COMP5450 Assessment 2 – Lambda Run (Brief v1.0.1)

Lambda Run is a game in which a plucky red panda ("Lambda") tries to find their way home through a pleasant land of grass, trees, and rocks, but who quickly gets tired. Can Lambda make it home to Haskell the Ultimate before running out of energy? Fortunately there are tasty chocolate snacks along the way to keep Lambda going, but a number of keys need to be found to open doors, making the way yet more perilous. Can Lambda find enough chocolate to sustain their journey home to Haskell?

The game is written in Haskell and JavaScript (with a little HTML/CSS for the front end). *Your goal is to complete the implementation of the Haskell component* which generates a JavaScript file providing the game data and JavaScript functions to implement the behaviour of various in-game items. You control the game using the arrow keys as input to navigate the map.

Provided code and scaffolding

Download `pack.zip` from the Moodle which provides:

- `frontend` directory – the frontend HTML/JS/CSS parts of the game, which you do not need to look at or understand.
- `game.html` – the entry point for playing the game.
- `config.js` – This is the file that your Haskell code will generate to provide the remaining functionality and setup for the game. In `pack.zip`, a configuration is provided as generated from a model solution that you might like to try to emulate. (Since your code will re-generate `config.js` you will probably want to make a copy of the provided one otherwise it will be overwritten).
- Two Haskell modules (`Game.hs` and `JavaScript.hs`) that give the scaffolding for your work. You should read the documentation in these files to understand what is provided.

The `Game` module provides core data representations and functions for the game. The key types are:

```
data MapItem          = ...
data KeyColor         = ...
type Room items       = [[Either MapItem items]]
type RoomId           = String

data GameConfig items = GameConfig {
    rooms         :: [(RoomId, Room items)]
  , actions       :: (items -> JSExpr)
  , actionItems   :: [items]
  }
```

- The `MapItem` data type represents items on the map that are largely inert (trees, rocks, etc.) which you do not need to implement the behaviour of. The `KeyColor` data type enumerates three colours for keys, though note that `MapItem` doesn't represent keys (you will implement this later).
- The `Room` type synonym defines a game room as a list of lists, whose elements represent squares ("cells") of the game and are either `MapItem` or of some parametric type `items` which will be the type of *action items* in the game and which you will later specialise with your own data type for action items.
- The name of a room is just a string, given by the type synonym `RoomId`.
- The `GameConfig` data type (shown in full here) has three arguments: (i) an association list between room names and `Room` representations; (ii) a compilation function that maps action items to a JavaScript expression syntax tree `JSExpr` which describes a transformation on player data if Lambda walks over this kind of item; (iii) a list of action items that we want to compile JS functions for as given by (ii).

The `Game` module then provides a key function which you will use in your work:

```
outputConfig :: GameConfig items -> (items -> String) -> IO ()
```

Given a `(GameConfig items)` value, i.e., a game configuration with action items of type `items`, and a function that computes a CSS class name (as a string) for those items, then `outputConfig` compiles everything to JavaScript and writes `config.js` for the game to be played via `game.html`.

The `JavaScript` module This module defines an AST representation of a subset of JavaScript code, with a representation of JavaScript expressions `JSExpr`, JavaScript statements `JSStmt`, and JavaScript code blocks `JSBlock`. Read the code documentation to see which data constructors are provided and what these map onto as JavaScript code.

You can always generate your own AST fragements to experiment and use the corresponding `compileExpr :: JSExpr -> String`, `compileStmt :: JSStmt -> String` and `compileBlock :: JSBlock -> String` functions to see what JavaScript code is produced.

Tasks

Complete your work in a file named after your login, e.g. `dao7.hs` for me, and submit just that file. You must not modify any of the other provided files apart from for Q5 if you do it (in which case submit a Haskell file for Q1-4 and `q5.zip` with everything for Q5).

You will need to import the `Game` and `JavaScript` modules. You may use any other library functions. **Your work must be your own. Please use comments to indicate to which question each piece of code belongs and please provide solutions in the order of the questions here.**

The total mark is 100. A 5% penalty will be applied for any code that does not compile due to type errors. A 10% penalty will be applied for any code that does not compile due to syntax errors.

1 Simple map generation (10 marks)

The first task is to create an initial version of the game configuration with an empty room.

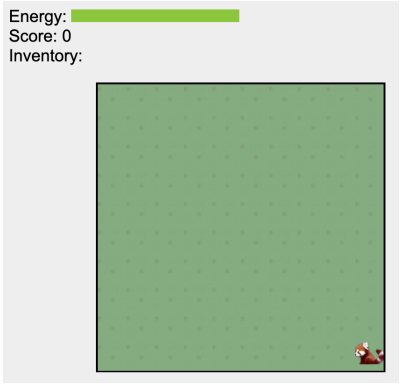
(a) Define a function to generate square empty rooms (containing just grass) of an arbitrary size.

That is, for an integer parameter n, generate a value of type `Room ()` representing a room of size $n \times n$ containing just grass, i.e., a list of length n containing lists of length n containing the map item `None`. The action item type here is just `()`, i.e., we are not giving any action items yet. (5 marks)

(b) Define a variable `initGameConfig` of type `GameConfig ()` which should contain an empty room of size 10×10 associated to the name `"start"` and a trivial actions function that always produces the `Null` JavaScript value, i.e., there are no actions items to compile functionality for. (3 marks)

(c) Define a value `q1` that applies `outputConfig` to your answer to (b) and with a function that just outputs the empty string for the action item names (since there aren't any yet). (2 marks)

Evaluating `q1` should generate `config.js` which should give you a view as follows in the game:



2 Map builder (25 marks)

The second task is to build some helper functions to make it easier to create interesting rooms in the game. The idea will be that you can define a map via a list of strings with characters standing in for items in the game, and then having a function translate this into the `Room` representation.

(a) Define a function of polymorphic type:

```
mapBuilder :: (Char -> Either MapItem a) -> [String] -> Room a
```

that takes as input a function for mapping characters into the representation of either a map item or an action item `a`, and takes a further input which is a list of strings to then convert into a `Room`, using the first function, e.g., each string in the list defines a row of the room map.

For example, the following would generate a 3×3 empty room with a tree in the middle:

```
example = mapBuilder (\x -> case x of 'T' -> Left TreeA; _ -> Left None)
  ["---"
  ,"-T-"
  ,"---"]
```

(5 marks)

(b) Define a function converting characters to `MapItem` values for each possible data constructor, with suitable choices of character for each. For doors, you can construct doors with no target and no requirements. (6 marks)

(c) This generates maps that can get quite dense so we will define a function for "dilating" a map by doubling its size by adding empty columns and rows.

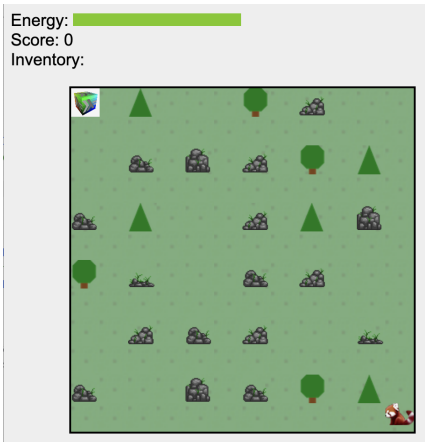
(i) Define a function `a -> [a] -> [a]` that produces a list double the size of the input list by inserting the first input after every element, e.g. `expand '-' "HDT" "H-D-T-"`.
Define this using `foldr` for full marks (otherwise max 3 marks). (5 marks)

(ii) Using your answer to the above, define a function `dilate :: Room a -> Room a` that insert an empty cell between each cell and an empty row between each row such that the output room is twice as wide and twice as high as the input room. (5 marks)

(d) Putting this all together, define a new map of size 12×12 by using `dilate` and `mapBuilder` with your answer to (b). Include a good selection of trees and rocks of different types, and a "home" square in the top-left. You need not include any doors yet.

Define `q2` to generate an output configuration from your new map (where, as in `q1`, you can use a function that just outputs `Null` for action compilation and a function that just outputs empty string for action item class names since we haven't specified any yet). (4 marks)

Evaluating `q2` should generate `config.js` which should give you a view like:



3 Action items (40 marks)

(a) Define your own data type `ActionItem` to represent keys with their colour given by the `KeyColor` type in module `Game`, chocolate snacks, and coins of either copper or gold colour. (6 marks)

(b) Define a function `actionItemName :: ActionItem -> String` to give for each action item its CSS class name for the front end, which should be `keyblue`, `keyyellow`, `keyorange`, `food`, `coincopper`, and `coingold` for your items here. (3 marks)

(c) Using your function from Q2(b), define a new function mapping `Char` to `Either MapItem ActionItem` that gives a conversion from a suitable character to represent map items and your action items in the map builder. (5 marks)

We now need to work with the JavaScript AST representation so each action item has a JavaScript function that tells the game what happens to the player data when the player moves over that item.

For this we need to know that player data is stored in a JavaScript object initialised as:

```
var playerData =
  { energy: 20, score: 0, inventory: [] }
```

We can access a property of such an object in the usual way in JavaScript, e.g., `playerData.energy` which returns the energy field and is represented by the `Property` constructor of `JSExpr`. We can construct new values of this object form similar to the above, which is represented by the `Object` constructor of `JSExpr`.

The JavaScript engine expects that for every action item there is a function taking a player data object as input and returning a player data object, with potentially some values changed. We will build such functions as `JSExpr`, for which the `Function` constructor can be used (see the documentation).

(d) Define `jsId :: JSExpr` that defines an AST representation of an identity function in JavaScript. (4 marks)

(e) As a key piece of helper code, define a function:

```
updatePlayerData :: [(String, JSExpr -> JSExpr)] -> JSExpr
```

which will create the AST of a JavaScript function from player data objects to player data objects.

The first input here gives an association list mapping from field names to a transformation on JavaScript syntax `JSExpr -> JSExpr`.

Your `updatePlayerData` should create an AST for a function that takes an input variable, call it `pdata` (which will be a player data object) and outputs code (in AST form) for a new player data object, where each field of `energy`, `score`, and `inventory` is derived from its corresponding field in `pdata` with any transformations applied that are given by the input association list.

For example:

```
updatePlayerData [("score", \x -> BinOp "+" x (Num 1))]
```

should produce an AST value equivalent to the JavaScript code:

```
function (pdata) {
  return {energy: pdata.energy, score: pdata.score + 1, inventory: pdata.inventory}
}
```

Your `updatePlayerData` should ignore any items in the association list that do not describe the three fields needed for player data. (10 marks)

(f) Using your answer to the previous question (or otherwise if you weren't able to complete it) define a function `myActions :: ActionItem -> JSExpr` that implements the following behaviour for action items: gold coins increase the player score by 2, copper coins increase the player score by 1, food increases the player energy by 10, and any other item acts as the identity function (e.g., as you defined it in part (d) above). (8 marks)

(g) Using `mapBuilder`, define a new 12×12 room map that includes your food and coins, as well as trees, rocks, and a home square in the top corner.

Define `q3` that configures the game with your map and uses `actionItemName` to give the class names for your action items, and `myActions` to give the behaviour of the action items. The third argument of `GameConfig` should list all action items we now want to compile functions for. (4 marks)

Evaluating `q3` should generate `config.js` which should give you a view like:

4 Doors (20 marks)

Finally, we will add the ability to work with keys and doors and define larger maps.

(a) Define a function:

```
insertAt :: Either MapItem item -> (Int, Int) -> Room item -> Room item
```

which allows an item (either a map item or an action item) to be inserted into a room (third input) at the co-ordinates specified by the second input (with the meaning `(row, column)`), e.g.

```
  insertAt (Right Food) (1, 0) [[Left None, Left None], [Left None, Left None]]
= [[Left None,Left None],[Right Food,Left None]]
```

(4 marks)

(b) Using your previous functions and `insertAt`, create three rooms.

- A starting room containing an orange and blue key and with a door to north that requires the orange key.
- A room containing a yellow key that will be to the north of the starting room with a door leading back to the south (in corresponding position to the original door), and a door to the east requiring orange and yellow keys.
- A final room leading off the previous with a corresponding door going back to previous room, and the final home square in the bottom-right corner.

Include food, coins, rocks, and trees in all. (8 marks)

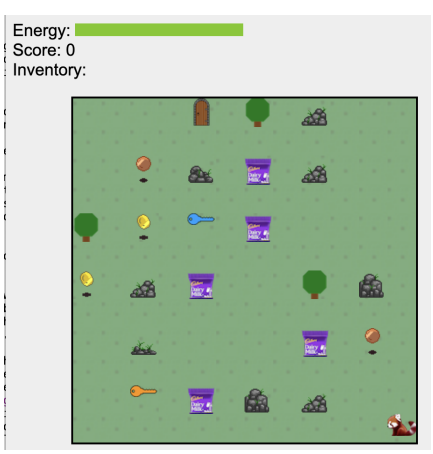
(c) Define a new function `ActionItem -> JSExpr` for specifying the JavaScript behaviour of action items which has the same behaviour as your previous `myActions` but now includes a case for keys.

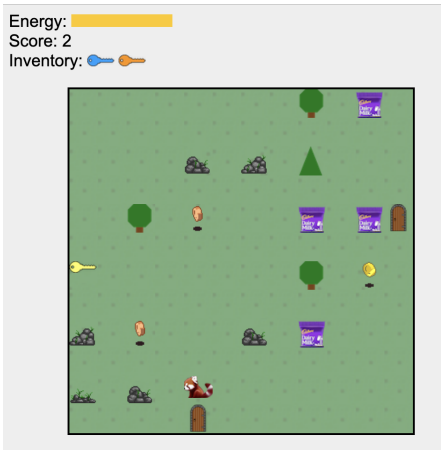
The behaviour should be to update the `inventory` property (which is a JS list) by adding in a string corresponding to the key colour. In `game.js` there is a function `cons` which takes an element, a list, and performs the cons operation, returning the new list. Your generated JavaScript can use this function. (5 marks)

(d) Define `q4` that puts all of the above together, specifying a game config with the three rooms (with names that match with the naming implied by the doors) and making sure to list all coin, food, and key items that are now covered in the `actionItems` argument of `GameConfig`.

You may need to tinker with the number of snacks to make your map playable but challenging. (3 marks)

You should now have a fully playable game! An indicative example of the starter and north rooms is:





5 Extension (5 marks)

There is a lot that one could do to make this game more fun! The last part of the assessment, if you have time, is to add a new kind of item to the game that has an interesting behaviour, or some other game behaviour. Be creative!

It must be something requiring further Haskell, but you will likely need to slightly modify the frontend code (e.g., to add a new CSS class for a new items image, see `frontent/style.css`). You may include new images. (5 marks)

If you answer this question please zip all files for the whole game and submit it as q5.zip. Make sure there is a clear `q5` function to generate `config.js` for your extension and include a brief explanation in your code about what you have done. This part will be marked as all or nothing. If you have done something new and different, it's worth 5 marks.