Another letrec implementation approach: Use `syntax-expand` to replace `letrec` by a `let` plus mutation.

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

Expands to …

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)`**

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.