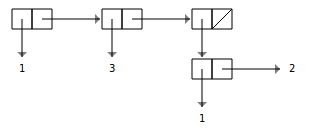
Quiz 4A Rubric

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.

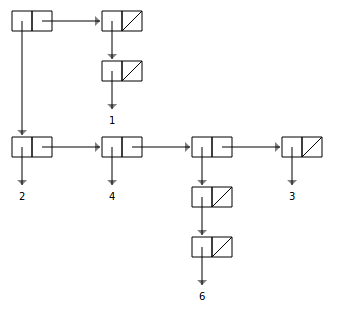
> (append (cons 1 (list 3)) (list (cons 1 2)))

(1 3 (1 . 2))



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

((2 4 ((6)) 3) (1))



Didn’t feel like drawing the box-and-pointer diagrams, ask any TA if you need the answer. Half a point for each part. If the student made the same mistake twice, take off only half a point. (For example, forgetting the starting arrow in both diagrams should only result in losing half a point. If the diagrams are otherwise correct, then give the student the other half a point.

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. For example, addition could be represented by:

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

**Write the constructors and selectors for operations.**

(define make-operation list)

(define operation-name car)

(define operation-num cadr)

(define operation-proc caddr)

0.5 points for each part. Don’t cut points repeatedly for the same error – for example, if a student keeps forgetting the “car” (i.e. uses cdr, cddr instead of cadr, caddr), don’t penalize him/her twice.

Write 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!**

(define (apply-operation op x y)

(if (= (operation-num op) 2)

((operation-proc op) x y)

(error “BAD”)))

1 point, award 0.5 points at your discretion. Don’t be too picky about how they use error – even if it is slightly incorrect, don’t take off points.

1. (5 points) So that you understand how Scheme prints pairs even better, you get to write your 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. **Write a 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 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.**

Hints: Make sure that you deal with the cdr appropriately – you need to do something special if the cdr is itself a pair. (Think back to why lists print the way they do.)

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

Since pair-as-sent only needs to take pairs as input, you can assume that the user won’t call it with the empty list, but your recursion will probably need to make that a special case anyway.

Remember that pair-as-sent takes in a pair and returns a sentence. Don’t lose a point because of a DAV!

(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)

(cond ((null? pair) '())

((not (pair? pair)) (se '. pair))

(else (se (pair-as-sent (car pair)) (cdr-as-sent (cdr pair))))))