MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1998

Lecture Notes – November 10, 1998

The Core Evaluator

```
(define (mini-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) (mini-eval (cond->if exp) env))
        ((application? exp)
         (mini-apply (mini-eval (operator exp) env)
                     (list-of-values (operands exp) env)))
        (else (error "Unknown expression type -- EVAL" exp))))

(define (mini-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 (mini-eval (first-operand exps) env)
                    (list-of-values (rest-operands exps) env)))))

(define (eval-if exp env)
  (if (true? (mini-eval (if-predicate exp) env))
      (mini-eval (if-consequent exp) env)
      (mini-eval (if-alternative exp) env)))

(define (true? value) (eq? value true))

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

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

(define (eval-definition exp env)
  (define-variable! (definition-variable exp)
                    (mini-eval (definition-value exp) env)
                    env))
```

Representing Expressions

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

(define (self-evaluating? exp)
  (or (number? exp) (string? 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 (cdadr exp)          ; formal params
                   (cddr exp))))        ; body

(define (lambda? exp) (tagged-list? exp 'lambda))
(define (lambda-parameters lambda-exp) (cadr lambda-exp))
(define (lambda-body lambda-exp) (cddr 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? (cdddr exp)))
      (cadddr exp)
      'false))
```

```
(define (make-if pred conseq alt) (list 'if pred conseq alt))

(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 (cond? exp) (tagged-list? exp 'cond))
(define (cond-clauses exp) (cdr exp))
(define (cond-else-clause? clause)
  (eq? (cond-predicate clause) 'else))
(define (cond-predicate clause) (car clause))
(define (cond-actions clause) (cdr clause))
(define (cond->if exp)
  (expand-clauses (cond-clauses exp)))
(define (expand-clauses clauses)
  (if (null? clauses)
      'false
    (let ((first (car clauses))
          (rest (cdr clauses)))
      (if (cond-else-clause? first)
          (if (null? rest)
              (sequence->exp (cond-actions first))
            (error "ELSE clause isn't last -- COND->IF"
                   clauses))
        (make-if (cond-predicate first)
                 (sequence->exp (cond-actions first))
                 (expand-clauses rest))))))

(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))
```

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))
```

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))))
```

Primitive Procedures and the Global 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?)
        ; ... 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))
```

The Read-Eval-Print Loop

```
(define input-prompt ";;; Mini-Eval input:")
(define output-prompt ";;; Mini-Eval value:")
(define (driver-loop)
  (prompt-for-input input-prompt)
  (let ((input (read)))
    (let ((output (mini-eval input the-global-environment)))
      (announce-output output-prompt)
      (user-print output)))
  (driver-loop))

(define (prompt-for-input string)
  (newline) (newline) (display string) (newline))
(define (announce-output string)
  (newline) (display string) (newline))
```

```
(define (user-print object)
   (if (compound-procedure? object)
       (display (list 'compound-procedure
                      (procedure-parameters object)
                      (procedure-body object)
                      '<procedure-env>))
       (display object)))
```

Things to Note

The evaluator is written without using any higher order procedures. Therefore it should be possible to implement an interpreter for Scheme in a language which does not support higher order procedures, such as C, or assembly language. The Scheme that is implemented does support higher order procedures.

Destroying Tail Recursion

```
(define (eval-sequence exps env)
  (let ((value (mini-eval (first-exp exps) env)))
    (if (last-exp? exps)
        value
        (eval-sequence (rest-exps exps) env))))
```

Changing to Infix notation

```
(define (operator app)
  (if (= 3 (length app))
      (cadr app)
      (car app)))

(define (operands app)
  (if (= 3 (length app))
      (list (car app) (caddr app))
      (cdr app)))
```

Changing to Dynamic Scoping

```
(define (mini-eval exp env)
  (cond ((self-evaluating? exp) exp)
        ...
        ((application? exp)
         (mini-apply (mini-eval (operator exp) env)
                     (list-of-values (operands exp) env)
                     env))                                ;*****
        (else (error "Unknown expression type -- EVAL" exp))))
```

```
(define (mini-apply procedure arguments env)                  ;*****
  (cond ((primitive-procedure? procedure)
         (apply-primitive-procedure procedure arguments))
        ((compound-procedure? procedure)
         (eval-sequence
          (procedure-body procedure)
          (extend-environment (procedure-parameters procedure)
                              arguments
                              env)))                         ;*****
        (else (error "Unknown procedure type -- APPLY" procedure))))
```

Adding Macros

```
(define (mini-eval exp env)
  (set! exp (macroexpand exp))                               ;*****
  (cond ((self-evaluating? exp) exp)
        ...
        ((macro-definition? exp) (eval-macro-definition exp));*****
        ...
        (else (error "Unknown expression type -- EVAL" exp))))

(define (macroexpand form)
  (let ((new (macroexpand-1 form)))
    (if (eq? new form)
        form
      (macroexpand new))))

(define (macroexpand-1 form)
  (if (and (pair? form)
           (symbol? (operator form)))
      (let ((definition (lookup-variable-in-global (operator form))))
        (if (macro? definition)
            (mini-apply (macro-procedure definition)
                        (operands form))
          form))
      form))

(define (eval-macro-definition exp)
  (define-variable! (definition-variable exp)
                    (make-macro (cdadr exp) (cddr exp))
                    the-global-environment))

(define (make-macro params body)
  (list 'macro (make-procedure params body the-global-environment)))
(define (macro-definition? exp) (tagged-list? exp 'define-macro))
(define (macro? proc) (tagged-list? proc 'macro))
(define (macro-procedure proc) (cadr proc))

(define (lookup-variable-in-global var)
  (define (scan vars vals)
    (cond ((null? vars) false)
  ((eq? var (car vars)) (car vals))
  (else (scan (cdr vars) (cdr vals)))))
  (let ((frame (first-frame the-global-environment)))
    (scan (frame-variables frame) (frame-values frame))))
```

Using Macros

```
(define-macro (unless test alternative)
  (list 'if test 'false alternative))

(define-macro (when test consequent)
  (list 'if test consequent 'false))

(define-macro (setf place value)
  (expand-setf place value))
(define (expand-setf place value)
  (cond ((symbol? place) (list 'set! place value))
        ((pair? place)
         (cond ((eq? (car place) 'car)
                (list 'set-car! (cadr place) value))
               ((eq? (cdr place) 'cdr)
                (list 'set-cdr! (cadr place) value))
               (else (let ((expand (macroexpand place)))
                       (if (eq? expand place)
                           (error "Expression does not expand to a settable place"
                                  place)
                         (expand-setf expand value))))))
        (else (error "Don't know how to SETF" place))))

(define-macro (make-astronaut age gender favorite-drink)
  (list 'list age gender favorite-drink))
(define-macro (astronaut-age astro) (list 'car astro))
(define-macro (astronaut-gender astro) (list 'cadr astro))
(define-macro (astronaut-favorite-drink astro)
  (list 'car (list 'cddr astro)))

;;; or with some syntatic sugar
(define-macro (make-astronaut age gender favorite-drink)
  `(,age ,gender ,favorite-drink))
(define-macro (astronaut-age astro) `(car ,astro))
(define-macro (astronaut-gender astro) `(cadr ,astro))
(define-macro (astronaut-favorite-drink astro)
  `(car (cddr ,astro)))
```