CSSE 304 Day 25 Summary (First we finish Day 24 material)

```
(define apply-continuation
   (lambda (k . v)
     (apply k v)))
```

1. *Succeed* and *fail* continuations example: **prod-cps**

```
(define prod-cps ; fill in the continuations at the end
  (lambda (L succeed fail)
    (cond [(null? L) (succeed 1)]
          [(zero? (car L)) (fail)]
          [else (prod-cps (cdr L)
```

```
(define print-list-product
 (lambda (list)
    (prod-cps list
        (lambda (prod)
           (printf "The product is ~s~n" prod))
        (lambda ()
           (printf "zero found, product is 0”)
```

Another similar example, substitute-leftmost (you may want to annotate this:

```
(define substitute-leftmost
  (lambda (new old slist)
    (subst-left-cps
      new
      old
      slist
      (lambda (v) v)    ; changed continuation
      (lambda () slist) ; unchanged continuation
    )))


define subst-left-cps  ; changed and unchanged are continuations
 (lambda (new old slist changed unchanged)
  (let loop ([slist slist] [changed changed] [unchanged unchanged])
   (cond
    [(null? slist) (apply-k unchanged)]
    [(symbol? (car slist))
     (if (eq? (car slist) old)
         (apply-k changed (cons new (cdr slist)))
         (loop (cdr slist)
               (make-k (lambda (substituted-cdr)
                         (apply-k changed
                                  (cons (car slist) substituted-cdr))))
                unchanged))]
       [else ; car is an s-list
       (loop (car slist)
             (make-k (lambda (substituted-car)
                       (apply-k changed (cons substituted-car (cdr slist)))))
             (make-k (lambda ()
                    (loop (cdr slist)
                          (make-k (lambda (substituted-cdr)
                                    (apply-k changed
                                             (cons (car slist)
                                                   substituted-cdr))))
                           unchanged))))]))))
```

Add `letrec` to the interpreted language.

We only handle a special case of `letrec`, where all `letrec` variables are bound to procedures.

Concrete syntax: **(letrec ([var <lambda-exp>] ...) body body2 . . .)**

Abstract syntax: a new variant for the `expression` datatype:

```
[letrec-exp
   (proc-names (list-of symbol?))
   (idss (list-of (list-of symbol?)))
   (bodiess (list-of (list-of expression?)))
   (letrec-bodies (list-of expression?))]
```

Today I include a lot of code from the slides so you can annotate it as we discuss it.

Letrec evaluation

- Closures are created and added to the `letrec` environment. Bodies of the `letrec` are evaluated in order.
- When one of the `letrec` closures is applied, new environment must extend the letrec environment

• If it were `let` instead of `letrec`, the new env when closure is applied would extend the enclosing environment instead

So the question becomes: how do we implement extend-env-recursively?

```
(define eval-exp
  (lambda (exp env)
    (cases expression exp
          . . .
     [letrec-exp
       (proc-names idss bodiess letrec-body)
          (eval-bodies letrec-bodies
                (extend-env-recursively
           proc-names idss bodiess env))]
```

0. Implement `extend-env-recursively` in terms of Scheme's `letrec`.

1. No mutation: A new kind of environment extension:
recursively-extended-env-record

2. Mutation: A normal extended environment, but it uses **vector-set!** to fix things up.

3. No-mutation details:

```
(define extend-env-recursively
  (lambda (proc-names idss bodies old-env)
    (recursively-extended-env-record
      proc-names idss bodies old-env)))
```

New case for apply-env

```
[recursively-extended-env-record
  (procnames idss bodies old-env)
    (let ([pos
      (list-find-position sym procnames)])
     (if (number? pos)
         (closure (list-ref idss pos)
                  (list-ref bodies pos)
                  env)
         (apply-env old-env sym)))])))
```

```
(define-datatype environment environment?
  [empty-env-record]
  [extended-env-record
   (syms (list-of symbol?))
   (vals (list-of scheme-value?))
   (env environment?)]
  [recursively-extended-env-record
    (proc-names (list-of symbol?))
    (idss (list-of (list-of symbol?)))
    (bodiess (list-of (list-of expression?)))
    (env environment?)])
```

Mutation solutions: Modified ribcage approach, syntax-expand approach. Details on slides (may happen on the next class day).