部分练习的答案

练习 2.2.1. （第20页）

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) 或 (* 1 -2 3 -4 5 -6 7)

练习 2.2.2. （第20页）

请参见第 6.4 节。

练习 2.2.3. （第20页）

一. （汽车 。

二. （这（很傻））

（二） （这很傻吗？

d. (+ 2 3)

e. (+ 2 3)

六. +

克。 (2 3)

h. #<程序>

一. 缺点

j. ’缺点

k. 报价

l. 5

嗯。 5

n. 5

o. 5

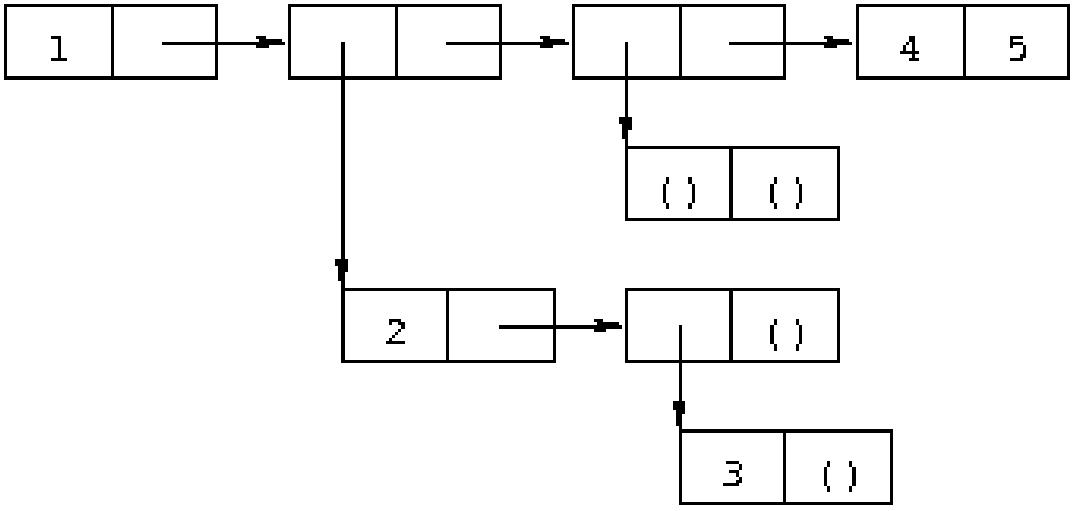
练习 2.2.4. （第21页）

（汽车（车“（（a b）（c d））））⇒ b
（car （car （car （cdr ’（（a b） （c d））））
⇒ c
（car （cdr （car （cdr ’（（a b） （c d））））
⇒ d

练习 2.2.5. （第21页）

’（（a .b） （（c） d） （））

练习 2.2.6. （第21页）



练习 2.2.7. （第21页）

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

练习 2.2.8. (第21页)
参见第2.3节。

练习 2.3.1. (第23页)

1. 计算变量列表 +、-、* 和 /，生成列表、加法、减法、乘法和除法过程。
2. 将列表过程应用于加法、减法、乘法和除法过程，生成按顺序包含这些过程的列表。
3. 计算变量 cdr，生成 cdr 过程。
4. 将 cdr 过程应用于步骤 2 中生成的列表，生成包含减法、乘法和除法过程的列表。
5. 评估可变汽车，生成汽车程序。

6. 将汽车程序应用于步骤4中生成的列表，产生减法过程。
7. 计算常量 17 和 5，得到 17 和 5。
8. 将减法过程应用于 17 和 5，得到 12。

其他订单是可能的。例如，变量汽车可以在其参数之前进行评估。

练习 2.4.1. （第25页）

一. (让我们 ([x (* 3 a)]) (+ (- x b) (+ x b)))

二. (let ([x (list a b c)]) (cons (car x) (cdr x)))

练习 2.4.2. （第25页）
值为 54。外部 let 将 x 绑定到 9，而内部 let 将 x 绑定到 3 (9/3)。内部 let 的计算结果为 6 (3 + 3)，外部 let 的计算结果为 54 (9 × 6)。

练习 2.4.3. （第26页）

一.

```
(let ([x0 'a] [y0 'b])
(list (let ([x1 'c]) (cons x1 y0))
(let ([y1 'd]) (cons x0 y1)))))
```

二.

```
(let ([x0 '((a b) c)])
```

```
(cons (let ([x1 (cdr x0)])
(car x1))
(let ([x2 (car x0)])
(cons (let ([x3 (cdr x2)])
(car x3))
(cons (let ([x4 (car x2)])
x4)
(cdr x2)))))
```

练习 2.5.1. （第30页）

一. 一个

二. （一）

（二）一个

d. ()

练习 2.5.2. （第30页）

见第31页。

练习 2.5.3. （第30页）

一. 无自由变量

二. +

（二）f

d. 缺点、f 和 y

e. 缺点和 y

六. cons、 y 和 z（y 也显示为绑定变量）

练习 2.6.1. （第34页）

该程序将无限期循环。

练习 2.6.2. （第34页）)

```
(define compose
(lambda (p1 p2)
(lambda (x)
(p1 (p2 x)))))

(define cadr (compose car cdr))
(define cddr (compose cdr cdr))
```

练习 2.6.3. （第34页）)

```
(定义caar（组成汽车））
(define cadr (compose car cdr))

(define cdar (compose cdr car))
(define cddr (compose cdr cdr))

(define caaar (compose car caar))
(define caadr (compose car cadr))
(define cadar (compose car cdar))
(define caddr (compose car cddr))

(define cdaar (compose cdr caar))
(define cdadr (compose cdr cadr))
(define cddar (compose cdr cdar))  (define
cddar (compose cdr cdar))
(define cdddr (compose cdr cddr))

(define caaaar (compose caar caar))
```

```
(define caaadr (compose caar cadr))
(define caadar (compose caar cdar))
(define caaddr (compose caar cddr))
(define cadaar (compose cadr caar))
(define cadadr (compose cadr cadr))
(define caddar (compose cadr cdar))
(define cadddr (compose cadr cddr))  (define
cadddr (compose cadr cddr))

(定义cdaaar（组成cdar caar））
(define cdaadr (compose cdar cadr))
(define cdadar (compose cdar cdar))
(define cdaddr (compose cdar cddr))
(define cddaar (compose cddr caar))
(define cddadr (compose cddr cadr))
(define cdddar (compose cddr cdar))
(define cddddr (compose cddr cddr))
```

练习 2.7.1. （第41页）

```
(定义原子?
(lambda (x)
(not (pair? x))))
```

练习 2.7.2. （第41页）

```
(定义较短的
(lambda (ls1 ls2)
(if (< (length ls2) (length ls1))
ls2
ls1))
```

练习 2.8.1. （第46页）
输出的结构将是输入结构的镜像。例如，（.b）将变为（b. a）和（（a .b）。（c . d））将成为（（d .c） 。（b . a））

练习 2.8.2. （第46页）)

```
(define append
(lambda (ls1 ls2)
(if (null? ls1)
ls2
(cons (car ls1) (append (cdr ls1)
ls2)))))
```

练习 2.8.3. （第46页）)

```
(定义 make-list
(lambda (n x)
(if (= n 0)
'()
(cons x (make-list (- n 1) x)))))
```

练习 2.8.4. （第47页）
请参阅第 160 页上的列表引用说明和第 160 页上的列表尾部说明。

练习 2.8.5. （第47页）)

```
(定义更短?
(lambda (ls1 ls2)
(and (not (null? ls2))
(or (null? ls1)
```

```
(shorter? (cdr ls1) (cdr ls2)))))

(define shorter
(lambda (ls1 ls2)
(if (shorter? ls2 ls1)
ls2
ls1)))
```

练习 2.8.6. （第47页）

```
(定义甚至?
(lambda (x)
(or (= x 0)
(奇数? (- x 1)))))
(定义奇数?
(lambda (x)
(and (not (= x 0))
(甚至? (- x 1)))))
```

练习 2.8.7. （第47页）

```
(定义转置
(lambda (ls)
(cons (map car ls) (map cdr ls))))
```

练习 2.9.1. （第54页）

```
(定义 make-counter
(lambda (init incr)
(let ([next init])
(lambda ()
(let ([v next])
```

```
 (set! next (+ next incr) )
v) ) ) ) )
```

练习 2.9.2. (第55页)

```
 (define make-stack
 (lambda ()
 (let ([ls '() ])
 (lambda (msg . args)
 (case msg
[ (empty? mt? ) (空? ls) ]
[ (推! (set! ls (cons (car args) ls) ) ]
[ (顶部) (汽车 ls) ]
[ (砰! (set! ls (cdr ls) ) ]
[else “oops”]) ) ) ) ) )
```

练习 2.9.3. (第55页)

```
 (define make-stack
 (lambda ()
 (let ([ls '() ])
 (lambda (msg . args)
 (case msg
[ (empty? mt? ) (空? ls) ]
[ (推! (set! ls (cons (car args) ls) ) ]
[ (顶部) (汽车 ls) ]
[ (砰! (set! ls (cdr ls) ) ]
[ (ref) (list-ref ls (car args) ) ]
[ (set! ) (设置车 ! (list-tail ls (car args) )
 (cadr args) ) ]
[else “oops”]) ) ) ) ) )
```

练习 2.9.4. (第55页)

```
(定义 make-stack
(lambda (n)
(let ([v (make-vector n)] [i -1])
(lambda (msg . args)
(case msg
[(empty? mt?) (= i -1)]
[(推!
(set! i (+ i 1))
(vector-set! v i (car args)))]
[(top) (vector-ref v i)]
[(pop!) (设置! i (- i 1))]
[(ref) (vector-ref v (- i (car args)))]
[(set!) (vector-set! v (- i (car args))
(cadr args))]
[else “oops”])))))
```

练习 2.9.5. （第56页）

```
(定义 emptyq?
(lambda (q)
(eq? (汽车q) (cdr q))))

(define getq
(lambda (q)
(if (emptyq? q)
(assertion-violation 'getq “the queue is
empty”)
(car (car q)))))

(define delq!
(lambda (q)
(if (emptyq? q)
(assertion-violation 'delq! “队列是空的”)
(set-car! q (cdr (car q))))))
```

练习 2.9.6. （第56页）

```
(定义 make-queue
(lambda ()
(cons '() '() ) ) )

(定义 putq!
(lambda (q v)
(let ([p (cons v '() ) ])
(if (null? (汽车 q) )
(begin
(set-car! q p)
(set-cdr! q p) )
(begin
(set-cdr! (cdr q) p)
(set-cdr! q p) ) ) ) ) )

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

(define delq!
(lambda (q)
(if (eq? (汽车q) (cdr q) )
(begin
(set-car! q '() )
(set-cdr! q '() ) )
(set-car! q (cdr (car q) ) ) ) ) ) )
```

练习 2.9.7. （第56页）

当要求打印循环结构时，某些实现会打印反映其循环结构的输出表示形式。其他实现不检测循环，并且不产生输出或无限输出流。当长度传递循

环列表时，将引发异常，并可能带有一条消息，指示该列表不正确。然而，第42页的长度定义将无限期地循环。

练习 2.9.8. （第56页）

```
(定义种族
( lambda （野兔）
(如果（配对？野兔）
(让（ [ 野兔（ cdr hare ） ] ）
(如果（对？野兔）
(和（不是（ eq ？ hare tortoise ））
(种族（ cdr hare ）（ cdr tor tortoise ））））
( null ？ hare ））））
( null ？ hare ））））
```

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

练习 3.1.1. （第64页）

```
(let （[x （memv 'a ls）]） （和 x （memv 'b x））））→
（（lambda （x） （and x （memv 'b x））） （memv 'a ls））→
（（lambda （x） （if x （and （memv 'b x））#f））） （memv 'a ls））→
（（lambda （x） （if x （memv 'b x） #f））
(memv 'a ls））
```

练习 3.1.2. （第64页）

```
（或 （memv x ’（a b c））  （列表 x））→
（let （（t （memv x ’（a b c））））  （if t t
（or （list x））））  →
（（lambda （t）  （if t t （or （list x））））
（memv x ’（a b c））） →
（（lambda （t）  （if t t （list x）））  （memv x
’（a b c））））
```

练习 3.1.3. （第64页）
见第97页。

练习 3.1.4. （第64页）

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

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

练习 3.2.1. （第72页）
尾递归：偶数？和奇数？，种族，阶乘的第二定义中的事实，斐波那契第二版中的斐波那契。非尾递归：斐波那契第一版中的和、阶乘、斐波那契。 两者：因素。

练习 3.2.2. （第72页）

```
(define factor
(lambda (n)
(letrec ([f (lambda (n i)
(cond
[(>= i n) (list n)]
[(integer? (/ n i))
(cons i (f (/ n i) i))]
[else (f n (+ i 1))]))
(f n 2))))))
```

练习 3.2.3. （第72页）

是的，但是我们需要两个命名的 let 表达式，一个用于偶数？，一个用于奇数？。

```
(让甚至？ ([x 20])
(或 (= x 0)
(让奇数？ ([x (- x 1)])
(和 (不是 (= x 0))
(甚至？ (- x 1)))))
```

练习 3.2.4. （第72页）

```
(定义 fibcount1 0)
(定义斐波那契1
(lambda (n)
(let fib ([i n])
(set! fibcount1 (+ fibcount1 1))
(cond
[(= i 0) 0]
[(= i 1) 1]
[else (+ (fib (-i 1)) (fib (- i
2)))))))))
```

```
（定义斐
波那契2
 (lambda (n)
 (if (= n 0)
0
 (let fib ([i n] [a1 1] [a2 0])
 (set! fibcount2 (+ fibcount2 1) )
 (if (= i 1)
a1
 (fib (- i 1) (+ a1 a2) a1) ) ) ) )
```

（斐波那契 10） 的计数为 177 和 10，（斐波那契 20）的计数为 21891 和 20，（斐波那契 30）的计数为 2692537 和 30。虽然第二个发出的调用数与输入成正比，但随着输入值的增加，第一个调用的调用数会迅速增长（实际上是呈指数级增长）。

练习 3.2.5. （第73页）
见第312页。

练习 3.2.6. （第73页）
在尾部位置或表达式的最后一个子表达式中的调用将不是具有修改了 or 定义的尾部调用。对于偶数？/奇数？例如，生成的 even？ 定义将不再是尾部递归的，并且对于非常大的输入可能会耗尽可用空间。

此定义执行的扩展在另一种方式上是不正确的，这与多个返回值有关（第 5.8 节）：如果最后一个子表达式返回多个值，则 or 表达式应返回多个

值，但对于不正确的定义，每个子表达式都出现在 let 的右侧，这需要一个返回值。更简单和不正确的定义和具有相同的问题。

练习 3.2.7. （第73页）

以下因子的三个版本中的第一个版本通过停止在 √n，避免冗余除法并跳过 2 之后的偶数因子来直接解决已识别的问题。停止√n可能产生最大的节省，其次是跳过大于2的偶数因子。避免冗余除法不太重要，因为它仅在找到因子时发生。

```
(定义因子
(lambda (n)
(let f ([n n] [i 2] [step 1])
(if (> i (sqrt n) )
(list n)
(let ([n/i (/ n i) ])
(if (integer? n/i)
(cons i (f n/i step) )
(f n (+ i step) 2) )
```

第二个版本将 (> i (sqrt n)) 替换为 (> (* i) n)，因为 * 通常比 sqrt 快得多。

```
(定义因子
(lambda (n)
(let f ([n n] [i 2] [step 1])
(if (> (* i) n)
(list n)
(let ([n/i (/ n i) ])
(if (integer? n/i)
```

```
(cons i (f n/i i step))
(f n (+ i step) 2)))))
```

第三个版本使用 gcd（参见第 179 页）来避免大多数划分，因为 gcd 应该比 / 快。

```
(定义因子
(lambda (n)
(let f ([n n] [i 2] [step 1])
(if (> (* i i) n)
(list n)
(if (= (gcd n i) 1)
(f n (+ i step) 2)
(cons i (f (/ n i) i step)))
```

要查看这些更改所产生的差异，请在 Scheme 系统中对因子的每个版本（包括原始版本）进行计时，以查看哪个版本的性能更好。尝试各种输入，包括较大的输入，如 (+(expt 2 100) 1)。

练习 3.3.1. （第77页）

```
(let ([k.n (call/cc (lambda (k) (cons k 0)))])
(let ([k (car k.n)] [n (cdr k.n)])
(write n)
(newline)
(k (cons k (+ n 1)))))
```

或具有多个值（请参见第 5.8 节）：

```
(call-with-values
(lambda () (call/cc (lambda (k) (values k 0))))
(lambda (k n)
(write n)
(newline)
(k k (+ n 1))))
```

练习 3.3.2. （第77页）

```
(定义产品
(lambda (ls)
(if (null? ls)
1
(if (= (car ls) 0)
0
(let ([n (product (cdr ls))])
(if (= n 0) 0 (* n (car ls)))))))
```

练习 3.3.3. （第77页）

如果其中一个进程返回而不调用 pause，则它将返回到首先导致其运行的暂停调用，或者返回到原始调用以启动（如果它是列表中的第一个进程）。下面是允许进程显式退出的系统重新实现。如果其他进程处于活动状态，lwp 系统将继续运行。否则，控件将返回到要启动的原始调用的延续。

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

```
(lambda ()
(call/cc
(lambda (k)
(set! quit-k)
(next) ) ) )
(define next
(lambda () (let
([p (car lwp-list) ])
(set! lwp-list (cdr lwp-list) )
(p) ) )
(define pause
(lambda () (call
/cc
(lambda (k)
(lwp (lambda () (k #f) ) )
(next) ) ) )
(define quit
(lambda (v)
(if (null? lwp-list)
(quit-k v)
(next) ) ) )
```

练习 3.3.4. （第77页）

```
(定义 lwp-queue (make-queue) )
(define lwp
(lambda (thunk)
(putq! lwp-queue thunk) ) )
(define start
(lambda ()
(let ([p (getq lwp-queue) ])
(delq! lwp-queue)
(p) ) ) )
(define pause
```

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

练习 3.4.1. （第80页）

```
 (定义倒数
 (lambda  (n success failure)
 (if  (= n 0)
 (failure)
 (success  (/ 1 n) ) ) ) )
```

练习 3.4.2. （第80页）

```
 (定义重试#f)

 (定义阶乘
 (lambda  (x)
 (let f  ([x x] [k  (lambda  (x)  x) ])
 (if  (= x 0)
 (begin  (set!  retry k)  (k 1) )
 (f  (- x 1)  (lambda  (y)  (k  (* x
y) ) ) ) ) ) )
```

练习 3.4.3. （第80页）

```
 (define map/k
 (lambda  (p ls k)
 (if  (null?  ls)
 (k ' () )
 (p  (car ls)
```

```
 (lambda (x)
 (map/k p (cdr ls)
 (lambda (ls)
 (k (cons x ls) ) )

 (定义倒数
 (lambda (ls)
 (map/k (lambda (x k) (if (= x 0) “zero
found” (k (/ 1 x) ) ) )
ls
 (lambda (x) x) ) ) )
```

练习 3.5.1. (第85页)

```
 (define-syntax complain
 (syntax-rules ()
[ (_ ek msg expr) (ek (list msg expr) ) ]) )
```

练习 3.5.2. (第85页)

```
 (define calc
 (lambda (expr)
 (call/cc
 (lambda (ek)
 (define do-calc
 (lambda (expr)
 (cond
[ (number? expr) expr]
[ (and (list? expr) (= (length expr) 3) )
 (let ([op (car expr) ] [args (cdr expr) ])
 (case op
[ (add) (apply-op + args) ]
[ (sub) (apply-op - args) ]
[ (mul) (apply-op * args) ]
```

```
[ (div)  (apply-op / args) ]
[否则 (抱怨 “无效的运算符” op) ) ) ) ]
[else (complain “invalid expression”
expr) ]) ) )
 (define apply-op
 (lambda (op args)
 (op (do-calc (car args) ) (do-calc (cadr
args) ) ) ) )
 (define complain
 (lambda (msg expr)
 (ek (list msg expr) ) )
 (do-calc expr) ) ) ) )
```

练习 3.5.3. (第85页)

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

```
(define complain
(lambda (msg expr)
(assertion-violation 'calc msg expr) ) )
(set! calc
(lambda (expr)
(do-calc expr) ) )
```

练习 3.5.4. （第85页）

这将添加 sqrt、times（mul 的别名），并加上减号。

```
(let ()
(define do-calc
(lambda (ek expr)
(cond
[(number? expr) expr]
[(and (list? expr) (= (length expr) 2) )
(let ([op (car expr)] [args (cdr expr)])
(case op
[(minus) (apply-op1 ek - args)]
[(sqrt) (apply-op1 ek sqrt args)]
[else (complain ek “invalid unary operator” op)]) ) )
[(和 (list? expr) (= (length expr) 3) )
(let ([op (car expr)] [args (cdr expr)])
(case op
[(add) (apply-op2 ek + args)]
[(sub) (apply-op2 ek - args)]
[(mul times) (apply-op2 ek * args)]
[(div) (apply-op2 ek / args)]
[(expt) (apply-op2 ek expt args)]
[else (complain ek “invalid binary operator” op)]) ) )
[else (complain ek “invalid expression”
```

```
expr) ]) ) ) )
 (定义 apply-op1
 (lambda  (ek op args)
 (op  (do-calc ek  (car args) ) ) ) )
 (define apply-op2
 (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)
 (call/cc
 (lambda  (ek)
 (do-calc ek expr) ) ) ) ) )
```

练习 3.6.1. （第87页）
当所有输入字母等级均为 x 时，此版本的 gpa 返回 x。

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

练习 3.6.2. （第87页）
在库中定义$distribution和分发后，如下所示：

```
(定义$distribution
(lambda (ls)
(let loop ([ls ls] [a 0] [b 0] [c 0] [f 0])
(if (null? ls)
(list (list a 'a) (list b 'b) (list c 'c)
(list d 'd) (list f 'f))
(case (car ls)
[(a) (loop (cdr ls) (+ a 1) b c d f)]
[(b) (loop (cdr ls) a (+ b 1) c d f)]
[(c) (loop (cdr ls) a b (+ c 1) d f)]
[(d) (loop (cdr ls) ab c (+ d 1) f)]
[(f) (loop (cdr ls) a b c d (+ f 1))]
;忽略 x 个等级，根据前面的练习
[(x) (循环 (cdr ls) a b c d f)]
[否则 (断言违规 '分布
"无法识别的等级字母"
(car ls))))))

($distribution '(g1 g2 ...)))))
```

修改导出行以添加分发（但不$distribution）。

练习 3.6.3. （第87页）
定义直方图后，如下所示：

```
(定义直方图
(lambda (port distr)
(for-each
(lambda (n g)
(put-datum port g)
(put-string port ": "))
(let loop ([n n])
```

```
(除非 (= n 0)
(put-char port #\*)
(loop (- n 1) ) ) )
(put-string port "\n" ) )
(map car distr)
(map cadr distr) ) ) )
```

修改导出行以添加直方图。解决方案使用 -每个，如第 118 页所述

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

http://www.scheme.com