set! Expressions

Assignment expressions have a different nature than the functional parts of MiniScheme. The set! expression introduces state into our language. We need something with state to enable state. There are a number of ways to achieve this; the one we will take uses a feature of Scheme that was introduced just for this purpose -- to model state.

box is a datatype in Scheme that holds a mutable value. The datatype has

- Constructor *box* that takes one value
- Recognizer *box?*
- Getter *unbox*
- Mutator or setter *set-box!* that takes two arguments: a box and a value. (set-box! b x) changes the value stored in box b to x.

For example

- We might create a box holding the value 32 with (define b (box 32))
- We can get the value stored in b with (unbox b), which returns 32
- We can change the value stored in b with (set-box! b 64)
- Now if we again (unbox b) we get the current value 64.

This models the way variables work in non-functional languages.

To implement set! we change our interpreter so that *everything* in the environment is boxed. When we lookup values in the environment, which only happens when we evaluate var-refs, we get a box containing the value. Usually we will unbox the box to get the actual value. When we evaluate a set! expression, such as (set! x 23), we will lookup x in the environment to get its box b, then set this box to 23 with (set-box! b 23).

We do this in three steps:

set! step 1: We need to box every value in the environment.

There are two ways to do this.

- If you are young and cocky and sure you can find every place you extend the environment you can replace each call

  ```
  (extended-env syms vals old-env)
  ```

  with

  ```
  (extended-env syms (map box vals) old-env)
  ```

- If you have 68 years of experience with screwing up, you might prefer to change the definition of extended-env:

  ```
  (define extended-env (lambda (syms vals old)
                         (list 'extended-env syms (map box vals) old)))
  ```

set! step 2:

- Do NOT change your lookup function.
- Do change your line in eval-exp that evaluates var-refs from

  ```
  [(var-ref? tree) (lookup env (var-ref-symbol tree))]
  ```

  to

  ```
  [(var-ref? tree) (unbox (lookup env (var-ref-symbol tree))]
  ```

At this point your interpreter should work exactly as it did before you introduced boxes. Check that out carefully, especially let and lambda expressions.

set! step 3

set! expressions have the form (set! symbol expression).
You will need a new datatype to handle such expressions. I call it *assign-exp* with constructor *assign-exp*, recognizer *assign-exp?* and getters *assign-exp-symbol* and *assign-exp-expression*.

When parsing put the unparsed symbol (i.e., x instead of (var-ref x)) into the datatype and the parsed expression.

set! step 3, continued

In eval-exp, your line for evaluating an assign-exp tree in environment env will

a) lookup the symbol in env to get a box. You might even do this in a let expression, as in

```
        (let ([b (lookup env (assign-exp-symbol tree))])
                .....
```

b) call eval-exp on (assign-exp-expression tree) to get the value v of this expression

c) (set-box! b v)

Now MiniScheme has set!, but it isn't of much use until we can execute a sequence of expressions, such as

```
(let ([x 0])
        (begin
                (set! x 23)
                (+ x 5)))
```

The begin expression has the form

(begin exp_1 exp_2 exp_3 ...exp_n)

We will parse this into a new datatype begin-exp.
We put into it a list of the parsed subexpressions, so it will look like

('begin-exp ((parse exp_1) (parse exp_2) ...(parse exp_n)))

eval-exp will take such a datatype and evaluate it one expression at a time. You might make a helper function that evaluates a list of parsed expressions (without the 'begin-exp). If the list has only 1 element return its value; otherwise evaluate the car of the list and return the value of recursing on the cdr.