

Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

[http://www.cs.uiuc.edu/class
/sp07/cs421/](http://www.cs.uiuc.edu/class/sp07/cs421/)

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

Question

- Observation: Functions are first-class values in this language
- Question: What value does the environment record for a function variable?
- Answer: *a closure*

Save the Environment!

- 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:

$$f \rightarrow \langle (v_1, \dots, v_n) \rightarrow \text{exp}, \rho_f \rangle$$

- Where ρ_f is the environment in effect when f is defined (if f is a simple function)

Closure for plus_x

- When plus_x was defined, had environment

$$\rho_{\text{plus_x}} = \{x \rightarrow 12, \dots, y \rightarrow 24, \dots\}$$

- Closure for plus_x:

$$\langle y \rightarrow y + x, \rho_{\text{plus_x}} \rangle$$

Evaluation of Application

- First evaluate the left term to a function (ie starts with **fun**)
- Evaluate the right term (argument) to a value
 - Things starting with **fun** are values
- Substitute the argument for the formal parameter in the body of the function
- Evaluate resulting term
- (Need to use environments)

Evaluation of application of `plus_x;;`

Have environment

$$\rho = \{\text{plus_x} \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus_x}} \rangle, \dots, \\ y \rightarrow 3, \dots\}$$

where $\rho_{\text{plus_x}} = \{x \rightarrow 12, \dots, y \rightarrow 24, \dots\}$

Eval (plus_x y) in ρ rewrites to

Eval (app $\langle y \rightarrow y + x, \rho_{\text{plus_x}} \rangle$ 3) rewrites to

Eval (y + x) in $(y \rightarrow 3) + \rho_{\text{plus_x}}$ rewrites to

Eval (3 + 12) = 15

Scoping Question

Consider this code:

```
let x = 27;;  
let f x =  
    let x = 5 in  
        (fun x -> print_int x) 10;;  
f 12;;
```

What value is printed?

- a) 5
- b) 10
- c) 12
- d) 27

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 *)  
(* More on this later *)
```

Tuples and Patterns

```
# let s = (5,"hi",3.2);;  
val s : int * string * float = (5, "hi", 3.2)  
# let (a,b,c) = s;;  
val a : int = 5  
val b : string = "hi"  
val c : float = 3.2  
# let x = 2, 9.3;; (* tuples don't require parens*)  
val x : int * float = (2, 9.3)
```

Tuples

```
# let d = ((1,4,62),("bye",15),73.95);;  
val d : (int * int * int) * (string * int) * float =  
      ((1, 4, 62), ("bye", 15), 73.95)  
# let (p,(st,_),_) = d;;  
val p : int * int * int = (1, 4, 62)  
val st : string = "bye"
```

Tuples

```
# let fst_of_3 (x,_,_) = x;;  
val fst_of_3 : 'a * 'b * 'c -> 'a = <fun>  
# s;;  
- : int * string * float = (5, "hi", 3.2)  
# fst_of_3 s;;  
- : int = 5  
# fst_of_3 d;;  
- : int * int * int = (1, 4, 62)
```

Notice the type of `fst_of_3`

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 *curried*;

add_triple is *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>
```

Match Expressions

```
# let triple_to_pair triple =
```

```
  match triple
```

```
  with (0, x, y) -> (x, y)
```

```
  | (x, 0, y) -> (x, y)
```

```
  | (x, y, 0) -> (x, y)
```

```
  | (x, y, _) -> (x, y);;
```

```
val triple_to_pair : int * int * int -> int * int =
```

```
  <fun>
```

- Each clause: pattern on left, expression on right
- Each x, y has scope of only its clause
- Use first matching clause

Lists

- First example of a recursive datatype (aka algebraic datatype)
- Unlike tuples, lists are homogenous in type (all elements same type)

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: $[x] == x :: []$
 $[x1; x2; \dots; xn] == x1 :: x2 :: \dots :: xn :: []$

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 Homogenous

```
# 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"]
```

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]
```

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 = ()
```

Iterating over lists

```
# 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 = ()
```

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

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

```
# 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, you split your input into components
- In forward recursion, you first call the function recursively on all the recursive components, and then build the final result from the partial results
- Wait until the whole structure has been traversed to start building the 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>
```

Mapping Recursion

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

```
# 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 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 rec

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

- 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
```

How long will it take?

- Remember the big-O notation from CS 225 and CS 273
- 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 \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:

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

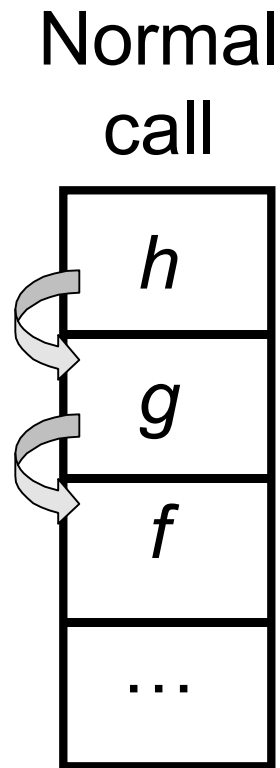
Exponential running time

- Hideous 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

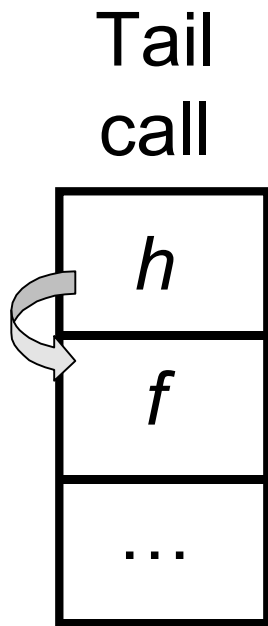
```
# let rec naiveFib n = match n
  with 0 -> 0
      | 1 -> 1
      | _ -> naiveFib (n-1) + naiveFib (n-2);;
val naiveFib : int -> int = <fun>
```

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

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

- $\text{poor_rev } [1,2,3] =$
- $(\text{poor_rev } [2,3]) @ [1] =$
- $((\text{poor_rev } [3]) @ [2]) @ [1] =$
- $(((\text{poor_rev } []) @ [3]) @ [2]) @ [1] =$
- $(([] @ [3]) @ [2]) @ [1] =$
- $[3] @ [2] @ [1] =$
- $3 :: ([] @ [2]) @ [1] =$
- $[3,2] @ [1] =$
- $3 :: ([2] @ [1]) =$
- $3 :: (2 :: ([] @ [1])) = [3, 2, 1]$

Comparison

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