

第1章 引言

Ada 是高级程序设计语言。Ada 语言由美国国防部发起研制,用于嵌入式系统应用领域。所谓嵌入式系统,是指作为大系统中一组成部分的计算机系统。例如,化学工厂控制系统,导弹控制系统等。

本章将简要地追述 Ada 的发展过程,介绍 Ada 语言的概况以及本书的总体结构。

1.1 历史

Ada 的历史要追溯到 1974 年。当时美国国防部意识到软件的生产费用太高。经过对各种应用领域内软件费用的详细调查和分析,发现一半以上的费用都直接投入到嵌入式系统的开发中。

再进一步对各种领域内使用的编程语言加以分析后发现,COBOL 被广泛地用于数据处理,FORTRAN 被广泛地用于科学和工程计算。虽然这两种语言并不先进,但在上述领域中一直广泛地被使用着,因此没有必要对它们重新投资。

而在嵌入式系统应用这一领域内,人们使用了名目繁多的语种。仅就美国三军而言,他们不仅有各自常用的高级语言,而且还大量地使用不同种类的汇编语言。此外,每种高级语言还有众多的变异版本,情形尤为混乱。就连那些看上去是很成功的开发合同,也都在不同程度上起了鼓励使用专用语言进行应用开发的作用。这种局面,导致把大量的钱花销于许多不必要的编译系统的研制上,加上缺乏统一标准,无形中增加了培训和维护的开支。

因此美国国防部认识到,要控制软件费用,必须在嵌入式系统领域内实行编程语言的标准化。最终的目标在于推出一种统一的标准语言,为了向标准语言过渡,美国国防部批准了一些在短期内暂时使用的语言。它们是 CMS - 2Y, CMS - 2M, SPL/1, TACPOL, JOVIALJ3, JOVIALJ73, 而 COBOL 和 FORTRAN 仍然还在其各自的应用领域内使用。

迈向标准语言的第一步,是拟定了一份有关标准语言性能要求的文件。文件的第一版称为“草人”(Strawman),1975 年公布。在征求了各方面意见的基础上,经过修订成为“木人”(Woodeman)。后来又经过进一步的修订,1976 年 8 月形成了“锡人”(Tinman),这是一份相当规范的文件,它标明了标准语言所应具有的性能。

在此阶段,依照“锡人”的要求对现存的许多语言进行分析,看是否有满足这些性能的语言存在,以便从中选出一种语言作为标准语言,同时对标准语言的性能细节进行分析和评估。正如事先已估计到的:没有一种现存的语言能符合要求,结论是应当设计一种新语言作为标准语言。

现存语言经分析,分为三种情况

1. 不合适: 这些语言或者已过时,或者不适用于嵌入式应用领域。这类语言包括 FORTRAN 和 CORAL 68。
2. 不太合适: 这些语言就其现状而言,不能满足要求,但有些特性可以借鉴。这类语言包括 RTL/2 和 LIS。

3. 推荐作为基础:有三种语言 Pascal, PL/I 和 Algol 68, 它们其中之一可作为标准语言的设计出发点。

根据对现行语言的分析, 在整理和修订“锡人”文件的基础上形成了“铁人”(Ironman)文件。

接下来是让合同商提出设计新语言的方案书, 方案书包括选择某个语言作为新语言设计的出发点。从收到的 17 份方案书中选出 4 份进行投标。为了在评选时不带偏见, 这 4 个合同商各用颜色作为评选时的匿名标志, 这 4 个方案是, CII Honeywell Bull(绿), Intermetrics(红), Softech(兰), SRI International(黄)。

初步设计于 1978 年初基本完成。在经过世界各有关组织的评议后, 美国国防部认为绿色和红色方案比兰色和黄色方案许诺的内容要多, 因此淘汰了后者。

开发进入到第二阶段, 美国国防部决定给入选的 2 个合同商一年时间来改进他们的设计, 根据初步设计所得到的经验对性能要求加以修订, 由此形成了最后文件——“钢人”(Steelman)。

1979 年 5 月 2 日对语言进行终选, 结果由琼·伊波斯领导的国际小组开发的绿色语言被选为优胜者。

美国国防部宣布, 将这一新语言命名为 Ada, 为的是纪念 Augusta Ada Byron 伯爵夫人(1815—1852)。Ada 是英格兰诗人拜伦(Byron)勋爵的女儿, 她曾帮助和资助 Charles Babbage, 并且参与了他的机械分析机工作, 从真正的意义上讲, 她是世界上第一个程序员。

Ada 开发进入的第三阶段, 是把语言交给大量的最终用户进行抽样程序设计并对结果进行评价, 为的是从用户那里得到该语言是否实用的意见。同时在美国和欧洲开设了各种课程, 并组织许多研究小组进行分析。1979 年 10 月在波斯顿会议上提出了 80 来份有关 Ada 语言的评价结论和修改意见的技术报告。总的结论认为 Ada 是好的, 但有几处需要进一步的修改。此外还收到了上千份较短的有关语言技术问题的报告。在考虑了所有这些报告后, 对初步设计的 Ada 又进行了审定, 终于在 1980 年 7 月公布了 Ada 语言报告的第一版, 并将其提交美国国家标准研究所(ANSI)作为标准。

ANSI 对 Ada 的标准化过程花了两年时间, 此间对语言做了一些改动, 其中多数的改动是微小的, 但却给编译程序的设计带来了巨大的便利。ANSI 标准的语言参考手册 1983 年 1 月出版, 它是编写本书的基础。

不要以为 Ada 仅仅是又一种新的程序设计语言, 而必须认识到语言本身仅仅是 Ada 为每个程序员提供的工具箱中的一个成份(尽管是一个重要的成份)。如果能建立起同一标准的程序设计环境, 则亦可得到额外的好处。因此在 Ada 语言设计的同时, 美国国防部相应地推出了有关 Ada 程序设计支持环境(APSE)性能需求的一系列文件。这些文件命名为“沙人”(Sandman), “泥人”(Pebbleman), 以及最终的“石人”(Stoneman)。

由于程序设计支持环境这一技术领域的发展尚处初级阶段, 因此这些文件比起相应的语言文件要简单。不管怎么说, 制订出有关 APSE 性能需求的文件, 至少能避免不必要的偏差, 为开发出高质量的 APSE 提供了条件。关于 APSE 讨论超出了本书的范围, 故不在此赘述。

1.2 技术背景

程序设计语言以其特定的方式发展和演变, 经历了三个主要的发展与进步过程。每次进步

都在一定的层次上引入了抽象。抽象使程序消除了不必要的有害细节。

第一次进步发生在 50 年代初期。当时的 FORTRAN 和 Autocode 引入了“抽象表达式”，使人们可以在程序中写出如下的语句：

X=A+B(I)

在这里利用机器的寄存器来计算表达式的值，对程序员来说是不可见的。由于在描述复杂的表达式时存在着不尽合理的限制，例如下标只能采用特别简单的形式，因此这些早期语言中的抽象表达式是不完善的。后来的语言，如 Algol 60 取消了这类限制，才真正地完成了抽象。

第二次进步是“控制抽象”。Algol 60 在这方面迈出了最显著的一步。没有别的语言能够象 Algol 60 那样对语言的发展产生过那么大的影响。**控制抽象的关键在于对构成的控制流和各个控制点不必命名或加以标号。例如我们可以写**

if X=Y then P:=Q else A:=B

此时，编译程序自动地生成出 goto 和标号，而在诸如 FORTRAN 等语言中则要在源程序中明显地给出。最初的抽象表达式中存在的缺陷此时再次在控制抽象中出现。其中明显的是 Algol 60 的开关语句，现在此种开关语句已由诸如 Pascal 等语言的 case 语句所代替。

第三次进步是“数据抽象”。数据抽象意味着把数据描述的细节从定义在数据上的抽象操作中分离出来。

多数现存的语言仅使用几种非常简单的数据类型。在所有的情况下，数据直接以数值单位加以描述。因此当使用的数据不是真正的数值（可能是交通灯的颜色）时，则必须经由程序员进行抽象类型的变换，将其变成数值类型（通常为整型）。除非是在注释中注明，这种变换仅仅在程序员脑子里进行而不写在程序中。这种现象的后果使得软件库中仅有数值分析类的成分。对于数值算法，比如找矩阵的特征值，由于仅涉及数值操作，因此那些把数据的值仅看作数值的语言是合适的。在其它非数值算法的情况下，因为不可能有符合所需的类型变换存在，故建不成程序库。实际上，最好在不同的情况下采用不同的类型变换，而且这些变换要能渗透到整个程序中。问题是，当改变这种类型变换时往往要导致重写整个程序。

Pascal 引入一些数据抽象，比如枚举类型。枚举类型使我们能在讨论交通灯问题时可用它们的术语，而不需要知道计算机中是如何对它们加以描述的。这使我们避免了在编程中可能造成的严重错误。例如把交通灯和其它抽象类型（例如鱼的名称）混为一谈。如果在程序中所有类型均以数值类型来描述的话，则有可能产生上述错误。

此外数据抽象还涉及到可见性，人们经过很长时间才认识到，传统的 Algol 60 的分程序结构是不能令人满意的。例如，要用 Algol 60 写两个过程对某公用数据进行操作，如果该数据不能被直接访问，则这两个过程也无法被调用。许多语言通过分别编译来保障对可见性的控制。这种技术能满足中等大小的系统。但由于分别编译功能依赖于外部系统，因而不能保证对可见性的完全控制。Modula 的模块是一个结构合理的例子。

另一个为数据抽象的发展做出巨大贡献的是 Simula 67 及其类别概念。此外，许多实验性的语言也做出了具体的贡献，因为太多故不在此一一介绍。

Ada 是第一个把各种类别的抽象数据结合在一起的实用语言。无疑 Ada 是很先进的，它为编写大量数值分析之外可重复使用的库程序提供了可能性。它将促进软件元件工业的建立。

1.3 本书的结构和目标

学习一种编程语言犹如学开车，必须先学会一些重要的技能。虽然不一定要知道怎样用风窗洗刷器，但至少能启动发动机，学会挂档，把握方向盘和刹车。同样，学习编程语言时，不必要学了所有内容后才能编写实用的程序。但是要进行程序设计必须先学习许多基本的东西。Ada的许多优点只有通过编写大型程序方能体会出来。

本书全面地介绍了怎样使用 Ada 编程，读者应具有一些用高级语言编写大型程序的知识和经验。如学过 Pascal 语言则将有助于学习 Ada，但也不是非其不可。

本书第 2 章简要地叙述了一些 Ada 的概念，为的是让读者体会出 Ada 的风格，并对该语言的设计思想有一定的了解。本书的其它部分均采用教科书的形式，直接地介绍主题。至第 7 章止，所述内容基本覆盖了小型语言如 Pascal 的传统功能。随后几章介绍了诸如抽象数据、大型编程以及并发处理等先进的让人感兴趣的内容。

大多数章节都配有习题，对读者来说做习题是非常重要的，如不能全做，最好也要做大部分。因为这些习题构成了论述的一部分，并且在后面的章节中往往要用到前面章节习题的结果。

大部分章的后面，给出了掌握的要点，尽管不完全，但它们有助于深入理解相应章节的内容。读者最好也能结合章节内容参考附录 4 的语法。该附录的语法是按各章介绍的主题依次组织的。

本书描述了 Ada 的全貌，但有几处的讨论是不完全的，它们是定点计算，基于具体机器的编程和输入/输出。定点计算是较高的专题，大多数程序员不一定感兴趣。基于具体机器的编程，顾名思义即为基于具体宿主机器上的程序设计，只简便地介绍一下即可。输入/输出虽然重要，但没有引入新的概念，只是大量的细节，因此只作简单介绍。有关这些方面的详细资料可在语言参考手册(LRM)中找到。但需要注意，不同时期的参考手册各不相同。

书后的附录使本书自成体系，它们基本上来自语言参考手册。建议选读 LRM 中正式的 Ada 定义。

1.4 参考书

1. Department of Defense Requirements for High Order Computer Programming Languages—“STEELMEN”, Defense Advanced Research Projects Agency, Arlington, Virginia, June 1978.
2. Reference Manual for the Ada Programming Language, (ANSI/MIL-STD-1815A) United States Department of Defense, Washington D. C., January 1983.
3. Department of Defense Requirements for Ada Programming Support Environments—“STONEMAN”, Defense Advanced Research Projects Agency, Arlington, Virginia, February 1980.

第 2 章 Ada 的概念

本章将简要地介绍 Ada 的概念,特点和设计目标。

2.1 主要目标

Ada 是一种大型语言。它比 Pascal 要大的多。Pascal 只适用于培训(即该语言的原设计目标)和进行小规模的所谓“个人程序设计”。Ada 涉及到许多有关实用系统编程的重要问题。它们是:

- 易读性——我们知道专业程序员读的程序要比写的程序多的多。因此必须避免象 APL 那种以过于简要的符号编程的语言。尽管用这种语言编程速度很快,但却导致所编的程序几乎是不可读的。

- 强类型——强类型保证了每个被说明的对象有明确定义的值域,并防止了不同概念的逻辑混淆。因此许多错误能够被编译程序发现,而同样的情况对于其它语言来说,则可能导致一个不正确的程序。

- 大型编程——封装机制,分段编译和库管理对于编写任何规模的可移植和可维护的程序来说都是不可少的。

- 异常处理——在实际情况下,程序很少能永远保持其运行的正确性。为此,需要把程序构造成分块结构和层次结构,并提供一种手段,以便把某一部分因发生错误所造成的影响控制在一定范围内。

- 抽象数据——如前所述,须把数据描述的细节同基于数据的特定逻辑操作相分离,以增强可移植性和可维护性。

- 多任务——把程序想象为一系列并行活动而不仅仅是单一的顺序活动,这对许多应用来说尤为重要。在语言中增设适当的并发机制,要比借助调用操作系统来提供这种性能,更有益于可移植性和可靠性。

- 类属设施——在许多情况下,程序的某逻辑部分与运行时值的类型无关。因此,需要提供一种机制,以便从唯一的样板中产生出多个相应的程序段来。这样建立的程序库特别有用。

2.2 概述

程序设计的一个主要方法是反复地引用已有的程序,使新编的程序量尽可能少。这就自然而然地产生了程序库的概念。同时,亦要求编程语言应具有引用程序库中内容的能力。

Ada 考虑到这种情形,从而引入了库单位的概念。一个完整的 Ada 程序可看作由一主程序(其本身亦可为库单位)及被它调用、为它服务的其它库单位所组成。这些库单位可看成是构成整个程序的最外层词法单位。

主程序是一个具有相应名字的过程。服务库单位可以看成是子程序(过程或函数),但它们看起来更象程序包。程序包是一组有关的项,而这些项又可以是如同子程序那样的实体。

例如我们要编写一个打印出 2.5 的平方根的程序。我们可期望有一些库单位来供我们计

算平方根和给出输出。我们只要编写一个主程序来调用这些为我们服务的库单位即可。

假定调用程序库中名为 SQRT 函数就能得到平方根。另外，假定我们的程序中含有名为 SIMPLE_IO 的程序包，该程序包含有各种简单的输入/输出功能，这些功能包括读数据、打印数据、打印字符串等。

程序如下：

```
with SQRT,SIMPLE_IO;
procedure PRINT_ROOT is
    use SIMPLE_IO;
begin
    PUT(SQRT(2.5));
end PRINT_ROOT;
```



此程序被设计成名为 PRINT_ROOT 的过程。在过程名前有一 with 子句，它给出了要用到的库单位的名字。该过程体只含一个语句

```
PUT(SQRT(2.5));
```

此语句带参地调用程序包 SIMPLE_IO 中的过程 PUT，而 PUT 的参数又是一个带有参数 2.5 的函数，就是说把 SQRT 函数计算的结果作为 PUT 的参数

```
use SIMPLE_IO;
```

使我们可直接取用 SIMPLE_IO 程序包中的成份。如果省去该子句，则就必须写成：

```
SIMPLE_IO.PUT(SQRT(2.5));
```

以便指出从何处能找到 PUT。

如果我们采用读入方式输入所要开方的数据，则该程序将变的更为有用，程序变为：

```
with SQRT,SIMPLE_IO;
procedure PRINT_ROOT is
    use SIMPLE_IO;
    X:FLOAT;
begin
    GET(X);
    PUT(SQRT(X));
end PRINT_ROOT;
```

不难看出，此时整个过程的结构更加清晰，在 is 和 begin 之间是定义。在 begin 与 end 之间为语句。概括地说：说明引入了我们要操作的实体，而语句则用来表示依次的动作。

在该过程中，引入了一类型为 FLOAT 的变量 X，FLOAT 是预定义类型。这种类型的变量是一组浮点数。对 X 的说明，表示 X 的值只能在这组值中。本例中通过调用 SIMPLE_IO 程序包中的过程 GET 来给 X 赋值。

应对某些细节加以注意，各种语句和说明都用分号终结。这不象其它语言，如 Algol 和 Pas-

cal 中分号作为分隔符,而不是终结符。程序中有多个标识符,如 procedure, PUT 和 X,这些标识符可分为两类,一类(共 63 个)如 procedure 和 is 等,用于标志程序的结构。它们为保留字,不能用于其它用途。另一类,如 PUT 和 X 可用于我们需要的地方。其中一些,如例子中的 FLOAT,有其预定的含义,虽然我们可以把它们再作其它的用途,但将会造成混乱。为了清楚起见,在本书中,保留字为小写,而其它标识符都为大写。除非字符本身作为被处理对象,语言不对大、小写字符加以区别。还要注意,用横划线可把较长的定义符分为有意义的多个部分。

最后要注意过程名,PRINT_ROOT 在最后的 end 与分号之间再次出现,尽管它可有可无,但是为了结构清晰,我们建议最好这样使用。

我们的程序很简单,如能对一串数据进行处理,并分行打印结果,则程序就会更加有用。我们以零值来终止程序运行。程序如下:

```
with SQRT,SIMPLE_IO;
procedure PRINT_ROOT is
    use SIMPLE_IO;
    X,FLOAT;
begin
    PUT("Roots of various numbers");
    NEW_LINE(2);
    loop
        GET(X);
        exit when X=0.0;
        PUT("Roots of");
        PUT(X);
        PUT("is");
        if X<0.0 then
            PUT("not calculable");
        else
            PUT(SQRT(X));
        end if;
        NEW_LINE;
    end loop;
    NEW_LINE;
    PUT("program finished");
    NEW_LINE;
end PRINT_ROOTS;
```

调用 SIMPLE_IO 程序包中的过程 NEW_LINE 和 PUT 增强了输出功能。调用 NEW_LINE 可按指定的行数换行,如果调用时参数缺省,则按约定值作为参数。还可以把字符串作为参数来调用 PUT,事实上它与打印 X 所使用的不是同一个过程。编译程序可根据参数类型来

区分应该使用那个过程。多个同名的过程称为重载。在此要注意字符串的形式，这里字符的大、小写是有区别的。

此处引入了新的控制结构，在 loop 和 end loop 之间的语句将重复执行，直到在 exit 语句中条件 $X=0.0$ 为真时止。当条件满足时就结束循环，直接执行 end loop 之后的语句。此外还要测试 X 是否为负数，如果是负数，则输出“not calculable”而不调用 SQRT，这是用条件语句来完成的。如果在 if 和 then 之间的条件为真，则执行 then 和 else 之间的语句，否则执行 else 和 end if 之间的语句。

要注意，括号式结构 loop 与 end loop, if 与 end if 的匹配，Ada 的所有控制结构都是封闭式的，而不象 Pascal 为开放式的（例如：if then 语句，没有 end if 与之匹配）。开放式结构是不好的结构，它往往导致编程错误。

试想，假如我们不测试 X 是否为负数，而用负的自变量来调用 SQRT。如果 SQRT 本身在编程时又只考虑参数仅为正数的情况，那么 SQRT 函数就不能做为 PUT 的参数值，而会产生异常。异常的产生标志着发生了不正常的事情，并且正常的执行顺序被打断。在我们的例子中，引发的异常为 NUMERIC_ERROR。如我们不对发生的异常进行处理，则我们的程序将会自行终止，并且无疑 Ada 语言支持环境(APSE)将给出程序运行和为什么失败的信息。当然，我们可以监测异常的发生，一旦异常发生，便采取补救措施。事实上，我们可以用下面的方法来替代条件语句：

```
begin
    PUT(SQRT(X));
exception
    when NUMERIC_ERROR=>
        PUT("not calculable");
end;
```

现在让我们来简要地讨论所使用的 SQRT 函数以及 SIMPLE_IO 程序包的一般格式。

SQRT 函数与我们的主程序结构相同，主要的区别在于具有参数

```
function SQRT(F:FLOAT) return FLOAT is
    R:FLOAT;
begin
    --计算 SQRT(F)的值，赋给 R;
    return R
end SQRT;
```

从这里可以看出形参的描述（这里只有一个）和结果类型，计算细节由两个横划线之后的注释来表示。注意，函数将返回结果，并且只能在表达式中被调用。过程不返回结果，用一个语句来调用，这是两者的主要区别。

SIMPLE_IO 程序包有两部分，规范式说明和程序包体。规范式说明用来对程序包与外部的接口加以描述，程序包体则含有程序包实现细节。如果程序包只含有我们所使用的过程，则

该程序包的规范式说明应为：

```
package SIMPLE_IO is
    procedure GET(F:out FLOAT);
    procedure PUT(F:in FLOAT);
    procedure PUT(S,in STRING);
    procedure NEW_LINE(N,in INTEGER := 1);
end SIMPLE_IO;
```

GET 的参数是输出参数,因为调用 GET
GET(X);

是从 GET 过程中输出一个值给实参 X,其它参数都是输入参数,因为是向过程输入值。在程序包的规范式说明中出现的仅为每个过程的一部分,这部分称为过程规范式说明,知道这些信息对调用程序包已足够了。在上述的程序包的规范式说明中有 PUT 的 2 个重载规范式说明,一个有 FLOAT 类型的参数,而另外一个有 STRING 类型的参数。最后要注意 NEW_LINE 参数缺省时,被默认为 1。

SIMPLE_IO 程序包体中包含所有的过程体,以及实现过程所需的其它支持成份。当然,这些对用户而言是不可见的。程序包体的大致结构如下:

```
with INPUT_OUTPUT;
package body SIMPLE_IO is
    ...
    procedure GET (F,out FLOAT) is
        ...
    begin
        ...
    end GET;
    -- 其它过程相似;
end SIMPLE_IO;
```

with 子句表示在 SIMPLE_IO 中过程的实现用到了更为一般的程序包 INPUT_OUTPUT。应注意,作为完整的 GET 体,仍含有此过程的规范式说明,尽管它已在相应程序包的规范式说明中给出了(2 个规范式说明相似,但程序包规范式说明中没有 is)。

本节中的例子说明了 Ada 的整体结构和控制语句。详细的数据类型将在下面的章节中加以讨论。本节的目的之一在于强调程序包的思想在 Ada 中是最为重要的概念之一。一个程序应看作为由多种成份组成,这些成份有的是提供服务,有的是接受其它成份的服务。在到第 8 章为止的下面几章,将讨论 Ada 的细微特征。但我们仍要从整个结构着眼。

顺便要提一下特殊的程序包 STANDARD。该程序包在每个实现中都有,它包含对所有预定义标识符,如 FLOAT 和 NUMERIC_ERROR 的说明。我们认为可自动地访问 STANDARD 而不必在 with 子句中给出它的名称。对该程序包的叙述请见附录 2。

练习 2.2

1. 在实际情况中 SQRT 函数不是在库中单独存在的,而是和其它数学函数一起于同一个程序包中。假设这个程序包的标识符为 SIMPLE_MATHS, 其它的函数有 LOG, EXP, SIN 和 COS, 按照 SIMPLE_IO 的规范式说明写出这个程序包的规范式说明, 向 PRINT_ROOTS 程序将要如何修改?

2.3 错误

一个 Ada 程序往往由于各种原因而有可能不正确。依据程序中错误的发现途径, 可把错误分为 4 类。

有些错误可被编译程序发现, 其中包括简单的书写错误, 例如忘了写分号和违反了类型规定, 把颜色与鱼的名称混在一起。在这些情况下程序将不被执行。

有些错误在程序运行时才能被发现。例如, 求一个负数的平方根或除数为 0。在这种情况下, 就会产生上节中所看到的异常。异常处理为我们提供了视情况予以克服的机会。

还有些情况, 程序违反了语言的规定, 但不能轻而易举地被发现。例如, 程序中不得使用事先未赋值的变量。如果使用了未赋值的变量, 则无法预料程序将如何运行。显然这种程序是错误的。

最后一种情况是发生在执行中, 即没有按照应做事情的顺序来描述处理过程。例如, 在没有定义过程参数前, 便对参数进行了赋值。如果程序依照这样的次序执行则显然是非法的, 这种错误称为依从次序错误。

2.4 输入/输出

Ada 语言的所有输入/输出与其它语言相同, 没有特殊之处。实际上, 输入/输出功能只是由程序包来提供。这随之而产生一种副作用, 不同的实现将提供不同的程序包, 从而影响了程序的可移植性。为了避免这种情况, 语言参考手册描述了一些实用的标准程序包, 其它复杂的程序包可在特殊情况下使用。对于非常简单的程序包, 象 SIMPLE_IO 在多数场合已足够了, 对输入/输出更深一步的讨论将留待第 15 章(程序与外部接口)时讨论。

2.5 术语

每个专业都有自己的术语。Ada 也不例外, 附录 3 给出了 Ada 的词汇表。

术语一般是按要求引入的, 在开始讨论 Ada 的细节之前, 我们将反复提到一些经常要用的概念。

静态: 是指在编译时能确定的情况。反之, 动态则是指执行时才能确定的情况。静态表达式是在编译时就能确定其值的表达式。如:

2 + 3

静态数组是在编译时就知道其下标范围的数组。

有时, 需要给编译程序一个带有括弧的标志。该标志不属程序的一部分, 但它是很有用的暗示。这是靠称之为编译注释的 pragma 来实现的。例如我们向编译程序指出, 把程序的某一部分列出来, 便可采用如下形式:

```
pragma LIST(ON);
```

和

```
pragma LIST(OFF);
```

通常 `pragma` 能出现在说明和语句能出现的任何地方, 以及在分号之后和其它上下文中。有时, 对特定的 `pragma` 的位置有限定, 详细情况请见附录 1。

第3章 词法形式

前两章,我们介绍了 Ada 的一些概念,本章我们将讨论一些细节。

我们的讨论是从最基本的内容,例如编程所需要的标识符、数据的详细结构等入手。这就如同人们要学好一门语言,首先要先学拼写一样。从程序设计语言的角度来看,编译程序是严格的检查者。

我们还要介绍描述 Ada 语言结构的语法表示法。一般情况下,我们避免用语法表示法来介绍概念。但有时要对某个概念进行精确地描述,最容易的办法就是用语法表示法。如果读者要参阅 LRM,语法知识是必不可少的。为了本书的完整性和使用方便,在附录中给出了 Ada 的全部语法。

3.1 语法表示法

Ada 的语法是用改进的后向回溯法来描述的。语法单位是用小写字符组成的名来表示。有些名中嵌有横划线,是为了增加可读性。一个语法单位是用其它的语法单位和一个类似于等号的符号来定义。有些语法单位是原子,不可进一步分解,这样的语法单位称为终止符。一个产生式是由被定义的名后跟一个特殊的符号 ::= 和定义来组成的。

所用到的其它符号有:

- 方括弧中是可选择项。
- () 括弧中的项可省略,也可出现一次或重复多次。
- | 垂直线用来分隔可选择项。

有时,词法单位的名字字首是斜体字,这种情况下,字首用于传递语言信息。在上、下文自由语法中,它被作为注释格式。有时一个产生式的格式为建议格式。

3.2 词法元素

Ada 程序是按有顺序的文本行写出的,文本行使用下列字符集:

- 字母 A-Z
- 数字 0-9
- 各种其它的符号 "# & ' () * + - .. / ; ; < = > _ |
- 空格符

除了用大写字母外,也可用小写字母,一般认为两者是相同的(只有在作为字符直接量时例外)。

有些特殊的字符如! 也可以作为串和字符直接量的值。当不可用|, # 和"时,可用别的字符来代替,我们不考虑如何进行代替和与之有关的特殊规定。在本书中只考虑通常的字符。

在 LRM 中没有描述如何从前一行到后一行。我们也不做考虑，我们只认为 Ada 程序是若干行组成，每一新行是由键盘提供的新行符开始。

Ada 文本中的一行可以被看做为词法元素的有序集。在第 2 章中，我们已遇到过几种词法元素的形式。例如：

AGE : = 43; —— 约翰的年龄

这里有 5 个元素。

- 标识符 AGE
- 复合词 : =
- 数据 43
- 单个符号 ;
- 注释 —— 约翰的年龄

词法元素的其它类是字符串和字符直接量。我们将在第 6 章中再介绍。

独立的词法元素内不可插入空格，除此之外，可以随意插入空格符以使程序美观。例如用空格来展示整个程序的结构。一个词法元素必须在一行内。

请注意，下面的复合定界符内不准许有空格：

- => 用于集合，状态等
- * * 用于范围
- * * 求幂
- : = 赋值
- / = 不等于
- >= 大于或等于
- <= 小于或等于
- << 标号的括号
- >> 另一个标号的括号
- <> 用于数组和类属的“盒子”

空格可以出现在字符串和字符直接量中，此时它本身即为字符。空格也可出现在注释中。注意，相邻的定界符和数据必须用空格分开，否则将会引起混乱。例如要写成 end loop 而不能写成 end loop。

3.3 标识符

在第 2 章的例子中遇到过标识符。作为如何使用语法表示的例子，我们现在用语法来定义标识符。

标识符 ::= 字母 { [横划线] 字母数字 }

字母数字 ::= 字母 | 数字

字母 ::= 大写字母 | 小写字母

上述定义指出一个标识符是由一个字母后跟零个、一个或多个字母或数字组成，并且在字母或数字之前可加横划线。字母可以大写，也可以小写。就此例而言，横划线，数字，大写字母和小写字母都是不可再分的。它们都是终结符。

通俗地说，一个标识符是由一个字母后跟单个或多个字母、数字组成，字母大、小写都可以。语法中没有说明标识符是否与字母的大、小写有关。实际上，只是字母大、小写不同的标识符表示的是同一个标识符。但是横划线则至关重要。

Ada 没有对标识符的长度加以限制，似乎构成标识符的所有字符都有效。但是实际上是有限制的，一个标识符不能超过一行，而一行的长度是有限的。应鼓励程序员用有意义的标识符，如 TIME_OF_DAY，而不要用无意义的缩写 T。使用长标识符可能在第一次写程序时使人感到麻烦。但在程序生命周期中，读程序比写程序要多的多，程序写得清楚有助于今后编程者和维护人员对程序的理解。当然，对于短的数学式子或抽象的子程序，采用 X 和 Y 这样标识符反而清晰。

在程序中用标识符来命名各种变量，有些标识符是特殊的语法保留字，不能再被另作它用，这在第 2 章中已遇到过一些，如 if, procedure 和 end。在 Ada 中总共有 63 个保留字（列在附录 1 中）。在程序中保留字与所有的标识符一样，可以大写亦可以小写，还可以大、小写混用。如 procedure, PROCEDURE 和 Procedure 都是允许的。不过作些约定将有助于理解。建议保留字为小写，其它定义符为大写，这只是一种风格而已。

保留字 delta, digits 和 range 等有些例外，以后会看到，它们可以用于属性，以大写出现，如 DELTA, DIGITS 和 RANGE。由于它们作为属性，在使用时单引号总是出现在第一个字母之前，一般不会引起混乱。有些标识符，如 INTEGER 和 TRUE 在 STANDARD 程序包中预定义了其含义，它们不是保留字，可另作它用，但这样做会使程序变的非常混乱，希望特别留意。

练习 3.3

1. 指出下列哪些不是合法的标识符，为什么？

- | | | |
|----------------|---------------|------------|
| a) Ada | d)UM0164G | g)X_ |
| b)fish&chips | e)TIME____LAG | h)tax_rate |
| c)RATE—OF—FLOW | f)77E2 | i)goto |

3.4 数

数（专业术语是数值直接量）具有两种形式，即整数（是精确的）和实数（是近似的）。整数又称整型直接量，实数又称实型直接量。它们的主要区别是实数总有一个小数点，而整数没有小数点。Ada 不允许数值类型的混用。在上、下文确定为实型直接量的场合使用一个整型直接量是非法的，反之亦然。如：

AGE:INTEGER := 43.0;

和

WEIGHIT:REAL := 150;

两者都是非法的。（用 REAL 类型而不用 FLOAT 类型的原因以后就会清楚）。

整型直接量的最简单的格式是一串十进制数字，如果数字较多时，可适当嵌入横线将其分为几组。如：

123_456_789

参照定义可知,这里的横划线无其它意义,只是增加可读性。

最简单的实型直接量是一串含有一小数点的十进制数字。注意,在小数点的两边都必须至少有一个数字。嵌入横划线亦可增加可读性,但横划线不得与小数点相连。如:

3.14159_26536

与很多语言不同,Ada的整数和实数都可带指数部分,其形式是字母E(大、小写都可以)跟着有符号或无符号的十进制整数。指数指出10的乘幂与E前面的直接量相乘。在整型直接量中指数值不允许是负的,否则它就可能不是真正的整数(整型直接量的指数不允许为-0,而实型直接量则允许)。

例如实型直接量98.4可以写成下列任何一种指数形式

9.84E1 98.4e0 984.0e-1 0.984E+2

注意,984e-1是不准确的。

同样,整型直接量1900可以写成

19E2 190e+1 1900E+0

但不能写成19000e-1,和1900E-0

如指数是由2个以上的数字组成,则可以使用横划线,但通常指数不会很长。指数本身不可再含有指数。

现介绍一下基数不为10的数,用一对#号把数字括在中间,而基数则在左边的#号前面

2#111#

的值为十进制数 $4+2+1=7$ 。

从2到16的任何数都可以做为基数,基数为10则用直接表示法。如果基数大于10则用A到F来表示10到15的数字。如:

14#ABC#

等于十进制数 $10 \times 14^3 + 11 \times 14^2 + 12 = 2126$ 。

带基数的直接量也可以有指数。但要注意,指数给出的不再是10的乘幂(除非基数是10)。指数本身是以十进制数字来表示的,如:

16#A#E2

等于 $10 \times 16^3 = 2560$

2#11#E11

等于 $3 \times 2^{11} = 6144$

实数也可采用带基表示,与整数带基表示的区别在于基的实数含有小数点,因此:

2#101.11#

等于 $4+1+\frac{1}{2}+\frac{1}{4}=5.75$

7#3.0#e-1

等于 $\frac{3}{7}=0.428571$

读者可能认为带基的数过于复杂,其实并不然。带基的数是很有用的,尤其对于定点类型,因为它能使程序员按照自己的想法随意地表示数值。当然,基数为2,8和16是最常用的。

练习 3.4

1. 下列哪些不是合法的直接量? 为什么? 对于合法的直接量, 指出它们是整数还是实数。
- a) 38.8 c) 2#1011 i) 16#FF#
b).5 f) 2.71828_18285 j) 27.4e_2
c) 32e2 g) 12#ABC# k) 1_0#1_0#E1_0
d) 32e-2 h) E+8 l) 2#11#1e-1
2. 下列表示的值是什么?
- a) 16#E#E1 c) 16#F.FF#E+2
b) 2#11#E11 d) 2#1.1111_1111_1111#E11
3. 能用多少种不同的方法来描述下列数? (不考虑横划线、E 和 e 的区别、在前面加 0 和在指数前面加十的情况)。
- a) 整型 41 b) 整型 150



3.5 注释

适当的注释有助于其他人乃至编程者本人阅读程序, 对提高程序的可读性尤为重要。在第 2 章中我们曾遇到过注释。

Ada 的注释是两个横划线之后的文字, 如:

--这是注释

注释可到行末, 在 Ada 中不可以在行中插入注释。注释可独占一行, 也可以占若干行。如:

--这个注释

--需要

--好几

--行

注释前面一定要加两个横划线, 而且这两个横划线之间不能有空格。

练习 3.5

1. 下列各行中有多少个词法元素?

- a) X := X + 2; --X 加 2 再赋给 X
b) --这是一个无意义注释
c) -----
d) - - - - - - -

2. 区分

- a) delay 2.0; b) delay2.0;

要点 3

- 除非在字符串和字符直接量中, 否则字母的大、小写无区别。
- 横划线在标识符中有意义, 而在数字中则无意义。
- 除非在字符串、字符直接量和注释中, 否则空格不许出现在单个词法元素的中间。
- 用有无小数点来区分是实型还是整型直接量。
- 整型不能有负指数。
- 数值直接量不能加正负号。
- 在数值直接量前面可以加无效的零。

第 4 章 纯量类型

本章介绍 Ada 程序的基础知识,我们将讨论对象说明和赋值,以及可见性和作用域的概念。然后我们还要引入重要的类型,子类型和约束的概念。以类型为例,要讨论数值类型 INTEGER 和 REAL,一般的枚举和特殊的布尔类型以及在这些类型上的操作。

4.1 对象说明和赋值

值可以存贮在被说明为特定类型的对象中,对象可能是变量,也可能是常量。所谓变量是指在程序执行时其值是变化的(或称改变)。而常量则是指在其整个生命周期中始终保持其初始值不变。变量经说明而被引入到程序中,变量的说明形式是变量的名字(标识符)后跟一个冒号,然后是类型名。也可选择使用 := 符号和初值来对被说明的变量赋值。分号表示说明结束。例如:

```
I:INTEGER;  
P:INTEGER := 38;
```

这里引入了整型变量 I,但未赋初始值。还引入了一个初值为 38 的整型变量 P。我们可一次对多个变量加以说明,各个变量用逗号分开:

```
I,J,K:INTEGER;  
P,Q,R:INTEGER := 38;
```

第二个式子中,所有 P,Q,R 的初始值都是 38。

如果在说明变量时,没有赋初始值,则必须非常小心,不能在变量赋值前使用它。否则程序执行是不可控的,这样的程序是错误的。在第 2 章中曾提到过,尽管这样的程序不合法,但编译和运行时系统不会向我们指出这种错误。

给变量赋值通常是用赋值语句,即变量的标识符后跟 :=,然后给出新值的表达式,语句末尾要有分号。

```
I := 38;
```

和

```
P := Q + R;
```

这两个赋值语句都是合法的,给 I 和 P 赋予新值,而冲掉了它们的原值。

注意, := 后面跟与被赋值变量类型相同的任意表达式,关于表达式的规定,我们以后再讨论。这里我们只认为表达式包括变量、常数、运算符(如“+”)和括号等,如同一般的数学表达式。

说明中的赋初始值和赋值语句很相似,两者在表达式前都有 :=,并且表达式都可能很复杂。

两者的主要区别在于说明中可以同时说明几个变量,并给他们赋同一初始值,但不能在一个赋值语句中给几个变量同时赋值。

要注意重复说明,

A,B;INTEGER := E;

实际上是下式的缩写,

A;INTEGER := E;

B;INTEGER := E;

这意味着计算表达式 E 的值并赋给几个变量。这样既精炼,亦可行,但是以后会遇到一些由此造成影响的例子。

常量的说明与变量的说明相似,但要在冒号后插入一个保留字 Constant。当然,常量应在说明时赋值。否则该常量是无用的(以后要提到一个例外情况)。例如:

PI;constant REAL := 3.14159_28536;

当且仅当在数值类型中,常量说明中的类型可以省略,如:

PI;constant := 3.14159_28536;

技术上称其为数据说明,它只为数据提供一个名字。有名数据是整数还是实数是靠初始值的形式来区分的。上例是实数,因为有小数点。在说明数值常量时,最好省略类型,以后可看到这样做的原因。在以后的例子中,我们也将这样做。但在数值变量说明时,即使有初始值,也不能省略类型。

常量说明(有类型)与数据说明(没有类型)是有区别的,前者初始值可以是任何表达式,当运行时遇到说明则赋值。而后者是静态表达式,在编译时赋值。更详细介绍见第 12 章。

练习 4.1

1. 说明实型变量 R, 并给出它的初始值。
2. 写出正确的实型常量为 0 和 1 的说明。
3. 在下列说明和语句中有什么错误?

- a) var I;INTEGER;
- b) G;constant := #1
- c) P,Q;constant INTEGER;
- d) P := Q := 7;
- e) MN;constant INTEGER := M * N;
- f) 2PI;constant := 2.0 * PI;

4.2 分程序和作用域

只有在说明中才引入标识符,而语句仅使用标识符而不引入它,对这一点 Ada 给予严格规定。也就是说,语句使用的标识符,只可由在其之前的说明中引入。说明和语句处在程序文本中的不同地方。包括说明和语句的最简单的文本形式是分程序。

一个分程序以保留字 declare 开始,接下来为一些说明而后是 begin,其后为一些语句。最后以保留字 end 和终结分号结束。例如:

```
declare
  I;INTEGER := 0;      -- 这是说明
begin
  I := I + 1;          -- 这是语句
end;
```

一个分程序本身也可被视为一个语句。因此，一个分程序体内的语句可能是另一个分程序。分程序的嵌套是无限的。

因为分程序是一个语句，它可象其它语句一样地被执行。当执行一个分程序时，按顺序地确立说明部分的各个说明（在 declare 与 begin 之间的部分），然后执行分程序体内的语句（在 begin 与 end 之间）。注意术语：确立说明、执行语句。说明的确立是使被说明的对象成为实体，从而有意义，并可给它赋初值。当遇到分程序的 end 时所有该分程序里被说明的实体便自行消失。上例的分程序看来是太蠢了，它引入了 I 并给它加 1，但在用其结果之前却又消失了。

另外，对象可作为初始值，它们可以和被赋值的对象在同一说明部分中被说明，但在使用它们之前必须先进行说明。例如：

```
declare
  I;INTEGER := 0;
  K;INTEGER := I;
begin
```

是合法的，但是

```
declare
  K;INTEGER := I;
  I;INTEGER := 0;
begin
```

通常是不允许的（以后我们将看到，也有允许这样的情况，但意义不同）。按顺序确立说明的概念是非常重要的（术语是：说明的线性确立）。

与其它的模块化结构语言一样，Ada 也有隐蔽概念。试看：

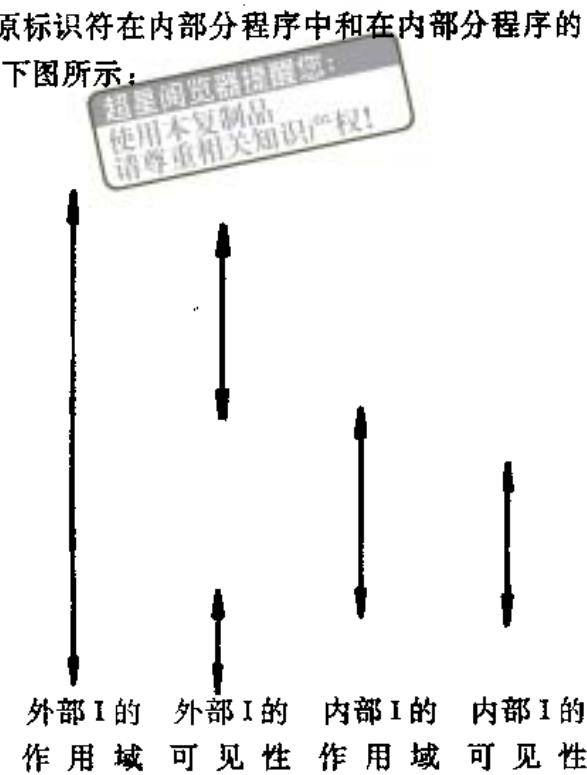
```
declare
  I,J;INTEGER;
begin
  ...
  -- 这里的 I 是外层的
  declare
    I;INTEGER;
  begin
    ...
    -- 这里的 I 是内层的
  end;
  ...
  -- 这里的 I 是外层的
end;
```

在这里，变量 I 在外部模块中被说明，然后在内部模块中又被说明。重新说明不会造成外部的 I 消失，而只是使它暂时不可见，内部的 I 指的是新 I，当我们一离开内部模块，新 I 就消失了，而外部的 I 就再现了。

名词“作用域”和“可见性”是有区别的。作用域是程序的一个区域，在这个区域里对象是潜在可见的，而如果在某一点标识符代表着该对象，则该对象在这一点即为可见。下面用图来解释，第 8 章我们再做全面的论述。

在分程序中,变量(常量)的作用域是从分程序的说明开始一直到分程序的结尾。但是在任何的内部分程序中重新说明了相同的标识符后,原标识符在内部分程序中和在内部分程序的说明处均是不可见的。作用域和可见性的区域如下图所示:

```
declare  
  I:INTEGER := 0;  
begin  
  ...  
  declare  
    K:INTEGER := I;  
    I:INTEGER := 0;  
  begin  
  ...  
end;  
...
```



变量 K 的初始值是外层分程序的 I,因为它在引入内部分程序的变量 I 之前就已经存在。

因此

```
K:INTEGER := I;  
I:INTEGER := 0;
```

是否合法取决于它所处的环境。

练习 4.2

1. 在下列程序中你能找到多少个错误?

```
declare  
  I:INTEGER := 7;  
  J,K,INTEGER;  
begin  
  J := I + K;  
  declare  
    P,INTEGER := I;  
    I,J,INTEGER;  
  begin  
    I := P + Q;  
    J := P - Q;  
    K := I * J;  
  end;  
  PUT(K);      --输出 K 的值  
end;
```

4.3 类型

“类型是指值域和在此值域上的操作集”(LRM3.3)。

对于类型 INTEGER, 其值域为:

..., -3, -2, -1, 0, 1, 2, 3, ...

而操作集为 +, -, *, 等。

除两种例外情况(数组和任务, 我们以后将讨论), 每种类型都有其自己的名, 类型名在定义时引入(内部类型如 INTEGER 被认为在程序包 STANDARD 中被定义)。而每个类型定义都唯一的引入一个新类型。

分属两个不同类型的值域一般彼此是不同的, 不排除在某些情况下, 它们的实际值在词法形式上相同。此时值的意义要根据上、下文来确定。用一个词法形式来表示两个或多个不同的事物, 称为重载。

一种类型的值不能赋给其它类型的变量, 这是强类型的基本规则。这很有助于快速地开发正确的程序, 因为这样就可使许多错误在编译时就被发现。

类型定义与对象说明采用不同的词法形式, 其目的在于强调它们不是同一概念。类型定义由保留字 type、与类型有关的标识符、保留字 is, 然后是类型的定义、最后是分号所组成。譬如程序包 STANDARD 包含的类型定义有:

```
type INTEGER is ...;
```

在 is 与 ";" 之间的类型定义, 是采用某种方法给出属于这个类型的值域。如下面这个具体例子:

```
type COLOUR is (RED, AMBER, GREEN);
```

这是一枚举类型的例子, 在本章的后面几节中还要详细介绍。

在此引入了新的类型 COLOUR, 而且指出这种类型有 3 个值, 用标识符 RED, AMBER 和 GREEN 加以表示。

可以按通常的方法来说明这个类型的对象

```
C; COLOUR;
```

还可给对象赋初值

```
C; COLOUR := RED;
```

或说明一个常量

```
DEFAULT; constant COLOUR := RED;
```

我们曾指出一种类型的值不能赋给其它类型的变量。因此不能把颜色和数相混,

```
I; INTEGER;
```

```
C; COLOUR;
```

...

```
I := C;
```

是不合法的。在较早的语言中, 需借助概念的扩展, 如用整型, 把值 0, 1, 2 赋给有名变量 RED, AMBER 和 GREEN 来实现枚举类型。在 Algol 60 中可以写成:

```
integer RED, AMBER, GREEN;
```

RED := 0; AMBER := 1; GREEN := 2; 然后使用 RED, AMBER 和 GREEN。显然这要比直接用代码值 0, 1, 2 容易理解。可是, 编译程序不能检查出把表示颜色的有名变量赋给程序员认为是普通整型变量的错误。而在 Ada 中, 正如我们看到的, 这种错误会被编译程序发现。

4.4 子类型

我们现在介绍子类型和约束。一个子类型, 顾名思义, 它所代表的一组值是基本类型值域的子集。通过约束(又称限制)来定义这个子集。根据基本类型的类别, 约束有多种格式。子集也可以是全集。然而, 由于无法对基本类型的操作集加以限制, 所以子集的概念只适用于值, 而基本类型的所有操作对子类型均有效。

例如我们打算对日期进行运算, 我们知道一个月的范围只可能为 1..31, 因此我们说明子类型:

```
subtype DAY_NUMBER is INTEGER range 1..31;
```

现在我们可用此子类型标识符来说明变量和常量。

```
D:DAY_NUMBER;
```

变量 D 此时只能在整型 1 到 31 中取值。如果需要, 编译程序可插入运行检查, 以防止变量 D 的值超出范围。如果检查发现超出范围, 则引发 CONSTRAINT_ERROR 异常。

要认识到子类型的说明并没有引入新的不同类型。对象如 D 的类型仍然是 INTEGER, 因此, 从语法上讲下例是完全合法的:

```
D:DAY_NUMBER;
```

```
I,INTEGER,
```

```
...
```

```
D := I;
```

当然, 在执行时, I 的值应在范围 1..31。如果 I 值在此范围内, 则没有问题。如果不在则引发 CONSTRAINT_ERROR。反向赋值

```
I := D;
```

始终是允许的。

为了进行约束, 有时不一定需要明显地引入子类型, 我们可以写成

```
D:INTEGER range 1..31;
```

反过来, 子类型不一定就要产生约束的效果, 因此, 下列的写法是完全合法的:

```
subtype DAY_NUMBER is INTEGER;
```

显然, 在这个例子中子类型的引入没有什么价值。

可以用以前定义过的子类型来定义另一个子类型(显式变成隐式):

```
subtype FEB_DAY is DAY_NUMBER range 1..29;
```

任何附加的约束必须符合现存的约束。

```
DAY_NUMBER range 0..10;
```

是不正确的, 它会引起 CONSTRAINT_ERROR。

上面的例子说明了带有静态范围的约束。通常范围亦能够由任意的表达式给出。因此, 子类型的值域未必非得是静态的不可, 而类型的则始终是静态的。

总之,子类型不引入新的类型,而只是用约束对现有类型的值域进行压缩。在后面几章中,我们将会遇到由于若干原因,不准许进行显型约束的情况,在这种情况下不得不引入子类型。我们把类型或子类型的名称之为类型标记,把由类型标记后跟可选约束而构成的形式称为子类型表示。

使用子类型有两个明显的优点,一是通过阻止给变量指定不合适的值而尽早地检查出程序设计的错误;二是提高了程序的运行效率。以后我们将看到子类型还适用于数组的下标。

至此我们已介绍了类型和子类型的基本概念,下面几节我们通过讨论 Ada 简单类型的详细特点,来进一步说明这些概念。

4.5 简单的数值类型

全面地介绍 Ada 的数值类型要到本书的末尾。数值分析问题(误差估算等)是非常复杂的,Ada 在这方面也相应地复杂,以便能完全满足数学家的要求。就眼前而言,可以不考虑这些复杂的东西。相应地在本节中,我们只考虑两种数值类型 INTEGER 和 REAL,以及对整型和实型的简单理解。通常对于程序员来说,这两种数值类型就足够了。

首先要说明,类型 INTEGER 是真正的 Ada 内部类型。而类型 REAL 则不是。它是由内部浮点类型来说明的,这样做的原因是考虑到可移植性,对此将在揭示了数值类型的实质后讨论。我们暂且把 REAL 看成为浮点类型。

如我们所看到的,用保留字 range 可在类型 INTEGER 上加约束。它后面跟着被 2 个点分开的表达式,当然表达式必须能产生整型值,如:

```
P:INTEGER range 1..I+J;
```

在上例中,如果 I+J 的结果为 0,则范围为空,空范围是没有用的。但在受限的情况下,经常会自动地出现这种情况。当出现了这种情况,则只能采用特别的方法,才能排除它。

INTEGER'FIRST 给出了 INTEGER 类型的最小值,INTEGER'LAST 给出了最大值。这是第一个关于属性的例子,Ada 具有多种属性,由一个单引号后面跟标识符来表示。

INTEGER'FIRST 的值取决于实现,但总是负的。在二进制补码机器上,它是-INTEGER'LAST-1,而在二进制反码机器上,它是-INTEGER'LAST。在典型的十六位二进制补码机器中。

```
INTEGER'FIRST = -32768
```

```
INTEGER'LAST = +32767
```

当然,我们最好写 INTEGER'LAST 而不是+32767。否则就会影响程序的可移植性。

二个有用的子类型是:

```
subtype NATURAL is INTEGER range 0..INTEGER'LAST;
```

```
subtype POSITIVE is INTEGER range 1..INTEGER'LAST;
```

由于它们非常有用,因此在 STANDARD 程序包中已事先为我们作了说明。

属性 FIRST 和 LAST 也可以用于子类型。

```
POSITIVE'FIRST = 1
```

```
NATURAL'LAST = INTEGER'LAST
```

顺便简要地讨论一下类型 REAL。可以把约束用于类型 REAL,以便缩小范围和降低精度。这需要涉及到以后讨论的内容,同样也有属性 REAL'FIRST 和 REAL'LAST,不再赘述。

在 INTEGER 和 REAL 类型上预定义的运算与现代编程语言相同,概述如下:

十，一 为一元运算(只有一个操作数)或为二元运算(有二个操作数)。作为一元运算符，操作数可以是整型也可以是实型，结果与操作数类型相同。一元“十”不起任何作用，而一元“一”则改变符号。

作为二元运算符，二个操作数必须都是整型或都是实型。结果与操作数类型相同，二元运算实现通常的加减法。

* 乘法，二个操作数必须都是整型或都是实型且获得同类型的结果。

/ 除法，二个运算数都必须是整型或都是实型，且获得同类型的结果。整数除法向零取舍。

rem 取余，二个操作数都必须是整型，结果也是整型，结果为除法所得的余数。

mod 取模，二个操作数都必须是整型，结果也是整型，取模实现算术求模运算。

abs 取绝对值，是一元运算，单个操作数，可以是整型也可以是实型。结果是与操作数同类型的绝对值。

* * 指数运算，第二个操作数为指数，如果第一个操作数是整型，则第二个操作数必须是正整数或是 0。如果第一个操作数是实型，则第二个操作数可以是任意整型。结果与第一个操作数同类型。

另外，可以进行 =, /=, <, <=, > 和 >= 的关系运算，结果为布尔值 TRUE 或 FALSE。二个操作数都必须是同类型，要注意不等式运算符是 /=。

虽然上述运算都很简单，但有几点要注意。在一般情况下，不准许混合类型运算。不能把整型与实型相加，但可以用相应的类型名(实际上是子类型名)后跟带括号的表达式来把整型转变为实型，或把实型转变为整型。

因此

I, INTEGER : = 3;

R, REAL : = 5.6;

我们不能写成

I + R

但可以写成

REAL(I) + R

用实型相加得到实型结果 8.6 或

I + INTEGER(R)

用整型相加得到整型结果 9。

实型转变为整型时，是 4 舍 5 入，而不是舍掉小数点后的部分。

1.4 变成为 1

1.8 变成为 2

rem 和 mod 之间有明显区别。rem 运算得到的是整数相除后的余数，整数相除后结果的符号取决于两操作数的符号。因而有：

$$7 / 3 = 2$$

$$(-7) / 3 = -2$$

$$7 / (-3) = -2$$

$$(-7)/(-3)=2$$

相应的余数为：

$$\begin{aligned}7 \bmod 3 &= 1 \\(-7) \bmod 3 &= -1\end{aligned}$$

$$\begin{aligned}7 \bmod (-3) &= 1 \\(-7) \bmod (-3) &= -1\end{aligned}$$

余数和商数有下列关系：

$$(I/J) * J + I \bmod J = I$$

余数的符号总是与第一个运算数 I 的符号相同。

然而, rem 并不能满足需要, 让我们来看一下 I rem J, 当 J 的值固定(假设为 5)则对于正、负 I 的值, 我们可得到如图 4.1 的图形:

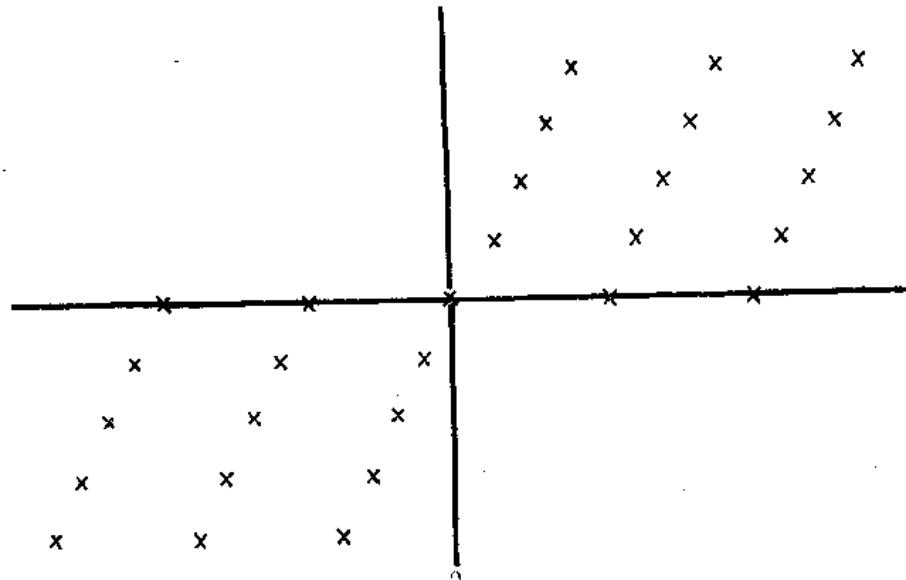


图 4.1 零周围 $I \bmod 5$ 的结果

可以看到, 图形是以零对称的。当大于零后就顺序地变为增值。

mod 运算, 则都为增长趋势。如图 4.2;

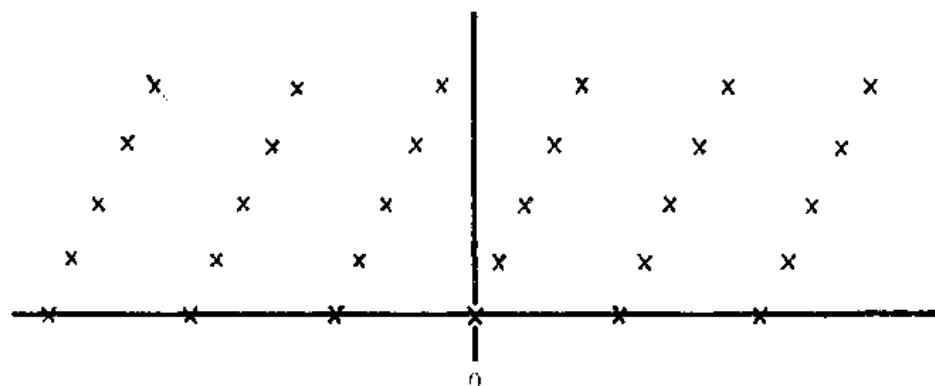


图 4.2 零周围 $1 \bmod 5$ 的结果

mod 运算可使我们进行通常的算术求模,对所有正或负的 A 和 B,有:

$$(A+B) \bmod n = (A \bmod n + B \bmod n) \bmod n$$

对正数 n,A mod n 总是在范围 0..n-1 之间;对负数 n,A mod n 总是在范围 n+1..0 之间。当然模运算通常只对正数 n 进行,但是 mod 也适合于 n 为负值的运算。因此

$$7 \bmod 3 = 1$$

$$(-7) \bmod 3 = 2$$

$$(-7) \bmod (-3) = -2$$

$$(-7) \bmod (-3) = -1$$

mod 运算的结果符号总是与第二个操作数一致,而 rem 与第一个操作数一致。

读者也许会感到,讨论太深入了。总之,用负数为操作数的整数除法是不多见的。

最后要注意求幂运算 **。如果二个操作数均为正数,则运算是相应的累乘

$$3^{**} 4 = 3 * 3 * 3 * 3 = 81$$

$$3.0^{**} 4 = 3.0 * 3.0 * 3.0 * 3.0 = 81.0$$

第二个操作数可以是 0,这时结果为 1。

$$3^{**} 0 = 1$$

$$3.0^{**} 0 = 1.0$$

$$0^{**} 0 = 1$$

$$0.0^{**} 0 = 1.0$$

如果第一个操作数是整数,则第二个操作数不能是负数,否则结果可能不是整数。实际上,这种情况发生时,会引发异常 CONSTRAINT_ERROR。但是第一个操作数为实数时,则准许第二个操作数为负数,生成相应的倒数

$$3.0^{**} (-4) = 1.0 / 81.0 = 0.0123456780123\dots$$

作为本节结尾让我们来简要地讨论一下,表达式中的混合运算。通常各种运算有其不同的优先级,用括号可以改变自然优先级。相同优先级的运算是按从左向右的顺序进行的。在括号中的表达式,在使用前要先计算其值。但要注意,二元运算中的二个操作数的求值顺序不确定。现将至此为止我们已讨论过的运算符的优先级以递增顺序列出如下:

= /= < <= > >=

+ - (二元)

+ - (一元)

* / mod rem

** abs

如

A/B * C 为 $(A/B) * C$

A+B * C+D 为 $A+(B * C)+D$

$A * B + C * D$ 为 $(A * B) + (C * D)$

$A * B * * C$ 为 $A * (B * * C)$

如上所示,相同优先级的多个运算则是从左到右进行,无需加括号。但是语法限定,如果进行多个乘幂运算。我们不能写成

$A * * B * * C$

必须明确地写成

$(A * * B) * * C$ 或 $A * * (B * * C)$

这个限定可以防止意外的书写错误。但是下式

$A - B - C$ 和 $A / B / C$

是准许的。语法规则还禁止在没有括号时把 abs 和 * * 混用。

要注意一元负运算的优先级:

$-A * * B$ 为 $-(A * * B)$ 而不是 $(-A) * * B$

如同在 Algol 68 中

$A * * -B$ 和 $A * -B$

是非法的,需要加括号。

最后要注意 abs 的优先级,与一元的负运算不同,容易混淆。我们可以写成。

$-abs X$ 而不能写成 $abs -X$

练习 4.6

1. 计算下列表达式的值

I,INTEGER := 7;

J,INTEGER := -5;

K,INTEGER := 3;

a) I * J * K e) J + 2 rem I

b) I / J * K f) K * * K * * K

c) I / J / K g) -J mod 3

d) J + 2 mod I h) -J mod 3

2. 用 Ada 写出下列数学表达式,用相应的类型标识符。

a) $b^4 - 4ac$ b) $4/\pi r^2$

c) $\rho \pi a^4 / 3t\eta$

4.6 枚举类型

先让我们来看一些枚举类型定义的例子:

```
type COLOUR is(RED,AMBER,GREEN);
```

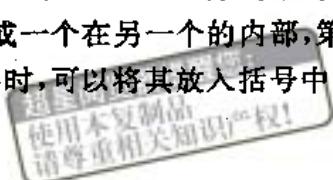
```
type DAY is(MON,TUE,WED,THU,FRI,SAT,SUN);
```

```
type STONE is(AMBER,BERYL,QUARTZ);
```

```
type GROOM is(TINKER,TAILOR,SOLDIER,SAILOR,RICH__MAN,POOR__MAN,
               BEGGER__MAN,THIEF);
```

```
type SOLO is (ALONE);
```

这里引入一个重载的例子，直接量 AMBER，既可表示一个 COLOUR，亦可表示一个 STONE。两个直接量用同一个名，不论它们是在同一说明部分或一个在另一个的内部，第二个说明都不覆盖第一个，通常根据上、下文来决定它的意思。必要时，可以将其放入括号中，在前面加上类型名（实际上是子类型名）和单引号以示区别。



COLOUR'(AMBER)

STONE'(AMBER)

该例子以后还会出现。

虽然我们能用 AMBER 在 2 个不同枚举类型中作为枚举直接量，但不允许枚举直接量与变量标识符重名。下面我们会看到用子程序可使枚举直接量重载。

在枚举类型中，值的个数可无上界，但至少必须有一个值，即不准许空的枚举类型。

枚举类型和子类型的约束与整型的一样。约束的格式为：

range 下界表达式..上界表达式

这表示了从下界到上界的值域，因此我们可以写为：

subtype WEEKDAY is DAY range MON..FRI;

D:WEEKDAY;

或

D:DAY range MON..FRI;

则我们知道 D 不能是 SAT 和 SUN。

当下界超过上界时，相应的取值范围为空。如：

subtype COLOURLESS is COLOUR range AMBER..RED;

注意：有一例外，我们不能有类型如 SOLO 的空子类型。

属性 FIRST 和 LAST 也可用于枚举类型和子类型。

COLOUR'FIRST=RED

WEEKDAY'LAST=FRI

内部属性 SUCC 和 PRED 可给出一个枚举值的后继和前驱。SUCC 和 PRED 跟在类型名和单引号后面。

COLOUR'SUCC(AMBER)=GREEN

STONE'SUCC(AMBER)=BERYL

DAY'PRED(FRI)=THU

括号内必须是类型正确的任意表达式。如果我们求第一个值的前驱和最后一个值的后继，则引起异常 CONSTRAINT_ERROR。对任何类型 T 的任何值 X，如若不会产生异常，定有

T'SUCC(T'PRED(X))=X

反之亦然。

另一个属性是 POS，它给出了枚举值在说明中的位置。第一个值的位置为 0。

COLOUR'POS(RED)=0

COLOUR'POS(AMBER)=1

COLOUR'POS(GREEN)=2



POS 的反属性是 VAL。它根据位置得到相应的枚举值。

COLOUR'VAL(0)=RED

DAY'VAL(8)=SUN

如果位置在范围之外,例如:

SOL'VAL(1)

就会引起 CONSTRAINT_ERROR。

很清楚,我们总有:

T'VAL(T'POS(X))=X

反之亦然。

我们还要注意

T'SUCC(X)=T'VAL(T'POS(X)+1)

它们同时给出相同的值,或同时引起异常。

应注意,SUCC,PRED,POS 和 VAL 这四个属性亦可用于子类型,其含义与相应的基本类型的相同属性一致。

应尽量避免把 POS 和 VAL 滥用。最后,运算符 =,/=,<,<=,>,>= 也可用于枚举类型,其结果是由类型说明中的值的顺序来确定的. RED < GREEN 是 TRUE

WED>=THU 是 FALSE 进行位置比较,也能得到相同结果。

T'POS(X)<T'POS(Y) 和 X<Y

总是相等的(除非 X<Y 有二义性)。

练习 4.6

1. 求值

- a) DAY'SUCC(WEEKDAY'LAST)
- b) WEEKDAY'SUCC(WEEKDAY'LAST)
- c) STONE'POS(QUARTZ)

2. 写出适当的枚举类型说明

- a) 彩虹的彩色
- b) 典型水果

3. 写出如谁吃了藏有石头的饼则谁就是新郎的表达式,用本节开头所述的类型 GROOM。

4. D 是类型 DAY, 月的第一天的星期在 D 中. 写一个赋值语句, 将该月第 N 天的星期赋给 D.

5. 为什么 X<Y 可能有二义性?

4.7 布尔类型

布尔类型可以被看作如下说明的枚举类型

type BOOLEAN is(FALSE'TRUE);

布尔值可以用于结构中, 如我们在第2章见过的 if 语句, 由运算符 =, /=, <, <=, > 和 >= 对类型实施运算得到布尔值。因此我们可以写出下列结构:

```
if TODAY=SUN then
    TOMORROW:=MON;
else
    TOMORROW:=DAY'SUCC(TODAY);
```

end if;

布尔类型(记念数学家布尔)具有所有枚举类型的性质,例如:

FALSE < TRUE = TRUE

BOOLEAN'POS(TRUE) = 1



我们甚至可以写

subtype ALWAYS is BOOLEAN range TRUE..TRUE;

尽管看起来用处不大。

布尔类型还有其独特的运算符如下:

not 一元运算,把 TRUE 变为 FALSE,反之亦然,与 abs 有相同的优先级。

and 二元运算,如果二个操作数都是 TRUE,则结果为 TRUE。反之结果为 FALSE。

or 二元运算,如果有一个操作数是 TRUE,或二个操作数都是 TRUE,则结果为 TRUE。
如果二个操作数都是 FALSE,则结果是 FALSE。

xor 二元运算,只有在二个操作数中的一个为 TRUE,且另一个不为 TRUE 时,结果是
TRUE(因此名字是异或),即当两个操作数不同时结果是 TRUE。

and, or 和 xor 的结果以下列真值表所示:

and	F	T	or	F	T	xor	F	T
F	F	F	F	F	T	F	F	T
T	F	T	T	T	T	T	T	F

图4.3 and, or 和 xor 的真值表

and, or 和 xor 的优先相同,但比其它所有运算符的优先级却要低。实际上,比关系运算
=, /=, <, <=, > 和 >= 的优先级低。不象 Pascal,在表达式中不需要括号。

P < Q and I = J

然而,虽然优先级相同,但 and, or 和 xor 在不加括号的表达式中不能混用(与 +、- 不一
样),因此

B and C or D 是不合法的

而

I + J - K 是合法的

我们必须写成

B and (C or D) 或 (B and C) or D

对其含义加以强调。

熟悉其它程序语言的读者,会记得,一般 and 和 or 的优先级是不同的,但这常使程序员搞
混而写错程序。为了避免这类错误,Ada 给 and 和 or 以相同的优先级,并且坚持在联用时要加

括号,当然使用相同运算符时可省却括号,如:

B and C and D 是合法的

通常计算是从左至右,即便不按此顺序亦无关,因为运算符 and 是可结合的。

要注意 not,象在其它语言中一样,它的优先级比 and, or 和 xor 要高,因此

not A or B

意味着

(not A) or B

而不是

not (A or B)

对逻辑熟悉的话,不难知道它如同于

(not A) and (not B)

可按通常方法来说明和操作布尔变量和常量。

DANGER: BOOLEAN;

SIGNAL: COLOUR;

...

DANGER:=SIGNAL=RED;

当信号是 RED 时,变量 DANGER 为 TRUE。此时,我们则可以写成:

if DANGER then

 STOP_TRAIN;

end if;

注意:我们不必要写成

if DANGER=TRUE then

虽然这种写法也正确,但它忽略了 DANGER 已是布尔数,可直接用作为条件。

最糟的是写成

if SIGNAL=RED then

 DANGER=TRUE;

else

 DANGER=FALSE;

end if;

而不是写成

DANGER:=SIGNAL=RED;

直接量 TRUE 和 FALSE,经如下说明可重载:

type ANSWER is (FALSE,DONT_KNOW,TRUE);

但这样做有可能使程序混乱。

练习 4.7

1. 写出常量 T 和 F 具有初值 TRUE 和 FALSE 的说明。

2. 用上题的 T 和 F,计算

- a) $T \text{ and } F \text{ and } T$ d) $(F=F)=(F=F)$
 b) $\text{not } T \text{ or } T$ e) $F=F=F=F$
 c) $T < T < T < T$
 3. 对变量 A 和 B 的代入所有的布尔值计算
 $(A /= B) = (A \text{ or } B)$

超星浏览器提醒您：
 使用本复制品
 请尊重相关知识产权！

4.8 类型分类

现在我们来巩固一下到目前为止所学的本章内容。

Ada 中的类型分类如图4.4所示

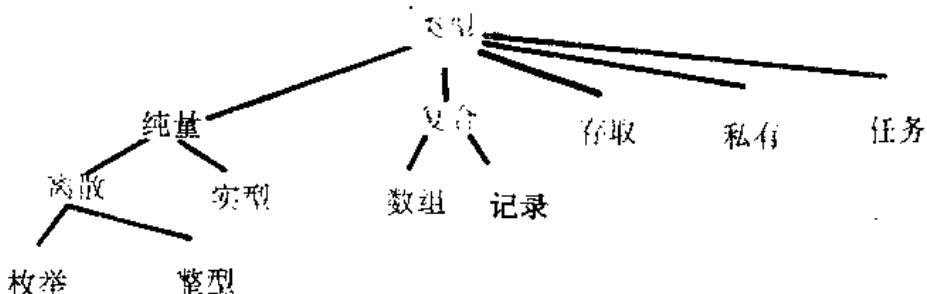


图4.4 类型的分类

本章讨论了纯量类型。在第6章我们将讨论复合类型，以后我们还将讨论存取类型和私有类型以及任务类型。

纯量类型本身可以分为实型和离散型两大类。至此讨论的实型的唯一例子是类型 REAL，在第12章将讨论其它的实型。其它类型如 INTEGER, BOOLEAN 和枚举类型都是离散类型。而字符类型实际上亦是枚举类型的一种形式，在第6章中介绍。

离散类型和实型的主要抽象区别是，前者有一组明确的可分离的各不相同的（离散）的值，而类型 REAL 的值域是连续的。我们知道，实际上有限位长的数字计算机把实数处理成离散的值，但这是具体实现方法，抽象概念仍把实数看成连续值。

属性 POS, VAL, SUCC 和 PRED 反映了值的离散性质，故它们适用于所有离散类型（和子类型）。在4.8节我们曾用枚举类型解释过这些属性的含义，在 INTEGER 类型中，位置号恰好是数本身。

$\text{INTEGER}'\text{POS}(N)=N$
 $\text{INTEGER}'\text{VAL}(N)=N$
 $\text{INTEGER}'\text{SUCC}(N)=N+1$
 $\text{INTEGER}'\text{PRED}(N)=N-1$

乍看起来把属性用于整型是相当无价值的。但当我们介绍到13章类属设施概念后，我们将看到，允许把属性用于所有的离散类型，会给我们带来很大的便利。

属性 FIRST 和 LAST 可用于所有的纯量类型和子类型，其中包括实型。我们再次强调，POS, VAL, SUCC 和 PRED 对子类型和对相应的基本类型的运算是一致的，但是 FIRST 和 LAST 则不同。

最后我们要注意类型转换和类型限定之间的区别。

REAL(I) —— 转变

INTEGER'(I) —— 限定

转换是转变类型,而限定只是进行强调(一般是为了克服二义性)。为了有助于区别,使用了引号。

对于上述这两种情况,我们都能用子类型名,但可能会引起 CONSTRAINT_ERROR 异常
POSITIVE(R)

把实型变量 R 的值转变为整数,并且测试其是不是正数,反之

POSITIVE'(I)

只是测试变量 I 的值是否为正,显然结果为检测所得结论。这些测试不能单独存在,可将其用于表达式中。

4.9 表达式

图4.5给出了到目前为止所见到过的所有运算符,按优先级分组,除了 ** 运算符,所有的二元运算,二个操作数都必须是相同类型。

实际上我们已介绍了除(&)之外所有的 Ada 的运算符。在第8章我们可以看到加法的更深层的含义。

对于所有的纯量类型有二种测试,即 in 和 not in。虽然它们的优先及与关系运算符 =, / = 等相同,但从技术上讲,它们不是运算符。它使我们可以测试某个值是不是在指定的范围(包括终值)内,或是否满足子类型的约束。它有两个操作数,第一个操作数是纯量表达式,第二个是范围或类型标志,结果自然是布尔类型。例如:

```
I not in 1..10  
I in POSITIVE  
TODAY in WEEKDAY
```

注意,对于最后一种情况,我们只能用类型标志而不能用子类型指示,即不能把上例写为

```
TODAY in DAY range MON..FRI
```

但我们可以写为

```
TODAY in MON..FRI
```

在这里 Ada 似乎很古怪。

测试 not in 等于先用 in,然后对结果用 not,但 not in 通俗易读,因此上面第一个表达式可以写成

```
not(I in 1..10)
```

这里需要加括号。

我们将在第7章介绍子程序时再来解释,为什么从技术上讲 in 和 not in 不是运算符。

有2个短路控制格式, and then 和 or else,也与 in 和 not in 一样从技术上讲不是运算符。

格式 and then 与运算符 and 很相象, or else 与运算符 or 很相象,它们可以出现在表达式中,与 and, or 和 xor 有相同的优先级,不同之处在于对操作数的计算规则。

在使用 and 和 or 时,二个操作数都进行计算,但没有指定顺序,而使用 and then 和 or

`else` 时, 总是先计算左边的操作数。只有当决定结果必须用到右操作数时, 才计算右操作数。
因此

X and then Y

的操作是先计算 X 的值, 如果 X 为假, 则不论 Y 的值是什么, 结果为假, 这时不计算 Y 的值。如果 X 是真, 则必须要计算 Y 的值, 并且 Y 的值即为结果。

运算符	运 算	操 作 数	结 果
and or xor	与 或 异或	BOOLEAN BOOLEAN BOOLEAN	BOOLEAN BOOLEAN BOOLEAN
= /= < <= > >=	相等 不相等 小于 小于或等于 大于 大于或等于	任何量 任何量 纯量 纯量 纯量 纯量	BOOLEAN BOOLEAN BOOLEAN BOOLEAN BOOLEAN BOOLEAN
+	加	数值	相同
-	减	数值	相同
+	正	数值	相同
-	负	数值	相同
*	乘	INTEGER	INTEGER
/	除	REAL	REAL
mod	模	INTEGER	INTEGER
rem	取余	REAL	REAL
		INTEGER	INTEGER
		INTEGER	INTEGER
**	乘幂	INTEGER; 非负数 INTEGER	INTEGER
not	非	REAL; INDEGER	REAL
abs	绝对值	BOOLEAN	BOOLEAN
		数值	相同

图4.5 纯量运算符

类似地

X or else Y

是先计算 X 的值,如果 X 是真,则不论 Y 的值是什么,结果为真,故不再计算 Y 的值。如果 X 是假,则必须计算 Y 的值,并且 Y 的值即为结果。

当对计算顺序有要求时,要用格式 and then 和 or else 形式。通常用第一个条件来防止第二个条件产生异常。

假设我们测试

I/J>K

并且要避免 J 是 0 的危险,我们就可以写成

J=0 and then I/J>K

于是,如果 J 是 0,我们就防止了除数为 0 的危险。读者会认为这是一个不好的例子,因为可以写成 I>K * J(假设 J 是正数),但这样就可能引起溢出。上述操作的最好例子是将其用于数组和存取类型中。

象 and 和 or 一样,在没有括号情况下,不得把 and then 和 or else 混用。

我们来总结一下已学过的组成表达式的主要成份(取决于哪种运算符的运算):

- 标识符 用于常量、数和枚举直接量
- 直接量 如 4.6, 2#101#
- 类型转换 如 INTEGER(R)
- 约束表达式 如 COLOUR'(AMBIR)
- 属性 如 INTEGER'LAST
- 函数调用 如 DAY'SUCC(TODAY)

函数的全面介绍(怎样说明和调用),留到后面几章,在这里要注意的是:调用只含一个参数的函数时,其调用形式为函数名加括号加参数,参数是任意表达式(包括函数调用)。我们假设有一个数学库,含有熟悉的函数如:

SQRT	平方根
SIN	正弦
COS	余弦
LOG	以 10 为底的对数
LN	自然对数
EXP	指数

它们的自变量为 REAL 类型,结果亦是 REAL 类型

现在我们能写出语句

```
ROOT := (-B + SQRT(B * * 2 - 4.0 * A * C)) / (2.0 * A);  
SIN2X := 2.0 * SIN(X) * COS(X);
```

最后要注意,书写要正确。当出现错误时,通常引起异常 CONSTRAINT_ERROR。这是一个用于各种违犯范围规定的异常。这里还需提一下另一个异常 NUMERIC_ERROR。在存贮表达式的结果之前计算算术表达式时有错,则会引起这个异常。一个典型例子是 0 做除数。有一个

一般用户不注意的明显区别：

INTEGER'SUCC(INTEGER'LAST)

会引起 CONSTRAINT_ERROR, 而

INTEGER'LAST +1

会引起 NUMERIC_ERROR。



在第2章中重点提到过错误, 到目前为止, 我们已遇到过两种顺序不确定的情况,

- 目标变量的计算是发生在对该变量赋值表达式的计算之前还是之后。
- 二元运算的二个操作分量的求值顺序不确定。

到第7章介绍函数时, 我们将对顺序所产生的影响加以讨论。

练习 4.8

1. 用 Ada 写出下列数学表达式

a) $2\pi \sqrt{l/g}$

b) $\frac{m_0}{\sqrt{1-v^2/c^2}}$

c) $\sqrt{2\pi n} \cdot n^{\alpha} \cdot e^{-\beta} \quad (n \text{ 为整数})$

2. 用实数 X 代替 n 重写 I(c)。

要点 4

- 说明和语句以分号作为结尾。
- 初始化时, 要象赋值语句一样用 :=。
- 在说明中为每个对象计算初值。
- 说明中使用常数而不用常量。
- 说明的确立是线性的。
- 标识符在其自身的说明中不可递归引用。
- 每个类型定义引入一个唯一的新类型。
- 子类型不是新类型, 只是对基本类型的限制。
- 类型总是静态的, 但子类型不一定。
- 运算时类型不能混合。
- 对于负的操作数, rem 和 mod 有区别。
- 指数为负只适用于实型。
- 注意一元运算优先级。
- 纯量类型不能是空的, 但子类型可以。
- 子类型的 POS, VAL, SUCC 和 PRED 与其类型一致。
- 类型与子类型的 FIRST 和 LAST 不同。
- 约束带有一个单引号而转换则没有。
- 二元运算的求值顺序不定。
- and, or 与 and then, or else 有不同之处。

第5章 控制结构

本章介绍 Ada 的三种控制结构。即 if 语句(在以前曾遇到过)、case 语句和循环语句。这三种控制结构不仅必要，而且足以使我们不用 goto 语句和标号便可写出控制流程清晰的程序。然而，因为编程的原因，Ada 实际上还保留了 goto 语句，goto 语句也在本章介绍。

三种控制结构体现了相似的括号风格。它们以保留字 if、case 或 loop 开头，并在各自的结尾处设有 end 及相应的保留字。其形如：

if	case	loop
...
end if;	end case;	end loop;

在循环语句中，保留字 loop 可以跟在以 for 或 while 开头的重复子句之后。

5.1 If 语句

最简单的 if 语句的格式是以保留字 if 开头，后跟布尔表达式和保留字 then，接下来跟一串语句，如果布尔表达式的结果为 TRUE，则执行这些语句，这些语句未尾用 end if 结束。布尔表达式可以相当复杂，if 中嵌入的语句亦可任意长。

简单的例子：

```
if HUNGRY then
    EAT;
end if;
```

这里 HUNGRY 是布尔变量，EAT 是子程序，描述 EAT(吃)的详细活动。语句 EAT 只是调用子程序(子程序将在第7章中介绍)。

执行这个 if 语句的结果是，如果变量 HUNGRY 是 TRUE，则调用子程序 EAT，反之什么也不做。不论在哪种情况下，都在执行了 if 语句后，执行跟在 if 语句后面的语句。

我们说过，在 then 和 end if 之间可以是很长的语句序列。我们可把动作分的更为详细。

```
if HUNGRY then
    COOK;
    EAT;
    WASH_UP;
end if;
```

应该注意程序的书写风格，当语句序列只有一个语句时，整个 if 语句可排列在一行上，一般 then 最好与 if 在同一行。

if $X < 0.0$ then $X := -X$; end if;

要注意 end if 之前总是有分号。分号是语句的结尾而不象在 Algol 和 Pascal 中的分隔符，熟悉那些语言的读者起初会觉得 Ada 的风格令人讨厌。然而，Ada 的一致性好，并且一行行的程序编辑非常简单。

我们总是希望根据条件来选择所要做的动作，在这种情况下要加上 else，后面跟条件为 FALSE 时，要执行的语句序列，在上一章中已见过这样的例子：

```
if TODAY=SUN then
    TOMORROW:=MON;
else
    TOMORROW:=DAY'SUCC(TODAY);
end if;
```

Algol 60 和 Algol 68 的用户会注意到，Ada 不是表达式语言，因此不准许有条件表达式。不能写成

```
TOMORROW:=
    if TODAY=SUN then MON else DAY'SUCC(TODAY) end if;
```

在语句串中的语句是相当随意的，if 语句可以重叠，例如要解二次方程

$$ax^2 + bx + c = 0$$

首先要检查 a ，如果 $a=0$ ，则方程将为线性方程，只有一个根 $-c/b$ ，（数学家会知道，另一个根到无穷之外去了）。如果 a 不是 0，则检查 $b^2 - 4ac$ 来看根是实数还是复数。可编程如下：

```
if A=0.0 then
    -- 线性情况
else
    if B * * 2 - 4.0 * A * C >= 0.0 then
        -- 实数根
    else
        -- 复数根
    end if;
end if;
```

可以看到，end if 重复出现。这样很难看，而且在程序中出现的非常频繁，因此引入一个附加结构，用保留字 elsif。

```
if A=0.0 then
    -- 线性情况
elsif B * * 2 - 4.0 * A * C >= 0.0 then
    -- 实根
else
    -- 复根
```

```
end if;
```

这个结构强调有三种情况,elsif 可以重复任意多次,而且最后的 else 是可选择的。这种结构的 if 语句也很简单,诸条件依次求值,直到遇到 TRUE 为止,然后执行相应的语句。如果没有条件为 TRUE,则执行 else 部分。如果没有 else 部分就什么也不执行。

要注意 elsif 的拼写,它只是 Ada 的一个保留字,而不是一个英文单词。还要注意格式,elsif,else 和 if 与 end if 都在同一列上,而且所有的凹进去的语句也以同一列开头。

请再看一个例子,假设我们训练士兵,他们可执行下述的四个不同的命令。

```
type MOVE is (LEFT,RIGHT,BACK,ON);
```

调用子程序 TURN_LEFT,TURN_RIGHT,TURN_BACK 来执行相应的命令,假设有类型为 MOVE 的变量 ORDER 用来存放待执行的命令。对队列操作的口令控制如下:

```
if ORDER=LEFT then
    TURN_LEFT;
else
    if ORDER=RIGHT then
        TURN_RIGHT;
    else
        if ORDER=BACK then
            TURN_BACK
        end if;
    end if
end if;
```

但如把程序写成下列形式就非常清楚和整洁。

```
if ORDER=LEFT then
    TURN_LEFT;
elsif ORDER=RIGHT then
    TURN_RIGHT;
elsif ORDER=BACK then
    TURN_BACK;
end if;
```

此例中没有 else 部分。采用了 elsif,虽然比用嵌套的 if 语句更好些,但仍不是好的解决方法,因为它掩盖了对称性和四种情况的相互排斥性(“相互排斥”意味着只能选择其中之一),我们以任意的秩序测试诸条件,不会产生任何本质问题。对上例的改进方法是采用 case 语句,我们将在下节看到。

对于二次方程的例子,情况是不互相排斥的,测试必须按顺序进行。如果我们先测 $b^2 - 4ac$,则我们必须在每个选择中都测 a 是否为 0。

Ada 没有象 Algol 68 的直接对 then if 相应的缩写。而实际上经常使用简化的控制格式 and then。

因此下列程序：

```
if J>0 then
  if I/J>K then
    ACTION;
  end if;
end if;
```

可以写为

```
if J>0 and then I/J>K then
  ACTION;
end if;
```

练习 5.1

1. 变量 DAY, MONTH 和 YEAR 中含有今天的日期, 说明如下:

```
DAY,INTEGER range 1..31;
MONTH,MONTH_NAME;
YEAR,INTEGER range 1901..2099;
type MONTH_NAME is (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC);
```

写段程序把变量改为含有明天的日期。如果今天是 31 DEC 2099 将怎么办?

2. X 和 Y 为二个实型变量, 要确保 X 的值大, 如需要则写一个交换其值的语句, 用一个分程序说明一下临时变量 T.

5.2 Case 语句

case 语句可使我们根据表达式的值来选择几个语句中的一个, 如训练战士的例子可以写成

```
case ORDER is
  when LEFT=>TURN_LEFT
  when RIGHT=>TURN_RIGHT;
  when BACK=>TURN_BACK
  when ON=>null;
end case;
```

必须提供 case 表达式所有的值, 要防止产生意外的遗漏。如果在这个例子中, 对一个或多个值没有对应的动作, 则要用空语句

空语句写成

```
null;
```

该语句什么事都不做, 它强调我们确实不想做任何事。这里的语句串同 if 语句一样, 必须至少含有一个语句, (不象 Algol 60, 可以没有语句)。实际应用时经常会出现表达式的多个值对应



同一行为的情况。请考虑下列程序：

```
case TODAY is
    when MON|TUES|WED|THU =>WORK;
    when FRI                  =>WORK;
                                PARTY;
    when SAT|SUN              =>null;
end case;
```

这表示了星期一到星期四要去工作，星期五也要去工作，但工作之后去参加晚会，周末什么也不做，选择值被竖线分开，请再次注意空语句的使用。

如果几个连续的值有同一行为则使用范围更为方便。

```
when MON..THU=>WORK;
```

有时，人们要表达所有未直接指出的候选值均对应同一动作。Ada为此提供了保留字 others，上面的例子我们可以写成

```
case TODAY is
    when MON..THU =>WORK;
    when FRI       =>WORK;
                                PARTY;
    when others    =>null;
end case;
```

在这里用语法可以清楚地解释，什么是合法的。

```
case 语句 ::= =
    case 表达式 is
        case 语句选择
        {case 语句选择}
    end case;
```

```
case 语句选择 ::= =
```

```
    when 选择{|选择}=>语句序列
```

```
选择 ::= 简单表达式|离散范围|others|简单元素名
```

```
离散范围 ::= 离散子类型指示|范围
```

```
子类型指示 ::= 类型标志[限制]
```

```
类型标志 ::= 类型名|子类型名
```

```
范围 ::= 范围限制|简单表达式..简单表达式
```

我们来看 when 后跟着用竖线分开的一个或多个选择，选择可以是简单表达式、离散范

注：在 case 语句选择生产式中竖线是代表它本身，而不是分隔符。

围,或 others(语法表示了选择也可以是简单元素名,但它是用于其它情况的选择,我们至今还没遇到过。在这里可以先不管它)。简单表达式,当然只给出一个单值,例如 FRI,离散范围可以是多种形式,它可以是语法的格式范围,即二个点分开的二个简单表达式,例如 MON..THU。范围也可以使用范围约束给出,我们在下章会遇到范围约束。离散范围也可以是子类型指示,即我们知道的类型标志(类型名或子类型名)后跟可选择的适当的约束,当然约束必须是范围约束。约束是保留字 range 后跟语法格式范围。

MON..THU

DAY range MON..THU

WEEKDAY

WEEKDAY range MON..THU

在这里好象并不需要这么多表示形式,但我们以后会看到,这些表示形式不仅用在 case 语句,还可用于其它上、下文中。在 case 语句中,没有必要用类型名,因为可从上、下文中得知。同样,也没有必要用子类型名后跟约束这种形式,因为只用约束即可。然而要严格地符合范围要求,单独使用子类型名是很有用的。例如我们可以写成

```
case TODAY is
    when WEEKDAY=>WORK,
        if TODAY=FRI then
            PARTY;
        end if;
    when others      =>null;
end case;
```

然而这种写书形式使人感到不太整齐。

还有许多限制,语法并没有告诉我们。例如:others 只能出现一个,并且必须作为最后的选择。如上指出,它代表所有那些未作为显式选择给出的 case 表达式的可能值(即使没有其它可能再需考虑,仍然可以使用 others)。

另一个重要的限制是,选择中的表达式必须是静态的,以便在编译时得到值。实际上,在我们的例子中用的都是直接量。

最后我们回到本节开始的要点,即选择必须包含 case 表达式的所有可能值,这意味着要包含 case 表达式类型的所有的值。当变量被说明为无约束类型(如 TODAY)时,如 case 表达式是简单形式,并且属于某个静态子类型(约束是静态表达式,并在编译时决定其值),则选择只需包含所有子类型的值。换句话说,如果编译程序能确定 case 表达式只能取某个值集中的值,则选择只要包含这个值集即可。表达式的简单格式可以是静态子类型的表达式转换。

在我们的例子中,如果 TODAY 是子类型 WEEKDAY,则它只可取 MON..FRI 中的值。因此选择只需要包含这些值,即使 TODAY 没被约束,我们仍可把 case 表达式写成约束表达式 WEEKDAY'(TODAY)同样,它也只能取 MON..FRI 的值,我们可以写成

```
case WEEKDAY'(TODAY) is
    when MON..THU      =>WORK;
```

```

when FRI          => WORK;
                    PARTY;
end case;

```

但是,如 TODAY 的值不在 WEEK'DAY 子类型的值域中,(如是 SAT 或 SUN),则会引起 CONSTRAINT_ERROR,约束不能防止 TODAY 的值为 SAT 或 SUN。因此,这并不是解决问题的好方法。

在下例中,我们有变量

```

I:INTEGER range 1..10;
J:INTEGER range 1..N;

```

N 不是静态的,我们知道 I 属于静态子类型(虽然是匿名的),但 J 则不同。如 I 做为表达式用于 case 语句中,则选择只需包含 1..10 之间的值。但要把 J 用于 case 语句中,则选择要包含类型 INTEGER(INTEGER'FIRST..INTEGER'LAST) 范围内的所有值。

以上关于 case 语句的讨论无疑给读者留下 case 语句相当复杂的印象。如果我们来归纳一下,实际上只需记住几个关键点:

- . case 表达式的每一个可能值都必须包含在某一选择中,且相同的选择不能在同一 case 语句内多次出现。
- . 所有值和范围都必须是静态的。
- . 如果用 others,则必须把它放在最后,并只能出现一次。

练习 5.2

1. 用 case 语句重写练习 5.1(1),给 END_OF_MONTH 赋正确的值。
2. 农民冬天耕地,春天播种,夏天照料植物生长,秋天收获。写出按月调用子程序 DIG,SOW,TEND,HARVEST 子程序的 case 语句,说明所需的子类型。
3. 不勤俭的人,月初领工资,前 10 天猛吃,再 10 天只能维持生活,剩下的 10 天只能挨饿,按照天来调用子程序 GORGE, SUBSIST 和 STARVE,假设设定了 END_OF_MONTH,D 被说明为如下:

```
D:INTEGER range 1..END_OF_MONTH;
```

5.3 循环语句

最简单的循环语句是

```
loop
```

语句

```
end loop;
```

该程序无限地执行循环内的语句,除非其中的某个语句因某种原因终止循环。可以永远重复执行的程序如下:

```

loop
    WORK;
    EAT;
    SLEEP;

```

```
end loop;
```

更为实用的无限循环的例子是计算自然对数的基本公式：

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \dots$$

其中 $n! = n * (n-1) * (n-2) \dots 3 * 2 * 1$

此问题的解为：

```
declare
  E:REAL:=1.0;
  I:INTEGER:=0;
  TERM:REAL:=1.0;
begin
  loop
    I:=I+1;
    TERM:=TERM/REAL(I);
    E:=E+TERM;
  end loop;
  ...

```

每次循环都是用 I去除前项(TERM)而得到一新项(TERM)，然后把新项(TERM)加入到累加和中，E 即为当前累加和。项数 I 是一个整数，逻辑上它是计数器，作为计数用。因此我们必须写成 REAL(I)。在程序开始时，给 E, I 和 TERM 置初值，即为第一项的值(I=0)。

原则上说计算将永远进行下去，使 E 越来越接近 e，但实际上因计算机的精度有限，TERM 最终将变为 0，此时再继续计算下去是没有意义的。也就是说为便于得到结果，我们应考虑在适当时刻脱离循环，脱离循环可以使用语句：

```
exit;
```

exit 称为出口语句，如在循环内执行该语句，则立即终止循环，控制转到 end loop 之后的语句。

假设我们决定在 n 项后终止循环(即当 I=N)，我们可以把循环写成：

```
loop
  if I=N then exit; end if;
  I:=I+1;
  TERM:=TERM/REAL(I);
  E:=E+TERM;
end loop;
```

鉴于语句形式

```
if 条件 then exit; end if;
```

尤为常用，故 Ada 为此提供了一种简洁的形式。



exit when 条件；
我们可以用这种简洁形式复写程序。

```
loop
    exit when I=N;
    I:=I+1;
    TERM:=TERM/REAL(I);
end loop;
```

虽然 exit 语句可以出现在循环内的任何地方，即可出现在循环中间亦可靠近循环结尾处，但 Ada 还是为我们提供了另外一种循环结构，即保留字 while 加循环条件，此循环语句为先测试循环条件，再决定是否进行循环。

```
While I/=N loop
    I:=I+1;
    TERM:=TERM/REAL(I);
    E:=E+TERM;
end loop;
```

每次循环开始时，先计算和检测条件，满足条件才进入循环体。

最后一种循环结构是，设置循环次数，循环时依次取离散范围内的各个值，直到所有的值都被用过后循环结束，借助此种循环结构，我们的例子可以改成：

```
for I in 1..N loop
    TERM:=TERM/REAL(I);
    E:=E+TERM;
end loop;
```

在此，循环控制变量 I 的值为 1, 2, 3, … N。

I 在循环子句中被隐含说明，不需要在外部再对其进行说明。它的类型由离散范围确定。循环控制变量局部于循环体，而且循环体内语句不能改变它。当我们离开循环（不论是怎样离开）后，I 就不复存在。因此，我们不能在循环外部读到其终值。

如果我们想知道循环的终值，我们可以用 exit 语句退出循环，而把 I 的值拷贝到一个在外部说明的变量中。

```
if condition_to_exit then
    LAST_I:=I;
    exit;
end if;
```

循环变量一般是以递增顺序取离散范围内的值。也可以指定循环为递减顺序

```
for I in reverse 1..N loop
```

但范围总是以递增顺序书写的。

循环步长只能为 1。大多数的应用情况是以步进 1 递增地取离散范围内的值，也有少数情况

以递减顺序取离散范围内的值,对于步长非1的情况可用 While 语句来解决。

离散范围可以是空(例如在我们的例子中 N 是零或是负数)。在这种情况下就不执行循环中的语句。离散范围只求值一次,在循环体内无法重新修改离散范围的值。例如:

```
N := 4;  
for I in 1..N loop  
...  
N := 10;  
end loop;
```

只执行四次循环,尽管事实上 N 已被修改成10。

我们的例子总是以1为下限,当然并非都要这样。上、下限可以是动态表达式。而且循环参数不必是整型,它可以是任意的离散型。

例如,我们可以写出一周的活动

```
for TODAY in MON..SUN loop  
case TODAY is  
...  
end case;  
end loop;
```

这里隐含说明了,TODAY 的类型是 DAY,顺序地按照值 MON,TUE,...SUN 来执行循环。

还有其它一些好的离散范围表示形式(如用类型和子类型名)。由于 MON..SUN 代表了子类型 DAY 的所有值,故最好用类型名 DAY 来描述循环语句的离散范围。如:

```
for TODAY in DAY loop  
...  
end loop;
```

考虑到周末不做任何事,我们可以写成:

```
for TODAY in DAY range MON..FRI loop
```

或更好地写成:

```
for TODAY in WEEKDAY loop
```

注意,有趣的是在 case 语句中类型的确立与 for 语句中的不同。在 case 语句中,离散范围的类型由 case 之后的变量表达式的类型所确定。而在 for 语句中,循环变量的类型是由离散范围的类型所确定的,二者恰好相反。

在 for 语句中,离散范围的类型不得有二义性。如果我们有两个具有两个重载直接量的枚举型,则往往会导致二义性的产生,例如:

```
type PLANET is (MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS,  
NEPTUNE, PLUTO);  
type ROMAN_GOD is (JANUS, MARS, JUPITER, JUNO, VESTA, VULCAN, SATURN,  
MERCURY, MINERVA);
```

则

```
for X in MAR..SATURN loop
```

便具有二义性,编译程序将不能编译这样的程序。我们通过对离散范围上、下界表达式中至少一个加以限定来消除二义性。

```
for X in PLANET'(MARS)..SATURN loop
```

或(可能更好)直接用类型描述离散范围

```
for X in PLANET range MARS..SATURN loop
```

如果我们更深一步接触数值时,将会认识到,范围1..10不一定是类型 INTEGER。按照一般的规则,为了在 for 语句中避免类型的二义性,则应写成:

```
for I in INTEGER range 1..10 loop
```

然而,在通常情况下这是 p 非常令人讨厌的,因此对用在 for 语句中的离散范围有一特殊的规则,即整型直接量表示的范围隐含为 INTEGER 类型。最后来看一下 exit 语句,以前所遇到的是它的简单形式,总是立即将控制转到最内层循环之外,因为循环可以是嵌套的。有时,我们可能需要退出嵌套结构,例如我们利用双重循环进行搜索。

```
for I in 1..N loop
  for J in 1..M loop
    --如果 I 的值和 J 的值都满足某个条件
    --则退出循环嵌套
    end loop;
end loop;
```

在内层循环的简单 exit 语句只能退出内层循环,我们还要再次检查条件和再次退出外层循环。可以采用给外部循环命名,并在 exit 语句中加入循环名的方法来进行一次性退出。

```
SEARCH:
for I in 1..N loop
  for J in 1..M loop
    if condition_O_K then
      I_VALUE:=I;
      J_VALUE:=J;
      exit SEARCH;
    end if;
  end loop;
end loop SEARCH;
--控制转向这里
```

在循环之前用标识符和冒号来给循环命名(它非常象其它语言中的标号,但它不是标号并且不能被用于 goto),标识符必须在对应的 end loop 和分号间再次出现。条件 exit 语句也可用于这种命名循环,如:

exit SEARCH when 条件;

练习 5.3

1. 语句 GET(I), 是从输入文件中读入下一个值并赋给变量 I, 写一段程序, 读一串数, 并相加, 用负数来作为这一串数的结束。
2. 计算因子为 N 的 2 的乘幂, 计算结果存入 COUNT, N 不能变。
3. 计算

$$g = \sum_{p=1}^n \frac{1}{p} - \log n \quad (\text{当 } n \rightarrow \infty \text{ 时}, g \rightarrow r = 0.577215665\cdots)$$

5.4 Goto 语句和标号

许多人会感到惊讶, 在这个现代化编程语言中还有 goto 语句。使用 goto 语句是非常不好的编程习惯, 因为它导致程序正确性证明的困难, 并难于维护等诸多问题。Ada 提供了充分的控制结构, 可使编程不必使用 goto 语句。

但是为什么还提供 goto 呢? 主要原因是自动生产程序。如果我们要把其它语言的程序转变为 Ada 语言的程序, 就可能会用到 goto。另一种可能的情况是程序由高层规范自动生成。最后, goto 提供了紧急出口, 例如从一些多层嵌套结构中退出。但此时须考虑可能会引起异常(见第10章)的情况。

Ada 的标号是用双括号括起来的标识符。

<<THE-DEVIL>>

goto 语句是由保留字 goto 后跟标号标识符和分号组成。

Goto THE-DEVIL;

goto 语句不能把控制从 if, case, loop 语句外转入其内。也不能转向 if 或 case 语句的诸多分支之间。

练习 5.4

1. 用标号<<SEARCH>>来重写5.3节的嵌套循环, 为什么这不是好方法?

5.5 语句分类

Ada 中的语句可分为见图 5.1:

我们在下一章讨论了复合类型后, 再进一步详细讨论赋值语句, 在第7章中将讨论过程调用和 return 语句, 在第10章中讨论有关异常的 raise 语句, 剩下的语句(entry call, delay, abort, accept 和 select 语句)在第14章中讨论。

所有的语句都可以有一个或多个标号, 简单语句不能再分解成其它语句, 复合语句则可以再分解, 也可以嵌套。程序按顺序执行, 除非遇到了控制语句(或强行中断)。

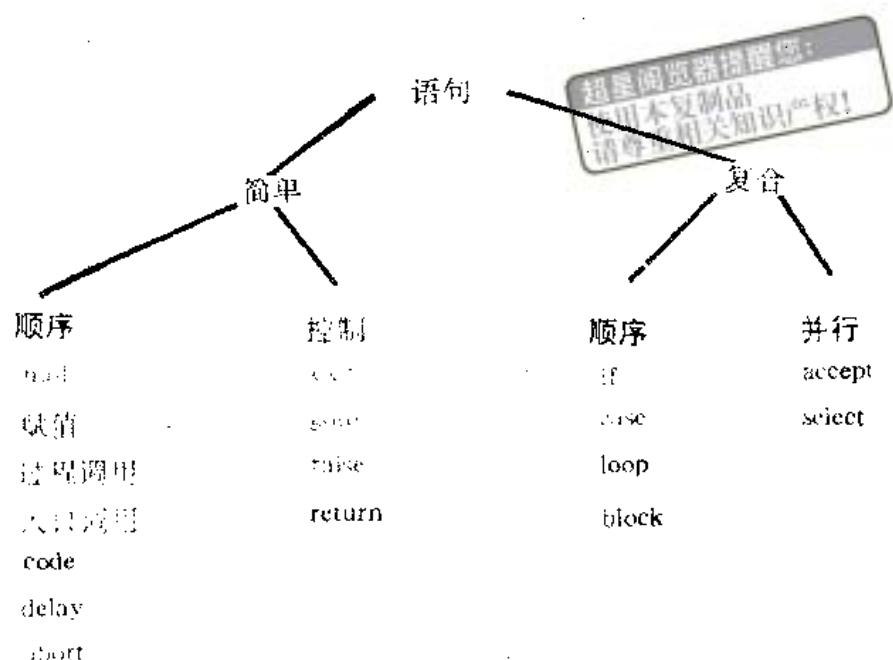


图 5.1 语句分类

要点 5

- 语句括号要正确地匹配。
- 使用 elif 要恰当。
- case 语句中的选择项必须是静态的。
- 使用 case 语句需考虑所有可能的情况。
- 如果用 others，则必须放在最后并只能 case 语句需考虑所有可能的情况。
- 如果用 others，则必须放在最后并只能有一个。
- 可通过限制 case 后面的表达式，来简化选择项。
- 循环参数相当于常量。
- 带名称的循环语句，必须在循环首尾两头都有循环语句名。
- 避免使用 goto。
- 使用推荐的格式。

第6章 复合类型

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

这一章将介绍复合类型,即数组和记录,同时也要介绍字符和字符串,从而完成关于枚举类型的讨论。本章将详细地介绍数组和记录的简单形式。较难的记录,如变体记录将留到第11章介绍。

6.1 数组

数组是由多个同类型元素组成的组合体。数组可以是一维、二维或多维。典型的数组说明为

```
A;array (INTEGER range 1..6) of REAL;
```

这里说明了 A 是一个变量,它有 6 个元素,每个元素的类型都是 REAL。数组元素是用数组名加括号和表达式来表示的,即数组名(表达式)。如上说明的数组,其表达式的值必须是离散范围 1..6 中的整数。如果表达式的值超出了该离散范围,则会引起异常 CONSTRAINT_ERROR。我们可以通过 for 语句给数组的每个元素置零

```
for I in 1..6 loop
  A(I):=0.0;
end loop;
```

数组可以是多维的。此时,要分别给出每一维的范围。如:

```
AA;array (INTEGER range 0..2, INTEGER range 0..3) of REAL;
```

数组 AA 总共有 12 个元素,每个元素都是由 2 个下标值来确定,第一个下标值的范围是 0..2,第二个下标值的范围是 0..3。可用多重循环来给这个二维数组的每个元素置零。

```
for I in 0..2 loop
  for J in 0..3 loop
    AA(I,J):=0.0;
  end loop;
end loop;
```

离散范围不必是静态的。如:

```
N;INTEGER:=...;
B;array(INTEGER range 1..N) of BOOLEAN;
```

仅当 B 的说明确立时,才可根据 N 的值确定 B 的元素个数。当然,由于 B 的说明可以位于一个循环内,因此,在程序运行期间它可被多次确立,并且因 N 的值每次可能不同,而使每次确立分别产生不同的数组。与其它对象一样,一旦执行到包含 B 说明的分程序末尾,数组 B 便不再存在。N 和 B 可以在同一说明部分被说明,但根据说明的线性确立原则,N 必须在 B 之前被说明。



数组下标的离散范围遵守与语句同样的规则。重要的一点是，象 1..6 这样的范围的类型隐含为 INTEGER。因此我们可以写成：

```
A;array (1..6) of REAL;
```

然而数组下标可以是任何离散范围，例如：

```
HOURS_WORKED;array (DAY) of REAL;
```

这个数组有 7 个元素，它们分别为 HOURS_WORKED(MON), … HOURS_WORKED(SUN)。我们可以用下列程序给这些变量赋适当的值：

```
for D in WEEKDAY loop
    HOURS_WORKED(D):=8.0;
end loop;
HOURS_WORKED(SAT):=0.0;
HOURS_WORKED(SUN):=0.0;
```

如果我们只想说明数组 HOURS_WORKED 有对应于 MON..FRI 的元素，则可以写成

```
HOURS_WORKED;array (DAY range MON..FRI) of REAL;
```

或(更好)

```
HOURS_WORKED;array (WEEKDAY) of REAL;
```

Ada 提供了有关数组的多种属性。A'FIRST 和 A'LAST 给出了 A 的第一维下标的上、下界。因此根据我们对 HOURS_WORKED 的上述说明

```
HOURS_WORKED'FIRST=MON;
```

```
HOURS_WORKED'LAST=FRI;
```

A'LENGTH 给出了第一维的大小即下标值的数目

```
HOURS_WORKED'LENGTH=5
```

A'RANGE 是 A'FIRST..A'LAST 的缩写，因此

```
HOURS_WORKED'RANGE 是 MON..FRI
```

属性可以用于多维数组的各维，只需在括号中加上表示是第几维的表达式，表达式必须都是静态的。以二维数组 AA 为例：

```
AA'FIRST(1) = 0
```

```
AA'FIRST(2) = 0
```

```
AA'LAST(1) = 2
```

```
AA'LAST(2) = 3
```

```
AA'LENGTH(1)=3
```

```
AA'LENGTH(2)=4
```

并且

```
AA'RANGE(1) 是 0..2
```

```
AA'RANGE(2) 是 0..3
```

第一维的下标，即(1)可以省略。但使用时，一般仅在一维数组中省略，而在多维数组中不



予省略。数组的属性是很有用的，应尽可能地使用属性，以反映对象在程序中的关系。这也意味着，如果修改程序，那么只需作局部修改。

属性 RANGE 在循环中尤其有用。我们的上述例子最好写成

```
for I in A'RANGE loop
    A(I) := 0.0;
end loop;

for I in AA'RANGE(1) loop
    for J in AA'RANGE(2) Loop
        AA(I,J) := 0.0;
    end loop;
end loop;
```

属性 RANGE 也可用在说明中

```
J;INTEGER range A'RANGE;
```

等于

```
J;INTEGER range 1..6;
```

如果在数组括号中的下标中用到变量，如 A(J)，则最好是使该变量与数组说明中的离散范围具有同样的约束。这可使运行时所需要进行的检查尽量少。一般情况下，给下标变量的赋值频度要小于对数组 A(J) 的存取。

我们所看到的数据元素只是一般的变量。它们可以被赋值，也可以用于表达式中。

象其它的变量一样，可以给数组赋初值。经常用聚集的形式给数组赋初值。最简单的聚集形式是多个并列表达式，按顺序给每个元素赋值聚集在括号内，各表达式之间用逗号分开。因此，我们可以按下列方法给数组 A 赋初值

```
A;array (1..6) of REAL:=(0.0,0.0,0.0,0.0,0.0,0.0);
```

如果数组是多维的，则聚集要写成嵌套形式

```
AA;array (0..2,0..3) of REAL:=((0.0,0.0,0.0,0.0),
                                (0.0,0.0,0.0,0.0),
                                (0.0,0.0,0.0,0.0));
```

聚集必须完整，如果我们要初始化数组的某些元素，则必须对整个数组进行初始化。

对数组中单个元素赋初始值，不一定采用直接量，亦可用初始表达式。初始表达式只有当其所在说明被确立时才被求值。在用聚集时，聚集中各个表达式的求值次序是不确定的。

数组可以被说明为常量，这种情况下，必须有初始值。常量数组常作为表格供查询用。下列数组可作为工作日表供查询

```
WORK_DAY;constant array (DAY) of BOOLEAN
          :=(TRUE,TRUE,TRUE,TRUE,TRUE,FALSE,FALSE);
```

一个有趣的例子是用一个数组来查看明天是星期几，而不必担心一周是否要结束了。

```
TOMORROW;constant array (DAY) of DAY
           :=(TUE,WED,THU,FRI,SAT,SUN,MON);
```

对于任意一天 D, TOMORROW(D)给出第二天是星期几。

最后要注意,数组元素可以是任何类型或子类型。而且多维数组的各维可以是不同的离散类型。例如:

```
STRANGE;array (COLOUR, 2..7,WEEKDAY range TUE..THU)
of PLANET range MARS..SATURN;
```

练习 6.1

- 说明一个整型数组 F,其下标从 0 到 N,写出数组 F 的各元素为 Fibonacci 数的程序。

$$F_0 = 0, F_1 = 1, F_i = F_{i-1} + F_{i-2}$$

- 试编写找出数组 A 中最大元素的下标值 I,J 的程序。

```
A;array (1..N,1..M) of REAL;
```

- 说明一个数组 DAYS_IN_MONTH,它给出每个月的天数,参看练习 5.1(1),用它重写那个例子,参看 5.2(1)。

- 说明一个类似于数组 TOMORROW 的数组 YESTERDAY。

- 说明一个常量数组 BORDER,

$$\text{BORDER}(P, Q) = P \text{ or } Q$$

- 说明一个 3 阶常数单位矩阵 UNIT。

6.2 数组类型

上节我们介绍的数组,没有明确的类型名。它们实际是匿名类型。这是 Ada 中说明一个对象,而没有类型名的情况之一,没有类型名的另一种情况是任务。

对于前一节的简单形式,我们可以写成

```
type VECTOR_6 is array (1..6) of REAL;
```

然后按通常的方法来说明 A

```
A;VECTOR_6;
```

用类型名的优点是,它可使我们对分别说明的数组整个地进行赋值,假如我们有

```
B;VECTOR_6;
```

则我们可以写成

```
B := A;
```

其效果是:

```
B(1) := A(1); B(2) := A(2); ... B(6) := A(6);
```

尽管赋值的次序有可能不同。

另一方面,如果我们写成。

```
C;array (1..6) of REAL;
```

```
D;array (1..6) of REAL;
```

则 D := C; 是非法的,因为 C 和 D 不是同一类型。它们是两个不同的匿名类型。基本规则是,每个类型定义引入一个新类型。在这里,语法告诉我们从 array(不包括它)到分号之间是数组的类型定义。

此外,如果我们写成

```
C,D;array (1..6) of REAL;
```

则 D := C 仍是非法的。这是因为在 4.1 节中提到的规则,复合说明只是前面两个说明的缩写。

因此,它们仍是二个不同的类型。

是否要为数组引入类型名,很大程度上取决于具体应用情况下数据的抽象程度。若某数组在程序中被多次作为整个对象使用时,宜将为它引入类型名。

若某数组仅作为一个可索引的聚集,且与其它数组无关,则它可为匿名类型。

象上节中的 TOMORROW 和 WORK_DAY 便是匿名类型数组的例子,它们仅作为检索的对象,而与其它数组无关。为这样的数组引入类型名,反而会引起不必要的混乱和错觉。

反之,如果我们要使用大量的维数相同的实型数组,且存在共同的基本抽象类型,则应当为该类型命名。

至今为止所介绍的数组类型的模式仍不能令人满意。因为这些模式仍不能表述类型相同而上、下界不同的数组间共性的抽象概念。实际上,写子程序时不能把无下标约束的数组作为实参,这是 Pascal 语言中最困难的地方。因此,Ada 引入无约束数组类型的概念,即不给出数组下标取值范围的约束。

```
type VECTOR is array (INTEGER range <>) of REAL;  
(符号<>称为“框”。
```

这就是说,VECTOR 是一个类型名,它是元素为实型的一维数组。它有类型为 INTEGER 的下标,但没有给出上、下界。range<>表示以后再给出其范围。

当我们说明类型 VECTOR 的对象时,必须提供上、下界。有几种方法来进行说明,我们可以引入中间子类型,然后说明对象。

```
subtype VECTOR_5 is VECTOR(1..5);  
V:VECTOR_5;
```

或者,我们也可以直接说明对象

```
V:VECTOR(1..5);
```

不论在那种情况下,下标值的约束即括号中的离散范围给出了数组的上、下界。以往所述的所有离散范围的表示方法在此均适用。

也可以用子类型名给出下标

```
type P is array (POSITIVE range <>) of REAL;
```

在这种情况下,被说明的任何对象的上、下界,都必须在被下标子类型 POSITIVE 所规定的范围内。注意,下标子类型必须用类型标志给出,而不能是子类型指示。

现在可以看到,我们写的

```
type VECTOR_6 is array (1..6) of REAL;
```

是下列的缩写

```
subtype index is INTEGER range 1..6;  
type anon is array (index range <>) of REAL;  
subtype VECTOR_6 is anon (1..6);
```

另一个有用的数组类型说明是:

```
type MATRIX is array (INTEGER range <>, INTEGER range <>)  
of REAL;
```

我们可再次引入子类型

```
subtype MATRIX_3 is MATRIX (1..3,1..3);  
M:MATRIX_3;
```

或直接说明为

```
M:MATRIX(1..3,1..3);
```

要注意的重要一点是,数组类型或子类型必须给出数组所有维的上、下界或者所有维的上、下界都不给。为 MATRIX 引入一个别名是完全合法的。

```
subtype MAT is MATRIX;
```

这里没有给出上、下界。不允许有只给出某一维的上、下界,而不给出其它维的上、下界的类型或子类型。

在我们讨论过的所有情况中,范围的上、下界不必是静态的。上、下界可以是任意表达式,表达式在下标约束确立时才被计算。因此,我们有

```
M:MATRIX(1..N,1..N);
```

M 的上界是在 M 被确立时 N 的值。范围可以是空,如上例中 N 的值是零。在这种情况下,M 将没有元素。

还有另外一种提供数组上、下界的方法,但只能用于常量数组。常量数组象其它常量一样具有初值。如果没有直接给常量数组提供上、下界,则可从初值中得到。初值可以是正确类型的任意表达式,但通常是上节中的聚集。聚集的形式是括号中的一系列表达式。这样的聚集是位置聚集,因为是按位置的顺序进行赋值。用位置聚集可给数组赋初值,并给出数组上、下界,数组的下界是 S'FIRST,其中 S 是下标的子类型,上界可根据元素的个数推断出(下面我们将讨论在其它上、下文中位置聚集的上、下界)。

假设我们有

```
type W is array (WEEKDAY range <>) of DAY;  
NEXT_WORK_DAY: constant W := (TUE,WED,THU,FRI,MON);
```

则数组的下界是 WEEKDAY'FIRST=MON,上界是 FRI。在 W 的定义中,用 DAY 或 WEEKDAY 都可以,因为 DAY'FIRST 和 WEEKDAY'FIRST 是一样的。

用初始值来提供上、下界时,需要小心。例如:

```
UNIT_2:constant MATRIX:=((1.0,0.0),(0.0,1.0));
```

本打算说明 2×2 矩阵为 $UNIT_2(1,1)=UNIT_2(2,2)=1.0$, 而 $UNIT_2(1,2)=UNIT_2(2,1)=0.0$ 。

但是灾难降临了! 我们实际上说明数组的下界是 INTEGER'FIRST, 它可能是 -32768 或类似的数, 而绝不是 1。

如果我们说明类型 MATRIX 为

```
type MATRIX is array (POSITIVE range <>,POSITIVE range <>)  
of REAL;
```

则一切问题都解决了, 因为 POSITIVE'FIRST=1。因此, 用赋初值的方法来推断数组的上、下界, 可能会引起奇怪的现象。

还有一种聚集的形式, 称为带名聚集, 即在每个分量前加相应的下标值和 =>, 例如

```
(1=>0.0,2=>0.0,3=>0.0,4=>0.0,5=>0.0,6=>0.0)
```

这种形式可以扩展为多维。带名聚集本身就表明了聚集的界, 因此 2×2 矩阵就可以写为

```
UNIT_2;constant MATRIX:=(1=>(1=>1.0,2=>0.0),  
2=>(1=>0.0,2=>1.0));
```

关于带名聚集的规则与 case 语句中选择项的规则相似。

每个选择可有多个选择分量。每个选择分量可以是单个值或离散范围。我们可以重写上述例子

```
A;array (1..6) of REAL:=(1..6=>0.0);  
WORK_DAY;constant array (DAY) of BOOLEAN  
:= (MON..FRI=>TRUE,SAT|SUN=>FALSE);
```

与位置聚集不同，这时下标值不一定要按顺序书写。我们可以写成

```
(SAT|SUN=>FALSE,MON..FRI=>TRUE)
```

我们也可以用 others。如同 case 语句，它必须放在最后，并只能用一个。

用 others 时会引起一些问题。因为，这时必须搞清楚所有剩余分量的值是什么。要知道，虽然我们只在赋初值时用到过聚集。但它却可以用在需要用数组类型表达式的任何地方。如果用了 others，则必须能够根据上、下文确定其下标的界限。一种用于确定下标界限的上、下文（如同我们区分重载枚举直接量一样）是约束聚集。为此，我们必须有适当的类型和子类型名。因此引入

```
type SCHEDULE is array (DAY) of BOOLEAN;
```

然后可以写成

```
SCHEDULE'(MON..FRI=>TRUE,others=>FALSE)
```

（注意，不必给约束聚集加括号，因为它已带有括号）。

在第7章讨论子程序参数与结果和第13章类属参数时，将会遇到其它确定下标界限的上、下文。

同样，上述上、下文也可提供位置聚集的上、下界。象有 others 的带名聚集一样，位置聚集没有表明其下标界限。但在某些上、下文中，如上面讨论的初始化，可使用位置聚集，因为下界将是 S'FIRST，S 是下标子类型，上界可从元素的个数中推断出来。当然，含有 others 的带名聚集不能使用该规则，因为我们不知道 others 所代表的元素个数。

在数组聚集的运用中不可把位置聚集和带名聚集混用，但可在位置聚集后面用 others。这样的位置聚集要遵守有关含有 others 的带名聚集的规则。

要注意，由类型给出下标界限的赋值或赋初值的上、下文是非常奇特的。它只考虑适用于位置聚集或 others 为唯一选择的带名聚集，来提供下标界限。而不考虑适用于通常的含有 others 的带名聚集。因此不能写成

```
WORK_DAY;constant array (DAY) of BOOLEAN  
:= (MON..FRI=>TRUE,others=>FALSE);
```

必须写成

```
WORK_DAY;constant array (DAY) of BOOLEAN  
:= (TRUE,TRUE,TRUE,TRUE,TRUE,others=>FALSE);
```

或用类型名写成

```
WORK_DAY;constant SCHEDULE
```

`:=SCHEDULE'(MON..FRI=>TRUE,others=>FALSE);`

关于数组聚集上、下界的规则似乎非常复杂，而且很难理解，它是 Ada 中最为奇妙的内容。如果你被搞糊涂了，不必担心，因为聚集总是可以被约束的。

数组聚集的确很复杂，还需要指出几点。第一点，在带名聚集中，所有在`=>`之前的范围和值必须是静态的（象在 case 语句中一样）。有一种特殊情况除外，这就是只含一个具有单个选择分量的选择时，该选择可以是动态范围或单个动态值。例如：

`A:array (1..N) of INTEGER:=(1..N=>0);`

是许可的； N 为零（或负数）给出一个空数组和空聚集。还有一个规则，`=>`后的表达式为每个相应的下标只求一次值。当然，通常不会产生什么问题，但是下例：

`A:array (1..N) of INTEGER:=(1..N=>1/N);`

如果 N 是零，则没有值，因此不计算 $1/N$ ，不会发生 NUMERIC_ERROR。读者会想到，当同时说明和初始化多个对象时，会进行多次相似的计算。

为了避免空聚集所带来的问题，规定在只有一个选择分量时，才准许出现空聚集。象

`(7..6|1..0=>0)`

这样笨拙的聚集表示是被禁止的，它根本没有下界。

还有一点，虽然我们不能混用带名和位置表示形式。然而，我们可以对多维聚集的不同元素和不同维使用不同的形式。因此，矩阵 UNIT_2 的 4 个初始值，可以写成

`(1=>(1.0,0.0), 或 ((1=>1.0,2=>0.0),`

`2=>(0.0,1.0)) (1=>0.0,2=>1.0))`

甚至写成：

`(1=>(1=>1.0,2=>0.0),`

`2=>(0.0,1.0))`

等等。

还要注意关于范围 RANGE 属性，可以用作带名聚集中一个选择。然而，我们不能把对象的范围属性用于它自身的初始化。因此

`A:array (1..N) of INTEGER:=(A'RANGE=>0);`

是不合法的。因为对象在说明结束前是不可见的。然而我们可以写成

`A:array (1..N) of INTEGER:=(others=>0);`

这可能比重复写 `1..N` 要好，因为它的的确立取决于 N 。

最后一点，位置聚集不能只含有单个元素，否则将会产生二义性。因为我们不能识别它是一个带括号的纯量还是只有单个元素的聚集。单个元素的聚集必须用带名方式表示。

现在让我们回到对整个数组赋值这个题目上来。为了进行这样的赋值，数组值和被赋值的数组必须是同一类型，并且元素必须匹配。这并不是要求下标界限必须相符，只是相应的维数及其元素个数必须一致。我们可写成

`V:VECTOR(1..5);`

`W:VECTOR(0..4);`

`...`

`V:=W;`

`V` 和 `W` 都是类型 VECTOR，并且都有 5 个元素。也可以有

```
P;MATRIX (0..1,0..1);  
Q;MATRIX(6..7,N..N+1);  
...  
P:=Q;
```

比较数组相等与不相等时，遵守与数组赋值相同的规则。只有当二个数组是相同类型时，才能进行比较。如果二个数组相应维的元素个数相等，并且各个对应元素又相等，则二个数组相等。

虽然只能对同一类型的数组进行赋值和比较。但是，如果二个数组的数组元素类型和数组下标的类型相同，一种类型的数组可以转换为另一种类型。这种转换通常采用转换格式来实现。因此，如果我们有

```
type VECTOR is array (INTEGER range <>) of REAL;  
type ROW is array (INTEGER range <>) of REAL;  
V;VECTOR (1..5);  
R;ROW(0..4);
```

则

```
R:=ROW(V);
```

是合法的。实际上，因为 ROW 是无约束类型，ROW(V) 的上、下界，就是 V 的上、下界，这时，遵循通常的赋值规则。然而，如果对约束类型和子类型进行转换，则上、下界必须为约束类型或子类型的上、下界，并且相应维的元素个数必须相同。

以后我们会看到，当同时使用不同库中的子程序时，数组类型转换尤其有用。

读者现在会得出结论，Ada 中的数组在某种意义上说，比较复杂。这种评价是正确的，在使用时会遇到一些困难。但是如果我们写错了，编译程序会发现并告诉我们。使用正确的类型和子类型名来进行约束，可以解决二义性的问题。聚集的大部分难点也与 case 语句相同。

如同属性 FIRST, LAST, LENGTH 和 RANGE 可用于数组一样，这些属性也可用于数组类型和子类型，因此

```
VECTOR'LENGTH = 8
```

而

```
VECTOR'LENGTH
```

 是非法的。

练习 6.2

- 用带名聚集来重写练习 6.1(3) 中的数组 DAYS_IN_MONTH 赋初值说明语句。
- 说明一个常量 MATRIX，其中 2 个维的上、下界都是 1..N，N 是动态的，元素全都是 0。
- 说明上题中的常量 MATRIX，使它成为矩阵。
- 把练习 6.1(5) 中的数组 BOR 说明成数组。

6.3 字符和字符串类型

我们现在介绍字符和字符串，从而完成关于枚举类型的讨论。我们至此为止所见到的枚举类型，如：

```
type COLOUR is (RED,AMBER,GREEN);
```

其值是由标识符来表示的。还可有这样的枚举类型，它有些值，或全部值都是字符直接量。

字符直接量是一词法单位。它是一对单引号中括起来的一个字符。字符必须是可打印字符，或是单个空格。它不能是如退格或回车等控制符。

这里字母大、小写字是有区别的。字符直接量

'A', 'a'

是不同的。

因此我们可以说明一个枚举类型

```
type ROMAN_DIGIT is ('I','V','X','L','C','D','M');
```

而

```
DIG:ROMAN_DIGIT:='D';
```

具有通常的枚举类型的特点。

```
ROMAN_DIGIT'FIRST='I'
```

```
ROMAN_DIGIT'SUCC('X')='L'
```

```
ROMAN_DIGIT'POS('M')=6
```

```
DIG<'L'=FALSE
```

Ada 中设有一个预定义的枚举类型 CHARACTER，其为字符类型。CHARACTER 的说明格式为

```
type CHARACTER is (nul, …, 'A', 'B', 'C', …, del);
```

预定义类型 CHARACTER 表示标准的 ASCII 字符集。并一般用于输入和输出。详细的资料，请读者参看 LRM。

应注意，在引入了类型 ROMAN_DIGIT 和预定义类型 CHARACTER 后，便产生了某些直接量的重载。此时，表达式

```
'X'<'L'
```

具有二义性。我们不知道比较的是类型 ROMAN_DIGIT 的字符，还是 CHARACTER 的字符。要避免这种二义性，我们必须对它们进行限定。

```
CHARACTER'('X')<'L'=FALSE
```

```
ROMAN_DIGIT'('X')<'L'=TRUE
```

象预定义类型 CHARACTER 一样，还有一个预定类型 STRING

```
type STRING is array (POSITIVE range <>) of CHARACTER;
```

这是完好的数组类型，并遵守上节的所有规则。因此，我们可以写成

```
S,STRING(1..7);
```

来说明范围是 1..7 的数组。如果是常量数组，则可从初值中推断出其下标界限。

```
G:constant STRING:=(‘P’,‘I’,‘G’);
```

从一般位置聚集中得到初值。G 的下界（即 G'FIRST）为 1，因为 STRING 的下标类型是 POSITIVE，并且 POSITIVE'FIRST 是 1。

还有一种每个元素都是字符直接量的位置聚集的形式。这就是字符串。因此，我们可以简便地写成：

```
G:constant STRING:="PIG";
```

字符串是我们要介绍的最后一个词法单位。它是由双引号括起来的（字符串中的双引号可



由两个双引号来表示)一串可打印的字符和空格所组成。因此有

`('A',' ','B') = "A" "B"`

当字符串只含一个字符或为空时其等价的聚集必须用带名聚集予以表示。

`(1=>'A') = "A"`

`(1..0=>'A') = ""`

要注意,我们必须在空的带名聚集中引入一个任意字符。

另一个关于字符串的语法规则是,字符串必须顺序地写在一行中,而且中间不能含有控制符,如 SOH。当然,作为字符直接量,在字符串中字母的大、小写是不同的。

`'PIG' / = "pig"`

下一节我们会看到怎样摆脱字符串只能占单行,且不能含有控制符的限制。

字符串主要是用来建立输出文本。调用子程序 PUT 可以输出简单的字符串,故

`PUT ("The Countess of lovelace");`

是将

`The Countess of lovelace`

输出到某文件中。

字符串除了用来表示 STRING 类型的值外,还可用来表示任何其它字符类型的数组的值。我们可以写成

`type ROMAN_NUMBER is array (POSITIVE range<>) of ROMAN_DIGIT;`

然后

`NINETEEN_EIGHTY_FOUR: constant ROMAN_NUMBER := "MCMLXXXIV";`

和

`FOUR,array (1..2) of ROMAN_DIGIT := "IV";`

练习 6.3

1. 说明一个常量数组 ROMAN_TO_INTEGER 作为查询表,把罗马数转换为相等的一般整型数(如转换'C'为 100)。
2. 给出类型 ROMAN_NUMBER 的对象 R,并编写出判断 R 与整型量 V 相等的程序。假设 R 遵守罗马数结构的一般规则。

6.4 一维数组运算

我们在第 4 章中见到的许多运算都可用于一维数组。布尔运算 and, or, xor 和 not, 也可以用于一维布尔数组。进行二元运算时,二个操作分量必须有相同的元素个数和相同的类型。一个元素一个元素地施行基本纯量运算,结果仍为数组,其类型与操作分量的类型相同。结果数组的下标的下界等于左面操作分量的下标的下界,或一元操作分量的下标的子类型的下界。

请考虑

```
type BIT_ROW is array (POSITIVE range<>) of BOOLEAN;
A,B:BIT_ROW(1..4);
C,D,array (1..4) of BOOLEAN;
T,constant BOOLEAN := TRUE;
```

F;constant BOOLEAN:=FALSE;

我们可以写

A:=(T,T,F,F);

B:=(T,F,T,F);

A:=A and B;

B:=not B;

则 A 等于 (T,F,F,F), B 等于 (F,T,F,T)。相类似地可进行 or 和 xor 运算。但要注意，C and D 是不准许的，因为它们的类型不同。在这里给数组类型命名比较有利，因为这样我们可以把数组作为整体来操作。

注意这些操作，象 6.2 节中介绍的赋值语句一样，只要求类型和相应维数之元素的个数相同，上、下界可以不相等。

```
type PRIMART is (R,Y,B);
type COLOUR is array (PRIMARY) of BOOLEAN;
C:COLOUR;
```

则 C 可以取 $8=2 \times 2 \times 2$ 个值。C 是一个有三个元素的数组，每个元素可取值 TRUE 或 FALSE，三个元素是：

C(R), C(Y) 和 C(B)

可以用如下列的带名常量来表示类型 COLOUR 的 8 个可能值：

```
WHITE :constant COLOUR:=(F,F,F);
RED   :constant COLOUR:=(T,F,F);
YELLOW :constant COLOUR:=(F,T,F);
BLUE  :constant COLOUR:=(F,F,T);
GREEN :constant COLOUR:=(F,T,T);
PURPLE :constant COLOUR:=(T,F,T);
ORANGE :constant COLOUR:=(T,T,F);
BLACK :constant COLOUR:=(T,T,T);
```

然后我们可以写下列表达式：

RED or YELLOW

等于 ORANGE。而

not BLACK

等于 WHITE。

由此可见，类型 COLOUR 的值，实际上是由三基色 R, Y, B 的所有组合构成的颜色集，WHITE 的值是空集，而 BLACK 的值为全集。

or, and 和 xor 运算可以理解为集的并、交以及对称差。可以通过检测集合的元素来检测该

元素是否属于某个集合。例如,对于

$C(R)$

如果 R 在 C 的集合中,则 $C(R)$ 的值是 TRUE。在此我们不能用预定义运算 \in ,也可用带名聚集来表示数组的直接量。例如:

$(R | Y = > T, \text{others} = > F)$

与 ORANGE 同值。在下章将介绍更好的方法。

现在我们来讨论关系运算。运算符 $=$ 和 $/=$ 适用于所有类型,并且在 6.2 节讨论赋值时,已给出了有关数组赋值的规则。

关系运算符 $<$, $<=$, $>$ 和 $>=$ 可用于任何一维离散类型的数组(注意离散)。按字典顺序决定大小,即基于定义元素的顺序关系。假定 STRING 类型之值无二义性,则下列关系表达式的值都是 TRUE

"CAT" < "DOG"

"CAT" < "CATERPILLAR"

"AZZ" < "B"

"" < "A"

两字符串的比较是从左到右,逐个元素地进行的,直到所比较的元素不同时止。比较结果是对应字符小的字符串小。如果某个字符串的元素少,如 CAT 比 CATERPILLAR 少,则字符少的字符串小。空串是最小的。

如果我们说明了类型 ROMAN_NUMBER,则

"CCL" < "CCXC"

是二义的,因为我们不知道正用来进行比较字符串的类型是 STRING,还是 ROMAN_NUMBER。为此我们必须约束一个或二个进行比较的字符串。对字符串的约束按通常方法即可,但字符串必须在括号内(不象带有括号的聚集),否则单引号和双引号将并列在一起,使我们很难看清这种约束的表示形式

STRING'("CCL") < "CCXC" 是 TRUE

ROMAN_NUMBER'("CCL") < "CCXC" 是 FALSE

编译程序无法知道我们脑子里对罗马数字的解释,仅由类型定义中“L”与“X”的顺序关系得出 $250 < 290$ 是假,下一章中,我们将介绍怎样重新定义“ $<$ ”以使之能用于罗马数字。

当然,关系运算也可以用于一般表达式,而不只是用于字符串直接量。

NINETEEN_EIGHTY_FOUR < "MM" 是 TRUE

关系运算也适用于任何离散类型的数组,因此

$(1, 2, 3) < (2, 3)$

$(JAN, JAN) < (FEB, FEB)$

预定义运算 $<=$, $>$ 和 $>=$ 是从 $<$ 类推而定义的。

现在介绍新的二元运算 $\&$,即一维数组的连接。它如同二元的加和减,二个操作分量必须是相同类型,结果类型与操作数类型相同,其值是二个操作分量的连接。结果的下界,是左操作分量的下界。

因此

"CAT" & "ERPILLAR" = "CATERPILLAR"

字符串的连接可用来构成不能在一行写出的长字符串

```
"This string goes" &  
"on and on"
```

&的一个或两个操作分量可以是元素类型的单个值。如果左操作分量是单个值，则结果的下界是数组下标子类型的下界。

```
"CAT" & 'S' = "CATS"  
'S' & "CAT" = "SCAT"  
'S' & 'S' = "SS"
```

利用&还可以在字符串中插入控制字符，如CR和LF。

```
"First line" & ASCII.CR & ASCII.LF & "Next line"
```

当然，可以先定义常量。

```
CRLF:constant STRING := (ASCII.CR, ASCII.LF);
```

然后可写成：

```
"First line" & CRLF & "Next line"
```

可对任意的一维数组类型进行&运算，因此可用于罗马数字。

```
R:ROMAN_NUMBER(1..5);  
S:STRING(1..5);  
R:="CCL" & "TV";  
S:="CCL" & "TV";
```

是合理的。它告诉我们，第一个赋值语句是对二个罗马数进行&运算，而第二个赋值语句是对二个类型为STRING的值进行&运算。下列语句无二义性。

```
B:BOOLEAN:="CCL" < "TV";
```

一维数组的最后一个特点是，我们可以表示数组的一部分，这部分称“片”。用一个对象名（变量或常量）后加带括号的离散范围来描述数组片。要注意，数组片只适用于对象，而不适用于值。

因此，如给出：

```
S:STRING(1..10);
```

则 S(3..8)指的是S中间的6个字符。数组片的界限是范围的界限，而不是下标子类型的界限，例如：

```
T:constant STRING:=S(3..8);
```

则 T'FIRST=3, T'LAST=8。

数组片的界限可以是任意表达式，而且不必是静态的。如果范围为空，则数组片也为空。使用数组片时，数组赋值的特性会得到充分体现，即在给任何元素赋值前，首先要计算出赋值表达式的值。数组片的重叠将不会产生什么问题。

```
S(1..4):="BARA";  
S(4..7):=S(1..4);
```

结果 S(1..7)="BARBARA"。在计算了表达式 S(1..4)后，才改变S(4)的值，置S(4)为'B'是无害的，表达式"BARA"使最终结果为

"BARBARB"



练习 6.4

1. 写出可有3个值 WHITE...BLACK 的类型 COLOUR, 其顺序为递增形式, 是以一维数组的<运算来确定的。

2. 计算

- a)RED or GREEN c)not GREEN
- b)BLACK xor RED

3. 证明 C 取任何值, not (BLACK xor C)=C 为真。

4. 为什么我们不能写成

(JAN,JAN)<(FEB)

5. 假设 R 的值是一个罗马数,写出一段程序,检查其对应的十进制数的最低位是否是4.如果是4,将其改为6.

6. 已给出

C,CHARACTER

S,STRING(5..10)

请问下列计算结果的下界是什么?

- a)C & S b)S & C

6.5 记录

在本章的开头已指出,在这里我们只考虑最简单的记录形式。第11章再全面地讨论记录,其中包括变体记录。

记录是由不同类型的带名元素组成的复合体。与数组不同,我们不能有匿名记录类型,它们都必须是带名的。

```
type MONTH_NAME is (JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,  
NOV,DEC);  
type DATE is  
record  
    DAY:INTEGER range 1..31;  
    MONTH:MONTH_NAME;  
    YEAR:INTEGER;  
end record;
```

说明了类型 DATE 是一个含有3个带名元素 DAY,MONTH 和 YEAR 的记录。我们可以按通常的方法来说明记录类型的常量和变量。

D:DATE;

说明了对象 D 是一个日期。在 D 后跟点和分量名来指定 D 的某个元素。我们可以写成

```
D.DAY:=4;  
D.MONTH:=JUL;  
D.YEAR:=1776;
```

上式对单个元素赋新值。

也可以把记录作为整体来操作。直接量可以写成象数组一样的聚集,位置和带名的格式都可使用。因此我们可以写成

```
D:DATE:=(4,JUL,1776),
```

```
E:DATE;
```

然后

```
E:=D;
```

或

```
E:=(MONTH=>JUL, DAY=>4, YEAR=>1776);
```

记录中不能使用复杂的聚集,因为记录元素的个数总是已知的。

在位置聚集中,分量按顺序排列。在带名聚集中,分量可以按任意顺序排列。而用带名聚集时,可以不必关心记录类型是在何处被说明的。

带名聚集不能用范围,因为分量之间没有任何联系。只有在分量具有相同类型时,才能用竖线。可用 others,但也只能在剩下的分量类型相同时使用。

对于记录有一例外,即在一个聚集中可以混合使用位置和带名形式。如果这样使用,则位置分量必须按顺序放在前面(不能有间隙)。换句话说,在聚集中任意一点开始我们都可改用带名形式,但从该点之后的所有分量都必须使用带名形式。上述日期可表示为:

```
(4,JUL, YEAR=>1776)
```

```
(4, YEAR=>1776, MONTH=>JUL)
```

等等。

在类型说明中,可以给出部分或全部元素初值的隐含初值表达式

```
type COMPLEX is
  record
    RL:REAL:=0.0;
    IM:REAL:=0.0;
  end record;
```

或进一步简化为

```
type COMPLEX is
  record
    RL,IM:REAL:=0.0;
  end record;
```

说明了具有二个类型为REAL的元素的记录类型,并给出了每个元素的隐含初值表达式为0.0。

我们可以说明

```
C1:COMPLEX;
```

```
C2:COMPLEX:=(1.0,0.0);
```

记录C1的两个元素的值都为0.0,它来自于隐含初值表达式,而在C2中则把隐含初值表达式的值覆盖了。要注意,即使有隐含初值表达式为一些元素提供了隐含初值,但聚集仍必须完整。

在2个分量具有相同类型的情况下,可写成下列带名格式

```
(RL|IM=>1.0)
```

(others=>1.0)

对于记录类型,预定义的运算只有=和/=。其它运算都必须在元素级上实施,可由子程序显式定义其它的操作,这在下一章我们将会看到。

记录类型可以有任意多个元素。也可以无元素,无元素记录的说明格式如下:

```
type HOLE is
    record
        null;
    end record;
```

在此保留字 null,表示我们说明的是一个空记录类型。空记录有其特定的用途,但现在还看不出来。

记录类型的元素可以是任意类型,可以是另一个记录或数组。然而,如果元素是数组,则它必须全面受约束(例如,数组的下标不能含有<>),并且它必须是有名类型,而不能是匿名类型。显然,记录不能含有其自身。元素不能是常量,但整个记录可以是常量

```
I:constant COMPLEX:=(0.0,1.0);
```

是准许的,它表示-1的平方根。

记录的更为复杂的例子如下:

```
type PERSON is
    record
        BIRTH,DATA;
        NAME,STRING(1..20):=(1..20=>');
    end record;
```

记录 PERSON 有两个元素。第一个元素是一个记录 DATE,第二个元素是一个数组,这个数组是长度为20的字符串,隐含初值为空格。

我们可以写成

```
JOHN:PERSON;
JOHN.BIRTH:=(19,AUG,1937);
JOHN.NAME(1..4):="JOHN";
```

我们则有

```
JOHN=((19,AUG,1937),"JOHN")
```

聚集可以嵌套,处理顺序是从左到右。我们可以用短点符来选择记录的元素,括号中的下标来选择数组元素,用括号中的范围来取数组片。我们写一描述人的数组:

```
PEOPLE,array (1..N) of PERSON;
```

然后

```
PEOPLE(8).BIRTH.DAY:=19;
PEOPLE(8).NAME(3):='H';
```

等等。

最后需要考虑的一点是记录说明中表达式的计算。当确立记录类型时,计算用于元素的约束表达式。因此,类型 PERSON 可有元素

NAME, STRING(1..N) := (others => '');

当它被确立时,元素 NAME 的长度是 N 的值,当然 N 不必是静态的。因此,类型如果是在循环中被说明,则每次执行循环就可形成元素个数不同的类型。然而,当记录类型说明被确立后,类型的所有对象都具有同样多的元素个数。

另一方面,记录类型中的隐含初值表达式只在类型的对象被说明,并且在没有提供初值时才被计算。当然,在一般的情况下,象类型 COMPLEX,没有什么不同,但有时可能会产生令人吃惊的结果。例如,我们有一元素

NAME, STRING(1..N) := (1..N => '');

当说明记录类型时,元素 NAME 的长度是 N 的值。而当说明没有初始值的 PERSON 时,则又用 N 的值计算聚集。然而,这时的 N 可能已被改变,从而引起 CONSTRAINT_ERROR。

练习 6.5

1. 用类型为 DATE 的变量 D,而不是用三个单独的变量来重写练习 6.1(3)。
2. 说明类型为 COMPLEX 的 3 个变量 C1, C2 和 C3,写出把 C1 和 C2 的
a) 和 b) 积
存入 C3 的语句。
3. 写出找数组 PEOPLE 中第一个生于 1950.1.1 之后人的下标。

要点 6

- 数组类型可以是匿名的,但记录不能。
- 聚集必须是完整的。
- 受约束数组类型和无约束数组类型(有<>)不同。
- 数组聚集中,带名和位置形式不能混用,而在记录聚集中对此不做限制。
- 用含有 others 的聚集时,必须有给出其上、下界的上、下文。
- 数组聚集中的选择可以是动态的。如果某个数组聚集只有一个选择,则该聚集可为空。
- 属性 FIRST, LAST, LENGTH 和 RANGE,适用于数组和受约束数组类型以及子类型,但不适用于无约束的类型和子类型。
- 数组赋值,每维的元素个数必须相同,而上、下界可以不同。
- 在字符直接量和字符串中,字母的大、小写不同。
- 只有一个元素的聚集必须用带名形式,这对记录和数组都一样。
- 记录元素不能是匿名数组。
- 当说明无初始值的对象时,才计算隐含初值表达式。

第7章 子程序

子程序是带有参数的编程单位。在 Ada 中，子程序分为两类：函数和过程。以表达式的运算分量的形式来调用函数，被调用的函数返回一结果。而以语句形式来调用过程，被调用的过程不返回结果。当调用子程序时，意味着执行在子程序中所描述的动作。子程序体以通常的方法在说明部分中被定义。

7.1 函数

函数是子程序的一种形式，表达式中被调用。第4章，我们曾见过调用函数 DAY'SUCC, SQRT 等。

我们现在讨论函数体，即当函数被调用时要执行的那些语句。例如函数 SQRT 的体为：

```
function SQRT(X:REAL) return REAL is
  R:REAL;
begin
  --计算 SQRT(X)的值，并赋给 R
  return R;
end SQRT;
```

所有的函数体是以保留字 function 开头的，并且定义了函数名。如果该函数带有参数，则在函数名后跟一个带括号的参数规范式说明表。如果参数表中有多个规范式说明，则用分号将各个规范式说明分开。每个规范式说明是由一个或多个参数标识符，后跟冒号和它们的类型或子类型所组成。参数表后面跟保留字 return 和函数结果的类型或子类型。不论是对参数还是结果，必须用类型标志给出它们的类型或子类型，而不能用子类型表示法。

到此为止，我们所介绍的是函数的规范式说明，它指出了外界要调用该函数所需提供的所有信息。

在规范式说明之后是 is，然后是函数体。函数体象一个分程序，它有说明部分、begin 和一系列语句，最后是 end。象分程序一样，子程序可以没有说明部分，但在语句部分中至少要有一个语句。在 end 和终结分号之间可以重现函数名。虽然此处的函数名是可有可无的。但是，如果写了函数名，则必须与 function 后的函数名一致。

根据需要，可以只给出函数的规范式说明，而没有函数体的其它部分。在这种情况下，规范式说明部分后面直接跟分号，这种形式称为函数说明。以后将讨论这种说明的用途。

函数的形参是起着局部常量的作用，其值来自相应的实参。当调用函数时，按通常的方法来确立函数中的说明部分，然后执行语句。返回语句用于返回结果，并使控制返回给调用该函数的表达式。

假设有

```
S:=SQRT(T+0.5);
```

则首先计算 $T + 0.5$, 然后调用 SQRT。在函数体内, 参数 X 相当于一个常量, 其初值来自于 $T + 0.5$ 。它好象

```
X:constant REAL:=T+0.5;
```

然后 R 的说明被确立, 接下来执行语句串, 即计算 X 的平方根, 并赋给 R。最后一个语句是 return R, 它将控制返回给调用本函数的表达式, R 的值作为函数的结果。然后把函数值赋给 S。

保留字 return(返回语句)后面可尾随一个任意复杂的表达式, 但该表达式的类型必须与函数规范式说明中的类型和约束相同, 如果类型不一致, 则会引起异常 CONSTRAINT_ERROR。

函数体内可以有多个返回语句。执行其中的任何一个, 都会退出函数体。例如函数 SIGN 根据参数为正、0、负, 返回 +1, 0, -1。SIGN 的函数体为

```
function SIGN(X:INTEGER) return INTEGER is
begin
    if X>0 then
        return +1;
    elsif X<0 then
        return -1;
    else
        return 0;
    end if;
end SIGN;
```

我们看到, 函数体的最后一个语句不必是返回语句, 因为在 if 语句的每个分支中都有 return 语句。如果运行到最后的 end, 则会引起异常 PROGRAM_ERROR。这是我们遇到的引起 PROGRAM_ERROR 的第一个例子。该异常一般用于程序运行时所遇到的非法的控制结构。

应注意, 当调用函数时, 函数中的对象(包括参数)被立即重新说明。而当退出函数, 这些对象将消失。这就使函数可以进行递归调用, 而不会产生任何问题。例如阶乘函数为

```
function FACTORIAL(N:POSITIVE) return POSITIVE is
begin
    if N=1 then
        return 1;
    else
        return N * FACTORIAL(N-1);
    end if;
end FACTORIAL;
```

如果我们写成

```
F:=FACTORIAL(4);
```

则函数调用其本身,直至到第4次调用(其它3次调用都只执行了函数的一部分,而在做乘法之前等待调用结果),当N是1时,返回结果并进行相乘。

不需要检查参数N是否是正数,因为参数是子类型POSITIVE。因此,调用FACTORIAL(-2)会产生CONSTRAINT_ERROR。当然,FACTORIAL(10000)会使计算机的存贮空间不够,而产生STORAGE_ERROR。即便是调用FACTORIAL(20),也可能会引起NUMERIC_ERROR。

形参可以是任意类型,但类型必须有名,不能如下:

```
array (1..6) of REAL
```

这样的匿名类型,因为实参必须与对应的形参类型一致,而在任何情况下均无实参可以同匿名类型的形参匹配。

然而,形参可以是无约束数组类型,如:

```
type VECTOR is array(INTEGER range<>) of REAL;
```

此时,形参的下标界限取决于实参。例如,求一维向量诸元素之和的函数SUM

```
function SUM(A:VECTOR) return REAL is
    RESULT:REAL:=0.0;
begin
    for I in A'RANGE loop
        RESULT:=RESULT+A(I);
    end loop;
    return RESULT;
end SUM;
```

随后我们可以写出:

```
V:VECTOR(1..4):=(1.0,2.0,3.0,4.0)
S:REAL;
...
S:=SUM(V);
```

形参A的下标界限则取决于实参V,执行函数调用SUM(V)时,

A'RANGE 为 1..4

通过循环计算A(1),A(2),A(3)和A(4)的和,返回RESULT的最终值,赋给S的值为10.0。

函数SUM可用于求具有任意下标界限的向量元素之和。Ada克服了Pascal中数组参数下标界限必须是静态的局限。当然,Ada函数,可以把约束数组类型作为形参。但要记住,在参数表中的这种约束不能用子类型给出,如:

```
function SUM_6(A:VECTOR(1..6)) return REAL;
```

而必须用约束数组类型名,如:

```
type VECTOR_6 is array(1..6) of REAL;
```

来作为类型标志

```
function SUM_6(A;VECTOR_6) return REAL
```

现考虑另一个例子：

```
function INNER(A,B;VECTOR) return REAL is
    RESULT:REAL:=0.0;
begin
    for I in A'RANGE loop
        RESULT:=RESULT+A(I)*B(I);
    end loop;
    return RESULT;
end INNER;
```



函数 INNER 通过把向量 A 和 B 相应元素之积相加来计算内积。这是第一个具有多个参数的函数的例子，调用这样的函数，要用函数名后跟括号和用逗号分开的多个表达式的形式，这些表达式给出了实参的值，对实参的计算顺序不确定。

因此

```
V;VECTOR(1..3):=(1.0,2.0,3.0);
W;VECTOR(1..3):=(2.0,3.0,4.0);
R;REAL;
...
R:=INNER(V,W);
```

R 中的结果为值

$$1.0 * 2.0 + 2.0 * 3.0 + 3.0 * 4.0 = 20.0$$

注意：函数 INNER 编写的不太好，因为它没有检查 A 和 B 的下标界限是否相等，盲目认为 A 和 B 总是对称的。因为循环变量 I 的取值范围仅考虑到它是 A'RANGE，而没考虑它还可能是 B'RANGE。因此，如果数组 W 的下标界限为 0 和 2，则在执行第三次循环时，便会引起 CONSTRAINT_ERROR。如果数组 W 的下标界限为 1 和 4，虽然不会引起异常，但结果则不是我们所期望的。

最好在调用函数 INNER 时，通过给 B 加以约束来保证 2 个数组的下标界限相同。然而，这是不可能实现的，我们只能在函数体内来检查 2 个数组下标界限是否相等。如果不等，则产生 CONSTRAINT_ERROR。

```
if A'FIRST /= B'FIRST or A'LAST /= B'LAST then
    raise CONSTRAINT_ERROR;
end if;
```

(在第 10 章再详细介绍 raise 语句的用途)。

我们看到，形参可以是无约束数组类型。相似地，函数的结果也可以是数组，只有调用了函数后才能知道其下标界限，结果的类型也可以是无约束数组，其下标界限经相应的 return 语句的表达式求值而得到。

下列函数返回一个向量,其下标界限与参数相同,但向量中的元素是按倒置顺序排列的。

```
function REV(X;VECTOR) return VECTOR is
    R;VECTOR(X'RANGE);
begin
    for I in X'RANGE loop
        R(I):=X(X'FIRST+X'LAST-I);
    end loop;
    return R;
end REV;
```



变量 R 被说明为 VECTOR, 具有与 X 相同的下标界限。要学习怎样使用循环来进行倒置元素的值。结果的下标界限为表达式 R 的下标界限。注意, 我们调用的是函数 REV 而不是 REVERSE, 这是因为 reverse 是保留字。

若函数结果是个复合类型对象(如记录或数组), 则函数调用还可从中选出某一元素返回, 而不必把整个值赋给一个变量。因此我们可以写成

REV(Y)(I)

表示调用函数 REV, 返回数组下标为 I 的元素值。

要注意, 无参函数调用, 象无参过程调用一样, 不带括号。有可能会出现调用带一个参数的函数与带下标地调用无参函数之间的二义性, 但可以用第8章中将要介绍重命函数名, 来排除这种二义性。

练习7.1

1. 写出一个函数 EVEN, 根据 INTEGER 参数的偶奇性返回 TRUE 或 FALSE。
2. 重写阶乘函数, 使参数可以是正数或零, 但不能是负数。记住 FACTORIAL(0) 的值是 1, 使用 4.5 节介绍的子类型 NATURAL。
3. 写一函数 OUTER, 计算 2 个向量的外积, 2 个向量 A 和 B 的外积 C 是一个矩阵, $C_{ij} = A_i \cdot B_j$ 。
4. 写一个函数 MAKE_COLOUR, 参数为类型是 PRIMARY 的数组, 返回类型为 COLOUR 值。请见 6.4 节。检验 $\text{MAKE_COLOUR}((R,Y)) = \text{ORANGE}$ 。
5. 写一个具有类型为 ROMAN_NUMBER 参数的函数 VALUE, 返回结果是与参数对应的整型数值。请见练习 8.2(2)。
6. 写一个函数 MAKE_UNIT, 只有单个参数 N, 返回一个 $N \times N$ 实型矩阵。用该函数来说明一个 $N \times N$ 矩阵。请见练习 8.2(3)。
7. 写一个函数 GCD, 返回 2 个非负数整数的最大公因子。用 Euclid's 算法。
$$\text{gcd}(x,y) = \text{gcd}(y, x \bmod y) \quad y \neq 0$$
$$\text{gcd}(x,0) = x$$

用递归来写该函数, 然后用循环重写一遍。

7.2 运算符

上一节我们已介绍了函数体是以保留字 function 后跟函数名开头的, 在上节的所有例子中, 函数名用的都是标识符。然而, 函数名也可以是字符串。而可做为函数名的字符串, 必须为



下列运算符，并且要给运算符加上双引号。

and	or	xor		
=	<	<=	>	>=
+	-		abs	not
*	/	mod	rem	**

这样，函数给运算符定义了新的含义。例如，我们用运算符来改写上节的 INNER 函数：

```
function"*(A,B:VECTOR) return REAL is
  RESULT,REAL := 0.0;
begin
  for I in A'RANGE loop
    RESULT := RESULT + A(I) * B(I);
  end loop;
  return RESULT;
end **;
```

我们按运算符 * 的一般语法规则来调用该函数，由原来的

R := INNER(V,W);

变为

R := V * W;

由实参 V 和 W 的类型，以及 R 所需的类型来识别 * 的含义。因此在上例中 * 不是整型和实型的乘法。

给予一个操作符多个含义是另一种重载，我们已见过枚举直接量的重载。至此可以说，任何的二义性可用约束来排除。预定义运算符的重载并不是新发明，它存在于过去 25 年，大部分编程语言中，而 Ada 只是能定义一个额外的重载，并且使用了重载这个词。

现在可以看出，所有运算符的预定含义只是相当于有一系列函数，它们的说明如下：

```
function"+"(LEFT,RIGHT;INTEGER) return INTEGER;
function"<"(LEFT,RIGHT;INTEGER) return BOOLEAN;
function"<"(LEFT,RIGHT;BOOLEAN) return BOOLEAN;
```

还有，每次说明一个新类型时，就要建立某些运算符如 = 和 < 的重载。

虽然我们可以给运算符赋予新的含意，但不能改变调用它的语法。函数 "*" 的参数总是二个，并且位置先后次序不能改变。新定义的运算符 + 和 -，根据它是一元运算符，还是二元运算符对其调用，从而决定它有一个还是二个参数。函数 SUM 可以被写为

```
function"+"(A:VECTOR) return REAL is
  RESULT,REAL := 0.0;
begin
  for I in A'RANGE loop
```



```
    RESULT:=RESULT+A(I);
end loop;
return RESULT;
end"+;
```

我们可以写成

```
S:=+V;
```

而不必是

```
S:=SUM(V);
```

函数名为运算符的函数体，经常出现一些使用重载运算符的有趣例子。例如在函数“*”体中含有在 A(I) * B(I) 中使用的 *。当然，这里没有二义性，因为表达式 A(I) 和 B(I) 的类型是 REAL，而新的重载运算只适用于类型 VECTOR。有时，在递归时会产生问题，尤其是如果我们要更换一个运算符现存的含义，而不是加入一个新含义。

除了运算符“=”外，没有关于重载运算符的操作数和结果类型的特殊规则。“<”的新重载不必返回类型为 BOOLEAN 的结果。在第 9 章将给出有关“=”的规则。注意，/=不可被重载，它总是从=得到其含义。

不能赋予“成员测试”in 与 not in 以及“短路形式”and then 与 or else 以新的含义。这就是为什么在 4.9 节中说，它们技术上不属于运算符之列。

最后要注意，由保留字表示的运算符，其字符在字符串中可以是大写或小写。or 的新重载可以被说明为“or”或“OR”甚至“Or”。

练习 7.2

1. 写一函数“<”，对 2 个罗马数字进行运算，对其相应的数量值来进行比较。因此“CCL”<“CCXC”。利用练习 7.1(5) 中的函数 VALUE。
2. 写出函数“+”和“*”，来对二个类型为 COMPLEX 数进行加和乘。请见练习 6.5(2)
3. 写出函数“<”，来测试一个类型为 PRIMARY 的值是否在类型为 COLOUR 的值集中，请见 6.4 节。
4. 写出函数“<=”，来测试一个类型为 COLOUR 的值是否在另一个子集中。

7.3 过程

子程序的另一个形式是过程，过程的调用是一语句。我们已见过许多过程调用的例子，那些都是无参过程，如 WORK, PARTY, ACTION 等。

过程体与函数体非常相似，不同的是：

- 过程是以 procedure 开头的。
- 过程名必须是标识符。
- 过程不返回结果。
- 参数必须是三种方式 in, out 或 in out 中的一种。

参数方式是由参数规范式说明中冒号后的 in, out 或 in out 指出的。如省略方式，则被认为是在 in 方式。在函数中，只准许参数为 in 方式，在本章以前的例子中，函数规范式说明都省略了 in，但可以写成

```
function SQRT(X;in REAL) return REAL;
function "*" (A,B;in VECTOR) return REAL;
对三种方式归纳如下(见 LRM 中的 6.2 节)。
```

- in 形参是常量,只能读相应的实参的值。
- in out 形参是变量,可读和可修改相应的实参值。
- out 形参是变量,可改变相应的实参值。

in 和 out 方式的简单例子为:

```
procedure ADD(A,B;in INTEGER;C;out INTEGER) is
begin
    C:=A+B;
```

调用形式可为

```
P,Q;INTEGER;
```

...

```
ADD(2+P,37,Q);
```

调用过程 ADD 时,计算表达式 $2+P$ 和 37 (先计算哪个表达式不确定),并把值赋给作为形参的常量 A 和 B, $A+B$ 的值赋给形参 C, 返回 C 的值, 赋给变量 Q。这相当于

```
declare
    A;constant INTEGER:=2+P;           --入
    B;constant INTEGER:=37;            --入
    C;INTEGER;                         --出
begin
    C:=A+B;                           --一体
    Q:=C;                            --出
end;
```

下列为 in out 方式的例子:

```
procedure INCREMENT(X;in out INTEGER) is
begin
    X:=X+1;
```

end;

可有

```
I;INTEGER;
```

...

```
INCREMENT(I);
```

调用 INCREMENT 时, I 的值赋给形参变量 X。然后 X 加1。返回时, X 的最终值赋给实参 I。这就象



```
declare
  X:INTEGER:=I;
begin
  X:=X+1;
  I:=X;
end;
```

对于任意的纯量类型(如:INTEGER),这些方式仅相当于:in 方式在调用时拷贝实参值至形参中,out 方式在返回时拷贝形参值到实参中,而 in out 则是上述两种情况的综合。如果是 in 方式,则实参可以是适当类型和子类型的任意表达式。如果是 out 或 in out 方式则实参必须是变量。这种变量自身(是指变量的地址,不是指其值)在过程调用时确定,在返回时仍把结果返回给该变量。假设有:

```
I:INTEGER;
A;array(1..10) of INTEGER;
procedure SILLY(X,in out INTEGER) is
begin
  I:=I+1;
  X:=X+1;
end;
```

则执行语句

```
A(5):=1;
I:=5;
SILLY(A(I));
```

后 A(5) 中的值为 2,I 变为 6,但 A(6) 不受影响,其值不变。若参数是复合类型(如:数组或记录),则可采用如上述的拷贝机制,而实际上也可采用访问机制,即形参直接访问实参。因此,编程时,复合类型的形参不能依赖于特定的信息传递机制,否则程序就是不正确的。在本节的练习中给出了这样的程序例子。注意,因为形参数组从实参数组中得到其下标界限,故总是在调用时把下标界限加以拷贝,即便对 out 参数亦如此。当然为了简单起见,实际上总是把整个数组加以拷贝。

我们现在来讨论关于参数的约束问题。

在纯量参数的情况下,对于 in 或 in out 方式的参数,在子程序调用开始时,实参之值必须满足对应形参变量的约束。而对于 out 方式的参数,从子程序返回时形参之值必须满足对应实参变量的约束。数组类型的情况则有些不同。若形参是约束数组类型,则实参的下标界限必须与之一致。这就是说只是每维的元素个数相同是不够的,还要求下标界限完全相同。可见,参数传递机制要比赋值机制更严格。另一方面,如果形参是无约束数组类型,则参数传递时形参数组的下标界限取决于实参的下标界限。此规则也适用于函数的结果返回,若结果是受约束数组类型,则 return 语句中的表达式的下标界限必须与之一致。另外,因为参数和结果的传递机制比赋值语句更为严密,从而准许含有 others 的数组聚集做为实参或出现在 return 语句中。而

正如在6.2中看到的，它们一般不准许出现在赋值语句中，除非受到约束。对于我们讨论过的简单记录，不存在约束问题，不在此赘述。对于其它类型的参数机制将在介绍该类型时再讨论。

我们前面指出，对应于out或in-out形参的实参必须是变量。这也包括实参可能是某些外层子程序的形参的情况。但是，作为in-out形参的变量就不能再作为out实参，否则，可能会产生读out参数。如上所述，与out或in-out方式形参对应的实参必须是变量，但必要时可以对实参变量的类型进行转换。例如，数值类型之间可以转换，因此可有

```
R:REAL;  
...  
INCREMENT(INTEGER(R));
```

如果R的初始值是2.3，则转换为整型数2，然后加1，结果为3，返回时再转换为3.0，并赋给R。

in-out和out参数的类型转换对于数组尤其有用。假如我们写了一个子程序，用于类型VECTOR，而从别人那里知道有些6.2节中的子程序用于类型ROW。类型ROW和VECTOR基本上相同，只是程序员用了不同的名称而已，数组类型转换可使我们在不必全面地修改类型名的情况下使用这两个子程序。

考虑下例：

```
procedure QUADRATIC(A,B,C,in REAL;ROOT_1,ROOT_2,out REAL;  
                      OK,out BOOLEAN) is  
  D:constant REAL:=B * * 2 - 4.0 * A * C;  
begin  
  if D<0.0 or A=0.0 then  
    OK:=FALSE;  
    return;  
  end if;  
  ROOT_1:=(-B+SQRT(D))/(2.0*A);  
  ROOT_2:=(-B-SQRT(D))/(2.0*A);  
  OK:=TRUE;  
end QUADRATIC;
```

过程QUADRATIC可解方程

$$ax^2 + bx + c = 0$$

如果是实根，则由参数ROOT_1和ROOT_2返回根的值，并且OK被置为TRUE。如果是复根($D < 0.0$)或方程降阶($A = 0.0$)，则OK被置为FALSE。

要注意return语句的使用。因为过程没有结果需要返回，因此return后不能跟表达式。它只是按需要来修改对应形参方式为out和in-out的实参，并把控制权返回调用此过程的地方。还要注意，过程不像函数，它可以运行到end，这就相当于执行return。

读者会注意到，如果OK被置为FALSE，则没有给out参数ROOT_1和ROOT_2赋值。根据纯量的拷贝规则，对应的实参被强制为无定义的。实际上，是把一个任意的值赋给实参。这样，如果实数是受约束的，则可能会引起CONSTRAINT_ERROR，因此最好是先把根赋为安全

值,如0.0,以防万一。(在这个例子中,out机制不象Algol 68或Pascal中的简单访问机制那样令人满意。)

在下列程序中调用过程 QUADRATIC

```
declare
  L,M,N:REAL;
  P,Q:REAL;
  STATUS:BOOLEAN;
begin
  ——给 L,M 和 N 赋值
  QUADRATIC(L,M,N,P,Q,STATUS);
  if STATUS then
    ——根值在 P 和 Q 中
  else
    ——失败
  end if;
end;
```

在本节结尾时,我们再强调一下,out参数不能被看成是一个真正的变量,因为不能读它。上例的程序不能改为

```
begin
  OK:=D>=0.0 and A/=0.0;
  if not OK then
    return;
  end if;
  ROOT_1:="";
  ROOT_2:="";
end QUADRATIC;
```

因为布尔表达式 not OK 要读 out 参数 OK 的值。

有一例外,尽管不能读 out 数组的元素值,但可以读其下标界限。记住,形参数组总是从实参数组中得到其下标界限,对于记录也是同样,我们在第11章会遇到。

练习 7.3

1. 写一个过程 SWAP,来交换二个实型参数的值。
2. 改写7.1节中的函数 REV 为一过程,具有一个参数,用于倒置类型为 ROW 的数组 R。
3. 试证明下列程序是错的:

```
A:VECTOR (1..1);
procedure P(V:VECTOR) is
begin
```

```

A(1):=V(1)+V(1);
A(1):=V(1)+V(1);
end;
...
A(1):=1.0;
P(A);

```

7.4 命名和隐含参数

到目前为止我们调用子程序的格式，都是按位置顺序给出各个实参。如同使用聚集，在子程序调用时，我们也可以使用命名方式，即在调用格式中给出形参名，从而实参就不必按顺序排列。

于是可以写成：

```
QUADRATIC (A=>L,B=>M,C=>N,ROOT_1=>P,
            ROOT_2=>Q,OK=>STATUS);
```

```
INCREMENT(X=>I);
```

```
ADD(C=>Q,A=>2+P,B=>37);
```

甚至可以写成：

```
INCREAMENT(X=>X);
```

这种方式也可以用于函数调用。

```
F:=FACTORIAL(N=>4);
```

```
S:=SQRT(X=>T+0.5);
```

```
R:=INNER(B=>W,A=>V);
```

命名方式不能用于内嵌于操作分量间以运算符为函数名的函数调用。

象记录聚集一样，在子程序调用时，命名和位置形式可以混用，位置参数必须在前面，并且顺序必须正确。然而不象记录聚集那样，在子程序调用时，必须单个地给出每个实参，且不能使用 others，因此可以写成

```
QUADRATIC(L,M,N,ROOT_1=>P,ROOT_2=>Q,OK=>STATUS);
```

采用命名方式可使我们能运用隐含参数。对于某个或多个 in 参数，在每次调用时其值都相同时，我们就可以在子程序中给出其隐含表达式，而在调用时就可以省略。

我们来考虑买马丁尼酒的问题。买酒人面临几种选择，由枚举类型描述如下：

```
type SPIRIT is (GIN,VODKA);
```

```
type STYLE is (ON_THE_ROCKS,STRAIGHT_UP);
```

```
type TRIMMING is (OLIVE,TWIST);
```

在过程规范式说明中，可给出其标准隐含表达式：

```
procedure DRY_MARTINI (BASE:SPIRIT:=GIN,
                        HOW:STYLE:=ON_THE_ROCKS,
                        PLUS:TRIMMING:=OLIVE);
```

典型的调用为

```
DRY_MARTINI(HOW=>STRAIGHT_UP);
```



```
DAY_MARTINI(VODKA,PLUS=>TWIST);  
DAY_MARTINI;  
DAY_MARTINI(GIN,STRAIGHT_UP);
```

第一个调用语句,用的是命名方式。我们得到杜松子酒、不加冰、加橄榄。第二个调用语句是位置和命名混合方式。当开始省略参数之后,参量就必须用命名的方式了。第三个调用语句表示可以省略所有参量。最后一个调用语句表示,参量值可以与隐含表达式的值相同,在这里可以避免第二个参量用命名方式。

注意,只能给 in 参数设隐含表达式。不能给函数名为运算符的函数设隐含表达式,但是可以给函数名是标识符的函数设隐含表达式。这里的隐含表达式(象记录类型说明中赋初值的隐含表达式一样)在需要时才计算,即在每次调用子程序并且没有对应实参时才计算。而且在每次调用时隐含值有可能不同。当然,一般情况不是相同的。隐含表达式广泛地用于标准输入/输出,以提供隐含格式。

隐含表达式说明了这样一个准则,参数的规范式说明形式

```
P,Q,in INTEGER:=E;
```

严格等同于:

```
P,in INTEGER:=E;  
Q,in INTEGER:=E;
```

(读者会想到在对对象进行说明时,有相似的规则,而且该规则也适用于记录元素。)在调用子程序时,为缺省实参的参数计算隐含表达式,在一般情况下是没问题的,但是如果表达式 E 中含有带副作用的函数时就可能会产生问题。

练习 7.4

1. 写一个函数 ADD,返回两个整数之和,并且第2个参数的隐含值是1.有多少种方法来调用该函数,使返回的结果是 N+1?这里 N 是第一个实参。
2. 重写 DRY_MARTINI 的规范式说明,来反映你在周末喜欢 VODKA. 提示:说明一个函数,按照全称变量 TODAY 返回你所喜欢的酒。

7.5 重载

我们在7.2节中见过,怎样给现存的运算符赋以新的含义。一般而言,重载也适用于子程序。

子程序可以在原意义上进行重载,而不是覆盖原意义。要实现之,子程序间的规范式说明必须不同。过程不能覆盖函数,反之亦然。注意,子程序重载时,参数的名称,方式和是否有隐含表达式都无关。可以在同一说明部分来说明多个重载子程序。

子程序与枚举直接量可以相互重载。事实上,可以认为枚举直接量是结果为枚举类型的函数。标识符可被分为二类,一类是不可重载的,另一类是可重载的。不论在何处,标识符或者是第一类,即表示一个不可重载的单个对象,或者是第二类,即用于一个或多个重载。说明标识符为某一类后,则它就不可再被说明为另一类,即不能在同一说明部分将某标识式符说明为不同类。

正如我们所看到的,重载所引起的二义性可用约束来排除。在8.3节中当把运算符“<”用

于字符串时就已需要加入约束。再请看有关一个英吉利海峡上岛屿的例子，最大的三个岛是 Guernsey, Jersey 和 Alderney。每个岛都出产羊毛。

```
type GARMENT is (GUERNSEY, JERSEY, ALDERNEY);
```

其中的二个岛出产牛 (Alderney 在第二次世界大战后就不再出产家畜了)。

```
type COW is (GUERNSEY, JERSEY);
```

我们可以设想出售羊毛和牛的商店

```
procedure SELL(G; GARMENT);
```

```
procedure SELL(C; COW);
```

则

```
SELL(ALDERNEY);
```

没有二义性，而

```
SELL(JERSEY);
```

则有二义性，因为我们不知道应该调用那个子程序，我们必须写成

```
SELL(COW'(JERSEY));
```

练习 7.5

1. 还可以怎样使 SELL(JERSEY) 不产生二义性？

7.6 说明、范围和可见性

以前我们说过，有时，需要只给出子程序的规范式说明而不给出子程序体。规范式说明之后紧跟分号的形式为子程序说明，而完整的子程序（亦可含有规范式说明）被称为子程序体。

子程序说明和子程序体都必须象其它说明一样位于程序的说明部分，并且子程序说明后必须在程序同一说明部分里跟有相应的子程序体。

在相互递归调用的子程序中就要用到子程序说明。例如，我们说明二个过程 F 和 G，它们相互调用。依据说明是线性确立的这一原则，我们只是在 F 被说明后才能在 G 体中调用 F，反之亦然。显然，如果我们只写子程序体是不可能实现上述调用的，因为总有一个在前，一个在后，然而，我们可以写成：

```
procedure F(...);           --F 的说明
procedure G(...) is         --G 的体
begin
  F(...);
end G;

procedure F(...) is           --F 的体，重复其规范式说明
begin
  G(...);
end F;
```

这样一切问题都解决了。

如果规范式说明重复出现，则规范式说明都必须是完整的，并且两者必须一致。也准许有些小的差异，如一个数值直接量可以被另一个相同值的数值直接量所取代。在语法上空格可以不同。值得注意的是，不准许在参数表中使用子类型指示。这是为了避免在一致性上遇到问题。因为这样就不致于出现计算二次约束表达式，也就不会因副作用而使表达式的二次计算结果不同。对于隐含表达式没有相应的问题（可以出现二次），因为只有在调用子程序时才计算其值。

另一个重要点是，应象写子程序体一样来写子程序说明。这将在下一步讨论程序包时见到。然而，并不是总需要这样，但象体一样来写子程序的说明会使人感到很清楚。例如许多子程序在一起出现时，子程序说明就好象是摘要。

子程序体和其它的说明不能任意混杂，体应跟在说明之后，以保证在“大”的子程序体之间不会丢掉“小”的说明。

因为子程序出现在程序的说明部分，子程序本身的规范式说明部分又嵌在被说明的子程序中。因此，4.2节中有关分程序的隐蔽规则都适用于子程序中的说明。例如：

```
procedure P is
    I; INTEGER := 0;

    procedure Q is
        K; INTEGER := I;
        I; INTEGER;
        J; INTEGER;
    begin
        ...
    end Q;
    ...
end P;
```

正如4.2节中的例子，内层的I屏蔽了外层I，因为，在过程Q内说明了内层的I之后外层的I就不再是可见的了。

然而，我们总可以用加点表示法来使用外层的I，即在I前加上含有说明I的那个单元名后跟一点。例如在Q中，我们用P.I来引用外层的I。因此，我们可以这样对J进行初始化。

```
J; INTEGER := P.I;
```

如果加点的前缀是自我屏蔽，则可按相同方式来书写，可以用P.Q.I来引用内层的I。

在分程序中一般不能按这种方式对对象进行说明。因为，一般分程序没有名。然而，可以象下例所示那样，给分程序用象给循环命名相似的方法来命名。

```
OUTER;
declare
    I; INTEGER := 0;
begin
```

```

...
declare
  K:INTEGER := I;
  I:INTEGER;
  J:INTEGER := OUTER.I;
begin
...
end;
end  OUTER;

```



这里外层分程序的名为标识符 OUTER,与子程序不同,该标识符必须在对应的 end 后重复出现。给分程序命名,使我们可以用外层的 I 为内层的 J 在说明时赋初值。

在循环中还可以用相同的方法来引用被屏蔽的循环参数。我们可以改写 6.1 节中给 AA 的元素赋 0 的例子:

```

L:
for I in AA'RANGE(1) loop
  for I in AA'RANGE(2) loop
    AA(L,I,I) := 0.0;
  end loop;
end loop L;

```

然而这种写法是不好的。

这种表示方式也适用于操作数和字符直接量,如在函数" * "中说明了变量 RESULT,可以用" * ".RESULT 来引用。同样,如果" * "是在分程序 B 中被说明的,则可用 B." * "来引用它。如果调用函数" * ",则必须按一般的函数调用形式:

R := B." * "(V,W);

实际上,总是可以用一般函数调用形式:

R := " * "(V,W);

正如我们所看到的,子程序可以修改全程变量,然而这会产生副作用(此处产生的副作用,不是因参数传递机制而引起的)。一般认为,具有副作用的子程序,尤其是函数是不好的。然而,任何实现输入/输出的子程序都对文件具有副作用。一个要产生多个随机函数序列的函数,只能靠修改全程变量之副作用来完成,等等。在使用含副作用的函数时,必须小心。因为在许多情况下,求值顺序不确定。

我们来简要讨论一下与 exit,return 有关的层次,与 goto 有关的分程序作用域以及与循环标识符、标号有关的范围,来作为本节的结尾。

return 语句立即终止包含它的最内层子程序的执行。它可放在子程序内的分程序或循环中,如在循环中执行 return 语句,则同时终止循环,exit 语句可以出现在循环体内的分程序中,但不能出现在分程序内的子程序体中,否则执行 exit 语句也终止此子程序.goto 语句可以把控制权转到循环或分程序外,但不能是子程序外。

就作用域而言,可将分程序和循环的命名标识符(即标号)看作是在包含它们的最内层子程序,分程序,程序包或任务体的说明部分之末处被说明的。因此,同一分程序内的两个循环不能有同样的命名标号。同一子程序中的两个标号,即使分别在不同层次的分程序内也不能同名。此规则降低了 goto 语句转到错误标号处的危险性,对修改程序也尤为重要。

要点7

- 必须用类型标志给出参数和结果的类型,不能用子类型指示。
- 函数必须返回一个结果,并且不能运行到最后的 end,而过程则可以。
- 先计算实参表中哪个实参的顺序不确定。
- 只有在调用子程序,并且相应的实参缺省时,才计算隐含表达式。
- 约束数组类型的实参和形参的下标界限必须一致。
- 复制纯量参数,但没有定义数组和记录的参数传递机制。
- 用分号来分隔形参规范式说明,而不是用冒号。

第8章 整体结构

在前面的章节中,我们相当详细地叙述了有关 Ada 的小规模特征。至此,Ada 所呈现的领域对应于 60 年代和 70 年代早期语言的相应领域,只是在这些领域,Ada 提供了更多的功能。现在让我们进入属于 Ada 的新领域,即数据抽象概念和第 1 章中提及的大规模编程。

为此,本章讨论程序包(Ada 总是涉及程序包)和分别编译机制。此外,再谈谈作用域和可见性。

8.1 程序包

传统的分程序结构语言,如 Algol 和 Pascal,其主要问题是不能提供严格的可见性控制。例如,有一用数组表示的栈和一指示栈顶元素的变量,以及往栈顶增加元素的过程 PUSH 和用来移去栈顶元素的函数 POP。其说明为:

```
MAX;constant:=100;
S;array (1..MAX) of INTEGER;
TOP;INTEGER range 0..MAX;

procedure PUSH(X;INTEGER) is
begin
    TOP:=TOP+1;
    S(TOP):=X;
end PUSH;

function POP return INTEGER is
begin
    TOP:=TOP-1;
    return S(TOP+1);
end POP;
```

这种情况下,一般存在一个正确的编程协定,即只允许访问已说明的用来对某结构进行操作的子程序,而不允许直接访问表示该结构的有关变量。但是简单的分程序结构语言无法保证我们一定会遵循这个正确的编程协定,因为我们既可访问子程序 PUSH 和 POP,也可访问变量 S 和 TOP 而直接使用栈的表示结构。

Ada 程序包允许我们对一组说明设置一个界来克服上述问题,即只允许访问那些我们让其可见的部分。程序包实际上由两部分组成,即给出了可见部分的规范式说明和给出了隐藏细节的包体。

上述例子可写成:

```

package STACK is
    procedure PUSH(X;INTEGER);
    function POP return INTEGER;
end STACK;

package body STACK is
    MAX,constant:=100;
    S,array(1..MAX) of INTEGER;
    TOP,INTEGER range 0..MAX;
    procedure PUSH(X;INTEGER) is
    begin
        TOP:=TOP+1;
        S(TOP):=X;
    end PUSH;

    function POP return INTEGER is
    begin
        TOP:=TOP-1;
        return S(TOP+1);
    end POP;

begin
    TOP:=0;
end STACK;

```



程序包规范式说明以保留字 package、程序包标识符和 is 开头。接下来是可见实体的说明。最后以 end、程序包标识符和终止分号结尾。本例中，它只含有子程序 PUSH 和 POP 的说明。

程序包体也以 package 开头，但紧跟着保留字 body，然后才是程序包标识符和 is。接下来是常规说明部分、begin、语句序列、end 及包标识符和分号（这些就象是一个分程序或子程序体）。本例中，常规说明部包括栈中用到的变量和 PUSH,POP 的体。begin 与 end 之间的语句序列将在定义程序包时执行而用作初始化。如果不需要初始化，那么该语句序列可以省去。实际上，本例中的初始化可以更简便地写到说明中

```
TOP,INTEGER range 0..MAX:=0;
```

这里，程序包本身亦看作为说明，所以在外层说明部中它仅为一项说明（假设它不是最外层程序单位）。

程序包要求区分子程序说明与子程序体。实际上，我们不能把某个体放在程序包规范式说明中。如果程序包规范式说明包含某子程序的规范式说明，那么程序包体就必须包含相应的子程序体。子程序体可以在程序包体中说明而无须要求其规范式说明在程序包规范式说明中给出，但这样的子程序仅在程序包内被调用，或被程序包中的其它子程序（它们也许可见）调用，或被初始化序列调用，而不能从程序包外调用。程序包规范式说明和包体可认为是只有某些项

可见的一个大说明部。

程序包体的确立,是指其说明部的确立和接着初始化序列的执行(如果有的话)。程序包一直存在到说明它的作用域结束。程序包中说明的实体与程序包有相同的生命期限。例如,变量 S 和 TOP 可看作如 Algol 60 中的“ own” 变量,它们的值在 PUSH 和 POP 的接连调用之间被保留。

程序包可在分程序、子程序,甚至另一程序包的说明部中被说明。假如某程序包的规范式说明在另一程序包的规范式说明中被说明,那么,与子程序的情形相同,其体也必须在那个程序包的体中说明。此外,程序包规范式说明和体两者也可同被说明在另一程序包的体中。

程序包规范式说明除了不包含程序体外,能包含我们已知的任何其它说明。

让我们再回到程序包的使用上。程序包本身有一个名,在某些情形下,可见的实体被认为是程序包的分量。因此,为了调用 PUSH,我们必须涉及 STACK,这只需采用加点引用方式如下:

```
declare
  package STACK is           ——规范式说明和体
  ...
  ...
end STACK;
begin
  ...
  STACK.PUSH(M)
  ...
  N:=STACK.POP;
  ...
end;
```

程序包内我们能用 PUSH 来调用 PUSH 过程,也可用 STACK.PUSH 来调用之(这正如前一章我们看到的,把过程 P 的局部变量 X 表示为 P.X 那样)。程序包内我们能引用 S 或 STACK.S 变量,但在程序包外,MAX,S 和 TOP 都不能以任何途径被访问到。

一般来讲,在程序包外频繁地以 STACK.PUSH 来调用 PUSH 过程显得很麻烦,这时我们可以写一个短说明

```
use STACK;
然后再直接对 PUSH 和 POP 加以调用。use 子句可在同一说明部或其它可见说明部中,跟在
```

```
...
N:=POP;
...
end;
```

use 子句类似一个说明,其作用域至分程序结尾。在其作用域之外,我们仍要采用加点引用方式。(在程序包内部用 use 子句指示本程序包不会造成危害,但却多余。)

两个或更多的程序包可被说明在同一说明部。一般来说,我们可将所有的规范式说明放在一处,然后将所有的体另放一处,或者交替地将相对应的规范式说明和体放在一起。例如,我们可以有:A 规范式说明, B 规范式说明, A 体, B 体。或者: A 规范式说明, A 体, B 规范式说明, B 体。

若在同一说明部中说明多个程序包(或子程序),存在一个简单的顺序规则,即:

- 线性排列说明部;
- 程序包(或子程序)的规范式说明必须排在其体之前;
- 较小说明项一般排在较大说明项之前。

最后这条规则是指一个体之后仅能跟其它的体、子程序、程序包(或任务)的规范式说明或 use 子句。

程序包可见部可不包含子程序而仅包含一组相关的变量、常量和类型。这时,程序包不需要包体,它不提供任何隐藏特性而仅赋予名以公共特性。(也许提供一个包体,但它仅仅是为了初始化。)

作为例子,我们给出一个仅包含类型 DAY 和一些有用的相关常量的程序包:

```
package DIURNAL is
    type DAY is (MON,TUE,WED,THU,FRI,SAT,SUN);
    subtype WEEKDAY is DAY range MON..FRI;
    TOMORROW:constant array (DAY) of DAY:=
        (TUE,WED,THU,FRI,SAT,SUN,MON);
    NEXT_WORK_DAY:constant array (WEEKDAY) of
        WEEKDAY:=(TUE,WED,THU,FRI,MON);
end DIURNAL;
```

最后一点,说明部构造中不能调用其体尚未出现的子程序。但这并不阻止它们相互递归调用,仅阻止将它们用于初始化。因为递归调用实际上是在执行语句序列时才发生。故

```
function A return INTEGER;
I:INTEGER:=A;
```

是非法的,结果引发 PROGRAM_ERROR 异常。

这一规则也适用于程序包中的子程序。程序包体构造完后,我们才能从程序包外调用其中的子程序。

练习 8.1

1. 一个大致的伪随机数序列可由 $X_{n+1} = X_n * 5^{\text{MOD} 2}$ 产生。初始值 X_0 为 $0..2^n$ 范围中的某奇数。编写程序包 RANDOM 使其包含用作该序列初始化的过程 INIT 和给出该序列下一个值的函数 NEXT。
2. 编写程序包 COMPLEX_NUMBERS，其可见部包含：
 - 类型 COMPLEX；
 - 常量 $i=\sqrt{-1}$ ；
 - 对 COMPLEX 类型值操作的函数 +, -, *, /, 参见练习 7.2(2)



8.2 库程序单位

以往许多语言都忽视了一个简单事实：程序是分段编写，分别编译，然后再组合在一起的。Ada 认识到分别编译的重要性，并且提供了两种分别编译机制——自顶向下机制和自底向上机制。

自顶向下机制适用于开发大型的相关程序。由于诸多原因，这些程序需要分离成一组能分别编译的程序子单位，这些子单位须在它们的父程序单位之后被编译。这个机制留待下节叙述。

自底向上机制则适用于构造程序库。程序库中的程序单位是为通用而编写的，因此它们必须在引用它们的程序单位之前编写好。现在便详细讨论这一机制。

库程序单位可以是子程序规范式说明或程序包规范式说明，其相应的体则称为次级程序单位。这里程序单位可以分别单独编译或为方便起见而集中编译。例如，我们可以集中编译某程序包的规范式说明和体，但正如我们将看到的那样，分别单独编译也许更有利。通常子程序体本身就足以完全定义这个子程序，这时它被看作是一个库程序单位而不仅是次级程序单位。

程序单位编译后即融进一个程序库。显然，依照用户和课题的需要，可以有许多这样的程序库。（关于程序库的创建和操作，本书不作讨论）。一旦入库，只须以 with 子句说明依赖关系，该程序单位便能被随后编译的任何程序单位使用。

假设我们编译程序包 STACK，这个程序包不依赖于其它程序单位，所以不需要 with 子句。为了集中编译其规范式说明和体，提交编译的文本是：

```
package STACK is
  ...
end STACK;
package body STACK is
  ...
end STACK;
```

现在假设我们编写过程 MAIN，它要使用程序包 STACK。过程 MAIN 是通常意义上的主程序，它没有参数。（Ada 并未规定主程序应有标识符 MAIN，这里采用仅仅是为了符合我们的习惯。）

此时提交编译的文本可以是：

```
with STACK;
```

```
procedure MAIN is
    use STACK;
    M,N;INTEGER;
begin
    ...
    PUSH(M);
    ...
    N:=POP;
    ...
end MAIN;
```

with 子句出现在程序单位前。这样，该程序单位对其它程序单位的依赖便一目了然。with 子句不能嵌入到某内部作用域中。

如果某程序单位依赖若干个其它的程序单位，那么能用同一个 with 子句表明，或用不同的 with 子句表明。例如，我们可以写：

```
with STACK,DIURNAL;
procedure MAIN is
    ...
```

或：

```
with STACK;
with DIURNAL;
procedure MAIN is
    ...
```

为了方便，我们可在 with 子句后写上 use 子句。例如：

```
with STACK;use STACK;
procedure MAIN is
    ...
```

这样 PUSH 和 POP 便直接可见而不必加点引用。

只有直接依赖的程序单位才需要用 with 子句给出。因此，如果程序包 P 使用了程序包 Q 的设施，Q 转而又使用了程序包 R 的设施，那么，若非 P 直接使用了 R 的设施，P 的 with 子句仅需写上 Q。P 的使用者不用担心也无需知道 R 的情况，否则导出的层次会变得相当复杂。

另外，程序包或子程序说明部前的 with 子句同样适用于相应的体，这里不再赘述。当然，体也可以有用 with 子句表明的附加依赖。但体所依赖的程序单位不应以规范式说明给出，否则体和规范式说明的独立性会被减弱。

如果程序包的规范式说明和体被分别编译，那么其体必须在规范式说明之后被编译。我们说：程序包体依赖于程序包规范式说明。然而，任何使用程序包的程序单位只依赖于程序包规范式说明而不依赖于程序包体。如果程序包体作了更改而其规范式说明并未变动，那么任何使用该程序包的程序单位就不需要重新编译。分别编译规范式说明和体可简化程序的维护。

有关编译顺序的一般规则很简单，某程序单位必须在它所依赖的所有程序单位之后被编

译。因此，假如某程序单位被改变和重新编译，那么所有依赖于它的程序单位均须重新编译。满足这一依赖规则的任何编译顺序皆是可行的。

标准程序包 STANDARD 不需要以 with 子句表明。它有效地包含诸如 INTEGER 和 BOOLEAN 的所有预定义类型的说明。它还包含一个内部程序包 ASCII。ASCII 定义了诸如 CR 和 LF 这样的控制字符常量。至此，我们便弄清楚了第 6 章中，控制字符为什么被表示成 ASCII. CR 等。当然，写上 use ASCII 子句后，它们便能表示成 CR 等了。程序包 STANDARD 在附录 3 中有详细的描述。

最后，有一条关于库程序单位的非常重要的规则，即所有库程序单位必须有各自的标识符，而且不能重载，也不能是运算符。

练习 8.2

1. 程序包 D 和子程序 P,Q,MAIN 有如下的 with 子句：

D 规范式说明	无 with 子句
D 体	with P, Q;
P 子程序	无 with 子句
Q 子程序	无 with 子句
MAIN 子程序	with D;

画出各程序单位之间的依赖关系图。问：这里存在多少可行的编译顺序？

8.3 子单位

本节我们介绍次级程序单位的进一步形式——子单位。程序包、子程序（或第 14 章中的任务）的体能够从包容它的库程序单位或次级程序单位中分离出来单独编译，而包容程序单位中的其相应体则由体残根替代。假设我们从程序包 STACK 中移出子程序 PUSH 和 POP 的体，那么 STACK 的体将变成：

```
package body STACK is
    MAX:constant:=100;
    S:array (1..MAX) of INTEGER;
    TOP:INTEGER range 0..MAX;
    procedure PUSH(X:INTEGER) is separate;      -- 残根
    function POP return INTEGER is separate;      -- 残根
begin
    TOP:=0;
end STACK;
```

被分离出的程序单位称为子单位，子单位可以分别单独编译，但它们必须以 separate 跟上其父程序单位名开头。因此子单位 PUSH 即为：

```
separate(STACK)
procedure PUSH(X:INTEGER) is
begin
```

```
TOP := TOP + 1;  
S(TOP) := X;  
end PUSH;
```

子单位 POP 的情况也类似。

上例中的父程序单位为库程序单位(体)。其实,父程序单位本身也可以是子单位,这时,父程序单位名必须是以其祖先库程序单位名开头并用加点引用方式给出的全名。因此,假如 R 是 Q 的子单位,而 Q 又是 P 的子单位,P 为库程序单位,那么 R 的文本开头必须是:

```
separate(P.Q)
```

因与库程序单位有关,所以子单位必须有各自可区分的标识符。当然这并不妨碍不同程序单位的子单位有相同的标识符。另外,子单位的标识符不能是运算符。

子单位依赖于它的父程序单位(任何库程序单位皆被显式提及),所以子单位必须在其父程序单位之后被编译。

子单位中的可见性即为相应的体残根处的可见性——确切地说,子单位似乎是从其所处环境中被毫无损伤地拔出来的。因此,任何适用于父程序单位的 with 子句都不再重复,因为子单位是分别单独编译的。然而,在子单位开头还可给出其自用的 with 子句(也可是 use 子句)以便访问其附加的库程序单位。这些子句须放在 separate 前,故 R 的文本可以这样开头

```
with X; use X;  
separate(P.Q)
```

...

这样做的原因是:我们可以从父程序单位 P.Q 中移去任何对库程序单位 X 的引用,从而减少一些依赖。这在编译顺序上将给我们更多自由。例如,由于某些原因 X 被重新编译,这时仅 R 需要重新编译,而无须对 Q 重新编译。

注意:with 子句仅用于指示库程序单位而不能用来指示子单位。最后应清楚,若干个子单位或库程序单位、库程序单位体及子单位的组合都能集中编译。

练习 8.3

1. 假设程序包 STACK 有分离出去的子单位 PUSH 和 POP,试画出下面五个程序单位之间的依赖关系图:过程 MAIN, 过程 PUSH, 函数 POP, 程序包规范式说明 STACK, 程序包 STACK 的体, 并指出有多少可能的编译顺序。

8.4 作用域和可见性

现在再讨论作用域和可见性问题。本节我们着重讨论与 Ada 程序有关的一些主要问题,其有关细节,读者可参阅语言参考手册 LRM。(LRM 利用项定义域来解释可见性和作用域规则)。关于分程序和子程序的定义域及有关作用域规则,在 4.2 节和 7.6 节中已有描述。现在我们考虑的是程序包引入后的影响。

程序包的规范式说明和体共同组成一个定义域。因此,如果我们在规范式说明中说明了变量 X,就不能再在体中重复说明 X(当然象局部子程序那样的内部域除外)。

程序包可见部中说明的作用域从说明开始,延伸至程序包本身作用域的结尾。注意:如果程序包处在外层程序包的可见部中,则它说明的作用域将延伸到外层程序包作用域的结尾(这

时程序包被看作是外层程序包可见部中的一项说明),相似情况可类推。

程序包体中的说明(或下章将描述的程序包私有部中的说明)的作用域伸展到包体末尾。

简单嵌套的分程序和子程序中,如果实体未被另一说明隐藏,那么它在其整个作用域内都可见。在7.8节中我们曾看到,一般来讲,即使实体被隐藏,它仍能以加点引用方式访问到。这时加点的前缀名即为包含该说明的程序单位的名。原则上,采用加点引用方式便能使程序单位中的实体可见。

在程序包内部,对于程序包中说明的实体,同样的规则也适用。但在程序包外,如果我们不写上程序包名或者写上 use 子句,这些实体便不可见。

程序单位中某处可见的标识符是指那些不考虑任何 use 子句就可见的标识符和那些使用了 use 子句而成为可见的标识符。

可见性的基本规则为:某程序包内的标识符能通过 use 子句的引进而成为可见,条件是它们不得与其它 use 子句引进的另外程序包中的标识符重名,也不能与以任何其它方式已成为可见的标识符重名。如果这些条件不满足,那么这个程序包中的标识符就不可见,而且我们不得不仍然以加点引用方式去使用它们。

如果标识符是子程序或枚举直接量,那么可见规则稍有不同。在这种情况下,除非规范式说明发生冲突(相互隐藏)而不可见,它们都可相互重载。因此,借助 use 子句而成为可见的标识符,虽然可以重载但决不能隐藏另一标识符。

这些规则总的目的在于确保使用 use 子句后不会使文本的已存部分失效。此外,Ada 编译对可见性给予了支持,它会指出任何二义的问题。而且加上有关限制或运用加点引用方式有助于保证一切正常。

另有一些有关记录分量名、子程序参数等的规则。例如,记录分量标识符与该标识符名在本记录类型定义之外的另外一次引用之间不存在冲突,原因是它们的作用域可重叠,但其可见性不会重叠。考虑下面例子:

```
declare
  type R is
    record
      I:INTEGER;
    end record;
  type S is
    record
      I:INTEGER;
    end record;
  AR:R;
  RS:S;
  I:INTEGER;
begin
  ...
  I:=AR.I+AS.I;           --合法
```

...
end;

类型 R 中 I 的作用域从分量说明处一直延伸到分程序末尾,然而其可见性却被限制在 R 的说明内,除非在选择分量上引用 AR 作前缀而成为可见。因此对于 I 名不存在冲突。

类似的考虑也防止了命名聚集和子程序调用命名参数中的冲突。

用一个 use 子句能引出若干程序包,而某程序包名可能作为选择分量给出。use 子句可以分号结尾。假设有嵌套程序包

```
package P1 is  
    package P2 is  
        ...  
    end P2;  
    ...  
end P1;
```

那么在 P1 外,我们可写

```
use P1; use P2;
```

或

```
use P1, P1.P2;
```

但不能写:

```
use P1, P2;
```

甚至我们可写:

```
use P1..P2;
```

来获得 P2 而不是 P1 内实体的可见性。

前面我们提到的程序包 STANDARD, 它包含所有预定义实体。此外, 所有库程序单位都被认为说明在 STANDARD 内, 且处在其末尾。这也解释了为什么无须对 STANDARD 使用 use 子句。另外, 只要未重定义来隐藏 STANDARD 名, 则某库程序单位 P 总能以 STANDARD.P 的形式被引用。因此, 在缺省匿名分程序的情况下, 程序中每个循环使用的以及重载的标识符都有其唯一的以 STANDARD 打头的名字。故最好不要对 STANDARD 重定义。

8.5 换名

某些实体可换名。例如, 我们可写

```
declare  
    procedure SPUSH(X, INTEGER) renames STACK.PUSH;  
    function SPOP return INTEGER renames STACK.POP;  
begin  
    ...  
    SPUSH(M);
```

```
...
N:=SPOP;
...
end;
```

这样做可减少二义性,也可避免过长的加点引用表示。例如,假若有两个程序包都含有 PUSH 过程(都带一个 INTEGER 参数),那么写上 use 子句并无多大帮助,因为仍然需要以加点引用方式来区分它们,这时可采用换名。

另外,假设我们希望同时使用函数 INNER 和第7章中的与其等价的运算符“*”,而又不想分别说明两个单独的子程序,则可以根据首先定义的子程序写换名说明:

```
function "*"(X,Y;VECTOR) return REAL renames INNER;
```

或

```
function INNER(X,Y;VECTOR) return REAL renames "*";
```

换名对于库程序单位也很有用。例如,我们希望有两个或两个以上的重载子程序,并且分别单独编译它们。这并不能直接做到,因为库程序单位必须有不同的名字。然而,库程序单位可被换名,结果达到了我们的要求。库程序单位不能是运算符这个限制也能类似地加以克服。同样,换名方法也可用于子单位。

如果子程序被换名,那么新子程序与原子程序中的对应参数(若是函数子程序则还包括其返回结果)的个数、基类型和参数传递型必须相同。这点可用来解决重载问题(如“*”的例子)。这个匹配在编译时生效。另外,参数或结果的任何限定在新子程序中将被忽略,而原子程序中的限定仍然有效。

另一方面,子程序被换名时,新子程序与原子程序中的参数默认情况(即参数的出现或缺省以及默认值)却不必一致。换名可用来引入、改变或删掉参数默认表达式。与新子程序名有关的默认参数在换名说明中给出。换名不能用来给算符以默认值。此外,对应的参数名也不必相同,但新参数名只能用作新子程序调用的命名参数。

枚举直接量可换名为具有合适结果的无参函数。这进一步说明了子程序与枚举直接量的一致性。例如:

```
function TEN return ROMAN_DIGIT renames "X";
```

换名也可用来给对象部分求值。假设我们有一个如6.5节中 PEOPLE 数组这样的记录数组,并希望遍历该数组,以数值形式输出生日。我们可以写

```
for I in PEOPLE'RANGE loop
    PUT(PEOPLE(I).BIRTH.DAY);PUT(":");
    PUT(MONTH_NAME'POS(PEOPLE(I).BIRTH.MONTH)+1);
    PUT(":");
    PUT(PEOPLE(I).BIRTH.YEAR);
end loop;
```

显然,如此每次重复 PEOPLE(I)是很笨拙的,为此我们可以说明一个 DATE 类型的变量 D,并将 PEOPLE(I).BIRTH 复制给它,但在记录很大时,这样做非常浪费。较好的解决办法即

使用换名。如：

```
for I in PEOPLE'RANGE loop
    declare
        D:DATE renames PEOPLE(I).BIRTH;
    begin
        PUT(D.DAY),PUT(":");
        PUT(MONTH_NAME'POS(D.MONTH)+1);
        PUT(",");
        PUT(D.YEAR);
    end;
end loop;
```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

注意：换名不同于文本替代，因为换名并不隐藏旧名。换名后原名仍有其正确的含义。换名说明中任何类型标志蕴含的限定都被忽略，但原对象的限定仍适用。

换名适用于对象（变量与常量）、异常（参见第10章）、子程序和程序包。对于程序包，换名形式很简单，如：

```
package P renames STACK;
```

虽然换名不直接适用于类型，但使用子类型说明也能达到同样的标识效果。如：

```
subtype S is T;
```

应避免任意使用换名，因为过多引入别名会使程序的正确性证明更加困难。

要点 8

- 程序包中的变量存在于程序包的子程序的调用中。
- 库程序单位必须在它的 with 子句所指到的所有其它库程序单位之后被编译。
- 子单位必须在其父程序单位之后被编译。
- 体必须在其相应的规范式说明之后被编译。
- 程序包的规范式说明和体共同组成一个定义域。
- 不要重定义 STANDARD。
- 换名不是文本替代。

第9章 私有类型

超星浏览器提醒您：
使用本复制品
请尊重相关知识版权！

我们已经知道了怎样利用程序包来对用户隐藏内部对象，本章讨论的私有类型使我们能对用户隐藏类型的构造细节。

9.1 一般私有类型

在练习8.1(2)我们写了一个程序包 COMPLEX_NUMBERS，该程序包提供了类型 COMPLEX、常量 I 和在该类型上的一些操作。程序包的规范式说明为：

```
package COMPLEX_NUMBERS is
    type COMPLEX is
        record
            RL, IM: REAL;
        end record;
    I: constant COMPLEX := (0.0, 1.0);
    function "+"(X, Y: COMPLEX) return COMPLEX;
    function "-"(X, Y: COMPLEX) return COMPLEX;
    function "*" (X, Y: COMPLEX) return COMPLEX;
    function "/" (X, Y: COMPLEX) return COMPLEX;
end;
```

在此程序包中，COMPLEX 类型采用直角坐标系加以表示。其中 RL 和 IM 分别表示复数的实部和虚部，常量 I 表示虚单位，函数“+”，“-”，“*”，“/”分别表示复数的四则运算。设 C 为 COMPLEX 类型的变量，若利用该包提供的“+”操作进行复数相加运算，则

C := C + I;

若用預定义“+”操作进行复数相加运算，则

C. IM := C. IM + 1.0;

然而，最好让用户不必知道复数表示的细节，而直接使用程序包提供的复数运算，这样即便将复数的表示改为极坐标系，用户程序亦不必修改也是正确的。为此，可将 COMPLEX 改为私有类型

```
package COMPLEX_NUMBERS is
    type COMPLEX is private;
    I: constant COMPLEX;
    function "+"(X, Y: COMPLEX) return COMPLEX;
    function "-"(X, Y: COMPLEX) return COMPLEX;
    function "*" (X, Y: COMPLEX) return COMPLEX;
```



```
function "/"(X,Y;COMPLEX) return COMPLEX;
function CONS(R,I;REAL) return COMPLEX;
function RL_PART(X;COMPLEX) return REAL;
function IM_PART(X;COMPLEX) return REAL;

private
type COMPLEX is
record
  RL,IM;REAL;
end record;
I;constant COMPLEX:=(0.0,1.0);
end;
```

保留字 `private` 前面的程序包规范式说明部分是可见部分,给出在程序包外可以用的信息。类型 `COMPLEX` 被说明为私有类型,意味着在程序包外不知道该类型的细节,对该类型可施加的操作只有赋值、`=`、`/=` 以及在程序包可见部分中以子程序形式说明的操作。在可见部分中也可说明象 `I` 那样的私有类型常量,常量的初值不能在可见部分给出,因为其类型的细节还不知道,因此在可见部分只说明 `I` 是常量。把这种常量称为延期常量(deferred constant)。

在保留字 `private` 后面我们必须给出私有类型的细节,以及延期常量的初值。

可以用任何与用户可见的操作相一致的方式实现私有类型,如记录、数组、枚举类型等等,甚至还可以用另一个私有类型来实现。在上面我们看到,类型 `COMPLEX` 是用记录来实现的,我们也可等价地用具有两个分量的数组来实现,如:

```
type COMPLEX is array(1..2) of REAL;
```

私有类型的细节被说明后,我们可以用它来说明常量并给出它们的初值。

必须注意的是,除了函数“+”,“-”,“*”,“/”,我们还提供了函数 `CONS`,根据实部和虚部创建一个复数;提供了函数 `RL_PART` 和 `IM_PART`,分别返回复数的实部和虚部。这些函数都是必要的,利用它们用户不必再直接引用类型的结构。当然,函数 `CONS`,`RL_PART` 和 `IM_PART` 仅与我们用直角坐标系表示的复数相对应。

程序包体可参见练习 8.1(2)

```
package body COMPLEX_NUMBERS is
  function "+"(X,Y;COMPLEX) return COMPLEX is
begin
  return(X.RL+Y.RL,X.IM+Y.IM);
end "+";
--'-'/'*'/相似
function CONS(R,I;REAL) return COMPLEX is
begin
  return(R,I);
end CONS;
```



```
function RL_PART(X,COMPLEX) return REAL is
begin
    return X.RL;
end RL_PART;
--IM_PART 相似
end COMPLEX_NUMBERS;
```

程序包 COMPLEX_NUMBERS 可在程序段中被引用,如

```
declare
    use COMPLEX_NUMBERS;
    C,D:COMPLEX;
    R,S:REAL;
begin
    C:=CONS(1.5,-8.0);
    D:=C+I;                                --复数 +
    R:=RL_PART(D)+8.0;                      --实数 +
    ...
end;
```

在程序包外面我们能以常规方式说明类型为 COMPLEX 的变量和常量。须注意创建复数直接量函数 CONS 的使用,由于我们不能在复数和实数之间做混合操作,因而不能简单地写为
 $C := 2.0 * C_1$;

而必须首先借助函数 CONS 将 2.0 创建为复数,上述操作应写为

```
C := CONS(2.0,0.0) * C_1;
```

如果感到这样的写法冗长,可以增加一个重载操作符以允许混合操作。

现在假设我们要以极坐标系表示复数,程序包的可见部分不用改变,而将私有部分变为

```
private
    PI:constant := 3.14159_26536;
    type COMPLEX is
        record
            R:REAL;
            THETA:REAL range 0.0..2.0 * PI;
        end record;
    I:constant COMPLEX := (1.0,0.5 * PI);
end;
```

注意,这里为了方便起见在私有部分说明了常量 PI。其实,除了程序包体以外,任何成份如果适用都可以在私有部分说明,而不仅仅局限于说明类型和常量。在私有部分说明的成份在程序包体中也是有效的。

以极坐标系表示复数后,程序包 COMPLEX_NUMBERS 的体必须全部重写,有些函数变得简单些而另一些函数变得更加复杂。特别是,如能提供一个归一化角度的函数,使 0 处在范围 0 到 2π 之间,是很有益的,该函数的细节留给读者来考虑。由于程序包可见部分没有改变,用户程序也不必修改,这是因为用户没有用到私有类型的细节去编写程序。尽管这样,鉴于 8.2 节中解释的一般相关规则,用户程序需要重新编译。原因在于,编译系统需要私有类型中的信息,以便能给在用户程序中说明的私有类型的对象分配存储空间,如果私有部分改动,则对象的空间也有可能改变。

有趣的是我们可以提供一个无参函数

```
function I return COMPLEX;
```

而不是说明一个延期常量,其好处是可以改变返回值而不要修改程序包规范式说明,因而不必重新编译用户程序。当然,就 I 而言我们是不希望改变其值的。

最后应注意,在私有类型说明和其后的全部类型说明之间,类型处于微妙的半定义状态,因而对它的应用有一些限制,即只能用来说明延期常量、其它类型和子类型以及子程序的规范式说明(也可用作任务入口,参见第 12 章),而不能用来说明变量。例如可以写

```
type COMPLEX_ARRAY is array(INTEGER range<>) of COMPLEX;
```

然后说明

```
C;constant COMPLEX_ARRAY(1..10);
```

但是直到全部说明给出之前,不能说明类型 COMPLEX 或 COMPLEX_ARRAY 的变量。

然而,我们可以说明子程序具有类型 COMPLEX 和 COMPLEX_ARRAY 的参数,甚至能提供默认表达式。这样的默认表达式可引用延期常量和函数,这是因为默认表达式只有当子程序被调用时才求值,而这只有在程序包体被说明后才会发生,并且必然在全部类型说明之后。

还有一些次要的特点与普通用户无关,但在 LRM 中作了描述。记住,一般的规则是你仅能用你所知道的成份。在程序包外面我们只知道类型是私有的,而在程序包中全部类型说明后我们才知道该说明隐含的所有性质。

作为一个例子,我们考虑类型 COMPLEX_ARRAY 和操作符 <<(仅适用于分量是离散类型的数组)。在程序包外面我们不能用 <, 因为不知道类型 COMPLEX 是否为离散的。在程序包内,若它不是离散的,则仍然不能用 <。如果它是离散的,则在全部类型说明之后可以用 <, 但在包外仍不能用 <。另一方面,数组分片适用于所有一维数组,因此在程序包内外都是可用的。

练习 8.1

1. 另写一个函数“*”,以实现实数和复数的混合乘法。
2. 完成程序包 RATIONAL_NUMBERS, 其可见部分为

```
package RATIONAL_NUMBERS is
    type RATIONAL is private;
    function "+"(X,RATIONAL) return RATIONAL; --一元 +
    function "-"(X,RATIONAL) return RATIONAL; --一元 -
    function "+"(X,Y,RATIONAL) return RATIONAL;
    function "-"(X,Y,RATIONAL) return RATIONAL;
    function "*" (X,Y,RATIONAL) return RATIONAL;
```



```
function /*(X,Y;RATIONAL) return RATIONAL;
function /*(X,INTEGER;Y,POSITIVE) return RATIONAL;
function NUMERATOR(R,RATIONAL) return INTEGER;
function DENOMINATOR(R,RATIONAL) return POSITIVE;
private
...
end;
```

有理数是形式为 N/D 的数,这里 N 是整数,D 是正整数,为使预定义等式有效,以消去公因子来简化有理数是必不可少的,这可以用练习 7.1(7) 中的函数 GCD 完成。

3. 为什么

```
function /*(X,INTEGER;Y,POSITIVE) return RATIONAL;
不隐藏预定义的整数除法。
```

9.2 受限私有类型

私有类型上的有效操作可完全被限制在程序包可见部分所说明的范围的,这可借助说明类型是受限私有类型来实现,如:

```
type T is limited private;
```

在这种情况下,赋值和预定义的=、/=在程序包外不再有效。然而,在程序包中可定义函数"=",返回一个 BOOLEAN 类型的值,以判断两个相同受限类型的参数是否相等。操作符/=的含义总可以由=得出,因此,可不必另行显式定义。

对受限私有类型不允许赋值,必然会导致其对象的说明不能包含初值,而这又使得常量在定义的程序包外不能被说明。同样,受限类型的记录分量也不能有默认的初始表达式。然而由于过程的参数机制不是形式地分配的,我们在定义的程序包外面可以用受限类型作为 in 方式或 in_out 方式参数来说明自己的子程序,甚至还能为 in 方式参数提供默认表达式,而 out 方式参数则不允许为受限类型,其理由见下节。

使用受限私有类型的好处是使程序包的作者具有对该类型对象的全部控制权——如资源的拷贝管理权等等。

作为例子,试看下面的程序包:

```
package STACKS is
    type STACK is limited private;
    procedure PUSH(S,in out STACK;X,in INTEGER);
    procedure POP(S,in out STACK;X,out INTEGER);
    function "="(S,T;STACK) return BOOLEAN;
private
    MAX:constant:=100;
    type INTEGER_VECTOR is
        array(INTEGER range<>) of INTEGER;
    type STACK is
        record
            S:INTEGER_VECTOR(1..MAX);
```

```

TOP:INTEGER range 0..MAX:=0;
end record;
end;

```

类型 STACK 的每个对象都是包含数组 S 和整型变量 TOP 的记录,注意 TOP 有一个默认的初值零,这就保证了当我们说明一个 STACK 类型的对象时,该栈被正确地初始化为空。在此,还应注意 INTEGER_VECTOR 的引入,这是由于记录分量不可以是匿名数组。

该程序包的体可以是:

```

package body STACKS is
procedure PUSH(S;in out STACK;X;in INTEGER) is
begin
  S.TOP:=S.TOP+1;
  S.S(S.TOP):=X;
end PUSH;
procedure POP(S;in out STACK;X;out INTEGER) is
begin
  X:=S.S(S.TOP);
  S.TOP:=S.TOP-1;
end POP;
function "="(S,T:STACK) return BOOLEAN is
begin
  if S.TOP/=T.TOP then
    return FALSE;
  end if;
  for I in 1..S.TOP loop
    if S.S(I)/=T.S(I) then
      return FALSE;
    end if;
  end loop;
  return TRUE;
end "=";
end STACKS;

```

从这个例子可以看出,PUSH 的参数 S 为 in out 方式,这是因为既要从栈中读取,又要往栈中写入。还要注意 POP 不能是函数,因为 S 必须是 in out 方式,而函数只能有 in 方式参数。然而,"="可以是函数,因为我们只要读两个栈中的值而不需要修改它们。

函数"="可解释为仅当两个栈的项数相同,且对应项有相同的值时才相等。很明显,用函数"="比较整个记录是完全错误的,因为该记录中没有用到的数组的分量也将被比较。正是由于这一点,类型 STACK 必须是受限私有类型而非一般私有类型,须重新定义"="以给出其正

确含义。

上例是一个典型的数据结构例子,对数组 S 的解释依赖于 TOP 的值。具有这种关系的数据结构通常被说明为受限私有类型。

还有一点要注意,我们将以两种方式用到标识符 S,即作为代表栈的形式参数的名字和在记录内作为数组。这并不冲突,因为尽管其作用域交迭,但可见域不同。当然,这对读者来说可能比较容易搞混,不是好的作法,但它说明了选择记录分量名的自由度。

将程序包 STACKS 用于某程序的片段中,如:

```
declare
    use STACKS;
    ST:STACK;
    EMPTY:STACK;
    ...
begin
    PUSH(ST,N);
    ...
    POP(ST,M);
    ...
    if ST=EMPTY then
    ...
    end if;
    ...
end;
```

这里我们说明了两个栈 ST 和 EMPTY。两个栈初始态都为空,因为它们的内部分量 TOP 具有初值零。假设我们不使用 EMPTY,那么调用函数 = 就可知道栈 ST 是否为空。但以此来测试空栈似乎是靠不住的,因为不能保证 EMPTY 不被使用。如果 EMPTY 是一个常量或许会更好些,但在前面已经提到,不能在程序包外说明受限私有类型的常量,因此常量 EMPTY 的说明只能在程序包可见部分。当然,为测试栈的状态,更好的方法是在程序包中提供函数 EMPTY 和与其对应的函数 FULL。

尽管在所定义的程序包外,不允许对受限私有类型的变量进行赋值,但是我们可以利用受限私有类型能作为子程序参数这一事实,通过编写一过程,使在不弹栈顶元素的前提下,取出栈顶之值,该过程如下:

```
procedure TOP_OF(S:in out STACK;X:out INTEGER) is
begin
    POP(S,X);
    PUSH(S,X);
end;
```

函数“=”的说明并不限制在包含受限类型说明的程序包内，只是在程序外面，我们不知道该类型的结构。此外，包含一个或多个受限类型分量的组合类型也被认为是受限的。因而可以为这种类型定义“=”操作。在程序包 STACKS 的外面可说明

```
type STACK_ARRAY is array(INTEGER range<>) of STACK;
function "="(A,B:STACK_ARRAY) return BOOLEAN;
```

相应的体的定义留给读者完成。

记住，在私有部分（全部类型说明后）和程序包体内，任何私有类型（受限或非受限）都是根据它的表示被处理。因此在 STACKS 体内，类型 STACK 只是一个记录类型，允许赋值操作和基于赋值的操作（如初始化、定义常量等）。可以定义一个过程 ASSIGN（在程序包外使用）如下：

```
procedure ASSIGN(S,in STACK;T,out STACK) is
begin
  T:=S;
end;
```

不幸的是 Ada 并不区分允许赋值和用户定义等式的能力。例如，给栈赋值是合理的，但是，预定义等式却没有意义。同样，对练习 9.1(2) 的 RATIONAL 这样的类型，只要等式被适当地重定义，对没有简化的值进行操作（包括赋值）是完全合理的。因此，我们不得不使用如 ASSIGN 这样的过程形式，或把所有的值简化成规范形式，在此规范形式中分量与分量的等式是满足的。对类型 STACK，适当的形式是所有无用的栈元素都具有象零那样的哑值（dummy value）。

练习 9.2

1. 重新写 STACK 的规范式说明，使其可见部分包含常量 EMPTY。
2. 为 STACKS 编写函数 EMPTY 和 FULL。
3. 给函数“=”编写一个相适应的体，以便适用于类型 STACK_ARRAY，并使其与 8.2 节末尾提到的数据等式的常规规则一致。
4. 为 STACKS 重写 ASSIGN，以便只拷贝记录的有意义部分。
5. 重写 STACKS，使 STACK 为一般私有类型，并保证满足预定义等式。

9.3 资源管理

应用受限私有类型的一个重要例子是提供可控的资源管理。试举现实生活中情况，假如我们为每个资源都设定一把“钥匙”，当分配资源时也将其“钥匙”发给用户，而且任何时候访问资源都必须出示“钥匙”。只要每个资源仅有一把“钥匙”并且不被复制和偷窃，那么系统是十分安全的。如果资源是可重分配的，则通常必须有个机制来提交“钥匙”和重新分配“钥匙”。典型的生活中例子是为邮政信箱配的金属钥匙以及信用卡等等。

现在看看下面的程序：

```
package KEY_MANAGER is
  type KEY is limited private;
  procedure GET_KEY(K,in out KEY);
  procedure RETURN_KEY(K,in out KEY);
```

```

function VALID(K;KEY) return BOOLEAN;
...
procedure ACTION(K;in KEY;...);
...
private
  MAX;constant:=100;    --钥匙号
  subtype KEY_CODE is INTEGER range 0..MAX;
  type KEY is
    record
      CODE;KEY_CODE:=0;
    end record;
end;
package body KEY_MANAGER is
  FREE;array (KEY_CODE range 1..KEY_CODE'LAST) of
  BOOLEAN;=(others=>TRUE);
function VALID(K;KEY) return BOOLEAN is
begin
  return K.CODE/=0;
end;
procedure GET_KEY(K,in out KEY) is
begin
  if K.CODE=0 then
    for I in FREE'RANGE loop
      if FREE(I) then
        FREE(I):=FALSE;
        K.CODE:=I;
        return;
      end if;
    end loop;
    --所有的钥匙被占用
  end if;
end;
procedure RETURN_KEY(K,in out KEY) is
begin
  if K.CODE/=0 then
    FREE(K.CODE):=TRUE;
    K.CODE:=0;
  end if
end;

```



```
...
procedure ACTION(K,in KEY,...) is
begin
  if VALID(K) then
  ...
end;
end KEY_MANAGER;
```



类型 KEY 由具有单个分量 CODE 的记录表示。CODE 有一个默认的零值，表示非法“钥匙”。从 1 到 MAX 的值表示对应资源的分配。当说明一个类型为 KEY 的变量时，它自动取零内码值。为了用这把“钥匙”，必须首先调用过程 GET_KEY，该过程分配第一把空“钥匙”号给这个变量，然后这把“钥匙”就可被诸如 ACTION 之类的各种过程调用，在这里 ACTION 表示访问“钥匙”授权的资源的典型请求。

最后，调用过程 RETURN_KEY 可以归还“钥匙”。因而典型的用户程序段可以是：

```
declare
  use KEY_MANAGER;
  MY_KEY;KEY;
begin
  ...
  GET_KEY(MY_KEY);
  ...
  ACTION(MY_KEY,...);
  ...
  RETURN_KEY(MY_KEY);
  ...
end;
```

类型 KEY 的变量可看作为“钥匙盒”。说明变量时默认值表示盒子为空。类型 KEY 必须是记录，因为只有记录分量才能取默认初值。为了避免“钥匙”的各种可能误用，在此采用了如下方法：

· 如果用已包含有效“钥匙”的变量调用 GET_KEY，则不分配新“钥匙”。不要改变已有效的“钥匙”，否则“钥匙”将丢失，这一点很重要。

· RETURN_KEY 的调用重置变量到默认状态，因此这个变量不能再被用作“钥匙”，除非调用 GET_KEY 得到一把新“钥匙”。注意，用户不能复制“钥匙”，因为类型 KEY 是受限的，赋值无效。

提供函数 VALID 是为了让用户知道“钥匙”变量的值是默认值或是分配值。调用 GET_KEY 后再调用 VALID 是很有用的，这保证了“钥匙”掌管者能提供一个新“钥匙”值。注意，一旦所有的“钥匙”分配完毕，GET_KEY 的调用将不做任何工作。

一个明显的不足之处是在离开包含 MY_KEY 说明的范围之前，没有强制调用 RETURN_KEY，因此“钥匙”有可能丢失，这对应于现实生活中丢失钥匙的情况（尽管在这里没有人能把它重新找回——好象它是被扔进了黑洞一样）。为了防止这一点，“钥匙”掌管者可以假设（就象现实生活中一样）在某段时间内没有使用的“钥匙”就不再有用。当然，同样的“钥匙”不重新发放，但保护的资源被认为是可重用的。为此，需要保存在用“钥匙”和在用资源的分离记录，并建立钥匙到资源的对应关系。

练习 8.3

- 完成下面的程序包，其可见部分为

```

package BANK is
    subtype MONEY is NATURAL;
    type KEY is limited private;
    procedure OPEN_ACCOUNT(K,in out KEY,M,in MONEY);
        --open account with initial deposit M
    procedure CLOSE_ACCOUNT(K,in out KEY,M,out MONEY);
        --close account and return balance
    procedure DEPOSIT(K,in KEY,M,in MONEY);
        --deposit amount M
    procedure WITHDRAW(K,in out KEY,M,in out MONEY);
        --Withdraw amount M;if account does not contain M
        --then return what is there and close account
    function STATEMENT(K,KEY) return MONEY;
        --returns a statement of current balance
    function VALID(K,KEY) return BOOLEAN;
        --checks the key is valid
private
...

```

- 假设你对上题的解答允许银行用储蓄的钱，重新写私有类型以表示一个家庭储蓄箱或安全储蓄箱，钱放在箱子里，由用户保管。

- 如果小偷编写下面的程序

```

declare
    use KEY_MANAGER;
    MY_KEY;KEY;
    procedure CHEAT(COPY,in KEY) is
begin
    RETURN_KEY(MY_KEY);
    ACTION(COPY,...);
    ...
end;
begin
    GET_KEY(MY_KEY);
    CHEAT(MY_KEY);
    ...
end;

```

他试图交还“钥匙”而用复印件，为什么不能得逞？

4. 某破坏者写了下面的程序

```
declare
    use KEY_MANAGER;
    MY_KEY,KEY;
procedure DESTROY(K,out KEY) is
begin
    null;
end;
begin
    GET_KEY(MY_KEY);
    DESTROY(MY_KEY);
    ...
end;
```

他试图摹调用不改变 OUT 参数的子程序来破坏他“钥匙”中的值——他预料复制规则将导致给“钥匙”分配一个假值，为什么他不能得逞？

要点9

- 为使预定义等式合理，值必须以规范形式给出。
- 一个非受限私有类型可根据另一个私有类型来实现，只要该私有类型也是非受限的。
- 一个受限私有类型可根据任何受限或非受限私有类型来实现。
- /=不能定义，它总是随=而定。
- =能在有关受限私有类型的程序包外面定义。
- 上述规则可传递地适用于组合类型。



第10章 异常

在前面的章节中,我们曾多次讲到,如果程序执行出错,那么一个异常——通常是 CONSTRAINT_ERROR 异常将被引发。本章我们便来叙述异常机制,讨论异常引发后如何补救,并介绍怎样去定义和利用自定义异常。与任务有关的异常,将在第14章中讨论。

10.1 异常处理

显然,如果编程时我们违背了语言规则,那么程序运行时就可能引发异常。

Ada 有五个预定义异常,至本章止,我们已遇到了其中的四个,它们是:

CONSTRAINT_ERROR:如果数组下标、记录分量等的取值超出范围,则该异常被引发。

NUMERIC_ERROR:如果算术运算出错,譬如试图用 0 去除一个数,则该异常被引发。参见 4.9 节。

PROGRAM_ERROR:如果违反了程序控制结构,譬如运行到达函数的 end 或调用尚未说明的子程序,则该异常被引发。参见 7.1 节和 8.1 节。

STORAGE_ERROR:如果运行超出存贮空间,譬如以一个很大的参数调用递归函数 FACTORIAL,则该异常被引发。参见 7.1 节。

还有一专为任务而设置的预定义异常:TASKING_ERROR,我们留在第14章中讨论。

如果我们预料到某个异常可能在程序中被引发,那么就可以编写一段处理程序来处理它。譬如程序:

```
begin
    --语句序列
exception
    when CONSTRAINT_ERROR=>
        --相应处理
end;
```

如果在执行 begin 与 exception 之间的语句序列时,CONSTRAINT_ERROR 异常被引发,那么其后的程序执行便被打断,而控制将转至=>后,去做相应处理。以 when 打头的子句即为异常处理段。

我们用根据今天推算明天是星期几这一算法为例,来说明异常的使用。程序如下:

```
begin
    TOMORROW:=DAY'SUCC(TODAY);
exception
    when CONSTRAINT_ERROR=>
```

```
TOMORROW:=DAY'FIRST;
end;
```

如果 TODAY 为 DAY'LAST(即 SUN), 那么当我们试图求 DAY'SUCC(TODAY) 时, 异常 CONSTRAINT_ERROR 就被引发, 从而控制将转至为 CONSTRAINT_ERROR 设置的异常处理段, 并执行语句 TOMORROW:=DAY'FIRST。接下来控制便抵达分程序的结尾。

异常应该用于那些不常发生的情况或那些在发生点不便测试的情况。本例中, 星期天(SUN)并不属于不常发生的情况, 约有 14% 多的日子是星期天, 而且在程序中测试星期天的出现亦不困难。所以这个例子程序应该写成:

```
if TODAY=DAY'LAST then
    TOMORROW:=DAY'FIRST;
else
    TOMORROW:=DAY'SUCC(TODAY);
endif;
```

上例说明了异常机制的简单情况。

在 exception 与 end 之间可以写上若干个异常处理段, 如:

```
begin
    -- 语句序列
exception
    when NUMERIC_ERR when NUMERIC_ERROR|CONSTRAINT_ERROR=>
        PUT("Numeric or Constraint error occurred");
        ...
    when STORAGE_ERROR=>
        PUT("Run out of space");
        ...
    when others=>
        PUT("Something else went wrong");
        ...
end;
```

本例将依照不同的异常情况, 给出相应的信息。这有点类似于 case 语句, 每个 when 都跟着一个或多个以竖杠符分隔的异常名。通常, 我们也可在最后写上 others, 用来处理前面没有列上的任何其它异常。

异常处理段可出现在分程序、子程序体、程序包体(或任务体)的尾部, 并能使用该程序单位中说明的所有实体。上例给出的是一个简化了的分程序, 它没有 declare 说明部分, 我们仅用它来说明在何处设置异常处理段。

异常引发后, 使程序控制不再返回到引发该异常的程序单位中是很重要的。上例中, =>



```
function TOMORROW(TODAY, DAY) return DAY is
begin
    return DAY'SUCC(TODAY);
exception
    when CONSTRAINT_ERROR =>
        return DAY'FIRST;
end TOMORROW;
```

特别要注意的是：不得使用 goto 语句将控制从程序单位转入异常处理段中，或者反过来从某异常处理段中转回它所在的程序单位，也不能从一个异常处理段转至另一异常处理段。然而，异常处理段中的语句可以相当复杂，它可以包含分程序以及子程序调用等等。分程序中的异常处理段，可以包含 goto 语句，使控制转到该分程序外的某标号处，如果该分程序是处在某循环之内的话，那么其异常处理段中也可含有 exit 语句。

如果是程序包体尾部的异常处理段，则它只用于处理该程序包初始化序列中的异常，而不是其中子程序的异常。如果要处理异常情况，这些子程序必须拥有自己的异常处理段。

现在让我们来考虑这样一个重要问题：如果一个程序单位对某异常没有提供异常处理段，那么情况将怎样呢？答案是：这个异常将动态地传播下去。即该程序单位的执行被终止，并且在调用该程序单位的地方引发这个异常。

对于分程序而言，异常将传播到包含该分程序的程序单位中，在那里寻找异常处理段。至于子程序的情况，则是其调用被终止，异常传播到调用该子程序的程序单位中且在那里寻找异常处理段，这个过程将一直重复下去，直至抵达包含这一特殊异常处理段的外层程序单位，或是抵达程序最外层。如果发现某程序单位含有相应的异常处理段，那么异常便在那里进行处理。但当返至主程序而仍未找到相应异常处理段，则整个程序便被放弃，这时运行环境会向我们提供适当的交互信息。（任务中的相应情况将在第14章中讨论。）

异常具有动态传播特性，它不是静态的。一个异常在子程序中没有被处理，它将传播到调用该子程序的程序单位中，而不波及到说明该子程序的程序单位。调用和说明该子程序的程序单位，两者未必同一。

如果异常处理段中的语句本身又引发了异常，那么该程序单位即终止运行，并且将这个异常传播到调用该程序单位的程序单位中。异常处理段不递归使用。

练习 10.1

1. 设用负值参数调用 SQRT 函数和用 0 去除一个数都将引发 NUMERIC_ERROR 异常，请重写 7.3 节的过程 QUADRATIC，避免直接检测 D 和 A。
2. 重写 7.1 节的函数过程 FACTORIAL，使得如果用负值参数去调用它（这将引发 CONSTRAINT_ERROR 异常），或用一个太大的参数去调用它（这将引发 STORAGE_ERROR 或 NUMERIC_ERROR 异常），那么便返回一个特定结果 -1。提示：可说明一内部函数 SLAVE 以实际完成该项工作。

10.2 异常说明与引发

借助预定义异常来检测不常发生但又是预期可能发生的事件，通常是不理想的。因为这时，我们不能肯定实际发生的异常就是由预期的事件引发的，相反，很可能是别的地方出了错。作为实例，让我们考虑8.1节的 STACK 程序包：如果栈已满而继续调用 PUSH 过程，那么当执行到语句 $TOP := TOP + 1$ 时，将引发异常 CONSTRAINT_ERROR；同样，如果栈空了而去调用 POP 过程，那么语句 $TOP := TOP - 1$ 的执行也将引发 CONSTRAINT_ERROR 异常，又因为 PUSH 和 POP 它们本身并没有异常处理段，于是这个异常将传播到调用它们的程序单位中去。相应的调用程序单位可这样编写：

```
declare
    use STACK;
begin
    ...
    PUSH(M);
    ...
    N:=POP;
    ...
exception
    when CONSTRAINT_ERROR=>
        -- 栈非法操作处理
end;
```

对栈的滥用将导致程序控制转到异常 CONSTRAINT_ERROR 的处理段中去。然而，我们并不能保证异常的引发是因栈的滥用而起，也可能是分程序中的其它地方产生了能引发 CONSTRAINT_ERROR 异常的错误。

对此，一个较好的解决办法是规范地说明一个特定异常以指出对栈的滥用。因此，可将 STACK 程序包改写为：

```
package STACK is
    ERROR;exception;
    procedure PUSH(X;INTEGER);
    function POP return INTEGER;
end STACK;
package body STACK is
    MAX;constant:=100;
    S;array (1..MAX) of INTEGER;
    TOP;INTEGER range 0..MAX;
    procedure PUSH(X;INTEGER) is
begin
```

超星浏览器提醒您：
使用本资源品
请尊重相关知识产权！

```
if TOP=MAX then
    raise ERROR;
end if;
TOP:=TOP+1;
S(TOP):=X;
end PUSH;
function POP return INTEGER is
begin
    if TOP=0 then
        raise ERROR;
    end if;
    TOP:=TOP-1;
    return S(TOP+1);
end POP;
begin
    TOP:=0;
end STACK;
```

这样,一个新的异常就犹如变量般地被说明了,并且可用一条标明了该异常名的显式语句来引发,它的处理以及传播规则与预定义异常相同。现在我们再来编写调用程序:

```
declare
    use STACK;
begin
    ...
    PUSH(M);
    ...
    N:=POP;
    ...
exception
    when ERROR=>
        -- 栈非法操作处理
    when others=>
        -- 其它出错处理
end;
```

这样,我们便成功地将滥用栈的异常的处理与其它的异常的处理区分开来了。

注意:没有用 use 语句时,则必须采用加点引用方式,如 STACK.ERROR,在处理段中指明相应的异常。

上例中,ERROR 异常处理段除了报告栈操作出错之外,还应设法使栈恢复到一个可接受

的状态(譬如,在 STACK 包中设立过程 RESET 用来恢复栈);另外,还可考虑是释放还是保留从分程序中获得的资源。例如,如果引用了 9.3 节中的 KEY_MANAGER 程序包,我们可以调用 RETURN_KEY 过程来确保返回预先从分程序中获得的“钥匙”。(RETURN_KEY 过程即使被不恰当地调用亦无危害。)

在其它异常情况下,也可能要重设栈和返回一把“钥匙”,所以,最好定义一个过程 CLEAN_UP 来完成这些要求。现在可将分程序扩充如下:

```
declare
    use STACK,KEY_MANAGER;
    MY_KEY;KEY;
    procedure CLEAN_UP is
        begin
            RESET;
            RETURN_KEY(MY_KEY);
        end;
        begin
            GET_KEY(MY_KEY);
            ...
            PUSH(M);
            ...
            ACTION(MY_KEY,...);
            ...
            N:=P()P;
            ...
            RETURN_KEY(MY_KEY);
    exception
        when ERROR=>
            PUT("Stack used incorrectly");
            CLEAN_UP;
        when others=>
            PUT("Something else went wrong");
            CLEAN_UP;
    end;
```

这里,设 RESET 是 STACK 程序包中说明的补充过程。不过也可在分程序中写上自定义的 RESET 过程。如:

```
procedure RESET is
    JUNK:INTEGER;
```

```
use STACK;  
begin  
    loop  
        JUNK := POP;  
    end loop;  
exception  
    when ERROR =>  
        null;  
end RESET;
```

它是通过反复调用 POP 而直到引发 ERROR 异常来实现的。这时栈已空。这里的异常处理段无须做更多的事情，只是要阻止异常的传播。所以我们仅写空操作 null。这个 RESET 过程象个骗局。所以，最好还是在 STACK 程序包中给出 RESET 过程。

有时，需要对异常造成的结果，在程序上作层层处理。如上例中，我们通过返回一把“钥匙”和重新设置栈而处理了分程序内的异常，但可能造成整个分程序的功能不能得以正确实现。为此，我们可以通过在相应异常处理段末尾再引发异常来表明这种情况：

```
exception  
    when ERROR =>  
        PUT("Stack used incorrectly");  
        CLEAN_UP;  
        raise ANOTHER_ERROR;  
    when others =>  
        ...  
end;
```

这时，异常 ANOTHER_ERROR 将传播到包含该分程序的程序单位中去。当然，我们也可在过程 CLEAN_UP 中写上语句：

```
raise ANOTHER_ERROR;
```

有时，处理某异常并将其传播下去是很容易做到的，这只要写上：

```
raise;
```

即可。这在用一个异常处理段来处理若干异常时特别有用，因为这时我们无法显式地表明是哪种异常被引发了。

所以我们有：

```
when others =>  
    PUT("Something else went wrong");  
    CLEAN_UP;  
    raise;  
end;
```

这时,引发的异常将被记住,即使异常处理段又引发了它自身的异常;如在异常处理段中又调用了那个骗局过程 RESET。不过要注意的是:我们仅需在异常处理段中直接写上 raise;语句,而不是在处理段所调用的过程,如 CLEAN_UP 中,写上 raise;语句。

这个栈的例子展示了异常的正规用法。异常 ERROR 不常发生,而且不宜在其每一可能引发点作条件测试。假如要作条件测试的话,我们可能不得不给 PUSH 过程增加一个 BOOLEAN 类型的 out 型参数以表明其正常与否,然后在每次调用之后再作条件测试。而且也不得不将函数 POP 转换为过程,因为函数不能带上 out 型参数。

这样 STACK 程序包的规范式说明将会变成:

```
package STACK is
    procedure PUSH(X,in INTEGER,B,out BOOLEAN);
    procedure POP(X,out INTEGER,B,out BOOLEAN);
end;
```

而且我们不得不将程序写成:

```
declare
    use STACK;
    OK:BOOLEAN;
begin
    ...
    PUSH(M,OK)
    if not OK then ... end if;
    ...
    POP(N,OK);
    if not OK then ... end if;
end;
```

显然,使用异常能得到一个较好的程序结构。

最后要说明的是:我们也可以显式地引发预定义异常。例如,在 7.1 节我们讨论函数 INNER 时曾提到,或许处理参数约束不匹配的最佳方法就是显式地引发 CONSTRAINT_ERROR 异常。

练习 10.2

1. 重写练习 8.1(1) 的 RANDOM 程序包,当初始化值不为奇数时,它说明并引发异常 BAD。
2. 重写练习 10.1(2) 的答案,如果调用参数为负或太大,函数 FACTORIAL 便引发 NUMERIC_ERROR 异常。

10.3 异常作用域

大体上,异常与其它实体遵循同样的作用域规则。异常能够隐藏另一异常或被另一异常隐藏,异常能通过加点引用而成为可见。异常也能换名:

```
HELP:exception renames BANK.ALARM;
```

然而,异常在有些方面又有些特别。例如,不能定义异常数组,而且异常不能成为记录的分量、



子程序的参数，等等。简而言之，异常不是对象，所以不能对其进行操作，异常仅仅是一种标志。

异常有一个非常重要的特征，即当程序执行时，异常不能被动态地创建，但却可认为它在程序的全部生命期都存在。这是由于异常将动态地传播而得到一个执行链，而不是静态的局限于其作用域。异常可以传播超出其作用域，当然，如果那样的话，它只能匿名地被 others 处理。下面的程序可说明这一点：

```
declare
    procedure P is
        X;exception;
    begin
        raise X;
    end P;
begin
    P;
exception
    when others=>
        --处理 X 异常
end;
```

其中，过程 P 说明了异常 X 并可能引发它，但却不处理它。当我们调用过程 P 时，异常 X 便传播到调用 P 的分程序中，在那里它将作为匿名异常而被处理。

甚至异常也可能传播到它作为匿名异常的作用域之外，并又回到它以正规名字被处理的地方。如：

```
declare
    package P is
        procedure F;
        procedure H;
    end P;
    procedure G is
    begin
        P.H;
    exception
        when others=>
            raise;
    end G;
    package body P is
        X;exception;
        procedure F is
    begin
```

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

```
G;
exception
when X=>
    PUT("Got it!");
end F;
procedure H is
begin
    raise X;
end H;
end P;
begin
P.F;
end;
```

这个分程序说明了包含过程 F 和 H 的程序包 P, 以及另一过程 G。该分程序调用了程序包 P 中的 F 过程, F 又调用 P 外的 G 过程, 而 G 回过头来又调用了 P 中的 H。过程 H 引发了异常 X, 异常 X 的作用域为程序包 P 的体, 可过程 H 并未处理异常 X, 所以该异常将传播到调用 H 的过程 G 中。但 G 又处在程序包 P 之外, 所以现在异常 X 超出了其作用域而成为匿名异常, 因此, 过程 G 将匿名地处理它, 接着通过重新引发而进一步使其传播下去。过程 G 是被程序包 P 中的过程 F 所调用的, 故异常 X 现在传回到了程序包 P 中, 它又能以其正规名字被处理了。

用包含异常说明的递归过程作例, 可对异常的性质作进一步的说明。不同于过程中说明的变量, 每次递归调用并不产生新的异常。每次递归中所指的都是同一异常。例如:

```
procedure F(N;INTEGER) is
X;exception;
begin
if N=0 then
    raise X;
else
    F(N-1);
end if;
exception
when X=>
    PUT("Got it");
raise;
when others=>
    null;
end F;
```

如果执行调用 F(4), 那么将产生递归调用 F(3), F(2), F(1) 直至 F(0), F(0) 以参数 0 去调
• 118 •

用 F 过程时,便引发了异常 X,并作相应处理,即输出一条固定信息,接着再引发它。F 的调用者(即 F(1))接收到了该异常,便再次把它作为 X 异常处理,如此重复。因此这条固定信息总共将输出 5 次,而且最后异常 X 将匿名地传播下去。显然,如每次递归调用都创建了新的异常,那么这条信息将只输出一次。

在前面的所有例子中,异常都是在语句中被引发的。然而,异常也能在说明中被引发。譬如作说明:

```
N:POSITIVE:=0;
```

它将引发 CONSTRAINT_ERROR 异常,因为 N 的初值超出了子类型 POSITIVE 的范围 1..INTEGER'LAST 的约束。说明部中引发的异常,不会被包含该说明部的程序单位中的异常处理段处理,而是即刻传播到该程序单位的上一层中去。这一点告诫我们:即使可以设置有关异常处理段,也要确保程序单位中的所有说明均正确无误,不要冒险去作并不存在的说明。

最后要重视有关 out 方式和 in_out 方式参数的情况:如果子程序被异常终止了,那么任何纯量类型的实参都不会被更新,因为其更新是在正常返回时发生的。对于数组或记录类型,有关的参数机制并不很规范,它们的实参可以有也可以没有初值。(由程序假定一特殊机制当然是错误的)。考虑练习 9.3(1) 程序包 BANK 中的 WITHDRAW 过程:

```
procedure WITHDRAW(K,in_out KEY,M,in_out MONEY) is
begin
    if VALID(K) then
        if M > amount remaining then
            M := amount remaining;
            FREE(K.CODE) := TRUE;
            K.CODE := 0;
            raise ALARM;
        else
            ...
        end if;
    end if;
end;
```

在此,试图取走“钥匙”是不允许的,而且会引起警觉。但是,假如参数机制是以复制来实现的话,那么银行方面就要想到,现在“钥匙”虽然交来了,但贪婪的顾客却有了复印件。

练习 10.3

1. 重写练习 9.3(1) 的程序包 BANK,说明异常 ALARM,使得当任何非法的银行活动试图进行时,便引发该异常。注意避免参数引起的问题。

2. 分析如下病态过程:

```
procedure P is
begin
    P;
```

```
exception
  when STORAGE_ERROR=>
    P;
end P;
```

调用 P 时将发生什么？假设有足够的栈空间支持 N 次 P 过程的递归调用，而第 N+1 次调用将引发异常 STORAGE_ERROR，那么 P 过程总共将被调用多少次且最终将发生什么？

要点 10

- 不要随意使用异常。
- 最好使用自行说明的异常而不是有关的预定义异常。
- 确保处理程序正确返回资源。
- 如果过程被异常终止，out 方式及 in-out 方式参数将不能正确地更改。



第11章 高级类型

本章我们将讨论可判别的记录类型、存取类型和派生类型。由于数值类型是以派生类型说明的，故留待下章论述；任务类型将在第十四章讨论。



11.1 可判别记录类型

迄今为止，我们所遇到的记录类型中，分量之间没有形式语言的相关性，亦即任何相关性纯粹保留在程序员的头脑中。例如，就9.2节中的受限私有类型 STACK 而言，对数组 S 的解释依赖于整数 TOP 的值。

而在可判别记录类型中，设有一些称之为判别式的分量，其它分量则依赖于这些判别式。从语法的角度看，判别式必须为离散类型，可将它们看作为类型的参数化。

作为一个简单的例子，现假设我们编写一个对方阵进行各种操作的程序包，特别是写一个函数 TRACE，求方阵对角元素之和。我们引用6.2节的类型 MATRIX

```
type MATRIX is array(INTEGER range<>, INTEGER range<>) of REAL;
```

函数 TRACE 首先必须判定作为实参传递的 MATRIX 矩阵是否确实为方阵，如：

```
function TRACE (M;MATRIX) return REAL is
    SUM;REAL:=0.0;
begin
    if M'FIRST(1)/=M'FIRST(2) or M'LAST(1)/=M'LAST (2) then
        raise NON_SQUARE;
    end if;
    for I in M'RANGE loop
        SUM:=SUM+M(I,I);
    end loop;
    return SUM;
end TRACE;
```

显然，在 TRACE 函数中再加判定，在编程上不够合理。我们应通过描述来确保矩阵是方阵。这可用可判别类型来实现，譬如：

```
type SQUARE(ORDER;POSITIVE) is
    record
        MAT;MATRIX(1..ORDER,1..ORDER);
    end record;
```

这是具有两个分量的记录类型：第一个分量 ORDER，属于离散子类型 POSITIVE 的判别式；第二个分量 MAT，是一个界限取决于 ORDER 值的矩阵。

类型 SQUARE 的变量可用常规方式说明,但判别式的值必须作为约束给出,例如

M:SQUARE(3);

或者

M:SQUARE(ORDER=>3);

作为约束提供的值可以是任何动态表达式,一旦变量被说明后,其约束就不能被改变。M 的初值可由一个聚集给定,但必须是完整的,且需重复匹配约束,如:

M:SQUARE(3):=(3,(1..3=>(1..3=>0.0)));

但不能写为

M:SQUARE(N):=(M.ORDER,(M.MAT'RANGE(1)=>
(M.MAT'RANGE(2)=>0.0)));

来避免重复 N,因为不能在对象的说明中涉及对象本身.而

```
declare  
  M:SQUARE(N);  
begin  
  M:=(M.ORDER,(M.MAT'RANGE(1)=>  
    (M.MAT'RANGE(2)=>0.0)));
```

却完全有效。如果试图给没有正确判别式值的 M 分配一个值,将引发 CONSTRAINT_ERROR 异常。

常量的说明如常规方式,并且象数组界限一样,判别式的约束可从初值导出。

我们还可以引入子类型

```
subtype  SQUARE_3  is  SQUARE(3);  
  M,SQUARE_3;
```

至此,可重写函数 TRACE 如下:

```
function  TRACE(M:SQUARE)  return  REAL  is  
  SUM,REAL:=0.0;  
begin  
  for  I  in  M.MAT'RANGE  loop  
    SUM:=SUM+M.MAT(I,I);  
  end  loop;  
  
  return  SUM;  
end  TRACE;
```

此函数 TRACE 只能用方阵作为实参来调用。注意,参数的判别式的传递与数组界限的传递是相似的。参数的判别式与数组界限有许多共同之处,例如,甚至在 OUT 型参数的情况下判别式也能被读。而且象数组一样,形式参数也能被约束,如:

```
function  TRACE_3(M:SQUARE_3)  return  REAL;
```

此时实参必须具有判别式值3,否则将引发 CONSTRAINT_ERROR 异常。

函数的结果也可以是可判别类型,如同数组一样,结果是一个值,该值的判别式直到函数被调用时才知道。例如我们可以写一个函数来返回方阵的转置:

```
function TRANSPOSE(M:SQUARE) return SQUARE is
    R:SQUARE(M. ORDER);
begin
    for I in 1..M. ORDER loop
        for J in 1..M. ORDER loop
            R. MAT(I,J):=M. MAT(J,I);
        end loop;
    end loop;
    return R;
end TRANSPOSE;
```

私有类型也可有判别式,它必须根据具有相应判别式的记录类型来实现。对于9.2节的类型 STACK,我们可以建立一个判别式 MAX,用来克服所有堆栈均具有相同最大长度100的问题。我们可写为:

```
package STACKS is
    type STACK(MAX:NATURAL) is limited private;
    procedure PUSH(S:in out STACK;X:in INTEGER);
    procedure POP(S:in out STACK;X:out INTEGER);
    function "="(S,T:STACK) return BOOLEAN;
private
    type INTEGER_VECTOR is array(INTEGER range<>) of
        INTEGER;
    type STACK(MAX:NATURAL) is
        record
            S:INTEGER_VECTOR(1..MAX);
            TOP:INTEGER:=0;
        end record;
end;
```

在此,类型 STACK 的每个变量包含有一个给出最大栈尺寸的判别式分量,当说明一个栈时必须给出这个值。例如:

```
ST:STACK(100);
```

如同类型 SQUARE,这里的判别式的值不能随后改变。当然,尽管其余成分是私有的,但这个判别式是可见的,并可按 ST. MAT 的形式引用。

程序包 STACKS 的体仍与以前一样(见9.2节)。尤其要注意函数“=” ,它可用来比较具有

不同 MAX 值的栈,因为它仅仅比较正在使用的内部数组分量。

尽管类型 STACK 的常量不能在定义的程序包外说明(因为类型是受限私有的),但我们在可见部分说明一个延期常量。这样的说明不需要为判别式提供一个值,因为它将在私有部分给出。

判别式在某些方面与子程序参数具有相似性。判别式的类型或子类型必须以类型标记而不是子类型表示给出。这样对带有判别式的私有类型(就象前面类型 STACK 说明的那样),当判别式说明必须重复时,可应用同样简单的一致性规则。

带有判别式的延期常量的情形也存在类似的一致性问题。假设我们要说明一个具有判别式常量 3 的 STACK,可以在可见部分省略判别式而仅仅写

```
C;constant STACK;
```

然后在私有部分中,作为约束或者通过强制性初值(或两者同时),给出其判别式

```
C;constant STACK(3):=(3,(1,2,3),3);
```

然而,如果我们要在可见部分给出判别式,那么就必须引进一个子类型(在子类型表示中,不能用显式的约束)。引入子类型后完全常量说明也必须用它,因为类型标记必须一致。例我们写

```
subtype STACK_3 is STACK(3);
C;constant STACK_3
```

然后在私有部分写

```
C;constant STACK_3:=(3,(1,2,3),3);
```

迄今我们所遇到的可判别类型,一旦变量被说明,其判别式就不能改变。然而,可以给判别式提供一个默认表达式,这样情况就不同了,可以带上或不带判别式约束来说明变量。如果提供判别式约束,则该值就取代默认值,并且与前面一样判别式不能改变;另一方面,如果变量说明时没有带判别式值,那么就取默认表达式的值,但它可被整个记录赋值而改变。

假设要计算如下形式的多项式:

$$p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$$

其中,如果 $n \neq 0$,则 $a_n \neq 0$ 。这样的多项式可表述为:

```
type POLY(N,INDEX) is
record
  A:INTEGER_VECTOR(0..N);
end record;
```

这里

```
subtype INDEX is INTEGER range 0..MAX;
```

类型 POLY 的变量在说明时必须带有约束,成为某固定尺寸的多项式。这很不方便,因为多项式的尺寸有可能是复杂计算的结果。例如,如果将两个 $n=3$ 的多项式作减法,那么只有当 x^n 的系数不同时其结果多项式才满足 $n=3$ 。然而,如果说明

```
type POLYNOMIAL(N,INDEX:=0) is
record
  A:INTEGER_VECTOR(0..N);
end record;
```

然后说明变量

P,Q:POLYNOMIAL;

这些变量没有约束,其判别式的初值是 N 的默认值零,这个值可由赋值改变。然而要注意判别式只能因整个记录赋值而被改变,因此

P.N:=8;

是非法的,这很自然,因为数组 P.A 不能任意改变其界限。

类型 POLYNOMIAL 的变量也可带约束说明

R:POLYNOMIAL(5);

这样,R 永远被约束成 n=5 的多项式。

在说明中可以用常规方式给出初值:

P:POLYNOMIAL:=(1,(5,0,4,2));

这表示多项式 $5 + 4x^2 + 2x^4$ 。注意除了初值,P 没有被约束。

实际上可以使类型 POLYNOMIAL 成为私有类型,以使 $a_0 \neq 0$ 。由于预定义等式是满足的,不必说明为受限私有类型。私有类型说明和完全类型说明都要给出 N 的默认表达式。

再次注意与子程序参数相似性:默认表达式只有当需要时被计算,因此不必每次产生相同的值。此外,对私有类型来说,当它不得不被再次写出时,可施加相同的一致性规则。

假如我们说明函数

function "-"(P,Q:POLYNOMIAL) return POLYNOMIAL;

那么,必须保证结果是规范的以使 a_0 不为零。这可由下面的函数实现:

```
function NORMAL(P:POLYNOMIAL) return POLYNOMIAL is
  SIZE:INTEGER := P.N;
begin
  while SIZE>0 and P.A(SIZE)=0 loop
    SIZE:=SIZE-1;
  end loop;
  return(SIZE,P.A(0..SIZE));
end NORMAL;
```

该函数返回值的判别式直到调用时才知道。请注意此处数组片的应用。

可以说明具有多个判别式的类型。作为例子,看一下没被约束为方阵但其两个下界都为1 的矩阵,这可用如下方法来实现:

```
type RECTANGLE(ROWS,COLUMNS,POSITIVE) is
  record
    MAT:MATRIX(1..ROWS,1..COLUMNS);
  end record;
```

然后说明

R:RECTANGLE(2,3);

或

R:RECTANGLE(ROWS=>2,COLUMNS=>3);

这里,常规规则也是适用的:位置值必须按顺序给出;命名值可按任意顺序给出;可以使用混合表示法,但首先必须是位置值。

如果提供默认表达式,那么对该类型的所有判别式都必须提供默认表达式。这个规则类似于多维数组类型的规则——所有的界限必须被约束。另外,如果提供默认表达式,那么对象必须完全约束或根本就不约束。类似地,子类型必须提供所有约束或一个也不提供。我们不能说明

subtype ROW_3 is RECTANGLE(ROWS=>3);

而必须说明为

```
type ROW_3(COLUMNS:POSITIVE) is
  record
    MAT:MATRIX(1..3,1..COLUMNS);
  end record;
```

属性 CONSTRAINED 适用于可判别类型的对象,它给出一个 Boolean 值以指示对象是否受约束。对如 SQUARE 和 STACK 这样没有判别式默认值的类型的任何对象,这个属性的值当然是 TRUE,但对象 POLYNOMIAL 这样有默认值的类型的对象,属性值可以是 TRUE,也可以是 FALSE。

P'CONSTRAINED=FALSE

R'CONSTRAINED=TRUE

前面我们提到,无约束的形参将接受实参的判别式值。就 out 型或 in_out 型参数而言,如果实参受约束则形参也受约束(in 型参数总为常量)。假设我们说明一个删去最高阶项以截断多项式的过程

```
procedure TRUNCATE(P,in_out POLYNOMIAL) is
begin
  P:=(P.N-1,P.A(0..P.N-1));
end;
```

然后给出

```
Q:POLYNOMIAL;
R:POLYNOMIAL(5);
```

则语句

TRUNCATE(Q)

是正确的,但

TRUNCATE(R);

将导致引发 CONSTRAINT_ERROR 异常。

在例子中我们已看到,判别式是如何作为数组的上界被用在下标约束中的(也可被用作数组下界),下一节我们将讲述怎样用判别式来引导变体部分。在这些情况下判别式必须被直接应用而不是作为表达式的一部分,因此不能说明

```
type SYMMETRIC_ARRAY(N:POSITIVE) is
  record
```

```
A:VECTOR(-N..N);  
end record;
```

或

```
type TWO_BY_ONE(N,POSITIVE) is  
record  
    A: MATRIX(1..N,1..2*N);  
end record;
```



练习 11.1

1. 假设 M 是类型 MATRIX 的一个对象, 写一个对函数 TRACE 的调用以决定 M 的轨迹, TRACE 的参数是类型 SQUARE 的聚集。如果 M 的两个维长不等时将发生什么情况?
2. 重写 STACKS 的规范式说明, 使得可见部分包含常量 EMPTY, 参见练习 9.2(1)。
3. 为 STACKS 写函数 FULL, 参见练习 9.2(2)。
4. 说明一个表示零的 POLYNOMIAL 多项式(即 0x⁰)
5. 编写两个多项式相乘的函数“*”。
6. 编写两个多项式相减的函数“-”, 可借助函数 NORMAL 实现。
7. 重写过程 TRUNCATE, 使得如果试图截断一个受约束的多项式则引发 TRUNCATE_ERROR 异常。

11.2 变体记录

如果记录中部分结构对该类型的所有对象是固定的, 而其余结构对不同对象可取几种不同形式之一, 这样的记录有时很有用。可用变体部分来实现这种记录, 对可选结构的选择由判别式的值控制。

考虑下面的说明:

```
type GENDER is (MALE,FEMALE);  
type PERSON(SEX,GENDER) is  
record  
    BIRTH:DATE;  
    case SEX is  
        when MALE=>  
            BEARDED:BOOLEAN;  
        when FEMALE=>  
            CHILDREN:INTEGER;  
    end case;  
end record;
```

这里说明了一个带有判别式 SEX 的记录类型 PERSON。类型为 DATE(见 6.5 节)的分量 BIRTH 对该类型的所有对象是相同的, 然而, 其余分量取决于 SEX 并且作为一个变体部分被说明。如果 SEX 的值是 MALE, 则有进一层的分量 BEARDED; 如果 SEX 是 FEMALE 则有分量 CHILDREN。这是因为只有男性才有胡须, 只有女性才能生育孩子。

由于没有为判别式提供默认表达式, 所以该类型的所有对象都必须被约束。因此可以说明

JOHN:PERSON(MALE),
BARBARA:PERSON(FEMALE);

或引入子类型,写成:

```
subtype MAN is PERSON(SEX=>MALE);  
subtype WOMAN is PERSON(SEX=>FEMALE);  
JOHN:MAN;  
BARBARA:WOMAN;
```

聚集取常规形式,但只给出变体中对应的可选结构的分量。控制变体的判别式的值必须是静态的,这样编译程序才能检查聚集的一致性。因此我们可以写:

```
JOHN:=(MALE,(19,AUG,1937),FALSE);  
BARBARA:=(FEMALE,(13,MAY,1943),2);
```

但不能写成

```
S;GENDER:=FEMALE;  
BARBARA:=(S,(13,MAY,1943),2);
```

因为 S 不是静态的而是变量。

变体的分量能以常规方式访问和改变,例如:

```
JOHN.BEARDED:=TRUE;  
BARBARA.CHILDREN:=BARBARA.CHILDREN+1;
```

但如试图访问象 JOHN.CHILDREN 这样错误的可选分量,将引发 CONSTRAINT_ERROR 异常。注意尽管类型 PERSON 对象的性别不能改变,但编译时无须知道。我们可以有:

```
S;GENDER:=...  
CHRIS,PERSON(S);
```

这里 CHRIS 的性别直到他(或她)被说明时才确定。“判别式必须是静态的”规则仅适用于聚集。

类型 PERSON 的变量必须被约束,因为该类型没有判别式的默认表达式,因此指派改变性别的值是不可能的,试图这样做将引发 CONSTRAINT_ERROR 异常。然而,象对类型 POLYNOMIAL 一样,我们可以说明判别式的默认初始表达式,从而说明无约束变量。之后,这种无约束变量可以靠整个记录的赋值来指派具有不同判别式的值。因此可以有:

```
type GENDER is (MALE,FEMALE,NEUTER);  
type MUTANT (SEX:GENDER:=NEUTER) is  
record  
  BIRTH:DATE;  
  case SEX is  
    when MALE=>  
      BEARDED:BOOLEAN;  
    when FEMALE=>  
      CHILDREN:INTEGER;  
    when NEUTER=>
```

```
    null;  
end case;  
end record;
```

注意，在 NEUTER 的情况下，不需要任何分量，必须以 null 作为可选分量。在 case 语句中用 null 语句还表示确实没有分量，而不是偶然遗漏掉的。

现在我们说明

M, MUTANT;

这个无约束突变体的性别是默认的中性，但可被整个记录的赋值改变。

注意以下两式的区别：

M, MUTANT := (NEUTER, (1, JAN, 1984));

和

N, MUTANT(NEUTER) := (NEUTER, (1, JAN, 1984));

第一种情况突变体没有受约束，但正好是初始的中性。这个例子也说明了在可选结构中没有分量时聚集的形式：没有分量就什么都不写，而不用写 null。

有关可选结构的规则与 5.2 节讨论的关于 case 语句的规则非常接近。每个 when 跟着一个或多个由竖杠分隔的选择项，每个选择或者是一个简单表达式或是一个离散范围。可用选择 others，但它必须处在最后并且只有它本身。所有的值和范围必须是静态的，并且判别式的所有可能的值必须用到一次且只用一次。判别式的可能值是它的静态子类型（如果有的话）或类型的值。每个可选结构可以包含若干分量说明，也可以是我们已知的 null。

一个记录只能包含一个变体部分，并且必须跟在其它分量后面。然而，变体可嵌套，变体部分中的分量列表本身可包含一个变体部分，但它也必须在其它分量之后。

注意：在变体的不同可选分量中不能使用同样的分量标识符，一个记录的所有分量必须具有相异的标识符。

有必要强调一下关于判别式改变的规则。如果一个对象具有判别式约束说明，那么它不能被改变（这个约束类似范围约束，判别式必须总是满足它）。因为约束只允许单值，这自然就意味着判别式只能取该值，而不能改变。

另一方面，所有对象必须具有判别式分量值。因此，如果类型没有提供默认初始表达式，那么在对象说明中就一定要提供，并且因为这是作为约束被表达，从而该对象便被约束。

重新命名可判别类型对象的分量是有限制的，如果分量的存在依赖于判别式的值，那么对象无约束时，该分量就不能被重新命名。因此我们不能写

C: INTEGER renames M. CHILDREN

这是因为不能保证分量 M. CHILDREN 在重新命名后继续存在，甚至有可能在重新命名时就已不存在了。然而

C: INTEGER renames BARBARA. CHILDREN

是有效的，因为 BARBARA 是具体的人，不能改变性别。

我们已经知道判别式可用作数组的界限，能作为控制变体的表达式，同样判别式还可作为内部分量的判别式约束，例如可以说一个包含两相同尺寸的多项式类型：

```

type TWO_POLYNOMIALS(N,INDEX:=0) is
record
  X:POLYNOMIAL(N);
  Y:POLYNOMIAL(N);
end record;

```

最后,判别式也能被用作记录中某分量的默认初值表达式的一部分。尽管判别式的值可能要到对象说明时才知道,但这并没有关系,因为默认初值表达式只有在对象说明时才被求值,并且无须提供其它初值。

除了上述用途外,判别式不能用作其它用途。很不幸,这意味着当我们说明前节中的类型 STACK 时,不能继续对 TOP 按如下方法施加约束:

```

type STACK(MAX,NATURAL) is
record
  S:INTEGER_VECTOR(1..MAX);
  TOP:INTEGER range 0..MAX:=0;
end record;

```

因为在范围约束中不允许用到判别式 MAX。

最后需指出,可以根本就不必用到判别式,而仅仅把它当作记录的一个分量。当我们希望有某些分量是私有的而其它分量是可见的类型时,这是非常合适的。作为一个有趣且最后的例子,重新考虑 9.3 节的类型 KEY,可将它改为

```
type KEY(CODE:INTEGER:=0) is limited private;
```

以及

```

type KEY(CODE:INTEGER:=0) is
record
  null;
end record;

```

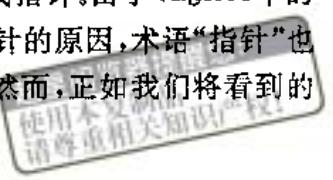
利用这种表示,用户可以读他的钥匙号码。但不能改变它。

练习 11.2

1. 编写过程 SHAVE,该过程接受类型 PERSON 的一个对象,如果对象是男士则剃掉胡子,如果是女士则引发 SHAVING_ERROR 异常。
2. 编写过程 STERILISE,接受类型 MUTANT 的一个对象,如需要且可能的话改变性别以保证为 NEUTER,否则引发适当的异常。
3. 说明一个描述圆或矩形等几何对象的类型 OBJECT,圆由半径表示,正方形由边表示,矩形由长和宽表示。
4. 编写返回 OBJECT 面积的函数 AREA。
5. 重写 11.1 节的类型 POLYNOMIAL 说明,使得 n 阶多项式的默认初值表示为 x^n 。提示:说明一个返回恰当阵列值的辅助函数。

11.3 存取类型

就迄今已遇到的类型而言,对象的名都受制于对象本身,并且对象的生存期从它的说明开



始,直到控制离开包含这个说明的程序单位为止。这对许多应用来说限制太大,希望有一个更加灵活的对象分配控制,Ada中的存取类型可以满足这个要求。存取类型的对象,正象其名称所隐喻的那样,提供对其它对象的存取,而被存取的对象能以独立于分程序结构的方式来分配。

对于熟悉其它语言的读者来说,存取类型可看作引用(reference)或指针。由于Algol68中的悬挂引用的原因,术语“引用”已是声名狼藉;而由于PL/I中的匿名指针的原因,术语“指针”也是名声不佳,因而新术语“存取”可被认为是引用或指针的友好名称。然而,正如我们将看到的那样,Ada的存取对象是强制类型的,不存在悬挂引用的问题。

存取类型的一个最简单的应用是表处理。例如:

```
type CELL;
type LINK is access CELL;
type CELL is
  record
    VALUE:INTEGER;
    NEXT:LINK;
  end record;
L:LINK;
```

这些说明引入了存取 CELL 的类型 LINK,变量 L 可看作仅能指向类型 CELL 对象的引用变量。类型 CELL 的对象是具有两个分量的记录,一个分量是类型为 INTEGER 的 VALUE,另一个分量是类型 LINK 的对象 NEXT,因此 NEXT 也可以存取类型 CELL。这样,这些记录就可形成一个链接表。初始时没有记录对象,只有单个指针 L,L 的默认值取不指向任何对象的 null。我们可以显式地给出 L 的默认值,如:

```
L:LINK:=null;
```

注意 LINK 和 CELL 定义中的循环。由于这个循环和线性确立(lineare laboration)规则,首先必须给出一个非完整的 CELL 说明,然后说明 LINK 并完成 CELL 的完整说明。在非完整和完整说明之间,类型名只能被用在存取类型的定义中,此外,除了下节我们将要提到的一种情况外,非完整和完整的说明必须处在同一说明表中。

存取对象由能够提供初值(但不是必要)的分配程序的执行而建立。在程序中,以分配符标记分配程序。分配符由保留词 NEW 跟着对象的类型组成,且在类型后面还可提供对象的初值表达式。分配符的执行结果是一个可以赋给存取类型变量的存取值。因此

```
L:=new CELL;
```

将创建一个类型 CELL 的记录并分配一个对象指针(引用到或指向对象)给 L。其结果可表示成图11.1:此记录的 NEXT 分量取默认值 null,而 VALUE 分量没有定义。

L 所指对象的分量可用通常的加点引用方式来存取,如给 VALUE 分量赋值可写为

```
L.VALUE:=37;
```

另一方面,也可以用分配符提供初值

```
L:=new CELL'(37,null);
```

这里,初值采取聚集形式,并且不论某些分量是否有默认的初始表达式,通常必须给所有分量提供值。

当然,在 L 被说明时,也能用分配符来初始化 L。

L:LINK:=new CELL'(37,null);

仔细区分类型 LINK 和 CELL,L 属于存取 CELL 的 LINK 类型,而 CELL 为跟随着 new 的被存取类型。

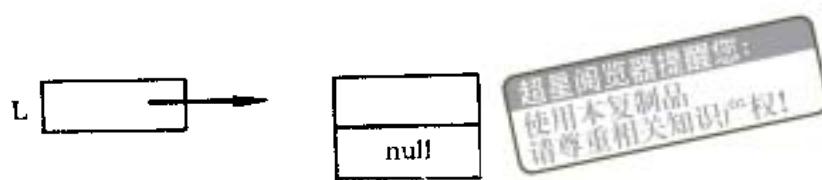


图11.1 存取对象

现在,假设要创建另一个记录并将它链接到现存的记录上,可以说明另一个变量。

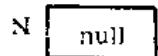
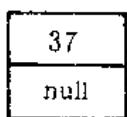
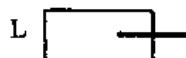
N:LINK;

然后执行

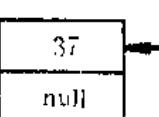
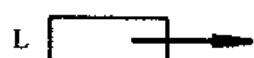
N:=new CELL'(10,L);

L:=N;

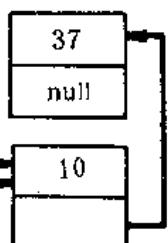
以上三步的结果可用图11.2表示:



(a)



(b)



(c)

图11.2 扩展表

注意,赋值语句

L:=N;



是拷贝存取值(即指针),而不是拷贝存取对象。如要拷贝对象,可逐个分量进行。

L.VALUE := N.VALUE;

L.NEXT := N.NEXT;

或用 all;

L.all := N.all;

L.all 代表由 L 存取的整个对象。实际上,可以认为 L.VALUE 是 L.all.VALUE 的缩写。

类似地如果 L 和 N 指向同一对象,则

L=N

为真,而如果 L 和 N 指向的对象正好具有相同的值,则

L.all = N.all

为真。

可以说明存取类型的常量。当然,作为常量必须提供初值。

C:constant LINK:=new CELL'(0,null);

C 是常量,这意味着它总是指向同一对象。然而,该对象的值是可以改变的,因此

C.all := L.all;

是允许的,但不允许出现

C:=L;

实际上,为了扩展表,并不需要引入变量 N,可以简单地写为

L:=new CELL'(V,L);

此语句可用来构造创建新记录并加到表头的通用过程

```
procedure ADD_TO_LIST(LIST;in out LINK,V;in INTEGER) is
begin
LIST:=new CELL'(V,LIST);
end;
```

调用过程

ADD_TO_LIST(L,10);

可将包含值10的新记录加到由 L 访问的表中。

存取类型的参数传递机制类似于纯量类型的参数传递机制。然而,为了防止存取值变得没有定义,开始时 out 型参数总要被赋实参的值。另外,没有初始化的存取对象将取特定的默认值 null。这两点用来防止可能引起程序混乱的无定义指针的出现。

值 null 对判断表在什么时候为空是有用的,下面的函数返回表中记录的 VALUE 分量之和

```
function SUM(LIST;LINK) return INTEGER is
```

L,LINK:=LIST;

S:INTEGER:=0;

begin

while L/=null loop

S:=S+L.VALUE;

```

L:=L.NEXT;
end loop;
return S;
end SUM;

```

此处,必须做一个 LIST 的副本,因为 in 型的形参是常量。变量 L 用来沿表往下工作直到表尾。这个函数即使在表为空时也能工作。

一个更为复杂的数据结构是二叉树,它由结点组成,每个结点有一个值和二棵子树,子树可以为空。适当的说明为:

```

type NODE;
type TREE is access NODE;
type NODE is
record
    VALUE:REAL;
    LEFT,RIGHT:TREE;
end record;

```

作为树应用的一个有趣的例子,考虑下面的过程 SORT,该过程将数组中的数值按递增序排列。

```

procedure SORT(A,in out VECTOR) is
I:INTEGER;
BASE:TREE:=null;

procedure INSERT(T,in out TREE,V,REAL) is
begin
    if T=null then
        T:=new NODE'(V,null,null);
    else
        if V<T.VALUE then
            INSERT(T.LEFT,V);
        else
            INSERT(T.RIGHT,V);
        end if;
    end if;
end INSERT;
procedure OUTPUT(T:TREE) is
begin
    if T/=null then

```



```
    OUTPUT(T.LEFT);
    A(I) := T.VALUE;
    I := I + 1;
    OUTPUT(T.RIGHT);
end if;
end OUTPUT;
begin -- SORT 的体
for J in A'RANGE loop
    INSERT(BASE, A(J));
end loop;
I := A'FIRST;
OUTPUT(BASE);
end SORT;
```

递归过程 INSERT 把包含值 V 的新结点加入树 T 中, 其方法是使得结点左子树上的值总是小于结点的值, 而右子树上的值总是大于(或等于)结点值。

递归过程 OUTPUT 将树中所有结点的值拷贝到数组中。首先输出左子树(具有最小值), 然后拷贝结点值, 最后输出右子树。

过程 SORT 依次以数组的每个元素调用 INSERT, 建立一棵树, 然后调用 OUTPUT 将有序值拷回数组中。

到目前为止, 我们所遇到的存取类型都是指向记录的, 这是最一般的情况。其实, 存取类型可以指向任何类型, 甚至是其它存取类型。因此我们有:

```
type REF_INT is access INTEGER;
R:REF_INT:=new INTEGER'(46);
```

注意, R 指向的整型值标记为 R.all(也许不是很合适), 因此我们能写

```
R.all:=13;
```

将值从 46 改到 13。

由存取类型指向的所有对象必须通过分配符获得, 这一点是很要的。我们不能写成

```
I:INTEGER;
```

```
R:REF_INT:=I;           -- 非法
```

被存取对象形成一个集合, 其作用域就是存取类型的作用域。仅当最终离开此作用域(当然, 那时存取变量已不存在), 该集合才不存在, 因此不会引起悬挂引用问题。

如果一个对象由于没有指针直接或间接指向它而变得不可存取时, 它占用的空间可以回收, 以便被别的对象再次使用。一个执行系统可以(但不是必须)提供一个无用单元收集器以完成此项工作。

另一方面, 具有一个机制, 借助于它程序能够指出某对象不再需要, 如果仍然对这个对象引用, 则程序将出错。更为详细的说明请参照 5.6 节。本章我们假定无用单元收拾器已准备好, 随时可用。

最后有几点要详细说明一下。分配符说明了重复求值的重要性以及何时在特定上下文中

表达式被求值。例如，在聚集中表达式为每个相关下标求值，因此

```
A:array(1..10) of LINK:=(1..10=>new CELL);
```

创建一个具有十个元素的数组，并将每个元素初始化为存取不同的新单元。又如

```
A,B:LINK:=new CELL;
```

创建两个新单元，

而 A:LINK:=new CELL;

```
B:LINK:=A;
```

只创建一个单元。还要记住，记录分量、判别式和子程序参数的默认表达式每当需要时可重新求值，如果这样的表达式包含分配符，则每次将创建一个新对象。

如果分配符提供初值，则可取任何受限表达式形式，因此可以有

```
L:LINK:=new CELL/(N.all);
```

在这种情况下，新对象的值与 N 指向的对象的值相同。还可以有

```
I:INTEGER:=48;
```

```
R:REF_INT:=new INTEGER'(I)
```

在此情况下，新对象取 I 的值，I 不是存取对象也没关系，因为我们只关心它的值。

被存取的类型可以受约束，例如

```
type REF_POS is access POSITIVE;
```

或者等价地

```
type REF_POS is access INTEGER range 1..INTEGER'LAST;
```

这样，该类型所指对象之值都被约束成正数。可以写

```
RN:REF_POS:=new POSITIVE'(10);
```

或者写成

```
RN:REF_POS:=new INTEGER'(10);
```

注意，如果写 new POSITIVE'(0)，则由于 0 不属于子类型 POSITIVE，将引发 CONSTRAINT_ERROR 异常。然而，如果写 new INTEGER'(0)，则由于分配符的上下文而同样引发 CONSTRAINT_ERROR 异常。

每个存取类型的说明都引入一个新的集合，这一点是很重要的。两个集合可以由相同类型的对象组成，但存取类型的值不能指向另一集合中的对象，因此

```
type REF_INT_A is access INTEGER;
type REF_INT_B is access INTEGER;
RA:REF_INT_A:=new INTEGER'(10);
RB:REF_INT_B:=new INTEGER'(20);
```

由这两个分配符创建的对象具有同样的类型，但存取值却属于由分配符上下文决定的不同类型，并且对象在不同的集合中。故尽管可以写

```
RA.all:=RB.all;
```

但是

```
RA:=RB;
```

却是非法的。

练习 11.3

1. 编写过程

```
procedure APPEND(FIRST,in out LINK,SECOND,in LINK);
```

将表 SECOND 扩展到表 FIRST 的表尾(不做拷贝),注意各种特殊情况。

2. 编写函数 SIZE 返回树的结点数。

3. 编写拷贝树的函数 copy。



11.4 存取类型和私有类型

私有类型可作为存取类型实现。重新考虑类型 STACK, 假设除了计算机可用空间影响外, 不对栈施加最大栈尺寸的限制, 这可将栈表示成表来实现。

```
package STACKS is
    type STACK is limited private;
    procedure PUSH(S,in out STACK,X,in INTEGER);
    procedure POP(S,in out STACK,X,out INTEGER);
private
    type CELL;
    type STACK is access CELL;
    type CELL is
        record
            VALUE,INTEGER;
            NEXT:STACK;
        end record;
    end;
    package body STACKS is
        procedure PUSH(S,in out STACK,X,in INTEGER) is
        begin
            S:=new CELL'(X,S);
        end;
        procedure POP(S,in out STACK,X,out INTEGER) is
        begin
            X:=S.VALUE;
            S:=S.NEXT;
        end;
    end STACKS;
```

当用户说明栈

```
S;STACK;
```

时, S 自动取默认值 null(即表示栈为空)。如果栈为空时调用 POP, 则将对
null. VALUE

求值, 会引发 CONSTRAINT_ERROR 异常。PUSH 失败的唯一可能是存贮空间用完, 此时执行
new CELL'(X,S)

将引发 STORAGE_ERROR 异常。

在这种栈形式中, 我们建立了一个受限私有类型。预定义等式仅仅用来测试两个栈是否为同一个栈, 而不是测试它们是否具有相同的值, 并且赋值只拷贝指向栈的指针而不是栈本身。编写函数“=”必须小心, 我们试写为:

```
function ==(S,T:STACK) return BOOLEAN is
  SS,STACK:=S;
  TT,STACK:=T;
begin
  while SS/=null and TT/=null loop
    SS:=SS.NEXT;
    TT:=TT.NEXT;
    if SS.VALUE/=TT.VALUE then
      return FALSE;
    end if;
  end loop;
  return SS=TT; --如果均为 null, 则 TRUE.
end;
```

但这将不能工作, 因为新定义隐藏了在“=”体内用到的预定义等式(以及不等式), 因此这个函数将无限地递归。解决的办法是以某种方式区分类型 STACK 和它的表示, 例如使类型 STACK 为只有一个分量的记录。

```
type CELL;
type LINK is access CELL;
type CELL is
  record
    VALUE:INTEGER;
    NEXT:LINK;
  end record;
type STACK is
  record
    LIST:LINK;
  end record;
```

这样, 就能区分 S(STACK 类型)和 S.LIST(其内部表示)了。

上一节我们已提到,如果由于循环的原因不得不首先写出不完整的说明(如在类型 CELL 中),那么完整的说明必须出现在同一说明表中。对这个一般规则有一个例外,也就是只要涉及这个规则,私有部分和相应的包体就被认为是单个说明表。因此,在上面的任何一种表示中,类型 CELL 的完整说明可被从私有部分中移至程序包 STACKS 的体中。这样做有一个好处,就是遵循独立规则(即由于类型 CELL 的详细说明改变而用户程序不用重新编译)。从实现的观点来看这是可能的,因为已假设所有访问类型的值占据的空间相等——典型的是一个字。

另一方面,存取类型反过来也能指向私有类型,如:

```
type REF_STACK is access STACK;
```

有意思的是如果存取类型为受限私有类型,那么分配符不能提供初值,因为这等价于赋值,而赋值对受限类型是不允许的。

练习 11.4

1. 假设在 STACK 规范式说明中说明了异常 ERROR, 重写过程 PUSH 和 POP 以便它们引发 ERROR 异常而不是 STORAGE_ERROR 和 CONSTRAINT_ERROR 异常。

2. 用下面的表示重写 PUSH, POP 和“=”

```
type STACK is
  record
    LIST,LINK;
  end record;

不必考虑可能产生异常的情况。
3. 完成可见部分如下的程序包
package QUEUES is
  EMPTY;exception;
  type QUEUE is limited private;
  procedure JOIN(Q,in out QUEUE,X,in ITEM);
  procedure REMOVE(Q,in out QUEUE,X,out ITEM);
  function LENGTH(Q,QUEUE) return INTEGER;
private
```

队列项在一端加入而在另一端移去,因此遵循常规的先来先服务协议。如试图从空队列中移去一项将引发 EMPTY 异常。用单链接表实现队列,有两个指针分别指向队列两端,以避免对队列的扫描。函数 LENGTH 返回队列中的项目数,也不用扫描队列。

11.5 存取类型与约束

存取类型也可指向数组和可判别记录类型,且均可受约束或不受约束。

如家族树的问题,可以说明为:

```
type PERSON;
type PERSON_NAME is access PERSON;
type PERSON is
  record
    SEX,GENDER;
    BIRTH,DATE;
```



```
SPOUSE;PERSON_NAME;
FATHER;PERSON_NAME;
FIRST_CHILD;PERSON_NAME;
NEXT_SIBLING;PERSON_NAME;
end record;
```

这个模式假定了一个一一对应的合法的家庭，孩子通过分量 NEXT_SIBLING 连接在一起，母亲被标识为父亲的配偶。

然而，用可判别类型表示人可能更好，对不同性别可用不同的分量，更为重要的是可应用适当的约束。如：

```
type PERSON(SEX;GENDER);
type PERSON_NAME is access PERSON;
type PERSON(SEX;GENDER) is
record
    BIRTH;DATE;
    FATHER;PERSON_NAME(MALE);
    NEXT_SIBLING;PERSON_NAME;
case SEX is
    when MALE=>
        WIFE;PERSON_NAME(FEMALE);
    when FEMALE=>
        HUSBAND;PERSON_NAME(MALE);
        FIRST_CHILD;PERSON_NAME;
end case;
end record;
```

PERSON 的不完整说明也给出了判别式(以及任何默认初始表达式)，当然，必须与后面完整说明中的一致。现在分量 FATHER 总是被约束为存取男性(或 null)，类似地，分量 WIFE 和 HUSBAND 也被约束。注意这里必须有不同的标识符而不能都是 SPOUSE。然而，分量 FIRST_CHILD 和 NEXT_SIBLING 没有被约束，因此能存取任何性别的人。我们还可使孩子仅属于母亲以节省空间。

当类型 PERSON 的对象由分配符创建时，必须为判别式提供一个值，可给出显式的初始值

```
JANET:=new PERSON'(FEMALE,(22,FEB,1967),JOHN,null,null,null);
```

或者提供一个判别式约束，如：

```
JANET:=new PERSON(FEMALE);
```

这两种情况有些区别，前者需要一个引号，而后者不要引号。这是因为第一种情况取受限定的表达式形式，而第二种情况仅为子类型表示法。

我们不能写

```
JANET:=new PERSON;
```

这是因为类型 PERSON 没有默认判别式。然而可以说明

```
subtype WOMAN is PERSON(FEMALE);
```

然后再写

```
JANET:=new WOMAN;
```

这样的对象以后不能改变它的判别式，这个规则甚至当判别式具有默认初始表达式时也适用。由分配符产生的对象在这方面不同于由常规对象说明产生的对象，读者可以回忆一下，在常规方式下，默认初始表达式允许无约束的对象被说明，且以后可改变其判别式。

另一方面，我们知道除了没有判别式的默认初始表达式以外，还是能说明类型 PERSON_NAME 的无约束对象的。当然，这种对象取默认初值 null，也不会出什么问题，因此，尽管已分配对象不能改变其判别式，一个无约束的访问对象有时还是能指向具有不同判别式的对象的。

一个已分配对象不能改变其判别式的原因是，它能被如分量 FATHER 这样的若干个约束对象存取，故很难保证这样的约束没有冲突。

为了方便起见，可以定义子类型

```
subtype MANS_NAME is PERSON_NAME(MALE);
```

```
subtype WOMANS_NAME is PERSON_NAME(FEMALE);
```

现在可写一个过程进行结婚登记

```
procedure MARRY(BRIDE:WOMANS_NAME;GROOM,MANS_NAME) is
begin
    if BRIDE.HUSBAND/=null or GROOM.WIFE/=null then
        raise BIGAMY;
    end if;
    BRIDE.HUSBAND:=GROOM;
    GROOM.WIFE:=BRIDE;
end MARRY;
```

当参数传递时其约束将被检查（记住存取参数总是由保留副本实现）。试图使同性结合在调用时将引发 CONSTRAINT_ERROR 异常。另一方面，试图使不存在的人结合将导致在过程体内引发 CONSTRAINT_ERROR 异常。请记住，尽管 in 型参数是常量，我们也不能改变 BRIDE 和 GROOM 的值以存取不同的对象，但能改变存取对象的分量。

如下例，函数将返回一个存取值。

```
function SPOUSE(P:PERSON_NAME) return PERSON_NAME is
begin
    case P.SEX is
        when MALE=>
            return P.WIFE;
        when FEMALE=>
```



```
    return P. HUSBAND;
  end case;
end SPOUSE;
```

该函数的调用结果能被直接用作名字的一部分,因此可以写

SPOUSE(P). BIRTH

以给出 P 的配偶的生日。(见 7.1 节的末尾)甚至可以写:

SPOUSE(P). BIRTH := NEWDATE;

但这仅仅当该函数返回一个存取值时才可能。如果函数实际上返回的是类型 PERSON 的值而非类型 PERSON_NAME 的值,那么是不能实现的。如试图以某人取代原配偶,我们不能写

SPOUSE(P) := Q;

而可写为

SPOUSE(P). all := Q. all

将配偶的所有分量改成与 Q 的一样。

下面是登记新生婴儿的函数,该函数以母亲、婴儿的性别和生日作为参数。

```
function NEW_CHILD (MOTHER; WOMANS_NAME,
                     BOY_OR_GIRL; GENDER; BIRTHDAY; DATE)
  return PERSON_NAME is
  CHILD; PERSON_NAME;
begin
  if MOTHER. HUSBAND = null then
    raise ILLEGITIMATE;
  end if;
  CHILD := new PERSON(BOY_OR_GIRL);
  CHILD. BIRTH := BIRTHDAY;
  CHILD. FATHER := MOTHER. HUSBAND;
  declare
    LAST; PERSON_NAME := MOTHER. FIRST_CHILD;
  begin
    if LAST = null then
      MOTHER. FIRST_CHILD := CHILD;
    else
      while LAST. NEXT_SIBLING /= null loop
        LAST := LAST. NEXT_SIBLING;
      end loop;
      LAST. NEXT_SIBLING := CHILD;
    end if;
  end;
end;
```

```
    return CHILD;
end NEW_CHILD;
```

注意,判别式约束不必是静态的——BOY_OR_GIRL 的值直到函数被调用时才知道,因此由于不知道需提供哪些分量,不能用分配符给出完整的初值,而只能分配判别式的值给孩子,然后分别分配出生日期和父亲,其余分量取默认值 null。现在可以写为:

```
HELEN:PERSON_NAME:=NEW_CHILD(BARBARA,FEMALE,(28,SEP,1969));
```

存取类型也可指向约束或无约束数组,如:

```
type REF_MATRIX is access MATRIX;
R:REF_MATRIX;
```

然后用分配符可得到新矩阵

```
R:=new MATRIX(1..10,1..10);
```

该矩阵可被初始化

```
R:=new MATRIX'(1..10=>(1..10=>0.0));
```

但至于可判别记录,不能写成

```
R:=new MATRIX;
```

因为所有数组对象必须有界限,且界限不能改变。然而,R 有时能指向不同界限的矩阵。

我们期望创建一个子类型

```
subtype REF_MATRIX_3 is REF_MATRIX(1..3,1..3);
R_3:REF_MATRIX_3;
```

这样,R_3 只引用具有相应界限的矩阵。另外,还可以写成

```
R_3:REF_MATRIX(1..3,1..3);
```

利用

```
subtype MATRIX_3 is MATRIX(1..3,1..3);
```

由于子类型提供了数组界限,我们能写

```
R_3:=new MATRIX_3;
```

就象由于子类型 WOMAN 提供了判别式 SEX,可以写

```
JANET:=new WOMAN;
```

一样。

这里引入了带有 others 的数组聚集的另一个例子。因为子类型 MATRIX_3 提供数组界限并且限定聚集,可以初始化新对象如下:

```
R_3:=new MATRIX_3'(others=>(others=>0.0));
```

存取数组的分量能由通常的机制指示,因此

```
R(1,1):=0.0;
```

将由 R 存取的矩阵的分量(1,1)值置为零。整个数组可由 all 表示,故

```
R_3.all=(1..3=>(1..3=>0.0));
```

将由 R_3 存取的矩阵的所有分量值置为零。可以认为 R(1,1) 是 R.all(1,1) 的缩写。正如记录的情况一样,解除引用是自动进行的。我们还能写属性 R'FIRST(1) 或 R.all'FIRST(1),对一维

数组来说还可以分片。

练习 11.5

1. 编写返回继承人的函数。继承权的法则是：如有儿子则长子为继承人，否则为长女。如果没有继承人则返回 null。
2. 编写一个离婚的过程。只有当没有孩子时才允许离婚。

11.6 派生类型

有时引入一个与现有类型相似的新类型是很有用的。如果 T 是一个类型，可以写

```
type S is new T;
```

称 S 为派生类型，T 是 S 的父类型。

派生类型与其父类型属于同样的类别，例如，如果 T 是记录类型则 S 也是记录类型，且其分量具有相同的名字。

必须记住，区分类型的关键在于其值集合和操作集合，下面就这些方面进行讨论。

派生类型的值集合是其父类型值集合的副本。关于这一点的一个重要例子是，如果父类型是存取类型，那么派生类型也是存取类型，且共享同样的集合。注意我们说值集合是一个副本，这表明派生类型与其父类型确实是不同类型，一个类型的值不能被赋给另一个类型的对象。然而，正如我们将见到的那样，两个类型之间的转换是可能的。字面值和聚集（如果有的话）的标记是相同的，类型或其分量的任何默认初始表达式也是相同的。

适用于派生类型的操作如下。首先它具有与父类型相同的属性。第二，除非父类型是受限类型（这样，派生类型也是受限类型），赋值和预定义等式以及不等式也适用。第三，派生类型将派生或继承某些适用于父类型的子程序。（我们说子程序适用于一个类型，是指子程序具有该类型或子类型的一个或多个参数或结果）。这样的派生子程序在派生类型定义的地方被隐含说明，但以后可在同一说明域内被重新定义。

如果父类型是预定义类型，则继承子程序正好就是预定义子程序；如果父类型是用户定义类型，那么也有一些预定义子程序如“<”将被继承；如果父类型本身也是派生类型，则继承子程序再次被传递。

此外，如果父类型在程序包可见部分说明，那么只要派生类型在可见部分之后说明，在可见部分说明的适用子程序也将被继承。因此，新子程序在这种意义上仅真正“属于”这样的类型，即当到达可见部分结尾时利用该类型能派生出这些子程序。这非常合理，因为它基于这样的观点，就是类型的定义和它的操作在抽象的意义上被认为是完整的。

现在让我们举例来说明这些规则。如果我们有

```
type INTEGER_A is new INTEGER;
```

那么，INTEGER_A 将继承所有诸如“+”、“-”、“abs”这样的子程序以及 FIRST 和 LAST 等属性。

如果从 INTEGER_A 派生另一类型，那么这些继承子程序会再被传递。然而，假设 INTEGER_A 在程序包规范式说明部分说明并且程序包规范式说明部分包括另外的子程序，如：

```
package P is
    type INTEGER_A is new INTEGER;
```

```
procedure INCREMENT(I,in out INTEGER_A);
function"&"(I,J,INTEGER_A) return INTEGER_A;
end;
```



现在如果在规范式说明部分结束后(在程序包外或在其体内)说明

```
type INTEGER_B is new INTEGER_A;
```

那么 INTEGER_B 将继承新子程序 INCREMENT 和“&”以及预定义子程序。

如果我们认为某继承子程序不适用,那么可在同一说明域内重定义。因此可写

```
package Q is
```

```
type INTEGER_X is new INTEGER;
function "abs"(X,INTEGER_X) return INTEGER_X;
end;
```

这里,重新说明了“abs”以取代预定义操作符。

现在如果在程序包说明部分结束之后写

```
type INTEGER_Y is new INTEGER_X;
```

则将继承新定义的“abs”。

还有一些较次要的规则。私有类型要到其完整类型说明之后才能派生;同样,如果父类型本身是在程序包可见部分说明的派生类型(如 INTEGER_A),则其派生类型(如 INTEGER_B)不能在可见部分说明,这个限制对父类型不是派生类型的情况不适用。但要记住,这样的父类型的派生类型仅继承预定义子程序,而不继承新的或替换的子程序。

最后,必须认识到预定义类型实际上并不是特殊情况,它们和预定义子程序可认为是在程序包 STANDARD 的可见部分中说明。

尽管派生类型是不同的,但使用与数值类型间转换时用到的相同的标记法,可使值从一种类型转换为另一种。因此给出

```
type S is new T;
TX:T;
SX:S;
```

可以写

```
TX:=T(SX);
```

或者

```
SX:=S(TX);
```

但是

```
TX:=SX; --非法
```

或

```
SX:=TX; --非法
```

是不允许的,如果包含多重派生,那么只需要总的转换,而不必给出每个步骤。因此如果有

```
type SS is new S;
type TT is new T;
```

SSX,SS;

TTX,TT;

那么可以较麻烦地写

SSX:=SS(S(TX));

和

TTX:=TT(T(SX));

也可以简单地写为

SSX:=SS(TX);

和

TTX:=TT(SX);

派生类型的引入增强了6.2节讨论的数组类型间转换的可能性。事实上，如果分量类型相同并且下标类型相同或可相互转换，则一个数组类型能被转换成另一个数组类型。

引入转换后，现在我们可以更为详细地讨论派生子程序的有效说明。它从父类型子程序得到，只要简单地用新的派生类型替换原始说明中的父类型的所有场合：子类型由具有相应约束的等价子类型代替，默认初始表达式靠加入类型转换而被转换。其它不同类型的任何参数和结果保留不变。作为一个抽象例子，考虑

```
type T is ...;
subtype S is T range L..R;
function F(X:T;Y:T:=E;Z:Q) return S;
```

这里，E是类型T的表达式，类型Q与类型T不相关。如果写

```
type TT is new T;
```

那么相应地就有

```
subtype SS is TT range TT(L)..TT(R);
```

派生类型的规范式说明将是

```
function F(X:TT;Y:TT:=TT(E);Z:Q) return SS;
```

在这里我们用TT代替T,SS代替S,表达式E要进行转换，而保留不相关的类型Q不变。注意参数名自然是相同的。

现在，可以利用派生类型重写11.4节的类型STACK如下

```
type CELL;
type LINK is access CELL;
type CELL is
record
    VALUE:INTEGER;
    NEXT:LINK;
end record;
type STACK is new LINK;
```

这样，函数“=”可定义为：



```
function "="(S,T;STACK) return BOOLEAN is
    SL:LINK:=LINK(S);
    TL:LINK:=LINK(T);
begin
    --类似练习11.4(2)的答案
end "=";
```

利用派生类型而不是使类型 STACK 作为具有一个分量的记录的好处是,11.4节的过程 PUSH 和 POP 不必修改仍然能工作,这是因为类型 STACK 仍然是存取类型,并且与类型 LINK 共享其集合。

派生类型经常用作私有类型,事实上,如要将私有类型表示成如 INTEGER 这样的已定义类型,必须用派生类型。

派生类型的另一用途是使我们能用已定义类型的操作,且避免概念上不同的类型的对象相混淆。例如,假设我们要对苹果和桔子计数,可以说明

```
type APPLES is new INTEGER;
type ORANGES is new INTEGER;
NO_OF_APPLES:APPLES;
NO_OF_ORANGES:ORANGES;
```

因为 APPLES 和 ORANGES 都是从类型 INTEGER 派生出来的,都继承“+”操作,因此可以写

```
NO_OF_APPLES:=NO_OF_APPLES+1;
```

和

```
NO_OF_ORANGES:=NO_OF_ORANGES+1;
```

但不能写

```
NO_OF_APPLES:=NO_OF_ORANGES;
```

如果确实需要转变桔子数为苹果数,必须写成

```
NO_OF_APPLES:=APPLES(NO_OF_ORANGES);
```

下章我们将更为详细地讨论数值类型,但这里值得提一下:严格地说象 INTEGER 这样的类型是没有直接量的,直接量(如1)属于称为一般整数的类型,而且如果上下文要求还可隐含转换到任何整数类型。因此1能与 APPLES 和 ORANGES 一起用,正是由于这个隐含的转换,而不是直接量被继承。另一方面,枚举直接量正好属于它们的类型且被继承,事实上,正如7.5节提到的,枚举直接量相当于无参函数,因此可认为与任何其它适用子程序一样能被继承。

再回到苹果和桔子问题上,假设有重载过程来卖苹果和桔子

```
procedure SELL(N:APPLES);
procedure SELL(N:ORANGES);
```

那么可写

```
SELL(NO_OF_APPLES);
```

但是

```
SELL(0);
```

的意义不明确,因为不知道要卖哪种水果。可以用限定来消除这种不明确性,如:

```
SELL(APPLE'(6));
```

当子程序被派生时并不是真正创建新子程序。派生子程序的调用实际上是父子程序的调用,in型和in out型参数恰好在调用前被隐式地转换;in out型或out型参数或函数的结果则刚好在调用后被隐式地转换。因此

```
MY_APPLES+YOUR_APPLES
```

实际上是

```
APPLES(INTEGER(MY_APPLIS)+INTEGER(YOUR_APPLES))
```

派生类型从某种意义上来说是私有类型的替换物。派生类型具有继承直接量的优点,但往往继承得过多。例如,我们能从REAL派生出类型LENGTH和AREA:

```
type LENGTH is new REAL;
```

```
type AREA is new REAL;
```

这样可防止混淆长度和面积,但也继承了很多不相关的操作,如长度相乘得到的单位还是长度,面积相乘得到的单位是面积,长度和面积可参加幂运算等等。当然,可以重新定义这些操作符使它们变得有用或引发异常,但通常用私有类型更简单,且只要定义所需要的操作。

作为应用派生类型的另一个例子,重新考虑练习9.3(1)的程序包BANK,在此程序包中,我们说明

```
subtype MONEY is NATURAL;
```

以引入含义明显的标识符MONEY,并保证钱为正。然而,由于MONEY是子类型,我们仍有可能错误地与整数相混。如果说明

```
type MONEY is new NATURAL;
```

会更好一些,这样就产生了适当的区别。

在答案所示的私有部分中,我们可类似地写

```
type KEY_CODE is new INTEGER range 0..MAX;
```

上面的情况说明,我们能用类型标记或者更通常的子类型表示从子类型派生出类型。派生类型实际上是匿名的,是从基类型派生出来的。就KEY_CODE而言就好象我们已经写了

```
type anon is new INTEGER;
```

```
subtype KEY_CODE is anon range 0..anon(MAX);
```

因此KEY_CODE实际上不是类型而是子类型。新派生类型的值集合实际上是类型INTEGER的值的全集,派生操作“+”、“>”等也在值的全集上工作。因此给出

```
K:KEY_CODE;
```

后,尽管-2永远不能赋给K,但我们能合法地写

```
K>-2
```

当然,这个布尔表达式总是为真。

读者可能会感到派生类型并不是十分有用的,因为我们已经看到在这个例子中还有另一种机制适用——通常包括记录类型或私有类型。然而,派生类型的重要性在于它是下章讨论的数值类型机制的基础。

我们提出一个有关BOOLEAN类型的奇异现象来结束本章,这种异常现象到目前为止还

不是普通用户所关心的。如果我们从 BOOLEAN 派生出类型，那么预定义关系运算符 =、< 等继续产生预定义 BOOLEAN 类型的结果，而逻辑运算符 and, or, xor 和 not 却被正常地继承，并产生派生类型的结果。这里，除了说这是由于 BOOLEAN 类型的基本特性外，不能归究于别的原因。然而，这意指练习 4.7(3) 中 xor 和 /= 是等价的定理仅适用于 BOOLEAN 类型，而不适用于从它派生的类型。

练习 11.6

- 说明一个包含类型 LENGTH 和 AREA 的程序包，对不正确的操作“*”进行适当的重定义。

要点 11

- 如果判别式没有默认表达式则所有对象必须被约束。
- 一个无约束对象的判别式只能由整个记录赋值而改变。
- 判别式能被用作数组界限，或用来控制变体，或作为嵌套判别式，或用在分量的默认初始表达式中。
 - 处在聚集中或控制变体的判别式必须是静态的。
 - 任何变体必须出现在分量表的最后。
 - 不完整的说明只能用在存取类型中。
 - 被存取对象的作用域就是存取类型的作用域。
 - 如果一个被存取对象具有判别式则它总是受约束的。
 - 返回存取值的函数能被用作名字的一部分。
 - 存取对象具有默认初值 null。
 - 聚集中的分配符给每个下标求值。
 - 具有完整初值的分配符使用引号。



第12章 数值类型

现在,让我们对数值类型进行更为详细的讨论。在 Ada 中,有两类数值类型:整数类型(整型)和实数类型(实型),而实型又可进一步分为浮点型和定点型。

计算机中与数值类型表示有关的问题有两个。第一,数的表示范围受到限制,数的表示范围越大,所占据的存贮空间就越大。实际上,很多计算机针对不同的表示范围都有相应的硬件操作供用户选择。第二,精度问题,要精确地表示某一类型所有可能的值,恐怕是不可能的。这两个问题直接影响到程序的可移植性,因为不同的机器对数值的限制各不相同。Ada 对可移植性予以充分的重视,采取了一个比较有效的措施,即可以指定类型的最小取值范围,从而使程序员将移植方面存在的问题降到最低程度。

我们先讨论整型,因为它只牵涉到范围而不牵涉精度问题。

12.1 整型

在 Ada 中设有预定义类型 INTEGER,此外,不同实现还有诸如 LONG_INTEGER,SHORT_INTEGER 等其它预定义类型。除了在二进制补码机器中一个特殊负值外(该值超出二进制反码机器数值范围),上述预定义类型值的范围与 0 对称。所有预定义整型都有同样的预定义操作。

假设在机器 A 上有类型 INTEGER 和 LONG_INTEGER,其中:

range of INTEGER:
-32768..+32767 (即16位)

range of LONG_INTEGER:
-21474_83548..+21474_83847 (即32位)

而在机器 B 上有类型 SHORT_INTEGER,INTEGER,LONG_INTEGER,其中:

range of SHORT_INTEGER:
-2048..+2047 (即12位)

range of INTEGER:
-83_88608..+83_88607 (即24位)

range of LONG_INTEGER:
-14073_74883_55328..+14073_74883_55327 (即48位)

迄今所有例子中,我们仅使用 INTEGER 类型,因为对多数情况,这已够用了。倘若需要的数值达百万之大,那么 A 机上的 INTEGER 类型是不能满足的,用 B 机上的 LONG_INTEGER 类型又显得过于浪费。此时可用派生类型来加以解决,以 A 机为例先写出:

```
type MY_INTEGER is new LONG_INTEGER;
```

然后,在我们的程序中始终使用 MY_INTEGER。

如果把程序从 A 机移到 B 机上,只需把上述说明改为:

```
type MY_INTEGER is new INTEGER;
type MY_INTEGER is range -1E6..1E6;
```

Ada 便自动地从所有符合条件的类型中选择范围最小的一个,如同写出:

```
type MY_INTEGER is new LONG_INTEGER range -1E6..+1E6;
```

或者

```
type MY_INTEGER is new INTEGER range -1E6..+1E6;
```

一样。实际上,MY_INTEGER 是从预定义类型之一派生的一个匿名类型的子类型,因而 MY_INTEGER 类型的对象值被约束在 -1E6..+1E6 范围内,而不是该匿名类型的整个范围。这个值域是静态的,因为在编译阶段就已确定了。

如果需要,利用 MY_INTEGER'BASE'FIRST 和 MY_INTEGER'BASE'LAST 可以得知匿名类型的整个值域。

属性 BASE 适用于任何类型或子类型,用来指出对应的基类型,但 BASE 只能辅助于其它类型来使用。必要时,甚至可以借助下列类型:

```
type X is range -1E6..+1E6;
```

```
type MY_INTEGER is range X'BASE'FIRST..X'BASE'LAST;
```

使用预定义类型的整个值域。但对 MY_INTEGER 类型的对象赋值时,是否可以免去约束检查是令人怀疑的,因为它破坏了可移植性。

利用一般的类型转换方法,可以在两类整数之间进行转换。考虑

```
MY_INTEGER is range -1E6..1E6;
type INDEX is range 0..10000;
M:MY_INTEGER;
I:INDEX;
```

然后,执行下列两个赋值语句:

```
M:MY_INTEGER(I);
I:=INDEX(M);
```

都是允许的。因为无需通过每个派生类型的转换,就可直接将一种数值类型转换成另一种数值类型。否则必须在 A 机上写出:

```
M:=MY_INTEGER(LONG_INTEGER(INTEGER(I)));
```

在 B 机上写出:

```
M:=MY_INTEGER(INTEGER(I));
```

从而使可移植性丧失。

整型直接量属于通用类型。这种类型没有界限,至少与预定义类型相当,且具有一般操作如 +, -, < 和 =。整型数在整数说明中加以说明(见 4.1 节),例如:

```
TEN:constant:=10;
```

中的 10 也属于通用整型。然而,没有通用整型变量,因此,大多数通用整型表达式均为静态的,这种表达式可精确地计算出。数值说明中的初值也必须是通用整型表达式,例如:

```
M:constant:=10;
```



MM;constant:=M * M;

都是允许的,但

N;constant INTEGER:=10;

NN;constant:=N * N;

却是不允许的,因为 N * N 不是通用整型的静态表达式,而是 INTEGER 类型的表达式。

需要指出的是,某些属性例如 POS 实际上可提供通用整型值,又由于 POS 允许设置动态参数,在这种意义下某些通用整型表达式实际上是动态的。

整型直接量、通用整型属性、整型数到其它整型的转换是自动进行的,并不需要显式的类型转换,但在容易产生歧义的场合要给予显式的表示。

读者可以回顾一下,在 FOR 语句或数组类型定义中范围 1..10 被认为是 INTEGER。确切地说范围应该是整型直接量或整型数或通用整型属性,而非一般表达式。顺便指出,表示范围的若为 INTEGER 而决非任何其它整型则意味着预定义类型 INTEGER 有着十分特殊的特性。

使用反映程序要求的整型说明,而不是动辄就使用 INTEGER 或 LONG_INTEGER 类型是个好习惯。因为这不仅有助于可移植性,而且易于区别诸对象的不同类型。前一章计算苹果数和桔子数就是一个例子。

然而,彻底的可移植性是不易得到的。

考虑

```
type MY_INTEGER is range -1E6..+1E6;
```

```
I,J;MY_INTEGER;
```

```
J:=I * I;
```

要搞清此例,重要的是要记住 MY_INTEGER 实际上只从 A 机上的 LONG_INTEGER 或从 B 机上的 INTEGER 派生出来的匿名类型的子类型。“+”,“-”,“*”,“/”等派生操作的操作数和结果均为匿名类型,而其子类型 MY_INTEGER 的约束仅当打算对 J 赋值时才予以考虑。因此

```
J:=I * I;
```

在 A 机上等价于

```
J:=anon(LONG_INTEGER(I) * LONG_INTEGER(I));
```

上面的乘法是借助 LONG_INTEGER 类型的操作来实现的,结果为 1E10,这恰好处在 LONG_INTEGER 的范围之内。但是,当试图将结果赋给 J 时便会引发约束异常。

在 B 机上,I 的类型是从 INTEGER 派生的,此时乘法运算的结果超出 INTEGER 的范围,从而引发数值异常。

上例说明,尽管同一程序在 A,B 机上均出问题,但引发的异常却是不同的。更为糟糕的是,同一程序在一个机器上执行正常,而移植到另一个机器上却会出问题。例如:

```
I:=100000;
```

```
J:=(I * I)/100000;
```

在 A 机上,其中间结果和最终结果均能正确计算出来,且最终结果落在 J 的类型范围内。但在 B 机上,却会引发数值异常 NUMERIC_ERROR。(以上分析忽视了选择最佳处理的可能性,Ada 语言允许使用值域更宽的类型,以避免引发数值异常,从而得到正确的结果。但,如果 B 机没有

LONG_INTEGER,此例得不到正确结果)。

上面所涉及到的最大正整数和最小负整数,分别由 SYSTEM, MAX_INT 和 SYSTEM, MIN_INT 表示,它们都是预定义程序包 SYSTEM 中说明的常数。

练习 12.1

1. 下述 P,Q,R 在机器 A 和 B 中表示什么类型?

```
type P is range ..1000;  
type Q is range 0..+32768;  
type R is range -1E14..+1E14;
```

2. 如果机器 A 和 B 都是补码机且具有相同的 bit 位,试问 A 和 B 有何区别?

3. 如果

```
N;INTEGER:=6;  
P;constant:=8;  
R;MY_INTEGER:=4;
```

则下式为何类型,

- a)N+P d)N * N
- b)N+R e)P * P
- c)P+R f)R * R

4. 试定义一个由已知机器支持的最大整数类型 LONGEST_INTEGER.

12.2 实型

整型是精确的,而实型是近似的。后者有个精度问题。

实型分为浮点类型和定点类型。浮点类型具有相对误差而定点类型具有绝对误差。本节对浮点型和定点型的一般概念进行讨论,在后续章节再对每种类型进行分别讨论。

有一种通用实型,它的特征与通用整型很相似。实型直接量属于通用实型。通用实型在概念上是无界的和无限精确的。与实型的通常操作一样,允许通用实型与通用整型进行混合操作,在除法运算时第一个操作数是通用实型,第二个操作数是通用整型是允许的。一个通用实型的数可乘以一个通用整型的数,反之亦然。这些混合运算的结果均为通用实型。

例如:

```
TWO_PI;constant:=2 * PI;  
TWO_PI;constant:=2.0 * PI;
```

但

```
PI_PLUS_TWO;constant:=PI + 2;
```

却是错误的,因为混合加法是不允许的。注意,不能在通用整型和通用实型之间进行显式的类型转换,但可以通过乘以 1.0 将前者转换为后者。

样板数(model number)是一个重要的概念。在说明一个实型 T 时,总要求它具有一定的精度。为保险起见,一般常采用一个更高的精度。正如一个整型的实现,采用一个比要求有更大范围的基类型一样。与实数要求的精度对应的是一组精确表示的样板数。因为样板数所表示的精度一般是较高的,因此其它值也被精确地表示了。每个值与一样板区间有关。如果某值是个样板数,那么其样板区间就是这个样板数,否则该样板区间就是由包围该值的两个样板数围成的区间。当某个值大于最大样板数 T'LARGE 时出现特殊情况。

对给定数进行某操作,其结果的界限是对该操作数的样板区间中任意数施行同一操作所产生的最小样板区间。

关系运算符“=”,“>”等也根据样板区间定义。若无论在该区间选取什么值,结果总相同,那么其值显然是没有异议的。若结果依赖于在该区间中(取值的先后次序),则结果是不确定的。

样板数的应用在下一节举例说明。

练习 12.2

1. 给定以下条件:

```
TWO,INTEGER:=2;  
E,constant:=2.71828_18285;  
MAX,constant:=100;  
试问下列各式为何类型?  
a)TWO * E      d)TWO * TWO  
b)TWO * MAX    e)E * E  
c)E * MAX      f)MAX * MAX
```

2. 已知 N,constant:=100,试定义一个与 N 具有相同值的实数 R。

12.3 浮点类型

与整型一样,Ada 提供了预定义类型 FLOAT,在此基础上还提供了预定义类型 SHORT_FLOAT, LONG_FLOAT 等。它们分别具有更小和更大的精确度。这些类型都有相应的预定义操作。

我们可以根据精度要求直接派生出所需的类型。例如:

```
type REAL is new FLOAT;  
type REAL is new LONG_FLOAT;
```

但更好的方法是象整型指出范围那样,指出所要求的精度,并允许进行适当的选择。例如:

```
type REAL is digits 7;
```

表示从预定义类型中派生出至少有七位十进制数字精度的 REAL 类型。

```
type REAL is digits D;
```

其中 D 是一个静态的整型(取正值)表达式,它表示所要求的十进制位数。我们先将它转换成具有相同精度的二进制位数 B。B 取大于 $D \cdot \log_{10} 10$ 的最小整数,即

$$B - 1 < 1 + (3.3219 \dots \times D) < B$$

二进制精度 B 决定了样板数。这些样板数被定义为零和所有下列形式的数:

<符号>. <尾数> $\cdot 2^{<\text{指数}>}$

其中

<符号> 是 +1 或 -1

$1/2 \leqslant <\text{尾数}> < 1$

$-4B \leqslant <\text{指数}> \leqslant +4B$

当以 2 为基时,该尾数在二进制小数点之后正好有 B 位数字。指数是一个在 $[-4B, +4B]$ 范围内的整数。例如:

```
type REAL is digits 7;
```

确保所选择的预定义浮点型样板数的精度为7位十进制数字。

我们假定一种极端情况：

```
type ROUGH is digits 1;
```

则 D 为 1, 因而 B 为 5。样板数的尾数可以取下列值之一：

$$\frac{16}{32}, \frac{17}{32}, \dots, \frac{31}{32}$$

而二进制指数位于【-20, 20】中。

数“1”周围的样板数为：

$$\dots, \frac{30}{32}, \frac{31}{32}, 1, 1\frac{1}{16}, 1\frac{2}{16}, 1\frac{3}{16}, \dots$$

数“0”周围的样板数为：

$$\dots, -\frac{17}{32} \cdot 2^{-20}, -\frac{16}{32} \cdot 2^{-20}, 0, +\frac{16}{32} \cdot 2^{-20}, +\frac{17}{32} \cdot 2^{-20}, \dots$$

请注意，“1”附近绝对精度的变化和“0”周围的空隙，“0”与最小样板数之间的空隙是其它两个相邻样板数之差的16倍。可见，在“0”附近的精度显著降低。上述样板数由图12.1所示：

(a) 1周围样板数

… × × × × × × × × …
1

(b) 0周围样板数

… × × × × × × × …
0

图12.1 类型 ROUGH 的样板数

其中最大几个样板数为：

$$\dots, \frac{29}{32} \cdot 2^{20}, \frac{30}{32} \cdot 2^{20}, \frac{31}{32} \cdot 2^{20}$$

或者

$$\dots, 950272, 983040, 1015808$$

考虑下列赋值语句：

```
R : ROUGH := 1.05;
```

我们不需考虑为 R 选择哪一个预定义浮点类型，直接量 1.05 将被转换为它附近的样板数 1 或 $1\frac{1}{16}$ （并不知道哪一个！），我们唯一知道的是 S 值落入样板区 $[1, 1\frac{1}{16}]$ 内。

当计算 $R := R * R$ 时，计算结果一定处于 $1^2 = 1$ 和 $\left(1\frac{1}{16}\right)^2 = \frac{289}{256} = 1\frac{33}{256}$ 之间，由于 $1\frac{33}{256}$ 不是样板数，故需取一个样板数，它大于结果的上界 $1\frac{33}{256}$ ，此样板数为 $1\frac{3}{16}$ 。由此得出，结果 R 的值落入样板区 $[1, 1\frac{3}{16}]$ 内。

那么结果 R 到底如何表示呢？

若写成 $R > 1.0$ 则不妥，因为有可能 $R = 1$ ，则 $1 > 1$ 显然是错误的，而 $1\frac{3}{16} > 1$ 则又是正确的。

若写成 $R = 1.0$ 也不妥，因为有可能 $R = 1$ ，则 $1 = 1$ 是正确的，而 $1\frac{3}{16} = 1$ 显然又是错误的。

因此写成

R>=1.0

比较妥当。由此知，在浮点类型的运算中，诸如=,>和>=这一类关系符用起来应小心。

以上例子虽然比较极端，但却说明了由误差带来的危险。前面已指出，“0”周围的空隙特别大，一旦结果落入该空隙内，便出现下溢，所有精度全部丢失，出现令人瞠目结舌的后果。

实际情况要比此极端情况好得多，因为 ROUGH 类型是从一个具有更高精度的预定义类型中派生出来的。假定在一特定机器上，只有一个 D=7 的预定义类型 FLOAT，此时

type ROUGH is digits 1;

等价于

```
type anon is new FLOAT;
subtype ROUGH is anon digits 1;
```

因此，ROUGH 类型实际上是派生类型 anon 的一个子类型，施加于 ROUGH 类型值之上的操作是借助于 FLOAT 类型的样板数来完成的。为了更准确地描述实现，我们引入“安全数”(safe number)的概念。

类型的安全数是样板数的集合及与样板数有同样的运算特性。安全数与 B 有相同的位数，但指数范围却是 $\pm 4E$ ，其中 E 至少等于 $4B$ ，这样安全数扩大了可靠范围并且能够让实现利用硬件特点。样板数与安全数的一个重要区别是：子类型的安全数就是基类型的安全数，而子类型的样板数却是由子类型的定义中所要求的精度隐含地给出的。

例如，ROUGH 类型的安全数与 FLOAT 类型的安全数一样，然而 ROUGH 类型的样板数却是通过定义 D=1 而得出。对 ROUGH 类型的操作也适用于安全数。安全数存贮在类型 ROUGH 的对象中。

如果我们想写一个可移植的程序，则必须遵循样板数的特性。反之，如果我们希望得到具体实现尽可能多的支持则可使用安全数的特性。在实现中为安全数提供了附加数，但从某种意义而言，这些附加数是不可靠的。因为除了用样板数的区间或安全数的区间隐含定义的那些数之外，Ada 没有对附加数进行定义。

我们可以设置一个对浮点子类型或浮点对象范围的约束，例如：

R:ROUGH range 0.0..100.0;

或者

subtype POSITIVE_REAL is REAL range 0.0..REAL/LAST;

倘若违背了限定的范围则引发 CONSTRAINT_ERROR。

Ada 提供了许多属性来帮助用户编写可移植的程序。F 为任意预定义或非预定义的类型（或子类型）。

F'DIGITS

十进制位数 D,

F'MANTISSA

表示对应的二进制的位数 B,

F'EMAX

最大的指数, $4 * F'MANTISSA$,

F'SMALL

最小的正样板数, $2.0 * * (-F'EMAX - 1)$,

F'LARGE

最大的正样板数, $2.0 * * F'EMAX * (1.0 - 2.0 * * (-F'MANTISSA))$,

F'EPSILON

表示样板数 1.0 与比 1.0 大的最小样板数之差的绝对值:

2. $0 * (-F'MANTISSA + 1)$ 。

另外还有 F'SAFE_EMAX, F'SAFE_SMALL 和 F'SAFE_LARGE 等属性, 它们给出了 F 类型的安全数的特性。

还有常见的属性为 F'FIRST 和 F'LAST, 它们是实际实现时的两个端点值, 不一定是样板数。

DIGITS, MANTISSA, EMAX 和 SAFE_EMAX 是通用整型; SMALL, SAFE_SMALL, LARGE, SAFE_LARGE 和 EPSION 是通用实型; FIRST 和 LAST 是 F 类型。

还可以利用 BASE 属性求出实现时的类型。

因此, 若

ROUGH'DIGITS = 1

则有

ROUGH'BASE'DIGITS = 7

我们不想更多地谈论浮点数, 但是希望读者能明白有关的原则。一般情况下, 使用者只需简略地作表示精度的工作, 只有当出现疑问时, 才用到专门的数值分析。更为详细的内容, 读者可参阅 LRM。

练习12.1

1. ROUGH'EPSILON 的值是什么?

2. 根据以下的式子来确定样板数的区间。

```
type REAL is digits 5;
P,REAL:=2.0;
Q,REAL:=3.0;
R,REAL:=(P/Q)*Q;
```

3. 执行下式会产生什么样的结果?

```
P,constant:=2.0;
Q,constant:=3.0;
R,real:=(P/Q)*Q;
```

4. 类型 ROUGH 有多少样板数?

5. 有下列函数

```
function HYPOTENUSE (X,Y,REAL) return REAL is
begin
    return SQRT(X**2+y**2);
END;
```

如果 X 和 Y 都很小, 则产生下溢, 将上式改写, 使其含有对 X 和 Y 的检查和调整。

6. 解释为什么 B 不是刚好比 D.log 大的数而是比 D.log 大1的数。用 ROUGH 类型来解释你的答案。

7. 0 与相近的第一个样板数之间的间隔, 与每两个样板数之间的间隔的比值是多少?

12.4 定点类型

定点类型通常只用于特殊的应用或在没有浮点硬件的机器上完成近似计算。本书只对其作简单的介绍。

从预定义类型引入的一般原则也适用于定点类型。然而与整型或浮点型不同的是, 预定义定点类型是匿名的, 不能被直接使用, 必须通过定点说明后方能使用。这种说明指出一个绝对



误差和一个取值范围。其形式如下：

```
type F is delta D range L..R;
```

说明了定点类型 F 的取值范围是在 L 到 R 之间，精度是 D(D 为正数)。D,L,R 须为静态的实型数。上式又可被表示为：

```
type anon is new F;
```

```
subtype F is anon range anon(L)..anon(R);
```

这里的 F 是一个适当的预定义匿名的定点类型。

定点类型(或子类型)可用两个属性来表示，一个是正整数 B，而另一个是正的实数 small，B 表示定点类型数的二进制位数(不包括符号位)，small 是绝对精度。样板数为 0 加上所有形式如下式的数

符号 × 尾数 × 绝对精度，这里符号取 +1 或 -1：

$$0 < \text{尾数} < 2^B$$

定义

```
delta D range L..R;
```

意味着以 2 的最大次幂表示的 small 小于或等于 D，而以最小整数 B 表示的定点数，其样板数的范围不超过 L 和 R。

例如：

```
type T is delta 0.1 range -1.0..+1.0;
```

在此 small 为 $\frac{1}{16}$ ，B 为 4，因而 T 的样板数为：

$$-\frac{15}{16}, -\frac{14}{16}, \dots, -\frac{1}{16}, 0, +\frac{1}{16}, \dots, +\frac{14}{16}, +\frac{15}{16}$$

L 和 R 均在样板数的范围之外，分别与最小和最大样板数只差一个 small 值。

假定我们是在 16 位机上实现，则派生出 T 类型的预定义类型便有很大的选择余地。选取这些预定义类型的唯一标准是，该预定义类型的样板数要能够包含 T 类型的样板数。假如某个预定义类型有 B=15，一种极端的情况，small 为 2^{-16} 。由此，对于指定的取值范围给很高的精度。反之，另一种极端的情况，small 为 $\frac{1}{16}$ ，则由此对于指定的精度给出很大的取值范围。

如果我们选择一个具有最高精度的预定义类型。那么它的样板数则为：

$$-\frac{2^{15}-1}{2^{15}}, \dots, -\frac{1}{2^{15}}, 0, +\frac{1}{2^{15}}, \dots, +\frac{2^{15}-1}{2^{15}}$$

对定点数，也定义了安全数的概念。某类型的安全数可以是其基类型的样板数，而预定义类型的安全数就是它的样板数本身。所以 T 类型的安全数即为由其选择的预定义类型的样板数。如同在浮点型移植性方面所述那样，如果要设计出可移植的程序，则必须掌握样板数的概念，若想最大限度地脱离具体实现，则必须依赖于安全数。

让我们再回到 T 类型来展开讨论，我们可能用一个 8 位机来表示此定点类型。为此，我们进一步用到表示子句(关于此子句的详细介绍见第 15 章)向编译程序指明有关事宜，比如：

```
for T'SIZE use 5;
```

此子句迫使实现至少用 5 个二进位。然而编译程序也可拒绝执行不合理的表示子句。

在使用表示子句时可不考虑 small 是 2 的指数次幂这一规则，如果写：

```
for T'SIZE use 5;
```

```
for T'SMALL use 0.1;
```

则得到的样板数为：

-1.5, -1.4, ..., -0.1, 0, +0.1, ..., +1.4, +1.5

这种情况下，预定义类型与 T 类型有相同的样板数。

small 取 2 的指数次幂这一默认的标准值，其优点为：定点数和其它类型数之间的转换可直接借助移位实现。small 取其它值，则定点数和其它类型数之间的转换需要进行乘法和除法。

必须认识到预定义定点类型在某种程度上说，其生命是短暂的，在需要使用它们的场合不是以预定义的类型出现而是以匿名类型的形式出现。因此，在实现上仅要求有匿名的预定义类型。正因为如此，作为某类型的 small 必须为 2 的指数次幂，任何指定 small 值为非 2 的指数次幂的企图均会失败。

下面给出一个标准且又是简单的例子：

```
DEL:constant:=2.0**(-15);  
type FRAC is delta DEL range -1.0..1.0;
```

在此，定点类型在 16 位的补码机器上以纯小数的形式表示，范围的上界写成 1.0 或者 1.0-DEL 都没有什么关系。因为无论上界为 1.0 或是 1.0-DEL，最大的样板数都是 1.0-DEL。

最后，给出一个有关 small 值的例子。此例用整个 16 位的字来表示一个角度 angle：

```
type ANGLE is delta 0.1 range -PI..PI;  
for ANGLE'SMALL use PI*2**(-15);
```

这里，由于用了表示子句 ANGLE'SMALL，故 small 的值与 delta 不相干。

“+”、“-”、“*”、“/”和“abs”等算术运算符均可施加于定点类型的值。

“+”和“-”只可施加于两个同样类型的值，结果类型不变。“*”和“/”运算可施加于两个不同定点类型的值，结果为通用定点类型。通用定点类型的值在执行其它运算之前，必须借助类型转换显式地转换为一特定类型。定点类型的值还可以与整数进行乘除运算，结果是该定点类型。例如，对于下列说明：

```
F,G:FRAC;
```

赋值语句：

```
F:=F+G;
```

是允许的，但是

```
F:=F*G;
```

却是非法的，必须将乘积的类型进行显式转换，如：

```
F:=FRAC(F*G);
```

同浮点运算一样，定点运算的特征也由样板数来定义。

定义类型及其子类型 F 的各种属性如下：

F'DELTA 表示 F 类型说明中所指的绝对误差 D

F'MANTISSA 表示二进制尾数的位数

F'SMALL 表示最小的正样板数

F'LARGE 表示最大的正样板数

另外，安全数也有相应的属性 F'SAFE_SMALL 和 F'SAFE_LARGE。应当指出，对于定点类型有：

$F'_{SAFE_SMALL} = F'_{BASE} / SMALL$

还有属性 F'_{FIRST} 和 F'_{LAST} , 这两个属性给出某个类型或某个子类型的上界和下界。

$DELF, SMALL, SAFE_SMALL, LARGE$ 和 $SAFE_LARGE$ 都是通用类型, $MANTISSA$ 为通用整型, $FIRST$ 和 $LAST$ 为 F 类型。

更详细的情况, 读者可参阅 LRM。

练习 12.4

1. F 是 $FRAC$ 类型, 计算 F 的样板数间隔, 这里 $F := 0.1$.

要点12

- 为了增强可移植性, 隐含地用到派生类型。
- 注意中间表达式的溢出情况。
- 注意由浮点0周围的空隙而产生的下溢情况。
- 注意与实型数有关的运算。
- 如有疑问可参考数值分析方面的内容。



第13章 类属

本章我们讨论类属机制。类属机制允许我们以数值、对象以及类型和子程序作为参数，将子程序和程序包参数化。



13.1 类属说明与类属例化

类型语言(如 Ada)的问题之一，即所有类型在编译时就被限定下来，也就是说我们不能把类型作为运行时的参数来传递。然而，经常存在这样的情况：某段程序的逻辑结构独立于它所操作的类型，因此，没有必要对于所有这段程序能适应的不同类型，都重复这个程序逻辑结构以分别对这些类型进行操作。考虑练习7.3(1)的 SWAP 过程：

```
procedure SWAP (X,Y;in out REAL) is
    T;REAL;
begin
    T:=X;X:=Y;Y:=T;
end;
```

可以看出，这段程序的逻辑结构独立于它交换的值的类型。如果我们还要交换整型或布尔型值，因此必须写出程序逻辑相同而仅仅是操作值类型有别的相应的过程，这样实在是冗余。为了克服这个问题，Ada 提供了类属机制。譬如，我们可以作说明：

```
generic
    type ITEM is private;
procedure EXCHANGE(X,Y;in out ITEM);
procedure EXCHANGE(X,Y;in out ITEM) is
    T;ITEM;
begin
    T:=X;X:=Y;Y:=T;
end;
```

过程 EXCHANGE 为类属子程序，它起到一个样板的作用。类属子程序的规范式说明以保留字 generic 开头，接下来是类属形参表(也可以没有)。类属子程序的体与常规子程序的写法一样，只是对于类属子程序，我们必须同时分别给出其规范式说明和体。

类属过程不能被直接调用，但可通过被称作类属例化的机制来说明实际可被调用的过程。譬如，我们可以写：

```
procedure SWAP is new EXCHANGE(REAL);
```

这是一个例化说明，它表明我们从样板过程 EXCHANGE 生成了实际过程 SWAP，实际的类属

参数以通常的方式在参数表中给出。对于形参 ITEM, 这时的实参即为 REAL 类型。另外, 我们也可以采用接名方式:

```
procedure SWAP is new EXCHANGE(ITEM=>REAL);
```

现在我们生成了针对 REAL 类型操作的 SWAP 过程, 因而也就能以通常的方式去调用它。此外, 我们可再作例化说明如下:

```
procedure SWAP is new EXCHANGE(INTEGER);
```

```
procedure SWAP is new EXCHANGE(DATE);
```

在此, 过程 SWAP 已经重载, 但重载的过程 SWAP 能按其参数的类型来相互区分。

从表面上看, 类属机制似乎只是一种文本替代, 而且在某些简单情况下其表现也大体如此, 但它不是! 重要的区别在于类属体内标识符的含义, 它们既不是参数也不局限于类属体。这样的标识符在类属体的说明中有恰当的含义, 而在类属例化时则将失去其含义。如果是文本替代, 那么这样的标识符当然在例化处仍具有其不变的含义, 这样可能会导出意外的结果。

同类属子程序的情形一样, 我们也可以有类属程序包。虽然 8.1 节中的 STACK 程序包的逻辑结构也适用于其它类型值的操作, 但它仅对 INTEGER 类型值加以操作。现在, 我们将其类属参量化, 另外再引进 MAX 参数, 以免栈的大小局限于 100。如:

```
generic
    MAX: POSITIVE;
    type ITEM is private;
package STACK is
    procedure PUSH(X:ITEM);
    function POP return ITEM;
end STACK;
package body STACK is
    S:array (1..MAX) of ITEM;
    TOP:INTEGER range 0..MAX;
    --其它细节如前, 仅将 INTEGER 更换为 ITEM
end STACK;
```

接下来便可通过如下方式将该类属程序包进行例化, 生成一特定类型和指定大小的栈, 并使用之:

```
declare
    package MY_STACK is new STACK(100,REAL);
    use MY_STACK;
begin
    ...
    PUSH(X);
    ...
    Y:=POP;
```

```
end;
```

例化得到的程序包 MY_STACK, 具有和常规方式直接写出来的程序包同样的特性。use 子句允许我们直接调用 PUSH 和 POP。如果又进行了例化,

```
package ANOTHER_STACK is new STACK(50,INTEGER);  
use ANOTHER_STACK;
```

那么 PUSH 以及 POP 便重载了, 但它们能按操作时参数的类型相互区分。当然, 如果 ANOTHER_STACK 也以 REAL 作为实际类属参数来进行例化说明, 那么无论是否写上 use 子句, 我们都不得不以加点引用方式来区分重载的 PUSH 和 POP。

类属程序单位和类属实例都可以是库程序单位。如果类属程序包 STACK 已加入了程序库, 那么它的实例便可自身单独地进行编译。例如:

```
with STACK;  
package BOOLEAN_STACK is new STACK(200,BOOLEAN);
```

如10.2节中那样, 如果把异常 ERROR 说明写进程序包, 那么类属程序包的定义即为:

```
generic  
    MAX,POSITIVE;  
    type ITEM is private;  
package STACK is  
    ERROR;exception;  
    procedure PUSH(X,ITEM);  
    function POP return ITEM;  
end STACK;
```

然后, 每次例化都会引出不同的异常, 又因为异常不能重载, 所以我们必须使用加点引用方式来区分它们。

当然, 我们也可使异常对于类属程序包全局化, 从而使异常 ERROR 对所有例化都有效。譬如, 将异常和类属程序包一起说明在某程序包内:

```
package ALL_STACKS is  
    ERROR;exception;  
    generic  
        MAX,POSITIVE;  
        type ITEM is private;  
    package STACK is  
        procedure PUSH(X,ITEM);  
        function POP return ITEM;  
    end STACK;  
end ALL_STACKS;
```

```

package body ALL_STACKS is
  package body STACK is
    ...
  end STACK;
end ALL_STACKS;

```

这是一种使某标识符对于类属程序单位全局化的方法。ERROR 的含义在类属定义前便固定了下来，因而在类属例化时其义不变。

前面的例子说明了类型和整型量作为类属形参的情况。实际上，类属形参可以是任何适用于子程序的参数形式，它也能是类型和子程序。在下面的章节里，我们将会看到如何表示类属的形式类型和形式子程序。这里，我们先研究那些也适用于常规子程序的参数形式。

对于那些熟知的适用于常规子程序的参数，类属中它们只能是 in 型或 in_out 型，而不能是 out 型。也正如常规子程序的情形，in 可以缺省。如上面例子中的 MAX。

in 型的类属参数在类属中起到一个常量的作用，其具体值由相应的实参提供。同常规子程序的参数形式一样，in 型的类属参数也允许有默认表达式，而且如果例化说明没有提供相应的实参，那么类属参数将取其默认表达式的值。显然，in 型类属参数不能为受限类型，这是因 a 能给受限类型赋值，而给 in 型参数提供值是作为赋值来处理的。注意：这与 in 型子程序参数的情形有所不同，子程序参数允许为受限类型。

in_out 型类属参数犹如相应实参的换名变量。因此，例化时的实参必须是一个变量名，并采用 8.5 节描述的换名规则，在例化时对参数进行标识。所以，实参的任何限定都适用于形参，而形参类型标志所蕴含的任何限定被完全忽略。另外，如果分量的存在依赖于判别式的值，那么实参不能是非受限判别式记录的分量。如 11.2 节的 M 为 MUTANT 类型，M.CHILDREN 便不能作为类属实参，因为如果 M.SEX 值不为 FEMALE，则该分量即不存在。然而，M.BIRTH 却可用作实参，因为它的存在不依赖 M.SEX 值。

类属体内，可以相当自由地运用形式类属参数，但有一条限制：许多地方，如 case 语句或变体的选择项、整型定义的范围、浮点类型定义的数字位数等等，表达式都必须静态给出，这些地方都不能用类属形式参数，因为含类属参数的表达式不是静态的。

最后介绍类属的嵌套。下面的类属过程用来循环交换三个值，而它又引用了 EXCHANGE 类属过程。

```

generic
  type THING is private;
procedure CAB(A,B,C;in out THING);
procedure CAB (A,B,C;in out THING) is
  procedure SWAP is new EXCHANGE(ITEM=>THING);
begin
  SWAP(A,B);
  SWAP(A,C);
end CAB;

```

虽然类属允许嵌套,但不能递归。

练习 13.1

1. 在11.1节 STACKS 程序包的基础上写一个类属程序包说明,以便用来说明任意类型的栈。试用接名方式说明一个长度为30,类型为 BOOLEAN 的栈 S。
2. 写一个包含 SWAP 与 CAB 两者的类属程序包。

13.2 类属类型参数

上节中已经提到类型作为类属参数,其形参如下给出:

```
type T is private;
```

这里 T 为形式类型名,而且我们可以假定在类属子程序或类属程序包内,为 T 类型定义了赋值与等价操作。(除非就象我们马上将看到的那样特别提供其它参数,这里不作其它假设。类属程序单位内,T 犹如一个私有类型,这也解释了形参的表示法。)因此,相应的实参必须提供赋值和等价,并且可以是除受限类型之外的任何类型。

此外,形式类属类型参数的给出形式可以是:

```
type T is limited private;
```

这时,T 为受限形式类型,赋值与等价操作不适用了,但相应的实参可以是任何类型。

以上两种形式都可以有判别式:

```
type T(...) is private;
```

而且实参也必须有同样类型的判别式。形式类型不必有默认的判别式表达式,但实际类型可以有默认的判别式表达式。

形参也可以是如下形式之一:

```
type T is (<>);  
type T is range <>;  
type T is digits <>;  
type T is delta <>;
```

对于第一种情况,实参必须为离散类型——枚举类型或整型。对于其它情况,实参必须分别为整型、浮点类型或定点类型。类属程序单位内,适当的预定义操作和属性仍有效。

考虑下面的简单例子:

```
generic  
type T is (<>);
```

```
        return T'SUCC(X);
    end if;
end NEXT;
```

形参 T 要求对应的实参必须为离散类型。因为所有的离散类型都具有属性 FIRST, LAST 和 SUCC, 因此我们能在类属体内利用实参能够提供的这些属性。

现在说明:

```
function TOMORROW is new NEXT(DAY);
```

然后必有 $\text{TOMORROW}(\text{SUN}) = \text{MON}$.

实在类属参数也能是子类型, 但不允许有显式的限定。(也就是说, 实参必须仅是类型标志, 而不能是子类型表示。) 这时, 形式类属参数也就代表相应的子类型。例如:

```
function NEXT_WORK_DAY is new NEXT(WEEKDAY);
```

结果 $\text{NEXT_WORK_DAY}(\text{FRI}) = \text{MON}$, 这时 LASTN 属性适用于子类型了, DAY'LAST 是 SUN, 而 WEEKDAY'LAST 则是 FRI。

这里的实参也可以是整型, 如有:

```
subtype DIGIT is INTEGER range 0..9;
function NEXT_DIGIT is new NEXT(DIGIT);
```

那么 $\text{NEXT_DIGIT}(9) = 0$.

现在改造 9.1 节的 COMPLEX_NUMBERS 程序包, 将其写成类属, 使类型 COMPLEX 所基于的特殊浮点类型可以是一个参数。以如下形式给出:

```
generic
    type REAL is digits <>;
package COMPLEX_NUMBERS is
    type COMPLEX is private;
    -- 如前
    I:constant COMPLEX:=(0.0,1.0);
end;
```

这里我们使用了直接量 0.0 与 1.0, 因为它们具有一般的实数类型, 因而能够转换成任何实参传过来的类型。这个程序包可如此例化:

```
package MY_COMPLEX_NUMBERS is
    new COMPLEX_NUMBERS(MY_REAL);
```

另外, 形式类属参数也可为数组类型。这时, 实参也必须为数组类型, 而且要有相同的分量类型和限定以及相同的维数和下标子类型。形参与实参是否为受限数组也必须一致, 要么都是, 要么都不是。如果它们同为受限数组, 那么其对应下标的下标范围也必须相同。

一个类属形参可能依赖于其前面的类型形参。这在数组类型作参数的情况下是经常有的。例如 7.1 节中的简单函数 SUM, 它实现整型下标实型数组的元素相加。我们将其类属参量化, 来实现任何下标类型的任何浮点数组的元素相加。如:



```
generic
    type INDEX is (<>);
    type FLOATING is digits <>;
    type VEC is array (INDEX range <>) of FLOATING;
function SUM (A:VEC) return FLOATING;
function SUM (A:VEC) return FLOATING is
    RESULT:FLOATING:=0.0;
begin
    for I in A'RANGE loop
        RESULT:=RESULT+A(I);
    end loop;
    return RESULT;
end SUM;
```

这里虽然 INDEX 为形参,但它没有在类属体内显式地出现,然而,由于循环参量 I 具有 INDEX 类型,所以 INDEX 被隐式地使用了。

我们作例化说明:

```
function SUM_VECTOR is new SUM(INTEGER,REAL,VECTOR);
```

这给出了如7.1节 SUM 的函数。

实在数组与形式数组的匹配,在形式数组中的任何形式类型都替代为对应实在类型之后才进行,如先要进行下标子类型匹配。如果有实参的说明:

```
type VECTOR is array (POSITIVE range (<>)) of REAL;
```

那么便要用 POSITIVE(或任何相等的子类型)作为实参来替代 INDEX。

最后一个可能的形式类型参数即存取类型,其形式可以是:

```
type A is access T;
```

这里的 T,可以是但也不必是前面的形参,但对应于 A 的实参必须也存取 T 类型对象,而且有关存取类型的限定也必须相同。

请注意:不存在形式记录类型的概念。因为记录的内部结构较随意,因而匹配的可能性极小。

最后我们再回到集合的问题上。在6.4节中,我们已经知道如何用布尔数组来表示集合。练习7.1(4),7.2(3)和7.2(4)也示范了怎样编写适当的函数对 COLOUR 类型的集合进行操作。在此基础上,类属机制允许我们编写程序包,来对任意类型的集合进行操作。

考虑:

```
generic
    type BASE is (<>);
package SET_OF is
    type SET is priv;
    type LIST is array (POSITIVE range <>) of BASE;
```

```

EMPTY,FULL;constant SET;
function MAKE_SET(X:LIST) return SET;
function MAKE_SET(X:BASE) return SET;
function DECOMPOSE(X:SET) return LIST;
function "+"(X,Y:SET) return SET;      --并
function "*" (X,Y:SET) return SET;     --交
function "-" (X,Y:SET) return SET;     --差
function "<" (X,BASE;Y,SET) return BOOLEAN;    --蕴含
function "<=" (X,Y:SET) return BOOLEAN;      --包含
function SIZE(X:SET) return NATURAL;       --元素个数
private
type SET is array (BASE) of BOOLEAN;
EMPTY;constant SET:=(SET'RANGE=>FALSE);
FULL;constant SET:=(SET'RANGE=>TRUE);
end;

```



其唯一的类属参数是基类型，当然，它必须是离散的。类型 SET 被定为私有的，目的是为了防止随意的布尔操作。类型 LIST 的聚集用来表示直接量集合。常量 EMPTY 和 FULL 分别表示集合的空和满。函数 MAKE_SET 用来从一个基值表或是一个单基值创建集合。DECOMPOSE 则使集合变回为表。

操作符 +、* 和 - 分别表示并、交、差，它们比基本的 or, and 和 xor 用起来更自然。运算符 < 用来测试一个基值是否在某集合中，而 <= 则用来测试一集合是否为另一集合的子集。最末的函数 SIZE 用来返回某特殊集合中出现的基值的个数（即集合的大小）。

在私有部分，类型 SET 被说明为以基类型为下标的布尔数组。常量 EMPTY 和 FULL 则分别说明为元素全为 FALSE 和 TRUE 的数组。（程序包体的构造留作练习。）

回顾 6.4 节，我们能例化一个程序包使其工作在类型 PRIMARY 上。如：

```

package PRIMARY_SETS is new SET_OF(PRIMARY);
use PRIMARY_SETS;

```

为对照起见，我们可以写：

```

subtype COLOUR is SET;
WHITE;COLOUR renames EMPTY;
BLACK;COLOUR renames FULL;

```

等等。

有关类型匹配的一般规则现在已交待清楚。形式类型表示一类具有共同特性的类型，而这些特性可以在类属程序单位中假定。相应的实在类型则必须提供这些特性。设定匹配规则是为了由引用参数作保证，而不用去考虑类属体细节。因此，使用类属程序单位时，无须为了调试而去考察类属体。这个由参数保证的匹配概念被称作收缩模型。

不幸的是，在非受限类型情况下，对收缩模型存在一个严重的违背。如果我们有基本形式类型：

```
type T is private;
```

若类属程序单位没有在要求数组为受限的地方使用 T(最明显的例子即没有用 T 类型说明对象), 则可用非受限数组类型, 如 VECTOR 类型来匹配 T。同样, 如果没有在要求为受限的地方使用 T, 那么 T 的实在类型可为非受限可判别类型。可见, 我们无法说明一个 T 类型的非受限对象, 既使实在类型具有默认的判别式也如此, 因为这样的默认在类属体内无效。

属性 CONSTRAINED 适用于形式类型 T, 并给出布尔值以表明实在参数是否为受限类型。考虑 11.2 节中的类型, 如果实参是 MAN, 则 T'CONSTRAINED 为 TRUE; 如果实参是 PERSON 或 MUTANT, 则 T'CONSTRAINED 为 FALSE。在最后一种情况下, 默认限定 NEUTER 并不起作用。请记住: 实参必须是一个类型标志, 而不能是子类型表示。

最后注意另一个细小的限制: case 语句中的表达式类型和变体中的判别式类型都不能是类属形式类型。

练习 12.2

1. 例化 NEXT, 给出一个类同 not 的函数。
2. 重写程序包 RATIONAL_NUMBERS 的规范式说明, 使其成为以整型作危数的类属程序包。参见练习 8.1(2)。
3. 重写练习 7.1(3) 的函数 OUTER, 使它成为有适当参数的类属函数, 并例化给出原先的函数。
4. 写出程序包 SET_OF 的包体。

13.3 类属子程序参数

Ada 类属参数也能是子程序。有些语言, 如 Algol 和 Pascal, 其子程序的参数可以是子程序, 这在数学计算上是很有用的, 如求积分。而在 Ada 中, 子程序只能是类属程序单位的参数, 所以, 可将 Ada 的类属机制用于这些应用。譬如, 我们可以写类属函数:

```
generic
  with function F(X;REAL) return REAL;
function INTEGRATE(A,B;REAL) return REAL;
```

用于计算积分 $\int_a^b f(x)dx$ 。

如果求某特殊函数的积分, 则必须以该特殊函数作为类属实参来例化 INTEGRATE。假设求积分: $\int_0^{\pi} e^{\sin t} dt$

我们须作如下说明:

```
function G (T;REAL) return REAL is
begin
  return EXP(T) * SIN(T);
end;
function INTEGRATE_G is new INTEGRATE(G);
```

然后, 结果由下式给出:

```
INTEGRATE_G(0.0,P)
```

类属的形式子程序参数犹如一个冠以 with 的子程序说明。(with 在这里被用来避免语法二义性而无其它意义。) 形式子程序与实在子程序的匹配就是将形式子程序换名为实在子程

序。因此,它们的有关参数、结果等的匹配规则与8.5节中描述的换名规则相同。特别是它们所带参数的限定即为实在子程序中的限定,而形式子程序中所蕴含的任何限定都被忽略。也正如换名,无参的形式函数能用结果类型的枚举直接量匹配。

另外,形式子程序的规范式说明中可用到前面的形式类型。因此,我们可以如下扩充积分函数,使其能对任何浮点类型的函数求积分:

```
generic
    type FLOATING is digits <>;
    with function F(X:FLOATING) return FLOATING;
    function INTEGRATE(A,B:FLOATING) return FLOATING;
```

然后有:

```
function INTEGRATE_G is new INTEGRATE(REAL,G);
```

实际上函数 INTEGRATE 还可有其它的参数,用来表示精确度要求等等。

有时,我们也可借用形式子程序来提供类型参数的某些特性。对上节的类属函数 SUM,借用形式子程序把运算符也作为类属参数,可使之更加参量化。如:

```
generic
    type INDEX is (<>);
    type ITEM is private;
    type VEC is array (INDEX range <>) of ITEM;
    with function "+"(X,Y:ITEM) return ITEM;
    function APPLY(A:VEC) return ITEM;
    function APPLY(A:VEC) return ITEM is
        RESULT:ITEM:=A(A'FIRST);
begin
    for I in INDEX'SUCC(A'FIRST)..A'LAST loop
        RESULT:=RESULT+A(I);
    end loop;
    return RESULT;
end APPLY;
```

现在,运算符“+”已成了参数,而且 ITEM 也为类属的私有类型而不固定为长浮点型了。这意味着,我们可以用该类属函数实现任何类型的任何二操作。但是,我们不再有0这个值,因而须用数组 A 的第一个分量来对 RESULT 初始化,然后迭代其它分量。这样做时,注意不能写:

```
for I in A'FIRST+1..A'LAST loop
```

因为运算符“+”现已不适用于类型 INDEX。不过,我们可利用 INDEX 为离散类型的属性特征 INDEX'SUCC。

对于7.1节的函数 SUM,现在可这样给出:

```
function SUM is new APPLY(INTEGER,REAL,VECTOR,"+");
```

同样，也可以有：

```
function PROD is new APPLY(INTEGER,REAL,VECTOR,"*");
```

类属子程序参数（如同类属对象参数）也能有默认值，它们在类属形式部分给出，而且有两种给出形式。上例中，如果在形式子程序参数说明时写成：

```
with function "+"(X,Y;ITEM) return ITEM is <>;
```

这表明，例化时若只有唯一的实在子程序与“+”有相同标识符和相匹配的规范式说明，那么只能用它来匹配，因而可以省去相应的例化实参。这样更改 APPLY 后，我们可以在例化给出 SUM 时省去最末的参数。

默认值的另一形式即给默认表达式一个显式名字。这时，默认的通常规则是：如果例化选择默认，那么默认名字即被赋值，但默认表达式中标识符的结合在说明处就要成立。例如：

```
with function "+"(X,Y;ITEM) return ITEM is PLUS;
```

永远无效，因为 PLUS 的规范式说明必须与“+”的相匹配，且参数 ITEM 到例化时才知道。这里仅存的默认有效的可能性是形式子程序不带依赖形式类型的参数，或者默认子程序仅是一个属性。譬如，我们可以有：

```
with function NEXT(X;T) is T'SUCC;
```

混合命名和位置记法的规则，既适用于子程序调用，也适用于类属例化。因此，如果一个参数被省略，那么，随后的参数则必须以按名方式给出。另外，当例化以如同子程序调用的形式实现时，类属程序单位不需要有参数（括号被省去）。

练习 13.3

1. 给出求等式 $f(x)=0$ 的根的函数：

```
generic
```

```
    with function F(X:REAL) return REAL;  
function SOLVE return REAL;
```

并求 $e^x+x=7$ 的根。

2. 例化 APPLY，给出函数“and”，求布尔数组所有分量的与操作。
3. 重写 APPLY，使得空数组作危。数不引发异常，并以这个新样板重做前面的练习。
4. 编写类属函数 EQUALS，定义受限私有类型一维数组等式。参见练习 9.2(3)，例化给出适用于类型 STACK_ARRAY 的函数“=”。
5. 在 11.3 节中的 SORT 过程的基础上建立一个类属排序过程，其参数提供被排序项的类型、相应的数组类型、下标类型以及比较规则。

要点 13

- 类属机制不是文本替代。
- 类属的 in out 型对象其参数以换名来匹配。
- 类属的 in 型对象其参数不象子程序的参数那样总是被复制。
- 类属子程序不能重载，但例化可以重载。
- 类属形式参数可依赖其前面的类型参数。

第14章 任 务

任务是我们将要引入的一个主要内容。之所以留在后面来讨论它，并不是因为它不重要，而是因为需要异常这一概念作为它的基础。

14.1 并行

在此之前我们仅讨论了顺序程序。在顺序程序中，语句是按序执行的。然而，在许多实际应用中，需要把程序写成若干个并行的活动。

在 Ada 中，并行活动是用任务这一概念来描述的。简单地看，任务在词法形式上很象程序包。它也是由规范式说明和体(又称任务体)两部分组成。规范式说明向其它任务提供接口，而任务体描述了任务的动态行为。

```
task  T  is      ——规范式说明
...
end  T;
task  body  T  is    ——体
...
end  T;
```

在有些情况下，任务不向其它任务提供接口，此时，规范式说明简化为

```
task  T;
```

为了说明并行，我们不妨以一个家庭为准备午餐而去商店购置食物的活动为例。假定这顿午餐需要肉、色拉和酒，并且可由称之为 BUY_MEAT, BUY_SALAD 和 BUY_WINE 的过程来分别购置这些食物。整个购买活动可以表示成：

```
procedure  SHOPPING  is
begin
BUY_MEAT;
BUY_SALAD;
BUY_WINE;
```

然而，上述表示的过程相当于这家人按顺序的方式购买每样食物。如果让家庭成员分别去购买食物显然要有效的多。譬如，母亲买肉，孩子买色拉而父亲买酒，然后他们再在停车场碰头。这种并行的活动，可以表示为

```
procedure  SHOPPING  is
task  GET_SALAD;
task  body  GET_SALAD  is
```

```
begin
  BUY_SALAD;
end GET_SALAD;
task GET_WINE;
task body GET_WINE is
begin
  BUY_WINE;
end GET_WINE;
begin
  BUY_MEAT;
end SHOPPING;
```

在解的表示中,母亲的活动被看作为主进程,直接在过程 SHOPPING 中调用 BUY_MEAT。而孩子和父亲则被视为辅助进程,分别执行被说明为局部任务的 GET_SALAD 和 GET_WINE,而它们各自调用 BUY_SALAD 和 BUY_WINE 过程。

从这个例子中,我们展示了任务的说明、启动和终止。任务如同程序包一样,也是一程序单位且以程序包同样的方式加以说明,其出现在一子程序、程序包、分程序乃至另一任务体内。任务的规范式说明也可以在程序包的规范式说明中出现,在这种情况下,任务体必须在相应的程序包体中加以说明。然而,一个任务的规范说明不能在另一个任务的规范式说明中出现,只能在该任务体中出现。

任务的激活是自动进行的,在上述的例子中,当主进程(或称根进程)到达跟在任务说明后的 begin 时,局部任务便被激活。

上述的每个任务只有当它到达自己的任务体最后一个 end 时,才被终止。因此有,当任务 GET_SALAD 调用了过程 BUY_SALAD 以后,它就被终止。

在子程序、分程序说明部定义的任务或在任务体中定义的任务从属于该程序单位。只有当所有从属于某个程序单位的任务均终止之后,该程序单位才终止。这就是所谓的“结束规则”。结束规则确保了对象在程序单位中可被说明,而且当存在一个任务试图访问这些对象时,这些作为对象的局部任务是可见的。(注意:任务不从属于程序包——我们以后再对这一点加以讨论。)

在 Ada 中,任务对应于进程。有一点是重要的,即是说主程序被认为由主任务(或称主进程)来调用。当我们把 SHOPPING 看作主任务时,我们便可以追踪其活动情况。首先,任务 GET_SALAD 和 GET_WINE 被说明,当主任务到达这两个任务后的 begin 时,这两个从属任务并行被激活,从属任务调用它们各自的过程后终止。除非所有的从属任务先于主任务终止,否则主任务一直处于等待。这正如,母亲在买回肉以后,等待着孩子和父亲带回他们各自购得的食品。我们说,当程序单位到达其最后一个 end 时,该程序单位终止。但在通常情况下,总是处于两种终止情形,一种是程序单位中所有从属任务终止后,该程序单位接着也终止。另一种情况是程序单位没有从属任务或者与从属任务同时终止。

14.2 汇合

在 SHOPPING 的例子中各个任务一旦被激活后,除了代表程序单位的主任务需要等待子

任务终止外,各个子任务间并不相互作用。一般而言,在任务的活动期间内,任务之间总是要相互作用的。在 Ada 中借助称之为“汇合”的机制来实现任务间的相互作用。这如同人们在日常生活中,往往需要会面,两人会面一起处理完事务后,再各自分头继续自己的活动。

汇合由一个任务调用在另一个任务体中的入口(entry)而引起。入口(entry)在任务的规范式说明中被说明,正如过程在程序包规范式说明中被说明一样。

```
task T is
entry E(...);
end;
```

和过程一样,入口具有 in,out 和 inout 使用模式的参数。然而任务不象函数那样具有返回结果,入口象过程那样被调用。

```
T.E(...);
```

由于任务名不可象函数名那样直接作为语句成份使用,故在任务外调用入口时,必须加点“.”来表示对任务入口的调用。局部任务可调用其父辈任务的入口——通常须满足作用域和可见性规则。

在汇合期间所执行的语句由任务体中相应的 accept 语句给予指明。accept 语句形式为:

```
accept E(...) do
--语句序列
end E;
```

此处入口 E 的形参与入口被说明时的形参一样。end 后跟有入口名。与过程的一个重要区别是 accept 语句体只能是一系列语句,任何局部说明或异常处理必须写成局部部分程序。

入口调用与过程调用两者间最重要的区别在于:在过程的情况下,任务调用过程则过程体立即执行。而在入口的情况下,任务调用入口只是入口所属的任务体中相应的 accept 语句执行,并且只有当任务调用入口与具有该入口的任务到达 accept 语句处恰好同时发生时,accept 语句方能被执行。显然,这种巧合往往不会发生。总是要么调用入口先发生,要么被调任务先到达 accept 语句,此时先到的任务被延迟到另一个任务到达相应的语句。在被调任务执行 accept 语句系列时,发出调用的任务始终被延迟。这种相互作用称之为汇合。当被调任务执行到 accept 语句的 end 处以后,会合完成,两任务又各自前进。

我们可以为任务 GET_SALAD 设置两个入口来重新加工 SHOPPING 例子。一个入口用来付钱给孩子供他们买色拉,另一个入口从孩子那里收集色拉。亦可同样设计任务 GET_WINE。

我们此时用函数来取代过程 BUY_SALAD,BUY_WINE,这些函数将钱作为一个参数且返回相应的成份。我们的 SHOPPING 现在变为:

```
procedure SHOPPING is
task GET_SALAD is
    entry PAY(M,in MONEY);
    entry COLLECT(S,out SALAD);
end GET_SALAD;
```

```
task body GET_SALAD is
    CASH;MONEY;
    FOOD;SALAD;
begin
    accept PAY(M;in MONEY) do
        CASH:=M;
    end PAY;

    FOOD:=BUY_SALAD(CASH);

    accept COLLECT(S;out SALAD) do
        S:=FOOD;
    end COLLECT;
end GET_SALAD;
--GET_WINE 类似
begin
    GET_SALAD.PAY(50);
    GET_WINE.PAY(100);
    MM:=BUY_MEAT(200);
    GET_SALAD.COLLECT(SS);
    GET_WINE.COLLECT(WW);
end;
```

在 SHOPPING 这个例子中，最终结果赋值于变量 MM,SS 和 WW 中。

关于变量 MM,SS 和 WW 的说明留给读者自己考虑。

现在让我们来看一下此程序的逻辑行为。一旦任务 GET_SALAD 和 GET_WINE 被启动后，它们立刻会遇到 accept 语句，此时这两个任务便处于等待主任务调用它们的入口 PAY。主任务在调用 BUY_MEAT 以后，调用 COLLECT 入口收集相应的食品，由于首先调用 GET_SALAD 的 COLLECT 入口，因此，在母亲从孩子处收集到色拉前，是不能从父亲处收集到酒的。

作为更为抽象的例子，让我们用一个任务充当信号缓冲，让一个或多个任务产生信号送入该缓冲而让一个或多个任务消费缓冲中的信号。假定充当缓冲的中间任务一次只能保存一个信号单位。

```
task BUFFERING is
    entry PUT(X;in ITEM);
    entry GET(X;out ITEM);
end;
task body BUFFERING is
    V;ITEM;
begin
```

```
loop
    accept PUT(X,in ITEM) do
        V:=X;
    end PUT;
    accept GET(X,out ITEM) do
        X:=V;
    end GET;
    end loop;
end BUFFERING;
```

其它任务则可以调用

```
BUFFERING.PUT(...);
BUFFERING.GET(...);
```

向缓冲中送信号或从缓冲中取出信号。用变量 V 作为信号的中间存贮单元。任务体是一个无限循环，在循环体中含有一个带有 PUT 入口的 accept 语句和一个带有 GET 入口的 accept 语句。该任务随机地每次接受一个或对 PUT 或对 GET 的调用，以填满变量 V 或取空变量 V。

若干个不同的任务均可能同时调用 PUT 和 GET，因此必须将它们排队。每个入口具有一个等待调用入口的任务队列——该队列遵循先进先出的原则。当然，在某一特定的瞬间等待队列也可能为空。在入口 E 的等待列中的任务数由 E 计数器指明，该计数器隐含在具有 E 入口的任务体内。

一入口可有多个对应的 accept 语句（通常只有一个）。每一 accept 语句执行，便从等待队列移出一任务。

需指出的是，汇合为非对称命名的，也就是说，发出调用的任务必须指出被调任务的名，反之则不然。此外，若干任务可以调用一个入口且被排在等待队列中，但每次一个任务只可能被排在一个队列中。

入口可用子程序或是其它实体来“重载”，无论是用其它实体还是子程序重载均遵循同样的规则。入口可被命名为过程

```
Procedure WRITE(X,in ITEM) renames BUFERING.PUT;
```

为了避免过多地使用加点表示法可采用这种重命名法。无论是否重命名或是具有参数，入口在形式上好象子程序。入口可以没有参数，例如：

```
entry SIGNAL;
```

其调用形式为

```
T.SIGNAL;
```

accept 语句也未必一定有语体，如：

```
accept SIGNAL;
```

此时，调用仅起同步作用而并不传递信息。然而，一入口在无参数的情况下亦可具有一个带有语句体的 accept 语句，反之亦然。

```
accept SIGNAL do
```

```
FIRE;  
end;
```

此时,任务调用 SIGNAL 只是让 FIRE 的调用完成后才让调用任务继续。尽管我们清楚参数值并不使用,我们仍可有

```
accept PUT(X;ITEM);
```

在 accept 语句体中的语句无特殊限制,它们可含入口调用、子程序调用、分程序以至 accept 语句(但不能对同一入口)。另外,虽然 accept 语句可出现在分程序或其它 accept 语句中,此时它可不出现在子程序体中但必须在任务体的语句中出现,accept 语句执行到最后一个 end 时,相当于执行一返回语句,汇合此时也就终止。任务可以调用其自身的一个入口,显然这会迅速导致死锁,这似乎荒唐,但程序设计语言允许许多似乎很荒唐的事,比如无限循环等。

14.3 分时和调度

正如我们已经看到的那样,一 Ada 程序可能会含有若干个任务。在概念上,最好认为这些任务每个都具有自己的处理机,无需等待其它事情的发生而直接在处理机上运行。在实际实现上,往往只有一个物理处理机。不可能为每个任务都指派一个独享处理机,因此需要根据某种调度算法给任务指派能使其执行的逻辑处理机。可采用很多种方法进行这种指派。

一个最简单的方法是,采用划分时间片的方法。这一方法的思想为,按某一固定的时间间隔如 10ms,把处理机轮流分配给任务。如果被轮到的任务因某种原因不能利用时间片时(也许该任务正处在等待与其合作任务汇合),则调度程序将把时间片分配给下个任务。划分时间片方法略显简单化了,因为它平均地对待所有任务。常有的情况为,某些任务往往比另外一些任务更为迫切,但面临平均划分时间片而不能及时处理,因此在时间片上的这种“平等”稍显不经济。人们引入了使任务具有优先权的想法。这种思想被普通的调度系统所采用,在该类调度系统中任务具有明确的优先权,处理机总是被分配给拥有最高优先权的任务以使该任务及时地运行。也可以把时间片划分法和优先权调度法结合起来使用。一系统可允许若干个任务具有相同的优先权且在这些任务中划分时间片。

Ada 允许采用多种调度策略。如果一执行具有优先权概念,则任务的优先权由出现在任务规范说明中的 pragma 指出,例如:

```
task BUFFERING is  
pragma PRIORITY(7);  
entry PUT...  
...  
end;
```

正如我们已经看到的那样,主程序亦可看作为任务,故 pragma 进入该主程序最外层的说明部分。

优先权必须是类型为 PRIORITY 的静态表达式,而 PRIORITY 是 INTEGER 的子类型,子类型 PRIORITY 的实际范围取决于处理。需强调的是:任务的优先权是静态的,因此指明的优先权不能随程序的执行而改变。优先数大,表示优先级高。一方面,若干个任务能够具有同一优先级,而另一方面,任务完全不需具有明确的优先级。Ada 任务在调度上,其优先权的含义来自

于 LRM, 我们给出其规则如下：“如果两个具有不同优先级的任务都有执行资格并且明显地由同样物理处理器和同样的其它处理资源来执行时, 不能出现低优先级在执行的情况。”

此规则没有谈及任务没有定义优先级的情况, 也没有说到具有相同优先级任务的情况, 似乎处于这两种情况可自由决定。此外, 没有采取什么措施防止由 PRIORITY'FIRST = PRIORITY'LAST 导致的执行, 对于这种情形所有的任务时间片相同且优先权概念不存在。

因为含有两个任务, 故汇合的情况是复杂的。如两个任务具有明确的优先权, 则汇合以高优先权执行。如果仅一个任务具有明确的优先权, 则汇合至少以该优先权执行。如果两任务均没有定义优先权, 则汇合的优先权也不定。当然, 如果 accept 语句还含有入口调用, 或还含有 accept 语句则根据情况再次使用优先规则。

汇合规则保证了一个高优先级任务不致于仅仅因为它从事与低优先级任务汇合而被挂起。换言之, 在入口队列中接受任务的次序始终是先进先出而不是由优先级起作用。如果一高优先级任务希望防止在同一入口队列中被低优先级任务挂起, 可使用“越时”或条件调用, 对此我们将在下一节论述。

优先权的使用需小心, 它被用来作为调节迫切度的方法而不是用于同步。不能认为由于任务 A 比任务 B 优先权高则一定有任务 A 的执行阻碍任务 B 的执行。由于不同的实时需求, 可能设有多个处理器或以后的程序维护, 这些均可能导致优先级的变化。借助汇合便可实现同步, 除非要精确地调整响应性, 否则可忽略优先级。

因为各种原因, 任务可被挂起, 比如它可能等待合作任务以实现汇合或相关任务业已终止, 任务也可能用执行 delay 语句来挂起。例如:

```
delay 3.0;
```

这条语句暂停任务(或主程序)执行至少3秒(我们必须说“至少”, 因为在期限满时, 未必有处理器能立即执行此任务)。保留字 delay 后的表达式为预定义定点类型 DURATION, 用它给出以秒为计数单位的暂停时限。

类型 DURATION 是定点型, 故时限相加可不忽略系统误差。当我们把两个定点型的数据形式加到一起时, 我们总能得到另一种形式的数据, 这种形式的数据不包括浮点。此外, 我们需要方便地来表示秒的小数形式, 为此, 采用实型比整型更令人满意。

采用适当的数据说明, 延时可以很容易地表示出, 因此

```
SECONDS:constant := 1.0;  
MINUTES:constant := 60.0;  
HOURS:constant := 3600.0;
```

我们便可写出例子

```
delay 2 * HOURS + 40 * MINUTES;
```

在这个例中, 表达式的类型不是普通的实型, 而是自动地转换为 DURATION 类。一个具有零或负变元的延迟语句无效。

虽然在处理上类型 DURATION 已被定义, 但我们只能保证让时限最大到至多一天(86400 秒)。多于一天的时限必须用循环来编程实现。DURATION 的最小值, 即 DURATION'SMALL 为不大于 20ms。

更为复杂的定时操作可用预定义的程序包 CALENDAR 来实现, 该程序包的规范式说明为:

```

type TIME is Private;
subtype YEAR_NUMBER is INTEGER range 1901..2090;
subtype MONTH_NUMBER is INTEGER range 1..12;
subtype DAY_NUMBER is INTEGER 1..31;
subtype DAY_DURATION is DURATION range 0.0..86_400.0;

function CLOCK return TIME;
function YEAR(DATE;TIME) return YEAR_NUMBER;
function MONTH(DATE;TIME) return MONTH_NUMBER;
function DAY(DATE;TIME) return DAY_NUMBER;
function SECONDS(DATE;TIME) return DAY_DURATION;

procedure SPLIT(DATE,in TIME;
                 YEAR;out YEAR_NUMBER;
                 MONTH;out MONTH_NUMBER;
                 DAY;out DAY_NUMBER;
                 SECONDS;out DAY_DURATION);

function TIME_OF(YEAR;YEAR_NUMBER;
                 MONTH;MONTH_NUMBER;
                 DAY;DAY_NUMBER;
                 SECONDS;DAY_DURATION:=0.0)
    return TIME;

function "+"(LEFT;TIME;RIGHT;DURATION) return TIME;
function "+"(LEFT;DURATION;RIGHT;TIME) return TIME;
function "-"(LEFT;TIME;RIGHT;DURATION) return TIME;
function "-"(LEFT;TIME;RIGHT;TIME) return DURATION;
function "<"(LEFT;RIGHT;TIME) return BOOLEAN;
function "<="(LEFT;TIME;RIGHT;TIME) return BOOLEAN;
function ">"(LEFT;TIME;RIGHT;TIME) return BOOLEAN;
function ">="(LEFT;TIME;RIGHT;TIME) return BOOLEAN;

TIME--ERROR;exception;
--能够由时间的"+"和"-"所引发

private
--依赖于执行
end CALENDAR;

```

私有类型 TIME 的值是时间和日期混合量,调用过程 SPLIT 可将其还原为年,月,日及由

午夜折算的时间。此外,函数 YEAR,MONTH,DAY 和 SECONDS 也可用来分别求得年,月,日和时间。反过来,函数 TIME_OF 能够用来根据上述四个分量(即 YEAR,MONTH 等)形成 TIME 的值,其中入参数以缺省为零的方式给出。YEAR_NUMBER,MONTH_NUMBER 和 DAY_NUMBER 为子类型,YEAR_NUMBER 的范围取闰年以使计算得到简化。在 TIME_OF 的参数满足约束但仍不能形成恰当的时时间 TIME_ERROR 异常将被引发。须要特别注意 TIME 和 DURATION 间的区别,TIME 是绝对的而 DURATION 是相对的。

调用函数 CLOCK 返回现行 TIME,返回的结果显然是一个混合量且该量能够借助分离函数提供现行时间和日期的各个分量。各“重载”的“+”,“-”和关系运算符允许我们以恰当的方式去加减,比较时间和日期。试图在年允许范围外或超越执行范围外建立时间均将引发异常 TIME_ERROR。对于参数名 LEFT 和 RIGHT 是在程序包 STANDARD 中预先定义了的。

现举例说明程序包 CALENDAR 的使用,假定我们希望以一定的时间周期(如5秒)调用过程 ACTION,可写为:

```
LOOP
    delay 5 * MINUTES;
    ACTION;
end loop;
```

然而,鉴于两方面原因,我们认为这种写法是不能令人满意的,首先,我们并没有把 ACTION 过程本身的执行时间和循环额外开销的时间计算在内;其次,延迟语句只能置最小的延迟,因此,不可避免地产生积累偏差。为此,可重写此例。

```
declare
    use CAENDAR;
    INTERVAL;constant DURATION:=5 * MINUTES;
    NEXT_TIME:TIME:=FIRST_TIME;
begin
    loop
        delay NEXT_TIME_CLOCK;
        ACTION;
        NEXT_TIME:=NEXT_TIME+INTERVAL;
    end loop;
end;
```

在这一写法中,NEXT_TIME 中含有 ACTION 再次调用的时间,其初值为 FIRST_TIME,每次加 INTERVAL 而被修改。延迟语句用来进行延迟的时间为 NEXT_TIME 与调用 CLOCK 所得的现行时间的差。这样做就不再有因 ACTION 本身和循环开销所导致的积累偏差。当然,也还会出现局部偏差,比如 ACTION 的一次特殊调用花了一段很长时间或其它任务暂时使用处理器。在此,对计时还有一个另外的条件必须满足,这就是间隔(INTERVAL)必须是样板数。

14.4 选择语句

选择语句允许一任务有选择的与若干个任务之一汇合。

为了说明选择语句的使用,现讨论对非控制存取的变量 V 的保护问题。我们用一个程序包及两个过程 READ 和 WRITE 对该变量进行读写。

```
package PROTECTED_VARIABLE is
    procedure READ(X,out ITEM);
    procedure WRITE(X,in ITEM);
end;
package body PROTECTED_VARIABLE is
    V:ITEM;
    procedure READ(X,out ITEM) is
        begin
            X:=V;
        end;
    procedure write(X,in ITEM) is
        begin
            V:=X;
        end;
    begin
        V:=初始值;
    end PROTECTED_VARIABLE;
```



然而,这种解决通常是不能令人满意的。必须保证首先做 WRITE 调用,否则初始值很难确定。为此,需要建立一个内部状态标志,当 READ 先被调用时由其引发异常处理,但这样做便使接口复杂化。原因在于系统中没有什么措施能防止 READ 和 WRITE 调用同时发生而导致此操作的相互干扰。

作为一个更为具体的例子,假定类型 ITEM 是一个指定飞机或船舶座标的记录。

```
TYPE ITEM is
    record
        X_COORD:REAL;
        Y_COORD:REAL;
    end record;
```

存在一任务 A 得到一对值,该任务调用 WRITE 将这对值存入 V 中,而有另一任务 B 随即调用 READ 读取这对值。现假定任务 A 在进入 WRITE 执行一半时,由于任务 B 随机调用 READ 而将 A 的执行中断,此时 B 得到的值可能是由新 X 座标和老 Y 座标组成,由这组矛盾的数据计算出的航向显然是不准确的。

读者也许会感到奇怪,为什么任务 A 的执行会被任务 B 随意地中断呢?原因在于:

在采用按时间片划分的单处理机系统中,可能 B 恰好在不合时宜的时刻被轮到执行了。此外,即便任务 B 具有比任务 A 更高的优先权,当 B 借助延时语句处于等待,允许 A 的执行过程中,正巧 B 所等待的延时结束,B 的再次执行正处 A 执行的中途这一错误时刻。在实际的实时场合下,随机情况允许发生在错误时刻!

正确的解决办法,是采用任务而不是程序包,采用入口调用而不是过程调用来对变量 V 进行读写保护。我们对此予以讨论:

```
task PROTECTED_VARIABLE is
    entry READ(X,out ITEM);
    entry WRITE(X,in ITEM);
end;
task body PROTECTED_VARIABLE is
    V,ITEM;
begin
    accept WRITE(X,in ITEM) do
        V:=X;
    end;
    loop
    select
        accept READ(X,out ITEM) do
            X:=V;
        end;
        or
        accept write(X,in ITEM) do
            V:=X;
        end;
    end select;
    end loop;
end PROTECTED_VARIABLE;
```

超星阅览器提醒您:
使用本资源
请尊重相关知识产权!

在任务体的起始处,设置一个具有 WRITE 入口的 accept 语句,保证首先接收对 WRITE 的调用,避免了在变量的值未确定之前就去读。当然,某任务可能在其它任务调用 WRITE 前发出对 READ 的调用,但这时,对 READ 的调用被排队,直至 WRITE 调用接受以后。

完成接收 WRITE 的调用之后,任务进入含有一选择语句的无限循环中。选择语句以关键字 select 开始,end select 结束。一选择语句含有两个或多个由 or 指明的分支。本例中的选择语句有两个分支,每个分支由一个 accept 语句组成。

对于本例的选择语句可能发生的各种情况为,或者 READ 调用发生,或者 WRITE 调用发生,或者它们都不发生。对这些情况依次为:

- 如果既无 READ 又无 WRITE 调用发生，则任务被暂时挂起直至发生某一调用，此时相应的接收语句(accept 语句)执行。

如果一个或多个READ调用被排队且无WRITE被排队，则队首READ被接收，反之亦然。

- 如果 READ 和 WRITE 均被排队，则任意选择一个接收。

select 语句的每次执行，其分支之一被选中，即或 READ 或 WRITE 被执行。我们可以把任务看作满足两消费者队伍不同服务请求的服务员。如果只有一个队列有顾客则服务员为该队顾客服务。如果两个队列均无顾客则服务员等待为无论是哪个队列到来的第一个顾客服务。如果两个队列都有顾客，服务员每次任选一顾客为其服务。

在进入任务 PROTECTED _ VARIABLE 以后，根据对该任务的调用请求，或者接收 READ 或者接收 WRITE 调用。由于每次仅能处理一个调用，故防止了对变量的“多重存取”。本例没有说明如何控制调用次序，关于调用次序的控制留待 14.2 节中加以讨论。

让我们再举典型的有界缓冲的例子说明 select 语句的较为复杂的形式。有界缓冲的解为：

```

task  BUFFERING  is
    entry  PUT(X,in  ITEM);
    entry  GET(X,out  ITEM);
end;

task  body  BUFFERING  is
    N;constant:=8;
    A:array(1..N)  of  ITEM;
    I,J:INTEGER  range  1..N:=1;
    COUNT:INTEGER  range  0..N:=0;
begin
    begin
        loop
            select
                when  COUNT<N=>
                    accept  PUT(X,in  ITEM)  do
                        A(I):=X;
                    end;
                    I:=I  MOD  N+1;COUNT:=COUNT+1;
                or
                when  COUNT>0=>
                    accept  GET(X,out  ITEM)  do
                        X:=A(J);
                    end;
                    J:=J  MOD  N+1;COUNT:=COUNT-1;
            end  select;
    end;
end;

```

```
end loop;  
end BUFFERING;
```

此例中有界缓冲为 N 个元素的数组 A(此处 N=8)。变量 I,J 分别被用来检索下一空单元和可用单元的位置,COUNT 用来表示缓冲中已填满的单元个数。设置 COUNT 给使用带来了方便,我们可借助 COUNT 和 I=J 来判别缓冲是否全部填满或者完全空。由于有界缓冲是循环地使用的,因此必须有 I 不大于 J。图14.1给出了缓冲使用过程中的某一情形。

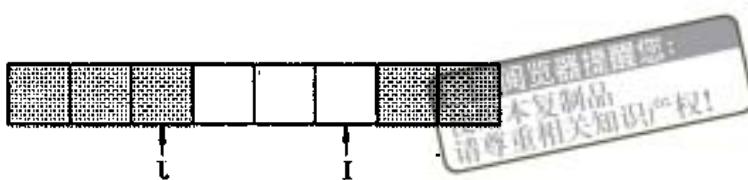


图14.1 有界缓冲

其中,COUNT=5,I=3以及J=6,阴影部分代表已填满的单元。变量 I 和 J 初始化为 1 而 COUNT 初始化为 0,此时表示缓冲被初始化为空。

使用缓冲的任务允许以先进先出的方式填缓冲单元或从缓冲单元中取走内容,但必须防止使缓冲“满溢”或“空溢”。(所谓满溢是指缓冲单元填满后仍填缓冲单元,而空溢则为缓冲空时仍从缓冲取单元内容)。为此需要使用带有保护条件的 select 语句,即 select 语句的更为一般的形式。

每个 select 语句的分支起始处具有:

when 条件=>

其后跟有一 accept 语句和一些其它语句。

每次遇到 select 语句时,就对所有的保护条件求值。只有条件为真的那些分支才被执行。当某分支被执行后,其对应的汇合便被实现。accept 语句后可有一些语句,这些语句由服务任务视为选择语句的一部分来执行,但它们的动作发生在汇合以外。

因此,一服务能够被提供的条件是其保护条件必须为真。accept 语句的执行意味着与消费者汇合并提供服务。accept 语句后的语句用来记录服务后的事宜,这种工作在消费者离开后能够而且必须做,以便为下一个消费者提供服务。

在我们的例子中,作为能够接收 PUT 调用的条件仅仅为缓冲必须不全满,此条件用 COUNT<N 加以表示。同理,接收 GET 调用的条件为缓冲不空,此条件有 COUNT>0 加以表示。对应的 accept 语句体中的语句是把单元内容送入缓冲或是从缓冲中取出。汇合完成后,拨动 I 或 J 指针且修改 COUNT 以反映状态变化。采用 mod 修改 I 和 J 是为了适应循环方式。

正如我们看到的那样,首次执行选择语句时 COUNT<N 为“真”而 COUNT>0 为“假”,因此只有 PUT 的调用能够被接收。调用 PUT 将内容送入缓冲。此后两个条件均为“真”,PUT 或 GET 中的任何一个都能够被接收,向缓冲送内容或从缓冲中取走内容,直至两个入口执行后出现保护条件为“假”止。从而防止了“满溢”或“空溢”的发生。

有一点需要强调:每次执行 select 语句前都要对保护条件求值(但对它们求值的次序不固

定)。保护条件缺省时,视其为“永真”。如果对所有分支的保护条件所求的值均为“假”,则将引发异常 PROGRAM_ERROR。

还有一点需要说明,当汇合发生以后,并不要求保护条件一直保持为“真”,因为在保护条件下可能使用全局变量且有可能由于其它任务的活动将其修改。以后我们将讨论保护条件被意外地改变的例子。

本例中,显然没有什么会导致错误,因为总有一个保护条件为“真”,并且两个分支中都只含有局部变量 COUNT,COUNT 的值不可能在汇合时或在对保护条件求值时发生被其它任务改变的情况。

对于任务 PROTECTED_VARIABLE,一次仅允许一个操作(或者为读或者为写)。这种对活动的限制稍微有些苛刻。典型的读一写问题是:一次只允许一个写操作,但一次允许多个读操作。下面给出典型读一写问题的一种简单的解,这个解采用程序包含局部任务的形式。

```
package READER_WRITER is
    procedure READ(X,out ITEM);
    procedure WRITE(X,in ITEM);
end;
package body READED_WRITER is
    V:ITEM;
    task CONTROL is
        entry START;
        entry STOP;
        entry WRITE(X,in ITEM);
    end;
    task body CONTROL is
        READERS:INTEGER:=0;
    begin
        accept WRITE(X,in ITEM) do
            V:=X;
        end;
        loop
            select
                accept START;
                READERS:=READERS+1;
                or
                accept STOP;
                READERS:=READERS-1;
                or
                when READERS=0=>
                    accept WRITE(X,in ITEM) do
```



```
V:=X;
end;
end select;
end loop;
end CONTROL;

procedure READ(X,out ITEM) is
begin
    CONTROL.START;
    X:=V;
    CONTROL.STOP;
end READ;

procedure WRITE(X,in ITEM) is
begin
    CONTROL.WRITE(X);
end WRITE;
end READED_WRITER;
```

任务 CONTROL 具有三个入口。WRITE 做写操作,而 START 和 STOP 一起配合起来做读操作。对 START 的调用表明希望开始进行读操作,对 STOP 的调用表示读操作结束。把任务包裹在程序包中,是希望提供多重读操作。多重读操作的实现是通过设置一个可重入的 READ 过程实现的。该过程先调 START 入口,再调 STOP 入口。

读一写问题的整个解正如我们看到的那样为一个程序包。在这个程序包中含有一个变量 V,一个任务 CONTROL 和供使用的 READ 过程和 WRITE 过程。READ 过程开始处调用入口 START,结束处调用入口 STOP,中间为赋值语句 $X := V$ 。WRITE 过程只是调用入口 WRITE。

任务 CONTROL 定义了一个变量 READERS,用这个变量指明现今有多少个读者。任务 CONTROL 的开始处设一个带有 WRITE 入口的 accept 语句,正如前面已经讨论过的那样,目的在于对变量 V 进行初始化。在这个语句后为一个含有 select 语句的 LOOP 语句。select 语句具有三个分支,每个入口为一个分支。每次的 START 或 STOP 调用使读者个数计数器加“1”或减“1”。WRITE 调用只有在读者个数计数器 $READERS = 0$ 为真时才可被接收。因此,有读者存在时写操作即被禁止。由于任务 CONTROL 在接收写操作的汇合期间不可能再进行其它动作,因此防止了多重写操作的发生。此外,任务 CONTROL 在与它任务调用入口 WRITE 的汇合期间亦不可能接收 START 调用,故此,写操作进行时也就不可能再进行读操作,但正如我们看到的那样,多重读操作被允许。

上述读一写问题的解尽管满足了一般情况,但它并不是真正令人满意的,因为存在读者队伍对变量读操作时,则要完全阻塞写者一段时间,写操作往往对一个系统来说也许更为重要和急迫,因此这种过久地延缓写操作的做法是不足取的。一种改进方法是:如果有一个或多个写者处于等待的话,则再有的读者将被禁止进入。为了实现这种改进方法,我们不妨使用属性 WRITE'COUNT 作为保护条件, select 语句现在变为:



```
select
when WRITE'COUNT=0=>
accept START;
READERS:=READERS+1;
or
accept STOP;
READERS:=READERS-1;
or
when READERS=0=>
accept WRITE(X,in ITEM) do
  V:=X;
end;
end select;
```

属性 WRITE'COUNT 是当前在入口 WRITE 队列中的任务个数,取属性计数作为保护条件使用需要谨慎。在对保护条件进行求值时它给出确切的值,而在汇合发生前它便有可能发生变化,因为很有可能另一任务又加入队列而它被加“1”。(在本例中属性 WRITE' COUNT 变化不造成影响)。我们以后将看到它不可预料的减“1”所带来的问题。

还有一些其它形式的 select 语句,其可能有一个或多个分支,分支开始处有可能为 delay 语句而不是 accept 语句,例如:

```
select
accept READ(...) do
...
end;
or
accept WRITE(...) do
...
end;
or
delay 10 * MINUTES;
--其它语句
end select;
```

如果在10分钟内既无调用 READ 又无调用 WRITE 的请求被接收,则第三个分支被选中,此时执行 delay 后的语句。这时,任务可被认为由于不再需要它们服务时,它们能够去做其它的事,或者它们可以临时去处理应急事件。比如,某过程控制系统中,需要等待操作员的回答以确定何种动作发生,如在等待一段时间后没有得到回答,系统便处理自己的应急事件。

```
OPERATOR CALL("PUT OUT FIRE");
```

```
select
  accept  ACKNOWLEDGE;
or
  delay  1 * MINUTE;
  FIRE_BRIGADE_CALL;
end select;
```

延迟方案可以出现多个延迟分支,但只有时间最短的有效。如果有一个 accept 语句执行,则其它的延时都被取消——我们可以认危。用延迟的方法好比拿着时钟等待汇合。

显然 select 语句开始执行时,延迟的时间被置,而后每遇到一次 select 语句,延迟时间就被重置一次。

另一种 select 语句是带有 else 部分的 select 语句。如:

```
select
  accept  READ(...)  do
  ...
end;
or
  accept  WRITE(...)  do
  ...
end;
else
  --语句
end select;
```

在这种形式的 select 语句中,最后的分支由 else 而不是 or 引导,该分支由一系列语句所组成。一旦没有其它分支的 accept 语句执行,则 else 分支立即执行。else 分支等效于 delay 0.0 为先导语句的分支。不难知,在其它分支没有调用请求时,便认为 delay 0.0 为先导的分支由于延时到达而立即执行。在 select 语句中不能既含 else 部分又含延迟分支。

accept 语句出现在一选择分支的开始处同它出现在任何其它处有着微妙的差别。出现在分支的开始处,accept 语句的执行与 select 所做的事有密切的关系并且隐含有分支入口条件的意义。而在其它地方出现的 accept 语句,一旦遇到,不管怎样都将被执行。

在分支开始处的 delay 语句与其它地方出现的 delay 语句亦有同样的差别。因此,如果我们把处理紧急事件处的 or 改成 else,

```
select
  accept  ACKNOWLEDGE;
else
  delay  1 * MINUTE;
  FIRE_BRIGADE_CALL;
```

```
end select;
```

则此处的 delay 语句的含义与以前完全不同,它仅作为一系列语句中的一个语句出现且按一般的方式执行。在这种表示中,如果不能立即接收 ACKNOWLEDGE 调用,则直接转入 else 部分,第一个语句为延时语句(纯属偶然),在延时一分钟后调 FIRE_BRIGADE。

如果 select 语句中有 else 部分,那么 PROGRAM_ERROR 不会被引发。由于 else 部分不能设有保护条件,因此当所有其它分支设有保护条件并且条件均为假时,总是要转入 else 给出的分支。

有两种完全不同形式的 select 语句,它们仅有一个入口调用而不是一个或多个 accept 语句,在给定的时间内该入口调用不被接收,则允许执行延时分支的一系列语句。有识别权

```
select OPERATOR.CALL("PUT OUT FIRE");
or
delay 1 * MINUTE;
FIRE_BRIGADE.CALL;
end select;
```

如果操作员在一分钟内不接收对其发出的调用,则将调用 FIRE_BRIGADE。因为重要的是汇合的开始,而不是汇合的终结,因此可没有延时,有

```
select
OPERATOR.CALL("PUT OUT FIRE");
else
FIRE_BRIGADE.CALL;
end select;
```

如果操作员不能立即接收对其发出的调用,则将调 FIRE_BRIGADE。延时和条件入口调用与一般的 select 语句十分不同,它们只含有一个非保护的调用,故选择语句总是只有两个分支——一个作为入口调用,另一个作为选择执行的语句序列。延时和条件调用仅适用于入口(entry),不能用于过程甚至不能用于重命名为过程的入口。在服务的任务很忙时,处于请求服务的任务不希望过久地等待,那么延时和条件调用是很有用的。它们相当于在商店外的顾客,在等了一会后,便放弃排队离开。条件调用相比之下则是个性情更加急躁的顾客,如果不能立即得到服务便马上离去。

利用 COUNT 属性的延时调用,在使用时需略加小心。基于属性值的决断也许失效,因为延时调用无法预料任务从一入口队列中离去。作为例子,我们讨论程序包 READER_WRITER,由于入口被封闭在过程 READ 和 WRITE 中,因此入口调用不能为延时调用,但我们可以通过提供这些过程的“重载”来设置延时调用。例如,我们可以加

```
procedure WRITE(X,in ITEM,T,DURATION,OK,out BOOLEAN) is
begin
```

```

select
  CONTROL.WRITE(X);
  OK:=TRUE;
or
  delay T;
  OK:=FALSE;
end select;
end WRITE;

```

不幸的很,这是无效的。假定一个写者正处于等待(有 $WRITE'COUNT=1$),但在对保护条件进行赋值和 accept 语句执行之间超过了延时,此时根据 READER 的值会导致两种可能的情况。如果 $READER=0$,由于没有现行读者则不可能有任务调用 STOP,又由于此时保护条件为假,START 的调用不可能被接收,又因为已越时所期望的 WRITE 也不会出现,结果所有新读者被不必要地阻塞,直至一个新写者的抵达。另外如果 $READER>0$,尽管保证了新来的 WRITE 调用正确地阻塞,但新来的 START 调用被不必要地延迟至存在一个读者调用 STOP。

鉴于此,我们需要寻找一种新的办法来解 READER_WRITE 问题。

以前我们采用 $WRITE'COUNT$,为的是用来防止读者插入写者的队列。有一种解是采取调用公共入口的办法。也就是说,无论是读者或是写者,在它们开始工作前均需调一公共入口,并保证它们的操作按序进行。此公共入口能够带参,用来指明所需的服务请求,并且需要的话,调用者可被置入辅助队列中。这一方法可用下面的解来加以说明。package 的规范式说明如前。

```

package body READER_WRITER is
  V,ITEM;
  type SERVICE is(READ,WRITE);
  task CONTROL is
    entry START(S,SERVICE);
    entry STOP_READ;
    entry WRITE;
    entry STOP_WRITE;
  end CONTROL;
  task body CONTROL is
    READERS:INTEGER:=0;
    WRITERS:INTEGER:=0;
begin
  loop
    select
      when WRITERS=0=>
        accept START(S,SERVICE) do
          CASE S is

```

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

```
when READ=>READERS:=READERS+1;
when WRITE=>WRITERS:=1;
end case;
end START;
or
accept STOP_READ;
READERS:=READERS-1;
or
when READERS=0=>
accept WRITE;
or
accept STOP_WRITE;
WRITERS:=0;
end select;
end loop;
end CONTROL;

procedure READ(X,out ITEM) is
begin
CONTROL.START(READ);
X:=V;
CONTROL.STOP_READ;
end READ;

procedure WRITE(X,in ITEM) is
begin
CONTROL.START(WRITE);
CONTROL.WRITE;
V:=X;
CONTROL.STOP_WRITE;
end WRITE;
end READER_WRITER;
```

我们引入变量 WRITERS 用来指明在系统中是否存在写者，此变量只能取值“0”或“1”。所有请求首先需调用入口 START，该入口使调用者等待直至无写者存在。READERS 或 WRITERS 能够适当地被修改。在无写者存在的情形下，由读者发出请求时，如以前所述那样，可被接收，进行读操作而最终调用 STOP_READ 完成其读操作。在由写者发出请求时，必须等待没有读者，这是由调用 WRITE 来保证的，在完成写操作后调用 STOP_WRITE 为使任务 CONTROL 将 WRITERS 重置为“0”。正如我们看到的那样，把 STOP_WRITE 从 WRITE 中分离出来，使我

们亦能够达到延时,利用过程中的 WRITE 来代替实际的延时语句,效果是一样的。读者的情形亦类似。

在我们的这个解中,变量 WRITERS 实现了以前 WRITE' COUNT 的功能,由我们自己来计数,使得在可以控制的状态下保持状态。START 和 WRITE 的调用提供延迟功能,一旦 START 调用接受后,必定要调用 STOP _ READ 或 STOP _ WRITE。

需要指出的是,在此解中忽略了需确保第一次调用为写,这可以用各种方法来解决。如用一个特殊的起始入口或把读者置入一辅助队列强迫它们等待第一个写者的到来。

现我们给出一个较为周全的解来结束本节。

```
task CONTROL is
    entry START(S,SERVICE);
    entry STOP;
end CONTROL;
task body CONTROL is
    READERS:INTEGER:=0;
begin
    loop
        select
            accept START(S,SERVICE) do
                case S is
                    when READ=>
                        READERS:=READERS+1;
                    when WRITE=>
                        while READERS>0 loop
                            accept STOP; --来自读者
                            READERS:=READERS-1;
                        end loop;
                end case;
            end START;
            if READERS=0 then
                accept STOP; --来自写者
            end if;
        or
            accept STOP; --来自一读者
            READERS:=READERS-1;
        end select;
    end loop;
end CONTROL;
```



```
procedure READ(X,out ITEM) is
begin
    CONTROL.START(READ);
    X:=V;
    CONTROL.STOP;
end READ;

procedure WRITE(X,in ITEM) is
begin
    CONTROL.START(WRITE);
    V:=X;
    CONTROL.STOP;
end WRITE;
```



这个解的本质是写者等待所有读者的会合完成，并且整个写进程的处理在 select 语句一分支内实现。CONTROL 任务只有两个入口，入口均不设保护条件。仅提供一个公共的 STOP 是可行的，因为在结构上每个 accept 语句仅涉及一个无条件调用者，稍有不利于解的是其失去了由定时调用的灵活性，一旦写者对 START 的调用被接收后，它需等待所有读者完成读操作。

14.5 任务类型和任务激活

在有些使用场合，往往需要在不能预先确定所需任务数量的情况下，设置许多类似但性质不同的任务。例如，我们希望通过创建任务来控制每架飞机在飞机控制系统的控制范围内飞行。显然，这种任务需要根据飞机的起飞而动态地创建，它们不再与程序的静态结构有关。尽管

```
end SIMPLE;
```

这完全等价于先说明任务类型 anon:

```
task type anon is  
...  
end anon;
```

继之以

```
SIMPLE;anon;
```

任务对象能够用于通常的结构中,因此我们能够说明任务数组。

```
AT:array(1..10) of T
```

记录亦可以含有任务

```
type REC is  
record  
CT:T;  
...  
end record;  
R:REC;
```

等等。

上述任务的接口,在被调用时需引用对象任务名,故可以写为:

```
X.E(...);  
AT(1).E(...);  
R.CT.E(...);
```

应该特别指出,任务实体是不变的,从这点看,它们很象常量。任务类型没有赋值和等同测试运算。可见任务类型是一种受限私有类型。尽管任务对象很象常量,但它们不能象常量那样说明,因为常量说明需要一个显式的初值。子程序参数可以是任务类型的,它们总是通过存取而有效地传递,形参和实参总是指同一个任务。如同其它限定类型的情况一样,参数的模式 in 或 out 在程序包外定义是不允许的,当然,亦不存在任何形式的用任务类型定义的程序包。

在节14.1我们简要地引入了“从属”的思想,在那里任务从属于某些程序单位。被说明的任务从属于其说明所处的闭合分程序、子程序或任务体。任务不从属于程序包,这是因为程序包仅仅是把动态活动范围与外部环境相隔离的墙,而其本身不具有动态特性。如果一任务在程序包内说明(或在嵌套的程序包内说明),则该任务从属于该程序包内的分程序、子程序或任务体。为了完整性,在库程序包中说明的任务,被认为从属于该程序包,然而,程序包不具有动态生命,因此这种从属并不影响此任务的结束。

我们先前已经看到,任务只有当其被说明的程序单位遇到说明后的 begin 时才被激活。一个任务的执行可以认为具有两级进程。第一级进程被视为激活,由任务体的详细说明所构成,而第二级进程显然由任务体的语句执行所构成。处在激活级阶段,父程序单位不得继续。如果

在一程序单位中说明了若干个任务则它们的激活以及它们的语句执行的子序列以独自的方式并发地执行,只有当所有任务的激活完成后父程序单位方能与任务并发地执行 begin 后的语句。这一过程如图14.2所述:

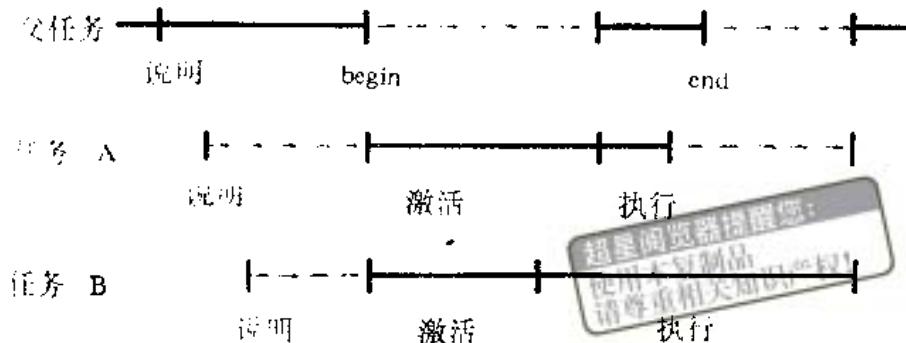


图14.2 任务的激活

图14.2展示了一个含有说明了两个任务 A 和 B 的情形:

```
declare
  ...
  A;T;
  B;T;
  ...
begin
  ...
end;
```

在图14.2中,时间依从左到右递增,实线表明一程序单位实际做某事,而虚线表明该程序单位被挂起。

为了说明起见,我们假定在任务 B 之后任务 A 完成它的激活,因此父程序单位在任务 A 进入执行阶段后方恢复执行。在图14.2中我们亦假定任务 A 先于任务 B 完成执行变为终止,而父程序单位到达 end 是在任务 A 终止后,任务 B 终止前。此时父程序单位被挂起直至任务 B 也终止。

任务也能够通过存取类型来建立,例如,先说明一个存取类型:

```
type REF_T is access T;
```

然后按一般方式借助分配符建立一任务:

```
RX:REF_T:=new T;
```

REF_T 是一个普通的存取类型,对该类型对象的赋值和等同测试都是允许的。借助下列形式,我们可以调用 RX 提供的入口 E。

```
RX.E(...);
```

由存取类型建立的任务,其激活与从属规则与前述的略有不同,即不论分配符是在语句序列中出现或是在对象说明的初值中出现,它们立即被激活——而不必等到遇 begin。这种借助

分配符建立的任务并不从属于它被建立时所在的程序单位,而从属于相应存取类型被说明时所在的分程序、子程序体或任务体。

当我们讨论的任务为复合对象的分量时,所说明的任务对象与经由分配符建立的任务完全相同。假定有如下的类型:

```
type R is
  record
    A:T;
    I:INTEGER;:=E;
    B:T;
  end record;
```



其中,A 和 B 为某个任务类型 T 的分量,而 E 为分量 I 的初值表达式。用类型 R 说明对象 X 则有

```
declare
  X:R;
begin
```

这与分别对这些对象加以说明完全一样。

```
declare
  XA:T;
  XI:INTEGER;:=E;
  XB:T;
begin
```

假定有存取类型

```
type REF_R is access R;
```

则可使用分配符建立对象

```
RR:REF_R:=new R;
```

在此,首先发生的事为各个分量被说明且初值表达式被求值并赋值于 RR.I。就如同在直接对对象进行说明的场合遇 begin 时对象任务被激活的情况一样,当测得到达 end record 时,任务 RR.A 和 RR.B 分别被激活。

分量任务被并发地激活时父程序单位被挂起,直至所有的激活完成后返回存取值,父程序单位方继续执行,这如同以前所述任务的激活过程一样。同样,如果由激活的任务引发异常,则返回分配符立即在父程序单位引发 TASKING_ERROR,引发异常的任务变为完成,但不影响任何其它分量任务的执行。如果 E 的求值在父程序单位引发异常的话,则分量任务将自动地变成结束而不被激活。

需要指出的是:一旦任务被说明,任务的入口就能被调用,甚至允许调用发生在激活之前——此时该调用被排入队列中,通常这种情况是很少有的。

任务类型通常的应用是建立“代理者”，所谓代理者是指一任务以另一任务的名义去做某些事。作为例子，我们假定有一任务 SERVER，只要调用其入口 REQUEST，它便提供某种服务。再假定提供服务要花一些时间，因此让调用任务 USER 在等待 SERVER 回答时，去做某些其它的事显然是合理的。USER 有许多方法来收集由 SERVER 回答的信息，譬如它可以调用 SERVER 的另一个入口 ENQUIRE 来收集回答信息，这也要求 SERVER 任务有某种识别请求任务的方法，如请求任务在调用 REQUEST 时先获得一个特定关键字，以后在调用 ENQUIRE 时提交该关键字供检验。这好比先发一张证券，以便适当时候供持券者凭券获取数据。一种不采用提供关键字的方法是建立代理者。这相当于留下你的地址后，便有维修单寄给你。我们现在来说明这种方法。

首先我们定义任务类型如下：

```
task type MAILBOX is
    entry DEPOSIT(x,in ITEM);
    entry COLLECT(X,out ITEM);
end;
task body MAILBOX is
    LOCAL;ITEM;
begin
    accept DEPOSIT(X,in ITEM) do
        LOCAL:=X;
    end;
    accept COLLECT(X,out ITEM) do
        X:=LOCAL;
    end;
end MAILBOX;
```

这种类型的任务充当着一个简易的信箱，一信件能够投放其中以及从中取出。接下来的事，是把同一个信箱给服务任务和用户任务，以便服务任务向其中投信而用户任务以后把信取走。为此我们需要一存取类型

```
type ADDRESS is access MAILBOX;
任务 SERVER 和 USER 现在取下面的形式：
```

```
task SERVER is
    entry REQUEST(A,ADDRESS;X,ITEM);
end;
task body SERVER is
    REPLY;ADDRESS;
    JOB;ITEM;
begin
    loop
```

```

accept REQUEST(A;ADDRESS,X;ITEM) do
    REPLY:=A;
    JOB:=X;
end;
--工作在JOB上
REPLY.DEPOSIT(JOB);
end LOOP;
end SERVER;
task USER;
task body USER is
    MY_BOX;ADDRESS:=new MAILBOX;
    MY_ITEM;ITEM;
begin
    SERVER.REQUEST(MY_BOX,MY_ITEM);
    --当等待时做某些事
    MY_BOX.COLLECT(MY_ITEM);end USER;

```

实际使用中,用户任务可以一次次地询问信箱是否投有信件,这可用条件入口调用很简便地加以实现。

```

select
    MY_BOX.COLLECT(MY_ITEM);
    --信件成功地接收
else
    --信件还没有准备好
end select;

```

让代理者服务于多个目的是重要的(在本例中代理者使得信件的存入与取出分开),从而有助于 SERVER 为下个工作服务。此外,更为重要的还在于,直接地调用用户任务要求用户任务必须是一个特殊的任务类型,这样的要求非常不合理,实现上,服务任务应该不需要知道用户任务任何东西。

当不要求将信件的投入与取出相分开时,代理者的体可写为:

```

task body MAILBOX IS
begin
    accept DEPOSIT(X,in ITEM) do
        accept COLLECT(X,out ITEM) do
            COLLECT.X:=DEPOSIT.X;
        end;
    end;

```



```
end MAILBOX;
```

在这里，代理者不再需要设置局部变量 LOCAT，因为该变量只是用作协助通信而存在。在嵌套的接收语句中使用了加点表示法，指在区分两个不同的 X，当然也可以写为 X:=DEPOSIT.X，用了 COLLECT 则更加清晰。

14.8 任务的终止与异常

一个任务除了运行到它最后一个 end 终止外，还可以用其它各种方式终止。在前面的一些例子中，任务体是一个无限循环，无疑它是不可能终止的。也就是说，该任务不可能脱离其从属于的程序单位。假如程序中需要一些保护变量，我们可以说明一任务类型：

```
task type PROTECTED_VARIABLE is
    entry READ(X,out ITEM);
    entry WRITE(X,in ITEM);
end;
```

然后对变量进行如下说明：

```
PV:PROTECTED_VARIABLE;
```

便可以

```
PV.READ(...);
PV.WRITE(...);
```

存取它们。

显然，在不采用某种方式结束该任务之前是不能离开 PV 的作用域的。为了终止该任务，可增加一个特殊的入口 STOP，仅在离开 PV 作用域之前调用它。在 Ada 中，可使用一种特殊形式的选择分支——终止分支，当任务不再使用时，借助它使任务本身自动终止。

此时，任务体可写为：

```
task body PROTECTED_VARIABLE is
    V,ITEM;
begin
    accept WRITE(X,in ITEM) do
        V:=X;
    end;
    loop
        select
            accept READ(X,out ITEM) do
                X:=V;
            end;
        end select;
    end loop;
end;
```

```

end;

or

accept WRITE(X,in ITEM) do
    V:=X;
end;

or

terminate;
end select;
end loop;
end PROTECTED_VARIABLE;

```



执行选择语句时,若该任务的父程序单位到达它最后的 end,并且所有伙伴任务及其子任务均已终止(或执行终止分支)时,该任务选择终止分支执行。

严格地说,最开始的 WRITE 也应位于含终止分支的选择语句内,否则,万一该任务一次也不被调用的话,任务仍不能终止。

终止分支可以带保护条件。然而,它不可出现在含延迟分支或 else 分支的选择语句内。选择语句中终止分支执行,是任务的正常结束。而夭折(abort)语句的执行却提供了任务非正常结束的手段。它们的执行迫使一个或多个任务无条件地结束。

夭折语句包括保留字 abort,并后继以一个任务名或被逗号分隔的若干任务名。例如:

```
abort X,AT(3),RX.all;
```

如果一任务夭折则所有从属于该任务的子任务以及当前被它调用的子程序或分程序都结束。

如果在夭折语句中涉及的任务处于挂起状态,则它直接结束,任何延迟亦被取消。

如果夭折所涉及的任务为发出调用的任务且该任务正处在入口队列中,则从入口队列中删除。发出调用的任务夭折,对被调任务不产生影响,汇合继续执行到结束。

如果夭折所涉及的被调任务正忙于汇合(执行接收语句、选择语句或等待语句),则该任务被夭折并且导致其入口队列上等待汇合的任务在调用入口处引发异常 TASKING_ERROR。这样做是符合情理的,因为一任务在请求服务时,服务者消失,那么,必须告诉消费者服务不能予以提供,反之,如果消费者在服务者为其服务期间消失,我们应当使这种服务继续下去,以避免由此干扰服务工作,因为有可能服务者所具有的数据库等正处于临界状态。

如果夭折时所涉及的任务尚未被激活,则此任务不仅变为完成而且也变为终止。上述规则除最后一条外,均指任务的结束而不是终止。这是由于父任务只有当所有从属于它的任务均被终止后方能被终止,并且从属的子任务只要有一个是处在与第三者汇合的调用者,则终止过程将被延迟。

不难知,任务的完成常能分别由强制方法而导致,而任务的终止则将以通常的方式自动地引起。

夭折语句具有很强的破坏性,只有在万不得已时才使用它,譬如,我们可以用它作为命令处理来响应操作员命令去夭折一已完成的工作。

另一种可能使用夭折语句的场合是在异常处理。由于仅当所有从属任务被终止后,方能终

止一程序单位。因此，在一个程序单位中引发异常，我们还不能完结该程序单位，只有让异常传播导致所有活动的从属任务都夭折。**10.2**节的过程 CLEAN_UP 用来实现这一点。

然而，借助夭折语句来控制程序单位的完结，也许是一种最好的方法。比如，在命令处理任务中的语句可采用如下形式：

```
select
  T. CLOSEDOWN;
or
  delay 60 * SECONDS;
  abort T;
end select;
```



如果从属任务不在一分钟内接收 CLOSEDOWN 调用，便夭折该任务。我们假设从属任务用控制接收语句至少每分钟查询一次 CLOSEDOWN 入口，例如：

```
select
  accept CLOSEDOWN;
  --进行夭折处理
else
  --按正常执行
end select;
```

如果即便从属任务在接收了入口调用后，我们仍然不能委托“关闭”，加上夭折一已终止的任务则夭折语句不起作用，则在此情况下命令处理任务可改写如下，使其始终在一个适当的时间间隔后产生一天折。

```
select
  T. CLOSEDOWN;
  delay 10 * SECONDS;
or
  delay 60 * SECONDS;
end select;
abort T;
```

尽管汇合发生后，从属任务可能永远地执行下去，但这亦不失为一个好的方法。

```
accept CLOSEDOWN do
  loop
    PUT("CANT CATCH ME");
  end loop;
end;
```

不难由此看出,在并发程序的设计上,进行某些细小程度上的协作往往是要的!任务 T 的状态,可以通过两个属性来查明。若任务 T 已终止,则属性 T' TERMINATED 为真。另一属性 T' CALLABLE 为真是在任务完成或终止或处于异常发生时。

使用上述两属性须格外小心。比如,在借助属性查到一任务还没有终止,当根据此信息决定某种行为时,所查的任务可能已终止了。反之根据一任务已终止的信息决定某种行为则是十分安全的,因为一任务不太可能被重新启动。

为了说明非正常终止导致的影响,以及如何来解决非正常终止导致影响的后果,我们在此重新讨论14.4节最终形式的READER _ WRITER 程序包。

```
task body CONTROL is
  READERS:INTEGER := 0;
begin
  loop
    select
      accept START(S,SERVICE) do
        case s is
          when READ=>
            READERS:=READERS+1;
          when WRITE=>
            while READERS>0 LOOP
              accept STOP; --来自读任务
              READERS:=READERS-1;
            end loop;
          end case;
        end START;
      if READERS=0 then
        accept STOP; --来自写任务
      end if;
      or
      accept STOP; --来自一读任务
      READERS:=READERS-1;
    end select;
  end loop;
end CONTROL;
```

假定一读任务已调 START 并且在它调 STOP 之前,此读任务被夭折。变量 READERS 的值则是不正确的,该值不再为 0,下个写任务将被永远挂起。同理,假定一写任务已调 START 并且在它调 STOP 之前夭折,则所有其它等待在入口队列中的任务被永远挂起。

问题出在,进入任务 CONTROL 的活动被认为是调用任务活动的一部分,并且这部份活动得到保障。为克服所存在的这一问题,可引入一代理者,我们设法保障充任中间代理者的任务

不被夭折。下面我们展示的是，为读任务引入代理者的使用情况，同样技术可用于写任务，故 READER_WRTTER 程序包体改写为：

```
package body READER_WRTTER is
V;ITEM;
type SERVICE is (READ,WRITE);
task type READ_AGENT is
entry READ (X,out ITEM);
end;
type RRA is access READ_AGENT;
task CONTROL is
entry START(S,SERVICE);
entry STOP;
end;
task body CONTROL is
--与以前相同
end CONTROL;
task body READ_AGENT is
begin
select
accept READ(X,out ITEM) do
CONTROL.START(READ);
X:=V;
CONTROL.STOP;
end;
or
terminate;
end select;
end READ_AGENT;
procedure READ(X,out ITEM) is
A:RRA:=new READ_AGENT;
begin
A.READ(X);
end READ;
procedure WRITE(X,in ITEM) is
begin
...
end WRITE;
end READER_WRTTER;
```



我们如若现在夭折调过程 READ 的任务，则要么与其代理者的汇合继续执行下去，从而该过程执行完，要么汇合不再继续，如同没有接口的情况一样。这样看来，表现在代理者方面，要么做完其工作，要么根本不做任何事。但是，倘若我们建立的代理者是直接任务对象（而不是一存取任务对象），那么夭折调用过程 READ 的任务也将立即夭折代理者，因为此时代理者仅仅被看作为从属任务。

用存取类型建立的代理者所从属的单位在 READER _ WRITER 中加以说明，因而其是有生命的。

代理者含有一个带终止选择分支的选择语句，这保证了，假如调用 READ 过程的任务处在建立代理者或调用代理者时被夭折，则在离开代理者所从属的程序单位后，代理者总能自行消失。

现在让我们对上述情况加以小结：

- 代理者是无形的，因此它不能被夭折。
- 如果调用任务在汇合时被异常所终止，不影响被调任务（代理者）。
- 代理者是一存取任务且不从属于调用者。

最后，让我们对有关异常所剩的要点加以讨论，以结束本节。

异常 TASKING _ ERROR 一般与通信失败有关，正如我们已经看到的那样，它由任务执行过程中出现了故障而引发，亦由汇合时因被调任务夭折而对调用者引发，当一任务完成后，仍然排在其入口队列中的任务均接收 TASKING _ ERROR，同理，调用一个已完成的任务入口也会引起调用者 TASKING _ ERROR。

若在汇合期间引发了异常，并且接收语句中无异常处理段，则异常传播到汇合双方，当然，若接收语句在内部处理异常，则异常结束于汇合一方。

此外，如若异常在一任务中根本不被处理，那么异常在导致该任务废弃的同时本身亦被丢失，在这种情况下，它不传播到父程序单位，主要是因为这种传播破坏性太大。

我们希望运行时，由运行环境提供诊断信息。对所有重要的任务，在最外层设有公共处理以防止异常丢失和任务悄然夭折，这一点十分重要。

14.6 练习

1. 以如下的规范式说明重写节 14.4 中的任务 BUFFERING

```
task BUFFERING is
  entry PUT(X,in ITEM);
  entry FINISH;
  entry GET(X,out ITEM);
end;
```

写任务在开始时调用 PUT，最后调 FINISH。读任务在开始时调用 GET，当无数据项时，对 GET 的调用将引发全程异常 DONE。

14.7 资源调度

在用 Ada 设计任务时，必须记住：我们能够控制的队列仅为入口队列，并且这个队列严格按先进先出(FIFO)的原则来管理的。在设有不同优先级的场合下，有可能会出现：迟来的调用请求被接受而较早的调用请求不得不等待。(注意，在本节我们假定发出调用的任务不会夭

折)。

具有优先级的调用请求可借助入口家族来管理。一个家族相当于一个一维数组。现假设有三个优先级，它们是

```
type PRIORITY is (URGENT,NORMAL,LOW);
```

有一个任务 CONTROLLER 提供访问类型 DATA 的一些动作。对这些动作的请求按优先级排成三个队列。虽然我们可以用三个不同的入口分别对应于这三个队列的各自请求，但用一个入口家族则要更好些。例如：

```
task CONTROLLER is
entry REQUEST(PRIORITY)(D;DATA);
end;

task body CONTROLLER is
begin
loop
select
accept REQUEST(URGENT)(D;DATA) do
ACTION(D);
end;
or
when REQUEST(URGENT)'COUNT=0=>
accept REQUEST(NORMAL)(D;DATA) do
ACTION(D);
end;
or
when REQUEST(URGENT)'COUNT=0
and REQUEST(NORMAL)'COUNT=0=>
accept REQUEST(LOW)(D;DAT) do
ACTION(D);
end;
end select;
end loop;
end CONTROLLER;
```

REQUEST 是一个入口家族，其下标变量是类型为 PRIORITY 的离散量。很显然，当优先级的分级数较小时，这种方法是可行的。当优先级的分级数较大时，则需要一个较为复杂的技术。比如我们可以采用轮流测定每个队列的办法：

```
task body CONTROLLER is
begin
```

```
loop
    for P in PRIORITY loop
        select
            accept REQUEST(P)(D;DATA) do
                ACTION(D);
            end;
            exit;
        else
            null;
        end select;
    end loop;
end loop;
end CONTROLLER;
```

上述算法并不能让人满意,因为当所有队列为空时,任务 CONTROLLER 仍然要反复地循环检测。为此,我们希望有一种机制,利用它,任务能够等待任何一先行到达的调用请求。为了做到这一点,我们设置两级进程。发出调用的任务必须先通过访问一公共入口来进行登记,然后再调用入口家族,实现细节作为练习留给读者。

现举例说明一个十分常用的技术。此技术允许在单个入口队列中的请求按任选的次序被处理。考虑从资源集合中分配一组资源的问题。对分配算法,我们不希望来得较早的请求仅由于等待其所希望资源的释放,而阻塞来得较晚的请求,从而妨碍来得较晚的请求可能获得资源。我们用离散类型 RESOURCE 表示资源,用 13.2 节中的类属程序包 SET_OF 能够很容易地表示出对资源集合的操作。

```
package RESOURCE_SETS is new SET_OF(RESOURCE);
use RESOURCE_SETS;
```

资源分配程序为:

```
package RESOURCE_ALLOCATOR is
    procedure REQUEST(S;SET);
    procedure RELEASE(S;SET);
end;
package body RESOURCE_ALLOCATOR is
    task CONTROL is
        entry FIRST(S;SET;OK;out BOOLEAN);
        entry AGAIN(S;SET;OK;out BOOLEAN);
        entry RELEASE(S;SET);
    end;
    task body CONTROL is
```

```

FREE:SET:=FULL;
WAITERS:INTEGER:=0;
procedure TRY(S:SET;OK,out BOOLEAN) is
begin
  if S<=FREE then
    FREE:=FREE-S;
    OK:=TRUE;   --分配成功
  else
    OK:=FALSE;  --无资源,以后再请求
  end if
end TRY;

begin
loop
  select
    accept FIRST(S:SET;OK,out BOOLEAN) do
      TRY(S,OK);
      if not OK then
        WAITERS:=WAITERS+1;
      end if;
    end;
  or
    accept RELEASE(S:SET) do
      FREE:=FREE+S;
    end;
    for I in 1..WAITERS loop
      accept AGAIN(S:SET;OK,out BOOLEAN) do
        TRY(S,OK);
        if OK then
          WAITERS:=WAITERS-1;
        end if;
      end;
    end loop;
  end select;
end loop;
end CONTROL;
procedure REQUEST(S:SET) is
  ALLOCATED:BOOLEAN;
begin
  CONTROL.FIRST(S,ALLOCATED);

```



```

        while not ALLOCATED loop
            CONTROL.AGAIN(S,ALLOCATED);
        end loop;
    end REQUEST;
procedure RELEASE(S;SET) is
begin
    CONTROL.RELEASE(S);
end RELEASE;
end RESOURCE_ALLOCATOR;

```

这是另一个含有控制任务的程序包的例子。其总体结构与节14.4中的程序包 READER_WRITER 相似。程序包 RESOURCE_ALLOCATOR 包含两个过程 REQUEST 和 RELEASE, 这两个过程均将资源集 S 作为须释放的入口参数或者已申请到的出口参数。类型 SET 由实例化的 SET_OF 给出。过程 REQUEST 和 RELEASE 调用任务 CONTROL 的入口。

任务 CONTROL 有三个入口: FIRST, AGAIN 和 RELEASE。其中入口 FIRST 和 AGAIN 是相似的, 不但要提供参数 S 给出所需资源集, 而且要提供出口参数 OK, 用以指明申请资源的请求是否能成功地被满足。FIRST 和 AGAIN 对应的接收语句都调用一公共过程 TRY。在 TRY 过程中, 使用来自 SET_OF 实例中的“包含”运算符 \leq , 根据可用资源集 FREE 来检测 S 集。如果可用资源满足 S, 则使用来自 SET_OF 实例中的“差”运算符相应地修改 FREE 且置 OK 为真。如果资源 FREE 不满足 S, 则置 OK 为假。入口 RELEASE 通过参数 S 将资源退还。此时, 用 SET_OF 实例中的“并集”操作符+修改 FREE, FREE 在说明时被赋初值 FULL, 而 FULL 也是由 SET_OF 实例定义的。

入口 FIRST 和 AGAIN 由过程 REQUEST 来调用。调用 FIRST 即刻产生一个申请资源的请求, 如果资源不能完全满足 ALLOCATED 便被置为 FALSE, 调用 AGAIN 使得发出请求的申请者排入队列, 反复调用 AGAIN 直至成功为止。入口 RELEASE 仅由过程 RELEASE 来调用。

CONTROL 的体是由位于循环中的选择语句所构成, 此选择语句有两个分支: 一个为 FIRST, 一个为 RELEASE。对 RELEASE 的调用总能够被接收。而对 AGAIN 的调用, 只要任务不是正处在执行 RELEASE 的语句序列时, 亦能立即被接收。每经过一个 RELEASE 调用, 都要对因请求资源未能满足而排入 AGAIN 队列中的申请者重新加以考虑。资源由于 RELEASE 的被调用而释放, 故可满足一个或多个申请者的请求。CONTROL 任务每与一调用任务汇合, AGAIN 队列就要被扫描一次。在过程 REQUEST 中, 一个未获得资源的申请者总是通过进一步调用 AGAIN 而将自己置入队列中。为了使在队列中的申请者能再次被扫描到, 我们用变量 WAITERS 控制循环次数。变量 WAITERS 指出有多少调用者因调用 FIRST 不成功(未能获得资源)而处于等待。WAITERS 初值为 0 且每一次 FIRST 调用不成功使其加 1。反之每次 FIRST 调用成功则使其减 1。注意: 我们不能用 AGAIN COUNT 来求得未获资源的申请者数, 因为当一个任务调用 FIRST 不成功而在调用 AGAIN 之前所有资源被释放了时, 会产生死锁。读一写问题的例子从本质上讲便如此, 为此应避免使用 COUNT 属性, 而改用我们自己来计数。

不管怎么说, 上述的解还是相当令人满意的。按队列从前到后的顺序, 在队列中的任务能够彼此超越, 也就是说, 一个新抵达者未必要等待转一轮。

把上述的解改为按请求的先后次序获得资源,即先抵达的请求得不到满足时,后抵达的请求总是被阻塞,即使它们申请的资源完全不同,亦不准许超越。这种改动的本质是:在系统中的某一时刻,仅允许一个等待任务存在。此改动留给读者作为练习。

将上述的资源分配程序,做如下的修改或许更好些。修改的目的在于:避免等待的申请者老是反复地调用 AGAIN。现在让我们看看改进后的任务 CONTROL 和过程 REQUEST:

```
task  CONTROL  is
    entry  SIGN_IN(S;SET);
    entry  REQUEST;
    entry  RELEASE(S;SET);
end;

task  body  CONTROL  is
    FREE,SET,:=FULL;
    WAITERS,INTEGER,:=0;
    WANTED,SET;
begin
loop
    select
        when  WAITERS=0=>
            accept  SIGN_IN(S;SET)  do
                WANTED:=S;
            end;
            WAITERS:=WAITERS+1;
        or
        when  WAITERS>0  and  then  WANTED<=FREE=>
            accept  REQUEST  do
                FREE:=FREE-WANTED;
            end;
            WAITERS:=WAITERS-1;
        or
            accept  RELEASE(S;SET)  do
                FREE:=FREE+S;
            end;
    end  select;
end  loop;
end  CONTROL;

procedure  REQUEST(S;SET)  is
begin
```



```

CONTROL.SIGN_IN(S);
CONTROL.REQUEST;
end REQUEST;

```

在此解中,我们使用了一个保护条件,当请求能够兑现时,此保护条件为真。由于保护条件不能用现行的入口参数来形成,故需要先调用 SIGN_IN 入口以便传入构成保护条件的参数。另外,当没有处于等待的任务时,短路条件方允许对 WANTED 赋值。变量 WAITERS 是一个布尔型,它只能为0或1。

练习 14.7

1. 改进第一种形式的程序包 RESOURCE_ALLOCATOR,以便严格地按序处理请求。
2. 将任务 CONTROLLER 改写成程序包,此程序包含有一任务,其用来避免不断地调整请求等待队列。程序包的规范式说明为:

```

package CONTROLLER is
procedure REQUEST(P,PRIORITY,D,DATA);
end;

```

14.8 与程序包的比较

从整体上看,任务和程序包在词法上是相似的——它们都有规范式说明和实体。尽管如此,任务和程序包之间亦存在许多不同:

- 任务是动态的,而程序包是静态的。
- 任务的规范式说明中只能有入口,而程序包可以有除入口外的其它成分。任务不可有私有部份。
- 程序包可以是类属的,而任务则不能是类属的。但由任务类型可以取得无参类属任务的效果。此外,任务可封装在类属程序包中。
- 程序包可以是一个库单位,而任务则不是,然而任务体可以是一子单位。总之,程序包被认为是实现应用目标的主要工具,而任务仅用于同步。因此典型的子系统是由含有一个或多个任务的程序包(亦可能是类属的)所组成。这种结构具有能对给定设备进行整个控制之优点。

最后,我们举例说明作为类属程序包 BUFFER 中私有类型的任务,及其使用。

```

generic
N;POSITIVE;
type ITEM is private;
package BUFFERS is
type BUFFER is limited private;
procedure PUT(B:in out BUFFER;X,in ITEM);
procedure GET(B:in out BUFFER;X:out ITEM);
private
task type CONTROL is
entry PUT(X,in ITEM);

```

```

        entry GET(X,out ITEM);
    end;
    type BUFFER is new CONTROL;
end;
package body BUFFERS is
    task body CONTROL is
        A;array(1..N) of ITEM;
        I,J;INTEGER range 1..N:=1;
        COUNT;INTEGER range 0..N:=0;
begin
    loop
        select
            when COUNT<N=>
                accept PUT(X,in ITEM) do
                    A(I):=X;
                end;
                I:=I mod N+1;COUNT:=COUNT+1;
            or
                when COUNT>0=>
                    accept GET(X,out ITEM) do
                        X:=A(I);
                    end;
                    J:=J mod N+1;COUNT:=COUNT-1;
            or
                terminate;
        end select;
    end loop;
end CONTROL;
procedure PUT(B,in out BUFFER,X,in ITEM) is
begin
    B.PUT(X);
end PUT;
procedure GET(B,in out BUFFER,X,out ITEM) is
begin
    B.GET(X);
end GET;
end BUFFERS;

```



在此，缓冲是由任务类型 CONTROL 派生出来的对象。当我们说明了一个类型为 BUFFER

的对像时,一个新的任务便被创建出来,同时定义了实际缓冲的存贮区。过程 PUT 和 GET 通过调用对应任务的相应入口,便可存取该缓冲。需要指出的是,选择语句含有一个终止选择分枝,它使得离开任务说明的作用域时,此任务对象便自动地消失。因此,即使发出调用的任务夭折,系统仍然是坚固的。

要点14

- 任务是动态的而程序包是静态的。
- 任务的规范式说明中只能有入口。
- 任务不能是类属的。
- 任务名不能出现在 use 子句中。
- COUNT 属性只能在属于任务自己的入口内使用。
- 入口可重载以及重命名为过程。
- 接收语句决不可出现在子程序中。
- 不能用优先级来实现同步。
- 一次延迟只能置一很小的量。
- 保护条件的求值次序不确定。
- 选择语句只能有一个 else 部份,一个终止选择,一个或多个延迟选择。
- 终止或延迟选择能被保护。
- 几个选择语句可对应于同一个人口。
- 任务类型是受限的。
- 被说明的任务对象局限于一分程序、子程序或任务体但不局限于内部程序包。



第15章 外部接口

本章我们将从各方面讨论 Ada 程序与外部的接口。除了输入输出、中断处理等内容外，还将讨论 Ada 程序在具体机器上的执行。由于这些问题涉及到具体机器，所以希望通过本章的讨论，亦使读者对机器硬件机构有一大致的了解。

15.1 输入和输出

与其它的一些语言不同，Ada 在输入输出方面，本身并没有什么特点，只是提供了一些诸如可重载的子程序和一些类属样本等。由此恰好产生了一个优点：可以在不受限于具体语言实现的情形下，开发出用于各应用领域的输入输出程序包。反过来，这往往又会导致在应用领域内出现“无政府状态”的危险性，这种危险性曾经是 ALGOL 60 “毁灭”的原因之一。为了防止“无政府状态”，LRM 为输入输出定义了标准程序包。在本节和下一节中，我们将讨论这些程序包的一般原理。详细情况读者可参阅 LRM。

输入输出分为二类：二进制(binary)类和文本(text)类。举例如下：

```
I : INTEGER := 75; WRITE(F,I);
```

将 I 以二进制的形式输出到文件 F 中，其格式为(在16位机器中)0000000001001011。实际上，文件 F 可看成是一个一维整数类型的数组。而

```
PUT(F,I);
```

将 I 以文本形式输出到文件 F 中，其格式为0010111100101101。它是设有奇偶校验位的“7”和“5”的字符。在这种情况下，文件 F 可看成是一维字符类型的数组。

二进制类的输入输出又分成顺序型和直接型。它们分别由 SEQUENTIAL_IO 和 DIRECT_IO 通用程序包提供。文本类的输入输出(仅为顺序型)是由非通用的程序包 TEXT_IO 提供。另外，还有一程序包 IO_EXCEPTIONS，提供了上述三个程序包可能出现的异常处理。下一节，我们将按照 SEQUENTIAL_IO, DIRECT_IO, TEXT_IO 的顺序展开讨论。

程序包 SEQUENTIAL_IO 的规范式说明如下：

```
with IO_EXCEPTIONS;
generic
  type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is
  type FILE_TYPE is limited private;
  type FILE_MODE is (IN_FILE, OUT_FILE);
  --File management
  procedure CREATE(FILE:in out FILE_TYPE;
                  MODE,in FILE_MODE:=OUT_FILE);
```

```
        NAME,in STRING;=" ";
        FORM,in STRING;=" ");
procedure OPEN(FILE,in out FILE_TYPE;
        MODE,in FILE_MODE;
        NAME,in STRING;
        FORM,in STRING;=" ");
procedure CLOSE(FILE,in out FILE_TYPE);
procedure DELETE(FILE,in out FILE_TYPE);
procedure RESET(FILE,in out FILE_TYPE;
        MODE,in FILE_MODE);
procedure RESET(FILE,in out FILE_TYPE);
function MODE(FILE,in FILE_TYPE) return FILE_MODE;
function NAME(FILE,in FILE_TYPE) return STRING;
function FORM(FILE,in FILE_TYPE) return STRING;
function IS_OPEN(FILE,in FILE_TYPE) return BOOLEAN;
--Inputand output operations
procedure READ(FILE,in FILE_TYPE;
        ITEM,out ELEMENT_TYPE);
procedure WRITE(FILE,in FILE_TYPE;
        ITEM,in ELEMENT_TYPE);
function END_OF_FILE(FILE,in FILE_TYPE) return BOOLEAN;
--Exceptions
STATUS_ERROR;exception renames
        IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR;exception renames
        IO_EXCEPTIONS.MODE_ERROR;
NAME_ERROR;exception renames
        IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR;exception renames
        IO_EXCEPTIONS.USE_ERROR;
DEVICE_ERROR;exception re names
        IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR;exception renames
        IO_EXCEPTIONS.END_ERROR;
DATA_ERROR;exception renames
        IO_EXCEPTIONS.DATA_ERROR;
private
```

```
-- implementation dependent  
end SEQUENTIAL_IO;
```

此程序包含有唯一的一个类属参数,该参数的类型将在使用时重新定义。注意,因为类属形参是私有型而不是限定私有型的,故而,在此程序包中的限定类型(包括任务类型),不能在使用时重新定义。从外部看,一文件具有一个名字,此名为一字符串;而从内部说,我们通过使用由 FILE_TYPE 说明的对象来访问该文件。一个打开的文件具有一个相应的值,其为枚举型 FILE_MODE。此枚举型 FILE_MODE 为文件操作模式,根据对文件是只读还是只写可取值为 IN_FILE 或 OUT_FILE。顺序文件不允许同时既读又写。文件被打开或创建时要对文件操作模式加以指定,以后亦可通过调用 RESET 过程来改变文件操作模式。对顺序文件的操作可借助预先设计好的子程序来实现。

举例,假设有一文件含有各种人口的统计值,现要对这些统计值求和。人口统计值是 INTEGER 型,文件名为 CENSUS47,所求出的和要写到名为 TOTAL47 这个新的文件中,程序如下:

```
with SEQUENTIAL_IO  
procedure COMPUTE_TOTAL_POPULATION is  
    package INTEGER_IO is new SEQUENTIAL_IO(INTEGER);  
    use INTEGER_IO;  
    DATAFILE,FILE_TYPE;  
    RESULTFILE,FILE_TYPE;  
    VALUE,INTEGER;  
    TOTAL,INTEGER:=0;  
begin  
    OPEN(DATAFILE,INFILE,"CENSUS47");  
    while not END_OF_FILE(DATAFILE) loop  
        READ(DATAFILE,VALUE);  
        TOTAL:=TOTAL+VALUE;  
    end loop;  
    CLOSE(DATAFILE);  
    --now write the result  
    CREATE(RESULTFILE,"TOTAL47");  
    WRITE(RESULTFILE,TOTAL);  
    CLOSE(RESULTFILE);  
end COMPUTE_TOTAL_POPULATION;
```

首先让我们结合这个具体的例子来说明带有实参 INTEGER 的类属程序包 SEQUENTIAL_IO,use 子句使用户可直接使用此程序包内的实体。

文件中的数据是借助对象 DATAFILE 来读的,而文件的输出借助 RESULTFILE 来实

现。DATA _ FILE 和 RESULT _ FILE 的类型均为 FILE _ TYPE, 而它们均为限定私有类型。其实现技术与第9章第3节的叙述类似, 在那里我们讨论过 key 管理者的例子。

OPEN 调用确立了 DATA _ FILE 是指外部文件 CENSUS47, 并且将 DATA _ FILE 置成只读操作模式。由于外部文件是读打开, 所以定位在起始处。

接下来执行 loop 语句直至 END _ OF _ FILE 函数指出文件已结束, 每一次循环调用 READ 都将读入的数据拷贝到 VALUE 值中, 并修改 TOTAL 值。当文件读完后, 调用 CLOSE 关闭文件。

CREATE 调用创建一名为 TOTAL47 的外部文件且将其与 RESULT _ FILE 相连。由于文件操作模式缺省, 故对此文件的操作为只写。因此, 可将前面求得的 TOTAL 值写到 RESULT _ FILE 中, 然后关闭此文件。

CREATE 和 OPEN 过程还有一个参数 FORM, 根据此参数给出的描述信息提供辅助性的操作。此参数缺省时为空串, 故可不写出。注意, 过程 CREATE 的 NAME 参数也可有一个缺省的空值, 此时其对应一个临时文件。CLOSE 过程与 DELETE 过程都可以关闭文件, 从而断开文件变量 DATA _ FILE 和 RESULT _ FILE 与外部文件的连接。文件变量可重新被其它外部文件所用。

RESET 过程可用来在文件打开后进行文件定位, 亦可用来改变文件操作模式, 如将只写改为只读。函数 MODE, NAME 和 FORM 返回相应的文件属性, 函数 IS _ OPEN 用来判定文件是否打开以及文件变量是否已经与外部文件连接。

READ 和 WRITE 过程可自动地对指定文件的文件元素进行顺序访问。

如果对文件的操作有错, 则 IO _ EXCEPTIONS 程序包中的某个异常将被引发, IO _ EXCEPTIONS 程序包如下:

```
package IO_EXCEPTIONS is
    STATUS_ERROR;exception;
    MODE_ERROR;exception;
    NAME_ERROR;exception;
    USE_ERROR;exception;
    DEVICE_ERROR;exception;
    END_ERROR;exception;
    DATA_ERROR;exception;
    LAYOUT_ERROR;exception;
end IO_EXCEPTIONS;
```

这也是一个程序包的例子, 但这个程序包不需要包体, 各异常定义在此程序包中而不是在 SEQUENTIAL _ IO 中, 使得各异常适用于 SEQUENTIAL _ IO。如果把异常分散在 SEQUENTIAL _ IO 中定义, 则必须在此程序包中针对每种情况建立异常处理段, 这需要考虑不同类型的文件处理情况, 因而特别不方便。故采用通用异常处理程序包以满足所有的异常处理。此外, 重命名说明, 可使得异常程序包不必用 IO _ EXCEPTIONS 的名。关于诸情况下引发异常的详细介绍

绍,读者可参阅语言参考手册 LRM。如果觉得顺序存取方式过于局限,可使用程序包 DIRECT _ IO,它与 SEQUENTIAL _ IO 类似且能直接存取文件中的元素。

正如以前所提到的,一个文件可以看作为一个一维数组。文件中的元素顺序排列,且有一下标,其取值范围为1到一上界值。此上界值可变,因随时可能有新元素加到文件尾。

对于直接存取的文件,设有一个现行指针,用来指明下一个要传送的元素位置。文件打开时,指针置为1,以使程序准备读或写第一个元素。顺序存取与直接存取方式的主要区别在于:顺序存取方式中,指针是隐含的,且只能通过调用 READ, WRITE 或 RESET 过程来改变其值;而在直接存取方式中,指针是显式的且可直接改变它的值。

DIRECT _ IO 的附加设施是,枚举类型 FILE _ MODE 还具有第三个值 INOUT _ FILE,其表示可对文件同时进行读一写访问,这也是创建新文件时的缺省模式。引入如下的类型和子类型:

```
type COUNT is range 0..具体实现定义;
subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;
```

来确定现行指针。还有一些附加的子程序,它们的规范式说明如下:

```
procedure READ(FILE,in FILE_TYPE,ITEM,out ELEMENT_TYPE,
              FROM,POSITIVE_COUNT);
procedure WRITE(FILE,in FILE_TYPE,ITEM,in ELEMENT_TYPE,
               TO,POSITIVE_COUNT);
procedure SET_INDEX(FILE,in FILE_TYPE,TO,in POSITIVE_COUNT);
function INDEX(FILE,in FILE_TYPE) return POSITIVE_COUNT;
function SIZE(FILE,in FILE_TYPE) return COUNT;
```

附加的重载 READ 和 WRITE 的起始位置由现行指针的值给出,而现行指针的值是由第三个参数给出的。调用函数 INDEX 返回现行指针值,调用 SET _ INDEX 以给定的位置指针,而调用 SIZE 函数,返回文件中现存元素的个数。要注意,文件中没有空隙,尽管有些元素值未定义,但从1到 SIZE 的所有元素都是存在的。现举例说明对 DIRECT _ IO 的使用。

写人口总和的值到一个已存在的文件“TOTALS”的末尾,程序的最后几句为:

```
--now write the result
OPEN(RESULT_FILE,OUT_FILE,"TOTALS");
SET_INDEX(RESULT_FILE,SIZE(RESULT_FILE)+1);
WRITE(RESULT_FILE,TOTAL);
CLOSE(RESULT_FILE);
end COMPUTE_TOTAL_POPULATION;
```

注意,如果我们所设的指针正好在文件尾以后,把人口总和数写入文件时,将导致在文件尾增加若干个未定义的元素,然后才是新近写入的内容。

还应指出,是否能以 SEQUENTIAL _ IO 写文件,然后以 DIRECT _ IO 读文件,或反之,Ada

并未说明,这类情况取决于具体执行。

练习 15.1

1. 写一个通用库过程,将一个文件以元素例排序的方式拷贝到另一个文件中,以参数的形式传递外部名。

15.2 文本输入输出

文本输入/输出在第2章中我们曾碰到过,因此比较熟悉。文本输入/输出提供两个重载过程 PUT 和 GET,用它们传递字符流。另外,还有一些其它的子程序如 NEW_LINE 用以实现格式控制等。为了避免在子程序中重复地使用文件名,这里,引入当前缺省文件的概念。即:如果 F 是当前缺省的输出文件,我们可写作为:PUT("MESSAGE")。而用不着 PUT(F,"MESSAGE");

有两个当前缺省的文件,一个对应于 OUT_FILE 输出模式,一个对应于输入模式 IN_FILE。在 TEXT_IO 程序包中不存在 IN_OUT_FILE 操作模式。

当进入我们的程序时,有两个文件被设定成标准的缺省文件且被自动地打开,它们分别与交互终端和打印机相连接。如果我们希望使用其它文件且避免在 PUT 和 GET 调用时使用文件名,则可改变当前缺省文件为我们所希望的文件。这是通过下面的子程序实现的:

```
function STANDARD_OUTPUT return FILE_TYPE;
function CURRENT_OUTPUT return FILE_TYPE;
procedure SET_OUTPUT(FILE:FILE_TYPE);
```

对于输入也有类似的子程序。函数 STANDARD_OUTPUT 返回最初缺省的输出文件,函数 CURRENT_OUTPUT 返回当前缺省的输出文件,而 SET_OUTPUT 可以改变当前缺省的输出文件。

因此,可勾画出一个程序框架为:

```
F:FILE_TYPE;
...
OPEN(F,...);
SET_OUTPUT(F);
-- 使用 PUT
SET_OUTPUT(STANDARD_OUTPUT);
```

在这里,不难知,使用 F,可将缺省文件重新设定为标准缺省文件。

更为一般的情况为,我们希望用以前的值重新设定缺省文件(亦可能在此以前值不为标准值)。由于 FILE_TYPE 是受限私有类型,故不能给其赋值,乍看起来似乎不能复制原始当前值,然而利用受限私有类型可作为子程序参数来实现重新设定缺省文件之目的。请看下面的程序

```
procedure JOB(OLD_FILE,NEW_FILE,FILE_TYPE) is
```



```
begin
    SET_OUTPUT(NEW_FILE);
    ACTION;
    SET_OUTPUT(OLD_FILE);
end;
```

然后

```
JOB(CURRENT_OUTPUT,F);
```

当调用 JOB 时,当前缺省值被保存在参数 OLD_FILE 中,通过重新调用 SET_OUTPUT 则当前缺省值又可恢复。

TEXT_IO 的全部规范式说明相当长,这里只给出其一般形式:

```
with IO_EXCEPTIONS;
package TEXT_IO is
    type FILE_TYPE is limited private;
    type FILE_MODE is (IN_FILE,OUT_FILE);
    type COUNT is range 0..由具体实现给定;
    subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;
    UNBOUNDED:constant COUNT:=0;      --行宽和页长
    subtype FIELD is INTEGER range 0..由具体实现给定
    subtype NUMBER_BASE is INTEGER range 2..16;
    type TYPE_SET is (LOWER_CASE,UPPER_CASE);
    --文件管理
    --CREATE,OPEN,CLOSE,DELETE,RESET,MODE,NAME,
    --FORM and IS_OPEN as for SEQUENTIAL_IO
    --Control of default input and out put files
    procedure SET_OUTPUT(FILE,in FILE_TYPE);
    function STANDARD_OUTPUT return FILE_TYPE;
    function CURRENT_OUTPUT return FILE_TYPE;
    --输入同上
    --行宽和页长的规范式说明
    procedure SET_LINE_LENGTH(TO,in COUNT);
    procedure SET_PAGE_LENGTH(TO,in COUNT);
    function LINE_LENGTH return COUNT;
    function PAGE_LENGTH return COUNT;
    --作为行、列和页控制的文件参数
    procedure NEW_LINE(SPACING,in POSITIVE_COUNT:=1);
    procedure SKIP_LINE(SPACING,in POSITIVE_COUNT:=1);
```



```
function END_OF_LINE return BOOLEAN;
procedure NEW_PAGE;
procedure SKIP_PAGE;
function END_OF_PAGE return BOOLEAN;
function END_OF_FILE return BOOLEAN;
procedure SET_COL(TO;in POSITIVE_COUNT);
procedure SET_LINE(TO;in POSITIVE_COUNT);
function COL return POSITIVE_COUNT;
function LINE return POSITIVE_COUNT;
function PAGE return POSITIVE_COUNT;
--作为字符输入_输出文件参数
procedure GET(FILE;in FILE_TYPE;ITEM;out CHARACTER);
procedure GET(ITEM;out CHARACTER);
procedure PUT(FILE;in FILE_TYPE;ITEM;in CHARACTER);
procedure PUT(ITEM;in CHARACTER);
--字符串输入_输出
procedure GET(ITEM;out STRING);
procedure PUT(ITEM;in STRING);
procedure GET_LINE(ITEM;out STRING;
LAST;out NATURAL);
procedure PUT_LINE(ITEM;in STRING);
--整型输入_输出的类属程序包
generic
    type NUM is range<>;
package INTEGER_IO is
    DEFAULT_WIDTH;FIELD:=NUM'WIDTH;
    DEFAULT_BASE;NUMBER_BASE:=10;
procedure GET(ITEM;out NUM;WIDTH;in FIELD:=0);
procedure PUT(ITEM;in NUM;
    WIDTH;in FIELD:=DEFAULT_WIDTH;
    BASE;in NUMBER_BASE:=DEFAULT_BASE);
procedure GET(FROM;in STRING;ITEM;out NUM;
    LAST;out POSITIVE);
procedure PUT(TO;out STRING;
    ITEM;in NUM;
    BASE;IN_NUMBER_BASE:=DEFAULT_BASE);
end INTEGER_IO;
```



--浮点型输入_输出的类属程序包

```
generic
    type NUM is digits<>;
package FLOAT_IO is
    DEFAULT_FORE:FIELD:=2;
    DEFAULT_AFT:FIELD:=NUM'DIGITS-1;
    DEFAULT_EXP:FIELD:=3;
    procedure GET(ITEM out NUM;WIDTH,in FIELD:=0);
    procedure PUT(ITEM,in NUM;
        FORE,in FIELD:=DEFAULT_FORE;
        AFT,in FIELD:=DEFAULT_AFT;
        EXP,in FIELD:=DEFAULT_EXP);

    procedure GET(FROM,in STRING;
        ITEM,in NUM;
        LAST,out POSITIVE);
    procedure PUT(TO,out STRING;
        ITEM,in NUM;
        AFT,in FIELD:=DEFAULT_AFT;
        EXP,in FIELD:=DEFAULT_EXP);
end FLOAT_IO;
```

--定点型输入输出的类属程序包

```
generic
    type NUM is delta<>;
package FIXED_IO is
    DEFAULT_FORE:FIELD:=NUM'FORE;
    DEFAULT_AFT:FIELD:=NUM'AFT;
    DEFAULT_EXP:FIELD:=0;
    --同浮点型输入_输出
end FIXED_IO;
```

--枚举类型输入_输出的类属程序包

```
generic
    type ENUM is(<>);
package ENUMERATION_IO is
    DEFAULT_WIDTH:FIELD:=0;
    DEFAULT_SETTING:TYPE_SET:=UPPER_CASE;
    procedure GET(ITEM,out ENUM);
    procedure PUT(ITEM,in ENUM);
```



```
    WIDTH : in  FIELD := DEFAULT_WIDTH;
    SET : in  TYPE_SET := DEFAULT_SETTING);
procedure GET(FROM : in  STRING;
    ITEM : out  ENUM;
    LAST : out  POSITIVE);
procedure PUT(TO : out  STRING;
    ITEM : in  ENUM;
    SET : in  TYPE_SET := DEFAULT_SETTING);
end ENUMERATION_IO;
-- 异常
STATUS_ERROR : exception renames
    IO_EXCEPTION.STATUS_ERROR;
...
LAYOUT_ERROR : exception renames
    IO_EXCEPTIONS.LAYOUT_ERROR;
private
    -- 由具体实现而定
end TEXT_IO;
```

类型 FILE_TYPE 和 FILE_MODE 以及各种文件管理过程都与 SEQUENTIAL_IO 类似，因为文本文件本身就是一种顺序文件。

PUT 和 GET 过程用于字符以及字符串两种形式的输入输出。字符形式对应于文件的输入输出而字符串形式对应于其它(I/O 设备)。

在 CHARACTER 类型的情况下，PUT 调用只输出字符串，而在 STRING 类型下，PUT 调用输出字符串。

由于没有设定数值类型或枚举类型，故对这两种类型的数值进行 PUT 或 GET 则会产生错误。为克服此不足，通过提供一通用类属程序包来满足每种类型。我们举例说明借助程序包 INTEGER_IO 来进行整数的输入—输出。INTEGER_IO 的应用类型为：

```
package MY_INTEGER_IO is new INTEGER_IO(MY_INTEGER); use MY_INTEGER;
```

对于整数输出，PUT 有三种形式，一种对应于文件，一种不是对文件输出(对输出设备等)，还有一种为整数字符串的形式，仅后两种可显示。

在用 PUT 输出文件时，有二种格式参数 WIDTH 和 BASE，此两参数缺省时由变量 DEFAULT_WIDTH 和 DEFAULT_BASE 给出它们的值。缺省宽度的初值为 NUM'WIDTH，其给出了一个最小域，而该域是以 10 为基数的。可通过给变量 DEFAULT_WIDTH 和 DEFAULT_BASE 直接赋值来改这两个缺省值。值得注意的是，每次用到缺省参数时都要重新计算，因此所得到的缺省值总是变量的当前值。整数输出是以整数字符的形式设有底线且没有前导 0，但如果是一个负数则前面用一减号加以表示。实际输出的整数长度小于定义的域宽时，左边填以

空格;如果域宽太小,则必须对其进行扩展。缺省值为0时是最小域宽。以10为基的数不论显式或缺省输出,都采用十进制数的词法,否则采用相应带基数的词法表示形式。

在用 PUT 输出字符串的情况下,域宽就是字符串长。如果域宽太小,则引发 LAYOUT_ERROR 异常。

类似的技术也适用于实型数,其输出也是一种设有底线和前导0的十进制数字。如果是负数,可带一个负号,如果 EXP 是零则没有指数,在小数点之前有 FORE 个数字,在小数点之后有 AFT 个数字。如果 EXP 不是零,则在字符 E 后有 EXP 个数字输出。如果有必要,可在小数点之前只设一个有效数字(此为实型数的标准形式),如若输出格式中的 FORE 和 EXP 部分的域宽不合适,可进行扩展。实型数仅为十进制形式,其值的精度取决于 AFT 长度。

对于浮点数最初的缺省格式参数是 2,NUM'DIGITS-1 和 3;由此描述的浮点数形式为:在小数点之前,有一个空格或负号加上一个数字,在小数点之后有 NUM'DIGITS-1 个数字和一个二位指数。对于定点类型的数其相应参数是 NUM'FORE,NOM'AFT 和 0,由这些参数给出非指数的形式以及定点数相应精度。

枚举类型亦采用同样的方法加以表示,用0表示域缺省,如果域必须被填满,则在数值后附加空格而不象数值类型将空格加在前面。通常使用大写字符,当然亦可使用小写字符。字符类型的值为字符文字,输出时需用单引号将其括起来。请注意,CHARACTER 类型和枚举值所定义的 PUT 之间的细微差别。

```
TEXT_TO.PUT('X');
```

输出单个字符 X,而

```
package CHAR_IO is
  new TEXT_IO.ENUMERATION_IO(CHARACTER);
CHAR_IO.PUT('X');
```

输出单引号间的字符 X。

在用 GET 过程实现输入时,总是跳过行结束符或页结束符。在 CHARACTER 类型的情况下,要读的总是下一个字符。在 STRING 类型的情况下,读入若干个字符,字符个数由实参指明。对于枚举类型,前导空白(空格或制表符)也被跳过,输入结束在不属于枚举值一部分的字符处或行终止符处。数值类型的输入也与上述类似,但有一个附加的可选参数 WIDTH,如果 WIDTH 有一个非0值,则读完 WIDTH 个字符(包括跳过的空白)后便停止。GET 读取的对象为一字符串而不是文件,LAST 的值指明所读的最后一个字符的位置,串结束相当于文件结束。如果数据项形式不正确,则引发 DATA_ERROR 异常。

正文文件可认为是由一系列的行所组成。一行中的字符都有一个列位置,起始列为1。输出时行长可固定亦可变化。固定行长适用于输出表格,而可变行长适用于交互的情况。在一个文件中,行长可变,且最初亦不被固定。多个行组成页。

调用 PUT 输出时,是将字符沿着现行行当前位置开始起顺序输出。如果行长固定且当前行所剩部分放不下字符时,则另起一行,所有字符将写在新行中,若新行仍不够写入字符则引发 LAYOUT_ERROR 异常。如果行长不固定,字符总是继续写到现行行尾。

格式可由各子程序来控制。在某些情况下,这些子程序既可用于输入文件又可用于输出文

件。如果忽略了指出文件，此时格式控制程序总是应用于输出情况。多数情况，一个子程序只对应于输入输出中的一种情形，这样，便可很自然地省略掉文件而给出缺省值。

COL 函数返回指定行中的现行位置，SET_COL 过程以给定的值置当前位置。但 SET_COL 调用不能实现位置退回。输出时，计数位置包括附加的空格，而输入时跳过空格。当 SET_COL 的参数等于 COL 的当前值，则 SET_COL 调用无效。如果 SET_COL 的参数大于 COL 的当前值则相当于执行了 NEW_LINE 或 SKIP_LINE。

过程 NEW_LINE(仅用于输出)输出指定数目的新行(缺省时为1)且现行列为1，一行后面的多余位置用空格填入。过程 SKIP_LINE(仅用于输出)同样移动给定的行数(缺省时亦为1)且新行的列位置为移动前的当前位置，如果已到了一行的末尾则函数 END_OF_LINE(仅用于输入)返回 TRUE。

函数 LINE_LENGTH(仅用于输入)在行长固定的情况下返回现行行的长度而行长不固定时返回 0。过程 SET_LINE_LENGTH(仅用于输出)以给定的值置行的固定长度，当给定的值为 0 则表示行长不固定。

用于控制页中的行还有一些功能类似的子程序。它们为 LINE, SET_LINE, NEW_PAGE, SKIP_PAGE, END_OF_PAGE, PAGE_LENGTH 和 SET_PAGE_LENGTH。最后函数 PAGE 返回从文件始至现行 PAGE 的 PAGE 数，没有 SET_PAGE 函数。

过程 PUT_LINE 和 GET_LINE 专门用于固定长度的行，调用 PUT_LINE 输出字符串并移至下一行。调用 GET_LINE 可连续地读字符到字符串中直至串结束或遇到行结束为止，然后移至下一行。LAST 指出送到字符串中的最后一个字符。因此，逐次地调用 PUT_LINE 和 GET_LINE 可对所有的行进行操作。

程序包 TEXT_IO 看起来稍微有些复杂，但对多数场合，我们只需使用 PUT 和 NEW_LINE。

练习 15.2

1. 以缺省参数作为初值，下列调用输出什么?
 - a) PUT("FRED");
 - b) PUT(120);
 - c) PUT(120,8);
 - d) PUT(120,8,8);
 - e) PUT(-38.0);
 - f) PUT(0.07,6,3,1);

假定实型值其 digits=6，整型值为 16 个 bit 位。

15.3 中断

中断是另一种形式的输入。在 Ada 中，中断是通过汇合机制来实现的。从程序的角度看，中断由一个优先级比本程序任何一任务都要高的外部任务调用入口而产生的，而中断处理程序则是被调任务中对应于入口的接收语句。利用地址子句可给出入口的相应物理地址。

举例加以说明，假定程序希望由电子开关的闭合引发中断，中断地址为 8#72#，则我们可写为：

```
task CLNTACT_HANDLER is
    entry CONTACT;
```



```
for CONTACTUSE use at 8#72#;
end;
task body CONTACT_HANDLER is
begin
loop
accept CONTACT do
...
end;
end loop;
end CONTACT_HANDLER;
```

accept 语句体为中断处理部分,由于规定外部虚拟任务的优先级比任何普通任务的优先级都高,所以保证了虚拟任务可越过普通任务而随时被处理。

中断入口通常没有参数,但亦可设有 in 参数,将控制信息传递给它。

对应于中断的 accept 语句也可出现在选择语句之中。中断入口的详细特征取决于中断处理过程。它们亦可作为条件入口而出现,如果中断响应任务没有准备好执行相应的 accept 语句,则引发中断的请求将被丢失。以地址子句描述的中断地址也取决于中断处理过程。

15.4 表示子句

表示子句又称子句,作为向编译程序告之诸如中断入口等附加信息的手段来使用。表示子句取多种形式并用于不同的目的。其最为常用的情况是向编译程序指出某机构将如何处理。地址表示子句(又称地址子句,下面类同)用于实体。它能够用来给对象分配一明确的地址,以指明一子程序、程序包或任务的程序代码的起始地址,或者正如我们已经看到的那样指明对应某中断的入口地址。

长度子句允许我们将指定的存贮空间分配给目标类型。如指明存取类型的聚集,任务类型的工作存贮空间。这是借助指明值的某种属性来实现的。因而

```
type BYTE is range 0..255;
for BYTE'SIZE use 8;
```

规定了类型 BYTE 的对象每个占 8 位(二进制)。

聚集和任务的存取空间将用属性 STORAGE_SIZE 来指明。在此种情况中,单位为存贮单位而不再是二进制位。存贮单位的二进制位数量由 SYSTEM 程序包中的常量 STORAGE_UNIT 给出。因此,当我们想确保足够的存取聚集,用长度子句:

```
type LIST is access CELL;
for LIST'STORAGE_SIZE use 500 * (CELL'SIZE/SYSTEM.STORAGE_UNIT);
```

为作为 LIST 值的聚集保留了 500 个 CELL 类型的存贮空间。同理,以下面的长度子句,可为一任务类型的每个任务指定数据空间。

```
for MAIL__BOX' STORAGE_SIZE use 128;
```

用长度子句也能指定定点类型的部分值。

枚举子句能够用来表示枚举类型直接量的聚集、内部代码等。我们可将一状态值转换成确定的二进制数的形式。

```
type STATUS is(OFF,READY,ON);  
for STATUS use(OFF=>1,READY=>2,ON=>4);
```

不难知,不同的枚举直接量对应不同的整数值,诸整数值的排序与对应枚举直接量的逻辑次序相同。虽然并未要求诸数值的大小必须一个紧挨一个,而属性 SUCC,PRED,POS,VAL 总是按逻辑次序处理的。最后,还有一种表示子句,被用来指出记录类型的结构。如果我们用

```
type REGISTER is range 0..15;  
type OPCODE is(...);  
type RR is  
record  
    CODE:OPCODE;  
    R1:REGISTER;  
    R2:REGISTER;  
end record;
```

表示 IBM370 系列计算机寄存器型(RR 型)的机器指令格式,那么我们可以用描述记录类型结构的记录子句给出此指令格式的确切映象。

```
for RR use  
record at mod 2;  
    CODE at 0 range 0..7;  
    R1 at 1 range 0..3;  
    R2 at 1 range 4..7;  
end record;
```

对齐子句

```
at mod 2;
```

表示该记录以双字节边界对齐,也就是说,记录在内存中的存放地址末位总为0。记录中各元素的位置和大小与记录的起始地址有关,它们分别由 at 后面的值加以指出。at 后面的值指出了对象(at 前的元素名)所在的存储单位和以 bits 个数决定的范围。bit 位数超出存储单位的情况是允许的。如:

```
R1 at 0 range 8..11;
```

如果不指明每个元素的位置,则编译系统以其最佳的形式加以安排,除非这些元素处于供选择的变量中,一般均需要给出足够的空间供编译系统指明元素位置和大小,以保证元素间不重叠。

最后有一点需要注意,表示子句必须与其所涉及的机构出现在同一个说明中。

15.5 实现考虑

由于要涉及具体的实现过程,故在这方面进行详细叙述是困难的。在程序包 SYSTEM 中,包含了许多与机器有关的常量,其规范式说明如下所述:

```
package SYSTEM is
    type ADDRESS is implementation_defined;
    type NAME is implementation_defined_enumeration_type;
    SYSTEM_NAME;constant NAME:=implementation_defined;
    STORAGE_UNIT;constant:=implementation_defined;
    MEMORY_SIZE;constant:=implementation_defined;
    --system_dependent name dnumbers;
    MIN_INT;constant:=implementation_defined;
    MAX_INT;constant:=implementation_defined;
    MAX_DIGITS;constant:=implementation_defined;
    MAX_MANTISSA;constant:=implementation_defined;
    FINE_DELTA;constant:=implementation_defined;
    TICK;constant:=implementation_defined;
    --other system_dependent declarations
    subtype PRIORITY is INTEGER range implementation_defined;
    ...
end SYSTEM;
```

类型 ADDRESS 用在地址子句中,由相应的属性给出。

STORAGE_UNIT 和 MEMORY_SIZE 常量给出以 bit 计的存贮单位的大小和以存贮单位计的存贮器的容量,它们都是通用整型。MIN_INT 和 MAX_INT 给出整型的最小值和最大值(注:最小值指绝对值最大的负数)。MAX_DIGITS 是浮点类型中所允许的最大十进制有效数字个数,而 MAX_MANTISSA 是定点类型中允许的最大二进制数字个数。它们全都是通用整型。

各编译注解(pragmas)可用来为实现过程设置某些参数,而属性让我们去读某些参数的值。预定义的编译注解和属性由附录给出,其中一些内容可随意增加。

某些编译注解使我们能够控制编译系统,求得完整性和有效性间的平衡以及存贮空间和处理时间之间的协调。

编译注解 SUPPRESS 用来指明对发生在运行中导致异常引发的状态的检测可忽略,从而得到更为有效的程序。然而,必须清楚:此编译注解仅保证不处理异常而不保证不发生异常,事实上异常总会从忽略检测的程序单位传播开来。

与异常 CONSTRAINT_ERROR 有关的检测是 ACCESS_CHECK(当试图访问一分量时检

测该存取值不为空),DISCRIMINANT _ CHECK(检测正读取的分量与识别值一致或受约束),INDEX _ CHECK(检测下标处在范围内),LENGTH _ CHECK(检测数组元素的个数)以及RANGE _ CHECK(检测是否满足所有约束)。

与 NUMERIC _ ERROR 有关的检测是 DIVISION _ CHECK(检测 l, rem 和 mod 的第二个操作分量)和 OVERFLOW _ CHECK(检测数值溢出)。

与 STORAGE _ ERROR 有关的检测是 STORAGE _ CHECK(检测存取聚集或任务的空间是否超过)。

与 PROGRAM _ ERROR 有关的检测是 ELABORATION _ CHECK(检测程序单位的体是否详尽描述出)。

编译注解的格式为:

```
pragma SUPPRESS(RANGE _ CHECK);
```

适用于程序单位所牵涉到的所有操作,或者编译注解中列出其适用的类型和对象,如:

```
pragma SUPPRESS(ACCESS _ CHECK,LIST);
```

表明对于存取类型 LIST 不提供 ACCESS _ CHECK 检测。

用于存贮空间和处理时间之间进行协调之类的编译注解为 CONTROLLED,INLINE,OPTIMIZE 和 PACK,它们在附录中给出了描述。

最后,有一些针对机器的实际类型的属性。例如属性 MACHINE _ ROUNDS 表示对所涉及的类型是否做四舍五入处理。另一个属性 MACHINE _ OVERFLOWS 表示计算超出类型范围是否引发 NUMERIC _ ERROR。如果编译注解用于这些属性又要保证移植性时则需十分小心。

15.6 无检查的程序设计

有些时候,一致性过强的语言在使用上反倒不方便。特别是在系统程序的设计中。在系统程序中,一对象在程序的不同部分被看成不同的项。对此在 Ada 中可用称之为 UNCHECKED _ CONVERSION 的类属函数来解决,该类属函数的规范式说明为:

```
generic
  type SOURCE is limited private;
  type TARGET is limited private;
  function UNCHECKED _ CONVERSION(S:SOURCE) return TARGET;
```

例如,15.4节中的类型 STATUS 的值,在我们的程序中可取类型 BYTE 的值。为了实现这种转换,我们首先实例化类属为:

```
function BYTE _ TO _ STATUS is
  new UNCHECKED _ CONVERSION(BYTE,STATUS);
```

然后写

```
B:BYTE:=...
S:STATUS;
```

...
S:BYTE_TO_STATUS(B);

无检查的转换实际上不做任何事。原类型的位图只是不加改变地看作目标类型的位图。很显然，这样做保证了某些条件必定是满足的，尤其对实现过程产生影响的那些位图中的二进制位两类型必然是相同的。我们不能做原本做不到的事。

另一个给程序员增设灵活性的领域是使程序员能重新定位存取类型。在11.3节中我们提到，系统没有无用单元收集器，但如果自己对释放存贮单元进行收集，不如我们在实时程序中没进行分时控制更好。为程序员提供的免于检查的存贮单元释放，依赖于称之为UNCHECKED__DEALLOCATION的类属过程，其规范式说明为：

```
generic
    type OBJECT is limited private;
    type NAME is access OBJECT;
procedure UNCHECKED_DEALLOCATION(X:in out NAME);
```

例如

```
type LIST is access CELL;
```

我们写为

```
procedure FREE is new UNCHECKED_DEALLOCATION(CELL,LIST);
```

则

```
L:LIST;
```

```
...
```

```
FREE(L);
```

调用FREE后，L的值则为null，存贮单元为自由单元。显然，如果在把这些单元指派给另一变量前，一旦使用不当则发生混乱，程序也将出错。



第16章 总 结

作为最后一章,我们将全面地总结 Ada。本章的前三节,总结了本书各章介绍的主要内容,通过归纳和总结来展示 Ada 的全貌以及 Ada 各部分间的基本关系。有一节专门讨论可移植性问题,最后我们还将讨论用 Ada 进行程序设计所涉及到的一般问题。

16.1 名与表达式

我们须把名这个概念与标识符区分开来。标识符是一语法形式,如 FRED,它有许多种用途,包括在说明时由其引入实体。而名则比较复杂,它是用来表示实体的一般形式。在 LRM 和实用的语法规则中,单词“名”指的就是标识符。

语法上,名以标识符(如 FRED)或以一运算符(如“+”)开头,后尾随下面的一项或几项(次序任意):

- 在括号中的一个或多个下标表达式。这表示一个数组元素。
- 在括号中的离散范围。这表示一个数组片(即数组的一部分)。
- 一个点号后跟标识符,运算符或 all。这表示一个记录元素,一个存取值或在程序包、任务、子程序、分程序或循环中的一实体。
- 一个单引号“,”和一标识符(亦可能为指引元),这表示一个约束。
- 在括号中的实参表。这表示函数调用。

名被用于函数调用场合时,须给出一数组,记录或存取值且在其后必须跟有下标、分片、约束或元素选择作为函数调用参数。但这不意味着在函数调用时都必须跟有上述成份之一。亦可用表达式的形式提供上述各成份的值。如果是作为名的一部分,则必须跟有这些成份。让我们来看一个赋值语句的例子,便不难弄清此问题。赋值语句左边必须是名,而右边则是表达式。如在11.5节中我们曾看到过的有:

SPOUSE(P).BIRTH := NEWDATE; 而非

SPOUSE(P); = Q;

尽管

Q; := SPOUSE(P);

是完全合法的(Ada 在这方面不如 Algol68 有很好的一致性)。

名只是表达式中的一种基本分量。而表达式中的其它基本分量是直接量(数值的、枚举的、字符串的直接量以及 null)、聚集、分配符、函数调用(不是作为名)、类型转换以及约束表示(如同在括号中的表达式一样)。为了讨论的简便,在表16.1中我们给出了所有包含在表达式中的运算符以及它们的预定义含义。需记住,在表达式中还可以出现“短路”形式, and then 和 or else 以及成员关系测试 in 和 not in。从技术上说,它们不属于运算符。

操作符	运算分量
and	or xor Boolean

结果
same

one dim Boolean array	same
$=/=/$ any, not limited	BOOLEAN
$<<=>>$ scalar	BOOLEAN
one dim discrete array	BOOLEAN
$+-(binary)$ numeric	same
$\&$ one dim array	same
$+-(unary)$ numeric	same
* integer integer	same
fixed INTEGER	same fixed
INTEGER fixed	same fixed
fixed fixed	univ fixed
floating floating	same
integer integer	same
fixed INTEGER	same fixed
fixed fixed	univ fixed
floating floating	same
* * integer non neg INTEGER	same integer
floating INTEGER	same floating
not Boolean	same
one dim Boolean array	same
abs numeric	same

表16.1 预定义的运算符

细心的读者会注意到,在附录的语法中,有些原本用“表达式”的地方,却只用了“简单表达式”的语法形式。其原因之一是回避在语法描述上仍要考虑用 in 对范围内成员进行测试的潜在可能性。

我们常看到,有时表达式需要是静态的。在第2章我们已解释了原因,使用静态表达式是使得在编译时便可计算出它们的值。若构成表达式的所有分量是下列形式之一,则该表达式是静态的:

- 数值直接量或枚举直接量。
- 命名数据。
- 由静态表达式赋初值的常量。
- 静态属性或带有静态参数的函数属性。
- 受约束的静态表达式且所有约束都是静态的。

关于表达式我们想要说明的最后一点是数组的下标界限。若表达式的值是一个数组值,则此表达式给出了每维的下标界限。根据下标界限的匹配原则,可根据使用场合把这种表达式分为两类。第一类,用在说明中对被说明的对象初始化和用在赋值语句中对变量赋值。此时,表达式的下标界限不必与对象的下标界限匹配,只需各维中的元素个数相同即可。(如我们在6.2所见)如:

```
V;VECTOR(1..5);  
W;VECTOR(0..4);  
...  
V:=W;
```

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

此匹配规则也适用于预定义的等同和关系运算。

第二类，用数组作为实参、函数结果、分配符中的初始量以及限定表达式中的数组。在所有这些情况中，均涉及到数组的类型和子类型。然而类型可受约束亦可不受约束。若受约束则下标界限必须完全匹配，否则“结果”从表达式中得到其下标界限。因此，如有函数：

```
function F return VECTOR_5 is  
  V;VECTOR(1..5);  
  W;VECTOR(1..4);  
begin  
  ...
```

则可以写成 `return V`，但不能写成 `return W`。

然而，如果函数的规范式说明为：

```
function F return VECTOR;
```

则可以写成 `return V` 或者 `return W`。数组的聚集比较复杂。不含 `others` 的带名聚集，其下标界限是明显的，这样的聚集可用于任何场合。不含 `others` 的位置聚集，尽管元素个数确定，但下标界限不明显。即便如此，这样的聚集也能用于上述的任何场合。对于第一类和第二类受约束类型，下标界限由约束将其确定。对于第二类无约束类型，下标界限由 `S'FIRST` 给出，这里的 `S` 是下标子类型。若聚集（位置聚集或带名）中含有 `others`，则下标界限及元素个数都是不可知的，这种聚集可在第二类受约束类型的情况下使用。此外含有 `others` 的位置聚集亦可用于第二类情况下。

对于嵌套聚集，也有类似的规则。只是在嵌套聚集中，把内层聚集看作为外层聚集的一个元素。

需要注意的是，8.2节中介绍的数组类型转换不能简单地套用上述二类的情况。如果类型是受约束的，则要遵循赋值语句的匹配规则；如果类型是不受约束的，则把运算分量的下标界限作为约束。

练习 16.1

1. 给定

```
L;INTEGER:=6;  
M;constant;INTEGER:=7;  
N;constant:=8;
```

指出下列表达式是静态的，还是动态的，并指出它们的类型。

- a) $L + 1$
- b) $M + 1$

16.2 类型等效



有关类型等效的规则值得在此加以强调。

基本规则是，每进行一次类型定义，便引入了一个新类型。注意，类型定义与类型说明是不同的。类型定义用来引入类型，而类型说明只是用来引入标识此类型的标识符。因此

```
type T is(A,B,C);
```

是类型说明而

```
(A,B,C);
```

是类型定义。

绝大多数类型有名。但在有些情况下，类型可以匿名。这类情况常出现在数组和任务的说明中。如

```
A;array(I range L..R) of C;
```

是下式的缩写

```
type anon is array(I range<>) of C;
```

```
A;anon(L..R);
```

而

```
task T is...
```

又是下式的缩写

```
task type anon is...
```

```
T;anon;
```

较难于理解的情况是，在显式类型说明中仅为子类型说明。这常出现在数组类型、派生类型和数值类型中。例如：

```
type T is array(I range L..R) of C;
```

是下列的缩写

```
subtype index is I range L..R;
```

```
type anon is array(index range<>) of C;
```

```
subtype T is anon(L..R);
```

而

```
type S is new T constraint;
```

是下列的缩写

```
type anon is new T;
```

```
subtype S is anon constraint;
```

而最后

```
type T is range L..R;
```

是下列的缩写

```
type anon is new integer_type;
```

```
subtype T is anon range L..R;
```

其中 integer_type 是一预定义整型。

在阐明了每个类型定义都引入一个新类型的规则时,应当记住:类属实例等价于文本替换,因此具有类型定义的程序包的每个实例均在其规范式说明中引入一个特定的类型。还应记住:对于不同异常的标志也有类似的规则,每个原文不同的异常说明都引入一个新的异常。在递归过程中的异常每次递归都是相同的,但类属实例则引入不同的异常。

注意,同时对多个对象的说明,等价于多个单独的说明。例如

A,B;array(I range L..R)of C;

则 A 和 B 是不同的匿名类型。

16.3 结构小结

Ada 有四种含说明的结构单位。它们是分程序、子程序、程序包和任务。可以按多种不同的方法对它们进行分类。所有的程序包和任务都分别有规范式说明和体。对子程序来讲规范式说明可有可无,而分程序没有规范式说明。就分别编译而言,程序包、任务和子程序体都可作为独立的子单位,但只有程序包和子程序可作为库单位。另外,只有程序包和子程序可以是类属的。最后,任务、子程序和分程序可以有从属任务,而程序包则不能有,因为程序包只是被动的范围控制单位。现将这些结构单位的特性总结如图16.2

特 性	分程序	子程序	程序包	任 务
分别编译	no	可 选	yes	yes
子单位	no	yes	yes	yes
库单位	no	yes	yes	no
类属设施	no	yes	yes	no
从属任务	yes	yes	no	yes

图16.2 结构单位的特性

我们再来从作用域和嵌套的角度观察这四种结构单位。(注意,一个分程序看作为一语句,而其它结构单位看作为说明)。每个结构单位均可嵌套在其它结构单位中。理论上讲,嵌套可以是无限层的。而实际上很少超过三层。唯一对嵌套加以限制的是语句,即分程序。分程序不能出现在程序包的规范式说明中,而只能出现在包体之中。显然这些结构单位亦不可出现在任务的规范式说明中,只能出现在任务体之中。然而,实际上这种混合难得出现。分程序通常出现在子程序和任务体中,偶尔出现在其它分程序中。子程序往往作为库单位出现在程序包中,很少出现在任务和其它子程序中。程序包通常作为库单位或出现在其它程序包中。任务一般是在程序包中,只是偶尔出现在其它任务或子程序中。

Ada 的语言参考手册,关于程序这个概念有点含糊不清。这或许是所希望的,因为 Ada 强调的是软件元件,因而过分地注重一个完整程序的组成是不妥当的。尤其对于牵涉到分布式和并发系统更是如此。但是,对于一个简单的系统,我们可以把程序看作是由程序库中各单位的组合而形成的。LRM 没有描述程序是怎样启动的,正象2.2节和8.2节中介绍的那样,我们可以想象为由语言自身之外的不可思议的东西调用库单位(即子程序)来导致程序的启动。进一步,

我们须想象到最初的控制流与某个匿名任务有关,该匿名任务的优先级可由此主程序的最外层的说明所设定。

主程序必然要用到类属库单位,如 TEXT_IO。这些类属库单位,必须在进入主程序前已被确立。我们可想象为,这些工作是由匿名任务来完成的。没有指出这些类属库单位的确立顺序,但确立顺序须与库单位间的关系一致。此外,可用 ELABORATE 编译注释来确保在调用库单位前,确立该单位的体。由此避免了在程序运行时引发 PROGRAM_ERROR。如果对类属库单位的确立顺序与库单位间的关系不一致则程序是非法的。如果存在着几种可能的顺序而程序只依据某一特定的顺序执行,则此程序也是非法的。

主程序可否带参,以及参数的类型及存取方式(指 in, in out 和 out)存在何限制,主程序可否为一个函数等问题。这些都取决于语言的实现。由于 Ada 属于开放性的单位式语言,不知主程序带参非常方便。对主程序的调用和参数传递,需借助非 Ada 的命令来实现。

本节要指出的最后一点是数学库。LRM 没有象描述输入和输出程序包那样描述数学库。我们不妨认为存在着如 4.9 节和练习 2.2(1) 中那样的数学库。在数学库中要解决的一个问题是用户定义的浮点类型所牵涉到的派生性质。派生类型的法则是,任何程序库中预定义类型的性质(如 FLOAT)不能自动地由派生类型(如 REAL)所继承。因此数学库最好用类属程序包来构造。

```
generic
  type F is digit<>;
package MATHS_LIB is
  function SQRT(X;F) return F;
  function LOG(X;F) return F;
  ...
end MATHS_LIB;
```

这样我们就可以用 REAL 类型来使用数学库,写作:

```
package REAL_MATHS_LIB is new MATHS_LIB(REAL);
use REAL_MATHS_LIB;
```

16.4 可移植性

一个 Ada 程序可被移植或者不可被移植。在多数情况下,程序与其运行的特定硬件密切相关。在嵌入式应用系统中尤其如此。对于与硬件密切相关的程序,只有经过很大的改动后才可能在其它机器上运行。因此人们特别希望写出可移植的程序库,以能够在不同的应用中反复地使用它们。在有些情形下,程序库中的成份是完全可移植的。但更多的情况是,程序库中的成份在移植时对机器有一定的要求。可采用设置参数的办法,通过参数调整使程序适应其运行环境。本节的内容在于指导程序员如何编制可移植的 Ada 程序。

首先应当避免编制含有错误的程序以及避免编制那种具有“依从关系”错误的程序。有些错误很可能是隐蔽的。有的程序在某特定的机器上可能运行很正常,且从表面上看也是可移植

的。然而,如果该程序恰巧是因为某些没有定义的特征而使它在一特定机器上运行正常,就很难保证它在另一个机器上工作仍然正常。最常见的例子是,程序中变量没有赋初值。通常未赋初值的变量其初值为0,这是因操作系统在装入程序前先清除了程序区。在这样的环境中,程序运行是正常的。但如果把程序移植到其它不同的机器上,很难保证在程序装入前仍然清除程序区,因而程序的运行就有可能出现令人吃惊的结果。在前几章中,我们曾经介绍过各种导致非法程序的情况,现归纳如下。

最重要的是表达式的计算顺序。因为表达式中可含函数调用,而函数调用有可能会产生副作用。这样,表达式的计算顺序不同,结果亦可能不同。下列的计算顺序是不确定的:

- 一个二元运算的各操作分量。
- 在赋值语句中目标与值。
- 聚集中的各个分量。
- 子程序和入口中的各个参数。
- 多维数组中的各个下标表达式。
- 范围中的各个表达式。
- 选择语句中的各保护条件。

导致有害值的主要情况有:

- 在给一个无初值变量赋值前就引用该变量。

在语言机制中有两种情况不确定:

- 数组、记录和私有参数的传递。
- 在选择语句中分枝选择算法。

在有些场合程序员可不用类型的强制方式以增加自由度,但在应该用类型强制方式但没有使用的场合可能引起程序错误。例如:

- 隐蔽异常。
- 无检查的存贮单元分配。
- 无检查的转换。

最后我们还应注意,上节中提到程序单位的确立顺序不确定的情形。

数值类型与可移植性也有密切的关系。原因在于既要使程序在所有机器上正常运行又要保证最大效率。Ada 利用样板数的概念实现折衷。原理上,如果我们遵循样板数的特性,则程序是可移植的。但实际上,这是不易做到的。

若能正确地使用各种属性,则可使程序有较强的可移植性。我们可用 BASE 找出基类型,用 MACHING _ OVERFLOW 查看是否会引起 NUMERIC _ ERROR。但错用这些属性反而会使程序可移植性变的特别差。

有一条简单的原则可依循,这就是,我们应定义自己需用的实型,而不是直接引用预定义类型(如FLOAT)。对整型也是如此,但Ada语言对预定义类型INTEGER 设置的范围是合理的。

关于可移植性还要考虑的方面是任务。程序中凡使用任务都将对可移植性带来损害。因为不同种类的机器执行指令的速度各不相同,由此引起任务的执行速度在不同种类的机器上亦不相同。有时,某个程序根本不能在某个特定的机器上运行,因为该机器不具有处理这一程序的能力。虽然我们无法给出硬指标,但要记住如下几点:

对应用来说,类型 DURATION 的精度已足够了。但如果定时循环时间间隔不采用样板数,则难以按要求结束。

尽量避免非同步地使用共享变量。有时,采用分时访问的办法亦难于满足速度要求且在技术上笨拙;因此尽可能采用同步机制来实现。最简单的办法是用编译注释 SHARED 来防止任务间的相互干扰。

夭折语句 (abort 语句) 的使用将会给可移植性带来问题,因为夭折语句具有异步性质。应避免过多地使用优先级,当你用优先级求得对程序的及时响应时,你的程序也许又要求机器资源。

机器的速度和可用空间均是有限的。不同种类机器的内存空间不同,因此也许一个程序在某类机器上运行得很好而到了另一类机器上便有可能产生 STORAGE_ERROR。此外,不同的 Ada 的实现系统对存取类型和任务的数据所采用的分配策略不尽相同。我们已知,用表示子句可对存贮分配进行控制从而降低了可移植性。因此最好避免使用存取类型。

练习 16.4

1. 全局变量 I 的类型是 INTEGER, 函数 F 如下:

```
function F return INTEGER is
begin
    I := I + 1;
    return I;
end F;
```

请说明下列语句为什么是非法的,假设 I 已被置为 1。

- a) I := I + F;
- b) A(I) := F;
- c) A(I, F) := 0;

16.5 程序设计

让我们利用最后一节来讨论一下有关 Ada 程序设计方面的问题。应该看到,在设计 Ada 编译程序时,人们实际应用 Ada 的经验并不多。因此,本节所述的只是一些提示性内容,且只着眼于 Ada 的程序设计,并假设读者已有一些程序设计的经验。

先让我们看一下初步的且与程序设计风格有关的问题。

标识符必须有意义。程序的书写要整齐,本书采用的是 LRM 中语法所推荐的风格。程序中要适当地加进注释,对子程序要说明其用途。对某一信息尽量只书写一次,例如采用对数据和常量进行说明而不用显式的直接量等等。

程序设计实际上是把实际问题用编程语言加以抽象,并将其映射出来。正如 1.2 节所述,编程语言的发展,引入了多层次的抽象,而 Ada 引入了其它实用语言所没有的数据抽象。

设计和使用抽象所涉及的一个重要概念是信息隐蔽。即只在那些需要使用信息的程序部分处信息才是可访问的。

利用程序包和私有数据类型来隐藏不必要的细节奠定了好的 Ada 程序设计的基础。正如我们曾经指出的,Ada 促进了可重复使用的软件元件的发展。程序包是关键成份。Ada 程序设计主要是设计一组程序包以及程序包间的接口,我们常期望利用现存的程序包。为了使程序包通用,须使设计出的程序包具有良好的一致性、明确性和通用性。程序包设计的难度在于判定哪些项密切相关或者说将哪些项可归并在一起构成一程序包,以及程序包要达到何种通用度。

如果程序包过于通用，则该包势必十分庞杂且效率低。如果程序包不够通用，则又不能达到预期的目的。

规范式说明为程序包提供了接口，或至少提供了如何使用接口的语法。当然还应为程序包定义语义，而语义可用注释来表示。如8.1节的程序包 STACK，它的规范式说明告诉我们该程序包有两个子程序分别为 PUSH 和 POP，以及每个子程序的参数和结果类型。

然而规范式说明本身并没有告诉我们，调用 POP 实际上是取栈顶元素并使栈顶元素退栈。从语言的角度看，下列程序包体对使用来说已足够了。

```
package body STACK is
    procedure PUSH(X:INTEGER) is
    begin
        null;
    end;
    function POP return INTEGER is
    begin
        return 0;
    end;
end STACK;
```

然而，可想而知，在未来的软件元件产业中，谁出售上述那种程序包，谁就会很快破产。

我们现在来讨论某些实用程序包。这些程序包的分类与构成程序包的项有关。

最简单的程序包形式是程序包中只含一组有关的类型和常量，而没有程序包体。程序包 SYSTEM 和 ASCII 就属于这一类。8.1节中给出了程序包 DIURNAL；这个程序包似乎不好，若它有数组 TOMORROW，则亦应有数组 YESTERDAY。较好的程序包是在程序包中含有数学常量、常量转换(公制到英制)、物理和化学常量表等等。化学常量表程序包为：

```
package ELEMENTS is
    type ELEMENTS is(H,He,Li,...);
    --请参看图
    ATOMIC_WEIGHT :array(ELEMENT) of REAL
        :=(1.008,4.003,6.940,...);
    ...
end ELEMENTS;
```

另一情况是，程序包中所含有的函数是与程序包的应用领域有关。突出的例子是16.3节中提到的数学库。尽管从提高效率的角度考虑，象 SIN 和 COS 这样的函数共享了一公用过程，但在该数学库中函数均是各自独立的。而程序包往往把这类公用过程隐蔽起来，因为它只是一实现细节。

有时为了隐藏副作用而需要使用程序包。明显的例子是练习8.1(1)中的程序包 RANDOM。程序包如 SEQUENTIAL_IO 将大量的隐蔽的细节信息封装起来并且提供了大量有关的

服务。问题是程序包仅用来满足基本需要呢？还是为了编程方便进一步提供一些额外的子程序呢？程序包 TEXT_IO 中就含有许多方便编程的子程序。

许多程序包属于提供存取控制某种数据结构的工具。数据库中可以只有一个项元，就象程序包 RANDOM 中的那样。亦可以是编译程序的符号表，或是商业上用的数据库。9.3 节中的程序包就是商用数据库的例子。

程序包的另一个重要用途是提供新的数据类型以及相应的运算。9.1 节中的程序包 COMPLEX_NUMBER, RATIONAL_NUMBERS(练习 9.1(2)) 和 QUEUES(练习 11.4(3)) 便是这方面的实例。在这种情形中，私有类型能使我们把类型的表示与类型操作分开。在某种意义上讲，新类型可以看作为语言的扩充。

有了上述的程序包后，随之而来的一个问题是，与其使用运算符不如使用函数。Ada 不象有些语言那样灵活，不能引入新的运算符，并且运算符的优先级是固定的。但这亦带来了一个好处，无论怎么变化运算符的使用形式，也不会使以 Ada 编制的程序看上去不象 Ada 程序了。而在 POP-2 语言中则不然。尽管如此，Ada 中有的情形也会令人吃惊。如我们可写为：

```
function "-"(X, Y : INTEGER) return INTEGER is
begin
  return STANDARD."+"(X, -Y);
end "-";
```

显然这样做是不好的，因为一般好的程序尽量不使读程序的人感到意外。具有数学风格的函数亦可看作为运算符。对此类函数，一般原则是，尽可能地使它们具有运算符的代数特性。如“+”和“*”要具有交换性。尽管有时是必须的，但最好不使用混合类型的运算。

程序包 COMPLEX_NUMBERS 中定义的运算符具有我们所期望的特性，且类型不混用。恐怕为了使程序包 COMPLEX_NUMBERS 完整，还应象程序包 RATIONAL_NUMBERS 那样在该程序包中添加一元“+”和“-”。如果能用重载的类型名实现类型转换就好了，这样我们可直接写 COMPLEX(2.0) 而不必写成 COS(2.0, 0.0)。然而 Ada 不允许这样做。这种形式的类型转换只限于预定义的和派生的数值类型。

让我们再回过头来看 14.3 节中的程序包 CALENDAR 中的运算符。这里的运算符可进行混合类型的相加，在该程序包中通过提供 2 个“+”的重载来保证加法运算的交换律。

当引入诸如 COMPLEX 和 RATIONAL 的数学类型后，最好将它们作为私有类型来使用。为此需要提供如同真正运算符那样的构造符和选择符函数。在数学运算中，运算符给出了运算的含义及类别，如除号“/”为有理数的除法运算。函数运算符最好也应具有真正运算符的这种含义。

一个最常遇到且最难解决的问题是，在程序包中到底要为数学类型提供多少个私有类型。如果私有类型提供的过少，会招致所有用户都须建立许多不必要的新外子程序。如果太多，则

地解释了这个问题。如果类型 POLYNOMIAL 是判别式记录,如:

```
type POLYNOMIAL(N;INDEX:=0) is
  record
    A:INTEGER_VECTOR(0..N);
  end record;
```

在用无约束的多项式场合,编译程序为它分配可能用到的最大空间。不难知,给出判别式范围的合理上界尤为重要。如果我们写成

```
type POLYNOMIAL(N;INTEGER:=0) is ...
```

则每个无约束多项式都占有长度为 INTEGER'LAST 的数组空间,这样存贮空间很快就会被用完。如果所有的多项式都比较小,则用判别式记录将能取得令人满意的结果。反之,如果各个多项式的长度差别都比较大,则最好采用存取类型。也可将它们混用,即前几项用固定数组,而其余的用存取类型。为了正确地组织各种实现方法并使用户不可见,显然应 用私有类型。作为练习,请读者设计一相应的程序包。

在设计程序包时,需要考虑当出错时怎么办。这就涉及到了异常。异常这个概念虽然对编程语言来说并不新颖,但除了在 PL/I 中外,并没有被广泛采用。即使在 PL/I 中,其功效也不能令人满意。Ada 中的异常与 PL/I 中的异常在许多重要方面都存在着不同。在 Ada 中,异常发生后并不返到引起异常的那点,而是强迫执行异常处理段以代替从出错点继续往下执行。

我们已提醒过,对待异常要十分小心。在第 10 章中我们曾反对使用不必要的异常。因为很难保证处理异常时,异常真正的引发原因就是我们想到的原因。

在运用异常机制时,首要的方针应是要有清晰和完整的接口。让我们再以阶乘函数为例进行讨论。当参数非法时采用的处理措施可能为下列之一:

- 打印出错信息。
- 返回一预定值。
- 利用布尔型参数返回状态。
- 调用错误处理过程。
- 引发一异常。

打印出错信息这种处理远不能令人满意。因为这会引起大量的细节问题,如输出文件的名、信息格式等等。而且调用阶乘函数的程序因出错而失去控制,无法实现预期的作用,同时某些文件也会被错误信息弄的乱七八糟。此外,还存在打印出错信息后,下步该怎么办的问题。另一个处理措施也许如练习 10.1(2)一样,返回一个如 -1 这样的预定值。这也不能令人满意。因为不能保证用户调用阶乘函数后都对返回值进行检查。与其让用户调用函数后检查返回值,不如让用户在函数调用前先对参数进行检查。因此函数的正确调用依赖于调用程序。

如果采用返回布尔类型的辅助状态值的办法,如象 10.2 节中 PUSH 和 POP 那样,我们就不能再使用函数而只能用过程了。并且仍然要求用户程序预先检查参数,这也要依赖于调用程序。

我们可以调用一个由用户提供的全程过程。这意味着我们需要一个标准名,这也是不能令人满意的。我们不能象参数那样来传递出错处理过程以及凭借类属机制实现出错处理,这如同

用汽锤砸核桃。无论采用上述哪种措施,仍存在象打印出错信息一样的问题,即下一步将做何处理以及怎样从函数返回。如果有谁认为打印出错信息后,下一步便可停机,那么他的想法就过于天真了。炼钢厂的经营者是不希望因某部分的一些小错而终止控制程序工作的;必须采取某种方法使程序照常运行下去。

在 Ada 中只有两种渠道可从子程序中出来;它们是返回子程序的调用外或异常传播(用异常有效地代替其它语言的全程 goto 和标号)。似乎可排除采用子程序进行出错处理,因为调用子程序总是要返回调用点,当出错后,从该点继续执行是不可靠的,结论是采用异常进行出错处理。

在决定是否使用异常时,还要考虑的一个问题是,导致异常引发的条件是否属正常事件,如果是,则此种情况下使用异常是不正确的。例如在程序包 SEQUENTIAL_IO 中采用逻辑函数来测试是否抵达文件尾,而不是用异常。因为我们总是能预计到可能到达文件尾而对文件进行测试的——见练习 15.(1)。

反之,如果在计算数学表达式时使用了程序包 STACK,为了使计算正确,不能让栈出现下溢,对这种难以预料的情况,便可使用异常。需要指出,当使用程序包 SEQUENTIAL_IO 时,如果意外地出现读文件尾之后的数,则会引发异常(END_ERROR)。

异常不是一把万能钥匙。我们不能用它解决所有的问题。一旦异常发生则必须进行处理,否则程序会终止运行。实际上这恰好是异常的一个优点,如果发生异常时不作任何处理,则程序会安然终止。如果异常处理仅返回状态值而不作任何处理,则程序会安然无恙的继续执行下去。

应避免在异常处理段中乱用 others。如果用了 others,则必须能够对付任何可能存在的错误情况,并采取相应的措施。我们不要对可料到的异常使用 others。

在程序设计中的另一个重要内容是任务的设计。人们往往清楚要用多个任务来解决一个问题,但如何把各种活动分配到不同的任务中人们并不总是清楚的。

有关任务间的交互作用有两个主要问题须解决,一个是任务间的信息传输,另一个是对多任务共享数据的访问控制。

汇合提供了信息闭合传送的机制。如 14.2 节中的过程 SHOPPING 中母亲与家庭其他成员的交互作用。如果需要进行非连接传输,以便于发送者在发送的信息没有被接收前就继续运行,则需要借助中间任务。如 14.1 节中的任务 BUFFERING 和 14.5 节中的任务类型 MAILBOX。

在 Ada 中对共享数据的存取控制也是由中间任务来完成的,而在其它语言中是采用被动结构,如管程,甚至更低级的信号量来实现共享数据的存取控制。Ada 的方法更加清晰和安全。14.4 节中的任务 PROTECTED_VARIABLE 就是一个例子。我们常把任务封装在程序包中以便实现所需的规程。例如 14.4 节中的程序包 RADER_WRITER 和 14.7 节中的程序包 RESOURCE_ALLOCATOR。

有时信息传递与数据存取控制间的区别很模糊。在宏观上看,任务 BUFFERING 传递信息,而在微观上看,它是控制缓冲区的存取。

任务分为用户任务和服务任务。纯服务任务具有入口且不调用其它任务,而纯用户任务则无入口且要调用其它任务。汇合的不对称性使这两种任务的差别变的明显。服务任务并不知用户任务名,而用户任务为了要调用服务任务则必须知道服务任务名。往往一个任务既是服务任

务又是用户任务。例如14.8节中的任务类型 READ_AGENT。在设计一组相互作用的任务时，其中的一个问题是确定任务的入口。分清属于服务任务还是用户任务会有益于我们的设计。服务任务总具有入口。另一种判别方法是查看选择语句，如果某任务利用选择语句进行汇合选择，则该任务必然有入口，那么此任务是服务任务。选择语句可用来接收多个调用中的一个，而不能被用在对发出多个调用中选择一个。

我们不能指望夭折的任务不给程序带来灾难。夭折语句原本就是在极端情况下使用的。一种方法是把夭折语句放在管理任务中，由它来中止整个子程序的运行，而等待操作员的命令。

我们希望尽量避免使用夭折语句，原因是夭折语句使任务难于提供可靠的服务。如同在14.6节中所看到的程序包READER_WRITER那样。

多任务的Ada程序常把一组相互合作的任务设计到一起，此时，我们便可以靠调用任务来获得所需的规程，并且服务任务的设计亦变的简单了。

延时的使用也要小心。在实时系统中一般需要用到延时等与时间有关的调用，服务任务尽量在指定时间范围内与调用任务进行汇合。

最后让我们再来讨论关于类属和参数的问题。应该把程序包设计成专用的还是通用的？这个问题实际上是指，是设计子程序还是类属？正象本章开头所指出的那样，我们现在应用Ada编程的经验还不多，因此在这方面提不出什么成熟的建议。我们所认识到的是，在今后的软件市场上将会出售各种通用的和各种专用的程序包。我们在此以假想的未来软件商店中，顾客与服务员的对话作为本章的结尾。

顾客：我可以看一下在柜台中的那个读一写程序包吗？

服务员：当然可以，如果你不一定非要对夭折加以限制的加强型，我们这里还有更为灵活的程序包，是上星期刚到货的。

顾客：好的，我要用于交互式系统的程序包。当然愈新愈好。多少钱？

服务员：250元，因为是新的，所以价格优惠，优惠10%并免费赠送一个随机数发生程序。

顾客：好的，我买下了。

服务员：是您直接把软件拿走呢？还是要我们为您安装。

顾客：谢谢，我自己来安装吧！

用上述这个幻想来结束本书。希望读者通过自己的编程实践进一步加深对Ada的理解。殷切地希望本书对读者的将来工作有所帮助。

附录1 预定义属性

本附录给出了 Ada 的预定义属性,此外,实现时还可定义附加的属性。

P'ADDRESS 前缀 P 表示对象、程序单位、标号或实体:

该属性给出分配给 P 的存贮单元的首址。对于子程序、程序包、任务程序单位或标号,该属性值指向与相应的体或语句的机器代码。对于已有地址子句的入口,该属性值与对应的硬件中断有关。该属性值为程序包 SYSTEM 中定义的 ADDRESS 类型。(参见 15.4 节)

P'AFT 前缀 P 表示某定点子类型:

该属性给出 P 的精度所要求的小数点后的十进制数字的位数。但 P 的增量(delta)大于 0.1 时,该属性值为 1。(P'AFT 为使 $(10^{**}N)^{*}P'\text{DELTA}$ 大于或等于 1 的最小正整数 N) 该属性值为通用整型(universal integer)。(参见 15.2 节)

P'BASE 前缀 P 表示某类型或子类型:

该属性用来表示 P 的基类型。它仅允许作为另一属性名的前缀,如 P'BASE'FIRST。(参见 12.1 节)

P'CALLABLE 前缀 P 表示任务类型:

当任务 P 的执行完成或终止,或者任务不正常时,该属性值为 FALSE,否则为 TRUE。该属性值为预定义类型 BOOLEAN。(参见 14.6 节)

P'CONSTRAINED 前缀 P 表示某个带判别式类型的对象:

如果某判别式限定适用于对象 P,或该对象为常量(包括 in 型的形参或类属参数),该属性值为 TRUE,否则为 FALSE。如果 P 为 in_out 型的类属形参,或是 in_out 型和 out 型的类属形参,并且在相应参数规范式说明中给出的类型标记代表某带判别式的非受限类型,那么该属性值从相应的实参那里获得。该属性值为预定义类型 BOOLEAN。(参见 11.1 节)

P'CONSTRINED 前缀 P 表示某私有类型或子类型:

如果 P 表示某带判别式的非受限私有类型,该属性值为 FALSE,如果 P 表示某类属形式私有类型,且相应的实在子类型为带判别式的非受限类型或为非受限数组类型,该属性值也为 FALSE,否则为 TRUE。该属性值为预定义类型 BOOLEAN。(参见 13.2 节)

P'COUNT 前缀 P 表示某任务程序单位的一个入口:

该属性给出当前排列在本入口上的入口调用数(如果该属性在对于入口 P 的某个接收语句内求值,那么其值不包括对调用任务的计数)。该属性值为通用整型。(参见 14.4 节)

P'DELTA 前缀 P 表示某定点子类型:

该属性给出 P 的固定精度定义中规定的增量(delta)值。该属性值为通用实型(universal real)。(参见 12.4 节)

P'DIGITS 前缀 P 表示某浮点子类型:

该属性给出 P 的样板数的十进制表示中的尾数位数。(12.3 节的数 D 即由该属性给出)该属性值为通用整型。

P'EMAX 前缀 P 表示某浮点子类型:

该属性给出 P 的样板数的二进制表示中的最大指数值。(12.3 节的积 $A * B$ 即由该属性给出)该属性值为通用整型。

P'EPSILON 前缀 P 表示某浮点子类型:

该属性给出 P 的样板数 1.0 与下一个样板数之差的绝对值。该属性值为通用实型。(参见 12.3 节)

P'FIRST 前缀 P 表示某纯量类型或子类型:

该属性给出 P 的下界。该属性值的类型与 P 的类型相同。(参见 4.8 节)

P'FIRST 前缀 P 表示某数组类型或受限数组子类型:

该属性给出 P 的第一个下标范围的下界。该属性值的类型与该下界的类型相同。(参见 6.1 节)

P'FIRST(N) 前缀 P 表示某数组类型或受限数组子类型:

该属性给出 P 的第 N 个下标范围的下界。该属性值的类型与该下界的类型相同。变元 N 必须为通用整型的静态表达式。N 的值必须为正且不得大于 P 的维数。(参见 6.1 节)

P'FIRS_BIT 前缀 P 表示某记录对象的一个分量:

该属性给出从这个对象占据的第一个存贮单元到给定分量的第一个存贮位的偏移量。该偏移量以位来测试。该属性值为通用整型。(参见 15.4 节)

P'FORE 前缀 P 表示某定点子类型:

该属性给出 P 的任何值的十进制表示下整数部分所需的最小字符数, 这里假定该表示不包括指数但包含一个减号或空格字符前缀。(该最小字符数不包括多余的 0 或下横线, 且至少为 2。)该属性值为通用整型。(参见 15.2 节)

P'IMAGE 前缀 P 表示某离散类型或子类型:

该属性为带单个参数的函数。实参 X 必须为 P 的基类型的某值。结果类型为预定义类型

STRING。结果是 X 的映象，即显示形式表示该值的字符序列。整数值的映象为相应的十进制直接量，它不带下横线，也无打头的 0 以及指数或尾空格，但有一个减号或空格字符前缀。枚举值的映象为相应的大写标识符或相应的字符直接量（包括双引号，但不包含首尾空格）。非图象字符的字符映象由实现定义。

P'LARGE 前缀 P 表示某实数子类型：

该属性给出子类型 P 的最大正样板数。该属性值为通用实型。（参见 12.3 节和 12.4 节）

P'LAST 前缀 P 表示某纯量类型或子类型：

该属性给出 P 的上界。该属性值的类型与 P 的类型相同。（参见 4.8 节）

P'LAST 前缀 P 表示某数组类型或受限数组子类型：

该属性给出 P 的第一个下标范围的上界。该属性值的类型与这个上界的类型相同。（参见 6.1 节）

P'LAST(N) 前缀 P 表示某数组类型或受限数组子类型：

该属性给出 P 的第 N 个下标范围的上界。该属性值的类型与这个上界的类型相同。实元 N 必须为通用整型的静态表达式。N 的值必须为正且不得大于 P 的维数。（参见 6.1 节）

P'LAST_BIT 前缀 P 表示某记录对象的某个分量：

该属性给出从这个记录对象占据的第一个存储单元到给定分量的第一个存储位的偏移。

该属性给出 P 的基类型的机器表示下尾数的位数(这些数字为 0 到 P'MACHINE_RADIX - 1 范围内的扩充数字)。该属性值为通用整型。(参见 15.5 节)

P'MACHINE_OVERFLOW 前缀 P 表示某实型或子类型:

如果 P 的基类型的任何预定义运算在溢出状态下提供一个正确结果或引发异常 NUMERIC_ERROR, 那么, 该属性值为 TRUE, 否则为 FALSE。该属性值为预定义类型 BOOLEAN。(参见 15.5 节)

P'MACHINE_RADIX 前缀 P 表示某浮点类型或子类型:

该属性给出 P 的基类型的机器表示下使用的基数值。该属性值为通用整型。(参见 15.5 节)

P'MANTISSA 前缀 P 表示某实数子类型:

该属性给出 P 的样板数的二进制尾位数。(12.3 节的浮点类型的 B 或 12.4 节的定点类型的 B 即由该属性给出)。该属性值为通用整型。

P'POS 前缀 P 表示某离散类型或子类型:

该属性为带单个参数的函数。实参 X 必须为一个 P 的基类型的值, 结果类型为通用整型。结果为实参值的位置数。(参见 4.8 节)

P'POSITION 前缀 P 表示某记录对象的某个分量:

该属性给出从这个记录对象占据的第一个存贮单元到给定分量的第一个存贮单元的偏移量。该偏移量以存贮单元来测试。该属性值为通用整型。(参见 15.5 节)

P'PRED 前缀 P 表示某离散类型或子类型:

该属性为带单个参数的函数。实参 X 必须为一个 P 的基类型的值, 结果为 X 的前一个位置上的值。如果 X 等于 P'BASE'FIRST 则异常 CONSTRAIN_ERROR 被引发。(参见 4.8 节)

P'RANGE 前缀 P 表示某数组类型或受限数组子类型:

该属性给出 P 的第一个下标范围, 即范围 P'FIRST..P'LAST。(参见 8.1 节)

P'RANGE(N) 前缀 P 表示某数组类型或受限数组子类型:

该属性给出 P 的第 N 个下标范围, 即范围 P'FIRST(N)..P'LAST(N)。(参见 8.1 节)

P'SAFE_EMAX 前缀 P 表示某浮点类型或子类型:

该属性给出 P 的基类型安全数的二进制表示下的最大指数值。(12.3 节的数 E 即由该属性给出)该属性值为通用整型。

P'SAFE_LARGE 前缀 P 表示某实型或子类型:

该属性给出 P 的基类型的最大正安全数。该属性值为通用实型。(参见12.3节和12.4节)

P'SAFE_SMALL 前缀 P 表示某实型或子类型:

该属性给出 P 的基类型的最小正安全数(非0)。该属性值为通用实型。(参见12.3节和12.4节)

P'SIZE 前缀 P 表示某对象:

该属性给出用来保存这个对象的存贮位数。该属性值为通用整型。(参见15.4节)

P'SIZE 前缀 P 表示任何类型或子类型:

该属性给出用来保存 P 的任何可能对象所需的最少存贮位数。该属性值为通用整型。(参见15.4节)

P'SMALL 前缀 P 表示某实数子类型:

该属性给出 P 的最小正样板数(非0)。该属性值为通用实型。(参见12.3节和12.4节)

P'STORAGE_SIZE 前缀 P 表示某存取类型或子类型:

该属性给出保留在 P 的基类型的相关收集的存贮单元总数。该属性值为通用整型。(参见15.4节)

P'STORAGE_SIZE 前缀 P 表示某任务类型或任务对象:

该属性给出保留在 P 的每次激活的存贮单元数。该属性值为通用整型。(参见15.4节)

P'SUCC 前缀 P 表示某离散类型或子类型:

该属性为带单个参数的函数。实参 X 必须为 P 的基类型的值。结果类型为 P 的基类型。结果为 X 的下一个位置上的值。如果 X 等于 P'BASE'LAST 则异常 CONSTRAINT_ERROR 被引发。(参见4.8节)

P'TERMINATED 前缀 P 表示某任务类型:

如果任务 P 被终止,那么该属性值为 TRUE,否则为 FALSE。该属性值为预定义类型 BOOLEAN。(参见14.6节)

P'VAL 前缀 P 表示某离散类型或子类型:

该属性为带单个参数的函数,所带参数 X 可以是任何整型。结果类型为 P 的基类型。结果是以对应 X 的通用整型值作为位置量的 P 的值。如果对应 X 的通用整型值不在范围 P'POS(P'

该属性为带单个参数的函数。实参 X 为预定义类型 STRING 的值。结果类型为 P 的基类型。任何对应于 X 的字符序列的首尾空格皆被忽略。如果 P 为枚举类型，X 的字符序列又符合枚举直接量的语法规则，且 P 的基类型内存在 X 对应的直接量，那么该属性结果值即为相应的枚举值。如果 P 为整型，X 的字符序列符合整数直接量的语法规则（并带单个字符前缀十或一），且 P 的基类型内存在相应的值，则该属性结果值即为这个值。除此之外的其它任何情况，异常 CONSTRAINT_ERROR 被引发。

P'WIDTH 前缀 P 表示某离散子类型：

该属性给出 P 的所有值的最大映象长度（该映象为属性 IMAGE 返回的字符序列）。该属性值为通用整型。（参见 15.2 节）

附录2 预定义杂注

本附录给出了 Ada 的预定义杂注,此外,实现时还可定义附加的杂注。

CONTROLLED 取某存取类型的简单名作单个变元。本杂注只许直接出现在说明部或包含该存取类型的程序包规范式说明内,该存取类型的说明必须先于本杂注。本杂注不能用于派生类型。本杂注说明:对该存取类型值所标示的对象不必进行自动存贮回收,除非程序执行离开包含该存取类型说明的分程序、子程序体、任务体或者离开主程序。(参见11.3节)

FLABORATE 取一个或多个表示库程序单位的名作变元。本杂注只允许直接出现在某编译单位的关联子句之后(随后的库程序单位或次级单位之前)。每个变元必须为关联子句指明的库程序单位的简单名。本杂注规定:相应的库程序单位体必须在给定编译单位之前加工。如果给定编译单位为子单位,那么相应的库程序单位体必须在该子单位的前趋库单位体之前加工。(参见16.3节)

INLINE 取一个或多个名作变元,每个名可以是子程序名或类属子程序名。本杂注只允许出现在说明部或程序包规范式说明中凡说明项可出现的地方,或出现在编译单位中某库程序单位之后和随后的编译单位之前。本杂注规定:这些子程序体在每个调用处作插入式展开,对于类属子程序,本杂注指的是其例化调用。(参见15.5节)

INTERFACE 取一语言的名称及子程序名作变元。本杂注允许出现在凡说明项可出现的地方,但它所指的子程序必须先作说明且在同一说明部内。本杂注也允许出现在某库程序单位中,这时它必须出现在这个子程序的说明之后和任何随后的编译单位之前。本杂注指出:这个子程序体可用指定的另一语言书写(但必须遵守该语言的调用约定),并通知编译程序给这个子程序提供目标模块。(参见15.7节)

LIST 取标识符 ON 或 OFF 作单个变元。凡杂注可出现的地方本杂注皆可出现。它指定:编译继续列表或暂停列表,直到在本次编译中给出一个带有相反变元的 LIST 杂注。若编译已在列表,那么本杂注本身将被列在表中。

MEMORY_SIZE 取一数值直接量作单个变元。本杂注仅允许出现在编译的开始位置(本次编译的第一个编译单位之前)。它指定:用该数值直接量来定义 MEMORY_SIZE 的大小。(参见15.5节)

OPTIMIZE 取标识符 TIME 或 SPACE 作单个变元。本杂注只允许出现在说明部中,它适用于包含该说明的分程序或体。它指定:基本优化的标准是时间(TIME)还是空间(SPACE)。

PACK 取某记录类型或数组类型的简单名作单个变元,本杂注可出现何处以及指定类型上的限制和表示子句的有关规定相同。本杂注规定:占用最少存贮空间是选择指定类型表示的主要标准。

使用本复印件
请尊重相关知识产权!

PAGE 本杂注无变元,允许出现在凡杂注可出现的地方。它规定:随后的杂注文本应从新的一页开始列出(如果编译处在列表状态)。PRIORITY 取预定义整数子类型 PRIORITY 的静态表达式作单个变元。本杂注只允许出现在任务程序单位的规范式说明内,或直接出现在主程序的最外层说明部内。它用来规定任务(或任务类型的所有任务)的优先数或主程序的优先数。(参见14.3节)

STORAGE_UNIT 取一数值直接量作单个变元。本杂注只允许出现在编译的开始位置(本次编译的第一个编译单位之前)。本杂注规定:用指定的数值直接量来定义 STORAGE_UNIT 的大小。(参见15.5节)

SYSTEM_NAME 取一枚举直接量作单个变元。本杂注只允许出现在编译的开始位置(本次编译的第一个编译单位之前)。本杂注规定:用该枚举直接量来定义常量

SYSTEM_NAME 的大小。本杂注要求指定的枚举直接量标识符对应于程序包 SYSTEM 中说明的类型 NAME 的某个直接量。(参见15.5节)

附录3 预定义语言环境

本附录概述给出了程序包 STANDARD 的规范式说明。该程序包定义了 Ada 的语言环境。程序包 STANDARD 中类型上的预定义运算符以注释形式给出。

```
package STANDARD is
```

```
    type BOOLEAN is (FALSE,TRUE);
```

——该类型的预定义关系运算符如下：

```
    ——function "=" (LEFT,RIGHT:BOOLEAN) return BOOLEAN;  
    ——function "/=" (LEFT,RIGHT:BOOLEAN) return BOOLEAN;  
    ——function "<" (LEFT,RIGHT:BOOLEAN) return BOOLEAN;  
    ——function "<=" (LEFT,RIGHT:BOOLEAN) return BOOLEAN;  
    ——function ">" (LEFT,RIGHT:BOOLEAN) return BOOLEAN;  
    ——function ">=" (LEFT,RIGHT:BOOLEAN) return BOOLEAN
```

——逻辑运算符如下：

```
    function "and" (LEFT,RIGHT:BOOLEAN) return BOOLEAN;  
    function "or" (LEFT,RIGHT:BOOLEAN) return BOOLEAN;  
    function "xor" (LEFT,RIGHT:BOOLEAN) return BOOLEAN;  
    function "not" (RIGHT:BOOLEAN) return BOOLEAN;
```

——通用类型 universal_integer 为预定义类型

```
    type INTEGER is ... 由实现定义;
```

——该类型的预定义运算符如下：

```
    ——function "=" (LEFT,RIGHT:INTEGER) return BOOLEAN;  
    ——function "/=" (LEFT,RIGHT:INTEGER) return BOOLEAN;  
    ——function "<" (LEFT,RIGHT:INTEGER) return BOOLEAN;  
    ——function "<=" (LEFT,RIGHT:INTEGER) return BOOLEAN;  
    ——function ">" (LEFT,RIGHT:INTEGER) return BOOLEAN;  
    ——function ">=" (LEFT,RIGHT:INTEGER) return BOOLEAN;
```

```
    ——function "+" (RIGHT:INTEGER) return INTEGER;  
    ——function "-" (RIGHT:INTEGER) return INTEGER;  
    ——function "abs" (RIGHT:INTEGER) return INTEGER;
```

```
    ——function "+" (LEFT,RIGHT:INTEGER) return INTEGER;
```



```
—function "—" (LEFT,RIGHT;INTEGER) return INTEGER;
—function "* " (LEFT,RIGHT;INTEGER) return INTEGER;
—function "/" (LEFT,RIGHT;INTEGER) return INTEGER;
—function "rem" (LEFT,RIGHT;INTEGER) return INTEGER;
—function "mod" (LEFT,RIGHT;INTEGER) return INTEGER;
—function "* *" (LEFT;INTEGER;RIGHT;INTEGER) return INTEGER;
```

—实现时可提供附加的预定义整型。值得注意的是：这样的附加类型名要以 INTEGER 作尾名，如 SHORT_INTEGER, LONG_INTEGER。类型 universal_integer 或任何附加预定义整型的运算符规范式说明是通过以本类型名去置换类型 INTEGER 的相应运算符规范式说明中的 INTEGER 得到，但指数运算符的右操作数除外。

—普适类型 universal_real 为预定义类型。

```
—type FLOAT is ... 由实现定义;

—function "==" (LEFT,RIGHT;FLOAT) return BOOLEAN;
—function "/=" (LEFT,RIGHT;FLOAT) return BOOLEAN;
—function "<" (LEFT,RIGHT;FLOAT) return BOOLEAN;
—function "<=" (LEFT,RIGHT;FLOAT) return BOOLEAN;
—function ">" (LEFT,RIGHT;FLOAT) return BOOLEAN;
—function ">=" (LEFT,RIGHT;FLOAT) return BOOLEAN;

—function "+" (RIGHT;FLOAT) return FLOAT;
—function "—" (RIGHT;FLOAT) return FLOAT;
—function "abs" (RIGHT;FLOAT) return FLOAT;

—function "+" (LEFT,RIGHT;FLOAT) return FLOAT;
—function "—" (LEFT,RIGHT;FLOAT) return FLOAT;
—function "* " (LEFT,RIGHT;FLOAT) return FLOAT;
—function "/" (LEFT,RIGHT;FLOAT) return FLOAT;
—function "* *" (LEFT;FLOAT;RIGHT;FLOAT) return FLOAT;
```

—实现时可提供附加的预定义浮点类型。值得注意的是：这样的附加类型名要以 FLOAT 作尾名，如 SHORT_FLOAT, LONG_FLOAT。类型 universal_real 或任何附加预定义浮点类型的运算符规范式说明，通过以本类型名去置换类型 FLOAT 的相应运算符规范式说明中的 FLOAT 得到。

—如下运算符预定义给通用类型：

```
—function "* " (LEFT;universal_integer;RIGHT;universal_real) return universal
_real;
```

```

——function "*" (LEFT;universal_real;RIGHT;universal_integer) return universal
_real;
——function "/" (LEFT;universal_real;RIGHT;universal_integer) return universal
_real;

```



——类型 universal_fixed 为预定义类型,其预定义运算符有:

```

——function "*" (LEFT:any_fixed_point_type;RIGHT:any_fixed_point_type)
      return universal_fixed;
——function "/" (LEFT:any_fixed_point_type;RIGHT:any_fixed_point_type)
      return universal_fixed;

```

——下面标准 ASCII 字符集组成。其控制字符的字符直接量不是标识符,它们以斜体字给出:

type CHARACTER is							
(nul,	soh,	stx,	etx,	eot,	enq,	ack,	bel,
bs,	ht,	if,	vt,	ff,	cr,	so,	si,
dle,	dc1,	dc2,	dc3,	dc4,	nak,	syn,	etb,
can,	em,	sub,	esc,	fs,	gs,	rs,	us,
" ,	'1'	'n'	'#'	'\$'	'%'	'8.'	'.'
'(,	')'	'*' ,	'+' ,	' ,'	'—'	' ,'	' /'
'0'	'1'	'2'	'3'	'4'	'5'	'8'	'7'
'8'	'9'	'.'	' ;'	' <'	' ='	' >'	' ?'
'@'	'A'	'B'	'C'	'D'	'E'	'F'	'G'
'H'	'I'	'J'	'K'	'L'	'M'	'N'	'O'
'P'	'Q'	'R'	'S'	'T'	'U'	'V'	'W'
'X'	'Y'	'Z'	'[]'	'\ '	'] '	' ^ '	' — '
'a'	'b'	'c'	'd'	'e'	'f'	'g'	
'h'	'i'	'j'	'k'	'l'	'm'	'n'	'o'
'p'	'q'	'r'	's'	't'	'u'	'v'	'w'
'x'	'y'	'z'	'{ }'	' '	'}'	'~'	del);

for CHARACTER use --128 ASCII 字符集
(0,1,2,3,4,5,...,125,126,127);

——类型 CHARACTER 的预定义运算符和任何枚举类型的相同。

package ASCII is

——控制字符:

NUL: constant CHARACTER := *nul*;
SOH: constant CHARACTER := *soh*;
STX: constant CHARACTER := *stx*;
ETX: constant CHARACTER := *etx*;
EOT: constant CHARACTER := *eot*;
ENQ: constant CHARACTER := *enq*;
ACK: constant CHARACTER := *ack*;
BEL: constant CHARACTER := *bel*;
BS: constant CHARACTER := *bs*;
HT: constant CHARACTER := *ht*;
LF: constant CHARACTER := *lf*;
VT: constant CHARACTER := *vt*;
FF: constant CHARACTER := *ff*;
CR: constant CHARACTER := *cr*;
SO: constant CHARACTER := *so*;
SI: constant CHARACTER := *si*;
DEL: constant CHARACTER := *del*;
DC1: constant CHARACTER := *dc1*;
DC2: constant CHARACTER := *dc2*;
DC3: constant CHARACTER := *dc3*;
DC4: constant CHARACTER := *dc4*;
NAK: constant CHARACTER := *nak*;
SYN: constant CHARACTER := *syn*;
ETB: constant CHARACTER := *etb*;
CAN: constant CHARACTER := *can*;
EM: constant CHARACTER := *em*;
SUB: constant CHARACTER := *sub*;
ESC: constant CHARACTER := *esc*;
FS: constant CHARACTER := *fs*;
GS: constant CHARACTER := *gs*;
RS: constant CHARACTER := *rs*;
US: constant CHARACTER := *us*;
DEL: constant CHARACTER := *del*;



——其它字符：

EXCLAM	:constant CHARACTER := '!' ;
QUOTATION	:constant CHARACTER := '"' ;
SHARP	:constant CHARACTER := '#' ;
DOLLAR	:constant CHARACTER := '\$' ;

```

PERCENT           :constant CHARACTER := '%';
AMPERSAND         :constant CHARACTER := '&';
COLON             :constant CHARACTER := ':';
SEMICOLON         :constant CHARACTER := ';';
QUERY              :constant CHARACTER := '?';
AT_SIGN            :constant CHARACTER := '@';
L_BRACKET          :constant CHARACTER := '[';
BACK_SLASH         :constant CHARACTER := '\';
R_BRACKET          :constant CHARACTER := ']';
CIRCUMFLEX        :constant CHARACTER := '^';
UNDERLINE          :constant CHARACTER := '_';
GRAVE              :constant CHARACTER := '`';
L_BRACE             :constant CHARACTER := '{';
BAR                :constant CHARACTER := '|';
R_BRACE             :constant CHARACTER := '}';
TILDE              :constant CHARACTER := '~';

```

——小写字母：

```

LC_A:constant CHARACTER := 'a';
...
LC_Z:constant CHARACTER := 'z';
end ASCII

```

——预定义子类型：

```

subtype NATURAL is INTEGER range 0..INTEGER'LAST;
subtype POSITIVE is INTEGER range 1..INTEGER'LAST;

```

——预定义字符串类型：

```

type STRING is array(POSITIVE range <>) of CHARACTER;
pragma PACK(STRING);

```

——该类型的预定义运算符如下：

```

---function "=" (LEFT,RIGHT:STRING) return BOOLEAN;
---function "/=" (LEFT,RIGHT:STRING) return BOOLEAN;
---function "<" (LEFT,RIGHT:STRING) return BOOLEAN;
---function "<=" (LEFT,RIGHT:STRING) return BOOLEAN;
---function ">" (LEFT,RIGHT:STRING) return BOOLEAN;
---function ">=" (LEFT,RIGHT:STRING) return BOOLEAN;
---function "&" (LEFT:STRING,RIGHT:STRING) return STRING;
---function "&" (LEFT:CHARACTER,RIGHT:STRING) return STRING;

```

— function "&." (LEFT:STRING;RIGHT:CHARACTER) return STRING;
— function "&." (LEFT:CHARACTER;RIGHT:CHARACTER) return STRING;

type DURATION is delta ... 实现时定义 range ... 实现时定义;
— 类型 DURATION 的预定义运算符和任何定点类型的相同。

— 预定义异常:

CONSTRAINT_ERROR:exception;
NUMERIC_ERROR:exception;
PROGRAM_ERROR:exception;
STORAGE_ERROR:exception;
TASKING_ERROR:exception;

end STANDARD;

Ada 还有其它的预定义库程序单位,如:CALENDAR,SEQUENTIAL_IO 和 DIRECT_IO,
TEXT_IO,SYSTEM,UNCHECKED_CONVERSION 和 UNCHECKED_DEALLOCATION。请分
别对照参见14.3节、15.1节、15.2节、15.5节和15.6节。



附录4 语法规则

本附录按书中的章节顺序给出了 Ada 的语法规则。注意：第3章的词法元素构成规则与其它规则稍有不同，词法元素之间可以自由地插入空格或新行，但单个词法元素中却不允许插入任何空格或新行。

第2章

1. 杂注 ::= pragma 标识符 [(变元结合 {, 变元结合})];
2. 变元结合 ::= 【变元标识符 =>】名 | 【变元标识符 =>】表达式

第3章

3. 图形字符 ::= 基本图形字符 | 小写字母 | 其它特殊字符
4. 基本图形字符 ::= 大写字母 | 数字 | 特殊字符 | 空格字符
5. 基本字符 ::= 基本图形字符 | 格式控制符
6. 标识符 ::= 字母 {【下横线】字母或数字}
7. 字母或数字 ::= 字母 | 数字
8. 字母 ::= 大写字母 | 小写字母
9. 数值字面值 ::= 十进制字面值 | 带基字面值
10. 十进制字面值 ::= 整数 【. 整数】 【指数】
11. 整数 ::= 数字 {【下横线】数字}
12. 指数 ::= E 【+】 整数 | E - 整数
13. 带基字面值 ::= 基 # 带基整数 【. 带基整数】 # 【指数】
14. 基 ::= 整数
15. 带基整数 ::= 扩充数字 {【下横线】扩充数字}
16. 扩充数字 ::= 数字 | 字母
17. 字符字面值 ::= ' 图形字符 '
18. 字符串字面值 ::= "(图形字符)"

第4章

19. 基本说明 ::= 对象说明 | 常量说明 | 类型说明 | 子类型说明 | 子程序说明 | 程序包说明
| 任务说明 | 类属说明 | 异常说明 | 类属例化 | 换名说明 | 延时常量说明
20. 对象说明 ::= 标识符表; 【constant】子类型表示 [= 表达式];
| 标识符表; 【constant】限定数组定义 [= 表达式];
21. 常量说明 ::= 标识符表; constant; = 通用静态表达式;
22. 标识符表 ::= 标识符 {, 标识符}
23. 赋值语句 ::= 变量名 := 表达式;
24. 分程序语句 ::= 【分程序简单名:】
【declare
说明部】



begin
 语句序列
 【exception
 异常处理段
 {异常处理段}
 end 【分程序简单名】;

25. 类型说明 ::= 完整类型说明 | 非完整类型说明 | 私有类型说明
26. 完整类型说明 ::= type 标识符 【判别式部分】 is 类型定义；
27. 类型定义 ::= 枚举类型定义 | 整型定义 | 实型定义 | 数组类型定义 | 记录类型定义
 | 存取类型定义 | 派生类型定义
28. 子类型说明 ::= subtype 标识符 is 子类型表示；
29. 子类型表示 ::= 类型标记 【约束】
30. 类型标记 ::= 类型名 | 子类型名
31. 约束 ::= 范围约束 | 浮点约束 | 定点约束 | 下标约束 | 判别式约束
32. 范围约束 ::= range 范围
33. 范围 ::= 范围属性 | 简单表达式 .. 简单表达式
34. 枚举类型定义 ::= (枚举直接量规范式说明 (, 枚举直接量规范式说明))
35. 枚举直接量规范式说明 ::= 枚举直接量
36. 枚举直接量 ::= 标识符 | 字符直接量
37. 名 ::= 简单名 | 字符直接量 | 运算符符号 | 下标分量 | 数组片 | 选择分量 | 属性
38. 简单名 ::= 标识符
39. 前缀 ::= 名 | 函数调用
40. 属性 ::= 前缀' 属性标志符
41. 属性标志符 ::= 简单名 【(通用静态表达式)】
42. 表达式 ::= 关系式 {and 关系式} | 关系式 {and then 关系式} | 关系式 {or 关系式}
 | 关系式 {or else 关系式} | 关系式 {xor 关系式}
43. 关系式 ::= 简单表达式 【关系运算符简单表达式】 | 简单表达式 【not】 in 范围
 | 简单表达式 【not】 in 类型标志
44. 简单表达式 ::= 【一元加法运算符】项 {二元加法运算符}项
45. 项 ::= 因子 {乘法运算符项}
46. 因子 ::= 初等量 【"+ * *!" 初等量】 | abs 初等量 | not 初等量
47. 初等量 ::= 数值直接量 | null | 聚集 | 字符串直接量 | 名 | 分配符 | 函数调用 | 类型变换
 | 受限表达式 | (表达式)
48. 逻辑运算符 ::= and | or | xor
49. 关系运算符 ::= /= | = | < | <= | > | >=
50. 二元加法运算符 ::= + | - | &
51. 一元加法运算符 ::= + | -
52. 乘法运算符 ::= * | / | mod | rem
53. 最高优先运算符 ::= * * | abs | not

54. 类型变换 ::= 类型标记(表达式)

55. 受限表达式 ::= 类型标记(表达式) | 类型标记' 累集

第5章

56. 语句序列 ::= 语句{语句}

57. 语句 ::= {标号}简单语句 | {标号}复合语句

58. 简单语句 ::= 空语句 | 赋值语句 | 过程调用语句 | 出口语句 | 入口调用语句 | 延迟语句
| 返回语句 | goto 语句 | 中止语句 | 引发语句 | 代码语句

59. 复合语句 ::= if 语句 | case 语句 | 循环语句 | 分程序语句 | 接收语句 | 选择语句

60. 标号 ::= {标号简单名}

61. 空语句 = null;

62. if 语句 ::= if 条件 then

语句序列

{else if 条件 then

语句序列}

【else

语句序列】

end if;

63. 条件 ::= 布尔表达式

64. case 语句 ::= case 表达式 is

case 语句选择

{case 语句选择}

end case;

65. case 语句选择 ::= when 选择 | 选择 => 语句序列

66. 选择 ::= 简单表达式 | 离散范围 | others | 分量简单名

67. 离散范围 ::= 离散子类型表示 | 范围

68. 循环语句 ::= 【循环简单名】

【重复流程】 loop

语句序列

end loop 【循环简单名】;

69. 重复流程 ::= while 条件 | for 循环参数规范式说明

70. 循环参数规范式说明 ::= 标识符 in 【reverse】 离散范围

71. 出口语句 ::= exit 【循环名】 【when 条件】;

72. goto 语句 ::= goto 标号名;

第6章

73. 数组类型定义 ::= 非受限数组定义 | 受限数组定义

74. 非受限数组定义 ::= array (下标子类型定义{,下标子类型定义}) of 分量子类型表示

75. 受限数组定义 ::= array 下标约束 of 分量子类型表示

76. 下标子类型定义 ::= 类型标志 range <>

77. 下标约束 ::= (离散范围{,离散范围})



78. 数组片 $::=$ 前缀(离散范围)
 79. 下标分量 $::=$ 前缀(表达式{,表达式})
 80. 聚集 $::=($ 分量结合{,分量结合})
 * 81. 分量结合 $::=[$ 选择(|选择)=>] $]表达式$
 82. 记录类型定义 $::=record$



分量表

end record

83. 分量表 $::=$ 分量说明{分量说明}|{分量说明}变体部分|null
 84. 分量说明 $::=$ 标识符表|分量子类型定义【:=表达式】;
 85. 分量子类型定义 $::=$ 子类型表示
 86. 选择分量 $::=$ 前缀.选择符
 87. 选择符 $::=$ 简单名|字符直接量|算符符号|all

第7章

88. 子程序说明 $::=$ 子程序规范式说明;
 89. 子程序规范式说明 $::=procedure$ 标识符【形式部分】
 |function 标识符【形式部分】return 类型标志
 90. 标识符 $::=$ 标识符|算符符号
 91. 算符符号 $::=$ 字符串直接量
 92. 形式部分 $::=($ 参数规范式说明{|规范式说明})
 93. 规范式说明 $::=$ 标识符表|参数传递型类型标志【:=表达式】
 94. 参数传递型 $::=in|in_out|out$
 95. 子程序体 $::=$ 子程序规范式说明 is

【说明部】

begin
 语句序列
 【exception
 异常处理程序段
 {异常处理程序段}】
 end 【标志符】;

96. 过程调用语句 $::=$ 过程名【实参部】;
 97. 函数调用 $::=$ 函数名【实参部】;
 98. 实参部 $::=($ 参数结合{,参数结合})
 99. 参数结合 $::=[$ 形参=>] $]实参$
 100. 形参 $::=$ 参数简名
 101. 实参 $::=$ 表达式|变量名|类型标志(变量名)
 102. 返回语句 $::=return$ 【表达式】;

第8章

103. 程序包说明 $::=$ 程序包规范式说明;
 104. 程序包规范式说明 $::=package$ 标识符 is

{基本说明项}

【private

{基本说明项}】

end 【程序包简名】



105. 程序包体 ::= package body 程序包简名 is

【说明部】

【begin

语句序列

【exception

异常处理程序段

{异常处理程序段}】】

end 【程序包简名】;

106. 说明部 ::= {基本说明项} | {以后说明项}

107. 基本说明项 ::= 基本说明 | 表示子句 | use 子句

108. 以后说明项 ::= 体 | 子程序说明 | 程序包说明 | 任务说明

| 类属说明 | use 子句 | 类属实例

109. 体 ::= 正文体 | 体残根

110. 正文体 ::= 子程序体 | 程序包体 | 任务体

111. use 子句 ::= use 程序包名 {, 程序包名};

112. 编译 ::= {编译单位}

113. 编译单位 ::= 关联子句库单位 | 关联子句次级程序单位

114. 库单位 ::= 子程序说明 | 程序包说明 | 类属说明 | 类属实例 | 子程序体

115. 次级程序单位 ::= 库单位体 | 子单位

116. 库单位体 ::= 子程序体 | 程序包体

117. 关联子句 ::= {with 子句 {use 子句}}

118. with 子句 ::= with 程序单位简名 {, 程序单位简名};

119. 体残根 ::= 子程序规范式说明 is separate;

| package body 程序包 is separate;

| task body 任务简名 is separate;

120. 子单位 ::= separate(父程序单位名) 正文体

121. 换名说明 ::= 标识符:类型标志 renames 对象名;

| 标识符:exception renames 异常名;

| package 标识符 renames 程序包名;

| 子程序规范式说明 renames 子程序或入口名;

第9章

122. 私有类型说明 ::= type 标识符 【判别式部分】 is 【limited】 private;

123. 延期常量 ::= 标识符表 constant 类型标志;

第10章

124. 异常处理段 ::= when 异常选择 { | 异常选择 } => 语句序列

125. 异常选择 ::= 异常名 | others

126. 异常说明 ::= 标识符表 : exception ;

127. 引发语句 ::= raise 【异常名】 ;

第11章

128. 判别式部分 ::= (判别式规范式说明 {, 判别式规范式说明})

129. 判别式规范式说明 ::= 标识符表 : 类型标志【:= 表达式】

130. 判别式约束 ::= (判别式结合 {, 判别式结合})

131. 判别式结合 ::= 【判别式简名 {, 判别式简名} =>】 表达式

132. 变体部分 ::= case 判别式简名 is

 变体

 { 变体 }

 end case ;

133. 变体 ::= when 选择 {, 选择} => 分量表

134. 存取类型定义 ::= access 子类型表示

135. 非完整类型说明 ::= type 标识符【判别式部分】 ;

136. 分配符 ::= new 子类型表示 | new 受限表达式

137. 派生类型定义 ::= new 子类型表示

第12章

138. 整型定义 ::= 范围约束

139. 实型定义 ::= 浮点约束 | 定点约束

140. 浮点约束 ::= 浮点精度定义【范围约束】

141. 浮点精度定义 ::= digits 静态简单表达式

142. 定点约束 ::= 定点精度定义【范围约束】

143. 定点精度定义 ::= delta 静态简单表达式

第13章

144. 类属说明 ::= 类属规范式说明 ;

145. 类属规范式说明 ::= 类属形式部分子程序规范式说明

 | 类属形式部分程序包规范式说明

146. 类属形式部分 ::= generic{类属参数说明}

147. 类属参数说明 ::= 标识符表 : 【in | out】 类型标志【:= 表达式】 ;

 | type 标识符 is 类属类型定义 ; 私有类型说明

 | with 子程序规范式说明【is 名】 ;

 | with 子程序规范式说明【is <>】 ;

148. 类属参数定义 ::= (<>) | range <> | digits <> | delta <>

 | 数组类型定义

 | 存取类型定义

149. 类属例化 ::= package 标识符 is new 类属程序包名【类属实在部分】 ;

 procedure 标识符 is new 类属过程名【类属实在名】 ;

 | function 标志符 is new 类属函数名【类属实在部分】 ;

150. 类属实在部分 ::= (类属结合 {, 类属结合})

151. 类属结合 ::= 【类属形式部分 =>】类属实在部分

152. 类属形式部分 ::= 参数简名 | 算符符号

153. 类属实在部分 ::= 表达式 | 变量名 | 子程序名 | 入口名 | 类型标志

第14章

154. 任务说明 ::= 任务规范式说明;

155. 任务规范式说明 ::= task 【type】 标识符 【is

{入口说明}

{表示子句}

end 【任务简名】 ;

156. 任务体 ::= task body 任务简名 is

【说明部分】

begin

语句序列

【exception

异常处理段

{异常处理段} ;

end 【任务简名】 ;

157. 入口说明 ::= entry 标识符 【(离散范围)】 【形式部分】 ;

158. 入口调用语句 ::= 入口简名 【实在部分】

159. 接收语句 ::= accept 【入口简名 【(入口下标)】 【形式部分】 do

语句序列

end 【入口简名】 ;

160. 入口下标 ::= 表达式

161. 延迟语句 ::= delay 简单表达式;

162. 选择语句 ::= 选择等待 | 条件入口调用 | 定时入口调用

163. 选择等待 ::= select

选择项

(or

选择项)

【else

语句序列】

end select;

164. 选择项 ::= 【when 条件 =>】 选择等待项

165. 选择等待项 ::= 接收选择 | 延迟选择 | 终止选择

166. 接收选择 ::= 接收语句 【语句序列】

167. 延迟选择 ::= 延迟语句 【语句序列】

168. 终止选择 ::= terminate;

169. 条件入口调用 ::= select

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

入口调用语句
【语句序列】

else

语句序列

end select;

170. 定时入口调用 ::= select

入口调用语句
【语句序列】

or

延迟选择

end select;

171. 中止语句 ::= abort 任务名{,任务名};

第15章

172. 表示子句 ::= 类型表示子句 | 地址子句

173. 类型表示子句 ::= 长度子句 | 枚举表示子句 | 记录表示子句

174. 长度子句 ::= for 属性 use 简单表达式;

175. 枚举表示子句 ::= for 类型简名 use 聚集;

176. 记录表示子句 ::= for 类型简名 use

record 【对齐子句】

(分量子句)

end record;

177. 对齐子句 ::= at mod 静态简单表达式;

178. 分量子句 ::= 分量名 at 静态简单表达式 range 静态范围;

179. 地址子句 ::= for 简名单 use at 简单表达式;

180. 代码语句 ::= 类型标志'记录聚集;

附录5 保留字

本附录列出了 Ada 的保留字，并给出了描述它们的章节。

begin	4. 2, 7. 1, 8. 1, 14. 1
body	8. 1, 14. 1
case	5. 2, 11. 2
constant	4. 1
declare	4. 2
delay	14. 3
delta	12. 4, 13. 2
digits	12. 3, 13. 2
do	14. 2
else	4. 9, 5. 1, 14. 4
elsif	5. 1
end	4. 2, 7. 1, 8. 1, 14. 1, 14. 2
entry	14. 2
exception	10. 1
exit	5. 3
for	5. 3, 15. 4
function	7. 1
generic	13. 1
goto	5. 4
if	5. 1
in	4. 9, 5. 3, 7. 3
is	4. 3, 5. 2, 7. 1, 8. 1, 11. 2, 14. 1
limited	9. 2, 13. 2
loop	5. 3
mod	4. 5, 15. 4

超星阅览器提醒您：
使用本复制品
请尊重相关知识产权！

new	11. 3, 11. 8, 13. 1
not	4. 7, 4. 9
null	5. 2, 6. 5, 11. 2, 11. 3
of	6. 1
or	4. 6, 4. 9, 14. 4
others	5. 2, 6. 2, 10. 1
out	7. 3
package	8. 1
pragma	2. 5
private	9. 1, 13. 1
procedure	7. 3
raise	10. 2
range	4. 4, 6. 2, 13. 2, 15. 4
record	6. 5
rem	4. 5
renames	8. 5
return	7. 1, 7. 3, 14. 2
reverse	5. 3
select	14. 4
separate	8. 3
subtype	4. 4
task	14. 1
terminate	14. 6
then	4. 9, 5. 1
type	4. 3
use	8. 1, 15. 4
when	5. 2, 5. 3, 10. 1, 11. 2, 14. 4
while	5. 3
with	8. 2, 13. 3
xor	4. 7

其中保留字 delta, digits 和 range 也可用作属性。

附录6 术语汇编

存取类型 存取类型的值(存取值)可为空值 `null`, 或为标示分配符创建的对象的值。被标示的对象可通过存取值读到和更新。存取类型说明规定了本存取类型值所标示的对象的类型。参见收集。

聚集 对聚集求值得到一个组合类型值, 该值由该类型的每个分量的值组成。位置结合或命名结合用来表示值与分量的结合。

分配符 对分配符求值创建一个对象, 并返回标示该对象的存取值。

数组类型 数组类型的值由具有相同子类型的分量组成。每个分量由一个下标(对于一维数组)或下标序列(对于多维数组)来区分。每个下标必须是离散类型值且不得超出正确的下标范围。

赋值 赋值是以新值置换变量当前值的操作。赋值语句规定变量在左, 求值表达式在右。

属性 对属性求值得到命名实体的预定义特征。某些属性为函数。

分程序语句 分程序语句为单条语句。它可包含语句序列, 也可含有说明部和异常处理段, 但它们的作用仅限于本分程序语句内。

体 体定义了子程序、程序包, 或任务的执行。体残根是体的一种形式, 它表示以分别单独编译子单位的形式来定义执行。

收集 收集为由分配符求值而创建的存取类型的对象的全集。

编译单位 编译单位是作为独立文本交付编译程序的程序单位的说明或体, 其前面冠以一个关联子句(`with`子句), 用来指明它所依赖的其它编译单位。

分量 分量即较大值的部分值, 或较大对象的部分对象。

组合类型 组合类型为其值具有分量的类型。组合类型有两种: 数组类型和记录类型。

约束 约束确定某类型值的子集, 该子集中的值满足本约束。

说明 说明用来结合标识符(或其它表示法)和实体, 该结合在称作说明作用域的文本范围内有效。在说明作用域内, 可以用该标识符来指示说明结合的实体。这时该标识符称作其实体的简单名, 简单名即为实体的表示。

说明部 说明部为说明序列。它可含有子程序体和表示子句等相关信息。

派生类型 派生类型的运算和值是某存在类型翻版。这个存在类型叫做该派生类型的父类型。

离散类型 离散类型具有一个包含不同值的有序集合。枚举类型和整数类型为离散类型。离散类型可用于下标和重复, 也可用于 `case` 语句和记录变体的选择。

判别式 判别式为某对象的特别分量或记录类型的值。其它分量的类型(子类型), 甚至它们的出现或缺省依赖于判别式的值。

判别式约束 记录类型或私有类型上的判别式约束规定了该类型每个判别式的值。

确立 说明的确立是说明达到其效果(如创建对象)的过程, 该过程在程序执行期间完成。

实体 实体可在程序中被命名或表示出来。对象、类型、值、程序单位等都是实体。

入口 入口用于任务间的通信。从外观上看，入口调用同子程序调用是一样的，入口的内部动作由一个或多个接收语句完成，这些语句规定了调用一个入口所必须完成的操作。

枚举类型 枚举类型为离散类型，其值由类型说明中显式列出的枚举直接量表示。这些枚举直接量可以是标识符或字符直接量。

求值 表达式的求值即计算表达式值的过程。该过程在程序执行时进行。

异常 异常为可能在程序执行期间出现的错误事件。引发异常就是放弃程序正常执行而产生错误发生的信号。异常处理段为一段程序，它规定了对异常所要作出的应答。执行这段程序即称为处理异常。

扩充名 扩充名用来表示某些结构内直接说明的实体。扩充名具有选择分量的形式：其前缀表示结构（分程序、循环或接收语句）；选择部分即实体的简单名。

表达式 表达式定义了值的计算。

类属程序单位 类属程序单位即一类子程序或程序包的样板程序单位。使用该样板创建的子程序或程序包称为该类属程序单位的实例。类属例化是创建实例的说明。类属程序单位作为子程序或程序包来书写，但其规范式说明前要加上可以说明类属形参的类属形式部分。类属形参可以是类型、子程序或对象。

下标约束 数组类型的下标约束规定了数组类型的每个下标范围的下限和上限。

下标分量 下标分量表示数组的分量。它为包含表达式的名的形式，这里的表达式规定了数组分量的下标值。下标分量也可表示一组入口中的某个入口。

整型 整型为离散类型，其值为规定范围内的所有整数。

词法元素 词法元素即标识符、直接量、定界符或注释。

受限类型 受限类型的赋值或等价操作都被显式地说明。所有任务类型均为受限类型。私有类型可定义成受限的。

直接量 直接量用字母或其它字符在字面上表示一个值，它可以是一个数、一个枚举值、一个字符或一个字符串。

样板数 样板数是实型可精确表示的值。实型的运算根据相应样板数上的运算来定义。样板数及其运算的特性是实型的所有实现都要予以保留的最小特性。

名 名为代表某实体的结构，即名表示实体，实体为名的含义。可参见说明与前缀。

命名结合 命名结合通过给位置命名来规定表中某一项与一个或多个位置的结合。

对象 对象句令值 程序通过确立对象说明或对分配符值来创建对象 说明或分配符值

相关的实体(如:一些类型、这些类型的对象和带这些类型参数的子程序)。程序包由程序包规范式说明和程序包体组成。程序包规范式说明有一可见部分,其包含那些能在此程序包外显式使用的所有实体的说明。它还可有一私有部分,其包含那些与用户无关的可见实体的规范式说明的构造细节。程序包体包含那些程序包规范式说明中说明的子程序(也可是任务和其它程序包)的实现。程序包是一种程序单位。

参数 参数是与子程序、入口或类属程序单位有关的命名实体,用于与相应子程序体、接收语句或类属体通信。形参为用来表示这些体内命名实体的标识符。实参为子程序调用、入口调用或类属例化中与相应形参结合的特定实体。参数传递型有 **in** 型、**out** 型和 **inout** 型,它们指定是由实参传入值给形参还是由形参传出值给实参,或传入传出都允许。实参与形参的结合形式有命名结合和位置结合。

位置结合 位置结合即用某一位置来指定若干项来规定表中项与位置的结合。杂注用来给编译程序传递信息。

前缀 前缀用作某些名的前面部分。前缀可以是函数调用或名。

私有类型 私有类型的结构和值的集合都被显式地定义。用户不能直接利用其结构。对私有类型值的访问要通过定义给私有类型的运算集合和它的判别式(如果有的话)进行。私有类型及其运算均在程序包的可见部分,或是类属形式部分中定义。赋值、等价与不等价也可定义给私有类型,但受限私有类型除外。

程序 程序由一组编译单位组成,其中之一称作主程序。程序的执行即为主程序的执行,该主程序可以调用本程序的其它编译单位中说明的子程序。

程序单位 类属程序单位、程序包、子程序或任务程序单位都是程序单位。

受限表达式 受限表达式为前面冠以其类型或子类型表示的表达式。限定表示用于当缺少该限定表示时表达式可能产生二义性的情况(如重载)。

范围 范围是一个纯量类型的相邻值集合,它通过给出上下界来规定。范围中的值属于该范围。

范围约束 类型的范围约束指定一个范围,并因此确定了属于这个范围的该类型值的子集。

实型 实型值表示了实数的近似值。实型有两种:定点类型和浮点类型。

记录类型 记录类型的值通常由具有不同类型的分量组成。对于记录值或记录对象的每个分量,记录类型定义规定了记录内唯一确定它们的标识符。

换名说明 换名说明用来为实体说明另一名字。

汇合 汇合是发生在两个并行任务之间,当一任务调用了另一任务的入口,且后者正在以前者的名义执行相应的接收语句时的相互作用。

表示子句 表示子句提供编译程序怎样选择类型、对象或任务到具体机器特性映象的信息。某些情况下,表示子句能完全确定该映象;另外一些情况下,它们只提供选择该映象的准则。

纯量类型 纯量类型的对象或值不含分量。纯量类型有离散类型和整型。其值是有序的。

作用域 某说明的作用域即该说明起作用的文本范围。

选择分量 选择分量为由前缀和称作选择部分的标识符所组成的名。选择分量用来表示记录分量、入口和存取值标示的对象。选择分量也用作扩充名。

语句 语句规定了程序执行期间完成的一个或多个动作。

子程序 子程序即过程和函数。过程规定了一串动作且通过过程调用来实现。函数也规定了一串动作，它还返回一个结果值，函数调用是一个表达式。子程序说明规定了子程序的名、形参和结果（仅对函数），子程序体规定了子程序的动作。子程序调用给出了与形参结合的实参。

子类型 某类型的子类型用来表示该类型值的子集。这个子集由该类型上的限定确定。子类型值集中的每个值都属于该子类型且满足确定该子类型的限定。

任务 任务是一种可同其它程序单位并发执行的程序单位。任务规范式说明规定了任务名及其入口名和形参，任务体规定了任务的执行。任务类型可说明若干该类型的相似任务。任务类型的值为任务的标示。

类型 类型用来指定一组值和这些值上的各种运算。类型定义是引进一个类型的语言结构。类型说明给类型以名。特殊的类型有存取类型、数组类型、私有类型、记录类型、纯量类型和任务类型。

use 子句 use 子句用来使程序包可见部的说明直接可见。

变体 记录的变体部分根据记录判别式指定候选记录分量，判别式的每一个值都确定了变体部分的一个特定候选。

可见度 如果某实体为一个标识符出现在程序中某给定点的可接受的含义，则该实体到这个标识符的说明在给定点是可见的。这个说明可通过在选择分量中的选择部分给出选择或在命名结合中的名字上给出选择而可见。如果这个标识符仅有一个含义，则该说明直接可见。