Exercise 3.57.

How many additions are performed when we compute the nth Fibonacci number using the definition of `fibs` based on the `add-streams` procedure? Show that the number of additions would be exponentially greater if we had implemented `(delay <exp>)` simply as `(lambda () <exp>)`, without using the optimization provided by the `memo-proc` procedure described in section 3.5.1.[1]

Answer.

Using the definition of `fibs` based on the `add-streams` procedure, the evaluator merely fetches the value of $\mathrm{Fib}(n-1)$ and $\mathrm{Fib}(n-2)$ and adds them together to produce $\mathrm{Fib}(n)$. In other words, it requires only one addition to compute the nth Fibonacci number starting from $n-1$. Hence, it takes $n-1$ additions to compute the nth Fibonacci number using the definition of `fibs` based on the `add-streams` procedure.

However, if we had implemented `(delay <exp>)` simply as `(lambda () <exp>)`, the evaluator would lose all the value it previously computed. To compute $\mathrm{Fib}(n)$, we compute $\mathrm{Fib}(n-1)$ and $\mathrm{Fib}(n-2)$. To compute $\mathrm{Fib}(n-1)$, we compute $\mathrm{Fib}(n-2)$ and $\mathrm{Fib}(n-3)$. This generates a process of tree recursion, as the `fib` procedure we saw in section 1.2.2. More precisely, the number of addition is $\mathrm{Fib}(n+1)$ which is the closest integer to $\phi^{n+1}/\sqrt{5}$, where

$$\phi = \left(1+\sqrt{5}\right)/2 \approx 1.6180$$

Thus, the process uses a number of addition that grows exponentially with the input.

Email address: informlarry@gmail.com

1. This exercise shows how call-by-need is closely related to ordinary memoization as described in exercise 3.27. In that exercise, we used assignment to explicitly construct a local table. Our call-by-need stream optimization effectively constructs such a table automatically, storing values in the previously forced parts of the stream.