Coursework 1 Due: Monday 9 April 6pm

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`.

Spoiler warning The coursework contains two sets of game data. The default set is spoiler-free, but the other has spoilers for the entire Game of Thrones TV series. This set is in a separate file; you are free to swap it in, if you like, or delete it.

Instructions Complete the given assignments in the file `coursework_1.hs`, and submit this on Moodle by Monday 9 April 6pm. We will provide feedback in the form of individual marks and a common feedback document within three semester weeks.

In this coursework we will complete the adventure game that we've been building in the tutorials. In your file `coursework_1.hs` we have collected the relevant parts from the tutorials 1–5. First, we will finalize the `Game` data type, and create a type `Event` for events that change the game state.

Part 1 (25%):

a) The data type `Game` contains the player's `Node` and `Party`. Change it to have two cases: a constructor `Won` with no arguments, for when the game is won, or the current `Game` for a game in progress. Then change the `Game` constructor so that it additionally stores the characters at each game location, as a list `[Party]`. Uncomment `start` and `end` (your file should type-check).

b) Create a new type `Event` as a synonym for `Game -> Game`.

c) Complete the function `applyAt` which takes an index `i`, a function `f :: a -> a`, and a list `xs :: [a]`, and applies `f` to element number `i` in `xs` (where the head is element zero). You may give an error when `i` is out of bounds for `xs`.

d) Complete the function `updateAt`. It takes a node `m`, parties `xs` and `ys`, and a game `(Game n p ps)`. It should update the party in `ps` at index `m` by first removing `xs` and then adding `ys`, and return the resulting game. You should use `merge`, `minus`, and `applyAt`. A game `Won` should be left unchanged.

e) Complete the function `update` similarly to `updateAt`. It takes parties `xs`, `ys`, and `zs`, and a game `(Game n p ps)`. It should remove `xs` from the party `p` and then add `ys`, and remove `xs` from the party in `ps` at index `n`, and then add `zs`.

```
*Main> applyAt 2 (+3) [1,2,3,4]
[1,2,6,4]

*Main> let g = Game 0 ["Mario"] [["Luigi"],["Bowser","Peach"]]
*Main> updateAt 1 ["Bowser"] ["Yoshi"] g
Game 0 ["Mario"] [["Luigi"],["Peach","Yoshi"]]

*Main> update ["Luigi","Mario"] ["Wario"] ["Waluigi"] it
Game 0 ["Wario"] [["Waluigi"],["Peach","Yoshi"]]

*Main> update ["Luigi","Mario"] ["Wario"] ["Waluigi"] Won
Won
```

In tutorial 4 we introduced the `Dialogue` type, to store the dialogue options for interacting with objects in the game. At the `End` of a dialogue only a string was displayed. Now, we want to trigger a game `Event`. This is what we will implement next, along with an IO function to work with `Dialogue` trees. To activate a dialogue within the game is done by combining the right characters. The list `dialogues` near the end of your file records which `Party` is needed to activate each `Dialogue`.

Part 2 (25%):

a) Add a constructor `Action` to the data type `Dialogue` which stores a `String` and an `Event`. Un-comment the `dialogues` list, and see if your file type-checks.

b) Complete the function `dialogue`. It should pattern-match on the `Dialogue` type. For `End`, the `String` should be displayed, and the input game returned unchanged. For `Action`, the `String` should be displayed, and the `Event` used to update the input `Game` —that is, it should return the input game with the event applied to it. For `Choice`, it should:

- Display the `String`.
- Display the player's response options, formatted with `enumerate`.
- Get user input.
- If the input is in `exitWords`, exit the dialogue and return the input game unchanged.
- Otherwise, 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.

c) Complete the function `findDialogue` that, given a `Party`, finds the corresponding `Dialogue` in the list `dialogues`. If no action is found, return a `Dialogue` that just displays the string `"There is nothing we can do."`. Note that we expect a `Party` to be a sorted list: we may expect e.g. `["Luigi","Mario"]` as input, but not `["Mario","Luigi"]`. (Every `Party` in `dialogues` is sorted.)

```
*Main> let g = Game 0 [] [["Mario"]]
*Main> dialogue g (snd (head dialogues))
I need to save the Princess.
  1. Sure.
  2. Not right now.
1
Let's go.
Game 0 ["Mario"] [[]]

*Main> dialogue Won (findDialogue [])
There is nothing we can do.
Won

*Main> dialogue Won (findDialogue ["Team Rocket"])
Oh, prepare for trouble, that's what they should do.
And make it double, we're grabbing Pikachu.
Won

*Main> dialogue Won (findDialogue ["Mario","Peach"])
Save me, Mario!
  1. Sure.
  2. Not right now.
2
Mario, pls.
Won
```

To complete our game, we will integrate the dialogues with the main game loop.

Part 3 (25%):

a) Take the functions `loop` and `game` from tutorial 5 (or from the solutions on Moodle, once they appear). Update `loop` to work with the new definition of `Game`. For an input `Won`, it should return nothing, and for `Game n p ps` it should leave `ps :: [Party]` unchanged.

```
*Main> game
You are in Princess Castle
You can travel to
  1. Church of Halo
  2. Nintendo Land
With you are
What will you do?
```

b) Update `loop` so that it also displays a line `"You can see"` followed by a numbered list of the objects at the current location (following up on the player's objects).

```
*Main> game
You are in Princess Castle
You can travel to
  1. Church of Halo
  2. Nintendo Land
With you are
You can see
  3. Peach
  4. Rochelle
What will you do?
```

c) Adapt `loop`, so that when the user inputs a set of numbers corresponding to objects at the current location or held by the player, the game runs the corresponding dialogue. Follow these steps:

- If the input is in `exitWords`, exit the game loop as before. Otherwise, interpret the user input as a list of numbers and bind that to a variable `is`. Use the functions `words` which splits a string into a list of strings, and `read` which can interpret a string as an integer (demonstrations below).
- Next, if the input is a single number corresponding to a location, the player should travel to that location, as before. Otherwise (if the input is not a location, or multiple numbers):
- The input may have numbers from both the `"With you are"` and `"You can see"` lists, in any order. First, select the chosen charaters from both lists, and combine them in a party (an ordered list) `js`. There are several ways to do this: one is to select the items and then sort them; another is to first sort the input numbers, select the inputs from both lists of characters, and then merge them; yet another is to zip both input lists with their input numbers, merge them, and then find those characters whose number occurs in `is`.

- Find the dialogue corresponding to `js` and run it (together with the input game), and continue the game `loop` with the `Game` that this returns.

```
*Main> words "A wild recursion appears!"
["A","wild","recursion","appears!"]

*Main> read "404" :: Int
404


*Main> game
You are in Princess Castle
You can travel to
  1. Church of Halo
  2. Nintendo Land
With you are
You can see
  3. Peach
  4. Rochelle
What will you do?
2

You are in Nintendo Land
You can travel to
  1. Church of Halo
  2. Pallet Town
  3. Princess Castle
With you are
You can see
  4. Chell
  5. Cortana
  6. Mario
  7. Master Chief
What will you do?
5

I must go with Master Chief.
  1. Sure.
  2. Not right now.
1
Let's go.
```

```
You are in Nintendo Land
You can travel to
  1. Church of Halo
  2. Pallet Town
  3. Princess Castle
With you are
  4. Cortana
You can see
  5. Chell
  6. Mario
  7. Master Chief
What will you do?
7

I want to marry Cortana. Can you escort us to the Church of Halo?
  1. Sure.
  2. Not right now.
1
Let's go.

You are in Nintendo Land
You can travel to
  1. Church of Halo
  2. Pallet Town
  3. Princess Castle
With you are
  4. Cortana
  5. Master Chief
You can see
  6. Chell
  7. Mario
What will you do?
1

You are in Church of Halo
You can travel to
  1. Nintendo Land
  2. Princess Castle
With you are
```

```
  3. Cortana
  4. Master Chief
You can see
  5. Priest
What will you do?
5 4 3                -- Note the order here: this should work!

Do you, Master Chief, accept Cortana to be your beloved bride?
...
```

Your game is now complete! If you have had quite enough of Mario and Pikachu, there is also a set of game data from Game of Thrones - beware that it is not spoiler-free! (And we are still looking for someone to update it with last season of the TV series. Volunteers?) To play the GoT version, use `game_data_GoT.hs` to replace the current set of game data. Also, you might like to build your own game content. We can't wait to play!

For the remainder of the coursework, we will implement the necessary algorithms to solve the game automatically, and produce a walk-through.

> **Time management warning** This part of the coursework is significantly more challenging. It is possible to sink in large amounts of time for only few marks awarded. If you attempt it, we strongly advise to **first** finish any other outstanding coursework (for other units), which will almost certainly carry a greater overall reward for your time.

Part 4 (25%): We will build an auto-solver in three parts: first, we will explore `Dialogue`s; second, we will traverse a `Map`; third, we will combine these into a step that progresses the game, by travelling to a location and then performing an action.

a) Complete the function `talk'` that takes a `Dialogue` and returns a list containing, for each branch leading to an `Action`, the pair of the `Event` plus the list `[Int]` of choices made to reach it. An `End` should be ignored.

b) Use `talk'` to complete the similar function `talk`, which returns a list of pairs of each `Event` in a dialogue plus a `String` giving the instructions needed to reach it, indicated with `"In the dialogue, choose"`. (If the dialogue ends immediately and there are no changes to be made, you may choose to return the empty string `""` instead of `"In the dialogue, choose"`, as the examples do.)

Your coursework file contains a test dialogue and tests for `talk'` and `talk`. Note that since Haskell can't show an `Event` (because it is a synonym for `Game -> Game`) we first turn each `Event` into a `Game`, by applying it to `Won`.

```
*Main> testTalk'
[(Game 0 ["Event: You wake up in bed"] [],[1])
,(Game 0 ["Event: You are a battery"] [],[2])]

*Main> testTalk
[(Game 0 ["Event: You wake up in bed"] []
 ,"\nIn the dialogue, choose 1")
,(Game 0 ["Event: You are a battery"] [],
 ,"\nIn the dialogue, choose 2")]

*Main> [ (e Won, xs) | (e,xs) <- talk' (snd (dialogues !! 13)) ]
[(Won,[1,2]),(Won,[2,2,1]),(Won,[2,3,2])]

*Main> (putStr . unlines . map snd . talk . snd) (dialogues !! 13)
In the dialogue, choose 1 2
In the dialogue, choose 2 2 1
In the dialogue, choose 2 3 2
```

Next, we will explore the map. We will work with pairs `(Node,[Int])` where the `Node` is a given location, and `[Int]` the path taken to travel there. The path indicates the choices made in the game while travelling. We will use accumulating functions, so the path will initialy be in reverse.

c) Complete the function `extend` which takes a `Map` and a pair `(Node,[Int])` of a current location and previous path (in reverse). It should return a list `[(Node,[Int])]` where the `Node`s are the locations that can be reached in one step from the input location, and each list `[Int]` is the new path to get to that location, extending the input path by one step (at the front). Note that the integers in a path are the choices made by the player in the game, not the `Node`s visited.

d) Complete the function `travel'`, which takes a `Map`, an accumulating list of previously visited locations `[(Node,[Int])]`, and a list `[(Node,[Int])]` of locations that can be reached in one step from the visited locations. It should output the locations (and their paths) reachable in any number of steps, as follows. When there are no more (one-step) reachable locations, we are done and we return the visited locations. Otherwise, we consider the next reachable location. If it has already been visited, we ignore it; if not, we add it as "visited". Then we use `extend` to find the one-step reachable locations from there, and add those to the appropriate list. (This is an algorithm for **graph search**; if you add new locations to the **front** of the one-step reachable locations, it is **depth-first**; if you add them to the **end**, it is **breadth-first**. Examples use breadth-first unless noted.)

e) Complete the function `travel` to wrap up. Given a `Map` and a `Game`, it should return a list `[(Game,String)]` of each `Game` that results after travelling to a reachable location, plus nicely formatted directions as in the examples below. Use `travel'` with suitable initial values, and remember to reverse the list of directions if necessary. Make sure that the current location is included (though you don't need to give a special message `"Stay in ..."` for it, as the examples do).

In the demonstration below, note that the paths found by `travel'` and `travel` on `theMap` (as well as the order in which they appear) may differ. To test if directions are correct, try them in-game!

```
*Main> extend theMap (2,[1,3])
[(4,[1,1,3]),(6,[2,1,3])]

*Main> travel' [(0,1),(1,2),(1,3),(2,4),(2,5)] [] [(3,[])]
[(3,[]),(1,[1]),(0,[1,1]),(2,[2,1]),(4,[2,2,1]),(5,[3,2,1])]

-- Breadth-first
*Main> (putStr . unlines . map snd . travel theMap) start
Stay in Princess Castle
Travel to Church of Halo: 1
Travel to Nintendo Land: 2
Travel to Pallet Town: 2 2
Travel to Aperture Science: 2 2 1
Travel to Macon: 2 2 2

-- Depth-first
*Main> (putStr . unlines . map snd . travel theMap) start
Stay in Princess Castle
Travel to Church of Halo: 1
Travel to Nintendo Land: 1 1
Travel to Pallet Town: 1 1 2
Travel to Aperture Science: 1 1 2 1
Travel to Macon: 1 1 2 2
```

Finally, we will put the above together to auto-solve the game.

f) Complete the function `act` which given a game, tries all possible dialogues at that location, returning a list `[(Game,String)]` of the resulting game plus the names of the characters needed at that location. Build it up as a list comprehension, as follows:

- examine all possible dialogues in `dialogues` ;
- for a given pair `(Party,Dialogue)` , see if all characters are available to the player in the game, i.e. in the current party or at the current location;
- for that `Dialogue` , use `talk` to explore all possible `Event` s;
- as output, collect the resulting game (after applying the `Event`) plus the instructions needed to give that outcome.

Format your instructions nicely. If the game is `Won` , return the empty list.

```
*Main> (putStr . unlines . map snd . act) (Game 4 [] characters)

Talk to Mario
In the dialogue, choose 1

Talk to Mario
In the dialogue, choose 2

Talk to Master Chief
In the dialogue, choose 1

Talk to Master Chief
In the dialogue, choose 2

Talk to Cortana
In the dialogue, choose 1

Talk to Cortana
In the dialogue, choose 2

Talk to Chell
In the dialogue, choose 1

Talk to Chell
In the dialogue, choose 2
```

Instead of finding **all** paths to a solution, we will design our solver such that, at any point, it can find a step that means guaranteed progress towards a solution. Then, we need only consider one step at a time. In the current game design, the steps that are **suitable** in this way are those that:

- win the game,
- add a character to the player's `Party`,
- add a new character to the game, in any location.
- move the game to a new location (GoT version only—see end of this document).

g) Complete the function `suitable` which, given a `Game` and an `Event`, determines whether the event is suitable in the sense above. Update your function `act` so it considers only suitable events.

```
*Main> (putStr . unlines . map snd . act) (Game 4 [] characters)

Talk to Mario
In the dialogue, choose 1

Talk to Master Chief
In the dialogue, choose 1

Talk to Cortana
In the dialogue, choose 1

Talk to Chell
In the dialogue, choose 1
```

h) Complete the function `solve` that solves the game and produces a walkthrough. A good way is to use the auxiliary function `solveLoop`, with an accumulator as `(Game,String) -> String` or without `Game -> String` (try which you find easier; solutions not using `solveLoop` are fine, too). The function `solveLoop` should travel to all locations with `travel`, find all suitable actions at that location with `act`, then recurse on one suitable action only, until the game is `Won`.

```
*Main> solve

Travel to Nintendo Land: 2
  Talk to Mario
  In the dialogue, choose 1
Stay in Nintendo Land
  Talk to Master Chief
  In the dialogue, choose 1
Stay in Nintendo Land
  Talk to Cortana
  In the dialogue, choose 1
Stay in Nintendo Land
  Talk to Chell
  In the dialogue, choose 1
Travel to Church of Halo: 1
  Talk to Cortana, Master Chief, and Priest
  In the dialogue, choose 2 2
Travel to Princess Castle: 2
  Talk to Mario and Peach
  In the dialogue, choose 1
Stay in Princess Castle
  Talk to Baseball Cap
  In the dialogue, choose 1
Travel to Church of Halo: 1
  Talk to Baseball Cap and Clementine (hiding)
  In the dialogue, choose 1
Travel to Aperture Science: 1 2 1
  Talk to Chell and Portal Gun
Travel to Macon: 1 2
  Talk to Clementine and Lee
  In the dialogue, choose 1
Travel to Princess Castle: 1 3 3
  Talk to Rochelle and Zombie Lee
Stay in Princess Castle
  Talk to Pikachu
  In the dialogue, choose 1
Travel to Nintendo Land: 2
  Talk to Ash and Pikachu
```