National Technical University of Athens
School of Electrical & Computer Engineering
Department of Computer Science
https://courses.softlab.ntua.gr/pl2/

Programming Languages II

Unless otherwise stated, exercises should be submitted in electronic form, via the collaborative learning environment `moodle.softlab.ntua.gr`. Deadlines will be strict. You are allowed at most one late exercise.

Exercise 4 Tree traversal and verification

Deadline: 2/2/2020

In Haskell, let `Tree` be the type of a tree with an arbitrary number of children per node, the same that we saw in the examples on DFS/BFS traversals, in the lecture of 30/10/2019.

```
data Tree a = Node a [Tree a]
```

Also, from the same lecture, the following functions traverse a tree and enumerate the nodes in DFS and BFS order, respectively. Their definitions can be found in the course's web page, but it will be more fun to try to write them again from scratch.

```
dfn :: Tree a → Tree (a, Int)
bfn :: Tree a → Tree (a, Int)
```

1. Implement a generator of arbitrary trees that enables automatic property-based testing for trees using `QuickCheck`. In particular, implement an `instance` of the form:

   ```
   instance Arbitrary a ⇒ Arbitrary (Tree a)
   ```

   Because `Tree` is a recursive type, make sure to use the notion of size in the generator (`sized`, `resize`), so that the generator terminates. Also, make sure that you generate trees that are not always too "wide" nor top "narrow".

2. Implement appropriately the function `shrink` in `instance Arbitrary (Tree a)`, so that "small" can be found as counterexamples of properties that do not hold.

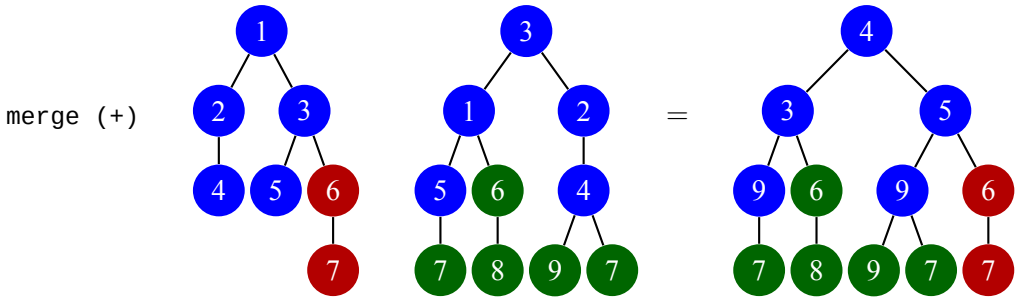
3. Define a set of properties that guarantee the correct functionality of functions `dfn` and `bfn`. Write these properties as Haskell functions, which can be verified using QuickCheck.

   Examples of such properties (although you may be able to find better ones):

   - Both functions preserve the tree size (i.e., the number of nodes).
   - For both functions, the tree's root should be numbered 1.

4. Implement a Haskell program that uses Quickcheck on arbitrary trees to verify the above properties for functions `dfn` and `bfn`.

5. You are aked to implement a function `merge` that "merges" two trees by combining the values of corresponding nodes using a function `f`. More precisely, this function must take as parameter the function `f` and two trees, `t1` and `t2`. If the two trees have a node *in the same position*, with values `x1` and `x2` respectively, then the result must also have a node in this position with value `f x1 x2`. On the other hand, if only one of the two trees has a node in some position, the result must have a node in this position with the same value. For example:



where, for your convenience, the nodes that are present in the same position in both trees are marked in blue colour, the nodes that are present only in `t1` are marked in red colour, and the nodes that are present only in `t2` are marked in green colour.

6. A friend of yours gives you the following function `wrong`, which, according to his claim, achieves the goal and is better than yours!

```
wrong :: (a → a → a) → Tree a → Tree a → Tree a
wrong f (Node x tsx) (Node y tsy) = Node (f x y) $ zipWith (wrong f) tsx tsy
```

Implement a Haskell program that, using QuickCheck on arbitrary trees, proves the following:

- that `wrong` is wrong — to do this, define some property that a correct implementation of `merge` must verify,
- that your implementation of `shrink` generates indeed small trees as counterexamples for false properties, and
- that your implementation of `merge`, that you defined in the previous question, verifies the property that you used to disprove your friend.