Ada 程序设计

孙乐昌　梁亚声
李冬青　陆余良　编译
王世明　审校

中国科学技术大学出版社

Ada 程 序 设 计

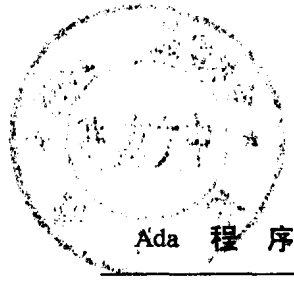
孙乐昌　梁亚声
李冬青　陆余良　编译
王世明　　审校

中国科学技术大学出版社出版
1991·合肥

内 容 简 介

本书从 Ada 的设计背景出发，全面系统地介绍了 Ada 语言的各个方面。突出阐明至 80 年代程序设计方法学与软件工程的一些新概念、新方法和新思维在 Ada 语言中的具体体现，并参照其它程序设计语言进行比较。包括：对数据类型的本质性论述；程序包论述中引进其它语言的特点作比较，全面地论述 Ada 的并发程序设计设施，以及如何利用这些设施编写并发程序、精心选例说明异常机制。对 Ada 语言的结构、特点、设计和使用进行了深入浅出和透彻地讲述。给出大量具体程序实例佐证论述。每章都配以习题和要点。附录给出 Ada 规则。

本书可用作计算机和自动化等专业本科生、研究生的教材，亦可作为教师及各计算机应用领域的科研、设计人员的参考书。也可作为学习软件工程的辅助用书。



Ada 程 序 设 计

孙乐昌　梁亚声
李冬青　陆余良　编译
王世明　　审校

责任编辑、封面设计：蓝海柯
中国科学技术大学出版社出版
（安徽省合肥市金寨路 96 号，邮政编码 230026）
中国人民解放军电子工程学院印刷所印刷
开本：787×1092/16　印张：17.5　字数：432 千
1991 年 6 月第 1 版　1991 年 6 月第 1 次印刷
印数：1—4000 册
ISBN7-312-00285-4/TP・30　　定价：8.00 元

前　言

Ada 语言作为至 80 年代程序设计语言的总结和代表，在美国被用于取代与军方有关领域正使用的一切其它高级程序设计语言。并越来越多地用于工业、商业、科学研究和大学教学领域，在美国及欧洲得到广泛的应用。

目前我国计算机技术领域，已日渐出现使用该语言的势头，并将很快得到普及。

Ada 直接体现了许多现代软件设计方法学。在 Ada 的语言成份、结构和机制中充分保证了软件工程原理的实现。它不仅鼓励使用好的程序设计方法，并能用软件工程原理强制进行程序设计实践。从这个意义上讲，Ada 堪称开发软件系统的优秀工具。

学习 Ada 语言和 Ada 程序设计本身就是一个令人激动的机会。它使学习者能够站在程序设计的现代立场上，领略软件工程的最新理论和 Ada 及 Ada 程序设计支持环境(APSE)代表的 Ada 文化，并在程序设计能力、水平、思维和素养等诸方面得到明显的提高。对此，我们亦有很深的体会。

由美国约翰·帮纳斯所著的《Ada 程序设计》一书，是近年在欧美有关 Ada 的诸多书籍中的一部十分畅销的计算机科技用书，多次再版。该书主要特点是，全面系统地介绍了 Ada 语言的各个方面。特别是从软件工程的高度将 Ada 语言本身以及它的设计背景，Ada 文化，Ada 思维方法，语言环境，编程技巧和隐含在其中的现代软件方法学都予以充分的叙述。讲述透彻，深入浅出。

作为本书的编译者，我们曾在国外接触到 Ada，并使用该语言实现了中、小规模的系统。为此，我们在约翰·帮纳斯所著的《Ada 程序设计》一书的基础上，加进自己的学习、使用 Ada 的体会和经验，编译了这本书，献给读者。希望能够对读者有所帮助。

由于编译者水平有限，错误之处在所难免，欢迎读者予以批评指正。

作为编译者的同事，汤波同志曾参于部分章节和内容的翻译工作，借此予以感谢。

本书的编译和出版过程曾得到解放军电子工程学院的王岚丽、张世男同志的关心和支持，给予积极的协助，在此一并致谢。

编 译 者

1990.12.

目　录

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6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　66。
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 Honewell 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，并且参于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设计员。

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。控制抽象的关键在於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, Virgina, 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_LENE;
    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，问 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);
```