Functional Programming Practice Session

February 12, 2016

Outline

We'll be covering functional programming concepts that are common to both Haskell and OCaml (no monads or module types)

- Fold
- Algebraic data types and pattern matching
- Sorting algorithms and practice problems

OCaml vs. Haskell syntax

OCaml:

```
[1;2;3;4] = 1::2::3::4::[]

[1;2] @ [3;4] = [1;2;3;4]

(* recursive functions must be declared with let rec, not let*)
let rec f (x:'a) (y:'b) : 'c =
  match x with
    val1 -> ...
  | val2 -> ...
;;

"here's how you " ^ "put things together"

(fun x -> x + 2)
```

OCaml vs. Haskell syntax

Haskell:

```haskell
[1,2,3,4] = 1:2:3:4:[]

[1,2] ++ [3,4]

-- recursive functions are declared the same
-- way as nonrecursive functions
f :: a -> b -> c -- type annotations are separate
f x y =
  case x of
    val1 -> ...
    val2 -> ...

-- equivalent to pattern matching above
f (val1) y = ...
f (val2) y = ...

"here's how you " ++ "put things together"
(\x -> x + 2)
```

Fold

In OCaml:

```
let rec fold_right (f:'a -> 'b -> 'b) (l:'a list) (acc:'b) : 'b =
  match l with
    [] -> acc
  | x::xs -> f x (fold_right f xs acc)
;;

let rec fold_left (f:'b -> 'a -> 'b) (acc:'b) (l:'a list) : 'b =
  match l with
    [] -> acc
  | x::xs -> fold_left f (f acc x) xs
;;
```

Fold

In Haskell:

```haskell
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f acc [] = acc
foldr f acc (x:xs) = f x (foldr f acc xs)

foldl :: (b -> a -> b) -> b -> [a] -> b
foldl f acc [] = acc
foldl f acc (x:xs) = foldl f (f acc x) xs
```

Fold

Find the sum of even numbers in a list.

In Ocaml:

```
fold_right (fun x acc -> if x mod 2 == 0 then x+acc else acc) [1,2,3,4,5] 0
```

In Haskell:

```
foldr (\x acc -> if x `mod` 2 == 0 then x+acc else acc) 0 [1,2,3,4,5]
```

In Python (without using `reduce`):

```
sum = 0 # sum is the accumulator
for n in [1,2,3,4,5]:
  if n % 2 == 0:
    sum += n
```

Fold Right vs. Fold Left

```
foldr (+) 0 [1,2,3,4,5] =
1 + (2 + (3 + (4 + (5 + 0))))

foldl (+) 0 [1,2,3,4,5] =
((((0 + 1) + 2) + 3) + 4) + 5
```

Fold Right vs. Fold Left

In Haskell:

Since Haskell is **lazily evaluated**, you can sometimes short-circuit evaluation by paying attention to which parameter is being pattern matched.

```
or :: Bool -> Bool -> Bool
or True _  = True
or False x = x

foldr or False [True,False,False] =
or True (foldr or False [False,False]) =
True

foldl or [True,False,False] False =
foldl or [False, False] (or False True) =
foldl or [False] (or (or False True) False) =
foldl or [] (or (or (or False True) False) False) =
(or (or (or False True) False) False) =
(or (or True False) False) =
(or True False) =
True
```

Fold Right vs. Fold Left

Since OCaml is **eagerly evaluated** – the interpreter evaluates the arguments before passing it to the function – this function can't be short-circuited.

```
let orfunc (x:bool) (y:bool) : bool =
  match x with True -> true | False -> y ;;

fold_right orfunc [true;false;false] false =
orfunc true (foldr orfunc [false;false] false) =
orfunc true (orfunc false (foldr orfunc [false] false)) =
orfunc true (orfunc false (orfunc false (foldr orfunc [] false))) =
orfunc true (orfunc false (orfunc false false)) =
orfunc true (orfunc false false) =
orfunc true false =
true

fold_left orfunc false [true;false;false] =
fold_left orfunc (orfunc false true) [false;false] =
fold_left orfunc true [false;false] =
fold_left orfunc (orfunc true false) [false] =
fold_left orfunc true [false] =
fold_left orfunc (orfunc true false) [] =
fold_left orfunc true [] =
true
```

From nothing, fold; from fold, everything

```
let map (f:'a -> 'b) (xs:'a list) : 'b list =
  List.fold_right (fun x acc -> (f x)::acc) xs [] ;;

let filter (f:'a -> bool) (xs:'a list) : 'a list =
  List.fold_right (fun x acc -> if f x then x::acc else acc) xs [] ;;

let length (xs:'a list) : int =
  List.fold_right (fun x acc -> acc + 1) xs 0 ;;

let reverse (xs:'a list) : 'a list =
  List.fold_left (fun acc x -> x::acc) [] xs ;;
```

From nothing, fold; from fold, everything

In Haskell:

```haskell
map :: (a -> b) -> [a] -> [b]
map f xs = foldr (\x acc -> (f x):acc) [] xs

filter :: (a -> Bool) -> [a] -> [a]
filter f xs = foldr (\x acc -> if f x then x:acc else acc) [] xs

length :: [a] -> Int
length xs = foldr (\x acc -> acc + 1) 0 xs

reverse :: [a] -> [a]
reverse xs = foldl (\acc x -> x:acc) [] xs
```

Algebraic Data Types

- User-defined types
- Can encode different variants (“subclasses”) of a particular type
- Can compactly encode recursive data structures
- Can be parametrized with type variables (cf. Java generics)

Algebraic Data Types (ADTs)

OCaml:

- Each constructor can be paired with at most 1 type
- `typename * typename * ...` is a *single tuple type*

```
type typename =
  | Constructor1 of typename * typename ...
  | Constructor2 of typename * typename ...
  | Constructor3 of typename * typename ...
```

Haskell:

- Each constructor can be paired with an arbitrary number of types

```
data TypeName =
    Constructor1 TypeName TypeName ...
  | Constructor2 TypeName TypeName ...
  | Constructor3 TypeName TypeName ...
```

Compare `Typename` to an **abstract base class** and `Constructors` to **child classes**.

Lists

OCaml:

```
type 'a list =
  | Cons of 'a * ('a list)
  | Nil
;;

Cons (1, Cons (2, Cons (3, Nil))) = [1;2;3]
```

Recall how `List` is implemented in Cool.

Haskell:

```
data List a = Cons a (List a) | Nil

Cons 1 (Cons 2 (Cons 3 Nil)) = [1,2,3]
```

Pattern Matching

- Like a `switch` statement in Java or C
- Usually used to have separate cases between different constructors of an ADT and to have separate cases between empty/non-empty lists

Pattern Matching

OCaml:

```
let rec sum (l:int list) : int =
  match l with
    Nil -> 0
  | Cons (x, tl) -> x + sum tl
;;
```

Haskell:

```
sum :: List Int -> Int
sum Nil = 0
sum (Cons n tl) = n + sum tl

sum = case l of Nil -> 0; Cons n tl -> n + sum tl
```

Binary Trees

In OCaml

```
type 'a btree =
    Node of 'a * ('a btree) * ('a btree)
  | Leaf of 'a
  | NilLeaf
;;
let rec preOrder (tree:'a btree) : 'a list =
  match tree with
    NilLeaf -> []
  | Leaf x -> [x]
  | Node (x, left, right) ->
      [x] @ (preOrder left) @ (preOrder right)
;;
```

Binary Trees

In Haskell:

```haskell
data BTree a =
    Node a (BTree a) (BTree a)
  | Leaf a
  | NilLeaf

preOrder :: BTree a -> [a]
preOrder (Leaf x)  = [x]
preOrder (Node x left right) =
  [x] ++ (preOrder left) ++ (preOrder right)
```

An Arithmetic Language

Hint: Remember this for PA3!

```
type arith =
  | Val of int
  | Add of arith * arith
  | Sub of arith * arith
  | Mul of arith * arith
;;

let rec serializeArith (ast:arith) : string =
  match ast with
    Val n -> "int\n" ^ (string_of_int n) ^ "\n"
  | Add (x,y) -> "add\n" ^ serializeArith x ^ serializeArith y
  | Sub (x,y) -> "sub\n" ^ serializeArith x ^ serializeArith y
  | Mul (x,y) -> "mul\n" ^ serializeArith x ^ serializeArith y
;;
```

An Arithmetic Language

In Haskell:

```haskell
data Arith =
    Val Int
  | Add Arith Arith
  | Sub Arith Arith
  | Mul Arith Arith

serializeArith :: Arith -> String
serializeArith (Val n) = "int\n" ++ show n
serializeArith (Add x y) = "add\n" ++ serializeArith x ++ serializeArith y
serializeArith (Sub x y) = "sub\n" ++ serializeArith x ++ serializeArith y
serializeArith (Mul x y) = "mul\n" ++ serializeArith x ++ serializeArith y
```

An Arithmetic Language

Hint: Remember this for PA4-PA5!

```
let rec eval (a:arith) : int =
  match a with
    Val n -> n
  | Add (x,y) -> (eval x) + (eval y)
  | Sub (x,y) -> (eval x) - (eval y)
  | Mul (x,y) -> (eval x) * (eval y)
;;
eval (Mul (Add (Val 2, Val 3), Val 4)) = 20 ;;
```

An Arithmetic Language

In Haskell:

```
eval :: Arith -> Int
eval (Val n)   = n
eval (Add x y) = (eval x) + (eval y)
eval (Sub x y) = (eval x) - (eval y)
eval (Mul x y) = (eval x) * (eval y)

eval (Mul (Add (Val 2) (Val 3)) (Val 4)) = 20
```

Option Types

Useful for capturing failure in a function

OCaml:

```
type 'a option =
  | Some of 'a
  | None ;;

let head (l:'a list) : 'a option =
  match l with
    [] -> None
  | x::xs -> Some x ;;
```

Haskell:

```
data Maybe a = Just a | Nothing

head :: [a] -> Maybe a
head []     = Nothing
head (x:xs) = Just x
```

Functional Programming Idioms

- Recursion and folding, not iteration
- Many tiny functions instead of one big function
- Keep track of state with parameters (accumulators)
- Type annotations are your friend!

Type Annotations

A lot of times you can guess what a function does just by reading its type annotation.

```
f : 'a list -> int -> 'a
g : 'a -> 'a list -> bool
h : 'a -> ('a * 'b) list -> 'b
```

Type annotations are your friend!

A lot of times you can guess what a function does just by reading its type annotation.

- `f` is `nth` (return the nth element of a given list)
- `g` is `mem` (returns whether an element is in a list)
- `h` is `assoc` (treats list like a map, returns value associated with key)

Sorting algorithms

- Insertsort, Mergesort, Quicksort
- No need to mess with array indices; intuitive implementations using recursion and folding

Insertsort

- Repeatedly insert elements into a sorted sublist
- Streaming algorithm

Insertsort

How is `insert` defined?

```
let insertSort (l:int list) : int list =
  let rec insert x il = ??? in
  List.fold_right insert l []
;;
```

Insertsort

```
let insertSort (l:int list) : int list =
  let rec insert x il = begin
    match il with
    (* insert at the end of sorted sublist *)
      [] -> [x]
    (* insert at current position or recurse to rest of list *)
    | y::ys -> if x >= y then y::(insert x ys) else x::y::ys
  end in
  (* add elements one by one to acc, which is kept sorted *)
  List.fold_right insert l []
;;
```

Insertsort

In Haskell:

```
insertSort :: Ord a => [a] -> [a]
-- add elements one by one to acc, which is kept sorted
insertSort xs = foldr insert [] xs
        -- insert at the end of sorted sublist
    where insert x [] = [x]
        -- insert at current position or recurse to rest of list
          insert x (y:ys) =
          if x >= y then y:(insert x ys) else x:y:ys
```

Executing insertsort

```
insertSort [2,3,1,4]

foldr insert [] [2,3,1,4]

insert 2 (foldr insert [] [3,1,4])

insert 2 (insert 3 (foldr insert [] [1,4]))

insert 2 (insert 3 (insert 1 (foldr insert [] [4])))

insert 2 (insert 3 (insert 1 (insert 4 (foldr insert [] []))))

insert 2 (insert 3 (insert 1 (insert 4 [])))
```

Executing insertsort

```
insert 2 (insert 3 (insert 1 (insert 4 [])

insert 2 (insert 3 (insert 1 [4]))

insert 2 (insert 3 [1,4])

insert 2 (1:(insert 3 [4]))

insert 2 (1:[3,4])

insert 2 [1,3,4]
```

Executing insertsort

```
insert 2 [1,3,4]

1:(insert 2 [3,4])

1:[2,3,4]

[1,2,3,4]
```

Mergesort

- Divide and conquer algorithm
- Divide list into two sublists
- Recursive sort sublists
- Merge sorted sublists into one sorted list

Mergesort

How to implement `merge`? (Assume `splitAt` is implemented; we'll go back to it)

What happens if case `[x]` is not there?

```
let rec mergeSort (l:int list) : int list =
  let rec merge xxs yys = ??? in
  match l with
    [] -> []
  | [x] -> [x]
  | _ ->
    let (left, right) = splitAt (List.length l / 2) l in
    merge (mergeSort left) (mergeSort right)
;;
```

Mergesort

```
let rec mergeSort (l:int list) : int list =
  let rec merge xxs yys = begin
    match (xxs,yys) with
      ([], []) -> []
    (* right list is empty; rest of left is end of sorted list *)
    | (xs, []) -> xs
    (* left list is empty; rest of right is end of sorted list *)
    | ([], ys) -> ys
    (* pick lower head and recurse on the rest *)
    | (x::xs, y::ys) ->
        if x < y then x::(merge xs (y::ys))
                 else y::(merge (x::xs) ys)
  end in
  match l with
    [] -> []
  | [x] -> [x]
  | _ ->
    (* split into two sublists *)
    let (left, right) = splitAt (List.length l / 2) l in
    (* merge sorted sublists *)
    merge (mergeSort left) (mergeSort right)
;;
```

Mergesort

```haskell
mergeSort :: Ord a => [a] -> [a]
mergeSort [] = []
mergeSort [x] = [x]
mergeSort l =
  -- merge sorted sublists
  merge (mergeSort left) (mergeSort right)
        -- split into two sublists
  where (left, right) = splitAt (length l `div` 2) l
        merge [] [] = []
        -- right list is empty; rest of left is end of sorted list
        merge xs [] = xs
        -- left list is empty; rest of right is end of sorted list
        merge [] ys = ys
        -- pick lower head and recurse on the rest
        merge (x:xs) (y:ys) =
          if x < y then x:(merge xs (y:ys))
                   else y:(merge (x:xs) ys)
```

Quicksort

- Divide and conquer algorithm
- Select a pivot element
- Partition rest of list into two sublists: lower/eq and higher
- Sort sublists and then append together with pivot

Quicksort

How are `left` and `right` defined?

```
let rec quickSort (l:int list) : int list =
  match l with
    [] -> []
  | x::xs ->
    let left = ??? in
    let right = ??? in
    (quickSort left) @ [x] @ (quickSort right)
;;
```

Quicksort

```
let rec quicksort (l:int list) : int list =
  match l with
    [] -> []
  | x::xs ->
    (* get lower/eq sublist *)
    let left = List.filter (fun y -> x >= y) xs in
    (* get higher sublist *)
    let right = List.filter (fun y -> x < y) xs in
    (* append into complete sorted list *)
    (quickSort left) @ [x] @ (quickSort right)
;;
```

Quicksort

In Haskell:

```haskell
quickSort :: Ord a => [a] -> [a]
quickSort [] = []
quickSort (x:xs) =
  -- append into complete sorted list
  (quickSort left) ++ [x] ++ (quickSort right)
  where left = filter (x >=) xs -- get lower/eq sublist
        right = filter (x <) xs -- get higher sublist
```

Quicksort with explicit partition

In OCaml:

```
let rec quickSort2 (l:int list) : int list =
  match l with
    [] -> []
  | x::xs ->
    (* put element in the right sublist *)
    let part y (lo, hi) = if x >= y then (y::lo, hi) else (lo, y::hi) in
    (* repeated put list elems into right sublist *)
    let partition lp = List.fold_right part lp ([],[]) in
    (* create partition *)
    let (left, right) = partition xs in
    (* append into complete sorted list *)
    (quickSort2 left) @ [x] @ (quickSort2 right)
;;
```

Quicksort with explicit partition

In Haskell:

```
quickSort2 :: Ord a => [a] -> [a]
quickSort2 [] = []
quickSort2 (x:xs) =
  -- append into complete sorted list
  (quickSort2 left) ++ [x] ++ (quickSort2 right)
  where (left, right) = partition xs -- create partition
        -- repeated put list elems into right sublist
        partition = foldr part ([], [])
        -- put element in the right sublist
        part y (low, hi) =
          if x >= y then (y:low, hi) else (low, y:hi)
```

Notice the partial application for `partition`!

Practice Problems

Put your functional programming knowledge to the test

Problem 1

Implement `append` recursively. Don't use other functions!

```
let append (xxs:'a list) (yys:'a list) : 'a list = ??? ;;
```

```
append [1;2;3] [4;5;6] = [1;2;3;4;5;6]
```

Problem 1 answer

Implement append recursively. Don't use other functions!

```
let rec append (xxs:'a list) (yys:'a list) : 'a list =
  match xxs with
    [] -> yys
  | x::xs -> x::(append xs yys)
;;
```

Problem 1 answer

In Haskell:

```haskell
append :: [a] -> [a] -> [a]
append [] ys = ys
append (x:xs) ys = x:(append xs ys)
```

Problem 2

Implement `reverse` recursively. Don't use other functions!

```
let rec reverse (l:'a list) : 'a list = ??? ;;
```

```
reverse [1;2;3;4;5] = [5;4;3;2;1]
```

Problem 2 answer

Implement `reverse` recursively. Don't use other functions!

```
let rec reverse (l:'a list) : 'a list =
  let rec insertBack y xxs = begin
    match xxs with
      [] -> [y]
    | x::xs -> x::(insertBack y xs)
  end in
  match l with
    [] -> []
  | x::xs -> insertBack x (reverse xs)
;;
```

Problem 2 answer

In Haskell:

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = insertBack x (reverse xs)
  where insertBack x [] = [x]
        insertBack x (y:ys) = y:(insertBack x ys)
```

Problem 3

Implement `treeSum`. (Hint: Remember how `preOrder` is implemented)

```
type 'a btree =
  | Node of 'a * 'a btree * 'a btree
  | Leaf of 'a
  | NilLeaf
;;

let rec treeSum (tree:int btree) : int = ??? ;;

treeSum (Node (4, Leaf 5, NilLeaf)) = 9
```

Problem 3 answer

Implement `treeSum`.

```
type 'a btree =
  | Node of 'a * 'a btree * 'a btree
  | Leaf of 'a
  | NilLeaf
;;

let rec treeSum (tree:int btree) : int =
  match tree with
    NilLeaf -> 0
  | Leaf n -> n
  | Node (n,left,right) -> n + (treeSum left) + (treeSum right)
;;
```

Problem 3 answer

In Haskell:

```haskell
data BTree a =
    Node a (BTree a) (BTree a)
  | Leaf a
  | NilLeaf

treeSum :: BTree Int -> Int
treeSum NilLeaf = 0
treeSum (Leaf x) = x
treeSum (Node x left right) =
  x + (treeSum left) + (treeSum right)
```

Problem 4

Implement `unzip`.

```
let unzip (l:('a * 'b) list) : ('a list) * ('b list) = ??? ;;

unzip [(1,"A");(2,"B");(3,"C")] = ([1;2;3], ["A";"B";"C"])
```

Problem 4 answer

Implement `unzip`.

```
let unzip (l:('a * 'b) list) : ('a list) * ('b list) =
  List.fold_right (fun (x,y) (xs,ys) -> (x::xs, y::ys)) l ([], []) ;;
```

Problem 4 answer

In Haskell:

```
unzip :: [(a,b)] -> ([a], [b])
unzip tups =
  foldr (\(x,y) (xs,ys) -> (x:xs, y:ys)) ([], []) tups
```

Problem 5

Implement `splitAt`.

```
let splitAt (n:int) (l:'a list) : ('a list) * ('a list) = ??? ;;

splitAt 2 [1;2;3;4] = ([1;2],[3;4])
splitAt 0 [1;2;3;4] = ([], [1;2;3;4])
splitAt 4 [1;2;3;4] = ([1;2;3;4], [])
```

Problem 5 answer

Implement `splitAt`.

Common technique: define an "inner" function with explicit accumulator parameter(s), then have the "outer" function call the inner function with a initial accumulator value(s)

```
let splitAt (n:int) (l:'a list) : ('a list) * ('a list) =
  let rec splitAt_ n2 l2 acc = begin
    match (n2,l2) with
      (0, ys) -> (acc, ys)
    | (n, []) -> (acc, [])
    | (n, y::ys) -> splitAt_ (n-1) ys (acc @ [y])
  end in
  splitAt_ n l []
;;
```

Problem 5 answer

In Haskell:

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs = splitAt_ n xs []
  where splitAt_ 0 ys acc     = (acc, ys)
        splitAt_ n [] acc     = (acc, [])
        splitAt_ n (y:ys) acc =
          splitAt_ (n-1) ys (acc ++ [y])
```

More Problems

- Implement some of the functions from the `List` module
  - Haskell: `https://hackage.haskell.org/package/base-4.8.2.0/docs/Data-List.html`
  - Ocaml: `http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`