More Basics…

```ocaml
# let l1 = [1;2;3];;
val l1 : int list = [1; 2; 3]

# let l2 = [1;2;3];;
val l2 : int list = [1; 2; 3]

# l1 == l2;;
- : bool = false (shallow equality)

# l1 = l2;;
- : bool = true (deep equality)
```

More Examples of Recursion

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

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

- `last l` (* last element of l *)
  ```ocaml
  let rec last l = match l with
  | [] -> []
  | (x::xs) -> last xs
  ```

More Examples (cont’d)

- `append (l, m)` (* return a list containing all the elements in the list l followed by all the elements in list m *)
  ```ocaml
  let rec append (l, m) = match l with
  | [] -> m
  | (x::xs) -> x::(append (xs, m))
  ```

- `rev l` (* reverse list; hint: use append *)
  ```ocaml
  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

```ocaml
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 [2; 3; 1] ⨿
rev_helper ([2;3;1], []) ⨿
rev_helper ([3;2;1], [1]) ⨿
rev_helper ([3], [2;1]) ⨿
rev_helper ([1], [3;2;1]) ⨿
[3;2;1]
```
More Examples

- **flattenPairs** (* ('a * 'a) list -> 'a list *)

  let rec flattenPairs l = match l with
  | [] -> []
  | ((a, b)::t) -> a :: b :: (flattenPairs t)

- **take** (* (n, l) -> 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)
  ```

  ```ocaml
  twice : ('a->'a) * 'a -> 'a
  ```

  ```ocaml
  let plus_four x = x + 4
  let pick_fn n = if n > 0 then plus_three else plus_four
  ```

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

  ```ocaml
  map : ('a -> 'b) * 'a list -> 'b list
  ```

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

  ```ocaml
  map ((fun x -> x + 2), [1; 2; 3; 4; 5; 6; 7; 8; 9])
  ```

  ```ocaml
  (* [1; 4; 8] *)
  ```

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

  ```ocaml
  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

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

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

  ```ocaml
  twice ((fun x -> x + 2), 4)
  ```

Pattern Matching with fun

- `match` can be used within `fun`

  ```ocaml
  map ((fun l -> match l with (b::_) -> b),
  ```

  ```ocaml
  [ [1; 2; 3]; [4; 5; 6; 7]; [8; 9] ])
  ```

  ```ocaml
  (* [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)])
  ```

  ```ocaml
  (* [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
  - `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`?

Examples

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

Another Example

- `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)`

- `let prepend (a, x) = x::a`
- `fold (prepend, [], [1; 2; 3; 4])` ➞
- `fold (prepend, [1], [2; 3; 4])` ➞
- `fold (prepend, [2; 3; 4])` ➞
- `fold (prepend, [3; 4; 5])` ➞
- `fold (prepend, [4; 5; 6])` ➞
- `fold (prepend, [5; 6; 7])` ➞
- `fold (prepend, [6; 7; 8])` ➞
- `fold (prepend, [7; 8; 9])` ➞
- `fold (prepend, [8; 9; 10])` ➞

- `We just built the `sum` function!`

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

- `We just built the `length` function!`