Factorial examples with Commentary – CSSE 304 Day 2 – Claude Anderson

Fall, 2017: I did this example in section 01, but I ran out of class time before doing it in Section 02. I will present it here with a substantial written commentary, which may be helpful for students in any section. It's hard to believe that it took two pages to write what took less than 5 minutes in class, but it did!

I also wanted to illustrate the use of `cond`, which behaves like nested ifs, but has a much simpler form. Each clause consists of a test, and something to do if that test is true. The first clause whose test is true is the only one that gets executed the optional (but always a good idea) `else` clause is placed at the end and its action is always executed, if none of the previous ones were executed.
To enhance readability, I use square brackets to surround each `cond` clause.

```
(define fact    ; factorial function
  (lambda (n)
    (cond
       [(zero? n) 1]
       [(negative? n) "Error"]
       [else (* n (fact (- n 1)))])))
```

Line-by-line commentary:

Lines 1 and 2 : Standard way of defining and naming a function.
Line 3: the `cond` syntax. This `cond` has three clauses.
Line 4 : first clause. If n is zero, we return 1 – this is zero factorial.
Line 5 : prevent infinite loop if `fact` is called with a negative argument. For simplicity, I simply return a string. There are better ways of handling errors, some of which we will see later.
Line 6: Do a recursive call to compute (n-1)!, then multiply the result by n to get n!.

```
> (fact 6)
720
> (fact -3)
"Error"
> (trace fact)
(fact)
> (fact 4)
|(fact 4)
| (fact 3)
| |(fact 2)
| | (fact 1)
| | |(fact 0)
| | |1
| | 1
| |2
| 6
|24
24
```

You can see where each recursive call happens, the result of each and the result of multiplying each result by that level's "n" value.

Something different: Next we try a different approach to factorial, one that is more like what you learned in middle school; multiply the members as we go along. To do that, we use an *accumulator* argument, `acc`. I also skip the "test for negative" step in order to make it simpler.

```
(define fact2
  (lambda (n acc)
    (if (zero? n)
         acc
        (fact2 (- n 1) (* n acc)))))
```

When we get down to n=0, the accumulator has accumulated the factorial of the original n. And notice that in `fact2`, we do the multiplication *on the way into* each recursive call, instead of *after* the recursive call, as we did in `fact`. **Things look very different when we trace `fact2`:**

```
> (trace fact2)
(fact2)
> (fact2 4 1)
|(fact2 4 1)
|(fact2 3 4)
|(fact2 2 12)
|(fact2 1 24)
|(fact2 0 24)
|24
24
```

There is no indentation! Why is this? In `fact`, whenever there is a return from a recursive call, there is more to be done, namely the multiplication. In `fact2`, when a recursive call is finished, there is nothing else to do. In `fact`, Scheme has to remember all of those values of n that are waiting to be multiplied (hence the need for stack frames). In `fact2`, there is no need to remember anything like that, so Scheme can and does re-use the same stack frame.

This "making the recursive call be the last thing to be done is called "tail recursion". Re-using the stack frame instead of making new ones essentially turns recursive calls into loops. Scheme does this when you use tail recursion. And that is good!