```fsharp
(* Daniel F. Hauge - Studentnumber: s201186 - DTU Course: 02157 Functional programming  *)
open System

(* Problem 1 types *)

type Person = string
type Contacts = Person list
type Register = (Person * Contacts) list


(* Problem 3 types *)

type Name = string
type Part =
    | S of Name                  // Simple part
    | C of Name * Part list   // Composite part

type OccurrenceCount =  Map<Name,int>


[<EntryPoint>]
let main argv =

    (* Problem 1 *)

    let reg1 = [("p1", ["p2"; "p3"]);
                ("p2", ["p1"; "p4"]);
                ("p3", ["p1"; "p4"; "p7"]);
                ("p4", ["p2"; "p3"; "p5"]);
                ("p5", ["p2"; "p4"; "p6"; "p7"]);
                ("p6", ["p5"; "p7"]);
                ("p7", ["p3"; "p5"; "p6"])]

    (* Question 1.1 *)
    let inv1 (reg:Register) : bool =
        let contacts = List.map (fun x -> snd x) reg
        List.forall (fun x -> List.length (List.distinct x) = List.length x ) contacts

    printfn "Question 1.1: %A" (inv1 reg1)

    (* Question 1.2 *)
    let inv2 (reg:Register) : bool =
        let people = List.map (fun x -> fst x) reg
        List.forall (fun x -> not (List.isEmpty (snd x))) reg && List.length (List.distinct people) = List.length people

    printfn "Question 1.2: %A" (inv2 reg1)
```

```


let rec insert p ps = if List.contains p ps then ps else p::ps

let rec combine ps1 ps2 = List.foldBack insert ps1 ps2


(* Question 1.3 *)
let rec immediateContacts (p:Person) (reg:Register) : Contacts =
    match reg with
    | (pr, c)::_ when p = pr -> c
    | _::tail -> immediateContacts p tail
    | _ -> []

printfn "Question 1.3: %A" (immediateContacts "p1" reg1)

(* Question 1.4
    Assuming adding contacts is bi-directional. ie. Adding p1 to p2 as a close contact implies adding p2 to p1 as a close contact aswell.
*)
let rec addContacts (p1:Person) (p2:Person) (reg:Register) : Register =
    match reg with
    | (p, c)::xs when p = p1 -> (p, insert p2 c)::addContacts p1 p2 xs
    | (p, c)::xs when p = p2 -> (p, insert p1 c)::addContacts p1 p2 xs
    | h::tail -> h::addContacts p1 p2 tail
    | _ -> []

printfn "Question 1.4: %A" (addContacts "p1" "p2" reg1)


(* Question 1.5 *)
let contacts (p:Person) (reg:Register) : Contacts =
    let imC = immediateContacts p reg
    let depth2contacts = List.map (fun x-> snd x) (List.filter (fun x-> List.contains (fst x) imC) reg)
    combine imC (List.reduce (fun a b -> combine a b) depth2contacts)

printfn "Question 1.5: %A \n" (contacts "p1" reg1)




(* Problem 2 *)

(* Question 2.1
    j is used with an addition operation, hence j is infered to be an itneger.
    xs can be infered to be some sort of list, by the pattern matching. (Can easily be seen by the [] -> [] case)
```

```
        output is also infered to be some sort of list by the pattern matching. (Can easily be seen by the [] -> [] case)
        f is a function which is used with j and an element of xs.
        there is no information for what kind of type the elements within xs has to be, also there is no information about what type f has as output,
        hence the generic types 'a for elements within xs and 'b for elements after f has been aplied.

        Therefor:
        j : int
        f : (int -> ’a -> ’b)
        xs : 'a list
        output : 'b list

        h : (int -> ’a -> ’b) -> ’a list -> int -> ’b list

    *)

    (* Question 2.2

        given
        f : (fun i x -> (i,x))
        xs : ["a";"b";"c"]
        (and 0 as j in h, as declared in mapi)

        => mapi (fun i x -> (i,x)) ["a";"b";"c"]   ( => )  h (fun i x -> (i,x)) ["a";"b";"c"] 0
        => (0,"a")::h (fun i x -> (i,x)) ["b";"c"] 1
        => (0,"a")::(1,"b")::h (fun i x -> (i,x)) ["c"] 2
        => (0,"a")::(1,"b")::(2,"c")::h (fun i x -> (i,x)) [] 3
        => (0,"a")::(1,"b")::(2,"c")::[]    (Concatinating elements from here.)
        => [(0,"a");(1,"b");(2,"c")]

    *)

    (* Question 2.3

        (fun i x -> (i,x)) : 'a -> 'b -> ('a * 'b)
        The expression is a function which takes 2 inputs respectively i and x, putting them together in a tuple. Types for i and x can be whatever, there is no operations which forces behavior or type restrictions.

        ["a";"b";"c"] : string list
        Double qoutes is in many programming languages including F# the way to indicate literal string values.

        'a in h is forced to be an integer because of the previously mentioned addition operation, as a result forces 'a or rather i in f to be an
```

```
         integer aswell.
        Using a list of strings will give string in 'b or rather x in f. Therefor the result of f will give a tuple of integer and string, 'b in h.

    *)


    (* Question 2.4 *)

    let rec h f xs j = match xs with
    | []      -> []
    | x::rest -> f j x :: h f rest (j+1)

    let rec h_tail_rec f xs j acc =
        match xs with
        | []      -> acc
        | x::rest -> h_tail_rec f rest (j+1) (acc@[(f j x)])

    let mapi f xs = h f xs 0
    let mapi_h_trec f xs = h_tail_rec f xs 0

    printfn "Question 2.4: %A - %A \n" (mapi_h_trec (fun i x -> (i,x)) ["a";"b";"c"] []) (mapi (fun i x -> (i,x)) ["a";"b";"c"])

    (* Problem 3 *)

    (* Question 3.1 -> Assuming that all composite sub-parts of p also has to comply, and all subsequent composite sub-parts also comply. *)
    let rec inv (p:Part) : bool =
        match p with
        | S(_) -> true
        | C(_, []) -> false
        | C(_,s) -> List.forall inv s


    (* Question 3.2 *)
    let rec depth (p:Part) : int =
        match p with
        | S(_) -> 0
        | C(_,s) -> 1 + List.max (List.map (fun x -> depth x) s)

    (* Question 3.3

        "Name" is a string type. (Declared as string type)

        "Part" is the name of the type being declared, which for F# is given after the keyword "type".

```

```fsharp
        "S" is one of valid union cases of Part. Part is a discriminated union type, a type which may contain a set of different kind of types, ie. The full set of all possible values within each union case.
        S consists of Name which is a string type, hence S is the union case where Part is just a simple part.

        "*" indicating a relation, said simply: making a record or tuple.

        "list" is a keyword for indicating a list type. ex. string list.

        "C" is the other valid union case of Part. The valid type for C is a tuple consisting of Name in the first value and a list of Parts in the second value.
        Part is recursive in nature as the union case C contain the same type in the form of a Part list.

    *)


    (* Question 3.4 Assuming conditions are atleast 5 different simple parts and atleast 4 different composite parts but can have more.*)
    let specialPart = C("C1",
                        [C("C2",
                            [C("C3", [S("S1");S("S2");S("S5");S("S3");S("S7");S("S7");S("S7");S("S7");])]);S("S9");S("S7");
                         C("C4", [S("S7");S("S4");S("S5");]);S("S2");S("S2");S("S1");S("S15");
                        S("S5");S("S5");S("S1");S("S15");S("S15");])


    (* Question 3.5 Assuming all names recursively in p (exluding names of composite parts) *)
    let simpleNames (p:Part) : Set<Name> =
        let rec simpleNamesRec (p:Part) (acc:Set<Name>) : Set<Name> =
            match p with
            | S(n) -> Set.add n acc
            | C(_,s) -> (Set.unionMany (List.map (fun x-> simpleNamesRec x acc) s))
        simpleNamesRec p Set.empty


    (* Question 3.6 *)
    let computeOccurences (p:Part) : OccurrenceCount =

        let addOccurences (i:int) (n:Name) (o:OccurrenceCount) : OccurrenceCount =
            match Map.tryFind n o with
            | None -> Map.add n i o
            | Some a -> Map.add n (i+a) (Map.remove n o)
```

```
    let mergeOccurences (o1:OccurenceCount) (o2:OccurenceCount) : OccurenceCount = Map.fold (fun s k v -> addOccurences v k s) o1 o2

    let rec computeOccurenceRec (p:Part) (acc:OccurenceCount) : OccurenceCount =
        match p with
        | S(n) -> addOccurences 1 n acc
        | C(n,s) -> mergeOccurences (addOccurences 1 n acc) (List.reduce (fun a b -> mergeOccurences a b) (List.map (fun x -> computeOccurenceRec x acc) s))

    computeOccurenceRec p Map.empty

printfn "Question 3.1: %A" (inv specialPart)
printfn "Question 3.2 & 3.4: depth = %A" (depth specialPart)
printfn "Question 3.5: %A" (simpleNames specialPart)
printfn "Question 3.6: %A\n" (computeOccurences specialPart)



(* Problem 4 *)

let rec gC i k =
    if i=0 then k 0
    else if i=1 then k 1
    else gC (i-1) (fun v1 -> gC (i-2) (fun v2 -> k(v1+v2)))


(* Question 4.1 *)
let rec g i = if i = 0 then 0 else if i = 1 then 1 else (g (i-1))+(g (i-2))
printfn "Question 4.1: %A = %A" (g 15) (gC 15 id)


(* Question 4.2 *)
let seq1 = seq { for i in Seq.initInfinite id do if (i%2 = 0) then yield (2*i+1) else yield -(2*i+1) }
printfn "Question 4.2: %A" seq1


(* Question 4.3 *)
let seq1float = seq { for i in seq1 do yield float i}
let seq2 = seq { for i in seq1float do yield (1.0/i)}
printfn "Question 4.3: %A" seq2


(* Question 4.4 *)
```

```
    let seq3 = seq { for i in Seq.initInfinite (fun x->x+1) do yield Seq.sum (Seq.take i seq2) }
    printfn "Question 4.4: %A" seq3

    0
```