

CS2006 Haskell 1 Group Report

Tutor: Edwin Brady

210013988, 220019716, 220018336

05-02-2024

Overview

The aim of this practical was to create a game for the command-line based on the requirements set out by the practical specification [1], which are listed below (some are listed exactly as written in the specification, others are paraphrased for brevity).

Basic Requirements

- Introduce custom data types to replace *String*. (Completed)
- Implement movement. (Completed)
- Implement "get" and "drop" commands. (Completed)
- Implement specific commands for pouring and drinking coffee, and opening the door. (Completed)
- Implement commands to help the user, to allow them to examine objects and list their inventory. (Completed)

"Easy" Requirements

- Add new rooms and objects. (Completed)
- Introduce new puzzles. (Completed)
- Set up a *cabal* file to describe how to build the program. (Completed)

"Medium" Requirements

- Refactor to include higher order functions. (Completed)
- Use *QuickCheck* to write property based tests for your functions. (Completed)

"Hard" Requirements

- Use *Parsing.hs* as opposed to the *words* command for parsing user input. (Completed)
- Extend the parser to understand longer phrases. (NOT Completed)
- Implement save and load functionality. (Completed)
- Use the *Haskeline* library for reading user input. (Completed)

"Very Hard" Requirements

- Use the *State* type to represent game state. (Completed)

Design

This program is split into three sections, the library (consisting of *Actions.hs* and *World.hs*), the executable (*Adventure.hs*), and the test suite.

The code for the library is contained within the `src` directory, the test suite is within the `test` directory, and the files related to the executable are stored in the `app` directory. These sections are all contained within a parent `Code` directory.

The *World.hs* file is where all of the types and data structures necessary for the game are defined. This includes the `WorldObject`, `Room`, `Exit`, `Direction`, `Argument` and `GameData` types, and the `Action`, `Command` and `ReturnValue` type aliases. `WorldObject` is a record type that describes the properties of the objects that can be found in the game world, and is a modified version of the original `Object` type (the name change was necessary due to a constructor name conflict). The `Action` type takes an `Argument` (which is either the `Direction` or `WorldObject` that is required for the action to be completed) and the game state.

We considered defining `Action` as an algebraic data type that would consist of both the name of an `Action`, and an associated `Direction` or `WorldObject` afterwards. This approach would allow us to pattern match in the functions themselves and would ensure that the argument was of the correct type. The big issue with this approach, however, is that we still need to convert the user's input from a *String* to an instance of this type (as well as performing input validation also). This would effectively create a massive amount of duplicate code and overly complicate our design. Instead, the `Argument` type was defined as a compromise. We still take the user's *String* and match it to find the associated function as before; however, we will now check their argument separately and use the `Maybe` Monad in order to handle any errors. This guarantees that any value stored in an instance of the `Argument` type is valid instance of that type (as it must match to a predefined `WorldObject` or `Direction`), but also allows us to pattern match in the `Action` functions themselves to ensure that the `Argument` instance contains a value of the correct type (as `WorldObjects` are wrapped with the *ObjArg* constructor and `Directions` are wrapped with the *DirArg* constructor).

The *Adventure.hs* file is where the code for the game loop is located. The "main" function is the entry point for the program and makes a single call to "repl", which will recursively call itself and make calls to "process" to interpret the user's input until the player reaches the *Street*, has their *laptop* with them, and has drank their coffee, at which point they have "won" and the game is exited.

The *Actions.hs* file is where all of the critical functions that modify game state exist. All *Commands* and *Actions* are defined here. Saving and loading was not considered to be a `Command` as the way that this was parsed was slightly different to the other commands. Also, since this was dealing with `IO` it would mean that the *commands* function to return the appropriate command based on the user's input would have to support `IO` also, which would clutter the code due to being completely unnecessary for all other aspects of the program.

While the player can drink the *beer* as opposed to the coffee, it should be noted that as long as the *coffeepot* and *mug* are in the player's inventory they will be able to fill the mug again to drink the coffee and undo the effects of the *beer*.

The player starts in the "bedroom" and must complete the following tasks in approximately this order:

1. The player must navigate to the "lounge" with the "go west" action in order to turn on the lights with the "press" command.
2. The player must pick up their laptop from the "lounge" with the "get laptop" action.
3. The player must navigate back to the "bedroom" with the "go east" action.
4. The player must pick up their mug with the "get mug" action while in the "bedroom", and then navigate to the "bathroom" with the "go east" action.

5. The player must take a shower with the “shower” command and then navigate to the “kitchen” via the “bedroom” with the “go west” action followed by the “go north” action.
6. The player must pick up the pot of coffee with the “get coffeepot” action, pour the coffee with the “pour” command, and drink the coffee with the “drink mug” action.
7. The player must navigate to the hall with the “go west” action and, assuming they have already consumed the coffee and have their “laptop” with them, open their door with the “open” command and exit the house with the “go out” action.
8. The player is now in the “street” and has won the game!

Provenance

Libraries Used

Library	Purpose	Author	Hackage Link
Haskeline	Provides greater control over user interface. Satisfies requirement.	Judash Jacobson	Link to Package
Aeson	Used for parsing objects to and from JSON for save and load requirement.	Bryan O'Sullivan	Link to Package
QuickCheck	Property-based testing framework for Haskell. Satisfies requirement.	Koen Claessen	Link to Package
MTL	Monad classes and transformers, used for State implementation.	Andy Gill	Link to Package

Files

README.md	Created by us
cabal.project.local	Created by us
defaultfile.json	Created by us
haskell-p1.cabal	Created by us
app/	
Adventure.hs	Adapted from provided Code
Parsing.hs	Unmodified from provided Code
src/	
Actions.hs	Adapted from provided Code
World.hs	Adapted from provided Code
test/	
ActionsTest.hs	Created by us
Test.hs	Created by us
WorldTest.hs	Created by us

Testing

To test the program we used *QuickCheck* and also performed manual runs of the program to test for any bugs and gauge the ease of use of our implementation.

What is being tested	Pre-conditions	Expected Outcome	Actual Outcome
<i>actions</i> function with known inputs	Known Action strings provided	Function returns <i>Just value</i> for action	As Expected
<i>actions</i> function with unknown inputs	Arbitrary strings not in known actions	Function returns <i>Nothing</i>	As Expected
<i>commands</i> function with known inputs	Known Command strings provided	Function returns <i>Just value</i> for command	As Expected
<i>commands</i> function with unknown inputs	Arbitrary strings not in known commands	Function returns <i>Nothing</i>	As Expected
<i>arguments</i> function with known inputs	Known Argument strings provided	Function returns <i>Just value</i> for argument	As Expected
<i>arguments</i> function with unknown inputs	Arbitrary strings not in known arguments	Function returns <i>Nothing</i>	As Expected
<i>move</i> function with invalid direction	Arbitrary Room and invalid direction	Function returns <i>Nothing</i>	As Expected
<i>objectHere</i> function for present object	Room and object that is in the Room	Function returns <i>True</i>	As Expected
<i>objectHere</i> function for absent object	Room and object not in the Room	Function returns <i>False</i>	As Expected
<i>addObject</i> function	Room and WorldObject to add	Object is in the Room 's object list after addition	As Expected
<i>removeObject</i> function	Room and WorldObject to remove	Object is not in the Room 's object list after removal	As Expected
<i>won</i> function with player in <i>Street</i> and has "laptop"	GameData with player in <i>Street</i> and has "laptop"	Function returns <i>True</i>	As Expected
<i>won</i> function with player not in <i>Street</i>	GameData with player not in the <i>Street</i>	Function returns <i>False</i>	As Expected

```

prage/OneDrive-UniversityofStAndrews/Documents/St Andrews/Modules/CS2006/Coursework/Haskell
World.hs Adventure.hs Actions.hs
app > Adventure.hs
1 module Main where
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
cabal-3.10.2.1
> /Users/tom/.ghcup/bin/cabal-3.10.2.1 run
The light is off so you cannot see any exits or objects.
You are in your bedroom.
> What now? help
----- Haskell-P1 -----
ACTIONS:
go [Direction] (Move in the specified direction, if applicable)
get [Object] (Pick up an object)
put [Object] (Put down an object)
examine [Object] (Examine an object)
drink [Object] (Consume a drinkable object)
COMMANDS:
pour (Pour coffee from the "coffeepot" into the "mug")
open (Open the door in the Hall leading to the Street)
press (Press the light switch in the Lounge to turn on the lights)
inventory (Get a list of all objects in your inventory)
help (Print this message)
quit (Exit the game)
OBJECTS:
mug (In the Bedroom)
laptop (In the Lounge)
coffeepot (In the Kitchen)
beer (In the Kitchen)
DIRECTIONS:
north
east
south
west
in
out
TASKS:
1) Turn on the lights
2) Obtain your "laptop"
3) Obtain your "mug"
4) Go for a shower
5) Obtain the "coffeepot"
6) Pour the coffee from the "coffeepot" into the "mug"
7) Drink the coffee
8) Open the door in the Hall to the Street
9) Exit to the Street and go to your lectures!
OPTIONAL) Drink the beer... are you sure that's a good idea before lectures?
GAME MAP:
-----
Ln 113, Col 72 Spaces: 4 UTF-8 LF Haskell Prettier

```

Figure 1: Example run (Part 1)

```

prage/OneDrive-UniversityofStAndrews/Documents/St Andrews/Modules/CS2006/Coursework/Haskell
World.hs Adventure.hs Actions.hs
app > Adventure.hs
1 module Main where
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS
zsh
GAME MAP:
-----
| Street |
^
| (out) |
^
| Hall | <--> | Kitchen |
^
| Lounge | <--> | Bedroom | <--> | Bathroom |
START
The light is off so you cannot see any exits or objects.
You are in your bedroom.
> What now? go west
--- OK ---
The light is off so you cannot see any exits or objects.
You are in the lounge. The light switch is off. (Use the "press" command to turn on the lights)
> What now? press
--- Light is switched on. ---
You are in the lounge. The light switch is on
To the east is a bedroom.
You can see: a Laptop ("laptop")
> What now? get laptop
--- Item picked up successfully ---
You are in the lounge. The light switch is on
To the east is a bedroom.
> What now? go east
--- OK ---
You are in your bedroom.
To the north is a kitchen. To the west is the lounge. To the east is the bathroom.
You can see: a coffee mug ("mug")
> What now? get mug
-----
Ln 113, Col 72 Spaces: 4 UTF-8 LF Haskell Prettier

```

Figure 2: Example run (Part 2)

The screenshot shows a Haskell REPL session with the file `Adventure.hs` open. The code defines a game world with rooms and items. The REPL output shows the following sequence of commands and responses:

```
You can see: a coffee mug ("mug")
> What now? get mug
--- Item picked up successfully ---

You are in your bedroom.
To the north is a kitchen. To the west is the lounge. To the east is the bathroom.

> What now? go east
--- OK ---

You are in the Bathroom. You have not showered today
To the west is your bedroom.

> What now? shower
--- You took a shower ---

You are in the bathroom. You have showered.
To the west is your bedroom.

> What now? go west
--- OK ---

You are in your bedroom.
To the north is a kitchen. To the west is the lounge. To the east is the bathroom.

> What now? go north
--- OK ---

You are in the kitchen.
To the south is your bedroom. To the west is a hallway.

You can see: a pot of coffee ("coffeepot"), a bottle of beer ("beer")

> What now? get coffeepot
--- Item picked up successfully ---

You are in the kitchen.
To the south is your bedroom. To the west is a hallway.

You can see: a bottle of beer ("beer")

> What now? get beer
--- Item picked up successfully ---

You are in the kitchen.
To the south is your bedroom. To the west is a hallway.

> What now? drink beer
--- Beer has been drunk and you are now intoxicated. You must have a coffee again to sober up ---
```

Figure 3: Example run (Part 3)

The screenshot shows the continuation of the Haskell REPL session. The REPL output continues with the following sequence of commands and responses:

```
> What now? get beer
--- Item picked up successfully ---

You are in the kitchen.
To the south is your bedroom. To the west is a hallway.

> What now? drink beer
--- Beer has been drunk and you are now intoxicated. You must have a coffee again to sober up ---

You are intoxicated, Drink a coffee to sober up.
You are in the kitchen.
To the south is your bedroom. To the west is a hallway.

> What now? pour
--- Coffee mug is now full and ready to drink ---

You are intoxicated, Drink a coffee to sober up.
You are in the kitchen.
To the south is your bedroom. To the west is a hallway.

> What now? drink mug
--- Coffee has been drunk and you are now caffeinated ---

You are in the kitchen.
To the south is your bedroom. To the west is a hallway.

> What now? go east
--- No room in that direction. ---

You are in the kitchen.
To the south is your bedroom. To the west is a hallway.

> What now? go west
--- OK ---

You are in the hallway. The front door is closed.
To the east is a kitchen.

> What now? open
--- Door has been opened to the street! ---

You are in the hallway. The front door is open.
To the east is a kitchen. You can go outside.

> What now? go out
--- OK ---
Congratulations, you have made it out of the house.
Now go to your lectures...
```

The status bar at the bottom indicates the file path, editor (VS Code), and execution time: `took 1m 25s at 04:07:30 PM`.

Figure 4: Example run (Part 4)

```

> /Users/tom/.ghcup/bin/cabal-3.10.2.1 v2-test
Build profile: -w ghc-9.8.1 -O1
In order, the following will be built (use -v for more details):
- Haskell-P1-0.1.0.0 (test:haskell-p1-test) (first run)
Preprocessing test suite 'haskell-p1-test' for Haskell-P1-0.1.0.0..
Building test suite 'haskell-p1-test' for Haskell-P1-0.1.0.0..
Running 1 test suites...
Test suite haskell-p1-test: RUNNING...
Test suite haskell-p1-test: PASS
Test suite logged to:
/Users/tom/Library/CloudStorage/OneDrive-UniversityofStAndrews/Documents/St
Andrews/Modules/CS2006/Coursework/Haskell-P1/CS2006-H1/Code/dist-newstyle/build/aarch64-osx/ghc-9.8.1/Haskell-P1-0.1.0.0/t/haskell-p1-test/test/Haskell-P1-0.1.0.0-haskell-p1-test.log
1 of 1 test suites (1 of 1 test cases) passed.

```

Figure 5: *QuickCheck* Results

Evaluation

Regardless of the number of requirements met, we do recognise that some areas of our submission could be improved.

We did not manage to extend the parsing library, which we believe was the only requirement that we did not complete; however, we are very happy with the fact that we managed to meet all others.

We also recognise that the user interface leaves a lot to be desired. The “>” character before the prompt to the user was supposed to highlight the line at which the user was entering text into the prompt, the “—” pattern before a message indicates success from the action and the “-” character indicates a failure, but with so much text on the screen and so much repeated information it can be hard for the player to keep track of their progress.

There is some dissonance between the player’s objectives and the things they need to type to achieve these objectives. While some instances of this are rather humorous (i.e. the *pour* command pours the coffee from the coffeepot into the user’s mug, but then the *drink mug* command is needed in order for the user to consume the coffee within the mug), other cases can make the game confusing at times. For example, just before the end of the game the player is told that they can now “go outside”, but the command the player must enter is “go out”. This is a small issue, but could still be irritating for the player. When the player starts the game (in the “bedroom”) the lights are off, meaning that they cannot see any objects or any exits to go through. The main problem here is that they have to know to go to the “lounge”, know how to reach the “lounge” (to the east; either by chance or already knowing this beforehand), *and* that the “press” command must be used in the “lounge” in order to turn on the lights. Since we were not able to extend the parsing library in time for the submission deadline, this means that commands and actions were limited to single-word names only. The ability to say “turn on the lights” or “drink coffee” would have been a game-changer, but instead the *help* command was introduced at the end as a slapdash substitute to the advanced parsing. Despite being primitive in design, the *help* command does improve the playability of the game, serving its purpose well and somewhat mitigating the other issues posed by the UI’s syntactic limitations.

There are a couple of instances of “head” and “tail”, although these appear in circumstances where it has already been determined that the lists they will be operating on will not be empty, and will therefore not crash the program. We decided that, while perhaps bad practice, it was better for readability that “head” and “tail” were kept in. Given more time, we would seek to refactor these parts of the code to remove occurrences of these functions, but relative to the other tasks we had this was considered relatively low priority.

We did our best to include higher-order functions, but while there are instances of *map* and *filter* scattered fairly frequently throughout the program, *foldr* does not appear often. Many cases in which we considered using folds would have been at the cost of readability, and therefore we decided that it would best not to follow through with the refactors; however, we have still considered the medium requirement satisfied as there is some inclusion of these higher-order functions, as requested.

Notwithstanding the game’s flaws, we are very happy overall with the outcome of the project and feel that this was a valuable learning experience, not just for improving our abilities with Haskell, but also for learning how to be effective contributors and communicators in our team.

References

- [1] *H1.html*, University of St Andrews School of Computer Science, 2024.
<https://studres.cs.st-andrews.ac.uk/CS2006/Coursework/H1/H1.html>