

## Introduction

A procedure is *tail recursive* (also known as *iterative*) if and only if the only recursive calls in the procedure are **outermost** calls in expressions that directly return a value from the procedure. For example, the following code is a tail-recursive version of `factorial`:

```
(define fact-1
  (lambda (n)
    (fact-iter n 1)))

(define fact-iter
  (lambda (n result)
    (if (= n 0)
        result
        (fact-iter (- n 1)
                    (* n result)))))
```

This procedure is tail recursive, because `fact-iter`'s only recursive calls return values from the procedure directly: no additional work needs to be done with the result of the recursive call.

In contrast, the standard definition of `factorial` is **not** tail recursive:

```
(define fact-2
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact-2 (- n 1))))))
```

This definition is not tail recursive, because the recursive call `(fact-2 (- n 1))` is buried inside the larger expression `(* n (fact-2 (- n 1)))`. As a result, when the recursive call `(fact-2 (- n 1))` returns its value, there is still more work to be done (namely, multiplying that value by `n`).

The difference between the two versions becomes clearer if you compare traces of their recursive calls. For example, tell DrScheme to `trace` them:

```
> (trace fact-1 fact-iter fact-2)
(fact-1 fact-iter fact-2)
```

You should observe the following behavior:

```
> (fact-1 4)
|(fact-1 4)
|(fact-iter 4 1)
|(fact-iter 3 4)
|(fact-iter 2 12)
|(fact-iter 1 24)
|(fact-iter 0 24)
|24
24

> (fact-2 4)
|(fact-2 4)
| (fact-2 3)
| |(fact-2 2)
| |(fact-2 1)
| |(fact-2 0)
| |1
| |1
| |2
| 6
|24
24
```

Notice that, in a tail-recursive procedure such as `fact-1`, **all** the work gets done in changing the values of its parameter(s): when each recursive call finishes, its value can be immediately returned to the previous caller. In a linear-recursive procedure such as `fact-2`, some work also gets done **after** the recursive call comes back. As a result, linear-recursive procedures typically require more memory to run: the run-time system needs to remember how to resume the computation once it's received the result of the recursive call.

Tail-recursive procedures—although more limited than linear-recursive procedures—are nice because they are really equivalent to `while` loops and the interpreter can take advantage of this fact to run them very efficiently. The procedure `fact-iter` is roughly equivalent to the following pseudo-code:

```
proc(n,result)
begin
  while (n!=0) do
    result := n * result;
    n      := n - 1;
  endwhile;
  return result;
end;
```

Note that the variables being manipulated in the while loop (`result` and `n`) correspond exactly to the arguments that `fact-iter` passes on each recursive call. At any given point, the current value of `result` corresponds to the value computed *so far*, while the value of `n` serves as the stopping condition.

## §1. Tail Recursion on Lists

Tail-recursive procedures (especially those that operate on lists) sometimes cause problems when you want to construct an answer from the bottom up, as in the following example. Consider the procedure `remove` from the textbook (page 19):

```
(define remove
  (lambda (s los)
    (if (null? los)
        '()
        (if (eqv? (car los) s)
            (remove s (cdr los))
            (cons (car los)
                  (remove s (cdr los)))))))
```

A first attempt as a tail-recursive version of this might look as follows:

```
(define remove
  (lambda (s los)
    (remove-iter s los '())))

(define remove-iter
  (lambda (s los result)
    (if (null? los)
        result
        (if (eq? (car los) s)
            (remove-iter s (cdr los) result)
            (remove-iter s
                        (cdr los)
                        (cons (car los) result))))))
```

This version almost works, except that the list that comes back is the reverse of what we want. So, a standard patch is to change `remove` to:

```
(define remove
  (lambda (s los)
    (reverse (remove-iter s los '()))))
```

Note that the only change here is the addition of `reverse` to the procedure `remove`, to un-reverse the reversed list: the procedure `removeIter` still remains tail recursive.

**Comment on efficiency** You might be wondering why the use of `reverse` is necessary at all: another solution would be to replace

```
(remove-iter s (cdr los) (cons (car los) result))
```

with

```
(remove-iter s (cdr los) (append result (list (car los)))) .
```

However, `append` is much more expensive than `cons`: `append`'s cost is proportional to the length of its first argument, whereas `cons` has constant time.

## §2. Parameters to Tail-Recursive Procedures

When writing a tail-recursive procedure, it often helps to classify your parameters into one of three categories:

1. Those that represent the *partial result computed so far* (e.g., `result` in all the above procedures)
2. Those that represent the *computation left to be done* (e.g., `n` in `fact-1` or `los` in `removeIter`)
3. Those that provide additional information necessary for defining the computation to be done (e.g., `item` in `remove`)

The parameters of category (2) provide the stopping conditions for the recursive calls; the parameters of category (1) provide the final result to send back, once the stopping condition has been met.

### Your problems

For each of these questions, you need to give a *tail-recursive* procedure: *you won't receive credit for anything other than a tail-recursive procedure*. Also, do not use `append` in cases where `cons` is more efficient.

If you are not sure whether your answer is tail recursive, try tracing its behavior: do values from recursive calls get returned immediately, or is more work done to them? You can also ask one of us to verify.

1. Write a **tail-recursive** version of the following linear-recursive procedure.

```
(define count-occ
  (lambda (m lon)
    (if (null? lon)
        0
        (+ (if (= m (car lon)) 1 0) (count-occ m (cdr lon)))
    )))
```

2. Consider the following procedure `downto`, which takes two integers `m` and `n` and returns the descending list from `m` down to `n`, inclusive:

```
(define downto
  (lambda (m n)
    (if (>= m n)
        (cons m (downto (- m 1) n))
        '())
    )))
```

For example, `(downto 5 2)` returns `(5 4 3 2)`, while `(downto 7 8)` returns `()`.

Give a **tail-recursive** version of this procedure.

3. Consider the function `fib` given by the equation.

$$fib(n) = \begin{cases} 1, & \text{if } n \leq 1; \\ fib(n-1) + fib(n-2), & \text{if } n > 1. \end{cases}$$

A C/Pascal way of computing this might be as follows:

```
proc fib(n)
begin
  if (n<2) then return 1;

  fib_n_1 := 1;
  fib_n_2 := 1;
  k := 1;

  while (k<n) do begin
    t1 := fib_n_1;
    t2 := fib_n_2;
    fib_n_1 := t1 + t2;
    fib_n_2 := t1;
    k := k+1;
  end;
  return fib_n_1;
end;
```

Write a **tail-recursive** version of `fib` in Scheme.

**What to Hand In** Hand in your source code (*including contracts and purposes for each procedure you're asked to write*), along with verified test cases that demonstrate that the code works as required.

You **do not** have to submit anything electronically for this lab.

**Due:** This lab is due by noon on Friday, February 8. You should submit it to the labeled bin in CST 3-212.