TREES

Inductive definitions:
Base cases
Rules for adding new elements.

Ex 1 $0 \in Nat$
if $n \in Nat$ then $succ(n) \in Nat$

$$Nat_0 = \emptyset$$
$$Nat_1 = \{0\}$$
$$Nat_2 = \{0, succ(0)\}$$
$$\vdots$$
$$Nat = \bigcup_{k \geq 0} Nat_k$$

Ex 2 $LIST = \{[]\} \oplus \cup\ Nat \times LIST$
A recursively defined type.

$$LIST_0 = \{\}$$
$$LIST_1 = \{[]\}$$
$$LIST_2 = \{[0], [1], [2], \ldots\}$$
$$\vdots$$
$$LIST_{k+1} = \{[]\} \cup Nat \times LIST_k$$

$$LIST = \bigcup_{k \geq 0} LIST_k$$

You can build your own recursive (inductive) types.

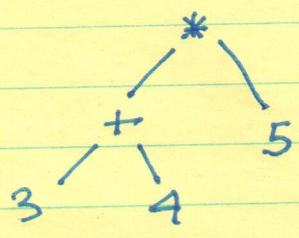
```
let max (n, m) = if n < m then m else n

let rec height (t : 'a tree) =
    match t with
  | Empty -> 0
  | Node (l, _, r) -> 1 + max (height l, height r)

let rec inOrder (t : int tree) =
    match t with
  | Empty -> print_string ""
  | Node (l, n, r) -> (inOrder l); (showInt n); (inOrder r)
                ; -> sequencing
```

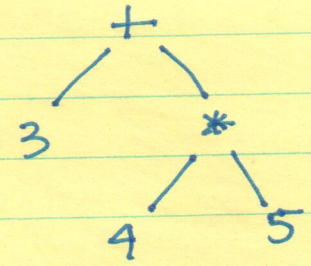
The type int tree -> unit,
A special type containing only one element ()

```
let rec flatten (t : 'a tree) =
    match t with
  | Empty -> []
  | Node (l, v, r) -> v :: ((flatten l) @ (flatten r))
```

EXPRESSION TREES

(3+4)*5 → 23





How to handle situations when we fail to find something we are looking for?

Option type

if 'a is any type 'a option is a new type containing all the values of 'a together with a special value to indicate that you did not find what you are looking for.

None

Some 17 → special tag Some

You can pattern match on none & some

How to abort a computation:

exception Not Found : declaring an
my chosen name exception

raise <exception name>

```
type exptree = Const of int
             | Var of char
             | Plus of exptree * exptree
             | Times of exptree * exptree

type binding = char * int
type env     = binding list

let rec eval (e : exptree) (tho : env) =
  match e with
  | Const n -> n
  | Var v -> (lookup v tho) matc (lookup v tho) with
      | None -> raise Not_Found
      | Some r -> r

  | Plus (e1, e2) ->
      let v1 = eval e1 tho in
      let v2 = eval e2 tho in
      v1 + v2
  | Times (e1, e2) ->
      let v1 = eval e1 tho in
      let v2 = eval e2 tho in
      v1 * v2

let rec lookup name (tho : env) =
  match tho with
  | [] -> None
  | (n,v) :: e -> if name = n then (Some v)
                  else lookup name e
```

type constructor