Contents

Exercise 1.1

In below expressions, the result printed by the interpreter is given. It's assumed that the sequence is to be evaluated in the order they are presented.

```scheme
10
```

10

```scheme
(+ 5 3 4)
```

12

```scheme
(- 9 1)
```

8

```scheme
(/ 6 2)
```

3

```scheme
(+ (* 2 4) (- 4 6))
```

6

```scheme
(define a 3)
```

implementation dependent a = 3

```scheme
(define b (+ a 1))
```

implementation dependent b = 4

```scheme
(+ a b (* a b))
```

19

```scheme
(= a b)
```

false

```scheme
(if (and (> b a) (< b (* a b)))
    b
    a)
```

4

```scheme
(cond ((= a 4) 6)
      ((= b 4) (+ 6 7 a))
      (else 25))
```

16

```scheme
(+ 2 (if (> b a) b a))
```

6

```
(* (cond ((> a b) a)
         ((< a b) b)
         (else -1))
   (+ a 1))
```

```
16
```

Exercise 1.2

Translation of the following expression into prefix notation

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

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

Exercise 1.3

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

```
(define (sum-squares-two-larger-numbers a b c)
  (cond ((and (< b a)
              (< b c)) (+ (* a a)
                          (* c c)))
        ((and (< a b)
              (< a c)) (+ (* b b)
                          (* c c)))
        (else (+ (* a a)
                 (* b b)))))
```

Exercise 1.4

Description of the behavior of the following procedure with the observation that our model of evaluation allows for combinations whose operators are compound expressions.

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

The sub-expression `(if (> b 0) + -)` will evaluate to `+` if $b > 0$ and `-` otherwise. So, the body of the procedure will become

- `(+ a b)` if $b > 0$
- `(- a b)` if $b \leq 0$

In sum, the procedure `a-plus-abs-b` is computing $a + |b|$.

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))`.

Let's devise the behavior that Ben will observe with an interpreter that uses

- Applicative-order evaluation

The interpreter will evaluate the arguments at first and `(p)` will call itself and will be an infinite loop that never ends.

- Normal-order evaluation

The interpreter will first expand the expression `(test 0 (p))` into

```
(if (= 0 0)
  0
  (p))
```

As `(= 0 0)` evaluates to `true`, the consequent `0` will be evaluated and the procedure evaluation will terminate with the result `0`.

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 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)))
```

When Alyssa attempts to use `new-if` to compute square roots, it will end into a never ending evaluation. In fact, according to the substitution model for procedure evaluation, the operands of `new-if` will be evaluated before applying the operation on them. It will then evaluate `sqrt-iter` call even if the guess is already good enough. The later would call another iteration, and so on…