Exercise 3.27

Problem statement

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 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 memoized version of the same procedure is

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

where the memoizer is defined as

```
(define (memoize 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 n^{th} 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)`?

Solution

Remember the tree for the recursive `fib` calls shown in Chapter 1 (see Figure 1. In it you can see that calling the vanilla `fib` function results in a lot of repeated calculations. This causes the

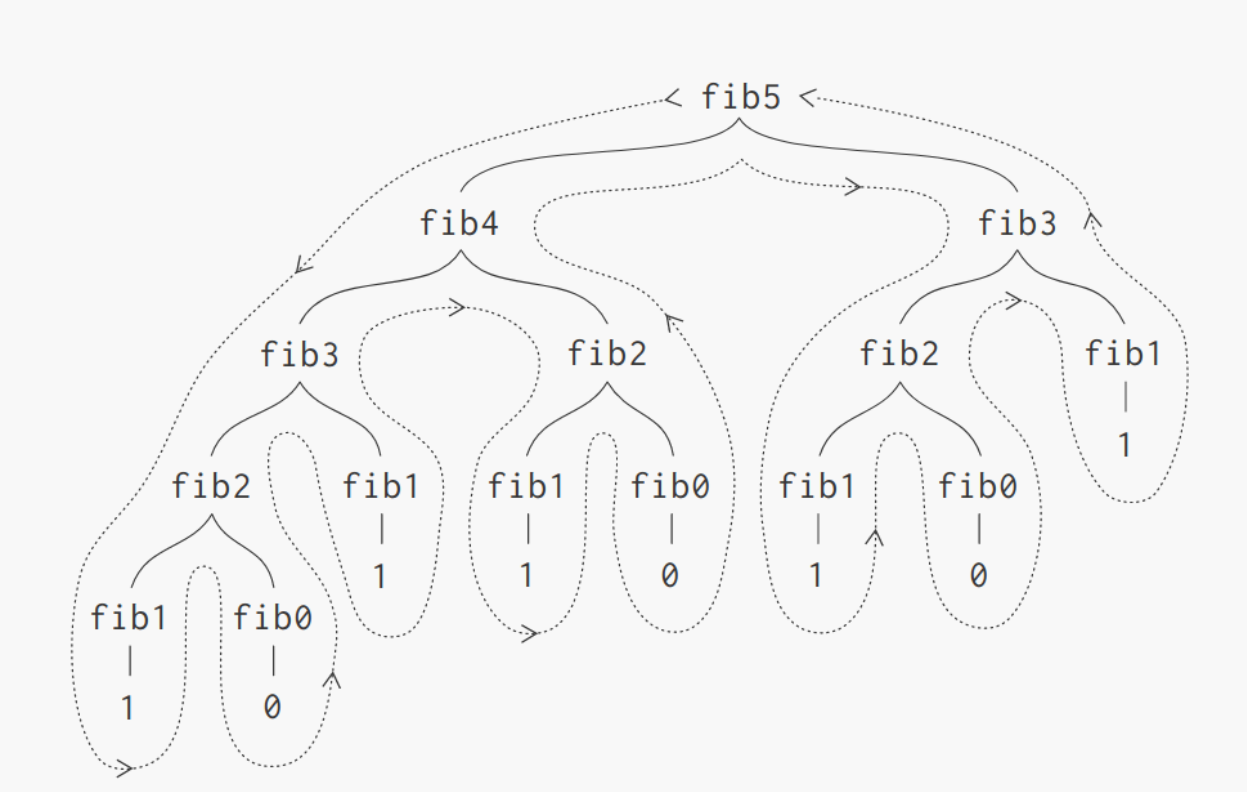

Figure 1: Recursive tree for calling `(fib 5)`. Observe the number of repeated operations. Running time for this procedure is $\simeq \varphi^N$

function to run quite slowly for large N.

The first thing we need to notice is that definining `memo-fib` creates an environment E1 where a table is initialized and the variable `memo-fib` if bound to a function $\lambda(x)$ that points to the E1 environment. So basically every time we call `memo-fib` and environment will be created under E1.

Figure 2 shows the frame structure for calling `(memo-fib 3)`. First an environment E2 is created under E1 where x takes the value 3. Since the values for $n-1$ and $n-2$ aren't found in the table yet, first the `memo-fib` function is called for $n-1=2$, but 2 is not found in the still empty table, so now `(memo-fib 1)` gets called, creates an environment E4 where x=1 which returns the value 1. 1 is inserted into the table. We go back up the recursive tree to the computation of `(memo-fib 2)` and realize that we still need to compute Fib(0), which adds 0 to the table. We can now calculate Fib(2) in `memo-fib`, add the value of 2 to the table. Going up the tree to the calculation of Fib(3) we notice that we need to compute Fib(1). However this value is already in the table and can be retrieved without calling `memo-fib` again.

The running time for the memoized solution is not exactly $O(N)$ in our case, but $O(N \log N)$, since the fastest lookup table we implemented so far is a binary tree, and lookups in a binary search tree are $O(\log N)$. Assuming we use a hash table with $O(1)$ lookups, the running time can be reduced down to $O(N)$.

This is the code used to check the running times in `ex_3.27.scm`.

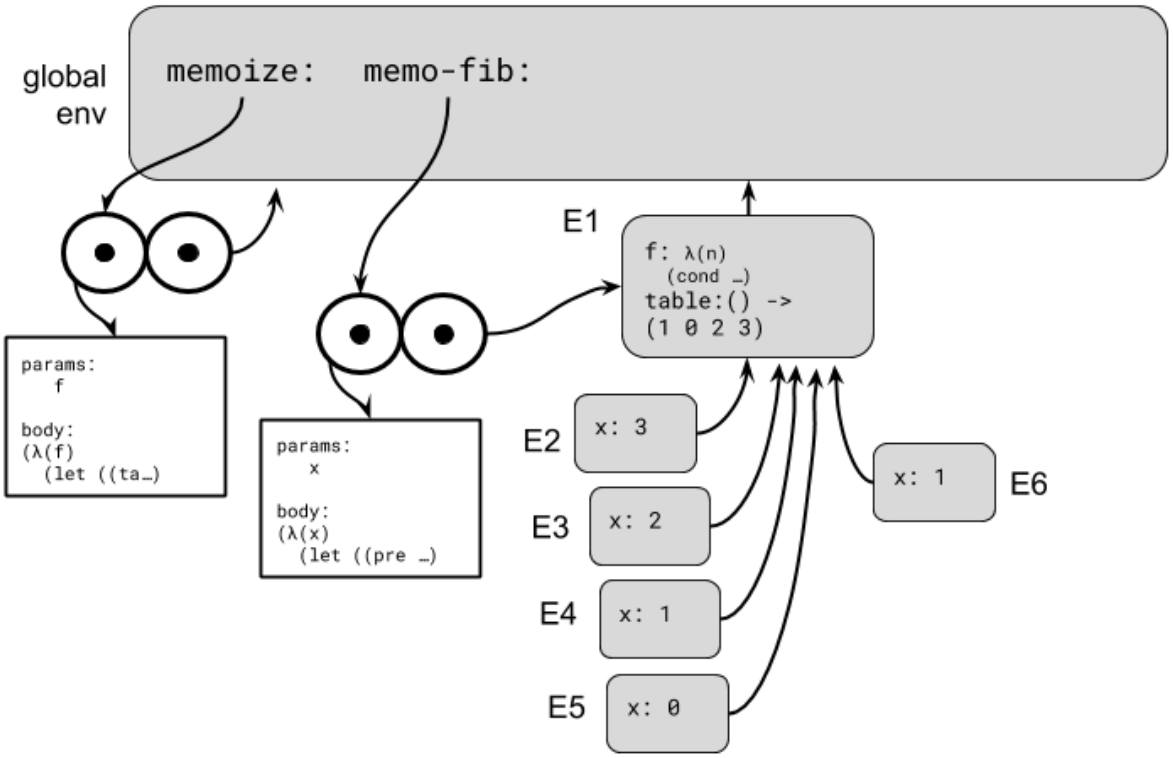

Figure 2: Frame structure when calling (`fib 3`). Running time for this procedure is (ideally) $O(N)$.

```
(define t0 (runtime))
(fib 40)
(display (list "time to run fib (vanilla)" (- (runtime) t0))) (newline)

(set! t0 (runtime))
(memo-fib 40)
(display (list "time to run fib with memo" (- (runtime) t0))) (newline)

(set! t0 (runtime))
((memoize fib) 40)
(display (list "time to run (memoize fib)" (- (runtime) t0))) (newline)
```

Running it on my laptop the results are

```
(time to run fib (vanilla) 2823518)
(time to run fib with memo 490)
(time to run (memoize fib) 2860769)
```

We can notice that the memoized version is impressively faster than the naive recursive implementation. But why is `(memoize fib)` just as slow as the vanilla implementation and why doesn't memoization help in this case? While both `fib` and `memo-fib` are recursive functions, `fib` points at the global environment, unlike `memo-fib` which points at E1. So when the recursive calls call `memo-fib` they will return functions that can be inserted into the lookup table in E1. Calls to `fib` however will just create environments pointing to the global env, hence we're effectively short circuiting the memoization of intermediate results.