Generic Text Adventure

Building a Text Adventure Game Functionally

Justin Wong - Rachael Lew - Max Savage - Gabriel de la Mora
jujwong@ucsc.edu - ralew@ucsc.edu - msavage@ucsc.edu - gdelamor@ucsc.edu

Today, the most popular programming language for game development is C++, due to its support for object-oriented programming as well as its efficiency. Other languages such as C, C# and Java are also used in limited capacities. However, games are never developed in functional programming languages or even with support for functional features. For our project, we wanted to explore *why* games are only built in object-oriented or imperative styles by developing a game in Haskell.

Motivation

Our reason to create a game was simple - we all love making and playing games, and building a game in Haskell seemed like a unique challenge. We wanted to learn Haskell by doing, as the only way to wrap one's head around a programming language is to dive in headfirst. Building a text adventure seemed like a more realistic goal for a one quarter project. In addition, we wanted to do something different. Instead of creating a static text adventure, we created the Generic Text Adventure. However, "Dynamic" might better describe our project; because the "Adventure" aspect is entirely user-generated, the game can be different every time it is played. We had originally planned on creating an implementation in Java and/or C++ but decided against it, due to the complexity of the Haskell implementation, as well as to the fact that such an implementation did not seem challenging enough, considering we had all built games in those languages previously.

Haskell

Haskell is a general purpose, purely functional programming language. Because of this, functions in Haskell do not have side effects. This, coupled with its strong static typing system, allows for the development of software that is (almost) bug-free. However, developing in Haskell requires a different approach from developing software in an imperative language, due to the fact that data is immutable in Haskell. Integral to the Haskell experience is the monad, a structure that represents computations defined as a sequence of steps.

The Generic Text Adventure

The use of the term "generic" refers to the fact that the text adventures are completely user-generated. Instead of providing a set story, we allow the user to write his or her own adventure game based on a template (hence *generic*) and then use the GTA to play it. Therefore, the GTA itself is not quite a text adventure; more accurately, it builds one.

The program takes a formatted text file, parses it, and then creates a text adventure according to the file's contents. Each file acts as a blueprint of the adventure, specifying all the "rooms" the game will contain, as well as their properties. A room consists of an ID number, story description, and actions that the player can take to navigate through the game.

Development

For this project, we split into two teams for the early development. One worked on parsing, while the other built the backend of the game. This division of labor allowed us to focus on the two main parts of the project at the same time, which was very efficient. Throughout the quarter, we would meet up as a group to ensure that both portions of the code synced up with each other.

There were two main problems that we faced while developing this project - the first was dealing with Haskell's input/output system, which is not well suited for making games since they are heavily based on user input. This led to problems with both parsing and building the data structure. In terms of the latter, only one function needed to retrieve user input, which required using I/O actions. However, because Haskell separates pure code from "impure" I/O actions, performing I/O could not be done inside that function. The solution involved creating a main function containing a do block that performed all the I/O actions and called every function, so that the user input could be passed in without running into scoping problems.

The other issue was simulating looping in Haskell. Text adventures typically use a main loop as the general approach; however, because Haskell does not support looping functionality, creating a main loop had to be tailored to a recursive implementation. This proved to be tricky, since values would be changed after one run-through, and it was necessary to retain those new values. However, with each recursive call, those values would be reset. It was resolved by creating a separate loop function that took in the new value.

The Text File Format

The format of the text file lists a room's ID number, description, and options, with each on its own line. The first line of each room is an `Int` that serves as the ID number, indicating to the program the current room. Navigation through the game also references this ID. In terms of the text file's organization, the `Int` determines the start of a new room and thus, functions as a separator between rooms. Following that line is a `String` that holds a room's description, which is the main story element for the text adventure. The description can be whatever length, so long as it is contained in a single `String` and contains no line breaks. Each room must have those first two lines specified. Any subsequent lines not carrying only an `Int` contain the room options. Each option consists of two elements: 1) a `String` that describes an action the player can take by typing it into the console, and 2) an `Int` that represents the ID of the room to which that action will move the player. Both the `String` and `Int` are on the same line, delimited by a white space. There can be any number of option lines depending on how many room connections and actions are desired.

Example format:

```
1
First Room's Description
First Option2
Second Option3
2
Second Room's Description
First Option1
Second Option3
3
Third Room's Description
```

```
First Option1
Second Option2
```

The GTA allows for both a win state and lose state, each specified as a room denoted by the numbers 0 and 99 respectively. The first room must be given the number 1.

Data Structure

To convey the game's organization as a system of rooms, the data structure consists of a list of tuples, with a tuple representing one room. Inside each tuple are three components: 1) an `Int` acting as the room number, 2) a `String` detailing the description of the room, and 3) a list of pairs, where each pair's type is a `String` holding the action to be taken and an `Int` that determines the new room number associated with the action.

```
[(Int, String, [(String, Int)])]
```

Though rather complicated, the data structure represents a list of rooms and functions as Haskell's version of a system of nodes, which is how this game would be implemented in an object-oriented programming language. Using tuples allows a room's properties to be grouped together as one value, making it easy to access fields for a particular room.

To traverse the data structure, we created a type of pointer through the function `roomIn`, which takes in the list of rooms (the data structure) and a room number (an `Int`). It uses pattern matching and list comprehension to locate the room specified by the given `Int` and returns that room (a tuple). The program then prints out the description and list of options associated with the room, as provided by the values inside the tuple. These actions are carried out by two functions: `printDescription` takes a room tuple and returns its `String` value (the description); `printOptions` takes a room and uses list comprehension to isolate the `String` contained inside the (`String`, `Int`) pair (the option).

Parsing

Parsing consisted of analysing the input file into the "Game" data structure. We used the same techniques discussed in class in order to accomplish this. Most of the difficulty of parsing came from the notion of *pure* functions in Haskell, which always return the same value. However, functions that perform IO are considered *impure* as their return changes depending on input, such as getLine. So parsing needed to be done separately from reading the file.

The parser conforms to the standard format for the text file described above. If a file has been correctly formatted to this specification the parser will read through the file, returning the values needed to build the data structure. There were only two functions needed to implement parsing, One to read in room numbers with their corresponding positions, and one to read the list of options associated with each room.

The function signature for parse is:

```
parse :: [String] -> [(Int, String, [(String, Int)])]
```

We first use getContents to return the entire file as a string, then use lines on that string to create a list of strings separated by newlines, so this list contains every line of the file. The parser then uses pattern matching to determine what is on each line of the file and process it accordingly. Conforming to the format for the text file, each line will either contain a number, a description or an option number tuple, the parser then interprets that data into the desired data structure that is used to actually build the game.

How the Game Runs

If using GHCI, after loading in the program file, the user types main into the terminal to run the main function , which consists of a `do` block that returns an I/O action. If running from the terminal, the user compiles with ghc then runs the executable. It first parses the text file specified in the program, organizing the contents into their respective fields inside the room list data type. The game always begins with the first room, printing out its description and options. Using `getLine`, the program reads in the action that the player has typed and binds the result to a `String`, which is used to decide which room to move to. The GTA also checks for invalid input by using a boolean to determine whether or not it matched any of the current room's options. If the boolean returns `False`, the function displays an error message and loops through itself to redisplay the description and options.

```
getOpt <- getLine
putStrLn ("\n")
if(valid getOpt (roomOpt (roomIn this (1))) == True)
 then
   do
     let getRoomNumber = (getRoom getOpt (roomOpt (roomIn
this                     (1))))
     putStrLn (mainFunc getRoomNumber)
     if((endState getRoomNumber) == False)
       then
           loop getRoomNumber
       else
           putStrLn ("")
 else
   do
     putStrLn ("Invalid option")
     play
```

A successful player input leads to the next function, `loop`, which runs the core loop of the game. This function's code is fairly similar to that of `play`, except that it takes in the new room number. Its function type

```
loop :: Int -> IO ()
```

allows it to call itself without losing the value of the new room number, so that the game can progress on to the next room.

The player wins when he or she reaches certain rooms that trigger either a victory or lose message. These rooms have specific numbers assigned to them (0 for win, 99 for lose), so that when pattern-matched, they serve as edge cases that will stop the game loop. A boolean determines the end state of the game by checking if the current room number is the win or lose room ID numbers.

```
endState :: Int -> Bool
endState i = if (i == 99 || i==0)
          then True
          else False
```

A short example using with the GTA

```
Gabes-MacBook-Pro:cmps112 GabedlM$ ./GenericText
What story would you like to play
SampleStory.txt
Quick Run, They're Right Behind You!!!! You don't even turn around as you frantically start sprinting. You see a door and take it,
but once your inside do you go Upstairs or Down?

Options:
up
down

up


Upstairs, You feel relieved just before you hear them bust in downstairs, quick should you jump out the Window or Hide!!

Options:
jump
hide
hide


Alright, you can hear them coming but are pretty confident in your spot underneath the table. As they get closer though you begin t
o panick, as your heart starts racing you panick; out of the corner of your eye you see something shiny, it's a knife and you pick
it up. You hear the door start creeking open AND They all yell SURPRISE!!! It's Your birthday, you've just been playing to many vid
eo games.

Options:
Cut the Cake

Cut the Cake


YOU WIN! GRATULATIONS!
```

Conclusion

After developing a game using a functional programming language, it is not difficult to understand why developers prefer object-oriented programming. It is definitely a much more natural approach to the task of designing and building a game. While we enjoyed learning Haskell by doing this project, none of us would suggest building a game in Haskell, as it can become very cumbersome. Throughout the duration of this project, we often found ourselves trying to mimic features of other more familiar languages. However, having gained more experience with Haskell, we now perhaps could more easily develop a game that fully utilizes the features of functional programming.