Programming Principles

Homework No. 5 [40 points]

Due: December 3, 2018 10:30AM

Write your report in English. Submit **your racket file** for the problems that are specified in the problem description. This is your midterm exam. You are requested to solve it again with Dr. Racket interpreter. When "run" button is pushed, your interaction window must show your testing results. If you need to write your solution on the paper as specified, write it as comments in the program at the proper position. In front of your solution, state which problems you are solving with the comment line, for example " : Problem No. 3–(a)". Submit your Racket file (in a single file) to TA (email: 204606131@qq.com), so that TA can test your program. Your file name must be "PP_hw5_*your_student_id_number.rkt*". Don't forget documenting your programs. Make the first comment be your student id number and name, so that TA can find who you are.

1. What will DrRacket print in response to the following expressions? If an expression produces an error message, you may just write "**error**"; you don't have to provide the exact text of the message. If the value of an expression is a procedure, just write "**procedure**"; you don't have to show the form in which DrRacket print the procedure.

```
A. ; Problem 1-A
   ((lambda (a + b) (+ b a)) 2 – 4)
   ; 2   ; this is a sample solution.

B. ((lambda (a)
       (lambda (b)
          (+ (sqrt a) (sqrt b))))
     5)

C. (define arg 7)
   (define local-arg 5)
   (define (proc arg)
      (let ((local-arg 3))
         (+ arg local-arg)))
   (proc 1)
```

```
D. (let ((a 3) (b 4))
        (lambda () (+ a b)))

E. ((lambda (+) (+ 3)) (lambda (*) (+ * 2) ))

F. (let ((x 5)(y x))  (* x x y ))

G. (let ((x 100)
         (f (lambda (y) (+ x y))))
       (f 25))

H. (define square (lambda (x) (* x x)))
   (define f
         (lambda (g)
               (lambda (f)
                  (f g))))
   ((f 5) square)

I. (define (square x) (* x x))
   (define (cube x) (* x x x))
   (define bop-funs
            (lambda (bop)
               (lambda (f g) (lambda (x) (bop (f x) (g x))))))
   (((bop-funs +) square cube) 2)

J. (define x 6)
    (define (f y)
       (define x 2)
       (define z (* x 4))
       (+ x y z))
   (f (+ x 1))
```

2. Consider the following program:

```
; (member? digit num): decimal-digit, integer -> boolean
; whether num(>0) has a digit in decimal representation
```

```
; (member? 5 10) => #f
; (member? 5 253) => #t
 (define (member? digit num)
     (cond ((= num 0) #f)
           ((= (remainder num 10) digit) #t)
           (else (member? digit (quotient num 10)))))
```

The primitive functions `remainder` and `quotient` give the results as in the following examples:

`(remainder 286 10)` ➔ `6` `(quotient 286 10)` ➔ `28`

`(remainder 3157 100)` ➔ `57` `(quotient 3157 100)` ➔ `31`

(a) Assuming that all the primitive procedures used take constant time, indicate the order of growth in time of **`(member? digit n)`** with respect to **`n`**. Which is the most reasonable approximation? Choose one and explain why? If you DO NOT give proper reasoning, you CANNNOT get the point.

; $\Theta(1)$_____ $\Theta(\log n)$ _____ $\Theta(n)$ _____ $\Theta(n^2)$_____

; Your comment why you choose the solution.

(b) What kind of process does **`(member? digit n)`** generate? Recursive or Iterative? Explain why.

; Recursive Process _____ Iterative Process _____

; Your comment why you choose the solution.

3. Consider the following program:

```
(define (even? n)
   (cond ((= n 0) #t)
         ((= n 1) #f)
         (else (if (even? (- n 2))
                   #t
                   #f))))
```

(a) Does this procedure generate an iterative process or a recursive process?

(b) If iterative, explain why in one sentence. If recursive, rewrite it, **<u>changing as little as possible</u>**, to make it generate an iterative process.

4. Design a procedure **(fast-expt-iter b n)** that returns b^n. Your program evolves an iterative exponentiation process that uses successive squaring and uses a logarithmic number of steps as does **fast-expt** below. (Hint: Using the observation that $(b^{n/2})^2 = (b^2)^{n/2}$, keep, along with the exponent ***n*** and the base ***b***, an additional state variable ***a***, and define the state transformation in such a way that the product ab^n is unchanged from state to state. At the beginning of the process ***a*** is taken to be **1**, and the answer is given by the value of ***a*** at the end of the process. In general, the technique of defining an invariant quantity that remains unchanged from state to state is a powerful way to think about the design of iterative algorithms.)

(hint)
```
(define (fast-expt b n)
        (cond ((= n 0) 1)
              ((even? n) (square (fast-expt b (/ n 2))))
              (else (* b (fast-expt b (- n 1))))))
```

5. Write a procedure **min-of-f-x-and-g-x** that takes two numerical procedures **f** and **g** and a number ***x*** as inputs, and returns the minimum of applying **f** to ***x*** and **g** to ***x***. (You may use the Scheme primitive **min**, which returns the minimum of its arguments.)

```
(min-of-f-x-and-g-x square cube -1)
-1
(min-of-f-x-and-g-x square cube 2)
4
```

```
(define (min-of-f-x-and-g-x f g x)
```

Then generalize these examples so that the procedure you apply to the results of **f** and **g** is a parameter too. For example:

```
(combine-f-x-and-g-x min square cube -1)
-1
```

```
(define (combine-f-x-and-g-x combiner f g x)
```

6. This is a problem about the message passing paradigm.

```
(define (triad left label right)
    (lambda (m)
        (cond ((eq? m 1) left)
              ((eq? m 2) label)
              ((eq? m 3) right))))
```

The above constructor implements a triad **[left label right]** through message passing. Triads can be used to create binary trees; each node has a label, a left child, and a right child.

(a) Define selectors that retrieve the label of the current node, and the left and right child branches.

```
(define (label node)    <some code1>    )
(define (left node)     <some code2>    )
(define (right node)    <some code3>    )
```

Example:

```
(define tridA (triad 3 7 4))
(label tridA)
7
(left tridA)
3
```

(define (label node)

7. We can implement a DrRacket interpreter in two different evaluation methods: applicative order evaluation and normal order evaluation. Two different implementations may give different results for the following cases. What will DrRacket print in response to the following expressions? If an expression produces an error message, you may just write "**error**". If the value of an expression is a procedure, just write "**procedure**". If a program does not terminate, write "**loop**".

Given the following definition:

```
(define (garply x)
    (+ (* x x) 1))
```

and expression:

```
(garply ( (lambda (y) (* y 2)) (* 2 2) ) )
```

(a) How many calls to the ***** (multiplication) operator will there be

in normal order? _____

in applicative order? _____

(b) What does the expression return

in normal order? _____

in applicative order? _____

8. Consider that, in a language that can manipulate procedures, we can get by without numbers (at least insofar as nonnegative integers are concerned) by implementing 0 and the operation of adding 1 as

```
(define zero (lambda (f) (lambda (x) x)))
(define (add-1 n) (lambda (f) (lambda (x) (f ((n f) x)))))
```

This representation is known as Church numerals, after its inventor, Alonzo Church, the logician who invented the calculus. We can define one and two directly.

```
(define one (lambda (f) (lambda (x) (f x))))
(define two (lambda (f) (lambda (x) (f (f x)))))
```

With this encoding, show that **`(add-1 zero)`** is **`one`** as we expect.

(Hint: Use substitution to evaluate **`(add-1 zero)`**). Show your reduction (evaluation) step by step.