Homework 2 - Mutable Objects and Procedures with State

记忆化 memo.scm

本次的作业中你需要：

1. 实现通用的表格 `Table` 用于存储键值对（对 Mutation 的理解）
2. 利用 `Table` 实现对一元和二元纯函数的记忆化（对 Environment Model 的理解）

Table 的实现

由于当前版本的 racket/scheme 把不可变列表和可变列表区别开来了（见 Getting rid of `set-car!` and `set-cdr!`），与可变列表相关操作的名字发生了变化，比如：`list` 变为 `mlist`；`cons` 变为 `mcons`；`pair?` 变为 `mpair?`；`car` 变为 `mcar`；`set-cdr!` 变为 `set-mcdr!` 等等，详细参考 Mutable Pairs and Lists。

基于 `mlist` 你需要实现一个表格来存储键值对，其中构造函数与类型判断已经写好：

```
; Constructor
; make-table :: () -> Table
(define (make-table)
  (mlist 'Table))

; Predicate
; table? :: Table -> Bool
(define (table? table)
  (and (mpair? table)
       (eq? (mcar table) 'Table)))
```

你需要实现增加或修改键值对、查找键是否存在、根据键查找值三种操作，单次操作时间复杂度不做要求，不高于 $O(n)$ 即可。

以下是一些使用例子：

```
(define t (make-table))
(table-has-key? t 'x) ; #f
(table-put! t 'x 123)
(table-has-key? t 'x) ; #t
(table-get t 'x) ; 123
(table-put! t 'x 234)
(table-put! t 3 5)
(table-get t 'x) ; 234
(table-get t 3) ; 5
```

1. 增加或修改键值对

```
; table-put! :: (Table, a, b) -> Table
(define (table-put! table key value)
  'your-code-here)
```

2. 查找键是否存在

```
; table-has-key? :: (Table, a) -> Bool
(define (table-has-key? table key)
  'your-code-here)
```

3. 根据键查找值

```
; table-get :: (Table, a) -> b
(define (table-get table key)
  'your-code-here)
```

> **Tip**
>
> 可以使用 `equal?` 来判断两个键是否相同，`equal?` 和 `eq?`、`=`、`eqv?` 的区别看 这里。

记忆化一元纯函数

现在我们来看递归实现的斐波那契数列：

```
; fib :: Int -> Int
; Calculate the n-th term of the Fibonacci sequence (A000045) mod 19260817
(define (fib n)
  (if (< n 2)
      (remainder n mod)
      (remainder (+ (fib (- n 1)) (fib (- n 2))) mod)))
```

不难注意到时间复杂度是 $O(\mathrm{Fib}_n)$ 的，即使是 `(fib 100)` 也难以计算，其原因在于相同的调用会被重复计算数次。

因为函数 `fib` 是纯函数，即函数无副作用且对于相同的 `n` 函数的值相同，那么可以利用 `Table` 把计算过的函数值缓存起来来加速。

你需要通过 `Table` 来实现把一元纯函数记忆化的高阶函数 `memoize-1`：

```
; memoize-1 :: (a -> a) -> (a -> a)
(define (memoize-1 f)
  'your-code-here)
```

使用 `memoize-1` 把求 斐波那契数列 和 纳拉亚纳奶牛数列 转化为记忆化的函数：

```
(set! fib (memoize-1 fib))
(set! narayana (memoize-1 narayana))
```

此时 `(fib 100)` 就能很快得出结果。

> **Tip**
>
> 1. 为什么 `(set! fib (memoize-1 fib))` 可以而 `(define fib-memo (memoize-1 fib))` 不可以
> 2. 结合 Environment Model (Lecture 4 Slides P40) 理解

记忆化二元纯函数

你可以通过复制 `memoize-1` 的实现并修改一小部分代码来得到 `memoize-2`：

```
; memoize-2 :: ((a, a) -> a) -> ((a, a) -> a)
(define (memoize-2 f)
  'your-code-here)
```

使用 `memoize-2` 把求 组合数 和 第二类斯特林数 转化为记忆化的函数：

```
(set! pascal (memoize-2 pascal))
(set! stirling (memoize-2 stirling))
```

更通用的 memoize?

我们的 `memoize-2` 实现的并不优雅，对于 n 元纯函数我们需要复制黏贴更多代码，可以思考一下如何实现更通用的 `memoize` 函数（不做要求）。

参考资料

1. D. Friedman, M. Felleisen, *The little Schemer (4th ed.)*. MIT Press, 1996. [pdf]
2. H. Abelson, G. Sussman, *Structure and Interpretation of Computer Programs*. MIT Press, 1996. [pdf]