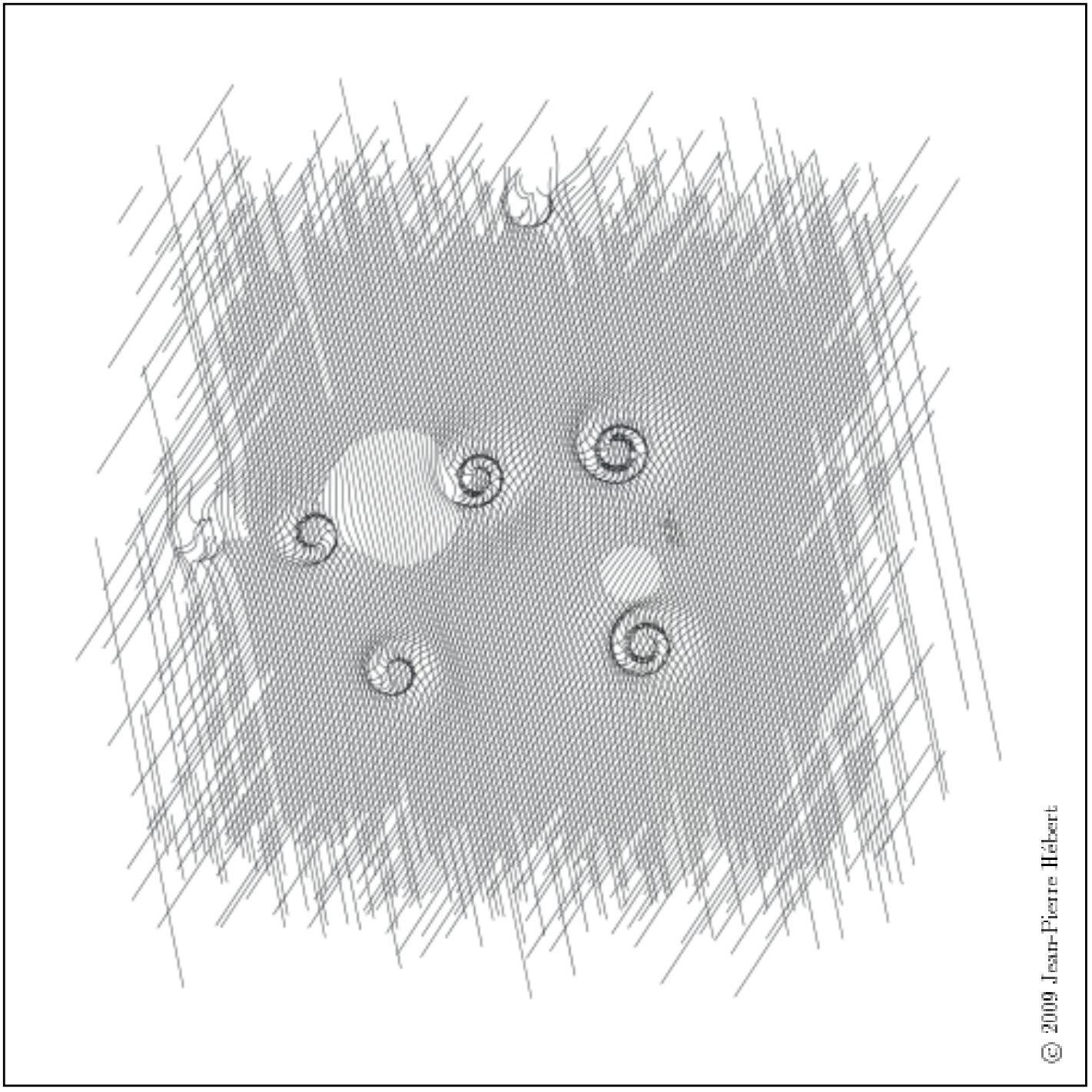


第 2 章。开始

本章是为刚接触该语言的程序员介绍 Scheme 的介绍。如果你坐在一个交互式 Scheme 系统前，边走边尝试这些示例，你将从本章中获得更多。

阅读本章并完成练习后，您应该能够开始使用Scheme。您将学习Scheme程序的语法及其执行方式，以及如何使用简单的数据结构和控制机制。

第 2.1 节. 与方案交互

大多数 Scheme 系统都提供交互式编程环境，可简化程序开发和实验。与 Scheme 最简单的交互遵循“读取-评估-打印”循环。程序（通常称为读取-评估-打印循环或 REPL）读取您在键盘上键入的每个表达式，对其进行计算并打印其值。

使用交互式 Scheme 系统，您可以在键盘上键入表达式并立即查看其值。您可以定义一个过程并将其应用于参数以查看其工作原理。您甚至可以键入由一组过程定义组成的整个程序，并在不离开系统的情况下对其进行测试。当您的程序开始变长时，将其键入文件（使用文本编辑器），加载文件并以交互方式进行测试会更方便。在大多数 Scheme 系统中，可能会使用非标准过程加载来加载文件，该加载过程会采用命名该文件的字符串参数。在文件中准备程序有几个优点：您有机会更仔细地编写程序，可以在不重新键入程序的情况下更正错误，并且可以保留副本以供以后使用。大多数 Scheme 实现将处理从文件加载的表达式与在键盘上键入的表达式相同。

虽然 Scheme 提供了各种输入和输出过程，但 REPL 负责读取表达式并打印其值。这使您可以专注于编写程序，而不必担心其结果将如何显示。

本章和本书其余部分中的示例遵循常规格式。首先给出您可能从键盘键入的表达式，可能跨越几行。表达式的

值在 之后给出，⇒读作“计算到”。⇒对于定义以及当表达式的值未指定时，将省略 。

示例程序的格式设置为“看起来不错”的样式，并传达了程序的结构。代码易于阅读，因为每个表达式与其子表达式之间的关系都清晰可见。但是，方案会忽略缩进和换行符，因此无需遵循特定样式。重要的是建立一种风格并坚持下去。Scheme 将每个程序视为在一行上，其子表达式按从左到右的顺序排列。

如果您有权访问交互式 Scheme 系统，最好立即启动它，并在阅读时键入示例。最简单的 Scheme 表达式之一是字符串常量。尝试键入“嗨，妈妈！”（包括双引号）以响应提示。系统应该用“嗨，妈妈！”来回应;任何常量的值都是常量本身。

“嗨，妈妈！”⇒ “嗨，妈妈！”

下面是一组表达式，每个表达式都有 Scheme 的响应。它们将在本章的后面几节中进行解释，但现在使用它们来练习与Scheme的交互。

```
“hello” ⇒ “hello”
42 ⇒ 42
22/7 ⇒ 22/7
3.141592653 ⇒ 3.141592653
+ ⇒ #<程序>
（+ 76 31） ⇒ 107
（* -12 10） ⇒ -120
’（a b c d） ⇒ （a b c d）
```

请注意不要错过任何单引号 （’）、双引号或括号。如果在上一个表达式中省略了单个引号，则可能会收到一

条消息，指示发生了异常。请重试。如果您省略了右括号或双引号，系统可能仍在等待它。

下面是一些可以尝试的表达式。您可以尝试自己弄清楚它们的含义，也可以等待本章后面的介绍。

```
(汽车'(a b c)) ⇒ a
(cdr '(a b c)) ⇒ (b c)
(cons 'a '(b c)) ⇒ (a b c)
(cons (car '(a b c))
(cdr '(d e f))) ⇒ (a e f)
```

如您所见，方案表达式可能跨越多行。Scheme 系统通过匹配双引号和括号来知道它何时具有整个表达式。

接下来，让我们尝试定义一个过程。

```
(定义平方
(lambda (n)
(* n n)))
```

过程平方计算任意数字 n 的平方 n^2。我们将在本章后面详细介绍构成此定义的表达式。现在，只需说定义建立变量绑定，lambda创建过程，*命名乘法过程就足够了。请注意这些表达式的形式。所有结构化形式都括在括号中，并用前缀表示法书写，即运算符在参数之前。如您所见，即使对于简单的算术运算（如 *）也是如此。

尝试使用正方形。

```
(正方形 5) ⇒ 25
(平方 -200) ⇒ 40000
(平方 0.5) ⇒ 0.25
(平方 -1/2) ⇒ 1/4
```

即使下一个定义很短，您也可以将其输入到文件中。假设您将该文件称为“reciprocal.ss”。

```
（定义倒数
（lambda （n）
（if （= n 0）
“oops！”
（/ 1 n））））
```

此过程是倒数的，计算任何数字 n ≠ 0 的数量 1/n。对于 n = 0，倒数返回字符串“oops！”。返回到“方案”，然后尝试使用过程加载加载文件。

```
（加载“reciprocal.ss”）
```

最后，尝试使用我们刚刚定义的过程。

```
（互惠 10）⇒ 1/10
（倒数 1/10） ⇒ 10
（倒数 0） ⇒ “哎呀！
（互惠（互惠1/10）） ⇒ 1/10
```

在下一节中，我们将更详细地讨论方案表达式。在本章中，请记住，您的 Scheme 系统是学习 Scheme 最有用的工具之一。每当您尝试文本中的某个示例时，请按照自己的示例进行跟进。在交互式 Scheme 系统中，尝试某些内容的成本相对较小---通常只是输入它的时间。

第 2.2 节.简单表达式

最简单的 Scheme 表达式是常量数据对象，如字符串、数字、符号和列表。Scheme 支持其他对象类型，但这四种对象类型对于许多程序来说已经足够了。我们在上一节中看到了一些字符串和数字的示例。

让我们更详细地讨论一下数字。数字是常量。如果您输入一个数字，Scheme 会将其回显给您。以下示例显示 Scheme 支持多种类型的数字。

```
12345678998765432 1 ⇒ 12345678998765432 1
3/4 ⇒ 3/4
2.718281828 ⇒ 2.718281828
2.2+1.1i ⇒ 2.2+1.1i
```

方案编号包括精确和不精确的整数、有理数、实数和复数。精确整数和有理数具有任意精度，即它们可以具有任意大小。不准确的数字通常在内部使用 IEEE 标准浮点表示形式。

方案为相应的算术过程提供了名称 +、-、* 和 /。每个过程接受两个数字参数。下面的表达式称为过程应用程序，因为它们指定将过程应用于一组参数。

```
(+ 1/2 1/2)⇒ 1
（- 1.5 1/2） ⇒ 1.0

（* 3 1/2） ⇒ 3/2
（/ 1.5 3/4） ⇒ 2.0
```

方案使用前缀表示法，即使对于常见的算术运算也是如此。任何过程应用程序，无论过程采用零个、一个、两个还是更多参数，都写为（过程参数...）。这种规律性简化了表达式的语法;无论操作如何，都使用一种表示法，并且没有关于运算符的优先级或关联性的复杂规则。

过程应用程序可以是嵌套的，在这种情况下，首先计算最里面的值。因此，我们可以嵌套上面给出的算术过程的应用程序，以计算更复杂的公式。

```
(+ (+ 2 2) (+ 2 2))⇒ 8
（- 2 （* 4 1/3）） ⇒ 2/3
（* 2 （* 2 （* 2 （* 2 2）））） ⇒ 32
（/ （* 6/7 7/2） （- 4.5 1.5）） ⇒ 1.0
```

这些示例演示了使用 Scheme 作为四功能桌面计算器所需的一切。虽然我们不会在本章中讨论它们，但Scheme支持许多其他算术过程。现在可能是转向第 6.4 节并尝试其中一些的好时机。

简单的数值对象足以满足许多任务的需求，但有时需要包含两个或多个值的聚合数据结构。在许多语言中，基本的聚合数据结构是数组。在方案中，它是列表。列表被编写为用括号括起来的对象序列。例如，（1 2 3 4 5）是一个数字列表，而（“this”is“”a“”list“）是一个字符串列表。列表不需要只包含一种类型的对象，因此（4.2“hi”）是包含数字和字符串的有效列表。列表可以是嵌套的（可能包含其他列表），因此 （（1 2） （3 4）） 是具有两个元素的有效列表，每个元素都是两个元素的列表。

您可能会注意到列表看起来就像过程应用程序一样，并想知道Scheme如何区分它们。也就是说，Scheme 如何区分对象列表 （obj1 obj2 ...） 和过程应用程序（procedure arg ...）？

在某些情况下，这种区别似乎是显而易见的。数字列表（1 2 3 4 5）很难与程序申请混淆，因为1是一个数字，而不是程序。因此，答案可能是Scheme查看列表或程序申请的第一个元素，并根据第一个元素是否是过程做出决定。这个答案还不够好，因为我们甚至可能希望将有效的过程应用程序（例如（+ 3 4））视为列表。答案是，

我们必须明确地告诉 Scheme 将列表视为数据，而不是过程应用程序。我们用报价来做到这一点。

```
（引用 （1 2 3 4 5））⇒ （1 2 3 4 5）
（引用 （“this” “is” “a” “list”））⇒ （“this” “is” “a” “list”）
（quote （+ 3 4））⇒ （+ 3 4）
```

引号强制将列表视为数据。尝试输入上面的表达式而不带引号;您可能会收到一条消息，指示前两个发生了异常，第三个发生了不正确的答案 （7）。

由于在 Scheme 代码中经常需要引号，因此 Scheme 将表达式前面的单引号 （ ’ ） 识别为引号的缩写。

```
’（1 2 3 4）⇒ （1 2 3 4）
’（1 2） （3 4））⇒ （（1 2） （3 4））
’（/ （* 2 -1） 3）⇒ （/ （* 2 -1） 3）
```

这两种形式都称为引号表达式。我们经常说，当一个对象被括在引号表达式中时，它被引用了。

引用表达式不是过程应用程序，因为它会抑制对其子表达式的计算。它是一种完全不同的句法形式。除了过程应用程序和引用表达式之外，Scheme还支持其他几种语法形式。每种句法形式的评估方式都不同。幸运的是，不同句法形式的数量很少。我们将在本章后面看到更多。

并非所有引号表达式都涉及列表。尝试使用以下表达式，使用和不带引号包装。

```
（引用你好）⇒ 你好
```

符号 hello 必须用引号括起来，以防止 Scheme 将 hello 视为变量。Scheme 中的符号和变量类似于数学表达式和方程式中的符号和变量。当我们计算数学表达式 1 - x 的某个 x 值时，我们将 x 视为一个变量。另一方面，当我们考虑代数方程$x^2 - 1 = (x - 1)(x + 1)$时，我们认为x是一个符号（实际上，我们象征性地想到整个方程）。正如引用列表告诉 Scheme 将括号形式视为列表而不是过程应用程序一样，引用标识符告诉 Scheme 将标识符视为符号而不是变量。虽然符号通常用于表示方程式或程序的符号表示中的变量，但符号也可以用作例如自然语言句子表示中的单词。

您可能想知道为什么应用程序和变量与列表和符号共享符号。共享表示法允许将 Scheme 程序表示为 Scheme 数据，从而简化了 Scheme 中解释器、编译器、编辑器和其他工具的编写。第12.7节中给出的方案解释器证明了这一点，该解释器本身写在方案中。许多人认为这是Scheme最重要的功能之一。

数字和字符串也可以用引号括起来。

```
'2 ⇒ 2
'2/3 ⇒ 2/3
（引用“嗨，妈妈！”）⇒ “嗨，妈妈！”
```

但是，在任何情况下，数字和字符串都被视为常量，因此没有必要引用它们。

现在，让我们讨论一些用于操作列表的方案过程。拆开列表有两个基本程序：car 和 cdr（发音为 could-er）。car 返回列表的第一个元素，cdr 返回列表的其余部分。（名称“car”和“cdr”派生自实现Lisp语言的

第一台计算机IBM 704所支持的操作。每个都需要一个非空列表作为其参数。

```
(汽车'(a b c)) ⇒ a
(cdr '(a b c)) ⇒ (b c)
(cdr '(a)) ⇒ ()

(car (cdr '(a b c))) ⇒ b
(cdr (cdr '(a b c))) ⇒ (c)

(car '(a b) (c d))) ⇒ (a b)
(cdr '(a b) (c d))) ⇒ ((c d))
```

列表的第一个元素通常称为列表的“汽车”，列表的其余部分通常称为列表的“cdr”。具有一个元素的列表的 cdr 是 ()，即空列表。

该过程缺点构造列表。这需要两个参数。第二个参数通常是列表，在这种情况下，cons 返回一个列表。

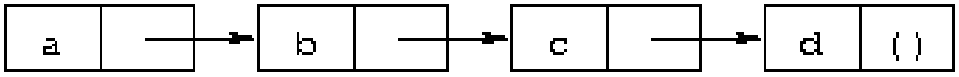
```
(缺点 'a '()) ⇒ (a)
(cons 'a '(b c)) ⇒ (a b c)
(cons 'a (cons 'b (cons 'c '()))) ⇒ (a b c)
(cons '(a b) '(c d)) ⇒ ((a b) c d)

(car (cons 'a '(b c))) ⇒ a
(cdr (cons 'a '(b c))) (b c)) ⇒ (b c)
(b c) (c) (crr '(a b c)))
⇒ (a b c)) ⇒ (a

b c)
```

正如“car”和“cdr”经常被用作名词一样，“cons”经常被用作动词。通过将元素添加到列表的开头来创建新列表称为将元素组合到列表中。

请注意cons的第二个参数描述中的“通常”一词。该过程实际上会生成对，并且没有理由认为一对的 cdr 必须是列表。列表是一系列成对;每对的 cdr 是序列中的下一对。



正确列表中最后一对的 cdr 是空列表。否则，对的序列将形成不正确的列表。更正式地说，空列表是一个正确的列表，而其 cdr 是正确列表的任何对都是一个正确的列表。

不正确的列表以点对表示法打印，在列表的最后一个元素前面有一个句点或点。

```
（缺点 ’a ’b）⇒ （a .b）
（cdr ’（a .b））⇒ b
（cons ’a ’（b .c））⇒ （a b .c）
```

由于其印刷符号，其cdr不是列表的一对通常被称为虚线对。但是，即使cdr是列表的对也可以用点对符号书写，尽管打印机总是选择编写没有点的正确列表。

```
’（a .（b .（c .（））））⇒ （a b c）
```

过程列表类似于 cons，不同之处在于它采用任意数量的参数并始终生成正确的列表。

```
（列出“a”b“c”⇒ （a b c）
（列表 ’a）⇒ （
列表）⇒ （）
```

第6.3节提供了有关列表的更多信息以及操作列表的方案程序。这可能是转向该部分并熟悉其中给出的其他过程

的好时机。

练习 2.2.1

将以下算术表达式转换为 Scheme 表达式并对其进行计算。

一. 1.2 × （2 - 1/3） + -8.7

二. （2/3 + 4/9） ÷ （5/11 - 4/3）

（二）1 + 1 ÷ （2 + 1 ÷ （1 + 1/2））

d. 1 × -2 × 3 × -4 × 5 × -6 × 7

练习 2.2.2

尝试使用过程 +、-、* 和 /，以确定在给定不同类型的数值参数时，每个过程返回的值类型的 Scheme 规则。

练习 2.2.3

确定以下表达式的值。使用您的方案系统来验证您的答案。

一. （缺点'汽车'cdr）

二. （列出“这个”（很傻））

（二）（缺点是“（这么傻？）”

d. （报价 （+ 2 3））

e. （缺点 '+ '（2 3））

六. （汽车 '（+ 2 3））

克。 （cdr '（+ 2 3））

h. 缺点

一. （引用缺点）

j. （引用（引用缺点））

k. （汽车（报价（报价缺点））

l. (+ 2 3)

嗯。 (+ ’2 ’3)

n. （+ （汽车 ’（2 3）） （汽车 （cdr ’（2 3））））

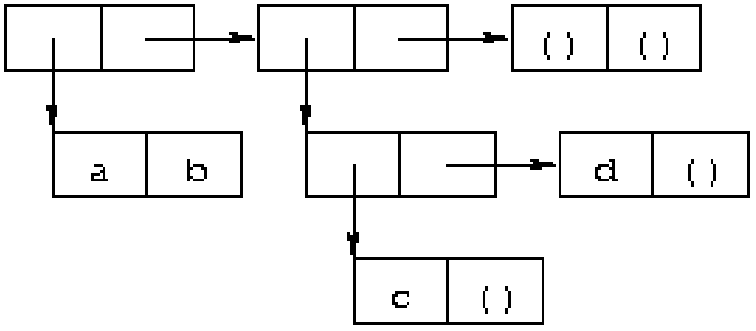
o. （（汽车 （列表 + - * /）） 2 3）

练习 2.2.4

（汽车（汽车’（（a b）（c d））））产生a。确定应用于（（a b）（c d））的汽车和cdr的哪些成分产生b，c和d。

练习 2.2.5

编写一个方案表达式，其计算结果为以下内部列表结构。



练习 2.2.6

绘制由以下表达式生成的内部列表结构。

（缺点 1 （缺点 '（2 .((3) . （） ））） （缺点 '（ （） ） （缺点 4 5） ） ））

练习 2.2.7

（汽车（汽车（汽车'（（a b） （c d） ） ） ） ）的行为未定义，因为（汽车'（（a b） （c d） ） ） ）是（a b），（汽车'（a b） ）是a，（汽车'a）是未定义的。确定适用于（（a b） （c d） ）的汽车和cdr的所有合法成分。

练习 2.2.8

尝试解释如何计算方案表达式。您的解释是否涵盖了练习 2.2.3 中的最后一个示例？

第 2.3 节. 计算方案表达式

让我们来讨论一下 Scheme 如何计算您键入的表达式。我们已经为字符串和数字等常量对象建立了规则：对象本身就是值。您可能还在脑海中制定了评估表单的程序应用程序的规则（程序$_{arg1}$ ...$_{argn}$）.在这里，过程是表示 Scheme 过程的表达式，而 $_{arg1}$ ...阿$_{根}$是表示其参数的表达式。一种可能性如下。

- 找到过程的价值。
- 查找 $_{arg1}$ 的值。

⋮

- 找到 $_{argn}$ 的值。

- 将过程值应用于 arg1 的值 ...唉。

例如，考虑简单的过程应用程序（+ 3 4） 。+ 的值是加法过程，3 的值是数字 3，4 的值是数字 4。将加法过程应用于 3 和 4 将得到 7，因此我们的值是对象 7。

通过在每个级别应用此过程，我们可以找到嵌套表达式的值 （* （+ 3 4） 2）。*的值是乘法过程，（+ 3 4）的值我们可以确定为数字7，而2的值是数字2。将7乘以2，我们得到14，所以我们的答案是14。

此规则适用于过程应用程序，但不适用于引号表达式，因为过程应用程序的子表达式是计算的，而引号表达式的子表达式不是。引号表达式的计算更类似于常量对象的计算。表单（引用对象）的引号表达式的值只是对象。

常量对象、过程应用程序和引用表达式只是 Scheme 提供的众多句法形式中的三个。幸运的是，只有少数其他语法形式需要由Scheme程序员直接理解;这些被称为核心句法形式。其余的句法形式是句法扩展，最终根据核心句法形式定义。我们将在本章的其余部分中讨论其余的核心句法形式和一些句法扩展。第3.1节总结了核心句法形式，并介绍了句法扩展机制。

在我们继续更多的句法形式和程序之前，与程序申请评估相关的两点值得注意。首先，上面给出的过程被过度指定，因为它要求从左到右评估子表达式。也就是说，在 arg1 之前计算过程，在 arg2 之前计算 arg1，依此类推。事实并非如此。Scheme 计算器可以自由地按任何顺序计算表达式---左到右、从右到左或任何其他顺序。实际上，即使在相同的实现中，也可以针对不同的应用程序以不同的顺序评估子表达式。

第二点是该过程的评估方式与arg1相同...唉。虽然过程通常是命名特定过程的变量，但情况并非如此。练习 2.2.3 让您确定表达式的值（（car （list + - * /）） 2 3））。这里，程序是（汽车（列表+ - * /））。（car （list + - * /）） 的值是加法过程，就像过程只是变量+一样。

练习 2.3.1

在下面记下计算表达式所需的步骤。

（（汽车 （cdr （列表 + - * /）） 17 5）

第 2.4 节. 变量和让表达式

假设 expr 是一个包含变量 var 的 Scheme 表达式。此外，假设我们希望 var 在评估 expr 时具有值 val。例如，我们可能希望 x 在计算 （+ x 3） 时具有值 2。或者，我们可能希望 y 在评估 （+ 2 y） 时具有值 3。以下示例演示如何使用 Scheme 的 let 语法形式执行此操作。

```
（让 （（x 2））
（+ x 3））⇒ 5

（让 （（y 3））
（+ 2 y）） ⇒ 5

（let （（x 2） （y 3））
（+ x y）） ⇒ 5
```

let 语法形式包括变量-表达式对的列表，以及一系列称为 let 主体的表达式。let 表达式的一般形式是

（让 （（var expr） ...）身体$_1$ 身体$_2$ ...）

我们说变量通过 let 绑定到值。我们将 let 绑定的变量称为 let 绑定变量。

let 表达式通常用于简化包含两个相同子表达式的表达式。这样做还可以确保只计算一次公共子表达式的值。

(+ (* 4 4) (* 4 4))⇒ 32

（让 （（a （* 4 4））） （+ a）） ⇒ 32

通常使用方括号代替括号来分隔 let 表达式的绑定。

```
(let ([list1 '(a b c)] [list2 '(d e f)])
(cons (cons (car list1)
(car list2))
(cons (car (cdr list1))
(car (cdr list2)))))⇒ ((a . d) b . e)
```

方案处理括在括号中的窗体就像处理括在括号中的窗体一样。左括号必须与右括号匹配，左括号必须与右括号匹配。我们使用括号来表示 let（以及，正如我们将要看到的，其他几种标准语法形式）以提高可读性，特别是当我们可能有两个或更多个连续的左括号时。

由于过程应用程序第一个位置的表达式的计算方式与其他表达式没有区别，因此也可以使用 let-bound 变量。

```
(让 ([f +])
(f 2 3))⇒ 5

(让 ([f +] [x 2])
(f x 3)) ⇒ 5
```

```
（让 （[f +] [x 2] [y 3]）
（f x y））⇒ 5
```

由 let 绑定的变量仅在 let 的主体内可见。

```
（让我们 （[+ *]）
（+ 2 3））⇒ 6

（+ 2 3）⇒ 5
```

这是幸运的，因为我们不希望 +的值成为任何地方的乘法过程。

可以嵌套 let 表达式。

```
（让 （[a 4] [b -3]）
（让 （[a-平方 （* a）]
[b-平方 （* b b）]）
（+ a 平方 b 平方）））⇒ 24
```

当嵌套 let 表达式绑定同一变量时，只有内部 let 创建的绑定在其主体中可见。

```
（let （[x 1]）
（let （[x （+ x 1）]）
（+ x x）））⇒ 4 个
```

外部 let 表达式在其主体内将 x 绑定到 1，这是第二个 let 表达式。内部 let 表达式将 x 绑定到其主体内的（+ x 1），即表达式 （+ x x）。（+ x 1） 的值是多少？由于 （+ x 1） 出现在外 let 的主体内，但不在内设的主体内，因此 x 的值必须为 1，因此 （+ x 1） 的值为 2。（+ x x）呢？它出现在两个 let 表达式的正文中。只有 x 的内部绑定是可见的，因此 x 是 2，（+ x x） 是 4。

x 的内部绑定称为阴影外部绑定。let 绑定变量在其 let 表达式主体内的任何位置都可见，除非它被阴影覆盖。变量绑定可见的区域称为其作用域。在上面的示例中，第一个 x 的作用域是外部 let 表达式的主体减去内部 let 表达式的主体，其中它被第二个 x 阴影。这种形式的范围界定称为词法范围界定，因为每个绑定的范围可以通过对程序的直接文本分析来确定。

通过为变量选择不同的名称可以避免重影。上面的表达式可以重写，以便由内部 let 绑定的变量是 new-x。

```
(let ([x 1])
(let ([new-x (+ x 1)])
(+ new-x new-x)))⇒ 4 个
```

虽然选择不同的名称有时可以防止混淆，但阴影可以帮助防止意外使用“旧”值。例如，对于前面示例的原始版本，我们不可能错误地引用内部 let 主体内的外部 x。

练习 2.4.1

重写以下表达式，使用 let 删除常见的子表达式并改进代码的结构。不要执行任何代数简化。

一. (+ (- (* 3 a) b) (+ (* 3 a) b))

二. (缺点 (汽车 (列表 a b c)) (cdr (列表 a b c))))

练习 2.4.2

确定以下表达式的值。说明如何派生此值。

```
(let ([x 9])
(* x
(let ([x (/ x 3)])
(+ x x)))
```

练习 2.4.3

重写以下表达式，为每个不同的 let 绑定变量指定唯一的名称，以便不重影任何变量。验证表达式的值是否与原始表达式的值相同。

一.

```
(let ([x 'a] [y 'b])
(list (let ([x 'c]) (cons x y))
(let ([y 'd]) (cons x y)))))
```

二.

```
(let ([x '((a b) c)])
(cons (let ([x (cdr x)])
(car x))
(let ([x (car x)])
(cons (let ([x (cdr x)])
(car x))
(cons (let ([x (car x)])
x)
(cdr x)))))))
```

第 2.5 节.Lambda 表达式

在表达式 (let ([x (* 3 4)]) (+ x x)) 中，变量 x 绑定到 (* 3 4) 的值。如果我们想要 (+ x x) 的值，其中 x 绑定到 (/ 99 11) 的值，该怎么办？其中 x 绑定到 (- 2 7) 的值？在每种情况下，我们都需要

不同的 let 表达式。然而，当让体很复杂时，不得不重复它可能是不方便的。

相反，我们可以使用语法形式 lambda 来创建一个新过程，该过程将 x 作为参数，并且与 let 表达式具有相同的主体。

```
（lambda （x） （+ x x））⇒ #<程序>
```

lambda 表达式的一般形式是

```
（lambda （var ...）身体1 身体2 ...）
```

变量 var ... 是过程的形式参数，表达式的序列 body1 body2 ... 是它的主体。（实际上，真正的一般形式比这更通用，稍后您将看到。

过程与数字、字符串、符号或对一样是一个对象。然而，就Scheme而言，它没有任何有意义的印刷表示，因此本书使用符号#<procedure>来证明表达式的值是一个过程。

对过程执行的最常见操作是将其应用于一个或多个值。

```
（（lambda （x） （+ x x）） （* 3 4））⇒ 23
```

这与任何其他程序应用程序没有什么不同。该过程的值是 （lambda （x） （+ x x）），唯一的参数是 （* 3 4）或 12 的值。参数值或实际参数绑定到 lambda 表达式主体内的形式参数，其方式与 let 绑定变量绑定到其值的方式相同。在本例中，x 绑定到 12，（+ x x） 的值为 24。因此，将过程应用于值 12 的结果是 24。

因为过程是对象，所以我们可以将过程建立为变量的值，并多次使用该过程。

```
(let ([double (lambda (x) (+ x x))])
(list (double (* 3 4))
(double (/ 99 11))
(double (- 2 7))))⇒ (24 18 -10)
```

在这里，我们为双精度值建立一个绑定到一个过程，然后使用此过程将三个不同的值加倍。

该过程期望其实际参数是一个数字，因为它将实际参数传递给 +。通常，实际参数可以是任何类型的对象。例如，考虑一个使用 cons 而不是 + 的类似过程。

```
(let ([double-cons (lambda (x) (cons x x))])
(double-cons 'a))⇒ (a .
```

注意到双重和双重缺点之间的相似性，您不应该惊讶地发现，通过添加额外的参数，它们可能会折叠成单个过程。

```
(let ([double-any (lambda (f x) (f x x))])
(list (double-any + 13)
(double-any cons 'a)))⇒ (26 ( a .
```

这表明过程可以接受多个参数，并且传递给过程的参数本身可能是过程。

与 let 表达式一样，当 lambda 表达式嵌套在其他 lambda 或 let 表达式中时，它们会变得更加有趣。

```
(let ([x 'a])
(let ([f (lambda (y) (list x y))])
(f 'b)))⇒ (一 b)
```

在 lambda 表达式中出现的 x 是指由外部 let 表达式绑定的 lambda 外部的 x。变量 x 在 lambda 表达式中自由出现，或者说是 lambda 表达式的自由变量。变量 y 在 lambda 表达式中不会自由出现，因为它受 lambda 表达式的约束。在 lambda 表达式中自由出现的变量应该被绑定，例如，通过封闭的 lambda 或 let 表达式，除非变量（如原始过程的名称）绑定在表达式之外，正如我们在下一节中讨论的那样。

当过程应用于过程内自由发生的变量的绑定范围之外的某个位置时，会发生什么情况，如以下表达式所示？

```
(let ([f (let ([x 'sam])
(lambda (y z) (list x y z)))])
(f 'i 'am)) ⇒ (山姆我是)
```

答案是，创建过程时生效的相同绑定在应用该过程时再次生效。即使 x 的另一个绑定在应用过程的位置可见，也是如此。

```
(let ([f (let ([x 'sam])
(lambda (y z) (list x y z)))])
(let ([x 'not-sam])
(f 'i 'am))) ⇒ (山姆我是)
```

在这两种情况下，名为 f 的过程中的 x 的值都是 sam。

顺便说一句，let 表达式只不过是将 lambda 表达式直接应用于一组参数表达式。例如，下面的两个表达式是等效的。

```
(let ([x 'a]) (cons x x)) ≡ ((lambda (x)
(cons x x)) 'a)
```

实际上，let 表达式是根据 lambda 和过程应用程序定义的句法扩展，它们都是核心句法形式。一般而言，任何表示形式

（让 （（var expr） ...）身体$_1$ 身体$_2$ ...）

等效于以下内容。

（（lambda （var ...）身体$_1$ 身体$_2$ ...）
expr ...）

有关核心形式和语法扩展的更多信息，请参见第 3.1 节。

如上所述，lambda的一般形式比我们之前看到的形式要复杂一些，因为形式参数规范（var ...）不需要是一个正确的列表，甚至根本不需要一个列表。正式参数规范可以是以下三种形式之一：

- 变量的正确列表，（var1 ... varn），就像我们已经看到的那样，
- 单个变量、varr 或
- 不正确的变量列表，（var1 ... 瓦尔恩 . varr）.

在第一种情况下，必须提供确切的 n 个实际参数，并且每个变量都绑定到相应的实际参数。在第二种情况下，任意数量的实际参数都是有效的;所有实际参数都放入单个列表中，单个变量绑定到此列表。第三种情况是前两种情况的混合。必须至少提供 n 个实际参数。变量 var1 ... 瓦恩绑定到相应的实际参数，变量 varr 绑定到包含其余实际参数的列表。在第二种和第三种情况下，varr 有

时被称为“rest”参数，因为它保存了除单独命名的参数之外的其余实际参数。

让我们考虑几个示例，以帮助阐明 lambda 表达式的更一般语法。

```
(let ([f (lambda x x)])
(f 1 2 3 4)) ⇒ (1 2 3 4)

(let ([f (lambda x x)])
(f)) ⇒ ()

(let (let ([g (lambda (x . y) (list x y))])
(g 1 2 3 4)
)
⇒ (let ([h (lambda (x y . z) (list x y z))])
(h 'a 'b 'c 'd)) ⇒ (a b (c d))
```

在前两个示例中，名为 f 的过程接受任意数量的参数。这些参数自动形成变量 x 绑定到的列表;f 的值是此列表。在第一个示例中，参数为 1、2、3 和 4，因此答案是 (1 2 3 4)。在第二个中，没有参数，因此答案是空列表 ()。在第三个示例中名为 g 的过程的值是一个列表，其第一个元素是第一个参数，其第二个元素是包含其余参数的列表。名为 h 的过程类似，但将第二个参数分开。虽然 f 接受任意数量的参数，但 g 必须至少接收一个参数，h 必须至少接收两个参数。

练习 2.5.1

确定以下表达式的值。

一.

```
(let ([f (lambda (x) x)])
```

```
(f 'a) )
```

二.

```
(let ([f (lambda x x) ])
(f 'a) )
```

(二)

```
(let ([f (lambda (x . y) x) ])
(f 'a) )
```

d.

```
(let ([f (lambda (x . y) y) ])
(f 'a) )
```

练习 2.5.2

如何定义基元过程列表?

练习 2.5.3

在下面列出每个 lambda 表达式中自由出现的变量。不要省略命名原始过程（如 + 或 cons）的变量。

一. `(lambda (f x) (f x) )`

二. `(lambda (x) (+ x x) )`

(二) `(lambda (x y) (f x y) )`

d.

```
(lambda (x)
(cons x (f x y) ) )
```

e.

```
(lambda (x)
```

```
(let ([z (cons x y)])
(x y z)))
```

六.

```
(lambda (x)
(let ([y (cons x y)])
(x y z)))
```

第 2.6 节. 顶级定义

由 let 和 lambda 表达式绑定的变量在这些表达式的主体外部不可见。假设您创建了一个对象，也许是一个过程，它必须可以在任何地方访问，例如 + 或 cons。您需要的是一个顶级定义，该定义可以使用定义建立。大多数交互式 Scheme 系统都支持的顶级定义在您输入的每个表达式中都可见，除非被另一个绑定遮蔽。

让我们建立上一节的双任意过程的顶级定义。

```
(定义双任意
(lambda (f x)
(f x x)))
```

变量 double-any 现在与 cons 或任何其他基元过程的名称具有相同的状态。我们可以使用双任意，就好像它是一个原始过程一样。

```
(双任意 + 10) ⇒ 20
(双倍任何缺点 'a) ⇒ (a . a)
```

可以为任何对象建立顶级定义，而不仅仅是过程。

（定义三明治“花生酱和果冻”）

三明治⇒“花生酱和果冻”

但是，大多数情况下，顶级定义用于过程。

如上所述，顶级定义可能被 let 或 lambda 绑定遮蔽。

```
（定义 xyz ’（x y z））
（let （[xyz ’（z y x）]）
xyz） ⇒ （z y x）
```

具有顶级定义的变量几乎就像它们受包含您键入的所有表达式的 let 表达式的约束一样。

鉴于到目前为止，您已经阅读过的简单工具，已经可以定义Scheme提供的一些原始过程，并在本书后面进行介绍。如果您完成了上一部分的练习，您应该已经知道如何定义列表。

```
（定义列表 （lambda x x））
```

此外，Scheme还为带有cdr的汽车和带有cdr的汽车的组合提供了缩写cadr和cddr。也就是说，（cadr列表）等效于（car（cdr list））， 同样，（cddr list）等价于（cdr（cdr list））。它们很容易定义如下。

```
（定义 cadr
（lambda （x）
（car （cdr x））））

（定义 cddr
（lambda （x）
（cdr （cdr x））））
```

```
(cadr '(a b c)) ⇒ b
(cddr '(a b c)) ⇒ (c)
```

任何定义(定义 var expr)其中 expr 是 lambda 表达式，都可以用更短的形式编写，以抑制 lambda。确切的语法取决于 lambda 表达式的形式参数说明符的格式，即它是正确的变量列表、单个变量还是不正确的变量列表。表单的定义

```
(定义 var0
(lambda (var1 ... varn)
e1 e2 ...))
```

可缩写

```
(定义 (var0 var1 ... varn)
e1 e2 ...)
```

而

```
(define var0
(lambda varr
e1 e2 ...))
```

可缩写

```
(定义 (var0 . varr)
e1 e2 ...)
```

和

```
(定义 var0
(lambda (var1 ... 瓦尔恩 . varr)
e1 e2 ...))
```

可缩写

```
（定义 （var0 var1 ... 瓦尔恩 . varr）
e1 e2 ...）
```

例如，cadr 和 list 的定义可以写如下。

```
（定义 （cadr x）
（car （cdr x））））

（定义 （列表 x） x）
```

本书不经常使用这种替代语法。虽然它更短，但它往往掩盖了这样一个现实，即过程不像许多其他语言那样与变量或名称紧密相关。这种语法通常被贬义地称为定义的“defun”语法，在Lisp语言提供的defun形式之后，其中过程与其名称更紧密地联系在一起。

顶级定义使我们更容易以交互方式试验过程，因为我们不需要在每次使用过程时都重新键入该过程。让我们尝试定义一个更复杂的双任意变体，它将“普通”的双参数过程转换为“加倍”的单参数过程。

```
（定义倍增器
（lambda （f）
（lambda （x） （f x x））））
```

doubler 接受一个参数 f，它必须是接受两个参数的过程。倍增器返回的过程接受一个参数，它将该参数用于 f 应用中的两个参数。我们可以用倍增器定义上一节的简单双精度和双缺点过程。

```
（定义双精度（倍频器 +））
（双 13/2） ⇒ 13
```

```
（定义双缺点 （倍增缺点））
（双缺点 'a） ⇒ （a . a）
```

我们还可以用倍增器定义双任意。

```
（定义双任意
（lambda （f x）
（（doubler f） x））））
```

在双精度和双精度值中，f 具有适当的值，即 + 或缺点，即使这些过程明显应用在 f 的范围之外。

如果您尝试使用不受 let 或 lambda 表达式约束且没有顶级定义的变量，会发生什么情况？尝试使用变量 i-am-not-defined 来查看会发生什么情况。

```
（i-am-un-defined 3）
```

大多数 Scheme 系统都会打印一条消息，指示发生了未绑定或未定义的变量异常。

但是，除非应用结果过程，否则系统不应抱怨 lambda 表达式中出现未定义的变量。以下内容不应导致异常，即使我们尚未建立 proc2 的顶级定义。

```
（定义 proc1
（lambda （x y）
（proc2 y x））））
```

如果在定义 proc2 之前尝试应用 proc1，则应收到未定义的异常消息。让我们给 proc2 一个顶级定义，并尝试 proc1。

```
（定义 proc2 缺点）
（proc1 'a 'b） ⇒ （b . a）
```

定义 proc1 时，系统将接受您定义 proc2 的承诺，并且不会抱怨，除非您在定义 proc2 之前使用 proc1。这允许您按任何顺序定义过程。当您尝试以一种使程序更具可读性的方式组织充满过程定义的文件时，这尤其有用。当在顶层定义的两个程序相互依赖时，这是必要的；我们稍后会看到一些这样的例子。

练习 2.6.1

如果您要键入，会发生什么情况

（双倍任意双任意双任意）

给定本节开头的双任意的定义？

练习 2.6.2

与本节中给出的相比，定义 cadr 和 cddr 的一种更优雅（尽管可能效率较低）的方法是定义一个过程，该过程由两个过程组成以创建第三个过程。编写过程 compose，使得（撰写 p1 p2）是 p1 和 p2 的组合（假设两者都采用一个参数）。也就是说，（编写 p1 p2）应返回一个参数的新过程，该过程将 p1 应用于将 p2 应用于参数的结果。使用 compose 定义 cadr 和 cddr。

练习 2.6.3

Scheme 还提供了 caar、cdar、caaar、caadr 等，在 c 和 r 之间最多有四个 a（表示汽车）和 d（代表 cdr）的任意组合（参见第 6.3 节）。使用前面练习的撰写过程定义其中的每一个。

第 2.7 节. 条件表达式

到目前为止，我们已经考虑了无条件执行给定任务的表达式。假设我们希望编写过程 abs。如果它的参数 x 为负，则 abs 返回 -x;否则，它将返回 x。写 abs 的最直接方法是确定参数是否为负数，如果是则否定它，则使用 if 句法形式。

```
（定义 abs
（lambda （n）
（if （< n 0）
（- 0 n）
n））））
```

```
（绝对 77） ⇒ 77
（绝对 -77） ⇒ 77
```

if 表达式具有以下形式（如果检验结果替代），其中 result 是用于计算检验是否为真的表达式，而备选表达式是用于计算检验是否为 false 的表达式。在上面的表达式中，检验是 （< n 0），结果是 （- 0 n），备选是 n。

程序 abs 可以用各种其他方式编写。以下任何一项都是abs的有效定义。

```
（定义 abs
（lambda （n）
（if （>= n 0）
n
（- 0 n））））
```

```
（定义 abs
（lambda （n）
（if （not （< n 0））
n
（- 0 n））））
```

```
(定义 abs
(lambda (n)
(if (> n 0) (= n 0))
n
(- 0 n)))))

(定义 abs
(lambda (n)
(if (= n 0)
0
(if (< n 0)
(- 0 n)
n)))))

(定义 abs
(lambda (n)
((if (>= n 0) + -)
0
n)))
```

这些定义中的第一个询问 n 是否大于或等于零，从而反转测试。第二个询问 n 是否不小于零，使用不与 < 的过程。第三个询问 n 是否大于零或 n 是否等于零，使用句法形式 or。第四种分别处理零，尽管这样做没有任何好处。第五个有点棘手;n 要么从零中相加，要么从零中减去，具体取决于 n 是大于还是等于零。

为什么是句法形式而不是过程？为了回答这个问题，让我们从本章的第一部分重新审视互惠的定义。

```
(定义倒数
(lambda (n)
(if (= n 0)
“oops!”
(/ 1 n)))))
```

除法过程的第二个参数不应为零，因为结果在数学上是未定义的。我们对倒易的定义通过在除法前测试零来避免这个问题。如果一个程序，其论点（包括（/ 1 n））将在它有机会在结果和备选方案之间做出选择之前进行评估。就像引用，它不评估其唯一的子表达式，如果不计算其所有子表达式，因此不能成为一个过程。

句法形式或操作方式类似于 if。或表达式的一般形式是（或 expr ...）。如果没有子表达式，即表达式是简单的（或），则值为false。否则，将依次计算每个 expr，直到 （a） 其中一个表达式的计算结果为 true 或 （b） 不再有表达式。在情况（a）中，该值为真;在情况（b） 中，该值为假。

更准确地说，在情况 （a） 中，or 表达式的值是计算的最后一个子表达式的值。这种澄清是必要的，因为有许多可能的真实值。通常，测试表达式的值是#t（对于 true）或#f（对于 false）的两个对象之一。

```
(< -1 0) ⇒ #t
(> -1 0) ⇒ #f
```

但是，每个 Scheme 对象都通过条件表达式和过程 not 被视为 true 或 false。只有#f才被认为是假的;所有其他对象都被视为真。

```
(如果#t“真”假) ⇒ true
(if #f 'true 'false) ⇒ false
(if '() 'true 'false) ⇒ true
(if 1 'true 'false) ⇒ true
(if '(a b c) 'true 'false) ⇒ true

(not #t) ⇒ #f
(not 'false“) ⇒ #f
```

```
（not #f） ⇒ #t

（或） ⇒ #f
（或 #f） ⇒ #f
（或 #f #t） ⇒ #t
（或#f “#f”⇒ 一个
```

和 句法形式在形式上与 or 相似，但如果 and 表达式的所有子表达式都是真，则为真，否则为假。在没有子表达式的情况下，即表达式是简单的（和），值为真。否则，将依次计算子表达式，直到不再留下子表达式或子表达式的值为假。和 表达式的值是计算的最后一个子表达式的值。

使用 and，我们可以定义一个稍微不同的倒数版本。

```
（定义倒数
（lambda （n）
（and （not （= n 0））
（/ 1 n））））

（倒数 3） ⇒ 1/3
（倒数 0.5） ⇒ 2.0
（倒数 0） ⇒ #f
```

在此版本中，如果 n 为零，则值为 #f，否则为 1/n。

过程 =、<、>、<= 和 >= 称为谓词。谓词是回答有关其参数的特定问题并返回#t或#f两个值之一的过程。大多数谓词的名称以问号 （ ？ ） 结尾;上面列出的常见数字过程是此规则的例外。当然，并非所有谓词都需要数字参数。谓词 null？ 如果其参数是空列表 （），则返回 true，否则返回 false。

```
(空? '()) ⇒ #t
(空? 'abc) ⇒ #f
(空? '(x y z)) ⇒ #f
(null? (cdddr '(x y z))) ⇒ #t
```

不得传递过程 cdr 对以外的任何内容，并且发生这种情况时会引发异常。然而，Common Lisp 将 (cdr '()) 定义为 ()。下面的过程 lisp-cdr 是使用 null? 定义的，如果它的参数是 () 则返回 ()。

```
(define lisp-cdr
(lambda (x)
(if (null? x)
'()
(cdr x)))))

(lisp-cdr '(a b c)) ⇒ (b c)
(lisp-cdr '(c)) ⇒ ()
(lisp-cdr '()) ⇒ ()
```

另一个有用的谓词是eqv?，它需要两个参数。如果这两个参数等效，则 eqv? 返回 true。否则，eqv? 返回 false。

```
(eqv? 'a 'a) ⇒ #t
(eqv? 'a 'b) ⇒ #f
(eqv? #f #f) ⇒ #t
(eqv? #t #t) ⇒ #t
(eqv? #f #t) ⇒ #f
(eqv? 3 3) ⇒ #t
(eqv? 3 2) ⇒ #f
(let ([x "Hi Mom!"])
(eqv? x x)) ⇒ #t
(let ([x (cons 'a 'b)])
(eqv? x x)) ⇒ #t
(eqv? (缺点 'a 'b) (缺点 'a 'b)) ⇒ #f
```

如您所见，如果参数是相同的符号、布尔值、数字、对或字符串，则 eqv? 返回 true。两对 eqv 是不一样的？如果它们是由不同的 cons 调用创建的，即使它们具有相同的内容。eqv? 的详细等效规则在第 6.2 节中给出。

Scheme 还提供了一组类型谓词，这些谓词根据对象的类型返回 true 或 false，例如，对、符号、数字和字符串？。例如，谓词对？仅当其参数是一对时才返回 true。

```
（配对？’（一.c））  ⇒ #t
（对？’（a b c））  ⇒ #t
（对？’（））  ⇒ #f
（对？’abc）  ⇒ #f
（对？“嗨，妈妈！”⇒ #f
（对？1234567890）  ⇒ #f
```

类型谓词对于确定传递给过程的参数是否属于适当的类型非常有用。例如，以下版本的倒数在针对零进行测试或执行除法之前，首先检查其参数是否为数字。

```
（定义倒数
（lambda （n）
（if （and （number？ n） （not （= n 0））））
（/ 1 n）
“oops！”））

（倒数 2/3）  ⇒ 3/2
（倒数 ’a）  ⇒ “哎呀！
```

顺便说一句，使用倒数的代码必须检查返回的值是否是数字而不是字符串。为了免除调用方的此义务，通常最好使用断言冲突来报告错误，如下所示。

```
（定义倒数
（lambda （n）
（if （and （number? n） （not （= n 0））））
（/ 1 n）
（断言违反 'reciprocal
“improper argument”
n）））））

（倒数 .25） ⇒ 4.0
（倒数 0） ⇒ 倒数中的异常：不适当的参数 0
（倒数 'a） ⇒ 倒数中的异常：不适当的参数 a
```

断言违规的第一个参数是标识消息来源的符号，第二个是描述错误的字符串，第三个和后续参数是要包含在错误消息中的“刺激物”。

让我们再看一个条件表达式 cond，它通常可以用来代替 if。cond 类似于 if，只是它允许多个检验和替代表达式。考虑以下符号定义，对于负输入返回 -1，对于正输入返回 +1，对于零返回 0。

```
（定义符号
（lambda （n）
（if （< n 0）
-1
（if （> n 0）
+1
0））））

（符号 -88.3）⇒ -1
（标志 0） ⇒ 0
（标志 333333333333） ⇒ 1
（* （标志 -88.3） （腹部 -88.3）） ⇒ -88.3
```

这两个 if 表达式可以替换为单个 cond 表达式，如下所示。

```
（定义符号
（lambda （n）
（cond
[（< n 0） -1]
[（> n 0） +1]
[else 0]））））
```

cond 表达式通常采用以下形式

```
（电导（测试例程） ...（否则除外））
```

尽管 else 子句可以省略。仅当所有测试都不可能失败时，才应执行此操作，如下面的新版本 sign 所示。

```
（定义符号
（lambda （n）
（cond
[（< n 0） -1]
[（> n 0） +1]
[（= n 0） 0]））
```

符号的这些定义不依赖于执行测试的顺序，因为对于任何 n 值，只有一个测试可以为真。以下过程计算累进税制中给定收入金额的税款，断点分别为 10，000、20，000 和 30，000 美元。

```
（定义所得税
（lambda（收入）
（cond
[（<=收入10000）（*收入.05）]
[（<=收入20000）（+（*（-收入10000）.08）500.00）]
[（<=收入30000）（+（*（-收入20000）.13）1300.00）]
][否则（+（*（-收入30000）.21）2600.00）]））））

（所得税5000）⇒ 250.0
（所得税 15000） ⇒ 900.0
```

```
（所得税 25000） ⇒ 1950.0
（所得税 50000） ⇒ 6800.0
```

在此示例中，执行测试的顺序（从左到右（从上到下）非常重要。

练习 2.7.1

定义谓词原子?，如果它的参数不是一对，则返回 true，如果它是则返回 false。

练习 2.7.2

过程长度返回其参数的长度，该参数必须是列表。例如，（长度 '（a b c）） 为 3。使用 length 定义较短的过程，该过程返回两个列表参数中较短的一个。如果它们具有相同的长度，请让它返回第一个列表。

```
（较短的'（a b）'（c d e））⇒ （a b）
（较短的'（a b） '（c d）） ⇒ （a b）
（较短的'（a b） '（c）） ⇒ （c）
```

第 2.8 节. 简单递归

我们已经看到了如何控制表达式是否使用 if、and、or 和 cond 进行计算。我们还可以通过创建包含表达式的过程并多次调用该过程来多次执行表达式。如果我们需要重复执行一些表达式，例如对于列表中的所有元素或从1到10的所有数字，该怎么办？我们可以通过递归来做到这一点。递归是一个简单的概念：从该过程中应用一个过程。一开始掌握递归可能很棘手，但一旦掌握，它提供的表达能力远远超出了普通的循环结构。

递归过程是应用自身的过程。也许最简单的递归过程是以下，我们将称之为再见。

```
（定义再见
（lambda （）
（再见））））

（再见） ⇒
```

此过程不带任何参数，只需立即应用即可。没有价值之后，⇒因为再见永远不会回来。

显然，为了实际使用递归过程，我们必须有某种方法来终止递归。大多数递归过程应至少具有两个基本元素：一个基本情况和一个递归步骤。基本情况终止递归，为某些基本参数提供过程的值。递归步骤根据应用于不同参数的过程的值给出值。为了使递归终止，不同的参数必须以某种方式更接近基参数。

让我们考虑以递归方式查找正确列表的长度的问题。我们需要一个基本情况和一个递归步骤。列表递归的逻辑基参数几乎总是空列表。空列表的长度为零，因此基本情况应为空列表提供值为零。为了更接近空列表，自然递归步骤涉及参数的 cdr。非空列表比其 cdr 长一个元素，因此递归步骤将值设置为比列表的 cdr 长度多一个值。

```
（定义长度
（lambda （ls）
（if （null？ ls）
0
（+ （length （cdr ls）） 1））））

（长度 ’（））⇒ 0
（长度 ’（a）） ⇒ 1
```

（长度 '（a b）） ⇒ 2

if 表达式询问列表是否为空。如果是这样，则该值为零。这是基本情况。如果不是，则该值比列表的 cdr 长度多一个。这是递归步骤。

许多 Scheme 实现允许您跟踪过程的执行，以查看其运行方式。例如，在 Chez Scheme 中，跟踪过程的一种方法是键入（跟踪名称），其中 name 是您在顶层定义的过程的名称。如果按照上面定义跟踪长度并将其传递给参数 '（a b c d），则应看到如下内容：

```
|（长度 （a b c d））
|（长度（b c d））
| |（长度（c d））
| |（长度（d））
| | |（长度 （））
| | |0
| |1
| |2
|3
|4
```

缩进显示递归的嵌套级别;垂直线在视觉上将应用程序与其值相关联。请注意，在每次应用长度时，列表都会变小，直到最终达到 （）。（） 处的值为 0，每个外部级别加 1 得出最终值。

让我们编写一个过程，list-copy，它返回其参数的副本，该副本必须是列表。也就是说，list-copy 返回一个新列表，该列表由旧列表的元素（但不是对）组成。如果原始列表或副本可能通过 set-car！ 或 set-cdr！ 进行更改，则制作副本可能很有用，我们将在后面讨论。

```
(列表复制 '()) ⇒ ()
(列表副本 '(a b c)) ⇒ (a b c)
```

在研究下面的定义之前，看看你是否可以定义列表副本。

```
(define list-copy
(lambda (ls)
(if (null? ls)
'()
(cons (car ls)
(list-copy (cdr ls)))))))
```

列表副本的定义类似于长度的定义。基本情况下的检验是相同的（空? ls）。但是，基本情况下的值是 () 而不是 0，因为我们正在构建一个列表，而不是一个数字。递归调用是相同的，但不是添加一个，而是列表复制将列表的 car 转换为递归调用的值。

没有理由不能有一个以上的基本情况。该过程 memv 采用两个参数，一个对象和一个列表。它返回其 car 等于对象的列表的第一个子列表或尾部，如果列表中找不到该对象，则返回#f。memv 的值可以用作条件表达式中的列表或真值。

```
(define memv
(lambda (x ls)
(cond
[(null? ls) #f]
[(eqv? (car ls) x) ls]
[else (memv x (cdr ls))])))

(memv 'a '(a b b d)) ⇒ (a b b d)
(memv 'b '(a b b d)) ⇒ (b b d)
(memv 'c '(a b b d)) ⇒ #f
(memv 'd '(a b b d)) ⇒ (d)
(if (memv 'b '(a b b d))
```

“yes”
“no”） ⇒ “yes”

这里需要检查两个条件，因此使用cond。第一个 cond 子句检查 （） 的基值;没有对象是 （） 的成员，因此答案是#f。第二个子句询问列表中的汽车是否是对象，在这种情况下，列表将返回，它是其汽车包含该对象的第一个尾部。递归步骤只是在列表中继续。

也可能有多个递归情况。与 memv 一样，下面定义的过程 remv 采用两个参数，一个对象和一个列表。它将返回一个新列表，其中包含从列表中删除的所有对象。

```
(define remv
(lambda (x ls)
(cond
[(null? ls) '()]
[(eqv? (car ls) x) (remv x (cdr ls))]
[else (cons (car ls) (remv x (cdr ls)))])))

(remv 'a '(a b b d)) ⇒ (b b d)
(remv 'b '(a b b d)) ⇒ (a d)
(remv 'c '(a b b d)) ⇒ (a b b d)
(remv 'd '(a b b d)) ⇒ (a b b)
```

此定义类似于上面的 memv 定义，只是 remv 在列表中的车中找到元素后不会退出。相反，它继续，只是忽略了这个元素。如果在列表的汽车中找不到该元素，remv将执行与上面的list-copy相同的操作：它将列表的汽车转换为递归值。

到目前为止，递归只在列表的 cdr 上。但是，对于在汽车上重复执行的过程以及列表的cdr，有时很有用。下面定义的过程树副本将对的结构视为树而不是列表，左侧子

树是对的汽车，右侧子树是对的 cdr。它执行与列表复制类似的操作，构建新对，同时保留元素（叶子）。

```
（定义 tree-copy
（lambda （tr）
（if （not （pair? tr））
tr
（cons （tree-copy （car tr））
（tree-copy （cdr tr））））））

（树形副本'（（.b）.c））⇒ （（.b）.c）
```

树结构的自然基参数是任何不是对的东西，因为递归遍历对而不是列表。在这种情况下，递归步骤是双重递归的，以递归方式查找汽车的值以及参数的 cdr。

在这一点上，熟悉提供特殊迭代构造的其他语言（例如，while 或 for 循环）的读者可能会想知道 Scheme 中是否需要类似的构造。这种结构是不必要的;Scheme 中的迭代通过递归更清晰、更简洁地表达出来。递归更通用，消除了许多其他语言的迭代构造所需的变量赋值的需求，从而使代码更可靠，更易于遵循。一些递归本质上是迭代并按此执行;关于这一点，第3.2节还有更多要说的。但是，通常不需要进行区分。相反，专注于编写清晰，简洁和正确的程序。

在我们离开递归主题之前，让我们考虑一种称为映射的特殊形式的重复。请考虑以下过程 abs-all，该过程将数字列表作为输入并返回其绝对值的列表。

```
（定义 abs-all
（lambda （ls）
（if （null? ls）
'()
```

```
（cons （abs （car ls））
（abs-all （cdr ls））））））
```

```
（abs-all ’（1 -2 3 -4 5 -6））⇒ （1 2 3 4 5 6）
```

此过程通过将过程 abs 应用于每个元素，从输入列表中形成一个新列表。我们说abs-all映射abs在输入列表上以产生输出列表。在列表上映射过程是一件相当常见的事情，因此 Scheme 提供了过程映射，该过程映射将其第一个参数（一个过程）映射到其第二个列表之上。我们可以使用map来定义abs-all。

```
（定义 abs-all
（lambda （ls）
（map abs ls））））
```

然而，我们真的不需要abs-all，因为地图的相应直接应用同样简短，也许更清晰。

```
（地图 abs ’（1 -2 3 -4 5 -6））⇒ （1 2 3 4 5 6）
```

当然，我们可以使用 lambda 来创建要映射的过程参数，例如，对数字列表的元素进行平方。

```
（map （lambda （x） （* x x））
’（1 -3 -5 7））⇒ （1 9 25 49）
```

我们可以将多参数过程映射到多个列表上，如以下示例所示。

```
（地图缺点’（a b c）’（1 2 3））⇒ （（a. 1） （b . 2）
（c . 3））
```

列表的长度必须相同，并且该过程应接受与列表一样多的参数。输出列表的每个元素都是将过程应用于输入列

表的相应成员的结果。

查看上面的 abs-all 的第一个定义，在研究之前，您应该能够推导出 map1 的以下定义，map1 是映射的受限版本，它将一个参数过程映射到单个列表上。

```
(define map1
(lambda (p ls)
(if (null? ls)
'()
(cons (p (car ls))
(map1 p (cdr ls))))))

(地图1 abs '(1 -2 3 -4 5 -6)) ⇒ (1 2 3 4 5 6)
```

我们所做的就是将 abs-all 中对 abs 的调用替换为对新参数 p 的调用。更一般地图的定义在第 5.4 节中给出。

练习 2.8.1

描述如果在树复制的定义中将参数的顺序切换为 cons 会发生什么情况。

练习 2.8.2

有关追加的说明，请参阅第 6.3 节，并定义它的双参数版本。如果在追加的定义中切换了调用中参数的追加顺序，会发生什么情况？

练习 2.8.3

定义过程 make-list，它采用非负整数 n 和一个对象，并返回一个长 n 个新列表，其中每个元素都是对象。

（制造列表 7 '（））⇒ （） （） （） （） （） （））

[提示：基本检验应为 （= n 0），递归步骤应涉及 （- n 1）。。而 （） 是列表上递归的自然基数，而 0 是非负整数递归的自然基数。同样，减去 1 是使非负整数接近 0 的自然方法。

练习 2.8.4

过程 list-ref 和 list-tail 返回 list ls 的第 n 个元素和第 n 个尾部。

```
（list-ref '（1 2 3 4） 0）⇒ 1
（list-tail '（1 2 3 4） 0） ⇒ （1 2 3 4）
（list-ref '（一个短（嵌套）列表） 2） ⇒ （嵌套）
（list-tail '（一个短（嵌套）列表） 2） ⇒ （（嵌套）列表）
```

定义这两个过程。

练习 2.8.5

练习 2.7.2 让您在“更短”的定义中使用“长度”，它返回其两个列表参数中的较短参数，如果两个参数具有相同的长度，则返回第一个参数。写得更短，不使用长度。[提示：定义一个递归帮助器，更短？，并用它来代替长度比较。

练习 2.8.6

到目前为止显示的所有递归过程都是直接递归的。也就是说，每个过程都直接将自己应用于新参数。也可以编

写两个相互使用的过程，从而导致间接递归。定义程序奇数和偶数？，每个程序都根据另一个来定义。[提示：当其参数为 0 时，每个应返回什么？

```
（甚至？ 17）⇒ #f
（奇数？17） ⇒ #t
```

练习 2.8.7

使用 map 定义一个过程（转置），该过程采用一对列表并返回一对列表，如下所示。

```
（转置’（（a. 1） （b . 2） （c . 3））⇒ （（a b c） 1 2 3）
```

[提示：（（a b c） 1 2 3） 与 （（a b c） 相同。（1 2 3）.]

第2.9节. 分配

尽管许多程序可以在没有它们的情况下编写，但分配给顶级变量或 let-bound 和 lambda 绑定变量有时很有用。赋值不会像 let 或 lambda 那样创建新绑定，而是更改现有绑定的值。作业是用 set!

```
（定义 abcde ’（a b c d e））
abcde ⇒ （a b c d e）
（set! abcde （cdr abcde））
abcde ⇒ （b c d e）
（let （[abcde ’（a b c d e）]）
（set! abcde （reverse abcde））
abcde） ⇒ （e d c b a）
```

许多语言需要使用赋值来初始化局部变量，这与变量的声明或绑定是分开的。在 Scheme 中，所有局部变量在绑定后立即被赋予一个值。除了使初始化局部变量的单独赋值变得不必要之外，它还确保程序员不会忘记初始化它们，这是大多数语言中常见的错误来源。

事实上，在其他语言中，大多数必要或方便的赋值在Scheme中都是不必要和不方便的，因为通常有更清晰的方式来表达没有赋值的相同算法。某些语言中的一种常见做法是使用一系列赋值对表达式计算进行排序，如下面的过程查找二次方程的根。

```
（定义二次公式
（lambda （a b c）
（let （[root1 0] [root2 0] [minusb 0] [radical 0] [divisor 0]）
（set! minusb （- 0 b））
（set! radical （sqrt （- （* b b） （* 4 （* a c））））））
（set! divisor （* 2 a））
（set! root1 （/ （+ minusb radical） divisor））
（set! root2 （/ （- minusb radical） divisor））
（cons root1 root2））））
```

根是根据众所周知的二次公式计算的，

$$\frac{-b \pm \sqrt{b^2-4ac}}{2a}$$

这就产生了方程 0 = ax^2 + bx + c 的解。此定义中的 let 表达式仅用于建立变量绑定，对应于其他语言所需的声明。前三个赋值表达式计算公式的子段，即 -b、$\sqrt{b^2-4ac}$和 2a。最后两个赋值表达式根据子段计算两个根。两个根中的一对是二次公式的值。例如，2x2 - 4x - 6 的两个根是 x = 3 和 x = -1。

```
（二次式 2 -4 -6）⇒ （3 . -1）
```

上面的定义有效，但是如果没有分配，它可以写得更清楚，如下所示。

```
（定义二次公式
（lambda （a b c）
（let （[minusb （- 0 b）]
[radical （sqrt （- （* b b） （* 4 （* a c））））
[除数 （* 2 a）])
（let （[root1 （/ （（ （ + minusb radical）
divisor）]
[root2 （/ （- minusb radical） divisor）])
（cons root1 root2）））））
```

在这个版本中，set！ 表达式消失了，我们只剩下基本上相同的算法。但是，通过使用两个 let 表达式，该定义清楚地表明了 root1 和 root2 对 minusb、radical 和除数的值的依赖性。同样重要的是，let 表达式清楚地表明了 minusb、radical 和 divisor 之间以及 root1 和 root2 之间缺乏依赖关系。

赋值在 Scheme 中确实有一些用法，否则语言将不支持它们。请考虑以下版本的 cons，该版本对调用它的次数进行计数，并将计数存储在名为 cons-count 的变量中。它使用 set！ 来递增计数;没有作业就无法实现相同的行为。

```
（定义 kons-count 0）
（define kons
（lambda （x y）
（set！ kons-count （+ kons-count 1））
（cons x y））)

（kons ’a ’（b c））⇒ （a b c）
kons-count ⇒ 1
```

```
（kons 'a （kons 'b （kons 'c '（））） ⇒ （a b c）
kons-count ⇒ 4
```

分配通常用于实现必须保持某种内部状态的过程。例如，假设我们想要定义一个过程，该过程在第一次调用时返回 0，第二次返回 1，第三次返回 2，依此类推。我们可以写一些类似于上面缺点计数定义的东西：

```
（定义下一个 0）
（定义计数
（lambda （）
（let （[v next]）
（set！ next （+ next 1））
v））））

（count） ⇒ 0
（count） ⇒ 1
```

此解决方案有些不可取，因为下一个变量在顶层可见，即使它不需要。由于它在顶层是可见的，因此系统中的任何代码都可以更改其值，可能会无意中以微妙的方式影响 count 的行为。我们可以通过在 lambda 表达式之外进行 let 绑定来解决此问题：

```
（定义计数
（让 （[next 0]）
（lambda （）
（let （[v next]）
（set！ next （+ next 1））
v））））
```

后一种解决方案还很容易推广到提供多个计数器，每个计数器都有自己的本地计数器。下面定义的过程生成计数器每次调用时都会返回一个新的计数过程。

```
（定义 make-counter
（lambda （）
（let （[next 0]）
（lambda （）
（let （[v next]）
（set！ next （+ next 1））
v））））））
```

由于 next 绑定在 make-counter 内部，但在 make-counter 返回的过程之外，因此它返回的每个过程都保持其自己的唯一计数器。

```
（定义 count1 （make-counter））
（定义计数2（计数器））

（计数1） ⇒ 0
（计数2） ⇒ 0
（计数1） ⇒ 1
（计数1） ⇒ 2
（计数2） ⇒ 1
```

如果一个状态变量必须由多个在顶层定义的过程共享，但我们不希望状态变量在顶层可见，我们可以使用 let 来绑定变量并 set！ 使过程在顶层可见。

```
（定义嘘#f）
（定义告诉#f）
（让 （[秘密 0]）
（设置！ 嘘
（lambda （消息）
（设置！ 秘密消息））））
（设置！ 告诉
（lambda （）
秘密））））

（嘘 “莎莉喜欢哈利”）
```

（告诉） ⇒ “莎莉喜欢哈利”
秘密⇒异常： 变量秘密不受约束

变量必须先定义，然后才能赋值，因此我们定义了 shhh 并告诉最初#f。（任何初始值都可以。我们将在第 3.5 节中再次看到此结构，并在第 3.6 节中将此类代码构建为库的更好方法。

本地状态有时可用于缓存计算值或允许懒惰地计算计算，即仅按需计算一次。下面的过程懒惰接受 thunk 或零参数过程作为参数。Thunks通常用于“冻结”由于某种原因必须延迟的计算，这正是我们在这种情况下需要做的。当传递 thunk t 时，lazy 将返回一个新的 thunk，该 thunk 在调用时返回调用 t 的值。计算完成后，该值将保存在局部变量中，以便无需再次执行计算。布尔标志用于记录是否已调用 t 并保存其值。

```
（定义懒惰
（lambda （t）
（let （[val #f] [flag #f]）
（lambda （）
（if （not flag）
（begin （set！ val （t））
（set！ flag #t））
val））））
```

此处首次使用的句法形式 begin 按从左到右的顺序计算其子表达式，并返回最后一个子表达式的值，如 let 或 lambda 表达式的主体。我们还看到，可以省略 if 表达式的替代子表达式。仅当 if 的值被丢弃时，才应执行此操作，在这种情况下就是这样。

惰性求值对于需要相当长的计算时间的值特别有用。通过延迟评估，我们可能会完全避免计算值，并且通过保

存值，我们可以避免多次计算它。

惰性的操作可以通过从传递给懒惰的thunk中打印消息来最好地说明。

```
（定义 p
（懒惰 （lambda （）
（显示“哎呀！”）
（换行符）
“得到我”））
```

第一次调用 p 时，将打印消息 Ouch！并返回字符串“got me”。此后，将返回“got me”，但不会打印该消息。过程显示和换行符是我们见过的第一个显式输入/输出示例;display 打印不带引号的字符串，换行符打印换行符。

为了进一步说明 set！ 的用法，让我们考虑一下堆栈对象的实现，其内部工作在外部不可见。堆栈对象接受以下四条消息之一：empty？，如果堆栈为空，则返回 #t;push！，它将一个对象添加到堆栈的顶部;top，返回堆栈顶部的对象;和 pop！，这会删除堆栈顶部的对象。下面给出的过程 make-stack 每次以类似于 make-counter 的方式调用时都会创建一个新堆栈。

```
（define make-stack
（lambda （）
（let （[ls ’（）]）
（lambda （msg . args）
（cond
[（eqv？ msg ’empty？）（空？ ls）]
[（eqv？ msg ’push！）（set！ ls （cons （car args）
ls））]
[（eqv？ msg ’top） （car ls）]
```

```
[（eqv？ msg ’pop！）（set！ ls （cdr ls））]
[else “oops”]））））））
```

每个堆栈都存储为绑定到变量 ls 的列表;设置！用于更改此绑定的推送！ 和 pop！。请注意，内部 lambda 表达式的参数列表使用不正确的列表语法将 arg 绑定到除第一个参数之外的所有参数的列表。这在这里很有用，因为在空？，top和pop的情况下！只有一个参数（消息），但在推送的情况下！有两个（消息和要推送到堆栈上的对象）。

```
（定义堆栈 1（生成堆栈））
（定义 stack2 （make-stack））
（list （stack1 ’empty？）（堆栈2“空？））⇒ （#t #t）

（堆栈 1 ’推！’a）
（列表 （堆栈 1 ’空？）（堆栈2“空？））⇒ （#f #t）

（堆栈 1 ’推！’b）
（stack2 ’push！’c）
（stack1 ’top） ⇒ b
（stack2 ’top） ⇒ c

（stack1 ’pop！
（stack1 ’top） ⇒ a
（list （stack1 ’empty？）（堆栈2“空？））⇒ （#f #f）

（堆栈1’啪！
（列表（堆栈 1’空？）（堆栈2“空？））⇒ （#t #f）
```

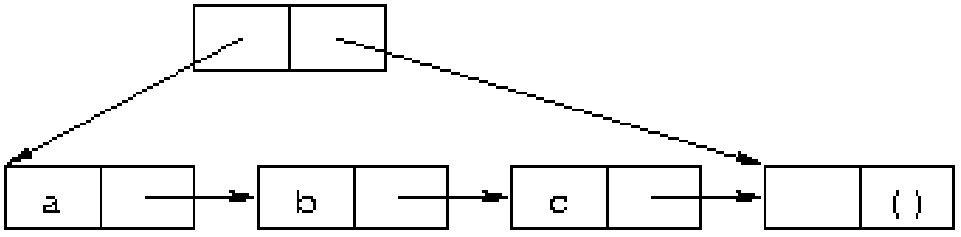
与 make-counter 创建的计数器一样，每个堆栈对象维护的状态只能在对象中直接访问。对此状态的每个引用或更改都是由对象本身显式进行的。一个重要的好处是，我们可以更改堆栈的内部结构，也许使用向量（参见第6.9节）而不是列表来保存元素，而无需更改其外部行为。因为对象的行为是抽象已知的（而不是操作上

的），所以它被称为抽象对象。有关创建抽象对象的更多信息，请参见第 12.8 节。

除了更改变量的值之外，我们还可以使用 set-car！ 和 set-cdr！ 过程来更改一对的 car 和 cdr 字段的值。

```
（定义 p（列表 1 2 3））
（设置车 ！（cdr p）’two）
p ⇒ （1 two 3）
（set-cdr！ p ’（））
p ⇒ （1）
```

我们可以使用这些运算符来定义队列数据类型，它类似于堆栈，除了在一端添加新元素并从另一端提取。以下队列实现使用 tconc 结构。tconc 由非空列表和标题组成。标题是一对，其汽车指向列表的第一对（头部），其 cdr 指向列表的最后一对（结束）。



列表的最后一个元素是占位符，不被视为队列的一部分。

下面定义了对队列的四个操作：构造队列的 make-queue;make-queue，它构造一个队列;putq！，它将元素添加到队列的末尾;getq，它检索队列前面的元素;和 delq！，它删除了队列前面的元素。

```
（定义 make-queue
（lambda （）
（let （[end （cons ’ignore ’（））]）
（cons end end））））
```

```
(定义 putq!
(lambda (q v)
(let ([end (cons 'ignore '() )])
(set-car! (cdr q) v)
(set-cdr! (cdr q) end)
(set-cdr! q end) ) ) )

(define getq
(lambda (q)
(car (car q) ) ) )

(define delq!
(lambda (q)
(set-car! q (cdr (car q) ) ) ) )
```

除了 putq! 之外，其他操作都是简单的操作，它修改了结束对以包含新值并添加了新的结束对。

```
(定义 myq (make-queue) )

(putq! myq 'a)
(putq! myq 'b)
(getq myq) ⇒ a
(delq! myq)
(getq myq) ⇒ b
(delq! myq)
(putq! myq 'c)
(putq! myq 'd)
(getq myq) ⇒ c
(delq! myq)
(getq myq) ⇒ d
```

练习 2.9.1

修改 make-counter 以采用两个参数：计数器使用的初始值代替 0，以及每次递增计数器的量。

练习 2.9.2

在第 5.3 节中查找案例说明。将 make-stack 中的 cond 表达式替换为等效的大小写表达式。添加 mt? 作为空消息的第二个名称。

练习 2.9.3

修改堆栈对象以允许两条消息引用并设置！。（堆栈 'ref i） 应返回堆栈顶部的第 i 个元素;（堆栈 'ref 0） 应等效于 （堆栈 'top）。（堆叠'设置！ i v） 应将堆栈顶部的第 i 个元素更改为 v。

```
（定义堆栈（make-stack））

（堆叠'推！'a）
（堆叠 'push！'b）
（堆叠 '推！'c）

（stack 'ref 0） ⇒ c
（stack 'ref 2） ⇒ a
（stack 'set！ 1 'd）
（stack 'ref 1） ⇒ d
（stack 'top） ⇒ c
（stack 'pop！）
（堆栈顶部） ⇒ d
```

[提示：使用 list-ref 实现 ref，使用 list-tail 与 set-car！ 实现 set！。

练习 2.9.4

方案支持向量和列表。与列表一样，矢量是包含其他对象的聚合对象。与列表不同，矢量具有固定大小，并且

布局在一个平面内存块中，通常带有包含矢量长度的标题，如下面的十个元素矢量所示。

这使得向量更适合需要快速访问聚合的任何元素的应用程序，但不太适合需要根据需要增长和收缩的数据结构的应用程序。

在第 6.9 节中查找基本矢量运算，并重新实现堆栈对象以使用矢量而不是列表来保存堆栈内容。包括练习 2.9.3 的引用和设置！ 消息。让新的 make-stack 接受 size 参数 n 并使向量长度为 n，但不要以其他方式更改外部（抽象）接口。

练习 2.9.5

定义谓词 emptyq?，用于确定队列是否为空。修改 getq 和 delq!，以便在找到空队列时引发异常，使用断言冲突。

练习 2.9.6

在队列实现中，封装列表中的最后一对是占位符，即它从不包含任何有用的东西。对队列运算符重新编码以避免此浪费的对。确保前面给出的一系列队列操作适用于新的实现。您更喜欢哪种实现？

练习 2.9.7

使用 set-cdr!，可以创建循环列表。例如，以下表达式的计算结果为一个列表，该列表的汽车是符号 a，其 cdr 是列表本身。

```
(let ([ls (cons 'a '())])
(set-cdr! ls ls)
ls)
```

当您在交互式方案会话期间输入上述表达式时，会发生什么情况？当给出一个循环列表时，第42页上的长度的实现将做什么？内置长度基元有什么作用？

练习 2.9.8

定义谓词列表?，如果其参数是正确的列表，则返回#t，否则#f（请参见第 6.3 节）。对于循环列表以及由 () 以外的对象终止的列表，它应该返回#f。

```
(列表? '()) ⇒ #t
(列表? '(1 2 3)) ⇒ #t
(列表? '(a . b)) ⇒ #f
(列表? (let ([ls (cons 'a '())])
(set-cdr! ls ls)
ls)) ⇒ #f
```

首先编写一个简化版本的列表?它不处理循环列表，然后扩展它以正确处理循环列表。修改您的定义，直到您满意它尽可能清晰简洁。[提示 ：使用以下“ 兔子和”算法来检测周期。定义两个参数的递归帮助过程，即兔子和。在列表的开头同时启动野兔和。让兔子每次前进一个 cdr 时前进两个 cdr 。如果兔子抓住了，一定有一个周期。

R. Kent Dybvig / The Scheme Programming Language, Fourth Edition

ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93
订购本书 / 关于这本书

http://www.scheme.com