安装环境及前言：

本来用的是 racket，结果做练习题的时候发现输出对不上，就该为了官方的 MIT-Scheme

安装过程如下：

```
cd ./mit-scheme/src/
./configure
make compile-microcode
make install
```

课程的代码部分在书籍的官网上有写，链接：https://mitpress.mit.edu/sites/default/files/sicp/code/index.html

B 站看学习视频比较舒服：https://www.bilibili.com/video/av8515129/

笔记：

note 1：

通常来说，在 Lisp 中键入了一个定义，它返回被定义的符号

如：(DEFINE A (* 5 5)) -- 返回 A

note 2：

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

这个句式的意思是定义一个过程，名为 square，意为定义 x 乘以它本身

square 是一种过程：compound procedure square（复合过程 square）

注：(define (square x) (* x x)) 等同于 (define square (lambda (x) (* x x)))

两种语句在计算机内无任何差别，只是 (define (square x) (* x x)) 的写法更便捷

还有一点就是之所以这个便捷形式在进行 define 时，spuare 的前面使用了左括号

是因为这是在定义一个过程，是特殊的写法，使用括号和不使用的区别如下：

使用：

输入：

```
(define (D) (* 5 5))
D
(D)
```

输出：

```
;Value: d
;Value: #[compound-procedure 13 d]
;Value: 25
```

不使用：

输入：

```
(define A (* 5 5))
A
(A)
```

输出（带有报错）：

```
;Value: a
;Value: 25
;The object 25 is not applicable.
;To continue, call RESTART with an option number:
; (RESTART 2) => Specify a procedure to use in its place.
; (RESTART 1) => Return to read-eval-print level 1.
```

note 3：

关于条件语句

假如我要定义 abs 函数，那么小于 0 的时候就要加个负号把它变为正数

表达方式 1：

```
(define (abs x)
  (cond ((> x 0) x)
        ((= x 0) 0)
        ((< x 0) (- x))))
```

cond 后面括号内，以逗号分隔，左边是条件，右边是返回值

以上可以当作是一堆 else if

```
(define (abs x)
  (cond ((< x 0) (- x))
        (else x)))
```

这个代码说明的是如果有不满足给出的所有条件的可以写 else 关键字来控制程序流程

表达方式 2：

```
(define (abs x)
  (if (< x 0)
      (- x)
      x))
```

这个 if 语句的意思是小于零就返回 x 的相反数，不然就返回本身

但是必须加上最后所条件都不符合会咋样

注意：

视频里说 if 和 cond 是等价的语法糖，但是它们编写代码时用的形式是不一样的

例如上述三个代码里，就不能把 if 和 cond 关键字置换，否则会报错

我还是偏爱用 cond

note 4：

主要记的是 Square Roots by Newton's Method 这一算法

通过几个过程分别进行

good-enough? 过程：

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))
```

这个过程用以确认最后求出来的值 guess

当它作平方后再减去 x 的值

之后所得的范围是否小于某个自己拟定的精度，这里的精度是 0.001

如果小于，就返回 #t，如果不小于，就返回 #f

improve 过程：

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

返回值为 x 除以 guess（原值除以猜测值）后再加上 guess 的和除以 2

即做完除法后的值与 guess 的平均数

average 过程：

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

这个过程没啥说的

sqrt-iter 过程（核心过程）：

步骤总结下来是这样的，首先我们看这段代码

```
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x)
                 x)))
```

这段代码里定义了 **sqrt-iter 过程**，过程内还带有两个其他过程 good-enough? 和 improve

比较令我神奇的是 scheme 定义过程时，过程名可以带?这一符号

这里的 guess 和 x 是变量名，guess 代表猜测的开方值，x 代表原值

过程步骤如下：

首先判断给的 guess 的值，在 **good-enough? 过程**里会返回 #t 还是 #f

返回 #t 就说明当前 guess 的值符合作为 x 的开方，就把他当作最后的返回值

返回 #f 说明 guess 不能作为最终的解，那么就进入 **sqrt-iter 过程**自身

只不过这回的参数要进行修改，第二个参数一直都会是 x，要改的是第一个参数

假设第一个参数为 guess，在经过 **improve 过程**后会变成如下的值：

guess = (guess + (x / guess)) / 2

这个式子本就是验证一个值是否为一个数的开方的，也可以在这里用于**逼近开方数**

sqrt 过程：

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

用于设定一个完整的 sqrt 函数，指定猜测值的初始值为 0.1，作用就是简化函数

习题

Exercise 1.1

Below is a sequence of expressions. What is the result printed by the interpreter in response to each expression? Assume that the sequence is to be evaluated in the order in which it is presented.

```
10
(+ 5 3 4)
(- 9 1)
(/ 6 2)
(+ (* 2 4) (- 4 6))
(define a 3)
(define b (+ a 1))
(+ a b (* a b))
(= a b)
(if (and (> b a) (< b (* a b)))
    b
    a)
(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
(+ 2 (if (> b a) b a))
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))
```

Solution:

```
10
12
8
3
6
a
b
19
#f
4
16
6
16
```

(= a b) 的意思是判断两数是否相等，返回的值是 #t 或者是 #f

分别代表 true 和 false

Exercise 1.2

Translate the following expression into prefix form

$$\frac{5+4+(2-(3-(6+\frac{1}{5})))}{3(6-2)(2-7)}$$

Solution:

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

Exercise 1.3

Define a procedure that takes three numbers as arguments and returns the sum of the squares of the two larger numbers.

Solution:

这题我一开始还以为都要写一起。。看了答案才知道分着写

```
(define (square x) (* x x))
(define (res x y) (+ (square x) (square y)))
(define (check x y z) (cond ((and (> x z) (> y z)) (res x y))
                            ((and (> x y) (> z y)) (res x z))
                            ((and (> y x) (> z x)) (res y z))))
```

Exercise 1.4

Observe that our model of evaluation allows for combinations whose operators are compound expressions. Use this observation to describe the behavior of the following procedure:

```
(define (a-plus-abs-b a b)
  ((if (> b 0) + -) a b))
```

Solution:

这代码给我看傻了，还能用 if 语句传递加减号的

反正意思就是 a + |b|

Exercise 1.5

Ben Bitdiddle has invented a test to determine whether the interpreter he is faced with is using applicative-order evaluation or normal-order evaluation. He defines the following two procedures:

```
(define (p) (p))

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

Then he evaluates the expression

```
(test 0 (p))
```

What behavior will Ben observe with an interpreter that uses applicative-order evaluation? What behavior will he observe with an interpreter that uses normal-order evaluation? Explain your answer. (Assume that the evaluation rule for the special form `if` is the same whether the interpreter is using normal or applicative order: The predicate expression is evaluated first, and the result determines whether to evaluate the consequent or the alternative expression.)

Solution:

这题给的知识点一个是求值顺序，一般分为**正则序**和**应用序**

正则序：完全展开而后归约。这种求值模型是先不求出运算对象的值，直到实际需要它们的值时再去做。首先用运算对象表达式去替换形式参数，直到得到一个只包含基本运算符的表达式，然后在去执行求值。

应用序：先求值参数而后应用。解释器将先对组合式的各个元素求值，即将过程体中的每一个形参用实参代替，然后在对这一过程体求值。

lisp 和 scheme 采用应用序求值

在调用题目给的函数的时候，因为 (p) 不断地调用自身，导致程序进入死循环

Exercise 1.6

Alyssa P. Hacker doesn't see why if needs to be provided as a special form. ``Why can't I just define it as an ordinary procedure in terms of cond?'' she asks. Alyssa's friend Eva Lu Ator claims this can indeed be done, and she defines a new version of if:

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

Eva demonstrates the program for Alyssa:

```
(new-if (= 2 3) 0 5)
5

(new-if (= 1 1) 0 5)
0
```

Delighted, Alyssa uses new-if to rewrite the square-root program:

```
(define (sqrt-iter guess x)
  (new-if (good-enough? guess x)
          guess
          (sqrt-iter (improve guess x)
                     x)))
```

What happens when Alyssa attempts to use this to compute square roots? Explain.

Solution:

本来 note 6 的内容没打算看，结果后面的题都需要用到

这道题的知识和 **Exercise 1.5** 是相连的，考察的是正则序和应用序

因为 scheme 在执行过程的时候用的是应用序，需要先扩展所有函数，最后再填值运行

在这个示例中就会导致死循环，造成内存溢出

如果是正常的 if 语句，那么就不会像这样，因为 if 关键字不采用应用序求值

Exercise 1.7

The `good-enough?` test used in computing square roots will not be very effective for finding the square roots of very small numbers. Also, in real computers, arithmetic operations are almost always performed with limited precision. This makes our test inadequate for very large numbers. Explain these statements, with examples showing how the test fails for small and large numbers. An alternative strategy for implementing `good-enough?` is to watch how `guess` changes from one iteration to the next and to stop when the change is a very small fraction of the guess. Design a square-root procedure that uses this kind of end test. Does this work better for small and large numbers?

Solution:

这题摸了，直接看答案揣摩意思了

首先是题目第一个问题，当前算法在应对大数和小数的弊端

下面是官方译文↓↓↓

当计算一个小值的平方根时，0.001 的绝对容差非常大。例如，在我使用的系统上，(sqrt 0.0001)的结果是0.03230844833048122，而不是预期的0.01(超过200%的错误)。

另一方面，对于非常大的根值，机器精度无法表示大数字之间的小差异。该算法可能永远不会终止，因为最佳猜测的平方不会在极值的0.001以内，并且试图改进它将会不断产生相同的猜测[即。(改进猜测x)将等于猜测]。尝试(sqrt 1000000000000)[有12个零]，然后尝试(sqrt 10000000000000)[13个零]。在我的64位intel机器上，12个0几乎可以立即得到答案，而13个0则进入一个无穷循环。

收↑↑↑

贴链接吧，代码优化这摸了：http://community.schemewiki.org/?sicp-ex-1.7

Exercise 1.8

Newton's method for cube roots is based on the fact that if *y* is an approximation to the cube root of *x*, then a better approximation is given by the value

$$\frac{x/y^2 + 2y}{3}$$

Use this formula to implement a cube-root procedure analogous to the square-root procedure. (In section 1.3.4 we will see how to implement Newton's method in general as an abstraction of these square-root and cube-root procedures.)

Solution:

这题就很~~舒服~~，应该还是要懂原理才会做，卡在 **good-enough? 过程**好久

没仔细搜，反正知道最后的判断条件是算上面这个公式，再和 y 相比，相等就满足条件了

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

(define (good-enough? guess x)
  (= (improve guess x) guess))

(define (average x y)
  (/ (+ x y) 3))

(define (improve guess x)
  (average (/ x (square guess)) (* 2 guess)))

(define (cbrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (cbrt-iter (improve guess x)
                 x)))

(define (cbrt x)
  (cbrt-iter 1.1 x))
```

注意这里不用 1.1 而是用 1.0 的话，测试 -2 就能看见错误

```
> (cbrt -2)
-inf.0
```

参考链接：

https://blog.csdn.net/jiangxuege/article/details/83506597