第一章:构造过程抽象

心智的活动,除了尽力产生各种简单的认识之外,主要表现在如下三个方面:
1)将若干简单的认识组合为一个复合认识,由此产生出各种复杂的知识.
2)将两个认识放在一起对照,不管它们如何简单或者复杂,在这样做时并不将它们合二为一.由此得到有关它们的相互关系的认识.
3)将有关认识与那些在实际中和它们所同在的有关其它认识隔离开,这就是抽象,所有具有普遍性的认识都是这样得到的.
John Locke 有关人类理解的随笔 1690

这一章是开篇,主要介绍的是 Scheme 语言的部分语法,并且通过使用简单的语法去达到一些不那么简单的数学问题.
注意:在第三章以前都不会用到赋值,所以并没有大家习惯中的"循环",更多地通过递归来实现.

练习:
1.1 略
1.2 略
1.3
CODEBOX
```
(define (max a b c)
  (if (> a b)
      (if (> a c) a c)
      (if (> b c) b c)))
```

1.4
这题为读者模糊地提供了一个高阶函数的概念,还有过程和数据的统一结合.

1.5
本题的知识点是正则序和应用序的区别.
考虑
```
(define (p) (p))
```
一旦(p)被求值,这个程序就会死掉.
在正则序中,对函数的求值是这样的:
先展开到只剩基本过程,再对其中"需要求值"的部分进行求值.
在应用序中,函数的求值会先对每一个参数进行求值,然后把参数的值代入函数体中.
所以,如果对(test 0 (p))使用应用序,那么(p)就会被求值,然后死循环.
如果是正则序,那么先展开,之后做 if 条件判断,然后求值 0,因为(p) 没有需要,所以没有求值,可以正确返回.

1.6
本题和 1.5 的关注点是相同的.如果使用 new-if 的话,它的三个参数都会被求值

(注意,Scheme 默认工作在应用序下).而它的第三个参数会对自己进行递归调用.所以会死循环.

1.7
原始的 good-enough?是检测两次的差的方式.对于很大的数,它的差可能根本无法精确到 good-enough?要求的精度.而对于很小的数,可能它发生了相对于自己来说很大的改变,但是却比 good-enough?要求的来得小.
用比率的方法写 good-enough?是可以的,但是要注意除以 0 这个问题.

1.8 略
1.2 节:
它 开始讲解递归和迭代之异同(大魔王有言"迭代者为人,递归者为神").实际上,迭代是递归的一种特殊情况,既 f(x) = f(x'),且 f(x')比 f(x)规模更小之时,递归可以化为迭代.递归的空间是线性增长的,而迭代始终为常数.这节主要掌握的是:函数的递归和迭代两 种写法,还有对程序的分析.

1.9
第一个是递归的.
第二个是迭代的.

1.10 略...

1.11

CODEBOX

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

(define (f2 n)
  (f2-iter 2 1 0 n))

(define (f2-iter a b c count)
  (if (= count 0) c
      (f2-iter (+ a (* 2 b) (* 3 c)) a b (- count 1))))
;f 是递归的,f2 是迭代的,其中用到了 f2-iter
```

1.12
CODEBOX

```
(define (combination x y)
  (cond ((= x y) 1)
        ((= y 0) 1)
```

```
        (else (+ (combination (- x 1) (- y 1))
                 (combination (- x 1) y)))))
```

1.13
先证明 Fib(n) = (a^n + b^n) / sqrt(5);
其中
a = (1 + sqrt(5)) / 2
b = (1 - sqrt(5)) / 2
可以使用归纳法,当然也可以通过解特征根方程解出来...似乎还可以用幂级数?(等等我去复习下离散数学)
然后发现|b / sqrt(5)| < 0.5,所以命题得证.

1.14
空间线性增长
步数是指数级的.因为其本质是一个线性递归方程.

1.15
SICP 这种细节题也很有意思.
a) 5 次.
我是用这个程序算的...
CODEBOX

```
(define (f x)
  (if (> (abs x) 0.1) (+ (f (/ x 3.0)) 1)
      0))
```

或者再偷懒一点的话每个 sine 前面加一行 display,这样最清楚...
啊,你说算么?嗯,注意现在用的 Scheme 是应用序的,注意每次调用的传值性,所以只要算 12.15 除以 3 多少次以后< 0.1 就好.

b)时间和空间都是对数级别增长.(关于角度 a)

1.16
CODEBOX

```
(define (fast-expt x n)
  (iter 1 x n))

(define (iter a b n)
  (cond ((= n 0) a)
        ((odd? n) (iter (* a b) b (- n 1)))
        (else (iter a (* b b) (/ n 2)))))
```

题目给了充分的提示,然而我还是想了一段时间...实际上就是倍增的思想.书中

关于"不变量"的那一句很棒.

1.17 同 1.16 的方法.

1.18 同 1.17 的方法.

1.19
个人觉得按照那样写变换很不方便,于是直接写矩阵就好.
观察发现
向量
(a b) * (0 1) = (b , a + b)
(1 1)

所以实际上只要做该矩阵的 n 次方就行了.根据 A^n 乘积与顺序无关这条定理,可以用 fast-expt 的方法做矩阵乘法,然后复杂度就是对数级的.
题目中 p' = p^2 + q^2,q' = q^2 + 2pq.

1.20
再次提醒应用序和正则序的区别.因为应用序的先算再传值的性质,应用序为 4 次.
正则序...先记 remainder = r.
(gcd a b)
-> (if (= b 0) a (gcd b (r a b)))
-> (if (= b 0) a (if (= (r a b) 0) b (gcd (r a b) (r (r a b) b)))
-> (if (= b 0) a (if (= (r a b) 0) b (if (= (r (r a b) (r (r a b) b)) 0) (r a b) (gcd ....)到这里就看得很清楚了。
r 分成两部分,第一部分是在 if 判断时,第二部分是在返回结果时,返回结果时自然是 4 次.
在 if 判断时,假设某一层判断的是 ak,bk,那么它的下一层判断的就是 bk,(r ak bk),而记得,在正则序中,ak 和 bk 是完全分开判断的.所以判断次数 f(k) = f(k-1) + f(k-2) + 1,f(1) = 0;左右两式各+1 可以构造 fibonacci 数列,F(K) = fibo(k) - 1.然后求解.再加上 4 次,一共是 1 + 2 + 4 + 7 + 4 = 18 次.

第二章:构造数据抽象

现在到了数学抽象中最关键的一步:让我们忘记这些符号所表示的对象...(数学家)不应在这里停步,有许多操作可以应用于这些符号,而根本不考虑它们到底代表着什么东西
Hermann Weyl 思维的数学方式

第三章:模块化,对象和状态

即使在变化中,它也丝毫未变
赫拉克利特(Heraclitus)

变得越多,它就越是原来的样子
阿尔芬斯.卡尔(Alphonse Karr)

第四章 元语言抽象

用普通的话来说,这个咒语就是------阿巴拉卡达巴拉,芝麻开门,而且还有另外的东西---在一个故事里的咒语在另外一个故事里就不灵了.真正的魔力在于知道哪个咒语有用,在什么时候,用于做什么,其诀窍就在于学会有关的诀窍.
而这些咒语也是我们的字母表里拼出来的,这个字母表不过是几十个可以用笔画出来的弯弯曲线.这就是最关键的!而那些珍宝也是如此,如果我们能将它们拿到手的话!这就像是说,通向珍宝的钥匙就是珍宝!
John Barth 奇想

(Greald Jay Sussman)
If you have the name of a spirit,you have the power of it.
in the class of MIT 6.001 1988

4.55
CODEBOX

```
(supervisor ?x (Ben Bitdiddle))

(job ?x (accounting .?type))

(address ?x (Slumerville .?type))
```

4.56
CODEBOX

```
(and (supervisor ?x (Ben Bitdiddle))
     (address ?x ?add))

(and (salary (Ben Bitdiddle) ?amount-1)
     (salary ?x ?amount-2)
     (lisp-value > ?amount-1 ?amount-2))

(and (supervisor ?x ?y)
     (not (job ?y (computer .?type)))
     (job ?x .?type-2))
```

4.57
CODEBOX

```
(rule (take-place ?x ?y)
      (and (not (same ?x ?y))
           (or (and (job ?x ?job-1)
                    (job ?y ?job-1))
               (and (job ?x ?job-2)
                    (can-do-job ?job-2 ?job-3)
                    (job ?y ?job-3)))))

(take-place ?x (Cy D. Fect))

(and (take-place ?x ?y)
     (salary ?x ?x-amount)
     (salary ?y ?y-amount)
     (lisp-value > ?x-amount ?y-amount))
```

4.58
CODEBOX

```
(rule (big-man ?x)
   (and (job ?x (?department .?type-x))
      (not (and (job ?x (?department .?type-x))
                (supervisor ?x ?y)
                (job ?y (?department .?type-y))))))
```

关于这个答案,我多说几句.
根据后面章节的信息,这个答案是错误的.但是我依旧只能把它摆在这里,这说明我不会做.
如果一个人有多个 boss 的话似乎不太好办...
CODEBOX

```
 (rule (bigshot ?person ?division)
        (and
          (job ?person (?division . ?r))
          (or
            (not (supervisor ?person ?boss))
            (and
              (supervisor ?person ?boss)
              (not (job ?boss (?division . ?q)))))))
```

这是另外一个人提供的解答,但是我觉得会有同样的问题...

哦,插播一句话,在写这里的程序中尽量避免用 same.比如说
CODEBOX

```
(and (job ?x (?department-x .?type-x))
     (job ?y (?department-y .?type-y))
     (same ?department-x ?department-y))
```

比起
CODEBOX

```
(and (job ?x (?department-x .?type-x))
     (job ?y (?department-x .?type-y)))
```

效果一样,但是效率上是相当糟糕的(为什么?).
而且,在后面会看到,改变 and 中条件的顺序可能会导致效率的不同*

4.59
CODEBOX

```
(meeting ?x (Friday ?t))

(rule (meeting-time ?person ?day-and-timne)
     (and (meeting ?department ?day-and-time)
          (job ?persone (?department .?type))))

(meeting-tinme (Hacker Alyssa P) (Wednesday ?t))
```

4.60
我觉得做不到.因为询问时的搜索是独立的,当我们搜索到A,B的时候B,A是否已经搜索过是不可知的.除非A,B这两个元素在搜索时能保证有一个确定的顺序.

4.61
CODEBOX

```
(1 next-to (2 3) in (1 (2 3) 4))

((2 3) next-to 4 in (1 (2 3) 4))
```

//这里说明一下,我觉得 2 next-to 3 不会出现.因为规则定义时并没有对里面的元素做递归展开.

```
(2 next-to 1 in (2 1 3 1))

(3 next-to 1 in (2 1 3 1))
```

4.62
CODEBOX

```
(rule (last-pair (?x nil) ?x))

(rule (last-pair (?u . ?v) ?x)
      (last-pair ?v ?x))
```

对于询问(last-pair ?x (3)) 这个程序会翘辫子的.因为这种?x 有无数个.在产生框架的时候已经死循环了.
倘若我们使用 Lazy 的方式进行求值,那么它是可以"生成"的,但是要全部都打印出来也太勉强人家了吧....

4.63
CODEBOX

```
(rule (grandson ?x ?y)
      (and (son ?x ?t)
           (son ?t ?y)))


(rule (son ?x ?y)
      (and (son ?x ?t)
           (wife ?y ?t)))
```

4.64
这个...我们不能用带递归的规则作为 and 的第一项.因为 and 的第一项是去生成新的框架流,在此例中,如果递归的话,框架流会无限地生成下去.

4.65
其实很简单,对于询问

CODEBOX

```
(and (supervisor ?middle-manager ?person)
      (supervisor ?x ?middle-manager))
```

Oliver 会出现 4 次啊 4 次.

4.66
根据 4.65 知道,同一个项目会被统计多次.
我的想法是在映射函数提取时用一个 Hash 表来判重.

4.67
我的想法是,加入当前这次询问的框架和正在使用的规则,因为死循环是由"规则"引起的,而如果落入简单死循环,也必然会出现在相同的规则时,该层的框架相同.

4.68
稍后放出...

4.69
CODEBOX

```
(rule (if-grandson ?x)
      (append-to-form ?x1 (grandson) ?x))

(rule ((great . ?mul) ?x1 ?x2)
      (and (if-grandson ?mul)
           (son ?x1 ?x3)
           (?mul ?x3 ?x2)))
```

这两个规则需要前面题的 grandson 规则才能作用.

第五章:寄存器机器里的计算

我的目的是想说明,这一天空机器并不是一种天赐造物或者生命体,它只不过是钟表一类的机械装置(而那些相信钟表有灵魂的人却将这一个工作归为其创造者的荣耀),在很大程度上,这里多种多样的运动都是由最简单的物质力量产生的,就像钟表里所有活动都是由一个发条产生的一样.
约翰尼斯.开普勒 给 Herwart von Hohenburg 的信 , 1605