

Tail Recursion

Summary: How to make your recursive procedures run more quickly by taking advantage of a program design strategy called *tail recursion*.

Recursive Strategies

In writing recursive procedures, you may have noticed that there are two general strategies for dealing with the result of the recursive call: (1) you can just return the result; or (2) you can do something with the result. For example, the `member?` procedure just returns the result of the recursive call and the `factorial` and `add-to-all` procedures do something with the result (multiply the result by something and `cons` something to the result, respectively).

```
;;; STANDARD POSTCONDITIONS:
;;; Unless specified otherwise, procedures
;;; (1) do not modify their parameters
;;; (2) do not read any input
;;; (3) do not write any output

;;; Procedure:
;;; member?
;;; Parameters:
;;; val, a value
;;; values, a list of values
;;; Purpose:
;;; Determines if val is in values.
;;; Produces:
;;; mem?, a boolean value
;;; Preconditions:
;;; The parameters have the appropriate types.
;;; Postconditions:
;;; mem? is #t if there exists an element of values that
;;; is equal to val.
;;; mem? is #f otherwise.
(define member?
  (lambda (val values)
    (cond (
      ; Nothing is in the empty list
      ((null? values) #f)
      ; If val is the first element, it's in values.
      ((equal? val (car values)) #t)
      ; Otherwise, look through the rest of values.
      (else (member? (val (cdr values))))))))

;;; Procedure:
;;; factorial
;;; Parameters:
;;; n, a non-negative integer
;;; Purpose:
;;; Computes n! = 1*2*3*...*n
;;; Produces:
```

```

;;; result, a positive integer
;;; Preconditions:
;;; n is a non-negative integer [unchecked]
;;; Postconditions:
;;; result = 1*2*3*...*n
(define factorial
  (lambda (n)
    (if (<= n 0) 1
        (* n (factorial (- n 1))))))

;;; Procedure:
;;; add-to-all
;;; Parameters:
;;; value, a number
;;; values, A list of numbers
;;; Purpose:
;;; Creates a new list by adding value to each member of values.
;;; Produces:
;;; A new list of numbers with the desired characterisitic.
;;; Precondition:
;;; value is a number [unchecked]
;;; values is a list of numbers [unchecked]
;;; Postconditions:
;;; Just the standard ones
(define add-to-all
  (lambda (value values)
    (if (null? values) null
        (cons (+ value (car values))
              (add-to-all value (cdr values))))))

```

If you think about how you might simulate these by hand, you'll notice that it's harder to deal with the second type of procedures because you have to spend extra effort remembering "the stuff left to do after the recursive call is done".

Husk-and-Kernel, Revisited

In previous labs, we've seen several examples illustrating the idea of separating the recursive kernel of a procedure from a husk that performs the initial call. Sometimes we've done this in order to avoid redundant precondition tests, or to prevent the user from bypassing the precondition tests. In other cases, we saw that the recursion can be written more naturally if the recursive procedure has an additional argument, not supplied by the original caller.

We can use husk-and-kernel techniques to make some non-tail-recursive procedures tail recursive.

```

(define factorial
  (lambda (n)
    ; Define a kernel that "accumulates" the previous multiplications
    ; as a second parameter. Now we're computing n*(n-1)*...*(m+1)*m!
    (letrec ((kernel (lambda (m acc)
                      ; If we've run out of values, return what we've
                      ; accumulated.
                      (if (<= m 0) acc
                          ; Otherwise, multiply the current accumulated

```

```

; product by m and continue with the
; remaining values.
(kernel (- m 1) (* acc m))))))
(kernel n 1)))

```

If we want to use named let, we can be even more concise.

```

(define factorial
  (lambda (n)
    (let kernel ((m n)
                 (acc 1))
      (if (<= m 0) acc
          (kernel (- m 1) (* acc m))))))

```

Efficiency and Tail Recursion

There is yet another reason for adopting tail recursion, and it has to do with efficiency. An implementation of Scheme is required to perform *tail-call elimination* -- to implement procedure calls in such a way that, if the last step in procedure A is a call to procedure B (so that A will simply return to its caller whatever value is returned by B), the memory resources supporting the call to A can be freed and recycled as soon as the call to B has been started. To make this possible, the implementer arranges for B to return its value directly to A's caller, bypassing A entirely. In particular, this technique is required to work when A and B are the same procedure, invoking itself recursively (in which case the recursion is called *tail recursion*), and even if there are a number of recursive calls, each of which will return to its predecessor the value returned by its successor. In the implementation, each of the intermediate calls vanishes as soon as its successor is launched.

However, this clever technique, which speeds up procedure calling and sometimes enables Scheme to use memory very efficiently, is guaranteed to work *only* if the procedure call is the last step. For instance, tail-call elimination cannot be used in the `sum` procedure as we defined it in an earlier lab:

```

(define sum
  (lambda (ls)
    (if (null? ls)
        0
        (+ (car ls) (sum (cdr ls))))))

```

The recursive call in this case is not a tail call, since, after it returns its value, the first number on the list still has to be added to that value.

As you might expect, it is possible to write a tail-recursive version of `sum`, just as we wrote a tail-recursive factorial:

```

(define sum
  (letrec ((sum-kernel (lambda (ls running-total)
                        (if (null? ls)
                            running-total
                            (sum-kernel (cdr ls)
                                         (+ (car ls) running-total))))))
    (lambda (ls)
      (sum-kernel ls 0)))

```

The idea is to provide, in each recursive call, a second argument, giving the sum of all the list elements that have been encountered so far: the running total of the previously encountered elements. We call this second argument the *accumulator*.

When the end of the list is reached, the value of the accumulator (the running total) is returned; until then, each recursive call strips one element from the beginning of the list, adds it to the running total, and *finally* calls itself recursively with the shortened list and the augmented running total. The “finally” part is important: `sum-kernel` is tail-recursive.

Here is a summary of the execution of a call to this version of `sum`:

```
(sum (list 97 85 34 73 10))
--> (sum-kernel (list 97 85 34 73 10) 0)
--> (sum-kernel (list 85 34 73 10) 97)
--> (sum-kernel (list 34 73 10) 182)
--> (sum-kernel (list 73 10) 216)
--> (sum-kernel (list 10) 289)
--> (sum-kernel null 299)
--> 299
```

Note that the additions are performed on the way into the successive calls to `sum-kernel`, so that when the base case is reached no further calculation is needed -- the value of the second argument in that last call to `sum-kernel` is returned without further modification as the value of the original call to `sum`.

Tail Recursion and List-Building Procedures

Tail recursion becomes a little bit more subtle when you think about how you would use tail recursion to build lists. Consider the `add-to-all` procedure from the start of this section. The straightforward way to make this procedure tail-recursive is to add a list of “things that have been added to” as a parameter to the kernel. When we’re done, we just return that list.

```
(define add-to-all
  (lambda (value values)
    ; kernel is a procedure of two parameters:
    ;   values-remaining, the values left to process
    ;   values-processed, a list of all values that have
    ;   already been added to.
    ; Initially, all values remain to be processed and
    ; no values have been added to.
    (let kernel ((values-remaining values)
                 (values-processed null))
      ; If there are no values remaining to process, we can
      ; use the values already processed.
      (if (null? values-remaining) values-processed
          ; Otherwise, process the first remaining value
          ; and then process any other remaining values.
          (kernel (cdr values-remaining)
                  (append values-processed
                          (list (+ value (car values-remaining))))))))))
```

As you may have noted from the code, it's a little bit complicated to extend that list. In particular, we have to add the next value to the *end* of the list of processed values. Right now, we only know two ways to add to the end of a list: (1) append another list (in this case a list containing just one value) or (2) reverse, cons, and reverse again. Both strategies are somewhat inefficient because we have to step all the way through the list to add to the end. Since we keep adding to the end, we step through the result list again and again and again and again.

Is there a better way to write `add-to-all`? Yes. We can consider other ways we know to add values to a list. Rather than appending, we can just cons each value on to the processed list.

```
(kernel (cdr values-remaining)
        (cons (+ value (car values-remaining))
              values-processed))))))
```

Now we have to figure out what to return in the base case. Since we're stepping through one list from left-to-right and adding them to the result list from right-to-left (that is, extending the list at the left each time), the result is the *reverse* of the result we want. Hence, in the base case, we need to reverse it again.

```
(define add-to-all
  (lambda (value values)
    ; kernel is a procedure of two parameters:
    ;   values-remaining, the values left to process
    ;   values-processed, a list of all values that have
    ;     already been added to; given in reverse order.
    ; Initially, all values remain to be processed and
    ; no values have been added to.
    (let kernel ((values-remaining values)
                (values-processed null))
      ; If we've run out of values to process, we can return
      ; the values already processed after putting them back
      ; in the correct order.
      (if (null? values-remaining) (reverse values-processed)
          ; Otherwise, process the first remaining value
          ; and then process any other remaining values.
          (kernel (cdr values-remaining)
                  (cons (+ value (car values-remaining))
                        values-processed))))))
```