**CSSE 304 Day 26**

## **Add set! to the interpreted language.** Some or all of this may not happen today.

**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)`**

Code for `apply-env-ref`

**Now all that is left is to implement cells.**

1. A cell could be a pair:

2. A cell could be a vector:

3. A cell could be a built-in *Chez* Scheme datatype: a box.

# Warmup for call/cc

1. Continuations review: Consider the evaluation of the expression:

```
(let ([x (+ y 2)])
   (if (< x 4) 5 (- x 6))
```

   What is the continuation of

   `(+ y 2) ?` `6 ?`

   `(- x 6) ?` `(< x 4) ?`

2. A **receiver** is an argument (which happens to also be a procedure) passed to a procedure, with the intention that the procedure will eventually pass values to that argument. In some situation, receivers are referred to as “callbacks”.

   a. The continuations that we pass to CPS procedures are receivers.

   b. The consumer procedures that we pass `to call-with-values` are receivers.

   c. `call-with-output-file` is another example of a procedure that expects a receiver as an argument.

3. Suppose that we have a procedure `escape-+` that adds its arguments and returns this sum as the final answer, no matter what the context.

   a. `(* (escape-+ 5 6) 3)` ➔

   b. `(escape-+ (escape-+ 2 4) 5)` ➔

4. More generally, suppose that we have a procedure `escaper` that takes a procedure as an argument and returns an equivalent escape procedure.

   a. (escaper +) creates a procedure that is equivalent to `escape-+`

   b. `(+ 3 ((escaper +) 4 5))` ➔

   c.

```
(+ ((escaper (lambda (x)
               (- (* x 3) 7)))
     5)
   4)          ➔
```

   d. A slide gives details of how you can experiment with `escaper`.

5. Let *p* be a procedure. If an application of *p* abandons the current continuation and does something else instead, we call *p* an ***escape procedure***.

   a. An example of a Scheme escape procedure that we have already used:

   Is `escaper` an escape procedure?