

CSE 130 : Spring 2011

Programming Languages

Lecture 6: *Higher-Order Functions*

Ranjit Jhala
UC San Diego



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 ?

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

Example : Factorial

```
let rec fac n =
```

```
  if n=0
```

```
  then 1
```

```
  else n * fac (n-1) ;;
```

Induction Condition

Base Expression

Inductive Expression

Example : Clone

```
let rec clone x n =
```

```
  if n=0
```

```
  then []
```

```
  else x :: (clone x (n-1)) ;;
```

Induction Condition

Base Expression

Inductive Expression

Example : interval

```
let rec interval i j =
```

```
  if i > j
```

Induction Condition

```
  then []
```

Base Expression

```
  else i :: (interval (i+1) j) ;
```

Inductive Expression

Example : List Append

Roll our own @

```
let rec append l1 l2 =  
  match l1 with
```

Base “pattern” `[] -> l2` *Base Expression*

Ind. “pattern” | `h::t -> h::(append t l2)`

`;;` *Inductive Expression*

Example : List Maximum

Find maximum element in +ve int list
... in a more ML-ish way

```
let max x y = if x > y then x else y
let listMax l =
  let rec helper cur l =
    match l with
    Base
    pattern [ ] -> Base Expression cur
    | Ind. pattern h :: t -> Inductive Expression helper (max cur h) t
  in
    helper 0 l
;;
```

Tail Recursion

“last thing” function does is a recursive call

```
let rec fac n =  
  if n=0  
  then 1  
  else n * fac (n-1) ;;
```

NOT TR

bad because height of stack = $O(n)$

Tail Recursion

“last thing” function does is a recursive call

```
let rec fac n =  
  if n=0  
  then 1  
  else n * fac (n-1) ;;
```

NOT TR

bad because height of stack = $O(n)$

Tail Recursive Factorial

“last thing” function does is a recursive call

```
let rec fac n =
```

News

- PA3 is up
 - Due 4/22
 - OH in CSE 250 (RJ: 2-4pm/Thu)
- Midterm 4/28
 - 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 ?

Functions are “first-class” values

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

Creating,
(Returning)
Functions

Returning functions

```
let lt = fun x -> fun y -> x < y;
```

Returned value is a function

```
let lt x y = x < y;;
```

Identical but easier to write!

In general, these two are equivalent:

```
let f = fun x1 -> ... -> fun xn -> e
```

```
let f x1 ... xn = e
```

Returning functions

```
let lt x y = x < y;
```

```
let lt = fun x -> fun y -> x < y
```

```
int ! (int ! bool)
```

```
lt 5 20;;
```

```
lt 20 7;;
```

```
let is5lt = lt 5;;
```

```
let is10lt = lt 10;;
```

Parameterized “tester”

- Create many similar testers
- Where is this useful ?

Remember this ?

Tail Rec ?

```
let rec sort lt l =  
  match l with  
  [] -> []  
  | (h::t) =  
    let (l,r) = partition (lt h) t in  
    (sort lt l)@(h::(sort lt r))  
;;
```

- Use “tester” to partition list
 - Tester parameterized by “pivot” `h`
- Reuse code to sort `any` type of list
 - Use different “lt” to sort in different orders

Function Currying

Tuple version: **let** $f \ (x_1, \dots, x_n) = e$

$T_1 * \dots * T_n \ ! \ T$

Curried version: **let** $f \ x_1 \ \dots \ x_n = e_j$

$T_1 \ ! \ \dots \ ! T_n \ ! T$

Function Currying

Multiple argument functions by returning a function that takes the next argument

- Named after a person (Haskell Curry)

```
let lt x y = x < y;
```

Could have done: **let** lt (x, y) = x < y;

- But then no “testers” possible
- Must pick good order of arguments

Using parameterized testers

```
let rec sort lt l =  
  match l with  
  [] -> []  
  | (h::t) =  
    let (l,r) = partition (lt h) t in  
    (sort lt l)@(h::(sort lt r))  
;;
```

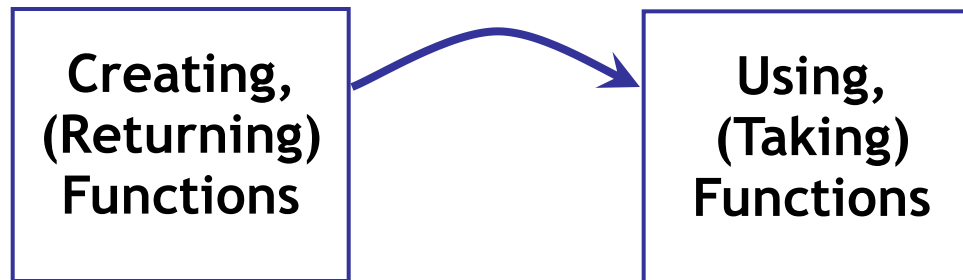
partition

- Takes a tester (and a list) as argument
- Returns a pair: (list **passing** test, list **failing** test)
- Can be called with **any** tester!

Functions are “first-class” values

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

*Parameterized,
similar functions
(e.g. Testers)*



Useful if parameterized functions can be passed to, hence used/called by other functions...

Why take functions as input ?

```
let rec evens l =  
  match l with  
    [] -> []  
  | h::t -> if is_even h then h::(evens t) else evens t
```

```
let rec lessers x l =  
  match l with  
    [] -> []  
  | h::t -> if h<x then h::(lessers x t) else lessers x t
```

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t) else filter f t
```

Factoring and Reuse

```
let rec lessers x l =  
  match l with  
    [] -> []  
  | h::t -> if h<x then h::(lessers x t) else lessers x t
```

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t) else filter f t
```

“Factor” code:

- Generic pattern
- Specific instance

```
let lessers x l =  
  filter (fun i -> i<x) l
```

Factoring and Reuse

```
let rec evens l =  
  match l with  
    [] -> []  
  | h::t -> if is_even h then h::(evens t) else evens t
```

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t) else filter f t
```

“Factor” code:

- Generic pattern
- Specific instance

```
let evens l =  
  filter is_even l
```

Encoding Patterns as functions

```
let rec filter f l =  
  match l with  
    [] -> []  
  | h::t -> if (f h) then h::(filter f t)  
             else (filter f t);;
```

```
let neg f = fun x -> not (f x)  
let partition f l= (filter f l, filter(neg f) l))
```

filter, neg, partition: higher-order functions

- Take a **any** tester as argument!

Iteration Pattern

```
let rec listUppercase xs =  
  match xs with  
    []      -> []  
  | h::t    -> (uppercase h)::(listUppercase t)
```

```
let rec listSquare xs =  
  match xs with  
    []      -> []  
  | h::t    -> (h * h)::(listSquare t)
```

```
let addPair (x,y) = x + y  
let rec listAddPair xs =  
  match xs with  
    []      -> []  
  | (hx,hy)::t -> (addPair (hx,hy))::(listAddPair t)
```

Iteration Pattern

```
let rec listUppercase xs =  
  match xs with  
  | [] -> []  
  | h::t -> (uppercase h) :: (listUppercase t)  
  | (h::t) -> (f h) :: (map f t)
```

```
let listUpperCase l = map uppercase l
```

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

```
let listAddpair l = map (fun (x,y) -> x+y) l
```

Higher-order functions: map

```
let rec map f l =  
  match l with  
  [] -> []  
  | (h::t) -> (f h) :: (map f t)
```

('a ! 'b) ! 'a list ! 'b list

Type says it all !

- Applies “f” to each element in input list
- Makes a list of the results

Factoring Iteration w/ “map”

```
let rec map f l =  
  match l with  
    [] -> []  
  | (h::t) -> (f h) :: (map f t)
```

“Factored” code:

- Reuse iteration template
- Avoid bugs due to repetition
- Fix bug in one place !

Another pattern: Accumulation

```
let max x y = if x > y then x else y ;
let listMax l =
  let rec help cur l =
    match l with
    [] -> cur
    | h::t -> help (max cur h) t
  in
    helper 0 l;;
```

```
let concat l =
  let rec help cur l =
    match l with
    [] -> cur
    | h::t -> help (cur^h) t
  in
    helper "" l;;
```

Whats the pattern ?

Whats the pattern ? Tail Rec ?

Whats the pattern ? Tail Rec ?

```
Let rec fold f cur l =  
  case l of  
    [] -> cur  
  | h::t -> fold f (f cur h) t
```

What is: `fold f base [v1;v2;...;vn]` ?

```
f (... ( f (
                                     , v3) , vn)
      f (base, v1)
    f ( f (
        , v2)
      , v3)
  f (... (
          , vn)
```

Examples of fold

Currying! This is a function!

```
let listMax = fold max 0
```

Currying! This is a function!

```
let concat = fold (^) ""
```

Pick correct
base case!

```
let multiplier =
```

Examples of fold

What does this do ?

```
let f l = fold (::) [] l
```

Funcs taking/returning funcs

Identify common computation “patterns”

- Filter values in a set, list, tree ...
- Iterate a function over a set, list, tree ...
map
- Accumulate some value over a collection
fold

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)  
    |> fold_left (+) 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



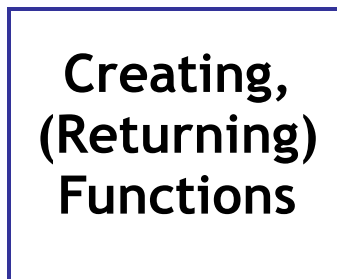
Using,
(Taking)
Functions

*Iterator, Accumul,
Reuse computation
pattern w/o
exposing local info*

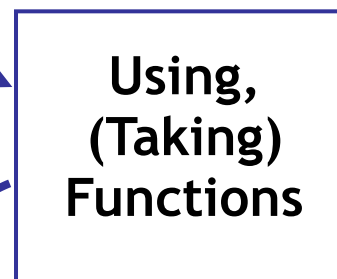
Functions are “first-class” values

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

*Parameterized,
similar functions
(e.g. Testers)*



*Iterator, Accumul,
Reuse computation
pattern w/o
exposing local info*



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

Funcs taking/returning funcs

Higher-order funcs enable **modular** code

- Each part only needs **local** information

