Exercise 4.17.

Draw diagrams of the environment in effect when evaluating the expression $<e3>$ in the procedure in the text, comparing how this will be structured when definitions are interpreted sequentially with how it will be structured if definitions are scanned out as described. Why is there an extra frame in the transformed program? Explain why this difference in environment structure can never make a difference in the behavior of a correct program. Design a way to make the interpreter implement the "simultaneous" scope rule for internal definitions without constructing the extra frame.

Answer.

Figure 1 and figure 2 each shows the point in the evaluation of $<e3>$ in the procedure in the text

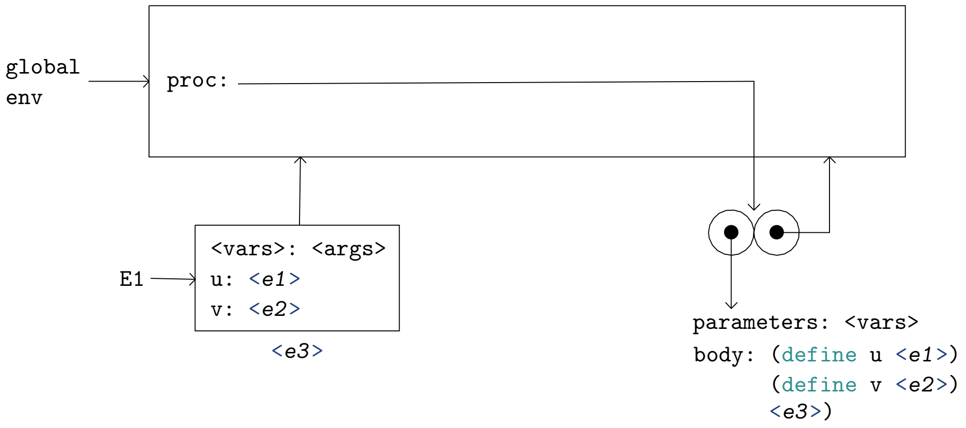

Figure 1. Environments created by evaluating $<e3>$ where definitions are interpreted sequentially.

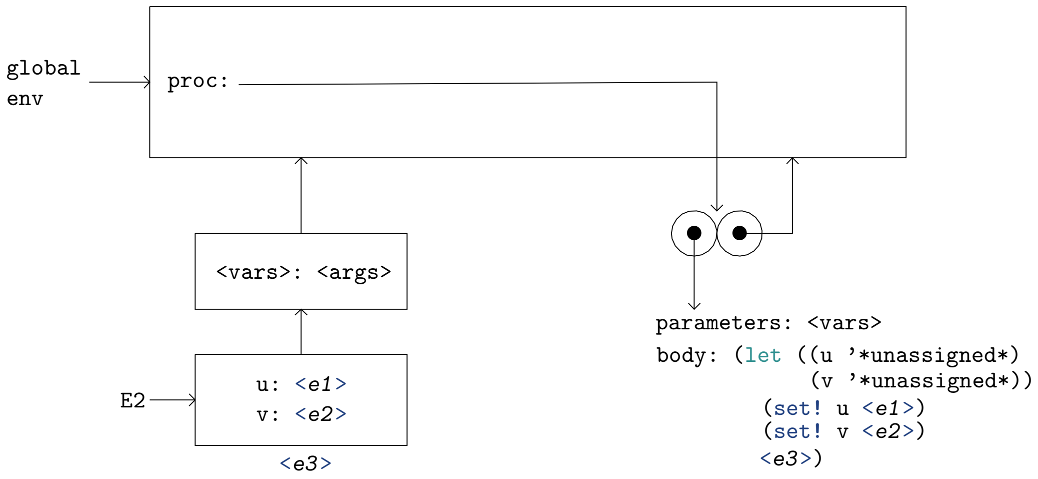

Figure 2. Environments created by evaluating $<e3>$ where definitions are scanned out.

where definitions are either interpreted sequentially or scanned out as described. Recall from section 1.3.2 that `let` is simply syntactic sugar for a procedure call:

```
(let ((u '*unassigned*)
      (v '*unassigned*))
  (set! u <e1>)
```

*. Creative Commons 2014, Lawrence X. Amlord (颜世敏, aka 颜 序).
Email address: informlarry@gmail.com

```
    (set! v <e2>)
    <e3>)
```

is interpreted as an alternate syntax for

```
((lambda (u v)
   (set! u <e1>)
   (set! v <e2>)
   <e3>)
 '*unassigned* '*unassigned*)
```

The latter one is a procedure application and by the environment model of evaluation, we extend the base environment to include a frame that binds the parameters of the procedure (`u` and `v`) to the arguments to which the procedure is to be applied (`'*unassigned*` and `'*unassigned*`). Hence, the diagram of transformed program includes an extra frame.

Since our syntax transformation creates a `let` to embrace all the internal definitions in the body of the `lambda` expression. This doesn't extend or shrink the scope of any variable inside the procedure. Hence this difference in environment structure can never make a difference in the behavior of a correct program.

Alternatively, there is a strategy in which the interpreter implement the "simultaneous" scope rule for internal definitions without constructing the extra frame. To do this, we rearrange the sequence of expressions and make internal definitions come first inside the `lambda` expression. Note that the body of a `lambda` expression is evaluated only when the procedure is applied to arguments. So all the internal definitions can use variables that defined after it. This automatically brings us the "simultaneous" scope for internal definitions and no extra frame is created in the process.