高 等 学 校 教 材

P A S C A L 程序设计语言

（一九九九年七月修订）

郝立中　张成现　高晓娟

陈向荣　张淑珍　宋安军　编

主编：　郝立中

内容简介

本书全面地介绍了 PASCAL 语言的有关概念及规则，介绍了用 PASCAL 语言设计计算机程序的一般方法。

本书适应于没有学过其它计算机语言的读者，编写尽量深入浅出，兼顾了不同基础和能力的读者的需要。鉴于历史的经验，本书在介绍 PASCAL 语言的同时，特别注意培养科学的程序设计思想方法。

本书可以作为大专院校非计算机专业和计算机应用专业的教材，也可以供初学者作为自学的读物。同时，本书的内容尽量做到了一定程度的严格性和完整性，所以，在实际工作中还可作为备查的资料。

前　言

PASCAL 语言是第一个体现结构化程序设计概念的计算机语言，因此它在 60 年代末 70 年代初的出现是程序设计语言发展史上的一个里程碑。

N. Wirth 教授研制 PASCAL 时提出的目标中有两条：

一、提供一种可用的表示法，以便按照系统化程序设计方法的新见解，以结构化的、精确的方式表达程序设计的基本概念和结构。

二、适用于课堂教学的特点，使程序设计的某些基本概念和技能得到系统的训练。

这些目标的实现，成了 PASCAL 语言的鲜明特色和突出的优点。因为这些原因，几十年来 PASCAL 一直被公认为一种理想的教学语言。尽管后来新的语言又出现了许多种，但人们普遍认为，PASCAL 中包含着一般程序设计语言的基本概念。有了 PASCAL 的基础，必要时再学其它语言也是不难的。所以，多年来 PASCAL 语言一直是计算机专业的一门重要基础课。

近年来，随着计算机日新月异地普及，学习计算机语言的需要已不限于计算机专业，而是扩大到了几乎所有专业的学生。这样一来，对教材就有了新的要求。

首先，教材必须适应不同基础和能力的读者的需要。为达到这一点，本书并没有在基本概念和基本方法上减少内容，而是注意从另外的方面降低难度。我们知道，实际的程序设计工作难以避免其复杂和繁琐，但是教材中的例题和练习假如都使读者陷入繁琐的枝节之中疲于奔命的话，他们就难以集中精力理解掌握基本概念和要领。因此，本书的例题编写力求其精，避免节外生枝的东西。同时为了在不增加负担的条件下拓宽思路，本书对某些有实际意义的问题给出同一问题的不同做法，并进行对比分析。

其次，几十年的历史表明，比起掌握一种语言的语法规则来说，更加重要的是掌握科学的程序设计思想方法。然而计算机语言课的教材只能以主要篇幅讲述语言本身的规则，难以深入灌输科学的思想方法。因为这些思想的道理只有在掌握了语言并且接触过较大型的复杂程序后才容易讲清楚。这是一个矛盾。这个矛盾，对计算机专业的读者来说问题不大，因为他们在学了语言课之后，还可以在其它计算机课程中接受科学方法的训练。但非计算机专业的读者可能就没有这种机会了。如果在学习计算机语言课的过程中，自发地养成了一种不科学的思想方法和坏的程序设计习惯，那就不是我们的希望了。因此本书在这方面作了努力。我们觉得，除了应在课文中说明科学方法的意义以外，在这方面更重要的是，应特别注意在课文和例题里杜绝可能出现的反映不科学的方法和不良习惯的毛病，以免无意中给读者树立坏的样板。在科学的思想方法中，我们首推抽象的方法。我们认为，抽象的能力是从事程序设计工作所需要的最重要的一项能力。

本书在上述两个方面照顾了非计算机专业读者的需要。同时，本书对 PASCAL 的概念、规则和程序设计原理的讲述并没有降低标准，而是尽量精确和完整。其目的有二：

一是希望本书不仅适用于非计算机专业，同时也适应计算机专业的要求。

二是本书不仅在初学时可用作教材，在实际工作中还可作为备查的资料。

按照后一目的，内容要求一定程度的严格性和完整性。当然，有些细节在初学时是不

必注意的，在教学和阅读中可适当掌握，不作重点。

我们力争做到，不论基础及能力较高或较低的读者，学习了本书之后都能有所提高。

本书的编写工作由张成现负责组织。各章的初稿编写者是：第一、二章，郝立中；第三、四、五章，高晓娟；第六、七、九章，陈向荣；第八章，张淑珍；第十、十一章，宋安军。最后由郝立中对全书进行了修改。

本版在一九九七年十月初版基础上作了部分修订。由于水平的限制，缺点错误在所难免，欢迎读者批评指正。

郝立中　于西北纺织工学院

一九九九年七月

目　　录

第一章　程序设计常识介绍

1.1　引言

本章是在讲述 PASCAL 语言之前，先向读者介绍一些有关程序设计的必要的基本知识。

什么是程序设计？N. Wirth 曾经提出过一个著名的公式：

数据结构＋算法＝程序

换句话说，程序设计就是设计数据结构和设计算法。要利用计算机解决实际问题，必须分析该问题的数学模型，设计出解决问题所需要的数据结构和算法，将设计用计算机可以接受的形式描述出来并送入计算机，才能让计算机按我们的规定去求出所需要的结果。这种用计算机可以接受的形式描述出来的数据结构和算法就是程序。

有人对这个公式作了扩充，加进了如“程序设计方法”、“语言工具和环境”等等。这里“语言工具”就是指计算机语言，一种计算机语言提供一种计算机可以接受的描述形式。本书要讲述的 PASCAL 语言也就是一种“语言工具”。但是我们认为，能反映程序的本质的仍是 N. Wirth 原来的公式。因为其它东西仅仅是设计数据结构和设计算法所用的工具而已。打个不太严格的比喻，若将程序设计比作文学创作，那么像 PASCAL 这样的语言可以比作汉字和汉语的语法。虽然用中文搞文学创作不能不懂汉语，但是汉语和文学创作毕竟是两门学问，总不能说学了汉语就算学了文学创作。因此，从这个意义上说，如果仅仅学了有关计算机语言的知识，离全面掌握程序设计还是存在相当距离的。

但这只是问题的一个方面。而另一方面，程序设计是一门发展中的学科。程序设计这一项工作随着计算机的问世很早就出现了，但早期的很长一段时间中并不存在一门现代意义上的“程序设计”课程。符合 N. Wirth 的公式的专门讲述设计数据结构和设计算法的课程在国外是六十年代末才出现的。而在此之前所有名之为“程序设计”的课程无不以讲述具体的计算机指令系统或语言为主要内容，而关于数据结构和算法的设计技巧只是在它的例题中体现出一些。换句话说，当时的“程序设计”课程也就相当于现在的计算机语言课程。直到今天，许多讲述某种计算机语言的书籍仍在书名上冠以“程序设计”，也是沿用了这个习惯。

现在的计算机语言课程中，同样要在大量的例题中介绍解决实际问题的常用算法。这些例题涉及的内容当然不能与专门讲计算机算法的课程相比，但也足以使学生建立程序设计的某些基本思想。一个有独立思考能力的学生，有了这些思想，就已经可以解决许多实际的问题了。

特别是，本书的读者对象中还包括非计算机专业的学生。而非计算机专业的学生除了语言课以外，许多人没有机会另修专门的程序设计课程。所以，本书例题中介绍的各种算法对他们的意义更大一些。而且正因为如此，本书要有较多的篇幅介绍一般的程序设计思想。

本书第一章介绍的某些思想，是基于程序设计工作的复杂性而提出来的。对于没有接

触过复杂程序的初学者来说，这些思想不容易全部领会。可以先建立一个初步印象，然后，在经过一段学习和实践，有了一些经验后再回过头来进一步理解。

1.2 计算机语言

1.2.1 机器语言

计算机不能直接理解和执行人们使用的自然语言，而只能接受特别规定的指令。这些指令是由计算机的设计者规定的，一般是二进制的编码。这种指令称作机器指令。每一种类型的计算机都有一套机器指令系统。要让计算机解决某一特定的问题，需要选用指令系统中的指令组成程序。这样编写的程序称作机器语言程序。

机器指令随不同的机器而不同。而且机器指令是代表机器可以直接执行的操作，通常是些最基本的操作，实际应用中的许多不算复杂的操作也常常需要很多条机器指令的组合才能实现。再加上二进制代码不符合人的阅读习惯，所以编写机器语言程序是一项相当繁重的工作。

不仅如此，程序设计过程中还经常需要修改，机器语言程序的修改更是困难。这是因为，机器语言程序对存储器的访问是按地址进行的，每一条机器指令，只要它的操作要访问存储器，其指令的代码中就要有一部分是反映地址的编码。这样，在修改程序时，只要某个变量或子程序等安排的地址变动了，则整个程序中所有要引用这个地址的指令代码都得修改。在机器语言程序设计过程中，为增、删一条指令而不得不修改大量指令的事情是常常遇到的。

1.2.2 汇编语言

为了减轻机器语言程序设计过程中人的工作量，在计算机出现后不久，就有人开始致力于“程序设计自动化”的研究。这方面的第一阶段成果是引入了汇编语言。汇编语言，又称符号语言，是将二进制代码的机器指令换用一些容易被人记忆的记号（助记符）代表，同时它又允许使用标号（符号）代表地址或数据。

这样编写的程序文本还需要经过翻译，翻成机器语言程序才能由机器执行。这个翻译工作称作“汇编”。人们研制出了专门用来自动进行这一翻译工作的工具程序———“汇编程序”。汇编程序将人们用汇编语言编写的程序文本（称作源程序）当作输入数据读入，处理后输出一组代码，就是翻译成的机器语言程序（称作目标程序）。

汇编语言阅读起来比机器语言省力多了，而且用标号代表地址，不需要程序员去具体计算地址，这样程序地址移动后重新修改各指令的地址码的繁琐工作可由汇编程序去自动完成，节省了程序员的大量劳动。

但是，汇编语言的指令仍然是和机器语言指令一一对应的，所以从某种意义上说它和机器语言没有本质的区别。用汇编语言编写程序从某种意义上说就是用机器语言编写程序，只是换了个写法而已。汇编语言和机器语言一般统称为低级语言。

具有不同机器指令系统的计算机也需要有不同的汇编语言和不同的汇编程序。解决同样实际问题的算法，要在不同的计算机上运行也需要编写不同的汇编语言源程序。与机器语言程序一样，实际应用中的许多简单操作仍然常常需要很多条指令的组合才能实现，所

以低级语言编写程序的篇幅往往是很大的。

用低级语言编写程序，需要了解该计算机的机器指令系统，因而也需要了解该计算机的内部结构，而且还需要考虑怎样用该机器的指令组成我们实际应用中所要求的操作的细节算法。所以，用低级语言编写程序对程序员的要求是比较高的。

1.2.3 高级语言

程序设计自动化方面的第二阶段成果是引入了高级语言，又称算法语言。第一个成形的高级语言是在 50 年代中出现的 FORTRAN 语言。后来在又出现了 COBOL，ALGOL 60，BASIC 等语言。本书介绍的 PASCAL 语言也是一种高级语言。到现在为止的几十年间，出于不同的需要和不同的考虑，人们开发了许多种不同的高级语言，有人统计，仅在 60 年代提出的就超过了 200 种。但是其中只有少数得到了广泛的应用。

高级语言采用符合人们习惯的形式（如数学公式）来描述算法。 如 FORTRAN 就是 Formula Translator（公式翻译）的缩写。但鉴于实际问题中算法的复杂性，远非仅用普通的数学公式可以包括的，所以每一种高级语言中还分别规定了许多特有的表达形式。这些形式中使用某些英文单词和符合人们习惯的符号，其意义接近人的自然语言，便于人去阅读理解。

用高级语言编写的程序文本一般也需要经过翻译，翻成机器语言程序才能由机器执行。这个翻译工作称作“编译”。专门用来自动进行这一翻译工作的工具程序称作“编译程序”。相对于编译程序，我们将用高级语言编写的程序文本称作源程序，将翻译成的机器语言程序称作目标程序。

对于高级语言编写的程序的处理，除了上述的编译方式外，还有另一种处理方式，称作“解释”。这种方式是：由“解释程序”将人们用高级语言编写的程序文本读入处理，但并不输出目标程序代码，而是由“解释程序”直接执行高级语言程序中规定的操作。也就是说，这是边翻译边运行的处理方式。这样做，将翻译和运行两道手续合而为一，有时可以简化操作，但边翻译边运行，运行的速度显然比直接运行目标程序要慢得多。

某些高级语言如 BASIC 等多采用解释方式，而包括 PASCAL 在内的另一些高级语言多采用编译方式。

可以这样说，低级语言是面向机器的语言，而高级语言是面向过程的语言。程序中规定的操作具体由哪些机器指令的组合来实现，则由编译程序自动去决定，高级语言的程序员可以不必考虑。同一个高级语言程序要用在不同类型的计算机上，只要经不同的编译程序处理，就可以生成适用不同机型的目标程序，源程序基本上可以不必改动。现在每一种通用的计算机系统在出厂销售时，大都已配置了该型号计算机上使用的各种常用高级语言的编译程序，所以说，高级语言基本上可以达到与具体机器无关。

高级语言的出现大大地提高了程序设计的工作效率。但是与低级语言相比，高级语言又有两个缺点：一是灵活性差，难以利用具体机器特有的性能去实现一些非通用的功能；二是目标程序的效率低，这是因为由机器硬性地翻译所得到的目标程序，不可能达到人工直接编写的精炼程度。所以在某些要求高灵活性或高效率的软件（如快速过程控制程序，以及操作系统等系统软件）设计中，常常不得不使用低级语言编写程序。为了一定程度上减少这个矛盾，又出现了几种保留一定低级特性的高级语言，如 C 语言，FORTH 语言等。

1.2.4 “更高级”的语言

纵观上面由机器语言到高级语言的发展，实际上是人们逐步解决“做什么”和“如何做”矛盾的过程。人们希望做到：只要指出“做什么”而不必具体指出“如何做”，由计算机自动去解决“如何做”的问题。但是“做什么”和“如何做”是相对的，低层次上的“做什么”同时又成为高层次上的“如何做”。从低层次上看，计算机一出现，这个矛盾就在一定意义上解决了。因为我们可以编写一条机器指令而不必考虑这条指令该怎样完成，比如用加法指令让机器作加法而不必指出每一位该如何加。但是从高一些的层次上看这又远远不够了。因为高层的一个操作需要许多条机器指令才能完成，要列出每一条指令，还应该算是在描述“如何做”。高级语言的出现已在较高层次上解决了这个问题，但实际应用中问题的复杂程度是没有界限的，用高级语言编写程序的工作量仍然嫌大，所以人们又希望能在更高层次上解决“做什么”和“如何做”的矛盾。这就出现了若干“更高级”的语言，如所谓的“非过程化语言”、“人工智能语言”等等。这些语言目前尚无统一的概念和术语，而且往往只适用于特定的应用领域（如数据库管理），这里不详细介绍。

因为所谓“更高级”的语言目前尚无统一的概念，所以现在一般仍然认为计算机语言包括三类：机器语言、汇编语言、高级语言。

1.3 算法的描述和程序结构

前面已介绍了计算机语言，计算机语言就是一种可用来描述算法的形式。但是因为它的主要目的是供计算机处理，显得“严格有余，形象不足”，供人阅读理解时不是最理想的形式。特别是它不易表达抽象的粗略的算法思路，如初步的“概要设计”中的算法等。所以人们又采用了若干种便于人理解的形式来描述算法。

1.3.1 自然语言

简单的算法可以直接用自然语言描述。例如根据系数求解一元二次方程的算法可以如下描述：

(1)输入系数 a，b，c。

(2)计算 b^2-4ac 赋值给 d。

(3)如果 d<0 则转去执行(8),否则继续。

(4)计算$(-b+\sqrt{d})/(2a)$赋值给 x_1。

(5)计算$(-b-\sqrt{d})/(2a)$赋值给 x_2。

(6)输出 x_1，x_2。

(7)结束。

(8)计算(-b)/(2a)赋值给 r。

(9)计算$\sqrt{|d|}/(2a)$赋值给 p。

(10)输出 r+pi，r-pi。

(11)结束。

自然语言灵活通俗，但它有以下缺点：

一、比较冗长，且有时不如用符号表示形象简洁；

二、不够严格，容易出现歧义性；

三、不容易说清楚包含分支、循环等的复杂操作流程。

1.3.2 流程图

另一种常用的描述形式是流程图，又称框图。它是用图形来表示算法的流程。

流程图中有多种图形符号，我们这里不作全面介绍，只介绍最常用的部分。一般流程图是若干个框用带箭头的线连接起来，每个框代表一个操作或判断，而连线则代表流程顺序。

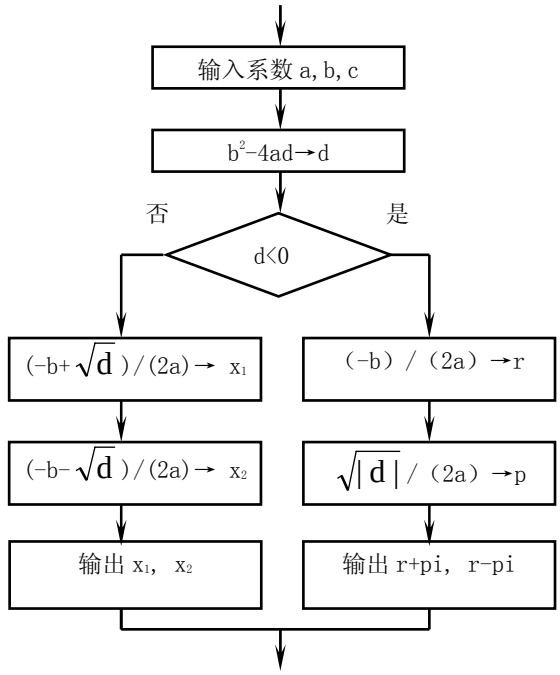


图 1。1　求解一元二次方程算法的流程图

常用的框有两种：一种是操作框，又称作处理框，画作矩形，框内写上要执行的操作。操作框有一个入口和一个出口。另一种是判断框，画作菱形，框内写上要判断的条件。判断框有一个入口和两个出口，出口处要标上哪个分支是条件成立时的去向，哪个分支是条件不成立时的去向。

上面举的根据系数求解一元二次方程的算法若用流程图描述可以如图 1.1。

1.3.3 基本结构

从上面的例子可以看到，实际的算法常常不是简单的步骤罗列，可能出现条件判断的分支。如果任意使用分支的办法来设计算法，可能使程序的结构变得非常复杂。

现在人们认识到（见下面将介绍的结构化程序设计的思想）复杂化是一种有害的倾向。为了限制程序的复杂性，有人证明了：只要三种控制结构就能等效表达用一个入口和一个出口的流程图所能表达的任何算法。如果我们只用这三种结构而避免任意的分支，可以在一定程度上减少程序的复杂和混乱。这三种控制结构一般称作基本结构，是：顺序结构、选择结构、循环结构（又称重复结构）。每一种基本结构本身都是只有一个入口和一个出口。

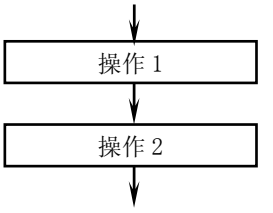


图 1。2　顺序结构

顺序结构的流程如图 1.2 所示。

选择结构的流程如图 1.3 所示。

选择结构还有一种变形如图 1.4 所示。

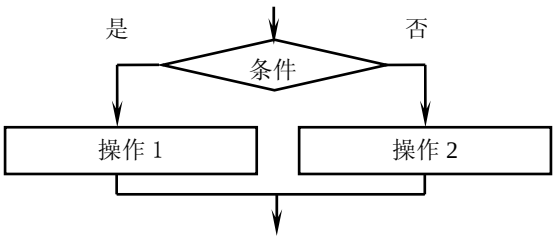


图 1.3　选择结构

图中“条件”一框出口处的“是”和“否”若反过来，并不改变结构的类型。显然，“是”和“否”若反过来，只要把条件换成其否定条件，则功能就和原来相同了。这里图中画的“是”和“否”的方向是按 PASCAL 中相应功能的构造型语句中的规定。下面各图也同样。

图 1.3 中的两个操作只要有一个是空操作就变成了图 1.4,所以图 1.4 可以看作图 1.3 的特例。但是图 1.4 的机器编码可以比图 1.3 更简单,所以大多数语言都对图 1.4 提供了单独的表达格式。

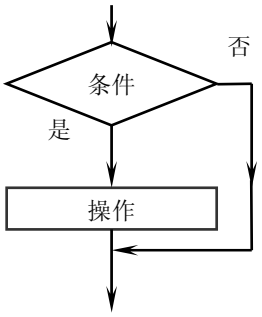


图 1。4　选择结构的另一种

循环结构的流程有几种，其中最基本的一种称作当型（WHILE 型）循环，其流程见图 1.5。

循环结构的另一种常用的流程称作直到型（UNTIL 型）循环，见图 1.6。

上述循环结构中的“操作”一框一般称作“循环体”。这两种循环相比，当型循环中假如一开始条件就不成立，则循环体就一次也不执行；而直到型循环中无论如何循环体至少执行一次。一般认为当型循环的程序逻辑较简单些，但是直到型循环的机器编码较简单。

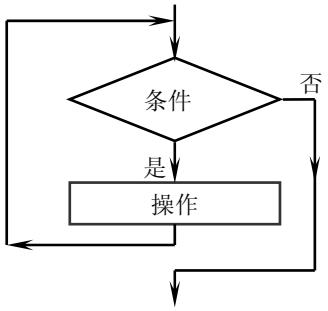


图 1.5　当型(WHILE 型)循环

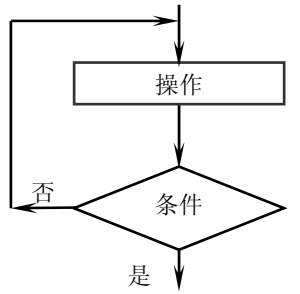


图 1.6　直到型(UNTIL 型)循环

上述各基本结构中只包括一个或两个操作框，假如一个框只能代表一条简单操作，显然一个结构是不足以表达一个实际的程序的。为了表达真实的复杂程序，可采用嵌套的办法。就是说一个操作框代表的可以不是一条简单操作，而是又一个完整的基本结构。我们知道，每一种基本结构本身都是只有一个入口和一个出口，所以用它取代更大结构里的一个操作框是够条件的。嵌套的层次没有限制。例如单是顺序结构的多重嵌套就可以表达任意长的操作序列。各种结构的嵌套可以构成无限复杂的程序。

如果一个循环结构的循环体本身或其更内层嵌套的操作又是一个循环结构，则一般习惯将其称作多重循环。

循环程序执行中假如一直不满足结束循环的条件（即上述当型循环图中条件一直为“是”，或直到型循环图中条件一直为“否”），程序就会无限次地循环下去，永远达不到

出口。我们称之为“陷入了死循环”。一般科学计算等应用程序中，出现死循环应算是错误。为避免死循环，循环体中的操作应能使条件在有限步中发生变化，使程序能够结束。

实际应用中常用到一种控制循环的方式，是：利用一个变量充当计数器，循环开始前先使该变量等于某个初值，循环过程中每执行一次循环体就使该变量改变一个固定的增量，以该变量的值超过某个终值作为循环结束的条件。这样构成的循环称作计数循环。计数循环是一种简单的不会陷入死循环的循环控制方式。计数循环可以按当型来构造，也可按直到型来构造。例如，选变量 i 充当控制变量（即上述计数器），初值为 0，终值为 100，增量为 1，用当型来构造计数循环，则流程如图 1.7。

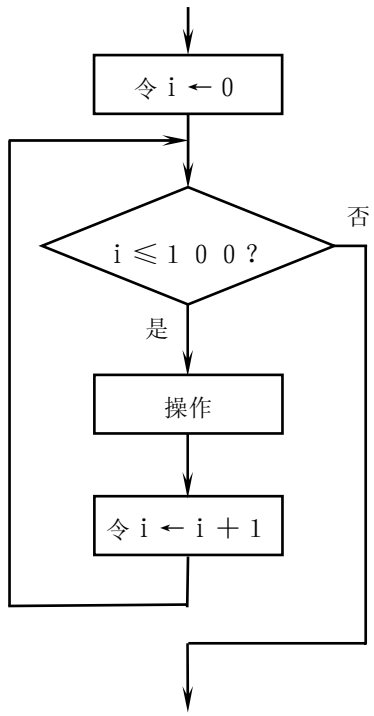


图 1.7　计数循环一例

计数循环虽然可以算作普通循环（当型或直到型）的特例，但因其常用，所以大多数高级语言都对它提供了单独的表达格式，如 PASCAL 中的 FOR 语句。 在表达计数循环的格式中，习惯上将上述“操作”一框中的操作（不包括循环变量增量）称作循环体。

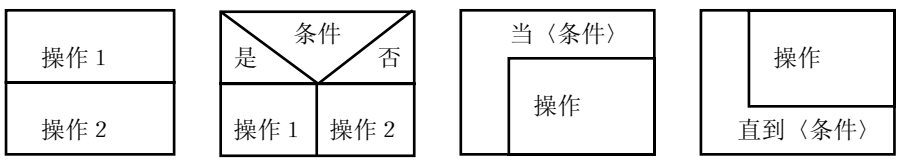
1.3.4 结构框图

前面介绍的普通流程图的主要优点是很直观，便于初学者掌握。从四十年代末到七十年代中期，流程图一直是软件设计的主要工具。直到现在，尽管不少学派反对流程图，它仍然被广泛使用着。但是流程图有若干个重要缺点：

流程图往往掩盖问题的结构，而诱使程序员去考虑程序的流程。流程图不是逐步求精的形式，不易表达算法的层次，容易使设计者在全局结构尚未筹划清楚时就去直接写出大量细节，不符合抽象方法的要求。而且，它对流程线不加限制，有些程序员主观上想按基本结构设计算法，但因对概念理解不够，可能不知不觉画出违反基本结构的流程图。

既然人们提倡只用基本结构来设计算法，就希望能有一种直接表示基本结构的图形。用这种图形来描述算法，可以迫使设计者不采用非结构化的流程。

七十年代中有人提出了一种新的框图，我们称之为结构框图（N-S 图，Chapin 图）。这



(1)顺序结构　(2)选择结构　(3)当型循环　(4)直到型循环

图 1.8　用结构框图表示的基本结构

种图省去了流程线，一个结构画成一个矩形框，结构内的操作框是包含在大框内的小矩形框。结构框图中基本结构的画法见图 1.8。

只要将内部的操作框代之以一个完整的结构框图，就可以表达结构的嵌套。

前面举的根据系数求解一元二次方程的算法若用结构框图描述可以如图 1.9。

对于计数循环，也可以画成类似当型循环的形式，只是在写条件的位置按照 FOR 语句中的习惯写法标出计数循环的各个要素。详见第五章的例子。

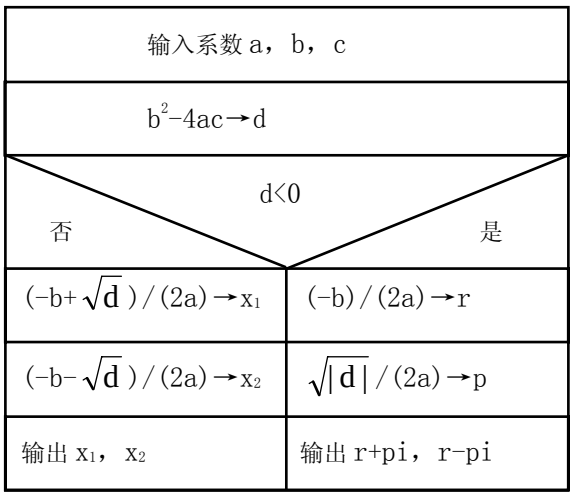


图 1.9　求解一元二次方程算法的结构框图

结构框图改进了普通流程图的缺点，但同时又在一定程度上减低了直观形象性。因此它在国外使用并不普遍。国外较多使用下面介绍的伪代码，而国内由于语言的原因伪代码使用较少，相对来说使用框图多一些。

1.3.5 伪代码

我们知道，大多数高级语言中都提供了各种基本结构的表示形式。如果不太要求流程的直观形象性的话，完全可以直接采用高级语言中规定的记号来描述算法。鉴于高级语言不易表达抽象的粗略的算法思路，可以将它与自然语言和习惯符号结合起来。这种形式就是伪代码。当然，基于不同的高级语言，可以有多种伪代码，也有人使用独立于各种高级语言的“伪代码语言”，目前并不统一。

前面举的根据系数求解一元二次方程的算法若用伪代码描述可以如下：

```
输入系数 a，b，c ；
b^2-4ac→d;
IF  d<0
  THEN
    BEGIN
      (-b)/(2a)→r;
      √|d|/(2a)→p;
      输出 r+pi，r-pi
    END
  ELSE
    BEGIN
      (-b+√d)/(2a)→x1;
      (-b-√d)/(2a)→x2;
      输出 x1，x2
    END
```

这种写法采用了 PASCAL 中的记号，比较容易转换成真正的 PASCAL 程序。

伪代码主要是给人看，并不要求十分严格，所以也可以按需要加以改变。如可将上述表示中的记号换成汉语：IF 换成“如果”，THEN 换成“则”，ELSE 换成“否则”，BEGIN 及 END 换成语句括号，如下

输入系数 a，b，c ；

b^2-4ac→d;

如果　d<0

　则

　　【 (-b)/(2a)→r; $\sqrt{|d|}$/(2a)→p; 输出 r+pi，r-pi 】

　否则

　　【 (-b+$\sqrt{d}$)/(2a)→x_1;

　　(-b-$\sqrt{d}$)/(2a)→x_2;

　　输出 x_1，x_2

　　】

这样就和自然语言几乎一样通俗易懂，同时又比自然语言更加条理化和严格化。

上面介绍的几种描述算法的形式各有优缺点。本书中在说明内部动作的原理时，多使用流程图，这是因为流程图相对来说还是最形象直观的一种描述形式。但在描述具体应用程序的算法时，因本书中的例题都属于简单程序，描述形式的重要性不太突出。所以，本书中，上述几种表达形式都有用到的地方，主要目的是为使读者有一个感性的认识，并不要求对每一种都掌握得很熟练。实际需要用到时，读者可以选择自己认为最适宜的形式。

1.4　程序开发的步骤

一个程序，从产生开发的要求，直到最后软件停止使用为止的整个生存期中，程序设计人员承担的工作一般可以分作五个阶段：系统分析和规范制定阶段、设计阶段、编码阶段、调试阶段和运行维护阶段。

第一个阶段即**系统分析和规范制定阶段**。这一阶段的工作是确定要开发的程序需要实现哪些功能，需要达到什么样的性能。程序的开发要求是根据实际的需求提出的，但提出时通常较模糊，需要由程序设计人员再对其进行严格描述。一般情况下，仅仅根据用户提出的技术要求是不能开始设计的，而要由程序设计部门经过系统分析和规范制定阶段的工作后提出一个软件需求说明书，经用户方面认可后作为整个设计的基础。设计阶段将严格按照这里制定的规范进行。

第二个阶段，**设计阶段**，即设计出一定的数据结构和算法来实现上面提出的要求。但是这一阶段的设计结果并不写成具体的计算机语言程序，而是写成便于人来阅读的形式，如上一节介绍的各种描述算法的形式。设计阶段通常可以进一步分为**概要设计**和**详细设计**两步。

第三个阶段是**编码阶段**，即根据设计阶段的结果写出具体的计算机语言程序。这一步实际上是对设计阶段的结果进一步细化。

第四，**调试阶段**。所谓调试，就是用试验的方法发现程序中的错误并修改。对于复杂的程序，仅靠调试来保证其绝无错误是困难的，但至少要达可以正常使用和用户可以接受的水平。

第五，**运行维护阶段**。这一阶段的工作除继续修正软件使用中发现的错误以外，还需要满足因各种原因（如因原来制定规范时缺乏经验，或因客观条件变化等）而对设计规范作的变更。

几十年来的实践表明，这五个阶段中，工作量和资金消耗最大的是第四和第五阶段。而第四和第五阶段工作量的大小，在很大程度上取决于第一和第二两个阶段设计的好坏。所以相对来说第三阶段的编码不算是最重要的。

但是，不编出具体的程序清单，第四、第五阶段的工作也无从着手。而且，对于某些特别简短的程序，第一、二、四、五阶段的工作有可能完全省去，而第三阶段却不可少。所以在有些时候人们又常把第三阶段看作整个工作的中心，特别是在早期，常常一说“程序设计”就是专指编码。这反映出早期人们对程序设计的片面认识。现在这些片面认识正在逐渐扭转。

编码牵涉到具体的计算机语言， 这就需要了解有关语法和各种规定。 本书就是介绍 PASCAL 语言的语法及各种规定。而且本书的例题都是简短程序，所以不可能反映上述五个阶段的全部工作，而只能反映第三阶段的编码。只是在某些稍复杂的例题中描述了算法的求精过程（大致上属于第二阶段的工作），希望能在一定程度上帮助读者体会程序设计的方法。

1.5 结构化程序设计方法简介

早期，人们对程序设计工作的复杂性没有充分的估计，随着计算机的应用不断向深度和广度的方向发展，矛盾就暴露出来了。大约从六十年代中开始，出现了称作“软件危机”的现象。软件价格急剧上升，使之成为许多以计算机为基础的系统中花钱最多的项目。虽然制定了进度表和完成日期，但很少按期实现。特别是花费在软件调试和维护上的资源和工作量急剧膨胀。软件质量难以保证，程序中的错误似乎永远改不完，甚至常常是越改越多。软件设计成了计算机应用的“瓶颈”。

由 E.W.Dijkstra 和 C.A.R.Hoare 等人提出的结构化程序设计的思想就是鉴于软件危机的教训而形成的。“结构化程序设计”一语目前尚无统一的定义，按我们的理解，它的思想大致包括以下几点：

其一：“好”程序的标准，除了正确性外，最重要的一条是要条理清晰，结构简明。包括软件文档的可读性，易修改性在内。程序的运行效率等还在其次。

从原则上说毫无异议，正确性当然是最重要的。但是一个程序编出来，怎么知道它是否正确呢？单靠盲目的试验来“证明”程序没有错误是不可靠的，Dijkstra 曾经论证过：程序调试只能指出错误的存在，而不能说明没有错误。要想尽量全面地检验其正确性，不得不深入考虑程序的内部结构。假如程序没有清晰的条理，这一工作的难度是可想而知的。

而且，程序的调试和维护期间经常需要修改。即使是原来正确的程序，如果条理混乱

的话，可以说每修改一次都难免产生一批新的错误。

总而言之，有了条理性才会有“可验证性”和“可修改性”。因此，必须充分认识清晰的条理和简明的结构对于软件的重要意义。

其二：程序设计的思想方法：

要想编出高度条理性的程序需要采用科学的思想方法。

(1) 枚举的方法。问题可分为有限种情况时，或某一动作可分为有限个步骤时，应逐一考虑，不应遗漏。特别应注意不要只考虑到常见的情况而忽略了罕见的特殊情况。

(2) 递推的方法，即“数学归纳法”。程序中有循环、递归一类大量重复的操作时不应只考虑有限步的正确性，而应采用数学归纳法的思路，从逻辑上保证其任意步都正确。

(3) 抽象的方法。“抽象”有多种体现，其中最突出的是自顶向下逐步求精的分析方法。上层描述中的一个名词是下层一系列操作的概括抽象，而下层一系列操作是上层中抽象概念的具体化。经过对问题的多层分解得出具体的程序。

其三：具体程序编码中，反对滥用转向语句（GOTO 语句），主张尽量只使用少数几种基本结构来组成程序。推荐的基本结构有：

(1) 顺序结构；

(2) 选择结构；

(3) 循环结构（又称重复结构）。

结构可以嵌套，即结构内的一个“操作”本身又可以是一结构。整个算法可由基本结构嵌套而成。

有人将子程序（PASCAL 中的过程与函数）也算作一种基本结构。实际上，若将子程序的每次调用看作是一个广义指令的话，那么子程序可算是抽象方法的一种实现形式。

其四：编程技巧力求规则化，反对滥用“小聪明”，为表现个人的特性而搞乱大局的规则，或增加其复杂程度。

关于最后这一点，除应重视复杂化带来的危害以外，还可指出这样的事实：许多人采用“小聪明”，“小技巧”，其动机大多是为了提高程序运行效率。但是影响效率的因素是多方面的，很难计算全面，复杂化后的程序更难全面考虑。人们往往片面地看到了某一影响效率的因素就轻易地采用复杂化的设计去“优化”它，却忽视了同时存在的其它因素。若真的全面计算一下，实际效果常常并没优化，反而更加劣化了。这种事例现实中可以找到许多。要能正确判断优化的效果，程序首先应有清晰的条理。Dijkstra 曾指出：“只有在程序具有可修改性时，我们才能优化它。”我们应充分认识这一点，从一开始就养成良好习惯。

上述各点中，第一、二、四点似乎都是“软指标”，只有第三点较具体，所以也有人把第三点看作“结构化”的唯一标准。这种说法有其片面性。不过，凡是不符合基本结构的转向语句都称作“非结构化跳转”，已成了一种习惯的叫法。

要做到上面的要求，对于程序员来说，先决条件是培养科学的严谨的思维方法和工作习惯。这对于从事任何工作都是需要的，但从事程序设计时这些就更显出其重要。这里特别强调一下抽象的思想方法，不善于抽象也就不善于条理化。举一个例子：某人批改了三个班的考试卷子，每个班 30 人，编号 1～30，卷子上有 10 个题目。 下面是对他批改的次

序所作的两种描述：

描述一：

先批改第二班，再批改第一班，最后批改第三班。

每班中先批改 30 号学生的卷子，再批改 29 号学生的卷子，……，最后批改 1 号学生的卷子。

每张卷子先批改第一题，再批改第二题，……，最后批改第十题。

描述二：

先批改第二班 30 号学生的卷子的第一题，再批改这张卷子的第二题，……，再批改这张卷子的第十题，接着批改这班 29 号学生的卷子的第一题，……，第十题，……，直到这班 1 号学生的卷子的第十题。

接着批改第一班 30 号学生的卷子的第一题，第二题，……，第十题，29 号学生的卷子的第一题，……，第十题，……，直到第二班 1 号学生的卷子的第十题。

再接下去批改第三班 30 号学生的卷子的第一题，第二题，……，第十题，等等，……，最后是第三班 1 号学生的卷子的第十题。

这两个描述说的都是同一个意思。比较这两个描述就会发现：第一个描述虽然粗，但很清楚；第二个描述虽然细，但很乱。而且，第一个描述中的省略号代表的意思很容易理解，第二个描述中省略号代表的意思就比较复杂，弄不好会被人理解错。

关键就是第一个描述采用了抽象法，第一句中用“第×班”一语抽象概括了下两句中将叙述的一大堆工作，第二句中又用“××学生的卷子”一语抽象概括了下一句中将叙述的一堆工作。第二种描述不善于抽象，每一句都具体化到“题”，反而费力不讨好。

这个道理虽然浅显，但掌握不好者大有人在。在实践中我们见到许多失败的程序设计，大都是不善于用抽象的思想方法去整理程序中的操作功能而造成的。因此希望读者给以充分的重视。

1.6　语法的形式化描述

1.6.1 语法和语义

每一种计算机语言都有自己的语法规则和语义规定。语法是指怎么样的符号序列才算是合法的程序；而语义则是指合法的符号序列表示什么意思，代表什么样的操作或运算。

计算机和人不一样，它不会猜测一段“语无伦次”的不合法语句是什么意思，也不会揣测没说清楚的“言外之意”。所以计算机语言的语法和语义的规定需要非常严格。虽然设计高级语言时考虑了照顾人的习惯，但人的自然语言的习惯中常有些模棱两可的地方，此时计算机语言就不能和人的习惯完全一致。

例如，人们描述某算法时如果说“令 y=1”，不难明白它的意思，不管 y 原来值是几，都将其改为 1。但是如果说“不令 y=1”，就有歧义了，是不改变 y 的原有值呢，还是要使 y 一定不等于 1 呢？计算机语言当然不能模棱两可。在计算机语言中， 对一个变量不作赋值操作，它就一定保持原有值。所以，PASCAL 语句

```
IF x=0 THEN y:=1
```

意思是“若 x=0 则令 y=1”，若 x 不等于 0 呢？不作“令 y=1”的操作，y 自然就保留原值。常有初学者误以为 x 不等于 0 时作了这句后 y 一定不等于 1，这是将自然语言中不严格的习惯带到计算机语言中来了。

计算机语言中不仅各种操作有严格的规定，而且在自然语言中常不被人重视的成分如标点符号等也都有严格规定。

另外，计算机语言需要由机器去自动处理（运行、编译或解释），所以它应遵循较简明的规则。对机器“简明”者并不一定对人也总是“简明”的。

例如，前面提到过的算法结构的嵌套，不管嵌套多少层，只要每层的规则都同样简明，机器就不难处理。而人的自然语言中虽然也允许复合语句的嵌套，但嵌套层次一多人就理解不清了。像

我不知道他知道不知道我知道不知道他知道不知道这件事。

这种句子很少有人能听明白。

因此，计算机语言的语法和语义不可能和自然语言一样。在我们的教材中是通过自然语言去解释计算机语言的，但鉴于自然语言和机器语言的上述差别，只用简单的说明是不够的。必须多方面补充解释（包括举例），弥补不严格之处，并提醒读者注意容易忽视的地方。应该说，本书的全部篇幅中，除了讲解一般程序设计思想以外，主要篇幅都是在讲解 PASCAL 语言的语法和语义。

由于自然语言的缺点，人们已经致力于设计严谨且易读的描述计算机语言的形式化方法。对于语义目前尚无便于推广的描述形式，但对于语法，已经有了若干种形式化的描述办法。在 PASCAL 语言的书籍资料中，常用的形式化描述有语法图和 Backus-Naur 范式。本书除用自然语言讲解外，对某些规则同时给出它的语法图，并在书末的附录 C 中附上 PASCAL 语言的全部 Backus-Naur 范式，供读者查阅。

但是这两种形式只能表示各语法成分组合为程序时在组合次序上的规则。而实际的计算机语言中除了组合次序上的规则外还有许多其它规则，如 PASCAL 中有类型的相容和赋值相容，标识符的定义和引用等。这些其它规则只能另行用自然语言补充说明。有时候人们只把组合次序上的规则称作“语法”，这是狭义的语法，而包括其它规则在内则是广义的语法。语法图和 Backus-Naur 范式只能描述狭义的语法，实际的程序当然应该遵守广义的语法。专业书籍中上述狭义的语法通常称作“文法”（grammar）。

1.6.2 语法图

语法图是如下面各图所示的框图。标在左端的名词是需要定义的语法成分。从图中左边的入口开始，沿着连线按箭头的方向直到右边的出口，将经过的各框中的语法成分按次序组合起来，就是左端所标语法成分的一种合法构成。有多少种可能的通路，就有多少种合法的构成。该图就是左端所标语法成分的定义。

例如，整个 PASCAL 程序的组成是“程序首部”后跟一分号，再后面是“分程序”，再后面是一句号（同小数点）。则“程序”的定义可用图 1.10 表示。

语法图中有两种框。一种是圆框(内容太多不好写时也可以画成椭圆

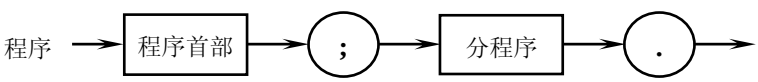


图 1.10　程序的语法图

框)，其中是“终
结符号”，即不需要进一步定义，可直接照写的符号。如图 1.10 中的分号和句号。另一种是矩形框，其中是“非终结符号”，即需要进一步在另一个语法图中对其定义的语法成分。例如图 1.10 中的非终结符号“程序首部”需要另行定义如图 1.11。

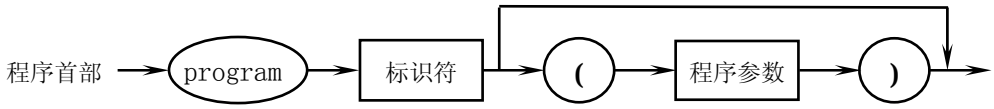


图 1.11　程序首部的语法图

图 1.11 中从左到右有两条通路，故可知道，程序首部可有两种构成，一种带有程序参数（两端加括号），另一种不带程序参数。图中的标识符是程序员给这个程序起的名字。“标识符”的定义见图 1.12。

图 1.12 中存在回路，所以它可以有无穷多种不同的通路。它表示标识符可以是任意个字母和数字组成的字符序列，但其中第一个必须是字母而不是数字。

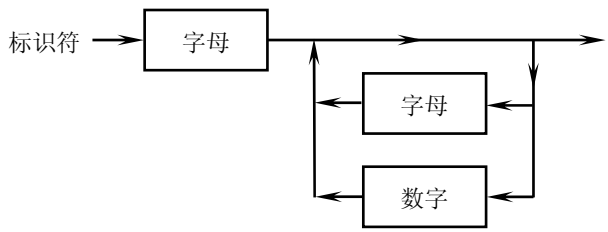


图 1.12　标识符的语法图

将语法图中的每个非终结符号都用另一个语法图定义出来，直到最后都归结为终结符号为止，“程序”的定义就描述完了。

1.6.3 Backus-Naur 范式

Backus-Naur（巴科斯—瑙尔）范式简称 BNF，是 1963 年发表 ALGOL 60 报告时使用的一种描述程序语言语法的形式化方法。因为它的提出应当归功于 Backus 和 Naur 两个人，所以人们将其称作 Backus-Naur 范式。

多年来人们在使用BNF的时候又对它作了若干局部改动，所以现在各资料中流行的BNF写法并不统一。本书采用国家标准《GB7591-87 程序设计语言 PASCAL》中的用法。

每一条 BNF 式中用等号＝表示“定义为”。 等号左边写上用汉字表示的要定义的非终结符号，右边写它的定义，最右边用句号表示该式结束。

在定义中用竖线 | 表示“或者”。如果竖线用在括号外，表示整个定义为有限几种形式。如果竖线用在括号内，表示括起的部分为有限几种形式。下面介绍的两种括号有特殊意义，如果不适用下面两种括号，而又需要表示中间某部分有两种构成形式的话，可以采用圆括号（　）。

定义中方括号［　］括起部分表示这一段可以有也可以没有。

定义中花括号｛　｝括起部分表示这一段可以有，也可以没有，也可以多次出现。

定义中用汉字表示非终结符号。

上述规定中用了几种符号，这些符号并不是真正的 PASCAL 语言中的用法。为了和真正的 PASCAL 中的字符区别，规定在定义中出现的终结符号一律用双引号括起来。

例如上文用语法图描述的几个语法定义改用 BNF 描述可以如下：

程序＝ 程序首部 “;” 分程序 “.” 。

程序首部＝“program” 标识符 [“(” 程序参数 “)”]。

标识符＝ 字母｛字母｜数字｝。

有时候我们需要描述部分定义，即表示左边的非终结符号的定义除右边描述的形式外还可以有别的构成形式，可以将上述等号＝改成大于号＞。如

赋值语句＞ 变量 “:=” 表达式 。

表示赋值语句可以按此格式构成，但有时还有另外的形式。以后我们会知道，赋值语句的另外一种形式是

赋值语句＞ 函数标识符 “:=” 表达式 。

这个形式仅在函数说明中出现。

赋值语句除这两种形式外再没有其它形式了，所以将这两条 BNF 合起来可以得到关于赋值语句的总的 BNF：

赋值语句＝（变量｜函数标识符）“:=” 表达式 。

习题

1.1 简述机器语言、汇编语言、高级语言的概念及其特点。

1.2 已经有了高级语言，为什么低级语言还不能淘汰？

1.3 画出下列算法的流程图：

本算法是用对分法求方程 sinx=0.8x 在（0.5,3）范围内的根的近似值（±0.00005），已知此根存在且唯一,且知道 0.5 及 3 均不是根。

对分法算法是:

设 f(x)=sinx-0.8x ，根所在区间下界为 A ，上界为 B ：

(1)若 f((A+B)/2)=0 或者 B 与 A 接近（B-A<0.0001），则以 (A+B)/2 为根结束计算；否则继续(2)。

(2)若 f((A+B)/2) 与 f(A) 同号，则以 (A+B)/2 代替原 A 作为区间下界；否则，以 (A+B)/2 代替原 B 作为区间上界。

(3)转(1)继续。

1.4 分析上题的流程图是由哪些基本结构嵌套组成的（假如上题所画流程图不符合基本结构的话，请先修改成符合基本结构的流程）。

1.5 为什么说将程序设计工作理解为仅仅是编写具体的计算机语言程序是一种片面的认识？

第二章　PASCAL 语言的基本常识

2.1　PASCAL 语言的由来及其特点

第一批高级语言是在 20 世纪 50 年代中到 60 年代初期出现的。 其中最主要的三个语言 FORTRAN，COBOL 和 ALGOL 60 中，FORTRAN 至今还在广泛使用，COBOL 也还在一定范围内使用。然而，这并不是因为它们比 ALGOL 60 好，而是另有原因。FORTRAN 很快被用户接受， 很大程度上是由于世界上最大的计算机公司 IBM 公司的支持。后来人们对 FORTRAN 的迷信变得非常顽固，以至想使他们接受新的更好的语言变成了一件无比困难的事情。COBOL 的流行则是由于美国国防部和工业界的合作所形成的强大力量的支持。

但是实际上要讲科学性，学术界公认 ALGOL 60 远比它们强得多。现代程序设计语言的许多重要特征是在 ALGOL 60 中第一次出现的。60 年代正是结构化的思想和软件工程的思想开始形成的时代，而语言的科学性对于程序本身的科学性来说，是一个很重要的条件。所以，后来开发的许多种新高级语言都采用与 ALGOL 60 相近的基本思想，这些语言被称作“类 ALGOL 语言”。现在的 PASCAL 实际上也可以算是一种“类 ALGOL 语言”。

随着计算机应用领域的扩大，人们看到 ALGOL 60 中缺少许多有用的特征和功能。经过一些人的工作提出了一个改进的版本，称作 ALGOL 68。但是在 ALGOL 68 的开发委员会中出现了一个持异议的反对派，认为这个版本违反了简明的原则，后来的情况证明这个反对意见是正确的。于是 ALGOL 68 进一步修改后最终于 1975 年才发表。

ALGOL 68 委员会中持异议的反对派中有一个是瑞士苏黎世联邦工业大学的 Niklaus Wirth，他后来开发了自己的新语言 PASCAL。

PASCAL 语言是由 N. Wirth 于 1968 年设计完成，于 1971 年正式发表的。这个语言的命名是为了纪念曾发明机械公式计算器的法国数学家 Blaise Pascal。后来，国际标准化组织 ISO 在作了一些修改后将它作为标准。我国制定的国家标准《GB7591-87 程序设计语言 Pascal》与 ISO 标准一致。本书基本上按此标准介绍。某些实际的系统可能与标准有部分差别。

PASCAL 语言是在 ALGOL W 的基础上发展起来的。它继承了 ALGOL 语言的科学严谨性，同时又进行了扩充，使之不仅适用于科学计算，也可在更广泛的领域中作为描述算法的工具。

PASCAL 语言是世界上第一个结构化程序设计语言。PASCAL 的发明人 N. Wirth 本人就是结构化程序设计思想的倡导者之一。PASCAL 中虽然保留了 GOTO 语句以备特殊需要时使用，但它提供的一系列构造型语句如 IF 语句，CASE 语句，WHILE 语句，REPEAT 语句，FOR 语句等，使得程序员更容易采用基本结构编程。它的过程和函数又为进行自顶向下，逐步求精的设计提供了方便。另外，PASCAL 中提供了丰富的数据类型和构造数据结构的手段。由于它为实现结构化程序设计方法提供了便利的条件，有利于培养学生良好的程序设计风格，所以它已被公认为一种理想的教学用语言。

2.2 程序的基本组成

第一章 1.6 节已经提到过：PASCAL 程序是“程序首部”后跟一个分号，再加“分程序”，再加一个句号构成的，语法图见前面的图 1.10。我们先来看一个简单的例子。前面提到过的根据系数求解一元二次方程的算法若写成 PASCAL 程序如下：

```
PROGRAM sample(input,output);
  { 根据系数求解一元二次方程, 要求 a≠0 }
  VAR
    a,b,c,d,r,p,x1,x2:real;
BEGIN
  read(a,b,c);              { 输入系数 a，b，c }
  d:=b*b-4*a*c;             { b²-4ac→d }
  IF  d<0
    THEN
      BEGIN
        r:=-b/2/a;                { 实部(-b)/(2a)→r }
        p:=sqrt(abs(d))/2/a;      { 虚部√|d|/(2a)→p }
        writeln(r,'+',p,'i');     { 输出二虚根 }
        writeln(r,'-',p,'i')
      END
    ELSE
      BEGIN
        x1:=(-b+sqrt(d))/2/a;     { (-b+√d)/(2a)→x1}
        x2:=(-b-sqrt(d))/2/a;     { (-b-√d)/(2a)→x2}
        write(x1,x2)              { 输出二实根 }
      END
END .
```

以上程序中，花括号{ }括起的部分是注释。注释不算程序的语法成分， 主要用途是在人阅读程序时帮助人理解程序，对计算机不起作用。所以注释在程序中可多可少，可有可无，视人的阅读需要而定。将注释全删掉，对程序的编译和运行不发生影响。上面的例子为了照顾初学者，注释较多，真实的程序中对简单的操作一般不加注释，而只加那些具有提示作用的或带有参考信息的注释。注释中间允许使用计算机能接受的几乎全部字符。过去的计算机不能接受汉字，只能用西文加注，现在的大多数 PASCAL 系统都允许在注释中使用汉字。

以上程序中的第一行，除最后的分号外， 就是所谓的“程序首部”。 编译程序在处理 PASCAL 源程序时，一般不区分大小写（除后面将讲到的“字符串”中外），所以前面提到过的 program 在这里可以换成大写。本书的例子中，为了醒目起见，重要的单词一般都

用大写。

程序首部中的标识符 sample 是我们给本程序起的名字。

关于程序参数以后再讲，现在只要记住两点：凡程序中需要通过标准输入设备（一般指键盘输入）输入数据者，程序参数中必须写上 input ；凡程序中需要通过标准输出设备（一般指显示器）输出数据者，程序参数中必须写上 output 。各参数间用逗号分隔。

以上程序中，从第三行直到最后的 END 为止（不包括最后的句号）， 就是所谓的“分程序”。PASCAL 语言中用“分程序”一词处有三种情况。本例中的分程序是作为整个程序的主体的，称作“程序分程序”。另外两种是“过程分程序”和“函数分程序”，分别是作为过程的主体和函数的主体，这在以后再讲。“程序分程序”在有些资料中称作“程序体”。

分程序又可分为“说明部分”和“语句部分”。以上程序中第三、四两行是说明部分，从第五行开始到最后是语句部分。

先介绍语句部分。语句部分又称作执行部分，它规定了这个程序运行时要执行的算法动作。我们将这一段程序和第一章中列出的框图及伪代码对照，可以很容易看懂它的意思。

按规定，语句部分应由 BEGIN 开头，END 结尾，中间是“语句序列”。所谓“语句序列”，就是若干个“语句”，其间由分号分隔。这个例子中用到的各种语句将在以后详细讲解，这里先给以简单介绍。

其中第一句 read(a,b,c)是从标准输入设备输入数据赋给变量 a，b 和 c。

第二句是赋值语句，其中的:=是赋值号，意思是求出右边表达式的值赋给左边的变量。

IF 开头的是“如果语句”，将在第四章讲解。“如果语句”的意思是根据条件是否成立来选择执行 THEN 后面的操作还是执行 ELSE 后面的操作。这个如果语句是一个复杂的构造型语句，从 IF 开始直到程序倒数第二行的 END 都属于这一个语句。 构造型语句的内部又可以包含别的语句，所以它里面又包含了几个更小的语句。其中的 writeln 语句和 write 语句是向标准输出设备输出数据。

以后我们会知道，这里“语句部分”的定义和“复合语句”的定义是一样的，所以也可以说，整个语句部分是一个大的复合语句。BEGIN 和 END 是一对语句括号。

再介绍说明部分。说明部分中依次包括“标号说明部分”、“常量定义部分”、“类型定义部分”、“变量说明部分”以及“过程与函数说明部分”。但是这五部分并不一定都有内容，根据需要每一部分都可以有也可以没有（但它们的次序不能颠倒）。上面的例子中就只有变量说明部分

```
VAR
  a,b,c,d,r,p,x1,x2:real;
```

其意思是定义了 a,b,c,d,r,p,x1,x2 八个变量名，并用 real 来说明它们都是实数类型的变量。

这五个说明（或定义）部分的前四个各有一个专门的“字符号”来标志它的开头，如下：

标号说明部分　label
常量定义部分　const
类型定义部分　type
变量说明部分　var

至于“过程与函数说明部分”没有总的开头标志，但“过程与函数说明部分”由若干个“过程说明”或“函数说明”组成，而每个“过程说明”或“函数说明”的开头字符号是：

过程说明　procedure
函数说明　function

本节讲到的有关分程序的语法图如图 2.1。

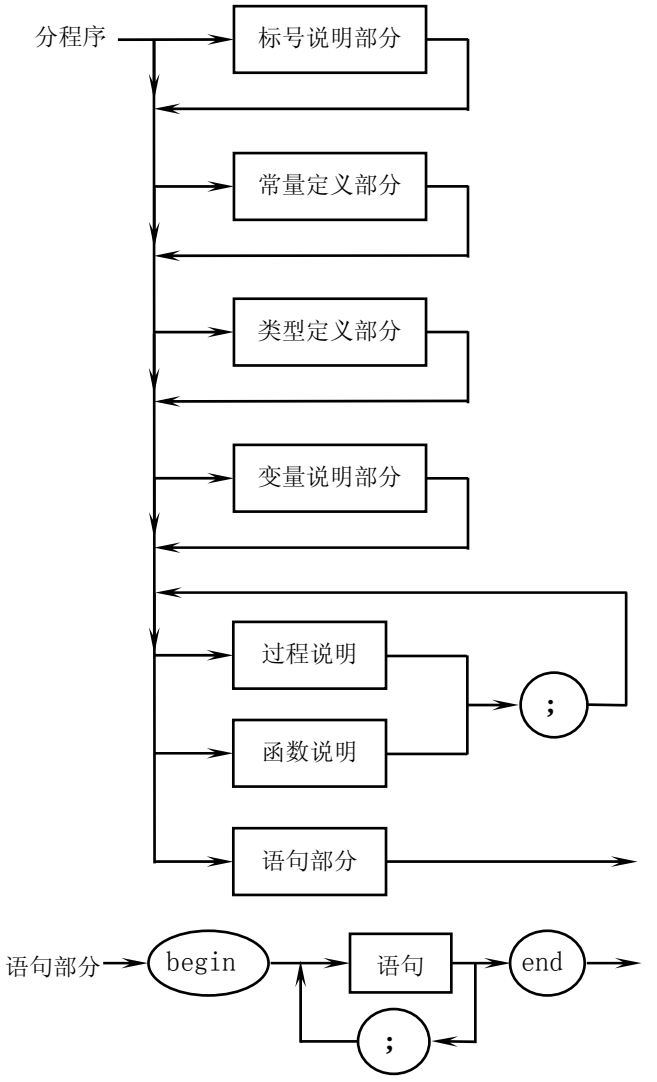


图 2.1　分程序的语法图

2.3　词法记号及分隔符

2.3.1 基本字符

编写 PASCAL 程序可用的基本字符有：

(1) 26 个英文字母 a～z；

(2) 10 个数字字符 0～9；

(3) 下面“其它特定符号”及“分隔符”中所列的标点符号。

英文字母只要不是写在字符串中，则编译时大小写就不加区分。所以编写程序时可以随程序员的爱好而任意用大写或小写。

除了上述三类字符外，通常计算机还可以接受其它一些字符，如!号，&号等，但这些其它的字符只能写在字符串中或注释中。除字符串中及注释中外，程序中只可以使用上述三类字符。

2.3.2 词法记号综述

就好像一篇文章由许多单词组成一样，一个 PASCAL 程序也是由基本的语言单位组成的，这种语言单位称作词法记号，简称为记号。记号之间可以有分隔符。

PASCAL 的词法记号有六种：

特定符号（包括字符号及其它符号）；
标识符；
指示字；
无正负号数；
标号；
字符串。

其中指示字、无正负号数、标号、字符串等以后再讲，这里介绍特定符号及标识符。

2.3.3 特定符号——字符号及其它符号

特定符号[①]是PASCAL语言中规定的有特殊意义的符号，用来界限PASCAL的语法单位。例如前面已介绍过program标志程序首部开头，var标志变量说明部分开头，以及:=号，分号等各有特定作用。

特定符号又分两类，一类是用英文单词或其缩写表示的，称作“字符号”（又称保留字或关键字）；另一类是用标点符号构成的其它符号。它们的意义及用法分别在用到的时候讲解。

字符号有以下 35 个：

```
and       array     begin     case      const     div
do        downto    else      end       file      for
function  goto      if        in        label     mod
nil       not       of        or        packed    procedure
program   record    repeat    set       then      to
type      until     var       while     with
```

其它特定符号有以下 22 个:

```
+       -       *       /       =       <       >       [       ]
.       ,       :       ;       ^       (       )       '       <>
>=      <=      :=      ..
```

2.3.4 标识符

标识符是程序中用来表记某些名字的。如程序名、变量名、常量名、类型名、过程名、函数名以及界限标识符等。上一章已经提到过，标识符是以字母开头的字母、数字组合。

[①] 有些书籍资料中将这里的特定符号称作界限符，但按标准，“界限标识符”一词有其它含义，见第八章。为了避免混乱，本书不用界限符一词。

如上一节例子中用 sample 作程序名，用 a，b，c，d，r，p，x1，x2 分别作八个变量名，这里的 sample，a，b，c，d，r，p，x1，x2 等等都是标识符。

标识符大多是程序员自己取的，但也有些是 PASCAL 语言中规定好的，如前面例子中用的函数名 sqrt，abs 以及数据类型名 real，过程名 read，write，writeln 等等。 这些规定的特殊标识符称作预定义标识符，详见后面介绍。

程序员自己取标识符时不得和 PASCAL 中的 35 个字符号相同，也最好不要和预定义标识符相同。

按标准，标识符的长度没有限制，其中每个字符都具有区分不同标识符的作用。但在某些实际的 PASCAL 系统中只要前 8 个字符相同就被认为是相同的标识符， 后面部分不起区分的作用。

程序员在选用标识符时，可以选择容易“顾名思义”的字符拼写，也可以选用符合数学或物理习惯的符号，以增加程序的可阅读性。但要注意两点：

其一，标识符中不可以有既不是英文字母又不是数字的其它字符，如 π， θ，小数点，逗号，短横（同减号）等。也不可以夹有下面将讲到的分隔符如空格。

其二，标识符是基本的词法记号，从语法上说是不可拆分的。这一点与数学物理上的习惯不同。例如上述例子中的 x1 和 x2 是两个独立的标识符，而不能看成 x 后加了下标 1 和 2。 按数学物理的习惯写法，若另有一变量 k=1，则 x_k 就可代表 x_1。但 PASCAL 中绝不可这样理解，PASCAL 中若写 xk，就成为和 x1 及 x2 完全不同的另一个标识符了。PASCAL 中要想表示带下标的变量，必须采用后面第七章将介绍的表示法。有时候，人们取的标识符是多个英文单词连成的，看起来很像是一个词组或一句话。但这样做仅仅是为了便于人来阅读，而从机器的立场看它仍然是一个独立不可分割的基本成分。

标识符写在程序里有两种情况，一是“定义”，一是“引用”。上节例子中变量说明部分里写的八个变量名就是定义，而语句部分里再使用这几个变量名处就是引用。除预定义标识符外，程序中的标识符必须有定义才能引用。一般情况下（除了后面将要讲到的少数情况外）定义必须出现在引用之前，这就是所谓的“先定义，后引用”的原则。各类标识符的定义格式不同，如程序名只要写在程序首部中就算是定义了，而作为变量名的标识符必须在变量说明部分里定义，本书后面将分别介绍。如果不定义就引用，编译程序会报告程序错误。

这种必须有定义才能引用的规定，可在很大的程度上避免由于拼写笔误造成的错误。若没有这种规定，一旦拼写错误，计算机会以为是另一个标识符，而实际程序的功能就不是原来应有的功能了。有了这种规定后，编译程序会按照“未定义的标识符”来报告错误，帮助我们发现笔误。

每一个定义都有它的有效作用范围，这个范围称作作用域。引用一个标识符必须在该定义的作用域以内。在同一作用域内不允许重复定义同名的标识符，如果重复定义也会报错。关于作用域本书后面专门讲解，在最初几章讲的简单程序中可以认为作用域就是整个程序。

2.3.5 预定义标识符

预定义标识符又称需求标识符，PASCAL 语言已预先定义好了它们的意义，程序中引用

时不必再去定义。标准 PASCAL 中提供了 40 个预定义标识符，它们的意义及用法分别在各章中讲解，这里列出如下：

预定义常量：

false　true　maxint

预定义类型：

boolean　char　integer　real　text

预定义文件变量：

input　output

预定义函数：

abs　arctan　chr　cos　eof　eoln　exp
ln　odd　ord　pred　round　sin　sqr
sqrt　succ　trunc

预定义过程：

reset　rewrite　get　put　read　readln　write
writeln　page　pack　unpack　new　dispose

“预定义”一词有时又常称作“标准”，如预定义类型称作“标准类型”、预定义文件变量称作“标准文件”、预定义函数和过程分别称作“标准函数”、“标准过程”等等。用以和程序员“自定义”的这些名字相区别。

2.3.6 分隔符

程序中最常用的分隔符是空格。分隔符不是语法成分。分隔符可以出现在词法记号之间起分隔作用。标识符、字符号、标号或无正负号数这四种记号若相连出现的话会分不清每个记号的界限，所以它们之间必须有一个或一个以上的分隔符。其它情况下记号之间可以没有分隔符也可以有一个或一个以上的分隔符。如

```
IF x>y THEN a:=24.5 ELSE a:=49
```

语句中 IF 和 x 间，y 和 THEN 间，THEN 和 a 间，24.5 和 ELSE 间，ELSE 和 a 间都按要求加了空格。但 x 和>间，:=号和数 24.5 间则可以不空（当然加上空格也可以）。

程序开头前和结尾后也允许任意加上分隔符。

但应注意，词法记号内部不应出现分隔符。如数 24.5 若写成 24. 5 则是错的，THEN 写成 TH EN 也是错的，:=号也不应写成: =。

除空格外，换行也算一种分隔符。

另外，注释也是一种分隔符。前面已说过，注释是由花括号中写上任意可由计算机接受的字符序列构成的。注释中的文字内还可以包括换行。但是注释中的文字内不应出现后花括号，否则计算机会误以为注释提前结束。

考虑到有些计算机键盘没有花括号，PASCAL 中允许用(*代替{号，用*)代替}号，这样一来，注释中的文字内当然也就不应出现*)了。

由于分隔符允许加任意个，所以我们可以选择合适处换行，并用加空格的办法使上下行按我们希望的形状对齐，将程序清单整理得清楚易读。

例如前面曾指出过：计算机算法中层次嵌套的情况是经常出现的，但层次一多就会给

人阅读理解带来困难。为了减少人的阅读困难，我们通常将同一层次的段落开头上下对齐，而内层的内容适当向右缩进，同一层语句前括号 BEGIN 和语句后括号 END 若不在同一行的话也上下对齐，使嵌套层次醒目些。在 2.2 节的例子程序中我们就可以看到这种做法。

不采用这种对齐和缩进的做法对机器来说不算错误，所以常有初学者因为嫌麻烦而不愿意采用这种做法，更不愿意花时间写注释。这样的态度是得不偿失的。大量的实践证明，通常查错改错的工作量远远大于写程序的工作量，写程序时稍费一些精力将清单整理好，可以大大减少查错的难度。希望读者认真体会，养成良好的习惯。

2.4 数据类型的概念及预定义的数据类型

2.4.1 概述

预定义类型又称需求类型、标准类型。

所谓“类型”，即数据类型，包含两方面的概念：一是每种类型的数据有一定的取值范围，即所谓的值域。程序中表记数据的东西有常量、变量、表达式等，都有个取值的问题，也都需要确定它们的类型。二是对每一种类型的数据规定了一组可以执行的运算或操作。如果程序中所编写的运算与它的类型不一致，编译程序就会报告错误信息。

PASCAL 数据类型的总表见表 2.1。

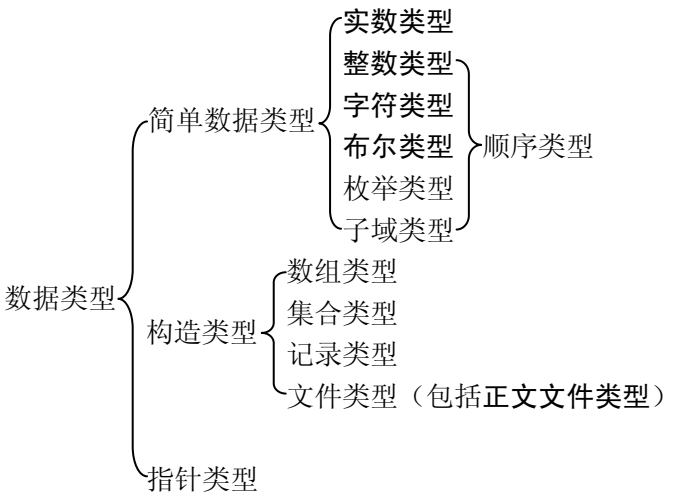


表 2.1 PASCAL 数据类型总表

PASCAL 最重要的特征之一是它提供了多种预定义数据类型，同时还允许程序员自己定义新的数据类型。表 2.1 中黑体字是预定义类型。除预定义类型外，表中所列都不是某一“个”类型的名字，而是某一“类”类型的总称。如“枚举类型”就是如此，需要由程序员自己定义，程序员可以引入多个不同的枚举类型。自定义类型以后才能讲到，本节下面只介绍预定义类型。

预定义类型有五个，它们是

real	实数类型（简称实型）
integer	整数类型（简称整型）
char	字符类型
boolean	布尔类型
text	正文文件（又称文本文件）类型

其中最后一个属于文件类型的一种，在第十一章再讲。本节先介绍前四个简单的预定义类型。

2.4.2 实数类型 real

实型数据的取值范围是计算机上可以表示的实数。

PASCAL 程序中要确定一个变量标识符属于实型的变量，需要在 VAR 开头的“变量说明”部分里说明。如 2.2 节的例子中的

```
VAR
  a,b,c,d,r,p,x1,x2:real;
```

其中类型标识符 real 就是说明 a,b,c,d,r,p,x1,x2 八个变量都是实数类型的变量。

要在程序中表示实型常量，一般可以用字面的形式书写。PASCAL 程序中规定实型常量的字面形式有两种：小数形式和指数形式。

小数形式的字面常量如

```
77.23
1.12
0.00044
-3.2
0.0
100.0
```

等。注意这种形式必须有小数点，而且小数点前后都要有数字，否则系统会认为出错，或认为它不是实数类型的数据。正数前面可以加正号，也可以不加正号。

指数形式又称浮点形式，或科学表示法，如

7.723e1	（表示 7.723×10^{1}）
1.12e0	（表示 1.12×10^{0}）
4.4e-4	（表示 4.4×10^{-4}）
-3.2e0	（表示-3.2×10^{0}）
0e0	（表示 0×10^{0}）
1e2	（表示 1×10^{2}）

从例子中可以看到 e 后的数字代表 10 的幂次。e 也可以写作大写 E。具体的规定：e 之前可以和小数形式的写法一样，也可以只写整数。e 之后只可写整数。 整个数若为正数前面可以加正号，也可以不加正号。另外，指数部分若为正，也允许加正号，如 1e2 也可以写作 1e+2。 这种表示法中 e 的前后都应有数字，如 1e2 不可写作 e2。

应注意不管是小数形式还是指数形式，整个字面常量（除开头的正负号外）是一个基本的词法记号，应看成一个整体，所以中间不可插入空格或其它分隔符。特别是指数形式，

若有人误以为它是一个复杂的式子，以为在程序执行时还会进行乘以 10 的几次方的运算，那就错了。这种形式只是源程序中的一种写法，实际编译成目标程序时早已化为内部的二进制浮点形式的一个数了。

实数字面常量以及下面将介绍的整数字面常量，除开头的正负号外，都应算是一个基本的词法记号，属于第 2.3 节介绍的“无正负号数”一类。

还应注意实数字面常量没有分数形式。如果写成如 3/4 这样的形式， 就不认为是常量，而认为是一个除法运算的表达式。

实型数据可以进行四则运算，其结果仍为实型：

+ （加）、- （减）、* （乘）、/ （除）

应注意乘号不是×而是 * ，除号不是÷而是 / 。而且写在表达式中时乘号不可省略，也不可换成圆点。

实型数据还可以进行比较运算，其结果为布尔型，详见下文。

用于实型量的预定义函数有：

abs （绝对值）、sqr （平方）、sqrt（平方根）、
sin （正弦）、cos （余弦）、arctan （反正切）、
exp （以 e 为底的指数）、ln （自然对数）、
trunc （截尾取整）、round （舍入取整）

其中除最后两个外，运算的结果仍为实型。

PASCAL 函数的参数（即自变量）必须写在括号内。如

sin2t 应写成 sin(2*t)
$\sqrt{2.5}$ 应写成 sqrt(2.5)
e^{x+y} 应写成 exp(x+y)
$(a+b)^2$ 应写成 sqr(a+b)

sin 和 cos 的自变量以及 arctan 的结果都是以弧度为单位。若要求 32.1 度的正弦应先化为弧度再引用 sin 函数：

sin(32.1*3.14159265/180.0)

exp 及 ln 都是以 e 为底，若要求十进对数可用换底公式。如

lg24.5 可写成 ln(24.5)/ln(10.0)

PASCAL 中没有一般的乘方运算符。要进行乘方运算，若指数为小的正整数，可以用连乘实现；否则，可以用以 e 为底的对数和指数来计算。如

x^y 可写成 exp(y*ln(x))

trunc 及 round 的运算结果是整数类型。trunc 是去掉小数部分，取其整数。round 是将小数部分四舍五入化整。

2.4.3 整数类型 integer

整型数据的取值范围是计算机上可以表示的整数。

要在变量说明部分里说明整型的变量可以使用类型标识符 integer。如

VAR

```
i,j,k:integer;
```

就是说明 i,j,k 三个变量都是整数类型的变量。

程序中整型常量的字面形式很简单，就是普通数学的写法。除了开头可有正负号以外，不得有非数字的字符。当然也不能有小数点。如

```
12326
-3001
   3
```

等。

整型数据可以进行

```
+  （加）、-  （减）、*  （乘）
```

三种算术运算，若两个运算数都是整型，则其结果仍为整型。

整型数据还可以进行两种除法：

```
DIV  （整除求商）、MOD  （整除求余）
```

要求两个运算数都是整型，其结果仍为整型。如

```
17 DIV 5 = 3
17 MOD 5 = 2
```

关于这两个除法，还要作一些进一步的说明。上例中 17 被 5 整除商 3 余 2，有以下关系

```
17 = 3 * 5 + 2
```

从算术上理解是很清楚的。但是如果被除数和除数有负数，关系就不太清楚了。例如-17 除以 5，从算术上理解，可以按照关系

```
-17 = (-3) * 5 + (-2)          (1)
```

认为商-3 余-2，也可以按照关系

```
-17 = (-4) * 5 + 3             (2)
```

认为商-4 余 3。

而按照标准 PASCAL 的规定，却是

```
(-17) DIV 5 = -3
(-17) MOD 5 = 3
```

可见 DIV 是按关系(1)确定的商，而 MOD 却是按关系(2)确定的余数。这两个关系的差别是它们对商的化整处理方向不同。

按实数除法(-17)/5 得-3.4，在-3 和-4 之间，有小数部分，需要化整。DIV 运算中是按绝对值减小的方向化整得-3，而 MOD 运算中是按带符号的值减小的方向化整得-4。二者商相差 1，余数相应也不同。若恰好除尽就没有这个差别，对于正数相除也没有这个差别。

顺便指出，trunc 函数取整是按绝对值减小的方向进行的，与 DIV 中商的化整方向一致。

其所以要作不同规定，在 DIV 中是为了照顾人们只舍不入处理时的习惯，而在 MOD 中则是为了保持数论中“同余”、“模”的概念和规律在负数中仍然成立，以便简化某些运算。所以，MOD 运算也称作“取模”。

另外，MOD 运算要求除数必须为正，而 DIV 运算中除数可正可负（当然不得是 0）。

初学者如果一时记不清这些细节规定，可在程序中回避负数的整除。

整型数据还可以进行比较运算，其结果为布尔型，详见下文。

可用于整型量且结果仍为整型量的预定义的算术函数有 abs 和 sqr，意义和前面的说明相同。

另外，整数类型属于一种顺序类型，故它可进行下面将介绍的各个顺序类型都有的函数运算 ord、succ 和 pred。这三个缩写单词的意思是

ord　　序号

succ　　后继

pred　　前启

具体对整型来说，ord 函数的值与自变量相同，succ 函数的值比自变量大 1，pred 函数的值比自变量小 1。如

ord(15)=15,　　succ(15)=16,　　pred(15)=14,

succ(-5)=-4,　　pred(-5)=-6

还有整数向字符型转换的函数 chr 在下面讲字符类型时介绍， 判断整数是否奇数的函数 odd 在下面讲布尔类型时介绍。同时下面还会看到，所有面向实型的运算也都可以用于整型。

2.4.4 实数类型与整数类型的联系及比较

虽然在数学意义下整数是实数的子集，但是因为它们在计算机内的表示形式不同，所以 PASCAL 语言把它们看作不同的类型。例如 13 和 13.0 被看作是不同的。

同时，为了符合数学上的习惯，PASCAL 中又采用了以下的附加规定：表达式中任何实数值的操作数都可以用整数值的操作数来代替。所以 2.5+13 与 2.5+13.0 的结果实际上又是一样的。不过应知道，编译程序对于这种情况的每次出现，都要自动插入隐含的转换操作，先将该整数值转换成实数的内部表示形式再参加运算。

按照这个规定，我们可以知道，上面用于实数类型的所有运算和函数也都可以用于整数类型。只是除 abs 和 sqr 外的算术函数结果总是实型，整型与实型混合的四则运算结果总是实型，而除法 / 即使两个运算数都是整型，其结果也是实型。例如

sqrt(9)

的结果是实型，相当于实数 3.0；而

30/5

的结果是实型，相当于实数 6.0。

然而反过来，在只有整型才是合法的地方，却不能直接代以实数值的操作数。例如除法 DIV 和 MOD 就不能对实型数据进行。假若 x 是实型变量，则

x MOD 5

是不允许的。需要在这里使用实数时，必须用一显式的转换函数如 trunc 或 round 等，将其转换为整数值。如

trunc(x) MOD 5

或

round(x) MOD 5

等（这两种形式的结果不完全相同）。

大多数的实际 PASCAL 系统中，整型数据的内部存储采用定点形式，最低有效位是个位，而实型数据的内部存储采用浮点形式。所以二者还有以下区别：

其一，整数运算没有误差。实数运算结果保留一定有效数字，可能会有误差。不仅运算时有误差，实数一存入计算机就可能有误差。这是因为输入的数据或程序中写的字面常量都是十进制，而十进制的有限位小数化为二进制后不一定是有限位，若是无限位就不得不舍入近似。

其二，整数保留到个位，而机器字长有限，所以其绝对值的范围不可以太大，而实数则范围可以大得多。常见的某几种 PASCAL 系统采用 16 位二进制补码表示整数，取值范围可以为-32768～32767；而实型数则绝对值最大可以达到 10^{38} 的数量级，绝对值最小除全 0 外可以达到 10^{-38} 的数量级。当然实型数据采用浮点存储，不同数据的绝对精度是不同的，绝对值越小的数绝对精度越高，绝对值大的数绝对误差也较大。

运算结果若超出该类型允许的最大范围称作溢出。溢出一般算作错误，但某些实际系统中为了照顾效率，对整数溢出并不报告出错。了解内部补码运算原理的读者，自己可以分析出整数溢出时所产生的结果，但这不在本书的叙述范围内，故不详述。而实数若绝对值过小通常称作“下溢”，“下溢”时一般将它看作全 0，不算错误，只有绝对值过大（“上溢”）才报告溢出错误。

预定义的标识符 maxint 代表该 PASCAL 系统中最大的整数，例如上面所举的 PASCAL 系统中 maxint 就代表 32767，作为一个整型的符号常量，可以直接用在程序中。 不同的机器或不同的 PASCAL 系统中 maxint 代表的数有可能不同。

顺便指出，除法运算若除数为 0，一般算作溢出，但也有的系统将它单列为一种错误。

2.4.5 字符类型 char

计算机不仅可以处理数值性的数据，还可以处理非数值性的数据。非数值性的数据中最常见的就是字符。

字符型数据的取值范围是计算机可以接收的单个字符。不同的具体系统可能会有不同。

要在变量说明部分里说明字符型的变量可以使用类型标识符 char。如

```
VAR
  ch,ch1:char;
```

就是说明 ch 和 ch1 两个变量都是字符类型的变量。

然而程序中字符型的字面常量却不能直接写字符。这是因为整个程序都是用字符拼成的，若字符型的字面常量直接写成字符，就无法和代表其它意义的字符区分。如字符 a 无法和名为 a 的标识符区分，字符 0 无法和整数常量 0 区分。所以 PASCAL 规定程序中字符型的字面常量要将字符前后用撇号括起来。如

```
'A'
'a'
'3'
```

'!'

' '　　（代表空格号）

等。应注意在这里大小写被看作是不同的，这一点与其它地方不一样。若要表示撇号字符本身，则应将中间的撇号写成两个，即

''''

才是代表撇号字符的字面常量。

以后将讲到的字符串常量写法形式上和这里的字符型字面常量是一致的， 所以在一般 PASCAL 的语法图和 Backus-Naur 范式中将字面形式的字符常量也叫做“字符串”。 但是应该知道，字符常量和一般的字符串（长度不为 1 时）概念是不同的，前者是简单类型数据，而后者是构造型数据。

字符型数据在机器内的存储形式取决于具体的实际系统设计。 现在最常见的一种是按 ASCII 编码存储。本书后的附录 A 给出了 ASCII 码的十六进制编码。表中只给出了前 128 个，编码为 0～127（十六进制 0～7F）。后 128 个是扩充部分，各个系统不同，在常见的汉字系统如 CCDOS、UCDOS 等中，编码 161～254（十六进制 A1～FE）用于汉字内码。

前面已提到函数 chr 可将整数转换为字符， 其具体的功能是：将自变量（整数）看作一字符的机内编码，则函数值就是该字符。显然，在采用不同编码方式的系统中，这个函数的功能是不同的。对于采用 ASCII 码的系统，有

chr(65)='A'

chr(97)='a'

chr(32)=' '

等等。因编码范围有限，故这个函数自变量的允许取值范围也有限。

字符类型也属于一种顺序类型，所以它也可以进行下面将介绍的各个顺序类型都有的函数运算 ord、succ 和 pred。具体对字符型来说，ord 函数的值等于自变量字符的机内编码（整数），succ 函数的值为自变量的后一个字符，pred 函数的值为自变量的前一个字符。这里的“后一个”、“前一个”是按该系统所用的字符编码表中的次序计算的。如对于采用 ASCII 码的系统，有

ord('A')=65,　　ord('@')=64,　　ord('B')=66,

succ('A')='B',　pred('A')='@'

在这里，可以认为 ord 和 chr 互为反函数。假如有一变量 ch 是字符类型的，则总有

chr(ord(ch))=ch

假如有一变量 n 是整数类型的，且其值在可用字符的编码范围之内，则总有

ord(chr(n))=n

2.4.6 布尔类型 boolean

布尔类型的数据表示一个逻辑判断的结果。如果某个条件成立，我们说它的判断结果为“真”；如果某个条件不成立，我们说它的判断结果为“假”。所以布尔类型数据的取值只有两个：“真”和“假”。

例如 2.2 节的例子程序中在 IF 后面有一个式子 d<0，这个式子的值就是布尔类型的数据：当 d 小于 0 时，它的值为“真”；当 d 不小于 0 时，它的值为“假”。

要在变量说明部分里说明布尔型的变量可以使用类型标识符 boolean。如

```
VAR
  flag:boolean;
```

就是说明变量 flag 是布尔类型的变量。

布尔类型没有字面形式的常量。有两个预定义的常量标识符可用作布尔类型的符号常量：

false　表示“假”
true　表示“真”

说明文字中习惯上就以这两个记号代表布尔型的两个值。PASCAL 中往显示器上或往其它正文设备或文件上输出布尔型值时也以这两个记号表示。

PASCAL 中将关系符看作一种运算符，其运算结果为布尔型的值。关系运算有以下 7 种：

=　（等于）
<　（小于）
>　（大于）
<>　（不等于）
<=　（小于等于）
>=　（大于等于）
in　（属于）

其中最后一个在第九章讲解。前面六个可以按数学上的意义来使用，用于整数或实数间的比较，也可以将整数和实数混合比较。应注意，由于实数不能保证无误差，所以进行实数间等于或不等于的判断时结果有时不能保证可靠。实际程序中大都避免对实数直接进行等于或不等于的比较，需要作相等判断的数值大都设成整型。有时需要判断两实数 x 和 y 是否近似相等，可以判断其差的绝对值是否在某个小范围以内。如

```
abs(x-y)<1e-5
```

下面会看到，这 6 种关系运算也可用于其它顺序类型。

前面还提到一个函数 odd，自变量是整数，作用是判断它是否奇数， 函数值为布尔型。如

odd(101)　结果为 true
odd(-100)　结果为 false

PASCAL 中对布尔型数据还提供了三种布尔运算（又称逻辑运算）：

AND　（与）
OR　（或）
NOT　（非）

它们在第四章中还要进一步讲解。

布尔类型也属于一种顺序类型，所以它也可以进行下面将介绍的各个顺序类型都有的函数运算 ord、succ 和 pred。具体对布尔型来说，有

```
ord(false)=0,          ord(true)=1,
succ(false)=true,      pred(true)=false
```

2.4.7 顺序类型综述

有一类类型称作顺序类型。前面讲过的四个类型中，除了实数类型外，后三个类型都是顺序类型。以后会讲到用户自定义的枚举类型和子域类型也是顺序类型。总之，简单类型中除实型外都是顺序类型。顺序类型有以下特征：

(1)该类型的取值是有限个；

(2)该类型的每个取值对应一个整数作为这个值的序号；

(3)该类型中不同值的序号不同；

(4)该类型所有取值对应的序号恰构成整数域中的一个连续区域。

前面讲到的三个顺序类型中，对于序号的规定是：

(1)整数类型值的序号就是该整数本身；

(2)字符类型值的序号就是该字符的机器编码，如 ASCII 码；

(3)布尔类型值中，false 的序号是 0，true 的序号是 1。

各个顺序类型都有函数运算 ord、succ 和 pred。具体功能是

```
ord      （求序号）
succ     （求后继）
pred     （求前启）
```

所谓“后继”是指同一类型中序号比自变量的序号大 1 的值，所谓“前启”（又称“前趋”、“前导”）是指同一类型中序号比自变量的序号小 1 的值。显然， 一个类型中序号最大的值没有后继，序号最小的值没有前启。如 maxint 没有后继，false 没有前启。 这三个函数中，ord 的函数值属于整数类型，succ 和 pred 的函数值属于和自变量相同的类型。

对顺序类型可以进行=，<，>，<>，<=，>=六种比较。 除整数允许和实数混合比较外，必须是同一种类型的值（更严格的说法是要求类型相容，见本书后面）才能进行这种比较。比较时“大”、“小”均以序号为准，序号小者认为该值也小。如下列关系均为真

```
true>false
'A'<'B'
pred(x)<x          （假设 x 取一顺序类型的值且 pred(x)存在，下同）
succ(pred(x))=x
```

对于字符类型，因不同系统可能采用不同编码，故序号有可能不一样。但只要是符合标准 PASCAL 要求的系统，下列关系一定成立：

```
succ('0')='1', succ('1')='2', ..... succ('8')='9',
'A'<'B', 'B'<'C', ..... 'Y'<'Z',
'a'<'b', 'b'<'c', ..... 'y'<'z'。
```

实数不算顺序类型。从数学观点上看，每个区间实数都有无数个值，因而一个实数找不到它的后继和前启。虽然从计算机的观点上看，因为精度有限，实际能表达的数值仍是有限个，但是这有限个数的间隔是由误差形成的，不可能明确确定。所以不可能给实数规定出一种序号。

实数虽不算顺序类型，但仍是“有序”的，可以进行大小比较。所有简单类型都是有序的。

2.5 常量、变量和表达式

2.5.1 常量

常量顾名思义就是其值不变的量。程序中表记同一个常量的记号在不同时候，不同的地方都代表同一个值。

常量在程序中有两种形式:字面形式和符号形式。即字面常量和符号常量。某些 PASCAL 允许写“常量表达式”来表记常量，但标准 PASCAL 中无此规定。

字面常量的写法已在上一节介绍。

符号常量就是用标识符作为名字的常量。作为常量名的标识符,除预定义的(如 maxint, false, true 等）以外，都需要定义后才能引用。定义的方法之一是写在“常量定义部分”中。

前面已经知道，“常量定义部分”是分程序中说明部分里的第二个部分。它的语法图见图 2.2。

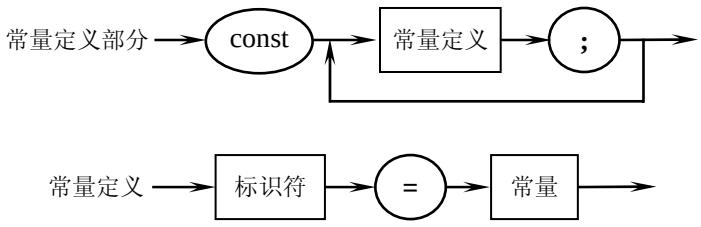


图 2.2 常量定义部分的语法图

从图中可以看到“常量定义部分”的格式是在字符号 const 后跟上一条或多条“常量定义”,且每条常量定义后跟上一个分号构成的。而“常量定义”的格式是在等号左边写一标识符，右边写一常量。左边的标识符就是要定义的常量名，右边的“常量”可写字面常量，也可写已知的常量名。这条定义就规定了以左边的标识符为名的常量表记右边的常量的值。

下面是常量定义部分的一个例子：

```
CONST
  pi=3.14159265;
  g=9.80665;
  cross='X';
  yes=true;
  no=false;
```

作了这样的定义后，在语句部分中就可以用 pi 来代表实数 3.14159265，用 cross 来代表字符常量'X'，等等。

使用符号常量的优越性主要有以下几点:

(1)少用字面常量可以减少抄写笔误。如上面的 π 值和标准重力加速度， 可能在程序中多处用到。若不用符号常量就得多遍誊写，每一处都必须分别校对以避免笔误。用了符号常量就可以只校对一处数字，其它地方只要名字不用错就行。而名字拼写的错误大部分可由编译系统自动查出。

(2)便于修改。有些常数是经验数据，试验过程中可能需要修改。若不用符号常量， 所有使用这一数据的地方都得修改。用了符号常量，只要修改它的定义就行了。

(3)可以按顾名思义或符合习惯的原则来命名，有助于阅读理解程序。例如用 cross 代表字符常量'X'，可能意味着本程序中用这个字母表示“打叉”， 熟悉英文的读者容易理解。还有 pi 英文读音同希腊字母π，数学上习惯表示圆周率，g 在物理上习惯表示重力加速度，等。

(4)允许常量有别名，从而增加灵活性，有时可便于程序的组合。如上例中将 yes 当做 true 的别名，将 no 当做 false 的别名。

符号常量只用一个标识符表示，形式上很像一个下面讲的变量，所以我们一定要注意不要把它误当成变量来用，即：程序中不可以对它赋值。常量定义中的等号是“定义”而不是“赋值”，该值是在编译时确定的而不是运行时才赋给这个名字的。即使是允许使用常量表达式的系统，该表达式也是在编译时计算以确定其值，运行时就不再算了。

以后我们会见到，还有另一种符号常量——表记枚举类型值的标识符，不是在常量定义部分里定义，而是写在枚举类型的新类型描述格式中定义的。程序中每出现一个新枚举类型的描述格式，就算是定义了一组常量标识符。

2.5.2 变量

变量，顾名思义，就是同一个名字的量在程序中不同时刻可以有不同的值。

Dijkstra 曾经引用一个不知名的人的话：“人们一旦了解在程序设计中如何使用变量，他就掌握了程序设计的精华。”他之所以如此重视变量的概念，是因为只有引入了变量的概念后，人们才有可能用简短的形式去描述繁琐的计算，从而真正发挥计算机的作用。

我们知道，计算机的动作终究是受人指挥的。虽然计算机的速度可以达到每秒成千上万甚至上亿次计算，但假如没有简短的描述办法，而是每步计算都要等人去分别告诉它，那么它的计算速度再快也是没有意义的，因为人写程序的速度太慢了。

例如有这样的问题：

已知 $x_0=1$，对任意正整数 k 有 $x_k=0.8x_{k-1}+1$，求 x_{10000}。

这个题目，若推出一个公式，将长得不可思议。若分步计算，由 x_0 求出 x_1，再由 x_1 求出 x_2，等等，则会产生 10000 个中间结果。假如每个中间值都用不同的名字， 则程序中就得写出 10000 多个不同的语句：

```
x0:=1;
x1:=x0*0.8+1;
x2:=x1*0.8+1;
......
......
x10000:=x9999*0.8+1;
write(x10000)
```

现在引入了变量的概念，我们就可以用同一个名字 x 来先后表记上述 10000 个不同的值。我们用 x:=x*0.8+1 来表示“将 x 原有的值乘以 0.8 再加上 1，结果赋给 x”，作完这一操作后 x 就具有了新的值。这样，上述 10000 个不同的语句就变成了同一语句重复 10000

次，可以用循环结构简短地编写为：

```
x:=1;
FOR i:=1 TO 10000 DO x:=x*0.8+1;
write(x)
```

于是 10000 多个语句减为 3 句，其中第二句意思就是“将 x:=x*0.8+1 重复 10000 次”。 第五章我们会讲到，这个例子是一个递推问题。递推问题都可以用循环结构来解决。

变量在程序中具有两重属性：名和值。一个变量，其名字固定，但其值随着每次赋值而变化。PASCAL 语言中，要给变量赋值可以用赋值语句，也可以用输入语句等，这些将在第三章及以后介绍。实际上，上例中的 x:=x*0.8+1 就是一个赋值语句。

PASCAL 程序中的变量有四种形式:

(1)整体变量，即直接用一个标识符作为其名字的变量;

(2)成分变量，即数组中的一个下标变量或记录中的一个域;

(3)标识变量，即利用指针开辟的动态变量;

(4)缓冲区变量，即系统开辟的缓冲存放文件当前成分的变量。

后三种各有规定的表示格式，后面才能讲到。现在只介绍整体变量，即直接用一个标识符作为其名字的变量。

前面的例题中已经看到，作为变量名的标识符一般是在变量说明部分里定义的。变量说明部分的语法图见图 2.3。

从图中可以看到“变量说明部分”的格式是在字符号 VAR 后面跟上一条或多条“变量说明”，且每条变量说明后跟上一个分号构成的。

而“变量说明”的格式是在冒号左边写若干个标识符，若标识符不只一个则其间用逗号隔开，冒号右边写一个“类型表记符”。

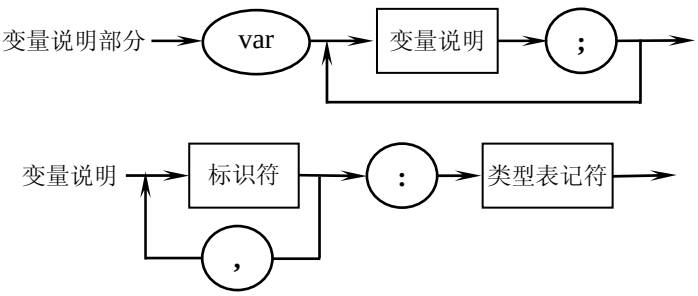


图 2.3　变量说明部分的语法图

所谓“类型表记符”是指已有定义的类型标识符或新类型的描述格式。目前尚未学到自定义的新类型，所以在“类型表记符”处暂时只能用前面学过的预定义类型名。关于类型表记符及新类型描述格式将在第六章及以后全面介绍。

每条变量说明中，冒号左边的那些标识符就是要定义的变量名，冒号右边的“类型表记符”是说明这些变量的类型。显然，同一条变量说明中定义的变量都属于同一数据类型。

下面是变量说明部分的一个例子：

```
VAR
  i,j,k:integer;
  ch,ch1:char;
  flag:boolean;
```

这里定义了三个整型变量，两个字符型变量及一个布尔变量。

2.5.3 表达式

PASCAL 中许多地方用到表达式，如赋值语句中:=号右边就是表达式。

PASCAL 的标准中规定：“表达式应表记一个值。”按照我们已经学过的概念，值是属于一定类型的。所以表达式也有一定的类型。如果表达式的值属于某某类型，一般就将它称作某某表达式。

单个的常量可以成为一个表达式。单个的变量也可以成为一个表达式。一个函数命名符（如 sin(x)）也可以成为一个表达式。

更一般的表达式是将这些成分用各种运算符按照规则组合而成的式子。应注意不管表达式的中间计算过程有多长，总是最后的运算结果才算是它的值。如

```
trunc(sin(3.5)+1.1)
```

应算是整型的表达式，虽然中间结果是实型的。

应注意，PASCAL 中应在哪些地方写表达式是有严格规定的，不该写表达式的地方即使有同样形式的成分也不算表达式。例如：虽然单个的变量可以成为一个表达式，但是写在赋值语句中:=号左边的变量不是表达式。以后我们还会看到：子程序的变量参数中代入的实在参数不是表达式，子域类型的新类型描述格式的上下界处写的常量也不是表达式，等等。

前面已说过，变量在程序中具有两重属性：名和值。出现在表达式中的变量是其“值”起作用[①]。同时PASCAL中又规定，变量在第一次被赋值之前其值是“未定义的”（唯一的例外是外部文件）。若在表达式中使用未定义的值，严格地说算错误。不过实际的PASCAL系统大多并不报错， 也能给出一个值， 但是不保证这个值一定是几。 这点和其它某些语言如BASIC是不同的。所以变量第一次出现在表达式中之前应该赋过初值。

下文某些地方（如 FOR 语句）对变量值的叙述中还会用到“未定义的”一语， 意思和这里相同。

表达式中可用的运算符及函数已在前面介绍过一些。应注意:

(1)不可使用 PASCAL 中没有规定的运算符和函数。如一般乘方运算等， 又如除号不能写成÷，>=号不能写成≥等。运算符都属于特定符号，函数名都是标识符。预定义的函数中没有正切，所以如果我们没有自定义一个正切函数的话，表达式中也就不可以出现正切。如果我们定义的正切是 tan，用在表达式中时也不可以换成 tg。

(2)表达式必须写成线性形式，即必须是由字符组成的普通序列， 而不能是其它形式。如不能有分子在上分母在下的分数形式，也不能有写在右上角和右下角的上下标等。

(3)乘号*不可省略，也不可改成圆点。

(4)函数的自变量一定要写在圆括号内。

---

① 严格地说，这里“出现在表达式中的变量”应说成“作为表达式的变量”。一般出现在表达式中的变量虽是作为一个大表达式的一部分，但本身也算一个表达式（子表达式）。既然规定“表达式应表记一个值”，当然此时变量应该是其值起作用。有时候，出现在表达式中的变量本身不是作为一个表达式，例如表达式中的函数命名符里变量参数的实参（第八章），此时该变量就不是值起作用。

(5)另外，为了指定运算次序，只可用圆括号( )而不可用方括号[ ]及花括号{ }。因为后两种括号在 PASCAL 中有其它意义。

表达式的概念是嵌套的。括号内的部分本身就应该是一个表达式，对于每一个运算来说，参加运算的运算数各自也是一个表达式。相对于整个表达式来说，可以把这种作为一个大表达式的一部分的小表达式叫做子表达式。当然，划分子表达式的范围时应该符合下面所述的优先次序。运算的次序，应该是先求出子表达式的结果再参加整个表达式的运算。显然，如果有多层括号，次序是先内后外。

除了括号的作用以外，运算的优先次序如下（由高到低）：

(1)函数，即函数名与参数表（即自变量）间的结合最优先。

(2)NOT。

(3)乘法类运算符，包括*，/，DIV，MOD，AND。

(4)加法类运算符，包括+，-，OR。

(5)关系运算符，包括=，<，>，<>，<=，>=，in。

同一优先级的运算符间的运算次序，凡是二元运算一律从左到右。

上述运算次序，与习惯上的“先乘除后加减”是一致的。不过，PASCAL 中将逻辑运算 OR 也算作加法类，AND 也算作乘法类，NOT 优先级更高，这些与其它地方的习惯不一定一致，不同的高级语言规定也有所不同，须特别注意。

应指出，虽然“先乘除后加减”已是众人皆知的，但仍然常有人将除号的优先级弄错。这是因为除号不是÷而是 / ，常被想象成分式的横线，因而漏掉括号。如分式

$$\frac{x-y}{x+y}$$

写成 PASCAL 的表达式应加上括号：

```
(x-y)/(x+y)
```

若丢掉括号，写成

```
x-y / x+y
```

就错了。

习题

2.1 简述程序的基本组成。

2.2 简述已学过的几类词法记号。下列记号各属于哪一种？

BEGIN	END	IF	THEN	VAR	PROGRAM
+	-	*	:=	<>	;
true	flase	sin	input	real	integer
xyz	a	a12	a12x	t1	s
100	1.2e-1	203.4	1E1	3e+5	0.0

2.3 简述已学过的几个类型，它们的名字，取值范围，适用的运算。

2.4 将下列 PASCAL 表达式的类型及值填入下表，值要求按 PASCAL 中合法的常量形式书写。

表 达 式	类 型	值
4.0-3		
Pred('n')		
Trunc(31/8)		
2.1e-7		
pred('M')		
Round(23/3)		
'A'		
25/5		
ord(odd(4))		
(x>1)=(x<=1)		
(x<5)<=(x<10)		
60 DIV 3 + 4.0		
64/4		
Ord(17<=10)		
Succ('T')		
1+66/22		
Ord('C')-Ord('A')		
'X'<='U'		
Succ(20>33)		

2.5 为什么说人们一旦了解在程序设计中如何使用变量，他就掌握了程序设计的精华？

第三章　简单程序设计

PASCAL 程序设计包括对数据的描述和操作两个方面。数据类型描述了数据的属性和取值范围。在上一章，我们已经介绍了数据类型的概念以及四种基本的数据类型，从本章开始，我们将要对已经说明的数据进行处理。在 PASCAL 语言中，对数据的操作是通过语句来实现的，也就是说，作为处理问题的程序，无论是简单的还是复杂的，都是由一系列语句组成。下面，我们先介绍一下 PASCAL 中的语句分类。

3.1　PASCAL 语句的分类

PASCAL 语句按语句标号的有无，可分为无标号语句和标号语句。无标号语句又分为简单语句和构造型语句两大类。

所谓简单语句，就是一个语句中不再包含有另一个语句。例如:

```
n:=10;
GOTO 5;
read(num);
```

等都是简单语句。简单语句包括赋值语句，过程语句，转向语句和空语句。其中，过程语句有许多种，我们将在本章介绍输入语句和输出语句，而其它的过程语句将在后面的章节中陆续介绍。

构造型语句比较复杂，它的一个语句结构的组成部分中又包含有语句。构造型语句包括复合语句、条件语句、循环类语句[①]和开域语句。条件语句又分为如果语句和情况语句两种。循环类语句又包括三种：WHILE语句、REPEAT语句和FOR语句。

所谓标号语句，就是在一个无标号语句前面加一个标号和一个冒号，例如

```
10:read(m);
```

这里，整数 10 是一个标号，它可以被转向语句所引用，关于标号语句以及引用标号的转向语句，我们将在第五章介绍。

可以将 PASCAL 语句分类列表，见表 3.1。

各种语句在其它章节讲解，这里仅说明一下空语句。空语句不包含任何内容，执行空语句时什么也不做。例如程序中语句序列内两个语句间本应由一个分号分隔，但假如我们多写了一个，成了两个分号，如:

```
m:=10;;
read(n);
```

也不算错。因为编译程序会理解为这两个分号间还有一个空语句，仍然符合语法。

[①]按《GB 7591-87》，WHILE 语句，REPEAT 语句，FOR 语句分别称作“当语句”，“重复语句”及“循环语句”，三者总称为“重复性语句”。但考虑到“循环”一词在各门课程中习惯表示各种循环结构的总称，用它专指 FOR 语句不合习惯， 而且“重复”与“重复性”也不易区分，故本书未采用这些名词 。

PASCAL 中引入空语句的目的主要是为了后面将讲到的转向语句的需要。有时候希望跳转到某个复合语句或分程序的结尾处，就要将语句标号写在 END 之前。但 END 不是语句，而语句标号又必须写在语句之前，若没有空语句的概念，就不能这样写了。现在有了空语句的概念，这样写可以理解为语句标号后 END 前还有一个空语句，就合法了。不过应注意， 这样一来前面的一个语句就不算是 END 前最后一个了，故其后的分号不可少。关于复合语句后面再详述。

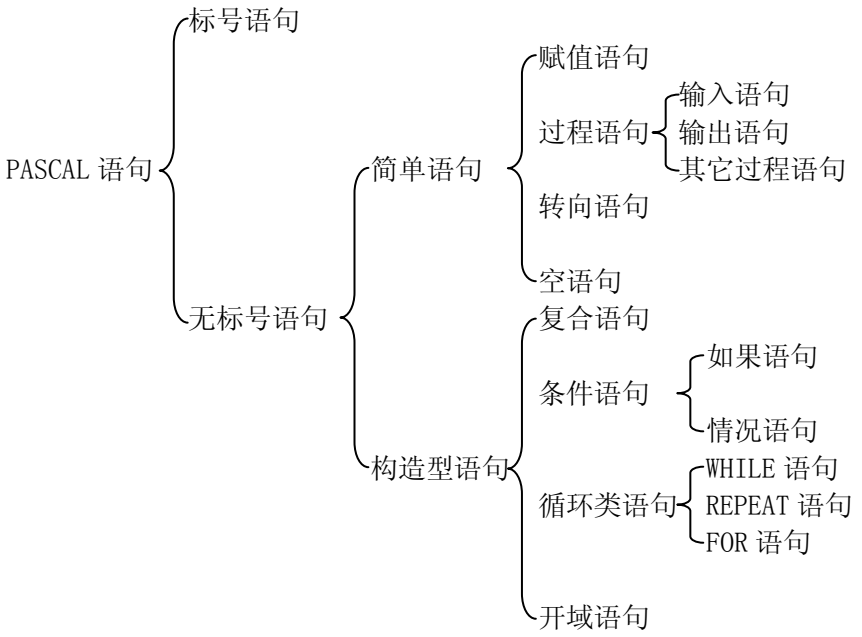


表 3.1 PASCAL 语句分类

顺便指出，采用基本结构编程，转向语句及标号很少用到。

3.2 赋值语句

赋值语句是最常用到的一种简单语句，前面的例子中已经出现过了。赋值语句的语法图见图 3.1。其相应的 BNF 描述可以参看第一章课文最后的例子。

从图中可以看到，赋值号:=左边可以是变量，还可以是函数名。但对函数名赋值的情况只在函数说明中用到(见后面第八章)，故目前讲到的赋值语句中赋值号左边只能是变量。例如前面 2.5 节的

x:=x*0.8+1

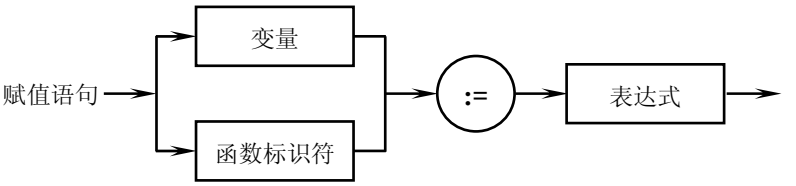


图 3.1 赋值语句的语法图

就是赋值语句。其中的 x 是变量，x*0.8+1 是表达式。

赋值语句的作用就是将该表达式所表记的值赋给这个变量。

变量在程序中具有两重属性：名和值。这两个属性分别对应于实际计算机存储器中的存储位置（地址）和该位置中存放的数据（内容）。同一个名字的变量不同时刻可有不同的值，实际上反映了一个存储单元中可先后存放不同的数据。

计算机存储器有这样的性质：每个单元中存放的数据可以多次读取，其值不变；而一旦存入新的内容，则原有内容就不存在了。

前面已说过（见 2.5.3），凡是写在表达式中作为运算数的变量（例如上述:= 号右边的表达式中的 x），都是其第二个属性——“值”起作用；而写在:=号左边的变量， 则是其第一个属性——“名”起作用，也就是其地址起作用，表示赋值时新值的存放位置。除此以外，PASCAL 语言中变量地址起作用之处还有过程和函数中的变量参数，这在后面再详述。

若赋值号两边都有同一个变量（如上例中两边都有 x）， 则右边表达式中的变量代表的是它在赋值语句执行前的值。

了解了这个道理，就可以明白赋值号:=和数学上=号的不同(虽然有时很相似)。如 N=N+1 在数学上看作方程是无解的，看作关系式是不成立的，而 PASCAL 中的 N:=N+1 则有明确的意义：“N 的原有值加 1 后作为 N 的新值。”

在 PASCAL 中若写 N=N+1 则可看作一个取值为 false 的布尔表达式（不是语句）。

从这个道理中也可以明白赋值号:=的左边只可以是单个的变量（除以后讲到的函数说明中确定返回值的语句外），而不可以是一般形式的表达式。如

```
x+y:=0.5
```

显然是错误的。因为“x+y”是没有“名”这个属性的，不存在一个专门存放“x+y”的地址，所以不能给它赋值。

PASCAL 中规定，赋值语句中赋值号右边表达式的值对于赋值号左边变量的类型必须“赋值相容”。关于什么叫赋值相容，本书后面还要严格叙述，现在可以暂时这样理解：“赋值相容”一般是说类型相同，但也允许表达式是整型而变量是实型。允许将整型数据赋给实型变量，但不允许将实型数据赋给整型变量，这与前面讲过的整型与实型的关系原则一致，实际上隐含了自动的类型转换。

假设变量说明部分是：

```
VAR
  x,y:     real;
  i,j,k:   integer;
  ch,ch1:  char;
  flag:    boolean;
```

则语句部分中以下语句是合法的:

```
x:=2.4;
y:=100;
i:=-66;
```

```
j:=round(x);
k:=i DIV j;
ch:='@';
ch1:=chr(-i);
flag:=(x>0)AND(x<5);
```

注意其中第二句将整数 100 赋给了 y，但 y 并不因此而变成整型，因为 100 在赋值前已经自动转换为实数了，y 仍是实型。

以下语句是不合法的:

```
ch:=j;
i:=100/2;
```

其中第二句虽然 100 及 2 都是整数，但我们已知道 / 运算的结果总是实型， 所以不可直接赋给 i。假如确实需要执行这样的操作，可以改写为

```
i:=round(100/2)
```

或

```
i:=100 DIV 2
```

PASCAL 中对变量的赋值，除用赋值语句进行外，还有其它手段，如后面将讲到的输入语句（read 语句）等。其赋值的内部机理与这里是一样的。

[例 1] 编写一个程序段，求出半径为 3.2 的圆的面积送入变量 s 中。

假设上文像第二章中的例子一样，已定义了符号常量 pi 代表圆周率π：

```
CONST
  pi=3.14159265;
```

则根据圆的面积公式 s=π*r2，这一段程序仅用以下一个语句即可：

```
s:=pi*sqr(3.2)
```

有时，我们需要将半径也保留下来以供其它地方引用，则应引入另一个变量来存放(如用 r 作为半径)。这样本程序段可如下写成两句:

```
r:=3.2;          {r 为半径，将 3.2 赋给实型变量 r}
s:=pi*sqr(r); {计算π*r2 的值赋给 s}
```

当然，程序上文的变量说明部分中必须定义好各个变量如 s,r 等。

[例 2] 编写一个程序段，交换两个变量 a 和 b 的值。

要交换两个变量的值，也就是说，要将 a,b 的原有值分别赋给 b,a。假如我们直接用两个赋值语句

```
a:=b;
b:=a
```

试图完成这个功能是不行的。这是因为在执行第一句把 b 的值赋给 a 时，a 原来的值就被冲掉了。因此，需要利用另一个同类型的变量作为暂存。如下，利用变量 t 作为暂存， 这一段程序可以写成三个赋值语句：

```
t:=a;
a:=b;
```

b:=t

先将 a 的值送入 t 中暂存，再将的 b 值送入 a 中，然后再将 t 中保留的 a 的原有值送入 b 中， 就完成了所要求的交换。

3.3 输出语句——写语句

在 3.2 节的[例 1]中，我们按要求编写了一个程序段，求半径为 3.2 的圆的面积， 但程序的运行结果如何，也就是说所求的面积到底是多少，用户并不知道。再如[例 2]， 程序段的功能是交换两个变量的值，但到底交换了没有，用户也看不到。输出语句则可以将程序的运行结果显示或者打印出来。

3.3.1 输出语句（写语句）

输出语句也叫写语句。在 PASCAL 语言中输出语句有两个，write 语句和 writeln 语句，它们可以将计算机内存中的数据输出到计算机的输出设备上。按本节讲的用法，其语法规则如图 3.1。

这个图只表示了本节讲的用法，是一个不完整的语法图，到第十一章还要讲到输出语句的其它用法，那里再给出完整的语法。

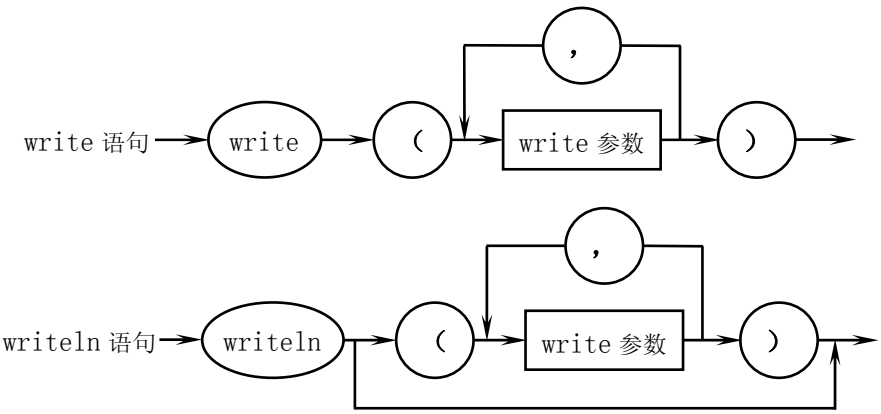


图 3.2 输出语句的语法图（不完整）

由输出语句的语法规则我们可以看出，一个输出语句可以有零个或多个 write 参数， 各 write 参数之间用逗号隔开。简单情况下 write 参数是一个表达式，它可以是一个常量、变量或者是一个更复杂的表达式。我们将这个表达式叫做输出项，它可以是整数类型、实数类型、字符类型、布尔类型或字符串类型，它的值就是我们要输出的数据。

关于“字符串类型”的一般概念在第七章讲到，这里先介绍字符串类型的字面常量，就称作“字符串”或“字符串常量”。字符串类型的数据就是由多个字符型的数据组成的一个串。与前面讲字符型字面常量时的道理一样，为了和程序中的其它内容区别，字符串类型的字面常量也不能简单地只写一串字符。规定：程序中字符串的写法是在要表示的一串字符前后各加一个撇号括起来。在这里大小写被看作是不同的字符。若要表示的串中包括撇号字符本身，则应将该撇号写成两个，即

'It''s a cat.'

才是代表以下句子的字符串常量：

```
It's a cat.
```

后面还会讲到，write 参数中在输出项后还可以带域宽，关于 write 参数的完整语法见下文。

例如

```
write('Hellow')
```

的输出结果为

```
Hellow
```

这里输出项为字符串常量，这个语句的功能等效于：

```
write('H','e','l','l','o','w')
```

又例如

```
write(1,2,3)
```

的输出结果为

```
123
```

这里输出项为三个整型常量（注：这里三个数紧密相连，是 Turbo Pascal 的格式，其它系统可能有所不同，见下文输出格式的叙述）。又例如，假定变量 x 的值为 5，变量 y 的值为 7，则

```
write(x,y)
```

的输出结果为

```
57
```

又例如，假定变量 x 的值为 3.1，则

```
write(sin(x))
```

的输出结果为

```
 4.1580662433E-02
```

这里输出项为表达式，其中引用了预定义函数。又例如，假定变量 x 的值为 5，变量 y 的值为 0，则

```
write(x*y=0)
```

的输出结果为

```
true
```

这里输出项为布尔表达式。又例如，假定变量 x 的值为 5，则

```
write(3*x)
```

的输出结果为

```
15
```

输出语句的功能是将输出项的值送往标准输出设备。什么是标准输出设备？如果我们编好的程序是在某操作系统(如 PC-DOS)下运行，则标准输出设备是由该操作系统指定的。一般标准输出设备默认的指定就是显示终端，但可以利用操作系统提供的输出重定向办法更改这个指定。这些不在本书的讨论范围，为简单起见，本书下文都按标准输出设备就是显示终端来叙述。程序首部中的 output 是代表标准输出设备（看作一种广义的文件）的预定义标识符。

输出语句除将输出项的值送往标准输出设备外，还可以输出到某个一般的文件上去，其用法将在“文件”那一章再详细介绍。本章只讲面向标准输出设备的输出，也就是说，显示在显示终端上。

对于上一节中的例子我们就可以加上输出语句，输出它们的运行结果，如：

[例 1]

```
PROGRAM area(input,output);
  CONST
    pi=3.14159265;
  VAR
    s,r:real;
BEGIN
  r:=3.2;          {r 为半径，将 3.2 赋给实型变量 r}
  s:=pi*sqr(r);    {先计算 pi*r*r 的值，再赋给 s}
  writeln(s:10:5)
END.
```

运行结果为

```
    32.16988
```

这里 writeln 语句及其中的域宽（:10:5）见下文详述。

3.3.2 write 语句和 writeln 语句的区别和联系

write 语句和 writeln 语句的功能大体相同，只是在输出格式上有差别。

一、write 语句将输出项一项一项地输出，执行完本 write 语句后，并不换行，后面若有下一个输出语句，则其中的输出项接着本行输出，即输出在同一行上。

例如，如果有 a,b,c,d 四个整型变量，它们的值分别为 1,2,3,4,若执行

```
write(a);
write(b,c);
write(d);
```

输出结果为

```
1234
```

可以看出，执行第一个 write 语句输出 a 的值后没有换行，继续执行后面的语句时在本行直接输出 b,c 以及 d 的值。

writeln 语句则是将本语句中的输出项一项一项地输出完后，接着输出一个换行的控制信号（行结束符），使显示换一行。于是再执行下一个输出语句时，从下一行的开头开始输出。

如果我们把上例中的 write 语句改为 writeln 语句，即为

```
writeln(a);
writeln(b,c);
writeln(d);
```

则输出结果为

```
 1
 23
 4
```

最后光标移到第四行的开头。

二、write 语句必须有参数而 writeln 语句可以没有参数，如:

```
writeln;
```

它的作用是输出一次换行。

假如我们将前例改动一下:

```
writeln(a);
writeln;
writeln(b,c);
```

则输出结果为:

```
1
        (空行)
23
```

以下语句是等价的。

```
write(a,b,c); writeln;     ———— writeln(a,b,c);
write(a,b);writeln(c);     ———— writeln(a,b,c);
write(a); write(b); write(c); —— write(a,b,c);
```

3.3.3 输出格式

在介绍输出格式之前，我们先介绍一个基本概念——域宽。域宽也称场宽，它规定一个数据所占的宽度，就是一个数据占几个字符的位置。

PASCAL 在输出时，整数按十进制数输出，实数则可以按照指数形式和小数形式两种形式输出。这实际上都是由若干个字符构成的。而对于布尔型，则输出为四个或五个字符（true 或 false）。

输出时，如果构成数据本身的实际字符数比规定的域宽小，那么数据向右靠齐，左边用空格补够宽度。

计算数据本身的实际宽度时，若为负数，应计入开头负号的位置。若为正数，则分两种情况：整数或小数形式的实数，不留符号位置；指数形式的实数，符号位置留一空格。

另外，下述的各种格式中，凡是实数，对小数位数都有规定。实数的小数位数，是在二进制向十进制转换时，按四舍五入的原则保留的。

PASCAL 的输出格式有两种，一种是标准输出格式，一种是用户自己定义的格式。前一种格式亦可称为缺省或默认格式。

输出语句中的 write 参数的语法规则见图 3.3。

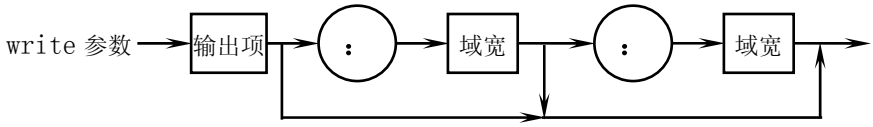


图 3.3　write 参数的语法图

图中“输出项”应是一个表达式，它可以是整数类型、实数

类型、字符类型、布尔类型或字符串类型。

图中“域宽”应是一个整数类型的表达式。

从图中可以看出，每个 write 参数可以仅是一个输出项， 也可以带有一个或两个域宽，每个域宽前面应有一个冒号。如果不带域宽，就按标准格式输出；如果带有域宽，就按自定义格式输出。前面我们举的许多例子中每个write参数仅是一个表达式(即输出项)，后面不带域宽，都是标准格式。

下面我们分别讨论这两种格式:

一、标准格式(缺省、默认格式)

各个 PASCAL 系统为整型量，实型量，布尔型量和字符串都规定了一个默认的域宽，我们把它叫做标准域宽，即标准输出格式所采用的域宽。PASCAL 的标准中只对字符和字符串的标准域宽作了规定，而其它类型的标准域宽则各实际 PASCAL 系统的规定有所不同，表 3.2 列出了几种典型的 PASCAL 系统所规定的标准域宽。

数据类型	标 准 域 宽				
	某种标准 PASCAL	VAX	Turbo Pascal	IBM PC	PDP-11
整型	10	10	实际宽度	14	13
实型	22	22	17	16	13
字符型	1	1	1	1	1
字符串	串长	串长	串长	串长	串长
布尔型	10	6	true 为 4 false 为 5	true 为 4 false 为 5	6

表 3.2 几种典型系统的标准域宽

实型数据的默认输出格式是指数形式。PASCAL 采用“标准科学表示法”形式输出实数，即：只要该数非零，则小数点前有且只有一位非零数字；如果系统采用两位阶码，则指数部分共占 4 位，其中包括一个'E'（有的系统是'e'），一个阶符号位以及两位阶码数。 另外数字之前有一个符号位（该实数为正数时，符号位用空格表示），还有小数点也占一位。剩下的位置就是有效数字。所以有效数字的位数为

域宽-6

有的系统的指数部分包含三位阶码，而有的系统则包含两位阶码，所以，一般说来标准科学表示法中

有效数字位数=域宽-4-阶码位数

注意标准科学表示法中阶码位数是固定的，如果实际指数不够位数，其前要补 0 ，另外阶符号位总是'+'或'-'，正号不省略。

机内的实数是二进制，输出时化为十进制时由计算机自动四舍五入保留上述的有效数字位数。

例如，若 a 和 b 都是整数，a 的值为 35，b 的值为-10，执行

```
writeln(a,b);
writeln(sqrt(a));
wrieln('sqrt(a)+b=',sqrt(a)+b);
```

显示结果为:

35-10

5.9160797831E+00

sqrt(a)+b=-4.04839202169E+00

注意这里后两行的表达式都是实型。

从表中可以看到，目前常用的 Turbo PASCAL 系统中标准域宽总是恰等于实际宽度，所以输出时同一行前后数据常常都连起来了。前面的例子都是这样的。如果想使其分开，最好采用下面讲的用户自定义格式。

二、用户自定义格式

用户自定义格式有两种，一种为单域宽，另一种为双域宽。

(一) 单域宽

单域宽就是在 write 参数中输出项后只带一个域宽。它规定一个数据所占的总宽度。

输出时，如果构成数据本身的实际字符数比规定的域宽小，那么数据向右靠齐，左边用空格补够宽度。如果数据本身的字符数比规定的域宽大，那么又分两种情况：如果是数值型数据，则按数据本身的字符数输出，实际宽度会超过规定的域宽；如果是字符串，则将最后多出的字符略掉，保持实际输出宽度等于规定的域宽。

例如，设 a 为整数，其值为 15：

write('sqr(':6,a:3,')=':3,sqr(a):2)

该输出语句中共有四个参数：

'sqr(':6

a:3

')=':3

sqr(a):2

第一个输出项为一字符串，它本身有 4 个字符，给它指定的域宽为 6，所以字符串向右靠齐，左边补两个空格，共占六个字符的宽度；第二项为整数，第三项为字符串它们的实际宽度都比指定域宽小，故向右靠齐，左边用空格补齐；最后一项为一表达式，其值的实际宽度为 3，而指定宽度为 2，小于实际域宽，故按实际域宽输出，所以本语句的输出结果为

sqr(15)=225

实数的单域宽格式仍为指数形式。例如：a 为实数，其值为 2.2，执行

write(a*a:10)

输出结果为

4.840E+00

由此例看出，对于实数，数字部分有一个小数点和一个符号位（若为正数，符号位为一空格)。其有效数字位数仍然由前述公式决定：

有效数字位数=域宽-4-阶码位数

所以对上例来说，共有 4 位有效数字。若域宽不同，保留的小数位数就不同。 应注意实数的机内表示往往带有误差，精度有限，所以若规定的域宽太大，输出的最后某些位不一

定有意义。

指数形式输出时小数点前是一位数字，而小数点后必须保证至少有一位小数，所以当指定域宽过小时，则按两位有效数字（小数点前后各一位）输出，例如上例中，若将域宽改为 4，执行

```
write(a*a:4)
```

则输出结果为

```
 4.8E+00
```

（二）双域宽

对于实数，按指数形式输出很不直观，而用小数形式输出就比较直观了。双域宽可将实数按小数形式输出。

双域宽就是在 write 参数中输出项后带两个域宽。 第一个域宽规定一个数据所占的总宽度，第二个域宽规定小数部分保留的位数（四舍五入）。

例如，我们将上例改为

```
write(a*a:10:5)
```

则输出结果为:

```
  4.84000
```

第一个域宽作用与前述的单域宽类似，不同的是：若域宽加大，单域宽时对实数可以增加小数位数；而双域宽时则由于小数位数已由第二个域宽定死，构成数据的实际字符数已确定，如果它比规定的第一个域宽小，就要在前面加空格。

计算构成数据的实际字符数时，除了整数位数、小数位数和一个小数点外，若是负数还要有负号的位置。正数不留符号位，这点和按指数形式输出时不同。

如果第一个域宽过小，那么，输出数据时仍服从第二个域宽的规定，实际宽度将突破第一个域宽的规定。如

```
write(a*a:2:1)
```

输出结果为:

```
4.8
```

一般地，在同一程序中，采用指定域宽，可以使输出上下行右对齐，比较美观，也便于阅读 。例如：对实型数采用“:8:2”的双域宽

```
writeln(a:8:2,b:8:2);
writeln(c:8:2,d:8:2);
```

则输出格式如下

```
  357.21    8.30
   21.01    7.77
```

实例中，域宽大都直接写常数，但实际上应该知道，不仅常数，任意复杂的表达式在这里都是允许的（只要其值是整型）。如果域宽中带有变量，其值按一定规律变化，可以使输出构成某种图案。这种例子以后会见到（第五章）。

3.4　输入语句——读语句

在前面，我们用赋值语句给变量赋一个确定的值。但是，仅用赋值语句是不够的。比方说，如果我们每次使用程序时需要给变量不同的值，就必须修改程序，在程序运行前还要重新编译，这就很不方便。因此，PASCAL 中引入了输入语句来解决这些问题。

输入语句也叫读语句，它有两种：read 语句和 readln 语句。输入语句可将数据通过计算机外部设备输入到内存中，供程序使用，它与赋值语句比较有两个不同点：

一、使用输入语句，在编写程序时不需要确定赋给变量的值，而是在程序运行过程中才从键盘输入变量的值，因此需要输入不同的值时不必修改程序。

二、一个赋值语句只能给一个变量赋值，而同一个输入语句可以给多个变量赋值。例如

```
read(a,b,c)
```

执行该语句时，可以从键盘输入三个数据依次赋给 a、b、c。

上一节[例 1]中的程序，只能求出半径为 3.2 的圆的面积，若要求其它圆的面积，则需要修改给 r 赋值的语句。如果我们把这一语句换成 read(r)，那么每次运行时，只要输入不同的值，就可求出不同的圆的面积，如

```
  ......
BEGIN
  read(r);
  s:=pi*sqr(r);
  writeln(s:10:5)
END.
```

运行时输入

```
5.1
```

运行结果为

```
  81.71276
```

另一次运行，输入

```
2.4
```

运行结果为:

```
  18.09556
```

3.4.1 输入语句（读语句）

输入语句的作用是将数据从外部设备输入到内存中，赋给变量。输入语句可以从标准输入设备读入数据。与标准输出设备相同，标准输入设备也是由操作系统指定的，一般标准输入设备的默认指定是键盘,程序首部中的 input 是代表标准输入设备的预定义标识符。除了从标准输入设备读数据外，输入语句也可以从文件中读数据。本章主要介绍从键盘输入数据。

输入语句包括 read 语句和 readln 语句，按本节讲的用法，它们的语法规则如图 3.4 所示。

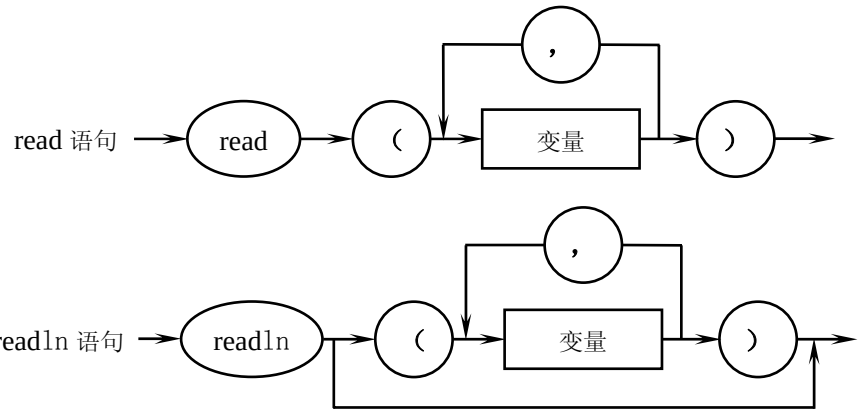


图 3.4　输入语句的语法图（不完整）

这个图只表示了本节讲的用法，所以不算是完整的语法，第十一章还会讲到其它用法。

从输入语句的语法规则可以看出，一个输入语句中可以有零个或若干个变量，这些变量可以是整型、实型、字符型，但不能是布尔型。

在执行输入语句时，用户从键盘输入数据，输入语句一个一个地读数据，并一一赋给与之对应的变量。这里，需要注意几点:

一、要满足赋值相容的原则，即要求输入的数据要和与它对应的变量类型一致，但对于实型变量可输入整型数据，系统先自动将整型数据转换为实型数据，然后再赋给实型变量。

例如，a 为字符型，b 为整型，c 为实型，执行

```
read(a，b，c);
```

则可输入

```
A 2 7.0
```

也可输入

```
A 2 7
```

二、输入多个数据时，同一行中的各个数值型数据之间用一个或多个空格隔开，在有些系统中，也可以用逗号来隔开。但必须注意，输入字符型数据时不能用空格分隔，因为系统将空格也当作一个字符来处理。

三、每次输入以一个数据行为单位。一个数据行可以包含一个或多个数据，以“回车键”表示本行的结束。

四、输入数据个数必须大于等于变量个数。如果输入数据不够，则程序停下来等待继续输入，如果输入数据过多，则多余数据的作用见下文详述。

例如，执行

```
read(a,b,c);
```

其中，a、b、c 均为整型数据，

(1)若输入

```
3    7
```

则将 3 赋给 a，7 赋给 b，然后等待继续输入 c 值。再输入

5

则将 5 赋给 c。

(2)若输入

3 4 5 6

则 a 值为 3，b 值为 4，c 值为 5，数据 6 多余。

[例 1] 已知三角形的两边 a、b 及其夹角 α，求第三边 c 及其面积 s。

问题分析:根据三角形公式

$c = sqrt(a^2 + b^2 - 2ab\cos\alpha)$

$s = ab\sin\alpha/2$

只要输入 a，b 和 α 的值，就可以计算出第三边 c 的值和面积 s 的值。若 α 以角度值输入，在计算时应将它转换为弧度。角度与弧度之间的关系为

弧度=角度*π/180

α 不是合法的标识符，改用 alpha 代替。变量 a,b,c,alpha,s 都说明为实型变量。

程序如下:

```
PROGRAM triangle(input,output);
CONST
  pi=3.14159265;
VAR
  a,b,c,alpha,s:real;
BEGIN
  read(a,b,alpha);
  writeln('a=',a,'b=',b,'alpha=',alpha);
  alpha:=alpha*pi/180;
  c:=sqrt(a*a+b*b-2*a*b*cos(alpha));
  s:=0.5*a*b*sin(alpha);
  writeln('c=',c,'s=',s)
END.
```

3.4.2 read 语句和 readln 语句的区别和联系

read 语句和 readln 语句在功能上基本上是相似的，但它们也有一些区别。

一、read 语句是从输入的一行数据中一个接一个地读数据，执行完本 read 语句后，下一个输入语句接着从本数据行读数据，而不换到下一行。只有本数据行已无数据可读时，才要等再次输入一行后再从中读取。

例如，执行

```
read(a,b);
read(c,d);
read(e);
```

若输入数据为

1　3　5　7　9　11

则先执行第一个 read 语句，将 1，3 分别赋给 a 和 b，然后执行第二个输入语句，接着从本数据行第三个位置读数据，将 5 和 7 分别赋给 c 和 d，然后再执行第三个输入语句，将第五个位置的数据 9 赋给 e，剩余的一个 11 保留，若下文再没有输入语句，或尚未读到它时，又遇到

```
readln;
```

则它作为多余的数据没有用上，这是允许的。

readln 语句则是一个接一个地读数据，执行完本 readln 语句后，也就是说读完本 readln 语句中变量所需的数据后，就略过本行多余的数据以及行结束符，这些略过的数据变得无用，下一个输入语句则从下次输入的一个数据行的开始位置接着读数据。

例如，执行

```
readln(a,b,c);
readln(d,e);
readln(f);
```

若输入数据为三行

```
1　2　3　4　5
6　7　8
9　10　11
```

则执行第一个 readln 语句时，将 1、2、3 分别赋给 a、b、c，将 4、5 两个数据略过不用；执行第二个 readln 语句时，从第二个数据行的开始位置读起，将 6 和 7 分别赋给 d 和 e；执行第三个 readln 语句时，从第三个数据行的开始位置读起，将 9 赋给 f，而每一行中多余的数据都被略过而变得无用。

二、从输入语句的语法规则我们可以看出，read 语句必须有参数，而 readln 语句可以没有参数，如果 readln 语句没有参数，即为

```
readln;
```

此时，其作用仅仅是略过本行多余的数据以及行结束符，若再有输入语句，则从下一行的开始位置读起。

例如，有三个输入语句

```
readln(a,b);
readln;
readln(c,d);
```

输入三个数据行

```
1　3　7
2　4　4
1　1　6
```

则执行第一个语句时，将 1、3 分别赋给 a、b，将 7 略过； 执行第二个语句时将第二行数据全部略过；执行第三个语句时，则从第三行的开始位置读起，将 1、1 分别赋给 c 和 d，将 6 略过。

三、如果一个数据行中的数据不足以提供给一个输入语句中的变量，则接着从下一个数据行读起，这一点对 read 和 readln 都适用。

例如

```
read(a,b,c,d);
readln(e,f);
readln(g)
```

若输入为

```
1   2   3
4   5
6   7
8   9
```

则 a、b、c、d、e、f、g 的值依次为 1、2、3、4、5、6、8。

四、如果程序中只有一个输入语句，则使用 read 与 readln 是等价的。

以下语句是等价的

```
read(a);read(b);read(c); ——— read(a,b,c);
read(a,b,c);readln;      ——— readln(a,b,c);
```

3.4.3 输入语句的内部实际实现过程

上文说明 read 语句和 readln 语句的功能，都是从宏观角度叙述的。而实际的内部实现，在大多数计算机系统中还要复杂些。所以，有时执行某些程序时会发现时间先后次序与想象不一样。一般情况下，上文所说的读数据，并不是直接读键盘，而是扫描文本输入缓冲区。键盘输入的数据，在一行未完时并不进入文本输入缓冲区，只有在打入回车键时才整行地送入该缓冲区，而在打回车键之前允许用 Backspace 键来修改。 上文所说的“略过本行多余的数据以及行结束符”实际上就是将文本输入缓存区内尚未读到的数据全部作废，使缓存为空。在文本输入缓存区内已空而又要执行新的输入操作时，程序停下来等待下一个回车键。

例如

```
read(x);writeln(x);
read(x);writeln(x);
read(x);writeln(x);
```

执行时若输入

```
10  20  30
```

则输出

```
10
20
30
```

有人会奇怪：程序中是输出第一个数后才输入第二个数，输出第二个数后才输入第三个数，……，而执行时却是三个数输入完才依次输出。这是因为执行时的键盘操作是将数据行送入文本输入缓冲区，而程序中的“输入”操作则是从文本输入缓冲区读取数据送给变

量，二者时间上有所不同。

3.4.4 输入语句和输出语句的连用

在执行输入语句时，系统不会自动提示任何信息。因此，运行程序时，往往会出现这样的一些情况：用户等待计算机执行程序，而计算机又在等待用户输入数据；或者，在要求输入数据较多的时候，用户常常搞不清楚应该输入什么样的数据。为了解决这些问题，可以在每个输入语句之前加一个输出语句，输出一些提示信息。例如：

```
writeln('Input r:');
read(r);
```

程序运行时，先显示一行信息：

```
Input r:

_
```

然后光标转到下一行，等待用户输入数据，读完所需的数据后，才继续运行程序。如果使用的不是 writeln 语句而是 write 语句，则不换行，光标在提示信息的后面，如：

```
Input r:_
```

现在大多数的 PASCAL 系统允许在字符串中使用汉字。如果在汉字编辑环境中编写这一段程序，可以为

```
write('请输入 r:');
read(r);
```

这样，程序在可以显示汉字的操作系统下运行时，提示信息为

```
请输入 r:_
```

另外，在过去的某些机器系统中，操作者输入的数据当时自己是看不到的。为了使操作者确认自己没打错，程序中可以在 read 语句后用 write 语句将刚输入的数据再输出一遍供操作者检查。这种做法称作“回打”，早期的程序设计常编入回打操作，但现在的系统中这样做已没有必要，所以本书的例题中大多没有回打操作。

习题

3.1 写出下列输出语句的显示结果（空格位置请用∪记号标出）：

(1) writeln('x=', 0.2368*100:6:1, '%')

(2) writeln('2>1':6, 2>1:6, '1>2':6, 1>2:6)

(3) writeln('2/3':10, 2/3:6:3)

3.2 编程：键盘输入三个边长，计算机求出三角形的面积 S 和周长 L。面积可以利用以下公式：

$$S=\sqrt{p(p-a)(p-b)(p-c)}$$

这里 p=(a+b+c)/2

3.3 编程：将键盘输入的华氏温度转换为摄式温度，公式为

C=(F-32)·5/9

第四章　逻辑判断及选择结构程序设计

PASCAL 语言是结构化的语言，可以很方便地编写出结构化的程序。本章和下一章分别介绍选择结构和循环结构。在选择结构和循环结构中，常常要用到复合语句，下面我们先介绍一下复合语句。另外，在选择结构和循环结构中，还要用到逻辑判断和逻辑运算，这些也是本章的内容。

4.1　复合语句

复合语句只是把若干个语句合成为一个语句而已。其语法规则如图 4.1 所示。其中的语句序列组成顺序结构。按照书写的顺序依次执行其中各个语句的操作，其总的效果就是整个复合语句的功能。

从语法规则可以看到，复合语句是由 BEGIN 和 END 之间包含一个或多个语句（语句间由分号分隔）构成的。BEGIN 和 END 之间的这些语句，可以是任何一种语句，如简单语句，复合语句，或其它构造型语句。当复合语句中包含有复合语句时，要注意 BEGIN 和 END 的数目要保持一致，如:

```
BEGIN
  ......
  BEGIN
    语句序列
  END
  ......
END
```

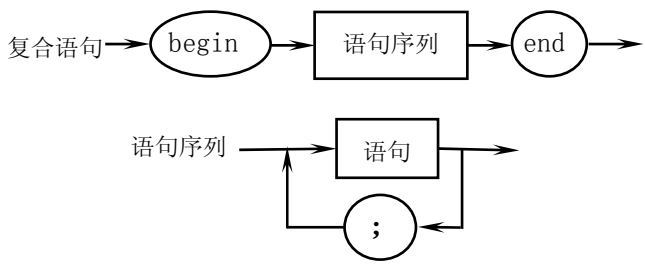


图 4.1　复合语句的语法图

例如

```
BEGIN
  a:=b;
  b:=c;
  c:=a
END;
```

这个复合语句由三个赋值语句构成，它的功能是交换 b 和 c 的值，这里，我们需要注意的是，最后一个语句与 END 之间没有分号，因为 END 不是语句，但如果加了分号，也不算错，因为系统会认为该分号后还有一个空语句，仍然合乎语法。例如

```
BEGIN
  a:=b;
  b:=c;
  c:=a;
END;
```

是由三个赋值语句和一个空语句组成的复合语句。

PASCAL 中引入复合语句，主要是构造其它构造型语句的需要。后面我们会看到，某些构造型语句的内嵌语句，按语法要求应是一个语句，如 IF 语句中的 THEN 子句和 ELSE 子句，以及 WHILE 语句和 FOR 语句中的循环体等。如果我们希望在该处编入多条操作，写成多条语句是不合法的。这时就可以用 BEGIN 和 END 将多条语句括成复合语句，复合语句在语法上算作一个语句，就合法了。

当然，有时为了分清概念层次，将语句序列中的某一段用 BEGIN 和 END 括起来，便于阅读理解，也是可以的。

4.2 逻辑判断和逻辑运算

电子计算机和以前传统的计算工具的重要区别之一就是电子计算机可以自动进行逻辑判断，并按照判断的结果来自动控制操作的流程。有了这种能力，才有可能自动进行复杂的运算。

第一章介绍过各种程序结构，这些结构中的“条件”指的就是逻辑判断。

第二章中我们已学过：布尔类型的数据表示一个逻辑判断的结果。换句话说，在程序中一个条件就应该写作一个取值为布尔类型的表达式。所以，一个布尔表达式相当于自然语言中的一个判断句（即陈述句）。

下面我们对学过的有关知识作一扼要回顾并介绍 PASCAL 中的几个逻辑运算。

4.2.1 布尔类型的数据

布尔类型数据的取值只有两个：“假”和“真”，分别用 false 和 true 表示。

布尔类型的类型名是 boolean。

布尔类型是一种顺序类型，PASCAL 将 false 的序号规定为 0，将 true 的序号规定为 1，因此有：

```
ord(false)=0;
ord(true)=1;
pred(true)=false;
succ(false)=true;
```

注意，false 没有前启，而 true 没有后继。

可以用赋值语句将布尔型数据赋给一个布尔型变量，但不能用输入语句从标准输入设备输入布尔数据赋给变量。例如，若 b 为布尔变量，那么

```
b:=3>4
```

合法，而

```
read(b)
```

则不合法。布尔型数据不能用输入语句来输入，但可以用输出语句来输出。如执行

```
writeln(3>4)
```

则输出结果为

```
false
```

4.2.2 关系运算和简单布尔函数

PASCAL 中提供了 7 种关系运算：=、<、>、<>、<=、>=、in。

关系运算的结果为布尔型的值，所以关系表达式属于布尔表达式。

前六种关系运算符可用于各种顺序类型数据的比较，也可以用于实数。一般地，这六种关系运算符两侧应该使用类型相同的数据，但整型量可以和实型量混合比较。

结果为布尔型的函数中，已学过的有 odd，其自变量是整数， 作用是判断该整数是否奇数。

4.2.3 逻辑运算

PASCAL 中对布尔型数据还提供了三种逻辑运算：

AND （与）

OR （或）

NOT （非）

这三种逻辑运算中要求参与运算的操作数都是布尔类型的数据，运算结果也是布尔型数据。其中，NOT 是一元运算，AND 和 OR 都是二元运算。

AND 运算中只有在参与运算的两个操作数都是“真”时运算结果才为“真”， 否则就为“假”。AND 运算的结果相当于自然语言中用“并且”一词将两个判断句连成的一个复合判断句。如

(x>0) AND (y>0)

意为

x>0 并且 y>0

只有在 x>0 和 y>0 都成立时整个判断句才算成立。

OR 运算中只要参与运算的两个操作数有一个“真”时运算结果就为“真”，只有在参与运算的两个操作数都是“假”时运算结果才为“假”。OR 运算的结果相当于自然语言中用“或者”一词将两个判断句连成的一个复合判断句。如

(x>0) OR (y>0)

意为

x>0 或者 y>0

只要 x>0 和 y>0 有一个成立整个判断句就算成立。

NOT 运算中，参与运算的操作数为“假”时运算结果就为“真”， 参与运算的操作数为“真”时运算结果为“假”。NOT 运算又称作“取反”。NOT 运算的结果是操作数所代表的判断句的否定判断句。如

NOT (x>0)

意为

x 不大于 0

当 x 小于或等于 0 时整个判断句成立。

可以用表 4.1 来表示逻辑运算的规则。

a	b	NOT a	a AND b	a OR b
true	true	false	true	true
true	false	false	false	true
false	true	true	false	true
false	false	true	false	false

表 4.1 逻辑运算

表达式中这三个逻辑运算的优先次序是：NOT 优先级最高，AND 次之，OR 优先级最低。若要改变优先次序可以使用圆括号。

逻辑运算有许多规律，可以用来化简复杂的布尔表达式，这可以成为一门专门的学问。本书不可能全面介绍这方面的知识和技巧，只能介绍最常用的几个规律。

一、双否定律

设 a 是一布尔表达式，则以下关系必成立：

NOT (NOT a) = a

二、等幂律

设 a 是一布尔表达式，则以下关系必成立：

a AND a = a

a OR a = a

三、交换律

设 a、b 都是布尔表达式，则以下关系必成立：

a AND b = b AND a

a OR b = b OR a

四、结合律

设 a、b 和 c 都是布尔表达式，则以下关系必成立：

(a AND b) AND c = a AND (b AND c)

(a OR b) OR c = a OR (b OR c)

五、分配律

设 a、b 和 c 都是布尔表达式，则以下关系必成立：

a AND (b OR c) = (a AND b) OR (a AND c)

a OR (b AND c) = (a OR b) AND (a OR c)

这两个分配律分别称作“与对或的分配律”及“或对与的分配律”，和算术中的“乘法对加法的分配律”相似。但要注意，算术中加法对乘法并不存在分配律，而这里“与”、“或”之间却有两个分配律。

六、吸收律

设 a、b 都是布尔表达式，而且已知由 a 可以推导出 b，则

a AND b 可以简化为 a

a OR b 可以简化为 b

例如 a 表示“x>5”，b 表示“x>3”， 此时由 a 可以推导出 b。显然“x>5 并且 x>3”一句话中后半句是废话，可以只说“x>5”。而“x>5 或者 x>3”一句话中前半句是废话，可以只说“x>3”。

七、德·摩根（De Morgan）律

设 a、b 都是布尔表达式，则以下关系必成立：

NOT (a AND b) = (NOT a) OR (NOT b)

NOT (a OR b) = (NOT a) AND (NOT b)

上述这些规律都比较简单，读者凭直观的想象大都可以明白它的道理。但其中的最后

一个德·摩根律，对于某些缺少逻辑思维能力的人来说却常常容易弄错。这个规律形式上有点像分配律，但括号展开后 AND 换成了 OR，OR 换成了 AND。德·摩根律可简述为“取反后与、或互换”。举一个实际例子：设 a 表示 x>0，b 表示 y>0，则 a AND b 就是

(x>0) AND (y>0)

即“x 和 y 都大于 0”的意思。如果要否定这一句话，只要 x 和 y 有一个不大于 0 就行了，即“x 和 y 不都大于 0”，并不要求二者同时不大于 0。也就是说 NOT (a AND b)应该相当于“x 不大于 0 或者 y 不大于 0”，即

(NOT (x>0)) OR (NOT (y>0))

假如写成

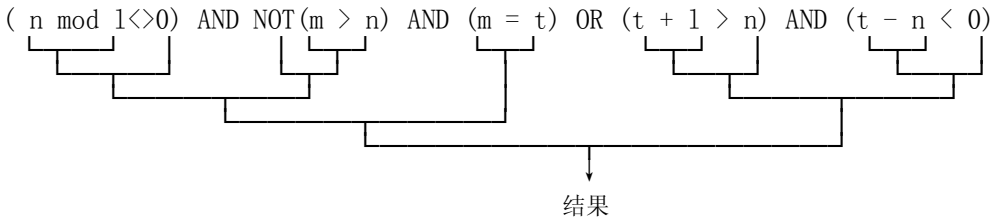
(NOT (x>0)) AND (NOT (y>0))

那就成了“x 和 y 都不大于 0”，就错了。 如果要被否定的式子本身包含多层逻辑运算的话，用德·摩根律反复展开括号，会发现每层中的与、或都要互换。这里不详细讲解，请读者自己练习。

4.2.4 带有逻辑运算的一般表达式

第二章中已讲过：在一个表达式中，AND 与乘除类运算优先级相同，OR 与加减类运算优先级相同，NOT 的优先级更高，关系运算的优先级更低。

例如，下面表达式的计算次序如框线所示。



关于带有逻辑运算的表达式，有一些初学者常犯的错误，需要引起注意。

一种错误是没注意优先级，少了括号。如

(x>0)AND(y>0)

若缺了括号，写成

x>0 AND y>0

按照运算符的优先级，就会先进行

0 AND y

的运算，计算机会报告“类型错误”。因为 AND 只能对布尔型运算，而 0 和 y 都不是布尔型。这一点，因为各种高级语言对优先级的规定有所不同，所以不仅初学者，有些熟悉其它高级语言的程序员也常疏忽。

还有数学上常用 0<x<1 表示“0<x 并且 x<1”的意思，写在 PASCAL 程序中应该是

(0<x) AND (x<1)

如果直接写成

0<x<1

机器就会按照从左到右的原则先作 0<x，得到一个布尔型的结果再拿来和 1 进行<比较。因为 1 不是布尔型，所以也会报告“类型错误”。

还有，自然语言中说“x 等于 1 或 2”，实际是“x=1 或者 x=2”的简略，写在 PASCAL 程序中应该是

(x=1) OR (x=2)

如果写成

x=(1 OR 2)

也是错的，也会报告“类型错误”。

还有，数学上常有“所有的 x>0”、“存在一个 x>0”等判断，这种判断在程序中必须通过一个复杂的算法才能实现，不能只写一个 x>0。因为变量 x 写在表达式中时只能代表它当时所具有的一个值，不可能同时代表所有取值。

4.3 如果语句（IF 语句）

4.3.1 如果语句的基本概念

如果语句又称作 IF 语句（因为它以字符号 IF 开头），是一种构造型语句。它可以用来描述第一章介绍的选择结构。IF 语句的语法规则如下：

图 4.2 如果语句的语法图

由语法图我们可以看出，IF 语句有两种形式。

(1) IF 布尔表达式 THEN 语句

(2) IF 布尔表达式 THEN 语句 ELSE 语句

前面我们已经说过，构造型语句组成部分中又包含有语句。这里我们可以看到，IF 语句的第一种形式里包含有一个语句，第二种形式里包含有两个语句。这些内嵌的语句可以是任何一种语句，它可以是一个简单语句，也可以是一个复合语句或其它构造型语句。我们把 THEN 后面的“语句”称为 THEN 子句，把 ELSE 后面的“语句”称为 ELSE 子句。

第一种形式的 IF 语句可实现第一章图 1.4 的选择结构。其作用是：首先求布尔表达式的值，如果值为真，那么执行 THEN 子句；如果布尔表达式的值为假，那么就什么操作也不执行。

第二种形式的 IF 语句可实现第一章图 1.3 的选择结构。其作用是：首先求布尔表达式的值，若值为真，则执行 THEN 子句；若布尔表达式的值为假，则执行 ELSE 子句。

显然，第二种形式中如果 ELSE 子句是空语句，就和第一种形式等效。

例如，设变量 score 是分数：

IF score<60

THEN write('不及格')
ELSE write('及格');

在这一语句中，首先对学生成绩是否低于 60 作出判断，若满足这一条件，就显示“不及格”字样，否则就是“及格”。其流程图如图 4.3 所示。

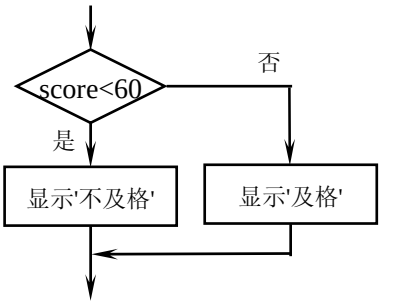


图 4.3

画成结构框图则如图 4.4。

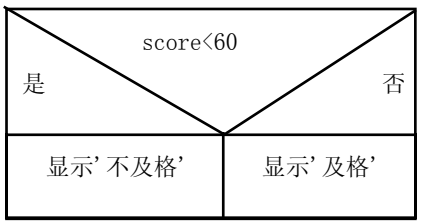


图 4.4

注意，构造型语句虽然是复杂的语句，但仍是一个语句，所以中间不可乱加分号。分号是在语句序列内分隔各个语句用的，如 4.1 节例子中 BEGIN 和 END 间四个语句间要用三个分号，三个语句间要用两个分号，其它地方不得乱用， 初学者一定要注意。 假如在第一种形式的 THEN 前加了分号，就形成了下面的格式：

IF 布尔表达式 ； THEN 语句

会被看成了两个语句，第一个语句只有 IF 没有 THEN 是不合法的，而第二个以 THEN 开头也不合法，在编译时，就会出现错误。而在 THEN 后加分号，即：

IF 布尔表达式 THEN ； 语句

一个语句就变成了两个语句，相当于一个空语句作为 THEN 的子句，无论布尔表达式是否为真，IF 语句都等效于一个空语句，而其后的“语句”作为 IF 语句以外的后一个语句，无论布尔表达式是真是假，都必然被执行，这样虽然编译能通过，但功能与我们的原意完全不同了。

在第二种形式中，如果在 ELSE 前误加了分号，构成如下语句形式：

IF 布尔表达式 THEN 语句 ； ELSE 语句

相当于一个第一种形式的 IF 语句之后跟了一个 ELSE 开头的语句。ELSE 开头的语句是不合法的，所以在编译时，会报告错误。

[例 1] 下面我们看一个简单的例子，找出 a 和 b 中的较大者，将其值赋给变量 max。 可以用第一种形式的 IF 语句表达如下：

IF a>=b THEN max:=a;
IF a<b THEN max:=b;

结构框图如图 4.5 所示。

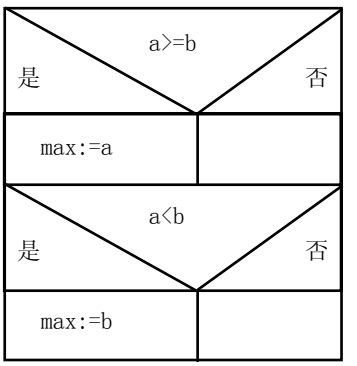


图 4.5 [例 1]的第一种做法

也可以用第二种形式的 IF 语句来表示：

IF a>=b
THEN max:=a
ELSE max:=b;

结构框图如图 4.6 所示。

还有一种做法：先无条件地赋值，再用第一种形式的 IF 语句有条件地重新赋值：

```
max:=a;
IF a<b THEN max:=b;
```

结构框图如图 4.7。

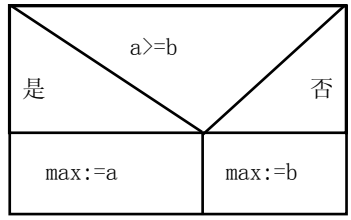


图 4.6 [例 1]的第二种做法

这三种做法完成了同样的功能。可以看出，同一问题可以有多种做法，其流程的简繁是不同的。具体这一问题中，第二个做法最简，但其它做法的思路也有可取之处。如后面的例子中我们会看到，在多个变量中选取最大值时，只有采用“重复赋值”的做法才能编出简短的程序。

这三种做法中第一种相对来说最不好，因为 a>=b 和 a<b 这两个条件互为否定， 本来一次比较就足以区分了，这里编成两次比较，有点多此一举。

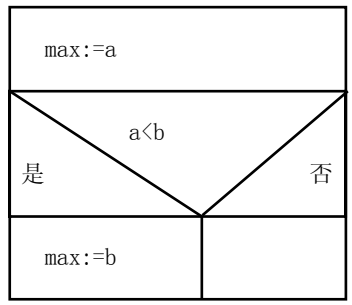


图 4.7 [例 1]的第三种做法

4.3.2 IF 语句内包含复合语句

前面说过,IF 语句中的 THEN 子句和 ELSE 子句也可以是复合语句。如，第二章 2.2 节例中有这样一段

```
IF  d<0
  THEN
    BEGIN
      r:=-b/2/a;                { 实部(-b)/(2a)→r }
      p:=sqrt(abs(d))/2/a;      { 虚部√|d|/(2a)→p }
      writeln(r,'+',p,'i');     { 输出二虚根 }
      writeln(r,'-',p,'i')
    END
  ELSE
    BEGIN
      x1:=(-b+sqrt(d))/2/a;     { (-b+√d)/(2a)→x1}
      x2:=(-b-sqrt(d))/2/a;     { (-b-√d)/(2a)→x2}
      write(x1,x2)              { 输出二实根 }
    END
```

这也是一个 IF 语句，其中的 THEN 子句和 ELSE 子句都是复合语句。

[例 2] 输入两个数，并按照从大到小的次序输出。

问题分析：这个题目有两种做法。一种做法是:读入两个数，分别赋给 m 和 n，如果 m 的值小于 n，则交换 m 与 n 的值，否则不交换，然后再统一输出。 另一种方法是：根据大小决定其输出次序，先输出较大的数再输出较小的数。第二种做法的程序留给读者自己考虑。第一种做法程序如下：

```
PROGRAM jh(input,output);
VAR
```

```
    m,n,t:real;
BEGIN
  write('Input m and n:');
  readln(m,n);
  IF m<n THEN
    BEGIN
      t:=m;
      m:=n;
      n:=t
    END;
  writeln(m:6:2,n:6:2)
END.
```

初学者应特别注意，在这种情况下，切不可把复合语句的 BEGIN 和 END 忽略掉。有时候丢掉 BEGIN 和 END 后形成语法错误，也有时候语法不错，但功能已经不对了。例如，上面例子中的 IF 语句假如去掉 BEGIN 和 END，如下所示：

```
  IF m<n THEN
      t:=m;
      m:=n;
      n:=t
```

这样 m:=n 和 n:=t 就成了 IF 语句以外的两个独立的语句了，于是当 m>=n 时，仍会执行 m:=n 和 n:=t 两个赋值语句，就会引起错误。

4.3.3 IF 语句的嵌套

IF 语句的嵌套指的是一个 IF 语句的内嵌语句又是 IF 语句。内层的 IF 语句可以嵌套在外层的 IF 语句的 THEN 子句中，也可以嵌套在 ELSE 子句中。

[例 3] 读入 x 的值，计算下面分段函数。

$$y=\begin{cases} 0 & ,\ x<10 \\ x & ,\ 10<=x<20 \\ 10 & ,\ 20<=x<30 \\ -0.5x+20 & ,\ x>=30 \end{cases}$$

若直接按题目的描述，这个问题可以编写程序如下，要注意，数学上如 20<=x<30 的习惯写法在程序中应该写成(x>=20) AND (x<30)。

```
PROGRAM hs(input,output);
  VAR
    x,y:real;
BEGIN
  write('Input x:');
  readln(x);
```

```
  IF x<10                THEN y:=0;
  IF (x>=10) AND (x<20) THEN y:=x;
  IF (x>=20) AND (x<30) THEN y:=10;
  IF x>=30               THEN y:=-0.5*x+20;
  writeln('x=',x:10:2,', y=',y:10:2)
END.
```

上述方案的优点是程序和原题目直接对应，读起来直观，比较容易理解。但它的缺点和[例 1]的第一种做法相同：其中有些比较是多余的。如已经判断过是否 x<10 后就没必要再判断是否 x>=10；而且，已经满足 x<10 后也没有必要再去作其它判断， 所以可以将其它判断全放在 x<10 后的 ELSE 子句中，等等。如果要改进这一点，就需要采用 IF 语句的嵌套。由此可得另一种方案如下：

```
PROGRAM  func(input,output);
  VAR
    x,y:real;
BEGIN
  write('Input x:');
  readln(x);
  IF x<10
    THEN y:=0
    ELSE IF x<20
           THEN y:=x
           ELSE IF x<30
                  THEN y:=10
                  ELSE y:=-0.5*x+20;
  writeln('x=',x:10:2,', y=',y:10:2)
END.
```

这个题目可以有多种不同做法，这里再举出一种如下：

```
PROGRAM  func(input,output);
  VAR
    x,y:real;
BEGIN
  write('Input x:');
  readln(x);
  IF x<20
    THEN IF x<10 THEN y:=0
                 ELSE y:=x
    ELSE IF x<30 THEN y:=10
                 ELSE y:=-0.5*x+20;
```

```
    write('x=',x:10:2,', y=',y:10:2)
END.
```

这三个程序运行结果相同，只是所花费的时间略有差别。第一个程序不管 x 等于几总要执行 6 次比较和两次“与”；第二个程序在 x 最大时要执行一次比较，x 最小时执行 3 次比较；第三个程序不管 x 等于几都执行两次比较。在平常情况下，这些差别并不重要。

这三个程序中，第一个没有用 IF 语句的嵌套，而后两个都用了 IF 语句的嵌套。为了看清楚嵌套的层次，我们将后两个程序中 IF 语句里的每个子句框起来，用下图示意。第二个程序中的 IF 语句是：

```
IF x<10
  THEN y:=0
  ELSE IF x<20
         THEN y:=x
         ELSE IF x<30
                THEN y:=10
                ELSE y:=-0.5*x+20
```

而第三个程序中的 IF 语句是：

```
IF x<20
  THEN IF x<10 THEN y:=0
               ELSE y:=x
  ELSE IF x<30 THEN y:=10
               ELSE y:=-0.5*x+20
```

[例 4] 读入三个实数，找出并输出最大数。

读入三个数后，先找出最大数赋给变量 max，再输出 max 的值。找最大数的算法可以是：

如果 c 不同时大于 a、b，
　　则：　将 a、b 中最大者赋给 max；
　　否则：将 c 赋给 max。

程序如下：

```
PROGRAM lar(input,output);
  VAR
    a,b,c,max:real;
BEGIN
  read(a,b,c);
  IF (c<=a) OR (c<=b)
    THEN
      IF a>b
        THEN max:=a
        ELSE max:=b
    ELSE
        max:=c;
  writeln('the largest number is',max:8:2)
END.
```

其中(c<=a) OR (c<=b)和“c 不同时大于 a、b”等效。按照前面学过的德·摩根律，有

```
NOT ( (c>a) AND (c>b) )
= ( NOT (c>a) ) OR ( NOT (c>b) )
=  (c<=a) OR (c<=b)
```

这里用了嵌套的 IF 语句，比较运算比较多。假如不这样，而是采用[例 1] 的第三种做法的思路，即“重复赋值”的做法，则这个程序的中间一段也可以改为

```
IF a>b
  THEN max:=a
  ELSE max:=b;
IF c>max
  THEN max:=c;
```

这里暂不管 c，先将 a、b 中最大者赋给 max，再有条件地重新赋值，即如果 c 比 max 大则再将 c 赋给 max。这个改后的算法不仅程序结构较简单（不用嵌套的 IF 语句），而且运行时的比较运算次数也减少了。这是在三个数中选最大的，假如数再多些，这一点就更突出，若不用重新赋值的办法程序就会复杂得几乎无法编写。所以我们前面说：有时采用有条件地重新赋值的做法可以简化程序。

在前面的嵌套的 IF 语句例子中，每一层的 IF 语句都包含有 THEN 子句和 ELSE 子句，此时结构总是明确的，但是如果某一层没有 ELSE 子句，有时就会出现结构不清楚的情况。先看一个例子：

[例 5] 输入三个数，判断以这三个数为边是否能够组成一个三角形。若能，则输出其特征（等边、等腰、直角、一般，但是等边的不再指出同时是等腰）及面积；若不能，则给出适当的信息。

问题分析：该例中要求判断输入的三个数是否能组成三角形，判断原则是任何两边之

和必须大于第三边。当满足这个条件时，要求做两步工作，即判断三角形特征和求面积。求面积的公式为

```
s=sqrt(l*(l-a)*(l-b)*(l-c))
```

其中 l=(a+b+c)/2

当输入的三个数不能构成三角形时，输出出错信息。

```
PROGRAM area(input,output);
  VAR
    a,b,c,l,s:real;
BEGIN
  writeln('input a,b,c:');
  read(a,b,c);
  IF (a+b>c) AND (b+c>a) AND (a+c>b)
    THEN
      BEGIN
        IF (a=b) AND (b=c)
          THEN write('等边,')
          ELSE
            IF (a=b) OR (a=c) OR (b=c)
              THEN   write('等腰,');
        IF (a*a+b*b=c*c) OR (b*b+c*c=a*a) OR (a*a+c*c=b*b)
          THEN  write('直角,');
        l:=(a+b+c)/2;
        s:=sqrt(l*(l-a)*(l-b)*(l-c));
        writeln('s=',s:6:2)
    END
    ELSE
      writeln('不能构成三角形')
END.
```

按照题目要求，等边的不再指出同时是等腰，故程序中给出“等边”及“等腰”特征的这一段是

```
IF (a=b) AND (b=c)
  THEN write('等边,')
  ELSE
    IF (a=b) OR (a=c) OR (b=c)
      THEN   write('等腰,');
```

这里内层的 IF 语句中没带 ELSE 子句。这个程序仍然没问题。但是假如有人把这一句外层反过来（同时条件也否定），编成

```
IF (a<>b) OR (b<>c)
```

```
    THEN
      IF (a=b) OR (a=c) OR (b=c)
        THEN   write('等腰,')
    ELSE write('等边,');
```

就有问题了。编写程序时的意图是这样的：

```
IF (a<>b) OR (b<>c)
  THEN | IF (a=b) OR (a=c) OR (b=c)
       |   THEN | write('等腰,') |
  ELSE | write('等边,') |
```

但计算机对这一句的理解却是这样的：

```
IF (a<>b) OR (b<>c)
  THEN | IF (a=b) OR (a=c) OR (b=c)
       |   THEN | write('等腰,') |
       |   ELSE | write('等边,') |
```

因为这两种理解文字上没有区别（我们书写时的上下对齐仅仅是供人阅读时好看，并没有语法意义），计算机只能取其一种理解。为排除二义性，PASCAL 的标准规定：

一个没有 ELSE 部分的如果语句，其后面不应直接跟有记号 ELSE。

这里“没有 ELSE 部分的如果语句”就是本节开头介绍的第一种形式的 IF 语句。这一段程序如按上面的前一种理解，外层的 THEN 子句本身就是一个“没有 ELSE 部分的如果语句”，但这个子句结尾后边紧接着就是外层的 ELSE，违反了上述规定。所以这种情况不可以按前一种构造来理解。于是计算机自动地将它按照后一种构造来理解了。

从这个规定可以得出一个推论：优先考虑了语句括号的因素后，

一个 ELSE 部分应与它前面最近的一个尚未配对的 THEN 配对。

这也可以称作“内层优先”的原则，即优先将 ELSE 划归内层。

所以这样编写的程序就会形成后一种理解的效果：不等边又不等腰时打出“等边”，而真正等边时却什么都不打。显然这是不对的。

为避免这种错误，我们可以尽量回避上面的这种编写法。有时候出于某种需要，非要将一个不带 ELSE 子句的 IF 语句用作外层带有 ELSE 子句的 IF 语句的 THEN 子句的话，可以将子句用 BEGIN 和 END 这一对“语句括号”括起来。如

```
    IF (a<>b) OR (b<>c)
      THEN
        BEGIN
          IF (a=b) OR (a=c) OR (b=c)
```

```
                THEN   write('等腰,')
            END
        ELSE write('等边,');
```

这样就不会被计算机理解错了，因为语句括号外的 ELSE 无论如何是不能和括号内的 THEN 配对的。

4.3.4 综合实例

[例 6] 根据系数求解一元二次方程，要求兼顾二次项为 0 的特殊情况。

问题分析：第二章已经有了要求 a≠0 时的程序，这里根据 a 的取值分为 a=0 和 a≠0 两种情况，a=0 时按一次方程处理：

(1)若 b=0，则输出出错信息。

(2)若 b<>0，则 a 的解为-c/b。

a≠0 时采用同第二章的算法。

```
PROGRAM sample(input,output);
{根据系数求解一元二次方程}
  VAR
    a,b,c,d,r,p,x1,x2:real;
BEGIN
  read(a,b,c);              { 输入系数 a，b，c }
  IF a=0
    THEN                    { 按一次方程处理 }
      IF b=0
        THEN                { 错误 }
          writeln('Invalid input')
        ELSE
          BEGIN             { 解一次方程 }
            x1:=-c/b;
            writeln(x1)
          END
    ELSE                    { 按二次方程处理 }
      BEGIN
        d:=b*b-4*a*c;           { b²-4ac→d }
        IF  d<0
          THEN
            BEGIN
              r:=-b/2/a;                { 实部(-b)/(2a)→r }
              p:=sqrt(abs(d))/2/a;      { 虚部√|d|/(2a)→p }
              writeln(r,'+',p,'i');     { 输出二虚根 }
```

```
            writeln(r,'-',p,'i')
          END
        ELSE
          BEGIN
            x1:=(-b+sqrt(d))/2/a;   { (-b+√d)/(2a)→x1}
            x2:=(-b-sqrt(d))/2/a;   { (-b-√d)/(2a)→x2}
            writeln(x1,',',x2)      { 输出二实根 }
          END
    END
END .
```

[例 7] 给出一个不多于三位数的正整数，求它是几位数，并分别打印出各位上的数字。

问题分析：输入一个数，首先对它进行判断，如果在 1～999 之间，则求出个位、十位、百位的数字，分别存入 a、b、c。

求个位数时，可用该数字除以 10 求余；

用该数字整除以 100，商是百位数，余数是低两位数；

求十位数时，可用低两位数整除以 10。

根据这些数字可判断输入的数是几位数，若百位数不为零，则是三位数；否则若十位不为零则为两位数；否则为一位数。

```
PROGRAM number(input,output);
  VAR
    m,t,a,b,c:integer;
BEGIN
  write('Input m:');
  readln(m);
  IF (m>0) AND (m<1000)
    THEN
      BEGIN
        c:=m div 100;
        b:=(m mod 100) div 10;
        a:=m mod 10;
        IF c<>0
          THEN t:=3
          ELSE IF b<>0
                 THEN t:=2
                 ELSE t:=1;
        writeln('这个数字是',t:1,'位数');
        writeln(c:4,b:4,a:4)
```

```
        END;
    END.
```

[例 8] 一天的时间可以用 12 小时制或 24 小时制来表示，若用 12 小时制，则还需指明上半天还是下半天。上半天用AM指明，下半天用PM指明。编写一个程序，读入 12 小时制的时间，转换为 24 小时制的时间，例如输入[①]

```
10:50:30 PM
```

则输出

```
22:50:30
```

要求在转换前对所提供的数据是否合理进行检查，若不合理，则输出错误信息。

程序如下：

```
PROGRAM sjzh(input,output);
  VAR
    h,m,s:integer;
    x,y:char;
BEGIN
  writeln('Input the time:');
  readln(h,x,m,x,s,x,x,y);
  IF      (h>0) AND (h<=12)
      AND (m>=0) AND (m<60)
      AND (s>=0) AND (s<60)
      AND ( (y='M') OR (y='m') )
      AND ( (x='p') OR (x='P') OR (x='a') OR (x='A') )
    THEN
      BEGIN
        IF (x='p') OR (x='P') THEN h:=h+12;
        writeln('The time is:',h:2,':',m:2,':',s:2)
      END
    ELSE writeln('Invalid data !')
END.
```

这个程序不复杂，但要注意两点：

其一，程序中判断数据是否合法时用了复杂逻辑表达式，应注意运算优先级及括号的使用。也可以将这个式子换成它的否定式并将其 THEN 子句和 ELSE 子句互换。对该式子否定时可以将它整个括起来前面加一个 NOT， 也可以用前面学过的德·摩根律展开后对每个关系式分别否定。若采用后一种方法，则这个 IF 语句最后可改为

```
 IF      (h<=0) OR (h>12)
```

[①] 某些非标准的 PASCAL 不允许输入整数紧接冒号，若在该系统上试验本程序，可以在输入时将冒号都换成空格。

```
    OR (m<0) OR (m>=60)
    OR (s<0) OR (s>=60)
    OR ( (y<>'M') AND (y<>'m') )
    OR ( (x<>'p') AND (x<>'P') AND (x<>'a') AND (x<>'A') )
  THEN
    writeln('Invalid data !')
  ELSE
    BEGIN
      IF (x='p') OR (x='P') THEN h:=h+12;
      writeln('The time is:',h:2,':',m:2,':',s:2)
    END
```

其二，程序中有

```
readln(h,x,m,x,s,x,x,y);
```

四次读入 x，只有最后一次读入的字符留下了。前三次是为了略过键盘打入的冒号和空格号。

[例 9] 输入一个年号，判断它是否是闰年。

判断闰年的规则是：

```
如果年号能被 100 整除
    那么
        如果此年号能被 400 整除
            则它是闰年
            否则它不是闰年
    否则
        如果此年号能被 4 整除
            则它是闰年
            否则它不是闰年
```

程序的一个方案如下：

```
PROGRAM LeapYear(Input,Output);
  VAR
    year:INTEGER;
    leap:BOOLEAN;
BEGIN
  write('Enter year:');
  readln(year);
  IF year MOD 100 = 0
    THEN leap:=year MOD 400=0
    ELSE leap:=year MOD 4=0;
  IF leap
```

```
    THEN writeln(year:6,'is a leap year.')
    ELSE writeln(year:6,'is not a leap year.')
END .
```

这里采用了一个布尔变量 leap 记载“是闰年”的判断结果。程序的另一个方案是省去一个 IF 语句及变量 leap，但要增加一些逻辑运算：

```
PROGRAM LeapYear(Input,Output);
  VAR
    year:INTEGER;
BEGIN
  write('Enter year:');
  readln(year);
  IF  (year MOD 100 <> 0) AND (year MOD 4 = 0)
      OR (year MOD 400 = 0)
    THEN writeln(year:6,'is a leap year.')
    ELSE writeln(year:6,'is not a leap year.')
END .
```

初学者常常不习惯将关系式的结果看作一个值，例如赋值语句

```
leap:=year MOD 4=0
```

常有人写作

```
IF  year MOD 4=0  THEN leap:=true  ELSE leap:=false
```

又常常不习惯直接将逻辑值看做条件，例如

```
IF leap THEN ......
```

常有人写作

```
IF leap=true  THEN ......
```

等等。这些写法虽不算错误，但毕竟反映了我们对逻辑量的概念不熟悉。这样，不仅自己编写的程序冗长，而且常常看不懂别人编写的程序，所以应该引起注意。

在这一节最后，再介绍一种情况。从上面可以看到，AND、OR 一类的逻辑运算常可用 IF 语句的选择结构来代替，所以遇到复杂判断的问题，常可以有不同的设计方案。有的方案分支多一些，少用或不用逻辑运算；有的方案用了逻辑运算，流程的分支少一些。一般说来，分支少的方案读起来比较简明，被认为较好。但要注意，布尔表达式是在其各部分都计算完后才能得出值的，而多分支的方案因为不必每次都将复杂条件中的各部分计算完，所以程序运行可以更快些。如，上面判断闰年的第一个程序中，若

```
year MOD 100 = 0
```

成立，就不必再计算

```
year MOD 4
```

若

```
year MOD 100≠0
```

则不必再计算

```
year MOD 400
```

而第二个程序中这些计算全要进行。

另外，有时候分支多的方案还可以回避错误。例如，有这样一段程序，功能要求为：若 n 是 0 或者 m 是 n 的倍数就输出字符 A，否则输出字符 B。按题意我们可能会编出

```
IF (n=0) OR (m MOD n = 0) THEN write('A') ELSE write('B')
```

但是，一旦 n 是 0，则在计算 m MOD n 时就会因除数为 0 而溢出。假如所用系统在溢出时不报错，继续执行下去，则虽然除法结果不对，但整段程序结果还是对的。可是有的系统一旦遇到这种溢出就停止执行程序并报告出错，这样就无法得出结果了。为此，可以改成

```
IF n=0 THEN write('A')
       ELSE IF m MOD n =0 THEN write('A')
                          ELSE write('B')
```

于是当 n 是 0 时不必计算 m MOD n，问题就解决了。

4.4 情况语句（CASE 语句）

IF 语句只能从两种情况中选择一种。有时候情况比较复杂，要从多种可能中选择其一，这时候使用 IF 语句就比较麻烦。例如我们读进一个 0～6 之间的整数，打印出相应的英文星期几，用 IF 语句

```
IF a=0 THEN write('Sunday');
IF a=1 THEN write('Monday');
    ......
    ......
IF a=6 THEN write('Saturday')
```

或者用嵌套的 IF 语句

```
IF a=0
  THEN   write('Sunday')
  ELSE  IF a=1
          THEN write('Monday')
             ......
             ......
              ELSE write('Saturday')
```

或

```
IF a<>0
  THEN IF a<>1
         THEN IF a<>2
                THEN
                   ......
                   ......
```

```
                    IF a<>5
                      THEN write('Saturday')
                      ELSE write('Friday')
                ......
                ......
          ELSE  write('Sunday')
```

无论我们使用哪种形式都比较麻烦。更主要的是这些写法形式上都比较繁琐，容易出现笔误。其中第一个形式虽然清楚些，但前面我们已经讨论过，这个形式有多余的操作，效率较低。因此引入了情况语句来解决这个问题，采用情况语句，可将上面的程序段改写如下：

```
CASE a OF
  0:write('Sunday');
  1:wriet('Monday');
  2:write('Tuesday');
  3:write('Wednesday');
  4:write('Thursday');
  5:write('Friday');
  6:write('Saturday')
END
```

情况语句也叫 CASE 语句或分情况语句，它可以从多种情况中选择其一。

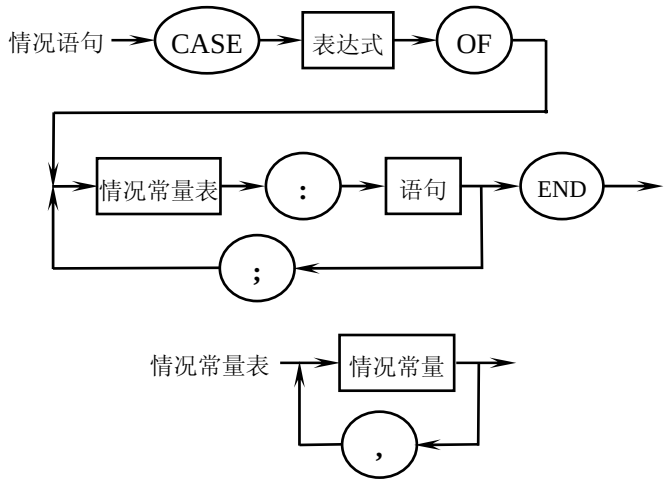


图 4.8　情况语句的语法图

情况语句的语法图如图 4.8。其简明形式如下：

```
CASE  表达式  OF
  情况常量表 1:语句 1;
  情况常量表 2:语句 2;
    ......
    ......
  情况常量表 n:语句 n
END;
```

其中“情况常量表”由若干个“情况常量”组成，其间由逗号分隔。在前面的例子中，该形式中的“表达式”就是 a，“情况常量表”共有七个， 每个“情况常量表”中各有一个“情况常量”，即 0、1、2、3、4、5、6。 因为这个例子的每个“情况常量表”中只有一个“情况常量”，故其间的逗号分隔就没有了。

情况语句的功能是：若该“表达式”的值等于句中某个“情况常量表”中的一个“情况常量”，就执行该“情况常量表”后的“语句”。

这里要说明几点：

1、CASE、OF 和 END 是情况语句的标志，END 与 CASE 对应而不是与 BEGIN 对应。除语

句 n 外，每一语句后都有一个分号，其作用是将它与下一个情况常量表分隔开来。

2、这里的“表达式”称为“情况下标”或“情况表达式”， 也可以称为“选择器”，它必须是顺序类型的表达式，可以是整型、字符型、布尔型等等，但不能是实型。

3、情况常量表中的情况常量又称为情况标号。它是情况表达式可能具有的值， 应写成常量的形式，可以是字面常量，也可以是符号常量，但不可以是其它形式的表达式。

4、情况常量应与表达式的数据类型相同。

5、情况常量出现的顺序是任意的，即每个情况常量表出现的顺序是任意的， 无先后之分，而且每个情况常量表内所包括的常量出现的次序也是任意的。

6、同一个情况语句中不可出现相等的情况常量。

7、如果情况表达式的值不落在情况常量的范围内，在不同的系统中有不同的处理方式。标准规定此时算出错，而有的系统则认为本情况语句无效，等效于一个空语句。也有的系统在最后一个情况常量表后的语句之后，END 之前增加一个 ELSE 部分（或 OTHERWISE 部分），格式是在字符号 ELSE 后跟一个语句，当情况表达式的值不落在情况常量的范围内时执行这个语句。需要注意的是，这种情况与 IF 语句中的 ELSE 有所不同，在 ELSE 之前应有一个分号与前面的语句分隔。

[例 1] 输入两个运算数及一个运算符，输出运算结果。

问题分析：这个问题相当于用计算器进行计算。设两个运算数为 x 和 y，一个运算符为 op，结果为 result。现在假定运算符只可能是+、-、*、/，并且*号允许用字母 X 代替，将 op 说明为字符型，x、y、result 说明为实型。当我们输入不同的运算符时，将选择不同的公式对两个操作数进行运算。程序如下：

```
PROGRAM cal(input,output);
  VAR
    x,y,result:real;
    op:char;
BEGIN
  writeln('Input x,y and op:');
  readln(x,y);
  readln(op);
  CASE op OF
    '+':    result:=x+y;
    '-':    result:=x-y;
    '*','X':result:=x*y;
    '/':    result:=x/y
  END;
  writeln(x:6:2,op,y:6:2,'=',result:6:2)
END.
```

这个例子中情况语句的第三个情况常量表中有两个情况常量：'*'和'X'。

关于情况语句，在以后的章节中还会见到别的例子。应该指出，情况语句中进行的是

“等于”判断，而 IF 语句进行的是一般的逻辑判断，所以，并不是任何多分支的 IF 语句都适合于改成情况语句。能不能改，取决于能不能设计出一个合适的情况表达式，使得该表达式的有限种取值恰好能代表我们需要的分支。下面是一个这种例子。

[例 2] 把上一节例 3 用情况语句重新改写（设已知输入的自变量小于 50 且非负， 程序不考虑输入超出此界的情况）。

分析：引入表达式 trunc(x/10)，从右表可以看出，在所考虑的范围内， 该表达式的取值可以代表我们需要的分支条件。可以编出以下程序：

条件	trunc(x/10)的值
0<=x<10	0
10<=x<20	1
20<=x<30	2
30<=x<50	3 或 4

```
PROGRAM  hs(input,output);
  VAR
    x,y:real;
BEGIN
  write('Input x');
  readln(x);
  CASE trunc(x/10) OF
    0: y:=0;
    1: y:=x;
    2: y:=10;
    3,4: y:=-0.5*x+20
  END;
  writeln('x=',x:10:2,', y=',y:10:2)
END.
```

顺便指出，有时虽然能构造出可用的情况表达式，但若过于复杂，就不如仍然用 IF 语句合适。如

```
IF x<0  THEN IF y<0 THEN ......
                    ELSE ......
        ELSE IF y<0 THEN ......
                    ELSE ......
```

虽然也可以改为

```
CASE ord(x<0)*2+ord(y<0) OF
  3: ...... ;
  2: ...... ;
  1: ...... ;
  0: ......
END
```

但这个情况表达式就有些过于复杂了。

习题

4.1 不用 IF 语句编程：键盘输入 x，y 坐标，程序判断该点是否在以原点为圆心的单位园内。是则输出 true，否则输出 false。恰在圆周上的点也算在圆内。

4.2 下面两组语句中，对于第一组的每一个语句，第二组中都有一个或几个与它等效，请分别指出来：

第一组：

```
b:= x>y
b:=b AND (x>y)
b:=b OR (x>y)
IF x*y<=0 THEN write('*')
IF (x>0) AND (y>0) THEN write('*')
```

第二组：

```
IF x>y THEN b:=true
IF x<=y THEN b:=false
IF (x>0)=(y>0) THEN write('*')
IF x*y=0 THEN write('*') ELSE IF x*y<0 THEN write('*')
IF x>0 THEN IF y>0 THEN write('*')
IF x>0 THEN write('*') ELSE IF y>0 THEN write('*')
IF (x=0) OR (y=0) OR ((x>0)<>(y>0)) THEN write('*')
IF x*y=0 THEN IF x*y<0 THEN write('*')
IF x>y THEN b:=true ELSE b:=false
```

4.3 编程：键盘输入 x，y，若二者均非 0，则按下式计算并输出 z 值；若二者中有 0，

$$z=\begin{cases} e^x+e^y, & \text{Ⅰ象限} \\ \sin x \sin y, & \text{Ⅱ象限} \\ \tan(x+y), & \text{Ⅲ象限} \\ x^2+y^2, & \text{Ⅳ象限} \end{cases}$$

则给出错误报警并结束程序。

4.4 键盘输入三个字母，机器按它们在字母表上的先后次序输出它们。

4.5 改写 4.4 节[例 2]的程序，使它在输入超出允许的范围时给出错误报警并结束程序。

4.6 说以下(1)和(2)两个语句一定等效，这话对吗？若不对，什么情况下不等效？

(1) IF 条件甲 THEN 语句甲 ELSE 语句乙

(2)
```
BEGIN IF 条件甲 THEN 语句甲 ;
      IF NOT 条件甲 THEN 语句乙
END
```

第五章　循环结构的程序设计

程序设计过程中，往往会遇到一些需要重复处理的问题，例如，在第二章 2.5 节中我们就遇到过“将 x:=x*0.8+1 重复 10000 次”这样的操作。

解决这类问题，若大量重复编写相同的程序段，显然是不现实的。那样不仅篇幅太大，而且重复的次数有时是事前不知道的，就更没法编写。第一章介绍过的基本结构中有一种叫循环结构，正是解决这种问题的最好手段。

PASCAL 中提供了专门用来编写循环结构程序的语句，称作循环类语句。循环类语句有三种：WHILE 语句、REPEAT 语句和 FOR 语句。其中 WHILE 语句和 REPEAT 语句可被称为条件循环语句，它们是根据条件是否成立而决定是否继续执行循环体；而 FOR 语句可被称为计数循环语句，它是根据计数器的初态和终态来决定是否执行循环体，循环次数是在循环开始前就可以预知的。

5.1　WHILE 语句

WHILE 语句可以实现第一章 1.3 节介绍的当型（WHILE 型）循环结构。

WHILE 语句的形式如下：

WHILE 布尔表达式 DO 语句

这里的“布尔表达式”就是当型循环结构中的条件，而“语句”就是循环体。要注意这里的循环体按规定应是一个语句，若想用多个语句组成循环体的话，不要忘了用 BEGIN 和 END 将其合成一个复合语句才能写入 WHILE 语句中。

WHILE 语句的执行过程叙述为：

1.首先判断布尔表达式的值，如果为真，那么就执行 2，否则，结束整个循环。

2.执行循环体语句，返回 1。

[例 1] 输入两个正整数，求最大公约数。

求最大公约数有多种方法，这里介绍两种。

一、用尝试法：

按定义，公约数是能同时整除二数的数，现在要求其最大者，可以先选一个不可能小于它的整数，检验它是否公约数；若不是就减 1，再检验；若还不是就再减 1，……直到找到为止。我们知道两个正整数的最大公约数不可能大于这两个数中的任一个，故开始时可以任选一个等于二者之一的数。

此算法的框图如图 5.1。

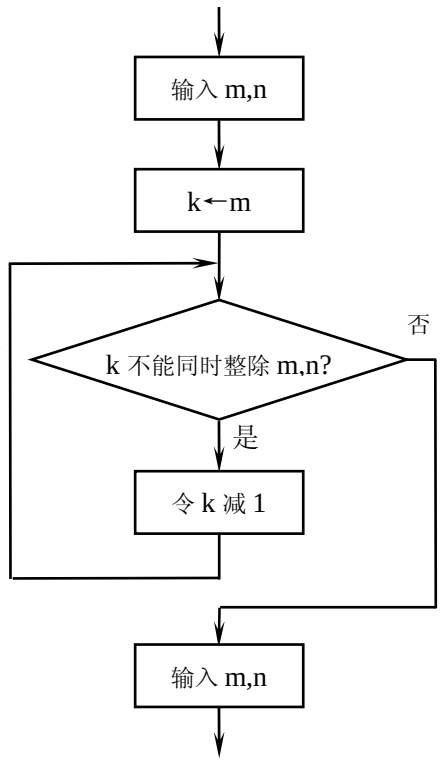


图 5.1　求最大公约数的尝试法

图的中部恰好是一个当型循环结构，故可以采用 WHILE 语句来实现。这里，“k 不能同时整除 m,n”等效于“k 不能整除 m 或者 k 不能整除 n”。程序如下：

```
PROGRAM gcd(input,output);
VAR
  m,n,k:integer;
BEGIN
  write('Input m and n: ');
  readln(m,n);
  k:=m;
  WHILE (m MOD k <>0) OR (n MOD k <>0) DO k:=k-1;
  writeln(k)
END.
```

上述尝试法的优点是道理简单，程序也简单；缺点是运算次数较多，如果输入数较大则程序运行较慢，不如辗转相除法高效。

二、辗转相除法：

这个方法的理论根据是数论中的以下定理：

若 m 被 n 除得余数 r（即 n≠0 且存在整数商 q 使 m=qn+r），则 m 和 n 的公约数与 n 和 r 的公约数相同。

这个定理的理由也较简单：设 m 和 n 有公约数 c，即存在整数 M 和 N 使

m=Mc, n=Nc

于是

r=m-qn=Mc-qNc=(M-qN)c

显然(M-qN)是整数，所以 c 也是 r 的约数。反过来，设 n 和 r 有公约数 d， 用同样方法也不难证明 d 也是 m 的约数。

根据这个定理，只要 n≠0，就可以作除法求出 r，于是求 m 和 n 的最大公约数就可以用求 n 和 r 的最大公约数来代替。可以用 n 代替原来的 m，用 r 代替原来的 n，再重复这个过程。

这样一次次重复后，一旦有一次 n=0，就不能再作除法了，此时，因为 0 被任何正整数都可以除尽，所以 0 和 m 的最大公约数就是 m。

此算法的框图如图 5.2。

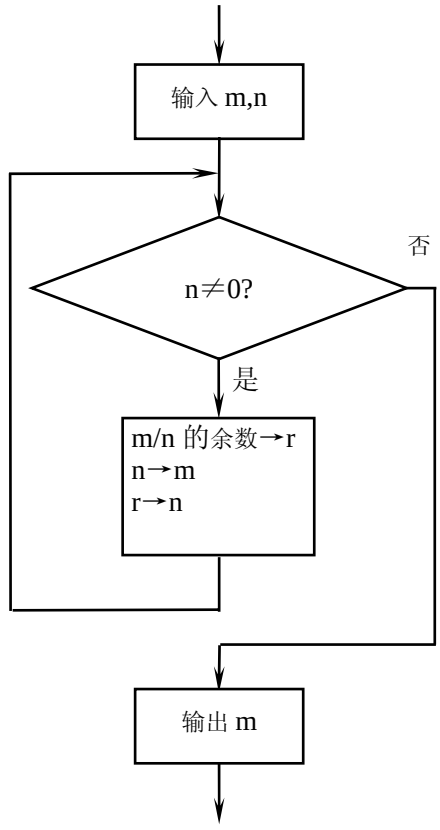


图 5.2　求最大公约数的辗转相除法

程序如下：

```
PROGRAM gcd(input,output);
VAR
  r,m,n:integer;
```

```
BEGIN
  write('Input m and n: ');
  readln(m,n);
  WHILE n<>0 DO
    BEGIN
      r:=m mod n;
      m:=n;
      n:=r
    END;
  writeln(m)
END.
```

前面第一章中已经说过，为了不陷入死循环，必须保证总有一步该“布尔表达式”取值为假。在尝试法求最大公约数的程序中，因为最大公约数理论上一定存在，而尝试的范围又有限，所以必然有一步会找到。在辗转相除法中，因为余数必然小于除数，且非负，所以每一步 n 值必然减小。但小于某个数的正整数是有限的，所以总有一步 n 会成为 0。

也有的时候，循环的条件不是由程序改变的，而是取决于外部的环境。这种情况在自动控制程序中是常见的。例如：某变量 state 不是普通的内存变量， 而是连接到某设备的信号输入口，设备正常时读得 0，不正常时非 0。程序运行到某步时要检查该设备状态，若不正常就反复报警，直到正常才停止报警并继续运行下面的程序。这一段程序可以如

```
WHILE state<>0 DO 报警
```

这样只有当外部的设备正常了，布尔表达式 state<>0 才会变为假，循环才会结束。这种操作通常称作“查询”，根据需要有时也可以编成下面将介绍的 UNTIL 型。 这方面的技术牵涉到硬件的知识较多，本书不打算进一步介绍。

5.2 REPEAT 语句

REPEAT 语句可以实现第一章 1.3 节介绍的直到型（UNTIL 型）循环结构，它的格式如下：

```
REPEAT
   语句 1;
   语句 2;
    ......
   语句 n
UNTIL 布尔表达式
```

它的执行过程如下：

1.先执行 REPEAT 与 UNTIL 之间的语句序列。

2.判断布尔表达式，若为假，转向 1，否则结束循环。

REPEAT 语句中的循环体是一个语句序列（语句序列的定义见第四章 4.1 节）， 可以不

是一个语句，这点与 WHILE 语句有所不同。这是因为 REPEAT 与 UNTIL 已经起了天然的括号作用，所以不用 BEGIN 与 END 也不会造成界限不清。

另外，REPEAT 语句中的“布尔表达式”是循环结束的条件，而不是循环继续的条件，这点与 WHILE 语句相反。所以，为避免陷入死循环， 程序的运行应能使布尔表达式的值趋向于真。

从功能上看，REPEAT 语句与 WHILE 语句的一个重要区别之处是：它第一次进入循环体是无条件的。所以，它至少要执行一次循环体；而 WHILE 语句则在某些情况下有可能一次也不执行循环体。

[例 1] 编程序，用牛顿迭代法求非负数 a 的平方根。牛顿迭代法的公式为

$$x_{n+1}=(x_n+a/x_n)/2$$

式中，x_{n+1}是一个远较 x_n更接近$\sqrt{a}$ 的准确值的近似值。如果 x_{n+1}和 x_n之差可忽略， 就可以认为结果已求出。可先粗略地取一个 x_0（如取 x_0=1），根据上式由 x_0求出 x_1，由 x_1求出 x_2，……等等，其值必然越来越逼近$\sqrt{a}$ 的准确值。 本程序精确到最后两次迭代结果相差绝对值缩小到 10^{-5}以内为止，输出舍入保留 4 位小数。

为便于检查迭代差，我们将公式改写如下：

$$x_{n+1}=x_n+\delta_n$$

其中 $\delta_n=(a/x_n-x_n)/2$

其中 δ_n就是迭代差。x_n用变量 x 存放，δ_n用变量 delta 存放。这个算法流程见图 5.3。

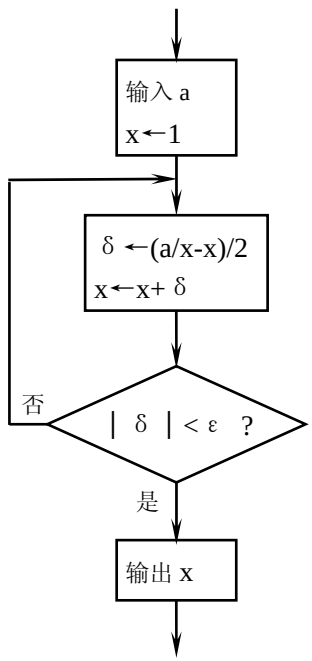


图 5.3　牛顿迭代法求非负数的平方根

图的中部恰好是一个直到型循环结构，故可以采用 REPEAT 语句来实现。程序如下：

```
PROGRAM squareroot(input,output);
  CONST
    eps=1E-5;
  VAR
    a,x,delta:real;
BEGIN
  write('Input a:');
  readln(a);
  x:=1;
  REPEAT
    delta:=(a/x-x)*0.5;
    x:=x+delta
  UNTIL abs(delta)<eps;
  writeln(x:6:4)
END.
```

这个程序中 x_0、x_1、x_2、……等等都用同一个变量存放，这是因为在这个程序里，迭

代差已经存在 delta 中，故下一个 x 一求出来，上一个 x 就没有用了，可以不再保留。

这个题目还有另一种做法：不像上面那样修改公式，而是直接用两个 x 相减的办法求迭代差。这样下一个 x 求出来时，上一个 x 还不能丢。为解决这个问题，可在新的 x 求出前先将旧的 x 暂存在另一变量里（下面的程序中用变量 x1）。这个做法的程序如下：

```
PROGRAM squareroot(input,output);
  CONST
    eps=1E-5;
  VAR
    a,x,x1:real;
BEGIN
  write('Input a:');
  readln(a);
  x:=1;
  REPEAT
    x1:=x;
    x:=(x+a/x)*0.5
  UNTIL abs(x-x1)<eps;
  writeln(x:6:4)
END.
```

一般的应用程序中，WHILE 语句和 REPEAT 语句可以相互转换。 转换时除应注意将布尔表达式取反以外，还应注意第一次进入循环体前的判断条件的问题。

如，上一节[例 1]中用尝试法求最大公约数的算法程序中的一段可以改写如下：

```
.....
k:=m+1;
REPEAT
  k:=k-1
UNTIL (m MOD k =0) AND (n MOD k =0);
.....
```

与原来的程序相比，这里 k 的初始值增加了 1。这是因为原来的程序在第一次进入循环之前就要检查 k 是否公约数，而现在第一次进入循环之前不检查，若 k 不增 1 就可能漏过 m 恰好是最大公约数的情况。

又如，该例中用辗转相除法求最大公约数的算法程序中的 WHILE 语句可以改写为如下的 REPEAT 语句：

```
REPEAT
  r:=m mod n;
  m:=n;
  n:=r
UNTIL n=0;
```

虽然没作额外的修改，但与原来的程序相比，功能已经有了细节上的不同：原来的程序在输入 n 为 0 时也能得出结果，而现在的程序若输入 n 为 0 就会发生 0 作除数的错误。

假如程序的上文已经保证刚进入 WHILE 语句时条件一定为真（例如， 辗转相除法程序中假设上文已保证 n 非 0）的话，这样改写就没有任何问题了。

反过来，要将 UNTIL 语句改写成 WHILE 语句，问题更多些。我们知道，布尔表达式是判断条件，然而条件往往是在上文产生的。也就是说，上文通常要有“产生条件”的操作，这些操作，在 REPEAT 语句中一般是包括在循环体中的。如本节[例 1]的第一个程序中，UNTIL 后要判断 delta 的值，则循环体中就包括有计算 delta 的操作。改成 WHILE 语句后， 判断条件放在开头，一般说来，“产生条件”是在上一轮循环中完成的。但第一次判断的条件，却必须在循环开始之前就已经产生了。如果原来上文没有这个功能，改写时就需要补上。如图 5.4 示意。

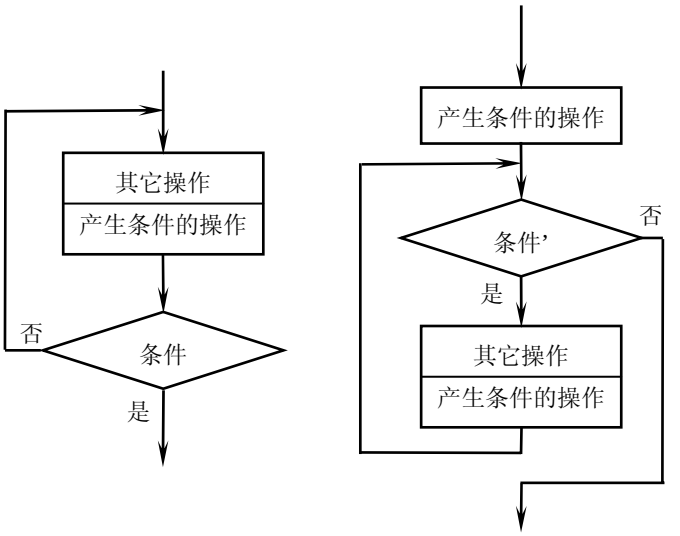


图 5.4　两种循环的对比

从图中可以看到，将 REPEAT 语句改写成 WHILE 语句，除条件要否定以外，常常需要在前面补写一些操作。如[例 1]第一个程序中的 REPEAT 语句和将它改写后的方案对比如下：

```
REPEAT                          delta:=(a/x-x)*0.5;
  delta:=(a/x-x)*0.5;           x:=x+delta;
  x:=x+delta                    WHILE abs(delta)>=eps DO
UNTIL abs(delta)<eps;             BEGIN
                                    delta:=(a/x-x)*0.5;
                                    x:=x+delta
                                  END;
```

补写的“产生条件”的操作是专门针对第一次判断的，故有时也可以简化，只要第一次判断能通过就行。例如这一段程序也可以为

```
delta:=1;
WHILE abs(delta)>=eps DO
  BEGIN
    delta:=(a/x-x)*0.5;
    x:=x+delta
  END;
```

5.3 FOR 语句

5.3.1 计数循环的概念

有许多实际问题适合于用第一章 1.3 节介绍的计数循环来解决， 特别是那些循环次数确定的问题。例如要在屏幕上连续输出 50 个星号，可以采用如下的程序段：

```
i:=1;
WHILE i<=50 DO
  BEGIN
    write('*'); i:=i+1
  END
```

也可以采用如下的程序段：

```
i:=1;
REPEAT
  write('*'); i:=i+1
UNTIL i>50
```

这实际上就是计数循环。计数循环中有几个要素，它们是：

一、要有一个控制变量起计数的作用，如上面的 i；

二、要为控制变量指定一个初值，如上面的 1；

三、要为控制变量指定一个终值，如上面的 50；

四、每次循环要有使控制变量增量的操作，如上面的 i:=i+1。

5.3.2 FOR 语句的语法规定

因为计数循环使用非常多，若都像上面那样用 WHILE 语句或 REPEAT 语句来描述， 显得不够简练。所以 PASCAL 中提供了一种专门用来描述计数循环的语句 FOR 语句。 上述在屏幕上连续输出 50 个星号的程序段，可以写成如下的一个 FOR 语句：

```
FOR i:=1 TO 50 DO write('*')
```

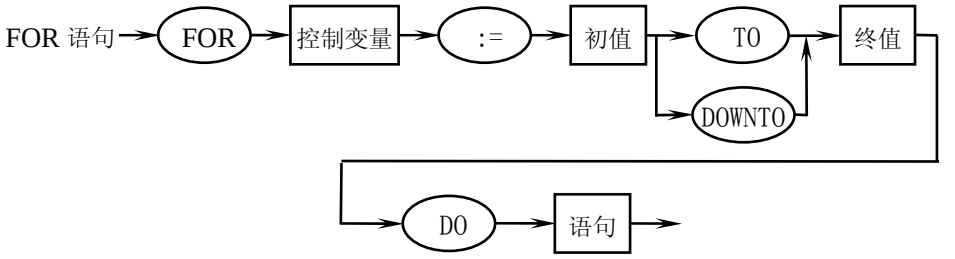


图 5.5 FOR 语句的语法图

FOR 语句的语法图见图 5.5。其简明形式如下：

```
FOR 控制变量:=初值 TO 终值 DO 语句
```

或

```
FOR 控制变量:=初值 DOWNTO 终值 DO 语句
```

其中的“控制变量”又称为“循环变量”。DO 后面的“语句”是循环体。这个循环体

中不包括“控制变量增量”的操作。要注意这里的循环体按规定应是一个语句，若想用多个语句组成循环体的话，不要忘了用 BEGIN 和 END 将其合成一个复合语句才能写入 FOR 语句中。

控制变量的增量由 TO 或 DOWNTO 来指定。如果控制变量是整数，则采用 TO 表示增量为 1 ，即每循环一次控制变量的值要增 1；采用 DOWNTO 表示增量为-1， 即每循环一次控制变量的值要减 1。换句话说，采用 TO 的 FOR 语句是递增型循环，采用 DOWNTO 的 FOR 语句是递减型循环。但控制变量也允许不用整数类型，而用其它顺序类型的变量，所以，对控制变量增量的一般规定是：递增型为“取后继”，递减型为“取前启”。递增型当控制变量的值大于终值时结束循环；递减型当控制变量的值小于终值时结束循环。设控制变量为 i，则两种 FOR 语句的流程见图 5.6。

从图中可以看出，PASCAL 中的 FOR 语句是按照当型循环结构构造的计数循环。 所以在某些特殊情况下 FOR 循环可能一次也不执行（如递增型中若所给初值大于终值的话）。

FOR 语句的功能仅靠图 5.6 描述尚不够全面严格。下面列出几点 PASCAL 中对 FOR 语句的规定，其中有些在图 5.6 中没有表示。

(1)控制变量应是顺序类型的变量。如可以是整数类型，也可以是字符型， 或以后讲到的枚举类型或子域类型等，要注意不可以是实数类型。

(2)初值、终值各是一个表达式，它们的值应对控制变量赋值相容。

(3)初值表达式、终值表达式只在开始时计算一次，循环过程中不再重新计算。 所以，假如终值表达式中有变量，而该变量在循环过程中被修改了，控制变量的终值并不因此而改变，循环次数也不受影响。

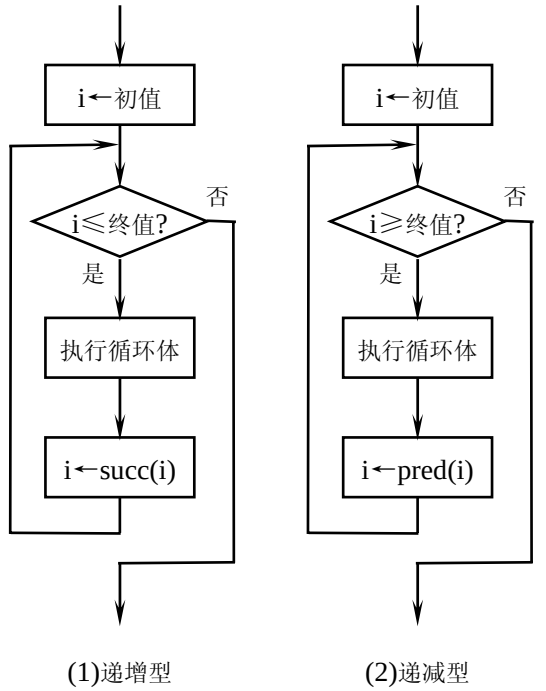


图 5.6　FOR 语句的功能流程

例如：

```
FOR i:=a TO b DO
  BEGIN
    a:=4;
    b:=6*a;
    write(i:4)
  END.
```

如果在执行 FOR 语句之前，a 和 b 的值分别为 1 和 5，那么循环次数是 5 次，不因 a 和 b 的值在循环体内改变而影响循环次数。故执行时仍输出

```
1   2   3   4   5
```

而不是改为执行 20 次循环。不过，以上程序段尽管不算错，但我们也应养成不在循环体内改变初值和终值的习惯。

(4)标准 PASCAL 规定，在结束了 FOR 语句之后，控制变量的值就变成“未定义的”了，因此在再次赋值之前原则上就不能在表达式中引用它（不过实际的 PASCAL 系统大多并不报错，也能给出一个值）。如果要再次引用该变量，可对它重新赋值。例如在同一个程序中，我们可以在几个并列的 FOR 语句中使用相同的控制变量，如

```
......
FOR i:=1 TO 10 DO 语句 1;
FOR i:=1 TO 10 DO 语句 2;
......
```

(5)在循环体中不允许给控制变量重新赋值。 注意这里的“重新赋值”不仅指使用赋值语句给它赋值，也包括广义的用任何方式（如用read语句）改变（或者更严格些，称作“危及”[①]）该变量的值。例如

```
FOR i:=1 TO 10 DO
  BEGIN
    write(i);
    i:=i*2
  END;
```

这样的语句是不合法的。

(6)控制变量只可以是整体变量。所谓整体变量， 就是直接由一个标识符标识的变量。以后我们会学到其它形式的变量，如作为数组成分的“下标变量”，以及由指针指示的变量（“标识变量”）等，这些形式的变量不可以用作 FOR 语句中的控制变量。

(7)控制变量必须在最紧包含此 FOR 语句的分程序中的变量说明部分里说明。以后我们会学到子程序（过程和函数），在子程序里面的 FOR 语句中， 只可以用该子程序中说明的局部变量作其控制变量。

5.3.3 例题

[例 1] 输入 200 个数，求其平均值。

问题分析：要求平均值，先要求其总和。200 个数输入到不同的变量里再加显然不好，变量太多，程序也太长。我们采用递推的方法。设 x_k 表示第 k 个数，s_k 表示前 k 个数的和，显然对任何的 k，都有以下关系：

$$s_k=s_{k-1}+x_k$$

而当 k=0 时，显然有 $s_0=0$。这样，我们就可以根据上面的关系由 s_0 和 x_1 求出 s_1，由 s_1 和

---

① 按《GB 7591-87》，“危及”一个变量指的是以下情况：

a. 用赋值语句给该变量赋值；

b. 调用一个子程序，用该变量作为变量参数相应的实在参数；

c. 使用 read 或 readln 语句，用该变量作为参数之一；

d. 使用一个 FOR 语句，用该变量作为控制变量。

按《GB 7591-87》规定，不仅 FOR 语句的循环体中不允许“危及”控制变量， 而且最紧包含该 FOR 语句的分程序的任何过程与函数说明部分里，也不允许“危及”该变量。

x_2求出 s_2，由 s_2和 x_3求出 s_3，……最后求出 s_{200}就是所要的总和。在这个题目中下一个 s 求出后，前面的 s 和 x 就不必保留了，所以 s_0～s_{200}可以使用同一个变量 s 表示，x_1～x_{200}可以使用同一个变量 x 表示。与第二章 2.5 节所讲的道理一样，可以编成重复结构（即循环）的算法。因为这里循环次数已定，故可以使用 FOR 语句。程序如下：

```
PROGRAM mean200(input,output);
  VAR
    x,s:real;
    k:integer;
BEGIN
  s:=0;
  FOR k:=1 TO 200 DO
    BEGIN
      read(x);
      s:=s+x
    END;
  writeln(s/200)
END .
```

这个程序里的 s:=s+x 中，左边的 s 是 s_k，右边的 s 是 s_{k-1}，而 x 是 x_k。

[例 2] 求 n!。

问题分析：这个题目与[例 1]有相似之处，[例 1]要求连加的和，这里要求连乘的积。同样，我们也要采用递推的办法。对于阶乘，有以下关系：

```
k!=(k-1)!*k;
0!=1
```

可以按这个关系由 0!求出 1!，再由 1!求出 2!，……等等。与[例 1]同样道理，用变量 p 存放 k!，可以编出以下程序：

```
PROGRAM fac(input,output);
VAR
  k,n:integer;
  p:real;
BEGIN
  write('Input n:');
  readln(n);
  p:=1;
  FOR k:=1 TO n DO  p:=p*k;
  writeln(n,'!=',p:5:0)
END.
```

这个程序里的 p:=p*k 中，左边的 p 是 k!，右边的 p 是(k-1)!。需要说明的是，这个程序里没有把 p 说明成整数，而是说明成了实数，是因为大多数实际的 PASCAL 中整数的

取值范围较小，而阶乘的结果常常很大，容易溢出。定成实数允许范围就大了。输出时第二个域宽（小数位数）定成了 0，故仍按整数的样子输出。

［例 3］输入一个大写字母，输出由 Z 开始按字母表相反的顺序到该字母为止的倾斜的字母序列。如输入为 W，则输出应为

```
Z
 Y
  X
   W
```

FOR 语句可以用任何一种顺序类型的控制变量，这一题既可以用整数又可以用字符作为控制变量。用整数的做法留给读者自己考虑，这里介绍用字符的做法。下面的程序用的是递减型的 FOR 循环。

```
PROGRAM pat1(input,output);
  VAR r,c:char;
BEGIN
  read(r);
  FOR c:='Z' DOWNTO r DO
    writeln(c:30+ord('Z')-ord(r))
END .
```

本程序中用带有变量的表达式作为输出的域宽，控制显示的形状，其中的常数 30 是为使显示处于屏幕上适当的位置而设的，根据需要可以调整。

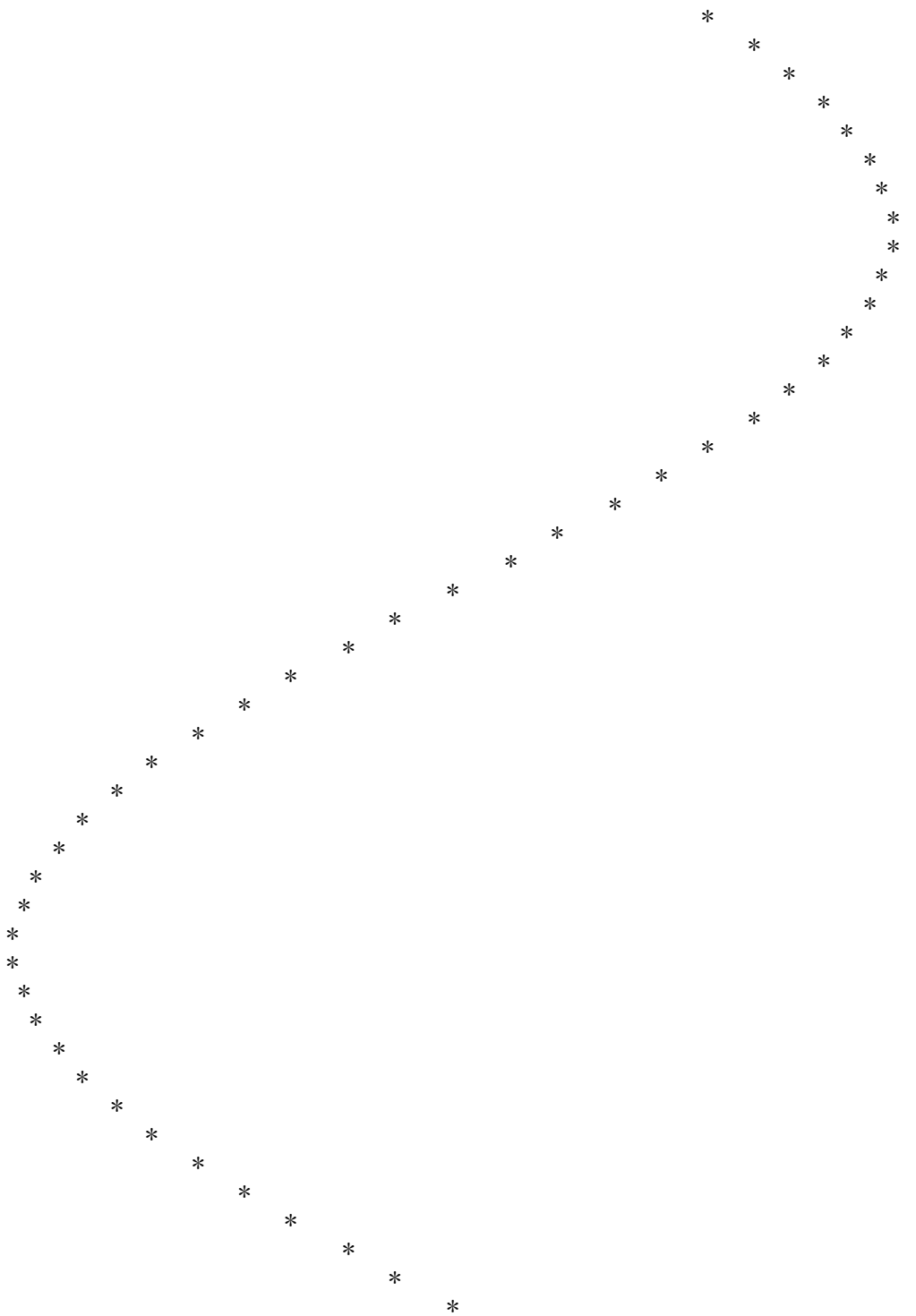
［例 4］用*号在屏幕上组成 y=sin(x)的图像。

与上题一样，可以用改变域宽的办法来控制*号的显示位置。为简单起见， 将向右的方向作为 y 的正方向，向下的方向作为 x 的正方向，y 的 0 点设在屏幕中间。采用适当的增量间隔和放大系数使图像大小合适。

```
PROGRAM pat2(output);
  VAR
    h:real;       {h 是 x 的增量间隔}
    i:integer;    {i 由 0 至 50 打出一个周期}
BEGIN
  h:=3.14159265/25;
  FOR i:=0 to 50 DO
    writeln('*':round(38*sin(i*h)+40))
END.
```

运行结果是：

```
                                    *
                                        *
                                            *
                                                *
                                                    *
```



因为域宽必须为整型，所以程序中用 round 舍入取整。舍入误差会影响图像的精度，

为减小相对误差，图像尽量放大，程序中的 38 就是我们选的放大系数。

5.4 适用于循环程序的某些实际算法

5.4.1 递推

所谓递推，就是指在一个数据序列中，下一个数据的值可以在前一个或前几个数据的基础上用一个公式（或算法）推算出来。许多实际问题都属于这一类型。如上一节的例中我们就见到如下的递推公式：

$$s_k=s_{k-1}+x_k$$

又，若用 p_k 表示 k!的值，则上一节求阶乘的例中也用到了如下的递推公式：

$$p_k=p_{k-1}*k$$

上面的例子中，递推公式只用到了前一个数据。这种情况下，只要确定了开头一个数据的值（称作初值），就可以依次推算出以后各个数据的值。如上一节[例 1]采用了 $s_0=0$ 作为初值，[例 2]采用了 $p_0=1$ 作为初值。

这两个例子的思路可以推广到任何处理累加或连乘问题的程序中。一般说来，累加问题中和的初值通常取 0，连乘问题中积的初值通常取 1。当然不这样做也可以，如累加时也可以直接取第一个加数为初值，而从第二项开始递推。这样的话，上一节[例 1]中可改成

```
.....
BEGIN
  read(s);
  FOR k:=2 TO 200 DO
    BEGIN
      read(x);
      s:=s+x
    END;
  writeln(s/200)
END .
```

这样可少作一次加法。连乘问题中也有类似处理方法。不过一般习惯仍然是分别取 0 取 1 的作法较多，因为它较简明。

[例 1]输入 n，求 1+2+...+n。

问题分析：这是一个累加问题，与上一节类似，递推公式是

$$s_k=s_{k-1}+k;$$

$$s_0=0$$

求出 s_n 作为结果。程序如下：

```
PROGRAM examp1(input,output);
  VAR
    n,s,k:integer;
```

```
BEGIN
  write('Input n :');
  readln(n);
  s:=0;
  FOR k:=1 TO n DO s:=s+k;
  writeln(s)
END .
```

[例 2]输入实数 a 和正整数 n，用连乘的办法求 a^n。

问题分析：用 p_k 来表示 a^k，与上一节类似，递推公式是

$p_k=p_{k-1}*a;$

$p_0=1$

求出 p_n 作为结果。程序如下：

```
PROGRAM examp2(input,output);
  VAR
    a,p:real;
    n,k:integer;
BEGIN
  write('Input a & n :');
  readln(a,n);
  p:=1;
  FOR k:=1 TO n DO p:=p*a;
  writeln(p)
END .
```

[例 3] 输入 200 个数，求其最大值。

问题分析：这个问题也可以采用递推的方法，只不过这里的递推关系不是一个简单的算术公式，而是一个带有逻辑判断的算法。设 x_k 表示第 k 个数，m_k 表示前 k 个数的最大值，则 m_{k-1} 与 m_k 间的递推关系是

若 $x_k>m_{k-1}$　　则　　$m_k=x_k$

　　否则　　$m_k=m_{k-1}$

可取 $m_1=x_1$ 作为初值，从第二项开始递推。最后求出 m_{200} 就是所要的总的最大值。程序如下：

```
PROGRAM examp3(input,output);
  VAR
    x,m:real;
    k:integer;
BEGIN
  read(m);
  FOR k:=2 TO 200 DO
    BEGIN
```

```
      read(x);
      IF x>m THEN m:=x
    END;
  writeln(m)
END .
```

前面的递推关系中有“否则”部分，但这个程序里的递推算法中却没有 ELSE 部分，这是因为 m_k 和 m_{k-1} 共用同一个变量存放，既然 $m_k=m_{k-1}$，赋值操作自然就不必要了。

这个问题中，一般 m_0 是没有意义的，所以通常取 $m_1=x_1$ 作为初值， 从第二项开始递推。但如果预先知道各数的取值范围的话，也可以取一个肯定不大于各 x 的数作为 m_0， 从第一项开始递推。例如，若输入的数是学生的分数，肯定非负，则这一段程序也可以为

```
......
BEGIN
  m:=-1;
  FOR k:=1 TO 200 DO
    BEGIN
      read(x);
      IF x>m THEN m:=x
    END;
  writeln(m)
END .
```

这种做法在许多情况下有它的方便之处。

这个算法中，变量 m 在一定条件下可多次重复赋值。这也就是我们在第四章 4.3 节中说的“有条件地重新赋值”的做法。

许多实际问题中，在求最大值的时候还需要记下与最大值相联系的某些信息（如需要记下最大值是第几个数据），则可以在执行 m:=x 的同时将该信息记入一个专门安排的变量中。例如，要求最大值是第几个数据，则可安排用整型变量 mv 来记录，将上述程序中的

```
      IF x>m THEN m:=x
```

改为

```
      IF x>m THEN BEGIN m:=x; mv:=k END
```

开始要给 mv 赋初值为 1（若采用上面那种以-1 为 m 初值的做法，则 mv 也可以不赋初值），最后 mv 就是最大数据的序号。完整的程序留给读者自己考虑。

[例 4] 求 $s=1/1^2+1/2^2+1/3^2+1/4^2+\ldots\ldots$，直到最后一项小于 10^{-7}，输出舍入保留 6 位小数。

问题分析：这也是一个累加问题，与前面不同的是项数在计算前不能确定，故一般不用 FOR 语句。这里采用 REPEAT 语句。程序如下

```
PROGRAM examp4(output);
  CONST
    eps=1e-7;
```

```
  VAR
    s,t:real;
    k:integer;
BEGIN
  s:=0;
  k:=1;
  REPEAT
    t:=1/sqr(k);
    s:=s+t;
    k:=k+1
  UNTIL t<eps;
  writeln(s)
END .
```

[例 5] 利用公式 e=1+1/1!+1/2!+1/3!+...计算 e 的近似值，直到最后一项小于 10^{-7}。

问题分析：这个例题看起来与上一例有些相似。但是，它的每一个加数中都有阶乘，假如在计算每一个加数（相当于上例中的 t）时都套用前面求阶乘的算法的话， 就会在循环体内嵌套循环，构成下面将讲到的多重循环。这样不仅程序复杂，而且实际上包含了大量多余的重复计算。然而我们观察其规律后会发现，这里不仅前 k 项和存在递推关系， 相邻的加数之间也有递推关系。这两个递推平行进行，就可以纳入同一个循环中。

为便于叙述，将上式开头的 1 称作第 0 项，则第 k 项为 1/k!。我们将第 0 项的 1 直接放到和的初值中，从第 1 项开始递推。

设 t_k表示上式中第 k 项，e_k表示上式中从开头到第 k 项的和，则当 k≥1 时，有以下递推关系：

$e_k=e_{k-1}+t_k$;

$t_k=t_{k-1}/k$;

初值为

$e_0=1$;

$t_0=1$;

递推的次序如下图示意：

$$\begin{array}{ccccccccccc} t_0 & \rightarrow & t_1 & \rightarrow & t_2 & \rightarrow & t_3 & \rightarrow & t_4 & \rightarrow & t_5 & \rightarrow & \cdots\cdots \\ & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \downarrow & & \\ e_0 & \rightarrow & e_1 & \rightarrow & e_2 & \rightarrow & e_3 & \rightarrow & e_4 & \rightarrow & e_5 & \rightarrow & \cdots\cdots \end{array}$$

当递推达到某个 k 使得 $t_k<10^{-7}$时，相应的 e_k就是最后的结果。

程序如下：

```
PROGRAM examp5(output);
  CONST
    eps=1e-7;
```

```
  VAR
    e,t:real;
    k:integer;
BEGIN
  t:=1; e:=1;
  k:=1;
  REPEAT
    t:=t/k;
    e:=e+t;
    k:=k+1
  UNTIL t<eps;
  writeln(e:10:7)
END .
```

运行结果为

2.7182818

上面的例子中，每一个数据序列里，递推时只用到相邻的前一个数据，新的数据求出后上一个数据就可以马上丢掉，所以一个序列前后可只用一个变量存放。而下面的例中，递推时要引用前面两个数据，这样新的数据求出后上一个数据还不能马上丢掉，只用一个变量存放就不够了。

[例 6]fibonacci 序列的前两项是 0 和 1， 且每个后继项均为前两项的和。 因此，fibanacci 序列为 0，1，1，2，3，5，8，13，21.....编写一个程序，输出 fibanacci 序列的前 20 项。

问题分析：用 f_k 表示 fibonacci 序列的第 k 项，则递推公式是：

$f_k=f_{k-1}+f_{k-2}$;

初值需要两个

$f_1=0$;

$f_2=1$。

程序如下：

```
PROGRAM fibo(output);
  VAR
    f,fa,fb,k:integer;
BEGIN
  fa:=0; f:=1;
  write(fa:10,f:10);
  FOR k:=3 TO 20 DO
    BEGIN
      fb:=fa; fa:=f;
      f:=fa+fb;
```

```
      write(f:10)
    END;
  writeln
END.
```

本程序用三个变量 f，fa，fb 来轮换存放当前用到的项。进循环体时，f 中是 f_{k-1}，fa 中是 f_{k-2}；出循环体时，f 中是 f_k，fa 中是 f_{k-1}，fb 中是 f_{k-2}。

若想只用两个变量存放当前用到的项也是可以办得到的，不过程序要复杂一些。我们知道，只用两个变量不便于交换，所以求下一项时这两个变量只能反过来用。这样，每一轮循环求出两个项。程序可以如下：

```
PROGRAM fibo(output);
  VAR
    f,fa,i:integer;
BEGIN
  fa:=0; f:=1;
  write(fa:10,f:10);
  FOR i:=2 TO 10 DO
    BEGIN
      fa:=f+fa;
      write(fa:10);
      f:=fa+f;
      write(f:10)
    END;
  writeln
END.
```

这个循环体中前一半求出的是 f_{2i-1}，后一半求出的是 f_{2i}。

5.4.2 迭代法

迭代法是解方程一类问题的一个方法。原则是针对一个方程给出一个递推公式（称作迭代公式），由此递推出来的序列的极限就是方程的一个解。在一定范围内任选一个值为初值，按公式递推到相邻两项近似相等时，就认为已求出解。例如对方程

$$x = \cos x$$

就可以按公式

$$x_k = \cos x_{k-1}$$

来迭代。同一个方程可以有多种迭代公式，其极限一样，但是逼近极限（称作收敛）的快慢不同，当然是越快越好。如上述方程也可以换用公式

$$x_k = (\cos x_{k-1} + x_{k-1})/2$$

来迭代。这个公式收敛比前一个公式更快。一般地，x_{k-1} 若是解的一个近似值， 迭代公式应能保证 x_k 是一个更好的近似值，其误差与 x_{k-1} 的误差的比越接近于 0 越好。

迭代法常常用“相邻两次的值接近到一定程度”来作为结束条件，为此有两种办法：

一种办法是在循环体中先求出相邻两次的值的差；另一种是新的值求出时上一个值不要丢掉，另外保存以备判断其差。这两种办法在 5.2 节[例 1]已经见过了。

公式设计得不好或初值选得不合适有可能所得序列没极限（称作发散）。也有的时候，选用不同的初值可以得到不同的解。例如 5.2 节[例 1]是针对方程

$x^2-a=0$

的迭代公式。该方程有两个解：$\pm\sqrt{a}$。初值选正数得到$+\sqrt{a}$，初值选负数得到$-\sqrt{a}$。

如何设计迭代公式及如何选定初值的理论是另一门课《计算方法》的内容。本课要求能做到的是：告诉你设计迭代公式及选定初值的办法后要能够编出程序来。

牛顿迭代法是设计迭代公式方法的一种，其特点是收敛快。设方程为

$f(x)=0$

若近似解 x_e 与准确解 x 差别不大，则根据导数的定义，应有

$$\frac{f(x_e)-f(x)}{x_e-x}\approx f'(x_e)$$

考虑到 f(x)=0，故可这样估算较准确的解：

$x\approx x_e-f(x_e)/f'(x_e)$

将这个关系当成递推关系，这就是牛顿迭代法：

$x_k=x_{k-1}-f(x_{k-1})/f'(x_{k-1})$

前面 5.2 节[例 1]的公式就是由此推导出来的。

[例 7]用牛顿迭代法求 a 的三次方根。

问题分析：这里是解方程

$x^3-a=0$

设 $f(x)=x^3-a$，则 $f'(x)=3x^2$，可写出牛顿迭代公式

$x_k=x_{k-1}-(x^3_{k-1}-a)/(3x^2_{k-1})$

$=x_{k-1}+(a/x^2_{k-1}-x_{k-1})/3$

与前面 5.2 节[例 1]相似，后一项就是迭代差，程序中用 delta 存放。程序如下：

```
PROGRAM cuberoot(input,output);
  CONST
    eps=1E-5;
  VAR
    a,x,delta:real;
BEGIN
  write('Input a:');
  readln(a);
  x:=1;
  REPEAT
    delta=(a/sqr(x)-x)/3;
    x:=x+delta
```

```
    UNTIL abs(delta)<eps;
    writeln(x:6:4)
END.
```

5.4.3 尝试法

尝试法又称枚举法或穷举法，就是将可能范围内的所有数据一一检验，找到满足一定条件的值，或判断有没有符合条件的值。尝试法的先决条件是可能范围内的所有数据个数有限。尝试法虽然算是“笨办法”，但许多实际问题没有任何别的办法时这是唯一有效的办法。而且，尝试法程序通常比较容易设计，直接按定义或按题意描述就可以编出。

具体做法视题目要求而不同。有时只要求出任意一个解，这时循环中一旦找到符合条件的数据后就可以结束循环。有时要求找到最大的（或最小的）符合条件的值，这时就应适当安排尝试的次序。如 5.1 节[例 1]就必须由大到小尝试。有时要求找到所有符合条件的值，就应将尝试进行到底，此时常采用 FOR 循环。

[例 8] 求出所有的“水仙花数”。所谓水仙花数，是指一个三位数，其各位数字的立方和等于该数本身。例如：153 是一个水仙花数，因为 $153=1^3+5^3+3^3$。

问题分析：尝试的范围是 100～999 的整数。为进行判断，需要将该数的三个数字分开来。下面程序中分别用 a，b，c 表示百位，十位，个位。

```
PROGRAM examp8(output);
  VAR
    i,a,b,c:integer;
BEGIN
  FOR i:=100 TO 999 DO
    BEGIN
      a:=i DIV 100;
      b:=(i MOD 100) DIV 10;
      c:=i MOD 10;
      IF i=a*a*a+b*b*b+c*c*c THEN write(i:5)
    END;
  writeln
END .
```

有时要求判断有没有符合条件的值，这有多种做法。如可以设一个布尔变量作为标志，在循环前令其初值为真，循环中一旦遇到符合条件的值就将该标志改为假。这样循环结束后该标志就可以代表“没有符合条件的值”的判断真假。

[例 9] 判断输入的正整数 n 是否素数。

问题分析：按定义，这里是要判断 2～n-1 间有没有能整除 n 的数。用布尔变量 f 作为上述标志，在循环结束后表示“是素数”的判断结果。

为了缩短时间，尝试的范围可以由 2～n-1 缩小为 2～$\sqrt{n}$。理由是：如果 n 不是素数，它至少应有两个非 1 的因子，其中必有一个不大于 $\sqrt{n}$。故只要 2～$\sqrt{n}$ 没有 n 的约数，则

n 就一定是素数。

```
PROGRAM prime0(input,output);
  VAR
    n,i:integer;
    f:boolean;
BEGIN
  write('Input n: ');
  readln(n);
  f:=true;
  FOR i:=2 TO trunc(sqrt(n)) DO
    IF n MOD i=0 THEN f:=false;
  IF f THEN writeln('Yes.')
       ELSE writeln('No.')
END .
```

注意 sqrt(n)是实型的表达式，对 i 不赋值相容，故要取整。

本来，在循环过程中，只要遇到一个能整除的数，就可肯定不是素数，尝试没有必要进行完，但上述程序却一定要进行到 trunc(sqrt(n))才结束。为了缩短时间，还可以改进，使它在遇到能整除的数时立即结束循环。为了在下文能够知道是怎样退出循环的，标志 f 还需要保留。这样就不能用 FOR 语句了。这个方案如下：

```
PROGRAM prime0(input,output);
  VAR
    n,i:integer;
    f:boolean;
BEGIN
  write('Input n: ');
  readln(n);
  f:=true;  i:=2;
  WHILE i<=sqrt(n) AND f DO
    BEGIN
      IF n MOD i=0 THEN f:=false;
      i:=i+1
    END;
  IF f THEN writeln('Yes.')
       ELSE writeln('No.')
END .
```

不过，这个方案在 n 不太大时不一定比上一个方案快。 这主要是因为终值表达式里有开方运算，前面讲过，FOR 语句中终值表达式只在开始计算一次，循环中就不再算了，而 WHILE 语句中的布尔表达式循环时要反复计算。开方又是一种较慢的运算。必要时，这一点还可

以改进，将开方运算移到循环之外。请读者自行考虑。

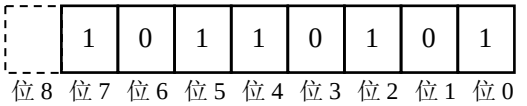
尝试法的一个缺点是对有些问题花费的运行时间较多。有些实际问题，若用尝试法，理论上估算其运行时间达到天文数字，这就在实际上是不可行的了。

5.4.4 其它例题

[例 10] 将输入的十进制非负整数 n（小于 256）以 8 位二进制形式输出。

问题分析：关于十——二进位制转换，我们过去学过“除以 2 取余”的算法， 但那样先得出最低位，与输出次序相反，不便处理。现在用另一种方法：“乘以 2 取整”的方法， 原理如下：

将 n 的 8 位二进制表示中的最低位称作位 0，最高位称作位 7。如下（图中以 181 为例）：



更高的位（图中虚线所标）称作位 8。可以设想，如果能将它左移一位，原位 7 就移到了位 8。此时再将位 8 去掉，剩下的就又成了不超过 8 位二进制的数了。若再左移一位，则此前的位 7 即开始的位 6 就移到了位 8。此时再将位 8 去掉，剩下的就又成了不超过 8 位二进制的数了。如此重复 8 次，若每次都将移到位 8 的数输出，所得的输出序列恰好就是该数的二进制形式。

由此可得算法（伪代码形式）如下：

```
输入 n;
FOR i:=1 TO 8 DO
  BEGIN
    使 n 的二进制形式左移一位;
    IF n 的二进制位 8 是 1
      THEN
        BEGIN
          将 n 的二进制位 8 的 1 去掉;
          输出'1'
        END
      ELSE 输出'0'
  END。
```

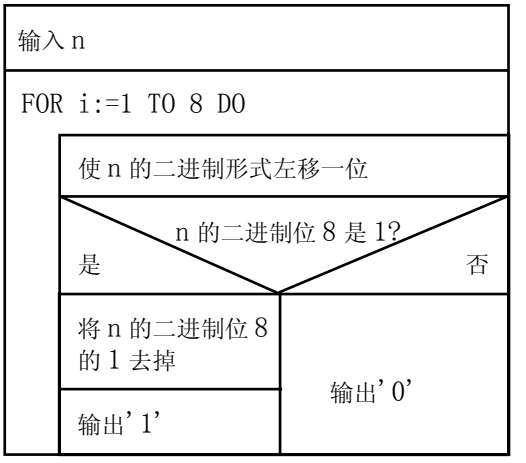


图 5.7 [例 10]的粗算法结构框图

也可以画成结构框图（图 5.7）。

下面进一步对其细化：

▲要使其二进制形式左移一位，只要将原数乘以 2 即可。

▲要判断其二进制形式中位 8 是 1 还是 0，只要看原数是否不小于 256（即 2^8）即可。

▲要将位 8 的 1 去掉，只要将原数减去 256 即可。

这就是所谓的“乘以 2 取整”，这里的“整”是将 2^8 看成一个整数。 将这些代入前面的算法中即可得出程序：

```
PROGRAM  dtob(input,output);
CONST
  m=256;
VAR
  n,i:integer;
BEGIN
  REPEAT
    write('Enter an integer number within [0～255] : ');
    readln(n)
  UNTIL (n>=0) AND (n<256);
  FOR i:=1 TO 8 DO
    BEGIN
      n:=n*2;
      IF n>=m
        THEN
          BEGIN
            n:=n-m;
            write('1')
          END
        ELSE
          write('0')
    END
END.
```

这个例子里输入 n 的操作也编成了一个循环， 是为了在一旦输入不合要求时作出处理。这里顺便介绍一下输入不合要求时的一般处理方法。

一般程序中，输入不合要求时的处理方法常见的有三种。

一种是程序中不作特别处理，编程时假定输入一定合要求。这样，一旦输入不合要求时会出现不一定什么样的错误。但只要操作者输入合要求，结果就是对的。这样做对操作者要求较严，对编程者要求较宽，因而程序较简单。本书中为了缩减篇幅，大多数例题都采用这种方法。

第二种方法是一旦输入不合要求就结束程序。本书第四章 4.3 节[例 7]就是如此，[例 8]也属于这种方法。[例 8]在结束程序的同时还给出报告信息，让操作者知道结束程序的原因。

第三种方法是在一旦输入不合要求时，程序要求操作者重新输入，直到输入合要求时为止。这种方法比较实用，通常较大的应用程序都采用这种方法，上面例题中用的就是这种方法。

5.5 多重循环

多重循环是指循环的嵌套。如果一个循环的循环体中不再包括循环，这个循环习惯上称作一重循环；循环体中包括一重循环的循环，习惯上称作二重循环；包括二重循环的，习惯上称作三重循环；等等。这只是习惯，不是严格的概念。

[例 1] 要编程输出一个九九乘法表，要求格式如：

```
*   1   2   3   4   5   6   7   8   9
1   1
2   2   4
3   3   6   9
4   4   8  12  16
5   5  10  15  20  25
6   6  12  18  24  30  36
7   7  14  21  28  35  42  49
8   8  16  24  32  40  48  56  64
9   9  18  27  36  45  54  63  72  81
```

这个表中最上面的一行是上表头。下面的 9 行分别称作第 1 行～第 9 行， 每行左边是左表头，然后是内容。这个程序可以用伪代码写出提纲：

```
输出上表头;
FOR i:=1 TO 9 DO 输出第 i 行。
```

其中，“输出第 i 行”可以细化为：

```
输出左表头 i;
FOR j:=1 TO i DO 输出 i*j ;
换行。
```

这两层提纲套在一起可以画出如图 5.8 的结构框图。

图 5.8 [例 1]算法的结构框图

还可以细化，如“输出上表头”可以细化，这些不再详述。程序如下（左边的框线是为了容易看清层次而标上的）：

```
PROGRAM cf(output);
VAR
  a,i,j:integer;
BEGIN
  write('*':5);
  FOR i:=1 TO 9 DO write(i:4);
  writeln;
  FOR i:=1 TO 9 DO
 ┌──BEGIN
```

```
|     write(i:5);
|     FOR j:=1 TO i DO
|     ┌ BEGIN
|     |    a:=i*j;
|     |    write(a:4)
|     └ END;
|     writeln
└──END
END.
```

该程序中，前三行是输出上表头。后面的语句采用了一个二重 FOR 循环， 外循环的控制变量为 i，i 值每变化一次，就打印出一行，在外循环体内又包含了内循环，内循环体每次计算和输出一个乘积。

循环的嵌套，当然要满足一般基本结构嵌套的原则。结构间只能嵌套不能交叉。这一点，只要采用 PASCAL 中的基本构造语句来编程，自然就能满足，即使有意想使结构交叉也无法写出来。

另外，若用 FOR 语句，还需注意，内外循环的控制变量不允许同名，这是由前面 5.3 节的第(5)条规定了的。但是，前面讲过，当一个循环结束后，其控制变量就变得没有意义， 可以为它重新赋值，所以，当外循环体内有多个内循环时，同一层次上的内循环语句的控制变量允许同名。例如：

```
FOR i:=1 TO n DO
┌BEGIN
|  ......
|  FOR j:=1 TO n DO
|  ┌ BEGIN
|  |    ......
|  |    FOR k:=1 TO n DO 语句;
|  |    ......
|  └ END;
|  ......
|  FOR j:=1 TO n DO
|  ┌ BEGIN
|  |    ......
|  └ END;
|  ......
└END;
```

一、二、三层的 FOR 语句的控制变量不允许同名，但是第二层上的两个 FOR 语句的控制变量允许同名。

如果不用 FOR 语句而用 WHILE 或 REPEAT 构成计数循环，则从语法上说内外层的控制变量并不禁止同名。但我们编程时也不应该使它同名，因为一旦同名，该变量的变化情况

将变得异常复杂，其功能就不再是普通的计数循环了。

内外层控制变量不得同名，但外层控制变量的值在内层的表达式中的引用，包括用在内层循环的初值和终值表达式中，却都是可以的，而且是经常出现的。如上面的例题。

[例 2] 求 2 到 100 之间的素数。

问题分析：判断一个数 n 是否为素数的算法在上一节的[例 9]已经作过，原则上只要将该算法作为内层，而外层令 n 由 2 至 100 构成一个 FOR 循环即可。不同点是：上一节的例题判断后输出“是”或“否”，而这里则是素数就输出该数，不是就不输出。程序如下，其中内层算法采用上一节的[例 9]的第一个程序。

```
PROGRAM prime(output);
VAR
  n,i:integer;
  f:boolean;
BEGIN
  FOR n:=2 TO 100 DO
    BEGIN
      f:=true;
      FOR i:=2 TO trunc(sqrt(n)) DO
        IF n MOD i=0 THEN f:=false;
      IF f THEN write(n:5)
    END;
  writeln
END.
```

[例 3] 求出所有的“水仙花数”。换用多重循环方法作。

问题分析:上一节[例 8]已经作过这个题目,采用的是尝试法,尝试的范围是 100～ 999 的整数。这里仍采用尝试法，但将百位，十位，个位数字分别作为控制变量，用三重循环尝试。请读者将这里的做法与上一节[例 8]作一对比。

```
PROGRAM examp8(output);
  VAR
    a,b,c:integer;
BEGIN
  FOR a:=1 TO 9 DO
    FOR b:=0 TO 9 DO
      FOR c:=0 TO 9 DO
        IF a*100+b*10+c=a*a*a+b*b*b+c*c*c THEN write(a:3,b,c);
  writeln
END .
```

[例 4] 把一张一元钞票换成一分，二分和五分的硬币，问有哪些换法。

问题分析：设有一分硬币 a 枚，二分硬币 b 枚，五分硬币 c 枚。只要满足条件

a+b*2+c*5=100

且 a，b，c 均非负，即为一种换法。

这一题可以用尝试法。要将 a，b，c 的各种组合尝试遍，需要多重循环。 首先要确定尝试范围，范围可大而不可小，粗略分析后可取

0≤a≤100；

0≤b≤50；

0≤c≤20。

程序可以如下：

```
PROGRAM yb(input,output);
  VAR
    a,b,c:integer;
BEGIN
  FOR c:=0 TO 20 DO
    FOR b:=0 TO 50 DO
      FOR a:=0 TO 100 DO
        IF a+b*2+c*5=100 THEN writeln(a:5,b:5,c:5)
END.
```

这个程序很简单，已是可行的了。但尝试范围较大，故运行较慢，可以改进。

这个程序中 b 的循环是 c 的循环的内层循环体，b 的循环过程中 c 是不变的，而 c 确定时 b 就可以确定一个更小的范围。对 a 更是如此，b，c 确定后 a 只可能有一个取值或没有，而且 b，c 只要不过大 a 就一定有一个。这样考虑后范围可取

0≤c≤20；

0≤b≤(100-5c)/2；

a＝100-5c-2b。

因为 100-5c 有两处用到，程序中将它放到一个变量 m 中。程序改为

```
PROGRAM yb(input,output);
  VAR
    a,b,c,m:integer;
BEGIN
  FOR c:=0 TO 20 DO
    BEGIN
      m:=100-c*5;
      FOR b:=0 TO m div 2 DO
        BEGIN
          a:=m-b*2;
          writeln(a:5,b:5,c:5)
        END
    END
```

```
END.
```

由于 a 只须确定一个数，不必再尝试，故变成了二重循环。

［例 5］打印杨辉三角形

```
                    1
                  1   1
                1   2   1
              1   3   3   1
            1   4   6   4   1
          1   5  10  10   5   1
        ......................
    1 10 45  ..............   45 10  1
```

问题分析：杨辉三角形中，每个数据值都可以用组合 c_m^n 来计算。上列杨辉三角形可以写成：

$$
\begin{array}{ccccccccccc}
 & & & & & c_0^0 & & & & & \\
 & & & & c_1^0 & & c_1^1 & & & & \\
 & & & c_2^0 & & c_2^1 & & c_2^2 & & & \\
 & & & \cdots & \cdots & \cdots & \cdots & \cdots & & & \\
c_{10}^0 & c_{10}^1 & c_{10}^2 & \cdots & \cdots & \cdots & \cdots & \cdots & c_{10}^9 & c_{10}^{10} &
\end{array}
$$

这里

$c_m^n=m(m-1)(m-2)...(m-n+1)/n!$

m=1，2，...，10

n=1，2，...，10

$c_m^0=1$，m=0，1，2，...，10

可以由递推公式计算 c_m^n

$c_m^0=1$　　m=0，1，2，...，10

$c_m^n=c_m^{n-1}*(m-n+1)/n$　　n=1，2，3，...，m

为了输出三角形，我们将第一行一个数据的域宽定成 40，大约处于屏幕正中。每一行的第一个数比上一行往左移 3 格。其它每个数域宽规定成 6。这是一个二重循环问题，外层控制求出 11 个行（m 为 0～10），内层是在每一行中用递推法求出各数。

```
PROGRAM yanghui(output);
  VAR
    m,n,c:integer;
BEGIN
  FOR m:=0 TO 10 DO
    BEGIN
      c:=1;
      write(c:40-3*m);
```

```
    FOR n:=1 TO m DO
      BEGIN
        c:=c*(m-n+1)div n;
        write(c:6)
      END;
    writeln
  END
END.
```

[例 6] 键盘输入 8 组实数，每组 10 个，程序将每组 10 个加成一个和，将这 8 个和乘在一起，输出其最后的积。

问题分析：这是一个两层的递推问题。外层是一个连乘，在上一节我们已考虑过它的递推关系，其算法是：

```
m:=1;
FOR i:=1 to 8 DO
  BEGIN
    输入一组数据并求出其和→s;
    m:=m*s
  END;
输出 m。
```

其中内层“输入一组数据并求出其和→s”的操作是一个累加，同样， 在上一节我们已考虑过它的递推关系，其算法可以是：

```
提示“请输入 10 个数”;
s:=0;
FOR j:=1 TO 10 DO
  BEGIN
    输入一个数 x;
    s:=s+x
  END;
readln
```

其中开始的提示及最后的 readln 也可以没有，但有了更好，可以减少输入数据的分组的混乱。将这两层算法套起来就可以编出整个的程序：

```
PROGRAM result(input,output);
  VAR
    i,j:integer;
    x,m,s:real;
BEGIN
  m:=1;
  FOR i:=1 to 8 DO
```

```
    BEGIN
      writeln('Input 10 numbers:');
      s:=0;
      FOR j:=1 TO 10 DO
        BEGIN
          read(x);
          s:=s+x
        END;
      readln;
      m:=m*s
    END;
  writeln('Result:',m)
END .
```

请读者注意这几个例子中初始值的处理。[例 6]中，m 的初值必须放在外层循环之外，这点与上一节的道理是一样的。同样 s 的初值也必须放在内层循环之外。但一定要注意，s 的初值必须放在外层循环之内，而不能放在外层循环之外。这是因为，每一组求和都要从头递推，所以内层循环每次开始前都须初始化。[例 5]中 c 的初值，[例 2]中 f 的初值等也是同样道理。[例 5]和[例 2]外层不是递推算法，没有初值。

除了初始值的处理外，如[例 6]中输出提示的位置，readln 的位置， 以及输出最后结果的位置等都是需要注意的。其实弄清这些问题，只要掌握好两点就应该没有问题了：

第一，掌握好一重循环的递推程序，如上一节的例题；

第二，掌握好算法的嵌套，如[例 6]中我们分别编出两层算法， 将内层算法套入到外层算法中去。

这些例子中的处理方法不是机械的规定。某些特殊例子中，有可能将两重递推简化成一重，初值的处理就和上例不同了。如下面的例子。

[例 7] 求 $\sum_{i=0}^{20}(\sum_{j=0}^{i}\sqrt{i^2+j^2})$

问题分析：这个例子题意上是二重累加，我们可以仿照上例中的方法编成一个二重的递推算法。但我们这里不这样编，而是在此基础上进一步简化。因为这两重都是加法，若将它的计算写成一个长式子，是这种形状：

...+(...+...+.....)+(...+...+.....)+...+(...+...+.....)+...

然而我们知道，加法满足“结合律”，上式中的括号可以去掉，改成一重的累加，不影响结果。一重的累加我们已经比较熟悉：开始设一次初值 0， 循环体中求出一项加到累加器变量上即可。不过因为每一项的取值随 i，j 的两重组合而变化，所以程序形式上还要编成二重循环，但递推计算只在最内层进行。

```
PROGRAM sum(output);
  VAR
```

```
    i,j:integer;
    s:real;
BEGIN
  s:=0;
  FOR i:=0 TO 20 DO
    FOR j:=0 TO i DO
      s:=s+sqrt(sqr(i)+sqr(j));
  writeln('Sum=',s)
END .
```

与前面的例子相比，这里内层与外层之间没有了赋初值，也没有了外层的递推计算。

凡是满足结合律的运算都可以如此简化，例如两重连乘的问题也可以这样做。还有，求最大值也是一种满足结合律的运算，如求一个矩阵各元素的最大值，先求每行的最大值，再求各行最大值的最大值，与不分行，求总的最大值，结果是一样的，所以也可以这样处理。

[例 8] 输出如下规律的图形，其右下角的字母由键盘输入：

```
          A
        A B
      A B C
    A B C D
  A B C D E
    B C D E
      C D E
        D E
          E
```

程序如下：

```
PROGRAM pat2(input,output);
  VAR ch,c1,c2:char;
BEGIN
  read(ch);
  FOR c1:='A' TO ch DO
    BEGIN
      write(' ':40-ord(c1)+ord('A'));
      FOR c2:='A' TO c1 DO write(c2:2);
      writeln
    END;
  FOR c1:='B' TO ch DO
    BEGIN
      write(' ':40+ord(c1)-ord('A'));
```

```
        FOR c2:=c1 TO ch DO write(c2:2);
        writeln
    END
END .
```

5.6 转向语句（GOTO 语句）

转向语句又称 GOTO 语句，它可以将程序流程转到指定的语句标号所标志的语句。例如

```
GOTO 4
```

其作用是转向带有标号 4 的带标号语句。下面先介绍一下标号和带标号的语句。

5.6.1 标号和带标号语句

带标号语句由普通的简单语句或构造语句前面加一个标号和一个冒号构成，例如

```
4: a:=sin(x)
```

这就是一个带标号的语句，它前面的 4 就是标号。 带标号语句本身的功能和不带标号的语句一样，标号只是为了把程序流程从其它地方转到本语句而设的标志。标号是范围为 0～9999 的一个无符号整数。标号必须在程序的标号说明部分说明后才允许在带标号语句中引用。标号也有作用域，具体的规定和标识符的作用域一样，见第八章。标号说明部分位于分程序中说明部分的最前面，在常量定义之前。标号说明部分的语法图如图 5.9。例如

```
LABEL 4;
```

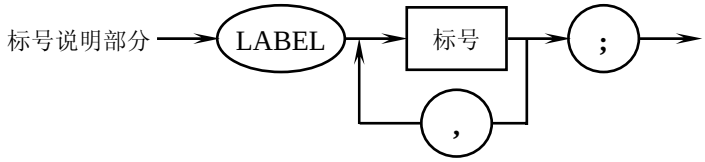


图 5.9 标号说明部分的语法图

又如

```
LABEL 4,3,10;
```

后者同时说明几个标号，其间用逗号隔开。

程序中若有多个标号，这些标号无大小先后之分。

分程序的标号说明部分中若说明了某个标号，则该分程序的语句部分中此标号必须作为一个带标号语句中冒号前的标号出现一次且只能这样出现一次。但标号作为 GOTO 语句中 GOTO 之后的部分，在程序中的引用次数不限。

5.6.2 GOTO 语句

GOTO 语句的格式如下：

```
GOTO 标号
```

结构化程序设计反对滥用GOTO语句。但是若完全禁止用它，会失去灵活性。所以PASCAL中一方面提供了各种构造型语句，为不用 GOTO 语句编程提供方便，同时还保留了 GOTO 语句以备特殊情况下的需要。为防止滥用，PASCAL 中规定 GOTO 语句除可以在同层次的语句序列中转向以外，只可以由结构内转向结构外，不可以由结构外转向结构内。这里“结构内”是指构造型语句的内嵌语句。如 IF 语句的分支内，或循环类语句的循环体内，等等。

GOTO 语句常用作 IF 语句的内嵌语句，如作为其 THEN 子句，形成条件分支。再和无条

件的 GOTO 语句配合，就可以组成各种基本结构。如 5.1 节[例 1]中用辗转相除法求最大公约数的程序可以改成

```
PROGRAM gcd(input,output);
  LABEL
    1,2;
  VAR
    r,m,n:integer;
BEGIN
  write('Input m and n: ');
  readln(m,n);
1:IF n=0 THEN GOTO 2;
  r:=m mod n;
  m:=n;
  n:=r;
  GOTO 1;
2:writeln(m)
END.
```

这里用一个嵌在 IF 语句内的 GOTO 语句和一个无条件的 GOTO 语句构成了一个当型循环。又如 5.2 节[例 1]中开平方的第一个程序可以改为

```
PROGRAM squareroot(input,output);
  LABEL 1;
  CONST
    eps=1E-5;
  VAR
    a,x,delta:real;
BEGIN
  write('Input a:');
  readln(a);
  x:=1;
1:delta=(a/x-x)*0.5;
  x:=x+delta;
  IF abs(delta)>=eps THEN GOTO 1;
  writeln(x:6:4)
END.
```

这里用一个嵌在 IF 语句内的 GOTO 语句构成了一个直到型循环。注意原来 WHILE 语句和 REPEAT 语句中的条件都是程序直接下行的条件，而这里 IF 语句中的条件则是转走的条件，所以现在的条件和原来相反。

但是，因为 PASCAL 中已经有了各种构造型语句，上述例子中的做法已没有实际意义。

所以这些例子并不是 GOTO 语句的典型应用。GOTO 语句的应用只是在少数特殊情况下才有意义。

一种情况是有时采用非结构化跳转可以显著缩短程序运行时间。

[例 1] 采用非结构化跳转提高 5.5 节[例 2]（求 2 到 100 之间的素数）程序的效率。

问题分析：原程序内层采用的是 5.4 节[例 9]的第一个程序的算法，这个算法的缺点是认定 n 不是素数以后循环还要进行到底，这就是多余的操作。现在可以在一旦认定 n 不是素数时用 GOTO 语句跳出内层循环体，内循环可以不必进行到底。另外，由于可以一次跳到 write 语句之后，不是素数就不输出，所以 f 标志也不必要了。

```
PROGRAM prime(output);
  LABEL 1;
  VAR
    n,i:integer;
BEGIN
  FOR n:=2 TO 100 DO
    BEGIN
      FOR i:=2 TO trunc(sqrt(n)) DO
        IF n MOD i=0 THEN GOTO 1;
      write(n:5);
    1:
    END;
  writeln
END.
```

注意这里标号 1 后是空语句，故其前 write(n:5)后的分号不可少，这在 3.1 节已经讲过。这里 GOTO 语句由内循环体中跳出到内循环外，外循环内，符合只可由内到外不可由外到内的原则。

5.4 节[例 9]的第二个程序的算法虽然也是一种改进方案，但比较后就会发现这里的方案操作更简单得多。应注意，这个方案的内层结构已不是结构化方法所推荐的基本结构了，所以我们将它称作非结构化的退层。

前面讲 FOR 语句时曾说过：在结束了 FOR 语句之后，控制变量的值就变成“未定义的”了。那里说的是“结束”，而如果用 GOTO 语句从中跳出，则该 FOR 语句并没有执行结束， 控制变量的值仍有意义，需要时可以在表达式中引用它。

另一种情况是：前面第四章 4.3 节最后曾说到过，将复杂条件分解， 化为多分支的程序有时可以回避错误。IF 语句遇到这种情况比较好分解，但 WHILE 语句或 REPEAT 语句分解后就会变成非标准结构，单用 WHILE 语句或 REPEAT 语句很不好表达，必须加上 GOTO 语句。 例如，有

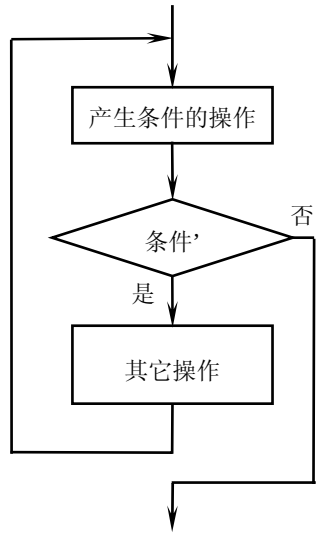


图 5.10 WHILE 型循环的一种变型

这样一段程序，功能要求为：反复输入 m 和 n，直到 n 是 0 或者 m 是 n 的倍数。按题意我们可能会编出

```
REPEAT
  readln(m,n)
UNTIL (n=0) OR (m MOD n = 0)
```

但是，与 4.3 节最后的例一样，一旦 n 是 0，则在计算 m MOD n 时就会因除数为 0 而溢出。为此，可以改成

```
REPEAT
  readln(m,n);
  IF n=0 THEN GOTO 1
UNTIL m MOD n = 0;
1:
```

于是当 n 是 0 时直接跳出，不必计算 m MOD n，问题就解决了。

还有，前面 5.2 节曾提到过，WHILE 型循环中因第一次判断的条件必须在循环之外先产生，有时给编程带来不便。为此，有人提议增加一种基本结构（可以算是 WHILE 型循环的一种变型），流程图如图 5.10。

但是，PASCAL 中没有直接描述这种结构的语句，如想构成这种循环，也只能使用 GOTO 语句。

不过，总的说来，适于使用 GOTO 语句的情况很少，初学者可以尽量不用 GOTO 语句。

习题

5.1 试将 5.4 节[例 4]的程序用 WHILE 语句改写。

5.2 试将 5.4 节[例 5]的程序用 WHILE 语句改写。

5.3 改写 5.4 节[例 3]的程序，使它不仅给出最大值，而且给出该值是输入的第几个。

5.4 改写 4.4 节[例 2]的程序,使它在输入不是小于 50 的非负数时自动要求重新输入，直到符合要求才开始计算。

5.5 用统计的办法求 sin(x)的平均值。令 x 在 0 ～ π 变化，间隔 0.001。

5.6 改写 5.4 节[例 9]的第二个程序，将开方计算移到循环之外以提高效率。

5.7 将上一题的程序作为内层算法去改写 5.5 节[例 2]的程序。

5.8 设 i，j 都是整数，取值范围都是 0～100，编程确定

$$f(i,j)=\sin(0.031i+0.02j)+2\cos(0.02i+0.031j)$$

当 i，j 各取几时 f(i,j)有最大值，该最大值是多少？

5.9 编程检验哥德巴赫猜想。即对输入的大偶数，由计算机将其分成两个素数的和（提示：用尝试法）。

第六章　枚举类型和子域类型

预定义的类型数据中，除了正文文件类型外，前面已经都介绍到了。从这一章开始，我们将陆续接触到各种由用户根据需要自己定义的数据类型。这一章介绍的枚举类型和子域类型就是两类由用户自定义的简单数据类型。

在此之前，我们先介绍一下 PASCAL 中自定义类型的一般方法。

6.1　定义新类型的一般方法

第二章 2.4 节我们已经介绍了数据类型的概念。除了使用预定义的类型外，PASCAL 中还提供了自定义新类型的办法，如

```
(sun,mon,tue,wed,thu,fri,sat)
1..100
```

等等，前者描述了一种以七个符号为值域的枚举类型，后者描述了以 1 至 100 的整数为值域的子域类型。各种描述格式将在以后章节中介绍。

使用自定义的新类型一般有两种办法，一种是先定义一个代表该新类型的标识符（即类型名），以后就以该名表示该类型，写在使用该类型的地方。定义类型标识符的格式就称作“类型定义”，形式是

```
标识符=类型表记符;
```

等号右面的“类型表记符”可以是规定的描述新类型的格式，左面的“标识符”就是我们新定义的类型名。这种“类型定义”应该写在以 TYPE 开头的“类型定义部分”。一个分程序中只能有一个“类型定义部分”，但一个类型定义部分中可以写多条“类型定义”。

例如

```
TYPE
  wdaytype=(sun,mon,tue,wed,thu,fri,sat);
  age=1..100;
```

其中，TYPE 是 PASCAL 的类型定义特定符号，只出现在程序的说明部分，标志着“类型定义部分”的开始。在这个类型定义部分中定义了一个名叫 wdaytype 的类型，它的值域为括号中的 sun，mon，tue，wed，thu，fri，sat，另一个类型名为 age，它的值域为 1～100，定义了类型名后，即可定义属于该类型的变量，例子如：

```
VAR
  day:wdaytype;
  mansage:age;
```

其中的 wdaytype 和 age 就是在前面 TYPE 中定义了的类型标识符。day 为 wdaytype 类型的变量，它的值只能取 sun，mon，tue.....sat 中的一个，而 mansage 为 age 类型变量，只能取 1 至 100 中的一个数，如果超出这个范围，则算是出错。

类型定义的等号右面的类型表记符也允许是已有的（已定义过的或预定义的）类型名。

如果那样写就表示新定义的标识符和原有的类型名代表同一类型。例如

```
TYPE
  int=integer;
```

作了这样的定义后，程序中就可以用 int 充当 integer 的别名。

使用自定义新类型的另一种办法是不定义新的类型名，而直接将规定的描述新类型的格式写在使用该类型的地方（如写在变量说明的冒号后）。例如

```
VAR
  day:(sun,mon,tue,wed,thu,fri,sat);
  mansage:1..100;
```

这里所说的“使用该类型的地方”，除变量定义中以外，还包括以后将讲到的数组类型描述格式中的“下标类型”、集合类型描述格式中的“基类型”，数组、文件、记录类型的新类型描述格式中的“成分类型”等等，这些地方都有两种写法，一是写已有的类型名，二是写新类型描述格式。这两种写法在语法上就统称为“类型表记符”。本书中，凡称“类型表记符”的地方指允许这两种写法，而称“类型标识符”的地方则只可写类型名，不可写新类型描述格式。应注意，类型名是标识符，故一般应遵循先定义后引用的原则。

有关的语法图见图 6.1。

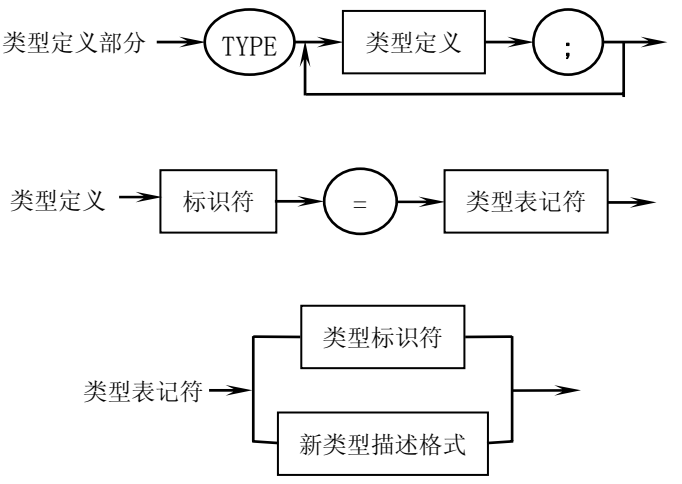


图 6.1　类型定义部分的语法图

由于五种预定义类型使用频繁，所以 PASCAL 系统对其进行了隐含的定义，用户可直接使用。如果要用到五种预定义类型以外的类型，PASCAL 并没有提供现成的定义，用户必须自己去定义。

有些初学者往往分不清类型名和变量名之间的区别，请大家在学习中务必留意。

在刚才我们所举的例子中，wdaytype 为枚举类型，age 为子域类型， 下面将对它们逐一进行讨论。

6.2　枚举类型

6.2.1 枚举类型的引入

计算机不仅能处理数值信息，还可以处理离散的非数值信息。如姓名、性别、职务、颜色、型号、状态等。要处理这种信息当然就需要有表达这种信息的形式。汉语中用“运筹”一词表示分析情况作出决策。古代“筹”字的原意是一种竹片。“运筹”就是摆弄竹片，这些竹片就是所处理的信息的代表。同样，计算机中也需要规定一组记号来充当“筹”，

即代表所处理的信息。换句话说，即作为某种非数值性的变量的各种可能取值的代号。

能不能就以数字来代表呢？如用 0 代表白色， 1 代表红色， 2 代表蓝色，……等。当然可以，但是这样编出的程序可读性往往较差，因为从这些数字不能直接看出它的实际意义。

那么，能不能以我们后面将讲到的字符串来代表呢？如采用这些信息的英文名字或汉语拼音（有些系统中还允许直接使用汉字）等组成字符串。这样当然也可以，但是以后我们会知道，字符串只是输入输出较方便，而在程序内部的操作如传送，比较等却效率很低。这是因为字符串须存储其中全部字符的编码，数据量较大。若应用程序要解决的实际问题本来只用一些简单的代号就可以解决的，这样做就不合算了。再者，字符串间的次序是由字符的编码定死了的，而我们却常希望根据这些信息的实际意义来灵活规定其间的次序关系。

所以，最好能有一种更直接的办法。PASCAL 中的枚举类型就是为解决这一类问题而引入的。我们在定义枚举类型的同时规定一组记号代表这种离散信息的各个值。在源程序中这些记号可以采用容易看懂其意义的形式，以增加程序的可读性。但这些记号不是字符串，而是标识符。在生成的目标程序中它们都换成了一些简单的代码。具体的代码由编译程序决定，我们不必了解，但它们之间的次序关系完全依我们的定义来决定。

6.2.2 枚举类型的定义和使用

枚举类型是一种自定义的顺序类型，它用顺序列举一组标识符的办法来定义。这些标识符是由户自己确定的，分别代表该类型可以取的各个值。

枚举类型的新类型描述格式如下:

(标识符 1，标识符 2，.....标符 N)

在括号中有一组作为常量的标识符，它们必须符合 PASCAL 语言关于标识符的规定，其中的各标识符之间须用逗号分隔 。这个格式可以出现在程序中任何应出现“类型表记符”的地方，例如可以出现在类型定义中的等号后面，或变量说明中的冒号后面。

其语法图如图 6.2。

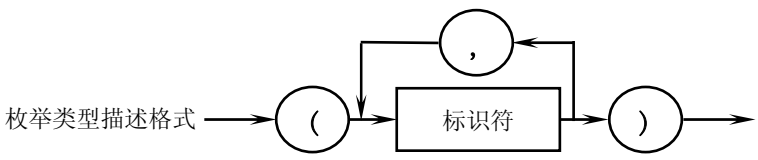


图 6.2　枚举类型描述格式的语法图

有了枚举类型，就可以用来解决这一节开始提到的一些问题。

例如，关于一周中的每一天，可以定义一个 weekday 类型：

```
TYPE
  weekday=(sun,mon,tue,wed,thu,fri,sat);
```

其中 weekday 为新定义的类型名，sun，mon，……等为 weekday 类型可以取的枚举值。

在定义了 weekday 之后，则在已有的数据类型之外，又有了 weekday 这个类型，以后我们就可把变量定义为此类型。如:

```
VAR
  today,workday,restday:weekday;
```

其中，today，workday，restday 是 weekday 类型的变量，可以取 sun 至 sat 中的某一个作为其值。在程序体中可以将枚举值赋给一个枚举变量，也可以对枚举变量的值是否等于某一枚举值进行判断。如：

```
restday:=sun;
IF today=sun THEN ..... ;
```

等等。

需要注意的是：枚举值只能是标识符，而不能是数值常量或字符串常量。例如下面的定义

```
TYPE
  weekday=('sun','mon','tue','wed','thu','fri','sat');
  day=(1,2,3,4,5,6,7);
```

都是错误的。也不要把作为枚举值的标识符视为变量名，如下面一条语句：

```
sun:='sun';
```

也是错误的。

枚举类型的描述格式中，括号内的标识符是作为常量标识符的定义而出现的。换句话说，每出现一个这种格式，就认为是定义了一组符号常量。由于标识符不得重复定义，所以同一个标识符不得重复出现在同一个枚举类型的描述格式中，也不得出现在同一程序中不同的枚举类型的描述格式中。如：

```
TYPE
  k1=(a1,a2,a3,a4,a1);
  k2=(b1,b2,a2);
  color1=(blue,red,black);
  color2=(white,red,green);
```

其中 k1 定义中，a1 重复出现 ，k2 定义中 a2 和 k1 中 a2 重复，color1 和 color2 中 red 重复。这几个定义都是错误的。

上一节中我们已经知道，引入新类型时也可以不定义类型名，而直接将新类型描述格式写到变量说明的冒号后面。如

```
VAR
  today,workday,restday:(sun,mon,tue,wed,thu,fri,sat);
```

但要注意，假如类型名 weekday 已经像上面那样定义过了， 则这里的变量说明部分中就不能再这样写了。因为相同的枚举类型描述格式在一个分程序中出现两次，其中的符号常量都算是重复定义，这是不允许的。

第二章中已说过：程序员自己取标识符时不得和 PASCAL 中的 35 个字符号相同，也最好不要和预定义标识符相同。选取作为枚举值名字的标识符时，当然也应遵循同样的原则。

应注意，这些作为枚举值的符号常量都是用户自己命名的。对机器来说，它只是一个代号，机器只能对它作有限的几种操作（如判断是否相等），而不可能自动识别其含义。我们编程时之所以要选用有意义的单词，只是为了便于人来阅读。

关于枚举类型，有几点需要说明：

其一，枚举类型是顺序类型，它的顺序就是新枚举类型描述格式中各个标识符的顺序，系统认为它的序号从 0 开始依次递增，因此，可以对枚举类型的数据使用 ord，succ，pred 函数。但其中第一个枚举值无前启，最后一个无后继。如 weekday 类型中：

```
ord(tue)=2 ,      ord(sat)=6,      ord(sun)=0,
succ(tue)=wed,    pred(sat)=fri
```

其二，除了上述函数（以及以后将讲到的自定义的过程和函数）外，对枚举类型只能进行赋值操作和关系运算，不能进行算术运算和逻辑运算。

对枚举元素进行比较时，按照顺序类型的一般原则，序号大的值“大于”序号小的值，如

```
mon<tue 为 true,   wed>thu 为 false
```

注意，只能对同一枚举型数据进行比较，不同枚举类型或枚举类型同其它类型相比较是错误的。如：

```
workday>6
sun<red
```

都是错误的。

其三，不允许直接从标准输入设备输入和向标准输出设备输出枚举类型数据，如:

```
read(today);
write(today);
```

都是错误的。

大家不要以为 write(sat)会输出 sat 三个字母，sat 只是一个代号，它的值不是一个字符串，不能用 write 语句来输出。不要把 write(sat)和 write('sat')混淆。

如果想从键盘上把 tue 送给枚举变量 today 怎么办呢？由于枚举类型不能直接从键盘上输入数据，一般采用转换的方法。即在键盘上输入一个机器可以接受的数据（下面的例子中用整数，以后我们会学到也可以用字符串），由程序将它转换为枚举值。例如

```
read(code);
CASE code OF
  0:today:=sun;
  1:today:=mon;
  2:today:=tue;
  3:today:=wed;
  4:today:=thu;
  5:today:=fri;
  6:today:=sat
END
```

其中先用 read 语句读一个数值，其值在 0～6 之间，通过 CASE 语句就可以使 today 得到相应枚举元素。如从键盘上输入 2，则 today:=tue。

如果想输出 today 的值，仍需用程序转换，如可以转换为字符串：

```
CASE  today OF
```

```
    sun : write('sun');
    mon : write('mon');
    wed : write('tue');
    thu : write('thu');
    fri : write('fri');
    sat : write('sat')
  END
```

因为枚举类型属于顺序类型，所以可以在 FOR 语句中用枚举类型变量作循环变量， 如可有以下语句:

```
FOR workday:=mon TO fri DO ......
```

这里用 workday 作循环变量，用 mon 和 fri 作循环的初值和终值，此循环共执行了五次。

6.2.3 枚举类型应用举例

[例 1] 已知六月一日是星期四，编程求出六月份任一天是星期几， 并显示这一天的前一天是星期几。采用如上文所述的枚举类型 weekday 来处理。

题目中有整数向枚举类型的转换，以及枚举类型的输出，用 CASE 语句去实现。

要把六月份某一天 date 与星期几对应起来，若在 CASE 语句中把每一天都罗列出来程序就太长了。但我们知道星期的周期是 7 天，只要将日期对 7 取模，就可以将情况数减少为 7 个。

(date+3)MOD 7 =0 时，today 为 sun;
(date+3)MOD 7 =1 时，today 为 mon;
.
.
.
等等。

用 pred 求出 yesterday 时，须注意 sun 无前启。

程序清单:

```
PROGRAM exam61(input,output);
  TYPE
    weekday=(sun,mon,tue,wed,thu,fri,sat);
  VAR
    date:integer;
    today,yesterday:weekday;
BEGIN
  write('Please input date:');
  readln(date);
  write('Today is ');
  CASE (date+3) MOD 7 OF
    0:begin today:=sun;writeln('sun')  end;
```

```
    1:begin today:=mon;writeln('mon')  end;
    2:begin today:=tue;writeln('tue')  end;
    3:begin today:=wed;writeln('wed')  end;
    4:begin today:=thu;writeln('thu')  end;
    5:begin today:=fri;writeln('fri')  end;
    6:begin today:=sat;writeln('sat')  end
  END;
  IF today=sun
    THEN yesterday:=sat
    ELSE yesterday:=pred(today);
  write('Yesterday was ');
  CASE yesterday  OF
    sun:writeln('sun');
    mon:writeln('mon');
    tue:writeln('tue');
    wed:writeln('wed');
    thu:writeln('thu');
    fri:writeln('fri');
    sat:writeln('sat')
  END
END.
```

运行结果如下：

```
Please input date:20
Today is:wed
Yesterday was:tue
```

细心的读者会发现，这个程序编得相当笨拙，若不用枚举类型可以编得简短得多。实际上，枚举类型输入输出很不方便，故对于那些以输入输出为主要操作的简单应用程序，采用枚举类型没有实际意义。这个例题仅仅是让你了解一下怎样处理枚举类型的输入输出，而不是要你照搬此程序去解决实际问题。在实用中一般是不这样编写的。

但是，对于那些不以输入输出为主要操作的较复杂的应用程序，采用枚举类型却常常可以提高程序的可读性。对于复杂程序，可读性是很重要的。只是限于篇幅和学时，例题不可能举复杂程序，选择典型题目较难。所以请学生自己在实践中逐步体会其意义。不要仅根据这一个例题而误以为枚举类型是多此一举的故弄玄虚。

在许多其它程序设计语言并中没有这种类型，这也是 PASCAL 的一个特点。

下面的例题中，我们用一个变量代表程序工作中当时所处的状态。假如只有两种状态，可以采用布尔变量。但状态数有三个以上的话用布尔变量就不方便了，故我们采用枚举变量。

[例 2] 从输入的 PASCAL 源程序文件中去掉注释，再输出为另一文件。

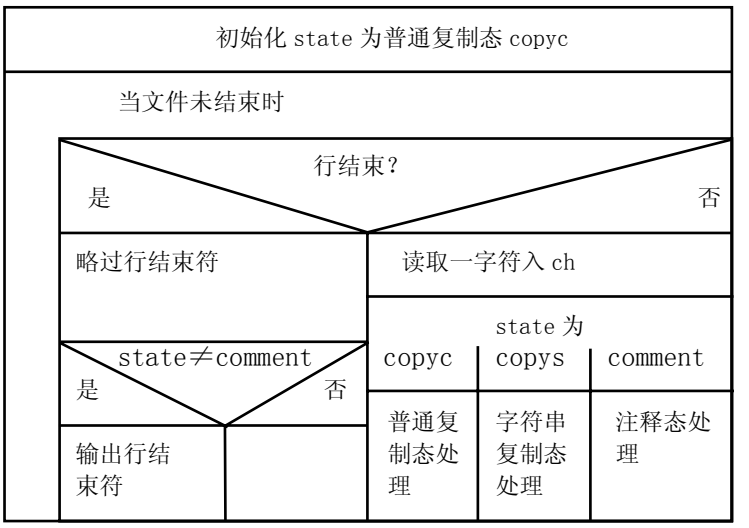
分析：文件操作虽然还未讲到，但本题目可以采用另外的办法解决。操作系统中对应用程序的标准输出和标准输入有重定向的功能，所以我们在编写程序时可以将输入的文件看作标准输入（就好像是从键盘打入的一样），将输出的文件看作标准输出（就好像是送到显示器上去显示一样），将来在运行程序时再用重定向的办法将标准输入指定为所要求的 PASCAL 源程序文件，将标准输出指定为要生成的文件就行了。这样就可以用已学过的 read 和 write 来处理输入和输出了。

只是，为了控制输入文件的结束，以及处理行结束符，还需介绍两个标准函数 eof(f)和 eoln(f)。它们的函数值都是布尔型，参数 f 是文件名，但在处理标准输入时，即 f 为 input 时，这两个函数都可以省略参数，只写 eof 和 eoln 就行了。

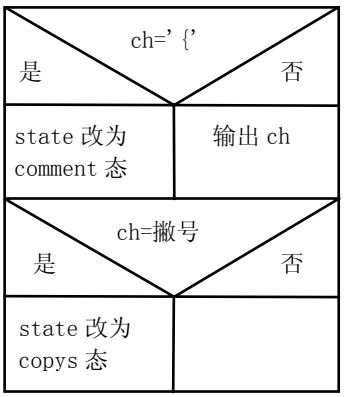
eof 的意思是文件结束（End of file)。当文件内容读完，下面就是文件结束标志时，eof 函数的值为 true；在其它时候，eof 的值为 false。

eoln 的意思是行结束（End of line)。当一行内容读完，下面就是行结束符时，eoln 函数的值为 true；在其它时候，eoln 的值为 false。 行结束符的实际编码在不同系统中可能不同；若将行结束符也当字符数据来读取， 读得的数据在不同系统中也可能不同。 所以，在复制文件时还不能简单地读一字符输出一字符。遇到行结束符时，须用 readln 将它略过，并用 writeln 输出一个行结束符。

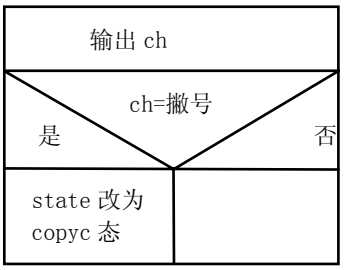
(1) 总算法



(2) 普通复制态处理



(3) 字符串复制态处理



(4) 注释态处理

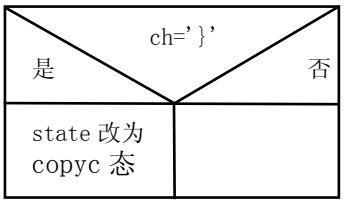


图 6.3 [例 2]算法的结构框图

本题目是在复制源程序时滤掉注释部分。可以规定两个状态：复制态和注释态，在注释态中时不复制。

一般情况下，复制态中读到“{”号时应进入注释态，但有一例外：字符串常量中的

“{”号不起此作用。所以我们又将复制态再分为两个状态：普通复制态及字符串复制态。于是状态就成了三个。

下面的算法中，定义了一个状态变量 state，其值域为三个状态：普通复制态 copyc，字符串复制态 copys，注释态 comment。

其算法的结构框图见图 6.3。

程序如下：

```
PROGRAM rmcomments(input,output);
  VAR
    ch:    char;
    state: (copyc,copys,comment);
BEGIN
  state:=copyc;  {初始化为普通复制态}
  WHILE NOT eof DO
    IF eoln
      THEN {处理行结束符}
        BEGIN
          readln; {略过行结束符}
          IF state<>comment THEN writeln
        END
      ELSE {处理普通字符}
        BEGIN
          read(ch);
          CASE state OF
            {普通复制态}
            copyc:  BEGIN
                      IF ch='{'  THEN state:=comment
                                 ELSE write(ch) ;
                      IF ch='''' THEN state:=copys
                    END;
            {字符串复制态}
            copys:  BEGIN
                      write(ch) ;
                      IF ch='''' THEN state:=copyc
                    END;
            {注释态}
            comment:IF ch='}' THEN state:=copyc
          END
        END
```

END.

本程序的具体用法，因为牵涉到输入输出重定向，故与所用的操作系统有关。这里假设是在 MSDOS 操作系统下，介绍一下它的用法。

首先需要将上述程序编译出来。假设我们给上述程序文件命名为 RMCOMM.PAS，在 Turbo PASCAL 中进行编译，编译时将编译选单中的 Destination 项设成 Disk，即可以在磁盘上生成一个可执行的文件 RMCOMM.EXE。这就是上述程序的目标程序，可以用它来把任一个带有注释的 PASCAL 源程序文件中的注释删掉。

假设盘上有一个带有注释的 PASCAL 源程序文件 TEST.PAS，要将它的注释删掉后生成文件 TEST0.PAS。在操作系统的等待命令状态下，只要当前目录下有刚才所说的文件 RMCOMM.EXE，就可以打以下命令来实现这个操作：

RMCOMM <TEST.PAS >TEST0.PAS

这里，<号是输入重定向的符号，>号是输出重定向的符号。

这个例子中采用了“状态”这个概念。在解决许多较复杂的问题时这是个有用的思路，所以我们再进一步作些说明。这种思路就是分析要解决的问题，看能不能确定几个状态，每个状态满足一定条件时就向别的状态转换，状态转换的同时执行确定的操作。只要能确定出这样的一个模型，就可以据以编出程序。例如上面例子的状态转换模型可以用图 6.4 示意。

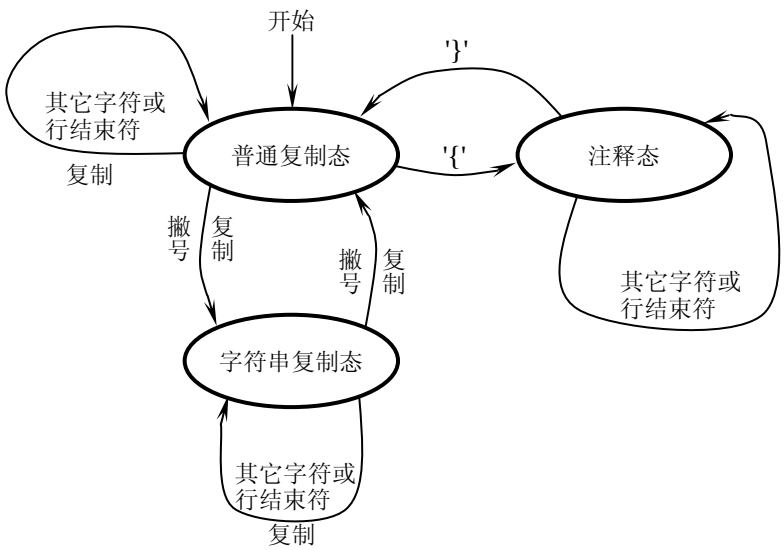


图 6.4　例[2]的状态转换模型

注意这个图不是流程图，而是状态转换示意图。其中带箭头的连线上方或左方标的是状态转换条件，下方或右方标的是状态转换时同时执行的操作。为避免过繁，有些规则图上没有画全，如：“各状态中遇文件结束就立即终止程序”这一转换就没有画出。

6.3　子域类型

6.3.1 子域类型的概念和意义

子域类型又称作子界类型。子域类型是一个已定义的顺序类型的部分连续值的集合，此顺序类型称作该子域类型的宿主类型，可以是 PASCAL 中的预定义类型，也可以是用户

自定义类型，但必须是顺序类型。子域类型通过对上界和下界的规定确定了该类型的取值只是其宿主类型中连续的一部分，即宿主类型的一个子集。

例如，要想用一个变量表示月份，可以定义一个取值为 1～12 的子域类型， 将该变量说明为这个类型的变量。这个子域类型的宿主类型就是整数类型。

引入子域类型主要有以下三方面的用处：

第一方面是可以提供一些自动查错的手段。例如月份若超出 1～12 的范围以外， 比如达到 100，肯定是错误的，但是若该变量是 integer 型，则计算机不认为错误。若该变量是上述的子域类型，超出 1～12 的范围以外就算溢出了，就有可能使计算机自动报告这种错误。

不过这第一方面的好处只是提供了一个可能性，即有可能设计出这样的能自动查错的编译系统，但常见的真实 PASCAL 编译系统大都并没有这样设计。这是因为，表达式的值在编译时是不知道的，只有运行后才可能知道它是否溢出。所以这种查错无法在编译时进行，只能将其功能编入到目标程序中，在运行期间再来查错。这样会大大降低程序的运行效率，在许多情况下这是不合算的。所以这第一方面的用处通常没起作用。

第二个方面是可以节省内存空间。例如某些系统中整型变量占两个字节，但若该变量属于一个取值范围很小的整数的子域类型，就可以只占一个字节。在某些需要保存大量数据的程序中，这种节省有时是很可观的。

第三个方面的用处，也是 PASCAL 中子域类型最主要的用处，是用来构造下一章将介绍的数组。数组是一种由多个成分组成的构造型数据，其成分是由下标来指示的。其下标有多大的取值范围，该数组就有多少个成分。如果下标的取值范围太大，则该数组的成分就太多，会占据大量的存储空间，甚至在机器中容纳不下。所以我们通常要根据需要来限定下标的取值范围，常用的办法之一就是将下标类型规定成某个子域类型。

6.3.2 子域类型的定义

子域类型的新类型描述格式是：

常量 1..常量 2

其中，常量 1 和常量 2 必须是两个属于同一顺序类型的数据，分别称之为子域类型的下界和上界，其中下界的值必须小于上界。注意上下界只能是常量，可以是字面常量，也可以是符号常量，但不能是变量或其它形式的表达式。

其语法图见图 6.5。

如

```
1..120
'a'..'z'
```

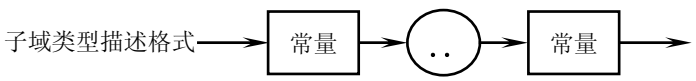


图 6.5　子域类型描述格式的语法图

这种格式可以出现在任何应出现类型表记符的地方，如可以出现在 TYPE 中的=号后面。 例如下面定义是正确的：

```
TYPE
  age=1..120;
  sletter='a'..'z';
```

其中，age，sletter 为子域类型变量名，age 中下界为 1，上界为 120， 也就是说以后凡

是属于 age 类型的变量，其值只能取 1～120 之间 ，它的宿主类型为整型。sletter 的下界为'a'，上界为'z'，其宿主类型为字符型，它的取值只能是所有小写字母。

下面的定义是错误的：

```
TYPE
  age=1.0..120.0;
  month=12..1;
  r=mon..fri;
```

原因是 age 中上、下界均为实型，而实型不属于顺序类型， 不能作为子域类型的宿主类型。在 month 中，错误之处在于下界值大于上界值。在 r 定义中，在于 mon 和 sat 是未定义的，不能作为上下界使用，除非在此之前已出现过

```
(sun,mon,tue,wed,thu,fri,sat)
```

或者以其它方式将 mon 和 sat 定义为常量标识符，才可在其后定义

```
r=mon..sat;
```

由于枚举类型属于顺序类型，故其值可作为子域类型的上下界使用，但要先定义后引用。

有了类型定义后，就可以定义此种类型的变量了，如:

```
VAR
  workday:r;
  manage:age;
```

与前面相同，也可将子域类型的描述格式直接写在变量说明的冒号后面，如:

```
VAR
  workday:mon..sat;
  manage:1..120;
```

6.3.3 子域类型的运算

子域类型的运算继承了其宿主类型的运算，适用于子域类型宿主类型的任何运算符、预定义函数和过程同样也适用于子域类型。例如，宿主类型为整型的子域类型可以进行算术运算、关系运算和赋值运算，而宿主类型为字符型的子域类型只能进行关系运算和赋值运算。

同样，相同宿主类型的不同子域类型可以进行混合运算，但必须指出，在对子域类型变量赋值时，应防止计算结果超过给定范围。如：

```
VAR
  a:1..100;
  b:50..6000;
  c:integer;
```

那么，执行

```
b:=80;
c:=90;
a:=b+c;
```

时，其中第三句就形成溢出错误。

子域类型的数据，当其宿主类型能用 read 和 write 直接输入输出时，它就可以用 read 和 write 直接输入输出。

[例 1] 学校只招收 16～25 岁的学生入校，从键盘上输入本班 100 个学生的年龄（整数），求出小于 20 岁的人数及所有学生的平均年龄。

程序清单如下：

```
PROGRAM  exam63(input,output);
  VAR
    ag:16..25;
    sum,i,n:integer;
BEGIN n:=0;sum:=0;
  FOR I:=1 TO 100 DO
    BEGIN
      read(ag);
      IF ag<20 THEN n:=n+1;
      sum:=sum+ag
    END;
  writeln('There are ',n,' students yonger than 20 years.');
  write('The average age is:',sum/100)
END.
```

6.4　类型间的相容关系

本节用到的某些概念，在后面的章节才给以介绍，这里可以先作粗略了解，待以后学到有关概念后再回过来进一步明确。

这一节我们讨论类型之间的关系。更严格些说，这里讨论的是值与值之间，变量与变量之间，以及值与变量之间，依据它们所属的类型而具有的关系。

这些关系有如下三种：

6.4.1 类型同一

PASCAL 中有些地方要求两个变量属于同一类型，如第八章将讲到的，子程序的变量参数的形参和实参必须属于同一类型。两个变量是否属于同一类型，大多数情况下是清楚的。如下面所述的几种情况。

被同一个类型标识符所说明的变量的类型同一。如

```
VAR
  x1:integer;
  x2:integer;
```

则 x1 和 x2 两个变量属于同一类型。

虽没用类型标识符（如直接使用新类型描述格式），但这些变量是在同一条变量说明

中说明的，则它们也是属于同一类型。如

```
VAR
  x3,x4:(red,blue,yelow,white,black);
```

则 x3 和 x4 两个变量属于同一类型。

还有，某些变量虽被两个不同的类型标识符说明，但这两个类型标识符在 TYPE 定义中已被定义成同一类型，则也认为这些变量是同一类型的。如

```
TYPE
  t1=(s1,s2,s3,s4,s5,s6);
  status=t1;
VAR
  x5:t1;
  x6:status;
```

则 x5 和 x6 两个变量属于同一类型。

前面的例子都是整体变量。以后还会学到其它形式的变量，那些情况虽然比整体变量复杂，但道理是一样的。例如

```
VAR
  x:real;
  a:ARRAY[1..100] OF real;
```

则 x 和 a[5]两个变量属于同一类型。下一章我们会知道，这里 a[5]是数组 a 的一个成分，而数组 a 所属的数组类型的成分类型是 real 型，所以它和 x 的类型同一。

上面这些情况都容易理解。但有一种情况却比较特殊，需要特别说明。PASCAL 的标准中规定：**程序中新类型描述格式的每次出现，都表记一个与其它新类型描述格式所表记的类型不同的类型**。我们之所以将这种描述格式称作“新”类型描述格式，就是因为有这样的规定。程序中第二次出现的这种格式即使和第一次出现的一字不差，也不算是同一类型。如

```
TYPE
  t2=1..200;
VAR
  x7:t2;
  x8:1..200;
```

则 x7 和 x8 两个变量不算同一类型的变量。因为 x7 属于第一次出现的 1..200 类型，而 x8 属于第二次出现的 1..200 类型。PASCAL 中之所以要如此规定，是为了减轻编译程序语法检查的难度。因为新类型描述格式的书写自由度很大，要确定两个格式语法上是否等效有时是很困难的。作了这样的规定后，程序中要想说明两个同一类型的变量，如果不能写在同一条变量说明里的话，就只能用类型标识符来说明，不能直接用新类型描述格式。如上例中，假如将 x8 也改用类型标识符 t2 来说明，就和 x7 的类型同一了。

6.4.2 类型相容

PASCAL 中的许多运算要求类型相容。如关系运算=、<>、<、>、<=、>= 要求两个运算

数或是类型相容，或是一个为整数而另一个为实数。类型相容比类型同一条件更宽些。上面例子中的变量 x7 和 x8 的类型虽不算同一，但算相容（按下面的规则，它们都是 integer 类型的子域），所以关系式 x7<x8 是合法的。

下列四个条件之一成立，则称两个类型相容：

(1)如果两个类型同一，则相容。

(2)如果一个类型是另一个类型的子域，或两个类型都是第三个类型的子域， 则该两个类型相容。如：

```
TYPE
  t1=1..50;
  t2=1..20;
  t3=(sun，mon，tue，wed，thu，fri，sat);
  t4=thu..sat;
  t5=sun..fri;
```

则 t1,t2,integer 三个类型中任两个都是相容的，t3,t4,t5 三个类型中任两个都是相容的。

(3)如果两个集合类型的基类型相容，并且都是紧缩的集合，或都不是紧缩的集合， 则这两个集合类型相容。如：

```
TYPE
   k1=set of char;
   k2=set of 'A'..'Z';
```

则 k1，k2 类型相容。关于“紧缩”的概念见第七章 7.4 节。

(4)如果两个字符串类型所含成分(字符)个数相等，则这两个字符串类型相容。

有时一个值可能同时属于不同的类型。例如整数 10 既属于(2)例中的 t1 类型，又属于 t2 类型，又属于 integer 类型。只有在这些类型相容时才有这种可能。

6.4.3 赋值相容

赋值相容是 PASCAL 中的又一个概念。

如有以下赋值语句:

```
a:=e
```

其中 a 为变量，e 为表达式，要求“e 的值对变量 a 的类型赋值相容”时赋值语句方合法。

一般地，设 a 为变量，e 为表达式，a 为 t1 类型的变量，e 的值属于 t2 类型，以下五个条件之一成立，则称“e 的值对类型 t1 赋值相容”或“e 的值对变量 a 赋值相容”：

(1)t1，t2 类型同一，且不是文件类型（更严格些说， 也不能是以文件为成分的其它结构类型，或以文件为成分的成分……的结构类型，等)。

(2)t1 为实数类型，t2 为整数类型。

(3)t1 和 t2 是类型相容的顺序类型，且表达式 e 的值不超出 t1 的值域。如变量 c 为 1..20 类型，当 b 的值为整数 10 时，赋值语句

```
c:=b
```

有效，此时为赋值相容。若 b 的值为 30，就不赋值相容了。

(4)t1 和 t2 是相容的集合类型，且集合 e 中的每一个成员都属于 t1 的基类型所规定

的值的范围。

(5)t1 和 t2 是相容的串类型。

一般说来，除上述第(2)条外，只有类型相容才可能赋值相容。

除赋值语句外，PASCAL 中还有许多地方要求赋值相容。如 read 语句执行时读入的数据必须对 read 参数中的变量赋值相容。又如 FOR 语句中， 初值和终值必须对控制变量赋值相容。又如，第八章将讲到，子程序（过程和函数）的调用中实参和形参的传递，如果参数是值形参，则要求实参表达式的值与对应的形式参数赋值相容。不过如果参数是变量形参，则要求的不是赋值相容，而是类型同一，应注意其区别。

类型相容和类型同一可以在编译阶段检查，而赋值相容则只能在运行阶段检查。由于检查赋值相容要花大量的机器时间，因此许多实际的 PASCAL 语言编译系统不提供或不完全提供检查赋值相容的功能，如不检查给子域类型变量赋的值是否越界。

习题

6.1 为什么要使用枚举类型？枚举类型的特点是什么？枚举类型能否直接输入输出，如何解决这个问题？

6.2 试建立一枚举类型来表示一年中的月份，并练习把从键盘上输入的数字用英文单词输出。(例如输入 1，则打印 JANURAY)

6.3 参照下面介绍的算法，编程序统计输入的正文文件中有多少个字符子串 HAO 及多少个字符子串 HUAI。

算法简述：

与 6.2 节[例 2]同样，用重定向的办法将输入的文件当做标准输入。

用变量 nhao 来统计串 HAO 的个数，变量 nhuai 来统计串 HUAI 的个数。程序中设六种状态，见图 6.6 示意。

图 6.6 中每个框表示一个状态，标在带箭头的连线左方或上方的字符是状态转换的条件，标在带箭头的连线右方或下方的是状态转换的同时要进行的操作。图中状态转换线画得不全（未画出各状态转回初始态的线以及转到结束态的线），以下面说明为准：

开始时初始化两个计数变量 nhao 和 nhuai 后进入初始态；

除结束态外的五个状态中，只要文件未结束，就读取字符，若读得的字符是图中引出线所标的条件字符，就按图所示转换状态；

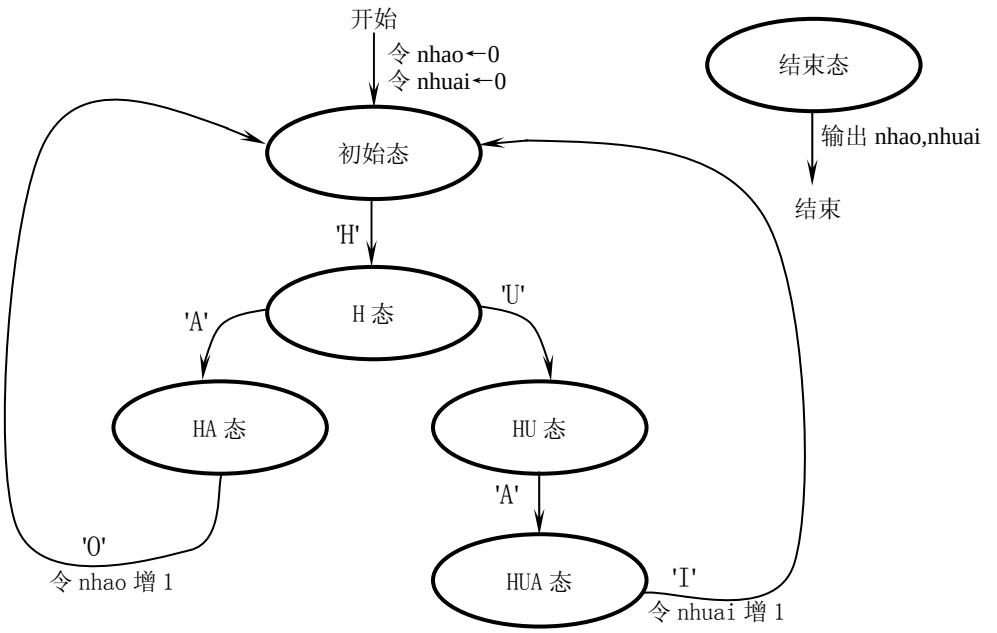


图 6.6　习题 6.3 的状态转换示意

若读得是'H'字符，都立即转换为 H 态；

若读得是其它任何字符，都立即转换为初始态；

若遇文件结束，都立即转换为结束态；

结束态中不停留，无条件地输出变量 nhao 和 nhuai 中的计数后结束程序。

可以自设一些记号代表各状态，用一个枚举类型的变量记载当前状态，编出相当于上述动作的程序。具体设计时可以灵活处理，如也可以不设结束态，只设五个状态也能实现等效于上述动作的程序。

6.4 使用子域类型有何用处？对子域类型的上下界有何规定？

6.5 编写一个程序，读入某一天的日期(1..31)，月份(1..12)，年份(1900..2099)，并以日期/月份/年份的形式输出，其中月份按英文缩写如 JAN，FEB，MAR，……的形式输出。

6.6 什么是类型相容？什么是赋值相容？

第七章　数组

7.1　数组概念的引入

我们已经讨论了 PASCAL 中的所有简单数据类型，它们的特点是其值不可再分。从本章起，我们要讨论构造类型。构造类型的每个值可以分成若干个成分。数组就是一种构造类型。

在前面的例子中，我们已经接触了处理大量数据的题目，例如第 5.3 节的[例 1]就是求输入的 200 个数的平均值。但这些例题中都有一个共同点， 就是用同一个变量来先后表示不同的数据，新的值放进变量中时旧的值就被冲掉了。

然而实际应用中常常会遇到这样的问题：大量的数据不能丢掉，因为后面还要用到，必须同时保存，这就不能采用上面说的办法了。这时，若采用本书以前讲过的手段，只能为 200 个数设置 200 个不同的变量。这样一来，不仅变量说明部分太长，而且因为每个变量的名字都不同，程序无法采用循环结构，整个程序将长得不可容忍。

其实，在数学中我们已经有了“下标变量”的习惯表示方法。如我们写一个 x_k，当 k=1 时 x_k 代表 x_1，当 k=2 时 x_k 代表 x_2，等。同一个记号 x_k 可以代表不同的变量。 若将数学中的这个表示方式引入计算机语言中，就可以用循环的算法处理上述问题，这个困难就解决了。不过在第二章的 2.3.4 中我们已经知道，在 PASCAL 程序中不可以直接写 xk，x1，x2 等来表达这种思想。PASCAL 中引入数组的概念可以实现与之相当的表达形式。

例如，采用下面将介绍的数组概念，我们可以定义一个名叫 x 的数组类型的变量。x 的值可以分成 200 个成分，每个成分都是实数，可以分别用 x[1]、x[2]、x[3]、……、x[200]来表示。

这样，标识符 x(整体变量)代表这个由 200 个实数组成的数组的整体。必要的时候可以赋给它一定的数学物理意义，例如可以看成一个 200 维空间中的向量，等。

其中的一个成分如 x[50]也是一个变量，但它是整体变量 x 中的一个成分，所以可以称作成分变量。其中方括号内是下标，它指出该成分变量是 x 中的哪一个成分。 数组的成分变量又叫做下标变量。方括号中的下标可以是一个含有变量的表达式，所以，当 k=50 时，x[k]就可以代表 x[50]。这就解决了上面说到的问题。

除了上面所说的一类问题以外，下面我们会看到，还有许多其它问题可以用数组来解决。

7.2　数组的定义及使用

7.2.1 数组的定义

数组类型也是自定义类型，数组类型的基本的新类型描述格式如下:

ARRAY［下标类型］ OF 成分类型

其中“下标类型”和“成分类型”都是类型表记符，也就是说，都可以写已有的类型标识符，也可以写新类型描述格式。下标类型必须是顺序类型，成分类型则没有限制。

这里的成分类型是规定这种数组里每个成分的类型的，而下标是用来指示成分的，不同的下标值指示不同的成分。所以下标类型有多少个可能的取值，这种数组里就要有多少个成分。

例如，若某种数组的下标类型是 boolean 型， 则这种数组就有两个成分， 一个由下标 false 来指示，另一个由下标 true 来指示。下标类型的取值范围越大，数组也就越大，每一个数组变量需要占据的内存空间也越大。

因为计算机内存的容量有限，所以一般不采用像 integer 这样的类型来作下标类型。因为 integer 的取值范围为-32768 至 32767，这样的范围太大了。一般是根据问题的需要来规定下标类型的取值范围，最常见的是用子域作为下标类型。

例如，上一节的例子中，可以用 1..200 作为下标类型，该数组类型可以描述为

```
ARRAY [1..200] OF real
```

按前面讲过的原则，这种格式可出现在类型定义中的等号后面，或变量说明中的冒号后面，或 PASCAL 中任何可以出现类型表记符的地方。

如：

```
TYPE
  weekday=(sun,mon,tue,wed,thu,fri,sat);
  kk=ARRAY [weekday] OF integer;
  ss=ARRAY [1..40] OF real;
VAR
  s,score:ss;
  tp:kk;
  x: ARRAY [1..200] OF real;
```

这里定义了名为 kk 和 ss 的两个数组类型标识符。

在 kk 定义中，以枚举类型 weekday 作为下标类型。它规定 kk 类型的每个数组可以有 7 个成分，分别用下标 sun、mon、tue、wed、thu、fri 和 sat 来指示。OF 后的 integer 说明了该类型数组的成分是整数。

在 ss 类型定义中，下标类型是一个以整型为宿主类型的子域类型，该类型的数组有 40 个成分，每个成分都是实数。其下标的下界为 1，上界为 40。

s 及 score 是 ss 类型的数组变量，tp 是 kk 类型的数组变量。

x 又是另一种类型的数组变量，但这里没有定义类型标识符， 而是直接用数组类型的描述格式来说明 x 变量。

同一个数组中，每个成分变量的类型都是相同的，这是数组类型的一个特点。而且，数组的下标是顺序类型的值，所以可以用下标递增或递减的方式来遍历数组的各个成分，这又是数组类型的另一个特点。

上面已说过，数组的成分类型没有限制，所以它既可以是简单类型，也可以是构造类型，还可以是后面将讲到的指针类型。习惯上将成分类型不是构造类型的数组称作一维数

组。而将成分类型为一维数组的数组称作二维数组，将成分类型为二维数组的数组称作三维数组，……等等。不过这只是习惯，不是严格的术语。我们先介绍数组的一般规定及应用，主要针对一维数组，而与多维数组有关的特殊问题在下一节专门介绍。

上面介绍的描述格式是基本的格式，对于多维数组还可以用简略格式，也在后面介绍。

每定义一个数组变量，PASCAL 编译程序就为它在内存中开辟一段存储空间。如 s 数组要占据可存放 40 个实数的内存空间。而 tp 数组则占据可存放 7 个整数的空间。

前面已学过：子域类型的下界和上界必须是常量，而不能是变量。所以需要说明的是：下标的起始值和终值一定要在类型定义中确定，不能在程序执行中通过计算机的操作来改变，也就是说，PASCAL 语句只允许静态分配数组，数组成分个数一经说明，在整个程序执行期间是固定不变的。如下写法是错误的：

```
TYPE
  tt=ARRAY [1..n] OF real;
VAR
  a:tt;
  i,n:integer;
BEGIN
  read(n);
  FOR i:=1 TO n DO read(a[i]);
     ...
     ...
```

7.2.2 数组变量的整体引用

整体的数组变量在程序中直接用数组变量名来表示。

因为数组变量的值是一构造型的数据，PASCAL 中没有提供一般整体数组间的算术逻辑运算符（只有对字符串的运算见下文），所以一般数组变量出现在表达式中时，按目前学到的情况，这个表达式中只能有这一个变量，而不能有运算符（以后会学到，数组变量还可以作为实参代入函数或过程的调用中）。

要通过键盘往一般数组（除字符串输入见下文）中送入数据时，也不能直接对整体变量操作，而需要分别对每个成分操作。

一般的数组变量虽然不能整体运算，但可以整体赋值。回忆上一章关于赋值相容的叙述，对普通数组只有类型同一时才能赋值相容。如前面定义的数组 s 和 score 都是 ss 类型，则以下赋值语句是合法的

```
s:=score
```

这个赋值语句可以将数组 score 原有的值整体地复制到数组 s 中去。

程序中对数组整体引用不多，更多的是对数组成分分别引用。

7.2.3 数组成分的引用

数组成分通过数组变量及其下标来引用。

数组变量[下标表达式]

其中下标表达式的值必须属于定义中的下标类型，它表示了现在引用的是数组中的哪

个成分。如对于前面定义的数组，可以有：

s[20]，　score[33]，　　tp[mon]，

x[i+j]　　（其中 i，j 都是整数且其和在 1～200 范围内）

数组成分的这种表示形式叫做“下标变量”，属于第二章中所说的“成分变量”，它的类型就是该数组规定的成分类型。例如上述 s[20]就是一个实型的变量， 它可以出现在任何可以使用实型变量的地方，如可被赋值，可以出现在表达式中，可出现在输入语句中，等。又如 tp[mon]是一个整型的变量， 它可以出现在任何可以使用整型变量的地方，唯一的例外是不能用作 FOR 语句中的控制变量，前面我们指出过，FOR 语句中的控制变量只可用整体变量，不可用成分变量。

7.2.4 应用举例

在实际的应用中，常常需要顺序处理（或按一定有规则的次序处理）一批相同类型的数据，而这一批数据又必须同时保存，不能采取由一个变量先后保存不同数据的办法。这时，通常就采用数组的结构，由一个规律变化的变量来指示其下标，用循环结构的程序实现其算法。如下面的例子：

[例 1] 从键盘上输入十个整数，然后按相反顺序打印输出。

程序清单：

```
PROGRAM  exam71(input,output);
  TYPE
    num=ARRAY [1..10] OF integer;
  VAR
    n:num;
    i:integer;
BEGIN
  writeln('Please input 10 number:');
  FOR i:=1 TO 10 DO read(n[i]);
  readln;
  FOR i:=10 DOWNTO 1 DO write(n[i]:5);
  writeln
END.
```

运行结果:

```
Please input 10 number:
10 78 34 90 38 27 8 19 67 55
   55   67   19    8   27   38   90   34   78   10
```

在实际应用中还有一种情况，有时需要对一批变量中的一个进行操作，而操作哪个则要由上文的结果决定。过去的例子中，这种情况要用选择结构来实现。如果选择的变量太多，则程序篇幅会很大。现在我们可以将这一组变量合并成一个数组，只要能找出一个规则从上文的结果中求出要操作的那一个成分变量的下标，就可以不再用选择结构。如下面的例子。

[例 2] 已知某班学生共 30 人，年龄在 16～20 岁之间，试统计该班各个年龄的人数，其中学生年龄由键盘输入。

分析：用本章以前的手段处理这种问题，需要设五个计数用的变量来统计各年龄的人数。如可以用 a16，a17，a18，a19，a20。开始给它们清零时，因无法用循环，只能罗列多条赋值语句。然后每读入一个年龄 x，就要用选择结构来使其中一个变量加一：

```
CASE x OF
  16: a16:=a16+1;
  17: a17:=a17+1;
  18: a18:=a18+1;
  19: a19:=a19+1;
  20: a20:=a20+1
END
```

假如年龄范围再宽，这个选择结构就会更加庞大（初始清零的篇幅也更大）。现在有了数组的概念，将这五个变量作为一个数组 a 的五个成分，只要根据 x 确定它的下标就能指定要操作的变量，这种 CASE 结构就不必要了。

程序清单：

```
PROGRAM exam72(input,output);
  TYPE
    ages=16..20;
  VAR
    a:ARRAY [ages] OF integer;;
    i,j:integer;
    x:ages;
BEGIN
  FOR i:=16 TO 20 DO a[i]:=0;       {初始化清零}
  write('Please input 30 ages:');
  FOR j:=1 TO 30 DO                 {统计}
    BEGIN
      read(x);
      a[x]:=a[x]+1
    END;
  FOR i:=16 TO 20 DO                {输出}
    writeln('There are ',a[i],' students aged ',i,' years')
END.
```

运行结果：

```
Please input 30 ages:16 20 19 17 18 19 18 18 16 20 17
17 16 17 18 19 18 17 17 17 17 18 18 19 17 18 17 17 18 20
There are 3 students aged 16 years
```

```
There are 11 students aged 17 years
There are 9 students aged 18 years
There are 4 students aged 19 years
There are 3 students aged 20 years
```

还有的时候，我们需要将某个值按照一定的对应规则变换为另一值。例如上一章处理枚举量的输入时，需要将输入的另一类型的数值（如整数）转换为枚举值。当时用的是 CASE 语句，那样，需要在 CASE 语句内罗列大量的赋值语句。现在，我们也可以利用数组下标与成分变量之间的对应关系来实现。例如

```
  ......
TYPE
  weekday=(sun,mon,tue,wed,thr,fri,sat);
VAR
  iw,today:weekday;
  idx:  ARRAY [0..6] OF weekday;
  x:  0..6;
BEGIN
  FOR iw:=sun TO sat DO idx[ord(iw)]:=iw;
   ......
```

这样，下文只要在 x 中得到了 0～6 的整数，就可以用

```
today:=idx[x]
```

将其变换为枚举值并送入 today 中。这样，从数学的角度看，数组 idx 相当于一个由序号求枚举值的转换函数。idx[x]不仅可以像上面那样用在赋值号右边，还可以用在许多其它地方，如

```
IF idx[x]<>sat THEN  ......
```

等等。应注意，PASCAL 中对普通数组只提供了数组变量， 而没有数组常量的表示法， 所以 idx 必须像例中那样先进行初始赋值。

上面说到的数组的几种典型应用中，大量遇到的是第一种。这从下文的其它例题中就可以看到。

[例 3] 输入学号为 1～10 的 10 个学生的成绩（百分制）， 找出其中最高分者和最低分者的学号及其成绩，求出所有学生的平均成绩，并输出每个学生的成绩与平均分之差。

分析：求最大值，最小值及平均值（需先求总和）三种算法在第五章已经讨论过。如果三个递推算法并在一起同步地进行，可以纳入一个循环中，采用一个变量先后存放不同数据的办法就可以解决，所以，假如只作这三项可不必采用数组。但是，本题目中除这三项外还要求各人成绩与平均分之差，这些必须在平均值求出来以后才能进行，所以它不能和前述的操作纳入同一个循环中。为此，10 个学生的分数在前一个循环中不能冲掉，必须同时保存以便后一个循环中使用。所以，必须用一个数组来存放成绩。

程序采用两遍循环。第一遍完成数据输入数组、求和、求最大值、求最小值四项任务。然后输出已求出的信息。第二遍循环输出所求的差，输出时采用表格形式，将学号、成绩、

所求的差对照排列。

程序清单：

```
PROGRAM  exam73(input,output);
  CONST
    n=10;
  VAR
    a:ARRAY [1..n] OF integer;
    max,min,i,mi,ni:integer;
    s:real;
BEGIN
  s:=0; max:=-1; min:=101;
  writeln('Please input ',n,' numbers:');
  FOR i:=1 TO n DO
    BEGIN
      read(a[i]);
      s:=s+a[i];
      IF a[i]>max THEN  BEGIN max:=a[i]; mi:=i END;
      IF a[i]<min THEN  BEGIN min:=a[i]; ni:=i END
    END;
  s:=s/n;
  writeln('Max. No. ',mi,', score: ',max);
  writeln('Min. No. ',ni,', score: ',min);
  writeln('mean=',s:6:1);  writeln;
  writeln('No.  score  score-mean');
  FOR i:=1 TO n DO writeln(i:3,a[i]:6,a[i]-s:10:1);
END.
```

运行结果：

```
Please input 10 numbers:
78 80 63 56 73 95 86 71 67 79
Max. No. 6, score: 95
Min. No. 4, score: 56
mean=  74.8

No.  score  score-mean
  1    78       3.2
  2    80       5.2
  3    63     -11.8
  4    56     -18.8
```

```
 5     73       -1.8
 6     95       20.2
 7     86       11.2
 8     71       -3.8
 9     67       -7.8
10     79        4.2
```

注意程序中 max 和 min 的初值没有取第一个成绩，而是分别设成了-1 和 101。 这种做法在第五章已有介绍。从这个例子中可以看到这种做法的方便之处：不必将第一个成绩的处理单独编到循环之外，程序可以简短。

[例 4] 输入十个整数，把这十个整数按由小到大的顺序输出。

排序的算法需要对数据多次比较和交换，所以通常必须把一批数据同时存放在存储器中才能进行。较简单的就是采用数组来存放。因此这个题目可分为三大步：

输入数据到数组中；

对数组中数据排序；

将数组中数据依次输出。

输入输出都比较简单，这里主要讨论排序的算法。排序的方法有很多种，这里介绍最简单的两种：冒泡法和选择法。下面按从小到大排序叙述，若要从大到小排序，则只要将其中比较的大小反过来就行了。

算法一：冒泡法。

冒泡法的基本操作是比较数组中相邻的成分，若不合次序要求（即前一个大于后一个）就将二者交换。若在数组 a 中从 a[1]开始每个成分都和后一个成分作这种操作， 依次进行下去，最后 a[9]和 a[10]进行这种操作，这样一遍下来，最大的一个成分必然就换到了 a[10]的位置。我们将这一遍称作“一趟冒泡排序”，一般说来，从 a[1]到 a[i]进行一趟冒泡排序，可以将 a[1]到 a[i]中最大的一个换到 a[i]的位置。下面列出 a[1]到 a[5]一趟冒泡排序的实例示意：

```
8  4  9  3  6      8 和 4 比较，对调
4  8  9  3  6      8 和 9 比较，不对调
4  8  9  3  6      9 和 3 比较，对调
4  8  3  9  6      9 和 6 比较，对调
4  8  3  6  9      结果，最大一个成分 9 落在第 5 个位置
```

从 a[1]到 a[i]进行一趟冒泡排序的算法可用如图 7.1 的结构框图示意：

整个冒泡法就是多次的“一趟冒泡排序”，如

先从 a[1]到 a[10]一趟冒泡排序，将 a[1]到 a[10]中最大者换到 a[10]的位置；

再从 a[1]到 a[9]一趟冒泡排序，将 a[1]到 a[9]中最大者换到 a[9]的位置；

再从 a[1]到 a[8]一趟冒泡排序，将 a[1]到 a[8]中最大者换到 a[8]的位置；

……

……

再从 a[1]到 a[2]一趟冒泡排序，将 a[1]到 a[2]中最大者换到 a[2]的位置。整个排序就完成了。可以用图 7.2 的结构框图示意。

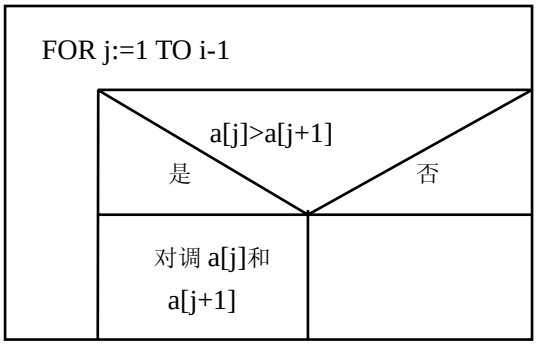


图 7.1　从 a[1]到 a[i]一趟冒泡排序的算法

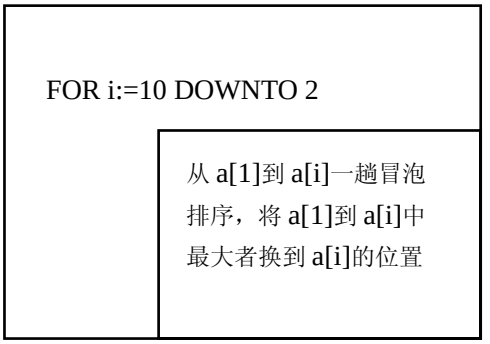


图 7.2　从 a[1]到 a[10]冒泡排序的外层算法

将图 7.1 整个代替图 7.2 中的循环体框就可以得到完整的冒泡法结构框图，图略。这种排序的方法总是最小的数像气泡一样往上冒而大的数逐个向下沉，所以形象地被称为冒泡排序法。程序清单如下：

```
PROGRAM sort(input,output);
  CONST n=10;
  VAR
    a:ARRAY [1..n] OF integer;
    t,i,j:integer;
BEGIN
  writeln('Input ',n,' numbers:');          {输入}
  FOR i:=1 TO n DO read(a[i]);
  readln;
  FOR i:=n DOWNTO 2 DO                      {冒泡法排序}
    FOR j:=1 TO i-1 DO
      IF a[j]>a[j+1] THEN
        BEGIN
          t:=a[j];a[j]:=a[j+1];a[j+1]:=t
        END;
  writeln('Output:');                       {输出}
  FOR i:=1 TO n DO write(a[i]:4);
  writeln
END.
```

运行结果：

```
Input 10 numbers:
  110   24   367   218   43   107   136   32   44   90
```

```
Output:
  24  32  43  44  90 107 110 136 218 367
```

冒泡排序法是一种比较简单的算法，缺点是数据交换次数太多，运行较慢。上面是冒泡法的基本算法，在对 n 个值进行排序时，第一次循环进行 n-1 次比较，第 i 次循环进行 n-i 次比较，故这个程序需比较 n(n-1)/2 次，最坏的情况下每次比较都有可能进行交换，故最多需交换 n(n-1)/2 次。

除了上述基本算法外，冒泡法还有几种改进方案，改进方案的思想都是在非最坏情况下减少比较次数，但不能减少交换次数。本书不作全面介绍，感兴趣者可以研究本章后面的习题。

算法二：选择法。

选择排序法的外层算法可以归结为多趟的“选择”：

第一趟：在 a[1]至 a[10]中选出最小的成分，将其换到 a[1]的位置；

第二趟：在 a[2]至 a[10]中选出最小的成分，将其换到 a[2]的位置；

第三趟：在 a[3]至 a[10]中选出最小的成分，将其换到 a[3]的位置；

……

第九趟：在 a[9]至 a[10]中选出最小的成分，将其换到 a[9]的位置。

可见外层的算法与冒泡法相比没有根本的差别（只是反了过来），其主要的差别在内层。选择法内层的“一趟”是“选择”最小成分。

从多个数中找出最小的那个数算法我们已经学过，但现在这里主要目的不是要找出最小的数值为几，而是要确定出它是哪一个（即确定它的下标），以便下面交换。故我们设一个变量 p 来记其下标。从 a[i]到 a[10]一趟选择的算法可以描述如下：

```
p:=i;
FOR j:=i+1 TO 10 DO
  IF a[j]<a[p] THEN p:=j;
IF p<>i THEN 交换 a[p]与 a[i]
```

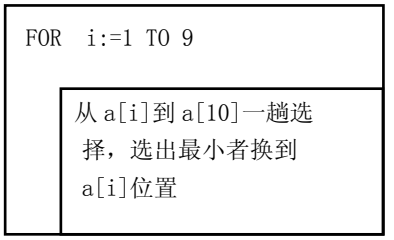
这里的第二句 j 从 i+1 到 10 循环一遍后 p 中就是 a[i]到 a[10]中最小者的下标。

该算法的结构框图见图 7.3 示意，其中总算法就是将内层算法嵌入外层算法中所得。

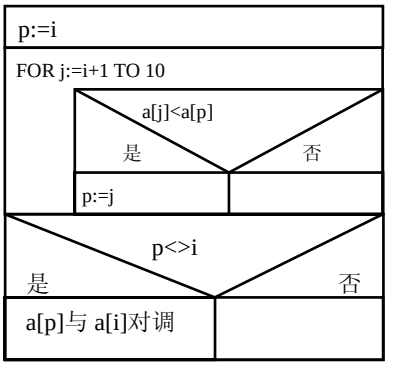
可以看到，这种算法在每“一趟”选择完成后才可能进行一次交换，也就是说 n 个数排序最多交换 n-1 次，比冒泡排序法交换次数少得多，所以程序效率比较高。

程序清单：

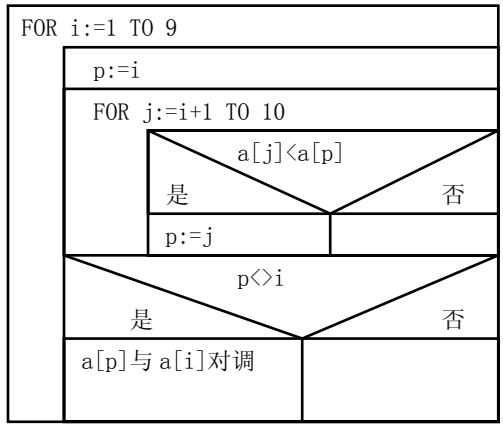
```
PROGRAM sort(input,output);
  CONST n=10;
  VAR
    a:ARRAY [1..n] OF integer;
    t,i,j,p:integer;
BEGIN
```



(1)外层算法



(2)内层：从a[i]到a[10]一趟选择



(3)总算法

图 7.3　选择法排序结构框图

```
    writeln('Input ',n,' numbers:');          {输入}
    FOR i:=1 TO n DO read(a[i]);
    readln;
    FOR i:=1 TO n-1 DO                        {选择法排序}
      BEGIN
        p:=i;
        FOR j:=i+1 TO n DO
          IF a[j]<a[p] THEN p:=j;
        IF p<>i THEN
          BEGIN
            t:=a[i];a[i]:=a[p];a[p]:=t
          END
      END;
    writeln('Output:');                       {输出}
    FOR i:=1 TO n DO write(a[i]:4);
    writeln
  END.
```

我们看到，这个算法效率虽然比冒泡法高，但程序也复杂了些。有人在此基础上简化，省去变量 p，一趟选择中每遇见一个较小的成分立即和 a[i]交换，选择中每次都和 a[i]

比较，一趟选择最后不再交换。这样改后程序简单了（和冒泡法相当），但交换次数多了，效率也降低了（也和冒泡法相当）。具体程序略，请读者自行考虑。

［例 5］十个运动员参加短跑比赛，跑道号 1～10。编程序， 按跑道号次序输入十个成绩（秒），计算机把成绩按名次（即时间由小到大）的顺序输出，输出时每个数前要标出它的跑道号。

这一题基本要求同上一题，可用冒泡法，也可用选择法，下面选用选择法。但是这里要求输出跑道号，则需要将跑道号放在另一个数组中，对成绩排序过程中凡需要交换数组成分的时候，都同时交换跑道号数组中的对应成分，最后将两个数组并列输出即可。

程序清单如下：

```
PROGRAM sort(input,output);
  CONST n=10;
  VAR
    a:ARRAY [1..n] OF real;                {时间排序数组}
    x:ARRAY [1..n] OF integer;             {跑道号数组}
    t:real;
    tx,i,j,p:integer;
BEGIN
  writeln('Input ',n,' numbers:');         {输入}
  FOR i:=1 TO n DO          {输入成绩到 a 数组，同时填写 x 数组}
    BEGIN read(a[i]); x[i]:=i END;
  readln;
  FOR i:=1 TO n-1 DO                       {选择法排序}
    BEGIN
      p:=i;
      FOR j:=i+1 TO n DO
        IF a[j]<a[p] THEN p:=j;
      IF p<>i THEN
        BEGIN              {交换 a[i],a[p]同时交换 x[i],x[p]}
          t:=a[i]; a[i]:=a[p]; a[p]:=t;
          tx:=x[i];x[i]:=x[p]; x[p]:=tx
        END
    END;
  writeln('Output:');                      {输出}
  FOR i:=1 TO n DO writeln(x[i]:2,a[i]:8:4)
END.
```

运行结果如下：

```
Input 10 numbers:
12  13.4  14.67  13.18  15.3  11.7  14.6  14.2  15.4  11.68
```

```
Output:
10 11.6800
 6 11.7000
 1 12.0000
 4 13.1800
 2 13.4000
 8 14.2000
 7 14.6000
 3 14.6700
 5 15.3000
 9 15.4000
```

[例 6] 用筛法求 n 以内的所有素数（实现时令 n 取 250）。

前面已作过直接按定义用尝试法确定素数的算法。那种算法思路虽然简单，但效率低。这里介绍的筛法，不仅减少了计算次数，而且免除了大量的除法运算。我们知道，对计算机来说，除法是四则运算中效率最低的一种。所以筛法远比以前所用的算法效率高。

筛法的思路是这样的：我们知道，素数的倍数（除该素数自身外）必然不是素数，而不是素数的数（除 1 以外）必然是某个素数的倍数。我们可以先将 2～n 的所有整数记在一个地方（称作筛），此时筛中最小的整数（即 2）必为素数。 再将该素数的所有倍数（除该数自身外，下同）从筛中划掉，此时筛中剩下的大于该数的数中，最小的一个（此时为 3 ）必为素数。如此继续，每确定一个素数，就将其倍数划掉。当 n/2 以内的素数找完时，即使后面还有素数，筛中已没有它的倍数了，可以不必再划。最后筛中就只剩素数了。

此算法描述如下：

```
1、将 2～n 记入筛中；
2、FOR i:=2 TO n DIV 2 DO
     IF i 未被划掉
       THEN 将 i 的所有倍数(除 i 自身)从筛中划掉;
3、输出筛中剩余的整数。
```

其中“将 i 的所有倍数从筛中划掉”一句又可以细化如下：

```
1、j:=2*i;
2、WHILE j<=n DO
      BEGIN 从筛中划掉 j; j:=j+i END
```

采用上述的算法已经可行了，但是还可以继续改进。考虑总算法的第二步。我们知道，2 以上 n 以内的每个非素数，必然有不少于两个的素因子，故最小的素因子必不大于 $\sqrt{n}$ 。当 $\sqrt{n}$ 以内素数的倍数都划掉后，筛中必没有非素数了。所以，循环的终值可以提前，不取 n DIV 2，而改为 trunc(sqrt(n))。这样程序循环次数还可以减少。

我们采用一个布尔数组 sieve 代表筛，下标取 2～n，sieve[i]为 true 表示筛中有 i，sieve[i]为 false 表示 i 已被划掉。这样，可以编出如下程序：

```
PROGRAM prime(output);
```

```
{用筛法求 250 以内的所有素数}
CONST  n=250;
VAR    sieve: ARRAY [2..n] OF boolean;
       i,j:   integer;
BEGIN
  FOR i:=2 TO n DO sieve[i]:=true;          {将 2～n 记入筛中}
  FOR i:=2 TO trunc(sqrt(n)) DO
    IF sieve[i] THEN                        {若 i 未被划掉}
      BEGIN                                 {划掉其倍数}
        j:=i+i;
        WHILE j<=n DO
          BEGIN sieve[j]:=false; j:=j+i END
      END;
  FOR i:=2 TO n DO IF sieve[i] THEN write(i:4); {输出}
  writeln
END.
```

[例 7] n（n<=50）只猴子选大王，方法如下：所有猴子按 1～n 编号围成一圈，从第一只开始按顺序 1，2，…，m 报数，凡报 m 的猴子退出圈外，如此循环报数，直到圈中只有一只猴子为止，它就是大王。输出这个猴子的编号。n 及 m 由键盘输入。

采用一个布尔数组，其前 n 个成分分别代表 1～n 号猴子。其值为 true 表示它在圈中，值为 false 表示它已退出。用变量 i 指示猴子号，k 表示报数的数字，s 作控制变量退掉圈中所有 n 只猴子，最后一次退出的就是大王。则筛选大王的过程可以描述如下：

```
初始化数组;
令 i=0;
FOR s:=1 TO n DO
  【 FOR k:=1 TO m DO 将 i 指向下一个在圈中的猴子;
    退出 i 号猴子
  】
```

作完上述操作后变量 i 的值（最后一次退出的猴子号）就是大王的号。上述操作中“将 i 指向下一个在圈中的猴子”算法可以细化如下：

```
REPEAT
  i 增 1;
  若 i>n 则 i:=1
UNTIL i 号猴子在圈中
```

整个算法的程序清单如下：

```
PROGRAM  exam78(input,output);
  CONST num=50;
  TYPE
```

```
    que=ARRAY[1..num] OF boolean;
  VAR
    a:que;
    n,m,i,s,k:integer;
BEGIN
  write('Input n & m: ');                {n 及 m 由键盘输入}
  readln(n,m);
  FOR i:=1 TO n DO a[i]:=true;           {初始化数组}
  i:=0;
  FOR s:=1 TO n DO                       {退掉圈中所有 n 只猴子}
    BEGIN
      FOR k:=1 TO m DO                   {作 m 步：}
        REPEAT                           {将 i 指向下一个在圈中的猴子}
          i:=i+1; IF i>n THEN i:=1
        UNTIL a[i];
      a[i]:=false;                       {退出 i 号猴子}
    END;
  writeln('The king is:',i:4,'th.')      {输出圈中最后退出的一个}
END.
```

运行结果如下：

```
Input n & m: 33 11
The king is:  30th.
```

[例 8] 求已知的 n 次多项式（实现时令 n=8）在给定点 x_0 的泰勒展开式。

具体说这个题目就是已知多项式

$$a_0x^n+a_1x^{n-1}+...+a_{n-1}x+a_n$$

的系数 a_0，a_1，……，a_n 以及给定点 x_0 的值，要化为

$$w_0(x-x_0)^n+w_1(x-x_0)^{n-1}+...+w_{n-1}(x-x_0)+w_n$$

的形式。问题就是要求出系数 w_0，w_1，……，w_n 值。

将该多项式记作 f(x)，则根据泰勒定理，可以知道

w_n 就是 $f(x_0)$；

w_{n-1} 就是 $f'(x_0)$；

w_{n-2} 就是 $f''(x_0)/2$；

……

等等。因此本题中的方法也可以用来求多项式在给定点的值及导数等。

数学上可以知道，这种结果是唯一的，也就是说，不管我们用什么方法，只要能化成这种形式，结果肯定一样。所以我们可以选取一种较方便的算法。用计算机解决这个问题可以用秦九韶法。秦九韶法的思路就是反复进行多项式除法。将这个 n 次多项式除以 $(x-x_0)$ 得到一个 n-1 次的商式和一个余数，将这个 n-1 次商式再除以 $(x-x_0)$ 得到一个 n-2 次的商

式和一个余数，……，等等。以 n=4 为例，如下

$$a_0x^4+a_1x^3+a_2x^2+a_3x+a_4$$
$$=(b_0x^3+b_1x^2+b_2x+b_3)(x-x_0)+b_4$$
$$=((c_0x^2+c_1x+c_2)(x-x_0)+c_3)(x-x_0)+b_4$$
$$=(((d_0x+d_1)(x-x_0)+d_2)(x-x_0)+c_3)(x-x_0)+b_4$$
$$=(((e_0(x-x_0)+e_1)(x-x_0)+d_2)(x-x_0)+c_3)(x-x_0)+b_4$$

显然，这里的 e_0，e_1，d_2，c_3，b_4就是我们所要求的 w_0，w_1，w_2，w_3，w_4。

我们先看一遍多项式除法的过程

$$\begin{array}{r|llllll} & b_0 & +b_1 & +b_2 & +b_3 & & \\ \hline 1\ -x_0 & a_0 & +a_1 & +a_2 & +a_3 & +a_4 & \\ & b_0 & -b_0x_0 & & & & \\ \hline & a_1+b_0x_0 & +a_2 & & & & \\ & b_1 & -b_1x_0 & & & & \\ \hline & & a_2+b_1x_0 & +a_3 & & & \\ & & b_2 & -b_2x_0 & & & \\ \hline & & & a_3+b_2x_0 & +a_4 & & \\ & & & b_3 & -b_3x_0 & & \\ \hline & & & & a_4+b_3x_0 & = & b_4 \end{array}$$

可以看出

$$b_0=a_0$$
$$b_1=b_0x_0+a_1$$
$$b_2=b_1x_0+a_2$$
$$b_3=b_2x_0+a_3$$
$$b_4=b_3x_0+a_4$$

如果我们把 a_0～a_4写在上行，b_0～b_4写在下行

$$\begin{array}{lllll} a_0 & a_1 & a_2 & a_3 & a_4 \\ b_0 & b_1 & b_2 & b_3 & b_4 \end{array}$$

则可以得出如下公式：

本数＝（左数乘以 x_0）＋上数　　（公式 1 ）

其中 b_0没有“左数”，但若将它看作左数为 0，同样符合上述公式。

这个问题中要作多遍的长除法，每一遍长除法都和上述的同理，可以列出下表

$$\begin{array}{lllll} a_0 & a_1 & a_2 & a_3 & a_4 \\ \| & \downarrow & \downarrow & \downarrow & \downarrow \end{array}$$

$b_0 \rightarrow b_1 \rightarrow b_2 \rightarrow b_3 \rightarrow b_4$

$\| \quad \downarrow \quad \downarrow \quad \downarrow$

$c_0 \rightarrow c_1 \rightarrow c_2 \rightarrow c_3$

$\| \quad \downarrow \quad \downarrow$

$d_0 \rightarrow d_1 \rightarrow d_2$

$\| \quad \downarrow$

$e_0 \rightarrow e_1$

表 7.1　秦九韶法的计算次序

表中的箭头表示计算过程的关系，例如 c_2 就是由 c_1 和 b_2 按公式 1 求出。这个表连同上面的公式 1，就是秦九韶法。

实现上表中的算法，具体的程序可以有多种设计。以下采用的程序是只用一个数组的方案。从上表中可以看出，下数求出后上数就没用了，可以冲掉。这样最后的

e_0，e_1，d_2，c_3，b_4

仍然放在

a_0，a_1，a_2，a_3，a_4

中。而 e_0，d_0，c_0，b_0 和 a_0 都是相等的，可以保留 a_0 不动。

程序清单如下：

```
PROGRAM qinjiushao(input,output);
  CONST
    n=8;
  VAR
    a:ARRAY [0..n] OF real;
    x0:real;
    i,j:integer;
BEGIN
  writeln('Coefficients in the polynomial:');
  FOR i:=0 TO n DO read(a[i]);
  readln;
  write('The value of x0 : '); readln(x0);
  FOR i:=n DOWNTO 1 DO
    FOR j:=1 TO i DO a[j]:=a[j-1]*x0+a[j];
  writeln('Result coefficients:');
  FOR i:=0 TO n DO writeln(a[i])
END .
```

运行结果：

```
Coefficients in the polynomial:
1  -0.9  1.1  -1.2  0.8  -0.8  0.5  -0.4  1
The value of x0 : 0.45
```

```
Result coefficients:
 1.0000000000E+00
 2.7000000000E+00
 3.9350000000E+00
 3.0457500000E+00
 1.4412500000E+00
-4.3589375005E-02
-1.4114959375E-01
-2.9106910469E-01
 8.6646425465E-01
```

上述算法求出结果后原系数就丢了，若需要保留原系数，可以先将其复制到另一个数组中再作上述计算。

秦九韶法在处理多项式及高次方程一类问题时很有用，它的计算步骤比较简单，故效率较高。

在实际应用中，有时候并不需要将表 7.1 中的步骤进行完。例如有时只需要求 $f(x_0)$ 的值，即求表中的 b_n，只需要进行表中的第一行运算。这时若不想破坏原系数，可以不将中间结果放入数组，例如可将 b_0～b_n 公用一个变量 y，最后得到 $f(x_0)$ 值就在 y 中，可以如下设计：

```
y:=a[0];
FOR i:=1 TO n DO y:=y*x0+a[i]
```

有时候需要求 $f(x_0)$ 及 $f'(x_0)$ 两个值，即需要计算表中的两行，此时可以引入 y，z 两个变量， y 先后存放 b_0～b_n，z 先后存放 c_0～c_{n-1}：

```
y:=a[0]; z:=y
FOR i:=1 TO n-1 DO
  BEGIN y:=y*x0+a[i]; z:=z*x0+y END;
y:=y*x0+a[n]
```

结果 $f(x_0)$ 的值在 y 中，$f'(x_0)$ 的值在 z 中。

比较一下就会发现，秦九韶法的计算步骤比起直接按原式及导数的解析式计算，要简单得多。

7.3 多维数组

7.3.1 多维数组的概念

前面已说过，数组的成分类型也可以是构造类型。习惯上将成分类型又是数组的数组称作多维数组。如下面的数组 a：

```
VAR
  a:ARRAY[1..5] OF ARRAY[1..7] OF integer;
```

就是一个二维数组。它是由 5 个成分组成，每个成分都是 ARRAY[1..7] OF integer 类型

的一维数组。而 ARRAY[1..7] OF integer 类型的数组又是由 7 个成分组成， 每个成分都是整数。习惯上将这样成分类型为一维数组的数组称作二维数组，将成分类型为二维数组的数组称作三维数组，……等等。

PASCAL 中规定，设 t1，t2，……，tn 是下标类型，T 是成分类型，则形如

```
ARRAY[t1] OF ARRAY[t2] OF ... ARRAY[tn] OF T
```

的格式可以缩写为

```
ARRAY[t1,t2, ... ,tn] OF T
```

按照这个缩写规则，上面的 a 数组的说明可以写作

```
VAR
  a:ARRAY[1..5, 1..7] OF integer;
```

多维数组中最简单的是二维数组。二维数组习惯上可以想象为由若干行，若干列组成的表，例如上面的 a 数组就可以看作由 5 个行组成，每行 7 个整数。如

```
10  21  34   5  22  45  89
 6  11 101  25  37 111  80
22  15  37  77  21  65  66
20  16  34  67  22  99 101
55   0  12 121  40  49  38
```

它可以看作数学上的一个矩阵。当然，从写出来的形式上看，它也可以看成由 7 个列组成，但是从上面类型描述的原则上看，这里的一个“列”并不是数组 a 中的成分。

又如，下面的 b 数组也是一个二维数组：

```
TYPE
  weekday=(sun,mon,tue,wed,thu,fri,sat);
  course=ARRAY [mon..fri,1..3] OF char;
VAR
  b:course;
```

其第一个下标类型是一个枚举的子域，第二个下标是整数的子域，内层的成分是字符。

从上面叙述可知，多维数组是一种层次性的结构。它在存储器中的实际存放形式虽然标准上不作规定，但是为了便于编译程序处理，实际的 PASCAL 系统中的存放形式总是和上述定义的层次一致的。如上述的 a 数组，既然可以看作 5 个大成分，而每个大成分又分成 7 个小成分，则它在内存的存放也是分成 5 个区段，每段中存放 7 个整数。如果与上面的行列模型对照，则是同一行的排在一起，后一行接在前一行之后。所以一般说，PASCAL 中数组是“以行为主序”存储的。

7.3.2 多维数组的引用

与上一节讲的一样，多维数组变量只要类型相同，也可以整体互相赋值。与一维数组的不同点是：多维数组的成分也是数组（降低维数），需要时也可以整体使用。

例如像上面定义的 a 数组，语句

```
a[3]:=a[2]
```

是合法的，它将 a 中的第二行和第三行分别看作一维数组，整行地将第二行送到第三行中。

又例如

```
TYPE
  col=ARRAY [1..7] OF integer;
VAR
  a:ARRAY [1..5] OF col;
  c:col;
```

这里 a 是二维数组，c 是一维数组。这里数组 a 的结构和前面所举的例子完全相同， 不过按现在的这种定义形式，a 数组中的一行就和 c 数组的整体属于同一类型。所以，像下面这样的语句

```
c:=a[2];
a[3]:=c
```

都是合法的。这两句将 a 中的第二行整个送入一维数组 c 中，又从一维数组 c 中整个送入 a 的第三行。

在实际应用中，像上面那样整体引用或部分整体引用并不太多，更多的是引用最内层的简单成分。

例如要引用上述 a 数组中第二行第四列的成分，应该看到它是一维数组 a[2] 中的第四个成分（下标为 4），所以按照上一节讲的下标变量的书写形式，应该写作

```
a[2][4]
```

这是一个二维数组的最内层成分。若是三维、四维或更高维数组中的最内层成分，这样书写后面会带更多的下标。

PASCAL 中又规定，设 v 是多维数组变量，e1，e2，……，en 是下标表达式，则形如

```
v[e1][e2]...[en]
```

的下标变量可以缩写为

```
v[e1, e2, ... , en]
```

按照这个缩写规则，上面的 a[2][4]可以写作 a[2,4]。

可以看到，这种写法中，两个下标是行在前，列在后，和数学上表示矩阵元素时的习惯是一致的。

上一节我们见到，对一维数组成分的遍历访问常使用 FOR 循环，同样道理， 对二维数组内层成分的遍历访问常使用二重的 FOR 循环。

例如，某车间五个职工 1996 年度完成工作量如下表：

工号	一季度	二季度	三季度	四季度
1011	426.4	398.2	489.1	430
1012	538.5	507.3	447.3	492.2
1013	378.8	477.7	462.2	399.4
1014	429	448.6	513.7	456.5
1015	511.1	492.5	543.9	524.1

假如某数据处理程序需要将这一批工作量数据存入内存，可以采用如下形式的二维数组 w：

```
VAR
  w:ARRAY [1011..1015, 1..4] OF real;
```

这里第一个下标是工号，第二个下标是季度。

程序中要通过键盘输入这一批数据可以采用如下的二重循环

```
FOR i:=1101 TO 1105 DO
  BEGIN
    FOR j:=1 TO 4 DO read( w[i,j]);
    readln
  END;
```

执行时，用户在键盘上需要以以下形式输入数据：

```
426.4  398.2  489.1   430
538.5  507.3  447.3   492.2
378.8  477.7  462.2   399.4
429    448.6  513.7   456.5
511.1  492.5  543.9   524.1
```

这种输入方式称为按行输入，即先输入第一行，再输入第二行……。如果需要的话也可以按列输入：

```
FOR i:=1 TO 4 DO
  BEGIN
    FOR j:=1101 TO 1105 DO read(w[j,i]);
    readln
  END;
```

这时键盘输入的每一行是上表中的一列，输入形式如下：

```
426.4  538.5  378.8  429    511.1
398.2  507.3  477.7  448.6  492.5
489.1  447.3  462.2  513.7  543.9
430    492.2  399.4  456.5  524.1
```

假设程序中另有一个一维数组 total 用来存放各人全年工作量，其变量说明的形式为

```
total:ARRAY [1101..1105] OF real;
```

则求各人全年工作量的程序可以如下：

```
FOR i:=1101 TO 1105 DO
  BEGIN
    total[i]:=0;
    FOR j:=1 TO 4  DO tolal[i]:=total[i]+w[i,j]
  END;
```

若需要将工号、各季度工作量、年工作量对照列表输出，程序可以如下：

```
FOR i:=1101 TO 1105 DO
  BEGIN
```

```
        write(i:4);
        FOR j:=1 TO 4 DO write(w[i,j]:7:1);
        writeln(total[i]:10:1)
    END;
```

7.3.3 应用举例

[例 1] 求一个 6 行 7 列的二维数组中这样的成分的位置，它在行上最小，在列上却是最大，如果没有这样的成分则输出'Not found'字样。假设已知同一行中各数必不相等， 编程时不考虑“并列最小”这种情况。

总算法如下（其中标志 found 记整个数组存在这种成分与否，标志 found0 记当前行的最小成分是否也是一列的最小成分）：

输入数据到 a 数组；
初始化标志 found 为 false；
FOR i:=1 TO 6 DO
【找出第 i 行最小成分 a[i,c]；
判断 a[i,c]在第 c 列是否最大（结果存入标志 found0)；
若是，则【输出 i，c，a[i,c]；令 found 为 true 】
】；
若 found 为 false 则输出“Not found.”字样 。

其中，“找出第 i 行最小成分 a[i,c]”的算法细化如下：

c:=1；
FOR j:=2 TO 7 DO 若 a[i,j]<a[i,c]则令 c:=j 。

“判断 a[i,c]在第 c 列是否最大”的算法细化如下：

令 found0 为 true；
令 j:=1；
WHILE j<=6 且 found0 DO
【若 a[j,c]>a[i,c]则令 found0:=false；
令 j 增 1
】 。

按照这个算法，可以编出如下的程序：

```
PROGRAM exam7a(input,output);
  VAR
   a:ARRAY [1..6,1..7] OF integer;
   i,j,c:integer;
   found,found0:boolean;
BEGIN
  writeln('Please input 6*7 matrix:');  {输入数据到 a 数组}
  FOR i:=1 TO  6 DO
    BEGIN
```

```
        FOR j:=1 TO 7 DO read(a[i,j]);
        readln
      END;
    writeln;
    found:=false;                          {初始化标志 found}
    FOR i:=1 TO 6 DO
      BEGIN
        c:=1 ;                             {找出第 i 行最小成分 a[i,c]}
        FOR j:=2 TO 7 DO
          IF a[i,j]<a[i,c] THEN c:=j;
        found0:=true; j:=1;                {判断 a[i,c]在第 c 列是否最大}
        WHILE (j<=6) AND found0 DO
          BEGIN
            IF a[j,c]>a[i,c] THEN found0:=false;
            j:=j+1
          END;
        IF found0 THEN                     {若是，则输出且置 found}
          BEGIN
            writeln(i:2,c:3,a[i,c]:5);
            found:=true
          END
      END;
    IF NOT found THEN writeln('Not found.')
  END.
```

以下是两次运算结果。第一次：

```
Please input 6*7 matrix:
93 72 68 21 34 92 22
15 43 52 91 88 76 23
42 99 18 63 50 44 27
77 62 59 83 40 55 42
86 97 32 38 61 69 63
74 85 46 57 68 33 44

Not found.
```

另一次：

```
Please input 6*7 matrix:
93 72 37 21 34 92 22
15 43 32 91 88 76 23
```

```
42 99 41 63 50 44 45
77 62 33 83 40 55 42
86 97 40 38 61 69 63
74 85 39 57 68 33 44

 3  3   41
```

这个程序还有改进的余地。因为这样的成分不会超过一个（为什么？请考虑），所以外层循环可以在一旦找到后就结束。具体程序留给读者考虑。

[例 2] 矩阵相乘（假设两矩阵分别为 3 行 4 列和 4 行 5 列）：

设矩阵 A 为 m×k 阶，矩阵 B 为 k×n 阶，矩阵 C=A*B。矩阵相乘的计算公式为

$$c_{ij}=\sum_{w=1}^{k}(a_{iw}*b_{wj}) \qquad i=1,2,\ \cdots\ m\ ;\ j=1,2,\ \cdots\ n$$

这个公式又可以称作“行乘列”法，即 A 矩阵的第 i 行与 B 矩阵的第 j 列的“点乘”积，等于 C 矩阵的第 i 行第 j 列的元素。程序清单：

```
PROGRAM exam7b(input,output);
  CONST
    m=3; k=4; n=5;
  VAR
    i,j,w,s:integer;
    a:ARRAY [1..m,1..k] OF integer;
    b:ARRAY [1..k,1..n] OF integer;
    c:ARRAY [1..m,1..n] OF integer;
BEGIN
  writeln('Please input matrix A :');       {输入 A 阵}
  FOR i:=1 TO m DO
    BEGIN
       FOR j:=1 TO k DO read(a[i,j]);
       readln
    END;
  writeln('Please input matrix B :');       {输入 B 阵}
  FOR i:=1 TO k DO
    BEGIN
       FOR j:=1 TO n DO read(b[i,j]);
       readln
    END;
  FOR i:= 1 TO m DO                         {乘法}
    FOR j:= 1 TO n DO
```

```
        BEGIN
          s:=0;
          FOR w:=1 TO k DO s:=s+a[i,w]*b[w,j];
          c[i,j]:=s
        END;
    writeln('Matrix C :');                    {输出 C 阵}
    FOR i:=1 TO m DO
      BEGIN
        FOR j:=1 TO n DO  write(c[i,j]:5);
        writeln
      END
END.
```

运行结果如下：

```
Please input matrix A :
  4 10 11 20
  6  9 17 19
 16  8 41 12
Please input matrix B :
 10 13  6  9 18
 11 16 18 11 12
 14 10 22  4 13
  8 15 11  5  7
Matrix C :
  464  622  666  290  475
  549  677  781  316  570
  918  926 1274  456 1001
```

7.4 紧缩数组及其它紧缩构造类型

7.4.1 非紧缩存储与紧缩存储

一般的电子计算机，总是以若干个二进制位作为存储单元。例如 50 年代我国研制的第一台计算机 103 机是以二进制 30 位作为存储单元，后来的一些小型机以二进制 16 位作为存储单元。现在的微型计算机大都是以一个字节（二进制 8 位为一个字节）为基本存储单元， 但同时又具有若干以 16 位或 32 位等为单位的操作功能。

为了便于编程，一般存储器中存放的一个数据，所占用的二进制位数最好是整个的存储单元，或它的整数倍。但是有些数据的信息不需要这么多位，例如布尔型的数据只要二进制一位就可以表示了，字符型的数据即使按扩充的 ASCII 编码也只需要 8 位。

这时，假如将多个变量合并占用存储空间，虽然可以充分利用数据空间，但是对这样

的变量读写操作复杂了，目标程序加长了，总的效果不合算。所以对于简单变量，这种情况下都是占用整个的存储单元，多余的位就空闲着。这就是普通的存储方式，“非紧缩”的存储方式。

但是对于构造型的数据，如数组，情况就不同了。数组中可能包括大量同类型简单成分数据，如果每个成分浪费几位，总的浪费可能就不小。如能合并存放，即使程序长一些，总的还是合算的。所以 PASCAL 语言对构造型数据提供了一种“紧缩”的存储方式。

如上所述，紧缩存储的数据访问操作复杂，所以它对空间的节省是以机时的浪费为代价的，因此在使用的时候要注意权衡利弊。

因为不同的计算机存储单元的大小可能不同，所以紧缩存储的数据在不同计算机上可能采取不同的具体存储办法。不过可以说，大多数的数值性数据，如整数、实数等等，本来其长度就是存储单元的整数倍，所以以这些数据为成分的构造型数据，紧缩与不紧缩实际上是一样的。

7.4.2 紧缩数组

本章前面所述的数组都是“非紧缩”的数组。需要的时候可以引入紧缩的数组类型。紧缩数组类型的新类型描述格式是在前面所述的非紧缩数组类型的描述格式之前加上一个字符号 PACKED。如下面的数组 a 就是一个紧缩的布尔数组：

```
VAR
  a:PACKED ARRAY[1..5] OF boolean;
```

注意按规定这里的 PACKED ARRAY[1..5] OF boolean 整体是一个新类型描述格式， 不要认为是在一个类型表记符前加上了 PACKED。例如，下面的写法是错误的：

```
TYPE
  t1=ARRAY [1..5] OF boolean;
VAR
  a: PACKED t1;
```

对紧缩数组的引用，不论是整体引用还是引用其成分，格式和规则都和前面讲过的非紧缩数组相同。

程序中有时需要将紧缩数组的内容转换为非紧缩的，或将非紧缩数组的内容转换为紧缩的。因为紧缩数组和非紧缩数组显然不是同一类型，所以它们整体间直接互相赋值是不合法的。为此，PASCAL 提供了两个预定义过程 pack 和 unpack，用来在紧缩数组和非紧缩型数组之间相互转换。pack 过程可将非紧缩数组中的数据压缩到一个紧缩数组中去，unpack 过程可将紧缩数组中的数据松展到一个非紧缩数组中去。

设数组 a 是一个

```
ARRAY [s1] OF t
```

类型的非紧缩数组变量，数组 z 是一个

```
PACKED ARRAY [s2] OF t
```

类型的紧缩数组变量，则过程语句

```
pack(a,i,z)
```

的作用是：将 a 数组中从下标为 i 的成分开始，依次将各成分赋给 z 数组从第一个成分开

始的各成分，直到 z 数组的最后一个成分被赋值为止。

过程语句

```
unpack(z,a,i)
```

的作用是：将 z 数组从第一个成分开始的各成分依次赋给 a 数组中从下标为 i 的成分开始的各成分，直到 z 数组的最后一个成分传送入 a 数组中为止。

显然，这两个过程都要求 a 数组中从下标为 i 的成分开始的后一部分成分个数不得少于 z 数组的成分个数。换句话说，紧缩数组 z 的下标要对应于非紧缩数组 a 的下标中的一段，而 i 就是这一段的起点。这两个过程语句中的 i 都是值参数，可以是表达式，其值必须对 a 数组的下标类型 s1 赋值相容。

对于普通的紧缩数组，了解以上所讲内容就可以了，但 PASCAL 中对于一类特殊的紧缩字符数组（字符串类型）还提供了若干特别的规定及特别的操作，详见下一节介绍。

7.4.3 其它紧缩构造类型

PASCAL 中不仅有紧缩的数组，对于后面几章将讲到的其它构造类型，也有相应的紧缩的该种构造类型，如紧缩集合、紧缩记录、紧缩文件等等。后面各章所述的构造类型的新类型描述格式都是描述“非紧缩”的类型。而紧缩的构造类型的新类型描述格式是在相应的非紧缩构造类型的描述格式之前加上一个字符号 PACKED。

对紧缩的构造型变量及其成分的引用格式和规则都和非紧缩的同一类构造型变量一样。

因为用得不多，而且也没有其它必须讲述的内容，所以以后几章讲述其它构造类型时均只按非紧缩型介绍。

7.5 字符串常量及字符串类型

7.5.1 字符串

前面讲 write 语句时我们已经介绍过字符串。如：

```
write('Please input number:')
```

中的'Please input number:'就是一个字符串。

这里要说明的是：这种字符串可以看作是下面将要介绍的字符串类型的字面常量。字符串常量不仅可以有字面常量，同样也可以有符号常量。字符串的符号常量可以在常量定义部分中定义，如：

```
CONST
  str='Please input number:';
```

作了这样的定义后，语句

```
write(str)
```

就和上面所举的一样了。

7.5.2 串类型

紧缩的数组类型，若成分为字符型，下标类型为整数的子域，且下界为 1，上界大于 1，则这种类型称作串类型，或称字符串类型。如：

```
TYPE
  st=PACKED ARRAY[1..10] OF char;
VAR
  a,b:st;
```

这里 st 是一个串类型，a 和 b 都是串类型的变量，或称“串变量”、“字符串变量”。

串类型的成分个数习惯上称作字符串的长度。

上面是标准的规定，有些实际系统有所放宽，只要是紧缩的字符数组就算是串类型。

串类型在文字信息处理中应用很多。为实用的需要，PASCAL 中对串类型提供了若干与普通数组不同的特有功能。这些功能包括以下四个方面：

1.可有常量。普通数组类型是没有“常量”的，串类型却有字面常量及符号常量的表示形式。这就是上面所述的字符串常量。

2.整体赋值。普通数组虽然也能整体赋值，但条件较严。按第六章 6.4 节所述， 普通数组只有类型同一时才算赋值相容。对于字符串类型，则只要长度相等就算类型相容，而只要类型相容就算赋值相容。所以长度相等的字符串变量就可以互相整体赋值。只要长度相等，也可以用字符串常量给串变量赋值。如：

```
a:='this book.'
```

等效于

```
BEGIN
  a[1]:='t';a[2]:='h';a[3]:='i';a[4]:='s';a[5]:=' ';
  a[6]:='b';a[7]:='o';a[8]:='o';a[9]:='k';a[10]:='.'
END
```

这里的赋值号右边的字符串长度必须与 a 数组长度相等。 应注意字符串书写时两端的撇号是字面常量的书写规则，并不是数据成分，计算长度时不可计入。还有，串中若有连续书写的两个撇号，实际上代表一个撇号字符，只应算一个成分。还有，串中空格字符不应忽略。

3.相容的字符串类型数据间可以比较大小。

这里“相容的”即长度相等的字符串类型。它们可以进行

```
=,  <,  >,  <>,  <=,  >=
```

六种比较运算。比较运算的结果都是布尔型值。

两个串类型数据间比较时，是从头开始，对应成分逐个比较，遇到第一对不相等的成分时，按字符型数据比较的原则比较其大小，结果算作整个字符数组的大小。若全部成分都对应相等，则认为两个串相等。如执行

```
a:='this book.';
b:='this bike.'
```

以后，若再作

```
a>b
```

的比较，结果应为 true。这是因为它们前六个字符都对应相等，而 a[7]>b[7]。

字符串数据的这种比较原则称作“词典序”，因为它和词典中单词排列先后的原则是

一样的。

4.字符串类型数据的输入输出

以前我们已经知道字符串常量可以在输出语句中直接输出，这里要说的是串类型的变量同样可以在 write 语句或 writeln 语句中作为输出项。如对于如上所述的串变量 a，

```
write(a)
```

等效于

```
write(a[1],a[2],a[3],a[4],a[5],a[6],a[7],a[8],a[9],a[10])
```

若执行

```
a:='this book.'; writeln(a)
```

则显示为

```
this book.
```

有的 PASCAL 系统还可以用 read 语句直接输入数据给串变量，不过标准中无此规定。有此功能的实际系统对此功能也往往有些细节上的特别规定。如有些系统规定，如果 read 参数中直接写串变量，则执行时依次从输入设备读入字符写到该变量从头开始的各成分中去；但若未读够长度时遇回车键，并不把回车号送到数组中，而是立即结束读操作（这个回车键也并不读入），并把数组中剩余的没有填满的部分全填入空格号。

如果我们所用的具体 PASCAL 系统没有直接读入字符串的功能，可以用循环读入字符的办法来编写输入字符串的功能。在没有直接读入字符串功能的系统上编写相当于上述细节规定的输入字符串功能，可以如下（以上面所定义的 a 数组为例）：

```
FOR i:=1 TO 10 DO
  IF eoln THEN a[i]:=' '
          ELSE read(a[i])
```

这里 eoln 是判断输入行结束的函数。

7.5.3 实例

[例 1] 从键盘输入十个单词，全用小写，每个单词一行，长度不超过 20 个字符。计算机输出其中按辞典序该排在最前面的一个及最后面的一个。

这个问题即以前见过的求最小值与最大值的问题，这里的“小”与“大”正好符合上面说的字符串比较的法则。

上面说过，字符串比较时要求长度相等，故我们可以把每个单词统一补够 20 个字符，后面不足部分用空格号补充。因为空格号的 ASCII 编码小于任何小写字母， 故这并不影响按辞典序比较先后。

程序清单见下，这里初始化时“最小值”取最“大”，“最大值”取最“小”，这种做法前面已经见过了：

```
PROGRAM words(input,output);
  VAR
    a,min,max:PACKED ARRAY [1..20] OF char;
    i,j:integer;
BEGIN
```

```
    min:='zzzzzzzzzzzzzzzzzzz';
    max:='                   ';
    FOR i:=1 TO 10 DO
      BEGIN
        readln(a);
        IF a<min THEN min:=a;
        IF a>max THEN max:=a
      END;
    writeln; writeln(min); writeln(max)
  END .
```

这里假定所用系统有直接读入字符串的功能，而且其功能细节正好如前面所述。如果所用系统没有直接读入字符串的功能的话，则可采用前面介绍的办法编写输入操作，将

```
  readln(a);
```

一句改为以下一段

```
  FOR j:=1 TO 20 DO
    IF eoln THEN a[j]:=' '
            ELSE read(a[j]);
  readln;
```

[例 2] 从键盘输入十个单词，每个单词全用小写，长度不超过 20 个字符，每个一行。计算机将其按照辞典序排列输出。

要排序，必须先将它们放入一个数组。但字符串本身已经是数组，再将 10 个字符串组成一个数组，这实际上就成了二维数组。不过与以前见到的二维数组有一个不同点：它的内层是“紧缩的”。

程序清单如下，这里排序用的是选择法：

```
  PROGRAM sortwords(input,output);
    TYPE
      str=PACKED ARRAY [1..20] OF char;
    VAR
      vec:ARRAY [1..10] OF str;
      t:str;
      i,j,p:integer;
  BEGIN
    writeln('Input 10 words:');              {输入}
    FOR i:=1 TO 10 DO
      BEGIN
        FOR j:=1 TO 20 DO
          IF eoln THEN vec[i,j]:=' '
                  ELSE read(vec[i,j]);
```

```
      readln
    END;
  FOR i:=1 TO 9 DO                          {选择法排序}
    BEGIN
      p:=i;
      FOR j:=i+1 TO 10 DO
        IF vec[j]<vec[p] THEN p:=j;
      IF p<>i THEN
        BEGIN
          t:=vec[i];vec[i]:=vec[p];vec[p]:=t
        END
    END;
  writeln; writeln('Output:');              {输出}
  FOR i:=1 TO 10 DO writeln(vec[i])
END.
```

运行结果：

```
Input 10 words:
tail
absent
photomask
revolution
revisionist
photomagnetoelectric
mostly
motherland
absence
absolute

Output:
absence
absent
absolute
mostly
motherland
photomagnetoelectric
photomask
revisionist
revolution
```

```
        tail
```

注意这个程序中，排序时是将一个字符串数据整体对待的，故只用一个下标，很像是一个一维数组的形式，而输入时又是按字符处理，故用两个下标。

这一题是按假定所用系统没有直接读入字符串的功能而设计的，如果所用系统有直接读入字符串的功能，则可将输入部分程序简为

```
        writeln('Input 10 words:');
        FOR i:=1 TO 10 DO  readln(vec[i]);
```

习题

7.1 输入 100 个学生的成绩，统计不及格的、60～69 分的、70～79 分的、80～89 分的、90～100 分的各有多少人。

7.2 设计一种改进的冒泡法程序来实现 7.2 节[例 4]。改进的思路如下：

原基本冒泡法每一趟结尾比上一趟只提前一步，现改进为：每一趟进行中记下最后一次交换的位置，若该位置不是最后一对，则说明后面的各成分必然都是更大的数且已经排好次序，所以下一趟结尾可以提前到这一趟中最后一次交换的位置的前一步。若某一趟最后交换位置是第一对数，或这一趟中就没有交换，则整个排序结束。

7.3 设计一种简化的选择法程序来实现 7.2 节[例 4]。简化的思路见课文中[例 4] 之后介绍。

7.4 输入 30 个学生学号和 3 门课成绩，按其总分从大到小的顺序打印学生成绩表。请按冒泡法和选择法设计两种方案。

7.5 输入一元五次方程的各项系数及一个根的较粗略的近似值，用牛顿迭代法（见第五章 5.4 节介绍）求该根的较精确的值。其中求函数值及导数使用秦九韶法（见 7.2 节结尾的介绍）。

注：本题目程序中不要求考虑迭代失败的情况，但实际上运行时若初值选取不当有可能迭代失败，即迭代序列发散，此时程序发生溢出或陷入死循环。故试运行此程序时注意选取接近一个非重根的值作为初值。

7.6 求二维数组中行上列上都是最小的成分，注意这种成分可能不止一个，要将它们的下标及值都打出来。

7.7 将 7.5 节[例 2]改用冒泡法实现。

第八章　子程序——过程和函数

8.1　PASCAL 中的子程序概述

子程序的概念早在高级语言出现以前，在用机器指令设计程序时就已经提出来了。但真正深刻认识到它的意义，则是后来的事。

最初采用子程序的目的主要是缩短程序篇幅。我们已经知道，规则化重复执行的操作可以编成循环，但无规则重复出现的操作就不能用循环结构来缩短篇幅了，若不采用子程序就只能在程序中多处重复编写相同的段落。所谓子程序是一种单独编写的程序段，主程序中凡需要这一段的功能处可写一条“调用”指令来调用这一段子程序。若有多处需要同样的功能只要多用几条调用指令就行了。这样，就可以避免多处重复编写相同的程序段落。运行时，遇调用指令就转去执行子程序，而到了子程序的结尾处则采用某种技术自动地转回到原调用指令的下一条指令处去继续执行。

如果仅仅是为了缩短程序篇幅，那么子程序只有在两处以上被调用时才有意义，若只在一处调用，则不如不用子程序，直接将这一段功能编入主程序中去更好。

后来，随着结构化程序设计方法的提倡，人们越来越认识到抽象思想方法在程序设计中的意义。而子程序恰好是抽象方法的一种很方便的实现形式。在前面的章节中，由上到下，由粗到精对算法的分层次描述，只能写在编程前的计划文字中，而具体的程序则不得不按最下层的描述编写。现在有了子程序，就可以将这种层次直接写入程序中。调用指令可以看作是对子程序整个功能的抽象代表，而编出的子程序则是它的细化。这样阅读主程序时就可以着眼于较高的抽象层次，容易首先看清大局而不至于陷入繁琐的细节，程序的可读性自然就好。子程序允许多层嵌套，正好可以表达我们对问题的多层分解。这样，我们就容易编出有高度条理性的程序，整个软件设计工作也就比较容易采用条理化的组织管理。

另外，有些具有通用功能的程序段，可能在许多不同的实际程序中都会用到，人们常常将其编成标准子程序的形式，需要时可以将它复制到自己的程序中。

所以，现在人们采用子程序，最主要的目的已不是缩短篇幅。程序中有些操作只在一处引用，也写成子程序，就是这个理由。

上述关于子程序的思想，对任何语言都是适用的。而对于 PASCAL 这样的高级语言，则还有某些特殊点。这一类高级语言中，有些操作要求产生一个隐式的运算结果(返回值)。所以我们就以此为区别将 PASCAL 中的子程序分成两种，分别称作过程和函数。过程不带返回值，调用时算作一个语句。函数带返回值，调用时算作表达式。如下表：

PASCAL 子程序	运行结果	调用形式
过程	不带返回值	过程语句
函数	带返回值	函数命名符（表达式）

表 8.1　PASCAL 子程序

PASCAL 中子程序须编写在程序的说明部分，称作过程说明和函数说明。过程说明或函数说明中定义一个标识符作为过程名或函数名，以便主程序中调用。

程序是从主程序语句部分的第一条语句开始运行的，运行过程中遇到程序中引用过程名或函数名来调用子程序时就转去执行函数说明或过程说明中编写的子程序。一个子程序执行完后会自动回到主程序中原调用处继续运行主程序。

PASCAL 中子程序可以带有参数，调用时可以通过代入的参数来决定其具体的操作。过程和函数的说明中代表参数的记号（标识符）称作形式参数（常简称为“形参”），调用时代入的参数称作实在参数（常简称为“实参”）。

子程序执行中还可以调用子程序，这是嵌套的调用。这时候，调用子程序的子程序，相对地可以看作是它所调用的那个子程序的“主程序”。“主程序”一词不是严格的术语，习惯上有时用来指整个程序的语句部分中编写的语句序列，也有时候取这种相对的意思。

除了这种须在说明部分中说明才能在语句部分中引用的过程和函数外，PASCAL 中还提供了一批预定义过程和预定义函数。预定义的过程和函数可以不必说明而直接调用，就好像已经作过说明一样。

已经学过的预定义过程有 read，readln，write，writeln 四个。已经学过的预定义函数比较多，如 abs，sin，cos，sqr，sqrt，……等等都是。

基本的语法图见图 8.1[①]，其中的“标识符”就是为过程或函数定义的名字。

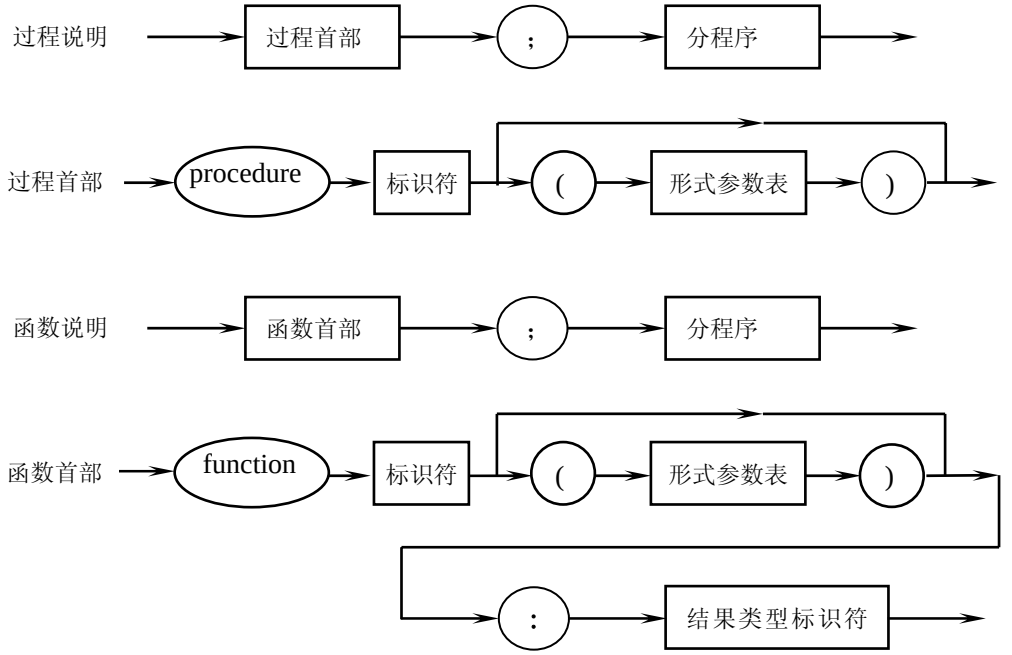


图 8.1　过程说明和函数说明的基本的语法图

第一章的图 1.10 和第二章的图 2.1 中已经指明了过程说明和函数说明在程序中的位

[①] 这个语法图尚未包括 8.7 节介绍的情况，故不算完整的语法图。

置。每个过程说明和函数说明的后面总是紧跟一个分号，这在图 2.1 中已经画出，故图 8.1 中就不再包括这最后的分号了。

图 8.1 中可以看到，过程说明和函数说明中又包含“分程序”。 过程说明中的分程序称作过程分程序，函数说明中的分程序称作函数分程序。这种分程序仍然符合图 2.1 的语法，也就是说，它也要分成说明部分和语句部分。执行子程序时是从其语句部分的第一条语句开始的。分程序的说明部分中可以定义一些局部的标识符，这些标识符只能在这个分程序中引用。关于局部标识符的作用域在后面还要详述。过程分程序和函数分程序的说明部分里还可以有过程说明和函数说明，这称作子程序的嵌套说明或嵌套定义。

作为形式参数的标识符是在首部中的“形式参数表”中定义的，它也属于以该分程序为作用域的局部标识符。“形式参数表”的语法规则是：整个形式参数表由一个或多个“形式参数段”组成，段间以分号分隔，每个“形式参数段”中定义一个或多个同一种类的形式参数。例如某个形式参数表为

```
x,y:real; VAR m,n:integer
```

则其中共两个形式参数段。第一段中定义了形式参数 x 和 y，第二段中定义了形式参数 m 和 n，总共四个形参。第二段开头的 VAR 表示 m 和 n 是变量参数，第一段开头没有字符号，表示 x 和 y 是值参数。第一段的两个形参是 real 形，第二段的两个形参是 integer 型。

PASCAL 中子程序的参数有五种：值参数、变量参数、过程参数、函数参数和可调节数组参数。PASCAL 标准文本中对定义这五种形式参数的“形式参数段”各有一个名称，称作各种“参数指明”，如定义值参数的形式参数段称作“值参数指明”，定义变量参数的形式参数段称作“变量参数指明”，等等，见表 8.2。

形式参数段	
	值参数指明，例　x,y:real
	变量参数指明，例　VAR　m,n:integer
	过程参数指明，例　PROCEDURE p(x,y:char)
	函数参数指明，例　FUNCTION f(t:real):real
	可调节数组参数指明，例　VAR a:ARRAY[s1..s2:integer] OF real

表 8.2　各种形式参数段

其中可调节数组参数由于目前各种机器上的大多数 PASCAL 语言尚未实现这一功能，所以本书只作简要介绍。过程参数和函数参数是在该子程序中又调用其它子程序时，需要通过实在参数中代入的过程名和函数名来控制其调用哪个子程序时才用的，在本章后面才介绍。而最简单常用的是值参数和变量参数，这两种形式参数都可以在子程序内作为变量标识符引用，将在 8.4 节详细讨论。

PASCAL 子程序也可以不带参数，那样的话，形式参数表连同园括号就都不要了。

8.2　过程

8.2.1 过程的说明

按上一节的图 8.1，我们可以知道过程说明形式如下：

```
PROCEDURE 过程名(形式参数表);
```

```
  说明部分
BEGIN
  语句序列
END;
```

其中形式参数表连同园括号也可以没有。可以看到，过程的结构与程序非常类似，好像一个缩小了的程序。但过程与程序还是有着很大的区别：

一、过程以 PROCEDURE 开头，而程序则以 PROGRAM 作标志。

二、过程的参数表示过程与主程序联系的渠道，程序参数表示程序与外界联系的渠道。

三、过程后边跟分号，而程序以句号结尾。

四、过程受主程序的控制，只有当主程序调用它时，它的语句部分才被计算机执行；而主程序则直接受操作系统的控制而运行。

8.2.2 过程的调用

过程调用形式如下：

过程名(实在参数表)

这种形式算作一个简单语句，称作过程语句。

实在参数表也简称为实参表，实参表中各实参之间用逗号隔开。实参的个数，顺序都必须与过程说明中形式参数表中的形参相对应。实参的名字与对应形参的名字无关。实参与相应形参的类型必须相匹配，具体的匹配原则是：值参数赋值相容，变量参数类型同一。实参与形参的关系本书后面还要详细说明。这里先指出一点，如果实参是一个变量，希望调用的过程中给这个变量赋值的话，相应的形参应该定义成变量参数，而不应是值参数。

若过程说明中不带形参表，调用时也不要实在参数表及括号。此时过程语句就只是一个过程名。

[例 1] 编写一个求 n!的过程，调用此过程求表达式

d=k!/(r!(k-r)!)

其中：r，k 为用户输入的数据。

程序如下：

```
PROGRAM examp(input,output);
VAR
  r,k:integer;
  a,b,c,d:real;
  PROCEDURE getfac(n:integer; VAR s:real);  {将 n!送入 s}
    VAR
      i:integer;
  BEGIN
    s:=1;
    FOR i:=1 TO n DO s:=s*i
  END;   {  getfac     }
BEGIN
```

```
    write('Input r & k : ');
    readln(r,k);
    IF r<k
      THEN
        BEGIN
          getfac(k,a);  getfac(r,b); getfac(k-r,c);
          d:=a/(b*c);
          writeln('k!/(r!*(k-r)!)=',d)
        END
      ELSE
        writeln('r>k,input data error!')
END.
```

在主程序的说明部分中说明了一个名为 getfac 的过程。在过程 getfac 的形参表中，说明了一个值形参 n 和一个变量形参 s；在过程 getfac 的说明部分中，说明了一个整型变量 i， 它们的作用域仅限于过程 getfac。过程 getfac 执行一结束，它们将不再代表任何存储单元。

过程 getfac 的功能就是求出 n!赋给 s。主程序中的调用语句 getfac(k,a)作用是将 k!赋给 a，getfac(r,b)的作用是将 r!赋给 b，getfac(k-r,c)的作用是将(k-r)!赋给 c。注意变量形参 s 对应的实参都是变量，而值形参 n 对应的实参却可以是一般的表达式，如 k-r。

程序从主程序的语句部分的第一句开始运行，每执行到一个过程语句时就会将实参代入相应的形参去执行一遍过程说明中的语句部分然后返回主程序。虽然实际执行的流程是这样前后地跳转，但我们阅读主程序时完全可以不必管它如何跳转，也不必管子程序内是如何动作，只要按照上面所说的功能去理解这几个过程语句，主程序就很容易读通。假想系统已经提供了运算符“!”，则

```
getfac(k,a)     可以看作语句  a:=k!
getfac(r,b)     可以看作语句  b:=r!
getfac(k-r,c)   可以看作语句  c:=(k-r)!
```

只有在专门要阅读子程序时才需要去考虑子程序内部细节。这就是我们所说的抽象的方法。一般程序设计时，凡重要一些的子程序，都要在注释中指明该子程序的外观功能，就是为了便于采用这种阅读方式。

最后再补充说明一点：上述关于过程语句中实在参数表的规定，不适用于 read，readln，write，writeln 这四个预定义过程语句。这四个语句有独特的规定，将在文件一章中给出全面介绍。

8.3 函数

8.3.1 函数的说明

按图 8.1，我们可以知道函数说明的形式如下：

```
FUNCTION 函数名(形式参数表):结果类型标识符;
  说明部分
BEGIN
  语句序列
END;
```

其中形式参数表连同园括号也可以没有。

函数和过程许多地方都很类似，这里主要讲一下它们的不同点。

函数与过程的主要不同是它带回一个隐式的返回值。由此而确定了以下几点：

一、函数必须说明其结果类型，即说明其返回值的类型。PASCAL 规定，函数的返回值只能是简单类型或指针类型而不能是构造类型。而且函数说明中的结果类型只能写标识符，不能写新类型描述格式。

二、在函数分程序内的语句部分中至少应该执行到一次这样的语句

函数名:=表达式

其中的“函数名”就是首部中定义的名字。这是个特殊的赋值语句，其作用是规定这个函数的返回值就取这个表达式的值。这个表达式的值必须对该函数的结果类型赋值相容。应注意，赋值相容有时不一定类型相同，例如函数结果类型是实型，而该表达式可以是整型。此时系统会自动转换类型，保证函数的返回值具有规定的函数结果类型。

8.3.2 函数的调用

函数调用的形式称作“函数命名符”，如下：

函数名(实在参数表)

当函数没有形式参数表时，函数命名符的形式变为：

函数名

关于形参，实参的使用与过程完全一样，将在后面详述。

函数命名符在程序中算作一个表达式，该表达式的值就是该函数子程序执行时得到的返回值，该表达式的类型就是该函数的结果类型。

8.3.3 实例

[例 1] 把上一节[例 1]中用过程编写的程序改用函数来编写。

程序如下：

```
PROGRAM examp(input,output);
VAR
  r,k:integer;
  d:real;
  FUNCTION fac(n:integer):real;  {求 n!}
    VAR
      i:integer;
      s:real;
  BEGIN
    s:=1;
```

```
    FOR i:=1 TO n DO s:=s*i;
    fac:=s
  END;   {  fac     }
BEGIN
  write('Input r & k : ');
  readln(r,k);
  IF r<k
    THEN
      BEGIN
        d:=fac(k)/(fac(r)*fac(k-r));
        writeln('k!/(r!*(k-r)!=',d)
      END
    ELSE
      writeln('r>k,input data error!')
END.
```

对比两种编法，请注意两个问题。

一是本程序与上一节相比省掉了 a，b，c 三个变量。 这是因为函数的返回值的存放位置对主程序来说是隐式的。主程序中写一个 fac(k)就可以得到 k!的值，但并不明确这个值在哪个变量里（变量 s 是子程序中的局部变量，回到主程序中 s 就不存在了）。如果需要，在主程序中可以用赋值语句将它送到一个变量里去。但是我们也可以让它直接参加下一步的运算。这个程序中 fac(k)是一个大表达式中的子表达式，故这个返回值应算是表达式运算的中间结果。我们知道，PASCAL 的表达式计算的中间结果都是由编译程序自动为它安排临时存放空间的，用户可以不必考虑。

第二点，函数名不是变量。函数内的语句

函数名:=表达式

是特殊的规定，这在第一、二章关于赋值语句的语法叙述中已经提到过。但应注意，不要因此而误以为函数名可以当变量用。前一个程序中用 s:=s*i 循环后可以在 s 中得到 n!，但后一个程序中若直接用 fac 代替 s，写出语句 fac:=fac*i，就大错特错了。函数符不能这样出现在赋值号右边的表达式里。函数名只有在函数调用时才能出现在表达式中。以后我们会知道，函数的说明中又调用自身是递归调用。

与过程一样，函数同样可以有助于采用抽象的方法来阅读理解程序。虽然实际执行的流程前后地跳转，但我们阅读主程序时完全可以不必管它如何跳转，也不必管子程序内是如何动作。假想系统已经提供了运算符“!”，则

```
fac(k)      可以看作  k!
fac(r)      可以看作  r!
fac(k-r)    可以看作  (k-r)!
```

主程序就很容易看懂。

[例 2] 用对分法求方程

cosx-x=0

在(0,1)范围内的一个近似根（±0.000005)。

对分法的原理是：对于形如

f(x)=0

的方程，若 f(x)是连续函数，且在区间两端 f(x)值的正负号相反，则根据连续实变函数的介值定理，可知区间内该方程必有根。只要满足这些条件，就可以用对分法。具体方法是：

先求出范围中点的 x 值及这点的 f(x)值。假如这点的 f(x)值为 0，显然根已求出，可结束运算。否则这点的 f(x)值必然与范围两端之一的函数值符号相反，根据介值定理，可以将根所在的范围缩小一半。即：如果中点和左端函数值符号相反，则将原中点作为新范围的右端点；如果中点和右端函数值符号相反，则将原中点作为新范围的左端点。对缩小后的范围重复上述操作。如果重复若干次后零点还未找到，但只要范围的宽度已缩得充分小，就可以认为根的近似值已求出。

按以上算法初步编出程序如下：

```
PROGRAM bisect(output);
  CONST
    eps=0.5e-5;
  VAR
    x1,x,x2:real;                    {左，中，右三点 x 值}
  FUNCTION f(t:real):real;           {定义 f(x) }
    BEGIN
      f:=cos(t)-t
    END;
BEGIN
  x1:=0; x2:=1;                      {范围初始化}
  REPEAT
    x:=(x1+x2)*0.5;                  {求中点 x}
    IF (f(x)>0) <> (f(x1)>0)         {中点、左端函数值反号？}
      THEN x2:=x                     {缩小范围}
      ELSE x1:=x
  UNTIL (f(x)=0) OR (x2-x1<eps);     {根已找到或范围已充分小？}
  writeln('Result=',x)               {输出结果}
END .
```

这里主程序中有三处调用 f 函数。运行时每调用一次都要执行一遍 f 的语句部分，因这里有不必要的多余调用，对机时是个浪费，可以进一步改进。

首先每一轮循环中两次 f(x)是重复的，可将 f(x)一次调用所得的计算结果放到一个变量（下面用变量 y）中，每次用到时直接从该变量中读取，不必两次调用 f(x)。

另外，循环中还要计算 f(x1)。虽说每次的 x1 可能变化，这个计算不算完全重复， 但

是我们知道，程序中并不需要知道 f(x1)的准确值，只需要知道其正负号就够了。 由对分法的原理可以知道，整个算法过程中，范围左端函数值的符号都是一样的。所以循环中也可以不再计算 f(x1)，而只要在程序开始将其符号放到一个变量（下面用变量 s）中即可。

改进后的程序如下：

```
PROGRAM bisect(output);
  CONST
    eps=0.5e-5;
  VAR
    x1,x,x2,y:real;              {左，中，右三点 x 值，中点 f(x)值}
    s:boolean;                   {左端函数值>0? }
  FUNCTION f(t:real):real;       {定义 f(x) }
  BEGIN
    f:=cos(t)-t
  END;
BEGIN
  x1:=0; x2:=1;                  {范围初始化}
  s:= f(x1)>0 ;                  {左端符号}
  REPEAT
    x:=(x1+x2)*0.5;              {求中点 x}
    y:=f(x);                     {求中点 f(x)}
    IF (y>0) <> s                {中点、左端函数值反号? }
      THEN x2:=x                 {缩小范围}
      ELSE x1:=x
  UNTIL (y=0) OR (x2-x1<eps);    {根已找到或范围已充分小? }
  writeln('Result=',x)           {输出结果}
END .
```

8.4 值参数和变量参数

PASCAL 子程序的参数中最简单常用的是值参数和变量参数。

8.4.1 值形参和变量形参的语法格式

形式参数表中值参数和变量参数的格式前面已经见到过，这里再给出它们的语法图。前面 8.1 节最后已经说过，形式参数表中定义值参数的形式参数段称作“值参数指明”，定义变量参数的形式参数段称作“变量参数指明”，它们的语法图见图 8.2。

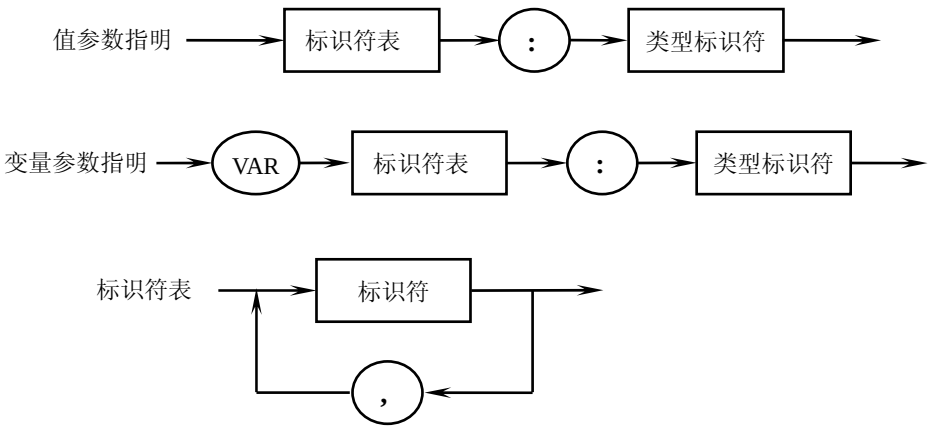


图 8.2　“值参数指明”和“变量参数指明”的语法图

它们的格式和变量说明有些相似，但应该注意，变量说明中冒号后是“类型表记符”，而这里是“类型标识符”，也就是说，这里不能直接写新类型描述格式。

8.4.2 值参数和变量参数的作用

值参数和变量参数的形式参数在子程序内都是作为一个变量标识符来使用的，但它们在子程序与主程序的联系方式上却不相同。概括地说，值参数是子程序自动开辟的临时变量，其初值取实参中代入的值；而变量参数则是一个代表记号，代表实参中代入的那个变量。

值参数（又称赋值参数）相应的实在参数应是一个表达式。子程序在开始执行时，自动地为值参数的形式参数开辟一个临时的存放空间，并将相应实在参数表达式的值赋给这个临时的单元。子程序执行过程中对值形参的访问一律都对这个单元进行。子程序结束时，这个临时单元就撤销了。显然，若子程序执行过程中对值形参赋了值，决不会影响到主程序中的变量的值。

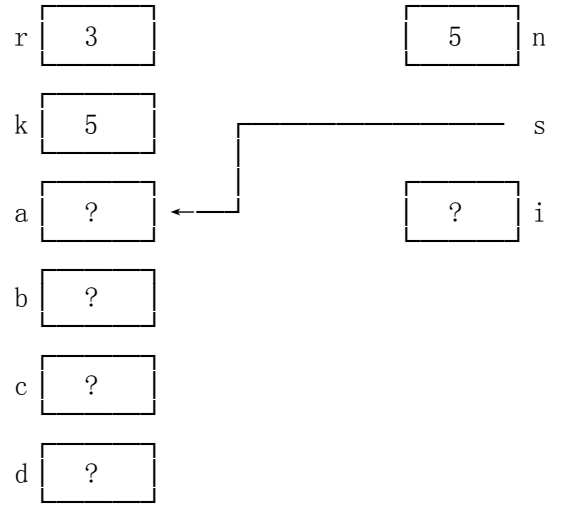


图 8.3　刚进入 getfac 时的情况

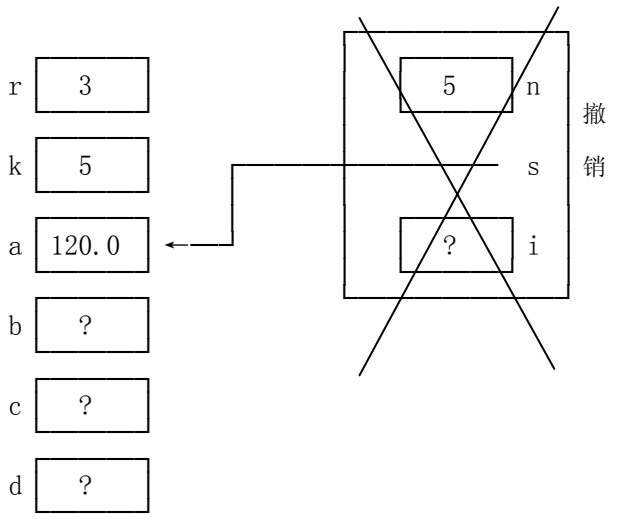


图 8.4　刚结束 getfac 时的情况

变量参数相应的实在参数应是一个变量。子程序并不为变量参数的形式参数开辟存储单元，而是指定该形式参数就代表相应实在参数中给定的变量。子程序执行过程中对变量形参的访问一律都对它代表的那个变量进行（即直接使用实参变量的存储空间）。子程序结束时，虽然形式参数的名字撤销了，但它实际访问的那个变量还存在。如果子程序执行过程中对变量参数赋了值，则这个值已经通过其实在参数而交给了主程序。

显然，变量参数的实在参数不可以是其它形式的一般表达式。

仍以 8.2 节的[例 1]中的程序为例。该程序中主程序开辟了六个变量空间 r，k，a，b，c,d。我们假设执行到语句 readln(r,k)时给 r 输入了 3,给 k 输入了 5。执行到语句 getfac(k,a)时进入过程 getfac。进入时系统为这个过程的值形参 n 开辟一个临时存储空间并将实参的值 5 送入 n 的空间。为变量形参 s 不开辟空间，但规定 s 代表主程序中的变量 a。同时又开辟了局部变量 i。刚进入过程 getfac 时变量空间的情况如图 8.3。

图中?表示该变量尚未赋值。可以看到 getfac 这次执行开始时 n 的初值是 5，s 的初值取决于主程序中的 a，a 的值未定义，故 s 的初值也未定义。getfac 执行中记号 s 代表主程序中的 a,故执行 s:=1 相当于 a:=1,执行 s:=s*i 相当于 a:=a*i。最后得到的 5!=120.0 放在了 a 中。 过程 getfac 执行完后，n，s，i 都撤销了，但放在 a 中的 120.0 还存在。如图 8.4。

我们再研究一下把 s 设成值参数会是什么结果。假设 getfac 的函数首部改为

```
PROCEDURE getfac(n:integer; s:real);
```

则执行语句 getfac(k,a)进入 getfac 时为 s 也开辟一个空间,并把 a 的值传送到 s 空间中。因原来 a 值未定义，故 s 此时的值也无意义。执行 getfac 得到的 120.0 留在 s 的空间中。当三个局部量撤销后我们发现结果丢掉了，并没有放到主程序的变量 a 中。如图 8.5。所以这个题目中将 s 设成值参数是不行的。

那么，s 仍为变参，但将 n 也设成变参行不行呢？下面分析一下。假设 getfac 的函数首部改为

```
PROCEDURE getfac(VAR n:integer; VAR s:real);
```

则执行语句 getfac(k,a)进入 getfac 时不给 n 开辟空间，而是指定 n 代表主程序中的 k，情况如图 8.6。

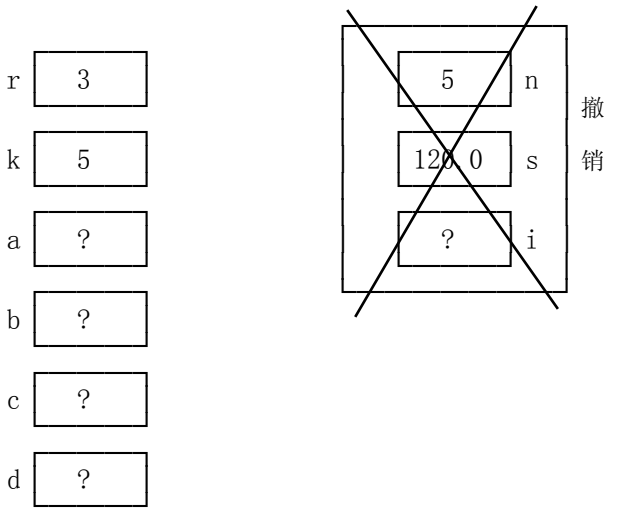
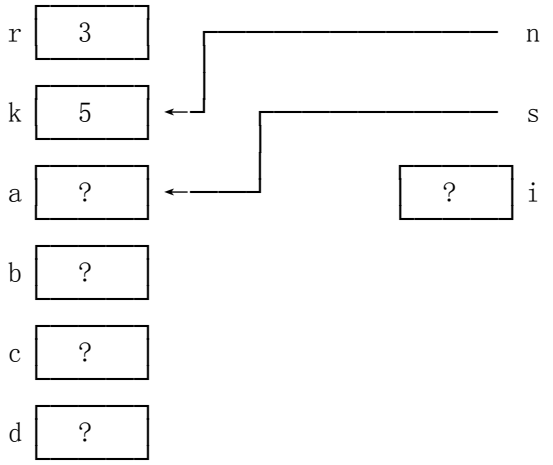


图 8.5　s 改用值参刚结束 getfac 时的情况

从图中可见，n 的初值取决于主程序中的 k，故效果和图 8.3 一样。 而且执行过程中并没有给 n 赋值，所以主程序中的 k 值没发生变化，结果也和图 8.3 一样。故执行语句 getfac(k,a)的效果仍然是正确的。

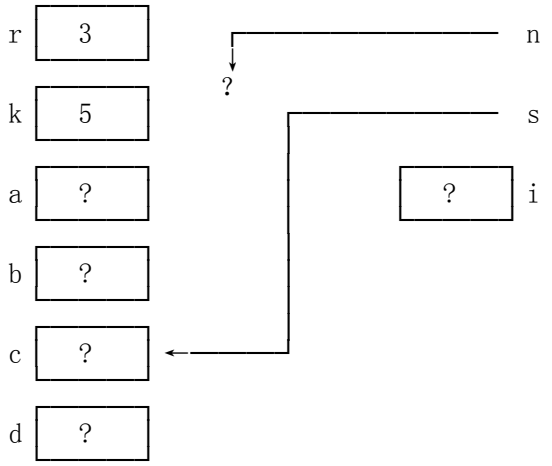


(1)主程序中的变量　　(2)为 getfac 开僻的变量

图 8.6　n 改为变参刚进入 getfac 时的情况

但是，程序中还有两次调用 getfac。第二次是 getfac(r,b)，也没有问题。 第三次调用是 getfac(k-r,c)，就不行了。因为 k-r 不是变量，无法指定 n 代表哪个变量。见图 8.7。此时算是语法错误。

所以，这个程序将 n 设成变参也是不行的。不过，假如程序中没有 getfac(k-r,c)，



(1)主程序中的变量　　(2)为 getfac 开僻的变量

图 8.7　n 改为变参第三次调用 getfac 时的情况

例如另设一个整型变量 x，将 getfac(k-r,c)改为两句

```
x:=k-r; getfac(x,c)
```

那么，将 n 设成变参就没问题了。

另外，我们注意到上述子程序中没有对 n 赋值。假如子程序内对 n 赋了值，例如借用 n 临时存放一些数据，则情况就又不一样了。如果 n 为值参， 这样做不会影响主程序中的变量。而如果 n 是变参，就会影响，如执行 getfac(k,a)后变量 k 的值就被改变。假如下文还要用到 k 的值，这一点显然是不可忽视的。

从上述分析中我们可以看到：值形参对于实参具有保护性，而变量形参则可以改变实参变量的值。如果子程序中对该参数只是临时赋值，不希望影响主程序，就应采用值参数；若为了通过参数将子程序的运算结果返回主程序，就应采用变量参数。

在调用子程序时，如果值参数的实在参数表达式只是一个变量，则形式上和变量参数就很相像，实际作用有时也是一样的。所以程序中有些时候是既可采用值参数也可采用变量参数的。但是一定要注意，如果子程序中对该参数赋了值，则二者作用就不同了。

在既可采用值参数也可采用变量参数的时候，一般的习惯是：若是简单类型，则大多用值参数。因为值参数较安全，可避免副作用。假如因考虑不周在修改时又对参数赋了值，也不会影响到主程序。若是构造类型如数组等，则大多用变量参数。因为构造类型的数据量往往较大，值参数开始时开辟临时空间要占用较大内存，而且向值形参空间传送数据，也要花费较多时间。

最后，我们再指出一个问题：上面说，变量参数的形参在运行时总是代表主程序中代入实参的那个变量，但它实际代表哪个变量，是在刚开始调用时确定的，子程序执行过程中不再重新确定。如下面的程序

```
PROGRAM ......
  VAR
    i:integer;
    a:ARRAY [1..100] OF char;
  PROCEDURE pp(VAR x,y:char);
    BEGIN
      i:=20;
      x:=y;
      ......
    END;
BEGIN
  ......
  i:=10;
  pp(a[i],a[i+1]);
  ......
END .
```

其中 i 是全局变量（见下一节），所以在主程序和子程序中都可以修改它。调用过程 pp 时，形参 x 和 y 分别代表主程序中的 a[i]和 a[i+1]。pp 过程中的语句

```
x:=y
```

之前虽然 i 值已被修改，但这里 x 仍然代表主程序中的 a[10]，y 仍然代表主程序中的 a[11] ，因为刚调用时 a[i]及 a[i+1]是 a[10]及 a[11]。

除了像这样实参是下标变量的情况以外，如果实参是后面第十章将讲到的标识变量，在子程序中修改了指向它的指针的话，也是这种情况。不过，虽有这种规定，但我们编程时最好避免在子程序中修改这种下标或指针的值，因为这样作增加了程序的理解难度。

8.4.3 实参和形参的类型匹配

PASCAL 中规定，值参数的实在参数表达式的值对于其相应的形式参数的类型应该赋值相容。变量参数的实在参数和其相应的形式参数的类型应该同一。

如上面例子中 getfac 的形参 n 是整型，如果出现了这样的过程语句

```
getfac(3.0,a)
```

则是错误的，因为实数 3.0 不能赋给整型变量。但对于值参数， 如果形参是实型而实参是整型则是可以的。

对于变量参数，要求的是类型同一。一般情况下类型同一比赋值相容要求更严。我们在第六章讲类型同一的时候已经讨论过，要满足类型同一的要求，实参的类型就不可以由新类型描述格式来表记。例如

```
PROGRAM test1(input,output);
  TYPE
    t1=0..9999;
    t2=ARRAY [1..10] OF 0..9999;
```

```
  VAR
    a:t2;
  PROCEDURE p1(VAR x:t1);
    ......
  BEGIN
    ......
  END;
BEGIN
  ......
  p1(a[1]);
  ......
END.
```

这个程序中的过程语句 p1(a[1])是错误的。粗看起来：a[1]是 a 数组的成分，a 是 t2 类型，而 t2 类型的成分类型是 0..9999；形参 x 的类型是 t1，而 t1 也等于 0..9999；似乎应该没有问题。但是须注意，程序中有两处 0..9999，a[1]是后一个 0..9999 型，x 是前一个 0..9999 型， 而 PASCAL 标准中不承认两次出现的新类型描述格式是同一类型，会认为类型不匹配。

要纠正这个错误，可将 t2 的定义改为

```
t2=ARRAY [1..10] OF t1 ;
```

就行了。这样 a[1]的类型也由标识符 t1 来表记，就和形参 x 相同了。

上面说的是变量参数。如果形参为普通数组名（除字符串外），即使是值形参，这个问题也一样。因为我们回顾第六章讲的赋值相容的条件，知道对普通数组，只有类型同一才能赋值相容，所以此时不管是值形参还是变量形参，都得满足类型同一才行。

另外，对于变量参数，PASCAL 中还规定，其实在参数不可以是紧缩数组中的成分，或其它紧缩结构类型变量中的成分变量，也不可以是带变体的记录中的标志域。其所以这样规定，是因为变量形参要直接使用实参的存放空间，而这几种成分变量的存储形式比较特殊，难以处理。

8.4.4 预定义过程和预定义函数的参数

预定义的过程和函数，因为程序中不再说明它们，所以我们也见不到它们的形式参数表。但是它们的参数仍然是有所不同的。一般我们可以这样区分：

凡是允许代入表达式的参数都是值参数。如学过的各算术函数的参数都是值参数，这些函数的实在参数可以是各种形式的表达式：

```
sqrt(x+y), sin(1.5/2.4), ......
```

等等。

凡是只允许代入变量，特别是那些执行后该变量有可能被赋以新值的参数都是变量参数。如 read 的参数。又如指针一章将讲到的 new 过程的参数等。

另外，从第六章赋值相容的叙述中我们可以知道，文件类型永远不会满足赋值相容的要求的。所以，若有某过程或函数的参数是文件（如第十一章介绍的 eof(f)函数中的参数

f)，则该参数只可能是变量参数，而不可能是值参数。

8.5 标识符的作用域及变量的生存期

这一节讲两个概念，一是标识符的作用域，二是变量实体的建立和撤销。这两个概念有些地方有联系，所以放在一节中。

8.5.1 标识符的作用域

第二章讲过，标识符必须有了定义才能引用。在程序的文本中，每一个定义都有一个有效作用的范围，只有在这个范围内才允许引用它。这个范围就叫做该标识符的作用域。

在以前，我们暂时认为作用域就是整个程序。现在引入了子程序后要作较全面的规定。

在比较简单的情况下，一个分程序的说明部分里定义的标识符，其作用域就是这个分程序（但要除掉下面补充规定里指出的情况）。例如，程序分程序的说明部分中定义的标识符的作用域是整个程序分程序，过程分程序的说明部分中定义的标识符的作用域是整个过程分程序，函数分程序的说明部分中定义的标识符的作用域是整个函数分程序。

如 8.2 节[例 1]的程序中变量名 i 是在过程 getfac 内的说明部分中定义的， 它只可在 getfac 内引用，而不得在其它部分如主程序的语句部分中引用。而变量 k，r，a，b，c，d 的作用域是整个外层的程序分程序，其中包含内嵌的 getfac 过程分程序的区域。

特殊情况下的补充规定有以下三点：

一、若某分程序的说明部分中定义了一个标识符，而该分程序内嵌套的另一分程序的说明部分中又定义了一个同样拼写的标识符，则系统将它们看作不同的标识符。外层标识符的作用域是外层的分程序中除掉内层同名标识符的作用域后剩下的部分。例如下面的程序中：

[例 1]

```
PROGRAM exam4(input,output);
  VAR
    i,j:integer;
  PROCEDURE p;
    VAR
      x,y:integer;
    PROCEDURE p1(n:integer);
     VAR
        x,y,t:integer;
    BEGIN
      x:=n; t:=x; x:=y; y:=t;
      writeln('x=',x:2,'y=',y:2)
    END;    (*  p1   *)
    PROCEDURE p2(m:integer);
      VAR
        t:integer;
    BEGIN
      x:=x*m; t:=x; x:=y; y:=t;
      writeln('x=',x:2,'y=',y:2)
    END;    (*  p2   *)
  BEGIN
    x:=1;  y:=2;
    p1(2); p2(3);
  END;   (*   p    *)
BEGIN    (*  main   *)
   i:=10;  j:=100;
   p;
END.
```

区域丙　区域乙　区域甲　区域丁

在主程序中定义了变量名 i，j 和过程名 p，这三个标识符的作用域是区域甲。过程 p 中又说明了两个过程 p1 和 p2 和两个局部变量 x，y。这四个标识符中，p1 和 p2 的作用域是区域乙，而 x 和 y 的作用域则是区域乙中扣除区域丙后剩下的部分。这是因为区域丙中定义了名字也叫 x 和 y 的另两个标识符。

阅读程序时一定要注意，区域丙中的 x 和外面的 x 不是同一个变量，仅仅是名字相同。所以在 p2 中给 x 赋值不会改变外面 x 的值。而区域丁中的 x 和外面（区域乙后部的语句部分）的 x 则是同一个，它们要互相影响的。

子程序中引用的标识符，如果就是本子程序中定义的，习惯上称这些标识符是“局部”的，如果是在包括本子程序的更大的区域中定义的，习惯上称之为“全局”的。如上面子

程序p2 中引用的t算是局部变量，而x和y算是全局变量。这里局部和全局是相对的， 比如这里的x和y若相对于外层的过程p，则也算是局部的[①]。

过去我们学 FOR 语句时已经讲过，如果子程序中有 FOR 语句，则它的控制变量只可用该子程序的变量说明部分中定义的局部变量，而不能用全局变量。如 8.2 节和 8.3 节例子程序中的变量 i 就是这样的局部变量。

二、关于过程首部中定义的标识符，其中过程名算是该过程说明所在的分程序中的定义，而形式参数则算是该过程说明内的过程分程序中的定义。如上面的例子中，过程名 p1 的作用域是区域乙，而形式参数 n 的作用域是区域丙。同样，函数首部中定义的标识符，其中函数名算是该函数说明所在的分程序中的定义，而形式参数则算是该函数说明内的函数分程序中的定义。

三、一般情况下（唯一的例外后面指针一章讲到）引用必须在定义之后。

我们再看下面的程序：

```
PROGRAM exam7(input,output);
  VAR
    x:integer;
  PROCEDURE p;
    VAR
      x,y:integer;
    PROCEDURE p1;
    BEGIN
      writeln('this is a book');
    END;
  BEGIN
    x:=1;   y:=2;
    writeln(x:2,y:2);
    p1;
  END;
BEGIN
  read(x);
  IF x=0 THEN p;
  p1;
END.
```

其中过程 p1 是过程 p 内的局部过程，在 p 的语句部分中调用它是可以的。但是主程序的语句部分已在它的作用域之外，所以主程序中调用 p1 是非法的。故这个程序错误。有人为了纠正这个错误，将 p1 的说明移出来，使其成为全局过程，如：

[①] “全局”一词不是标准术语，有些资料上只把最外层主程序中定义的标识符才称作全局的。本书则按相对的意义来使用。

```
PROGRAM exam7(input,output);
  VAR
    x:integer;
  PROCEDURE p;
    VAR
      x,y:integer;
  BEGIN
    x:=1;   y:=2;
    writeln(x:2,y:2);
    p1;
  END;
  PROCEDURE p1;
  BEGIN
    writeln('this is a book');
  END;
BEGIN
  read(x);
  IF x=0 THEN p;
  p1;
END.
```

但是这样仍然错误。错误在于：虽然主程序中调用 p1 合法了，但在 p 内也有调用 p1 的语句，它的位置是在 p1 的过程说明之前，这就违反了上述的第三点。要纠正它，其实只要将 p1 的过程说明放到 p 的过程说明之前就行了。

8.5.2 变量实体的建立和撤销

过去我们说，变量一旦在程序中被说明，系统就为它在内存中开辟存储单元。其实，这种说法只对最外层的全局变量才成立。

子程序内定义的变量，在子程序没有执行时，这些变量的实体并不建立。只有在调用到子程序时，才为子程序内定义的变量建立其实体。在子程序执行结束时，这些实体就都撤销了。

所谓变量实体的“建立”是指以下操作：

对值形式参数：开僻一个存储空间，并将实参的值赋给它；

对变量形式参数：设置一个指示器，指向相应的实参变量；

对其它局部变量：开僻一个存储空间。

所谓“撤销”，对变量形式参数，就是取消该指示器；对值形式参数和其它局部变量，就是释放这些存储空间。

上一节的图 8.3～8.6 已经描述了这个过程。

应注意，这些变量实体只有在建立它的那个子程序结束时才撤销。如果外层程序中已经建立了某些变量的实体，运行中又调用一个子程序，则进入内层子程序时外层的变量并

不撤销，仍然存在。这就是子程序内可以引用全局变量的原因。

有时内层子程序已经超出了外层变量的标识符作用域，当然子程序内就不能直接引用外层的这个变量标识符了，但是在这时外层的变量仍然存在着，一直要保存到子程序执行结束回到外层后再继续发挥作用。所以变量的生存期比变量名作为标识符的作用域所限定的范围可以更大。

比如，假设主程序说明部分中先后定义了两个过程 pr1 和 pr2，pr2 中又定义了局部变量 x，而且 pr2 的语句部分中又调用了过程 pr1，如下：

```
……
PROCEDURE pr1;
  ……
BEGIN
  ……
  ……
END;
PROCEDURE pr2;
  VAR x:real;
BEGIN
  ……
  pr1;
  ……
END;
……
BEGIN  {以下是主程序}
  ……
```

则在程序由 pr2 转到 pr1 后，在 pr1 内 x 虽不再能直接引用，但它并没有撤销，其值也不丢失，待 pr1 结束程序回到 pr2 后继续发挥作用。

又比如，前面的[例 1]中我们说过，过程 p 中标识符 x 和 y 的作用域是区域乙中扣除区域丙后剩下的部分，这是因为区域丙中定义了名字也叫 x 和 y 的另两个标识符。但是实际上当程序进入过程 p1（区域丙）时，在外层 p 中建立的变量 x 和 y 的实体并没有撤销，而是和内层 p1 中新建的名字也叫 x 和 y 的另两个变量实体同时存在，互不干扰。只是在 p1 之内引用的 x 和 y 是内层建立的实体。p1 结束时内层的变量实体撤销了，剩下外层的变量实体继续可用。

上面说到外层的 x 和内层的 x 不算是同一个标识符，不是同一个标识符则它们所建立的变量实体当然也不是同一个。那么同一个标识符建立的变量实体是不是就一定是同一个呢？这里需要注意。我们说，程序中每次调用到子程序时，就为子程序内定义的变量建立其实体。如果两次调用同一个子程序，即使对子程序中的同一个变量标识符，不同次调用时建立的变量实体也不是同一个。

这个问题，简单情况下可以不考虑，因为同一个子程序第二次调用时第一次调用已经

结束了，第一次建立的变量实体已经撤销，互相没有什么牵连。但是复杂情况下这个问题就必须清楚，如下一节将讲到的递归调用，第一次调用一个子程序还未运行结束时就又调用同一个子程序，此时第一次建立的各局部变量的实体并没有撤销就又为它们建立另一批变量的实体，应注意第二次建立的实体与第一次建立的不是同一个。这些，下文的例子中还会进一步讨论。

8.6 递归调用

8.6.1 递归子程序的概念及应用

PASCAL 中允许递归调用。所谓递归调用，指子程序执行过程中直接或者间接调用自己。前者称为直接递归，即在子程序的语句部分中有调用其自身的过程语句或函数命名符。后面一种称为间接递归，它指的是一个子程序中调用到另一个子程序，而第二个子程序又直接或间接调用到第一个子程序。本节的例子都是直接递归。

递归调用的子程序运行时的实际流程分析起来虽然复杂，但用它解决实际问题的程序却常常很简单。所以我们先看几个实例。

[例 1] 编写求 n!的递归函数。

求阶乘的程序前面已经举过了。前面的例子中都是采用循环程序，根据 5.3 节[例 2]公式

$$k!=\begin{cases}1 & ,\text{当 } k=0 \text{ 时}\\ (k-1)!*k & ,\text{当 } k>0 \text{ 时}\end{cases}$$

重复推算，由 0!推出 1!，由 1!推出 2!，等等，最后得出 n!。

现在，采用递归子程序，可以有另外一种设计：直接照搬上面的公式，在求 k!的子程序中调用求(k-1)!的子程序，而求 (k-1)!的子程序与求 k!的子程序是同一个， 只是代入不同的参数而已。如下：

```
FUNCTION fac(k:integer):integer;
  BEGIN
    IF k=0
      THEN fac:=1
      ELSE fac:=fac(k-1)*k
  END;
```

主程序中只要调用 fac(n)就可得到 n!。

[例 2] 用递归调用的方法编写辗转相除法求最大公约数的函数。

前面 5.1 节[例 1]中已经用循环的形式编写过这个算法，所依据的规则是：

若 n=0，则 m 与 n 的最大公约数就是 m；

若 n>0，设 m 被 n 除得余数 r，则 m 和 n 的公约数与 n 和 r 的公约数相同。

现在用递归调用的办法，可以直接照搬上述规则编写。如下：

```
FUNCTION gcd(m,n:integer):integer;
```

```
BEGIN
  IF n=0
    THEN gcd:=m
    ELSE gcd:=gcd(n, m MOD n)
END;
```

由这两个例题，我们可以看出使用递归方法解决问题的几个特点：

一、要解决的问题可以转化为一个新的问题，但新问题的解决方法与原问题相同，只是所处理的对象不同而已。如[例 1]中求(k-1)!与原问题求 k!是同一个算法，[例 2]中求 m 和 n 的公约数与求 n 和 r 的公约数是同一个算法。

二、必须要有一个明确的结束递归的条件。这种程序中，从字面上虽然看不到循环一类的重复操作，但实际运行时仍然是大量的重复操作，这是因为递归子程序嵌套的层次是不确定的。所以，必须保证总能在进入某个层次时不再递归调用，否则无限制地嵌套下去程序就不能正常结束了。如[例 1]中，假设要求 fac(5)，则执行中会调用 fac(5-1)即 fac(4)， 而执行 fac(4)中又会调用 fac(4-1)即 fac(3)，等等，一层层地嵌套进去，总有某一层代入 k 中的实参等于 0，则这一层执行中就不走 fac:=fac(k-1)这个分支了，而只是执行 fac:=1 后就开始返回。这是保证算法“有终”的必要条件。

我们过去接触过的问题中，符合这些特点的有许多，它们都可以改用递归子程序的方法来编写。如过去讨论过的连乘、累加、求最大值等都曾给出过递推公式或递推算法，直接根据这些公式或算法都可以编出递归子程序。请读者自己练习。

实际上，凡是能用循环实现的算法，原则上都能用递归子程序实现。下面再看两个简单的例子。

[例 3] 键盘输入正整数 n，计算机依次输出 1～n 的所有整数。用递归子程序实现。

我们可以编一个“依次输出 1～k 的所有整数”的一般过程 seqout(k)， 这个过程的算法如下：

若 k=0 则空操作，否则：

(1)依次输出 1～k-1 的所有整数

(2)输出 k。

其中的(1)就是递归调用。主程序中调用 seqout(n)即可依次输出 1～n 的所有整数。

程序如下：

```
PROGRAM recpro(input,output);
  VAR
    n:integer;
  PROCEDURE seqout(k:integer);
    BEGIN
      IF k>0 THEN          { 若 k=0 则空操作 }
        BEGIN
          seqout(k-1);     {依次输出 1～k-1 的所有整数}
          write(k:6)       { 输出 k }
```

```
            END
        END;
    BEGIN
      readln(n);
      seqout(n);
      writeln
    END .
```

[例 4] 下面是求数组中前 n 个成分中的最大值及其下标的递归子程序。 因为这里要得到两个结果，用函数不太方便，所以用过程，结果利用变量参数传回来。最大值用变参 max ，其下标用变参 mi。设所处理的数组是如下的类型：

```
TYPE arr=ARRAY [1..M] OF real;
```

其中 M 是上文已定义的一个整数符号常量。显然这里的 n 不应大于 M。

“求数组 a 中前 n 个成分中的最大值 max 及其下标 mi”的算法采用递归的形式描述如下：

若 n=1 则令 max←a[1]，mi←1；否则

(1)求数组 a 中前 n-1 个成分中的最大值 max 及其下标 mi

(2)若 a[n]>max 则令 max←a[n]，mi←n；否则保留(1)所得 max 及 mi。

这个过程的程序如下：

```
PROCEDURE getmax(VAR a:arr; n:integer; VAR max:real; VAR mi:integer);
  BEGIN
    IF n=1
      THEN BEGIN max:=a[1]; mi:=1 END
      ELSE BEGIN
             getmax(a,n-1,max,mi);
             IF a[n]>max THEN BEGIN max:=a[n]; mi:=n END
           END
  END;
```

这里数组参数 a 采用了变参，若采用值参也可以，只是效率低些，这在 8.4 节已讨论过。而参数 max 和 mi 则必须用变参。

这四个例中，两个是递归函数，两个是递归过程。它们形式上虽有不同，但递归的道理是一样的。

上面的例子既可用循环结构，又可用递归调用的形式解决。但递归调用的内部操作复杂，故程序的运行效率比用循环结构的程序低，所以大多数情况下人们并不用递归子程序来解决这种问题。

但是还有一些问题，用递归子程序比较容易编写，而不用递归子程序就很难编写，这时，递归子程序就体现出它的用处了。如下面的河内（Hanoi）塔问题就是这样。 将来我们还可能遇到许多这一类的问题。

[例 5] 河内（Hanoi）塔问题。如图 8.8 所示：有三根细柱 A，B 和 C，A 上套有 m 个从小到大的圆盘，小的在上，大的在下。要求把这 m 个盘移到 C 柱上，可以借助 B 柱， 在移动的过程中每次只许移动一个盘，除三根柱外园盘不得放在其它地方，且在三根柱上必须始终保持大盘在下，小盘在上。编写程序，根据输入的 m，打出需要移动的步骤。

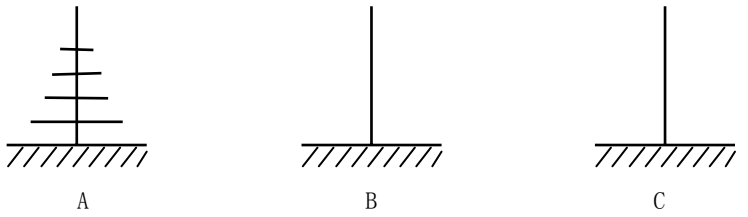


图 8.8　河内塔问题示意

假设有两个盘子，则移动这两个盘子的步骤为：

A --> B
A --> C
B --> C

假如盘子数是任意的 n 个，问题就不那么直观了。 这个问题需要采用数学归纳法的思想来考虑。

首先应考虑 n=1 时的解法。此时问题很简单， 直接将盘子一步就移过去了。

其次再考虑 n>1 时的一般解法。按数学归纳法，考虑 n 个盘子的解法时要假设 n-1 个盘子已经能解。这样，“将 n 个盘子从 x 柱借助 y 柱移到 z 柱”的算法可以这样设计：

(1)将 n-1 个盘子从 x 柱借助 z 柱移到 y 柱；

(2)将 1 个盘子由 x 柱移到 z 柱；

(3)将 n-1 个盘子从 y 柱借助 x 柱移到 z 柱。

其中第(1)步及第(3)步都是递归调用。

我们用过程 hanoi(n,x,y,z)来描述“将 n 个盘子从 x 柱借助 y 柱移到 z 柱”的整个操作，可以编出以下程序：

```
PROGRAM hanoiprg(input,output);
  VAR
    m:integer;
  PROCEDURE move(v,w:char);
    {将 1 个盘子由 v 柱移到 w 柱}
  BEGIN
     writeln(v,'-->',w)
  END;
  PROCEDURE hanoi(n:integer;x,y,z:char);
    {将 n 个盘子从 x 柱借助 y 柱移到 z 柱}
```

```
  BEGIN
    IF n=1
      THEN move(x,z)            {将 1 个盘子由 x 柱移到 z 柱}
      ELSE
        BEGIN
          hanoi(n-1,x,z,y);     {将 n-1 个盘子从 x 柱借助 z 柱移到 y 柱}
          move(x,z);            {将 1 个盘子由 x 柱移到 z 柱}
          hanoi(n-1,y,x,z)      {将 n-1 个盘子从 y 柱借助 x 柱移到 z 柱}
        END
  END;
BEGIN
  write('Input the number of disks:');
  readln(m);
  writeln('Steps:');
  hanoi(m,'A','B','C')
END.
```

运行结果为：

```
Input the number of disks:4
Steps:
A-->B
A-->C
B-->C
A-->B
C-->A
C-->B
A-->B
A-->C
B-->C
B-->A
C-->A
B-->C
A-->B
A-->C
B-->C
```

河内塔问题是一个带有些神秘色彩的古老的游戏，游戏中取 n=64，据说这个游戏作完就到世界末日了。

其实，这是一个典型的适宜采用数学归纳法来研究的问题。不仅上述算法可用数学归纳法来设计，而且我们说，n 个盘子时的解法包括 2^n-1 次移动， 其理由也可以用数学归

纳法来证明：

首先考虑 n=1 时的情况，此时只须 1 步，而 $1=2^1-1$，成立。

其次再考虑 n>1 时的一般情况。按数学归纳法，考虑时要假设 n-1 个盘子时结论成立，即假设 n-1 个盘子需要 $2^{n-1}-1$ 次移动。于是，上述算法中的三大步为：

(1)将 n-1 个盘子从 x 柱借助 z 柱移到 y 柱，需要 $2^{n-1}-1$ 步；

(2)将 1 个盘子由 x 柱移到 z 柱，需要 1 步；

(3)将 n-1 个盘子从 y 柱借助 x 柱移到 z 柱，需要 $2^{n-1}-1$ 步。

总共需要

$$(2^{n-1}-1)+1+(2^{n-1}-1)=2^n-1$$

步，证完。

这个程序虽然不长，但随着 n 的增大，2^{n-1} 的值会急剧增大，程序的运行时间也会很长。比如设 n=64，算一下就会发现 $2^{64}-1\approx1.8447\times10^{19}$，是一个天文数字。 如果人来解决这个问题，每秒移动两步，需要用 2923 亿年。中国历史上最长的朝代周朝有 800 年， 这个时间是它的 3.65 亿倍。一个核桃若放大这么多倍，就和地球差不多大了。其实，太阳的燃烧寿命才 150 亿年。看来，说“到世界末日”似乎不算夸张。

第一章介绍结构化程序设计时，曾把递推的方法（数学归纳法）列为程序设计的基本思想方法之一。程序设计中，具体实现这种思想方法的技术主要有二：一是循环，二是递归子程序。

现实中许多概念本身具有“递归”的性质。如我们已经知道，PASCAL 中表达式的组成部分又可以是表达式，语句的组成部分又可以是语句，构造型数据的成分又可以是构造型数据，等等。所以 PASCAL 的编译程序中处理这一类问题时必须采用递推的思想方法来设计。又如以后如果学了《数据结构》，就会知道树的组成部分又是树，图的组成部分又是图，所以处理这一类数据结构的算法也大量采用递推的思想方法来设计。

数学训练不足的读者可能不习惯这种思想方法，可以通过解决实际问题逐步熟悉起来。

8.6.2 递归子程序的运行

递归子程序运行中的动作原理实际上和普通子程序一样，其要点在前几节都已讲过。只是这里许多容易搞混的问题同时存在，人要在脑子中勾画它的动作过程时就感到复杂。

其实，要弄清它的动作，需要注意两个方面的问题。

一是调用和返回的层次。系统自动保证每次调用执行结束时能回到原调用点。主程序中调用该子程序时即进入该子程序中从头运行，我们称作该子程序的第一层运行。第一层运行中途又调用该子程序时再次进入该子程序中从头运行，我们称作该子程序的第二层运行。一般地，第 k 层运行中调用该子程序即开始第 k+1 层运行。而第 k+1 层运行结束时自动回到第 k 层中的原调用点去继续执行第 k 层中未完部分的程序。 最后第一层运行结束时才回到主程序。各层运行执行的虽然是同一段程序文本，但很像是将这一段程序文本复制了许多份，每层分别执行它的一个“复制件”。

二是局部变量实体的建立与撤销。

上一节中已经说过，程序中每次调用到子程序时，就为子程序内定义的变量建立其实

体。即使对同一个子程序中的同一个变量标识符，不同次调用时建立的变量实体也不是同一个。

按照这个原则，我们知道，递归子程序调用进入了多少层，系统中就为该子程序中的局部变量建立了多少个不同的实体。如果是值形参，或者是子程序的变量说明中定义的局部变量，则不同的实体就是不同的存储空间。此时，各层存放的局部变量占用不同的空间，当然是互不干扰的。

以本节的[例 1]为例，见图 8.9 所示。

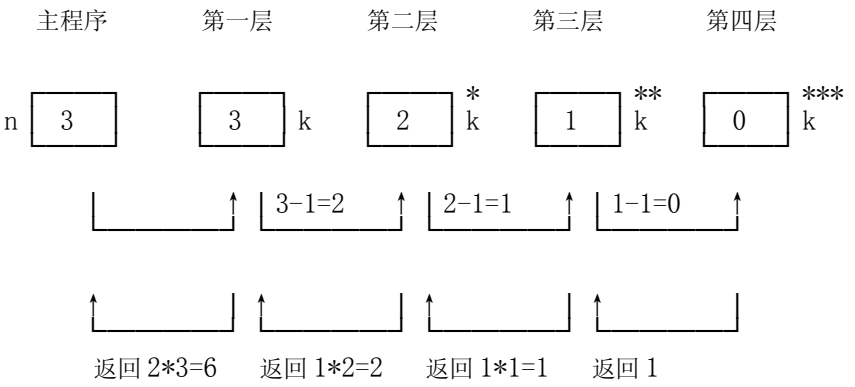


图 8.9 [例 1]运行中建立的变量实体

假设主程序中有变量 n 值为 3，主程序中执行 fac(n)时，进入子程序 fac，此时先为 k 开僻一个空间（即图中标 k 的方框中最左边的一个，不带*号）并将 n 的值 3 代入其中。接着开始该子程序的第一层运行。

第一层运行中执行到 fac(k-1)时要对 fac 子程序进行第二层调用， 进入第二层时又为形参 k 开僻一个空间（即图中标一个*号的框），将实参（即第一层 k-1 的值 2）代入其中。接着开始第二层运行。

第二运行中执行到 fac(k-1)时又对 fac 第三层调用，又为形参 k 开僻一个空间（即图中标两个*号的框），将实参的值 1 代入其中，开始第三层运行。

同样道理，第三层运行中又进行第四层调用，此时实参为 0，又为形参 k 开僻一个空间（即图中标三个*号的框）。第四层中 k=0 成立，故按 IF 语句执行 fac:=1 后第四层运行就结束。结束时标三个*号的变量实体撤销，返回值为 1 回到第三层。

回到第三层时 fac(k-1)的结果（即第四层返回的值）为 1，而第三层的 k 为 1， 故第三层中 fac(k-1)*k=1*1=1。所以第三层结束时返回值为 1（同时撤销标两个*号的变量实体）。

以此类推，每退一层撤销一个 k 的实体，最后回到主程序中时返回值是 6。

从图中可见，这个例子执行中可以有多个 k 的空间并存，分别为不同层的运行服务。

上面例中是值形参，若是子程序的变量说明中定义的局部变量，情况也一样，也是每层开僻一个空间。

但若是变量形参，情况就有些不同。变量形参的变量实体并不开僻空间，只是设立一个指示器。建立多个实体也就是设立多个指示器，至于这些指示器所指的空间相同与否则视具体情况而定。程序运行中访问变量实体时并不是对指示器进行操作，而是对它们所指的变量空间操作。

以本节的[例 4]为例，见图 8.10 所示。图中假设主程序中有一个数组名为 data， 有一实型变量名为 most，有一整型变量名为 loc，假设主程序中用语句 getmax(data,4,most,loc)来试图求出 data 数组前四个成分中的最大值送到 most，其下标送到 loc 中。

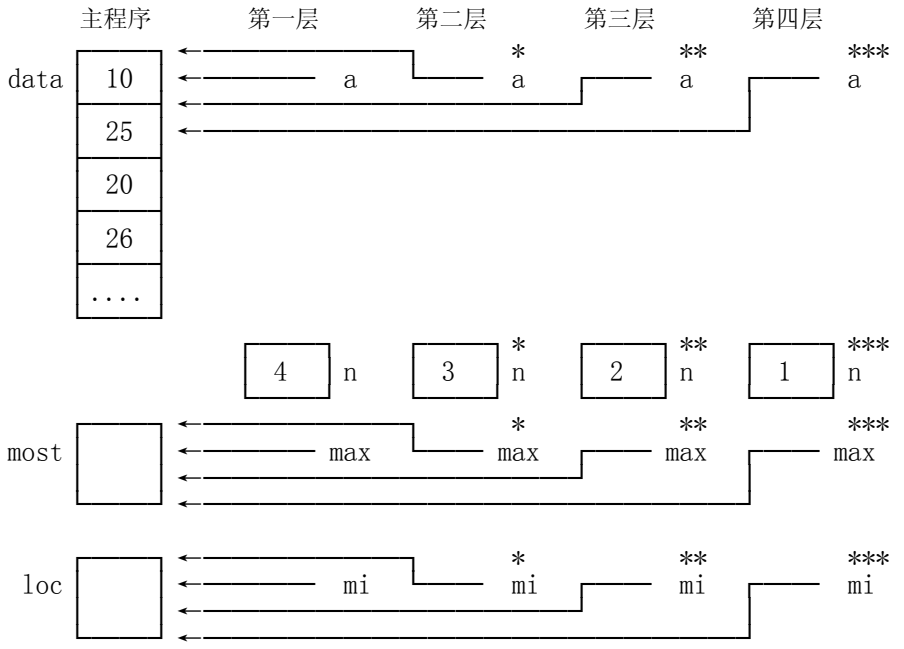


图 8.10 [例 4]运行中建立的变量实体

限于篇幅，具体动作不再详述，只说明几点。

子程序每层进入时建立的四个变量实体 a，n，max，mi 中，只有 n 开僻了空间，另三个都是只设了指示器。

具体这个例子中，四层的 a 代表的都是主程序中的 data 数组， 这是由这个例子中实际的参数代入方式决定的。四个 max，四个 mi 的道理也是如此。

再分析下面例子的情况：

[例 6] 将前面[例 2]辗转相除法求最大公约数的函数改用变量参数实现。

这个子程序内没有对形参赋值，所以原则上是可以改用变量参数的，只是有两个问题：

一是改后主程序调用时实参只能用变量而不能用常数或一般表达式，这在主程序中适当处理即可。

二是原来递归调用时实参用了表达式 m MOD n，现在也须变动，为此增设局部变量 r。

改后程序如下：

```
FUNCTION gcd(VAR m,n:integer):integer;
  VAR
    r:integer;
  BEGIN
    IF n=0
      THEN gcd:=m
      ELSE BEGIN r:=m MOD n; gcd:=gcd(n,r) END
  END;
```

现在我们看改用变量参数后的运行情况。假设主程序中有变量 a 值为 21，b 值为 12，主程序中用 c:=gcd(a,b)调用上述函数，则这个程序运行中变量实体的建立情况如图 8.11 示意。

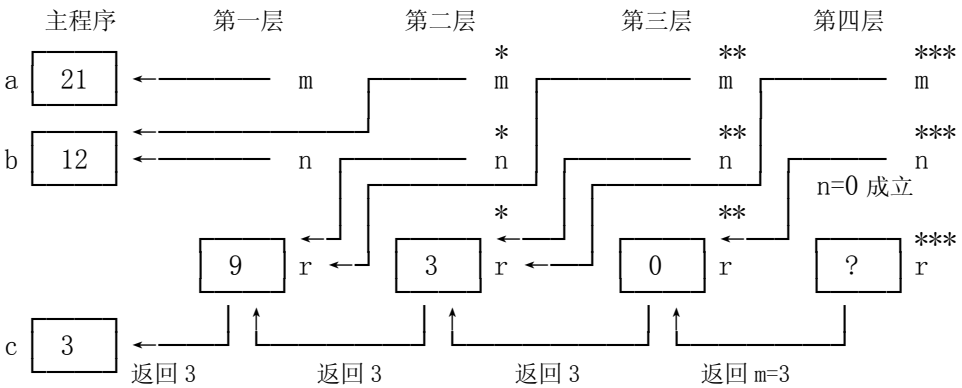


图 8.11 [例 6]运行中建立的变量实体

这里每层建立的三个变量实体 m，n，r 中，只给局部变量 r 开僻了空间，而 m 和 n 都只设立了指示器。这个程序中指示器所指比图 8.10 要复杂些。如第四层的 m 代表的是第三层的 n，而第三层的 n 代表的又是第二层的 r，所以第四层的 m 也就代表第二层的 r。

上面三个都是简单的例子，我们对它们详细探讨，是为了说明一般的原理。再大一点的例子如果也这样画出图来分析，就会发现相当繁琐，所以真正编程序时没有必要都来这样做。其实，运行中内部的这些繁琐的动作，是为了使它宏观上具有明确的功能。所以，了解了这些原理后，实际编程序时只需要对算法进行宏观上的设计就行了。

同时我们也可以知道，由于递归子程序运行时内部的控制动作比较繁琐，所以效率通常比不用递归子程序的算法要低一些。另外，除了由于控制动作繁琐而降低效率以外，有时候算法本身还会带来多余重复的操作，这样效率就更低，如下文的[例 7]。

这三个例子中子程序内只有一次递归调用，所以每一层除最下面一层外，都是执行一半时进入下一层，从下一层返回后再执行完剩下的一半。所以总共深入进几层，程序实际上就执行了几遍。如果子程序内不止一次递归调用，如前面的[例 5]，以及下面的[例 7]，

则执行的遍数就会远大于递归的深度。

[例 7] 编写求 fibonacci 序列第 k 项的递归函数。

fibonacci 序列在 5.4 节的[例 6]已经介绍过，设其第 k 项是 f_k，则递推公式是：

$$f_k=\begin{cases} f_{k-1}+f_{k-2} & ,当\ k>2\ 时 \\ 0 & ,当\ k=1\ 时 \\ 1 & ,当\ k=2\ 时 \end{cases}$$

按递归的方法设计，程序如下：

```
FUNCTION f(k:integer):integer;
BEGIN
  IF k=1 THEN f:=0
         ELSE IF k=2 THEN f:=1
                     ELSE f:=f(k-1)+f(k-2)
END;
```

这个程序中，除 k=1 和 k=2 时外，每层都要递归调用两次。如求 f(5)时要调用 f(4)和 f(3)，而求 f(4)时要调用 f(3)和 f(2)，求 f(3)时要调用 f(2)和 f(1)，等等。这样，求 f(5)时总共要执行 9 遍这段程序。如图 8.12 示意。

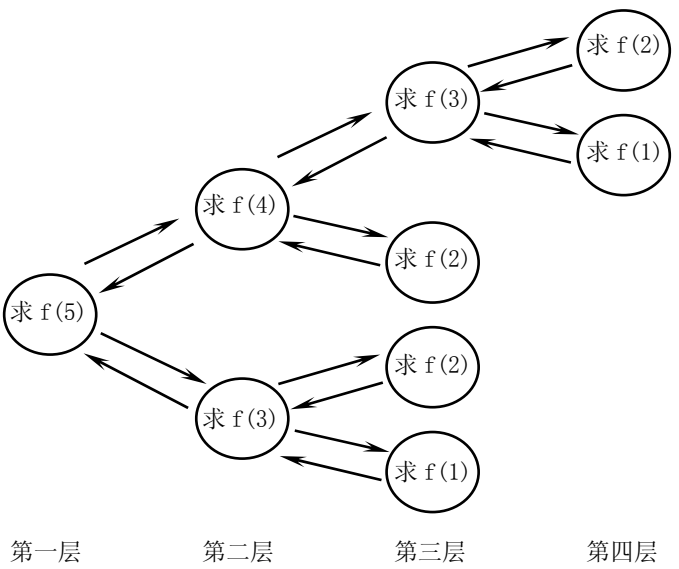


图 8.12 [例 7]程序的执行次序

从图中可以看到，运行过程中有许多遍操作是重复的。因此这个程序比 5.4 节的循环算法效率低。这才只求出了一个 f(5)，若下文又要调用这个子程序求 f(6)，而求 f(6)时第二层又要调用 f(5)，还须把上图的操作再全部重作，效率就更低。

[例 8] 将输入的非负整数转换成十六进制形式(用 A，B，C，D，E，F 分别表示 10，11，12，13，14，15 的十六进制位)。如十进制数 469，转换为十六进制数应为 1D5。 设计两种方案，一种不用递归子程序，一种用递归子程序。

用常规的“除以 16 取余”的算法可以将十进制数转换为十六进制数。但是这种算法是先求出最低位，再依次求出较高位，而输出时却需要先输出最高位。所以，不能采用产生一位输出一位的方法，必须设一个数组将各位暂存起来。考虑常见系统中整数的可取值的范围，估计产生的十六进制数不超过四位，故可取一个长度为 4 的字符数组。程序是：

```
PROGRAM dectohex(input,output);
```

```
  VAR
    a:PACKED ARRAY[1..4] OF char;
    n,m,k,i:integer;
  FUNCTION hex(x:integer):char;        {转换一位}
  BEGIN
    IF x<10
      THEN hex:=chr(ord('0')+x)
      ELSE hex:=chr(ord('A')+x-10)
  END;
BEGIN
  readln(n);
  k:=0;
  REPEAT                               {转换出各位存入数组 a}
    k:=k+1;
    m:=n MOD 16;
    n:=n DIV 16;
    a[k]:=hex(m)
  UNTIL n=0;                           {k 中是位数}
  FOR i:=k DOWNTO 1 DO write(a[i]);    {输出}
  writeln
END.
```

运行情况如下：

```
469
1D5
```

下面我们用递归的方法对这个问题重新来描述一下。“将 n 以十六进制输出”的算法可描述如下：

(1) m ← n MOD 16 ；

(2) n ← n DIV 16 ；

(3) 若 n≠0 则将 n 以十六进制输出；

(4) 将 m 转换为一位十六进制字符输出。

其中第(3)步的内嵌语句是递归调用。这个方案的程序如下：

```
PROGRAM dectohex(input,output);
  VAR
    n:integer;
  FUNCTION hex(x:integer):char;        {转换一位}
  BEGIN
    IF x<10
      THEN hex:=chr(ord('0')+x)
```

```
      ELSE hex:=chr(ord('A')+x-10)
  END;
  PROCEDURE dtoh(n:integer);              {递归转换输出子程序}
    VAR
      m:integer;
  BEGIN
    m:=n MOD 16;
    n:=n DIV 16;
    IF n<>0 THEN dtoh(n);
    write(hex(m))
  END;
BEGIN              {主程序}
  readln(n);
  dtoh(n);
  writeln
END .
```

我们可以看到：用递归的方法可以不用定义数组。但不要以为这就可以节省存储空间。因为调用每进一层就要为形参 n 和局部变量 m 开僻一组空间，大多数情况下所需空间都比上一方案要大。只是上一方案所开僻的空间是固定的，而后一方案递归的层次可多可少，如果位数很少，则所需空间也较少。

8.7　子程序的超前引用

前面我们已经看到，为了满足标识符先定义后引用的规则，如果一个分程序中并列地定义了两个子程序，而其中一个子程序内调用了另一个，则被调用的那个子程序的说明应该放在调用者的前面。

我们又讲到了递归调用，一个子程序可以调用它自身。从上一节的例中可以看到，这种情况与“先定义后引用”的原则不抵触。因为过程标识符或函数标识符是在首部中定义的。而引用该标识符的过程语句或函数命名符则是写在后面的语句部分中。

不过，上一节的例子都是直接递归。间接递归时问题就出现了。如果甲子程序调用乙子程序，乙子程序又调用甲子程序，则不管哪个放在前面都不行。

PASCAL 为避免这个矛盾，规定了一种超前引用子程序的方法，如下：

允许将一个过程分作两条过程说明来编写。这两条都是不完整的过程说明。前一条只有过程首部而没有过程分程序，将过程分程序处代之以指示字 FORWARD。 后一条没有过程首部而只有过程分程序，将过程首部处代之以一个“过程标识”。这样，只要将前一条不完整的说明放到要调用它的子程序的说明之前，将后一条不完整的说明放到它要调用的子程序的说明之后，上述矛盾就解决了。

指示字 FORWARD 的英文意思是“向前”，这里表示“本过程的过程分程序将在下文描

述”的意思。

“过程标识”的格式是 PROCEDURE 后跟过程名。 这个格式看起来有点像过程首部（只是少了形式参数表），但它和过程首部的意义是不同的，这里的过程名不是定义，而是引用，其作用是表示这里的过程分程序与前面相应的过程首部的联系。

对函数也有同样的规定，即：

允许将一个函数分作两条不完整的函数说明来编写。前一条只有函数首部而没有函数分程序，将函数分程序处代之以指示字 FORWARD。后一条没有函数首部而只有函数分程序，将函数首部处代之以一个“函数标识”。

指示字 FORWARD 在这里表示“本函数的函数分程序将在下文描述”的意思。

“函数标识”的格式是 FUNCTION 后跟函数名。其作用是表示这里的函数分程序与前面相应的函数首部的联系。

如下面的例子：

```
PROGRAM exam14(input,output);
  ......
  PROCEDURE pb(x:integer);              {过程 pb 的首部}
    FORWARD;                            {指示字}
  FUNCTION fc(x:integer):integer;       {函数 fc 的首部}
    FORWARD;                            {指示字}
  FUNCTION fa(m,n:integer):integer;
    ......
    BEGIN
      ......
      pb(100);                          {fa 中调用 pb}
      w:=fc(15)+1;                      {fa 中调用 fc}
      ......
    END;
  PROCEDURE pb;                         {pb 的过程标识}
    ......                              {过程 pb 的分程序}
    BEGIN
      ......
      s:=fa(x,y);                       {pb 中调用 fa}
      ......
    END;
  FUNCTION fc;                          {fc 的函数标识}
    ......                              {函数 fc 的分程序}
    BEGIN
      ......
      y:=fa(p,q);                       {fc 中调用 fa}
```

```
      ......
    END;
BEGIN
  ......
END.
```

这里，fa 中要调用 pb 和 fc，因 pb 和 fc 的首部都在前面，故没有问题。而 pb 和 fc 中又要调用 fa，但 pb 和 fc 的语句部分又都在 fa 的说明之后，所以也没有问题。

8.8 子程序名作为参数

这一节介绍过程参数和函数参数。前面已经提到过，过程参数和函数参数是在该子程序中又调用其它子程序时，需要通过实在参数中代入的过程名和函数名来控制其调用哪个子程序时才用的。先看一个实例。

[例 1] 用对分法求下列四个方程在指定范围内的近似根（±0.000005）：

cosx-x=0, (0,1)

sinx-0.8x=0, (0.5,3)

$lnx-e^{x}+5=0$, (1,5)

cosx=0, (0,3)

对分法在 8.3 节[例 2]中已经介绍过。现在一个程序要解四个方程，可以将对分法编成一个子程序，在主程序中调用四次。四个方程指定的求根范围不同，可以将范围的上下界作为对分法子程序的参数。这些都不难解决。现在的问题是：8.3 节[例 2]算法中调用了函数 f(x)，而现在四个方程需要调用不同的函数。这就需要将函数名也作为对分法子程序的参数。

下面先列出这个问题的程序清单，其中将对分法编成了一个函数 bisect，其返回值是根，三个参数中，f 是函数参数，x1，x2 是值参数。

```
PROGRAM get4root(output);
  FUNCTION bisect(FUNCTION f(t:real):real;x1,x2:real):real;
    {对分法子程序，返回近似根}
    CONST
      eps=0.5e-5;
    VAR
      x,y:real;
      s:boolean;
  BEGIN
    s:= f(x1)>0 ;                    {左端符号}
    REPEAT
      x:=(x1+x2)*0.5;                {求中点 x}
      y:=f(x);                       {求中点 f(x)}
```

```
      IF (y>0) <> s                    {中点、左端函数值反号？}
        THEN x2:=x                     {缩小范围}
        ELSE x1:=x
    UNTIL (y=0) OR (x2-x1<eps);        {根已找到或范围已充分小？}
    bisect:=x                          {返回根}
  END;
  FUNCTION f1(t:real):real;    {定义 f1(x) }
    BEGIN
      f1:=cos(t)-t
    END;
  FUNCTION f2(t:real):real;    {定义 f2(x) }
    BEGIN
      f2:=sin(t)-t*0.8
    END;
  FUNCTION f3(t:real):real;    {定义 f3(x) }
    BEGIN
      f3:=ln(t)-exp(t)+5
    END;
BEGIN
  writeln('Root #1 : ',bisect(f1,0,1));    {输出方程 1 的根}
  writeln('Root #2 : ',bisect(f2,0.5,3)); {输出方程 2 的根}
  writeln('Root #3 : ',bisect(f3,1,5));    {输出方程 3 的根}
  writeln('Root #4 : ',bisect(cos,0,3))    {输出方程 4 的根}
END .
```

运行结果是：

```
Root #1 :  7.3908615112E-01
Root #2 :  1.1310987473E+00
Root #3 :  1.7115211487E+00
Root #4 :  1.5707960129E+00
```

我们看一下 bisect 的函数首部：

```
FUNCTION bisect(FUNCTION f(t:real):real;x1,x2:real):real;
```

注意它的形式参数表，其中以分号分隔有两个形式参数段。第二个形式参数段是“值参数指明”，其中定义了两个值形参 x1 和 x2，这在过去已经学过了。现在要讲的是第一个形式参数段，这种形式参数段称作“函数参数指明”，它定义了一个函数形式参数 f。

一般规定，“函数参数指明”的书写格式与普通的函数首部相同。如这里的

```
FUNCTION f(t:real):real
```

其中的函数名标识符 f 就是所定义的函数形式参数， 其余部分则规定了该函数应有的结果类型和形式参数表的构成。

在调用子程序 bisect 时与函数形式参数相对应的实参应是一个已有定义的函数名，该函数的结果类型应与形参的“函数参数指明”中规定的结果类型相同，该函数的形式参数表应与形参的“函数参数指明”中规定的形式参数表构成一致。

具体这个例子中,四次调用 bisect 时与函数形参对应的实参分别是 f1,f2,f3 和 cos 。其中 f1，f2 和 f3 是本程序中定义的，而 cos 是预定义的函数。如 f1 的结果类型是 real，而上述 f 的结果类型也是 real，满足“结果类型相同”；f1 的形式参数表是(t:real)，而 f 的形式参数表也是(t:real)，可算是“构成一致”。

至于什么是“形式参数表构成一致”，像上例那样完全相同，当然可以算作“一致”。但如果书写有些差别还算不算“一致”呢？PASCAL 的标准中有更详细的规定，这里指出其中主要的两点：

其一，形参所用的标识符是否相同无关紧要。例如，若将上面 bisect 的函数首部改为

```
FUNCTION bisect(FUNCTION f(u:real):real;x1,x2:real):real;
```

即将其中的 t 换成 u，程序仍正确。也就是说，此时 f 的形式参数表仍然和 f1，f2 等的形式参数表算是构成一致的。

其二，其中相对应的类型标识符只要求所表记的类型相同即可，不限制具体写法。例如，假设前边有

```
TYPE
  float=real;
```

则将 f1 的形式参数表(t:real)改写成(t:float)，仍然算是“一致”。

除这两点以外，其它因素都必须对应相同。例如，若将上面 bisect 的函数首部中 t 后面的 real 换成 integer，就不算一致了。或者，在 t 之前加一个 VAR 使它成为变参， 也是错的。如果要换为变参，则 f1，f2 等的形式参数表中与 t 相应的参数也必须同时换为变参才算一致。还有，如果 f 的参数个数和 f1 的参数个数不一样，当然也不行。

至于 cos 函数,虽是预定义的,但我们知道它的结果类型和参数类型都是 real 型， 而且参数只有一个，是值参。如果按照这种理解，可以说 cos 函数和上述的 f 也是一致的。

不过，对于像 cos 这样的预定义函数，程序中毕竟没有显式地给出它的形式参数表，而 PASCAL 的标准中又没有对“隐式”的形式参数表作出规定。若严格按照标准，PASCAL 系统并不一定要承认 cos 和 f 的“形式参数表构成一致”。所以，只是有些系统允许用预定义函数作为函数参数的实参，上面例子的程序并非严格满足标准。

如要严格满足标准，可将上例中的

```
writeln('Root #4 : ',bisect(cos,0,3))    {输出方程 4 的根}
```

改为

```
writeln('Root #4 : ',bisect(f4,0,3))    {输出方程 4 的根}
```

并在前面增加 f4 的定义如

```
FUNCTION f4(t:real):real;    {定义 f4(x) }
  BEGIN
    f4:=cos(t)
```

```
    END;
```

即可。

除了上面介绍的函数参数外，PASCAL 中还有一种过程参数。定义过程形式参数的形式参数段称作“过程参数指明”。

“过程参数指明”的书写格式与普通的过程首部相同。其中的过程名标识符就是所定义的过程形式参数，其余部分则规定了该过程应有的形式参数表的构成。

在调用子程序时与过程形式参数相对应的实参应是一个已有定义的过程名，该过程的形式参数表应与形参的“过程参数指明”中规定的形式参数表构成一致。

[例 2] 下面是一个使用过程参数的程序例子，请读者自己分析它的功能。

```
PROGRAM test(input,output);
  VAR
    a,b,c:integer;
    ch1,ch2:char;
  PROCEDURE line(PROCEDURE p; n:integer);
    VAR i:integer;
  BEGIN
    FOR i:=1 to n DO
      BEGIN p; write('------') END;
    p;
    writeln
  END;
  PROCEDURE p1;
  BEGIN
    write(ch1,ch1);
    ch1:=succ(ch1);
    IF ch1>'Z' THEN ch1:='A'
  END;
  PROCEDURE p2;
  BEGIN
    write(ch2,ch2);
    ch2:=succ(ch2);
    IF ch2>'z' THEN ch2:='a'
  END;
BEGIN
  ch1:='A'; ch2:='a';
  readln(a,b,c);
  line(p1,5);
  line(p1,3);
```

```
  writeln(a:4,b:4,c:4);
  line(p2,3);
  line(p2,5)
END .
```

本程序运行时，如果键盘输入

```
12 34 56
```

计算机就会显示：

```
AA------BB------CC------DD------EE------FF
GG------HH------II------JJ
  12  34  56
aa------bb------cc------dd
ee------ff------gg------hh------ii------jj
```

由于过程参数及函数参数的编译处理比较复杂，所以某些较简单的非标准 PASCAL 系统中没有提供这些功能。也有的系统虽然提供过程参数及函数参数的功能，但有若干限制。比如有些系统规定过程参数及函数参数所带的参数只能是值参数。

8.9 可调节数组参数介绍

PASCAL 标准的基本部分没有可调节数组的规定，可调节数组是作为标准的扩充部分而规定的。目前常见的 PASCAL 系统大都没有实现这一扩充，故这里只作简要的介绍。

按前面所讲的基本规定，数组作为参数时，实参和形参必须类型同一。这样一来，对数组进行的操作，只要数组的下标范围不同，就无法引用同一个子程序来完成。有了可调节数组参数，就允许不同下标范围的数组作为实参代入到同一个子程序的参数表中。

如下面是一个求任意下标范围的实数数组的最大成分及其下标的过程：

```
PROCEDURE getmax(VAR a:ARRAY[s1..s2:integer] OF real;
                 VAR max:real; VAR mi:integer);
  VAR i:integer;
BEGIN
  max:=a[s1]; mi:=s1;
  FOR i:=s1+1 TO s2 DO
    IF a[i]>max THEN
      BEGIN max:=a[i]; mi:=i END
END;
```

假设有一个数组变量 x 的类型为

```
ARRAY [1..10] OF real
```

则用语句

```
getmax(x,m,n)
```

来调用上述过程就可以将 x[1]～x[10]中最大成分的值送入 m 中，将其下标送入 n 中。

我们看上面过程首部中第一个形式参数段里冒号后面的

```
ARRAY[s1..s2:integer] OF real
```

如果将这些换成一个类型标识符，那就和前面学过的格式一样了。这里的标识符 s1 和 s2 分别代表实参数组下标类型的下界和上界，这里的 integer 规定 s1 和 s2 必须属于整型。s1 和 s2 的值随调用时代入的数组的下标类型不同而不同，如上面实参代成 x 时它们就分别是 1 和 10，若代入别的数组则可能又是别的值，所以它们不是常量。但它们也不算变量，在子程序内不允许给它们赋值。所以我们给它们另一个名称，叫做界限标识符。

上例中的第一个形式参数段开头有 VAR，故 a 属于“变量可调节数组参数”。PASCAL 中还可以有“值可调节数组参数”，其格式中不要 VAR，其余相同。 二者功能的差别与前面讲过的变量参数和值参数的差别一样。

另外，在可调节数组参数的 ARRAY 前加上 PACKED，则是紧缩的可调节数组参数。 如果形参是紧缩的，则实参也必须是紧缩的；如果形参是非紧缩的，则实参也必须是非紧缩的。

下面是一个求任意阶矩阵中所有元素的平均值的函数：

```
FUNCTION xs(VAR a:ARRAY[m1..m2:integer;n1..n2:integer] OF real):real;
  VAR
    i,j:integer;
    s:real;
BEGIN
  s:=0;
  FOR i:=m1 TO m2 DO
    FOR j:=n1 TO n2 DO s:=s+a[i,j];
  xs:=s/((m2-m1+1)*(n2-n1+1))
END;
```

与上一例子不同之处是这里的可调节数组是二维的。应注意与普通二维数组类型的描述格式不同，普通二维数组类型的描述格式中两个下标类型之间用逗号分隔，而这里两个“下标类型指明”之间是用分号分隔，不是逗号。

这里的

```
ARRAY [m1..m2:integer; n1..n2:integer] OF real
```

是缩写形式，也可以写成完整形式

```
ARRAY [m1..m2:integer] OF ARRAY [n1..n2:integer] OF real
```

类似的，还可以有更高维的多维可调节数组参数。

标准 PASCAL 规定，紧缩的可调节数组只能是一维的形式，而不能是多维的。不过，允许在外层非紧缩的形式中 OF 之后内层是一维紧缩的形式。

8.10 函数和过程应用举例

为了下面的例子，我们先介绍随机函数。所谓“随机”数，本来的意思是指完全无规

则的数，每次得到的值事前不可预测。但计算机算法中的随机函数，所得通常并不是真的随机数，而只是其统计特征尽可能接近真的随机数。主要的特征有：概率的均匀性，即范围内每个值被取到的机会一样大；相邻两次取值的差可正可负，可大可小，其差取各种值的机会分布与真的随机数接近；等等。但它既然有确定的算法，下一次将得到的值理论上说是可以预测的，故不算真的随机数，有时候将它们叫做“伪随机数”。

随机函数的设计方法有多种，下面的函数假如不溢出的话，可以得到 0～65535 范围的伪随机整数。

```
VAR seed:integer;
FUNCTION random:integer;
  BEGIN
    seed:=(25173*seed+13849) MOD 65536;
    random:=seed
  END;
```

这里 seed 是一个全局变量，每调用一次 random 函数 seed 的值就发生一次变化，才能使得每次 random 的结果不相同。

但是，上述运算在常见的实际 PASCAL 系统中已经溢出了。不过，常见的实际 PASCAL 系统这种溢出并不报告错误，而且常见的系统以 16 位二进制补码表示整数，溢出处理时实际上隐含了一个取模运算。所以将上述程序中取模运算省去，在常见系统中就可以正常运行。只是范围不再是 0～65535，而变成了-32768～32767。下面的例子就是采用的这种办法。

因为这还不是真随机数，为使每次启动程序时所得不致总是相同，可使 seed 具有不同的初值。在某些操作系统下，可以读取当时机器上所带的实时时钟的一部分作为 seed 的初值。但这种方法与特殊的系统有关，本书不作讨论。下面的例子中由键盘读入 seed 的初值。

有了这个函数，就可以换算出其它范围的随机数。例如求 0～999 范围内的随机整数，有以下两个办法可用：

```
random MOD 1000
trunc((random/65536.0+0.5)*1000)
```

其中第一个办法效率较高，但均匀性比第二个略差一些。

[例 1] 随机产生一些算术题，请操作者给出答案，由机器判断正确与否，若不正确则给出正确答案的提示。算术题的范围是：

不超过三位正整数的加法；

不超过三位正整数的减法（被减数不小于减数）；

不超过二位正整数的乘法。

下面我们给出完整的程序：

```
PROGRAM examin(input,output);
  VAR  x,y,z,t,seed:integer;
  FUNCTION random:integer;       {-32768～32767 的随机整数}
```

```
    {本算法适用于以 16 位补码表示整数且溢出不报错的系统}
  BEGIN
    seed:=25173*seed+13849;
    random:=seed
  END;
  FUNCTION rnd(n:integer):integer;      {0～n-1 的随机整数}
  BEGIN
    rnd:=trunc((random/65536.0+0.5)*n)
  END;
BEGIN
  write('Input a random number:');
  readln(seed);                         {seed 初始化}
  REPEAT
    CASE rnd(3) OF                {随机选定三种运算之一}
      0: BEGIN x:=rnd(1000);y:=rnd(1000);       {加法}
           z:=x+y; write(x,'+',y,'=')
         END;
      1: BEGIN x:=rnd(1000);y:=rnd(1000);       {减法}
           IF x<y THEN BEGIN t:=x;x:=y;y:=t END;
           z:=x-y; write(x,'-',y,'=')
         END;
      2: BEGIN x:=rnd(100);y:=rnd(100);         {乘法}
           z:=x*y; write(x,'*',y,'=')
         END
    END;
    readln(t);                                  {读入答案}
    IF z=t THEN writeln('Right!')
           ELSE writeln('Wrong! Correct answer: ',z)
  UNTIL false
END .
```

程序中采用 false 作为条件的 UNTIL 循环（实际是死循环）。结束时可以利用操作系统提供的强迫退出手段（如 PC-DOS 下的 Ctrl-Break 键）。

[例 2] 求出 2～1000 以内的亲密数对。

何谓亲密数对？如果正整数 A 的因子和为 B，B 的因子和又为 A，同时 A 和 B 又不相等，则称 A 和 B 为一对亲密数。正整数 A 的因子，指的是能够整除 A 的所有正整数，但不包括 A 本身。 如 12 的因子为 1，2，3，4 和 6，其因子和为 1+2+3+4+6=16，而 16 和因子和为 1+2+4+8=15，故 12 和 16 不是一对亲密数。

6 的因子和为 1+2+3=6，但上面已指出过亲密数对不包括相等的数，故 6 和 6 不算亲

密数对。

我们先给出程序：

```
PROGRAM examp5(output);
  VAR   a,b,c:integer;
  PROCEDURE fac(x:integer; VAR y:integer);
    {求出 x 的因子和送入 y}
    VAR   k:integer;
  BEGIN
    y:=0;
    FOR k:=1 TO x DIV 2 DO
      IF x MOD k=0 THEN y:=y+k
  END;
BEGIN
  FOR a:=2 TO 1000 DO
    BEGIN
      fac(a,b);
      fac(b,c);
      IF (a=c) and (a<>b) THEN writeln(a:6,b:6)
    END;
END.
```

运行结果：

```
  220   284
  284   220
```

程序中计算因子和是通过过程 fac 来实现的。由于整数 x 的最大因子不大于 x 的一半，所以循环终值取 x 的一半。主程序中用尝试法确定亲密数对。

这个题目中求因子和的算法也可以编成函数的形式，请读者自行考虑。

8.11 小结

这里不打算全面回顾本章的内容，只对几个要点作一些讨论。

8.11.1 子程序的意义及抽象思想方法

子程序的引入，不仅可以用来缩短程序篇幅，更重要的是它为程序设计的抽象思想方法提供了一种方便的表达形式，便于设计的层次化、模块化，可以产生条理清晰的设计。

但是要注意，子程序只是提供了方便的条件，它并不能代替人来思考。抽象的思想方法要求将一组具体的操作概括为尽可能简明的一个概念。如果思想方法不好，不符合这个原则，即使用了子程序，也不能保证编出的程序不混乱。请看下面的程序：

```
PROGRAM examp(input,output);
VAR
```

```
  r,k:integer;
  a,b,c,d:real;
  PROCEDURE getfac(m,n:integer; VAR s:real);
    VAR
      i:integer;
  BEGIN
    {要求主程序中将 s 事先初始化为 1}
    FOR i:=1 TO n DO s:=s*i;  {将 n!送入 s}
    CASE m OF
      1: d:=s;
      2: d:=d/s;
      3: d:=d/s
    END
  END;   {  getfac     }
BEGIN
  write('Input r & k : ');
  readln(r,k);
  IF r<k
    THEN
      BEGIN
        a:=1; getfac(1,k,a);
        b:=1; getfac(2,r,b);
        c:=1; getfac(3,k-r,c);
        writeln('k!/(r!*(k-r)!=',d)
      END
    ELSE
      writeln('r>k,input data error!')
END.
```

这个程序与 8.2 节[例 1]中的程序功能一样，只是子程序和主程序的分工有些调整。首先它在子程序内没有对 s 初始化，将这个工作推给了主程序。其次，它在主程序中没有作

```
d:=a/(b*c)
```

的操作，而是把这个工作编入了子程序。为此，增加了一个参数 m，令 m 为 1，2，3 时分步完成这个操作。

但是这样一调整，它的条理性就比 8.2 节[例 1]中的程序差远了。原来的程序中，子程序 getfac 的功能可以概括为一句话

n!→s

而现在的 getfac 就难以概括了。现在的 getfac，前一半基本上可说是“将 n!送入 s”， 但

还缺少 s 的初始化操作；后一半的功能就更难以说清了。原来的 8.2 节[例 1] 中的程序可以将子程序和主程序分给两个程序员来设计，而现在的程序若要分给两个人，则必须要有第三个总负责人来交代清楚他们各自的任务，而“交代清楚任务”所需花费的工作量不小于编完整个程序所需的工作量。

在某些缺少良好训练的程序员设计的程序中，与这个程序类似的毛病是很常见的。这一类的毛病使得软件调试及维护的难度成倍地甚至成几个数量级地增加。

现代软件工程的思想，要求模块化的程序中，各模块（如上例主程序和子程序可以分别算作一个“模块”）应做到尽可能高的“内聚”和尽可能低的“耦合”。换句通俗些的话说就是模块要有“独立性”，一个模块内的各操作最好都是为实现同一个功能而结合在一起的，不同模块之间要尽量减少依赖和影响。对比上面的程序和 8.2 节[例 1]中原来的程序可以体会这个思想。

8.11.2 子程序数据的传递

按照上面的讨论，子程序和主程序之间，子程序和子程序之间，不必要的数据传递应该越少越好。

例如，我们设计了一个函数 f，调用时写一个 f(x)，希望将 x 的值传给子程序，将求出的函数值传回来，这两个传送是必要的。但是如果在子程序内使用全局变量 m 暂存某个数据，这就把数据传送给了主程序管辖下的变量 m。主程序中执行 f(x)时会自动改变 m 的值，这样 f(x)就成了“有副作用的函数”，会带来许多复杂因素。所以，子程序内临时存放数据最好不使用全局变量。

但是，为了避免失去必要的灵活性，PASCAL 中还不能禁止“有副作用的函数”之类的设计（如前面有些例子不用全局变量是无法设计的），所以只能由程序员自行掌握其限度，只在必须传递数据时才进行传递。

按我们已经讲过的知识，主程序向子程序传递数据可以有以下手段：

(1)利用值参数；

(2)利用变量参数；

(3)利用全局变量。

这三种手段中通常使用前两种，而第(3)种手段只在数据本身具有全局的意义时才用， 不宜滥用。

子程序向主程序传回数据可以有以下手段：

(1)利用返回值（只限于函数）；

(2)利用变量参数；

(3)利用全局变量。

同样我们反对滥用全局变量。

以后学了指针一章，我们会知道子程序向主程序传回数据还有第四种手段。

在有些应用程序中，子程序内有时希望把某个数据保留下来供下次调用这个子程序时使用。此时，如果采用前两个手段将它传回主程序，下次调用再从主程序传入子程序，则因为主程序里还须安排变量来存放它，略有些复杂化，不如将这个数据直接放到全局变量里简单些。例如 8.10 节介绍的随机函数中的 seed，以及 8.8 节[例 2]中的 ch1 和 ch2 都

是这种情况。这种“全局”变量，除了初始化以外，在主程序中及别的子程序中基本上是不引用的。如果随便引用，也会增加“耦合”度，从而增加程序的复杂性。顺便指出，某些其它语言如 C 语言，引入了一种“静态局部变量”来解决这一类问题，而 PASCAL 中只能使用全局变量。

8.11.3 本章的学习方法

上面的讨论有一个共同的精神，就是程序设计应提倡简明，反对复杂化。

但是在本章的学习中我们会发现，系统在实现本章的若干概念时有许多内部的细节动作。如调用和返回时的实际程序流程，局部变量的建立和撤销，形参和实参的关系等。这些动作由人分析起来难免感到繁琐和复杂。有时我们在例题、练习题以及测验题、竞赛题等等中间还会遇到些故意复杂化的“怪”程序来让我们分析。

其实，这是同一个事物的两个方面。要能应用本章的概念来设计条理简明的应用程序，首先必须对这些概念有正确而深入的理解。如果我们的理解不够深入，或有误解，也不可能有正确而灵活的应用。而分析这些内部细节正是检验我们对概念的理解是否正确。所以对于概念的理解尚不清楚的初学者来说，作这一类的练习是有必要的。

但是，了解了原理以后我们不应该停留在这些细节动作上，应该知道这些细节动作是为实现外观的功能服务的，有了这些繁琐的内部动作才会有简单易用的外观功能。所以真正设计应用程序时没有必要再去抠内部的动作，只要按照外观功能进行宏观的设计就行了。内部的动作是机器做的，宏观的设计是人做的。对机器繁琐对人简单，正是这些技术的一个优点。这也是我们所说的“抽象方法”的一个体现。假如我们把这些细节当成故弄玄虚的游戏，摹仿那些供练习分析用的“怪”程序去设计真正的应用程序，那就是彻底的南辕北辙了。

所以说，学习本章的知识，应该既能“深入”，又能“浅出”。不仅要知道 PASCAL 语言的有关规则和原理，还应学会科学的程序设计方法。

习题

8.1 分析以下程序的运行结果。

```
PROGRAM exam1(output);
  VAR
    a,b:integer;
  PROCEDURE p(x,y:integer);
    VAR
      t:integer;
  BEGIN
    t:=x; x:=y; y:=t;
  END;
  PROCEDURE q(VAR x,y:integer);
    VAR
```

```
    t:integer;
  BEGIN
    t:=x; x:=y; y:=t;
  END;
BEGIN
  a:=3;   b:=8;
  writeln('Before calling p:','a=',a:2,'b=',b:2);
  p(a,b);
  writeln('After  calling p:','a=',a:2,'b=',b:2);
  a:=3;    b:=8;
  writeln('Before calling q:','a=',a:2,'b=',b:2);
  q(a,b);
  writeln('After  calling q:','a=',a:2,'b=',b:2);
END.
```

8.2 分析以下程序的运行结果（设运行时输入 3　7　20）。这个程序编译时无错，但实际设计有错，请估计设计者的原意及错在何处。

```
PROGRAM exam2(input,output);
  VAR
    a,b,c:integer;
  FUNCTION diff(a,b:integer):integer;
  BEGIN
    IF a<b THEN
      BEGIN c:=a; a:=b; b:=c END;
    a:=a-b;
    diff:=a
  END;
BEGIN
  readln(a,b,c);
  writeln('abs(a-b)=',diff(a,b));
  writeln('abs(b-c)=',diff(b,c))
END .
```

8.3 编写将方阵转置的过程。设所处理的方阵是如下的二维数组：

```
TYPE mat=ARRAY [1..N,1..N] OF real;
```

其中 N 是上文已经定义的一个整数符号常量。以数组名作为参数。

8.4 用递归调用的方法编写求数组中前 n 个成分的连乘积的函数。 设所处理的数组是如下的类型：

```
TYPE arr=ARRAY [1..M] OF real;
```

其中 M 是上文已定义的一个整数符号常量。这里的 n 不大于 M。设计的函数以数组名和 n

作为参数，所得的积作为返回值。

第九章　集合和记录

前面我们学习了所有的简单数据类型及数组类型，有了它们，许多实际问题都可以得到解决。然而，作为一种高级程序设计语言，还需要有能够简单方便地解决问题的更多手段，这就需要引入更多的数据形式。

举例来说：比如我们要判断用户输入的月份 month 是否为 31 天的大月， 按以前学过的办法，可作以下逻辑判断

```
(month=1) OR (month=3) OR (month=5) OR (month=7) OR
  (month=8) OR (month=10) OR (month=12)
```

这个表达式是相当冗长的。如果引入本章介绍的集合的概念，则这个判断也可以写作

```
month IN [1,3,5,7,8,10,12]
```

这里的[1,3,5,7,8,10,12]是由 7 个整数组成的集合，IN 可读作“属于”。引入了集合的概念，不仅可用于上述的判断，而且有关集合的许多运算都可以处理。

又如，数组可以将一组数据当成一个整体操作，这一特点在许多实际问题中是有用的。但是，数组的成分必须是同一类型。有时我们希望将相关联的几个数据合在一起整体处理，而它们可能不是同一类型，如整理考试结果时需要将一个考生的三项数据：考号、姓名、成绩当成一个整体来处理，这就不能用数组，而可以采用本章介绍的记录。

下面我们将分别来介绍这两种新的构造型数据类型——集合和记录。

9.1　集合

9.1.1 什么是集合

按数学上的描述，一个集合是能作为整体论述的事物的集体。组成集合的每个事物叫做这个集合的成员。

PASCAL 语言中对此又作了限制：一个集合中的每个成员必须是同一个顺序类型的值。这样，确定了一个顺序类型后就可以确定一类集合。这样的一类集合也就是一个集合类型。而这个顺序类型就称作这个集合类型的基类型。用数学上的说法就是：PASCAL 中的集合必须以某个顺序类型的值域作为其论述域，而 PASCAL 中一个集合类型的值域就是同一论述域中的所有集合。

例如，确定了基类型为 1..3，所确定的集合类型中的集合就可以有以下取值：

```
[ ], [1], [2], [1,2], [3], [1,3], [2,3], [1,2,3]
```

这里所用的方括号形式叫“集合构造符”，是 PASCAL 中表记集合值的一种表达式，下文还要详细说明。

关于集合的概念，还有几点需要强调：

一个集合的取值，只能表示它含有哪些成员，而不表示它所含成员间的其它关系。例如，上述的[2,3]只表示它含有 2 和 3，而不表示 2 在前还是 3 在前，所以，不管写作[2,3]还是写作[3,2]，表示的都是同一个集合。

一个集合的取值，只能表示它是否含有某个成员，而对同一成员，则不可能表示它重复含有“几次”。例如，上述的[2,3]，即使写成[2,2,3]，也仍然只表示它含有 2 和 3，和原来写作[2,3]表示的仍是同样的一个集合。

9.1.2 集合类型的定义及其变量说明

集合类型的新类型描述格式如下：

```
SET OF 基类型
```

其中“基类型”应是一个表记顺序类型的类型表记符，也就是说，可以是顺序类型的标识符，也可以是新顺序类型的描述格式。应注意，基类型不可以是实型，也不可以是构造类型。

我们来看一下下面的定义：

```
TYPE
  weekday=(sun,mon,tue,wed,thu,fri,sat);
  dayset=SET OF weekday;
  num=SET OF integer;
  ch=SET OF 'A'..'Z';
VAR
  a1,a5:dayset;
  a2,a3:ch;
  a4:SET OF 1..12;
  a6:SET OF (red,green,blue);
```

其中 TYPE 部分定义了三个集合类型标识符：dayset，mun 及 ch。dayset 和 num 的基类型写的是类型标识符，而 ch 的基类型写的是新类型描述格式。VAR 部分中定义了 6 个集合变量。其中 a1，a2，a3 及 a5 是引用前面定义的类型标识符来说明的，而 a4 及 a6 没有定义集合类型标识符，是直接将描述集合类型的格式写到了变量说明中。第六章已经说过，这两种用法都是可以的。

但注意，在某些实际系统中对集合的成员个数有所限制，上述 num 的定义在某些 PASCAL 编译中是通不过的，即某些系统不允许用 integer 作为集合的基类型，因为 integer 类型的取值范围太大了。实际编程时请参考所用的具体 PASCAL 系统的说明书。一般应用中，通常是根据实际需要选取一个较小的子域作为集合的基类型。

9.1.3 集合构造符

集合构造符是表记集合的一种表达式。

把表示集合中各个成员的值的表达式放入一对方括号中，成员之间用逗号隔开，就构成了一个集合构造符，如

```
[1,3,5,7,8,10,12]
[sun,sat]
[x+y,x-y,x*2]
```

其中最后一行的三个表达式的值应属于某集合的基类型。

集合构造符表记由这些成员组成的集合。有这样几个问题要说明：

(1)集合构造符中成员出现的次序不影响集合的值，即

[1,3,5]=[5,3,1]

(2)一个集合构造符中如果有相同的成员，视为同一个，如

[1,3,5,7,5]=[1,3,5,7]

(3)无任何成员的集合称为空集合，用[ ]表示。

(4)集合构造符中的每个成员可以用基类型的表达式来表示，如

[1+3,5,7]=[4,5,7]

又如

[2*x,6,7,8]

当 x=2 时上式等于[4,6,7,8]，当 x=3 时上式等于[6,7,8]。

(5)在集合构造符中，对于连续的值可用..来“省略”，如

[5,10,11,12,13,14,15,16,17,18,19,20,27]

可以写作

[5, 10..20, 27]

注意这里的形式很像子域类型的描述格式，但有一个很大的不同点：这里的“下界”和“上界”都允许不是常量，而是一般的表达式，只要值属于基类型就行。如

[x..y]

在 x 和 y 的值未定时代表的成员个数是不定的。所以这一条规定不仅是简单的“省略”，而是提供了很大的灵活性。此时，若“下界”大于“上界”，则这种写法不代表任何成员。因此，若 x>y，则[x..y]是空集。

当一个集合的基类型的值域有 N 个取值，则可赋给这个集合的值有 2^N个， 如本节开头已经说过，SET OF 1..3 类型集合的取值有 8 个。

学会了集合的表示，我们来看一下如何对集合变量赋值，集合变量的赋值只能通过赋值语句来完成，不能通过输入语句（指从标准输入设备输入）来完成。如 read(a1)是错的，而

```
a1:=[sun,sat];
a2:=[ ];
a3:=['B','D','M'];
a4:=[1,3,5,7,8,10,12];
```

是正确的。按照第六章讲过的赋值相容的原则，只要右边集合的每一个成员的值都属于左边变量所属类型的基类型，赋值语句就是合法的。

9.1.4 集合的运算

对集合的操作比较特殊，它不同于其它的结构类型，我们不能对其中的某一个成员进行操作，而只能对集合整体进行操作。对于集合类型的数据，PASCAL 提供了八个运算，分别是+，-，*，=，<>，<=，>=，IN。其中+，*，-的运算结果为集合类型数据， 后五个运算为关系运算，运算结果为布尔型 。下面分别介绍这些运算符。

这八个运算都是二元运算。前七个运算中，两个运算数必须是类型相容的集合。最后一个 IN 运算中，第一运算数是基类型的值，第二运算数是集合。

(一)集合并 +

作用是把两个集合的所有成员合并起来，形成一个新的集合。例如

a1　　等于　[mon,sat]

a5　　等于　[sun,sat]

则

a1+a5　等于　[sun,mon,sat]

注意同一成员合并后只出现一次。

(二)集合交 *

作用是建立一个由两个集合公共成员构成的集合。如

a1　　等于　[sun,mon,sat]

a5　　等于　[mon,sat,fri,tue];

则

a1*a5　等于　[mon,sat]

若

a1　　等于　[sun,sat]

a5　　等于　[tue,wed]

则

a1*a5　等于　[ ]

若

a1　　等于　[sun,sat]

a5　　等于　[sun..sat]

则

a1*a5　等于　[sun,sat]

(三)集合差 -

作用是建立一个由属于第一个集合但不属于第二个集合的成员构成的集合。如:

a1　　等于　[sun,sat]

a5　　等于　[thu,fri,sat]

则

a1-a5　等于　[sun]

a5-a1　等于　[thu,fri]

若

a1　　等于　[sun,sat]

a5　　等于　[mon,tue,wed,thu]

则

a1-a5　等于　[sun,sat]

a5-a1　等于　[mon,tue,wed,thu]

若

a1　　等于　[sun,sat]

a5　　等于　[sun..sat]

则

a1-a5　等于　[ ]

a5-a1　等于　[mon..fri]

(四)集合等于 = 和不等于 <>

作用是判断两个集合是否由完全相同的成员组成。如:

[sat,sun]=[sun,sat]　为 true

[sat,sun]=[sat,fri]　为 false

[sat,sun]<>[sat,fri]　为 true

(五)被包含 <= 和包含 >=

作用是判断一个集合是否为另一个集合的子集。若前一个集合的所有成员都是后一个集合的成员,则称前一个集合是后一个集合的子集,或称前一个集合被后一个集合“包含”,此时<=结果为 true，否则为 false。反之，若后一个集合是前一个集合的子集， 则称前一个集合“包含”后一个集合，此时>=结果为 true，否则为 false。空集[ ]是任何集合的子集。如：

[sat,sun]<=[sun,sat,fri] 为 true

[sat,sun]<=[sun..sat]　为 true

[sat,sun]<=[sun,sat]　为 true

[ ]<=[sun,sat]　为 true

[sat]<=[mon..fri]　为 false

[sat]>=[ ]　为 true

[mon..sat]>=[sun,sat]　为 false

注意上述集合运算要求类型相容，基类型不相容则集合也不相容，如

[sat]*[1,3,5]

是不合法的。

(六)属于 IN

作用是测试某值是否为该集合的成员，是则结果为 true，否则结果为 false。其中 IN 后必须是一个集合，而 IN 前面是一个和此集合的基类型相容的表达式。如：

4 IN [1..5]　为 true

6 IN [1..5]　为 false

注意如写成[4] IN [1..5]则是错误的，在 IN 的前面，不能是一个集合构造符。

9.1.5 集合的输入输出

集合是不能直接通过标准输入输出设备来输入输出的，这里介绍实践中与集合有关的某些输入输出操作方法。

有时我们想输入一批成员数据来构成一个集合,可以先使一个集合变量为空集[]，然后每读入一个成员数据，就把它并入到该集合变量中去，如此反复输入并将输入值并入变量中，最后在这个变量中就得到一个由输入数据组成的集合。例如，假设 a 是一个本节开头定义的 ch 类型的集合变量，s 是 char 型的变量，则以下程序

```
a:=[ ];
read(s);
WHILE s<>'#' DO
  BEGIN
    a:=a+[s];
    read(s)
  END
```

如果执行时输入

```
ABSCR#
```

则得到 a 集合为['A','B','S','C','R']。

要注意我们这里需要的是“成员”并入“集合”，而前面学过的+ 运算则是集合与集合相并，这是不同的。所以这里必须先把“成员”变成由一个成员组成的集合，才能使用 + 运算与集合变量相并。程序中的

```
a+[s]
```

其中的[s]就是由一个成员 s 构成的集合，如果写成 a+s 则是错的。

有时我们想通过标准输出设备列出某集合中的全部成员，则只能逐个判断其基类型的每个可能取值是否属于该集合，把属于该集合的值输出出来。如

```
FOR s:='A' TO 'Z' DO
  IF s IN a THEN write(s:3)
```

这时若将 s IN a 写成 [s] IN a 又是错的。

由于成员在集合内是无次序的，故输出的次序完全由我们的编程算法决定，上述程序是按字母表的顺序（也就是 ASCII 码的次序）输出的。

9.1.6 应用举例

[例 1] 输入一系列字符，将其中元音字母，辅音字母，非字母字符分别计数，并输出计数结果，当输入'!'时结束输入，最后的字符'!'也统计在内。

这三类字符的编码比较分散，直接判断比较麻烦，可以先建立几个集合，用 IN 运算来判断，可使程序简短些。程序清单如下：

```
PROGRAM exam91(input,output);
  TYPE
    chset=SET OF char;
  VAR
    vowel,conson:chset;      {元音、辅音集合}
    ch:char;
    cv,cc,co:integer;        {元音、辅音、其它字符计数}
BEGIN
  vowel:=['a','o','e','i','u','A','O','E','I','U']; {元音}
  conson:=['a'..'z','A'..'Z']-vowel;                 {辅音}
  cv:=0; cc:=0; co:=0;
```

```
  writeln('Please input:');
  REPEAT
    read(ch);
    IF ch IN vowel
      THEN cv:=cv+1
      ELSE IF ch IN conson
             THEN cc:=cc+1
             ELSE co:=co+1
  UNTIL ch='!';
  writeln('Vowel chr. :    ',cv:5);
  writeln('Consonant chr. :',cc:5);
  writeln('Other chr. :    ',co:5)
END.
```

运行结果如下：

```
Please input:
AF?3E)5hjTgsJ#D79=XEkLdi!
Vowel chr. :        4
Consonant chr. :   12
Other chr. :        9
```

一个集合类型的数据表达的信息中，只可以确定含有哪些成员，不含有哪些成员，而不可以确定第几个成员是什么。这么说，它表达的信息相当于一个以其基类型为下标类型的布尔数组所能表达的信息。第七章中我们在筛法求素数的例题中曾用一个布尔数组代表一批整数，显然用集合也可以代表同样的信息。原则上说，凡是能用一维布尔数组解决的问题，都可以用集合解决。下面我们将给出用集合编写的筛法程序。

另外，集合比布尔数组有更强的功能。这是因为布尔数组没有整体运算，而集合却有许多整体的操作功能，如判断两集合是否相等，是否包含等等。所以程序可以编得更简单。这在下面的例子中就可以看到一些。

[例 2] 用筛法求 n 以内的所有素数（n 取 250），利用集合实现。

思路与第七章一样，只是不用布尔数组而用一个集合代表筛。筛中有 i，或 i 已被划掉，均以 i 是否筛的成员而定。这样，可以编出一个与第七章程序完全对应的程序如下：

```
PROGRAM prime(output);
  {用筛法求 250 以内的所有素数}
  CONST
    n=250;
  VAR
    sieve: SET OF 2..n;
    i,j:   integer;
BEGIN
```

```
  sieve:=[2..n];                                  {将 2～n 记入筛中}
  FOR i:=2 TO trunc(sqrt(n)) DO
    IF i IN sieve THEN                            {若 i 未被划掉}
      BEGIN                                       {划掉其倍数}
        j:=i+i;
        WHILE j<=n DO
          BEGIN sieve:=sieve-[j]; j:=j+i END
      END;
  FOR i:=2 TO n DO IF i IN sieve THEN write(i:4); {输出}
  writeln
END.
```

对比这个程序和第七章的程序，可以看到集合与布尔数组的某些对应关系。

下面再讨论可否利用集合的特点对上述程序作一些改动。

首先，原来确定一个素数后，要将其倍数从筛中划掉，但保留它本身。现在若改为连同它本身也划掉（当然要先输出），则整个循环就可以改为以“筛空”作为结束条件，不必再考虑 i 的终值了，可以不用计数循环。

其次，原来只在 i 未被划掉时才划掉其倍数，若 i 已被划掉则整个循环体成为空操作，i 增量并判断终值后进入下一循环。现在既然不必判断 i 的终值，则 i 的增量可以编作内层循环，进一步简化操作。

下面是改后的程序：

```
PROGRAM prime(output);
  {用筛法求 250 以内的所有素数}
  CONST
    n=250;
  VAR
    sieve: SET OF 2..n;
    i,j:   integer;
BEGIN
  sieve:=[2..n];                          {将 2～n 记入筛中}
  i:=2;
  REPEAT
    WHILE NOT (i IN sieve) DO i:=i+1;     {i 增量，跳过已划掉的}
    write(i:4);                           {输出}
    j:=i;                                 {划掉自身及倍数}
    WHILE j<=n DO BEGIN sieve:=sieve-[j]; j:=j+i END
  UNTIL sieve=[ ];                        {筛空结束}
  writeln
END.
```

这些改动，在采用布尔数组时是不行的，因为布尔数组要判断“筛空”必须经过一个复杂的循环，而现在只要与空集进行一次比较判断就行了。

9.2 记录

记录是一种构造型数据。与数组类似，它也是由若干个成分组成的。但记录与数组有两个方面的不同。

一是记录中各个成分的类型可以不同。

二是记录中的各个成分不是靠下标来区分，而是靠不同的“域名”来区分的（记录中的成分称作“域”）。因此不能用下标递增或递减的方式来遍历记录的各个成分。但是取域名时可以照顾各个域的实际意义，有助于人理解程序。

9.2.1 普通记录的定义

不同的域结构可以构成不同类型的记录，所以记录是自定义的类型。记录类型的新类型描述格式一般如下（未包括变体部分）：

```
RECORD
  标识符，标识符，…，标识符：类型表记符；
  标识符，标识符，…，标识符：类型表记符；
        ……
  标识符，标识符，…，标识符：类型表记符；
  标识符，标识符，…，标识符：类型表记符
END
```

其中 RECORD 和 END 之间的部分称作“域表”。

域表包括多个“记录段”，记录段间由分号隔开。上面的每一行除最后的分号外就是一个记录段。

每个记录段中冒号之前的标识符表中的每个“标识符”都是定义的域名，冒号后的“类型表记符”规定了这个记录段中定义的域的类型。

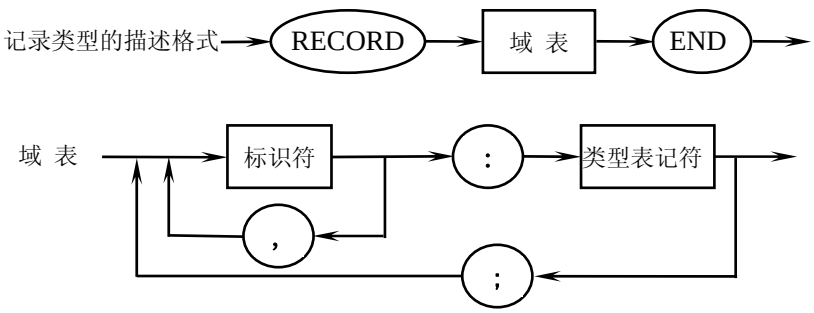


图 9.1 记录类型的描述格式的语法图（未包括变体部分）

整个域表中定义了多少个域标识符（即域名），这种记录就包括多少个域。

上述格式的语法图见图 9.1。

可以看到，一个记录段的语法形式和变量说明的形式一样，但这里定义的标识符是域标识符。标准中规定域标识符的作用域很小，因此，除同一个记录类型的描述格式里不可以重名以外，域标识符与其它地方的标识符允许重名，不同记录类型的描述格式里的域标识符也允许互相重名，互不影响。当然初学者为了避免混乱，还是不重名为好。

例如可以定义如下的记录类型作为学生通讯录：

```
RECORD
  name:PACKED ARRAY [1..20] OF char;
  class,dormitory:integer;
  address:PACKED ARRAY [1..30] OF char
END
```

此记录类型包含有 4 个域。第一个域 name 存放姓名，第二个域 class 是班级代号，第三个域 dormitory 是宿舍号，第四个域 address 是家庭地址。其中姓名和家庭地址都是字符串，班级代号及宿舍号都是整数。

这样的格式可以出现在类型定义的 TYPE 部分中，如

```
TYPE
  memo=RECORD
         name:PACKED ARRAY [1..20] OF char;
         class,dormitory:integer;
         address:PACKED ARRAY [1..30] OF char
       END;
```

按照第六章 6.1 节的原则，这个格式也可以出现在变量说明的冒号后面， 或其它允许写类型表记符的地方（如第十一章中用在文件类型描述格式中作为成分类型）。

记录中一个域的类型可以是简单类型，也可以是构造类型，如上面例子中以数组作为记录的一个域。记录的域还可以又是记录，从而形成多层的记录。例如：有学生记录包括学号、姓名、性别、出生日期及 4 门课成绩这样几项，其中出生日期又包括年、月、日几项， 可以定义如下:

```
TYPE
  date=RECORD
         year:integer;
         month:1..12;
         day:1..31
       END;
  stdata=RECORD
           num:PACKED ARRAY [1..7] OF char;
           name:PACKED ARRAY [1..15] OF char;
           sex:char;
           birthday:date;
           score:ARRAY [1..4] OF real
         END;
```

从上面看到，在 stdata 的定义中用到了前面定义的 date，按照“先定义，后引用”的原则，这两个定义的顺序不可颠倒。

记录类型还可以作为其它构造类型的成分，如下面的例子里会看到以记录为成分的数

组，第十一章会看到以记录为成分的文件。另外，第十章还会见到，以记录作为指针的定义域类型，可以构成复杂的动态数据结构。总之，在较大型的数据处理应用程序中记录是很有用的数据形式。

有了记录类型后，就可以定义记录类型的变量了。假如在 TYPE 部分中已经定义了如上所述的 memo，date，stdata，则可有以下变量说明：

```
VAR
  a,b:memo;
  vec:ARRAY [1..20] OF memo;
  cur:date;
  doc:stdata;
  cycque:RECORD
           queue:ARRAY [0..255] OF real;
           rear,front: integer
         END;
```

其中 vec 就是一个以 memo 型记录为成分的数组变量。cycque 也是一个记录变量， 只是它的类型没有在 TYPE 中定义，而是直接写在了变量说明的冒号后面了。

9.2.2 记录的引用

一、同一类型的记录变量之间可以直接进行赋值，如：假设各变量已经像上文 VAR 中那样说明，则

```
a:=b
vec[2]:=a
b:=vec[10]
```

都是合法的。将一个记录赋给另一个记录变量，实际上同时传送的各个对应的域。

二、除了上述情况外，对记录的访问只能通过对其域的访问来实现，记录变量中的每个域分别作为一个成分变量，都能参与其成分类型所允许的运算和操作。对记录中一个域的引用格式是

```
记录变量.域名
```

如对 a 记录中各域的引用如下：

```
a.name
a.class
a.dormitory
a.address
```

这一种形式称作域命名符，它也属于第二章中所讲的“成分变量”的一种。它的数据类型就是记录类型的描述格式中所规定的该域的类型。如：a.class 就是一个整型的变量，可以出现在任何允许出现整型变量的地方（另有规定处除外）。

例如，下列语句都是合法的：

```
a.class=9513;
doc.sex:='M';
```

```
read(b.dormitory);
read(doc.birthday.year,doc.birthday.month,doc.birthday.day);
writeln(vec[2].name:20,vec[2].class:6);
```

注意上述引用格式中的“记录变量”并不限定是整体变量，也可以是其它形式。如上面的

```
vec[2].class
```

其中的 vec[2]属于下标变量，是数组 vec 的一个成分，但它本身是 memo 类型的记录变量。又如上面的

```
doc.birthday.year
```

其中的 doc.birthday 是一个域命名符，是记录 doc 的一个成分，但它本身是 date 类型的记录变量，故后面可以再带“.域名”。

[例 1] 从键盘上读入 N 个学生的数据（下面实现时取 N=5 为例），包括学生学号、姓名、四门课成绩，由计算机算出各人四门课的平均分，并按照平均分由高到低的次序列表输出。

排序的算法在第七章已经见过。如果仅仅是对一组实数（或整数）排序，需要建立一个一维数组，以实数（或整数）为成分，经过一系列的比较判断和交换来完成排序。但是如果数据不仅是这一组实数（或整数），还有其它相关信息需要同时处理，则必须在每交换两个实数（或整数）的同时交换其它相应的信息。如 7.2 节[例 5]中所见到的那样，在交换排序的成绩的同时，还要交换跑道号数组的对应成分。用那样的办法，如果相关信息较多，就得建立多个数组，使用多个交换操作，程序就会变得繁琐。现在有了记录这种形式，就可以将相关信息合成一个记录，采用以记录为成分的数组，排序中以整个记录为交换单位。

```
PROGRAM exam94(input,output);
  CONST
    N=5; M=4;
  TYPE
    sturec=RECORD                                {记录类型}
             num:PACKED ARRAY[1..7] OF char;     {学号 7 字符}
             name:PACKED ARRAY[1..15] OF char;   {姓名 15 字符}
             score:ARRAY[1..M] OF real;          {M 门成绩}
             ave:real                            {平均分}
           END;
  VAR
    stud:ARRAY [1..N] OF sturec;                 {记录数组}
    temp:sturec;                                 {记录暂存}
    s:real;
    c:char;
    i,j,p:integer;
```

```
BEGIN
  writeln('Please input num,name & 4 scores:');
  FOR i:=1 TO N DO                          {输入 N 个学生数据}
    BEGIN
      FOR j:=1 to 7 DO read(stud[i].num[j]);    {学号 7 字符}
      read(c);                                  {学号、姓名间空一格}
      FOR j:=1 to 15 DO read(stud[i].name[j]); {姓名 15 字符}
      s:=0;
      FOR j:=1 to M DO                          {M 门成绩}
        BEGIN
          read(stud[i].score[j]);
          s:=s+stud[i].score[j]
        END;
      stud[i].ave:=s/M;                         {平均分}
      readln
    END;
  FOR i:=1 TO N-1 DO          {选择法排序，以 ave 域为关键字}
    BEGIN
      p:=i;
      FOR j:=i+1 TO  N DO
        IF stud[j].ave>stud[p].ave THEN p:=j;
      IF p<>i THEN
        BEGIN                   {交换第 i 项记录与第 p 项记录}
          temp:=stud[i];
          stud[i]:=stud[p];
          stud[p]:=temp
        END
    END;
  writeln; write('No. ':7,'Name      ':16); {输出表头}
  FOR j:=1 TO M DO write('score':7,j:1);
  writeln('average':10);
  FOR i:=1 TO N DO                          {输出数据}
    BEGIN
      write(stud[i].num:7,stud[i].name:16);
      FOR j:=1 TO M DO write(stud[i].score[j]:8:1);
      writeln(stud[i].ave:10:1)
    END
END .
```

运行结果如下：

```
Please input num,name & 4 scores:
9402137 Zhang Sanjiang  68.5  77  79  88
9402138 Wang Weipeng    72  100  61  90
9402139 Li Shuanghong   94  96  85  99.5
9402142 Zhao Zhihui     56  80  72.5  67
9402143 Sun Xiaobiao    100  88  87  76
```

No.	Name	score1	score2	score3	score4	average
9402139	Li Shuanghong	94.0	96.0	85.0	99.5	93.6
9402143	Sun Xiaobiao	100.0	88.0	87.0	76.0	87.7
9402138	Wang Weipeng	72.0	100.0	61.0	90.0	80.7
9402137	Zhang Sanjiang	68.5	77.0	79.0	88.0	78.1
9402142	Zhao Zhihui	56.0	80.0	72.5	67.0	68.9

将上面例子里选择法排序的程序与第七章相比，会发现这里交换时是对整个记录操作，而比较判断时仅仅检查记录中的 ave 域。在数据处理程序对大量的记录进行操作时，常常要将记录中的一个（或几个）域作为特征，据以进行查找、排序等操作。习惯上将这一个（或几个）域称作记录中的“关键字”。

9.2.3 开域语句

程序中，常常在某个段落里处理同一个记录变量的各个域，如果都使用完整的域命名符格式，常常显得过分冗长。为了简化对域名的引用，PASCAL 提供了一个开域语句，又称 WITH 语句，它的基本格式为：

```
WITH  记录变量 DO 语句
```

如果我们程序中一个语句里多处引用同一个记录变量的各个域，可以简写成开域语句的形式，将该记录变量作为上述格式中的“记录变量”部分，将原语句中的所有引用该记录变量的各个域的域命名符中的该记录变量连同其后的园点都删掉后，作为上述格式中的“语句”部分。

反过来也可以说，将开域语句中的“语句”部分里，所有使用该“记录变量”所属的记录类型中的域名处，前面都加上该“记录变量”及一个园点，然后将开头的“WITH 记录变量 DO”删掉，最后所得语句就是原开域语句的等效语句。

如（下面的例句都假定各变量已按本节开头所举的 VAR 部分说明过）：

```
WITH a DO read(name,class,dormitory,address)
```

等效于

```
read(a.name,a.class,a.dormitory,a.address)
```

这里的“记录变量”也可以不是整体变量，如

```
WITH doc.birthday DO read(year,month,day)
```

等效于

```
read(doc.birthday.year,doc.birthday.month,doc.birthday.day)
```

这里的“语句”也可以不是简单语句。所以如果上述情况发生在一个程序段落中，我们可以将此段落用 BEGIN 和 END 括成一个复合语句，再作如上处理。

这里的“语句”也可以又是一个开域语句，这就构成了多重的开域语句。例如

```
WITH doc DO
  WITH birthday  DO
    BEGIN
      read(num,name,year,month,day,sex);
      FOR I:=1 TO M DO read(score[I]);
      readln
    END
```

我们来分析它的等效功能。首先，将外层的 doc 化掉（即将 doc.加到后边的 birthday，num，name，sex，score 之前），得到等效语句

```
WITH doc.birthday  DO
  BEGIN
    read(doc.num,doc.name,year,month,day,doc.sex);
    FOR I:=1 TO M DO read(doc.score[I]);
    readln
  END
```

但它又是一个开域语句，再将 doc.birthday 化掉(即将 doc.birthday.加到 year，month，day 之前)，得到

```
BEGIN
  read(doc.num,doc.name,
       doc.birthday.year,doc.birthday.month,doc.birthday.day,
       doc.sex);
  FOR I:=1 TO M DO read(doc.score[I]);
  readln
END
```

这就是它的等效功能。

PASCAL 中规定对于形如

```
WITH v1 DO
  WITH v2 DO
    ...
      WITH vn DO 语句
```

这样的多重开域语句，可以写成

```
WITH v1, v2, ..., vn DO 语句
```

的形式。这就是开域语句的一般格式。如上面的例子可以写成

```
WITH doc, birthday DO
  BEGIN
```

```
    read(num,name,year,month,day,sex);
    FOR i:=1 TO M DO read(score[i]);
    readln
  END
```

这个一般格式中 WITH 和 DO 之间的部分称作“记录变量表”。

注意在记录变量表中，有些只是一个域名，例如上面的 birthday，它要待前面加上 doc.后才可以算是记录变量。此时记录变量表中的次序不可颠倒，次序的原则应是外层在前，内层在后。其道理从上面的分析中不难明白。

也有时候，记录变量表中是无关的几个变量，如

```
WITH vec[2], cur DO
  BEGIN
    writeln(name:20, class:6);
    writeln(year:4, '/', month:2, '/', day:2)
  END
```

此时 vec[2]和 cur 的次序就无关紧要了。它的等效语句是：

```
BEGIN
  writeln(vec[2].name:20, vec[2].class:6);
  writeln(cur.year:4, '/', cur.month:2, '/', cur.day:2)
END
```

对于 WITH 语句，要说明以下两个问题：

一、假如用一个 WITH 语句处理多个同类记录变量，或虽不是同类，但其域名有重名的情况时，问题就会复杂化。如：

```
WITH a,b DO read(name,class,dormitory,address);
```

这时，就说不清句中的name等域是属于a还是属于b。PASCAL系统的处理是认为它们属于后一个记录变量b[①]。但我们编程时最好避免这种情况，因为这降低了程序的可读性。

二、WITH 语句中，记录变量表中所写的变量实际代表哪个变量，是在整个开域语句刚开始执行时确定的，执行过程中不再重新确定。如下面的语句

```
i:=2;
WITH vec[i] DO
  BEGIN
    ...
    i:=i+1;
    writeln(name:20, class:6);
    ...
  END;
```

虽然 i 在语句中被改动了，但 name 仍是代表 vec[2].name，class 仍是代表 vec[2].class，

[①] PASCAL 中是按照标识符定义的区域相包围时内层优先的原则处理的。

因为刚开始时的 vec[i]是 vec[2]。除了像这样“记录变量”是下标变量的情况以外，如果“记录变量”是后面第十章将讲到的标识变量，在语句中修改了指向它的指针的话，也是这种情况。不过，虽有这种规定，但我们编程时最好避免在语句中修改这种下标或指针的值。有些实际的系统将这种修改判作错误。

这两条说明中所举的情况，如果硬按前面所讲的等效法则来理解，会得到与上面不同的结果。实际上本书前面所讲的等效法则并不是严格按照 PASCAL 中的真正规定叙述的，编译程序处理时并不是按上述的等效法则去变换，而是采用了一种特殊的寻址方式，从而可以有较高的效率。不过，除了这两个问题外，上述等效法则实际上都是成立的，而且这个法则也比较简单，所以本书这样叙述，可以满足实用的需要。

9.2.4 带变体的记录

前面介绍的记录类型，其域结构是固定的。但实用中有时希望记录的域结构可以有所变化。比如图书馆中的每本书的信息，除了它的书号、书名外，还有一项是借出与否，如果借出，就要给出借书人的证号和借出日期，如尚未借出，则给出它在书库中的位置，也就是说后面的几个域是什么取决于借出与否这个域。PASCAL 语言提供了一种带变体的记录来解决这类问题。例如图书馆的书卡记录类型可以描述为以下格式：

```
RECORD
  num:integer;
  name:PACKED ARRAY[1..40] OF char;
  CASE borrowed:boolean OF
    true: (znum:integer;  bdate:date);
    false:(where:integer)
END;
```

这里，RECORD 和 END 之间的部分仍然称作“域表”。 整个“域表”分为“固定部分”和“变体部分”。两个部分均可有可无，但次序不能颠倒。例中前两个域 num（书号）和 name（书名）的记录段就是固定部分，从 CASE 开始是“变体部分”。

固定部分的语法和前面讲的不带变体时的整个域表的语法一样，不过，只要固定部分和变体部分都存在，则固定部分的最后一个记录段后就应有一个分号，以便和变体部分隔开。

在变体部分中有一个特殊的域称作“标志域”，即上例中的 borrowed 域。其后的冒号后面应给出标志域的类型，称作标志类型。按规定，标志类型必须写顺序类型的标识符，上例中是 boolean。标志域及标志类型写在 CASE 和 OF 之间。例中的 borrowed 表示借出与否。

在字符号 OF 后面列出若干个“变体”（例中是两个变体），其间用分号分开。每一个变体表示记录中变体部分可以有的一种构造情况。在程序执行过程中任何一个时刻，这几个变体中最多只有一个是“活动”的。这实际上表示的是存储器中这同一部分空间可以有几种形式安排存放数据，每个变体描述一种形式，当前“活动”的那个变体就表示当前数据的存放形式。

每个“变体”中冒号的左边是“情况常量表”，冒号右边圆括号内又是一个“域表”。

它表示：当“标志域”的值取左边情况常量表中一个常量的值时，记录中存在右边圆括号里域表所描述的构造。如上例中，当 borrowed 域取值为 true 时（表示已借出）记录中包括 znum 和 bdate 这两个域;当 borrowed 域取值为 false 时(表示未借出)记录中包括 where 这个域。换句话说，标志域的值等于哪个变体的一个情况常量，哪个变体就是活动的。

注意，冒号和园括号是必不可少的，即使在某个情况常量下不存在任何域，也应写出一个空的圆括号，可以认为括号内是一个空的域表。假如上述规则改为书未借时不作任何标记的话，则记录类型应描述为：

```
RECORD
  num:integer;
  name:PACKED ARRAY[1..40] OF char;
  CASE borrowed:boolean OF
    true: (znum:integer;  bdate:date);
    false:()
END;
```

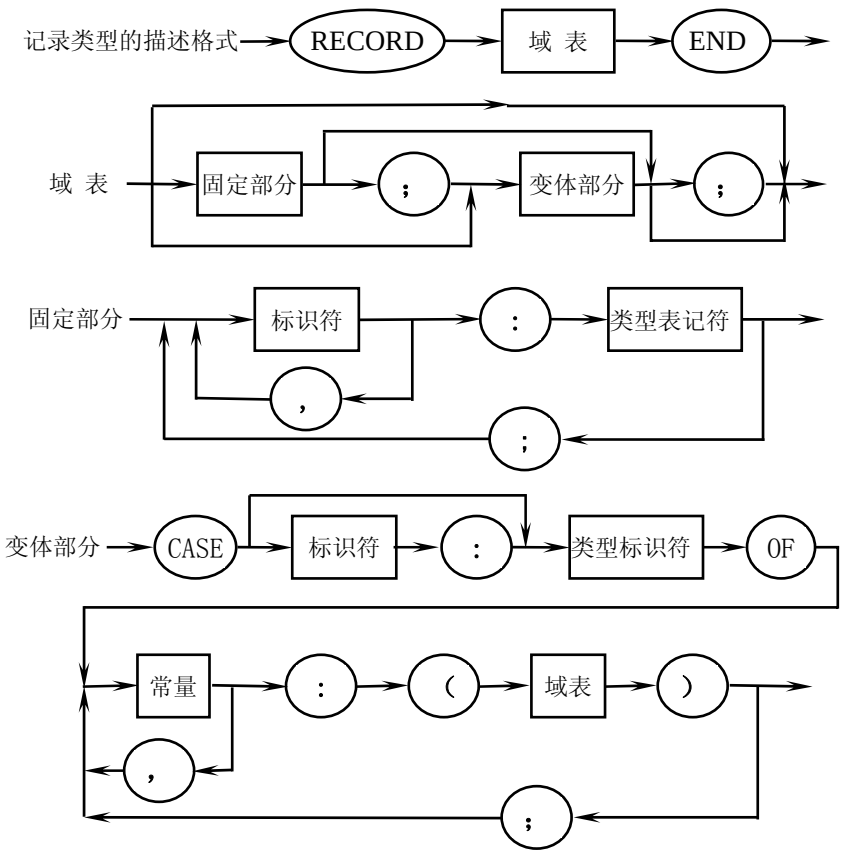


图 9.2　记录类型的描述格式的完整语法图

标志域和变体中的各个域的引用形式与前面所讲的相同，只是不可以访问不活动的变体中的域，否则算作错误。如上例中，若某个记录的 borrowed 域为 false 时，不可以访问它的 znum 域。一个变体由活动的变为不活动的时候，其中各个域的值都“丢了”，即变为“未定义”的了。这是因为各个变体表示的是同一个空间中存放信息的形式，一个变体变为不活动的时候，另一个变体“活动”了，也就是说这个空间要按另一个形式存放信息了，当然原来的信息就不可能保证不被冲掉了。

变体中可以没有标志域，但仍然要有标志类型，这样记录中相当于标志域的那个信息仍然存在，只是它不算一个域，我们不能直接访问它，对它的改变是在访问变体中的域时由系统自动进行的。

注意变体部分格式有些像情况语句，但要注意它没有最后的 END。上例中的 END 是记录类型描述格式中与 RECORD 相匹配的，而不是和变体中的 CASE 相匹配的。

还有，变体中圆括号内又是一个域表，应遵守整个“域表”的语法，所以它又可以分为固定部分和变体部分。因此，变体是可以嵌套的。

包括变体部分，记录类型完整的语法图见图 9.2。

注意，按这个语法允许有空域表。

习题

9.1 阅读下列程序，写出运行结果（提示：eoln 是行结束函数，在第六章 6.2 节[例 2]中讲解过，键盘输入时每按一次回车键产生一个行结束符）。请注意其中有几个输出项目是和输入无关的，分析其理由。

```
PROGRAM test91(input,output);
  TYPE
    alp='A'..'Z';                          {大写字母}
    alps=SET OF alp;                       {大写字母集}
  VAR
    empty,univ,wk1,wk2,wk3,wk4,wk5,wk6,wk7:alps;
  PROCEDURE compl(VAR s1,s2:alps);         {s1 的补集→s2}
  BEGIN
    s2:=['A'..'Z']-s1
  END;
  PROCEDURE xcl(VAR s1,s2,s3:alps);        {s1 和 s2 的异运算→s3}
  BEGIN
    s3:=(s1-s2)+(s2-s1)
  END;
  PROCEDURE outset(VAR s:alps);            {输出 s 集成员}
    VAR c:alp;
  BEGIN
    FOR c:='A' TO 'Z' DO
      IF c IN s THEN write(c);
    writeln
  END;
  PROCEDURE inpset(VAR s:alps);            {输入成员构成 s 集}
    VAR c:alp;
```

```
  BEGIN
    s:=[];
    WHILE NOT eoln DO
      BEGIN read(c); s:=s+[c] END;
    readln
  END;
BEGIN                                    {主程序}
  empty:=[]; compl(empty,univ);
  inpset(wk1); inpset(wk2);
  wk3:=wk1+wk2;
  xcl(wk1,wk2,wk4);
  wk5:=wk1*wk2;
  compl(wk1,wk6); compl(wk2,wk7);
  writeln(univ>=wk3:6, wk3>=wk4:6, wk4*wk5=empty:6);
  outset(wk1); outset(wk2); outset(wk3); outset(wk4); outset(wk5);
  writeln(wk1*wk7+wk2*wk6=wk4:6);
  outset(wk6); outset(wk7)
END .
```

设输入是：

```
DDFDRGHJKLEXRCRBETYHRGTYHTGB
BKTMHKYIOJGNQASDSFPLIOM
```

9.2 将 7.2 节[例 7]“猴子选大王”的例题程序改用集合编写。要求设计两种方案：

方案一：与原程序控制结构完全对应的算法；

方案二：将原用 s 作控制变量的 FOR 循环改成不用计数的条件循环，以猴子退完（空集）作为结束循环的条件，最后一次退出的猴子号就是大王。

9.3 编程：键盘输入 30 个学生的考试资料，每个学生的资料包括

学号（整数）

姓名（20 个字符的字符串）

性别（字符）

成绩（实数）

四个项目。计算机输出其中最高分者的资料及最低分者的资料。

9.4 利用 WITH 语句改写 9.2 节[例 1]程序。

9.5 设有如下的数据类型：

```
TYPE
  date=RECORD
         year:integer;
         month:1..12;
         day:1..31
```

```
        END;
    bookcard=RECORD
                num:integer;
                name:PACKED ARRAY[1..40] OF char;
                CASE borrowed:boolean OF
                  true: (znum:integer;  bdate:date);
                  false:(where:integer)
              END;
    booklib=ARRAY[1..500] OF bookcard;
```

设计一个过程 lookup，有两个参数，第一个参数是一个 booklib 类型的数组， 第二个参数是作为书号的整数，lookup 过程的功能是从该数组中查找该书号的书的信息并输出。若该数组中有此书号，则输出的格式是：先输出书号及书名。然后，若已借出，则输出

Borrowed by

字样，接着输出借阅者的证号（znum）及日期；若未借出，则输出

Not borrowed

字样，接着输出它在库中的架号（where）。若无此书号，则输出

Not found

字样。

第十章　指针和动态数据结构

在本章之前所学习的PASCAL的变量在存储器中的存放方式，从某种意义上说都是属于静态[①]存储。

进入某个子程序时开辟的变量，在该子程序执行过程中，在内存中占有的存储单元是固定不变的。虽然第八章也讲到变量实体的建立与撤消，但那都是随着子程序的调用和返回而发生的，不能在程序的执行过程中随时使用随时分配存储单元，也不能随时释放这些存储单元另作它用。因此称具有这种性质的数据结构为静态数据结构。

静态数据结构的应用具有一定的局限性。

例如，利用数组存放一个飞机乘客表，需要规定数组的长度。由于乘客的人数总在变化，如果数组定义过大，则有些存储单元会被浪费，如定义过小，在运行中可能会发生数组下标超界的错误。

又如，静态结构存放大量数据时只能采用数组一类的顺序存储形式，而在顺序存储的数据中插入或删除个别数据时不得不平移大量数据，操作效率低。

PASCAL 提供了一种称为指针的数据类型，可以通过指针在程序的运行阶段根据需要动态地分配和释放存储单元，因此，我们利用指针可以建立起动态的数据结构。

经常使用的一种结构形式是令一个指针指向一个记录，而在该记录本身还包含有同一类型指针的域，以指向另一个记录，从而利用指针把一个个记录链结起来，形成链式存储结构。

本章着重讨论指针类型定义和指针变量的说明，指针变量的各种操作，以及指针在构造线性链表中的简单应用。

链式存储结构除线性链表外，还可以有“树”，“图”等更复杂的结构。这些不在本书介绍的范围内。

10.1　指针和动态存储

10.1.1 指针的概念

指针是一种特殊的数据。采用不太严格的形象说法，可以说指针就表示存储器中存放变量的地址。如果一个变量 p 是指针类型的变量，那么，变量 p 的值（也就是变量 p 中实际存放的内容）就可以是另一个变量的地址。如果变量 p 的值是变量 x 的地址，我们通常就说“指针 p 指向变量 x”。这样，根据变量 p 就可以找到变量 x。这种根据变量 p 去访问（即读写）变量 x 的访问方式称作“间接寻址”。反之，直接使用变量 x 的名字去访问 x 称作“直接寻址”。

PASCAL 中规定，指针所指的变量不可以是在变量说明部分里说明的变量，也不可以是

[①] “静态”和“动态”两个词在不同的地方使用有不同的含义。本书中将整体变量及其成分都称作“静态”的，但有些资料在描述它们随子程序的调用和返回而建立与撤消的情况时也用“动态”一词。

它们的成分变量，也就是说，不可以是我们学过的静态的变量，而只可以是下面将讲到的由过程 new 创建的“无名”的变量。这种变量可由程序创建，也可由程序撤消， 这也就是我们说的动态变量。这个规定和其它某些语言（如 C 语言和汇编语言）有所不同（比它们更严）。PASCAL 中的动态变量没有一个标识符作为名字，所以它们不能直接寻址，只能间接寻址。而 PASCAL 中的静态的变量则只能直接寻址。

PASCAL 中还规定，一种指针只能指向某一种数据类型的变量，也就是说一个指针变量只能存放某一种数据类型的变量的地址。它所能指向的这个数据类型称为这种指针类型的“定义域类型”。依定义域类型的不同，可以有不同的指针类型，所以指针类型也得由用户根据需要自行定义。

10.1.2 指针类型及指针类型的变量

为了引入自定义的指针类型，PASCAL 规定了新指针类型的描述格式为

```
^定义域类型
```

这里的符号“^”在某些机器上是“↑”。“定义域类型”应是一个已有定义的类型标识符，它可以是任何类型。

例如

```
TYPE
  point=^integer ;
```

定义了一个名为 point 的指针类型，它的定义域类型是整型。又如

```
VAR
  p,p1:^integer;
```

定义了两个可以指向整型变量的指针变量 p 和 p1。为了便于记忆，可以把符号“^”读作“指向”，这样“^integer”就是“指向整数的指针”。

按照第六章 6.1 节讲过的道理，我们知道上述两处的^integer 用法都是对的。 但是应该注意一个问题：

回顾 6.4 节关于赋值相容的叙述，我们知道对于指针类型只有类型同一时才能赋值相容。再回顾关于类型同一的叙述，知道 PASCAL 中不承认两次出现的新类型描述格式是同一类型。假如再有一条变量说明

```
p2:^integer;
```

则这个 p2 和上述的 p1 就不能互相赋值了。所以，习惯上在变量说明中直接写新类型描述格式的不多（虽然这是允许的），而通常是写已在 TYPE 中定义过的类型标识符。例如将上面 p ，p1，p2 后面的^integer 都换成 point，则它们之间赋值就没有问题了。

指针变量的值除了是它所指向的动态变量的地址外，还可以是 NIL。NIL 是一个特定的字符号，表示一个空指针。NIL 不指向任何地址， 把它赋给指针变量后则该指针就不指向任何动态变量。NIL 可用于比较，常用来作指针链的结束标志。

另外，和一般的变量一样，指针变量在第一次被赋值之前其值是“未定义”的。我们说过，引用未定义的值时许多实际的系统能给出一个值，只是不保证这个值是几。对普通的变量，这种情况不太要紧，但对指针变量就应特别注意。因为指针的值是地址，“未定义”的指针值不一定是内存中什么地方的地址，如果程序中用这个指针间接寻址地向动态

变量赋值，则数据被送到内存中不知道什么地方去，就有可能冲坏系统中的重要信息。

综上所述，指针变量的值有以下三种情况：

(1)指向一个动态变量（属于该指针的定义域类型）；

(2)空指针 NIL；

(3)未定义。

注意“空指针”和“未定义”不是一回事。

10.1.3 动态变量的创建和撤消

动态变量只能用预定义过程 new 来创建。过程 new 的调用格式是：

```
new(指针变量)
```

过程 new 的作用是创建一个动态变量的存储空间， 该动态变量的类型属于该指针的定义域类型，同时将该存储空间的地址赋给该指针变量。

例如，假设如下定义了指针变量 p：

```
VAR
  p:^integer;
```

这里说明了 p 是一个可以指向整型动态变量的的指针变量。但刚说明后指针 p 的值尚属“未定义”。然后，若程序中执行了

```
new(p)
```

则在内存中创建了一个整数类型的动态变量，并使指针变量 p 指向这个动态变量。

与其它变量一样，一个指针变量同一时间只能有一个值，即只能指向一个动态变量。若程序再一次执行 new(p)，内存中就又开辟另一个整型动态变量，并把新的地址放入 p 中，于是 p 就不再指向第一次创建的动态变量了。因为动态变量只能间接寻址，如果谁也不指向它，这个动态变量就会“丢失”，无法访问了。所以，如果想用 new(p)来创建新的动态变量，而 p 原来已经指向了一个动态变量，则应考虑到有没有别的指针变量也指向它，必要时可以将 p 的值先赋给另一个指针变量。

当不再需要某动态变量时，可以通过预定义过程 dispose 来撤消它，其形式如下：

```
dispose(指针)
```

执行 dispose 过程后，实参指针所指的动态变量被撤消， 它所占据的存储空间被释放另作它用，程序中所有原来指向这个动态变量的指针变量的值变得无定义。这里 dispose 的参数是值参数。

这种动态变量的生存期不受子程序的调用和返回的限制。主程序中创建的，进入子程序后仍然可用；子程序中创建的，回到主程序后也仍然存在，仍然可用。只要不用 dispose 去撤消，它就一直存在到整个程序结束。如果程序中对动态变量只创建不撤消，创建太多时有可能会出现“存储器不够用”的错误。

10.1.4 动态变量的引用

动态变量没有一个作为名字的标识符，必须通过指针引用。引用一个指针变量所指向的动态变量的形式如下：

```
指针变量^
```

如上例用 new(p)开辟了一个整型动态存储单元后，就可以用形式

p^

来引用这个动态变量，它是一个整数类型的变量。同样可以把符号“^”读作“指向”， 这样“p^”就是“p 所指向的动态变量”。

这种形式表示的变量称作标识变量（第二章的 2.5 节已经提到过）， 它和学过的其它形式的变量（整体变量、下标变量、域命名符）一样可以出现在任何允许出现变量的地方（除另有规定的情况外）。如可以出现在表达式中，可以出现在赋值号左边，可以作为参数出现在 read 语句中，等等。

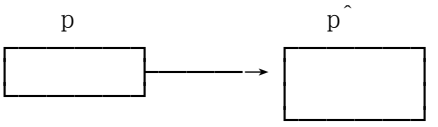


图 10.1

指针变量 p 和它所指的动态变量 p^的关系可用图 10.1 表示。

以下语句把整数 256 存入 p 所指的动态变量中:

p^:=256

结果如图 10.2 所示。

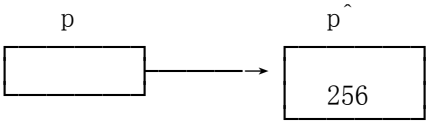


图 10.2

可以用指针指向一个记录，如在程序中如下说明:

```
TYPE
  node=RECORD
         a:integer;
         b:char
       END;
  point=^node
VAR
  p,q:point;
```

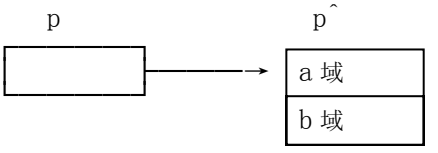


图 10.3

则指针变量 p，q 均可以指向 node 类型的记录。

用 new(p)在内存中开辟了一个 node 记录类型的动态变量后，情况如图 10.3 所示。

这个动态记录中的两个域可分别用

p^.a

和

p^.b

表示。

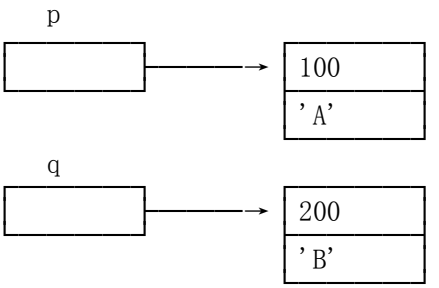


图 10.4

10.1.5 指针变量的操作

一、类型相同的指针变量之间可以相互赋值。

应注意对指针变量赋值和对指针变量所指向的动态变量赋值是不同的。假如一个指针变量已经指向一个动态变量，则给这个指针变量赋值可以迫使它改指向另一个动态变量，但并不改动动态变量的值。

例如，假设指针变量 p，q 已经各指向一个动态数据，如图 10.4 所示。

若执行了 q:=p，则 p 中所存的地址指针送入 q，冲掉了 q 中存放的地址，指针变量 p 和 q 都指向同一个动态数据，结果如图 10.5 表示。

此后，pˆ和 qˆ都可以用来表示图中上面的那个（即 p 原指向的那个）动态数据，而 q 原指向的动态数据已不可访问(假如再没有别的指针变量指向它的话)。

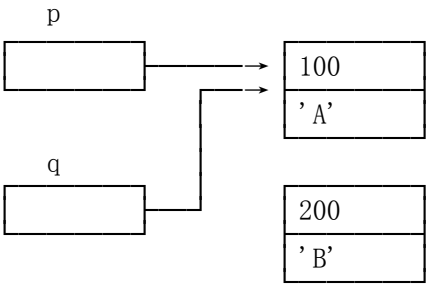


图 10.5

假若不执行 q:=p，而是执行了 qˆ:=pˆ，则动态变量 pˆ的值赋给动态变量 qˆ，改变了 qˆ中原来的值，但 q 中的指针值并没有改变。结果如图 10.6 所示。

从图上可以清楚地看到这两种操作的后果是不同的。虽然它们有一个共同点，就是执行后都能使 pˆ和 qˆ的值相等，所以有些应用问题可以用两种方法实现，但是再多作一些操作就显出不同的效果了。如在图 10.5 的基础上若再给 qˆ赋值，则 pˆ的值同样变化，因为二者是同一个记录变量；而在图 10.6 的基础上若再给 qˆ赋值，则 pˆ的值不会变化，因为二者不是同一个记录变量。所以，务必搞清楚它们的区别。

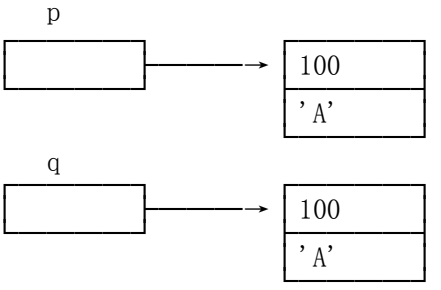


图 10.6

二、给指针变量赋值时，赋值号右边不仅可以是指针变量，也可以是以指针为值的其它形式的表达式，如可以是以指针为返回值的函数。该函数的结果类型必须与赋值号左边的指针变量类型一致。

因 PASCAL 中没有结果为指针的运算符，所以“以指针为值的表达式”只可能是指针变量或以指针为返回值的函数。因用到的很少，为节省文字起见，本书下文对这些情况不再详细解释。本书中凡只提“指针”而不说“指针变量”处均指“以指针为值的表达式”。

三、还可以给指针变量赋以空指针 NIL，例如

```
p:=NIL
```

作此赋值后指针变量 p 就不指向任何动态数据。

四、同一类型的指针可以进行=或<>的比较运算， 用于判断两指针是否指向同一个动态变量。指针还可以和 NIL 进行=或<>的比较，判断该指针是否为空指针。比较的结果都是布尔型的值。

例如

```
p=q          指针 p 和 q 是否指向同一个动态数据
p<>q
p=NIL        指针 p 是否为空指针
p<>NIL
```

记号 NIL 在上述操作中很像一个指针类型的符号常量。 我们说“很像”而不说“是”，是因为常量要有确定的类型，而 NIL 却是“万能的”指针类型。比如假设 s 是^real 型，而 t 是^char 型，则可以有

```
s=NIL
```

和

```
t=NIL
```

却不可以有

```
s=t
```

因为 s 和 t 类型不相容。NIL 只是个特定符号。

10.1.6 程序举例

下面的例题在第四章 4.3 节[例 2]中已经有更简单更好的做法，这里多此一举地将它改用动态变量实现，而且有意将其中的部分操作编成过程，目的是为了使读者了解这些问题的一般原理。实际上 PASCAL 中引入指针主要并不是为了处理这一类问题，而是为了处理后面介绍的动态数据结构。

[例 1] 输入两个整型数，要求按小的在前，大的在后的顺序输出。

```
PROGRAM exswap(input, output);
  TYPE
    point=^integer;
  VAR
    p1,p2:point;
  ..... {编入过程 swap}
  .....
BEGIN
  new(p1);new(p2);
  write('Please input 2 intger:');
  readln(p1^,p2^);
  IF p1^>p2^ THEN swap(p1,p2);
  writeln('Output 2 integer:',p1^:5,p2^:5)
END。
```

程序中利用指针 p1，p2 指示两个动态数据 p1^和 p2^，当 p1^的值大于 p2^的值时，就调用过程 swap(p1,p2)，使得 p1 总指向存放较小数的那个动态变量，而 p2 总指向存放较大数的那个动态变量，最后按顺序输出 p1^，p2^。

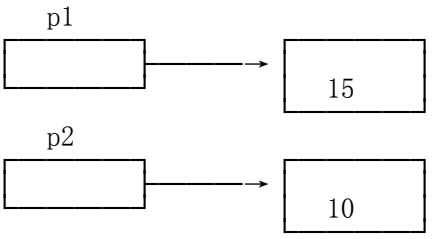
过程 swap 可用不同的方法实现。第一种方案如下：

```
PROCEDURE swap(VAR q1,q2:point);
  VAR
    q:point;
BEGIN
  q:=q1;q1:=q2;q2:=q
```

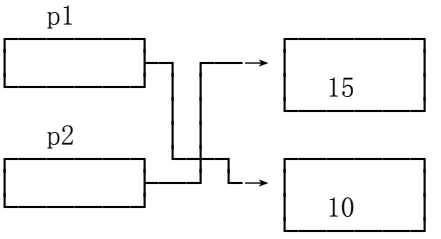
```
END;
```

在这个过程中，q1，q2 是变量参数，当在过程中把 q1 和 q2 中存放的指针值交换时，实际上也就是交换了主程序中的实参变量 p1 和 p2 中存放的指针值，从而使得 p1 总指向存放小数的那个动态变量，而 p2 总指向存放大数的那个动态变量，但这时存放整数的两个动态变量中的值并没有改变。过程 swap 执行前后指针变量变化情况如图 10.7 所示。图中假设输入的数据是

```
15  10
```



(a)执行 swap 之前指针情况



(b)执行 swap 之后指针情况

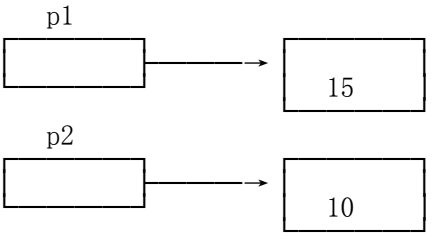
图 10.7　第一种方案

第二种方案如下：

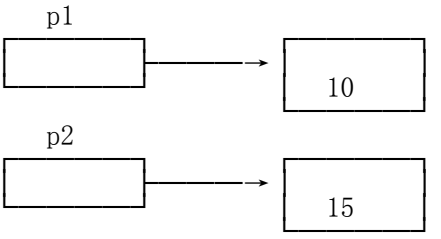
```
PROCEDURE swap(q1，q2:point);
  VAR
    t:integer;
BEGIN
  t:=q1^;q1^:=q2^;q2^:=t
END;
```

这个算法没有交换指针，而是直接交换动态变量的内容。按第二种方案，过程 swap 执行前后指针变量变化情况如图 10.8 所示。

前面图中只画出了主程序中的变量，下面再分析一下调用过程 swap 的情况。第一种方案情况较简单，不详述，请读者自行分析。这里分析第二种方案。



(a)执行 swap 之前指针情况



(b)执行 swap 之后指针情况

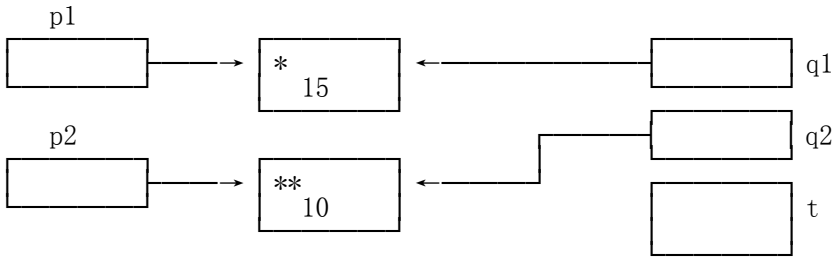
图 10.8　第二种方案

第二种方案的过程 swap 中，q1，q2 是值参数，故调用时要建立临时空间。调用时要将实参 p1 和 p2 的值赋给 q1 和 q2 的空间，因 p1 和 p2 的值是两个动态变量的地址，所以赋值后 q1 和 q2 也就指向这两个动态变量。同时，调用时还要为局部变量 t 开辟空间。刚进入过程 swap 瞬间的情况见图 10.9。图中标*和标**的就是上述的两个动态变量。

然后 swap 过程中执行 t:=q1^时，图中标*的框中的 15 送入变量 t；执行 q1^:=q2^时，图中标**的框中的 10 送入标*的框；执行 q2^:=t 时，t 中的 15 送入标**的框。然后子程序 swap 结束时 q1，q2 和 t 都撤消了，内存中剩下的形势正好如图 10.8(b)。

应注意，这里的 q1 和 q2 是值形参，子程序内如果直接给 q1 和 q2 赋值，是不会带回

主程序的。但是这里是给 q1 和 q2 所指向的变量赋值，却可以带回主程序。第八章归纳子程序向主程序传回数据有三种手段，而这里用的是第四种手段：利用指针给动态变量赋值。



(a)主程序中的变量　　(b)swap 的局部变量

图 10.9　第二种方案刚进入过程 swap 瞬间的情况

对比下面将介绍的解决这个问题的第三种方案，我们会看到，这种手段原理上与变参很相似。

另外我们应注意，前面介绍动态变量，和过去学的静态的普通变量有许多不同点，但是不要误以为它们肯定是不同的数据类型。如图 10.9 中的变量 t 和标*的变量，虽然一是普通变量，一是动态变量，但都是 integer 类型，所以它们之间互相赋值没有问题。 又如下面第三种方案里，普通变量形式的形参可以代表动态变量的实参。

前面两种方案都是以指针作为参数，若直接以整型变量作为参数，可以有第三种方案。第三种方案在过程内不用动态变量和指针，但在主程序中可以将动态变量作为实参来调用过程。第三种方案如下：

首先，将主程序内的

```
IF p1^>p2^ THEN swap(p1,p2);
```

改为

```
IF p1^>p2^ THEN swap(p1^,p2^);
```

其次，将过程 swap 改为

```
PROCEDURE swap(VAR x1,x2:integer);
  VAR
    t:integer;
BEGIN
  t:=x1;x1:=x2;x2:=t
END;
```

注意这里的 x1 和 x2 都是变参。调用时为变参建立变量实体时不开辟空间而是设立指示器指向实参变量，而主程序里实参代入的是动态变量，所以这里 x1 和 x2 都代表主程序中的动态变量。读者可以仿照第八章的办法画出示意图，会发现这里 x1 和 x2 的指示器的作用与图 10.9 中的指针 q1 和 q2 作用相当。

从这里我们看到，利用指向一个变量的指针作参数，可以设计出等效于直接将这个变量作为变参的子程序。顺便说一下，某些其它语言（如 C 语言）没有变参的功能， 但它的指针的功能比 PASCAL 全，PASCAL 中需用变参解决的问题在那里都可以用指针解决。

10.2 简单链表

在 PASCAL 语言中， 利用指针变量和动态变量可以构造出各种复杂的动态数据结构，如线性链表和树结构等。其中最简单的是线性链表，简称链表。本节主要介绍链表结构的最简单的一种——普通单向链表。

10.2.1 简单链表的构成

动态的数据结构都是由一些称作“结点”的元素组成的。在 PASCAL 中通常可以用记录来充当结点。这种记录中若干个域存放要处理的数据，有一个或几个域存放指针，而该指针的定义域类型应该又是这种类型的记录。这样，指针域可以指向又一个结点，从而将多个结点连接成一个结构。简单链表的结点中只需要一个指针域。结点中的指针域又被称作链接域。

假定建立一个存放学生信息的简单链表，在程序中可以有以下数据类型定义：

```
TYPE
  namestr=PACKED ARRAY[1..20] OF char;
  point=^student;
  student=RECORD
            number:integer;
            name:namestr;
            score:integer;
            next:point
          END;
```

其中，student 类型的记录包含四个域：number 域存放学生学号，name 域存放学生姓名，score 域存放学生的成绩，next 是指针域。

注意上述类型定义中，student 记录的 next 域的说明中引用了类型名 point，而 point 类型的定义中又引用了类型名 student。这两个类型定义时互相引用， 如果按照“先定义后引用”的规则，这种定义就无法表达了。为此，PASCAL 的标准对“先定义后引用”的规则规定了一个唯一的例外，即：

新指针类型描述格式里引用定义域类型标识符，允许在该标识符的定义之前，但必须在同一个类型定义部分里。

针对上例也就是说，^student 里引用的 student，可以在 student 的定义之前，只要在同一个 TYPE 部分里就行。所以，上例的定义是合法的。

图 10.10 是一个表示记录学生信息的简单链表的示意图。

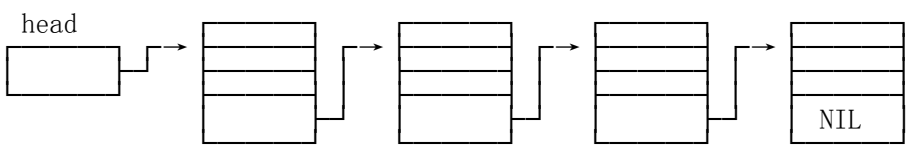


图 10.10 简单链表

图 10.10 中的链表由四个结点组成。每个结点三个数据域，一个指针域。 每个结点的指针域指向下一个结点。因为动态变量只能间接寻址，所以必须至少有一个指针变量指向它的头一个结点，否则整个链表的数据都会“丢失”，无法访问了。图中用了一个 point 类型的指针变量 head 指向第一个结点。我们将这个指针变量称作“链头指针”。从链头指针出发，链中的每个结点都有办法访问到，所以习惯上就把链头指针看作整个链的代表。在表尾结点的指针域中， 由于不需要指向任何结点，所以放入 NIL 作为链表结尾的标志。

按照结点数的多少，链表可以有不同的长度。如果结点数为 0，称作空表， 空表只有一个链头指针，且被赋值为 NIL。

实际应用中链表结点里至少要有一个数据域，这样，连同指针域，每个结点至少要包含两个域。数据域和指针域显然类型是不同的。由于结点内至少包含了两种不同的数据类型，因此，通常用 PASCAL 语言中的记录类型实现，而表中的所有结点都是同一类型的记录。

另外应说明，示意图中所画的次序只是表示各个结点由指针确定的逻辑联系，并不是它们在存储器中存放位置的物理次序。每一个结点分别由 new 过程创建， 它们在内存中的位置由系统自动安排，我们可以不考虑。要想访问它们，唯一的办法是利用指向它们的指针。

从这里，我们可以了解两个特点。一是：当链表中的某个结点的指针域失去下一个结点的地址时，链表就会断开，后面的数据就有可能丢失。二是：如果我们要修改链表的结构，如改变次序，增加或删除结点等，只需要修改指针，而不必在内存中移动各个结点。

我们在本章的开头曾说过，在顺序存储的数据中插入或删除个别数据时不得不平移大量数据，操作效率低。显然，采用链表就可以避免这个缺点。

链表由一个一个结点构成，因此可根据需要动态的增加或删除结点，改变链表的长度，满足记录个数变化的应用情况。数组一类的静态结构就无此能力。

但是，链表结构和数组结构比较起来，数组成分的引用简单，利用数组名和下标即可，而链表则只能顺序访问，引用较复杂。而且数组在内存中是连续存放的，不会发生数据丢失的情况。

10.2.2 简单链表的基本操作

下面举例介绍简单链表的基本操作。下面的例子大都是程序片断，如一个子程序，此时均假定主程序中已经有了上述的类型定义。

[例 1] 以下过程 create 用来建立一个存放学生信息的链表，键盘依次输入各学生的学号、姓名、成绩，当输入学号为-1 时，表示输入结束。

这个例子可以不算基本操作，因为它可以引用后面将介绍的基本操作（如插入操作）组合而成。例如，它的算法可以是：

```
建立空链表 head;
读入学号 n;
WHILE n<>-1 DO
  BEGIN 创建一个动态记录;
        将 n 以及键盘读入的各数据写入该记录中;
```

将该记录插入到 head 表中；
再读入学号 n
END。

如果完全照抄后面介绍的插入操作编入上述算法中当然是可以的。但是这个题目有若干特殊性，可以灵活选择不同的编法，可以编得更简单些。

其一，本题输入数据和插入链表穿插进行，也可以不是将记录填好数据再插入链表，而是先在链中增加一个空结点再填入数据，这样有时可以少用一个指针变量。

其二，本题没有要求插入的位置，可以自由选择，为简单起见，可以每次都插入到表头之前，也可以每次都插入到链尾之后。

其三，某些操作可以简省，如单独在链尾后加入一个结点，必须在其指针域填入 NIL，但是如果相继加入一批结点，就没必要每加入一个赋一次 NIL， 只要全作完后给最后一个赋一次 NIL 就行了。

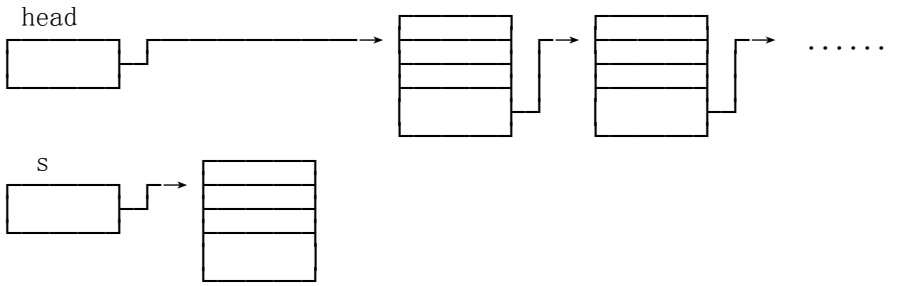
下面给出几种方案。第一个方案是每次都加到表头之前：

```
PROCEDURE create(VAR head:point);
  VAR
    s:point;
    n:integer;
BEGIN
  head:=NIL;                          {建立空表}
  read(n);                            {读入学号 n}
  WHILE n<>-1 DO
    BEGIN
      new(s);                         {创建一个动态记录}
      s^.number:=n;                   {将各数据写入该记录}
      readln(s^.name,s^.score);
      s^.next:=head; head:=s;         {将该记录插入到 head 表中}
      read(n)                         {再读入学号 n}
    END;
  readln;
END;
```

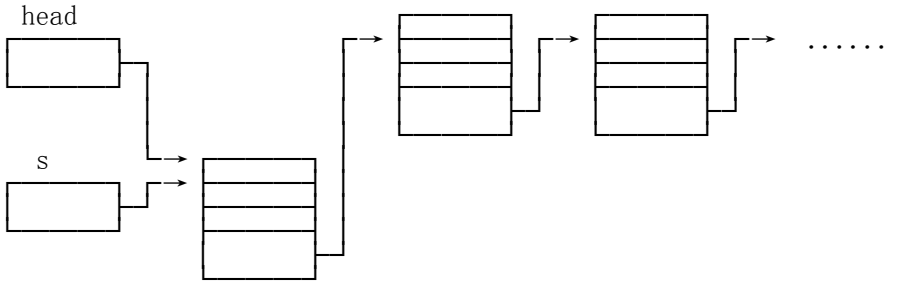
这个方案的程序和前面的提纲一致。程序中的两句：

```
s^.next:=head; head:=s;    {将该记录插入到 head 表中}
```

作用就是将指针 s 所指向的结点插入到 head 所指的链头之前成为新链头。 这两个语句的作用见图 10.11 所示。



(a)插入前的情况



(b)插入后的情况

图 10.11　新结点插入在链头之前

注意这个过程中参数 head 是变量参数。假如用值参，过程一结束链头就丢了。现在用变参，过程结束时链头指针就留在了实参变量中，可以供主程序引用。

另外，这个程序中用到了直接读入字符串的功能，若所用的系统不允许这样作，应适当修改。若在 Turbo PASCAL 下运行，可将

```
namestr=PACKED ARRAY[1..20] OF char;
```

改为

```
namestr=STRING[20];
```

下面的例子同样，不再一一指出。

第二个方案是每次新结点都加到链尾之后。与上一个方案的不同点是：上一个方案中每次可以根据链头指针找到插入的位置，而这个方案中需要设一个工作指针记住刚插入的链尾，以便下次插入时引用。下面程序中的 p 就是这个工作指针。而语句

```
p:=p^.next
```

的作用就是使工作指针前进一步，由原链尾改指新链尾。第二个方案的程序是：

```
PROCEDURE create(VAR head:point);
  VAR
    p:point;
    n:integer;
BEGIN
```

```
head:=NIL;                          {建立空表}
read(n);                            {读入学号 n}
IF n<>-1 THEN
  BEGIN
    new(head); p:=head;             {建立一个新结点}
    p^.number:=n;                   {将各数据写入该结点}
    readln(p^.name,p^.score);
    read(n)                         {再读入学号 n}
    WHILE n<>-1 DO
      BEGIN
        new(p^.next); p:=p^.next;   {增加一个新结点}
        p^.number:=n;               {将各数据写入该结点}
        readln(p^.name,p^.score);
        read(n)                     {再读入学号 n}
      END;
    p^.next:=NIL;                   {链尾的指针域标志}
  END;
readln;
END;
```

这个程序中的

```
new(p^.next);
p:=p^.next;
        {增加一个新结点}
```

两个语句的作用见图 10.12 所示。

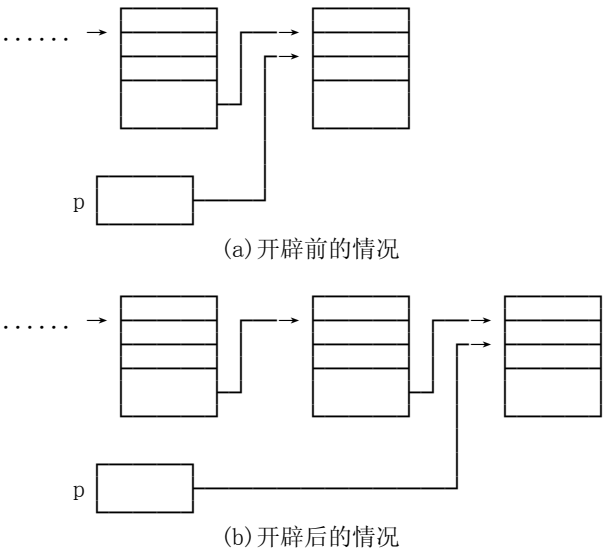


图 10.12　在链尾之后开辟新结点

图中链尾结点的指针域尚未赋以 NIL,这是因为前面已说过的，要等一批结点创建完成后再给最后一个赋以 NIL。

从上面的程序中可以看出，因为第一个结点的处理与后面的其它结点不同，无法编入统一的循环体中，所以程序较长。而有关链表的其它操作如删除、插入等，也存在同样的问题。所以实际应用中，为简化编程，常采用一种多设一个结点的办法。多设的结点作为链头，因为它的数据域中一般不放有用信息（也有时存放些参考信息，例如链表中结点的个数等），有些资料中将它称作“哑记录”或“伪结点”。哑记录不须要处理，所以程序可以简短。采用带哑记录的结构时，即使是空表，也要有一个结点。

下面介绍 create 过程采用带哑记录结构的方案，算是本题的第三种方案。

```
PROCEDURE create(VAR head:point);  {带哑记录的方案}
  VAR
    p:point;
    n:integer;
BEGIN
  new(head); p:=head;              {建立空表，带哑记录}
  read(n);                         {读入学号 n}
  WHILE n<>-1 DO
    BEGIN
      new(p^.next); p:=p^.next;    {增加一个新结点}
      p^.number:=n;                {将各数据写入该结点}
      readln(p^.name,p^.score);
      read(n)                      {再读入学号 n}
    END;
  p^.next:=NIL;                    {链尾的指针域标志}
  readln;
END;
```

在此算法中，头结点由于作为哑记录不作任何处理，程序大大简化。

第二、第三方案中都是采用先在链中增加一个空结点再填入数据的办法。如果不这样，而是采用开辟一个记录填好数据再插入链表的办法，则需要再增加一个工作指针，以便指示新开辟的动态记录。

[例 2] 依次输出链表中所有结点的数据。

访问链表中的结点不能像数组那样通过数组名和下标进行随机访问，而必须从第一个结点开始采用顺序的方法逐项访问。

下面过程 print 中，head 是指向链表的头结点的指针，工作指针 p 从第一个结点开始，直到遇 p=NIL 时表示最后一个结点已经输出完。

```
PROCEDURE print(head:point);
  VAR
    p:point;
BEGIN
  p:=head;
  WHILE p<>NIL DO
   BEGIN
     writeln(p^.number,'    ',p^.name,'    ',p^.score);
     p:=p^.next;
   END
END;
```

从本例中再次看到工作指针所起的作用，例中语句

```
p:=p^.next
```

的作用就是使指针所指前进一步。这是有关链表的程序中很常用的一个操作，从下面的其它例中就可以看到。但是这种链的工作指针只能前进，不能后退，所以有些算法中需要设两个或更多工作指针才能满足需要。

上面是针对不采用哑记录的结构的程序。如果是采用哑记录的结构，则只要将程序中第一句

```
p:=head
```

改为

```
p:=head^.next
```

就行了。

[例 3] 查找链中结点。

下面的过程 search 是顺序查找 head 链中学号等于给定的 x 值的结点。 查找的结果通过变参 p 和 q 送回。其中 p 指向找到的结点，而 q 指向它的前一个结点（下面称作前驱结点)。如果链表中不存在这样的结点，则结果 p 为 NIL，而 q 指向链尾结点。

本例是针对采用哑记录的结构的程序。如果不采用哑记录的结构，则因为有可能查找到第一个结点，而第一个结点不存在前驱结点，问题更加复杂，这里不详述。

顺便解释一下为什么结果要给出两个指针。这是因为工作指针不能后退，如果子程序结果只提供一个指针指向找到的结点，回到主程序后有时可能不够用。例如，主程序接着若要删掉该结点，则从下面[例 4]中就可以看到，没有前驱指针是不行的。 假如只提供前驱指针不提供当前指针虽然也行，但不如现在这样方便。

search 过程的算法可以描述如下：

```
PROCEDURE search(head:point; x:integer; VAR p,q:point);
BEGIN
  q:=head; p:=q^.next;
  WHILE (p<>NIL) AND (p^.number<>x) DO
    BEGIN q:=p; p:=p^.next END
END;
```

上面的 q:=p; p:=p^.next 作用就是“工作指针 p，q 各前进一步”。

不过，上述程序直接使用起来尚有一些问题。式子

```
(p<>NIL) AND (p^.number<>x)
```

中，AND 前后的两个括号都运算完才能得出判断结果。可是若 p=NIL，则后一括号中写 p^ 就会出错。有些系统遇这种错误并不报错，程序最后结果还是对的。但有些系统一出错操作就会打断。我们希望若 p=NIL 就不再计算后一括号，直接退出循环，以免因出错而打断程序。

解决这个问题有多种办法。一个办法是采用 GOTO 语句的非结构化跳转，程序清单如下：

```
PROCEDURE search(head:point; x:integer; VAR p,q:point);
  LABEL 1;
```

```
BEGIN
  q:=head; p:=q^.next;
  WHILE (p<>NIL) DO
    BEGIN
      IF (p^.number=x) THEN GOTO 1;
      q:=p; p:=p^.next
    END;
1:
END;
```

另一个办法不用 GOTO 语句，但增加一个标志 b，程序清单如下：

```
PROCEDURE search(head:point; x:integer; VAR p,q:point);
  VAR b:boolean;
BEGIN
  q:=head; p:=q^.next; b:=true;
  WHILE (p<>NIL) AND b DO
    BEGIN
      b:= p^.number<>x ;
      IF b THEN  BEGIN q:=p; p:=p^.next END
    END
END;
```

还有一种办法是提前一次结束循环，在循环外再判断一次（此法要求不得是空表）：

```
PROCEDURE search(head:point; x:integer; VAR p,q:point);
BEGIN
  q:=head; p:=q^.next;
  WHILE (p.next<>NIL) AND (p^.number<>x) DO
    BEGIN q:=p; p:=p^.next END;
  IF p^.number<>x THEN BEGIN q:=p; p:=NIL END
END;
```

这几种办法都可以。

[例 4] 删除链表中指定结点。

删除链表中不用的结点，并释放其内存空间，也是链表的基本操作之一。

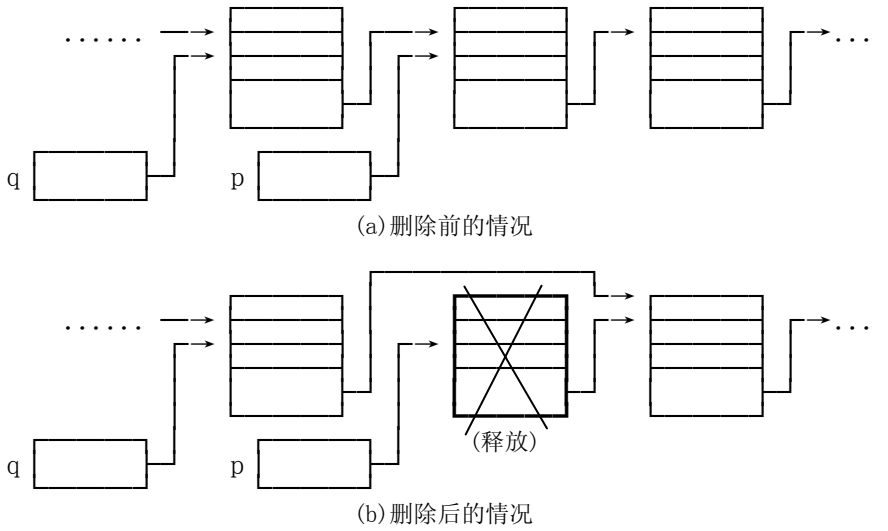
下面的过程 del 删除链表中的结点。本过程调用时代入的参数中，p 应指向要删的结点，q 应指向其前驱结点。若 p=NIL，本过程为空操作。

```
PROCEDURE del(p,q:point);
BEGIN
  IF p<>NIL THEN
    BEGIN
      q^.next:=p^.next;
```

```
            dispose(p)
        END
    END;
```

删除的过程见图10.13示意。

假如要删除学号为x的结点，则只要将[例3]和[例4]合起来就行了。可以在主程序中先后调用两个过程，也可以将这两个算法合编到一起，请读者自己考虑。

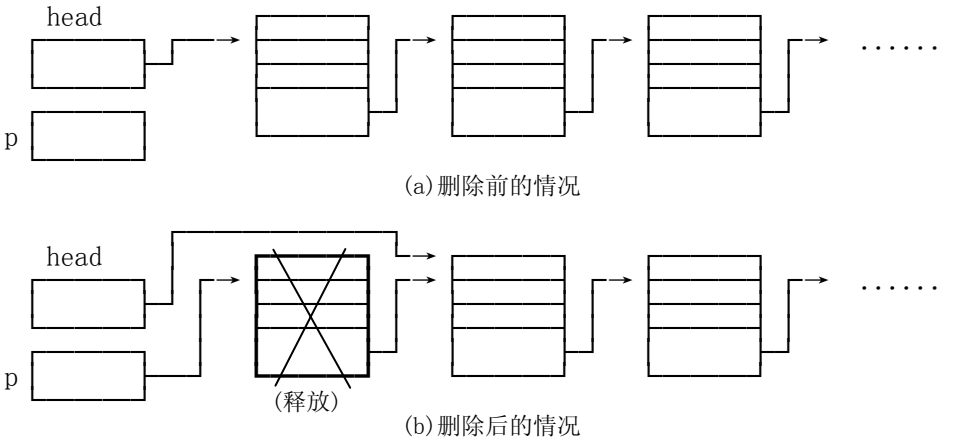


(a)删除前的情况

(b)删除后的情况

图 10.13　删除链中结点

[例 5] 删除链表头结点。关于删除的算法，对于采用哑记录的结构，[例 4] 的算法就已经够用了。但对于不采用哑记录的结构，有时会要删除头结点，而头结点没有前驱，就不能用[例 4]的算法了。下面给出一个删除头结点的过程。对于空表，这个过程等效于空操作。



(a)删除前的情况

(b)删除后的情况

图 10.14　删除链头结点

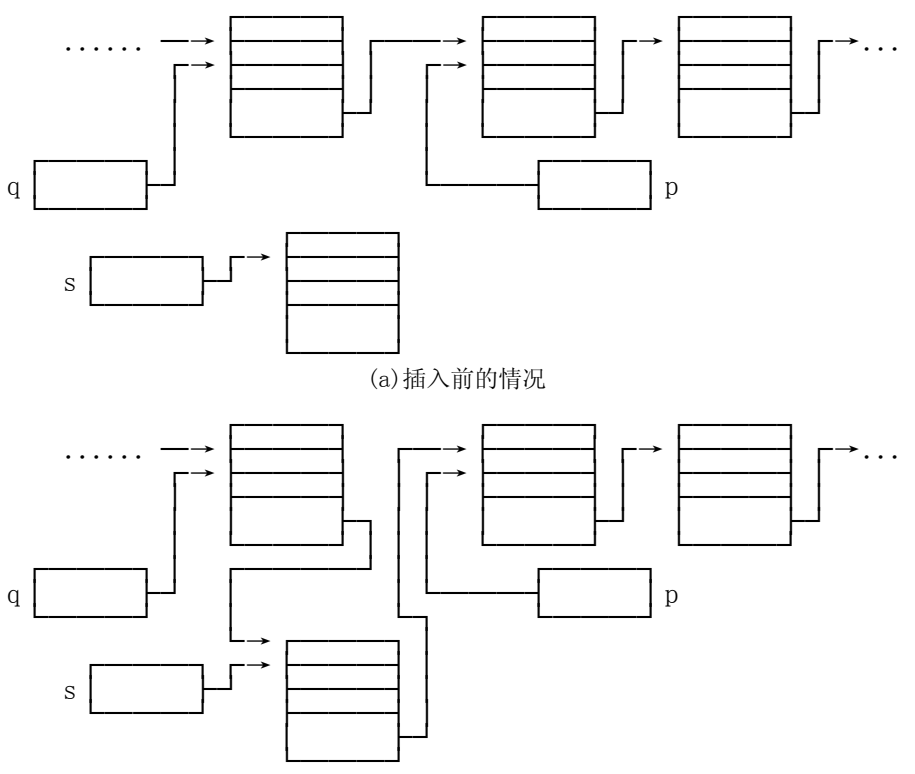
```
PROCEDURE delhead(VAR head:point);
 VAR
```

```
  p:point;
BEGIN
  IF head<>NIL THEN
    BEGIN
      p:=head;
      head:=head^.next;
      dispose(p)
    END
END;
```

删除的过程见图 10.14 示意。

[例 6] 在链表中插入一个结点。

下面的过程 insert 在链表中插入一个结点。本过程调用时代入的参数中：s 应指向待插入的结点；q 和 p 是两个工作指针，若指向表中相邻的两个结点，则本过程将 s 所指的结点插入到 p 所指的结点之前，q 所指的结点之后；若 q 指向链尾，而 p=NIL，则本过程将 s 所指的结点插入到链尾之后成为新链尾。



(a)插入前的情况

(b)插入后的情况

图 10.15　新结点插入链中

```
PROCEDURE insert(s,p,q:point);
BEGIN
  s^.next:=p;
  q^.next:=s
```

```
END;
```

插入的过程见图 10.15 示意。

这个题目若参数中只给出 q 不给出 p，也是可以的，程序如下

```
PROCEDURE insert(s,q:point);
BEGIN
  s^.next:=q^.next;
  q^.next:=s
END;
```

这个算法中没有包括插入到链头之前的情况。若采用带哑记录的结构，当然不必考虑插入到链头之前的情况。若不采用带哑记录的结构，当有必要插入到链头之前时，采用[例 1]中方案一里的两个语句就行了。

上面各例中的每一个算法都写成了一个过程的形式，我们这样写是为了便于读者各个击破，分别掌握其要领。如果遇到一个综合性的题目，需要引用上述算法，完全可以用过程语句来分别调用。

但是上面的算法有些非常简短，如[例 6]的算法只有两句， 单独编成子程序往往不如直接将其算法写入主程序中，这样整个程序可以短些。只要上述每个子程序的道理掌握了，这样混编在一起的综合程序也不难掌握。下面是一个这样的例子。

[例 7] 键盘依次输入各学生的学号、姓名、成绩，当输入学号为-1 时，表示输入结束。然后机器依分数由高到低的顺序输出全部数据。

可以采用边输入边插入链表的办法实现排序。即每输入一个学生的数据，就在链表中查找到第一个分数低于它的结点的位置，插入到该位置的前面；若没有分数低于它的结点，则插入到链尾后边。

整个程序的算法是：

```
建立空链表 head;
读入学号 n;
WHILE n<>-1 DO
  BEGIN
    创建一个动态记录;
    将 n 以及键盘读入的各数据写入该记录中;
    检索 head 链表找到该插入的位置;
    将该记录按找到的位置插入到表中;
    再读入学号 n
  END;
依次输出链表中所有结点的数据。
```

这里用到的各个基本操作中，只有“检索 head 链表找到该插入的位置”这一条前面没有介绍到。但是前面的[例 3]和它很相似，[例 3]是检索学号等于已知数的结点，这里是检索成绩小于已知数的结点。假设 x 是刚读入的分数，则只要将[例 3]程序中的

```
(p^.number<>x)
```

改成

```
(p^.score>=x)
```

将

```
(p^.number=x)
```

改成

```
(p^.score<x)
```

就行了。注意这里的 x 就是下面程序里的 s^.score。这样所得的 p 和 q 恰好符合我们需要，接下来的插入可以采用[例 6]的算法。

为了处理简单，我们采用带哑记录的结构。

```
PROGRAM sortout(input,output);
  LABEL 1;
  TYPE
    namestr=PACKED ARRAY[1..20] OF char;
    point=^student;
    student=RECORD
              number:integer;
              name:namestr;
              score:integer;
              next:point
            END;
  VAR
    head,p,q,s:point;
    n:integer;
BEGIN
  new(head); head^.next:=NIL;    {建立空表，带哑记录}
  read(n);                       {读入学号 n}
  WHILE n<>-1 DO
    BEGIN
      new(s);                    {创建一个动态记录}
      s^.number:=n;              {将各数据写入该记录}
      readln(s^.name,s^.score);
      q:=head; p:=q^.next;       {检索 head 链表找到该插入的位置}
      WHILE (p<>NIL) DO
        BEGIN
          IF (p^.score<s^.score) THEN GOTO 1;
          q:=p; p:=p^.next
        END;
    1:s^.next:=p; q^.next:=s;    {将该记录按找到的位置插入到表中}
```

```
    read(n)                          {再读入学号 n}
  END;
p:=headˆ.next;                       {依次输出链表中所有结点的数据}
WHILE p<>NIL DO
  BEGIN
    writeln(pˆ.number,'    ',pˆ.name,'    ',pˆ.score);
    p:=pˆ.next;
  END
END.
```

这个程序中没有采用子程序，这是因为所用到的基本操作都只引用一次，而且基本操作算法都较简短，不用子程序整个编在一起程序较短。但用子程序也有用子程序的好处，读者可以自行练习将这个程序改写成采用子程序的形式。希望读者能够熟悉采用子程序的编程方式。

10.3 其它结构的线性链表

线性链表除上节所述的简单链表（普通单向链）外，还有其它的形式。本节介绍常见的循环链表和双向链表。

10.3.1 循环链表

简单链表的尾结点的指针域指向表头结点时,简单链表就构成为循环链表。图 10.16 表示一循环链表。

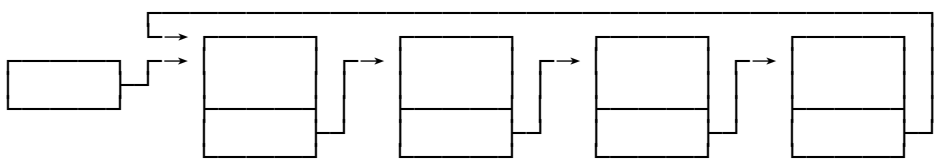


图 10.16 循环链表

与前面学过的普通链表相比，循环链表有三个特点：

其一，普通链表中，当工作指针指向链尾结点时，若再“进一步”，工作指针的值就成为 NIL；而循环链表中，当工作指针指向链尾结点时，若再“进一步”， 工作指针就指向链头结点。在某些应用问题中，这一特点是有用的。

其二，从头结点开始扫描整个链表时，普通链表以工作指针为 NIL 确定“已扫描完”，而循环链表以工作指针等于头指针确定“已扫描完”。

其三，循环链表不管从哪个结点出发，都能扫描整个链表，所以由指向哪个结点的指针代表这个链不是绝对的，可以根据应用的需要灵活安排。

[例 1] 利用循环链表实现猴子选大王的算法。

从 n 个围成一圈的猴子中选举大王。设这些猴子的编号依次是 1～n。 选举办法是从第一号猴子开始连续报数，凡所报数能被 m 除尽者退出圈外，直到圈内只剩下一只猴子为

止， 此猴即为大王。n 和 m 由键盘输入。

这个题目在第七章已经用布尔数组作过，现在改用循环链表作。两种方法相比，有两点不同：一是现在采用动态数据结构，n 的大小不受数组大小的限制； 二是用数组时直接判断数组中剩几只猴子很不方便，故只能用另外计数的办法控制结束循环，而现在可以不用计数，直接以圈中只剩一个结点为循环结束条件。

程序清单如下：

```
PROGRAM monkey(input,output);
  {猴子选大王程序}
  TYPE
   point=^body;              {指针类型}
   body=RECORD               {结点类型}
          num:integer;           {猴子号}
          next:point             {指针域}
        END;
  VAR
    head:point;
    n,m:integer;
  PROCEDURE creating(VAR head:point; n:integer);
    {建立循环链表}
    VAR
      p:point;
      i:integer;
  BEGIN
    new(p);head:=p;p^.num:=1;     {第一个结点}
    FOR i:=2 TO n DO              {第 2～n 个结点}
      BEGIN
        new(p^.next); p:=p^.next;
        p^.num:=i;
      END;
    p^.next:=head;                {闭环}
  END; { creating }
  PROCEDURE king(VAR head:point; m:integer);
    {筛选大王，最后 head 指向大王}
    VAR
      p,q:point;
      t:integer;  {报数，达 m 时改为 0}
  BEGIN
    p:=head;t:=1;q:=p;
```

```
    REPEAT
      p:=q^.next; t:=t+1;
      IF t=m          {若 t 达 m 则删掉 p 所指结点}
        THEN BEGIN t:=0; q^.next:=p^.next; dispose(p) END
        ELSE q:=p
    UNTIL q=q^.next;  {只剩最后一个结点? }
    head:=q;
  END; { king }
BEGIN              {  主程序  }
  write('Number of monkeys (n) : '); readln(n);
  write('m : '); readln(m);
  creating(head,n);
  king(head,m);
  writeln('The king is No. ',head^.num);
END .
```

以下是程序的两次运行结果：

```
Number of monkeys (n) : 15
m : 7
The king is No. 5

Number of monkeys (n) : 55
m : 9
The king is No. 42
```

10.3.2 双向链表

前面介绍的单向的链表中，工作指针只能进不能退，这一点给许多应用带来不便。假如链表中的每个结点有两个指针域，一个指向前一个结点，另一个指向后一个结点，即构成了双向链表。对双向链表操作时，工作指针既能进又能退。扫描双向链既能从头结点开

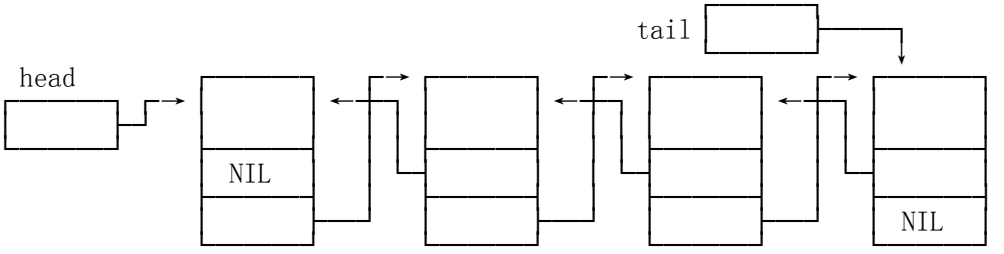


图 10.17　双向链表

始也能从尾结点开始，可以对链表进行双向操作。图 10.17 是一个双向链表的示意图。

图中指针 tail 指向尾结点，通常称为尾指针。可以利用头指针和尾指针对双向链表进行双向操作。图 10.17 中的结点可以用以下记录类型实现：

```
TYPE
  point=^node;
  node=RECORD
         data:integer;      {数据域根据需要也可以是别的类型}
         priou,next:point; {priou 是逆向指针域，next 是正向指针域}
       END;
```

现就以图 10.17 的双向链表为例给出双向链表的基本操作。

假定 head 为链表的头指针，tail 为链表的尾指针，p 和 s 都是 point 类型指针。

1.将 s 所指结点作为双向链表中的第一个结点：

```
head:=s; s^.priou:=NIL; s^.next:=NIL; tail:=s;
```

2.把 s 所指结点插入到尾结点之后作为新的尾结点：

```
tail^.next:=s; s^.priou:=tail; s^.next:=NIL; tail:=s;
```

3.把 s 所指结点插入到头结点之前作为新的头结点：

```
head^.priou:=s; s^.next:=head; s^.priou:=NIL; head:=s;
```

4.把 s 所指结点插在链表中由 p 所指结点（非头结点）之前：

```
s^.next:=p; s^.priou:=p^.priou;
p^.priou^.next:=s; p^.priou:=s;
```

5.把 s 所指结点插在链表中由 p 所指结点（非尾结点）之后：

```
s^.priou:=p; s^.next:=p^.next;
p^.next^.priou:=s; p^.next:=s;
```

6.从链表中删除 p 所指结点（非头结点，非尾结点）：

```
p^.priou^.next:=p^.next;
p^.next^.priou:=p^.priou;
dispose(p);
```

7.删除头结点（设原有结点不止一个）：

```
head:=head^.next; dispose(head^.priou);
head^.priou:=NIL;
```

8.删除尾结点（设原有结点不止一个）：

```
tail:=tail^.priou; dispose(tail^.next);
tail^.next:=NIL;
```

10.3.3 双向循环链表

在前面介绍的双向的普通链表中，若将链头结点的逆向指针域指向链尾结点，将链尾结点的正向指针域指向链头结点，就构成了双向循环链表。图 10.18 是一个双向循环链表的示意图。

双向循环链表操作比上述双向普通链表更简单灵活些。首先，从上面介绍的双向普通链表的操作中可以看到，其头尾结点操作特殊，程序中经常需要分不同情况处理；而循环链表则可以看作“无头无尾”的链，操作可以统一，下面就可以看到其基本操作少了许多种。其次，不必经常保留头尾两个指针，只要一个就够用了。

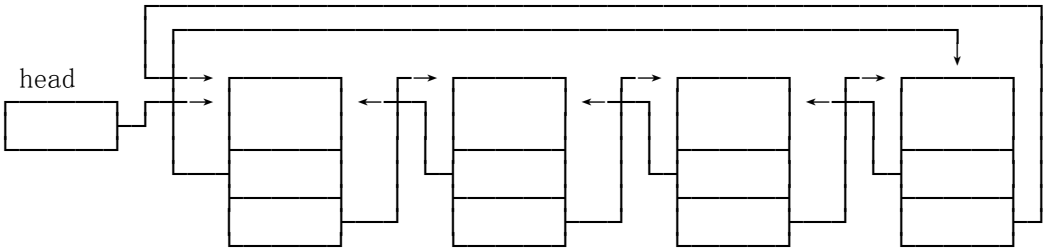


图 10.18　双向循环链表

双向循环链表的数据类型说明和双向普通链表可以一样。现在就按 10.3.2 里中的类型说明，假定 head 为链表的头指针，给出双向循环链表的基本操作。

1.将 s 所指结点作为双向链表中的第一个结点：

```
head:=s; s^.priou:=s; s^.next:=s;
```

2.把 s 所指结点插在链表中由 p 所指结点之前：

```
s^.next:=p; s^.priou:=p^.priou;
p^.priou^.next:=s; p^.priou:=s;
```

3.把 s 所指结点插在链表中由 p 所指结点之后：

```
s^.priou:=p; s^.next:=p^.next;
p^.next^.priou:=s; p^.next:=s;
```

4.从链表中删除 p 所指结点（设原有结点不止一个）：

```
p^.priou^.next:=p^.next;
p^.next^.priou:=p^.priou;
dispose(p);
```

其中除第一个外，都和双向普通链表中的相应操作一样。唯一需要另外考虑的问题是：如果删掉的结点是头指针所指的结点，需要使头指针改指向另一个适宜的结点。

10.4　返回指针值的函数

我们说过，函数的结果类型不仅可以是简单类型，还可以是指针类型。这里举一个实用的例子。

[例 1] 以下函数是用顺序查找法从一个带头结点（哑记录）的双向循环链表中查找已知关键字的结点。参数 h 是链头指针，k 是待查关键字。若查找成功，函数返回指向找到的结点的指针。若查找失败（不存在关键字等于 k 的结点），函数返回空指针。

```
TYPE
  pointer=^node;
  node=RECORD
         key:        integer;
         data1,data2:real;
         left,right: pointer
       END;
```

```
FUNCTION srch(h:pointer; k:integer):pointer;
  VAR
    q:pointer;
BEGIN
  q:=h;                          {工作指针初始化}
  REPEAT q:=q^.right             {工作指针右移一步}
  UNTIL (q^.key=k) OR (q=h);     {已找到或已查到头}
  IF q=h THEN srch:=NIL
         ELSE srch:=q
END;
```

上面例中，结点里的 key 域是作为关键字的数据域，data1 及 data2 是另两个数据域。例如可以 key 是学号，data1 及 data2 是两门成绩。left 及 right 是左指针及右指针（即逆向指针及正向指针）。头指针所指的头结点是哑记录，位于第一个有数据的结点的左边，也可以看作是在最后一个有数据的结点的右边（因为是循环链）。

有了上述函数，假若程序中已经建立了如上的链表，头指针是 head，设 x 是整数， p 是 pointer 型指针变量，则要想输出学号为 x 的学生的学号和成绩，可如下操作：

```
p:=srch(head,x);
IF p=NIL
  THEN writeln('Not found!')
  ELSE writeln(p^.key, p^.data1:8:1, p^.data2:8:1)
```

要想删除学号为 x 的学生的记录，可如下操作：

```
p:=srch(head,x);
IF p<>NIL THEN
  BEGIN
    p^.left^.right:=p^.right;
    p^.right^.left:=p^.left;
    dispose(p)
  END
```

当然，这些问题不用函数，用带变参的过程同样可以解决。但可返回指针值的函数给我们增加了一种可以选择的手段。

习题

10.1 改写 10.2 节[例 7]的程序，采用子程序使其功能层次分明。

10.2 用单向普通链表设计一个运动员成绩登记程序。 每个运动员的信息包括运动员号及成绩。假设运动员号为整数，不一定连续，但无重复，成绩为实数。程序的功能要求如下：

启动后显示主选单：

1．录入　　2．查询　　3．删除
4．增添　　5．修改　　6．列表
7．全清　　其它数字结束程序
请输入选单数字：__

若所用系统不能用汉字也可改用英文或其它简单形式。输入一个数字后即进入该选项的操作。该选项的操作结束时又回到主选单重复操作。各选项功能如下：

录入：键盘依次输入各运动员的运动员号及成绩，当输入运动员号为-1 时，表示这一批录入结束，程序将其编入链表之中。注意三点：一是主选单中“录入”一项并非只能选取一次，故设计时要照顾链中可能已有数据的情况。二是为便于将来列表输出，链表中要保持成绩由大到小的次序。三是每输入一个运动员的信息时机器自动检查运动员号是否和链中已有结点重复，如重复，给出提示要求重新输入。

查询：键盘输入运动员号，机器显示出成绩。若链中无此运动员号，显示“无此运动员号”。

删除：键盘输入运动员号，机器从链中删除相应结点。若链中无此运动员号，显示“无此运动员号”。

增添：功能与“录入”同，只是录入一个运动员后即自动结束，不必再输入-1。

修改：键盘输入一个运动员的运动员号和成绩，该运动员号应是链中已有的，将其成绩修改为新打入的成绩。为保持成绩的次序，可设计成先删除再增添。

列表：将链中全部数据按成绩由大到小的次序输出。

全清：将链中数据结点删除完，成为空表。

10.3 键盘输入若干个整数，机器输出一个方阵。 方阵是如下例所示的那样将输入序列循环移位构成的。如输入是

7　4　3　9　0　5　8

则输出应为

7　4　3　9　0　5　8
4　3　9　0　5　8　7
3　9　0　5　8　7　4
9　0　5　8　7　4　3
0　5　8　7　4　3　9
5　8　7　4　3　9　0
8　7　4　3　9　0　5

试用循环链表实现此程序。

第十一章　文件

11.1　文件的概念

文件一词原来是指外存储器中信息的一种组织形式。现在的计算机应用中，文件概念的外延已经扩大，也包括不在外存储器中的某些信息。如在计算机系统中，也把一些输入，输出设备如打印机，键盘，显示器等外部设备作为文件对待。但文件主要地是指外存储器中的文件。

前面所学的各种形式的 PASCAL 数据，其生存期都不超过程序的运行期限，程序运行一结束，这些数据就都“不存在”了。但是实践中常常遇到需要长期保存数据的情况。例如，有时程序某次运行产生的数据，要保留到下次运行时使用；或者，一个程序产生的数据要给别的程序使用；等等。这些场合就需要将数据保存在外存储器。还有，程序的中间数据过多，内存不能容纳的时候，也需要使用外存储器。PASCAL 使用外存储器就是采用文件的形式实现的。

在大多数应用中，文件一词可以这样定义：文件是可以按名字访问的一批相关信息的集合。这里所谓的相关信息的集合，通常是指由同一类型的数据组成的序列。

在 PASCAL 中，文件是又一种构造类型数据。既然文件是可以按名字访问的数据，那么文件也就可以算是一种变量。

但是，如果把文件仅仅作为一种构造类型的普通变量，那么本书的前面已经说过，一般变量在第一次被赋值之前其值是“未定义的”，而且，程序结束时变量的值也就不存在了。这样作显然不符合我们上面所说的目的。所以，大多数 PASCAL 应用程序中引入的文件不是作为普通的变量，而是作为程序的参数，办法是将文件名写进程序首部括号内的参数表中。就好像子程序首部的形式参数是子程序与主程序交接数据的窗口一样，程序首部的程序参数是程序与外部环境交接数据的窗口。这种作为程序参数的文件一般称作外部文件。对于外部文件，可以认为它在程序运行开始前就已经可能赋过值了，而且它在程序运行结束后值仍然存在。

相应地，仅仅作为一种普通变量引入的文件称作内部文件。内部文件在程序结束时就撤销。

不管是内部还是外部文件，其文件名都必须在变量说明部分中说明。

因内部文件用得不多，所以本书后面的举例都只讲外部文件。

在前面所学的内容中，程序首部经常使用两个参数 input 和 output。这是 PASCAL 的两个预定义的文件，分别表示标准输入设备和标准输出设备，一般代表计算机的键盘输入和显示终端输出。因此，在程序中，只要使用了输入和输出语句，就必须在程序首部写入这两个参数。

文件的每个成分类型都一样，这一点和数组有些类似。但它和数组有重要的不同点。首先，它的序列长度（即成分的个数）在程序运行中是可变的，最短可以为 0 个（此时成

为空文件），最长只受设备容量的限制。其次，它不能像数组那样根据下标来直接访问其成分，必须按照一定的手续来访问。

按照对文件成分的访问方式，文件通常可分为顺序文件和随机存取文件（又称直接访问文件）。

顺序文件有以下特点：

1.程序把数据输出到文件中去时，总是从文件的起始位置开始依次存放，不能从文件中间某一位置开始存放数据。而且，如果文件里原来有内容，则一旦开始重新写入，通常原有内容就全部清除了。某些系统提供了不清除原有信息而是接在原有内容尾部增写信息的功能，但标准 PASCAL 无此功能。

2.当从文件中输入数据到内存时，也必须从起始位置开始依次输入，按数据当初存到文件上的顺序一个接一个地读入内存。

3.同一文件的输入和输出不能交叉进行。

随机存取文件则可以对文件中任一成分随机读写而不必依次进行。

标准 PASCAL 只支持顺序文件，现在常用的某些 PASCAL 系统如 Turbo PASCAL 也支持随机存取文件。但关于随机文件的操作尚未形成标准，所以本书只介绍顺序文件。实际应用中，如果要用随机文件，请参阅所用具体 PASCAL 系统的说明资料。

PASCAL 文件的成分，按规定可以是前面学过的任何一种类型的数据。成分的类型不同，可以有不同类型的文件。所以“文件”并不是一个类型，而是一类类型。用户可以根据需要来定义文件类型。

作为文件的成分类型，有两种情况很常见。

一种是以记录作为文件的成分类型。这种文件常用来构成数据库，应用非常广泛。如果把一个记录看作表格中的一行，其中的一个域看作一行中的一个栏目，则整个文件就是一个表格。

另一种是以字符作为文件的成分类型，整个文件相当于一个很长的字符串。这种文件适合于表示文字信息，习惯上称作正文文件，或称文本文件，也有广泛的应用。

由于正文文件应用广泛，所以 PASCAL 中提供了一个预定义类型标识符 text 来表示正文文件。不过 PASCAL 中的正文文件不仅是以字符作为成分的文件，还另外增加了许多规定，详见下文介绍。预定义文件 input 及 output 都是正文文件。

习惯上，人们把除了正文文件以外的各种类型的文件叫做“二进制文件”。

人们这样称呼，并不是说正文文件不以二进制存储。通常正文文件中每个字符仍然是按二进制代码如 ASCII 码存放，每个字符占一个字节。

这样称呼，是因为通常正文文件也可以存入数值性数据，但不是直接将这些数值的二进制代码写入文件，而是将这些数值换算为十进制，将十进制的每一位用数字字符表示，连同十进制表示所需的各种字符（如小数点、负号、分隔的空格等）构成一个字符序列，将序列中的字符依次写入正文文件。而其它类型的文件写入数值时不经过这种进位制转换，而是直接将数据在内存中原来存放的二进制形式搬到文件上去。

许多工具软件具有显示正文文件内容的功能，如 PC-DOS 中的 type 命令，它们在显示时是将文件中每个字节显示为一个字符。如果不是正文文件，而用这些功能来显示，则因

为文件中原来不是字符的ASCII码,现在强行将其每个字节当做一个字符的ASCII码来看，只能看到些杂乱无章的字符。

大多数的编译系统中，PASCAL源程序都是按正文文件的形式存放的，所以我们才可以有如第六章6.2节[例2]那样的例题。

本章先介绍一般的二进制文件，然后再介绍正文文件所特有的规定。

11.2 一般二进制文件

11.2.1 一般文件类型及文件类型的变量

前面已经说过，一般二进制文件需要由用户自定义其类型。新文件类型的描述格式为

```
FILE OF 成分类型
```

其中“成分类型”应是一个类型表记符。例如

```
TYPE
  sturec=FILE OF RECORD
                   stunum:integer;
                   score1,score2:real
                 END;
VAR
  fa,fb:sturec;
  fc,fd:FILE OF integer;
```

其中定义的文件变量fa和fb都是以记录为成分的文件，fc和fd都是以整数为成分的文件。

按PASCAL的标准，以前学过的任何类型都可以作为文件的成分类型，但文件类型不可以又作为文件类型的成分。更严格些说，以文件为成分的其它结构类型，或以文件为成分的成分……的结构类型，等，也都不可以作为文件的成分。不过这些情况实用中一般遇不到，可以不去考虑。

在程序中使用文件，除必须将文件名说明为文件类型的变量以外，如果是外部文件，还必须在程序首部的程序参数表中列入该文件名。例如

```
PROGRAM test(f1,f2,input,output);
  VAR
    f1,f2:FILE OF real;
  ......
```

11.2.2 文件操作的一般步骤

如上所述建立了文件变量，程序中就可以对它进行操作。

第六章讲赋值相容的时候已经说过，文件类型即使是同一类型也不赋值相容。所以对文件变量不能直接整体赋值。如下的赋值语句

```
f1:=f2
```

是不合法的。PASCAL文件的读写只能以文件成分为单位逐个进行。

由于文件的读写操作不可交叉进行，所以一遍操作之前必须确定操作是“读”方式还是“写”方式。

由于文件的读写是顺序进行的，可以想象系统内部有一个指向文件当前位置的指示器，我们将它称作文件指针。文件指针由系统管理，随着读写的进行，自动调整，总是指向下一个将要访问的文件成分。文件的最后一个成分访问过以后，文件指针指向文件结束位置。

因此，在读写文件之前，要先作一个“打开文件”的操作。打开文件主要包括两个功能：一是确定操作方式是“读”还是“写”，二是使文件指针指向文件开头。另外，若是“写”方式，打开时文件的原有内容自动被清除掉。同时，系统内部还要完成为读写作准备的初始化工作。

打开以后，就可以对该文件进行输入输出操作。如果是“写”方式，可以用下述的输出操作（如 write 过程等）顺序地写入信息以生成整个文件。如果是“读”方式， 则可以用下述的输入操作（如 read 过程等）顺序地从文件读取信息。

一遍操作后，如果需要，还可以重新打开。例如，第一遍若是“写”方式，写完以后第二遍又打开成为“读”方式，则可以从头读出第一遍写入文件的内容。

另外，我们注意到文件不可能满足赋值相容的要求，所以如果有一个子程序的参数是文件，那么它不可以是赋值参数，只可以是变量参数。

11.2.3 和文件操作有关的预定义过程和函数

为了对文件进行操作，PASCAL 系统预定义了一些过程和函数，可以调用它们完成所需要的操作。

1.文件结束函数 eof(f)

参数 f 是文件变量，函数值是布尔类型。当文件中最后一个成分已访问完， 文件指针指向文件结束位置时，eof(f)的值为 true，否则为 false。在“读”方式中利用这个函数， 可以判断文件的内容是否已经读完。在写方式中 eof(f)总是 true。

eof 是英文 end of file（文件结束）的缩写。

2.按“读”方式打开文件的过程 reset(f)

参数 f 是文件变量。本过程将文件 f 确定为“读”方式，为读作准备，并使文件指针指向文件开头位置。要求在打开之前这个文件必须已经存在，否则算是错误。

3.按“写”方式打开文件的过程 rewrite(f)

参数 f 是文件变量。本过程将文件 f 确定为“写”方式，为写作准备，并使文件指针指向文件开头位置。若在打开之前这个文件已经存在，则将它清除为空文件。若在打开之前这个文件不存在，则建立一个空文件。

4.输入过程 read(f,v)

参数 f 是文件变量，v 是普通变量。本过程的功能是从文件 f 中读取当前成分的值赋给变量 v，文件指针自动前移。只有在“读”方式中且 eof(f)为 false 时才可以执行本过程。要求文件中当前成分的值对变量 v 赋值相容。

如果程序中对同一个文件有连续多个 read 语句如

```
read(f,v1); read(f,v2); read(f,v3); ...
```

可以合写为一句

```
read(f,v1,v2,v3,...)
```

5.输出过程 write(f,e)

参数 f 是文件变量，e 是表达式。本过程的功能是将表达式 e 的值作为一个成分写入文件 f 中，文件指针自动前移。只有在“写”方式中才可以执行本过程。要求表达式 e 的值对文件成分的类型赋值相容。

如果程序中对同一个文件有连续多个 write 语句如

```
write(f,e1); write(f,e2); write(f,e3); ...
```

可以合写为一句

```
write(f,e1,e2,e3,...)
```

请注意，我们过去学过的 read 语句与 write 语句和这里讲的用法区别是：这里的第一个参数都是文件变量。如果第一个参数不是文件变量，则按过去所学的意义理解。

11.2.4 程序实例

[例 1] 把 100 到 200 之间的所有素数写入文件 sfile 中去。

程序清单如下:

```
PROGRAM prime(sfile);
  VAR
    i,k,m:integer;
    sfile:FILE OF integer;
BEGIN
  rewrite(sfile);
  FOR i:=100 TO 200 DO
    BEGIN
      k:=2; m:=trunc(sqrt(i));
      WHILE (k<=m) AND (i MOD k<>0) DO k:=k+1;
      IF k>m THEN write(sfile,i);
    END
END.
```

这个程序中求素数仍然是采用第五章介绍的尝试法的思路，只是程序细节与过去的例子有些差别。

[例 2] 将上例程序存入文件 sfile 中的一批整数取出显示在屏幕上。

由于事先不知道文件成分个数，所以利用函数 eof 判断文件的结束。程序清单如下：

```
PROGRAM  main(sfile,output);
  VAR
    sfile:FILE OF integer;
    x:integer;
BEGIN
  reset(sfile);
```

```
  WHILE NOT eof(sfile) DO
    BEGIN
      read(sfile,x);
      write(x:5)
    END;
  writeln
END.
```

[例 3] 从键盘上读入 10 个学生的记录存入文件 stu 中去。

程序清单如下:

```
PROGRAM StuRecord(input,stu);
  CONST
    N=10;
  TYPE
    student=RECORD
              name:PACKED ARRAY[1..20] OF char;
              age:integer;
              score:real
            END;
  VAR
    stu:FILE OF student;
    stud:student;
    i,j:integer;
BEGIN
  rewrite(stu);
  writeln('Please input data');
  FOR i:=1 TO N DO
    BEGIN
      FOR j:=1 TO 20 DO read(stud.name[j]);
      readln(stud.age,stud.score);
      write(stu,stud);
    END;
END.
```

利用此程序，把 10 个学生的记录存入了文件 stu 中。按此程序的设计， 键盘输入的格式应是：每个学生的信息占一行，前 20 个字符是姓名（短者要用空格补足 20 个字符），然后是年龄，成绩。

[例 4] 已知磁盘上有两个文件 f1 和 f2，成分类型都是如上例中 student 那样的记录，现把两个文件合并成一个文件 f3，合并时原 f2 内容接在原 f1 内容之后。

把文件 f1 和 f2 按“读”方式打开，f3 按“写”方式打开，然后把 f1 和 f2 文件中的

数据写入 f3 中去。

程序清单如下:

```
PROGRAM unitef1f2(f1,f2,f3);
  TYPE
    student=RECORD
              name:PACKED ARRAY[1..20] OF char;
              age:integer;
              score:real
            END;
  VAR
    f1,f2,f3:FILE OF student;
    stud:student;
BEGIN
  reset(f1);
  reset(f2);
  rewrite(f3);
  WHILE not eof(f1) DO
    BEGIN
      read(f1,stud);
      write(f3,stud);
    END;
  WHILE not eof(f2) DO
    BEGIN
      read(f2,stud);
      write(f3,stud);
    END;
END .
```

程序执行后新文件 f3 中存放的是 f1 和 f2 中的数据。

11.3 正文文件

11.3.1 什么是正文文件

正文文件又称为文本文件，是一种特殊文件，正文文件类型由预定义的类型标识符 text 来表示。除了类型名用 text 以外，上一节所讲的对一般二进制文件的有关规定，对正文文件同样适用。

在 11.1 节已经讲过，正文文件是以字符为成分的文件。从某种不严格的意义上说，可以把 text 类型看作就是 FILE OF char 类型。这样说是因为凡是能对 FILE OF char 类型文件进行的操作都能对 text 类型文件进行。

但是，严格地说，text 文件又和 FILE OF char 不同。这样说是因为 PASCAL 中对 text 文件还规定了若干特有的属性和特有的操作。这些特有的属性和特有的操作是：

一、 text 文件的成分除了普通字符外， 还可以有一种称为“行结束符”的特殊成分。完整文件的最后一个成分应是行结束符。这样，整个文件被划分为若干个行，每行都以行结束符结尾。行结束符的特殊作用见下面 eoln 函数、readln 过程及 writeln 过程的说明。

二、用 read 过程和 write 过程读写 text 文件时，其实参不仅可以是字符型， 还允许是其它某些类型。实际上输出时是将数据自动地转换为字符序列才写入文件，输入时是从文件上读取字符序列，转换为所需类型的数据后再赋给变量。

三、提供了预定义的 writeln 过程、readln 过程及 eoln 函数。

顺便指出，一般编辑软件生成的文件都是正文文件，与 PASCAL 中对正文文件的要求是兼容的。所以，也可以在 PASCAL 以外用别的工具生成文件，然后再用 PASCAL 程序去处理。

11.3.2 正文文件的行结构及行结束函数 eoln

前面已经说过，正文文件的内容是具有行结构的字符序列，每行最后一个成分是行结束符。

按标准，行结束符可以看作一个特殊的字符，若直接读取，读得值为空格号（非标准的 PASCAL 中可能读得其它值）。但是，既然它具有特殊作用，显然它在文件上存放时所用的代码和空格号不会相同。所以我们不能用读字符的办法来判断当前成分是不是行结束符，也不能用写字符的办法来往文件上写一个行结束符。

判断当前成分是不是行结束符可以使用行结束函数 eoln，往文件上写入一个行结束符可用下面介绍的 writeln 过程，在读文件时要使文件指针移过行结束符可用下面介绍的 readln 过程。

行结束函数 eoln(f)

这个预定义函数只适合具有行结构的 text 类型文件。参数 f 是文件变量， 函数值是布尔类型。当文件指针指向符时，eoln(f)的值为 true，否则为 false。利用这个函数，可以判断文件的某一行是否读完。eoln 是英文 end of line(行结束)的缩写。

在常见的某些实际系统中，物理文件上的行结束符由回车号跟一个换行号组成，也有的系统仅用回车号表示。

某些非标准的 PASCAL 系统中，将行结束符当成字符来读写效果与上面所述不同。例如在 Turbo PASCAL 中，必须两次读字符才能将一个行结束符读过去，第一次读得回车号（ ASCII 码为十六进制 0D），第二次读得换行号（ASCII 码为十六进制 0A）。反之， 往文件上依次写入这两个控制字符，同样可以构成一个行结束符。换句话说，Turbo PASCAL 中行结束符不是一个字符，而是两个字符。不过，虽然这些操作效果与标准不同，但 eoln 函数、writeln 以及 readln 过程对行结束符的宏观作用仍然和标准是一致的。所以，只要不将行结束符当字符来读写，程序就可以是通用的。

另外，前面说过，标准 PASCAL 要求正文文件的最后一个成分是行结束符，但目前大多数的实际系统都允许最后一行没有行结束符。所以我们编写应用程序时最好能兼顾这种情况。

这里再顺便介绍一下“文件结束”的标志问题。如何标志文件结束，标准中不作规定，常见的实际系统是这样的：一般二进制文件，结束位置由目录中登记的文件长度来决定。而正文文件，在文件最后一个成分之后再往磁盘上记上一个“文件结束号”（ASCII 码为十六进制 1A）作为标志。这个控制符不算在文件的成分之中，但正文文件中间不允许出现这个控制符。有时，我们将一个一般二进制文件当成正文文件来显示时，会发现很长的文件显示出来很短，就是因为文件中某个字节代码恰好等于这个“文件结束号”，被误以为结束了。

[例 1] 将正文文件 f1 复制为文件 f2，复制时其中的大写字母都换成小写。

```
PROGRAM utol(f1,f2);
  VAR
    f1,f2:text;     {说明 f1 和 f2 都是正文文件}
    c:char;
BEGIN
  reset(f1); rewrite(f2);          {打开文件}
  WHILE NOT eof(f1) DO             {检查文件结束与否}
    BEGIN                          {复制一行}
      WHILE NOT eoln(f1) DO        {检查行结束与否}
        BEGIN                      {复制一字符}
          read(f1,c);              {读取字符}
          IF (c>='A') AND (c<='Z') THEN
             c:=chr(ord(c)-ord('A')+ord('a'));
          write(f2,c)              {写字符}
        END;
      readln(f1); writeln(f2)      {复制行结束符}
    END
END .
```

这个程序中 write 和 read 语句的用法和上一节一样，也就是说将 text 文件看成了 FILE OF char 文件。而 readln 和 writeln 下文还要介绍。

这个程序编成了二重循环，外层是处理文件中的各个行，内层是处理行中的各个字符。

这个程序内层每读一个字符时只检查 eoln 而不检查 eof， 这是因为按标准最后一个成分应是行结束符，所以只需要在行结束符后检查 eof。假如文件不标准， 最后一行没有行结束符就结束了整个文件，则用这个程序就会出错。

如要兼顾不标准的文件，可将程序改写如下：

```
PROGRAM utol(f1,f2);
  VAR
    f1,f2:text;     {说明 f1 和 f2 都是正文文件}
    c:char;
BEGIN
```

```
reset(f1); rewrite(f2);                {打开文件}
WHILE NOT eof(f1) DO                   {检查文件结束与否}
  IF eoln(f1)                          {检查行结束与否}
    THEN
      BEGIN
        readln(f1); writeln(f2)        {复制行结束符}
      END
    ELSE
      BEGIN                            {复制普通字符}
        read(f1,c);
        IF (c>='A') AND (c<='Z') THEN
           c:=chr(ord(c)-ord('A')+ord('a'));
        write(f2,c)
      END;
END .
```

后一方案适应面宽些，但运行效率不如第一方案。

11.3.3 正文文件的读写

从正文文件读取信息可以用 read 过程和 readln 过程。

用 read 过程读 text 文件时，read 语句的形式和上一节介绍的一般二进制文件中的形式一样，只是现在其中的变量不仅可以是字符型（或其子域），还允许是整数类型（或其子域）及实数类型。有的系统中还允许读入字符串。实际上输入时是从文件上读取字符序列，转换为上述类型的数据后再赋给变量。

读取字符序列时的规则和我们过去讲的 read 语句读键盘输入的字符一样。若变量是字符型（或其子域）则只读一个字符，若变量是数值（整型或实型），则指针向前略过开头的空格及行结束符，从遇到的第一个可以构成字面形式数值的字符开始，依次读取各个字符，直到一个不能用来构成字面形式数值的字符的前一个字符为止，将这一段字符序列看作十进制的字面常数，化为二进制后再赋给变量。

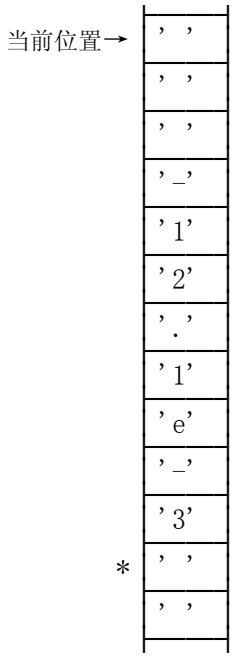


图 11.1　正文文件中的字符序列

例如，假设文件上从当前位置开始的各字符如图 11.1 所示。若变量是实型，则读取时略过三个空格号，这里的“第一个可以构成字面形式数值的字符”就是指的图 11.1 中第一个'-'号，而“不能用来构成字面形式数值的字符”就是指'3'后面的空格号（即图 11.1 中标*号处），因此读得字符序列

-12.1e-3

将其看作指数形式的常数，化为二进制浮点数后赋给变量。然后文件指针就移到图 11.1 中标*号处，下一次读取就从标*处开始。

若在遇到第一个“可以构成字面形式数值的字符”之前先遇到了既非空格又非行结束符的其它字符，或文件已结束，则算是出错。另外，这样转换所得的值还必须对参数中相应的变量赋值相容，否则也算错误。

某些非标准的系统中细节规定与上面所述有些不同。例如按上面所述，数字后面可以跟任何“不能用来构成字面形式数值的字符”，但Turbo PASCAL规定只能跟空格或行结束符，否则就算是出错[①]。

从上面的叙述可以知道，read 语句中为读得一个数据，实际从正文文件上读取的可能是多个成分。

再介绍 readln 语句。语句

```
readln(f,v1,v2,...)
```

的作用是：先执行

```
read(f,v1,v2,...)
```

接着再前移 f 的文件指针，直到移过一个行结束符。而语句

```
readln(f)
```

的作用是：前移 f 的文件指针，直到移过一个行结束符。

向正文文件上写信息可以用 write 过程和 writeln 过程。

write 语句的完整的语法见图 11.2。

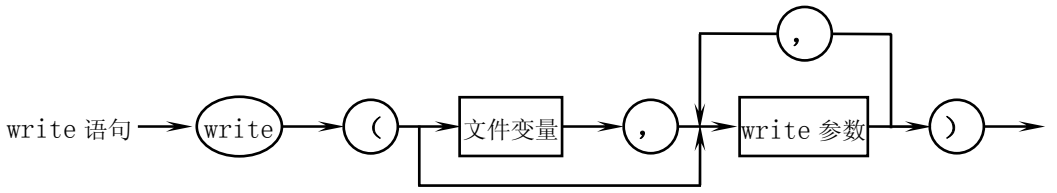


图 11.2　write 语句的完整语法图

在第三章我们没学文件时曾给出一个不完整的 write 语句语法图，即图 3.2。与之相比，现在的图多了一个可选的“文件变量”参数。其中的“write 参数”仍然是第三章图 3.3 的定义，即“输出项”后面可以带或不带域宽。如果不带域宽，就和上一节用于一般二进制文件时的格式一样了。用于一般二进制文件时只能不带域宽，而用于正文文件时允许带域宽。

用 write 过程写 text 文件时，上述 write 参数中的输出项不仅可以是字符类型，还允许输出整数类型、实数类型、布尔类型、字符串类型的值。实际上输出时都自动地转换为字符序列，必要时还包括进位制转换，将转换成的各个字符依次写入文件。具体的转换规则和第三章介绍的向显示终端显示数据时的规则完全一样，即可以有标准格式和自定义格式，标准格式取决于具体 PASCAL 系统的规定，自定义格式则取决于 write 参数中所带的域宽。 为了构成需要的格式，系统自动地生成必要的空格号字符。

从上面叙述可以知道，write 语句中的一个输出项，写到正文文件上可能是多个成分。

[①] 本书的有些例题在 Turbo PASCAL 中不能通过（例如 4.3 节的[例 8]，见该页的注），正是这个原因。

至于 writeln 语句的语法原则上也类似，即在图 3.2 的基础上增加一个可选的“文件变量”参数。以下形式的 writeln 语句

writeln(f,参数 1,参数 2,...)

的作用是：先执行

write(f,参数 1,参数 2,...)

接着再往 f 文件上写入一个行结束符。而语句

writeln(f)

的作用是：往 f 文件上写入一个行结束符。

只有对 text 类型的文件才可以使用 readln 语句和 writeln 语句。

从上面叙述可以知道，同一个正文文件可以保存不同类型的数据，这一点比上一节介绍的普通二进制文件灵活。但是它保存数值性数据时输入和输出都要经过进位制转换，所以运行效率较低，而且生成的文件也较长。另外，若是实数，进位制转换时还会带来误差。

[例 2] 下面的程序运行时往正文文件 fdat 中写入四个整数，同时按同样的格式在屏幕显示。

```
PROGRAM fileout(output,fdat);
  VAR
    x,y:integer;
    fdat:text;
BEGIN
  rewrite(fdat);
  x:=10;
  y:=20;
  writeln(x:6,y:6);
  writeln(fdat,x:6,y:6);
  writeln(2*x:6,2*y:6);
  writeln(fdat,2*x:6,2*y:6);
END .
```

程序执行后在屏幕上显示的结果：

```
    10    20
    20    40
```

这时在磁盘上也有了一个 fdat 文件，如在操作系统下用显示正文文件内容的命令显示 fdat 文件内容，会看到和屏幕上显示的完全一样。实际在文件上，每个数用 6 个字符（包括格式中所加的空格号），还有两个行结束符，整个文件共 26 个成分。

[例 3] 以下程序从上例建立的文件 fdat 中读取数据，显示在屏幕上。

```
PROGRAM filein(fdat,output);
  VAR
    x:integer;
    fdat:text;
```

```
BEGIN
  reset(fdat);
  WHILE not eof(fdat) DO
    BEGIN
      WHILE not eoln(fdat) DO
        BEGIN
          read(fdat,x);
          write(x);
        END;
      readln(fdat);
      writeln;
    END;
END .
```

程序的运行结果如下:

```
1020
2040
```

注意文件中所加的空格号在读取时已被略过去。

11.3.4 预定义文件 input 和 output

标准输入文件 input 和标准输出文件 output 也是正文文件。系统已经预定义它们为 text 类型，所以在程序中不必再说明 input 和 output 的类型。另外，PASCAL 中规定，对于 input 和 output 也不必执行“打开”操作，即在使用它们时不必执行 reset 和 rewrite 过程，可以认为 input 已经自动打开为“读”方式，output 已经自动打开为“写”方式。

另外，PASCAL 中规定，本章介绍的所有有关文件操作的预定义过程和预定义函数中，凡是作为文件变量的参数是 input 或 output 处，该文件变量参数均可省去不写。省掉时，有关的逗号或括号也相应地去掉。

如

```
eof(input)                 可以写作    eof
read(input,x,y)            可以写作    read(x,y)
write(output,'x=',x:8:1)   可以写作    write('x=',x:8:1)
writeln(output)            可以写作    writeln
```

等等。

现在可以知道，第三章所讲的输入输出语句的语法格式实际上只是现在所讲的语法格式中省掉 input 和 output 后的形式。

再顺便说明一个问题：以前各章节讲到过某些类型的数据“不能用 read 语句和 write 语句直接输入输出”，其实当时指的都是省略了文件参数的 read 和 write 语句， 实际上就是以 input 或 output 为文件参数 read 和 write 语句。因为 input 和 output 都是正文文件， 而正文文件只能输入输出上面所介绍过几个类型，所以说其它类型都不能用这样的语句输入输出。假如不省略文件参数，而采用一般二进制文件，则上一节我们已经知道，

前面所学过的各种类型的数据都可以输入输出，当然，这需要事先规定好文件的成分类型。

[例 4] 从键盘键入一篇正文，将它存到磁盘正文文件 diskfile 中。然后，为检验存盘是否正确，再从该文件中读回来显示在显示器上。

这个程序实际上是两次复制文件，第一次是从 input 文件复制到 diskfile 文件， 第二次是从 diskfile 文件复制到 output 文件。程序中两次复制的算法是一样的，不同点只在于：参数中的 input 及 output 可省略，input 及 output 文件不必作打开操作。

```
PROGRAM CopyText(input,output,Diskfile);
  VAR
    Diskfile:text;
    ch:char;
BEGIN
  writeln('Please input text');
  writeln;
  rewrite(Diskfile);
  WHILE not eof DO
    BEGIN
      WHILE not eoln DO
        BEGIN
          read(ch);
          write(Diskfile,ch);
        END;
      readln;
      writeln(Diskfile);
    END;
  writeln;
  reset(Diskfile);
  WHILE not eof(Diskfile) DO
    BEGIN
      WHILE not eoln(Diskfile) DO
        BEGIN
          read(Diskfile,ch);
          write(ch);
        END;
      readln(Diskfile);
      writeln;
    END;
END.
```

这里需要说明一下在键盘输入时如何产生所需的控制符。这些由所用的操作系统决

定，在常见的 PC-DOS 操作系统下是这样的：按回车键可产生行结束符，同按<Ctrl>键和<Z>键可产生标志文件结束的“文件结束号”代码。不过我们在 3.4.3 中曾说过， 键盘输入的信息只有在打回车键时才整行地送入文本输入缓冲区，所以只按一下<Ctrl-Z>程序还不能收到这个代码，必须再按一下回车键。最后的这个回车键因为是在“文件结束”之后，所以对程序不起作用。

下面是一次程序运行时的显示，其中的“^Z”是在按<Ctrl-Z>时屏幕上显示的：

```
Please input text

LIFe is dear indeed,
Love is price less TOO,
But FOR freeDOm's sake,
I may part with the two.
^Z

LIFe is dear indeed,
Love is price less TOO,
But FOR freeDOm's sake,
I may part with the two.
```

11.3.5 正文文件存放数值性数据应注意的某些问题

正文文件存放数值性数据时，不仅进位制转换降低效率和引入误差，还有若干其它问题需要注意。

例如下面的程序：

```
PROGRAM test1(t1,output);
  VAR
    x,y:integer;
    c:char;
    t1:text;
BEGIN
  x:=3; y:=45; c:='A';
  rewrite(t1);
  write(t1,x:2);
  writeln(t1,y:2);
  writeln(t1,c);
  reset(t1);
  read(t1,x);   writeln('x=',x);
  readln(t1,y); writeln('y=',y);
  readln(t1,c); writeln('c=',c);
END .
```

程序中往文件 t1 上写了三个数据，又读回来，我们可能会认为结果是

```
x=3
y=45
c=A
```

可是实际运行结果却是

```
x=345
Invalid numeric format
```

这是因为往文件上写 y 时恰好两位，前面没有空格，而前面写 x 时又没有换行，两个数连起来了，读取时被当成一个数 345。继续往下读 y 时遇到字符'A'， 就被认为是非法数值格式而报错。

又例如下面的程序：

```
PROGRAM test2(t2,output);
  VAR
    x:real;
    c1,c2:char;
    t2:text;
BEGIN
  x:=3.4881;
  rewrite(t2);
  writeln(t2,x:4:2);
  reset(t2);
  read(t2,c1,c2,x);
  writeln('c1=',c1,'    c2=',c2,'    x=',x:5:1);
END .
```

运行的结果是

```
c1=3    c2=.    x= 49.0
```

写入文件 t2 的是一个数据，读回来却成了三个。若把写文件时的域宽改动一下，结果就又不一样，请读者自行分析。

还有，我们知道 read 语句读数值时可以略过空格和行结束符，所以可以不管行结束而读取数据。于是有人可能会把前面的[例 3]程序修改如下

```
PROGRAM filein(fdat,output);
  VAR
    x:integer;
    fdat:text;
BEGIN
  reset(fdat);
  WHILE not eof(fdat) DO
    BEGIN
```

```
            read(fdat,x);
            writeln(x);
        END;
    END .
```

运行后发现结果是

```
10
20
20
40
0
```

比预计的结果多出来一个 0。这是因为读完最后一个数 40 后，文件指针指向 40 后边的行结束符,还差一步没到文件结束位置,故执行 eof 函数时判断文件没完。再执行 read(fdat,x)继续读取，刚略过一个行结束符还未读到数时文件就结束，出现错误。但并非对所有错误系统都能报错。这个系统在遇到这种错误时并没有报错，而是给 x 赋了个 0，就出现这个现象。须知道 read 语句读入数值性数据时只能略过数前面的空格和行结束符，而不会略过后面的空格和行结束符。

以上问题的主要原因是存在文件上都是字符序列，而字符序列与数值间的转换可以有不同的规则。这一方面给我们提供了灵活性，另一方面也增加了复杂程度，需要注意。

11.4 缓冲区变量及 put 和 get 过程

前面介绍的输入和输出语句等已经完全可以实现各种文件操作，因此可以不必使用本节介绍的 put 和 get 过程。但本节介绍的 put 和 get 过程实际上是 PASCAL 文件操作的基本动作，早期的 PASCAL 版本只提供这些输出输入功能，而相当于 write 和 read 的功能都可以用它们实现。为使读者有全面的了解，这里给以简单介绍。

11.4.1 缓冲区变量

程序中每说明一个文件变量，就自动地建立一个缓冲区变量。缓冲区变量的类型就是该文件的成分类型。程序中表示缓冲区变量的格式是

```
文件变量^
```

缓冲区变量是缓冲存放文件当前成分的变量。

在“读”方式中，缓冲区变量的值自动地取文件当前成分的值，程序中只要作为表达式引用缓冲区变量就可以取得文件当前成分的值。

在“写”方式中，由于当前位置是文件的结束位置，故缓冲区变量的初始值是“未定义”的。但程序中可以给缓冲区变量赋值，赋值后缓冲区变量就有值了，其值是准备将来写进文件当前位置的。

11.4.2 put(f)过程

其中参数 f 是文件变量。只有在“写”方式中才可以调用 put 过程。put 过程的作用是将缓冲区变量的值写入到文件当前位置（由于“写”方式中当前位置是文件的结束位置，

故实际上接到了文件末尾)，并使文件指针前移指向新的结束位置，使缓冲区变量的值又成为“未定义”的。

前面 11.2 节介绍的语句

```
write(f,e)
```

功能等效于

```
BEGIN f^:=e; put(f) END
```

11.4.3 get(f)过程

其中参数 f 是文件变量。只有在“读”方式中且 eof(f)为 false 时才可以调用 get 过程。get 过程的作用是使文件指针前移指向下一成分， 并将新的当前成分的值赋给缓冲区变量。假如新的当前位置是文件的结束位置，则使缓冲区变量的值成为“未定义”的。

前面 11.2 节介绍的语句

```
read(f,v)
```

功能等效于

```
BEGIN v:=f^; get(f) END
```

11.4.4 实例

[例 1] 将 11.2 节[例 3]改用本节介绍的功能实现。

```
PROGRAM StuRecord(input,stu);
  CONST
    N=10;
  TYPE
    student=RECORD
              name:PACKED ARRAY[1..20] OF char;
              age:integer;
              score:real
            END;
  VAR
    stu:FILE OF student;
    i,j:integer;
BEGIN
  rewrite(stu);
  writeln('Please input data');
  FOR i:=1 TO N DO
    BEGIN
      FOR j:=1 TO 20 DO read(stu^.name[j]);
      readln(stu^.age,stu^.score);
      put(stu)
    END;
END.
```

这里 stu^是文件 stu 的缓冲区变量，属于 student 记录类型。而程序中

```
FOR j:=1 TO 20 DO read(stu^.name[j]);
readln(stu^.age,stu^.score);
```

两行的作用是给 stu^的各个域赋值，然后用 put(stu)将其写到文件上。这里直接使用缓冲区变量，省去原来程序中的工作记录变量 stud。

[例 2] 将 11.2 节[例 4]改用本节介绍的功能实现。

```
PROGRAM unitef1f2(f1,f2,f3);
  TYPE
    student=RECORD
              name:PACKED ARRAY[1..20] OF char;
              age:integer;
              score:real
            END;
  VAR
    f1,f2,f3:FILE OF student;
BEGIN
  reset(f1);  reset(f2);
  rewrite(f3);
  WHILE not eof(f1) DO
    BEGIN
      f3^:=f1^; get(f1); put(f3)
    END;
  WHILE not eof(f2) DO
    BEGIN
      f3^:=f2^; get(f2); put(f3)
    END;
END .
```

同样，我们看到，直接使用缓冲区变量，可以省去一个中间记录变量 stud。

现在的发展趋势，倾向于不再使用 put 和 get 功能。某些版本的 PASCAL 已不提供这两个功能。不过 PASCAL 的标准中仍然有这些功能。

11.5　综合实例

[例 1] 归并排序

假设已有两个文件 play1 和 play2，其中分别登记了一部分运动员的得分资料，每个记录包括运动员号及得分两个域，文件均已按得分由高到低的顺序排好，且末尾多录一个“哑记录”,“哑记录”中得分域记一任意负数作为标志（真实得分都是正数）。现要求将这两部分运动员的资料合并到一个文件 play3 中，合并后仍按得分由高到低的顺序排列，

且末尾也要有同样的“哑记录”。

解决这个问题,可以先将 play1 和 play2 的头一个记录分别读入记录变量 r1 和 r2 中。若 r1 的得分不低于 r2，则将 r1 写入 play3 并从 play1 中再读一个记录到 r1 中；否则将 r2 写入 play3 并从 play2 中再读一个记录到 r2 中。重复这个比较及读写。若有一个文件读完，则读入记录变量中的必然是“哑记录”，“哑记录”得分是负数，低于任何真实得分，故继续循环时写入 play3 的必然是另一个文件的内容。若两个文件都读完，r1 和 r2 中必然都是“哑记录”，此时可结束循环，将一个“哑记录”写入 play3 即可。

```
PROGRAM sort2(play1,play2,play3);
  TYPE
    rec=RECORD
          num:integer;
          score:integer
        END;
  VAR
    play1,play2,play3:FILE OF rec;
    r1,r2: rec;
  BEGIN
    reset(play1); reset(play2);
    rewrite(play3);
    read(play1,r1); read(play2,r2);
    WHILE NOT(eof(play1) AND eof(play2)) DO
      IF r1.score>=r2.score
        THEN  BEGIN write(play3,r1); read(play1,r1) END
        ELSE  BEGIN write(play3,r2); read(play2,r2) END;
    write(play3,r1)
  END .
```

程序中的

```
NOT(eof(play1) AND eof(play2))
```

也可以改为

```
(r1.score>=0) OR (r2.score>=0)
```

这种在每个文件的末尾加一个“哑记录”的办法是简化程序的一个技巧，如果不要这个“哑记录”，程序更复杂些，读者可自行分析。

[例 2] 考分换算

一般统考的结果，单看一个得分往往不能全面反映考生的水平，因为它受试题性质的影响。试题较难，大家得分都会低些；试题较容易，大家得分都会高些。有的试题，大家得分集中，高分及低分都较少；有的试题，大家得分很分散，分数相差较大。分数相差同样的几分，若是得分集中的试题，考生水平差别较大；若是得分分散的试题，考生水平差别不大。

现在采用的“标准分”，是利用统计的方法消除上述因素的影响，故能较客观地反映考生的水平。由原始分换算标准分的算法如下：

设考生总数为 N 个，原始分分别为 x_1，x_2，……，x_N。下面式中的Σ表示对于 i 从 1 到 N 作累加。

平均分 $e=(\Sigma x_i)/N$

标准差 $s=\sqrt{(\sum(x_i-e)^2)/N}$

$z_i=(x_i-e)/s$

$y_i=100z_i+500$

最后的 y_i 就是标准分。换算时减去平均值，消去试题难易的因素；除以标准差，消去得分集中度的因素。不直接用 z 作结果而要再换算成 y，是为使结果可用正整数，符合习惯。

下面的程序，是设全部考生的原始得分已经录在一个文件 examrec 上， 每个考生一个记录，记录包括两个域，一是 8 个字符的字符串作为考生号，二是一个整数表示原始分。换算结果录到另一个文件 report 上。为便于打印，report 采用正文文件，每个考生一行，内容为考生号、原始分、标准分。结果标准分四舍五入取整。

这里先给出直接按上述公式编写的程序。

```
PROGRAM stdscore(examrec,report);
  TYPE
    rec=RECORD                          {原始分记录类型}
          num:PACKED ARRAY[1..8] OF char;
          x:integer
        END;
  VAR
    examrec:FILE OF rec;                {原始分记录文件}
    report:text;                        {换算结果报告文件}
    buf:rec;                            {输入记录暂存}
    e,s,z:real;
    y,n:integer;
BEGIN
  reset(examrec);                       {第一遍扫描，求平均分}
  e:=0; n:=0;
  WHILE NOT eof(examrec) DO
    BEGIN  read(examrec,buf); e:=e+buf.x; n:=n+1  END;
  e:=e/n;
  reset(examrec);                       {第二遍扫描，求标准差}
  s:=0;
  WHILE NOT eof(examrec) DO
    BEGIN  read(examrec,buf); s:=s+sqr(buf.x-e)  END;
```

```
    s:=sqrt(s/n);
    reset(examrec); rewrite(report);  {第三遍扫描，产生结果}
    WHILE NOT eof(examrec) DO
      BEGIN
        read(examrec,buf);
        z:=(buf.x-e)/s;
        y:=round(z*100+500)
        writeln(report, buf.num, buf.x:6, y:6)
      END
  END .
```

这个程序已经是可以实用的了。不过我们看到，程序中将输入的文件扫描了三遍。计算机对外存的访问速度远远低于访问内存，若能合并操作，减少扫描遍数，不仅可简化程序，还能节省机时。下面我们讨论能否进一步改进。第三遍扫描的操作因每一轮循环都要引用前面扫描的最后结果 s 和 e，无法合并到前面的扫描中。第二遍扫描求 s，若按原公式， 累加时要引用第一遍的结果 e，也不便合并。可对 s 的公式作一些变换，公式中根号下的部分

$$(\sum(x_i-e)^2)/N$$
$$=(\sum(x_i^2-2ex_i+e^2))/N$$
$$=(\sum x_i^2-2e\sum x_i+e^2\cdot N)/N$$
$$=(\sum x_i^2)/N-e^2 \qquad (\text{代入：} e=(\sum x_i)/N\ )$$

按此变换后公式，循环累加可先求$\sum x_i^2$，不必引用 e，其余计算循环完后再作，于是这个循环就可以和第一遍扫描合并。这样改进后可以减少一遍扫描，程序如下：

```
PROGRAM stdscore(examrec,report);
  TYPE
    rec=RECORD                          {原始分记录类型}
          num:PACKED ARRAY[1..8] OF char;
          x:integer
        END;
  VAR
    examrec:FILE OF rec;                {原始分记录文件}
    report:text;                        {换算结果报告文件}
    buf:rec;                            {输入记录暂存}
    e,s,z:real;
    y,n:integer;
BEGIN
  reset(examrec);                       {第一遍扫描，求平均分及标准差}
  e:=0; n:=0; s:=0;
  WHILE NOT eof(examrec) DO
```

```
    BEGIN
      read(examrec,buf); e:=e+buf.x;
      s:=s+sqr(buf.x); n:=n+1
    END;
  e:=e/n;
  s:=sqrt(s/n-sqr(e));
  reset(examrec); rewrite(report);  {第二遍扫描，产生结果}
  WHILE NOT eof(examrec) DO
    BEGIN
      read(examrec,buf);
      z:=(buf.x-e)/s;
      y:=round(z*100+500)
      writeln(report, buf.num, buf.x:6, y:6)
    END
END .
```

习题

11.1 编程将键盘输入的一批考生的考号和成绩录入文件（请自己设计记录格式）。

11.2 编程统计上一题所得文件中考生总人数，及格（不低于 60 分）人数，不及格人数，最高成绩，最低成绩，平均成绩。

11.3 假设已有一个文件 sfile 中录有一批考生的考号和成绩，已按成绩由高到低排序，记录的格式同 11.1 题文件，且文件末尾录有一个“哑记录”,“哑记录”的成绩域记-1。编程将键盘又打入的一个考生的考号和成绩按次序插入到这一批考生记录中， 整个转存到 tfile 文件中去。

11.4 编程统计一个文本文件 docum 中的字符个数，要求行结束符不算在字符之内，但空格及其它控制符都要算在内。

附录A　ASCII 代码表

高位 / 低位	0	1	2	3	4	5	6	7
0	NUL	DLE	SP	0	@	P	`	p
1	SOH	DC1	!	1	A	Q	A	q
2	STX	DC2	″	2	B	R	B	r
3	ETX	DC3	#	3	C	S	C	s
4	EOT	DC4	$	4	D	T	D	t
5	ENQ	NAK	%	5	E	U	E	u
6	ACK	SYN	&	6	F	V	F	v
7	BEL	ETB	′	7	G	W	G	w
8	BS	CAN	(	8	H	X	H	x
9	HT	EM	)	9	I	Y	I	y
A	LF	SUB	*	:	J	Z	J	z
B	VT	ESC	+	;	K	[	K	{
C	FF	FS	,	<	L	\	L	|
D	CR	GS	-	=	M	]	M	}
E	SO	RS	.	>	N	^	N	~
F	SI	US	/	?	O	_	O	DEL

注：(1)表中“高位”、“低位”指 ASCII 代码的十六进制表示。

(2)表中：

SP 是空格（Space）；

CR 是回车（Carriage Return）；

LF 是换行（Line Feed）；

FF 是换页（Form Feed）。

附录Ｂ　Turbo PASCAL 文件系统的特点

本书课文按照标准 PASCAL 叙述，但常用的 Turbo PASCAL 系统与标准有所不同。为便

本书课文按照标准 PASCAL 叙述，但常用的 Turbo PASCAL 系统与标准有所不同。为便于使用这一系统的文件操作，这里简述 Turbo PASCAL 有关文件操作的某些规定中与课文的不同之处。

(1)系统增加了 assign 语句。Turbo PASCAL 中的 assign 语句是指定程序中的文件变量代表操作系统管理下的哪个文件。我们知道 PASCAL 对作为变量名的标识符的命名规则和操作系统中对于外部标识符的命名规则并不相同（外部文件名可以包括路径及盘符）互不兼容，所以要求它们之间建立对应关系。assign 语句的格式是

assign(文件变量,字符串)

其中文件变量是 PASCAL 程序中说明的文件变量名，而字符串则规定了它代表的外部文件按操作系统的规则所具有的文件标识符。如 assign(f1,'A:\SUB1\FILE1.TXT')

程序中每个文件在第一次打开前必须采用 assign 语句指定。

(2)文件要求它们在读或写结束后应该"关闭"，因此需要使用 close 语句。格式为

close(文件变量)

在“写”方式中如果没有关闭就结束程序的话，有可能最后若干数据没有真正写入到文件上去。不用 close 语句而直接用 reset 语句再次打开也可以起到先自动关闭再打开的作用。

(3)没有 put 和 get 功能，因为它们可以用 write 和 read 代替。

附录C　标准 PASCAL 语法汇集

（根据《GB 7591-87 程序设计语言 PASCAL》）

非终极符指针类型、程序、有正负号数、简单类型、特殊符号和构造类型仅被语义引用，不在任何产生式的右边使用。非终极符程序是文法的开始符号。

程序=程序首部“;”程序分程序“.”。

程序首部=“program”标识符[“(”程序参数“)”]。

程序参数=标识符表。

标识符表=标识符{“,”标识符}。

标识符=字母{字母|数字}。

字母=“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”。

数字=“0”|“1”|“2”|“3”|“4”|“5”|“6”|“7”|“8”|“9”。

程序分程序=分程序。

分程序=标号说明部分　常量定义部分　类型定义部分　变量说明部分　过程与函数说明部分　语句部分。

标号说明部分=[“label”标号{“,”标号}“;”]。

标号=数字序列。

数字序列=数字{数字}。

常量定义部分=[“const”常量定义“;”{常量定义“;”}]。

常量定义=标识符“=”常量。

常量=[正负号](无正负号数|常量标识符)|字符串。

正负号=“+”|“-”。

无正负号数=无正负号整数|无正负号实数。

无正负号整数=数字序列。

无正负号实数=无正负号整数“.”小数部分[“e”比例因子]|无正负号整数“e”比例因子。

小数部分=数字序列。

比例因子=有正负号整数。

有正负号整数=[正负号]无正负号整数。

有正负号数=有正负号整数|有正负号实数。

有正负号实数=[正负号]无正负号实数。

常量标识符=标识符。

字符串=“'”串元素{串元素}“'”。

串元素=撇号映象|串字符。

撇号映象=“''”。

串字符=由实现所定义的字符集合中的一个字符。
类型定义部分=[“type”类型定义“;”{类型定义“;”}]。
类型定义=标识符“=”类型表记符。
类型表记符=类型标识符|新类型。
类型标识符=标识符。
新类型=新顺序类型|新构造类型|新指针类型。
新顺序类型=枚举类型|子域类型。
枚举类型=“(”标识符表“)”。
子域类型=常量“..”常量。
新构造类型=[“packed”]非紧缩的构造类型。
非紧缩的构造类型=数组类型|记录类型|集合类型|文卷类型。
数组类型=“array”“[”下标类型{“,”下标类型}“]”“of”成分类型。
下标类型=顺序类型。
顺序类型=新顺序类型|顺序类型标识符。
顺序类型标识符=类型标识符。
成分类型=类型表记符。
记录类型=“record”域表“end”。
域表=[(固定部分[“;”变体部分]|变体部分)[“;”]]。
固定部分=记录段{“;”记录段}。
记录段=标识符表“:”类型表记符。
变体部分=“case”变体选择符“of”变体{“;”变体}。
变体选择符=[标志域“:”]标志类型。
标志域=标识符。
标志类型=顺序类型标识符。
变体=情况常量表“:”“(”域表“)”。
情况常量表=情况常量{“,”情况常量}。
情况常量=常量。
集合类型=“set”“of”基类型。
基类型=顺序类型。
文卷类型=“file”“of”成分类型。
新指针类型=“^”定义域类型。
定义域类型=类型标识符。
简单类型=顺序类型|实数类型标识符。
实数类型标识符=类型标识符。
构造类型=新构造类型|构造类型标识符。
构造类型标识符=类型标识符。
指针类型=新指针类型|指针类型标识符。
变量说明部分=[“var”变量说明“;”{变量说明“;”}]。

变量说明=标识符表“:”类型表记符。

过程与函数说明部分={(过程说明|函数说明)“;”}。

过程说明=过程首部“;”指示字|过程标识“;”过程分程序|过程首部“;”过程分程序。

过程首部=“procedure”标识符[形式参数表]。

形式参数表=“(”形式参数段{“;”形式参数段}“)”。

形式参数段=值参数指明|变量参数指明|过程参数指明|函数参数指明|可调节数组参数指明。

值参数指明=标识符表“:”类型标识符。

变量参数指明=“var”标识符表“:”类型标识符。

过程参数指明=过程首部。

函数参数指明=函数首部。

函数首部=“function”标识符[形式参数表]“:”结果类型。

结果类型=简单类型标识符|指针类型标识符。

简单类型标识符=类型标识符。

指针类型表示符=类型标识符。

可调节数组参数指明=值可调节数组指明|变量可调节数组指明。

值可调节数组指明=标识符表“:”可调节数组模式。

可调节数组模式=紧缩的可调节数组模式|非紧缩的可调节数组模式。

紧缩的可调节数组模式=“packed”“array”“[”下标类型指明“]”“of”类型标识符。

下标类型指明=标识符“..”标识符“:”顺序类型标识符。

非紧缩的可调节数组模式=“array”“[”下标类型指明{“;”下标类型指明}“]”“of”(类型标识符|可调节数组模式)。

变量可调节数组指明=“var”标识符表“:”可调节数组模式。

指示字=字母{字母|数字}。

过程标识=“procedure”过程标识符。

过程标识符=标识符。

过程分程序=分程序。

函数说明=函数首部“;”指示字|函数标识“;”函数分程序|函数首部“;”函数分程序。

函数标识=“function”函数标识符。

函数标识符=标识符。

函数分程序=分程序。

语句部分=复合语句。

复合语句=“begin”语句序列“end”。

语句序列=语句{“;”语句}。

语句=[标号“:”](简单语句|构造语句)。

简单语句=空语句|赋值语句|过程语句|转向语句。

空语句= 。

赋值语句=(变量存取|函数标识符)“:=”表达式。

变量存取=整体变量|成分变量|标识变量|缓冲区变量。

整体变量=变量标识符。

变量标识符=标识符。

成分变量=下标变量|域命名符。

下标变量=数组变量“[”下标表达式{“,”下标表达式}“]”。

数组变量=变量存取。

下标表达式=表达式。

表达式=简单表达式[关系运算符 简单表达式]。

简单表达式=[正负号]项{加法运算符 项}。

项=因式{乘法运算符 因式}。

因式=变量存取|无正负号常量|界限标识符|函数命名符|集合构造符|“(”表达式“)”|“not”因式。

无正负号常量=无正负号数|字符串|常量标识符|nil。

界限标识符=标识符。

函数命名符=函数标识符[实在参数表]。

实在参数表=“(”实在参数{“,”实在参数}“)”。

实在参数=表达式|变量存取|过程标识符|函数标识符

集合构造符=“[”[成员命名符{“,”成员命名符}]“]”。

成员命名符=表达式[“..”表达式]。

乘法运算符=“*”|“/”|“div”|“mod”|“and”。

加法运算符=“+”|“-”|“or”。

关系运算符=“=”|“<>”|“<”|“>”|“<=”|“>=”|“in”。

域命名符=记录变量“.”域区分符|域命名符标识符。

记录变量=变量存取。

域区分符=域标识符。

域标识符=标识符。

域命名符标识符=标识符。

标识变量=指针变量“^”。

指针变量=变量存取。

缓冲区变量=文卷变量“^”。

文卷变量=变量存取。

过程语句=过程标识符([实在参数表]|read 参数表|readln 参数表|write 参数表|writeln 参数表)。

read 参数表=“(”[文卷变量“,”]变量存取{“,”变量存取}“)”。

readln 参数表=[“(”(文卷变量|变量存取){“,”变量存取}“)”]。

write 参数表=“(”[文卷变量“,”]write 参数{“,”write 参数}“)”。

write 参数=表达式[“:”表达式[“:”表达式]]。

writeln 参数表=[“(”(文卷变量|write 参数){“,”write 参数}“)”]。

转向语句=“goto”标号。

构造语句=复合语句|条件语句|重复性语句|开域语句。

条件语句=如果语句|情况语句。

如果语句=“if”布尔表达式“then”语句[否则部分]。

布尔表达式=表达式。

否则部分=“else”语句。

情况语句=“case”情况下标“of”情况表元素{“;”情况表元素}[“;”]“end”。

情况下标=表达式。

情况表元素=情况常量表“:”语句。

重复性语句=重复语句|当语句|循环语句。

重复语句=“repeat”语句序列“until”布尔表达式。

当语句=“while”布尔表达式“do”语句。

循环语句=“for”控制变量“:=”初值(“to”|“downto”)终值“do”语句。

控制变量=整体变量。

初值=表达式。

终值=表达式。

开域语句=“with”记录变量表“do”语句。

记录变量表=记录变量{“,”记录变量}。

特定符号=“+”|“-”|“*”|“/”|“=”|“<”|“>”|“[”|“]”|“.”|“,”|“:”|“;”|“^”|“(”|“)”|“'”|“<>”|“<=”|“>=”|“:=”|“..”|字符号。

字符号=“and”|“array”|“begin”|“case”|“const”|“div”|“do”|“downto”|“else”|“end”|“file”|“for”|“function”|“goto”|“if”|“in”|“label”|“mod”|“nil”|“not”|“of”|“or”|“packed”|“procedure”|“program”|“record”|“repeat”|“set”|“then”|“to”|“type”|“until”|“var”|“while”|“with”。