Eval

(define (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))
        ((application? exp)
          (apply (eval (operator exp) env)
                 (list-of-values (operands exp) env)))
        (else
          (error "Unknown expression type -- EVAL" exp)))))
(let ((\(x_1 e_1\) \(x_2 e_2\) \(\cdots\) \(x_n e_n\)) \(b\))
        \(\Rightarrow\)
        ((\(\text{lambda} \ (x_1 x_2 \cdots x_n) \ b\) \(e_1 e_2 \cdots e_n\))

(define (unsugar-let exp)
    (let ((vars (map car (cadr exp)))
          (vals (map cadr (cadr exp)))
          (body (caddr exp)))
      (cons (list 'lambda vars body) vals)))
;Value: unsugar-let

(unsugar-let '(let ((x1 e1) (x2 e2) (x3 e3)) \(b\)))
;Value: ((\(\text{lambda} \ (x1 x2 x3) \ b\) \(e_1 e_2 e_3\))


\[
\begin{align*}
\text{(cond } (p_1 \ e_1) \ (p_2 \ e_2) \ \cdots \ (p_n \ e_n)) &= \\
\text{⇒ } \quad \text{(if } p_1 \ e_1 \text{ (if } p_2 \ e_2 \ \cdots \ \text{(if } p_n \ e_n \ \text{false}) \ \cdots))
\end{align*}
\]

(define (cond->if clauses)
  (if (null? clauses)
      'false
      (list 'if
          (caar clauses)
          (cadar clauses)
          (cond->if (cdr clauses)))))

;Value: cond->if

(cond->if '((p1 e1) (p2 e2) (pn en)))

;Value: (if p1 e1 (if p2 e2 (if pn en false)))
\[ \text{(while \ predicate \ do \ command)} \] = \] 
\[ \text{(if \ predicate} \] 
\[ \text{(begin} \ command \ \text{(while \ predicate \ do \ command)}) \] 
\[ \text{'done) } \]

(define (unsugar-while exp)
  (let ((pred (cadr exp))
         (command (cadddr exp)))
    (list 'if pred (list 'begin command exp) 'done)))

;Value: unsugar-while

(unsugar-while '(while p1 do c))
;Value: (if p1 (begin c (while p1 do c)) 'done)
(define (eval-while exp env)
  (let ((pred (cadr exp))
            (command (cadddr exp)))
    (if (true? (eval pred env))
        (begin (eval command env)
                (eval-while exp env))
        'done)))
;Value: eval-while
(for \( i \in \mathbb{E}_1 \rightarrow \mathbb{E}_2 \) command)
\[ \Rightarrow \]
(let ((\( i \in \mathbb{E}_1 \)) (while (\( \leq i \in \mathbb{E}_2 \)) do
  (begin command (set! \( i \) (1+ \( i \))))))

(define (unsugar-for exp)
  (let ((index-var (cadr exp))
        (from (caddr exp))
        (to (cadddr exp))
        (command (cadr (cdddr exp))))
    (list 'let
      (list (list index-var from))
      (list 'while
        (list '\( \leq \) index-var to)
        'do
        (list 'begin
          command
          (list 'set!
            index-var
            (list '1+ index-var))))))))

;Value: unsugar-for

(pretty-print (unsugar-for '(for count 1 10 (eat-candy)))))
(let
  ((count 1))
  (while (\( \leq \) count 10) do
    (begin (eat-candy) (set! count (1+ count))))))
;No value
(for \(i\) \(e_1\) \(e_2\) \textit{command})

\[\Rightarrow\]

(let ((\(i\) \(e_1\)))
  (while (\(\leq\) \(i\) \(e_2\)) do
    (begin \textit{command} (set! \(i\) (1+ \(i\))))))

(define (unsugar-for exp)
  (let ((\(\text{index-var}\) (cadr exp))
        (from (caddr exp))
        (to (cadddr exp))
        (command (cadr (cdddr exp))))
    `(let ((\(\text{index-var}\) ,from))
      (while (\(\leq\) \(\text{index-var}\) ,to) do
        (begin ,command
          (set! ,\text{index-var} (1+ ,\text{index-var})))))))

;Value: unsugar-for

(pretty-print (unsugar-for ' (for (count 1 10) (eat-candy))))

(let
  ((\(\text{count}\) 1))
  (while (\(\leq\) \(\text{count}\) 10) do
    (begin (eat-candy) (set! \(\text{count}\) (1+ \(\text{count}\)))))

;No value
(define (unsugar-for exp env)
  (let ((index-var (cadr exp))
         (from (caddr exp))
         (to (eval (cadddr exp) env))
         (command (cadr (cdddr exp))))
    '(let ((,index-var ,from))
      (while (<= ,index-var ,to) do
        (begin ,command
                   (set! ,index-var (1+ ,index-var))))))
  )

;Value: unsugar-for

Suppose (upper-bound) evaluates to 320:

(unsugar-for '(for count 1 (upper-bound) (eat-candy)))

(let
  ((count 1))
  (while (<= count 320) do
    (begin (eat-candy) (set! count (1+ count))))))

;No value

So what is the difference between this and the code below?

(let
  ((count 1))
  (while (<= count (upper-bound)) do
    (begin (eat-candy) (set! count (1+ count))))))

Answer: In the above code, the body of the while loop cannot change
the value of the upper bound.

This suggests the delicacy of the decisions that language designers must
make in defining semantics.