sicp-chapter-1

yuen

May 11, 2014

Contents

1 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))
```

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

```
6
16
```

2 Exercise 1.2.

Translate the following expression into prefix form

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

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

 -37/150
```

3 Exercise 1.3.

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

```
(define (bigger x y)
  (if (> x y) x y))

(define (smaller x y)
  (if (< x y) x y))

(define (bigger-sum a b c)
  (+ (sqr (bigger a b)) (sqr (bigger (smaller a b) c))))

(bigger-sum 1 2 3)

 13
```

4 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))

(a-plus-abs-b 1 1)
(a-plus-abs-b 1 -1)
```

```
 2
 2
```

a-plus-abs-b a b

```
(if (> b 0) + -)
```

b 0 + -
(op a b)

5 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.)

applicative-order: (test 0 (p))
0(p) (p)
normal-order: 0 (p) (test x y)
(if (= x 0) 0 (p)) (= x 0) : (= 0 0) 0 (p)

6 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)

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

 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.

new-if

new-if (goo-enough) (sqrt-iter) (sqrt-iter)

(if <predicate> <consequent> <alternative>) <predicate>

<predicate> <consequent>

<alternative>

```
(if #t (display "hello") (display "world"))

 hello
```

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

(new-if #t (display "hello") (display "world"))

 helloworld

   :

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

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

(define (good-enough? guess x)
  (display (format "~a\n" guess))
  (< (abs (- (square guess) x)) 0.0001))

(define (improve guess x)
  (/ (+ guess (/ x guess)) 2))

(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))

(sqrt 9)

(sqrt 0.000000000000000000009)

1.0
5.0
3.4
3.023529411764706
3.00009155413138
3.000000001396984
```

```
3.000000001396984

1.0
0.5
0.25
0.125
0.0625
0.03125
0.015625
0.0078125
0.0078125
```

7 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?

good-enough

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

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

(define (good-enough? old-guess new-guess)
  (display (format "~a\n" old-guess))
  (< (/ (abs (- new-guess old-guess)) old-guess) 0.0001))

(define (improve guess x)
  (/ (+ guess (/ x guess)) 2))

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

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

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

(sqrt 0.000000000000000000009)
```

```
1.0
0.5
0.25
0.125
0.0625
0.03125
0.015625
0.0078125
0.00390625
0.001953125
0.0009765625000000002
0.0004882812500000005
0.0002441406250000012
0.00012207031250000244
6.10351562500049e-05
3.0517578125009826e-05
1.5258789062519658e-05
7.629394531289321e-06
3.814697265703643e-06
1.9073486329697864e-06
9.536743167208228e-07
4.768371588322706e-07
2.384185803598537e-07
1.1920929206736365e-07
5.960464980855534e-08
2.9802332454024236e-08
1.4901181326501416e-08
7.450620862198706e-09
3.725370828750545e-09
1.8628062077192604e-09
9.316446748819594e-10
```

```
4.663053542092095e-10
2.341177099868953e-10
1.1898096502957087e-10
6.327260009878856e-11
3.8748383309721934e-11
3.098757940327563e-11
3.001573716141363e-11
3.000000412547338e-11
3.000000412547338e-11
```

8 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

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

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.)

'improve'

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

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

(define (good-enough? old-guess new-guess)
  (display (format "~a\n" old-guess))
  (< (/ (abs (- new-guess old-guess)) old-guess) 0.0001))

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

(define (cube-root-iter guess x)
  (let ((new-guess (improve guess x)))
    (if (good-enough? guess new-guess) guess
```

```
        (cube-root-iter new-guess x))))

(define (cube-root x)
  (cube-root-iter 1.0 x))

(cube-root 0.000000008)
```

```
1.0
0.6666666693333334
0.44444445222222223
0.296296314981481
0.19753090702931686
0.13168733969659827
0.08779171357094338
0.05852815503580874
0.03901954848882324
0.026014783802133092
0.017347129492363524
0.011573614622429779
0.007735651222160343
0.005201663871345835
0.003566332162406482
0.0025872193951923263
0.0021231972308747436
0.0020070101831742913
0.002000024457002051
0.002000024457002051
```

9 Exercise 1.9.

Each of the following two procedures defines a method for adding two positive integers in terms of the procedures 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))))

(define (+ a b)
```

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

Using the substitution model, illustrate the process generated by each procedure in evaluating (+ 4 5). Are these processes iterative or recursive?

```
(define (inc x)
  (display (format "inc ~a\n" x))
  (+ x 1))

(define (dec x)
  (display (format "dec ~a\n" x))
  (- x 1))

(define (plus a b)
  (display (format "plus ~a ~a\n" a b))

  (if (= a 0)
      b
      (inc (plus (dec a) b))))

(plus 4 5)
```

```
(define (inc x)
  (display (format "inc ~a\n" x))
  (+ x 1))

(define (dec x)
  (display (format "dec ~a\n" x))
  (- x 1))


(define (plus a b)
  (display (format "plus ~a ~a\n" a b))

  (if (= a 0)
      b
      (plus (dec a) (inc b))))

(plus 4 5)
```

10 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))))))
```

What are the values of the following expressions?

```
(A 1 10)

(A 2 4)

(A 3 3)
```

Consider the following procedures, where A is the procedure defined above:

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

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

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

(define (k n) (* 5 n n))
```

Give concise mathematical definitions for the functions computed by the procedures f, g, and h for positive integer values of n. For example, (k n) computes $5n^2$.

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

```
(A 1 10)

(A 2 4)

(A 3 3)

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

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

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

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

(display "(f n)\n")
(for ([i 10])
  (display (format "~a: ~a\n" i (f i))))


(display "(g n)\n")
(for ([i 10])
  (display (format "~a: ~a\n" i (g i))))

(display "(h n)\n")
(for ([i 5])
  (display (format "~a: ~a\n" i (h i))))
```

(f n) = 2n
(g n) = 2^n
(h n) = $2^{22} \ldots$ n}

11 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> 3.
Write a procedure that computes f by means of a recursive process.
Write a procedure that computes f by means of an iterative process.

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

(rec-f 0)
(rec-f 1)
(rec-f 2)
(rec-f 3)
(rec-f 4)

(define (iter-f-iter a b c i n)
  (display (format "a=~a, b=~a, c=~a, i=~a\n" a b c i))
  (if (= i n)
      c
      (iter-f-iter (+ a (* 2 b) (* 3 c))
                   a
                   b
                   (+ i 1)
                   n)))

(define (iter-f n)
  (iter-f-iter 2 1 0 0 n))

(iter-f 4)
```

12 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.35 Write a procedure that computes elements of Pascal's triangle by means of a recursive process.

```
(define (pascal row col)
  (cond ((> col row) (error "row less then cow"))
        ((or (= col 0) (= row col)) 1)
        (else (+ (pascal (- row 1) (- col 1)) (pascal (- row 1) col)))))

(pascal 1 1)

(pascal 3 2)
```

13 Exercise 1.13.

Prove that Fib(n) is the closest integer to ^{n} / 5, where = (1 + 5) / 2. Hint: Let = (1 - 5) / 2. Use induction and the definition of the Fibonacci numbers (see section 1.2.2) to prove that

Fib(n) = (^{n} - ^{n}) / 5.

Fib(n) = (((1 + 5) / 2)^{n} - ((1 - 5) / 2)^{n}) / 5.

```
(define (fib n)
  (cond ((< n 0) (error "n is less then 0"))
        ((= n 0) 0)
        ((= n 1) 1)
        ((= n 2) 1)
        (else (+ (fib (- n 1)) (fib (- n 2))))))

(displayln (fib 6))

(define (fib n)
  ( / (- (expt (/ (+ 1 (sqrt 5)) 2) n)
         (expt (/ (- 1 (sqrt 5)) 2) n))
     (sqrt 5)))

(displayln (fib 6))
```

Fib(1) = (((1 + 5) / 2) - ((1 - 5) / 2)) / 5 = 1

...

Fib(n) = (((1 + 5) / 2)^{n} - ((1 - 5) / 2)^{n}) / 5 = (^{n} - ^{n}) / 5

Fib(n + 1) = ($^{(}n + 1) -^{(} n + 1))/5 = (^{n} - ^{n}) / 5 = (((1 + 5) / 2)^{n} - ((1 - 5) / 2)^{n}) / 5 = (1/2)(Fib(n) + ^{n} + ^{n})

Fib(n + 2) = (3/2)(Fib(n) + ^{n} + ^{n}) = Fib(n) + Fib(n + 1)
Fib

14 Exercise 1.14.

Draw the tree illustrating the process generated by the count-change procedure of section 1.2.2 in making change for 11 cents. What are the orders of growth of the space and number of steps used by this process as the amount to be changed increases?

```
(require trace)

(define (first-denomination kinds-of-coins)
  (cond ((= kinds-of-coins 1) 1)
        ((= kinds-of-coins 2) 5)
        ((= kinds-of-coins 3) 10)
        ((= kinds-of-coins 4) 25)
        ((= kinds-of-coins 5) 50)))


(define (cc amount kinds-of-coins)
  (cond ((= amount 0 ) 1)
        ((or (< amount 0) (= kinds-of-coins 0)) 0)
        (else (+ (cc amount (- kinds-of-coins 1))
                 (cc (- amount (first-denomination kinds-of-coins))
                     kinds-of-coins)))))

(define (count-change amount)
  (cc amount 5))


(displayln (count-change 11))
```

racket=(require trace)=

15 Exercise 1.15.

The sine of an angle (specified in radians) can be computed by making use of the approximation sin x x if x is sufficiently small, and the trigonometric identity

$$\sin x = 3 \sin \frac{x}{3} - 4 \sin^3 \frac{x}{3} \tag{3}$$

to reduce the size of the argument of sin. (For purposes of this exercise an angle is considered "sufficiently small" if its magnitude is not greater than 0.1 radians.) These ideas are incorporated in the following procedures:

```
(define (cube x) (* x x x))
(define (p x) (- (* 3 x) (* 4 (cube x))))
(define (sine angle)
   (if (not (> (abs angle) 0.1))
       angle
       (p (sine (/ angle 3.0)))))
```

a. How many times is the procedure p applied when (sine 12.15) is evaluated?

b. What is the order of growth in space and number of steps (as a function of a) used by the process generated by the sine procedure when (sine a) is evaluated?