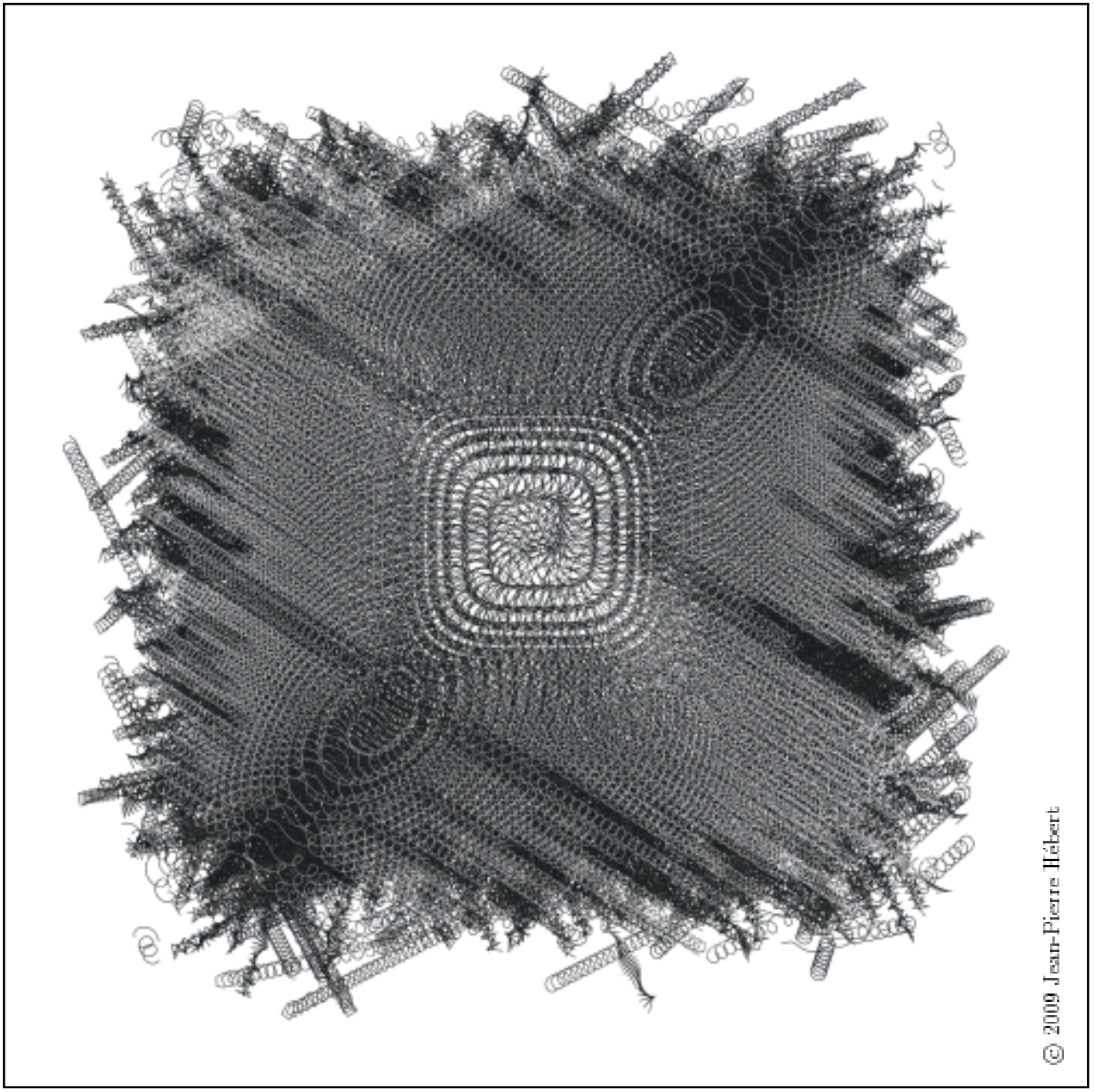


第 12 章。扩展示例

本章介绍了一系列程序，这些程序执行的任务比本书前面几章中的大多数示例都要复杂。它们说明了各种编程技术，并演示了特定的编程风格。

本章的每个部分都详细描述了一个程序，并给出了其使用示例。接下来是代码清单。每个部分的末尾都是练习，旨在激发对程序的思考并提出可能的扩展建议。这些练习通常比第2章和第3章中的练习更困难，其中一些是主要项目。

第 12.1 节提供了一个简单的矩阵乘法包。它演示了一组几乎可以用任何语言编写的过程。它最有趣的功能是，所有乘法运算都是通过调用单个泛型过程 mul 来执行的，该泛型过程根据其参数的维度调用适当的帮助过程，并且它动态分配适当大小的结果。第 12.2 节介绍了一种合并排序算法，用于根据任意谓词对列表进行排序。第 12.3 节描述了用于构造集合的语法形式。它演示了从集合表示法到方案代码的简单但有效的语法转换。第12.4节介绍了一个从C编程语言[19]借用的字数统计程序，从C翻译成Scheme。它显示字符和字符串操作、数据结构创建和操作以及基本文件输入和输出。第 12.5 节介绍了一个 Scheme 打印机，它实现了基准、写入和显示的基本版本。第12.6节提供了一个简单的格式化输出工具，类似于许多Scheme系统和其他语言中的输出工具。第12.7节为Scheme提供了一个简单的解释器，它说明了Scheme作为语言实现工具，同时为Scheme提供了非正式的操作语义，并为调查Scheme的扩展提供了有用的基础。第 12.8 节介绍了一个小型的、可扩展的抽象对象工具，它可以作为整个面向对象子系统的基础。第 12.9 节介绍了一种递归算法，用于计算输入值序列的傅里叶变换。它突出了Scheme的复杂算术的使用。第 12.10 节介绍了一个简洁的统一算法，该算法显示了如何将过程用作 Scheme 中的延续和替换（统一器）。第 12.11 节描述了多任务处理功能及其在延续方面的实现。

第 12.1 节. 矩阵和矢量乘法

此示例程序主要涉及基本编程技术。它演示了简单的算术和向量运算、使用 do 语法形式循环、基于对象类型的调度以及引发异常。

标量到标量、标量到矩阵或矩阵到矩阵的乘法由单个通用过程执行，称为 mul。mul 使用两个参数调用，它根据其参数的类型决定要执行的操作。由于标量运算使用 Scheme 的乘法过程 *，因此 mul 标量可以是任何内置数值类型（精确或不精确的复数、实数、有理数或整数）。

一个 m × n 个矩阵 A 和一个 n × p 矩阵 B 的乘积是 m × p 矩阵 C，其条目由下式定义

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}.$$

标量 x 和 m × n 个矩阵 A 的乘积是 m × n 个矩阵 C 的乘积，其条目由等式定义

Cij = xAij.

也就是说，C 的每个元素都是 x 和 A 的相应元素的乘积。向量-向量、向量-矩阵和矩阵-向量乘法可被视为矩阵-矩阵乘法的特殊情况，其中向量表示为 1 × n 或 n × 1 矩阵。

下面是几个示例，每个示例前面都有标准数学符号的等效运算。

- 标量乘以标量：

3 × 4 = 11

（穆尔 3 4）⇒ 11

- 标量乘以矢量（1 × 3 矩阵）：

$$1/2 \times (1 \quad 2 \quad 3) = (1/2 \quad 1 \quad 3/2)$$

（mul 1/2 '#（#（1 2 3））⇒ #（#（1/2 1 3/2））

- 标量时间矩阵：

$$-2 \times \begin{pmatrix} 3 & -2 & -1 \\ -3 & 0 & -5 \\ 7 & -1 & -1 \end{pmatrix} = \begin{pmatrix} -6 & 4 & 2 \\ 6 & 0 & 10 \\ -14 & 2 & 2 \end{pmatrix}$$

```
（骡子 -2
'#（#（#（3 -2 -1）
#（-3 0 -5）
#（7 -1 -1））⇒ #（#（-6 4 2）
   #（6 0 10）
   #（-14 2 2））
```

- 矢量时间矩阵：

$$(1 \quad 2 \quad 3) \times \begin{pmatrix} 2 & 3 \\ 3 & 4 \\ 4 & 5 \end{pmatrix} = (20 \quad 26)$$

```
（mul '#（#（#（1 2 3））
'#（#（2 3）
#（3 4）
#（4 5））⇒ #（#（20 26））
```

- 矩阵时间矢量：

$$\begin{pmatrix} 2 & 3 & 4 \\ 3 & 4 & 5 \end{pmatrix} \times \begin{pmatrix} 1 \\ 2 \\ 3 \end{pmatrix} = \begin{pmatrix} 20 \\ 26 \end{pmatrix}$$

```
（骡子 '#（#（2 3 4）
#（3 4 5））
'#（#（1） #（2） #（3））） ⇒ #（#（20） #（26））
```

- 矩阵乘以矩阵：

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \times \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} 14 & 20 & 26 & 32 \\ 32 & 47 & 62 & 77 \end{pmatrix}$$

```
（mul '#（#（#（1 2 3）
#（4 5 6））
'#（#（1 2 3 4）
#（2 3 4 5）
#（3 4 5 6））） ⇒ #（#（14 20 26 32）
  #（32 47 62 77））
```

mul 及其帮助程序的代码（结构为库）显示在下面。前几个定义建立了一组支持矩阵数据类型的过程。矩阵是向量的向量。包括创建矩阵的过程、访问和分配矩阵元素的过程以及矩阵谓词。遵循这些定义是 mul 本身的定义。mul 的 lambda 表达式内部是一组支持 mul 的帮助过程的定义。

mul 检查其参数的类型，并选择适当的帮助过程来完成工作。每个帮助程序都对特定类型的参数进行操作。例如，mat-sca-mul 将矩阵乘以标量。如果任一参数的类型无效或参数不兼容，例如，行或列不匹配，则 mul 或其帮助程序之一将引发异常。

```
（library （tspl matrix）
（export make-matrix matrix? matrix-rows matrix-
```

```
 columns
 matrix-ref matrix-set!  mul)
  (import  (rnrs) )

 ; make-matrix 创建一个矩阵（向量的向量）。
  (定义 make-matrix
  (lambda  (rows columns)
  (do  ([m  (make-vector rows) ]
 [i 0  (+ i 1) ])
  ( (= i rows)  m)
  (vector-set!  m i  (make-vector columns) ) ) ) )

 ;矩阵？检查其参数是否为矩阵。
 ;它不是万无一失的，但它通常足够好。
  (定义矩阵?
  (lambda  (x)
  (and  (vector?  x)
  (>  (vector-length x)  0)
  (vector?  (vector-ref x 0) ) ) ) )

 ;矩阵行返回矩阵中的行数。
  (定义矩阵行
  (lambda  (x)
  (vector-length x) ) )

 ;矩阵列返回矩阵中的列数。
  (定义矩阵列
  (lambda  (x)
  (vector-length  (vector-ref x 0) ) ) )

 ;matrix-ref 返回第 i 行的第 j 个元素。
  (define matrix-ref
  (lambda  (m i j)
  (vector-ref  (vector-ref m i)  j) ) )

 ;矩阵集！更改第 i 行的第 j 个元素。
  (定义矩阵集!
  (lambda  (m i j x)
  (vector-set!  (vector-ref m i)  j x) ) )
```

```
;mat-sca-mul 将矩阵乘以标量。
(define mat-sca-mul
(lambda (m x)
(let* ([nr (matrix-rows m)]
[nc (matrix-columns m)]
[r (make-matrix nr nc)])
(do ([i 0 (+ i 1)])
((= i nr) r)
(do ([j 0 (+ j 1)])
((= j nc))
(matrix-set! r i j (* x (matrix-ref m i j)))))))

;mat-mat-mul 在验证后将一个矩阵乘以另一个矩阵
;第一个矩阵的列数与第二
个矩阵一样多;矩阵包含行。
(定义 mat-mat-mul
(lambda (m1 m2)
(let* ([nr1 (matrix-rows m1)]
[nr2 (matrix-rows m2)]
[nc2 (matrix-columns m2)]
[r (make-matrix nr1 nc2)])
(除非 (= (matrix-columns m1) nr2) (match-error m1
m2))
(do ([i 0 (+ i 1)])
((= i nr1) r)
(do ([j 0 (+ j 1)])
((= j nc2))
(do ([k 0 (+ k 1)]
[a 0 (+ a
(* (matrix-ref m1 i k)
(matrix-ref m2 k j)))])
((= k nr2)
(matrix-set! r i j a)))))))))

;当 mul 收到 无效
时 调用 type-error 来抱怨 ;参数类型。
(定义类型错误
(lambda (什么)
```

```
 (断言违规'mul
 "不是一个数字或矩阵"
什么)))

;当 mul 收到一对
;不兼容的参数。
 (定义匹配错误
 (lambda (what1 what2)
 (assertion-violation 'mul
 "incompatible operationnds" what1
what2)))

;mul 是通用矩阵/标量乘法过程
 (定义 mul
 (lambda (x y)
 (cond
[(number? x)
 (cond
[(number? y) (* x y)]
[(matrix? y) (mat-sca-mul y x)]
[else (type-error y)])]
[(matrix? x)
 (cond
[(number? y) (mat-sca-mul x y)]
[(matrix? y) (mat-mat-mul x y)]
[else (type-error y)])]
[else (type-error x)]))))
```

练习 12.1.1

进行必要的更改以将 mul 重命名为 *。

练习 12.1.2

谓词矩阵通常就足够了，但不是完全可靠的，因为它可能会为不是矩阵的对象返回#t。特别是，它不验证所有矩

阵行都是向量，每行是否具有相同的元素数，或者元素本身是否为数字。修改矩阵？以执行这些附加检查中的每一项。

练习 12.1.3

练习 12.1.2 的另一个解决方案是定义一个矩阵记录类型，封装矩阵的向量。如果矩阵创建例程从不允许创建格式错误的矩阵记录，则矩阵记录检查是确保输入格式正确的唯一检查。定义矩阵记录类型并重新编码库以使用它。

练习 12.1.4

编写类似的加法和减法通用过程。设计一个通用调度过程或语法形式，以便不需要为每个新操作重写类型调度代码。

练习 12.1.5

此版本的 `mul` 使用向量的向量来表示矩阵。重写系统，使用嵌套列表来表示矩阵。这种变化会带来什么效率，会失去什么效率？

第 12.2 节. 排序

本节演示一种基于称为合并排序的简单技术的列表排序算法。此处定义的过程排序接受两个参数：谓词和列表，就像内置的列表排序一样。它返回一个列表，其中包含根据谓词排序的旧列表的元素。与 `list-sort` 一样，谓词

应该是一个过程，该过程需要两个参数，如果其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 否则为 false。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 sort 将执行稳定的排序;使用稳定排序时，已相对于彼此排序的两个元素将以它们在输入中出现的相同顺序出现在输出中。因此，对已排序的列表进行排序将导致不会重新排序，即使存在等效元素也是如此。

```
(排序 < '(3 4 2 1 2 5)) ⇒ (1 2 2 3 4 5)
(排序 > '(0.5 1/2)) ⇒ (0.5 1/2)
(排序 > '(1/2 0.5)) ⇒ (1/2 0.5)
(list->string
(sort char>?
(字符串>列出“硬币”)) ⇒ “声波”
```

代码还定义了一个伴随过程 merge。merge 接受一个谓词和两个排序列表，并按两个列表的元素的排序顺序返回合并列表。使用正确定义的谓词，合并也是稳定的，因为第一个列表中的项将显示在第二个列表中的项之前，除非第二个列表中的项必须首先显示。

```
(合并字符<?
'(#\a #\c)
'(#\b #\c #\d)) ⇒ (#\a #\b #\c #\c #\c #\d)
(merge <
'(1/2 2/3 3/4)
'(0.5 0.6 0.7)) ⇒ (1/2 0.5 0.6 2/3 0.7 3/4)
(list->string
(merge char>?
(字符串>列表“旧”)
(字符串>列表“脚趾”))) ⇒ “工具”
```

合并排序算法简单而优雅。输入列表被拆分为两个大致相等的子列表。这些子列表以递归方式排序，生成两个排序列表。然后，排序列表将合并为单个排序列表。递归的基本情况是一个元素的列表，该元素已经过排序。

为了减少开销，该实现以排序方式计算一次输入列表的长度，而不是在递归的每个步骤中计算 dosort。这也允许 dosort 仅通过将长度减半来隔离列表的前半部分，从而节省了分配包含一半元素的新列表的成本。因此，ls 可能包含多个元素，但只有前 n 个元素被视为列表的一部分。

```
(library (tspl sort)
(export sort merge)
(import (rnrs))

(define dosort
(lambda (pred? ls n)
(if (= n 1)
(list (car ls))
(let ([i (div n 2)])
(domerge pred?
(dosort pred? ls i)
(多索排序捕食? (list-tail ls i) (- n i)))))))

(define domerge
(lambda (pred? l1 l2)
(cond
[(null? l1) l2]
[(null? l2) l1]
[(pred? (车 l2) (车 l1)
(缺点 (汽车 l2) (domerge pred? l1 (cdr l2)))]
[else (缺点 (汽车 l1) (domerge pred? (cdr l1)
l2)))))

(define sort
```

```
(lambda (pred? l)
(if (null? l) l (dosort pred? l (length
l) ) ) ) )

(define merge
(lambda (pred? l1 l2)
(domerge pred? l1 l2) ) ) )
```

练习 12.2.1

在 dosort 中，当 n 为 1 时，为什么返回 (list (car ls)) 而不是 ls？通过将 (list (car ls)) 替换为 (if (null? (cdr ls)) ls (list (car ls)) ?

练习 12.2.2

在dosort中拆分输入列表时，如果不复制输入列表的第一部分，实际上节省了多少工作？

练习 12.2.3

如果算法具有破坏性地工作，则可以保存所有或几乎所有分配，使用set-cdr!来分离和加入列表。编写破坏性版本排序!并合并!排序和合并。确定两组过程在各种输入的分配和运行时方面的差异。

第 12.3 节. 集合构造函数

此示例描述了一个语法扩展 set-of，它允许构造表示为没有重复元素的列表的集合 [22]。它使用定义语法和语法规则将 set 表达式编译为递归表达式。扩展的代码通常与手动生成的代码一样有效。

一组表达式采用以下形式。

（一组 expr 子句 ...）

expr 根据集合子句建立的绑定来描述集合的元素...。每个子句可以采用以下三种形式之一：

1. 形式的子句（x in s）依次为 x 建立与集合 s 的每个元素的绑定。此绑定在其余子句和表达式 expr 中可见。

2. 形式为（x is e）的子句为x到e建立约束。此绑定在其余子句和表达式 expr 中可见。此形式实质上是（x in （列表 e）） 的缩写。

3. 采用任何其他形式的子句被视为谓词;这用于强制拒绝某些元素，如下面的第二个示例所示。

```
（x
的集合（x 在 ’（a b c））中）⇒ （a b c）

（x in ’（1 2 3 4）的集合
）
（偶数？ x）） ⇒ （2 4）

（集合 （缺点 x y）
（x in ’（1 2 3））
（y 是 （* x x））） ⇒ （（1 . 1） （2 . 4） （3 . 9））

（集合 （缺点 x y）
（x 在 ’（a b））
（y 在 ’（1 2））） ⇒ （（a. 1） （a . 2） （b . 1） （b
. 2））
```

一组表达式将转换为嵌套的 let（命名为 let）和 if 表达式，对应于每个表达式 is、in 或谓词子表达式。例

如，简单表达式

```
（x 的集合（x 在 ’（a b c））中）
```

转换为

```
（let loop （[set ’（a b c）]）
（if （null? set）
’（）
（let （[x （car set）]）
（set-cons x （loop （cdr set））））））
```

表达式

```
（x 的集合 （x in ’（1 2 3 4）） （甚至? x））
```

转换为

```
（let loop （[set ’（1 2 3 4）]）
（if （null? set）
’（）
（let （[x （car set）]）
（if （even? x）
（set-cons x （loop （cdr set））））
（loop （cdr set））））））
```

更复杂的表达式

```
（集合 （缺点 x y） （x in ’（1 2 3）） （y 是 （* x x））））
```

转换为

```
（let loop （[set ’（1 2 3）]）
（if （null? set）
’（）
（let （[x （car set）])
```

```
(let ([y (* x x)])
(set-cons (cons x y)
(loop (cdr set)))))
```

最后，表达式

```
(集合 (缺点 x y) (x 在 '(a b)) (y 在 '(1
2)))
```

转换为嵌套的命名 let 表达式：

```
(let loop1 ([set1 '(a b)])
(if (null? set1)
'()
(let ([x (car set1)])
(let loop2 ([set2 '(1 2)])
(if (null? set2)
(loop1 (cdr set1))
(let ([y (car set2)])
(set-cons (cons x y)
(loop2 (cdr set2)))))
```

这些是相当简单的转换，除了嵌套命名 let 表达式上的递归的基本情况因级别而异。最外层命名 let 的基本情况始终是空列表 ()，而内部命名 let 的基本情况是下一个外部命名 let 的递归步骤。为了处理这个问题，set-of 的定义使用了一个帮助语法扩展集-of-help。set-of-help 采用一个额外的表达式 base，这是当前级别递归的基本情况。

```
(库 (tspl sets)
(导出 set-cons in is)
(import (rnrs))

; set-of 使用 helper 语法扩展 set-of-help，传递它
; '()
```

```
的初始基本表达式 （define-syntax set-of
（syntax-rules （）
[（_ e m ...）
（set-of-help e ’（） m ...）]））

;帮助集识别 in、 is 和谓词表达式和
;将它们更改为嵌套的命名 let、let 和 if 表达式。
（define-syntax set-of-help
（syntax-rules （in is）
[（_ e base） （set-cons e base）]
[（_ e base （x in s） m ...）
（let loop （[set s]）
（if （null? set）
base
（let （[x （car set）]）
（set-of-help e （loop （cdr set）） m ...））））））
[（_ e base （x is y） m ...）
（let （[x y]） （set-of-help e base m ...））]
[（_ e base p m ...）
（if p （set-of-help e base m ...） base）]））

;由于 in 和 被 用作辅助关键字，因此 ；
库也必须导出它们的定义
（定义语法在
（lambda（x）
（语法违规’在“错放的辅助关键字”x））））
）
（定义语法是
（lambda（x）
（语法违规’是“错放的辅助关键字”x）））；

set-cons 返回原始集合 y，如果 x 已经在 y 中（
定义 set-cons
（lambda （x y）
（if （memv x y）
y
（cons x y））））
```

练习 12.3.1

编写一个过程，并集，该过程采用任意数量的集合（列表）作为参数，并返回集合的并集，仅使用语法形式的集合。例如：

```
（联合）⇒ （）
（工会 '（a b c））⇒ （a b c）
（工会 '（2 5 4） '（9 4 3））⇒ （2 5 9 4 3）
（工会 '（1 2） '（2 4） '（4 8））⇒ （1 2 4 8）
```

练习 12.3.2

地图的单列表版本可以（几乎）定义如下。

```
(define map1
(lambda (f ls)
(set-of (f x) (x in ls))))

(地图1 - '（1 2 3 2））⇒ （-1 -3 -2）
```

为什么这不起作用？可以更改哪些内容以使其正常工作？

练习 12.3.3

设计一个不同的 set-cons 定义，以某种排序顺序维护集合，使 set 成员资格的测试以及 set-cons 本身可能更有效。

第 12.4 节. 字频计数

此程序演示了几种基本的编程技术，包括字符串和字符操作、文件输入/输出、数据结构操作和递归。该程序改编自C编程语言[19]的第6章。使用这个特定示例的一个原因是显示C程序在几乎字面上转换为Scheme时的外观。

Scheme程序和原始C程序之间的一些差异值得注意。首先，方案版本对文件输入和输出采用不同的协议。它不是隐式使用标准输入和输出端口，而是要求传入文件名，从而演示文件的打开和关闭。其次，过程 get-word 返回以下三个值之一：字符串（单词）、非字母字符或 eof 值。原始的C版本返回字母标志（表示已阅读单词）或非字母字符。此外，C版本传入了要填充的字符串，并限制了字符串中的字符数;Scheme 版本构建了一个所需长度的新字符串（单词中的字符保存在列表中，直到找到单词的末尾，然后转换为具有 list-> 字符串的字符串）。最后，char-type 使用原始 Scheme 字符谓词 char-alphabetical? 和 char-numeric? 来确定字符是字母还是数字。

主程序 frequency 将输入文件名和输出文件名作为参数，例如（频率“pickle” “freq.out”）将文件“pickle”中每个单词的频率计数输出到文件“freq.out”。当频率从输入文件中读取单词时，它会将它们插入到二叉树结构中（使用二进制排序算法）。通过递增与每个单词关联的计数来记录重复条目。到达文件末尾后，程序将遍历树，打印每个单词及其计数。

假定文件“pickle”包含以下文本。

彼得·派珀（Peter Piper）采摘了一小堆腌制辣椒。
彼得·派珀（Peter Piper）采摘了一小撮腌制辣椒。
如果彼得·派珀（Peter Piper）采摘了一堆腌制辣椒，
那么彼得·派珀（Peter Piper）采摘的腌制辣椒在哪里？

然后，在键入（频率“pickle”“freq.out”）之后，文件“freq.out”应包含以下内容。

```
1 A
1 如果
4 彼得
4 派珀
1 其中
2 一
4 的
4 啄
4 辣椒
4 采摘
4 腌
1 s
1
```

字数统计程序的代码被构建为顶级程序，在修订[6]报告[25]的非规范附录的脚本章节中推荐了脚本标题。它从命令行获取输入和输出文件的名称。

```
#!/usr/bin/env scheme-script
 (import (rnrs))

;;;如果 p 上的下一个字符是字母，则 get-word 读取单词
;;;从 p 并在字符串中返回它。如果字符不是
;;;字母，get-word 返回字符（在 eof 上，eof 对象）。
 (定义 get-word
 (lambda (p)
 (let ([c (get-char p)])
 (if (eq? (字符型 c)'letter)
 (list->string
 (let loop ([c c])
 (cons
c
 (if (memq (char-type (lookahead-char p))
'(letter digit))
```

```
 (loop (get-char p))
' () ) ) )


c) ) )
;;;首先对 eof 对象进行 char-type 测试，因为 eof-object
;;;可能不是字符字母的有效参数？还是字符数字？
;;;它返回 eof 对象、符号字母、符号数字
;;;或参数本身，如果它不是字母或数字。
 (定义 char-type
 (lambda (c)
 (cond
[ (eof-object? c) c]
[ (char-alphabetic? c) 'letter]
[ (char-numeric? c) 'digit]
[else c]) ) )

;;;树节点表示为具有四个字段的记录类型：word，
;;;左，右和计数。只有一个字段，单词，由
;;;构造函数过程 make-tnode 的参数。其余的
;;;字段由构造函数初始化，并由后续
的 ;;;操作。
 (define-record-type tnode
 (fields (immutable word)
 (mutable left)
 (mutable right)
 (mutable count) )
 (protocol
 (lambda (new)
 (lambda (word)
 (new word '() '() 1) ) ) )

;;;如果该单词已存在于树中，则树将递增其
;;;计数。否则，将创建一个新的树节点并将其放入
;;;树。在任何情况下，都会返回新的或修改过的树。
 (定义树
 (lambda (节点词)
 (cond
[ (null? node) (make-tnode word) ]
[ (string=? word (tnode-word node) )
```

```
 (tnode-count-set! node (+ (tnode-count node) 1))
node]
[(string<? word (tnode-word node))
 (tnode-left-set! node (tree (tnode-left node)
word))

[else
 (tnode-right-set! node (tree (tnode-right node)
word))
节点]))))

;;;tree-print 按“顺序”打印树，即左子树
;;;然后是节点，然后是右子树。对于每个单词，计数和
;;;字印在一行上。
 (定义 tree-print
 (lambda (node p)
 (除非 (null? node)
 (tree-print (tnode-left node) p)
 (put-datum p (tnode-count node))
 (put-char p #\space)
 (put-string p (tnode-word node))
 (newline p)
 (tree-print (tnode-right node) p))))

;;;频率是驱动程序例程。它打开文件，读取
;;;单词，并将它们输入到树中。当输入端口
;;;到达文件末尾，它会打印树并关闭端口。
 (定义频率
 (lambda (infn outfn)
 (let ([ip (open-file-input-port infn (file-
options)
 (buffer-mode block) (native-transcoder)))
[op (open-file-output-port outfn (file-options)
 (buffer-mode block) (native-transcoder))])
 (let loop ([root '()])
 (let ([w (get-word ip)])
 (cond
[(eof-object? w) (tree-print root op)]
[(string? w) (loop (treeroot w))]
```

```
[else （loop root）]）））
（close-port ip）
（close-port op））））

（除非 （= （length （command-line）） 3）
（put-string （current-error-port） “usage: ”）
（put-string （current-error-port） （car （command-line））））
（put-string （current-error-port） “ input-filename output-filename\n”）
（exit #f））

（frequency （cadr （command-line）） （caddr （命令行））））
```

练习 12.4.1

在前面显示的输出文件中，大写单词出现在输出文件中其他单词之前，并且大写字母 A 未被识别为与小写字母 a 相同的单词。修改树以使用字符串比较的不区分大小写的版本，以便不会发生这种情况。

练习 12.4.2

“单词”出现在文件“freq.out”中，尽管它实际上只是收缩Where's的一部分。调整获取词以允许嵌入单引号。

练习 12.4.3

修改此程序以“清除”某些常用词，例如a，an，the，is，of等，以减少长输入文件的输出量。尝试设计其他方法来减少无用的输出。

练习 12.4.4

get-word 缓冲列表中的字符，为每个字符分配一对新字符（带缺点）。通过使用字符串缓冲字符来提高效率。设计一种方法，允许字符串在必要时增长。[提示：使用字符串追加或字符串输出端口。

练习 12.4.5

树实现的工作原理是创建树，然后填充它们的左字段和右字段。这需要许多不必要的分配。重写树过程，完全避免树左集！和树右集！

练习 12.4.6

重新编码程序以使用哈希表（第6.13节）代替二叉树，并比较大型输入文件上新旧程序的运行时间。哈希表总是更快还是总是更慢？是否有盈亏平衡点？盈亏平衡点是取决于文件的大小还是取决于文件的其他特征？

第 12.5 节. 方案打印机

打印 Scheme 对象可能看起来是一个复杂的过程，但实际上，基本的打印机非常简单，如本示例所示。放置基准、写入和显示都由相同的代码实现。复杂的打印机通常支持各种打印机控件并处理循环对象的打印，但这里给出的是完全基本的。

该程序的主要驱动程序是一个过程 wr，它采用一个对象来打印 x、一个标志 d? 和一个端口 p。如果代码要显示对象，则#t标志 d? （用于显示），否则#f。d? 标志仅

对字符和字符串重要。回想一下第 7.8 节中显示的不带括引号的字符串和不带 #\ 语法的字符。

写入和显示的入口点处理第二个（端口）参数的可选性，在未提供端口参数时传递当前输出端口的值。

过程、端口和文件结尾对象打印为 #<procedure>、#<port> 和 #<eof>。无法识别的值类型显示为 #<未知>。因此，例如，某些特定于实现类型的哈希表、枚举集和对象都将打印为 #<未知>。

```
（库 （tspl 打印机）
（导出 put-datum 写入显示）
（导入 （除了 （rnrs） put-datum 写入显示））；

；在此处定义这些以避免混淆 paren-balancers
（定义 lparen #\（）
（定义 rparen #\））；

wr 是驱动程序，调度在 x
的类型上 （定义 wr
（lambda （x d? p）
（cond
[（符号? x） （put-string p （symbol->string x））]
[（pair? x） （wrpair x d? p）]
[（number? x） （put-string p （number->string x））]
[（null? x） （put-string p “（）”）]
[（boolean? x） （put-string p （if x “#t”
“#f”））]
[（char? x） （if d? （推查 p x）（wrchar x p）]。
[（字符串? x） （如果 d? （put-string p x）（wrstring x
p）]。
[（vector? x） （wrvector x d? p）]
[（bytevector? x） （wrbytevector x d? p）]
[（eof-object? x） （put-string p “#<eof>”）]
[（port? x） （put-string p “#<port>”）]
[（procedure? x） （put-string p “#<procedure>”）]
```

```
[else (put-string p "#<unknown>") ) ) )

;wrpair 处理对和非空列表
(定义 wrpair
(lambda (x d? p)
(put-char p lparen)
(let loop ([x x])
(wr (car x) d? p)
(cond
[(pair? (cdr x)) (put-char p #\space) (循环 (cdr x)) ]
[(空? (cdr x)]
[else (put-string p ". ") (wr (cdr x) d? p)) ) )
(put-char p rparen)) )

;wrchar 处理字符。仅在 d? is #f.
(define wrchar
(lambda (x p)
(put-string p "#\\")
(cond
[(assq x '(#\alarm . "警报") (#\退格 . "退格")
(#\delete . "delete") (#\esc . "esc")
(#\newline . "换行符") (#\nul . "nul")
(#\page . "page") (#\return . "return")
(#\space . "空格") (#\tab . "tab")
(#\vtab . "vtab")) ) ) =>
(lambda (a) (put-string p (cdr a)) )]
[else (put-char p x)]) )

;扭绳处理字符串。仅在 d? is #f.
(define wrstring
(lambda (x p)
(put-char p #\")
(let ([n (string-length x)])
(do ([i 0 (+ i 1)])
((= i n))
(let ([c (string-ref x i)])
(case c
```

```
[ (#\alarm)  (put-string p "\\a") ]
[ (#\backspace)  (put-string p "\\b") ]
[ (#\newline)  (put-string p "\\n") ]
[ (#\page)  (put-string p "\\f") ]
[ (#\return)  (put-string p "\\r") ]
[ (#\tab)  (put-string p "\\t") ]
[ (#\vtab)  (put-string p "\\v") ]
[ (#\")  (put-string p "\\\") ]
[ (#\\)  (put-string p "\\\\") ]
[else  (put-char p c) ]) ) ) )
 (put-char p #\") ) ) )

 (define wrvector
 (lambda  (x d?  p)
 (put-char p #\#)
 (let  ([n  (vector-length x) ])
 (do  ([i 0  (+ i 1) ] [sep lparen #\space])
 ( (= i n) )
 (put-char p sep)
 (wr  (vector-ref x i)  d?  p) ) )
 (put-char p rparen) ) )

 (define wrbytevector
 (lambda  (x d?  p)
 (put-string p "#vu8")
 (let  ([n  (bytevector-length x) ])
 (do  ([i 0  (+ i 1) ] [sep lparen #\space])
 ( (= i n) )
 (put-char p sep)
 (wr  (bytevector-u8-ref x i)  d?  p) ) )
 (put-char p rparen) ) )

;在提供
端口时调用 check-and-wr （定义 check-and-wr
 (lambda  (who x d?  p)
 (除非  (和  (output-port?  p)  (textual-port?  p) )
 (断言违规谁 "无效参数"  p) )
 (wr x d?  p) ) ) ;
```

```
;put-datum 调用 wr with d? 设置为 #f
(定义 put-da
(lambda (p x)
(check-and-wr 'put-datum x #f p) ) ) ;

用 d 写调用 wr? 设置为 #f
(define write
(case-lambda
[(x) (wr x #f (current-output-port) )]
[(x p) (check-and-wr 'write x #f p)]) )

;显示调用 wr 与 d? 设置为 #t
(定义显示
(case-lambda
[(x) (wr x #t (current-output-port) )]
[(x p) (check-and-wr 'display x #t p) ) ) )
```

练习 12.5.1

数字是在数字>字符串的帮助下打印的。正确打印所有 Scheme 数字类型（尤其是不准确的数字）是一项复杂的任务。但是，处理精确的整数和比率相当简单。修改代码以直接打印精确的整数和比率数字（不带数字>字符串），但继续对不准确和复数使用数字>字符串。

练习 12.5.2

修改 wr 及其帮助程序，以将其输出定向到内部缓冲区而不是端口。使用修改后的版本实现过程对象>字符串，该过程与数字>字符串一样，返回包含其输入的打印表示形式的字符串。例如：

```
(对象>字符串'(a b c)) ⇒ "(a b c)"
(object->string "hello") ⇒ "\"hello\""
```

您可能会惊讶于这种更改是多么容易。

练习 12.5.3

某些符号未通过 wr 正确打印，包括以数字开头或包含空格的符号。修改 wr 以调用 wrsymbol 帮助程序，该帮助程序根据需要使用十六进制标量转义来处理此类符号。十六进制标量转义采用 #\xn; 的形式，其中 n 是用十六进制表示法表示的字符的 Unicode 标量值。请参阅第 458 页上符号的语法，以确定何时需要十六进制标量转义。

第 12.6 节. 格式化输出

通常需要打印包含 Scheme 对象的打印表示形式的字符串，尤其是数字。使用 Scheme 的标准输出例程执行此操作可能很繁琐。例如，第 12.4 节的树形打印过程需要对输出例程进行四次调用，以打印一条简单的单行消息：

```
(put-datum p (tnode-count node))
(put-char p #\space)
(put-string p (tnode-word node))
(换行符 p)
```

本节中定义的格式化输出工具允许将这四个调用替换为下面对 fprintf 的单个调用。

```
(fprintf p "~s ~a~%" (tnode-count node) (tnode-word node))
```

fprintf 需要一个端口参数、一个控制字符串和无限数量的附加参数，这些参数将按照控制字符串的指定插入到

输出中。在示例中，首先写入（tnode-count 节点）的值，以代替 ~s。这后面是一个空格和显示的值（tnode-word node），代替~a。输出中的 ~% 将替换为换行符。

过程 printf（本节中也定义了该过程）与 fprintf 类似，只是不需要端口参数，并且输出将发送到当前输出端口。

~s、~a 和 ~% 是格式指令;~s 导致控制字符串之后的第一个未使用的参数通过写入输出，~a 导致第一个未使用的参数通过显示打印，~% 只是导致打印换行符。下面 fprintf 的简单实现只能识别另一个格式指令 ~~，该指令在输出中插入一个波浪号。例如

```
(printf "字符串 ~s 显示为 ~~.~%" "~")
```

指纹

字符串“~”显示为 ~。

```
(library (tspl formatted-output)
(export printf fprintf)
(import (rnrs));

dofmt 完成所有的工作。它循环遍历控制字符串
;识别格式指令并打印所有其他字符
;没有解释。控件字符串末尾的波浪号是
;被视为普通角色。没有检查是否正确
;输入。指令可以用小写或大写给出。
(define dofmt
(lambda (p cntl args)
(let ([nmax (- (string-length cntl) 1)])
(let loop ([n 0] [a args])
(if (<= n nmax)
(let ([c (string-ref cntl n)])
(if (and (char=? c #\~) (< n nmax))
```

```
 (case (string-ref cntl (+ n 1)))
[(#\a #\A)
 (display (car a) p)
 (loop (+ n 2) (cdr a)))
[(#\s #\S)
 (write (car a) p)
 (循环 (+ n 2) (cdr a))]
[(#\%)
 (换行符 p)
 (循环 (+ n 2) a)]
[(#\~)
 (put-char p #\~) (loop (+ n 2) a)]
[else
 (put-char p c) (loop (+ n 1) a)])
 (begin
 (put-char p c)
 (loop (+ n 1) a)))

;printf 和 fprintf 的不同之处仅在于 fprintf 传递其
;端口参数到 dofmt，而 printf 传递当前输出
;港口。
 (define printf
 (lambda (control . args)
 (dofmt (current-output-port) control args)))

 (define fprintf
 (lambda (p control . args)
 (dofmt p control args))))
```

练习 12.6.1

向代码中添加错误检查，以查找无效的端口参数（fprintf）、无效的波浪号转义以及额外或缺少的参数。

练习 12.6.2

使用可选的 radix 参数进行数字>字符串，增强 printf 和 fprintf，并支持以下新格式指令：

一. ~b 或 ~B：以二进制形式打印下一个未使用的参数，该参数必须是数字；

二. ~o 或 ~O：打印下一个未使用的参数，该参数必须是数字，以八进制形式;和

（二）~x 或 ~X：以十六进制打印下一个未使用的参数，该参数必须是数字。

例如：

```
(printf "#x~x #o~o #b~b~%" 16 8 2)
```

将打印

```
#x10  #o10  #b10
```

练习 12.6.3

添加一个“间接”格式指令 ~@，该指令处理下一个未使用的参数（必须是字符串），就好像它被拼接到当前格式字符串中一样。例如：

```
(printf "--- ~@ ---" "> ~s <" '(a b c))
```

将打印

```
---> (a b c) <---
```

练习 12.6.4

实现格式，fprintf 的一个版本，将其输出放入字符串中，而不是写入端口。利用练习 12.5.2 中的对象>字符串来支持 ~s 和 ~a 指令。

```
(let ([x 3] [y 4])
(格式 "~s + ~s = ~s" x y (+ x y))) ⇒ "3 + 4 = 7"
```

练习 12.6.5

不要使用对象>字符串，而是使用字符串输出端口定义格式。

练习 12.6.6

修改 format、fprintf 和 printf 以允许在 ~a 和 ~s 格式指令中的波浪号之后指定字段大小。例如，指令 ~10 将导致下一个未使用的参数插入到大小为 10 的字段中的左对齐输出中。如果对象需要的空格多于指定的数量，则允许它扩展到字段之外。

```
(let ([x 'abc] [y '(def)])
(格式 "(cons '~5s '~5s) = ~5s"
x y (cons x y))) ⇒ "(cons 'abc '(def)) = (abc def)"
```

[提示：以递归方式使用格式。

第 12.7 节. 方案的元循环解释器

本节中描述的程序是Scheme的元循环解释器，即它是Scheme中编写的Scheme的解释器。解释器显示，当核心

结构独立于其语法扩展和基元时，Scheme 是多么小。它还说明了可以同样适用于Scheme以外的语言的解释技术。

口译员的相对简单性有些误导。用Scheme编写的Scheme的解释器可能比用大多数其他语言编写的解释器简单得多。以下是为什么这个更简单的几个原因。

- 之所以能够正确处理尾部调用，只是因为解释器中的尾部调用由主机实现正确处理。所需要的只是解释器本身是尾递归的。
- 解释型代码中的一流过程由解释器中的一流过程实现，而解释器又由宿主实现支持。
- 使用 call/cc 创建的一等延续由主机实现的 call/cc 提供。
- 原始过程（如 cons 和 assq）以及服务（如存储管理）由主机实现提供。

将解释器转换为以 Scheme 以外的语言运行可能需要显式支持其中部分或全部项目。

解释器将词法绑定存储在环境中，该环境只是一个关联列表（请参阅第 165 页）。计算 lambda 表达式会导致在保存环境和 lambda 主体的变量范围内创建过程。该过程的后续应用将新绑定（实际参数）与保存的环境相结合。

解释器仅处理第 3.1 节中描述的核心语法形式，并且它仅识别少数基元过程的绑定。它不执行任何错误检查。

```
 (解释3) ⇒ 3

 (interpret '(cons 3 4)) ⇒ (3 . 4)

 (interpret
'((lambda (x . y)
 (list x y))
'a 'b 'c 'd)) ⇒ (a (b c d))

 (interpret
'((call/cc (lambda (k) k))
 (lambda (x) x))
 "HEY!")) ⇒ "嘿!"

 (interpret
'((lambda (memq)
 (memq memq 'a '(b c a d e)))
 (lambda (memq x ls)
 (if (null? ls) #f
 (if (eq? (car ls) x)
ls
 (memq memq memq x (cdr ls)))))))) ⇒ (a d e)

 (interpret
'((lambda (reverse)
 (set! reverse
 (lambda (ls new)
 (if (null? ls)
new
 (reverse (cdr ls) (cons (car ls) new)))))
 (reverse '(a b c d e) '())))
#f ⇒ (e d c b a)

 (library (tspl interpreter)
 (export interpret)
 (import (rnrs) (rnrs mutable-pairs));

primitive-environment 包含少量的原语
; 过程; 它可以很容易地用额外的原语进行扩展。
```

```
 (定义原始环境
’( (apply . , apply) (assq . , assq) (call/cc . ,
call/cc)
 (car . , car) (cadr . , cadr) (caddr . , caddr)
 (cadddr . , cadddr) (cddr . , cddr) (cdr . , cdr)
 (cons . , cons) (eq? (列表, 列表) (地图, 地图)
 (memv . , memv) (null? . , null? ) (对? 。
 (读, 读) (设置车!
 (set-cdr! 。(符号? 。

;new-env 从形式化参数
返回新环境;规范、实际参数列表和外部
;环境。符号? 测试识别“不当”
;参数列表。环境是关联列表,
;将变量与值相关联。
 (define new-env
 (lambda (formals actuals env)
 (cond
[ (null? formals) env]
[ (symbol? formals) (cons (cons (cons formals)
env) ]
[else
 (cons
 (cons (car formals) (car actuals) )
 (new-env (cdr formals) (cdr actuals) env) ) ) )

;查找查找环境中变量 var 的值
;env, using assq.假设 var 绑定在 env 中。
 (define lookup
 (lambda (var env)
 (cdr (assq var env) ) )

;赋值类似于查找,但改变了 ; 的
绑定;变量 var 通过更改关联对
的 cdr (定义赋值
 (lambda (var val env)
 (set-cdr! (assq var env) val) ) )

;exec 计算表达式,识别一小组核心形式。
```

```
 (define exec
 (lambda (expr env)
 (cond
[(symbol? expr) (lookup expr env)]
[(pair? expr)
 (case (car expr)
[(quote) (cadr expr)]
[(lambda)
 (lambda vals
 (let ([env (new-env (cadr expr) vals env)])
 (let loop ([exprs (cddr expr)])
 (if (null? (cdr exprs))
 (exec (car exprs) env)
 (begin
 (exec (car exprs) env)
 (loop (cdr exprs)))))))]
[(if)
 (if (exec (cadr expr) env)
 (exec (caddr expr) env)
 (exec (cadddr expr) env))]
[(set!) (assign (cadr expr) (exec (caddr expr)
env) env)]
[else
 (apply
 (exec (car expr) env)
 (map (lambda (x) (exec x env)) (cdr expr)))])]
[else expr]))

;解释在基元环境中开始执行。
 (define interpret
 (lambda (expr)
 (exec expr primitive-environment)))
```

练习 12.7.1

正如所写的，解释器无法解释自己，因为它不支持其实现中使用的几种语法形式：let（命名和未命名），内部

定义，大小写，cond和start。重写解释器的代码，仅使用它支持的语法形式。

练习 12.7.2

完成上述练习后，使用解释器运行解释器的副本，并使用该副本运行解释器的另一个副本。重复此过程，看看在系统停止之前它将深入多少层。

练习 12.7.3

乍一看，似乎lambda案例可以写得更简单，如下所示。

```
[(lambda)
 (lambda vals
 (let ([env (new-env (cadr expr) vals env)])
 (let loop ([exprs (cddr expr)])
 (let ([val (exec (car exprs) env)])
 (if (null? (cdr exprs))
val
 (loop (cdr exprs)))))))
```

为什么这是不正确的？[提示：计划会侵犯什么属性？

练习 12.7.4

尝试通过寻找提出更少问题或分配更少存储空间的方法，使口译员更有效率。[提示：在计算之前，将词法变量引用转换为(access n)，其中 n 表示相关值前面的环境关联列表中的值数。

练习 12.7.5

Scheme 在应用过程之前计算过程的参数，并将该过程应用于这些参数的值（按值调用）。修改解释器以传递未计算的参数，并安排在引用时评估它们（按名称调用）。[提示：使用 lambda 延迟评估。您将需要创建原始过程（car，null？等）的版本，这些版本不计算其参数。

第 12.8 节.定义抽象对象

此示例演示了便于定义简单抽象对象的语法扩展（请参见第 2.9 节）。这个工具具有无限的潜力，作为Scheme中完整的面向对象子系统的基础。

抽象对象类似于基本数据结构，如对和向量。但是，抽象对象不是通过访问和赋值运算符进行操作，而是响应消息。有效消息和要对每条消息执行的操作由对象本身内的代码定义，而不是由对象外部的代码定义，从而产生更模块化和可能更安全的编程系统。抽象对象的本地数据只能通过对象为响应消息而执行的操作来访问。

特定类型的抽象对象是使用 define-object 定义的，它具有一般形式

```
（定义对象 （名称 var1 ...）
（（var2 expr） ...）
（（味精动作） ...））
```

可以省略第一组绑定 （（var2 expr） ...）。define-object 定义了一个过程，该过程被调用以创建给定类型的新抽象对象。此过程称为 name，此过程的参数将成为局部变量 var1 ... 的值。调用过程后，变量 var2 ... 按顺序绑定到值 expr ...，消息 msg ... 以相互递归的方

式绑定到过程值操作...（与 letrec 一样）。在这些绑定中，将创建新的抽象对象;此对象是创建过程的值。

语法形式 send-message 用于将消息发送到抽象对象。（发送消息对象 msg arg ...）发送对象消息 msg 与参数 arg ....当对象收到消息时，arg ... 将成为与消息关联的操作过程的参数，并且此过程返回的值由 send-message 返回。

以下示例应有助于阐明如何定义和使用抽象对象。第一个示例是一个简单的 kons 对象，它类似于 Scheme 的内置对对象类型，只是要访问或分配其字段需要向它发送消息。

```
(define-object (kons kar kdr)
( (get-car (lambda () kar) )
(get-cdr (lambda () kdr) )
(set-car! (lambda (x) (set! kar x) ) )
(设置 cdr! (lambda (x) (set! kdr x) ) ) ) )

(define p (kons 'a 'b) )
(send-message p get-car) ⇒ a
(send-message p get-cdr) ⇒ b
(send-message p set-cdr! 'c)
(send-message p get-cdr) ⇒ c
```

简单的 kons 对象不执行任何操作，只是根据请求返回或分配其中一个字段。抽象对象的有趣之处在于，它们可用于限制访问或执行其他服务。以下版本的 kons 要求在请求分配其中一个字段时提供密码。此密码是 kons 过程的参数。

```
(define-object (kons kar kdr pwd)
( (get-car (lambda () kar) )
(get-cdr (lambda () kar) )
```

```
(set-car!
(lambda (x p)
(if (string=? pwd)
(set! kar x) ) )
(设置 cdr!
(lambda (x p)
(if (string=? pwd)
(set! kar x) ) ) ) ) )

(定义 p1 (kons 'a 'b "magnificent") )
(send-message p1 set-car! 'c "magnificent")
(send-message p1 get-car) ⇒ c
(send-message p1 set-car! 'd "please")
(send-message p1 get-car) ⇒ c

(define p2 (kons 'x 'y "please") )
(send-message p2 set-car! 'z "please")
(send-message p2 get-car) ⇒ z
```

抽象对象的一个重要功能是它可以保留发送给它的消息的统计信息。以下版本的 kons 对两个字段的访问进行计数。此版本还演示了显式初始化的本地绑定的使用。

```
(define-object (kons kar kdr)
( (count 0) )
( (get-car
(lambda ()
(set! count (+ count 1) )
kar) )
(get-cdr
()
(set! count (+ count 1) )
kdr) )
(accesses
(lambda () count) ) )

(定义 p (kons 'a 'b) )
(send-message p get-car) ⇒ a
```

```
(send-message p get-cdr) ⇒ b
(send-message p accesses) ⇒ 2
(send-message p get-cdr) ⇒ b
(send-message p accesses) ⇒ 3
```

定义对象的实现非常简单。对象定义将转换为对象创建过程的定义。此过程是 lambda 表达式的值，其参数是定义中指定的参数。lambda 的主体由一个用于绑定局部变量的 let* 表达式和一个用于将消息名称绑定到操作过程的 letrec 表达式组成。letrec 的主体是另一个 lambda 表达式，其值表示新对象。此 lambda 表达式的正文使用大小写表达式将传入的消息与预期的消息进行比较，并将相应的操作过程应用于其余参数。

例如，定义

```
(define-object (kons kar kdr)
( (count 0) )
( (get-car
(lambda ()
(set! count (+ count 1) )
kar) )
(get-cdr
()
(set! count (+ count 1) )
kdr) )
(accesses
(lambda () count) ) )
```

转换为

```
(define kons
(lambda (kar kdr)
(let* ([count 0])
(letrec ([get-car
(lambda ()
```

```
 (set! count (+ count 1)) kar)]
[get-cdr
 (lambda ()
 (set! count (+ count 1)) kdr)]
[accesses (lambda () count)])
 (lambda (msg . args)
 (case msg
[(get-car) (apply get-car args)]
[(get-cdr) (apply get-cdr args)]
[(accesses) (apply accesses args)]
[else (assertion-violation 'kons
 "invalid message"
 (cons msg args)))))))

 (library (tspl oop)
 (export define-object send-message)
 (import (rnrs))

; define-object 创建一个对象构造函数，它使用 let* 来绑定
; 本地字段和 letrec 来定义导出的过程。一
个 ;对象本身就是一个接受相应
消息的过程;添加到导出过程的名称中。第二种模式是
;用于允许省略本地字段集。
 (define-syntax define-object
 (syntax-rules )
[(_ (name . varlist)
 ((var1 val1) ...)
 ((var2 val2) ...))
 (定义名称
 (lambda varlist
 (let* ([var1 val1] ...)
 (letrec ([var2 val2] ...)
 (lambda (msg . args)
 (case msg
[(var2) (apply var2 args)]
...
[else
 (assertion-violation 'name
 "invalid message"
```

```
 (cons msg args) ) ) ) ) ) )
[ (_  (name . varlist)  ( (var2 val2) ...) )
 (define-object (name . varlist)
 ()
 ( (var2 val2) ...) ) ) ) )

;发送消息抽象出从行为发送消息的行为
;应用过程并允许取消引用消息。
 (define-syntax send-message
 (syntax-rules )
) [ (_ obj msg arg ...)
 (obj 'msg arg ...) ) ) ) )
```

练习 12.8.1

使用 define-object 定义第 2.9 节中的堆栈对象类型。

练习 12.8.2

使用 define-object 定义队列对象类型，其操作类似于第 2.9 节中所述的操作。

练习 12.8.3

用一个对象来描述另一个对象通常是有用的。例如，第二个 kons 对象类型可以描述为与第一个对象类型相同，但具有密码参数以及与 set-car! 和 set-cdr! 消息关联的不同操作。这称为继承;新类型的对象被称为继承第一个对象的属性。修改 define-object 以支持继承，方法是允许在消息/操作对之后显示可选声明（继承对象名称）。这将需要保存有关每个对象定义的一些信息，以便在后续对象定义中可能使用。应不允许使用冲突的参数

名称，但应使用新对象定义中指定的初始化或操作来解决其他冲突。

练习 12.8.4

基于第 317 页上的方法定义，定义一个完整的对象系统，但使用记录而不是向量来表示对象实例。如果做得好，生成的对象系统应该比上面给出的系统更有效，更易于使用。

第 12.9 节. 快速傅里叶变换

本节中描述的过程使用Scheme的复算术来计算值序列[4]的离散傅里叶变换（DFT）。离散傅里叶变换用于分析和处理各种数字电子应用中的采样信号序列，如模式识别、带宽压缩、雷达目标检测和天气监视。

N 个输入值序列的 DFT，

$\{x(n)\}_{n=0}^{N-1}$，

是 N 个输出值的序列，

$\{X(m)\}_{m=0}^{N-1}$，

每个由等式定义

$$X(m) = \sum_{n=0}^{N-1} x(n) e^{-i\frac{2\pi nm}{N}}.$$

可以方便地抽象出常量（对于给定的N）

$$W_N = e^{-i\frac{2\pi}{N}},$$

为了获得更简洁但等效的等式

$$X(m) = \sum_{n=0}^{N-1} x(n) W_N^{mn}.$$

直接计算 N 个输出值（每个输出值作为 N 个中间值的总和）需要按 N2 操作的顺序进行。快速傅里叶变换（FFT） 适用于 N 为 2 的幂时，只需要 Nlog2N 运算的数量级。虽然通常呈现为相当复杂的迭代算法，但快速傅里叶变换最简洁，最优雅地表示为递归算法。

递归算法，这是由于Sam Daniel [7]，可以通过操作前面的求和来推导，如下所示。我们首先将求和拆分为两个求和，并将它们重新组合成一个从 0 到 N/2 - 1 的求和。

$$\begin{aligned} X(m) &= \sum_{n=0}^{N/2-1} x(n) W_N^{mn} + \sum_{n=N/2}^{N-1} x(n) W_N^{mn} \\ &= \sum_{n=0}^{N/2-1} [x(n) W_N^{mn} + x(n+N/2) W_N^{m(n+N/2)}] \end{aligned}$$

然后，我们拉出共同因素 W_N^{mn}。

$$X(m) = \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2) W_N^{m(N/2)}] W_N^{mn}$$

当 m 为偶数时，我们可以减少到 1，当 m 为奇数时，我们可以减少到 $W_N^{m(N/2)}$ -1，因为

$$W_N^{m(N/2)} = W_2^m = e^{-i\pi m} = \begin{cases} 1, & m \text{ even} \\ -1, & m \text{ odd}. \end{cases}$$

这使我们能够对 m = 2k 和 m = 2k + 1， 0 ⩽ k ⩽ N/2 - 1 的偶数和奇数情况求和进行专业化。

$$
\begin{aligned}
X(2k) &= \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)] W_N^{2kn} \\
&= \sum_{n=0}^{N/2-1} [x(n) + x(n+N/2)] W_{N/2}^{kn} \\
X(2k+1) &= \sum_{n=0}^{N/2-1} [x(n) - x(n+N/2)] W_N^{(2k+1)n} \\
&= \sum_{n=0}^{N/2-1} [x(n) - x(n+N/2)] W_N^{n} W_{N/2}^{kn}
\end{aligned}
$$

得到的求和是 N/2 元素序列的 DFT

$\{x(n) + x(n+N/2)\}_{n=}^{0N/2-1}$

和

$$\{[x(n) - x(n+N/2)] W_N^n\}_{n=0}^{N/2-1}.$$

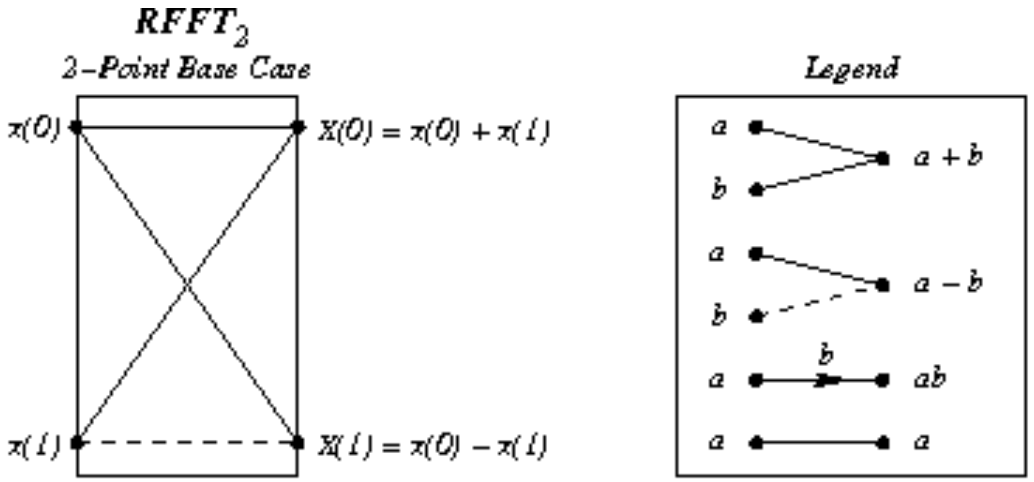
因此，N 元素序列的 DFT 可以通过交错两个 N/2 元素序列的 DFT 来递归计算。如果我们选择两个元素的基本情况，我们可以描述递归快速傅里叶变换（RFFT）算法，如下所示。对于 N = 2，

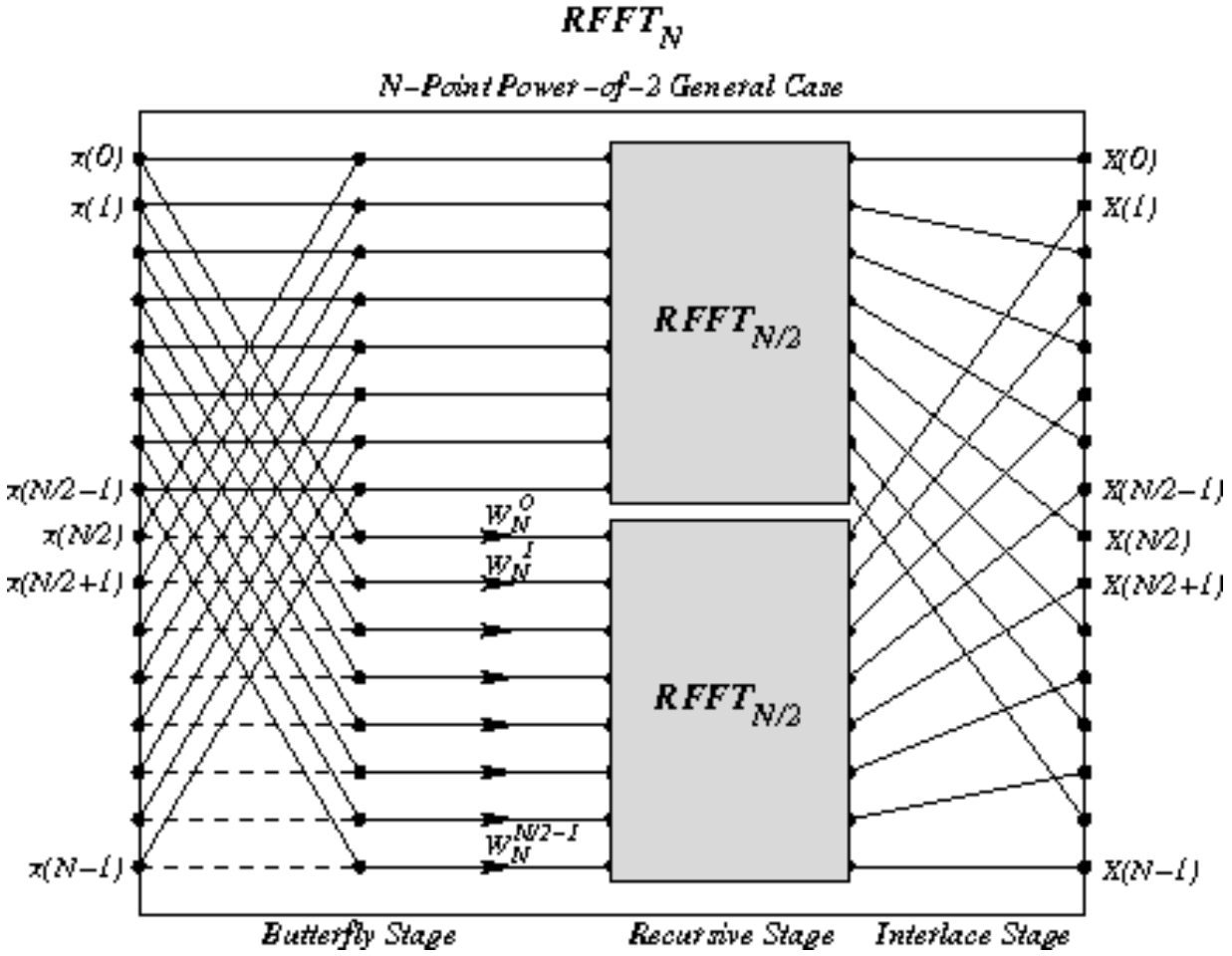
$$
\begin{aligned}
RFFT\{x(n)\}_{n=0}^{1} &= \{X(m)\}_{m=0}^{1} \\
&= \{x(0) + x(1), [x(0) - x(1)] W_2^0\} \\
&= \{x(0) + x(1), x(0) - x(1)\},
\end{aligned}
$$

因为 $W_2^0 = e^0 = 1$. 对于 N > 2，

$$
\begin{aligned}
RFFT\{x(n)\}_{n=0}^{N-1} &= \{X(m)\}_{n=0}^{N-1} \\
&= \begin{cases} RFFT\{x(n) + x(n+N/2)\}_{n=0}^{N/2-1}, & m \text{ even} \\ RFFT\{(x(n) - x(n+N/2)) W_N^n\}_{n=0}^{N/2-1}, & m \text{ odd} \end{cases}
\end{aligned}
$$

伴随的偶数和奇数分量交错。





上图改编自Sam Daniel [7]的一张图，显示了RFFT算法的计算结构。第一级成对计算输入的第一半和第二半的总和和和差值;这个阶段被标记为蝴蝶阶段。第二阶段在生成的子序列上重复出现。第三级将两个递归调用的输出交错到RFFT，从而产生正确排序的序列{X（m）}$_{m=}^{0N-1}$。

过程 dft 接受值的序列（列表），x，其长度假定为 2 的幂。dft 预先计算了 、的W_N幂序列，$\{W_N^n\}_{n=0}^{N/2-1}$并调用 rfft 来启动递归。rfft 遵循上述算法。

```
 (define (dft x)
 (define (w-powers n)
 (let ([pi (* (acos 0.0) 2)])
 (let ([delta (/ (* -2.0i pi) n)])
 (let f ([n n] [x 0.0])
 (if (= n 0)
' ()
 (cons (exp x) (f (- n 2) (+ x delta)))))))
 (定义 (evens w)
 (if (null? w)
' ()
 (cons (car w) (evens (cddr w)))))
 (define (interlace x y)
 (if (null? x)
' ()
 (缺点 (汽车 x) (缺点 (汽车 y) (隔行扫描 (cdr x)
 (cdr y)))))
 (define (split ls)
 (let split ([fast ls] [slow ls])
 (if (null? fast)
 (values ' () slow)
 (let-values ([(front back) (split (cddr fast)
 (cdr slow))])
 (values (cons (cons (car slow) front)
back))))))
 (define (butterfly x w)
 (let-values ([(front back) (split x)])
 (values
 (map + front back)
 (map * (map - front back) w))))
 (define (rfft x w)
 (如果(空? (cddr x)
 (let ([x0 (car x)] [x1 (cadr x)])
 (list (+ x0 x1) (- x0 x1)))
```

```
(let-values ([(front back) (butterfly x w)])
(let ([w (evens w)])
(interlace (rfft front w) (rfft back w))))
(rfft x (length x))))
```

练习 12.9.1

更改算法以采用四个点的基本情况。可以进行哪些简化以避免将任何基本情况输出乘以 w 的元素？

练习 12.9.2

重新编码 dft 以接受向量而不是列表作为输入，并让它生成一个向量作为输出。如有必要，请在内部使用列表，但不要简单地在输入时将输入转换为列表，在退出时将输出转换为向量。

练习 12.9.3

该代码不是为每个步骤重新计算每个步骤中 w 的幂，而是简单地使用前面幂列表中的偶数元素。表明这样做会产生适当的权力列表。也就是说，表明（偶数（w-幂n））等于（w-幂（/ n 2））。

练习 12.9.4

递归步骤会创建几个立即被丢弃的中间列表。对递归步骤进行重新编码，以避免任何不必要的分配。

练习 12.9.5

输入值序列的每个元素都可以通过等式从序列的离散傅里叶变换再生

$$x(n) = \frac{1}{N}\sum_{m=0}^{N-1} X(m)e^{i\frac{2\pi mn}{N}}.$$

注意此方程与定义 X（m） 的原始方程之间的相似性，创建执行逆变换的 dft 的修改版本，即逆 dft。验证（inverse-dft （dft seq）） 是否返回多个输入序列 seq 的 seq。

第 12.10 节.统一算法

统一[23]是一种模式匹配技术，用于自动定理证明，类型推断系统，计算机代数和逻辑编程，例如Prolog [6]。

统一算法尝试通过计算表达式的统一替换来使两个符号表达式相等。替换是用其他表达式替换变量的函数。替换必须以相同的方式处理变量的所有匹配项，例如，如果它用 a 替换变量 x 的一个匹配项，则必须用 a 替换所有出现的 x。两个表达式 e_1 和 e_2 的统一替换或统一器是替换，σ使得 $\sigma(e_1) = \sigma(e_2)$。

例如，两个表达式 f（x） 和 f（y） 可以通过用 x 代替 y（或 y 代替 x）来统一。在这种情况下，统一器σ可以被描述为用x替换y并保持其他变量不变的函数。另一方面，两个表达式 x + 1 和 y + 2 不能统一。似乎用 3 代替 x，用 2 代替 y 会使两个表达式等于 4，从而彼此相等。但是，符号表达式 3 + 1 和 2 + 2 仍然不同。

两个表达式可以有多个统一体。例如，表达式 f（x，y） 和 f（1，y） 可以统一为 f（1，y），用 1 代替 x。它们也可以统一为 f（1，5），用 1 代替 x，用 5 代替 y。第一个替换是可取的，因为它不会承诺不必要的 y 替换。统一算法通常为两个表达式生成最通用的统一器或 mgu。两个表达式的 mgu 不会进行不必要的替换;表达式的所有其他统一器都是 mgu 的特例。在上面的示例中，第一个替换是 mgu，第二个替换是一个特例。

出于此程序的目的，符号表达式可以是变量、常量或函数应用程序。变量由方案符号表示，例如x;函数应用程序由一个列表表示，其中函数名称位于第一个位置，其参数位于其余位置，例如（f x）;和常量由零参数函数表示，例如（a）。

此处介绍的算法使用延续传递样式或 CPS（参见第 3.4 节）在子项上递归的方法查找两个项的 mgu（如果存在）。该过程 unify 采用两个术语，并将它们传递给帮助过程 uni，以及初始（标识）替换、成功延续和失败延续。成功延续返回将其参数（替换）应用于其中一个项的结果，即统一结果。失败延续仅返回其参数，即消息。由于控件在 unify 中通过显式延续传递（始终使用尾部调用），因此成功或失败延续的返回是 unify 本身的返回。

替换是程序。每当一个变量要被另一个项替换时，就会从该变量、项和现有替换中形成一个新的替换项。给定一个术语作为参数，新的替换将在其保存的变量中用其保存的术语替换其保存的项，结果是在参数表达式上调用保存的替换。直观地说，替换是一个过程链，替换中的每个变量一个。链由初始标识替换终止。

```
 (统一 'x 'y) ⇒ y
 (unify '(f x y) '(g x y)) ⇒ "clash"
 (unify '(f x (h)) '(f (h) y)) ⇒ (f (h)
 (h))
 (unify '(f (g x) y) '(f y x)) ⇒ "cycle"
 (unify '(f (g x) y) '(f y (g x))) ⇒ (f (g
x) (g x))
 (unify '(f (g x) y) (f (g x) (f (g x) (g
x))) ⇒ (f (g x)) (f (g x) (g x))

 (library (tspl unification)
 (export unify)
 (import (rnrs))

;happens? 当且仅当出现在 v
中时才返回 true (定义发生?
 (lambda (u v)
 (and (pair? v)
 (let f ([l (cdr v)])
 (and (pair? l)
 (or (eq? u (car l))
 (happens? u (car l))
 (f (cdr l)))))))

;sigma 返回一个新的替换过程，将 s 扩展为
;用 v
替换 u (定义 sigma
 (lambda (u v s)
 (lambda (x)
 (let f ([x (s x)])
 (if (symbol? x)
 (if (eq? x u) v x)
 (cons (car x) (map f (cdr x))))))

;try-subst 尝试用 u 代替 v，但可能需要 ;
完全统一，如果 (s u) 不是变量，并且它可以
;如果它看到 u 出现在 v 中
，则失败 (定义 try-subst
 (lambda (u v s ks kf)
```

```
 (let ([u (s u)])
 (if (not (symbol? u))
 (uni u v s ks kf)
 (let ([v (s v)])
 (cond
[(eq? u v) (ks s)]
[(happens u v) (kf "cycle")]
[else (ks (sigma u v))))))

;uni 试图通过延续传递
来统一 u 和 v ;返回成功参数
的替换的样式;ks 或失败参数 kf 的错误消息。;
替换本身由 来自
;变量到术语。
 (define uni
 (lambda (u v s ks kf)

 (cond
[(symbol? u) (try-subst u v s ks kf)]
[(symbol? v) (try-subst v u s ks kf)]
[(and (eq? (车 u) (汽车 v))
 (= (length u) (length v)))
 (let f ([u (cdr u)] [v (cdr v)] [s])
 (if (null? u)
 (ks s)
 (uni (car u)
 (car v)
s
 (lambda (s) (f (cdr u) (cdr v) s))
kf)))))
[else (kf "clash")))))

;unify 显示了 uni 的一个可能的接口，其中初始
;替换是身份识别程序，初始成功
;延续返回统一项，以及初始失败
;继续返回错误消息。
 (define unify
 (lambda (u v)
 (uni u
```

```
v
 (lambda (x) x)
 (lambda (s) (s u))
 (lambda (msg) msg)))))
```

练习 12.10.1

修改统一，以便它返回其替换项而不是统一项。将此替换应用于两个输入项，以验证它是否为每个项返回相同的结果。

练习 12.10.2

如上所述，对项的替换是按顺序执行的，需要对每个替换的变量进行一次完整的输入表达式传递。以不同的方式表示替换，以便只需通过表达式进行一次传递。确保不仅对输入表达式执行替换，而且对在替换期间插入的任何表达式执行替换。

练习 12.10.3

将延续传递样式统一算法扩展为整个连续传递样式逻辑编程系统。

第 12.11 节. 使用引擎进行多任务处理

引擎是支持定时抢占的高级流程抽象 [10，15]。引擎可用于模拟多处理、实现轻量级线程、实现操作系统内核以及执行非确定性计算。引擎实现是Scheme中延续的更有趣的应用之一。

通过将 thunk（无参数过程）传递给过程生成引擎来创建引擎。砰的主体是由发动机执行的计算。引擎本身是一个由三个参数组成的过程：

1. ticks，一个正整数，指定要提供给发动机的燃油量。发动机一直执行到燃料耗尽或直到其计算完成。

2. complete，由两个参数组成的过程，指定在计算完成时要执行的操作。它的参数将是剩余的燃料量和计算结果。

3. expire，一个参数的过程，指定在计算完成之前燃料耗尽时该怎么办。它的论点将是一个能够从中断点继续计算的新引擎。

当引擎应用于其参数时，它会设置一个计时器，以以刻度时间单位触发。如果引擎计算在计时器关闭之前完成，则系统将调用 complete，向其传递剩余的刻度数和计算的值。另一方面，如果计时器在引擎计算完成之前关闭，则系统将从中断计算的继续创建一个新引擎，并通过此引擎使其过期。完成和过期是在引擎调用的继续中调用的。

下面的示例从一个简单的计算 3 创建一个引擎，并为该引擎提供 10 个刻度。

```
(定义 eng
(make-engine
(lambda () 3)))

(eng 10
(lambda (ticks value) value)
(lambda (x) x)) ⇒ 3
```

将列表作为完整过程传递给引擎通常很有用，如果计算完成，则引擎返回剩余刻度和值的列表。

```
(eng 10
list
(lambda (x) x)) ⇒ (9 3)
```

在上面的示例中，值为 3，还剩下 9 个刻度，即只用一个燃料单位来评估 3。（此处给出的燃料量仅供说明之用。实际金额可能有所不同。

通常，引擎计算不会在一次尝试中完成。下面的示例按步骤显示如何使用引擎来计算第 10 个斐波那契数列（参见第 3.2 节）。

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

(define eng
(make-engine
(lambda ()
(fibonacci 10)))))

(eng 50
list
(lambda (new-eng)
(set! eng new-eng)
“expireed”)) ⇒ “expireed”

(eng 50
list
(lambda (new-eng)
(set! eng new-eng)
```

```
 “expireed”） ⇒ “expireed”

 （eng 50
list
 （lambda （new-eng）
 （set！ eng new-eng）
 “expireed”））） ⇒ “expireed”

 （eng 50
list
 （lambda （new-eng）
 （set！ eng new-eng）
 “expireed”））） ⇒ （22 55）
```

每当发动机的燃料耗尽时，过期程序就会分配给新发动机。整个计算需要四个 50 个价格变动的分配才能完成;在最后的50个中，它使用了除23个之外的所有。因此，燃料使用总量为177个刻度。这就引出了以下过程，里程，它使用引擎来“计时”计算。

```
 （定义里程
 （lambda （thunk）
 （let loop （[eng （make-engine thunk）] [total-ticks
0]）
 （eng 50
 （lambda （ticks value）
 （+ total-ticks （- 50 ticks））））
 （lambda （new-eng）
 （loop new-eng （+ total-ticks 50））））））

 （里程 （lambda （） （斐波那契 10））） ⇒ 178
```

当然，每次使用的价格变动数量选择 50 是任意的。传递一个更大的数字（例如 10000）可能会更有意义，以减少计算中断的次数。

下一个过程，即循环，可以成为简单分时操作系统的基础。轮循机制维护进程队列（引擎列表），并以轮循机制方式循环浏览该队列，从而允许每个进程运行一段设定的时间。轮循机制按计算完成的顺序返回引擎计算返回的值的列表。

```
(定义 round-robin
(lambda (engs)
(if (null? engs)
'()
( (car engs) 1
(lambda (ticks value)
(cons value (round-robin (cdr engs) ) ) )
(lambda (eng)
(round-robin
(append (cdr engs) (list eng) ) ) ) ) ) ) )
```

假设对应于一个价格变动的计算量是恒定的，则轮循机制的作用是返回按从最快完成到最慢完成排序的值列表。因此，当我们在引擎列表上调用循环时，每个引擎计算一个斐波那契数列，输出列表首先使用较早的斐波那契数列进行排序，而不管输入列表的顺序如何。

```
(round-robin
(map (lambda (x)
(make-engine
(lambda ()
(fibonacci x) ) ) )
'(4 5 2 8 3 7 6 2) ) ) ⇒ (1 1 2 3 5 8 13 21)
```

如果每次通过循环的燃料量都发生变化，则可能会发生更有趣的事情。在这种情况下，计算将是非确定性的，即结果因调用而异。

下面的语法形式 por （parallel-or） 返回其第一个表达式，以完成一个 true 值。por 是使用第一个 true 过程实现的，该过程类似于循环，但当任何引擎以 true 值完成时，该过程将退出。如果所有引擎都已完成，但没有一个具有真实值，则 first-true（因此是 por）将返回 #f。

```
(define-syntax por
(syntax-rules )
[(_ x ...)
(第一真
(列表 (制造引擎 (lambda () x)) ...)))))

(定义第一真
(lambda (engs)
(if (null? engs)
#f
((car engs) 1
(lambda (ticks value)
(or value (first-true (cdr engs)))))
(lambda (eng)
(first-true
(append (cdr engs) (list eng))
```

即使其中一个表达式是无限循环，por 仍然可以完成（只要其他表达式之一完成并返回 true 值）。

```
(孔 1 2) ⇒ 1
(por ((lambda (x) (x x)) (lambda (x) (x x)))
(fibonacci 10)) ⇒ 55
```

第二个 por 表达式的第一个子表达式是非终止的，所以答案是第二个子表达式的值。

让我们转向引擎的实现。任何抢占式多任务处理基元都必须能够在给定的计算量后中断正在运行的进程。在某些 Scheme 实现中，此功能由原始计时器中断机制提供。我们将在这里构建一个合适的。

我们的计时器系统定义了三个过程：启动计时器，停止计时器和递减计时器，可以在操作上描述如下。

- （启动计时器滴答器处理程序）将计时器设置为滴答，并在计时器过期（即达到零）时将处理程序安装为要调用的过程（不带参数）。

- （停止计时器）重置计时器并返回剩余的逐笔报价数。

- （递减计时器）如果计时器处于打开状态，则计时器递减一个刻度，即如果它不是零。当计时器达到零时，递减计时器将调用保存的处理程序。如果计时器已达到零，则递减计时器返回而不更改计时器。

下面给出了实现这些过程的代码以及引擎实现。

使用计时器系统需要在适当的位置插入对递减计时器的调用。在进入过程时使用计时器刻度通常可提供足够的粒度级别。这可以通过使用下面定义的定时 lambda 代替 lambda 来实现。timed-lambda 只是在执行其主体中的表达式之前调用递减计时器。

```
(define-syntax timed-lambda
(syntax-rules ()
[(_ formals exp1 exp2 ...)
(lambda formals (decrement-timer) exp1 exp2
...)]))
```

重新定义命名的 let 和 do 以使用计时 lambda 可能很有用，以便用这些构造表示的递归是定时的。如果您使用此机制，请不要忘记在引擎内运行的代码中使用 lambda 和其他表单的定时版本，否则不会消耗任何刻度。

现在我们有了合适的计时器，我们可以在计时器和延续方面实现引擎。我们在引擎实现中的两个位置使用 call/cc：（1） 获取调用引擎的计算的延续，以便在引擎计算完成或计时器过期时可以返回到该延续;（2） 在计时器过期时获取引擎计算的延续，以便在随后运行新创建的引擎时返回到此计算。

发动机系统的状态包含在发动机系统本地的两个变量中：do-complete 和 do-expire。当引擎启动时，引擎会分配 do-complete 和 do-expire 过程，这些过程在调用时会返回到引擎调用方的延续以调用 complete 或 expire。引擎通过调用作为参数传递给具有指定刻度数的 make 引擎的过程来启动（或重新启动）计算。然后，使用刻度和本地过程计时器处理程序来启动计时器。

假设计时器在引擎计算完成之前过期。然后调用过程计时器处理程序。它启动对启动计时器的调用，但通过调用具有 do-expire 的 call/cc 来获取刻度。因此，do-expire 将使用一个延续来调用，如果调用该延续，将重新启动计时器并继续中断的计算。do-expire 从此延续创建一个新引擎，并安排使用正确延续中的新引擎调用引擎的过期过程。

另一方面，如果引擎计算在计时器到期之前完成，则计时器将停止，并且剩余的价格变动数将与要完成的值一起传递;do-complete 安排使用正确延续中的刻度和值调用引擎的完整过程。

让我们讨论一下此代码的几个微妙方面。第一个问题涉及在调用引擎时用于启动计时器的方法。通过让新引擎在启动或恢复引擎计算之前启动计时器，而不是将刻度传递给计算并让它启动计时器，显然可以简化代码。但是，在计算中启动计时器可防止过早消耗刻度。如果发动机系统本身消耗燃料，则提供少量燃料的发动机可能无法完成。（事实上，它可能会取得消极进展。如果使用上述软件计时器，则通过使用非定时版本的 lambda 编译引擎生成代码，实际上可以避免此问题。

第二个微妙之处涉及由 do-complete 和 do-expire 创建的过程，以及随后通过继续调用/抄送应用程序应用的过程。看起来 do-complete 可以首先调用引擎的完整过程，然后将结果传递给继续（对于 do-expire 也是如此），如下所示。

（转义（完整值刻度））

但是，这将导致对尾递归的不当处理。问题是，在返回对完整过程的调用之前，当前继续不会替换为存储在转义中的继续。因此，正在运行的引擎的继续和引擎调用的继续都可以无限期地保留，而实际上实际的引擎调用可能看起来像是尾递归的。这尤其不合适，因为引擎接口鼓励使用延续传递样式，因此鼓励使用尾部递归。轮循机制调度程序和 first-true 提供了很好的示例，因为每个中的 expire 过程都以递归方式调用引擎尾部。

我们通过在调用完整或过期过程之前安排 do-complete 和 do-expire 来避免运行引擎的继续，从而保持对尾部递归的正确处理。由于引擎调用的延续是过程应用程序，因此传递给它一个没有参数的过程会导致在引擎调用的继续中应用该过程。

```
(库 (tspl timer)
(导出启动计时器停止计时器递减计时器)
(导入 (rnrs))

(定义时钟 0)
(定义处理程序 #f)

(定义启动计时器
(lambda (ticks new-handler)
(set! handler new-handler)
(set! clock ticks))))

(定义 stop-timer
(lambda ()
(let ([time-left clock])
(set! clock 0)
)))

(定义递减计时器
(lambda ()
(当 (> 时钟 0)
(设置! 时钟 (- 时钟 1))
(当 (= 时钟 0) (处理程序)))))

(定义语法 timed-lambda
(语法规则 ()
[(_ formals exp1 exp2 ...)
(lambda formals (decrement-timer) exp1 exp2
...)]))))

(库 (tspl 引擎)
(导出生成引擎 timed-lambda)
(import (rnrs) (tspl timer)))

(定义 make-engine
(let ([do-complete #f] [do-expire #f])
(定义 timer-handler
(lambda ()
(start-timer (call/cc do-expire) timer-
```

```
handler) ) ) )
 (定义 new-engine
 (lambda (resume)
 (lambda (ticks complete expire)
 ( (call/cc
 (lambda (escape)
 (set! do-complete
 (lambda (ticks value)
 (escape (lambda () (complete ticks value) ) ) ) )
 (set! do-expire
 (lambda (resume)
 (escape (lambda ()
 (expire (renew-engine resume) )
 (resume ticks) ) ) ) ) ) )
 (lambda (proc)
 (new-engine
 (lambda (ticks)
 (start-timer ticks timer-handler)
 (let ([value (proc) ])
 (let ([ticks (stop-timer) ])
 (do-complete ticks value) ) ) ) ) ) )

 (define-syntax timed-lambda
 (syntax-rules ()
[ (_ formals exp1 exp2 ...)
 (lambda formals (decrement-timer) exp1 exp2
...) ]) ) )
```

练习 12.11.1

如果您的 Scheme 实现允许在交互式顶层定义和导入库，请尝试定义上面的库，然后键入

```
 (import (rename (tspl engines) (timed-lambda
lambda) )
```

以定义生成引擎并重新定义 lambda。然后尝试本节前面给出的示例。

练习 12.11.2

看起来嵌套的 let 表达式在 make-engine 的主体中：

```
(let ([value (proc)])
(let ([ticks (stop-timer)])
(do-complete ticks value)))
```

可以替换为以下内容。

```
(let ([value (proc)] [ticks (stop-timer)])
(do-complete value ticks))
```

为什么这是不正确的？

练习 12.11.3

将前面练习中讨论的嵌套 let 表达式替换为以下内容也是不正确的。

```
(let ([value (proc)])
(do-complete value (stop-timer)))
```

为什么？

练习 12.11.4

修改引擎实现以提供立即从引擎返回的过程（引擎返回）。

练习 12.11.5

使用进程引擎实现小型操作系统的内核。进程应通过计算表单的表达式（陷阱'请求）来请求服务（例如读取用户的输入）。使用前面练习中的 call/cc 和引擎返回来实现陷阱。

练习 12.11.6

在不使用引擎的情况下编写相同的操作系统内核，而是从延续和计时器中断进行构建。

练习 12.11.7

这种引擎实现不允许一个引擎调用另一个引擎，即嵌套引擎[10]。修改实现以允许嵌套引擎。

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