Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

https://courses.engineering.illinois.edu/cs421/sp2023

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

```ocaml
# 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]
```
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"]
How can we efficiently answer if two lists have the 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 =

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

```ocaml
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

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

2/1/23
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>
# 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

```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
# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]
```

```ocaml
# 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

```ml
# 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]
```
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]
```

Same function, but no explicit recursion
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
```

- Computes \((2 \times (4 \times (6 \times 1)))\)
Folding Recursion: Length 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
```

- 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

```ocaml
# 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
```

```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>

Base Case  Operator  Recursive Call
```
# 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 = ()
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:

```ocaml
# 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

```ocaml
# 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

```ocaml
# 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>

# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
Encoding Forward Recursion with Fold

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

Base Case
# 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

```ocaml
# 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

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

# 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.
Terminology

- **Available**: A function call that can be executed by the current expression.

- The fastest way to be unavailable is to be guarded by an abstraction (anonymous function, lambda lifted).

  - if \((h \, x)\) then \(f \, x\) else \((x + g \, x)\)
  - if \((h \, x)\) then \((\text{fun } x \rightarrow f \, x)\) else \((g \,(x + x))\)

Not available
Terminology

- Tail Position: A subexpression $s$ of expressions $e$, which is **available** and such that if evaluated, will be taken as the value of $e$
  - if \((x > 3)\) then $x + 2$ else $x - 4$
  - let $x = 5$ in $x + 4$

- Tail Call: A function call that occurs in tail position
  - if \((h x)\) then $f x$ else $x + g x$
Tail Recursion - length

- How can we write length with tail recursion?

```ocaml
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
```