

Exercise 4.29.

Exhibit a program that you would expect to run much more slowly without memoization than with memoization. Also, consider the following interaction, where the `id` procedure is defined as in exercise 4.27 and `count` starts at 0:

```
(define (square x)
  (* x x))

;;; L-Eval input:
(square (id 10))
;;; L-Eval value:
<response>

;;; L-Eval input:
count
;;; L-Eval value:
<response>
```

Give the responses both when the evaluator memoizes and when it does not.

Answer.

The following procedure for computing Fibonacci numbers could be a good example to demonstrate efficiency gained from memoization:

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


Without memorization, the evaluator computes Fibonacci numbers in a hard way and its performance is not so satisfactory.

```
;;; L-Eval input:
(fib 10)
;;; L-Eval value:
89
;;; Elapsed time:
.13

;;; L-Eval input:
(fib 16)
;;; L-Eval value:
1597
;;; Elapsed time:
3.2800000000000002

;;; L-Eval input:
(fib 20)
;;; L-Eval value:
10946
;;; Elapsed time:
27.48
```

Once equipped with memoization, computing Fibonacci numbers becomes less suffering and, the larger `n` grows the more time it saved.

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

```

;;; L-Eval input:
(fib 10)
;;; L-Eval value:
89
;;; Elapsed time:
.05000000000000071

;;; L-Eval input:
(fib 16)
;;; L-Eval value:
1597
;;; Elapsed time:
.7699999999999996

;;; L-Eval input:
(fib 20)
;;; L-Eval value:
10946
;;; Elapsed time:
5.260000000000002

```

This divergence in efficiency arises from their distinction in handling the result of evaluated arguments. The evaluator with memoization stores the values it forced from a thunk, so identical arguments are evaluated only once. For example, consider the interaction mentioned above:

```

(define (square x)
  (* x x))

;;; L-Eval input:
(square (id 10))
;;; L-Eval value:
100

;;; L-Eval input:
count
;;; L-Eval value:
1

```

With memoization, arguments in the body of `square` are computed only once and stored by the evaluator for further use. Hence, `count` increments only once.

By contrary, an evaluator without memoization would spend much more effort to do repeated evaluation of identical arguments. For example, when interact with the evaluator without memoization, we can see a little bit difference:

```

(define (square x)
  (* x x))

;;; L-Eval input:
(square (id 10))
;;; L-Eval value:
100

;;; L-Eval input:
count
;;; L-Eval value:
2

```

For those two identical argument `x` are evaluated once for their own, making the variable `count` increment twice.