CMSC 330: Organization of Programming Languages

OCaml

Higher Order Functions

Anonymous Functions

- Recall code blocks in Ruby

```
(1..10).each { |x| print x }
```

  - Here, we can think of `{ |x| print x }` as a function

- We can do this (and more) in OCaml

Anonymous Functions

- As with Ruby, passing around functions is common
  - So often we don't want to bother to give them names
- Use fun to make a function with no name

Parameter

Body
(in which parameter x
is bound)

```
fun x -> x + 3
```

```
(fun x -> x + 3) 5
```

= 8

Anonymous Functions

- Syntax
  - **fun *x1* … *xn* -> *e***
- Evaluation
  - An anonymous function is an expression
  - In fact, *it is a value* – no further evaluation is possible
    - As such, it can be passed to other functions, returned from them, stored in a variable, etc.
- Type checking
  - (**fun *x1* … *xn* -> *e***) : (***t1* -> … -> *tn* -> *u***)

    when ***e*** : ***u*** under assumptions ***x1*** : ***t1*, …, *xn*** : ***tn***.
    - (Same rule as **let *f x1* … *xn* = *e***)

Calling Functions, Generalized

Not just a variable ***f***

- Syntax ***e0 e1 … en***
- Evaluation
  - Evaluate arguments ***e1*** … ***en*** to values ***v1*** … ***vn***
    - Order is actually right to left, not left to right
    - But this doesn't matter if ***e1*** … ***en*** don't have side effects
  - Evaluate ***e0*** to a function **fun *x1* … *xn* -> *e***
  - Substitute ***vi*** for ***xi*** in ***e***, yielding new expression ***e′***
  - Evaluate ***e′*** to value ***v***, which is the final result
- Example:
  - `(fun x -> x+x) 1` ⇒ `1+1` ⇒ `2`

Calling Functions, Generalized

- Syntax ***e0 e1*** … ***en***
- Type checking (almost the same as before)
  - If ***e0*** : ***t1*** **->** … **->** ***tn*** **->** ***u*** and ***e1*** : ***t1***, … , ***en*** : ***tn*** then ***e0 e1*** … ***en*** : ***u***
- Example:
  - **`(fun x -> x+x) 1 : int`**
  - since **`(fun x -> x+x)`**: **`int -> int`** and **`1`** : **`int`**

Quiz 1: What does this evaluate to?

```
let y = (fun x -> x+1) 2 in
(fun z -> z-2) y
```

A. Error

B. 2

C. 1

D. 0

Quiz 1: What does this evaluate to?

```
let y = (fun x -> x+1) 2 in
(fun z -> z-2) y
```

A. *Error*
B. **2**
C. **1**
D. **0**

Quiz 2: What is this expression's type ?

```
(fun x y -> x) 2 3
```

A. *Type error*

B. `int`

C. `int -> int -> int`

D. `'a -> 'b -> 'a`

Quiz 2: What is this expression's type ?

```
(fun x y -> x) 2 3
```

A. *Type error*

B. **`int`**

C. `int -> int -> int`

D. `'a -> 'b -> 'a`

Functions and Binding

- Functions are first-class, so you can bind them to other names as you like

```
let f x = x + 3;;
let g = f;;
g 5        = 8
```

- In fact, let for functions is syntactic shorthand

```
let f x = body
```

↓ is semantically equivalent to

```
let f = fun x ->  body
```

Example Shorthands

- `let next x = x + 1`
  - Short for `let next = fun x -> x + 1`

- `let plus x y = x + y`
  - Short for `let plus = fun x y -> x + y`

- ```
  let rec fact n =
        if n = 0 then 1 else n * fact (n-1)
  ```
  - Short for
    ```
    let rec fact = fun n ->
        (if n = 0 then 1 else n * fact (n-1))
    ```
    ```

Quiz 3: What does this evaluate to?

```
let f = fun x -> 0 in
let g = f in
g 1
```

A. *Error*

B. **2**

C. **1**

D. **0**

Quiz 3: What does this evaluate to?

```
let f = fun x -> 0 in
let g = f in
g 1
```

A. *Error*

B. **2**

C. **1**

D. **0**

Defining Functions Everywhere

```
let move l x =
  let left x = x - 1 in  (* locally defined fun *)
  let right x = x + 1 in (* locally defined fun *)
  if l then left x
  else      right x
;;

let move' l x = (* equivalent to the above *)
  if l then (fun y -> y - 1) x
  else      (fun y -> y + 1) x
```

Pattern Matching With Fun

- match can be used within fun

```
(fun l -> match l with (h::_) -> h) [1; 2]
        = 1
```

- But use named functions for complicated matches
- May use standard pattern matching abbreviations

```
(fun (x, y) -> x+y) (1,2)

        = 3
```

Passing Functions as Arguments

- In OCaml you can pass functions as arguments (akin to Ruby code blocks)

```
let plus_three x = x + 3 (* int -> int *)

let twice f z = f (f z) (* ('a->'a) -> 'a -> 'a *)
twice plus_three 5 = 11
```

- Ruby's `collect` is called `map` in OCaml
  - `map f l` applies function `f` to each element of `l`, and puts the results in a new list (preserving order)

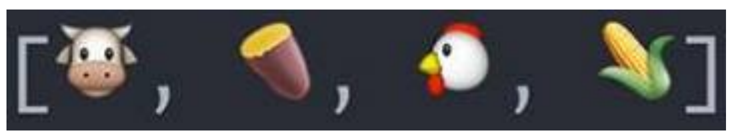
```
map plus_three [1; 2; 3] = [4; 5; 6]
map (fun x -> (-x)) [1; 2; 3] = [-1; -2; -3]
```

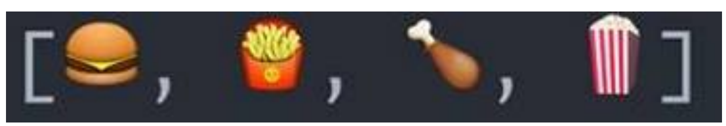
map function

What is Map?

Map generates a new list by applying a function to every item in the given list

```
map f [n1;n2;n3] == > [f n1; f n2;  f n3]
```

map cook 

== > 

Why do we need Map?

```
let rec double lst =
  match lst with
    []->[]
   |h::t-> h * 2 :: double t
```

```
let rec neg lst =
  match lst with
    []->[]
   |h::t-> h * (-1) :: neg t
```

```
double [1; 2; 3; 4];;
- : int list = [2; 4; 6; 8]
```

```
neg [1;2;3;4];;
- : int list = [-1; -2; -3; -4]
```

Why do we need Map?

```
let rec double lst =
  match lst with
    []->[]
   |h::t-> h * 2 :: double t
```

```
let rec neg lst =
  match lst with
    []->[]
   |h::t-> h * (-1) :: neg t
```

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

How to implement Map?

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

Type of Map

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

```
('a -> 'b) -> 'a list -> 'b list
```

How to use Map?

```
let double x  = x * 2 ;;

let lst = [1; 2; 3; 4; 5] ;;

let t = map double lst ;;
```

```
t : int list = [2; 4; 6; 8; 10]
```

Example 1

Subtract 1 from every item in an int list

```
let t = [1; 2; 3; 4];;
map (fun x-> x-1) t;;
```

```
let t = [1; 2; 3; 4];;
let sub1 x = x - 1;;
map sub1 t;;
```

```
int list = [0; 1; 2; 3]
```

Example 2

Negate every item in an int list

```
let t = [1; 2; 3; 4];;
let neg x = x * (-1);;
map neg t;;
```

```
int list = [-1; -2; -3; -4]
```

Example 3

Apply a list functions to an int list

```
let lst = [1;2;3];;
let neg x = x * (-1);;
let sub1 x = x-1;;
let double x = x + x;;
```

```
let fs = [neg; sub1; double];;
map (fun x -> map x lst) fs;;
```

```
int list list = [[-1; -2; -3]; [0; 1; 2]; [2; 4; 6]]
```

Example 4: Permute a list

```
let permute lst =
  let rec rm x l = List.filter ((<>) x) l
  and insertToPermute lst x =
    let t = rm x lst in
    List.map ((fun a b->a::b) x )(permuteall t)
  and permuteall lst =
    match lst with
    |[]->[]
    |[x]->[[x]]
    |_->List.flatten(List.map (insertToPermute lst) lst)
  in permuteall lst
;;
```

```
# permute [1;2;3];;
- : int list list =
[[1; 2; 3]; [1; 3; 2]; [2; 1; 3]; [2; 3; 1]; [3; 1; 2];
[3; 2; 1]]
```

Example 5: Power Set

```
let populate a b =
  if b=[] then [[a]]
  else  let t = List.map (fun x->a::x) b in
        [a]::t@b
;;

let powerset lst = List.fold_right populate lst []
;;
```

```
# populate 1 [[2];[3]];;
- : int list list =
- [[1]; [1; 2]; [1; 3]; [2];
  [3]]
```

```
# powerset [1;2;3];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]; [1; 3];
[2]; [2; 3]; [3]]
```

What we learned?

- Map:
  - A higher order function.
  - List module
  - Takes a function and a list as arguments, applies the function to each member of the list, generates a new list
  - It is powerful.

fold function

What is Fold

- Fold generally
  - takes a function of two arguments, a list, and an initial value (accumulator)
  - combines the list by applying the function to the accumulator and one element from the list and the result of recursively folding the function over the rest of the list.

Accumulator: (i.e. 0 for addition, 1 for multiplication, false for boolean OR, negative infinity for maximum, etc.)

What is Fold

```
fold (fun x y-> x+y) 0 [1;2;3;4;5];;

- : int = 15
```

Why do we need Fold?

sum a list of integers

```
let rec sum l =
  match l with
  [] -> 0
  |h::t -> h + (sum t)
```

```
 sum [1;2;3;4];;
- : int = 10
```

Concatenate a list of strings:

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

```
concat ["a";"b";"c"];;
- : string = "abc"
```

Why do we need Fold?

sum a list of integers

```
let rec sum l =
  match l with
  [] -> 0
  |h::t -> h + (sum t)
```

Concatenate a list of strings:

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

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

How to implement Fold

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

Type of Fold

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

```
    f                    acc     lst    -> return type

('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

How to use Fold?

```
let add x y  = x + y ;;

let lst = [2; 3; 4] ;;

let t = fold add 0 lst ;;


t : int = 9
```

How to use Fold?

```
let add x y  = x + y ;;
let lst = [2; 3; 4] ;;
let t = fold add 0 lst ;;
t : int = 9
```

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

```
fold add 0 lst
fold add (add 0 2) [3;4]
fold add    2      [3;4]
fold add (add 2 3) [4]
fold add     5     [4]
fold add (add 5 4) [ ]
fold add 9 [ ]
9
```

Example 1: Product of an int list

```
let mul x y = x * y;;

let lst = [1; 2; 3; 4; 5];;

fold mul 1 lst
- : int = 120
```

Wrong accumulator

```
 fold mul 0 lst;;
- : int = 0
```

Example 2: Count elements of a list satisfying a condition

```
let countif p l =
fold (fun counter element -> if p element then counter+1
                             else counter) 0 l ;;


countif (fun x -> x > 0) [30;-1;45;100;0];;

- : int = 3
```

Exaple 3: Collect even numbers in the list

```
let f acc y = if (y mod 2) = 0 then y::acc
              else acc;;

fold f [] [1;2;3;4;5;6];;

- : int list = [6; 4; 2]
```

Example 4: Inner Product

first compute list of pair-wise products, then sum up

```
  [x1;x2;x3]*[y1;y2;y3] = [x1*y1 + x2*y2 + x3*y3]

let rec map2 f a b =
       match (a,b) with
       |([],[])->([])
       |(h1::t1,h2::t2)->(f h1 h2):: (map2 f t1 t2)
       |_->invalid_arg "map2";;

let product v1 v2 =
      fold (+) 0 (map2 ( * ) v1 v2);;
# val product : int list -> int list -> int = <fun>
product [2;4;6] [1;3;5];;
#- : int = 44
```

Example 5: Find the maximum from a list

```
let maxList lst =
       match lst with
        []->failwith "empty list"
       |h::t-> fold max h t ;;

maxList [3;10;5];;
- : int = 10
```

```
(*
maxList [3;10;5]
fold max 3 [10:5]
fold max (max 3 10) [5]
fold max (max 10 5) []
fold max 10 []
10   *)
```

Quiz: Sum of sublists

Given a list of int lists, compute the sum of each int list, and return them as list.

For example:

```
sumList [[1;2;3];[4];[5;6;7]]
- : int list = [6; 4; 18]
```

Solution: Sum of sublists

```
let sumList  = map (fold (+) 0 );;

sumList [[1;2;3];[4;5;6];[10]];;
- : int list = [6; 15; 10]
```

Quiz: Maximum contiguous subarray

Given an int list, find the contiguous sublist, which has the largest sum and return its sum.

Example:

Input: [-2,1,-3,**4,-1,2,1**,-5,4]

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6

Quiz: Maximum contiguous subarray

```
let f (m, acc) h =
   let m = max m (acc + h) in
   let  x = if acc < 0 then 0 else acc in
   (m, x+h)
;;
let submax  lst = let (max_so_far, max_current) =
           fold f (0,0) lst in
           max_so_far
;;
```

```
 submax [-2; 1; -3; 4; -1; 2; 1; -5; 4];;
- : int = 6
```

Summary

- **map** ***f*** **[*v1*; *v2*; …; *vn*]**

  = [*f v1*; *f v2*; …; *f vn*]

  - e.g., `map (fun x -> x+1) [1;2;3] = [2;3;4]`

- **fold** ***f*** ***v*** **[*v1*; *v2*; …; *vn*]**

= fold *f* (*f v v1*) [*v2*; …; *vn*]

= fold *f* (*f* (*f v v1*) *v2*) […; *vn*]

= …

= *f* (*f* (*f* (*f v v1*) *v2*) …) *vn*

- e.g., `fold add 0 [1;2;3;4] =`

  `add (add (add (add 0 1) 2) 3) 4 = 10`

Quiz 4: What does this evaluate to?

```
map (fun x -> x *. 4) [1;2;3]
```

A. `[ 1.0; 2.0; 3.0 ]`
B. `[ 4.0; 8.0; 12.0 ]`
C. Error
D. `[4; 8; 12 ]`

Quiz 4: What does this evaluate to?

```
map (fun x -> x *. 4) [1;2;3]
```

A. `[ 1.0; 2.0; 3.0 ]`

B. `[ 4.0; 8.0; 12.0 ]`

C. **Error -- the *. function takes floats, not ints**

D. `[4; 8; 12 ]`

Quiz 5: What does this evaluate to?

```
fold (fun a y -> y::a) [] [3;4;2]
```

A. `[ 9 ]`

B. `[ 3;4;2 ]`

C. `[ 2;4;3 ]`

D. Error

Quiz 5: What does this evaluate to?

```
fold (fun a y -> y::a) [] [3;4;2]
```

A. [9]

B. [3;4;2]

C. [2;4;3]

D. Error

Quiz 6: What does this evaluate to?

```
let is_even x = (x mod 2 = 0) in
map is_even [1;2;3;4;5]
```

A. `[false;true;false;true;false]`
B. `[0;1;1;2;2]`
C. `[0;0;0;0;0]`
D. `false`

Quiz 6: What does this evaluate to?

```
let is_even x = (x mod 2 = 0) in
map is_even [1;2;3;4;5]
```

A. `[false;true;false;true;false]`

B. `[0;1;1;2;2]`

C. `[0;0;0;0;0]`

D. `false`

Combining map and fold

- Idea: map a list to another list, and then fold over it to compute the final result
  - Basis of the famous “map/reduce” framework from Google, since these operations can be parallelized

```
let countone l =
  fold (fun a h -> if h=1 then a+1 else a) 0 l

let countones ss =
  let counts = map countone ss in
  fold (fun a c -> a+c) 0 counts

countones [[1;0;1]; [0;0]; [1;1]] = 4
countones [[1;0]; []; [0;0]; [1]] = 2
```

fold_right

- Right-to-left version of fold:

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

- Left-to-right version used so far:

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

Left-to-right vs. right-to-left

```
fold f v [v1; v2; …; vn] =
 f (f (f (f v v1) v2) …) vn

fold_right f [v1; v2; …; vn] v =
 f (f (f (f vn v) …) v2) v1
```

```
fold (fun x y -> x - y) 0 [1;2;3] = -6
```

since ((0-1)-2)-3) = -6

```
fold_right (fun x y -> x - y) [1;2;3] 0 = 2
```

since 1-(2-(3-0)) = 2

When to use one or the other?

- Many problems lend themselves to **`fold_right`**
- But it does present a performance disadvantage
  - The recursion builds of a deep stack: One stack frame for each recursive call of fold_right
- An optimization called tail recursion permits optimizing **`fold`** so that it uses no stack at all
  - We will see how this works in a later lecture!