Question 0:

Write your name and login on each page, and read and sign the statement below:

|  |
| --- |
| **By signing on the following line, I acknowledge that I’m a huge jerk if I miss Question 0 on the real exam.**  \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |

Question 1:

What will Scheme print? If an expression produces an error message, simply write “error”; if the value of an expression is a procedure, simple write “procedure”.

(keep (lambda (x) x)

(every (lambda (y) (if (even? y) #t y))

‘(1 3 3 7))) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(and or (not #f) (not not) 2) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Question 2:

What will Scheme print in response to the following expressions? If an expression produces an error message, simply write “error”. **Also, draw a box and pointer diagram for the value produced by each expression.**

(cons (list 1 3) (append (list (cons 2 3)) (list 4))) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(list (append (list 3) (cons 4 ‘()))) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Question 3:

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

(define (foo x y) (+ x (\* y y)))

(foo (\* 2 2) (square 3))

How many times is \* called in:

Normal order \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Applicative order \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Question 4:

What is the order of growth in time of the following procedure foo, in terms of its argument value *n*? Also, does it generate an iterative process or recursive process?

(define (foo n)

(if (even? n)

(mystery n)

(mystery (+ n 1))))

(define (mystery x)

(if ((x < 1) 1)

((even? x) (+ 1 (mystery (- x 1))

(mystery (- x 2))))

(else (+ 1 (mystery (- x 2))))))

\_\_\_\_\_ Θ(1) \_\_\_\_\_ Θ(*n*) \_\_\_\_\_ Θ(*n*2) \_\_\_\_\_ Θ(2*n*)

\_\_\_\_\_ Iterative \_\_\_\_\_ Recursive

Question 5:

Write sent-fn that takes an arithmetic function and a list of sentences of numbers and returns a new list of sentences that is the result of calling the function each number in each sentence. For example:

> (sent-fn square ‘((2 5) (3 1 6)))

((4 25) (9 1 36))

**Use higher order functions, not recursion, and respect all relevant data abstractions!**

Question 6:

Sometimes when we see lots of parentheses around a single variable, we get confused as to what it’s supposed to be doing:

((((f))) 1 3)

Write a procedure make-nested that takes a number parens and a procedure end-with and returns a procedure that, when called with parens number of nested parentheses, will invoke end-with after removing itself from the nested parentheses. For the example above, we would use:

(define f (make-nested 3 +))

Which would cause ((((f))) 1 3) to return 4 because there are three nested parens directly around f, and then + is called on 1 and 3.