CS 61A Struct. and Interp. of Computer Programs
Summer 2009 Min Xu Final

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 12 questions, of varying credit (100 points total). The questions are of varying difficulty, so avoid spending too long on any one question.

NOTE: On Multiple Choice or True/False questions, we will deduct points for the wrong answer. If a question is worth x points and has n choices, we will deduct $\frac{x}{n-1}$ points for a wrong answer. DO NOT GUESS if you do not know the answer.

Do not turn this page until your instructor tells you to do so.

Problem 1	/ 8
Problem 2	/ 6
Problem 3	/ 10
Problem 4	/ 6
Problem 5 + 6	/ 12
Problem 7 + 8	/ 13
Problem 9	/ 6
Problem 10	/ 16
Problem 11	/ 13
Problem 12	/ 10
Total	/ 100

Problem 1. Recursion (8 points)

The Quicksort algorithm sorts a list of numbers by the following steps:

1. Choose the first element as the pivot
2. For the rest of the list, recursively sort all the elements smaller than or equal to the pivot, call that sorted list *S*
3. For the rest of the list, recursively sort all the elements bigger than the pivot, call that sorted list *B*
4. Combine *S*, the pivot, *B* into one list.

For example, to Quicksort `(4 2 8 6 1 7 4)`

We would look at the elements of the list (excluding the first element) less than or equal to 4, get `(2 1 4)`, and sort that into *S* = `(1 2 4)`.

Then we would look at the elements greater than 4, `(8 6 7)`, sort that into *B* = `(6 7 8)` and then combine *S*, *B*, with the pivot into `(1 2 4 4 6 7 8)`.

Of course, to sort `(2 1 4)` and `(8 6 7)`, we would recursively use the same process.

Fill in the blank below to complete the Quicksort program. DO NOT fight the problem and write your own program, we will not grade additional code.

```
(define (qsort ls)
  (if (null? ls) '()

    (let ((p (car ls)))



      (append (qsort (filter (lambda (x) (<= x p)) (cdr ls)))



              (list p)



              (qsort (filter (lambda (x) (> x p)) (cdr ls)))

    ))))
```

Problem 2. Asymptotic Runtime (6 points)

Part 1: Suppose `foo` is a procedure of one argument that runs in time $\Theta(n)$ and `bar` is a procedure of one argument that runs in time $\Theta(n^2)$. Supposing that the return value of `bar` is a valid input to `foo`, what is the runtime of `(compose foo bar)`?

A. $\Theta(n)$ B. $\Theta(n^2)$ C. $\Theta(n^3)$ D. Not enough information to tell

Explanation: If `bar` returned a value whose size is $\Theta(n^2)$, then the runtime of `(compose foo bar)` would be $\Theta(n^2)$. However, knowing the runtime does not mean you know how big the **output** of `bar` is. For example, `square` runs in constant time since it is just one multiplication but it returns a number of size n^2.

Hence, it could be that `bar` runs in $\Theta(n^2)$ but returns a value of size $\Theta(n^3)$ or $\Theta(n^4)$. Since `foo` is linear with respect to the input, which is in this case, exactly the OUTPUT of `bar`, the composition of `foo` and `bar` would have run-time either $\Theta(n^3)$ or $\Theta(n^4)$.

Part 2: Consider the following program:

```
(define (remove elem ls)
  (cond ((null? ls) '())
        ((eq? (car ls) elem) (cdr ls))
        (else (cons (car ls) (remove elem ls)))))
```

What is the run-time of `remove` in term of length of list n?

A. $\Theta(n)$ B. $\Theta(n^2)$ C. $\Theta(n^3)$ D. $\Theta(n^4)$ E. $\Theta(2^n)$

Does `remove` use an iterative process or a recursive process?

A. Iterative B. Recursive

Part 3:

Consider the following program which takes two lists and checks if one is a permutation of the other: (the `remove` procedure is from the previous problem)

```
(define (permute? ls1 ls2)
  (cond ((not (= (length ls1) (length ls2))) #f)
        ((null? ls1) #t)
        ((memq (car ls1) ls2) (permute? (remove (car ls1) ls1)
                                        (remove (car ls1) ls2)))
        (else #f)))
```

Supposing the two lists have the same length n, what is the run-time of `permute?` in term of length of the lists?

A. $\Theta(n)$ B. $\Theta(n^2)$ C. $\Theta(n^3)$ D. $\Theta(n^4)$ E. $\Theta(2^n)$

Does `permute?` use an iterative process or a recursive process?

A. Iterative B. Recursive

Problem 3. Object-Oriented Programming Below the Line (10 points)

Many Object-oriented Programming languages have **Class Methods**. **A class method ONLY has access to the class variables and ONLY the class can call the class method.** Consider the following below the line implementation:

```
(define student                                  ;; line A
                                                     ;; point 1
 (let ((school 'berkeley))                       ;; line B
   (DEFINE (CHANGE-SCHOOL NEW-SCHOOL)                ;; point 2
      (SET! SCHOOL NEW-SCHOOL))
   (lambda (name)                                ;; line C
     (IF (EQ? NAME 'CHANGE-SCHOOL)                   ;; point 3
         CHANGE-SCHOOL
     (let ((age 18))                             ;; line D
                                                     ;; point 4
       (define (set-age new-age)
         (set! age new-age))

       (define (dispatch m)                                ;; line E
         (cond ((eq? m 'set-age) set-age)
               ((eq? m 'school) (lambda () school))
                                                           ;; point 5
               (else (error ``Unknown Method''))))

       dispatch)))))                                        ;; line F
```

We would like to implement a Class Method `change-school` so that we would get the following desired behavior:

```
> (define chris (student 'chrislin))
 chris
> ((student 'change-school) 'UCLA)
 okay
> ((chris 'school))
 ucla
> ((chris 'change-school) 'UCSD)
 ERROR
```

In the space below and on the next page, write down any new code needed to implement `change-school` as a class method and **state at which of the marked POINTS you would insert your code**. Furthermore, if you would like to change any of the existing lines, make your changes directly on the code above and **state which of the marked LINES you changed** so that we may find it more easily.

Solution:

We modified nothing and inserted code at Point 2 and Point 3. The additional code are in all-caps.

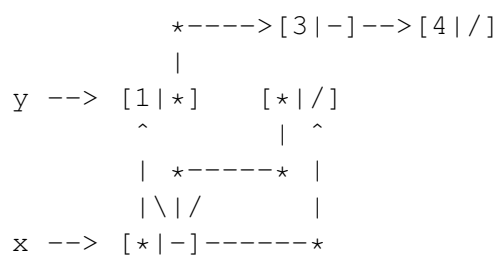
Problem 4. List Mutation (6 points)

Part A: Fill in the blanks below so that we would get the desired results. **Do NOT create any additional pairs**. (HINT: draw box-and-pointer diagram to help you out)

```
> (define y (cons 3 nil))
 y
> (define x (list 1 y 2))
 x


> __(set-car! (cddr x) y)__
 okay


> __(set-car! x (cddr x))___
 okay

> x
 (((3)) (3) (3))
```

Part B: Draw the box-and-pointer diagram created by the following expressions. You DO NOT need to write down the return value of the final expression.

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



Problem 5. Environmental Model (8 points)

Part A: Suppose we run the following code:

```
(define (loop n)
  (if (= n 0) 1
      (loop ((lambda (x) (- x 1)) n))))
(loop 50)
```

How many procedures and frames (NOT including Global) are created by evaluating the above expressions?

Procedures: 51 Frames: 101

Part B: Suppose we run the following code:

```
(define (compose f g) (lambda (x) (f (g x))))
(define (foo x) (let ((y 2)) (+ x y)))
((compose foo foo)  3)
```

How many procedures and frames (NOT including Global) are created by evaluating the above expressions?

Procedures: 5 Frames: 6

Problem 6. Lazy Evaluator (4 points)

Min is super lazy! He argues that the lazy evaluator can be even lazier by delaying the evaluation of the value-expression in define/set! statements.

For example, Min says that if we evaluate `(set! x (+ 1 2))`, we can turn the `(+ 1 2)` into a thunk and only force it when we actually need to evaluate the `x`.

Suppose we implement Min's idea into the Lazy Evaluator and not modify anything else. From below, circle the most accurate description of the consequence:

1. Every program will have the same return value and side effect in Min's lazy evaluator as in the original lazy evaluator. AND the order in which we evaluate the subexpressions stay the same.
2. Every program will have the same return value and side effect in Min's lazy evaluator as in the original lazy evaluator. AND the order in which we evaluate the subexpressions will be different.
3. Some programs will have different return values or side effect in Min's lazy evaluator than in the original lazy evaluator. AND every expressions that do not error or infinite loops in the original lazy evaluator will still not error nor infinite loop in Min's lazy evaluator.
4. **Some expressions that do not error or infinite loop in the original lazy evaluator will now error or infinite loop in Min's lazy evaluator.**

(HINT: consider the expression `(set! x (+ x 1))`)

Explanation:

Let's take the hint and consider `(set! x (+ x 1))`. Suppose we first thunk-ify `(+ x 1)` and bind x to it. If we then try to evaluate x, then we have to force the thunk `[+ x 1]`. But to force that thunk, we need to know the value of x. Hence, we'd get an infinite loop.

Problem 7. Analyzing Evaluator (6 points)

For Part A to Part C, **circle either TRUE or FALSE**.

Part A:
Charles would like the analyzing evaluator to handle the `let` special form by first converting it into a lambda-invocation.

TRUE / *FALSE*:
converting a let-expression into a lambda-invocation can be done by the `analyze` procedure

Explanation: Here we are only changing the text of the program from one form to another. This could easily be done by `analyze`. In fact, `analyze` already ocnverts the `(define (foo x) ...)` into `(define foo (lambda (x) ...))`.

Part B:
Matloob would like to be able to check to see that every variable in his Scheme code is bound to a value before it is evaluated.

TRUE / *FALSE*:
we can modify the `analyze` procedure to check this and correctly signal Unbound Variable error

Explanation: To find out if a variable is bounded or not, we need to evaluate the expressions and set-up the environments. `analyze` only processes the text; it DOES NOT evaluate anything.

Part C:
Chris argues that analyze is inefficient on if-expressions because there is no need to analyze the true-expression if the predicate is false and the false-expression if the predicate is true.

TRUE / *FALSE*:
we can modify the `analyze` procedure to evaluate the predicate of the if-expression and then based on that decide whether to analyze the true-expression or the false-expression.

Explanation: The `analyze` procedure only looks at the text of the code and never evaluates.

Part D:
We say that a lambda-procedure call is *direct* if the procedure is a lambda expression. For example, `((lambda (x) (+ x 1)) 2)` is a *direct* lambda-procedure call while expressions like `(foo 2)` is NOT a *direct* lambda-procedure call assuming `foo` is bound to `(lambda (x) (+ x 1))`.

Only one statement below is CORRECT. **Circle only the correct statement**:

1. We CAN modify `analyze` procedure to check whether all lambda-procedure calls are done with the correct number of arguments and correctly signal Wrong Number of Arguments error for all lambda-procedure calls.
2. **We CANNOT modify** `analyze` **procedure to check whether all lambda-procedure calls are**

done with the correct number of arguments. BUT we CAN modify `analyze` procedure to check whether all DIRECT lambda-procedure calls are done with the correct number of arguments and correctly signal Wrong Number of Arguments error for all DIRECT lambda-procedure calls.

3. We CANNOT modify `analyze` procedure to check whether all DIRECT lambda-procedure calls are done with the correct number of arguments.

Explanation: If the procedure is a variable, then we have to first evaluate the variable and find out what lambda binds to in order to know how many arguments are needed. This requires setting up the environments and hence cannot be done by `analyze`. However, if the procedure is a lambda expression, then we don't need to evaluate the expression to know how many parameter we need.

Problem 8. Metacircular Evaluator (7 points)

Matloob, been the trickster that he always is, decided to play a prank with Charles' Metacircular Interpreter. Matloob made one modification in Charles' Metacircular evaluator: inside `mc-eval`, Matloob changed

```
((lambda-exp? exp)
    (make-procedure (cadr exp) (cddr exp) env))
```

to the code below with the addition provided in caps:

```
((lambda-exp? exp)
    (make-procedure (cadr exp) (LIST (EVAL-SEQUENCE (cddr exp) ENV)) env))
```

For each of the expressions below, circle whether they will have the same return value or side effect or whether they will behave differently after Matloob's change:

1. `(+ 1 (* 1 2))`
   Same *Different*

2. `((lambda () (+ 1 2)))`
   Same *Different*

3. `((lambda (x) (+ x 1)) 2)`
   Same *Different*

Circle whether the following statements describing the consequence of Matloob's change is True or False: (Note that we only care about return values in the statements below)

1. *TRUE / FALSE*: After Matloob's change, all calls to `lambda` procedures with ZERO PARAMETERS will have the SAME return value as before.
2. *TRUE / FALSE*: After Matloob's change, all calls to `lambda` procedures with ONE PARAMETER OR MORE will have DIFFERENT return values than before.

Explanation:

A procedure with no parameters can still access other variables defined in outer environments. For, example, we could have:

```
(define glob 0)
(define (foo) glob)
(set! glob (+ 1 glob))
(foo)
```

With Matloob's change, when we are defining `foo`, we will have evaluated `glob` and stored the value 0 as the body of our procedure even though the correct return value should be 1.

Likewise, if a lambda procedure with one or more parameters does not use any of its parameters nor does it access any variables, then Matloob's change will make no difference. For example:

```
(define (foo x y) 3)
```

Problem 9. Streams (6 points)

Write down the first 8 terms of the following Stream:

```
(define mystery
  (cons-stream 1
     (cons-stream 2
        (interleave mystery
                    (stream-map + mystery (stream-cdr mystery))))))
```

1 2 1 3 2 3 1 4

Extra Credit: Social Implications of Computer Science (1 point)

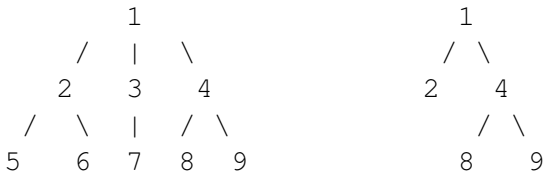
In one sentence, describe who is Elisha Gray and Antonio Meucci:

They both invented the telephones before Alexander Graham Bell. This was mentioned during the lecture on Social Implications of Computer Science.

Problem 10. Tree Recursion (16 points)

A Tree catch on fire,
Certain sub-trees are burnt down,
a new Tree emerge.

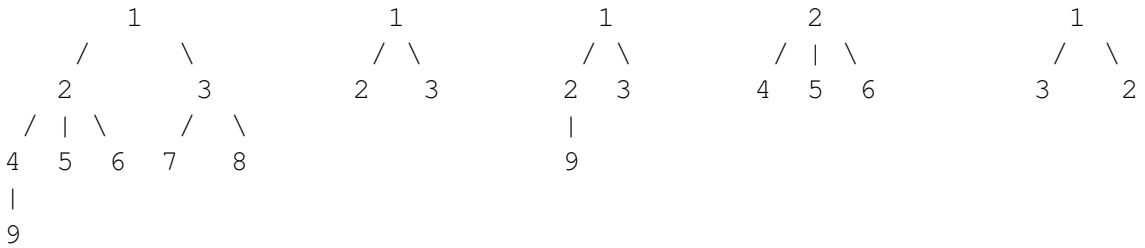
Consider the following two Trees:



Tree A Tree B

We say that Tree B is a **burned-down version** of Tree A because both Trees have the same root and Tree B has the same structure (as in every node has same children in the same order) as Tree A except certain sub-trees got destroyed like the leaves `5, 6` and the branch `3--7`.

As another example:



Tree C Tree D Tree E Tree F Tree G

Here, Tree D is a **burned-down version** of Tree C. Tree E is NOT a **burned-down version** of Tree C because the node 2 does not have 9 has a direct child in Tree C. Tree F is NOT a **burned-down version** of Tree C because they have different roots. Tree G is NOT a **burned-down version** of Tree C because order of the direct children of the root node 1 is wrong. **Note also** that the **burned-down version** might have less children than the original Tree.

On the next page, write a procedure `(burned? full-tree burnt-tree)` that takes two Trees and returns `#t` is the input `burnt-tree` is a **burned-down version** of the input `full-tree`.

Tree Recursion continued...

Write your `burned?` procedure on the space below:

Solution:

```
(define (burned? full-tree burnt-tree)
  (and (equal? (datum full-tree) (datum burnt-tree))
       (burned-f? (children full-tree) (children burnt-tree))))

(define (burned-f? full-ls burn-ls)
  (cond ((null? burn-ls) #t)
        ((null? full-ls) #f)
        ((burned? (car full-ls) (car burn-ls))
            (burned-f? (cdr full-ls) (cdr burn-ls)))
        (else
            (burned-f? (cdr full-ls) burn-ls))))
```

Problem 11. Non-deterministic Programming (13 points)

Consider the `count-change` program you've seen in Lab:

```
(define (count-change total denom-ls)
  (cond ((= total 0) 1)
        ((or (null? denom-ls) (< total 0)) 0)
        (else (+ (count-change (- total (car denom-ls)) denom-ls)
                 (count-change total (cdr denom-ls))))))
```

Write a procedure `(choose-change total denom-ls)` in the Non-deterministic Evaluator. `choose-change` takes a total amount of cents, a list of possible denominations, and successively returns ways to form the total amount of cents with the given possible denominations.

For example:

```
> (choose-change 50 '(25 10 5 1))
 (25 25)
> choose-another
 (25 10 10 5)
> choose-another
 (25 10 5 5 5)
```

Solution:

```
(define (choose-change total denom-ls)
  (require (>= total 0))
  (require (not (null? denom-ls)))
  (if (= total 0) '()
      (choose (cons (car denom-ls) (choose-change (- total (car denom-ls))
                                                  denom-ls))
              (choose-change total (cdr denom-ls)))))
```

Some people also had a solution where `choose-change` could return the same ways of making the change many different times. We reluctantly gave these solutions full credit since we did not specifically say you can't do this.

```
(define (choose-change total denom-ls)
  (require (>= total 0))
  (require (not (null? denom-ls)))
  (if (= total 0) '()
     (let ((coin (an-element-of denom-ls)))
        (cons coin (choose-change (- total coin) denom-ls)))))

(define (an-element-of ls)
```

```
(require (not (null? ls)))
(choose (car ls) (an-element-of (cdr ls))))
```

Problem 12. Concurrency Control (10 points)

Part A:

```
(define x 2)
(define y 4)
(parallel-execute (lambda () (set! x (* x 2)) (set! y (+ y 5)))
                  (lambda () (set! x (+ x y))))
(list x y)
```

What are all possible values of the `(list x y)` expression?

`(12 9)` `(13 9)` `(11 9)` `(4 9)` `(6 9)` `(8 9)`

Part B:
Later on in life, Matloob and Chris became roommates and share a fridge. Periodically, they would check to see if the fridge has any milk left and go to the supermarket to buy some if there are none.

Suppose Matloob and Chris live a very rigid life described by the following code:

```
(define milk-level 100)
(define (has-milk?) (> milk-level 0))
(define (buy-milk)
            (walk-to-supermarket) ;; takes A LONG TIME to finish
            (set! milk-level 100))
(define (drink-milk)
            (set! milk-level (- milk-level 50)))

(define (live)
  (if (not (has-milk?))
    (buy-milk)
    (drink-milk))
  (do-other-things)
  (live))
```

We would like to simulate the lives of Matloob and Chris with the following:

```
(parallel-execute (lambda () (live))
                  (lambda () (live)))
```

This question is continued on the next page.

Concurrency Control Continued...

Our attempt at concurrency control goes as followed:

```
(define s (make-serializer))
(define serial-has-milk? (s has-milk?))
(define serial-buy-milk (s buy-milk))
(define serial-drink-milk (s drink-milk))

(define (live)
  (if (not (serial-has-milk?))
      (begin (serial-buy-milk)
             (live))
      (begin (serial-drink-milk)
             (live))))

;; Everything else is unchanged
```

Circle whether the following statements regarding our attempt is True or False:

1. *TRUE* / *FALSE*: we might have INCORRECTNESS: milk magically appear or disappear
2. *TRUE* / *FALSE*: we might have DEADLOCK: Matloob and Chris wait on each other forever

Explanation:

Suppose Matloob calls `serial-has-milk?` and found that there are 50 milk left. And before he goes on to call `serial-drink-milk`, Chris interrupts and checks to see that there are 50 milk left also and also calls `serial-drink-milk`. Since the `drink-milk` procedure just decrements `milk` by 50, we are going to end up with negative amount of milk since Chris and Matloob both drank milk and there was only enough for one person. There is no deadlock since we have only one serializer.

Aside from possible incorrectness or deadlock, there are other concurrency problems with our attempt. Find **one other concurrency problem** with our code and describe it **concisely** in the space below. DO NOT list more than one concurrency issues; we will only grade the first issue that you describe. (HINT: think about INEFFICIENCY)

Write your answer below:

Solution: There are two inefficiency problems here. One: suppose there are no milk left and Matloob and Chris both checks `serial-has-milk` at the same time. Now both people will go off to supermarket to buy milk even though only `buy-milk` just set milk to 100 so only one person need to go buy milk.

Two: suppose Matloob is off to buy milk. Now if Chris wants to check if there is milk or not, Chris has to wait for a very long time for the serializer. Instead, Chris should be off doing other things.

```
(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)) (cdr 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))))

;;;;;;;;;;;;;;;;;;;;;;
```

```
;; Procedure ADT ;;;;;
;;;;;;;;;;;;;;;;;;;;;
(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))

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

(define (extend-environment vars vals base-env)
  (cons (cons vars vals) base-env))

(define (define-variable! var val env)
  (define first-frame (car env))
  (define (scan vars vals)
    (cond ((null? vars)
     (set-car! first-frame (cons var (car first-frame)))
     (set-cdr! first-frame (cons val (cdr first-frame))))
```

```
    ((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)))))
  (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)
```

This page is intentionally left blank as scratch paper.

This page is intentionally left blank as scratch paper.