Exercise 4.30.

Cy D. Fect, a reformed C programmer, is worried that some side effects may never take place, because the lazy evaluator doesn't force the expressions in a sequence. Since the value of an expression in a sequence other than the last one is not used (the expression is there only for its effect, such as assigning to a variable or printing), there can be no subsequent use of this value (e.g., as an argument to a primitive procedure) that will cause it to be forced. Cy thus thinks that when evaluating sequences, we must force all expressions in the sequence except the final one. He proposes to modify eval-sequence from section 4.1.1 to use `actual-value` rather than eval:

```
(define (eval-sequence exps env)
  (cond ((last-exp? exps) (eval (first-exp exps) env))
        (else (actual-value (first-exp exps) env)
              (eval-sequence (rest-exps exps) env))))
```

a. Ben Bitdiddle thinks Cy is wrong. He shows Cy the `for-each` procedure described in exercise 2.23, which gives an important example of a sequence with side effects:

```
(define (for-each proc items)
  (if (null? items)
      'done
      (begin (proc (car items))
             (for-each proc (cdr items)))))
```

He claims that the evaluator in the text (with the original `eval-sequence`) handles this correctly:

```
;;; L-Eval input:
(for-each (lambda (x) (newline) (display x))
          (list 57 321 88))
57
321
88
;;; L-Eval value:
done
```

Explain why Ben is right about the behavior of `for-each`.

b. Cy agrees that Ben is right about the `for-each` example, but says that that's not the kind of program he was thinking about when he proposed his change to `eval-sequence`. He defines the following two procedures in the lazy evaluator:

```
(define (p1 x)
  (set! x (cons x '(2)))
  x)

(define (p2 x)
  (define (p e)
    e
    x)
  (p (set! x (cons x '(2)))))
```

What are the values of `(p1 1)` and `(p2 1)` with the original `eval-sequence`? What would the values be with Cy's proposed change to `eval-sequence`?

c. Cy also points out that changing `eval-sequence` as he proposes does not affect the behavior of the example in part a. Explain why this is true.

d. How do you think sequences ought to be treated in the lazy evaluator? Do you like Cy's approach, the approach in the text, or some other approach?

Answer.

a. Ben is right about the behavior of `for-each` because `car` is a primitive operator in Scheme. Hence, thunks would be forced automatically before passed to it (in Ben's case, it is the thunk packaged `items`). Besides, `proc` will also be automatically forced before applied. Note that our lazy evaluator delays arguments in an expression rather than expressions in a sequence. So long as the values of its arguments are presented, any expression of a sequence can be completely evaluated by `eval`, no matter its value is used or not. Thus the expression `(proc (car items))` will be utterly evaluated in every call to `for-each`, which surely leads to the interaction above.

b. Evaluating `(p1 1)` and `(p2 1)` in the original `eval-sequence` involves the following interaction:

```
;;; L-Eval input:
(p1 1)
;;; L-Eval value:
(1 2)

;;; L-Eval input:
(p2 1)
;;; L-Eval value:
1
```

The value of `(p1 1)` remains the same whereas that of `(p2 1)` varies in Cy's proposed change to `eval-sequence`:

```
;;; L-Eval input:
(p1 1)
;;; L-Eval value:
(1 2)

;;; L-Eval input:
(p2 1)
;;; L-Eval value:
(1 2)
```

c. Because `proc` as well as the thunk contains `items` inside `(proc (car items))` is automatically forced by the lazy evaluator, no matter evaluated by `eval` or `actual-value`.

d. Personally, I prefer the approach in the text to deal with sequence in the lazy evaluator. As I mentioned in part a, so long as the values of its arguments are presented, any expression of a sequence can be completely evaluated by `eval`, no matter its value is used or not. Hence, it's excessive to exploit `actual-value` to evaluate expressions in the sequence. Besides, this modification can arise confusion in the presence of side effect, as the interaction in part c shows.