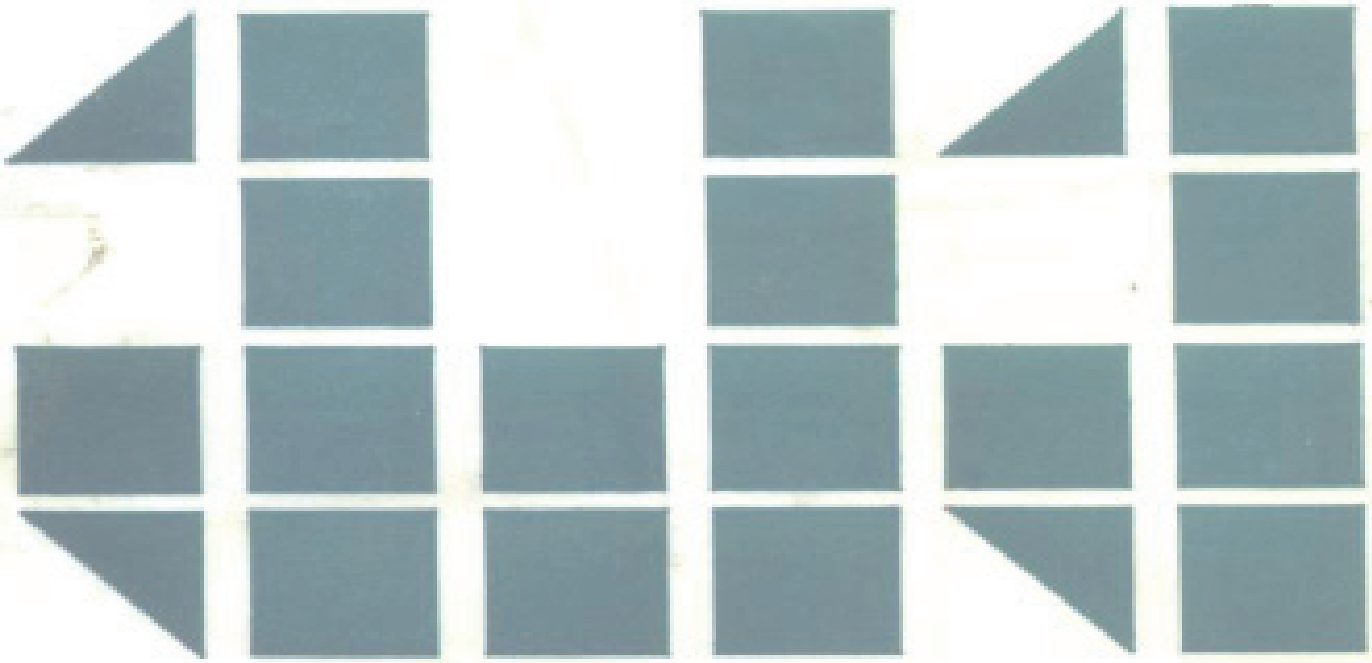
万达 编译
梁琨培 编译
马淑雅 编译
沈尧中 校
李航出版社

ADA语言



ADA 语言

万 达　梁琨瑶　马淑雅　编 译

沈 尧 中　校对

宇航出版社

内 容 简 介

ADA语言是美国国防部在实现一种通用高级语言进行的招标设计中开发出来的。它是一种功能极强的计算机高级语言，是在对比十六种方案基础上最后选定并由法国人以PASCAL语言为起点进行开发的。该语言可用于数值分析计算和系统程序设计，满足实时分析及并行操作要求，能在各种规律的大、中、小、微型计算机上实现。目前已受到国际上计算机的应用机构、大学和公司的高度重视，成为八十年代最有影响和最有代表性的一种计算机高级语言。

本书以法国计算机语言学专家Daniel-Jean David所著的Le Langage ADA为基础，参考其它有关资料编译而成。适合于计算机研制、应用、开发单位的科技人员及大专院校、职业高中计算机系的师生作参考。

ADA 语 言

编译者：万达 梁琨璐 马淑雅

校 对：沈尧中

特约编辑：高敬松

☆

宇航出版社出版

新华书店北京发行所发行

各地新华书店经售

四二二九工厂印刷

☆

开本：787×1092 1/32 印张：7.75 字数：170千字

1987年7月第一版第一次印刷 印数：1—6,500册

统一书号：15244·0097 定价：2.10元

编 译 者 前 言

目前，ADA作为一种功能极强的计算机高级语言，已经受到全世界的计算机应用研究机构、大学和公司的高度重视，成为80年代最有影响和最有代表性的一种高级语言。

ADA是为纪念历史上第一个程序设计者，在Charles Baffage设计的历史上最早的计算机——齿轮传动计算机上进行程序设计的Augusta ADA Byron (1815—1852) 女士、Lovelace伯爵夫人而得名。它是在美国国防部为实现一种通用高级语言而进行的招标设计中，由以Jean Ichablah领导的八人法国组设计的。ADA语言可以用于数值分析计算程序和系统程序的设计，并能满足实时分析及并行操作的要求，适用于各种规模的计算机。

随着计算机科学的发展，涌现出适应各种需要的各式各样计算机。这些计算机，特别是嵌入式计算机的出现，使得在软件上的花费急剧增加，与此同时也出现了各种各样的程序设计语言。现在世界上用于各个方面的计算机语言不下数百种，它们在设计过程中缺乏统一标准，因此在程序交流中要实现这些语言的相互转换花费极大。基于这种原因，美国国防部为适应三军日益增长的程序应用中语言的要求，于1975年2月成立了高级语言工作组 (Higher-Order Language Working Group，简称HOLWG)，其主要使命是建立一套适用于嵌入式计算机系统的高级标准语言。

HOLWG的第一步工作是根据三军、工厂和大学的实用情况建立一套文件，这套文件能够对公共的计算机程序设计语言性能提出要求。该项工作由David Fisher领导，在四年中建立的这一系列文件分别称为“Strawman”(1975)“Woodenman”(1975)，“Tinman”(1976)，Ironman (1978)，Steelman (1979) 等。这些文件的最后文本包括了100多条要求，它们对语言在数据类型、控制结构、模块、任务和异常等方面规定了明确的特性，同时也提出了可读性、非过分通用性、简明性和可验证性等方面的要求。

第二步是分析现有语言，看是否有满足这些性能要求的语言存在。如果没有，则由HOLWG提出一个设计和研究一种能够满足这些要求的语言的方案。

经过对26种现有语言的广泛调查研究，在1977年得出结论：没有哪一种现存语言能满足要求 (Tinman)，应当设计一种新语言。HOLWG还推荐选择Pascal，Algol 68，PL/1语言之一作为新语言设计的出发点。

体现HOLWG全部要求的新语言设计招标细则是1977年5月发出的。从16种方案中选出4种，给定6个月(1977.8～1978.2)作为初步设计阶段，这4种方案分别是CII-Honeywell-Bull的绿色方案，Intermetrics的红色方案，SRI International的黄色方案和Soft Tech的兰色方案（颜色作为评选时的匿名标志）。

他们都选择了PASCAL作为起点，这一有利因素和HOLWG的具体要求大大缩短了语言设计所花费的时间，使有可能在6个月内完成初步设计。尽管设计一种新语言还是有很大的余地，但是四种初步设计仍然存在着惊人的差别。

1978年2月到3月，约有80名来自各大学、工厂和政府部门的专家对四种初步方案进行评议。选出绿、红两种方案并在1978年4月至1979年3月的时间内作进一步的设计。

完整的设计于1979年3月15日完成并由50多个分析小组进行了全面的分析。1979年4月底在华盛顿召开的4天会议上讨论了他们的分析结果。在5月2日HOLWG会议上，绿色方案被评为优胜者。这一方案以ADA重新命名，为的是纪念Augusta ADA Byron。ADA参考手册和理论基础第一次在1979年6月SIGPLAN Notices杂志上发表。

绿色方案胜过红色方案的重要原因在于：

1. 绿色方案的结构更为现代化和更具有可行性。同时，在几个月的过程中它的设计是稳定的（而红色方案则到最后一分钟仍在变动）。因而被认为是一种较少冒险性的语言。

2. 绿色方案对于指明和能由编译器对接口部分进行有效检验的分离编译程序单元有一套完整的处理方法。并有语言设计方法学的语言级支持。而红色方案没有提到为建立包括许多相关模块的大系统所需要的语言级支持。

1979年5月以后，ADA经历了广泛的技术测验和在100多个应用场合通过进行实用程序设计来对其进行评价。这些结论和对语言的修改意见曾在1979年10月的4天会议上提出，交给ADA语言的设计者们。经过这些鉴定和修改，在1980年7月产生了最早能作为研究编译程序基础的基本ADA语言。并被美国国防部确认为三军标准高级语言。

在1979年8月15日通过语言的全部语法检验和子集的试验编译程序之后，经过进一步努力，于1981年夏季完成最早

的实用编译程序。1982年与1983年经过两次修改补充，ADA的实现工作告一段落，并正式投入使用。

目前，美国各军兵种都制定了自己的实现计划，世界各国的研究机构也都围绕ADA开展了一系列的研究工作。各计算机公司也相继为自己的系列机配置了ADA或其子集的编译程序。其中有的子集编译已经在我国流传。

鉴于ADA语言强有力的功能和目前的发展趋势，尽管它在世界范围内尚未普及，我们还是愿意根据已有资料尽早对它作一些介绍，希望能对我国的程序设计人员有所启发和对加速我国计算机应用科学的现代化起到作用。

本书是以法国计算机语言学专家Daniel-Jean David所著的ADA语言（Le Langage ADA）为基础，参考其他有关资料编译而成的。第一章介绍ADA语言概貌。指出它与Pascal的相似之处以及哪些方面是Pascal的延伸、推广或填补其空白。还列举了ADA对Pascal的重大创新。其余各章节对第一章中简述的概念一一阐明。其关键内容有

——IF THEN ELSE，LOOP，CASE等结构

——类型

——程序色、模块化、分离编译及类属元素

——标准环境及输入/输出

——并行和任务

——异常与错误处理

——表示指明和参注

书中的例题都比较新颖。没有采用那些太古典、太论证性的例子。题的来源是Jean Ichablach的“ADA程序语言设计的基本原理”一书及其他参考资料。

要很好地理解ADA语言，需要对一种甚至几种高级语言都比较熟悉。如果已经对诸如FORTRAN，BASIC，COBOL等语言有所了解的读者，可以先从了解PASCAL语言入手来理解ADA。因为ADA是以PASCAL为基础发展起来的：它几乎与PASCAL有相同的结构化程序构造。特别是在数据类型方面，它与PASCAL有相同的管理办法。

最后需要说明的是，对于ADA语言的一些术语，目前国内尚无统一译法。我们参考了南京大学徐家福教授对ADA介绍的文章和709所王振宇教授编译的ADA语言文本参考手册中所采用的一些专用名词译法。由于编译者水平有限肯定有不少错误及不当之处，敬请读者批评指教。

编　译　者

一九八四年四月

目　录

第一章　ADA语言的概貌

美国国防部的招标细则曾建议：新语言可以借鉴于Pascal，Algol 68或PL/1。一个很有意义的事实是：初选出的四家公司（CII—Honeywell—Bull，Intermetrics，SRI International及Soft—Tech.）不谋而合地都以PASCAL作为开发新语言的基础。本章将介绍ADA语言的概貌，目的在于指出：ADA语言有些概念是PASCAL的延伸，有些概念则摆脱了PASCAL的局限性。但是ADA并不是PASCAL的简单扩充，许多新的、重要的概念是ADA所特有的。

为便于阅读，本书采用如下约定：

1.ADA语言的键词用粗体大写字母；

2.标识符（无论是语言预定义的还是用户定义的）一律用普通大写字母。

1.1　结构语句

ADA语言最重要的结构语句是条件语句和循环语句。条件语句以**IF**语句为代表，循环语句以**LOOP**为代表。可举几个例子来说明ADA结构语句的特点及其源程序的书写格式：

例1.1　用A表示B的绝对值

①　用单分支**IF**

```
A : = B;
IF  B < 0  THEN
A : = - B;
END  IF;
```

② 用对称IF

```
IF  B < 0  THEN
        A : = - B;
        ELSE
        A : = B
END  IF;
```

从上面的例子中初步看到：

1.与PASCAL相同，ADA中赋值操作也由符号：＝表示。

2.在ADA语言中也采用；号，但分号的用法与PASCAL有所不同。ADA中的分号仅仅表示语句的结束，因此不必象在PASCAL中那样小心地使用。分号只要加在每一个语句的末尾即可。

3.在ADA语言中，注释是通过两条短线（双减号）引入的。注释可以放在语句之后，但在一行中，注释后面不得再接语句。例如：

```
A : = A + 1 ;  - - 将A的值加1
```

关于循环语句，LOOP允许建立一个无条件循环，其格式如下：

```
LOOP
    语句列;
END  LOOP;
```

要跳出这个循环，需要在循环中引入条件，使之变为条件循环。这可以通过**EXIT**语句或**EXIT WHEN**语句来实现：

```
LOOP
    语句列；
EXIT  WHEN条件；
END  LOOP；
```

处于语句列之后的**EXIT WHEN**语句与Pascal中的**REPEAT UNTIL**（ADA不存在这一语句）语句功能相同。

通过**FOR……LOOP**结构也可以建立一个条件循环。

例1.2 求一维矢量V前十个元素之和

```
SUM：＝0；
FOR I IN 1 . . 10 LOOP
SUM：＝SUM＋V（I）；
END  LOOP；
```

例1.1和1.2只是ADA语言的程序段。可以由它们构成函数（**FUNCTION**）、过程（**PROCEDURE**）以及定义自身数据的分程序（**BLOC**），进而构成完整的ADA程序。

1.2 注、说明与可执行语句

通常在高级语言中把语句分为可执行语句与不可执行语句（或说明语句）。这种分类方法同样存在于汇编语言中。汇编语言中不可执行的语句是指导汇编器工作的命令（伪指令）。

由于Pascal语言有丰富的数据类型，说明语句有了很大的发展。这一点在ADA中被进一步强化。对于数据的定义十

分细致，以致从编译器的角度来看，一条说明的出现隐含着一种真实的处理。我们把这种处理过程叫做**加工（ELABORATION）**。对于说明的加工，一部分在编译时进行，另一部分则在程序运行时进行。

除可执行语句、说明语句之外，ADA中还包括给编译器提供的简单指令，称为**参注（PRAGMA）**。由于参注的存在，程序员可以对时空比例作灵活的选择，例如

```
PRAGMA  OPTIMIZE  (SPACE);
```

要求对生成码实行优化以节省存贮空间。

这样，在ADA语言中，语句就有三个等级：参注；说明语句与可执行语句。

1.3 数据类型管理

首先研究一个例子。我们要处理一周的各天，若采用Basic、Fortran，或PL/I就不得不用数码去一一对应。例如：1对应于星期一，2对应于星期二……。采用这种对应关系，由于汇编器无法去检验数值的有效性，赋值总是可以进行的。

Pascal第一个用定义数据类型的方式向数据抽象化的概念迈出了一步。它允许用户根据所要求的特性定义数据，而不是根据某种内部数字表达方式定义数据。例如，在Pascal中我们可以定义一个类型DAY：

```
TYPE  DAY = (MON, TUE, WED, THU,FRI,
          SAT, SUN);
VAR   TODAY : DAY;
```

变量TODAY属于DAY类型，所以下述语句都是非法的：

TODAY：＝1；

TODAY：＝JAN；

因为编译器要检验赋值语句两边的一致性。除赋值语句外，在Pascal中还可以有：

IF TODAY＝WED **THEN**……；

或

FOR TODAY：＝MON **TO** FRI **DO**……；

等语句。

遗憾的是Pascal就此止步，从而大大限制了类型定义的用途。譬如：

TYPE DAY＝（MON，TUE，WED，THU，
FRI，SAT，SUN）；
WEEKEND＝（SAT，SUN）；

在PASCAL中是非法的。二个类型不相容，因为SAT和SUN同时属于这两个类型。另外PASCAL中的子界和实数处理也受到一定的限制。

ADA在类型管理方面比PASCAL更加强化和完善。一方面每一个数据具有唯一的类型，决不能对某种类型的变量赋予其他类型的值。另一方面，我们将看到，在ADA中使用派生型、子类型以及**重载**（**OVER LOAD**）弥补了PASCAL语言的缺陷。

1.3.1 派生型

派生型是由下面的格式定义的：

TYPE T **IS NEW** S；

它表示由类型S派生T。S叫做父型（**PARENT**），T称为S的派生型。T具有S所有的字面值（**LITERAL**）和对S所定义的一切运算。然而父型和派生型是不同的类型，赋值时必须进行类型转换。在ADA中类型转换是使用类型名实现的。例如：

```
X : T;
Y : S;
```

在上述变量说明中，X为T类型，Y为派生型S，因此

```
X : = Y;
Y : = X;
```

都是非法的。然而

```
X : = T (Y) ;
Y : = S (X) ;
```

是合法的。

使用派生型，可以避免逻辑上属于不同种类的对象的运算。例如我们要清点苹果和桔子的个数，同时又要避免把二者混淆，这时用户可以定义两个整型的派生型APPLE—NB和ORANGE—NB：

```
TYPE  APPLE—NB  IS  NEW  INTEGER;
TYPE  ORANGE—NB  IS  NEW INTEGER;
A :     APPLE—NB;
B :     ORANGE—NB;
I :     INTEGER;
```

那么：

```
A : = 0 ; —是正确的，苹果数为零
A : = A + A; —是正确的
```

I：＝A＋B；--是非法的

I：＝INTEGER（A）＋INTEGER（B）；

最后这个算术表达式是正确的。在这里我们可以看到：类型名（如上式中的INTEGER）可以象函数名一样加以使用，将某一变量转换成为指定类型。

使用派生型，还可以实现区间约束。例如：

TYPE INT **IS NEW** INTEGER

RANGE—1000 · · 1000；

也可以简化为：

TYPE INT **IS NEW RANGE**—1000 · · 1000；

我们可以定义许多派生型，它们具有相重迭的区间或相同的数值，这也是**重载**的含义之一。例如：

TYPE COLOR **IS**(GREEN, ORANGE, RED)；

可以对十字路口的交通灯定义成COLOR的派生型：

TYPE LIGHT—OF—CROSS—ROAD **IS**

NEW COLOR；

以上两个枚举型具有相同的元素。我们可以使用类型名区别不同类型中的相同元素。例如：COLOR′（RED）表示颜色中的红色，而LIGHT—OF—CROSS—ROAD′(RED)表示交通灯中的红色，两者没有任何含义上的模糊性。

1.3.2 子类型

子类型也称子界，它是在预定义的类型上加一个约束（如区间约束）。子类型的元素保持其父型的一切特性，并且在赋值时没有必要进行类型转换。也就是说，定义子类型并没有引进新的类型。例如：

SUBTYPE POSITIF **IS** INTEGER
RANGE 1 . .INTEGER′ LAST;

这时，POSITIF是整型，其区间为1至INTEGER′ LAST。INTEGER′ LAST是类型属性。每一个类型T，它的第一个元素叫做T′FIRST，最后一个元素叫做T′LAST。在类型T中的一个元素X有一个后续值T′SUCC (X) 和一个前导值T′PRED (X) 。

1.4 实 数 处 理

我们知道在Pascal中仅有的标准数值类型是整型和实型。对于Pascal的分析表明，不能调整数的精度是该语言不可移植的一个重要因素。

其他的语言提供了一些解决办法。如FORT-RAN允许用户选择单精度和双精度，PL/1允许用户规定有效数字的位数。ADA综合了这两种解决办法。对于整型和实型，ADA具有标准型、长型和短型，即：

INTEGER：标准整型，
SHORT—INTEGER：短整型，
LONG—INTEGER：长整型，
FLOAT：标准浮点型，
SHOPT—FLOAT：短浮点型，
LONG—FLOAT：长浮点型。

但我们建议用户不要直接使用这些标准型，而是使用用户定义的派生型。例如可以写成：

TYPE REAL **IS NEW** FLOAT;

如果所使用的机器的 FLOAT 类型精度不够，则可以写成：

TYPE REAL **IS NEW** LONG—FLOAT;

ADA提供的第二种解决办法是由用户规定有效数字的位数。例如：

TYPE REAL **IS DIGITS** 10;

这时由编译器选择适当的类型（标准型，短型或长型）以保证十位数字的精度。

另外，ADA允许定义实数子界（在Pascal中是不允许的），同时要给出所希望的步长。例如：

TYPE NUMBER **IS DELTA** 0.01 **RANGE** 10.0 . . 100.0;

定义了由10到100步长为0.01的实数。但是实数子界既不能作为数组的下标也不能作为循环变量。

1.5 复合类型

关于复合类型，ADA同Pascal一样具有数组和记录型。但ADA对复合类型的处理有许多重要的发展。

1.5.1 数组

在Pascal中，数组的维长是固定的，没有可变维长的数组。ADA弥补了这一缺陷。其数组的说明可以是带约束的，也可以是不带约束的。例如：

TYPE VECTOR **IS ARRAY** (INTEGER **RANGE** 1 . . 3) OF FLOAT;

```
TYPE MATRIX IS ARRAY (INTEGER
        RANGE < >, INTEGER RANGE
        < > ) OF FLOAT;
```

VECTOR的维长是带约束的，而MATRIX的维长是不带约束的。属于MATRIX类型的对象，其维长可以按照以下方法确定：

① 如果MATRIX是一个子程序的形式参数，在调用此子程序时，由实际参数确定它的维长。

② 在用MATRIX这一类型来描述一个下标带约束的对象的同时确定它的维长。或者根据这个对象的初始值推断出它的维长。例如：

```
M:  MATRIX (1 . . 2, 1 . . 3);
N:  CONSTANT MATRIX := ((0.0, 0.0),
        (0.0, 0.0));
```

M为二维，维长分别为2和3。而N为二维，维长为2。

③ 引入MATRIX子类型，确定这个子类型的维长。然后用子类型描述对象。例如：

```
SUBTYPE MATRIX_2_2 IS MATRIX
        (1 . . 2, 1 . . 2);
N: MATRIX_2_2;
```

④ 解决可变维长的另一种办法来源于ADA语言的高度参数化（在带参记录型中还将研究这一问题）。即对于带约束的类型，仍可以写成：

```
TYPE TABLE IS ARRAY (1 . . N) OF
        FLOAT;
```

其中N是一个变量，当然在对说明加工时，它必须有确定的

数值，这种数组称为动态数组。

1.5.2 字符串

ADA能够辨别128个ASCII字符，它们组成预定义的CHARACTER型；

TYPE CHARACTER **IS** (NUL, SOH, …'A', 'B', …DEL)；

其中，可打印的字符写在引号' '之间，而控制字符用标识符表示，例如CR，LF等等。

预定义的字符串型为

TYPE STRING **IS ARRAY** (NATURAL **RANGE** ⟨ ⟩) **OF** CHARACTER;

其中NATURAL是预定义的整型子界型；

SUBTYPE NATURAL **IS** INTEGER **RANGE** 1 . . INTEGER' LAST;

因此可以用下述方式定义常数字符串：

X : **CONSTANT** STRING : = "HALLO"；

或

X : **CONSTANT** STRING : = ('H', 'A', 'L', 'L', 'O')；

此时X的界为1 . . 5。所有的字符串都有一个实际的长度。例如：对于字符串X其长度记作：X'LENGTH。

在Pascal（苏黎世文本）中一个棘手的难题，即对字符串的处理。这个问题在ADA语言得到了较好的解决：ADA中存在的连接符（&）可以用于截取字符串的片段。譬如对于上面的字符串X，X（2 . . 4）为"ALL"。

1.5.3 记录

ADA中的记录与PASCAL中的记录基本上相同，但是有两点改进：

其一，ADA允许定义缺省初值。例如：

```
TYPE COMPLEX IS RECORD
        RE, IM : FLOAT : = 0.0;
        END RECORD;
```

就这样可以保证用户所定义的一切复数在不加说明的情况下其初始值为零。

其二，ADA中的记录可以是带参记录。带参记录是Pascal中变体记录的改进。例如：

```
TYPE GENDER IS (M, F);
TYPE PERSON (SEX : GENDER) IS
RECORD
    BIRTH_DATE : DATE;
        CASE SEX IS
        WHEN M⇒BEARDED : BOOLEAN;
        WHEN F⇒CHILDREN : INTEGER;
        END CASE;
END RECORD;
```

记录型PERSON带有一个参数SEX(SEX为枚举型 GENDER)。SEX称为**判别式**，PERSON称为**带参记录**。在本例中，DATE被假定为已经定义过的记录型。根据上述定义可以建立下述形式的变量说明：

```
SUBTYPE FEMALE IS PERSON (F);
```

```
JULIE : FEMALE : = (F, (10,DEC,1980), 0),
WANG : PERSON (M) : = (M, (15, MAY,
                1950) , FALSE) ;
```

在带参记录中，可以给判别式缺省初值，例如：

```
TYPE PERSON (SEX : =F) IS……;
```

最后要说明的是：这种参数化的手段为改变字符串的长度提供了另一种解决办法。

例1.3 可变长度字符串示例

```
TYPE STRING (LMAX : INTEGER
        RANGE 1 . . 256) IS
RECORD
LENGTH : INTEGER RANGE 0 . . 256 : = 0;
CONTENT : ARRAY (1 . . LMAX)
            OF CHARACTER;
END RECORD;
```

1.6 子 程 序

现在我们将逐步接触到ADA一些新的概念。它与Pascal一样，也存在有函数和过程。但是ADA对于给出参数和回授参数做了更明确的区别。即参数有三种结合方式：输入（IN）、输出（OUT）和输入/输出（IN/OUT）。

例1.4

```
PROCEDURE INSERTION (V : IN INTEGER;
        INTO : IN OUT PTR) IS
    T: PTR;
```

```
BEGIN
    T: NEW OBJECT (V, INTO);
    INTO := T;
END INSERTION;
```

这个过程是在INTO指向的OBJECT对象之前把信息V插入链表。可以看出：指针型的处理方法与Pascal极为相似。上述过程的说明部分可以写作：

```
TYPE OBJECT; --不完整的说明
TYPE PTR IS ACCESS OBJECT;
TYPE OBJECT IS
    RECORD
        VALUE : INTEGER;
        NEXS : PTR;
    END RECORD;
```

对象的产生可以由类型名冠以键词**NEW**来实现，

```
T : = NEW OBJECT (0, NULL);
```

这个语句将产生一个由PTR型变量T指向的对象。其值为零，且没有接续对象。

ADA过程的特点是：

① 参数**IN**的数值可以通过缺省值来确定，因此在调用过程时可以省略**IN**参数。

② 调用过程时，可直接写出形参名和实参名的对应关系。对于例1.4的过程，可以有如下调用语句：

```
INSERJION (5, INTO⇒LJST);
```

③重载。在节1.3中，已经看到几个枚举型包含相同常数的例子。同样，几个子程序也可以在相同名字下共存。即

使这些子程序的参数不同，也只是类型不同。我们把这种功能称为子程序的**重载**。注意这里所指的共存是在同一个调用等级上，它不同于在分程序中对一个标识符的再定义。

在ADA语言中，**重载**这一概念应用到许多方面。首先是对经典运算符的重载。在这一点上它又填补了PASCAL的一项空白。例如，在ADA和PASCAL中都可以定义复数类型：

ADA	PASCAL
TYPE COMPLEX **IS**	**TYPE** COMPLEX =
RECORD	**RECORD**
RE, IM : FLOAT;	RE, IM : REAL
END RECORD;	**END**;
Z1, Z2, Z3 :	**VAR** Z1, Z2,
COMPLEX;	Z3 : COMPLEX;

关于COMPLEX的定义两者是相似的。但是在PASCAL中，尽管Z1、Z2和Z3已被说明为复数，下述复数加法都是非法的：

```
Z3 := Z1 + Z2;
```

也就是说PASCAL中途止步。它允许用户定义复数类型，但不允许两个复数直接相加。为了达到直接相加的目的，用户不得不定义如下的一个函数：

```
FUNCTION ADD (A, B : COMPLEX) :
            COMPLEX;
    BEGIN
        ADD.RE := A.RE + B.RE;
        ADD.IM := A.IM + B.IM
    END;
```

然后通过调用函数实现Z 1 和Z 2 相加，即：

```
Z 3 : = ADD (Z 1 , Z 2 );
```

而在ADA中，我们可以将加法运算符“+”重载。

即：

```
FUNCTION "+" (X, Y : COMPLEX) RETURN
            COMPLEX IS
    BEGIN
    RETURN (X.RE + Y.RE, X.IM + Y.IM) ;
END "+" ;
```

这样，直接将两个复数相加。

```
Z 3 : = Z 1 + Z 2 ;
```

便成为合法的。这个例子在讨论**程序包（PACKAGE）**时还会用到。

同样，我们可以将运算符“*”重载。如果定义一个矩阵乘法算符。

```
FUNCTION "*" (X, Y : MATRIX) RETURN
            MATRIX IS
      ⋮
END "*" ;
```

可以象在APL语言中那样，把矩阵之积简单地写作：

```
C : = A * B;
```

1.7 类属类型 (GENERIC)

我们知道，任何一种运算都是对某些确定类型的操作数定义的。如果试图将它用到其它类型的操作数上，必然会产

生错误（这在ADA中叫做**异常—EXCEPTION**）。根据上面介绍的重载概念，当然可以把一个运算符用于所有的类型，但是必须对可接受的类型每种组合都要定义一个重载。显然这样做是繁锁枯燥的。

ADA引入**类属（GENERIC）**这一概念来解决这一问题。它允许先写出一个模型。例如：为了实现两个对象的互换，可以写成：

```
GENERIC TYPE T IS PRIVATE;
PROCEDURE SWAP (X, Y : IN OUT T IS
    Z : T;
BEGIN
    Z : = X;
    X : = Y;
    Y : = Z;
END SWAP;
```

这个过程叫做**类属过程**。其参数（T）是一个类型，称为**类属类型**。但是类属过程不能直接调用，使用时必须经过衍生（**INSTANTIATION**）。例如将两个矩阵进行交换，衍生过程为：

```
PROCEDURE SWAP--MAT IS NEW SWAP
                    (T⇒MATRIX) ;
```

然后调用语句写为：

```
SWAP—MAT (A, B) ;
```

1.8 程序包（PACKAGE）

现在介绍ADA引入的一个最强有力的工具——程序包。

如果把1.6节中关于复数的运算加以扩充，即可得到一个复数处理系统程序。这样的程序在 ADA中称为程序包。程序包由两部分组成：程序包式与程序包体。前一部分定义部分、它列举程序包的内容、描述它们的特性。后一部分为程序包的实体，规定程序包的功能。例如：复数程序包的形式如下：

```
PACKAGE COMPLEX—NUMBER IS
    TYPE COMPLEX IS PRIVATE;
    I : CONSTANT COMPLEX;
    FUNCTION"+"(X, Y : COMPLEX) RETURN
              COMPLEX;
    FUNCTION"-"(X, Y : COMPLEX) RETURN
              COMPLEX;
    FUNCTION"*"(X, Y : COMPLEX) RETURN
              COMPLEX;
    FUNCTION "/" (X、Y : COMPLEX) RETURN
              COMPLEX;
    FUNCTION CMPLX(RP,IP:FLOAT) RETURN
              COMPLEX;
    FUNCTION REAL(X : COMPLEX) RETURN
              FLOAT;
    FUNCTION IMAG(X : COMPLEX) RETURN
              FLOAT;
PRIVATE
    TYPE COMPLEX IS
        RECORD
            RE : FLOAT;
```

```
        IM : FLOAT;
    END RECORD;
  I : CONSTANT COMPLEX : =( 0 ,  0 , 1 ,0);
END COMPLEX_NUMBER;
--以下为程序包体:
PACKAGE BODY COMPLEX_NUMBER IS
  FUNCTION "+"(X, Y : COMPLEX) RETURN
                    COMPLEX IS
    BEGIN
      RETURN (X.RE + Y.RE,X.IM + Y.IM);
    END "+" ;
    .
    .
    .
  FUNCTION "*"  (X, Y : COMPLEX)
      RETURN COMPLEX IS
    BEGIN
      RETURN (X.RE*Y.RE - X.IM*
              Y.IM, X.RE*
              Y.IM + Y.RE*X.IM) ;
    END "*" ;
    .
    .
    .
  FUNCTION CMPLX (RP, IP : FLOAT)
          RETURN COMPLEX IS
    REGIN
      RETURN (RP, IP) ;
```

```
        END CMPLX;
        ⋮
    FUNCTION REAL (X : COMPLEX) RETURN
                ELOAT IS
        BEGIN
            RETURN X.RE;
        END REAL;
        ⋮
NED COMPLEX—NUMBER;
```

下面是在分程序（BLOC）中使用复数程序包的一个例子：

```
DECLARE
    USE COMPLEX__NUMBER;
    A, B : COMPLEX;
    C, D : ELOAT;
BEGIN
    A : = CMPLX (1, 0, -5, 0);
    B : = A + 1;
    D : = 1, 0;
    C : = REAL (B) + D;
END;
```

程序包是ADA中最有用的工具之一。当把一组逻辑上相联系的资源，（数据类型，子程序……）付诸使用的时候，可以建立一个程序包。例如：标准输入/输出过程组成一个

INPUT—OUTPUT程序包。我们将在第五章详细介绍这个程序包，现在仅介绍它的两个基本过程GET和PUT：

GET（变量）—在键盘上读一个变量，

PUT（表达式）—在屏幕上显示表达式的值。

通过适当的重载，GET能够在键盘上读入用户定义的枚举型元素。例如：

TYPE DAY **IS** (MONDAY, TUESDAY, ……);

TODAY : DAY;

那么当执行GET(TODAY)时，如果在键盘上键入TUSEDAY，系统将对TODAY赋予TUSEDAY。这在PASCAL中是不可能的，它的输入/输出过程只能接受标准类型，从而大大限制了在PASCAL中使用类型的好处。

与PASCAL相比，ADA唯一缺少的类型是集合型(SET)，这种类型需要用布尔数组来处理。

1.9 私有类型 (PRIVATE)

关于复数程序包的例子中所涉及到的**私有类型**还需要作以下说明。私有类型是ADA根据不同的人有不同的要求这一特点建立的。仍以复数程序包为例，对于用户来说，他所关心的是有一个COMPLEX类型，用它可以实现复数的加、减、乘、除……等等。至于完成这些运算的方式，对用户来说是无关紧要的。但是对于数学库的建立者来说，运算方式是至关紧要的。例如，他希望建立极坐标中的复数，那么私有部分应写为：

PRIVATE

```
TYPE COMPLEX IS
    RECORD
        R : FLOAT;
        THETA : FLOAT;
    END RECORD;
I : CONSTANT COMPLEX : =(1.0, PI/2.0);
```

--设PI已定义过在程序包体中，函数也必须作相应的变化，例如：

```
FUNCTION "*" (X, Y : COMPLEX ) RETURN
        COMPLEX IS
BEGIN
    RETURN (X.R*Y.R, X.THETA
            +Y.THETA);
    END "*";
```

这些变动对用户使用复数程序包来说没有丝毫影响，这也是把复数定义成私有类型的原因之一。

ADA的私有类型也是非常重要的工具，它向数据抽象化又迈出了一步，允许用户按照数据所具有的性质去定义数据，而不是按照实现方式去定义。

1.10 异常和任务

以上，我们看到ADA的某些方面是对PASCAL所开创的数据管理手段的延伸。但是应该强调的是，ADA并不是PASCAL的简单的延伸或扩充。本节先简单地介绍一下**异常**（错误处理）、**任务**（并行处理）等概念。

1.10.1 异 常（EXCEPTION）

在ADA语言中，一些所谓的标准错误能引发相应的预定义的异常。此外，异常也可以由用户自行定义。例如：奇异矩阵异常定义如下：

SINGULAR—MATRIX：EXCEPTION;

这种异常的引发由**RAISE**语句来实现。例如：

```
IF D = 0.0 THEN RAISE
                SINGULAR—MATRIX;
```

在异常引发之后，异常的处理是由下述的程序段给出的：

```
EXCEPTION
WHEN SINGULAR_MATRIX⇒
适当的语句列;
```

1.10.2 任务（TASK）

任务定义的是可以并行执行的处理过程。ADA允许用户定义一些任务，这些任务必须在调用程序结束之前全部完成。譬如一个家庭在机场下飞机之后，需要提取行李，租用汽车和订旅馆。这三件事可以同时进行：父亲负责订旅馆，母亲去租汽车，孩子照管行李。必须在这三件事都完成以后才能离开机场。于是可以写成：

```
PROCEDURE ARRIVAL IS
    TASK BAGAGE IS
    ⋮
    END;
    TASK BODY BAGAGE IS
```

```
        ⋮
        ⋮
    END BAGAGE;
    TASK TAXI IS
        ⋮
        ⋮
    END;
    TASK BOBY TAXI IS
        ⋮
        ⋮
    END TAXI;
BEGIN
    HOTEL;
END ARRIVAL;
```

各项任务之间的同步由语句**ACCEPT**和**ENTRY**实现。

以上对ADA语言做了简单介绍。由此可以看到：

——ADA语言的特点和强有力的功能。

——ADA是如何延伸 PASCAL 的概念并在它的基础上发展起来的。

第二章　程序结构与可执行语句

在读者对ADA语言有所了解的基础上，从本章开始介绍ADA语言的细节。先介绍ADA语言的字符集、程序结构以及可执行语句。

2.1　ADA 的 字 符 集

ADA语言的基本字符集由下列几部分组成：

——大写字母：A B……Z

——数字：0 1 …… 9

——专用符号" * % ' ( ) ※ + , - . / : ;
< = > _ | &

——空格

注意区分 - （减号或短线）与_ （下划线），数字0与字母O。在本书中不使用符号上面加线的方法（如把零记做∅），根据上下文总是可以区别它们的。

ADA的扩充字符集还包括：

——小写字母：a b……z

——其他专用符：！ $ ？ @ [\] ^ | {}~

除字符串外，小写字母与其对应的大写字母是等效的。这些字母还可以用另外的形式来表示。例如：

ASCII.LC_X　表示字母X

ASCII.DOLLAR　表示美元符 $

本书仅限于使用ADA的基本字符集。

由字符组成字（或称词汇单位）。ADA的字有如下几类：

①语言保留的键词。如：**IF**，**PROCEDURE**；

②预定义或用户定义的标识符。如：PUT，COMPLEX；

③数字或字符字面值。例如：3.1416，'A'，"ZOZO"；

④注释。例如：--IT IS A COMMENT；

⑤定界符。例如：&' ( ) * + -/' . : ; < = > |

或者是组合符。如：⇒ .. ** : = /= >= <= « » <>

两个字之间至少由一个空格（或者回车）分开。反之在一个字之间（除字母字面值以外）不得插入空格或回车。

ADA语句的书写格式比较自由。几个语句写成一行或者一个语句占用几行都是允许的。

语句以分号（；）作为结束标志。尽管如此，我们仍然建议最好是一行一个语句，并且仔细地分页，以增加程序的易读性。

在PASCAL或PL/I中，凡在能够出现空格的地方，都可以插入注释。ADA则比较严格，因为它的注释只有开始符（双短线）没有终止符，所以只有在一行最后一个语句的终止符（；）之后，方可插入注释。下面的注释方式是不允许的：

语句开始--注释语句接续部分；注释的正确形式为：

语句；--注释

当然注释可以单独占据一行。例如：

--IT IS A COMPLEX NUMBER。

一行全是短划线可以看做是注释的特殊情况（因为头两个符号是短线）。这种注释可以做为分页标志，以便把不同的模块明显地加以区分。

2.2 程序的一般结构

本节我们不涉及分离编译，这个问题留在以后有关章节中讨论。ADA中每个程序都是一些编译单位（过程、程序包、任务或类属）或一些分程序的集合。这些编译单位是可嵌套的。即一个过程可以包含另外的过程，一个任务可以包含一些过程，等等。每一个模块都由明显区别的两部分组成：

一说明段（如程序包式，任务式等）。它由某些关键词所代表的模块的特征字引入。例如：

PROCEDURE 引入过程

FUNCTION 引入函数

PACKAGE 引入程序包

TASK 引入任务

DECLARE 引入分程序

等等。换句话说，在每一个模块的首部都有一个特征字。说明段由一系列说明组成，它的作用是描述本模块中要操作的数据。

一执行段（如程序包体，任务体等）。它包括一系列可

执行语句，由它们确定数据的处理过程，并以 **BEGIN** 开始，**END**终止。

2.2.1 标识符

ADA语言中的标识符，第一个字符为字母，其后是字母、数字或下划线的组合(但不能几个下划线连用)。例如：

PROCEDURE

GET

MY__PROGRAM

H_2SO_4

HENRI__8

标识符字符的个数不受限制。原则上讲，任何长度的标识符都是有意义的。但是我们建议字符数还是应该限制在一个合理的范围之内。

ADA的标识符包括：

①语言的关键词（见附录Ⅰ）。

②预定义的标识符（用户可以重新定义）。

③用户定义的标识符。

除此之外，在表达式中还可以出现：

④带下标的标识符。例如：M（5）表示数组M的第五个元素。

⑤数组片标识符。例如：M（3 . . 6）表示M的第三个至第六个元素。

⑥限制名标识符。例如：SYSTEM·MAX__INT表示程序包SYSTEM中定义的最大整数。

⑦属性标识符。例如：ZOZO'SIZE表示ZOZO所占

据的字节数。

标识符主要用作常数、变量、类型、过程、程序包、异常和任务的名。

现在研究标识符的作用域。这个概念来源于PASCAL。这里只做简单说明，在第四章中再详细地介绍。一个标识符在程序中可见的区域称为该标识符的作用域。决定作用域的一般规则（称为可见性法则）是：一个对象若在某个模块中定义，那么这个对象在该模块中以及它所包含的诸模块中是可见的(对象作为全局对象)。相反，在定义模块以外的模块以及包含定义模块的模块中是不可见的（对象作为局部对象）。但是，如果在模块A中定义了变量X，而在A包含的模块B中再定义X，那么B中的X与A中的X是有区别的。我们把B中的X记作X′。在这种情况下，A中的X在B以及B所包含的模块中变为不可见的，这种情况称为“掩蔽”。

例2.1 在下面的嵌套程序段中，给出标识符的作用域。

```
PROCEDURE A;
X : ……;
Y : ……;
PROCEDURE B;
X : ……;
Z : ……;
T : ……;
    BEGIN
      ⋮
    END B;
```

```
PROCEDURE C;
Z : ……;
    BEGIN
      ⋮
    END C;
BEGIN
      ⋮
END A;
```

解：我们把B中的 X记作 X′，把C中的 Z记作Z′，各个标识符的作用域如图2.1所示：

```
PROCEDURE A，标识符
    X; ……;
    Y; ……;
PROCEDURE B;
    X; ……;
    Z; ……;
    T; ……;
BEGIN
END B
PROCEDURE C;
    Z; ……;
BEGIN
      ⋮
END C;
      ⋮
BEGIN
      ⋮
END A;
```

可见域

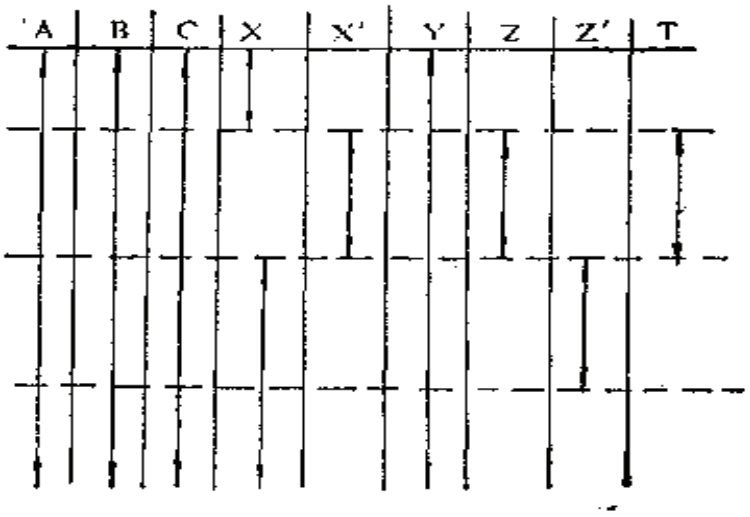


图 2．1　各标识符的作用域

2.2.2 字面值

一个常数可以用一个标识符表示，也可以用字面值表示。字面值用于表示某一给定类型的变量或常量的显式值，它与习惯上采用的形式是相同的。字面值可分为数字面值、字符串（或字符）字面值、枚举字面值和NULL。例如：

```
3.14159          --数字字面值
CLUB             --枚举字面值
'A'              --字符字面值
"HALLO"          --字符串字面值
```

数字字面值可以是十进制（基为10），也可以是其他进制。十进制的形式与其他语言是相同的。例如：

```
254          -1609      (整型)
12.34        -0.459     (定点型)
0.123E -8    -0.52E18   (浮点型)
```

唯一的区别是，ADA允许在数字字面值中插入下划线作为分位号，以改善数字的可读性。例如：

```
1_619_754_312
3.141_592_65
```

ADA能够把数字字面值表示成其他进制的形式，称为带基数。例如：

```
基#数#
```

或者　　基#数#E幂次

按照这种表示法，

2#0111_1111#或16#7F#代表127。

16#E#E1代表$14\times16^{1}=224$，

16#FF.F#E1 代表 $(FFF)_{16}=4095$，

3 #12#代表 5 。

基可以选 2 至16之内的数。对于10以上的基，10、11、12、13、14和15分别由A、B、C、D、E和F来表示。

字符字面值是夹在单引号间的单个字符（如：‘A’，‘*’），字符串字面值是夹在双引号间的一串字符（如：“HALLO”）。如果要想在字符串中包括一个双引号 ”，那么只需要将 ” 重复一次。例如：“O” “CLOCK” 中间的二个双引号并不代表字符串的结束，而代表其中有一个符号 ”。“” 表示一个空字符串。

字符串要求写在同一行中，否则需要在接续部分前面加一个接续符&。如果想 把 一 个 字符串和一个控制符连接起来，必须使用接续符和控制字符的符号名。例如：“HALLO” &ASCII.CR表示在字符串之后回车。

在ADA中‘A’ 与 “A”是有区别的。前者是一个标量，后者是单字符的字符串。

2.2.3 算术表达式

在ADA语言中，算术表 达式表示要进行的算术运算。它是基本的表达式，由操作数和运算符组成。

操作数可以是字面值，也可以是由标识符表示的常数、变量或函数调用。

变量又可以以不同的形式出现：

①简单变量。如：M，MY_VARIABL，等等；

②下标变 量。如：V（3）表示矢量V的第三个元素，MAT（5）（4）表 示 矩 阵 MAT 第 五行的第四个元素。

CHESSBOAD（12，12）是表示矩阵元素的另一种办法。下标变量又可以是一个算术表达式。如V（3 *I + 1）；

③数组片。如V（5 · ·12）表示数组第五至第十二个元素，而V（5 · ·12）（9）与V（9）的意义相同；

④限制变量。例如：

BIRTH__DATE · DAY表示记录BIRTH__DATE中的DAY项。

PTR · **ALL**表示指针PTR所指的对象（在PASCAL中记作PTR↑），

⑤属性。每一个类型的变量都具有一些预定义的属性，例如：

VAR′ADDRESS表示存放变量VAR的第一个字节的地址。

T′FIRST表示类型T的第一个元素。

X′LENGTH表示X的元素数目。

算术表达式中的函数调用是相当方便的，例如：ABS（X）表示X为绝对值。ABS为标准标识符。

用于算术表达式的运算符有：

* *（幂运算）

*（乘法）

/（除法）

MOD（求模）　　**RED**（余数）

+（加法）

-（减法或负号）

ADA中的运算顺序是：先算括号，再按运算符的优先级运算，同一优先级内则按自左向右的规则运算。算符的优

先级与FORTRAN或PL/I相同，如果按照等级递降排列，则先是＊＊，然后是并列的＊、/、MOD与REN，接下去是单目运算符＋与－，最后为双目运算的＋与－。单目运算是指仅有一个操作数（位于运算符的右侧）。例如－×（取×相反的值），而A－B中的减号为双目运算符。

注意A＊＊B＊＊C是不允许的。在这种情况下必须使用括号：A＊＊（B＊＊C）。括号用来指明需要首先计算的表达式。

以上这些规则可以由下面的定义来概括（其中::＝读作“由……组成”，｜读作“或”）：

初级量::＝〈字面值〉｜〈变量〉｜〈函数调用〉｜〈类型转换〉｜〈限制表达式〉｜＜（表达式）＞

因子::＝＜初级量＞｜＜初级量＞＊＊＜初级量＞

项::＝＜因子＞｜＜因子＞＜乘法算符＞＜项＞

简单表达式::＝＜项＞＜加法算符＞＜简单表达式＞

算术表达式::＝＜简单表达式＞｜＜单目运算符＞＜简单表达式＞

以上定义代表了表达式运算的正确顺序。应该指出：ADA语言没有规定一项运算中两个操作数的求值顺序。因此表达式的值必须与两个操作数的求值顺序无关，否则将产生错误。还要指出：一次运算的两个操作数必须为同一类型（并且对于这个类型来说，这种运算是有定义的），或者运算符对所涉及的类型是重载的。

接续符&也是一个运算符，其功能用来实现连接（即两个字符串的合并。例如：“HAL”&“LO”＝“HALLO”）。

另外接续符也可以实现元素为同类型的一维数组的连接。从优先级来讲，&与双目＋与－运算符相同。

2.2.4 逻辑表达式

首先定义关系符。它用于说明两个算术表达式的关系，其结果是布尔量。如果关系成立，结果为TRUE，反之结果为FALSE。例如：3＜5为FALSE。关系符包括：

=	（相等）
/=	（不等，注意与BASIC不同，不使用＜＞）
＜	（小于）
＜=	（小于等于）
＞	（大于）
＞=	（大于等于）

所有关系符都有相同的优先级，并且在算术运算符最低优先级之下。因此，对于下述逻辑表达式：

＜算术表达式1＞＜关系符＞＜算术表达式2＞应首先计算两个算术表达式，然后再判断所指出的关系是否成立。

例2.2 指出（1＜5）（2＞3）与1＜（5＜2）＞3有何区别？

解：对于第一个逻辑表达式，首先计算1＜5，其结果为TRUE，而2＞3为FALSE，剩下的是TRUE＜FALSE，其结果为FALSE。最后一个判断是由布尔型的定义决定的，布尔型为预定义的枚举型：

TYPE BOOLEAN **IS** (FALSE, TRUE)；FALSE在TRUE之前，因而FALSE＜TRUE。

第二个逻辑表达式是不正确的。（5＜2）是布尔型，

接着要计算的是 1 （整型）＜布尔量，这种计算是非法的。

IN和**NOT IN**是具有另外一种意义的关系符。它表示属有关系。其结果为布尔量。它们的优先级与关系符相同。NOT IN与IN的意义相反。IN所表示的属有关系逻辑表达式格式如下：

〈元素〉**IN**〈区间〉

或者

〈元素〉**IN**〈子类型〉

例如：　DAY **IN** MONDAY··FRIDAY

K **IN** NATURAL

第一个关系表达式为 TRUE 的条件是 DAY 确实包括在MONDAY至FRIDAY之内。第二个关系表达式为 TRUE 的条件是K恰巧在子类型 NATURAL 所确定的约束之内。如果条件不满足，则关系表达式的值为FALSE。

现在我们来说明逻辑算符。使用逻辑算符可以组成表示逻辑关系的逻辑表达式，其结果为布尔量。所有的双目逻辑算符具有相同的优先级，并在关系符之下。因此，对于〈关系 1〉逻辑算符〈关系 2〉应当先确定两个关系的真伪，接着进行逻辑算。不要忘记在确定每个关系的真伪时，首先计算关系中所出现的算术表达式。

逻辑算符有：

AND　　（与）

OR　　（或）

XOR　　（异或）

AND THEN（与然后）

OR ELSE（或否则）

它们所对应的真值如表2.1所示：

表 2.1　　逻 辑 符 真 值 表

A	B	A **AND** B	A **OR** B	A **XOR** B
FALSE	FALSE	FALSE	FALSE	FALSE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	TRUE
TRUE	TRUE	TRUE	TRUE	FALSE

AND THEN与**OR ELSE**是快速形式的逻辑算符。快速形式逻辑算符允许在能由关系1得出结论的情况下不再计算关系2，例如：

A **AND THEN** B

如果A为伪，不必计算B就可以确定其结果为伪。若A为真则需要计算B，B为真结果为真，B为伪结果为伪。再如：

A **OR ELSE** B

如果A为真，不必计算B就可以确定结果为真。若A为伪则需要再计算B，结果与B的真伪相同。

此外，还有单目逻辑算符：

NOT　（取操作数的非）

它的优先级与单目算符＋和－相同。

运算符、关系符以及逻辑算符的优先级如表2.2所示。

注意：尽管有自左至右的规则，ADA仍然不允许逻辑算符不加括号而连在一起使用。

例2.3　如果没有XOR，应该怎样实现这一逻辑运算？

解：可以采用下述逻辑表达式实现XOR（异或）：

表 2.2　　各 符 号 的 优 先 级

名　称	算　　符	优先级
逻辑算符	AND OR XOR	低
关 系 符	= /= < <= > >= IN NOT IN	
加 减 符	+ − &	
单目算符	+ − NOT	
乘 除 符	* / MOD REM	
幂	**	↓ 高

（A **AND** （**NOT** B））**OR**（B **AND** （**NOT** A））

或　（A **OR** B）**AND**（**NOT**（A **AND** B））

或　**NOT** A = B　（A和B都是布尔量）

例2.4　用另外的方式表达(X> =. 3)**AND**（X< = 10）

解：可以用属有关系表示：X **IN** 3..10

到目前为止，我们已经介绍了ADA语言程序的一般结构、标识符、算术表达式和逻辑表达式。并定义了常用的算术运算。还有一些是与数据类型有关的特殊运算。

2.3　顺序语句和结构常值

顺序语句在整个信息处理过程中完成三类基本操作：

——接受数据；

——对数据进行计算；

——输出计算结果。

数据的输入/输出是通过调用标准程序包中的过程来完成的。在第五章之前，由于还没有介绍输入/输出程序包，我们暂且使用下述过程：

GET（变量名）；

PUT（算术表达式）；

分别完成接受数据和输出计算结果的操作。

现在所剩下的是对数据的计算，本节只讨论其中最基本的处理语句——赋值语句。赋值语句的形式为：

变量：＝算术表达式；

其中赋值标号与其他高级语言一样是非对称的。也就是说，计算算术表达式所得到的结果（可以简化为一个变量）只能赋予：＝左边的变量。

赋值运算的基本约束是两个量必须属于同一类型，决不能对一个变量赋予不同类型的结果。在ADA中没有类型相容的概念，也不存在隐式转换。类型的转换必须明确标出，标出的方式是十分简单的：只要引入转换结果的类型名就行了。例如：

I：INTEGER；

R：FLOAT；

那么，

I：＝INTEGER（R）；

是正确的，它表示把实型量R转换成整型赋予整型变量I。

上述这些性质同样适用于复合型对象的赋值。如数组和记录的赋值。如果A与B属于同一类型的变量，并且如果它们是数组的话维长相等，如果是记录的话结构相同，那么，下述赋值语句是允许的：

A := B;

A中每一个元素的取值和B中所对应的元素一一相等。如果赋予A一个常数，那么这个常数必须是与A的元素数相等的多重常数，我们把这样的常数叫做**结构常值**（AGREGAT）。

假定有一个数组型变量V，其定义为：

```
TYPE VECT 5 IS ARRAY (1 . . 5)
        OF INTEGER;
V: VECT 5;
```

对V赋值有两种写法：

① 赋予定位结构常值。如：

```
V := (1, 0, 1, 0, 0);
```

以类似于（1，0，1，0，0）的形式确定的结构常值，叫做定位结构常值。

② 赋予以名定义的结构常值。如：

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

或者

```
V := (1 | 3 => 1, OTHERS => 0);
```

对于记录型也是相同的。例如：

```
TYPE DATE IS
  RECORD
    DAY : INTEGER RANGE 1 . . 31;
    MONTH: (JAN, FEB, MAR, APR, MAY,
            JUN, JUL, AUG, SEP, OCT, NOV,
            DEC) ;
  YEAR : INTEGER RANGE 1800 . . 2000;
  END RECORD;
```

D：DATE；

对D赋予定位结构常值，例如：

D：＝（15，AUG，1981）；

或者赋予以名定义的结构常值，例如：

D：＝（MONTH⇒AUG，DAY⇒15，
YEAR⇒1981）；

在以名定义的结构常值中，由于赋值是按元素名来对应的，所以元素的顺序无关紧要。

例2.4 在上述例子中，用另外的方式对D赋值。

解：可以对纪录的每一个元素分别赋值。如：

D.DAY：＝15；

D.MONTH：＝AUG；

D.YEAR：＝1981；

本节只讨论了顺序语句的基本语句——赋值语句，下面几节将研究顺序转向语句或者叫做程序结构词句。

2.4 GOTO 语 句

尽管ADA属于结构化语言，它仍然保留了无条件转移语句**GOTO**，只不过将尽量地少使用它罢了。**GOTO**语句的格式是：

GOTO标号；

在有标号的语句中，标号出现在《 》之中。例如：

《LABAS》I：＝I＋1；

⋮

GOTO LABAS;

在特殊情况下需要跳出循环时可以使用GOTO语句（EXIT可以更有效地实现这一功能）。GOTO语句不能实现由外部进入一个复合结构（分程序、IF、CASE或循环），也不能跳出一个子程序体、程序包体或任务体。

2.5 IF 语 句

IF语句的格式是：

```
IF 条件1 THEN
    语句列1；
ELSIF 条件2 THEN
    语句列2；
ELSE
    语句列3；
END IF;
接续语句4；
```

其中条件由一般的逻辑表达式给出（参考2.2.4）。在IF语句的格式中，可以有几个ELSIF（加相应的语句列），也可以没有ELSIF，还可以没有ELSE。IF语句的执行方式如下：

——若条件1满足，执行语句列1，然后跳到接续语句4；

——若条件1不满足且有几个ELSIF，则按顺序检查。碰到条件满足的则执行相应的语句列，然后跳到语句4。其余的条件不再检查（尽管有其他条件还可能是满足的）；

——若条件1和条件2都不满足，执行语句列3，然后

跳到语句 4 。如果没有**ELSE**，直接跳到语句 4 。

总之，在**IF**… …**THEN**… …**ELSE**……**END IF**； 语 句中，**ELSE**相当于**ELSIF** TRUE。语句：

IF…**THEN**…**END IF**；

与 **IF**…**THEN**…**ELSE**…**END IF**；

是IF语句的简化形式。

END IF的写法解决了 PASCAL 中需要特别小心检查分号这一难题，这种检查总是给书写带来麻烦。另一方面，我们可以在子句 (CLAUSE)里直接放上语句列，而用不着象PASCAL那样由BEGIN…END来产生一 个 分程序 (ADA中的分程序另作他用)。

是不是可以使用多层嵌套**IF**呢？我们注意到 子句**ELSIF**的存在已经使得这种多层IF没有多少 用 处，但ADA语言并不排除使用多层IF。

下面介绍一下不太好理解的语句结构问题。整个问题可以归结为弄清**ELSIF**或**ELSE**与哪一个**IF**相 对 应。其规则是简单的：**ELSE**或**ELSIF** 对应于最后遇到的但 还 没 有 出 现 **END IF**； 的那个**IF**。例如：

```
IF A THEN
        ①……;
    IF B THEN
        ②……;
    ELSIF C THEN
        ③……;
    END IF;
        ④……;
```

ELSIF D THEN……;

语句列①到④组成第一个IF的THEN的语句，其中还含有一个IF。ELSIF C与IF B相关。ELSIF D与IF A相关，因为遇到ELSIF D时，IF B已经由它的END IF关闭了。

例2.5 上例中，在什么情况下执行语句列③？接下去又执行什么？

解： 由于A、B和C都是布尔量，我们可以把执行语句列③的条件表示成：

A AND ((NOT B) AND C)

例2.6 写出两种程序用以解一元一次方程AX+B=0

解 ① IF A = 0 THEN

```
        IF B = 0 THEN
            PUT ( "INDETERMINATE" ) ;
            ELSE
            PUT ( "IMPOSSIBLE" ) ;
        END IF;
    ELSE
        X: = -B/A
        PUT ( "THE ROOT IS" ) ;
        PUT (X) ;
    END IF;
```

② IF A /= 0 THEN

```
        X: = -B/A
        PUT ( "THE ROOT IS" ) ;
        PUT (X) ;
    ELSIF B = 0 THEN
```

```
        PUT ( "INDETERMINATE" ) ;
    ELSE
        PUT ( "IMPOSSIBLE" ) ;
    END IF;
```

2.6 分 程 序

分程序由**BEGIN**……**END**所包含的程序段组成。它可以按照标识符的规则取一个名字。分程序名位于首部，后接一个冒号，而且在**END**之后必须重复这一名称。

分程序的特点在于它的开始部分 可以是由**DECLARE**引入的说明段。分程序中说明的变量都是局部变量。也就是说，如果一个标识符在分程序中被再次定义，那么该标识符将具有新的意义。原来在分程序外的意义同时被掩蔽。分程序内定义的变量，在该分程序外是不可见的。下面是两个对象交换的分程序：

```
SWAP:
DECLARE
TEMP：指定类型;
BEGIN
    TEMP : = A;
    A : = B;
    B : = TEMP;
END SWAP;
```

在这个例子中，TEMP是局部变量，而 A和B是全局变量。

2.7 循 环

所有的循环都是由键词**LOOP**引入的。在ADA语言中有几种不同形式的循环。

2.7.1 无条件循环

无条件循环的格式为：

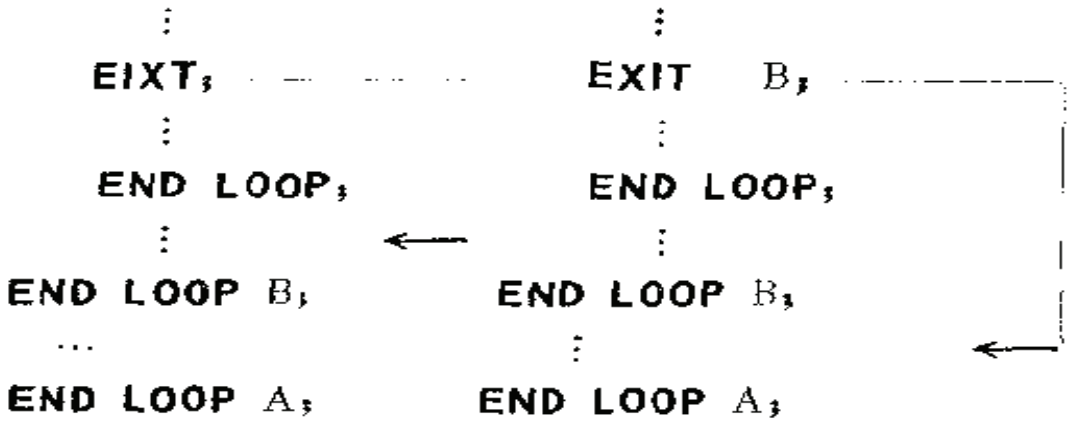
```
LOOP
    语句列;
END LOOP;
```

它将重复地执行语句列。原则上应该避免这种死循环，因此在语句列中应该有一个停止条件。例如循环内部用下述语句

```
IF 条件 GOTO……;
```

明确规定循环停止条件。但是采用这种方式与结构化程序的要求太不相称。退出循环最好的方式是使用**EXIT**语句，这个语句使程序转向执行紧跟在**END LOOP**之后的语句。对于处在嵌套循环最内层的**EXIT**，一般只表示跳出最内层循环。但可以给循环取一个名（如同分程序），在EXIT之后标明需要跳出的循环名(当然同时跳出这个循环所包含的诸循环)。例如：

```
A:                      A:
LOOP                    LOOP
 ⋮                       ⋮
B:                      B:
LOOP                    LOOP
  LOOP                    LOOP
```



我们已经看到，当一个循环有名时，在**END LOOP**之后必须重复这一名称。在这个例子中，是无条件退出循环。当然也可以使用下述形式实现有条件退出：

IF 条件 **THEN EXIT**；

这个语句还可以用下述形式表示：

EXIT WHEN 条件；

EXIT也可以用在下面将要讨论的条件循环中。

2.7.2 条件循环

当**EXIT WHEN**出现在无条件循环末尾时，例如：

```
A : LOOP
      ⋮
EXIT A WHEN 条件;
END LOOP A;
```

它起到结构程序中的REPEAT UNTIL功能。

象PASCAL语言一样，ADA也存在**WHILE**语句，其格式为：

WHILE 条件 **LOOP**

```
    ⋮
END LOOP;
```

例 2.7 已知序列 $\begin{cases} U_0 = 1 \\ U_n = \dfrac{1}{2}\left(U_{n-1} + \dfrac{a}{U_{n-1}}\right) \end{cases}$

收敛于$\sqrt{a}$，编写$\sqrt{a}$的程序段，要求相对精度为千分之一。

解

```
U : =1.0;
WHILE ABS (U*U-A) /A>0.002
    LOOP
        U : =0.5* (U+A/U) ;
END LOOP;
```

ABS (U*U - A) /A是U^2的相对精度，为U的相对精度的二倍，因在条件中取作0.002。

2.7.3 下标循环

上面提到的条件循环WHILE语句，可以非常方便地实现循环直到某一条件满足为止。但是，用户有时需要对一个元素的集合（例如一个数组的全部元素）进行某一运算，为此必须让下标在整个可能值的范围内变化。例如：求数组A诸元素的和可以写成：

```
A : ARRAY (1 . . N) OF FLOAT;
    ⋮
I : = 1;
S : = 0.0;
WHILE I<=N LOOP
S : =S+A (I) ;
```

```
    I: =I+1;
    END LOOP;
```

在这个程序段中，使用的是条件循环语句。但是通常避免下标变量I的显式管理，而把这个程序段改写成：

```
    S: =0.0;
    FOR I IN 1 . . N LOOP
    S: =S+A (I) ;
    END LOOP;
```

在这个程序段中的**FOR…IN…LOOP**语句，称为下标循环。在这种循环中甚至可以不用对变量I作出说明，而是在出现**FOR**时隐含说明它。它作为所在子程序的局部变量处理，这样做解决了其他语言中感到相当棘手的一个问题，这就是某些变量过于全局化以致在循环内部不允许修正这些变量。

上例中的下标循环还可以写成：

```
    FOR I IN A'RANGE LOOP……
```

这种格式更容易理解，它意味着当I的值在A的下标范围内就执行所规定的循环。这时A的下标可以是整型数（如由－5至＋5），也可以是任意的标量枚举型（不要忘记用户可以指明任意的离散子界）。例如：

```
    FOR I IN FRUIT LOOP……
```

或 **FOR I IN** APPLE..ORANGE **LOOP**……

其中FRUIT和APPLE..ORANGE都是用户定义的子界。在子界的范围内，执行的顺序是由低至高。下述语句执行顺序恰恰相反（相当于PASCAL中的DOWNTO语句）：

```
    FOR I IN REVERSE……LOOP……
```

比如：计算N的阶乘可以写成：

```
FOR I IN REVERSE 1..N LOOP
    F := F*I;
END LOOP;
```

例2.8 给定一个数组A（1..N），检查数值B是否在此数组中。若是指出在哪一列。若否，回答K = 0或K = N + 1。用不同的解法以阐明ADA语言提供的各种可能性。

解 ① 第一种可能性：

```
K := 0; --或者N + 1;
FOR I IN 1..N LOOP
IF A (I) = B THEN
    K := I;
    EXIT;
    END IF;
END LOOP;
```

如用PASCAL写这个程序段，则为：

```
K := 0;
FOR I := 1 TO N DO
    BEGIN
        IF A (I) = B THEN K := I
    END;
```

可见两者是非常相似的。但是在ADA程序段中，由于EXIT语句的存在，当找到B时即退出循环，而PASCAL程序段则必须执行N次。

② 第二种可能性：

在上述程序中必须管理两个变量I和K，每次循环必须进行两次检验。为了避免这一不便，我们可以定义：

```
AA : ARRAY (NATURAL RANGE < >)
        OF A元素的类型 : = A & B;
```

AA是不定维长数组，用匿名类型说明。AA的维长在A接续B初始化时加以确定。因此AA的维长为N + 1 。前N个为A的元素，最后一个元素为B。

```
FOR K IN AA' RANGE LOOP
    EXIT WHEN AA (K) = B;
END LOOP;
```

当然也可以用WHILE语句编写这个程序段。

例2.9 用欧几里德（EUCLID）算法计算两个整型数A与B（A>B）的最大公约数PGCD：从A减B直至新的A<B，再以B中减去新A直至新的B<新的A，如此反复，直到A = B即为最大公约数。其程序段为：

解：

```
LOOP
    WHILE A>B LOOP
        A : = A - B;
    END LOOP;
    WHILE A<B LOOP
        B : = B - A;
    END LOOP;
EXIT WHEN A = B;
END LOOP;
```

2.8 CASE 结 构

CASE结构允许在几个出口处设置检测条件。同时它还

包括一个出口用于各种假定条件都不成立时确定程序的去向，类似于PASCAL中的OTHERWISE。CASE结构的形式如下：

```
CASE 表达式 IS
    WHEN值1 | 值2 ⇒语句1；
    WHEN值3 ⇒语句2；
    WHEN OTHERS⇒语句3；
END CASE；
```

值1、值2和值3必须是与表达式类型相同的常数。表达式只能取其中的一个值。此时与该值相对应的语句或语句列被执行。然后跳到**END CASE**之后的语句。

在**CASE**结构中，允许几个值对应一个语句（或语句列），值与值之间加一个或符（|）；也允许某一区间的值（如V1．．V2）对应一个语句（或语句列）。当表达式的值与所假定的值都不相符时，则执行**OTHERS**所对应的语句（或语句列）。在**CASE**结构中也可以没有**WHEN OTHERS**。如果有，则意味着表达式取任意一值（类型必须相符）都有与其相对应的语句（或语句列）。下面的程序段是非法的：

```
TYPE COLOR IS (BLUE, WHITE, RED)；
     X, COLCOR;
     ⋮
CASE X IS
    WHEN BLUE⇒……;
    WHEN RED⇒……;
```

```
    END ⇒CASE;
```

其错误在于没有给出X取WHITE时的操作，在这个程序段中要么缺少**WHEN** WHITE⇒……；要么缺少**WHEN OTHERS**⇒……。如果对应于某一值不需要任何操作，则可放一个空语句（**NULL**）；例如：

```
    WHEN OTHERS⇒NULL;
```

例2.10 在顾客记录上读入一个字符（LC：CHARACTER），根据这个字符确定对比顾客的减价百分比。其规则如下：

'A' ：减价10%；
'B' ：减价5%；（当总额超过1000元时，否则没有）
'C' ：减价10%；
'D' ：减价5%（在任何情况下）
'E' ：不减价；

编写顾客付款程序段。

解：

```
TODAY: =AMOUNT;
    CASE LC IS
        WHEY'A' | 'C' ⇒TODAY: =0.9*AMOUNT;
        WHEY 'B' ⇒IF AMOUNT>1000.0
            THEN TODAY: =0.95*AMOUNT;
            END IF;
        WHEN 'D' ⇒TODAY: =0.95*AMOUNT;
        WHEN OTHERS⇒NULL;
    END CASE;
```

第三章 数 据 类 型

ADA是一种类型强化语言，每一种数据具有唯一的、确定的类型。本章在简述ADA数据类型之后，将详细研究ADA的标量类型、复合类型、存取型以及每种类型所具有的基本操作。

3.1 类 型 概 述

ADA中能够操作的“简单”对象是变量和常数。这里的“简单”二字系指除子程序、程序包、任务等之外的对象。一个对象的说明形式为：

对象名：〔CONSTANT〕类型指示〔：＝初始值〕；

对象名必须遵守2.2.1节的规则。方括号表示其中的CONSTANT和初始值可能有，也可能没有。有CONSTANT时，表示被说明的对象是一个常数。在大多数情况下，需要对常数规定一个初始值，并且在整个程序过程中该对象不能再赋予其他值，因为常数不能作为赋值对象。如果没有指明初始值，则意味着它是一个私有类型的待定常数。如果没有CONSTANT，表示被说明的对象是一个变量。第二个方括号内所指明的值是变量的初始值，它可以在程序的任何时刻由赋值语句加以改变。初始值可以是**字面值**，也可以用算术表达式来表示。这个算术表达式必须在对定义**加工**时完全确定。

类型指示可以是语言预定义的类型名，也可以是程序中先前定义的类型名。类型名的定义格式为：

TYPE 类型名 **IS**类型描述；

或者

SUBTYPE类型名**IS**类型描述；

例如：**TYPE** VECTOR **IS ARRAY** (1..N) **OF** FLOAT；

TYPE REAL **IS NEW** FLOAT；

V：VECTOR；

X：REAL；

PI：**CONSTANT** FLOAT：=3.14159；

这里，对象PI的类型指示为预定义的类型FLOAT。而对象X，V的类型指示REAL和VECTOR则是由先前的类型定义确定的。第二种方式，类型指示直接表示为类型定义，称为匿名定义。

例如：

V：**ARRAY** (1..N) **OF** FLOAT；

匿名定义的类型仅仅用在属于此类型的对象仅有一个或者该对象是局部使用的时候。这样做的好处在于书写简单，并可以省掉一个类型的标识符。但一般地说，使用匿名定义会带来许多不便，因此建议不要采用这种方式。例如：

A，B：类型定义；

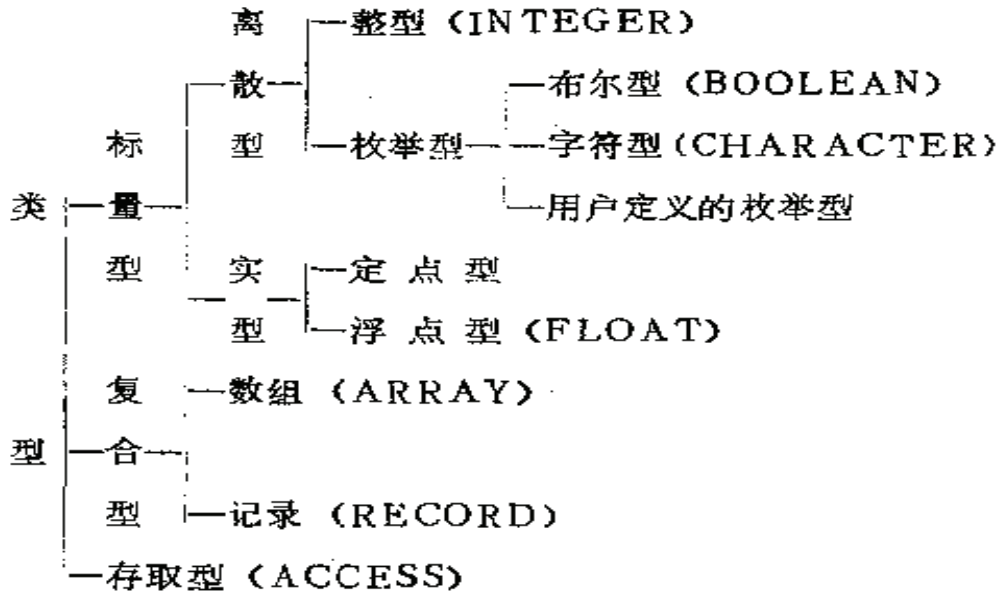
C：完全同样的定义；

按照语言的理解，A与B是相同类型的对象，而C是不同的类型，尽管它们有完全相同的性质。例如：A：=B；是合法的，而C：=B；却是非法的（不同类型不能直接赋

值1)。

另外由匿名定义的类型不能作为过程和函数的形式参数传递。显而易见，这种对象的用处是十分有限的。

在ADA中，数据分为标量型、复合型和存取型三大类（如下表所示）。



标量型的数据是单一的。也就是说，一个数据仅有一个元素。标量型又包括离散型和实型两种，离散型的变量可以作为数组的下标和循环变量使用。复合型的数据由多个元素组成。对于数组而言，这些元素属于同一类型。对于记录型，这些元素属于不同的类型。例如：

```
ARRAY (1..10)  OF INTEGER
```

这个数组由10个元素组成，这些元素都是整型。然而，

```
RECRD
    NAME : ARRAY(1..10) OF CHARACTER;
```

```
        REGISTRY : INTEGER;
        FASIT : (MRRIED,SINGLE, DIVORCED);
    END RECORD;
```

则是由一个10个字符的字符串、一个整型数和一个枚举型（三种情况选一）组成的。

存取型是用于存贮器动态管理的一些指针。

我们再着重说明标量型。在类型分类图中已经看到，标量型包括离散型和实型。离散型是可数数值的有序集合，而实型数的集合是不可数的（但是由于计算机的字长是有限的，只能给出实数的量化近似值。从集合理论的意义上讲，实型数也是可数数值的有序集合。且不谈这个问题，仍把实型数理想化地近似为一个不可数的数的集合）。

离散型是指整型和枚举型。枚举型是通过对有限个可能值的枚举来定义的。在ADA语言中，有两个预定义的枚举型。即：

```
    TYPE BOOLEAN IS (FALSE, TRUE);
    TYPE CHARACTER IS (NUL, SOH, … 'A' ,
                'B' , …) ;
```

除这两个预定义的枚举型外，还可以由用户定义一个标识符序列。每个标识符代表一个抽象常数，由这些常数构成一个枚举型。例如：

```
    TYPE DIRECTION IS (NORTH, SOUTH, EA-
            ST, WEST) ;
```

定义枚举型的同时，也确定了值的顺序。如在DIRECTION型中，NORTH<SOUTH<EAST<WEST。

在标量型中，只有整型和实型是数字型，这两种类型可

以相互转换。

对于ADA的所有类型而言，对任何两个同类型元素都可以实施两种操作：赋值与测等。

A：＝B；　（赋值）

A＝B 或 A/＝B （测等）

为了不使这两种操作用于私有类型的极端情况，需要加入特别的批准**LIMITED**。这种私有型称为受限私有型（参考第四章）。

复合型的赋值操作是元素之间一一对应的多重赋值。

其他操作对于每一种类型是否有意义取决于这种类型规定的数据的性质。例如：加法操作对于一个整型数区间总是有意义的，相反，枚举型颜色的乘法大概不会有什么意义。

对于某一类型定义的操作可以用于另一新类型，这种操作定义叫做**重载**。根据新类型定义的方式，重载可能是自动的（例如：对于派生型就是如此），也可能需要再作明确的定义。

3.2 派生型和子类型

在ADA语言中，有两种方法可以由一种类型（语言预定义的或者用户事先定义的）得到另一种类型。得到的类型是派生型或者是子类型（子界）。

我们可以通过以下方式来构成一个类型的派生型：

TYPE 名 **IS NEW** 父类型名；

派生型的特点是因袭其父类型的所有常数和所有运算，然而它却是与父类型不同的一个新的类型。派生型可以用来从逻

辑上区分不同的集合。例如：

```
TYPE COLOR IS (GREEN, YELLOW, RED);
TYPE LIGHT_OF_CROSS_ROAD IS NEW
        COLOR;
TEINTE : COLOR;
PASSAGE : LIGHT_OF_CROSS_ROAD :
                = GREEN;
```

注意对象TEINTE和PASSAGE分别属于父类型及它的派生型。尽管对象TEINTE也可以取值GREEN，但TEINTE：=PASSAGE却是非法的，这是因为两者是不同的类型。

父类型和派生型之间的转换很容易实现，只要在对象前面冠以要转换成的类型名即可。例如，上面的非法赋值语句可以改写成：

```
TEINTE : =COLOR (PASSAGE);
```

便成为合法的。

ADA语言中的派生型也可以做为父类型进一步构成新的派生型，如此可以形成一个派生链。例如：

```
TYPE A IS……;
TYPE B IS NEW A;
TYPE C IS NEW B;
TYPE D IS NEW C;
TYPE E IS NEW A;
TYPE F IS NEW E;
```

这个派生链相应于图3.1。

对于派生链这样的结构，类型转换只能在邻近类型之间进行。例如由A到B，由B到C…。

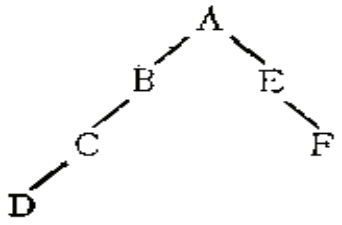


图 3.1 派生链

例3.1 对于上面的派生链，如果对象BB与FF的说明为：

BB : B;

FF : F;

怎样才能实现把FF的值赋于BB？

解： 对于这个派生链，如果写成BB : = B (FF)；将是错误的，只能按照派生链逐级转换。即：

BB : = B (A (E (FF))) ;

如上所述，派生型因袭其父类型的所有常数，因此有可能造成意义上的含混。当遇到这种情况时，采用指明类型的形式可以去掉这种含混性。例如：在本节开始举的COLOR的例子中，如果有下述语句将是含混不清的：

FOR I **IN** GREEN..RED

这是因为这些常数标识符对COLOR和LIGHT_OF_CROSS_ROAD两个类型是重载的。这时，可以采用指明类型的形式。即：

FOR I **IN** COLOR′ (GREEN)..COLOR′ (RED)

现在我们来说明子类型。子类型可以通过以下方式构成：

SUBTYPE 名 **IS** 父类型名〔约束〕；

子类型与派生型不同，它并不导出新类型，它只是在父类型上加一个约束，因而也称为子界。如果在定义子类型时未加

约束，那么这种子类型的作用仅仅在于将父类型更名。例如：

```
TYPE COLOR IS (RED, ORANGE, YELLOW,
      GREEN, BLUE, VIOLET, INDIG O);
SUBTYPE RAINBOW IS COLOR;
```

子类型仅仅是将COLOR更名。因此可以得出结论：只要不破坏约束条件，一个算术表达式中父类型及其子类型的元素是没有任何区别的。

3.3 约　　束

一个类型定义经常伴随一个约束，这个约束限制了被定义类型的取值范围。约束是相对于其父类型的取值范围而言的。这个父类型可以是语言预定义的一种类型，它也具有一个可以接受的取值范围。对于不同的类型，约束的种类也不同。

3.3.1 离散类型的约束

对于离散类型，约束表现为类型取值范围的限制。例如：

```
TYPE THOUS AND IS NEW INTEGER RANGE
      1..1000;
TYPE INDEX IS NEW INTEGER
       RANGE 1..N;
TYPE DAY IS (MONDAY, TUESDAY,
      WEDNESDAY, THURSDAY, FRIDAY,
```

SATURDAY, SUNDAY);

SUBTYPE WORK_DAY **IS** DAY **RANGE** MONDAY..FRIDAY;

在上例中，INDEX的上界只有在执行时才能确定。类型说明还可以采用父类型匿名的方式。例如：

TYPE THOUSAND **IS RANGE** 1..1000;

在这个类型说明中，父类型INTEGER不再出现。但是，对于子类型的说明不得采用这种方式。由于子类型并不导出新的类型，父类型名必须在说明中出现。例如：

SUBTYPE THOUSAND **IS RANGE** 1..1000; 将是非法的，其正确形式为：

SUBTYPE THOUSAND **IS** INTEGER **RANGE** 1..1000;

整数类型的父类型有三种类型。到预定义的整型（INTEGER）、长整型（LONG_INTEGER）和短整型（SHORT_INTEGER）。

3.3.2 实型的约束

对于实型有区间约束和精度约束。精度约束对定点数来说表示为绝对偏差的形式。对于浮点数来说则表示为有效数字的位数。例如：

TYPE REAL **IS NEW** FLOAT DIGITS 6 **RANGE** -1.0E+10..1.0E+10;

定义的浮点型至少有六位有效数字，数值范围为-10^{10}～$+10^{10}$。

TYPE FIX **IS NEW** FLOAT **DELTA** 0.001 **RAN-**

GE 0.0..5.0;

定义的定点型取值范围为0.0至5.0，步长为0.001。

在类型定义中也可以不出现区间约束，这时取值范围则为机器能够表示的最小数到最大数。

定义一个实数类型，实质上是根据给定的约束条件确定**标准数**的集合。标准数是以二进制表示的，它具有足够的位数以满足精度约束。

对于由**DIGITS** D 定义的浮点型，标准数的位数 B（整型数）应该是大于

Dlog10/log2 = D/0.30103

的最小的一个整型数。因此，标准数将由零和如下形式的数组成：

±尾数* 2 * *指数

其中：

——尾数的形式为0. × ×…× ×。也就是说，尾数可以由B位二进制的数精确表示，其值在0.5至 1 之间。

——指数是一个在 - 4 *B至 + 4 *B之间的整数。

例3.2 若定义

TYPE FF1 IS **NEW** FLOAT **DIGITS** 1；写出标准数。0.3的近似值为多少？若用DIGITS 3又有些什么变化？

解：对于DIGITS1，必须有$B \geqslant \frac{1}{0.3}$，因此 B = 4。 9个尾数为：0.1000 ($\frac{1}{2}$)，0.1001 ($\frac{9}{16}$)，0.1010 ($\frac{5}{8}$)，0.1011 ($\frac{11}{16}$)，0.1100 ($\frac{3}{4}$)，0.1101 ($\frac{13}{16}$)，0.1110($\frac{7}{8}$)，

0.1111 ($\frac{15}{16}$) 和0.0000 (0)。

指数在－16至＋16之间，因此值的范围在2^{-16} ($\approx 10^{-5}$) 至2^{16} ($\approx 10^{5}$)。

0.3等于$\frac{3}{10}$。最接近0.3的$\frac{\times}{16}$形式的数是$\frac{5}{16}$。因此0.3将由0.1010* 2^{-1} (=0.3125) 来表示。

对于DIGITS 3，B＝10位。将有$2^{9}+1=513$个尾数。指数在－40至＋40之间。因此数的范围为：

2^{-40} (10^{-13}) 至2^{40} (10^{13})

例3.3 指出下列定义有什么错误，

TYPE FF2 **IS NEW** FLOAT **DIGITS** 2 **RANGE**

0.0 . . 1.0E20;

解：对于DIGITS 2，B＝7位。因此最大指数为28。而$2^{28}\approx 10^{9}$，不可能达到$1.0*10^{20}$。

当引入一个浮点实型数的时候（例如：一个字面值），这个数将被转换成与它最接近的标准数。这个标准数对于欲表示的真实数的偏差决不会超出所要求的精度。

现在讨论定点实型数。对于一个由**DELTA** X定义的定点数，则要确定一个**二进制增量**或**实际增量**。这个增量通常是一个2的幂，其数值刚刚小于X。定点数的标准数形式为：

K* ＝进制增量

这里K为由$-2^{n}+1$至$2^{n}-1$所包括的整数集。n的选择原则是：使所要求的上下界在标准数所确定的范围内。

例3.4 写出下面定义的定点实型数的标准数：

TYPE FF1 **IS NEW** FLOAT **DELTA**
0.01 **RANGE** 0.0..10.0;

解：首先确定二进制增量，其值为：

$$2^{-7}=\frac{1}{128}\doteq 0.007<0.01$$

所需要的数和标准数表示如图3.2

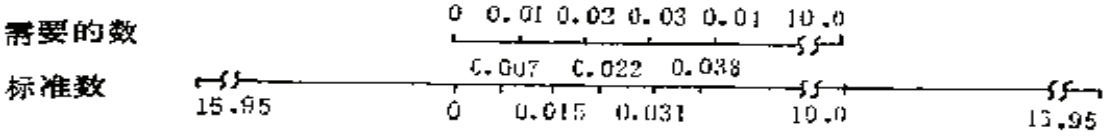


图3.2 需要的数和标准数

上限10恰恰是一个标准数，即：

$$10=1280*\text{二进制增量}=1280*\frac{1}{128}$$

由 $2^n-1=1280$可以求得$n\doteq 10.3$。但是标准数必须覆盖所规定的范围，因此取$n=11$。

这样，最后一个标准数为：

$$(2^{11}-1)*\frac{1}{128}=\frac{2047}{128}\doteq 15.99$$

这就是说，这些标准数可以由11位的二进制数表示（小数部分7位，整数部分4位），小数点加在第七位和第八位之间。即：

11	10	9	8	7	6	5	4	3	2	1

3.3.3 预定义实型的约束

在ADA语言中预定义的实型有FLOAT、LONG_FLOAT和SHORT_FLOAT。就精度而言，LONG_FLOAT高于FLOAT，而SHORT_FLOAT低于FLOAT。预定义实型FLOAT的特性总是与所使用的机器有关，但是ADA编译器允许用户根据需要确定精度。这是ADA程序便于移植的一个重要因素。例如：用户需要定义：

TYPE REAL **IS NEW** FLOAT **DIGITS** 10;

如果使用的机器FLOAT型不能保证给出10位数字的精度，只要改写成：

TYPE REAL **IS NEW** LONG_FLOAT **DIGITS** 10;

即可。而这是程序中唯一需要修改的语句！如果事先写成：**TYPE** REAL **IS DIGITS** 10；那么甚至连上面仅有的一条修改也不必要。在这种情况下，将由编辑器选择能够满足精度约束条件的适当的父类型。

需要指出的是：**DIGITS** 10应理解为“至少十位数字”。因此系统要保证给出的精度高于、至少等于所要求的精度。显然，只要用户所要求的精度能保障用户算法的正确执行，那么可以肯定在所有机器上都可以得到足够准确的结果。

由此可以看出，对于数值分析问题，ADA具有较强的移植能力。当然上述讨论也完全适用于 **DELTA** 约束，在此不赘述。

思考题： 对于本节中关于REAL的定义，如果使用子类

型，定义的方式是否相同？

3.4 属　　性

属性是与类型或者与给定类型中的对象有关的函数。这些函数提供了与系统有关的信息，或者提供了与涉及的对象在系统中的表示方法有关的信息。几乎没有其他的高级语言能够提供这种信息。例如：

X'ADRESS

表示对象X的第一个字所占据的存贮器的地址。属性可用于任何对象，包括变量和子程序。属性的表示形式如下：

对象名'属性名

3.4.1 通用属性

对于每一个类型（或子类型）T定义的属性列于表3.1

表 3.1　　通 用 属 性 表

属　　性	意　　义
T'BASE	基类型（对于子类型系指其父型）
T'SIZE	T类型元素所占据的最大存贮空间（位）

3.4.2 标量型的属性

对于标量型（或其子类型）定义的属性如表3.2所示。例如：

TYPE COLOR **IS** (BLUE, WHITE, RED)；

表 3.2　　标 量 型 的 属 性

属　　性	意　　义
T′FIRST	T类型的第一个元素
T′LAST	T类型的最末一个元素
T′IMAGE(X)	T类型对象X的字符串表示
T′VALUE(X)	IMAGE的逆，由字符串表示的对象X在类型T中的值

```
X : COLOR;
    ⋮
X : = BLUE;
```

则　COLOR′FIRST为BLUE
COLOR′LAST为RED
COLOR′IMAGE (X) 为BLUE
COLOR′VALUE (X) 为 0

3.4.3 离散型的属性

对于离散型（或其子类型）T定义的属性如表3.3所示。

表 3.3　　离 散 型 的 属 性

属　　性	意　　义
T′POS (X)	在序列T′FIRST..T′LAST中的T型对象X的位置
T′VAL (P)	位置为P的T型对象的值
T′PRED (X)	T类型中X的前导数（当X = T′FIRST时无意义）
T′SUCC (X)	T类型中X的后续数（X不得等于T′LAST）

在整型中，T′POS (T′FIRST) = T′FIRST，在枚举型中，T′POS (T′FIRST) = 0

例3.5 T′POS〔T′SUCC (X)〕与T′POS (X) 之间有什么关系？

解 T′POS〔T′SUCC (X)〕= T′POS (X) + 1

3.4.4 实数型的属性

对于定点实型和浮点实型定义的主要属性如表3.4和3.5所示。

表 3.4　　定 点 实 型 的 属 性

属　　性	意　　义
T′DELTA	规定的步长
T′ACTUAL—DELTA	实际增量（或二进制增量）
T′BITS	T的标准数所需的位数
T′LARGE	T型最大标准数

表 3.5　　浮 点 型 的 属 性

属　　性	意　　义
T′DIGITS	类型定义中指明的数字位数
T′MANTISSA	标准数中尾数的位数
T′EMAX	标准数的最大指数
T′EMIN	标准数的最小指数
T′SMALL	最小正标准数(它为最小不为零的数，与2 * *（－T′EMAX）具有相同的数量级)

续 表

属　　性	意　　义
T′LARGE	最大标准数
T′EPSILON	大于1.0的最小标准数与1.0之差（注意1.0总是标准数）
T′MACHINE—MANTISSA	机器表示的尾数位数（≥T′MANTISSA）

显而易见，对于实数类型，总有：

T′LAST≤T′LARGE

3.4.5 数组型的属性

对于数组型T的定义的属性如表3.6所示。

表 3.6　　数 组 型 的 属 性

属　　性	意　　义
T′FIRST (I)	数组型T第I维下标的下限
T′LAST (I)	数组型T第I维下标的上限
T′LENGTH (I)	数组型T第I维下标的数目（0为空维）
T′RANGE (I)	由T′FIRST (I) ‥T′LAST (I) 所定义的子类型

如I = 1（或者是只有一维的数组），那么（I）可以省略。即 T′FIRST（1）记作 T′FIRST。注意：T′FIRST 是指第一个元素的下标，而不是第一个元素的本身。因

此：

T′FIRST＝T′RANGE′FIRST

例3.6 若数组定义为：

```
TYPE MAT_RECT IS ARRAY (1..10, 1..20)
     OF FLOAT;
M : MAT_RECT;
```

写出M的属性。

解

M′FIRST = 1	M′LAST = 10
M′FIRST (2) = 1	M′LAST (2) = 20
M′LENGTH = 10	M′LENGTH (2) = 20
M′RANGE = 1..10	M′RANGE (2) = 1..20

关于其他类型所定义的属性，将在以后有关章节中说明。

3.5 标量型的运算

如前所述，赋值和测等对于所有的类型都是允许的。除此而外，各类型还有一些特定的运算。

3.5.1 枚举型的运算

枚举型允许将两个元素进行比较，结果为布尔型。比较运算与元素的次序有关，例如：

```
TYPE COLOR IS (BLUE, WHITE, RED);
```

那么，BLUE<RED为真（TRUE）。

需要注意的是，如果枚举元素符号已经重载，那么进行比较运算时，任何模糊不清的概念都是不允许的。例

如：

```
TYPE COLOR IS (BLUE, WHITE, RED);
TYPE TEINTE IS (BLUE, GREEN, YELLOW,
     ORANGE, RED) ;
```

那么，

BLUE＜WHITE指的是COLOR中的BLUE

BLUE＜GREEN指的是TEINTE中的BLUE

然而，

BLUE＜RED是含混不清的，也是不允许的。为避免这种含混性，应该写作：

COLOR′ (BLUE) ＜COLOR′ (RED)

或者TEINE′ (BLUE) ＜TEINTE′ (RED)

3.5.2 布尔型的运算

布尔型是预定义的枚举型。除比较运算外，还具有逻辑运算即：**NOT**，**AND**，**OR**，**XOR**等。

3.5.3 整型的运算

整型允许习惯的算术运算，其中除法指的是整除。**MOD**表示求模数，**REM**表示求余数。

当A与B同为正或同为负时，A **REM** B以及A **MOD** B符号与A（或B）一致。否则**REM** 与 A 同符号，**MOD**与B同符号。

例3.7 根据A与B的值，确定A/B，A **REM** B及A **MOD** B的值与符号。

解：

A	B	A/B	A REM B	A MOD B
10	3	3	1	1
9	3	3	0	0
-10	3	-3	-1	2
-9	3	-3	0	0
10	-3	-3	1	-2
9	-3	-3	0	0
-10	-3	3	-1	-1
-9	-3	3	0	0

整数幂运算A**B只限A为整型数，B为正整数或零。

3.5.4 实型数的运算

实型数的除法用实数的算术运算来实现。对于定点型与整型的混合运算，其定义如表3.7所示。

表 3.7

A	B	A*B	A/B
定点	整型	A型	A型
整型	定点	B型	不存在
定点	定点	先得最大精度的定点数，再转换为适当的定点数	最大精度的定点数

幂运算只允许：浮点型**整型。对于A**B，如果B

是负的整型数，那么其值等于1.0/A**ABS (B)。如果B= 0，那么A** 0 =1.0。

3.6 数组型及其运算

在数组类型中，要定义下标和元素的类型。数组本身可以带约束（有确定的维长）、也可以不带约束。例如：

```
TYPE VECTOR IS ARRAY (INTEGER RANGE
        <>) OF FLOAT;
TYPE VECT IS ARRAY (1..10) OF FLOAT;
```

VECTOR是不带约束的，而VECT是带约束的。也可以定义VECTOR的子类型：

```
SUBTYPE V IS VECTOR (1..10);
```

它们都是一维数组。当然还可以有多维数组。任何离散型都可以做为数组的下标。例如：

```
TYPE COLOR IS (BLUE, YELLOW, RED);
TYPE MARIAGE IS ARRAY
        (COLOR, COLOR) OF BOOLEAN;
V : MARIAGE;
```

数组MARIAGE的下标是枚举型，如果有：

```
V (BLUE, YELLOW) := TRUE;
```

表示蓝和黄可以调色。

数组的运算包括测等、赋值和连接。

数组的测等和赋值是一个元素一个元素进行的。如果两个数组的所有对应元素一一相等，则两个数组相等。前面提到的结构常值（参考§2.3）可以用于数组的赋值。对于一

个无约束的数组，赋值的同时也就确定了数组的维长。

例3.8 将零矢量赋予矢量V。

解：如果V的维长是确定的，那么可以采取下述方法赋值：

```
V := (V'RANGE => 0.0);
```

否则，如果写：`V := (1..10 => 0.0);`

那么，在赋值的同时，将V的维长确定为10。

对于一维数组而言，连续符&操作是使两个操作并值。例如：

```
U, V, W : ARRAY (INTEGER<>) OF
                FLOAT;
W := U + V;
```

那么，W的属性有：

```
W'LENGTH = U'LENGTH + V'LENGTH
W (1..U'LEGTH) = U
W ((U'LENGTH + 1)..W'LENGTH) = V
```

我们可以把上面这种方法加以推广：如果A和B属于区间V'RANGE，那V (A..B)则是由V的一系列元素V (A)、V (A + 1)、…V (B) 组成的矢量，我们把V (A..B)称为数组片。

数组片之间可以赋值。例如：

```
V (1..5) := W (I..J);
```

只要J = I + 4就是合法的。但是对于取自同一数组的不同数组片，它们不得相互重叠。

通过对数组的运算，可以解决查表问题。下面我们用一个例子来说明：设有一维数组TAB，常数K与该数组元素的类型相同。如TAB中有数值为K的元素，则打印出第一个等

于K的元素的列，否则给出没有K的信息。这是一个查表问题。我们可以引入布尔量FOUND。如果有等于K的元素，FOUND为TRUE。对应的程序如下：

```
FOUND : =FALSE;
FOR I IN TAB'RANGE LOOP
    IF TAB (I) = K THEN
        PUT (I) ;   --印出所在行
        FOUND : =TRUE;
        EXIT;
        END IF;
END LOOP;
IF NOT FOUND THEN PUT ("NO FOUND");
END IF;
```

若数组TAB的元素是递增排列的，可以用两分法查找。即将查找区逐次平分为二，每次选择区间的中点为I。如TAB (I) =K，即找到。如TAB (I) >K则在下半部分查找，否则在上半部查找。

由此我们可以看到，ADA对数组的处理与经典语言是相同的。但是增加属性以后，带来了很大的方便。

例3.9 用两分法查TAB中是否有等于99的元素（设TAB的元素是1000之内的整型数）。

解：

```
TYPE T IS NEW INTEGER RANGE 1..1000;
TYPE TABLE IS ARRAY (1..N) OF T;
TAB : TABLE;
FOUND : BOOLEAN;
K : CONSTANT  T : =99;
```

```
INF, SUP, I : T;
BEGIN
    INF : = 1;
    SUP : = N;
    FOUND : = FALSE;
    LOOP
        I : (INF + SUP) DIV 2;
        IF  TAB (I) = K  THEN
            PUT (I) ;
            FOUND : = TRUE;
            ELSIF  TAB (I) <K  THEN
                INF : = I + 1;
            ELSE
                SUP : = I - 1;
        END IF;
        EXIT WHEN FOUND OR INF>SUP;
    END LOOP;
    IF NOT FOUND THEN
        PUT ( "NO FOUND" );
    END IF;
END
```

我们建议读者自行编写标量积、矩阵乘法等程序。

3.7 字符串及其运算

一维数组的特殊情况是预定义类型字符串STRING。这

个类型定义为：

TYPE STRING **IS ARRAY** (NATURAL **RANGE**
〈 〉) **OF** CHARACTER;

其中CHARACTER是预定义类型，系由128个ASCII码构成的枚举型（参考第五章）。NATURAL 是预定义的子类型，其类型定义为：

SUBTYPE NATURAL **IS** INTEGER
RANGE 1..INTEGER'LAST;

这就是说，以字符串形式出现的一维数组，其下标总是由1开始的。而字符串的最大长度可以在初始化时确定。例如：

CHAINE：STRING = "HALLO"；

那么，CHAINE的最大长度为5。也可以在对象说明时以参数指明的方式确定一个最大长度约束。如：

CHAINE：STRING（1..80）；

ADA中的这一参数化约束是需要精心确定的。

字符串常数有两种形式：其一为结构常值。例如：

（'H'，'A'，'L'，'L'，'O'）

或者 （1 | 2 ⇒X，OTHRS⇒''）

其二为字面值。即夹在双引号中的字符串。例如：

"HALOO"

或者 "XX"

对于字符串的经典处理方法，在ADA中大都存在。它们是：赋值、连接、截取、删除和比较。赋值与连接操作与一维数组相同。截取和删除可以通过数组片给出。例如：

X（5..7）

表示由X的5、6、7三个元素组成的子字串（在BASIC写作

MID$（X$，5，3））。对于比较操作，除测等之外，所有的比较算符（如：＜，＜=，＞=，/=）都有定义，这些比较操作是以词典中字母顺序为依据的。

BASIC中的LEN，VAL……在ADA中可以借助于LENGTH，IMAGE，……来表达。例如：ASC（A$）在ADA中可以写作CHARACTER'POS（A）。

ADA中没有PACKED语句来规定在存贮器中以紧缩的方式存放数组，而是通过参注(PRAGMA)来实现。其形式为：

PRAGMA PACK（数组或记录型名）；

例3.10 把一个字符数未知的字符串X转换成长度为K的字符串Y。若K＜X'LENGTH，则取X的最右边的K个字符（在BACIC中写作RIGHT$）。若K＞X'LENGTH，则在Y的左边加上适当的空格以使得Y由K个字符组成。

```
解：Y：STRING(1..K)：=(1..K=> '' );
    BEGIN
        IF K>X'LENGTH THEN
            Y：=Y(1..K-X'LENGTH) &X;
            ELSIF K<X'LENGTH THEN
                Y：=X(1+X'LENGTH
                    -K..X'LENGTH);
        ELSE Y：=X;
        END IF;
    END;
```

例3.11 检验字符串X是否在字符串Y中存在。如果存在，指出第一个重叠字符在Y中的位置。例如："MIS"在"HALLO MISTER"中的重叠位置为7。

解：这个例子与例3.9非常相似。

```
X : STRING;
Y : STRING;
L : NATURAL;
BEGIN
    FOUND : = FALSE;
    L : = X'LENGTH - 1;
    FOR I IN 1..Y'LENGTH-L LOOP
        IF Y (I..I+L) = X THEN
            PUT ( "FOUND" );
            PUT (I);
            FOUND : = TRUE;
            EXIT;
        END IF;
    END LOOP;
    IF NOT FOUND THEN
            PUT ( "NO FOUND" );
    END IF;
END;
```

3.8 记 录 型

记录型允许将一些性质可能完全不同的成分罗列在一起产生一个集合对象。它与Pascal中的RECORD非常相似。从功能上说与PL/1或COBOL中的“结构”相同。

这种性质互不相同的数据在文件中是司空见惯的。例

如：在顾客登记表中，对于每一顾客有：编号、名称、地址、邮政信箱、营业额等等。我们可以把这样的文件定义为一个记录型：

```
TYPE CLIENT IS RECORD
    NUMBER : INTEGER;
    NAME : STRING (1..10) ;
    ADDRESS : STRING (1..30) ;
    PBOX : INTEGER;
    VOLUME : FLOAT;
END RECORD
CLI : CLIENT;
```

在这里CLI是记录型CLIENT的变量。

3.8.1 记录的存取

对记录型变量的成分可以用限制名赋值或访问。例如：

```
CLI.NAME : = "DUPONT";
CODE : = CIL.NUMBER;
```

对完整的记录型变量可以用结构常值赋值。例如：

```
CLI : = (10,"DUPONT", "2 ,PAUL__FEUGA,
        TUULOUSE" , 2729, 10000.0);
```

或者

```
CLI : = (ADDRESS=> "2 , PAUL__FEUGA,
        TOULOUSE" , NAME=> "DUPONT" ,
        PBOX=>2729, VOLUME=>10000.0,
        NUMBER=>10);
```

如果在类型说明中对某些元素赋予初始值，那么这一初始值即为缺省值。也就是说，这一类型的对象在没有赋值时

都取这一值。例如：

```
TYPE COMPLEX IS RECORD
        RE : FLOAT;
        IM : FLOAT := 0.0;
    END RECORD
```

规定所有的复数的初始值均为实数（虚部为零）。

3.8.2 变体记录

ADA允许同一系列的不同记录具有不尽相同的结构（包括记录的成分数目不同或类型不同）。例如：一个企业的职工记录中，包括名称、出生日期、家庭状况。家庭状况是因人而异的。以 'S' 表示单身、'M' 表示已婚、'D' 表示离婚、'W' 表示鳏寡。如果是单身不再有别的内容。如果是已婚，则还有配偶姓名与结婚日期。如果是离婚或鳏寡还有离婚或丧偶日期。可见，记录的结构与描述家庭状况这一元素有关。我们把它称为判别式（**discriminant**）。在ADA中，判别式作为记录的一个参数。例如：

```
TYPE NNN IS STRING (1..10) ;
TYPE FAM IS ( 'S' , 'M' , 'D' , 'W' );
TYPE DATE IS RECORD
        DAY : INTEGER RANGE 1..31;
        MONTH : INTEGER RANGE 1..12;
        YEAR : INTEGER RANGE 0..99;
END RECORD;
TYPE EMPLOYE (FAM—SIT : FAM) IS RECO-
        RD
```

```
        NAME : NNN;
        BIRTH—DATE : DATE;
        CASE FAM-SIT IS WHEN 'S' =>NULL;
            WHEN 'M' =>SPOUSE—NAME : NNN;
                       MAR-DATE : DATE;
            WHEN 'D' |'W' =>DW—DATE:DATE;
        END CASE;
    END RECORD;
```

对于每个职工可以声明为：

```
    EMP : EMPLOYE ( 'S' );
```

对于EMP可以用结构常值赋值。例如：

```
    EMP : = ( 'S' , "SEVELY", (15, 8 ,35));
```

或者使用有名结构常值：

```
    EMP: = (FAM—SIT⇒'S' , NAME⇒"SEVELY",
                BIRT—DATE⇒ (15, 8 , 35));
```

变体记录也可以对单个元素分别赋值。如：

```
    EMP.NAME : = "SEVELY" ;
```

但是，我们不能对EMP.FAM－SIT赋值，因为判别式的赋值只能在对整个记录赋值时进行。

记录型也可以使某些元素接受一个初始值，这个初始值起缺省值的作用。判别式也可以采用这种方法。比如大部分职工是已婚的，则可以写成如下形式：

```
    TYPE EMPLOYE (FAM—SIT : FAM : = 'M' )
        IS……
```

这时，EMP : EMPLOYE; 是允许的。EMP自然地被认为是已婚的。

3.8.3 录中的约束

能够用于RECORD类型的约束是对判别式取值的约束。例如，可以定义：

SUBTYPE EMP-SINGLE **IS** ENPLOYE ('S');

或者采用匿名类型表示为：

EMP : EMPLOYE (FAM-SIT⇒ 'S') ;

另一类判别式引入长度可变的数组作为记录的成分。这个数组的长度构成判别式。例如：

```
TYPE CHAINE (LONG : INTEGER RANGE
        0··MAX) IS
RECORD
    INRUT : INTEGER RANGE 0..MAX : = 0;
    OUTPUT:INTEGER RANGE 0..MAX: = 0;
    CONTENT : ARRAY (1..LONG) OF类型;
END RECORD;
```

由这一类型可以定义下面的对象：

```
BUF 1 : CHAIN (100) ;
BUF 2 : CHAIN;
```

记录型对象接受了判别式的约束后，其布尔型属性X'CONSTRAINED的值为真。

因此，BUF1'CONSTRAINED为真，而

BUF2'CONSTRAINED为伪。

3.8.4 变体成分的存取

在一项变体记录中，能够存取的当然是那些根据判别式

的值在记录中存在的那些成分。ADA 包含有一切希望的检测，而且在需要的时候可以引发异常 CONSTRAINT_ERROR。

当然最好由用户自己选择所需要的检测。对于3.8.2中的例子，如果希望写出已婚职工的结婚日期，那么可以写成：

```
IF EMP.FAM_SIT = 'M' THEN
    PUT (EMP.MAP_DATE);
END IF
```

在ADA语言中，对于变体记录，经常使用与记录中CASE相一致的CASE结构实现检测。

记录型的属性如表3.8所示。

表 3.8　　记 录 的 属 性

属　　性	意　　义
T'CONSTRAINED	记录型T的判别式是否已接受一个约束（布尔型）
X'POSITION	记录的成分X距该记录起点的距离（以字为单位）
X'FIRST—BIT	成分X的首位
X'LAST—BIT	成分X的末位

3.9 布尔数组

ADA没有Pascal中的SET类型，因此在某种程度上来说，使用受到了一些限制。ADA 使用布尔型数组来处理集合型数据，集合运算可以由逻辑运算来模拟：

OR — 并集；

AND — 交集；

⇐ — 包含于；

IN — 检测元素是否属于由某区间所代表的集合。

例3.12 利用 Eratosthene 筛法求 2 至N的素数。这个方法的步骤是：

1．建立由 2 至N的数组；

2．取留在数组里最小的数（第一次应为 2），该数为素数；

3．从数组里删除它和它的倍数。这样可以构成一个下标在 2 ..N范围内的布尔数组。布尔量 CRIB（K）= TRUE表示仍留在数组内，CRIB（K）= FALSE表示已被删除；

4．重复以上过程，直到数组中的元素全部被删除。

解：
```
PROCEDURE ERA IS
    TYPE NB IS NEW INTEGER RANGE 1..10000
        N : NB;
PROCEDURE CRIBLE (N : IN NB) ;
    TYRE CR IS ARRAY (2..N) OF BOOLEAN
            : = (2..N⇒TRUE) ;
    CRIB : CR;
    REST : INTEGER : = N - 1 ;
    K : 2..N;
    L; INTEGER;
    BEGIN
        WHILE REST /= 0  LOOP
        K : = 2
```

```
            WHILE NOT CRIB(K) LOOP--寻找最小数
                K:=K+1;
            END LOOP;
            PUT (K); --K为素数
            L:=K; --删除K及K的倍数
            WHILE L<=N LOOP
                IF CRIB (L) THEN
                    CRIB (L) :=FALSE;
                    REST:=REST-1;
                END IF;
                L:=L+K;
            END LOOP;
        END LOOP;
    END CRIBLE;
    BEGIN--主程序
        GET (N);
        CRIBLE (N);
    END ERA;
```

在这个例子中，过程的输入参数N是作为CRIBLE的可变维长给出的。N的范围为1至10000，可由用户确定。

还要说明的是，如果在ERA的首部加上一个**USE**语句，并在此语句中说明包含输入/输出过程GET与PUT的程序包名，那么它即成为一个完整的程序。

〔思考题〕如果把上例的说明做下面的改动，将有何区别？

```
TYPE CR IS ARRAY (2..N) OF BOOLEAN;
```

CRIB : CR : = (2..N⇒TRUE) ;

3.10 存 取 型

ACCESS类型允许通过指针动态管理存贮器，其方式与Pascal非常相似。**ACCESS**类型的说明形式如下：

TYPE TA **IS ACCESS** 类型指明；

它说明TA类型的对象是一些指针，这些指针能够访问指明类型的对象。例如：

```
TYPE TA IS ACCESS ARRAY (1..3) OF
        FLOAT;
P, Q : TA;
```

在此，P和Q是指针，可以指向由三个实数分量组成的一维数组。然而此时并没有产生任何对象，只有在进行了地址分配以后（也就是说给出了必要的存贮器的位置）才产生对象。例如，语句

```
P : =NEW TA ((0.5, 2.0, 8.24)) ;
```

保留了三个实数组成的矢量所需要的存贮器位置。这个矢量在存贮器内的地址放在变量P当中。把矢量记作P.**ALL**（代替Pascal中的标记P↑）。因此，对于上面这个例子，P.**ALL**的数值为（0.5，2.0，8.24）。矢量的一个成分写作：

P.**ALL**（1）或P（1），其值为0.5，

P.**ALL**（2）或P（2），其值为2.0，

P.**ALL**（3）或P（3），其值为8.24。

请注意，不要将下面两个赋值语句混淆：

```
P.ALL : = (0.0, 1.0, 2.0) ;
```

及 P：=Q；

前者是给对象赋值，后者是给指针赋值。例如：

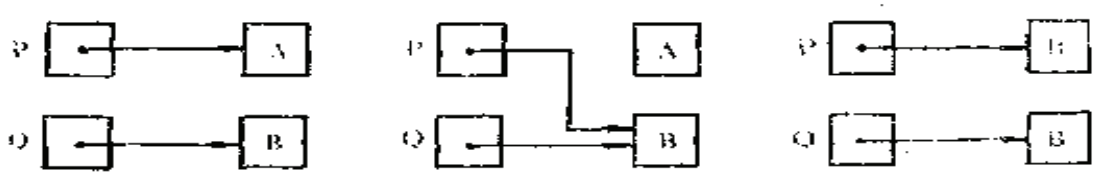
TYPE PTR **IS ACCESS** OBJECT；

P，Q：PTR；

P：=**NEW** OBJECT（A）；

Q：=**NEW** OBJECT（B）；

两种赋值的意义如图3.3所示：



P=Q；以后

P.ALL：=Q.ALL；以后

图3.3 两种赋值的意义

指针也可以取值**NULL**。这意味着不指向任何部分。

关于符号**ALL**的意义可做如下解释：指针通常是与记录型对象联在一起使用的，以3.8.2的例子为例，可以定义：

TYPE PT **IS ACCESS** EMPLOYE；

P：PT；

P：=**NEW** EMPLOYE('S'，'SEVELY'，（5，8，35））；此时P.NAME为职工的名字，P.FAM_SIT为其家庭状况，而P.ALL则表示职工的全部记录。

NEW用来动态地建立指针所指的对象。

对于存取型所定义的属性如表3.9所示

表 3.9　　存 取 型 的 属 性

属　　性	意　　义
PT'STORAGE_SIZZ	所有PT型对象保留的存贮空间的大小（以字为单位）

3.10.1 递归定义

当我们处理链表或树形结构的时候，经常会遇到这种情况：对象是 **RECORD** 型，而其中的某些元素可能是指向这一对象的指针。对于链表而言，它的元素系由从严格意义上讲的信息和一个指向它的接续元素的指针构成；对于树形结构而言，它的元素系由信息和指向相邻元素的一些指针构成。链表中最后一个元素的指针显然应取值 **NULL**，而将有一个外部的附加指针指向链表的第一个元素。

上面的结构可以采用递归定义。也就是说，定义一个元素类型需要用到指针的类型，而指针的类型又是根据元素的类型来定义的。因而出现了一个先定义元素的类型还是先定义指针的类型的问题。ADA 解决这一问题的方法是允许存在一个暂时的不完整的定义。例如：

```
TYP ELEMENT; --暂时的不完整定义
TYPE PTR IS ACCESS ELEMENT;
TYPE ELEMENT IS --现在给出完整定义
    RECORD
        DATA: DATATYPE;
        NEXT: PTR;
    END RECORD;
```

```
    DEPART : PTR;
```

链表中第一个元素的产生方式是：

```
DEPART : = NEW ELEMENT (DATAVALUE,
                NULL) ;
```

例3.13 设已经建立了如上的一个链表。**按链表顺序打印出它的元素所包含的信息。**

解：

```
P: PTR;
    P : =DEPART;
    WHILE P/=NULL LOOP
        PUT (P.DATA) ;
        P : =P.NEXT;
    END LOOP;
```

例3.14 设链表上的信息为STRING型，链表是按词典的字母顺序排列的，也就是：P.DATA<P.NEXT.DATA。**在正确的位置**上插入一个字符串K。

解：

```
K : STRING;
    P.Q, R : PTR;
    BEGIN;
        P : =DEPART;
        WHILE P/=NULL AND THEN P.DATA<K
            LOOP
            R : =P;
            P : =P.NEXT;
        END LOOP;
            Q : =NEW ELEMENT ( (DATA⇒K,
```

```
                NEXT⇒P) ) ;
            R.NEXT := Q;
    END INTERSECTION;
```

注意在判定中**AND THEN**的使用。当P = **NULL**时，没有必要去检测P.DATA。

我们还可以建立指针数组。例如：

```
TYPE PTR IS ACCESS对象类型;
TYPE TABP IS ARRAY (1..N) OF PTR;
P : TABP;
```

那么，P (I) .**ALL**表示第I个指针所指的元素。这种结构对于分类是非常有用的。分类中用到交换信息时，可以只交换指针。当被指对象的信息量很大时，这样做就大大减少了信息的流动。

3.10.2 存贮单元的释放

在PASCAL中，被指对象所占据的存贮单元是通过调用标准过程DISPOSE实现释放的。

在ADA则采用自动释放。自动释放发生在对象变为不可存取之际。例如：发生在分程序的出口或定义该对象的过程的出口。这种释放是否真正发生取决于编译器以及这个对象使用存贮单元的多少。

此外，还可以使用参注语句要求在脱离该存取类型定义域的时候释放存贮位置。其格式如下：

```
PRAGMA CONTROLLED (存取类型名);
```

还可以通过类属过程 UNCHEKED_DEALLOCATION的衍生进行显式释放：

```
PROCEDURE FREE IS NEW
    UNCHEKED_DEALLOCATION (ELEMENT,
                           PTR);
```

其中ELEMENT表示对象类型，PTR表示存取型，这样，FREE (P) 即可导致：

```
P := NULL;
```

于是P.ALL所占据的位置被释放。

至此，我们结束了有关数据类型的介绍。可以看到，数据类型是ADA强有力的工具之一。语言的设计者致力于最大限度地发挥数据类型的潜力，使它具有比Pascal更高的严谨性和更好的可移植性。另一方面它克服了Pascal中的一些局限性，如对实数和字符串的处理。从而使得ADA成为在科学计算方面取代FORTRAN的一个应选对象。

ADA数据定义的基本概念之一是参数化。我们可以先定义一些无约束的类型，然后在需要的时候对它们加上必要的约束。作为变体记录的参数的判别式，是类型定义参数化的一个具体例子。由3.8中的例子可以看到，参数化是非常有效的工具。

在第四章中，我们将介绍ADA的另外一些特点：一是出现在过程中的实际参数能够约束这些参数的类型（这正是FORTRAN子程序可变维长的推广）；二是类型可以成为类属对象，特别是成为程序包的参数。

第四章　子程序、程序包、分离编译及类属

本章将介绍ADA语言的一些重要概念。除子程序外，程序包、类属以及第六章的任务都是ADA语言所特有的模块化工具。另外，前面三章所涉及的例子大都是一些程序段，从本章开始，可以看到完整的ADA程序实例。

4.1　子　程　序

与其它语言相同，ADA有两类子程序：过程与函数。它们同样是ADA语言基本的模块化工具。

过程是按照如下格式来描述的：

PROCEDURE 名（参数表）**IS**
　　局部说明
　　BEGIN
　　语句列
　　END　名；

函数的描述格式为：

FUNCTION　名（参数表）**RETURN** 子类型 **IS**
　　局部说明
　　BEGIN
　　语句列

END　名;

过程的调用是通过调用语句实现的。在ADA中调用语句十分简单，只要写出过程名与参数就行了。函数则在算术表达式的内部调用，调用后回授一个指定类型的结果。

函数语句列中按执行顺序的最后一个语句必须是：

RETURN　表达式;

这个表达式的类型为函数说明首部中指定的子类型。而表达式的值用来替代调用算术表达式中的函数名。

END之后可以重复子程序名，也可以不重复，但是重复子程序名可以增加程序的易读性。

4.1.1　子程序中的参数表

子程序中的参数表形式为：

（名表：结合形式　类型；名表：结合形式　类型；……）

其中名表由结合形式相同、类型也相同的参数名结合在一起构成。结合形式有以下三种：

IN（或省略）：传送给子程序的参数，在子程序中视为不能修正的常数；

OUT：子程序作为结果而回授的参数；

IN OUT：混合参数。先传送给子程序，修正以后回授。

例如，可以把求一个实数平方的运算写成一个函数：

```
FUNCTION SQR (X : IN FLOAT)
                RETURN FLOAT IS
    BEGIN
```

```
        RETURN X**2;
    END SQR;
```

调用SQR时则采用如下形式：

```
    Y, Z : FLOAT;
    A := SQR (Z) + SQR (5.0+4.0*Y) ;
```

也可以把对一个整数加倍的运算写成一个过程：

```
PROCEDURE DOUBLE (K:IN OUT INTEGER) IS
    BEGIN
        K := 2 *K;
    END DOUBLE;
```

调用时可以写成：

```
DOUBLE (I) ;
```

上面两个例子用于说明子程序的描述格式参数表的格式和子程序的调用格式。调用子程序时，参数的传递可以通过下述两种方式进行。

一复制参数的方式。对于**OUT**结合方式，是在返回时复制；对于**IN OUT**结合方式，调用时复制过来，返回时再复制过去。

一传送地址的方式。通过地址的传送，使调用环境中的变量在子程序中成为可存取的变量。

ADA并未确定这两种方式中应当使用哪一种，而是由编译器任选或者根据某种优化的要求选择其中一种。因此，在ADA中任何指定传递方式的做法都是错误的，而在其他语言中可能是允许的。

结合方式为**IN**的参数可以有一个缺省值。例如：

```
FUNCTION SQR (X : IN FLOAT := 0.0) ……;
```

规定N的缺省值为零。调用时如果没有提供参数值，参数N则取缺省值0.0。当然，如果要处理的值与缺省值不同，调用时必须提供参数值。

ADA中还可以有不带参数的子程序。比如一个过程只涉及整体变量，或者一个子程序的参数都取缺省值而不需要提供参数，或者一个函数只回授结果（如：随机数RANDOM），在这类情况下，调用应在子程序名后加空括号。即：

子程序名（ ）；

例如：

A：＝B＊RANDOM（ ）＋C；

除无参子程序外，调用时参数的数目（除缺省值）、顺序以及各参数的类型都必须与子程序参数表所指明的相一致。如果是以参数名给出参数，那么调用时可以不考虑参数的顺序。

还应该说明的是，调用时的参数名没有必要与子程序中的参数名一致，这是模块化的一个特点。因此，用户可以使用别人写的子程序。唯一需要知道的是这个子程序要求提供参数的数目、顺序、类型以及子程序给出的是什么类型的结果。至于写子程序的人使用的参数名是什么是无关紧要的，用户可以按照自己的愿望去选择参数名。但是，类型必须一致。在ADA中，这种一致性是通过类型名来保证的。由此可以知道为什么使用匿名定义的类型是危险的。例如：

```
A：ARRAY（1..10）OF FLOAT；
PROCEDURE PROC（X：ARRAY（1..10）OF
                FLOAT）IS……END；
```

那么，

PROC（A）；

是非法的，因为在编译器看来，A和X是不同的类型。然而，如果改写成：

```
TYPE VECT IS ARRAY (1..10) OF FLOAT;
A : VECT;
PROCEDURE PROC (X : VECT) IS……FND;
```

那么

```
PROC (A) ;
```

是正确的。

在子程序中，参数总是以变量的形式出现。在调用中，结合方式为**OUT**或者**IN OUT**的参数也是以变量的形式出现。但是以**IN**为结合方式的参数可以是变量、常数，也可以是算术表达式。调用时首先计算算术表达式的值，然后将其值赋予参数。例如：

```
TYPE V IS ARRAY (1..3) OF FLOAT;
    W : V;
    T : FLOAT;
PROCEDURE PROC (X : IN V; Y : IN
                FLOAT) IS……END PROC;
```

那么，

```
PROC (W, 3 *T +4.0) ;
PROC (W, 5.0) ;
PROC ((1.0, 2.0, 3.0), T) ;
```

这些调用都是正确的。

前面已经提到，调用时可以以参数名提供参数。这样做似乎与“不管参数名”的原则是相违背的。这是因为对于程序库的子程序，用户没有必要搞清楚参数名是什么，只要

知道参数的数目、顺序、类型就够了。但是如果用户已知参数名，调用时重复参数名可以增加程序的易读性。因此调用时，可以按参数表中参数的顺序对应关系提供参数，也可以通过写出参数名的对应关系提供参数（不必按顺序），还可以两种方式混合使用。混合使用时，无名参数在前，并按顺序排列。从某一参数开始为有名参数，其后不得再有无名参数，即不能穿插使用。例如：

```
PROC (W, T);        --使用无名参数;
PROC (Y⇒5.0, X⇒ (1.0, 2.0, 3.0));
                        --使用有名参数，而
PROC ((1.0, 2.0, 3.0), Y⇒ 5 );
                        --使用混合参数。
```

以上调用都是正确的。但是

```
PROC (X⇒(1.0, 2.0, 3.0), 5.0);
```

是非法的。

4.1.2 含子程序的过程

在第三章关于Eratosthene筛法（例3.12）的例子中，已经看到过含有子程序的过程结构：一个过程（或函数）的说明部分包含了对另一过程（或函数）的说明。后一个过程中说明的参数只是这个过程的局部参数，而其余的是全局参数，这是一种能够实现分离编译的程序结构，但并不是唯一的结构，在介绍分离编译一节中将进一步讨论这个问题。

例4.1 设我们已经建立了如同例3.13和3.14的链表，其信息为字符串STRING（1..10）。试编写一个布尔函数和一个指针型函数；对布尔函数的要求是当链表中包含有字

符串K时，布尔函数为TRUE。对指针型函数的要求是指向字符串K，如果链表内不包含字符串K，指针指向NULL。并给出调用以上两个函数的例子。

解：

```
PROCEDURE LIST IS
    TYPE STR10 IS STRING (1..10) ;
    TYPE ELEMENT;
    TYPE PTR IS ACCESS ELEMENT;
    TYPE ELEMENT IS
        RECORD
            DATA : STR10;
            NEXT : PTR;
        END RECORD;
    DEP1 , DEP2 , DEP3 : PTR;
    K : STR10;
    FUNCTION PRESENT (K : IN STR10;
        DEP : IN PTR) RETURN BOOLEAN IS
        P : PTR
        BEGIN;
            P : =DEP;
            MHILE P/=NULL AND THEN
            P.DATA/=K LOOP
            P : =P.NEXT;
            END LOOP;
        IF P=NULL THEN
            RETURN FALSE;
        ELSE
```

```
            RETURN TRUE;
        END IF;
END PRESENT;
FUNCTION POSITION  (K : IN STR10;
    DEP : IN PTR)  RETURN PTR IS
    P : PTR;
    BEGIN
    P : =DEP;
    WHILE P/ =NULL AND THEN
        P.DATA/ =K LOOP
        P : =P.NEXT;
    END LOOP;
    RETURN P;
END POSITION;
BEGIN --LIST
    GET (X) ; --读入一个
    IF PRESENT (K⇒X, DEP⇒DEP 3) …;
    PUT (PRESENT (X, DEP 2) ) ;
    IF (POSITION (DEP⇒DEP 2 , K⇒Z) …;
            / =NULL……;
        DEP3 : =POSITION(X, DEP3) NEXT;
     ⋮
END LIST;
```

例4.2 将例3.14补成完整的过程。

解：**PROCEDURE** LIST **IS**

```
TYPE STR10 IS STRING (1..10);
TYPE ELEMENT;
TYPE PTR IS ACCESS ELEMENT;
TYPE ELEMENT IS
    RECORD
        DATA : STR10;
        NEXT : PTR;
    END RECORD;
DEP1,DEP2,DEP3……PTR;
X : STR10;
PROCEDURE INSERTION (K : IN STR10;
                     DEP : IN PTR);
    P,Q,R : PTR;
BEGIN
    P : =DEP;
    WHILE P/=NULL AND THEN P.DATA
                           <K LOOP
        R : =P;
        P : P.NEXT;
    END LOOP;
    Q : =NEW ELEMENT (K,P) ;
    R.NEXT : =Q;
END INSERTION;
BEGIN--LIST
    --建立链表
    GET (X) ;
```

```
        INSERTION (X,DEP 2);
        INSERTION (K⇒"TAGADAGADA",
                   DEP⇒DEP 1;
END LIST;
```

注意：

1.这里的过程INSERTION与第一章中的不同。第一章中的INSERTION是由INTO所指的链表的始端加进一个元素。

2.INSERTION过程中使用标识符 ELEMENT和NEXT 是 合法的，因为这些标识符未曾在过程中再定义，他们是全局变量。

4.1.3 无约束数组作为参数传递

结合方式为IN的参数可以是一个无约束类型。但是若调用时给出一个确定的值，那么这个类型即变为带约束型。最常见的例子是过程的参数为一个维长不确定的数组。比如，两个矢量的标乘积可以写为以下过程：

```
PROCEDURE PROD IS
    TYPE VECT IS ARRAY(INTEGER RANGE
                 <>) OF FLOAT;
    PR : FLOAT;
    PROCEDURE SCALPROD(X, Y : IN VECT;
                       XY : OUT FLOAT) IS
    BEGIN
        XY := 0.0;
        FOR I IN X'RANGE LOOP
```

```
            XY := XY + X(I) * Y(I);
        END LOOP;
    END SCALPROD;
    BEGIN
        SCALPROD ( (1.0, 2.0, 3.0) ,
            (4.0, 5.0, 6.0) , PR) ;
        PUT (PR) ;
    END PROD;
```

在这个例子中，以矢量 (1.0, 2.0, 3.0) 和(4.0, 5.0, 6.0) 调用SCALPROD使数组的维长确定为3。如果在调用过程中再次调用它，矢量的维长也必须为3。为了使被调用过程能适用于不同维长的矢量（即时而以一种维长的矢量调用，时而用另一种维长的矢量调用），可以编写一个子程序，通过分离编译植入程序库，从而一劳永逸。

这个例子中的被调用过程SCALPROD有一个漏洞，就是必须保证X'FIRST = Y'FIRST与X'LENGTH=Y'LENGTH。否则求积是不可能的。在ADA的最初文本中曾设置过一个特别的语句：

```
ASSERT X'LENGTH = Y'LENGTH;
```

在条件不满足时将引发一个异常。但这一语句在以后的ADA文本中被删除了。为达到同样的目的，用户只要在 **BEGIN** 之后写入下述语句即可：

```
IF X'LENGTH/ = Y'LENGTH THEN RAISE
    LENGTH_ERROR;
END IF;
```

当然还要给出对异常LENGTH-ERROR的处理办法。

例4.3 编写实现两个矩阵交换的过程并举出调用的例子。

```
PROCEDURE EX-MAT IS
    TYPE MATRIX IS ARRAY(INTEGER RANGE
          <>, INTEGER RANGE<>)OF FLOAT;
        V1,V2 : MATRIX;
    PROCEDURE EXCHANGE(M1,M2 : IN OUT
                  MATRIX) IS
        X : MATRIX;
        BEGIN
            X : = M1;
            M1 : = M2;
            M2 : = X;
        END EXCHANGE;
    BEGIN
        V1 : = (1,2,3,4,5,);
        V2 : (V2' RANGE⇒0);
        EXCHANGE (V1,V2);
    END EX_MAT;
```

例4.4 下面是两个矩阵乘积的过程。

```
PROCEDURE PRODMAT (A,B : IN MATRIX;
              C : OUT MATRIX) IS
    Z : FLOAT;
    BEGIN
        IF A'RANGE (2) /= B'RANGE THEN
            RAISE LENGTH_ERROR;
```

```
        END IF;
        FOR I IN A'RANGE LOOP
            FOR J IN B'RANGE(2) LOOP
                Z := 0.0;
            FOR K IN B'RANGE LOOP
                Z := Z + A (I,K) * B (K,J) ;
            END LOOP;
            C (I,J) := Z;
    END LOOP;
    END LOOP;
END PRODMAT;
```

4.1.4 递归调用

与大多数高级语言一样，在ADA中一个函数或过程可以调用它，称为递归调用。例如：阶乘的计算可以写成算术表达式：

FACT (N) ≡N*FACT (N－1)

其中，函数的定义为：

```
FUNCTION FACT (N : INTEGER) RETURN
        INTEGER IS
    BEGIN
        IF N > 1 THEN
            RETURN N*FACT (N－1) ;
            ELSE RETURN 1;
        END IF;
    END FACT;
```

例4·5 汉诺（Hanoi）塔

有三根竖立的柱子。第一根柱子上套着N个中心带孔的圆盘，盘的直径自下而上递减。现在要把这些盘上移到第二根柱子上，遵循的规则是：

1）一次只允许移动一个盘子；

2）盘子必须套在柱子上；

3）大盘不能放在小盘之上。

解：这是一个具有递归性质的问题。实际上，

1）移动一个盘子，只要写出：取盘的柱号和放盘的柱号。这样就可以得到实现移动盘子的清单；

2）由第i个柱子移动n个盘至第j个柱子，其步骤为：

先将n－1个盘由第i个柱子移到第3个柱子上；再将一个盘由第i个柱子移到第j个柱子上；最后将n－1个盘由第3个柱子移到第j个柱子上。

Hanoi塔的程序如下：

```
PROCEDURE HANOI IS
    N : INTEGER;
    PROCEDURE MVT (I,J : IN INTEGER) ;
        BEGIN
            PVT (I) ;
            PVT ( "⇒" ) ;
            PVT (J) ;
        END MVT;
    PROCEDURE DEPL (N,I,J : IN INTEGER);
        BEGIN
        IF N = 1 THEN MVT(I,J);
```

```
ELSE
DEPL (N - 1,I, 6 - I - J) ;
DEPL (1,I,J) ;
DEPL (N - 1, 6 - I - J,J) ;
END IF;
END DEPL;
BEGIN
GET (N) ;
DEPL (N, 1, 2) ;
END HANOI;
```

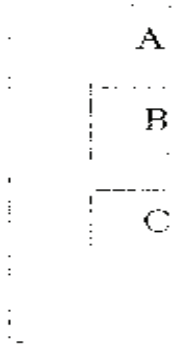


图 4.1

4.1.5 重载

假设在过程或分程序A中定义过程B与C（图4.1），那么从C开始可以调用B。现在，在C中重新定义过程B，那么这一定义将掩蔽外部的B。也就是说，从此时此刻起，C中对B的调用是指的新B（即C中的B）。如果在C中想调用外部的B，必须使用限制名，即A.B；它表示A中的B，这是关于定义域的经典规则。显然由于在ADA中能够使用限制名，还是带来了一定的灵活性（图4.2）。

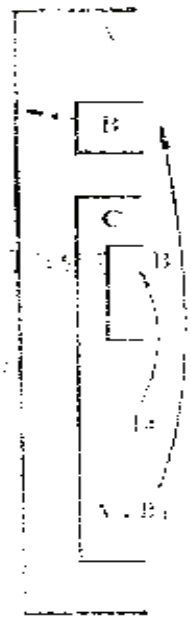


图 4.2

然而，ADA还提供了另一种可能：即便在同一个分程序（或过程）A中，我们仍可以定义第二个过程B。这个过程与第一个过程的参数类型和（或）参数数目是不同的。例如：

```
PROCEDURE B (X : T1) IS……;
```

PROCEDURE B（Y：T 2） **IS**……；

在这种情况下，我们把第二个B称为对第一个B的重载。

那么，在调用过程B时会不会产生错误呢？ADA根据参数的类型来判别调用的是哪一个B。比如：

M：T 1；

N：T 2；

B（M）；　--为调用第一个B

B（N）；　--为调用第二个B

任何含糊不清都是错误的。若同一个常数被重载为几种枚举性的元素，常常会发生这种错误。例如：

TYPE T 1 **IS**（I，J，K）；

TYPE T 2 **IS**（K，L，M）；

B（I）；　--不含糊，调用第一个B；

B（L）；　--不含糊，调用第二个B；

B（K）；　--是不正确的。

为了消除这种含糊性，可以使用指明类型的办法：

B（T1′（K））；　——调用第一个B。

4.1.6　算符的重载

在ADA语言中，一个函数名可以和一个标准算符名相同。我们把这个函数称为该标准标符的重载。这个函数能够为其他类型（与标准算符定义的类型不同）定义一个算符。这样做的好处在于，可以把新算符按习惯的写法用在算术表达式中。

例如，我们要对复数进行取反的操作，就可以将单目标符“－”重载：

```
FUNCTION "-" (A : COMPLEX) RETURN
                    COMPLEX IS……
```

同样，我们可以定义两个矩阵相乘的乘算符“*”：

```
FUNCTION "*" (A, B : MATRIX) RETURN
                    MATRIX IS……
```

如果C、A与B是矩阵，那么，

```
C : = A *B;
```

即实现了矩阵的乘法运算。当然，必须保证类型一致和没有含糊性。

下面让我们写一下矩阵乘算符的完整定义。

```
FUNCTION "*" (A,B : IN MATRIZ) RETURN
                              MATRIX IS
Z : FLOAT;
C : MATRIZ;
BEGIN
    IF A'RANGE (2) /= B'RANGE THEN
        RAISE NUMERIC_ERROR;
    END IF; ——引发一个预定义的异常
    FOR I IN A'RANGE LOOP
        FOR J IN B'RANGE (2) LOOP
            Z : = 0.0;
            FOR K IN B'RANGE LOOP
                Z : = Z + A (I,K) *B (K,J) ;
            END LOOP
            C (I,J) : = Z;
        END LOOP;
```

```
        END LOOP;
        RETURN C;
    END "*";
```

这个函数与例4.4几乎是完全相同的，只是把过程改写成函数，把过程名PRODMAT改成函数名“*”，从而使算符“*”对矩阵类型重载。

4.1.7 关于子程序参数

在实际应用中，子程序参数是非常有用的。譬如表示函数积分的积分函数，用于描绘一个函数曲线的过程等。对于这类情况，函数或过程的一个参数用来指明要处理的是哪一种函数。也就是说，一个子程序作为另一个子程序的参数，这是非常必要的。

子程序参数在FORTRAN以及某些文本的Pascal中是允许的。但在ADA中是不允许的：在ADA中，无论是类型还是子程序都不能作为另一个子程序的参数。

ADA解决这一问题的办法是使用类属元素。类属元素能够使一个子程序或一个程序包依赖于某些子程序或某些类型。

4.1.8 插入式参注

通常，一个子程序一次植入存贮器，每次调用时就跳到这一存贮地址上。这样做节省了存贮位置，但却延长了执行时间。如果在定义子程序时说明部分加上参注：

```
PRAGMA INLINE（子程序名；
```

那么，在每个调用点将开辟一个存贮区把子程序插入。从而

起到与汇编器的密指令相同的作用。这种做赢得了时间但占据了较多的存贮空间。

4.2 程 序 包

程序包（PACKAGE）是ADA最大的发明之一。一个程序包是一些资源的集合，所谓资源系指供程序使用的类型、变量和进行某种处理的子程序。

经常遇到的情况是用户使用已经存在的库程序包。库程序包推广了库函数的概念，并丰富了它的内容：程序包不仅提供了一个算法的运算工具（子程序），而且还提供了与之相联系的数据、类型及其结构。ADA提供这样的扩充是完全合理的，因为它的特点之一就在于不只是注意数据的处理而且对数据也给予充分的注意。

4.2.1 程序包结构

程序包结构的最大特点是把程序包特性的描述与这一特性的实现分开。在大多数情况下，一个程序包由**程序包式**、**程序包体**两部分组成。这两部分能够分别编译。这对于通过自上而下的分析来展开一个程序是非常有益的：如果已知用户有这样或那样的要求，可以将这些要求在用户程序包的程序包式中加以说明，而把编辑程序包体的工作留到以后完成（或交给助手去完成）。

程序包还有另外的优点。比如用户买了一个复数处理程序包。用户所感兴趣的只是有哪些运算可以用于复数。至于实现这些运算的手段对用户来说则是无关紧要的。因此，在

这种情况下，只有程序包式与用户打交道，程序包体并不与用户往来。

再进一步说，购入的程序包所选择的复数表示法细节对于用户来说也不感兴趣。这就是把复数的类型被处理成**私有型（PRIVATE）**。而不把结构细节告诉用户的道理。

ADA能够严格控制用户在某一对象上所进行的操作，这对于解决资源的安全和保护问题是很有意义的。例如，在一个口令管理系统中，不仅口令内部结构不告诉用户，而且禁止用户进行一切不必要的操作。这样的类型可以由**受限私有型（LIMITED PRIVATE）**来实现。对于受限私有型，**赋值和测等**都是严格禁止的。因为如果允许这两种操作，使用者有可能破译其结构。

因此，一个程序包可以描述如下：

```
PACKAGE 名 IS
        说明;
〔PRIVATE                          程序包式
        隐蔽说明; 〕
END 〔名〕
PACKAGE BODY 名 IS
        说明;
〔BEGIN                            程序包体
        初始化语句列; 〕
END 〔名〕;
```

在极个别情况下，程序包没有程序包体。这时程序包蜕化成一些类型说明及变量说明。也就是说，它只是一些说明的集合。这种程序包是通过**USE**语句活化的：

USE 程序包名；

例如：

```
PACKAGE DATA IS
    I,J : INTEGER;
    TYPE VECT IS ARRAY (1 . .100) OF
                        INTEGER;
    V,W : VECT;
END DATA;
PROCEDVRE USE_DATA IS
    USE DATA;
BEGIN
    I : = 5;
    J : =100;
    V (J) : = 1;
END USE_DATA;
```

USE语句与在说明段加入包含语句的作用是相近的。包含语句是通过参注PRAGMA来完成的：

PRAGMA INCLUDE（含说明段的文件名）；其意义是在希望的地方把某些一劳永逸地写成的源程序段落包括到正文中去。下面我们不再对这个参注作更多的讨论。

程序包体所包含的语句部分也是极少使用的。这个语句部分只包括一些数据初始化语句，它们完全可以由另外的方式来实现。

在程序包的说明式里，函数与过程只是由它们的首部引入。这些首部用于确定参数类型与结果的类型。而子程序体和以严格意义上讲的语句则在程序包体的说明部分给出，同

时需要重复函数与过程的首部。这些子程序常常是一些标准算符的重载，通过重载把标准算符的使用扩展到由程序包引入的新类型。

在程序包体的说明部分，还可能包含一些属于程序包的局部变量说明。这些变量具有一个重要的特性：它们形成一些**暂留对象**。即在对程序包中的过程的两次调用之间，它们的值保持不变。这些变量在程序包之外是不可存取的，它们只能由程序包的过程存取。这充分体现出程序包体是“掩蔽”的。

程序包体的语句列常常以异常处理的定义语句做结束。

最后，程序包式与程序包体的END之后是否要重复程序包名可由用户选定。可以重复其中一个，也可以两个都重复，以使程序易读为原则。

我们再回到第一章中关于复数的那个有代表的例子。当有一个私有型时，这个类型应该在可见部分说明：

```
PACKAGE……IS
      ⋮
TYPE P IS PRIVATE;
      ⋮
PRIVATE
      ⋮
TYPE P IS……;
```

P将在程序包式的私有部分完整地定义。

如果有私有型常数，它应该在可见部分以待定常数的形式说明，然后在私有部分完整地定义（初始化）。例如：

```
PACKAGE……IS
```

```
TYPE P IS PRIVATE;
    C: CONSTANT P;
PRIVATE
TYPE P IS……;
    C: CONSTANT P: = ……;
```

下面是另一个例子。我们要建立一个资源（例如文件）管理程序包，使用者有三种操作（对应于三个过程）：

1） NOUVELUT：新用户注册。此操作提供一个口令并将此口令编入目录（提供系统使用），用户得到一个用户号数；

2） GETRESSOURCE：对用户提供文件。使用者提供用户号、口令，并得到一个文件名；

3） ACCESRESSOURCE：为了使用文件。用户提供用户号、口令及文件名。不论是用户号不符，还是口令不符，都将受到惩处。

类型RESSOURCE将是**LIMITED PRIVATE**。因此它仅有的操作是GETRESSOURCE和ACCESRESSOURCE。整个程序包可以定义如下：

```
PACKAGE GESTION_RESSOURCES IS
TYPE PASSWORD IS STRING (1 . . 10);
                                --可见类型
TYPE RESSOURCE IS LIMITED PRIVATE
PROCEDURE NOUVELUT (MOT-PASSE: IN
                        PASSWORD) ;
PROCEDURE GETRESSOURCE (NUT: IN
                        INTEGER;
```

```
                    MOT__PASSE : IN PASSWORD;
                    NOMR : OUT RESSOURCE) ;
PROCEDURE ACCESRESSOURCE (NUT : IN
                    INTEGER; MOT__PASSE : IN
                    PASSWORD;
                    NOMR : IN RESSOURCE) ;
--可能还有一些与资源使用有关的
--其它参数。以上部分为提供给
--用户使用的全部内容。下面是程序包
--的私有部分:
PRIVATE
    TYPE RESSOURCE IS
        RECORD
        NAME : STRING (1 . . 10) ;
NUMBER : INTEGER;
        END RECORD;
END GESTION__RESSOURCES;
```

--在私有部分，资源的内部表示方法--与使用种类有关。但不管怎样，用--户是不知道的。

--程序包体将用内部的方式描述程序--包中所使用的数据，从外部来看它--是未知的。特别是那个包含有注册--用户、口令及可使用的资源的一览表。

```
PACKAGE BODY GESTION-RESSOURECES IS
    MAX__UT : CONSTANT INTEGER : =100;
                        --最多100个用户
    MAX__RES: CONSTANT INTEGER: =10;
```

——每个用户的最多资源数

NB__UT：INTEGER；——注册用户数

TYPE UTIL IS

RECORD

MOT：PASSWORD；口令

NBRES：1 . . MAX_RES；提供的资源数

END RECORD；

UTILIS：**ARRAY**（1 . . MAX_UT） **OF** UTIL；

RESS：**ARRAY** （1 . . MAX-UT，1 . . MAXR-ES）**OF** RESSOURCE；

--这里已经详细规定了要进行的管--理。这一部分内容用户是绝对不--可存取的，否则就可能作弊。比--如，得到的资源超过所允许的最--大数目，或者盗用别人的口令，--等等。下面是三个过程的梗概：

```
PROCEDURE NOUVELUT (MOT_PASSE : IN
                    PASSWORD)  IS
    BEGIN
        NB_UT : = NB_UT + 1 ;
        IF NB_UT > MAX_UT THEN
            RAISE 异常;
        END IF;
        PUT ("YOUR NUMBER WILL BE") ;
        PUT (NB_UT) ;
        UTILIS(NB_UT_) . MOT : = MOT_PASSE;
                                --记录口令
        UTILIS (NB_UT) . NBRES : = 0 ;
```

```
                                    --尚未得到资源
END NOUVELUT;
PROCEDURE GETRESSOURCE  (NUT : IN
                INTEGER;
          MOT_PASSE :  IN PASSWORD;
          NOMR : OUT RESSOURCE)  IS
     NR : 1 . . MAX_RES;
     BEGIN
     IF NUT NOT IN UTILIS'RANGE THEN
          以适当方式抗议;
     END IF; --用户号不符;
     IF UTILIS (NUT) . MOT/ = MOT_PASSE
          THEN 抗议;
     END IF; --口令不符
     NR: = UTILIS (NUT) . NBRES + 1;
          --如果超过MAX-RES自动引发异常
          NOMR: =生成适当的资源;
          UTILIS (NUT) . NBRES : = NR;
          RESS (NUT, NR) : = NOMR; --记录资源
END GETRESSOURE;
PROCEDURE ACCE SRESSOURCE  (NUT : IN
          INTEGER; MOT_PASSE : IN PASSW-
               ORD;
          NOMR :  IN RESSOURCE)  IS
          BEGIN
--与GETRESSOURCE相同的方式检查用
```

```
    --户号、口令并检查NOMR是否在记
    --录资料之中。
        FOR I IN 1 . . UTILIS(NUT).NBRES LOOP
            IF RESS (NUT, I) =NOMR THEN……
    --用户有权在程序包内部赋值、
    --测等及对资源存取
            END IF;
        END ACCESRESSOURCE;
    END GESTION__RESSORCES;
```

以上是GESTION__RESSOURCES程序包，使用这个程序包的过程如下：

```
    PROCEDURE UTILISATION IS
        USE GESTION-RESSOURCE;
        MYRESSOURCES : ARRAY (1 . . 5) OF
                    RESSOURCE; --用户资源表
        BEGIN
            NOUVELUT ("TAGADA");
            (YOUR NUMBER WILL BE 5--显示)
            GETRESSOURCE(5,"TAGADA", MYR-
              ESSOURCES (1)); --得到第一个资源
            ACCESRESSOURCE (5, "TAGADA",
            MYRESSOURCES(1));--使用第一个资源
    ……
    END UTILISATION;
```

问题：对于UTILISATION这个过程，用户是否能写：

```
    R : RESSOVRCE;
```

GETRESSOURCE（5，“TAGADA”，R）；
MYRESSOURCES（1）：=R；

答：不行。这是因为用户无权在程序包外部对受限私有型赋值。如果仅仅是私有型则是可以的。即有权写成：

R：RESSOURCE；
GETRESSOURCE（……，R）
ACCESRESSOURCE（……，R）；

例4.6 建立栈管理程序包

栈的管理应遵守LIFO（后进先出）的规则。设入栈元素的类型为T（为明确起见，设为由10个字符组成的字符串）。用户可以使用的只是几个子程序。有：进栈过程（在栈上加一个元素），出栈过程（在栈顶取走一个元素）和表示栈空、栈满的布尔型变量。出栈可以是一个以出栈元素为值的函数。程序包对栈的管理方式用户是不应该知道的。根据下面两个假设的管理方式，分别写出程序包说明式及程序包体：

——栈作为一个数组来管理、保持最后进栈元素的下标；
——栈通过地址分配及指针来管理。

解：
```
PACKAGE GESTION_PILE IS
    TYPE T IS STRING (1 . . 10);
    PROCEDURE IN_PILE (OBJECT : T);
    FUNCTION OUT_PILE RETURN T;
    PILE_FULL,PILE_EMPTY : BOOLEAN;
END GESTION_PILE;
```

通过数组管理，程序包体可以写成：

```
PACKAGE BODY GESTION_PILE IS
```

```
SIZE : CONSTANT INTEGER : =100;
--设计者可以取其他的缺省值
PILE : ARRAY (1 . . SIZE) OF T;
POINTER : INTEGER RANGE O · · SIZE;
PROCEDURE IN__PILE (OBJECT : T) IS
    BEGIN
        POINTER : =POINTER + 1 ;
        PILE (POINTER) : =OBJECT;
        IF POINTER=SIZE THEN
                PILE-FULL : =TRUE;
        END IF;
    END IN-PILE;
FUNCTION OUT-PILE RETURN T IS
    BEGIN
        POINTER : =POINTER - 1 ;
        IF POINTER<= 0 THEN
          PILE__EMPTY : =TRUE;
        END IF ;
    END OUT__PILE;
    BEGIN --初始化
        POINTER : = 0 ;
        PILE-EMPTY : =TRUE;
        PILE__FULL : =FALSE;
    END GESTION__PILE;
```

如采用动态存贮器管理的方式，程序包体如下：

```
PACKAGE BODY GESTION__PILE IS
```

```
SIZE : CONSTANT : =100;
TYPE ELEM
TYPE PTR IS ACCESS ELEM;
TYPE ELEM IS RECORD
    DATA : T ;
    PREC : PTR;
    END RECORD;
POINTER : PTR
NUMBER : INTEGER RANGE O . . SIZE;
PROCEDURE IN-PILE (OBJECT : T)  IS
    P : PTR :
    BEGIN
        NUMBER : =NUMBER + 1 ;
        IF  NUMBER=SIZE  THEN  PILE_
                    FULL : =TRUE;
        END  IF;
        P : =NEW ELEM ( (OBJECT,
              POINTER) ) ;
        POINTER : =P;
    END IN-PILE;
FUNCTION OUT-PILE RETURN T IS
                X : ELEM;
    BEGIN
        X : =POINTER • ALL;
            --可以加一个存贮位置的显式释放
        POINTER : =X • PREC;
```

```
            NUMBER : = NUMBER - 1 ;
            IF NUMBER <= 0 THEN
               PILE_EMPTY : = TRUE;
            END IF;
            RETURN X . DATA;
         END OUT_PILE;
      BEGIN --初始化
         POINTER : = NULL
         NUMBER : = 0 ;
         PILE_EMPTY : = TRUE;
         PILE_FULL : = FALSE;
      END GESTION_PILE;
```

使用这个程序包的形式是：

```
USE GESTION_PILE
X . Y : T;                    ] 在说明部分
IF NOT PILE_EMPTY THEN X : = OUT_
   PIL E (1) ;
END IF;
IF NOT PILE_FULL THEN IN_PILE (Y) ;
END IF;
```

上述调用，如果没有检验保护，可能引发异常。因此可以把这些检验植入过程中，注意：对于无参数函数的调用，须在函数名后加一个空括号。

4.2.2 程序包的使用—USE语句

如果一个模块（分程序或子程序）包含了对一个程序包

的说明，那么这个程序包在该模块所包围的所有模块中是可存取的。这是通常所说的可见性法则。而一个程序包可存取，是指在它的可见部分定义的元素是可存取的。

例如：

```
PROCEDURE A IS
    PACKAGE P IS
        X : ……;
    END P;
    PROCEDURE B IS
    ③
    BEGIN --B
    ①
    END B;
    ⋮ ④
    BEGIN --A
    ②
    END A;
```

在这个过程轮廓中，过程A中有程序包说明，因此程序包P的元素X在①②中是可存取的。存取通过限制名 P.X 进行。为了避免使用程序包名P，可以在③和④中放置

USE P;

语句，这时存取X只需要简单地用元素名X。

在这里，USE语句的作用并不是使X成为可见的，X已经是可见的了。USE语句的作用仅仅在于简化引用X的方式，它避免了使用限制名。

USE语句可以出现在说明部分的任何地方。

一个（或几个）USE语句有可能会使一个名的意义含糊，要避免这种含糊性还得使用限制名。

避免使用限制名的另一种办法是使用——**RENAMES**语句，对于程序包内的变量(如P·X) 通过这一语句重新命名为一变量，其形式为：

PX：类型 **RENAMES** P·X；

对于程序包内的过程（如：P.ZOZO）重新命名的形式为：

PROCEDURE TOTO（参数）**RENAMES**

P.ZOZO；

在例3·12 中 我们曾经提到，当我们使用标准输入/输出过程时需要使用USE语句（USE之后加上包含PUT与GET两个过程的程序包名），道理就在这里。

4.3 分 离 编 译

子程序、程序包和下面将要介绍的类属、任务都能一个一个地被编译。从这个意义上来讲它们都是编译单位。

对于标准操作库程序，情况更是如此。这些程序为用户所使用，但它又独立于用户程序，已经被一劳永逸地编译过。

4.3.1 WITH语句

在用户的编译单位之首，必须详细列出要使用的而且已经编译过的程序包清单。**WITH**语句正是用于这一目的。

WITH语句的形式如下：

WITH 程序包名，〔程序包名〕；

例如：

```
WITH INPUT_OUTPUT, REAL_ARITHMETIC;
PROCEDURE MY_PROGRAM IS……
```

这将使INPUT-OUTPUT程序包和REAL-ARITHMETIC程序包（假定它们存在于程序库中）连接到用户程序上。

注意不要混淆**WITH**语句和**USE**语句。**WITH**语句只有一个，置于用户编译单位之首；**USE**语句就不同了，在每一个过程的说明部分都可以使用**USE**语句，而且引入一个程序包用一个**USE**语句，引入两个程序包就得用两个**USE**语句，**USE**语句的作用在于不使用限制名而直接存取程序包的对象。

如果编译主程序（包含程序其他部分的程序）需要引用在程序包库中的一个程序包内已经定义的类型或一个子程序包，那么可以同时使用WITH语句和USE语句，例如：

```
WITH A, B, C;
USE A;
PROCEDURE AA IS
AAA : AAAA; --AAAA已在A中定义
USE A-A; --A-A为在A中已定义的程序包
    ⋮
PROCEDURE BB IS
    USE B, C;
    ⋮
```

只有那些显式引用的程序包，需要在**WITH**中列举。例如：

程序①

```
PROCEDURE PP IS
PACKAGE   PA IS
:
END PA;
PACKAGE BODY PA IS
:
END PA;
PROCEDURE SP IS
    USE PA;
END SP;
BEGIN --PP
:
END PP;
```

程序②

```
PACKAGE PA IS
:
END PA;
PACKAGE BODY PA IS
:
END PA;
WITH PA;
PROCEDURE PP IS
    PROCEDURE SP IS
    USE PA;
    END SP;
BEGIN --PP
:
END PP;
```

将程序①改写成程序②，用户即可分别编译程序包PA与过程PP。PA的程序包式和程序包体也可以分离编译。②中的每个大括号表示一个可以独立的编译单位。我们可以看到，由①改写成②只需在 **PROCEDURE** PP之前加一个 **WITH** 语句。

4.3.2 SEPARATE 语句

我们可以分离编译一个过程或一个过程内的程序包体。在被分离出来的过程或程序包体的位置上需要加一个 **SEPARATE**语句，例如：

PROCEDURE 名（参数） **IS SEPARATE;**

或者 FUNCTION 名（参数） IS SEPARATE；
PACKAGE BODY 名 IS SEPARATE；

而被分离出来的过程体成程序包体则以如下形式给出：

```
SEPARATE（被分离的分程序名）；
PROCEDURE 名 IS
    ⋮
```

下面的例子，①为分离前的形式，②为分离后的形式：

①

```
WITH BIBLI;
  PROCEDURE PP IS
    PACKAGE PA IS
      FUNCTION F (X : FLOAT)
        RETURN FLOAT;
      PROCEDURE G (Y, Z : FLOAT) ;
    END PA;
    PACKAGE BODY PA IS
      FUNCTION F (X : FLOAT)
                RETURN FLOAT IS
        ⋮
    END F;
    PROCEDURE G (Y, Z : FLOAT) IS
                        --使用BIBLE
        ⋮
      END G;
    END PA;
    PROCEDURE SP (A, B : FLOAT) IS
```

```
            USE PA;
              ⋮
          END SP;
      BEGIN --PP
              ⋮
      END PP;
              ②
  PROCEDURE PP IS
      PACKAGE PA IS
          FUNCTION F (X : FLOAT) RETURN
            FLOAT;
          PROCEDURE G (Y, Z : FLOAT) ;
      END PA;
          PACKAGE BODY PA IS SEPARATE;
  PROCEDURE SP (A,B : FLOAT) IS SEPARATE;
  BEGIN--PP
              ⋮
  END PP;
  SEPARATE  (PP)
  PROCEDURE SP (A, B : FLOAT)  IS
      USE PA;
              ⋮
  END SP;
  SEPARATE  (PP)
  PACKAGE BODY PA IS
      FUNCTION F(X : FLOAT) RETURN
```

```
                    FLOAT IS
        ⋮
    END F;
    PROCEDURE G(Y, Z:FLOAT)IS SEPARATE;
END PA;
WITH BIBLI;
SEPARATE (PP.PA)
PROCEDURE G (Y,Z : FLOAT)  IS
        --使用BIBLI
    ⋮
END G;
```

程序②有四个分离编译单位，PA与SP构成PP的子单位而G构成PA的子单位。注意在程序②中，对于PA的子单位**SEPARATE**是如何使用所选名的。另外由于过程G使用程序包BIBLI，因此在这个编译单位的首部使用了**WITH**语句。

例4.7 对于例4.6分离编译程序包体和过程IN__PILE、函数OUT__PILE。

解：

```
PROCEDURE UT—PILE IS
    PACKAGE GESTION—PILE IS
        TYPE T IS……;
        PROCEDURE IN—PILE (OBJECT : T) ;
        FUNCTION OUT—PILE RETURN T;
        PILE_FULL,PILE_EMPTY:BOOLEAN;
END GESTION__PILE;
    PACKAGE BODY GESTION-PILE IS
            SEPARATE;
```

```
    X, Y : T;
    BEGIN --UT_PILE
        IF NOT PILE_EMPTY THEN
            X = OUT_PILE ( ) ;
        END IF;
        IF NOT PILE_FULL THEN
            OUT_PILE (Y) ;
        END IF
    END UT_PILE;
SEPARATE (UT_PILE)
PACKAGE BODY GESTION_PILE IS
    SIZE : CONSTANT INTEGZR : = 100;
    PILE : ARRAY (1..SIZE) OF T;
    POINTER : INTEGER RANGE 0..SIZE;
    PROCEDURE IN_PILE (OBJECT : T) IS
                SEPARATE;
    FUNCTION OUT_PILE RETURN T IS
              SEPARATE;
END GESTION_PILE;
SEPARATE (UT_PILE.GESTION_PILE)
PROCEDURE IN_PILE (OBJECT : T) IS
    BEGIN
        POINTER : = POINTER + 1 ;
        PILE (POINTER) : = OBJECT;
        IF POINTER = SIZE THEN PILE_
           FULL : = TRUE;
```

```
        END IF;
    END IN_PILE;
    SEPARATE (UT—PILE.GESTION—PILE)
FUNCTION OUT_PILE RETURN T IS
    BEGIN
        POINTER : =POINTER - 1;
        IF POINTER<= 0 THEN PILE_
                        EMPIY : =TRUE;
        END IF;
        RETURN PILE (POINTER + 1) ;
    END OUT_PILE;
```

4.3.3 编译顺序

对需要编译而相互依赖的模块进行编译的顺序是：在模块A的**WITH**中所列举的所有模块应当在A之前编译；所有的子程序体或程序包体应在相应的子程序说明或程序包式之后编译。

再编译的规则是：修改子单位不需要对包含它的单位再编译，反之对一个单位的修改则要求对它所包含的子单位进行再编译。

4.4 类属元素

在4.1.7中我们曾提到，一个函数作为过程的参数是由类属实现的。类属语句（**GENERIC**）通过把变量、类型或子程序作为形式参数使得子程序、程序包或任务参数化。这些

形式参数分别称为变量参数、类型参数及子程序参数。

一个类属对象仅仅是一个实际对象的模型。为了使用这一模型，用户需要用确定的值定义类属参数，这样即可产生属于这一模型的实际对象，这种产生方式叫做**衍生（INSTANTIATION）**。

我们仍然以交换两个对象的例子说明这一概念。显然，无论对象的类型如何，交换方法则是相同的。因此我们可以把所有的交换过程汇总起来，把要交换的对象抽象为一个形式参数定义一个类属过程：

```
GENERIC
    TYPE T IS PRIVATE
    PROCEDURE EXCHANGE(X,Y:IN OUT T) IS
        Z : T;
        BEGIN
            Z : = X;    X : = Y;    Y : = Z;
        END EXCHANGE;
```

在这型，并没有定义任何实际的交换过程。现在假定我们要交换整型数，那末我们可以衍生一个用于整型数的交换过程：

```
PROCEDURE EXCHANGE_INTEGER IS
    NEW EXCHANGE (INTEGER) ;
```

EXCHANGE_INTEGER 则是一个实际可以使用的真实过程，可以写成：

```
EXCHANGE_INTEGER (I, J) ;
```

如法炮制，可以得到用于各种类型的交换过程。例如对于矩阵交换：

```
TYPE MATRIX IS ARRAY (1..10, 1..10)
        OF FLOAT;
    A, B: MATRIX;
    PROCEDURE EXCHANGE_MAT IS
        NEW EXCHANGE (T⇒MATRIX);
```

对于这个过程的调用，可以写成：

```
EXCHANGE_MAT (A, B);
```

这个例子指出，用户可以列举类型名加习惯的⇒符号确定类属参数。一个过程的衍生，不必重复过程的诸参数，这些参数可以通过检查模型找到。

如果几个衍生使用同样的名，那末这就是衍生形成的重载。

4.4.1 变量参数

变量参数的使用方法与子程序参数的使用方法非常相似，衍生时由数值替代变量参数，**OUT**形式的变量参数则没有替代对象。

```
GENERIC
    SIZE: NATURAL;
PACKAGE ACTION_ON_TABLE IS
TAB: ARRAY (1..SIZE) OF INTEGER;
```

衍生

```
PACKAGE TABLEAU IS NEW
            ACTION_ON_TABLE (100);
```

即把数组的维长固定为100。

4.4.2 类型参数

类型参数可以表成如下形式：

（1）**TYPE** 形式名 **IS**[**LIMITED**] **PRIVATE**；衍生时形式名可由除无约束数组外的任何类型代替。如果指明是受限私有型**LIMITED**，那么就没有必要定义赋值与测等。这时形式类型名甚至可以由任务类型来代替。

（2）**TYPE** 形式名 **IS**（〈 〉）；

形式名可以由离散类型来代替。

（3）**TYPE** 形式名 **IS RANGE** 〈 〉

形式名可以由整型来代替。

（4）**TYPE** 形式名 **IS DIGITS** 〈 〉

形式名可以由浮点实型来代替。

（5）**TYPE** 形式名 **IS DELTA** 〈 〉

形式名可以由定点实型来代替

（6）**TYPE** 形式名 **IS ARRAY**(下标指明**OF**类型)；

形式名可以由一切相容的数组（下标数目相同，约束相同）来代替。

同一类属的类型参数是可以相互依赖的，例如：

```
GENERIC
    TYPE OBJECT IS PRIVATE
    TYPE TABLE IS ARRAY (INTEGER RANGE
                         〈 〉) OF OBJECT;
GENERIC
    TYPE INDEX IS (〈 〉);
    TYPE VECT IS ARRAY(INDEX) OF FLOAT;
```

对于类型参数最通常的指明形式是**PRIVATE**，ADA还

引入一些别的指明，用于更细致地控制类型参数。

例4.8 以进栈元素的类型及栈的许用长度为参数试将例4.6栈管理程序包参数化。

解：
```
GENERIC
    SIZE : INTEGER;
    TYPE T IS PRIVATE;
PACKAGE GESTION_PILE IS
    PROCEDURE IN_PILE (OBJECT : T) ;
    FUNCTION OUT_PILE RETURN T;
    PILE_FULL, PILE_EMPTY : BOOLEAN;
END GESTION_PILE;
PACKAGE BODY GESTION_PILE IS
    PILE : ARRAY (1..SIZE) OF T;
    POINTER: INTEGER RANGE 0..SIZE;
    PROCEDURE IN_PILE (OBJECT : T) IS
        BEGIN
            POINTER := POINTER + 1;
            PILE (POINTER) := OBJECT;
            IF POINTER = SIZE THEN;
                PILE_FULL := TRUE;
            END IF;
        END IN_PILE;
    FUNCTION OUT_PILE RETURN T IS
        BEGIN
            POINTER := POINTER - 1;
            IF POINTER <= 0 THEN PILE_
```

```
                    EMPTY : = TRUE;
                END IF
            END OUT_PILE;
    BEGIN
        POINTER : = 0;
        PILE—EMPTY : = TRUE;
        PILE—FULL : = FALSE;
    END GESTION_PILE;
```

我们可以用下述语句衍生一个许用长度为200的整型栈和一个许用长为100的实型栈：

```
    PACKAGE PILE_INT IS NEW GESTION_PILE
                (200, INTEGER) ;
    PACKAGE PILE—REAL IS NEW GESTION_
                PILE  (SIZE⇒100,  T⇒FLOAT) ;
```

PILE_INT和PILE_REAL是可以使用的栈。例如在程序中可以有：

```
    I : INTEGER;
    R : FLOAT;
        ⋮
    I : = PILE_INT . OUT_PILE [ ] ;
    PILE_REAL.IN_PILE (R) ;
```

我们也可以把这些栈中的过程再命名。例如：

```
    PROCEDURE EMP—R (R : FLOAT) RENAMES
                PILE_REAL.IN_PILE;
```

若想使R进栈，可以写作：

```
    EMP_R (R) ;
```

4.4.3 子程序参数

一个过程或一个函数也可以做为形式参数。这些参数在类属部分中出现在变量参数与类型参数之后。子程序参数有三种表达方式：

（1）**WITH PROCEDURE** 形式名（参数）；

WITH FUNCTION 形式名(参数)**RETURN** 类型；过程和函数的参数也可以是同一类属的形式参数。例如：

```
GENERIC
    TYPE REAL IS DIGITS < >;
    WITH FUNCTION F (X : REAL) RETURN
      REAL;
PROCEDURE ZOZO IS
    :
Z : = F (X) ;
    :
```

这个类属的衍生例子如下：

```
TYPE R IS DIGITS 10;
FUNCTION TRUC (X : R)  RETURN R IS
    :
    END TRUC;
PROCEDURE TOTO IS NEW ZOZO
                (REAL⇒R,  F⇒TRUC) ;
```

对于这种形式，衍生时总是需要提供实际的子程序。而对于第二种形式可以使用子程序缺省名。

(2) **WITH** FUNCTION 形式名 (参数) **RETURN**

类型 IS 缺省名;

WITH PROCEDURE 形式名（参数）

IS缺省名;

例如：

```
GENERIC
    WITH PROCEDURE A (Z : REAL) IS B;
    PROCEDURE D IS……;
```

衍生

```
PROCEDURE DC IS NEW D (C) ;
```

以过程C为参数。而衍生

```
PROCEDURE DB IS NEW D;
```

以缺省过程B为参数。

（3）当子程序参数是系统的一个标准算符时，采用第三种表达方式。例如：

```
GENERIC
    TYPE T IS PRIVATE;
    WITH FUNCTION "*" (A, B : T)
        RETURN T IS < > ;
FUNCTION SQR (A : T) RETURN T IS
BEGIN
    RETURN A*A;
END SQR;
```

现在定义一个矩阵的乘积：

```
TYPE MATRIX IS ARRAY (1..10, 1..10)
        OF FLOAT;
FUNCTION PRODMA (A, B : IN MATRIX)
```

```
        RETURN MATRIX IS
    BEGIN
    :--矩阵之积
    END PRODMA;
```

那么，通过SQR的衍生可以计算矩阵的平方，

```
    FUNCTION SQRMA IS NEW SQR
                (MATRIX, PRODMA);
```

在这个衍生中，把类属中的两个形式参数（类型参数T、子程序参数“*”）由实际参数MATRIX和PRODMA来确定。然而对于整型数的平方，衍生形式为：

```
    FUNCTION INT SQR IS NEW SQR(INTEGER);
```

这时，没有必要给定实际参数代替“*”。因标准“*”算符的重载可以用于这里给定的类型。形式算符在这里取标准算符。

例4.9 将求函数积分的函数参数化，并给出衍生的例子。为求函数F由A至B的积分采用下述梯形公式：

$$H = (B - A) / N$$

$$S = H \left[(F(A) + F(B)/2 + \sum_{i=1}^{N-1} F(A + iH) \right]$$

解：

```
GENERIC
    WITH FUNCTION F (X : FLOAT)
                RETURN FLOAT;
FUNCTION INTEG (A, B : IN FLOAT; N : IN
            INTEGER) RETURN FLOAT IS
    S, H, X : FLOAT;
    BEGIN
```

```
            S : = (F (A) +F (B) ) /2.0;
            H : = (B-A) /N;
            X : = A;
            FOR I IN 1..N-1 LOOP
                X : = X+H;
                S : = S+F (X) ;
                END LOOP;
            RETURN S*H;
        END INTEG;
    USE FUNCTIONS_MATH; --在这个程序包内包含
```
有SIN和COS函数，在这个**USE**语句之后，可以衍生：
```
    FUNCTION INTEG_SIN IS NEW INTEG(SIN),
    FUNCTION INTEG_COS IS NEW INTEG(COS);
```
在程序中调用形式如下例所示：
```
    Z : = INTEG_SIN (0.0, 2.0*PI, N=>50);
    W : = INTEG_COS (0.0, PI/2.0, 20) ;
```
上述过程可以分离编译：
```
    WITH FUNCTIONS_MATH;
    USE FUNCTIONS_MATH;
        PROCEDURE INTEGRATION IS
            GENERIC
                WITH FUNCTION F(X : FLOAT)
                                RETURN FLOAT;
                FUNCTION INTEG  (A, B : IN
                            FLOAT;
                        N: IN INTEGER) RETURN
```

```
                                FLOAT IS SEPARATE;
            ⋮
          FUNCTION INTEG_SIN IS NEW INTEG
                                    (SIN) ;
BEGIN
      ⋮
Z: = INTEG_SIN (0.0, 2.0*PI, N⇒50) ;
      ⋮
END INTEGRATION;
SEPARATE (INTEGRATION)
FUNCTION INTEG (A, B: IN FLOAT; N: IN
          INTEGER) RETURN FLOAT IS
    S, H, X: FLOAT;
    BEGIN
        S: = (F(A)+F(B))/2.0;
        H: = (B-A)/N
        X: = A;
        FOR I IN 1..N-1 LOOP
            X: = X+H;
            S: = S+F (X) ;
            END LOOP;
        RETURN S*H;
    END INTEG;
```

还有一点需要指出的是：派生型的定义与类属衍生定义是非常相似的，例如：

```
TYPE T IS NEW INTEGER RANGE 1..N;
```

PACKAGE P **IS NEW** Q (SIZE⇒N)；

本章介绍的子程序、程序包、类属以及在第六章将要研究的任务是ADA程序模块化的重要工具。其中程序包概念的引入是ADA语言最重要的发明。它允许把程序分割成可以分离编译的一些单位，这对建立程序库是非常方便的。另一方面，程序包结构能够做到让用户使用资源而又不让用户知道资源的内部结构。

本章介绍的另一个基本概念是资源的参数化。使用这些资源的方法有：

——在一个无约束类型上加入约束；

——对子程序的参数赋值（在经典语言中这是唯一的手段）；

——对类属元素进行衍生。

第五章　标准环境和输入输出

5.1　标　准　环　境

所有 ADA 程序都是在预先定义了数据类型和算符的环境中运行的。这些定义集中在标准程序包里，犹如在用户程序的最外层模块中加入下述语句：

WITH STANDARD; **USE** STANDARD;

PROCEDURE PRINCIPAL—PROGRAM **IS**……;

便可引入这些定义。但是，这些定义是自动引入的，因此用户用不着写第一行。WITH语句的作用在于指出在标准程序包的后面加入指定的程序包。

下面给出标准程序包的轮廓。完整内容见参考手册（其中凡是由编译器确定的，我们记作dc）。

```
PACKAGE STANDARD IS
TYPE BOOLEAN IS (FALSE, TRUE);
FUNCTION "NOT" (X : BOOLEAN) RETURN
                BOOLEAN;
FUNCTION "AND" (X, Y : BOOLEAN) RETURN
                BOOLEAN;
                --对算符 "OR" 与 "XOR" 也类似。
TYPE SHORT—INTEGER IS RANGE dc;
TYPE INTEGER IS RANGE dc;
```

```
        --对LONG—INTEGER也同样;
    FUNCTION "+" (X : INTEGER) RETURN
        INTEGER;
        --对称符 "-" ABS和双目算符
        -- "+" 、 "-" 、 "*" 、 "/" 、
           "REM" 、 "MOD" 也同样
    FUNCTION "**" (X:INTEGER, Y:INTEGER
        RANGE
      0..INTEGER'LAST)RETURN INTEGER;
        --对其他两个整型也给出同样的算符。
        --对三个实数型SHORT_REAL,
          FLOAT
        --and LONG_FLOAT也同样
    TYPE FLOAT IS DIGITS dc RANGE dc;
        --对三个实型给出以下算符的说明:
        --单目算符 ("+", "-", ABS)
          和双目算符
        --("+","-","*","/"and"**")
```

例如:

```
    FUNCTION "**" (X : FLOAT; Y : INTEGER)
        RETURN FLOAT;
        --定义ASCII字符:
    TYPE CHARACTER IS
     (nul,soh,……,' ',' | ','0', '1','2',
……, '9', ':', ……, '@', 'A', 'B',
'C',…… 'Z', '[',…… 'a', 'b',…… 'z',
```

'{', '|', '}', '~', del);
--标识符和字符相对应的程序包
PACKAGE ASCII **IS**
NUL:**CONSTANT** CHARACTER:=nul;
⋮
DEL:**CONSTANT** CHARACTER:=del;
EXCLAM:**CONSTANT** CHARACTER:='!';
⋮
LC_A:**CONSTANT** CHARACTER:='a';
⋮
LC_Z:**CONSTANT** CHARACTER:='z';
--完整的对应表见附录Ⅱ
END ASCII;
—预定义的其它类型和子类型:
SUBTYPE NATURAL **IS** INTEGER **RANGE**
1..INTEGER'LAST;
SUBTYPE PRIORITY **IS** INTEGER **RANGE** dc;
TYPE STRING **IS ARRAY** (NATURAL **RANGE**
<>) **OF** CHARACTER;
TYPE DURATION **IS DELTA** dc **RANGE** dc;
—各种异常。例如:
CONSTRAINT_ERROR:EXCEPTION;
--由编译器定义的程序包SYSTEM
PACKAGE SYSTEM **IS**
TYPE SYSTEM_NAME **IS** 枚举类型dc;
NAME:**CONSTANT** SYSTEM_NAME:=dc;

```
        STORAGE_UNIT:CONSTANT:=dc;
        MEMORY_SIZE:CONSTANT:=dc;
        MIN_INT:CONSTANT:=dc;  ——最小整数
        MAX_INT:CONSTANT:=dc;  ——最大整数
    END SYSTEM;
    PRIVATE
        FOR CHARACTER USE (0, 1, 2, ……
            126, 127);
        --定义ASCII码用连续整数
        PRAGMA PACK (STRING);
    END STANDARD;
```

除标准程序包之外，在ADA 语言中预先定义的其他程序包（或过程）还有：

——程序包CALENDAR(日历)和类属过程SHARED_VARIABLE_UPDATE（共用变量刷新），这个过程用于任务之间的联系。

——类属过程UNCHECKED_DEALLOCATION（存贮单元分配免检）和UNCHECKED_CONVERION（转换免检）。

——输入/输出程序包。它包括一般输入/输出程序包INPUT_OUTPUT、正文输入/输出程序包TEXT_IO与控制字符输入/输出程序包LOW_LEVEL_IO。

5.2 输入/输出程序包

在高级语言中，输入/输出的确定把设计者置于进退维谷的困境：要么不做任何规定，结果是不同的编译器规定了

不同的标准（如：ALGOL），因而缺乏可移植性；要么确定一个输入/输出指令族（如：FORTRAN）或者标准过程（如：PASCAL），因而缺乏灵活性。但是，ADA既具有充分地灵活性(它采用参数化法确定输入/输出程序包，对这些程序包做了必要的规定，使得不同的编译器之间有良好的一致性)；同时，借助于重载和衍生，又保证了各种特殊情况下的可移植性。

5.2.1 一般输入/输出程序包INPUT_OUTPUT

这个程序包提供文件的类型和对文件的操作。文件是同一类型元素的无穷序列。包括的参数就是元素的类型，用户可以根据所需要处理的数据类型衍生这个一般输入/输出程序包。在每个衍生出的程序包中，定义三种类型的文件：

只读文件IN_FILE；

只写文件OUT_FILE；

读写文件INOUT_FILE。

例如，我们可以把这个一般输入/输出程序包衍生成为一个实数型输入/输出程序包：

```
PACKAGE I_O_REAL IS NEW INPUT_OUTPUT
                (ELEMENT_TYPE⇒FLOAT);
```

这样衍生之后，只要我们写：

```
RESULTS: I_O_REAL.OUT_FILE;
```

就可以确定RESULTS是一个只写文件。如果只有实型数要处理，也可以用下列语句确定文件RESULTS：

```
USE I_O_REAL;
RESULTS:OUT_FILE;
```

象RESULTS这样的名字是文件的内部标识符。所有的文件操作将有相同的格式。例如文件的写操作为：

WRITE (RESULTS, DATA)；

也就是说，RESULTS在程序中是作为一个文件来引用的。文件也有一个由字符串组成的外部名称，它用在开发系统中作为标识。例如，一些名称可以用来确定文件驻留在哪一个外围设备上。有一定数量的操作用于实现内部文件和外部名之间的联系。字符串的组成取决于系统。我们不详细研究这些问题，权把这种字符串记作“×××××××”。

对于文件的操作有：

a）建立文件；

b）打开文件；

c）关闭文件；

d）删除文件；

e）顺序读文件；

f）顺序写文件；

g）返绕读文件；

h）返绕写文件；

i）设置顺序读文件的位置；

j）设置顺序写文件的位置；

k）截短文件

下面逐次阐明这些操作的意义：

建立文件：

在进行所有操作之初，必须建立一个文件，这个文件便是过程CREATE（建立文件）的对象。过程CREATE有两种模式分别用于建立OUT__FILE文件和INOUT__FILE

文件。但是IN_FILE文件不能作为过程CREATE的对象，因为建立文件的过程意味着写一个文件，而IN_FILE是不能被写的只读文件。

```
PROCEDURE CREATE(FILE:INOUT OUT_FILE;
                 NAME: IN STRING);
PROCEDURE CREATE (FILE: INOUT INOUT_
                 FILE; NAME: IN STRING);
```

例如：

```
CREATE (FILE⇒RESULTS; NAME⇒ "@0:
ZOZO");
```

式中，RESULTS是文件的内部标识符，"@0:ZOZO"是文件的外部名称。

打开文件：访问一个文件之前，这个已存在的文件必须是打开的。过程OPEN（打开文件）用来建立这种状态：

```
PROCEDURE OPEN (FILE: IN OUT IN_FILE;
                NAME: IN: STRING);
```

注意：建立文件CREATE的操作包含了只写文件的打开操作。因此，如果在建立RESULTS文件之后，接着在这个文件上进行写操作，系统将知道这是"@0:ZOZO"文件。

输入、输出程序包的大多数过程可以重载。所谓重载是指过程的名称相同但参数类型或参数数目不同。例如过程OPEN中，参数FILE的类型只是读文件IN_FILE。但是FILE的类型也可以是只写文件OUT_FILE或读写文件IN-OUT_FILE，我们把这种现象叫做过程OPEN对OUT_FILE文件和INOUT_FILE文件重载。

关闭文件：在所有操作之后，必须取消内部文件名和外部文

件名的对应关系。即将文件关闭：

PROCEDURE CLOSE(FILE:**IN OUT** IN_FILE);

这个过程对OUT—FILE文件和INOUT—FILE文件重载。关闭文件不应与下述过程相混淆：

PROCEDURE DELETE (NAME:**IN** STRING) ;

这个过程是删除有关的外部文件。例如：

CLOSE (RESULTS) ;

指出目前时刻停止以RESULTS为名称使用文件"@0:ZO-ZO"。但是当需要时，可以重新打开这一文件。然而，

DELETE (@0:ZOZO") ;

则从实体上删除这一文件，此后不可能再对它进行访问。

下面两个函数分别用于指出一个文件是否是打开的，以及这个文件的外部名称是什么（对OUT_FILE文件和INOUT_FILE文件重载）：

FUNCTION IS—OPEN (FILE : **IN** IN_FILE) **RETURN** BOOLEAN;

FUNCTION NAME (FILE: **IN** IN_FILE) **RETURN** STRING;

其他的操作本质上都是对文件的访问（读或写）。被读文件具有一个当前读位置。这个位置是指下一次准备读的元素的号数。在打开文件时，它被置为1。处于顺序写的文件也有一个当前写位置，它是准备写的下一个元素的号数。在建立文件或打开文件时，这个位置被置为1。当前读（或写）位置都是FILE_INDEX（足标文件）预先确定的整型数。每一个文件都有实际长度和最后位置。实际长度是写入元素的数目，最后位置是写入最后一个元素的位置。文件操

作可能引发下列异常：

STATUS_ERROR：试图在未打开的文件上进行操作；

USE_ERROR：试图进行与文件特性不相容的操作（如：在IN_FILE文件上进行写操作）；

DEVICE_ERROR：硬件功能故障；

DATA_ERROR：获得的数据是不确定或不正确的；

END_ERROR：在文件结束后试图进行读操作。

对文件的访问，包括下述操作：

顺序读一个文件：

```
PROCEDURE READ(FILE:IN IN_FILE; ITEM :
                    OUT ELEMENT_TYPE) ;
        (对INOUT_FILE文件重载)
```

这个过程在FILE文件的当前读位置上读一个元素ITEM，并且把当前读位置顺序移动到下一个有定义的位置上。

顺序写一个文件：

```
PROCEDURE WRITE (FILE : IN OUT_FILE;
                  ITEM;   IN ELEMENT_TYPE);
    (对INOUT_FILE文件重载)
```

这个过程在FILE文件的当前写位置上写一个元素ITEM，并且把当前写位置顺序移到下一个位置，文件的当前长度加1。

下面两个函数分别提供当前读和当前写的位置：

```
FUNCTION NEXT_READ(FILE:IN IN_FILE)
                RETURN FILE_INDEX;
FUNCTION NEXT_WRITE (FILE:IN OUT_
                FILE) RETURN FILE_INDEX;
```

(对INOUT_FILE文件重载)

文件返绕：

```
PROCEDURE RESET_READ (FILE : IN IN_
                      FILE);
PROCEDURE RESET_WRITE (FILE:IN OUT_
                       FILE) ;
```

(对INOUT_FILE文件重载)

这两个过程把当前读或当前写位置重新设置为1。它们与FORTRAN程序中的REWIND相似，允许进行顺序文件的处理。

设置顺序读、顺序写的当前位置：

```
PROCEDURE SET-READ (FILE:IN IN_FILE;
                    TO:IN FILE_INDEX) ;
PROCEDURE SET-WRITE(FILE:IN OUT_FILE;
                    TO:IN FILE_INDEX) ;
```

(对INOUT_FILE文件重载)

这两个过程把当前读或当前写的位置 设置在由TO表示的数值上，例如：

```
SET—READ (FILE⇒FICHIER, TO⇒10);
READ (FICHIER, ITEM⇒DATA);
```

第一个语句将当前读位置定位在记录的第10号上，第二个语句是在记录的第10号上读数据。由于人们可以指出在那里读（或写）而不关心其顺序，因此这两个过程可以管理直接访问的文件。直接访问的位置是根据文件的原点确定的。这样确定访问位置的结构叫做相对结构，采用相对结构，人们可以建立所需要的访问系统，如：链表结构，顺序足标结构，

等等。

下面的函数分别提供文件的实际长度和文件的终止位置：

```
FUNCTION SIZE (FILE : IN IN_FILE) RETURN
                FILE_INDEX;
FUNCTION LAST (FILE:IN IN_FILE) RETURN
                FILE_INDEX;
```

（对OUT_FILE和INOUT_FILE文件重载）

这两个函数是不同的。只有在建立文件时将所有有效的元素连续写入文件。也就是说，在不留空白（即没有不确定的记录）的情况下（大多数是这种情况），它们才是相等的。

文件结束函数格式如下：

```
FUNCTION END_OF_FILE (FILE : IN IN_
                FILE) RETURN BOOLEAN;
```

（对INOUT_FILE文件重载）

如果当前读位置超过文件终止位置，即文件结束，这个函数为真（TRUE）。

截断文件：

```
PROCEDURE TRUNCATE(FILE:IN OUT_FILE;
                        TO:IN FILE_INDEX) ;
```

（对INOUT_FILE文件重载）

这个过程以TO规定的值建立文件的终止位置。TO之后的元素被取消，文件的实际长度重新计算。

下面是输入、输出程序包INPUT_OUTPUT的轮廓。

```
GENERIC
TYPE ELEMENT_TYPE IS LIMITED PRIVATE;
```

```
PACKAGE INPUT—OUTPUT IS
    TYPE IN--FILE IS LIMITED PRIVATE;
    TYPE OUT--FILE IS LIMITED PRIVATE;
    TYPE INOUT—FILE IS LIMITED PRIVATE;
    TYPE FILE--INDEX IS RANGE 0..dc;
        --文件操作过程及其重载由
PROCEDURE CREATE……;
        --到
PROCEDURE TRUNCATE……;
        --输入/输出操作及其重载由
PROCEDURE READ……;
        --到
FUNCTION END_OF_ FILE……;
        --异常
PRIVATE
        --私有部分
END INPUT_OUTPUT;
```

例5.1 已有一个实型文件，在此文件的最后添加一个数X。

解：

```
PROCEDURE AJOUT IS
    PACKAGE I—O—REALS IS NEW INPUT_
                OUTPUT (FLOAT) ;
    USE I_O_REALS;
    X: FLOAT;
    FICHIER : INOUT—FILE;
BEGIN
```

```
        OPEN (FICHIER, "XXXX"),
        SET - WRITE(FICHIER, LAST(FICHIER)
            + 1);
        --对X值的计算
        WRITE (FICHIER) X);
        CLOSE (FICHIER);
    END AJOUT;
```

例5.2 采用比顺序访问更简单的相关访问进行下述登记。

企业职工文件具有如下记录结构：

职工编号：（整型数）；

姓名：（十个字符的字串）；

住址：（三十个字符的字串）；

职务：（十个字符的字串）；

级别：（整型数）；

文件是按职工编号递增的顺序排列成的。实际上，职工编号在文件中表示记录的位置。另外有一个待整理的文件，它有相同的记录结构，但顺序杂乱无章。这个记录可以理解为由两部分组成：职工编号和有关该职工的数据。请将后面一个文件的内容整理登记到企业职工文件中。

解：

```
PROCEDURE MISAJOVR IS
    TYPE EMPLOYER IS
        RECORD
            NUMBER : INTEGER;
            NAME : STRING (1 . .10) ;
        ADDRESS : STRING (1 . .30) ;
        EMPLOY : STRING (1 . .10) ;
```

```
        DEGREE : INTEGER;
    END RECORD;
    PACKAGE IO IS NEW INPUT_OUTPUT
                        (EMPLOYER) ;
    USE IO;
    PERSFILE : INOUT_FILE;
    MOUVT : IN_FILE;
    EMP : EMPLOYER;
    N : FILE_INDEX;
BEGIN
    OPEN (PERSFILE, "XXXX" ) ,
    OPEN (MOUVT, "YYYY" ) ;
    WHILE NOT END_OF_FILE (MOUVT)
          LOOP READ (MOUVT, EMP) ;
    N : = FILE_INDEX (EMP.NUMBER) ;
    SET_WRITE (PERSFILE, TO⇒N) ;
    WRITE (PERSFILE, EMP) ;
END LOOP;
CLOSE (PERSFILE) ;
CLOSE (MOUVT) ;
END MISAJOUR;
```

5.3 正文输入输出程序包 TEXT_IO

这个程序包提供与外部文件进行输入-输出的操作。外部文件是以字符形式表示的，称为正文（TEXT）。其目的

在于为用户在标准外围设备上（如：键盘显示终端）提供可以阅读的输入与输出。

这个程序包可以调用输入-输出程序包 INPUT__OUTPUT，因此，它具有上节所叙述的各种功能。另外它还提供两个过程GET和PUT，这两个过程可以把某一类型的对象转化成字符串，并且把它与外部进行交换。这两个过程的参数涉及到两种类型的文件：IN__FILE和OUT__FILE。对于INOUT__FILE文件，不规定正文的输入、输出操作。PUT和GET除了参数ITEM用于确定要交换的对象之外，还有另一个参数用于确定所涉及的文件。如果这个参数不存在（有些重载允许这种情况），那么可以认为操作所涉及的是当前输入文件或当前输出文件。程序开始执行时，所涉及的是由系统确定的标准输入文件和标准输出文件（通常是键盘和屏幕，或者是读卡机和打印机）。

在5.1节中，已经注意到类型CHARACTER（字符型）规定了ADA语言所使用的ASCII字符共128个。其中95个是可打印的，其余的是控制ASCII字符。通过TEXT__IO读或写一个控制ASCII字符的作用取决于编译器。

下面我们将研究：

——标准文件的操作；

——输出格式操作；

——字符输入—输出；

——其他类型数据的输入-输出。

标准文件

下面的函数提供标准输入文件：

FUNCTION STANDARD__INPUT **RETURN** IN__

FILE；如果初始化中不存在这一函数，则标准输入文件系指当前文件。同样，下述函数将提供标准输出文件：

```
FUNCTION STANDARD_OUTPUT RETURN
OUT_FILE;
```

下面两个函数分别提供当前输入文件和当前输出文件。操作中涉及到这些文件时，将不提供文件参数：

```
FUNCTION CURRENT_INPUT RETURN IN_
        FILE;
FUNCTION CURRENT_OUTPUT RETURN OUT
        _FILE;
```

下面两个过程指定当前输入文件或当前输出文件参数。这些文件必须是打开的：

```
PROCEDURE SET_INPUT(FILE:IN IN_FILE);
PROCEDURE SET_OUTPUT (FILE : IN OUT_
        FILE) ;
```

输出格式

正文文件可以认为是由一系列的行组成的。行与行之间由行终止符分开（行终止符不算行内的字符）。行又由列组成，每个字符占一列，行和列都从1开始编号。行的宽度可以是固定的，也可以是不固定的。在写文件的不同时刻，行的宽度可以固定在不同的数值。但是在所有时刻，对于打开的正文文件是知道当前行与当前列的。由它们决定下一次GET或PUT的起始位置。

下面的函数将提供当前处于那一列：

```
FUNCTION COL (FILE IN IN_FILE) RETURN
        NATURAL;
```

FUNCTION COL (FILE : **IN** OUT_FILE)
RETURN NATURAL;

FUNCTION COL **RETURN** NATURAL; （缺省文件参数为当前输出文件）。

下面的过程把当前列号置为TO所规定的值：

PROCEDURE SET_COL (FILE : **IN** IN_FILE;
TO : **IN** NATURAL) ;

这个过程对OUT—FILE文件重载，并取当前输出文件为缺省文件参数。例如，如果

TAB : **ARRAY** (1 . .10) **OF** NATURAL；是一个定位的表格化数组，则

SET_COL (TAB(P));

将把标准文件定位在表格中的第P个位置上。

同样，下述函数将提供当前的行号：

FUNCTION LINE (FILE : **IN** IN—FILE) **RETURN**
NATURAL;

这个函数对OUT—FILE文件重载。并取当前输出文件为缺省文件参数。

PROCEDURE NEW-LINE(FILE:**IN** OUT_FILE;
SPACING : **IN** NATURAL : = 1);

这个过程把列号设置为1，行号加SPACING的值。SPACING等于1表示换行，等于2表示跳一行。如果行的宽度是固定的，则未用的列添加上空白。这个过程对于当前输出文件缺省文件重载，但是对于IN—FILE文件不重载（参考过程SKIP—LING）。

PROCEDURE SKIP—LINE (FILE:**IN** IN_FILE;

SPACING : **IN** NATURAL : = 1)；

这个过程适用于只读型文件IN—FILE。它把列号置于1，行号加SPACING的值，即跳（SPACING—1）行。这个过程对于当前输入文件（缺省文件）重载，但是对OUT—FILE文件不重载（参考过程NEW__LINE）。

函数END__OF__LINE(行结束)用于回答当前行上是否还存在可以读的字符。如果不再有可读的字符，则该函数值为真（TRUE）：

FUNCTION END__OF__LINE(FILE:**IN** IN—FILE)
RETURN BOOLEAN；

FUNCTION END__OF__LINE **RETURN** BOOLEAN；（指当前输入文件），例如：

IF END__OF__LINE **THEN** SKIP-LINE；**END** IF；

这条语句的意思是：在当前输入文件上读完一行后换行。

函数LINE__LENGTH（行长度）提供实际规定的宽度。函数的值为零则表示行的宽度是不固定的。

FUNCTION LINE__LENGTH (FILE : **IN** IN__FILE) **RETURN** INTEGER；

函数LINE—LENGTH对OUT—FILE文件重载，并取当前输出文件为缺省参数。

过程SET—LINE—LENGTH（设置行长度）把行的宽度设置为N所规定的值，N = 0指宽度是不固定的：

PROCEDURE SET__LINE—LENGTH (FILE : **IN** IN—FILE；N : **IN** INTEGER)；

这个过程对OUT__FILE文件重载，并取当前输出文件为缺省参数。例如：

SET—LINE—LENGTH（N⇒80），将行的宽度设置为80列。

字符的输入—输出

字符的输入、输出是由过程GET（取走字符）和过程PUT（放上字符）实现的：

```
PROCEDURE GET(FILE : IN IN—FILE; ITEM :
                                OUT CHARACTER) ;
```

过程GET把在指定文件的当前位置上读出的字符取走，并且把当前列号加1。除非下一个要读的字符恰恰是行终止符（这种情况下，定位于下一行的第一个字符）。过程GET对于当前输入文件（缺省文件）重载。

```
PROCEDURE PUT (FILE : IN OUT—FILE;
                        ITEM : IN CHARACTER);
```

过程PUT把所提供的字符写在指定文件的当前位置上，并且把当前列号加1。除非行的宽度是固定的，并且刚写入的恰好是本行的最后一个字符（在这种情况下，写一个行终止符后定位于下一行的第一个字符）。PUT过程对于当前输出文件缺省文件重载。

这两个过程对字符串型的ITEM也是重载的。如果字符串的长度为n，那么以字符串调用GET或PUT相当于以单个字符连续调用n次。

函数GET—STRING（取字符串）的作用是跳过开头的空白（如：空格、表格、行终止符等）连续执行GET（取字符），取走一串字符，直到下一个空白为止（这个空白不计入字符串内）；

```
FUNCTION GET—STRING (FILE:IN IN-FILE)
```

RETURN STRING;

对当前输入文件（缺省文件）重载。

例如：GET—STRING（ ）的字符串是GOOD MORNING。如图5.1：由开始位置起，跳过空白行和空白直到G开始取字符。在GOOD和MORNING之间的空白处停止。

FUNCTION GET—LINE (FILE : **IN** IN—FILE)
RETURN STRING;

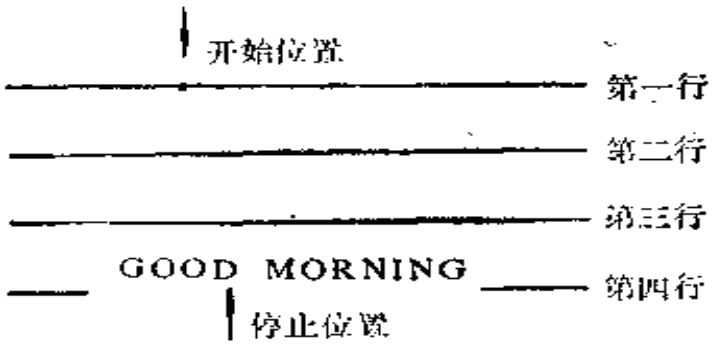


图 5.1 GET—STRING（ ）的字符串

这个函数的作用是取走后面的一串字符，直到遇到行终止符为止（行终止符不计其内），接着定位于行终止符之后。因此连续调用GET—LINE可以提供其后诸行的字符。这个函数对当前输入文件（缺省文件）重载。

PROCEDURE PUT—LINE(FILE:**IN** OUT—FILE;
ITEM : **IN** STRING);

这个过程的功能是在指定的文件上写入给定的字符串，然后写一个行终止符。这个过程对当前输出文件(缺省文件)重载。

例5.3 ZOZO和ZAZA是两个正文文件，要求在ZA—ZA上复制ZOZO文件，并保持行的结构不变。

解：**PROCEDURE** COPY **IS**

```
USE TEXT_IO;
ZOZO : IN_FILE;
ZAZA : OUT_FILE;
X : STRING;
BEGIN
    CREATE (ZAZA, "XXXX") ;
    OPEN (ZOZO, "ZZZZ") ;
    SET—INPUT (ZOZO) ;
    SET—OUTPUT (ZAZA) ;
    WHILE NOT END—OF—FILE (ZOZO) LOOP
        X : = GET—LINE ( ) ;
        PUT_LINE (X) ;
    END LOOP;
    CLOSE (ZOZO) ;
    CLOSE (ZAZA) ;
END COPY;
```

其他类型数据的输入-输出

其他类型数据的处理方法与处理字符的方法相同。即一个元素接一个元素地处理。在此，使用“元素”这个词是从词汇单位这个意义上讲的。

输出时，写一个服从该类型常数语法规则的字符串；输入时，取一个服从语法规则的尽可能长的字符串。这意味着在数据开始输入之后，遇到第一个空白时即停止输入。

另一方面，输入数据时，将跳过开头的所有空白（空格、表格、行终止符等）。这意味着一个数据不得跨越两行。如果所遇到的数据不符合语法规则，将引发异常 DATA_

ERROR。输出数据时，参数 WIDTH（宽度）规定写入区域的宽度。数字型数据紧靠右边写，枚举型数据紧靠左边写。如果规定的宽度不够，则取所需要的字符数。参数WIDTH的缺省值为零。因此，如果不规定这个参数，宽度则按需要取值。如果在一行上剩下足够的位置，那么由当前位置开始写入数据；在固定行的宽度的情况下，如果剩下的位置不多，则从下一行的开头写入数据。

下面的各个过程，都对缺省文件重载，对于过程GET来说，缺省文件是当前输入文件，对于过程 PUT 来说缺省文件是当前输出文件。

对于数值类型，在TEXT_IO 程序包内提供了三个类属程序包：INTEGER__IO,FLOAT—IO，和FIXED__IO，使用这些程序包时需要衍生（但TEXT__IO却不出现）。而使用TEXT—IO非类属部分，只要使用**USE** TEXT—IO就行了。

a) 整型

类属程序包INTEGER—IO包含如下过程：

```
PROCEDURE GET (FILE : IN IN-FILE; ITEM :
              OUT NUM) ;
PROCEDURE PUT (FILE : IN OUT-FILE;
              ITEM : IN NUM) ;
              WIDTH : IN INTEGER : = 0 ;
              BASE : IN INTEGER RANGE
              2 ..16 : = 10) ;
```

过程GET读入一个前面有时带符号的十进制数或以#…#为基的字面值，并且按照基进行转换。过程PUT写出ITEM

的数值（对于负数数值前面可带一负号）。ITEM的数值之间没有分位号，并且数值面前没有0（但是，数值零仍写成0）。如果数的基不是10，那么采用带符号#的习惯表示法。

例5.4 如果X：＝150；那么下面各个语句产生的数字串的形式是什么？

```
PUT (X);
PUT (-X, 10);
PUT (X, WIDTH⇒15, BASE⇒2);
```

解：150

□□□□□□□-150

□□□□2*10010110*

b）浮点型

类属程序包FLOAT_IO包括下面两个过程：

```
PROCEDURE GET (FILE:IN IN_FILE; ITEM:
                OUT NUM);
PROCEDURE PUT (FILE:IN OUT_FILE; ITE-
                M:IN NUM); WIDTH:IN INTE-
                GER:=0; MANTISSA:IN INT-
                EGER:=NUM'DIGITS; EXPON-
                ENT:IN INTEGER:=2);
```

对于GET过程，ITEM的基可以不为10（表示法为b#XX.XX#EYY）。然而对于过程PUT只能获得十进制数。对于定点型也是这样。这就是说，输出只能是十进制的。在PUT中MANTISSA指出所需要的有效位数，其缺省值取NUM'DIGITS有时有舍入（在声明中规定该类型的有效位数）EXPONENT是指数的位数。

c) 定点型

从现在起，我们介绍过程时，不再写出参数FILE。当然，带有这个参数的重载是存在的。对于GET这个参数是IN-FILE（只读文件），对于PUT是OUT-FILE（只写文件）。

类属程序包FIXED_IO包括：

```
PROCEDURE GET (ITEM : OUT NUM) ;
PROCEDURE PUT (ITEM : IN NUM, WIDTH :
              IN INTEGER : = 0 ; FRACT : IN
              INTEGER : = DEFAULT—DECIM-
              ALS) ;
```

FRACT是十进位小数点之后所希望的位数。如果小数点后的位数多于规定的位数，多余部分将被舍去。反之，如果少于规定的位数，则由零来补齐。如果不规定这个值，将由这个类型的属性推导出FRACT的值。

d) 布尔型

布尔型的输入、输出可以直接使用下述两个过程：

```
PROCEDURE GET (ITEM : OUT BOOLEAN);
PROCEDURE PUT (ITEM : IN BOOLEAN, WI-
              DTH : IN INTEGER : = 0 ; LOWER
              CASE : IN BOOLEAN : = FALSE);
```

过程GET读入一个字符串TRUE或者FALSE（大写或小写），并且给参数ITEM赋相应的逻辑值。过程PUT根据ITEM的值打印TRUE或FALSE。当LOWER—CASE⇒TRUE时，以小写字母打印。

e) 枚举型

在 PASCAL 语言中，用户可以定义一个枚举型变量 DAY（日期）。它可取值 Monday，Tuesday，……，但是却不能以字符串MONDAY的形式由键盘读取这个值。在这一点上，ADA远远超过了PASCAL，借助于类属程序包 ENUMERATION-IO 能完成这一操作，ENUMERATION—IO包括下面两个过程：

```
PROCEDURE GET (ITEM : OUT ENUM) ;
PROCEDURE PUT (ITEM:IN ENUM; WIDTH:IN
              INTEGER : = 0 ; LOWER-CASE :
              IN BOOLEAN : = FALSE);
```

过程GET读取一个标识符（枚举型常数的符号名称）或一个字符字面值（字符在引号之间）。对大写或小写字符不加区别。如果所读取的是该类型的一个常数，那么就把这值赋予参数ITEM，否则引发异常DATA—ERROR。过程PUT打印ITEM所代表的值的符号，这个值是枚举型ENUM中的一个常数。可以要求以小写字母打印。如果枚举型是一个字符集合（'A'，'B'，'C'……），那么这个字符被打印在引号之间，如：'A'。

例5.5 由键盘读一个日期，如果用户键入的是星期一至星期五当中的一天，在屏幕上显示WORKDAY，如果是星期六或星期日，显示HOLIDAY，键盘和屏幕可以认为是标准外围设备。

解：

```
PROCEDURE WHATDAY IS
    USE TEXT—IO;
    TYPE DAY IS (MONDAY, TUESDAY, W-
         EDNESDAY, THCRDAY, FRIDAY,
```

```
            SATDAY, SUNDAY);
        TODAY : DAY;
        PACKAGE I-O IS NEW ENUMERATION-IO
                (DAY);
        USE I-O;
    BEGIN
        PUT ( "WHAT DAY IS TODAY? " );
        GET (TODAY);
        NEW--LINE (SPACING=> 3 );
        PUT ( "* * * * *" );
        IF TODAY IN SATDAY..SUNDAY THEN
            PUT ( "HOLIDAY" );
        ELSE
            PUT ( "WORKDAY" );
        END IF;
            PUT ( "* * * * *" );
    END WHATDAY;
```

下面是以上程序执行的例子（黑体字是用户键入的字母）：

```
WHAT DAY IS TODAY? SATDAY
* * * * HOLIDAY * * * *
```

TEXT—IO程序包轮廓：

```
PACKAGE TEXT_IO IS
    PACKAGE CHARACTER--IO IS NEW INP-
            UT--OUTPUT (CHARACTER);
        TYPE IN_FILE IS NEW CHARACTER
```

```
            -IO.IN_FILE;
        TYPE OUT-FILE IS NEW CHARACTER
            -IO.OUT_FILE;
--字符及字符串输入输出过程
--整型数类属程序包:
GENERIC
    TYPE NUM IS RANGE < >;
    WITH FUNCTION IMAGE (X : NUM)
        RETURN STRING IS NUM'IMAGE;
    WITH FUNCTION VALUE (X : STRING)
        RETURN NUM IS NUM'VALUE;
PACKAGE INTEGER-IO IS
--过程
END INTEGER-IO;
--浮点数类属程序包
GENERIC
    TYPE NUM IS DIGITS ( ) ;
    :
    PACKAGE FLOAT-IO IS
    --过程
    END FLOAT-IO;
    --定点数类属程序包:
GENERIC
    TYPE NUM IS DELTA ( ) ;
    :
    PACKAGE FIXED-IO IS
```

```
        DELTA-IMAGE : CONSTANT STRING :
                =IMAGE WUM'DELTA-INT
                EGER (NUM'DELTA) ) ;
        DEFAULT-DECIMALS:CONSTANT INT-
          EGER:-DELTA-IMAGE' LENGTH-2;
    --过程
END FIXED—IO;
    --布尔量过程
    --枚举型变量类属程序包:
GENEIC
    TYPE NUM IS ( < > ) ;
    WITH FUNCTION IMAGE (X : ENUM)
        RETURN STRING IS ENUM'IMAGE;
WITH FUNCTION VALUE (X : STRING) RETU-
    RN ENUM IS ENUM'VALUE;
PACKAGE ENUMERATION_IO IS
    --过程
END ENUMERATION—IO;
    --输出格式控制过程;
    --标准文件操作过程;
    --异常: 从NAME_ERROR至END--ERROR
    --重新命名:
NAME—ERROR, EXCEPTION RENAMES CHAR-
        ACTER—IO.NAME—ERROR;
    :
LAYOUT—ERROR : EXCEPTION;
```

END TEXT_IO;

5.4 控制字符输入输出程序包 LOW—LEVEL—IO

这个程序包允许向外围设备发送控制字符串。它由一些对外围设备重载的过程组成。这些过程体可以借助于指令码用机器语言编写。

```
PACKAGE LOW—LEVEL—IO IS
    --外围设备类型说明和所采用的控制数
    --据以及如下过程：
PROCEDURE SEND—CONTROL (DEVICE：外设
                类型；DATA：IN OUT数据类型)；
PROCEDURE RECEIVE—CONTROL (DEVICE：
                外设类型；DATA：IN OUT数据类型)；
END LOW—LEVEL—IO；
```

这样既实现了对外围设备的控制，同时又保持了ADA语言的结构形式。

第六章　并行和任务

ADA提供了并行处理的可能性。这就是说，ADA允许把一个处理过程分解成几个假定可以并行执行的过程。执行并行过程的时候，在过程之间应该有同步条件，当几个过程共用一些数据的时候，还应该有互斥条件。

ADA提供了表示这些条件的工具。另外，关于并行执行的实现，ADA既没有做任何的假设，也没有规定任何前提条件。因此它的程序可以在完全没有并行可能性的机器上执行。例如：具有简单编译器的微处理机。

并行可以用多重处理器来实现，也可以用一个处理器的分时执行来实现。但是，无论哪一种情况，编译器必须保证不改变ADA所提出结构的语义。本章我们不准备介绍并行机构、并行过程及互斥等概念的细节。

在ADA语言中，并行的基本工具是任务（TASK）。所谓任务是一个处理单位，它可以与其他处理单位并行地执行。一个任务的生存期包括：任务的活化、任务的执行和任务的结束。在第一章，我们已经看到任务之间实现同步的最简单机构。尽管其简单，却是可用的。即一个处理单位只有当由它启动的任务都结束时才告终止。ADA语言中，还有另外一种更为精确的同步机构——汇合。这就是本章将要介绍的内容。

6.1 任务的活化、执行和终止

在ADA最初文本中，任务由一个专门的语句**INITIATE**来活化。没有活化的任务体是不能执行的。新文本ADA语言中，活化是自动进行的，即在一个模块或子程序中确定的所有任务，在模块的**BEGIN**开始执行时，所有的任务都被活化。

一个任务被活化后，它要么处于执行阶段，要么被挂起（等待汇合）。所有的任务都由两部分组成（任务式和任务体）：

```
TASK 任务名称 IS
--入口说明：
⋮
END;
TASK BODY 任务名称 IS
    --可执行语句：BEGIN ——局部说明：
⋮
END 任务名称;
```

任务式指定任务入口（**ENTRY**），任务体包含局部说明和可执行的语句。任务体中可执行语句的形式如同一个无限循环。循环中有可供选择的不同操作，而操作的选择是根据和外部任务的同步情况确定的。如下面的轮廓所示：

```
TASK BODY 名 IS
        BEGIN
            LOOP
```

```
            SELECT
              :
            OR……;
            OR……;
        END SELECT;
    END LOOP;
  END 名;
```

任务终止的条件可能是：

a）任务执行到自身的**END**语句；

b）在其选择中执行到**TERMINATE**语句；

c）某种外部原因强迫任务停止执行（如**ABORT**命令或者异常）。

6.2 汇 合 机 构

任务中的同步点叫做入口。在任务式中用下述格式确定一个入口名：

ENTRY 名（参数）；

在任务体中用下述语句接收入口调用

```
ACCEPT 名（参数）DO
    ：接受体
END;
```

一个入口类似于一个过程，在其他任务中将有一个调用语句。因此需要区分被调用任务和调用任务（调用其他任务的任务）。请注意：对于任务而言，调用是通过入口的限制名来进行的，因为没有用于任务的**USE**语句。

例如：

被调用的任务T

```
TASK T IS
    ENTRYE (X：类型)；
END；
TASK BODY T IS
    BEG IN
    ：①
    ACCEPT E(X：类型)DO
    ：②
    END；
    ：③
    END T；
```

调用任务U

```
TASK U IS
 ：
END；
TASK BODY U IS
    BEG IN
    ：④
    T.E (X) ；…
    调用
    ：⑤
    END U；
```

即在调用任务U中写出入口限制T.E(X)来调用任务T的入口E。当然也可以在U任务体的局部说明中（在BEGIN之前），用下述语句对入口E重新命名：

```
PROCEDURE TE (X：类型) RENAMES T.E；
```

现在我们讨论两个任务的执行过程。当开始执行程序时，两个任务自动被活化。程序段①和④被并行执行（是一步一步地执行，还是分段交替执行，ADA未作具体规定。但这不会改变逻辑关系）。因此将有两种情况：任务T首先到达ACCEPT，或者任务U首先到达调用处T.E (X)，约会的作用在于：先到达ACCEPT的任务等待调用任务到达调用处。反之先到达调用处的调用任务等待被调用任务到达ACCEPT。因此，如果T先到达ACCEPT，它被挂起，等待U调用。如果U先到达调用，那么U被挂起，等待T到达

ACCEPT。这就是任务的同步。一旦实现了同步，也就是说当两个任务都到达汇合点的时候，将进行汇合。汇合的内容包括：

1）任务U继续被挂起，直到汇合结束为止；

2）象子程序一样传递参数（在此为X）。在汇合的第二阶段传递输入（IN）参数，并在第三阶段结束时传递输出（OUT）参数；

3）执行接受体（ACCEPT）（程序段②）。执行接受体而挂起任务U这一事实，使汇合具有所谓转换段（critical Section）的特性：互斥。也就是说，程序段②可以使用两个任务共用的数据（包括被传递的参数），而不受任务U中修正这些数据的语句的影响。并能保证当有多个任务调用同一个入口项时，每次只接受一个入口项调用。请注意：如果转换段退化成参数的传递则没有DO……END；这时只要写成下面的形式就行了：

ACCEPT E（X：类型）；

当汇合结束的时候，每一个任务继续进行各自的计算，程序段③和⑤被并行执行。

例8.1 现有如下一个顺序执行的程序：

```
PROCEDURE ZOZO IS
    X：类型；
    BEGIN
    --与X无关的程序段；
    ：①
    ：—下面是与①无关的计算X的程序段；
    ：②
```

```
        X := ...;
        --下面是使用X的程序段:
        ⋮③
    END ZOZO;
```

用并行原理，改写这个程序。

解：

```
PROCEDURE ZOZO IS
    X：类型;
    TASK T IS
        ENTRY PRET;
    END;
    TASK BODY T IS
    BEGIN
    ⋮①
    ACCEPT PRET;
    ⋮③
    END T;
    TASK U IS
    END;
    TASK BODY U IS
        ⋮②
        T.PRET;
    END U;
    BEGIN  --现在ZOZO已成为空程序段
        NULL;
    END ZOZO;
```

这样写我们得到的好处是使得程序段①和②并行地执行。对

于单处理器的情况而言，执行的时间和没有改写的程序相同。但在多处理器的情况下，此程序将赢得执行时间上的节省。

6.3 接受队列和任务属性

应该注意到汇合的不对称性：调用任务引用被调用任务，被调用任务却不引用调用任务。因此，保留字**ACCEPT**可以解释为："无论从哪里来的调用，都从入口接受"。

事实上，可能有几个调用T.E的任务，如U，V和W。如果在T到达**ACCEPT**之前，已经有了这三个调用，它们将按照到达的顺序排成队列：(U) T.E/ (V) T.E/ (W) T.E。当任务到达**ACCEPT**的时候，执行队列中的第一个调用 (U)，并且从队列中将 (U) T.E删除。剩下的还有(V) T.E和 (W) T.E。由于任务体经常呈现为无限循环，当T再次经过**ACCEPT**的时候，将执行任务 (V) 的调用。在此期间，可以再将任务U新的调用或者其他任务的调用加入队列中：

(V) T.E/ (W) T.E/ (Y) T.E

其中 (V) T.E正处于待删除状态。

例6.2 在一个队列中，可能有多少个任务U对入口T.E的调用？

解：只能有一个。因为自任务U第一次调用入口T.E后，U即被挂起，它不可能再产生第二次调用。换言之，在所有队列的集合中，最多只能有一个由U产生的调用。

例6.3 如图有几个任务都"想"访问某一资源，则可

以用信号标（Semaphore）作为实现互斥的一种工具。当一个任务要访问该资源时，该任务就产生一个SEM.P信号，使得除这个任务外，其他的任务都不能访问这一资源。访问结束后，通过SEM.V解除信号标（V是荷兰语Vrejgeven的缩写，意思为释放）。请编写任务SEM。

解：
```
TASK SEM IS
    ENTRY P;
    ENTRY V;
END;
TASK BODY SEM IS
BEGIN
    LOOP
        ACCEPT P;
        ACCEPT V;
    END LOOP;
END SEM;
```

就是如此简单。P一旦被接受，任何对资源的访问即不再可能，直至信号标从另一个入口V接受调用。一旦调用任务给出SEM.V，资源即获释放，于是又准备接受下一个P的调用（借助于循环）。

其实，信号标在ADA之前就是一种管理工具，ADA中如法炮制，所以如此简单是毫不奇怪的。

现将任务的属性（或者任务类型的属性）列成表6.1：

表 6.1　　任 务 的 属 性

属　　性	意　　义
T'TERMINATED	任务是否结束(布尔型,已结束为TRUE)
T'PRIORITY	任务的优先权（整型）
T'FAILURE	可由外部引发的异常
T'STORAGE-SIZE	执行任务T占用的内存单元数
T'COUNT	等待调用任务T的入口数目（整型）

6.4 选 择 语 句 SELECT

在被调用任务中，语句**SELECT**实现条件判定选择。这个语句具有如下格式：

```
SELECT
    WHEN      条件 1 =>     选择 1;
OR WHEN       条件 2 =>     选择 2;
OR WHEN       条件 3 =>     选择 3;
    ⋮
    ELSE 语句 4;
END SELECT;
```

在上述格式中，可以没有选择 2，选择 3 等，也可以没有子句**ELSE**。只有一种选择时，可以不用 **WHEN**及判定条件。

选择可能有如下几种形式：

a) **ACCEPT……DO**

　　⋮

END;

其他语句;

b) DELAY时间;

其他语句;

c) TERMINATE;

在各种选择当中，形式c）最多只能有一个。而且不能既有形式b）又有形式c）。带有ACCEPT的形式a）至少要有一个。如果已有形式b）或者形式c），就不能再有ELSE。

选择分如下两个阶段进行：

1）首先判定条件WHEN。把那些条件已经实现的选择（或者无条件的选择）叫做打开的，其他的则称为关闭的。如果没有一个选择是打开的，但是有子句ELSE，那么将执行ELSE之后的有关语句，接着转到END SELECT之后的语句。如果既没有打开的选择又没有子句ELSE，将引发SELECT—ERROR（选择异常）。如果有打开的选择，那么这些选择（而且仅仅是这些选择）将在第二个阶段加以处理。

2）首先处理打开的具有ACCEPT的形式a）。如果在形式a）的选择中有一个是可以执行的，就是说它的相应的入口至少有一个任务正在等待调用它，那么就执行这个选择。接着执行ACCEPT…END;之后的其他语句。最后跳到END SELECT之后。

如果打开的具有ACCEPT的选择没有一个能发生汇合，并且有一个打开的具有延迟时间的形式b），那么将根据指定的时间（以秒表示）开始等待循环。

如果在等待循环期间，没有任何一个开放的形式a）的选择发生汇合，在规定的时间结束时，将执行DELAY之后

的其他语句。最后跳到**END SELECT**之后。

如果既没有形式b）又没有形式c），那么将无限期地等待，直到有一个**ACCEPT**能够被执行为止。

如果对某一个任务的调用只来自一个分程序或一个子程序，而这个分程序或子程序在这个任务执行之后即告结束，那么只有这种情况下才可以选择**TERMINATE**。也就是说，任务完成之后不再等待。

如果有子句 **ELSE**，并且任何一个开放的选择都没被调用（因为有**ELSE**，都是形式a）的选择），那么执行**ELSE**。

选择语句虽然相当复杂的，但是它提供了各种可能性，例如：

1）如果在十秒钟内请求调用，则在入口E接受调用，否则引发异常TIME—OUT（时间到）：

```
SELECT
    ACCEPT  E;
    ⋮
OR DELAY 10.0;
    RAISE TIME_OUT;
END SELECT;;
```

2）如果在入口E的调用队列是空的，那么在入口F接受调用，否则在入口E接受调用。

```
SELECT
    WHEN E'COUNT = 0 =>ACCEPT F;
    ⋮
    OR ACCEPT  E;
END SELECT;
```

在本例中，可以看到属性COUNT的应用。

例6.4 在上面（2）中，将无限期地等待调用。然而如果程序已经结束，或者脱离了任务的规定范围，将不再有调用。上面的写法会妨碍程序结束，试改写之。

解：只要在**END SELECT**之前加入

OR TERMINATE;

即可。

读者自然会问：当有几个打开的具有**ACCEPT**的选择都请求调用时，选择哪一个？

ADA没有规定这种类型的选择。这种选择是确定的，我们把这个问题留给编译器。因此，凡建立在特殊选择上的程序都是错误的。如果由于某种原因需要使一种选择优先于另一种选择，必须在**SELECT**的条件下加以说明。

下面是一个管理“邮政信箱”的例子，用于进一步说明**SELECT**语句的应用。在本例中有两种类型的任务，一种是使用“邮政信箱”的任务T，U，V，……；另一种是管理“邮政信箱”的任务BAL。任务T，U，V，…都有一个常数标记（记作ID），它可以向“邮政信箱”发送信息，也可以由“邮政信箱”接受信息，以实现信息交换。向“邮政信箱”发送的信息由三部分组成：收信任务标记（ID__DEST），寄信任务标记（ID__EM）和信件内容（TEXT）。为了发送信息，任务必须把以上内容写入一个变量MESSAGE。MESSAGE在此充当了“邮政信箱”。但是，只有“邮政信箱”是空的时候才能发送，布尔变量FULL用于表示这种状态。当某一任务准备从“信箱”中接受信息时，在汇合期间，它首先出示它的标记。管理任务BAL负责将这个标

记和收信任务标记（ID__DEST）进行比较，必须两者相等才提供信息。任务自“邮政信箱”取出信息后，“信箱”即变成空的。接着由信息中指定的任务来处理这些信息。下面是“邮政信箱”的管理程序：

```
PROCEDURE MAIL-BOX IS
    --全局说明：
    TYPE INDIC IS (T, U, V, …) ;
    TYPE MES IS
            RECORD
                ID__DEST : INDIC;
                ID__E M  : INDIC;
                TEXT     : STRING(1 ..50);
            END RECORD;
    --信箱管理任务
    TASK BAL IS
        ENTRY RE (ID : IN INDIC; MSG : OUT
            MES);
        ENTRY WR (MSG : IN MES) ;
    END BAL;
    TASK BODY BAL IS
        FULL : BOOLEAN : =FALSE;
        MESSAGE : MES;
        BEGIN
          LOOP
            SELECT
            WHEN FULL = >ACCEPT RE(ID:IN
```

```
                    INDIC; MSG : OUT MES) DO
                      MSG : = MESSAGE;
                      IF MESSAGE.ID_DEST = ID
                        THEN FULL : = FALSE;
                    END IF;
                END; --DO
            OR WHEN NOT FULL⇒
                ACCEPT WR (MSG : IN MES) DO
                    MESSAGE : = MSG;
                    FULL : = TRUE;
                END;
            OR WHEN NOT FULL⇒TERMINATE;
            END SELECT;
            END LOOP;
            END BAL;
```

使用“邮政信箱”的任务体中可写：

```
    TASK BODY T IS
        ID : CONSTANT INDIC : = T;
        MSG : MES;
        I_WANT_TO_RECEIVE : BOOLEAN;
        I_WANT_TO_SEND : BOOLEAN;
    BEGIN
        ⋮
        ⋮
    MSG.ID_DEST : = U; --给一个与T不同的初始值
    WHILE I_WANT_TO_RECEIVE AND THEN
```

```
        MSG.ID__DEST/=ID
        LOOP
        BAL.RE (ID, MSG);
    END LOOP;--只要没有找到寄给T的信息，一直进行
                --这个循环。处理接受到的信息。
    IF I__WANT__TO__SENO THEN
        BAL.WR (MSG);
    END IF;
        ⋮
    END T;
```

由于只有一个缓冲器存贮信息，因此只有当每一个任务不超过缓冲交换信息所用的时间的$\frac{1}{N}$时，这个系统才可能供N个任务使用。但是，在这种情况下，如果有寄往任务T的信息，而T没有及时接受，整个系统将可能被堵塞。

例6.5 循环缓冲器的管理

打印程序以不规则的节拍发送打印行。相反，打印机却以规则的节拍打印。因此即使发送打印行的平均速度不超过打印速度，仍可能在不发送打印行期间出现打印高峰。为此可以使用将接受的打印行积累起来的循环缓冲加以调整。循环缓冲器的管理是通过两个下标实现的（见图6.1）FIRST—FREE（第一个空闲位置）和NEXT—GET（下一个取走打印行的位置）。由于管理任务使用了模除（**MOD**）缓冲器长度，所以缓冲器是循环的。请概述：全局变量的说明，发行打印行任务和打印任务，并且完整地编写循环缓冲器的管理任务。

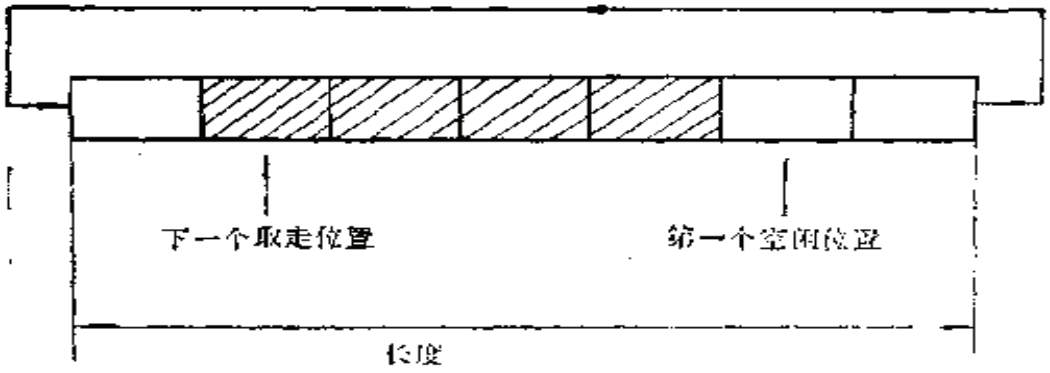


图6.1 循环缓冲器的管理

解：--总体说明：

```
TYPE TLINE IS STRING (1 …132) ;
--在产生打印行的任务中：
LOOP
 ⋮
LINE : = …;
SPOOLING.PRINT (LINE); --如果入口PRINT未被接受则挂起等待。
 ⋮
END LOOP;
--在执行打印的任务中：
LOOP
 ⋮
SPOOLING.LIT (LINE) ; --如果缓冲器中没有要打印的行，入口不被接受，因而挂起等待。
PUT (LINE)
 ⋮
END LOOP;
```

```
TASK SPOOLING IS
    ENTRY LIT(LINE : OUT TLINE);
    ENTRY PRINT(LINE : IN TLINE);
END;
TASK BODY SPOOLING IS
    SIZE : CONSTANT  INTEGER : = 2000; --此值由系统决定
    BUFFER : ARRAY (1 ..SIZE) OF TLINE;
    FIRST_FREE,NEXT_GET : INTEGER RANGE
            1 ..SIZE : = 1;
    NUMBER : INTEGER RANGE 0 ..SIZE : = 0; --缓冲器中的行数
    BEGIN
        LOOP
            SELECT
            WHEN NUMBER<SIZE = >--若缓冲器不满，接受新行。
    ACCEPT PRINT(LINE : IN TLINE)DO
    BUFFER(FIRST_FREE) : = LINE;
    END; ——DO
    FIRST_FREE: = FIRST_FREE MOD SIZE + 1;
    NUMBER : = NUMBER + 1;
    OR
    WHEN NUMBER> 0 = >--给出一个打印行，直至缓冲器空
    ACCEPT LIT(LINE : OUT TLINE) DO
```

```
LINE : = BUFFER(NEXT_GET);
END; --DO
NEXT_GET : = NEXT_GET MOD SIZE + 1;
NUMBER : = NUMBER — 1;
OR
    TERMINATE;
    END SELECT;
END LOOP;
END SPOOLING;
```

通过上面的介绍，我们已经看到选择语句**SELECT**在被调用任务中的功能是非常强的。用它可以很容易地表达并行算法，我们建议读者试用ADA写出经典算法，如生产者——消费者，读者——编者，银行家算法。这些算法的基本问题在于避免内部堵塞（即所有任务都挂起互相等待）。下面我们进一步介绍调用任务以及调用过程中的**SELECT**语句。

在调用任务中，**SELECT**语句有两种形式：

1）条件调用

条件调用的格式如下：

```
SELECT
    调用一个入口;
    程序段①;
ELSE
    程序段②;
END SELECT;
```

条件调用的功能是：力求调用指定的入口（属于其他的任务）。如果汇合能立即发生，那么将实现汇合，尔后执行程

序段①，最后跳到END SELECT之后。如果汇合不能立即发生，与简单调用相反，将不再等待汇合，而执行程序段②再通向END SELECT之后。例如：

```
LOOP
SELECT
    T.E (X);
ELSE
    --执行正在等待的某一操作
    :
    END SELECT;
END LOOP; --再次调用入口T.E (X)
```

（2）延迟调用

延迟调用的格式如下：

```
SELECT
    调用一个入口;
    程序段①;
OR DELAY 时间;
    程序段②;
END SELECT;
```

延迟调用的作用是：试图在规定的时间内调用入口。如果在规定的时间内，能发生汇合，那么将实现汇合，接着执行程序段①。反之如果在规定的时间内，汇合不能发生，则执行程序段②。在执行程序段①或②之后，都跳到后面的END SELECT。例如：

```
SELECT
    T.E (X);
```

```
OR DELAY 10.0;
    --在十秒钟内不能接受调用，执行下述操作：
    ⋮
END SELECT;
```

例6.6

在例6.5中的循环缓冲器管理程序中，为了在操作台上打印打印机“超时”的信息，请修改发送打印行任务。在此假设打印机每分钟打印1200行（即20行/秒），只要等待的时间大于$\frac{1}{20}$秒，就显示打印机“超时”。

解：

```
    ⋮
LINE := …
SELECT
    SPOOLING.PRINT (LINE);
OR
    DELAY 0.05; PUT (CONSOLE, "PRINTER OVERLOAD");
END SELECT;
    ⋮
```

在SELECT语句中已多次使用过语句**DELAY**，这个语句的格式为：

DELAY 表达式;

它的作用是把一个任务挂起，挂起的持续时间等于表达式的值（以秒为单位）。DURATION是预定义的一个持续时间，它允许的值（为浮点型）至少为86400秒（1天等于86400秒）。DURATION与预定义的程序包CALENDAR

（日历）有关。显示CALENDAR包括一个提供时间的CLOCK（时钟）。下面是CAENDAR程序包：

```
PACKAGE CALENDAR IS
    TYPE TIME IS
        RECORD
            YEAR: INTEGER RANGE 1986..2099;
            MONTH: INTEGER RANGE 1 . . 12;
            DAY: INTEGER RANGE 1 . . 31;
            SECOND: DURATION;
    END RECORD;
FUNCTION CLOCK RETURN TIME;
    --下述函数对DURATION和TIME重载
FUNCTION "+" (A : TIME; B : DURATION)
            RETURN TIME;
FUNCTION "+" (A : DURATION; B : TIME)
            RETURN TIME;
FUNCTION "-" (A : TIME; B : DURATION)
            RETURN TIME;
FUNCTION "-" (A : TIME; B : TIME) RETU-
            RN DURATION;
END CALENDAR;
```

6.5 任务类型和入口簇

在这一节里，我们将把任务中入口的定义推广到任务类型和入口簇。

在例6.3中我们曾定义过一个任务SEM（信号标），用来保护对某一资源(例如一个参量)的访问。如果我们有三个变量需要保护，如A，B和C，那么必须写三个信号标SEM-A，SEMB和SEMC。在调用任务中将写成：

```
SEMA.P --访问A
SEMA.V
SEMB.P --访问B
SEMB.V
SEMC.P --访问C
SEMC.V
```

而且为了定义三个信号标（三个任务），差不多是相同的内容，我们还得要重复三次。通过定义任务类型可以避免这种重复：

```
TASK TYPE SEM IS
    ENTRY P;
    ENTRY V;
END;
TASK BODY SEM IS
    BEGIN
    LOOP
        ACCEPT P;
        ACCEPT V;
    END LOOP;
END SEM;
```

这样做，我们就定义了一个任务模式（有点像任务的类属），称为任务类型。为了建立服从这个模式的任务SEM-

A，只要写成下面这种形式就行了：

SEMA：SEM；

因此为了保护三个变量，按照上面的办法建立任务型SEM，接着做如下说明：

SEMA，SEMB，SEMC：SEM；

人们也可以用任务类型建立指针型数据，其格式如下：

TYPE PTSEM **IS ACCESS** SEM；

P：PTSEM；

在此，声明属于PTSEM类型的所有变量都是能够访问类型为SEM的任务指针。P则是其中一个准备访问SEM的指针。在分配存贮单元的时候，

P：=**NEW** SEM；

则能产生并活化一个SEM类型的任务，这个任务可以用分号（；）执行操作。例如，调用指针P所指向的任务可以写作：

P.P；

请注意，在这语句中两个P的作用是不相同的。（前一个P代表指针，后一个P代表SEM型任务中的入口项）。任务本身是由P.**ALL**表示的。例如，任务是否终止的属性，可以表达为：

P.ALL'TERMINATED

注意任务既没有赋值操作又没有测等操作。因此任务的作用有点像**LIMITED PRIVATE**的对象。把一个任务传送给另一个任务唯一的方法是使用指针。例如：

SEMA：=SEMB；

是非法的，但是

P，Q：SEM；

P：=**NEW** SEM；

Q：=**NEW** SEM；

P：=Q；

则是合法的。

下面讨论入口簇。定义入口簇的方式和以足标定义一个数组类似。如果R是离散子界，那么，

ENTRY E（R）（X：类型；Y：类型）；

E(R) 就是一个入口簇。只要给出足标R的值，就可以调用簇中的一个入口。例如：

T.E（RR）（X⇒A；Y⇒B）；

T为被调用任务，RR为足标的值。

同样，对于如下定义的“操作台——打印机”入口簇：

ENTRY CONSOL_PRINT（1..5）（C：**IN** CHARACTER）；如果在电传打字机任务中，用上述方式定义了五个入口，现在想在第四台操作台上打印一个字符X，在调用任务中采用下述格式即可：

TELETYPE.CONSOL_PRINT（4）（'X'）；

6.6 优先权、共用变量和停车

1）优先权

在任务式中采用：

PRAGMA PRIORITY（静态整型表达式）；

可以对一种任务赋予优先权。整数越大，任务越紧急，优先权越高。当可以使用的处理器少于需要执行的任务数目的时

候，需要对这些任务加以选择。这时优先权给编译器一种指令，使得优先权高的任务先执行。当任务的优先权相等时，ADA语言没有规定执行的顺序。

2）共用变量

任务之间的正常联络是在汇合期间通过交换参数实现的。如果两个任务使用同一个全局变量，那么编者必须保证这两个任务不得同时修正这个变量（由此可以看到例6.3中信号标的重要性）。但是，即使对于一次读取也存在着问题。为了保证我们所访问到的变量由其他任务修正后的当前值，应该在访问这个变量之前先调用过程库里适当的类属过程：

```
GENERIC
    TYPE SHARED IS LIMITED PRIVATE;
    PROCEDURE SHARED_VARIABLE_UPDA-
                    TE (X: IN OUT SHARED);
```

例如：

```
X: FLOAT; --共用变量
PROCEDURE MAJ IS NEW SHARED_VARIABLE
                UPDATE (FLOAT);
    ⋮
MAJ (X); --清理X的值
Z := X; --访问X
    ⋮
```

3）停车　ABORT

语句

```
ABORT T, U, V,
```

将强迫任务T、U和V非正常地终止执行。**ABORT**语句只有在绝境中才被使用。但是用下面的语句实现停车更为可取：

RAISE T'FAILURE;

这个语句使任务T保留一个机会，用以处理异常FAILURE从而采取一种补救措施。

在文章中，我们看到ADA所提供的并行控制机构功能是非常强的，水平是高的，使用也是十分方便的。正因如此，ADA能成功地用于实时处理和系统程序。

第七章　异常和错误处理

异常是ADA语言用来处理错误或一些特殊情况的工具。它给ADA以坚实性，允许一个程序带着错误运行。在ADA中，一个异常像所有的对象一样有一个名（标识符），它遵循惯用的可见性法则。语言有一系列预定义的异常，这些异常在整个程序中可见。

全部异常在附录Ⅰ表中列出。现仅举几例：

CONSTRAINT—ERROR：违反类型约束；

NUMERIC—ERROR：被零除或计算中溢出。

以上为通用故障。当使用输入/输出时，还可能有：

END_ERROR：试图超越文件的终结位置。

与所有预定义的对象一样，用户可以对这些异常再定义，还可以按用户希望的名称定义所需要的异常。

异常的定义格式为：

名表：**EXCEPTION**；

例如：

SINGULAR_MATRIX：**EXCEPTION**；

定义的异常表示奇异矩阵。

应该分清异常的两个概念：异常的引发和异常的处理。下面我们逐一加以说明。

7.1 RAISE 语 句

故障的引发可以分为自动引发和显式引发。

预定义的异常可以根据环境自动引发。例如，当数组的下标超过预定值时，CONSTRAINT—ERROR就自动引发。预定义异常也能采用显式引发，但是用户定义的异常必须是显式引发。

显式引发是通过RAISE语句来实现的。其格式为：

RAISE异常名；

例如：

```
IF DETERMINANT = 0.0
    THEN RAISE SINGULAR_MATRIX;
```

或 **RAISE** NUMERIC_ERROR; --预定义异常。

RAISE语句的简化形式为**RAISE**。它仅在异常处理的程序段中使用，其作用是再次引发同一异常。显然，异常的引发只能出现在它的可见区域之内。

7.2 异常的处理

异常引发定义在产生它的模块上。**BLOC**这个词在本章指的是分程序（**DECL ARE BEGIN END**），子程序体，程序包或任务。

异常的处理包括两项内容：

1.停止执行引发异常的模块，而由异常处理程序代替该模块继续执行。对一个分程序**BEGIN…END**来说，将跳到

本模块的结束语句。对一个子程序来说，将返回到调用这个子程序的下一个语句；

2.执行由一定数量的语句组成的异常处理程序。处理程序以下面形式出现：

WHEN异常名⇒语句组；（与**CASE**的语法相同）。

例如：

```
WHEN NUMERIC_ERROR|SING ULAR_MATRIX⇒PUT ("ERROR IN THIS PLACE")
```

可见，即便是对于预定义的异常，用户仍可以自行提出一个处理程序。否则，对所有预定义的异常，系统有它自己的处理方法。

异常名称的位置上也可象**CASE**语句那样写入键词**"OTHERS"**，用以说明没有给出异常处理程序的那些异常如何处理。

由**WHEN**引入的异常处理程序冠以键词**EXCEPTION**，并处于模块的末尾。

例如：

```
PROCEDURE C IS
    X, Y : EXCEPTION;
    BEGIN
        《A》……
        |
        |
        |……;
        EXCEPTION
        WHEN X⇒
```

```
        PUT ("ACCIDENTX");
WHEN Y⇒
    PUT ("ACCIDENT Y");
WHEN OTHERS⇒
    PUT ("FOR SAVE");
END C;
```

再看异常引发之后的情况。先从上面的简单例子谈起。如果“事故”X发生在语句《A》执行的过程中，那么这一过程就停止执行（跳过所有I指令）而执行X的处理程序。这一处理程序已经在该模块中给出，即打印“ACCIDENT_X”，然后回到调用C的下一个语句。如果没有事故，过程C本应是全部执行的，由于有了异常而代之以打印出错信息（在正式程序中，并不是打印这样一个信息而是给出试图补救事故所造成的错误结果的措施）。

7.3 异常的传播

如果在发生的模块中没有异常处理程序，将会产生所谓的“异常的传播”。

异常传播是这样进行的：异常在一个模块中被引发之后，返回直接包围这个模块的模块。如果产生的模块是子程序，则返回调用此子程序的模块。如果在返回的模块中有异常处理程序，则执行处理程序，然后结束这个模块。如果仍然没有处理程序，则返回直接包围这后一模块的模块。

由此得到的结果是：随调用情况的不同，所执行的处理程序也可能是不相同的。例如：

```
PROCEDURE A IS
X : EXCEPTION;
PROCEDURE B IS
    BEGIN--B
    RAISE X
    END B;
PROCEDURE C IS
    BEGIN--C
    …B; --调用B
EXCEPTION
    WHEN X=>
        处理程序 1 ;
    END C;
BEGIN--A
    C; --调用点 1
    B; --调用点 2
EXCEPTION
    WHEN X=>
        处理程序 2 ;
    END A;
```

B引发异常X，但没有提供处理程序。C调用B并提供处理程序（1）。A调用C也直接调用B。A包括另一个处理程序（2）。

主程序A调用C的时候（即调用点1），C又调用B，异常X被引发。B中处于引发X的语句之后的部分被放弃，故返回C。C中处于调用B后面的部分被放弃，而去执行处理程

序1然后回到主程序A中调用点1后面的语句。

主程序A调用B时（即调用点2），在B内异常X又被引发。B中处于引发X语句之后的部分被放弃，故返回A。A中处于调用点2后面的部分被放弃，而去执行处理程序(2)，然后即告结束。如果A不是主程序，那么还要回到调用A的语句之后。

以上规则还可能有如下的变化形式：

（1）如果异常产生在对模块说明部分的加工期间，则立即返回外层模块或调用该子程序的模块；

（2）如果异常产生在没有处理程序的程序包体的初始化语句执行过程中，那么对程序包的加工即被放弃。同时异常传播到把该程序包作为子单位来说明的模块。如果这个模块是一个库内程序包，整个程序将停止执行。

（3）如果异常产生在没有处理程序的任务体中，则任务结束。没有异常传播发生。

7.4 任务中的异常

当任务调用另一个任务的一个入口时，如果被调任务已经结束，那么在调用任务中将引发异常“TASKING_ERROR”。

在下列情况下，汇合将失败：

（1）异常在接受过程中被引发，且没有进行局部处理时，汇合失败。异常同时传播到包含**ACCEPT**的模块和调用任务中。

（2）包含**ACCEPT**的任务非正常终止（例如由于**ABO**

RT），汇合失败。异常“TASKING_ERROR”在调用任务中被引发。

但是，调用任务的非正常终止并不影响被调任务。此时，如果汇合尚未开始，则取消调用。如果汇合已经开始，汇合继续进行，被调任务也继续执行。

所有的任务U都可以通过“**RAISE** T′ FAIL URE引发另一个任务T中的异常T′FAILURE。接受这一异常的任务则在它当时所处的那一点上立即接受。如果此任务正在**DELAY**语句中挂起，则取消延迟。如果此任务正在调用另一任务而汇合尚未开始，则取消调用。如果汇合已经开始，则汇合继续进行，且在任何情况下第三个任务（即被调任务）不会受到影响。如果接受任务T正在等待**SELECT**或**ACCEPT**，异常则被处理。如果任务T是在一个ACCEPT内部（在**ACCEPT**的**DO**后面的语句列中），并且在T程序体内没有“T′FAILURE”的处理程序，那么汇合中止，异常“TASKING_ERROR”在调用任务中引发。

“T′FAILURE”的处理程序只能在任务T或任务类型T当中。

例7.1 对于例4.6中的栈管理程序包，插入两种异常：满栈时调用“IN_PILE”和空栈时调用“OUT_PILE”。

解：在程序包体的局部说明中（变量POINTER说明之后）加入：

FULL, EMPTY : **EXCEPTION**;

然后，在过程IN_PILE语句段的首尾分别加入：

IF PILE_FULL **THEN RAISE** FULL; **END IF**;

和异常处理程序。

```
EXCEPTION
    WHEN FULL⇒PUT ( "PILE FULL" ) ;
```

在函数OUT_PILE语句段的首尾分别加入：

```
IF PILE_EMPTY THEN RAISE EMPTY; END
        IF;
```

和异常处理程序：

```
EXCEPTION WHEN EMPTY⇒PUT ( "PILEEM-
        PTY" ) ;
```

异常处理程序也可以加在调用程序中。

例7.2 对于例4.4中矩阵相乘的程序，如果两个相乘矩阵的维长不协调就引发"LENGTH_ERROR"，给出一个异常的处理程序，并增加必要的说明。

解：需要在**BEGIN**之前加进说明：

```
LENGTH_ERROR: EXCEPTION;
```

和在结束前加进异常处理程序：

```
EXCEPTION
    WHEN LENGTH_ERROR⇒
        PUT("NO CONFORM DIMENSIONS");
        NEW_LINE;
        PUT ( "MATRIX SET TO ZERO" ) ;
        C : = (A'RANGE⇒ (B'RANGE ( 2 ) ⇒
                0.0 ) ) ;
```

7.5 检测的免除

在各种操作中，编译器都插入一些检测，用来监视是否

有异常发生。因此，在任何环境下，预定义的异常都能自动引发。

例如使用下标变量时，这个下标与一个变量相联系。于是有一个检测用来监视这个下标是否超过了它的最大允许值。如果超过了，则引发CONSTRAINT_ERROR。

预先设计的这种检测项目非常多。因此它们在程序中占用了大量的时间。ADA提供一种参注（**PRAGMA**）能够请求编译器免除某种检测。这些参注的格式如下：

PRAGMA SUPPRESS检测名）；

或者只对某个变量免检：

PRAGMA SUPPRESS检测名，**ON**⇒变量名）；

对于每一异常，可能的检测包括：

（1） 对CONSTRAINT_ERROR：

ACCESS_CHECK

证明没有试图访问零指针所指的对象。

DISCRIMINANT_CHECK

证明根据判别式的值，记录（**RECORD**）的某个分量是存在的。

INDEX_CHECK

下标检测。

LENGTH_CHECK

下标数目检测。

RANGE_CHECK

子界检测。

（2） 对NUMERIC_ERROR：

DIVISION_CHECK

检测是否被零除（/，REM或MOD）。

OVERFLOW_CHECK

检测是否超出容量。

（3） 对STORAGE_ERROR（在存贮分配过程中超出了提供给指针所指向的那一类型元素整体的存贮单元数）。

STORAGE_CHECK

例如：

PRAGMA SUPPRESS （INDEX_CHECK）；--免除所有的下标检测。

PRAGMA SUPPRESS （INDEX_CHECK，**ON**⇒TAB）；

--只取消TAB表中的下标检测。

注意，检测免除只应用于对已经完全调整好的程序重新编译的时候。实际上，检测免除并不能阻止错误发生，而只是阻止由ADA提供的异常机构去控制程序的损坏。所以，在错误发生时，如果取消了检测，后果会是不可预料的。

第八章　参注和表示指明

本章介绍ADA语言所提供的指导编译器工作的两种工具：参注和表示指明。

参注是对编译器提出的要求，它与汇编语言中指令的作用相似。语言中有一些预定义的参注。一个特定的编译器还可以引入其他的参注。

出于对编译器的复杂程度和价格的考虑，一个ADA编译器也可以不设置预定义的参注。

表示指明对编译器来说是更严格的约束。它对一些数据或类型的表达方式提出要求，有时精确到位。在某些应用中，表示指明能够最大限度地发挥所用机器的特点。

8.1　参　注（PRAGMA）

ADA的基本参注可以列举如下：

PRAGMA CONTROLLED（存取类型名）；

要求存取类型的对象所占有的存贮单元只有在离开存取类型可见域时才收回。

PRAGMA INCLUDE（“文件名”）；

在参注出现的位置上，将指定文件的正文插进待编译的正文。

PRAGMA INLINE（子程序名）；

正文中每次调用指定子程序的地方应该将子程序展开。

PRAGMA INTERFACE（语言，子程序）；

表明子程序是用指定语言写的，因而该遵循这种语言的规范。

例如：**PRAGMA** INTERFACE（FORTRAN，
ATAN）；

PRAGMA LIST（ON）；

表明用户要打印程序清单。

PRAGMA LIST（OFF）；

中止打印清单（用ON恢复打印）。

PRAGMA OPTIMIZE（TIME或SPACE）；

要求编译器在时间方面或存贮空间方面进行优化（对于在说明部分含此参注的那个模块）。

PRAGMA PRIORITY（整数n）；

给含此参注的任务以优先权n。

PRAGMA SUPPRESS（检测名，ON⇒标识符）；

取消对指定对象自动生效的检测。

下面的参注与表示指明出现在同样的位置上：

PRAGMA PACK（**RECORD**型或**ARRAY**型）；

要求指定类型的表示尽可能地紧缩。

下面的参注只能位于库程序单位之首。它们的作用与表示指明有关：

PRAGMA MEMORY_SIZE（整数n）；

把库程序单位所要求的存贮单元数目固定为n。

PRAGMA STORAGE_UNIT（整数n）；

固定一个存贮单元为n位。

例如：**PRAGMA** STORAGE_UNIT（8）；

——一个存贮单元为一个字节（8位）

PRAGMA SYSTEM（名）；

建立目标机器名。此名应为SYSTEM程序包中所规定的SYSTEM_NAME类型的一个符号。

8.2 表示指明 (SPECIFICATION OF REPR ESENTATION)

表示指明可以用来指明类型的长度、记录的结构、枚举型的取值范围或某一对象的地址。它可能出现在一个分程序说明部分的末尾，并作用于在这个说明部分说明过的现象。也可能出现在任务式或程序包式中。尤其可能出现在程序包式的私有部分。这时，表示指明作用于在程序包的可见部分或私有部分说明过的对象，或作用于任务的入口。

没有表示指明时，表示方式由编译器选择。如果一个类型有几个指明，那么这些指明应该是互相补充的。比如一个指明代码，一个指明长度。

一种类型的表示方式是不能改变的。但可以定义一个派生型并给它一个不同于其父型的表示方式。

8.2.1 长度指明

这一指明的格式为：

FOR 属性 **USE** 表达式；

例如：

FOR INTEGER'SIZE **USE** 16;

其更符合口语的表达形式为

OCTET : **CONSTANT** : = 8 ;

FOR INTEGER'SIZE **USE** 2 *OCTET;

长度指明以位表示，可用于除任务型以外的所有类型，并且没有非静态约束。

用户还可以作如下指明（其中存贮空间的单位为存贮单元：STORAGE_UNIT）：

FOR T'STORAGE_SIZE **USE**…；--为存取型T指向的对象整体所保留的存贮单元数目。

FOR T'STORAGE_SIZE USE…；--任务或任务类型T的规模。

FOR RF'ACTUAL_DELTA **USE**数值；--定点型RF的实际步长（其中“数值”应为实型数）。例如：

TYPE REAL **IS NEW** FLOAT **DELTA** 0.001
RANGE-1000..1000;

FOR REAL'ACTUAL_DELTA **USE**-1.0/2 * *12;

如果没有以上指明，系统很可能取实际步长为1.0/2 * *10。

有些实数型的属性与内部表示法有关而与标准数无关。使用它们可以开发出一些新特性，这些特性比我们说明过的类型特性更强。但这可能有损于程序的可移植性。

例8.1 指出以下说明中的毛病：

TYPE REAL **IS DELTA** 0.01 **RANGE** - 100.0..
100.0;

FOR REAL'SIZE **USE** 12;

解：类型REAL有大约20000个标准数，而12位只能对4096

个不同值编码。因此至少应该有：

FOR REAL'SIZE **USE** 15;

8.2.2 枚举型编码指明

枚举型的每一个枚举符号都要和一个整数相对应（称为内码）。这些整数的顺序应该和枚举顺序一致，但可以是不连续的。

编码指明的格式如下：

FOR 枚举型 **USE** 结构常值;

结构常值可以用习惯的表示法列成整数表。例如，对于枚举型COLOR的内码可以指明如下：

```
TYPE COLOR IS (BLACK, BLUE, RED);
FOR COLOR USE (BLACK⇒ 1, BLUE⇒ 2,
               RED⇒ 6);
```

注意：尽管这些整数是不连续的，SUCC和PRED等函数仍有定义。系统在寻找SUCC（X）所表示的对象时，不一定每次加1，而是自行加入必要的级差。内码不连续的唯一缺点是计算较长（比如以枚举型为下标时）。

这一性质可以很容易地用来管理汇编语言。为此只要定义一个枚举型，枚举符号是一些便于记忆的指令符号，表示它们的内码由相应的操作码构成。

例如，下面是定义6502微处理机机器语言的表(摘要)。操作码为十六进制码。

LAD	STA	CMP	ADC	SBC	BIT	JMP	BEQ
AD	8D	CD	6D	ED	2C	4C	FO

BNE	CLC	SEC	JSR	RTS	BCC	BCS
DO	18	38	20	60	90	BO

可以将上表写为：

```
TYPE INSTRUC IS (CLC, JSR, BIT, SEC,
      JMP, RTS, ADC, STA, BCC, LDA,
      BCS, CMP, BNE, SBC, BEQ) ;
FOR INSTRUC USE(CLC⇒16*18*, JSR⇒16*20*,
    …, BEQ⇒16*FO*) ;
```

如果想使指令 I的代码为K，应写：

```
I : INSTRUC;
K : INTEGER;
FUNCTION CODOP IS NEW UNCHECKED_
         CONVERSION (INSTRUC, INTEG-
         ER) ;
K : = CODOP (I) ;
```

如果想反汇编代码K的指令I，应写：

```
FUNCTION MNEMO IS NEW UNCHECKED_CO-
         NVERSION(INTEGER, INSTRUC);
I : =MNEMO (K) ;
```

如此，就轻而易举地近乎于构成一个汇编器和一个反汇编器！

8.2.3 记录型结构的指明

一个记录型的表示方法可以按下格式指明：

```
FOR 记录型 USE
```

RECORD [排行子句；]

域名　　位置

　　⋮

END RECORD；

可能存在的排行子句使得记录占据具有指定规模的多个地址。例如，可以规定这一类型的记录由偶数字节开始。其格式则为（设一个存贮单元为一个字节）：

RECORD AT MOD 2；

排行子句的一般格式为

AT MOD 静态表达式；

而每个分量的指明格式为：

域名 **AT** 静态简单表达式 **RANGE** 整型子界；

其中，表达式指出该分量开始处对于记录起点而言的相对地址（地址以存贮单元为单位，记录起点为零）。整型子界描述该分量存放在从哪一位到哪一位。第一位编号为零。

并不是所有的分量都要指明。对于那些未被指明的分量，由编译器来选择它们的表示方式。

地址区间不能交迭。如果是变体记录，那么不同的变体所占的地址区间可以交迭，因为它们是以一个代替另一个。

例如，PIA 6520/6820是一种接口组件。其功能可以概括如下：

该接口由4个寄存器组成。每个寄存器占一个字节。

地址	位号	表示符	功　能
0		PA	A通道数据寄存器或方向寄存器
1		CRA	A通道控制寄存器，由以下各位组成：

0	CA1S	规定CA1端是否对正脉冲敏感。
1	CA1I	说明CA1端的脉冲能否引起中断。
2	DDA	说明是否正在存取A通道的数据或A通道的方向。
3，4，5	CA2CTL	规定CA2端的性质。
6	IRQA2	由CA2引起中断。
7	IRQA1	由CA1引起中断。
2	PB	B通道数据寄存器或方向寄存器。
3	CRB	B通道控制寄存器。与CRA结构相同。

由此我们可以定义：

```
(PRAGMA STORAGE_UNIT (8) ; )
   OCTET: CONSTANT: = 1 ; ——一个八位字节
   TYPE PIA IS
      RECORD
         PA: ARRAY(0 . . 7) OF BOOLEAN;
         CA1S : BOOLEAN;
         CA1I : BOOLEAN;
         DDA : BOOLEAN;
         CA2CTL:INTEGER RANGE 0 . . 7;
         IRQA1: BOOLEAN;
         IRQA2: BOOLEAN;
         PB : ARRAY (0 . . 7) OF IBOOLEAN;
   ——CB1S, CB1B, DDB, CB2CTL, IRQB1,
IRQB2 的说明与A类似。
   END RECORD;
```

然后给出指明：

```
FOR PIA USE
    RECORD ——这里没有需用的排行子句
      PA AT O*      OCTET RANGE 0..7;
      CA1S AT 1*    OCTET RANGE 0..0;
      CA1I AT 1*    OCTET RANGE 1..1;
      DDA AT 1*     OCTET RANGE 2..2;
      CA2CTL AT1*   OCTET RANGE 3..5;
      IRQA2 AT 1*   OCTET RANGE 6..6;
      IRQA1 AT 1*   OCTET RANGE 7..7;
      PB AT 2*      OCTET RANGE 0..7;
--CBIS到IRQB1的指明与上面类似，但位置为AT 3*OCTET
    END RECORD;
FOR BOOLEAN USE(FALSE⇒0, TRUE⇒1);
```

这最后一个指明尽管可以有，但在大多数情况下是由系统自动规定的。

这样，可以很容易地对PIA操作。读者可以仿照上例，试对微处理机6502的状态寄存器在存贮中的复制写出指明。这个状态寄存器的结构如图8.1所示。其中第五位未被使用。

N	V		B	D	I	Z	C
7	6	5	4	3	2	1	0

图8.1 状态寄存器结构

8.2.4 地址指明

指明一个对象或程序的初始地址的格式为：

FOR 名 **USE AT** 静态简单表达式；

一个子程序、程序包或任务的起始地址即由此模块体所产生的机器码所占有的起始地址。例如：

```
FOR CHR_GET USE AT 16* 0070*; --程序起始于
                              70hex
FOR KEYBD_BUF_INDEX USE AT 16* 9E*
      --变量起始于 9E hex
```

例8.2 有两个与前节例中相似的PIA置于E810及E820 hex位置上。加入必要的指明。

解：

```
PIA1, PIA2 : PIA
FOR PIA1 USE AT 16* E810*;
FOR PIA2 USE AT 16* E820*;
```

这一步一旦完成，对PIA的操作就非常容易。例如允许PIA 1 的CA 1 端引起中断，只需要写：

```
PIA1.CA1I := TRUE;
```

其他操作也同样简便。

ADA既是一种很高级的语言，又能方便地在机器语言的等级上操作。因而成为能够用于复杂过程控制的优秀语言。此外，**RECORD**型的指明是一种描述硬件特性的极好的工具（如对PIA的描述）。

8.2.5 中断

一个任务入口的地址被指明的同时，这个地址即把这一

入口与一个中断联系起来(中断地址即为处理程序的开始)：

```
TASK INTERRUPTION IS
    ENTRY NMI;
    ENTRY IRQ;
    FOR NMI USE AT 16# 1000#;
    FOR IRQ USE AT 16# 2000#;
END;
```

例8.3 在BASIC.中，“POKE”语句可以用来把需要的数据写进预期的地址。这在ADA中如何实现？

解：可以利用如下指明：

```
TYPE V8 IS RANGE 0 . . 255;
VARIABLE : V8 ;
FOR V8′ SIZE USE 8 ;
FOR VARIABLE USE AT 预期地址;
```

然后把需要的数据赋予变量VARIABLE：

```
VARIABLE : = 需要的数据;
```

8.2.6 引入用汇编语言的程序段

可以定义一个程序包，它实质上是与机器相关的。这个程序包规定了不同的指令及指令格式（寻址方式）。例如，对于6502，可以有：

```
PACKAGE MACH_6502 IS
    TYPE IMMEDIATE_ADDRS IS (…LDA,
                ADC, …接受立即寻址的指令…) ;
    TYPE ABSOLUTE_ADDRS IS (……)
    ⋮
```

如此对所有寻址方式定义。这样，如果用户试图以某种寻址方式使用一条指令，而这种寻址方式是该指令所不接受的，那么将会报告出错。

```
TYPE IMMEDIATE_FORMAT IS
RECORD
    CODE : IMMEDIATE_ADDRS;
    FACTOR : INTEGER RANGE 0 . . 255;
END RECORD;
```

：对其他寻址方式的格式相同。

对于以上类型，写出便于记忆的表示指明：

```
FOR IMMEDIATE_ADDRS USE (…LDA⇒16*
                          A 9*, …);
```

对每种寻址方式写出如上指明，然后再对每条指令格式写出如下指明：

```
FOR IMMEDIATE_FORMAT USE
    RECORD
        CODE AT O* OCTET RANGE 0 . .7;
        FACTOR AT 1* OCTET RANGE 0. .7;
END RECORD;
    ⋮
程序包还可包括内部寄存器的定义等
END MACH_6502;
```

接下去，我们可以写出一个过程INLINE，其说明部分只包括一些**USE**子句和一些**PRAGMA**。过程体则只包括汇编指令。例如：

```
PROCEDURE MACHINE_LANGUAGE IS
```

```
    PRAGMA INLINE (MACHINE_LANGU
                AGE) ;
    USE MACH_6502;
BEGIN
    IMMEDIATE_FORMAT' (CODE⇒LDA,
          FACTOR⇒16*FF*) , --即LDA # $FF
    ABSOLUTE_FORMAT' (CODE⇒STA,
          ADDRS⇒16*1000*; --即STA $1000
    :
END;
```

总之，我们可以把ADA作为汇编语言使用。此外，如果需要调用由汇编语言写成的外部程序，可以利用参注**PRAGMA** INTERFACE。

附录 I　关 键 词

键　　词	作　　用	页
ABORT（停车）	任务非正常终止执行	
ACCEPT（接受）	任务的汇合	
ACCESS（存取）	定义指针类型	
ALL（整体）	指针指向的全部元素	
AND（与）	逻辑算符	
ARRAY（数组）	数组指明	
AT	地址指明或排行指明	
BEGIN	分程序、子程序、程序包和任务可执行部分的起始点	
BODY（体）	程序包体、任务体……	
CASE	CASE 结构或记录的变体	
CONSTANT（常数）	常数的引入。	
DECLARE	分程序说明部分的开始	
DELAY（延迟）	产生一个延迟	
DELTA（偏差）	定点实型的绝对偏差	
DIGITS	浮点型有效数字位数	
DO	汇合接受体的开始	
ELSE	IF语句的选择， SELECT 语句的选择， 或由OR ELSE 构成的逻辑结构。	
ELSIF	在IF语句中，否则如果选择	

续 I

键　　词	作　　用	页
END	分程序，子程序或结构的结	
ENTRY（入口）	束任务的入口	
EXCEPTION（异常）	异常指明	
	异常程序的开始	
EXIT（出口）	循环出口	
	下标循环中的下标管理，	
	指示指明	
FUNCTION（函数）	函数指明	
GENERIC（类属）	类数参数的引入	
GOTO	无条件转移	
IF	如果结构	
IN	括号算符，	
	下标循环范围，	
	参数结合方式。	
IS	引入描述	
LIMITED	受限私有型	
LOOP（循环）	（FOR、WHILE或不定）	
	循环	
MOD	模　数，	
	排行子句	
NEW	派生类型的引入，	
	类属的衍生，	
	指针变量地址分配。	
NOT（非）	非算符	
NULL	空语句	
	空元素（指针不指向任何元	

续 I

键　　词	作　　用	页
	素）	
OF	数组的基类型	
OR （或）	或算符	
OTHERS	SELECT语句中的选择整体常值中的其他元素，CASE结构中的其他假设，异常中的其他假设。	
OUT	参数结合方式	
PACKAGE （程序包）	程序包定义	
PRAGMA （参注）	引入命令	
PRIVATE （私有）	私有型，引入程序包的私有部分。	
PROCEDURE （过程）	过程指明	
RAISE （引发）	引发异常	
RANGE	数值范围，分配给记录型分量内存长度（以位计算）	
RECORD （记录）	记录型	
REM	余　数	
RENAMES （重命名）	重命名	
RETURN （返回）	函数的返回	
REVERSE	递　减	
SELECT （选择）	任务中的选择	
SEPARATE	分离编译	
SUBTYPE （子类型）	子类型指明	
TASK （任务）	任务指明	

续 I

键　　词	作　　用	页
TERMINATE	任务结束	
THEN	IF语句中的选择，由AND THEN构成的逻辑结构	
TYPE(类型)	类型指明	
USE	程序包的使用	
	表示指明	
WHEN	出口条件	
	CASE结构中的假定条件，异常中的假定条件	
WHILE	条件循环	
WITH	引入库程序包	
	在类属中引入子程序参数	
XOR(异或)	逻辑算符	

ADA中的专有字符及组合符

符　　号	作　　用	出现次页
+	加	
−	减	
*	乘	
/	除	
&	接续	
<	小于	
>	大于	
=	等于	

续 I

符　　号	作　　用	出现次页
.	小数部分或限制名	
:	变量指明	
	分程序标识符	
#	使用10以外的基	
E	指数	
' （单引号）	字符定界符	
	属性	
	限制名	
" （双引号）	字符串	
, （逗号）	清单分隔符	
; （分号）	语句结束符	
	在说明中两个参数的分隔符	
（…） （括号）	结构常值	
	下标	
	参数	
| （直杠）	在选择中两个元素的分隔符	
——	引入注释	
**	指数运算符	
<=	小于或等于	
>=	大于或等于	
/=	不等于	
«	标号的开始	
»	标号的结束	
=>	参数的结合或选择的结合	
<>	不定子界	
-	子界	
:=	赋值或衍生	

附录Ⅱ　ADA字符集

列举128个ASCII字符的CHARACTER型定义了ADA字符集。不可打印字符（控制字符）用符号来表示。

```
TYPE CHARACTER IS
 (NUL, SOH, STX, ……, VS,
  ' ', '! ', '" ', '#', '&', '%', ……,
'/',
  '0', '1', '2', '3', 4', '5', ……,
'?',
  '@', 'A', 'B', 'C', 'D', 'E', ……, 'O',
  'P', 'Q', 'R', 'S', 'T', 'U',……, '←',
  '、', 'a', 'b', 'c', 'd', 'e', ……, 'o',
  'p', 'g', 'r', 's', 't', 'u', ……, DEL);
```

在简化字符集中，程序包ASCII给每个字符一个表示名：

```
PACKAGE ASCII IS
——控制字符表示名如下：
NUL : CONSTANT CHARACTER : = NUL;
SOH : CONSTANT CHARACTER : = SOH;
    :
VS : CONSTANT CHARACTER: = VS;
DEL : CONSTANT CHARACTER: = DEL;
——每个专用字符授与一个名：
EXCLAM : CONSTANT CHARACTER: = '! ';
SHARP : CONSTANT CHARACTER: = '#';
```

```
    :
--每个小写字母授与一个名，形式为LC……
LC_A : CONSTANT CHARACTER := 'a' ;
    :
LC_Z : CONSTANT CHARACTER := 'Z'
END ASCII;
```

下面给出专用字符表：

EXCLAM	!	SHARP	#
DOLLAR	&	QUERY	?
AT_SIGN	@	L—BRACKET	[
BACK_SLASH	\	R—BRACKET	]
CIRCUMFLEX	^	GRAVE	`
L_BRACE	{	BAR	1
R_BRACE	}	TILDE	~

最重要的控制字符有：

BS (backspace)　　退格
HT (horizontal tabulation) 水平格式
LF (line feed)　　换行
VT (Vertical tabulation) 垂直格式
FF (form feed) 换格式单元
CR (Carrige Return) 回车

附录 Ⅲ　预定义元素

预定义属性已经在第Ⅲ章和第Ⅵ章中说明；
预定义参注已经在第Ⅷ章中说明；
程序包STANDARD和SYSTEM已经在第Ⅵ章中说明；
校核测试在第Ⅶ章中已列表说明；
一般预定义异常有：
CONSTRAINT_ERROR
NUMERIC_ERROR
SELECT_ERROR
STORAGE_ERROR
TASKING_ERROR
在程序包INPUT_OUTPUT和TEXT_IO 定义的异常有：
NAME_ERROR（使用一个不正确的名）
USE_ERROR（试图在一个文件上进行不可能的操作）
STATUS_ERROR（试图再建立一个已经已存在的文件）
DATA_ERROR（发现错误的数据）
DEVICE_ERROR（外设硬件功能错误）
END_ERROR（文件结束后进行读操作）
LAYOUT_ERROR（格式错误，试图超出行宽）

附录Ⅳ　法英汉名词索引

agregat	aggregate	聚集
allocateur	allocater	分配器
attribut	attribute	属性
bloc	block	分程序
boucle	loop	循环
boucle á indice courant	indexed loop	下标循环
boucle indefinie	indefinite loop	不定循环
boucle while	While loop	当型循环
cache	hide	隐藏
chaine de caracteres	character string	字符串
clause	clause	子句
clause generique	generic clause	类属子句
clause use	use clause	使用子句
compilateur	compiler	编译器
composant	component	成分
composant indicé	indexed component	下标成分
composant selectionné	selected component	所选成分
concaténation	concatenation	联结
configuration	configuration	配置
commentaire	comment	注解

construction case	case statement	分情况语句
contexte specification	context specification	上下文指定
contrainte	contraint	约束
corps	body	体
déclaration	declaration	说明
declenchement d'une exception	raise an exception	引发例外
définition	definiition	定义
délimiteur	delimiter	定界符
dimension	dimension	维长
discriminant	discriminant	判别式
élaboration	elaboration	拟定、确立
élément	element	元素
entité	entity	实体
entrée	entry	输入、入口
énumeration	enumeration	枚举
exception	exception	异常、例外
exécution suspendue	suspend execution	挂起执行
expression	expression	表达式
fonction	function	函数
generique	generic	类属的
identificateur	identifier	标识符
incompatible	incompatible	不相容
indece	index、subscript	下标
inisialisation	inisialization	赋初值
instantiation	instantiation	衍生

instruction d'affectation	assignment statement	赋值语句
instruction exécutable	executable instruction	可值行语句
intervalle	interval	子界
limite	Limited	限制
littéral	literal	字面值
modéle	template	模型
nombre basé	based number	带基数
non contraint	unconstrained	无约束的
objet	object	对象
package	package	程序包
parametre	parameter	参数
parametre formel	formal parameter	形式参数
pragma	pragma	参注
prédécesseur	predecessor	前导值
prive	private	私有型
procédure	procedure	过程
programme structure	structure program	结构化程序
propagation d'exception	propagation of exception	例外的传播
qualification	qualification	限制
record	record	记录
record avec variante	variant record	变体记录
record parametré	parametric record	带参记录
recursivité	recursion	递归

rendez-vous	rendezvous	会合、约会
reste	remainder	余
sortie	exit	输出
sousprogramme	subprogram	子程序
sous-type	sub-type	子类型
sousunité	subunit	子单元
successeur	successor	后续值
surcharge	overload	重载
synchronisation	synchronization	同步
táche	task	任务
táche activation	task activation	任务激活
terminaation asynchrone	asynchronous termination	异步终结
type	type	类型
type acces	access type	存取类型
type composé	composite type	组合类型
type de donnée	data type	数据类型
type derivé	derived type	派生型
type discret	discrete type	离散类型
type enumératif	enumeration type	枚举类型
type générique	generic type	类属类型
type intégral	interger type	整型
type point fix	fixed point type	定点类型
type point flottant	floating point type	浮点类型
type privé	private type	私有类型
type record	record type	记录型
type réel	real type	实型

type réel universel	universal real type	泛 实 型
type tableau	array type	数组类型
typé tache	task type	任务类型
unite du programme générique	generic program unit	类属程序单元
urgence	urgency	紧 急
valeur initiale	initial value	初 值
valeur par defaut	default initial value	缺省值
variable	variable	变 量
variable partage	shared variable	共享变量
variante	variant	变 体
visibilité	visibility	可见性
vraie	true	真 值