SICP课后习题解答

BY SISYS

1 第一章

1.1 略

1.2 将下面表达式变换为前缀

```
(/ (+ 5
      4
      (- 2
         (- 3
            (+ 6
               (/ 4 5)))))
   (* 3 (- 6 2) (- 2 7)))
```

1.3 定义个过程三个数为参数返回较大两个的和

```
(define (max3 a b c)
  (if (> a b)
      (if (> b c)
          (+ a b)
          (+ a c))
      (if (> a c)
          (+ a b)
          (+ b c))))
```

1.4 略

1.5 验证正则序还是应用序

```
(define (p) (p))
(define (test x y)
  (if (= x 0)
      0
      y))
```

如果是正则序结果是过程会返回0, 而应用序则会陷入死循环。

1.6 解答

```
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
```

同样是正则序和应用序的问题， 如果if是普通过程那么在前面的牛顿开平方方法中， 在应用序中会把参数都求值， 而else-clause的语句是(sqrt-iter xxx)所以会无限的递归没法停下来。这也是我疑惑的地方我通过1.5题在本机的guile/scheme中验证发现能正常返回， 说明是正则序这就跟书中说scheme要求是应用序矛盾， 而此处代码陷入无限循环又符合了应用序列。所以我只能猜测是因为guile/scheme实现针对1.5练习题这种情况做了优化。

1.7 解答

```
;; 新的判断方法 下一个猜测值相对现在的猜测值的变化率
;; 低于0.001则停止
(define (new-good-enough? guess x)
  (<
   (abs
    (- (improve guess x) x))
   0.001))
```

1.8 略过

1.9 解答

```
(define (+ a b)
  (define (my-iter a sum )
    (if (= a 0)
        sum
        (my-iter (- a 1) (inc sum))))
  (my-iter a b))

(define (+ a b)
  ;; (define (inc a)
  ;;   (+ a 1))
  ;; (define (dec a)
  ;;   (- a 1))
  (define (my-iter a b)
    (if (= a 0)
        b
        (my-iter (dec a) (inc b))))
  (my-iter a b))
```

对于第一种情况代换模型， 假设a=5,b=4

```
(+ 5 4)
(inc (+ 4 4))
(inc (inc (+ 3 4)))
(inc (inc (inc (+ 2 4))))
(inc (inc (inc (inc (+ 1 4)))))
(inc (inc (inc (inc (inc (+ 0 4))))))
(inc (inc (inc (inc (inc 4)))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
9
```

第二种代换模型同样的假设

```
(+ 5 4)
(+ 4 5)
(+ 3 6)
(+ 2 7)
(+ 1 8)
```

```
(+ 0 9)
```

1.10 解答

下面是一个叫做Ackermann的数学函数。

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

它的数学表达式应该是这样的:

$$f(x,y) = \begin{cases} 0 & ,y=0 \\ 2y & ,x=0 \\ 2 & ,y=1 \\ f(x-1, f(x,y-1)) & \end{cases}$$

先看下`(A 1 10)`的值

```
(A 1 10)
(A 0 (A 1 9))
(A 0 (A 0 (A 1 8)))
(A 0 (A 0 (A 0 (A 1 7))))
(A 0 (A 0 (A 0 (A 0 (A 1 6)))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 1 5))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 1 4)))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 1 3))))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 1 2)))))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 1 1))))))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 2)))))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 4))))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 (A 0 8)))))))
(A 0 (A 0 (A 0 (A 0 (A 0 (A 0 16))))))
(A 0 (A 0 (A 0 (A 0 (A 0 32)))))
(A 0 (A 0 (A 0 (A 0 64))))
(A 0 (A 0 (A 0 128)))
(A 0 (A 0 256))
(A 0 512)
1024
```

所以(A 1 n)的结果是2^n。接着看下`(A 2 4)`

```
(A 2 4)
(A 1 (A 2 3))
(A 1 (A 1 (A 2 2)))
(A 1 (A 1 (A 1 (A 2 1))))
(A 1 (A 1 (A 1 2)))
(A 1 (A 1 (A 0 (A 1 1))))
(A 1 (A 1 (A 0 2)))
(A 1 (A 1 4))
(A 1 (A 0 (A 1 3)))
(A 1 (A 0 (A 0 (A 1 2))))
(A 1 (A 0 (A 0 (A 0 (A 1 1)))))
```

```
(A 1 (A 0 (A 0 (A 0 2))))
(A 1 (A 0 (A 0 4)))
(A 1 (A 0 8))
(A 1 16)
```

根据上面的展开最终会变成(A 1 16)所以是2^{16}。现在来看下(A 3 3)的展开

```
(A 3 3)
(A 2 (A 2 2 ))
(A 2 (A 1 (A 2 1)))
(A 2 (A 1 2))
(A 2 (A 0 (A 1 1)))
(A 2 (A 0 2))
(A 2 4)
```

可以看到展开变成了(A 2 4) 所以结果是2^{16}

现在来分析下 `(define (f n) (A 0 n))`的数学定义是$2n$。

而`(define (g n) (A 1 n))`。结合上面的数学公式其实更容易看出来$f(1,y)=f(0,f(1,y-1))$而$f(0,y)=2y$同时$f(x,1)=2$所以结果就是y个2想乘。所以$f(1,n)=2^n$。

接着分析下$f(2,n)$的数学定义。

$$f(2,y)=f(1,f(2,y-1))=f(1,f(1,f(2,y-2)))=\cdots=f(1,f(1,\ldots.f(1,f(2,1))))$$

嵌套了y-1个$f(1,)$最内层的$f(1,\ f(2,1))=f(1,2)=2^2=4$。 其外剩下n-2层f$(1,\ldots)$也就是说从内到外反复应用$f(1,n)$,根据上面的$f(1,n)=2^n$。 所以$f(2,n)=2^{2^{2^{\cdots^{2^1}}}}$总共n-1次。 所以$f(2,3)=2^{2^2}=16$, $f(2,4)=2^{2^{2^2}}=2^{16}$

1.11 解答

写出如下函数的递归计算过程和迭代计算过程

$$f(n)=\begin{cases} n & ,n<3 \\ f(n-1)+2f(n-2)+3f(n-3) & ,n\geqslant 3 \end{cases}$$

代码如下

```
(define (f n)
  (if (< n 3)
      n
      (+ (f (- n 1))
         (* 2 (f (- n 2)))
         (* 3 (f (- n 3))))))

(define (f n)
  ;; a b c => f(n-1) f(n-2) f(n-3)
  (define (f-iter a b c count)
    (cond ((< n 3) n)
          ((= n count) a)
          (else (f-iter
                 (+ a (* 2 b) (* 3 c))
                 a
                 b
```

```
                    (+ count 1)))))
  (f-iter 2 1 0 2))
```

1.12

1.13 略

1.14 画图 略

1.15 解答

角足够小时， 正弦值可以近似$\sin x \cong x$计算。而还有个三角恒等式

$$\sin x = 3\sin\frac{x}{3} - 4\sin^3\frac{x}{3}$$

可以减小sin的参数。同样假设x足够小不大于0.1弧度

a) (sin 12.15)时p被使用了多少次? 应该是$\log_{\frac{12.15}{3}}0.1$

b) 空间是O($\mathrm{Log}_{\frac{a}{3}}0.1$)

1.16 解答

给出b^n的迭代版本代码，

```
(define (fast-expt b n)
  (fast-expt-iter b n 1))
(define (fast-expt-iter b n ans)
  (cond ((= n 0) ans)
        ((even? n) (fast-expt-iter (* b b) (/ n 2) ans))
        (else (fast-expt-iter b (- n 1) (* ans b)))))
```

1.17 解答

假设语言没有乘法， 只有加减法请使用加减法函数实现类似习题1.16的迭代式乘法实现， 同时假设已经存在double和halve函数， 前者将一个数字翻倍后者将数字减半。代码如下

```
(define (double a)
  (+ a a))
(define (halve a)
  (/ a 2))
(define (* a b)
  (mul-iter a b 0))
(define (mul-iter a b ans)
  (cond ((= b 0) ans)
        ((even? b) (mul-iter (double a) (halve b) ans))
        (else (mul-iter a (- b 1) (+ a ans)))))
```

1.18 解答

说实话我看不懂题目， 说参考1.16和1.17实现一个只用加法和double,halve的迭代过程实现a*b。可是1.17不就是符合要求么? 估计是1.17要求递归过程1.18要求改写为迭代过程。我一步到位了 所以不需要做了

1.19 解答

$$
\begin{aligned}
a &\leftarrow \mathrm{bq}+\mathrm{aq}+\mathrm{ap}\\
b &\leftarrow \mathrm{bp}+\mathrm{aq}
\end{aligned}
$$

现在再做一次变换

$$
\begin{aligned}
a &\leftarrow (\mathrm{bp}+\mathrm{aq})\,q+(\mathrm{bq}+\mathrm{aq}+\mathrm{apb})\,q+(\mathrm{bq}+\mathrm{aq}+\mathrm{ap})\,p\\
b &\leftarrow (\mathrm{bp}+\mathrm{aq})\,p+(\mathrm{bq}+\mathrm{aq}+\mathrm{ap})\,q
\end{aligned}
$$

现在将第二次变换的结果整理成第一次的形式（均基于变换之前， 也就是a和b)

$$
\begin{aligned}
a &\leftarrow b(q^2+2\mathrm{pq})+a(2\mathrm{pq}+q^2)+a(p^2+q^2)\\
b &\leftarrow b(p^2+q^2)+a(2\,\mathrm{pq}+q^2)
\end{aligned}
$$

所以证明了两次变换可以通过计算变成一次变换。所以菲波那契算法可以改写成

```
(define (fib n)
  (fib-iter 1 0 0 1 n))
(define (fib-iter a b p q count)
  (cond ((= count 0) b)
        ((even? count)
         (fib-iter a b
                   (+ (* q q) (* p p))
                   (+ (* q q) (* 2 (* p q)))
                   (/ count 2)))
        (else (fib-iter (+ (* b q) (* a q) (* a p))
                        (+ (* b p) (* a q))
                        p
                        q
                        (- count 1)))))
```

1.20 解答

用正则序计算gcd(206,40)执行了多少次(remainder)

```
;; 正则序解释gcd(206 40)
;; 为了缩短代码长度 将remainder替换为%
(gcd 206 40)
(if (= 40 0) 206 (gcd 40 (% 206 40)))
(gcd 40 (% 206 40))
(if (= (% 206 40))                      ;这里+1
    40
    (gcd (% 206 40) (% 40 (% 206 40))))
(gcd (% 206 40) (% 40 (% 206 40)))
(if  (= (% 40 (% 206 40)) 0)            ;+2
     (% 206 40)
     (gcd  (% 40 (% 206 40))
           (%  (% 206 40) (% 40 (% 206 40)))))
(gcd  (% 40 (% 206 40))
      (%  (% 206 40) (% 40 (% 206 40))))
(if (= (%  (% 206 40) (% 40 (% 206 40))) 0) ;+4
    (% 40 (% 206 40))
    (gcd (%  (% 206 40) (% 40 (% 206 40)))
         (%  (% 40 (% 206 40))
             (%  (% 206 40) (% 40 (% 206 40))))))
(gcd (%  (% 206 40) (% 40 (% 206 40)))
```

```
      (%  (% 40 (% 206 40))
          (%  (% 206 40) (% 40 (% 206 40)))))
(if (= (%  (% 40 (% 206 40))
            (%  (% 206 40) (% 40 (% 206 40))))
       0)                                ;+7
    (%  (% 206 40) (% 40 (% 206 40)))
    (gcd (%  (% 40 (% 206 40))
              (%  (% 206 40) (% 40 (% 206 40))))
         (% (%  (% 206 40) (% 40 (% 206 40)))
            (%  (% 40 (% 206 40))
                (%  (% 206 40) (% 40 (% 206 40)))))))
(%  (% 206 40) (% 40 (% 206 40)))        ;+4
```

所以remainder执行的次数是1+2+4+7+4=18次

1.21 解答 略

补充知识点

```
;; expmod 求得是base^exp % m
;; 对数步骤
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)                    ;exp是偶数 可以变成 下面的代码
         ;; a^m=bk+c
         ;; (a^m)^2 = b^2 k^2+2bck+c^2=b(bk^2+2ck)+c^2
         ;; 所以 base^exp%m的余数是base^(exp/2)%m的平方倍数
         (remainder (square (expmod base (/ exp 2) m))
                    m))
        ;; a^m=bk+c
        ;; a^(m+1)=abk+ac 所以余数自然就是exp-1的base倍数
        (else (remainder (* base (expmod base (- exp 1) m)) ;同理 不再赘述
                         m))))
;; 费马检测
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1))))) ;; 随机选出一个a 此处需要+1 使得范围是[1,n]
(define (fast-prime? n times); 做times次检查 每次都是true才能说很有可能是素数
  (cond ((= times 0) true)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else false)))
```

1.22 解答

```
(define (smallest-divisor n)
  (find-divisor n 2))
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (+ test-divisor 1)))))
(define (divides? a b)
  (= (remainder b a) 0))
(define (prime? n) ; 判断是否是素书的sqrt(n) 方法
  (= n (smallest-divisor n)))

(define (runtime)
```

```
  (let ((x (gettimeofday)))                    ; 函数返回的是(second . millsecond)
    (+ (* (car x) 100000) (cdr x))))           ; 微秒
(define (next-odd n)
  (if (even? n)
      (+ n 1)
      (+ n 2)))
; 求start-num到end-num之间的最小素数
(define (search-for-primes start-num end-num)
  (search-for-primes-iter start-num end-num (runtime)))
(define (search-for-primes-iter start-num end-num start-time)
  (cond ((> start-num end-num)
         (display "not found"))
        ((prime? start-num)
         (begin
           (newline)
           (display start-num)
           (display " is prime num, cost ")
           (display (- (runtime) start-time))
           (display " millsecond")))
        (else (search-for-primes-iter (nex-odd start-num) end-num
start-time))))
```

求出1000附近的素数， 10000附近的素数按理来说耗时倍率应该是$\sqrt{10}$的关系。但是通过在本地上跑测试发现并未体现出$\sqrt{10}$的关系。

1.23 解答

相比前一题， 本题的问题是如果修改函数， 跳过偶数的检查， 问性能是否有提升为前面的 2倍。

这个问题在电脑上波动太大。不过并不存在时间比值是2的情况。电脑性能比这本书出版时的性能好很多， 所以耗时很短导致， 一次结果小几十微妙(10000和100000附近的数字)， 并且使用next-odd跳过整数之后耗时反而变长了， 应该是(next-odd)本身的耗时造成的。。。

```
(define (smallest-divisor n)
  (find-divisor n 2))
(define (find-divisor n test-divisor)
  (cond ((> (square test-divisor) n) n)
        ((divides? test-divisor n) test-divisor)
        (else (find-divisor n (next-odd test-divisor)))))
(define (divides? a b)
  (= (remainder b a) 0))
(define (prime? n)
  (= n (smallest-divisor n)))
(define (next-odd n)
  (if (even? n)
      (+ n 1)
      (+ n 2)))

(define (search-for-primes start-num end-num)
  (search-for-primes-iter start-num end-num (runtime)))
(define (search-for-primes-iter start-num end-num start-time)
  (cond ((> start-num end-num)
         (display "not found"))
        ((prime? start-num)
         (begin
           (newline)
           (display start-num)
           (display " is prime num, cost ")
```

```
            (display (- (runtime) start-time))
            (display " millsecond")))
         (else (search-for-primes-iter (next-odd start-num) end-num
start-time))))
```

1.24 解答

将(prime?)替换为费马检测对比耗时， 相比前面的方法耗时大幅度降低

```
;; expmod 求得是base^exp % m
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)                    ;exp是偶数 可以变成 下面的代码
         ;; a^m=bk+c
         ;; (a^m)^2 = b^2 k^2+2bck+c^2=b(bk^2+2ck)+c^2
         ;; 所以 base^exp%m的余数是base^(exp/2)%m的平方倍数
         (remainder (square (expmod base (/ exp 2) m))
                    m))
        ;; a^m=bk+c
        ;; a^(m+1)=abk+ac 所以余数自然就是exp-1的base倍数
        (else (remainder (* base (expmod base (- exp 1) m)) ;同理 不再赘述
                         m))))
;; 费马检测素数
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
(define (fast-prime? n times)
  (cond ((= times 0) #t)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else #f)))
(define (square x)
  (* x x))
(define (next-odd n)
  (if (even? n)
      (+ n 1)
      (+ n 2)))

(define (search-for-primes start-num end-num)
  (search-for-primes-iter start-num end-num (runtime)))
(define (search-for-primes-iter start-num end-num start-time)
  (cond ((> start-num end-num)
         (display "not found"))
        ((fast-prime? start-num 5)
         (begin
           (newline)
           (display start-num)
           (display " is prime num, cost ")
           (display (- (runtime) start-time))
           (display " millsecond")))
        (else (search-for-primes-iter (next-odd start-num) end-num
start-time))))
```

比如跑 (search-for-primes 1000000000000000 10000000000000000) 费马检测耗时是697微秒而前者是373504微妙即3秒。 但是并没有达到$\frac{\sqrt{n}}{\mathrm{Log(n)}}$的倍率。 同时对于自身计算100000000跟(search-for-primes 10000000000000000 100000000000000000)的结果发现耗时有波动， 3倍-4倍， 差不对是对数比值。$\frac{\mathrm{Log}[n]}{\mathrm{Log}[m]}$。不过通过对比体现出了费马检测的算法时间的优越性。

总之上面的三道题在我的电脑配置下波动有点大。

1.25 解答

这道题问的是expmod的本体改成fast-exp是否可行,相比前一题, 这里附上完整代码

```
(define (fast-expt b n)
  (fast-expt-iter b n 1))
(define (fast-expt-iter b n ans)
  (cond ((= n 0) ans)
        ((even? n) (fast-expt-iter (* b b) (/ n 2) ans))
        (else (fast-expt-iter b (- n 1) (* ans b)))))
(define (expmod base exp m)
  (remainder (fast-expt base exp) m));这里进行改写
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a n n) a))
  (try-it (+ 1 (random (- n 1)))))
(define (fast-prime? n times)
  (cond ((= times 0) #t)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else #f)))
(define (square x)
  (* x x))
(define (runtime)
  (let ((x (gettimeofday)))                ; 函数返回的是(second . millsecond)
    (+ (* (car x) 100000) (cdr x))))       ;微秒
(define (next-odd n)
  (if (even? n)
      (+ n 1)
      (+ n 2)))

(define (search-for-primes start-num end-num)
  (search-for-primes-iter start-num end-num (runtime)))
(define (search-for-primes-iter start-num end-num start-time)
  (cond ((> start-num end-num)
         (display "not found"))
        ((fast-prime? start-num 5)
         (begin
           (newline)
           (display start-num)
           (display " is prime num, cost ")
           (display (- (runtime) start-time))
           (display " millsecond")))
        (else (search-for-primes-iter (next-odd start-num) end-num
start-time))))
```

这个问题一开始觉得没啥问题, 我做了验证发现结果和耗时上没啥问题, 但是当跑很大的数字时guile直接coredump了。

```
scheme@(guile-user)> (search-for-primes 10000000000 100000000000)
段错误 (核心已转储)
```

所以仔细看了fast-expt就得出原因了。 因为fast-expt先会求出$base^{exp}$。 这个数字可能会很大。 而原先的expmod则是将数字拆小了降幂处理。

1.26 解答

问题问的是expmod过程中的square被改成了*之后程序变慢了。改写之后的程序如下：

```
;; expmod 求得是base^exp % m
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)                        ;exp是偶数 可以变成 下面的代码
         ;; a^m=bk+c
         ;; (a^m)^2 = b^2 k^2+2bck+c^2=b(bk^2+2ck)+c^2
         ;; 所以 base^exp%m的余数是base^(exp/2)%m的平方倍数 这里将square改成*
         (remainder (* (expmod base (/ exp 2) m)) (expmod base (/ exp 2) m))
                    m))
        ;; a^m=bk+c
        ;; a^(m+1)=abk+ac 所以余数自然就是exp-1的base倍数
        (else (remainder (* base (expmod base (- exp 1) m)) ;同理 不再赘述
                         m))))
```

原因很明显忽略squre这段代码， 整个函数的过程调用是折半的照着exp大小进行折半。但是将square进行改写为*之后， 进入了两个递归导致(expmod base (/ exp 2)m)要计算两次折半的效果被抵消， 。

写成数学函数大概就是含有square的递推公式如下(把remainder和一些计算的代价看成1)

$$f(n)=\begin{cases} f\left(\frac{n}{2}\right) & n\text{是偶数} \\ f(n-1) & n\text{是奇数} \\ 1 & n=0 \end{cases}$$

而将square改写成*之后的递推公式如下

$$f(n)=\begin{cases} 2\,f\left(\frac{n}{2}\right) & n\text{是偶数} \\ f(n-1) & n\text{是奇数} \\ 1 & n=1 \end{cases}$$

1.27 解答

其实就是吧Carmichael数作为参数调用费马检查函数。

```
scheme@(guile-user)> (fast-prime? 561 5)
$1 = #t
```

显然561不是素书至少可以被3整除。

1.28 解答

代码如下， 该函数的问题就是$(\mathrm{fermat}-\mathrm{test}\ n)$的参数必须大于0啦， 函数实现有问题

```
(define (expmod base exp m)
  (cond ((= exp 0) 1)
        ((even? exp)                        ;exp是偶数 可以变成 下面的代码
         ;; a^m=bk+c
         ;; (a^m)^2 = b^2 k^2+2bck+c^2=b(bk^2+2ck)+c^2
         ;; 所以 base^exp%m的余数是base^(exp/2)%m的平方倍数
         (remainder (square (expmod base (/ exp 2) m))
                    m))
```

```
          ;; a^m=bk+c
          ;; a^(m+1)=abk+ac 所以余数自然就是exp-1的base倍数
          (else (remainder (* base (expmod base (- exp 1) m)) ;同理 不再赘述
                           m))))
(define (fermat-test n)
  (define (try-it a)
    (= (expmod a (- n 1) n) 1))                  ; 这里做了改动 a^(n-1) mod n = 1
  (try-it (+ 1 (random (- n 1)))))
(define (fast-prime? n times)
  (cond ((= times 0) #t)
        ((fermat-test n) (fast-prime? n (- times 1)))
        (else #f)))
(define (square x)
  (* x x))
```

前面提到的Carmichael数也不再返回true， 当然数学原理什么的不懂， 不去纠结。需要注意的是 变形后的费马检查同样是概率检查， 并不能百分百筛出Carmichael数， 同样存在这样的数字让程序结果返回true。为了降低被欺骗的可能性。文中提到还可以继续检查， 但是结果为true的情况下检查输入的a其不是"不等于1和n-1的情况下a^2 mod n = 1"若存在则n不是素数， 因此进一步改写

高阶函数

高阶函数就是以函数为输入或返回值的函数。比如下面的求和可以拆解

$$\sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b)$$

```
;; 求和的抽像，term a 就是f(a) (next a)就是如何产生下一个数字
;; 这就是求f(a)+f(a+1)+.....+f(b)
(define (sum term a next b)
  (if (> a b)
      0
      (+ (term a)
         (sum term (next a) next b))))
```

现在来利用上面的例子实现定积分求法

$$\int_a^b f = \left[f\left(a + \frac{\mathrm{dx}}{2}\right) + f\left(a + \mathrm{dx} + \frac{\mathrm{dx}}{2}\right) + f\left(a + 2\mathrm{dx} + \frac{\mathrm{dx}}{2}\right) + \cdots + \right] \mathrm{dx}$$

```
;; 定积分
(define (integral f a b dx)
  (define (add-dx x) (+ x dx))
  (* (sum f (+ a (/ dx 2.0)) add-dx b)
     dx))
;; 求x^3在[0,1]的定积分
(integral cube 0 1 0.01)
```

1.29 解答 辛普森规则求定积分

辛普森规则是一种比上面方法更好的求数值积分的算法， 求函数f在a和b之间的定积分的近似值是:

$$\frac{h}{3}[y_0 + 4y_1 + 2y_2 + 4y_3 + 2y_2 + \cdots + 2y_{n-2} + 4y_{n-1} + y_n]$$

其中$h=(b-a)/n$ ， n是某个偶数而$y_k=f(a+\mathrm{kh})$， 增大n能提高近似指的精度。定义一个函数求出cube在0到1之间的积分（n分别取值100和1000）， 并对比上面的结果

```scheme
(define (simpson-rule f a b n)
  ;; 此处定义成数字变量是否更好 (define h (/ (- b a) n))
  (define h (/ (- b a) n))
  (define (next k) (+ a (* k h)))
  (define (inc n) (+ n 1))
  (define (term i)
    (cond ((= i 0) (f (next i)))
          ((= i n) (f (next i)))
          ((even? i) (* 2 (f (next i))))
          (else (* 4 (f (next i))))))
  (* (sum term 0 inc n)
     (/ h 3)))
```

运行结果本该是$n=1000$时的结果比$n=100$时更好。但是我使用的scheme解释器是gun/guile。求出来的是精确值， 反而没法对比了， 只能说确实比前者好

```
scheme@(guile-user)> (simpson-rule cube 0 1 1000)
$7 = 1/4
scheme@(guile-user)> (simpson-rule cube 0 1 100)
$8 = 1/4
scheme@(guile-user)> (simpson-rule cube 0 1 50)
$9 = 1/4
scheme@(guile-user)> (simpson-rule (lambda (x) (* 2 x))  0 1 50)
$10 = 1
scheme@(guile-user)> (simpson-rule (lambda (x) (* 3 x))  0 1 50)
$11 = 3/2
scheme@(guile-user)>
```

1.30 解答 实现sum函数的迭代版本

```scheme
;; 实现上面sum函数的迭代版本
(define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ (term a) result))))
  (iter a 0))
```

测试结果如下

```
scheme@(guile-user)> (define (sum term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (+ (term a) result))))
  (iter a 0))
scheme@(guile-user)> (simpson-rule cube 0 1 100)
$14 = 1/4
scheme@(guile-user)> (simpson-rule (lambda (x) (* 3 x))  0 1 50)
$15 = 3/2
scheme@(guile-user)> (simpson-rule (lambda (x) (* 2 x))  0 1 50)
$16 = 1
scheme@(guile-user)>
```

1.31 解答 写出连乘的高阶函数

写出连乘法的高阶函数并据此实现factorial和常数π的近似值公式:

$$\frac{\pi}{4}=\frac{2}{3}\frac{4}{3}\frac{4}{5}\frac{6}{5}\frac{6}{7}\frac{8}{7}\cdots$$

product的迭代和递归版本如下:

```
;; 实现一个连乘的类似sum的函数
;; 需要实现递归和迭代两个版本
(define (product term a next b)
  (if (> a b)
      1
      (* (product term (next a) next b)
         (term a))))
(define (product term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (* result (term a)))))
  (iter a 1))
```

连乘的代码如下

```
(define (factorial n)
  (define (identity x) x)
  (define (inc n) (+ n 1))
  (product identity 1 inc n))
```

π的近似实现先分析公式的实现

$$\frac{\pi}{4}=\prod_{n=1}^{n} f(n)$$

其中$f(n)$的公式如下

$$f(n)=\begin{cases}\frac{n+1}{n+2} & ,n\text{是奇数}\\ \frac{n+2}{n+1} & ,n\text{是偶数}\end{cases}$$

所以写成代码就是

```
;; pi/4 的近似值实现如下
(define (pi n)
  (define (term n)
    (if (even? n)
        (/ (+ n 2) (+ n 1))
        (/ (+ n 1) (+ n 2))))
  (define (inc n) (+ n 1))
  (product term 1 inc n))
```

运行结果测试如下

```
scheme@(guile-user)> (pi 1)
$22 = 2/3
scheme@(guile-user)> (pi 2)
$23 = 8/9
scheme@(guile-user)> (pi 3)
$24 = 32/45
```

```
scheme@(guile-user)> (pi 4)
$25 = 64/75
scheme@(guile-user)>
```

1.32 解答

题目要求对前面两题的sum和product做进一步的抽象， 总结出两者都是一种累计的结构。同时分别写出迭代和递归两种版本。

```
;; 递归版本
(define (accumulate combiner null-value term a next b)
  (if (> a b)
      null-value
      (combiner (term a)
                (accumulate combiner null-value term (next a) next b))))
;; 迭代版本
(define (accumulate combiner null-value term a next b)
  (define (iter a result)
    (if (> a b)
        result
        (iter (next a) (combiner result (term a)))))
  (iter a null-value))
```

然后重新实现factorial

```
;; 连乘 测试代码
(define (factorial n)
  (define (term  n) n)
  (define (next n) (+ n 1))
  (accumulate * 1 term 1 next n))
```

测试结果

```
scheme@(guile-user)> (factorial 5)
$28 = 120
scheme@(guile-user)> (factorial 2)
$29 = 2
scheme@(guile-user)>
```

1.33 解答 对上一题做进一步抽象和优化

这里要求实现一个filter让accumulate更一般化。按照题目意思应该是用filter函数传入数值和谓词符合条件则返回数字不符合则返回null-value， accumulate目前是只能处理一个区间内的全部数据， 将之替换之后则更通用， 比如一个区间内的偶数和等等。由于前面的素数检测似乎实现的有问题这里使用偶数和替代， 顺带减少代码量

```
(define (filtered-accumulate combiner predic null-value term a next b)
  (define (iter a result)
    (cond ((> a b) result)
          ((predic a) (iter (next a) (combiner result (term a))))
          (else (iter (next a) result))))
  (iter a null-value))
;; 区间a-b的偶数和
(define (sum-of-even a b)
  (define (term  n) n)
  (define (next n) (+ n 1))
```

```
  (filtered-accumulate + even? 0 term a next b))
```

1.34 解答

问题问如下的代码的运行情况， 结果应该是报错。因为(2 2)不是一个过程

```
(define (f g)
  (g 2))
;; 问题问如果 (f f)会怎样
```

过程作为一般性方法

下面代码是找出零点的函数， 书中原文， 做了一些补齐能在guile/scheme中运行

```
;; 1.3.3 找函数的根 零点
;; 这个问题就是 输入的两个点的值 可能并不具备一正一负的特点
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value) (search f neg-point midpoint))
                ((negative? test-value) (search f midpoint pos-point))
                (else midpoint))))))
(define (average a b)
  (/ (+ a b) 2))
(define (close-enough? x y)
  (< (abs (- x y)) 0.001))
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-value (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else (error "Values are not of opposite sign" a b)))))
```

接下来是一个不动点的计算方法，$f(x) = x$的点。存在某些函数可以通过$f(x), f(f(x)), f(f(f(x)))....$去逼近不动点， 因此可以写出如下的代码

```
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

比如求出$y = \sin y + \cos y$的解

```
scheme@(guile-user)> (fixed-point (lambda (y) (+ (sin y) (cos y))) 1.0)
$7 = 1.2587315962971173
scheme@(guile-user)>
```

1.35 解答 不动点求黄金分割

证明黄金分割率ϕ是变换$x \longmapsto 1+\frac{1}{x}$的不动点， 并利用不动点函数写出求值函数。原书的公式是

$$x^2 = x + 1$$

两边同时除以x即可:

$$x = 1 + \frac{1}{x}$$

则写成代码如下

```
(fixed-point (lambda (x) (+ (/ 1 x) 1)) 1)
```

执行结果如下

```
scheme@(guile-user)> (fixed-point (lambda (x) (+ (/ 1 x) 1)) 1)
$8 = 987/610
scheme@(guile-user)> (fixed-point (lambda (x) (+ (/ 1 x) 1)) 1.0)
$9 = 1.6180327868852458
scheme@(guile-user)>
```

1.36 解答

这道题要求给fix-point函数添加代码输出每一步的中间x值

```
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (newline)
      (display next)
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

执行结果如下

```
scheme@(guile-user)> (fixed-point (lambda (x) (+ (/ 1 x) 1)) 1.0)

2.0
1.5
1.6666666666666665
1.6
1.625
1.6153846153846154
1.619047619047619
1.6176470588235294
1.6181818181818182
1.6179775280898876
1.6180555555555556
1.6180257510729614
1.6180371352785146
1.6180327868852458$12 = 1.6180327868852458
scheme@(guile-user)>
```

1.37 解答

无穷连分式如下

$$f = \frac{N_1}{D_1 + \frac{N_2}{D_2 + \frac{N_3}{D_3 + \cdots + .}}}$$

现在当N_i和D_i都为1时， 该式子产生的结果是黄金分割率的倒数$\frac{1}{\phi}$。实现一个函数输入k给出截断后的k项有限连分式。题目要求写出递归和迭代两种方法

```
;; 递归的方法
(define (n k) 1.0)
(define (d k) 1.0)
(define (cont-frac n d k)
  (define (try counter)
    (if (= counter k)
      (/ (n counter) (d counter))
      (/ (n counter) (+ (d  counter) (try (+ counter 1))))))
  (try 1))
```

输出结果如下

```
scheme@(guile-user)> (cont-frac n d 10)
$15 = 0.6180555555555556
scheme@(guile-user)> (cont-frac n d 100)
$16 = 0.6180339887498948
scheme@(guile-user)>
```

现在写出迭代式。迭代式写法跟递归的差别在于递归式可以从第一个项开始， 而迭代式需要从第k项开始。

```
;; 迭代方法
(define (cont-frac n d k)
  (define (iter result counter)
    (if (= counter 0)
        (/ (n counter) (+ (n counter) result))
        (iter (/ (n counter) (+ (n counter) result)) (- counter 1))))
  (iter (n k) k))
```

测试结果如下

```
scheme@(guile-user)> (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 10)
$21 = 0.6180257510729613
scheme@(guile-user)> (cont-frac (lambda (i) 1.0) (lambda (i) 1.0) 100)
$22 = 0.6180339887498948
scheme@(guile-user)>
```

当k=9时， 结果具有十进制的4位精度

1.38 解答 利用连分式过程求自然对数的底e

```
(define (n k) 1.0)
(define (d k)
  (if (= (remainder k 3) 2)
      (* 2 (+ (quotient k 3) 1))
      1))
(cont-frac n d 100)
```

结果如下， 将结果加上2即可

```
scheme@(guile-user)> (cont-frac n d 100)
$24 = 0.7182818284590453
scheme@(guile-user)>
```

1.39 解答 连分式求正切值

先看下数学定义， 其中x是弧度

$$\tan x = \cfrac{x}{1 - \cfrac{x^2}{3 - \cfrac{x^2}{5 - \cdots -}}}$$

可以看到过程D_i是一个奇数序列， 而过程N_i则是除了N_1是x其余都是$-x^2$

```
(define (tan-cf x k)
  (define (d k)
    (- (* 2 k) 1.0))
  (define (square x)
    (* x x))
  (define (n k)
    (if (= k 1)
        x
        (- (square x))))
  (cont-frac n d k))
```

测试结果如下

```
scheme@(guile-user)> (tan-cf 3 100)
$31 = -0.14254654307427775
scheme@(guile-user)> (tan 3)
$32 = -0.1425465430742778
scheme@(guile-user)> (tan 9)
$33 = -0.45231565944180985
scheme@(guile-user)> (tan-cf 9 10)
$34 = -0.6248168140610412
scheme@(guile-user)> (tan-cf 9 100)
$35 = -0.45231565944180974
scheme@(guile-user)>
```

从这道题可以发现对比上道题， 这道题的连分式需要更多的项即k更大时结果才能更好。

知识点回顾

接下来的题目由于需要复用牛顿开平方， 这里特地回复知识点。要求$\sqrt{x}$的值有一种牛顿逐步逼近的方法。比如要求$\sqrt{2}$， 先假定猜测初始值是1:

猜测	商	平均值
1	$\frac{2}{1} = 2$	$\frac{2+1}{2} = 1.5$
1.5	$\frac{2}{1.5} = 1.3333\ldots$	$\frac{1.3333+1.5}{2} = 1.4167$
1.4167	$\frac{2}{1.4167} = 1.4118$	$\frac{1.4167+1.4118}{2} = 1.4142$
1.4142	$\ldots$	

也就是说$\frac{\frac{x}{\text{guess}} + \text{guess}}{2}$能比原先猜测值更接近$\sqrt{x}$。 本节刚好介绍了平均阻尼刚好可以用该过程来i表示求平均值的过程， 而不动点过程可以用来表示该逼近过程

```scheme
;; 平均阻尼
;; ( x + f(x)) /2 x和f的平均值
;; 所以 平均阻尼可以用在求平方根上
(define (average-damp f)
  (lambda (x) (average x (f x))))
;; 而不动点就是f(x)=x的点 y^2=x , 刚好结合平均阻尼 可以用来求
;; 平方根，真牛逼啊 这
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

接下来还引申出函数特例当$x \longmapsto g(x)$可微时，$g(x)=0$的一个解就是$x \longmapsto f(x)$的一个不动点， 其中

$$f(x)=x-\frac{g(x)}{\mathrm{Dg}(x)}$$

1.40 解答

通过 对比书中的牛顿求平方根的例子 g是$y^2-x=0$ 零点。所以cubic也是直接写出函数即可

```scheme
;; 1.40
(define tolerance 0.00001)
(define (fixed-point f first-guess)
  (define (close-enough? v1 v2)
    (< (abs (- v1 v2)) tolerance))
  (define (try guess)
    (let ((next (f guess)))
      (newline)
      (display next)
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
(define dx 0.00001)
(define (deriv g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))
;; x - g(x)/Dg(x)
(define (newton-transform g)
  (lambda (x)
    (- x (/ (g x) ((deriv g) x)))))
(define (newton-method g guess)
  (fixed-point (newton-transform g) guess))
(define (cubic a b c )
  (lambda (x)
    (+ (* x x x)
       (* a x x)
       (* b x)
       c)))
(newton-method (cubic 1 1 0) 1)
```

一个样例输出如下

```scheme
(newton-method (cubic (- (/ 9 2)) (/ 9 2) 0) 1)
```

```
1.6666600001110894
1.4957266272198368
1.5000000691263737
1.5$4 = 1.5
scheme@(guile-user)>
```

之所以选这么奇怪的a, b, c是因为根值比较好。来自于$x\left(x-\frac{3}{2}\right)(x-3)$。

1.41 解答

```
(define (double proc)
  (lambda (x)
    (proc (proc x))))
;; 问下面的代码输出是多少
(define (inc x)
  (+ x 1))
(((double (double double)) inc) 5)
```

21

1.42 解答

```
(define (compose f g)
  (lambda (x)
    (f (g x))))
((compose square inc) 5)
```

1.43 解答

题设的意思是可以利用上一题的函数做折半, 我偏不。

```
(define (repeated f n)
  (define (iter result counter)
    (if (= counter n)
        (f result)
        (iter (f result) (+ counter 1))))
  (lambda (x)
    (if (< n 1)
        (error "n must be grater than 0: " n)
        (iter x 1))))
```

测试输出如下

```
scheme@(guile-user)> ((repeated (lambda (x) (+ x 1)) 10) 0)
$60 = 10
scheme@(guile-user)> ((repeated (lambda (x) (* x x)) 2) 5)
$61 = 625
scheme@(guile-user)>
```

1.44 解答

```
;; 利用上一题的函数实现smooth函数,
(define dx 0.00001)
(define (smooth f n)
  (repeated
```

```
    (lambda (x)
      (/ (+ (f (- x dx) )
            (f x)
            (f (+ x dx)))
         3))
    n))
```

测试输出如下

```
scheme@(guile-user)> ((smooth (lambda (x) (* x x) ) 2) 5)
$62 = 625.0000000033998
scheme@(guile-user)> ((repeated (lambda (x) (+ x 1)) 10) 0)
$63 = 10
scheme@(guile-user)>
```

1.45 解答 略

这道题比较简单， 直接参照上面的代码即可。要重复很多次实验， 麻烦 略

1.46 解答

```
(define (iterative-improve f good-enough? improve)
  (define (iter x)
    (if (good-enough? x (improve x))
        x
        (iter (improve x))))
  (lambda (x)
    (iter x)))
```

测试代码如下， 用的是求黄金分割率的过程

```
(define tolerance 0.00001)
((iterative-improve (lambda (x) (+ (/ 1 x) 1))
                    (lambda (v1 v2)
                      (< (abs (- v1 v2)) tolerance))
                    (lambda (x)
                      (+ (/ 1 x) 1)))
 1.0)
```

输入如下

```
scheme@(guile-user)> ((iterative-improve (lambda (x) (+ (/ 1 x) 1))
                    (lambda (v1 v2)
                      (< (abs (- v1 v2)) tolerance))
                    (lambda (x)
                      (+ (/ 1 x) 1)))
 1.0)
$67 = 1.6179775280898876
scheme@(guile-user)> (define tolerance 0.00001)
((iterative-improve (lambda (x) (+ (/ 1 x) 1))
                    (lambda (v1 v2)
                      (< (abs (- v1 v2)) tolerance))
                    (lambda (x)
                      (+ (/ 1 x) 1)))
 1.0)

$68 = 1.6180371352785146
```

```
scheme@(guile-user)>
```

2 第二章