Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha
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]
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 (e.g., 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.
# let rec length list = match list
  with [ ] -> 0 (* Nil case *)
  | x :: xs -> 1 + length xs;; (* 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 xs
Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse on components

- 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>

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

- How do you write length with forward recursion?

```plaintext
let rec length l =
```
Question

How do you write length with forward recursion?

```ml
let rec length l =
  match l with [] ->
  | (a :: bs) ->
```
Question

- How do you write length with forward recursion?

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

Question

How do you write length with forward recursion?

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

Try Problem 2 on ML2
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 \textit{tail call})?
- Then $h$ can return directly to $f$ instead of $g$. 
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.
Example of Tail Recursion

- **Forward recursive:**
  ```ocaml
  # let rec prod l = 
      match l with [] -> 1 
      | (x :: rem) -> x * prod rem;;
  val prod : int list -> int = <fun>
  ```

- **Tail recursive:**
  ```ocaml
  # let prod list = 
      let rec prod_aux l acc = 
          match l with [] -> acc 
          | (y :: rest) -> prod_aux rest (acc * y) 
      (* Uses associativity of multiplication *) 
      in prod_aux list 1;;
  val prod : int list -> int = <fun>
  ```
Question

- How do you write length with tail recursion?

```ml
let length l =
```
Question

- How do you write length with tail recursion?

```ocaml
let length l =
  let rec length_aux list n =
  in
```

9/11/17
Question

- How do you write length with tail recursion?

```haskell
let length l =
    let rec length_aux list n =
        match list with [] ->
            | (a :: bs) ->
    in
```
Question

- How do you write length with tail recursion?

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

    in
```
How do you write length with tail recursion?

```ml
let length l =

  let rec length_aux list n =
  match list with [] -> n
  | (a :: bs) -> length_aux


```
Question

- How do you write length with tail recursion?

```ml
let length l =
    let rec length_aux list n =
        match list with [] -> n
        | (a :: bs) -> length_aux bs
    in
```
Question

- How do you write length with tail recursion?

```ocaml
let length l =
  let rec length_aux list n =
    match list with [] -> n
    | (a :: bs) -> length_aux bs (n + 1)
  in
```
Question

- How do you write length with tail recursion?

```ml
let length l =
  let rec length_aux list n =
    match list with [] -> n
    | (a :: bs) -> length_aux bs (n + 1)
  in length_aux l 0
```
Your turn now

Try Problem 4 on MP2
One common form of structural recursion applies a function to each element in the structure.

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

```ocaml
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```
Mapping 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

```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]

- Same function, but no rec
Your turn now

Write a function

make_app : (('a -> 'b) * 'a) list -> 'b list

that takes a list of function – input pairs and gives the result of applying each function to its argument. Use map, no explicit recursion.

let make_app l =
Folding Recursion

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

```ml
# 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 Functions over Lists

How are the following functions similar?

```ocaml
# let rec sumlist list = match list with
    [ ] -> 0 | x::xs -> x + sumlist xs;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9

# let rec prodlist list = match list with
    [ ] -> 1 | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```
Folding Functions over Lists

How are the following functions similar?

# let rec sumList list = match list with
  [ ] -> 0 | x::xs -> x + sumList xs;;
val sumList : int list -> int = <fun>
# sumList [2;3;4];;
- : int = 9
# let rec multList list = match list with
  [ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 24
Folding Functions over Lists

How are the following functions similar?

```
# let rec sumList list = match list with
  [ ] -> 0 | x::xs -> x + sumList xs;;
val sumList : int list -> int = <fun>
# sumList [2;3;4];;
- : int = 9
```

```
# let rec multList list = match list with
  [ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 24
```
Folding Functions over Lists

How are the following functions similar?

```ocaml
# let rec sumList list = match list with
  [ ] -> 0 | x::xs -> x + sumList xs;;
val sumList : int list -> int = <fun>
# sumList [2;3;4];;
- : int = 9
# let rec multList list = match list with
  [ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 24
```

Head Element
How are the following functions similar?

```ocaml
# let rec sumList list = match list with
  [ ] -> 0 | x::xs -> x + sumList xs;;
val sumList : int list -> int = <fun>
# sumList [2;3;4];;
- : int = 9
# let rec multList list = match list with
  [ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 24
```

Combining Operation
Folding Functions over Lists

How are the following functions similar?

# let rec sumList list = match list with
  [ ] -> 0 | x::xs -> x + sumList xs;;
val sumList : int list -> int = <fun>
# sumList [2;3;4];;
- : int = 9

# let rec multList list = match list with
  [ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 24R

Combining Operation
Recurring 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 -> 'b = <fun>
# fold_right
  (fun s -> fun () -> print_string s)
  ["hi"; "there"]
  ();;
therehi- : unit = ()
```
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
```
Encoding 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 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]
Question

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

How do you write length with fold_right, but no explicit recursion?
let rec length l =
  match l with [] -> 0
  | (a :: bs) -> 1 + length bs

How do you write length with fold_right, but no explicit recursion?

let length list =
  List.fold_right (fun x -> fun n -> n + 1) list 0
Map from Fold

```ocaml
# let map f list =
    fold_right (fun x -> fun y -> f x :: y) list [ ];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map ((+)1) [1;2;3];;
- : int list = [2; 3; 4]
```

- Can you write fold_right (or fold_left) with just map? How, or why not?
Iterating over lists

```ocaml
# 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 = ()
```
Encoding Tail Recursion with fold_left

```ocaml
# let prod list = let rec prod_aux l acc =
    match l with [] -> acc
    | (y :: rest) -> prod_aux rest (acc * y)
  in prod_aux list 1;;
val prod : int list -> int = <fun>

# let prod list =
    List.fold_left (fun acc y -> acc * y) 1 list;;
val prod : int list -> int = <fun>

# prod [4;5;6];;
- : int = 120
```
Question

let length l =

let rec length_aux list n =

match list with [] -> n
| (a :: bs) -> length_aux bs (n + 1)

in length_aux l 0

- How do you write length with fold_left, but no explicit recursion?
let length l =
  let rec length_aux list n =
  match list with [] -> n
  | (a :: bs) -> length_aux bs (n + 1)
in length_aux l 0

How do you write length with fold_left, but
no explicit recursion?

let length list =
  List.fold_left (fun n -> fun x -> n + 1) 0 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)...))
Recall

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

What is its running time?
Quadratic Time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

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

- \text{poor\_rev}[1,2,3] =
- (\text{poor\_rev}[2,3]) @ [1] =
- ((\text{poor\_rev}[3]) @ [2]) @ [1] =
- (((\text{poor\_rev}[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]`
Folding - Tail Recursion

```ml
let rev list = fold_left (fun l -> fun x -> x :: l) [] 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
Continuation Passing Style

- A programming technique for all forms of “non-local” control flow:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls

- Essentially it’s a higher-order function version of GOTO
Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an argument to which to pass its result; outer procedure “returns” no result
- Function receiving the result called a continuation
- Continuation acts as “accumulator” for work still to be done
Example of Tail Recursion

# let rec app fl x =
    match fl with [] -> x
    | (f :: rem_fs) -> f (app rem_fs x);

val app : ('a -> 'a) list -> 'a -> 'a = <fun>

# let app fs x =
    let rec app_aux fl acc =
        match fl with [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
            (fun z -> acc (f z))
        in app_aux fs (fun y -> y) x;;

val app : ('a -> 'a) list -> 'a -> 'a = <fun>
Continuation Passing Style

- Writing procedures so that they take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)
Example of Tail Recursion & CSP

```ocaml
# let app fs x =
    let rec app_aux fl acc =
        match fl with [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
            (fun z -> acc (f z))
    in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>

# let rec appk fl x k =
    match fl with [] -> k x
    | (f :: rem_fs) -> appk rem_fs x (fun z -> k (f z));;
val appk : ('a -> 'a) list -> 'a -> ('a -> 'b) -> 'b
```
Continuation Passing Style

- A compilation technique to implement non-local control flow, especially useful in interpreters.

- A formalization of non-local control flow in denotational semantics

- Possible intermediate state in compiling functional code
**Terms**

- A function is in **Direct Style** when it returns its result back to the caller.

- A **Tail Call** occurs when a function returns the result of another function call without any more computations (e.g., tail recursion).

- A function is in **Continuation Passing Style** when it passes its result to another function.

- Instead of returning the result to the caller, we pass it forward to another function.
Continuation Passing Style

- A compilation technique to implement non-local control flow, especially useful in interpreters.

- A formalization of non-local control flow in denotational semantics

- Possible intermediate state in compiling functional code
### Example

- **Simple reporting continuation:**
  ```ml
  # let report x = (print_int x; print_newline( ));;
  val report : int -> unit = <fun>
  ```

- **Simple function using a continuation:**
  ```ml
  # let plusk a b k = k (a + b)
  val plusk : int -> int -> (int -> 'a) -> 'a = <fun>
  # plusk 20 22 report;;
  42
  - : unit = ()
  ```
Simple Functions Taking Continuations

- Given a primitive operation, can convert it to pass its result forward to a continuation

- Examples:

```ocaml
# let subk x y k = k(x + y);;
val subk : int -> int -> (int -> 'a) -> 'a = <fun>
# let eqk x y k = k(x = y);;
val eqk : 'a -> 'a -> (bool -> 'b) -> 'b = <fun>
# let timesk x y k = k(x * y);;
val timesk : int -> int -> (int -> 'a) -> 'a = <fun>
```
Nesting Continuations

# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>

# let add_three x y z= let p = x + y in  p + z;;
val add_three : int -> int -> int -> int = <fun>

# let add_three_k x y z k =
  addk x y (fun p -> addk p z k);
val add_three_k : int -> int -> int -> (int -> 'a) -> 'a = <fun>