CSE 130 : Programming Languages

Higher-Order Functions

Ranjit Jhala
UC San Diego
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?

```ml
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 tomorrow @ 11:59:59 pm

• PA3 goes up soon

• 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

```ocaml
(* 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]
Write: fourLetters

(* 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 x mod 2 = 0
               then x::(foo xs')
               else (foo xs')

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

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

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

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

The “filter” pattern
Factor Into Generic + Specific

Specific Operations

let evens xs =
  filter \( \text{fun } x \to x \mod 2 = 0 \) xs

let fourLetters xs =
  filter \( \text{fun } x \to \text{length } x = 4 \) xs

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

Generic “filter” pattern
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
  | [] -> ...
  | 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')
Yuck! Most code is same!

```ocaml
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')

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')

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')

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

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

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

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

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

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

```ml
let listUpper = map uppercase
```

```ml
let rec map f xs =
  match xs with
  | [] -> []
  | x::xs'-> (f x)::(map f 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 listSquare = map (fun x -> x*x)
Factor Into Generic + Specific

\[
\begin{align*}
\text{let}\ & \text{listSquare} = \text{map} \ (\text{fun}\ x \rightarrow \ x \times x) \\
\text{let}\ & \text{listUpper} = \text{map} \ \text{uppercase}
\end{align*}
\]

Specific Op

\[
\begin{align*}
\text{let rec}\ & \text{map} \ f \ \text{xs} = \\
& \text{match} \ \text{xs} \ \text{with} \\
& | \ [] \rightarrow [] \\
& | x :: xs’ \rightarrow (f x) :: (\text{map} \ f \ xs’)
\end{align*}
\]

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

```
let rec map f xs =
  match xs with
  | []       -> []
  | x :: xs'  -> (f x) :: (map f xs')
```
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]```
Don’t Repeat Yourself!

“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

\[
(*) \ 'a \ list \ \rightarrow \ int \ *)
\]

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

\[
len \ [] \ \Rightarrow \ 0
\]

\[
len \ ["\text{carne}"; \ "\text{asada}"] \ \Rightarrow \ 2
\]
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' -> ...

concat []
  ===> ""

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

```ocaml
(* 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?

```ocaml
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 len =
foldr (fun \( x \) \( n \) -> \( n+1 \)) 0

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

let concat =
foldr (fun \( x \) \( n \) -> \( x^n \)) ""
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]

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

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

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

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

Tail Recursive?
The “fold” Pattern

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

Tail Recursive?
No!
Write: `concat` (TR)

```haskell
let concat xs = ...
```

`concat []` 

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

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

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

```
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: \( \text{sum} \) (TR)

\[
\text{let } \text{sum } \text{xs} = \ldots
\]

\[
\text{sum } [] \quad \Rightarrow \quad 0
\]

\[
\text{sum } [10;20;30] \quad \Rightarrow \quad 60
\]
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?

```ocaml
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) ""

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

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

Specific Op
Q: What does this evaluate to?

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”

```
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?

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)

Create 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

---

**Data Structure**

**Library**

list

**Provides** meta-functions:
map, fold, filter
to traverse, accumulate over
lists, trees etc.

Meta-functions don’t need client
info (tester ? accumulator ?)