Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
https://courses.engr.illinois.edu/cs421/fa2017/CS421D

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Recursive Functions

```ocaml
# let rec factorial n =  
  if n = 0 then 1 else n * factorial (n - 1);;  
val factorial : int -> int = <fun>  
# factorial 5;;  
- : int = 120  
# (* rec is needed for recursive function declarations *)
```

Recursion Example

Compute \( n^2 \) recursively using:
\[
 n^2 = (2 \times n - 1) + (n - 1)^2
\]

```ocaml
# let rec nthsq n =  
  match n with  
  | 0 -> 0  
  | n -> (2 * n -1) + nthsq (n -1) ;;  
val nthsq : int -> int = <fun>  
# nthsq 3;;  
- : int = 9
```

Structure of recursion similar to inductive proof

Evaluating expressions in OCaml

- Evaluation uses an environment \( \rho \)
  - \( \text{Eval}(e, \rho) \)
- A constant evaluates to itself, including primitive operators like + and =
  - \( \text{Eval}(c, \rho) = \text{Val}(c) \)
- To evaluate a variable \( v \), look it up in \( \rho \):
  - \( \text{Eval}(v, \rho) = \text{Val}(\rho(v)) \)

- Base case is the last case; it stops the computation
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- \text{if} or \text{match} must contain base case
- Failure of these may cause failure of termination

Evaluating expressions in OCaml

To evaluate a tuple \((e_1, \ldots, e_n)\),

- Evaluate each \( e_i \) to \( v_i \), right to left for Ocaml
- Then make value \((v_1, \ldots, v_n)\)
- \( \text{Eval}((e_1, \ldots, e_n), \rho) = \text{Eval}((e_1, \ldots, \text{Eval}(e_n, \rho)), \rho) \)
- \( \text{Eval}((e_1, \ldots, e_i, \text{Val}(v_{i+1}, \ldots, \text{Val}(v_n), \rho)) = \text{Eval}((e_1, \ldots, \text{Eval}(e_i, \rho), \text{Val}(v_{i+1}, \ldots, \text{Val}(v_n), \rho)) \)
- \( \text{Eval}((\text{Val}(v_1, \ldots, \text{Val}(v_n)), \rho) = \text{Val}(v_1, \ldots, v_n) \)
Evaluating expressions in OCaml

- To evaluate uses of +, -, *, etc, eval args, then do operation +, -, *, +., …)
  - Eval(e1@e2, ρ) => Eval(e1@Eval(e2, ρ), ρ))
  - Eval(e1@Val e2, ρ) => Eval(Eval(e1, ρ)@Val v2, ρ))
  - Eval(Val v₁@Val v₂) => Val (v₁⦿v₂)

- Function expression evaluates to its closure
  - Eval (fun x -> e, ρ) => Val < x -> e,r

9/12/22 7

9/12/22 8

Evaluating expressions in OCaml

- To evaluate a conditional expression:
  - if b then e₁ else e₂

  - Evaluate b to a value v
  - If v is True, evaluate e₁
  - If v is False, evaluate e₂
  
    Eval(if b then e₁ else e₂, ρ) => Eval(Eval(b, ρ) then e₁ else e₂, ρ)

  - Eval(if Val true then e₁ else e₂, ρ) => Eval(e₁, ρ)
  
    Eval(if Val false then e₁ else e₂, ρ) => Eval(e₂, ρ)

9/12/22 9

Evaluation of Application with Closures

- Given application expression f e

  - In Ocaml, evaluate e to value v

  - In environment ρ, evaluate left term to closure, c = <(x₁,...,xₙ) → b, ρ’>

    - (x₁,...,xₙ) variables in (first) argument
    - v must have form (v₁,...,vₙ)

  - Update the environment ρ’ to

    ρ” = {x₁ → v₁,..., xₙ →vₙ}⁺ ρ’

  - Evaluate body b in environment ρ”

9/12/22 10

Evaluation of Application with Closures

- Eval(f e, ρ) => Eval(f (Eval(e, ρ)), ρ)

  - Eval(f (Val v), ρ) => Eval((Eval(f, ρ)) (Val v), ρ)

- Eval((Val <(x₁,...,xₙ) → b, ρ’>)(Val (v₁,...,vₙ)), ρ) => Eval(b, {x₁ → v₁,..., xₙ → vₙ}⁺ p’)

9/12/22 11

9/12/22 12

Evaluation of Application of plus_x;;
Evaluation of Application of \texttt{plus}_x;

- Have environment:
  \[ \rho = \{ \text{\texttt{plus}_x} \to <y \to y + x, \text{\texttt{plus}_x}>, \ldots, y \to 19, x \to 17, z \to 3, \ldots \} \]
  where \( \rho_{\text{\texttt{plus}_x}} = \{ x \to 12, \ldots, y \to 24, \ldots \} \)
  - \text{Eval} \left( \text{\texttt{plus}_x} \ z, \rho \right) =>
  - \text{Eval} \left( \text{\texttt{plus}_x} \left( \text{Eval}(z, \rho) \right), \rho \right) =>
  - \text{Eval} \left( \text{\texttt{plus}_x} \left( \text{Val} \ 3 \right), \rho \right) => \ldots

Evaluation of Application of \texttt{plus}_x;

- Have environment:
  \[ \rho = \{ \text{\texttt{plus}_x} \to <y \to y + x, \text{\texttt{plus}_x}>, \ldots, y \to 19, x \to 17, z \to 3, \ldots \} \]
  where \( \rho_{\text{\texttt{plus}_x}} = \{ x \to 12, \ldots, y \to 24, \ldots \} \)
  - \text{Eval} \left( \text{\texttt{plus}_x} \left( \text{Eval}(y + x, \rho_{\text{\texttt{plus}_x}}) \right)(\text{Val} \ 3 \ ), \rho \right) => \ldots

Evaluation of Application of \texttt{plus}_x;

- Have environment:
  \[ \rho = \{ \text{\texttt{plus}_x} \to <y \to y + x, \text{\texttt{plus}_x}>, \ldots, y \to 19, x \to 17, z \to 3, \ldots \} \]
  where \( \rho_{\text{\texttt{plus}_x}} = \{ x \to 12, \ldots, y \to 24, \ldots \} \)
  - \text{Eval} \left( \text{\texttt{plus}_x} \left( \text{Val} <y \to y + x, \rho_{\text{\texttt{plus}_x}} > \right)(\text{Val} \ 3 \ ), \rho \right) => \ldots

Evaluation of Application of \texttt{plus}_x;

- Have environment:
  \[ \rho = \{ \text{\texttt{plus}_x} \to <y \to y + x, \text{\texttt{plus}_x}>, \ldots, y \to 19, x \to 17, z \to 3, \ldots \} \]
  where \( \rho_{\text{\texttt{plus}_x}} = \{ x \to 12, \ldots, y \to 24, \ldots \} \)
  - \text{Eval} \left( \text{\texttt{plus}_x} \left( \text{Val} <y \to y + x, \rho_{\text{\texttt{plus}_x}} > \right)(\text{Val} \ 3 \ ), \rho \right) => \ldots
  - \text{Eval} \left( y + x, \{ y \to 3 \} + \rho_{\text{\texttt{plus}_x}} \right) => \ldots

Evaluation of Application of \texttt{plus}_x;

- Have environment:
  \[ \rho = \{ \text{\texttt{plus}_x} \to <y \to y + x, \text{\texttt{plus}_x}>, \ldots, y \to 19, x \to 17, z \to 3, \ldots \} \]
  where \( \rho_{\text{\texttt{plus}_x}} = \{ x \to 12, \ldots, y \to 24, \ldots \} \)
  - \text{Eval} \left( y + x, \{ y \to 3 \} + \rho_{\text{\texttt{plus}_x}} \right) =>
  - \text{Eval(y + Eval(x, \{ y \to 3 \} + \rho_{\text{\texttt{plus}_x}}),}
  \{ y \to 3 \} + \rho_{\text{\texttt{plus}_x}} \right) => \ldots
Evaluation of Application of plus_x;;

Have environment:
\[ \rho = \{ \text{plus}_x \to <y \to y + x, r_{\text{plus}_x}>, \ldots, y \to 19, x \to 17, z \to 3, \ldots \} \]
where \( r_{\text{plus}_x} = \{ x \to 12, \ldots, y \to 24, \ldots \} \)

\[ \text{Eval } ((\text{Val}<y \to y + x, r_{\text{plus}_x}>(\text{Val } 3), \rho) => \]
\[ \text{Eval } (y + x, \{ y \to 3 \} + r_{\text{plus}_x}) => \]
\[ \text{Eval}(y + \text{Eval}(x, \{ y \to 3 \} + r_{\text{plus}_x}), \{ y \to 3 \} + r_{\text{plus}_x}) => \]
\[ \text{Eval}(y + \text{Val } 12, \{ y \to 3 \} + r_{\text{plus}_x}) => \ldots \]

Evaluation of Application of plus_pair

Assume environment
\[ \rho = \{ \text{plus}_x \to <y \to y + x, r_{\text{plus}_x}>, \ldots, y \to 19, x \to 17, z \to 3, \ldots \} \]
where \( r_{\text{plus}_x} = \{ x \to 12, \ldots, y \to 24, \ldots \} \)

\[ \text{Eval}((\text{Val}<y \to y + x, r_{\text{plus}_x}>(\text{Val } 4, \text{Val } 3), \rho)) => \]
\[ \text{Eval}((\text{Val}<y \to y + x, r_{\text{plus}_x}>(\text{Val } 4, \text{Val } 3), \rho)) => \]
\[ \text{Eval}((\text{Val}<y \to y + x, r_{\text{plus}_x}>(\text{Val } 4, \text{Val } 3), \rho)) => \]
\[ \text{Eval}((\text{Val}<y \to y + x, r_{\text{plus}_x}>(\text{Val } 4, \text{Val } 3), \rho)) => \]
\[ \text{Eval}((\text{Val}<y \to y + x, r_{\text{plus}_x}>(\text{Val } 4, \text{Val } 3), \rho)) => \]
\[ \text{Eval}((\text{Val}<y \to y + x, r_{\text{plus}_x}>(\text{Val } 4, \text{Val } 3), \rho)) => \]
\[ \text{Eval}((\text{Val}<y \to y + x, r_{\text{plus}_x}>(\text{Val } 4, \text{Val } 3), \rho)) => \]
\[ \text{Eval}((\text{Val}<y \to y + x, r_{\text{plus}_x}>(\text{Val } 4, \text{Val } 3), \rho)) => \]
Lists

List can take one of two forms:
- Empty list, written \([\ ]\)
- Non-empty list, written \(x :: xs\)
  - \(x\) is head element, \(xs\) is tail list, \(::\) called “cons”
- Syntactic sugar: \([x]\) == \(x :: [\ ]\)
- \([x_1; x_2; \ldots; x_n]\) == \(x_1 :: x_2 :: \ldots :: x_n :: [\ ]\)

Functions Over Lists

# let rec double_up list = match list with [\ ] -> [\ ] (* pattern before ->, expression after *) | (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>

Answer

- Which one of these lists is invalid?
  1. \([2; 3; 4; 6]\)
  2. \([2,3; 4,5; 6,7]\)
  3. \([(2.3,4); (3.2,5); (6,7.2)]\)
  4. \["hi"; "there"; ["wahcha"]; [ ]; ["doin"]\]
  - 3 is invalid because of last pair

Question

- Which one of these lists is invalid?
  1. \([2; 3; 4; 6]\)
  2. \([2,3; 4,5; 6,7]\)
  3. \([(2.3,4); (3.2,5); (6,7.2)]\)
  4. \["hi"; "there"; ["wahcha"]; [ ]; ["doin"]\]
Functions Over Lists

```ocaml
# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]
# let rec poor_rev list =
  match list
  with [] -> []
  | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
# poor_rev silly;;
-: string list = ["there"; "there"; "hi"; "hi"]
```

Structural Recursion

- Functions on recursive datatypes (eg lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
  - Recursive calls made to components of structure of the same recursive type
  - Base cases of recursive types stop the recursion of the function

Question: Length of list

- Problem: write code for the length of the list
- How to start?
  ```ocaml
  let rec length list =
  ```

- What patterns should we match against?
  ```ocaml
  let rec length list =
    match list with
    | [] -> | (a :: bs) ->
  ```
Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when list is empty?

```ocaml
let rec length list =
  match list with [] -> 0
  | (a :: bs) ->
```

9/12/22 37

Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when list is not empty?

```ocaml
let rec length list =
  match list with [] -> 0
  | (a :: bs) ->
```

9/12/22 38

Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when list is not empty?

```ocaml
let rec length list =
  match list with [] -> 0
  | (a :: bs) -> 1 + length bs
```

9/12/22 39

Structural Recursion: List Example

```ocaml
# let rec length list = match list
with [ ] -> 0 (* Nil case *)
| a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list - int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

9/12/22 40

Nil case [ ] is base case
Cons case recurses on component list bs

Question: Same Length

- How can we efficiently answer if two lists have the same length?

```ocaml
let rec same_length list1 list2 =
  match list1 with [] ->
    (match list2 with [] -> true
    | (y::ys) -> false)
  | (x::xs) ->
    (match list2 with [] -> false
    | (y::ys) -> same_length xs ys)
```

9/12/22 41

Same Length

- How can we efficiently answer if two lists have the same length?

```ocaml
let rec same_length list1 list2 =
  match list1 with [] ->
    (match list2 with [] -> true
    | (y::ys) -> false)
  | (x::xs) ->
    (match list2 with [] -> false
    | (y::ys) -> same_length xs ys)
```
Your turn: doubleList : int list -> int list

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

let rec doubleList list =
  match list with
  | [] -> []
  | x :: xs -> (2 * x) :: doubleList xs

Higher-Order Functions Over Lists

```ocaml
# let rec map f list =
  match list with
  | [] -> []
  | (h::t) -> (f h) :: (map f t);
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```ocaml
# : int list = [10; 7; 5; 4; 3; 3]
# : int list = [12; 7; 4; 2; 1; 0; 0]
```

Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list =
  List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
```

```ocaml
# : int list = [4; 6; 8]
```
Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list =  
  List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
-: int list = [4; 6; 8]
```

Folding Recursion

- Another common form “folds” an operation over the elements of the structure

```ocaml
# let rec multList list =  
  match list with  
  | [] -> 1  
  | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

Folding Recursion: Length Example

```ocaml
# let rec length list =  
  match list with  
  | [] -> 0  
  | a :: bs -> 1 + length bs;;
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion form of Structural Recursion
- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
- Wait until whole structure has been traversed to start building answer

Forward Recursion: Examples

```ocaml
# let rec double_up list =  
  match list with  
  | [] -> []  
  | x::xs -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>
```

```ocaml
# let rec poor_rev list =  
  match list with  
  | [] -> []  
  | (x::xs) -> let r = poor_rev xs in r @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```

Forward Recursion: Examples

```ocaml
# let rec double_up list =  
  match list with  
  | [] -> []  
  | x::xs -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>
```

```ocaml
# let rec poor_rev list =  
  match list with  
  | [] -> []  
  | (x::xs) -> let r = poor_rev xs in r @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```
Recursing over lists

```ocaml
# let rec fold_right f list b = match list with [] -> b | (x :: xs) -> f x (fold_right f xs b);
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b = <fun>
# fold_right (fun s -> fun () -> print_string s) [
  "hi";
  "there"
] ();;
therehi-: unit = ()
```

Folding Recursion

```ocaml
Folding Recursion : Length Example

# let rec length list = match list with [] -> 0 (* Nil case *) | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>
# let length list =
  fold_right (fun a -> fun r -> 1 + r) list 0;;
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

Forward Recursion: Examples

```ocaml
Forward Recursion: Examples

# let rec double_up list = match list with [] -> [] | (x :: xs) -> x :: x :: double_up xs;;
val double_up : 'a list -> 'a list = <fun>
# double_up ["a"; "b"];;
- : string list = ["a"; "a"; "b"; "b"]
```

Encoding Forward Recursion with Fold

```ocaml
Encoding Forward Recursion with Fold

# let rec append list1 list2 = append : 'a list -> 'a list -> 'a list = <fun>

val append : 'a list -> 'a list -> 'a list = <fun>
```

The Primitive Recursion Fairy
Encoding Forward Recursion with Fold

# let rec append list1 list2 = match list1 with
[ ] -> list2
val append : 'a list -> 'a list -> 'a list = <fun>

Base Case

# let append list1 list2 = fold_right (fun x y -> x :: y) list1 list2
val append : 'a list -> 'a list -> 'a list = <fun>

# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]

Base Case Operation Recursive Call
Encoding Forward Recursion with Fold

```
# let rec append list1 list2 = match list1 with
[ ] -> list2 | x::xs -> x :: append xs list2;
val append : 'a list -> 'a list -> 'a list = <fun>
```

<table>
<thead>
<tr>
<th>Base Case</th>
<th>Operation</th>
<th>Recursive Call</th>
</tr>
</thead>
<tbody>
<tr>
<td>list1 = []</td>
<td>-&gt;</td>
<td>list2</td>
</tr>
<tr>
<td>list1 = x::xs</td>
<td></td>
<td>list2</td>
</tr>
</tbody>
</table>

```
# let append list1 list2 = fold_right (fun x -> fun y -> x :: y) list1 list2;
val append : 'a list -> 'a list -> 'a list = <fun>
```

```
append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Tail Recursion

- A recursive program is tail recursive if all recursive calls are tail calls
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls
- Tail recursion generally requires extra “accumulator” arguments to pass partial results
- May require an auxiliary function

```
# let rec rev_aux list revlist = match list with
[ ] -> revlist | x :: xs -> rev_aux xs (x::revlist);
val rev_aux : 'a list -> 'a list = <fun>

# let rev list = rev_aux list [];;
val rev : 'a list -> 'a list = <fun>
```

Tail Recursion - length

- How can we write length with tail recursion?
- Let length list =
  
  let rec length_aux list acc_length =
  match list with
  | [] -> acc_length
  | (x::xs) ->
  length_aux xs (1 + acc_length)
  in length_aux list 0

```
# let length list = let rec length_aux list acc_length=
  match list with
  | [] -> acc_length
  | (x::xs) ->
  length_aux xs (1 + acc_length)
  in length_aux list 0
```

Tail Recursion - Example

```
# let rec rev_aux list revlist =
  match list with
  | [] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);
val rev_aux : 'a list -> 'a list -> 'a list = <fun>

# let rev list = rev_aux list [];;
val rev : 'a list -> 'a list = <fun>
```

Comparison

- poor_rev [1;2;3] =
  (poor_rev [2;3]) @ [1] =
  (((poor_rev [3]) @ [2]) @ [1] =
  (((([ ] @ [3]) @ [2]) @ [1] =
  ([3] @ [2]) @ [1] =
  (3:: ([ ] @ [2])) @ [1] =
  [3;2] @ [1] =
  3 :: ([2] @ [1]) =
  3 :: (2:: ([ ] @ [1]))) = [3; 2; 1]

```
# poor_rev [1;2;3] =
(poor_rev [2;3]) @ [1] =
(((poor_rev [3]) @ [2]) @ [1] =
((([ ] @ [3]) @ [2]) @ [1] =
([3] @ [2]) @ [1] =
(3:: ([ ] @ [2])) @ [1] =
[3;2] @ [1] =
3 :: ([2] @ [1]) =
3 :: (2:: ([ ] @ [1]))) = [3; 2; 1]
```
Iterating over lists

# let rec fold_left f a list =
match list
with [] -> a |
(x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left
(fun () -> print_string) ()
["hi"; "there"];;
hithere : unit = ()

Folding

# let rec fold_left f a list =
match list
with [] -> a |
(x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>fold_left f a [x1; x2;...;xn] = f...(f (f a x1) x2)...xn

Folding - Tail Recursion

- # let rev list =
  - fold_left
  - (fun l -> fun x -> x :: l)  //comb op
    []  //accumulator cell
    list

Folding

- Can replace recursion by fold_right in any forward primitive recursive definition
  - Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any tail primitive recursive definition

How long will it take?

- Remember the big-O notation from CS 225 and CS 374
- Question: given input of size $n$, how long to generate output?
- Express output time in terms of input size, omit constants and take biggest power

Common big-O times:
- Constant time $O(1)$
  - input size doesn’t matter
- Linear time $O(n)$
  - double input $\Rightarrow$ double time
- Quadratic time $O(n^2)$
  - double input $\Rightarrow$ quadruple time
- Exponential time $O(2^n)$
  - increment input $\Rightarrow$ double time
**Linear Time**

- Expect most list operations to take linear time $O(n)$
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: `multList`, `append`
- Integer example: `factorial`

**Quadratic Time**

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:
  ```ml
  # let rec poor_rev list = match list
  | [] -> []
  | (x::xs) -> poor_rev xs @ [x];;
  val poor_rev : 'a list -> 'a list = <fun>
  ```

**Exponential running time**

- Poor worst-case running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear

**Exponential running time**

```ml
# let rec slow n =
  if n <= 1
  then 1
  else 1+slow (n-1) + slow(n-2);;
val slow : int -> int = <fun>
# List.map slow [1;2;3;4;5;6;7;8;9];;
- : int list = [1; 3; 5; 9; 15; 25; 41; 67; 109]
```  

**An Important Optimization**

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if $f$ calls $g$ and $g$ calls $h$, but calling $h$ is the last thing $g$ does (a tail call)?

  ![Normal call diagram]

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished
- What if $f$ calls $g$ and $g$ calls $h$, but calling $h$ is the last thing $g$ does (a tail call)?
- Then $h$ can return directly to $f$ instead of $g$