Genealogical repositories

Genealogical repositories are data structures that register relationships between several generations of individuals.

Those repositories can be used to register historical information about human families and communities, but also as a genetic tool for the development of breeds of animals and plants.

Here, we restrict ourselves to biological sexual reproduction, ignoring other aspects like, for example, adoption.

Module "Genealogy"

The aim of this project is to write a closed OCaml module named "Genealogy" containing a data representations for genealogical information.

The module interface has already been fully written and you are not allowed to change it: Genealogy.mli. As you can see, the data representation is public and there is also a small number of public functions declared. All the other entities you might define in the module body will become private. The representation is public to allow Mooshak to check the module deeply.

Use this file as a starting point to write your module body: Genealogy.ml.

Repository representation

For the representation of genealogical repositories, we use the following OCaml types:

```
type item = string * string list

type repository = item list
```

Here is an example:

```
let example = [
          ("a", ["f";"g"]);
          ("b", ["f";"h"]);
          ("c", ["h";"i"]);
              ("f", ["g"; "j"]);
              ("g", ["j"]);
              ("h", []);
```

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

A repository is simply a list of pairs, each pair consisting on the identification of an individual, plus the list of the known children of that individual. Note that in the case of a stable couple, the list of their children will appear twice, in the two pairs of the parents.

The data in the repository is, normally, incomplete. We cannot know all the individuals in the universe and also we might not be aware of every children of the known individuals. Furthermore, we do not register whether each individual is male or female. We don't need that information in this project and, anyway, there exists natural sex change in some species like the clown fish, moray eels, chicken (rarely), etc. [meaning that, in the biologic world, it is possible for the two parents to be both female or both male (after the sex change of one of them).]

Not all repositories can represent a real situation. To consider a repository valid, we require two kinds of restrictions:

- Structural restrictions - (1) There are no duplicates among the first components of the various pairs. (2) All the known individuals occur in the first component of the pairs, even if its list of children needs to be empty. (Both restrictions make the list a dictionary, with the first component representing the index.)
- Semantic restrictions - (1) There are no loops in the structure, meaning that nobody can be ancestor of itself. (2) Each individual can have two parents, at most.

As for the restrictions, note that we will not rule out other situations, no matter how strange they are. It is important to be able to represent those situations, even if to prevent them from happening in the future. For example, dealing with strange situations may be important for the successfully development of a breed of dogs. Our example includes some strange situations of inbreeding, with "f" mating with its parent "a", generating a child "g" and, after that, "f" mating with this new "g" to generate another child "j". Of course, in this example, "f" is at the same time sibling and parent of "g", and "f" is at the same time parent and grandparent of "j".

Other representations

In this project, we will also use some other OCaml types, although in a very limited way. These types are useful when we gather data from the point of view of a single individual.

An ancestors tree is a binary tree. When is not empty, the tree is represented by a triple consisting in: the individual at the root, plus the ancestors trees of both parents.

```
type aTree = ANil | ANode of string * aTree * aTree
```

A descendants tree is an n-ary tree. When is not empty, the tree is represented by a tuple consisting in: the individual at the root, plus the list of the descendant trees of each of its child.

```
type dTree = DNil | DNode of string * dTree list
```

Processing a repository

A genealogical repository is a directed acyclic graph, with individuals at the vertices and with oriented arcs from each individual to its children.

The repository representation offers greater flexibility than a tree-like structure, simplifying many operations. We can skillfully use functions of the module List, and some more auxiliary functions, to solve many problems. Technically, all these will be solutions of category 4 (cf. lesson 08). For example:

```
let size rep = (* number of individuals *)
    len rep

let all1 rep = (* all the individuals *)
    map fst rep

let all2 rep = (* all the children (of anyone) *)
    cFlatMap snd rep

let roots rep = (* individuals without any parents *)
    diff (all1 rep) (all2 rep)

let inners rep = (* individuals with children *)
    let xs = filter (fun (p,cs) -> cs <> []) rep in
        all1 xs

let leaves rep = (* individuals without any children *)
    let xs = filter (fun (p,cs) -> cs = []) rep in
        all1 xs
```

Other problems require dealing with the repository in a more structured way. The following functions offer ways to partition a repository in two parts, enabling the use of the inductive method:

```
let cut1 rep l = (* partition based on first component of the
repository *)
    partition (fun (p,cs) -> mem p l) rep

let cut2 rep l = (* partition based on second component of the
repository *)
    partition (fun (p,cs) -> inter cs l <> []) rep

let cut rep = (* partition -> (root pairs, rest pairs) *)
    cut1 rep (roots rep)
```

The function **cut** is the most important of these. It partitions the pairs of a repository in two groups:

1. the roots of the repository, that is all the pairs representing individuals with no parents - let's call this **repository head**;
2. the remaining pairs - let's call this **repository tail**.

To count the elements in the repository we don't need this technique. However, to exemplify, and also to compare with the previous implementation, here is the function **size** reprogrammed using an inductive function of category 1:

```
let rec size rep =
    if rep = [] then 0
    else
        let (x,xs) = cut rep in
            len x + size xs
```

The following two functions can also be used with some advantage to solve certain problems:

```
let children rep l = (* get all the children of the list l *)
    let (a,b) = cut1 rep l in
        all2 a

let rec parents rep l = (* get all the parents of the list l *)
    let (a,b) = cut2 rep l in
        all1 a
```

Example of use:

```
let grandparents rep l = (* get all the grand parents of the
list l *)
    let ps = parents rep l in
        parents rep ps
```

The public functions of the module

There are thirteen public functions to implement. Please, write the functions in good functional style, avoiding any kind of code based on imperative reasoning. Anyway, note that within the functional style, there are several possibilities you might want to consider and select, depending on the problem at hand. The possibilities include: the direct use of the inductive method; decomposition into simpler functions; taking advantage of the function of the module List; a mixture of all these.

In the descriptions below, the preconditions of each function are clearly stated. The task of each function is to calculate the result according to each specification and also to ensure the stated postconditions over the result.

```
height : repository -> int
(* pre: validStructural rep && validSemantic rep *)
(* post: none *)
let height rep = ...
```

Height of the repository. Can be defined as the maximum distance between a root and a leaf. The height of the empty repository is zero.

```
makeATree : repository -> string -> aTree
(* pre: validStructural rep && validSemantic rep *)
(* post: none *)
let makeATree rep a = ...
```

Using the repository, make the more complete possible aTree for a particular individual. Such individual may not occur in the repository, in which case the result will be a tree with only the root.

```
repOfATree : aTree -> repository
(* pre: saneATree t *)
(* post: validStructural result *)
let repOfATree t = ...
```

Convert all the data in an aTree to the form of a repository.

```
makeDTree : repository -> string -> dTree
(* pre: validStructural rep && validSemantic rep *)
(* post: none *)
let makeDTree rep a = ...
```

Using the repository, make the more complete possible dTree for a particular individual. Such individual may not occur in the repository, in which case the result will be a tree with only the root.

```
repOfDTree : dTree -> repository
(* pre: saneDTree t *)
(* post: validStructural result *)
let repOfDTree t = ...
```

Convert all the data in an dTree to the form of a repository.

```
descendantsN : repository -> int -> string list -> string list
(* pre: validStructural rep && validSemantic rep && n >= 0 &&
noDuplicates lst *)
(* post: noDuplicates result *)
let descendantsN rep n lst = ...
```

Given a list of individuals, return all their descendants, **n** levels below. If **n=0** then the original list of individuals is returned. If **n=1** then the list of all the children of the individuals is returned. If **n=2** then the list of all the grandchildren of the individuals is returned. Etc. According to

this definition, it is possible for an individual to appear in different results, for different values of **n**.

```
siblings : repository -> string list -> string list
(* pre: validStructural rep && validSemantic rep && noDuplicates lst
*)
(* post: noDuplicates result *)
let siblings rep lst = ...
```

Given a list of individuals, return all their siblings (brothers and sisters). Two siblings are two individuals that share at least one parent. For the purpose of this function, please consider that any individual is sibling of itself.

```
siblingsInbreeding : repository -> (string * string) list
(* pre: validStructural rep && validSemantic rep *)
(* post: noDuplicates result *)
let siblingsInbreeding rep = ...
```

Find all the pairs of distinct siblings that have at least one child in common. Try not to blindly generate and check all pairs of individuals. It is possible to consider only pairs of siblings, from the outset. The returned pairs are seen as unordered pairs - that is, the order of the two elements does not matter.

```
waveN : repository -> int -> string list -> string list
(* pre: validStructural rep && validSemantic rep && n >= 0 &&
noDuplicates lst *)
(* post: noDuplicates result *)
let waveN rep n lst = ...
```

Collect all the individuals that are at a distance **n** from at least one element of **lst** and cannot be interpreted as being at a lower distance from at least one element of **lst**. The distance is measured as the number of transversed arcs. If **n=0** then the original list of individuals is returned. If **n=1** then the result is the list of all children and all parents of **lst** (minus the elements whose distance have already been established as zero). For **n=2**, we need to find the children and parents of the previous wave (minus the elements whose distance have already been established as zero or one). Etc. The result must not have repetitions. According to this definition, it is not possible for an individual to appear in the result, for different values of **n**.

```
merge : repository -> repository -> repository
(* pre: validStructural rep1 && validSemantic rep1 && validStructural
rep2 && validSemantic rep2 *)
(* post: validStructural result *)
let merge rep1 rep2 = ...
```

Creates a single repository that joins all the information contained in the two given repositories.

```
supremum : repository -> string list -> string list
(* pre: validStructural rep && validSemantic rep && noDuplicates lst
*)
(* post: noDuplicates result *)
let supremum rep l = ...
```

Given a list of individuals, return the list of their **commom** ancestors that are at a level as low as possible of the repository (that is as far as possible from the roots).

```
validStructural : repository -> bool
(* pre: none *)
(* post: none *)
let validStructural rep = ...
```

Check the two structural restrictions on the repository.

```
validSemantic : repository -> bool
(* pre: validStructural rep *)
(* post: none *)
let validSemantic rep = ...
```

Check the two semantic restrictions on the repository.

Examples

This section includes some examples to help clarifying what the functions should do. Note that ordering of the the results is not important.

The design of good tests is an essential part of any programming task. Try to be creative by inventing tests that make your program go wrong.

```
# height example;;
- : int = 4

# makeATree example "g";;
- : aTree =
ANode ("g", ANode ("a", ANil, ANil),
 ANode ("f", ANode ("a", ANil, ANil), ANode ("b", ANil, ANil)))

# repOfATree (ANode("g", ANode("a", ANil, ANil), ANode("f",
ANode("a", ANil, ANil), ANode("b", ANil, ANil))));;
- : (string * string list) list =
[("a", ["f"; "g"]); ("b", ["f"]); ("f", ["g"]); ("g", [])]

# makeDTree example "a";;
- : dTree =
DNode ("a",
 [DNode ("f", [DNode ("g", [DNode ("j", [])]); DNode ("j",
[])]);
  DNode ("g", [DNode ("j", [])])])

# repOfDTree (DNode("a", [DNode("f", [DNode("g", [DNode("j",
[])]); DNode("j", [])]);  DNode("g", [DNode("j", [])])]));;
- : (string * string list) list =
[("a", ["f"; "g"]); ("f", ["g"; "j"]); ("g", ["j"]); ("j", [])]

# descendantsN example 1 ["a"];;
- : string list = ["f"; "g"]

# siblings example ["g"];;
- : string list = ["f"; "g"; "j"]

# siblingsInbreeding example;;
- : (string * string) list = [("f", "g")]

# waveN example 1 ["a"];;
- : string list = ["f"; "g"]
```

```
# merge [("b", ["k"]); ("k", [])] example;;
- : (string * string list) list =
[("b", ["f"; "h"; "k"]); ("k", []); ("a", ["f"; "g"]); ("c",
["h"; "i"]);
 ("f", ["g"; "j"]); ("g", ["j"]); ("h", []); ("i", []); ("j",
[])]

# supremum example ["h";"j"];;
- : string list = ["b"]

# validStructural example;;
- : bool = true

# validSemantic example;;
- : bool = true
```

Evaluation and grades

You will submit the file "Genealogy.ml" via Mooshak.

Around 80% of the grade of your group is automatically assigned by Mooshak. The remaining 20% is assigned manually by the teachers, who will analyze the quality of your code.

A special case: In case of code of extremely bad quality, or code that uses the forbidden imperative mechanisms of OCaml, or code that constantly simulates imperative mechanisms and concepts, a special rule will be used so that the grade will be always below 50%, even if the program works well.

To develop the project, we strongly recommend you use the OCaml interpreter, probably inside the Eclipse IDE.

However, you should know that Mooshak will compile your module. Mooshak will use the following command in Ubuntu 16.04:

```
ocamlc -c Genealogy.mli Genealogy.ml
```

After the compilation, Mooshak will test the module in the interpreter like this:

```
$ ocaml
    Objective Caml version 4.02.3
# #load "Genealogy.cmo";;
# open Genealogy;;
...
...
```

Please, make a backup of the file "Genealogy.ml" once a while, because the file can disappear as result of human error or as result of a software/hardware malfunction.