PRINT your name: ______________________, ______________________
(last) (first)

SIGN your name: ____________________________________________

PRINT your Inst account login: ________________________________

Your section time (e.g., Tue 3pm): ______________________________

Full name of the person sitting to your left: _____________________

Full name of the person sitting to your right: ____________________

You may consult any books, notes, or other paper-based inanimate objects available to you. Calculators and computers are not permitted. Please write your answers in the spaces provided in the test; in particular, we will not grade anything written on the scratch paper unless specifically told to.

The exam is untimed. There are 7 questions, of varying credit (100 points total). The questions are of varying difficulty, so avoid spending too long on any one question.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem 1</td>
<td>12</td>
</tr>
<tr>
<td>Problem 2</td>
<td>14</td>
</tr>
<tr>
<td>Problem 3</td>
<td>12</td>
</tr>
<tr>
<td>Problem 4</td>
<td>14</td>
</tr>
<tr>
<td>Problem 5</td>
<td>18</td>
</tr>
<tr>
<td>Problem 6</td>
<td>12</td>
</tr>
<tr>
<td>Problem 7</td>
<td>18</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
</tr>
</tbody>
</table>
Problem 1. Box and Pointer

What will the Scheme interpreter print in response to the last expression in each of the following sequence of expressions? Also, draw a Box and Pointer diagram for the final result of each sequence of expressions. If any expression results in an error, just write “ERROR”. HINT: it will be a lot easier if you draw the box and pointer diagram first.

(define x (list 2 3))
(define y (list 1 x 4))
(set-car! x (cddr y))
(set-car! (car x) 5)
x

y -> [*|->5|] x

y -> [1|->[*|->5|]] x

y -> [*|->[5|]*]

((5) 3)

(define x (cons 1 2))
(define y 3)
(set! y (cons x y))
(set-cdr! y x)
(set-car! x 4)
y

y -> [*|->[*|->[*|]]]

((4 . 2) 4 . 2)
Problem 2. Short Answers

**Part A:** What is the value of $x, \ y$ at the end of the following program:

```
(define x 3)
(define y (cons x 2))

(define (foo var1 var2)
  (set-cdr! var1 var2)
  (set! var2 4))

(foo y x)
```

\[ x: 3 \quad y: (3 . 3) \]

**Part B:** We would like to use Object-oriented Programming to represent a CS course. We want our program to have the following organization:

1. A course will have students and a professor
2. each student will maintain a grade
3. each student can also access the class average grade

Consider the following OOP design decisions and circle the ones that implement our desired program organization

1. grade is an instance variable of student class
2. grade is a class variable of student class
3. average-grade is an instance variable of course class
4. average-grade is an class variable of course class
5. A course class will have a list of student objects as instance variable
6. professor class is a subclass of the student class
7. student class is a subclass of the course class
Problem 3. Vector Programming

Write a procedure \(\text{vector-skip vec n}\) that takes a vector and a positive integer \(n\) and returns a new vector that contains every \(n\)-th element of the input vector starting from the first. \text{vector-skip} \ should ignore any extra elements at the end. For example:

\[
\begin{align*}
&> \ (\text{vector-skip} \ #(a \ b \ c \ d \ e) \ 2) \\
&\quad \quad #(a \ c \ e) \\
&> \ (\text{vector-skip} \ #(a \ b \ c \ d \ e) \ 3) \\
&\quad \quad #(a \ d) \\
&> \ (\text{vector-skip} \ #(a \ b \ c \ d \ e) \ 7) \\
&\quad \quad #(a)
\end{align*}
\]

The new vector that is returned should be exactly long enough and not have any empty spaces. You may find the procedure \text{div} \ useful: \text{div} takes two integers, divides them, and discards the decimal part of the quotient. For example:

\[
\begin{align*}
&> \ (\text{div} \ 5 \ 2) \\
&\quad \quad 2 \\
&> \ (\text{div} \ 5 \ 3) \\
&\quad \quad 1 \\
&> \ (\text{div} \ 5 \ 7) \\
&\quad \quad 0
\end{align*}
\]

\textit{Solution:}

\[
\begin{align*}
&(\text{define} \ (\text{vector-skip} \ \text{vec} \ \text{n}) \\
&(\quad (\text{let} \ ((\text{result} \ \text{(make-vector} \ (+ \ 1 \ (\text{div} \ (- \ (\text{vector-length} \ \text{vec}) \ 1) \ \text{n}}))) \ \text{nil}))) \\
&(\quad \quad \text{(define} \ (\text{loop} \ \text{index}) \\
&(\quad \quad \quad \text{(if} \ (= \ \text{index} \ \text{(vector-length} \ \text{result})) \ \text{result} \\
&(\quad \quad \quad \quad \text{(begin} \ \text{(vector-set!} \ \text{result} \ \text{index} \\
&(\quad \quad \quad \quad \quad \text{\ (vector-ref} \ \text{vec} \ (+ \ \text{index} \ \text{n})))))) \\
&(\quad \quad \quad \text{)} \\
&(\quad \quad \text{(loop} \ 0))))
\end{align*}
\]

We need the subtract-by-one in \((+ \ 1 \ (\text{div} \ (- \ (\text{vector-length} \ \text{vec}) \ 1) \ \text{n})))\) for cases when the skip-factor \(n\) is a factor of the length of the original vector. This was tricky to see and we did not take off points for it.
Problem 4. Streams

We can create two dimensional streams by creating a stream whose every element is another stream. Without defining any helper procedure, define an infinite stream int-matrix whose first element is the stream 1, 2, 3, 4, 5..., second element is the stream 2, 4, 6, 8, 10..., third element is the stream 3, 6, 9, 12, 15..., etc.

So, if we start indexing from 1, the \( j \)-th element of the \( i \)-th stream of int-matrix is \( i \cdot j \).

You may use lambda procedures inside stream-map if you want. You may also use the stream ints which is the stream 1, 2, 3, 4, 5...

Solution 1:

\[
\text{(define int-matrix}
\text{  (cons-stream ints}
\text{    (stream-map (lambda (s) (stream-map + ints s)) int-matrix)))}
\]

Solution 2:

\[
\text{(define int-matrix}
\text{  (stream-map (lambda (n) (stream-map (lambda (x) (* n x)) ints)) ints))}
\]
Problem 5. List Mutation

Write a procedure \(\text{exchange \, } ls1 \, \text{ls2} \) that takes two flat lists and exchanges the 2nd, 4th, 6th, 8th ... elements of \(ls1\) and \(ls2\). For example:

\[
\begin{align*}
> & (\text{define \, foo \, } '(a \, b \, c \, d \, e \, f)) \\
> & (\text{define \, bar \, } '(1 \, 2 \, 3 \, 4 \, 5 \, 6)) \\
> & (\text{exchange \, foo \, bar}) \\
& \quad \text{okay} \\
& \quad (a \, 2 \, c \, 4 \, e \, 6) \\
> & \text{foo} \\
& \quad (1 \, b \, 3 \, d \, 5 \, f) \\
\end{align*}
\]

exchange should return okay, preserve original pointer. You may assume that the input lists will have the same length.

Define \text{exchange} procedure \textbf{WITHOUT} using set-car!.

\[
\begin{align*}
(\text{define \, (exchange \, ls1 \, ls2)}) \\
(\text{(cond \, ((null? \, (cdr \, ls1)) \, \, \text{\textquotesingle-okay\textquotesingle})}) \\
(\text{((null? \, (cdr \, ls2)) \, \, \text{\textquotesingle-okay\textquotesingle})}) \\
(\text{(else \, (let \, ((temp \, (cdr \, ls1)))}) \\
\quad (\text{set-cdr! \, ls1 \, (cdr \, ls2)}) \\
\quad (\text{set-cdr! \, ls2 \, temp}) \\
\quad (\text{exchange \, (cdr \, ls1) \, (cdr \, ls2)})))})
\end{align*}
\]
Problem 6. Local State

Consider the following program. Write down what the three calls to foo-1, foo-2 will return on the blank lines provided.

> (define (foo a)
   (let ((b 0))
     (lambda (c)
       (let ((d 0))
         (!a (+ a 1))
         (!b (+ b 1))
         (!c (+ c 1))
         (!d (+ d 1))
         (list a b c d)))))

> (define foo-1 (foo 2))
foo-1
> (foo-1 3)

=> (3 1 4 1)

> (define foo-2 (foo 4))
foo-2
> (foo-2 5)

=> (5 1 6 1)

> (foo-1 7)

=> (4 2 8 1)
Problem 7. Metacircular Evaluator

The metacircular evaluator we have defined requires that all procedures have fixed number of arguments. However, in Scheme, procedures created in the form of \((\text{lambda} \ \text{args} \ ...\)\) or defined like \((\text{define} \ (\text{foo} \ . \ \text{args}) \ ...\)\) can have any number of arguments. For example:

\[\begin{align*}
> & (\text{define} \ \text{num-args} \ (\text{lambda} \ x \ (\text{count} \ x))) \\
& \ \text{num-args} \\
> & (\text{num-args} \ 1 \ 2 \ 3 \ 4) \\
& \ 4 \\
> & (\text{num-args}) \\
& \ 0 \\
\end{align*}\]

\[\begin{align*}
> & (\text{define} \ (\text{add} \ . \ \text{arg-list}) \ (\text{accumulate} \ + \ 0 \ \text{arg-list}))) \\
& \ \text{add} \\
> & (\text{add} \ 1 \ 2) \\
& \ 3 \\
> & (\text{add} \ 1 \ 2 \ 3 \ 4) \\
& \ 15 \\
\end{align*}\]

Directly modify the code of the Metacircular Evaluator given in the next couple of pages to implement this feature. Write down the names of all the procedures you have added/removed/modified in the space below so that we may find it more easily.

**Solution:**

This problem was quite short if you understood the Metacircular evaluator.

If we define lambda in the above way, \((\text{params} \ \text{fn})\) will sometimes be an atom. If that happens, we want to make sure to bind that atom to the list of all arguments. We can either modify \text{extend-environment} or \text{mc-apply}. We offer the former solution here: we added two lines to \text{extend-environment} only.
(define (input-loop)
  (display "=61A=> ")
  (flush)
  (let ((input (read)))
    (if (equal? input 'exit)
      (print "Au Revoir!")
      (begin
        (print (mc-eval input the-global-env))
        (input-loop))))

(define (mc-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((if-exp? exp)
          (if (not (eq? (mc-eval (cadr exp) env) ’nay))
            (mc-eval (caddr exp) env)
            (mc-eval (cadddr exp) env)))
        ((begin-exp? exp)
          (eval-sequence (cdr exp) env))
        ((quote-exp? exp) (cadr exp))
        ((set-exp? exp)
          (set-variable-value! (cadr exp)
            (mc-eval (caddr exp) env)
            env))
        ((definition? exp)
          (if (list? (cadr exp))
            (mc-eval (define->lambda exp) env)
            (define-variable!
              (cadr exp)
              (mc-eval (caddr exp) env) env)))
        ((lambda-exp? exp) (make-procedure (cadr exp) (cddr exp) env))
        ((list? exp) (mc-apply (mc-eval (car exp) env)
          (map (lambda (arg-exp)
            (mc-eval arg-exp env)) (cadr exp))))
        (else (error "UNKNOWN expression"))))

(define (mc-apply fn args)
  (cond ((lambda-proc? fn) (eval-sequence (body fn)
      (extend-environment (params fn) args
        (env fn))))
      (else (do-magic fn args)))))
(define (make-procedure params body env)
  (list 'procedure params body env))

(define (params p)
  (cadr p))

(define (body p)
  (caddr p))

(define (env p)
  (cadddr p))

(define (lambda-proc? p)
  (and (list? p)
       (eq? (car p) 'procedure)))

;;;;;;;;;;;;;;;;;;;;;;;
;; Helper Procedures ;;;
;;;;;;;;;;;;;;;;;;;;;;;

(define (quote-exp? exp)
  (eq? (car exp) 'quote))

(define (define->lambda exp)
  (list 'define (caadr exp)
        (append (list 'lambda (cdadr exp)) (cddr exp)))))

(define (set-exp? exp)
  (eq? (car exp) 'set!))

(define (lambda-exp? exp)
  (eq? (car exp) 'lambda))

(define (begin-exp? exp)
  (eq? (car exp) 'begin))

(define (eval-sequence exps env)
  (cond ((null? (cdr exps)) (mc-eval (car exps) env))
        (else
        (mc-eval (car exps) env)
        (eval-sequence (cdr exps) env))))
(define (if-exp? exp)
  (and (list? exp)
       (eq? (car exp) 'if)))

(define (boolean? exp)
  (or (eq? exp 'aye)
      (eq? exp 'nay)))

(define (do-magic fn args)
  (apply fn args))

(define (definition? exp)
  (eq? (car exp) 'define))

(define (variable? exp)
  (symbol? exp))

(define (self-evaluating? exp)
  (or (number? exp)
      (boolean? exp)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Additional Primitives ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (yell wd)
  (word wd '!!))

(define (square num)
  (* num num))

(define (factorial num)
  (if (= num 0) 1
      (* num (factorial (- num 1)))))

(define (new-null? ls)
  (if (null? ls)
      'aye
      'nay))

(define (new-= num1 num2)
  (if (= num1 num2)
      'aye
      'nay))
(define (new-< num1 num2)
  (if (< num1 num2)
    'aye
    'nay))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Environment Related ;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (extend-environment vars vals base-env)
  (if (atom? vars) ;;ADDED Soln
      (cons (cons (list vars) (list vals)) base-env);;ADDED Soln
      (cons (cons vars vals) base-env)))

(define (define-variable! var val env)
  (define first-frame (car env))
  (define (scan vars vals)
    (cond ((null? vars)
            (if (eq? env the-global-env)
                (error "Unbound Variable")
                (set-variable-value! var val (cdr env))))
            ((eq? var (car vars))
                (set-car! vals val))
            (else
                (scan (cdr vars) (cdr vals))))))
  (scan (car first-frame) (cdr first-frame))
  var)

(define the-global-frame
  (cons (list '+ '- '/ '* 'car 'cdr 'cons 'null? 'nil 'yell 'square 'factorial '=' '< 'list)
    (list + - / * car cdr cons new-null? nil yell square factorial new-= new-< list ))))

(define the-global-env
  (cons the-global-frame nil))

(define (set-variable-value! var val env)
  (define first-frame (car env))
  (define (scan vars vals)
    (cond ((null? vars)
            (if (eq? env the-global-env)
                (error "Unbound Variable")
                (set-variable-value! var val (cdr env))))
            ((eq? var (car vars))
                (set-car! vals val))
            (else
                (scan (cdr vars) (cdr vals))))))
((eq? var (car vars)) (set-car! vals val))
(else (scan (cdr vars) (cdr vals)))
(scan (car first-frame)
(cdr first-frame)))

(define (lookup-variable-value var env)
  (define first-frame (car env))
  (define (scan vars vals)
    (cond ((null? vars)
      (if (eq? env the-global-env)
        (error "Unbound Variable")
        (lookup-variable-value var (cdr env))))
      ((eq? var (car vars)) (car vals))
      (else (scan (cdr vars) (cdr vals))))
  (scan (car first-frame)
    (cdr first-frame)))

(input-loop)