**SOLUTIONS**

Can I quote you on that?

(`/quasiquote` `expr`) is like `/quote`, but can selectively evaluate pieces. Much like `/quote` can be abbreviated as ``, `/quasiquote` is often shortened as `\`. `/Quasiquote` acts just like `/quote`, except where the following two operators appear in the body of the quotation:

1. `(unquote x)` - give value of `x`. Can be abbreviated with `,`, as in `,x`.

2. `(unquote-splicing x)` - give value of `x`, assume it’s a list, and splice the element into the outer list. Can be abbreviated `,@`, as in `,@x`.

For example, if `foo` is bound to `#t` and `bar` is bound to `(yay rah):

```
`foo bar baz` ; (foo bar baz)
`!,foo bar baz` ; (#t bar baz)
`foo ,bar baz` ; (foo (yay rah) baz)
`foo ,@bar baz` ; (foo yay rah baz)
`foo bar ,baz` ; error: unbound variable baz
`foo bar ,baz` ; (#f bar baz)
```

As demonstrated by the last example, the unquoted expressions aren’t limited to just names.

If `x` is bound to `3`, `y` is bound to `(5 6)`, and `z` is bound to `(7 8 9)`, use quasiquote to build the value `(a 1 2 3 b 4 5 6 (7 8 9) c).

```
`(a 1 2 ,x b 4 ,@y ,z c)
```

If `name` and `value` are bound, use quasiquote to build a define expression that would bind the name to the value.

```
`define ,name ,value`
```

If `params` and `body` are bound, use quasiquote to build a lambda expression with the given parameters and body.

```
`lambda ,params ,@body`
```
And iff’n you know what I mean... 

The `and` special form evaluates its arguments one at a time. If it ever encounters an expression that evaluates to `#f`, it skips evaluating the rest of the expressions, and immediately returns `#f`. If none of the expressions evaluate to `#f`, it returns the value of the last expression. That is:

```lisp
(and #f (/ 1 0)) ; => #f, not an error!
(and #t 2) ; => 2
(and 2) ; => 2
(and) ; => #t
```

Write a syntactic transformer called `and->if` that changes any given `and` expression into a series of nested `if` statements.

```lisp
(define (and->if exp)
  (define (helper exp)
    (if (null? exp)
        #t
        (if (= (length exp) 1)
            (car exp)
            `(if ,(car exp)
              ,(helper (cdr exp))
              #f))
        (helper (cdr exp))))

(define (and->if exp)
  (define (helper exp)
    (if (null? exp)
        #t
        (if (= (length exp) 1)
            (car exp)
            `(if ,(car exp)
              ,(helper (cdr exp))
              #f))
        (helper (cdr exp))))
```

... or iff’n you don’t...

Relatedly, the `or` special form evaluates its arguments one at a time, and returns the first non-false value that it sees. If none of its arguments are true, it returns `#f`. Write a syntactic transformer called `or->if` which changes and given `or` expression into a series of nested `if` statements.

```lisp
(define (or->if exp)
  (define (helper exp)
    (if (null? exp)
        #f
        `(let ((value ,(car exp)))
          (if value
              value
              ,(helper (cdr exp)))))
    (helper (cdr exp))))
```

Double the bubble, double the trouble!

Louis Reasoner thinks it would simplify the evaluator a lot to condense `m-eval` and `m-apply` as follows:
(define (m-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((quoted? exp) (text-of-quotation exp))
        ((assignment? exp) (eval-assignment exp env))
        ((definition? exp) (eval-definition exp env))
        ((if? exp) (eval-if exp env))
        ((lambda? exp)
         (make-procedure (lambda-parameters exp) (lambda-body exp) env))
        ((begin? exp) (eval-sequence (begin-actions exp) env))
        ((cond? exp) (m-eval (cond->if exp) env))
        ((let? exp) (m-eval (let->application exp) env))
        ((application? exp)
         (let ((procedure (m-eval (operator exp) env))
               (arguments (list-of-values (operands exp) env)))
          ;; code from m-apply inserted here
          (cond ((primitive-procedure? procedure) (apply-primitive-procedure procedure arguments))
                ((compound-procedure? procedure)
                 (eval-sequence (procedure-body procedure) (extend-environment (procedure-parameters procedure) arguments env))) ;; can just use env here
                (else (error "Unknown procedure type -- APPLY" procedure))))))
  (else (error "Unknown expression type -- EVAL" exp))))

Does this work? Why or why not?
No – because you drop a frame from the current environment, thus giving you dynamic
scoping, which is useless.

Here's an example of code that would exhibit different behavior with this m-eval:

(define g "good")
(define f (lambda () g))
(define answer
  (let ((g "bad")
        (f)))

See let rec. Rec, let, rec!

The let special form is very useful for defining local variables. Of course, it can also be used to
define local procedures. What is the output of the following? Why?

(let ((fact
       (lambda (x)
         (if (= x 1)
             1
             (* x (fact (- x 1)))))))
  (+ (fact 3) (fact 4)))
How might you extend let to fix this issue? Scheme has a special form which handles this case, called letrec. Write a syntactic transformer, letrec->let, for m-eval.

```
(define (letrec->let exp)
  `(let (,@(map (lambda (var) (list var #f)) (let-bound-variables exp)))
    ,(map (lambda (var val) `(set! ,var ,val))
      (let-bound-variables exp)
      (let-values exp))
    (begin ,@(let-body exp))))
```

Is this the right place for an argument?

As we’ve alluded to a couple times already, some procedures in normal scheme can take an arbitrary number of arguments. This is done by providing an unusual parameter list to lambda, as follows:

```
(define foo (lambda (x y . z) (cons (+ x y) z)))
(foo 1 2) ; => (3)
(foo 1 2 5) ; => (3 5)
```

Remember that ’(x y . z) is interpreted by the reader as an improper list – that is, the same as (cons ’x (cons ’y ’z)). Our version of m-eval doesn’t object to the lambda definition above, but fails to do the right thing when the lambda is called. Alter the extend-environment procedure to support this form.

```
(define (extend-environment vars vals base-env)
  (define (helper rest-vars rest-vals)
    (cond ((and (null? rest-vars) (null? rest-vals)) '
      ()
    ((null? rest-vars)
      (error "Too many args supplied" vars vals))
    ((symbol? rest-vars)
      (list (list rest-vars rest-vals)))
    ((null? rest-vals)
      (error "Too few args supplied" vars vals))
    (else
      (cons (list (car rest-vars) (car rest-vals))
        (helper (cdr rest-vars) (cdr rest-vals))))))
  (let ((matching (helper vars vals)))
    (cons (make-frame (map first matching) (map second matching)) base-env)))
```