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?

```text
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?

```text
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?

Moral of the day...

Recursion good... ...but HOFS better!

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!

News

• PA2 due tomorrow @ 5PM

• PA3 goes up tomorrow
  - Due 10/27

• Midterm 11/6
  - In class
  - Open book etc.
  - Practice materials on webpage

Thursday, October 18, 2012
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

\[
\begin{align*}
\text{let rec } & \text{ evens } xs = \text{ match } xs \text{ with } \\
& | [] \rightarrow \ldots \\
& | x::xs' \rightarrow \ldots
\end{align*}
\]

\[
\begin{align*}
\text{evens } [\ ] & \quad \Rightarrow [\ ] \\
\text{evens } [1;2;3;4] & \Rightarrow [2;4]
\end{align*}
\]

Write: fourLetters

\[
\begin{align*}
\text{let rec } & \text{ fourLetters } xs = \text{ match } xs \text{ with } \\
& | [] \rightarrow \ldots \\
& | x::xs' \rightarrow \ldots
\end{align*}
\]

\[
\begin{align*}
\text{fourLetters } [\ ] & \quad \Rightarrow [\ ] \\
\text{fourLetters } [“\text{cat”}; “\text{must”}; “\text{do”}; “\text{work”}] & \Rightarrow [“\text{must”}; “\text{work”}]
\end{align*}
\]
Write: evens

```ocaml
(* 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!

```ocaml
(* 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')
```

```
Yuck! Most code is same!
```

```
(* 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')
```

```
Moral of the Day...

“D.R.Y”
Don’t Repeat Yourself!
```

Thursday, October 18, 2012
Moral of the Day...

HOFs Allow “Factoring”

General “Pattern”

+ Specific “Operation”

Thursday, October 18, 2012
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!

```ocaml
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')
```

What’s the Pattern?

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

“Refactor” 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')
```

```ocaml
let rec pattern ...
```

Thursday, October 18, 2012
### “Refactor” Pattern

#### listUpper

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

#### listSquare

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

#### map

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

#### listUpper (Refactor Pattern)

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

```ocaml
let listUpper = map (fun x -> uppercase x)
```

#### listSquare (Refactor Pattern)

```ocaml
let listSquare = map (fun x -> x*x)
```

#### Refactor Pattern

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

```ocaml
let listUpper = map (fun x -> uppercase x)
```

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

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

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')

Generic “iteration” pattern

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

Thursday, October 18, 2012
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!

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

“Factored” code:
- Reuse iteration template
- Avoid bugs due to repetition
- Fix bug in one place!

Don’t Repeat Yourself!

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

Made Possible by Higher-Order Functions!

Recall: len

\[
\text{len}\ [\] \Rightarrow 0
\]
\[
\text{len}\ [\text{"carne"}; \text{"asada"}] \Rightarrow 2
\]
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?

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

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

\[
\text{foldr } f \ b \ [x1; x2; x3] \\
\quad \Rightarrow f \ x1 \ (\text{foldr } f \ b \ [x2; x3]) \\
\quad \Rightarrow f \ x1 \ (f \ x2 \ (\text{foldr } f \ b \ [x3])) \\
\quad \Rightarrow f \ x1 \ (f \ x2 \ (f \ x3 \ (\text{foldr } f \ b \ []))) \\
\quad \Rightarrow f \ x1 \ (f \ x2 \ (f \ x3 \ (\text{foldr } f \ b \ []))) \\
\quad \Rightarrow f \ x1 \ (f \ x2 \ (f \ x3 \ (b)))
\]

The "fold" Pattern

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

Tail Recursive?

No!

Write: concat (TR)

\[
\text{let } \text{concat } xs = ...
\]

\[
\text{concat } [] \Rightarrow "" \\
\text{concat } ["\text{carne}"; "\text{asada}"; "\text{torta}" ] \\
\Rightarrow "\text{carneasadatorta}"
\]
Write: **concat**

```ml
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"
```

**What’s the Pattern?**

```ml
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
```

Write: **sum (TR)**

```ml
let sum xs = ...

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

```
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
```

Q: What does this evaluate to ?

(a) [1;2;3]
(b) [3;2;1]
(c) []
(d) [[3];[2];[1]]
(e) [[1];[2];[3]]

Thursday, October 18, 2012
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

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

Data Structure Library
Uses list

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

Thursday, October 18, 2012