```
//1.1
let rec apply x rel =
    match rel with
    | (x', ys')::rels when (x = x') -> ys'
    | _::rels -> apply x rels
    | [] -> [];

let rel1 = [(1, ["a"; "b"; "c"]);(4, ["b";"e"])];;

apply 1 rel1

//1.2
let rec isMember x = function
    | y::ys -> x=y || (isMember x ys)
    | [] -> false;;

let inRelation x y rel = isMember y (apply x rel);;

inRelation 1 "c" rel1;;
inRelation 2 "c" rel1;;

//1.3
let rec insert x y rel =
    match rel with
    | (x', ys)::rels when (x = x') -> (x, y::ys)::rels
    | (x', ys)::rels when (x < x') -> (x, [y])::((x', ys)::rels)
    | curr::rels -> curr::(insert x y rels)
    | [] -> [(x, [y])];;

insert 1 "d" rel1;;
insert 2 "m" rel1;;

//1.4
let rec aux pairs rel =
    match pairs with
    | (x,y)::rest when (List.isEmpty (apply x rel)) -> aux rest ((x,[y])::rel)
    | (x,y)::rest -> aux rest (insert x y rel)
    | [] -> [];;

let rec toRel pairs = aux pairs List.empty;;

toRel [(2,"c");(1,"a");(2,"b")];;
aux [(2,"c");(1,"a");(2,"b")] [];;

//2.1
let multTable n = Seq.take 10 (Seq.initInfinite (fun i -> n*i));;
multTable 3
//2.2
let tableOf n m f = seq {for i in [1..n] do
                           for j in [1..m] do
                                yield (i,j,f i j) }

```

```

//2.3
let infA = Seq.initInfinite (fun v -> String.replicate (v+1) "a");;
infA;;

//2.4 - int -> int list -> int list
//f adds i^(i*index+1) to each element in the input list.
#time
let rec f i = function
    | [] -> []
    | x::xs -> (x+i)::f (i*i) xs;;

f 2 [1..1000];;

//2.5
//1
let rec fA i = function
    | (a, []) -> List.rev a
    | (a, x::xs) -> fA (i*i) (((x+i)::a),xs);;

fA 2 ([],[1..1000]);;

//2
let rec fC i c = function
    | [] -> c []
    | x::xs -> fC (i*i) (fun v -> c(x+i::v)) xs;;

fC 2 id [1..1000];;


//3.1
type T<'a> = N of 'a * T<'a> list;;

N ("a",[]);;

N ("i",[N ("j",[])]);;

let p1 = N ("p",[N("q",[N("r",[])])]);;


//3.2
//f: T<'a> -> 'a list
//g: T<'a> list -> 'a list

//f and g computes a concatenated list of all the variables of type 'a in T<'a> element, when matching it as N(e,es).

//h: ('a -> bool) -> T<'a> -> T<'a>

//h takes a T<'a> element as input and then iterates through the element,
//and then stops whenever P is true for the current element, and then outputs however far it came.
```

```
//example
//p e = e = "q"
//t N ("p",[N("q",[N("r",[])])])
//h p t results in N ("p",[N ("q",[])])


//k: T<'a> -> int

//k counts the number of 'a elements in the T<'a> element.


let rec f1(N(e,es)) = e :: g es
and g = function
    | [] -> []
    | e::es -> f1 e @ g es;;

f1(p1)

let rec h p t =
    match t with
    | N(e,_) when p e -> N(e,[])
    | N(e,es) -> N(e, List.map (h p) es);;

let rec k (N(_, es)) = 1 + List.fold max 0 (List.map k es);;
let p2 = N (1,[N(2,[N(3,[])])]);;
k p1
k p2

let pred e = e = "q";;
h pred p1


//4.1
type Outcome = | S | F // S: for success and F: for failure
type Sample = Outcome list
type ProbTree = | Branch of string * float * ProbTree * ProbTree
                | Leaf of string

let exp = Branch(">2",0.67, Branch(">3",0.5, Leaf "A", Leaf "B")
                          , Branch(">3",0.5, Leaf "C", Leaf "D"))

let expbad = Branch(">2",0.67, Branch(">3",1.1, Leaf "A", Leaf "B")
                          , Branch(">3",0.5, Leaf "C", Leaf "D"))

let rec probOK = function
    | Leaf (lbl) -> true
    | Branch(ds,p,tl,tr) ->
        0.0 <= p && p <= 1.0 &&
        probOK (tl) && probOK (tr);;


probOK exp //should return true
```

```
probOK expbad // should return false

//4.2
//ProbTree -> bool
let rec isSample os t =
    match (os, t) with
    | [], Leaf (lbl) -> true
    | F::rst, Branch(_,_,tl,tr) -> isSample rst tl
    | S::rst, Branch(_,_,tl,tr) -> isSample rst tr
    | _ -> false;

isSample [F;S] exp
isSample [S;F;F] exp
isSample [] exp


//4.3
type Description = (Outcome * string) * float * string;;

let rec makeDescription os t path prob =
    match (os, t) with
    | _, Leaf (lbl) -> ((List.rev path), prob, lbl)
    | F::rst, Branch (ds,p,tl,tr) -> makeDescription rst tr ((F,ds)::path) (prob*(1.0-p))
    | S::rst, Branch (ds,p,tl,tr) -> makeDescription rst tl ((S,ds)::path) (prob*p)
    | _, _ -> failwith "invalid input somehow";;

let descriptionOf os t =
    if isSample os t then makeDescription os t [] 1.0
    else failwith "not a correct sample";;

descriptionOf [F;S] exp

//4.4
let rec findLeaves ptree cpath =
    match ptree with
    | Leaf l -> [cpath]
    | Branch (ds,p,tl,tr)
        -> (findLeaves tl (cpath @ [S]))
            @ (findLeaves tr (cpath @ [F]));;

let allDescriptions ptree =
    let ds = List.map (fun s -> descriptionOf s ptree) (findLeaves ptree List.empty)
    Set.ofList(ds);;

allDescriptions exp;;

//4.5
let allDescriptions2 ptree = List.map (fun s -> descriptionOf s ptree) (findLeaves ptree List.empty);; //returns list insteaf of set
```

```
let pred_CD s = (s = "D") || (s = "C"); //test predicate

let probabilityOf ptree prd =
    List.fold (fun value (ds,p,_) -> value + p) 0.0 (List.filter (fun (_,_,lbl) -
      > prd lbl) (allDescriptions2 ptree));;

//4.6 trivial
probabilityOf exp pred_CD
```