CSSE 304 Day 25 Summary

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?))]
```

Notes on the even-odd example:

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?)])
```

Next page: Mutation solutions: Modified ribcage approach, syntax-expand approach.

```
(define extend-env-recursively
  (lambda (proc-names idss bodiess old-env)
    (let ([len (length proc-names)])
      (let ([vec (make-vector len)])
        (let ([env (extended-env-record
                    proc-names vec old-env)])
          (for-each
             (lambda (pos ids bodies)
                (vector-set! vec
                             pos
                               (closure ids bodies env)))
             (iota len)
               idss
               bodiess)
           env)))))
```

Code to be syntax expanded:

```
(define odd?
  (letrec ([o? (lambda (n)
                 (if (zero? n)
                     #f
                     (e? (- n 1))))]
           [e? (lambda (m)
                 (if (zero? m)
                     #t
                     (o? (- m 1))))])
    o?))
```

(If there is time) Add set! to the interpreted language.

Reference ADT: `(deref ref)` and `(set-ref! ref val).` If we have `apply-env-ref`, we get `apply-env` "almost for free"

The `set!` case of `eval-exp` is simple:

```
[set!-exp (id exp)
          (set-ref! (apply-env-ref env id)
                    (eval-exp exp env))]
```

One approach to implementing references: the *cell* ADT:

`(cell value)` creates a cell containing the value.
`(cell-ref cell)` gives us the value in the cell.
`(cell-set! cell value)` replaces the value in the cell.
`(cell? obj)` asks if an object is a cell.

Use cells to implement the *reference* ADT. (i.e. to implement `deref` and `set-ref!`)

In the `extend-env` implementation, replace **`vals`** with **`(map cell vals)`**