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

# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>
# plus_two 17;;
- : int = 19
# let plus_two = fun n -> n + 2;;
val plus_two : int -> int = <fun>
# plus_two 14;;
- : int = 16

First definition syntactic sugar for second
Closure for plus_x

- When plus_x was defined, had environment:
  \[ \rho_{\text{plus}_x} = \{..., \, x \rightarrow 12, \, ...\} \]

- Recall: let plus_x y = y + x
  is really let plus_x = fun y -> y + x

- Closure for fun y -> y + x:
  \[ <y \rightarrow y + x, \, \rho_{\text{plus}_x}> \]

- Environment just after plus_x defined:
  \[ \{\text{plus}_x \rightarrow <y \rightarrow y + x, \, \rho_{\text{plus}_x}>, \} + \rho_{\text{plus}_x} \]
Recall: let \( \text{plus}_x = \text{fun } x \Rightarrow y + x \)

- let \( x = 12 \)
  - \( X \rightarrow 12 \)
  - \( \ldots \)

- let \( \text{plus}_x = \text{fun } y \Rightarrow y + x \)
  - \( y \rightarrow y + x \)
  - \( x \rightarrow 12 \)

- let \( x = 7 \)
  - \( x \rightarrow 7 \)
  - \( \ldots \)
Functions on tuples

```ocaml
# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;
- : int = 7
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```
A closure is a pair of an environment and an association of a sequence of variables (the input variables) with an expression (the function body), written:

\(< (v_1, \ldots, v_n) \rightarrow \text{exp}, \rho >\)

Where \(\rho\) is the environment in effect when the function is defined (for a simple function)
Closure for `plus_pair`

- Assume $\rho_{plus\_pair}$ was the environment just before `plus_pair` defined.
- Closure for `fun (n,m) -> n + m`:
  
  $$\langle (n,m) \mapsto n + m, \rho_{plus\_pair} \rangle$$

- Environment just after `plus_pair` defined:
  
  $$\{-plus\_pair \mapsto \langle (n,m) \mapsto n + m, \rho_{plus\_pair} \rangle\} + \rho_{plus\_pair}$$
Functions with more than one argument

```ocaml
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let t = add_three 6 3 2;;
val t : int = 11
# let add_three =
    fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

Again, first syntactic sugar for second
Partial application of functions

```ocaml
define add_three (x y z) = x + y + z

let h = add_three 5 4
val h : int -> int = <fun>

h 3
- : int = 12

h 7
- : int = 16
```
Curried vs Uncurried

- Recall
  val add_three : int -> int -> int -> int = <fun>

- How does it differ from
  # let add_triple (u,v,w) = u + v + w;;
  val add_triple : int * int * int -> int = <fun>

- add_three is \textit{curried};
- add_triple is \textit{uncurried}
Curried vs Uncurried

# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;

Characters 0-10:
  add_triple 5 4;;
  ^^^^^^^^^^^^^

This function is applied to too many arguments, maybe you forgot a `;'
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
Functions as arguments

```ml
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```
Higher Order Functions

- A function is *higher-order* if it takes a function as an argument or returns one as a result.
- Example:

  ```ocaml
  # let compose f g = fun x -> f (g x);;
  val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
  ```

- The type ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b is a higher order type because of ('a -> 'b) and ('c -> 'a) and -> 'c -> 'b.
Thrice

- Recall:

```ocaml
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?
Thrice

- Recall:

```ocaml
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?

```ocaml
# let thrice f = compose f (compose f f);;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- Is this the only way?
Partial Application

# (+);;
- : int -> int -> int = <fun>

# (+) 2 3;;
- : int = 5

# let plus_two = (+) 2;;
val plus_two : int -> int = <fun>

# plus_two 7;;
- : int = 9

- Partial application also called *sectioning*
Partial Application and “Unknown Types”

- Recall `compose plus_two`:
  ```ocaml
  # let f1 = compose plus_two;;
  val f1 : ('_a -> int) -> '_a -> int = <fun>
  ```

- Compare to lambda lifted version:
  ```ocaml
  # let f2 = fun g ->  compose plus_two g;;
  val f2 : ('a -> int) -> 'a -> int = <fun>
  ```

- What is the difference?
Partial Application and “Unknown Types”

- ‘_a can only be instantiated once for an expression

```ocaml
# f1 plus_two;;
- : int -> int = <fun>
# f1 List.length;;
```

Characters 3-14:

```ocaml
f1 List.length;;
```

This expression has type 'a list -> int but is here used with type int -> int
Partial Application and “Unknown Types”

- ‘a can be repeatedly instantiated

```ocaml
# f2 plus_two;;
- : int -> int = <fun>
# f2 List.length;;
- : '_a list -> int = <fun>
```
Lambda Lifting

- You must remember the rules for evaluation when you use partial application

```ocaml
# let add_two = (+) (print_string "test\n"; 2);;

val add_two : int -> int = <fun>
```

```ocaml
# let add2 = (* lambda lifted *)
    fun x -> (+) (print_string "test\n"; 2) x;;

val add2 : int -> int = <fun>
```
Lambda Lifting

#  thrice add_two 5;;
- : int = 11
#  thrice add2 5;;
test
test
test
test
- : int = 11

- Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied
Evaluating declarations

- Evaluation uses an environment $\rho$
- To evaluate a (simple) declaration $\text{let } x = e$
  - Evaluate expression $e$ in $\rho$ to value $v$
  - Update $\rho$ with $x \mapsto v$: $\{x \mapsto v\} + \rho$

- Update: $\rho_1 + \rho_2$ has all the bindings in $\rho_1$ and all those in $\rho_2$ that are not rebound in $\rho_1$

\[
\{x \mapsto 2, y \mapsto 3, a \mapsto \text{“hi”}\} + \{y \mapsto 100, b \mapsto 6\}
\]

\[= \{x \mapsto 2, y \mapsto 3, a \mapsto \text{“hi”}, b \mapsto 6\}\]
Evaluating expressions

- Evaluation uses an environment $\rho$
- A constant evaluates to itself
- To evaluate an variable, look it up in $\rho (\rho(v))$
- To evaluate uses of +, -, etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: let $x = e1$ in $e2$
  - Eval $e1$ to $v$, eval $e2$ using $\{x \rightarrow v\} + \rho$
1. In environment $\rho$, evaluate the right term to values, $(v_1, \ldots, v_n)$

2. In environment $\rho$, evaluate left term to closure, $c = \langle(x_1, \ldots, x_n) \rightarrow b, \rho \rangle$

3. Match $(x_1, \ldots, x_n)$ variables in (first) argument with values $(v_1, \ldots, v_n)$

4. Update the environment $\rho$ to
   $\rho' = \{x_1 \rightarrow v_1, \ldots, x_n \rightarrow v_n\} + \rho$

5. Evaluate body $b$ in environment $\rho'$
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
Structural Recursion: List Example

```ocaml
# let rec length list = match list
    with [ ] -> 0 (* Nil case *)
    | x :: xs -> 1 + length xs;; (* Cons case *)

val length : 'a list -> int = <fun>
```

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

# 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) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
Question

- How do you write length with forward recursion?

``` Ocaml 
let rec length l = 
```
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?

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

9/6/16
Question

- How do you write length with forward recursion?

```ml
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 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

```ocaml
# let rec prod l =  
  match l with [] -> 1  
  | (x :: rem) -> x * prod rem;;
val prod : int list -> int = <fun>
# 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>
```
How do you write length with tail recursion?

```ocaml
let length l =
```
Question

How do you write length with tail recursion?

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

- How do you write length with tail recursion?

```ml
let length l =
  let rec length_aux list n =
    match list with [] ->
    | (a :: bs) ->
      n
    in
```
How do you write length with tail recursion?

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

9/6/16
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 in
```

9/6/16
How do you write length with tail recursion?

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

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

9/6/16
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
```

9/6/16
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
Mapping Recursion

- One common form of structural recursion applies a function to each element in the structure

```haskell
# let rec doubleList list = match list
  with [ ] -> [ ]
  | x::xs -> 2 * x :: doubleList xs;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```
Mapping 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]
```
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]
```

- Same function, but no rec
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 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 prodlist list = match list with
  [ ] -> 1 | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```
Iterating 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

```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

# 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]
```
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?
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 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

# 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>
```

```
<table>
<thead>
<tr>
<th>Init Acc Value</th>
<th>Recursive Call</th>
<th>Operation</th>
</tr>
</thead>
</table>
```

```ocaml
# 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?
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 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

```ml
# 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

- \text{rev} [1,2,3] = \\
- \text{rev\_aux} [1,2,3] [ ] = \\
- \text{rev\_aux} [2,3] [1] = \\
- \text{rev\_aux} [3] [2,1] = \\
- \text{rev\_aux} [ ] [3,2,1] = [3,2,1]
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
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

# 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
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 (eg 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.
Example

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

- Simple function using a continuation:
  
  ```ocaml
  # 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

```ocaml
# 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>
```