SICP Chapter 1

ProducerMatt

HOW THIS DOCUMENT IS MADE

TODO

```
(define (foo a b)
  (+ a (* 2 b)))

(display
 (foo 5 3))
```

```
11
```

^ Dynamically evaluated when you press "enter" on the `BEGIN_SRC` block!

Also consider:

- `:results output` for what the code prints
- `:exports code` or `:exports results` to just get one or the other

$a + (\pi \times b)$ <~ inline Latex btw :)

Helper for org-mode tables

```
(define (transpose-list l)
  (map (λ (i) (list i)) l))
```

```
(define (p-nl a)
  (display a)
  (newline))
(define (print-spaced args)
  (let ((a (car args))
        (d (cdr args)))
    (if (null? d)
        (p-nl a)
        (begin (display a)
               (display " ")
               (print-spaced d)))))
(define (print-as-rows . args)
```

```
(let ((a (car args))
      (d (cdr args)))
  (if (list? a)
      (if (= 1 (length args))
          (apply print-as-rows a)
          (print-spaced a))
      (p-nl a))
  (if (null? d)
      '()
      (apply print-as-rows d))))
```

Exercise 1.1

Q

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.

A

```
10 ;; 10
(+ 5 3 4) ;; 12
(- 9 1) ;; 8
(/ 6 2) ;; 3
(+ (* 2 4) (- 4 6)) ;; 6
(define a 3) ;; a=3
(define b (+ a 1)) ;; b=4
(+ a b (* a b)) ;; 19
(= a b) ;; false
(if (and (> b a) (< b (* a b)))
    b
    a) ;; 4
(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25)) ;; 16
(+ 2 (if (> b a) b a)) ;; 6
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1)) ;; 16
```

Exercise 1.2

Q

Translate the following expression into prefix form:

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

A

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

```
1/75
```

Exercise 1.3

Text

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

Q

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

A

```
<<square>>
(define (sum-square x y)
  (+ (square x) (square y)))
(define (square-2of3 a b c)
  (cond ((and (>= a b) (>= b c)) (sum-square a b))
        ((and (>= a b) (> c b)) (sum-square a c))
        ((and (> b a) (>= c a)) (sum-square b c))
        (else "This shouldn't happen")))

(display
 (list (square-2of3 7 5 3)
       (square-2of3 7 3 5)
       (square-2of3 3 5 7)))

```

74	74	74

Exercise 1.4

Q

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))
```

A

This code accepts the variables `a` and `b`, and if `b` is positive, it adds `a` and `b`. However, if `b` is zero or negative, it subtracts them. This decision is made by using the `+` and `-` procedures as the results of an if expression, and then evaluating according to the results of that expression. This is in contrast to a language like Python, which would do something like this:

```
if b > 0: a + b
else: a - b
```

Exercise 1.5

Q

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.)

A

In either type of language, `(define (p) (p))` is an infinite loop. However, a normal-order language will encounter the special form, return `0`, and never evaluate `(p)`. An applicative-order language evaluates the arguments to `(test 0 (p))`, thus triggering the infinite loop.

Exercise 1.6

Text code

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

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

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

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

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

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

Q

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.

A

Using Alyssa's `new-if` leads to an infinite loop because the recursive call to `sqrt-iter` is evaluated before the actual call to `new-if`. This is because `if` and `cond` are special forms that change the way evaluation is handled; whichever branch is chosen leaves the other branches unevaluated.

Exercise 1.7

Text

```
(define (mean-square x y)
  (average (square x) (square y)))
```

Q

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?

A

The current method has decreasing accuracy with smaller numbers. Notice the steady divergence from correct answers here (should alternate between values starting with "316" and "100"):

```
<<transpose-list>>
<<txt-sqrt>>
(transpose-list (map sqrt '(0.1 0.01 0.001 0.0001 0.00001)))
```

0.316245562280389
0.10032578510960605
0.04124542607499115
0.03230844833048122
0.03135649010771716

And for larger numbers, an infinite loop will eventually be reached. 10^{12} can resolve, but 10^{13} cannot.

```
<<txt-sqrt>>
(sqrt 1000000000000)
```

```
1000000.0
```

My original answer was this, which compares the previous iteration until the new and old are within an arbitrary dx.

```
<<txt-sqrt>>
(define (inferior-good-enough? guess lastguess)
  (<=
   (abs (-
         (/ lastguess guess)
         1))
   0.0000000000001)) ; dx
(define (new-sqrt-iter guess x lastguess) ;; Memory of previous value
  (if (inferior-good-enough? guess lastguess)
      guess
      (new-sqrt-iter (improve guess x) x guess)))
(define (new-sqrt x)
  (new-sqrt-iter 1.0 x 0))
```

This solution can correctly find small and large numbers:

```
<<inferior-good-enough>>
(display (new-sqrt 10000000000000))
```

```
3162277.6601683795
```

```
<<transpose-list>>
<<inferior-good-enough>>
(transpose-list (map new-sqrt '(0.1 0.01 0.001 0.0001 0.00001)))
```

0.31622776601683794
0.1
0.03162277660168379
0.01
0.003162277660168794

However, I found this solution online that isn't just simpler but automatically reaches the precision limit of the system:

```
<<txt-sqrt>>
(define (good-enough? guess x)
   (= (improve guess x) guess))
```

```
<<transpose-list>>
<<new-good-enough>>
(define (sqrt-iter guess x)
  (if (good-enough? guess x)
      guess
      (sqrt-iter (improve guess x) x)))

(define (sqrt x)
  (sqrt-iter 1.0 x))
(transpose-list (map sqrt '(0.1 0.01 0.001 0.0001 0.00001)))
```

0.31622776601683794
0.1
0.03162277660168379
0.01
0.003162277660168794