Syntactic Sugar: Using the Metacircular Evaluator to Implement the Language You Want

Here is one of the big ideas of this course:

That in many, many programming languages and systems, there is—at their core—something that looks like Scheme.

Why? What really is in Scheme? A way of talking about

Constants and built-in operations
Variables (and naming—i.e., a namespace)
Data structures (and something to make them out of. . . )
Conditional/branching
Procedures (making them, and calling them)—an abstraction mechanism

An inductive syntactic structure to programs

What isn’t as important is the specific way these are realized (applicative order? normal order? dynamic binding? static binding?) or the niceties of the syntax.

Consider the following kind of conditional:
(C (only if) P (and otherwise) A)
(while P do E)
(for i from B to E do C)

The syntax of a language—like a the shape of a nose—doesn’t really matter: it’s what’s in it that counts...
(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 ((x₁ e₁) (x₂ e₂) … (xₙ eₙ)) b)
 ⇒
 ((lambda (x₁ x₂ … xₙ) b) e₁ e₂ … eₙ)

(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: ((lambda (x₁ x₂ x₃) b) e₁ e₂ e₃)
(cond \((p_1 \ e_1) \ (p_2 \ e_2) \ \cdots \ (p_n \ e_n)\))
\[ \implies \]
\((\text{if } p_1 \ e_1 (\text{if } p_2 \ e_2 \ \cdots \ (\text{if } p_n \ e_n \ \text{false}) \ \cdots))\)

(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)))
(while predicate do command)
  \[\Rightarrow\]
  (if predicate
   (begin command (while predicate do command))
   '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: unsugar-while
(for i e1 e2 command)

⇒

(let ((i e1)) (while (<= i e2) 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 '<= 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 (<= count 10) do
    (begin (eat-candy) (set! count (1+ count)))))

;No value
(for i e1 e2 command)

⇒

(let ((i e1)) (while (<= i e2) 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)))
        `((let ((,index-var ,from))
            (while (<= ,index-var ,to) do
              (begin ,command
                (set! ,index-var (1+ ,index-var)))))))))
;Value: unsugar-for

(pretty-print (unsugar-for `((for count 1 10 (eat-candy))))
(let
  ((count 1))
  (while (<= count 10) do
    (begin (eat-candy) (set! count (1+ count)))))
;No value
(for i e1 e2 command)
[Another semantics]

(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.