

## CS421 Topic 3: Patterns of Recursion<sup>1</sup>

Sameer Sundresh  
sundresh@uiuc.edu

University of Illinois at Urbana-Champaign

May 31, 2007

<sup>1</sup>Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, Elsa Gunter, and Mark Hills

### Overview

Recursion lets you operate on arbitrary sized data. Pretty handy.

- ▶ Recursion & induction
- ▶ Common patterns of recursion: *Iterating, Mapping, Folding (or Reductions)*
- ▶ Complexity of recursive operations
- ▶ Make it practical: tail call optimization!

### Overview

#### Recursion and Lists

#### Complexity

#### Accumulating Recursion

#### Activities

### Recursion Example

Compute  $n^2$  recursively using formula:  $n^2 = (2n - 1) + (n - 1)^2$ :

```

1 # let rec nthsq n =
2   match n with
3   | 0 -> 0
4   | n -> (2*n - 1)
5     + nthsq (n-1);; (* recursive case *)
6 val1 nthsq : int -> int = <fun>
7 # nthsq 3;;
8 - : int = 9

```

- ▶ Recursive functions must be declared with a rec keyword.
- ▶ Structure of a recursive function is similar to proof-by-induction

### Recursion and Induction

```

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

```

- ▶ The base case is the last case: it stops the computation.
- ▶ The recursive case calls the same function with a *smaller* argument (by some measure) than the current call  
Key: *Must take progress towards base case*
- ▶ The if or match statement has to be able to tell when the base case is reached.
- ▶ Failure to do any of the above will cause failure to terminate.

### Structural Recursion

Use match + recursion to traverse a data structure.

- ▶ Functions on recursively defined data types (lists, trees, etc) are generally recursive
- ▶ Recursion over recursive datatypes generally by *structural recursion*
  - ▶ Recursive calls made to sub-components of same recursive type
  - ▶ Base case (empty list, tree leaf, etc) stops recursion

## Lists

## Activity 1

```
1 # let rec length lst = match lst with
2 | [] -> 0
3 | x::xs -> 1 + length xs;;
4 val length : 'a list -> int = <fun>
5 # length [2;3;4;6];;
6 - : int = 4
```

- ▶ The base case `[]` stops the computation.
- ▶ Your recursive case calls itself with a *smaller* argument (`xs`) than the original call

## Forward Recursion

### Recurse—Wait for result—Finish up

- ▶ In recursion, split the input into the “first piece” and the “rest of the input”.
- ▶ In forward recursion: recursive call computes the result for rest of input; then combine result with the first piece.

```
1 # let rec badReverse lst = match lst with
2 | [] -> []
3 | x::xs -> (badReverse xs) @ [x];;
4 val badReverse : 'a list -> 'a list = <fun>
```

(108) Write an OCaml function that returns the maximum element of a list. (Assume the list always has at least 1 element, and assume you have a `max` function.)

## Lists

## Activity 1

```
1 # let rec length lst = match lst with
2 | [] -> 0
3 | x::xs -> 1 + length xs;;
4 val length : 'a list -> int = <fun>
5 # length [2;3;4;6];;
6 - : int = 4
```

- ▶ The base case `[]` stops the computation.
- ▶ Your recursive case calls itself with a *smaller* argument (`xs`) than the original call

## Forward Recursion

### Recurse—Wait for result—Finish up

- ▶ In recursion, split the input into the “first piece” and the “rest of the input”.
- ▶ In forward recursion: recursive call computes the result for rest of input; then combine result with the first piece.

```
1 # let rec badReverse lst = match lst with
2 | [] -> []
3 | x::xs -> (badReverse xs) @ [x];;
4 val badReverse : 'a list -> 'a list = <fun>
```

## Mapping Recursion

Apply a function to each element of a data structure.

```
1 # let rec doubleList lst = match lst with
2 | [] -> []
3 | x::xs -> 2 * x :: doubleList xs;;
4 val doubleList : int list -> int list = <fun>
5 # doubleList [4;6;8];;
6 - : int list = [8; 12; 16]
```

## Generic Maps

Map on lists of arbitrary type!

```
1 # let rec map f l =
2   match l with
3   | [] -> []
4   | (x :: xs) -> (f x) :: (map f xs) ;;
5 val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
6 # let double n = 2 * n ;;
7 val double : int -> int = <fun>
8 # map double [1;2;3;4;5];;
9 - : int list = [2; 4; 6; 8; 10]
```

## List.map

OCaml built-in map function

```
1 # List.map double [1;2;3;4;5];;
2 - : int list = [2; 4; 6; 8; 10]
3 # let doubleList lst = List.map double lst;;
4 val doubleList : int list -> int list = <fun>
5 # doubleList [1;2;3;4;5];;
6 - : int list = [2; 4; 6; 8; 10]
```

## Lists

## Activity 1

```
1 # let rec length lst = match lst with
2 | [] -> 0
3 | x::xs -> 1 + length xs;;
4 val length : 'a list -> int = <fun>
5 # length [2;3;4;6];;
6 - : int = 4
```

- ▶ The base case `[]` stops the computation.
- ▶ Your recursive case calls itself with a *smaller* argument (`xs`) than the original call

## Forward Recursion

### Recurse—Wait for result—Finish up

- ▶ In recursion, split the input into the “first piece” and the “rest of the input”.
- ▶ In forward recursion: recursive call computes the result for rest of input; then combine result with the first piece.

```
1 # let rec badReverse lst = match lst with
2 | [] -> []
3 | x::xs -> (badReverse xs) @ [x];;
4 val badReverse : 'a list -> 'a list = <fun>
```

## Mapping Recursion

Apply a function to each element of a data structure.

```
1 # let rec doubleList lst = match lst with
2 | [] -> []
3 | x::xs -> 2 * x :: doubleList xs;;
4 val doubleList : int list -> int list = <fun>
5 # doubleList [4;6;8];;
6 - : int list = [8; 12; 16]
```

## Generic Maps

Map on lists of arbitrary type!

```
1 # let rec map f l =
2   match l with
3   | [] -> []
4   | (x :: xs) -> (f x) :: (map f xs) ;;
5 val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
6 # let double n = 2 * n ;;
7 val double : int -> int = <fun>
8 # map double [1;2;3;4;5];;
9 - : int list = [2; 4; 6; 8; 10]
```

## List.map

OCaml built-in map function

```
1 # List.map double [1;2;3;4;5];;
2 - : int list = [2; 4; 6; 8; 10]
3 # let doubleList lst = List.map double lst;;
4 val doubleList : int list -> int list = <fun>
5 # doubleList [1;2;3;4;5];;
6 - : int list = [2; 4; 6; 8; 10]
```

## Folding Recursion

Apply a *binary* function across a list.

```
fold_left (+) init [1; 2; 3] => ((init + 1) + 2) + 3
fold_right (+) init [1; 2; 3] => 1 + (2 + (3 + init))
```

## Folding Example

```
1 # let rec multList lst = match lst with
2 | [] -> 1
3 | x::xs -> x * multList xs;;
4 val multList : int list -> int = <fun>
5 # multList [2;4;6];;
6 - : int = 48
```

This computes  $(2 * (4 * (6 * 1)))$ .

## Generic Folds: Fold Left

Apply f, update i, then recurse

```
1 # let rec fold_left f i l =
2   match l with
3   | [] -> i
4   | (x :: xs) -> fold_left f (f i x) xs;;
5 val fold_left :
6 ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
7 # fold_left ( * ) 1 [2;4;6] ;;
8 - : int = 48
```

## Generic Folds: Fold Right

Recurse, then apply f.

```
1 # let rec fold_right f l i =
2   match l with
3   | [] -> i
4   | (x :: xs) -> f x (fold_right f xs i);;
5 val fold_right :
6 ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
7 # fold_right ( * ) [2;4;6] 1 ;;
8 - : int = 48
```

## List.folds

Folds are also defined in the standard libraries.

```
1 # List.fold_left ( * ) 1 [2;4;6] ;;
2 - : int = 48
3 # List.fold_right ( * ) [2;4;6] 1 ;;
4 - : int = 48
```

## How long will it take?

For an input of size  $n$ ,  
how long will it take to generate the output?

- Common big- $\mathcal{O}$  times for recursive computations:
- ▶ Constant time  $\mathcal{O}(1)$  — input size doesn't matter
  - ▶ Linear time  $\mathcal{O}(n)$  — double input  $\Rightarrow$  double time
  - ▶ Quadratic time  $\mathcal{O}(n^2)$  — double input  $\Rightarrow$  quadruple time
  - ▶ Exponential time  $\mathcal{O}(2^n)$  — increment input  $\Rightarrow$  double time

Review big- $\mathcal{O}$  notation from CS 225 and CS 273.

## How long will it take?

## Linear Time

- ▶ Expect most list operations to take linear time ( $\mathcal{O}(n)$ ).
- ▶ Each step of the recursion can be done in constant time.
- ▶ Each step makes exactly one recursive call.
- ▶ List example: `multList`, `append`
- ▶ Integer example: `factorial`

## Quadratic Time

- ▶ Each step of the recursion takes time proportional to the input.
- ▶ Each step of the recursion makes exactly one recursive call.
- ▶ List example: badly written "reverse".
- ▶ Integer example: Twelve days of Christmas.

```
1 # let rec badReverse lst = match lst with
2 | [] -> []
3 | x::xs -> (badReverse xs) @ [x];;
4 val badReverse : 'a list -> 'a list = <fun>
```

## Exponential Time

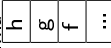
- ▶ Hideous running times.
- ▶ Each step of the recursion takes constant time.
- ▶ But each recursion makes two recursive calls.
- ▶ Worst part: it is very simple to make a linear function into an exponential one!
- ▶ Examples: naïve Fibonacci sequence (1,1,2,3,5,8,13,21,34,...)

```
1 # let rec badFib n = match n with
2 | 0 -> 0
3 | 1 -> 1
4 | _ -> badFib (n-1) + badFib (n-2);;
5 val badFib : int -> int = <fun>
```

## An important optimization: Tail Recursion

- ▶ When you make a function call, you have to save the return address on the stack, so we know where to return
- ▶ Suppose  $f$  calls  $g$ , and then  $g$  calls  $h$ . What if the call to  $h$  was the very last thing  $g$  did?
- ▶ Such a call is called a *tail call*. We don't need to save the stack frame of the function making the tail call in such a case.
- ▶ This optimization can allow recursive programs to run with the same efficiency as imperative programs.
- ▶ Modern optimizing compilers will usually do this for you

Normal call:



$g$  tail calls  $h$ :



## 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, which may require an auxiliary function

## Identifying Tail Calls

(7) Which of the following functions is tail recursive?

```
1 let rec foo lst = match lst with
2 | [] -> 1
3 | x::xs -> let r = foo xs in r * x

1 let rec foo lst = match lst with
2 | [] -> 1
3 | x::xs -> x * (foo xs)

1 let rec foo lst acc = match lst with
2 | [] -> acc
3 | x::xs -> foo xs (x * acc)
```

## Accumulating Recursion

- ▶ In accumulating recursion: generate an intermediate result now, and give that to the recursive call.
- ▶ May be referred to as “iterative” behavior

```

1 # let rec goodRevAux lst acc = match lst with
2 | [] -> acc
3 | x::xs -> goodRevAux xs (x::acc);;
4 (* notice that this one is tail recursive! *)
5 val goodRevAux : 'a list -> 'a list -> 'a list = <fun>
6 # let goodReverse lst = goodRevAux lst [];
7 val foo : 'a list -> 'a list = <fun>

```

### What is the running time?

## Comparison: Bad Reverse

```

1 badReverse [1;2;3] =
2 (badReverse [2;3]) @ [1] =
3 ((badReverse [3]) @ [2]) @ [1] =
4 (((badReverse []) @ [3]) @ [2]) @ [1] =
5 (([] @ [3]) @ [2]) @ [1] =
6 ([3] @ [2]) @ [1] =
7 (3 :: ([] @ [2])) @ [1] =
8 [3;2] @ [1] =
9 3 :: (2 :: ([] @ [1])) =
10 3 :: 2 :: ([] @ [1]) =
11 [3;2;1]

```

## Comparison: Good Reverse

```

1 goodReverse [1;2;3] =
2 goodReverseAux [4;2;3] [] =
3 goodReverseAux [2;3] [1] =
4 goodReverseAux [3] [2;1] =
5 goodReverseAux [] [3;2;1] =
6 [3;2;1]

```

## Activity 2

(109) Write a function that returns the maximum element of a list, using tail recursion. (Assume the list always has at least 1 element, and assume you have a max function.)

## Problem 1

Write a function that returns the maximum element of a list. Use forward recursion. (Assume the list always has at least 1 element, and assume you have a max function.)

```

1 # let rec maxlist lst = match lst with
2 | [x] -> x
3 | x::xs -> max x (maxlist xs);;
4 Warning: this pattern-matching is not exhaustive.
5 Here is an example of a value that is not matched:
6 []
7 val maxlist : 'a list -> 'a = <fun>

```

## Problem 2

Write a function that returns the maximum element of a list, using tail recursion. (Assume the list always has at least 1 element, and assume you have a max function.)

```

1 # let rec maxlistaux lst a = match lst with
2 | [] -> a
3 | x::xs -> maxlistaux xs (max x a);;
4 val maxlistaux : 'a list -> 'a -> 'a = <fun>
5 # let maxlist (x::xs) = maxlistaux xs x;;
6 Warning: this pattern-matching is not exhaustive.
7 Here is an example of a value that is not matched:
8 []
9 val maxlist : 'a list -> 'a = <fun>

```