Group D

Text Adventure

Written in Haskell

090010514
11/17/2010

Table of Contents

Haskell: Text Adventure

This project implements a text adventure game in Haskell. Haskell is a polymorpgically statically typed, lazy, purely functional programming language, and in many ways quite different from most others[i]. The player starts in their room and the goal is to leave the house, but before they can leave their house they have to drink some coffee.

Features

The solution described below features all the basic requirements:

- Implementation of movement, get, drop, pouring, drinking and open commands, which all involve updating the game state.
- Implementation of examine and inventory commands which do not update the game state.

Additionally it offers:

- Additional rooms, objects and puzzles.
- The majority of function definitions which could have been written recursively have been written using filter or list comprehensions.
- Save and load functionality.
- Directions and Objects represented as data types.
- A more advanced parser which accepts multiple commands using the keyword " then ".

Personal Participation

My participation to coming up with a solution to this project included:

- Implementation of methods in Actions.hs for the basic requirement.
- Replacing recursive methods with more efficient alternatives, such as filter and list comprehensions.
- Implementing and integrating Direction and Object data types.
- Advanced parser.

Sample Output

```
You are in your bedroom.
To the north is a kitchen.

You can see: a coffee mug
What now? get mug then go north
You go to the kitchen.
You are in the kitchen.
To the south is your bedroom. To the west is a hallway.

You can see: a pot of coffee
What now? get coffee then pour coffee
You pour some coffee.
You are in the kitchen.
To the south is your bedroom. To the west is a hallway.
What now? examine fullmug
A coffee mug containing freshly brewed coffee
You are in the kitchen.
To the south is your bedroom. To the west is a hallway.
What now? go west then open door
It's too heavy!
You are in the hallway. The front door is closed.
To the east is a kitchen.
What now? inventory
You are carrying:
a full coffee mug
a pot of coffee
You are in the hallway. The front door is closed.
To the east is a kitchen.
What now? drink coffee
You drink the coffee.
You are in the hallway. The front door is closed.
To the east is a kitchen.
What now? inventory
You are carrying:
a coffee mug
a pot of coffee
You are in the hallway. The front door is closed.
To the east is a kitchen.
What now? open door
You open the door.
You are in the hallway. The front door is open.
To the east is a kitchen. To the west is the front door.
What now? go west
You go to the street.
Congratulations, you have made it out of the house.
Now go to your lectures...
```

As seen above the basic functionality exists and the parser supports multiple commands.

Design

The design is mostly unchanged from the files given to us on StudRes. The solution consists of three modules, Actions, World and Adventure.

Module	Purpose
World	Description of the data types used and instances of them in the game world.
Actions	Contains methods which process commands, some of which change the game state and some of which do not.
Adventure	Runner module which parses commands and passes them to Actions where they are processed.

World

Types defined in World are Actions and Commands. Actions are command which take at least one argument, and tend to update the state (examine does not) and Commands to not take arguments. Examples of Commands are “quit” and “inventory.” They are used as types for functions of these types, i.e. Command is used as the type for inventory.

Data constructors

Data constructors are Object, Exit, Room and GameData.

Object has three fields (all Strings): obj_name, obj_longname and obj_desc. obj_name which should be passed into commands entered by the user, such as “examine”, “get” and “drop”. obj_longname is used in room descriptions and in listing the inventory and obj_desc is displayed when examining an object.

Exit also has three fields: exit_dir (Direction), exit_desc (String) and room (String). exit_dir is the direction in the player has to go in order to go through this exit while in a room which contains it. exit_desc is a description of the exit, for example "To the west is a hallway. " room is just a single string identifier for the room to which an exit leads.

Direction has four different possible values: North, West, South and East, and is used to represent directions in which the player can move in the game.

Having Objects and Directions defined as data types makes it possible to implement a “command”-type of the format below in the future:

```
data Command = Go Direction | Get Object | Drop Object | Pour Object | Drink Object | Quit | Inventory | ...
```

Implementation

File Handling: Saving and Loading

Whereas all other commands are processed in processCommand, "save" and "load" have to be checked for in Adventure's repl function, because they are of the IO type which have to be called from an IO context. Saving and loading is currently done using quite complex algorithms which have been defined manually. An alternative way of doing this, which would have required much less effort and would look quite significantly neater would be to have all the data types derive show and read and save and load data in single statements.

Save would then become something like:

```
save :: GameData -> String -> IO ()
save state filename = writeFile filename (show state)
```

And Load becomes something like:

```
load :: String -> IO GameData
load filename = do data <- readFile filename
                   return (read data :: GameData)
```

Recursion vs Alternatives

For a medium difficulty requirement we were required to rewrite the functions which we had previously defined recursively using filter, map, list comprehensions or foldr. The main reason for why recursive methods are not as efficient as the alternatives is that recursion requires function calls to be stored in a stack which increases in size with the number of elements in the list on which the function is called. Recursive methods are often regarded as more elegant than alternative implementations because they are usually shorter and simpler[ii].

There are certain cases where recursive functions are still in use and the reason for this is that it is not desirable to iterate through an entire list if the program is only looking for a single value in it. An example of this is seen in findObj:

```
findObj :: String -> [Object] -> Object
findObj obj (d:ds)
        | obj == obj_name d = d
        | otherwise = findObj obj ds
```

Testing

Actions

The Actions module is essential for the game to work because it does the final processing of commands. The pour function is a good example of a function which passes certain tests, while it fails others. Similarly, drink and examine do not check if the argument is the correct one.

Pour-function

Works when given the right argument:

```
What now? pour coffee
You pour some coffee.
You are in the kitchen.
To the south is your bedroom. To the west is a hallway.
```

It also works for any other non-empty argument, because it does no checking of the value of the argument:

```
What now? pour .
You pour some coffee.
You are in the kitchen.
To the south is your bedroom. To the west is a hallway.
What now?
```

If no argument is passed in, it will not work, but does not cause the problem to terminate.

```
What now? pour
I don't understand
You are in the kitchen.
To the south is your bedroom. To the west is a hallway.
```

If the mug is already full an error message is printed:

```
What now? pour coffee
Your mug is already full!
You are in the kitchen.
To the south is your bedroom. To the west is a hallway.
```

A testing interface which could be used to do unit tests is HUnit, which is inspired by JUnit in Java[iii].

Limitations

Limitations of the solution include:

- Only having a single save file at the time (unless it is moved or renamed manually)
- Small game world, and therefore a very limited adventure
- No checking of argument passed into drink, pour or open functions, thus “open .”, has the same effect as “open door”
- Objects’ obj_name currently have to be unique for the save/load functionality to work.
- It is not always easy to guess what things are called (for example to examine a mug full of coffee, the user has to enter “examine fullmug”). If the alternative implementation of load/save was implemented, this problem could have been avoided since the Objects’ obj_name do not have to be unique.

Potential Changes and Additions

- Extend game world with new rooms, objects and puzzles (for instance turning on the light)
- More interactive objects (can be changed) such as apple trees (pick apples) and other plants, light switches (turn on/off light), and computers (maybe solve some sort of puzzle?).
- Command data type as outlined in design section.
- Implement alternative save/load function outlined as described in Implementation section.
- User-interface which is easier to read.
- NPCs (Non Player Characters) with which the player can interact.
- Threading -> More dynamic world. For instance, the user could be sinking in sand and have to have to hold onto a tree in order not to drown in the sand. Threading would be required in this instance because one thread would be accepting keyboard input and another would time how long it takes the player to react, and thus how deep they sink into the sand.
- Rewriting recursions functions using foldr.
- Coloured text. Could make certain elements expressed more clearly.
- Help command for inventory, examine, quit, etc
- Have a hint command which gives the user a hint on how to advance in the game.
- Get backspace to work in the terminal
- Implement saving and loading by deriving show and read on all types instead of using the current overcomplicated implementation. This would mean that the gameworld and inventory would have to be displayed differently, but that is not a huge problem compared to the complexity of the current saving and loading algorithms.

Conclusion

In this project we implemented a text adventure game in Haskell. Features which were implemented include a more advanced parser, direction and object types and the majority of Action functions written without use of recursion.

This project has given me significantly more experience using Haskell, and has given me a better understanding of many fundamental features such as list comprehensions, monads, data-, type- and instance-keywords. Much to my frustration when working with the language often arose only because of indentation in the context of monads and I find that the Haskell compiler's error messages are rarely accurate. However, I find that the indentation issues are usually solved by using mostly tabs and then at most three spaces at the beginning of the line, because replacing tabs with spaces does not always help.

While working as a team has its advantages in terms of being able to discuss ideas with fellow group members, there were certain issues. One member of our group gave us the impression that he was willing to put in the effort required, and was assigned tasks, but these were never completed despite countless attempts at communicating with the member. We wanted to give him something easy, because he did not seem as confident as the rest of us, and therefore his lack of participation is the reason the game world was never extended.

---

[i] http://www.haskell.org/haskellwiki/Introduction, Accessed: 17/11/10
[ii] https://studres.cs.st-andrews.ac.uk/CS2001-FC/Lectures/L22-Recursion/Lecture22.pdf, Accessed: 12/11/10
[iii] http://hunit.sourceforge.net/HUnit-1.0/Guide.html, Accessed: 12/11/10