1.3-1.8

2022年12月19日 14:26

1.3

定义一个过程，以三个数为参数，返回其中较大地两个数之和

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

1.5

```
;定义一个死循环递归
(define (p) (p))

;检测解释器是应用序还是正则序
(define (test x y)
  (if (= x 0)
      0
      y))
(test 0 (p))
```

应用序：要先求值参数，求(p)时就会进入(p)的死循环

正则序：完全展开而后归约，会正常返回0，在执行(test 0 (p))后就替换为(if (= 0 0) 0 (p))，条件表达式满足，(p)就被忽略掉了，所以会正常返回0。

用lazyracket这个惰性求值的解释器，可以返回0。

1.6

为什么不能通过cond将if定义为一个常规过程呢？也就是写成：

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

看起来好像可以，用new-if重写求平方根的程序：

```
;自己定义new-if
(define (new-if predicate then-clause else-clause)
  (cond (predicate then-clause)
        (else else-clause)))
;改写
(define (sqrt-iter guess x)
   (new-if (good-enough? guess x)
           guess
           (sqrt-iter (improve guess x) x)))
```

这样写会发生什么呢?

答案是：

程序运行不完，在 new-if 还没有展开为 cond special forms 时，else-clause 子式已经陷入了无限递归。会递归直到堆栈溢出。

因为是应用序，先求值参数而后应用，程序一直在计算下一个预测值，没有应用自己写的new-if，没有递归出口自然也就停不下来。

特殊块if就不会有这个问题，new-if是自己定义的函数，其应用要遵守应用序的规则

1.7

对于确定很小的数的平方根而言，在计算平方根中使用的good-enough?是很不好的。在现实的计算机中，算术运算总是以一定的有限精度进行，这也会使这个检测不适合非常大的数。

例如：

精度设置为0.0001时

输入：(sqrt 0.0002)

输出：0.014920008896897232

这个结果偏差就很大

实现good-enough?的另一种策略是监视猜测值的变化情况，当改变值先对于猜测值的比率很小时就结束，也就是这样：

```
;迭代时要增加保存一个last值，用来做判断
(define (sqrt-iter last guess x)
  (if (good-enough? last guess x)
      guess
      (sqrt-iter guess (improve guess x) x)))

;精度判断（以变化率为指标）
(define (good-enough? last guess x)
  (< (/ (abs (- guess last)) last) 0.000000001))

(define (improve guess x)
  (average guess (/ x guess)))

(define (average x y)
  (/ (+ x y) 2))

(define (square x)
  (* x x))

(define (sqrt x)
      (sqrt-iter 0.5 1.0 x))
```

1.8

计算立方根：

```
(define (cube-root x)
  (define (cube-root-iter guess x)
    (if (good-enough? guess x)
        guess
        (cube-root-iter (improve guess x) x)))
  (define (good-enough? guess x)
    (< (abs (- (square guess) x)) 0.0001))
  (define (improve guess x)
    (average guess (/ x guess)))
  (define (average x y)
    (/ (+ (* 2 x) y) 3))
  (define (cube x)
    (* x x x))
  (define (square x)
    (* x x))
  (cube-root-iter 1.0 x))
```

1.9-1.12

2022年12月31日 10:31

1.9

两种定义两个参数相加的方法:

1.递归法:

```
(define (+ a b)
  (if (= a 0)
      b
      (inc (+ (dec a) b))))
```

2.迭代法:

```
(define (+ a b)
  (if (= a 0)
      b
      (+ (dec a) (inc b))))
```

1.10

Ackermann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))))))
```

考虑下面过程的数学定义:

```
(define (f n) (A 0 n))
(define (g n) (A 1 n))
(define (h n) (A 2 n))
```

答:

(f n)计算的是2n

(g n)计算的是2^n

解释:

对于g(1,n), n>=1

当 n > 1 时，根据上一题，有

g(1,n) = f(0, g(1, n-1)) = 2*g(1,n-1)，以此类推

g(1,n) = $2^{n-1} * g(1,1)$

当 n = 1 时，

g(1,1) = 2 = 2^1

故 g(1,n) = 2^n, n>0.

执行步骤:

```
 (A 1 n)
=(A 0 (A 1 n-1))
=(A 0 (A 0 (A 1 n-2)))
=(A 0 (A 0 (A 0 (……(A 0 (A 0 (A 1 1)))))
=(A 0 (A 0 (A 0 (……(A 0 (A 0 2)))
```

=(A 0 (A 0 (A 0 (……(A 0 2^2))

=(A 0 (A 0 (A 0 (……2^3))

=2^n

所以 (A 1 10) =1024

(h n)计算的是2^(2^(...   (n个2)

解释:

对于h(2,n), n>0

n > 1 时，根据上一个题:

h(2,n) = g(1, h(2, n-1)) = 2^h(2,n-1),

直到 n = 1, f(2,1) = 2

h(2,n) = 2^(2^(... (n个2)

1.11

用递归和迭代两种方法定义函数(f n):

如果n<3，那么f(n)=n；如果n>=3，那么f(n)=f(n-1)+2f(n-2)+3f(n-3)。

```
#lang sicp
;递归
(define (f n)
  (if (< n 3)
      n
      (+ (f (- n 1))
         (* 2 (f (- n 2)))
         (* 3 (f (- n 3))))))

;迭代
(define (f-iter a b c n counter)
  (if (= counter n)
      c
      (f-iter b
              c
              (+ (f (- n 1))
                 (* 2 (f (- n 2)))
```

```
                  (* 3 (f (- n 3))))
               n
               (+ counter 1))))
(define (f2 n)
  (if (< n 3)
      n
      (f-iter 0 1 2 n 2)))
```

1.12

帕斯卡三角形 (pascal)

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

三角形边界上的数都是1，内部的每个数是位于它上面的两个数之和。用递归计算出帕斯卡三角形的某一个元素。形式：(pas n k)，n是行，k是每行中第几个。

答：

递归等式： (pas n k) = (pas (- n 1) (- k 1)) + (pas (- n 1) k)

递归出口：三角形边界上的数都是1：k=1 or k=n 时(pas n k)=1

```
(define (pas n k)
  (if (or (= k 0) (= k n))
      1
      (+ (pas (- n 1) (- k 1))
         (pas (- n 1) k))))
```

1.14-1.18

2023年1月1日 17:37

1.14

画出相关的树，展示换零钱count-change函数在将11美分换成硬币时所产生的计算过程。这一计算过程的空间和时间增长阶各是什么?

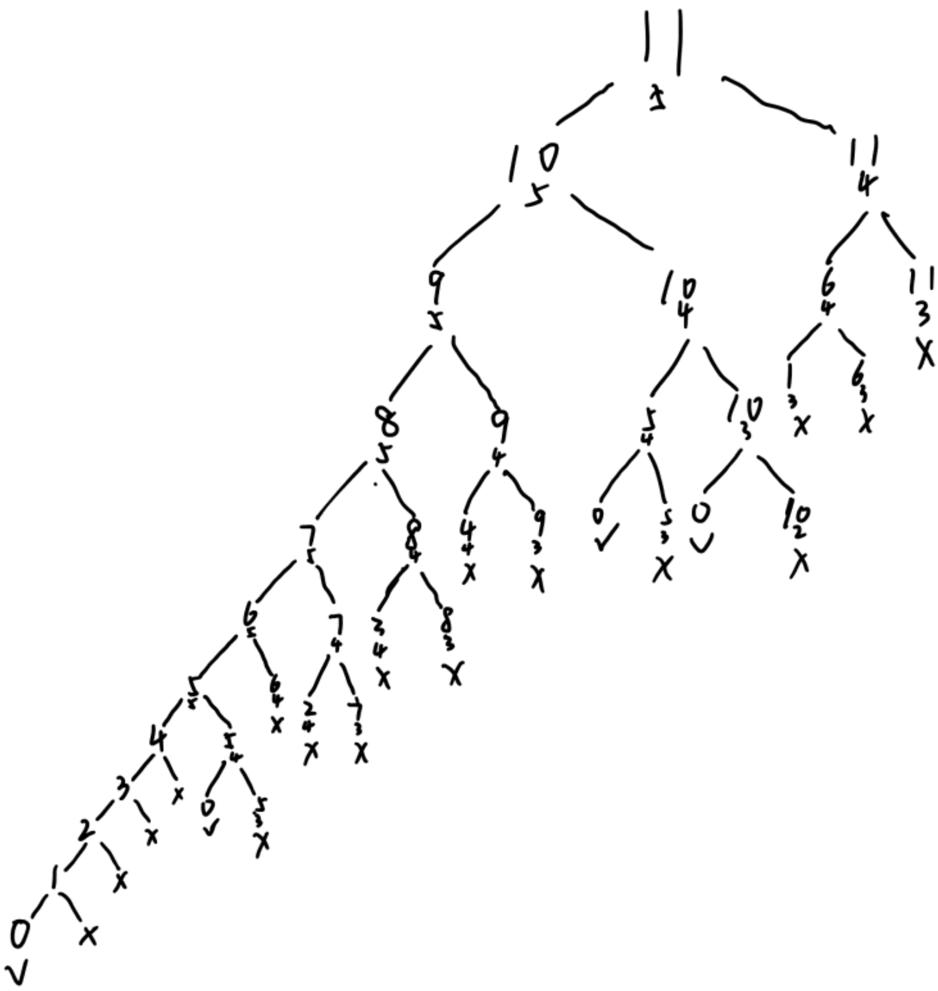
函数如下:

```
#lang sicp
(define (count-change amount)
  (cc amount 5))

(define (cc amount kinds-of-coins)
  (cond ((= amount 0) 1)
        ((or (< amount 0)(= kinds-of-coins 0)) 0)
        (else (+ (cc amount
                     (- kinds-of-coins 1))
                 (cc (- amount
                        (first-denomination kinds-of-coins))
                     kinds-of-coins)))))

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 50)
        ((= kinds-of-coins 2) 25)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 5)
        ((= kinds-of-coins 5) 1)))
```

树如图所示：共有四种换法



计算过程的空间需求，也就是树的深度，递归嵌套最深的就是全部用1块来找换，

<https://blog.csdn.net/qq_33974741/article/details/84488245?ops_request_misc=%257B%2522request%255Fid%2522%253A%2522167257802216800215044362%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=167257802216800215044362&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~baidu_landing_v2~default-1-84488245-null-null.142^v68^control,201^v4^add_ask,213^v2^t3_esquery_v3&utm_term=sicp1.14&spm=1018.2226.3001.4187>

1.15

在角（弧度制）x足够小时，其正弦值可以用 sinx≈ x 计算，而三角恒等式：

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

可以减小sin的参数的大小（这里认为一个角是”足够小“，如果数值不大于0.1弧度）。过程如下：

```
(define (cube x) (* x x x))

(define (p x) (- (* 3 x) (* 4 (cube x))))

(define (sine angle)
  (if (not (> (abs angle) 0.1))
      angle
      (p (sine (/ angle 3.0)))))
```

问:

a) 在求值(sine 12.15)时，p将被使用多少次?

b) 在求值(sine a)时，由过程sine所产生的计算过程使用的空间和步数增长的阶是什么?

答:

a) 12.15 连除 5次 3 小于 0.1 ，所以是 5次

b) 可以看出每调用一次 p 过程，需要递归1次 sine ，空间加1，计算步数加2，关键是p的次数:

对于a，调用次数t，那么 a*3^(-t) < 0.1 ，即 10a < 3^t ==> lg(10a)/lg3 < t，

所以增长阶 空间和时间 都为 Θ(log a)

1.16

定义一个迭代过程求幂，其中使用一系列的求平方，就像fast-expt只用对数个步骤那样。(利用关系(b^n/2)^2=(b^2)^n/2)，除了指数n和基数b之外，还应维持一个附加的状态变量a，并定义好状态变换，使得从一个状态转到另一个状态时乘积a*b^n不变。在计算过程开始时令a取值1并用计算过程结束时的a的值作为回答。

```
#lang sicp
(define (square x)
  (* x x))

(define (fast-expt b n )
  (fast-expt-iter b n 1))

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

过程中，每当遇到n为奇数时，就给a乘上一个那时的b（可能已经经过了好几次square的新的b）

1.17

求幂算法的基础是反复做乘法，那么也可以通过反复做加法的方式求乘积，如:

```
(define (* a b)
  (if (= b 0)
      0
      (+ a (* a (- b 1)))))
```

这一算法具有相对于b的线性步数，现在假设还有double（求一个整数的2倍）和halve（将一个偶数除以2），使之只用对数的计算步骤。请用这些运算设计一个类似fast-expt的求乘积过程。

```
(define (double x) (* x 2))
(define (halve x) (/ x 2))

(define (muti-new a b)
  (cond ((= b 0) 0)
        ((even? b) (double (muti-new a (halve b))))
        (else (+ a (muti-new a (- b 1))))))
```

1.18

利用1.16和1.17的结果设计一个过程，它能产生出一个基于加、加倍和折半运算的迭代计算过程，只用对数的步数就能求出两个整数的乘积。

```
(define (double x) (* x 2))
(define (halve x) (/ x 2))

(define (muti-new a b)
  (define (muti-new-iter a b c)
  (cond ((= b 0) c)
        ((even? b) (muti-new-iter (double a) (halve b) c))
        (else (muti-new-iter a (- b 1) (+ c a)))))
  (muti-new-iter a b 0))
```

2.4

2023年2月9日 14:48

2.4

对序对除了dispatch以外的另一种过程性表示：

```
(define (cons x y)
  (lambda (m) (m x y)))

(define (car z)
  (z (lambda (a b) a)))

(define (cdr z)
  (z (lambda (a b) b)))
```

它是怎样执行的?

解释：

输入(car (cons x y)) => (car (lambda (m) (m x y)))

=> ((lambda (m) (m x y)) (lambda (a b) a)) （这里的(lambda (a b) a)就是(lambda (m) (m x y))的参数，所以代进去）

=> ((lambda (a b) a) x y) => x

2.23

2023年2月10日 0:14

```
(define (map p sequence)
  (accumulate (lambda (x y) (cons (p x) y)) nil sequence))

(define (append seq1 seq2)
  (accumulate cons seq2 seq1))

(define (length sequence)
  (accumulate (lambda (a b)
                (+ 1 b))
              0
              sequence))
```

2.27

2023年2月9日 22:46

```
;练习2.27
(define (deep-reverse items)
  (cond ( (null? items) items)
        ( (not (pair? items)) items)
        ( (null? (cdr items)) items)
        (else (deep-reverse-iter (cdr items) (cons (deep-reverse (car
items)) nil)))))

(define (deep-reverse-iter a b)
  (if (null? a)
      b
      (deep-reverse-iter (cdr a) (cons (deep-reverse (car a)) b))))
```

2.28

构造过程fringe。(fringe items)返回树的叶节点构成的表

例子：(fringe (list (list 1 2) 3 (list 4 5))) = (1 2 3 4 5)

2.33

2023年2月10日 13:20

使用累积定义一些基本的表操作

```
(define (map p sequence) (accumulate (lambda (x y) <>) nilsequence))
(define (append seq1 seq2) (accumulate cons <> <>))
(define (length sequence) (accumulate <> 0 sequence))
```

accumulate是这样的:

```
define (accumulate op initial items)
  (if (null? items)
      initial
      (op (car items)
          (accumulate op initial (cdr items)))))
```

答案:

```
(define (map+ p seque)
 (accumulate (lambda(x y) (cons (p x) y)) null seque))

(define (append seq1 seq2)
 (accumulate cons seq2 seq1))

(define (length seque)
 (accumulate (lambda(x y) (+ 1 y)) 0 seque)
```