QUESTION

Write lookup-variable-value, which takes a variable and starting environment and returns the value associated with the variable or an error if it isn't found after the global environment.

```scheme
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ((null? vars)
          (env-loop (enclosing-environment env)))
          ((eq? var (car vars))
            (car vals))
          (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan (frame-variables frame)
            (frame-values frame))))
    (env-loop env))
)
```

Regular Metaevaluation

QUESTIONS

1. (define (eval-assignment exp env)
   (set-variable-value! (assignment-variable exp) ;; (cadr exp)
     (mc-eval (assignment-value exp) env) ;; (caddr exp)
     env)
   'okay)
   Modify your lookup-variable-value code above to create set-variable-value! (which takes an additional value argument).

   If we find the variable, instead of returning the corresponding value, we should change it:

   ```scheme
   ...                      ...
   (eq? var (car vars)) => (eq? var (car vars))
   (car vals))           (set-car! vals val))
   ...                      ...
   ```

2. (define (eval-definition exp env)
   (define-variable! (definition-variable exp) ;; (cadr exp)
     (mc-eval (definition-value exp) env) ;; (caddr exp)
     env)
   'okay)
   Modify your set-variable-value! code above to create define-variable!. You should write a helper add-binding-to-frame! that takes a variable, value, and frame, and adds the binding into the given frame.

   This should be easier than the questions before, as we don't have to traverse through other environments at all!

   ```scheme
   (define (define-variable! var val env)
     (let ((frame (first-frame env)))
       (define (scan vars vals)
         (cond ((null? vars)
            (add-binding-to-frame! var val frame)))
           (else (scan (cdr vars) (cdr vals))))
     }
   ```
((eq? var (car vars))
  (set-car! vals val))
(else (scan (cdr vars) (cdr vals)))))
(scan (frame-variables frame)
  (frame-values frame)))

(define (add-binding-to-frame! var val frame)
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))

3. Write (extend-environment vars vals base-env) that takes in a list of variables, a list of values, and an environment to extend, and creates the new environment (as when you call a procedure in the environment model).

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (error "Too many arguments supplied" vars vals)
          (error "Too few arguments supplied" vars vals)))))

4. Scheme's map won't work in mc-eval. Why?

The procedure we would try to map is not a Scheme procedure, but a mc-eval procedure. This will fail since you can't map a list onto some arguments.

5. Write (mc-map fn ls) to work with mc-eval. It will be installed as the primitive procedure associated with map. fn is defined in our new representation.

(define (mc-map fn ls)
  (if (null? ls)
      ls
      (cons (mc-apply fn (list (car ls))) (mc-map fn (cdr ls)))))