及其方法的演进过程中所起的作用。该论题重点强调工程因素，包括设计、规格说明、正确性和可靠性诸方面。

本文一开始就回顾（第 1.2 节）一下对研究现代技术具有重要作用的有关程序开发和分析的思想。其中许多思想不仅具有历史意义，也具有现实意义。回顾部分用一定篇幅综述了从当前研究中获得的某些思想，这些思想正影响着现代语言的设计与软件方法学（第 1.3 节）。1.4 节用两种不同的语言 Ada 和 Pascal 编写一个例子，说明由于上述工作所导致的程序组织的变化。尽管 1.2 和 1.3

节介绍了一定的技术细节，但 1.4 节仍用一个对所有读者都能理解的例子来说明抽象思想。本文以对当前状况及当前抽象技术的潜力（第 1.5 节）进行评价作为结束。

1.2 抽象技术的历史回顾

控制软件开发和维护总涉及到对程序与程序系统的智能复杂性的管理。这种系统不仅能创建，而且也可测试、维护和扩充。于是，各种不同的人必须能在其生命周期内的任何时候理解并修改它们。本节介绍有关管理程序复杂性的部分思想，并展示这些思想在过去十至十五年内是如何构筑程序设计语言及其方法学的。

在程序设计语言及其方法学的发展过程中，一个主要论题是研制用来处理抽象的工具。所谓抽象就是系统的简化描述或规格说明。它强调系统中的某一部分细节或特性，而忽略了其它部分。一个好的抽象应该强调对读者（即用户）至关重要的信息，而忽略至少目前不重要或转移视线的细节。

我们在程序设计系统中所说的"抽象"与其它许多领域中所说的"分析模型"密切相关。它们存在许多共同的问题：确定系统中哪些特征是重要的，应该包括哪些变量（即参数），应用何种描述格式以及怎样确认模型等等。和在许多领域中一样，我们常常要定义层次模型体系，其中较低层次的模型为较高层次模型中出现的现象提供较详细的解释。我们的模型也具有以下特性，即描述与需要显式确认的基本系统明显不同。我们把一个模型的抽象描述称做它的规格说明，而将模型体系中的下一个较低层次的模型称做它的实现。确认规格说明与实现的一致性叫做验证。我们用于软件的抽象倾向于强调软件的功能特性，即强调将获得什么结果，而忽略有关怎样得到这一结果的细节。

组织程序和语言的许多重要技术都基于抽象原理。这些技术已经逐步演进，它们不仅表现在我们对程序设计问题的理解上，而且

表现在我们把抽象用作为所描述系统的形式规格说明的能力上。例如，在六十年代，关于方法学和语言的重要研究是以函数和过程为中心的，它们根据名字和参数表来归结一程序片段。那时我们只知道怎样进行语法确认检查，规格说明技术反映在"规格说明"只不过是一"过程头部"，这一涵义一直延续到六十年代末期。到了七十年代末期，研究集中到数据结构的设计上，规格说明技术凭借了相当复杂的数理逻辑技术，程序设计语言的语义已为人们充分理解，人们能够进行程序与规格说明一致性的形式确认。

程序设计语言和方法学的研究常常是为了反映如何对付程序与程序系统的复杂性的新思想。随着语言的发展以满足这些思想，我们会根据新的经验去修正我们对问题和答案的体会。加深的体会反过来在进化周期中又会产生新的思想。本文正是探讨方法学和规格说明上的这种周期性的演进是怎样导致了当今的程序设计语言原理的。

1.2.1 早期的抽象技术

在六十年代末期之前，人们认为重要的程序设计课题由程序设计语言的语法、编译技术和对特定实现问题的解决所支配。因而我们可以看到许多有关解决词法分析、存储分配和数据表示等诸如此类的特定问题的论文。过程是很好理解的，并且提出了过程库。这些库并非在所有方面都获得了成功，这是因为（非正式的规格说明）文档常常是不充分的，或者是因为过程参数化对所关心的情况没提供支持。象堆栈和链表这样的基本数据结构恰恰刚开始为人们所理解，但它们是相当陌生的，难于把概念和具有实现分开。也许在这一领域的历史上，通用性和综合性的出现还为时尚早，但在任何场合，抽象所发挥的作用都微乎其微。

六十年代末期，抽象第一次有意识地用在程序组织技术中。较早期的语言至少支持包括整型、实型和数组等内部数据类型，有些还支持布尔型和高精度实型等。数据结构于 1968 年第一次系统地

作了论述（[Knuth 73a]第一版），程序员可定义适合于特殊问题的数据类型这一概念只是在 1967 年才第一次出现（例如[Standish 67]）。虽然有关程序设计技术的讨论可追溯到这一领域的开始，但提出程序设计是一项值得研究并要遵守某些规范的活动这一概念还是 1968 年[Naur 69]和 1969 年[Buxton 70]的“NATO 软件工程会议”的事情。

1.2.2 可扩充的语言

在六十年代末期人们就能从程序设计语言的内部标记中看到抽象的效果，在这种程序设计语言中，程序员能够在基本语言中增加新的标记和新的数据类型。研究可扩充语言的目的包括允许每个程序员扩充程序设计语言的语法，定义新的数据结构，为新旧数据结构增加新的运算符（包括中缀运算符和一般函数）以及在基本语言中增加新的控制结构。中断了有关扩充性的这一研究[Schuman 71]的部分原因是由于人们低估了定义有意义扩充的难度。问题表现在：当修改基本语言的语法时，难以保持单一的扩充与原始的定义相兼容，使得相关的信息聚集在公用的单元中；以及难以找到精确地描述扩充（并非显示扩充的代码）的技术。然而，它对七十年代的抽象数据类型和类属定义留下了深刻的影响。

1.2.3 结构化程序设计

到了七十年代早期，程序开发方面出现了一种新的方法学，这就是“逐步求精”或叫做“自顶向下程序设计”，它通过不断细化的中间阶段，使目标说明逐步过渡到最终代码[Dijkstra 72, Wirth 71]。更具体地说，它通过写出一个程序，该程序可自由假定存在任何可直接适用于正解决问题的数据结构及操作来逼近一个问题，而不管这些结构和操作是多么复杂或者难以实现。因而可以想象开始写出的程序规模较少，思路清晰，直接面向问题，并且是“明显”正确的。虽然假定的结构和操作仅仅是非形式描述，但程序员的直觉使

得他能集中于程序的总体组织，而推迟考虑假定定义的实现。在实现前面假定的任何定义时，同样又可以应用与上相同的技术，用较高层操作的实现来替换相应的请求。这样就得到了一个新的更详细的程序，可以相信它完全类似于前面的程序，只是所依赖的定义更少，或者更简单（因而更接近于可编译）。继续运用上述步骤，即增加那些与程序设计语言更有关的细节，直到该程序完全可用存在编译程序的基本语言的操作和数据类型来表达。

这种把用于解决问题的结构与实现这些结构的方式之间分开考虑的思想，提供了把复杂问题分解为较小的、相对独立的片段的一种方法。这种方法学成功的关键在于所选择的高级数据结构及操作的抽象程度。它的主要限制是最终程序没有保存程序建立期间的抽象过程，因而在程序完成后，如果要作修改，则比以任何其它方法开发的程序都要麻烦，这一限制现在暂不加评论，留待使用它时再评述。这种方法学的另一个限制是操作的非形式描述不能传播精确的信息。如果对一操作究竟应该干什么理解有误，就可能要使程序开发过程复杂化，并且过程的非形式描述不足以保证模块的独立性。有关程序形式化规格说明技术的研究有助于缓解这类问题。

几乎在这种方法学出现的同时，人们也开始关注他们怎样理解程序，程序怎样组织就能易于理解，因而易于修改。我们认识到如果能确定程序中任何位置的程序状态，这将是非常重要的。进一步说，任意跳越大量程序正文的控制转移与这一目的相违背。有助于理解程序的控制流模式是具有一个单入口点（在正文的开始），并且至少在原理上具有一个单出口点（在正文的结尾）。满足这一规则的语句例子有 **if...then...else** 语句和 **for** 及 **while** 循环。与该规则相违背的主要是 **go to** 语句。

第一次讨论这个问题是在 1968 年[Dijkstra 68]，稍后几年，人们集中在想找到一组公用的"理想"控制结构上[Dijkstra 72, Hoare 73]。虽然人们一直没有得到一致公认的这组结构，但该问题不再是一个争论点。

1.2.4 程序验证

在研究"理想"的控制结构的同时——实际上，是其动机之一，计算机科学家开始将兴趣放在找到一种方法，它能精确地表述程序所进行的计算，并且这种表述在数学上可运算。能够作出这种表述，这对于研究程序推理技术，特别是研究依赖于结果的抽象规格说明技术都是十分重要的。新技术是必需的，因为过程头部，即使有一段注释，也没有提供程序完全推理所需的足够信息，并且模糊的陈述导致了所写程序的二义性，也没有使模块充分分离。

六十年代后期第一次出现了下述概念，就是有可能作出变量值的形式化描述（一组程序变量的值称之为程序状态）以及利用程序状态严格地推导出一语句执行的结果[Floyd 67, Hoare 69]。形式化描述可用谓词运算公式表达，例如

$$y > x \cdot (x > 0 \vee z = x^{**}2).$$

程序设计语言可用一组规则描述，这些规则定义了每条语句在用程序状态描述的逻辑公式上所对应的结果。语言规则要适合程序中的断言，以得到其证明能确保程序与规格说明相一致的定理①。到了七十年代早期，验证简单程序的断言及用一种可能的方式描述语言的基本原理已经得以解决[Hoare 73, London 78]。当要实际运用它时，验证技术易于出错，形式化规格说明和非形式化规格说明一样，都受到遗漏错误的影响[Gerhart 76]。为解决这一问题，已开发出能自动地一步一步执行验证的系统[Gerhart 79]。验证要求把用逻辑断言注释的程序转换为逻辑公理，这些公理具有如下特性：当且仅当公理为真时程序正确。这一称为验证条件生成的转换过程是不难理解的，但大量的工作要留待证明这些公理时去做。

---

① 这些思想的概述参见[London 75]；这些方法的介绍请参见[Manna 74]中的第三章及[Wulf 81]中的第 5 章.

当程序设计方法学的重点移到使用数据结构作为程序组织的基础时，对应的问题导致了规格说明和验证技术的产生。最初的研究提出了何种信息在规格说明中是有用的这一问题[Parnas 72a]。以后的注意力集中在使这些规格说明更形式化和解决验证问题[Hoare 72a]。在这一基础上，着手解决抽象数据类型验证工作，这将在1.3节讲述。

1.2.5 抽象数据类型

七十年代人们认识到根据有关实现细节的知识尽可能局部化的原则把程序组织成模块的重要性。这导致了语言对数据类型[Hoare 72b]、利用与数据类同的结构化的规格说明[Guttag 78, Lampson 77, Wulf 76]和类属定义[Schuman 76]的支持。语言设施基于Simula的**class**结构[Dahl 68, Dahl 72]、有关定义模块策略的思想[Parnas 71, Parnas 72b]以及局部化对程序组织的冲击因素[Wulf 73]。相应的规格说明技术包括强类型和有关功能正确性的断言验证技术。

七十年代后期，在抽象技术方面的大多数研究活动集中在由于考虑到上述因素而引起的语言和规格说明问题上；其中多数工作可认为与抽象数据类型有关。如同结构化程序设计，抽象数据类型方法学的重点是在相关信息集合的局部化。这时的注意力集中在数据，而不是控制结构，其策略是形成由数据结构及其相关操作组成的模块。它的目标是想把模块同整型和实型等一般类型一样处理；这需要声明、中缀运算符和例程参数规格说明等支持。称之为抽象数据类型所产生的结果是有效地扩展了程序可用的类型集合——它通过规定新的变量组中的某个变量应具有的值来解释该组变量的特性，它通过给出允许用于新类型变量上的操作在这些变量值上所产生的结果来解释这些操作。

在数据类型抽象中，我们先要规定数据结构及其操作的功能特性，然后根据已有的语言结构（及其它数据类型）来实现它们，并

展示其规格说明的精确性。以后每当使用该抽象时，我们只要根据它的规格说明来处理新类型（这种技术将在 1.3 节详细讨论）。这一基本原理在最近的几个语言研制项目中得到了应用，它们包括 Ada[Department of Defense 82]、Alphard[Wulf 76]、CLU[Liskov 77]、并发 Pascal[Brinch Hansen 75]、Euclid[Lampson 77]、Gypsy [Ambler 77]、Mesa[Geschke 77]和 Modula[Wirth 77]。

用于抽象数据类型的规格说明技术是从用于简单顺序程序的谓词演化过来的。所增加的表达能力用来应付将信息封装成模块的方式，解决从实现中抽象数据类型等问题[Guttag 80a]。其中一类规格说明技术强调数据类型和称之为代数的数学结构之间的类似性[Guttag 78, Liskov 75]。另一类技术根据公用的不难理解的类型的特性来定义新定义类型的特性，以此来显式地模型化新定义的类型[Wulf 76]。

连同抽象数据类型和形式化规格说明一道，人们也在研究类属定义技术，它始源于可扩充语言，但后来的表达力及精确性远远超出其创始人的期望。这些定义不仅对于程序执行过程中可操作的变量来说是参量化的，而且对于数据类型来说也是参量化的，它将在 1.3.3 节详细讨论。它们现可用来相当具体地描述可接受的参数类型的限制，参见[Bentley 79]。

1.2.6 抽象技术和规格说明技术之间的相互影响

正如以上综述所表明，程序设计语言和方法学是随着软件设计人员和实现人员所提出的需求而逐步发展的。然而，这些需求本身也是随着过去实践中所获得的经验而演进的。原始的结构化程序设计中的抽象技术是过程或宏①；现在已经发展到抽象类型和类属定义。每当我们找到了公用的模式并且力图把它们用作模型时，就会

① 虽然过程原本是作为节省代码空间的设施，但它不久就象宏一样当作抽象工具了。

出现新的程序开发方法学；每当这些模型变得如此通用和稳定以至它们可当作标准时，语言也就演进到能支持这些方法学。有关软件抽象研究的更广泛综述请参见[Guarino 78]。正如抽象技术已经能应用于各种不同的程序组织，形式化规格说明已经变得更精确，并且在程序设计过程中起着愈来愈关键的作用。

对于有效使用的抽象技术，它的规格说明必须能表达程序员所需要的所有信息。规格说明的最初企图是用在程序设计语言的标记中，以表达能由编译程序检查出来的信息，如例程名及其参数的个数与类型。诸如对一例程所进行的计算以及它适用条件的描述等其它事实，还是采用非形式的表达方式[Yeh 80]。现在我们已经发展到能够写出例程间许多重要关系的精确描述，这包括有关它们的输入值的假定以及它们对程序状态所产生的结果，但是，抽象的许多其它特性仍只能作非形式的规定。它们包括时空开销、与专用设备的交互、非常复杂的聚集行为、硬件出错时的可靠性以及并发处理等许多方面。希望将来在规格说明技术与程序设计语言方面的研究会对这些问题作出回答，应该说这是完全合理的。

程序设计语言的历史显示出语言思想与形式化技术间的平衡；在每种方法学中，我们规定的特性与我们当前对规格说明与其实现的一致性的确认（验证）相匹配。于是，由于仅在我们能确信形式化规格说明与其实现相匹配时，我们才能应用形式化规格说明，因而对抽象技术、规格说明技术以及验证规格说明与其实现相一致的方法的研究必须齐步前进。将来，我们希望看到在程序中具有更多的用作模块化基础的设施；我们也希望看到规格说明将涉及到程序的许多方面，而不只是我们现在所考虑的功能特性这一个方面。

1.3 现代程序设计语言中的抽象设施

有了 1.2 节的历史背景，现在我们回到当前在程序设计语言界正在研究的抽象方法论和规格说明技术。其中某些思想正准备运用

到实际语言中，但其它的仍处于研究之中。

虽然在现代抽象技术背后的思想能够独立于程序设计语言来进行探讨，但这些思想在实际语言中的应用也相当重要。程序设计语言是我们表述各种非常复杂思想的基本概念工具；我们必须掌握的原理不仅包括数学中的函数关系，也包括处理时间关系的结构，例如顺序与同步。通过使某些程序结构比其它结构更易于描述，语言设计影响了我们对算法的思维方式。此外，程序设计语言既可用于人们之间的通讯，也能用于控制机间的通讯。这种作用对于具有较长使用寿命的程序尤为重要，因为程序在许多方面都是用来表达由设计者施加于结构的最实际的介质，且对于维护文档在时间上的精确性也十分重要。因此，尽管大多数程序设计语言在技术上说来都具有同等的表达力，但语言间的区别严重地影响到它们的实际使用。

1.3.1 新思想

程序设计语言的当前动向是由三个全局因素决定的，即设计的简单化、在形式化规格说明中应用精确分析技术的潜力以及控制长生命期程序的整个生存周期的费用的需要。

简单化已成为评价程序语言设计的主要准则。我们将看到在完成任务的"恰当结构"与要求语言足够小使之能够完全理解这两种需要之间存在着冲突。如果提供高度专用化的结构，单个程序将变得较小，但这是以整个系统的复杂性（和对将来的影响）为代价，这是权衡专用化与通用化的一个例子。当今的倾向是提供一个相当小的基语言，它具有以一种规范的方式定义特殊设施的设施[Shaw 80]。简单化的着重点是以现已通用的大量设计准则为基础。例如，当程序组织成可将信息局部化时，在程序部分和模块接口间共有的假定就能作出重大简化。在程序设计语言中支持抽象数据类型，使得程序员能设计出专用结构，并能以一种简单方式处理它们；其中一种方式就是提供定义设施，据此语言就能进行规范的可

预测的扩展．在应用这些设施时引入的规范性实质上能减少维护问题，它使不熟悉程序代码的程序员更容易理解有关程序在程序给定点的状态的假定，因而增加了程序员在不引入新的错误的情况下对程序作出修改的可能性．

我们对程序设计语言中所用的原理的理解是逐步加深的，现已认识到形式化技术和定量技术两者都具有灵活性，也很有用．当前规定抽象数据类型特性和验证规格说明与其实现相一致的方法将在1.3.2节讨论．目前正在进行测试方法的关键技术的研究[Howden 79]，对评价程序的定量方法的兴趣也越来越浓[Quant 79]．人们会有兴趣地注意到，在能写出语言结构的证明规则的难易程度与程序员能正确地使用这些结构并能理解用过这些结构的程序的难易程度之间似乎需要进行较大的矫正。

七十年代才真正认识到软件费用不仅仅是软件最初开发与运行的费用，而应包括程序在整个生存周期中所花的费用．对于大型的长生命期程序，软件完善和维护的费用通常超过软件设计、开发和运行的费用，常常占有较大比例．由此引起了两个问题[Dekemer 76]：第一，为了正确地修改程序，程序员必须能确定程序的哪些部分依赖于正要修改的片段。如果信息已局部化且程序结构与设计结构相吻合，则上述确定问题将变得简单．除非在脱机设计笔记和其它文档经过细致（和无误）的修改这种未必可靠的情况下，它们均不是合适的代用品．第二，大型程序很少只有一个版本．有关控制大规模程序开发的主要问题是管理问题，绝不是程序设计问题．而且，语言相关的工具能有效地缓解这些问题．现在可以得到能对程序的多个版本进行管理的工具．

1.3.2 抽象数据类型的语言支持

在过去五年里，程序设计语言与方法学研究界的主要贡献是对抽象数据类型的有关研究，当前状态直接源自1.2.5节讲述的历史根源．方法论上的问题包括对信息隐藏[Parnas 71, Parnas 72b]和数

据存取局部化[Wulf 73]的需要、数据结构的总体观点[Hoare 72b]、以 Simula 语言的 class 结构为示范的程序组织策略[Dahl 68, Dahl 72]以及类属定义的概念[Schuman 76]. 主要根源包括从实现中抽象出特性的提出[Hoare 72a]以及在类型原理上的思考，正是它们最终导致了类型具有抽象代数的共同特征这一观点[Guttag 77, Guttag 78, Liskov 75, Morris 73a].

结构化程序设计是通过不断在程序的控制结构中增加细节来开发程序的，而利用抽象数据类型的程序设计首先是将程序分解为模块，这些模块对应于最终系统的主要数据结构. 两种方法是互补的，因为结构化程序设计技术可用在类型定义模块中，反之亦然. 文献[Brown 80]中举了一个例子，说明这两种风格的相互影响.

在具有抽象设施的多数语言中，抽象数据类型的定义是由包括以下信息的程序单元组成:

- 在类型定义外部可见的信息: 类型名和允许使用该类型的所有操作（包括过程和函数）的名字及例程头部; 有些语言还包括该类型变量可具有的值和操作特性的形式化规格说明.
- 在类型定义外部不可见的信息: 利用内部数据类型或其它已定义类型写出的类型表示、可见例程的体以及仅可在模块内部调用的隐藏例程.

图 1-5 是一个例子，它定义了一个抽象数据类型的模块.

抽象数据类型的一般问题已经在好几个研究项目中提出来了，这些项目包括 Alphard[Wulf 76]、CLU[Liskov 77]、Gypsy[Ambler 77]、Russell[Demers 80]、并发 Pascal[Brinch Hansen 75]和 Modula [Wirth 77]. 尽管它们在具体实现上有所不同，但它们都具有一个共同目标，那就是提供足够的语言支持，使之能够从数据结构中抽象出抽象数据类型以及使这些抽象定义具有内部数据类型相当的地位。有兴趣的读者最好再仔细地研究一下这些项目，来获得它们之间差异的详细描述。其中多数项目对其它语言的冲击很可能只是影响而已，不可能完全为其它语言所接受. 的确，其中几个

研究项目对 Ada[Department of Defense 82]和 Euclid[Lampson 77]的影响就很明显。

具有抽象数据类型的程序设计需要程序设计语言的支持，而不是简单的有关程序组织管理上的约定。适当的语言支持需要解决大量的技术问题，它们涉及到设计与实现双方面，包括：

- 命名：作用域规则需要保证恰当的名字可见性。另外，为了保证隐藏信息保持私有，应该考虑使用保护机制[Jones 76, Morris 73b]。而且，如果依赖于当前的验证技术，必须防止程序员以多种方式命名（别名）同一数据。
- 类型检查：最好在编译时检查例程的实参，以确信该例程可接受它们，这是相当必要的。由于可以在编译过程中增加新类型，并且类型的参量化在有效类型检查规则的定义中需要微妙的决策，该问题比常规语言的类型检查更复杂。
- 规格说明标记：抽象数据类型的形式化规格说明应能传播程序员需要的所有信息。当前这还是不可能做到，下面只是描述一下在当前这方面的进展。和任何规格说明的形式化一样，也有必要开发一种方法，用它来验证规格说明与其实现的一致性。
- 分布式特性：除了所谓的例程或中缀运算符操作以外，抽象数据类型常常必须提供一些额外的定义，以支持程序设计语言中各种特定于类型的结构的解释。这些结构包括存储分配、在未知表示的数据结构元素上的操作循环及同步等。其中有些问题已得以解决，但还有很多问题尚未有答案[Liskov 77, Shaw 77, Shaw 80]。
- 分别编译：抽象数据类型对分别编译过程引入了两个新的问题。其一，类型检查应在编译单元间和单元内进行。其二，类属定义为优化（或为低效的实现）提供了有效潜力。

抽象数据类型的规格说明技术是当前许多研究项目中的课题，现已提出的技术包括非形式化但具有一定风格的严谨英文[Hening-

er 79]、把新定义的类型与已定义的类型联系在一起的模型[Wulf 76]及规定独立于其它类型的新类型的代数公理[Guttag 77]，但仍存在着很多问题尚未解决。当前的重点是在代码特性的规格说明，这些规格说明要符合非形式化的可理解的需求也很重要[Davis 79]。并且，当前的工作几乎集中在定义的功能特性上，而没有注意其它特性，例如性能和可靠性。

并非所有的语言研制过程中，都将形式化规格说明作为代码的一部分。例如，Alphard 语言结构将每个规格说明与一模块的实现联系在一起，Ada 和 Mesa 期望接口定义至少包括为支持分别编译所必需的信息。不过，所有这些工作都基于一个前提，那就是规格说明必须包含抽象数据类型用户应得到的所有信息。当已验证实现符合它的公有规格说明[Hoare 72a]时，抽象规格说明可以安全地用作权威性证据，证明更高层程序可正确地使用该实现模块。在某种意义上说，我们在"较小"的定义之上构造了一个"较大"的定义；但由于正是规格说明使程序便于理解，因而在另一种意义上说，新的定义并不比业已存在的部件大。正是这种微妙的关系，使得规格说明技术具有强大的功能。

1.3.3 类属定义

特别强大的抽象数据类型定义可允许一个抽象将另一个抽象（即数据类型）作为参数。这种类属定义提供了模型灵活性的一个方面，它是一般参数化定义所没有的。

例如，考虑为某应用定义数据类型这个问题，该应用要使用三种无序集合：整型集合、实型集合及用户定义的三维空间点类型集合。一种方法是分别为这三个定义各写一个定义，但这将要导致大量的重复正文存在，因为所有定义的规格说明及其代码都很类似。事实上，这些程序很可能只在要引用特定类型的集合元素的地方才有区别，机器码很可能只在进行集合元素操作（如用来将新值存入数据结构中的赋值运算）的地方才有所不同。这种方法的明显不足

是存在重复的代码，程序设计中存在重复性劳动以及使维护复杂化(因为必须定位错误，且必须要修改所有版本)。

另一种途径是把无序集合的特性从它们的元素特性中分离开来。这是有可能的，因为集合定义只依赖很少的特定于元素的定义——可假定只为元素类型定义了一般的赋值运算和相等比较。在这种假定下，可以只写出一个定义：

```
type UnOrderedSet(T : type) is ...
```

根据支持类属定义设施语言的语法，用上述定义可声明具有各种不同元素类型的集合：

```
var
    Counters : UnOrderedSet(integer);
    Timers   : UnOrderedSet(integer);
    Sizes    : UnOrderedSet(real);
    Places   : UnOrderedSet(PointIn3Space);
```

UnOrderedSet 的定义将提供 Insert、TestMembership 等操作，变量的声明将为所有有关元素类型例举(Instantiate)这些操作版本，并且编译程序将通过检查例程的参数确定在程序特定位置到底使用了哪个操作。

由类属定义得到的这种灵活性可以通过文献[Bentley 79]中的算法变换加以演示，该变换自动地把一类问题的任何解转换为对应的较大类问题的解。可为这种类属定义规定参数，使之具有假定的精度要求。

1.4 实现

已有好多程序设计语言提供了支持抽象数据类型所需的所有或部分设施。除了在研究项目中实现了而外，已有几种程序语言也实现了，包括 Ada[Department of Defense82]、Mesa[Geschke 77]、

Pascal[Jensen 74]以及 Simula[Dahl 68]. 在这些语言中，Pascal 语言当前拥有最多的用户，Ada 语言的开发目标是获得一种语言，它支持当代程序设计的大部分思想。由于 Pascal 和 Ada 语言在程序设计语言界发挥了较大的作用，下文将详细讨论它们。

下面我们用一个小程序来说明引入抽象技术后程序设计语言的演变。这个程序用 Fortran Ⅳ编写，以说明在 60 年代末期我们对抽象技术的理解程度，再用 Pascal 和 Ada 语言重写这个程序，以说明抽象技术是怎样应用于 Ada 语言的。

1.4.1 一个小程序例子

为了说明现代语言在程序组织和程序设计风格上产生的影响，我们来看一个小例子。本节给出该例子的 Fortran 程序，1.4.2 节和 1.4.3 节将分别给出该例子对应的 Pascal 程序和 Ada 程序。

该程序的目的是为一个小公司分部产生并打印内部电话号码表所需的数据。假定已存在一个数据库，它包含有该公司所有雇员的信息，包括他们的姓名、分部名、电话号码及薪水等。该程序必须产生一种数据结构，它是选定的分部及其电话分机内的所有雇员的一张有序表。

为 Fortran 的实现声明适当的雇员数据库及单个电话号码表，如图 1-1 所示。建立电话号码表的程序片段如图 1-2 所示。

```
C       包含 Employee 信息的向量:
C       Name 存放在 EmpNam (24 个字符) 中
C       Phone 存放在 EmpFon (整型) 中
C       Salary 存放在 EmpSal (实型) 中
C       Division 存放在 EmpDiv (4 个字符) 中
        integer EmpFon(1000), EmpDiv(1000)
        real EmpSal(1000)
        double precision EmpNam(3, 1000)

C       包含电话号码表信息的向量:
```

```
C       Name 存放在 DivNam （24 个字符） 中
C       Phone 存放在 DivFon （整型） 中
        integer DivFon(1000)
        double precision DivNam(3, 1000)

C       在程序中用到的标量的声明
        integer StafSz, DivSz, i, j
        integer WhichD
        double precision q
```

图 1-1　电话号码表程序 Fortran 版本的声明

雇员数据库可表示为一组向量，每个雇员信息对应其中一个单元，这些向量被当作单一的数据结构"并行"使用——也就是第 i 个雇员的信息存放在每个向量的第 i 个元素中．与之类似，电话号码表也建立在两个数组中，DivNam 存放姓名，Divfon 存放电话号码．

电话号码表分两步建立．首先，扫描雇员数据库，提取出其分部(EmpDiv(i))与正查找分部(WhichD)相匹配的所有雇员．每当找到一个匹配，就将该雇员的姓名及电话号码加到电话号码表中．然后，使用插入排序法①对电话号码表排序．

```
C       只要得到分部名 WhichD
        DivSz = 0;
        do 200 i = 1, StafSz
            if (EmpDiv(i) .ne. WhichD) go to 200
            DivSz = DivSz + 1;
            DivNam(1,DivSz) = EmpNam(1,i)
            DivNam(2,DivSz) = EmpNam(2,i)
            DivNam(3,DivSz) = EmpNam(3,i)
```

① 通常不选用插入排序法，不过大多数读者将会看懂这种算法，且本文的议题是程序设计语言的演进，而不是排序技术．

```
            DivFon(DivSz)    = EmpFon(i)
200         continue

C           对电话号码表排序
            if (DivSz .eq. 0) go to 210
            do 220 i = 1, DivSz
               do 230 j = i+1, DivSz-1
                  if (DivNam(1,i) .gt. DivNam(1 j)) go to 240
                  if (DivNam(1,i) .lt. DivNam(1 j)) go to 230
                  if (DivNam(2,i) .gt. DivNam(2,j)) go to 240
                  if (DivNam(2,i) .lt. DivNam(2,j)) go to 230
                  if (DivNam(3,i) .gt. DivNam(3,j)) go to 240
                  go to 230
240               do 250 k = 1, 3
                     q = DivNam(k,i)
                     DivNam(k,i) = DivNam(k,j)
250                  DivNam(k,j) = q
                  k = DivFon(i)
                  DivFon(i) = DivFon(j)
                  Divfon(j) = k
230            continue
220         continue
210         continue
```

图 1-2　电话号码表程序 Fortran 版本的代码

在这个程序中，有几个重要的地方值得注意。第一，雇员数据存放在四个数组中，这些数组间的关系仅能通过相似的命名和声明中的注释看出。第二，每个雇员姓名的字符串必须保存在八个字符的段中，且在涉及该字符串的声明和代码中没有明显的指示①。如果把 DivNam(* ,i)< DivNam(* ,j)改为小于等于测试，则程序中

① 的确，在有些 Fortran 编译版本中，浮点实现与这种类型违例相冲突；在 Fortran 标准中，字符串的处理较恰当。

的六行测试可减少到只要三行，但这样有可能使排序不可靠。第三，所有雇员数据，包括薪水，都易于存取和修改，从管理的角度来说，是不希望这样做的。

1.4.2 Pascal

Pascal[Jensen 74]是简单的代数型语言，它的设计目标主要有三条，即它要支持现代程序开发方法；语言本身要足够简单，适合于教学；即使在小型计算机，它也易于可靠地实现。多数人认为，Pascal 语言在这三方面都取得了成功。

Pascal 为支持结构化程序设计提供了大量的设施。它具有结构化程序设计的标准控制结构以及便于 Pascal 程序验证的形式化定义设施[Hoare 73]，它支持一组适合于定义抽象的数据组织结构。这些设施包括将任何常数表定义为一枚举类型的能力；能定义具有多个有名域的变体记录数据类型，且这种类型能动态分配，也可被指针引用；以及能把一数据结构命名为类型（尽管还不能把该数据类型与一组操作紧密联系在一起）。

该语言得到了广泛应用。除了作为大学本科的教学语言之外，它还用于微型计算机的实现语言[Bowles 77]，且已加以扩充，使之能够处理并发程序设计[Brinch Hansen 75]。目前正努力制定它的国际标准[ISO 79]。

Pascal 语言并不是尽善尽美的。它对大型程序只提供了有限的支持，缺少分别编译设施，只具有嵌套过程，而没有块结构。类型检查也不具有人们所要求的那么多参数传递上的控制，并且不支持相关定义的封装，因而不可能把这些定义与程序的其余部分隔离开来。其中许多不足已在其语言扩充、语言派生及标准制定中得到纠正。

```
type
   String = packed array [1..24] of char;
   ShortString = packed array [1..8] of char;
```

```
    EmpRec = record
        Name     : String;
        Phone    : integer;
        Salary   : real;
        Division : ShortString;
        end;
    PhoneRec = record Name : String; Phone : integer; end;

var
    Staff : array [1..1000] of EmpRec;
    Phones : array [1..1000] of PhoneRec;
    StaffSize, DivSize, i, j : integer;
    WhichDiv : ShortString;
    q: PhoneRec;
```

图 1-3　电话号码表程序 Pascal 版本的声明

现在我们回到产生电话号码表程序，并以此来说明 Pascal 语言的某些特点。图 1-3 给出了恰当的数据结构，包括类型定义和数据声明，图 1-4 是建立电话号码表的程序片段。

声明由四个类型定义开始，它们都不是 Pascal 语言预定义的，其中两个(String 和 ShortString)使用较为频繁，另外两个(EmpRec 与 PhoneRec)则是专为这个问题设计的。

将 String 和 ShortString 定义为类型，使得已命名变量可当作单一单元来处理；操作可以在整个串变量上执行，而不是在单个字符组上。这种抽象使问题得以简化，但更为重要的是，它使得程序员能够把精力集中于用串作名字的算法上，而不需记住名字的单个片段。图 1-2 与图 1-4 中代码复杂性的差异似乎并不大，但是当它由多个具有不同表示的单个复合结构组成时，两者之间的差异就会表现得较为突出。

```
{只要得到分部名 WhichDiv}
DivSize := 0;
```

```
for i := 1 to StaffSize do
    if Staff[i].Division = WhichDiv then
        begin
        DivSize := DivSize + 1;
        Phones[DivSize].Name  := Staff[i].Name;
        Phones[DivSize].Phone := Staff[i].Phone;
        end;

{对电话号码表排序}
for i := 1 to DivSize-1 do
    for j := i+1 to DivSize do
        if Phones[i].Name > Phones[j].Name then
            begin
            q := Phones[i];
            Phones[i] := Phones[j];
            Phones[j] := q;
            end;
```

图 1-4　电话号码表程序 Pascal 版本的代码

类型 EmpRec 和 PhoneRec 的定义是从特定的数据项抽象出来的概念，它的含意是"雇员信息记录"和"电话号码表信息记录"。因而雇员数据库和电话号码表两者都能表示为向量，其元素为相应类型的记录。

Staff 和 Phones 的声明具有如下效果，它指示所有分量都与同一信息结构有关。另外，该定义被组织成一组记录，其中每个记录对应一个雇员——因此该数据结构的主要组织由雇员信息所决定。另一方面，Fortran 程序的数据组织是由对应于其域的数组所支配，而雇员仅处于次要地位。

与 Fortran 程序一样，Pascal 版本的电话号码表也分两步来建立（如图 1-4 所示）。值得注意的是由于 Pascal 语言能把串和记录当作单一单元来操作，所以简化了对名字的操作及排序的交换步骤。两个程序间的另一个值得注意的不同是有关条件语句的使用。

Pascal 程序中，使用 **if...then** 语句，强调的是使 **if** 语句的体被执行的条件。而 Fortran 语言的 **if** 语句带有 **go to** 语句，描述的是其代码将不执行的条件，让程序的读者去计算对应于实际动作的条件。

还值得提起的是，如果没有雇员在分部 WhichDiv 工作（即 DivSize 为 0），则 Pascal 程序将不执行排序循环体。而对应 Fortran 的循环如果其前没有显式的空表测试，在这种情况下循环体将要执行一次。本题中，在空表上执行一次循环并没有什么危害；但在通常情况下，必须保证 Fortran 循环的上界不应小于下界。

1.4.3 Ada

Ada 语言在美国国防部的领导下，目前正处于开发阶段，其目的是要为了减少嵌入式计算机系统的软件费用。该项目包括语言本身和程序支持环境两个部分。研制 Ada 语言的特定目标包括有效地减少在国防部内必须学会、支持和维护的程序设计语言数目。该语言的设计强调提高程序可靠性，降低维护费用，支持现代程序设计方法，以及改善编译程序和目标程序的效率[Department of Defense 82, Ichbiah 79]。

Ada 语言在一组需求的约束下，通过招标设计来开发的[Department of Defense 78]。该语言的修订于 1980 年夏季完成，语言参考手册初稿于 1980 年 9 月出版。1981 年和 1982 年两次对手册进行了修订，推荐的 ANSI 标准是 1982 年 7 月发行的。程序设计环境的开发持续其后好几年[Department of Defense 80]。由于该语言的编译程序在市场上见到的还不多，所以现在要评价它满足其目标的好坏程度还为时尚早。不过，还是可以说说该语言的各种特征是如何解决本文关心的抽象问题的。

虽然 Ada 语言的原型产生于 Pascal，但它对 Pascal 语言的语法进行了许多扩充修改，对 Pascal 语言的语义也进行了大量扩充，所以它与 Pascal 差异很大。主要扩充包括大型程序组织和分

别编译所需的模块结构与接口规格说明、支持抽象数据类型的封装设施与类属定义、支持并发处理以及控制与目标机结构有关的低级实现问题。

```
package Employee is
   type PrivStuff is limited private;
   subtype ShortString is String(1..8);
   type EmpRec is
      record
         Name     : string(1..24);
         Phone    : integer;
         PrivPart : Privstuff;
      end record;
   procedure SetSalary(Who: in out EmpRec; Sal: float);
   function  GetSalary(Who: EmpRec) return float;
   procedure SetDiv(Who: in out EmpRec; Div: ShortString);
   function  GetDiv(Who: EmpRec) return ShortString;
private
    type PrivStuff is
      record
         Salary   : float;
         Division : ShortString;
      end record;
end Employee;
```

图 1-5　雇员记录的 Ada 程序包定义

Ada 语言主要有三个抽象工具。**Package** 用于封装一组相关定义，并使它们与程序的其它部分隔离开来；**type** 确定一变量（或数据结构）可取的值及对它可能进行的操作；**generic** 定义使得许多相似的抽象可以由一个模板生成，正如 1.3.3 节所讲述的。

Ada 语言贯穿了许多这些思想，这可通过 1.4.1 节的例子来说明。Pascal 程序（图 1-3 和图 1-4）的数据组织几乎可直接搬到 Ada 程序中，得到的 Ada 程序也相当合理。不过，Ada 提供了附

加的设施，它们完全适用于本问题。回顾一下，在 Fortran 程序和 Pascal 程序中，程序员既能存取姓名、电话号码和分部名，也能存取私有信息，如此处的薪水。Ada 程序提供了有选择的存取，我们将通过扩充前面的例子来说明这一点。

现在我们用三个部件来组织程序：每个雇员记录的定义（图 1—5）、程序所需数据的声明（图 1—6）和建立电话号码表的代码（图 1—7）。

雇员信息程序包说明了 Ada 抽象设施工具箱中的一个主要部件，其规格说明如图 1—5 所示。该定义建立了 EmpRec 数据类型，它带有一小组特许操作。这儿仅给出了程序包的规格说明。Ada 不要求程序包体和规格说明放在一起（虽然其体在程序能运行之前必须定义）；而且，使程序员只需了解规格说明，而不需要知道程序包体。规格说明本身可分为可见部分（从 **package** 至 **private** 部分）和私有部分（从 **private** 至 **end**）。私有部分仅用来提供分别编译的信息。

```
declare
   use Employee;

   type PhoneRec is
      record
         Name     : string(1..24);
         Phone    : integer;
      end record

   Staff : array (1..1000) of EmpRec,
   Phones : array (1..1000) of PhoneRec;
   StaffSize, DivSize : integer range 1..1000;
   WhichDiv: ShortString;
   q: PhoneRec;
```

图 1-6　电话号码表程序 Ada 版本的声F

假定 EmpRec 的使用策略如下：域 Name 和 Phone 对任何人都可存取，允许任何人读取 Division 域，但不能往里写，只有授权程序才能存取 Salary 域或修改 Division 域。Ada 语言中有两个特征支持这种策略。第一个特征是作用域规则，即除了在规格说明的可见部分列出的名字以外，程序包外的任何程序部分不能存取程序包内的任何名字。对于程序包 Employee 这种特殊情况，意味着记录 EmpRec 的域 Salary 与 Division 在程序包外不能直接读写。因而，通过验证从程序包输出的例程的正确性，就能够控制数据的完整性。于是，例程 SetSalary、GetSalary、SetDiv 和 GetDiv 正如它们的名字所暗示的那样，执行读写操作；它们也可以记住是谁在何时对其进行了修改。第二，Ada 语言提供了控制每个例程及变量名的可见性的方式。

虽然域名 PrivPart 是和 Name 及 Phone 一同从程序包 Employee 输出的，但这样做并没有什么危险。定义了一个附加类型，来保护有关薪水及分部的信息；声明

```
type PrivStuff is limited private;
```

暗示不仅该数据结构的内容和结构对用户隐蔽(**private**)，而且除了能调用从程序包输出的例程外，禁止在类型 PrivStuff 的数据上进行其它操作。对 **limited private** 类型，甚至连赋值和相等比较也不允许。自然，程序包 Employee 的体内代码可操作这些隐蔽域；封装的目的是为了保证只有程序包里的代码能操作隐蔽类型的数据。

```
-- 只要得到分部名 WhichDiv 数据
DivSize := 0;
for i in 1..StaffSize loop
   if GetDiv(Staff(i)) = WhichDiv then
      DivSize := DivSize + 1;
      Phones(DivSize) := (Staff(i).Name, Staff(i).Phone);
   end if;
```

```
end loop;

-- 对电话号码表排序
for i in 1..DivSize-1 loop
    for j in i+1..DivSize loop
        if Phones(i).Name > Phones(j).Name then
            q := Phones(i);
            Phones(i) := Phones(j);
            Phones(j) := q;
        end if;
    end loop;
end loop;
```

图 1-7　电话号码表程序 Ada 版本的代码

强制数据结构只能在已知的例程集合上进行操作，这是支持抽象数据类型的重心。这不仅对这儿给出的例子有意义，而且对表示可随时改变以及在一定要保持域间内部一致性（如检查和）的情况下也是有意义的。支持安全计算并不是 Ada 的目的，但只要把额外封装层与在子程序中执行的某些控制结合起来，就能达到这一目的。即使没有安全性保证，有关雇员数据如何处理信息的封装也为程序的开发与维护提供了有效的结构。

图 1-6 的声明很象 Pascal 程序的声明，用 Employee 程序包取代了简单的记录类型，两种语言间只有微小的差异。子句

```
use Employee;
```

声明 Employee 程序包的所有可见的名字在当前块中可得到。

在 Ada 程序本身的代码（图 1-7）中，Ada 语言的可见性规则允许使用记录 EmpRec 的非 **private** 域名和函数 GetDiv。Ada 提供了一种方式，来创建一个完整的记录值以及只用一个赋值语句对其赋值，因此赋值

```
Phones(DivSize) := (Staff(i).Name, Staff(i).Phone);
```

一次设置了 PhoneRec 的两个域。除了这一点和一些微小的语法差别外，该程序片段非常类似图 1—4 的 Pascal 程序片段。

1.5 现状与潜力

很清楚，在软件工程的发展过程中，基于抽象原理的方法学和分析技术已经发挥了很大的作用，而且会继续发挥更大的作用。本节讲述当前我们的程序设计习惯是如何变化以反映这些思想。我们也注意到当前技术的局限性，并考虑到将来如何解决它们，最后我们提出了一些有关抽象技术方面的进一步读物。

1.5.1 新思想是如何影响程序设计的

随着抽象数据类型的有关技术的出现，它们已对程序的总体结构和编写短代码段的风格都产生了巨大的影响。

新技术将对我们用在程序系统的高层结构中的技术产生最深刻的影响，因而也会影响到设计和开发项目的管理。对变量、例程和类型名分配施加的控制使得语言具有模块化特征，由此形成了把程序分解为模块这一策略。而且，由于具有精确（且可行的）的模块接口规格说明，这将对软件项目的管理产生巨大的影响[Ych 80]。例如，大型航空系统的需求文档已经转变为精确的（可能是非形式化的）规格说明[Heninger 79]。项目组织也将受到日益增多的管理多版本、多模块的支持工具的影响[Miller79]。

模块内代码的组织和风格也将受到影响。1.4 节说明了在具有日益增长的强大的抽象技术的语言中，是如何解决对模块内部控制的数据变化的处理的。

抽象数据类型方法学背后的思想仍未全部加以确认。利用各种方法学——如基于数据类型的设计但非形式化规格说明，或者反之，即非模块化的规格说明及确认——的项目已取得了成功，但在大型项目上还未得到完整的证明[Shaw 78]。虽然没有做过完整的

确认实验，能产生最初的测试和正式鼓励。已经用无封装设施的语言中的数据类型结构编写出一个有价值的大型程序，并且该程序已基本通过确认[Gerhart 79]。通过代数公理规定的抽象数据类型作为设计工具也已证明是行之有效的[Guttag 80b]。

1.5.2 当前抽象技术的局限性

在使用抽象数据类型的过程中，也暴露出这项技术的某些局限性。其中某些问题不可完全归咎于数据类型，要不然必要的机制利用现有的规格说明技术也能简单地表达出来。另一些问题需要有一组明显类似的定义，但这些定义不能利用系统设置或调用数据类型定义甚至类属定义来表达。

好多众所周知的结构性较好的程序结构不能完全适合于用抽象数据类型模式，这包括诸如 Unix 核心的过滤程序和外壳(Shell)及规格说明受命令语法支配的交互式程序等等。这些结构无疑是有用的，且有可能与抽象数据类型一样很好理解，完全有足够的理由相信能开发出类似的精确的形式化模型。某些类似的观点已在软件的高层设计系统中提出来过[Goodenough 80, Peters 80]。

尽管对定义其参数可为类属（即在语言中不能操作的类型）的例程或模块的设施在过去五年中一直在研究，但对类属定义的类属性很少有过探讨。某些问题正是由于缺少一种设施来精确规定类属定义对其类属参数的依赖性。一个复杂的类属的特殊例子已经编写出来并已得到验证[Bentley 79]，它给出了能适用于各种不同问题的算法转换。

上述语言研究与其它研究项目[Gerhart 79, Guttag 78, Guttag 80b, Hoare 72a, Liskov 75, Parnas 72b]都相当明确地指出了功能规格说明问题。即它们提供了一些形式化的表示方法，诸如输入／输出谓词、抽象模式和代数公理，后者用来建立关于运算符在程序值上产生作用的断言。在多数情况下，不能由系统的规格说明归纳出形式断言，这时为了增加对程序的确信度，我们要藉助于测试

[Goodenough 80]。此外，在其它情况下，程序员也要关心除功能正确性以外的其它特性。这些特性包括时空需求、内存访问模式、可靠性、同步及进程独立性，它们并未被数据类型研究所强调。注重这些特性的规格说明方法学必须具有两大重要特征。其一，程序员必须有可能建立并可验证这些特性的断言，而不只是简单地分析程序正文，以求出精确值或完整的规格说明。这与我们的功能规格说明方法相似——我们力图不要正式得出由程序定义的数学函数，而是要规定某些重要且一定要保存的计算特性。其二，在增加新的特性时，要避免增加新的概念框架。这意味着处理新特性机制应与已用在功能正确性的机制兼容。

一些在功能特性以外的形式化规格说明和验证的工作也已开始。其中大多数研究是针对特定的特性，而不是能适用于各种特性的技术，不过其结果是有意义的。在现实的实时系统中，强调对各种需求的需要已在"可靠软件规格说明"国际会议[SRS 79]上受到强烈的反响，尤其在 Heninger 的论文[Heninger 79]中更为突出。其它工作包括安全特性[Feiertag 79, Millen 76, Walker 80]、可靠性[Wensley 78]、性能[Ramshaw 79, Shaw 79]和通讯协议[Good 77]等的规格说明。

1.5.3 进一步读物

为使未加详细讨论的论题的有关资料易于查找，本论文在行文中已列出了许多资料，本节的目的是强调一些书籍及论文，它们对了解一般及背景知识帮助很大。

软件开发，包括管理和实现的一般问题在 Brook 的一本很畅销的书[Brook 75]中讲述过。结构化程序设计及数据结构的一般原理是表达抽象数据类型问题的基础，文献[Dahl 72, Dijkstra 72, Hoare 72b]在技术上详细地讨论了它们。"可靠软件规格说明"国际会议论文集中有好几篇论文，研究了需求的具体描述和抽象的数学规格说明。

更具专业性的（且技术性更深的）读物包括 Parnas 在信息隐蔽方面的突破性论文[Parnas 72b]、Guttag 与 Horning 对用代数公理作为设计工具的讨论[Guttag 80b]、London 对确认技术的综述[London 75]以及在代数公理[Guttag 77]和抽象模型[Wulf 76]等规格说明技术方面的论文。

第二部分

Ada 程序设计举例

彼得·希伯特
安迪·希思琼
乔纳森·罗松伯格
马克·谢尔曼

卡内基梅隆大学计算机科学系
1982 年 9 月

第 1 章

程序举例导言

Ada[Department of Defense 82]作为一种语言，它具有许多现代特征，它们是以前广泛使用的语言所没有的。这些特征包括：数据抽象机制（程序包、私有类型、派生类型、重载和用户可重定义运算符等）、显式并行性与同步（任务、入口和接收语句）、丰富的分别编译设施和强大的强类型机制。使用这些设施使得程序可读性好、效率较高和便于维护。不过，以前在使用其它语言时所获得的经验、技术和模式不能直接搬到 Ada 程序的设计上。经验表明，为了有效地用 Ada 编写程序，必须花相当一部分精力重新学习语言。

本报告既不是 Ada 语言的入门，也不是程序设计方面的引论。我们假定该部分的读者已对修订版 Ada 有相当水平的了解。此外，要求读者了解数据结构、结构化程序设计和并发程序设计方面的知识，它们是理解本部分内容的前导知识。

为了选取本报告包括的例子，我们考虑以下几条实质性准则：

- 例子应具现实性。入选例子包含的特征应高于学术价值①。
- 例子必须是自封闭的，能通过Ada编译的完整例子。不采用程序的某部分和节录。
- 例子必须相当小，使读者不需花费很大精力就能理解它。

我们没有选取那些力图覆盖任何预定数目的语言特征的例子。

---

① 例如，象求素数[Knuth 69]的 Eratosthenes 筛选法程序，就是因为缺少学术以外的实用性，而未选入。

我们的研究目的是推进 Ada 程序设计的良好方法，而非探讨语言的设计。因此，我们确定让程序本身去展示语言的特征。

本报告包括五个例子。第一个例子是类属程序包，它给出了队列（先进先出表）的两种抽象。该例子展示了力图实现一个方便的、高效率的和可移植的库程序包将涉及的一些方面。两个队列类型都要在本报告后面的例子中用到。

一个简单的有向图程序包将在第三章给出。该类属库程序包的主要目的是提供一个迭代算子设施。第三章并讨论了迭代算子的使用和可选的图遍历方法。同时还讨论了某些有意义的但不能同时兼顾的问题，包括与使用方便和可读性好相对应的信息隐蔽。

下一个例子高度依赖于机器。它是 PDP-11® ①控制台打字机的驱动程序，将说明表示法规格说明、中断处理和与机器有关的程序设计。

第四个例子将在第五章介绍，它是一个提供了串类型表的创建和搜索机制的程序包，展示了类属程序包及参数的有意义的用法，同时也描述了在提供"保护"机制时遇到的问题。

第六章给出了最后一个例子，它给出了一个实现松弛方法的过程，该方法确定一矩形板上的温度分布。该算法由用户指定数目的任务实现。它在并发与同步程序设计中是个有意义的问题。

所有这些例子都已通过修订版的 Ada 语义分析器[Sherman 80]（由 Intermetrics 公司提供给我们）和 NYU 解释程序[Dewar 80]的检测，以保证（编译时的）语义正确性。

在所有这些例子中，我们都列出了与 Ada 语言参考手册[Department of Defense 82]相关章节的引用，这些引用形如[§ <节号>]。

---

① PDP 是数字设备公司的注册商标。

第 2 章

队列的实现

2.1 描述

在程序中，一个最常用的数据结构是队列，它常用于进程元素间的缓冲区。

下面的类属程序包提供了两种队列：一个有穷队列用于顺序程序，另一个有穷队列用于多任务程序。在例举程序包时，需要有一个类型参数，它指定排队元素的类型。在例举之后，利用类型 Queue 和 Blocking_queue 就可声明任意数量的队列。

除了程序包的规格说明外，在使用程序包时，还应考虑以下提示：

- 对任何队列变量Q，Init_Queue(Q)必须仅调用一次，且该调用须放在使用队列 Q 的任何子程序 Append、Remove、Is_Full、Is_Empty 或 Destroy_Queue 之前。
- 当不再需要队列变量Q时，应调用Destroy_Queue(Q)。
- 队列类型的判别式MaxQueuedElts表示最小性能规格说明。Queue_Package 的所有实现应保证队列至少能保存 MaxQueuedElts 个项。

两种队列类型的 Append 与 Remove 操作具有不同的语义。类型 Queue 的语义为：

- 如果指定的队列为空，这时若调用Remove，则引发异常 Empty_Queue。

· 如果指定的队列已满，这时若调用Append，则将冻结调用 Full__Queue。

类型 Blocking__Queue 的语义是：

· 如果指定的队列为空，这时若调用Remove，则将冻结调用任务，直到作了相应的 Append 调用。
· 如果指定的队列已满，这时若调用Append，则将冻结调用任务，直到作了相应的 Remove 调用。
· 为类型Blocking__Queue提供的所有操作是不可分割的。

2.2 实现

在静态队列中所用的技术只是简单的循环数组，它在别处[Knuth 73a]已彻底分析过。

冻结队列中所用的技术基本相同，只是它要在每个操作外加一条 accept 语句，以得到任务间相互的排斥与冻结。

2.3 程序正文

```
generic

   type EltType is private;

package Queue__Package is

   type Queue(MaxQueuedElts : Positive) is limited private;

   procedure Append(Q : in out Queue; E : in EltType);
   procedure Remove(Q : in out Queue; E : out EltType);
   function Is__Empty(Q : in Queue) return Boolean;
   function Is__Full(Q : in Queue) return Boolean;
   procedure Init__Queue(Q : in out Queue);
```

```
   procedure Destroy__Queue(Q : in out Queue);

   Full__Queue, Empty__Queue : exception;

   type Blocking__Queue(MaxQueuedElts : Positive) is limited private;

   procedure Append(Q : in out Blocking__Queue; E : in EltType);
   procedure Remove(Q : in out Blocking__Queue; E : out EltType);
   function Is__Empty(Q : in Blocking__Queue) return Boolean;
   function Is__Full(Q : in Blocking__Queue) return Boolean;
   procedure Init__Queue(Q : in out Blocking__Queue);
   procedure Destroy__Queue(Q : in out Blocking__Queue);

   pragma Inline(Is__Empty, Is__Full, Init__Queue, Destroy__Queue);

private

   type ElementArray is array (Positive range <>) of EltType;

   type Queue(MaxQueuedElts : positive) is
      record
         FirstElt, LastElt : Positive := 1;
         CurSize : Natural := 0;
         Elements: ElementArray(1..MaxQueuedElts);
      end record;

   task type Blocking__Queue__Task is
      entry Pass__Discriminants(Queue__Size : in Positive);
      entry Put__Element(E : in EltType);
      entry Get__Element(E : out EltType);
      entry Check__Full(B : out Boolean);
      entry Check__Empty(B : out Boolean);
      entry ShutDown;
   end Blocking__Queue__Task;

   type Blocking__Queue(MaxQueuedElts : Positive) is
      record
         Monitor : Blocking__Queue__Task;
```

```
    end record;

end Queue_Package;
```

```
package body Queue_Package is

    procedure Append(Q : in out Queue; E : in EltType) is
    begin
        if Q.CurSize = Q.MaxQueuedElts then
            raise Full_Queue;
        else
            Q.CurSize := Q.CurSize + 1;
            Q.LastElt := (Q.LastElt rem Q.MaxQueuedElts) + 1;
            Q.Elements(Q.LastElt) := E;
        end if;
    end Append;

    procedure Remove(Q : in out Queue; E : out EltType) is
    begin
        if Q.CurSize = 0 then
            raise Empty_Queue;
        else
            Q.CurSize := Q.CurSize - 1;
            Q.FirstElt := (Q.FirstElt rem Q.MaxQueuedElts) + 1;
            E := Q.Elements(Q.FirstElt);
        end if;
    end Remove;

    function Is_Full(Q : in Queue) return Boolean is
    begin
        return Q.CurSize = Q.MaxQueuedElts;
    end Is_Full;

    function Is_Empty(Q : in Queue) return Boolean is
    begin
        return Q.CurSize = 0;
    end Is_Empty;
```

```
procedure Init_Queue(Q : in out Queue) is
begin
   null;
end Init_Queue;

procedure Destroy_Queue(Q : in out Queue) is
begin
   null;
end Destroy_Queue;

task body Blocking_Queue_Task is

   MaxSize: Positive;

begin

   select
      accept Pass_Discriminants(Queue_Size : in Positive) do
         MaxSize := Queue_Size;
      end Pass_Discriminants;
   or
      terminate;
   end select;

   declare
      Queued_Elements: Queue(MaxSize);
   begin
      Init_Queue(Queued_Elements);

   Monitor_Operations:
      loop
         select
            when not Is_Full(Queued_Elements) = >
               accept Put_Element(E : in EltType) do
                  Append(Queued_Elements, E);
               end Put_Element;
         or
            when not Is_Empty(Queued_Elements) = >
```

```
                accept Get_Element(E : out EltType) do
                    Remove(Queued_Elements, E);
                end Get_Element;
            or
                accept Check_Full(B : out Boolean) do
                    B := Is_Full(Queued_Elements);
                end Check_Full;
            or
                accept Check_Empty(B : out Boolean) do
                    B := Is_Empty(Queued_Elements);
                end Check_Empty;
            or
                accept ShutDown;
                exit Monitor_Operations;
            or
                terminate;
            end select;
        end loop Monitor_Operations;

        Destroy_Queue(Queued_Elements);
    end;

end Blocking_Queue_Task;

procedure Append(Q : in out Blocking_Queue; E : in EltType) is
begin
    Q.Monitor.Put_Element(E);
end Append;

procedure Remove(Q : in out Blocking_Queue; E : out EltType) is
begin
    Q.Monitor.Get_Element(E);
end Remove;

function Is_Full(Q : in Blocking_Queue) return Boolean is
    Temp : Boolean;
begin
    Q.Monitor.Check_Full(Temp);
```

```
        return Temp;
    end Is_Full;

    function Is_Empty(Q : in Blocking_Queue) return Boolean is
        Temp: Boolean;
    begin
        Q.Monitor.Check_Empty(Temp);
        return Temp;
    end Is_Empty;

    procedure Init_Queue(Q : in out Blocking_Queue) is
    begin
        Q.Monitor.Pass_Discriminants(Q.MaxQueuedEits);
    end Init_Queue;

    procedure Destroy_Queue(Q : in out Blocking_Queue) is
    begin
        Q.Monitor.ShutDown;
    end Destroy_Queue;

end Queue_Package;
```

2.4 讨论

类型 Queue 的实现如图 2-1 和图 2-2 所示，该队列程序包已用类型 Character 例举。图 2-1 展示了一个含有三个元素的队列的抽象队列表示，图 2-2 是用已声明的记录类型表示的等价形式。

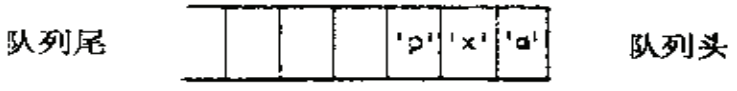


图 2-1 队列实现的抽象表示

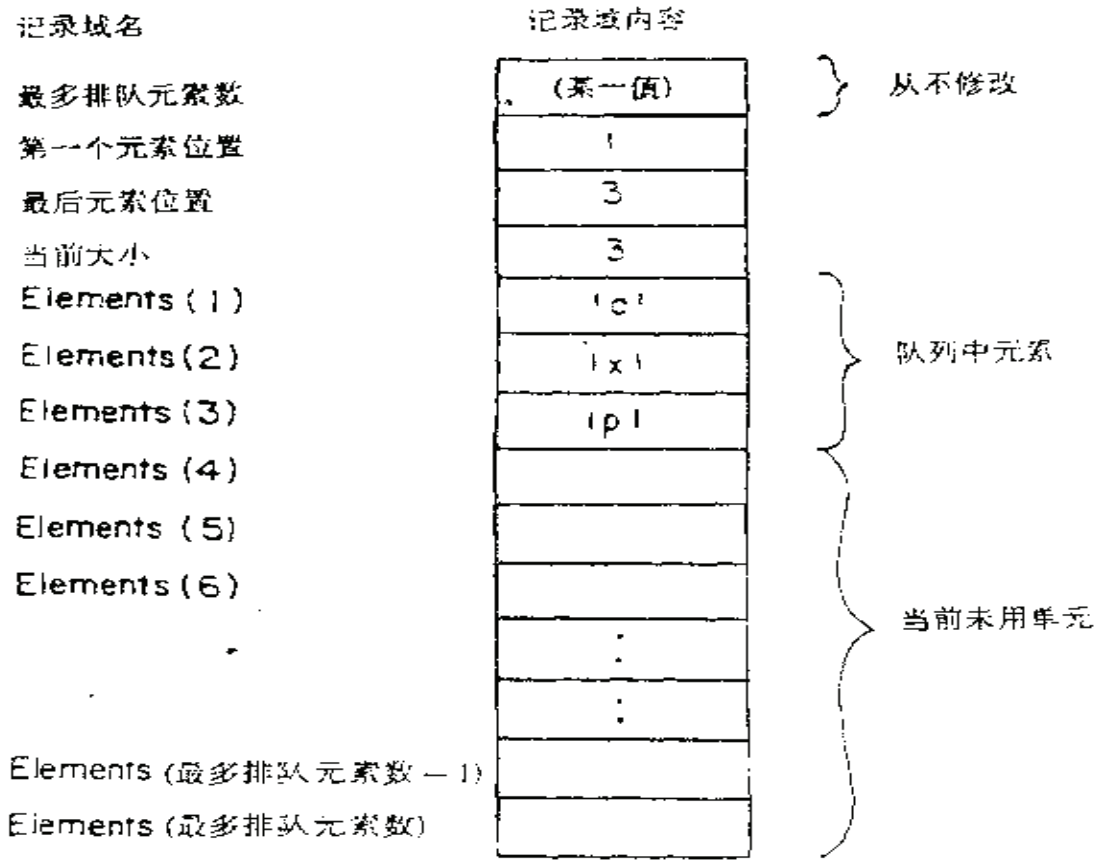


图 2-2　队列实现的实现表示

2.4.1　受限私有类型的使用

通过受限私有类型 Queue 和 Blocking_Queue，提供了声明队列的能力[§ 7.4.1,7.4.2, 7.4.4]。需要私有规格说明应该是显而易见的。程序包能保证过程 Remove 和 Append 正确操作的唯一方法是禁止程序包的用户访问其内部表示。为允许队列表示可修改，程序包也必须保证没有那一部分用户程序依赖于当前的实现。

要求类型受限更是不言自明。假设允许在类型 Queue 的两变量间进行赋值，于是在把一个队列变量赋给另一个队列变量后，我们需要知道两变量是否表示具有相同的排队元素和顺序的两种不同

队列，或者这种赋值是否意味着两个变量代表同一队列。在目前的实现中，赋值语句支持前种语义。如果在使用指针和动态存储分配的特殊实现中，则很可能更倾向于支持后一语义。这就是说，赋值语句具有两种不同的语义。相等运算符甚至更模糊。例如，考虑下述情况，假定在上面的特殊实现中，两个不同的队列具有相同的元素，但可能放在数组的不同位置上。在这些条件下，预定义的相等运算符将返回不期望的 Falsc 值。为消除用户程序依从于以上能随时改变的某一解释，可见类型应设计成受限私有类型。

2.4.2 初始化与终止化

在本程序包的描述中，我们要确保所有队列必须用 Init__Queue 子程序初始化。看一看对应程序包的体，就会发现，对于非冻结队列，子程序 Init__Queue 并没有干什么。类型 Queue 是一记录类型，在类型 Queue 的对象中写入该记录分量的缺省表达式[§ 3.7]，来实现该对象的初始化。因此，子程序 Init__Queue 似乎是不必要的。

假定类型 Queue 的表示从数组变为指向双向链表的指针，队列头是一哑元素①。这种表示在使用前需要进行一些预处理，以初始化数据结构。完成这种初始化，可考虑以下两种方法。一种方法是编出一过程 Init__Queue，在需用队列程序包前必须显式调用它。另一种方法是把队列指针放在记录里，利用记录分量的缺省表达式完成初始化。该缺省表达式通过调用一函数得到，该函数分配且初始化队列头哑元素，并返回指向它的指针。

```
-- 程序包规格说明的私有部分
type QueueElem;
type QueuePtr is access QueueElem;
function InitDummy(MaxQueueSize: Positive) return QueuePtr;
```

① 该技术在 Grogono 的一本专著[Grogono 78]中论述过。

```
.ype  Queue(MaxQueuedElts : Positive) is
     record
         TheQueue : QueuePtr := InitDummy(MaxQueuedElts);
    end record;

type  QueueElem is
    record
        V : EltType;
        Front, Back : QueuePtr;
    end record;

...

--   程序包体
function  InitDummy(MaxQueueSize: Positive) return  QueuePtr is
    --   MaxQueueSize 在这里并不是非要不可,
    --   若想预分配所有存储则可能要用它.
    QH : QueuePtr;
begin
    QH := new QueueElem;
    QH.Front := QH;
    QH.Back := QH;
    return  QH;
end  InitDummy;
```

使用记录分量的缺省表达式完成初始化的一个主要问题是，不能处理受限类型的分量[§ 3.7](由于任务类型是受限类型，因此，在上面的队列程序包中，Blocking__Queue__ Task 是受限类型)。解决这一问题的一种途径是利用指针和动态存储，记录分量的私有类型 LT 用指向私有类型 LT 的指针类型 PtrLT 代替，该指针分量的缺省表达式通过调用一函数得到，该函数（通过 new）分配一 LT 类型的对象，进行各种初始化，并返回一指向新分配对象的指针。若记录类型中具有判别式，它也应传给初始化函数，作为将要分配的 LT 对象的参量。遗憾的是，这一技术要在每次访问 LT 对

象时指定一个附加的间接引用，这样为复原不再需要的对象可能需要一个相当复杂的存储分配系统。当然一个良好的优化编译程序和运行系统是能够缓解这些问题的，但我们在上面的队列程序包中采取了一个更保守的方法，只运用了数组、整型和分程序结构的存储分配方案。

实现初始化还有其它方法。例如，可把为抽象所需的数据和一个布尔变量一起放在一个记录里，用布尔变量指示该记录是否已初始化[§3.7]。记录分量的缺省表达式能保证这个布尔变量的初始值为 False。程序包中的所有例程在使用该包中的数据结构前，首先要（通过检测记录中的布尔域）确认它。我们认为这种方法常常是不经济的。数据结构中的多数更精巧的算法都力图通过一些特殊情况，如上面提到的哑块，来避免这些额外的测试。若在变量的每个使用之前都需要专门的测试，将会使这些算法的额外价值降低许多。

本节我们考察了依靠记录分量缺省表达式的几种初始化技术。希望程序包的使用者能避免显式调用初始化过程。为初始化具有受限分量的任意记录类型，需要采用布尔变量技术，或运用指针这一动态存储技术。这两种技术都会使源代码曲解，带来运行时的一些额外开销，其严重程度取决于基本 Ada 的实现。我们对曲解源代码感到反感，且也不想接受运行时的额外开销。因此，我们认为最好还是采用显式初始化过程。为其使用者提供抽象数据类型的程序包应同时提供显式的初始化过程，这样确保了可改变数据结构的表示。程序包的使用者在对程序包中类型的某一对象进行任何其它操作前必须要调用这个初始化过程。

终止化的处理不象初始化那样容易。非冻结队列的当前实现，如数组，在退出包含该队列变量的作用域前，不需要任何最终处理。从块和过程中退出时会自动收复存储。然而，假定队列元素存储在通过 new 显式分配的存储中，在退出包含队列变量的作用域时，队列元素所占的存储仍将保留。通过提供一个用户层机制，如

Destroy_Queue 子程序，程序包减轻了提供一个释放无用对象工具的负担。

对数据结构的初始化与终止化问题的全面讨论及其解决方法可查看最近发表的一篇论文[Sherman 82]。

2.4.3 向任务传递判别式

正如上例所示，向抽象数据类型传递一些参数，这可能是有效的，如上例中的队列大小规格说明和元素类型。有两种方法来实现它，其一是将类属参数传给程序包[§ 12.1.1, 12.1.2]，其二是为被声明的类型提供一判别式[§ 3.7.1]。

为每种大小的队列例举一次程序包搅乱了程序正文。而且，这意味着每次例举将生成一个新的队列类型，用户将不得不复制子程序以对付每种大小的队列①。

允许把抽象数据类型的大小作为子类型的一部分，使程序员更方便，更明潦。这可在记录类型中使用判别式约束来实现。因而两种队列都可用记录来实现。

对于冻结队列，必须要有一种方式来把这些判别式的值传给任务。虽然它已用 2.4.2 节讨论的隐式初始化实现，但也能借助 Init_Queue 过程和 Pass_Discriminants 入口来实现。

2.4.4 Remove 当作一过程

把 Remove 规定为一个函数而不是一个过程，这似乎是自然的事。我们认为，如果一队列被一操作修改，它必须当作 **in out** 参数传递。然而，Ada 禁止传给函数的参数为 **in** 模式[§ 6.5]，因此我们强制 Remove 为一过程。

---

① 这类似于 Pascal 中的著名问题：具有不同大小的数组具有不同的类型。这意味着人们不能简单地编出一个过程以对任意整型数组排序。

第 3 章

提供迭代算子的简单图包

3.1 描述

在计算机科学中，图是一种重要的数据结构．本范例描述了一个简单的、提供有向图抽象的程序包的实现，规格说明包含了最终用 Obj 记录类型实现的类型定义．Obj 用来定义有向图的结构．用户的职责是对形成图的结点进行分配、初始化和处理．

本图包的基本功能是提供一个迭代算子，它将提供一种手段，使用户有可能来定义循环抽象机构．该迭代算子提供了可用来进行宽度优先遍历的一种设施．

人们通常希望有一个完整的程序库图包，它提供的功能比我们的实例所提供的功能还要强．此外，这个程序包应该隐蔽当前可见的大量细节，能提供一个规范化的、独立于具体实现的节点创建、处理以及删除的方法．不过，我们未采纳这种实现方法，以免使读者在了解迭代算子时感到很吃力．

宽度优先遍历算法是对霍尔维兹（Horwitz）和塞福宁（Sahni）的著作[Horowitz 78]（《计算机算法基础》，译者注）第 264 页上的算法的改进版本（参阅 3.4.1 节），用迭代算子表达的其他遍历算法则很容易地得到．例如，深度优先遍历就是主要的实例．

3.2 规格说明

为使用本迭代算子，必须声明一个 Breadth__First 类型的变量，该迭代算子通过调用过程 Start 被初始化，过程 Start 带有三个参数：

B　　要初始化的迭代算子。

N　　进行宽度优先遍历的起始节点，如果 N = **null**，那么引发异常 Null__Node，并且没有完成初始化。

Max__Nodes　本次搜索中所能达到的最大节点数，可以用很少的额外开销提供该值的有富裕量的估计值，如果这个数值小了，则由函数 Next（在以后描述）在某点引发异常 Too__Many__Nodes。

根据所选择的算法，存在一个可以进行图迭代（无论迭代完成与否）的总次数的上限值。如果超出此上限值，则引发异常 Too__Many__Traversals（见 3.4.1 节）。

一旦迭代算子被正确地初始化，则可以使用三个例程：

- 调用函数More，确定是否有尚未被产生然而是可以到达的节点。
- 调用函数Next以得到下一搜索点，对任意迭代算子B，如果 More（B）= False，则调用 Next（B）将引发异常 End__of__Graph。
- 调用过程Stop来传递有用的迭代终止的信息。Stop调用之后，这个迭代算子还可被再用。

不对迭代算子进行初始化就试图调用子程序 More、Next 或 Stop 将导致引发异常 Start__Error，不调用过程 Stop 而要多次用过程 Start 启用同一迭代算子也将引发异常 Start__Error。此外，在任一时刻，每个包实例只能有一个激活的迭代算子，否则将引发

异常 Start__Error.

在遍历期间，如果图的边被修改，则该程序包的实体也不依赖于与所发生动作有关的语义.

3.3 程序正文

```
-----------------------------------------------------------
--具有迭代算子的有向图包
-----------------------------------------------------------
generic
   type Item is limited private;

package Graph__Package is
   type Obj;
   type Node is access Obj;
    --
    -- 下面两个类型定义使得一个节点可以有任意多个后继节点
    --
   type Sons__Array is array (Positive range <>) of Node;
   type Sons is access Sons__Array;
   type Hidden__Type is limited private;
   type Obj is
      record
         Contents : Item;
         Descendants : Sons;
         Hidden : Hidden__Type;  -- "隐蔽" 域
      end record ;

   --
   -- 宽度优先迭代算子的说明
   --
   type Breadth__First is limited private;
   procedure Start(B : in Breadth__First;
                   N : in Node;
```

```
                    Max_Nodes : in Positive);
    function More(B : in Breadth_First) return Boolean;
    function Next(B : in Breadth_First) return Node;
    procedure Stop(B : in Breadth_First);

    Start_Error, Null_Node, End_Of_Graph : exception;
    Too_Many_Traversals, Too_Many_Nodes : exception;

private
    type Hidden_Type is
        record          -- 为了能隐式地初始化
            Counter : Integer := Integer'FIRST;
        end record ;

    task type Breadth_First is
        entry Start(N : in Node;
                    Max_Nodes : in Positive);
        entry More(B : out Boolean);
        entry Stop;
    end Breadth_First;
end Graph_Package;

with Queue_Package;      -- 取自本报告的第 2 章
package body Graph_Package is
    Start_Flag : Boolean := False;
    Counter : Integer := Integer'FIRST;

  -- 在每次遍历开始时 Counter 加 1，每一节点的 Counter 域（在
  -- Hidden 域内）包含有上次该节点所产生的 Counter 值的副本。
  -- 在本次遍历中，当达到一个节点时，将其 Counter 域的值与
  -- 变量 Counter 的值进行比较，以确定该节点是否已被搜索过。

    procedure Start(B : in Breadth_First;
                    N : in Node;
                    Max_Nodes : in Positive) is
    begin
        if Start_Flag then
            raise Start_Error;
```

```
    end if;
    if N = null then
       raise Null_Node;
    end if;
    B.Start(N, Max_Nodes);
    Start_Flag := True;
end Start;

function More(B : in Breadth_First) return Boolean is
    Flag : Boolean;
begin
    if not Start_Flag then
       raise Start_Error;
    end if;
    B.More(Flag);
    return Flag;
end More;

function Next(B : in Breadth_First) return Node is
    N : Node;
begin
    if not Start_Flag then
       raise Start_Error;
    end if;
    B.Next(N);
    return N;
end Next;

procedure Stop(B : in Breadth_First) is
begin
    if not Start_Flag then
       raise Start_Error;
    end if;
    B.Stop;
    Start_Flag := False;
end Stop;

task body Breadth_First is separate;
```

```
end Graph__Package;

-- ------------------------------------------------------------
-- Breadth__First 程序包体
-- ------------------------------------------------------------
separate (Graph__Package)
task body Breadth__First is
   Current : Node;     -- 下一个要扩展的节点
   Size : Positive;    -- 为 Start 入口调用保存 Max__Nodes
                       -- 参数的副本，因为入口调用参数的作
                       -- 用域被限定在访问任务体内。所以这
                       -- 是必需的[§8.1，8.2]

   package Q is new Queue__Package(Node);
   procedure Next__Body(N : out Node;
                        Queue : in out Q.Queue) is separate;
   begin
      accept Start(N : in Node; Max__Nodes : in Positive) do
         if Counter = Integer'LAST then
            raise Too__Many__Traversals;
         end if;
         Counter := Counter + 1;
         Current := N;
         N.Hidden.Counter := Counter;
         Size := Max__Nodes;
      end Start;

      declare
         Queue := Q.Queue(Size);
      begin
         Q.Init__Queue(Queue);
      Iterator__Operations:
         loop
            select
               accept More(B : out Boolean) do
                  B := Current /= null;
               end More;
```

```
            or
               accept Next(N : out Node) do
                  Next_Body(N, Queue);
               end Next;
            or
               accept Stop do
                  Q.Destory_Queue(Queue);
               end Stop;
               exit Iterator_Operations;
            or
               terminate;      -- 当超出作用域时则简单终止
            end select;
         end loop Iterator_Operations;
      end ;
   end Breadth_First;

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Next_Body 的体
-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
separate (Graph_Package.Breadth_First)
procedure Next_Body(N : out Node;
                    Queue : in out Q.Queue) is
begin
   if Current = null then
      raise End_Of_Graph;
   end if;
   N := Current;
   if N.Descendants /= null then
      for I in N.Descendants.all'RANGE loop
         declare
            Desc : Node renames N.Descendants(I);
         begin
            if Desc /= null
            and then Desc.Hidden.Counter < Counter then
               -- 该节点在本次搜索中尚未被检索过
               Q.Append(Queue, Desc);
               Desc.Hidden.Counter := Counter;
            end if;
```

```
            end;
         end loop;
      end if;
      if Q.Is_Empty(Queue) then
         Current := null;
      else
         Q.Remove(Queue, Current);
      end if;
   exception
      when Q.Full_Queue = >
         Start_Flag := False;
         Q.Destory_Queue(Queue);
         raise Too_Many_Nodes;
   end Next_Body;
```

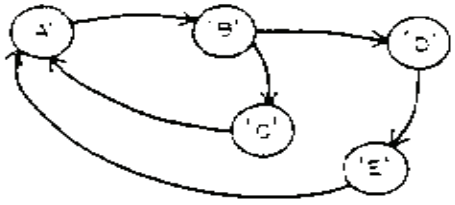
3.4 讨论

3.4.1 算法

我们之所以选择这个用于图遍历的特殊算法，是因为它不必在每个节点上设置"已被访问"的标志，这使得我们能够节省用来访问所有标志的额外开销（在每次遍历前还要清除这些标志）。

图的表示如图 3-1 所示，它表示了一个由 5 个节点组成的图，其中每一节点都包含有一个字符。注意每一节点都是由一个 Obj 记录表示的，从节点出发的边被保存在单独分配的指针数组中。

通过维护在每个迭代算子内的全局计数器和在每一节点内的局部计数器使算法工作。全局计数器在每次遍历（通过调用过程 Start 初始化）开始时加 1。节点中的局部计数器保存有在全局计数器中最后一次搜索节点的数值。在遍历期间，节点中的局部计数器与全局计数器比较，以确定该节点在本次遍历中是否已被搜索过，当且仅当局部计数器与全局计数器相等时，该节点已经被搜索过。



由 new Graph_Package(Character) 得到实例

(抽象表示)

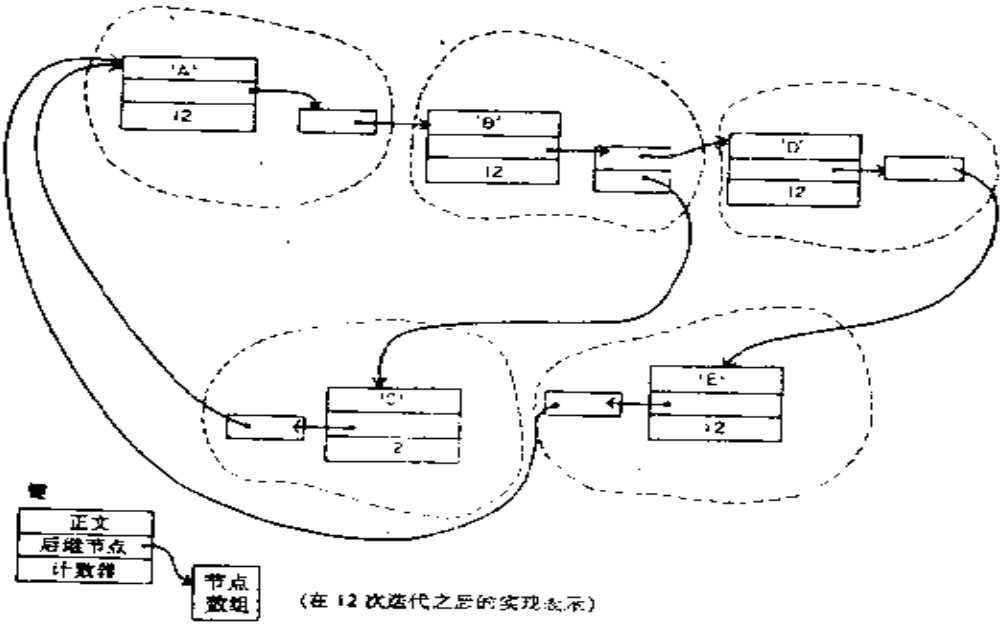
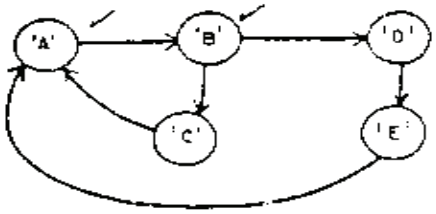


图 3-1　图的表示

(开始于节点 A 的第 13 次迭代)
(✓ = 已被访问过)
(当前正在访问节点 B)



(抽象表示)

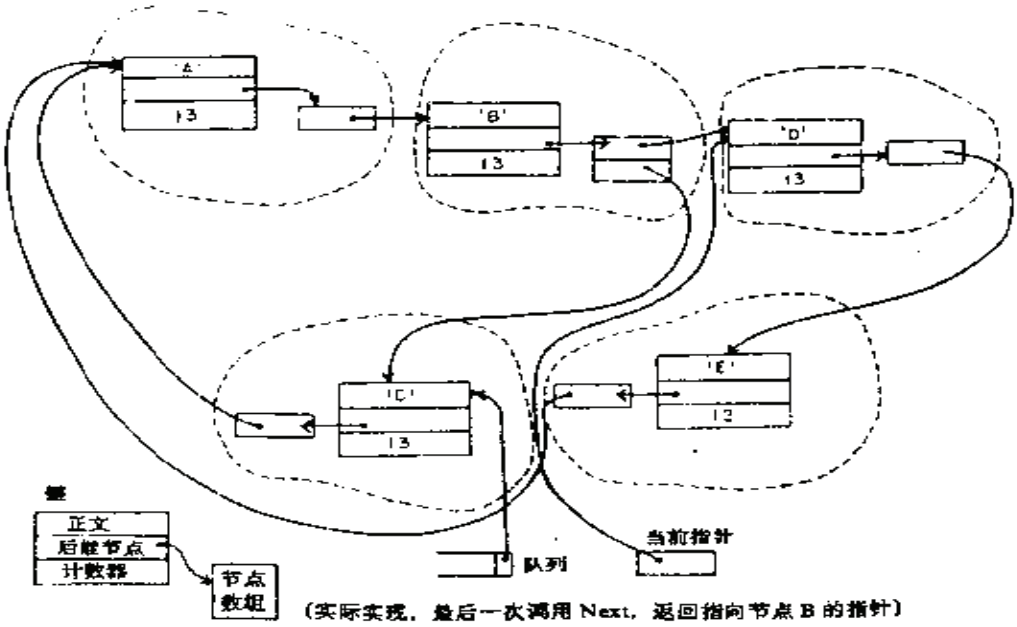


图 3-2　图遍历的状态

迭代过程中图的状态如图 3.2 所示。节点 A 已经被处理过，过程 Next 的最后一次调用返回节点 B，节点 C 和节点 D 已由迭代任务作了部分处理，宽度优先算法还未搜索到节点 E。

为保证正确性，当全局计数器溢出时，应该终止遍历，可以认为，对于某些编译程序的实现来说，这是对使用这个设施的严重限制。例如，PDP-11 标准整型只有 16 位，仅允许 64000 次遍历。然而，因为全局计数器的分配基于每一实例基线，故只有对于特殊程序包实例的遍历才受影响。

缓解这个问题的可行机制是给程序包 Graph__Package 增加第二个类属参数：

```
generic
   type Item is private;
   type Counter__Type is range < >;
```

用户可以将任一 Integer 类型用作 Counter__Type 类型，这个整数类型可以用于声明全局计数器和局部计数器，这将允许用户使用较大整数类型，诸如 Long__Integer 和 Long__Long__Integer，来获得遍历次数的更大上限值①。

3.4.2 信息隐蔽

在一般情况下，当设计一个具有抽象数据类型的程序包时，人们希望提供两类结构的类型：一类是对实体可见的（且通常由实体处理），而另一类则对实体隐蔽，隐蔽部分一般包含了特殊实现的专用数据。良好的编程实践表明，用户应当避免自己去访问这些隐蔽信息。在用 Ada 实现时似乎没有这个硬性规定，但是，有许多

---

① PDP-10 上的 36 位整数类型允许大约 640 亿次遍历，如果遍历以每秒 100 次的平均速率初始化，也将需要用 20 年的时间才能完成一个迭代算子的工作。

可用的选择。

显而易见的方法是对用户隐蔽所有信息，这可以很容易地通过把抽象数据类型说明成（受限）私有类型来实现。这种方法的缺点是程序包必须提供用于访问和修改类型结构的可见部分的设施。通过在程序包体内说明完成这些工作的过程（也可以是函数）很容易实现这个设施。我们有限的经验表明，这大大增加了规格说明和使用抽象的复杂度。

例如，语句

```
X.Contents.Flag := True;
```

必须表示成

```
Temp__Contents := Get__Contents(X);
Temp__Contents.Flag := True;
Set__Contents(X, Temp__Content);
```

不过，由于严格控制了对对象的所有访问，这种方法确实允许在抽象层次上进行更细一层的控制。

我们认为，与以前的方案相比，现在选择的方案具有较少的控制，而且可读性更好。

这种思想是通过在公有类型内放入所期望的部分来强制形成隐蔽，但在域内所包含的这些类型是受限私有类型。这样就避免了用户去做任何明显与数据有关的工作。遗憾的是，用户仍然可以选择该类型的隐蔽域、声明该类型的变量、记录域和形参。然而，这些对象最终还是不会执行任何操作。

一个更重要的问题是用户不可能在抽象数据类型的对象之间执行赋值操作，这是因为它包含有一个受限私有域[§ 7.4.4]。相反，如果需要的话，可以用程序包显式地提供该类型的等价数据类型。

这种解决方法公认是不完善的，但它在上面所述的方法与无信

息隐蔽的方法之间提供了一种信息隐蔽的形式。

3.4.3 In 与 In Out 参数

在 2.4.4 节曾经指出，欲被修改的参数应当用 **in out** 参数传递。在 Start、Next 和 Stop 的规格说明中，我们没有遵守这条原则。在这种情况下，我们认为通过将 Next 声明成函数而获得的可读性超过了对其它方面的考虑（请记住函数不允许有 **in out** 参数[§ 6.5]）。

3.4.4 使用迭代算子

下面的例子给出迭代算子设施的一种使用方法。

```
function Reach(From, To : in Node) return Boolean is
--
--确定节点 To 是否可从节点 From 到达
--
   B : Breadth_First;
begin
   Start(B, From, Size);          -- Size 是全局变量
   while More(B) loop
      if Next(B) = To then
         Stop(B);
         return True;
      end if;
   end loop;
   Stop(B);
   return False;
end Reach;
```

利用选择循环终止技术，可以将 Reach 的体改写为

```
Start(B, From, Size);          -- Size 是全局变量
loop
   begin
```

```
            if Next(B) = To then
                Stop(B);
                return True;
            end if;
        exception
            when End_Of_Graph =>
                Stop(B);
                return False;
        end;
    end loop;
```

3.4.5 迭代算子与类属过程

我们决定提供迭代算子是基于这样一个概念，即迭代算子提供了一个用于循环的自然且熟悉的机构。Ada 的 **for** 循环就是迭代算子的一种简单形式。许多语言，著名的有 IPL-V[Newell 64]，CLU[Liskov 77]和 Alphard[Hilfinger 78，Shaw 77]，都包含有在语言定义中用户可直接定义的迭代算子设施。

对迭代算子的一种选择是能够进行图遍历的类属过程[§ 12.1.3]。考虑下面的规格说明：

```
generic
    with procedure Visit(N : in Node; Continue : out Boolean);
procedure Breadth_First(N : in Node;
                        Max_Nodes : in Positive);
```

为了使用这一设施，实体在参数 Node 上用完成期望动作的过程对 Breadth_First 取实例，当要停止时，Continue 参数向 Breadth_First 过程传递消息。对每一个已到达的节点，Breadth_First 通过调用 Visit（实际上是已经为 Visit 取得实例的实际过程参数）完成一次操作。当 Continue 为 False 时，Breadth_First 终止其动作。

前面提到的过程 Reach 可写成

```
function Reach(From, To : in Node) return Boolean is
   Result : Boolean := False;
   procedure Visit(N : in Node; Continue : out Boolean) is
   begin
      if N = To then
         Result := True;
         Continue := False;
      else
         Continue := True;
      end if;
   end Visit;
   procedure Walk is new Breadth_First(Visit);
begin
   Walk(From, Size);        -- Size 是全局变量
   return Result;
end Reach;
```

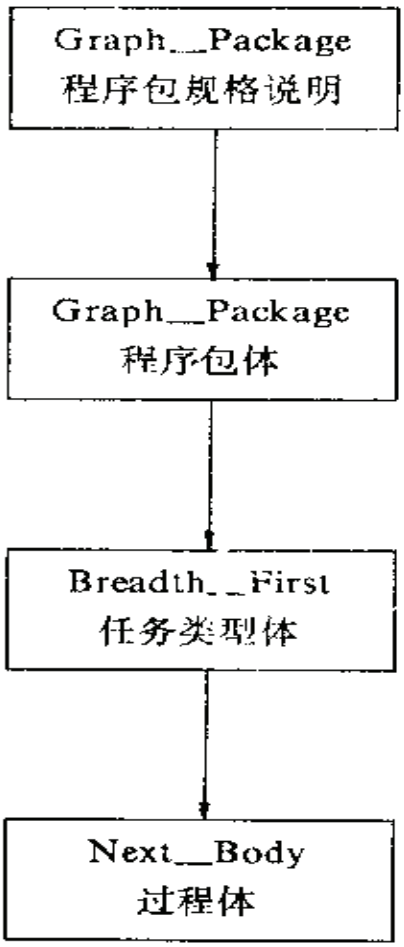
一些人反对采用类属过程的主要论点是类属过程的使用使得循环的执行变得难以理解，而迭代算子却提供了一种展示循环的设施。

3.4.6 分别编译

图设施可分为四个编译单元，它们必须以下面的顺序编译：Graph_Package 程序包规格说明、Graph_Package 程序包体、任务类型 Breadth_First 的体、Next_Body 过程体，这个顺序如图 3-3 所示。这样的分解似乎给程序开发带来的好处并不大：因为改变图节点的表示很可能需要对编译单元进行修改和重新编译。

将图设施分为多个编译单元的部分动机是改善程序的可读性。为此，我们认为很有必要将 Next_Body 的体作为一个编译单元。Ada 仅允许在编译单元最外层的声明部分中声明的单元能被分别编译，如果任务 Breadth_First 的体在程序包 Graph_Package 的体中出现时，则 Next_Body 的体就不可能被分别编译，故需要将

Breadth_First 的体作为一个编译单元。类似的情况还会出现在以后的章节中。



"→" 表示 "必须在前编译"

图 3-3　图设施的编译顺序

第 4 章

PDP-11 控制台驱动程序

4.1 描述

在嵌入式系统中，典型的功能是由设备驱动程序完成的，它提供系统与输入／输出设备的特殊硬件需求之间的一个便利接口。该程序提供的功能包括设备的缓冲请求、确保这些请求的完整性和有效性、以及来自硬件的字段中断。

这个实例用 PDP-11 控制台终端的某些特有细节展示了一个可被广泛应用的总体结构[DEC 76a，DEC 76b]。

设备驱动程序在关于基本运行系统和实现方面做了如下假设：

- 为Low_Level_IO程序包描述通用硬件类型，包括常量 Console_Keyboard_Control、Console_Keyboard_Data、Console_Printer_Control 和 Console_Printer_Data[§ 14.6]。并且假设系统把调用 Send_Control 和 Receive_Control 看作是具有正确长度以及在正确存储单元上的读操作和写操作(例如，将在控制台设备上的操作变换到读写内存单元 $777560_{⑧}$～$777566_{⑧}$)。
- 为入口指定存储单元意味着使用该存储单元的中断将被转换为对该入口的调用[§ 13.5.1]。
- 中断不能直接传递数据。因此，所有具有地址子句的入口都不应当有参数。

此外，为了能使用本程序包，对于任务规格说明，还需要下列

信息：

- 本程序包不能保证所有中断都能得到服务，如果基本运行系统能保证将所有的中断转换成入口调用，则本程序包就不会丢失任一中断。这就是说，对这些中断不需要做什么适当的服务。对 PDP-11 计算机来说，在下一次中断处理之前，若没有读取设备数据寄存器中的数据，则本中断所指示的数据就会丢失。
- 如果对设备的请求快于设备对这些请求的处理，则驱动程序使请求处理阻塞，直到请求能被正确处理。
- 当等待输出设备时，0.5秒的延迟足以完成一个请求。如果在 0.5 秒的时间内输出设备没有响应，驱动程序将初始化下一个字符的传输，而不是对字符再传输一次。
- 提供的数据缓冲为264个字符，传递给过程Write__Character 的字符不加处理就被输出。
- ShutDown入口用于完全终止设备驱动程序的通道。当调用了 ShutDown 之后，当前的缓冲数据就被清除。设备驱动程序也不再对这些数据做进一步的处理或对未完成的中断做进一步的服务。
- Reset入口调用在功能上等价于 ShutDown入口调用之后紧接着重新确定一任务声明。Reset 过程也将力图送出必要的控制信号以重新设置硬件设备。

本程序还对硬件的体系结构做如下假设：

- 输入设备的设备中断向量起始于八进制60，输出设备的中断向量起始于八进制 64.
- 向设备控制寄存器发送八进制值100激活设备中断。

4.2 实现

本实例的中心问题是用 Ada 提供的同步会合机构实现异步处理，做到这一点要显式地用到三个任务：来自程序的请求监控任务、来自输入设备的中断任务和来自输出设备的中断任务，所有这三个任务都通过共享队列进行通讯。只要该队列允许快速访问，系统的任一部分都不会因等待其它任务完成会合而阻塞。然而，如果程序产生的请求快于设备对请求的处理，则队列就会饱和，并阻塞请求任务。

4.3 程序正文

```
package Terminal_Driver_Package is
   task Terminal_Driver is
      entry Read_Character(C : out Character);
      entry Write_Character(C : in Character);
      entry Reset;
      entry ShutDown;
   end Terminal_Driver;
end Terminal_Driver_Package;

with Queue_Package, Low_Level_IO, System;
use Low_Level_IO;

package body Terminal_Driver_Package is
   task body Terminal_Driver is
      --将所有依赖于机器的常数集中在一起
      Console_Input_Vector :
         constant System.Address := 8#60#;
      Console_Output_Vector :
         constant System.Address := 8#64#;
      Enable_Interrupts : Integer := 8#100#;
```

```
Write_Time_Out : constant Duration := 0.5;
Number_Of_Lines : constant := 2;
LineLength : constant := 132;

task type Device_Reader is
   entry Interrupt;
   entry StartUpDone;
    for Interrupt use at Console_Input_Vector;
end Device_Reader;

task type Device_Writer is
   entry Interrupt;
   entry StartUpDone;
   for Interrupt use at Console_Output_Vector;
end Device_Writer;

package Char_Queue_Package
            is new Queue_Package(Character);
use Char_Queue_Package;

type DriverStateBlock is
   record
      InputCharBuffer, OutputCharBuffer :
         Blocking_Queue(Number_of_Line * LineLength);
      CurReader : Device_Reader;
      CurWriter : Device_Writer;
   end record ;

type RefToBlock is access DriverStateBlock;
CurState : RefToBlock;

task body Device_Reader is
   TempInput : Character;
begin
   accept StartUpDone;
   Send_Control(Console_Keyboard_Control,
                  Enable_Interrupts);
   loop
```

```
        accept Interrupt do
            Receive_Control(Control_Keyboard_Data,
                            TempInput);
        end Interrupt;
        Append(CurState.InputCharBuffer, TempInput);
    end loop;
end Device_Reader;

task body Device_Writer is
    TempOutput : Character;
begin
    accept StartUpDone;
    Send_Control(Console_Printer_Control,
                 Enable_Interrupts);
    accept Interrupt;      --由 Send_Control 引起的伪中断
    loop
        Remove(CurState.OutputCharBuffer, TempOutput);
        Send_Control(Console_Printer_Data, TempOutput);
        select
            accept Interrupt;
        or
            delay Write_Time_Out;
        end select;
    end loop;
end Device_Writer;

procedure ShutDownOld is
begin
    abort CurState.CurReader;
    abort CurState.CurWriter;
    Destroy_Queue(CurState.InputCharBuffer);
    Destroy_Queue(CurState.OutputCharBuffer);
end ShutDownOld;

procedure StartUp is
begin
    CurState := new DriverStateBlock;
    Init_Queue(CurState.InputCharBuffer);
```

```
            Init_Queue(CurState.OutputCharBuffer);
            CurState.CurReader.StartUpDone;
            CurState.CurWriter.StartUpDone;
        end StartUp;

    begin
        StartUp;
    Console_Operations:
        loop
            select
                accept Read_Character(C : out Character) do
                    Remove(CurState.InputCharBuffer, C);
                end Read_Character;
            or
                accept Write_Character(C : in Character) do
                    Append(CurState.OutputCharBuffer, C);
                end Write_Character;
            or
                accept Reset do
                    ShutDownOld;
                    StartUp;
                end Reset;
            or
                accept ShutDown;
                ShutDownOld;
                exit Console_Operations;
            or
                terminate;
            end select;
        end loop Console_Operations;
    exception
        when others => ShutDownOld;
    end Terminal_Driver;
end Terminal_Driver_Package;
```

4.4 讨论

4.4.1 包含任务的程序包的使用

Ada 不允许任务作为库单元[§ 10.1]，为了能把任务 Terminal_Driver 包含进库中，我们选择在程序包中封装任务。

4.4.2 任务类型与任务的区别

与本报告中绝大多数的任务例子不同，这里的任务没有明显地被定义为类型[§ 9.2]。任务类型一般在希望有多个特殊类别的对象时使用。而每个物理设备只可能需要一个驱动程序。

进一步说，不把任务说明成类型或类属程序包还有其它原因。一些参数是必不可少的：四个设备名，缓冲区大小以及超时值。设备的这些信息的使用也与任务息息相关。例如，在激活输出设备中断的同时也产生了一个不提供任何数据的伪中断。这一特殊性并不是对所有设备都适用，描述一个任务类型有可能导致用户不适当地使用该驱动程序。

用户可能希望产生一些层次较高的操作，并且能捕获终端驱动程序作为某一抽象类型表示的一部分。遗憾的是，描述一个任务而不是描述一个任务类型的决策使得用户不可能使用终端驱动程序作为用户类型的一个分量。

4.4.3 重启动与终止终端驱动程序

使用终端驱动程序的程序必须能有效地处理设备故障。Ada 语言将 abort 语句[§ 9.10]用于这个目的，但对采用终端驱动程序的程序来说，执行 abort Terminal_Driver 语句将导致从终端驱动程序处理作用域内的不可逆转的退出。入口 Reset 和 ShutDown 允许用户请求终端驱动程序任务恢复设备故障而不会扼杀任务 Terminal_Driver。

4.4.4 与设备的接口

本实例展示了二种设计方案，说明了在硬件和用于 Ada 运行系统之间的一种可行接口。我们假设将中断映射到无形参的入口，我们还决定用程序包 Low__Level__IO 去访问设备寄存器而不是直接去读或写(虚拟存储)设备寄存器。

当设备中断 PDP-11 时，它通常意味着能从设备的数据寄存器中读取数据。将这个数据传递给映射到中断的入口调用似乎是理所当然的。然而，由于三个原因而否定了这种方案。首先，它需要对 Ada 语言参考手册[Department of Defense 82]附录 F 中的入口调用的实现作详细说明。入口调用假定是独立于机器的语言部分，描述与机器有关的部分应当放在 Low__Level__IO 程序包中，这个程序包是依赖于具体机器的[§ 14.6]。

第二，这些细节与入口调用语义联系在一起使得增加设备更加困难，每次在增加新设备时，都要修改编译以便为设备提供操作语义，特别地，要提供新的一类调用序列。用本例中的解释，编译器能将所有使用地址子句的入口调用转换成为具有同一风格的代码。

第三，它将用于中断的缓冲策略从设备驱动程序任务中移到运行系统中，还未被程序接收的挂起中断的数目取决于运行系统提供的缓冲量。如果输入数据也需要保存，则(来自中断的)挂起的入口调用的数目比不保存这个数据更有限，而且，入口调用块在不同的入口之间不能共享，这个存储量的判定将完全由运行系统确定，如果仅仅是中断处理，则运行系统能优化它的存储区以仅仅保留与中断有关的信息，该程序能提供自己的缓冲区来保存或多或少的期望数据。

有些情况，诸如雷达系统，新数据存在后要立即确认，并且需要保存旧数据。用户程序员要想把这样的需求加入进运行系统中是不容易的。在控制台终端驱动程序的情况下，驱动程序的设计者提供 264 个字符的缓冲，只要运行系统真实地传递中断信息，则这个

程序包就能提出有用缓冲区大小的要求。

通过对指定内存单元的读操作或写操作来引用 PDP-11 设备寄存器。通过声明一个带有将其映射到指定存储单元的地址子句[§ 13.5]的变量来获得这些寄存器中的数据也是合情合理的。决定使用 Low_Level_IO 程序包的动机是希望尽可能局部地保持与机器有关。如果使用直接映射的变量，则当读写某给定类型的变量时，赋值语句与类型系统的语义就必须显式地定义其意义。仅提供输入／输出操作的方案是不合适的。对系统支持的每一设备，精确地规定 Low_Level_IO 的子程序是可行的，而且也比较实际。更进一步地说，只要在不同的计算机体系结构的实现之间能保证这些语义，本程序也就能从一台机器移植到另一台机器上。

第 5 章

表创建与表搜索

5.1 描述

本章我们将介绍一个利用类属程序包的例子，这个程序包提供表搜索和对存储在表中的项进行检索的例程。这个表，以及产生用来帮助搜索的辅助数据对象，都被封装在通过对类属程序包取实例而产生的程序包内。除了从这个程序包输出的例程可访问这些表外，其他例程都不能访问这些表。一个程序可以有多个它可以搜索的表;一个程序包必须为每一表取实例，且例程完成特定动作依赖于所对应程序包内的表。

程序包一经取实例，则这个程序包就被确定，表的大小也就固定了。表搜索所返回的项中的键与给定键相"匹配"。如果给定键是一个与某入口相关的键的前导子串，则认为发生匹配。这样的功能可以用于诸如存储编译程序的关键字或存储命令解释器中的命令序列表。

这些表的每个入口都由两个部分组成：一个是键，用 String 串类型(在 Ada 的预定义程序包 Standard 中定义) 实现，用这个键值检索入口，另一个是项，其类型由用户定义。类属程序包 Symbol_Table_Package_Generator 利用项的类型和键的最大长度作为参数来取实例，并且产生一个类属程序包 Symbol_Table，用表作参数的第二个类属程序包实例将产生一个允许进行快速查表的辅助对象，同时产生在表中搜索和检索入口的函数。

提供的类型有：

Entry__Type	表中每一入口的类型。
Table__Type	表的类型——一个具有Entry__Type值的数组
Table__Pointer	这个类型是程序包的私有类型，当搜索表时，返回该类型的值，且它们用于检索入口。
Entry__Collection	一个具有Table__Pointer值的数组。

产生的函数包括：

Make__Entry	该函数带有String和Item__Type类型的参数并返回 Entry__Type 类型的记录。
Search	该函数以String类型为参数，它搜索表并返回 Entry__Collection 数组，这些元素指向其键与给定键匹配的入口。如果没有键匹配，则 Entry__Collection 返回一个空数组。如果有若干个键匹配，则 Entry__Collection 有若干个元素，且 Table__Pointer 按键值的递增顺序存储。
GetKey	该函数带有Table__Pointer类型的参数并检索被指明元素的键。
GetItem	该函数带有Table__Pointer类型的参数并检索被指明元素的项。

如果传给 GetItem 或 GetKey 的 Table__Pointer 参数值越界，则引发异常 Invalid__Table__Pointer，如果一个无初值的 Table__Pointer 类型的变量被用作参数，就可能会发生这种异常。

5.2 实现

该程序不修改作为参数传递的入口表，相反地，它要产生一个辅助的指针数组，该数组利用堆排序[Knuth 73b]在程序包取实例期间进行排序。堆排序的性能比快速排序[Knuth 73b]的性能优越，因为在输入表接近有序的情况下堆排序有较好的性能。

5.3 程序正文

```
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
--类属程序包规格说明
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
generic
   type Item_Type is private;
   MaxSize : in Positive;

package Symbol_Table_Package_Generator is
   type Entry_Type is private;

   function Make_Entry(S : in String;
                       I: in Item_Type) return Entry_Type ;

   type Table_Type is array (Integer range <>) of Entry_Type;

   generic
      Table : in Table_Type;
   package Symbol_Table is
      type Table_Pointer is private;
      type Entry_Collection is array
               (Positive range <>) of Table_Pointer;
      function GetKey(Index : in Table_Pointer) return String;
```

```
        function GetItem(Index : in Table_Pointer) return Item_Type;
        function Search(Key : in String) return Entry_Collection;

        pragma Inline(GetKey, GetItem);
        Invalid_Table_Pointer : exception;

    private
        type Table_Pointer is new Integer
                range Table'FIRST .. Table'LAST;
    end Symbol_Table;

    private
        type Entry_Type is
            record
                Len : Integer := 0;
                Key : String(1 .. MaxSize);
                Item : Item_Type;
            end record ;
end Symbol_Table_Package_Generator;

--------------------------------------------------------------------
--程序包体
--------------------------------------------------------------------
package body Symbol_Table_Package_Generator is
    function Make_Entry(S : in String;
                        I : in Item_Type) return Entry_Type is
        E : Entry_Type;
    begin
        E.Len := S'LENGTH;
        E.Key(1..S'LENGTH) := S;
        E.Item := I;
        return E;
    end Make_Entry;

    package body Symbol_Table is separate;

end Symbol_Table_Package_Generator;
```

```
-- Symbol_Table 程序包体
separate (Symbol_Table_Package_Generator)
package body Symbol_Table is
   Ptr : Entry_Collection(1 .. Table'LENGTH);
   L, R, I, J : Integer;
   Reg : Table_Pointer;

   function GetKey(Index : in Table_Pointer) return String is
      --返回入口的键
   begin
      return Table(Integer(Index)).Key(1 .. Table(Integer(Index)).Len);
   exception
      when Constraint_Error => raise Invalid_Table_Pointer;
   end ;

   function GetItem(Index : in Table_Pointer) return Item_Type is
      --返回入口的项
   begin
      return Table(Integer(Index)).Item;
   exception
      when Constraint_Error => raise Invalid_Table_Pointer;
   end GetItem;

   function Search(Key : in String) return Entry_Collection is
      separate;

   begin
      --初始化 Ptr
      for Cntr in Ptr'RANGE loop
         Ptr(Cntr) := Table_Pointer(Table'FIRST +
                        Cntr - Ptr'FIRST);
      end loop;
      --通过 Ptr 间接地对表进行堆排序
      L := Ptr'LAST / 2 + 1;
      R := Ptr'LAST;
```

```
    Outer__Loop:
        loop
            if L > 1 then
                L := L-1;
                Reg := Ptr(L);
            else
                Reg := Ptr(R);
                Ptr(R) := Ptr(1);
                R := R - 1;
                exit Outer__Loop when R = 1;
            end if;
            J := L;
        Inner__Loop:
            loop
                I := J;
                exit Inner__Loop when J > R / 2;
                J := 2 * J;
                if J < R
                    and then GetKey(Ptr(J)) < GetKey(Ptr(J + 1)) then
                    J := J + 1;
                end if;
                exit Inner__Loop
                    when GetKey(Reg) > = GetKey(Ptr(J));
                Ptr(I) := Ptr(J);
        end loop Inner__Loop;
        Ptr(I) := Reg;
    end loop Outer__Loop;
    Ptr(1) := Reg;
end Symbol__Table;

-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- Search 函数体
-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
separate (Symbol__Table__Package__Generator.Symbol__Table)
function Search(Key : in String) return Entry__Collection is
    --用匹配键搜索入口
    L, U, Posn, P, Psave, Garbage : Integer;
    Found : Boolean;
```

```
   procedure Find(Key : in String;
                  Lower_Bound, Upper_Bound : in Integer;
                  L, U, I : out Integer;
                  Found : out Boolean) is separate;
begin
   --第一次试探
   Find(Key, Ptr'FIRST, Ptr'LAST, L, U, Posn, Found);
   if U > L then          --未检查 L 和 U 之间的所有入口
      P := Posn;          --试探数组的下半段
      loop
         Psave := P;
         Find(Key, L, P-1, L, Garbage, P, Found);
         exit when not Found;
      end loop;
      L := Psave;
      P := Posn;          --试探数组的上半段
      loop
         Psave := P;
         Find(Key, P+1, U, Garbage, U, P, Found);
         exit when not Found;
      end loop;
      U := Psave;
   end if;
   return Ptr(L .. U);
end Search;

-------------------------------------------------
-- Find 过程体
-------------------------------------------------
separate (Symbol_Table_Package_Generator.Symbol_Table.Search)
procedure Find(Key : in String;
               Lower_Bound, Upper_Bound : in Integer;
               L, U, I : out Integer;
               Found : out Boolean) is
   --二分法搜索
begin
   L := Lower_Bound;
```

```
   U := Upper__Bound;
Find__Loop:
   while U >= L loop
      I := (L + U) / 2;
      declare
         Key__Of__Probe : constant String := GetKey(Ptr));
      begin
         if Key__Of__Probe'LENGTH > Key'LENGTH then
            exit Find__Loop when
               Key = Key__Of__Probe(Key__Of__Probe'FIRST ..
                     Key__Of__Probe'FIRST + Key'LENGTH-1);
         else
            exit Find__Loop when Key = Key__Of__Probe;
         end if;
         if Key < Key__Of__Probe then
       U := I - 1;
         else
            L := I + 1;
         end if;
      end ;
   end loop Find__Loop;
   Found := U >= L;
end Find;
```

5.4 讨论

5.4.1 程序包的使用

下面是本程序包用作命令扫描程序的一部分的例子。

```
------------------------------------------------------------
--类属程序包实例
------------------------------------------------------------
type Actions is (Find, Delete, Insert, Alter, Execute, Quit);
package Actions__Symbol__Table is
   new Symbol__Table__Package__Generator(Actions, 8);
```

```
package My_Symbol_Table is
   new Actions_Symbol_Table.Symbol_Table(
      (Make_Entry("find", Find),
       Make_Entry("delete", Delete),
       Make_Entry("insert", Insert),
       Make_Entry("enter", Insert),
       Make_Entry("alter", Alter),
       Make_Entry("amend", Alter),
       Make_Entry("execute", Execute),
       Make_Entry("quit", Quit),
       Make_Entry("leave", Quit)
      )
   );
```

5.4.2 搜索函数的应用

带有匹配键的入口表指针数组 Entry_Collection 由下述策略得到。首先由 Find 例程完成二分搜索，该操作是通过探查表的分段的中点来寻找匹配项，其中分段的段界由 L 和 U 来确定。调用返回后使下述三个条件之一成立。若 L 大于 U，则没有与给定键匹配的元素；若 L 等于 U，则仅有一个匹配元素；若 L 小于 U，则由 P 所指向的元素匹配给定键，且在 L 与 U 之间还可能会有其它匹配键，为了检查这一点，在由 L 与 P−1 确定的数组下半段和由 P+1 与 U 确定的数组上半段用重复的二分搜索完成进一步的探查。这个重复的搜索工作一直要做到对 Find 的调用不再产生指向匹配键 P 为止。前次对 Find 调用所产生的 P 值，要么是表中匹配给定键的最小入口，要么是表中匹配给定键的最大入口。

图 5−1、5−2 和 5−3 以图解的形式说明了这个搜索过程。每一个图解都表示出表中的键、Find 的输入参数值(用向下箭头表示)以及 Find 的输出参数值(用向上箭头表示)。这些图用键值"abm"追踪了对 Find 的调用。

对 Find 的第一次调用定位出一个入口，其键具有"abm"的前缀，该调用的结果如图 5−1 所示。接着函数 Search 调用 Find 以

确定匹配键序列的下限位置，这个调用结果如图 5-2 所示。没有用图说明的其他调用则检查表的下界，作为 Found 参数返回的布尔值表示前次的调用找到了匹配键序列的起点。

接着函数 Search 确定匹配键序列的上限位置，图 5-3 表明了因此而第一次调用函数 Find 的结果。虽然找到了匹配键序列的新的上界，但并不等于所有的匹配键都已找到，因此，没有用图解形式说明的另一次调用用于确定上界。这次调用返回布尔值 False，表示上限已经找到。然后函数 Search 返回符号表的指定分片。

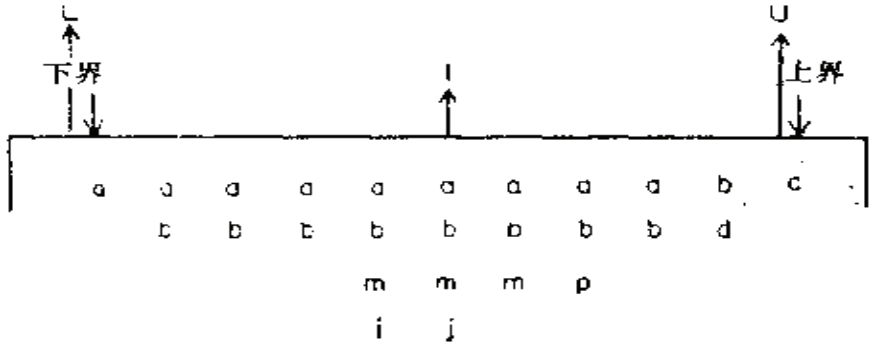


图 5-1　符号表的首次探查

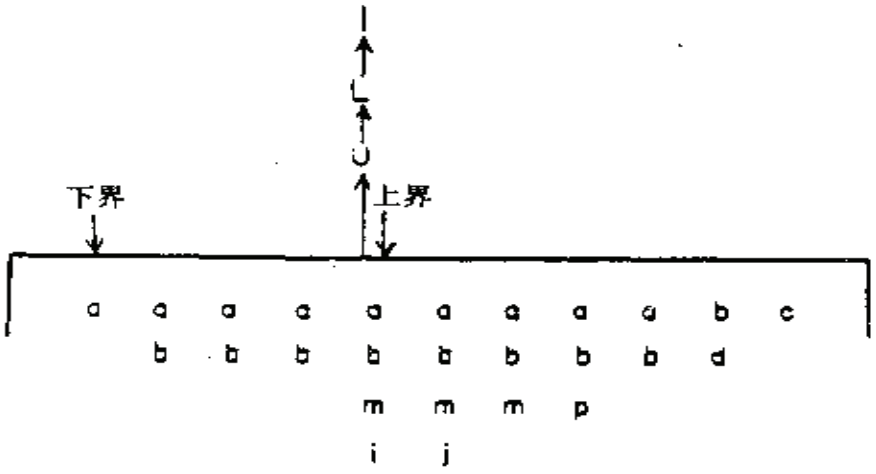


图 5-2　确定匹配的下限位置

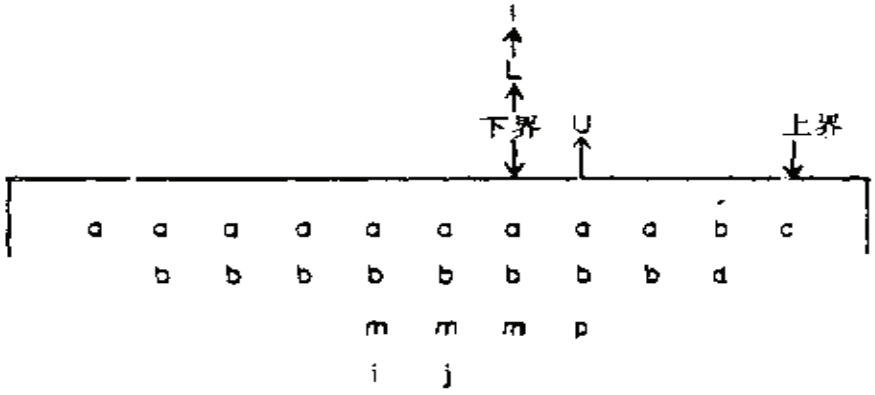


图 5-3　确定匹配的上限位置

5.4.3　多程序包的使用

本实例说明了利用类属程序包实现数据封装的若个特性。

- 表以in参数的形式传入类属程序包Symbol_Table，由于它是 in 参数，则类属实例得到了表的一个备份，这个备份不能被其他程序包控制或修改[§ 12.1.1]。这一点保证了搜索函数在使用这个程序包的程序时，无论发生什么错误都不会失败，尽管该函数依赖于由开始排序时建立的键的适当顺序。
- 由程序包输出的函数Search、GetItem 和GetKey是由该程序包所特有的，故也为该程序包内封装的数据所特有。因此，没有必要向这些函数提供进一步的参数以指明它们要访问的是这些表中的哪一张表。此外，类型Table_Pointer是由该程序包的实例所特有，因此，不可能用其他程序包中的入口指针去访问另外一个程序包中的表。

5.4.4 表中入口的类型

在排序与搜索中，没有用到每个入口中的项，故可把它们声明为程序包的私有类型。在辅助的指针数组上排序，而不是在入口本身的数组上排序，程序包不必为这些入口作出备份，它的优点是项可以是任意长度和任意复杂度的类型，而且不降低排序速度。

为键选择一个合适的类型会引起一些麻烦，大致有三种基本选择，但都并非十全十美。此外，就例子中阐明的选择，应该具备有如下条件：

在上面的例子中，类型Entry__Type有一个附加域Len，它用来记录键的实际长度。相反，如果用 String 类型来实现键域，我们就应当考虑用程序包 Text__Handler[§ 7.6]中的 Text 类型。Text 类型其本身就有 .Pos 域用来记录字符串的实际长度，这时，Entry__Type 应当声明为：

```
type Entry__Type is
   record
      Key : Text__Handler.Text(MaxSize);
      Item : Item__Type;
   end record ;
```

函数 Make__Entry 则应为：

```
function Make__Entry(S : in String;
                     I : in Item__Type)
              return Entry__Type is
begin
   return Entry__Type'(Key = >
              Text__Handler.To__Text(S, MaxSize),
                     Item = > I);
end Make__Entry;
```

遗憾的是，用这种方法有一个问题。类型Text是受限私有类型，故类型 Entry__Type 也应当是受限私有的，因为它有一个元素是受限私有的[§ 7.4.4]。类似地，数组类型 Table__Type 也应当是受限私有的。受限私有类型不能用作类属程序包的 in 模式形参[§ 12.1.1]，就本例而言，类属形参 Table 不能为 in 模式，这使得我们不能将 Table 的备份作为类属实例的一部分。类似地，由于类型 Table__Type 是受限私有的，我们不能在程序包内用赋值语句显式地得到 Table__Type 的备份[§ 7.4.4]。程序包的使用者以及程序包本身被强制地使用了同一表对象。这表明程序包的使用者能够在表开始排序后修改表的内容，并且搜索函数不再以期望的顺序依赖于表。

我们可以将键声明成字符串访问类型，则Entry__Type的声明就变为:

```
type Pointer__To__String is access String;
type Entry__Type is
   record
      Key   : Pointer__To__String;
      Item : Item__Type;
   end record ;
type Table__Type is array (Integer range <>) of Entry__Type;
```

Table__Type具有任意长度的字符串都可以用作键的优点，并且不需要用参数来指明该键的长度（因此外层的类属程序包仅需要类型参数即可）。遗憾的是，在这种实现中也存在一个严重的问题。由于程序包外的使用者可以保存访问表中入口的路径，故有可能改动键值。因此，不正确的用户程序可能以无法预料的方式改变程序包的行为。

在所有三个版本中，表的聚集（类属实例的实参）都是在运行

果表很大，则由于类属实例中表的额外副本过大，就削弱了符号表的用途。

5.4.5 用作表指针的私有类型的应用

为了防止程序包的使用者通过指针对表进行操作和可能将合法的指针转换为非法的指针，Table__Pointer 应当声明为私有类型。即使这样，也仍然存在着未赋初值的 Table__Pointer 经由使用者传递给例程 GetKey 和 GetItem 的可能性。如果发生未定义的值违背了范围约束，则要引发异常 Invalid__Table__Pointer。通过声明一个记录类型和提供初始化的方法[§ 3.7]，我们就可以修改实现，以保证对所有 Table__Pointer 都赋初值。为了避免增加复杂性，本例没有选用这种实现方法。

5.4.6 在类属程序包中嵌套类属程序包

如果要求程序包具有我们所期望的特性，则它至少需要二个类属参数：项的类型以及待搜索的表。然而，由于在将表作为参数传递之前，表的类型应当是确定的，同时还由于 Ada 语言不允许同时向同一类属程序包传递这两个参数，因此需要有两个嵌套的类属程序包：外层的类属程序包带有类型参数，除了其他功能之外，它还要产生出表类型和内层的类属程序包；内层的类属程序包用表作为参数取得实例。

5.4.7 字符串比较

在过程体 Find[§ 4.5.2]内，由于需要进行字符串的比较而引出一些感兴趣的问题。在对表进行试探时，需要对表中被检索出的键完成多种操作。我们宁可对表作一个副本，并在这个副本上执行键操作，也不愿重复地检索这些键。由于在检索前，我们不知道字符串的大小，因此我们必须或者对新的访问值做一个副本，或者对内部块的声明做一个副本。我们选用了后者，以避免对无用存储单元

的收集。

5.4.8 整型在 Find 中的应用

变量 Lower__Bound、Upper__Bound、L、U 和 I 都被声明成整型，尽管它们用于数组 Ptr 的下标，但也没有将其声明成受 Ptr 范围约束的整型量，这是因为本算法允许这些变量的取值范围为 Ptr'FIRST−1 到 Ptr'LAST+1。在 Find 和 Search 的子程序体内，除必须进行的检查外，还会有其他更多的边界检查，但还没有一种简单的方法来避免这一点。我们采取了对 Find 的参数附加范围约束的方法（例如 Lower__Bound 和 Upper__Bound 应该约束在 1..Table__Length 的范围之内，L 和 U 应该约束在 0..Table__Length+1 的范围之内）。这一点有助于更精确地进行错误定位，但它也在一定程度上增加了由程序员引起的附加错误以及源程序的复杂性。

第 6 章

用多个 Ada 任务解拉普拉斯方程

6.1 描述

本节介绍用 Ada 的多任务求解数值问题的方法，该方法是为有多处理器的计算机系统（如 C.mmp[Wulf 72]或 Cm*[Swan 77]）而设计的，在这些计算机系统中，处理器能访问共享存储器。

我们的问题是求解在矩形域 D 上的拉普拉斯偏微分方程①：

$$\frac{\partial^2 U}{\partial x^2} + \frac{\partial^2 U}{\partial y^2} = 0$$

已知在矩形域 D 上的边界条件值为 U。通过对 D 内部的每一网格点计算 U 值来得到拉普拉斯方程的近似解。网格是 m × n 的矩形点阵，(i, j)，i = 1，…，m; j = 1，…，n。

拉普拉斯方程可以由差分方程来近似：

$$U_{i+1,\,j} + U_{i,\,j+1} + U_{i-1,\,j} + U_{i,\,j-1} - 4U_{i,\,j} = 0$$

在点(i, j)的 U 值是 4 个相邻点的 U 值的平均值。对 D 内的每个

---

① 本文中，我们严格按德尔科斯特(Dehlquist)和伯约克(Bjorck)方法进行处理[Dahlquist 74]，参见（参考文献[Dahlquist 74]，译者注）5.6 节和 8.6.3 节。

点(i，j)，我们都可以得到这样一个方程式。将这些方程式联立起来，我们就得到所要求解的线性系统的联立方程组。

我们以行主式对点(i，j)进行编号，这样就得到编号 1，…，mn。设向量 u 为：

$$\mathbf{u} = (U_1, U_2, \cdots, U_{mn})$$

则线性系统就可以表示为 Au＝0。它的系数矩阵 A 是一个对称的带状矩阵，且具有：

- 所有的对角线元素等于−4；
- 对于非对角线元素，如果点p相邻于点q，则$a_{p,q} = 1$，否则 $a_{p,q} = 0$。

我们求解这个系统所采用的方法是具有 N 个 Ada 任务的高斯−塞德尔(Gauss−Seidel)迭代的并发程序①。每个任务进程都分配在系统的某一区域上工作。在每个区域上，求解方法就变成一般单进程的高斯−塞德尔迭代。鲍德特(Baudet)对并发高斯−塞德尔法进行了研究，并在 C.mmp 计算机系统上实现了该算法[Baudet 78a, Baudet 78b]。

从第 n 个 $U_{i,j}$ 得到第(n+1)个 $U_{i,j}$ 的近似高斯−塞德尔迭代公式是：

$$U^{(n+1)}_{i,j} = (U^{(n+1)}_{i-1,j} + U^{(n+1)}_{i,j-1} + U^{(n)}_{i+1,j} + U^{(n)}_{i,j+1}) / 4$$

加速高斯−塞德尔法收敛的修正方法是逐次超松弛法，其迭代

① 除了德尔科斯特和伯约克[Dehjquist 74]采用的高斯−塞德尔迭代求解方法和其他教科书中有关的数值方法外，读者还可以参考麦克莱克的著作[McCracken 72]，他在实例研究 11B 部分中给出了该问题的一种单处理机解法。

加速高斯-塞德尔法收敛的修正方法是逐次超松弛法，其迭代公式为：

$$U^{(n+1)}_{i,j} = U^{(n)}_{i,j} + Omega \times (U^{(n+1)}_{i-1,j} + U^{(n+1)}_{i,j-1} + U^{(n)}_{i+1,j} + U^{(n)}_{i,j+1} - 4U^{(n)}_{i,j}) / 4$$

其中 Omega 是逐次超松弛法的参数①。当 Omega 为 1 时，上式就变为前面所说的高斯-塞德尔公式。

6.2 实现

我们所给出的过程 Parallel_Relaxation 以矩阵 U 为参数，并假设 U 的外边通过调用者用所期望的值初始化。该过程计算内部点(即非边上的点) 的 U 值。

在本实现中，系数矩阵 A 并没有明显地给出，更确切地说，这些系数只反映在逐次超松弛的迭代公式中。

过程 Parallel_Relaxation 还有一个参数 NumberOfRegions，该参数指明了创建 Ada 任务的个数，Ada 任务片的分配如图 6-1 所示。

每一个 Ada 任务都在矩阵 U 的一个区域上工作，每一个区域本身又是一个矩阵。这个区域的矩阵网格分布在矩阵 U 的内部，每一个区域上都应有一个 Ada 任务。

应该认真地进行系统整体收敛性的测试。为保证收敛，每一个任务应在其区域上收敛，更进一步地说，所有任务都必须"同时"具有这种收敛性，也就是说，无论这些任务收敛与否，它们都必须是一致的。尽管由于相邻域的变化会很快地引起任务所在域的变化，

① Omega 近似值的选择已在德尔科斯特和伯约克法[Dahlquist 74]中讨论过，参见（参考文献[Dahlquist 74]，译者著）5.6 节和 8.5.3 节。

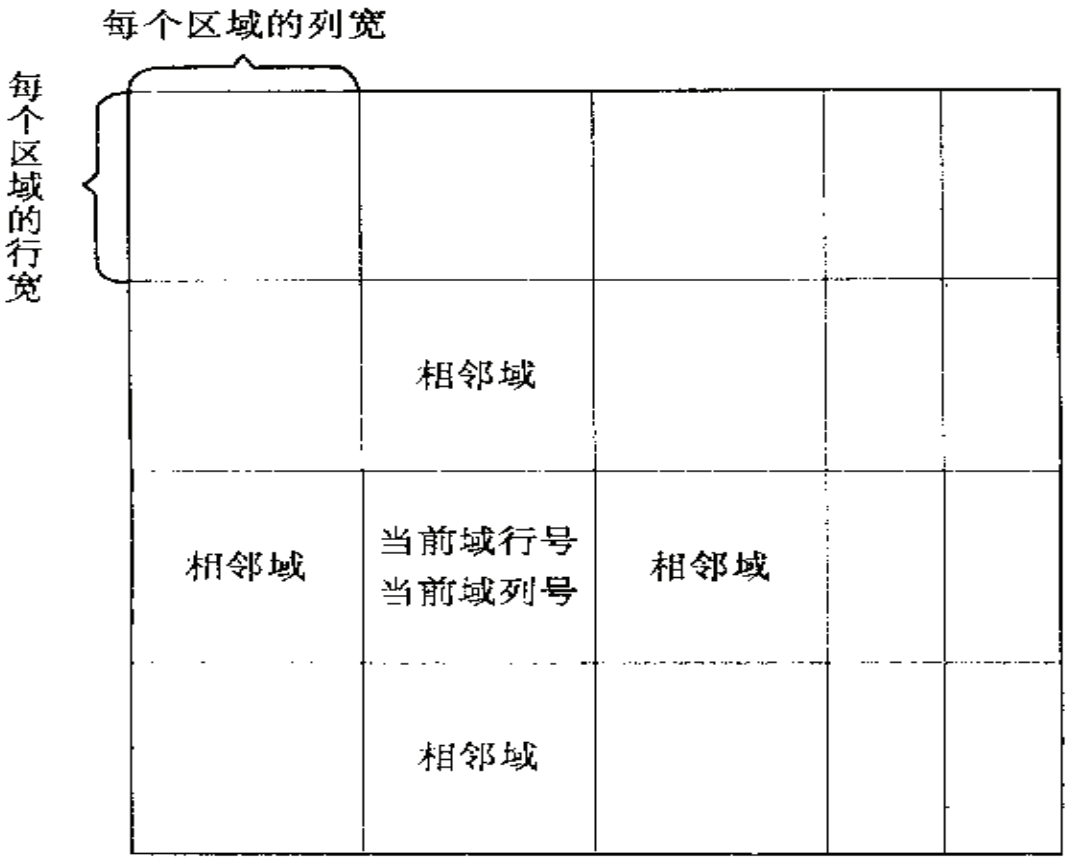


图 6—1　任务的分配

对于每一区域上的任务，有一个协调任务用于区域任务间的通讯。通讯由一个区域任务 A 组成，它负责通知相邻域任务 B，指出任务 A 在其区域上尚未收敛。也就是说，由于 A 尚未收敛，尽管可以局部地认为 B 收敛，但也不能作出 B 收敛的结论。因此，我们可以把从 A 到 B 的消息看作是使 B 继续运行的命令。为了避免由于这些区域任务彼此间直接进行入口调用而引起死锁和资源争夺问题，该消息实际上是通过任务 A 对任务 B 的协调任务作一次

入口调用进行发送。

区域任务间的一致性是通过协调任务彼此传递消息以及由一个未收敛的区域任务数目的全局计数器来达到。如果确信一个区域任务已经收敛，则它要将该计数器值减 1。当一个协调任务收到的消息是其区域任务还未收敛，则它要将该计数器值加 1。当该计数器值为 0 时，则说明所有区域任务都已收敛。

每个区域任务重复执行下列动作。首先，它计算出其对应区域内部点的一组完整的新值。如果该区域还没有局部收敛，则该任务将通知它的相邻任务的协调任务继续执行。如果这个区域任务已经局部收敛，则该任务将未收敛任务的全局计数器值减 1。如果该计数器值为 0，我们的工作也就到此结束了;该任务必须清除任务体系（下面将会讲到），然后我们就回到 Parallel__Relaxation 过程。如果计数器值非零，则区域任务通过调用其协调任务的入口 Wait 而进入睡眠状态。当接收到继续执行的消息时，协调任务接收 Wait 入口调用，唤醒这个区域任务。一旦接收到了这一消息，协调任务就立即接收 Wait 入口调用。协调任务将自上次 Wait 入口调用后所接收到的多条继续执行消息合并成一条消息，因此，多条继续执行消息仅只引起接收一次 Wait 入口。

在上面的测试中，当一个区域任务发现未收敛的区域任务计数器值为 0 时，则它应当设法唤醒其它的区域任务，这些区域任务正在它们各自的协调任务的 Wait 入口上进入睡眠状态。由于任务是在 Ada 的块中声明的，因此，在退出块之前，所有局部于该块的任务都必须终止[§ 9.4]，故必须唤醒其他区域任务，由另一个进入到协调任务的入口 Finish 来负责任务的唤醒。在接收到 Finish 入口调用之后，协调任务将立即接收任一 Wait 入口调用。当一个区域任务从 Wait 入口调用返回后，它再次检查全程计数器值是否已为 0，如果为 0，则该区域任务终止已做的工作并退出。

最后，有些方程组可能不会收敛。作为一个实际问题，提供一个迭代次数的上限很重要。为了做到这一点，Parallel__Ralcxation

[illegible]参数。如果超出这个上限值，我们就将另一参数 DidNotConverge 置为 True 并返回。

6.3 程序正文

6.3.1 受保护的计数器任务类型

本节我们给出一个任务类型，它用增量和减量操作提供一个保护型计数器。在增量和减量操作不可分割时，该计数器将受到保护。通过在接收语句体内实现操作[§ 9.5]获得不可分割性。

我们用程序包封装任务，使得任务可以分别编译并被加进库中[§ 10.1]。

```
package Protected_Counter_Package is
   task type Protected_Counter is
      entry Initialize(Z : in Integer);
      entry Incr(Z : in Integer := 1);
      entry Decr(Z : in Integer := 1);
      entry Read(Z : out Integer);
   end Protected_Counter;
end Protected_Counter_Package;
package body Protected_Counter_Package is
   task body Protected_Counter is
      Counter : Integer;
   begin
      accept Initialize(Z : in Integer) do
         Counter := Z;
      end ;
      loop
         select
            accept Incr(Z : in Integer := 1) do
               Counter := Counter + Z;
            end ;
         or
            accept Decr(Z : in Integer := 1) do
```

```
            Counter := Counter - Z;
         end ;
      or
         accept Read(Z : out Integer) do
            Z := Counter;
         end ;
      or
         terminate;
      end select;
   end loop;
end Protected_Counter;
end Protected_Counter_Package;
```

6.3.2 并行松弛过程

```
generic
   type Real is digits <>;
   type RealMatrix is array
            (Integer range <>, Integer range <>) of Real,

procedure Parallel_Relaxation(U : in out RealMatrix;
                              MaxErr : in Real;
                              MaxIterations : in Positive;
                              NumberOfRegions : in Positive;
                              DidNotConverge : out Boolean;
                              Omega : in Real := 1.0);
```

```
with Protected_Counter_Package, Math_Lib;
with Integer_MaxMin_Lib;
use Integer_MaxMin_Lib;

procedure Parallel_Relaxation(U : in out RealMatrix;
                              MaxErr : in Real;
                              MaxIterations : in Positive;
                              NumberOfRegions : in Positive;
                              DidNotConverge : out Boolean;
                              Omega : in Real := 1.0) is
```

```
   -- MaxErr 用来确定何时收敛（即，当每一点值的变化
   -- 都小于或等于 MaxErr 时，则表示已收敛）
   -- MaxIterations 是要执行的迭代次数的上限，
   -- 如果在执行了 MaxIterations 次迭代后还没有收敛，
   -- 则给 DidNotConverge 赋 True 值并返回，
   -- NumberOfRegions 是要用到的 Ada 任务的最大数目
   -- Omega 是逐次超松弛公式的加速参数

   RowRegions, ColRegions : Integer;
   NumRegions : Integer;
   RowsPerRegion, ColsPerRegion : Integer;

   RowLo : constant Integer := U'FIRST(1);
   RowHi : constant integer := U'LAST(1);
   ColLo : constant Integer := U'FIRST(2);
   ColHi : constant Integer := U'LAST(2);

   subtype InteriorRows is Integer
           range RowLo+1 .. RowHi-1;
   subtype InteriorCols is Integer
           range ColLo+1 .. ColHi-1;

   LenInteriorRows : constant Integer :=
                        (InteriorRows'LAST - InteriorRows'FIRST)+1;
      --离散子类型没有长度属性 LENGTH[§ Appendix A]
   LenInteriorCols : constant Integer :=
                        (InteriorCols'LAST - InteriorCols'FIRST)+1;

   package Real_Math_Lib is new Math_Lib(Real);
   use Real_Math_Lib;

   procedure ParRelax_Inner_Proc is separate;

begin
   --首先假设系统能收敛
   DidNotConverge := False;
   --检查矩阵 U 是否有内部点，如果没有则返回
   if U'LENGTH(1) <= 2 or U'LENGTH(2) <= 2 then
```

```
    end if;
    --给 U 的内部点赋初值 0
    for I in InteriorRows  loop
       for J in InteriorCols  loop
          U(I, J) := 0.0;
       end loop;
    end loop;
    -- 确定矩阵的域的格局
    -- 每个区域本身是一个矩形子矩阵,
    -- 将区域的矩形数组置于该矩阵的上部,
    -- 这个数组是 RowRegions × ColRegions 维的
    RowRegions := Floor(Sqrt(Real(NumberOfRegions)));
    ColRegions := NumberOfRegions / RowRegions;
    NumRegions := RowRegions * ColRegions;
       -- 例如, 当 NumberOfRegions = 33 时, 我们便得到
       -- 5, 6 和 30, 而实际用到的仅仅是 30 个区域

    -- 每个区域是 RowsPerRegion × ColsPerRegion 的矩形域,
    -- 然而, 在右边和底部的区域可能是较小的
    RowPerRegion := (LenInteriorRows + (RowRegions - 1)) /
                        RowRegions;
    ColPerRegion := (LenInteriorCols + (ColRegions - 1)) /
                        ColRegions;
    -- 现在我们知道实际需要多少个任务了,
    -- 调用过程来声明它们并做松弛迭代
    ParRelax__Inner__Proc;
end Parallel__Relaxation;

- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
-- ParRelax__Inner__Proc 的过程体
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
separate (Parallel__Relaxation)
procedure ParRelax__Inner__Proc is

   task type Region__Task is
      entry SetParameter(SetMyRowRegion,
                         SetMyColRegion : in Integer);
```

```
end Region__Task;

task type Coordinator__Task is
   entry Wait;
   entry KeepOnGoing;
   entry Finish;
end Coordinator__Task;

Regions : array (1 .. RowRegions,
                 1 .. ColRegions) of Region__Task;
Coordinators : array (1 .. RowRegions,
                      1 .. ColRegions) of Coordinator__Task;
Unfinished_Counter :
   Protected__Counter__Package.Protected__Counter;

task body Coordinator__Task is
   Had__KeepOnGoing : Boolean := False;
   Had__Finish : Boolean := False;
begin
   loop
      select
         accept KeepOnGoing do
            if not Had__KeepOnGoing then
               Unfinished__Counter.Incr;
               Had__KeepOnGoing := True;
            end if;
         end KeepOnGoing;
      or
         when Had__KeepOnGoing or Had__Finish =>
            accept Wait do
               Had__KeepOnGoing := False;
            end Wait;
      or
         accept Finish do
            Had__Finish := True;
         end Finish;
      or
         terminate;
```

```
        end select;
    end loop;
end Coordinator_Task;

procedure All_Finish is separate;
task body Region_Task is separate;

begin              -- ParRelax_Inner_Proc 的过程体
    --未收敛的任务计数初始化为所有区域任务数
    Unfinisfed_Counter.Initialize(NumRegions);
    --给这些区域任务设置参数.
    for I in Regions'RANGE(1) loop
        for J in Regions'RANGE(2) loop
            Regions(I, J).SetParameter(I, J);
        end loop;
    end loop;
    --现在可以简单地在该过程的尾部等待任务的终止
end ParRelax_Inner_Proc;

-------------------------------------------------------
--过程 All_Finish 的体
-------------------------------------------------------
separate (Parallel_Relaxation.ParRelax_Inner_Proc)
procedure All_Finish is
    --调用所有协调程序的 Finish 入口

begin
    for Rreg in Coordinators'RANGE(1) loop
        for Creg in Coordinators'RANGE(2) loop
            Coordinators(Rreg, Creg).Finish;
        end loop;
    end loop;
end All_Finish;

-------------------------------------------------------
--任务 Region_Task 的体
-------------------------------------------------------
separate (Parallel_Relaxation.ParRelax_Inner_Proc)
```

```
task body Region_Task is
   MyRowRegion, MyColRegion : Integer;
begin
   --通过找出已经被分配的区域启动任务
   accept  SetParameter(SetMyRowRegion,
                        SetMyColRegion : in Integer)  do
      MyRowRegion := SetMyRowRegion;
      MyColRegion := SetMyColRegion;
   end;

Region_Inner_Block:
   declare
      MyDone : Boolean;
      New_Value : Real;
      CurCount : Integer;
      --计算当前区域的边界，它们是本次将要计算的点
      MyRowLo : constant Integer := (MyRowRegion -1) *
                                     RowsPerRegion +
                                     InteriorRows'FIRST;
      MyColLo : constant Integer := (MyColRegion -1) *
                                    ColsPerRegion +
                                    InteriorCols'FIRST;
      -- 如果已经算到底边,则 MyRowHi 不应超出 Interiors'LAST
      MyRowHi : constant Integer :=
                    Min(InteriorRows'LAST,
                        MyRowLo + RowsPerRegion - 1);
      --如果已经算到右边，则类似地有…
      MyColHi : constant Integer :=
                    Min(InteriorCols'LAST,
                        MyColLo + ColsPerRegion - 1);

begin
   Itersloop:
      for Iters in 1 .. MaxIterations loop
         MyDone := True;
         --在当前区域上计算每一点的新值
         for I in MyRowLo .. MyRowHi loop
            for J in MyColLo .. MyColHi loop
```

```
            New__Value := U(I, J) + Omega * (U(I-1, J) +
                          U(I, J-1) + U(I+1, J) +
                          U(I, J+1) - 4.0 * U(I, J)) / 4.0;
            if Abs(New__Value - U(I, J)) >= MaxErr then
               MyDone := False;
            end if;
            U(I, J) := New__Value;
         end loop;          --超出 Cols
      end loop;             --超出 Rows

      if not MyDone then
         --通知当前域的相邻域继续执行
         if MyRowRegion /= 1 then
            Coordinators(MyRowRegion - 1,
                         MyColRegion).KeepOnGoing;
         end if;
         if MyRowRegion /= RowRegions then
            Coordinators(MyRowRegion + 1,
                         MyColRegion).KeepOnGoing;
         end if;
         if MyColRegion /= 1 then
            Coordinators(MyRowRegion,
                         MyColRegion-1).KeepOnGoing;
         end if;
         if MyColRegion /= ColRegions then
           Coordinators(MyRowRegion,
                        MyColRegion+1).KeepOnGoing;
         end if;
      else
         Unfinished Counter.Decr;
         Unfinished Counter.Read(CurCount);
         if CurCount = 0 then
            -- 我们所做的工作已经完成，剩下的只是唤醒每--
            ---个正在睡眠的任务
            All__Finish;
            goto EndOfTask;
         else
           -- 等待，以得到从当前域的相邻域传来的某些
```

```
                -- 变化消息，或等待所有的任务结束:
                Coordinators(MyRowRegion, MyColRegion).Wait;
                -- 当前任务被唤醒，检查这是否是因为每一个任务
                -- 都已完成,或是因为接收到了 KeepOnGoing 的消息:
                Unfinished_Counter.Read(CurCount);
                if CurCount = 0 then
                   goto EndOfTask;
                end if;
             end if;
          end if;
          --检查是否有其他任务已经进行了过多的迭代，
          --如果是这样的话，则停止迭代:
          exit Itersloop when DidNotConverge;
       end loop Itersloop;
       -- 下面的执行当且仅当某一任务(既可能是当前任务
       -- 也可能是其他任务)已经进行了过多的迭代
       DidNotConverge := True;
       All_Finish;
    end Region_Inner_Block;

 <<EndOfTask>>
    null ;
end Region_Task;
```

6.4 讨论

6.4.1 共享变量的应用

在本解法中，我们采用数组 U 作为共享变量。所有的区域任务都可以直接存取 U 而不要有任何附加的协议。这样同时存取共享变量的影响在 Ada 语言中未作详细说明，而是随着实现的不同而变化[§ 9.11]。在某些计算机系统中，一个浮点变量可以占有若干个存储字节，而同时进行读操作和写操作会产生字节交错：在写操作之前和写操作之后，读操作都可以从这个变量值中读取一些字节。因此，读操作所读取的值可能既不是变量的先前值，也不是变

量的当前值。我们要求对共享变量的读操作和写操作是彼此不可分的，所以，本解法还是合理的。注意，我们确实不需要不可分的读操作—修改操作—写操作。

6.4.2 寄存器中共享变量的修改

另一个与共享变量有关的潜在困难是在局部寄存器中保持这种变量的编译问题[§ 9.11]。仅在任务的同步点，即会合点，需要将这些后援存储到共享变量中。有幸的是，在我们的这个解法中，区域任务与协调任务之间具有有效的同步机构，使得在需要的时候可以修改共享变量 U。因此，不必启用预先定义的类属过程 Shared_Variable_Update[§ 9.11]。

6.4.3 类属与类属实例

过程 Parallel_Ralexation 是一个具有二个类属形参的类属过程：一个是浮点数据类型，另一个是该浮点数据类型的二维数组[§ 12.1.2]，这使得用户可以用他声明的任何浮点数据类型对这个过程取实例。当然，他还可以多次地取实例，例如，一次用具有有效数字 6 位(**digits** 6)的浮点类型 HisShort 取实例，另一次用有效数字 12 位(**digits** 12)的浮点类型 HisLong 取实例。

在这个过程体内，我们需要一些公用的数学函数用于类属形参类型 Real（例如，Floor 和 Sqrt）。由于这些函数在预定义程序包 Standard 中不是预定义的子程序，因此我们必须给出这些定义。我们假设类属程序包 Math_Lib 存在，它包含有我们在其他的设施中所需要的函数，我们进一步假设 Math_Lib 有一个浮点类型的类属形参，它提供浮点类型的设施。用类属参数 Real 对 Math_Lib 取实例，我们便得到所需要的例程。

对于在预定义程序包 Standard 中定义的用于浮点类型的子程序(例如 Abs 或"+")，则通过 Ada 的派生类型机制[§ 1.4，§ 3.5.7]，类型 Real 就可以自动地接受这些子程序。

6.4.4 Ada 任务到处理器的调度

本过程力图在多处理器计算机系统上实现 Ada 实例，这个计算机系统备有若干个物理处理器 P。通过 Ada 系统和／或操作系统的控制，N 个任务能调度给这些处理器。当然，我们希望计算机系统能为本程序分配多个处理器，但是，我们还没有办法在 Ada 内部详细说明或保证这一点。由于计算机系统有可能是多道程序的，或者在多个独立的用户程序之间是分时的，因此，必须先假定 P 小于 N，或系统分配给本程序的专用处理器数小于 N。无论正在运行 Ada 任务的处理器数有多少，以及它们之间的相对速度有多大，我们的程序都会正确地执行。

考虑只有一个处理器运行程序的情况。假设初始时这个处理器正在运行某个特殊的区域任务 A，由于该区域任务的相邻域任务没有发生变化，该区域任务最终将局部收敛。任务 A 将会发现全局变量 Unfinished_Counter 非零，于是在其协调任务的 Wait 入口调用处阻塞。由于任务 A 不再具有备选资格，处理器将寻找其它某个有备选资格的 Ada 任务进入运行。由于在会合期间任务的阻塞和解阻，我们的任务在单处理器上将是多路的。

应当看到在单处理器上这个程序能正确执行是这个程序的一个特性，而不是所有具有多个 Ada 任务的任一 Ada 程序的共性。特殊地，如果一个正在运行的 Ada 任务不参与任一会合，就没有必要在基本 Ada 系统的控制下解除调度而让另一个 Ada 任务执行[§ 9.8]。

第三部分

Ada 编程格式约定

北京航空航天大学软件工程研究所

1988 年 6 月

第 1 章

Ada 编程格式约定

前两部分讨论了 Ada 程序设计风格的理论与实例，这一部分以原航空工业部制订的 Ada 编程格式约定为例，阐述在规定语言的编程格式约定时应该注意的问题。这个约定由北京航空航天大学软件工程研究所负责起草，已于 1988 年 6 月经原航空工业部部级鉴定会通过，目前正在为上升成航空航天工业部部级标准而作筹备工作。在我们引用这份资料时，对部分节段作了增删与改动，对若干文字作了修饰，并加上一些必要的解释。

1.1 前言

Ada 语言是一种允许用自由格式书写程序的语言，没有统一的编程格式，因此每个编程者都是按自己的习惯和风格编写程序，这种因人而异的程序格式势必降低程序的可读性，从而对软件的测试、交流和维护带来不利影响，本约定就是为解决该问题而制订的。但应指出，本格式约定只对 Ada 程序的书写格式提出规范化要求，并不涉及程序设计方法及编程技巧等范畴。

1.2 程序清单

根据屏幕显示和行式打印机的打印宽度，约定程序清单的宽度为 80 列，从左端起前 5 列为行号区，后跟一列为空格区，其余 74 列为程序区。

1.3 程序长度

一个程序单元的可执行语句的条数不允许超过 200 行，这里所说的程序单元系指程序包、任务、过程和函数以及它们的规格说明。

1.4 符号定义

符号"←→"用作缩格指示符（在实际的程序行文中并不出现），指示该行对于上行向右的缩格，箭头左端与上行左端对齐，箭头右端指示本行开始位置（具体缩格数，可在 2—7 格范围内任选）。符号"[…]"为可选择符号，表示括在其内的内容是可有可无的。符号"{…}"为迭代符号，表示其中内容可出现 0 次到多次。非终结符由一汉语单词表示。

1.5 程序结构

{**with** 库单元名 {，库单元名} ；}
{**use** 程序包名 {，程序包名} ；}
procedure 过程名 **is**
←→声明部分
begin
←→语句段
end [过程名];

1.6 程序描述体

所有程序单元都由一段注释（称为描述体）开始，它至少包括下述信息，按顺序为：

a. 程序单元名及缩写词的原文；
b. 该程序单元的功能；
c. 对计算机及某些外部设备的依赖性；
d. 程序作者姓名及所在单位和编写日期；
e. 参数类型和含义；
f. 本程序单元调用的其它程序单元名；
g. [其它需说明的问题]。

1.7 注释

利用注释可以使程序变得清晰可读，例如，程序语句只能表示逻辑含义，利用注释可说明其相应的物理含义；程序某部分在以后维护时应注意的事项；等等。注释语句的行数应占整个程序语句总数的 1／5～1／3 之间。注释行要书写整齐，每一注释行（包括继续行）其首部冠以注释的起始标志，并不与源程序语句同占一行。注释行左端与下行语句左端对齐。

1.8 声明部分

1.8.1 基本声明项

1.8.1.1 子程序声明

procedure 标识符 [形参部分];

或

function 标识符 [形参部分] **return** 类型标记;

1.8.1.2 程序包声明

package 标识符 **is**
←→{基本声明项}
[**private**
←→{基本声明项}
end [标识符];

1.8.1.3 任务声明

task [**type**] 标识符 [**is**
←→{入口声明}
←→{表示子句}
end [标识符]];

1.8.1.4 类属声明

generic
←→[标识符表 : [**in** [**out**]] 类型标记 [:= 表达式] ;]
←→[私有类型声明]
←→[**with** 子程序规格说明 [**is** 名字] ;]
←→[**with** 子程序规格说明 [**is** < >] ;]
←→[**type** 标识符 **is** 类属类型定义 ;]
←→程序包规格说明与子程序规格说明

1.8.1.5 记录数据类型声明

type 标识符 [记录判别式] **is**
←→**record**
←→成分表
end record;

1.8.1.6 表示子句中的记录表示法子句

for 类型简单名 **use**
←→**record**
←→[准线子句]
←→[成分子句]
end record;

1.8.1.7 其他基本声明项

以下的基本声明项没有什么特殊结构，他们除按上下文缩格外或超过规定长度需折行外均与前一行左端对齐。

a. 地址子句

for 简单名 **use at** 简单表达式;

b. 使用子句

use 程序包名 {，程序包名};

c. 枚举表示法子句

for 类型名 **use** 聚集;

d. 长度子句

for 属性 **use** 简单表达式;

e. 成分子句

at 静态简单表达式 **range** 静态范围;

f. 准线子句

at mod 静态简单表达式;

g. 延迟常数声明

标识符表 : **constant** 类型标记;

h. 异常声明

标识符表 : **exception**;

i. 常数声明

标识符表 : **constant** := 通用静态表达式;

j. 对象声明

标识符表 : [constant] 约束数组定义与子类型指示
[:= 表达式];

k. 换名声明

标识符 : 类型标记 **renames** 对象名;
或 标识符 : **exception renames** 异常名;
或 **package** 标识符 **renames** 程序包名;
或 子程序规格说明 **renames** 子程序或入口名;

l. 类属设置

function 标识符 **is new** 类属函数名 [类属实在部分];
或 **package** 标识符 **is new** 类属程序包名 [类属实在部分];
或 **procedure** 标识符 **is new** 类属过程名 [类属实在部分];

m. 类型声明 (除记录外)

type 标识符 **is** 类型定义;

n. 子类型声明

subtype 标识符 **is** 子类型指示;

1.8.2 体声明项

1.8.2.1 子程序体

子程序规格说明 **is**
←→[声明部分]
begin
←→语句段;
[**exception**
←→{ 异常处理段}]
end [子程序名];

1.8.2.2 程序包体声明

package body 程序包简单名 **is**

←→[声明部分]
begin
←→语句段
[**exception**
←→｛异常处理段｝]
end [程序包简单名];

1.8.2.3 任务体声明

task body 任务简单名 **is**
←→[声明部分]
begin
←→语句段
[**exception**
←→｛异常处理段｝]
end [任务简单名];

1.9 语句

1.9.1 块语句

1.9.1.1 带声明部分的块语句

[块名：]
declare
←→声明部分
begin
←→语句段
[**exception**
←→｛异常处理段｝]
end [块名];

1.9.1.2 不带声明部分的块语句

[块名：]
begin
←→语句段
[exception
←→｛异常处理段｝]
end [块名];

1.9.2 循环语句

1.9.2.1 基本循环语句

[循环名：]
loop
←→语句段
end loop [循环名];

1.9.2.2 for 循环语句

[循环名：]
for 标识符 **in** [**reverse**] 范围
←→**loop**
←→语句段
end loop [循环名];

1.9.2.3 while 循环语句

[循环名：]
while 条件
←→**loop**
←→语句段
end loop [循环名];

1.9.3 情况语句

```
case 情况选择表达式 is
←→when 备选情况 = >
    ←→语句段
←→{when 备选情况 = >
    ←→语句段}
←→[when others = >
    ←→语句段]
end case;
```

1.9.4 条件语句

```
if 条件
then
←→语句段
{elsif 条件
←→then
    ←→语句段}
[else
←→语句段]
end if;
```

1.9.5 接受语句

```
accept 入口简单名 [ (入口程序) ] [形参部分] [do
←→语句段
end [入口简单名] ];
```

1.9.6 选择语句

```
select
```

←→语句段
{or
←→语句段}
[else
←→语句段]
end select;

1.9.7 简单语句

以上是 6 种复合语句，下面是 11 种简单语句，他们除按上下文缩格外或超长需折行外均与前一行左端对齐，这些语句是：

a. 空语句

null;

b. 返回语句

return 表达式;

c. 出口语句

exit [循环名] [when 条件];

d. 延迟语句

delay 时间表达式;

e. 夭折语句

abort 任务名 {，任务名};

f. goto语句

goto 标识符标号;

《标识符标号》 语句段;

g. 引发语句

raise [异常名];

h. 代码语句

类型标记' 记录聚集;

i. 入口调用语句

入口名 [实参部分];

j. 过程调用语句

过程名 [实参部分];

k: 赋值语句

变量名 := 表达式;

1.10 关于换行的说明

在程序中，除去括在字符串内的";"之外，其余在任何地方遇";"均需换行。换行后，除按本格式约定应缩格外，其余的均与上行左端对齐。

1.11 关于折行的处理原则

当一个逻辑行超出由打印纸或屏幕所提供的宽度时，需要折行，在折行时不应将一个单词拆开，续行也要缩格。若遇到表达式中需折行，续行表达式与前一行表达式左端对齐，例如：

```
MAX := (A + B) * C /
       (D + E);
```

若遇到参数表需要折行，续行参数表与前一行参数表左端对齐，如下

```
procedure FIND(FREE: LINK;
               KEY, DEN: INTEGER);
```

1.12 关于缩格的限制

若程序嵌套深度很深，每次嵌套内层相对外层向右缩格，为避免缩格太多，影响程序书写，甚至缩格超过了程序清单宽度，因此规定从第 7 列起缩到第 50 列以后不再缩格。

1.13 程序的总体结构

根据上述规范，鉴于 Ada 程序总体上为嵌套结构，因此当规定了必要的缩格之后，从外观上看整个程序呈现出锯齿形结构（可参见附录 B 给出的实例）。

附录 A 描述体细节

过程和函数等程序单元均应写出描述体，描述体由各描述部分组成，各描述部分均以注释形式给出。程序描述体写在程序前面，其左界符写在第 7 列上。过程和函数等的描述体写在第一个说明语句之前，其左界符与上行左端对齐。所有描述体的右界符均写在第 80 列上。下面以程序为例给出描述体的书写格式。

```
1 3 5 7·····80
      --程序名及缩写词的原文
      --FUNCTION
      --程序功能说明
      --DEPENDENCE
      --对计算机及某些外部设备的依赖性
      --AUTHOR
      --程序作者姓名，所在单位和编写日期
      --INPUT
      --输入参数名，输入参数类型说明，[输入参数功能说明]
      --（若无输入则填 NONE）
      --OUTPUT
      --输出参数名，输出参数类型说明，[输出参数功能说明]
      --（若无输出则填 NONE）
```

```
--INTERNAL VARIABLE
--局部变量名及说明（若无，则填 NONE）
--EXTERNAL VARIABLE
--全局变量名及说明（若无，则填 NONE）
--CALLS
--所调用的函数名及/或过程名（若无，则填 NONE）
```

附录 B　实例

```
1 3 5 7 ..... 80
--BEAUTIFY
--FUNCTION
--   This program is used to beautify all Ada programs which
--   have a complete grammatcal structure.
--DEPENDENCE
--   This program is independent of machines.
--AUTHOR
--   Ling-Lin, Computer Science and Engineering Department,
--   Beijing University of Aeronautics and Astronautics,
--   May 28, 1986.
--INPUT
--   SOURCE: FILE_TYPE;
--OUTPUT
--   OBJECT: FILE_TYPE;
--INTERNAL VARIABLE
--   NAME, WORD: STRING(1..80);
--   I: INTEGER;
--EXTERNAL VARIABLE
--   All external variable are given by package COMMON.
--CALLS
--   STA, COMMENT, LABLE, SKIPBLANK, COMMON.

with TEXT_IO;
use TEXT_IO;
```

```
package COMMON is
   type STACK is array(1..10) of COUNT;
   SOURCE, OBJECT: FILE_TYPE;
   FIRST, IN_SPACE: POSITIVE_COUNT;
   SPACE: STRING(1..80) := (1..80 = > ');
   CHAR: CHARACTER;
   JUMP: array (0..10) of BOOLEAN := (0..10 = > FALSE);
   X: STACK := (1..10 = > 0);
   IN_LENGTH: INTEGER;
   FW_COUNT, IF_COUNT: INTEGER := 0;
end;

--main procedure
with TEXT_IO, STA, COMMENT,
     LABLE, SKIPBLANK, COMMON;
use TEXT_IO, COMMON;

procedure BEAUTIFY is

   package PO_COUNT is new
           INTEGER_IO(POSITIVE_COUNT);
   use PO_COUNT;

   NAME, WORD: STRING(1..80) := (1..80 = > ');
   I: INTEGER;

begin
   --begin to input source file which needs to be beautified
   PUT("Please input source file name:");
   I := 0;
   while not END_OF_LINE
      loop
         I := I + 1;
         GET(NAME (I));
      end loop;
   OPEN(SOURCE, IN_FILE, NAME(1..I));
   SKIP_LINE;
```

```
--Choose output file's name
PUT("Please input object file name[OBJ.ADA]:");
I := 0;
while not END_OF_LINE
    loop
        I := I + 1;
        GET(NAME(I));
    end loop;
if I = 0
then
    CREATE(OBJECT, OUT_FILE,"OBJ.ADA");
else
    CREATE(OBJECT,OUT_FILE,NAME(,···));
end if;

--choose start column number
PUT ("Please input program start position:");
GET(FIRST);

--choose in_space number
PUT("Please input in_space length:");
GET(IN_SPACE);
SKIPBLANK;

--check to see if the progrom has descriptions?
if CHAR /= '-'
then
    SET_COL(OBJECT,FRIST);
    PUT(OBJECT,"--NOTE: No Comment!");
    NEW_LINE(OBJECT);
    PUT("NOTE: No Comment!");
end if;

--begin to beautify the text
while not END_OF_FILE(SOURCE)
    loop
        if CHAR = '<'
        then
```

```
            LABLE;
            SKIPBLANK;
        end if;
      if CHAR = '-'
      then
          GET(SOURCE, CHAR);
          COMMENT;
      else
          STA;
      end if;
      SKIPBLANK;
      NEW_LINE(OBJECT);
    end loop;
    CLOSE(SOURCE);
    CLOSE(OBJECT);
end BEAUTIFY;
--The other procedures are ignored
```

注：其余部分在此不一一例出了。

参考文献

[Ambler 77] A.L.Ambler, D.I.Good, J.C.Browne, W.F.Burger, R.M.Cohen, C.G.Hoch and R.E.Wells.
Gypsy: A Language for Specification and Implementation of Verificable Programs.
ACM SIGPLAN Notices 12(3), March, 1977.

[Baudet 78a] G.M.Baudet.
Asynchronous Iterative Methods for Multiprocessors.
Journal of the ACM 25(2):226–244, April, 1978.

[Baudet 78b] G.M.Baudet.
The Design and Analysis of Algorithms for Asynchronous Multiprocessors.
PhD thesis, Dept. of Computer Science, Carnegie–Mellon University, April, 1978.

[Bentley 79] J.L.Bentley and M.Shaw.
An Alphard Specification of a Correct and Efficient Transformation on Data Structures.
In Proceedings of IEEE Conference on Specifications of Reliable Software, pages 222–237. IEEE, April, 1979.

[Bowles 77] K.L.Bowles.
Microcomputer Problem Solving Using Pascal.
Springer–Verlag, 1977.

[Brinch Hansen 75]
The Programming Language Concurrent Pascal.
IEEE Transactions on Software Engineering SE–1, June, 1975.

[Brooks 75] F.P.Brooks, Jr.
The Mythical Man–Month: Essays on Software Engineering.
Addision–Wesley, Reading, Massachusetts, 1975.

[Brosgol 82] B.M.Brosgol.
Summary of Ada Language Changes.
ACM Ada Letters 1(3), March, 1982.

[Browne 80] J.C.Browne.
The Interaction of Operating Systems and Software Engineering.
Proceedings of the IEEE 68(9), September, 1980.

[Buxton 70] J.N.Buxton and B.Randell(editors).
Software Engineering Techniques.
NATO, 1970.
Report on a Conference Sponsored by the NATO Science Committee,Rome, Italy, 27th to 31st October 1969.

[Cohen 82] N.H.Cohen.
Parallel Quicksort: An Exploration of Concurrent Programming in Ada.
ACM Ada Letters II(2), September, 1982.

[Dahl 68] O.J.Dahl.
Simula 67 Common Base Language.
Technical Report, Norwegian Computing Center, Oslo, 1968.

[Dahl 72] O.J.Dahl and C.A.R.Hoare.
Hierarchical Program Structures.
In Structured Programming, pages 175–220, Academic Press, 1972.

[Dahlquist 74] G.Dahlquist and A.Björck.
Numerical Methods.
Prentice Hall, Englewood Cliffs, New Jersey, 1974.

[Davis 79] A.M.Davis and T.G.Rauscher.
Formal Techniques and Automatic Processing to Ensure Correctness in Requirement Specifications.
In Proceedings of the IEEE Conference on Specifications of Reliable Software, pages 15–35, IEEE Computer Society, 1979.
IEEE Catalog Number 79 CH1401–9C.

[DEC 76a] Digital Equipment Corporation.
PDP-11 Peripheral Handbook.
Digital Equipment Corporation, Maynard, Massachusetts, 1976.

[DEC 76b] Digital Equipment Corporation.
PDP-11 04 / 34 / 45 / 55 Pocessor Handbook.
Digital Equipment Corporation, Maynard, Massachusetts, 1976.

[Demers 80] A.J.Demers and J.E.Donahue.
Data Types, Parameters and Type Checking.
In Proceedings of the ACM Symposium on Principles of Programming Languages, pages 12–23, ACM SIGACT and SIGPLAN, January, 1980.

[Department of Defense 78]
Department of Defense.
Steelman Requirements for High Order Computer Programming Languages, 1978.

[Department of Defense 80]
Department of Defense.
Requirements for Ada Programming Support Environments: Stoneman, 1980.

[Department of Defense 82]
Department of Defense, Ada Joint Program Office.

Reference Manual for the Ada Programming Language (Draft revised MIL–STD 1815), Draft proposed ANSI Standard document for editorial review.
July 1982 edition, United State Department of Defense, Ada Joint Program Office, 1982.

[DeRemer 76] F.DeRemer and H.H.Kron.
Programming–in–the–Large vs. Programming–in–the–small.
IEEE Transactions on Software Engineering SE–2(2), June, 1976.

[Dewar 80] Dewar, Robert B.K., Fisher Jr., Gerald A., Schonberg, Edmond, Froehlich, Robert, Bryant, Stephen, Goss, Clinton F. and Burke, Michael.
The NYU Ada Translator and Interpreter.
In Symposium on the Ada Programming Language, pages 194–201, ACM, Boston, December, 1980.

[Dijkstra 72] E.W.Dijkstra.
Notes on Structured Programming.
In Structured Programming, pages 1–82, Academic Press, 1972.

[Feiertag 79] R.Feiertag and P.G.Neumann.
The Foundations of a provably secure operating system (PSOS).

In Proceedings of the National Computer Conference, pages 329–334, National Computer Conference, 1979.

[Floyd 67] R.W.Floyd.
Assigning Meanings to Programs.
in J.T.Schwartz(editor), Proceedings of the Symposium in Applied Mathematics, pages 19–32, American Mathematical Society, 1967.

[Gerhart 76] S.L.Gerhart and L.Yelowitz.
Observations of Fallibility in Applications of Modern Programming Methodologies.
IEEE Transactions on Software Engineering SE–2(5), September, 1976.

[Gerhart 79] S.L.Gerhart and D.S.Wile.
Preliminary Report on the Delta Experiment: Specification and Verification of a Multiple–User File Updating Module.
In Proceedings of the IEEE Conference on Specifications of Reliable Software, pages 198–211, IEEE Compater Society, 1979.
IEEE Catalog Number 79 CH1401–9C.

[Geschke 77] C.M.Geschke, E.H.Satterthwaite and J.H.Morris Jr..
Early Experience with Mesa.
Comm. of the ACM 20(8), August, 1977.

[Goldberg 73] J.Goldberg.
Proceedings of the Symposium on the High Cost of Software.
Technical Report, Standard Research Institute, Stanford, California, September, 1973.

[Good 77] D.I.Good.
Constructing Verified and Reliable Communications Processing Systems.
ACM Software Engineering Notes 2(5), October, 1977.

[Goodenough 80] J.B.Goodenough and C.L.McGowan.
Software Quality Assurance: Testing and Validation.
Proceedings of the IEEE 68(9), September, 1980.

[Grogono 78] P.Grogono.
Programming in Pascal.
Addision–Wesley, Reading, Massachusetts, 1978.

[Guarino 78] L.R.Guarino.
The Evolution of Abstraction in Programming Languages.
Technical Report CMU–CS–78–120, Carnegie–Mellon University, May, 1978.

[Guttag 77] J.V.Guttag.
Abstract Data Types and the Development of

Data Structures.
Communications of the ACM 20(6), June, 1977.

[Guttag 78] J.V.Guttag, E.Horowitz and D.R.Musser.
Abstract Data Types and Software Validation.
Communications of the ACM 21(12), December, 1978.

[Guttag 80a] J.V.Guttag.
Notes on Type Abstraction(Version 2).
IEEE Transactions on Software Engineering SE–6(1):13–23, January, 1980.

[Guttag 80b] J.Guttag and J.J.Horning.
Formal Specification As a Design Tool.
In Proceedings of the ACM Symposium on Principles of Programming Languages, pages 251–261, ACM SIGACT and SIGPLAN, January, 1980.

[Heninger 79] K.L.Heninger.
Specifying Software Requirements for Complex Systems: New Techniques and Their Applications.
In Proceedings of the IEEE Conference on Specifications of Reliable Software, pages 1–14, IEEE Computer Society, 1979.
IEEE Catalog Number 79 CH1401–9C.

[Hilfinger 78] P.Hilfinger, G.Feldman, R.Fitzgerald, I.Kimura, R.L.London, K.V.S.Prasad, V.R.Prasad, J.Rosenberg, M.Shaw, W.A.Wulf.
(Preliminary)An Informal Definition of Alphard.
Research Report CMU–CS–78–105, Carnegie–Mellon University, Computer Science Department, February, 1978.

[Hoare 69] C.A.R.Hoare.
An Axiomatic Basis for Computer Programming.
Communications of the ACM 12, October, 1969.

[Hoare 72a] C.A.R.Hoare.
Proof of Correctness of Data Representations.
Acta Informatica 1(4), 1972.

[Hoare 72b] C.A.R.Hoare.
Notes on Data Structuring.
In Structured Programming, pages 83–174, Academic Press, 1972.

[Hoare 73] C.A.R.Hoare and N.Wirth.
An Axiomatic Definition of the Programming Language Pascal.
Acta Informatica 2(4), 1973.

[Horowitz 78] E.Horowitz and S.Sahni.
Fundamentals of Computer Algorithms.
Computer Science Press, Inc., 1978.

[Howden 79] W.E.Howden.
An Analysis of Software Validation Techniques for Scientific Programs.
Technical Report DM-171-IR, University of Victoria Department of Mathematics, March, 1979.

[Ichbiah 79] J.D.Ichbiah, et al.
Rationale for the Design of the ADA Programming Language.
ACM SIGPLAN Notices 14(6B), June, 1979.

[ISO 79] Draft Specification for the Computer Programming Language Pascal.
International Organization for Standardization, 1979.
ISO / TC97 / SC 5 N.

[Jensen 74] K.Jensen and N.Wirth.
Pascal User Manual and Report.
Springer-Verlag, 1974.

[Jones 76] A.K.Jones and B.H.Liskov.
An Access Control Facility for Programming Languages.
MIT Memo 137, Massachusetts Institute of Technology Computation Structures Group and Carnegie-Mellon University, 1976.

[Kernihan 76] B.W.Kernihan and P.J.Plauger.
Software Tools.
Addison–Wesley, 1976.

[Knuth 69] D.E.Knuth.
The Art of Computer Programming. Volume 2: Seminumerical Algorithms.
Addison–Wesley, 1969.

[Knuth 73a] D.E.Knuth.
The Art of Computer Programming. Volume 1: Fundamental Algorithms.
Addison–Wesley, 1973.
Second Edition.

[Knuth 73b] D.E.Knuth.
The Art of Computer Programming. Volume 3: Sorting and Searching.
Addison–Wesley, 1973.

[Lampson 77] B.W.Lampson, J.J.Horning, R.L.London, J.G.Mitchell and G.J.Popek.
Report on the Programming Language Euclid.
ACM SIGPLAN Notices 12(2), February, 1977.

[Liskov 75] B.H.Liskov and S.N.Zilles.
Specification Techniques for Data Abstractions.
IEEE Transactions on Software Engineering SE–1, March, 1975.

[Liskov 77] B.H.Liskov, A.Snyder, R.Atkinson and C.Schaffert.
Abstraction Mechanisms in CLU
Comm. of the ACM 20(8), August, 1977.

[London 75] R.L.London.
A View of Program Verification.
In Proceedings of the International Conference on Reliable Software, pages 534–545, IEEE Computer Society, April, 1975.

[London 78] R.L.London, J.V.Guttag, J.J.Horning, B.W.Lampson, J.G.Mitchell and G.J.Popek.
Proof Rules for the Programming Language Euclid.
Acta Informatica 10(1):1–26, 1978.

[Manna 74] Z.Manna.
Mathematical Theory of Computation.
McGraw--Hill, 1974.

[McCracken 72] D.D.McCracken.
A Guide to Fortran IV Programming, 2nd Edition.
John Wiley and Sons, New York, 1972.

[Millen 76] J.K.Millen.
Security Kernel Validation in Practice.
Communications of the ACM 19(5), May, 1976.

[Miller 79] E.Miller(editor).
Tutorial: Automated Tools for Software Engineering.
IEEE Computer Society, 1979.
IEEE Catalog No. EHO 150–3.

[Morris 73a] J.H.Morris.
Types Are Not Sets.
In Proceedings of the ACM Symposium on Principles of Programming Languages, pages 120–124, ACM, 1973.

[Morris 73b] J.H.Morris.
Protection in Programming Languages.
Communications of the ACM 16, January, 1973.

[Naur 69] P.Naur and B.Randell(editors).
Software Engineering.
NATO, 1969.
Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968.

[Newell 64] A.Newell, F.Tonge, E.A.Feigenbaum, B.F.Green, Jr. and G.H.Mealy.
Information Processing Language–V Manual.
Prentice Hall, 1964.

[Parnas 71] D.L.Parnas.
Information Distribution Aspects of Design Methodology.
In Proceedings of IFIP Congress, pages 26–30, IFIP, 1971.
Booklet TA–3.

[Parnas 72a] D.L.Parnas.
A Technique for Software Module Specification with Examples.
Communications of the ACM 15, May, 1972.

[Parnas 72b] D.L.Parnas.
On the Criteria to be Used in Decomposing Systems into Modules.
Communications of the ACM 15(12), December, 1972.

[Peters 80] L.Peters.
Software Design Engineering.
Proceedings of the IEEE.68(9), September, 1980.

[Quant 79] IEEE Computer Society(editor).
Workshop on Quantitative Software Models for Reliabilty, Complexity, and Cost: an Assessment of the State of the Art.
IEEE Computer Society, 1979.
IEEE Catalog No. TH0067–9.

[Ramshow 79] L.H.Ramshow.
Formalizing the Analysis of Algorithms.
PhD thesis, Stanford University, 1979.

[Schuman 71] S.A.Schuman(editor).
Proceedings of the International Symposium on Extensible Languages.
ACM SIGPLAN Notices 6, December, 1971.

[Schuman 76] S.A.Schuman.
On Generic Functions.
In New Directions in Algorithmic Languages — 1975. pages 169-192, IRIA, 1976.

[Shaw 77] M.Shaw, W.A.Wulf and R.L.London.
Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators.
Comm. of the ACM 20(8), August, 1977.

[Shaw 78] M.Shaw, G.Feldman, R.Fitzgerald, P.Hilfinger, I.Kimura, R.London, J.Rosenberg and W.A.Wulf.
Validating the Utility of Abstraction Techniques.
In Proceedings of ACM National Conference, pages 106–110, ACM, December, 1978.

[Shaw 79] M.Shaw.
A Formal System for Specifying and Verifying Program Performance.

Technical Report CMU–CS–79–129, Carnegie–Mellon University, June, 1979.

[Shaw 80] M.Shaw and W.A.Wulf.
Toward Relaxing Assumptions in Language and Their Implementations.
SIGPLAN Notices 13(3):45–61, March, 1980.

[Sherman 80] M.Sherman and M.Borkan.
A Flexible Semantic Analyzer for Ada.
In Symp. on the Ada Programming Language, pages 62–71, ACM, Boston, December, 1980.

[Sherman 82] M.Sherman, A.Hisgen and J.Rosenberg.
A Methodology for Programming Abstract Data Types in Ada.
In Proceedings of the AdaTEC'82 Conference on Ada, ACM, Arlington, VA., October, 1982.

[SRS 79] IEEE Computer Society(editor).
Proceedings of the Conference on Specifications of Reliable Software.
IEEE Computer Society, 5855 Naples Plaza, Suite 301, Long Beach, California 90803, 1979.
IEEE Catalog No. 79 CH1401–9C.

[Standish 67] T.A.Standish.
A Data Definition Facility for Programming Languages.

PhD thesis, Carnegie–Mellon University, Department of Computer Science, 1967.

[Swan 77] R.J.Swan, S.H.Fuller and D.P.Siewiorek.
Cm*: A Modular, Multi–Microprocessor.
In Proc. 1977 National Computer Conference. American Federation of Information Processing Societies, 1977.

[Walker 80] B.J.Walker, R.A.Kemmerer and G.J.Popek.
Specification and Verification of the UCLA Security Kernel.
Comm. of the ACM 23(2), February, 1980.

[Wensley 78] J.H.Wensley, L.Lamport, M.W.Green, K.N.Levitt, P.M.Melliar–Smith, R.E.Shostak and C.B.Weinstock.
SIFT: Design and Analysis of a Fault–Tolerant Computer for Aircraft Control.
Proceedings of the IEEE 66(10):1240–1255, October, 1978.

[Wirth 71] N.Wirth.
Program Development by Stepwise Refinement.
Communications of the ACM 14(4), April, 1971.

[Wirth 77] N.Wirth.
Modula: A Language for Modular Programming.
Software––Practice and Experience 7(1), January, 1977.

[Wulf 72] W.A.Wulf and C.G.Bell.
C.mmp——A Multi-Mini-Processor.
In Proc. 1972 Fall Joint Computer Conference.
American Federation of Information Processing Societies, 1972.

[Wulf 73] W.A.Wulf and M.Shaw.
Global Variable Considered Harmful.
ACM SIGPLAN Notices 8, February, 1973.

[Wulf 76] W.A.Wulf, R.L.London and M.Shaw.
An Introduction to the Construction and Verification of Alphard Programs.
IEEE Transactions on Software Engineering SE-2(4), December, 1976.

[Wulf 81] W.A.Wulf, M.Shaw, P.N.Hilfinger and L.Flon.
Fundamental Structures of Computer Science.
Addison-Wesley, 1981.

[Yeh 80] R.T.Yeh and P.Zave.
Specifying Software Requirements.
Proceedings of the IEEE 68(9), September, 1980.