计算机程序结构及其描述

袁淑君　等译

上海交通大学出版社

高等学校教学参考用书

计算机程序结构及其描述

〔美〕 Abelson, Sussman 著
袁淑君等 译

上海交通大学出版社

内 容 简 介

本书介绍程序结构及其描述，它用人工智能程序设计最常用的语言 Lisp 作为描述语言。全书共分五章。前两章讲解程序设计中处理的两种对象：过程和数据，并建立了过程抽象和数据抽象的概念。第三章阐明组织一个大型程序系统的两种策略。第四章指出不断建立新语言的重要性及建立新语言的方法。第五章以类似硬件的观点，设计一台可执行任何过程的寄存器机器，并提出了一些实现递归的程序结构方法及一种描述寄存器机器设计的语言。书中每章都附有练习。

本书可作为大专院校计算机有关专业的教材或参考书，也可作为计算机工作者的参考书。

计算机程序结构及其描述

上海交通大学出版社出版

（淮海中路 1984 弄 19 号）

新华书店上海发行所发行

常熟市梅李印刷厂印装

开本 787×1092毫米 1/16 印张 20.5 字数 509,000

1988 年 8 月第 1 版 1988 年 8 月第 1 次印刷

印数：1—3,500

ISBN 7-313-00192-4/TP39 科技书目：176—294

定价：3.40元

译 者 的 话

《计算机程序结构及其描述》是美国麻省理工学院(MIT)计算机科学系的教材之一，是MIT电子工程和计算机专业学生的一门必修课。本书是一本关于程序设计的书，但讲述的内容不是特定的程序设计语言的语法。它采用了描述程序结构的优秀工具——Lisp语言，来表达程序思维，描述程序中处理的两种对象：过程和数据。采用组合和抽象手段，组合基本元素以形成复合对象(复合数据、复合过程)，抽象复合对象以形成高层积木块(高层过程、抽象数据)，增强了程序设计的模块性，使程序易于设计、维护和修改。本书还论述了大型程序系统的结构模块化的两种策略：对象处理方式和流处理方式。同时，本书对程序设计语言的控制结构与逻辑运算系统操作合并产生的逻辑程序设计也作了较详细的讲解。本书的另一个愿望是建立一种概念：计算机语言不仅是计算机借以执行操作的工具，而且是表达方法论概念的新颖、有效的媒介。因此，书写程序的目的首先是给人看的。

参加本书翻译的有袁淑君(前言和第一章)，李伟(第二章)，崔靖(第三章)，崔佑铣(第四章)，杨讴(第五章)，由袁淑君主译并校阅全书。由于译者水平有限，译文中一定有不妥和错误之处，敬请指正。

译者

1987年10月

前　言

《计算机程序结构及其描述》是MIT(Massachusetts Institute of Technology)计算机科学的一门入门科目。在MIT，对所有主修电子工程或计算机科学的学生来说这是一门必修课，是四门公共核心课程之一。其他三门是：两门电路和线性系统的课程，一门数字系统设计的课程。我们已注意到这门课程自1978年以来的发展。从1980年秋季开始，每年在600～700名学生中用现在的方式讲授了这个内容。这些学生的大多数事先几乎没有受过正规的计算训练，但是，大多数学生用过计算机，少数学生已有广泛的程序设计经验或硬件设计经验。

我们对这门计算机科学的入门课程的设计，考虑了两件重要事情。第一，我们希望建立一种概念，计算机语言不仅是使计算机执行操作的手段，而且是表达方法论概念的新颖、有效的媒介。因为书写程序的目的首先必须是要给人看，其次才是给机器执行。第二，我们相信，入门课程要讲述的基本内容既不是特定的程序设计语言结构的语法，也不是有效地计算特定函数的巧妙算法，更不是算法的数学分析和计算基础，而是用来控制大型软件系统的知识复杂性技术。

我们的目标是，当学生学完这门课之后，对程序设计风格基础和程序设计美学有良好的感性认识；他们应掌握控制大系统中复杂性的主要技术；他们应能阅读50页长的程序。如果程序是用典型风格编写的话，他们应当知道在任何时刻不要读什么，什么不需要理解；他们应当有把握修改程序，并保持原作者的精神和风格。

我们探讨这门课程的基础在于我们深信“计算机科学”不仅是一门科学，这门课程对计算机的重要性毋需多言。计算机革命是我们思考方式，以及用来表达我们思考的一场革命。这场革命的实质，最恰当的可能是所谓程序设计认识论的出现——相对于经典数学更多采用描述的观点，程序设计认识论则强调知识结构的研究。数学提供一种精确处理“是什么”的概念的结构，计算则为精确处理“怎样做”的概念提供结构。

在我们讲授的内容中，采用了程序设计语言Lisp的一种方言。我们以前并没有正规讲授过这种语言，然而学生在两三天内就学会了。这是Lisp这类语言的一个突出优点：只有很少几种形成复合表示式的方法，并且几乎没有语法结构，所有规范化的特性可在一个小时内讲完，这类似于国际象棋规则。在一个短时间之后，我们忘记了语言的语法细节(因为没有语法)并进入实际问题——计划我们需要计算什么，怎样把问题分解成可管理的部分，对各部分怎样进行工作。Lisp的另一优点是，它比我们所知道的任何其他语言支持更多的对程序的模块化划分的大范围策略(large—scale strategies)。我们可以建立过程和数据的抽象；我们可以使用高层函数以获得通用模型；我们可以用流(streams)和延迟(delay)求值连接程序的各部分；我们可以容易地实现嵌套语言。所有这些都被置于一个交互式环境中，这个环境极好地支持分步程序设计、构造、测试和调整。我们感谢各个时期的Lisp奇才们，首先是John. McCarthy。他创造了空前有效、雅致而精巧的工具。

Scheme，我们使用的Lisp方言，企图把Lisp和Algol的功能和风格结合起来。我们从Lisp得到元语言的功能，这来源于简单的语法程序与数据对象的统一表示，收集无用单元的堆式

数据分配。从Algol，我们获得词法辖域和块结构，这些是程序设计语言设计的倡导者们的贡献。这些倡导者都是Algol委员会成员。我们希望提到John Reynolds和Peter Landin，他们洞察了Church的λ积分与程序设计语言的结构之间的关系。我们也认识到对那些数学家们欠下的情谊，他们在计算机出现之前花了几十年的时间寻找这个领域。这些先驱者包括Alonzo Church，Barkley Rosser，Stephen Kleene和Haskell Curry。

ALan J. Perlis

目　录

第一章　建立过程抽象

思维活动的简单概念方式主要有三种：1.把几个简单概念组合成一个复合概念，从而建立全部复杂概念。2.汇集两个概念，无论它们是简单的还是复杂的，并且相互影响以便同时观察它们，但是不把它们合并为一个概念，由此，得到全部关系概念。3.把它们从共存于它们的真实存在中的所有其他概念中分离出来：这就是所谓抽象，从而建立一般概念。

我们准备讨论计算进程的概念。计算进程是栖息于计算机的抽象事物。随着进程的发展，它们处理其他称作数据的抽象事物。进程的发展是受程序的规则模式支配的。于是，人们为了指挥进程而建立各种程序。实际上，我们正是用自己的“咒语”演变出计算机的精灵。

计算进程的确很像巫师的幽灵，它既看不见，也摸不着，完全不是由物质构成的。然而，它的确是现实的，它可以进行智力劳动，还能回答各种问题；它能支付银行存款或控制工厂里的机器人，从而影响世界。我们用以实现进程的程序就像巫师的咒语。这些程序是由深奥莫测的程序设计语言中的符号表达式精心构造的，这些程序设计语言规定我们的进程执行的任务。

在正常工作的计算机中，计算进程准确地执行程序。因此，像巫师的学徒一样，新程序员必须学会推测他们变出的结果。因为，即使程序中存在很小的错误(通常称为故障)，也可能导致不可预料的后果。

幸运的是，学会编程序比学习巫术的危险要小得多。因为，我们对付的精灵可以方便地用一种安全的方法控制。然而，现实世界的程序设计则需要细心、经验和智慧。例如，在计算机辅助设计程序中，有一个小故障，就可能导致飞机或水坝的灾难性事故，或者工业机器人的自身毁灭。

主软件工程师应该有能力组织程序。他的进程会执行预期的任务。他能事先设想他的系统的性能，知道如何设计他的程序，使不可预料的问题不会导致灾难性的后果。果真出现问题时，他能排除程序中的错误。设计良好的计算机系统和设计良好的汽车及核反应堆一样，是模块式的设计。这样，各部分能独立的构成、替换和调整。

用Lisp编程序

我们需要一种适于描述进程的语言。为此，我们打算使用程序设计语言Lisp。正如我们通常用自然语言(如英语、法语和日语)表达日常思维，用数学符号表达量化现象一样，我们将用Lisp表达我们的程序思维。Lisp以一种推理形式被发明于20世纪50年代末期。它是关于用做计算模型的逻辑表达式的推理形式，这些逻辑表达式叫做递归方程。Lisp语言是John McCarthy构想出来的，并且以他的论文《符号表达式的递归函数及它们的机器计算》为基础。

尽管Lisp开端像数学体系，但它是一种实用的程序设计语言。一个Lisp解释器是为实现

用Lisp描述的进程的一台机器。第一个Lisp解释器是McCarthy在MIT电子学研究实验室人工智能小组及MIT计算中心的同事和学生的帮助下实现的注1。Lisp是LIST Processing 的缩写，设计目标是提供符号处理能力，以着手解决如代数表达式的符号微分和积分这样一类程序设计问题。为此，它包含了新的数据对象：称为子和表列，使Lisp明显地不同于这个时期的所有其他语言。

Lisp不是预定的设计工作的产品。相反，它是根据用户的需要出于实际的实现考虑，用实验的方法非正规地逐渐形成的。Lisp的非正规发展持续了好多年。据说，Lisp用户协会一直反对公布任何“官方的”语言定义的企图。这个发展以及原始构思的灵活和优雅，使Lisp，这个当今广泛使用的第二古老的语言(仅次于FORTRAN)，不断地适应于包括程序设计的最现代的概念。因此，Lisp现在已经是一簇方言。虽然它们共用大部分原始特征，但在一些重要方面，却可能彼此不同。本书使用的Lisp方言叫做Scheme注2。

由于Lisp的实验特征，以及它着眼于符号处理的特点，开始，它对于数字计算来说效率非常低，至少比Fortran效率低。然而，长年累月，为了把程序翻译成机器码，而且能像任何其他高级语言生成的代码一样，有效地执行数字计算，Lisp编译器一直在发展着。尽管如此，Lisp仍未丢掉“无可救药的效率低的语言”这个名声。因此，它的使用仍然局限在几个研究室中。

既然Lisp不是一种流行的语言，为什么我们要用它作为我们讨论程序设计的框架呢？这是因为Lisp语言具有独特的性能，使它成为研究重要的程序结构的极好工具，也是使这些结构与支撑它们语言特征相联系的极好工具。其中显著的特征是：进程的Lisp描述(叫做过程)可以表示为Lisp的数据，并作为Lisp数据来处理。这个特征的重要性在于它有强有力的程序设计技术，依靠这种能力可以使“被动”数据和“活动”进程之间的传统区别变得模糊。正如我们将会发现的，Lisp把过程当作数据处理的灵活性，使它成为研究这些技术的现有的最方便的一种语言。把过程表示为数据的能力也使Lisp成为编写这样一些程序的极好的语言，被编写的程序必须把其他程序当作数据操纵，比如支撑计算机语言的解释程序和编译程序。前述的这些理由，使得用Lisp编程成为非常有趣的事。

§1.1 程序设计要素

一种功能强的程序设计语言，不仅只是指挥计算机执行各种任务的工具，它还作为一种结构。在这种结构内，我们组织自己的过程概念。因此，当我们描述一种语言时，应特别注意该语言为组合简单概念而形成一个更复杂概念提供的手段。为了完成这个任务，每一种功

注1：Lisp 1程序员手册1960年问世，Lisp 1.5程序员手册(见McCarthy 1965)1962年出版。Lisp的早期历史在McCarthy 1978中描述。

注2：被用来编写20世纪70年代最主要Lisp程序的两个方言是MacLisp(Moon1978; Pitman1983)和Inter Lisp(Teitelman 1974)。前者是在MIT的MAC项目上开发的，后者是在Bolt Beranek, Newman公司，以及Xerox Palo Alto研究中心开发的。可移植的标准Lisp(Portable Standard Lisp) (Hearn1969, Griss1981)是另一种Lisp方言，它的设计目标是易于在不同机器之间移植，而且已开始成为广泛可用的语言。Mac Lisp还产生了若干子方言。比如，在Berkeley的California大学开发的Franz Lisp和在MIT人工智能实验室，以专用处理器为基准，为高效运行Lisp而设计的Zetalisp(Moon1981)。另一个正在开发中的方言Common Lisp是计划用作支持将来的生产Lisp系统(Steele1975)。它是Goy Lewis Steele Jr和MIT人工智能实验室的Gerald Jay Sussman提出的，后来在MIT为教学使用而重新被实现。

能强的语言都具有三种机制：

1．原始表达式：它描述与语言有关的最简单的实体；

2．组合方法：用这种方法，由较简单的表达式构造复合表达式；

3．抽象手段：使用抽象，复合对象可以作为命名和操纵单位。

在程序设计中，我们处理两类对象：过程和数据。（稍后，我们将发现，它们实在没有什么差别）简略地说，数据是“素材”，它表示我们要操纵的对象；过程是操纵这些数据的规则描述。因此，任何一种功能强的程序设计语言应能描述本原数据和原始过程，并且应该具有组合与抽象过程和数据的方法。

为了能把我们的注意力集中在建立过程的规则上，本章将只处理简单的数值计算[注3]。在以后几章中，我们将看到，这些规则，也能使我们建立操纵复合数据的过程。

1.1.1 表达式

为了开始进行程序设计，一种容易的方法是考查几个与Lisp的Scheme方言的解释器的典型对话。设想你正坐在计算机的终端前，解释器在一空白行的开始显示一提示：

```
==>
```

表明它已准备好为你服务。如果打入一个表达式作为回答，那么，解释器对该表达式求值，并显示其结果。

你可能打入的一种基本表达式是一个数。（更确切地说，你打入的表达式由十进制数组成）如果给Lisp一个数：

```
==>486
```

解释器将打印出：[注4]

```
486
```

表示多个数的多个表达式可与表示基本过程（如+或*）的表达式组合，形成一个复合表达式，它描述过程对那些数的作用。如：

```
==>(+137 349)
486
==>(-1000 334)
666
==>(*5 99)
495
==>(/ 10 5)
2
==>(/ 10 6)
1.66667
==>(+ 2.7 10)
12.7
```

上述的表达式，是用圆括弧定界的表达式表列，被称为组合式。表列最左边的元素为操作符，其他元素称为操作数。把操作符规定的过程施加于具有操作数值的变元上，就获得了

注3：作为简单数据的数涉及到的典型问题有：整数和实数，比如2和2.00，这涉及舍入误差和截去误差的一堆问题。因为本书的中心是大规模程序设计而不是数值技术，因此，我们打算忽略这些问题。凡是可能的地方，Lisp的Scheme方言都不区别整数和实数。

注4：本书中，当我们希望强调用户的输入和解释器的响应之间的区别时，将用斜体字符显示后者。

组合式的值。

操作符置于操作数左边的约定，称为前缀表示法。开始，可能有点令人不解，因为它大大地违反了通常的数学惯例。但是，前缀表示法有几个优点。首先，能使过程适应取任意多个变元，如下列所示：

```
==>(+ 21 35 12 7)
75
==>(* 25 4 12)
1200
```

不会出现二义性。因为操作符始终是最左元素，并且整个组合式用圆括弧定界。

前缀表示法的第二个优点是：它扩展直接的方法，使组合式能嵌套，即组合式的元素还是组合式：

```
==>(+ (* 3 5) (- 10 6))
19
```

原则上，对这样的嵌套深度没有限制，因而对Lisp解释器能够求值的表达式的总的复杂程度也没有限制。人们被相当简单的表达式弄得手足无措。如：

```
(+ (* 3 (+ (* 2 4) (+ 3 5))) (+ (- 10 7) 6))
```

而解释器却毫不费力地就会求出其值是57。我们可以把表达式书写成下述形式以得到帮助：

```
(+ (* 3
      (+ (* 2 4)
         (+ 3 5)))
   (+ (-10 7)
      6))
```

后一种格式约定，称为美化打印(Pretty-printing)。以这种方式书写长组合式时，操作数垂直排列成凹槽形，它清楚地显示出表达式的结构。(Lisp系统提供一些典型特征，帮助用户格式化表达式。两个特别有用的特征是：1.每当开始打印新的一行时，自动地缩进到适当的美化打印位置；2.每当打入右括弧时，集中注意力于左括弧符的匹配)。

即使对于复杂的表达式，解释器总是以同样基本的循环方式操作：从终端读一个表达式，对该表达式求值，打印其结果。这种操作方式，常被说成解释器以读—求值—打印循环(read—eval—print)方式运行。特别地，注意到不需要明确的命令解释器打印表达式的值。

1.1.2 命名和环境

程序设计语言的关键之一是为用名字引用计算对象提供一种方法。我们说名字标识一个变量，变量的值是对象。

在Lisp的Scheme方言中，给事物命名的操作符叫定义。打入：

```
==>(define size 2)
size
```

使解释器把值2与名字size相联系。注意：
解释器打印定义的名字回答组合式define。(打印的符号实际上是define组合式的值。在Lisp中，人们约定所有表达式都只具有一个值)。

在名字size定义为数2之后，我们便可以用它来引用值2：

```
==>size
```

```
2
==>( * 5 size)
10
```

下面还有几个使用define的例子：

```
==>(define pi 3.14159)
pi
==>(define radius 10)
radius
==>( * pi ( * radius radius))
314.159
==>(define circumference ( * 2 pi radius))
circumference
==>circumference
62.8318
```

define是该语言的最简的抽象方法，因为它使我们能用简单的名字引用复合操作的结果，如上面计算圆周长的例子。通常，计算对象可以具有非常复杂的结构。每当我们想使用它们时，必须记住并重复它们的细节，那将是极不方便的。的确，复杂的程序是逐步建立复杂性渐增的计算对象构成的，解释器使这种步进程序结构特别方便。因为，我们可以在连续对话中逐步地建立名字-对象联系。这个特性促进了不断增长的程序的开发和测试，并且大都造成这样的事实：Lisp程序通常由大量相当简单的过程组成。

值与符号相联系的可能性应该是很清楚的。检索它们意味着解释器必须保持某种存贮器，以便记住这些名字-对象对。这个存贮器叫做环境。（稍后，我们将会看到一个计算可能涉及若干个不同的环境）。

1.1.3 对组合的求值

本章我们的一个目标是用过程独立地思考问题。作为一个恰当的例子，考虑在对组合求值时，Lisp解释器仿效一个过程。对迄今为止我们讨论过的表达式，简单地描述求值进程。

为了对组合（而不是定义）求值，要执行下列两步：

1．对组合的各子表达式求值；

2．把过程施加于变元。这里，过程系指最左边的子表达式（操作符）的值，而变元系指其余的子表达式（操作数）的值。

这个规则虽然简单，它仍说明进程的某些重要论点。首先，我们注意到步骤1指出，为了完成组合的求值进程，首先必须执行组合的每一个元素的求值进程。因此，实际上求值规则是递归的。也就是说，作为步骤的一步，仍需调用这个规则的本身。

注意，递归的概念能够在深度嵌套的组合情况下，简洁地表示用别的方法可能视为相当复杂的过程。例如，计算：

```
(* (+ 2 (* 4 6))
   (+ 3 5 7))
```

需要将这个求值规则施加于四个不同的组合。以树的形式描述组合，我们可以得到这个进程的一个图形，如图1.1所示。每一个组合用一个节点表示，始于节点的枝对应于该组合的操作符和操作数。末端节点（即没有枝始于它们的那些节点）或者是操作符，或者是数。观察通

过树求值时，我们可以想象操作数的值从末端节点向上穿透，然后在愈来愈高的层次上组合。通常，我们将发现递归是处理层次的树状对象的非常强有力的技术。事实上，求值规则的“使值向上穿透”的形式就是通常所说的树累加(tree accumulation)进程的一般类型的例子。

其次，我们注意到步骤1的重复应用使我们达到一种程度，这时我们需要求值的，不是组合，而是原始表达式，如数字、内部操作符或其他名字。我们按下述规定处理原始情况：

1．数字的值就是数字本身；

2．内部操作符的值是原始的机器指令序列，它们执行对应的操作；

3．其他名字的值是环境中与那些名字相联系的对象。

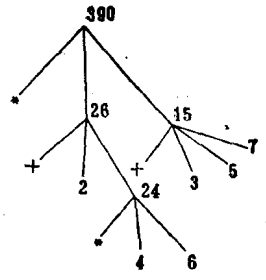

图1.1 表明每一子组合值的树图

像+和*这类符号也包括于全局环境中，并且与它们的“值”(机器指令序列)相联系。我们可以把第二个规则看作第三个规则的一种特殊情况。在确定表达式中符号的含义时，关键的是要注意环境的作用。在像Lisp这样的交互式语言中，对像(+ × 1)这样的表达式，没指定任何环境信息，只说(+ × 1)的值是没有意义的。因为，环境将赋予符号×(甚至给符号+)以实际意义。正如在第三章中我们将会看到的，视环境为提供求值范围的一般概念，对我们理解程序的执行将起重要的作用。

最后注意，define是上述一般的求值规则的一种例外情况。例如，对表达式(define× 3)求值时，并不把define施加于两个变元(一个是符号×的值，另一个是3)，因为define的作用是精确地使×与一个值相联系。

通用求值规则的这些例外被称做特殊形式(special forms)。define是目前我们见到的特殊形式的唯一的例子。但是，我们立刻会遇到其他例子。每一种特殊形式有自己的求值规则。特殊形式及与它们相联系的求值规则构成了程序设计语言的语法。与大多数其他程序设计语言相比，Lisp的语法非常简单。也就是说，表达式的求值规则可以用一个简单的通用规则，加上少量特殊形式的专用规则描述注5。

1.1.4 复合过程

我们确定了Lisp的一些元素，这些元素在任何强功能的程序语言中也一定出现：

1．数和算术操作符是本原数据和原始过程。

2．嵌套组合提供组合操作符的手段。

3．用define使名字与值相联系，提供一种有限的抽象方法。

现在，我们学习过程定义(procedure definition)，这是一种更强的抽象技术。利用这种技术，能够给复合操作命名，然后可以作为一个单位被引用。

注5：这些特殊的语法形式，对于能用更一致的方法书写的东西，仅仅是方便的可供选择的外观结构，有时用peter landin杜撰的成语称它们为语法糖。与掌握其他程序设计语言的用户们相比，Lisp程序员一般地较少关心语法实体。(考查任何一本Pascal手册，注意它有多少篇幅是用于语法描述)这种对语法的轻视，部分地由于Lisp的灵活性，这使Lisp易于改变外部语法。

首先，我们考察求平方的概念。我们可以认为“求某数的**平方**，就是用它乘它自身。”表示如下：

```
(define (square x) (* x x))
```

对此，可以理解如下：

```
(define(squre x )          (*  x   x ))
  ↑     ↑   ↑              ↑   ↑   ↑
  求   平方  某数           乘 某数 自己
```

这里，有一个被命名为square的复合过程。它描述一个实体自乘的操作。给自乘的实体赋予局部名x，x的作用与自然语言中代名词的作用相同。

求define形式的值时，导致指定的过程名与环境中对应的过程定义相联系，解释器打印定义的过程名回答define：

```
==>(define (square x) (* x x))
square
```

过程定义的一般形式是：

```
(define(〈name〉〈formal parameters〉)〈body〉)
```

〈*name*〉是必须与环境中过程定义相联系的符号注6。〈*formal parameters*〉是过程体内用于引用过程的对应变元的名字。〈*body*〉是一个表达式，当以实元代替形参，且过程施加于实元时，它将产生过程应用的值注7。〈*name*〉和〈*formal parameters*〉被聚集在圆括弧内，正如在实际调用定义的过程时它们的形式一样。

我们已定义了square，现在可以使用它：

```
==>(square 21)
441
==>(square (+2 5))
49
==>(square (square 3))
81
```

定义其他过程时，还可用square作积木块。例如，x^2+y^2可以表示为

```
(+ (square x) (square y))
```

我们能更容易地定义一个过程sum—of—squares，给定任意两数作为变元，产生两个数的平方和

```
(define (sum-of-squares x y)
   (+ (square x) (square y)))
==>(sum-of-squares 3 4)
25
```

现在，我们可以用sum—of—squares作为构造更多的过程的积木块：

```
(define (f a)
   (sum-of-squares (+ a 1) (* a 2)))
==>(f 5)
136
```

注6：本书中，我们用尖括弧定界的斜体符号，如〈*name*〉，描述表达式的一般语法。这指示实际使用这样的表达式时，在表达式中要填充的位置(slots)。

注7：更一般地说，过程体是一个表达式序列。在这种情况下，解释器依次对各表达式求值，并且返回最后一个表达式的值作为过程应用的值。