CSSE 304 Day 21

1. When we have a highly recursive procedure, we can perhaps speed it up by **memoizing** (caching) the previously-computed values). It's a classic space-vs-time trade-off.

2. In `fib-memo`, the `sofar` variable saves previously-computed values.
   a. For example, if the highest `n` for which we have called `fib-memo` so far is 4, the value of `sofar` is `((4 . 5) (3 . 3) (2 . 2) (1 . 1) (0 . 1))` and max has the value 4.
   b. On the slides,w e can see a dramatic difference between the time and space usage by the "normal" Fibonacci procedure and the memoized version. A15 has an exercise about abstracting the memoization process.

3. **Multiple return values.** I am placing the code on this page so you can annotate it if you wish.

```
(define list-average
  (letrec ([helper
           (lambda (L)
             (if (null? L)
                 (list 0 0) ; sum, length
                 (let ([return (helper (cdr L))])
                    (list (+ (car return) (car L))
                          (+ 1 (cadr return))))))])
    (lambda (L)
      (apply / (helper L)))))


(call-with-values   (lambda () (values 3 4))
                               cons)

(call-with-values  values
                   (lambda args args))


(call-with-values + list)


(call-with-values list list)


(define split
  (lambda (ls)
    (if (or (null? ls) (null? (cdr ls)))
        (values ls '())
        (call-with-values
          (lambda () (split (cddr ls)))
          (lambda (odds evens)
            (values (cons (car ls) odds)
                    (cons (cadr ls) evens)))))))


(split '(a b c d e f))


(list (split '(a b c d e f)))


(call-with-values (lambda () (split '(a b c d e f)))
                  list)
```

```
(define list-average
  (letrec ([helper (lambda (L)
                     (if (null? L)
                         (values 0 0) ; sum, length
                         (call-with-values
                             (lambda () (helper (cdr L)))
                             (lambda (sum len)
                               (values (+ sum (car L))
                                       (+ 1 len))))))])
    (lambda (L)
      (call-with-values (lambda ()(helper L))
                        /))))

(define-syntax with-values
  (syntax-rules ()
    [(_ expr consumer)
     (call-with-values (lambda () expr)
                       consumer)]))

(with-values (split '(a b c d e f)) list)

(define list-average
  (letrec ([helper
           (lambda (L)
             (if (null? L)
                 (values 0 0) ; sum, length
                 (with-values
                    (helper (cdr L))
                    (lambda (sum len)
                      (values (+ sum (car L))
                              (+ 1 len))))))])
    (lambda (L)
      (with-values (helper L) / ))))

(define-syntax mvlet
  (syntax-rules ()
    ((_ ((x ...) e0) e1 e2 ...)
     (with-values e0
       (lambda (x ...) e1 e2 ...)))))

(define list-average
  (letrec
    ([helper
       (lambda (L)
          (if (null? L)
              (values 0 0) ; sum, length
              (mvlet ((sum len)
                      (helper (cdr L)))
                (values (+ sum (car L))
                        (+ 1 len)))))])
    (lambda (L)
      (with-values (helper L) / ))))
```

4. Basic Scheme Control flow:
   a. What is the current expression to be evaluated?
   b. Once that is done, what remains to be done with the value of the current expression?
   c. Consider the evaluation of `(+ a 5)` in the process of evaluating `(- 4 (* b (+ a 5)))`.
   d. What remains to be done with the *value* of `(+ a 5)` ?
   e. Can we express that as a procedure?
   f. We can call that procedure the continuation of the `(+ a 5)` computation
   g. The process of Scheme evaluation can be expressed as
   h. Loop:
      i. Evaluate the current expression
      ii. Apply the current continuation to the result
   i. In A18, you will rewrite your interpreter in this style, which is known as continuation-passing style (CPS).
5. What is the continuation of `(< x 5)` in `(if (< x 5) (+ x 3) (* x 2))` ?

6. What is the continuation of `(+ x 3)` in `(if (< x 5) (+ x 3) (* x 2))` ?

7.
```
(define fact
  (lambda (n)
    (if (zero? n)
        1
        (* n (fact (- n 1))))))
```
   a. In the evaluation of `(fact 5)`, what is the continuation of the call to `(fact 2)`?
   b. We see here that continuation is not merely a syntactic notion. (lambda (v) v)In "normal language" interpreters, continuations are represented by stack frames.
8. In "normal language" interpreters, continuations are represented by stack frames.
   But we may (for various reasons) want to do "stackless" programming.
9. We pass an explicit continuation to each procedure call, in order to keep the code in tail-form.
10. Thus it is continuation-passing style (CPS)
11. When CPSing our code, we divide the set of procedures into two groups:
    a. Primitive procedures can be called without a continuation argument.
    b. Substantial procedures (I made up this name) expect a continuation argument.
12. By default, built-in procedures and non-recursive procedures will be considered primitive; recursive procedures substantial.
13. Sometimes it will be useful to write a substantial version of a procedure that would normally be primitive.
14. A procedure definition is in *tail form* if all calls to non-primitive procedures are in tail position. Usually *primitive* will mean "built into Scheme", To enhance practice with CPS, in some examples we will sometimes designate one of the built-in procedures as non-primitive.
15. In a tail-form expression
    a. all calls to substantial procedures are in tail position.
    b. I.e., any such call is the last thing to be done in the current procedure application.
16. Which expressions are in tail position in the following code segments?

    `(begin e1 e2 e3)`

    `(if e1 e2 e3)`

    `(cond [e1 e2] [e3 e4] … [else e])`

    `(let ([v1 e1] [v2 e2] …) e)`

    `(e1 e2 e3)` ; procedure application.

17. in `(lambda (x) e0 … en)`, the expression `en` is in tail position.
    a. `en` is not evaluated when the lambda expression is evaluated.
    b. It only gets evaluated when the procedure is applied.
    c. Then `en` is the last thing to be evaluated.