CSc 335 Class 23 April 30 2024

We start by looking at the evaluation of more complex expressions, to better understand the recursive structure of the interpreter.

```
(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
       (cond
         ((eq? (car e) (quote quote))
          *quote)
         ((eq? (car e) (quote lambda))
          *lambda)
         ((eq? (car e) (quote cond))
          *cond)
         (else *application)))
      (else *application))))


; with

(define *application
  (lambda (e table)
    (myapply
     (meaning (function-of e) table)
     (evlis (arguments-of e) table))))

(define function-of car)

(define arguments-of cdr)


; and

(define myapply
  (lambda (fun vals)
    (cond
      ((primitive? fun)
       (myapply-primitive
        (second fun) vals))
      ((non-primitive? fun)
       (myapply-closure
        (second fun) vals)))))
```

applicative order is imposed here!

```
(define primitive?
  (lambda (l)
    (eq? (first l) (quote primitive))))


; and

(define first car)

(define second cadr)

(define third caddr)


; and

(define myapply-primitive
  (lambda (name vals)
    (cond
      ((eq? name (quote cons))
       (cons (first vals) (second vals)))
      ((eq? name (quote car))
       (car (first vals)))
      ((eq? name (quote cdr))
       (cdr (first vals)))
      ((eq? name (quote null?))
       (null? (first vals)))
      ((eq? name (quote eq?))
       (eq? (first vals) (second vals)))
      ((eq? name (quote atom?))
       (:atom? (first vals)))
      ((eq? name (quote zero?))
       (zero? (first vals)))
      ((eq? name (quote add1))
       ((lambda (x) (+ x 1)) (first vals)))
      ((eq? name (quote mul))
       (* (first vals) (second vals)))
      ((eq? name (quote sub1))
       (sub1 (first vals)))
      ((eq? name (quote number?))
       (number? (first vals))))))
```

Now we can evaluate:

(value '(mul (add1 3) (sub1 4)))

(meaning $\underbrace{\text{'(mul (add1 3) (sub1 4))}}_{e}$ '()) →

((expression-to-action e) e '()) →

((list-to-action e) e '()) →

(*application e '()) →

(myapply
((meaning 'mul '()) (evlis ((add1 3) (sub1 4)) '()))

maps meaning with the '() table over the list

table

(('primitive mul) (4 3))) →

(myapply-primitive (4 3)) → 12

```scheme
(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
       (cond
         ((eq? (car e) (quote quote))
          *quote)
         ((eq? (car e) (quote lambda))
          *lambda)
         ((eq? (car e) (quote cond))
          *cond)
         (else *application)))
      (else *application))))

; with

(define *application
  (lambda (e table)
    (myapply
     (meaning (function-of e) table)
     (evlis (arguments-of e) table))))

(define function-of car)

(define arguments-of cdr)

; and

(define myapply
  (lambda (fun vals)
    (cond
      ((primitive? fun)
       (myapply-primitive
        (second fun) vals))
      ((non-primitive? fun)
       (myapply-closure
        (second fun) vals)))))
```

fun is ('primitive some-function-name)

R5RS cond used to implement TLS-cond

env. passing interpreter! Table is always a parameter

```scheme
; cond is a special form that takes any number of
; cond-lines ...  if it sees an else-line, it treats
; that cond-line as if its question part were true.

(define evcon
  (lambda (lines table)
    (cond
      ((else? (question-of (car lines)))
       (meaning (answer-of (car lines))
                table))
      ((meaning (question-of (car lines))
                table)
       (meaning (answer-of (car lines))
                table))
      (else (evcon (cdr lines) table)))))
```

seq. of question-answer pairs $(q_i\ a_i)$

The same table! IE: evaluating an answer does NOT change the env.

```scheme
(define else?
  (lambda (x)
    (cond
      ((atom? x) (eq? x (quote else)))
      (else #f))))

(define question-of first)

(define answer-of second)

(define *cond
  (lambda (e table)
    (evcon (cond-lines-of e) table)))

(define cond-lines-of cdr)
```

(no side effects)

```scheme
; now we can evaluate:

(value '(cond (#f 1) (#t 2)))

(value '(cond ((number? (quote x)) 1) (else 2)))

(value '(mul (add1 (cond (#f 1) (#f 2) (else 3))) 4))
```

meaning will transform this to

```scheme
; suggested exercise: add primitives
;  and, or and not to
; the interpreter and experiment
; with various conditionals
; created using them
```

a number — again
emphasizing the
importance of
recursion in the
working of the
meaning function.

; finally, we add the lambda subsystem --
; in TLS, the only way we associate values with
; variables is via lambda

```
(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
       (cond
         ((eq? (car e) (quote quote))
          *quote)
         ((eq? (car e) (quote lambda))
          *lambda)
         ((eq? (car e) (quote cond))
          *cond)
         (else *application)))
      (else *application))))
```

; with

```
(define *application
  (lambda (e table)
    (myapply
     (meaning (function-of e) table)
     (evlis (arguments-of e) table))))

(define function-of car)

(define arguments-of cdr)
```

; and

```
(define myapply
  (lambda (fun vals)
    (cond
      ((primitive? fun)
       (myapply-primitive
        (second fun) vals))
      ((non-primitive? fun)
       (myapply-closure
        (second fun) vals)))))
```

```
(define *lambda
  (lambda (e table)
    (build (quote non-primitive)
           (cons table (cdr e)))))

(define table-of first)

(define formals-of second)

(define body-of third)

(define non-primitive?
  (lambda (l)
    (eq? (first l) (quote non-primitive))))
```

builds a closure (so you can see what closures actually look like)

```
(define myapply-closure
  (lambda (closure vals)
    (meaning (body-of closure)
             (extend-table
              (new-entry
               (formals-of closure)
               vals)
              (table-of closure)))))
```

call meaning to eval body in the original env. extended by a single entry

(formals vals)
i.e.
(formals actuals)

; with this last definition, we pause to make
; sure everyone understands the concept of closure

; we now actually have a turing-powerful programming system --
; we will start by looking at lambda without recursion, and
; then see how one can use a device called the Y-combinator
; to implement arbitrary recursive functions.

(value '((lambda (x) (add1 x)) 3))

(*application '((lambda (x)(add1 x)) 3)) '())

(myapply (lambda (x) (add1 x)) (3))

(*lambda (lambda (x) (add1 x)) '())

(cdr e):
(formal param list, body)

('non-primitive ('() (x) (add1 x)))

This is the closure created by this particular lambda:

general form of a closure in TLS (table formals body)

What does myapply do with this?

(myapply-closure ('() (x) (add1 x)) (3))

(meaning (add1 x) (((x) (3))))

ie - eval (add1 x) in (x) — (3)

```
(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
       (cond
         ((eq? (car e) (quote quote))
          *quote)
         ((eq? (car e) (quote lambda))
          *lambda)
         ((eq? (car e) (quote cond))
          *cond)
         (else *application)))
      (else *application))))


; with

(define *application
  (lambda (e table)
    (myapply
     (meaning (function-of e) table)
     (evlis (arguments-of e) table))))

(define function-of car)

(define arguments-of cdr)

; and

(define myapply
  (lambda (fun vals)
    (cond
      ((primitive? fun)
       (myapply-primitive
        (second fun) vals))
      ((non-primitive? fun)
       (myapply-closure
        (second fun) vals)))))
```

```
(define *lambda
  (lambda (e table)
    (build (quote non-primitive)
           (cons table (cdr e)))))

(define table-of first)

(define formals-of second)

(define body-of third)

(define non-primitive?
  (lambda (l)
    (eq? (first l) (quote non-primitive))))


(define myapply-closure
  (lambda (closure vals)
    (meaning (body-of closure)
             (extend-table
              (new-entry
               (formals-of closure)
               vals)
              (table-of closure)))))
```

```
(value '(((lambda (y)
            (lambda (x) (cons x y)))
          3)
         4))
```

next page

```
(value '((lambda (x) (add1 x))
             ((lambda (x) (add1 x)) 4)))
```

⋮

X (meaning ((lambda (x)(add1 x)) (5)))

will this particular call ever occur?

[NO]

via list-to-action to *application, then to myapply

(myapply (meaning (lambda (x) (add1 x)) ()))

meaning called recursively

(evlis (((lambda (x) (add1 x)) 4)) ()))

↓

(myapply-closure ('non-primitive (() (x) (add1 x)))

(5))

↓

(meaning (add1 x) (((x) (5))))

```
(value '(((lambda (y)
           (lambda (x) (cons x y)))
          3)
         4))
```

<u>conjecture</u>

we expect at some point to see a table with 2 ribs:

(x)	(3)
(y)	(4)

work this out - first by hand, as we've been doing — and then check your computation by "instrumenting" the tls-scm code with display and newline

```
(define list-to-action
  (lambda (e)
    (cond
      ((atom? (car e))
       (cond
         ((eq? (car e) (quote quote))
          *quote)
         ((eq? (car e) (quote lambda))
          *lambda)
         ((eq? (car e) (quote cond))
          *cond)
         (else *application)))
      (else *application))))

; with

(define *application
  (lambda (e table)
    (myapply
     (meaning (function-of e) table)
     (evlis (arguments-of e) table))))

(define function-of car)

(define arguments-of cdr)

; and

(define myapply
  (lambda (fun vals)
    (cond
      ((primitive? fun)
       (myapply-primitive
        (second fun) vals))
      ((non-primitive? fun)
       (myapply-closure
        (second fun) vals)))))
```

```
(define *lambda
  (lambda (e table)
    (build (quote non-primitive)
           (cons table (cdr e)))))

(define table-of first)

(define formals-of second)

(define body-of third)

(define non-primitive?
  (lambda (l)
    (eq? (first l) (quote non-primitive))))

(define myapply-closure
  (lambda (closure vals)
    (meaning (body-of closure)
             (extend-table
              (new-entry
               (formals-of closure)
               vals)
              (table-of closure)))))
```

```
(value '((lambda (f y)
           (f y))
         (lambda (x) (add1 x))
          4))
```

will pick up here on Tuesday

```
(value '((lambda (x z)
           (cons x
                 ((lambda (x y) (cons z x))
                  3 4)
                 ))
         1 2))
```

Table when this x is evaluated is

(x z) (1 2)

need an interesting
conjecture on the
tables which appear
as this computation
proceeds.

perhaps

(x y)	(3 4)
(x z)	(1 2)

Table for inner lambda

will we see the table
both grow and
shrink?

Hmmm... looks like
growth, but no
shrinkage.

leads to your asking:
how could the program be
modified so that we see
both growth and shrinkage?

```
(value '((lambda (x z)
        (cons x
            ((lambda (x y) (cons z x))
             3 4)
            ))
       1 2))
```

maybe add a subsequent
reference to x??
(outside the inner
lambda)

perhaps

```
(value '((lambda (x z)
   (cons ((lambda (x y) (cons z x)) 3 4)
      x))   1 2))
```

```
; next, let's translate the following simple illustration of a closure to tls-scheme

(let ((x 3))
  (let ((x 4)
        (f (lambda (y) (+ y x))))
    (f 2)))


; to get

(value '((lambda (x)
           ((lambda (x f) (f 2))
            4
            (lambda (y) (+ y x))))
         3))



; you see that tls-scheme features lexical scope and first-class functions --
; after you have had some time to think about how one might go about
; certifying that TLS correctly implements
; lexical scope and first class functions, I will describe proof outlines in class,
; and then ask you to write
; up the arguments for homework.  Big surprise: induction is involved.
```