Quiz 11a

1. (2 points) Ben Bitdiddle decides to modify mc-eval a bit. He figures that most of the calls to mc-eval will be procedure calls, so to make things faster, he moves the application? call to the top:

(define (mc-eval exp env)

(cond

((APPLICATION? EXP) ;; MOVED TO TOP

(MC-APPLY (MC-EVAL (OPERATOR EXP) ENV)

(LIST-OF-VALUES (OPERANDS EXP) ENV)))

((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) (mc-eval (cond->if exp) env))

;; OLD LOCATION OF APPLICATION?

(else

(error "Unknown expression type -- EVAL" exp))))

What did Ben Bitdiddle break?

\_\_\_\_\_\_\_\_\_ Application of procedures.

\_\_\_\_\_\_\_\_\_ Self-evaluating expressions.

\_\_\_\_\_\_\_\_\_ Variable lookups.

\_\_\_\_\_X\_\_\_\_ Special forms.

\_\_\_\_\_\_\_\_\_ None of the above.

2. (2 points) Suppose we make the following modification to the metacircular evaluator (adding the part in capital letters):

(define (eval-assignment exp env)

(set-variable-value! (MC-EVAL (assignment-variable exp) ENV)

(mc-eval (assignment-value exp) env)

env)

'ok)

This will allow:

(A) (set! (car lst) 3) instead of (set-car! lst 3)

(B) (set! x foo) instead of (set! x 'foo)

***(C) (for-each (lambda (x) (set! x 0)) '(a b c)) to set three variables a, b, and c to zero***

(D) none of the above

3. (3 points) Prog1 is a special form in Common Lisp that's like begin except that it returns the value of its *first* argument expression instead of the last one. (It still evaluates them left to right, though.)

The expression

(prog1 exp1 exp2 exp3 exp4)

is equivalent to this:

(let ((result exp1))

exp2 exp3 exp4 result)

(You may assume that the variable name result is not used within the prog1 subexpressions.)

Implement prog1 in the metacircular evaluator as a derived expression, by writing the procedure prog1->let that translates a prog1 expression into a let expression. Also, say what procedure in the evaluator you'd modify to call prog1->let, and show the code you'd add to that procedure.

**(define (prog1->let expr)**

**(cons 'let (cons (list (list 'result (cadr expr)))**

**(append (cddr expr) (list 'result)))))**

**Also can have a helper:**

**(define (prog1? expr)**

**(and (list? expr) (equal? (car expr) ‘prog1)))**

**Anywhere this clause to the mce-eval cond before the application? clause:**

**((prog1? expr) (mce-eval (prog1->let expr) env))**

4. (3 points) Rewrite *one procedure* in the metacircular evaluator so that it will understand infix arithmetic operators. That is, if a compound expression has three subexpressions, of which the second is a procedure but the first isn't, then the procedure should be called with the first and third subexpressions as arguments:

> (2 + 3)

5

> (+ 2 3)

5

You may write new helper procedures if needed.

**In the**  **mce-eval add an if clause to the application? clause.**

**(if (procedure? (eval (operator exp))**

**;;Normal eval application**

**(mc-apply (mc-eval (car exp) env)**

**(list-of-values (cdr exp) env))**

**(mc-apply (mc-eval (cadr exp) env)**

**(list-of-values (cons (car exp) (cddr exp)) env)))**