# CMSC330 NOTES - OCAML

## I. Data Types

### 1.int

operator: **+, -, *, /, mod**
others: abs, max_int, min_int
note: 5/2 = 2

### 2. float

operator: **+., -., *., /.**, **, sqrt (or ** 0.5), log (ln),...
note: 2.5 +. 2 will raise an error (it must be 2.)

### 3. string

operator: ^ (concatenate)
note: string is surrounded with " ", char is surrounded with ' ' but char can only contain 1 character ('hi' will raise a syntax error)

### 4. bool

operator: **&&, ||, not** (weird, it's **not** NOT !)
comparisons: =, <>, <, <=, >, >=

There are functions that help us convert any of those types to another type:
**type1_of_type2 v: type1 → type2 (change v from type2 to type1)**
example: int_of_float 5.46 => int of 5.46 = 5
note: there are only bool_of_string and string_of_bool, not bool_of_int or others
Other helpful functions:
compare: 'a → 'a → int (compare x y return 0 if x is equal y, neg int if x < y,...)
min: 'a → 'a → 'a
max: 'a → 'a → 'a

### 5. list (all elements in a list must be **same type**, separated by ; inside [])

operator: **x::lst** (unshift x to lst ), **lst1@lst2** (concat lst1 & lst2)
note: if want to push x to lst, do lst@[x]
example:
```
1::2::[3;4;5];;      => [1;2;3;4;5]
[1;2]@[3;4;5];;      => [1;2;3;4;5]
```

### 6. tuple (elements in a tuple can be **different types**, separated by , inside ())

no operator since it is immutable
example: ([2;3;1],"hi",true) => int list * string * bool

### 7. record (weird hash)

define: **type record_name = {key_1:val_1; key_2:val_2; ...}**
get val: record_name.key
example:
```
type data = { month: string; day: int; year: int };;
let today = { day=1; year=2023; month="mar"};;
today.year;; (* 2023 *)
```
note: notice the order does not matter but must be enough!

### 8. variant (enum)

define: **type variant_name = Type_1 of 'a | Type_2 of 'b | ...**
important: Type_1, Type_2,... **must be capitalized the first letter**
example:
```
type linked = Null | Item of string * linked;;
let items = Item("banana", Item("apple", Null));;
```
note: can be generic
```
type 'a option = Some of 'a | None;;
let opt = Some 2;;   => val opt: int option
```

## II. Functions

### 1. anonymous function

example: fun x y z -> x + y + z;; (* int → int → int → int *)

### 2. 'normal' function

example: let sum_1 x y z = x + y + z;; (same type but now its name is sum_1)
or let sum_2 = fun x y z -> x + y + z;; (same idea but different :D)
sum_1 1 2 3;; => 6  sum_2 1 2 3;; => 6

### 3. recursive function (must have rec keyword)

example: let **rec** fac n = if n = 1 then 1 else n * fac (n - 1);; (* int → int *)

III. More on `let`

**1. let bindings**

a let binding is **not an expression** and just binds an expression to a variable.

```
example: let a = 10;; (val a : int = 10)
         let add a b = a + b;; (val add : int → int → int = <fun>)
```

=> a or add can be re-used later.

note: we should consider **add** as a variable but its type is <fun> (function)

**2. let expressions**

a let expression is like setting a local variable to be used in **another expression**.

```
example: let a = 10 in
         let add a b = a + b in (* a in this let is different from the earlier a *)
         add 15 a;; (* output: 25 *)
```

=> a can't be re-used later.

note: let expressions mostly make our code more readable

IV. Pattern Matching

**0. Syntax**

```
match x with
  |p1 -> e1       (* | is optional for this line *)
  |p2 -> e2
  ...
  |_  -> e3
```

note: x, p1, p2 must be the same type
e1, e2, e3 must be the same type
_ is wildcard, simply make our matching exhaustive (match all possible cases)

**1. int, float, string, char, or bool**

nothing special, just like switch statement in other programming languages

**2. list**

```
match lst with
  |[] -> e1              (* empty list *)
  |h::[] -> e2           (* list with exactly 1 element *)
  |h1::h2::[] -> e3      (* list with exactly 2 elements h1 and h2 *)
  |h::t -> e4      (* list with more than 2 elements but h is the first element*)
```

note: e1, e2, e3, e4 are the same type, the idea here is to extract values from list

**3. tuple**

since length of tuple is fixed, be careful when matching it

```
example: match tup with (x,y) -> x=y;;
error-example: match tup with
                 |(x,y) -> x + y
                 |(x,y,z) -> x + y + z;; (* don't do this *)

valid-example: match (1,2,3) with
                 |(2,_,a) -> a
                 |(1,2,a) -> a + 1   (* match here *)
                 |(_,_,a) -> a + 2;; (* output: 4 *)
```

**4. record**

```
type data = { month: string; day: int; year: int };;
let today = { day=29; year=2023; month="feb"};;
match today with
   |{day=4; month="jun"} -> "cool, my birthday"
   |_ -> "sad, not my birthday";; (* output: "sad, not my birthday" *)
```

**5. variant**

```
type linked = Null | Item of string * linked;;
let items = Item("banana", Item("apple", Null));;
let rec count_items items = match items with
  |Null -> 0 (* base case *)
  |Item(name, next) -> 1 + count_items next;;
count_items items;; (* output: 2 *)
```

V. Higher Order Functions

**1. map**

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

type: val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

note: we use map() when the output is a list having the same length as input list

**2. fold left**

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

type: val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

note: 1. when the output of function is completely different from the input -> fold
2. the type of a (accumulator) must be same as output type, also a base case
3. the function we use for fold is on element inside lst, usually go with 1 or more restrict conditions, hardest part, try to find it.

example:
count_occ lst target
Type: 'a list -> 'a -> int
Description: returns how many elements in lst are equal to target.

=> analysis:
1. input type is 'a list, but output is int => use fold
2. what is type of a? => output type => int
base case? => no occurrence => 0
3. what is a function we need?
=> we know: that fun is on every element in the list
=> we want: what is relation between that element and our concern?
=> we think: is it equal to target?
=> we do: if yes then increase accumulator by 1, if no then nothing happens
=> we implement: how can we write that fun? (* it's fun but not fun :D *)

=> final answer:

```
count_occ lst target = fold_left (fun a h -> if h = target then a + 1 else a) 0 lst;;
```

**3. fold right**

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

technically, fold_right is similar to fold_left but we iterate each element from right to left instead of from left to right

note: the order of arguments is slightly different from fold_left

*Illustration of fold_left and fold_right

**fold_left f init [a;b;c] => f (f (f init a) b) c**
**fold_right f [a;b;c] init => f a (f b (f c init))**