Recursion

• A way of life
• A different way to view computation
  - Solutions for bigger problems
  - From solutions for sub-problems

Why know about it?
1. Often far simpler, cleaner than loops
   - But not always...
2. Forces you to factor code into reusable units
   - Only way to “reuse” loop is via cut-paste
Q: What does this evaluate to?

```ocaml
let rec foo i j =
  if i >= j then []
  else i::(foo (i+1) j)
in foo 0 3
```

(a) [0;1;2]
(b) [0;0;0]
(c) []
(d) [2;2;2]
(e) [2;1;0]
Q: What does this evaluate to?

```ocaml
let rec range i j =  
  if i >= j then []  
  else i :: (range (i+1) j)
```

range 3 3 ===> []
range 2 3 ===> 2 :: (range 3 3) ===> 2 :: []
range 1 3 ===> 1 :: (range 2 3) ===> 1 :: 2 :: []
range 0 3 ===> 0 :: (range 1 3) ===> 0 :: 1 :: 2 :: []
Q: What does this evaluate to?

```
let rec range i j =
  if i >= j then []
  else i:::(range (i+1) j)
```

Tail Recursive?
Q: What does this evaluate to?

let range lo hi =
  let rec helper res j =
    if lo >= j then res
    else helper (j::res)(j-1)
in helper [] hi

Tail Recursive!
Moral of the day...

Recursion good...
...but HOFs better!
News

• PA2 due **tonight** @ 11:59:59 pm

• PA3 goes up soon

• Midterm **Fri 5/8**
  - In class
  - Open book etc.
  - Practice materials on webpage
Today’s Plan

• A little more practice with recursion
  - Base Pattern -> Base Expression
  - Induction Pattern -> Induction Expression

• Higher-Order Functions
  - or, why “take” and “return” functions?
Write: evens

(* val evens: int list -> int list *)
let rec evens xs = match xs with
| []  -> ... 
| x::xs' -> ...  

evens [] ===> []
evens [1;2;3;4] ===> [2;4]
Write: evens

(* val evens: int list -> int list *)

let rec evens xs = match xs with
  | [] -> []
  | x::xs' -> if x mod 2 = 0 then x::(evens xs') else (evens xs')

evens [] ===> []
evens [1;2;3;4] ===> [2;4]
(* fourLetters: string list -> string list *)

let rec fourLetters xs = match xs with
  | [] -> ...
  | x::xs' -> ...

fourLetters []
  =====> []

fourLetters ["cat";"must";"do";"work"]
  =====> ["must"; "work"]
(* fourLetters: string list -> string list *)
let rec fourLetters xs = match xs with
| [] -> []
| x::xs' -> if length x = 4
  then x::(fourLetters xs')
  else (fourLetters xs')

fourLetters []
==> []
fourLetters ["cat";"must";"do";"work"]
==> ["must"; "work"]
Yuck! Most code is same!
Yuck! Most code is same!
Moral of the Day…

“D.R.Y”
Don’t Repeat Yourself!
Moral of the Day...

HOFs Allow “Factoring”

General “Pattern”

+ Specific “Operation”
let rec evens xs =
      match xs with
      | []     -> []
      | x::xs' -> if \text{x mod 2 = 0} \then x::(foo xs')
                   \else (foo xs')

let rec filter f xs =
      match xs with
      | []     -> []
      | x::xs' -> if f \text{x} \then x::(filter xs')
                   \else (filter xs')

letrec fourLetters xs =
      match xs with
      | []     -> []
      | x::xs' -> if \text{length x = 4} \then x::(foo xs')
                   \else (foo xs')

The “filter” pattern
let rec evens xs =
  match xs with
  | [] -> []
  | x::xs' -> if x mod 2 = 0
    then x::(foo xs')
    else (foo xs')

let evens xs =
  filter (fun x -> x mod 2 = 0) xs

let rec fourLetters xs =
  match xs with
  | [] -> []
  | x::xs' -> if length x = 4
    then x::(foo xs')
    else (foo xs')

let fourLetters xs =
  filter (fun x -> length x = 4) xs

let rec filter f xs =
  match xs with
  | [] -> []
  | x::xs' -> if f x
    then x::(filter xs')
    else (filter xs')

The “filter” pattern
Factor Into Generic + Specific

Specific Operations

\[
\begin{align*}
\text{let evens } &\text{ xs = filter } (\text{fun } x \rightarrow x \text{ mod } 2 = 0) \text{ xs} \\
\text{let fourLetter } &\text{ xs = filter } (\text{fun } x \rightarrow \text{length } x = 4) \text{ xs}
\end{align*}
\]

Generic “filter” pattern

\[
\begin{align*}
\text{let rec filter } f \text{ xs }= \\
\text{ match } xs \text{ with} \\
\text{ | [] } &\rightarrow \text{ []} \\
\text{ | x::xs’ } &\rightarrow \text{ if } f x \\
&&\text{ then x::(filter xs’)} \\
&&\text{ else (filter xs’)}
\end{align*}
\]
Write: listUpper

(* string list -> string list *)

let rec listUpper xs =
  match xs with
  | [] -> ...
  | x::xs' -> ...

listUpper [] ===> []

listUpper [“carne”; “asada”] ===> [“CARNE”; “ASADA”]
Write: listUpper

(* string list -> string list *)

let rec listUpper xs =
  match xs with
  | [] -> []
  | x::xs' -> (uppercase x)::(listUpper xs')

listUpper [] ===> []

listUpper ["carne"; "asada"] ===> ["CARNE"; "ASADA"]
Write: listSquare

(* int list -> int list *)
let rec listSquare xs =
  match xs with
  | [] -> ...  (* default case *)
  | x::xs' -> ...

listSquare [] ===> []
listSquare [1;2;3;4;5] ===> [1;4;9;16;25]
Write: listSquare

(* int list -> int list *)

let rec listSquare xs =
  match xs with
  | [] -> []
  | x::xs' -> (x*x)::(listSquare xs')

listSquare [] ===> []
listSquare [1;2;3;4;5] ===> [1;4;9;16;25]
let rec listUpper xs =
    match xs with
    | []    -> []
    | x::xs'->(uppercase x)::(listUpper xs')

let rec listSquare xs =
    match xs with
    | []    -> []
    | x::xs'-> (x*x)::(listSquare xs')
What’s the Pattern?

```ocaml
let rec listUpper xs =
  match xs with
  | []    -> []
  | x::xs'->(uppercase x)::(listUpper xs')
```

```ocaml
let rec listSquare xs =
  match xs with
  | []    -> []
  | x::xs'-> (x*x)::(listSquare xs')
```
What’s the Pattern?

let rec listUpper xs =
    match xs with
    | [] -> []
    | x::xs'->(uppercase x)::(listUpper xs')

let rec listSquare xs =
    match xs with
    | [] -> []
    | x::xs'-> (x*x)::(listSquare xs')
“Refactor” Pattern

```ocaml
let rec listSquare xs = 
  match xs with
  | []   -> []
  | x::xs'-> (x*x)::(listSquare xs')

let rec listUpper xs = 
  match xs with
  | []   -> []
  | x::xs'->(uppercase x)::(listUpper xs')

let rec pattern ...
```
let rec listUpper xs =
  match xs with
  | [] -> []
  | x::xs' -> (uppercase x)::(listUpper xs')

let rec listSquare xs =
  match xs with
  | [] -> []
  | x::xs' -> (x*x)::(listSquare xs')

let rec map f xs =
  match xs with
  | [] -> []
  | x::xs' -> (f x)::(map f xs')
“Refactor” Pattern

```ocaml
define listUpper recursively as:
match xs with
| [] -> []
| x::xs' -> (uppercase x)::(listUpper xs')

let listUpper xs = map (fun x -> uppercase x) xs

define map recursively as:
match xs with
| [] -> []
| x::xs' -> (f x)::(map f xs')
```
“Refactor” Pattern

```ocaml
let rec listUpper xs =
  match xs with
  | [] -> []
  | x::xs' -> (uppercase x)::(listUpper xs')

let listUpper = map uppercase
```

```ocaml
let rec map f xs =
  match xs with
  | [] -> []
  | x::xs' -> (f x)::(map f xs')
```
“Refactor” Pattern

```ocaml
let rec listSquare xs =
  match xs with
  | [] -> []
  | x::xs' -> (x*x)::(listSquare xs')

let rec map f xs =
  match xs with
  | [] -> []
  | x::xs' -> (f x)::(map f xs')
```

```ocaml
let listSquare = map (fun x -> x*x)
```
Factor Into Generic + Specific

let listSquare = map \( \text{fun } x \rightarrow x \times x \) 

let listUpper = map \text{uppercase} 

let rec map \( f \) \( xs \) = 
match \( xs \) with 
| [] \rightarrow [] 
| \( x::xs' \) \rightarrow (\( f \) \( x \))::(map \( f \) \( xs' \))
Moral of the Day…

“D.R.Y”
Don’t Repeat Yourself!
Q: What is the type of map?

let rec map f xs =
  match xs with
  | [] -> []
  | x::xs' -> (f x)::(map f xs')

(a) (`a -> `b) -> `a list -> `b list
(b) (int -> int) -> int list -> int list
(c) (string -> string) -> string list -> string list
(d) (`a -> `a) -> `a list -> `a list
(e) (`a -> `b) -> `c list -> `d list
Q: What is the type of map?

```
let rec map f xs =
  match xs with
  | [] -> []
  | x::xs' -> (f x)::(map f xs')
```

(a) (`a -> `b) -> `a list -> `b list

Type says it all!

- Apply “f” to each element in input list
- Return a list of the results
Q: What does this evaluate to?

map (fun (x, y) -> x+y) [1;2;3]

(a) [2;4;6]
(b) [3;5]
(c) Syntax Error
(e) Type Error
Don’t Repeat Yourself!

“Factored” code:

- Reuse iteration template
- Avoid bugs due to repetition
- Fix bug in one place!
Don’t Repeat Yourself!

Made Possible by **Higher-Order Functions**!
Recall: len

\[
\text{let rec } \text{len } \text{xs } = \\
\text{match } \text{xs } \text{with} \\
| [] -> 0 \\
| \text{x::xs'} -> 1 + \text{len } \text{xs'}
\]

\text{len } [] \implies 0

\text{len } \text{[“carne”; “asada”]} \implies 2
Recall: sum

\[ \text{let rec sum \ xs =} \]
\[ \begin{align*}
\text{match } \ \text{xs} \ \text{with} \\
| \ [ ] & \rightarrow 0 \\
| \ x::\text{xs'} & \rightarrow x + \text{len} \ \text{xs'}
\end{align*} \]

\[
\begin{align*}
\text{sum } [ ] & \quad \Rightarrow 0 \\
\text{sum } [10; 20; 30] & \quad \Rightarrow 60
\end{align*}
\]
Write: concat

(* string list -> string *)
let rec concat xs =
  match xs with
  | [] -> ...
  | x::xs' -> ...

concat []
  ===> “”

concat [“carne”; “asada”; “torta”]
  ===> “carneasadatorta”
Write: concat

(* string list -> string *)

let rec concat xs =
    match xs with
    | [] -> ""
    | x::xs' -> x^(concat xs')

concat []
  ===> ""

concat ["carne"; "asada"; "torta"]
  ===> "carneasadatorta"
What’s the Pattern?

let rec len xs =
  match xs with
  | []   -> 0
  | x::xs'-> 1 + (len xs')

let rec sum xs =
  match xs with
  | []   -> 0
  | x::xs'-> x + (sum xs')

let rec concat xs =
  match xs with
  | []   -> ""
  | x::xs'-> x^(concat xs')
What’s the Pattern?

let rec len xs =
  match xs with
  | []   -> 0
  | x::xs'-> 1 + (len xs')

let rec sum xs =
  match xs with
  | []   -> 0
  | x::xs'-> x + (sum xs')

let rec foldr f b xs =
  match xs with
  | []   -> b
  | x::xs'-> f x (foldr f b xs')

let rec concat xs =
  match xs with
  | []   -> 
  | x::xs'-> x^(concat xs')
let rec foldr f b xs =
    match xs with
    | []       -> b
    | x::xs'   -> f x (foldr f b xs')

let rec len xs =
    match xs with
    | []       -> 0
    | x::xs'   -> 1 + (len xs')

let rec sum xs =
    match xs with
    | []       -> 0
    | x::xs'   -> x + (sum xs')

let rec concat xs =
    match xs with
    | []       -> ""
    | x::xs'   -> x^(concat xs')
let rec foldr f b xs =
match xs with
  | []  -> b
  | x::xs'-> f x (foldr f b xs')

let len =
foldr (fun x n -> n+1) 0

let sum =
foldr (fun x n -> x+n) 0

let concat =
foldr (fun x n -> x^n) ""
Q: What does this evaluate to? 

foldr (fun x n -> x::n) [] [1;2;3]

(a) [1;2;3]
(b) [3;2;1]
(c) []
(d) [[3];[2];[1]]
(e) [[1];[2];[3]]
“fold-right” pattern

```ocaml
let rec foldr f b xs =
  match xs with
  | [] -> b
  | x::xs' -> f x (foldr f b xs')

foldr f b [x1;x2;x3]
  ===> f x1 (foldr f b [x2;x3])
  ===> f x1 (f x2 (foldr f b [x3]))
  ===> f x1 (f x2 (f x3 (foldr f b [])))
  ===> f x1 (f x2 (f x3 (foldr f b [])))
  ===> f x1 (f x2 (f x3 (b)))
```
The “fold” Pattern

let rec foldr \texttt{f} \texttt{b} \texttt{xs} =
match \texttt{xs} with
| []  -> \texttt{b}
| \texttt{x::xs'} -> \texttt{f} \texttt{x} (foldr \texttt{f} \texttt{b} \texttt{xs'})

Tail Recursive?
The “fold” Pattern

let rec foldr \( f \) \( b \) \( xs \) = 
match \( xs \) with 
| [] \rightarrow \( b \) 
|x::xs'\rightarrow \( f \) x (foldr \( f \) \( b \) xs')

Tail Recursive? No!
Write: concat (TR)

```plaintext
let concat xs = ...
```

concat []

```plaintext
    ===> ""
```

concat ["carne"; "asada"; "torta"]

```plaintext
    ===> "carneasedatortas"
```
Write: concat

```ocaml
let concat xs =
  let rec helper res = function
    | [] -> res
    | x::xs'-> helper (res^x) xs'
  in helper "" xs

helper "" ["carne"; "asada"; "torta"]
 ===> helper "carne" ["asada"; "torta"]
 ===> helper "carneasada" ["torta"]
 ===> helper "carneasadatorta" []
 ===> "carneasadatorta"
```
Write: sum (TR)

```
let sum xs = ...
```

```
sum [] ===> 0

sum [10;20;30] ===> 60
```
Write: concat

```ocaml
let sum xs =
  let rec helper res = function
  | []     -> res
  | x::xs' -> helper (res+x) xs'
  in  helper 0 xs

helper 0 [10; 100; 1000]
=====> helper 10 [100; 1000]
=====> helper 110 [1000]
=====> helper 1110 []
=====> 1110
```
What’s the Pattern?

```plaintext
let sum xs = 
  let rec helper res = function 
    | []  -> res 
    | x::xs'-> helper (res + x) xs'
  in helper 0 xs

let sum xs = 
  foldl (fun res x -> res + x) 0

let concat xs = 
  let rec helper res = function 
    | []  -> res 
    | x::xs'-> helper (res ^ x) xs'
  in helper "" xs

let sum xs = 
  foldl (fun res x -> res ^ x) ""
```

```plaintext
let foldl f b xs = 
  let rec helper res = function 
    | []  -> res 
    | x::xs'-> helper (f res x) xs'
  in helper b xs
```
let foldl f b xs =
    let rec helper res = function
        | [] -> res
        | x::xs' -> helper (f res x) xs'
    in helper b xs

let sum xs =
    foldl (fun res x -> res + x) 0

let sum xs =
    foldl (fun res x -> res ^ x) ""
Q: What does this evaluate to?

\[
\text{foldl (fun res x -> x::res)} \; [\;] \; [1;2;3]
\]

(a) [1;2;3]
(b) [3;2;1]
(c) []
(d) [[3];[2];[1]]
(e) [[1];[2];[3]]
Funcs taking/returning funcs

Identify common computation “patterns”
• **Filter** values in a set, list, tree ...

• **Iterate** a function over a set, list, tree ...

• **Accumulate** some value over a collection

Pull out (factor) “common” code:
• **Computation Patterns**
• **Re-use** in many different situations
Another fun function: “pipe”

```ocaml
let pipe x f = f x
```

```ocaml
let (|>) x f = f x
```

Compute the sum of squares of numbers in a list?

```ocaml
let sumOfSquares xs =
    xs |> map (fun x -> x * x)
    |> foldl (+) 0
```

Tail Rec ?
Funcs taking/returning funcs

Identify common computation “patterns”
- **Filter** values in a set, list, tree ...
- **Convert** a function over a set, list, tree ...
- **Iterate** a function over a set, list, tree ...
- **Accumulate** some value over a collection

Pull out (factor) “common” code:
- **Computation Patterns**
- **Re-use** in many different situations
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions

Using, (Taking) Functions

Iterator, Accumul, Reuse computation pattern w/o exposing local info
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions

Using, (Taking) Functions

Compose Functions:
Flexible way to build Complex functions from primitives.

Iterator, Accumul, Reuse computation pattern w/o exposing local info
Higher-order funcs enable modular code
• Each part only needs local information

Data Structure

Client
Uses list

Uses meta-functions:
map, fold, filter
With locally-dependent funs
(lt h), square etc.
Without requiring Implement.
details of data structure

Provides meta-functions:
map, fold, filter
to traverse, accumulate over lists, trees etc.
Meta-functions don’t need client info (tester ? accumulator ?)