CMSC 330: Organization of Programming Languages

Functional Programming with OCaml
OCaml Equality

# let a1 = [1;2;3];;
val a1 : int list = [1; 2; 3]
# let a2 = [1;2;3];;
val a2 : int list = [1; 2; 3]
# a1 == a2;;
- : bool = false  (shallow equality)
# a1 = a2;;
- : bool = true   (deep equality)

- <> is negation of =
- != is negation of ==
Recursion Exercises

• sum l (* sum of elts in l *)

• negate l (* negate elements in list *)

• last l (* last element of l *)
Recursion Exercises

• sum l (* sum of elts in l *)
  let rec sum l = match l with
      [] -> 0
    | (x::xs) -> x + (sum xs)

• negate l (* negate elements in list *)
  let rec negate l = match l with
     [] -> []
  | (x::xs) -> (-x) :: (negate xs)

• last l (* last element of l *)
  let rec last l = match l with
   [x] -> x
 | (x::xs) -> last xs
Recursion Exercises (cont'd)

(* return a list containing all the elements in the list l followed by all the elements in list m *)

• append (l, m)

• rev l  (* reverse list; hint: use append *)

• \texttt{rev} takes $O(n^2)$ time. Can you do better?
Recursion Exercises (cont'd)

(* return a list containing all the elements in the list l followed by all the elements in list m *)

- append (l, m)
  
  let rec append (l, m) = match l with
  
  | [] -> m
  | (x::xs) -> x::(append (xs, m))

- rev l  (* reverse list; hint: use append *)
  
  let rec rev l = match l with
  
  | [] -> []
  | (x::xs) -> append ((rev xs), [x])

- rev takes $O(n^2)$ time. Can you do better?
A Clever Version of Reverse

let rec rev_helper (l, a) = match l with
    [] -> a
  | (x::xs) -> rev_helper (xs, (x::a))
let rev l = rev_helper (l, [])

• Let’s give it a try

  rev [1; 2; 3] →
  rev_helper ([1;2;3], []) →
  rev_helper ([2;3], [1]) →
  rev_helper ([3], [2;1]) →
  rev_helper ([], [3;2;1]) →
  [3;2;1]
More Examples

• flattenPairs l (* (a * a) list -> a list *)

• take (n, l) (* return first n elts of l *)
More Examples

• flattenPairs l (* ('a * 'a) list -> 'a list *)
  let rec flattenPairs l = match l with
      [] -> []
    | ((a, b)::t) -> a :: b :: (flattenPairs t)

• take (n, l) (* return first n elts of l *)
  let rec take (n, l) =
      if n = 0 then []
    else match l with
        [] -> []
      | (x::xs) -> x :: (take (n-1, xs))
Working with Lists

• Several of these examples have the same flavor
  – Walk through the list and do something to every element
  – Walk through the list and keep track of something

• Recall the following example code from Ruby:

  ```ruby
  a = [1,2,3,4,5]
  b = a.collect { |x| -x }
  ```

  – Here we passed a code block into the collect method
  – Wouldn’t it be nice to do the same in OCaml?
Higher-Order Functions

- In OCaml you can pass functions as arguments, and return functions as results

```ocaml
let plus_three x = x + 3
let twice (f, z) = f (f z)
twice (plus_three, 5) (* returns 11 *)
twice : ('a->'a) * 'a -> 'a

let plus_four x = x + 4
let pick_fn n =
  if n > 0 then plus_three else plus_four
(pick_fn 5) 0 (* returns 3 *)
pick_fn : int -> (int->int)
```
The map Function

• Let’s write the map function (just like Ruby's collect)
  – Takes a function and a list, applies the function to each element of the list, and returns a list of the results

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

let add_one x = x + 1
let negate x = -x

map (add_one, [1; 2; 3])
map (negate, [9; -5; 0])

• Type of map?
  map : ('a -> 'b) * 'a list -> 'b list
Anonymous Functions

• Passing functions around is very common
  – So often we don’t want to bother to give them names

• Use `fun` to make a function with no name

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

```
map ((fun x -> x + 13), [1; 2; 3])

twice ((fun x -> x + 2), 4)
```
Pattern Matching with fun

• **match** can be used within **fun**

```ocaml
map ((fun l -> match l with (h::_) -> h),
     [ [1; 2; 3]; [4; 5; 6; 7]; [8; 9] ])
(* [1; 4; 8] *)
```

– For complicated matches, though, use named functions

• Standard pattern matching abbreviation can be used

```ocaml
map ((fun (x, y) -> x + y), [(1, 2); (3, 4)])
(* [3; 7] *)
```
All Functions Are Anonymous

• 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 (* returns 8 *)`

• `let` for functions is just a shorthand
  - `let f x = body` stands for
  - `let f = fun x -> body`
Examples

• 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
  – Which is short for
    * let plus = fun z ->
      (match z with (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))
The fold Function

• Common pattern: iterate through a list and apply a function to each element, keeping track of the partial results computed so far

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

- a = “accumulator”
- this is usually called “fold left” to remind us that f takes the accumulator as its first argument

• What's the type of fold?

```
fold : ('a * 'b -> 'a) * 'a * 'b list -> 'a
```
Example

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

```ocaml
let add (a, x) = a + x
fold (add, 0, [1; 2; 3; 4]) →
fold (add, 1, [2; 3; 4]) →
fold (add, 3, [3; 4]) →
fold (add, 6, [4]) →
fold (add, 10, []) →
10
```

We just built the `sum` function!
Another Example

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

```
let next (a, _) = a + 1
fold (next, 0, [2; 3; 4; 5]) →
fold (next, 1, [3; 4; 5]) →
fold (next, 2, [4; 5]) →
fold (next, 3, [5]) →
fold (next, 4, []) →
4
```

We just built the length function!
Using fold to Build rev

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

• Can you build the reverse function with fold?

```
let prepend (a, x) = x::a
fold (prepend, [], [1; 2; 3; 4]) →
fold (prepend, [1], [2; 3; 4]) →
fold (prepend, [2; 1], [3; 4]) →
fold (prepend, [3; 2; 1], [4]) →
fold (prepend, [4; 3; 2; 1], []) →
[4; 3; 2; 1]
```