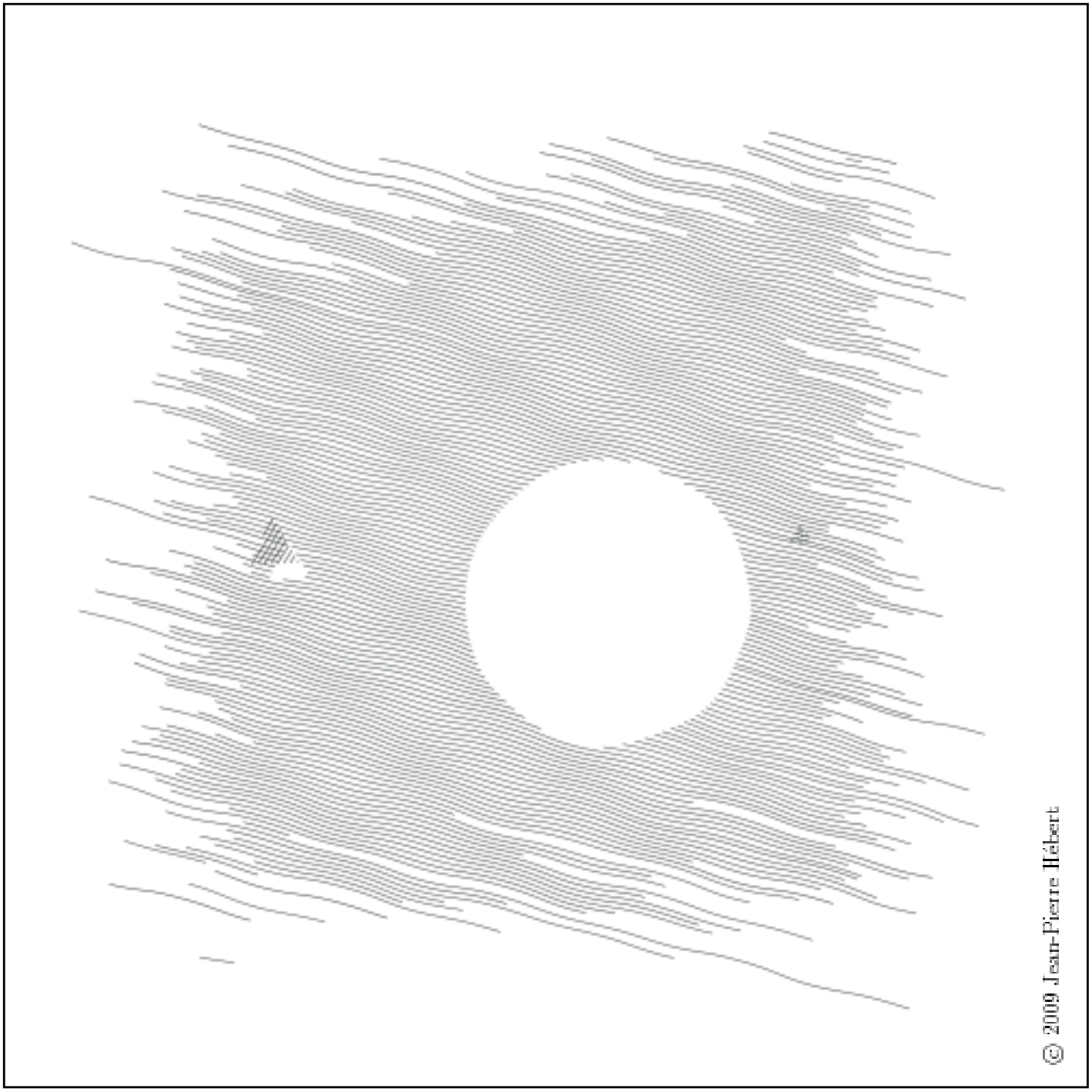

第 3 章。走得更远

上一章准备使用一小组最有用的原始语法形式和过程来编写 Scheme 程序。本章介绍了许多附加功能和编程技术，这些功能和编程技术将允许您编写更复杂、更高效的程序。

第 3.1 节. 句法扩展

正如我们在第 2.5 节中看到的，let 句法形式只是一个句法扩展，根据 lambda 表达式和过程应用程序进行定义，这两者都是核心句法形式。此时，您可能想知道哪些句法形式是核心形式，哪些是句法扩展，以及如何定义新的句法扩展。本节提供了这些问题的一些答案。

事实上，我们没有必要区分核心形式和句法扩展，因为一旦定义，句法扩展就具有与核心形式完全相同的地位。然而，区分使理解语言更容易，因为它使我们能够将注意力集中在核心形式上，并理解所有其他形式。

Scheme 实现必须区分核心形式和句法扩展。Scheme 实现将语法扩展扩展扩展为核心形式，作为编译或解释的第一步，允许编译器或解释器的其余部分仅关注核心形式。但是，扩展后剩余的由编译器或解释器直接处理的核心表单集依赖于实现，并且可能与此处描述为核心的表单集不同。

因此，构成语言核心的确切句法形式集是有争议的，尽管必须有可能从任何一组被宣布为核心形式的形式中推导出所有其他形式。此处描述的集合是满足此约束的最简单集合之一。

核心语法形式包括顶级定义形式、常量、变量、过程应用程序、引号表达式、lambda 表达式、if 表达式和 set! 表达式。下面的语法描述了Scheme根据这些定义和表达式的核心语法。在语法中，竖线（|）将替代项分开，后跟星号（*）的形式表示该形式的零个或多个出现次数。<可变>是任何方案标识符。<基准>是任何 Scheme 对象，例如数字、列表、符号或矢量。<布尔>是#t或#f，<

数字>是任何数字，<字符>是任何字符，<字符串>是任何字符串。我们已经看到了数字、字符串、列表、符号和布尔值的示例。有关这些对象和其他对象的对象级语法的更多信息，请参阅第 6 章或从第 455 页开始的正式语法描述。

```
<程序>      —→<形式>*
<形式>      —→<定义> |<表达>
<定义>      —→<可变定义> |（开始<定义>*)
<可变定义>—→（定义<可变><表达式>)
<表达>      —→<常量>
              | <可变>
              | （引用<基准>)
              | （lambda <形式><表达式><表达式>*)
              | （如果<表达><表达><表达>)
              | （设置！<可变><表达>)
              | <适用>
<常量>      —→<布林> |<编号> |<特征> |<弦>
<形式>      —→<可变>
              | (<可变>*)
              | (<可变><可变>* <可变>)
<适用>      —→(<表达><表达>*)
```

语法不明确，因为过程应用程序的语法与 quote、lambda、if 和 set！ 表达式的语法冲突。为了有资格成为过程应用程序，第一个<表达式>不得是这些关键字之一，除非该关键字已被重新定义或本地绑定。

第 2.6 节中给出的定义“defun”语法未包含在核心中，因为该形式的定义直接转换为更简单的定义语法。同样，if 的核心语法不允许省略替代项，如第 2.9 节中的一个示例所示。缺少替代项的 if 表达式可以转换为 if 的核心语法，只需将缺少的子表达式替换为任意常量（如 #f）。

仅包含定义的开头被视为语法中的定义;这是允许的，以便允许语法扩展扩展到多个定义。begin 表达式，即包含表达式的开始形式，不被视为核心形式。表单的开始表达式

```
（开始 e1 e2 ...）
```

等效于 lambda 应用程序

```
（（lambda （） e1 e2 ...））
```

因此不必被视为核心。

现在我们已经建立了一组核心句法形式，让我们转向讨论句法扩展。语法扩展之所以如此命名，是因为它们将 Scheme 的语法扩展到了核心语法之外。Scheme 程序中的所有句法扩展最终都必须派生自核心形式。然而，一个句法扩展可以根据另一个句法扩展来定义，只要后者在某种意义上“更接近”核心语法。句法形式可以出现在任何预期有表达式或定义的地方，只要扩展形式适当地扩展为定义或表达式即可。

语法扩展是使用定义语法定义的。define-syntax 与 define 类似，不同之处在于 define-syntax 将语法转换过程或转换器与关键字（如 let）相关联，而不是将值与变量相关联。下面是如何使用定义语法来定义 let。

```
(define-syntax let
(syntax-rules ()
[(_ ((x e) ...) b1 b2 ...)
((lambda (x ...) b1 b2 ...) e ...)]))
```

在定义语法之后出现的标识符是正在定义的语法扩展的名称或关键字，在本例中为 let。语法规则形式是计算为转换器的表达式。语法规则后面的项是辅助关键字的列表，并且几乎总是 ()。辅助关键字的一个示例是 cond 的 else。（其他需要使用辅助关键字的示例在第8章中给出。辅助关键字列表后面是一个或多个规则或模式/模板对的序列。在我们的 let 定义中只出现了一个规则。规则的模式部分指定输入必须采用的形式，模板指定输入应转换为什么。

模式应始终是结构化表达式，其第一个元素是下划线 (_)。。（正如我们将在第 8 章中看到的，使用 _ 只是一个约定，但它是一个很好的遵循。如果存在多个规则，则通过在扩展期间按顺序将模式与输入匹配来选择适当的规则。如果没有任何模式与输入匹配，则存在语法冲突。

模式中出现的下划线或省略号以外的标识符是模式变量，除非它们被列为辅助关键字。模式变量匹配任何子结构，并绑定到相应模板中的该子结构。模式中的表示法 pat ... 允许零个或多个表达式与输入中的省略号原型 pat 匹配。同样，模板中的表示法 expr ... 从输出中的省略号原型 expr 生成零个或多个表达式。输入中的拍打次数决定了输出中的外显次数;为了使其正常工作，模板中的任何省略号原型必须至少包含一个来自模式中省略号原型的模式变量。

我们定义let的单一规则应该是相当不言自明的，但有几点值得一提。首先，let的语法要求主体至少包含一种形式;因此，我们指定了 b1 b2 ...而不是 b ...，这可能看起来更自然。另一方面，let不要求至少有一个变量/值对，所以我们能够简单地使用（x e）...。其次，模式变量 x 和 e 虽然在模式中的同一原型中一起，但在模板中是分开的;任何形式的重新排列或重组都是可能的。最后，出现在模式中的省略号原型中的三个模式变量 x、e 和 b2 也出现在模板的省略号原型中。这不是巧合，而是巧合。这是一项要求。通常，如果模式变量出现在模式的省略号原型中，则它不能出现在模板中的省略号原型之外。

和下面的定义比 let 的定义要复杂一些。

```
（定义语法和
（语法规则 （）
[（_） #t]
[（_ e） e]
[（_ e1 e2 e3 ...）
（如果 e1 （和 e2 e3 ...） #f））））
```

此定义是递归的，涉及多个规则。回想一下，（和）评估#t;第一条规则负责处理这种情况。第二条和第三条规则指定递归的基本情况和递归步骤，并将具有两个或多个子表达式的表达式一起转换为嵌套的 if 表达式。例如，（和 a b c） 首先展开为

```
（如果 a（和 b c）#f）
```

然后

```
（如果 a （如果 b （和 c） #f） #f）
```

最后

```
（如果 a （如果 b c #f） #f）
```

通过此展开，如果 a 和 b 的计算结果为真值，则该值是 c 的值，否则根据需要#f。

和下面的版本更简单，但不幸的是，不正确。

```
（定义语法和;不正确!
（语法规则 （）
[（_） #t]
[（_ e1 e2 ...）
（如果 e1 （和 e2 ...） #f））））
```

表达式

```
（和 （不是 （= x 0）） （/ 1 x））
```

当 x 不为零时，应返回 （/ 1 x） 的值。使用 不正确版本的 and，表达式将按如下方式展开。

```
（如果不是 （= x 0）） （和 （/ 1 x）） #f）→
（如果 （不是 （= x 0）） （如果 （/ 1 x） （和） #f） #f） →
（如果 （不是 （= x 0）） （如果 （/ 1 x） #t #f） #f）
```

如果 x 不为零，则最终答案是#t，而不是 （/ 1 x） 的值。

或以下的定义类似于 for 的定义，不同之处在于必须为每个中间值引入一个临时变量，以便我们既可以测试该值，又可以在该值为真值时返回该值。（由于只有一个 false 值，因此不需要临时变量，#f。

```
(define-syntax or
(syntax-rules )
[(_) #f]
[(_ e) e]
[(_ e1 e2 e3 ...)
(let ([t e1])
(if t t (or e2 e3 ...)))))))
```

与由 lambda 或 let 绑定的变量一样，模板引入的标识符在词法上是作用域的，即仅在模板引入的表达式中可见。因此，即使其中一个表达式 e2 e3 ... 包含对 t 的引用，引入的 t 绑定也不会“捕获”这些引用。这通常是通过自动重命名引入的标识符来实现的。

与上面给出的 更简单版本一样，或更低的简单版本是不正确的。

```
(定义语法或 ;不正确!
(语法规则 ()
[(_) #f]
[(_ e1 e2 ...)
(让 ([t e1])
(如果 t t (或 e2 ...)))))))
```

然而，原因更为微妙，并且是练习3.2.6的主题。

练习 3.1.1

写出展开所需的扩展步骤

```
(let ([x (memv 'a ls)])
(和 x (memv 'b x)))
```

变成核心形式。

练习 3.1.2

写出展开所需的扩展步骤

```
（或 （memv x ’（a b c）） （列表 x））
```

变成核心形式。

练习 3.1.3

let* 类似于 let，但按顺序计算其绑定。每个右侧表达式都在早期绑定的范围内。

```
（让我们* （[a 5] [b （+ a）] [c （+ a b）]）
（列表 a b c））⇒ （5 10 15）
```

let* 可以实现为嵌套的 let 表达式。例如，上面的 let* 表达式等效于下面的嵌套 let 表达式。

```
（让 （[a 5]）
（让 （[b （+ a）]）
（让 （[c （+ a b）]）
（列表 a b c））））⇒ （5 10 15）
```

使用定义语法定义 let*。

练习 3.1.4

正如我们在第 2.9 节中所看到的，省略 if 表达式的第三个或替代子表达式是合法的。然而，这样做往往会导致混乱。方案提供了两种句法形式，当和除非，可以使用它们来代替这种“单臂”如果表达。

```
（当测试 expr1 expr2 时 ...）
（除非测试 expr1 expr2 ...）
```

对于这两种形式，首先评估测试。当 test 的计算结果为 true 时，其余表单将按顺序计算，就像包含在隐式开始表达式中一样。如果检验的计算结果为 false，则不评估其余形式，并且未指定结果。除非相似，但仅当检验计算结果为 false 时才评估其余形式。

```
（让 （[x 3]）
（除非 （= x 0） （设置！ x （+ x 1））））
（当 （= x 4） （设置！ x （* x 2））））
x）⇒ 8
```

根据 if 和 begin 将 when 定义为句法扩展，并定义“除非”（when）。

第 3.2 节. 更多 递归

在第 2.8 节中，我们看到了如何使用顶级定义来定义递归过程。在此之前，我们了解了如何使用 let 为过程创建本地绑定。很自然地想知道 let-bound 过程是否可以递归。答案是否定的，至少不是直截了当的。如果尝试计算表达式

```
（let （[sum （lambda （ls）
（if （null？ ls）
0
（+ （car ls） （sum （cdr ls））））））
（sum ’（1 2 3 4 5））））
```

它可能会引发一个异常，并显示一条消息，大意是 sum 是未定义的。这是因为变量 sum 仅在 let 表达式的主体内可见，而在值绑定为 sum 的 lambda 表达式中不可

见。我们可以通过将过程总和传递给自身来解决此问题，如下所示。

```
(let ([sum (lambda (sum ls)
(if (null? ls)
0
(+ (car ls) (sum sum (cdr ls)))))])
(sum sum '(1 2 3 4 5))) ⇒ 14
```

这是有效的，是一个聪明的解决方案，但有一个更简单的方法，使用letrec。与 let 一样，letrec 语法形式包括一组变量值对，以及一系列称为 letrec 主体的表达式。

```
(letrec ((var expr) ...) 身体1 身体2 ...)
```

与 let 不同，变量 var ... 不仅在 letrec 的主体内可见，而且在 expr ... 内也可见。因此，我们可以重写上面的表达式，如下所示。

```
(letrec ([sum (lambda (ls)
(if (null? ls)
0
(+ (car ls) (sum (cdr ls)))))])
(sum '(1 2 3 4 5))) ⇒ 14
```

使用 letrec，我们还可以定义相互递归的过程，例如过程偶数和奇数？这是练习 2.8.6 的主题。

```
(莱特雷克([甚至?
(lambda (x)
(or (= x 0)
(奇数? (- x 1)))]]
[奇怪?
(lambda (x)
(and (not (= x 0))
```

```
（甚至？（- x 1））））））
（列表（偶数？20）（奇数？20））（ ⇒ #t #f）
```

在 letrec 表达式中，expr... 通常是 lambda 表达式，尽管情况并非如此。但是，必须遵守对表达式的一个限制。必须能够在不计算任何变量 var ... 的情况下评估每个 expr。如果表达式都是 lambda 表达式，则始终满足此限制，因为即使变量可能出现在 lambda 表达式中，在 letrec 的主体中调用结果过程之前，也无法计算它们。以下 letrec 表达式遵循此限制。

```
（letrec （[f （lambda （） （+ x 2））]
[x 1]）
（f））⇒ 3 个
```

而以下情况则不然。

```
（莱特累克 （[y （+ x 2）]
[x 1]）
y）
```

在这种情况下，将引发异常，指示未在引用 x 的位置定义 x。

我们可以使用 letrec 来隐藏 “help” 过程的定义，以便它们不会使顶级命名空间混乱。下面的 list ？ 定义证明了这一 点，该定义遵循练习 2．9．8 中概述的 “ 兔子和”算法。

```
（定义列表？
（lambda （x）
（letrec （[race
（lambda （h t）
（if （pair？ h）
（let （[h （cdr h）]）
```

```
(if (pair? h)
(and (not (eq? h t))
(race (cdr h) (cdr t))
(null? h)))
(null? h))
(race x x)))))
```

当递归过程仅在过程外部的一个位置调用时（如上面的示例所示），使用命名 let 表达式通常更清晰。命名 let 表达式采用以下形式。

```
(让名称 ((var expr) ...)
身体1 身体2 ...)
```

命名 let 类似于未命名的 let，它将变量 var ... 绑定到 body1 body2 ... 中的 expr ... 的值。与未命名的 let 一样，变量仅在主体内可见，而不在 expr 内可见。。此外，变量名称在正文中绑定到一个过程，该过程可以被调用以重复出现;过程的参数将成为变量 var ...的新值。

list? 的定义已在下面重写为使用命名 let。

```
(定义列表?
(lambda (x)
(let race ([h x] [t x])
(if (pair? h)
(let ([h (cdr h)])
(if (pair? h)
(and (not (eq? h t))
(race (cdr h) (cdr t))
(null? h)))
(null? h)))))
```

正如 let 可以表示为 lambda 表达式对参数的简单直接应用一样，名为 let 可以表示为递归过程对参数的应用。表单的命名 let

```
（让名称 （（var expr） ...）
身体1 身体2 ...）
```

可以按照以下方式重写 letrec。

```
（（letrec （（name （lambda （var ...） body1 body2
...））））
名称）
expr ...）
```

或者，可以将其重写为

```
（letrec （（name （lambda （var ...） body1 body2
...））））
（名称 expr ...））
```

前提是变量名称在 expr ... 中不显示为自由。

正如我们在 2.8 节中所讨论的，一些递归本质上是迭代并按此执行。当过程调用相对于 lambda 表达式处于尾部位置（见下文）时，它被视为尾部调用，Scheme 系统必须将其正确视为“转到”或跳转。当过程尾部调用自身或通过一系列尾部调用间接调用自身时，结果是尾部递归。由于尾部调用被视为跳转，因此尾部递归可用于无限期迭代，而不是其他编程语言提供的更严格的迭代构造，而不必担心溢出任何类型的递归堆栈。

如果调用的值直接从 lambda 表达式返回，则调用相对于 lambda 表达式处于尾部位置，也就是说，如果在调用后除了从 lambda 表达式返回之外，还剩下任何事情要做。例如，如果调用是 lambda 表达式主体中的最后一个表达式，则调用处于尾部位置，如果是尾部位置的 if 表达式的后续或替代部分，则调用处于尾部位置，是尾部位置的 and/ 或 表达式的最后一个子表达式，是 let 或 letrec

在尾部位置的最后一个表达式， 等。在下面的表达式中，对 f 的每个调用都是尾部调用，但对 g 的调用不是。

```
(lambda () (f (g)))
(lambda () (if (g) (f) (f)))
(lambda () (let ([x 4]) (f)))
(lambda () (or (g) (f)))
```

在每种情况下，对 f 的调用的值都是直接返回的，而对 g 的调用则不是。

一般的递归，特别是命名的let提供了一种自然的方式来实现许多算法，无论是迭代，递归，还是部分迭代和部分递归;程序员没有两种截然不同的机制。

以下两个名为 let 表达式的阶乘用法定义用于计算非负整数 n 的阶乘 n！。第一个采用递归定义 n！= n × （n - 1）！，其中 0！定义为 1。

```
(定义阶乘
(lambda (n)
(let fact ([i n])
(if (= i 0)
1
(* i (fact (- i 1)))))))

(阶乘 0) ⇒ 1
(阶乘 1) ⇒ 1
(阶乘 2) ⇒ 2
(阶乘 3) ⇒ 6
(阶乘 10) ⇒ 3628800
```

第二个是采用迭代定义 n 的迭代版本！= n × （n - 1） × （n - 2） × ... × 1，使用累加器 a 来保存

中间产物。

```
（定义阶乘
（lambda （n）
（let fact （[i n] [a 1])
（if （= i 0）
a
（fact （- i 1） （* a i））））））
```

一个类似的问题是计算给定n的第n个斐波那契数列。斐波那契数列是整数的无限序列，0，1，1，2，3，5，8等，其中每个数字都是序列中前面两个数字的总和。计算第n个斐波那契数列的过程最自然地以递归方式定义，如下所示。

```
（定义斐波那契
（lambda （n）
（let fib （[i n])
（cond
[（= i 0） 0]
[（= i 1） 1]
[else （+ （fib （- i 1））（fib （- i 2）））））））

（斐波那契 0） ⇒ 0
（斐波那契 1） ⇒ 1
（斐波那契 2） ⇒ 1
（斐波那契 3） ⇒ 2
（斐波那契 4） ⇒ 3
（斐波那契 5） ⇒ 5
（斐波那契 6） ⇒ 8
（斐波那契 20） ⇒ 6765
（斐波那契 30） ⇒ 832040
```

此解决方案要求在每一步中计算前面的两个斐波那契数列，因此是双重递归的。例如，计算（斐波那契4）需要同时计算（斐波那契3）和（菲布2），计算（斐波那契3）需要

同时计算（斐波那契2）和（斐波那契1），计算（斐波那契2）需要同时计算（斐波那契1）和（斐波那契0）。这是非常低效的，并且随着n的增长，它变得更加低效。更有效的解决方案是调整上面阶乘示例的累加器解，以使用两个累加器，a1表示当前斐波那契数，a2表示前一个累加器。

```
（定义斐波那契
（lambda （n）
（if （= n 0）
0
（let fib （[i n] [a1 1] [a2 0]）
（if （= i 1）
a1
（fib （- i 1） （+ a1 a2） a1））））））
```

此处，零被视为特殊情况，因为没有前面的值。这允许我们使用单个基本情况（= i 1）。使用此迭代解计算第 n 个斐波那契数列所需的时间随 n 线性增长，与双递归版本相比，这会产生显著差异。为了了解差异，请尝试使用这两个定义进行计算（斐波那契35）和（斐波那契40），以查看每个定义需要多长时间。

我们还可以通过查看小输入上每个跟踪来感受差异。下面的第一条迹线显示了斐波那契的非尾递归版本中对 fib 的调用，输入为 5。

```
|（fib 5）
|（fib 4）
| |（fib 3）
| |（fib 2）
| | |（fib 1）
| | |1
| | |（fib 0）
| | |0
| |1
| |（fib 1）
```

```
| |1
| |2
| | (fib 2)
| | (fib 1)
| |1
| | (fib 0)
| |0
| |1
|3
| (fib 3)
| | (fib 2)
| | (fib 1)
| |1
| | (fib 0)
| |0
| |1
| | (fib 1)
| |1
|2
|5
```

请注意，有多个参数为 2、1 和 0 的 fib 调用。第二条跟踪显示尾递归版本中对 fib 的调用，同样输入为 5。

```
| (fib 5 1 0)
| (fib 4 1 1)
| (斐比 3 2 1)
| (斐比 2 3 2)
| (斐比卜 1 5 3)
|5
```

显然，这是有很大区别的。

到目前为止，显示的命名 let 示例要么是尾递归的，要么是非尾递归的。经常发生的情况是，同一表达式中的一个递归调用是尾递归调用，而另一个则不是。下面的

因子定义计算其非负整数参数的质因数。对 f 的第一次调用不是尾递归，但第二次调用是尾递归。

```
(define factor
(lambda (n)
(let f ([n n] [i 2])
(cond
[(>= i n) (list n)]
[(integer? (/ n i))
(缺点 i (f (/ n i) i))]
[否则 (f n (+ i 1))])))))

(系数 0) ⇒ (0)
(系数 1) ⇒ (1)
(系数 12) ⇒ (2 2 3)
(系数 3628800) ⇒ (2 2 2 2 2 2 2 2 2 3 3 3 5 7)
(系数 9239) ⇒ (9239)
```

在下面的(因子 120)的评估中，Chez Scheme 中通过将 let 替换为 trace-let 而产生的对 f 的调用的跟踪突出显示了非尾部调用和尾部调用之间的差异。

```
|(f 120 2)
|(f 60 2)
| |(f 30 2)
| |(f 15 2)
| |(f 15 3)
| | |(f 5 3)
| | |(f 5 4)
| | |(f 5 5)
| | |(5)
| |(3 5)
| |(2 3 5)
|(2 2 3 5)
|(2 2 2 3 5)
```

对 f 的非尾部调用相对于其调用方显示为缩进，因为调用方仍处于活动状态，而尾部调用显示为相同缩进级别。

练习 3.2.1

第 3.2 节中定义的哪些递归过程是尾递归过程，哪些不是？

练习 3.2.2

使用 letrec 重写因子来绑定 f 来代替命名 let。您更喜欢哪个版本？

练习 3.2.3

下面的 letrec 表达式是否可以使用命名 let 重写？如果没有，为什么不呢？如果是这样，请执行。

```
（莱特雷克（[甚至？
（lambda （x）
（or （= x 0）
（奇数？ （- x 1））））]]
[奇怪？
（lambda （x）
（and （not （= x 0））
（甚至？ （- x 1））））））
（甚至？ 20））
```

练习 3.2.4

重写本节中给出的斐波那契的两个定义，以计算对 fib 的递归调用次数，使用类似于第 2.9 节的 cons-count 示例中使用的计数器。计算在每种情况下对多个输入值进行的递归调用数。你注意到了什么？

练习 3.2.5

使用两个规则扩充第 3.1 节中给出的 let 的定义，以处理命名 let 和未命名 let。

练习 3.2.6

以下定义比第 3.1 节中给出的定义更简单。

```
（定义语法或 ;不正确！
（语法规则 （）
[（_） #f]
[（_ e1 e2 ...）
（让 （[t e1]）
（如果 t t （或 e2 ...））））））
```

说出为什么它是不正确的。[提示：想想如果这个版本的或 被用在偶数？ 和奇数？ 第 66 页上给出的非常大的输入中会发生什么。

练习 3.2.7

因子的定义不是最有效的。首先，除了 n 本身之外，不可能在 之外$\sqrt{n}$找到 n 的因子。其次，当找到因子时，除法 （/ n i） 执行两次。第三，在2之后，不可能找到偶数因素。重新编码因子以更正所有三个问题。哪个是最重要的问题要解决？是否有任何其他改进？

第 3.3 节. 延续

在计算 Scheme 表达式期间，实现必须跟踪两件事：（1） 要评估的内容和 （2） 如何处理值。考虑以下表达式中 （null? x） 的计算。

```
（if （null? x） （quote （） ） （cdr x） ）
```

实现必须首先评估 （null? x），然后根据其值评估 （quote （） ） 或 （cdr x）。“要评估什么”是（null? x），而“如何处理值”是决定（引用（）和（cdr x）中的哪一个进行评估并这样做。我们将“如何处理值”称为计算的延续。

因此，在任何表达式的计算过程中的任何时候，都有一个延续准备完成或至少继续从该点开始计算。假设 x 的值为 （a b c）。我们可以在评估（如果（空? x）（引用（））（cdr x））期间隔离六个延续，这些延续等待

1. 的值 （如果 （空? x） （引号 （） ） （cdr x） ） ，
2. 的值 （空? x），
3. 空值?，
4. x 的值，
5. cdr 的值，以及
6. x 的值（再次）。

未列出 （cdr x） 的延续，因为它与等待的继续相同（如果 （null? x） （引号 （） ） （cdr x） ） ）。

方案允许使用过程调用/cc 捕获任何表达式的延续。call/cc 必须传递给一个参数的过程 p。call/cc 构造当前延续的具体表示形式，并将其传递给 p。延续本身由

过程 k 表示。每次将 k 应用于值时，它都会将该值返回到 call/cc 应用程序的延续。实质上，此值成为 call/cc 应用程序的值。

如果 p 在不调用 k 的情况下返回，则过程返回的值将成为 call/cc 应用程序的值。

请考虑下面的简单示例。

```
(call/cc
(lambda (k)
(* 5 4)))⇒ 20

(call/cc
(lambda (k)
(* 5 (k 4))))⇒ 4

(+ 2
(call/cc
(lambda (k)
(* 5 (k 4))))) ⇒ 6
```

在第一个示例中，连续被捕获并绑定到 k，但从不使用 k，因此该值只是 5 和 4 的乘积。在第二种情况下，在乘法之前调用延续，因此值是传递给延续的值 4。在第三个中，延续包括增加2;因此，该值是传递给延续的值 4 加 2。

下面是一个不太平凡的示例，显示了如何使用 call/cc 从递归提供非本地退出。

```
(define product
(lambda (ls)
(call/cc
(lambda (break)
(let f ([ls ls])
```

```
（cond
[（null？ ls） 1]
[（= （car ls） 0） （break 0）]
[else （* （car ls） （f （cdr ls））））

（产品’（1 2 3 4 5））⇒ 120
（产品 ’（7 3 8 0 1 9 5）） ⇒ 0
```

当检测到零值时，非局部退出允许产品立即返回，而无需执行挂起的乘法。

上述每个延续调用都返回到延续，而控制权仍保留在传递给 call/cc 的过程中。下面的示例在返回此过程后使用延续。

```
（let （[x （call/cc （lambda （k） k））]）
（x （lambda （ignore） “hi”））） ⇒ “嗨”
```

通过调用 call/cc 的调用捕获的延续可以描述为“获取值，将其绑定到 x，然后将 x 的值应用于（lambda （忽略）”hi“）的值。由于 （lambda （k） k） 返回其参数，因此 x 绑定到延续本身;此延续应用于由 （lambda （忽略） “hi”） 的评估生成的过程。这具有将 x（再次！）绑定到此过程并将该过程应用于自身的效果。该过程忽略其参数并返回“hi”。

上面示例的以下变体可能是其规模中最令人困惑的Scheme程序;可能很容易猜到它返回的内容，但需要一些思考才能弄清楚为什么。

```
（（（call/cc （lambda （k） k）） （lambda （x） x））
“HEY！”）⇒ “嘿！”
```

call/cc 的值是其自身的延续，如前面的示例所示。这应用于标识过程 （lambda （x） x），因此 call/cc 将使用

此值第二次返回。然后，将标识过程应用于自身，从而生成标识过程。这最终应用于“嘿！”，产生“嘿！”

以这种方式使用的延续并不总是那么令人费解。请考虑以下阶乘定义，该定义通过分配顶级变量重试，在返回 1 之前将延续保存在递归的基数处。

```
（定义重试#f）

（定义阶乘
（lambda （x）
（if （= x 0）
（call/cc （lambda （k） （set! retry k） 1））
（* x （阶乘 （- x 1））））））
```

有了这个定义，阶乘的工作方式与我们期望的阶乘一样，除了它具有分配重试的副作用。

```
（阶乘 4）⇒ 24
（重试 1） ⇒ 24
（重试 2） ⇒ 48
```

绑定到重试的延续可以描述为“将值乘以1，然后将此结果乘以2，然后将此结果乘以3，然后将此结果乘以4。如果我们给延续传递一个不同的值，即不是1，我们将导致基值不是1，从而改变最终结果。

```
（重试 2）⇒ 48
（重试 5） ⇒ 120
```

此机制可以成为使用 call/cc 实现的断点包的基础;每次遇到断点时，都会保存断点的延续，以便可以从断点重新启动计算（如果需要，可以多次重新启动）。

延续可用于实现各种形式的多任务处理。下面定义的简单“轻量级过程”机制允许多个计算交错。由于它是非抢占式的，因此它要求每个进程不时自愿“暂停”，以便允许其他进程运行。

```
(定义 lwp-list '())
(define lwp
(lambda (thunk)
(set! lwp-list (append lwp-list (list
thunk)))))

(define start
(lambda ()
(let ([p (car lwp-list)])
(set! lwp-list (cdr lwp-list))
(p))))

(定义暂停
(lambda ()
(call/cc
(lambda (k)
(lwp (lambda () (k #f)))
(start)))))
```

以下轻量级过程合作打印包含“嘿!

```
(lwp (lambda () (let f () (pause) (display
“h”) (f)))))
(lwp (lambda () (let f () (pause) (display
“e”) (f)))))
(lwp (lambda () (let f () (pause) (display
“y”) (f)))))
(lwp (lambda () (let f () (pause) (display
“!”) (f))))
(lwp (lambda () (let f () (pause) (newline)
(f)))))
(start) ⇒ hey!
```

嘿！
嘿！
嘿！
⋮

有关支持使用 call/cc 进行抢占式多任务处理的引擎的实现，请参见第 12.11 节。

练习 3.3.1

使用 call/cc 编写一个无限期循环的程序，打印从零开始的数字序列。不要使用任何递归过程，也不要使用任何赋值。

练习 3.3.2

在不调用/cc 的情况下重写产品，保留在任何列表元素为零时不执行乘法的功能。

练习 3.3.3

如果由上面定义的 lwp 创建的进程终止，即只是返回而不调用 pause，会发生什么情况？定义一个退出过程，该过程允许进程终止，而不会影响 lwp 系统。请务必处理唯一剩余进程终止的情况。

练习 3.3.4

每次调用 lwp 时，都会复制进程列表，因为 lwp 使用 append 将其参数添加到进程列表的末尾。修改原始 lwp

代码以使用第 2.9 节中开发的队列数据类型以避免此问题。

练习 3.3.5

轻量级流程机制允许动态创建新流程，尽管本节中给出的示例不会这样做。设计一个需要动态创建新流程的应用程序，并使用轻量级流程机制实现它。

第 3.4 节. 延续传递样式

正如我们在上一节中讨论的那样，延续等待每个表达式的值。特别是，延续与每个过程调用相关联。当一个过程通过非尾调用调用另一个过程时，被调用的过程将接收一个隐式延续，该延续负责完成调用过程主体的剩余部分并返回到调用过程的延续。如果调用是尾部调用，则被调用过程仅接收调用过程的延续。

我们可以通过在每次调用传递的显式过程参数中封装“要做什么”来显式继续。例如，对 f 的调用的继续

```
(letrec ([f (lambda (x) (cons 'a x))]
[g (lambda (x) (cons 'b (f x)))]
[h (lambda (x) (g (cons 'c x)))])
(cons 'd (h '())))⇒ (d b a c)
```

将符号 b 转换为返回给它的值，然后将此缺点的结果返回到对 g 的调用的延续。此延续与调用 h 的延续相同，后者将符号 d 与返回给它的值相加。我们可以用延续传递风格或CPS重写它，用显式过程替换这些隐式延续。

```
(letrec ([f (lambda (x k) (k (cons 'a x)))]
[g (lambda (x k)
(f x (lambda (v) (k (cons 'b v)))))]
[h (lambda (x k) (g (cons 'c x) k))])
(h '() (lambda (v) (cons 'd v))))
```

就像前面例子中 h 和 g 的隐式延续一样，显式延续传递给 h 并传递给 g，

```
(lambda (v) (cons 'd v))
```

将符号 d 转换为传递给它的值。同样，延续传递给 f，

```
(lambda (v) (k (cons 'b v)))
```

将 b 转换为传递给它的值，然后将其传递给 g 的延续。

当然，用 CPS 编写的表达式更复杂，但这种编程风格有一些有用的应用。CPS 允许过程将多个结果传递给其延续，因为实现延续的过程可以采用任意数量的参数。

```
(define car&cdr
(lambda (p k)
(k (car p) (cdr p))))

(car&cdr '(a b c)
(lambda (x y)
(list y x))) ⇒ ((b c) a)
(car&cdr '(a b c) cons) ⇒ (a b c)
(car&cdr '(a b c a d) memv) ⇒ (a d)
```

（这也可以使用多个值来完成;请参阅第 5.8 节。CPS 还允许过程采用单独的“成功”和“失败”延续，这可以接受不同数量的参数。下面是整数除法，它将前两个参数的商和余数传递给第三个参数，除非第二个参数（除

数）为零，在这种情况下，它会将错误消息传递给其第四个参数。

```
（定义整数除法
（lambda （x y success failure）
（if （= y 0）
（failure “除以零”）
（let （[q （quotient x y）]）
（success q （- x （* q y））））））

（整数除法 10 3 列表 （lambda （x） x））⇒（3 1）
（整数除法 10 0 列表 （lambda （x） x））⇒“除以零”
```

由整数除法采用的过程商返回其两个参数的商，截断为零。

显式成功和失败延续有时有助于避免将过程的成功执行与失败的执行分开所需的额外通信。此外，对于不同风格的成功或失败，可以有多个成功或失败延续，每个可能采用不同的数字和类型的参数。有关采用延续传递样式的扩展示例，请参见第 12.10 节和第 12.11 节。

此时，您可能想知道 CPS 与通过 call/cc 捕获的延续之间的关系。事实证明，任何使用 call/cc 的程序都可以在没有 call/cc 的情况下在 CPS 中重写，但可能需要完全重写程序（有时甚至包括系统定义的基元）。在查看下面的版本之前，请尝试将第 75 页上的产品示例转换为 CPS。

```
（定义产品
（lambda （ls k）
（let （[break k]）
（let f （[ls ls] [k k]）
（cond
[（null? ls） （k 1）]
```

```
[ (= (car ls) 0) (break 0) ]
[else (f (cdr ls)
 (lambda (x)
 (k (* (car ls) x) ))

 (产品 '(1 2 3 4 5) (lambda (x) x) ) ⇒ 120
 (产品 '(7 3 8 0 1 9 5) (lambda (x) x) ) ⇒ 0
```

练习 3.4.1

重写第 2.1 节中首先给出的倒数示例，以接受成功和失败的延续，如上面的整数除法。

练习 3.4.2

重写第 75 页中的重试示例以使用 CPS。

练习 3.4.3

在 CPS 中重写以下表达式以避免使用 call/cc。

```
 (定义倒数
 (lambda (ls)
 (call/cc
 (lambda (k)
 (map (lambda (x)
 (if (= x 0)
 (k "zero found")
 (/ 1 x) ) )
ls) ) ) )

 (互惠'(2 1/3 5 1/4) ) ⇒ (1/2 3 1/5 4)
 (倒数 '(2 1/3 0 5 1/4) ) ⇒ "零发现"
```

[提示：地图的单列表版本在第 46 页上定义。

第 3.5 节. 内部定义

在第 2.6 节中，我们讨论了顶级定义。定义也可能出现在 lambda、let 或 letrec 主体的前面，在这种情况下，它们创建的绑定是主体的局部绑定。

```
(定义 f (lambda (x) (* x x)))
(let ([x 3])
(define f (lambda (y) (+ y x)))
(f 4)) ⇒ 7
(f 4) ⇒ 16
```

受内部定义约束的过程可以是相互递归的，就像 letrec 一样。例如，我们可以使用内部定义重写第 3.2 节中的偶数和奇数示例，如下所示。

```
(让 ()
(定义偶数?
(lambda (x)
(or (= x 0)
(奇数? (- x 1)))))
(定义奇数?
(lambda (x)
(and (not (= x 0))
(甚至? (- x 1)))))
(甚至? 20)) ⇒ #t
```

同样，我们可以在我们的第一个列表定义中，用种族的内部定义来代替使用 letrec来约束种族。

```
(定义列表?
(lambda (x)
(define race
```

```
(lambda (h t)
(if (pair? h)
(let ([h (cdr h)])
(if (pair? h)
(and (not (eq? h t))
(race (cdr h) (cdr t)))
(null? h)))

(race x)))  (race x)))
```

事实上，内部变量定义和 letrec 实际上是可以互换的。除了语法上的明显差异之外，唯一的区别是变量定义保证从左到右计算，而 letrec 的绑定可以按任何顺序计算。因此，我们不能用 letrec 表达式完全替换包含内部定义的 lambda、let 或 letrec 主体。但是，我们可以使用 letrec*，它和 let*一样，保证了从左到右的评估顺序。表单的主体

```
(定义 var expr0)
...
expr1
expr2
...
```

等效于 letrec* 表达式将定义的变量绑定到包含表达式的主体中的关联值。

```
(letrec* ((var expr0) ...) expr1 expr2 ...)
```

相反，形式的 letrec*

```
(letrec* ((var expr0) ...) expr1 expr2 ...)
```

可以替换为包含内部定义和正文中表达式的 let 表达式，如下所示。

```
 (let ()
 (define var expr0)
 ...
expr1
expr2
 ...
)
```

这些转换之间似乎缺乏对称性，这是因为 letrec* 表达式可以出现在表达式有效的地方，而内部定义只能出现在主体的前面。因此，在用内部定义替换 letrec* 时，我们通常必须引入一个 let 表达式来保存定义。

内部定义与 letrec 或 letrec* 之间的另一个区别是，语法定义可能出现在内部定义中，而 letrec 和 letrec* 仅绑定变量。

```
 (let ([x 3])
 (define-syntax set-x!
 (语法规则 ()
[(_ e) (set! x e)]))
 (设置 x! (+ x x)
x) ⇒ 6
```

与内部变量定义一样，由内部语法定义建立的句法扩展的范围仅限于语法定义所在的主体。

内部定义可以与顶级定义和分配结合使用，以帮助模块化程序。程序的每个模块应仅使其他模块所需的那些绑定可见，同时隐藏其他绑定，否则这些绑定会使顶级命名空间混乱，并可能导致意外使用这些绑定或重新定义。构建模块的常用方法如下所示。

```
 (定义导出变量#f)
 ...
 (let ()
 (define var expr)
```

```
...
init-expr
...
 (set! export-var export-val)
...
)
```

第一组定义为我们想要导出的变量建立顶级绑定（使全局可见）。第二组定义建立仅在模块中可见的本地绑定。表达式 init-expr ... 执行在建立本地绑定后必须执行的任何初始化。最后，set! 表达式将导出的变量分配给适当的值。

这种模块化形式的一个优点是，在程序开发过程中可以删除或“注释掉”括号中的let表达式，使内部定义成为顶级，以便于交互式测试。这种形式的模块化也有几个缺点，正如我们在下一节中讨论的那样。

以下模块导出单个变量 calc，该变量绑定到实现简单四函数计算器的过程。

```
 (定义计算#f)
 (let ()
 (define do-calc
 (lambda (ek expr)
 (cond
[ (number? expr) expr]
[ (and (list? expr) (= (length expr) 3) )
 (let ([op (car expr) ] [args (cdr expr) ])
 (case op
[ (add) (apply-op ek + args) ]
[ (sub) (apply-op ek - args) ]
[ (mul) (apply-op ek * args) ]
[ (div) (apply-op ek / args) ]
[else (complain ek “invalid operator” op) ]) ) ]
[else (complain ek “invalid expression”
expr) ]) ) )
```

```
(define apply-op
(lambda (ek op args)
(op (do-calc ek (car args)) (do-calc ek (cadr
args)))))
(define complain
(lambda (ek msg expr)
(ek (list msg expr))))
(set! calc
(lambda (expr)
; grab an error continuation ek
(call/cc
(lambda (ek)
(do-calc ek expr)))))

(calc '(add (mul 3 2) -4)) ⇒ 2
(计算 '(div 1/2 1/6)) ⇒ 3
(calc '(add (mul 3 2) (div 4))) ⇒ ("invalid
expression" (div 4))
(calc '(mul (add 1 -2) (pow 2 7))) ⇒
("invalid operator" pow)
```

此示例使用大小写表达式来确定要应用的运算符。case 与 cond 类似，只是测试始终相同：(memv val (key ...))，其中 val 是第一个 case 子窗体的值，(key ...) 是每个 case 子句前面的项目列表。可以使用 cond 重写上面示例中的 case 表达式，如下所示。

```
(let ([temp op])
(cond
[(memv temp '(add)) (apply-op ek + args)]
[(memv temp '(sub)) (apply-op ek - args)]
[(memv temp '(mul)) (apply-op ek * args)]
[(memv temp '(div)) (apply-op ek / args)]
[else (complain ek "invalid operator" op)]))
```

练习 3.5.1

在计算示例中将抱怨重新定义为等效的句法扩展。

练习 3.5.2

在 calc 示例中，错误延续 ek 在每次调用 apply-op、complain 和 do-calc 时传递。根据需要将 apply-op、complain 和 do-calc 的定义向内移动，以从这些过程的定义和应用程序中消除 ek 参数。

练习 3.5.3

从计算中消除 call/cc 并重写投诉以使用断言违规引发异常。

练习 3.5.4

扩展计算以处理一元减号表达式，例如，

（计算'（减号（加2 3））） ⇒ -5

以及您选择的其他运营商。

第 3.6 节. 图书馆

在上一节的末尾，我们讨论了一种模块化形式，它涉及从 let 中分配一组顶级变量，同时将未发布的帮助程序保留在 let 的本地。这种形式的模块化有几个缺点：

- 它是不可移植的，因为修订后的[6]报告无法保证交互式顶级的行为甚至存在。

- 它需要赋值，这使得代码看起来有些尴尬，并可能抑制编译器分析和优化。
- 它不支持关键字绑定的发布，因为没有要为关键字设置的模拟项！。

不具有这些缺点的替代方法是创建一个库。库导出一组标识符，每个标识符在库中定义或从其他库导入。导出的标识符不需要绑定为变量;它可以被绑定为关键字。

以下库导出两个标识符：变量 gpa->grade 和关键字 gpa。变量 gpa->grade 绑定到一个过程，该过程采用平均绩点 （GPA），表示为数字，并根据四分制返回相应的字母等级。关键字 gpa 命名句法扩展，其子形式必须全部为字母等级，其值是从这些字母等级计算得出的 GPA。

```
（库（等级）
（导出 gpa->级 gpa）
（导入 （rnrs））

（定义范围内？
（lambda （x n y）
（and （>= n x） （< n y））））

（define-syntax range-case
（syntax-rules （- else）
[（_ expr （（x - y） e1 e2 ...） ...[else ee1 ee2
...])
（let （[tmp expr])
（cond
[（in-range？ x tmp y） e1 e2 ...]
...
[else ee1 ee2 ...]））]
[（_ expr （（x - y） e1 e2 ...） ...）
（let （[tmp expr])
（cond
```

```
[ (in-range?  x tmp y)  e1 e2 ...]
...) ) ]) )

 (定义字母>数字
 (lambda  (x)
 (case x
[ (a)  4.0]
[ (b)  3.0]
[ (c)  2.0]
[ (d)  1.0]
[ (f)  0.0]
[else  (断言违规 'grade  "invalid letter grade"
x) ]) )

 (定义 gpa->grade
 (lambda  (x)
 (range-case x
[ (0.0 - 0.5)  'f]
[ (0.5 - 1.5)  'd]
[ (1.5 - 2.5)  'c]
[ (2.5 - 3.5)  'b]
[else 'a]) ) )

 (define-syntax gpa
 (syntax-rules  ()
[ (_ g1 g2 ...)
 (let  ([ls  (map letter->number '(g1 g2 ...) ) ])
 (/  (apply + ls)  (length ls) ) ) )
```

库的名称是（成绩）。这似乎是一个有趣的名称，但所有库名称都带有括号。该库从标准（rnrs）库导入，该库包含我们在本章和最后一章中使用的大部分基元和关键字绑定，以及我们实现gpa->grade和gpa所需的一切。

除了 gpa->grade 和 gpa 之外，库中还定义了其他几个语法扩展和过程，但其他任何一个都不导出。未导出的

那些只是那些被导出的帮助者。库中使用的所有内容都应该是熟悉的，除了应用过程，如第 107 页所述。

如果您的 Scheme 实现支持在交互式顶层导入，则可以测试这两个导出，如下所示。

```
（进口（等级））
（gpa c a c b b） ⇒ 2.8
（gpa->grade 2.8） ⇒ b
```

第10章更详细地描述了库，并提供了其他使用它们的示例。

练习 3.6.1

修改 gpa 以处理“x”级，这些成绩不计入平均绩点。小心谨慎地处理每个等级为x的情况。

```
（进口（等级））
（gpa a x b c） ⇒ 3.0
```

练习 3.6.2

从（grades）导出一个新的语法形式，分布，它采用一组等级，如gpa，但返回形式（（n g）...）的列表，其中n是g在集合中出现的次数，每个g有一个条目。让分发调用未导出的过程来执行实际工作。

```
（进口（等级））
（分布 a b a c c c a f b a） ⇒ （（4 a） （2 b） （3 c） （0 d） （1 f））
```

练习 3.6.3

现在阅读第 7.8 节中的输出操作，并将新的导出直方图定义为采用文本输出端口和分布（例如可能由分布生成的分布）的过程，并以以下示例所示的样式打印直方图。

```
（进口（等级））
（直方图
（电流输出端口）
（分布 a b a c c a c a f b a））

打印：
a:  *****
b:  **
c:  ***
d:
f:  *
```

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

http://www.scheme.com