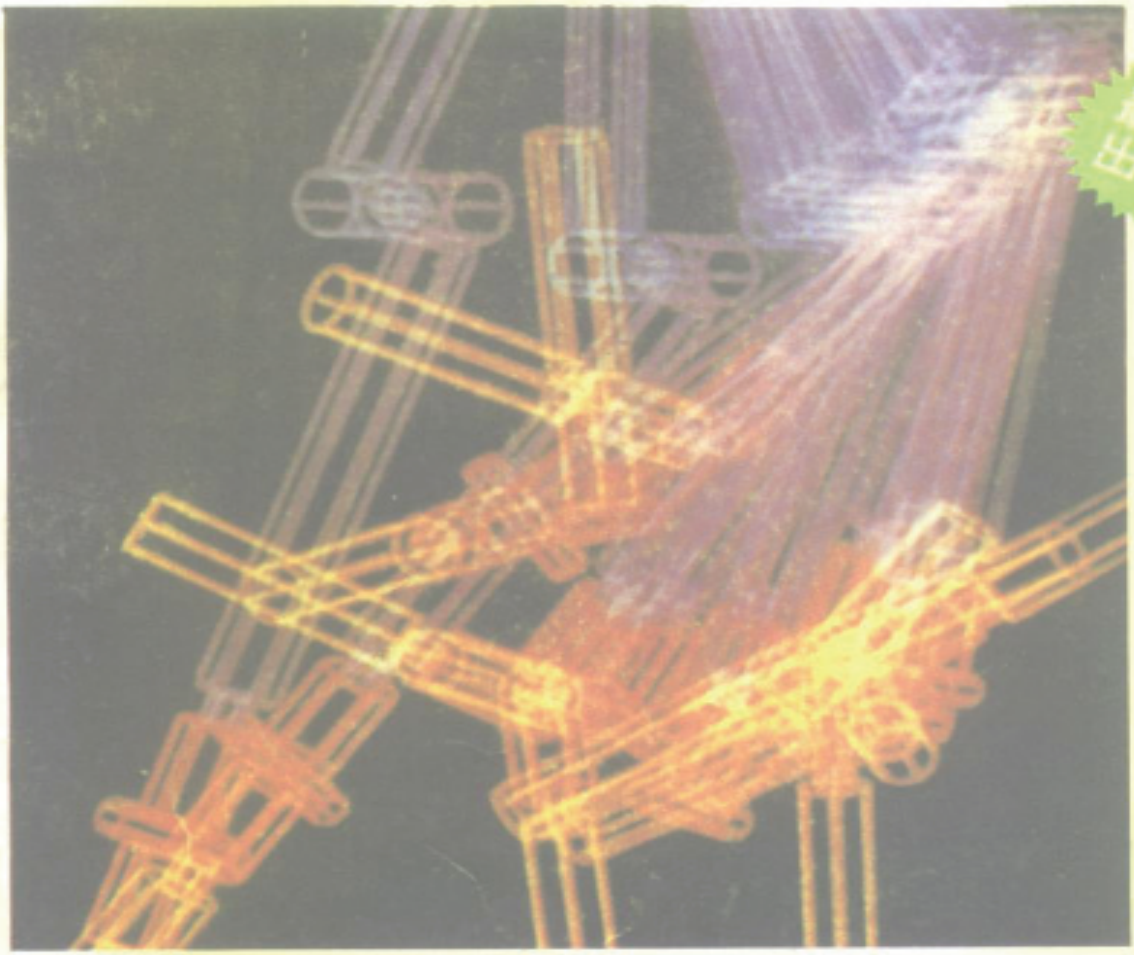
Ada

高级程序设计语言

陶辅周　纪希禹　李旭伟　编著



最新出版

電子工業出版社

Ada高级程序设计语言

陶辅周　纪希禹　李旭伟　编著

电子工业出版社

京(新)登字 055 号

内 容 简 介

本书主要讲述 Ada 程序设计语言、如何利用程序包组织大型程序和利用程序包隐藏信息，以及如何定义抽象数据类型，最后还介绍了分别编译、异常和文件。书中所述为最新 Ada 语言版本。

本书语言生动，内容深入浅出，可读性强。其读者对象可以是初学程序设计的高校学生，也可以作为具有一定 PASCAL、FORTRAN 语言程序设计经验的读者的 Ada 语言参考书。

Ada 高级程序设计语言

陶辅周　纪希禹　李旭伟　编著

责任编辑　秦　梅

电子工业出版社出版(北京市万寿路)

电子工业出版社发行　各地新华书店经售

北京顺义李史山胶印厂印刷

开本：787×1092 毫米 1/16 印张：10.75 字数：272 千字

1993 年 3 月第 1 版　1993 年 3 月第 1 次印刷

印数：6000册　　定价：8.00元

ISBN7-5053-1952-3/TP・477

序

本书旨在作为从事计算机应用工作者和高等院校学生学习计算机程序设计的一本入门教材，并且假定读者以前没有任何程序设计的知识；同时也试图为那些有一定 PASCAL 或 FORTRAN 语言程序设计经验的读者提供一本有益的 Ada 语言参考书。

目前，PASCAL 语言是较流行的语言。事实上，它最初是为教学目的而设计的，如果不加以适当地扩充是不能用其编制大型软件的。可靠性是现代计算机语言的重要特征，Ada 语言中的程序包概念是研制可靠程序的有效工具，而 PASCAL 语言就没有程序包概念，可以说程序包的概念是 Ada 语言对程序设计语言的最大贡献。

软件的可靠性及研制费用的增长曾导致了一场"软件危机"，为此，美国政府在 1975 年起草了一份需求草案，并希望这些要求能体现在某一语言中。然而，在当时还没有一种语言能满足所提出的要求，于是决定研制一种新的语言。在这场国际招标中，由 J．I Chbiah 所领导的 C ll Honeywll Bull 的研究小组中标获胜。将这种新语言命名为 Ada 是为了纪念 Augusta，她被公认为是最早的程序员。

最初的 Ada 语言参考手册于 1980 年问世，自那时起，Ada 语言已经过数次修改，本书所述的是最新的版本。国内也于最近完成了汉化 C-Ada 的研制，从而推动了 Ada 语言在国内的普及。

Ada 是一个相当大型的语言，本书不可能面面俱到，关于类型和对象的描述已被简化，但这并不影响 Ada 语言的体系。本书的后半部分介绍了如何利用程序包组织大型程序和利用程序包隐藏信息，以及如何定义抽象数据类型。最后还介绍了分别编译、异常和文件。

本书的写作过程中，始终得到中科院数学所徐泽同研究员的热心支持，在此表示衷心感谢！

编者

1991 年 2 月于四川大学

目　录

第一章　怎样用计算机解题

§1-1　引　　言

本书介绍怎样用计算机解题。主要描述怎样系统地设计算法以及怎样在计算机上实现这些算法。进行这一系列工作的最终目的是编制计算机程序，编制的程序应具有易理解、简洁、可靠、效率高等特点。

Ada 程序设计语言能够适应上面的要求，是一种比较好的语言。它广泛应用于国防部门，日益受到用户的喜爱。由于它的设计严谨、合理，弥补了其他语言的不足，因此越来越受到社会各个领域计算机工作者的欢迎。为了跟上时代的步伐，从读者的愿望出发，在本书中我们采用通俗易懂的语言，较系统地介绍 Ada 语言的程序设计方法。即使以前没学过任何程序设计语言的读者，也可顺利地阅读本书。

目前，计算机已深入社会生活的各个领域，以致普通的人对计算机也略知一二。为了使读者对计算机的内部结构和原理有一个较清晰的认识，我们首先给出一个简单的定义——计算机是一个能接受指令并执行指令序列的电子装置。

为了使计算机完成某一任务，必须首先给计算机输入明确的指令，这些指令必须是由某种计算机能理解的语句写成。这些指令集就称为计算机程序。

当用计算机解题时，不仅仅是坐下来写一个程序而已。首先必须明确题意，然后再设计相应的算法，最后才开始编制程序。因此，程序设计不仅仅是编制程序，明确题意和算法设计比编制程序更为复杂。

在介绍程序设计之前，首先简介计算机的结构。

§1-2　关于计算机的一般知识

从个人微机到巨型计算机，尽管它们在体积、性能、价格上的差异可能很大，但它们都是由相同的基本器件构成的。

1. 中央处理单元(CPU)

CPU 是实际执行指令的单元，每条指令是相当简单的。计算机的性能由执行指令的速度确定，一般每秒钟可完成几万次至几千万次的操作。对于计算机来说，用秒作时间单位已显得过大，一般都用毫秒(10^{-3} 秒)、微秒(10^{-6} 秒)、毫微秒(10^{-9} 秒)。运行速度快是计算机的一大特点，计算机迅速地进行一系列简单的操作就可以完成很复杂的任务。

2. 主存储器

主存储器用于存放程序指令和指令所需的数据。计算机在执行指令前，这些指令须事先装入计算机的主存储器中。当计算机执行这些指令时，就称作运行或执行程序。

3. 后备存储器

通常对不需立即调用的程序和数据可以存储在所谓的后备存储器中。后备存储器可以是磁带或磁盘。一般它所能存储的信息要比主存储器多得多，因此可用它长期保存程序和数据。后备存储器可被视为一个大型文件编排系统，它把程序之类的信息归档为单个文件。程序被运行前，必须事先把程序指令从后备存储器的文件中拷贝后调入主存储器中。

4. 输入、输出设备

输入设备用于把信息读入计算机，而输出设备则用于输出程序运行的结果。最常用的设备是交互式终端，它能同时用作输入、输出设备。交互式终端通常由一个显示器（相当于家用电视机的屏幕）和一个打字机键盘组成，一般把它称为直观显示部件（VDU）。信息由键盘敲入，键入的信息以及程序的运行结果都显示在屏幕上。用这样的方式，用户就可和计算机进行"对话"。打印机是另一种输出设备，它可以把程序或结果打印出来以供进一步研究。

主存储器、CPU、后备存储器、输入输出设备都被称为计算机硬件，而程序被称为计算机软件。如果只有硬件，则计算机尤如一个白痴，是干不了什么事情的；相反，没有硬件，则"皮之不存，毛将焉附"，软件的功能也就无从谈起。

§1-3 计算机软件

当使用计算机时，总会发现机器中事先已装入了一系列的程序，这些程序的功能是对用户提供帮助。这个程序集叫做计算机操作系统。用户所使用的计算机实际上是硬件和操作系统的结合体。我们可以简单地把这两部分视为一个整体而不必关心它们的具体分工。以后将称硬件与软件的组合体为计算机系统。

至此，尚有一重要的问题没有考虑，即每台计算机仅能理解一种语言——即它的机器语言。由于机器语言很难理解，用它来编写程序很不方便，故代之以特殊设计的、易于使用的语言来编写。这些语言被称为高级程序设计语言，如 FORTRN、COBOL、C 语言、PASCAL 和 Ada 等等。

然而，计算机是不能理解这些高级语言的。为使计算机能理解采用高级语言编制的程序，必须首先将其翻译成机器语言。因此，除了操作系统外，还需有特殊的翻译程序，即编译程序。如果想在计算机上运行一个用 FORTRAN 或 PASCAL 编制的程序，就需要有相应的 FORTRAN 或 PASCAL 编译程序，同样，若要运行一个用 Ada 编写的程序就要有 Ada 编译程序。总之，对于每一种高级语言都要有相应的编译程序。

由于需要编译程序，使用高级语言显得很麻烦，但事实上却有其很大的优越性。例如，在一台计算机中配置有 Ada 编译程序，就可以运行一个由 Ada 语言写成的程序。因此人们通常将 Ada 等这类高级语言称为独立于机器的语言。但在实际应用中，语言并不完全独立于机器。因为大部分程序设计语言存在不同的版本，不同的机器性能也不一样，这就可能造成为一类机器设计的程序在另一类机器上运行不了的问题。针对可能出现的这些问题，Ada 语言的设计者通过对语言的严格定义以及避免"方言"等手段做了处理，通过这种处理，Ada 语言真正地和机器相互独立开来了。

使用高级语言编写程序还有另一个优越之处：能够及早发现程序中的一些错误以供程序员修改。程序设计语言和自然语言一样，也有其语法规则，这些规则规定了什么是允许的以及什么是不允许的。如果在编写程序时违犯了语言规则，则在从高级语言转换成机器语言的过程中，编译程序会发现并指出这些错误并通知程序员修改。这样，程序员就可以纠正错误然后再进行编译调试。

由高级语言转换成机器语言的过程如图 1.1 所示。

注意，计算机实际执行的是 Ada 程序的目标程序（由机器语言写成），而不是源程序自身。运行

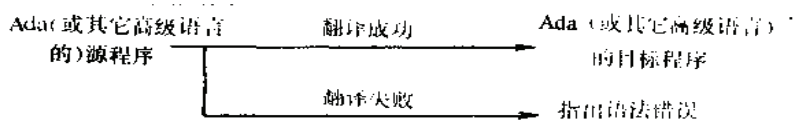


图 1.1

期间，可能会发现运行时的错误。当计算机系统发现这个错误后就会中断运行，然后显示提示信息以告之程序员。显示的信息是以 Ada 源程序的形式表示的，懂 Ada 的程序员就可以理解这些信息并对源程序进行适当的修改。

有时，某一个程序运行通过之后，并不能保证其运行结果的正确性。这是因为所设计的算法中可能会有逻辑错误，故精心选择测试数据来调试程序是相当重要的。用这种方法来检查和纠正错误称为调试(debugging)。

或许读者已看出，检查纠正编译时的错误比运行时的错误容易；而检查和纠正运行时的错误比发现和改正逻辑错误容易。Ada 语言的设计使得许多错误尽可能在其编译的过程中被检查出来。当然，逻辑错误仍将会引起运行错误，这就需要通过调试手段来测试纠正。

§1-4 逐步求精程序设计方法

下面主要介绍逐步求精的程序设计方法。在编制程序之前，必须事先进行几个重要的步骤。

一、明确题意

对于一个复杂的问题，明确题意是最关键的，并且也是整个过程中最难的步骤。这个步骤中所出现的任何错误将会导致付出昂贵的代价。因为这会导致解决一个不正确的问题而误入歧途。牢记这个步骤是非常重要的，特别是在处理商业、工业以及科学研究中的各类问题。

二、设计算法

如何设计算法是我们所感兴趣的问题。算法可以定义为：一个明确的指令序列，执行这个序列，可在有限的时间里得出所需的结果。

下面，我们通过一个简单的实例来看一下如何设计算法。

例 1-1 求 1 至 99 之间所有奇数的平方和。

首先考虑如何用笔和纸解这道题：依次取一个奇数并计算其平方；然后将其加到某个累加器中。于是产生计算机解题的雏形。这个解题步骤可以写成：

置累加器之值为 0

取 1 至 99 之间的奇数，将其平方值加到累计器中

写出累计器的终值

这个算法是相当粗的，没有涉及怎样生成奇数序列。实际应用时还需要对其精细化，用以指明如何依次产生奇数。我们可以从第一个奇数开始，在当前奇数上加 2 其值总是奇数。由此该算法被扩展成：

```
置累计器之值为 0
置奇数为 1
循环，当奇数小于或等于 99 时
    将当前奇数之平方加至累计器
    当前奇数加 2
```

结束循环

写出累计器之终值

对于 1 至 99 中的每个奇数，都要执行介于“循环”与“结束循环”之间的步骤。

这种解题方法称为逐步求精算法设计方法。首先产生一个粗略的算法，它不涉及一些内容细节。然后用逐步求精的方法将其进行精细，直到可以用某种程序设计语言编程为止。算法以及它的精细是用文字表述的，它与某一特定的程序设计语言毫无关系。

由于本题是相当简单的，现在就可以编制计算机程序了。当算法设计完成后，用 Ada 或者其它语言编制程序就比较容易了。所以脑力劳动主要花在算法设计而不是用在编制程序上。

下面给出了上述算法相应的 Ada 程序，读者可以看到一个完整 Ada 程序。这个 Ada 程序与用文字描述的算法是很相似的，甚至对 Ada 一无所知的读者也会理解。

程序1-1

```
WITH student_io; USE student_io;
PROCEDURE odd_square IS
    sum_odd: integer := 0;
    odd_number: integer := 1;
BEGIN
    —— 求 1 到 99 范围内奇数的平方和
WHILE odd_number <= 99 LOOP
    sum_odd := sum_odd + odd_number * odd_number;
    odd_number := odd_number + 2;
    ——sum_odd 中存入了奇数平方的和
END LOOP;
    put("The sum of the odd number is ");
    put(sum_odd); new_line;
END odd_square;
```

至此，还有一个重要问题：如何将已编制好的 Ada 程序输入计算机呢？答案是依赖于你所使用的计算机系统。

要想把程序输入计算机需要借助于操作系统。操作系统中有一个编辑程序，利用它可以把新的信息（如程序）输入计算机，也可以修改或编辑已输入计算机的信息。在编辑程序的控制下，可在终端键入 Ada 源程序。当 Ada 源程序被输入计算机后，编辑程序就可以将它拷贝至后备存储器的一个文件中。

如何具体使用编辑程序与所使用的计算机系统有关，这已超出本书的范围，在此不再赘述。

§1-5 大型程序的组织

计算机已广泛应用于社会经济各个领域，如控制原子能电站、管理银行账务等。由于计算机在这些应用中所承担工作的复杂性，使得为这些系统所编制的程序也相当复杂、庞大。因此，程序是否可靠、运行是否正确是至关重要的。

六十年代末期国际上出现了“软件危机”，其主要表现是：软件质量差，可靠性难以保证，软件成本增长难于控制，极少有在预定的成本预算内完成的；软件开发进度难于控制，周期拖得很长；软件的维护很困难，以使维护人员和费用不断增加。有的软件耗费了大量的人力财力，结果半途而废。

考虑到研制一个大型而复杂的软件系统同研制一台机器或建造一座楼房这样的工程有许多相

似之处，因此，人们就试图用“工程化”的思想作指导来解决软件研制中面临的困难和混乱，从根本上解决软件危机。这样就产生一个新的学科——软件工程。软件工程就是研究如何生产高质量、低成本、可靠性好的软件产品的一门学科。

在学习程序设计时，总是通过编写小型程序来提高编程技巧。一旦程序运行正确后，就将其搁置一边，需要时就来调用。在研制大型软件时，庞大的程序往往是由程序员小组共同协作完成，而且所编制的程序可能要使用许多年。在使用期间，根据系统的要求修改程序也是经常的事，这些修改通常也由原来编制程序的不同程序员分别进行。

解决设计大型系统的最好办法是将其分解为若干相对独立的子问题，这些子问题可以独立解决。然后进一步将每个子问题再分解成若干较简单的子问题；依照这种办法继续下去，直到每个子问题足够简单。这也就是上节所说的逐步求精方法，只是从不同角度来看而已。本书的前半部分将介绍如何组织小型程序，只包括 Ada 语言的基本特征，其它部分留待后面介绍。

在介绍 Ada 语言的基本特征之后，引入了 Ada 程序包的概念。程序包概念是 Ada 对程序设计语言研制所做的最大贡献，是支持大型软件系统的有力设施。大型系统允许对子问题设计程序包，程序已可与程序的其余部分分开来单独实现与测试。当所有的子问题都得以解决并运行正确之后，就可将它们连接起来形成最终程序。对于最终程序，也可以对程序的一部分进行修改和调试而不影响整个程序。

当某一问题分成若干子问题后，或许会发现某个子问题已有现成的程序包。这样，只要直接利用这些程序包即可。大部分 Ada 程序的实现都有这样一些有用的程序包供用户调用，这就使我们研制大型软件系统时不必从头开始，省时省力。

第二章　简单的 Ada 源程序

§2-1　一个完整的 Ada 程序

首先仔细考虑如下一个 Ada 程序的结构，该程序将计算并输出数 94 和 127 的和。

程序 2-1

```
WITH student_io; USE student_io;
PROCEDURE add IS
    first, second, sum: integer;
BEGIN
    --program to print the sum of two integers
    (打印两整数之和)
    first := 94;
    second := 127;
    sum := first + second;
    put("sum =");
    put(sum);
    new_line;
END add;
```

下面对程序 2-1 逐行解释。

第一行：

```
WITH student_io; USE student_io;
```

就是所谓的上下文子句(context clause)，这是一个非常有用的语句，我们暂时承认它，其意义将在以后说明。

程序中的大写单词，如 WITH、USE、IS、PROCEDURE、BEGIN 和 END，这些词称为保留字(reserved word)。保留字是 Ada 语言的组成部分。为了使程序结构清晰，保留字采用大写形式，但对 Ada 而言，任何一种字型都是一样的。其它的保留字将在以后介绍，附录一给出了 Ada 语言的所有保留字。

一个简单 Ada 程序包含一个过程。程序 2-1 的第二行：

```
PROCEDURE add IS
```

引入名为 add 的过程。过程名是由程序员确定的，但必需养成好的习惯，就是名称的选择有助于了解程序的内容。在程序中，名字由标识符表示。

第三行：

```
first, second ,sum: integer;
```

称为过程的说明部分。它说明了三个变量对象并分别取名为 first、second、sum，以后就可以调用这三个变量。

在本程序中，变量对象或变量是用来存放值的，程序 2-1 中的三个变量说明为整数类型，也就

是说，变量对象 first、second、sum 只能用来存放整数。之所以把它们称为变量，是因在程序的执行过程中，它们所存储的值可以变化多次。

一个变量对象限制只能存放一特殊类型的值。程序 2-1 中的三个变量说明为整数类型，也就是说，变量 first、second、sum 只用来存放整数，我们称之为整数变量。

一个变量的类型确定了其所存放值的范围以及可以实施的运算。Ada 有一些内部类型，例如 integer 和 float(整数类型和浮点类型)。整数变量可进行加、减、乘、除等算术运算。一个变量必须有一个名字、一个值和类型。一旦一个变量被说明后，其名字和类型就被固定了，但其值在程序的执行过程中可以改变。

下面看一下程序的执行情况。运行程序时，依次执行 BEGIN 与 END 之间的指令语句。

BEGIN 之后的第一条语句为解释语句：

```
--program to print the sum of two integers
```

注释以两个连续的连字号作为开始，并以该行的结尾作为结束。注释仅仅是为了帮助阅读程序，执行程序时，计算机将不作任何处理。

注释语句之后是两个赋值语句：

```
first := 94;
second := 127;
```

即把右边的数值储存在左边的变量中。复合符号":="应读成"取值"。执行这二个语句的结果是，整数 94 和 127 分别被放在变量 first 与 second 中，我们说变量 first 和 second 分别被赋值为 94 和 127。

接着再执行下一语句：

```
sum := first+second;
```

首先把变量 first 和 second 的值相加，所得结果放在变量 sum 中。

语句"put("sum=");"为输出语句，输出"sum="字样。在程序中，用引号括起来的字符序列称为串(string)。

另一个输出语句：

```
put(sum);
```

输出 sum 的值。注意，这两个输出语句是不相同的。在第一个输出语句中，因为我们希望字符序列按原样输出，所以用双引号括起来；而在第二个输出语句中，则输出变量的值。执行这二个输出语句的结果是：

```
sum=221
```

连续的输出语句将所有信息显示在同一行上，如果需要另起一行则可用语句："new_line;"。上述语句已是语句序列的最后一个语句，程序运行到此为止。

在程序的最后一行中，过程名 add 又在 END 之后出现，这在 Ada 程序中并非是必要的。但我们提倡这样做，因为这样可以帮助人们阅读理解程序。

由上述分析可知，一个简单的 Ada 程序由两部分构成。在保留词 IS 与 BEGIN 之间是说明部分，它对程序中所使用的标识符进行说明；BEGIN 与 END 之间是语句序列，每个语句以分号";"作为分隔符，执行程序时，按语句的先后次序执行。稍加分析可以发现，程序 2-1 只能把数 94 与 127 相加，如果要计算另外两个整数的和就要修改源程序，因此不具有灵活性。我们将对程序 2-1 加以修改，增加两个输入语句，这样就可以把任何两个整数相加。

修改程序 2-1，它将读入两个整数并输出两数之和，程序如下。

程序 2-2

```
WITH student_io; USE student_io;
PROCEDURE add_any IS
    first,second, sum: integer;
BEGIN
    --program to read two integers and to print their sum
    (读入两个整数并输出其和)
    put("two integer numbers please");
    get (first); get (second);
sum := first+second;
    put("sum=");
    put(sum);
    new_line;
END add_any;
```

程序 2-2 使用户与程序的执行联系在一起，用户必须输入信息计算机才能计算并输出结果，以后我们总是在交互式终端上运行程序。

考察程序 2-2，三个整数变量 first、second、sum 再次被说明。第一个要执行的语句是：

```
put("two integer numbers please");
```

执行结果是打印出信息：

```
two integer number please
```

其作用是提示用户输入两个整数，因为紧接着的是两个输入语句，它要求输入两个整数作为数据。当键入两个整数后，这两个整数就被程序读入并分别存放在变量 first 和 second 中。接下来的语句与程序 2-1 相同，就不再描述了。

§2-2 注释与源程序的书写格式

在编程序时必须牢记：程序不仅仅被计算机读入和理解，而更重要的是被人阅读和理解。所以，我们总是选择富有意义的标识符，并且精心编排程序以使其结构清晰。例如程序 2-1、2-2，介于 BEGIN 与 END 之间的语句序列是采用分层缩进对齐的写法，这样有助于了解程序的结构。本书的示例可以作为编写 Ada 程序的示范。

程序的格式是不影响其意义的。标识符、保留字及数之间至少要用一个空格或空行将它们彼此分开。在允许使用一个空格的地方，总可以留几个空格或另起一行。

注释是用来帮助理解程序的，它以两个连续的连字号为开始并以该行末尾作为结束。尽管注释并不影响程序的内容，但对于阅读程序的人来说则带来极大的方便。

一般地，在过程开始处加一注释以描述该过程的内容或目的，程序 2-1 中的注释就是这样的。其它的注释可以放在程序的难点或关键处（本书中的注释都采用英文，适当用中文解释）。

§2-3 语法图

在详细介绍 Ada 语句之前，我们先用语法图这种简单的概念来描述 Ada 语言。语法图对于学习 Ada 语言是极有帮助的。

和自然语言一样，Ada 语言也有其自身的语法规则。语法规则规定了什么是合法的、什么是非

2

法的程序，这些规则可以用文字来描述。

例如，对于已提及的标识符尚未给出严格的定义，可以用如下的一段文字来描述标识符的语法规则：

标识符由字母开头，其后可跟或不跟其它字母或数字，一个下划线也可以出现在任何两个字母或数字之间。下面是一些合法标识符的例子：

x，first _ Case，Julia，Chapter _ 2

用文字来描述语法规则显得很长而且可能语义不清，采用语法图描述语法规则是较好的方法。标识符可用如下的语法图描述(图 2.1)。

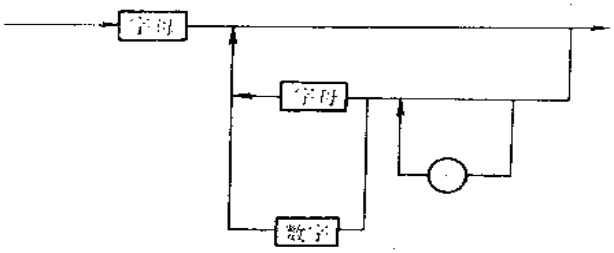


图 2.1

图 2.1 中所有可能的路径都构成合法的 Ada 标识符。语法图是一个等价的语法规则定义，它具有简洁明了的特点。

在语法图中，圈或弧矩形中的内容(如下划线)表示出现在 Ada 程序中的实际字符；而矩形框中描述了出现在 Ada 程序中的条目。矩形中的项必须用其它语法图加以定义。例如数字可定义为图 2.2 的形式。

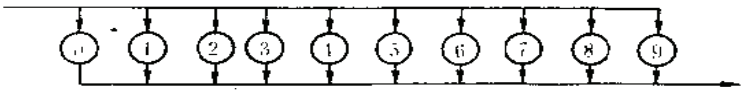


图 2.2

这表明数字是数 0～9 中的一个。

如果两个标识符仅仅是字母大小写的区别，则被视为同一标识符。因此 Same、same、SAME、sAME 是同一标识符 same。

最后，看一下简单 Ada 程序的结构语法图(如图 2.3 所示)。

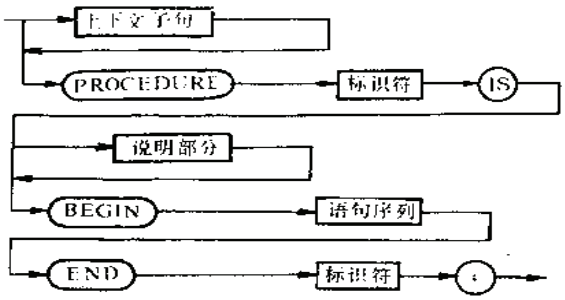


图 2.3

款项"上下文子句"、"标识符"、说明部分"及"语句序列"都在矩形框中，它们需要用语法图进一步定义；而 procedure、is、begin、end 都在圆弧框中，它们将直接出现在程序中。注意，保留字不能作为通常的标识符使用。

练习

1. 指出下列哪些是合法的 Ada 标识符。

   total　Total sum _ 1 sum 1　Summ _ 1　sum _ 1. 5　SECOND　begin　average　"fonal"

   Total　count　2nd　BEGIN　END

2. 习题 1 中哪些是已学过的保留字？

3. 在程序 2-2 中，用户在提示：

   two integer numbers please

   下输入：

   15　28

   请写出运行结果。

4. 为什么说程序的格式以及在程序中使用注释是很重要的？

5. 编一程序，要求读入两个整数，输出第一个数减第二个数的结果。

第三章　值与类型

本章主要介绍变量对象的说明以及整数类型、浮点类型、实数类型和枚举类型、布尔类型等。

§3-1　变量对象的说明

在第二章中，我们已使用说明：

```
first, second, sum: integer;
```

来说明三个名为 first、second、sum 的整数变量。一个变量必须有一个名字和类型，变量是用来存放值的。变量对象由说明生成并确定变量对象的名字及其类型。

变量的类型确定了该变量可存放的值的范围以及可以施行的运算。在上述例子中，first、sum 被说明为预定义类型——整数类型。

在程序 2-1 和程序 2-2 的说明部分中都只有一个说明，但说明可以不止一个，甚至可以把上述说明分成三部分。例如：

```
first:integer;
second:integer;
sum:integer;
```

一旦对一个变量定义了名字和类型，在说明它时就可以赋于一个初值。例如：

```
table_size:integer := 20;
maximum, minimum:integer := 0;
```

如果一个变量在说明时未赋初值，则该变量的值是未定义的(undefined)，直到用赋值语句或输入语句对其赋值。企图调用一个未定义值的变量是没有意义的并将引发一个异常

§3-2　常量对象的说明

某一常量可能在程序中多次用到，例如，在统计某班学生考试成绩时，班级人数 100，或许会不止一次地要用到。如果某个学生成绩也是 100，这个值就会对阅读程序的人带来麻烦，不能确定它们所指的是否为同一件东西。同变量对象一样，在 Ada 程序中可以说明一个常量对象。例如：

```
class_size: CONSTANT integer := 100;
```

定义了一个称为 class_size 的常量整数对象，并赋值为 100。

现在，可以使用常量标识符 class_size 替代数 100，用其来表示班级人数。如果碰巧某学生的成绩为 100 分，则这个分数 100 是不能用 class_size 代替的。常量对象不同于变量对象，常量对象必须在说明时赋于一个初值，并且这个值在以后的程序中是不可以改变的。

使用常量标识符有许多好处，如果要统计另一个班级的学生成绩，假设学生人数为 120，只需把常量说明改为：

```
class_size:CONSTANT integer := 120;
```

在编制程序时，对重要的常量使用有意义的标识符有助于理解程序。这或许是使用常量标识符

的一个重要原因。例如，如果要用到一周的天数，使用如下的常量标识符说明要比简单使用数 7 要好得多。

```
days_in_week: CONSTANT integer := 7;
```

例 3-1 下面的一个程序用来计算职工的工资。其中有两个常量标识符，一个表示通常工作时间每小时的工资额；另一个表示加班时间每小时的工资额。此外需要输入两个整数，一个表示职工每周的总工作时数，第二个表示加班时数。利用这些数据就可以计算并输出某职工的工资。

程序 3-1

```
WITH student_io; USE student_io;
PROCEDURE wages IS
    total_hours, overtime, normal: integer; --hours worked (工作量)
    pay, basic_pay, overtime_pay: integer;
    rate_of_pay: CONSTANT integer := 8; --hourly $ rate(每小时酬金)
    overtime_rate: CONSTANT integer := 12;
BEGIN
    -- program to read hours and calculate pay
    (输入工作时间，计算报酬)
    put("Total number of hours worked is");
    get(total_hours);
    put("Number of hours of overtime is ")
    get(overtime);
    --Calculate number of hours worked at normal rate
    normal := total_hours-overtime;
    --calculate pay
    basic_pay := normal * rate_of_pay;
    overtime_pay := overtime * overtime_rate;
    pay := basic_pay+overtime_pay;
    put("pay= $ ");
    put(pay);new_line;
END wages;
```

符号“*”是 Ada 程序中用来表示乘法的。在程序中，共有 6 个整数变量、两个整数常量被说明。可以发现，常量标识符的使用使得程序便易于阅读和理解。

程序执行时，首先在用户终端上显示提示：

```
Total number of hours worked is
```

程序的执行暂时中止，直到在终端上键入一个整数，例如“38”。程序接着读入该数并将其存储在变量 total_hours 中。紧接着又显示提示：

```
Number of hours of overtime is
```

再次中断运行，直到输入第二个整数，例如“3”。整数 3 被存在变量 overtime 中。接下来，程序就计算工作时间以及职工在两种时间里的工资总和，并显示计算结果如下：

```
PAY= $          316
```

注意，如果需要增加工资，只需修改常量说明，其它部分不需改动。

§3-3 整数类型

3.3.1 整数的表示

在 Ada 语言中，整数值或常量整数可以用十进制整数表示；一个十进制整数含有一个数字序列，其后面可接一个指数部分。例如：

```
56    0    543    10    2E6    200000    200    000
```

指数表示 10 的幂，它同其前面的整数相乘就得到该整数的值。因此，2e6 与 2e+6 都表示值 2×10^6，即 200 万。在数字序列中，可以在相邻的数字中插入下划线而不影响其值，下划线通常用来分隔较长的数以便于阅读。

一个十进制数的语法图如图 3.1 所示。

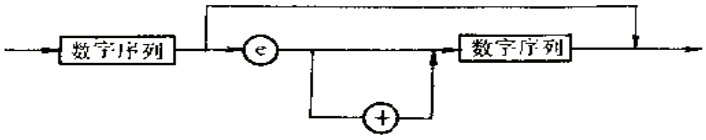


图 3.1

指数可以用大写"E"，也可以用小写"e"。

3.3.2 整数输出

整数在计算机中总是精确表示的，所允许的最小值和最大值与实现的方法有关。当输出一个整数时，需有足够的空格来保证输出最大的整数，并且至少加一个前导空格。对于位数少的整数则在其前面补足空格。若是负整数，则在其前加一负号。

在前面的例子中，已假定允许的最大整数是 10 位数。因此程序 3-1 中的语句

```
put("pay = $ "); put(pay);
```

将打印如下内容：

```
pay = $          316
```

可以发现用这种方式表示结果是不理想的。为了使 316 紧挨着"$"，可以使用宽度参数。例如：

```
put("$ "); put(pay, width=>4);
put("$ "); put(pay, width=>3);
```

上述两个语句分别显示如下结果：

```
$  316
$  316
```

如果所设置的宽度参数小于输出数的位数，程序会自动增加所需的数位来显示所有的数。事实上，通常使用宽度参数 1 来保证所输出的整数之前没有多余的空格。例如：

```
put("$ "); put(pay,width=>1);
```

将打印出

```
$316
```

§3-4 浮点类型与实数类型

浮点类型变量可用于存储带小数的实数。在 Ada 中，可以用十进制实数来表示小数。实数不同于整数，它含有一个小数点。例如：

576.8　　297.0　　1.0e－6　　6.023e23　　3.141_592_653

在实数中允许负的指数，小数的两边都必须有数字。例如 1.0e－6 表示“1×10－6”，即 0.000001。对于较长的数，可插入适当的下划线以便于阅读。

表示实数的语法图见图 3.2。

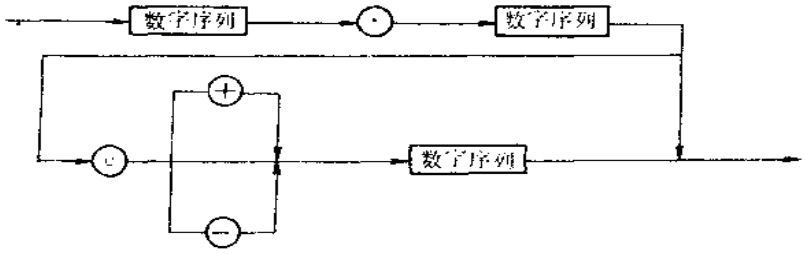


图 3.2

注意，2 表示整数；而 2.0 表示实数。这两个数在计算机中存储的方式是不一样的。通常，实数是近似表示的，实数之间的运算所得的结果也是近似的。而整数间的运算结果是精确的，这是整数和实数的一个重要区别。

一个数值的相应精度可以放在浮点类型变量中，浮点变量可存储的最大、最小值依赖于所使用的 Ada 语言的实现方式。与实数不同的是，我们可以说明浮点类型的相应精度。在如下的类型说明中说明了所需的最小的有效十进小数位：

```
TYPE real IS DIGITS 8;
```

它产生一个新的浮点类型 real，其值的类型为实数、有效数字至少为 8 位。如果用变量说明：

```
sum:float;
```

则 sum 之值的相应精度就依赖于你所使用的计算机。但是如果有变量说明

```
total:real;
```

则不管使用何种计算机，total 的值都具有 8 位有效数字。

在本书中，总假定 real 类型可以通过访问 student_io 获得，并通过这种类型来处理带有小数的值的运算。

以下是一些实数常量或实数变量对象说明的示例：

```
radius, circumference: real;
frequency, percentage: real := 0.0;
AVogadro: CONSTANT real := 6.023e23;
pi: CONSTANT real := 3.141_592_7;
```

可以发现，实数变量或实数常量对象说明类似于整数变量或整数常量对象的说明。

例 3.2　已知圆的半径计算其周长和面积。编制程序时，要求圆的半径由程序读入，然后输出相应的周长和面积。程序如下：

程序 3-2

```
WITH student_io; USE student_io;
PEOCEDURE circle IS
    radius, area, circumference: real;
    pi: CONSTANT real := 3.141_592_7;
BEGIN
    --read radius and calculate circumference and area
    (输入半径,计算周长和面积)
    put("Please radius?");
    get(radius);
    circumference := 2.0 * pi * radius;
    area := pi * radius * radius;
    put("Circumference=");
    put(circumference); new_line;
    put("Area=");
    put(area); new_line;
END circle;
```

程序 3-2 的说明部分里,共说明了三个实数变量和一个实数常量.不难看到采用常量标识符的优越性。执行时首先显示提示:

Please radius?

然后由用户在终端输入所需的值,该值被存放在变量 radius 中。接着就计算圆的周长与面积并将显示结果。

§3-5 字符类型与串类型

除了科学计算之外,计算机还用于处理大量的非数值的信息,而这些信息通常包含处理字符或字符串。在此之前,我们已经在程序中使用了字符串来输出提示或说明信息。本节将介绍字符和串两种类型以及相应的变量说明。

3.5.1 字符

单个字符可存放在字符(character)类型变量中。在 Ada 程序中,字符值是用单引号括起来的字符表示的。例如:

'a' 'A' '+' ',' ''' ':' ' ' '6'

一个空格用两个单引号括起来就表示一个空格字符,空格字符是一特殊字符。用同样的方法可以表示其余的任何字符。

在 Ada 语言中,字符类型取值于 ASCII 字符集中的 128 个字符,其中 95 个是可打印字符,其余的则是特殊的控制字符。只有这 95 个可打印字符才能出现在字符记号中,也就是说只有这 95 个可打印字符才可以存放于字符变量中。可打印字符又分大、小写字母,数字以及最常用的标点符号和算术运算符(+、-、*)等。

例 3-3 读入任何三个字符,首先三个紧挨着输出;然后在字符之间插入空格输出。

程序 3-3

```
WITH student_io; USE student_io;
PROCEDURE three_chars IS
```

```
    char_1, char_2, char_3: character;
    space: CONSTANT character := ' ';
BEGIN
    --read and print three characters
    (读入并打印三个字符)
    put("Please input three characters");
    get(char_1); get(char_2); get(char_3);
    --write out the characters
    (输入字符)
    put(char_1); put(char_2); put(char_3);
    new_line;
    --write again, but include spaces
    (再输出一次,但含有空隔)
    put(char_1); put(space); put(char_2); put(space);
    put(char_3); new_line;
END three_chars;
```

假设输入字符 a,b,c,则输出情况如下：

```
abc
a b c
```

3.5.2 串

字符串是指由一个或更多字符用双引号引起来的字符序列。在一个字符串中，许多字符可以出现在一起，但也允许一个串是空的。例如：

```
"This is a string"
```

同 integer、float、character 一样，string 也是一种内部类型，但与它们有明显区别。string 类型是由简单的类型 character 组成的，因此也称 string 类型为复合类型，称类型 integer、float、real、character 为标量类型。

任何一个可打印的 ASCII 码都可以出现在串中，由于双引号是用于限制串的终止的，所以双引号不能单独出现在串中。如果一定要用双引号，则采用两个双引号就可以解决这个问题。例如执行语句：

```
put("The variable""sum""has been declared.")
```

就可以输出：

```
The variable"sum "has been declared.
```

下面先介绍串常量，再介绍串变量。例如说明：

```
greetings: CONSTANT string := "Hello!";
```

就说明了一个串常量 greetings，其值为"Hello!"。

在一个程序中，某一长串需要多次用到，设置一个常量串标识符是很有益的。

例 3-4 打印一个由"*"号构成的三角形，在三角形的上下两侧各有两条也由"*"号组成的横条。即：

```
* * * * * * * * * * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
                    * * *
                  * * * * * *
          * * * * * * * * * * * * * * * * * * * * * * * * *
          * * * * * * * * * * * * * * * * * * * * * * * * *
```

程序 3-4

```
WITH student_io; USE student_io;
PROCEDURE triangle IS
    border:CONSTANT string := "* * * * * * * * * * * * * * * * * * * * * * * * * * * * *";
    spaces:CONSTANT string := "                    ";
BEGIN
    --print a bordered triangle
    (打印带边界的三角形)
    put(border);new_line;
    put(border);new_line;
    new_line;--porduce a blank line (产生一空行)
    put(spaces); put("* *");new_line;
    put(spaces); put("* * * *"); new_line;
    put(spaces); put("* * * * * *"); new_line;
    new_line; --another blank line (另一空行)
    put (border); new_line;
    put(border); new_line;
END triangle;
```

在程序中，一个完整的串必须写在同一行中，如果一行写不下，可使用连接运算符"&"。如：

```
"The first part of the string"&
"and the rest of the string "
```

尽管分成了两部分，但被认为是一个串：

```
"The first part of the string and the rest of the string"
```

§3-6 枚举类型与布尔类型

3.6.1 布尔类型

在计算机程序中经常需要进行逻辑判断，因此 Ada 语言专门用 boolean(布尔)类型来处理逻辑判断。布尔值只有两个，在 Ada 程序中是用标识符 false、true 来表示的，其含义分别表示"假"和"真"。因此可有以下的说明：

```
flag: boolean := false
success, failure : boolean;
found, finish:boolean := true;
```

布尔类型是枚举类型的一个特例，其原因稍后就会明白。

3.6.2 枚举类型

我们以前介绍的类型都是内部型的，Ada 语言允许通过类型说明产生一个新的类型。对于处理

星期的问题，可由如下说明：

TYPE day IS (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);

产生一个新的类型 day。day 为枚举类型的名字。括号中所列出的标识符是类型为 day 的变量可以取值的范围。这些标识符称为枚举记号。

当说明了一个新类型后，就可以用常规方法说明相应的变量与常量。例如：

```
weekday:day := Monday;
arrival:day;
midweek:CONSTANT day := Wednesday;
```

作为枚举类型的另一个例子，考虑编制一程序来控制交通灯，灯的颜色只有三种可能：red、yellow、green。因此类型可以这样定义：

TYPE color IS (red, yellow, green);

我们可以说明一个 color 类型变量并赋初值 yellow：

```
color_of_light:color := yellow;
```

下面我们给出枚举类型说明的语法图（见图 3.3 所示）。

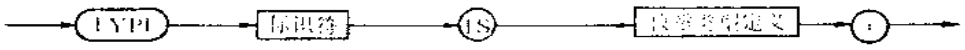


图 3.3

其中，枚举类型定义可用如下语法图表示（见图 3.4）。

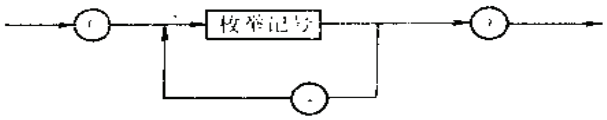


图 3.4

不难发现，boolean（布尔）类型可用：

TYPE boolean IS (false, true);

说明，因此说 Boolean 类型也是枚举类型。

至此，我们介绍了 integer float、real、boolean 、String 等 Ada 的内部类型以及相应的常量或变量的说明，此外还介绍了枚举类型说明。下一章将介绍枚举记号的表达式，关于类型在第八章中还要再次讨论。

练习

1. 将下列值按类型分类。

92　3.5　false　4.9e−4　','　"Sum="　4e3　4.0e3　"false"

2. 指出下列值所属的不同类型。

9　9.0　'9'　"9"

3. 在编制程序时，何时使用常量标识符更佳？

4. 考虑如下的 Ada 说明：

```
sum, total:integer := 0;
TYPE tree IS (oak, rowan, birch, fir);
tall:tree := oak;
```

mountain _ ash:CONSTANT tree : =rowan;

pressure: real;

letter: character : ='a';

请指出变量标识符以及相应的类型和值共有几个变量是未定义值的。

5. 写出下列 Ada 说明:

(a) 整数变量 number 和 size;

(b) 字符常量 first _ in _ alphabet;

(c) 枚举类型 month,按自然顺序处理 12 个月份;

(d) 类型为 month 的变量;

(e) 类型为 month 的常量 first _ month;

(g) 一实常量 zero;

6. 试编一程序,要求输入一个盒子的长、高、宽,计算盒子的体积并打印。

第4

表达式和赋值语句是每个 Ada 程序[illegible]整数、实数以及布尔表达式，还要介绍如何确定表达式中运算符[illegible]

程序示例中已使用了表达式和赋值语句，[illegible]的赋值语句：

```
pay := basic_pay + overtime_pay;
```

这个语句的作用是，首先计算表达式"basic_pay + overtime_pay"之值，然后将所得的值赋给变量 pay。

表达式是计算新值的方式，它由一个或多个运算对象组成，各个运算对象之间由"+"、"*"这样的运算符连接起来。运算对象通常为常量或变量对象。例如：

```
86    3+5    first    2*second+18    "j"    pay>200
```

就是一些简单的表达式。

注意，表达式可以只有一个运算对象。

变量或常量都有一个类型和一个值，同样一个表达式也有一个类型，其类型依赖于组成表达式的运算和运算的对象。因此，表达式"3+5"之值为 8，其类型为 integer。表达式"j"的类型为 character；而表达式"pay>200"为 boolean 类型，因其值由变量 pay 是否大于 200 来确定。

在以下的各节中，我们将详细介绍整数、实数和布尔表达式，以及如何确定表达式中的运算符的优先级。

需要重申的是，Ada 语言是一种强类型语言，不同类型的值不允许出现在同一表达式中，例如不能把一个整数与一个实数相加或相乘。同样，在赋值语句中，符号":="右边的表达式类型必须与其左边的变量的类型相同。

§4-2 算术表达式

算术表达式由数值与通常的加、减、乘、除等算术运算符组成。若设 a、b 是变量标识符，则下列表达式之值的计算如下：

```
a+b    ;表示变量 a 加变量 b 之值
a-b    ;表示变量 a 减去变量 b 之值
a*b    ;表示变量 a 乘以变量 b 之值
a/b    ;表示变量 a 除以变量 b 之值
```

变量 a 与变量 b 必须说明为同一类型。如果它们都是 integer 变量，则以上四个表达式都是 integer 类型；如果变量 a、b 是 real 类型，则上述表达式也是 real 类型。

两个整数相除之值为一个整数，其余数被忽略。例如：

```
17/8        等于 2
```

而两个实数相除结果为一实数

Ada 语言中有一个余数运算 …算对象。表达式"a REM b"表示整数变量 a 除以整数变量 b 的余数。

通常，运算符"+"和"-"有两个 …个运算对象。故称之为一元运算符。-a 表示变量 a 的负值。一元运算符"+ …达式+a、a 表示同一个值。另一个一元运算符为 ABS，ABS a 的值表示变量 a 。

"* *"是 Ada 中的指数运算符。表达 …为指数。当指数是正时，指数的作用与连乘是等效的，如 5 * *3 等价于 5 * 5 * 5

对于整数表达式，指数必须是正整数或是 …不可能得到一个整数的。而对于实数表达式，指数可正可负。例如，表达式 4.5 * …4.5 * 4.5；表达式 4.5 * *(-3)等价于 1/(4.5 * 4.5 * 4.5)。

一个数的零次幂之值为 1。因此，5 * *0 等于 …于 1.0。

当一个表达式中有若干个运算时，就需明确各个运 …关系。运算符的优先级原则类似于一般的算术运算符。我们可以把这些规则表述如下： …由上至下：

1. 指数和绝对值
2. 乘、除和余数
3. 加、减

当运算符处于同一级别时，运算按从左至右的顺序进行。但是圆括号可改变运算顺序，例如：

9-5+2	等于 6
9-(5+2)	等于 2
3+4 * 5	等于 23
(3+4) * 5	等于 35
16/(2 * 4 * 2)	等于 1
5 * 2 * *3	等于 40

§ 4-3 应用举例

例 4-1 读入一个单位为米的距离，试将其转换成码、英尺、英寸并输出。

正如第一章所述，编制计算机程序的第一步是构思算法，这个算法也称"高级算法"。高级算法通常与用纸和笔解题的步骤是一样的。下面给出算法：

读入一单位为米的数
将米转换成英寸
计算英寸中的英尺数及剩下的英寸数
计算英尺中的码数以及剩下的英尺数
分别写出所得的码、英尺、英寸数

编制高级算法可以使我们在处理细节问题之前有一个正确无误的主算法框图，这种解题方法称为"自上而下的设计方法"。

因为本题是很简单的，因此可以直接从高级算法开始编制 Ada 程序。考虑所需的变量，需要用

变量来存放米数、码数、英尺数以及英寸数。接着考虑这些变量的类型。显然，英尺数和码数应是整数。为使问题更一般，我们希望米数为一个实数(尽管英寸数可以是一实数，但在此处限制英寸数为整数)。

另外还需要知道将米转换成英寸的因子。这个因子是不难在其它书中查到的。

综上所述，如下的说明是编制程序所必须的。

```
yards, feet ,inches: integer;
metres : =real;
    conversion _ factor:CONSTANT real : =39.37;
```

接下来考虑将单位为米的数转换成英寸。存储在变量 metres 中的值与常量 conversion _ factor 的值相乘就可以得到相应的英寸数，但其类型为 real。要想将这个实数存储在 integer 变量 inches 中是非法的。那么如何解决这个问题呢？

Ada 中有一特殊的转换运算，允许把一个类型的值转换成另一类型的相应值。例如，要把 2 这样的整数转换成 real 类型的值，只需写 real(2)；同样，若需把实数对象或实数 3.8 转换成类型 integer 之值时，可写 integer(3.8)。一个实数转换成整数是一个四舍五入的过程。例如：

real(2)+3.7	等于 5.7
2+integer(3.7)	等于 6

将以米为单位的实数换成以英寸为单位的整数，并将该整数赋给 integer 型变量 inches 的赋值语句如下：

```
inches : =integer(metres * conversion _ factor)
```

当 inches 被赋值之后，就可以利用除法和余数运算计算相应的码和英尺。完整的程序如下：

程序4-1

```
WITH student _ io; USE student _ io;
PROCEDURE convert IS
    yards, feet ,inches: integer;
    metres: real;
    conversion _ factor: CONSTANT real : =39.37;
BEGIN
    --convert metres to yards, feet,inches  (把米转换成码，英尺，英寸)
    put("number of metres="); get(metres);
    inches : =integer(metres * conversion _ factor);
    feet : =inches/12; --convert inches to feet  (将英尺转换成英寸)
    inches : =inches REM 12; --find number of inches left over  (求英尺余数)
    yards: =feet/3; --convert feet to yards  (将英寸转换成码)
    feet : =feet REM 3; --find number of feet left over  (求英寸余数)
    put(yards); put("yards");
    put(feet, width=>2); put("feet");
    put (inches, width=> 3); put("inches");
    new _ line;
END convert;
```

运行程序时，若在"number of metres="的提示下输入数 2.1，则输出结果为：2 yards 0 feet 11 inches

当编完一个程序后，不要急于输入计算机中进行调试。最好的办法是选择简单的数据进行人工测试，以诊断所编程序是否正确。用人工方法测试程序，在以后的章节中还将要提及。

§4-4 关系表达式

布尔表达式只有两种取值可能：true 和 false，最常见的布尔表达式是关系表达式，由关系运算符连接两个值组成。Ada 中有 6 个关系运算符：

=	等于	<=	小于等于
/=	不等于	>	大于
<	小于	>=	大于等于

例如：表达式"3<7"的值为 true。

对关系表达式中的比较对象是没有什么限制的，关系运算符可以用来比较类型 integer、real、character 以及任何枚举类型的值；但必须遵守类型规则，即两个相比较的对象必须是同一类型。

所有的关系运算符的运算优先级都低于算术运算符，因此表达式：

```
basic_pay+overtime_pay>200
```

的功能为：先把变量 basic_pay、overtime_pay 之值相加，然后将所得的值与 200 进行比较。同样，假设 number 是一个 integer 变量，若 number 之值为偶数则表达式：

```
number REM 2=0
```

的值为 true。

当比较两个字符时，实际进行比较的是两个字符的相应序号，即 ASCII 码。附录中给出了所有字符的序号。不必记住所有字符的 ASCII 码，但牢记下述关系表达式是很有用的；以下这些关系表达式的值都为 true。

'a'<'b'	'b'<'c'	…	'y'<'z'
'A'<'B'	'B'<'C'	…	'Y'<'Z'
'0'<'1'	'1'<'2'	…	'8'<'9'

两个字符串进行比较时，如果一个串的第一个字符大于另一个串的第一个字符，则说这个串大于另一个。如果两个串的第一个字符相同，则继续比较两个串中的第二个字符。如此继续。

下面所列的关系表达式的值都为 true。

```
"Jane"<"Mary"
"Julie">"Julia"
"John">"James"
"Anne">" Ann"
```

我们曾讲过，一个实数在计算机中是近似表示的，实数之间的操作所得的结果也是近似的。因此表达式：

```
(1.0/3.0)*3.0=1.0
```

就不能肯定这个结果是否正确，因为"="左边的表达式的计算结果可能是 0.999…。所以，最好尽量避免在实数中使用等于或不等于这样的运算符。

一个枚举文字的相应值取决于其说明时的次序。因此，对于说明：

```
TYPE wedding IS (silver, golden, diamond);
```

如下所列的关系表达式之值都为 true：

```
silver<golden
golden<diamond
```

§ 4-5 成员测试(membership tests)

当需要判断某一标量类型之值是否落在某一给定的范围或限制(range)内时，可以使用成员测试(membership tests)：IN 和 NOT IN。例如表达式：

```
pay IN 150 .. 250
```

如果 pay 之值介于 150 至 250 之间(包括端点)，则其值为 true；反之则为 false。而表达式：

```
pay NOT IN 150 .. 250
```

的功能与之相反。依照类型原则，pay 必须是 integer 类型。

对于类型 wedding 的文字，可知表达式：

```
silver IN golden .. diamond
```

的值为 false；而表达式：

```
silver NOT IN golden .. diamond
```

的值则为 true。

成员测试的运算优先级与关系运算等级。例如：

```
element, lower: integer;
```

并且两个变量已赋值，则对于表达式：

```
element IN lower+4 .. lower+17
```

的计算是这样的：先计算表达式 lower+4 和 lower+17 之值，然后检查 element 是否在限制之中。

关于限制，第一个值为下界，第二个值为上界，下界与上界用两个圆点".."分隔。如果下界大于上界，则说这个限制是空的，一个空的限制没有任何值。IN、NOT IN 的优先级与关系运算符相同，但它们不是运算符。

§ 4-6 逻辑表达式

有时希望将两个或更多个布尔表达式连接在一起，这可通过逻辑运算符 AND、OR 和 XOR 来实现。

首先看几个逻辑表达式。

表达式①：

```
element REM 2=0 AND element>0
```

若 element 是大于零的偶数，则其值为 true；反之则为 false。

表达式②：

```
element REM 2=0 OR element>0
```

只要两个关系表达式中有一个为 true，则其值为 true；否则其值才为 false。

表达式③：

```
element REM 2=0 XOR element>20
```

其中含有一个"排斥"运算符"XOR"，如果表达式中只有一个值为 true，则其值也为 true，当两个表达式之值均为 false 或 true 时，其值为 false。

运算符 AND、OR 和 XOR 具有相同的运算优先级，但低于关系运算符，这也是为什么上述表达式中不用括号的原因；但是有时为了增加程序的可读性可以使用括号。如：

(element REM 2=0) XOR element>20

NOT 是一元逻辑运算符，使用 NOT 可以得一个布尔表达式的相反值。例如表达式 NOT(element<4)与表达式 element>=4 具有相同的值。NOT 的优先级高于乘、除的优先级。

假设已有说明：

bright, cheerful: Boolean;

下面列出一些逻辑表达式：

NOT cheerful OR bright	同(NOT cheerful) OR bright
NOT (cheerful OR bright)	
birght AND NOT cheerful	同 bright AND (NOT cheerful)
bright OR lower <10	同 bright OR (lower<10)
NOT (lower =5)	这里括号是必须的

当布尔表达式中有不止一个逻辑运算符 AND、OR 及 XOR 时，为明确和容易理解起见，适当使用括号是必要的。

例如，如下两个表达式具有不同的运算顺序，这里的括号是必需的：

(bright AND cheerful) OR lower <10

bright AND (cheerful OR lower <10)

§4-7 标量类型的属性

一个变量的类型决定了它可储存值的范围以及可实施何种运算操作。对于一个标量类型来说，它还有一些属性，如最小值、最大值等。对于枚举类型也可以使用关系表达式，相应的枚举文字有一个序号值，该值取决于类型说明时的顺序。

假如有如下说明：

TYPE day (sun, mon, tues, wed, thurs, fri, sat);

TYPE card (two, three, four, five, six, seven, eight, nine,ten, jack, queen, king, ace);

appointment: day := wed;

则下列表达式的值为 true：

appointment=wed

mon<thurs

four>two

queen<ace

在表达式中还可以直接使用一个类型的属性。一个类型的属性是类型标识符后跟一个单引号"'"，其后面是一个说明特殊属性的标识符，如 first、last、pred 等。

例如，类型 day 的最小值 sun 和最大值 sat 可表示为：

day'first

day'last

属性 succ 和 pred 给出一枚举值的前驱值和后继值，例如表达式：

day'succ(mon)

card'pred(queen)

的值分别为 tues 和 jack。又若执行语句：

```
appointment := day'succ(appointment);
```

后，appointment 的值将由 wed 变成 thurs。

注意，一个类型的最小值没有前驱，最大值也没有后继。

除了枚举类型具有属性定义外，Ada 中所有的类型都存在属性。但某些如 integer、float 类型的最小值和最大值与 Ada 的实现有关，利用表达式：

```
integer'first, integer'last, float'first, float'last
```

就可获知与你所使用的计算机系统相对应的这些值。如果在编程时需要用到上述这些值，则应使用属性表达式来替代实际的值，这样编制后的程序会尽可能与特殊的机型无关而具有可移植性。对于所有的机器，integer'first、float'first 总是小于零的，integer'last、float'last 总是正的。

对于 integer、character 类型也可以使用属性 succ 和 pred，例如表达式：

```
integer'succ(220)
integer'pred(222)
```

的值都是 211。

显然，对实数值谈论其前驱值和后继值是没有任何意义的。我们称具有前驱和后继属性的类型 integer、charater、boolean 和枚举类型为离散类型。除 real 和 float 类型之外，所有其它标量类型都是离散类型。

下表列出了截止到目前所引入的全部运算，并按优先级分组。

运算符	操作	运算数	结果
AND	与	BOOLEAN	BOOLEAN
OR	相容或	BOOLEAN	BOOLEAN
XOR	排斥或	BOOLEAN	BOOLEAN
=	等于	任意	BOOLEAN
/=	不等于	任意	BOOLEAN
<	小于	任意	BOOLEAN
<=	小于或等于	任意	BOOLEAN
>	大于	任意	BOOLEAN
>=	大于或等于	任意	BOOLEAN
+(二元)	加	数值	相同
-(二元)	减	数值	相同
+(一元)	恒等	数值	相同
-(一元)	取负	数值	相同
*	乘	INTEGER	INTEGER
/	除	REAL	REAL
MOD	取模	INTEGER	INTEGER
REM	余数	INTEGER	INTEGER
**	乘幂	INTEGER 非负	INTEGER
	INTEGER	REAL	INTEGER
	REAL	INTEGER	REAL
NOT	取反	BOOLEAN	BOOLEAN
ABS	绝对值	数值	相同

练习

1. 设有说明：

```
speed:real := 35.5;
time:real := 3.0
start,stop:real := 2.0;
distance:real;
```

按下列顺序执行赋值语句，请指出最后所有变量的值。

```
distance := speed * time;
time := start+distance/speed+stop;
speed := distance/time;
```

2. 设有说明：

```
centigrade:integer
```

试指出当变量 centigrade 的值分别为 12,15,40 和 100 时下列表达式的值。

```
centigrade * 9/5+32
real(centigrade) * 1.0+32.0
```

3. 试编制一个完整的 Ada 程序，要求输入三个数，计算三数之和及其平均值并输出结果。用数据 2.5,17.3,12.1 测试所编的程序。

4. 计算如下表达式的值：

```
5+4/3*2      17-3+9      17-(3+9)
168/4/2      168/(4/2)   (168/4)/2
30*4/5       30*(4/5)
2+3<6 and 15 <=12 "MARY"> "Margaret"
card'succ(card'first) integer'first<0
six NOT IN ten .. card'last
```

5. 假设 first、second 为 integer 变量，写出下面的布尔表达式。

(a) first 大于 second

(b) first 被 second 整除

(c) first, second 都大于或等于 0

(d) first, second 不同时为 0

(f) first 大于或等于 1，但小于等于 8

6. 假设已有如下说明：

```
TYPE day IS (sun, mon, tues, wed, thurs, fri, sat);
my_appointment:day := mon;
your_appointment:day := wed;
intermediate:day;
```

执行下列两个赋值语句序列，请指出执行后各变量的值。

```
(a) intermediate := my_appointment;
    my_appointment := your_apppointment;
    your_appointment := intermediate;
(b) my_appointment := your_appointment;
    your_appointment := my_appointment;
```

7. 设年通货膨胀率为 x%，则 n 年之后，1000 元就只值 1000*(100/(100+x)**n)。试编一程序，读入 x、n，计算 n 年之后 1000 元值多少。

第五章　控制结构

本章介绍 Ada 的三种顺序控制结构，即条件语句、情形语句和循环语句，使用这三种控制结构可使程序具有清晰的控制流。

三种控制结构具有相似的词法风格。它们以保留字 IF、CASE 或 LOOP 开始，在结构的末尾由同一原保留词前面冠以 END 相匹配，整体是分号终止。如下所示：

```
IF          CASE          LOOP
...         ...           ...
END IF;     END CASE;     END LOOP;
```

就循环语句而言，LOOP 前面可以加上 FOR 或 WHILE 开始的重复子句。

§5-1　IF 语句

5.1.1　IF ... END IF 型语句

条件语句最简单的形式是以保留字 IF 开始，接着是一个布尔表达式和保留字 THEN，然后是语句序列，序列的末尾由 END IF 指明。若表达式的值为 true，就执行 THEN 之后的语句序列。这里，布尔表达式可以任意复杂，语句序列的长度也可以是任意的。

现在，让我们再考察程序 4-1，该程序的结果是由下列语句输出的：

```
put(yard); put("yards");
put(feet, width=>2); put("feet");
put(inches, width=>3); put("inches");
```

如果输入以米为单位的数 2.1，则可知 2.1 米为 2 码 11 英寸；计算机输出的结果为：

```
2 yard 0 feet 11 inches
```

显然，这种输出形式是不够理想的，我们希望根据变量 yards、feet 和 inches 之值是否大于零来确定是否打印相应的内容。利用 IF 语句就可以实现这个目的：

```
IF yards > 0 THEN
    put(yards); put("yards");
END IF;
IF feet > 0 THEN
    put(feet, width => 2); put ("feet");
END IF;
    put(inches,width=> 3); put ("inches");
```

这里用了两个 IF 语句。如果仍输入 2.1，执行情况是：首先计算保留字 IF 之后的布尔表达式。由于 yards 的值为 2，所以 yards> 0 之值为 true，因此，执行保留字 IF 与 END IF 之间的语句：

```
put(yards); put("yards");
```

接着，又是一个 IF 语句，同样先计算表达式 feet>0 之值。因为其值为 false，所以跳过介于 THEN 与 END IF 之间的语句：

```
put(feet, width=>2); put("feet");
```

最后执行语句：

```
put (inches, width =>3); put("inches");
```

打印结果为：

```
2 yards 11 inches
```

IF 语句是我们所遇到的第一个复合语句。尽管 IF 与 END IF 之间含有其它语句，但仍视其整体为单个语句，这也是在 END IF 之后用分号的原因。

5.1.2 IF ... ELSE ... END IF 型语句

通常在解题时，需要在两个或几个可能途径中选择其一，IF 语句的其它形式可以处理此类情况。例如，假设 a 已说明为 integer 类型，下列语句根据 a 是否大于零打印不同信息。

```
IF a >0 THEN
    put("greater than zero");
ELSE
    put("not greater than zero");
END IF;
```

上述语句将根据布尔表达式 a>0 之值是 true 或 false，确定执行 THEN 之后的语句还是执行保留字 ELSE 之后的语句。

5.1.3 IF…ELSE…END IF 型语句

考虑如下形式的 IF 语句：

```
IF a>0 THEN
    put("greater than zero");
ELSIF a<0 THEN
    put("Less than zero");
ELSE
    put("egual to zero");
END IF;
```

执行这个语句时，首先计算表达式 a>0 的值，如其值为 true 就执行 THEN 之后的语句；只有当表达式 a>0 之值为 false 时，才继续计算 ELSIF 之后的表达式 a>0，如其值为 true 就执行第二个 THEN 之后的语句；如果表达式 a>0 之值也为 false，就执行 ELSE 之后的语句。

现在，用如下的语法图来表示一般的 IF 语句，其中"语句序列"是指由一个或多个语句所组成。

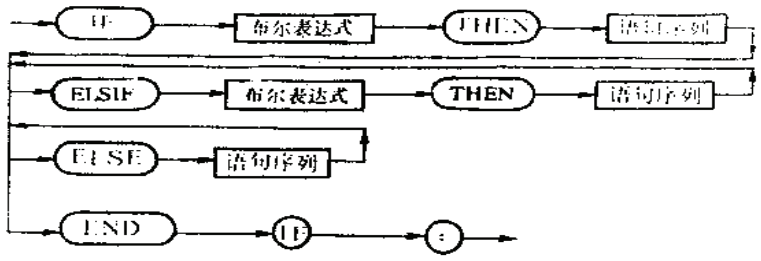


图 5.1

执行一个 IF 语句时，首先计算 IF 后面的布尔表达式，如其值为 true 就执行 THEN 之后的语句序列；只有当其值为 false 才执行其它的语句。其中 ELSIF 部分可有可无，并且 ELSIF 可有多个。ELSIF 之后的布尔表达式按顺序计算，直到某一个的值为 true 为止，然后执行相应的 THEN 之后的语句序列。如果所有的布尔表达式都是 false，则就执行 ELSE 之后的语句序列（如果有 ELSE 部分的话）。ELSIF 是 Ada 中唯一的一个非英文单词的保留字，读者在使用时需加以注意，同时也请读者注意 IF 语句的书写格式，本书所采用的是分层缩进、对齐的格式。这样一来，不仅使程序的易读性相应得以提高，而且这也是确保程序的易维护性、可靠性的重要措施，对于初学者来说养成好的习惯是相当重要的。最后，让我们看一个例子，其中含有多个 ELSIF，并假设变量 ch 已说明为字符 character 类型。

```
IF ch IN 'a' .. 'z' THEN
    put ("lower case");
ELSIF ch IN 'A' .. 'Z' THEN
    put ("upper case");
ELSIF ch IN '0' .. '9'
    put("digit");
ELSE
    put("other character");
END IF ;
```

执行上述语句时，依次计算成员测试表达式，直到某一值为 true 为止，然后打印出相应的信息；如果所有的成员测试值都为 false，则打印出："other character"。

§5-2 应用举例

现在我们用自上而下的（top-down）设计方法考虑算法的设计。

例 5-1 读入两个整数并判别是否能整除。

对于例 5-1，首先求其解法，这个过程的第一步是理解题意及要求。当用一个整数去除另一个整数时，若余数为零，则说两数整除。因此，判别两数是否整除时，总是用小的整数去除大的整数，然后再判别其余数是否为零。问题是，对于任意输入的两个整数（设分别存放在 integer 类型变量 first、second 中），则不可能预知哪一个大；但通过交换两个数就可使得第一个数 first 始终大于或等于第二个数 second。上述的分析可写成如下的算法：

```
读入两个整数(first, second)
IF second 大于 first THEN
    两数交换
END IF
计算 second 除 first 的余数 remainder
IF 余数 remainder 为零 THEN
    打印"One is divisible by the other"
ELSE
    打印"One is not divisible by the other"
END IF
```

这种用英文单词 IF-THEN-ELSE-ENDIF 书写成的算法，可以视为非正式的"程序设计语言"，

一个算法通常含有顺序、分支(选择)及循环等结构,语句所遵守的次序就是书写时的先后次序。分支可描述为:

```
IF 条件为真 THEN
    做某件事
ELSE
    做另一件事
END IF
```

关于循环将在下节介绍。这里所采用的书写格式也是分层缩进、对齐格式。

高级算法中的每一个语句,可以认为是对某一子问题的说明,这些子问题有待进一步解决。在寻求高级算法时,总是首先将初始问题分解成若干相对独立的子问题,而每个子问题应比原问题更易求解,这是高级算法的基本准则。首先应把精力集中在解法的结构上,然后考虑子问题。

在简单问题中,子问题不需要进一步分解;但在一般情况下,某些子问题还需分解成若干更小的子问题,然后用同样的方法编制每个子问题的算法。这种解题方法有多种称法,通常称之为自上而下的设计方法或结构化程序设计,或逐步优化程序设计方法。

编制好一个算法后,可以用样本数据来测试其是否正确。必须精心选拔测试数据,以确保算法中的每个部分都至少被测试一次,也必须考虑所有的可能性。

如例 5-1,若除数为零,我们就会发现忽视了一个重要情形,即一个数被零除其值为无穷大;因此必须对原来的算法加以修改使之能处理除数为零的情形。由此可见,算法的编制并不是一个线性过程,而是一个反复过程,必须对最先的算法不断加以修改、完善。现在,算法成为:

```
读入两个整数(first, second)
IF second 大于 first THEN
    两数互换
END IF
IF second 等于 0 THEN
    打印"Cannot divide by zero"
ELSE
    求 second 除 first 的余数 remainder
IF 余数 remainder 为 0 THEN
    打印"One is divisible by the other"
ELSE
    打印" One is not divisible by the other"
END IF
END IF
```

检查上述修改后的算法之后,可以考虑两数互换的问题。不难设想,我们需要一个中间变量,不妨设为 intermediate。交换的算法为:

```
把 first 之值赋给 intermediate
将 second 之值赋给 first
将 intermediate 之值赋给 second
```

至此,我们已经完成了算法的编制,接下来就是用 Ada 来实现,其程序如下。

程序 5-1

```
WITH student_io; USE student_io;
PROCEDURE divisible IS
    first, second: integer; --input data
                                (输入数据)
    intermediate:integer; --used in swapping
                                    (交换时用)
    remainder: integer;
BEGIN
    --find if one number can exactly divide another
    (判别两数是否整除)
    put ("two positive integers please");
    get (first); get (second);
    --ensure that first>=second
    (保证 first>=second)
    IF second> first THEN
        intermediate := first;
        first := second;
        second := intermediate;
    END IF;
    IF second=0 THEN
        put ("Cannot divide by zero);
    ELSIF
        --calculate remainder (计算余数)
        remainder := first REM second;
        IF remainder =0 THEN
        put(first, width =>1);
        put("is exactly divisible by ");
        put(second, width=>1);
    ELSE
        put("Not exactly divisible");
    END IF ;
    END IF;
        new-line;
END divisible;
```

在上一章中，我们曾提及检查程序的正确与否的重要手段是选择适当的测试数据跟踪程序运行(即人工测试手段)。但必须注意，所选的数据必须使得程序的每一部分都至少测试一次，就本例而言，数据的选择应考虑如下几点：

(i) first 大于或等于 second

(ii) first 小于 second

(iii) 整除

(iv) 不整除

(v) 用 0 作除数

按照上述 5 点，可以选择如下这样的数据对：

24	6
8	16

5　　9

7　　0

以上每组数据可测试至少一种情形。

最后，值得提请注意的是，一个 IF 语句可以嵌入在另一个 IF 语句中。由于每个 IF 语句都视为单个语句，所以这种嵌套是不会引起麻烦的；但如果使用嵌套 IF 语句，必须有相应的 ENDIF 相匹配。在 Ada 语言中，对语句的内部序列语句的种类是不加限制的。

§5-3　短路控制形式

例 5-1 中，进行取余运算之前，总是先检查除数是否为零。在 Ada 中有两个短路控制形式 AND THEN 和 OR ELSE，使用短路控制形式可以将上述两个过程组合在一个表达式中。

AND THEN 形式与运算符 AND 紧密相连，而 OR ELSE 与运算符 OR 紧密相连。AND THEN 与 OR ELSE 也可以出现在表达式中，与运算符 AND、OR 和 XOR 具有相同的优先级，区别在于它们计算时的规则。

在 AND 和 OR 情形时，两个运算对象都计算，但未限定次序。例如对于表达式：

seond/＝0 AND (first REM senond＝0)

虽然，当 second 等于 0 时整个布尔表达式之值为 false，但这并未满足要求，因为运算符两边的操作总是都要计算的，并且计算的次序也未指明，所以上述表达式仍可能包含除数为零的情形。若使用表达式：

second/＝0 AND THEN (first REM second＝0)

就可以排除除数为零的情形。因为只有当条件 second/＝0 成立时(即表达式之值为 true)，才计算第二条件，否则整个表达式之值为 false。这就避免了用零作除数的情形。

对 OR ELSE 形式而言，也是先进行左边的操作，根据运算结果决定是否进行右边的操作。例如对于表达式：

second ＝0 OR ELSE (first REM second /＝0)

只有当 second 不为零时才计算 OR ELSE 之后的表达式。当 second＝0 之值为 true 时不管 OR ELSE 之后的表达式之值怎样，整个表达式之值总是 true。

一般地，设 X、Y 分别表示布尔表达式，在表达式：

X AND THEN Y

中，先计算 X，若 X 为 false，则无论 Y 之值如何，整个表达式之值也为 false，不再计算 Y 之值；若 X 为 true，则必须计算 Y，Y 之值就是整个表达式的值。

同样，在表达式：

X OR ELSE

中。首先计算 X，若 X 之值为 true，则不管 Y 之值如何，整个表达式之值也为 true，不再计算 Y 之值；若 X 之值为 false，则须计算 Y，Y 之值即为整个表达式之值。

短路控制形式是为那些特别强调运算次序的情形而专门设计的，可以用来阻止计算第二条件，使程序更简洁。

现在，用短路控制形式来改写上一节中的程序，其修改的程序如下。

程序 5-2

```
WITH student_io; USE student_io;
```

```
PROCEDURE divisible IS
    first, second: integer; --input data
                                (输入数据)
    intermediate:integer; --used in swapping
                                (交换时用)
BEGIN
    --find if one number can exactly divide another
    (判别两数是否整除)
    put("two positive integers please");
    get(first); get(second);
    --ensure that first>=second
    (保证 first>= second)
    IF second >first THEN
        intermediate := first;
        first := second;
        second := intermediate;
    END IF;
    IF second/=0 AND THEN (first REM second=0) THEN
        put(first, width=>1);
        put("is exactly divisible by ");
        put(second, width=>1);
    ELSE
        put ("Not exactly divisible");
    END IF;
        new_line;
END divisible;
```

§5-4 LOOP(循环)语句与 EXIT(出口)语句

LOOP 语句的最简单形式为：

```
LOOP
    语句序列
END LOOP;
```

但对于上述的语句，如果不采用某种手段加以终止就会无限制地循环。

重复进行相同或类似的运算或操作是计算机的一大特点。

例 5-3 设有一系列正数，最后一个为负数(作为终止标志)，要求依次读入每个正数并求它们的和。

利用计算机解此题是相当方便的，只需重复下述操作：

读入一个整数

将该数加至累加器中

当读入一个负数时，就终止上述操作，打印结果。

这就是解该题的算法。该算法可由 Ada 中的 LOOP 语句来实现。每次循环时，必须测试读入的数是否小于 0，以确定是否终止循环，这可由 EXIT 语句完成。假设已有说明：

```
sum, number: real;
```

如下程序段可实现上述算法：

```
Sum := 0.0; --the sum is initally set to zero
(将 sum 初值置为 0)
LOOP
    get(number);
EXIT WHEN number<0.0;
    Sum := sum+number;
    --sum contains the total so far
    (累计当前之和)
END LOOP;
```

这个语句序列的执行情况是：首先将累加器 sum 的值置为零，然后进入循环体内。读入一个数，接着执行 EXIT 语句以确定读入的数是否小于零，如果小于零就跳出循环，执行 END LOOP 之后的语句；否则就执行 EXIT 语句之后的语句，即将所读入的数与当前的 sum 的值相加并赋给变量 sum。遇到保留字 END LOOP，于是转到该循环语句的开始，接着再读入下一个数并加以测试，如此继续循环，直到 EXIT 语句之后的布尔表达式之值为 true 为止才能跳出循环。因而，如果输入以下数据：

15.7　　18.9　　14.3　　9.7　　6.5　　−2.5

就将前 5 个正数相加，当读入第 6 个负数时，就终止循环。

在编制程序时，如果由于某个逻辑错误而导致 EXIT 语句的出口条件永远不成立，则称这种情形为“无限循环”。

简单的 LOOP 语句可由语法图图 5.2 表示。

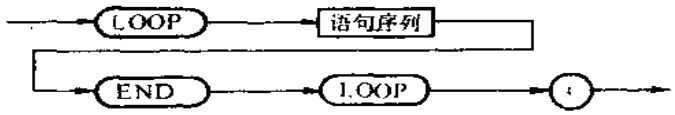


图 5.2

EXIT 语句可表示为图 5.3 的形式。

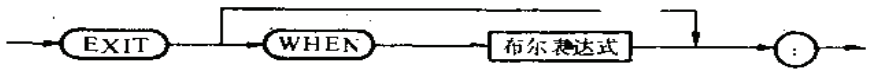


图 5.3

由 EXIT 语句的语法图可知，EXIT 语句分为无条件和有条件两种形式，下列两个语句反映两者之间的关系。

```
EXIT WHEN number<0.0;
```

与

```
IF number <0.0 THEN
    EXIT;
END IF;
```

是等价的，但显然前一个语句更好，并可增加程序的可读性。

例 5-4　输入一个英文句子，统计其中有多少大写、小写、空格、标点符号以及其它字符，句子可以不在同一行上。

程序 5-4

```
WITH student_io; USE student_io;
PROCEDURE sentence IS
    capital_count, lower_count, blank_count,
    punct_count : integer := 0;
    --counts initialised (累加器之初始化)
    ch : =character;
BEGIN
    --program to read and analysis the characters in a sentence
    (分析语句成分的程序)
    put ("Type in a sentence please");
    LOOP
        --get and classify the next character
        (读入一个字符并分类)
        get(ch);
        IF ch IN 'a' .. 'z' THEN
            lower_count : =lower_count+1;
            ELSIF ch IN 'A' .. 'Z' THEN
                capital_count : =capital_count+1;
            ELSIF ch = ' ' THEN
                blank_count : =blank_count+1;
            ELSE
                punct_count : =punct_count+1;
        END IF;
            --the counts contain the totals so far
            (累计至当前为止)
        EXIT WHEN ch='.';
        END LOOP;
    put("The number of capitals is");
    put(capital_count, width=>1);
    new_line;
    put("The number of lower case is");
    put (lower_count, width =>1);
    new_line;
    put("The number of blanks is");
    put(blank_count, width =>1);
    new_line;
    put("The number of punctation");
    put("characters is");
put(punct_count,width=>1); new_line;
END sentence;
```

如果我们输入句子：

It is a long, long way to Hainan.

则运行结果如下：

```
The number of capitals is 2
The number of lower case is 22
The number of blanks is 7
The number of punctuation characters is 2
```

该程序执行时，首先在其说明部分里将累加器 capital _ countlower _ count、blank _ count、punct _ count 说明为 integer 类型并赋初值为 0。每一次循环都读入一个字符，四个计数器中的一个就增加 1，然后再测试是否为句子的末尾，一旦遇到圆点就终止循环；否则回到循环开始处，再读入下一个字符。需注意的是，圆点字符也是句子的一部分，作为标点符号计数。

至此，我们已介绍了简单 LOOP 语句以及 EXIT 语句。循环语句还有两种形式——WHILE 循环语句和 FOR 循环语句，将在以下两节分别介绍。

§5-5 WHILE 循环语句

上一节所介绍的 LOOP 循环，其 EXIT 语句可出现在循环体中的任何地方，并且可以不止一个 EXIT 语句；只有一个出口，并且这个出口就在循环的开始之处。

例 5-5 读入一个整数，求 1 到该整数之间所有奇数的和。设有说明：

```
Limit, integer;
odd _ number:integer := 1>
sum _ odd:hntegnr := 0;
```

求奇数之和可由下列语句完成：

```
get(limit);
WHILE odo _ number<=limit LOOP
    sum _ odd := sum _ odd+odd _ ntmber3
    odb _ number := odc _ numbex+2;
    --sum _ odd contains the sum of
    --the odd number so far
    (sum _ odd 为当前所有奇数之和)
END LOOP;
```

看一下上述语句序列执行时的情况：读入 limit 的值之后就进入 WHILE 循环，测试计算保留字 WHILE 之后的布尔表达式，若其值为 true，就执行介于 LOOP 与 END LOOP 之间的语句序列。然后转至循环开始处，再次执行上述语句。如此继续直到 WHILE 之后的布尔表达式之值为 false 为止才终止循环。在本例中，每循环一次，变量 odd _ number 之值就加 2，直到超过终值 limit 时，下一次计算布尔表达式时其值就为 false，因而终止循环。

一个 WHILE 循环语句总是可以用简单 LOOP 语句改写，如上面的 WHILE 循环语句可写成：

```
get(limit);
LOOP
    EXIT WHEN odd _ number >Limit;
    sum := sum _ odd+odd _ number;
    odd _ number := odd _ number+2
END LOOP;
```

注意，如果所读入的数小于 0，则循环中的语句就根本未被执行，我们称这样的循环为空循环。

对于 WHILE 循环而言，其循环体中也可含有 EXIT 语句，但是不提倡这样做，因为这样会破坏 WHILE 循环语句的单出口结构而无助于程序的理解和阅读。如果必须使用 EXIT 语句时，则最好在循环开始之前加注释以说明之。

最后我们给出 WHILE 循环语句的语法图（见图 5.4）。

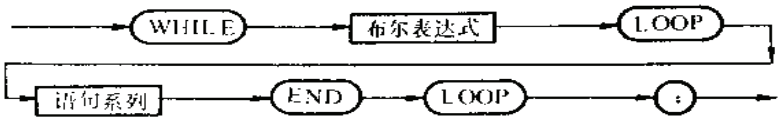


图 5.4

§5-6 FOR 循环语句

对于 LOOP 和 WHILE 循环语句，在循环之前，不能预知循环的执行次数。本节将介绍 Ada 语言中最后一种循环形式——FOR 循环语句，它可用来处理给定循环次数的情形。

例 5-6 设有说明：

```
sum: integer := 0;
```

下列循环语句将 1 至 100 的整数依次相加：

```
FOR count IN 1 .. 100 LOOP
    sum := sum + count;
END LOOP;
```

标识符 count 称为"循环参数"，介于 LOOP 与 END LOOP 之间的语句将被执行 100 次，循环参数 count 依次取值为 1 至 100。

不必说明循环参数的类型，因为其类型由后面的范围表达式隐含说明了。第一次作循环时，循环参数 count 取范围的下界值 1，以后每进行一次循环之前就自动增加 1，直到达到终值 100 为止才跳出循环。试图用赋值语句或其它方式改变一个循环参数的值是非法的。一旦退出循环，循环参数 count 就不存在了，因为它是由循环隐含说明的。因此，无法从循环外部读到 count 的终值。

例 5-7 设有一个班级，每个学生都参加了三门考试，每门考试成绩都在 90 分以上才算优秀，平均分数在 60 分以上才算及格。数据是依次给出的，即每组三个数表示相应的学生的三门考试成绩，在给出考试成绩之前先给出班级人数。要求统计该班考试及格和考试优秀的人数。

对于本例，仍采用逐步求精的方法设计算法：

```
对计数器进行初始化
读入班级人数
LOOP 对每个学生进行循环
    处理该生的考试成绩
    根据学生成绩，更新相应的计数器
END LOOP
写出结果
```

算法中的大多数语句容易用 Ada 语言实现，只有语句"处理学生的考试成绩"需要进一步扩展。

一个问题的解法通常不止一个，一个较好的算法是：

```
LOOP  3 次
    读入一门成绩
    如果成绩小于或等于 90，就置优秀标志为 false
    累计目前的总分
END LOOP
```

计算平均分数

对上述算法稍加修改，就可以处理每个学生参加多门考试的情形。采用该方法所编制的程序如下。

程序 5-7

```
WITH student_io; USE student_io;
PROCEDURE statistics IS
    exam_results, class_size, exam_total: integer;
    merit_flag: Boolean;
    merit_passes, ord_passes, fails :integer := 0;
    no_of_exams: CONSTANT integer := 3;
BEGIN
    --statistics student's exam results
    (统计学生成绩)
    put("Number of class?");
    get(class_size);
        FOR student IN 1 .. class_size LOOP
        --deal with the next student's exam results
        (处理下一个学生的成绩)
        merit_flag := true;
        exam_total := 0;
        FOR exam IN 1 .. no_of_exmas LOOP
        get(exam_result);
        IF exam_result <= 90 THEN
        merit_flag := false;
        END IF;
        exam_total := exam_total+exam_result;
    END LOOP;
    --if merit flag still true all results > 90
    (如果所有成绩都高于 90 分，meri_flag 的值为 true.)
    IF merit_flag THEN
    merit_passes := merit_passes+1;
    ELSIF exam_total/no_of_exams > 60 THEN ord_passes := ord_passes+1;
    ELSE
    fails := fails+1;
    END IF;
    --the counts give information on the students so far
    (统计至当前学生情况)
    END LOOP;
    put("Number of merit passes =");
    put(merit_passes, width =>1); new_line;
    put("Number of ordinary passes =");
    put(ord_passes, width=>1); new_line;
    put("Number of fails =");
    put(fails, width =>1); new_line;
END statistics;
```

这个程序中含有两个 FOR 循环语句，一个称为外循环，另一个称为内循环，即循环相互嵌套。

如果某班人数为 30，则必须执行外循环 30 次，而每次执行外循环时，都遇到：

FOR exam IN 1 .. no _ of _ exams LOOP

在更新某个计数器之前，需执行内循环中的语句 3 次，所以整个程序共读入并处理了 30×3=90 个考试成绩。

上面所述的 FOR 循环的范围都是按升序取值的。FOR 循环还有一种形式，就是循环参数按降序取值，但范围本身总是按升序写。例如语句：

```
FOR ch IN REVERSE 'a' .. 'z' LOOP
    put (ch);
END LOOP;
```

将 26 个英文字母按由 z 到 a 的顺序打印出来。

FOR 循环语句的一般形式可由语法图图 5.5 表示。

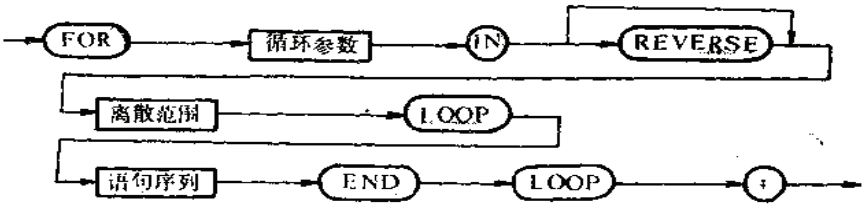


图 5.5

FOR 循环中的范围必须是离散范围，因此只有具有前驱后继的离散类型才可以作为离散范围，所以实数类型是不能作为 FOR 循环的离散范围的。

离散范围的上下界可以是表达式，在这种情形下，在开始循环之前，先计算表达式的值。假设 a、b、c、d 已有相应的说明，则语句：

FOR X IN a+b .. c * d LOOP

在进入循环之前，先计算表达式 a+b、c * d 的值。如果在执行循环时，变量 a、b、c、d 的值被改变，这对循环执行的次数并没有任何影响。此外，如果最初的下界大于上界，则得到一个空的离散范围，此时就不执行循环。

例 5-8　设有说明：

```
TYPE month IS (Jan, Feb, March, April, May, June, July, Aug, Sep, Oct, Nov, Dec)
    no _ of _ visitors: integer;
    income: integer := 0;
    summer _ price: CONSTANT integer := 3;
    other _ price: CONSTANT integer := 2;
```

根据博物馆的门票张数，就可以计算一年的门票收入，其中夏季的门票价格比其它季节贵。程序段是这样的：

```
FOR this _ month IN Jan .. Dec LOOP
    get(no _ of _ visitors);
    IF this _ month IN June .. Sep THEN
        income := income+no _ of _ visitors * summer _ price;
    ELSE
        income := income+no _ of _ visitors * other _ price;
    END IF;
END LOOP;
```

上面所写的离散范围不是最好的方法，还可以用属性 first 及 last 来指明范围的上界及下界，这样就可以写成：

```
FOR this_month IN month'first .. month'last LOOP
```

甚至，可以只写类型的名字：

```
FOR this_month IN month LOOP
```

同样，如果需要在循环中依次考虑所有的字符，则可写成：

```
FOR ch IN character LOOP
```

同 WHILE 循环语句一样，在 FOR 循环语句内也可出现出口语句，但不提倡在 FOR 循环内使用 EXIT 语句，因为使用 EXIT 语句会使程序的控制流不清晰而影响程序的可读性。

§5-7 CASE(情形)语句

当有更多的可能途径供选择时，通常可用 CASE(情形)语句来处理。CASE 语句允许我们根据一个表达式的值在几个语句序列中选择其一执行。例如，对于门票收入的例子，可以用如下的程序来统计年门票收入情况。

例 5-9

```
FOR this_month IN month LOOP
        get(no_of_visitors);
    CASE this_month IS
        WHEN June .. Sep =>
            income := income + no_of_visitors * summer_price;
        WHEN Jan .. May | Oct .. Dec =>
            income := income + no_of_visitors * other_price;
    END CASE;
END LOOP;
```

其执行情况为：读入 no_of_visitors 的值后遇到 CASE 语句，则先计算介于保留字 CASE 与 IS 之间的表达式，然后决定应执行那个语句序列。若 this_month 的值落在离散范围 June .. Sep 中，就执行语句：

```
income := income + no_of_Visitors * summer_price;
```

若落在范围 Jan .. May 或范围 Oct .. Dec 中，就执行语句：

```
income := income + no_of_visitors * other_price;
```

this_month 所有可能取的值(共 12 个)都由 WHEN 之后的选择所指明，选择必须为常量表达式或常量离散范围；若有几个常量表达式或常量离散范围出现在选择中，则它们之间需用符号"|"作间隔，符号"|"应读作"或"。自然，根据严格类型原则，选择必须与 CASE 之后的表达式是同一类型。

例 5-10 作为 CASE 语句的另一个例子，首先设有说明：

```
leap_year : boolean;
which_month : month;
no_of_days : integer;
```

并假定这些变量都已被赋于适当的值，利用下列 CASE 语句就可计算某月的相应天数。

```
CASE which_month IS
    WHEN April |June|Sep|Nov| =>
```

```
        no_of_days := 30;
    WHEN Feb =>
        IF leap_year THEN
        no_of_days := 29;
        ELSE
            no_of_days := 28;
        END IF;
    WHEN Jan|March|May|July|Aug|Oct|Dec| =>
        no_of_days := 31;
END CASE;
```

表达式 which_month 的每个可能的值都提到一次并且只有一次。就本例而言，类型 month 只有 12 个值，都只在保留字 WHEN 之后的选择中出现一次。当类型的取值范围较大时，可用保留字 OTHER 来指明，但 OTHERS 必须在最后说明，如上例中的最后一个选择可用 others 改写为：

```
WHEN OTHERS =>
no_of_days := 31;
```

但我们认为原来的表示方式更好些，因为可以一目了然地看到哪些月份的天数是 31 天。

下面用语法图来表示 CASE 语句的规则。

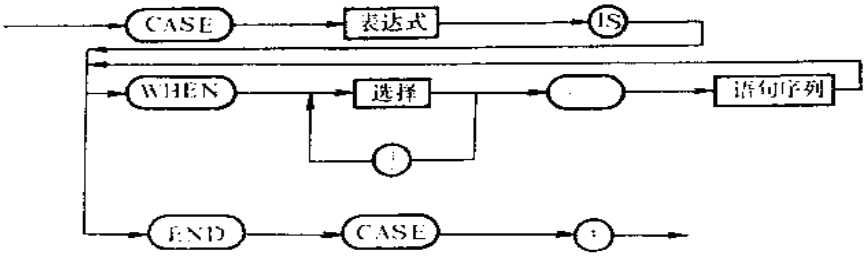


图 5.6

其中，选择可定义为：

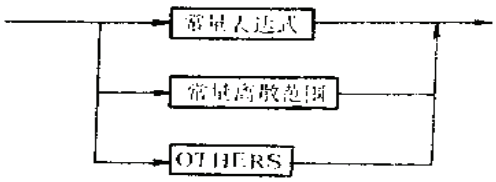


图 5.7

练习

1. 仔细阅读下列程序，当数据为 21、7、28、14 时，考察其执行过程。

```
WITH student_io; USE student_io;
PROCEDURE large IS
    largest, number: integer;
    char : character;
BEGIN
```

```
        get(largest); get(char);
    WHILE char/= ';' LOOP
            get(number );get(char);
            IF number>largest THEN
                largest := number;
            END IF;
        END LOOP;
        put("The largest number is");
        put(largest); new_line;
    END Large;
```

2. 在 5.1 节中，曾用 IF 语句来确定一个字符是大写还是小写英文字母，或者是数字还是其它字符，请用 CASE 语句改写它。

3. 设 ch 已说明为 character 类型，指出下列语句执行的结果。

```
ch='a';
WHILE ch <='z' LOOP
    put(ch);
    ch := character'succ(ch );
END LOOP;
```

若用 FOR 循环，则结果如何？

4. 在 Ada 语言中，每种循环形式都可以使用一个或几个 EXIT 语句，请问那种循环方式较好？

5. 对如下几个题，先设计算法，然后再编制相应的程序：

(i) 读入 3 个数，求最小者；

(ii) 读入 100 个数，求最小者；

(iii) 读入一个大于零的整数 n，求 1+1/2+…1/n；

(iv) 读入一个大于零的整数 n，求 1-1/2+1/3-…1/n；

6. 假设你有存款 500 元，每月收入 100 元，支出 110 元，每月的利息为 2.5%，试编一个程序，计算过几个月你将花完所有的存款。

7. 改写练习 6，使得程序能处理任何初值、利息，以及月收入、月支出。

8. 指出下列程序的运行结果。

```
WITH student_io;      USE student_io;
PROCEDURE mowing IS
BEGIN
    FOR verses IN 2 .. 10 LOOP
        put(verses, width=>2);
        put("men went to mow, went to mow a meadow,");
        new_line;
        FOR men IN RESERVE 2 .. verses LOOP
            put(men, width=>2);
            put("men"); new_line;
        END LOOP;
    put("1 man and his dog, went to mow a meadow.");
    END LOOP;
END mowing;
```

第六章　输入与输出

本章将介绍 Ada 程序与外部的一个界面，即输入与输出。Ada 与其它语言不同，它没有任何用于输入与输出的内在特性，而是通过子程序（包括过程与函数）来实现的。关于子程序将在下章介绍，本节只介绍 put、get、end _of _line 等用于输入与输出的预定义过程或函数。

§6-1　运行结果的输出

在前几章中，我们已经使用了 put、new _line 等标识符来输出程序的运行结果，本节将作更详细的介绍。

我们总假定，程序是由交互式终端的键盘输入计算机的，运行结果由 VDU 显示或在打印机上输出。为获得明了、清晰可读的结果，需要控制运行结果的输出格式。

标识符 put 可用于输出字符、字符串、数值等，参数 width 可用于控制整数的输出格式。

6.1.1　整数输出

设有说明：

```
number : =integer : =17;
```

则执行下列语句

```
put("value="); put(number, width=>1); new _line;
put("value="); put(number, width=>2); new _line;
put("value="); put(number, width=>4); new _line;
```

的结果为：

```
value=17
value=17
value=  17
```

由于参数 width 之后的数字不同，输出同一整数 17 的格式略有不同。这是因为：参数 width 后面的整数是用于指明输出整数所占的宽度或格数。例如语句：

```
put("value="); put(number, width=>3);
```

中的参数"3"表示 number 之值可占 3 格。

如果参数值小于输出整数的位数，则自动补充所需的格数，因此一般情况下，可设置参数 width 之值为 1，这样可以保证所输出的整数之前没有多余的空格。如果 width 参数值比要输出的整数的位数大，则在该数之前补足若干空格。当然，输出整数时也可不说明 width 参数，此时将分配足够的格数以便能输出值 integer' last，并在其前加一空格（用于输出正负符号，一般称为符号位置）。

对参数 width 的说明可以采用简写的方式。下列两个语句是等效的：

```
put(number,2);
put(number,width=>2);
```

但是，我们不提倡用简写的方式，因为第二个语句比第一个语句更明确。

6.1.2 实数输出

put 同样可以用于实数的输出，不过比整数的输出更为复杂一些。通常，实数的输出格式都采用浮点形式。例如：

```
amount: real := 12345.678;
```

则语句

```
put("number=");put(amount);
```

的输出形式为：

```
nnmber=  1.2345678E+04
```

该数之前留有一空格，它是留作输出实数的符号。如果是正数就输出一空格，如果是负数就在该处输出减号"-"。在小数点之前为一非零数字，而小数点之后的数字个数小于定义类型 real 时所指的有效数字的位数。最后为指数部分，E 后面总是有一符号——加号或减号，其后是两位数字。

6.1.3 输出格式控制参数 fore、aft、exp

同样，可以利用参数 fore、aft、exp 来控制实数的输出格式。参数 fore 用于控制小数点前的输出位数；而小数点之后数字位数由参数 aft 控制，参数 exp 可控制指数的数字位数。例如有一个实数 12345.678＝1.2345678×10⁴，则下列语句：

```
put("number="); put(amount, fore=>3); new_line;
put("number="); put(amount, aft=>5); new_line;
put("number="); put(amount, fore=>1,aft=>3, exp=>2); new_line;
put("number="); put(amount,1,3,2); new_line;
```

输出形式为：

```
number=  1.2345678E+04
number=  1.23457E+04
number=1.235E+04
number=1.235E+04
```

对于第一个语句，由于 fore 参数之值为 3，而小数点之前只有一位数字，所以在最前面有两个空格。参数 aft 未说明，所以取足够的格数将 2345678 全部输出。

第二个语句中说明了参数 aft，指定小数点之后的数字个数为 5，这样就对多余的数进行四舍五入。

第三、第四语句是等效的，其中一个是简写形式。参数 fore、aft、exp 之值分别为 1、3、2。同样需对多余的小数部分的数进行四舍五入。

如果使用 put 输出实数时，未说明参数 fore，则在数的前端留有一个符号位。对于正数而言，则留一空格，而负数则输出一个减号。

如果希望所输出的实数不带有指数部分，则只要置参数 exp 之值为零。例如执行语句：

```
put("number="); put(amount, aft=>2,exp=>0); new_line;
put("number="); put (amount, fore=>8,aft=>1, exp=>0);
new_line;
```

其结果为：

```
number=  12345.68
number=  12345.7
```

6.1.4 put_line 语句

当 put 语句用于输出字符或串时，就没有什么参数可言了。有时在输出一个串后需要换行，这时可以使用 put_line 语句。下列两个语句是等效的：

```
put_line("Good luck!");
put("Good luck"); new_line;
```

例 6-1　设球的半径从 1 变化至 10，分别计算球的周长、体积及表面积。

下面的程序用列表的方式输出结果，这样可以清楚地看到这些值随半径的增加而变化的情况：

程序 6-1

```
WITH student_io; USE student_io;
PROCEDURE tabulate IS
    pi:CONSTANT real := 3.141_59;
    rr:real;
BEGIN
    --tabulate properties of sphere as raduis changes
    (列出不同半径的球的体积、表面积等)
    put_line ("raduis      circumference      volume      surface");
    put_line ("                                          of sphere");
    FOR r IN 1..10 LOOP
        put(r, width=>6);
        rr := real(r);
        --write circumfernce
        (输出周长)
        put(2.0 * pi * rr, fore=>5, aft=>3,exp=>0);
        --write volume
        (输出体积)
        put(4.0/3.0 * pi * rr * * 3, fore=>11, aft=>3);
        --write surface area
        (输出表面积)
        put(4.0 * pi * rr * rr, fore=>6, aft=>3);
        new_line;
    END LOOP;
END tabulate;
```

程序运行结果如下：

radius	circumference	volume	surface of sphere
1	6.283	4.189E+00	1.257E+01
2	12.566	3.351E+01	5.027E+01
3	18.850	1.131E+02	1.131E+02
4	25.133	2.681E+02	2.011E+02
5	31.416	5.236E+02	3.142E+02
6	37.699	9.048E+02	4.524E+02
7	43.982	1.437E+03	6.158E+02
8	50.265	2.145E+03	8.042E+02
9	56.549	3.054E+03	1.018E+03
10	62.832	4.189E+03	1.257E+03

这里，两数之间的间隔是由参数 fore 来控制的。显然，用这种方法极不方便，为此可以使用 set _col 来控制输出格式。

6.1.5 set _col 语句

设 position 为整数类型，则语句

```
set _col (position);
```

的执行结果依赖于 position 取的值。当 positin 大于当前的列数，则输出足够的空格使得当前列为 position；如果 postion 等于当前列数，则本语句不产生任何动作；当 position 小于当前列数时，就执行并置当前列数为 1，然后再输出适当的空格使得当前列数为 position。

现在我们用 set _col 语句来改写程序 6-1，改写后的程序的运行结果与原来是一样的。

程序 6-2

```
WITH student _io; USE student _io;
PROCEDURE tabulate IS
    pi: CONSTANT real := 3.141 _59;
    rr: real;
BEGIN
    --tabulate properties of sphere as raduis changes
    (用表列出周长等)
    put("radius");
    set _col(10); put("circumference");
    set _col(25); put("volume");
    set _col(39); put _line("surface");
    set _col(39); put _line(of sphere");
    FOR r IN 1 .. 10 LOOP
        put(r, width=>6);
        rr := real(r);
        --write circumference
        (输出周长);
        set _col(10);
        put(2.0 * pi * rr, fore=>2, aft=>3, exp=>0);
        --write volume
        (输出体积)
        set _col(25);
        put(4.0/3.0 * pi * * rr * * 3, aft=>3);
        --write surface area
        (输面表面积)
        set _col(39);
        put(4.0 * pi * rr * rr, aft=>3);
        new _line;
    END LOOP;
END tabulate;
```

尽管程序比原来的长了，但是表头与相应的数可以互相对齐，使输出的表格更为清晰。

§6-2 student _io 及预定义子程序

Ada 是一种强类型语言，所有的标识符必须被说明之后才能使用。但是前面的程序中已使用的

new _ line、put _ get 以及类型 real、integer、character 等，标识符都未在程序中加以说明，那么它们又是在何处说明的呢？

在 Ada 中，有一个标准库程序包(standard library package)，所有的内部类型 integer、character、string、real 等就是在这个库包里说明的，标识符 get、put、new _ line 等也是在这个库包里说明的。标准库包对所有 Ada 程序都是可访问的，并且是 Ada 语言的一部分，附录三中给出了标准库包的规格。

本书中所使用的输入、输出语句就是在库包 student _ io 中定义的，因此在每一程序的首部都有下面的语句：

```
WITH student _ io; USE student _ io;
```

这样，库包中的输入、输出语句就可被程序访问。

关于程序包的结构以及如何编制程序包，将在以后的章节中介绍，程序包可以容纳许多说明以及如何使用程序包。

本书中的程序包 student _ io 不是 Ada 语言的一部分，但是作为教学而用的 Ada 系统都有一个类似的程序包。当你使用某一 Ada 系统时，必须首先确定它的名字以及与 student _ io 的不同之处。

程序包 text _ io 才是 Ada 语言的一部分，附录中列出了 student _ io 的一些细节。如果手头上没有 student _ io 这样的程序包，只要稍加修改就可以自编一个。

我们已经介绍了 get、put、new _ line 是在程序包 student _ io 中说明的，那么它们究竟是什么呢？事实上，标识符 get、put、new _ line、set _ col 等都是一些过程或函数的名字。过程与函数是 Ada 中两种子程序的形式。由于它们的定义是相当复杂的，因此其细节被隐藏在程序包内，使用者不需了解 get、put 等的具体实施过程，只需了解如何使用它们。

put、get、new _ line、set _ col、put _ line 语句都是"过程调用"(procedure calls)的例子。当执行语句：

```
put(2.0 * pi * * rr);
```

时，其实是调用过程 put；圆括号内的表达式称为参数。而调用过程 new _ line 不需要参数，可只写：

```
new _ line;
```

此外，程序包 stndent _ io 中定义了一些函数，函数不同于过程，调用函数时，它还额外返回一个值。函数调用是表达式的一部分，而过程调用是一个独立的语句。

col 和 end _ of _ line 是两个可访问的函数。函数 col 给出当前列的值，例如语句

```
set _ col(40);
position : =col;
```

的执行情况为：第一个语句将当前列数置为 40；执行第二个语句时，先调用函数 col，它返回当前列数之值并将该值赋给变量 position。函数 col 没有参数，但大多数函数都有参数。

end _ of _ line 是一个布尔函数，如果读完一行信息，则函数 end _ of _ line 返回的值为 true，否则是 false。

作为程序数据而键入的信息和程序输出的结果可以看作是由行组成的字符序列。在 Ada 中，这些行的结构是通过行尾符(line termintor)来组织的。行尾符不是通常意义下的字符，但是它可以通过调用特殊的过程输出和读入，并且可由函数 end _ of _ line 来识别。调用 new _ line 的结果是输出一个行尾符并且置当前列数值为 1。调用过程 skip _ line 也可以读入行尾符，其结果是跳过一行的所有信息然后读入行尾符。

下面举例说明如何使用这些函数和过程。

读入 5 行数据并且输出读入的信息，要求是在每行的前端输出行数，行数与输出信息之间用一空格分隔。假设 char 为字符类型变量，语句序列如下：

```
FOR line_number IN 1 .. 5 LOOP
    put(line_number); put(' ');
    --read and write contents of a line
    (读入并输出一行)
        WHILE NOT end_of_line LOOP;
            get(char); put(char);
        END LOOP;
    --deal with the line terminator
    (处理行尾)
    skip_line; new_line;
END LOOP;
```

我们已经看到，在调用过程 get 和 put 时的参数，有时是字符参数，有时是整数参数，而有时又是实数参数。可以这样使用的原因是，student_io 中定义了多个过程 get 和 put，当使用不同的参数调用过程 get 和 put 时，Ada 系统会自动识别并调用相应的过程 get 和 put。这种同一标识符具有多个含义的情形称之为重载(overload)。重载是一个十分重要但又很复杂的概念，在此不详细介绍了，读者可参阅其它 Ada 语言参考书。除非你是一个有经验的 Ada 程序设计员，否则在初学阶段，最好不要使用重载标识符。

练习

1. 设 n 为从 1 到 20 之间的整数，试用表格的方式表示值 n、n**2、n**3、1/n、1/n**2，请编一程序打印这样的表格。
2. 写一段语句序列输出 95 个可打印字符，并且按 ASCII 表中的顺序打印，每行不超过 8 个字符。

第七章　过程与函数

本章介绍子程序的两种形式——过程与函数的语法规则，以及参数方式等，此外还要介绍局部说明与递归。

§7-1　子程序的重要性

用逐步求精方法编制程序时，首先设计高级算法(top-level algorithm)。高级算法中的每个语句可以视为是对某一子问题的说明，然后再对每个子问题设计相应的高级算法，该算法中的语句，又可以看成是对一些更为简单子问题的说明。总之，解题的基本方法是：将较大的问题分解为若干容易求解的、相对独立的子问题。

最简单的 Ada 程序由一个过程组成，而将大型问题分解成若干子问题的方法体现了编制大型程序的基本原则。对每个子问题可以用适当的过程或函数来处理，过程与函数是 Ada 中子程序的两种形式。

首先用简单的例子来说明如何定义和使用子程序。重新考虑第三章中的例子，该题要求画一个带边界的三角形，其算法为：

画一边界
画一三角形
画一边界

现采用过程 draw _ border 改写原来的程序。

例 7-1

程序 7-1

```
WITH student _ io; USE student _ io;
PROCEDURE triangle IS
    spaces :CONSTANT string := "          ";
    PROCEDURE draw _ border IS
        border: CONSTANT string := "* * * * * * * * * * * * * * * * * * * * * * * * * * * * * *";
    BEGIN
        --draw a two _ line border surrounded by blank lines
        (画两条边界)
        new _ line;
        put _ line(border); put _ line(border);
        new _ line;
    END draw _ border;
BEGIN
    --print a bordered triangle
    (打印带边界的三角形)
    draw _ border;
    put(spaces); put _ line("* *");
    put(spaces); put _ line("* * *");
```

```
    put(spaces); put_line("* * * * *");
    draw_border;
END triangle;
```

在这个程序中，有两个过程 triangle 和 draw _ border，其中过程 triangle 作为主程序。在主程序的说明部分里，有一个称为 draw _ border 的过程体(procedure body)的说明，这个过程的作用就是画两条由星号组成的直线，即边界。

现在来看一下程序的运行情况，当变量在主程序的说明部分里被说明及初始化后，就从主程序的第一个语句开始执行，即调用过程 draw _ border，从而进入过程 draw _ border 并执行其中的语句，执行结果是输出两行"*"号。

当调用过程 draw _ border 完毕后，就跳出该过程体并返回调用过程处的下一语句，于是输出一个三角形。接着又遇到语句：

```
draw_border;
```

所以第二次调用过程 draw _ border，再次进入过程 draw _ border 并执行过程体中的语句，执行完后返回主程序，此时已到达主程序的结尾，于是运行结束。

这里，我们将画两行星号的若干语句组成一个过程，其功能就是画一条边界。如果需要画边界，只需调用这个过程。当过程编制完毕并且测试其正确无误之后，就不必了解它是如何实施的，正如不必了解预定义过程 get 和 put 是如何读入、输出信息一样，只要在需要的时候调用它即可。

可以这样认为，子程序是编制大型程序的一个重要手段。将高级算法中某一些子问题编制成子程序，当这些子程序通过测试后，在编制主程序时只需调用这些子程序就可以了，而这些子程序可由不同的程序员同时进行编制和测试。

§7-2 局部说明

上一章中已经介绍了变量、常量、类型及子程序体是如何在主程序的说明部分中被说明的，同样，它们也可以在过程的说明部分里加以说明。例如在过程 draw _ border 中有一个常量 border 被说明。

这是我们所遇到的第一个局部说明(local declaration)的例子。由于常量 border 是在过程 draw _ border 内说明的，所以它只在该过程的内部才有定义，故称之为局部说明。主程序是不能识别常量 border 的，即主程序是不能使用标识符 border。

但是，由于常量 spaces 是在主程序中说明的，因此不仅主程序可以使用标识符 spaces，而且也可以在过程 draw _ border 内使用。如果标识符是在主程序中说明的，则在其包含的内部过程中是可见的，并且对该过程而言称之为全局的或非局部的(global，non-local)。

如果标识符 border 是在主程序内说明的，则 border 对过程 draw _ border 是可见的，即可以在其内部使用。但是尽可能使用局部说明标识符，以使标识符的说明与使用保持一致，这样有利于保证过程的独立性。当一个程序的各个部分是相对独立时，这些部分可以单独编制和调试，也可以重复使用程序中的一些过程来组织另一个程序。所以，应尽量减少使用全局标识符。

只要标识符在使用之前已被说明，则类型、常量、变量的说明次序是无关紧要的，但是必须在任何子程序体说明之前加以说明。

§7-3 参　　数

程序 7-1 中所使用的过程 draw _ border 很不灵活，因为它只能打印固定宽度的边界，如果使用参数可以使过程 draw _ border 更灵活实用，一行中的星号数可由参数指定。

例 7-2

程序 7-2

```
WITH student _ io; USE student _ io;
PROCEDURE triangle IS
    spaces :CONATANT string := "          ";
    PROCEDURE draw _ border(width :IN integer) IS
    BEGIN
        --—draw a two-line border surrounded by blank lines
        --—there are width asteriks in each border line
        (画两条边，星号 * 数由 width 确定)
        new _ line;
        FOR border _ line IN 1 .. 2 LOOP
          --—write width asterisks
            (输出 width 个 * )
          FOR asterisks IN 1 .. width LOOP
            put(' * ');
          END LOOP;
          new _ line;
          END LOOP;
      new _ line;
    END draw _ border;
    BEGIN
        --—print a bordered triangle
        draw _ border(25);
        put(spaces); put _ line("  * ");
        put(spaces); put _ line(" * * * ");
        put(spaces); put _ line("* * * * * ");
        draw _ border(25);
    END triangle;
```

该程序的运行结果与程序 7-1 是一样的。在 draw _ border 过程的定义中，有一个 width 形参 (formal parameter)，其类型为 integer，保留字 IN 指明了参数的方式，IN 表示这个参数是用来向过程传递信息的，以后还会遇到参数的其它方式。

调用过程 draw _ border 时，必须给形参 width 一个实参值，如在过程调用"draw _ border(25);"中，数值"25"就是实参。一旦给形参一个实参值，则在执行过程时，该值是不可改变的。

现在来说明该程序的执行情况，首先执行主程序的第一语句：

```
draw _ border(25);
```

从而进入过程 draw _ border，实参 25 被送至过程内，并将该值赋给相应的形参 width。接着执行过程中的语句，其结果是先输一个空行，相互嵌套的循环输出两行星号，每行的星号数都是 25 个，然后再输出一个空行。

过程调用完毕后，就由过程 draw _ border 返回主程序，于是打印一个三角形。然后再调用过程 draw _ border，打印第二个边界后又返回主程序，至此运行结束。过程 draw _ border 的这种新形式，其调用结果依赖于实参值。尽管 draw _ border 是用不同的方式输出边界的，但调用过程时不必了解这些细节，知道如何调用及其作用就足够了。

从表面上看，带参数的过程并不比不带参数的过程优越多少，并且要复杂得多。但在下面这个例题中就体现出带参数过程的优点。

例 7-3 打印一个任意行的三角形，三角形由星号组成，并且在其左边留有 10 个空格；三角形被由星号组成的上下边界围在中央。

采用逐步求精方法解题，该题的最初算法为：

给定三角形所需的行数
画一适当的边界
画出相应的三角形
画一适当的边界

首先考虑三角形的尺度，如果三角形共有占 size 行，则三角形之底应有(2 * size－1)个星号。由于三角形的两侧各有 10 个星号，所以边界的长度应为"三角形之底＋20"。

由上述的分析，算法成为：

给定三角形所需的行数 size
计算底边的长度 triangle _ base
画一个长度为"20＋triangle _ base"的边界
画一个 size 行的三角形
画一个长度为"20＋triangle _ base"的边界

因为已经有了一个带参数的过程和 draw _ border，它可以画任意长度的边界，因此只需考虑子问题"画一个 size 行的三角形"的求解。

下面就来考虑这个子问题。三角形的第一行打印一个星号，以后每行的星号数等于前行星号数加 2 此外还需算出最初的空格数。由于每行上的前导空格数比上一行的前导空格数少 1，因此对该子问题有如下的算法：

置星号数为 1
对空格计数器 space _ counter 初始化
LOOP size 次
　打印相应的空格数
　打印相应的星号数
　space _ counter－1
　星号计数器 num _ asterisks 加 2
　换行
END LOOP

这个算法可用一个带参数的过程来实现，然后再组织主程序，最终的程序如下所示。

程序 7-3

```
WITH student _ io; USE student _ io;
PROCEDURE triangle IS
    triangle _ base, triangle _ size : integer;
```

```
    PROCEDURE draw_border(width : IN integer) IS
    BEGIN
        --draw a two--line border surrounded by blank lines
        --there are width asterisks in each border line
        (画两条边,星号*数由 width 决定)
        new_line;
        FOR border_line IN 1 .. 2 LOOP
            --write width asterisks
            FOR asterisks IN 1 .. width LOOP
                put("*");
            END LOOP;
            new_line;
        END LOOP;
        nwe_line;
    END draw_border;

    PROCENURE draw_triangle(number_lines: IN integer) IS
        space_counter: integer := 10+number_lines-1;
        num_asterisks:integer := 1;
    BEGIN
        --draw triangle on number_lines lines
        (画三角形的第 number_lines 条线)
        FOR lines IN 1 .. number_lines LOOP
            --write leading spaces (输出空格)
            FOR space IN 1 .. space_counter LOOP
            put(' ');
            END LOOP;
            --wirte asterisks(输出*号)
            FOR asterisks IN 1 .. num_asterisks LOOP
                put('*');
            END LOOP;
            new_line;
            space_counter := space_counter-1;
            num_asterisks := num_asterisks+2;
        END LOOP;
    END draw_triangle;
BEGIN
    --print a bordered triangle on triangle_size lines
    put("How many lines in the triangle?"); get (triangle_size);
    triangle_base := 2 * triangle_size-1;
    draw_border(20+triangle_base);
    draw_triangle(triangle_size);
    draw_border(20+triangle—base);
END triangle;
```

如果希望打印如下图形：

```
                    * * * * * * * * * * * * * * * * * * * * * * * * * * *
                    * * * * * * * * * * * * * * * * * * * * * * * * * * *
```

```
                    *
                  * * *
                * * * * *
        * * * * * * * * * * * * * * * * * * * * * * * * *
        * * * * * * * * * * * * * * * * * * * * * * * * *
```

只需在执行上述程序时，输入行数 3。

主程序是很短的，而且与高级算法很相似。一般都用子程序来实现高级算法中的细节。由于子程序的名字都是具有一定意义的标识符，因此只要阅读主程序，就可以对程序的目的略知一二。

至此，我们已介绍了几种过程，现在对过程作一总结。子程序可由语法图来表示(见图 7.1)。

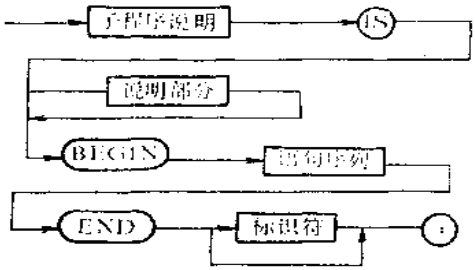


图 7.1

其中，对于过程而言，子程序说明图 7.2 的形式。

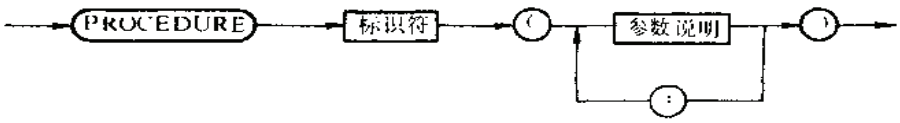


图 7.2

§7-4 参数调用的位置记号法、命名记号法及缺省参数

7.4.1 位置记号法

程序 7-1、7-2、7-3 中的过程 draw_border 都只能打印一个星号组成的边界，现在修改该过程，使之能打印由任何可打印字符组成的边界。这个过程体可写为：

```
PROCEDURE draw_border(width: IN integer; ch:IN character) IS
BEGIN
    --draw a two-line border surrounded by blank lines
    --there are width characters in each border line
    new_line;
    FOR chars IN 1 .. 2 LOOP
        -- write width characters
        FOR border_line IN 1 .. width LOOP
```

```
            put(ch);
        END LOOP;
        new_line;
    END LOOP;
    new_line;
END draw_border;
```

这个过程中有 width、ch 两个形参，其方式都是 IN。调用该过程时，必须有两个实参。第一个实参与第一个形参 width 相对应，并且其类型须为 integer；第二个实参与第二个形参 ch 相对应，所以其类型必须为 character。这就是参数调用的位置记号法。例如，过程调用语句：

```
draw_border(25,%);
```

中，width 之值为 25，ch 之值为字符%，其执行结果如下：

```
%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%
```

7.4.2 命名记号法和缺省参数

对于形参与实参相匹配的位置方式，在过程调用时可以使用命名形式显式说明形参的实参，这就是命名记号法。例如：

```
draw_border(width=>25, ch=>'%');
```

用命名形式调用过程时，参数的次序可以是任意的，因此上述语句也可写成

```
draw_border(ch=>'%',width=>25);
```

但是对于位置形式，不可以写成：

```
draw_border ('%',25);
```

位置和命名形式可以混合使用，但在这种情况下，参数的次序是很重要的，位置方式必须放在前面。我们可以写：

```
draw_border(25, ch='%');
```

但是不能写：

```
draw_boder(width=>25, '%');
```

事实上，我们已经遇到过命名记号这种形式。如使用预定义过程 put 时，就曾用命名的方式来控制数值的输出格式。例如，设 num 是 real 类型，可以有如下的语句：

```
put(num, fore=>5, aft=>6, exp=>2);
put(num, fore=>4);
put(num);
```

但是过程 put 中的一些形参并未指明，这是为什么呢？正如变量说明时可以对变量进行初始化，IN 方式的参数也可以给一个缺省(default)的初值，其语法规则可由语法图来表示(见图 7.3)。

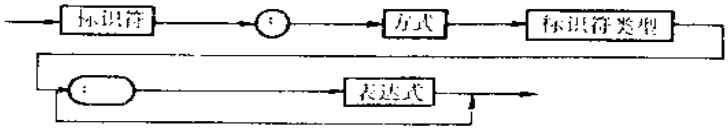


图 7.3

如果希望过程 draw _ border 一般打印由星号组成的边界，同时在需要的时候也可以打印由其它字符组成的边界，则可用如下的说明：

```
PROCEDURE draw _ border(width: IN integer; ch IN character := '*') IS
    ...
END draw _ border;
```

这时，语句

```
draw _ border(25,'%');
```

的作用是打印由百分号"%"组成的边界；但由于形参 ch 已有一个缺省值，所以下面这两个语句的作用是一样的：

```
draw _ border(25);
draw _ border(25,'*');
```

在预定义过程 put 中，形参 fore、aft、exp 都有各自的缺省值；过程 new _ line 也有一个缺省值，其值为 1。如果需要换 3 行，则可以用语句：

```
new _ line(3);
```

过程调用时，如果省略实参，则该实参之值为缺省值，而其后的参数必须用相应的命名记号形式。对于过程 put 而言，由于需输出的值放在最先，所以可以使用位置记号形式；但是由于参数说明的次序不易记住，所以对其它参数最好使用相应的命名记号方式。

§7-5 函　　数

本节介绍子程序的另一种形式——函数。过程调用是一个语句，而函数不同于过程，它还额外返回一个值，因此函数调用是表达式的一部分。第六章中已经讲述了如何使用预定义函数 col 和 end _ of _ line。下面讨论如何说明函数体。

例 7-4 假设已有枚举类型 day 的说明：

```
TYPE day IS(sun, mon, tues, wed, thurs, fri, sat);
```

函数 next _ day 的作用是计算下一天为星期几，其说明如下：

```
FUNCTION next _ day (this _ day: IN day) RETURN day IS
BEGIN
    IF this _ day=day'last THEN
        RETURN day'first;
    ELSE
        RETURN day'succ(this _ day);
    END IF;
END next _ day;
```

函数返回值的类型必须在函数体里说明，且是在保留字 RETURN 之后指明的。本例中的类型为 day。

RETURN 语句是执行函数的终止标志，RETURN 语句的形式如图 7.4 所示：

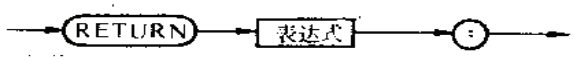


图 7.4

其中，表达式的值返回到调用函数的程序中，表达式之值的类型就是函数体中给定的类型。在

函数中，形参只能用 IN 方式，这时，形参须为一常量，且只允许读相应实参的值。

设 today、tomorrow 是类型为 day 的变量，并且 today 已赋值 thurs，则执行语句

```
tomorrow := next_day(today);
```

的结果是将函数返回的值 fri 赋给变量 tomorrow。其具体执行情况为：首先计算右边的表达式，由于该表达式是一个函数调用，于是进入函体体并将值 thurs 传递给参数 this_day，然后执行函数体内的语句，由于 thurs/=day'last，所以返回的值为 day'succ(this_day)，即值 fri；至此脱离函数体，返回函数调用处并将返回的值 fri 赋给变量 tomorrow。

例 7-5 试编制一程序，要求将读入的码、英尺、英寸转换成米并打印出来。转换由一个函数来完成，这个函数接受三个整数，返回一个实值，完整的程序如下。

程序 7-5

```
WITH student_io; USE student_io;
PROCEDURE convert IS
    yards, feet, inches: integer;
    metres: real;

    FUNCTION no_of_metres(yds, ft, ins: IN integer) RETURN real IS
        inches:integer;
        inches_to_metres: CONSTANT real := 0.02540;
    BEGIN
        --convert yds, ft and ins to metres(把码、英尺、英寸转换米)
        inches := yds * 36 + ft * 12 + ins;
        RETURN real(inches) * inches_to_metres;
    END no_of_mertes;
BEGIN
    --read number of yards, feet and inches(读入初值)
    put("yards?"); get(yards);
    put("feet?"); get(feet);
    put("inches?"); get(inches);
    --calculate and print corresponding number of metres(计算并打印转换值)
    metres := no_of_metres(yards, feet, inches);
    put("number of metres=");
    put(metres, aft=>3, exp=>0);
    new_line;
END convert;
```

请注意，在这个程序中，一个 integer 类型变量 inches 是在函数中说明的，而另一个 integer 类型变量 inches 是在主程序中说明的，它们是两个完全不同的变量。在主程序中使用的 inches 是指主程序中所说明的 inches，而函数体内使用的 inches 是指在函数中所说明的 inches。

至此，我们已介绍了函数与过程，事实上，函数体与过程体的定义是子程序说明的不同形式。过程体的说明已在上一节中讨论了，下面给出函数体的语法图（见图 7-5）。

§7-6 参数方式——OUT 和 IN OUT

前几节，已经介绍了参数方式 IN，它用于向子程序传递信息。这种参数的值在程序执行过程中

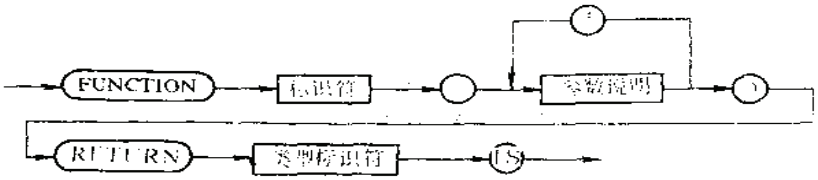


图 7.5

始终保持常量。但是有时参数传递给子程序的信息在运行过程中是要改变的，并且改变后的信息又要传回到调用该子程序外。这时，就需把参数方式设置为 IN OUT 方式。本节将介绍参数的两种方式，即 OUT 和 IN OUT。

例 7-6 试编一程序，将读入的两个实数按由小至大的顺序打印出来。

程序 7-6

```
WITH student_io; USE student_io;
PROCEDURE order IS
    smaller, larger:real;
    PROCEDURE swap(first, second:IN OUT real) IS
        old_first: real;
    BEGIN
        --swap the values of the two parameters
        (交换两个参数)
        old_first:=first;
        first:=second;
        second:=old_first;
    END swap;
BEGIN
    --read and order two numbers(读入两数并排序)
    get (smaller); get (larger);
    IF smaller>larger THEN
        swap(smaller, larger);
    END IF;
    put("The ordered numbers are:")
    put(smaller); put(larger);
    new_line;
END order;
```

程序中，有一个过程 swap，这个过程中有两个方式为 IN OUT 的参数及一个局部变量 old_first。程序运行时，首先读入两个数，并将它们分别赋给变量 smaller 及 larger，如果 smaller 大于 larger，就调用过程 swap。

对于方式为 IN OUT 的形参，也必须有同类型的实参相匹配。调用过程 swap 时，IN OUT 形参作为局部变量，已经被初始化，即相应的实参值已赋给了这些变量，因此，first、second 的初值分别是实参 smaller 和 larger 的值。执行过程 swap 中的语句，使 first、second 之值互换，然后脱离过程。

对于 IN OUT(以及 OUT)参数，其实参必须是变量，因为调用过程时，实参的值要被更新。而 IN 参数，仅仅是向过程传递信息，所以实参可以是表达式。如果希望过程能改变实参的值，则必须用 IN OUT 或 OUT 方式的形参。

开始执行过程时，OUT 方式的参数是未被定义的(undefined)，在过程的运行中，必须对其赋值。脱离过程时，这个值就转给了相应的实参。注意，即使 OUT 方式的参数已被赋值，它也不能出现在表达式中，只有当对其赋值时才能在过程内出现。

总而言之，使用 IN 参数是向过程传递信息，其实参可以是表达式；而 OUT 参数是用于接收一个过程的信息，其实参必须是变量。对于 IN OUT 参数，它首先向过程传递信息，然后再接收返回的信息。同 OUT 参数一样，其实参也必须为变量。如果未指明参数的方式，就是指隐含方式 IN。

在下面的过程 read _ pos 中，使用了一个 OUT 方式的参数，并且对读入的数加以限制，只有正数才被过程接收，否则显示出错误信息并提示用户重新输入。

```
PROCEDURE read_pos(number: OUT real) IS
    any_value:real;
BEGIN
    -- read numbers until a positive number is obtained
    LOOP
        put("positive number please:");
        get (any_value);
        IF any_value<0.0 THEN
            put_line("error:number must be positive!");
        ELSE
            number := any_value;
            RETURN;
        END IF;
    END LOOP;
END read_pos;
```

这个过程中使用了一个 RETURN 语句，执行 RETURN 语句后，就脱离过程，这是脱离过程的另一方法。由于该过程不返回一个值，所以保留词 RETURN 之后没有表达式。对于函数来说，必须执行一个 RETURN 语句之后才能脱离函数体。

§7-7 应用举例

例 7-7 试编一程序，要求读入一个－999999 至 999999 之间的整数，然后将其用英文的形式输出。例如，将 653 转换成“six hundred and fifty-three”。

首先用逐步求精方法设计该题的高级算法，因此有如下这样一个简单算法：

```
读入一个整数
IF 该数是负的 THEN
    打印负号
    将数转换成正数
END IF
将数用英文形式打印出来
```

现在可以把注意力集中在解题的主要部分，即如何用英文表达一个正整数。首先考虑一个数用英文书写的形式，例如数 234567 用英文表达是：

two hundred and thirty-four thousand five hundred and sixty-seven

注意，从千位数开始前面部分的写法与千位数后面部分的写法是一样的。因此有如下的算法：

```
IF  数大于 999  THEN
    写出从千位数开始前面的部分
    打印"thousand"
END IF
    输出小于一千的部分
```

这样，问题已转化为如何打印 1 至 999 之间的数，此问题的算法为：

```
IF  数大于 99  THEN
    打印百位数
    打印"hundred"
END IF
IF  十位数大于零  THEN
    打印十位数
END IF
如果个位数非零，则打印个位数
```

现在，只需解决如何用英文写出 1 至 9 的数，这个问题是容易的。至此，解题的算法已大致形成，但仍有一些细节问题未解决。例如，10 至 19 之间的数的写法是不规范的，因此需对算法加以修改，但这一部分的修改不影响算法的基本结构。因此上述算法的最后 4 行可改写为：

```
IF  十位数是 1  THEN
    用英文写出最后两位数字
ELSE
    IF  十位数大于 1  THEN
      写出十位数字
    END IF
    如果是非零的话，写出个位数
END IF
```

当算法设计完毕后，就可以用 Ada 实现算法。算法的设计是自顶向下的，但这并不意味着算法实现也是这样。对于大型程序而言，通常先实现子问题的算法，然后分别编译、测试，最后才连接成一个主程序。

对于本例来说，首先编制一个过程用以处理 0 到 9 之间的数，如果数为 0，则不输出任何信息，否则输出相应的"one"、"two"等。下面这个程序可以完成这个测试工作。

程序 7-7

```
WITH student_io; USE student_io;
   PROCEDURE write_number IS
      number,integer;
      PROCEDURE write_digit(digit,IN integer) IS
      BEGIN
         --write number in the range 1 to 9
         (输出数 1~9)
         CASE digit IS
            WHEN 1=> put("one");
            WHEN 2=>put("two");
            WHEN 3=> put("three");
```

```
            WHEN 4=> put("four");
            WHEN 5=> put("five");
            WHEN 6=> put("six");
            WHEN 7=> put("seven");
            WHEN 8=> put("eight");
            WHEN 9=> put("nine");
            WHEN OTHERS =>NULL;
        END CASE;
    END write_digit;
BEGIN
    put("number please"); get(number);
    IF number <0 THEN
        -- deal with negative number
        (处理负数)
        put("minus")
        number := -number;
    END IF;
    IF number<10 THEN
        write_digit(number);
    ELSE
        put("number too large");
    END IF;
  new_line;
END write_number;
```

在 CASE 语句中，当 digit 超过 1 至 9 的范围时，由于不希望产生任何动作，所以用一空语句(NULL)指明。执行一个空语句不会发生任何操作，这样就可以处理 digit 的各种不同情形。

当上述程序调试通过后，就可以考虑算法的其余部分，例如考虑如何处理 1 至 99 之间的数。分段实现算法的方法，通常称为"逐步实现"(incremental inplementation)，其优点是：对于每个阶段，只需测试相对独立的模块。

下面写出了本例完整的程序。

```
WITH student_io; USE student_io;
PROCEDURE write_number IS
    number:integer;

    PROCEDURE write_1_to_999(num:IN integer) IS
        under_100: CONSTANT integer := num REM 100;
        PROCEDURE(wite_digit : IN integer) IS
        BEGIN
            --write number in the range 1 to 9
            (输出 1 到 9)
            CASE digit IS
                WHEN 1=> put("one");
                WHEN 2=> put("two");
                WHEN 3=> put("three");
                WHEN 4=> put("four");
                WNEH 5=> put("five");
```

```
            WHEN 6=> put("six");
            WHEN 7=> put("seven");
            WHEN 8=> put("eight");
            WHEN 9=> put("nine");
            WHEN OTHERS=> NULL;
        END CASE;
    END write_digit;
    PROCEDURE write_tens(tens:IN integer) IS
    BEGIN
        --deal with twenty to ninety
        (处理 20 到 90)
        CASE tens IS
            WHEN 2=> put("twenty");
            WHEN 3=> put("thirty")
            WHEN 4=> put("forty");
            WHEN 5=> put("fifty");
            WHEN 8=> put("eighty");
            WHEN OTHERS =>write_digit(tens); put("ty");
        END CASE;
    END write_tens;

BEGIN
    --wirte numbers in range 1 to 999
    (数的范围:1 到 999)
    IF num>99 THEN
        --write_number of hundreds
        (输出百位数)
        write_digit(num/100);
        put("hundred");
        END IF;
        IF under_100/=0 THEN
            --and required if anything after hundred
            (打印"and ")
            put("and");
        END IF;
        IF under_100 IN 10 .. 19 THEN
            --deal with 10 to 19
        (处理 10 到 19)

        CASE under_100 IS
            WHEN 10=> put("ten");
            WHEN 11=> put("eleven");
            WHEN 12=> put("twelve");
            WHEN 13=> put("thirteen");
            WHEN 15=> put("fifteen");
            WHEN 18=> put("eighteen");
            WHEN OTHERS=>write_digit (unber_100 REM 10);
```

```
                                    put("teen");
            END CASE;
        ELSE
            IF under_100>=20 THEN
                --write tens part(处理十位数)
                write_tens(under_100/10);
            END IF;
                --write digits part
                write_digit(under_100 REM 10);
        END IF;
    END write_1_to_999;

BEGIN
    --write number in the range -999999 to 999999
    (处理 -999999 到 999999)
    put("number please"); get(number);
    IF number <0 THEN
        --deal with negative number
        (处理负数)
        put("minus");
        number := --number;
    END IF;
    IF number >999_999 THEN
        put("number too large");
    ELSIF number = 0 THEN
        --zero is a special case(处理零的情形)
        put("zero");
    ELSE
        IF number >999 THEN
            --deal with number of thousands
            (处理千位数)
            write_1_to_999(number/1000);
            put("thousand");
            number := number REM 1000;
            IF number IN 1 .. 99 THEN
                put("and ");
            END IF;
            --deal with part under 1000
            (处理小于 1000 的部份)
            write_1_to_999(number);
        END IF;
        new_line;
END write_number;
```

注意，程序中有两个过程——write_digit 和 write_tens 是在过程 write_1_to_999 内局部说明的。当然，它们也可以在主程序中说明，但在过程 writ_1_to_999 内说明可以保证过程 write_1_to_999 的独立性。

§7-8 递　　归

子程序也可以调用其自身，这就是所谓的递归调用，递归是一种简单而实用的方法。我们通过一个例子来说明如何使用递归。

例 7-8　计算整数 n 的阶乘 n!，其计算公式为：

$$n! = n \times (n-1) \times \cdots \times 2 \times 1$$

这个公式可写成：

$$n! = n \times (n-1)!$$

即 n! 可由(n－1)! 来定义。利用这个思想来构造阶乘函数 factorial：

```
FUNCTION factorial(n:integer) RETURN integer IS
BEGIN
    --n must be non-negative(n 须非负)
    IF n>1 THEN
        RETURN n * factorial(n-1);
    ELSE
        RETURN 1;
    END IF;
END factorial;
```

在 factorial 函数体内，有一个函数调用 factorial(n－1)。这是一个简单的递回函数。

现在来看一下函数调用：

```
factorial(4)
```

是如何进行的。进入函数时，值 4 被赋给 IN 参数 n，由于 4 大于 1，所以计算表达式"4 * factorial(3)"的值返回的是函数的值。但是表达式中又有一个函数调用 factorial(3)；而计算 factorial(3)又需计算 factorial(2)；计算 factorial(2)时又需计算 factorial(1)，此时不需进一步递归了。factorial(1)返回的值为 1。因此，表达式"2 * factorial(1)"之值为 2，进而表达式"3 * factorial(2)"之值为 6，所以表达式"4 * factorial(3)"之值为 24。于是，调用 factorial(4)返回的值为 24。

递归子程序最终必须终止，否则称为"无限递归"，即递归部分成为死循环，对于许多问题，灵活使用递回过程或函数会达到事半功倍的效果。

§7-9 注意事项

1. 子程序是一个相当重要的概念，在 Ada 中，子程序有两种形式——过程与函数。过程调用是作为一个语句，而函数还额外返回一个值，所以函数调用作为表达式的一部分。

2. 过程与函数都可以有参数，调用时，相对应的形参与实参的类型必须相匹配，参数的指名有三种方式：位置、命名和缺省，混合使用时须注意参数的次序。参数的方式共有三种：IN、OUT 和 IN OUT，请注意每种方式的含义和适用范围。

3. 利用子程序是编制大型程序的一种最基本方法，在子程序中使用的变量，尽量使用局部变量，因为这样可使子程序具有相对独立性，因此，对其可单独进行测试。

练习

1. 仔细阅读下列程序：

```
WITH student_io; USE student_io;
PROCEDURE factors IS
    large, small:integer;
    FUNCTION hcf(low, high:integer) RETURN integer IS
        remainder:integer;
        bigger:integer := high;
        smaller:integer := low;
    BEGIN
        --find the highest common factor
        (求最大公因数)
        LOOP
            remainder := bigger REM smaller;
            EXIT WHEN remainder=0;
            bigger := smaller;
            Smaller := remainder;
        END LOOP;
        RETURN smaller;
    END hcf;
    BEGIN
        put_line("input two positive integers, the smaller first");
        get(small); get(large);
        IF small<large AND small>0 THEN
            put("Their highest common factor is");
            put(hcf(small, large), width=>1); new_line;
        ELSE
            put_line ("The data has the wrong format");
        END IF;
    END factors;
```

请指出该程序中的形参、实参、过程调用、函数调用以及局部说明，利用参数调用的命名记号法改写函数调用。设输入数据为 12、35，试分析程序的运行过程。

2. 设有如下的函数说明：

```
FUNCTION increase(number:integer;by:integer := 1) RETURN integer IS
BEGIN
    RETURN number+by;
END increase;
```

请指出下列各函数调用的结果：

```
val := increase(3);
val := increase(3,1);
val := increase(3,2)
val := increase(3,by=>4);
val := increase(by=>4, numbe=>3);
```

3. 试编制一个过程，包含两个参数，一个是某月中的天数，另一个是该月第一天的星期，要求打印如下所示的日历。

SU	M	TU	W	Th	F	Sa
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

然后再利用所编的过程编一个程序打印年历，设一年的天数和一月份第一天的星期是已知的。

4. 设函数 hcf 说明为：

```
FUNCTION hcf(low, high: integer) RETURN integer IS
    remainder: integer := high REM low;
BEGIN
    IF remainder = 0 THEN
        RETUEN low;
    ELSE
        RETURN hcf(remainder, low);
    END IF;
END hcf;
```

试指出语句：

```
put(hcf(21,35));
```

的结果，并与练习 1 相比较。

第八章　子类型与离散类型的属性

本章是第三、四章的继续，主要介绍子类型、限制和离散类型的属性。

§8-1　子类型说明

通常，为了保证程序能达到预定的目的，必须采用一定的手段将程序中变量的值限制在一定的范围内。例如，变量 today 为类型 day，类型 day 的说明为：

TYPE day IS (sun, mon, tues, wed, thurs, fri, sat);

但是，我们不希望 today 取类型 day 的所有值，只取 mon 到 fri 之间的值，这可以通过子类型说明或限制来实现。例如：

today : day RANGE mon .. fri;

通过子类型说明可以给子类型取一名字，如：

SUBTYPE weekday IS day RANGE mon .. fri;

然后再说明变量 today 的类型：

today :weekday;

所以，通过限制范围可以从已有的类型中派生出一个子类型，子类型说明的语法图为如图 8.1 所示。

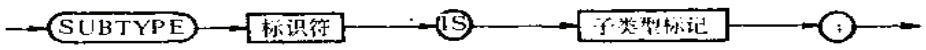


图 8.1

其中，子类型标记为图 8.2 所示的形式。

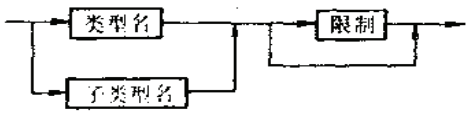


图 8.2

上述例子中所使用的限制称为“范围限制”，使用了保留字 RANGE。

在程序中限制变量的取值范围会有两方面的好处。首先可使程序的读者清楚地了解编程者的意图，因而增加了程序的可读性和可理解性。其次由于某一逻辑错误，试图将范围之外的值赋给变量 today，则系统将引发异常——constraint _ error(关于异常概念将在以后介绍)，因而有助于诊断程序中的错误。

任何原始类型或称为基类型(base type)可以进行的操作都适用于其子类型，而且只要能使用基类型对象的表达式中同样可以使用其子类型对象。另外，只要类型名适用的地方，都可以使用子类型名，我们也可以有子类型的子类型，例如：

SUBTYPE early _ weekday IS weekday RANGE mon .. wed;其中 weekday 的说明同前。

使用类型名可以测试一个变量是否落在某个范围中，所以，如果变量 tomorrow 的类型为 day，

则如下三个布尔表达式是等价的：

```
tomorrow IN mon .. fri;
tomorrow IN weekday'first .. weekday'last;
tomorrow IN weekday;
```

在 Ada 中也有内部定义的子类型，比如对于 integer 类型而言，它有两个内部子类型 natural 和 positive，可分别说明为：

```
SUBTYPE natural IS 0 .. integer'last;
SUBTYPE positive IS 1 .. integer'last;
```

在前几章的一些程序中，可以用子类型 natural 或 positive 变量替换类型 integer 变量来改写这些程序，建议读者仔细检查这些程序以确定如何修改。

作为使用子类型的例子，考虑下面的函数 mix，该函数接受两个基本颜色，返回混合的颜色，假设已有说明：

```
TYPE color IS (red, yellow, blue, purple, green, orang);
SUBTYPE primary IS color RANGE red .. blue;
```

函数 mix 的说明为：

```
FUNCTION mix(prim_1, prim_2: IN primary) RETURN color IS
BEGIN
    --mix two primary colors
    (将两种基本颜色混合)
    IF prim_1 = prim_2 THEN
        RETURN prim_1;
    END IF;
    CASE prim_1 IS
        WHEN red =>
            IF prim_2=blue THEN
                RETURN purple;
            ELSE
                RETURN orang;
            END IF;
        WHEN blue=>
            IF prim_2=red THEN
                RETURN purple;
            ELSE
                RETURN green;
            END IF;
        WHEN yellow=>
            IF prim_2=red THEN
                RETURN orang;
            ELSE
                RETURN green;
            END IF;
    END CASE;
END mix;
```

在函数 mix 中，由于所有可能组合的结果都有一个 RETURN 语句来终止函数调用，因此在函数的末尾就不需再设置 RETURN 语句了。所有可能的组合都由 CASE 语句指明。在本例中，参数

prim _ 1 属于子类型 primary，取值只能是 red、yellow、blue 三种情形。

需要注意的是，使用子类型要特别慎重。例如，有说明：

```
SUBTYPE small _ int IS integer RANGE 1 .. 10;
count ,small _ int : =1;
sum, natural : =0;
```

如果用下列语句序列求 1 到 10 的和。

```
WHILE count <=10 LOOP
    sum : =sum+count;
    count : =count+1;
END LOOP;
```

则第十次循环时，执行语句：

```
count : =count+1;
```

的结果是试图将值 11 赋给变量 count，但是值 11 已超出了范围，因此这个语句序列是有毛病的。

§8-2 离散类型的属性

在第四章中介绍的属性 first、last、succ、pred 同样适用于子类型。事实上，在上一节中已经使用了 weekday 的属性 first、last。其实这些属性都是一些预定义单参函数。

设有说明：

```
working : weekday : =fri;
holiday :day;
```

则赋值语句：

```
holiday : =weekday'succ(working);
```

是合法的，它将值 sat 赋给变量 holiday，而执行如下赋值语句将引发 constraint _ error 异常：

```
working : =weekday'succ(working);
```

此外，试图计算 day'succ(sat)或 day'pred(sun)之值时，也要引发 constraint _ error 异常。

现在再介绍其它几个属性，属性 pos 指出一个枚举值在其被说明时的位置，位置的值总是从零开始。例如：

```
day'pos(sun)=0
day'pos(mon)=1
```

属性 val 是属性 pos 的逆向运算，例如：

```
day'val(3)=wed
day'val(4)=thurs
```

这是因为，类型 day 的第三、第四位置的值分别为 wed 和 thurs。

如果试图计算 day'val(7)，则将引发 constraint _ error 异常。

属性 pos、val 同样适用于子类型，其结果与基类型相同。如 weekday'val(6)是合法的，其结果与 day'val(6)相同，其值都是 sat。

我们不提倡广泛使用属性 pos、val，但它们在某些情形下确实很有用。考虑如下语句，其中 ch、num 已分别说明为 character 和 integer 型变量。

```
IF ch IN '0' .. '9' THEN
    num : =character'pos (ch)-character'pos('0');
```

END IF ;

一个字符的 pos 属性之值就是它在 ASCII 中的位置。由于在 ASCII 表中，数字字符是连续的，所以如果 ch 是数字字符'0'… '9'中的某一个，则执行上述语句就把相同的整数值赋给 num。例如，若 ch 为'8'，则 num 之值为 8。

另一属性是 image。例如，day'image(sun)之值为串"SUN"。因此，image 属性把一个枚举值转换成一个大写字符序列。利用 image 属性可将枚举值写成一个串。例如 integer'image(34)之值为串"34"。属性 image 将整数转换常一个串时，串中有一个前导空格或减号。

相反，属性 value 将串转换成指定类型的值，它是属性 image 的逆向运算。例如，day'value("SUN")之值为 sun，integer'value("34")之值为整数 34。

与 image 相关联的是属性 width，它给出一个类型或子类型的最大象的宽度值。例如，枚举值 thurs 是类型 day 中最长的串，且 thurs 由 5 个字符组成，所以 day'width 之值为 5。

例 8-1 在第七章中，曾讨论了如何编制一个程序将一个整数转换成相应的英文形式，现在用属性 image 来编制程序。为方便起见，本程序中的整数限制在－99 到 99 之间。

程序 8-1

```
WITH student_io; USE student_io;
PROCEDURE write_number IS
    TYPE zero_to_19 IS (zero, one, two, three, four, five, six,
                        seven, eight, nine,
                        ten, eleven, twelve, thirteen, fourteen, fifteen, sixteen,
                        seventeen, eighteen, nineteen);
    SUBTYPE zero_to_nine IS zero_to_19 RANGE zero .. nine;
    low_val :zero_to_19;
    digit :zero_to_nine;
    TYPE tens IS (place-0, place-1, twenty, thirty, forty, fifty, sixty,
                  seventy,eighty, ninety);
    tens_part: tens;
    number :integer;
BEGIN
    --wirte number in the range -99 to 99
    (输出数－99 至 99)
    put("please input number "); get (number );
    IF number<0 THEN
        --deal with negative number
        (处理负数)
        put("minus");
        number := -number;
    END IF;
    IF number > 99 THEN
        put(number too large");
    ELSIF number <= 19 THEN
        --  convert number to enumeration value and write its image
        (进行转换)
        low_val := zero_to_19'val(number);
        put(zero_to_19'image (low_val));
    ELSE
```

```
        --deal with numbers from 20 to 99
        (处理数 20 至 99)
        tens_part := tens'val(number/10);
        Put (tens'image(tens_part));
        --find digit part and write its image
        (处理个位数字部分)
        digit := zero_to_nine'val(number REM 10);
        IF digit /= zero THEN
            put(zero_to_nine'image (digit));
        END IF;
    END IF;
    new_line;
END write_number;
```

看一下该程序的运行情况。假定读入数据 37，并已赋给变量 number。接着首先检查输入的数，即 number 之值是否小于零，是否大于 99 或是否小于等于 19，所有的测试都不成立，于是进入 ELSE 部分，执行语句：

```
tens_part := tens'val(number/10);
```

37 除以 10 得 3，则 tens'val 之值为枚举值 thirty，thirty 被赋给变量 ten_part 。接着执行语句：

```
put(tens'image(tens_part));
```

其结果打印出串：

```
THIRTY
```

程序中的枚举值 place－1、place－2 为虚拟值，用以保证 twenty、thirty 等的正确位置。继续执行程序，于是将枚举值 seven 赋给变量 digit，由于 digit 非零，因而输出 SEVEN。最后的结果为：

```
THIRTYSEVEN
```

至此，我们已介绍了离散类型的属性：first、last、pred、succ、pos、val、image、value、width。其中，pos 与 val、image 与 value 都是互为逆向运算。

§8-3 浮点子类型

前面已经提到，浮点值在计算机中的存储是近似的。Ada 允许在程序中指明浮点值的相对精度，这样可以不必依赖于所使用的计算机硬件所规定的浮点数精度。由程序指明浮点数所需精度，这在数值分析、工程计算等领域里是相当重要的，并且这样编制后的程序与机器无关。有关这方面的内容已超出本书的范围，我们仅用一些例子加以说明。

对于子类型说明：

```
SUBTYPE short IS real DIGITS 6;
```

其作用是：任何 short 类型变量都可存储至少 6 位有效数字。范围限制也可以写成：

```
SUBTYPE probability IS real RANGE 0.0 .. 1.0;
```

此外，一个说明中可有两种限制，但是精度限制必须放在最前面，例如：

```
SUBTYPE short_probability IS real DIGITS 4 RANGE 0.0 .. 1.0;
```

对于浮点子类型及浮点类型而言，也有各种浮点属性，例如，属性 small 给出可精确表示的最小非零正数，而属性 large 给出可精确表示的最大数。此外，属性 digits 给出了十进制数的精度，因此，float'digits 的值与 Ada 的实现有关。

练习

1. 考虑如下说明：

```
TYPE card IS (two, three, four, five, six, seven, eight, nine, ten, jack, queen, king, ace);
SUBTYPE face_card IS card RANGE jcak .. king;
value :card;
high_value: face_card;
```

请指出下面各语句中，变量 value 与 high_value 的值，哪些语句将会引发 constriant_error 异常？

```
value := card'succ (three);
value := face_card'pre(five);
high_value := face_card'succ(face_card'first);
high_Value := card'pred(card'last);
value := face_card'first;
value := face_card'val(0);
```

2. 设有说明：

```
SUBTYPE lower_case IS character RANGE 'a' .. 'z';
```

试指出下列语句的执行结果：

```
FOR ch IN 'a' .. 'z' LOOP
    put(ch);
END LOOP;
new_line;
FOR ch IN lower_case'first .. lower_case'last LOOP
    put(ch);
END LOOP;
new_line;
FOR ch IN REVERSE lowre_case LOOP
    put(ch);
END LOOP;
```

4. 参考 ASCII 表，试说明一个称为 printable_character 的子类型，其值为所有可打印的 ASCII 字符。

第九章　数　　组

在很多实际问题中，都需要把大量的数据用一种系统的方法进行处理，这时就需要用到数组。这一章讲述数组的概念、类型以及它的属性。数组与第十章的无约束数组和第十一章的记录构成 Ada 中的复合类型。

§9-1　数组类型

数组是由一些具有相同类型或子类型的分量组成的。有一维数组、二维数组和更高维的数组。

9.1.1　数组类型的定义

如果有一百个人，考虑他们的年龄组成的数组，可以如下定义数组类型：

```
TYPE ages IS ARRAY (1 .. 100 ) OF natural;
```

这里 ages 是类型名，1 .. 100 是数组下标范围，natural 是数组分量的子类型。

有了这个数组类型的定义，就可以对这 100 个人的年龄组成的数组 how _ old 说明如下：

```
how _ old: ages;
```

这表明 how _ old 是一个数组，类型为 ages，有 100 个分量，且每个分量的子类型为 natural。分量可表示为 how _ old(1)、how _ old(2)、how _ old(3) …how _ old(100)。这里的下标值最多只能是 100，如果超过这个数，则将引发 constraint _ error 异常。

对于数组的分量可以如下赋值：

```
how _ old(1) : =10;
how _ old(2) : =how _ old(1)+7;
```

如果有另外一个数组 A，类型也为 ages，则可把上面 how _ old 的各个分量的值赋给数组 A。语句为：

```
A : =how _ old;
```

这相当于：

```
A(1) : =how _ old(1); A(2)=how _ old(2); … A(100) : =how _ old(100);
```

注意：两个数组必须具有相同的类型。

9.1.2　匿名数组类型

Ada 语言中，一般的数组都有一个类型名。然而在有些数组中，可以不加类型名。比如说明：

```
b: ARRAY(integer RANGE 1 .. 6) OF real;
```

表示 b 是一个有 6 个分量的数组，各分量的类型为 real。由于下标范围 1..6 隐含了类型 integer，所以可更简单地写为：

```
b: ARRAY (1 .. 6) OF real;
```

要注意的是，虽然数组 b 没有类型名，但不能认为它就真的没有。实际上，它有一个隐含的、不可见的类型名。每个这种数组的定义都规定了一个新的类型，因此数组

C:ARRAY(1 .. 6) OF real;

D:ARRAY(1 .. 6) OF real;

应该具有不同的类型。所以不能写 D := C。即使有：

C, D:ARRAY (1 .. 6) OF real;

也不能写 D := C，因为这种说明和前面的说明是等价的。实际上它还是引入了两种类型。

那么到底什么时候才需引入一个类型名呢？这没有一个明确的规则。如果我们把一组对象看做一个完整的对象，或者需引入几个不同的数组来处理这一对象时，就需要引进一个新的类型名（否则这些数组就各自有一个特定的类型）；如果只把这一组对象看作一个混合体，并且做为整体与其他数组没有什么联系，那么它就可以是匿名类型。

9.1.3 静态数组与动态数组

如果数组的下标范围是确定的常量，这种数组就是静态数组。前面的例子都是属于这种类型的数组。

但这种下标范围不一定都为静态的，例如：

e:ARRAY (integer RANGE (1 .. n) OF boolean;

就是一个动态数组的。当 n 确定时，这个数组的分量个数才能确定。如果把它放在一个循环里面，而每次循环产生的 n 值不同，则每次确立的数组 e 也就有不同的分量个数。

灵活应用这种动态型数组对提高编程技巧有很重要的意义。

§9-2 一维数组运算

例 9-1 假设班中有 100 个学员，并已知他们的年龄。试求平均年龄以及他们中超过这个平均年龄的人数。

解决这个问题的思路很简单。首先通过一个循环依次读入这些年龄，读一个、加一个，等循环结束，100 个年龄之和也就算出来了，然后求平均年龄。接下来就把那 100 个年龄逐个与平均年龄对比，这里也要用到一个循环。在这个循环里设计一个计数器，每找到一个比平均年龄大的年龄，计数器就加 1。一旦循环结束，就可知道共有多少人超过这个年龄。下面就是解这个问题的程序，平均年龄取得是整数。

程序 9-1：

```
WITH student_io; USE student_io;
PROCEDURE older IS
    sample_size : CONSTANT integer := 100;
    TYPE ages IS ARRAY(1 .. sample_size) OF natural;
    how_old :ages;
    average:natrural;
    Sum_ages, number_older: natural := 0;
BEGIN
    --read ages and calculate their average
      (读入年龄计算平均值)
    FOR person IN 1 .. sample_size LOOP
        get (how_old(person));
        sum_ages := sum_ages+how_old (person);
```

```
    END LOOP;
    average := sum_ages/sample_size;
    --Count the number older than average
      (求大于平均年龄的人数)
    FOR person IN 1 .. asample_size LOOP;
        IF how_old (person)> average THEN
            number_older := number_older+1;
        END IF;
    END LOOP;
    put("Average age is "); put(average, width=>1);
    new_line;
    Put("Number older than avergae is ");
    Put(number_older, width=>1);
    new_line;
END older;
```

程序中用了一个常量标识符 sample_size 来代替整数文字 100，引进的数组也就变为动态的。这样做的好处是：如果想处理另外的一组年龄，比如 1000 个人的年龄时，只要将语句：

```
Sample_size: CONSTANT integer := 100
```

中的 100 改成 1000 就可以了。

在数组类型的说明中，对分量来说没有类型或子类型的限制。分量可以是实值、整数值或其它枚举类型，但同一数组中的分量应该具有相同的类型或子类型。数组的下标可以是任何离散类型或子类型。下面的例子重点让大家更多的了解关于数组的说明。

例 9-2 用数组描述一周中电影院每天卖出的票数。

首先可以这样来说明数组类型：

```
TYPE day IS (sun, mon, tues, wed, thurs, fri, sat);
TYPE daily_number IS ARRAY (sun .. sat) OF natural;
```

然后说明一个变量：

```
ticket_number:daily_number;
```

数组变量 ticket_number 有七个分量，每个分量的子类型为 natural。数组下标是类型为 day 的 sun、mon、tues、wed、thurs、fri、sat。

输入一周中售出的票数可以用下面的程序段：

```
FOR today IN sun .. sat LOOP
    get (ticket_number (today));
END LOOP;
```

类型标记用来表示数组说明中的下标范围。因此可以把上面的数组类型说明改写成：

```
TYPE daily_number IS ARRAY (day) OF natural;
```

或是在此基础上指明下标范围：

```
TYPE daily_number IS ARRAY( day RANGE sun .. sat) OF natural;
```

如果说明改为：

```
ticket_number:ARRAY (sun .. sat ) OF natural;
```

则数组 ticket_number 就变为没有类型名的数组，即匿名数组。

除了+、-、*、/等运算可以用于一维数组外，其他的一些运算也可用于一维数数组，如布尔运

算符 AND、OR、XOR 和 NOT。但由二目运算符 AND、OR、XOR 连结的两个数组必须具有相同个数的分量和相同数组类型。

例 9-3 若有类型说明：

```
TYPE bool IS ARRAY ( positive RANGE 1 .. 4) OF boolean,
  a, b: bool;
  c,d:ARRAY (1 .. 4) OF boolean;
  T: CONSTANT boolean : =true;
  F: CONSTANT boolean : =false;
```

然后给 a,b,赋初值：

```
  a : =(T, T, F,F);
  b : =(T, F, T, F);
```

则这时 a AND b 的值为(T, F, F, F)；

NOT b 的值为(F, T, F, T)；

a OR b 的值为(T, T, T, F)；

a XOR b 的值为(F, T, T, F)。

从这个例子可以看出，布尔运算用于数组的结果就是对数组分量进行布尔运算。但要注意 c and d 这样的写法是不允许的，因为 c 和 d 属于不同类型的数组。

§9-3 二维数组

数组可以是多维的，这时每一维都有一个独立的下标范围。在多维数组中最常用的是二维数组，如线性代数中的矩阵、文字处理中的表格等都要用二维数组表示。下面是几个多维数组说明的例子。

例 9-4

```
aa: ARRAY (integer RANGE 0 .. 2, integer RANGE 0 .. 3) OF real;
```

这个数组有 12 个元素，每个元素的类型为 real。表示一个元素要用两个下标，这 12 个元素可以表示成

aa(0,0)、aa(0,1)、aa(0,2)、aa(0,3)

aa(1,0)、aa(1,2)、aa(1,3)、aa(1,1)

aa(2,0)、aa(2,1)、aa(2,2)、aa(2,3)

多维数组的各个下标可以具有不同的类型或子类型，但必须是离散类型。请看下面的例子。

例 9-5 假设学生一周中每天有 8 学时用来学习。听课用 boolean 型值 T 表示，上自习用 boolean 型值 F 表示，应用前面说明的类型 day，则描述学生一周中每天每个学时的学习安排可用下面的数组说明：

```
study :ARRAY (day, 1 .. 8) OF boolean;
```

这个数组的第一个下标具有类型 day，范围为 sun .. sat，第二个下标是 integer 型的。

如果一周中只安排星期一到星期五学习，则可如下说明数组：

```
study:ARRAY (day RANGE mon .. fri, 1 .. 8) OF boolean;
```

下面的循环可以把 boolean 值读入这个数组：

```
FOR days IN mon .. fri LOOP
```

```
        FOR hours IN 1 .. 8 LOOP
            get (study (days, hours));
        END LOOP;
    END LOOP;
```

例 9-6 考虑一个电影院四周中每天售出的票数，找出平均人数最多的一天是星期几，打印这个日期以及四周中这一天分别售出的票数。

解决这个问题的思路是：

1. 读入并存储 28 天的售票数；
2. 计算并存储四周中同一个日期售出的票数之和；
3. 找出平均人数最多的一天是星期几；
4. 打印这个日期；
5. 打印四周中这个日期分别售出的票数。

根据这个思路编制程序需要两个数组。第一个数组存放 28 天的票数，我们把它设计成二维数组。第二个数组用来存放七个和数——即四周中每个日期售出的票数的小计。通过第二个数组的七个分量，可以找出一周中星期几看电影的人最多。

通过上面的分析，很容易把程序设计出来。

程序 9-6

```
WITH student_io; USE student_io;
PROCEDURE popular IS
    SUBTYPE week_range IS positive RANGE 1 .. 4;
    TYPE day IS (sunday, monday, tuesday, wednesday, thursday, friday, saturday);
    TYPE month IS ARRAY (week_yrange, day) OF natural;
    TYPE sub_total IS ARRAY (day) OF natural;
    tickets : month;
    day_total ,sub_total;
    best_yday: day := sunday;
    best_total :natural := 0;
BEGIN
    -- --Read the number of tickets sold in a 28_day period
      (读入 28 天售出的电影票数)
    -- --and calculate which day in the week is,
      (并且计算星期几)
    -- --on average, the most popular
      (平均看来是人数最多的日期)
    FOR days IN day LOOP;
        day_total (days) := 0;
    END LOOP;
    -- --Read data and calculate the sub_ytotals
      (读入 28 天的数据并计算出小计)
    FOR week IN week_range LOOP
        FOR days IN day LOOP
            get (tickets (week, days));
            day_total(days) := day_total(days)+tickets
                (week, days);
```

```
        END LOOP;
    END LOOP;
    --Find the most popular day in the week
      (找出人数最多的一天是星期几)
    FOR days IN day LOOP
            IF day_total (days )> best_total THEN
            best_day := days;
            best_total := day_total (days);
            END IF;
    END LOOP;
    put("on average the most popular day is ");
    put_line (day'image (best_yday));
    --write values making up the most popular day
      (打印人数最多的那个日期的 4 个值)
    put_line("The four components are");
    FOR week IN week_range LOOP
        put(tickets (week, best_day));
    END LOOP;
    new_line;
END popular;
```

程序中的 tickets 表示存储 28 个值的二维数组。day_total 表示存放 7 个小计值的一维数组。在第一个循环中，给数组 day_total 的七个分量全部赋初值 0，然后进入两个嵌套的循环。内层的循环每循环一次就读入一个值到二维数组里面，然后把这个值加到相应的那一天的小计里去。内层循环每进行七次，外层只进行一次。等外层循环进行 4 次过后，就读入了 28 天分别售出的票数并计算了 7 个小计。接下来设计一个循环找 7 个小计中的最大值和相应于这个最大值的日期。最后一个循环打印与这个日期对应的 4 天的售出票数。

程序输出的结果可能是这样的：

```
on average the most popular day is SATURDAY
The four components are
3460 3282 3101 3009
```

§9-4. 数组类型定义和数组属性

前几节的例子使我们对数组有了初步的了解。现在看一下 Ada 语言对数组类型的确切定义和数组类型说明的语法结构，最后介绍数组属性。

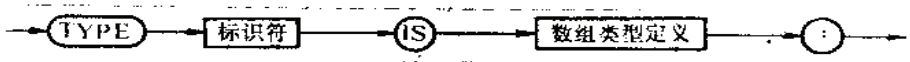


图 9.1

9.4.1 数组类型说明的语法图和数组类型定义方式

图 9.1 是数组类型说明的语法图，其中大写的词是 Ada 的保留字，在数组类型说明中不能改变。

数组类型定义的方式如图 9.2 所示。

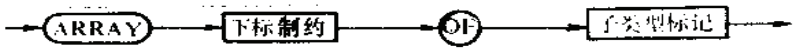


图 9.2

在这里下标制约定义为如图 9.3 所示。

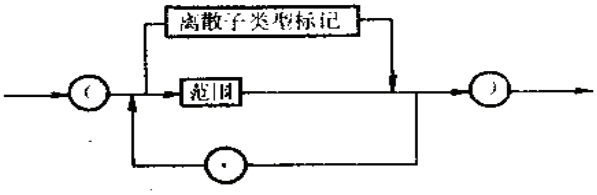


图 9.3

如果有子类型说明：

```
SUBTYPE letter IS character RANGE 'a' .. 'z';
```

则可按下面 4 种方式定义数组：

```
ARRAY character RANGE 'a' .. 'z') OF integer;
ARRAY ( 'a' .. 'z') OF integer;
ARRAY (letter) OF integer;
ARRAY (letter RANGE 'a' .. 'z') OF integer;
```

由此可以看出，数组类型定义中的下标界限可以是离散范围的所有一般形式。

9.4.2 数组属性

就象标量类型有属性一样，Ada 也给数组定义了属性。如果 matrix 是一个多维数组，则：

```
matrix'first(n)     表示第 n 个下标的下界；
matrix'last(n)      表示第 n 个下标的上界；
matrix'length(n)    表示第 n 个下标范围中所包含的值的个数；
mattrix'range(n)    表示第 n 个下标的取值范围。
```

例 9-7 如果有一个数组类型说明为：

```
TYPE table IS ARRAY (1 .. 3, 0 .. 7) OF real;
results : table;
```

则有下面的结果：

```
(1) results'first(1)=1          table'first(1) =1
(2) results'last(1)=3           table'last(1)=3
(3) results'length(1)=3         table'length(1)=3
(4) results'range(1)=1 .. 3     table'range(1)=1 .. 3
```

对于第二个下标有：

```
(5) results'first(2)=0          table'first(2)=0
(6) results'last(2)=7           table'last(2)=7
(7) results'length(2)=8         tble'length(2)=8
(8) results'range(2)=0 .. 7     table'range(2)=0 .. 7
```

如果不标明下标，则指的是第一个下标。例如：

```
results'first, results'last;
```

和

```
results'first(1), results'last(1);
```

是一样的。

range 属性对循环特别有用。例如：

```
FOR i IN results'range LOOP
    FOR j IN results'range(2) LOOP
        results(i,j) := 0.0;
    END LOOP;
END LOOP;
```

属性 range 也可用在说明里，例如：

```
n:integer RANGE results'range(2);
```

和

```
n:integer RANGE 0 .. 7;
```

是一样的。

§9-5 数组作为参数的使用方法

数组可以做为参数用在函数或过程中（匿名数组不可以作为参数）。在例 9-6 中说明了数组类型 sub_total，求出了看电影人数最多的一天是星期几。现在我们用函数的形式解这个问题。

例 9-8 假设已有数组类型说明：

```
TYPE sub_total IS ARRAY (day) OF natural;
```

则函数可写成下面的形式：

```
FUNCTION best (totals: IN sub_total)RETURN day IS
    popular_day: day := sunday;
    largest_total :natural := 0;
BEGIN
    --find the position of the largest value in the array
      (找出数组中的最大值及之对应的日期)
    FOR which_day IN totals'range LOOP
        IF totals (which_day)> largest_total THEN
            popular_day := which_day;
            largest_total := totals(which_day);
        END IF;
    END LOOP;
    RETURN popular_day;
END best;
```

若有说明：

```
day_total:sub_total;
```

且数组 day_total 的值即七个小计已给定，则调用：

```
best_day := best(day_total);
```

就可求出看电影人数最多的一天。

这里用到的数组参数应该取为 IN 方式，因为我们并没有改变它的值，而只是用到了这些值。函数调用中实参数组类型一定要和函数说明中的形参类型一致。在本例中都取为 sub _ total 类型。

从下面的例子中读者可以了解数组怎样用于过程。

例 9-9 把 15 个人的年龄按增序排列起来。

因为是递增排列，我们可以用这种方法：

(1)固定 15 个年龄形成的一维数组中的第一个分量，用它和第二至第十五个分量依次比较，每比较一次，如发现第一个分量大，则交换相对位置，把小的分量放在第一个分量的位置。这样，比较到第十五个分量之后，第一个分量的位置放的就是最小年龄。

(2)对于(1)形成的数组，固定它的第二个分量，然后分别和第三至第十五个分量比较大小。如发现第二个分量小就仍然放在第二个位置，否把它与比它小的那个分量交换位置。这样比较到第十五个分量之后，第二个分量的位置放的年龄就大于或等于第一个位置放的年龄，但不会比其他的那些年龄大。

(3)这个过程一直继续下去，直到第十四个分量固定下来和最后一个年龄比较大小，然后确定位置。到此就把 15 个年龄按照递升的顺序排列起来了。

这个比较过程可以用两个嵌套的循环来完成。外层循环要执行 14 次，每完成一次循环就排定一个年龄。内层循环比较剩下的年龄的大小，并把小的年龄放在已排定的年龄的后面。

假设已有说明：

```
TYPE list IS ARRAY (1 .. 15) OF integer;
```

则过程可以这样写：

```
PROCEDURE selection _ sort (ages ;IN OUT list )IS
    small _ pos, natural RANGE list'range ;
    smallest ;integer;
BEGIN
    --sort the components of the ages list
      (把年龄数组的分量排序)
    FOR low IN list'first .. last-1 LOOP
        --Components in positions list'first to low _ 1 are
        --now ihn their final pos itions
        --find the smallest remaining component
        (现在第一到第 low _ 1 个分量已按升序排好了位置，接下来找剩余的分量的最小值)
        small _ pos := low; smallest := ages (low);
        FOR pos IN low+1 .. list'last LOOP;
            IF smallest>ages (pos) THEN
                small _ pos := pos; smallest := ages (pos);
            END IF;
        END LOOP;
            --swap samallest with the ages in position low
              (交换最小年龄与第 low 个分量所在的位置)
            ages(small _ pos ) := ages (low)
            ages(low) := smallest
            --Components in position list' first to low are
            --now in their final positions
              (已经排好了 low 个年龄 的顺序)
    END LOOP;
```

```
    --Components in positions list'first to list'last-1
    --are now in their final positions
      (前14个年龄已经排好了顺序)
    --the last component must therefore be in its
    --final position, all components are in their
    --final sorted positions
      (最后一个年龄的位置也就找到了,因此所有的年龄都已排好了序)
END selection_sort;
```

在这段程序中,数组 ages 不仅被使用,而且它的分量的值被改变了(除非所有的年龄都一样),因此这个数组应该规定成 IN OUT 方式。

注意,在过程中数组属性被用来控制循环的执行次数,这样,当类型 list 的下标界变化时,过程不需改变。

§9-6 数组聚集

使用数组时经常要给数组赋初值,这可通过数组聚集来实现。

9.6.1 一维数组的赋值

对一维数组的赋值,有以下几种聚集形式。

1. 位置记号法

例如:

```
TYPE list IS ARRAY(1 .. 7 ) OF natural;
different : list := ( 5,6,7,8,9,10,11)
```

括号中的第一个值就表示数组 different 的第一个分量,第二个值表示 different 的第二个分量……,因此有 diferent(1)=5,different(2)=6,…,different(7)=11。

2. 指名记号法

例如:

```
same:list := (1 .. 7=>1);
```

表示 same 的第一至第七个分量的值都为 1。这种表示法是指名下标或下标范围,然后写=>,最后给出要赋的值。对 same 的说明也可写成:

```
same : list := (list'range =>1);
```

当然,这是一种最简单的情况。如果用指名记号法来说明上面的 different,则有:

```
different : list := (1=>5, 2=>6, 3=>7, 4=>8, 5=>9, 6 =>10, 7=>11);
```

这显然是很繁琐的。因此对数组赋值时,应区别情况选择不同的方法。

如果一个 list 型数组第一个和第三个分量赋值为 0;第二个和第四个赋值为 1,其余三个分量赋值为 2,则可以选择指名记号法:

```
another : list := (1|3=>0, 2|4=>1, 5.. 7=>2);
```

竖杠"|"用来分隔赋相同值的分量的不同下标。

如果有说明:

```
TYPE day IS (sun, mon tues, wed, thur, fri, sat);
TYPE study IS ARRAY (day ) OF natural;
```

```
study_time : study;
```

study_time 是一个有七个分量的一维数组，表示学生一周中每天学习的小时数。如果每天学习 6 个小时，可用下面几种方式赋值：

```
study_time :study := (6,6,6,6,6,6,6)
study_time :study := (study'range => 6);
study_time :study := (sun .. sat => 6);
study_time :study := (day => 6);
```

9.6.2 多维数组的赋值

对于多维数组的赋值，聚集形式和一维聚集差不多。假设有类型说明：

```
TYPE table IS ARRAY (1 .. 2, 1 .. 5 ) OF real;
```

则类型为 table 的数组(results)的赋值形式可以写成：

```
results : table := (1 .. 2 => (1 .. 5 => 0.0 ));
```

或

```
results : table := ((1 .. 2 => 0.0), (1 .. 5 => 0.0));
```

如果 mat 是另外一个类型为 table 的数组，可以这样赋值：

```
mat: table := ((1 => 1.0, 2 => 0.0), (1 => 0.0, 2 .. 5 => 1.0));
```

或

```
mat :table := ((1.0, 0.0), (0.0, 1.0, 1.0, 1.0, 1,0));
```

这种聚集形式是把多维数组中的每一维写成一个一维聚集，把这些一维聚集用逗号隔开，则整个聚集形式就象是一个一维聚集。

最后需要指出，常量数组被说明时一定要有一个值。

§9-7 注意事项

1. 用一个数组的值给另一个数组赋值时，两个数组必须是同类型的，分量个数也应一样多。
2. 数组类型可以是匿名的。两个匿名数组类型应视为不同的类型，因此不能互相赋值，匿名数组不能作为子程序的参数。
3. 用聚集形式给数组赋值时，聚集必须是完整的，即所有数组分量都应对应聚集中一个确定的值。
4. 位置记号法和指名记号法不能在数组聚集中混和使用。
5. 只有一个分量的数组赋值聚集必须使用指名记号法。

练习

1. 选择输入一个数据并分析程序的执行情况。

```
WITH student_io ; USE student_io;
PROCEDURE search IS
    TYPE vals IS ARRAY (0 .. 8) OF integer;
    value_list : vals := (0,7,17,5,29,13,19,3,11);
    position : natural := 8;
    number : integer;
BEGIN
```

```
        get(number);
        value_list(0) := number;
        WHILE value_list(position) /= number LOOP;
            position := position - 1;
        END LOOP;
        IF position > 0 THEN
            put("position is "); put(position);
            new_line;
        ELSE
            put_line ("number not in list");
        END IF;
    END search;
```

2. 假设有如下数组说明：

```
        TYPE list IS ARRAY (1..50) OF positive;
        A:list;
```

写一个语句序列实现下列功能：

(1) 给数组 A 读入 50 个正整数；

(2) 找出数组中最小整数所在的位置。

3. 写一个函数实现功能：调用类型为 list 的数组并返回数组中最小分量所在的位置。

4. 写一个程序：读入十个数，然后按读入顺序逆向写出这些数。

5. (a) 说明一类型为 month 的枚举类型来表示一年的十二个月。

(b) 说明一数组常量表示十二个月中每月的天数。

(c) 写一个函数：给定一个日期，可得出这个日期是一年中的第几天。

6. (a) 假设有以下说明：

```
        SUBTYPE lower_case IS character RANGE 'a'..'z';
        SUBTYPE upper_case IS character RANGE 'A'..'Z';
```

说明一个有 26 个分量的数组，分量类型为 upper_case，下标取值为类型 lower_case。给这个数组赋值使得每个分量的值是分量下标对应的大写字母。

(b) 写一个过程：读入一行由大写或小写字母以及标点符号组成的文字，然后把小写字母换成大写，再将该行输出。

第十章　无约束数组

本章引入了无约束数组的概念，并对它的一个特殊情形——字符串，及其处理进行全面的介绍。

§10-1　无约束数组的引入

到目前为止，我们所引入的数组都是所谓有约数组定义法定义的。然而只用这种数组来解决实际问题是很不够的。例如，当要处理具有不同个数的分量但类型相同的许多数组时，有约束数组就不能满足要求。因为这种数组的下标有明确的上下界，即使对动态数组而言，下标也是有明确的上下界的。由于这种原因，就不能同时说明多个具有不同长度而类型相同的数组。为解决这一问题，Ada 设计了一种专门的下标无明确上下界的数组——无约束数组。

10.1.1　无约束数组的例子

考虑语句：

```
TYPE vector IS ARRAY (integer RANGE < > ) OF real;
```

它定义了一个一维数组，其类型名为 vector，分量类型为 real，下标类型为 integer，唯一和以前的数组类型定义不同的是下标范围用符号< >表示。不妨把它读作盒子，它表示下标没有给出上下限。当说明一个类型为 vector 的数组时，必须指明下标的界限。例如：

```
V: vector (1 .. 10);
W: vector ( 1 .. 5);
u: vector ( 1 .. 15);
```

也可以先引入一个子类型，再说明数组。例如：

```
SUBTYPE vector _ 20 IS vector (1 .. 20);
v1: vector _ 20;
v2: vector _ 20;
v3: vector _ 20;
```

说明一个无约束数组类型时，下标也可以由子类型名给出，下面的语句：

```
TYPE p IS ARRAY ( positive RANGE < >) OF real;
```

就定义了下标范围取正整数范围的一个数组类型。

在一些大型问题的计算中，都需要用到矩阵这样的二维数组。可以把它说明成无约束数组类型：

```
TYPE matrix IS ARRAY (positive RAGNE < >, positive RANGE < >) OF real;
```

应用这种类型可以说明不同阶数的矩阵。例如：

```
table : matrix (1 .. 6, 1 .. 4);
large _ table: matrix (1 .. 7, 1 .. 20);
```

在这种数组说明中，上下界可以是表达式。这时的数组为动态数组。例如：

m : matrix (1 .. n , 1.. n);

当 n 确定时，数组 m 的下标范围也就确定了。

有了上面的这些例子，我们对无约束数组就有一个大概的了解。接下来就谈一下它的定义和语法规则。

10.1.2 无约束数组定义

请看下面的语法图(图 10.1)。

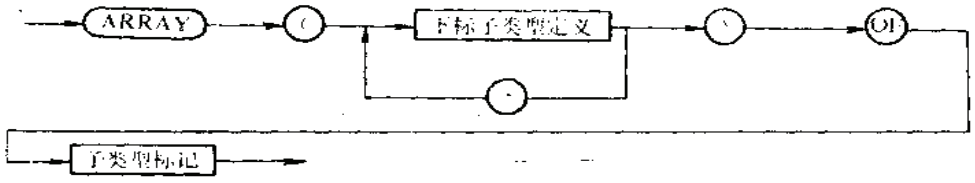


图 10.1

其中的下标子类型定义如图 10.2 所示。

图 10.2

从图中可以看出，无约束数组类型定义和有约束数组类型定义的区别仅在于下标子类型的定义。它指明了数组下标可能的取值范围，但没有明确上下限到底是什么，而是等到说明一个数组对象时给出。

一定要区分开数组类型说明和数组说明，后者指的就是数组对象的说明。

对于一维数组，需要一个下标子类型或类型定义，它指定数组下标可能的取值范围，用一个类型或子类型标记后面跟 RANGE < >组成。如果是多维数组，则需要多个下标类型或子类型定义。这时要么对每一维都指定下标上下界，要么都不指定。如果指定就变成有约束数组定义。

对于一般的数组对象指明下标取值范围的方法，就是在括号里写出每一维下标的上界和下界。而对于常量数组，可以用另外的方法指明它的下标的上下界。

如果有说明：

```
TYPE matrix IS ARRAY (postive RANGE < >, positive RANGE < >) OF real;
mat_2: CONSTANT matrix := ((1.0, 0.0), (0.0, 1.0));
```

就表示 mat_2 的第一个下标取值为 1 .. 2，第二个下标取值为 1 .. 2。各个分量为 mat_2(1,1) =1.0, mat_2(1,2) =0.0, mat_2(2,1)=0.0, mat_2(2,2)=1.0。

如果把上面的数组类型说明改为：

```
TYPE matrix IS ARRAY (integer RANGE < >, integer RANGE < >) OF real;
```

则各个分量的下标就不一样了。这要看 integer' first 到底是等于什么。总之，常量数组下标的下限由下标子类型的属性 first 决定。至于上限，那要看常量数组到底包含了多少个分量。

在下面的例子中，我们把无约束数组用于函数来说明它的用法。

考虑两个数组的相加。假设有说明：

```
TYPE vector IS ARRAY (natural RANGE < >) OF real;
```

则我们对两个类型为 vector 的一维数组的加法可以用下面的函数来实现。

```
FUNCTION add (vector_a, vector_b : vector )RETURN vector IS
    new_vector : vector (vector_a'range);
BEGIN
    --add components of two vectors
      (把两个向量的对应分量相加)
    FOR pos IN vector_a'range LOOP
        new_vector(pos) := vector_a(pos)+vector_b(pos);
    END LOOP;
    RETURN new_vector;
END add;
```

函数中两个数组参数和返回的结果数组都是无约束数组。参数的下标范围由函数调用时实际参数的下标范围确定；返回的结果数组的下标范围由返回语句中规定的数组的下标范围确定，在这个例子中由 vector_a 的下标范围确定。

因此，如果我们有说明：

```
first, second, sum : vector(1..5);
```

并给 first、second 两个数组赋以初值，则函数调用

```
sum := add(first, second);
```

将给出 first 和 second 对应分量的和，也即 sum 的分量。

注意，在这个函数调用中，几个数组必须有相同的下标范围，否则就会引发 constraint_error 异常。

§10-2 字符串

10.2.1 字符串定义

字符串就是由引号括起来的零个或多个字符序列。例如：

```
"This is a boy"
```

虽然 Ada 规定一般情况下字母的大小写无什么区别，但在字符串中应看成是有区别的。

字符串应看成是类型为 string 的数组。类型 string 的说明为：

```
TYPE string IS ARRAY(positive RANGE< >) OF character;
```

这是一个一维无约束数组类型，只不过数组的分量都是字符。因为是数组类型，所以它遵循数组的一切规则。例如说明：

```
s: string(1..7);
```

表示 s 由 7 个字符组成。

假如字符串 G 是一个常量"girl"，则可说明为：

```
G:CONSTANT string := "girl";
```

对字符串变量的赋值可以采用上面这种串的形式，也可以采用聚集的形式，例如

```
s:string(1..7) := ('s', 'h', '0', 'r', 't', 'e', 'r');
```

这和语句：

```
s:string(1..7) := "shorter";
```

是一样的。

如果对数组变量赋很长一串同样的字符，则聚集形式显得方便些。例如：

```
t:string(1..40):=(1..40=>'*')
```

表示 t 由 40 个 * 号组成。

对于常量数组，可以不指明下标界限，而由实际的字符串文字得出。比如前面的语句：

```
G :CONSTANT string := "girl";
```

实际上下标界限为 1..4。

10.2.2 字符串运算

关系运算符<、<=、>、>=可以用于字符串的比较运算。如果假设所有的串都是 string 型的，则下列表达式的结果都为 true。

```
"Jane"< "Margaret"
"Julie"> "Julia"
"John">"James"
```

但如果类型不为 string，则得出的结果就不见得和现在一样了。例如，当有说明：

```
TYPE letter IS ('J','o','h','n','a','m','e','s');
TYPE word IS ARRAY (positive RANGE < >) OF letter;
```

如果字符串"John"和 "James "都是属于类型 word 的串时（为了说明它们不是 string 类型的，需要用记号 word'("John ")、word'("James")），表达式 word'("John ") >word '("James")的结果就为 false。

用于字符串运算的还有运算符 &。例如：

```
"st"&"ing"="sting"
```

这个运算符也可以把一个字符串和一个单个字符连接起来，但要求这个单个字符的类型和字符串这个数组的分量类型相同。例如：

```
'J'&"ames"="James"
"Joh"&'n' ="John"
'J'&'J' ="JJ"
```

此外，字符串还有一种"片段"运算。片段是一维数组中一个连续的分量序列。如果：

```
s:string (1..7) := "shorter";
```

则

```
s(1..3)="sho",
```

而片段

```
s(5..7)="ter"。
```

一个数组变量的片段也是一个数组变量。因此可以给它们赋值。如果：

```
s(1..3)="bet";
```

则这时 s 的值为"betrter"。

下面举一个例子说明片段怎样用于过程。

例 10-1 把一个字符串拷贝到另一个字符串中，如果被拷贝的字符串太短，则在后面补充足够的空格 6；如果太长，就只拷贝前面的部分字符。此过程可用于任何长度的字符串。注意，在数组赋值时，数组变量的分量个数和所赋值的分量个数应该相同，否则将引发 *contraint _error* 异常。

```
PROCEDURE copy (old :IN string; new: OUT string) IS
```

```
BEGIN
    -- Copy old string to new string
      (把旧串拷贝到新串)
    IF new'length >=old'length THEN
        --old' length characters copied from old string to new
        ---and new string padded with trailing blanks
          (旧串的所有字符拷贝到新串,因旧串短所以要在新串中补足空格。)
        new := (new'range=>' ');
        new (new'first..new'first+old'length-1) := old;
    ELSE
        -- new'length characters Copied from old string to new
          (按照新串的长度把旧串中前面的字符拷贝过来。)
        new := old (old'first..old'first+new'length-1);
    END IF;
END copy;
```

下面再举一个数组分量是数组的例子,以使读者对数组的规则有更多的了解。

例 10 2 微机屏幕一次可显示 24 行,每行 80 个字符。由数组表示就可以这样来说明:

```
SUBTYPE line IS string (1..80);
screen: ARRAY (1..24) OF line;
```

这样,screen 的每个分量都是有 80 个字符的串。如果表示屏幕的第三行字符串就可用 screen(3),而要表示第三行第五个字符就可用 screen(3)(5)。

§ 10-3 过程 get _ line

在字符串的处理中,经常要用到两个过程:一个是已用过的 put _ line 过程,另一个就是过程 get _ line。它是由程序包 student _ io 定义的。

10.3.1 get _ line 的功能

请看语句:

```
get _ line(st, length);
```

其功能是读一串字符并把它存在 st 中。st 是字符串变量,而 length 具有子类型 natural。当读字符串时,如果遇到行结束或串结束就停止读入。读入的最后一个字符的下标位置就存放在 length 中。如果读的是空行,就表示没读任何字符。这时把 length 设置为 st'first _ 1。这个过程被用来读入一个行并存储起来以便于查看它的内容。

当要表示一个行时,只要写:

```
st(st'first..length)
```

就行了。

应用这种形式,就可以由下面的两条语句读入和输出一个行:

```
get _ line(st,length);
put _ line(st(st'first..length));
```

10.3.2 get _ line 的应用

例 10-3 下面是一个关于提问和回答的例子,读者可以从中了解 get _ line 的用法。

程序 10-3

```
WITH student_io; USE student_io;
PROCEDURE question IS
    SUBTYPE line IS string (1..80);
    name, answer: line;
    name_length, answer_length: natural := 0;
BEGIN
    put_line("Hello, what is your name?");
    get_line(name, name_length);
    put_line("Hello," &name (1..name_length));
    put_line("Would you like to answer a question?");
    LOOP
        get_line (answr, answer_length);
        IF answer(1..answer_length)="yes" THEN
            put_line("What is the capital of the United States?");
            get_line (answer, answer_length);
        IF answer(1..answer_length)="Washington"THEN
            put_line("Well done, "&name(1..name_length)
                    &", that is correct.");
            EXIT;
        ELSE
            put_line("Sorry,"& name(1..name_length)
                    &", that is wrong.");
            put_line("Would you like to try again?");
        END IF;
    ELSIF answer(1..answer_length)="no" THEN
        EXIT;
    ELSE
        put_line("Please type either yes or no.");
    END IF;
END LOOP;
    put_line("Bye," &name(1..name_length));
END question;
```

为了使大家容易理解程序，我们给出如下的一个可能的执行结果。

```
Hello, what is your name?
John
Hello, John
Would you like to answer question?
OK
Please type either yes or no.
yes
What is the capital of the United States?
New York
Sorry, John, that is wrong.
Would you like to try again?
yes
What is the capital of the United States?
```

```
Washington.
Well done, John, that is correct.
Bye, John
```

练习

1. 设有类型说明：

```
TYPE list IS ARRAY (integer RANGE < >) OF positive;
```

写一个过程：能够输出具有这种数组类型的任意数组的全部分量。过程中设制一个参数指定每行输出的分量个数。如果不指定，则认为每行输出一个分量。

2、设有说明：

```
name:string(1..15) := "Pascal Programs";
new_name:string (1..12);
```

经过下面的赋值后，new_name 的值是什么？

```
new_name(1..3) := "Ada";
new_name(4..12) := name(7..15);
```

3. 写一个过程，实现过程 get_line 的功能。

4. 分析下面程序的执行情况：

```
WITH student_io; USE student_io;
PROCEDURE join IS
    st:string(1..50);
    length : natural := 0;
    PROCEDURE append(stem: IN OUT string; suffix:
        IN string; stem_length: IN OUT natural) IS
        start : positive := stem_length+1;
    BEGIN
        IF stem_length+suffix'length<=stem'last THEN
        stem_length := stem_length +suffix'length;
        stem(start..stem_length):= suffix;
      END IF;
    END append;
BEGIN
    st(1..6) := "return";
    put_line(st(1..6)&"ing");
    length := 6;
    append(st,"ing", length);
    put_line(st(1..length));
END join;
```

第十一章 记　　录

记录是一种很重要的复合数据类型，它主要应用于管理方面。和前面介绍的数组不同，数组的各个分量都具有相同的类型，而记录中的分量可以是不同的类型。因此，记录使得具有不同类型的许多信息可以集合在一起而被看成一个整体，用来表示一个日期、表示一个人的档案等等。

§11-1 记录的定义

在介绍 Ada 对记录的语法定义之前，先看几个例子，以便对记录的形式和说明方法有一个初步的了解。

例 11-1 对一个日期类型的记录有如下说明：

```
TYPE which_month IS (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct, Nov, Dec);
TYPE date IS
    RECORD
        day : positive RANGE 1..31;
        month : which_month;
        year : positive RANGE 1000 .. 2000;
    END RECORD;
```

这个记录的类型是 date，包含 day、month、year 三个分量。

怎样说明一个记录类型的变量和常量呢？方法和原来一样。例如：

```
D: date;
birthday : date;
```

记录变量的分量记为：

```
birthday.day    birthday.month    birthday.year
```

定义一个记录类型的方式为可以表示为图 11.1 的形式。

图 11.1

而分量说明的形式如图 11.2 所示。

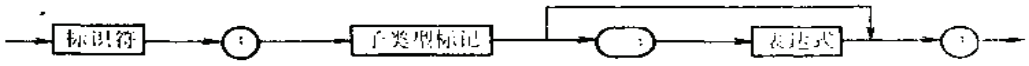


图 11.2

上面所说的是一种比较简单的记录定义方式。实际上还有更为复杂的记录类型，在这里就不介绍了。

需要注意的是，所定义的记录都必须有一个类型名，不能定义一个匿名记录类型，而这对数组是可以的。

§11-2 对记录的赋值和运算

对记录的赋值可以采用以下几种形式。

1. 分别给每个分量赋值，例如：

```
birthday.day := 14;
birthday.month := Mar;
birthday.year := 1963;
```

这种方法是先写记录的名称，而后加一圆点，再写上分量名。

2. 使用位置记号法赋值，例如：

```
birthday := (14, Mar, 1963);
```

使用这种方法时，聚集的分量必须按照记录类型说明中分量的顺序给出。

3. 使用指名记号法赋值

上面的例子可以写成下面几种形式：

```
birthday := (day => 14, month => Mar, year => 1963);
birthday := (month => Mar, year => 1963, day => 14);
birthday := (year => 1963, month => Mar, day => 14);
```

.
.
.

因此，应用这种形式给记录赋值时，可以不管分量原来的次序。

4. 混合法

指的是混和使用位置记号法和指名记号法。用这种方法时，位置记号法用于前面部分，指名记号法用于后面部分。要注意的是，使用位置记号法赋值的分量要求按分量说明中的顺序排定。当一个分量开始用指名记号法赋值时，后面的分量就都要用指名记号法赋值。

例如：

```
birthday := (14, Mar, year => 1963);
birthday := (14, year => 1963, month => Mar);
```

注意，这种方法不能用于数组的赋值。

另外，如果 D 是和 birthday 同类型的记录，则可以把 birthday 的值赋给 D。写作

```
D := birthday;
```

当说明一个常量记录时必须要赋一个值。例如：

```
special_day:CONSTANT date := (1, Jan, 1990);
```

在记录类型上定义的运算有=（等于号）和/=（不等于号），其他的运算都要在分量上进行。下面举一个例子说明记录怎样用在函数中参加运算。

例 11-2 写一个函数实现下面的功能，给定一个日期，求第二天的日期。

按照前面说明的记录类型 date，可以把这个函数写成下面的形式：

```
FUNCTION next_day(today: date ) RETURN date IS
    next : date := today;
BEGIN
    --find and return the date following today
```

```
    (计算并返回第二天的日期)
CASE today.month IS
    WHEN Apr|Jun|Sep|Nov=>
        IF today.day=30 THEN
            next.day:=1;
            next.month:=which_month'succ(today.month);
        ELSE
            next.day:=today.day+1;
        END IF;
    WHEN Dec=>
        IF today.day=31 THEN
            next:=(1,Jan,today.year+1);
        ELSE
            next.day:=today.day+1;
        END IF;
    WHEN Fed=>
        IF (today.year REM 4=0 AND today.day=29) OR
            (today.year REM 4/=0 AND today.day=28) THEN
            next.day:=1;
            next.month:=Mar;
        ELSE
            next.day:=today.day+1;
        END IF;
    WHEN Jan|Mar|May|Jul|Aug|Oct=>
            IF today.day=31 THEN
                next.day:=1;
                next.month:=which_month'succ(today.month);
            ELSE
                next.day:=today.day+1;
            END IF;
END CASE;
    RETURN next;
END next_day;
```

§11-3 分量为复合类型的记录

前面介绍的记录分量类型都是标量类型。其实记录的分量可以是任意类型。它们可以是其它的记录或数组，但不允许是无约束数组即数组的下标范围不能是<>，并且必须要有一个数组类型名。记录不能把自身做为分量。

例 11-3 描述一个人的姓名和生日的记录可以说明为：

```
TYPE Person IS
    RECORD
        birth :date;
        name :string(1..20):=(1..20=>' ');
    END RECORD;
```

这是一个分量为复合类型的记录的例子。分量 birth 是一个记录，分量 name 是一个数组，对这

个数组赋的值是 20 个空格。

如果小王的生日是 1970 年 6 月 5 日，就可以如下赋值：

```
Wang :Person
Wang.birth := (5, Jun, 1970);
Wang.name (1..4) := "Wang";
```

这时，Wang 的值为

```
((5, Jun, 1970), "Wang                    ")。
```

注意，这里的空格是必须的。

小王生日的年月日可分别表示成：

```
Wang.birth.year = 1970;
Wang.birth.month = Jun;
Wang.birth.day = 5;
```

小王姓名的第一个字母就是：

```
Wang.name(1) = "W";
```

例 11-4 假设一个小型图书馆有 500 册图书被借出。每本书的代码用 8 个字符表示，每个借阅者用 6 位数表示。对每本书标明借阅日期和是否过期。这些信息了可以放在一个以记录为分量的数组中。说明如下：

```
TYPE book IS
    RECORD
        book_number : string(1..8);
        reader_number : positive RANGE 100000..999999;
        time : date;
        overtime : Boolean := false;
    END RECORD;
loan: ARRAY (1..500)OF book;
```

记录分量可以给一个默认值，本例中 overtime 的默认初值是 false。如果借阅信息都已放在 loan 数组中，则下面这段程序可以把超过借阅期限的书号及其借阅者代码打印出来。

```
FOR i IN loan'range LOOP;
    IF loan(i). overtime THEN
        put(loan(i).book_number);
        put(loan(i).reader_number);
        new_line;
    END IF;
END LOOP;
```

下面，研究堆栈元素的存取。堆栈可以理解成上面开口的桶，往里放元素和往外取元素都要通过这个开口。这两种操作分别叫做进栈和出栈，最后进栈的元素最先出栈，最上面的元素叫栈顶元素。

例 11-5 用一个数组表示堆栈，对 100 个元素描述进栈和出栈过程（假设已说明了类型 item）。

首先说明一个数组类型：

```
TYPE item_list IS ARRAY (positive RANGE <> OF item;
```

然后，再说明一个堆栈记录类型：

```
TYPE item_stack IS
    RECORD
        element : item_list(1..100);
        top : natural := 0;
    END RECORD;
```

分量 element 用来存放元素，分量 top 指向最上面的元素（也即栈顶元素）的堆栈指针。这里设置了一个默认初始值 0 表示栈中无元素即空栈，因此空栈可以说明为：

```
empty_stack : item_stack;
```

往栈顶加一个元素和从栈顶取走一个元素（即进栈和出栈）的操作表示成下面的过程。

```
PROCEDURE push (x: item; stack : IN OUT item_stack) IS
BEGIN
    --add item to the stack
      (元素进栈)
    IF stack.top=100 THEN
        put_line("Stack is already full");
    ELSE
        stack.top := stack.top+1;
        stack.element(stack.top) := x;
    END IF;
END push;
PROCEDURE pop(x :OUT item; stack : IN OUT item_stack) IS
BEGIN
    --remove top item from the stack
      (元素出栈)
    IF stack.top=0 THEN
        put_line ("Stack is empty")
    ELSE
        x := stack element (stack.top);
        stack.top := stack.top-1;
    END IF;
END pop;
```

每次操作时要注意：不能向已满的栈中再加元素，也不能从一个空栈中向外再取元素。由于类型 item 并不是指的某一特殊类型，因此这个过程可以用于各种类型的元素的进栈和出栈操作，这只要改一下类型说明就可以了。

§11-4 注意事项

1. 记录类型不能是匿名的。
2. 记录中的分量不能是匿名数组，也不能是无约束数组。
3. 只有一个分量的记录赋值时聚集只能使用指名记号法。

练习

1. 定义一个记录类型 time_of_day，用它来表示一天中某个时间的时、分、秒数。
2. 定义一个记录类型 exams，用它来表示班中 50 名学生的姓名、学号以及数学、物理、计算机课的考试成绩。然后写

一个过程计算每个学生三门课的平均成绩。

3. 图书的信息包括书名、书号、出版单位、作者姓名等，写一个适当的类型说明表示这些信息，然后写一个能够用于图书检索的过程。即给定一个书号或作者，能够把相应的书名找出来。

第十二章　程序包

程序包是 Ada 语言中最重要的概念之一。引入程序包是为了隐藏一些细节，而这些细节是本来就没有必要让使用程序的人知道的。程序包包括两部分：一是程序包规格说明，它的内容通过程序的其它部分可以查看；另一部分是可选择的，叫做程序包体。它的内容是隐藏起来的。这一章主要介绍程序包规格，下一章介绍程序包体。

§12-1　程序包的引入

一般我们要解决一个大课题时，总是把它先划分成一系列基本上是互相独立的子课题，也可以细分下去，就有所谓一级子课题、二级子课题、三级子课题等等。把这些小课题解决之后的结果综合起来就是大课题的答案。编制程序时，我们也是采用这种类似的方法。先把一个大问题化为一系列基本上独立的小问题，对这些小问题编制一系列子程序，然后写一个主程序调用这些子程序（Ada 语言中的子程序是由一些函数和过程组成的）。

子程序写好之后，我们所关心的就是如何去调用它，至于子程序内部的内容可以不管，只要它能够实现所需要的操作就行了。因此我们希望子程序能够隐藏一些细节，使得子程序中的东西在外面是不可见的。但子程序是否有这个功能呢？看一下下面的例子就清楚了。

例 12-1　把班上 100 个人的学号放在一个数组中（假设学号用正整数表示）。

解决这个问题可以仿照向堆栈中存放元素的方法。数组 S 表示一个堆栈，top 表示指针指向栈顶元素，number 表示学号，是任意正整数。下面的程序可以实现我们的要求。

程序 12-1

```
WITH student_io; USE student_io;
PROCEDURE stack IS
    max: CONSTANT: =100;
        S: ARRAY(1..max) OF positive;
        top : integer RANGE 0.. max;
        number : positive;
    PROCEDURE push (x : positive) IS
    BEGIN
        top: =top+1;
        s(top): =x;
    END push;
BEGIN--main program
      (主程序)
    top: =0;
    FOR i IN s'range LOOP
        get(number);
        puxh(number);
    END LOOP;
END stack;
```

在这个程序中，输入一个学号(number)，然后调用 push 把这个学号放在数组中。我们实际上所需要的结果是：只要把所有的学号装到一个数组中去就行了。至于把每个学号装在什么位置，已经装了多少学号等信息都不必知道。然而，由于这些变量在主过程中被说明，它们在整个程序中都是可见的。这样一来，变量没有受到应有的保护，有可能因一时疏忽而遭到破坏。

由于子程序本身没有很好的隐藏细节的功能，就需要对它建另外一种结构，使得子程序中的信息从程序的其他地方是不可见的。这就需要把一组逻辑上有联系的子程序、类型放在一起，建立一堵包墙。它可以选择(安排)哪种信息可由包外看到，哪种信息被隐藏在包内。

Ada 设计的程序包就是具有这种功能的结构。通过这种结构，我们可以化大问题为一些自完备单元(self _ contained units)，然后独立地去解决它们。程序包由两部分组成：程序包规格和程序包体。前者可由程序的其他部分查看，后者是隐藏起来的，用户不能查看它的内容。

§12-2 程序包的使用与说明

在以前的例子中，我们使用过一个程序包 student _ io。它是一个预定义程序包，使用时只要在主过程之前写上语句：

```
WITH student _ io; USE student _ io;
```

就行了。WITH 语句使得我们能够调用程序包中的内容，USE 语句又可使我们直接应用程序包中的子程序而不必写上程序包的名字。如果省略 USE 语句，仍可使用 student _ io 程序包的子程序，只是每次调用子程序时必须加上程序包的名字。例如：

```
student _ io. get (number);
student _ io. put("This is a student");
```

这样写的一个好处就是指明了哪个子程序属于哪个程序包，但用起来太麻烦。如果使用 USE 语句则可直接写成：

```
get(number);
put(This is a student");
```

怎样说明一个新的程序包呢？就象一个子程序一样，程序包可以放在一个主过程中说明，也可以放在一个子程序中或者另外的程序包中说明。但通常程序包是放在主过程的外面单独说明的。例如：

```
PACKAGE 程序包名 IS
  语句序列;                    (程序包规格)
END 程序包名;
PACKAGE BODY 程序包名 IS
  语句序列;                    (程序包体)
END 程序包名;
```

§12-3 程序包规格

程序包有两种类型：一种类型是既包含程序包规格，又包含程序包体；另一种则只包含程序包规格。下面我们就看看第二种情形的几个例子。

例 12-2 此程序包含类型 day 及一些有关的数组常量的说明。

```
PACKAGE one IS
    TYPE day IS (Mon, Tue, Wed, Thu, Fri, Sat, Sun);
    SUBTYPE weekday IS day RANGE Mon..Fri;
    tomorrow: CONSTANT ARRAY (day) OF day := (Tue, Wed, Thu, Fri, Sat, Sun, Mon);
    next_work_day: CONSTANT ARRAY (weekday) OF weekday := (Tue, Wed, Thu, Fri, Mon);
END one;
```

这就是一个完整的程序包。在这种只有程序包规格的程序包中，不能包含子程序，而只含一组组的变量、常量和类型说明。它不提供任何隐蔽的东西。这种程序包的调用和预定义程序包 student_io 的调用一样。例如：

```
WITH one ; USE one;
```

例 12-3 商场的有奖销售，每期的中奖号码放在一个程序包中，顾客要知道自己是否得奖，只需查询程序包就行了。下面就是实现这个查询的程序，其中 win_list 表示中奖号码。

```
PACKGE winning IS
    TYPE list IS ARRAY (positive RANGE <>) OF natural;
    win_list : CONSTANT list := (2546, 3468, 7236, 8921, 9112);
END winning;
WITH student_io; USE student_io;
WITH winning; USE winning;
PROCEDURE look_up IS
    --identifiers list and win_list declared in winning
      (类型 list 和常量数组 win_list 在程序包 winning 中说明过了)
    ticket_number : natural;
    FUNCTION in_list(number:natural; search_list:list)
          RETURN Boolean IS
    BEGIN
        --find of number is in the list
          (检查奖卷号码是否是中奖号码)
        FOR pos IN search_list'range LOOP
            --note multiple exit from loop
              (注意跳出循环的方式)
            IF number = search_list(pos) THEN
                RETURN true;
            END IF;
        END LOOP
        RETURN false;
    END in_list
BEGIN
    --read ticket number and check if it wins a prize
      (读入一个奖卷并检查是否得奖)
    put("What is your number ?")
    get(ticket_number);
    IF in_list (ticket_number, win_list) THEN
        put_line("Well done, you have won a prize");
    ELSE;
        put_line ("Sorry, better luck next time");
    END IF;
```

```
END loop_up;
```

程序包 winning 包含一个类型说明和一个常量数组说明。这些说明通过语句

```
WITH winning;
```

适用于其它的程序包或主过程。

由于主过程依赖于程序包 student_io 和 winning，所以要写：

```
WITH student_io; USE student_io;
WITH winning; USE winning;
```

它们也可以写成：

```
WITH student_io, winning ;USE student_io, winning;
```

注意，WITH 语句必须写在 USE 语句之前。

现在让我们看一下程序包规格的一般形式(见图 12.1)。

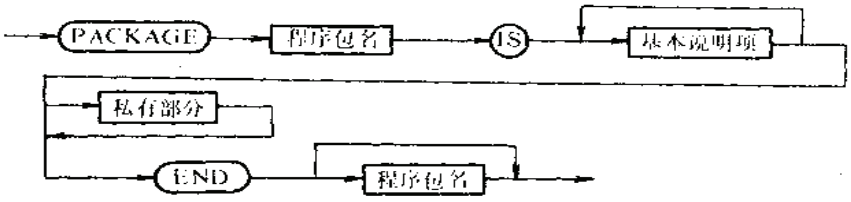


图 12.1

其中的基本说明项可以包括一个对象说明、一个类型说明、一个子程序说明、一个子类型说明、一个程序包说明或一个 USE 语句。但不能包括一个子程序体或程序包体的说明。

私有部分的形式和作用以及在程序包规格说明中的一些限制将在下一章讨论。

§12-4 使用 USE 语句要注意的问题

使用 USE 语句时，有可能出现变量之间的冲突。例如在下面的程序中，就出现了这样的问题。

```
PACKAGE example IS
    first, second : integer;
END example;
WITH example; USE example;
PROCEDURE main IS
    first : Boolean;
BEGIN
    .
    .
    .
END main;
```

在这段程序中，程序包 example 说明了一个变量 first，为整型变量。在主过程 main 中，也说明了一个变量 first，是布尔型的。而主过程又调用了程序包 example，这就引起了两个 first 的冲突。对于这种情况，Ada 规定主过程中布尔变量 first 的局部说明优先于程序包中整型变量 first 的说明。因此，在主过程中涉及到这个整型变量时，我们必须给出它的全名 example.first，即使用了语句：

```
WITH example; USE example;
```

也是一样。否则就认为指的是布尔变量 first。

因为有这种规定，所以当把程序包的内容引进到一个新的程序包或子程序中时，就不必担心会发生变量之间的冲突。

下面考虑一个更为复杂的情形。即两个互相独立的程序包包含相同的变量时应该怎么办。

```
PACKEAG example IS
    first , second : integer;
END example;
PACKAGE another _ example IS
    first, third : Boolean;
END another _ example;
WITH example, another _ example; USE example, another _ example;
PROCEDURE main IS
BEGIN
   .
   .
   .
END main;
```

在这个例子中，两个程序包都引入了变量 first，只不过类型不同。而主过程又同时调用了这两个程序包。这时为了确定变量 first 到底属于哪一个程序包，就必须使用全名 example. first 或 another _ example. first。系统本身是不能确定它到底属于哪个程序包的。因此，在主过程中可以直接使用标识符 second 和 third，但不能直接用标识符 first。

有一个例外，就是当两个互相冲突的标识符都应用于子程序或都用做枚举文字时，系统可以根据上下文来确定到底使用的是哪一个，这是第六章中遇到的重载现象。在没有熟悉 Ada 的使用之前，最好不要在程序中引入这种重载现象。即使到很熟练的时候，也不要轻易地使用。

解决一个很大的问题时，经常要用到很多个程序包，这时对不常用的类型和对象最好给出全名，而不要用它们的简写形式。这样做是为了使读者更清楚地知道它们是在什么地方说明的。如果不给出全名，也可加一条注释。

练习

1. 写一个程序包和主过程实现：给一个 91 年的日子，能够得出这个日子是星期几。

2. 下面是一个星球之间的距离的程序包。

```
PACKAGE astronomical IS
    light _ year : CONSTANT real := 5. 88E12      miles
                                                  (英里)
    parsec : CONSTANT real := 3. 26      light years
                                                  (光年)
    nearest _ star : CONSTANT real := 4. 311      light years
    earth _ to _ sun : CONSTANGE real := 9. 3E7      miles
    earth _ to _ moon : CONSTANT real := 2. 391E5      miles
END astronomical;
```

写一个程序提问下面的问题：

How far is the Earth from the Sun?

如果回答的结果与准确值相差不超过 10%，则认为正确，否则重新回答。

第十三章　程序包体

这一章我们介绍程序包体，读者从中可以看到信息是怎样隐藏在它之内的。

§13-1　程序包体的说明与定义

13.1.1　怎样通过程序包体隐藏细节

在具体介绍程序包体的说明和定义之前，还是让我们先通过一个例子来看一下它究竟是个什么样子，怎样通过它把一些程序中的细节隐藏起来的。建立好一个初步印象之后，再细谈 Ada 关于程序包体的说明和定义。

请先记住我们提到过的一点：程序包规格中的内容是可访问的（即可从程序的其它部分查看这些信息），程序包体中的内容是不能访问的（即从程序的其它部分不能查看到这些信息）。

分析一下上一章举过的有奖销售的例子。在这个例子中，所有的中奖号码做为一个数组被放在了程序包中。主过程中说明的函数 in_list 用来检查给定的号码是否落在中奖号码之内，进而确定是否得奖的。由此看来，它只是对放在程序包中的中奖号码进行操作，因此它应该在程序包中说明而不是在主过程中说明。

那么到底应把它放在程序包的什么位置呢？因为在主过程中，要调用这个函数来确定每次输入的号码是否为中奖号码。因此这个函数的名字在程序包的外部应该是可见的（否则就没法调用它了），这就意味着它应放在程序包的可见部分，即程序包规格中。再看一下这个函数执行具体操作的部分——函数体，它执行的操作就是检验输入的号码是否为中奖号码。这个操作的执行过程对我们来说知道不知道是无关紧要的。我们所需要的只是执行这个过程后返回的结果。因此表示这个操作的函数体应该放在隐蔽的地方——程序包体中。

注意，这里把一个函数分成了两个部分。一是函数说明，它包含函数的名字、参数的个数和类型，以及返回结果的类型。因为调用这个函数时需要这些信息，因此应把它们放在程序包规格中；二是函数体，其操作的具体内容没有必要保留在整个程序中，因为调用者只需要调用后得出的结果，所以应把它放在程序包体中。

程序包规格必须在相应的体之前给出，规格中说明的所有标识符自动适合于程序包体。这样，我们就有如下的程序包：

```
PACKAGE winning IS
    TYPE list IS ARRAY (positive RANGE < >) OF natural;
    win_list : CONSTANT list := (2546, 3468, 7236, 8921, 9112);
    FUNCTION in_list (number : natural;
                      search_list:list) RETURN Boolean;
END winning;
PACKAGE BODY winning IS
    FUNCTION in_list(number : natural;
                     search_list : list) RETURN Boolean IS
```

```
    BEGIN
        --find if number is in the list
          (检查奖卷号码是否为中奖号码)
    FOR pos IN search_list'range LOOP
        --note multiple exit from loop
          (注意跳出循环的方式)
        IF number = search_list (pos) THEN
            RETURN true;
        END IF;
    END LOOP;
    RETURN false;
  END in_list;
END winning;
```

在这个程序包中，因为函数 in _ list 在程序包规格中说明，因此可以象以前一样在主过程中被调用。

把子程序分成两部分来说明是可以的。子程序说明给出了调用这个子程序所需的全部信息，而子程序体给出了它所要完成的操作的全部细节。每个子程序说明都必须有一个子程序体。

13.1.2 程序包体的说明与定义

一个程序包体可以包括对象、类型、子类型、子程序说明和程序包说明，USE 语句，也可包括子程序体和程序包体(当然是另外的程序包体)的说明。它的语法规则如图 13.1 所示。

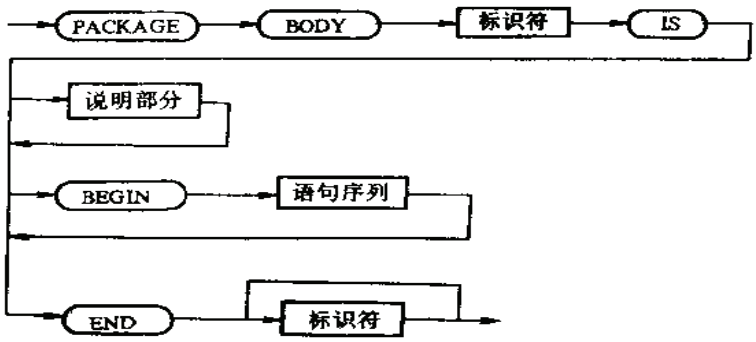


图 13.1

由语法规则可以看出，程序包体由保留字 PACKAGE 开始，后面紧跟 BODY、程序包名和 IS，然后是一个说明部分。这个说明部分可以包括对象、类型、子程序和程序包的说明。说明部分的后面是 BEGIN、语句序列、END、程序包名和分号。这一部分是可以选择的，可以要也可以不要。如果有这一部分，则它是用来对程序包进行初始化的，即对程序包中说明的变量赋初值。但这通常是不必要的，因为大多数变量都可在说明时初始化。

在一个程序包中，要区别开子程序说明和子程序体。不能把体放到程序包规格中，而只能放在程序包体中。如果程序包规格中包含有子程序说明，则程序包体中就必须包含相应的子程序体；如果程序包体中有一个子程序体，那它的对应子程序说明却不一定要放在程序包规格中，该说明也可以放在程序包体中。这时，该子程序就是一个只属于程序包内部的子程序了。要调用它则只能在程序包的内部调用，或者通过其它子程序调用，但这些子程序必须是可见的。

13.1.3 用程序包体对程序设计的优化

当我们设计一个程序包时，总是想把最少量的必要的信息放在程序包的可见部分，而在程序包体中隐藏尽可能多的信息。

应用这个原则来重新设计程序包 winning。中奖号码表仅用在函数 in_list 中，因此可以把它放在程序包体中。让一般的人接触不到这些中奖号码是很保险的。另外，函数 in_list 中的参数 search_list 可以省略掉而只保留一个参数 number。因为参数 search_list 只是中奖号码表 win_list 的形式参数，并不使用其它的实参，因此可以省略。

通过分析我们可以把程序修改成下面的形式。

程序 13-1

```
PACKAGE winning IS
    FUNCTION in_list (number , natural) RETURN
            Boolean;
    --—Search a list in the body for number
      (检查号码 number 是否为中奖号码)
END winning;
PACKAGE BODY winning IS
    TYPE list IS ARRAY (positive RANGE < > )OF natural;
    win_list : CONSTANT list : = (2546, 3468, 7236, 8921, 9112);
    FUNCTION in_list (number:natural) RETURN Boolean IS
    BEGIN
    --—find if number is in the non_local win_list
      (检查 number 是否属于数组 win_list)
    FOR pos IN win_list'range LOOP
          --—note multiple exit from loop
          (注意跳出循环的方式)
      IF number : = win_list (pos) THEN
        RETURN true;
      END IF;
    END LOOP;
    RETURN false;
  END in_list;
END winning;
WITH student_io, winning; USE student_io;
PROCEDURE look_up IS
    ticket_number :natural;
BEGIN
    --—read ticket number and check if it wins a prize
      (输入一个奖卷号码并检查是否得奖)
    put("What is your number ?")
    get (ticket_number );
    IF winning. in_list(ticket_number )THEN
        put_line("Well done, you have won a prize");
    ELSE
        put_line ("Sorry, better luck next time");
    END IF ;
```

```
END look_up;
```

只要在程序的前面有 WITH 语句，程序包规格中说明的标识符就可以应用。但程序包体中说明的标识符不行。因此在上面的程序中，只有函数 in_list 能够从程序包 winning 中存取，中奖号码表就不行。可以存取中奖号码表的唯一方式是通过函数 in_list 的调用完成的。但因为这个表不再是函数的参数，应用这个程序包的程序是不会发现这些中奖号码的。

§13-2 文字处理的例子

例 13-1 假设有一篇文章，它由很多个行组成。每个行最多不超过 80 个字符。我们要在每个行中找出是否出现下面的一些特殊词：

```
begin else end for if loop then while
```

规定文章中出现这些单词的方式可以是大写也可以是小写。

在解这个问题之前先要搞清词的意思，我们规定它为用一个非字母符号作为结束的一串字母序列。另外，如果文章中有一行只有一个字符“·”并且位于行的第一个位置，则表示文章到此结束。

考虑一下数据结构。因为每行由 80 个字符组成，所以可用：

```
line : string (1..80);
```

表示一个行。对于 8 个特殊词我们最好把它们放在一个数组中，以便于行中的每个词与它们相比较。因为每个特殊词都不超过 5 个字符，所以可把它们看成一律由 5 个字符组成，不足 5 个字符的特殊词添足空格。这样就有下面的说明：

```
SUBTYPE word IS string(1..5);
TYPE special_list IS ARRAY (positive RANGE < >) OF word;
word_list : CONSTANT special_list := ("begin ", "else ",
"end ", "for ", "if ", "loop ", "then ", "while")
```

我们可以按下面的框架设计算法。

```
LOOP;
    读入一个行
    如果读入的是最后一行就跳出循环
    LOOP 当一行中还有单词时
        取下一个词
        IF 这个词在特殊词表中
            存储这个信息
        END IF;
    END LOOP
    输出该行的特殊词
END LOOP
```

实现这个算法，需要一个过程去读入文章中的行，另一个过程去读行中的每个单词。还需要某种方式用来指出文章是否结束，行中是否还有单词要处理。处理完一行后需要有一个过程打印当前行的内容。

因为最后需要输出的结果只是每行中的特殊词，这可通过调用上面所说的一些过程来实现。这些过程是没有必要放在主过程中的，可以把它们放在程序包中。根据上面的分析，关于行的信息都

可隐藏在程序包体，过程的说明放在程序包规格中。

下面给出了有关处理的程序包。

```
PACKAGE lines IS
    PROCEDURE next_line;
    --read the next line;
      (读入一行)
    PROCEDURE get_word (the_word : OUT string);
    --get the next word from the current_line
      (从当前行中取下一个词)
    FUNCTION end_of_text RETURN Boolean;
    --true when all lines have been read
      (所有行读入完毕就输出逻辑值true)
    FUNCTION more_words RETURN Boolean;
    --true if there Could be more words in the line
    PROCEDURE write_current_line;
END lines;
WITH student_io; USE student_io;
PACKAGE BODY lines IS
    line : string (1..80);
    line_length : natural RANGE 0..line'last + 1 := 0;
    char_no : natural RANGE 0..line'last + 1 := 0;
    --char_no is the current position in line;
      (char_no是字符在行中的位置)
    FUNCTION letter (ch : character ) RETURN Boolean IS
    BEGIN
        --determine if the character is a letter
          (鉴别一个字符是否为字母)
        RETURN ch IN 'a' .. 'z' OR ch IN 'A' .. 'Z';
    END letter;
    FUNCTION lower_case(ch : character ) RETURN
            character IS
    BEGIN
        --Convert upper case letters to lower case
          (这是一个变大写字母为小写的函数)
        IF ch IN 'A'.. 'Z' THEN
            RETURN Character'val (character'pos(ch) +
                    character'pos('a')_character'pos( 'A'));
        ELSE
            RETURN ch;
        END IF;
    END lower_case;
    PROCEDURE next_line IS
    BEGIN
        --get the next line
          (取下一个行)
        get_line (line, line_length);
        char_no := 0;
```

```
END next_line;
PROCEDURE get_word (the_word : OUT string) IS
    pos : positive := the_word'first;
BEGIN
    --get next word from line starting at char_no + 1
      (从 char_no+1 位置开始取下一个词)
    --the size of the_word depends on the actual parameter
      (词的长度依赖于实际参数)
    --find start of next word
      (找出这个词的开始位置)
    LOOP
        char_no := char_no+1;
        EXIT WHEN char_no = line_length OR
                  letter(line(char_no));
    END LOOP;
    --scan the next word
    --ignore words which cannot fit into the_word
      (检查这个词,如果发现它不适合作 the_word 的实际参数,就跳过去)
    the_word := (the_word'range => ' ');
    WHILE char_no <= line_length AND THEN
          letter(line(char_no)) LOOP
        IF pos <= the_word'last THEN
          the_word(pos) := lower_case (line(char_no));
        ELSIF pos = the_word'last+1 THEN
            --word too long, reset the_word
              (这个词太长,需重新给 the_word 赋值)
            the_word := (the_word'range => ' ');
        END IF;
        char_no := char_no+1; pos := pos+1;
    END LOOP;
END get_word;
FUNCTION end_of_text RETURN Boolean IS
BEGIN
    --Check for terminating line
      (检查是否为结束行)
    RETURN line_length = 1 AND THEN line (1) = '.';
END end_of_text;
FUNCTION more_words RETURN Boolean IS
BEGIN
    --can there be more words on the line?
      (检查当前行中是否还有单词)
    RETURN char_no < line_length;
END more_words;
PROCEDURE write_current_line IS
BEGIN
    put_line(line (1..line_length));
END write_current_line;
```

```
END lines;
```

在上面的程序包中，因为函数 letter 和 lower _ case 仅用在程序包中，所以它们是在程序包体中说明的。这就是说它们不能被程序包的外部调用。

程序包 lines 只处理了关于行的操作。还有一些信息没有包括在 lines 中，如特殊词表和查找这个表所需的函数等。不把它们放在 lines 中是因为它们和行上的操作没有什么逻辑上的联系。我们将把这些信息放在一个叫 special _ word 的程序包中。

```
PACKAGE special _ word IS
    SUBTYPE word IS srting (1..5);
    FUNCTION in _ list(new _ word : word )RETURN Boolean;
        --in _ list searches a list for occurrances of new _ word
          (检查词 new _ word 是否属于表中的特殊词)
END special _ word;
PACKAGE BODY special _ word IS
    TYPE special _ list IS ARRAY (positive RANGE < >) OF word;
    word _ list :CONSTANT special _ list := ("begin ", "else ",
            "end ", "for ", "if ", "loop ", "then ", "while");
    FUNCTION in _ list (new _ word : word ) RETURN Boolean IS
    BEGIN
        --find if new _ word is in the non _ local word _ list
          (检查单词 new _ word 是否在表 word _ list 中)
        FOR pos IN word _ list'range LOOP
            --note multiple exit from loop
              (注意跳出循环的方式)
            IF new _ word =word _ list(pos) THEN
                RETURN true;
            END IF;
        END LOOP;
        RETURN false;
    END in _ list;
END special _ word;
```

有了上面这两个程序包之后，要解决我们的问题只要写一个主过程调用两个程序包规格中说明的子程序就行了。

程序 13-2

```
WITH student _ io, lines, special _ word;
USE student _ io, lines,special _ word;
PROCEDURE find _ words IS
    --line manipulation subprograms declared in package lines
      (调用程序包中的子程序)
    next _ word : special _ word. word;
    word _ found : Boolean := false;
BEGIN
    --Program to find ocurrences of certain words in a text
      (在文章中找出特殊词的程序)
    LOOP
        --read the next line
          (读入一行)
```

```
            next_line;
            EXIT WHEN end_of_text;
            --take each word and check it against the special list
            word_found := false;
            WHILE more_words LOOP
                get_word (next_word);
                --find if the word is in the list
                  (检验这个词是否为表中的词)
                IF special_word.in_list (next_word ) THEN
                    put_line(next_word);
                    word_found := true;
                END IF;
            END LOOP;
            IF word_found THEN
                put_line ("found in ");
                write_current_line;
            END IF;
        END LOOP;
    END find_words;
```

§13-3 程序中不同部分的相互影响

一个 Ada 程序是由许多程序包和主过程组成的。主过程调用程序包规格中说明的子程序完成所需要的操作。

一个程序包是通过它的可访问部分与程序的其余部分发生联系的。因此，程序包体的任何改变都不影响到程序的其余部分，而程序包规格的改变就会产生这种影响。因此，在对程序做修改时，不要轻易地去修改程序包规格。

在程序包 special_word 中，和外界的联系是由一个输入变量类型为 word 、输出结果类型为布尔型的函数来实现的。这个函数就是 in_list，它的体是隐藏在程序包体内的。如果我们按照下列方式改变函数 in_list 的体：对每个 word 型参数的可能的值，它都按照执行原来的函数体所应得的答案给出相同的布尔型结果，则我们就能肯定这种改变对程序的其余部分没有丝毫的影响。

当我们设计一个大程序时，非常需要这种特性。这就好比改装一辆汽车，需要换哪个零件的话就单独换，没必要把整个汽车都换掉。同样，要修改一个程序时，只需要改变它的一部分——程序包体。这种变化对程序的其余部分带来的影响是通过程序包规格中说明的项引起的。如果一个程序包规格很小，则它对程序的其余部分影响就很小。那么，怎样去控制一个大的复杂系统的不同部分之间的相互影响呢？

现在考虑在函数 in_list 中，如果要检索的特殊词不是 8 个而是几千几万时，有无必要改变一下检索方法。

在原来的函数 in_list 中所用的查询方法是非常简单的线性查询。首先从表的一端开始，对逐个的特殊词进行查看直到找出要找的词或者到达表的另一端。这种方法的缺点是速度慢。

我们介绍一种双向检索的方法。使用这种方法进行查询时有一个条件，就是所要检索的词表必须是有序的。在这里我们可以不考虑这个问题，因为在 word_list 中的词是按字母顺序排好了的。

进行双向检索时，首先把刚在查找的项和居于表中间位置的分量比较，有三种可能：

1. 这一次刚好就找到了要查找的词；

2. 如果发现查找项的值小于中间分量的值，就说明这个词在表的前半部分；

3. 如果发现这个项的值大于中间项的值，就说明这个词是在表的后半部分。

这样比较一次之后，我们或者发现了检索对象，或者取表的一半来重新考虑。这样一步步进行下去，要么找到所要的词要么最后已经没有词可找了。应用这种方法，因为每次比较都可把表缩短一半，所以我们可以很快得出结果。具有 1000 个特殊词的表最多只需要十次比较，而如果用原来的线性检索方法则平均需要 500 次比较。因此双向检索速度快得多。

下面就是用双向检索方法写出的函数 in _ list。

```
FUNCTION in _ list(new _ word:word) RETURN Boolean IS
    middle : positive;
    low : positive := word _ list'first;
    high : natural := word _ list'last
BEGIN
    LOOP
        --find if new _ word is in word _ list (low..high)
          (检查 new _ word 是否在 word _ list(low..high)中)
        middle := (low+high)/2
        IF new _ word < word _ list(middle) THEN
            high := middle-1;
            --range of new list is (low..middle-1)
              (现在要考虑的表范围是(low..middle-1))
        ELSIF new _ word > word _ list (middle) THEN
            low := middle+1;
            --range of new list is (middle+1..high)
              (现在要考虑的表的范围是(middle+1..high))
        ELSE
            --item found
              (已经发现要找的词)
            RETURN true;
        END IF;
        --range of new list is (low .. high)
          (现在要考虑的表的范围是(low..high)
        EXIT WHEN low > high ;
        --when list is empty
                                (检查表范围是否为空)
    END LOOP;
    RETURN false;
END in _ list;
```

这样，改变后的函数 in _ list 给出的结果和原来一样，不同的只是现在的运算速度快了，因为这些改变都是在程序包体中进行的，所以对整个程序没有什么影响。只要用新的 in _ list 替换原来的 in _ list，其它地方都不需改变。

练习

1. 我们设计程序包时，为什么总是把最少信息放入程序包规格中，而把尽可能多的信息放入程序包体中？

2. 写一个程序包包含下面信息：本学期所学的课程以及任课教师，把课程名和教师名隐藏在程序包体中。然后写一

个过程具有下面的功能：给定一个课程名和教师姓名，如果这位教师教的刚好是这门课，就返回 Boolean 型值 true，否则返回 false 值。

第十四章　私有类型

前面我们已经看到，程序包可以把对象、类型、以及子程序说明组织在一起。程序包体能够把内部对象对程序包的用户隐蔽起来，而私有类型则使我们把类型的构造细节对用户隐蔽起来。

§14-1　为什么要引入私有类型

我们可以用程序包说明类型和子程序，这使我们可以建立新的数据类型以及它们的运算。类型怎样表示和运算怎样建立等细节都隐藏在程序包体内，新类型的使用者只需知道怎样说明一个对象以及对对象进行什么样的操作或运算。由于不能够了解细节，我们必须从抽象角度去考虑这种类型，正象使用整数的人仅需对什么是整数有一个抽象的认识一样。一个抽象类型被说明后，它可以象任何预定义类型一样被使用。如果说明了一个属于抽象类型的对象，就不要试图去发现它的内部表示。对新对象的运算应完全由程序包中说明的子程序来实现。但有时情况并不是这样，请看下面的例子。

例 14-1　这是一个处理字符的堆栈的例子。在程序包中说明了一个数据类型 stack，以及在这种数据类型的对象上的两种操作：push 和 pop。

```
PACKAGE char_stack IS
    TYPE values IS ARRAY (1..100) OF character;
    TYPE stack IS
        RECORD
            item : values;
            top : natural := 0;
        END RECORD;
    PROCEDURE push (x: character; st : IN OUT stack);
    PROCEDURE pop(x: OUT character; st: IN OUT stack);
END Char_stack;
WITH student_io; USE student_io;
PACKAGE BODY Char_stack IS
    PROCEDURE push(x: character; st: IN OUT stack) IS
    BEGIN
        --add Character to stack
          (往栈中加入字符)
        IF st.top=100 THEN
            put_line ("Stack is already full");
        ELSE
            st.top := st.top+1;
            st.item (st.top) := x;
        END IF;
    END push;
    PROCEDURE pop(x: OUT Character; st: IN OUT stack ) IS
    BEGIN
```

```
        --remove top Character from the stack
          (从栈中取出栈顶字符)
        IF st. top=0 THEN
            put _ line ("Stack is empty");
        ELSE
            x : =st. item (st. top);
            st. top : =st. top-1;
        END IF;
    END pop;
END char _ stack;
```

建立了这样一个程序包之后，我们就可以应用 push 和 pop 两种运算对类型为 stack 的对象进行操作。然而，除了应用这些定义好的运算之外，也可以有另外的方法改变类型为 stack 的栈。例如，若有一个对象 stack _ 1。

```
stack _ 1 : stack;
```

则往栈 stack _ 1 中加一字符"*"的操作可以用语句：

```
push('*', stack _ 1)
```

或

```
stack _ 1. top : =stack _ 1. top+1;
stack _ 1. item (stack _ 1. top) : ='*'
```

来实现。由此引出的后果是对栈操作的不一致并容易导致错误，另外就是我们所定义的新类型没有显示出它的实际价值：使使用新类型的人把这种类型的对象作为一个整体（不能分拆的）来考虑在它上面的操作，不能让用户注意到这种抽象类型的内部表示。

怎样避免这种不一致以及由此引起的混乱呢？用 Ada 定义的私有类型就可以达到这个目的。

§14-2 私有类型的定义

私有类型的定义由下面的语法图（图 14.1）给出。

图 14.1

它由保留字 PRIVATE 开始，后面跟着一些类型和对象说明。

前面我们提到程序包规格中的东西是可见的。如果我们在程序包规格中说明了一个新的类型，而我们又不想让用户知道这种类型的对象的内部表示，则我们可以把它说明成私有类型。在私有类型中引入的标识符是不可见的。

现在我们就可以把上一节中的程序包改写成下面的形式。

```
PACKAGE Char _ stack IS
    TYPE stack IS PRIVATE;
    PROCEDURE push(x:character; st : IN OUT stack);
    PROCEDURE pop(x:OUT character; st : IN OUT stack);
    PRIVATE
        TYPE values IS ARRAY (1..100) OF Character;
        TYPE stack IS
```

```
        RECORD
            item:values;
            top:natural:=0;
        END RECORD;
END char_stack;
```

注意这个程序包的写法。因为要想对类型为 stack 的对象进行操作，类型名 stack、过程 push 和 pop 都应是可见的，所以它们都被放在程序包的可见部分。而我们又不想让用户知道类型为 stack 的对象的内部表达细节，因此把它定义成私有类型。在从 PRIVATE 开始的私有部分中引入的标识符都是不可见的，即我们不能在程序包外用标识符 values、item 和 top。这样一来，对类型为 stack 的对象进行的操作就只能是程序包中定义的两种操作 push 和 pop，不能用其他任何方式存取栈中的元素。

一旦说明一个私有类型及对于它的操作，我们就可以象使用预定义数据类型一样去使用私有类型，但不可试图去发现它的内部结构细节。

例 14-2 说明一个复数类型并定义复数之间的加、减、乘、除运算。

```
PACKAGE Complex_numbers IS
    TYPE Complex IS PRIVATE;
    i: CONSTANT Complex;
    FUNCTION "+"(x, y: Complex) RETURN Complex;
    FUNCTION "-"(x, y: Complex) RETURN Complex;
    FUNCTION "*"(x, y: Complex) RETURN Complex;
    FUNCTION "/"(x, y: Complex) RETURN Complex;
PRIVATE
    TYPE Complex IS
        RECORD
            rl, im : real ;
        END RECORD;
    i: CONSTANT Complex := (0.0, 1.0);
END;
RACKAGE BODY Complex_numbers IS
    FUNCTION "+"(x, y : Complex)RETURN Complex IS
    BEGIN
        RETURN (x.rl+y.rl, x.im+y.im);
    END "+"
    FUNCTION "-"(x, y : Complex)RETURN Complex IS
    BEGIN
      RETURN (x.rl-y.rl, x.im-y.im);
    END "-";
    FUNCTION "*"(x,y:Complex) RETURN Complex IS
    BEGIN
        RETURN (x.rl * y.rl-x.im * y.im, x.rl * y.im+x.im * y.rl);
    END "*";
    FUNCTION "/"(x,y: complex ) RETURN Complex IS
        d:real := y.rl * *2+y.im * *2;
    BEGIN
        RETURN ((x.rl*y.rl+x.im*y.im )/d, (x.im*y.rl-x.rl*y.im)/d);
    END "/";
```

```
END Complex_numbers;
```

在这个程序包中，我们定义了一个私有类型 Complex(即复数类型)以及复数之间的四种运算。在程序包的可见部分说明了一个常量 i。由于这时不知道复数类型 Complex 的内部表示细节，并且不能对它赋初值，所以只能说明它是一个常量。在 PRIVATE 后面即私有部分里，必须给出 Complex 的内部表示细节及常量的初始值。

注意：由于 Ada 是强类型的，因此我们不能把实数和复数放在一起进行混和运算。如果 c 是一个类型为 Complex 的对象(即复数)，则：

2.0+C

这样的表达式是不允许的。要进行这样的运算，必须首先把 2.0 这个实数变成复数，然后才能和 C 相加。

把实数变成复数可通过设计一个函数来完成。这个函数可以写成下面的形式：

```
FUNCTION cons (r,i :real)RETURN Complex IS
BEGIN
   RETURN (r,i);
END cons;
```

这个函数的功能是从复数的实部和虚部生成复数。如对于实数 2.0，通过：

cons(2.0,0)

就变成了复数(2.0,0)。对于一个实部为 1.0，虚部为 2.0 的复数，通过：

cons (1.0, 2.0)

就可生成复数(1.0, 2.0)。

§14-3 受限私有类型

前面介绍的私有类型是一般的私有类型。它首先把一个类型说明成私有的，然后在私有部分描述这种类型的内部结构细节。对这种私有类型所定义的操作则放在程序包体中描述。由于私有类型的内部结构细节和对这种类型定义的操作的具体细节都是不可见的，我们就不能试图存取这种类型的内部结构，只能抽象地去理解它和对它进行操作。

对于一般私有类型，除了能够对它进行指定的操作之外，还可以进行赋值、等于或不等于这样的操作。如果对私有类型不允许进行后面的那些操作(即赋值、相等、不等)，则可以把它说明成受限私有类型。

受限私有类型的语法结构和一般私有类型的差不多，区别只是在类型说明中加一个保留字 LIMITED。

下面是一个整数栈的例子，其中类型 stack 被定义成受限私有类型。

例 14-3 在这个整数栈的例子中，由于 stack 是受限私有类型，预定义操作"="不能自动适用于这种类型的对象，所以专门设计了一个函数来进行这种操作。

```
PACKAGE int_stack IS
   TYPE stack IS LIMITED PRIVATE;
   PROCEDURE push (st:IN OUT stack; x:integer);
   PROCEDURE pop(st:IN OUT stack; x: OUT integer);
   FUNCTION "="(st, tt : stack ) RETURN Boolean;
PRIVATE
   max : CONSTANT := 100;
```

```
    TYPE integer_vector IS ARRAY (integer RANGE < >) OF integer;
    TYPE stack IS
        RECORD
            S: integer_vector(1..max);
            top : integer RANGE 0..max:=0;
        END RECORD;
END int_stack;
PACKAGE BODY int_stack IS
    PROCEDURE push (st:IN OUT stack; x:integer) IS
    BEGIN
        st.top:=st.top+1;
        st.s(st.top):=x;
    END push;
    PROCEDURE pop(st: IN OUT stack; x:OUT integer) IS
    BEGIN
        x:=st.s(st.top);
        st.top:=st.top-1;
    END pop;
    FUNCTION "="(st, tt:stack) RETURN Boolean IS
    BEGIN
        IF st.top/=tt.top THEN
            RETURN false;
        END IF;
        FOR j IN 1.. st.top LOOP
            IF st.s(i)/=tt.s(i) THEN
                RETURN false;
            END IF;
        END LOOP;
        RETURN true;
    END "=";
END int_stack;
```

在这个程序包中，类型 stack 为私有类型。这种类型的每个对象都包含数组 S 和整数 top 的记录。注意 top 有省缺值 0，这样当我们说明一个栈对象时，就自动进行了初始化。对栈对象不能象以前一样进行赋值或给初始值，因为现在的私有类型是受限的。对它们进行的操作只有专门定义的 push、pop 和"="。

如果在一个复合类型中含有一个或多个受限私有类型分量，则这个复合类型也被认为是受限私有类型。

设置受限私有类型的好处是可对类型的对象进行完全的控制，对复制进行检查。

§14-4 注意事项

1. 受限私有类型的变量说明中不能包含明显的初始值。
2. 受限私有类型的参数不能有隐含值。
3. 不能在定义受限私有类型的程序包之外说明此受限私有类型的常量。

练习

1. 写一个能够对实数和复数做混和乘法的函数"*"。
2. 设计一个程序包，对你的各项财产分别定义不同的数据类型，对于贵重物品定义成受限私有类型。

第十五章　Ada 程序结构

设计大型程序时遇到的主要问题是程序的长度和复杂性。为了解决这个问题，我们就把它化分成一些自包含单元(self _ contained units)。这些单元所包含的就是逻辑上相关的一些对象、类型和子程序的说明，单元之间只能通过很小的严格定义的界面互相影响。这里的自包含意思就是指相对独立的意思。在 Ada 语言中，我们把这些自包含单元编制成了不同的程序包。

一个 Ada 程序是由一系列编译单元组成的。每个单元通常包含一个主过程、一个程序包规格和一个程序包体。每个程序包有一个相对小的规格，这个规格就是和其他单元之间的界面。程序包的功能隐藏在程序包体内。不同的编译单元之间的相互依赖关系取决于它们的上下文子句(即 WITH 语句、USE 语句)。按照这种方式，单元之间的相互作用保持在最小程度且清晰可见。

本章介绍与程序结构有关的一些概念、程序编译问题及程序设计的一些问题。

§ 15-1　有关编译单元的概念

15.1.1　库单元

库单元可以是一个子程序说明、子程序体或是一个程序包规格。

15.1.2　二级单元

二级单元是一个程序包体。

在前面关于编译单元的定义中，我们说编译单元通常包含一个主过程、一个程序包规格和一个程序包体。现在根据库单元和二级单元的概念，我们可以说一个编译单元由一个上下文子句后面跟一个库单元或一个二级单元组成。

请看下面一段程序：

```
WITH useful _ things ; USE useful _ things;
  PACKAGE outer IS
        .
        .
        .
  END outer;
WITH student _ io; USE student _ io;
  PACKAGE BODY outer IS
        .
        .
        .
  END outer;
WITH student _ io, outer; USE student _ io, outer;
PROCEDURE start IS
        .
        .
        .
END start;
```

这个程序由程序包 useful _ things、outer、student _ io、主过程 start 及 Ada 提供的标准程序包 standard 组成。

在这个程序中有五个库单元：程序 useful _ things、outer、student _ io、standard 及主过程 start；有下面一些二级单元：程序包 outer、student _ io 和 standard 的体，及程序包 useful _ things 的体；程序包 student _ io 和 useful _ things 可能还要依赖于其它的库单元，只是这种信息我们在这里没法看到。

15.1.3 程序库

一个 Ada 程序中的所有单元组成程序库。程序库包含预定义程序包以及人们为解决某种问题而建立的程序包及子程序。

人们在长期的软件设计工作中，肯定会积累一批有用的程序包和子程序。也就是说有一些现成的库单元。在进行一项新的软件工程时，我们就没有必要从头开始工作，而是把一些现成的库单元收集到一个程序库中去，用这个程序库中的单元编制程序就可以解决这个新的软件工程。

15.1.4 确立

确立是指对说明（类型说明、对象说明等）的确立，这是一种处理过程，通过这个过程说明才能发挥作用。

在一些简单情况下，确立可以把一个名字和某个对象联系起来并可能给它赋一个初值。

对于前面我们提到的那个程序，在主过程开始运行之前，它所依赖的程序包 standard、student _ io 和 outer 应首先被确立。先确立程序包规格，再确立程序包体。程序包 outer 的规格被确立之前，一定要先确立程序包 useful _ things 的规格。程序包 outer 的体被确立之前，要先确立程序包 outer 和 student _ io 的规格。在程序包 student _ io 和 useful _ things 被确立之前，它们所依赖的任何其它的程序包的规格必须先被确立。

从这里我们看出，任何 Ada 程序中的程序包都按照某种确定的路线依次确立。需注意的是：

（1）过程 start 和程序包体 outer 都需要 student _ io，它必须在这些单元之前被确立，但只需确立一次。

（2）虽然过程 start 依赖于程序包 outer，而 outer 又依赖于程序包 useful _ things，但我们不必在过程 start 的上下文子句中提到程序包 useful _ things。直接依赖的程序包才需要给出。

（3）在程序包规格或子程序说明之前的 WITH 语句也自然适用于其体。因此在体前就没有必要重复相同的 WITH 语句了。当然，重复出现 WITH 语句也不会产生错误。

（4）只用于体的程序包不要在规格或说明前给出，否则就会减低体与规格或说明的独立性。

§15-2 分别编译

编译是在一个程序能够运行之前必须做的一项工作，它通过编译程序进行。对一个程序的开发有两个主要阶段，一是编译，另一个就是运行阶段。程序通过编译阶段就可变成机器代码（计算机只能对这些代码进行运算），然后在运行阶段首先确立说明，然后执行语句。

程序在编译时可以对几个单元同时编译，也可以对它们分别编译。分别编译并不是独立地编译，因为每个编译单元依赖于它的上下文子句中所提到的库单元。在新的单元被编译之前，这些库单元应该被提前编译好并放在程序库中。如果编译成功，新的单元也会被放在程序库中。

分别编译可以把一个程序包以及它的上下文子句中提到的其它库单元与整个程序的其余部分

区别开来。对它进行编译之后如发现没有错误，则可把它放在程序库中并可用于其它程序包的上下文子句中进行这些程序包的编译。

对一个单元进行编译之后，如发现有错误，改正错误后需重新编译，并且依赖于它的其它单元都需重新编译。

现在看一下 15.1.2 中提到的那段程序。各单元之间的关系可用图 15.1 描述。

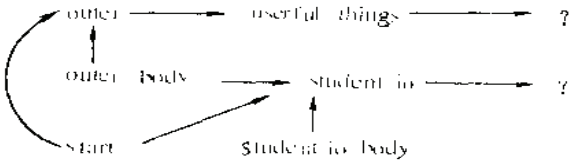


图 15.1

箭头所指的方向就是各单元所依赖的其它单元。

如果发现程序包 outer 的规格中有错误，则只需对 outer 的规格和体以及过程 start 进行重新编译。程序包 useful _ things 和 student _ io 不依赖于 outer，因此没必要重新编译。

如果在 useful _ things 中发现错误，则改正错误后，useful _ things 的规格和体（如果有的话）、outer 的规格和体以及主程序 start 都需重新编译。对一个程序包进行编译时，可以把规格和说明放在一起编译，也可以分别编译，但这时编译顺序是规格在前、体在后。依赖于一个程序包的编译单元只依赖于这个程序包的规格，而不依赖于体。

如果在 outer 的体中发现错误，改正这个错误时又没有影响到规格，则只需对这个体重新编译就行了，其它的单元都可原封不动。

当我们处理一个由许多编译单元组成的大程序时，只改变并重新编译一部分单元而不影响到整个程序，这种功能可以节约不少时间。

当一个编译单元被重新编译之后，它就要在程序库中把原来的替换掉。

有关编译次序的一般规则和确立差不多，即对一个单元的编译必须在此单元所依赖的全部单元都编译完之后进行。若一个单元改动并重新编译，则所有依赖于它的单元都需重新编译。

§15-3 子单元

程序包、子程序的体可以从直接包含它们的库单元或二级单元中取出，进行单独编译。这样，就需要一个体存根把体替换出来。替换出来的单元就叫子单元。例如，对于前一章中给出的程序包 char _ stack，如果我们把过程 push 和 pop 的体取出，则 char _ stack 的体变成：

```
PACKAGE BODY char_stack IS
  PROCEDURE puch(x:character; st:IN OUT stack) IS
    SEPARATE;
  PROCEDURE pop(x:OUT character; st:IN OUT stack) IS
    SEPARATE;
END char_stack;
```

移出来的体应在前面写上保留字 SEPARATE，以及放在括号里的包含这个体的单元的名字。这里是 char _ stack。如子单元 push 的形式为：

```
SEPARATE (char_stack)
  PROCEDURE push(x:character; st:IN OUT stack) IS
```

```
BEGIN
  --add character to stack
  (往栈中加入字符)
  IF st.top=100 THEN
    put_line("Stack is already full");
  ELSE
    st.top:=st.top+1;
    st.item(st.top):=x;
  END IF;
END push;
```

过程 pop 的形式类似。

相对于子单元，包含移出的体的单元叫做父单元。如果父单元自身也是一个子单元，则移出的体前的名字要用打点的办法写全。如 a 是 b 的子单元，b 又是库单元 c 的子单元，则这时 a 这个子单元的前面必须写成：

SEPARATE(c.b)

子单元的编译须放在父单元之后。

§15-4 注意事项

1. 一个库单元必须在其 WITH 语句中提到的其它库单元之后编译。
2. 子单元必须在其父单元之后编译。
3. 必须在相应的说明(规格)之后编译。
4. 加在程序包前的 WITH 语句只应当指明与它直接有关的那些程序包。
5. 当 WITH 语句加在一个程序包规格之前时，它的作用范围延伸到相应的程序包体。

练习

1. 如果一个程序包体在修改之后被重新编译，依赖于这个程序包的其它程序包和子程序是否需要重新编译？
2. 分析前面几章中的程序，找出每个程序中共有多少库单元和二级单元，并用图画出这些单元的编译顺序。

第十六章　可见性和作用域

在前面的章节中，我们曾提到过标识符的可见性和作用域。这是 Ada 语言的两个很重要的概念。Ada 关于可见性和作用域的规则是十分复杂的，在这里我们只介绍一部分。

§16-1　可见性

关于可见性的一个最重要的规则就是所有的标识符必须在使用之前说明，并且只能在其可见区域使用。

当标识符在程序包和子程序的说明部分被说明之后，它在从被说明之后到程序包和子程序结束之前这个区域中都是可见的。如果这些程序包和子程序内又包含了其它的程序包和子程序，则外层说明的标识符在其内也是可见的。但内层说明的标识符则不然。

例 16-1

```
PROCEDURE outer IS
  Val : CONSTANT positive := 1;
  ch : character;
  PROCEDURE enclosed (flag : Boolean ) IS
    one : posifve := Val;
    Val:CONSTANT Boolean := true;
    answer : Boolean := Val;
    ein : positive := outer. val;
    BEGIN
      .
      .
      .
  END enclosed;
BEGIN
      .
      .
      .
END outer;
```

在这个例子中，过程 outer 说明的标识符在过程 enclosed 中是可见的，而 enclosed 中说明的标识符就不能在它之外看到。

在过程 outer 中标识符 Val 被说明成一个正常数而在 enclosed 中又被说明成了一个 Boolean 型常数。因为一个标识符只有在它被说明之后才是可见的，因此在 enclosed 中的第一个说明：

```
one : positive := val;
```

表示的是把 val 的值 1 送到 one 中。而经过说明：

```
Val: CONSTANT Boolean := true;
```

之后，整型常量 val 暂时隐蔽起来，现在的 val 是一个 Boolean 型常量，直到：

```
END enclosed;
```

如果在这期间还想用 positive 型常量 val，则需写成 outer. val 的形式。

在同一说明部分不能有两个相同的标识符，除非它们是有不同参数的子程序或者是枚举文字、

这就是所谓的重载现象。除非有特别重要的理由，否则在程序编制中最好不要使用重载。

另外要注意的是，一个标识符不能用在它自己的说明中。例如：

```
border : string (1..10) := (border'range => '*');
```

是非法的。

关于程序包中的可见性规则和子程序不完全一样。在程序包规格的可见部分说明的标识符在程序包外面也是可见的，但需有上下文子句作为引导。基本规则是，用 USE 语句可使在某个程序包内的标识符可见。但应注意下列特殊情况：

(1)如果一个 USE 语句引导了两个程序包(如 USE a,b)，而在这两个程序包中都有标识符 c(或其它的标识符)，这时 USE 语句不能使标识符成为可见。

(2)一个子程序或程序包中说明了标识符 c，而在这个子程序或程序包的前面又有用 USE 语句引导了一个包含标识符 c 的程序包 a，则这个 USE 语句不能使 a 中的标识符 a 在子程序或程序包中可见。要想使它在指定的子程序或程序包中可见，必须写成打点的形式 a.c。

有时一个 USE 语句可涉及到多个程序包，这时就要注意一些细节。如有嵌套的程序包：

```
PACKAGE P1 IS
  PACKAGE P2 IS
    .
    .
    .
  END P2;
    .
    .
    .
END P1;
```

则用 USE 语句引导 P1 和 P2 时可以写成：

```
USE P1; USE P2;
```

或

```
USE P1, P1.P2;
```

但不能写成：

```
USE P1, P2;
```

§16-2 对象的生存期

为了解一个对象在程序的哪个区域可以使用，需要搞清楚在程序的运行过程中对象的存在与消失。

程序运行时需要给对象分配存储空间，然后可能要对某些对象进行赋值。存储空间被释放后，对象也就随之消失。所以一个对象的生存期是和程序运行过程中它能存在的时间密切联系在一起的。

下面我们举一个例子来说明什么时候给对象分配存储空间以及赋值等。

例 16-2

```
PACKAGE outer IS
  visible : natural := 0;
  PROCEDURE examine (answer : OUT Boolean );
END outer;
PACKAGE BODY outer IS
```

```
    invisible : positive;
  PROCEDURE examine (answer : OUT Boolean )IS
    inner : character;
    loc : integer;
  BEGIN
      .
      .
      .
  END examine;
BEGIN
  invisible := 1;
END outer;
WITH outer;
PROCEDURE main IS
  result: boolean;
BEGIN
      .
      .
      .
    outer. examine(result);
      .
      .
      .
END main;
```

在这个程序中有三个编译单元：程序包 outer 的说明，outer 的体以及过程 main。

在程序运行之前，所有的编译单元中的指令都需调入计算机内存。然后主过程所依赖的程序包逐一被初始化。在这个例子中需要把程序包 outer 的规格和体初始化。对于说明部分中引入的对象需分配存储空间并赋初值，这个过程就是确立说明。如果一个说明部分中包含许多说明，则按顺序逐一进行确立。

在对程序包 outer 的规格的说明部分进行确立的过程中就对变量 visible 分配了存储空间并赋初值 0。对 outer 的体的确立包括两步：(1)对体中说明部分的确立；(2)运行体中的语句序列。第一步的结果是给 invisible 分配存储空间；第二步的结果是给 invisible 赋值 1。

现在就可以开始运行主过程了。首先确立它的说明，给 boolean 变量 result 分配存储空间，然后开始执行主过程中的语句序列。

在运行主过程的过程中需要调用 examine 这个子程序。这时就要对它的说明部分进行确立并给 answer、inner 和 loc 分配存储空间。当调用完这个子程序返回结果时，对它们所占用的存储空间就要全部释放出来。子程序中的局部变量只有在子程序被调用时才能生存，调用完毕后变量也就随之消失。而象在 outer 中说明的对象就可以在整个程序的运行过程中生存。

下面看一个递归调用的例子。

例 16-3

```
FUNCTION factorial(n :natural) RETURN positive IS
  BEGIN
  --Calcula ten factorial
  (计算 n 的阶乘)
    IF n>1 THEN
      RETURN n * factorial (n-1);
    ELSE
```

```
        RETURN 1;
    END IF;
END factorial;
```

当按照"y := factorial(3);"调用函数 factorial 时，首先给变量 n 分配存储空间并赋值 3。运行 factorial 的过程中需再次调用这个函数自身，所以需重新给 n 分配存储空间并赋值 2，继续运行下去又要调用这个函数，这时再次给 n 分配存储空间并赋值 1。这里有三个对象都叫做 n，但它们具有不同的值。在最后退出这个函数之前它们是同时存在的。当我们从函数 faciorial 的这些调用中返回时，要释放这三个对象所占的空间。释放顺序和给它们分配空间的顺序刚好相反，实际上就是堆栈操作的顺序：先进栈的元素后出栈。因此对这三个对象的空间分配是由一个栈来负责完成的。

练习

1. 区分对象的可见性和对象的生存期。
2. 当程序运行时，如果一个变量被分配了存贮空间则说它是存在的(生存)。在程序包的说明部分说明的变量的生存期与在子程序的说明部分说明的变量的生存期有什么不同？

第十七章　异　　常

在前面我们曾经提到过，如果在程序运行的过程中出了错，就会引发异常。通常出现的是 constraint _error 异常。Ada 允许在程序的运行过程中出现错误，并可用异常处理程序进行处理。本章将介绍异常的定义、处理以及异常的使用。

§17-1　预定义异常

下面的异常都是预定义异常：

```
constraint _error
numeric _error
program _error
storage _error
```

constraint _error 对应于一些超出范围的情形。例如，设有说明：

```
list : ARRAY (1..20) OF integer;
```

如果想存取分量 list(21)，则可引发 constraint _error 异常。

在数值运算出错时会出现 numeric _error 异常，例如除数为零时就要出现这种异常。如果我们想从调用的函数中返回到调用程序而又不通过 return 语句实现的话，就会产生异常 program _error。

当我们用完存储空间时就会产生异常 storage _error。如求一个很大的数的阶乘时，就有可能引发这种异常。

以上讨论的几种情形是我们经常遇到的，但也有其它的情形会引发以上这些异常。另外，data _error 异常不是由程序包 standard 定义的，而是由 student _io 定义的。这种异常可用来检查我们读入的数据是否具有正确的类型以及范围。

§17-2　异常处理

例 17-1　下面我们讨论一下 data _error 异常的处理。

假设我们要读入范围在 1 到 15 之间的整数，则可以如下定义子类型：

```
SUBTYPE small _pos IS positive RANGE 1..15;
```

读入数据时可能会出现两种情形：一是读入的数据不在 1 到 15 范围内；二是读入的数据根本就不是正整数。这两种情形的出现都将产生 data _error 异常。

为了预防出现这种异常，可以在程序中加一段异常处理程序，以便运行不致于中断。请看下面的程序。

```
PROCEDURE get _small(num : OUT samll _pos)IS
  BEGIN
    -- read number in small _pos' range
    (在子类型 small _pos 的范围之内读入数据)
```

```
--if wrong data is read repeated attempts are made
(如果数据有错就重新读入)
LOOP
  BEGIN
  get(num);
  RETURN;
  EXCEPTION
  WHEN data_error=>
    skip_line;
    put_line("Small positive integer expected");
  END;
  put_line("try again")
END LOOP;
END get_small;
```

如果在执行语句序列：

```
get(num);
RETURN;
```

时，引发 data_error 异常，就转到=>后面的语句序列。由 WHEN 开始的语句序列(直到第一个 END 为止)叫做异常处理程序。

现在我们看一下运行过程 get_small 时会发生什么情况。首先进入循环，开始执行语句：

```
get(num);
```

如果读入的是 1 到 15 范围内的整数，则继续执行 RETURN 语句，从而离开这个过程返回到调用它的程序；否则将引发 data_error 异常。这时就要转到异常处理程序执行其中的语句序列。然后执行：

```
put_line("try again");
```

接下来又回到循环的开始位置，重新读一个整数。

如果在可能发生异常的地方不写程序，情况会怎样呢？请看下面的例子。

例 17-2 考虑字符栈的处理。如果有类型说明：

```
TYPE list IS ARRAY (1..100) OF Character;
TYPE char_stack IS
  RECORD
    item : list;
    top : natural:=0;
  END RECORD;
```

POP 过程写成：

```
PROCEDURE pop(x: OUT Character; stack: IN OUT Char_stack)IS
BEGIN
  --pop top Character from the stack
  x:=stack.item(stack.top);
  stack.top:=stack.top-1;
END pop;
```

如果在 stack.top 的值为 0 时还要调用这个过程，则执行到语句：

```
x:stack.item(stack.top);
```

时就会中断。因为没有包含 constraint_error 异常处理程序，中断发生后就回到调用 pop 的程序，并

在这个程序中再次引发 constraint _error 异常。检查这个程序中有没有一个适当的异常处理程序。如果没有，则异常将继续传播，直到遇到一个异常处理程序为止，或者一直到主程序仍未找到异常处理程序，这时就只能终止程序的执行。

除了在执行语句的过程中会引发异常外，在对说明部分的确立的过程中也会引发异常。例如确立说明：

```
number : small _pos : =16;
```

时，也会引发异常。因为 small _pos 的值超过范围 1..15。

异常处理程序的语法图如图 17.1 所示。

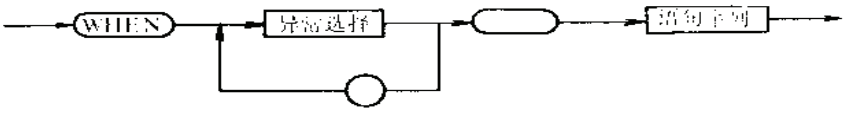


图 17.1

异常选择可以是异常名，也可以是保留字 OTHERS，但它只能有一个并且要放在最后。例如：

```
BEGIN
  --sequence of statements
  (语句序列)
  EXCEPTION
    WHEN numeric _error |constraint _error=>
      PUT("Numberic or constraint error occurred ");
      .
      .
      .
    WHEN  storage _error=>
      put("Run out of space");
      .
      .
      .
    WHEN  others=>
      put( "Something else went wrong");
      .
      .
      .
END;
```

§17-3 异常的引发和说明

如果我们不完全地去处理一个异常，则可进行某些提示后再次引发这个异常。例如，对前面提到的 pop 过程，我们可以这样写：

```
PROCEDURE pop(x:OUT Character; stack: IS OUT Char _stack) IS
BEGIN
  --pop top Character from the stack
  (弹出位于栈顶的字符)
  x : =stack. item (stack. top);
  stack. top : =stack. top-1;
  EXCEPTION
  WHEN Constraint _error=>
  put _line("Stack is empty");
```

```
    RAISE Constraint_error;
END pop;
```

对这种重复引发异常的情形，我们也可以省略掉异常名。因此异常处理部分可写为：

```
EXCEPTION
    WHEN Constraint_error=>
      put_line ("Stack is empty");
      RAISE;
```

如果要引入一个新的异常，可如下定义：

异常名：EXCEPTION；

这样，调用栈处理子程序引发的异常可定义为：

stack_error：EXCEPTION；

下面这段异常处理程序中将会引发这个异常：

```
EXCEPTION
  WHEN Constraint_error=>
    put_line("Stack is empty");
    RAISE stack_error;
```

写一个程序包，使得在任何栈处理过程中出现错误时都引发 stack_error 异常。程序包的可见部分可以写成下面的形式：

```
PACKAGE stack IS
  stack_error : EXCEPTION;
  PROCEDURE push(x: Character);
  PROCEDURE pop(x: OUT Character);
  FUNCTION stack_is_empty RETURN Boolean;
  FUNCTION stack_top RETURN Character;
  PROCEDURE reset_stack;
END stack;
```

程序包体的内容对使用程序包的单元是隐蔽起来的。其中的过程 push 可以写成下面的形式：

```
PROCEDURE push(x: character) IS
BEGIN
  --and Character to stack
  (把字符压入栈中)
  st.top := st.top+1
  st.item(st.top) := x;
EXCEPTION
  WHEN Constraint_error=>
    put_line("Stack is already full");
    RAISE stack_error;
END push;
```

§17-4 注意事项

1. 不必要时不要使用异常。
2. 最好使用自己定义的异常，而不用预定义异常。
3. 若过程由某个异常终止，则返回的 OUT 参数或更新的 IN OUT 参数的值可能不对。

练习

1. 设计一个能处理整数的栈程序包，使其能提示执行过程 push 和 pop 时可能引发的异常。
2. 考虑下面这个过程：

```
PROCEDURE p IS
BEGIN
  p;
  EXCEPTION
    WHEN storage_error =>
      p;
END p;
```

假设栈空间只够 p 的 N 次递回调用，N+1 次调用时会引发 storag_error 异常。p 一共能被调用多少次？最后会出现什么情况？

第十八章　任　　务

任务是 Ada 语言中重要的一个部分，也是 Ada 语言与其它高级语言的重要区别之一。

到目前为止，我们所考虑的程序都是按顺序执行的。但是，在许多应用中，按需要将一个程序写成几个协作并行的活动是比较有效的，尤其是在处理实时控制等问题时。

在 Ada 中，采用任务(TASK)来描述并行活动。每个任务的性质由对应的任务单元单元定义，任务单元由一个任务规范说明和任务体组成。本章将描述任务单元、任务和入口的性质以及任务间交互作用的语句(入口语句、接收语句、选择语句、延迟语句等)。

§18-1　任务说明与任务体

一个任务单元由任务说明和任务体构成，其语法描述与程序包的语法描述相类似。它以保留字 TASK TYPE 为开头的说明一个任务类型；任务类型的对象的值指明一个任务，这个任务具有任务说明中所说明的所有入口；这些入口也叫做这一对象的入口。任务的执行由对应的任务体定义。

不带保留字 TYPE 的任务说明定义一个单个的任务。以这种形式的说明所作的任务说明等价于一个无名任务类型，再立即说明该任务类型的一个对象，而任务单元标识符就是这个对象的名字。关于任务类型说明和任务对象说明以后会作用解释。

任务说明和任务体的语法图见 18.1 和图 18.2

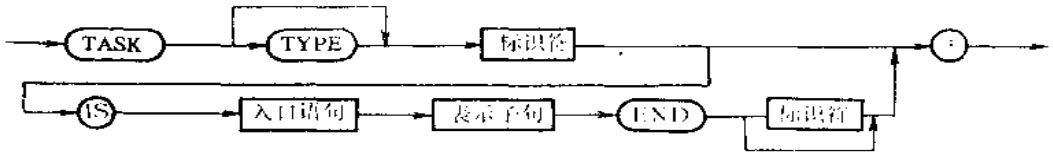


图 18.1

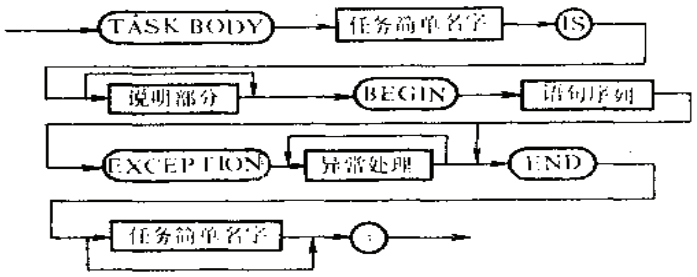


图 18.2

其中，在任务体开头的简单名字必须是任务单元标识符的重复。类似地，如果在任务说明或任务体的终点出现标识符，则此标识符也必须是任务单元标识符的重复。

由语法图可知，一个最简单的任务说明可以是：

```
TASK eating;
```

在下一章中我们将给出一个简单例子用以说明任务的激活和终止。

§18-2 举　　例

例 18-1 考虑一家人星期日上街采购一顿午餐，妈妈买肉、买鱼、买酒分别调用子程序 buy _ meat、buy _ fish、buy _ wine。

一个简单解法可以是：

```
PROCEDURE shopping
BEGIN
    buy _ meat;
    buy _ fish;
    buy _ wine;
END;
```

但是这对应于他们依次出门购买。现在我们考虑他们分别进行采购，然后他们也许在停车场会合，一起回家准备午餐。对于这样一个并行活动我们可采用任务来解决，可表示成：

程序 18-1

```
PROCEDURE shopping IS
    TASK   get _ fish;
    TASK BODY get _ fish IS
    BEGIN
        buy _ fish;
    END get _ fish;
    TASK get _ wine;
    TASK BODY get _ wine IS
    BEGIN
        buy _ wine;
    END get _ wine;
BEGIN
    buy _ meat;
END shopping;
```

在这种安排下，妈妈表示主处理器，直按从过程 shopping 中调用 buy _ meat，把孩子们看作是辅助处理器，完成局部说明的任务 get _ fish 和 get _ wine；它们分别调用过程 buy _ fish 和 get _ wine。

本例表明了任务的说明、激活和终止。任务是一个象程序包似的程序组成部分，对它可以用程序包类似的方式在子程序、程序包或另一任务体内加以说明。一个任务说明也可在一个程序包说明中另以说明，此时任务体必须在相应的程序包体中说明。注意，任务说明不能在另一个任务的说明中另以说明，只能在任务体中说明。

任务的激活是自动的。本例中，父母单元即 shopping 到达跟 在任务说明后的 BEGIN 时，局部任务就变成活跃了。当到这它最后一个 END 时就会终止。所以任务 get _ fish 调用过程 buy _ fish 后立即终结。只有当 shopping 中所有的任务都终结后才能脱离过程 shopping，这对应于妈妈等候爸爸和孩子们带着他们购得的物品返回家中。

在说明部分中说明了一个或多个任务的子程序、任务或一条 DECLARE 语句被称为双亲(父母),而它所说明的那些任务则是它的后代,也称为双亲所依赖的任务。因此,shopping 为双亲,get _ fish 和 get _ wine 为 shopping 的后代。当任务到达其末尾时,任务就终止。当双亲所有的依赖任务都终止时它才终结。

例 18-2 要求一个过程来计算如下所定义的两个整型向量的和值与差值。

```
TYPE Vector IS ARRAY(1 : 1000) OF integer;
```

我们可以利用任务设计一个过程,使得在计算和值的同时平行地计算差值。程序如下:

程序 18-2

```
PROCEDURE sumdif (a,b ,IN vector;sum,out vector;dif; out vector;) IS
    TASK minus;
    TASK BODY minus IS
    BEGIN
        FOR i IN 1 : 1000 LOOP
            dif(i) : =a (i)-b (i);
        END LOOP;
    END minus;
BEGIN
    FOR i:= 1..1000 LOOP
        sum(i) : = a(i)+b(i);
    END  LOOP;
END sumdif;
```

其中,sumdif 为父母(双亲),任务 minus 为其后代。

18-3 任务的入口与会合(呼叫)

在 shopping 和 sumdif 的例子中,各个任务一旦置成活跃之后,除它们的双亲必须等待它们终止外,就不再有相互的影响或作用。但是,通常任务在它们的生命期中会有彼此的通讯。Ada 中利用入口与会合(也称呼叫)这一机制来完成任务间的通讯。这类似于两个人会见,完成一笔交易之后,继续各干各的。

一个任务对另一个任务中说明的入口进行调用(呼叫)致使两个任务之间出现会合,入口在任务说明中说明,其方式与过程在程序包说明中相类似。

入口说明的语法如图 18.3 所示。

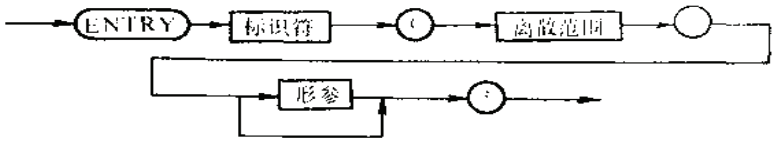


图 18.3

入口调用的语法图如图 18.4 所示。

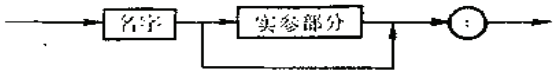


图 18.4

接受语句 ACCEPT 的语法图如图 18.5 所示。

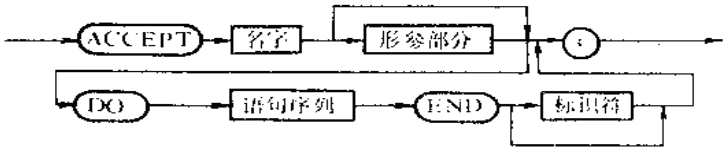


图 18.5

入口说明只能出现在任务的说明部分，它包含一个关键字 ENTRY，其后跟着入口的名字，通常还接着一个入口参数表。入口的各参数可以是有过程参数的一样形式：IN、OUT IN OUT 方式。而参数的传递规则与实参、形参结合所采用的规则完全相同。值得注意的是，这里的任务说明中只能包括入口说明而不能包括子程序类型、常量、变量说明。这一点与程序包相同。于是，任务只可能调用入口来实现通讯。

ACCEPT 语句具有一个过程体的形式，但不带说明部分。关键字 ACCEPT 后跟被接受的入口名字和形参(如果有的话)。后者的规格必须与对应的入口说明中给出的那些参数严格匹配。放在 ACCEPT 语句的 DO 和 END 之间语句序列在会合期间执行，如果会合期间不需要产生动作，那么这些语句可以省去。

入口调用和过程调用的最重要的区别是：对于过程而言，调用过程的任务还直接执行程序体；而对入口而言，一个任务调用入口，但却由拥有入口的任务执行相应的接收语句。另外，在一个任务对入口调用和拥有入口的任务到达接收语句之前，不能执行接收语句。当然其中之一可能发生，那它所相关的任务就被暂时挂起，直到另一任务到达相应语句。这时，由被调用的任务执行接受语句中的语句序列，而调用任务仍然提起。这种相互作用叫做一个会合。到当达接受语句的末尾时，会合就完成了，两个任务各自独立继续执行下去。

现在我们考虑例 18.1，给任务 get _ fish 两个入口，一个用于妈妈给孩子们买鱼的钱，另一个用于后来把鱼收集起来，对于 get _ wine 做同样的事(尽管爸爸他自己可能有钱!)。

我们可以用函数代替过程 buy _ fish、buy _ wine、buy _ meat，函数将钱作为参数，返回适当的配料。这样，买东西的过程变成：

程序 18-3

```
PROCEDURE shopping IS
    TASK get _ fish is
        ENTRY pay(m:IN money);
        ENTRY collect(s:out fish);
    END get _ fish;
    TASK BODY get _ fish IS
        cash:money; food:fish;
    BEGIN
        ACCEPT pay (m:IN money) DO
            cash := m;
        END pay;
        food := buy _ fish(cash);
        ACCEPT collect (s:out fish) IS
            s := food;
        END collect;
```

```
    END get_fish;
    TASK get_wine  IS
        ENTRY pay (m:IN money);
        ENTRY collect(s:out wine);
    END get_wine;
    TASK BODY get_wine IS
        cash,money, food,wine;
    BEGIN
        ACCEPT pay(m:IN money )DO
            cash := m;
        END pay;
        food := buy_wine(cash);
        ACCEP coolect (w,out wine )DO
            w := food;
        END collect;
  END get_wine;
BEGIN
    get_fish. pay (50);
    get_wine. pay (200);
    mm := buy_meat (300);
    get_fish. collect(ss);
    get_wine. collect(ww);
END shopping;
```

最后的结果是各种配料分别放在变量 mm,ss,ww 中,请读者考虑变量 ww,ss,mm 的说明。

提请读者注意程序的逻辑行为。一旦任务 get_fish 和 get_wine 成为活跃的,它们就遇到接受语句,等待,直到父母(双亲)单元调用各自的入口 pay。在调用函数 buy_meat 之后,调用 collect 入口。注意,妈妈只能从孩子们那儿收回鱼之后才能从父亲那里收回酒。

这里,与子程序调用一样,从任务外面调用入口时,需要用打点的方式调用。当然,一个局部任务对其父母(双亲)单元的入口可以直接调用。

一个入口可以设有参数,如:

```
ENTRY waiting;
```

对其调用可有 t. waiting。一个接受语句也可以设有体。如:

```
ACCEPT waiting;
```

在这种情况下,调用的目的只是起同步作用而不是为了交换信息。但是,一个无参入口可以有一个带体的接受语句,反之亦然。

§18-4 延迟语句

执行延迟语句 DELAY 时,首先计算它的简单表达式,然后将执行这一延迟语句的任务进一步挂起来,至少挂起由计算结果之值所指明的那一个期间;之所以说"至少",是因为它或许正在等候一个同伴的会合。

延迟语句的语法如图 18.6。

简单表达式的类型必须是预定义定点类型 duration;其值是以秒表示的;如果延迟语句中的值为负的,则等价于其表达式值为 0。

图 18.6

虽然类型 duration 与具体的 Ada 实现有关，但必需保证它所允许的最大间隔(正式负)至少要有 86400 秒(一天)。多于一天延迟可由循环来完成。同时，还要保证 duration'small 不大于 20 微秒。

例如语句

```
DELAY 3.0;
```

它将执行此语句的任务(或主程序)至少提起三秒种。

使用适当的数字说明更易表达延迟，例如：

```
seconds:CONSTANT := 1.0;
minutes:CONSTANT := 60.0;
hours:CONSTANT := 3600.0;
```

这样，我们可以写：

```
DELAY 2 * hours + 40 * minutes;
```

例如，考虑每过一个小时在用户的终端上响一次铃，响铃由调用过程 ring_bell 来实现。我们可以写成一个任务。

程序 18-4

```
TASK_ring_a_bell;
TASK BODY ring_a_bell IS
    period:CONSTANT duration := 3600.0;
BEGIN
    LOOP
        DELAY period;
        ring_bell;
    END LOOP;
END ring_a_bell;
```

DELAY 语句的语义是：实际上的延迟至少是指定的那个时间，但可能大于那个时间。此外，在循环中执行其它语句也会占用有限的时间，因此两次调用 ring_bell 的实际周期将改变。我们要尽量减少这些偏差，这是可能的。任何一个 Ada 都提供有一个叫做 calendar 的预定义程序包，其规格如下：

```
PACKAGE calendar IS
    TYPE time IS
        RECORD
            year:integer RANGE 1901 .. 2099;
            month:integer RANGE 1 .. 12;
            day:integer RANGE 1 .. 31;
            second:duration;
        END RECORD;
    FUNCTION clock RETURN time;
    FUNCTION "+" (a:time; b:duration) RETURN time;
    FUNCTION "+" (a:duration; b:time) RETURN time;
    FUNCTION "-" (a:time; b:duration) RETURN time;
```

```
    FUNCTION "-" (a:duration; b:time) RETURN time;
END calendar;
```

该程序包定义了一个 time 类型以及对 time 的操作，还定义了可用来确定当前系统时间的 clock 函数。有了这个程序包，任务 ring_a_bell 就可以重新构造。从而将会平均一小时响一次铃，而不存在由于每个循环延迟的偏差所积累起来的时间偏移。

程序 18-5

```
TASK ring_a_bell;
TASK BODY ring_a_bell IS
    period:CONSTAN duration:=3600.0;
    USE calendar;
    next_time:time:=clock( )+period;
BEGIN
    LOOP
        DELAY next_time-clock( );
        ring_bell;
        next_time:=next_time+period;
    END LOOP;
END ring_a_bell;
```

每进行一次循环，实际所需的时间是这样计算的：从下一次响铃的时间中减去从 clock 函数所得的实际时间值，这样就消除了累计误差和抵消了执行循环体中其它语句时所占用的时间。

DELAY 语句也可以作为选择对象出现在选择语句(SELECT)中，这在下一节中讨论。

§18-5 选择语句

到目前为止，在所有的例子中，ACCEPT 语句执行的顺序都是完全确定的，即按事先确定的次序接受入口调用。然而，在很多应用中，任务必须根据实际产生的调用次序来接受这些入口调用。在 Ada 中，由选择语句(SELECT)实现这一机制，允许一个任务从多个可能的会合点进行挑选。

SELECT 语句的语法如图 18.7 所示。

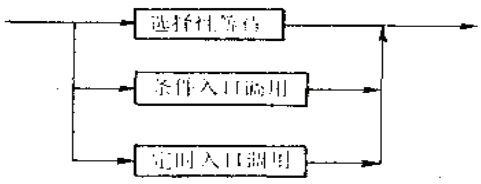


图 18.7

因此，我们有三种形式的选择语句，下面将分别讨论。

18.5.1 选择等待

选择等待和语法如图 18.8 所示。

其中，选择对象的语法如图 18.9 所示。

仔细研究语法图，可以看到各种可供任选的选择等待语句形式。一个选择等待必须至少包含一个选择对象。此外还能包含一个终止语句(只能一个)、一个或多个延迟语句、或一个 ELSE 部分。这三种可能性是两两互斥的。不过其基本形式为：

```
SELECT
    WHEN b1 => ACCEPT e1…;
OR
    WHEN b2 => ACCEPT e2…;
OR
```

…

OR

WHEN bn=> ACCEPT en…;

END SELECT;

它包含了一系列的可选择的 ACCEPT 语句。每个语句之前都有一个称为保护的 WHEN 条件。执行以上语句的作用为：

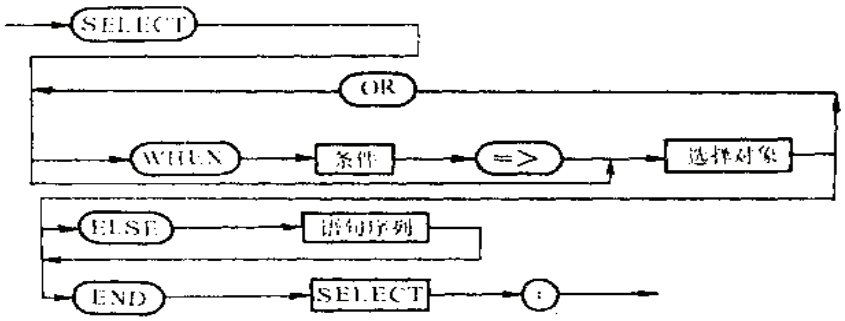


图 18.8

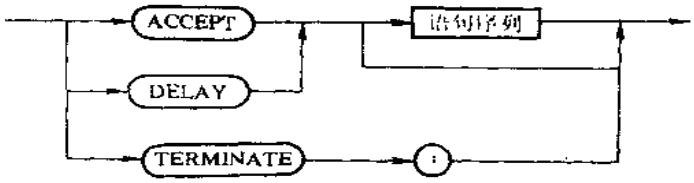


图 18.9

(1) 求从 b1 到 bn 的几个保护条件的各个值。条件值为 true 的那保护被称为打开的(开放的)，否则称为封闭的。

(2) 从入口调用正挂着的这一组打开的接受语句中，随机地选择某个接受语句。该接受语句以通常的方式执行(即与调用任务会合)。当然接受语句执行完时，选择等待语句结束。

(3) 如果对于所有打开的 ACCEPT 语句没有挂起的入口调用，于是拥有选择等待语句的任务简单地处于等待状态，直到入口调用出现为止。注意，在一个选择等待语句中的保护条件表达式绝不会再次被求值。因此，在选择语句内，当任务正处于等待状态时，如果保护发生变化，将不会影响这组可能被接受的入口调用。

(4) 当对保护求值时，如果发现它们都处于封闭状态，将会发生选择异常(select _ error)。这种情况必须从程序的逻辑上保证不会发生。

在任何 ACCEPT 选择对象前的保护都可省略，此时认为 ACCEPT 语句总是打开的。

假设一个时钟任务，把当前时间持续不断地复制到一个能被系统中所有任务来访问的变量中。该任务将以有规律的时间间隔来更新这个变量的内容，并且使系统中其它任务在执行期间都可以读取这个变量而得到当前的时间。

解决这个问题的一种方法是：将此共享变量封装在一个任务中，此任务的唯一目的就是保证对它的访问是按互拆方式进行的。其程序如下：

程序 18-6

```
TASK SHARE _ time IS
    ENTRY read (t:OUT time);
    ENTRY write (t:IN time);
END share _ time;
TASK BODY share _ time IS
    time _ current:time;
BEGIN
    LOOP
        SELECT
            ACCEPT read (t:OUT time)DO
                t := time _ current;
            END read;
        OR
            ACCEPT write(t:IN time)DO
                time _ current := t;
            END write;
        END SELECT;
    END LOOP;
END share _ time;
```

在 share _ time 的任务体内有一包含 SELECT 语句的循环体，该语句具有两个可供选择的永远打开着的 ACCEPT 语句。每次执行这条选择语句时，共有四种要考虑的可能性：

(1)对 read 或 write 都没有正挂着的入口调用；

(2)有一个对 write 的调用正挂着，但 read 没有；

(3)有一个对 read 的调用正挂着，但 write 没有；

(4)对于 read 和 write 都有调用正挂着。

对第一种情况而言，任务 share _ time 一直等待到对 read 或 write 有一个入口调用产生。对于第二或第三种情形，则接受对 write 或 read 的调用。就第四种情况而言，将随机选择一个被挂起的 read 或 write 调用。由此看来，share _ time 不考虑入口调用的顺序，它的作用就是保证所有对 time _ current 的存取是排斥的。

对于其它形式的选择对象和 ELSE 部分的选择如下进行：

(1)开放(打开的)的延迟将被选中，如果指定的延迟期间已过而没有挂起的调用；于是就执行该延迟语句后面的语句。若有几个打开的延迟语句，则任选其一。

(2)如果所有的接受语句都是封闭的或没有挂起调用，就执行 ELSE 部分的语句。

(3)当 TERMINATE 语句的双亲到达一个终止位置时它就立即终止运行，即选择开放的终止语句。但是，只要任务的任何入口还有一个挂着的调用，就不能选择终止语句。

18.5.2 条件入口调用

在两个人的约定见面中，由于某种原因会取消见面。同样地，一个入口调用者也可以避免自己去进行会合。首先它可以采用一个有条件的入口调用，其语法如图 18.10 所示。

对于条件入口调用的执行，首先计算入口名字，然后计算必要的双参。如果被调用的任务尚未到达可以接收这一调用的地点，或这一入口已有挂着的调用，那么这一调用即被取消。如果被调用的任务到达一选择语句之处，但用于这一入口的一个接收语句未被选中，那么也取消这一调用。

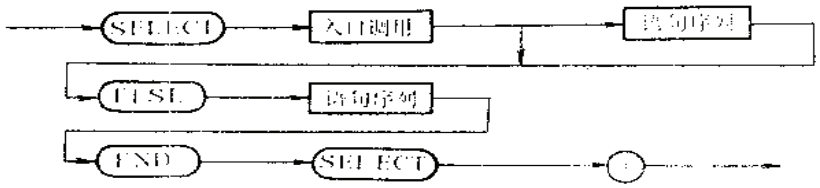


图 18.10

如果这一调用被取消，就执行 ELSE 部分的语句。否则就进行会合；然后执行入口调用后面可能出现的语句序列。

例如，执行下列语句：

```
SELECT
    buffer.put (ch);
    blocked : =false;
ELSE
    blocked : =ture;
END SELECT;
```

只有当任务 buffer 准备立即接受调用时，才会导致对 buffer.put 的入口调用。接着入口调用之后可以出现一个语句序列，这些语句序列只有在这个入口调用完成以后才被执行，本例中将 blocked 的布尔值置为 false。如果此入口调用不能立即被接受，则执行 ELSE 部分的语句序列，这时，其结果是将 blocked 的值置成 true。

18.5.3 定时(限时)调用

一个入口调用者能够使自己避免进行会合的第二种方法是构造一个定时入口调用。其语法图见图 18.11。

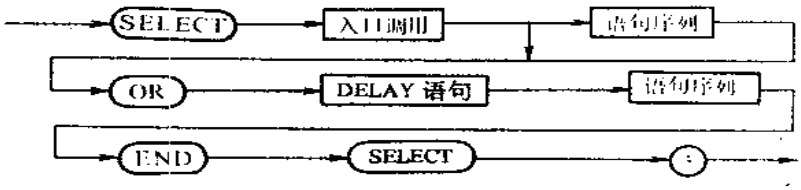


图 18.11

对一个定时入口调用，先计算入口名字，然后计算必要的实参，再后计算规定的延这期间的表达式，最后发出入口调用。

如果会合能在指定的期间之内开始，则就进行会合并执行在这一调用后面任选出现的语句。否则，当指定的期间已过时立即取消这一调用，并执行延迟语句之后可任选出现的语句序列。

例如执行语句：

```
SELECT
  buffer.put(ch);
  st : =ok;
```

```
    OR
        DELAY 30.0;
        st := =timed-out;
    END SELECT;
```

的结果是去调用在任务 buffer 中的入口 put。如果这个调用在 30 秒之内被接受，则像通常一样去进行会合，并将 st 变量置为 ok；如果超过了延迟的期限，这个调用仍没有被接受，于是这个入口调用就被取消并且变量 st 被赋值为 time _ out.

§18-6 任务类型

在有些情况下，建立几个相似但又不同的任务是必要的，因为我们常常不可能预言所需的这种任务的数目。比如说，在一个空中交通控制系统的某个控制区域内，为跟踪每架飞机建立不同的任务，此时飞机的数目是不可知的。因此，需要以与程序静态结构无关的动态方式建立和释放这种任务。而以前我们所讨论的任务均为简单任务。

在 Ada 中，利用类型来说明相似的任务模块。除了在任务说明 TASK 之后跟着保留字 TYPE 外，它与我们所述的简单任务说是完全一样的。但是，此时的任务说明定义了一个任务类型而不是实际的任务。对应任务体的确立则定义了这种类型的任务具体做什么，它并不使得一个任务激活，而是通过说明任务类型的对象来分别激活任务。

例如：

```
TASK TYPE a _ task  IS
        ENTRY an _ entry(...);
END   a _ task;
TASK BODY a _ task  IS
      ....
END a _ task;
```

任务体所遵循的规则同前。

为了建立一个实际的任务，我们可使用一般的对象说明。因而可以写

```
x:a _ task;
```

它说明一个类型为 a _ task 的任务。事实上，在这以前我们所使用的简单任务，象

```
TASK example IS
      ...
END example;
```

等价于

```
TASK TYPE anon IS
      ...
END anon;
```

后面跟着：

```
example:anon;
```

通常，只要是标准数据类型能够使用的地方都可以使用任务类型。例如，我们可以说明任务数组和包含任务的记录：

```
arr:ARRAY (1..20) OF a_task;
TYPE rec IS
    RECORD
        d:a_task;
    END RECORD;
r:rec;
```

对这种任务的入口调用为：

```
arr(1).an_entry(...);
r.d.an_entry(...);
```

值得注意的是，任务对象不是变量，其表现类似于常量。任务对象说明建立了一个永久局限于该对象的任务。不允许对该对象赋值，也不允许进行等或不等的比较。由此可见，任务类型是另一种形式的受限类型，尽管任务对象的表现很象常量，但不能象常数那样说明它们。因为常量说明需要赋于初值。子程序参数可是有任务类型，它们总是通过引用而有效地被传递，因而开参和实参总是指向同一个任务。

§18-7 任务优先级

对于不同任务，有时需要根据任务的紧迫程度给予不同的优先级。为此我们可以在任务说明中某处加一杂注 PRAGMA 指示任务的优先级。例如：

```
TASK a IS
    PRAGMA priority(8);
END a;
```

优先级必须是类型 integer 的子类型 priority 的静态表达式，但子类型 priority 的实际范围依于 Ada 的实现。注意，任务的优先级是静态的，因而不能在程序的运行过程中加以改变。较大的优先级表示较高级别的紧迫程度。多个任务可以有同一优先级，但另一方面，任务根本不必非有一明显的优先级。

到目前为止，我们已介绍了任务的主要内容，由于任务是 Ada 的重要部分和难点，读者可以参阅有关参考书以获得更多的了解。

第十九章　文　　件

几乎每种程序设计语言都有一个非常重要的概念——文件。程序的输入、修改、删除和调用等操作无不与文件的操作联系在一起。这一章我们重点介绍外部文件的形成及其操作。

§19-1　文件的形成

在编译和运行一个 Ada 程序之前，首先必须把它输入到计算机中去，然后以文件形式存储在磁盘或磁带上。这个工作由一个叫做编辑软件的特殊程序来完成。编辑软件阅读我们从计算机终端输入的东西，而后把它们放在一个外部文件中（存储在磁盘或磁带上的文件叫作外部文件）。

如果以后要改变这个 Ada 程序，就需要再次使用这个编辑软件。它读入包含原程序的外部文件的内容以及我们从终端输入的修改部分，把这两者结合起来产生一个新的程序的版本并存放在一个新的外部文件中。

不仅程序可存放在外部文件中，大量的数据也可以存放在外部文件中。例如，当运行一个程序而它所需的初始数据又很多时，从终端输入这些数据是很麻烦的，特别是运行不止一次时更是如此。数据的一部分或全部都装在一个外部文件中，程序可以从这个外部文件读这些数据。

同样，为了避免在屏幕上显示所有的计算结果，可以把结果放在一个外部文件中。当程序运行的结果只是被其它程序利用而没有必要去阅读它们时，就可以这样处理。在这种情况下，结果输出到一个外部文件中，因此它们可用来作为另一个程序的输入数据，或可作为同一个程序下一次运行时的初始数据（如要进行多次迭代运算）。

§19-2　文件的说明、打开与关闭

每个外部文件都可起一个名字，称为外部文件名。文件名由一串字符表示。

在一个 Ada 程序中，可以把文件对象说明成一个预定义类型 file _ type，例如：

```
file _ 1, file _ 2: file _ type;
```

说明了文件类型为 file _ type 的两个文件 file _ 1 和 file _ 2。

文件有三种类型，它们是：

in _ file——只读文件

out _ file——只写文件

in out _ file——读写均可的文件

文件的存取方式有顺序存取和直接存取两种，分别有相应的程序包提供这些功能。对于顺序存取的文件，in out _ file 方式是不允许的，而对于直接存取的文件则没有这个限制。有关论述请参阅《Ada 程序设计》一书（J. G. P. 巴恩斯著；张乃孝、候世君译；人民邮电出版社出版）。

在对文件进行各种操作之前，一定要先说明。说明后就可以用过程 open 和 close 打开和关闭文件。这两个过程的形式为：

```
PROCEDURE open(file : IN OUT file _ type; name : IN string);
```

和

```
PROCEDURE close (file : IN OUT file_type);
```

其中的 IN OUT file_type 表示文件的类型。

如果有一个名为 Chapter1.text 的外部文件，则可用下面的方式打开该文件：

```
open (my_file, out_file, "Chapter1.text");
```

这个语句使 file_type 类型的对象 my_file 和名为 Chapter1.text 的外部文件联系在一起，并打开这个文件，out_file 表示文件 my_file 是只写文件。

下面的语句执行后可把这个文件关闭：

```
close(my_file);
```

例 19-1 假设有多个文件，每个文件至少包含 50 个整数。写一个过程实现如下功能：从某个文件读前 50 个整数并求它们的和。具体算法如下：

```
打开数据文件
给表示和的变量赋初值 0
循环 50 次
    每次从文件中读一个整数
    将该整数与和变量相加
结束循环
关闭文件
```

这个过程需要两个参数：一是外部文件名，二是类型为 integer 的 OUT 参数，用于将和返回到调用程序。

```
PROCEDURE read_fifty(external_name : string; total : OUT integer) IS
  numbers_file : file_type;
  value : integer;
  sum : integer := 0;
BEGIN
  --read and sum the first 50 integers in the file
  (从外部文件读前 50 个整数并求和)
  open (numbers_file, in_file, external_name);
  FOR i IN 1..50 LOOP
    get(numbers_file, value);
    sum := sum + value;
  END LOOP;
  close(numbers_file);
  total := sum;
END read_fifty;
```

过程中的第一条语句：

```
open(numbers_file, in_file, external_name);
```

使 file_type 类型的对象 numbers_file 与对应于 external_name 的实际外部文件联系在一起。并由于这个文件已经被打开，因此可读出其中的数据。而语句：

```
get(numbers_file, value);
```

表示从与 numbers_file 联系在一起的外部文件中读入一个整数。这是一个重载的过程。

当过程 read_fifty 从文件中读完 50 个数并计算出它们的和之后，与 numbers_file 联结的外

部文件就由过程调用：

```
close (numbers_file);
```

关闭。这个调用截断了外部文件和类型 file_type 的对象 numbers_file 之间的联系。

假如有一个名为 people_ages 的外部文件，包含了 50 个人的年龄，要求这些年龄的和放在一个整型变量 sum_ages 中，则可用下面的调用：

```
read_fifty("people_ages", sum_ages);
```

注意这里外部文件名的写法。

§19-3 文件的建立与删除

建立文件需要用过程

```
PROCEDURE create (file : IN OUT file_type; name : IN string);
```

删除文件需要用过程：

```
PROCEDURE delete (name : IN string);
```

下面举例说明这两种过程的用法。

例 19-2 有一个外部文件包含 500 个学生的姓名及其考试成绩。每个学生的信息包含在两个连续的行中。第一行包含学生的姓名，假设每个姓名不超过 30 个字符；第二行包含学生的三门考试成绩。建立一个新文件，使之包含学生姓名和三门课的平均成绩。建好这个文件后，就把原来的外部文件删除。

```
PROCEDURE produce_averages (old_file, new_file : string ) IS
  exams, averages : file_type;
  exam_result, exam_total : natural;
  exam_average : real;
  name : string (1..30);
  name_length : natural := 0;
BEGIN
  open (exams, in_file, old_file);
  create(averages, out_file, new_file);
  FOR students IN 1..500 LOOP
    --read name and results from the next two lines
    (从外部文件读入连续的两行)
    get_line(exams, name, name_length);
    exam_total := 0;
    FOR exam IN 1..3 LOOP
      get(exams, exam_result);
      exam_total := exam_total + exam_result;
    END LOOP;
    skip_line(exams);
    exam_average := real(exam_total)/3.0;
    --write name and average on the same line
    (把名字和平均成绩写在同一行)
    put(averages, name (1..name_length));
    put(averages, exam_average); new_line(averages);
  END LOOP;
```

```
    delete(exams); close(averages);
END produce_averages;
```

这个过程中用到了几个重载的过程:put、get_line、skip_line 和 new_line。

在 put 和 new_line 过程的调用中,如果没有指定文件,则信息由屏幕输出;如果指定了文件,则信息就被写进这个文件中。

假设考试成绩放在 class_results 这个外部文件中,而平均成绩放在一个叫做 student_averages 的新文件中,则这个过程的调用可以写成:

```
produce_averages ("class_results", "student_averages");
```

当运行这段程序时,首先打开文件 class_results,并与类型为 file_type 的对象建立起联系。这时读入与 exams 对应的外部文件中的姓名和成绩,再把 500 个姓名和平均成绩写到与 averages 相应的外部文件中。这个结果是由重复执行下面的过程调用而得到的:

```
put(averages, name(1..name_length));
put(averages,exam_average); new_line(averages);
```

当所有的平均成绩都写入这个新文件后,就删除与 exams 对应的外部文件(即 class_results),然后关闭新建的文件 student_averages。

对一个文件的操作应注意以下几点:

(1)如果对一个已经打开的文件进行打开和建立等操作,或对一个没有打开的文件进行关闭和删除操作,则会引发 status_error 异常。一个文件是否打开可以用函数:

```
FUNCTION is_open (file : IN OUT file_type)RETURN Boolean;
```

来测试,如果指定的 file 是打开的,则值为 true。

例 19-3 测试文件 varages 是否已打开,如已打开则关闭这个文件。

```
IF is_open (averages ) THEN
  close(averages);
END IF;
```

(2)如果表示外部文件名的字符串有形式错误,或试图去打开一个根本不存在的外部文件,则会引发 name_error 异常。利用函数:

```
FUNCTION name (file : IN file_type) RETURN string;
```

可以得出与文件 file 相连的外部文件名。如果 file 没有打开,就会产生 status_error 异常。

(3)如果我们试图去打开、建立或删除一个文件,而对这个文件来说这些操作都是不允许的,则会产生 use_error 异常。

(4)如果试图从一个文件类型不是 in_file 的文件中读数据,或者往一个文件类型不是 out_file 的文件中写数据,则会产生 mode_error 异常。

调用函数 mode 可以找出文件的当前方式。它有一个类型为 file_type 的参数,返回值的类型是 file_mode。

§ 19-4 文件的复制

在进行文件的复制时,要注意是否已到了文件的末尾,这可由预定义函数 end_of_file 来测试。此函数与 end_of_line 类似,当读到文件结束时结果为真,否则结果为假。

现在我们写一个过程,把一个文件按照它原有的行结构复制到另一个文件中。

```
PROCEDURE copy(old_file, new_file : string ) IS
  ch: character;
  source, destination : file_type;
BEGIN
  --Copy contents of source to destination
  (把原文件的内容复制到目标文件)
  open(source, in_file, old_file);
  create(destination, out_file, new_file);
  LOOP;
    EXIT WHEN end_of_file (source);
    IF end_of_line(source) THEN
      skip_line (source);
      new_line(destination);
    ELSE
      get(source, ch);
      put(destination, ch);
    END IF;
  END LOOP;
  close(source);close(destination);
END copy;
```

正文文件是由一行行的字符组成的，很多行组成一页。过程 new _ page 的作用是结束当前行并写一页结束标志，然后开始新的一页。同样，过程 skip _ page 表示跳过当前页的剩余部分转到下一页，而过程 skip _ line 表示跳过当前行的剩余部分后转到下一行。

每一个正文文件都有一个行结束标志，后面跟一个页结束标志，再后面就是文件结束标志。当遇到这串行、页、文件的结束标志时，函数 end _ of _ file 的值也为真。

练习

1. 写一个过程：在两个已有文件的基础上建立第三个文件。该文件的内容前半部分由第一个文件形成，后半部分是第二个文件的内容。
2. 有两个文件，每个文件都包含一些按非降序排列的整数。写一个过程建立第三个文件，使得其中的数据就是原来两个文件中的数据，要求也按非降序排列。例如第一个文件为：

   3　19　467　467　543

   第二个文件为：

   3　7　9　19　219

   则新建的文件就为：

   3　3　7　9　19　19　219　467　467　543
3. 有两个社团，每一个社团的成员名字都按字母顺序排列放在一个文件中。现在这两个社团要合并。假设没有哪个成员同时属于这两个社团。写一过程，从这两个文件中读入成员的名字，并按字母顺序排列放在第三个文件中。
4. 有一个文件存放了从 1 月 1 日开始的一年的气温情况，包括每天的最高温度和最低温度以及中午的温度。求出哪一天的温度最高，哪一天的温度最低，哪一天中午的温度最高；哪一个月的最低温度平均最低，哪一个月最低温度平均最高。

附录一　保留字

下面是 Ada 语言中的所有保留字。

ABOUT ABS ACCEPT ACCESS ALL AND ARRAY AT
BEGIN BODY
CASE CONSTANT
DECLARE DELAY DELTA DIGITS DO
ELSE ELSIF END ENTRY EXCEPTION EXIT
FOR FUNCTION
GENERIC GOTO
IF IN IS
LIMITED LOOP
MOD
NEW NOT NULL
OF OR OTHERS OUT
PACKAGE PRAGMA PRIVATE PROCEDURE
RAISE RANGE RECORD REM RENAMES RETURN REVERSE
SELECT SEPARATE SUBTYPE
TASK TERMINATE THEN TYPE
USE
WHEN WHILE WITH
XOR

附录二　程序包 student _ io

Ada 语言中定义了一个程序包 text _ io。所有对文件进行操作的过程和函数如 open、create、delete 以及读写字符串的操作都是在这个程序包中定义的。本书使用的程序包 student _ io 包含了所有这些过程和函数，且形式和 text _ io 中的过程与函数完全一致。

如果要对整数或实数进行读写操作，直接应用程序包 text _ io 就会出现一个问题：即重载。读写整数或实数的过程 get 和 put 定义在 text _ io 的类属程序包中，因此不能直接用。而在 student _ io 中的这些过程可以象存取字符和串一样去用。

如果 Ada 是用于教学目的的，则一般有一个象 student _ io 这样的可以直接调用的程序包，但名字可能不同。如果名字不同，则应用时需换成与 student _ io 对应的名字。

如果我们用的 Ada 系统没有这样一个程序包，则我们的程序需作一些改动。开始的上下文子句应写成：

```
WITH text _ io; USE text _ io;
```

如果要读写的是整型数，则还需在主过程的说明部分的开始位置写上：

```
PACKAGE int _ io IS NEW integer _ io(integer);
USE int _ io;
```

如果要用实型数，则需加上说明：

```
TYPE real IS DIGITS 8;
```

要对实数进行读写操作则需要说明：

```
PACKAGE real _ io IS NEW float _ io (real); USE real _ io;
```

加上这些说明后，程序的其余部分保持原样。

附录三　关于程序包 text _ io

```
PACKAGE text _ io IS
    PACKAGE character _ io IS NEW input _ output (character);
    TYPE in _ file IS NEW character _ io. in _ file;
    TYPE out _ file IS NEW character _ io. out _ file;
    --字符输入输出
    PROCEDURE get(file   ;IN in _ file; item   ;OUT character);
    PROCEDURE get(item   ;OUT character);
    PROCEDURE put(file   ;IN out _ file; item   ;IN character);
PROCEDURE put (item;IN character);
    --串输入输出
    PROCEDURE get(file   ;IN in _ file; item   ;OUT string);
    PROCEDURE get(item   ;OUT string);
    PROCEDURE put(file   ; IN out _ file; item   ; IN string);
    PROCEDURE put(item   ;IN string);
    FUNCTION get _ string(file   ;IN in _ file) RETURN string;
    FUNCTION get _ string RETURN string;
    FUNCTION get _ line(file   ;IN in _ file) RETURN string;
    FUNCTION get _ line RETURN string;
    PROCEDURE put _ line(file   ;IN out _ file; item   ;IN string);
    PROCEDURE put _ line(item   ;IN string);
    --整型输入输出的类属程序包
  GENERIC
  TYPE num IN RANGE < >;
  WITH FUNCTION image(x   ;num) RETURN string IS num'image;
  WITH FUNCTION value(x   ;string) RETURN num IS num'value;
    PACKAGE integer _ io IS
      PROCEDURE get(file   ;IN in _ file; item   ;OUT num);
      PROCEDURE get(item   ;OUT num);
      PROCEDURE put(file   ;IN out _ file;
      item   ;IN num;
      width   ;IN integer   : =0);
      base   ;IN integer RANGE 2. . 16   : =10);
      PROCEDURE put(item   ;IN num;
        width   ;IN integer   : =0;
        base   ;IN integer RANGE 2. . 16   : =10);
    END integer _ io;
    --浮点输入输出的类属程序包
 GENERIC
 TYPE num IS DIGITS < >;
```

```
WITH FUNCTION image(x   ;num)RETURN string IS num' image;
WITH FUNCTION value(x   ;string)RETURN num IS num' value;
  PACKAGE float _io IS
    PROCEDURE get(file   ;IN in _file; item   ;OUT num);
    PROCEDURE get(item   ;OUT num);
    PROCEDURE put(file   ;IN out _file;
      item   ; IN num;
      width   ;IN integer   : =0;
      mantissa   ;IN integer   : =num'digits;
      exponent   ;IN integer   : =2);
  PROCEDURE put(item   : IN num;
    width   ;IN integer   : =0;
    mantissa   ;IN integer   : =num'digits;
    exponent   ;IN integer   : =2);
    END float _io;
--定点输入输出的类属程序包
GENERIC
TYPE num IS DELTA < >;
WITH FUNCTION image(x   ;num) RETURN string IS num'image;
WITH FUNCTION value(x   ;string) RETURN num IS num'value;
  PACKAGE fixed _io IS
    delta _image ;CONSTANT string   : =image(num'delta-integer (num'delta));
    default _decimals   ;CONSTANT integer   : =delta _image'length-2;
  PROCEDURE get(file   ;IN in _file; item   ;OUT num);
  PROCEDURE get(item   ;OUT num);
  PROCEDURE put(file   ;IN out _file;
    item   ;IN num;
    width   ;IN integer   : =0;
    fract   ;IN integer   : =default _decimals);
  PROCEDURE put(item ;IN num;
    width ;IN integer : =0;
    fract   ;IN integer   : =default _decimals);
  END fixed _io;
  --布尔型输入输出
    PROCEDURE get(file   ;IN in _file; item   ;OUT boolean);
    PROCEDURE get(item   ;OUT boolean);
    PROCEDURE put(file   ;IN out _file;
      item   ;IN boolean;
      width   ;IN integer   : =0;
      lower _case   ;IN boolean   : =false);
    PROCEDURE put(item   ;IN boolean;
      width   ;IN integer   : =0;
      lower _case   ;IN boolean   : =false);
    --枚举类型的类属程序包
```

```
GENERIC
TYPE enum IS (<>);
WITH FUNCTION image(x    :enum) RETURN string IS enum'image;
WITH FUNCTION value(x    :string) RETURN enum IS enum'value;
  PACKAGE enumeration_io IS
    PROCEDURE get(file    :IN in_file; item    :OUT enum);
    PROCEDURE get(item    :OUT enum);
    PROCEDURE put(file    :IN out_file;
      item    :IN enum;
      width    :IN integer    := 0;
      lower_case    :IN boolean    := false);
    PROCEDURE put(item    :IN enum;
      width    :IN integer    := 0;
      lower_case    :IN boolean    := false);
  END enumeration_io;
      -- 布局控制
  FUNCTION line(file:IN in_file) RETURN natural;
  FUNCTION line(file:IN out_file) RETURN natural;
  FUNCTION line RETURN natural;    -- for default output file
  FUNCTION col(file:IN in_file) RETURN natural;
  FUNCTION col(file:IN out_file) RETURN natural;
  FUNCTION col RETURN natural;  --for default output file
  PROCEDURE set_col(file:IN in_file; to:IN natural);
  PROCEDURE set_col(file:IN out_file; to:IN natural);
  PROCEDURE set_col(to:IN natural);  -- for default output file
  PROCEDURE new_line(file    :IN out_file; n    :IN natural    := 1);
  PROCEDURE new_line(n    :IN natural    := 1);
  PROCEDURE skip_line(file    :IN infile; n    :IN natural    := 1);
  PROCEDURE skip_linen    :IN natural    := 1);
  FUNCTION end_of_line(file    :IN in_file) RETURN Boolean);
  FUNCTION end_of_line RETURN boolean;
  PROCEDURE set_line_length(file    :IN in_file; n    :IN integer);
  PROCEDURE set_line_length(file    :IN out_file; n    :IN integer);
  PROCEDURE set_line_length(n    :IN integer);
  FUNCTION line_length(file    :IN in_file) RETURN integer;
  FUNCTION line_length(file    :IN out_file) RETURN integer;
  FUNCTION line_length RETURN neger;
    -- 隐含输入和输出处理
  FUNCTION standard_input RETURN in_file;
  FUNCTION standard_output RETURN out_file;
  FUNCTION current_input RETURN in_file;
  FUNCTION current_output RETURM out_file;
  PROCEDURE set_input(file    :IN in_file);
  PROCEDURE set_output(file    :IN out_file);
```

```
    --异常
    name_error: EXCEPTION RENAMES character_io.name_error;
    use_error: EXCEPTION RENAMES character_io.use_error;
    status_error: EXCEPTION RENAMES character_io.status_error;
    data_error: EXCEPTION RENAMES character_io.data_error;
    device_error: EXCEPTION RENAMES character_io.device_error;
    end_error: EXCEPTION RANAMES character_yio.end_error;
    layout_error: EXCEPTION;
end TEXT_IO;
```

附录四　预定义语言环境

本附录略述了 STANDARD 程序包的规格，此程序包定义了 Ada 语言的环境。

```
    PACKAGE  standard  IS
TYPE  boolean  IS  (false,  true);
FUNCTION"NOT"  (x:boolean)  RETURN  boolean;
FUNCTION"AND"  (x,y:boolean  )  RETURN  boolean;
FUNCTION"OR"  (x,y:boolean)  RETURN  boolean;
FUNCTION"XOR"  (x,y:boolean)  RETURN  boolean;
TYPE  short_integer  IS  RANGE…;  ---实现定义
TYPE  integer IS  RANGE  …;  ---实现定义
TYPE  long_integer    IS  RANGE  …;  --  实现定义
FUNCTION  "+"  (x:integer)  RETURN  integer;
FUNCTION  "-"  (x:integer)  RETURN  integer;
FUNCTION  ABS  (x:integer)  RETURN  integer;
FUNCTION  "+"  (x,y:integer)  RETURN  integer;
FUNCTION  "-"  (x,y:integer)  RETURN  integer;
FUNCTION  "*"  (x,y:integer)  RETURN  integer;
FUNCTION  "/"  (x,y:integer)  RETURN  integer;
FUNCTION  "REM"  (x,y:integer)  RETURN  integer;
FUNCTION  "MOD"  (x,y:integer)  RETURN  integer;
FUNCTION  "**"(x:integer;y:integer  RANGE  0..integer'last)  RETURN  integer;
      和 short_integer  AND  long  integer  相似
TYPE  short_float  IS  DIGITS  …RANGE  …;    实现定义
TYPE  float  IS  DIGITS  …RANGE  …;    实现定义
TYPE  long_float    IS  DIGITS  …RANGE  …;    实现定义
FUNCTION  "+"(x:float)  RETURN  float;
FUNCTION  "-"  (x:float)  RETURN  float;
FUNCTION  abs  (x:float)  RETURN  float;
FUNCTION  "+"  (x,y:float)  RETURN  float;
FUNCTION  "-"  (x,y:float)  RETURN  float;
FUNCTION  "*"  (x,y:float)  RETURN  float;
FUNCTION  "/"  (x,y:float)  RETURN  float;
FUNCTION  "**"  (x:float;y:integer)  RETURN  float;
    和 short_float  AND  long  float  相似
    以下字符构成标准的 ASCII 字符集，但与控制字符对应的字符字不是标识符。
    TYPE  character  IS
    (NUL,SOH,STX,ETX,EOT,ENQ,ACK,BEL,
    BS,HT,LF,VT,FF,CR,SO,SI,
    DLE,DC1,DC2,DC3,DC4,NAK,SYN,ETB,
    CAN,EM,SUB,ESC,FS,GS,RS,US,
```

```
' ', '!', '"', '#', '$', '%', '&', ''',
'(', ')', '*', '+', '-', '.', '/', ',',
'0', '1', '2', '3', '4', '5', '6', '7',
'8', '9', ':', ';', '<', '=', '>', '?',
'@', 'A', 'B', 'C', 'D', 'E', 'F', 'G',
'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
'X', 'Y', 'Z', '[', '\', ']', '^', '_',
'`', 'a', 'b', 'c', 'd', 'e', 'f', 'g',
'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o',
'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
'x', 'y', 'z', '{', '|', '}', '~', 'del);
PACKAGE  ASCII  IS        控制字符
nul :CONSTANT character : =NUL;
soh :  CONSTANT character : =SOH;
stx :CONSTANT character : =STX;
etx :CONSTANT characte : =ETX;
eot :CONSTANT character : =EOT;
enq :CONSTANT character : =ENQ;
ack :CONSTANT character : =ACK;
bel :CONSTANT character : =BEL;
bs :CONSTANT character : =BS;
ht :CONSTANT character : =HT;
lf :CONSTANT character : =LF;
vt :CONSTANT character : =VT;
ff :CONSTANT character : =FF;
cr :CONSTANT character : =CR;
so :CONSTANT character : =SO;
si :CONSTANT character : =SI;
dle :CONSTANT character : =DLE;
dc1 :CONSTANT character : =DC1;
dc2 :CONSTANT character : =DC2;
dc3 :CONSTANT character : =DC3;
dc4 :CONSTANT character : =DC4;
nak :CONSTANT character : =NAK;
syn :CONSTANT character : =SYN;
etb :CONSTANT character : =ETB;
can :CONSTANT character : =CAN;
em :CONSTANT character : =EM;
sub :CONSTANT character : =SUB;
esc :CONSTANT character : =ESC;
fs :CONSTANT character : =FS;
gs :CONSTANT character : =GS;
rs :CONSTANT character : =RS;
```

```
us :CONSTANT character := US;
del:CONSTANT character := DEL;
        其它字符
exclam :CONSTANT character := '!';
sharp :CONSTANT character := '#';
dollar :CONSTANT character := '$';
query :CONSTANT character := '?';
at sign :CONSTANT character := '@';
l bracket :CONSTANT character := '[';
back slash :CONSTANT character := '\';
r bracket :CONSTANT character := ']';
circumflex :CONSTANT character := '^';
grave :CONSTANT character := '`';
l_brace :CONSTANT character := '{';
bar :CONSTANT character := '|';
r_brace :CONSTANT character := '}';
tilde :CONSTANT character := '~';
        小写字母
lc_a : CONSTANT character : 'a';
lc_b : CONSTANT character : 'b';
  etc.
lc_z : CONSTANT character : 'z';
END ASCII;
        预定义类型和子类型
SUBTYPE natural IS integer RANGE 1..integer'last;
SUBTYPE priority IS integer RANGE ...;        实现定义
TYPE string IS ARRAY (natural RANGE <>) OF Character;
TYPE duration IS DELTA ... RANGE ...;        实现定义
        预定义例外
constraint_error : EXCEPTION;
numeric_error : EXCEPTION;
select_error : EXCEPTION;
storage_error : EXCEPTION;
tasking_error : EXCEPTION;
PACKAGE system IS        取决于机器
TYPE system_name IS impl_defined_enumeration_type;
name: CONSTANT system_name := impl_defined
storage_unit : CONSTANT := ...; impl_defined
memory_size : CONSTANT := ...; impl_defined
min_int : CONSTANT := ...; impl_defined
max_int : CONSTANT := ...; impl_defined
        任何取决于机器的其它常量
END system;
PRIVATE
```

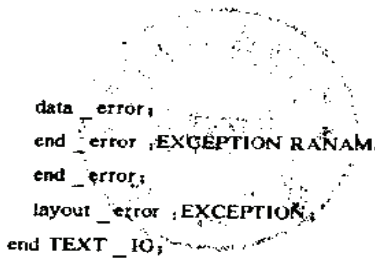
```
FOR character USE      128 ASCII CHARACTER SET WITHOUT HOLES
(0,   1,   2,   3,   4,   5,   ...,  125,   126,   127);
PRAGMA pack    (string);
END standard;
```

附录五　关于程序包 text _ io

```
PACKAGE  text _ io  IS
PACKAGE  character _ io  IS  NEW  input _ putput  (character);
TYPE  in _ file  IS  NEW  character _ io _ in _ file;
TYPE  out _ file  IS  NEW  character _ io. out _ file;
    -- 字符输入输出
PROCDDURE  get(file:IN  in _ file;  item:OUT  character);
PROCEDURE  put(file:OUT  character);
PROCEDURE  put(file:IN  out _ file;  item:IN  character);
PROCEDURE  put(item:IN  character);-- 串输入输出
PROCEDURE  get(file  :IN  in _ file;  item:IN  string);
PROCEDURE  get(item:OUT  string);
PROCEDURE  put(item:IN  string);
FUNCTION  get _ string(fille:IN  in _ file)  RETURN  string;
FUNCTION  get _ string  RETURN  string;
FUNCTION  get _ line  (file:IN  in _ file)  return  STRING;
FUNCTION  (get _ line  RETURN  string;
PROCEDURE  put _ line(file:IN  out _ fle;  item:IN  string);
PROCEDURE  put _ line(item:IN  string);
      -- 整型输入输出的类属程序包
GENERIC
TYPE  num  IN  RANGE  <>;
WITH  FNCTION  image(x:num)RETURN  string  IS  nnm'Image;
WITH  FUNCTION  value(x:string)RETURN  num  IS  num'value;
  PACKAGE  integer _ io  IS
    PROCEDURE  get(flle:IN  in _ file;  item:OUT  num);
    PROCEDURE  get(item:OUT  num);
    PROCEDURE  put(flle:IN  out _ file;
      item:IN  num;
      width:IN  integer := 0;
      base:IN  integer  RANGE  2..16 := 10);
    PROCEDURE  put(item:IN  num;
      width:IN  integer := 0;
      base:IN  integer  RANGE  2..16 := 10);
END  integer _ io;
      -- 浮点输入输出的类属程序包
GENERIC
TYPE  num  IS  DIGITS  <>;
WITH  FUNCTION  image(x:num)  RETURN  string  IS  num' Image;
WITH  FUNCTION  value(x:string)  RETURN  num  IS  num' value;
```

```
PACKAGE  float_io  IS
  PROCEDURE  get(file:IN  in_file;  item:OUT  num);
  PROCEDURE  get(item:OUT  num);
  PROCEDUER  put(file:IN  out_file;
    item;  IN  num;
    width:IN  integer := 0;
    mantissa  :IN  integer  := num'digits;
    exponent  :IN  integer  := 2);
  PROCEDRE  put(item  :IN  num;
    width  :IN  integer  := 0;
    mantissa  :IN  integer  := num'digits;
    exponent  :IN  integer  := 2);
END  float_io;
    --定点输入属出的类属程序包
GENERIC
TYPE  num  IS  delta  <>;
WITH  FUNCTION  image(x:num)  RETURN  string  IS  num'image;
WITH  FUNCTION  value(x:string)  RETURN  num  IS  num'value;
  PACKAGE  fixed_io  IS
    delta_image  :CONSTANT  string :=
    image(num'  delta_integer  (num'delta));
    default_decimals  :CONSTANT  integer  := delta_image'length-2;
PROCEDURE  get(file  :IN  in_file;  iten  :OUT  num);
PROCEDURE  get  (item  :OUT  num);
PROCEDURE  put(file  :IN  out_file;
    item  :IN  num;
    width  :IN  INTEGER  := 0;
    fract  :IN  integer  := deeault_decimals);
PROCEDURE  put(item  :IN  num;
    width  :IN  integer  := 0;
    fract  :IN  integer  := default_decimals);
END  fuxed_io;
    --布尔型输入属出
  PROCEDURE  get(file  :IN  in_file;  item  :OUT  boolean);
  PROCEDURE  get(item  :OUT  boolean);
  PROCEDURE  put(file  :IN  out_file;
    item  :IN  boolean;
    width  :IN  integer  := 0;
    lower_case  :IN  boolean  := false);
  PROCEDURE  put(item  :IN  boolean;
    width  :IN  integer  := 0;
    lower_case  :IN  boolean  := false);
    --枚举类型的类属程序包
GENERIC
```

```
TYPE enum IS (< >);
WITH FUNCTION image(x :enum) RETURN string IS enum'image;
WITH FUNCTION value(x :string) RETURN enum IS enum'value;
PACKAGE enumeration_io IS
  PROCEDURE get(file :IN in_file; item :OUT enum);
  PROCEDURE get(item :OUT enum);
  PROCEDURE put(file :IN out_file;
    item :IN out_file;
    item :IN integer := 0;
    lower_case :IN boolean := false);
  PROCEDURE put(item :IN enum;
    width :IN integer := 0;
    lower_case :IN boolean := false);
END enumeration_io;
    --布局控制
  FUNCTION line(file :IN in_file) RETURN natural;
  FUNCTION line(FILE :IN out_file) RETURN natural;
  FUNCTION line RETURN natural; --FOR DEFAULT OUTPUT FILE
  FUNCTION col(file :IN in_file) RETURN natural;
  FUNCTION col(file :IN in_file) RETURN natural;
  FUNCTION col RETURN natural; --FOR DEFAULT OUTPUT OUTPUT FILE
  PROCEDURE set_col(file :IN in_file; to :IN natural);
  PROCEDURE set_col(to :IN natural); --FOR DEFAULT OUTPUT FILE
  PROCEDURE new_lime(file :IN out_file; n :IN natural := 1);
  PROCEDURE ckip_line(n :IN in_file) return BOOLEAN;
  FUNCTION end_of_line RETURN boolean;
  PROCEDURE set_line_length(file :IN in_file; n :IN integer);
  PROCEDURE set_line_length(file :IN out_file; n :IN integer);
  PROCEDURE set_line_length (n :IN integer);
  FUNCTION line_length(file :IN in_file) RETURN integer;
  FUNCTION line_length(file :IN out_file) RETRN intgter;
  FUNCTION line_length RETURN integer;
    --隐含输入和输出处理 FUNCTION standard_input RETURN in_file;
  FUNCTION standard_output RETURN out_file;
  FUNCTION current_input RETURN in_file;
  FUNCTION current_output RETURM out_file;
  PROCEDURE set_input(file :IN in_file);
  PROCEDURE set_output(file :IN out_file);
    --异常
  name_error :EXCEPTION RENAMES character_io.
  name_error;
  use_error :EXCEPTION RENAMES character_io.
  use_error;
  status_error :EXCEPTION RENAMES character_io.
```



```
    data _ error;
    end _ error : EXCEPTION RANAMES   character _ io.
    end _ error;
    layout _ error : EXCEPTION;
end TEXT _ IO;
```

参 考 文 献

[1] 《Ada 程序设计》 J. G. P 巴恩斯著，张乃孝、侯世君等译，人民邮电出版社

[2] 《Understanding ADA With Abstract Types》 Ken Shunmate

[3] 《Ada》S. J. 杨著，田淑清、谭浩强等译，清华大学出版社

[4] 《Ada 导引》H. 莱德加德著，袁崇义、徐泽同译，科学出版社