Text-based Adventure Game

Overview and Summary of Functionality

For this practical, we were asked to implement a text-based adventure game using Haskell. The user would be able to input commands using text to navigate rooms, interact with objects and perform actions.

The program and the tests can be run using “cabal run CoffeeTime”, “cabal run Tests” in command line respectively from the source directory. (“cabal build” may need to be run first)

Our repository link is:

https://kb267.hg.cs.st-andrews.ac.uk/CoffeeTime

Our program implements the following functionality:

- Basic Requirements:
  - Uses helper methods to parse input arguments from Strings into Commands, Actions, Objects and Directions so that they can be used to modify the game state
  - Uses helper methods to perform and validate user actions.
  - Allows the user to move between areas in the game world
  - Allows the user to pick up and drop objects
  - Allows the user to pour and drink coffee
  - Allows the user to open the front door, but only when caffeinated
  - Allows the user to examine objects and list their inventory
- Additional Requirements:
  - Allows for objects to be assigned as heavy, which means they cannot be picked up
  - Allows for rooms to be assigned as useless or locked which prevents entering them
  - Includes additional rooms (Hallway, Diner, Basement) featuring new puzzles (See fig 1)
  - Allows the user to use a custom fix action to repair the added statue
  - Allows the user to save, load, quit and reset the game.
  - Implements Haskeline for taking user input instead of the inbuilt IO library
  - Uses QuickCheck tests to ensure the program is robust
  - Contains a cabal file including instructions on how to build the program and information on its dependencies
  - Uses list comprehensions and higher order functions
  - Includes an alternative endgame condition where the user can lose after being eaten by a monster

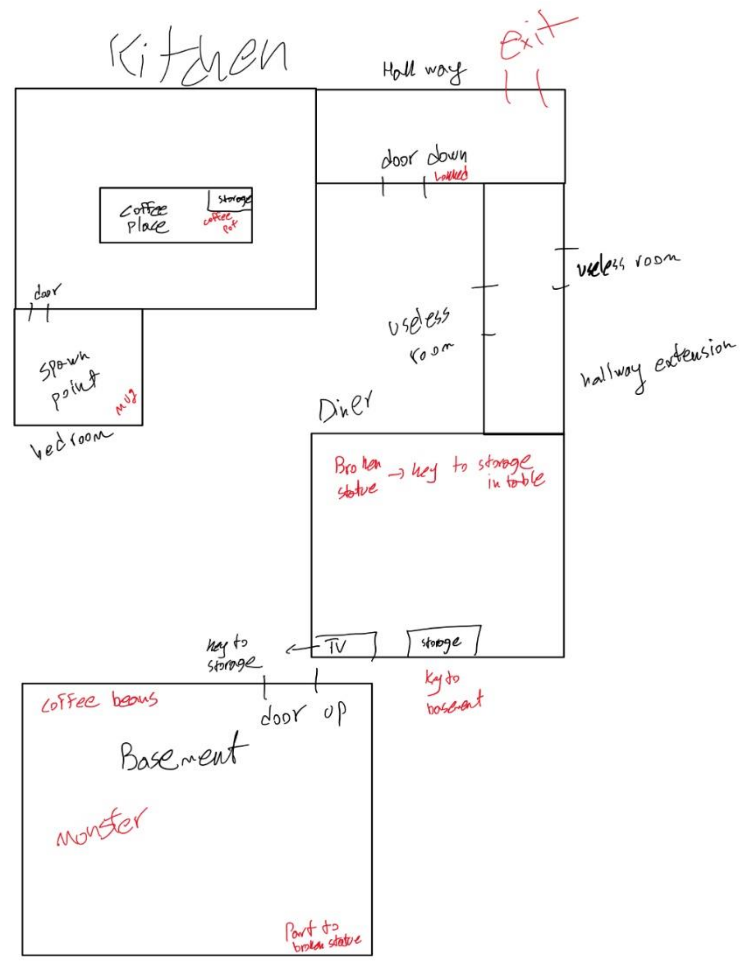


Fig. 1: Schematic view of the extended game map

Notable Challenges & Solutions

Moving Objects interfering with Validation

To prevent opening/repairing objects from the wrong room, validation for these scenarios include their default location. By moving these objects this functionality could become very broken.

To intuitively solve this, we added the ability to assign objects as 'heavy' which meant they could not be picked up, avoiding these awkward scenarios while adding to the game's features.

Loading not being able to Parse the (String, Room) pair

When using a pre-written instance of show for room and gameData, the parser expected a different string than the one that would be provided, causing errors.

To fix this, the show instances of GameData and Room had to be removed and rewritten as functions that could be called when needed. Show and Read derivations were then added to GameData, Room, Exit and Objects. This allowed for the (String, Room) tuples to be easily parsed when reading the save file.

Loading leaving a Semi-open Handle when Reading a File and Issues when forcefully closing handles using hClose.

When loading in files, it could leave a semi-open handle that meant it could not be written to later.

The first approach taken to tackle this issue by causing an early closure of the handle meant the SaveData was not read in fully so it could not be converted to GameData.

The second, successful approach was to use putStr to read the file content fully before closing the handle. However, as this printed unnecessary output, a forceRead function was made that would read the String without outputting it. This meant the necessary information could be read before closing the handle.

Having issues with type coercion between the already implemented code and haskeline Implementation

When implementing the haskleine user input the main program broke and to tackle this issue, many of the IO functions had to be converted to their InputT IO counterparts such as putStrLn to outputStrLn.

Having issues adding a load/save command using the function.

This issue arose from the idea that haskeline uses the InputT IO() type while most of the interactive program is written in IO() and InputT IO is not a sub-type of IO. Therefore, the save/load functions would not work in haskeline.

To tackle the named issues the Control library had to be added to the program and the specific package MonadIO was used to aid this issue where if any function is IO type the liftIO function converts it into inputT IO type.

Testing

The following test scenarios were used to ensure our game was robust and consistent.

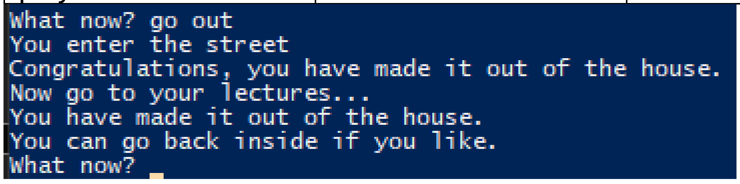
We used a mix of manual and QuickCheck-based tests as some cases were easier to test manually as they did not map easily into a property test.

Evidence for QuickCheck based tests can be found by running the relevant test method.

Test	Test Method	Expected Outcome	Actual Outcome
Manual Tests			
Checking the game can be completed	Running the minimum commands to beat the game (can be found in cheatsheet.txt)	The game is beaten	The correct win condition message is output (see Fig. 2)
Checking only valid commands are accepted	Attempting to run non-defined commands	The invalid commands are not accepted.	Appropriate errors are returned (see Fig. 3)
Checking that doors/objects can only be opened when the required condition is met.	Attempting to open each locked object without their requirements. (As the valid case was seen in test 1).	An appropriate error is returned.	An appropriate error is returned (see Fig. 4)
Checking that the statue can only be fixed when the	Attempting to fix the statue without the hand. (As the valid	An appropriate error is returned.	An appropriate error is returned (see Fig. 5)

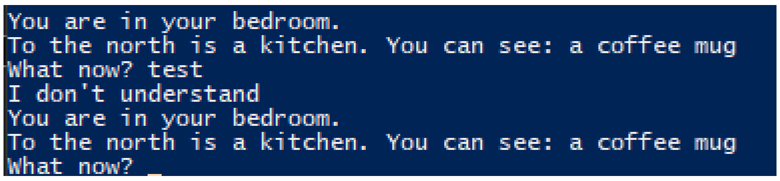
required condition is met.	case was seen in test 1).		
Checking that error handling on invalid commands works correctly.	Attempting to run targeted commands with invalid targets.	The invalid commands are rejected.	Appropriate errors are returned (see Fig. 6)
Checking that examining the monster ends the game.	Examining the monster.	The program ends upon examining.	The program exits with the appropriate message (see Fig. 7)
QuickCheck Based Tests			
Checking if the base case for move works correctly (nothing)	Attempting to move south from the bedroom	The test returning nothing	The test passed
No room has two exits in the same direction	Comparing exits in each direction from a given room to ensure there are not two in one direction.	Movement from two different rooms with the same direction does not end in the same place	The test passed
Checking that locked rooms cannot be moved into	Attempting to go through a locked door	When attempting to go through locked doors the current room does not change	The test passed
Checking that valid movements can be made	Attempting to move in different directions and comparing the state before and after the move	Valid moves update the state	The test passed
Checking that heavy objects cannot be moved	Checking an appropriate failure message is returned when trying to pick up heavy messages and that the state does not change	When trying to get objects that are heavy the prompt string is that object is too heavy to pick up while the state is kept the same	The test passed
Checking that saving to a file works	Saving a state to a temporary file then reads the save and compares it against a stringified version of the state	The temporary file is the same as the save	The test passed
Checking that reading a save file works (based on the assumption that save file works)	Saving a state to a file then reads it using readSavePoint function and compares that against the state that was sent to be saved	The read state and the sent state are the same	The test Passed
Checking room data changes after adding an object to it	Checking if the room is the same as before an object was added to it	The room data before and after the change is different	The test Passed

Checking room data changing after removing an object from it	Check if the room is the same as before an object was removed from it	The room data before and after the change is different	The test Passed
Checking that addObject and removeObject result in different room data	Check if the room is equal for an add case and a remove case	The room data for the two cases are different	The test Passed
Checking that the get method adds to the inventory of the player	Check if the inventory after the method has been called is different to before	The inventory has been updated after the method runs	The test Passed
Checking that the put method removes the object from the inventory of the player	Check if the inventory after the method has been called is different to before	The inventory has been updated after the method runs	The test Passed



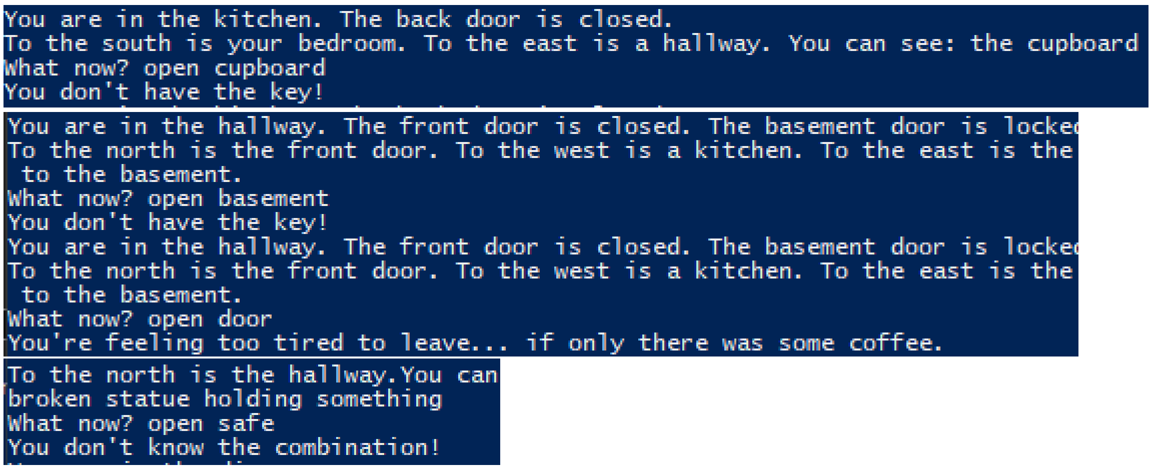
```
What now? go out
You enter the street
Congratulations, you have made it out of the house.
Now go to your lectures...
You have made it out of the house.
You can go back inside if you like.
What now?
```

Fig. 2: The win message after beating the game



```
You are in your bedroom.
To the north is a kitchen. You can see: a coffee mug
What now? test
I don't understand
You are in your bedroom.
To the north is a kitchen. You can see: a coffee mug
What now?
```

Fig. 3: The error message after inputting an invalid command

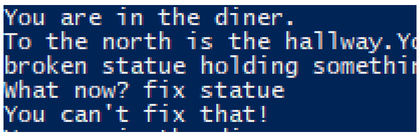


```
You are in the kitchen. The back door is closed.
To the south is your bedroom. To the east is a hallway. You can see: the cupboard
What now? open cupboard
You don't have the key!
```

```
You are in the hallway. The front door is closed. The basement door is locke
To the north is the front door. To the west is a kitchen. To the east is the
 to the basement.
What now? open basement
You don't have the key!
You are in the hallway. The front door is closed. The basement door is locke
To the north is the front door. To the west is a kitchen. To the east is the
 to the basement.
What now? open door
You're feeling too tired to leave... if only there was some coffee.
```

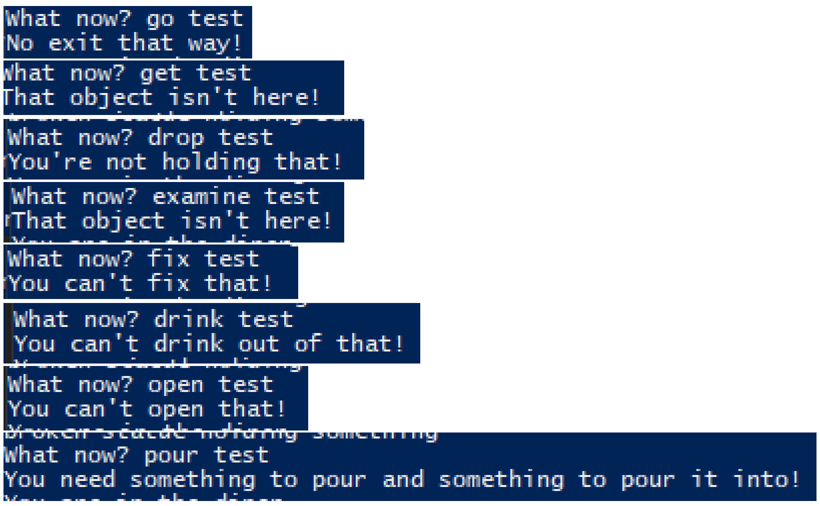
```
To the north is the hallway.You can
broken statue holding something
What now? open safe
You don't know the combination!
```

Fig. 4: The error messages after not meeting the conditions to open locked objects



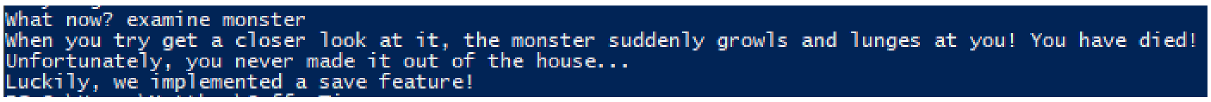
```
You are in the diner.
To the north is the hallway.Yo
broken statue holding somethi
What now? fix statue
You can't fix that!
```

Fig. 5: The error message after not meeting the condition to repair the statue



```
What now? go test
No exit that way!
What now? get test
That object isn't here!
What now? drop test
You're not holding that!
What now? examine test
That object isn't here!
What now? fix test
You can't fix that!
What now? drink test
You can't drink out of that!
What now? open test
You can't open that!
What now? pour test
You need something to pour and something to pour it into!
```

Fig. 6: The error messages for actions with invalid targets



```
What now? examine monster
When you try get a closer look at it, the monster suddenly growls and lunges at you! You have died!
Unfortunately, you never made it out of the house...
Luckily, we implemented a save feature!
```

Fig. 7: The lose message after examining the monster

Summary of Provenance

The following code was written directly by us:

- The directions and fix methods in Actions.hs
- The whole SaveRW.hs file
- The whole Tests.hs file

The following code was built upon the provided source files:

- The functions provided in the Actions.hs file were implemented and expanded upon
- The main and process functions in the Adventure.hs file were expanded upon
- The World.hs file was modified and expanded.

The following code was built upon or inspired by externally sourced files.

- The main and loop functions in Adventure.hs were implemented and expanded upon the code found at [1]
- The error handling code was inspired by the code in the question found at [2]
- The saveRead and SaveWrite property tests were inspired by the top two answers provided at [3]
- The implementation of quickCheck all was inspired by the answers found at [4]
- Implementation of the generators for data-types was inspired by the code found at [5]

References

[1] Haskell.org. (2022). System.Console.Haskeline. [online] Available at: https://hackage.haskell.org/package/haskeline-0.8.2/docs/System-Console-Haskeline.html#g:1 [Accessed 5 Feb. 2022]

[2] G-J (2014). Exception handling for “readFile.” [online] Stack Overflow. Available at: https://stackoverflow.com/questions/21208771/exception-handling-for-readfile [Accessed 8 Feb. 2022]

[3] Thurn, D. (2011). Testing functions in Haskell that do IO. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/7370073/testing-functions-in-haskell-that-do-io [Accessed 8 Feb. 2022].

[4] Guillaume Chérel (2017). Complete minimal example for using quickCheckAll. [online] Stack Overflow. Available at: https://stackoverflow.com/questions/42669087/complete-minimal-example-for-using-quickcheckall [Accessed 9 Feb. 2022].

[5] Vasconcelos, P. (2021). Property Testing using QuickCheck. [online] Fc.up.pt. Available at: https://www.dcc.fc.up.pt/~pbv/aulas/tapf/handouts/quickcheck.html [Accessed 9 Feb. 2022].