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 in response to the following expression? If the expression produces an error message, simply write “error”. **Also, draw a box and pointer diagram for the value produced. Don’t forget the start arrow!**

(let ((x (list 2 3))

 (y (list 4 5)))

 (let ((z (append x y)))

 (cons y z))) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Question 2:

Modify the calc code below to support the square procedure. It should work as follows:

STk> (calc)

calc: (square 6)

36

(define (calc-eval exp)

 (cond ((number? exp) exp)

 ((list? exp) (calc-apply (car exp) (map calc-eval (cdr exp))))

 (else (error "Calc: bad expression:" exp))))

(define (calc-apply fn args)

 (cond ((eq? fn '+)

 (accumulate + 0 args))

 ((eq? fn '-)

 (cond ((null? args) (error "Calc: no args to -"))

 ((= (length args) 1) (- (car args)))

 (else (- (car args) (accumulate + 0 (cdr args))))))

 ((eq? fn '\*)

 (accumulate \* 1 args))

 ((eq? fn '/)

 (cond ((null? args) (error "Calc: no args to /"))

 ((= (length args) 1) (/ (car args)))

 (else (/ (car args) (accumulate \* 1 (cdr args))))))

 (else (error "Calc: bad operator:" fn))))

Question 3:

We want to write a system to calculate the average grades of students. However, we'd like to be able to break them down by type of student: graduate student, undergraduate, lazy, motivated, etc.

Write a procedure average-maker that takes a list of type-tagged student grades (i.e. ‘((grad 80) (ugrad 93) (lazy 60))) and **returns a procedure** that takes a type of student and returns the average of that type of students. If the type of student doesn’t exist, return 0.

> ((average-maker '((grad 80) (grad 90) (ugrad 50))) 'grad)

85

You may use the get and set methods discussed in class.

Question 4:

We’d like to represent a set of numbers using object oriented programming. Recall that a set of numbers has no duplicates. (For example, {4 2 3} is a set, but {4 3 3} is not.) Define the set class, with methods add, remove, union, and intersection.

add takes a number as an argument, and adds it into the set if the number doesn’t already exist. Return #t if it successfully adds it to the set, and #f otherwise.

remove takes a number as an argument, and removes it from the set. If it isn’t in the set, don’t do anything. The return value is unimportant.

union takes another set object as an argument and returns a **new** set object that is the (mathematical) union of the current set and the set argument. The union of two sets is defined as all the elements that appear in either set. (Of course, the new set is still a set, so there are no duplicates.)

intersection takes another set object as an argument and returns a **new** set object that is the (mathematical) intersection of the current set and the set argument. The intersection of two sets is defined as all the elements that appear in both sets. (Of course, the new set is still a set, so there are no duplicates.)

Question 5:

At Depth University, a student must complete at least one advanced class to graduate. However, every advanced class has a prerequisite, which may itself have a prerequisite, and so on. Write a procedure fast-grad that, given a tree of prerequisites, with constructor make-tree and selectors datum and children, returns the shortest possible list of courses needed to graduate. If there is a tie, fast-grad may return any of the shortest lists. You may assume that all leaf nodes are advanced classes, and vice versa.

For example, fast-grad called on the following tree can return (CS61A CS70 CS170) or (CS61A CS61B CS184); either one is correct.

 CS61A

 / \

 CS61B CS70

 / \ \

 CS61C CS184 CS170

 / \

 CS164 CS150