CSSE 304 Day 19 Summary

Now that we have seen how environments are supposed to work, we examine some possible implementation approaches

Environment ADT As usual for an ADT, we separate interface from representation and implementation.

Interface (mathematical model) An *environment* is a particular kind of finite function (from symbols to values):

a. **Interface:** If f is an environment, and s, s_1, …, s_k are symbols, then

(empty-env) ➔ ⌈∅⌉ (representation of the empty environment)

(apply-env ⌈f⌉ s) ➔ f(s) (get the value associated with s in the environment f)

(extend-env '(s_1 … s_k) (all of the s_i must be distinct, the v_i may be any Scheme values)
'(v_1 … v_k)
⌈f⌉) ➔ ⌈g⌉ where g(s) is v_i if $s=s_i$ (for some i, $1 \le i \le k$)
f(s) otherwise

One of the slides shows examples of the usage of this interface.

b. Possible environment representations in our interpreter

a. Scheme procedure

b. Record (via define-datatype)

c. List of lists (simple ribcage)

d. List of pairs (first part of each pair is a list, second part is a vector), a more efficient ribcage.

2. Overview of the interpreter project. Details are on the slides.
Starting code is on the back of this page.

```
; Datatype for procedures.  At first there is only one
; kind of procedure, but more kinds will be added later.
(define-datatype proc-val proc-val?
  [prim-proc
   (name symbol?)])

(define-datatype expression expression?
  [var-exp        ; variable references
   (id symbol?)]
  [lit-exp
   (datum (lambda (x)
           (ormap
             (lambda (pred) (pred x))
             (list number? vector? boolean?
                   symbol? string? pair? null?))))]
  [app-exp        ; applications
   (rator expression?)
   (rands (list-of expression?))]
  [else (eopl:error 'expression "not an expression")]
  )

; datatypes and procedures for environments are in the
; code and in the slides

; The main interpreter cod:

 (define eval-exp  ;additional comments
  (lambda (exp)    ; are on PPT slides
    (cases expression exp
     [lit-exp (datum) datum]
     [var-exp (id)
       (apply-env init-env)]
     [app-exp (rator rands)
       (let ([proc-value (eval-exp rator)]
             [args (eval-rands rands)])
         (apply-proc proc-value args))]
      [else (eopl:error 'eval-exp
                        "Bad abstract syntax: ~a"
                         exp)])))


; Evaluate list of operands, return list of results
(define eval-rands  (lambda (rands)
                      (map eval-exp rands)))
```

```
; Apply a procedure to its arguments.
(define apply-proc
  (lambda (proc-value args)
    (cases proc-val proc-value
      [prim-proc (op) (apply-prim-proc op args)]
      [else (error 'apply-proc
                   "Attempt to apply bad procedure: ~s"
                    proc-value)])))

 (define apply-prim-proc
  (lambda (prim-proc args)
    (case prim-proc
      [(+) (+ (1st args) (2nd args))]
      [(-) (- (1st args) (2nd args))]
      [(*) (* (1st args) (2nd args))]
      [(add1) (+ (1st args) 1)]
      [(sub1) (- (1st args) 1)]
      [(cons) (cons (1st args) (2nd args))]
      [(=) (= (1st args) (2nd args))]
      [else (error 'apply-prim-proc
            "Bad primitive procedure name: ~s"
            prim-op)])))

; Set up the initial environment.
(define *prim-proc-names* '(+ - * add1 sub1 cons =))

(define init-env      ; For now, our initial environment
  (extend-env         ; only contains procedure names.
     *prim-proc-names*        ; An environment
     (map prim-proc           ; associates values (not
          *prim-proc-names*)  ; expressions) with
     (empty-env)))            ; variables.


(define rep      ; "read-eval-print" loop.
  (lambda ()
    (display "--> ")
    (let ([answer (top-level-eval (parse-exp (read)))])
       (eopl:pretty-print answer)
       (newline)
       (rep)))  ; tail-recursive, so stack doesn't grow.
```