Quiz 11b Rubric

1. (2+1 points) Suppose we'd like to add the primitive procedure procedure? into the metacircular evaluator. (Remember, procedure? is a procedure that takes one argument, and returns #t if that argument is a procedure, or #f otherwise.)
   1. Ben Bitdiddle proposes adding the change in the definition of primitive-procedures:

(define primitive-procedures

(list (list 'car car)

...

(list 'procedure? procedure?)))

What would be the return values of the following expressions, typed into the modified evaluator?

MCE> (procedure? +)

#f

MCE> (define (square x) (\* x x))

MCE> (procedure? square)

#f

* 1. Implement the procedure? primitive correctly:

(define primitive-procedures

(list (list 'car car)

...

(list 'procedure? (lambda (x) (or (primitive-procedure? x)

(compound-procedure? x))) )))

If the student said #t for the first part of (a), give full credit for:

(lambda (x) (or (compound-procedure? x) (procedure? x)))

Give 0.5 points for having either primitive-procedure? or compound-procedure?.

1. (3 points) Consider the following procedure:

(define (mystery! var val env)

(add-binding-to-frame! var val (first-frame env)))

Hint: In the following parts, one answer is yes and one answer is no. (However, points will only be given for examples and/or explanations.)

* 1. Suppose we change the definition of set-variable-value! in the metacircular evaluator:

(define set-variable-value! mystery!)

Does this break the metacircular evaluator? If yes, give an example that shows that it is broken. If no, explain why not (in particular, say what the differences between the old definition and the new definition are, and why those differences don’t break the MCE).

Yes, this breaks the evaluator:

(define x 10)

((lambda () (set! x 0)))

x

The last expression “x” should return 0, but with the modification, will return 10.

1 point for a correct example, all or nothing. No partial credit for just “Yes”.

* 1. Repeat part a, but assuming we redefine define-variable!:

(define define-variable! mystery!)

The main difference between the original definition and the new definition is in the case where there is already an existing binding for the variable, we don’t change that variable – instead we add a new binding for the same variable.

This is fine because of the way lookup-variable-value and set-variable-value! work – they look at the bindings in a frame from the front to the back. So, even if we redefine the variable, whenever we try to use the variable (lookup or assignment), we will use the first binding we see in the frame, which will be the new binding of the variable, as required.

1 point for the difference, 1 point for explanation of why it still works.

1. (4 points) Modify the metacircular evaluator (on the next page) to implement a new special form repeat that takes two arguments, a number and an expression. It should evaluate the expression repeatedly, the number of times given by the first argument, which is actually an *expression* whose value is a number. The expression in the second argument should be evaluated each time in an environment in which the variable repcount has the repetition number as its value. For example, if the following expressions are typed into the evaluator:

(define (foo num text)

(repeat num (print (se repcount text))))

(foo 4 'mississippi)

The result (supposing that print and se are available as metacircular primitives) would look like this:

(1 mississippi)

(2 mississippi)

(3 mississippi)

(4 mississippi)

okay

The okay at the end is the value that repeat should return.

You're asked to add a special form, so an obvious part of the problem is to set up the mechanism for MC-EVAL to recognize it. That is, you add

((repeat? exp) (eval-repeat exp env))

as a clause in MC-EVAL's big COND, and you write supporting procedures

(define (repeat? exp)

(tagged-list? exp 'repeat))

(define repeat-number cadr)

(define repeat-expr caddr)

Then comes the hard part, writing EVAL-REPEAT. There are several ways to do this, but the key part is to extend the environment with a frame in which REPCOUNT is bound before evaluating the REPEAT-EXPR each time.

The easiest way is with an explicit call to EXTEND-ENVIRONMENT:

(define (eval-repeat exp env)

(define (helper countup countdown)

(if (<= countdown 0)

'okay

(begin (mc-eval (repeat-expr exp)

(extend-environment '(repcount)

(list countup)

env))

(helper (+ countup 1) (- countdown 1)))))

(helper 1 (mc-eval (repeat-num exp) env)))

Another approach is to turn the repeat-expr into a procedure of the form

(lambda (repcount) ...)

and then MC-APPLY it each time, instead of calling EXTEND-ENVIRONMENT explicitly:

(define (eval-repeat exp env)

(define (helper countup countdown proc)

(if (<= countdown 0)

'okay

(begin (mc-apply proc (list countup))

(helper (+ countup 1) (- countdown 1) proc))))

(helper 1

(mc-eval (repeat-num exp) env)

(make-procedure '(repcount) (repeat-expr exp) env)))

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

((application? exp)

(mc-apply (mc-eval (operator exp) env)

(list-of-values (operands exp) env)))

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

(define (mc-apply procedure arguments)

(cond ((primitive-procedure? procedure)

(apply-primitive-procedure procedure arguments))

((compound-procedure? procedure)

(eval-sequence

(procedure-body procedure)

(extend-environment

(procedure-parameters procedure)

arguments

(procedure-environment procedure))))

(else

(error "Unknown procedure type -- APPLY" procedure))))

(define (make-procedure parameters body env)

(list 'procedure parameters body env))

(define (compound-procedure? p) (tagged-list? p 'procedure))

(define (primitive-procedure? proc) (tagged-list? proc 'primitive))

(define (primitive-implementation proc) (cadr proc))

(define (primitive-procedure-objects)

(map (lambda (proc) (list 'primitive (cadr proc)))

primitive-procedures))

(define (make-frame variables values) (cons variables values))

(define (add-binding-to-frame! var val frame)

(set-car! frame (cons var (car frame)))

(set-cdr! frame (cons val (cdr frame))))

(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))

(define (set-variable-value! var val env)

(define (env-loop env)

(define (scan vars vals)

(cond ((null? vars)

(env-loop (enclosing-environment env)))

((eq? var (car vars))

(set-car! vals val))

(else (scan (cdr vars) (cdr vals)))))

(if (eq? env the-empty-environment)

(error "Unbound variable -- SET!" var)

(let ((frame (first-frame env)))

(scan (frame-variables frame)

(frame-values frame)))))

(env-loop env))

(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))

((eq? var (car vars))

(set-car! vals val))

(else (scan (cdr vars) (cdr vals)))))

(scan (frame-variables frame)

(frame-values frame))))