Coursework 1: Haskell Adventure **Due: 1 December 2023, 8pm**

Deadline	Friday 1 December 2023, 8pm
Submission	Moodle
Submission file	`Coursework_1.hs`
Weighting	50%
Anonymous marking	Yes

Plagiarism warning This coursework is an **individual assignment**. Collaboration on this assignment is not permitted: you should not discuss your work with others, and the work you submit should be your own. Disclosing your solutions to others, and using other people's solutions in your work, both constitute plagiarism: see `http://www.bath.ac.uk/quality/documents/QA53.pdf`.

Introduction

In this coursework we will build an adventure game. The assignment consists of two parts. First, you will build the game according to a given specification. Second, you are given the more open-ended task of finding a solution to the game.

The game will work as follows. You will travel between different locations, where you will meet different characters (NPCs). You will be able to talk to them, and to more characters at the same time, to make them interact. Some will join your party, so you can take them to other locations, where they can talk to other characters.

The coursework is broken up into five assignments. These are incrementally more challenging, with less guidance provided. Nevertheless, each part may contain easier and harder tasks. Make sure to at least inspect each assignment, so you are not inadvertently skipping parts you could have completed.

When managing your time, if you have to choose between doing a few things very well, or doing many things haphazardly, it is generally better to choose the former. This is reflected in marks for code quality in the assessment scheme, which are awarded for overall quality. Be careful: adding a low-quality part to an overall high-quality solution will reduce the quality marks, which may result in an overall lower mark.

Submission

Submit the file `Coursework_1.hs` at the submission point on Moodle:

`https://moodle.bath.ac.uk/mod/assign/view.php?id=1272200`

Submit only this file, and do not zip it. Marking is anonymous, so do not add your name or student number to the submission. (Moodle knows who you are, but it will only tell us after marking is completed.)

Assessment

Your assignment will be marked both for **functionality** (60%) and **quality** (40%).

- Functionality is assessed by a semi-automated process. A number of tests is run, and for recognised output, marks and feedback are produced automatically. Unrecognised output is marked manually, and added to the database of recognised output.
- Code quality is assessed manually by the lecturers and senior tutors, by running and inspecting your code.

There are 100 marks available, allocated as follows:

Functionality	1. The game world	15 marks
	2. Dialogues	15 marks
	3. The game loop	10 marks
	4. Safety features	5 marks
	5. Solving the game	15 marks
Quality	Presentation	10 marks
	Technique	20 marks
	Abstraction	10 marks
Penalties	Syntax & type errors	-10 marks
	Incorrect submission	-5 marks

The requirements for code quality are as follows.

- **Presentation:** your code should be well laid out, with attention to whitespace, indentation, alignment, ordering of functions, commenting where necessary, and function and variable naming. Your style should be consistent.
- **Technique:** your code should use appropriate techniques for each situation, such as recursion, tail recursion, higher-order functions, and list comprehension. Your functions should have a single, coherent purpose, and not be unnecessarily complex. Functions should be defined locally instead of globally where appropriate. There should not be spurious imports or superfluous constructions (e.g. `b == True` instead of `b`).
- **Abstraction:** your code should put commonly used functionality in a single function, to avoid repetition and aid readability.

Penalties: Your should not have **syntax** and **type** errors. We may correct minor errors for a penalty of up to -10 marks, and then mark as normal. Major errors will result in a functionality

mark of zero. (Note: **runtime** errors are part of functionality marking and not penalized.) We will also deduct marks for incorrect submission formats.

Allowed imports: The library `Data.Char` will be useful in Assignment 4. You should not need any further libraries, but you may use `Data.List`, `Data.Maybe`, `Data.Set`, `Data.Map`, `System.Random`, and `Control.Monad`. (If you would like other libraries, please ask us.) Only import libraries if you make essential use of them—spurious imports are penalised, and simple solutions will get higher quality marks than too-complex ones.

The requirements for functionality are set out in the remainder of this document.

The game world

Your game will consist of several components. There are **dynamic** data, which change during the game, and **static** data, which remain the same. The dynamic data are stored in the **game state**. The **types** for game data are at the start of your file; the game data themselves are at the end of your file. We will address this in two parts: first, the game world; second, the dialogues between characters. The game world has the following data.

- The **characters** are represented by strings. A **party** is a non-repeating ordered list of characters. You are given the following type aliases, and the functions `merge` and `msort` are included to manage parties.

```
type Character = String
type Party     = [Character]
```

- The **locations** in the game are represented as **integers** for internal use, called **nodes**, and as **strings** for display. You are given the following type aliases.

```
type Node      = Int
type Location  = String
```

  At the end of your file, the names the locations are given by the list `theLocations`; the name of a node `n` is then `theLocations !! n`.

- The **map** of the game is represented by an ordered list of the connections between nodes. The direction is not important: two nodes are connected or not, and when they are, you can travel between them. Nodes are never connected to themselves, and never connected to the same node twice. You are given the following type alias:

```
type Map       = [(Node,Node)]
```

- The **game state** contains four pieces of data:
  - the current **map**,

  - your current **location**,
  - your current **party**,
  - the current **party** at each **location**

  It is given as the following data type, which has two cases: the case `Over` when game has concluded (and no data is needed), or the case `Game` when the game is in progress. The game state holds all the dynamic data.

```
data Game = Over | Game Map Node Party [Party]
```

  In a game `Game m n p ps` the list `ps :: [Party]` holds the characters at each location; for a location (a node) `i`, the characters are `ps !! i`. The start state is `start :: Game`, given with the other game data, where the characters at each location are given by `theCharacters`.

- An **event** is something that changes the game state. You are given the type alias:

```
type Event      = Game -> Game
```

You are asked to implement the following auxiliary functions that manage the game world. You should maintain the characteristics of `Party` and `Map`: parties are ordered without duplicates, and maps store whether two nodes are connected or not, with no duplicate connections or self-connections. Adding a character or connection that is already present, or removing one that is not, should do nothing. The functions `merge` and `msort` (from Tutorial 2) may be helpful, and you should introduce auxiliary functions where appropriate.

Assignment 1: **(15 marks)**

a) `connected`: given a map `m` and a node `n`, return those nodes connected to `n` on the map `m` (note that a connection goes in both directions).

b) `connect`, `disconnect`: given two nodes `i` and `j`, create a connection between them on the map (if none exists), respectively remove the connection (if it exists).

c) `add`: given a party `p`, add the characters in `p` to the player's party in a game.

d) `addAt`: given a node `n` and party `p`, add the characters in `p` to the party at node `n` in a game.

e) `addHere`: given a party `p`, add the characters in `p` to the party at the current location in a game.

f) `remove`, `removeAt`, `removeHere`: these functions are as `add`, `addAt`, and `addHere`, except they remove the given party instead of adding it.

```
ghci> connected [(0,2),(0,1),(1,2),(2,3),(2,4),(3,4)] 2
[0,1,3,4]

ghci> connect 4 0 [(0,2),(0,1),(1,2),(2,3),(2,4),(3,4)]
[(0,2),(0,1),(0,4),(1,2),(2,3),(2,4),(3,4)]

ghci> disconnect 2 0 it
[(0,1),(0,4),(1,2),(2,3),(2,4),(3,4)]

ghci> add ["Dijkstra"] (testGame 0)
Game [(0,1)] 0 ["Dijkstra","Russell"] [[],["Brouwer","Heyting"]]

ghci> remove ["Brouwer","Russell"] it
Game [(0,1)] 0 ["Dijkstra"] [[],["Brouwer","Heyting"]]

ghci> removeAt 1 ["Heyting"] it
Game [(0,1)] 0 ["Dijkstra"] [[],["Brouwer"]]

ghci> addHere ["Heyting","Russell"] it
Game [(0,1)] 0 ["Dijkstra"] [["Heyting","Russell"],["Brouwer"]]
```

Dialogues

A dialogue in the game plays out as follows. The game displays some text, and offers you a list of possible responses, numbered up from 1. It gives a prompt, and you input your choice by entering the corresponding number. Based on this choice, the dialogue continues. The dialogue may end by returning an event (which then updates the game state), and it may branch according to a condition on the game state. This is implemented in the datatype:

```
data Dialogue = Action  String  Event
              | Branch  (Game -> Bool)  Dialogue  Dialogue
              | Choice  String  [( String , Dialogue )]
```

The data type `Dialogue` gives the three options. In the case `Action`, the dialogue should end by displaying a `String` and carrying out an `Event`. In the case `Branch`, the dialogue should test the current game against the given condition `Game -> Bool`, and branch into the first `Dialogue` if `True`, and the second if `False`. In the case `Choice`, the dialogue displays a `String`, and then carries a list of options to display, and for each a continuing `Dialogue`.

A dialogue will be triggered by letting several characters (i.e. a party) talk to each other. Thus, the dialogues for the game are given as the list `theDialogues :: [(Party,Dialogue)]`,

in which we can look up the dialogue for a given party.

Assignment 2: **(15 marks)**

a) Complete the function `dialogue`. For the case `Action`, it should display the `String` and update the game with the `Event`. For the case `Branch`, it should continue according to the condition on the input game. For the case `Choice`, it should:

- Display the `String`.
- If there are no response options, end the dialogue and return the original game state. Otherwise:
- Display the player's response options, numbered from 1.
- Display a prompt (e.g. `>>`) and get user input.
- If the user input is an integer in the displayed list, continue the dialogue with the corresponding `Dialogue` from the list.

You may insert blank lines (`putStrLn ""`) into your output to improve readability. Use `read` to convert a `String` to an `Int`, and if Haskell is confused about the types, try providing one explicitly.

b) Complete the function `findDialogue` that, given a `Party`, finds the corresponding `Dialogue` in the list `theDialogues`. If nothing is found, return a `Dialogue` that just displays the string `"There is nothing we can do."`.

```
ghci> dialogue (testGame 0) testDialogue
Russell: let's get our team together and head to Error
Game [(0,1)] 0 ["Russell"] [[],["Brouwer","Heyting"]]

ghci> dialogue (testGame 1) testDialogue
Brouwer: How can I help you?
  1. Could I get a haircut?
  2. Could I get a pint?
  3. Will you join us on a dangerous adventure?
>> 3
Brouwer: Of course.
Game [(0,1)] 1 ["Brouwer","Russell"] [[],["Heyting"]]

ghci> dialogue Over (findDialogue [])
There is nothing we can do.
Over
```

```
ghci> dialogue Over (findDialogue ["Haskell Curry","William Howard"])
Curry:  You know the way to Error, right?
Howard: I thought you did?
Curry:  Not really. Do we go via Computerborough?
Howard: Yes, I think so. Is that along the I-50?
Curry:  Yes, third exit. Shall I go with them?
Howard: sure. I can watch the shop while you're away.
Over
```

The game loop

The main game loop will display the following things:

- the present location,
- the locations you can travel to, numbered from 1,
- the characters in your party, numbered up from the last location,
- the characters at the present location, numbered up from the previous character.

The game should then ask for the player's input and display a prompt. It should look like this:

```
You are in the 'Non Tertium Non Datur' Brewpub & Barber's.
You can travel to:
  1. Home
  2. Hotel
  3. Takeaway
With you are:
  4. Bertrand Russell
You can see:
  5. Arend Heyting
  6. Luitzen Brouwer
What will you do?
>>
```

The player's input is expected to be one of two kinds:

- a single number for a location to travel to (in the example, `1`, `2`, or `3`), or
- one or more numbers for the characters to start a dialogue (e.g. `4 5` or `6 5 4`).

The game then moves to the requested location or initiates the requested dialogue.

Assignment 3: **(10 marks)**

Complete the functions `step` and `game` that implement the main game loop, in stages:

a) Let `step` display the game state as required. It may return the input game unchanged. Short location names (for travel) are given in `theLocations`, long names (for descriptions) are given in `theDescriptions`.

b) Let `step` display a prompt and take user input. It may display the user input without taking action, and return the game unchanged.

c) Let `step` check if the user input corresponds to a single location, and if so, return the game state with the new location.

d) Let `step` check if the user input matches one or more characters, and if so: build the `Party` corresponding to these inputs; look up the corresponding `Dialogue`; run the dialogue (with the current game state as input) and return the resulting game state.

e) Complete `game` to do the following: start a loop from the `start` state, which ends if the game is `Over`, and otherwise takes a `step` and continues the loop.

Your functions do not need to account for unexpected input just yet; they may return the game state unchanged, or crash, or anything really. We will fix that in the next assignment.

```
ghci> step (testGame 0)

You are in your own home. It is very cosy.
You can travel to:
  1. Brewpub
With you are:
  2. Russell
What will you do?
>> 2

Russell: Let's go on an adventure!
  1. Sure.
  2. Maybe later
>> 1

You pack your bags and go with Russell.
Game [(0,1)] 0 ["Russell"] [[],["Brouwer","Heyting"]]
ghci> step it
```

```
You are in your own home. It is very cosy.
You can travel to:
  1. Brewpub
With you are:
  2. Russell
What will you do?
>> 1
Game [(0,1)] 1 ["Russell"] [[],["Brouwer","Heyting"]]
ghci> step it

You are in the ‘Non Tertium Non Datur’ Brewpub & Barber’s.
You can travel to:
  1. Home
With you are:
  2. Russell
You can see:
  3. Brouwer
  4. Heyting
What will you do?
>> 4 2

Heyting: Hi Russell, what are you drinking?
Russell: The strong stuff, as usual.
Game [(0,1)] 1 ["Russell"] [[],["Brouwer","Heyting"]]

ghci> game

You are in your own home. It is very cosy.
You can see:
  1. Bertrand Russell
What will you do?
```

Safety upgrades

The game now works, but it may crash on unexpected inputs. Moreover, crashing (or winning) is the only way to exit a dialogue or the game. Here, we will fix that.

Assignment 4: **(5 marks)**

Add the following safety features. The library `Data.Char` will be helpful.

a) In a dialogue, entering the number zero should immediately end the dialogue and return the original game.

b) In the main game loop, entering zero should exit the game, back to the prompt.

c) An unexpected input should no longer crash the game, and only re-iterate the prompt (or display a message about expected inputs). Expected inputs are: in a dialogue, either 0 or a number corresponding to a response choice; in the main loop, either 0, a single number corresponding to a location (in the given list), or one or more numbers each corresponding to a character in your party or at the present location.

Solving the game

For the final part of this coursework, we will implement the necessary algorithms to solve the game automatically and produce a crude walk-through. You are given a data type `Command` to store three kinds of instructions: a sequence of travel steps; selecting a party for a dialogue; and choosing a path through a dialogue. These correspond to the kinds of input the player may give in the game. The solution will be a list of such commands.

The structure of the solver will be as follows. A single **step** in the algorithm will take all the necessary actions to progress to the next event: travel to a location, select characters to talk to, and navigate the dialogue to find an action. To take a step, the solver will find all possible steps, and then select the first that progresses the game (in a way that will be made precise). The solver will then continue taking steps until the game is won.

Assignment 5: **(15 marks)**

a) Complete the function `talk` which finds all the paths through a dialogue that end in an `Action`. Apply the `Event` to the input game, and return the list of selections needed to reach it.

b) Complete the function `select` which, given a `Game`, finds all possible parties composed of characters in the player's party or at the present location (i.e. all the possible ways of starting a dialogue).

c) Complete the function `travel` which finds a path `[Int]` on the map to each reachable node. The indices in the path should be the indices of the choices made, as entered by the player, and not the nodes visited. There should be only one path for any given location. It is preferable to give a shortest path for each node, and to list nodes by distance (closest first).

d) Complete the function `allSteps` which gives all combinations of travelling to a location (including staying put), selecting a dialogue, and reaching an `Action`. For each such step, return the new `Game`, and give the `Solution` as a list of a `Travel`, `Select`, and `Talk` commands.

e) Complete the function `solve` that solves the game. It should keep taking steps until the game is `Over`, and build up a `Solution` while doing so.

When `solve` is complete, you can use the provided function `walkthrough` to display your solution.

```
ghci> map snd (talk (testGame 1) testDialogue)
[[2,1],[2,2],[3]]

ghci> select (testGame 1)
[[],["Russell"],["Heyting"],["Heyting","Russell"],["Brouwer"],
["Brouwer","Russell"],["Brouwer","Heyting"],
["Brouwer","Heyting","Russell"]]

ghci> travel [(0,2),(0,1),(1,2),(2,3),(2,4),(3,5)] 0
[(0,[]),(2,[1]),(1,[2]),(3,[1,3]),(4,[1,4]),(5,[1,3,2])]

ghci> g = Game theMap 1 [] theCharacters
ghci> map fst (allSteps g)
[ [Travel  [], Select ["Luitzen Brouwer"], Talk [3,1]]
, [Travel [1], Select ["David Hilbert"]  , Talk [1,1,1,1]]
, [Travel [1], Select ["David Hilbert"]  , Talk [2]]]

ghci> walkthrough
Select: Bertrand Russell
Talk:   1 1
Travel: 1
Select: Luitzen Brouwer
Talk:   3 1
Travel: 2
Select: David Hilbert
Talk:   1 1 1 1
Travel: 2
Select: William Howard
Talk:   1
Travel: 1 1 3
Select: Haskell Curry, William Howard
Travel: 2
Select: Bertrand Russell, Gottlob Frege, Luitzen Brouwer
Talk:   1 1 1
```