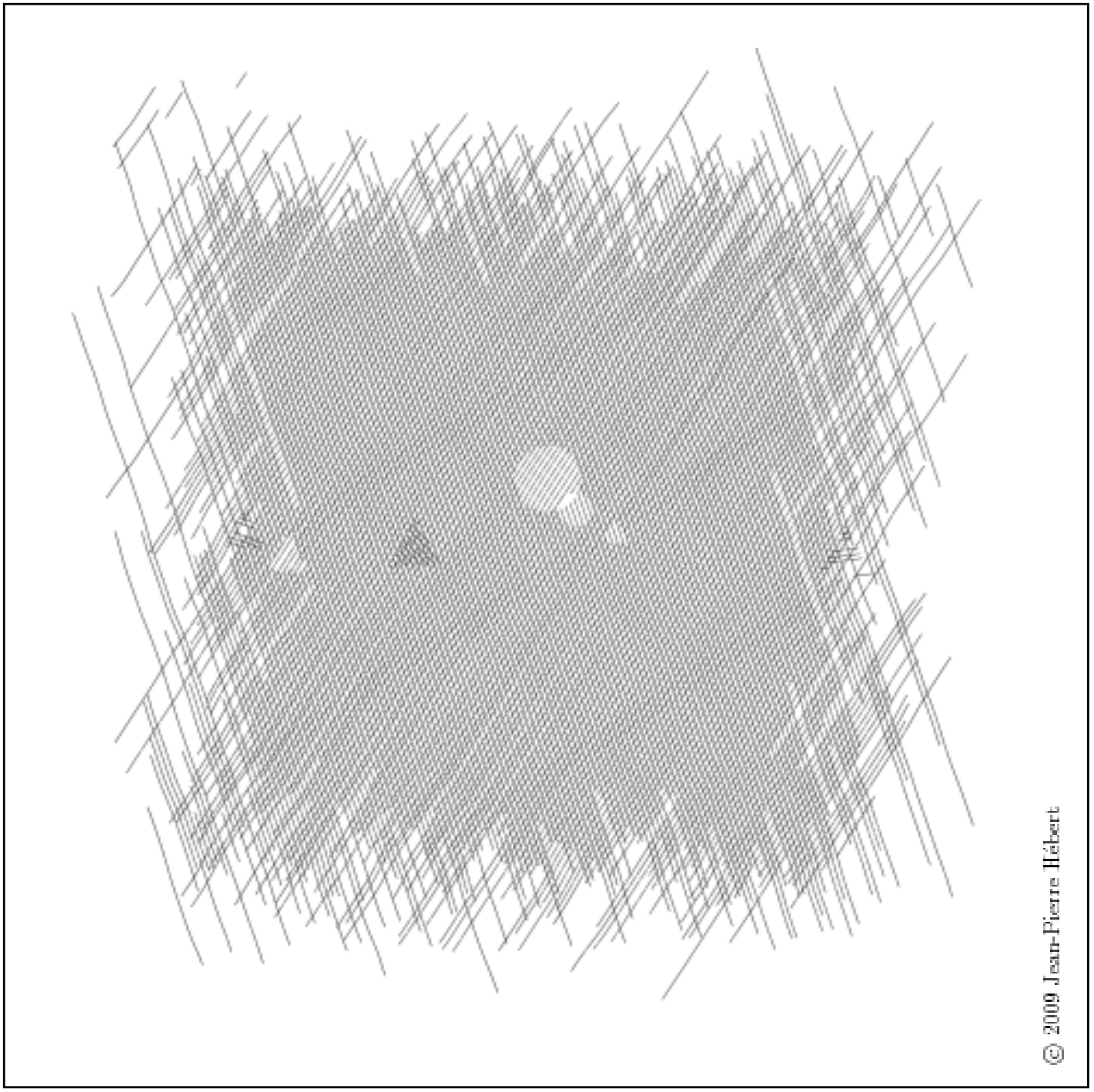

第 5 章。控制操作

本章介绍用作 Scheme 程序控制结构的语法形式和过程，第一部分介绍最基本的控制结构、过程应用，其余部分介绍排序、条件评估、递归、映射、延续、延迟评估、多个值以及在运行时构造的程序的评估。

第 5.1 节.程序申请

语法： （expr0 expr1 ...）
返回：将 expr0 的值应用于 expr1 的值的值 ...

程序应用是最基本的方案控制结构。任何在第一个位置没有语法关键字的结构化表单都是过程应用程序。计算表达式 expr0 和 expr1 ... ;每个都应计算为单个值。计算完这些表达式中的每一个后，expr0 的值将应用于 expr1 ... 的值。如果 expr0 未计算为过程，或者过程不接受提供的参数数，则会引发条件类型 &断言的异常。

过程和参数表达式的计算顺序未指定。它可以是从左到右，从右到左或任何其他顺序。但是，可以保证评估是连续的：无论选择什么顺序，在开始计算下一个表达式之前，每个表达式都会被完全计算。

```
(+ 3 4)⇒ 7

（（如果 （奇数？ 3） + -） 6 2） ⇒ 8

（（lambda （x） x） 5） ⇒ 5

（let （[f （lambda （x） （+ x x））]）
（f 8）） ⇒ 16
```

程序： （应用程序 obj ...列表）
返回：将过程应用于 obj ... 的值和列表
库的元素：（rnrs base）、 （rnrs）

apply 调用过程，将第一个 obj 作为第一个参数传递，第二个 obj 作为第二个参数，依此类推，用于 obj ... 中的每个对象，并按顺序传递 list 的元素作为其余参

数。因此，调用过程时要使用尽可能多的参数，因为有 objs 加上 list 的元素。

当要传递给过程的部分或全部参数都在列表中时，apply 很有用，因为它使程序员不必显式反构列表。

```
(申请 + '(4 5)) ⇒ 9

(apply min '(6 8 3 2 5)) ⇒ 2

(apply min 5 1 3 '(6 8 3 2 5)) ⇒ 1

(apply vector 'a 'b '(c d e)) ⇒ #(a b c d e)

(define first
(lambda (ls)
(apply (lambda (x. y) x) ls)))
(define rest
(lambda (ls)
(apply (lambda (x. y) ls)))
(first '(a b c d)) ⇒ a
(其余 '(a b c d)) ⇒ (b c d)

(应用附加
'(1 2 3)
'((a b) (c d e) (f))) ⇒ (1 2 3 a b c d e f)
```

第 5.2 节. 测 序

语法： (开始 expr1 expr2 ...)
返回：最后一个子表达式
库的值：(rnrs base)， (rnrs)

表达式 expr1 expr2 ... 按从左到右的顺序计算。begin 用于对赋值、输入/输出或其他导致副作用的操作进行排

序。

```
(定义 x 3)
(开始
(设置! x (+ x 1))
(+ x x)) ⇒ 8
```

开始形式可以包含零个或多个定义来代替表达式$_{expr1\ expr2\ ...}$，在这种情况下，它被认为是一个定义，并且可能仅在定义有效的情况下出现。

```
(let ()
(begin (define x 3) (define y 4))
(+ x y)) ⇒ 6
```

这种形式的 begin 主要由必须扩展到多个定义的句法扩展使用。（请参阅第 101 页。

许多句法形式的主体，包括 lambda、case-lambda、let、let*、letrec 和 letrec*，以及 cond、case 和 do 的结果子句，都被视为好像它们位于隐式开头;即，构成正文或结果子句的表达式按顺序执行，并返回最后一个表达式的值。

```
(定义交换对!
(lambda (x)
(let ([temp (car x)])
(set-car! x (cdr x))
(set-cdr! x temp)
x)))
(swap-pair! (缺点 'a 'b)) ⇒ (b .
```

第 5.3 节. 条件

语法：（如果测试结果替代）
语法：（如果测试结果）
返回：结果或替代的值取决于测试
库的值：（rnrs base），（rnrs）

检验、后续和备选子窗体必须是表达式。如果 test 的计算结果为真值（#f 以外的任何值），则计算 result 并返回其值。否则，将计算备选方案并返回其值。对于第二种“单臂”形式，它别无选择，如果测试评估为假，则结果未指定。

```
(let ([ls '(a b c)])
(if (null? ls)
'()
(cdr ls))) ⇒ (b c)

(let ([ls '()])
(if (null? ls)
'()
(cdr ls))) ⇒ ()

(let ([abs
(lambda (x)
(if (< x 0)
(- 0 x)
x))])
(abs -4)) ⇒ 4

(let ([x -4])
(if (< x 0)
(list 'minus (- 0 x))
(list 'plus 4))) ⇒ (减去 4)
```

过程：（不是obj）
返回：如果obj为假，则#t，否则#f
库：（rnrs base），（rnrs）

not 等价于 （lambda （x） （if x #f #t））） 。

```
（不#f）⇒ #t
（不是#t） ⇒ #f
（不是’（）） ⇒ #f
（不是（< 4 5）） ⇒ #f
```

语法： （和 expr ...）
返回：请参阅下面的
库：（rnrs base）， （rnrs）

如果不存在子表达式，则 and 窗体的计算结果为#t。否则，并按从左到右的顺序计算每个子表达式，直到只剩下一个子表达式或子表达式返回#f。如果还剩下一个子表达式，则计算该子表达式并返回其值。如果子表达式返回 #f，并在不计算其余子表达式的情况下返回#f。和 的语法定义出现在第 62 页上。

```
（让 （[x 3]）
（和 （> x 2） （< x 4））） ⇒
#t
（让 （[x 5]）
（和 （> x 2） （< x 4））） ⇒ #f

（和 #f ’（a b） ’（c d）） ⇒ #f
（和 ’（a b） ’（c d） ’（e f）） ⇒
```

语法： （或 expr ...）
返回：请参阅下面的
库：（rnrs base）， （rnrs）

如果不存在子表达式，则 or 窗体的计算结果为#f。否则，或按从左到右的顺序计算每个子表达式，直到只剩下一个子表达式或子表达式返回#f以外的值。如果还剩下一个子表达式，则计算该子表达式并返回其值。如果

子表达式返回#f以外的值，或者返回该值而不计算其余子表达式。或 的语法定义出现在第 63 页上。

```
（让 （[x 3]）
（或 （< x 2） （> x 4））） ⇒
#f
（让 （[x 5]）
（或 （< x 2） （> x 4））） ⇒ #t

（或 #f '（a b） '（c d）） ⇒ （a b）
```

语法： （cond 子句 1 子句 2 ...）
返回：请参阅下面的
库：（rnrs base）， （rnrs）

每个子句（但最后一个子句）必须采用以下形式之一。

```
（测试）
（测试 expr1 expr2 ...）
（测试 => expr）
```

最后一个子句可以是上述任何形式，也可以是该形式的“else子句”

```
（否则 expr1 expr2 ...）
```

每个测试都按顺序进行评估，直到一个评估为真实值或直到所有测试都已评估完毕。如果测试计算结果为真值的第一个子句采用上面给出的第一种形式，则返回 test 的值。

如果检验计算结果为真值的第一个子句采用上面给出的第二种形式，则按顺序计算表达式 expr1 expr2...，并返回最后一个表达式的值。

如果检验计算结果为真值的第一个子句采用上面给出的第三种形式，则计算表达式 expr。该值应是一个参数的过程，该参数应用于 test 的值。将返回此应用程序的值。

如果所有测试的计算结果均不为 true 值，并且存在 else 子句，则按顺序计算 else 子句的表达式 expr1 expr2 ...，并返回最后一个表达式的值。

如果所有测试的计算结果均不为 true 值，并且不存在 else 子句，则表示该值或值未指定。

有关 cond 的语法定义，请参见第 305 页。

```
(let ([x 0])
(cond
[(< x 0) (list 'minus (abs x))]
[(> x 0) (list 'plus x)]
[else (list 'zero x)])) ⇒ (零 0)

(定义选择
(lambda (x)
(cond
[(not (symbol? x)))
[(assq x '((a . 1) (b . 2) (c . 3))) => cdr]
[else 0])))

(选择 3) ⇒ #t
(选择 'b) ⇒ 2
(选择 'e) ⇒ 0
```

语法： else

语法： =>

库： (rnrs base)， (rnrs exceptions)， (rnrs)

这些标识符是 cond 的辅助关键字。两者都用作防护的辅助关键字，否则也用作大小写的辅助关键字。引用这些标识符是语法冲突，除非在上下文中将它们识别为辅助关键字。

语法： （当 test-expr1 expr2 时 ...）
语法： （除非 test-expr1 expr2 ...）
返回： 参见下面的
库： （rnrs control）， （rnrs）

当 test-expr 的计算结果为真值时，将按顺序计算表达式 expr1 expr2 ...，并返回最后一个表达式的值。如果 test-expr 的计算结果为 false，则不计算任何其他表达式，并且未指定 when 的值。

除非，如果 test-expr 的计算结果为 false，则按顺序计算表达式 expr1 expr2 ...，并返回最后一个表达式的值。如果 test-expr 的计算结果为真值，则不会计算任何其他表达式，并且未指定 if 的值。

A时或除非表达通常比相应的“单臂”如果表达更清晰。

```
（let （[x -4] [符号 ’加号]）
（当 （< x 0）
（设置！ x （- 0 x））
（设置！ 符号 ’减号））
（列表符号 x））⇒ （减去 4）

（定义 check-pair
（lambda （x）
（除非 （pair? x）
（语法违反 ’check-pair ’invalid argument“ x））
x））

（check-pair ’（a b c））⇒ （a b c）
```

当可以定义如下：

```
（定义语法时
（语法规则 （）
[（_ e0 e1 e2 ...）
（如果 e0 （开始 e1 e2 ...））））
```

除非可以定义如下：

```
（定义语法，除非
（语法规则 （）
[（_ e0 e1 e2 ...）
（如果（不是 e0）（以 e1 e2 开头...））））
```

或以下情况：

```
（定义语法，除非
（语法规则 （）
[（_ e0 e1 e2 ...）
（当（不是 e0） e1 e2 ...））））
```

语法： （大小写 expr0 子句 1 子句 2 ...）
返回：请参阅下面的
库：（rnrs base）， （rnrs）

每个子句，但最后一个子句必须采用以下形式

```
（（键 ...）expr1 expr2 ...）
```

其中每个键都是一个不同于其他键的基准。最后一个子句可以是上述形式，也可以是该形式的 else 子句

```
（否则 expr1 expr2 ...）
```

计算 expr0，并按顺序将结果（使用 eqv？）与每个子句的键进行比较。如果找到包含匹配键的子句，则按顺序计算表达式 expr1 expr2 ...，并返回最后一个表达式的值。

如果所有子句都不包含匹配键，并且存在 else 子句，则按顺序计算 else 子句的表达式 expr1 expr2 ...，并返回最后一个表达式的值。

如果所有子句都不包含匹配键，并且不存在 else 子句，则表示一个或多个值未指定。

有关大小写的语法定义，请参阅第 306 页。

```
(let ([x 4] [y 5])
(case (+ x y)
[(1 3 5 7 9) 'odd]
[(0 2 4 6 8) 'even]
[else 'out-range])) ⇒ 奇怪
```

第 5.4 节. 递归和迭代

语法： (让名称 ((var expr) ...) body1 body2 ...)
返回：最终正文表达式
库的值：(rnrs base)， (rnrs)

这种形式的 let，称为 let，是一种通用的迭代和递归构造。它类似于更常见的 let 形式（参见第 4.4 节），将变量 var... 绑定到 body1 body2 ... 中 expr ... 的值，该函数像 lambda 主体一样进行处理和计算。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或迭代;过程的参数成为变量 var ... 的新值。

表单的命名 let 表达式

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

可以使用 letrec 重写，如下所示。

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

可以在第 312 页上找到实现此转换并处理未命名 let 的 let 的语法定义。

下面定义的过程除数使用命名 let 来计算非负整数的非平凡除数。

```
（定义除数
（lambda （n）
（let f （[i 2]）
（cond
[（>= i n） '（）]
[（integer? （/ n i））（缺点 i （f （+ i 1））））]
[else （f （+ i 1））））））

（除数 5） ⇒ （）
（除数 32） ⇒ （2 4 8 16）
```

当找到除数时，上面的版本是非尾递归的，当找不到除数时，是尾递归的。下面的版本是完全尾递归的。它以相反的顺序建立列表，但如果需要，这很容易通过在退出时反转列表来补救。

```
（定义除数
（lambda （n）
```

```
(let f ([i 2] [ls '()])
(cond
[(>= i n) ls]
[(integer? (/ n i)) (f (+ i 1) (缺点 i ls))]
[else (f (+ i 1) ls)]))
```

语法：(do ((var init update) ...) (测试结果 ...) expr ...)
返回：最后一个结果表达式
库的值：(rnrs 控件)、(rnrs)

do 允许简洁地表示常见的受限迭代形式。变量 var ... 最初绑定到 init ... 的值，并在每次后续迭代时反弹到更新 ...的值。表达式测试、更新 ...、expr ... 和结果 ... 都在为 var ... 建立的绑定范围内。

在每个步骤中，计算测试表达式测试。如果 test 的值为 true，则迭代停止，按顺序计算表达式结果 ...，并返回最后一个表达式的值。如果不存在结果表达式，则 do 表达式的值未指定。

如果 test 的值为 false，则按顺序计算表达式 expr ...，计算表达式更新 ...，创建 var ... 到 update ... 值的新绑定，然后继续迭代。

表达式 expr... 仅用于效果计算，并且通常完全省略。可以省略任何更新表达式，在这种情况下，效果与更新只是相应的 var 相同。

尽管大多数语言中的循环构造要求通过赋值更新循环迭代，但确实要求循环迭代器通过重新绑定进行更新。事实上，do 表达式的计算不涉及任何副作用，除非它们由其子表达式显式执行。

有关 do 的语法定义，请参阅第 313 页。

下面阶乘和斐波那契的定义是对第 3.2 节中给出的尾递归命名-let 版本的直接翻译。

```
(定义阶乘
(lambda (n)
(do ([i n (- i 1)] [a 1 (* a i)])
((零? i) a))))

(阶乘 10) ⇒ 3628800

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

(斐波那契 6) ⇒ 8
```

下面除数的定义类似于上面命名 let 的描述给出的除数的尾递归定义。

```
(定义除数
(lambda (n)
(do ([i 2 (+ i 1)]
[ls '()
(if (integer? (/ n i))
(cons i ls)
ls)])
((>= i n) ls))))
```

下面的 scale-vector! 定义，它将 vector v 的每个元素缩放一个常量 k，演示了一个非空 do 主体。

```
(定义比例矢量!
(lambda (v k)
(let ([n (vector-length v)])
(do ([i 0 (+ i 1)])
((= i n))
(vector-set! v i (* (vector-ref v i) k)))))))

(define vec (vector 1 2 3 4 5))
(scale-vector! vec 2)
vec ⇒ #(2 4 6 8 10)
```

第 5.5 节. 映射和折叠

当程序必须重复或循环访问列表的元素时，映射或折叠运算符通常更方便。这些运算符通过逐个将过程应用于列表的元素来抽象出 null 检查和显式递归。一些映射运算符也可用于向量和字符串。

过程： (映射过程列表$_1$ 列表$_2$ ...)
返回：结果
库列表：(rnrs base)， (rnrs)

map 将过程应用于列表 list1 list2 ... 的相应元素，并返回结果值的列表。列表 list1 list2... 必须具有相同的长度。过程应接受与列表一样多的参数，应返回单个值，并且不应改变列表参数。

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

(地图 (lambda (x y) (* x y))
'(1 2 3 4)
'(8 7 6 5)) ⇒ (8 14 18 20)
```

虽然未指定应用程序本身出现的顺序，但输出列表中值的顺序与输入列表中相应值的顺序相同。

map 可以定义如下。

```
(define map
(lambda (f ls . more)
(if (null? more)
(let map1 ([ls ls])
(if (null? ls)
'()
(cons (f (car ls))
(map1 (cdr ls)))))
(let map-more ([ls ls] [more more])
(if (null? ls)
'()
(cons
(apply f (car ls) (map car ls) (map car more)))
(map-more (cdr ls) (map cdr more))))
```

此版本的地图不会执行任何错误检查;f 假定为一个过程，其他参数假定为相同长度的正确列表。这个定义的一个有趣的特点是，地图使用自身来拉出输入列表列表的汽车和cdr;这是有效的，因为对单一列表案例的特殊处理。

过程：（对于每个过程列表$_1$列表$_2$ ...）
返回：未指定的
库：（rnrs base），（rnrs）

for-each 与 map 类似，不同之处在于 for-each 不会创建并返回结果值的列表，并且 for-each 保证按顺序在从左到右的元素上执行应用程序。过程应接受与列表一样多的参数，并且不应改变列表参数。可以定义每个，而无需进行错误检查，如下所示。

```
(define for-each
(lambda (f ls . more)
(do ([ls ls (cdr ls)] [more (map cdr more)])
((null? ls))
(apply f (car ls) (map car more)))))

(let ([same-count 0])
(for-each
(lambda (x y)
(when (= x y)
(set! same-count (+ same-count 1))))
'(1 2 3 4 5 6)
'(2 3 3 4 7 6))
same-count) ⇒ 3
```

过程：（存在过程列表 $_1$ 列表$_2$ ...）
返回：请参阅下面的
库：（rnrs 列表）、（rnrs）

列表 list1 list2... 必须具有相同的长度。过程应接受与列表一样多的参数，并且不应改变列表参数。如果列表为空，则存在返回#f。否则，exists 将过程应用于列表 list1 list2 的相应元素，直到列表每个元素只有一个元素或过程返回 true 值 t。在前一种情况下，存在尾部调用过程，并将其应用于每个列表的其余元素。在后一种情况下，存在返回 t。

```
(存在符号?'(1.0 #\a "hi" '()))  ⇒ #f

(存在成员
'(a b c)
'((c b) (b a) (a c))) ⇒ (b a) (b a) (b a)

(exists (lambda (x y z) (= (+ x y) z))
'(1 2 3 4)
```

```
'(1.2 2.3 3.4 4.5)
'(2.3 4.4 6.4 8.6)) ⇒ #t
```

存在可以定义（效率低下，没有错误检查）：如下所示：

```
(定义存在
(lambda (f ls . more)
(and (not (null? ls)))
(let exists ([x (car ls)] [ls (cdr ls)] [more])
(if (null? ls)
(apply f x (map car more))
(or (apply f x (map car more))
(exists (car ls) (cdr ls) (map cdr more)))))
```

过程：（对于所有过程列表$_1$列表$_2$ ...）
返回：请参阅下面的
库：（rnrs 列表）、（rnrs）

列表 list1 list2... 必须具有相同的长度。过程应接受与列表一样多的参数，并且不应改变列表参数。如果列表为空，则 for-all 将返回#t。否则，for-all 将按顺序将过程应用于列表 list1 list2... 的相应元素，直到每个列表只剩下一个元素或过程返回#f。在前一种情况下，for-all 尾部调用过程，将其应用于每个列表的其余元素。在后一种情况下，for-all 返回#f。

```
(对于所有符号?'(a b c d)) ⇒ #t

(for-all =
'(1 2 3 4)
'(1.0 2.0 3.0 4.0)) ⇒ #t

(for-all (lambda (x y z) (= (+ x y) z))
'(1 2 3 4)
'(1.2 2.3 3.4 4.5)
'(2.2 4.3 6.5 8.5)) ⇒ #f
```

可以定义 for-all（效率有点低，没有错误检查）：如下所示：

```
(定义 for-all
(lambda (f ls . more)
(or (null? ls)
(let for-all ([x (car ls)] [ls (cdr ls)] [more more])
(if (null? ls)
(apply f x (map car more))
(and (apply f x (map car more))
(for-all (car ls) (cdr ls) (map cdr more)))
```

过程：（折叠过程 obj list1 list2 ...）
返回：请参阅下面的
库：（rnrs 列表）、（rnrs）

列表参数应具有相同的长度。过程应接受比列表参数数多一个参数，并返回单个值。它不应改变列表参数。

如果列表参数为空，则 fold-left 返回 obj。如果它们不为空，fold-left 将过程应用于 obj 和 list1 list2 ... 的汽车，然后用过程返回的值代替 obj，并用每个列表的 cdr 代替列表。

```
(折叠左缺点'() '(1 2 3 4)) ⇒ (((() . 1) . 2) . 3) . 4)

(fold-left
(lambda (a x) (+ a (* x x)))
0 '(1 2 3 4 5)) ⇒ 55

(fold-left
(lambda (a . args) (append args a))
'(question)
'(that not to)
```

’（is to be）
’（be： or））⇒（to be or not be： that is question）

过程：（折叠右键过程 obj list1 list2 ...）
返回：请参阅下面的
库：（rnrs 列表）、（rnrs）

列表参数应具有相同的长度。过程应接受比列表参数数多一个参数，并返回单个值。它不应改变列表参数。

如果列表参数为空，则 fold-right 将返回 obj。如果它们不为空，则右折叠会用每个列表的 cdr 替换列表，然后将过程应用于 list1 list2 的车辆.....以及递归返回的结果。

（折叠右缺点’（）’（1 2 3 4））⇒（1 2 3 4）

（fold-right
（lambda（x a）（+ a（* x x））））
0’（1 2 3 4 5））⇒ 55

（fold-right
（lambda（x y a）（cons* x y a））⇒（分手是如此甜蜜的悲伤
’（（（带着歉意））
（带着歉意））
’（甜蜜的明天去看））

过程：（矢量映射过程 vector1 vector1 ...）
返回：结果
库的向量：（rnrs base），（rnrs）

vector-map 将过程应用于 vector1 vector2 ... 的相应元素，并返回结果值的向量。vectors vector1 vector2 ... 必

须具有相同的长度，并且过程应接受与向量一样多的参数并返回单个值。

```
（矢量地图 abs '#（1 -2 3 -4 5 -6））⇒ #（1 2 3 4 5 6）
（矢量映射 （lambda （x y） （* x y））
'#（1 2 3 4）
'#（8 7 6 5））⇒ #（8 14 18 20）
```

虽然未指定应用程序本身出现的顺序，但输出向量中值的顺序与输入向量中相应值的顺序相同。

过程： （每个过程的向量 vector1 vector2 ...）
返回：未指定的
库：（rnrs base），（rnrs）

vector-for-each 与 vector-map 类似，不同之处在于 vector-for-each 不会创建和返回结果值的向量，并且 vector-for-each 保证按顺序在从左到右的元素上执行应用程序。

```
（let （[same-count 0]）
（vector-for-each
（lambda （x y）
（when （= x y）
（set! same-count （+ same-count 1））））
'#（1 2 3 4 5 6）
'#（2 3 3 4 7 6））
same-count）⇒ 3 个
```

过程： （每个过程的字符串字符串 $_1$ 字符串$_2$ ...）
返回：未指定的
库：（rnrs base），（rnrs）

string-for-each 与 for-each 和 vector-for-each 类似，只是输入是字符串而不是列表或向量。

```
(let ([ls '()])
(string-for-each
(lambda r (set! ls (cons r ls))) )
"abcd"
"===="
"1234")
(map list->string (reverse ls))) ⇒ ("a=1"
"b=2" "c=3" "d=4")
```

第 5.6 节. 延续

Scheme 中的延续是表示从计算中的给定点开始计算的其余部分的过程。它们可以通过“带电流的继续调用”获得，这可以缩写为 call/cc。

过程：（调用/抄送过程）
过程：（使用当前继续过程调用）
返回：请参阅下面的
库：（rnrs base），（rnrs）

这些过程是相同的。较短的名称通常用于明显的原因，即键入所需的击键较少。

call/cc 获取其延续并将其传递给过程，过程应接受一个参数。延续本身由过程表示。每次将此过程应用于零个或多个值时，都会将值返回到 call/cc 应用程序的延续中。也就是说，当调用延续过程时，它将其参数作为 call/cc 应用程序的值返回。

如果在传递继续过程时过程过程正常返回，则 call/cc 返回的值是过程返回的值。

延续允许实现非局部退出，回溯[14，29]，协程[16]和多任务处理[10，32]。

下面的示例说明了如何使用延续从循环执行非局部退出。

```
(define member
(lambda (x ls)
(call/cc
(lambda (break)
(do ([ls ls (cdr ls)])
((null? ls) #f)
(when (equal? x (car ls))
(break ls)))))))

(成员'd'(a b c)) ⇒ #f
(成员'b'(a b c)) ⇒ (b c)
```

第 3.3 节和第 12.11 节提供了更多示例。

当前延续通常在内部表示为过程激活记录的堆栈，获取延续涉及将堆栈封装在过程对象中。由于封装的堆栈具有无限范围，因此必须使用某种机制来无限期地保留堆栈内容。这可以以惊人的轻松和高效完成，并且对不使用延续的程序没有影响[17]。

过程：（动态风进入正文输出）
返回：由正文
库应用产生的值：（rnrs base），（rnrs）

动风提供“保护”，防止持续调用。它对于执行每当控件进入或离开正文时必须执行的任务非常有用，无论是正常还是通过继续应用程序。

三个参数输入，正文和输出必须是程序，并且应该接受零参数，即它们应该是thunks。在应用 body 之前，并且每次随后通过应用在 body 中创建的延续来输入 body 时，都会应用 in thunk。在正常退出身体时，每次通过应用在外部身体之外创建的延续来退出身体时，都会应用退出。

因此，可以保证至少调用一次 in。此外，如果 body 返回，则至少调用一次 out。

下面的示例演示如何使用动态风来确保输入端口在处理后关闭，而不管处理是否正常完成。

```
(let ([p (open-input-file “input-file”)])
(dynamic-wind
(lambda () #f)
(lambda () (process p))
(lambda () (close-port p))))
```

Common Lisp 提供了类似的功能（展开保护），用于防止非本地退出。这通常就足够了。但是，展开保护仅提供等效于 out，因为 Common Lisp 不支持完全通用的延续。以下是使用动态风指定放松保护的方法。

```
(定义语法展开保护
(语法规则 ()
[(_ 正文清理 ...)
(dynamic-wind
(lambda () #f)
(lambda () body)
(lambda () cleanup ...)))))

((call/cc
(let ([x 'a])
(lambda (k)
(unwind-protect
```

```
(k (lambda () x))
(set! x 'b))))) ⇒ b
```

一些 Scheme 实现支持一种称为流体绑定的受控赋值形式，其中变量在给定计算期间采用临时值，并在计算完成后恢复为旧值。下面根据动态风定义的语法形式fluid-let允许单个变量x的流体绑定到主体b1 b2 ...内的表达式e的值。

```
(define-syntax fluid-let
(syntax-rules )
[(_ ((x e)) b1 b2 ...)
(let ([y e])
(let ([swap (lambda () (let ([t x]) (set! x
y) (set! y t)))])
(dynamic-wind swap (lambda () b1 b2 ...)
swap)))
```

支持 fluid-let 的实现通常会扩展它以允许无限数量的 (x e) 对，就像 let 一样。

如果在 fluid-let 的主体内未调用任何延续，则其行为与在输入时为变量分配新值并在返回时为变量赋值的行为相同。

```
(让 ([x 3])
(+ (流体让 ([x 5])
x)
x)) ⇒ 8
```

如果调用了在流体 let 外部创建的延续，则流体绑定变量也会恢复为旧值。

```
(let ([x 'a])
(let ([f (lambda () x)])
(cons (call/cc
```

```
(lambda (k)
(fluid-let ([x 'b])
(k (f)))))
(f))))) ⇒ (b .
```

如果控制离开了流体放体（正常或通过调用延续），并且控制通过调用延续重新进入主体，则恢复流体绑定变量的临时值。此外，对临时值的任何更改都将保留，并在重新进入时反映出来。

```
(定义重新输入#f)
(定义 x 0)
(fluid-let ([x 1])
(call/cc (lambda (k) (set! reenter k))))
(set! x (+ x 1)))
x) ⇒ 2
x ⇒ 0
(reenter '*) ⇒ 3
(reenter '*) ⇒ 4
x ⇒ 0
```

下面给出了一个库，显示了如果尚未内置动态风，如何实现动态风。除了定义动态风之外，该代码还定义了一个版本的 call/cc，该版本尽其所能支持动态风。

```
(library (dynamic-wind)
(export dynamic-wind call/cc
(rename (call/cc call-with-current-continuation)))
(import (rename (rnrs) dynamic-wind) (call/cc
rnrs: call/cc)))

(define winders '())

(define common-tail
(lambda (x y)
(let ([lx (length x)] [ly (length y)])
(do ([x (if (> lx ly) (list-tail x (- lx ly))
```

```
 x)  (cdr x) ]
 [y  (if  (> ly lx)  (list-tail y  (- ly lx) )  y)
  (cdr y) ])
  ( (eq?  x y)  x) ) ) ) )

  (定义 do-wind
  (lambda  (new)
  (let  ([tail  (common-tail new winders) ])
  (let f  ([ls winders])
  (if  (not  (eq?  ls tail) )
  (begin
  (set!  winders  (cdr ls) )
  ( (cdar ls) )
  (f  (cdr ls) ) ) ) )
  (let f  ([ls new])
  (if  (not  (eq?  ls tail) )
  (begin
  (f  (cdr ls) )
  ( (caar ls) )
  (设置!  卷绕机 ls) ) ) ) ) ) ) ) ) )

  (define call/cc
  (lambda  (f)
  (rnrs: call/cc
  (lambda  (k)
  (f  (let  ([save winders])
  (lambda  (x)
  (除非  (eq?  save winders)  (do-wind save) )
  (k x) ) ) )

  (定义动态风
  (lambda  (在体输出)
  (在)
  (设置!  卷绕器  (缺点  (缺点在输出)  卷绕器) )
  (let-值  ([ans*  (身体) ])
  (设置!  卷绕器  (cdr 卷绕器) )
  (输出)
  (应用值和*) ) ) ) )
```

动态风和呼叫/抄送一起管理卷绕机列表。卷绕机是通过调用动态风而建立的一对进出的砰砰声。每当调用动态风时，都会调用 in thunk，将包含进出 thunk 的新卷绕器放在卷绕器列表中，调用主体 thunk，从卷绕器列表中删除卷绕器，并调用 out thunk。这种排序可确保只有当控制权已通过且尚未进入时，卷绕机才在卷绕机列表中。每当获得延续时，都会保存卷绕器列表，每当调用延续时，都会恢复保存的卷绕器列表。在恢复期间，将调用当前卷绕机列表中未在已保存的卷绕器列表中的每个卷绕机的出声，然后调用已保存的卷绕器列表中每个卷绕机的砰砰声，该卷绕器列表中也不在当前卷绕器列表中。卷绕机列表以增量方式更新，以确保只有当控制已通过其在 thunk 而不是输入其 out thunk 时，卷绕机才在当前卷绕器列表中。

在 call/cc 中执行的测试（不是（eq? 保存卷绕机））并非绝对必要，但每当保存的卷绕机列表与当前卷绕机列表相同时，调用延续的成本就会降低。

第 5.7 节. 延迟评估

语法形式延迟和过程力可以结合使用来实现延迟求值。在需要延迟求值之前，不会计算受延迟求值的表达式，并且一旦计算，则永远不会重新计算。

语法：　（延迟 expr）
返回：　一个承诺
过程：　（强制承诺）
返回：　强制承诺
库的结果：　（rnrs r5rs）

第一次通过延迟产生的承诺被强制（用力）时，它会评估 expr，“记住”结果值。此后，每次强制承诺时，它都会返回记住的值，而不是重新评估 expr。

延迟和力通常仅在没有副作用的情况下使用，例如，分配，因此评估的顺序不重要。

使用延迟和强制的好处是，如果延迟到绝对需要，则可以完全避免一定量的计算。延迟评估可用于构建概念上无限的列表或流。下面的示例显示了如何在延迟和强制下构建流抽象。流是一个承诺，当强制时，返回其cdr是流的一对。

```
(定义流汽车
(lambda （s）
(car （force s））））

(定义 stream-cdr
(lambda （s）
(cdr （force s））））

(定义计数器
(let next （[n 1])
(delay （cons n （next （+ n 1））））））

(stream-car counters) ⇒ 1

(stream-car （stream-cdr counters)) ⇒ 2

(define stream-add
(lambda （s1 s2）
(delay （cons ））
(+ （流车 s1） （流车 s2））
(stream-add （stream-cdr s1） （stream-cdr s2））））

(定义偶数计数器
```

```
（stream-add 计数器计数器））

（stream-car even-counters） ⇒ 2

（stream-car （stream-cdr even-counters）） ⇒ 4
```

延迟可由下式定义

```
（define-syntax delay
（syntax-rules （）
[（_ expr） （make-promise （lambda （） expr））））
```

其中，“许诺”的定义如下。

```
（定义 make-promise
（lambda （p）
（let （[val #f] [set? #f]）
（lambda （）
（除非 set?
（让 （[x （p）]）
（除非设置?
（设置! val x）
（set! set? #t））））
val））））
```

根据延迟的这个定义，force只是调用承诺来强制评估或检索保存的值。

```
（定义力
（lambda （promise）
（promise））)
```

在 make-promise 中对变量集进行第二次检验是必要的，因为由于应用 p，则 promise 是递归强制的。由于承诺必须始终返回相同的值，因此将返回第一次应用 p 完成的结果。

延迟和强制处理多个返回值是否未指定;上面给出的实现没有，但下面的版本确实如此，在值调用和应用的帮助下。

```
(定义 make-promise
(lambda (p)
(let ([vals #f] [set? #f])
(lambda ()
(除非 set?
(使用值 p
调用 (lambda x
(除非设置?
(设置! vals x)
(set! set? #t) ) ) ) )
(apply values vals) ) ) ) )

(定义 p (delay (value 1 2 3) ) )
(force p) ⇒ 1
  2
  3
(call-with-values (lambda () (force p) ) +) ⇒ 6
```

这两个实现都不是完全正确的，因为如果 force 的参数不是承诺，则必须使用条件类型和断言引发异常。由于不可能将“作出承诺”所创造的程序与其他程序区分开来，因此武力不能可靠地做到这一点。以下对“许诺”和“强制”的重新实现将“许诺”表示为“允诺”类型的记录，以允许强制进行所需的检查。

```
(定义记录类型承诺
(字段(不可变 p)(可变 vals)(可变集?
(protocol (lambda (new) (lambda (p) (new p #f
#f) ) ) )

(define force
(lambda (promise)
(除非 (promise? promise)
```

```
 (assertion-violation 'promise 'invalid argument'
promise) )
 (除非 (promise-set? promise)
 (call-with-values (promise-p promise)
 (lambda x
 (除非 (promise-set? promise)
 (promise-vals-set! promise x)
 (promise-set? -set! promise #t) ) ) )
 (应用值 (承诺-价值承诺) ) ) )
```

第 5.8 节. 多个值

虽然所有 Scheme 基元和大多数用户定义过程只返回一个值，但某些编程问题最好通过返回零值、多个值甚至可变数量的值来解决。例如，将值列表划分为两个子列表的过程需要返回两个值。虽然多个值的创建者可以将它们打包到数据结构中，并且使用者可以提取它们，但使用内置的多值接口通常更干净。此接口由两个过程组成：值和值调用。前者生成多个值，后者将生成多值值的过程与使用这些值的过程链接在一起。

过程： (值 obj ...)
returns: obj ...
libraries: (rnrs base), (rnrs)

过程值接受任意数量的参数，并简单地将参数传递（返回）为其延续。

```
(值) ⇒

(值 1) ⇒ 1

(值 1 2 3) ⇒ 1
  2
  3
```

```
(定义头和尾
(lambda (ls)
(值 (car ls) (cdr ls))))

(头尾 '(a b c)) ⇒ a
  (b c)
```

过程：（带值调用的生产者使用者）
返回：请参阅下面的
库：(rnrs base)，(rnrs)

生产者和消费者必须是程序。对值的调用将使用者应用于通过调用不带参数的生产者返回的值。

```
(call-with-values
(lambda () (values 'bond 'james))
(lambda (x y) (cons y x))) ⇒ (james . bond)

(call-with-values values list) ⇒ '()
```

在第二个示例中，值本身充当生产者。它不接收任何参数，因此不返回任何值。因此，list 不应用于任何参数，因此返回空列表。

下面定义的过程 dxdy 计算坐标由 (x .y) 对。

```
(定义 dxdy
(lambda (p1 p2)
(值 (- (car p2) (car p1))
(- (cdr p2) (cdr p1)))))

(dxdy '(0 . 0) '(0 . 5)) ⇒ 0
 5
```

dxdy 可用于计算由两个端点表示的线段的长度和斜率。

```
(定义 segment-length
(lambda (p1 p2)
(call-with-values
(lambda () (dxdy p1 p2))
(lambda (dx dy) (sqrt (+ (* dx dx) (* dy
dy)))))

(定义段斜率
(lambda (p1 p2)
(call-with-values
(lambda () (dxdy p1 p2))
(lambda (dx dy) (/ dy dx)))))

(segment-length '(1 . 4) '(4 . 8)) ⇒ 5
(segment-slope '(1 . 4) '(4 . 8)) ⇒ 4/3
```

当然，我们可以将它们组合成一个返回两个值的过程。

```
(define describe-segment
(lambda (p1 p2)
(call-with-values
(lambda () (dxdy p1 p2))
(lambda (dx dy)
(values
(sqrt (+ (* dx dx) (* dy dy)))
(/ dy dx))))))

(描述段 '(1 . 4) '(4 . 8)) ⇒ 5
⇒ 4/3
```

下面的示例使用多个值以非破坏性方式将列表划分为交替元素的两个子列表。

```
(定义拆分
(lambda (ls)
(if (or (null? ls) (null? (cdr ls)))
(values ls '())
(call-with-values
```

```
(lambda () (split (cddr ls)))
(lambda (odds evens)
(values (cons (car ls) odds)
(cons (cadr ls) evens)))))

(split '(a b c d e f)) ⇒ (a c e)
  (b d f)
```

在每个递归级别，过程拆分返回两个值：参数列表中奇数编号元素的列表和偶数元素的列表。

对值的调用的延续不必是通过对值调用建立的调用来建立的，也不必仅使用值来返回到由值调用建立的延续。特别是，(值 e) 和 e 是等效表达式。例如：

```
(+ (值 2) 4) ⇒ 6

(if (值 #t) 1 2) ⇒ 1

(调用值
(lambda () 4)
(lambda (x) x)) ⇒ 4
```

同样，值可用于将任意数量的值传递给忽略这些值的延续，如下所示。

```
(开始(值 1 2 3) 4) ⇒ 4 个
```

由于延续可以接受零个或多个值，因此通过 call/cc 获得的延续可以接受零个或多个参数。

```
(call-with-values
(lambda ()
(call/cc (lambda (k) (k 2 3))))
(lambda (x y) (list x y))) ⇒ (2 3)
```

当期望正好一个值的延续接收零个值或多个值时，该行为未指定。例如，以下每个表达式的行为都是未指定的。某些实现会引发异常，而其他实现会静默地禁止显示其他值或为缺失值提供默认值。

```
（如果（值 1 2） ’x ’y）

（+ （值） 5）
```

希望在特定上下文中强制忽略额外值的程序可以通过显式调用“与值的调用”来轻松执行此操作。可以定义一种语法形式，我们可以先调用它，以便在只需要一个值时抽象出丢弃多个值的过程。

```
（define-syntax first
（syntax-rules （）
[（_ expr）
（call-with-values
（lambda （） expr）
（lambda （x . y） x））））

（如果（第一个（值#t #f））’a’b） ⇒ a
```

由于如果过程不接受传递给它的参数数，则实现需要使用条件类型 &断言引发异常，因此以下每个过程都会引发异常。

```
（带值
的调用 （lambda （） （值 2 3 4））
（lambda （x y） x））

（call-with-values
（lambda （） （call/cc （lambda （k） （k 0））））
（lambda （x y） x））
```

由于生产者通常是 lambda 表达式，因此为了便于阅读，使用语法扩展来抑制 lambda 表达式通常很方便。

```
(define-syntax with-values
(syntax-rules ()
[(_ expr consumer)
(call-with-values (lambda () expr) consumer))))

(带值 (值 1 2) 列表) ⇒ (1 2)
(带值 (拆分 '(1 2 3 4)))
(lambda (赔率偶数)
偶数)) ⇒ (2 4)
```

如果使用者也是 lambda 表达式，则第 4.5 节中描述的 let 和 let* 的多值变体通常更方便。

```
(let-values ([(odds evens) (split '(1 2 3 4))])
evens) ⇒ (2 4)

(let-values ([ls (values 'a 'b 'c)])
ls) ⇒ (a b c)
```

许多标准的句法形式和过程传递多个值。其中大多数是“自动的”，从某种意义上说，实现人员必须做任何特殊的事情来实现这一点。let 通常扩展到直接 lambda 调用中，会自动传播由 let 主体生成的多个值。其他运算符必须经过特殊编码才能传递多个值。例如，“使用端口调用”过程（第 7.6 页）调用其过程参数，然后在返回过程的值之前关闭 port 参数，因此它必须临时保存这些值。这可以通过 let 值、apply 和值轻松实现：

```
(定义使用端口
的调用 (lambda (port proc)
(let-values ([val* (proc port)])
(close-port port)
(apply values val*)))))
```

如果在返回单个值时这似乎开销太大，则代码可以使用带值调用和 case-lambda 来更有效地处理单值情况：

```
(define call-with-port
(lambda (port proc)
(call-with-values (lambda () (proc port))
(case-lambda
[(val) (close-port) val]
[val* (close-port port) (apply values val*)))))
```

下面的库中值和用值调用的定义（以及随之而来的 call/cc 的重新定义）表明，如果多返回值接口尚未内置，则可以在 Scheme 中实现该接口。但是，对于将多个值返回到单值上下文的情况，例如 if 表达式的测试部分，无法执行错误检查。

```
(库 (mrvs)
(导出带值的调用值 call/cc
(重命名 (call/cc call-with-current-continuation)))
(导入
(重命名 (除了
(rnrs) 值 call-with-values)
(call/cc rnrs: call/cc)))

(定义魔术 (缺点 '多个 '值))

(定义魔术?
(lambda (x)
(and (pair? x) (eq? (汽车 x) 魔术)))))

(define call/cc
(lambda (p)
(rnrs: call/cc
(lambda (k)
(p (lambda args
(k (apply values args)))))
```

```
(定义值
(lambda args
(if (and (not (null? args)) (null? (cdr
args)))
(car args)
(cons magic args)))))

(定义与值
的调用 (lambda (producer consumer)
(let ([x (producer)])
(if (magic? x)
(apply consumer (cdr x))
(consumer x)))))))
```

可以更有效地实现多个值 [2]，但此代码用于说明运算符的含义，并可用于在不支持运算符的旧的非标准实现中提供多个值。

第 5.9 节. 埃瓦尔

Scheme的评估过程允许程序员编写构建和评估其他程序的程序。这种进行运行时元编程的能力不应该被过度使用，但在需要时很方便。

过程：(eval obj 环境)
返回：在环境
库中由 obj 表示的 Scheme 表达式的值：(rnrs eval)

如果 obj 不表示语法上有效的表达式，则 eval 会引发条件类型和语法的异常。环境、方案报告环境和空环境返回的环境是不可变的。因此，如果表达式中出现对环境中任何变量的赋值，eval 还会引发条件类型和语法的异常。

```
(定义缺点'不是缺点)
(eval '(let ([x 3]) (cons x 4)) (environment
```

```
’（rnrs））） ⇒ （3 . 4）

（define lambda ’not-lambda）
（eval ’（lambda （x） x） （environment ’（rnrs））） ⇒ #<procedure>

（eval ’（cons 3 4） （environment）） ⇒ exception
```

过程： （环境导入规范 ...）
返回： 环境
库： （rnrs eval）

环境返回由给定导入说明符的组合绑定形成的环境。每个导入规范都必须是表示有效导入说明符的 s 表达式（请参见第 10 章）。

```
（定义 env （environment ’（rnrs） ’（前缀 （rnrs lists） $））））
（eval ’（$cons* 3 4 （* 5 8）） env） ⇒ （3 4 . 40）
```

过程： （空环境版本）
过程： （方案-报告-环境版本）
返回：R5RS 兼容性环境
库： （rnrs r5rs）

版本必须为确切的整数 5。

null-environment 返回一个环境，其中包含关键字的绑定，其含义由修订的5 方案报告定义，以及辅助关键字 else、=>、...和 _ 的绑定。

scheme-report-environment 返回一个环境，该环境包含与 null-environment 返回的环境相同的关键字绑定，以及变量的绑定，这些变量的含义由 Revision5 Report on

Scheme 定义，但 Revision6 报告未定义的变量的绑定除外：load、interaction-environment、transcript-on、transcript-off 和 char-ready? 。

这些过程返回的环境中每个标识符的绑定都是相应的 Revision6 Report 库的绑定，因此，即使未使用例外标识符绑定，这也不会提供完全的向后兼容性。

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

http://www.scheme.com