

CMSC 330: Organization of Programming Languages

Tail Recursion

CMSC330 Spring 2025

Factorial

$$\text{fact } n = \begin{cases} 1 & n=0 \\ n * \text{fact } (n-1) & n>0 \end{cases}$$

```
let rec fact n =  
    if n = 0 then 1  
    else n * fact (n-1)
```

```
fact 4 = 24
```

Factorial

$$\text{fact } n = \begin{cases} 1 & n=0 \\ n * \text{fact } (n-1) & n>0 \end{cases}$$

fact 3 = 3 * fact 2
= 3 * 2 * fact 1
= 3 * 2 * 1 * fact 0
= 3 * 2 * 1 * 1
= 3 * 2 * 1
= 3 * 2
= 6

fact 0

fact 1

fact 2

fact 3

Stack

	1
1	1 * fact 0
2	2 * fact 1
3	3 * fact 2

Stack Overflow


```
# let rec fact n = if n = 0 then 1 else n * fact (n-1);;  
val fact : int -> int = <fun>  
# fact 1000000 ;
```

Stack overflow during evaluation (looping recursion?).

let rec length lst =
 match lst with
 | [] -> 0
 | _ :: t -> 1 + length t ;

adds stack frames
for each iteration.

Doesn't happen in
C, Java, b/c use tail
loops.



Yet Another Factorial

$$\text{aux } x \ a = \begin{cases} a & x=0 \\ \text{aux } (x-1) \ x*a & x>0 \end{cases}$$

`fact n = aux n 1`

```
let fact n =  
  let rec aux x a =  
    if x = 0 then a  
    else aux (x-1) x*a  
  in  
  aux n 1
```

		Stack	
fact 3			6
	1,6		aux 1 6
	2,3		aux 2 3
	3,1		aux 3 1

Yet Another Factorial

$\text{aux } x \ a = \begin{cases} a & x=0 \\ \text{aux } \underbrace{(x-1)} & x>0 \end{cases}$	
$\text{fact } n = \text{aux } n \ 1$	

$\begin{aligned} \text{fact } 3 &= \text{aux } 3 \ 1 \\ &= \text{aux } 2 \ 3 \\ &= \text{aux } 1 \ 6 \\ &= 6 \end{aligned}$	<p>One call per value.</p> <p>no extra processing, no need for extra stack frame.</p>
---	---

Tail Recursion

- Whenever a function's result is **completely computed by its recursive call**, it is called **tail recursive**
 - Its “tail” – the last thing it does – is recursive
- Tail recursive functions can be implemented **without requiring a stack frame for each call**
 - **No intermediate variables need to be saved**, so the compiler overwrites them
- Typical pattern is to use an **accumulator** to build up the result, and return it in the base case

Compare fact and aux

```
let rec fact n =  
  if n = 0 then 1  
  else n * fact (n-1)
```

Waits for recursive call's result to compute final result

```
let fact n =  
  let rec aux x acc =  
    if x = 1 then acc  
    else aux (x-1) (acc*x)  
  in  
  aux n 1
```

final result is the result of the recursive call

Exercise: Finish Tail-recursive Version

```
let rec sumlist l =  
  match l with  
    [] -> 0  
  | (x::xs) -> (sumlist xs) + x
```

Tail-recursive version:

```
let sumlist l =  
  let rec helper l a =  
    match l with  
      [] -> a  
    | (x::xs) -> helper xs (x+a)  
  in  
  helper l 0
```

Quiz #1

True/false: `map` is tail-recursive.

```
let rec map f = function
  [] -> []
| (h::t) -> (f h) :: (map f t)
```

- A. True
- B. False

Quiz #1

True/false: `map` is tail-recursive.

```
let rec map f = function
  [] -> []
| (h::t) -> (f h) :: (map f t)
```

A. True

B. False

Quiz #2

True/false: `fold` is tail-recursive

```
let rec fold f a = function
  [] -> a
| (h::t) -> fold f (f a h) t
```

- A. True
- B. False

Quiz #2

True/false: **fold** is tail-recursive

```
let rec fold f a = function
  [] -> a
| (h::t) -> fold f (f a h) t
```

A. True

B. False

Quiz #3

True/false: `fold_right` is tail-recursive

```
let rec fold_right f l a =  
  match l with  
  | [] -> a  
  | (h::t) -> f h (fold_right f t a)
```

- A. True
- B. False

Quiz #3

True/false: `fold_right` is tail-recursive

```
let rec fold_right f l a =  
  match l with  
  | [] -> a  
  | (h::t) -> f h (fold_right f t a)
```

A. True

B. False

Quiz #4

True/false: this is a tail-recursive `map`

```
let map f l =  
  let rec helper l a =  
    match l with  
    [] -> a  
    | h::t -> helper t ((f h)::a)  
  in helper l []
```

- A. True
- B. False

Quiz #4

True/false: this is a tail-recursive `map`

```
let map f l =  
  let rec helper l a =  
    match l with  
    [] -> a  
    | h::t -> helper t ((f h)::a)  
  in helper l []
```

A. True

B. False (elements are reversed)

A Tail Recursive `map`

```
let map f l =  
  let rec helper l a =  
    match l with  
    [] -> a  
    | h::t -> helper t ((f h)::a)  
  in rev (helper l [])
```

Could instead change `(f h)::a` to be `a@(f h)`

Q: Why is the above implementation a better choice?

A: $O(n)$ running time, not $O(n^2)$ (where n is length of list)