SICP Chapter 1

ProducerMatt

Testing

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

(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 `results` to just get one or the other

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

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

```
(/ (+ 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")))

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

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

(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.