Functional Programming 1 — 1DL330
Assignment 4

Lab: Friday, 4 October

Submission Deadline: 18:00, Wednesday, 16 October, 2019

This is an **individual** assignment. You must construct your own solution. Keep any discussions with other students at the level of abstract solution methods; do *not* share your code. If you think there is a risk that such a discussion leads to very similar solutions, then report this to the instructor in advance, e.g., by clearly stating any collaboration in your submission. If you have any questions about this assignment, please contact the lab assistant.

Instructions

- Start by reading the General Lab Instructions (on the Student Portal).
- Download the files `Graph.hs` and `Lab4.hs` from the Student Portal. Modify these files as needed to add your solutions. Do *not* rename the files.
- The files that you submit must be valid Haskell files, so the answers to some of the questions need to be given as comments using $--\ldots$ or $\{-\ \ldots\ -\}$.
- Remember to follow our Coding Convention (also on the Student Portal), and provide a specification for every function that you write. Also provide a variant for every recursive function.
- Make sure that your solutions pass the tests in `Lab4Tests.hs` before you submit. A submission that does not pass these tests will get grade K. See the *Testing* section near the end of this assignment for details.

1 A Module for Graphs

Graphs model the connections between objects in a network. They have applications in computer science and many other areas. For instance, the topology of the Internet, the interactions between proteins in molecular biology, and the interactions between users on a social networking site can all be modeled by graphs.

Formally, a *graph* $G = (V, E)$ consists of a set V of *vertices* and a set $E \subseteq V \times V$ of *edges* between these vertices. We will only consider simple graphs in this assignment. A simple graph is an undirected graph without multiple edges or loops. In other words, for all vertices u and v, an edge between u and v is the same as an edge between v and u,

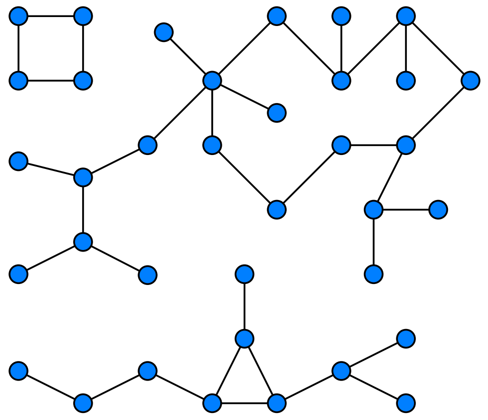


Figure 1: A simple graph with three connected components.

there is at most one edge between u and v, and there is no edge between u and u. See Figure 1 for an example of a simple graph.

Define a module Graph for simple graphs. This module shall export a polymorphic type Graph a of simple graphs with vertices of type a, and the following operations on them. There shall be a way to

1. obtain an empty[1] graph:
   `empty :: Graph a`

2. add a vertex to a graph:
   `addVertex :: Eq a => Graph a -> a -> Graph a`

3. add an edge to a graph:
   `addEdge :: Eq a => Graph a -> (a,a) -> Graph a`

4. obtain a list of all vertices in a graph:
   `vertices :: Eq a => Graph a -> [a]`

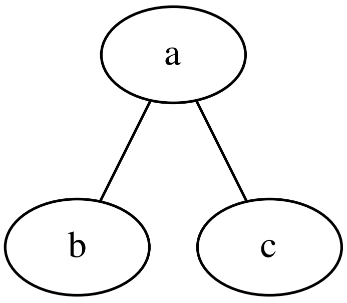
5. obtain a list of all neighbors[2] of a given vertex:
   `neighbors :: Eq a => Graph a -> a -> [a]`

You will need to decide what the exact preconditions for some of these functions should be. The lists returned by vertices and neighbors shall not contain any duplicates.

Your module shall not export any other operations. The type Graph a shall be implemented by a data type. The constructor(s) of this data type shall *not* be exported. (This means that the implementation of your graph type will not be visible outside of the Graph module. Your graphs will appear as an abstract data type that can only be accessed through the operations provided.)

You may use any suitable data structure that has been presented in class, or invent one yourself to implement graphs. Data.Graph might provide some inspiration.

[1]The empty graph does not contain any vertices or edges.
[2]Vertex u is a *neighbor* of vertex v if there is an edge between u and v.



```
graph {
  a;
  b;
  c;
  a -- b;
  a -- c;
}
```

Figure 2: A simple graph (left) and its DOT representation (right).

2 Complexity

For all operations provided by your Graph module from Exercise 1, state their (worst-case) time complexity, expressed as a function of the number of vertices or edges (or both) that are contained in the graph.

3 DOT Representation

In `Lab4.hs`, import your Graph module from Exercise 1, and define a function

```
dot :: Graph String -> String
```

that is specified as follows. Given a graph whose vertices are strings, `dot` shall compute a string representation of the graph in the DOT graph description language. See Figure 2 for an example. You may assume that all vertex names are valid DOT identifiers (i.e., there is no need to quote or escape strings). See here for further examples and explanations regarding DOT.

Hint: A graph may have multiple valid DOT representations; it is *not* necessary to generate the shortest/simplest representation. You can test your `dot` function—and inspect your graphs—by using its output as input for the `dot` or `neato` command line tools.

4 Connected Components

A *connected component* of an undirected graph is a non-empty subgraph[3] in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the original graph. For instance, the simple graph in Figure 1 consists of three connected components.

In `Lab4.hs`, define a function

```
connectedComponent :: Eq a => Graph a -> a -> Graph a
```

[3]The vertices and edges of a *subgraph* are a subset of the vertices and edges, respectively, of the original graph.

such that connectedComponent g v returns the connected component of g that contains vertex v.

Of course, your connectedComponent function should terminate for any input graph. Remember to provide a variant (or several variants, one for each recursive function used in your implementation of connectedComponent) that shows termination.

Hint: Given a vertex v in a simple graph, the connected component that contains v can be found by a (depth-first or breadth-first) search that starts at v and explores all vertices of the graph that can be reached via edges. See here for further references. Use additional helper functions and data structures as necessary.

Testing

Use the test cases provided in `Lab4Tests.hs` (from the Student Portal) to test your solution. See the instructions for Assignment 1 for further information on running the tests.

Some tests might fail because they assume other datatype definitions or function signatures than what you are using. If necessary, *change your solutions* to make all tests compile and run successfully.

Good luck!