CMSC 330: Organization of Programming Languages

Tail Recursion
Facorial

\[
\text{fact } n = \begin{cases} 
  1 & n = 0 \\
  n \times \text{fact} \ (n-1) & n > 0
\end{cases}
\]

\[
\text{let rec } \text{fact} \ n = \\
\quad \text{if } n = 0 \text{ then } 1 \\
\quad \text{else } n \times \text{fact} \ (n-1)
\]
Factorial

\[ \text{fact } n = \begin{cases} 
1 & \text{n=0} \\
 n \times \text{fact } (n-1) & \text{n>0}
\end{cases} \]

\[ \text{fact } 3 = 3 \times \text{fact } 2 \]
\[ = 3 \times 2 \times \text{fact } 1 \]
\[ = 3 \times 2 \times 1 \times \text{fact } 0 \]
\[ = 3 \times 2 \times 1 \times 1 \]
\[ = 3 \times 2 \times 1 \]
\[ = 3 \times 2 \]
\[ = 6 \]

Stack

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>fact 0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>fact 1</td>
<td>1 * fact 0</td>
<td></td>
</tr>
<tr>
<td>fact 2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>fact 3</td>
<td>3 * fact 2</td>
<td></td>
</tr>
</tbody>
</table>
# 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?).
Yet Another Factorial

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

\[
\text{fact } n = \text{aux } n \ 1
\]

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

---

**Look, Ma! No Stack!**

No need to push a new frame on each call

- The result of the evaluation is exactly the result of the recursive call – nothing to remember

- So: Reuse the current frame

\[
\text{fact } 3 = \text{aux } 3 \ 1 \\
= \text{aux } (3-1) \ 3*1 = \text{aux } 2 \ 3 \\
= \text{aux } (2-1) \ 2*3 = \text{aux } 1 \ 6 \\
= \text{aux } (1-1) \ 1*6 = \text{aux } 0 \ 6 \\
= 6
\]
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
Comparison of 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
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

```ocaml
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

```ml
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
True/false: \texttt{fold\_right} is tail-recursive

\begin{verbatim}
let rec fold_right f l a =
  match l with
  [] -> a
| (h::t) -> f h (fold_right f t a)
\end{verbatim}

A. True  
B. False
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
        [] -> _____
        | (x::xs) -> ________ in
    helper l 0
Exercise: Finish Tail-recursive Version

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

*Tail-recursive version:*

```
let rec helper l a =
  match l with
  | [] -> a
  | (x::xs) -> helper xs (x+a) in
helper l 0
```
Tail Recursion Pattern (1 argument)

let func x =
  let rec helper arg acc =
    if (base case) then acc
    else
      let arg’ = (argument to recursive call)
      let acc’ = (updated accumulator)
      helper arg’ acc’ in (* end of helper fun *)
  helper x (initial val of accumulator -- used for base case)
Tail Recursion Pattern with fact

let fact x =
  let rec helper arg acc =
    if arg = 0 then acc
    else
      let arg' = arg – 1 in
      let acc' = acc * arg in
      helper arg' acc' in (* end of helper fun *)
  helper x 1
;;
Tail Recursion Pattern with \texttt{rev}

\begin{verbatim}
let rev x =
    let rec rev_helper arg acc =
        match arg with
            [] -> acc
        | h::t ->
            let arg' = t in
            let acc' = h::acc in
            rev_helper arg' acc' in (* end of helper fun *)
    rev_helper x []
;;
\end{verbatim}

\textit{Can generalize to more than one argument, and multiple cases for each recursive call}
True/false: this is a tail-recursive map

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

```ocaml
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)
WHY DO YOU LIKE FUNCTIONAL
PROGRAMMING SO MUCH? WHAT
DOES IT ACTUALLY GET YOU?
TAIL RECURSION IS
ITS OWN REWARD.
Tail Recursion is Important

• Pushing a call frame for each recursive call when operating on a list is dangerous
  – One stack frame for each list element
  – Big list = stack overflow!

• So: favor tail recursion when inputs could be large (i.e., recursion could be deep). E.g.,
  – Prefer `List.fold_left` to `List.fold_right`
    • Library documentation should indicate tail recursion, or not
  – Convert recursive functions to be tail recursive
Outlook: Tail Recursion is General, too

• A function that is tail-recursive returns at most once (to its caller) when completely finished
  – The final result is exactly the result of a recursive call; no stack frame needed to remember the current call

• Is it possible to convert an arbitrary program into an equivalent one, except where no call ever returns?
  – Yes. This is called continuation-passing style
  – More later!