CMSC 330: Organization of Programming Languages

Tail Recursion
Factorial

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

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

;;

fact 4 = 24
Factorial

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

\texttt{fact 1000000?}

\begin{verbatim}
# 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?).
\end{verbatim}
Yet Another Factorial

\[
\text{fact } n = \text{ aux } n \ 1 \\
\text{aux } 1 \ \text{acc} = \text{acc} \\
\text{aux } n \ a = \text{aux } (n-1) \ (n*\text{acc})
\]

\[
\text{fact } 3 = \text{ aux } 3 \ 1 \\
\quad \text{aux } 2 \ 3 \\
\quad \quad \text{aux } 1 \ 6 \\
\quad \quad \quad 6
\]
Yet Another Factorial

\[
\text{fact } n = \text{aux } n 1 \\
\text{aux } 1 \text{ acc } = \text{acc} \\
\text{aux } n \ a = \text{aux } (n-1) (n*\text{acc})
\]

\[
\text{fact } 3 = \text{aux } 3 1 \\
\text{aux } 2 3 \\
\text{aux } 1 6 \\
6
\]
YAF: Yet Another Factorial

fact n = aux n 1
aux 1 acc = acc
aux n a = aux (n-1) (n*acc)

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

• Whenever a function ends with a recursive call, it is called *tail recursive*
  – Its “tail” 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 helper

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

*Waits for recursive call’s result to compute final result*

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

**Exercise: Finish Tail-recursive Version**

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
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.

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

A. True
B. False
True/false: `fold_left` is tail-recursive

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

A. True
B. False
Quiz #2

True/false: fold_left is tail-recursive

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

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
True/false: fold_right is tail-recursive

\[
\text{let rec fold_right } f \ l \ a = \\
\quad \text{match } \ l \ \text{with} \\
\quad \quad [] \rightarrow a \\
\quad \quad \mid (h::t) \rightarrow f \ h \ (\text{fold_right } f \ t \ a)
\]

A. True
B. False
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
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)
;;
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
helper x 1

;;
Tail Recursion Pattern with \texttt{rev}

let \texttt{rev} \texttt{x} =

let \texttt{rec} \texttt{rev}\_helper \texttt{arg} \texttt{acc} =

\texttt{match} \texttt{arg} \texttt{with} \texttt{[]} \rightarrow \texttt{acc}

\mid \texttt{h::t} \rightarrow

\quad \texttt{let} \texttt{arg}' = \texttt{t} \texttt{in}

\quad \texttt{let} \texttt{acc}' = \texttt{h::acc} \texttt{in}

\quad \texttt{rev}\_helper \texttt{arg}' \texttt{acc}' \texttt{in} (* \text{end of helper fun} *)

\texttt{rev}\_helper \texttt{x} []

;;

\textit{Can generalize to more than one argument, and multiple cases for each recursive call}
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

```ocaml
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)
https://xkcd.com/1270/

**Why do you like functional programming so much? What does it actually get you?**

*Tail recursion is its own reward.*
Outlook: Is Tail Recursion General?

• 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
  – We will look at this later, if we have time