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.

```
10
```

`10`

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

`12`

```
(- 9 1)
```

`8`

```
(/ 6 2)
```

`3`

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

`6`

```
(define a 3)
```

implementation dependent a = 3

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

implementation dependent 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

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…

Exercise 1.7

The `good-enough?` test used in computing square roots will not be very effective for finding square roots of very small numbers because the margin of error could be higher than the number itself.

e.g.

```
(square (sqrt 0.0000001))

;Value: 9.765291022445155e-4
```

For very large numbers, the procedure evaluation can take very long.

e.g.

```
(sqrt 1e100)
```

The evaluation of this expression causes the interpreter to become stuck.

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 very small fraction of the guess.

```
(define (good-enough? guess x)
  (< (abs (- guess (improve guess x))) (* 0.0001 guess)))
```

Using this new definition of `good-enough?` the previous expressions are respectively more accurate and terminable within a reasonable time.

```
 (square (sqrt 0.0000001))

;Value: 1.0001264334838593e-7
```

```
(sqrt 1e100)

;Value: 1.0000006331051179e50
```

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

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

Implementation of a cube-root procedure analogous to the square-root procedure defined earlier.

```
(define (improve guess x)
  (/ (+ (/ x (square guess)) (* 2 guess)) 3))

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

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

Exercise 1.9

Illustration of the process generated by each of the following procedure in evaluating `(+ 4 5)` which are defined in terms of `inc`, which increments its argument by 1, and `dec`, which decrements its argument by 1.

```
(define (+ a b)
  (if (= a 0)
    b
    (inc (+ (dec a) b))))

(+ 4 5)
(inc (+ (dec 4) 5))
(inc (+ 3 5))
(inc (inc (+ (dec 3) 5)))
```

```
(inc (inc (+ 2 5)))
(inc (inc (inc (+ (dec 2) 5))))
(inc (inc (inc (+ 1 5))))
(inc (inc (inc (inc (+ (dec 1) 5)))))
(inc (inc (inc (inc (+ 0 5)))))
(inc (inc (inc (inc 5))))
(inc (inc (inc 6)))
(inc (inc 7))
(inc 8)
9
```

This process is recursive as the interpreter has to keep track of the deferred increment processes.

```
(define (+ a b)
    (if (= a 0)
        b
        (+ (dec a) (inc b))))
```

```
(+ 4 5)
(+ (dec 4) (inc 5))
(+ 3 6)
(+ (dec 3) (inc 6))
(+ 2 7)
(+ (dec 2) (inc 7))
(+ 1 8)
(+ (dec 1) (inc 8))
(+ 0 9)
9
```

This process is iterative as the interpreter has only to keep track of a finite number of variable state at each stage of the computation.

Exercise 1.10

The following procedure computes a mathematical function called Ackermann's function

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= x 0) (* 2 y))
        ((= y 1) 2)
        (else (A (- x 1)
                 (A x (- y 1))))))
```

Let's compute the value of the following expressions.

```
(A 1 10)


(A (- 1 1)
   (A 1 (- 10 1)))


(A 0 (A 1 9))
(* 2 (A 1 9))


(* 2 (A (- 1 1)
        (A 1 (- 9 1))))


(* 2 (A 0
        (A 1 8)))
```

```
(* 2 (* 2 (A 1 8)))


(* 2 (* 2 (A (- 1 1)
             (A 1 (- 8 1)))))


(* 2 (* 2 (A 0
             (A 1 7))))


(* 2 (* 2 (* 2 (A 1 7))))


(* 2 (* 2 (* 2 (A (- 1 1)
                  (A 1 (- 7 1))))))


(* 2 (* 2 (* 2 (A 0
                  (A 1 6)))))


(* 2 (* 2 (* 2 (* 2 (A 1 6)))))


(* 2 (* 2 (* 2 (* 2 (A (- 1 1)
                       (A 1 (- 6 1)))))))


(* 2 (* 2 (* 2 (* 2 (A 0
                       (A 1 5))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (A 1 5))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (A (- 1 1)
                            (A 1 (- 5 1))))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (A 0
                            (A 1 4)))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A 1 4)))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A (- 1 1)
                                 (A 1 (- 4 1)))))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A 0
                                 (A 1 3))))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A 1 3))))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A (- 1 1)
                                      (A 1 (- 3 1))))))))))
```

```
(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A  0
                                       (A 1 2)))))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A 1 2)))))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A (- 1 1)
                                           (A 1 (- 2 1)))))))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A 0
                                           (A 1 1))))))))))


(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (A 1 1))))))))))
(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 2)))))))))
(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 4))))))))
(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 (* 2 8)))))))
(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 16))))))
(* 2 (* 2 (* 2 (* 2 (* 2 (* 2 16))))))
(* 2 (* 2 (* 2 (* 2 (* 2 32)))))
(* 2 (* 2 (* 2 (* 2 64))))
(* 2 (* 2 (* 2 128)))
(* 2 (* 2 256))
(* 2 512)
1024
```

```
(A 2 4)


(A (- 2 1)
   (A 2 (- 4 1)))


(A 1
   (A 2 3))


(A 1
   (A (- 2 1)
      (A 2 (- 3 1))))


(A 1
   (A 1
      (A 2 2)))


(A 1
   (A 1
      (A (- 2 1)
         (A 2 (- 2 1)))))


(A 1
   (A 1
      (A 1
         (A 2 1))))
```

```
(A 1
   (A 1
      (A 1
         (A 2 1))))


(A 1
   (A 1
      (A 1 2)))


(A 1
   (A 1
      (A (- 1 1)
         (A 1 (- 2 1)))))


(A 1
   (A 1
      (A 0
         (A 1 1))))


(A 1
   (A 1
      (A 0 2)))


(A 1
   (A 1
      (* 2 2)))


(A 1
   (A 1 4))


(A 1
   (A (- 1 1)
      (A 1 (- 4 1))))


(A 1
   (A 0
      (A 1 3)))


(A 1 (* 2 (A 1 3)))

(A 1 (* 2 (A (- 1 1)
             (A 1 (- 3 1)))))


(A 1 (* 2 (A 0
             (A 1 2))))


(A 1 (* 2 (* 2 (A 1 2))))
```

```
(A 1 (* 2 (* 2 (A (- 1 1)
                  (A 1 (- 2 1))))))


(A 1 (* 2 (* 2 (A 0
                  (A 1 1)))))


(A 1 (* 2 (* 2 (* 2 (A 1 1)))))
(A 1 (* 2 (* 2 (* 2 2))))
(A 1 (* 2 (* 2 (* 2 2))))
(A 1 (* 2 (* 2 4)))
(A 1 (* 2 8))
(A 1 16)

2^16
```

```
(A 3 3)

(A (- 3 1)
   (A 3 (- 3 1)))

(A 2
   (A 3 2))

(A 2
   (A (- 3 1)
      (A 3 (- 2 1))))

(A 2
   (A 2
      (A 3 1)))

(A 2
   (A 2 2))

(A 2
   (A (- 2 1)
      (A 2 (- 2 1))))

(A 2
   (A 1
      (A 2 1)))

(A 2
   (A 1 2))

(A 2
   (A (- 1 1)
      (A 1 (- 2 1))))

(A 2
   (A 0
      (A 1 1)))

(A 2
   (A 0 2))

(A 2
   (* 2 2))
```

```
(A 2 4)
```

```
2^16
```

Let's give a concise mathematical definitions for the functions computed by the following procedures:

```
(define (f n) (A 0 n))
```

computes $2n$

```
(define (g n) (A 1 n))
```

computes 2^n

```
(define (h n) (A 2 n))
```

computes 2_n

Example from section 1.2.2 Tree recursion: Counting change

Let's walk through the detail of how the reduction rule applies to the problem of making change for 10 cents using pennies and nickels.

Let's note a = 10 four the amount in pennies and c = {1, 5} for the set of all kinds of coins.

(1)
- using all but first kind of coin a = 10 c = {5} (2)
- using all n kind of coins a = 9 c = {1, 5} (3)

(2)
- using all but first kind of coin a = 10 c = {} => 0
- using all kinds of coin a = 5 c = {5} (4)

(3)
- using all but first kind of coin a = 9 c = {5} (5)
- using all kind of coin a = 8 c = {1, 5} (6)

(4)
- using all but first kind of coin a = 5 c = {} => 0
- using all kind of coin a = 0 c = {5} => 1

(5)
- using all but first kind of coin a = 9 c = {} => 0
- using all kind of coin a = 4 c = {5} (7)

(6)
- using all but first kind of coin a = 8 c = {5} (8)
- using all kind of coin a = 7 c = {1, 5} (9)

(7)
- using all but first kind of coin a = 4 c = {} => 0
- using all kind of coin a = -1 c = {5} => 0

(8)
- using all but first kind of coin a = 8 c = {} => 0
- using all kind of coin a = 3 c = {5} (10)

(9)
- using all but first kind of coin a = 7 c = {5} (11)
- using all kind of coin a = 6 c = {1, 5} (12)

(10)
- using all but first kind of coin a = 3 c = {} => 0
- using all kind of coin a = -2 c = {5} => 0

(11)
- using all but first kind of coin a = 7 c = {} => 0
- using all kind of coin a = 2 c = {5} (13)

(12)
- using all but first kind of coin a = 6 c = {5} (14)
- using all kind of coin a = 5 c = {1, 5} (15)

(13)
- using all but first kind of coin a = 2 c = {} => 0
- using all kind of coin a = -3 c = {5} => 0

(14)
- using all but first kind of coin a = 6 c = {} => 0
- using all kind of coin a = 1 c = {5} (16)

(15)
- using all but first kind of coin a = 5 c = {5} (17)
- using all kind of coin a = 4 c = {1, 5} (18)

(16)
- using all but first kind of coin a = 1 c = {} => 0
- using all kind of coin a = =4 c = {5} => 0

(17)
- using all but first kind of coin a = 5 c = {} => 0
- using all kind of coin a = 0 c = {5} => 1

(18)
- using all but first kind of coin a = 4 c = {} => 0
- using all kind of coin a = 3 c = {1, 5} (19)

(19)
- using all but first kind of coin a = 3 c = {5} (20)
- using all kind of coin a = 2 c = {1, 5} (21)

(20)
- using all but first kind of coin a = 3 c = {} => 0
- using all kind of coin a = -2 c = {5} => 0

(21)
- using all but first kind of coin a = 2 c = {5} (22)
- using all kind of coin a = 1 c = {1, 5} (23)

(22)
- using all but first kind of coin a = 2 c = {} => 0
- using all kind of coin a = -3 c = {5} => 0

(23)
- using all but first kind of coin a = 1 c = {} => 0
- using all kind of coin a = 0 c = {1, 5} => 1

=> There are 2 ways to make change for 10 cents using pennies and nickels.

This can be accomplished also intuitively by taking the 2 cases separately

1. all but the first kind of coin {5, 5}
2. all kind of coins: number of ways to change 9 cents. {5, 1, 1, 1, 1} and {1, 1, 1, 1, 1, 1, 1, 1, 1}

Exercise 1.11

A function f is defined by the rule that $f(n) = n$ if $n < 3$ and $f(n) = f(n-1) + 2f(n-2) + 3f(n-3)$ if $n \geq 3$.

Let's write a procedure that compute f by the means of a

- recursive process

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

- iterative process

```
(define (f n)
  (define (f-iter c b a k)
    (if (<= (+ k 3) n)
      (f-iter (+ c (* 2 b) (* 3 a)) c b (+ k 1))
      c))
  (if (< n 3)
    n
    (f-iter 2 1 0 0)))
```

Exercise 1.12

The following pattern of numbers is called Pascal's triangle.

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

The numbers at the edge of the triangle are all 1, and each number inside the triangle is the sum of the two numbers above it.

Let's write a procedure that computes elements of Pascal's triangle by means of a recursive process.

```scheme
(define (pascal x y)
  (if (or (= 1 y) (= x y))
    1
    (+ (pascal (- x 1) (- y 1))
       (pascal (- x 1) y))))
```