Quiz 4a

1. (2 points) What will Scheme 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 Scheme prints procedures. Also, draw a box and pointer diagram for the value produced by each expression.

> (append (cons 1 (list 3)) (list (cons 1 2)))

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

> (list (append (list 2 4) (list (list (list 6))) ‘(3)) ‘(1))

\*Note: append can take 2+ arguments as well

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

1. (3 points) We’re thinking about building a simple calculator using Scheme. This simple calculator is complicated enough that we want to use data abstraction. Our “data” in this case will be called an operation. The constructor, make-operation, takes as input a name, a procedure, and a number (representing the number of arguments to the procedure), and returns a new operation with those characteristics. **Write the constructors and selectors for operations.**

For example, addition could be represented by:

(define addition (make-operation ‘plus 2 (lambda (x y) (+ x y))))

Now, let’s use our data by writing a procedure apply-operation which takes as input an operation and two numbers. If the operation needs something other than 2 arguments, an error is caused (by saying (error “BAD”)). Otherwise, it will apply the operation’s procedure to the two arguments and return that value. **Respect the data abstraction!**

> (apply-operation addition 5 10)

15

1. So that you understand how Scheme prints even better, we are going to write our own version of the procedure that prints pairs! However, we don’t want to talk about printing yet, so instead we’ll just create a sentence of the stuff to be printed, and let the built-in printer print that instead. We are going to write the procedure **pair-as-sent**, which takes as input some pair (possibly containing more pairs inside it) and converts it to a sentence that shows the structure and elements of the pair, in the same way that Scheme does. Unfortunately, we can’t use parentheses, so we’ll use ‘< and ‘> in their place. ‘. will give you the period – there’s no problem with that.

> (pair-as-sent (cons 1 2))

(< 1 . 2 >)

> (pair-as-sent (list 1 2 3))

(< 1 2 3 >)

> (pair-as-sent (cons (cons (cons 1 4) 5) (cons 3 (list 4))))

(< < < 1 . 4 > . 5 > 3 4 >)

1. (2 points) The **pair-as-sent** procedure written below does not print correctly for some inputs. Give us an example of one such input, along with the expected output and what the buggy version of pair-as-sent outputs.

Note: The pair? procedure returns #t if its argument is a pair, and #f otherwise. In particular, (pair ‘()) is #f.

(define (pair-as-sent pair)

(cond ((null? pair) '(< >))

((not (pair? pair)) pair)

(else (se '< (pair-as-sent (car pair)) (pair-as-sent (cdr pair)) '>))))

1. (2 + 1 points) To fix the bug, we have replaced the second call to pair-as-sent with cdr-as-sent because printing cdr-s should be treated differently. Complete the definition of cdr-as-sent below so that pair-as-sent behaves correctly.

1 point will be reserved for not making any data abstraction violations (DAVs). In other words, if you make a decent effort, and don’t make any DAVs, you will get at least 1 point.

(define (pair-as-sent pair)

(cond ((null? pair) '(< >))

((not (pair? pair)) pair)

(else (se '< (pair-as-sent (car pair)) (cdr-as-sent (cdr pair)) '>))))

(define (cdr-as-sent pair)