

MASSACHUSETTS INSTITUTE OF TECHNOLOGY  
Department of Electrical Engineering and Computer Science  
6.001—Structure and Interpretation of Computer Programs  
Fall Semester, 2005

**Final Exam – Auxiliary Material**

The following material may be of use to you in answering questions on the final.

## BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

### 0.1 The Core Evaluator

```
(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))
        ((application? exp)
         (m-apply (m-eval (operator exp) env) (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (m-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 (list-of-values exps env)
  (cond ((no-operands? exps) '())
        (else (cons (m-eval (first-operand exps) env)
                      (list-of-values (rest-operands exps) env)))))

(define (eval-if exp env)
  (if (m-eval (if-predicate exp) env)
      (m-eval (if-consequent exp) env)
      (m-eval (if-alternative exp) env)))

(define (eval-sequence exps env)
  (cond ((last-exp? exps) (m-eval (first-exp exps) env))
        (else (m-eval (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))

(define (eval-assignment exp env)
  (set-variable-value! (assignment-variable exp)
                        (m-eval (assignment-value exp) env)
                        env))

(define (eval-definition exp env)
```

```

(define-variable! (definition-variable exp)
                  (m-eval (definition-value exp) env)
                  env))

```

## 0.2 Representing Expressions

```

(define (tagged-list? exp tag)
  (and (pair? exp) (eq? (car exp) tag)))

(define (self-evaluating? exp)
  (or (number? exp) (string? exp) (boolean? exp)))

(define (quoted? exp) (tagged-list? exp 'quote))
(define (text-of-quotation exp) (cadr exp))

(define (variable? exp) (symbol? exp))
(define (assignment? exp) (tagged-list? exp 'set!))
(define (assignment-variable exp) (cadr exp))
(define (assignment-value exp) (caddr exp))

(define (definition? exp) (tagged-list? exp 'define))
(define (definition-variable exp)
  (if (symbol? (cadr exp)) (cadr exp) (caadr exp)))
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp)
      (make-lambda (cadadr exp) (caddr exp)))) ; formal params, body

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters lambda-exp) (cadr lambda-exp))
(define (lambda-body lambda-exp) (caddr lambda-exp))
(define (make-lambda parms body) (cons 'lambda (cons parms body)))

(define (if? exp) (tagged-list? exp 'if))
(define (if-predicate exp) (cadr exp))
(define (if-consequent exp) (caddr exp))
(define (if-alternative exp)
  (if (not (null? (caddr exp))) (caddr exp) 'false))
(define (make-if pred consequent alt) (list 'if pred consequent alt))

(define (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))

(define (begin? exp) (tagged-list? exp 'begin))
(define (begin-actions begin-exp) (cdr begin-exp))
(define (last-exp? seq) (null? (cdr seq)))
(define (first-exp seq) (car seq))
(define (rest-exps seq) (cdr seq))
(define (sequence->exp seq)
  (cond ((null? seq) seq)

```

```

      ((last-exp? seq) (first-exp seq))
      (else (make-begin seq))))
(define (make-begin exp) (cons 'begin exp))

(define (application? exp) (pair? exp))
(define (operator app) (car app))
(define (operands app) (cdr app))
(define (no-operands? args) (null? args))
(define (first-operand args) (car args))
(define (rest-operands args) (cdr args))

```

### 0.3 Representing procedures

```

(define (make-procedure parameters body env)
  (list 'procedure parameters body env))
(define (compound-procedure? exp)
  (tagged-list? exp 'procedure))
(define (procedure-parameters p) (list-ref p 1))
(define (procedure-body p) (list-ref p 2))
(define (procedure-environment p) (list-ref p 3))

```

### 0.4 Representing environments

```

;; Implement environments as a list of frames; parent environment is
;; the cdr of the list. Each frame will be implemented as a list
;; of variables and a list of corresponding values.
(define (enclosing-environment env) (cdr env))
(define (first-frame env) (car env))
(define the-empty-environment '())

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

(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 args supplied" vars vals)
          (error "Too few args supplied" vars vals))))

(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 -- LOOKUP" 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)) ; Same as lookup except for this
            (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))))


```

## 0.5 Primitive Procedures and the Initial Environment

```

(define (primitive-procedure? proc) (tagged-list? proc 'primitive))
(define (primitive-implementation proc) (cadr proc))
(define primitive-procedures
  (list (list 'car car)
        (list 'cdr cdr)
        (list 'cons cons)
        (list 'null? null?)
        (list '+ +)
        (list '> >)
        (list '= =)
        (list '* *)
        (list 'display display)
        (list 'not not)
        ; ... more primitives
  ))

(define (primitive-procedure-names) (map car primitive-procedures))
(define (primitive-procedure-objects)
  (map (lambda (proc) (list 'primitive (cadr proc))) primitive-procedures))
(define (apply-primitive-procedure proc args)
  (apply (primitive-implementation proc) args))


```

```
(define (setup-environment)
  (let ((initial-env (extend-environment (primitive-procedure-names)
                                        (primitive-procedure-objects)
                                        the-empty-environment)))
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    initial-env))
(define the-global-environment (setup-environment))
```

## 0.6 The Read-Eval-Print Loop

```
(define input-prompt ";;; M-Eval input:")
(define output-prompt ";;; M-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (m-eval input the-global-environment)))
      (announce-output output-prompt)
      (display output)))
    (driver-loop))
(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))
(define (announce-output string)
  (newline) (display string) (newline))
```

## BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS

```
;; Instance
```

```
; instance is a structure which holds the "self" of a normal  
; object instance. It passes all messages along to the handler  
; procedure that it contains.  
;
```

```
(define (make-instance)  
  (list 'instance #f))
```

```
(define (instance? x)  
  (and (pair? x) (eq? (car x) 'instance)))
```

```
(define (instance-handler instance) (cadr instance))
```

```
(define (set-instance-handler! instance handler)  
  (set-car! (cdr instance) handler))
```

```
(define (create-instance maker . args)  
  (let* ((instance (make-instance))  
        (handler (apply maker instance args)))  
    (set-instance-handler! instance handler)  
    (if (method? (get-method 'INSTALL instance))  
        (ask instance 'INSTALL))  
    instance))
```

```
;;-----
```

```
;; Handler
```

```
; handler is a procedure which responds to messages with methods  
; it automatically implements the TYPE and METHODS methods.
```

```
(define (make-handler typename methods . super-parts)  
  (cond ((not (symbol? typename)) ;check for possible programmer errors  
        (error "bad typename" typename))  
        ((not (method-list? methods))  
         (error "bad method list" methods))  
        ((and super-parts (not (filter handler? super-parts)))  
         (error "bad part list" super-parts))  
        (else  
         (let ((handler  
                (lambda (message)  
                  (case message  
                    ((TYPE)  
                     (lambda () (type-extend typename super-parts)))  
                    ((METHODS)  
                     (lambda ()  
                       (append (method-names methods)  
                               (append-map (lambda (x) (ask x 'METHODS))                                           ))                  ))         handler)))
```

```

                                super-parts))))
      (else
        (let ((entry (method-lookup message methods)))
          (if entry
            (cadr entry)
            (find-method-from-handler-list message super-parts))))))
    handler
  ))))

(define (handler? x)
  (procedure? x))

(define (->handler x)
  (cond ((instance? x)
        (instance-handler x))
        ((handler? x)
         x)
        (else
         (error "I don't know how to make a handler from" x))))

; builds a list of method (name,proc) pairs suitable as input to make-handler
; note that this puts a label on the methods, as a tagged list

(define (make-methods . args)
  (define (helper lst result)
    (cond ((null? lst) result)

          ; error catching
          ((null? (cdr lst))
           (error "unmatched method (name,proc) pair"))
          ((not (symbol? (car lst)))
           (if (procedure? (car lst))
               (pp (car lst))
               (error "invalid method name" (car lst))))
          ((not (procedure? (cadr lst)))
           (error "invalid method procedure" (cadr lst))))

        (else
         (helper (cddr lst) (cons (list (car lst) (cadr lst)) result))))))
  (cons 'methods (reverse (helper args '()))))

(define (method-list? methods)
  (and (pair? methods) (eq? (car methods) 'methods)))

(define (empty-method-list? methods)
  (null? (cdr methods)))

(define (method-lookup message methods)
  (assq message (cdr methods)))

(define (method-names methods)

```

```

(map car (cdr methods)))

;;-----
;; Root Object

; Root object. It contains the IS-A method.
; All classes should inherit (directly or indirectly) from root.
;
(define (root-object self)
  (make-handler
   'ROOT
   (make-methods
    'IS-A
    (lambda (type)
      (memq type (ask self 'TYPE))))))

;;-----
;; Object Interface

; ask
;
; We "ask" an object to invoke a named method on some arguments.
;
(define (ask object message . args)
  ;; See your Scheme manual to explain '. args' usage
  ;; which enables an arbitrary number of args to ask.
  (let ((method (get-method message object)))
    (cond ((method? method)
           (apply method args))
          (else
           (error "No method for" message 'in
                  (safe-ask 'UNNAMED-OBJECT
                             object 'NAME))))))

; Safe (doesn't generate errors) method of invoking methods
; on objects. If the object doesn't have the method,
; simply returns the default-value. safe-ask should only
; be used in extraordinary circumstances (like error handling).
;
(define (safe-ask default-value obj msg . args)
  (let ((method (get-method msg obj)))
    (if (method? method)
        (apply ask obj msg args)
        default-value)))

;;-----
;; Method Interface
;;
;; Objects have methods to handle messages.

; Gets the indicated method from the object or objects.

```

```

; This procedure can take one or more objects as
; arguments, and will return the first method it finds
; based on the order of the objects.
;
(define (get-method message . objects)
  (find-method-from-handler-list message (map ->handler objects)))

(define (find-method-from-handler-list message objects)
  (if (null? objects)
      (no-method)
      (let ((method ((car objects) message)))
        (if (not (eq? method (no-method)))
            method
            (find-method-from-handler-list message (cdr objects))))))

(define (method? x)
  (cond ((procedure? x) #T)
        ((eq? x (no-method)) #F)
        (else (error "Object returned this non-message:" x))))

(define no-method
  (let ((tag (list 'NO-METHOD)))
    (lambda () tag)))

; used in make-handler to build the TYPE method for each handler
;
(define (type-extend type parents)
  (cons type
        (remove-duplicates
         (append-map (lambda (parent) (ask parent 'TYPE))
                     parents))))

(define (append-map proc lst)
  (apply append (map proc lst)))

```

**BACKGROUND INFORMATION—THIS PAGE CONTAINS NO QUESTIONS**