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?

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

Tail Recursive?

Moral of the day...

Recursion good...

...but HOFS better!

Q: What does this evaluate to?

```ocaml
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!

News

- PA2 due tomorrow @ 5PM
- PA3 goes up tomorrow
  - Due 4/20
- Midterm 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: fourLetters

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

fourLetters [] ===> []
fourLetters [“cat”;“must”;“do”;“work”] ===> [“must”; “work”]
Write: evens

(* 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"]

(* 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')

(* 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')

Yuck! Most code is same!

Moral of the Day...

“D.R.Y”
Don’t Repeat Yourself!

Yuck! Most code is same!
Moral of the Day...

HOFs Allow “Factoring”

General “Pattern”
+
Specific “Operation”

The “filter” pattern

Factor Into Generic + Specific

Specific Operations
Write: listUpper

(* string list -> string list *)

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

listUpper [] ===> []
listUpper ["carne"; "asada"] ===> ["CARNE"; "ASADA"]

Write: listSquare

(* int list -> int list *)

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

listSquare [] ===> []
listSquare [1;2;3;4;5] ===> [1;4;9;16;25]
Yuck! Most code is same!

```
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')
```

What’s the Pattern?

```
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')
```

“Refactor” 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 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') 

let listUpper = map uppercase 

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

Generic “iteration” pattern

let listSquare = map (fun x -> x * x)
let listUpper  = map uppercase

Specific Op

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

Moral of the Day...

“D.R.Y”
Don’t Repeat Yourself!

Q: What is the type of map?

(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

Type says it all!
• Apply “f” to each element in input list
• Return a list of the results
Q: What does this evaluate to?

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

(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!

Recall: \(\text{len}\)

\[
\text{let rec len xs =}
\begin{align*}
\text{match xs with} \\
\quad &| \[] \rightarrow 0 \\
\quad &| x::xs' \rightarrow 1 + \text{len xs'}
\end{align*}
\]

\(\text{len} [\[] \====> 0\)
\(\text{len} [\text{“carne”; “asada”}] \====> 2\)

Made Possible by Higher-Order Functions!
Recall: sum

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

sum []           ===>  0
sum [10;20;30]   ===>  60

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?

(* string list -> string *)
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** concat `xs` =
  - match `xs` with
    - `[]` -> ""
    - `x::xs'` -> `x^(concat xs')`

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

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

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

Q: What does this evaluate to? (a) `[1;2;3]` (b) `[3;2;1]` (c) `[]` (d) `[[3];[2];[1]]` (e) `[[1];[2];[3]]`
The “fold-right” pattern

\[
\text{foldr } f \ b \ [x1;x2;x3] \\
====> f x1 \ (\text{foldr } f \ b \ [x2;x3]) \\
====> f x1 \ (f x2 \ (\text{foldr } f \ b \ [x3])) \\
====> f x1 \ (f x2 \ (f x3 \ (\text{foldr } f \ b \ []))) \\
====> f x1 \ (f x2 \ (f x3 \ (\text{foldr } f \ b \ []))) \\
====> f x1 \ (f x2 \ (f x3 \ b))
\]

The “fold” Pattern

\[
\text{foldr } f \ b \ xs = \\
\text{match } xs \text{ with} \\
| [] -> b \\
| x::xs’-> f x \ (\text{foldr } f \ b \ xs’)
\]

Tail Recursive?

No!

Write: concat (TR) Online

\[
\text{concat } [] \\
====> "" \\
\text{concat } ["carne"; "asada"; "torta"] \\
====> "carneasadatorta"
\]
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"

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

sum []  
====> 0

sum [10; 20; 30]  
====> 60

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

What's the Pattern?
"Accumulation" Pattern

```
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
```

Specific Op

Q: What does this evaluate to?

(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 ...
• Map 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”

```
let pipe x f = f x

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

Compute the sum of squares of numbers in a list?

```
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 ?

Higher-order funcs enable modular code
• Each part only needs local information

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