Exercise 3.27.

Memoization (also called *tabulation*) is a technique that enables a procedure to record, in a local table, values that have previously been computed. This technique can make a vast difference in the performance of a program. A memoized procedure maintains a table in which values of previous calls are stored using as keys the arguments that produced the values. When the memoized procedure is asked to compute a value, it first checks the table to see if the value is already there and, if so, just returns that value. Otherwise, it computes the new value in the ordinary way and stores this in the table. As an example of memoization, recall from section 1.2.2 the exponential process for computing Fibonacci numbers:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

The memorized version of the same procedure is

```
(define memo-fib
  (memorize (lambda (n)
              (cond ((= n 0) 0)
                    ((= n 1) 1)
                    (else (+ (memo-fib (- n 1))
                             (memo-fib (- n 2))))))))
```

where the memorizer is defined as

```
(define (memorize f)
  (let ((table (make-table)))
    (lambda (x)
      (let ((previously-computed-result (lookup x table)))
        (or previously-computed-result
            (let ((result (f x)))
              (insert! x result table)
              result))))))
```

Draw an environment diagram to analyze the computation of `(memo-fib 3)`. Explain why `memo-fib` computes the nth Fibonacci number in a number of steps proportional to n. Would the scheme still work if we had simply defined `memo-fib` to be `(memoize fib)`?

Answer.

We can see in the problem description that `memo-fib` is defined to be a procedure object by applying `memorize` to `memo-fib` itself in a recursive way. This indicates us to transform the application form `(memorize f)` in a more formal way, that is, interprete the `let` syntatic sugar into `lambda` expression:

```
(define (memorize f)
  ((lambda (table)
     (lambda (x)
       ((lambda (previously-computed-result)
          (or previously-computed-result
              ((lambda (result)
                 (insert! x result table)
                 result)
               (f x))))
        (lookup x table))))
   (make-table)))
```

We are now able to transform `memo-fib` in terms of table manipulation by substituting `f` with

```
(lambda (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (memo-fib (- n 1))
                 (memo-fib (- n 2))))))
```

in the body of `memorize` we obtained just now:

```
(define memo-fib
  ((lambda (table)
       (lambda (x)
         ((lambda (previously-computed-result)
            (or previously-computed-result
                ((lambda (result)
                   (insert! x result table)
                   result)
                 ((lambda (n)
                    (cond ((= n 0) 0)
                          ((= n 1) 1)
                          (else (+ (memo-fib (- n 1))
                                   (memo-fib (- n 2))))))
                  x))))
          (lookup x table))))
     (make-table)))
```

This can be further simplified into:

```
(define memo-fib
  (lambda (x)
    ((lambda (previously-computed-result)
       (or previously-computed-result
           ((lambda (result)
              (insert! x result table)
              result)
            ((lambda (n)
               (cond ((= n 0) 0)
                     ((= n 1) 1)
                     (else (+ (memo-fib (- n 1))
                              (memo-fib (- n 2))))))
             x))))
     (lookup x table))))
```

Therefore, in evaluating `(memo-fib 3)`, we first construct a procedure object of `memorize` with its enclosing environment set to the global environment. Notice that the body of `memorize` is an application rather than a procedure and it performs insertion on a local table according to the memorized value in that table. This was done by constructing an environment E1 in which, the formal paramenter of the procedure object `(memorize f)` was assigned to a local table. The local table was created by invoking the procedure `make-table` in another newly constructed environment E2.

Later on, another procedure object was created which has its enclosing environment E1 and whose body is the `lambda` that `(memorize f)` returns. It then was associated to `memo-fib` in the global environment. The computing of `(memo-fib 3)` was handle in this framework, figure 1 shows the environment structure in the computing process.

This environment model perfectly interprets the reason why `memo-fib` computes the nth Fibonacci number evolves only linear step of n: the `memo-fib` computes a value only if there was not such a value keyed by the given arguments in the local table and this newly obtained "`(arg . value)`" pair was later inserted into the local table. This in fact flattened a duplicated tree into a linear list with unique elements and avoid repeated computation.

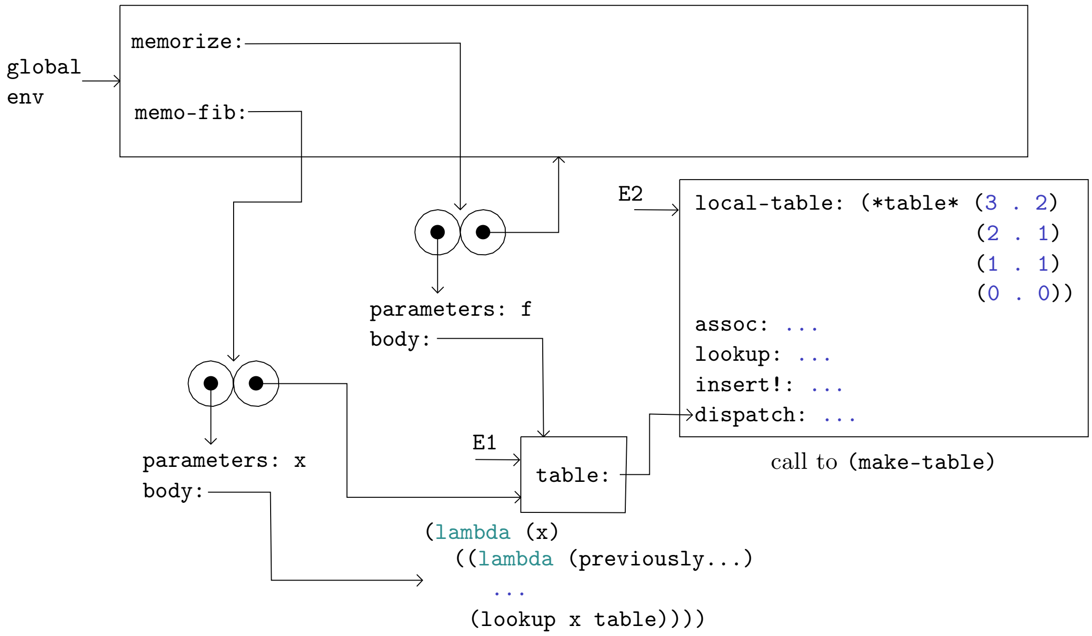

Figure 1. The environment structure in computing `(memo-fib 3)`.

Hence, in this strategy, computing the nth Fibonacci number requires only an increment of the computing $\text{Fib}\,(n-1)$ which is in case the increment of the computing of $\text{Fib}\,(n-2)$, that is

$$\begin{aligned} T\,(\text{Fib}\,(n)) &= T\,(\text{Fib}\,(n-1))+1 \\ &= T\,(\text{Fib}\,(n-2))+1+1 \\ &= T\,(\text{Fib}\,(n-3))+1+1+1 \\ &= \ldots \\ &= T\,(\text{Fib}\,(n-(n-1))+n-1 \\ &= T\,(\text{Fib}\,(0))+n \\ &= \Theta\,(n) \end{aligned}$$

which shows that `memo-fib` computes the nth Fibonacci number in a number of steps proportional to n.

On the other hand, the scheme would fail to accomplish the task in $\Theta\,(n)$ steps if we had simply defined `memo-fib` to be `(memoize fib)`. For this time, `memo-fib` would call `fib` instead of `memo-fib` to address the subproblem. This would not maintain anything, but the pair "`(n . fib(n))`" in the local table during the computation, all the result for smaller problems was lost, making the computation degenerate back to evolve an order of growth $\Theta\,(\text{Fib}\,(n))$.