# 211lab05

## Notes on Tail Recursion

When a program makes a function call, the code has to remember the location from where the call was made in order to return to it after the function finishes its work. The calling program also needs to know where to retrieve the result of the computation performed by the function. Behind the scene, the compiler sets up a stack of "call frames." Each call frame keeps track of the return address, the arguments for the function call and the result of the function's computation. Creating a call frame, pushing it on the call stack and popping it off the call stack are relatively costly operations in execution time. In addition, each call frame takes space in the portion of memory allocated for program control.

When a recursive function creates a deep call stack, it is costly both in time and space. Really deep recursion can exhaust the space available for program control (which is usually a fixed partition of memory). Fortunately, there is a special form of recursion called tail recursion that can be converted into an iterative loop by smart compilers, eliminating the creation of multiples call frames and thus greatly improving performance. A recursive function is said to be tail recursive if all of the recursive calls in the function body occur in tail positions. A position in the function body is a tail position if it is the last operation performed in any execution of the body that reaches it. In other words, after returning from a recursive call in tail position there is no pending operation waiting to be performed.

Consider the following function that counts the number of elements in a list.

```
;; how-many1: list of X -> number
;; (how-many lox) returns the number of elements in lox.
;; examples
(check-expect (how-many1 empty) 0)
(check-expect (how-many1 '(a)) 1)
(check-expect (how-many1 '(a b c)) 3)
;; code
(define (how-many1 lox)
  (cond
    [(empty? lox) 0]
    [(cons? lox) (+ (how-many1 (rest lox)) 1)]))
```

The function how-many1 is not tail recursive. Why?

After the recursive call (how-many1 (rest lox)), the calling function still needs to perform an addition before returning the result.

Now consider the following function.

```
;; how-many-acc: list-of-X number -> number
;; (how-many-acc lox a) returns a if lox is empty otherwise it returns
;; a + the number of elements in (rest lox).
;; examples
(check-expect (how-many-acc empty 5) 5)
(check-expect (how-many-acc '(a) 0) 1)
(check-expect (how-many-acc '(a b c) 0) 3)
;; code
(define (how-many-acc lox acc)
  (cond
    [(empty? lox) acc]
    [(cons? lox) (how-many-acc (rest lox) (+ acc 1))]))
```

The function how-many-acc is tail recursive. Why?

The function how-many-acc returns immediately to the caller after making its recursive call (how-many-acc (rest lox) (+ acc 1)), without having to perform any additional computation. The function how-many-acc updates its argument acc (by adding 1 to it) efore passing it on to the recursive call. By the time the recursive call reaches the end of the list, marked by empty, it simply returns the argument acc. The function how-many-acc uses its acc parameter to "accumulate" the result of its computation. As such the parameter acc is called an accumulator.

We now can make use of how-many-acc to write a highly efficient function to compute the number of elements in a list.

```
;; how-many2: list-of-X -> number
;; (how-many2 lox) returns the number of elements in lox.
;; examples
(check-expect (how-many2 empty) 0)
(check-expect (how-many2 '(a)) 1)
(check-expect (how-many2 '(a b c)) 3)
;; code
(define (how-many2 lox)
  (how-many-acc lox 0))
```

Note that how-many2 is not tail recursive. It simply calls a (helper) recursive function to do the job.

Aside: Many proofs of theorems in Mathematics are carried out by first proving a series of lemmas, which are put together to prove the theorem. Think of the helper functions as "lemmas" for the calling function, the "theorem."

# Accumulators

Accumulators are closely linked with tail recursion because conversion of a simple structural recursion to tail recursive form typically involves introducing a help function with an accumulator.

On the other hand, many important uses of accumulators do not involve tail recursion. Accumulators are useful in solving many problems that do not fit the tail-recursive model.

# Exercises

1. Write a function called prod-nums that takes a list of numbers and returns the product of the number in the list, using a tail recursive helper function.

What should the function return on an empty list?

2. Revise the preceding definition to terminate immediately if it encounters a 0 in the list of numbers being multiplied.

3. Write a function called last-elt that takes in a list of X and returns the last element in the list, using a tail recursive helper function. For an empty list, call (error 'last-elt "applied to empty list").

4. Write a function called find-min that takes a list of numbers and returns the smallest number in the list, using a tail recursive helper. For an empty list, call (error 'last-elt "applied to empty list").

5. Write a function called make-palindrome that takes a list and returns a list consisting of the input list and its mirror around the last element, using a (non tail-recursive) helper with an accumulator. For example, (make-palindrome '(a b c)) returns '(a b c b a).

## !! Access Permissions

- Set ALLOWTOPICCHANGE = Main.TeachersComp211Group