Pattern matching
Pattern matching in OCaml can break apart data structures and do pattern matching on the data. Here is the syntax:

```
    match e with
       p1 -> e1
     | p2 -> e2
     | p3 -> e3
```

pattern matching will detect:

- missed cases
- unused case

wild card for catch all. Be careful when you use it.

Type Checking rules

If e and p1, ..., pn each have type ta and e1, ..., en each have type tb, then entire match expression has type tb

Pattern matching examples:

```
match 1+2 with
3 -> true
| _ -> false;;
```

Check if a value is odd or not

```
let is_odd x =
match x mod 2 with
    0 -> false
       | 1 -> true
       | _ -> raise (Invalid_argument "is_odd");;    (* why do we need this? *)
```

Negate a value

```
let neg b =
        match b with
        | true -> false
        | false -> true;;
        val neg : bool -> bool = <fun>

        neg true;;
                - : bool = false
        neg (10 >20);;
                - : bool = true
```

Logical implication

```
let imply v = match v with
 (true,true)   -> true
| (true,false)  -> false
| (false,true)  -> true
 | (false,false) -> true;;

        val imply : bool * bool -> bool = <fun>


 let imply v = match v with
 (true,x)  -> x
| (false,x) -> true;;

                val imply : bool * bool -> bool = <fun>
```

For characters, OCaml also recognizes the range patterns in the form of 'c1' .. 'cn' as shorthand for any ASCII character in the range.

```
let is_vowel = function
 ('a' | 'e' | 'i' | 'o' | 'u') -> true
        | _ -> false;;


let is_upper = function
        'A' .. 'Z' -> true
        | _ -> false;;
```

Abbreviated pattern matching

```
let f p = e
```

is the same as

```
let f x = match x with p -> e
```

Examples:

```
let hd (h::_) = h

let f(x::y::_) = x + y

let g [x; y] = x + y
```

Pattern matching with lists

```
let x = [1;2];;
match x with
[]  -> print_string "x is an empty list\n"
| _   -> print_string "x is anything but an empty list\n";;
```

You probably won't do things quite like the following, but...

```
let addFirsts ((x::_) :: (y::_) :: _) = x + y;;

addFirsts [ [1;2;3]; [4;5]; [7;8;9] ];;
```

Will the following work?

```
addFirsts [ [1;2;3]; [4;5]; [7;8;9]; [10;11;12] ];;
```

We can read data out of a list using a pttern matching.

```
let is_empty ls =
match ls with
[] -> true
| (h::t) -> false;;

is_empty [];;
is_empty [1;2];;
is_empty [1];;
is_empty [ [] ];;
```

Get the head of the list

```
let hd ls = match ls with (h::_) -> h;;
hd [];;
```

More examples:

```
let f ls = match ls with (h1::(h2::_)) -> h1 + h2;;

f [2;4;8];;
        - : int = 6


let g ls = match ls with [h1; h2] -> h1 + h2;;
g [1;2];;
        - : int = 3

g [1;2;3];;
        Exception: Match_failure
```

Lists and Recursive Funcions

get a head of a list

```
let hd l =
        match l with
        []->[]
        |h::t-> [h]
        ;;
```

get the last element of a list

```
let rec last l=
        match l with
        []->[]
        |[x]->[x]
        |_::t->last t
;;
```

Or

```
let rec last l=
        match l with
        []->None
        |[x]->Some x
        |_::t->last t
;;
```

calculate the length of a list

```
        let rec length lst =
                match lst with
                |[]->0
                |_::t->1 + length t
        ;;
```

calculate the sum of a int list

```
        let rec sum lst=
                match lst with
                |[]->0
                |h::t->h + sum t
        ;;
```

check if x is member of a list

```
        let rec member lst x=
                match lst with
                |[]->false
                |h::t->if h = x then true else member t x
        ;;
```

append list b to list a

```
        let rec append a b=
                match a with
                |[]->b
                |h::t-> h::append t b
        ;;
```

insertion sort

```
        let rec insert x l=
                match l with
                        |[]->[x]
                        |h::t->if x < h then x::h::t
                                        else h::insert x t
        ;;

        let rec sort l =
                match l with
                []->[]
                |[x]->[x]
                |h::t->insert h (sort t)
        ;;
```