# lang racket #| Memoization. |

Let f' be the returned function, and assume ‘uf’ has no side-effects.
1. Given the same argument, f' returns the same value as ‘uf’ would.
2. The first time f' is called with any particular argument, it records the result
   in a hash table.
   Later, if called with that same argument it returns the recorded result instead of
calling ‘uf’ again. |

(define (caching uf)
  (define cache (make-hash))
  (λ (a)
    (unless (hash-has-key? cache a)
      (hash-set! cache a (uf a)))
    (hash-ref cache a)))

#| A few test cases. Decide carefully what the specification above says about these. |
(define (f n) (displayln 'hi) 'bye)
(define caching-f (caching f))
; Expect to see two printings, but third printing suppressed because of the cached value.
(map caching-f '(1 2 1))
; This makes a new instance of a caching version of ‘f’, so will print.
((caching f) 2)

#| Convince yourself by a memory model diagram that the following won't make ‘fibonacci,'
a fast version ‘fibonacci₀’. |

(define (fibonacci₀ n)
  (if (< n 2)
    1
    (+ (fibonacci₀ (- n 1)) (fibonacci₀ (- n 2)))))

; This will still recurse through the non-caching ‘fibonacci₀’.
; It will be faster when called twice “from the outside” with the same argument.
(define fibonacci, (caching fibonacci₀))

(fibonacci, 3) ; Diagram:
(define fib₀)
  (caching (λ (n)
    (if (< n 2)
      1
      (+ (fib₀ (- n 1)) (fib₀ (- n 2))))))

(fib₀ 3) ; Diagram (notice the second recursive call for (fib₀ 3) uses the cache):
(define-cache (make-hash))
(lambda (a)
  (unless (hash-has-key? cache a)
    (hash-set! cache a (uf a)))
  (hash-ref cache a))

(lambda (n)
  (if (< n 2)
      1
      (+ (fibbo (- n 1)) (fibbo (- n 2)))))

(time (fibbo 100))

(define-syntax-rule
  (define-cached (<f-id> <p-id>)
    <body-expr>)
  (define <f-id> (caching (lambda (<p-id>)
                                  <body-expr>)))))

(define-cached (fibonacci n)
  (if (< n 2)
      1
      (+ (fibonacci (- n 1)) (fibonacci (- n 2)))))