CSc 335 Class 22 April 18 2024

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; we start off our construction of a scheme interpreter by building its environment subsystem, follow
; interpreter presented in chapter 10 of the little schemer

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; environments implemented as tables, i.e., rib-cages

; pre:
; name is a symbol
; table is a rib-cage, that is, a list of entries, where each entry is a list (names values), where names
;    is a list of symbols and names is the corresponding list of values.  The lists names and values hav
;    same length, and the kth name corresponds to the kth value.
; table-f is a function of one argument, name, which specifies the action taken when name does no
;    any of the entries of the table


; post:
; returns the value associated with name in table - note that the search is entry by entry, starting w
; entry - if this exists, otherwise returns (table-f name)


(define lookup-in-table
  (lambda (name table table-f)
    (cond
      ((null? table) (table-f name))
      (else (lookup-in-entry name
                             (car table)
                             (lambda (name)
                               (lookup-in-table name
                                                (cdr table)
                                                table-f)))))))
```

mutual recursion

catastrophic fail function

topmost rib

local fail function — when name does not occur in the current entry

This is the def of entry-f

Program

program needs to guarantee the pre

func call

func needs to guarantee what the pgm needs from the call.

You might try working out a pf for the suite of functions implementing the environment subsystem. In particular, (1) work out the specification of this subsystem, and (2) see what you can do with the mutual recursion.

```
; pre: name is a symbol
;      entry is a pair of lists (names vals), where names is a lat, and vals is a list of values
;         with length at least as long as that of names
;      entry-f is a function which specifies the action taken when name does not occur in names

; post: returns the value associated with name, with positional association: the kth name is ass
;       if name occurs in the names component of entry.  Otherwise calls entry-f with argument
```

ie, a single rib

```
(define lookup-in-entry
  (lambda (name entry entry-f)
    (lookup-in-entry-help name
                          (names entry)
                          (vals entry)
                          entry-f)))
```

local fail function

```
(define lookup-in-entry-help
  (lambda (name names vals entry-f)
    (cond
      ((null? names) (entry-f name))
      ((eq? (car names) name) (car vals))
      (else (lookup-in-entry-help name
                                  (cdr names)
                                  (cdr vals)
                                  entry-f)))))
```

Positional correspondence between names and vals

```
; simultaneously cdr down the lists names and vals
```

```
; here are constructor and selectors for entries

(define build
  (lambda (s1 s2)
    (list s1 s2)))

(define new-entry build)

(define names
  (lambda (entry) (car entry)))

(define vals
  (lambda (entry) (cadr entry)))




; here are the basic definitions for tables

(define initial-table '())

(define table-f
  (lambda (name)
    (car (quote ()))))

; note that as (car (quote ())) throws an error: when the table is empty,
; attempting to look anything up in it is an error.  See the definition
; for lookup-in-table, above.

; we could just use the error primitive, but I wanted to explain
; the presentation given in tls.  Why didn't they use error?  Perhaps
; to make the point that it is not necessary to have this primitive.
; Or perhaps because error was not included in standard scheme at
; the time the book was written.


(define extend-table cons)
```

```
; testing data

(define names1 '(a b))
(define vals1 (list 1 2))
(define entry1 (new-entry names1 vals1))

(define names2 '(a c))
(define vals2 (list 3 4))
(define entry2 (new-entry names2 vals2))

(define table1 (extend-table entry1 initial-table))
(define table2 (extend-table entry2 table1))
```

Table2 front

(a c) —— (3 4)

(a b) —— (1 2) back

```
; a few tests

(lookup-in-table 'a table2 table-f)

; observe that this returns 3, not 2: a is found in the topmost rib, and so the second rib is never se

(lookup-in-table 'b table2 table-f)

; b is not present in the first rib, and so the second rib is searched
```

what if (lookup-in-table 'z table2 table-f) ?

```
; from the call

;; (lookup-in-entry name
;;              (car table)
;;              (lambda (name)
;;                (lookup-in-table name
;;                                 (cdr table)
;;                                 table-f)))
```

z is not defined in either rib, so the lookup needs to fail. That's the job of table-f.

```
; we see that entry-f has been bound to (lambda (name) (lookup-in-table name (cdr table) table-f))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;^^^^^^^^^^^^

; and that the topmost entry is discarded for the continuation of the search.  We are cdring down t

(lookup-in-table 'd table2 table-f)

; as expected, this fails
```

```
; Perhaps one additional remark is in order for this first look at the environment subsystem of the tls scheme
; interpreter: observe that ANY scheme value at all can occur in the values lists.  We used integer values for
; our example, but soon we will take advantage of the genericity of scheme's lists by (for example) having function
; descriptions occur in them


(define names3 (list 'e 'id))
(define vals3 (list 17 '(lambda (x) x))) ; (lambda (x) x) is here regarded as a description of the identity function
(define entry3 (new-entry names3 vals3))
(define table3 (extend-table entry3 table2))
```

This point is important to understanding how TLS implements higher-order functions.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Now we begin discussion of the entire scheme interpreter presented in Chapter 10 of The Little Schemer.
; This interpreter, which we shall refer to as TLS Scheme, or just TLS, implements a subset of Scheme, which
; we shall also refer to as TLS.  Whether we mean the interpreter or the language will be clear from context.


; TLS scheme has just 6 types of expressions, named *const, *quote, *identifier, *lambda, *cond, and
; *application

; More precisely, the syntax of TLS can be given inductively as follows
```

We will see one way of implementing types as functions

```
; basis step of an inductive definition of TLS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; numbers, #t, #f, cons, car, cdr, null?, eq?, atom?, zero?, add1, sub1, number? have type *const -- we may find it
; desirable to add others later  -- as you will see, this is very easy to do

; symbols have type *identifier


; inductive step of an inductive definition of TLS
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; (quote x) has type *quote, where x is any TLS expression

; (lambda varlist lambda-body) has type *lambda, where varlist is a list of *identifiers and lambda-body is any TLS

; (cond (p1 e1) ... (pk ek)) has type *cond, where the pi and ei are any TLS expressions

; (e1 e2 ... en) has type *application, where the ei are any TLS expressions



; Of course, syntax says nothing at all about the meaning of expressions - it is the interpreter which supplies
; meaning for expressions in TLS.


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

; It will be helpful to look at TLS (the interpreter) in layers. Let us begin by examining the evaluation of
; expressions without variables. You may find the complete interpreter posted on Piazza.

user interface

; If the top level of the interpreter is given as

what about (value 'sub1)?

work through (value '1)

init env always empty

```
(define value
  (lambda (e)
    (meaning e (quote () ))))
```

(meaning 1 '())

syntax directed

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

(expression-to-action 1)

*const

; then we can say that the interpreter computes the value of a given TLS expression e by calling the meaning function
; with the empty list as table parameter. Here the system will use tables - as sketched above - to organize
; information about variables' values. Our limitation to variable-free expressions for this initial look allows us to
; assume that table always has the value (quote ()).

; from the definition of the meaning function, we are motivated to look into expression-to-action

```
(define expression-to-action
  (lambda (e)
    (cond
      ((atom? e) (atom-to-action e))
      (else (list-to-action e)))))
```

get (atom-to-action 1)

; where

```
(define (atom? e)
  (and (not (null? e)) (not (pair? e))))
```

; now one thing at a time, in an attempt to manage complexity -- we look at atom-to-action

```
(define atom-to-action
  (lambda (e)
    (cond
      ((number? e) *const)
      ((eq? e #t) *const)
      ((eq? e #f) *const)
      ((eq? e (quote cons)) *const)
      ((eq? e (quote car)) *const)
      ((eq? e (quote cdr)) *const)
      ((eq? e (quote null?)) *const)
      ((eq? e (quote eq?)) *const)
      ((eq? e (quote atom?)) *const)
      ((eq? e (quote zero?)) *const)
      ((eq? e (quote add1)) *const)
      ((eq? e (quote mul)) *const)
      ((eq? e (quote sub1)) *const)
      ((eq? e (quote number?)) *const)
      (else *identifier))))
```

(atom-to-action 1) = *const

; where

Again,

(value 'sub1) reduces to (*const 'sub1 '())

(*const 1 '()) = 1

('primitive sub1)

```
(define *const
  (lambda (e table)
    (cond
      ((number? e) e)
      ((eq? e #t) #t)
      ((eq? e #f) #f)
      (else (build (quote primitive) e)))))

; with

(define build
  (lambda (s1 s2)
    (list s1 s2)))

; and

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


; as we are assuming there are no identifiers, for the moment, we say nothing further at this time regarding
; *identifier.
```

; As we noted last time, we can already evaluate some simple expressions with the code we have:

(value '1)

why quoted? Well – without the quote, Racket Scheme will of course eval. the arg to value. This is ok for numbers and booleans, but what about

; returns 1

(value '#t)

sub1 is not defined in R5RS, so this would trigger an error — even though sub1 is defined in TLS

; returns #t

(value 'sub1)

The main idea: we quote the input to value so that OUR interpreter gets to evaluate the input.

; returns ('primitive sub1)

; here the tag 'primitive will be used later to further steer the evalution of expressions involving sub1.

; We can evaluate some additional expressions, still without variables. Consider, for example, '(sub1 4). Let's go back to the
; expression-to-action function, and fill in the missing definition

```
(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))))
```

(value '(sub1 4)) reduces to
(*application '(sub1 4) '())

```
; 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)
```

(myapply
(meaning sub1 '())
(evlis (4) '()))

```
; and

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

looks for 'primitive

(myapply
('primitive sub1)
(4))

(myapply-primitive
sub1
(4))

```
; with

(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)))
```

```
(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))))))
```

(sub1 A)

note that sub1 is an R5RS function defined by the user — ie - the user has augmented the interpreter code by (define (sub1 x) (- x 1))

```
; and finally

(define evlis
  (lambda (args table)
    (cond
      ((null? args) (quote ()))
      (else
       (cons (meaning (car args) table)
             (evlis (cdr args) table))))))



; now we can evaluate (sub1 4)

; (on board, in class)

(value '(sub1 4))
```

gives just 3.