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

```
# 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:
$n^2 = (2 * n - 1) + (n - 1)^2$

```
# let rec nthsq n =          (* rec for recursion *)
   match n                (* pattern matching for cases *)
   with 0 -> 0                (* base case *)
   | n -> (2 * n -1)          (* recursive case *)
          + nthsq (n -1);;   (* recursive call *)
val nthsq : int -> int = <fun>
# nthsq 3;;
-  : int = 9
```

Structure of recursion similar to inductive proof

Recursion and Induction

```
# let rec nthsq n = match n with 0 -> 0
       | n -> (2 * n - 1) + nthsq (n - 1) ;;
```

- 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
- **if** or **match** must contain base case
- Failure of these may cause failure of termination

Evaluating expressions in OCaml

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

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 $(v_1,\ldots,v_n)$
  - $\mathrm{Eval}((e_1,\ldots,e_n),\rho) \Rightarrow \mathrm{Eval}((e_1,\ldots,\mathrm{Eval}(e_n, \rho)), \rho)$
  - $\mathrm{Eval}((e_1,\ldots,e_i, \mathrm{Val}\ v_{i+1},\ldots, \mathrm{Val}\ v_n) , \rho) \Rightarrow \mathrm{Eval}((e_1,\ldots,\mathrm{Eval}(e_i, \rho), \mathrm{Val}\ v_{i+1},\ldots, \mathrm{Val}\ v_n) , \rho)$
  - $\mathrm{Eval}((\mathrm{Val}\ v_1,\ldots,\mathrm{Val}\ v_n) , \rho) \Rightarrow \mathrm{Val}\ (v_1,\ldots,v_n)$

Evaluating expressions in OCaml

- To evaluate uses of +, - , etc, eval args, then do operation ⦿ (+, -, *, +., ….)
  - Eval(e_1⦿e_2, ρ) => Eval(e_1⦿Eval(e_2, ρ), ρ))
  - Eval(e_1⦿Val e_2, ρ)=>Eval(Eval(e_1, ρ)⦿Val v_2, ρ))
  - Eval(Val v_1⦿ Val v_2) => Val (v_1⦿ v_2)

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

Evaluating expressions in OCaml

- To evaluate a local dec: let x = e1 in e2
  - Eval e1 to v, then eval e2 using {x → v} + ρ

  - Eval(let x = e_1 in e_2, ρ) =>
    Eval(let x = Eval(e_1, ρ) in e_2, ρ)
  - Eval(let x = Val v in e_2, ρ) =>
    Eval(e_2, {x → v} + ρ)

Evaluating expressions in OCaml

- To evaluate a conditional expression:
  if b then e_1 else e_2
  - Evaluate b to a value v
  - If v is True, evaluate e_1
  - If v is False, evaluate e_2

  - Eval(if b then e_1 else e_2, ρ) =>
    Eval(if Eval(b, ρ) then e_1 else e_2, ρ)
  - Eval(if Val true then e_1 else e_2, ρ) =>Eval(e_1, ρ)
  - Eval(if Val false then e_1 else e_2, ρ) =>Eval(e_2, ρ)

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 = <(x_1,…,x_n) → b, ρ'>
  - (x_1,…,x_n) variables in (first) argument
  - v must have form (v_1,…,v_n)
- Update the environment ρ' to
  ρ'' = {x_1 → v_1,…, x_n →v_n}+ ρ'
- Evaluate body b in environment ρ''

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 <(x_1,…,x_n) → b, ρ'>)(Val (v_1,…,v_n)), ρ)=>
  Eval(b, {x_1 → v_1,…, x_n → v_n}+ρ')

Evaluation of Application of plus_x;;

- Have environment:
  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}
  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval (plus_x z, ρ) =>
- Eval(plus_x (Eval(z, ρ)) => …

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval (plus_x z, ρ) =>
- Eval(plus_x (Eval(z, ρ)), ρ) =>
- Eval(plus_x (Val 3), ρ) => …

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval (plus_x z, ρ) =>
- Eval (plus_x (Eval(z, ρ)), ρ) =>
- Eval (plus_x (Val 3), ρ) =>
- Eval ((Eval(plus_x, ρ)) (Val 3), ρ) => …

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval (plus_x z, ρ) =>
- Eval (plus_x (Eval(z, ρ)), ρ) =>
- Eval (plus_x (Val 3), ρ) =>
- Eval ((Eval(plus_x, ρ)) (Val 3), ρ) =>
- Eval ((Val<y → y + x, ρ_{plus_x} >)(Val 3), ρ) => …

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval ((Val<y → y + x, ρ_{plus_x} >)(Val 3), ρ) => …

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval ((Val<y → y + x, ρ_{plus_x} >)(Val 3), $\cancel{\rho}$) =>
- Eval (y + x, {y → 3} +ρ_{plus_x}) => …

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval ((Val<y → y + x, ρ_{plus_x} >)(Val 3), ρ) =>
- Eval (y + x, {y → 3} +ρ_{plus_x}) =>
- Eval(y+Eval(x, {y → 3} +ρ_{plus_x}),
  {y → 3} +ρ_{plus_x}) => …

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval ((Val<y → y + x, ρ_{plus_x} >)(Val 3), ρ) =>
- Eval (y + x, {y → 3} +ρ_{plus_x}) =>
- Eval(y+Eval(x, {y → 3} +ρ_{plus_x}), {y → 3} +ρ_{plus_x}) =>
- Eval(y+Val 12,{y → 3} +ρ_{plus_x}) => …

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval(y+Eval(x, {y → 3} +ρ_{plus_x}), {y → 3} +ρ_{plus_x}) =>
- Eval(y+Val 12,{y → 3} +ρ_{plus_x}) =>
- Eval(Eval(y, {y → 3} +ρ_{plus_x}) + Val 12,{y → 3} +ρ_{plus_x}) =>…

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval(Eval(y, {y → 3} +ρ_{plus_x}) + Val 12,{y → 3} +ρ_{plus_x}) =>
- Eval(Val 3 + Val 12 ,{y → 3} +ρ_{plus_x}) =>…

Evaluation of Application of plus_x;;

- Have environment:

  ρ = {plus_x → <y → y + x, ρ_{plus_x} >, … ,
  y → 19, x →17, z →3, …}

  where ρ_{plus_x} = {x → 12, … , y → 24, …}
- Eval(Eval(y, {y → 3} +ρ_{plus_x}) + Val 12,{y → 3} +ρ_{plus_x}) =>
- Eval(Val 3 + Val 12 ,{y → 3} +ρ_{plus_x}) =>
- Val (3 + 12) = Val 15

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3…,
plus_pair →<(n,m) →n + m, ρ_{plus_pair}>} + ρ_{plus_pair}

- Eval (plus_pair (4,x), ρ)=>
- Eval (plus_pair (Eval ((4, x), ρ)), ρ) =>
- Eval (plus_pair (Eval ((4, Eval (x , ρ)), ρ)), ρ) =>
- Eval (plus_pair (Eval ((4, Val 3), ρ)), ρ) =>
- Eval (plus_pair (Eval ((Eval (4, ρ), Val 3), ρ)), ρ) =>
- Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>

Evaluation of Application of plus_pair

- Assume environment

ρ = {x → 3…,
plus_pair →<(n,m) →n+m, ρ_{plus_pair}>} + ρ_{plus_pair}

- Eval (plus_pair (Eval ((Val 4, Val 3), ρ)), ρ) =>
- Eval (plus_pair (Val (4, 3)), ρ) =>
- Eval (Eval (plus_pair, ρ), Val (4, 3)), ρ) => …
- Eval ((Val<(n,m)→n+m, ρ_{plus_pair}>)(Val(4,3)) , ρ)=>
- Eval (n + m, {n -> 4, m -> 3} + ρ_{plus_pair}) =>
- Eval (4 + 3, {n -> 4, m -> 3} + ρ_{plus_pair}) => 7

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 :: [ ]
  - [x1; x2; …; xn] == x1 :: x2 :: … :: xn :: [ ]

Lists

```
# let fib5 = [8;5;3;2;1;1];;
val fib5 : int list = [8; 5; 3; 2; 1; 1]
# let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
# (8::5::3::2::1::1::[ ]) = fib5;;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1;
  1]
```

Lists are Homogeneous

```
# let bad_list = [1; 3.2; 7];;
Characters 19-22:
  let bad_list = [1; 3.2; 7];;
                     ^^^
This expression has type float but is here
  used with type int
```

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"]]

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

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>
# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1;
  1; 1; 1]
```

Functions Over Lists

```
# 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?

```
let rec length list =
```

Question: Length of list

- Problem: write code for the length of the list
  - How to start?

```
let rec length list =
    match list with
```

Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

```
let rec length list =
    match list with
```

Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

```
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?

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

Question: Length of list

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

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

Question: Length of list

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

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

Structural Recursion : List Example

```
# 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
```

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

Same Length

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

Same Length

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

```
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 =
```

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
```

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

```
# let rec map f list =
  match list
  with [] -> []
  | (h::t) -> (f h) :: (map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Higher-Order Functions Over Lists

```
# let rec map f list =
  match list
  with [] -> []
  | (h::t) -> (f h) :: (map f t);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
```

Mapping Recursion

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

```
# 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]
```

Mapping Recursion

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

```
# 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]
```

- Same function, but no explicit recursion

Folding Recursion

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

```
# 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
```

- Computes (2 * (4 * (6 * 1)))

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>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [ ] is base case, 0 is the base value
- Cons case recurses on component list bs
- What do multList and length have in common?

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

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

# 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

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

Base Case | Operator | Recursive Call

```
# 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>
```

Base Case | Operator | Recursive Call

Recursing over lists


The Primitive Recursion Fairy

```
# 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 -> 'b =
  <fun>
# fold_right
    (fun s -> fun () -> print_string s)
    ["hi"; "there"]
    ();;
therehi- : unit = ()w
```

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
```

Folding Recursion

- multList folds to the right
- Same as:

```
# let multList list =
    List.fold_right
    (fun x -> fun p -> x * p)
    list 1;;
val multList : int list -> int = <fun>
# multList [2;4;6];;
- : int = 48
```

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>
```

Base Case Operator Recursive Call

```
# let double_up =
    fold_right (fun x -> fun r -> x :: x :: r) list [ ]
```

Operator Recursive result Base Case

```
# double_up  ["a";"b"];;
- : string list = ["a"; "a"; "b"; "b"]
```

Encoding Forward Recursion with Fold

```
# let rec append list1 list2 =

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

Encoding Forward Recursion with Fold

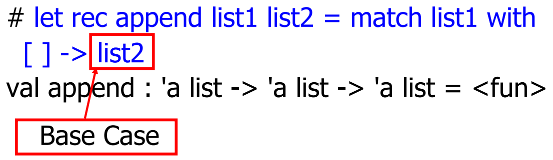
```
# let rec append list1 list2 = match list1 with

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

Encoding Forward Recursion with Fold

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

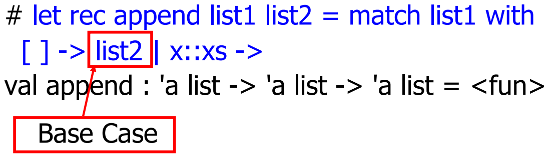
Encoding Forward Recursion with Fold



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

Base Case

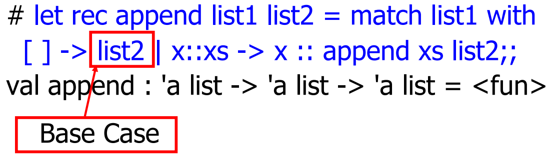
Encoding Forward Recursion with Fold



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

Base Case

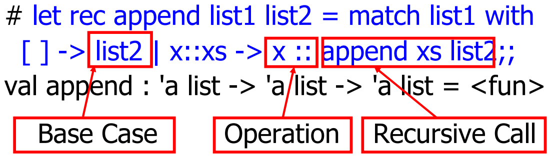
Encoding Forward Recursion with Fold



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

Base Case

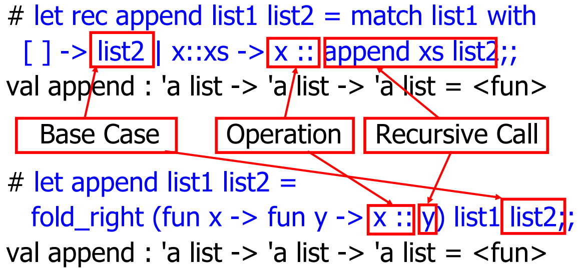
Encoding Forward Recursion with Fold



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

Base Case Operation Recursive Call

Encoding Forward Recursion with Fold

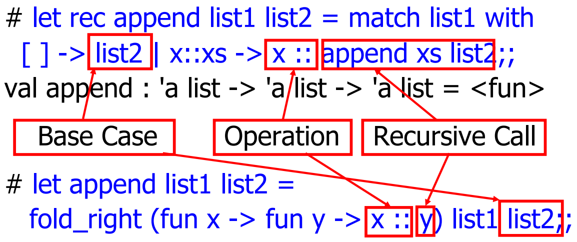


```
# let rec append list1 list2 = match list1 with
  [ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

Base Case Operation Recursive Call

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

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>
```

Base Case
Operation
Recursive Call

```
# 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

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
```

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>
```

- What is its running time?

Comparison

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

Comparison

- rev [1;2;3] =
- rev_aux [1;2;3] [ ] =
- rev_aux [2;3] [1] =
- rev_aux [3] [2;1] =
- rev_aux [ ] [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 - Tail Recursion

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

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 [x_1; x_2;…;x_n] = f(…(f (f a x_1) x_2)…)x_n

```
# 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 -> 'b =
   <fun>
```

fold_right f [x_1; x_2;…;x_n] b = f x_1(f x_2 (…(f x_n b)…))

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

How long will it take?

Common big-O times:

- Constant time $O(1)$
  - input size doesn't matter
- Linear time $O(n)$
  - double input ⇒ double time
- Quadratic time $O(n^2)$
  - double input ⇒ quadruple time
- Exponential time $O(2^n)$
  - increment input ⇒ 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:

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

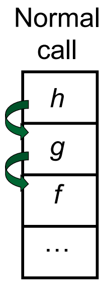
Exponential running time

- 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

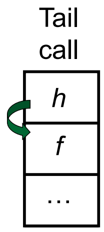
```
# let rec slow n =
       if n <= 1
      then 1
      else 1+slow (n-1) + slow(n-2);;
val slow : int -> int = <fun>
# List.map slow [1;2;3;4;5;6;7;8;9];;
- : int list = [1; 3; 5; 9; 15; 25; 41; 67;
  109]
```

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*)?

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*)?
- Then *h* can return directly to *f* instead of *g*