"Meta-Circular Interpreter"

Compiler

Program → Tokenizer → Parser → Code Generator → Machine

Input → Output

Sequence of characters → (Abstract Syntax Tree) → Sequence of machine instructions

Scheme Machine

Program → Simple Scheme Parser → Map to Scheme Primitives → Scheme Interpreter

Input → Output

Subset of Scheme Expressions (LISTS)
Simple Scheme Interpreter

in Scheme

Interpreter (written in Scheme)

(Virtual Machine)

Input Expression (subset of Scheme)

Value of the Input Expression

1. Use a TABLE to interpret identifiers in the input expression.

2. Use the PRIMITIVE OPERATIONS SUPPORTED BY SCHEME to interpret functions in the input expression.
# Classification of Expressions

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Self evaluating</td>
<td>6</td>
</tr>
<tr>
<td>2) Quote</td>
<td>(quote x)</td>
</tr>
<tr>
<td>3) Identifier</td>
<td>a</td>
</tr>
<tr>
<td>4) Lambda</td>
<td>(lambda (x) (x))</td>
</tr>
<tr>
<td>5) Cond</td>
<td>(cond (f (x) (x)) (t ... y) (... x))</td>
</tr>
<tr>
<td>6) Application</td>
<td>(fun ... x)</td>
</tr>
</tbody>
</table>
(define value
  (lambda (e)
    (meaning e '(()))
  )
)

(define meaning
  (lambda (e table)
    (expression-to-action e e table)
  )
)

[Initial Environment (TABLE)]

Code to evaluate expression depends on its type (DYNAMIC BINDING)
Identification and Syntactic Classification of Expressions

**Expressions**

- **Atoms**
  - (1) Self-Evaluating (numbers)
  - (2) Identifiers

- **Lists**
  - (1) Quote-exp
  - (2) Lambda-exp
  - (3) Cond-exp
  - (4) Application-exp

**Discriminate** using number?

**Type of expr.** \(\rightarrow\) **expression-to-action** \(\rightarrow\) **Action to be taken**
(define expression-to-action
  (lambda (e)
    (cond
      ((Symbol? e) *identifier*)
      ((number? e) *self-evaluating*)
      (else
        (case (car e)
          ((quote) *quote*)
          ((lambda) *lambda*)
          ((cond) *cond*)
          (else *apply*)
        )
      )
    )
  )
)
(define *self-evaluating
  (lambda (e table)
    e
  )
)

(define *identifier
  (lambda (e table)
    (lookup-in-table e table)
  )
)

(define initial-table
  (lambda (CName)
    (case CName
      (t #t
      (nil #f
      (else (cons #primitive (list Name)))
    )
  )
)
(define * lambda
  (lambda (e table)
    (cons 'non-primitive
      (cons table
        (cdr e)))))

For non-primitive procedures build a closure.

/\ Needed for lexical scoping of free-variables
(define + cond
  (lambda (e table)
    (evcon (cdr e) table)
  ))

(define evcon
  (lambda (lines table)
    (cond
      ((meaning (question-of (car lines)))
        table
      )
      ((meaning (answer-of (car lines)))
        table
      )
      (else (evcon (cdr lines) table)
    ))
  ))
(define application
  (lambda (e table)
    (APPPLY
     (meaning (function-of e)
               
               (argument-of e) table)
     
     (evlis (argument-of e) table))
    
    (define evlis
      (lambda (args table)
        (if (null? args) ()
          (cons (meaning (car args) table)
                (evlis (cdr args) table))))))}
(define apply
  (lambda (fun vals)
    (case (car fun)
      ((primitive)
        (apply-primitive
          (second fun) vals))
      ((non-primitive)
        (apply-closure
          (second fun) vals))))
(define apply-closure
  (lambda (Closure vals)
    (meaning
      (body-of Closure)
    (extend-table
      (new-entry
        (formals-of Closure)
        vals
        (table-of Closure)
    )
  )))
(define apply-primitive
  (lambda (name vals)
    (case name
      ((car)  (car (first vals)))
      ((cdr)  (cdr (first vals)))
      ((cons) (cons (first vals) (second vals)))
      ((null?) (null? ...))
      ((eq?)  (eq? ...))
      ((number?) (number? ...))
      ((add1)  (+ (first vals)))
      ((zero?)  (zero? (first vals))))))
Given the interpreter, can the user obtain the functionality of RECURSIVE DEFINITIONS (or do we need to explicitly support "define"-construct)?

The non-recursive definition of Y and its use in defining other "recursive" functions implies that "define" is redundant.
"Another Implementation of Car - Cons - Cdr"

(define (car l) (l #t))
(define (cdr l) (l #f))
(define (cons u v)
  (lambda (c b)
    (cond ((b u) (#t v))
          (c a l))))

**Essence**

(cons (car l) (cdr l)) = l
(car (cons a l)) = a
(cdr (cons a l)) = l