Abduction

A Text-Based Adventure Game

Group Members: Alice Yu, Deekshita Chigullapally

Motivation

When learning how to program for the first time, the most common languages used to teach the ideas of programming are imperative languages. These languages allow programmers to easily turn pseudocode into actual code. With imperative languages, the process of turning pseudocode into actual code is relatively simple. However, when it comes to programming in functional languages, one cannot use that same thought process. In order to learn how to convert a program that is written in an imperative language into a program that is written in a functional language, we decided to go with a fairly simple, commonplace introductory imperative programming project. Therefore, our project idea was to create a text-based adventure game, similar to the games common in the 1980's, like Zork. We chose a text-based adventure game because it would be fun to create and it uses a lot of properties of imperative programming languages which we are familiar with. We also wanted to see how these properties would change when programming in a functional language.

Project Overview

Our text-based adventure game takes the player on a journey as they are captured by aliens and must figure out how to get home before their health runs out. The player must type in commands on the command prompt in order to move around and commit actions throughout the game. All possible commands are shown by the capitalized phrases in the storyline. The player starts off with 100 health points (HP) and zero potions. Each time the player makes a move or commits an action they lose 10 HP, whereas each time the player drinks a potion they gain 30 HP. The potions are hidden in secret locations throughout the game, with one potion hidden on each level. Once a player reaches level 3, the final level, and encounters actual aliens, they have the chance of getting seriously injured. A serious injury to the player results in a loss of 20 HP. There are two ways in which the game will end: you either run out of HP (which results in a loss), or you complete the game and return home safely (which results in a win).

This text based adventure game, Abduction, was implemented in two different languages: Java and Haskell. We implemented it in Java first, and then proceeded to implement it again in Haskell. The game involves a player object, which contain the properties *Name*, *Health Points*, and *Potions*. The HP of the Player object is decremented when the player makes a move or commits an action, and is incremented when the Player takes a potion. Player information can be accessed using the special STATUS command, which reports the amount of HP and potions that the player currently has. In addition, invoking the STATUS command allows the player to drink a potion if they are in possession of at least one.

The code for this project can be found at:
https://github.com/aljyu/cmps-112

Languages Used

The languages that were used in this project were Java and Haskell.

Java

Java is an object-oriented, imperative programming language, a language that was designed to follow *imperative* (aka *procedural*) programming. Imperative programming is about telling the computer ***how*** to do something. With an imperative approach, a programmer would write code that would detail exactly what steps the computer must take to accomplish a goal. With imperative languages such as Java, the primary flow control stems from loops, conditionals, and method calls. Imperative programming languages support both mutable and immutable objects, meaning that the programmer gets to decide if he or she wants an object's state to be modified after its creation. With Java, the programmer can easily change an object's state after it has been created at any time he or she wishes.

Haskell

Haskell is a purely functional programming language, a language that was designed to follow *declarative* programming. Declarative programming is about telling the computer ***what*** needs to happen before being able to do something. With a functional approach, a programmer would decompose the problem into a set of functions that would be executed to accomplish a goal. From there, the programmer would have to define the inputs and result of each function. In functional programming languages there are no side-effects since the only thing that a function can do is calculate something and return it as a result. With functional languages such as Haskell, the primary flow control stems from function calls that may include recursion. Functional programming languages support only immutable objects, meaning that an object's state cannot be modified after its creation. With Haskell, the programmer cannot change an object's state after it has been created no matter what. Therefore, in order to change the object's state, a new instance of the object containing the desired changes would have to be created.

Challenges

The first challenge that we faced during the span of the project was immutability. As described in the section above, Haskell is a purely functional programming language and therefore does not allow for mutability. Every object in Haskell is immutable and therefore a new instance of an object must be created whenever changes need to be made to a pre-existing object. This was a huge challenge because of the fact that we are both used to programming in imperative programming languages, which allow for both mutability and immutability. When we first started the Haskell implementation portion of the project, we did not know that Haskell did not allow for mutability. We kept attempting to change the objects' states and therefore ran into the same errors over and over again. However, after researching and debugging the reason behind the immutability errors, we realized that it was because of the immutability property of functional programming languages. From that point on, we experimented and researched if there was any way we could imitate object mutability. After countless failed attempts, we admitted defeat and proceeded to work with Haskell's immutability property. Once we discovered the fact that we had to create new instances of an object whenever we wanted to change its pre-existing state, the rest was straightforward and easy to implement.

Immutable Player in the Haskell program

```haskell
module Player
(
  Player(..)
  ,move
  ,extra
  ,hit
  ,drink
) where

data Player = Player{ hp :: Int, potions :: Int} deriving (Show,Eq)

move :: Player -> Player
move hero = Player ((hp hero) - 10) (potions hero)

extra :: Player -> Player
extra hero = Player (hp hero) ((potions hero) + 1)

hit :: Player -> Player
hit hero = Player ((hp hero) - 10) (potions hero)

drink :: Player -> Player
drink hero
      | ((hp hero) + 30) > 100    = Player 110 ((potions hero) - 1)
      | otherwise                      = Player ((hp hero) + 40) ((potions hero) - 1)
```

The second challenge that we faced during the span of the project was the primary flow control. As described in the section above, functional programming languages' primary flow control stems from function calls. Haskell does not allow for loops and instead relies on function calls to do the same procedure. At first, we attempted to use some type of recursion that would run a function *x* number of times where *x* was determined by the user's input. However, we quickly realized that the user's input could result in *x* equaling any number from zero to infinity. We quickly found a way around this roadblock by using function calls. By having one function call a different function, we were able to create a control flow similar to how branching with labels works. By treating each function as if it was a different label, we realized that we could eliminate the problem of not knowing how many times a specific function was called by not moving onto the next function to be called until a certain condition was met.

```haskell
goToDogHouse str character = do
    let newPlayer = move character
    if (isAlive newPlayer == False) then gameOverFail else do
        putStrLn "\n\"Blue?\" you call out as you walk out the front door towards the dog house."
        putStrLn "You hear your dog, Blue, trotting towards you with something in his mouth."
        putStrLn "\"Should I take the ITEM, check out the KITCHEN, or check out the BACKYARD?\" you wonder to yourself as you absent-mindedly pet Bl
        input <- getLine
        if input == "STATUS" then do
                                                          let x = (read (numPots newPlayer) :: Int)
                                                          if (x > 0) then do
                                                                                          status newPlayer
                                                                                          putStrLn "Would you like to drink a potion? YES or NO?"
                                                                                          input <- getLine
                                                                                          if (input == "YES") then do
                                                                                                                                  let newCharacter
                                                                                                                                  goToDogHouse str
                                                                                          else if (input == "NO") then goToDogHouse str character
                                                                                          else do
                                                                                                          putStrLn "Command not recognized. Please try
                                                                                                          goToDogHouse str character
                                                          else do
                                                                            status newPlayer
                                                                            goToDogHouse str character
        else if input == "ITEM" then takeItem str newPlayer
        else if input == "KITCHEN" then goToKitchen str newPlayer
        else if input == "BACKYARD" then goToBackyard newPlayer
        else do
```

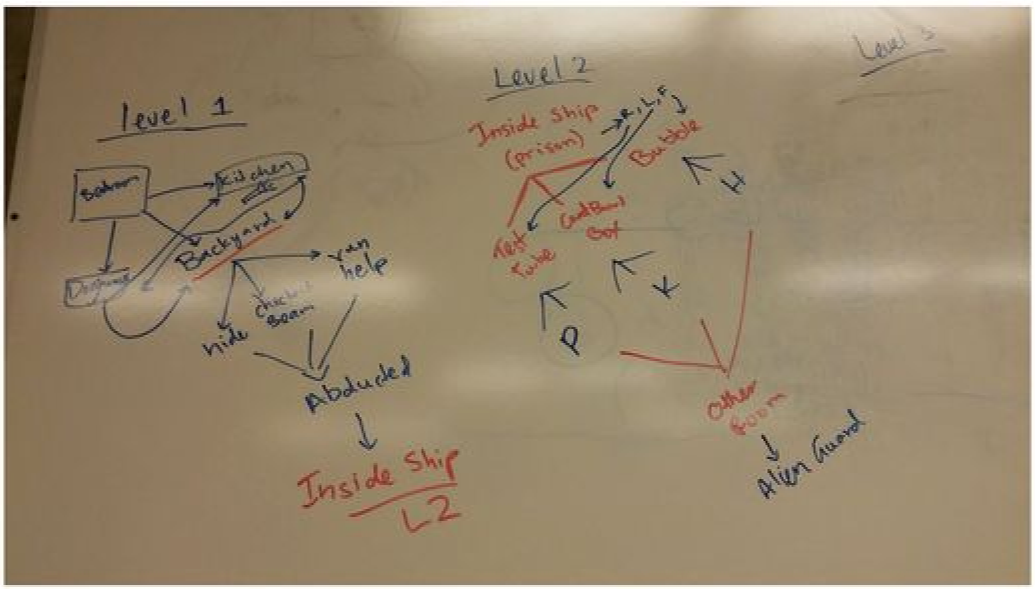
The third challenge that we faced during the span of the project was creating an original storyline. Although coming up with ideas of what our story should entail was easy, finding a way to connect all of our ideas and ensuring that the story flowed smoothly was much more difficult. In the end, after many revisions, our storyline was completed.

Storyline

Abduction: Set in a parallel galaxy, you have to find your way out of the maze by finding clues.

Beginning: a difficult start where there is no way to escape.

Ending: You were in your backyard all along (lol)



The fourth and final challenge that we faced during the span of the project was group dynamics. When we started the project we originally had two additional members in our group: Francisco Rocha and Vanessa Hurtado. At the beginning of the project, our original group took a very long time to decide what our theme and exact scope of the project was going to be. In addition, at least half of the group members did not show up often to the predetermined meetings nor did they do their share of the work using the excuse that their other classes were just too hard and took up too much time. The script was supposed to be a collaborative effort, however the parts that Vanessa wrote had numerous grammatical errors and often made no sense with respect to the rest of the script. Therefore, we had to re-write and finish the script ourselves. Since there was no significant progress being done on the project collaboratively, Alice took it upon herself to implement the entire Java version of the game. Although Francisco did submit a file containing some lines of code that he copied off of a Youtube video tutorial, none of the copied lines were helpful for the Java implementation nor did they make any contributions towards the program. Afterwards, Deekshita helped Alice debug and test the Java implementation. Three weeks before the group's scheduled presentation, our original group met up in order to work on the project together. During this time while Deekshita and Alice were working on testing the Java part, Vanessa and Frankie were working on their CMPS 111 classwork instead of on their share of the project, the Haskell part, like they were supposed to. We (Deekshita and Alice) let this matter slide because Frankie and Vanessa said they had made significant progress on the Haskell part. However, upon looking at their code on GitHub, we realized that nothing they had implemented was actually working correctly. The only progress they had made was the copying and translation of the Java's *System.out.println* statements to Haskell's *putStrLn* statements containing the level one script. None of the complicated Player or functions logic was implemented. The weekend before the original group's scheduled presentation, we were worried about the progress of the Haskell program, so we got together multiple times and nearly completed the Haskell part, just in case Vanessa and Frankie did not do their part since they had been consistently letting us down throughout the quarter. By the end of that week, Frankie and Vanessa still had not made much progress nor had they even made much of an effort to complete the share of the project. Therefore, after talking with the professor, TA, and the entire group, everyone agreed that the best thing to do considering the group dynamics was to split the group in half. As a result, the work that had originally been meant for four people was completed by only two by the end of the project.

Project Architecture

Java:

- Main.java: The main program which goes through the entire script using loops and conditional statements
- Player.java: Player object, which specifies the player's properties

Haskell:

- Main.hs: The main program which includes function calls which take in a player and modify it at each move
- Player.hs: Player module, which specifies the type of the Player object and all of its functions

Commands used for Runtime test:

1. Start Program
2. Enter Name: Tester
3. STATUS
4. BACKYARD
5. COMPLY
6. STRANGE NOISES
7. SCREWDRIVER
8. STATUS
9. SCREAM
10. end game

Project Compilation and Execution

How to compile the Java program:

1) Download the zipped project code folder.

2) Unzip the folder and extract all of the contents to wherever you desire. Remember that location.
3) Open up your computer's terminal.
    a) On Linux machines the default application for the terminal is called "Terminal".
    b) On Windows machines the default application for the terminal is called "Command Prompt".
    c) On Mac machines the default application for the terminal is normally called "Terminal".
4) Type in "cd " and the name of the location to which you extracted the folder contents.
5) Change directories into wherever you have the project downloaded.
6) Type in "javac Main.java"

How to execute the Java program:

1) Type in "java Main"
2) Enjoy the game!

How to compile the Haskell program:

1) Download the zipped project code folder.
2) Unzip the folder and extract all of the contents to wherever you desire. Remember that location.
3) Open up your computer's terminal.
    a) On Linux machines the default application for the terminal is called "Terminal".
    b) On Windows machines the default application for the terminal is called "Command Prompt".
    c) On Mac machines the default application for the terminal is normally called "".
4) Type in "cd " and the name of the location to which you extracted the folder contents.
5) Change directories into wherever you have the project downloaded.
6) Type in "ghci Main.hs"

How to execute the Haskell program:

1) Type in "main"
2) Enjoy the game!

Results

Lines of code breakdown:

Main.java	808
Player.java	46
Main.hs	459
Player.hs	24
Subtotal (Java)	854
Subtotal (Haskell)	483

Total	**1,337**

Implementation time (includes testing) in hours:

Creative process	9
Java	18
Haskell	25
Total	52

Program execution time:

Java - System calls	0.034 sec
Java - User actions	0.200 sec
Subtotal (Java)	0.234 sec
Haskell - Compile Time	0.140 sec
Java - Runtime (Full Game)	**0.290 sec**
Java - System Calls	0.052 sec
Java - User Actions	0.238 sec
Haskell - Runtime (Full Game)	**0.730 sec**

Our Java game runtime was actually less than half our Haskell game runtime. We believe this is because in our Haskell game, each time we call the player and edit it's health, we create a new instance. This way the Haskell program spends a lot of time creating new instances of the player, so it has a higher runtime. If we had implemented the player as a type in the IORef monad, we would have been able to create the global, mutable variable to speed up the Haskell program. One of the main benefits of the Haskell program was that it required much fewer lines of code to write, almost half of the Java program.

Conclusion

Through the extent of this project, we really learned how to convert a Java program into a Haskell one. We got a further understanding of the IO monad in Haskell and how to use it. We also learned how to deal with mutable variables in Haskell, so that when they are passed into another function they take on a new value, therefore creating a new variable and assigning it the new value, pulled from the old variable.

While this project was initially planned out for a group of four people, we had to finish it with only two members because of the group dynamics mentioned above. This increased each of our workloads, but we were still able to complete the game to the requirements we set at the beginning of the quarter.

This project, the text-based adventure game, is heavily dependant on global, mutable variables for the Player class and the player's features. And since Haskell is a language based on immutable variables, finding a workaround was hard, and costly. Also, the logic is simple and easy to translate into basic code in Java, which is why we believe that Java is a better language for this type of program. However, the upside to writing the program in Haskell was that it required fewer lines of code, even though it may have been less intuitive to write. In addition, it was easier to identify and fix bugs and errors in Haskell than it was in Java. In Java, the testing and debugging process was much longer. At the end, performance wise, our Java game was faster than our Haskell game. So overall Java is an easier language to write text-based adventure games in.

Works Cited

"Functional Programming Vs. Imperative Programming." *Learn to Develop with Microsoft Developer Network | MSDN*. N.p., n.d. Web. 6 Mar. 2016.

"Functors, Applicative Functors and Monoids." *Learn You a Haskell for Great Good!* N.p., n.d. Web. 6 Mar. 2016.