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

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

- **Induction Condition**
- **Base Expression**
- **Inductive Expression**
Tail Recursion

“last thing” function does is a recursive call

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

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

Tail Recursive Factorial

“last thing” function does is a recursive call

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

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

Example: List Maximum

```
let rec listMax xs =
  match xs with
  | []       -> 0
  | x::xs'   -> max x (listMax xs')
```

bad because height of stack = $O(#listsize)$
Example: List Maximum

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

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

News

- PA2 due tomorrow @ 5PM
- PA3 goes up tomorrow
  - Due 10/21
- Midterm 11/1
  - 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?
Returning functions

In general, these two are equivalent:

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

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

Returned value is a function

Identical but easier to write!

```ml
let f = fun x1 -> … -> fun xn -> e
let f x1 … xn = e
```

Parameterized “tester”

- Create many similar testers
- Where is this useful?

Remember this?

- 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

Using parameterized testers

```ml
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))

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)

Creating, (Returning) Functions

Using, (Taking) Functions

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:

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
```

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

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

```
let listUpperCase l = map upperCase
let listSquare l    = map (fun x -> x*x) l
let listAddpair l   = map (fun (x,y) -> x+y) l
```

**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 l with 
    []   -> [] 
  | (hx,hy)::t -> (addPair (hx,hy))::(listAddPair t)
```

```
let listUpperCase l = map upperCase
let listSquare l    = map (fun x -> x*x) l
let listAddpair l   = map (fun (x,y) -> x+y) 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)
```

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

```
let listUpperCase l = map upperCase
let listSquare l    = map (fun x -> x*x) l
let listAddpair l   = map (fun (x,y) -> x+y) l
```

**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 l with 
    []   -> [] 
  | (hx,hy)::t -> (addPair (hx,hy))::(listAddPair t)
```

```
let listUpperCase l = map upperCase
let listSquare l    = map (fun x -> x*x) l
let listAddpair l   = map (fun (x,y) -> x+y) l
```
Higher-order functions: map

Type says it all!
- Applies "f" to each element in input list
- Makes a list of the results

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

Factoring Iteration w/ “map”

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

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

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;;
```

Whats the pattern?

```
let concat l =
  let rec help cur l =
    match l with
      [] -> cur
    | h::t -> help (cur^h) t
  in
    helper "" l;;
```
What is 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(... (f(base,v1),v2),v3),vn)

Examples of fold

let concat =
 let multiplier =

Currying! This is a function!

let listMax = fold max 0

Currying! This is a function!

let concat =
 let multiplier =

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 ...
- Convert 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”

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

Compute the sum of squares of numbers in a list?

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

Tail Rec ?

Functions are “first-class” values

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

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.

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

Funcs taking/returning funcs

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 ?)