OCaml

- Pattern matching and recursion
- there is cost in functions and memory management

Patterns

- used to take apart data structures

```
match E with
| P1 -> E1
| P2 -> E2
...
| PN -> EN
```

- basic patterns
  - 0 - (any constant) value that it matches
  - a - (any identifier) any value (binds identifier to that value)
  - __ - don't care pattern
  - (P) whatever P matches
  - P1, P2, . . ., PN - any n-tuple where the first component matches P1
  - $[\mathrm{P1}; \mathrm{P2}; ...; \mathrm{PN}]$ - any list of length n, with corresponding submatches
  - P1 :: P2 - any list of length >0 s.t. P1 matches the first item, P2 matches the rest of the list

```
let cons(a, b) = a :: b
let cons x = match x with | (a, b) -> a :: b
let cons = fun x -> (match x with | (a, b) -> (a :: b))
```

^ All the statements have the same machine code

Defining Recursive Functions

```
let rec (not plain rec)
let ID = EXPR -> general rule is ID can't be in EXPR
let n = n + 1 -> violates rule
let n = n + 0 -> violates rule
```

```
@ - concatenate 2 lists
A @ B - O(|A|)
A :: B - O(1)
reverse A is O(|A|^2)

let rec rev = fun a -> fun l ->
match l with
| [] -> a
| h :: t -> rev (h :: a) t

Or equivalently

let rec rev = fun a -> function
| [] -> a
| h :: t -> rev (h :: a) t

let reverse2 = rev [];;

let reverse3 =
  (let rec rev = fun a -> function
       | [] -> a
       | h :: t -> rev (h :: a) t
    in rev [])

rev a l returns the reverse of l concatenated with a

fun x y z -> E == fun x -> fun y -> fun z -> E
```

fun - currying
function - pattern matching

min of list:

```
let min = function
| h :: t ->
    let tmin = min t in
        if h < tmin then
            h
```

```
        else
            tmin
```

Defining your own types

```
type myfntype = int -> int

discriminant union type:
my dutype =
| foo
| bar of int
| baz of int * string
```