1. This question deals with frogs and amoebae. The following data type describes a sequence of actions of a frog:

```
data FrogActions = Croak FrogActions | Jump FrogActions | Die
```

(a) A frog got stuck at the bottom of a (very) deep well. By jumping up, and clinging to the wall, it manages to move up by 3cm. When croaking, it slides back down 1cm (unless it it already at the bottom). Define a function `distance`, which given `FrogActions`, returns the distance the frog has managed to climb up before dying. You may assume that the frog will never reach the surface. For example, `distance (Jump (Croak (Croak (Jump Die))))` should return 4, while `distance (Croak (Jump (Croak (Jump Die))))` should return 5.

(b) Define a function `initialCroaks` which, given `FrogActions`, returns the number of times the frog repeatedly croaks at the start of the sequence. For example, `initialCroaks (Croak (Croak (Jump (Croak (Jump Die))))))` should return 2, while `distance (Jump (Croak (Croak (Jump Die))))` should return 0.

(c) Make `FrogActions` an instance of the `Show` class where the `show` function should translate a sequence of frog actions into a textual description, grouping together sequential copies of croaks and jumps. For example, `show (Jump (Croak (Croak (Jump Die))))` should return "1 x Jump, 2 x Croak, 1 x Jump, Die".
(d) An amoeba can do actions similar to a frog, except that it can also spawn and split into two amoebae:

```haskell
data AmoebaActions = Creak AmoebaActions | Climb AmoebaActions |
                     Spawn AmoebaActions AmoebaActions | EatenByFrog
```

While climbing up the (same) well, the amoeba can spawn (split into two) any number of times. In the following example, the amoeba spawns twice:

```haskell
Creak (Spawn (Climb (Creak EatenByFrog))
  (Creak (Spawn (Climb (Climb EatenByFrog))
    (Creak EatenByFrog)
  )
)
)
```

Define a function `splits` which, given `AmoebaActions`, counts how many times the amoeba splits.

2. This question deals with roads and EU grants. Consider the following definition:

```haskell
data Road a = OneWay (a,a) |
             TwoWay (a,a)

type Map a = [Road a]
```

An object of type `Road a` (where `a` is any type which is used to identify locations) always contains a pair of objects of type `a`, with the constructor stating whether the direct road between the two is a one-way road (from the first to the second) or a two-way road. A map is simply a collection of roads which one may take. For example, a small map can be:
[ TwoWay ("A","B"), OneWay ("A","C"), OneWay ("C","B")
, TwoWay ("C","D"), OneWay ("A","X"), TwoWay ("X","Z") ]

(a) Define a function, which given a sequence of locations and a map, checks whether the sequence of locations can be legally followed using the roads on the given map. For example, ["A", "C", "B", "A"] is a legal route through the map given earlier (since one can go directly from "A" to "C", from "C" to "B" and from "B" to "A" using the given map. On the other hand, ["A", "C", "B", "C"] would return false, since it is impossible to go directly from "B" to "C".

(b) Define a function smallEUGrant, which given a map returns a new one which is identical except that the first one-way road is upgraded to a two-way one. For example, applying smallEUGrant to the map given earlier, one would get:

[ TwoWay ("A","B"), TwoWay ("A","C"), OneWay ("C","B")
, TwoWay ("C","D"), OneWay ("A","X"), TwoWay ("X","Z") ]

Give the type of smallEUGrant.

(c) Hence or otherwise define biggerEUGrant, which upgrades two one-way roads.

(d) Consider the following higher-order function, already given in the notes:

apply f 0 x = x
apply f n x = f (apply f (n-1) x)

Give the type of apply, and use it to define upgrade, which is given a number n and a map m, and upgrades the first n one-way roads in m to two-way ones.

(e) Give the type of the following functions:

example1 "" = []
example1 (c:c':cs) = OneWay (c, c'): example1 (c':cs)

example2 = example1 . show