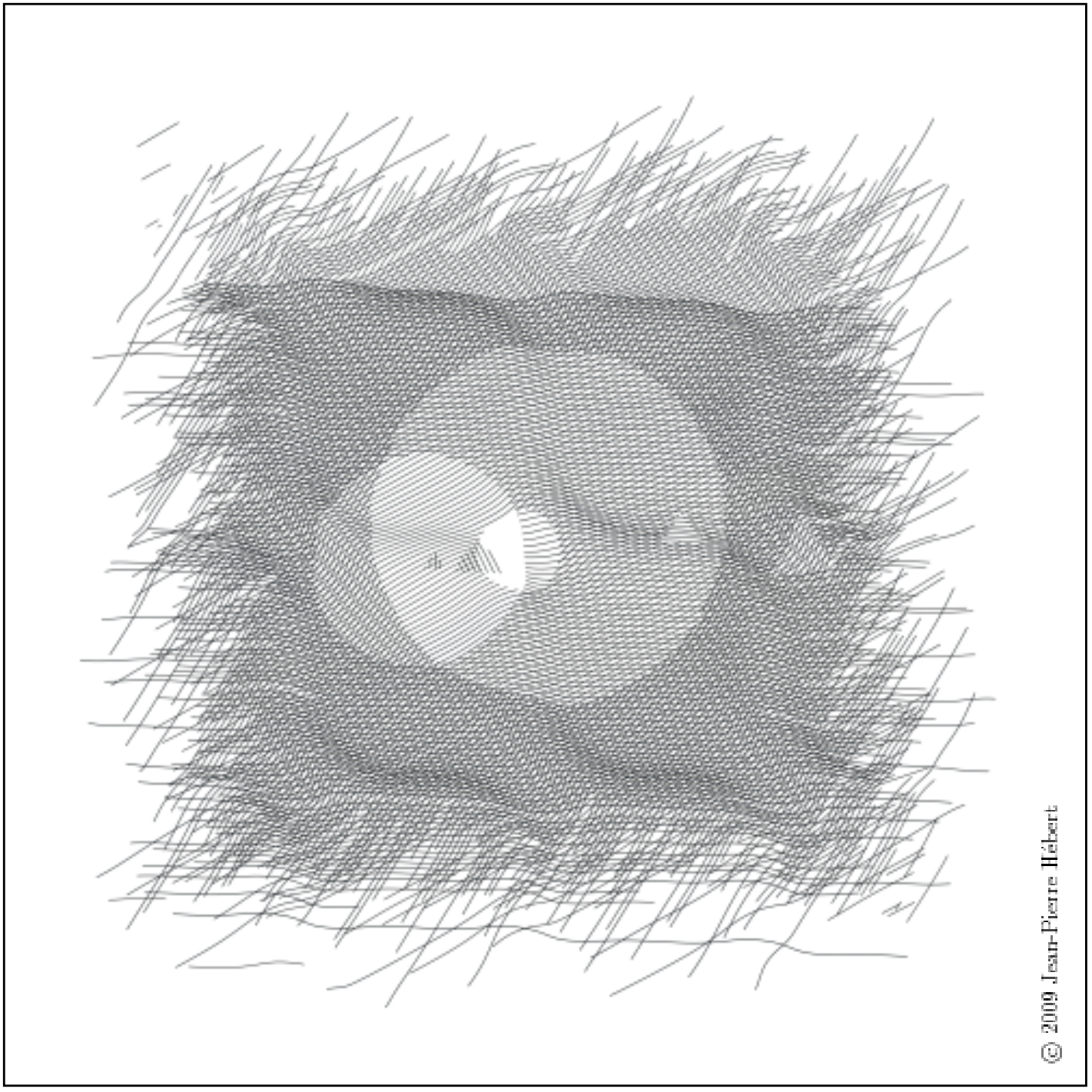

第 4 章。过程和变量绑定

过程和变量绑定是 Scheme 程序的基本构建块。本章介绍一小组语法形式，其主要目的是创建过程和操作变量绑定。它从 Scheme 程序的两个最基本的构建块开始：变量引用和 lambda 表达式，并继续描述变量绑定和赋值形式，如 define、letrec、let-values 和 set! 。

绑定或赋值变量的各种其他形式（这些变量的绑定或赋值不是主要目的）在第 5 章中可以找到（例如命名 let）。

第 4.1 节. 变量引用

语法： 变量
返回： 变量的值

如果存在标识符的可见变量绑定，则该标识符出现在由 define、lambda、let 或某些其他变量绑定构造创建的绑定的范围内，则在程序中显示为表达式的任何标识符都是变量。

```
列表 ⇒ #<过程>
(定义 x 'a)
(列表 x x) ⇒ (a)
(让 ([x 'b])
(列表 x x)) ⇒ (b b)
(让 (让 ([让'让]) 让) 让) ⇒ 让
```

如果标识符引用未绑定为变量、关键字、记录名称或其他实体，则在库窗体或顶级程序中显示标识符引用是语法冲突。由于库、顶级程序、lambda 或其他局部主体中定义的范围是整个主体，因此变量的定义不必在其第一个引用出现之前出现，只要在定义完成之前未实际计算引用。因此，例如，在下面f的定义中对g的引用

```
(定义 f
(lambda (x)
(g x)))
(定义 g
(lambda (x)
(+ x x)))
```

是可以的，但是下面q的定义中对g的引用不是。

```
（定义 q （g 3））
（定义 g
（lambda （x）
（+ x x））））
```

第 4.2 节.Lambda

语法： （lambda formals body1 body2 ...）
返回： 过程
库： （rnrs base）， （rnrs）

lambda 语法形式用于创建过程。任何创建过程或建立局部变量绑定的操作最终都是根据 lambda 或 case-lambda 定义的。

形式中的变量是过程的形式参数，子窗体 body1 body2 的序列 ... 是其主体。

正文可以从一系列定义开始，在这种情况下，由定义创建的绑定是正文的本地绑定。如果存在定义，则在展开正文时使用并丢弃关键字绑定，并将正文展开为由变量定义和其余表达式组成的 letrec* 表达式，如第 292 页所述。lambda 的此描述的其余部分假定，如有必要，此转换已发生，因此主体是没有定义的表达式序列。

创建过程时，主体内自由出现的所有变量（不包括形式参数）的绑定都将保留在过程中。随后，每当将过程应用于一系列实际参数时，形式参数将绑定到实际参数，恢复保留的绑定，并计算主体。

应用时，由形式定义的形式参数绑定到实际参数，如下所示。

- 如果形式是变量的正确列表，例如（x y z），则每个变量都绑定到相应的实际参数。如果提供的实际参数太少或太多，则会引发条件类型 &断言的异常。

- 如果形式是单个变量（不在列表中），例如 z，则绑定到实际参数的列表。

- 如果形式是由变量终止的变量的不正确列表，例如（x y . z），则每个变量但最后一个变量都绑定到相应的实际参数。最后一个变量绑定到其余实际参数的列表。如果提供的实际参数太少，则会引发条件类型 &断言的异常。

计算正文时，将按顺序计算正文中的表达式，并且该过程返回最后一个表达式的值。

过程没有通常意义上的印刷表示。方案系统以不同的方式打印程序;本书使用符号#<procedure>。

```
(lambda (x) (+ x 3)) ⇒ #<procedure>
((lambda (x) (+ x 3)) 7) ⇒ 10
((lambda (x y) (* x (+ x y))) 7 13) ⇒ 140
((lambda (f x x)) + 11) ⇒ 22
((lambda () (+ 3 4))) ⇒ 7

((lambda (x . y) (list x y))
28 37) ⇒ (28 (37))
((lambda (x . y) (list x y))
28 37 47 28) ⇒ (28 (37 47 28))
((lambda (x y. z) (list x y z))
1 2 3 4) ⇒ (1 2 (3 4))
((lambda x) 7 13) ⇒ (7 13)
```

第 4.3 节. 案例-Lambda

Scheme lambda 表达式始终生成一个具有固定参数数或具有大于或等于特定数量的无限数量的参数的过程。特别

（lambda （$var1$ ... $varn$）身体$_1$ 身体$_2$ ...）

正好接受 n 个参数，

（lambda r body1 body2 ...）

接受零个或多个参数，并且

（lambda （$var1$ ... 瓦尔$_{恩}$ .r） body1 body2 ...）

接受 n 个或多个参数。

但是，lambda 不能直接生成一个接受两个或三个参数的过程。特别是，lambda 不直接支持接受可选参数的过程。上面显示的后一种形式的lambda可以与长度检查以及汽车和cdr的组合结合使用，以实现具有可选参数的过程，尽管以清晰度和效率为代价。

case-lambda 语法形式直接支持具有可选参数的过程以及具有固定或不确定参数数的过程。case-lambda 基于文章“具有可变 Arity 的过程的新方法”[11] 中介绍的 lambda* 句法形式。

语法： （大小写-lambda 子句 ...）
返回： 过程
库： （rnrs 控件）， （rnrs）

case-lambda 表达式由一组子句组成，每个子句类似于 lambda 表达式。每个子句都有以下形式。

[正式的正文$_1$正文$_2$ ...]

子句的形式参数由形式定义，其方式与 lambda 表达式相同。case-lambda 表达式的过程值接受的参数数由各个子句接受的参数数确定。

当调用使用大小写 lambda 创建的过程时，将按顺序考虑子句。选择接受给定数量的实际参数的第一个子句，将其形式定义的形式参数绑定到相应的实际参数，并按照上面对 lambda 的描述计算主体。如果子句中的形式是标识符的正确列表，则子句接受的实际参数与形式中的形式参数（标识符）的数量完全相同。与 lambda 形式一样，大小写-lambda 子句形式可以是单个标识符，在这种情况下，子句接受任意数量的参数，或者由标识符终止的标识符的不正确列表，在这种情况下，子句接受大于或等于不包括终止标识符的正式参数数的任意数量的参数。如果没有子句接受提供的实际参数数，则会引发条件类型 &断言的异常。

以下 make-list 定义使用 case-lambda 来支持可选的填充参数。

```
（定义 make-list
（case-lambda
[（n） （make-list n #f）]
[（n x）
（do （[n n （- n 1）] [ls ’（） （cons x ls）]）
（（zero? n） ls））））
```

可以使用 case-lambda 扩展子字符串过程，以接受无结束索引（在这种情况下，它默认为字符串的末尾）或没有

开始和结束索引，在这种情况下，子字符串等效于字符串复制：

```
（定义子字符串1
（case-lambda
[（s） （substring1 s 0 （string-length s））]
[（s start） （substring1 s start （string-length
s））]
[（s start end） （substring s start end）]））
```

当仅提供一个索引时，也可以默认使用起始索引而不是结束索引：

```
（定义子字符串 2
（大小写-lambda
[（s） （子字符串 2 s 0 （字符串长度 s））]
[（s end） （substring2 s 0 end）]
[（s start end） （substring s start end）]））
```

甚至可以要求同时提供两个或两个都不提供开始索引和结束索引，只需省略中间子句：

```
（定义子字符串3
（大小写 - lambda
[（s） （子字符串3 s 0（字符串长度s））]
[（s 开始结束） （子字符串 s 开始结束）]））
```

第 4.4 节.本地绑定

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

let 建立局部变量绑定。每个变量 var 都绑定到相应表达式 expr 的值。绑定变量的 let 的主体是子窗体 body1

body2 的序列，并且像 lambda 主体一样进行处理和计算。

let、let*、letrec 和 letrec* （其他形式在 let 之后描述）是相似的，但用途略有不同。与 let*、letrec 和 letrec* 相比，表达式 expr ... 都超出了变量 var ... 的范围。此外，与 let* 和 letrec* 相比，表达式 expr ... 的计算没有隐含排序。它们可以从左到右，从右到左或由实现自行决定的任何其他顺序进行评估。每当值与变量无关且计算顺序不重要时，请使用 let。

```
(let ([x (* 3.0 3.0)] [y (* 4.0 4.0)])
(sqrt (+ x y))) ⇒ 5.0

(let ([x 'a] [y '(b c)])
(cons x y)) ⇒ (a b c)

(let ([x 0] [y 1])
(let ([x y] [y x])
(list x y))) ⇒ (1 0)
```

let 的以下定义显示了 let 从 lambda 的典型派生。

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

另一种形式的 let，名为 let，在第 5.4 节中描述，完整 let 的定义可以在第 312 页找到。

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

let* 与 let 类似，只是表达式 expr ... 按从左到右的顺序计算，并且这些表达式中的每一个都在左侧变量的范围内。当值之间存在线性依赖关系或计算顺序很重要时，请使用 let*。

```
(let* ([x (* 5.0 5.0)]
[y (- x (* 4.0 4.0))])
(sqrt y)) ⇒ 3.0

(let ([x 0] [y 1])
(let* ([x y] [y x])
(list x y))) ⇒ (1 1)
```

任何 let* 表达式都可以转换为一组嵌套的 let 表达式。let* 的以下定义演示了典型的转换。

```
(define-syntax let*
(syntax-rules ()
[(_ () e1 e2 ...)
(让 () e1 e2 ...)]
[(_ ((x1 v1) (x2 v2) ...) e1 e2 ...)
(let ((x1 v1))
(let* ((x2 v2) ...) e1 e2 ...)))))
```

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

letrec 类似于 let 和 let*，只不过所有表达式 expr ... 都在所有变量 var ... 的范围内。letrec 允许定义相互递归过程。

```
(letrec ([sum (lambda (x)
(if (zero? x)
0
```

```
（+ x （sum （- x 1））））
（sum 5））⇒ 14
```

表达式 expr ... 的计算顺序未指定，因此在计算完所有值之前，程序不得计算对 letrec 表达式绑定的任何变量的引用。（在 lambda 表达式中出现的变量不计为引用，除非在计算完所有值之前应用了生成的过程。如果违反此限制，则会引发条件类型为 &断言的异常。

一个 expr 不应该返回超过一次。也就是说，它不应正常返回，也不应通过调用在其评估期间获得的延续返回，也不应通过两次调用此类延续而返回两次。不需要实现来检测违反此限制的行为，但如果这样做，则会引发条件类型和断言的异常。

当变量及其值之间存在循环依赖关系并且计算顺序不重要时，请选择 letrec 而不是 let 或 let*。当存在循环依赖关系并且需要从左到右计算绑定时，选择 letrec* 而不是 letrec。

表单的 letrec 表达式

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

可以用 let 和 set 来表示！

```
（让 （（var #f） ...）
（让 （（温度外显） ...）
（设置！var temp） ...
（让 （） body1 body2 ...））））
```

其中 temp ... 是新的变量，即尚未出现在 letrec 表达式中的变量，每个 （var expr） 对一个。外部 let 表达式建立变量绑定。给定每个变量的初始值并不重要，因

此任何值都足以代替#f。首先建立绑定，以便 expr ... 可以包含变量的出现次数，即，以便在变量的范围内计算表达式。中间的 let 计算这些值并将它们绑定到临时变量，set! 表达式将每个变量分配给相应的值。如果主体包含内部定义，则存在内部允许。

使用此转换的 letrec 的定义如第 310 页所示。

此转换不会强制实施 expr 表达式不得计算变量的任何引用或赋值的限制。可以进行更复杂的转换，以强制执行此限制并实际生成更有效的代码[31]。

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

letrec* 与 letrec 类似，只是 letrec* 按从左到右的顺序计算 expr ...。虽然在评估相应的expr之前，程序仍然不能评估对任何var的引用，但此后的任何时间都可以评估对var的引用，包括在评估任何后续绑定的expr期间。

表单的 letrec* 表达式

(letrec* ((var expr) ...) 身体$_1$ 身体$_2$ ...)

可以用 let 和 set 来表示！

```
(让 ((var #f) ...)
(设置！var expr) ...
(让 () body1 body2 ...)))
```

外部 let 表达式创建绑定，每个赋值计算一个表达式，并立即按顺序将相应的变量设置为其值，内部 let 计算主体。let在后一种情况下使用，而不是开始，因为主体可能包括内部定义以及表达式。

```
(letrec* ([sum (lambda (x)
(if (zero? x)
0
(+ x (sum (- x 1)))))]
[f (lambda () (cons n n-sum))]
[n 15]
[n-sum (sum n)])
(f)) ⇒ (15 . 120)

(letrec* ([f (lambda () () lambda () g))]
[g (f)])
(eq? (g) g)) ⇒ #t

(letrec* ([g (f)]
[f (lambda () (lambda () g))])
(eq? (g) g)) ⇒ 例外：试图引用未定义的变量f
```

第 4.5 节. 多个值

语法： (let-values ((formals expr) ...) body1 body2 ...)
语法： (let*-values ((formals expr) ...) body1 body2 ...)
返回： 最终 body 表达式
库的值： (rnrs base)， (rnrs)

let-values 是接收多个值并将其绑定到变量的便捷方法。它的结构类似于 let，但允许在每个左侧都有一个任意的形式列表（如 lambda）。let*-值类似，但按从左到

右的顺序执行绑定，与 let* 一样。如果 expr 返回的值数不适合相应的形式，则会引发条件类型 &断言的异常，如上面的 lambda 条目中所述。let-values的定义在第310页给出。

```
(let-values ([(a b) (values 1 2)] [c (values 1 2 3)])
(list a b c)) ⇒ (1 2 (1 2 3))

(let*值 ([(a b) (值 1 2)] [(a b) (值 b a)])
(列表 a b)) ⇒ (2 1)
```

第 4.6 节.变量定义

语法: (定义 var expr)
语法: (定义 var)
语法: (定义 (var_0 var_1 ...) $body_1$ $body_2$ ...)
语法: (定义 (var_0 . var_{rest}) $body_1$ $body_2$ ...)
语法: (定义 (var_0 var_1 var_2 ... . var_{rest}) $body_1$ $body_2$ ...)
库: (rnrs base), (rnrs)

在第一种形式中，define 创建 var 到 expr 值的新绑定。expr 不应返回超过一次。也就是说，它不应正常返回，也不应通过调用在其评估期间获得的延续返回，也不应通过两次调用此类延续而返回两次。不需要实现来检测违反此限制的行为，但如果这样做，则会引发条件类型和断言的异常。

第二种形式等效于(定义未指定的 var)，其中未指定的是某个未指定的值。其余的是将变量绑定到过程的简写形式;就 lambda 而言，它们与以下定义相同。

```
(define var
(lambda formals
body1 body2 ...))
```

其中 formals 是 (var1 ...)、 varr， or (var1 var2 ... . varr)对于第三、第四和第五个定义格式。

定义可以出现在库主体的前面，顶级程序主体形式的任何地方，以及lambda或case-lambda主体的前面，或者从lambda派生的任何形式的主体，例如let或letrec*。任何以定义序列开头的主体都会在宏扩展期间转换为 letrec* 表达式，如第 292 页所述。

语法定义可以与变量定义一起出现在变量定义中;见第8章。

```
(定义 x 3)
x ⇒ 3

(define f
(lambda (x y)
(* (+ x y) 2)))
(f 5 4) ⇒ 18

(define (平方和 x y)
(+ (* x x) (* y y)))
(平方和 3 4) ⇒ 25

(define f
(lambda (x)
(+ x 1)))
(let ([x 2])
(define f
(lambda (y)
(+ y x)))
```

```
(f 3)) ⇒ 5
(f 3) ⇒ 4
```

可以通过以开始形式将一组定义包含起来对它们进行分组。以这种方式分组的定义可能出现在普通变量和语法定义可能出现的位置。它们被视为单独书写，即没有封闭的开始形式。此功能允许语法扩展扩展到定义组。

```
(define-syntax multi-define-syntax
(syntax-rules )
[(_ (var expr) ...)
(begin
(define-syntax var expr)
...)])
(let ()
(define plus
(lambda (x y)
(if (zero? x)
y
(plus (sub1 x) (add1 y)))))
(multi-define-syntax
(add1 (syntax-rules () [(_ e) (+ e 1)]))
(sub1 (syntax-rules () [(_ e) (- e 1)]))
(plus 7 8)) ⇒ 15
```

许多实现支持交互式“顶层”，其中变量和其他定义可以交互方式输入或从文件加载。这些顶级定义的行为超出了修订[6] 报告的范围，但只要在评估对顶级变量的任何引用或赋值之前定义顶级变量，行为在大多数实现中都是一致的。因此，例如，如果尚未定义 g，则在下面 f 的顶级定义中对 g 的引用是可以的，并且假定 g 在以后的某个时间点命名要定义的变量。

```
(定义 f
(lambda (x)
(g x)))
```

如果在计算 f 之前，紧随其后的是 g 的定义，则 g 将被定义为变量的假设被证明是正确的，并且对 f 的调用按预期工作。

```
（定义 g
（lambda （x）
（+ x x））)
（f 3） ⇒ 6
```

如果将 g 定义为句法扩展的关键字，则 g 将绑定为变量的假设被证明是错误的，如果在调用之前未重新定义 f，则实现可能会引发异常。

第 4.7 节. 分配

语法： （设置！var expr）
返回： 未指定的
库： （rnrs base）， （rnrs）

设置！不会为 var 建立新绑定，而是更改现有绑定的值。它首先评估 expr，然后将 var 赋值到 expr 的值。在更改的绑定范围内对 var 的任何后续引用都将计算为新值。

在 Scheme 中，分配的使用频率不如大多数其他语言高，但它们对于实现状态更改很有用。

```
（定义触发器
（let （[state #f]）
（lambda （）
（set！ state （not state））
state））)

（人字拖） ⇒ #t
```

```
（人字拖） ⇒ #f
（人字拖） ⇒ #t
```

赋值对于缓存值也很有用。下面的示例使用一种称为记忆的技术，其中过程记录与旧输入值关联的值，因此无需重新计算它们，以实现斐波那契函数的指数双递归定义的快速版本（见第69页）。

```
(define memoize
(lambda (proc)
(let ([cache '()])
(lambda (x)
(cond
[(assq x cache) => cdr]
[else
(let ([ans (proc x)])
(set! cache (cons (cons (cons x ans) cache))
ans)))))

(定义斐波那契
(memoize
(lambda (n)
(if (< n 2)
1
(+ (fibonacci (- n 1)) (fibonacci (- n
2)))))))

(fibonacci 100) ⇒ 573147844013817084101
```

ISBN 978-0-262-51298-5 / LOC QA76.73.S34D93

订购本书 / 关于这本书

http://www.scheme.com