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

- Base Expression
- Inductive Expression
- Induction Condition
Example: Clone

```ml
let rec clone x n =
  if n=0 then []
  else x::(clone x (n-1));;
```

**Base Expression**
- `[]`

**Inductive Expression**
- `x::(clone x (n-1))`

**Induction Condition**
- `if n=0`

---

Example: interval

```ml
let rec interval i j =
  if i > j then []
  else i::(interval (i+1) j);
```

**Base Expression**
- `[]`

**Inductive Expression**
- `i::(interval (i+1) j)`

**Induction Condition**
- `if i > j`

---

Example: List Append

Roll our own @

```ml
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | h::t -> h::(append t l2));
```

**Base Expression**
- `[]` -> `l2`

**Inductive Expression**
- `h::t` -> `h::(append t l2)`

---

Example: List Maximum

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

```ml
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
```

**Base Expression**
- `[]` -> `cur`

**Inductive Expression**
- `h::t` -> `helper (max cur h) t`

**Base "pattern"**
- `[]` -> `12`

**Ind. "pattern"**
- `h::t` -> `h::(append t 12)`

---

Example: List Maximum

```ml
let rec interval i j =
  if i > j then []
  else i::(interval (i+1) j);
```

**Base Expression**
- `[]`

**Inductive Expression**
- `i::(interval (i+1) j)`

**Induction Condition**
- `if i > j`

---

Example: List Append

Roll our own @

```ml
let rec append l1 l2 =
  match l1 with
    [] -> l2
  | h::t -> h::(append t l2));
```

**Base Expression**
- `[]` -> `l2`

**Inductive Expression**
- `h::t` -> `h::(append t l2)`

---

Example: List Maximum

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

```ml
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
```

**Base Expression**
- `[]` -> `cur`

**Inductive Expression**
- `h::t` -> `helper (max cur h) t`

**Base "pattern"**
- `[]` -> `12`

**Ind. "pattern"**
- `h::t` -> `h::(append t 12)`
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 =
```

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 =
```

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?

Returning functions

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

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

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

Function Currying

**Multiple argument functions by returning a function that takes the next argument**
- Named after a person (Haskell Curry)

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

Could have done:  

```plaintext
let lt (x,y) = x<y;
```

Using parameterized testers

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

```plaintext
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)) 
;;
```
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

"Factor" code:
- Generic pattern
- Specific instance

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

let evens l = filter is_even l
Encoding Patterns as functions

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

- `filter`, `neg`, `partition`: higher-order functions
  - Take a any tester as argument!

Iteration Pattern

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

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

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

Higher-order functions: map

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

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

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

Whats the pattern?
Tail Rec ?
What's 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(   ,v2)
  f(   ,v3)
f(...(   ,vn)

Examples of fold

let concat =

let multiplier =

Currying! This is a function!

let listMax =

Currying! This is a function!

Pick correct base case!

let concat =

Currying! This is a function!

let multiplier =

Examples of fold

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

What does this do?

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”

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

Compute the sum of squares of numbers in a list?

```ocaml
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

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions

Using, (Taking) Functions

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions

Using, (Taking) Functions

Compose Functions: Flexible way to build Complex functions from primitives.
Higher-order func enable modular code
• Each part only needs local information

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Client Uses List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses meta-functions:</td>
<td>map, fold, filter</td>
</tr>
<tr>
<td>With locally-dependent funs</td>
<td>(lt h), square etc.</td>
</tr>
<tr>
<td>Without requiring implement.</td>
<td>details of data structure</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Library list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides meta-functions:</td>
<td>map, fold, filter</td>
</tr>
<tr>
<td>to traverse, accumulate over</td>
<td>lists, trees etc.</td>
</tr>
<tr>
<td>Meta-functions don’t need client info</td>
<td>(tester ? accumulator ?)</td>
</tr>
</tbody>
</table>