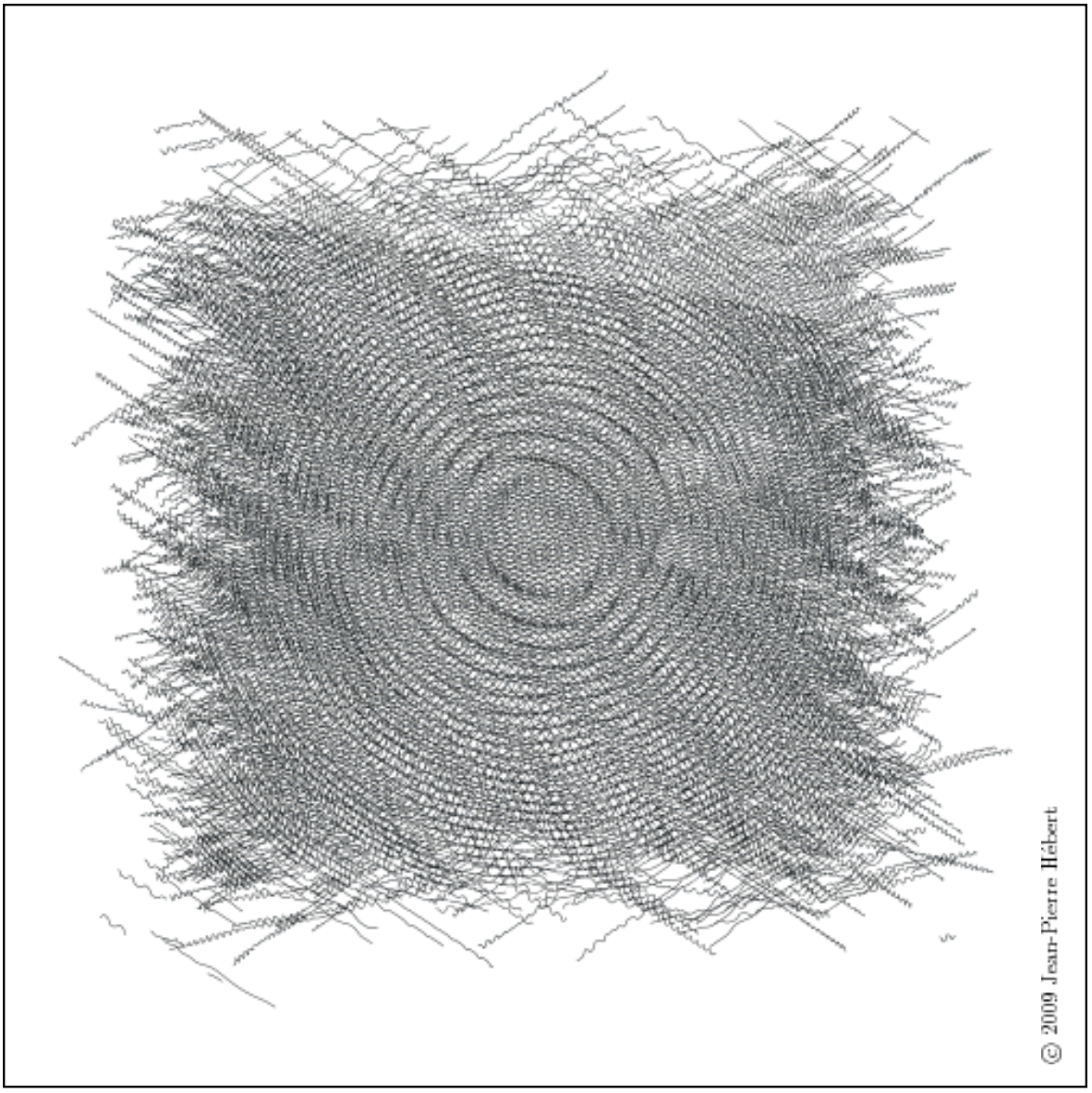

第 6 章。对对象的操作

本章介绍对对象的操作，包括列表、数字、字符、字符串、向量、字节向量、符号、布尔值、哈希表和枚举。第一部分涵盖常量对象和报价。第二部分介绍用于比较两个对象的泛等价谓词和用于确定对象类型的谓词。后面的部分将介绍主要处理上述对象类型之一的过程。没有处理过程操作的部分，因为专门为过程定义的唯一操作是应用程序，这在第5章中进行了描述。端口上的操作在第 7 章中关于输入和输出的更一般性讨论中介绍。定义新数据类型的机制在第 9 章中进行了介绍。

第 6.1 节. 常量和报价

语法： 常量
返回： 常量

常量是任何自评估常量，即数字、布尔值、字符、字符串或字节向量。常量是不可变的;请参阅下面引用说明中的注释。

```
3.2 ⇒ 3.2
#f ⇒ #f
#\c #\c ⇒
“hi” ⇒ “hi”
#vu8（3 4 5） ⇒ #vu8（3 4 5）
```

语法： （引用 obj）
语法： ’obj
返回： obj
库： （rnrs base）， （rnrs）

’obj 等价于 （引用 obj）。缩写形式由方案阅读器转换为较长的形式（见阅读）。

引用抑制了obj的正常评估规则，允许obj用作数据。尽管可以引用任何 Scheme 对象，但对于自评估常量（即数字、布尔值、字符、字符串和字节向量）来说，引用不是必需的。

引用和自评估常量是不可变的。也就是说，程序不应通过 set-car!、string-set! 等方式更改常量，并且如果尝试进行此类更改，则允许实现引发具有条件类型和断言的异常。如果未检测到更改不可变对象的尝试，则程序的行为未指定。实现可以选择在不同常量之间共享存储以节省空间。

```
(+ 2 3)⇒ 5
’（+ 2 3） ⇒ （+ 2 3）
（报价 （+ 2 3）） ⇒ （+ 2 3）
’a ⇒ ’缺点
⇒
’() ⇒
’7 ⇒ 7
```

语法： （准引号 obj ...）
语法： ’obj
syntax： （unquote obj ...）
syntax： ，obj

syntax: (unquote-splicing obj ...)
语法: ,@obj
返回: 参见下面的
库: (rnrs base), (rnrs)

'obj 等价于 (quasiquote obj)，，obj 等价于 (unquote obj)，而 ,@obj 等价于 (unquote-splicing obj)。缩写形式由 Scheme 阅读器转换为较长的形式（参见阅读）。

准引号类似于引用，但它允许引用的文本的某些部分被“未引用”。在准引号表达式中，计算未引号和未引号拼接的子窗体，并引用其他所有内容，即不计算。每个未报价子窗体的值将插入到输出中以代替取消报价的表单，而每个未报价拼接子表单的值将拼接到周围的列表或矢量结构中。取消引号和取消引号拼接仅在准引号表达式中有效。

准引号表达式可以是嵌套的，每个准引号都会引入一个新的引用级别，每个取消引用或取消引用拼接会占用一个级别的引用。嵌套在 n 个准引号表达式中的表达式必须位于 n 个未引用或未引用拼接表达式中才能进行求值。

```
'(+ 2 3) ⇒ (+ 2 3)

'(+ 2 , (* 3 4)) ⇒ (+ 2 12)
'(a b (, (+ 2 3) c) d) ⇒ (a b (5 c) d)
'(a b , (反向 '(c d e)) f g) ⇒ (a b (e d c) f g)
(let ([a 1] [b 2])
'(, a . b)) ⇒ (1 . 2)

'(+ , @(cdr '(* 2 3))) ⇒ (+ 2 3)
'(a b , @(反向 '(c d e)) f g) ⇒ (a b e d c f g)
(let ([a 1] [b 2])
'(, a , @b)) ⇒ (1 . 2)
'#(, @(列表 1 2 3)) ⇒ #(1 2 3)

'', (cons 'a 'b) ⇒ ', (cons 'a 'b) '', (cons 'a 'b)
', (cons 'a 'b) ⇒ '(a .b)
```

具有零个或多个子窗体的取消引用和取消引用拼接形式仅在拼接（列表或矢量）上下文中有效。(取消引用 obj ...) 等价于 (unquote obj) ...，和 (unquote-splicing obj ...) 等价于 (unquote-splicing obj) ...。这些形式主要用作准报价扩展器输出中的中间形式。它们

支持某些有用的嵌套准引号习语 [3]，例如 ，@，@，当在双重嵌套和双重计算的准引号表达式中使用时，它具有双重间接拼接的效果。

```
’(a (unquote) b) ⇒ (a b)
’(a (unquote (+ 3 3)) b) ⇒ (a 6 b)
’(a (unquote (+ 3 3) (* 3 3)) b) ⇒ (a 6 9 b)

(let ([x ’(m n)]) ’’(a , @, @x f)) ⇒ ’(a (unquote-
splicing m n) f)
(let ([x ’(m n)])
(eval ’([m ’(b c)] [n ’(d e)]) ’(a , @, @x f))
(environment ’(rnrs))))  ⇒ (a b c d e f)
```

unquote 和 unquote-splicing 是准引号的辅助关键字。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。

第 6.2 节.泛型等价和类型谓词

本节介绍用于确定对象类型或两个对象的等效性的基本 Scheme 谓词（返回#t或#f的布尔值之一的过程）。首先讨论等价谓词 eq?、eqv? 和 equal?，然后是类型谓词。

过程：（方程式? obj1 obj2）
返回：如果 obj1 和 obj2 相同，则#t，否则#f
库：（rnrs base）、（rnrs）

在大多数 Scheme 系统中，如果两个对象在内部由相同的指针值表示，则认为它们是相同的，如果它们在内部由不同的指针值表示，则认为它们是不同的（不相同的），尽管其他条件（如时间戳）是可能的。

尽管对象标识的特定规则因系统而异，但以下规则始终适用。

- 两个不同类型的对象（布尔值、空列表、对、数字、字符、字符串、向量、符号和过程）是不同的。
- 具有不同内容或值的两个相同类型的对象是不同的。
- 布尔对象#t无论出现在何处，都与自身相同，无论它出现在哪里，#f都与自身相同，但#t和#f是不同的。

- 无论空列表 （） 出现在何处，它都与自身相同。
- 当且仅当两个符号具有相同的名称（按 string=? ） 时，两个符号是相同的。
- 常量对、向量、字符串或字节向量与自身相同，就像由 cons、vector、string、make-bytevector 等应用程序创建的对、向量、字符串或字节向量一样。由缺点、向量、字符串、生成字节向量等不同应用创建的两对向量、向量、字符串或字节向量是不同的。例如，一个后果是，缺点可能用于创建与所有其他对象不同的唯一对象。
- 两个可能行为不同的过程是不同的。由 lambda 表达式的计算创建的过程与自身相同。由同一 lambda 表达式在不同时间或由相似的 lambda 表达式创建的两个过程可能不同，也可能不同。

情 商? 不能用于可靠地比较数字和字符。尽管每个不准确的数字都与每个确切的数字不同，但两个确切的数字、两个不准确的数字或具有相同值的两个字符可能相同，也可能不同。

由于常量对象是不可变的，即程序不应通过 vector-set！、set-car！或任何其他结构突变操作来修改它们，因此不同引用常量或自评估文本的全部或部分可以在内部由同一对象表示。因此，当应用于不同不可变常量的相等部分时，eq? 可能会返回#t。

情 商? 最常用于比较交易品种或检查已分配对象的指针等效性，例如，对、向量或记录实例。

```
（等式？’a 3） ⇒ #f
（eq？ #t ’t） ⇒ #f
（eq？ “abc” ’abc） ⇒ #f
（eq？ “hi” ’（hi） ） ⇒ #f
（eq？ #f ’（） ） ⇒ #f

（eq？ 9/2 7/2） ⇒ #f
（eq？ 3.4 53344） ⇒ #f
（eq？ 3 3.0） ⇒ #f
（eq？ 1/3 #i1/3） ⇒ #f

（eq？ 9/2 9/2） ⇒ 未指定
（eq？ 3.4 （+ 3.0 .4） ） ⇒ 未指定
（let （[x （* 12345678987654321 2） ]）
```

```
(eq? x x)) ⇒未指定

(eq? #\a #\b) ⇒ #f
(eq? #\a #\a) ⇒ 未指定
(let ([x (string-ref “hi” 0)])
(eq? x x)) ⇒ 未指定

(eq? #t #t) ⇒ #t
(eq? #f #f) ⇒ #t
(eq? #t #f) ⇒ #f
(eq? (空?'()) #t) ⇒ #t
(eq? (空?'(a)) #f) ⇒ #t

(等式? (cdr '(a))'()) ⇒ #t

(eq?'a 'a) ⇒ #t
(eq?'a 'b) ⇒ #f
(eq?'a (string->symbol “a”)) ⇒ #t

(eq?'(a) '(b)) ⇒ #f
(等式?'(a) '(a)) ⇒ 未指定
(让 ([x '(a .b)]) (eq? x x)) ⇒ #t
(let ([x (cons 'a 'b)])
(eq? x x)) ⇒ #t
(eq? (缺点 'a 'b) (缺点 'a 'b)) ⇒
#f
(方程式? “abc” “cba”) ⇒ #f
(eq? “abc” “abc”) ⇒ 未指定
(let ([x “hi”]) (eq? x x)) ⇒ #t
(let ([x (string #\h #\i)]) (eq? x x)) ⇒ #t
(eq? (字符串 #\h #\i)
(string #\h #\i)) ⇒ #f

(eq?'#vu8(1) '#vu8(1)) ⇒ 未指明
(eq?'#vu8(1) '#vu8(2)) ⇒ #f
(let ([x (make-bytevector 10 0)])
(eq? x x)) ⇒ #t
(let ([x (make-bytevector 10 0)])
(eq? x (make-bytevector 10 0))) ⇒ #f

(eq?'#(a) '#(b)) ⇒ #f
(eq?'#(a) '#(a)) ⇒ 未指定
(let ([x '#(a)]) (eq? x x)) ⇒ #t
(let ([x (vector 'a)])
(eq? x x)) ⇒ #t
(eq? (向量'a) (向量'a)) ⇒
#f
```

```
(eq? car car car) ⇒ #t
(eq? car cdr) ⇒ #f
(let ([f (lambda (x) x)])
(eq? f f)) ⇒ #t
(let ([f (lambda () (lambda (x) x))])
(eq? (f) (f))) 未 ⇒ 指明
(等式? (lambda (x) x) (lambda (y) y)) ⇒ 未指定

(let ([f (lambda (x)
(lambda ()
(set! x (+ x 1)))
x)))
(eq? (f 0) (f 0)) ⇒ #f
```

过程： (eqv? obj1 obj2)
返回：如果 obj1 和 obj2 是等效的，则#t，否则#f
库：(rnrs base)、 (rnrs)

eqv? 与 eq? 相似，除了 eqv? 保证返回#t对于两个字符，这两个字符被 char=? 视为相等，并且 (a) 被 = 和 (b) 通过 eq? 和 eqv? 之外的任何其他操作都无法区分。(b) 的结果是 (eqv? -0.0 +0.0) #f即使 (= -0.0 +0.0) 在区分 -0.0 和 +0.0 的系统（例如基于 IEEE 浮点算术的系统）中#t。这是因为诸如 / 之类的操作可以公开差异：

```
(/ 1.0 -0.0)⇒ -inf.0
(/ 1.0 +0.0) ⇒ +inf.0
```

同样，尽管 3.0 和 3.0+0.0i 在数值上相等，但如果 -0.0 和 0.0 具有不同的表示形式，则 eqv 不认为它们等效。

```
(= 3.0+0.0i 3.0) ⇒ #t
(eqv? 3.0+0.0i 3.0) ⇒ #f
```

当参数为 NaN 时，未指定 eqv? 返回的布尔值。

```
(eqv? +nan.0 (/ 0.0 0.0)) ⇒ 未指定
```

eqv? 对实现的依赖性较低，但通常比eq? 更昂贵。

```
(eqv? 'a 3) ⇒ #f
(eqv? #t 't) ⇒ #f
(eqv? “abc” 'abc) ⇒ #f
```

```
(eqv? “hi” '(hi)) ⇒ #f
(eqv? #f '()) ⇒ #f

(eqv? 9/2 7/2) ⇒ #f
(eqv? 3.4 53344) ⇒ #f
(eqv? 3 3.0) ⇒ #f
(eqv? 1/3 #i1/3) ⇒ #f

(eqv? 9/2 9/2) ⇒ #t
(eqv? 3.4 (+ 3.0 .4)) ⇒ #t
(let ([x (* 12345678987654321 2)])
(eqv? x x)) ⇒ #t

(eqv? #\a #\b) ⇒ #f
(eqv? #\a #\a) ⇒ #t
(let ([x (string-ref “hi” 0)])
(eqv? x x)) ⇒ #t

(eqv? #t #t) ⇒ #t
(eqv? #f #f) ⇒ #t
(eqv? #t #f) ⇒ #f
(eqv? (空? '()) #t) ⇒ #t
(eqv? (空? '(a)) #f) ⇒ #t

(eqv? (cdr '(a)) '()) ⇒ #t

(eqv? 'a 'a) ⇒ #t
(eqv? 'a 'b) ⇒ #f
(eqv? 'a (string->symbol “a”)) ⇒ #t

(eqv? '(a) '(b)) ⇒ #f
(eqv? '(a) '(a)) ⇒ 未指定
(let ([x '(a .b)]) (eqv? x x)) ⇒ #t
(let ([x (cons 'a 'b)])
(eqv? x x)) ⇒ #t
(eqv? (缺点 'a 'b) (缺点 'a 'b)) ⇒
#f
(eqv? “abc” “cba”) ⇒ #f
(eqv? “abc” “abc”) ⇒ 未指定
(let ([x “hi”]) (eqv? x x)) ⇒ #t
(let ([x (string #\h #\i)]) (eqv? x x)) ⇒ #t
(eqv? (字符串 #\h #\i)
(string #\h #\i)) ⇒ #f

(eqv? '#vu8(1) '#vu8(1)) ⇒ 未指定
(eqv? '#vu8(1) '#vu8(2)) ⇒ #f
(let ([x (make-bytevector 10 0)])
```

```
(eqv? x x)) ⇒ #t
(let ([x (make-bytevector 10 0)])
(eqv? x (make-bytevector 10 0))) ⇒ #f

(eqv? '#(a) '#(b)) ⇒ #f
(eqv? '#(a) '#(a)) ⇒ 未指定
(let ([x '#(a)]) (eqv? x x)) ⇒ #t
(let ([x (vector 'a)])
(eqv? x x)) ⇒ #t
(eqv? (向量'a) (向量'a)) ⇒
#f
(eqv? car car car) ⇒ #t
(eqv? car cdr) ⇒ #f
(let ([f (lambda (x) x)])
(eqv? f f)) ⇒ #t
(let ([f (lambda () (lambda (x) x))])
(eqv? (f) (f))) ⇒ 未指明
(eqv? (lambda (x) x) (lambda (y) y)) ⇒ 未指定

(let ([f (lambda (x)
(lambda ()
(set! x (+ x 1))
x)])
(eqv? (f 0) (f 0)) ⇒ #f
```

过程：(等于? obj1 obj2)
返回：如果 obj1 和 obj2 具有相同的结构和内容，则返回#t，否则#f
库：(rnrs base)、(rnrs)

如果两个对象根据 eqv? 等效，则它们是相等的;字符串是 string=? ;的字节向量是 bytevector=? 的，汽车和 cdr 相等的对，或者是相同长度的向量，其相应的元素相等。

平等? 即使对于循环参数，也需要终止并返回#t“当且仅当其参数到常规树中的（可能是无限的）展开等于有序树”[24]。从本质上讲，两个值是等价的，在相等的意义上，如果两个对象的结构不能通过任何配对和向量访问器的组合以及eqv?，string=? 和 bytevector=? 过程来区分，用于比较叶子处的数据。

有效地实现相等是很棘手的[1]，即使有一个好的实现，它也可能比 eqv? 或eq? 更昂贵。

```
(相等? “a 3”) ⇒ #f
(等于? #t't) ⇒ #f
(等于? “abc” 'abc) ⇒ #f
(equal? “hi” '(hi)) ⇒ #f
(equal? #f '()) ⇒ #f

(equal? 9/2 7/2) ⇒ #f
(equal? 3.4 53344) ⇒ #f
(equal? 3 3 3.0) ⇒ #f
(equal? 1/3 #i1/3) ⇒ #f

(equal? 9/2 9/2) ⇒ #t
(equal? 3.4 (+ 3.0 .4)) ⇒ #t
(let ([x (* 12345678987654321 2)])
(equal? x x)) ⇒ #t

(equal? #\a #\b) ⇒ #f
(equal? #\a #\a) ⇒ #t
(let ([x (string-ref “hi” 0)])
(equal? x x)) ⇒ #t

(equal? #t #t) ⇒ #t
(equal? #f #f) ⇒ #t
(equal? #t #f) ⇒ #f
(equal? (空?'()) #t) ⇒ #t
(相等? (空?'(a)) #f) ⇒ #t

(相等? (cdr '(a))'()) ⇒ #t

(等于?'a 'a) ⇒ #t
(相等?'a 'b) ⇒ #f
(相等?'a (string->symbol “a”)) ⇒ #t

(equal?'(a) '(b)) ⇒ #f
(相等?'(a) '(a)) ⇒ #t
(let ([x '(a .b)]) (等于? x x)) ⇒ #t
(let ([x (cons 'a 'b)])
(等于? x x)) ⇒ #t
(等于? (缺点 'a 'b) (缺点 'a 'b)) ⇒
#t
(等于? “abc” “cba”) ⇒ #f
(等于? “abc” “abc”) ⇒ #t
(let ([x “hi”]) (equal? x x)) ⇒ #t
(let ([x (string #\h #\i)]) (equal? x x)) ⇒ #t
(equal? (字符串 #\h #\i)
(字符串 #\h #\i)) ⇒ #t
```

```
(等于? '#vu8(1) '#vu8(1)) ⇒ #t
(等于? '#vu8(1) '#vu8(2)) ⇒ #f
(let ([x (make-bytevector 10 0)])
(equal? x x)) ⇒ #t
(let ([x (make-bytevector 10 0)])
(equal? x (make-bytevector 10 0))) ⇒ #t

(equal? '#(a) '#(b)) ⇒ #f
(相等? '#(a) '#(a)) ⇒ #t
(let ([x '#(a)]) (等于? x x)) ⇒ #t
(let ([x (vector 'a)])
(equal? x x)) ⇒ #t
(等于? (向量'a) (向量'a)) ⇒
#t
(等于? 汽车汽车) ⇒ #t
(等于? 汽车 cdr) ⇒ #f
(让 ([f (lambda (x) x)])
(等于? f f)) ⇒ #t
(let ([f (lambda () (lambda (x) x))])
(等于? (f) (f))) 未 ⇒ 指明
(相等? (lambda (x) x) (lambda (y) y)) ⇒ 未指定

(let ([f (lambda (x)
(lambda ()
(set! x (+ x 1))
x))))
(等于? (f 0) (f 0)) ⇒
#f
(相等?
(let ([x (cons 'x 'x)])
(set-car! x x)
(set-cdr! x x)
x)
(let ([x (cons 'x 'x)])
(set-car! x x)
(set-cdr! x x)
(cons x x))) ⇒ #t
```

过程：（布尔值? obj）
返回：如果obj是#t或#f，则#t，否则#f
库：（rnrs base），（rnrs）

布尔? 等效于 (lambda (x) (or (eq? x #t) (eq? x #f)))))。)。

```
（布尔值？ #t）⇒ #t
（布尔值？#f） ⇒ #t
（或（布尔值？'t） （布尔值？'（））） ⇒ #f
```

过程：（空？obj）
返回：如果 obj 是空列表，则#t，否则#f
库：（rnrs base）、 （rnrs）

零？等效于 （lambda （x） （eq？ x '（））） 。

```
（空？'（）） ⇒ #t
（空？'（a）） ⇒ #f
（空？ （cdr '（a））） ⇒ #t
（空？ 3） ⇒ #f
（空？ #f） ⇒ #f
```

过程：（配对？obj）
返回：如果 obj 是一对，则#t，#f否则
库：（rnrs base）、 （rnrs）

```
（配对？'（a b c）） ⇒ #t
（对？'（3 . 4）） ⇒ #t
（对？'（）） ⇒ #f
（对？'#（a b）） ⇒ #f
（对？ 3） ⇒ #f
```

过程：（数字？obj）
返回：如果obj是数字对象，#t，#f否则
过程：（复杂？obj）
返回：#t如果obj是复数对象，#f否则
过程：（实数？obj）
返回：#t如果obj是实数对象，#f否则
过程：（有理数？obj）
返回：如果obj是有理数对象，#t，#f否则
过程：（整数？obj）
返回：如果 obj 是整数对象，则#t，否则#f
库：（rnrs base）、 （rnrs）

这些谓词形成一个层次结构：任何整数都是有理数，任何有理数都是实数，任何实数都是复数，任何复数都是数字。大多数实现不提供无理数的内部表示，因此所有实数通常也是有理数。

实数、有理数和整数谓词不识别为实数、有理数或整数复数，虚部为零。

```
(整数? 1901) ⇒ #t
(有理? 1901) ⇒ #t
(实数? 1901) ⇒ #t
(复数? 1901) ⇒ #t
(数字? 1901) ⇒ #t

(整数? -3.0) ⇒ #t
(有理数? -3.0) ⇒ #t
(实数? -3.0) ⇒ #t
(复数? -3.0) ⇒ #t
(数字? -3.0) ⇒ #t

(整数? 7+ 0i) ⇒ #t
(理性? 7+ 0i) ⇒ #t
(实数? 7+0i) ⇒ #t
(复数? 7+0i) ⇒ #t
(数字? 7+0i) ⇒ #t

(整数? -2/3) ⇒ #f
(有理数? -2/3) ⇒ #t
(实数? -2/3) ⇒ #t
(复数? -2/3) ⇒ #t
(数字? -2/3) ⇒ #t

(整数? -2.345) ⇒ #f
(有理数? -2.345) ⇒ #t
(实数? -2.345) ⇒ #t
(复数? -2.345) ⇒ #t
(数字? -2.345) ⇒ #t

(整数? 7.0+0.0i) ⇒ #f
(有理数? 7.0+0.0i) ⇒ #f
(实数? 7.0+0.0i) ⇒ #f
(复数? 7.0+0.0i) ⇒ #t
(数字? 7.0+0.0i) ⇒ #t

(整数? 3.2-2.01i) ⇒ #f
(有理? 3.2-2.01i) ⇒ #f
(实数? 3.2-2.01i) ⇒ #f
(复合? 3.2-2.01i) ⇒ #t
(数字? 3.2-2.01i) ⇒ #t

(整数? 'a) ⇒ #f
```

```
(理性？'(a b c)) ⇒ #f
(真的？ "3") ⇒ #f
(复杂？'#(1 2)) ⇒ #f
(number？ #\a) ⇒ #f
```

程序：(实值？obj)
返回：#t如果obj是实数，#f否则
过程：(有理值？obj)
返回：如果obj是有理数，#t，#f否则
过程：(整数值？obj)
返回：如果 obj 是整数，则#t，#f否则
库：(rnrs base)、 (rnrs)

这些谓词类似于实数、有理数和整数？，但被视为实数、有理数或整数复数，虚数部分为零。

```
(整数值？ 1901)⇒ #t
(有理值？-1901) ⇒ #t
(实值？1901) ⇒ #t

(整数值？-3.0) ⇒ #t
(有理值？-3.0) ⇒ #t

(实值？-3.0) ⇒ #t
(整数值？7 + 0i) ⇒ #t
(有理值？7 + 0i) ⇒ #t
(实值？7 + 0i) ⇒ #t

(整数值？ -2/3) ⇒ #f
(有理值？ -2/3) ⇒ #t
(实值？ -2/3) ⇒ #t

(整数值？ -2.345) ⇒ #f
(有理值？ -2.345) ⇒ #t
(实值？ -2.345) ⇒ #t

(整数值？ 7.0+0.0i) ⇒ #t
(有理值？ 7.0+0.0i) ⇒ #t
(实值？ 7.0+0.0i) ⇒ #t

(整数值？ 3.2-2.01i) ⇒ #f
(有理值？ 3.2-2.01i) ⇒ #f
(实值？ 3.2-2.01i) ⇒ #f
```

与实数？、有理数？和整数？一样，这些谓词为所有非数值返回#f。

```
（整数值？’a） ⇒ #f
（理性价值？’（a b c）） ⇒ #f
（实际价值？ “3”） ⇒ #f
```

过程：（字符？obj）
返回：如果 obj 是字符，则#t，否则#f
库：（rnrs base）、 （rnrs）

```
（字符？’a） ⇒ #f
（char？ 97） ⇒ #f
（char？ #\a） ⇒ #t
（char？ “a”） ⇒ #f
（char？ （string-ref （make-string 1） 0））⇒ #t
```

过程：（字符串？obj）
返回：如果 obj 是字符串，则#t，否则#f
库：（rnrs base）、 （rnrs）

```
（字符串？ “hi”） ⇒ #t
（字符串？’hi） ⇒ #f
（string？ #\h） ⇒ #f
```

过程：（矢量？obj）
返回：如果 obj 是向量，则#t，否则#f
库：（rnrs base）、 （rnrs）

```
（矢量？’#（）） ⇒ #t
（vector？’#（a b c）） ⇒ #t
（矢量？（向量’a ’b ’c））⇒ #t
（矢量？’（）） ⇒ #f
（矢量？’（a b c）） ⇒ #f
（矢量？ “abc”） ⇒ #f
```

过程：（符号？obj）
返回：如果 obj 是符号，则#t，#f否则
库：（rnrs base）， （rnrs）

```
（符号？’t） ⇒ #t
（符号？ “t”） ⇒ #f
（符号？’（t）） ⇒ #f
（符号？ #\t） ⇒ #f
```

```
（符号？ 3） ⇒ #f
（符号？ #t） ⇒ #f
```

程序：（程序？obj）
返回：如果 obj 是一个过程，则#t，否则#f
库：（rnrs base）、（rnrs）

```
（程序？⇒ #t
（程序？’汽车） ⇒ #f
（程序？（lambda （x） x））⇒ #t
（程序？’（lambda （x） x）） ⇒ #f
（call/cc procedure？）⇒ #t
```

过程：（字节向量？obj）
返回：如果 obj 是字节向量，则#t，否则#f
库：（rnrs bytevectors）、（rnrs）

```
（bytevector？ #vu8（））⇒ #t
（字节向量？’#（）） ⇒ #f
（字节向量？“abc”） ⇒ #f
```

程序：（哈希算？obj）
返回：如果obj是哈希表，#t，#f否则
库：（rnrs hashtables），（rnrs）

```
（可读物？（make-eq-hashtable））⇒ #t
（哈希算符？’（不是哈希表）） ⇒ #f
```

第 6.3 节.列表和对

对或缺点单元格是 Scheme 最基本的结构化对象类型。对最常见的用途是构建列表，这些列表是通过 cdr 字段将一对链接到下一个对的有序序列。列表中的元素占据了对的汽车场。正确列表中最后一对的 cdr 是空列表 （）;不正确列表中最后一对的 cdr 可以是（） 以外的任何内容。

对可用于构造二叉树。树结构中的每对都是二叉树的内部节点;它的 car 和 cdr 是节点的子节点。

正确的列表打印为用空格分隔并用括号括起来的对象序列。可以使用匹配的括号（ [ ] ）对代替括号。例如，（1 2 3） 和 （[嵌套列

表]） 是正确的列表。空列表写为 （）。

不正确的列表和树需要稍微复杂的语法。单个对写为由空格和点分隔的两个对象，例如（.b）。这称为点对表示法。不正确的列表和树也是用虚线对符号编写的;点出现在必要时，例如（1 2 3 . 4）或（（1 . 2） . 3）。正确的列表也可以用虚线对表示法编写。例如，（1 2 3） 可以写为 （1 .(2 .(3 .()))）.

通过使用 set-car! 或 set-cdr! 破坏性地改变一对的 car 或 cdr 字段，可以创建一个圆形列表或循环图。此类列表不被视为适当的列表。

接受列表参数的过程需要检测列表是否不正确，只有当它们实际遍历列表足够远（a）尝试在非列表尾部操作或（b）由于循环性而无限期循环。例如，如果成员实际找到正在查找的元素，则不需要检测列表是否不正确，并且 list-ref 永远不需要检测循环，因为它的递归受 index 参数的约束。

过程： （cons obj1 obj2）
返回： 一对新对，其 car 和 cdr 是 obj1 和 obj2
库： （rnrs base）， （rnrs）

缺点是配对构造函数过程。obj1 成为汽车，obj2 成为新对的 cdr。

```
（缺点 'a '（））⇒ （a）
（缺点 'a '（b c））⇒ （a b c）
（缺点 3 4）⇒ （3 . 4）
```

过程： （汽车对）
返回：对库的车
：（rnrs base），（rnrs）

空列表不是一对，因此参数不能是空列表。

```
（汽车'（a））⇒ a
（汽车 '（a b c））⇒ a
（汽车 （缺点 3 4））⇒ 3
```

过程： （cdr 对）
返回：对

库的 cdr：（rnrs base），（rnrs）

空列表不是一对，因此参数不能是空列表。

```
（cdr ’（a））⇒ （）
（cdr ’（a b c））⇒ （b c）
（cdr （cons 3 4））⇒ 4
```

程序：（设置车！pair obj）
返回：未指定的
库：（rnrs 可变对）

设置车！将配对的汽车更改为obj。

```
（let （[x （list ’a ’b ’c）]）
（set-car！ x 1）
x）⇒ （1 b c）
```

过程：（设置 cdr！pair obj）
返回：未指定的
库：（rnrs 可变对）

设置- cdr！将配对的 cdr 更改为 obj。

```
（let （[x （list ’a ’b ’c）]）
（set-cdr！ x 1）
x）⇒ （a. 1）
```

过程：（caar 对）
过程：（cadr 对）⋮
过程：（cddddr 对）
返回：caar、cadr、...或 cddddr 的对
库库：（rnrs base）、（rnrs）

这些程序被定义为最多四辆车和CDR的组成。c 和 r 之间的 a 和 d 表示 car 或 cdr 的应用，从右到左的顺序排列。例如，应用于一对的过程 cadr 产生该对的 cdr 的 car，并且等效于 （lambda （x）（car （cdr x））））。

```
（caar ’（（a））⇒ a
（cadr ’（a b c））⇒ b
```

```
(cdddr '(a b c d)) ⇒ (d)
(cadadr '(a (b c))) ⇒ c
```

过程: (列表 obj ...)
返回: obj ...
库的列表: (rnrs base), (rnrs)

list 等效于 (lambda x x)。

```
(列表) ⇒ ()
(清单 1 2 3) ⇒ (1 2 3)
(清单 3 2 1) ⇒ (3 2 1)
```

程序: (缺点* obj ... final-obj)
返回: 由 final-obj 库终止的 obj
... 列表: (rnrs 列表)、 (rnrs)

如果省略对象 obj ...，则结果只是 final-obj。否则，将构造 obj ... 的列表，就像 list 一样，只是最终的 cdr 字段是 final-obj 而不是 ()。如果 final-obj 不是列表，则结果是不正确的列表。

```
(缺点* '()) ⇒ ()
(缺点* '(a b)) ⇒ (a b)
(缺点* 'a 'b 'c) ⇒ (a b . c)
(缺点* 'a 'b '(c d)) ⇒ (a b c d)
```

过程: (列表? obj)
返回: #t obj是否是一个正确的列表，#f否则
库: (rnrs base), (rnrs)

列表?必须为所有不正确的列表（包括循环列表）返回#f。列表的定义见第67页。

```
(列表? '()) ⇒ #t
(列表? '(a b c)) ⇒ #t
(列表? 'a) ⇒ #f
(名单? '(3 . 4)) ⇒ #f
(list? 3) ⇒ #f
(let ([x (list 'a 'b 'c)])
(set-cdr! (cddr x) x)
(list? x)) ⇒ #f
```

过程：（长度列表）
返回：列表
库中的元素数：（rnrs base），（rnrs）

长度 可以定义如下，使用第 67 页上用于定义 列表 的野兔和算法的改编。

```
(定义长度
(lambda (x)
(定义不适当的列表
(lambda ()
(断言违规 'length "不是适当的列表" x)))

(让 f ([h x] [t x] [n 0])
(如果 (对? h)
(让 ([h (cdr h)])
(如果 (成对? h)
(如果 (eq? h t)
(不当列表)
(f (cdr h) (cdr t) (+ n 2)))
(如果 (null? h)
(+ n 1)
(不当列表)))))
(如果 (null? h)
n
(improper-list)))))

(长度 '()) ⇒ 0
(长度 '(a b c)) ⇒ 3
(长度 '(a b . c)) ⇒ 异常
(长度
(let ([ls (list 'a 'b)])
(set-cdr! (cdr ls) ls) ⇒ exception
ls))
(length
(let ([ls (list 'a 'b)])
(set-car! (cdr ls) ls) ⇒ 2
ls))
```

过程：（list-ref list n）
返回：列表
库的第 n 个元素（从零开始）：（rnrs base）、（rnrs）

n 必须是小于列表长度的精确非负整数。可以定义 list-ref 而不进行错误检查，如下所示。

```
(define list-ref
(lambda (ls n)
(if (= n 0)
(car ls)
(list-ref (cdr ls) (- n 1)))))

(list-ref '(a b c) 0) ⇒ a
(list-ref '(a b c) 1) ⇒ b
(list-ref '(a b c) 2) ⇒ c
```

过程：（列表尾列表 n）
返回：列表
库的第 n 个尾部（从零开始）：（rnrs base）、（rnrs）

n 必须是小于或等于列表长度的精确非负整数。结果不是副本;尾部是eq? 到列表的第n个cdr（或者列表本身，如果n为零）。

列表尾部可以在不进行错误检查的情况下定义，如下所示。

```
(定义 list-tail
(lambda (ls n)
(if (= n 0)
ls
(list-tail (cdr ls) (- n 1)))))

(列表尾 '(a b c) 0) ⇒ (a b c)
(list-tail '(a b c) 2) ⇒ (c)
(list-tail '(a b c) 3) ⇒ ()
(list-tail '(a b c. d) 2) ⇒ (c . d)
(list-tail '(a b c. d) 3) ⇒ d
(let ([x (list 1 2 3)])
(eq? (列表尾 x 2)
(cddr x))) ⇒ #t
```

过程：（追加）
过程：（追加列表...obj）
返回：输入列表
库的串联：（rnrs base），（rnrs）

append 返回一个新列表，该列表由第一个列表的元素后跟第二个列表的元素、第三个列表的元素等组成。新列表由除最后一个参数之外的所有参数的新对组成;最后一个（不必是列表）只是放在新结构的末尾。可以在不进行错误检查的情况下定义追加，如下所示。

```
(define append
(lambda args
(let f ([ls '()] [args args])
(if (null? args)
ls
(let g ([ls ls])
(if (null? ls)
(f (car args) (cdr args))
(cons (car ls) (g (cdr ls))))))))

(附加 '(a b c) '()) ⇒ (a b c)
(附加 '() '(a b c)) ⇒ (a b c)
(附加 '(a b) '(c d)) ⇒ (a b c d) (a b c)
(附加 '(a b) 'c) ⇒ (a b .c)
(让 ([x (列表 'b)])
(eq? x (cdr (附加 '(a) x)))) ⇒ #t
```

过程：（反向列表）
返回：一个新列表，其中包含以相反顺序排列的列表的元素库：（rnrs base），（rnrs）

可以定义反向，而无需进行错误检查，如下所示。

```
(定义反向
(lambda (ls)
(let rev ([ls ls] [new '()])
(if (null? ls)
new
(rev (cdr ls) (cons (car ls) new))))))

(反向 '()) ⇒ ()
(反向 '(a b c)) ⇒ (c b a)
```

过程：（memq obj list）
过程：（memv obj list）
过程：（成员 obj 列表）
返回：列表的第一个尾部，其汽车等效于 obj，或#f
库：（rnrs 列表）、（rnrs）

这些过程按顺序遍历参数列表，将 list 的元素与 obj 进行比较。如果找到等效于 obj 的对象，则返回其第一个元素为该对象的列表尾部。如果列表包含多个等效于 obj 的对象，则返回第一个元素等效于 obj 的第一个尾部。如果未找到等效于 obj 的对象，则返回#f。memq 的等价检验是 eq?，memv 是 eqv?，成员是相等的?。

这些过程最常用作谓词，但它们的名称不会以问号结尾，因为它们返回一个有用的 true 值来代替#t。memq 可以在不进行错误检查的情况下进行定义，如下所示。

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

memv 和成员可以定义类似，用 eqv? 和 equal? 代替 eq?。

```
(memq 'a '(b c a d e)) ⇒ (a d e)
(memq 'a '(b c d e g)) ⇒ #f
(memq 'a '(b a c a d a)) ⇒ (a c a d a)

(memv 3.4 '(1.2 2.3 3.4 4.5)) ⇒ (3.4 4.5)
(memv 3.4 '(1.3 2.5 3.7 4.9)) ⇒ #f
(let ([ls (list 'a 'b 'c)])
(set-car! (memv 'b ls) 'z)
ls) ⇒ (a z c)

(成员 '(b) '((a) (b) (c))) ⇒ ((b) (c))
(成员 '(d) '((a) (b) (c))) ⇒ #f
(成员 “b” '(“a” “b” “c”)) ⇒ (“b” “c”)

(让 ()
(定义成员?
(lambda (x ls)
(and (member x ls) #t)))
(成员?'(b) '((a) (b) (c)))) ⇒ #t

(定义计数出现次数
(lambda (x ls)
(cond
[(memq x ls) =>
(lambda (ls)
```

```
（+ （count-occurrences x （cdr ls）） 1））
[else 0]）））
```

```
（计数出现次数 'a '（a b c d a）） ⇒ 2
```

过程：（memp 过程列表）
返回：其汽车过程返回 true 的列表的第一个尾部，或#f
库：（rnrs 列表）、（rnrs）

过程应接受一个参数并返回单个值。它不应修改列表。

```
（咕噜咕噜？'（1 2 3 4）） ⇒ （1 2 3 4）
（甚至？'（1 2 3 4）） ⇒ （2 3 4）
（let （[ls （列表 1 2 3 4）]）
（方程式？（memp odd？ ls） ls）） ⇒ #t
（let （[ls （list 1 2 3 4）]）
（eq？（甚至吗？ ls）（cdr ls））⇒ #t
（奇怪的？'（2 4 6 8）） ⇒ #f
```

过程：（remq obj list）
过程：（remv obj list）
过程：（删除 obj 列表）
返回：包含所有出现的 obj 删除
库的列表的元素的列表：（rnrs 列表）、（rnrs）

这些过程遍历参数列表，删除任何等效于 obj 的对象。输出列表中剩余的元素与它们在输入列表中的显示顺序相同。如果列表的尾部（包括列表本身）不包含 obj 的出现次数，则结果列表的相应尾部可能与输入列表的尾部相同（按 eq？）。

remq 的等价检验是 eq？，remv 是 eqv？，remove 是相等的？。

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

（remv 1/2 '（1.2 1/2 0.5 3/2 4）） ⇒ （1.2 0.5 3/2 4）

（删除'（b） '（（a） （b） （c））） ⇒ （（a） （c））
```

过程：（remp 过程列表）
返回：过程返回#f
库的列表的元素列表：（rnrs 列表）、（rnrs）

过程应接受一个参数并返回单个值。它不应修改列表。

remp 将过程应用于 list 的每个元素，并返回一个列表，该列表仅包含过程返回#f的元素。返回列表的元素的显示顺序与它们在原始列表中出现的顺序相同。

```
（雷普奇怪？’（1 2 3 4））⇒ （2 4）
（remp
（lambda （x） （and （> x 0） （< x 10））））
’（-5 15 3 14 -20 6 0 -9））⇒ （-5 15 14 -20 0 -9）
```

过程：（筛选过程列表）
返回：列表的元素列表，过程为其返回真正的
库：（rnrs 列表）、（rnrs）

过程应接受一个参数并返回单个值。它不应修改列表。

filter 将过程应用于 list 的每个元素，并返回一个新列表，该列表仅包含过程为其返回 true 的元素。返回列表的元素的显示顺序与它们在原始列表中出现的顺序相同。

```
（过滤器奇怪？’（1 2 3 4））⇒ （1 3）
（过滤器
（lambda （x） （and （> x 0） （< x 10））））
’（-5 15 3 14 -20 6 0 -9））⇒ （3 6）
```

过程：（分区过程列表）
返回：请参阅下面的
库：（rnrs 列表）、（rnrs）

过程应接受一个参数并返回单个值。它不应修改列表。

partition 将过程应用于 list 的每个元素，并返回两个值：一个新列表仅包含过程返回 true 的元素，另一个新列表仅包含过程返回#f的元素。返回列表的元素的显示顺序与它们在原始列表中出现的顺序相同。

```
（分区奇怪？’（1 2 3 4））⇒ （1 3）
  （2 4）
（分区
（lambda （x） （and （> x 0） （< x 10））））
```

```
’(-5 15 3 14 -20 6 0 -9)) ⇒ (3 6)
   (-5 15 14 -20 0 -9)
```

分区返回的值可以通过分别调用 filter 和 remp 来获取，但这需要对 list 的每个元素进行两次过程调用。

过程：（查找过程列表）
返回： 过程为其返回 true 的列表的第一个元素，或#f
库：（rnrs 列表）、（rnrs）

过程应接受一个参数并返回单个值。它不应修改列表。

find 按顺序遍历参数列表，依次将过程应用于每个元素。如果 procedure 为给定元素返回 true 值，则 find 将返回该元素，而不对其余元素应用过程。如果过程为列表的每个元素返回#f，则查找返回#f。

如果程序必须区分在列表中查找#f和根本不查找任何元素，则应改用 memp。

```
(发现奇怪? ’(1 2 3 4)) ⇒ 1
(找到偶数? ’(1 2 3 4)) ⇒ 2
(发现奇数? ’(2 4 6 8)) ⇒ #f
(find not ’(1 a #f 55)) ⇒ #f
```

过程： (assq obj alist)
过程： (assv obj alist)
过程： (assoc obj alist)
返回： alist 的第一个元素，其汽车等效于 obj，或#f
库： (rnrs 列表)、 (rnrs)

参数列表必须是关联列表。关联列表是一个正确的列表，其元素是表单的键值对（键 . value）。关联对于存储与某些对象（键）关联的信息（值）非常有用。

这些过程遍历关联列表，测试每个键是否与 obj 等效。如果找到等效键，则返回键值对。否则，将返回#f。

assq 的等价检验是 eq?，对于 assv 是 eqv?，对于 assoc 是相等的?。assq 可以在不进行错误检查的情况下进行定义，如下所示。

```
(define assq
(lambda (x ls)
(cond
[(null? ls) #f]
[(eq? (caar ls) x) (car ls)]
[else (assq x (cdr ls))]))))
```

assv 和 assoc 的定义可能类似，用 eqv? 和 equal? 代替 eq? 。

```
(assq 'b '(a . 1) (b . 2))) ⇒ (b . 2)
(cdr (assq 'b '((a . 1) (b . 2)))) ⇒ 2
(assq 'c '((a . 1) (b . 2))) ⇒ #f

(assv 2/3 '((1/3 . 1) (2/3 . 2))) ⇒ (2/3 . 2)
(assv 2/3 '((1/3. a) (3/4 .b))) ⇒ #f

(assoc '(a) '(((a) . a) (-1 .b))) ⇒ ((a)
(assoc '((b) .b) (a .c))) ⇒ #f

([alist (cons 2 'a) (cons 3 'a))])
(设置 cdr! (assv 3 alist)'c)
alist) ⇒ ((2 . a) (3 .c))
```

第 12.7 节中给出的解释器将环境表示为关联列表，并使用 assq 进行变量查找和赋值。

过程：（assp 过程 alist）
返回：其 car 过程返回 true 的 alist 的第一个元素，或#f
库：（rnrs 列表）、（rnrs）

alist 必须是关联列表。关联列表是一个正确的列表，其元素是表单的键值对（键 . value）。过程应接受一个参数并返回单个值。它不应修改列表。

```
(屁股奇怪?'(1 . a) (2 .b))) ⇒ (1 . a)
(assp even?'((1 . a) (2 .b))) ⇒ (2 .b)
(let ([ls (list (cons 1 'a) (cons 2 'b))])
(eq? (assp odd? ls) (汽车 ls)) ⇒ #t
(let ([ls (list (cons 1 'a) (cons 2 'b))])
(eq? (屁股甚至? ls) (cadr ls)) ⇒ #t
(屁股奇数?'(2 .b)) ⇒ #f
```

过程：（列表排序谓词列表）
返回：包含根据谓

词库排序的列表元素的列表：（rnrs 排序）、（rnrs）

谓词应该是一个过程，它需要两个参数，如果它的第一个参数必须在排序列表中的第二个参数之前，则返回#t。也就是说，如果谓词应用于两个元素 x 和 y，其中 x 出现在输入列表中的 y 之后，则仅当 x 应出现在输出列表中的 y 之前时，它才应返回 true。如果满足此约束，则 list-sort 将执行稳定排序，即仅在必要时根据谓词对两个元素进行重新排序。不会删除重复的元素。此过程可以调用谓词，最多 nlogn 次，其中 n 是列表的长度。

```
(列表排序< '(3 4 2 1 2 5)) ⇒ (1 2 2 3 4 5)
(列表排序> '(0.5 1/2)) ⇒ (0.5 1/2)
(列表排序> '(1/2 0.5)) ⇒ (1/2 0.5)
(list->string
(list-sort char>?
(字符串>列表“你好”)) ⇒ “哎呀”
```

第 6.4 节.数字

方案数可分为整数、有理数、实数或复数。这种分类是分层的，因为所有整数都是有理数，所有有理数都是实数，所有实数都是复数。第 6.2 节中描述的谓词整数、有理数、实数和复数用于确定数字属于这些类中的哪一个。

方案编号也可以分为精确或不精确，具体取决于用于派生编号的操作质量以及这些操作的输入。谓词 exact? 和 inexexact? 可用于确定数字的精确性。Scheme 中对数字的大多数运算都是精确性保留的：如果给定精确的操作数，它们将返回精确的值，如果给定不精确的操作数或精确和不精确操作数的组合，则它们返回不精确的值。

精确整数和有理算术通常支持任意精度;整数的大小或比率的分母或分子的大小仅受系统存储约束的限制。尽管其他表示形式是可能的，但不准确的数字通常由主机硬件或系统软件支持的浮点数表示。复数通常表示为有序对（实部、意象部分），其中实部和意数部分是精确整数、精确有理数或浮点数。

方案编号的编写方式与编写数字的普通惯例没有太大区别。精确整数通常写为一系列数字，前面有可选符号。例如，3、+19、-100000 和 208423089237489374都表示确切的整数。

精确的有理数通常写为两个数字序列，由斜杠（/）分隔，前面有一个可选符号。例如，3/4、-6/5 和 1/1208203823 都是精确的有理数。当一个比率被读取时，它会立即减少到最低项，实际上可能会减少到一个精确的整数。

不准确的实数通常以浮点或科学记数法书写。浮点表示法由一系列数字后跟一个小数点和另一个数字序列组成，所有数字前面都有一个可选符号。科学记数法由一个可选符号，一个数字序列，一个可选的小数点后跟第二串数字和一个指数组成;指数写为字母 e，后跟可选符号和数字序列。例如，1.0 和 -200.0 是有效的不精确整数，1.5、0.034、-10e-10 和 1.5e-5 是有效的不精确有理数。指数是 10 的幂，指数前面的数字应按此比例缩放，因此 2e3 相当于 2000.0。

尾数宽度|w 可以显示为实数的后缀，也可以显示为以浮点或科学记数法书写的复数的实数分量。尾数宽度 w 表示数字表示中的有效位数。尾数宽度默认为 53，即归一化 IEEE 双浮点数中的有效位数，或更多。对于非规范化的 IEEE 双浮点数，尾数的宽度小于 53。如果实现不能表示具有指定尾数宽度的数字，则它使用具有至少与请求的尽可能多的有效位（如果可能）的表示形式，否则它将使用具有最大尾数宽度的表示形式。

精确和不精确的实数写成精确或不精确的整数或有理数;对于非理性实数，即无理数，在方案数的语法中没有规定。

复数可以写成矩形或极性形式。在矩形形式中，复数写为 x+yi 或 x-yi，其中 x 是整数、有理数或实数，y 是无符号整数、有理数或实数。可以省略实部 x，在这种情况下，假定它为零。例如，3+4i、3.2-3/4i、+i 和 -3e-5i 是以矩形形式书写的复数。在极性形式中，复数写为 x@y，其中 x 和 y 是整数、有理数或实数。例如，1.1@1.764 和 -1@-1/2 是以极性形式书写的复数。

语法 +inf.0 和 -inf.0 表示表示正无穷大和负无穷大的不精确实数。语法 +nan.0 和 -nan.0 表示一个不精确的“非数字”（NaN）值。无穷大可以通过将不精确的正值和负值除以不精确的零来产生，而NaN也可以通过将不精确的零除以不精确的零等方式产生。

数字表示的精确性可以通过在表示形式之前由#e或#i来覆盖。#e迫使数字准确，#i迫使它不准确。例如，1、#e1、1/1、#e1/1、#e1.0 和

#e1e0 都表示确切的整数 1，而 #i3/10、0.3、#i0.3 和 3e-1 都表示不精确的有理数 0.3。

默认情况下，数字以 10 为基数，但特殊前缀#b（二进制）、#o（八进制）、#d（十进制）和#x（十六进制）可用于指定基数 2、基数 8、基数 10 或基数 16。对于基数 16，字母 a 到 f 或 A 到 F 用作表示数字值 10 到 15 所需的附加数字。例如，#b10101 是 21_{10} 的二进制等效值，#o72是 58_{10} 的八进制等效值，#xC7是 199$_{10}$ 的十六进制等效值。以浮点和科学记数法书写的数字总是以10为基数。

如果两者都存在，则基数和精确度前缀可以按任一顺序显示。

Scheme 实现可能支持多个大小的内部表示形式，用于不准确的数量。指数标记 s（短）、f（单）、d（双）和 l（长）可能代替默认指数标记 e 出现，以覆盖用科学记数法书写的数字的默认大小。在支持多种表示形式的实现中，默认大小的精度至少与双精度一样高。

方案编号的精确语法在第459页给出。

任何数字都可以以各种不同的方式书写，但系统打印机（通过放置基准，写入和显示调用）和数字>字符串以紧凑的形式表示数字，使用所需的最少数字来保留读取时打印的数字与原始数字相同的属性。

本节的其余部分介绍对数字进行操作的“通用算术”过程。本节后面的两节介绍特定于固定数和絮状体的操作，它们是精确、固定精度整数值和不精确实数值的表示形式。

本节中过程接受的数字参数类型由参数的名称暗示：复数（即所有数字）的 num，实数表示实数，rat 表示有理数，int 表示整数。如果需要实数、大数或整数，则该参数必须被视为实数、有理数或实数、有理数或整数的积分，即，数字的虚部必须正好为零。在需要精确整数的情况下，将使用名称 exint。在每种情况下，名称上都可能出现后缀，例如 int2。

程序：（确切？数字）
返回：如果 num 是精确的，则#t，#f否则
库：（rnrs base）、（rnrs）

```
(精确? 1) ⇒ #t
(精确? -15/16) ⇒ #t
(精确? 2.01) ⇒ #f
(精确? #i77) ⇒ #f
(精确? #i2/3) ⇒ #f
(精确? 1.0-2i) ⇒ #f
```

程序：(不准确? 数字)
返回：如果 num 不精确，则#t，#f否则
库：(rnrs base)、(rnrs)

```
(不准确? -123) ⇒ #f
(不精确? #i123) ⇒ #t
(不精确? 1e23) ⇒ #t
(不精确? +i) ⇒ #f
```

过程：(= num1 num2 num3 ...)
过程：(< real1 real2 real3 ...)
过程：(> real1 real2 real3 ...)
过程：(<= real1 real2 real3 ...)
过程：(>= real1 real2 real3 ...)
返回：如果关系成立，则#t，#f否则
库：(rnrs base)， (rnrs)

谓词 = 返回 #t，如果它的参数相等。谓词<返回#t，如果它的参数单调递增，即每个参数都大于前面的参数，而>返回#t如果其参数单调递减。谓词 <= 返回 #t，如果它的参数是单调不递减的，即每个参数不小于前面的参数，而 >= 返回 #t如果它的参数是单调不递增的。

正如参数名称所暗示的那样，= 是为复杂参数定义的，而其他关系谓词仅为实数参数定义。如果两个复数的实部和虚部相等，则认为它们相等。涉及 NaNs 的比较总是返回#f。

```
(= 7 7) ⇒ #t
(= 7 9) ⇒ #f

(< 2e3 3e2) ⇒ #f
(<= 1 2 3 3 4 5) ⇒ #t
(<= 1 2 3 4 5) ⇒ #t

(> 1 2 2 3 3 4) ⇒ #f
```

```
(>= 1 2 2 3 3 4) ⇒ #f

(= -1/2 -0.5) ⇒ #t
(= 2/3 .667) ⇒ #f
(= 7.2+0i 7.2) ⇒ #t
(= 7.2-3i 7) ⇒ #f

(< 1/2 2/3 3/4) ⇒ #t
(> 8 4.102 2/3 -5) ⇒ #t

(let ([x 0.218723452])
(< 0.210 x 0.220)) ⇒ #t

(let ([i 1] [v (vector 'a 'b 'c)])
(< -1 i (vector-length v))) ⇒ #t

(apply < '(1 2 3 4)) ⇒ #t
(apply > '(4 3 3 2)) ⇒ #f

(= +nan.0 +nan.0) ⇒ #f
(< +nan.0 +nan.0) ⇒ #f
(> +nan.0 +nan.0) ⇒ #f
(>= +inf.0 +nan.0) ⇒ #f
(>= +nan.0 -inf.0) ⇒ #f
(> +nan.0 0) ⇒ #f
```

过程： (+ 数字 ...)
返回：参数之和数 ...
库：(rnrs base)， (rnrs)

当调用时不带任何参数，+ 返回 0。

```
(+)⇒ 0
(+ 1 2) ⇒ 3
(+ 1/2 2/3) ⇒ 7/6
(+ 3 4 5) ⇒ 12
(+ 3.0 4) ⇒ 7.0
(+ 3+4i 4+3i) ⇒ 7+7i
(申请 + '(1 2 3 4 5)) ⇒ 15
```

过程： (- num)
返回： num
的加法逆过程： (- num1 num2 num3 ...)

返回： num1 和 num2 num3 之和之间的差值 ...
库： （rnrs base）， （rnrs）

(- 3)⇒ -3
（- -2/3） ⇒ 2/3
（- 4 3.0） ⇒ 1.0
（- 3.25+4.25i 1/4+1/4i） ⇒ 3.0+4.0i
（- 4 3 2 1） ⇒ -2

过程： （* 数字 ...）
返回：参数数的乘积 ...
库：（rnrs base）， （rnrs）

当调用时不带任何参数，* 返回 1。

(*)⇒ 1
（* 3.4） ⇒ 3.4
（* 1 1/2） ⇒ 1/2
（* 3 4 5.5） ⇒ 66.0
（* 1+2i 3+4i） ⇒ -5+10i
（适用 *’（1 2 3 4 5）） ⇒ 120

过程： （/ num）
返回： num
的乘法逆过程： （/ num1 num2 num3 ...）
返回： 将 num1 除以 num2 num3 的乘积的结果 ...
库： （rnrs base）， （rnrs）

(/ -17)⇒ -1/17
（/ 1/2） ⇒ 2
（/ .5） ⇒ 2.0
（/ 3 4） ⇒ 3/4
（/ 3.0 4） ⇒ .75
（/ -5+10i 3+4i） ⇒ 1+2i
（/ 60 5 4 3 2） ⇒ 1/2

过程：（零？数字）
返回：如果 num 为零，则#t，#f否则
库：（rnrs base）、 （rnrs）

零？等效于 （lambda （x） （= x 0））） 。

```
(零? ⇒ #t
(零? 1) ⇒ #f
(零? (- 3.0 3.0))⇒ #t
(零? (+ 1/2 1/2))⇒ #f
(零? 0 + 0i) ⇒ #t
(零? 0.0-0.0i) ⇒ #t
```

程序：（阳性? 真实）
返回：如果实数大于零，则#t，#f否则
库：（rnrs base），（rnrs）

阳性?等效于 （lambda （x） （> x 0））） 。

```
(好评? 128)⇒ #t
(正? 0.0) ⇒ #f
(正? 1.8e-15) ⇒ #t
(正? -2/3) ⇒ #f
(正? .001-0.0i) ⇒ 异常：不是实数
```

过程：（阴性? 真实）
返回：如果实数小于零，则#t，否则#f
库：（rnrs base），（rnrs）

阴性?等效于 （lambda （x） （< x 0））） 。

```
(否定? -65)⇒ #t
(负? 0) ⇒ #f
(负? -0.0121) ⇒ #t
(负? 15/16) ⇒ #f
(负? -7.0+0.0i) ⇒ 异常：不是实数
```

程序：（甚至? int）
返回：如果 int 为偶数，则#t，否则#f
过程：（奇数? int）
返回：如果 int 为奇数，则#t，#f否则
库：（rnrs base）、（rnrs）

```
(甚至? 0)⇒ #t
(偶数? 1) ⇒ #f
(偶数? 2.0) ⇒ #t
(偶数? -120762398465) ⇒ #f
(偶数? 2.0+0.0i) ⇒ 异常：不是整数
```

```
（奇数？0） ⇒ #f
（奇数？1） ⇒ #t
（奇数？2.0） ⇒ #f
（奇数？-120762398465） ⇒ #t
（奇数？2.0+0.0i） ⇒ 异常：不是整数
```

程序：（有限？ 真实）
返回：#t如果实数是有限的，#f否则
过程：（无限？ 真实）
返回：#t如果实数是无限的，#f否则
过程：（nan？ 真实）
返回：如果 real 是 NaN，则#t，否则#f
库：（rnrs base）、 （rnrs）

```
（有限？ 2/3）⇒ #t
（无限？ 2/3） ⇒ #f
（nan？ 2/3） ⇒ #f

（有限？ 3.1415） ⇒ #t
（无限？ 3.1415） ⇒ #f
（nan？ 3.1415） ⇒ #f

（有限？ +inf.0） ⇒ #f
（无限？ -inf.0） ⇒ #t
（nan？ -inf.0） ⇒ #f

（有限？ +nan.0） ⇒ #f
（无限？ +nan.0） ⇒ #f
（nan？ +nan.0） ⇒ #t
```

过程：（商 int1 int2）
返回：int1 和 int2
过程的整数商：（余数 int1 int2）
返回：int1 的整数余数和int2
过程：（模数 int1 int2）
返回：int1 和 int2
库的整数模量：（rnrs r5rs）

余数的结果与 int1 具有相同的符号，而模的结果具有与 int2 相同的符号。

```
（商 45 6）⇒ 7
（商 6.0 2.0） ⇒ 3.0
（商 3.0 -2） ⇒ -1.0

（余数 16 4） ⇒ 0
（余数 5 2） ⇒ 1
（余数 -45.0 7） ⇒ -3.0
（余数 10.0 -3.0） ⇒ 1.0
（余数 -17 -9） ⇒ -8

（模 16 4） ⇒ 0
（模 5 2） ⇒ 1
（模 -45.0 7） ⇒ 4.0
（模 10.0 -3.0） ⇒ -2.0
（模 -17 -9） ⇒ -8
```

过程： （div x1 x2）
过程： （mod x1 x2）
过程： （div-and-mod x1 x2）
返回： 参见下面的
库： （rnrs base）， （rnrs）

如果 x1 和 x2 是精确的，则 x2 不得为零。这些过程实现数论整数除法，其中 div 运算与商相关，mod 运算与余数或模数相关，但在这两种情况下都扩展以处理实数。

（div x1 x2） 的值 nd 是整数，（mod x1 x2） 的值 xm 是实数，使得 x1 = nd • x2 + xm 和 0 ⩽ xm < |x2|.在实现无法将这些方程规定的数学结果表示为数字对象的情况下，div 和 mod 返回未指定的数字或引发具有条件类型 &实现限制的异常。

div-and-mod 过程的行为类似于定义如下。

```
（定义 （div-and-mod x1 x2） （值 （div x1 x2） （mod x1 x2））
```

也就是说，除非它在上述情况下引发异常，否则它将返回两个值：对两个参数调用 div 的结果和对两个参数调用 mod 的结果。

```
（第17 3部分）⇒ 5
（模组 17 3） ⇒ 2
（模组 -17 3） ⇒ -6
（模组 -17 3） ⇒ 1
（模组 17 -3） ⇒ -5
```

```
(模组 17 -3) ⇒ 2
(模组 -17 -3) ⇒ 6
(模组 -17 -3) ⇒ 1

(模组 17.5 3) ⇒ 5.0
  2.5
```

过程： (div0 $_{x1}$ $_{x2}$)
过程： (mod0 $_{x1}$ $_{x2}$)
过程： (div0-and-mod0 $_{x1}$ $_{x2}$)
返回： 参见下面的
库： (rnrs base)， (rnrs)

如果 $_{x1}$ 和 $_{x2}$ 是精确的，则 $_{x2}$ 不得为零。这些过程类似于 div、mod 和 div-and-mod，但对 “mod” 值的约束不同，这也会影响 “div” 值。(div0 $_{x1}$ $_{x2}$) 的值 $_{nd}$ 是整数，(mod0 $_{x1}$ $_{x2}$) 的值 $_{xm}$ 是实数，使得 x1 = nd • x2 + xm 和 $-|x2/2| \leqslant xm < |x2/2|$。在实现无法将这些方程规定的数学结果表示为数字对象的情况下，div0 和 mod0 返回未指定的数字或引发具有条件类型 &实现限制的异常。

div0 和 mod0 过程的行为类似于定义如下。

```
(定义 (div0-and-mod0 x1 x2) (值 (div0 x1 x2) (mod0 x1
x2)))
```

也就是说，除非它在上述情况下引发异常，否则它将返回两个值：对两个参数调用 div0 的结果和对两个参数调用 mod0 的结果。

```
(div0 17 3) ⇒ 6
(mod0 17 3) ⇒ -1
(div0 -17 3) ⇒ -6
(mod0 -17 3) ⇒ 1
(div0 17 -3) ⇒ -6
(mod0 17 -3) ⇒ -1
(div0 -17 -3) ⇒ 6
(mod0 -17 -3) ⇒ 1

(div0-and-mod0 17.5 3) ⇒ 6.0
  -0.5
```

过程： (截断实数)
返回： 最接近实数的整数朝向零

库：（rnrs base），（rnrs）

如果实数是无穷大或 NaN，则截断返回实数。

```
（截断 19）⇒ 19
（截断 2/3） ⇒ 0
（截断 -2/3） ⇒ 0
（截断 17.3） ⇒ 17.0
（截断 -17/2） ⇒ -8
```

过程：（底实数）
返回：最接近实数的整数指向-∞
库：（rnrs base），（rnrs）

如果实数是无穷大或 NaN，则底数返回实数。

```
（19楼）⇒ 19
（楼层 2/3） ⇒ 0
（楼层 -2/3） ⇒ -1
（楼层 17.3） ⇒ 17.0
（楼层 -17/2） ⇒ -9
```

过程：（上限实数）
返回：最接近实数的整数朝向+∞
库：（rnrs base），（rnrs）

如果实数是无穷大或 NaN，则上限返回实数。

```
（天花板 19）⇒ 19
（天花板 2/3） ⇒ 1
（天花板 -2/3） ⇒ 0
（天花板 17.3） ⇒ 18.0
（天花板 -17/2） ⇒ -8
```

过程：（舍入实数）
返回：最接近实数
库的整数：（rnrs base），（rnrs）

如果 real 正好位于两个整数之间，则返回最接近的偶数整数。如果实数是无穷大或 NaN，则 round 返回实数。

```
（第19轮）⇒ 19
（回合 2/3） ⇒ 1
```

```
(回合 -2/3) ⇒ -1
(回合 17.3) ⇒ 17.0
(回合 -17/2) ⇒ -8
(回合 2.5) ⇒ 2.0
(回合 3.5) ⇒ 4.0
```

过程: (abs real)
返回: 真实
库的绝对值: (rnrs base), (rnrs)

abs 等价于 (lambda (x) (if (< x 0) (- x) x))。abs 和 magnitude（见第 183 页）对于实际输入是相同的。

```
(腹部 1) ⇒ 1
(男量 -3/4) ⇒ 3/4
(男高 1.83) ⇒ 1.83
(男重 -0.093) ⇒ 0.093
```

过程: (最大 real1 real2 ...)
返回: real1 real2 的最大值 ...
库: (rnrs base), (rnrs)

```
(最大 4 -7 2 0 -6) ⇒ 4
(最大 1/2 3/4 4/5 5/6 6/7) ⇒ 6/7
(最大 1.5 1.3 -0.3 0.4 2.0 1.8) ⇒ 2.0
(最大 5 2.0) ⇒ 5.0
(最大 -5 -2.0) ⇒ -2.0
(let ([ls '(7 3 5 2 9 8)])
(应用最大 ls)) ⇒ 9
```

过程: (最小 real1 real2 ...)
返回: real1 real2 的最小值 ...
库: (rnrs base), (rnrs)

```
(分钟 4 -7 2 0 -6) ⇒ -7
(分钟 1/2 3/4 4/5 5/6 6/7) ⇒ 1/2
(分钟 1.5 1.3 -0.3 0.4 2.0 1.8) ⇒ -0.3
(分钟 5 2.0) ⇒ 2.0
(分钟 -5 -2.0) ⇒ -5.0
(让 ([ls '(7 3 5 2 9 8)])
(应用分钟 ls)) ⇒ 2
```

过程： （gcd int ...）
返回：其参数的最大公除数 int ...
库：（rnrs base），（rnrs）

结果始终是非负数，即忽略 -1 的因子。当调用时不带任何参数，gcd 返回 0。

```
（gcd）⇒ 0
（gcd 34）⇒ 34
（gcd 33.0 15.0）⇒ 3.0
（gcd 70 -42 28）⇒ 14
```

过程： （lcm int ...）
返回：其参数的最小公倍数 int ...
库：（rnrs base），（rnrs）

结果始终是非负数，即忽略 -1 的公共倍数。尽管 lcm 在调用时可能应返回∞，但不带任何参数，但它被定义为返回 1。如果一个或多个参数为 0，则 lcm 返回 0。

```
（长厘米）⇒ 1
（长厘米 34）⇒ 34
（长厘米 33.0 15.0）⇒ 165.0
（长厘米 70 -42 28）⇒ 420
（长厘米 17.0 0）⇒ 0.0
```

过程： （expt num1 num2）
返回：num1 提升到 num2 电源
库：（rnrs base）、（rnrs）

如果两个参数均为 0，则 expt 返回 1。

```
（例 2 10）⇒ 1024
（expt 2 -10）⇒ 1/1024
（expt 2 -10.0）⇒ 9.765625e-4
（expt -1/2 5）⇒ -1/32
（expt 3.0 3）⇒ 27.0
（expt +i 2）⇒ -1
```

过程： （不准确的数字）
返回：数字
库的不精确表示：（rnrs base），（rnrs）

如果 num 已经不精确，则返回原样。如果实现不支持 num 的不精确表示形式，则可能会引发条件类型 &实现冲突的异常。对于大小超过实现的不精确数字表示范围的输入，inexact 也可能返回 +inf.0 或 -inf.0。

```
（不准确 3）⇒ 3.0
（不精确 3.0） ⇒ 3.0
（不精确 -1/4） ⇒ -.25
（不精确 3+4i） ⇒ 3.0+4.0i
（不精确 （expt 10 20）） ⇒ 1e20
```

过程：（精确数字）
返回：数字
库的精确表示：（rnrs base）， （rnrs）

如果 num 已经是精确的，则返回它保持不变。如果实现不支持 num 的确切表示形式，则可能会引发条件类型 &实现冲突的异常。

```
（精确 3.0）⇒ 3
（精确 3） ⇒ 3
（精确 -.25） ⇒ -1/4
（精确 3.0+4.0i） ⇒ 3+4i
（精确 1e20） ⇒
100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

过程：（exact->inexact num）
返回：num
过程的不精确表示：（inexact->exact num）
返回：num
库的精确表示：（rnrs r5rs）

这些是不准确和准确的替代名称，支持与修订的5报告兼容。

过程：（合理化 real1 real2）
返回：参见下面的
库：（rnrs base）、 （rnrs）

rationalize 返回与 real1 相差不超过 real2 的最简单有理数。有理数 q1 = n1/m1 比另一个有理数 q2 = n2/m2 更简单，如果|n1|≤|n2|和|m1|≤ |平方米|和任一|n1|< |n2|或|m1|< |m2|.

```
（合理化 3/10 1/10）⇒ 1/3
（合理化 .3 1/10） ⇒ 0.3333333333333333 （
eqv? （合理化 .3 1/10） #i1/3） ⇒ #t
```

过程：（分子大鼠）
返回：大鼠
库的分子：（rnrs基），（rnrs）

如果大鼠是整数，则分子是大鼠。

```
（分子 9）⇒ 9
（分子 9.0） ⇒ 9.0
（分子 0.0） ⇒ 0.0
（分子 2/3） ⇒ 2
（分子 -9/4） ⇒ -9
（分子 -2.25） ⇒ -9.0
```

过程：（分母大鼠）
返回：大鼠
库的分母：（rnrs基），（rnrs）

如果 rat 是整数，包括零，则分母为 1。

```
（分母9）⇒ 1
（分母 9.0） ⇒ 1.0
（分母 0） ⇒ 1
（分母 0.0） ⇒ 1.0
（分母 2/3） ⇒ 3
（分母 -9/4） ⇒ 4
（分母 -2.25） ⇒ 4.0
```

过程：（实部数字）
返回：数字
库的实部组件：（rnrs base），（rnrs）

如果 num 是实数，则实部返回 num。

```
（实部 3+4i）⇒ 3
（实部 -2.3+0.7i） ⇒ -2.3
（实部 -i） ⇒ 0
（实部 17.2） ⇒ 17.2
（实部 -17/100） ⇒ -17/100
```

过程：（imag-part num）
返回：num
库的虚部：（rnrs base），（rnrs）

如果 num 是实数，则 imag-part 返回精确零。

```
（伊玛格第3部分+4i）⇒ 4
（意象部分 -2.3+0.7i） ⇒ 0.7
（意象部分 -i） ⇒ -1
（意象部分 -2.5） ⇒ 0
（意象部分 -17/100） ⇒ 0
```

过程：（make-矩形 real1 real2）
返回：具有实数组件 real1 和虚部组件 real2
库的复数：（rnrs base）、 （rnrs）

```
（矩形 -2 7）⇒ -2+7i
（矩形 2/3 -1/2） ⇒ 2/3-1/2i
（矩形 3.2 5.3） ⇒ 3.2+5.3i
```

过程：（make-polar real1 real2）
返回：一个具有幅度实数 1 和角度实数 2
库的复数：（rnrs base）、 （rnrs）

```
（制造极地 2 0）⇒ 2
（制造极性 2.0 0.0） ⇒ 2.0+0.0i
（制造极性 1.0 （asin -1.0）） ⇒ 0.0-1.0i
（eqv？（制造极地 7.2 -0.588） 7.2@-0.588） ⇒ #t
```

过程：（角度数）
返回：数字
库的极性表示的角度部分：（rnrs base），（rnrs）

结果的范围是−π（排除）到+π（包括）。

```
（角度 7.3@1.5708）⇒ 1.5708
（角度 5.2） ⇒ 0.0
```

过程：（数量级数）
返回：数字
库的大小：（rnrs base），（rnrs）

对于实际参数，量级和abs（见第178页）是相同的。复数 x + yi 的大小为 $+\sqrt{x^2+y^2}$。

```
（星等1）⇒ 1
（星等 -3/4） ⇒ 3/4
（等 1.83） ⇒ 1.83
（星等 -0.093） ⇒ 0.093
（星等 3+4i） ⇒ 5
（震级 7.25@1.5708） ⇒ 7.25
```

过程： （sqrt num）
返回： num
库的主平方根： （rnrs base）， （rnrs）

鼓励（但不是必需）实现在可行的情况下将精确输入的精确结果返回到 sqrt。

```
（16平方米）⇒ 4
（sqrt 1/4） ⇒ 1/2
（sqrt 4.84） ⇒ 2.2
（sqrt -4.84） ⇒ 0.0+2.2i
（sqrt 3+4i） ⇒ 2+1i
（sqrt -3.0-4.0i） ⇒ 1.0-2.0i
```

过程： （exact-integer-sqrt n）
返回： 参见下面的
库： （rnrs base）， （rnrs）

此过程返回两个非负精确整数 s 和 r，其中 n = s2 + r 和 n < （s + 1）^{2}。

```
（exact-integer-sqrt 0）⇒ 0
  0
（exact-integer-sqrt 9） ⇒ 3
⇒ 0
（exact-integer-sqrt 19） ⇒ 4
⇒ 3
```

过程： （exp num）
返回： e 到 num 幂
库： （rnrs base）， （rnrs）

```
（exp 0.0）⇒ 1.0
（exp 1.0） ⇒ 2.7182818284590455
（exp -.5） ⇒ 0.6065306597126334
```

过程：（对数 num）
返回：num
的自然对数 过程：（对数 num1 num2）
返回：num1
库的 base-num2 对数：（rnrs base），（rnrs）

```
（日志 1.0）⇒ 0.0
（日志 （exp 1.0）） ⇒ 1.0
（/ （日志 100） （日志 10）） ⇒ 2.0
（日志 （制造极地 （exp 2.0） 1.0）） ⇒ 2.0+1.0i

（日志 100.0 10.0） ⇒ 2.0
（日志 .125 2.0） ⇒ -3.0
```

过程：（sin num）
过程：（余弦数）
过程：（tan num）
返回：num
库的正弦、余弦或正切：（rnrs base），（rnrs）

该参数以弧度为单位指定。

```
（罪 0.0）⇒ 0.0
（cos 0.0） ⇒ 1.0
（棕褐色 0.0） ⇒ 0.0
```

过程：（asin num）
过程：（acos num）
返回：num
库的弧正弦或弧余弦：（rnrs base），（rnrs）

结果以弧度为单位。复数 z 的弧正弦和弧余弦定义如下。

$$\sin^{-1}(z) = -i\log(iz + \sqrt{1 - z^2})$$

$$\cos^{-1}(z) = \pi/2 - \sin^{-1}(z)$$

```
（定义圆周率 （* （asin 1） 2））
（= （* （acos 0） 2） pi） ⇒ #t
```

过程： （atan num）
过程： （atan real1 real2）
返回： 参见下面的
库： （rnrs base）， （rnrs）

当传递单个复参数 num（第一种形式）时，atan 返回 num 的弧切线。复数 z 的弧切线定义如下。

$$\tan^{-1}(z) = (\log(1+iz) - \log(1-iz))/(2i)$$

当传递两个实数参数（第二种形式）时，atan 等价于 （lambda （y x） （angle （make-rectangular x y））））。

```
（定义圆周率 （* （atan 1） 4））
（= （* （atan 1.0 0.0） 2） pi） ⇒ #t
```

过程： （位按- 不显出）
返回： 位不 exint
过程： （位- 和 exint ...）
返回： 位和 exint ...
过程： （位- ior exint ...）
返回： 位包含或exint ...
过程： （bitwise-xor exint ...）
返回： 按位排他性或 exint ...
库： （rnrs 算术按位）， （rnrs）

输入被视为以二的补码表示，即使它们在内部不是以这种方式表示的。

```
（按位不 0）⇒ -1
（按位-非 3） ⇒ -4

（按位和#b01101 #b00111） ⇒ #b00101
（按位 ior #b01101 #b00111） ⇒ #b01111
（按位异或#b01101 #b00111） ⇒ #b01010
```

过程： （按位-if exint1 exint2 exint3）
返回：其参数
库的按位“if”： （rnrs算术按位）， （rnrs）

输入被视为以二的补码表示，即使它们在内部不是以这种方式表示的。

对于 exint1 中设置的每个位，结果的相应位取自 exint2，对于未在 exint1 中设置的每个位，结果的相应位取自 x3。

```
（按位- 如果#b101010 #b111000 #b001100）⇒ #b101100
```

按位-if 可以定义如下：

```
（定义 bitwise-if
（lambda （exint1 exint2 exint3）
（bitwise-ior
（bitwise-and exint1 exint2）
（bitwise-and （bitwise-not exint1） exint3））））
```

过程：（按位-位计数 exint）
返回：请参阅下面的
库：（rnrs 算术按位）、（rnrs）

对于非负输入，按位计数返回在 exint 的两个补集表示形式中设置的位数。对于负输入，它返回一个负数，其大小大于在exint的二补集表示中未设置的位数1，这等效于（按位不（按位-位-计数（按位-不 exint））））。

```
（按位计数#b00000）⇒ 0
（按位位计数 #b00001） ⇒ 1
（按位位计数 #b00100） ⇒ 1
（按位位计数 #b10101） ⇒ 3

（按位位计数 -1） ⇒ -1
（按位位计数 -2） ⇒ -2
（按位位计数 -4） ⇒ -3
```

过程：（按位长度 exint）
返回：请参阅下面的
库：（rnrs 算术按位）、（rnrs）

此过程返回 exint 的最小两个补码表示形式的位数，不包括负数的符号位。对于 0，按位长度返回 0。

```
（按位长度#b00000）⇒ 0
（按位长度 #b00001） ⇒ 1
（按位长度 #b00100） ⇒ 3
（按位长度 #b00110） ⇒ 3

（按位长度 -1） ⇒ 0
（按位长度 -6） ⇒ 3
（按位长度 -9） ⇒ 4
```

过程：（按位第一位设置 exint）
返回：exint
库中设置的最低有效位的索引：（rnrs 算术按位）、（rnrs）

输入被视为好像用二的补码表示，即使它在内部不是以这种方式表示的。

如果 exint 为 0，则按位第一位集返回 -1。

```
（按位第一位集#b00000）⇒ -1
（按位第一位集 #b00001） ⇒ 0
（按位第一位集 #b01100） ⇒ 2

（位先位集 -1） ⇒ 0
（按位第一位集 -2） ⇒ 1
（按位第一位集 -3） ⇒ 0
```

过程：（按位设置? exint1 exint2）
返回：#t如果设置了 exint1 的位 exint2，则#f否则
库：（rnrs 算术按位）、（rnrs）

exint2 被视为两者对 exint1 的补集表示中的位的从零开始的索引。两者对非负数的补集表示在概念上以无限数量的零位向左扩展（朝向更多有效位），并且两者对负数的补集表示在概念上向左扩展，具有无限数量的一位。因此，精确的整数可用于表示任意大的集合，其中 0 是空集，-1 是宇宙，按位设置? 用于测试成员资格。

```
（按位设置? #b01011 0）⇒ #t
（按位位集? #b01011 2） ⇒ #f

（按位位集? -1 0） ⇒ #t
（位位集? -1 20） ⇒ #t
（位比特集? -3 1） ⇒ #f
```

```
（位位集？ 0 5000） ⇒ #f
（位对位集？ -1 5000） ⇒ #t
```

过程：（按位复制位 exint1 exint2 exint3）
返回：exint1，其中位 exint2 替换为 exint3
库：（rnrs 算术位），（rnrs）

exint2 被视为两者对 exint1 的补集表示中的位的从零开始的索引。exint3 必须为 0 或 1。此过程根据 exint3 的值有效地清除或设置指定的位。exint1 被视为在二的补码中表示，即使它在内部不是以这种方式表示的。

```
（按位复制位 #b01110 0 1）⇒ #b01111
（按位复制位#b01110 2 0） ⇒ #b01010
```

过程：（按位位字段 exint1 exint2 exint3）
返回：请参阅下面的
库：（rnrs 算术按位），（rnrs）

exint2 和 exint3 必须是非负数，并且 exint2 不能大于 exint3。此过程返回通过从 exint1 中提取从 exint2（包括）到 exint3（独占）的位序列所表示的数字。exint1 被视为在二的补码中表示，即使它在内部不是以这种方式表示的。

```
（按位位字段#b10110 0 3）⇒ #b00110
（按位位场#b10110 1 3） ⇒ #b00011
（按位位场#b10110 2 3） ⇒ #b00001
（按位位场#b10110 3 3） ⇒ #b00000
```

过程：（bitwise-copy-bit-field exint1 exint2 exint3 exint4）
返回：请参阅下面的
库：（rnrs arithmetic bitwise），（rnrs）

exint2 和 exint3 必须是非负数，并且 exint2 不能大于 exint3。此过程返回 exint1，其中从 exint2（包括）到 exint3（独占）的 n 位被 exint4 的低位 n 位替换。exint1 和 exint4 被视为在二的补码中表示，即使它们在内部不是以这种方式表示的。

```
（按位复制位字段 #b10000 0 3 #b10101）⇒ #b10101
（按位复制位字段#b10000 1 3 #b10101） ⇒ #b10010
```

```
（按位复制位字段#b10000 2 3 #b10101） ⇒ #b10100
（按位复制位字段#b10000 3 3 #b10101） ⇒ #b10000
```

过程：（位-算术-移位-右 exint1 exint2）
返回： exint1 算术通过 exint2 位
向右移过程： （位-算术-移位-左 exint1 exint2）
返回： exint1由 exint2 位
库向左移位：（rnrs 算术按位）、（rnrs）

exint2 必须是非负数。exint1 被视为在二的补码中表示，即使它在内部不是以这种方式表示的。

```
（按位算术右移 #b10000 3）⇒ #b00010
（位按算术-偏移-右 -1 1） ⇒ -1
（位-算术-移位-右 -64 3） ⇒ -8

（位-算术-偏移-左 #b00010 2） ⇒ #b01000
（位-算术-移位-左 -1 2） ⇒ -4
```

过程：（按位算术移位 exint1 exint2）
返回：请参阅下面的
库：（rnrs 算术按位）、 （rnrs）

如果 exint2 为负数，则按位算术移位返回算术移位 exint1 向右移动 - exint2 位的结果。否则，按位算术移位返回移位 exint1 由 exint2 位留下的结果。exint1 被视为在二的补码中表示，即使它在内部不是以这种方式表示的。

```
（按位算术移位#b10000 -3）⇒ #b00010
（位算术移位 -1 -1） ⇒ -1
（位算术移位 -64 -3） ⇒ -8
（位算术移位#b00010 2） ⇒ #b01000
（位算术移位 -1 2） ⇒ -4
```

因此，按位算术移位的行为就像定义如下一样。

```
（定义位-算术-移位
（lambda （exint1 exint2）
（if （< exint2 0）
（bitwise-arithmetic-shift-right exint1 （- exint2））
（bitwise-arithmetic-shift-left exint1 exint2））））
```

过程：（按位旋转位字段 exint1 exint2 exint3 exint4）
返回：请参阅下面的
库：（rnrs 算术位），（rnrs）

exint2、exint3 和 exint4 必须是非负数，并且 exint2 不能大于 exint3。此过程返回将 exint1 的位从位 exint2（含）移动到（mod exint4（- exint3 exint2））位留下的位 exint3（独占）的结果，并将位移出插入到范围底端的范围内。exint1 被视为在二的补码中表示，即使它在内部不是以这种方式表示的。

（按位旋转位场#b00011010 0 5 3）⇒ #b00010110
（按位旋转位场#b01101011 2 7 3）⇒ #b01011011

过程：（按位反向位字段 exint1 exint2 exint3）
返回：请参阅下面的
库：（rnrs 算术按位）、（rnrs）

exint2 和 exint3 必须是非负数，并且 exint2 不能大于 exint3。此过程返回将 exint1 的位从位 exint2（包括）反转到位 exint3（独占）的结果。exint1 被视为在二的补码中表示，即使它在内部不是以这种方式表示的。

（按位反向位场#b00011010 0 5）⇒ #b00001011
（按位反向位场#b01101011 2 7）⇒ #b00101111

过程：（字符串>数字字符串）
过程：（字符串>数字字符串基数）
返回：由字符串表示的数字，或#f
库：（rnrs base），（rnrs）

如果 string 是数字的有效表示形式，则返回该数字，否则返回#f。该数字以基数基数解释，基数基数必须是集合 {2，8，10，16} 中的精确整数。如果未指定，则基数默认为 10。字符串中的任何基数说明符（例如，#x）都会覆盖基数参数。

（字符串->编号“0”）⇒ 0
（字符串>编号 “3.4e3”）⇒ 3400.0
（字符串>编号 “#x#e-2e2”）⇒ -738
（字符串>编号 “#e-2e2” 16）⇒ -738

```
（字符串>编号 “#i15/16”） ⇒ 0.9375
（字符串->编号 “10” 16） ⇒ 16
```

过程：（数字>字符串数字）
过程：（数字>字符串数字基数）
过程：（数字>字符串数字基数精度）
返回：数字>字符串
库的外部表示形式：（rnrs 基数），（rnrs）

数字以基数基数表示，基数基数必须是集合 {2，8，10，16} 中的精确整数。如果未指定，则基数默认为 10。在任何情况下，生成的字符串中都不会显示基数说明符。

外部表示形式是这样的，当使用字符串>数字转换回数字时，生成的数值等效于数字。也就是说，对于所有输入：

```
（eqv? （字符串->number
（number->string num radix）
radix）
数字）
```

返回#t。如果无法做到这一点，则会引发具有条件类型和实现限制的异常。

如果提供了精度，则它必须是精确的正整数，num 必须是不精确的，基数必须为 10。在这种情况下，实部和数字的虚部（如果存在）都打印有一个显式尾数宽度 m，其中 m 是大于或等于使上述表达式为 true 的精度的最小可能值。

如果基数为 10，则在不违反上述限制的情况下，使用尽可能少的有效位数 [5] 表示 num 的不精确值。

```
（数字>字符串 3.4）⇒ “3.4”
（数字>字符串 1e2） ⇒ “100.0”
（数字>字符串 1e-23） ⇒ “1e-23”
（数字>字符串 -7/2） ⇒ “-7/2”
（数字>字符串 220/9 16） ⇒ “DC/9”
```

第 6.5 节.Fixnums

Fixnums 表示 fixnum 范围内的确切整数，该范围必须是闭合范围 $[-2^{w-1}, 2^{w-1} - 1]$，其中 w（固定数宽度）至少为 24。w 的实现特定值可以通过过程固定点宽度确定，范围的端点可以通过过程最小固定数和最大固定数确定。

仅对固定数操作的算术过程的名称以前缀“fx”开头，以将它们与通用对应项区分开来。

需要成为固定数的过程参数被命名为fx，可能带有后缀，例如$_{fx2}$。

除非另有指定，否则特定于 fixnum 的过程的数值为 fixnum。如果 fixnum 操作的值应为 fixnum，但数学结果将超出 fixnum 范围，则会引发条件类型 &实现限制的异常。

fixnum 上的位和移位运算假定 fixnum 以 2 的补码表示，即使它们在内部不是以这种方式表示的。

过程：（固定数字？obj）
返回：如果 obj 是 fixnum，则#t，#f否则
库：（rnrs arithmetic fixnums），（rnrs）

```
(固定数字？ 0) ⇒ #t
(fixnum？ -1) ⇒ #t
(fixnum？ (- (expt 2 23))) ⇒ #t
(fixnum？ (- (expt 2 23) 1)) ⇒ #t
```

过程：（最小修复）
返回：实现
过程支持的最小（最负）fixnum：（最大修复）
返回：实现
库支持的最大（最正）fixnum：（rnrs 算术修复），（rnrs）

```
(固定数字？ (- (最小固定数) 1)) ⇒ #f
(fixnum？ (最小固定数)) ⇒ #t
(fixnum？ (最大固定)) ⇒ #t
(fixnum？ (+ (最大固定数) 1)) ⇒ #f
```

过程：（fixnum-width）
返回：依赖于实现的 fixnum 宽度
库：（rnrs arithmetic fixnums），（rnrs）

如本节的引入中所述，固定数宽度确定固定数范围的大小，并且必须至少为 24。

```
(定义 w (固定宽度))
(= (minimum-fixnum) (- (expt 2 (- w 1)))) ⇒ #t
(= (greatest-fixnum) (- (expt 2 (- w 1)) 1)) ⇒ #t
(>= w 24) ⇒ #t
```

过程： (fx=? fx1 fx2 fx3 ...)
过程： (fx<? fx1 fx2 fx3 ...)
过程： (fx>? fx1 fx2 fx3 ...)
过程： (fx<=? fx1 fx2 fx3 ...)
过程： (fx>=? fx1 fx2 fx3 ...)
返回：如果关系成立，则#t，#f否则
库：(rnrs 算术固定)，(rnrs)

谓词 fx=? 返回 #t，如果它的参数相等。谓词fx<? 返回#t，如果它的参数单调递增，即每个参数都大于前面的参数，而fx>? 如果其参数单调递减，则返回#t。谓词fx<=? 如果其参数单调不递减，即每个参数不小于前面的参数，则返回#t，而fx>=? 如果其参数单调不递增，则返回#t。

```
(fx=? 0 0) ⇒ #t
(fx=? -1 1) ⇒ #f
(fx<? (minimum-fixnum) 0 (greatest-fixnum)) ⇒ #t
(let ([x 3]) (fx<=? 0 x 9)) ⇒ #t
(fx>? 5 4 3 2 1) ⇒ #t
(fx<=? 1 3 2) ⇒ #f
(fx>=? 0 0 (minimum-fixnum)) ⇒ #t
```

过程： (fxzero? fx)
返回：如果fx为零，#t，#f否则
过程： (fxpositive? fx)
返回：如果fx大于零，#t，#f否则
过程： (fxnegative? fx)
返回：如果 fx 小于零，则#t，#f否则
库：(rnrs 算术固定)、(rnrs)

fxzero? 等价于 (lambda (x) (fx=? x 0))， fxpositive? 等价于 (lambda (x) (fx>? x 0)) 和 fxnegative? to (lambda (x) (fx<? x 0)))。

```
(fxzero? 0) ⇒ #t
(fxzero? 1) ⇒ #f

(fxpositive? 128) ⇒ #t
(fxpositive? 0) ⇒ #f
(fxpositive? -1) ⇒ #f

(fxnegative? -65) ⇒ #t
(fxnegative? 0) ⇒ #f
(fxnegative? 1) ⇒ #f
```

过程: (fxeven? fx)
返回: #t如果fx是偶数，#f否则
过程: (fxodd? fx)
返回: 如果 fx 为奇数，则#t，#f否则
库: (rnrs 算术 fixnums)、(rnrs)

```
(fxeven? 0) ⇒ #t
(fxeven? 1) ⇒ #f
(fxeven? -1) ⇒ #f
(fxeven? -10) ⇒ #t

(fxodd? 0) ⇒ #f
(fxodd? 1) ⇒ #t
(fxodd? -1) ⇒ #t
(fxodd? -10) ⇒ #f
```

程序: (fxmin fx1 fx2 ...)
返回: 最小值 fx1 fx2 ...
过程: (fxmax fx1 fx2 ...)
返回: fx1 fx2 的最大值 ...
库: (rnrs arithmetic fixnums), (rnrs)

```
(fxmin 4 -7 2 0 -6) ⇒ -7

(let ([ls '(7 3 5 2 9 8)])
(apply fxmin ls)) ⇒ 2

(fxmax 4 -7 2 0 -6) ⇒ 4

(let ([ls '(7 3 5 2 9 8)])
(apply fxmax ls)) ⇒ 9
```

过程： (fx+ fx1 fx2)
返回： fx1 和 fx2
库的总和： (rnrs arithmetic fixnums)， (rnrs)

```
(fx+ -3 4) ⇒ 1 个
```

过程： (fx- fx)
返回： fx
过程的加法逆： (fx- fx1 fx2)
返回： fx1 和 fx2
库之间的差值： (rnrs 算术固定数)， (rnrs)

```
(fx- 3) ⇒ -3
(fx- -3 4) ⇒ -7
```

过程： (fx* fx1 fx2)
返回： fx1 和 fx2
库的乘积：(rnrs 算术修正)、(rnrs)

```
(汇率* -3 4) ⇒ -12
```

过程： (fxdiv fx1 fx2)
过程： (fxmod fx1 fx2)
过程： (fxdiv-and-mod fx1 fx2)
返回： 见下面的
库： (rnrs arithmetic fixnums)， (rnrs)

fx2 不能为零。这些是通用 div、mod 和 div-and-mod 的特定于 fixnum 的版本。

```
(fxdiv 17 3) ⇒ 5
(fxmod 17 3) ⇒ 2
(fxdiv -17 3) ⇒ -6
(fxmod -17 3) ⇒ 1
(fxdiv 17 -3) ⇒ -5
(fxmod 17 -3) ⇒ 2
(fxdiv -17 -3) ⇒ 6
(fxmod -17 -3) ⇒ 1

(fxdiv-and-mod 17 3) ⇒ 5
  2
```

过程： (fxdiv0 fx1 fx2)
过程： (fxmod0 fx1 fx2)
过程： (fxdiv0-and-mod0 fx1 fx2)
返回： 见下面的
库： (rnrs arithmetic fixnums)， (rnrs)

fx2 不能为零。这些是通用 div0、mod0 和 div0 和 mod0 的特定于 fixnum 的版本。

```
(fxdiv0 17 3) ⇒ 6
(fxmod0 17 3) ⇒ -1
(fxdiv0 -17 3) ⇒ -6
(fxmod0 -17 3) ⇒ 1
(fxdiv0 17 -3) ⇒ -6
(fxmod0 17 -3) ⇒ -1
(fxdiv0 -17 -3) ⇒ 6
(fxmod0 -17 -3) ⇒ 1

(fxdiv0-and-mod0 17 3) ⇒ 6
  -1
```

程序： (fx+/carry fx1 fx2 fx3)
程序： (fx-/carry fx1 fx2 fx3)
程序： (fx*/carry fx1 fx2 fx3)
退货： 见下文
库： (rnrs arithmetic fixnums)， (rnrs)

当普通的 fixnum 加法、减法或乘法运算溢出时，将引发异常。相反，这些替代程序返回携带，并允许携带传播到下一个操作。它们可用于实现多精度算术的可移植代码。

这些过程返回以下计算的两个固定值。对于外汇+/携带：

```
(let* ([s (+ fx1 fx2 fx3)]
[s0 (mod0 s (expt 2 (fixnum-width)))]
[s1 (div0 s (expt 2 (fixnum-width)))])
(值 s0 s1))
```

对于外汇/携带：

```
(let* ([d (- fx1 fx2 fx3)]
[d0 (mod0 d (expt 2 (fixnum-width)))]
```

```
[d1 (div0 d (expt 2 (fixnum-width)))])
(值 d0 d1))
```

和对于外汇*/携带：

```
(let* ([s (+ (* fx1 fx2) fx3)]
[s0 (mod0 s (expt 2 (fixnum-width)))]
[s1 (div0 s (expt 2 (fixnum-width)))])
(值 s0 s1))
```

过程： (fxnot fx)
返回： 按位不 fx
过程： (fxand fx ...)
返回： 按位和 fx ...
过程： (fxior fx ...)
返回： 按位包含或 fx ...
过程： (fxxor fx ...)
返回： 按位排他或 fx ...
库： (rnrs 算术 fixnums)， (rnrs)

```
(fxnot 0) ⇒ -1
(fxnot 3) ⇒ -4

(fxand #b01101 #b00111) ⇒ #b00101
(fxior #b01101 #b00111) ⇒ #b01111
(fxxor #b01101 #b00111) ⇒ #b01010
```

过程： (fxif fx1 fx2 fx3)
返回：其参数
库的按位“if”： (rnrs arithmetic fixnums)， (rnrs)

对于 fx1 中设置的每个位，结果的相应位取自 fx2，对于未在 fx1 中设置的每个位，结果的相应位取自 x3。

```
(fxif #b101010 #b111000 #b001100) ⇒ #b101100
```

fxif 可以定义如下：

```
(define fxif
(lambda (fx1 fx2 fx3)
(fxior (fxand fx1 fx2)
(fxand (fxnot fx1) fx3))))
```

过程：（fxbit-count fx）
返回：参见下面的
库：（rnrs arithmetic fixnums），（rnrs）

对于非负输入，fxbit-count 返回在 fx 的两个补码表示形式中设置的位数。对于负输入，它返回一个负数，其大小大于未在fx中设置的位数1，这等效于（fxnot（fxbit-count（fxnot fx））））。

```
（#b00000比特比特计数）⇒ 0
（fxbit-count #b00001） ⇒ 1
（fxbit-count #b00100） ⇒ 1
（fxbit-count #b10101） ⇒ 3

（fxbit-count -1） ⇒ -1
（fxbit-count -2） ⇒ -2
（fxbit-count -4） ⇒ -3
```

过程：（fxlength fx）
返回：请参阅下面的
库：（rnrs arithmetic fixnums），（rnrs）

此过程返回 fx 的最小两个补码表示形式的位数，不包括负数的符号位。对于 0，fxlength 返回 0。

```
（特长#b00000）⇒ 0
（fxlength #b00001） ⇒ 1
（fxlength #b00100） ⇒ 3
（fxlength #b00110） ⇒ 3

（fxlength -1） ⇒ 0
（fxlength -6） ⇒ 3
（fxlength -9） ⇒ 4
```

过程：（fxfirst-bit-set fx）
返回：在fx
库中设置的最低有效位的索引：（rnrs arithmetic fixnums），（rnrs）

如果 fx 为 0，则 fxfirst 位集返回 -1。

```
（fxfirst-bit-set #b00000）⇒ -1
（fxfirst-bit-set #b00001） ⇒ 0
（fxfirst-bit-set #b01100） ⇒ 2
```

```
(fxfirst-bit-set -1) ⇒ 0
(fxfirst-bit-set -2) ⇒ 1
(fxfirst-bit-set -3) ⇒ 0
```

过程：(fxbit-set? fx1 fx2)
返回：如果设置了 fx1 的 bit fx2，则#t，否则#f
库：(rnrs 算术修复)、(rnrs)

fx2 必须是非负数。它被取为两者对 fx1 的补集表示中的位的从零开始的索引，符号位几乎复制到左侧的无限多个位置。

```
(fxbit-set? #b01011 0) ⇒ #t
(fxbit-set? #b01011 2) ⇒ #f

(fxbit-set? -1 0) ⇒ #t
(fxbit-set? -1 20) ⇒ #t
(fxbit-set? -3 1) ⇒ #f
(fxbit-set? 0 (- (fixnum-width) 1)) ⇒ #f
(fxbit-set? -1 (- (fixnum-width) 1)) ⇒ #t
```

过程：(fxcopy-bit fx1 fx2 fx3)
返回：fx1 with bit fx2 替换为 fx3
库：(rnrs arithmetic fixnums)，(rnrs)

fx2 必须是非负数且小于 (- (固定数字宽度) 1) 的值。fx3 必须为 0 或 1。此过程根据 fx3 的值有效地清除或设置指定的位。

```
(fxcopy-bit #b01110 0 1) ⇒ #b01111
(fxcopy-bit #b01110 2 0) ⇒ #b01010
```

过程：(fxbit-field fx1 fx2 fx3)
返回：参见下面的
库：(rnrs arithmetic fixnums)，(rnrs)

fx2 和 fx3 必须是非负数且小于 (固定数字宽度) 的值，并且 fx2 不得大于 fx3。此过程返回通过从 fx1 中提取从 fx2（含）到 fx3（独占）的位序列所表示的数字。

```
(fxbit-field #b10110 0 3) ⇒ #b00110
(fxbit-field #b10110 1 3) ⇒ #b00011
```

```
(fxbit-field #b10110 2 3) ⇒ #b00001
(fxbit-field #b10110 3 3) ⇒ #b00000
```

过程： (fxcopy-bit-field $fx1$ $fx2$ $fx3$ $fx4$)
返回： 参见下面的
库： (rnrs arithmetic fixnums)， (rnrs)

$fx2$ 和 $fx3$ 必须是非负数且小于 (固定数字宽度) 的值，并且 $fx2$ 不得大于 $fx3$。此过程返回 $fx1$，其中从 $fx2$（含）到 $fx3$（独占）的 n 位替换为 $x4$ 的低阶 n 位。

```
(fxcopy-bit-field #b10000 0 3 #b10101) ⇒ #b10101
(fxcopy-bit-field #b10000 1 3 #b10101) ⇒ #b10010
(fxcopy-bit-field #b10000 2 3 #b10101) ⇒ #b10100
(fxcopy-bit-field #b10000 3 3 #b10101) ⇒ #b10000
```

过程： (fxarithmetic-shift-right $fx1$ $fx2$)
返回： $fx1$ 算术通过 $fx2$ 位
向右移动 过程： (fxarithmetic-shift-left $fx1$ $fx2$)
返回： $fx1$由 $fx2$ 位
库向左移动：(rnrs arithmetic fixnums)、 (rnrs)

$fx2$ 必须是非负数且小于 (固定数字宽度) 的值。

```
(fxarithmetic-shift-right #b10000 3) ⇒ #b00010
(fxarithmetic-shift-right -1 1) ⇒ -1
(fxarithmetic-shift-right -64 3) ⇒ -8

(fxarithmetic-shift-left #b00010 2) ⇒ #b01000
(fxarithmetic-shift-left -1 2) ⇒ -4
```

过程： (fxarithmetic-shift $fx1$ $fx2$)
返回： 参见下面的
库： (rnrs arithmetic fixnums)， (rnrs)

$fx2$ 的绝对值必须小于 (固定宽度) 的值。如果 $fx2$ 为负，则 fxarithmetic-shift 返回算术上将 $fx1$ 向右移动 $fx2$ 位的结果。否则，fxarithmetic-shift 将返回 $fx1$ 由 $fx2$ 位左移的结果。

```
(fxarithmetic-shift #b10000 -3) ⇒ #b00010
(fxarithmetic-shift -1 -1) ⇒ -1
```

```
(fxarithmetic-shift -64 -3) ⇒ -8
(fxarithmetic-shift #b00010 2) ⇒ #b01000
(fxarithmetic-shift -1 2) ⇒ -4
```

因此，fxarithmetic-shift 的行为就像定义如下一样。

```
(定义fxarithmetic-shift
(lambda (fx1 fx2)
(if (fx<? fx2 0)
(fxarithmetic-shift-right fx1 (fx- fx2))
(fxarithmetic-shift-left fx1 fx2))))
```

过程： (fxrotate-bit-field fx1 fx2 fx3 fx4)
返回： 参见下面的
库： (rnrs arithmetic fixnums)， (rnrs)

fx2、fx3 和 fx4 必须是非负数且小于 (fixnum-width) 的值，fx2 不得大于 fx3，并且 fx4 不得大于 fx3 和 fx2 之间的差值。

此过程返回将 fx1 的位从 fx2（含）移动到 fx4 位留下的位 fx3（独占）的结果，并将位移出插入到范围底端的范围之外。

```
(fxrotate-bit-field #b00011010 0 5 3) ⇒ #b00010110
(fxrotate-bit-field #b01101011 2 7 3) ⇒ #b01011011
```

过程： (fxreverse-bit-field fx1 fx2 fx3)
返回： 参见下面的
库： (rnrs arithmetic fixnums)， (rnrs)

fx2 和 fx3 必须是非负数且小于 (固定数字宽度) 的值，并且 fx2 不得大于 fx3。此过程返回将 fx1 的位从位 fx2（包括）反转为位 fx3（独占）的结果。

```
(fxreverse-bit-field #b00011010 0 5) ⇒ #b00001011
(fxreverse-bit-field #b01101011 2 7) ⇒ #b00101111
```

第 6.6 节. 弗洛姆

Flonums代表不精确的实数。实现需要将任何不精确的实数表示为flonum，其词法语法不包含竖线，也不包含除 e 以外的指数标记，但不需要将任何其他不精确的实数表示为 flonum。

实现通常对 flonum 使用 IEEE 双精度浮点表示法，但实现不需要这样做，甚至不需要使用任何类型的浮点表示，尽管名称为“flonum”。

本节介绍在浮选体上的操作。特定于 Flonum 的过程名称以前缀“fl”开头，以将其与通用对应项区分开来。

需要成为 flonum 的过程参数被命名为 fl，可能带有后缀，例如 fl2。除非另有指定，否则特定于 flonum 的过程的数值为 flonum。

程序： （氟？ obj）
返回：如果 obj 是 flonum，则#t，否则#f
库：（rnrs 算术 flonums）、 （rnrs）

```
（弗洛姆？ 0）⇒ #f
（flonum？ 3/4） ⇒ #f
（flonum？ 3.5） ⇒ #t
（flonum？ .02） ⇒ #t
（flonum？ 1e10） ⇒ #t
（flonum？ 3.0+0.0i） ⇒ #f
```

过程： （fl=？ fl1 fl2 fl3 ...）
程序： （fl<？ fl1 fl2 fl3 ...）
程序： （fl>？ fl1 fl2 fl3 ...）
过程： （fl<=？ fl1 fl2 fl3 ...）
过程： （fl>=？ fl1 fl2 fl3 ...）
返回：如果关系成立，则#t，#f否则
库：（rnrs 算术 flonums）、 （rnrs）

谓词 fl=？ 返回 #t，如果它的参数相等。谓词 fl<？ 返回 #t，如果它的参数单调递增，即每个参数都大于前面的参数，而 fl>？ 返回#t，如果它的参数单调递减。谓词 fl<=？ 返回 #t 如果它的参数是单调非递减的，即每个参数不小于前面的参数，而 fl>=？ 返回#t如果它的参数是单调非递增的。当仅传递一个参数时，这些谓词中的每一个都返回#t。

涉及 NaNs 的比较总是返回#f。

```
（fl=？ 0.0 0.0）⇒ #t
（fl<？ -1.0 0.0 1.0） ⇒ #t
```

```
(fl>? -1.0 0.0 1.0) ⇒ #f
(fl<=? 0.0 3.0 3.0) ⇒ #t
(fl>=? 4.0 3.0 3.0) ⇒ #t
(fl<? 7.0 +inf.0) ⇒ #t
(fl=? +nan.0 0.0) ⇒ #f
(fl=? +nan.0 +nan.0) ⇒ #f
(fl<? +nan.0 +nan.0) ⇒ #f
(fl<=? +nan.0 +inf.0) ⇒ #f
(fl>=? +nan.0 +inf.0)⇒ #f
```

程序： (flzero? fl)
返回：如果 fl 为零，#t，#f否则
过程： (flpositive? fl)
返回：如果 fl 大于零，#t，#f否则
过程： (flnegative? fl)
返回：如果 fl 小于零，则#t，否则#f
库：(rnrs 算术 flonums)、(rnrs)

flzero? 等价于 (lambda (x) (fl=? x 0.0))， flpositive? 等价于 (lambda (x) (fl>? x 0.0))和 flnegative? to (lambda (x) (fl<? x 0.0))。

即使 flonum 表示将 -0.0 与 +0.0 区分开来，-0.0 也被视为零和非负数。

```
(flzero? 0.0)⇒ #t
(flzero? 1.0) ⇒ #f

(flpositive? 128.0) ⇒ #t
(flpositive? 0.0) ⇒ #f
(flpositive? -1.0) ⇒ #f

(flnegative? -65.0) ⇒ #t
(flnegative? 0.0) ⇒ #f
(flnegative? 1.0) ⇒ #f

(flzero? -0.0) ⇒ #t
(flnegative? -0.0) ⇒ #f

(flnegative? +nan.0)⇒ #f
(flzero? +nan.0) ⇒ #f
(flpositive? +nan.0) ⇒ #f
```

```
(flnegative? +inf.0) ⇒ #f
(flnegative? -inf.0) ⇒ #t
```

程序：（flinteger? fl）
返回：如果 fl 是整数，则#t，否则#f
库：（rnrs 算术 flonums）、（rnrs）

```
(flinteger? 0.0) ⇒ #t
(flinteger? -17.0) ⇒ #t
(flinteger? +nan.0) ⇒ #f
(flinteger? +inf.0) ⇒ #f
```

程序：（不定? fl）
返回：#t如果 fl 是有限的，#f否则
过程：（flinfinite? fl）
返回：如果 fl 是无穷大，#t，否则#f
过程：（flnan? fl）
返回：如果 fl 是 NaN，则#t，#f否则
库：（rnrs 算术 flonums）、（rnrs）

```
(flfinite? 3.1415) ⇒ #t
(flinfinite? 3.1415) ⇒ #f
(flnan? 3.1415) ⇒ #f

(flfinite? +inf.0) ⇒ #f
(flinfinite? -inf.0) ⇒ #t
(flnan? -inf.0) ⇒ #f

(flfinite? +nan.0) ⇒ #f
(flinfinite? +nan.0) ⇒ #f
(flnan? +nan.0) ⇒ #t
```

程序：（fleven? fl-int）
返回：#t fl-int 是否为偶数，#f否则
程序：（flodd? fl-int）
返回：如果 fl-int 是奇数，则#t，否则#f
库：（rnrs 算术 flonums）、（rnrs）

fl-int 必须是整数值的 flonum。

```
(0.0) ⇒ #t
(fleven? 1.0) ⇒ #f
(fleven? -1.0) ⇒ #f
```

```
(fleven? -10.0) ⇒ #t

(flodd? 0.0) ⇒ #f
(flodd? 1.0) ⇒ #t
(flodd? -1.0) ⇒ #t
(flodd? -10.0) ⇒ #f
```

程序: (flmin fl1 fl2 ...)
返回: fl1 fl2 的最小值 ...
过程: (flmax fl1 fl2 ...)
返回: fl1 fl2 的最大值 ...
库: (rnrs arithmetic flonums), (rnrs)

```
(flmin 4.2 -7.5 2.0 0.0 -6.4) ⇒ -7.5

(let ([ls '(7.1 3.5 5.0 2.6 2.6 8.0)])
(apply flmin ls)) ⇒ 2.6

(flmax 4.2 -7.5 2.0 0.0 -6.4) ⇒ 4.2

(let ([ls '(7.1 3.5 5.0 2.6 2.6 8.0)])
(apply flmax ls)) ⇒ 8.0
```

过程: (fl+ fl ...)
返回: 参数 fl ...
库的总和: (rnrs 算术 flonums), (rnrs)

当调用时不带任何参数，fl+ 返回 0.0。

```
(fl+) ⇒ 0.0
(fl+ 1.0 2.5) ⇒ 3.25
(fl+ 3.0 4.25 5.0) ⇒ 12.25
(apply fl+ '(1.0 2.0 3.0 4.0 5.0)) ⇒ 15.0
```

过程: (fl- fl)
返回: fl
过程的加法逆: (fl- fl1 fl2 fl3 ...)
返回: fl1 和 fl2 fl3 之和之差 ...
库: (rnrs arithmetic flonums), (rnrs)

对于 flonums 的 IEEE 浮点表示，单参数 fl- 等价于

```
(lambda (x) (fl* -1.0 x))
```

或

```
(lambda (x) (fl- -0.0 x))
```

但不是

```
(lambda (x) (fl- 0.0 x))
```

因为后者返回 0.0 而不是 -0.0 表示 0.0。

```
(fl- 0.0) ⇒ -0.0
(fl- 3.0) ⇒ -3.0
(fl- 4.0 3.0) ⇒ 1.0
(fl- 4.0 3.0 2.0 1.0) ⇒ -2.0
```

程序: (fl* fl ...)
返回: 参数的乘积 fl ...
库: (rnrs 算术 flonums), (rnrs)

当调用时不带任何参数，fl* 返回 1.0。

```
(fl*) ⇒ 1.0
(fl* 1.5 2.5) ⇒ 3.75
(fl* 3.0 -4.0 5.0) ⇒ -60.0
(应用 fl* '(1.0 -2.0 3.0 -4.0 5.0)) ⇒ 120.0
```

过程: (fl/ fl)
返回: fl
过程的乘法逆: (fl/ fl1 fl2 fl3 ...)
返回: 将 fl1 除以 fl2 fl3 的乘积的结果 ...
库: (rnrs arithmetic flonums), (rnrs)

```
(fl/ -4.0) ⇒ -0.25
(fl/ 8.0 -2.0) ⇒ -4.0
(fl/ -9.0 2.0) ⇒ -4.5
(fl/ 60.0 5.0 3.0 2.0) 2.0) ⇒ 2.0
```

过程: (fldiv fl1 fl2)
过程: (flmod fl1 fl2)
过程: (fldiv-and-mod fl1 fl2)

返回： 见下面的
库： （rnrs 算术 flonums）， （rnrs）

这些是通用 div、mod 和 div-and-mod 的特定于 flonum 的版本。

```
(fldiv 17.0 3.0) ⇒ 5.0
(flmod 17.0 3.0) ⇒ 2.0
(fldiv -17.0 3.0) ⇒ -6.0
(flmod -17.0 3.0) ⇒ 1.0
(fldiv 17.0 -3.0) ⇒ -5.0
(flmod 17.0 -3.0) ⇒ 2.0
(fldiv -17.0 -3.0) ⇒ 6.0
(flmod -17.0 -3.0) ⇒ 1.0

(fldiv-and-mod 17.5 3.75) ⇒ 4.0
  2.5
```

过程： (fldiv0 fl1 fl2)
过程： (flmod0 fl1 fl2)
过程： (fldiv0-and-mod0 fl1 fl2)
返回： 见下面的
库： （rnrs 算术 flonums）， （rnrs）

这些是通用 div0、mod0 和 div0-and-mod0 的特定于 flonum 的版本。

```
(fldiv0 17.0 3.0) ⇒ 6.0
(flmod0 17.0 3.0) ⇒ -1.0
(fldiv0 -17.0 3.0) ⇒ -6.0
(flmod0 -17.0 3.0) ⇒ 1.0
(fldiv0 17.0 -3.0) ⇒ -6.0
(flmod0 17.0 -3.0) ⇒ -1.0
(fldiv0 -17.0 -3.0) ⇒ 6.0
(flmod0 -17.0 -3.0) ⇒ 1.0

(fldiv0-and-mod0 17.5 3.75) ⇒ 5.0
  -1.25
```

过程： (flround fl)
返回： 最接近 fl
过程的整数： (fltruncate fl)
返回： 最接近 fl 的整数 接近零
过程： (flfloor fl)
返回： 最接近 fl 的整数朝向−∞

过程：（flceiling fl）
返回：最接近 fl 到+∞
库的整数：（rnrs 算术 flonums），（rnrs）

如果 fl 是整数、NaN 或无穷大，则每个过程都返回 fl。如果 fl 正好位于两个整数之间，则 flround 将返回最接近的偶数。

```
(17.3) ⇒ 17.0
(flround -17.3) ⇒ -17.0
(flround 2.5) ⇒ 2.0
(flround 3.5) ⇒ 4.0

(fltruncate 17.3) ⇒ 17.0
(fltruncate -17.3) ⇒ -17.0

(flfloffer 17.3) ⇒ 17.0
(flflfloor -17.3) ⇒ -18.0

(flceiling ⇒
-17.3) ⇒ -17.0
```

过程：（flnumerator fl）
返回：fl
过程的分子：（fldenominator fl）
返回：fl
库的分母：（rnrs 算术 flonums），（rnrs）

如果 fl 是整数（包括 0.0 或无穷大），则分子为 fl，分母为 1.0。

```
(膨胀机 -9.0) ⇒ -9.0
(fldenominator -9.0) ⇒ 1.0
(fldenominator 0.0) ⇒ 0.0
(fldenominator 0.0) ⇒ 1.0
(flnumerator -inf.0) ⇒ -inf.0
(fldenominator -inf.0) ⇒ 1.0
```

以下适用于 IEEE 浮点数，但不一定是其他 flonum 表示形式。

```
(压数器 3.5) ⇒ 7.0
(膨胀机 3.5) ⇒ 2.0
```

过程： (flabs fl)
返回： fl
库的绝对值： (rnrs 算术 flonums)， (rnrs)

(3.2) ⇒ 3.2
(鳞片 -2e-20) ⇒ 2e-20

过程： (flexp fl)
返回： e 到 fl 功率
过程： (fllog fl)
返回： fl
过程的自然对数： (fllog fl1 fl2)
返回： 基数-fl1
库的 fl2 对数： (rnrs arithmetic flonums)， (rnrs)

(flexp 0.0) ⇒ 1.0
(flexp 1.0) ⇒ 2.7182818284590455

(fllog 1.0) ⇒ 0.0
(fllog (exp 1.0)) ⇒ 1.0
(fl/ (fllog 100.0) (fllog 10.0)) ⇒ 2.0

(fllog 100.0 10.0) ⇒ 2.0
(fllog .125 2.0) ⇒ -3.0

过程： (flsin fl)
返回： fl
的正弦 过程： (flcos fl)
返回： fl
过程的余弦： (fltan fl)
返回： fl
库的切线： (rnrs arithmetic flonums)， (rnrs)

过程： (flasin fl)
返回： fl
的弧正弦 过程： (flacos fl)
返回： fl
的 arc 余弦 过程： (flatan fl)
返回： fl
的弧正切线过程： (flatan fl1 fl2)

返回： fl1/fl2
库的弧切线： （rnrs 算术 flonums）， （rnrs）

过程： （flsqrt fl）
返回： fl
库的主平方根： （rnrs 算术 flonums）， （rnrs）

返回 fl 的主平方根。-0.0 的平方根应为 -0.0。其他负数的结果可能是NaN或其他一些未指定的fluan。

```
（flsqrt 4.0）⇒ 2.0
（flsqrt 0.0） ⇒ 0.0
（flsqrt -0.0） ⇒ -0.0
```

过程： （flexpt fl1 fl2）
返回： fl1 提升到 fl2 幂
库： （rnrs 算术 flonums）， （rnrs）

如果 fl1 为负且 fl2 不是整数，则结果可能是 NaN 或其他一些未指定的 flonum。如果 fl1 和 fl2 均为零，则结果为 1.0。如果 fl1 为零，fl2 为正，则结果为零。在其他情况下，fl1 为零，结果可能是 NaN 或其他一些未指定的 flonum。

```
（flexpt 3.0 2.0）⇒ 9.0
（柔性 0.0 +inf.0） ⇒ 0.0
```

过程： （fixnum->flonum fx）
返回： 最接近 fx
过程的 flonum 表示： （real->flonum real）
返回： 最接近真实
库的 flonum 表示： （rnrs 算术 flonums）， （rnrs）

fixnum->flonum是inexact的受限变体。实>虚无是当输入为精确实数时，不精确的受限变体;当它是一个不精确的非浮选实数时，它将不精确的非浮选实数掩盖到最接近的浮球中。

```
（固定数->氟烷 0）⇒ 0.0
（固定数->期 13） ⇒ 13.0

（实>色 -1/2） ⇒ -0.5
（实>色 1s3） ⇒ 1000.0
```

第 6.7 节. 字符

字符是表示字母、数字、特殊符号（如 $ 或 -）以及某些非图形控制字符（如空格和换行符）的原子对象。字符以 #\ 前缀书写。对于大多数字符，前缀后跟字符本身。例如，字母 A 的书面字符表示形式是 #\A。换行符、空格和制表符也可以以这种方式编写，但它们可以更清楚地写为 #\换行符、#\空格和 #\tab。还支持其他字符名称，如第 457 页上字符对象的语法所定义的那样。任何 Unicode 字符都可以使用语法 #\xn 编写，其中 n 由一个或多个十六进制数字组成，并表示有效的 Unicode 标量值。

本节介绍主要处理字符的操作。另请参阅以下有关字符串的部分和有关输入和输出的第 7 章，了解与字符相关的其他操作。

过程： （字符=? char1 char2 char3 ...）
过程： （字符<? char1 char2 char3 ...）
过程： （字符>? char1 char2 char3 ...）
过程： （字符<=? char1 char2 char3 ...）
过程： （字符>=? char1 char2 char3 ...）
返回：如果关系成立，则#t，#f否则
库： （rnrs base）， （rnrs）

这些谓词的行为方式与数字谓词 =、<、>、<= 和 >= 类似。 例如，当 char=? 的参数是等效字符时，char=? 返回 #t;当其参数是单调递增的字符（Unicode 标量）值时，char<? 返回#t。

```
(字符>? #\a #\b) ⇒ #f
(char<? #\a #\b) ⇒ #t
(char<? #\a #\b #\c) ⇒ #t
(let ([c #\r])
(char<=? #\a c #\z)) ⇒ #t
(char<=? #\Z #\W) ⇒ #f
(char=? #\+ #\+) ⇒ #t
```

过程： （字符 ci=? char1 char2 char3 ...）
过程： （字符 ci<? char1 char2 char3 ...）
程序： （字符 ci>? char1 char2 char3 ...）
过程： （char-ci<=? char1 char2 char3 ...）
过程： （char-ci>=? char1 char2 char3 ...）

返回：如果关系成立，则#t，#f否则
库：（rnrs unicode），（rnrs）

这些谓词与谓词 char=?、char<?、char>?、char<=? 和 char>=? 相同，只是它们不区分大小写，即比较其参数的大小写折叠版本。例如，char=? 将 #\a 和 #\A 视为非重复值;char-ci=? 不。

```
(char-ci<? #\a #\B) ⇒ #t
(char-ci=? #\W #\w) ⇒ #t
(char-ci=? #\= #\+) ⇒ #f
(let ([c #\R])
(list (char<=? #\a c #\z)
(char-ci<=? #\a c #\z))) ⇒ (#f #t)
```

过程：（字符字母? 字符）
返回：如果 char 是字母，#t，#f否则
过程：（char-numeric? 字符）
返回：如果 char 是数字，#t，#f否则
过程：（char-whitespace? 字符）
返回：如果 char 是空格，则#t，否则#f
库：（rnrs unicode）、（rnrs）

如果字符具有 Unicode “Alphabetic” 属性，则该字符是按字母顺序排列的;如果字符具有 Unicode “Numeric”属性，则为数字字符;如果具有 Unicode “White_Space”属性，则该字符为空格。

```
(字符字母? #\a) ⇒ #t
(char-alphabetic? #\T) ⇒ #t
(char-alphabetic? #\8) ⇒ #f
(char-alphabetic? #\$) ⇒ #f

(char-numeric? #\7) ⇒ #t
(char-numeric? #\2) ⇒ #t
(char-numeric? #\X) ⇒ #f
(char-numeric? #\space) ⇒ #f

(char-whitespace? #\space) ⇒ #t
(char-whitespace? #\newline) ⇒ #t
(空格? #\Z) ⇒ #f
```

过程：（小写字符? 字符）
返回：如果 char 是小写，#t，#f否则

过程：（char-大写？ 字符）
返回：如果 char 是大写，#t，#f否则
过程：（char-title-case？ 字符）
返回：如果 char 是标题大小写，则#t，#f否则
库：（rnrs unicode）、（rnrs）

如果字符具有 Unicode“大写”属性，则为大写;如果字符具有“小写”属性，则为小写;如果字符位于 Lt 常规类别中，则为标题大小写。

```
(小写字符？ #\r) ⇒ #t
(char-small-case？ #\R) ⇒ #f

(char-maximum-case？ #\r) ⇒ #f
(char-upper-case？ #\R) ⇒ #t

(char-title-case？ #\I) ⇒ #f
(char-title-case？ #\x01C5) ⇒ #t
```

过程：（字符-通用-类别 char）
返回：表示字符
库的 Unicode 常规类别的符号：（rnrs unicode）、（rnrs）

返回值是符号之一 Lu， Ll， Lt， Lm， Lo， Mn， Mc， Me， Nd， Nl， No， Ps， Pe， Pi， Pf， Pd， Pc， Po， Sc， Sm， Sk， So， Zs， Zp， Zl， Cc， Cf， Cs、Co 或 Cn。

```
(字符-一般-类别 #\a) ⇒ Ll
(char-general-category #\space) ⇒ Zs
(char-general-category #\x10FFFF) ⇒ Cn
```

过程：（char-upcase char）
返回：char
库的大写字符对应物：（rnrs unicode），（rnrs）

如果 char 是小写或标题字符，并且具有单个大写对应物，则 char-upcase 将返回大写对应物。否则，字符大写将返回字符。

```
(字符大写 #\g) ⇒ #\G
(字符大写 #\G) ⇒ #\G
(字符大写 #\7) ⇒ #\7
(字符大写 #\ς) ⇒ #\Σ
```

过程：（字符小写字符）
返回：字符
库的小写字符等效项：（rnrs unicode），（rnrs）

如果 char 是大写或标题字符，并且具有单个小写对应物，则 char-downcase 将返回小写对应物。否则，字符大小写将返回字符。

```
（字符大小写 #\g）⇒ #\g
（字符下写 #\G） ⇒ #\g
（字符下写 #\7） ⇒ #\7
（字符下写 #\ς） ⇒ #\ς
```

过程：（字符标题大小写字符）
返回：字符
库的标题大小写字符等效项：（rnrs unicode），（rnrs）

如果 char 是大写或小写字符，并且具有单个标题大小写对应物，则 char-titlecase 将返回标题大小写对应物。否则，如果它不是标题大小写字符，没有单个标题大小写对应物，但确实有一个大写对应物，则 char-titlecase 将返回大写对应物。否则，字符标题大小写将返回字符。

```
（字符标题大小写 #\g）⇒ #\G
（字符标题大小写 #\G） ⇒ #\G
（字符标题大小写 #\7） ⇒ #\7
（字符标题大小写 #\ς） ⇒ #\Σ
```

过程：（字符折叠字符）
返回：大小写字符等效于字符
库：（rnrs unicode），（rnrs）

如果 char 具有大小写折叠的对应物，则 char 折叠套将返回大小写折叠的对应项。否则，字符折叠将返回字符。对于大多数字符，（char-foldcase char）等同于（char-downcase（char-upcase char）），但对于突厥语i和ı，char-foldcase充当标识。

```
（字符折叠套 #\g）⇒ #\g
（char-foldcase #\G） ⇒ #\g
（char-foldcase #\7） ⇒ #\7
（char-foldcase #\ς） ⇒ #\σ
```

过程：（char->integer char）
返回：char的Unicode标量值作为精确的整数
库：（rnrs base），（rnrs）

```
(char->integer #\newline) ⇒ 10
(char->integer #\space) ⇒ 32
(- (char->integer #\Z) (char->integer #\A)) ⇒ 25
```

过程：（整数>字符 n）
返回：与 Unicode 标量值 n 个
库对应的字符：（rnrs base）、（rnrs）

n 必须是精确的整数和有效的 Unicode 标量值，即 $0 \leq n \leq$ #xD7FF 或 #xE000 $\leq n \leq$ 10FFFF。

```
(整数>48) ⇒ #\0
(整数>字符#x3BB) ⇒ #\λ
```

第 6.8 节. 字符串

字符串是字符序列，通常用作消息、字符缓冲区或文本块的容器。Scheme 提供了用于创建字符串、从字符串中提取字符、获取子字符串、连接字符串以及更改字符串内容的操作。

字符串写为用双引号括起来的字符序列，例如“hi there”。双引号可以通过在字符串前面加上反斜杠来引入字符串，例如，“两个\”引号\“在”中。反斜杠也可以通过在它前面加上反斜杠来包含，例如，“a \\slash”。各种特殊字符可以与其他双字符序列一起插入，例如，\n 表示换行符，\r 表示回车符，\t 表示 Tab。任何 Unicode 字符都可以使用语法 #\xn; 插入，其中 n 由一个或多个十六进制数字组成，表示有效的 Unicode 标量值。定义字符串精确语法的语法在第 458 页上给出。

字符串由精确的非负整数编制索引，任何字符串的第一个元素的索引为 0。给定字符串的最高有效索引比其长度小 1。

过程：（字符串 =? string1 string2 string3 ...）
过程：（字符串<? string1 string2 string3 ...）
过程：（字符串>? string1 string2 string3 ...）

过程：（字符串<=? string1 string2 string3 ...）
过程：（字符串>=? string1 string2 string3 ...）
返回：如果关系成立，则#t，#f否则
库：（rnrs base），（rnrs）

与 =、<、>、<= 和 >= 一样，这些谓词表示所有参数之间的关系。例如，string>? 确定其参数的词典排序是否单调递减。

比较基于字符谓词 char=? 和 char<? 。如果两个字符串的长度相同，并且根据 char=? 由相同的字符序列组成，则它们在词典学上是等效的。如果两个字符串仅在长度上不同，则较短的字符串在字典上被视为小于较长的字符串。否则，根据 char<? ，字符串不同的第一个字符位置（通过 char=? ）确定哪个字符串在词典上小于另一个字符串。

可以定义双参数 string=? ，但不进行错误检查，如下所示。

```
(定义字符串 =?
(lambda (s1 s2)
(let ([n (string-length s1)])
(and (= (string-length s2) n)
(let loop ([i 0])
(or (= i n)
(and (char=? (字符串引用 s1 i) (string-ref s2 i))
(循环 (+ i 1)))))))))
```

可以定义双参数字符串<? ，但不进行错误检查，如下所示。

```
(定义字符串<?
(lambda (s1 s2)
(let ([n1 (string-length s1)] [n2 (string-length s2)])
(let loop ([i 0])
(and (not (= i n2))
(or (= i n1)
(let ([c1 (string-ref s1 i)] [c2 (string-ref s2 i)])
(or (char<? c1 c2)
(and (char=? c1 c2)
(loop (+ i 1)))))))))
```

这些定义可以直接扩展以支持三个或更多参数。string<=? 、string>? 和 string>=? 可以类似地定义。

```
(字符串 =? “妈妈” “妈妈”) ⇒ #t
(字符串<? “妈妈” “妈妈”) ⇒ #t
(字符串>? “爸爸” “爸爸”) ⇒ #f
(string=? “妈妈和爸爸” “妈妈和爸爸”) ⇒ #f
(字符串<? “a” “b” “c”) ⇒ #t
```

过程： (字符串 ci=? string1 string2 string3 ...)
过程： (字符串 ci<? string1 string2 string3 ...)
过程： (字符串 ci>? string1 string2 string3 ...)
过程： (字符串 ci<=? string1 string2 string3 ...)
过程： (字符串 ci>=? string1 string2 string3 ...)
返回：如果关系成立，则#t，#f否则
库：(rnrs unicode)，(rnrs)

这些谓词与 string=? 、string<? 、string>? 、string<=? 和 string>=? 相同，只是它们不区分大小写，即比较其参数的大小写折叠版本。

```
(字符串-ci=? “妈妈和爸爸” “妈妈和爸爸”) ⇒ #t
(string-ci<=? “说什么” “说什么！？”⇒ #t
(string-ci>? “N” “m” “L” “k”) ⇒ #t
(string-ci=? “Straße” “Strasse”) ⇒ #t
```

过程： (字符串字符 ...)
返回：包含字符字符的字符串 ...
库：(rnrs base)， (rnrs)

```
(字符串)⇒ “”
(字符串 #\a #\b #\c) ⇒ “abc”
(字符串 #\H #\E #\Y #\! )⇒ “嘿！”
```

过程： (make-string n)
过程： (make-string n char)
返回： 长度为 n 个库的字符串： (rnrs base)， (rnrs)

n 必须是精确的非负整数。如果提供了 char，则字符串将填充 n 个出现的 char，否则字符串中包含的字符是未指定的。

```
(组成字符串 0)⇒ “”
(make-string 0 #\x) ⇒ “”
(make-string 5 #\x) ⇒ “xxxxx”
```

过程：（字符串长度字符串）
返回：字符串
库中的字符数：（rnrs base），（rnrs）

字符串的长度始终是精确的非负整数。

```
（字符串长度“abc”）⇒ 3
（字符串长度 “”） ⇒ 0
（字符串长度 “hi there”） ⇒ 8
（字符串长度 （make-string 1000000）） ⇒ 1000000
```

过程：（字符串引用字符串 n）
返回：字符串
库的第 n 个字符（从零开始）：（rnrs base）、（rnrs）

n 必须是小于字符串长度的精确非负整数。

```
（字符串引用“你好”0）⇒ #\h
（string-ref “hi there” 5） ⇒ #\e
```

过程：（字符串集！字符串 n char）
返回： 未指定的
库： （rnrs 可变字符串）

n 必须是小于字符串长度的精确非负整数。字符串集！将字符串的第 n 个元素更改为 char。

```
（let （[str （string-copy “hi three”）]）
（string-set！ str 5 #\e）
（string-set！ str 6 #\r）
str）⇒ “你好”
```

过程：（字符串复制字符串）
返回：字符串
库的新副本：（rnrs base），（rnrs）

此过程将创建一个与字符串具有相同长度和内容的新字符串。

```
（字符串复制“abc”）⇒ “abc”

（let （[str “abc”]）
（eq？ str （string-copy str））） ⇒ #f
```

过程：（字符串追加字符串 ...）
返回：通过连接字符串字符串形成的新字符串 ...
库：（rnrs base），（rnrs）

```
(字符串追加) ⇒ ""
(string-append "abc" "def") ⇒ "abcdef"
(string-append "Hey" "you" "there!") ⇒ "嘿，你在那里！"
```

以下 string-append 实现沿字符串列表向下递归以计算总长度，然后分配新字符串，然后在展开递归时填充它。

```
(define string-append
(lambda args
(let f ([ls args] [n 0])
(if (null? ls)
(make-string n)
(let* ([s1 (car ls)]
[m (string-length s1)]
[s2 (f (cdr ls) (+ n m))])
(do ([i 0 (+ i 1)] [j n (+ j 1)])
((= i m) s2)
(string-set! s2 j (string-ref s1 i)))))))
```

过程：（子字符串字符串开始结束）
返回：从开始（包括）到结束（独占）
库的字符串副本：（rnrs base），（rnrs）

开始和结束必须是精确的非负整数;start 必须小于或等于 end，而 end 必须小于或等于字符串的长度。如果 end = start，则返回长度为零的字符串。可以在不进行错误检查的情况下定义子字符串，如下所示。

```
(define substring
(lambda (s1 m n)
(let ([s2 (make-string (- n m))])
(do ([j 0 (+ j 1)] [i m (+ i 1)])
((= i n) s2)
(string-set! s2 j (string-ref s1 i)))))

(子字符串 "hi there" 0 1) ⇒ "h"
(子字符串 "hi there" 3 6) ⇒ "the"
(子字符串 "hi there" 5 5) ⇒ ""

(let ([str "hi there"])
```

```
(let ([end (string-length str)])
(substr str 0 end))) ⇒ "hi there"
```

过程：（字符串填充！字符串 char）
返回： 未指定的
库： （rnrs 可变字符串）

字符串填充！将字符串中的每个字符设置为 char。

```
(let ([str (string-copy "sleepy")])
(string-fill! str #\Z)
str) ⇒ "ZZZZZZ"
```

字符串填充！可能定义如下：

```
(定义字符串填充!
(lambda (s c)
(let ([n (string-length s)])
(do ([i 0 (+ i 1)])
((= i n))
(string-set! s i c)))))
```

第276页给出了另一个定义。

过程：（字符串大写字符串）
返回：字符串
过程的大写等效项：（字符串小写字符串）
返回：字符串
的小写等效项过程：（字符串折叠字符串）
返回： 字符串
过程的大小写折叠等效项：（字符串-titlecase string）
返回：字符串
库的标题大小写等效项：（rnrs unicode）、 （rnrs）

这些过程实现 Unicode 与区域设置无关的案例映射，从标量值序列到标量值序列。这些映射并不总是将单个字符映射到单个字符，因此结果字符串的长度可能与字符串的长度不同。如果结果字符串与字符串相同（by string=?），则可能会返回字符串或字符串的副本。否则，将新分配结果字符串。string-foldcase不使用突厥语的特殊映射。

字符串标题大小写将字符串中每个单词的第一个大小写字符转换为其标题大小写对应字符，并将其他字符转换为小写对应字符。分词符可识别为 Unicode 标准附录 #29 [8] 中指定的分词符。

```
(字符串大写“Hi”) ⇒ “HI”
(string-downcase “Hi”) ⇒ “hi”
(string-foldcase “Hi”) ⇒ “hi”

(string-upcase “Straße”) ⇒ “STRASSE”
(string-downcase “Straße”) ⇒ “straße”
(string-foldcase “Straße”) ⇒ “strasse”
(string-downcase “STRASSE”) ⇒ “”
```

Σ⇒σ

```
(字符串标题大小写“kNock KNoCK”) ⇒ “Knock Knock”
(字符串标题大小写“谁在那里？”) ⇒ “谁在那里？”
(string-titlecase “r6rs”) ⇒ “R6rs”
(string-titlecase “R6RS”) ⇒ “R6rs”
```

过程：（字符串规范化-nfd 字符串）
返回：字符串
过程的 Unicode 规范化形式 D：（字符串规范化-nfkd 字符串）
返回：字符串
过程的 Unicode 规范化形式 KD：（字符串规范化-nfc 字符串）
返回：字符串
过程的 Unicode 规范化形式 C：（字符串规范化-nfkc 字符串）
返回：字符串
库的 Unicode 规范化形式 KC：（rnrs unicode），（rnrs）

如果结果字符串与字符串相同（by string=? ），则可能会返回字符串或字符串的副本。否则，将新分配结果字符串。

```
(string-normalize-nfd “\xE9;”) ⇒ “e\x301;”
(string-normalize-nfc “\xE9;”) ⇒ “\xE9;”
(string-normalize-nfd “\x65;\x301;”) ⇒ “e\x301;”
(string-normalize-nfc “\x65;\x301;”) ⇒ “\xE9;”
```

过程：（字符串>列表字符串）
返回：字符串
库中的字符列表：（rnrs base），（rnrs）

string->list 允许将字符串转换为列表，以便 Scheme 的列表处理操作可以应用于字符串的处理。字符串>列表可以在不进行错误检查的情况下定义，如下所示。

```
(define string->list
(lambda (s)
(do ([i (- (string-length s) 1) (- i 1)]
[ls '() (cons (string-ref s i) ls)])
((< i 0) ls))))

(string->list "") ⇒ ()
(string->list "abc") ⇒ (#\a #\b #\c)
(apply char<? (字符串>列表 "abc")) ⇒ #t
(map char-upcase (string->list "abc")) ⇒ (#\A #\B #\C)
```

过程：（列表>字符串列表）
返回：列表
库中字符的字符串：（rnrs base），（rnrs）

列表必须完全由字符组成。

list->string 是 string->list 的函数反函数。程序可以同时使用这两个过程，首先将字符串转换为列表，然后对此列表进行操作以生成新列表，最后将新列表转换回字符串。

可以定义 list-> 字符串，而无需进行错误检查，如下所示。

```
(define list->string
(lambda (ls)
(let ([s (make-string (length ls))])
(do ([ls ls (cdr ls)] [i 0 (+ i 1)])
((null? ls) s)
(string-set! s i (car ls))))))

(list->string '()) ⇒ ""
(list->string '(#\a #\b #\c)) ⇒ "abc"
(list->string
(map char-upcase
(string->list "abc"))) ⇒ "ABC"
```

第 6.9 节. 向量

对于某些应用程序，矢量比列表更方便、更高效。访问列表中的任意元素需要对列表进行线性遍历，直到所选元素，而任意矢量元素则在常量时间内访问。矢量的长度是它包含的元素数。向量由精确的非负整数编制索引，任何向量的第一个元素的索引为 0。给定向量的最高有效索引小于其长度 1。

与列表一样，矢量的元素可以是任何类型的，并且单个矢量可以包含多种类型的对象。

向量被写成一系列由空格分隔的对象，前面是前缀 #(后跟)。。例如，由元素 a、b 和 c 组成的向量将写为 #(a b c)。

过程： (向量 obj ...)
返回： 对象 obj ...
库的向量： (rnrs base)， (rnrs)

```
(矢量) ⇒ #()
(向量 'a 'b 'c) ⇒ #(a b c)
```

过程： (make-vector n)
过程： (make-vector n obj)
返回： 长度为 n
个库的向量： (rnrs base)， (rnrs)

n 必须是精确的非负整数。如果提供了obj，则向量的每个元素都用obj填充;否则，元素未指定。

```
(生成向量 0) ⇒ #()
(make-vector 0 '#(a)) ⇒ #()
(make-vector 5 '#(a)) ⇒ #(#(a) #(a) #(a) #(a) #(a) #(a) #(a))
```

过程：(矢量长度矢量)
返回：矢量
库中的元素数：(rnrs base)， (rnrs)

向量的长度始终是精确的非负整数。

```
(矢量长度 '#()) ⇒ 0
(vector-length '#(a b c)) ⇒ 3
```

```
(vector-length (vector 1 '(2) 3 '#(4 5))) ⇒ 4
(vector-length (make-vector 300)) ⇒ 300
```

过程：（向量引用向量 n）
返回：向量
库的第 n 个元素（从零开始）：（rnrs base）、（rnrs）

n 必须是小于向量长度的精确非负整数。

```
(vector-ref '#(a b c) 0) ⇒ a
(vector-ref '#(a b c) 1) ⇒ b
(vector-ref '#(x y z w) 3) ⇒ w
```

过程：（向量集！vector n obj）
返回： 未指定的
库： （rnrs base）， （rnrs）

n 必须是小于向量长度的精确非负整数。矢量集！将向量的第 n 个元素更改为 obj。

```
(let ([v (vector 'a 'b 'c 'd 'e)])
(vector-set! v 2 'x)
v) ⇒ #(a b x d e)
```

过程：（矢量填充！vector obj）
返回： 未指定的
库： （rnrs base）， （rnrs）

矢量填充！将向量的每个元素替换为 obj。可以在不进行错误检查的情况下定义它，如下所示。

```
(定义矢量填充！
(lambda (v x)
(let ([n (vector-length v)])
(do ([i 0 (+ i 1)])
((= i n))
(vector-set! v i x)))))

(let ([v (vector 1 2 3)])
(vector-fill! v 0)
v) ⇒ #(0 0 0)
```

过程：（向量>列表向量）
返回：向量
库的元素列表：（rnrs base），（rnrs）

向量>list 提供了一种将列表处理操作应用于向量的便捷方法。可以在不进行错误检查的情况下定义它，如下所示。

```
(定义 vector->list
(lambda (s)
(do ([i (- (vector-length s) 1) (- i 1)]
[ls '() (cons (vector-ref s i) ls)])
((< i 0) ls)))

(vector->list (vector)) ⇒ () (vector
->list '#(a b c)) ⇒ (a b c)

(let ((v '#(1 2 3 4 5)))
(apply * (vector->list v))) ⇒ 120
```

过程：（列表>向量列表）
返回：列表
库元素的向量：（rnrs base），（rnrs）

list->vector 是 vector->list 的函数反函数。这两个过程通常结合使用以利用列表处理操作。可以将向量转换为具有向量>列表的列表，以某种方式处理此列表以生成新列表，并将新列表转换回具有列表>向量的向量。

可以定义 list->vector 而不进行错误检查，如下所示。

```
(define list->vector
(lambda (ls)
(let ([s (make-vector (length ls))])
(do ([ls ls (cdr ls)] [i 0 (+ i 1)])
((null? ls) s)
(vector-set! s i (car ls))))))

(list->vector '()) ⇒ #()
(list->vector '(a b c)) ⇒ #(a b c)

(let ([v '#(1 2 3 4 5)])
(let ([ls (vector->list v)])
(list->vector (map * ls ls ls))))) ⇒ #(1 4 9 16 25)
```

过程：（向量排序谓词向量）
返回：包含向量元素的向量，根据谓词过程排序：（向量排序！谓词向量）
返回： 未指定的
库： （rnrs 排序）， （rnrs）

谓词应该是一个过程，它需要两个参数，如果它的第一个参数必须在排序向量中的第二个参数之前，则返回#t。也就是说，如果谓词应用于两个元素 x 和 y，其中 x 出现在输入向量中的 y 之后，则仅当 x 出现在输出向量中的 y 之前时，谓词才应返回 true。如果满足此约束，则向量排序将执行稳定排序，即仅在必要时根据谓词对两个元素进行重新排序。矢量排序！破坏性地执行排序，并且不一定执行稳定的排序。不会删除重复的元素。谓词不应有任何副作用。

向量排序可以调用谓词到 nlogn 时间，其中 n 是向量的长度，而向量排序！ 可以将谓词调用 n^2 次。vector-sort！ 的松散边界允许实现使用快速排序算法，在某些情况下，这比具有更紧密 nlogn 绑定的算法更快。

```
(矢量排序< '#(3 4 2 1 2 5)) ⇒ #(1 2 2 3 4 5)
(vector-sort > '#(0.5 1/2)) ⇒ #(0.5 1/2)
(vector-sort > '#(1/2 0.5)) ⇒ #(1/2 0.5)

(let ([v (vector 3 4 2 1 2 5)])
(vector-sort! < v)
⇒ #(1 2 2 3 4 5)
```

第 6.10 节. 字节向量

字节向量是原始二进制数据的向量。虽然名义上组织为精确无符号 8 位整数的序列，但字节向量可以解释为精确有符号 8 位整数、精确有符号或无符号 16 位、32 位、64 位或任意精度整数、IEEE 单精度或双浮点数或上述任意组合的序列。

字节向量的长度是它存储的 8 位字节数，并且向字节向量的索引始终以字节偏移量的形式给出。任何数据元素都可以在任何字节偏移处对齐，而不管底层硬件的对齐要求如何，并且可以使用与硬件规定的字节序不同的指定字节序（见下文）来表示。为 16 位、32 位和 64 位整数以及本机格式的单浮点数和双浮点数提供了特殊的、

通常更有效的运算符，即具有基础硬件的字节序，并存储在一个索引中，该索引是整数或浮点数的字节大小的倍数。

多字节数据值的字节序决定了它在内存中的布局方式。在大端格式中，值在较低索引处使用较高有效字节进行布局，而在小端格式中，该值在较高索引处使用较高有效字节进行布局。当字节向量过程接受字节序参数时，该参数可能是表示大端格式的符号大，也可能是表示小端格式的符号小。实现可以扩展这些过程以接受其他字节序符号。实现的本机字节序可以通过过程本机字节序获得。

字节向量用#vu8（前缀代替#（向量的前缀）编写，例如，#vu8（1 2 3）。以这种方式指定的字节向量的元素始终以 8 位无符号精确整数的形式给出，即从 0 到 255（包括 0 到 255）的整数，使用此类数字的任何有效语法编写。与字符串一样，字节向量是自我评估的，因此不需要引用它们。

```
'#vu8（1 2 3） ⇒ #vu8（1 2 3）
#vu8（1 2 3） ⇒ #vu8（1 2 3）
#vu8（#x3f #x7f #xbf #xff） ⇒ #vu8（63 127 191 255）
```

语法： （字节序符号）
返回： 符号
库： （rnrs bytevectors）， （rnrs）

符号必须是符号小、符号大或被实现识别为字节序符号的其他符号。如果符号不是符号，或者如果实现未将其识别为字节序符号，则存在语法冲突。

```
（字节序小）⇒小
（字节序大）⇒大
（字节序“垃圾邮件”） ⇒ 例外
```

过程：（本机字节序）
返回：命名实现的本机字节序
库的符号：（rnrs bytevectors）， （rnrs）

返回值是符号小、符号大或实现识别的其他一些字节序符号。它通常反映底层硬件的字节序。

```
（符号？（原生字节序））⇒ #t
```

过程： (make-bytevector n)
过程： (make-bytevector n fill)
返回： 长度为 n
的新字节向量 库： (rnrs bytevectors)， (rnrs)

如果提供了填充，则字节向量的每个元素都初始化为填充;否则，元素未指定。填充值必须是有符号或无符号的 8 位值，即介于 -128 到 255 之间的值（包括 -128 到 255）。负填充值被视为其二的补码等效值。

```
(make-bytevector 0) ⇒ #vu8()
(make-bytevector 0 7) ⇒ #vu8()
(make-bytevector 5 7) ⇒ #vu8(7 7 7 7 7 7)
(make-bytevector 5 -7) ⇒ #vu8(249 249 249 249 249 249)
```

过程： (字节向量长度字节量)
返回：8 位字节
库中字节向量的长度： (rnrs bytevectors)、 (rnrs)

```
(字节向量长度 #vu8()) ⇒ 0
(字节向量长度 #vu8(1 2 3)) ⇒ 3
(字节向量长度 (制造字节向量 300)) ⇒ 300
```

过程： (bytevector=? bytevector1 bytevector2)
返回：如果关系成立，则#t，否则#f
库： (rnrs bytevectors)、 (rnrs)

两个字节向量等于 bytevector=? 当且仅当它们具有相同的长度和相同的内容。

```
(bytevector=? #vu8() #vu8()) ⇒ #t
(bytevector=? (make-bytevector 3 0) #vu8(0 0 0)) ⇒ #t
(bytevector=? (make-bytevector 5 0) #vu8(0 0 0)) ⇒ #f
(bytevector=? #vu8(1 127 128 255) #vu8(255 128 127 1)) ⇒ #f
```

过程： (字节向量填充! bytevector fill)
返回：未指定的
库： (rnrs bytevectors)、 (rnrs)

填充值必须是有符号或无符号的 8 位值，即介于 -128 到 255 之间的值（包括 -128 到 255）。负填充值被视为其二的补码等效值。

字节向量填充！用填充替换字节向量的每个元素。

```
(let ([v (make-bytevector 6)])
(bytevector-fill! v 255)
v) ⇒ #vu8(255 255 255 255 255 255)

(let ([v (make-bytevector 6)])
(bytevector-fill! v -128)
v) ⇒ #vu8(128 128 128 128 128 128)
```

过程：(bytevector-copy bytevector)
返回：一个新的字节向量，它是字节向量
库的副本：(rnrs bytevectors)， (rnrs)

bytevector-copy 创建一个与 bytevector 具有相同长度和内容的新 bytevector。

```
(字节向量复制#vu8(1 127 128 255)) ⇒ #vu8(1 127 128 255)

(let ([v #vu8(1 127 128 255)])
(eq? v (bytevector-copy v))) ⇒ #f
```

过程：(字节向量复制！src src-start dst dst-start n)
返回： 未指定的
库： (rnrs bytevectors)， (rnrs)

src 和 dst 必须是字节向量。src-start、dst-start 和 n 必须是精确的非负整数。src-start 和 n 的总和不得超过 src 的长度，dst-start 和 n 的总和不得超过 dst 的长度。

字节向量复制！覆盖从 dst 开始的 dst 开始的 dst 的 n 个字节，而 src 的 n 个字节从 src 开始。即使 dst 与 src 是同一个字节向量，并且源位置和目标位置重叠，这也有效。也就是说，目标中充满了在操作开始之前出现在源中的字节。

```
(定义 v1 #vu8(31 63 95 127 159 191 223 255))
(定义 v2 (make-bytevector 10 0))

(bytevector-copy! v1 2 v2 1 4)
v2 ⇒ #vu8(0 95 127 159 191 0 0 0 0 0)

(bytevector-copy! v1 5 v2 7 3)
v2 ⇒ #vu8(0 95 127 159 191 0 0 191 223 255)
```

```
(bytevector-copy! v2 3 v2 0 6)
v2 ⇒ #vu8(159 191 0 0 191 223 0 191 223 255)

(bytevector-copy! v2 0 v2 1 9)
v2 ⇒ #vu8(159 159 191 0 0 191 223 0 191 223)
```

过程： (bytevector-u8-ref bytevector n)
返回：字节向量库的索引 n 处的 8 位无符号字节
（从零开始）：（rnrs bytevectors）、（rnrs）

n 必须是小于字节向量长度的精确非负整数。

该值以精确的 8 位无符号整数的形式返回，即介于 0 到 255 之间的值（包括 0 到 255）。

```
(bytevector-u8-ref #vu8(1 127 128 255) 0) ⇒ 1
(bytevector-u8-ref #vu8(1 127 128 255) 2) ⇒ 128
(bytevector-u8-ref #vu8(1 127 128 255) 3) ⇒ 255
```

过程： (bytevector-s8-ref bytevector n)
返回：字节向量库的索引 n（从零开始）处的 8 位有符号字节
：（rnrs bytevectors）、（rnrs）

n 必须是小于字节向量长度的精确非负整数。

返回的值是一个精确的 8 位有符号整数，即介于 -128 到 127（包括 -128 和 127）之间的值，并且等效于被视为二进制补码值的存储值。

```
(字节向量-s8-ref #vu8(1 127 128 255) 0) ⇒ 1
(bytevector-s8-ref #vu8(1 127 128 255) 1) ⇒ 127
(bytevector-s8-ref #vu8(1 127 128 255) 2) ⇒ -128
(bytevector-s8-ref #vu8(1 127 128 255) 3) ⇒ -1
```

过程： (bytevector-u8-set! bytevector n u8)
返回： 未指定的
库： (rnrs bytevectors)， (rnrs)

n 必须是小于字节向量长度的精确非负整数。u8 必须是 8 位无符号值，即介于 0 到 255 之间的值（包括 0 到 255）。

bytevector-u8-set! 将字节向量的索引 n（从零开始）处的 8 位值更改为 u8。

```
(let ([v (make-bytevector 5 -1)])
(bytevector-u8-set! v 2 128)
v) ⇒ #vu8(255 255 128 255 255)
```

过程： (字节向量-s8-set! bytevector n s8)
返回： 未指定的
库： (rnrs bytevectors)， (rnrs)

n 必须是小于字节向量长度的精确非负整数。s8 必须是 8 位有符号值，即介于 -128 到 127 之间的值（包括 -128 和 127）。

bytevector-s8-set! 将字节向量的索引 n（从零开始）处的 8 位值更改为两者的补集等效值 s8。

```
(let ([v (make-bytevector 4 0)])
(bytevector-s8-set! v 1 100)
(bytevector-s8-set! v 2 -100)
v) ⇒ #vu8(0 100 156 0)
```

过程： (bytevector->u8-list bytevector)
返回：字节向量
库的8位无符号元素的列表： (rnrs bytevectors)， (rnrs)

```
(bytevector->u8-list (make-bytevector 0)) ⇒ ()
(bytevector->u8-list #vu8(1 127 128 255)) ⇒ (1 127 128 255)

(let ([v #vu8(1 2 3 255)])
(apply * (bytevector->u8-list v))) ⇒ 1530
```

过程： (u8-list->bytevector list)
返回：列表
库元素的新字节向量： (rnrs bytevectors)， (rnrs)

list 必须完全由精确的 8 位无符号整数组成，即介于 0 到 255 之间的值（包括 0 到 255）。

```
(u8-list->bytevector '()) ⇒ #vu8()
(u8-list->bytevector '(1 127 128 255)) ⇒ #vu8(1 127 128 255)
```

```
(let ([v #vu8(1 2 3 4 5)])
(let ([ls (bytevector->u8-list v)])
(u8-list->bytevector (map * ls ls)))) ⇒ #vu8(1 4 9 16 25)
```

过程：(bytevector-u16-native-ref bytevector n)
返回：字节向量过程：(bytevector-s16-native-ref bytevector n)的索引 n 处的 16 位无符号整数（从零开始）
返回：索引 n 处的 16 位无符号整数字节向量
过程：(bytevector-u32-native-ref bytevector n)
返回：字节向量过程 (bytevector-s32-native-ref bytevector n)的索引 n 处的 32 位无符号整数（从零开始）
返回：索引 n 处的 32 位有符号整数 字节向量
过程的（从零开始）：(bytevector-u64-native-ref bytevector n)
返回：字节向量
过程：(bytevector-s64-native-ref bytevector n)的索引 n 处的 64 位无符号整数（从零开始）
返回：字节向量
库的索引 n 处的 64 位有符号整数（从零开始）：(rnrs bytevectors)、 (rnrs)

n 必须是精确的非负整数。它对值的起始字节编制索引，并且必须是该值所占用的字节数的倍数：2 表示 16 位值，4 表示 32 位值，8 表示 64 位值。n 的总和以及该值占用的字节数不得超过字节向量的长度。假定为本机字节序。

返回值是值所占用的字节数的适当范围内的精确整数。有符号值等效于被视为二进制补码值的存储值。

```
(定义 v #vu8(#x12 #x34 #xfe #x56 #xdc #xba #x78 #x98))
```

如果本机字节序很大：

```
(bytevector-u16-native-ref v 2) ⇒ #xfe56
(bytevector-s16-native-ref v 2) ⇒ #x-1aa
(bytevector-s16-native-ref v 6) ⇒ #x7898

(bytevector-u32-native-ref v 0) ⇒ #x1234fe56
(bytevector-s32-native-ref v 0) ⇒ #x1234fe56
(bytevector-s32-native-ref v 4) ⇒ #x-23458768

(bytevector-u64-native-ref v 0) ⇒ #x1234fe56dcba7898
(bytevector-s64-native-ref v 0) ⇒ #x1234fe56dcba7898
```

如果原生字节序很小：

```
(bytevector-u16-native-ref v 2) ⇒ #x56fe
(bytevector-s16-native-ref v 2) ⇒ #x56fe
(bytevector-s16-native-ref v 6) ⇒ #x-6788

(bytevector-u32-native-ref v 0) ⇒ #x56fe3412
(bytevector-s32-native-ref v 0) ⇒ #x56fe3412
(bytevector-s32-native-ref v 4) ⇒ #x-67874524

(bytevector-u64-native-ref v 0) ⇒ #x9878badc56fe3412
(bytevector-s64-native-ref v 0) ⇒ #x-67874523a901cbee
```

过程： (bytevector-u16-native-set! bytevector n u16)
过程： (bytevector-s16-native-set! bytevector n s16)
过程： (bytevector-u32-native-set! bytevector n u32)
过程： (bytevector-s32-native-set! bytevector n s32)
过程： (bytevector-u64-native-set! bytevector n u64)
过程： (bytevector-s64-native-set! bytevector n s64)
返回： 未指定的
库： (rnrs bytevectors)， (rnrs)

n 必须是精确的非负整数。它对值的起始字节编制索引，并且必须是该值所占用的字节数的倍数：2 表示 16 位值，4 表示 32 位值，8 表示 64 位值。n 的总和以及该值占用的字节数不得超过长度字节。u16 必须是 16 位无符号值，即介于 0 到 $216 - 1$ 之间的值（包括 0 到 216 - 1）；s16 必须是 16 位有符号值，即介于 -215 到 $215 - 1$ 之间的值（包括 -215 到 $215 - 1$）；u32 必须是 32 位无符号值，即介于 0 到 232 - 1 之间的值（包括 0 到 $232 - 1$）；s32 必须是 32 位有符号值，即介于 -231 到 $231 - 1$ 之间的值（包括 $-231 - 1$）；u64 必须是 64 位无符号值，即介于 0 到 264 - 1 之间的值（包括 0 到 $264 - 1$）；并且 s64 必须是 64 位有符号值，即介于 -263 到 $263 - 1$ 之间的值（包括 -263 到 $263 - 1$）。假定为本机字节序。

这些过程将给定值存储在字节向量的索引 n（从零开始）开始的 2、4 或 8 个字节中。负值存储为两者的补数等效值。

```
(定义 v (make-bytevector 8 0))
(bytevector-u16-native-set! v 0 #xfe56)
```

```
(bytevector-s16-native-set!  v 2 #x-1aa)
(bytevector-s16-native-set!  v 4 #x7898)
```

如果本机字节序很大：

```
v ⇒ #vu8(#xfe #x56 #xfe #x56 #x78 #x98 #x00 #x00)
```

如果原生字节序很小：

```
v ⇒ #vu8(#x56 #xfe #x56 #xfe #x98 #x78 #x00 #x00)
```

```
(定义 v  (make-bytevector 16 0))
(bytevector-u32-native-set!  v 0 #x1234fe56)
(bytevector-s32-native-set!  v 4 #x1234fe56)
(bytevector-s32-native-set!  v 8 #x-23458768)
```

如果本机字节序很大：

```
v ⇒ #vu8(#x12 #x34 #xfe #x56 #x12 #x34 #xfe #x56
  #xdc #xba #x78 #x98 #x00 #x00 #x00 #x00)
```

如果原生字节序很小：

```
v ⇒ #vu8(#x56 #xfe #x34 #x12 #x56 #xfe #x34 #x12
  #x98 #x78 #xba #xdc #x00 #x00 #x00 #x00)
```

```
(定义 v  (make-bytevector 24 0))
(bytevector-u64-native-set!  v 0 #x1234fe56dcba7898)
(bytevector-s64-native-set!  v 8 #x1234fe56dcba7898)
(bytevector-s64-native-set!  v 16 #x-67874523a901cbee)
```

如果本机字节序很大：

```
v ⇒ #vu8(#x12 #x34 #xfe #x56 #xdc #xba #x78 #x98
  #x12 #x34 #xfe #x56 #xdc #xba #x78 #x98
  #x98 #x78 #xba #xdc #x56 #xfe #x34 #x12)
```

如果原生字节序很小：

```
v ⇒ #vu8(#x98 #x78 #xba #xdc #x56 #xfe #x34 #x12
  #x98 #x78 #xba #xdc #x56 #xfe #x34 #x12
  #x12 #x34 #xfe #x56 #xdc #xba #x78 #x98)
```

过程：（bytevector-u16-ref bytevector n eness）
返回：字节向量过程（ bytevector-s16-ref bytevector n eness）的索引n处的16位无符号整数（从零开始）：索引n处的16位有符号整数（从零开始）的字节向量
过程：（bytevector-u32-ref bytevector n eness）
返回：字节向量
过程：（bytevector-s32-ref bytevector n eness）的索引n处的32位无符号整数（从零开始）
返回： 字节向量过程：（bytevector-u64-ref bytevector n eness）的索引 n 处的 32 位有符号整数（从零开始）：

（bytevector-s64-ref bytevector n eness）
返回：字节向量
库的索引 n（从零开始）处的 64 位有符号整数：（rnrs bytevectors）、 （rnrs）

n 必须是精确的非负整数，并为值的起始字节编制索引。n 的总和该值占用的字节数（2 表示 16 位值，4 表示 32 位值，8 表示 32 位值）不得超过字节向量的长度。n 不必是该值所占用字节数的倍数。eness 必须是命名字节序的有效字节序符号。

返回值是值所占用的字节数的适当范围内的精确整数。有符号值等效于被视为二进制补码值的存储值。

```
（定义 v #vu8（#x12 #x34 #xfe #x56 #xdc #xba #x78 #x98 #x9a #x76））
(bytevector-u16-ref v 0 (endianness big)) ⇒ #x1234
(bytevector-s16-ref v 1 (endianness big)) ⇒ #x34fe
(bytevector-s16-ref v 5 (endianness big)) ⇒ #x-4588

(bytevector-u32-ref v 2 'big) ⇒ #xfe56dcba
(bytevector-s32-ref v 3 'big) ⇒ #x56dcba78
(bytevector-s32-ref v 4 'big) ⇒ #x-23458768

(bytevector-u64-ref v 0 'big) ⇒ #x1234fe56dcba7898
(bytevector-s64-ref v 1 'big) ⇒ #x34fe56dcba78989a

(bytevector-u16-ref v 0 (endianness little)) ⇒ #x3412
(bytevector-s16-ref v 1 (endianness little)) ⇒ #x-1cc
(bytevector-s16-ref v 5 (endianness little)) ⇒ #x78ba

(bytevector-u32-ref v 2 'little) ⇒ #xbadc56fe
```

```
(bytevector-s32-ref v 3 'little) ⇒ #x78badc56
(bytevector-s32-ref v 4 'little) ⇒ #x-67874524

(bytevector-u64-ref v 0 'little) ⇒ #x9878badc56fe3412
(bytevector-s64-ref v 1 'little) ⇒ #x-6567874523a901cc
```

过程： (bytevector-u16-set! bytevector n u16 eness)
过程： (bytevector-s16-set! bytevector n s16 eness)
过程： (bytevector-u32-set! bytevector n u32 eness)
过程： (bytevector-s32-set! bytevector n s32 eness)
过程： (bytevector-u64-set! bytevector n u64 eness)
过程： (bytevector-s64-set! bytevector n s64 eness)
返回： 未指定的
库： (rnrs bytevectors)， (rnrs)

n 必须是精确的非负整数，并为值的起始字节编制索引。n 的总和以及该值占用的字节数不得超过字节向量的长度。n 不必是该值所占用字节数的倍数。u16 必须是 16 位无符号值，即介于 0 到 $2^{16} - 1$ 之间的值（包括 0 到 216 - 1）;s16 必须是 16 位有符号值，即介于 -2^{15} 到 $2^{15} - 1$ 之间的值（包括 -2^{15} 到 $2^{15} - 1$）;u32 必须是 32 位无符号值，即介于 0 到 232 - 1 之间的值（包括 0 到 $2^{32} - 1$）;s32 必须是 32 位有符号值，即介于 -2^{31} 到 $2^{31} - 1$ 之间的值（包括 $-2^{31} - 1$）;u64 必须是 64 位无符号值，即介于 0 到 264 - 1 之间的值（包括 0 到 $2^{64} - 1$）;并且 s64 必须是 64 位有符号值，即介于 -2^{63} 到 $2^{63} - 1$ 之间的值（包括 -2^{63} 到 $2^{63} - 1$）。eness 必须是命名字节序的有效字节序符号。

这些过程将给定值存储在字节向量的索引 n（从零开始）开始的 2、4 或 8 个字节中。负值存储为两者的补数等效值。

```
(定义 v (make-bytevector 8 0))
(bytevector-u16-set! v 0 #xfe56 (endianness big))
(bytevector-s16-set! v 3 #x-1aa (endianness little))
(bytevector-s16-set! v 5 #x7898 (endianness big))
v ⇒ #vu8(#xfe #x56 #x0 #x56 #xfe #x78 #x98 #x0)

(define v (make-bytevector 16 0))
(bytevector-u32-set! v 0 #x1234fe56 'little)
(bytevector-s32-set! v 6 #x1234fe56 'big)
(bytevector-s32-set! v 11 #x-23458768 'little)
```

```
v⇒ #vu8（#x56 #xfe #x34 #x12 #x0 #x0
  #x12 #x34 #xfe #x56 #x0
  #x98 #x78 #xba #xdc #x0）

（define v （make-bytevector 28 0））
（bytevector-u64-set! v 0 #x1234fe56dcba7898 'little）
（bytevector-s64-set! v 10 #x1234fe56dcba7898 'big）
（bytevector-s64-set! v 19 #x-67874523a901cbee 'big）
v ⇒ #vu8（#x98 #x78 #xba #xdc #x56 #xfe #x34 #x12 #x0 #x0
#x12 #x34 #xfe #x56 #xdc #xba #x78 #x98 #x0   #x98 #x78 #xba #xdc
#x56 #xfe #x34 #x12 #x0）
```

过程： （bytevector-uint-ref bytevector n eness size）
返回： 大小-字节无符号整数在索引 n（从零开始）的字节向量
过程： （bytevector-sint-ref bytevector n eness size）
返回：字节向量
库的索引 n 处的大小字节有符号整数（从零开始）：（rnrs bytevectors）、（rnrs）

n 必须是精确的非负整数，并为值的起始字节编制索引。size 必须是精确的正整数，并指定值占用的字节数。n 和大小之和不得超过字节向量的长度。n 不必是该值所占用字节数的倍数。eness 必须是命名字节序的有效字节序符号。

返回值是值所占用的字节数的适当范围内的精确整数。有符号值等效于被视为二进制补码值的存储值。

```
（定义 v #vu8（#x12 #x34 #xfe #x56 #xdc #xba #x78 #x98 #x9a
#x76））

（bytevector-uint-ref v 0 'big 1） ⇒ #x12
（bytevector-uint-ref v 0 'little 1） ⇒ #x12
（bytevector-uint-ref v 1 'big 3） ⇒ #x34fe56
（bytevector-uint-ref v 2 'little 7） ⇒ #x9a9878badc56fe

（bytevector-sint-ref v 2 'big 1） ⇒ #x-02
（bytevector-sint-ref v 1 'little 6） ⇒ #x78badc56fe34
（bytevector-sint-ref v 2 'little 7） ⇒ #x-6567874523a902

（bytevector-sint-ref （make-bytevector 1000 -1） 0 'big 1000） ⇒
-1
```

过程： (bytevector-uint-set! bytevector n uint eness size)
过程： (bytevector-sint-set! bytevector n sint eness size)
返回： 未指定的
库： (rnrs bytevectors)， (rnrs)

n 必须是精确的非负整数，并为值的起始字节编制索引。size 必须是精确的正整数，并指定值占用的字节数。n 和大小之和不得超过字节向量的长度。n 不必是该值所占用字节数的倍数。uint 必须是介于 0 到 $2^{size \cdot 8} - 1$（包括 0 到 2size•8 - 1）之间的精确整数。sint 必须是介于 $-2^{size \cdot 8-1}$ 到 $2^{size \cdot 8-1} - 1$（包括 $-2^{size \cdot 8-1}$-1）之间的精确整数。eness 必须是命名字节序的有效字节序符号。

这些过程将给定值存储在字节大小中，从字节的索引 n（从零开始）开始。负值存储为两者的补数等效值。

```
(定义 v (make-bytevector 5 0))
(bytevector-uint-set! v 1 #x123456 (endianness big) 3)
v ⇒ #vu8(0 #x12 #x34 #x56 0)

(define v (make-bytevector 7 -1))
(bytevector-sint-set! v 1 #x-8000000000 (endianness little)
5)
v ⇒ #vu8(#xff 0 0 0 0 #x80 #xff)
```

过程： (bytevector->uint-list bytevector eness size)
返回：字节-字节无符号元素的新列表 bytevector
过程： (bytevector->sint-list bytevector eness size)
返回：大小的新列表字节向量
库的字节签名元素： (rnrs bytevectors)， (rnrs)

eness 必须是命名字节序的有效字节序符号。size 必须是精确的正整数，并指定值占用的字节数。它必须是平均划分字节向量长度的值。

```
(bytevector->uint-list (make-bytevector 0) 'little 3) ⇒ ()

(let ([v #vu8(1 2 3 4 5 6)])
(bytevector->uint-list v 'big 3)) ⇒ (#x010203 #x040506)

(let ([v (make-bytevector 80 -1)])
(bytevector->sint-list v 'big 20)) ⇒ (-1 -1 -1 -1)
```

过程： (uint-list->bytevector list eness size)
过程： (sint-list->bytevector list eness size)
返回： 列表
库元素的新字节向量： (rnrs bytevectors)， (rnrs)

eness 必须是命名字节序的有效字节序符号。size 必须是精确的正整数，并指定值占用的字节数。对于 uint-list->bytevector，list 必须完全由大小字节的精确无符号整数组成，即介于 0 到 2size•8 - 1（包括 0 到 2size•8 - 1）之间的值。对于 sint-list->bytevector，list 必须完全由大小字节的精确有符号整数组成，即介于 _2size•8-1 到 2size•8-1 之间（包括 -2size•8-1）的值。每个值在生成的字节数中占据大小字节，因此其长度是大小乘以列表的长度。

```
(uint-list->bytevector '() 'big 25) ⇒ #vu8 ()
(sint-list->bytevector '(0 -1) 'big 3) ⇒ #vu8 (0 0 0 #xff #xff
#xff)

(define (f size)
(let ([ls (list (- (expt 2 (- (* 8 size) 1)))
(- (expt 2 (- (* 8 size) 1)) 1)))])
(sint-list->bytevector ls 'little size)))
(f 6) ⇒ #vu8 (#x00 #x00 #x00 #x00 #x00 #x80
  #xff #xff #xff #xff #xff #x7f)
```

过程： (bytevector-ieee-single-native-ref bytevector n)
返回：字节向量
过程： (bytevector-ieee-double-native-ref bytevector n) 的索引 n 处的单浮点值（从零开始）：
索引 n 处的双浮点值 字节向量
库的（从零开始）： (rnrs bytevectors)， (rnrs)

n 必须是精确的非负整数。它索引值的起始字节，并且必须是值所占用字节数的倍数：4 表示单浮点数，8 表示双精度型。n 的总和以及该值占用的字节数不得超过字节向量的长度。假定为本机字节序。

返回值是不精确的实数。示例出现在下面的突变运算符之后。

procedure: (bytevector-ieee-single-native-set! bytevector n x)

过程： (bytevector-ieee-double-native-set! bytevector n x)
返回： 未指定的
库： (rnrs bytevectors)， (rnrs)

n 必须是精确的非负整数。它索引值的起始字节，并且必须是值所占用字节数的倍数：4 表示单浮点数，8 表示双精度型。n 的总和以及该值占用的字节数不得超过字节向量的长度。假定为本机字节序。

这些过程将给定值存储为 IEEE-754 单浮点值或双浮点值，位于字节向量的索引 n（从零开始）。

```
(定义 v (make-bytevector 8 0))
(bytevector-ieee-single-native-set! v 0 .125)
(bytevector-ieee-single-native-set! v 4 -3/2)
(list
(bytevector-ieee-single-native-ref v 0)
(bytevector-ieee-single-native-ref v 4)) ⇒ (0.125 -1.5)

(bytevector-ieee-double-native-set! v 0 1e23)
(bytevector-ieee-double-native-ref v 0) ⇒ 1e23
```

过程： (bytevector-ieee-single-ref bytevector n eness)
返回：字节向量
过程 (bytevector-ieee-double-ref bytevector n eness) 的索引n（从零开始）
处的单浮点值返回：索引处的双浮点值n（从零开始）的字节向量
库： (rnrs bytevectors)， (rnrs)

n 必须是精确的非负整数，并为值的起始字节编制索引。n 之和和值占用的字节数（单个浮点数为 4，双精度型为 8）不得超过字节向量的长度。n 不必是该值所占用字节数的倍数。eness 必须是命名字节序的有效字节序符号。

返回值是不精确的实数。示例出现在下面的突变运算符之后。

过程： (bytevector-ieee-single-set! bytevector n x eness)
过程： (bytevector-ieee-double-set! bytevector n x eness)
返回： 未指定的
库： (rnrs bytevectors)， (rnrs)

n 必须是精确的非负整数，并为值的起始字节编制索引。n 之和和值占用的字节数（单个浮点数为 4，双精度型为 8）不得超过字节向量的长度。n 不必是该值所占用字节数的倍数。eness 必须是命名字节序的有效字节序符号。

这些过程将给定值存储为 IEEE-754 单浮点值或双浮点值，位于字节向量的索引 n（从零开始）。

```
(定义 v (make-bytevector 10 #xc7))
(bytevector-ieee-single-set! v 1 .125 'little)
(bytevector-ieee-single-set! v 6 -3/2 'big)
(list
(bytevector-ieee-single-ref v 1 'little)
(bytevector-ieee-single-ref v 6 'big)) ⇒ (0.125 -1.5)
v ⇒ #vu8(#xc7 #x0 #x0 #x0 #x3e #xc7 #xbf #xc0 #x0 #x0)

(bytevector-ieee-double-set! v 1 1e23 'big)
(bytevector-ieee-double-ref v 1 'big) ⇒ 1e23
```

第 6.11 节.符号

符号用于各种目的，作为方案程序中的符号名称。字符串可以用于大多数相同的目的，但符号的一个重要特征使符号之间的比较更加有效。这个特征是两个同名的符号在eq? 的意义上是相同的。原因是 Scheme 读取器（由 get-datum 调用并读取）和过程字符串>符号目录符号在内部符号表中，并且每当遇到相同的名称时，始终返回相同的符号。因此，不需要逐个字符进行比较，就像比较两个字符串一样。

两个符号可以快速比较为等效性的性质使它们非常适合用作程序表示中的标识符，从而允许快速比较标识符。此属性还使符号可用于各种其他用途。例如，符号可以用作在过程之间传递的消息、列表结构化记录的标签或存储在关联列表中的对象的名称（请参见第 6.3 节中的 assq）。

符号的编写不带双引号或其他括号字符。括号、双引号、空格和大多数对方案阅读器具有特殊含义的字符不允许出现在符号的打印表示中。这些字符和任何其他 Unicode 字符可能出现在语法为 #\xn; 的符号的打印表示形式的任意位置，其中 n 由一个或多个十六进制数字组成，并表示有效的 Unicode 标量值。

第 458 页上符号的语法给出了符号语法的精确定义。

过程： （符号=? symbol1 symbol2）
返回： 如果两个符号相同，则#t，#f否则
库： （rnrs base）、 （rnrs）

符号也可以与eq？进行比较，eq？通常比stick=？更有效。

```
（符号=? ’a ’a） ⇒ #t
（symbol=? ’a （string->symbol “a”））⇒ #t
（symbol=? ’a ’b）⇒ #f
```

过程：（字符串>符号字符串）
返回：名称为字符串
库的符号： （rnrs base）， （rnrs）

string->symbol 记录它在与系统读取器共享的内部表中创建的所有符号。如果表中已存在名称等效于字符串（根据谓词 string=？）的符号，则返回此符号。否则，将创建一个以字符串作为其名称的新交易品种;此符号将输入到表中并返回。

在字符串用作字符串>符号的参数后修改字符串的效果是未指定的。

```
（字符串>符号“x”）⇒ x

（等式? （字符串>符号“x”）’x）⇒ #t
（eq? （字符串>符号“X”）’x）⇒ #f

（eq? （字符串>符号“x”）
（string->symbol “x”））⇒ #t

（string->symbol “（）”）⇒ \x28;\x29;
```

过程：（符号>字符串符号）
返回：一个字符串，符号
库的名称： （rnrs base）， （rnrs）

符号>字符串返回的字符串应被视为不可变的。如果传递给 string->symbol 的字符串被 string-set！ 或任何其他方式更改，则可能会导致不可预知的行为。

```
(符号->字符串 'xyz) ⇒ "xyz"
(symbol->string 'Hi) ⇒ "Hi"
(symbol->string (string->symbol "()")) ⇒ "()"
```

第 6.12 节. 布尔 值

虽然每个 Scheme 对象在条件上下文中使用时都有一个真值，但#f的每个对象都算作 true，但 Scheme 提供了专用的 true 值，#t表达式的值只应传达它为 true 时使用。

过程： (布尔值 =? boolean1 boolean2)
返回：如果两个布尔值相同，则#t，#f否则
库： (rnrs base)、 (rnrs)

#t和#f的布尔值也可以与eq? 进行比较，eq? 通常比布尔=? 更有效。

```
(布尔值=? #t #t) ⇒ #t
(boolean=? #t #f) ⇒ #f
(boolean=? #t (< 3 4)) ⇒ #t
```

第 6.13 节. 哈希表

哈希表表示任意 Scheme 值之间的关联集。它们的作用与关联列表基本相同（见第165页），但在涉及大量关联时通常要快得多。

过程： (make-eq-hashtable)
过程： (make-eq-hashtable size)
返回： 一个新的可变 eq hashtable
库： (rnrs hashtables)， (rnrs)

如果提供了 size，则它必须是一个非负精确整数，指示哈希表最初应包含的元素数。哈希表根据需要增长，但是当哈希表增长时，它通常必须重新哈希所有现有元素。提供非零大小有助于限制在最初填充表时必须执行的重新哈希处理量。

eq 哈希表使用 eq? （指针相等）过程比较键，并且通常使用基于对象地址的哈希函数。它的哈希和等价函数适用于任何 Scheme 对象。

```
（定义ht1 （make-eq-hashtable））
（定义 ht2 （make-eq-hashtable 32））
```

过程： （make-eqv-hashtable）
过程： （make-eqv-hashtable size）
返回： 一个新的可变 eqv hashtable
库： （rnrs hashtables）， （rnrs）

如果提供了 size，则它必须是一个非负精确整数，指示哈希表最初应包含的元素数。哈希表根据需要增长，但是当哈希表增长时，它通常必须重新哈希所有现有元素。提供非零大小有助于限制在最初填充表时必须执行的重新哈希处理量。

eqv 哈希表使用 eqv? 过程比较键，并且通常采用基于对象地址的哈希函数，用于可识别 eq? 的对象。它的哈希和等价函数适用于任何 Scheme 对象。

程序： （使哈希算符等效？
过程： （make-hashtable hash equiv? size）
返回：一个新的可变哈希表
库： （rnrs hashtables）， （rnrs）

哈希和等价？必须是程序。如果提供了 size，则它必须是一个非负精确整数，指示哈希表最初应包含的元素数。哈希表根据需要增长，但是当哈希表增长时，它通常必须重新哈希所有现有元素。提供非零大小有助于限制在最初填充表时必须执行的重新哈希处理量。

新的哈希表使用哈希计算哈希值，并使用 equiv? 比较键，这两者都不应修改哈希表。等价？应比较两个键，并且仅当应区分两个键时才返回 false。哈希应该接受一个键作为参数，并返回一个非负的精确整数值，每次调用它时，这个值都是相同的，而 equiv? 没有区别。只要哈希表仅用于它们接受的键，哈希和等价过程就不需要接受任意输入，并且这两个过程都可以假定，只要在表中存储关联时不修改键，这两个过程都可以假定键是不可变的。哈希表操作可以调用哈希和等效？一次，根本不调用，或者为每个哈希表操作多次。

```
（定义ht （make-hashtable string-hash string=? ））
```

过程： （哈希表可变？哈希算）
返回： 如果哈希表是可变的，则#t，否则#f

库：（rnrs hashtables），（rnrs）

由上述哈希表创建过程之一返回的哈希表是可变的，但由哈希表副本创建的哈希表可能是不可变的。不可变哈希表不能通过任何过程 hashtable-set!、hashtable-update!、hashtable-delete! 或 hashtable-clear! 进行更改。

```
(hashtable-mutable? (make-eq-hashtable)) ⇒ #t
(hashtable-mutable? (hashtable-copy (make-eq-hashtable)) ⇒ #f
```

过程：(hashtable-hash-function hashtable)
返回：与 hashtable
过程关联的哈希函数：(hashtable-equivalence-function hashtable)
返回：与 hashtable
库关联的等效函数：（rnrs hashtables），（rnrs）

hashtable-hash-function 为 eq 和 eqv hashtable 返回#f。

```
(定义ht (make-eq-hashtable))
(hashtable-hash-function ht) ⇒ #f
(eq? (hashtable-equivalence-function ht) eq?) ⇒
#t
(define ht (make-hashtable string-hash string=?))
(等式? (hashtable-hash-function ht) string-hash) ⇒ #t
(eq? (hashtable-equivalence-function ht) string=?) ⇒ #t
```

过程：(等哈希 obj)
过程：(字符串哈希字符串)
过程：(字符串-ci-哈希字符串)
过程：(符号 -哈希符号)
返回：一个精确的非负整数哈希值
库：（rnrs 哈希表）、（rnrs）

这些过程是哈希函数，适合与适当的 Scheme 谓词一起使用：equal? 表示 equal-hash，string=? 表示 string-hash，string-ci=? 表示 string-ci-hash，symbol=?（或 eq?）表示 symbol-hash。由等哈希、字符串哈希和字符串 ci 哈希返回的哈希值通常依赖于输入值的当前结构和内容，因此，如果在哈希表中具有关联时修改键，则不适合这些哈希值。

过程：（哈希表集！hashtable key obj）
返回：未指定的
库：（rnrs hashtables），（rnrs）

哈希表必须是可变哈希表。key 应该是哈希表的哈希和等价函数的适当键。obj 可以是任何方案对象。

哈希表集！将密钥与哈希表中的 obj 关联，替换现有关联（如果有）。

```
（定义ht （make-eq-hashtable））
（hashtable-set！ ht ’a 73）
```

过程：（hashtable-ref hashtable key default）
返回：参见下面的
库：（rnrs hashtables），（rnrs）

key 应该是哈希表的哈希和等价函数的适当键。默认值可以是任何方案对象。

hashtable-ref 返回与 hashtable 中的键关联的值。如果没有值与 hashtable 中的键相关联，则 hashtable-ref 将返回默认值。

```
（定义 p1 （cons ’a ’b））
（define p2 （cons ’a ’b））

（define eqht （make-eq-hashtable））
（hashtable-set！ eqht p1 73）
（hashtable-ref eqht p1 55） ⇒ 73
（hashtable-ref eqht p2 55） ⇒ 55

（define equalht （make-hashtable equal-hash equal？））
（hashtable-set！ equalht p1 73）
（hashtable-ref equalht p1 55） ⇒ 73
（hashtable-ref equalht p2 55） ⇒ 73
```

过程：（哈希表包含？hashtable key）
返回：#t如果 hashtable 中存在键的关联，#f否则
库：（rnrs hashtables），（rnrs）

key 应该是哈希表的哈希和等价函数的适当键。

```
(定义ht (make-eq-hashtable))
(定义 p1 (cons 'a 'b))
(定义 p2 (cons 'a 'b))
(hashtable-set! ht p1 73)
(hashtable-contains? ht p1) ⇒ #t
(hashtable-contains? ht p2) ⇒ #f
```

过程：（哈希表更新！哈希表键过程默认值）
返回：未指定的
库：（rnrs 哈希表）、（rnrs）

哈希表必须是可变哈希表。key 应该是哈希表的哈希和等价函数的适当键。默认值可以是任何方案对象。过程应接受一个参数，应返回一个值，并且不应修改哈希表。

哈希表更新！将过程应用于与哈希表中的键关联的值，或者，如果没有值与哈希表中的键相关联，则应用到默认值。如果过程返回，则 hashtable-update！ 将键与过程返回的值相关联，替换旧的关联（如果有）。

不验证它是否接收正确类型的参数的哈希表更新版本可能定义如下。

```
(定义哈希表更新!
(lambda (ht key proc value)
(hashtable-set! ht key
(proc (hashtable-ref ht key value)))))
```

但是，通过避免多次哈希计算和哈希表查找，实现可能能够更有效地实现哈希表更新！

```
(定义ht (make-eq-hashtable))
(hashtable-update! ht 'a
(lambda (x) (* x 2))
55)
(hashtable-ref ht 'a 0) ⇒ 110
(hashtable-update! ht 'a
(lambda (x) (* x 2))
0)
(hashtable-ref ht 'a 0) ⇒ 220
```

过程：（哈希表删除！hashtable key）
返回：未指定的
库：（rnrs hashtables）、（rnrs）

哈希表必须是可变哈希表。key 应该是哈希表的哈希和等价函数的适当键。

哈希算删除！从哈希表中删除密钥的任何关联。

```
(定义ht (make-eq-hashtable))
(定义 p1 (cons 'a 'b))
(定义 p2 (cons 'a 'b))
(hashtable-set! ht p1 73)
(hasht-contains? ht p1) ⇒ #t
(hashtable-delete! ht p1)
(hashtable-contains? ht p1) ⇒ #f
(hasht-contains? ht p2) ⇒ #f
(hasht-delete! ht p2)
```

过程：（哈希表大小哈希表）
返回：哈希表
库中的条目数：（rnrs hashtables），（rnrs）

```
(定义ht (make-eq-hashtable))
(定义 p1 (cons 'a 'b))
(定义 p2 (cons 'a 'b))
(hashtable-size ht) ⇒ 0
(hashtable-set! ht p1 73)
(hashtable-size ht) ⇒ 1
(hashtable-delete! ht p1)
(hashtable-size ht) ⇒ 0
```

过程：（hashtable-copy hashtable）
过程：（hashtable-copy hashtable mutable？）
返回：一个新的哈希表，其中包含与哈希表
库相同的条目：（rnrs hashtables），（rnrs）

如果可变？存在而不是假，则副本是可变的;否则，副本是不可变的。

```
(定义ht (make-eq-hashtable))
(定义 p1 (cons 'a 'b))
(hashtable-set! ht p1 “c”)
(define ht-copy (hashtable-copy ht))
(hashtable-mutable? ht-copy) ⇒ #f
(hashtable-delete! ht p1)
(hashtable-ref ht p1 #f) ⇒ #f
```

```
(hashtable-delete! ht-copy p1) ⇒ exception: not mutable
(hashtable-ref ht-copy p1 #f) ⇒ “c”
```

程序：（哈希表清除！哈希算）
程序：（哈希表清除！hashtable size）
返回：未指定的
库：（rnrs hashtables）、（rnrs）

哈希表必须是可变哈希表。如果提供了 size，则它必须是非负精确整数。

哈希算清楚！从哈希表中删除所有条目。如果提供了 size，则哈希表将重置为给定大小，就像由具有 size 参数大小的哈希表创建操作之一新创建的一样。

```
(定义ht (make-eq-hashtable))
(定义 p1 (cons 'a 'b))
(定义 p2 (cons 'a 'b))
(hashtable-set! ht p1 “first”)
(hashtable-set! ht p2 “second”)
(hashtable-size ht) ⇒ 2
(hashtable-clear! ht)
(hashtable-size ht) ⇒ 0
(hashtable-ref ht p1 #f) ⇒ #f
```

过程：（hashtable-keys hashtable）
返回：一个包含哈希表库中键的
向量：（rnrs hashtables），（rnrs）

这些键可以按任何顺序出现在返回的向量中。

```
(定义ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(hashtable-set! ht p1 “one”)
(hashtable-set! ht p2 “two”)
(hashtable-set! ht 'q “three”)
(hashtable-keys ht) ⇒ #(a .b) q (a .b))
```

过程：（hashtable-entry hashtable）
返回：两个向量：一个键和一个值
库：（rnrs hashtables），（rnrs）

哈希表条目返回两个值。第一个是包含哈希表键的向量，第二个是包含相应值的向量。键和值可以按任意顺序显示，但对于键和相应的值，顺序是相同的。

```
(定义ht (make-eq-hashtable))
(define p1 (cons 'a 'b))
(define p2 (cons 'a 'b))
(hashtable-set! ht p1 "one")
(hashtable-set! ht p2 "two")
(hashtable-set! ht 'q "three")
(hashtable-entries ht) ⇒ #((a .b) q (a .b))
  #("two" "three" "one")
```

第 6.14 节. 枚举

枚举是有序的符号集，通常用于命名和操作选项，就像创建文件时可能指定的缓冲区模式和文件选项一样。

语法： (定义枚举名称 (符号 ...) 构造函数)
库： (rnrs 枚举)， (rnrs)

定义枚举表单是一个定义，可以出现在任何其他定义可以出现的任何位置。

定义枚举语法创建一个新的枚举集，其中指定的符号按指定顺序组成枚举的宇宙。它定义了一种以名称命名的新句法形式，可用于验证符号是否在宇宙中。如果 x 在宇宙中，(名称 x) 的计算结果为 x。如果 x 不在宇宙中，则这是语法冲突。

define-enumeration 还定义了由构造函数命名的新语法形式，该表单可用于创建枚举类型的子集。如果 x ... 在宇宙中，(构造函数 x ...) 计算为包含 x ... 的枚举集。否则，这是语法冲突。同一符号可能在 x ... 中出现多次，但生成的集仅包含该符号的一次匹配项。

```
(定义枚举天气元素
(热 暖 冷 晴 雨 雪 多风)
天气)

(天气元素热) ⇒ 热
(天气元素乐趣) ⇒ 语法违规
```

```
（天气热晴风） ⇒ #<枚举集>
（枚举集>列表 （天气雨冷雨）） ⇒ （冷雨）
```

过程：（生成枚举符号列表）
返回：枚举集
库：（rnrs 枚举）、（rnrs）

此过程将创建一个新的枚举类型，其 universe 由符号列表的元素组成，这些元素必须是符号列表，按它们在列表中首次出现的顺序排列。它将新枚举类型的宇宙作为枚举集返回。

```
（定义位置（生成枚举’（顶部底部在顶部旁边））
（枚举集>列表位置） ⇒ （上下角旁边）
```

过程：（枚举集构造函数枚举集）
返回：枚举集构造过程
库：（rnrs枚举），（rnrs）

此过程返回一个过程 p，该过程可用于创建枚举集的宇宙的子集。p 必须传递一个符号列表，并且列表的每个元素都必须是枚举集宇宙的一个元素。p 返回的枚举集包含它传递的列表中的所有且仅包含符号。如果枚举集的宇宙包含枚举集的宇宙包含这些元素，则 p 返回的值可能包含不在枚举集中的元素。

```
（定义 e1 （make-enumeration ’（一二三四））
（定义 p1 （枚举集构造函数 e1））
（定义 e2 （p1 ’（一个三））
（枚举集>列表 e2） ⇒ （一个三）
（定义 p2 （枚举集构造函数 e2））
（定义 e3 （p2 ’（一二四））
（枚举集>列表 e3） ⇒ （一二四）
```

过程：（枚举集-宇宙枚举集）
返回：枚举集的宇宙，作为枚举集
库：（rnrs枚举），（rnrs）

```
（定义 e1 （make-enumeration ’（a b c a b c d））
（enum-set->list （enum-set-universe e1）） ⇒ （a b c d）
（define e2 （（enum-set-constructor e1） ’（c））
（enum-set->list （enum-set-universe e2）） ⇒ （a b c d）
```

过程：（枚举集>列表枚举集）
返回：枚举集
库的元素列表：（rnrs枚举），（rnrs）

结果列表中的符号按创建枚举集的枚举类型时给定的顺序显示。

```
(定义 e1 (make-enumeration '(a b c a b c d)))
(枚举集>列表 e1) ⇒ (a b c d)
(定义 e2 ((枚举集构造函数 e1) '(d c a b)))
(枚举集>列表 e2) ⇒ (a b c d)
```

过程：（枚举集子集? enum-set1 enum-set2）
返回：#t如果 enum-set1 是 enum-set2 的子集，则#f否则
库：（rnrs enums）、（rnrs）

枚举集枚举集 $_1$ 是枚举集枚举集 $_2$ 的子集，当且仅当枚举集 $_1$ 的宇宙是枚举集 $_2$ 的宇宙的子集，并且枚举集 $_1$ 的每个元素都是枚举集 $_2$ 的元素。

```
(定义 e1 (make-enumeration '(a b c)))
(定义 e2 (make-enumeration '(a b c d e)))
(enum-set-subset? e1 e2) ⇒ #t
(enum-set-subset? e2 e1) ⇒ #f
(define e3 ((enum-set-constructor e2) '(a c)))
(enum-set-subset? e3 e1) ⇒ #f
(enum-set-subset? e3 e2) ⇒ #t
```

过程：（枚举集=? enum-set1 enum-set2）
返回：如果 enum-set1 和 enum-set2 是等效的，则#t，否则#f
库：（rnrs enums）、（rnrs）

如果两个枚举集是另一个枚举集的子集，则两个枚举集枚举集 1 和枚举集 $_2$ 是等效的。

```
(定义 e1 (make-enumeration '(a b c d)))
(定义 e2 (make-enumeration '(b d c a)))
(enum-set=? e1 e2) ⇒ #t
(define e3 ((enum-set-constructor e1) '(a c)))
(define e4 ((enum-set-constructor e2) '(a c)))
(enum-set=? e3 e4) ⇒ #t
(enum-set=? e3 e2) ⇒ #f
```

枚举集=？可以用枚举集子集来定义吗？如下所示。

```
（定义枚举集=？
（lambda （e1 e2）
（and （enum-set-subset？ e1 e2） （enum-set-subset？ e2 e1））））
```

过程：（枚举集成员？符号枚举集）
返回： #t 如果符号是枚举集的元素，#f 否则
库：（rnrs 枚举）、 （rnrs）

```
（定义 e1 （make-enumeration '（a b c d e））)
（定义 e2 （（enum-set-constructor e1） '（d b））)
（enum-set-member？'c e1） ⇒ #t
（枚举集成员？'c e2） ⇒ #f
```

过程：（enum-set-union enum-set1 enum-set2）
返回：enum-set1 和 enum-set2
过程的并集：（enum-set-intersection enum-set1 enum-set2）
返回：的交集enum-set1 和 enum-set2
过程： （enum-set-difference enum-set1 enum-set2）
返回： enum-set1 和 enum-set2
库的区别： （rnrs enums）， （rnrs）

枚举集 $_1$ 和枚举集 $_2$ 必须具有相同的枚举类型。每个过程返回一个新的枚举集，表示两个集的并集、交集或差集。

```
（定义 e1 （make-enumeration '（a b c d））)
（定义 e2 （（枚举集-集合-构造函数 e1） '（a c））)
（定义 e3 （（枚举集-集构造函数 e1） '（b c））)
（枚举集>列表 （枚举集-集-> ⇒ 列表
（枚举集-集->列表 （枚举集-集->）） ⇒ （c）
（枚举集->列表 （枚举集-差额 e2 e3）） ⇒ （a）
（枚举集->列表 （枚举集-差额 e3 e2）） ⇒ （b）
（定义 e4 （make-枚举 '（b d c a））））
（枚举集并集 e1 e4）⇒ 异常：不同的枚举类型
```

过程：（枚举集补集枚举集）
返回：枚举集相对于其宇宙
库的补集：（rnrs枚举），（rnrs）

```
（定义 e1 （make-enumeration '（a b c d））)
（enum-set->list （enum-set-complement e1）） ⇒ （）
```

```
(定义 e2 ((enum-set-constructor e1) '(a c)))
(enum-set->list (enum-set-complement e2)) ⇒ (b d)
```

过程: (枚举集-投影枚举-$_{set1}$ 枚举-$_{set2}$)
返回: 枚举集 $_1$ 投影到枚举集 $_2$
库的宇宙中: (rnrs 枚举), (rnrs)

枚举集 $_1$ 中不在枚举集 $_2$ 的宇宙中的任何元素都将被删除。结果与枚举集 $_2$ 的枚举类型相同。

```
(定义 e1 (make-enumeration '(a b c d)))
(定义 e2 (make-enumeration '(a b c d e f g)))
(定义 e3 ((枚举集-构造函数 e1) '(a d)))
(定义 e4 ((枚举集-集-构造函数 e2) '(a c e g)))
(枚举集>列表 (枚举集-集-投影 e4 e3)) ⇒ (a c)
(枚举集->列表
(枚举集-集-联合 e3
(枚举集-集-投影 e4 e3))) ⇒ (a c d)
```

过程: (枚举集索引器枚举集)
返回: 一个过程, 返回枚举集
库宇宙中符号的索引: (rnrs 枚举)、 (rnrs)

枚举集索引器返回一个过程 p, 当应用于枚举集的宇宙中的符号时, 将返回构成宇宙的有序符号集中的符号的索引(从零开始)。如果应用于不在宇宙中的符号, p 将返回#f。

```
(定义 e1 (make-enumeration '(a b c d)))
(定义 e2 ((枚举集构造函数 e1) '(a d)))
(定义 p (枚举集索引器 e2))
(列表 (p 'a) (p 'c) (p 'e)) ⇒ (0 2 #f)
```

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

http://www.scheme.com