Background and Concepts

I chose to make a text-adventure game in Haskell as my project. I knew I wanted to make a somewhat novel piece of software, and after researching several ideas, I decided that a simple game would be the most realistic and scalable with time.

A text-adventure game is just that, a game based entirely in a world described by text. A videogame without the video part (most of the time, anyway). The player is placed into a scenario, which is described to them explicitly with text. The player can then interact with the world around them by entering various commands using the keyboard. That’s basically it!

I was born a little after the time when text-adventure games were popular, so I didn’t really know much about them until recently. In preparation for this project, I started playing some classic text-adventure games that I had often heard others talk about or reference fondly in passing. The two that I spent the most time on were *Adventure* and *Zork I*. These are both very fun and interesting games, and they’ve held up to the test of time quite well, in my opinion.

But those are big and rather complex games! I’m sure they took much more than four weeks to make, and I didn’t expect to match them in any real way, but they gave me a lot of inspiration for what I was going to do over the next four weeks.

There were some basic features, common to many text-adventure games, that I wanted to implement:

- A mutable player inventory that the player can view.
- Items throughout the world that can be interacted with.
- Objects in the world that can be examined more closely.

- The ability for the player to save and load their game.
- A way for the player to move about the world.
- A useful help screen that the player can access.
- Barriers that block progress and require problem-solving to overcome.
- Interesting responses to (at least some) player input.
- A short but interesting story for the player to uncover.

There were many other ideas floating through my head about different features and fun little mechanics, but I didn’t put those down on paper, so they weren’t part of the plan explicitly. Apart from jotting this list down, there was almost no planning, and I just started working.

Procedures and Progress

Game loop:

The first thing I needed to do was figure out how to create a game loop. I was very baffled by this idea at first, as I hadn't really been able to fully grasp how looping could even be done in Haskell. I tried to wrap my head around the State Monad, which seemed like it would work for what I wanted to do, but it was simply too complex for me. Luckily, after the lecture where we looked at the little number guessing game, I finally understood how effectively recursion could be used in place of explicit looping to maintain a game state and keep track of things. Soon after, I had a game loop (the **play** function), which kept track of the player inventory and location. Success!

```
> play          :: GState -> Prompt -> IO ()
> play (is,r) p = do
>      loadScreen r is
>      l <- readline (p++"\n> ")
>      case l of
>        Nothing    -> play (is,r) noCommand
>        Just ""    -> play (is,r) noCommand
>        Just input -> do
>          addHistory input
>          exeCmd (is,r) input
```

[*Note: For information about the functions, etc. that are used in this and other pieces of code, see Appendix C on pg. 30*.]

In this function definition, **GState** is a type that describes the current game state (Inventory and Location information) and **Prompt** is simply a type alias for the String type and describes what should be displayed to the player during the current execution of the **play** function.

[*Note: It's worth mentioning that this is the final version of the **play** function, after multiple refactoring iterations. This will likely be the case with all code fragments unless otherwise noted.*]

I was originally using the basic IO Monad functions to read and write to and from the console, but soon realized that the backspace button did not behave at all desirably when the player entered text with this setup. After some digging through the Hackage libraries, I found the *Readline* library, which allows for a much more intuitive input experience, while making the output experience easier as well, mainly with the very versatile **readline** function, which allows you to specify a prompt to be displayed to the user and reads in the users response all in a single call to **readline**. In addition, the *Readline* library allowed me to use the **addHistory** function to store the user's past inputs and allow them to use the up and down arrows to go through them and avoid reentering text, much like any command-line environment would. Essentially, this library made the user experience significantly better, more intuitive, and more familiar. Since the **readline** function returns a **Maybe String**, it was also necessary to import the *Data.Maybe* library, which is also used in many other places.

The key idea with this implementation is that the game loop function should only ever end up doing one of two things: calling itself recursively *or* calling a function that eventually ends in a call to the game loop function. It helped me to think of that second case as a kind of *indirect recursion*. The only exception to this is the case where the game is over, in which the game loop is purposefully *not* called, causing the program to exit.

This is one way that a game loop can be done in Haskell. There are surely other, more sophisticated ways, but for a small game like this it definitely gets the job done.

An ANSI aside: I want to mention a very useful library that I found: System.Console.ANSI. Among many other things, this library allowed me to change the player console's background and foreground colors, as well as clear the console screen when changing scenes, and even hide the player's cursor when they just need to press a key to continue on. This is why the screenshots in Appendix B have green text on a black background, even though my console colors are actually set to black text on a white background. With all of these functions, it's worth noting that one must change back anything that they change before the program exits, otherwise the changes will persist. There's even a function for changing the console title, but peculiarly, no way to store the original title, and thus no way to change it back. I've covered about 10 percent of the functionality here, as it's quite a sizable library. [*Information on this and the System.Console.Readline libraries can be found in the References section of this paper*.]

Parsing player input:

Now that I had a game loop, it was time to start doing some interesting things with it! I wanted to get some player interaction working first, since that seemed like perhaps the most important game mechanic.

The first thing I needed to do was create a function that takes in a string, parses it to find the player's command and arguments, and then executes the specified command by calling an auxiliary function for that command with the arguments.

In order to get this working, I needed a way to take the input from the player, look at it, and determine whether or not a valid command was entered:

```
> readCmd  :: Maybe Command -> Command
> readCmd s | (isJust s) && lowCmd `elem` cmds = lowCmd
>           | otherwise                        = "BAD_CMD"
>   where lowCmd = map toLower (fromJust s)
```

The **readCmd** function expects to be given **Just** a potential **Command** or **Nothing**, where **Command** is a type alias for the **String** type. If **s** is **Nothing**, or **s** is not a valid command (according to the **cmds** list), we return **"BAD_CMD"**, which tells the calling function to tell the player that their input was not valid. This function uses the **toLower** function, which is imported from the *Data.Char* module, to make sure we always have a uniform lowercase version of any command entered by the user for comparing to the list of valid commands.

A lazy aside: Haskell's lazy evaluation allows us to define the function **lowCmd** in the readCmd function's **where** clause that uses **fromJust** to convert what is *assumed* to be **Just s** to **s**. This is allowed, even though, if **s** happened to be **Nothing**, we would get a massive error! The interpreter won't complain because lowCmd is *only* executed when it we call it *explicitly*. So, if we *always* check whether **s** is **Nothing** and *never* call **lowCmd** if it is, then we will *never* encounter this error, while also having this convenient function for when **s** isn't **Nothing**.

All asides aside, there were still several things needed to get the input parsing functionality up and running. With user commands taken care of, I turned my attention to the argument(s) that are assumed to be entered after a given command in an input string. In order to make things less complicated, I decided to look through every word after the first word of the user input string and pick out the *first* meaningful word (i.e. that which corresponds to some keyword that is defined as being in the world and not another command):

```
> parseArgs  :: String -> String
> parseArgs s = let x = find (`elem` keywords) tailS in
>                   case x of
>                        Just x  -> x
>                        Nothing -> unwords tailS
>   where tailS = (tail . words) s
```

The **parseArgs** function takes the user input string, converts it into a list of individual words with the **words** function, gets rid of the first word (assumed to be the command word), and then looks through the remaining words for the first word that matches any word from the list of **keywords.** If we find a valid word, we just return that word. If we can't find any meaningful words, we return the list we just searched through so that it can be showed to the user (along with a failure message) by the calling function.

Executing player commands:

Now that we have a way of parsing commands and arguments entered by the user, we need a function that can use the results to do something useful, preferably what the user is expecting, where possible. To do this, I created the **exeCmd** function, which is essentially one big **case** statement that passes the **GState** and user argument to a specific auxiliary function for each possible **Command**. [*Friendly reminder that Appendix C contains the code and descriptions of all functions mentioned in this text.*]

The commands that I ultimately decided to support are those which allow the player to *take* an item, *drop* an item, *move* from one area to another, *examine* an object, item, or area, *open* a door, *save* the game, *load* the game, bring up the *help* menu, and *quit* the game. [*Note: There are some additional reactions that the* ***exeCmd*** *function can have to certain commands, but they aren't particularly important for this discussion.*]

Although the functions for *taking* and *dropping* items were the things I worked on next, neither are particularly interesting, so I won't talk much about them. The function for *moving* from area to area, however, was what I worked on after those and it was particularly challenging for me.

The first step was finally solidifying the world that my game would take place. I hadn't given this part of the project much thought up to this point, but my original idea was to have the whole game take place in a house, where a player would only ever be moving from room to room. This is the concept I ended up going with, mainly because it made things less complex. The next step, however, was figuring out how the player entering a *move* command could translate into an actual relocation that is understood by the game state. Luckily, I had accounted for this originally in my design of the **GState** type, which included a **Room** (type alias for **String** type) variable to hold the name of the room that the player is currently in. So, it was set up to always know where the player was, which is good, but I needed a way to change this in logical ways. We don't want a player moving from a room to a room at the opposite side of the house in one step, although this is an arbitrary rule, as that would likely seem like a bug to the player if it was allowed. In particular, we only want the player to move to a specified room if that room is *adjacent* to the room the player is currently in and there is a direct path to it. Since there were only seven rooms that I had decided to define in the house, it seemed easier to essentially designate all this things manually. I defined lists of adjacent rooms for every room and had the **moveTo** function call a function called **isAdjTo** as an infix operator to check whether a specified room was valid to move to. In addition to checking whether the desired room is valid, I wanted to check whether certain items had been collected before allowing the player to proceed, for which I defined a new function called **isForbidden**. For example, if the player was on the *porch* and hadn't found and taken the **key**, then the door would be locked and progress would be impossible. However, once the player had found the key and had it in their inventory, they could proceed through the door with no issues.

I had originally designed the move functionality to require that the player enter the room name explicitly as the argument to the move command, but I eventually realized that I had no way to guarantee that the player would know what the rooms were called without first entering them, which made no sense. I then decided to also implement a way to handle *forward*, *back*, *left*, and *right* arguments, which allowed the player to move around without knowing the names of the rooms. To get this working, I made helper functions for each direction, essentially just tacking on this additional functionality.

The functions that I made next allowed the player to *examine* things and *open* doors, but those functions are not as interesting as the **saveGame** and **loadGame** functions, which were surprisingly difficult to pin down:

```
> saveGame       :: GState -> IO ()
> saveGame (is,r) = writeFile "./Save.lhs"
>  ("> module Save where\n\n> save = ("++(show is)++","++(show r)++")")
```

Originally, I figured I would just read and write to a text file to save and load a player's game. Implementing the save functionality in this way was trivial, but loading was a different story. Reading from a file is an IO Action, and so, instead of reading the file into a **String** and doing what I wanted with it, I was given an **IO String**. In order to handle this, I would have had to change most of my functions to take and use the **IO String** type, and my entire program would have to be "tainted" with IO. This was the moment that I truly began to understand the separation of purity and impurity in Haskell, and the words "There is no escape!" reverberated through my head as I stared at my code.

So, instead of dealing with all that, I found a workaround that does the trick but is *far* from ideal.

Instead of writing to a text file, **saveGame** modifies a *module* that is part of the program. This way, there's no need to do anything with IO, **saveGame** could just rewrite the contents of the *Save module* with **writeFile** and **loadGame** could access the **save** state directly throughout the program because the **save** state *is part of* the program structure. This does present some problems, though. If the player saves their game, they must then *quit* the game and reload the *Save module* in the interpreter in order to have their new save be recognized. I don't know if there is a way around this, but I couldn't find one.

Everything else:

With these core mechanics essentially worked out, I continued on, but the rest of what I chose to implement would probably be better discussed in the next section, in which I focus on what I should have done differently or simply not done at all.

Reflection and Observations

Aesthetics:

Very early on, I started thinking that about displaying things beyond just text descriptions to the player. Specifically, I wanted to give the player something useful or interesting to look at; some crude graphics. So, after I had the core mechanics worked out, I started working on some ASCII art "screens", which were essentially just pictures that could be shown to the player in addition to the text descriptions.

The choice to go this route, in my opinion, was a huge mistake.

I ended up spending a lot of time making these *screens*, and, although I'm overall happy with the results [*See Appendix B*], it was not necessary at all and doesn't really add all that much to the experience. The biggest issue is that the implementation of showing the player these *screens* ate up most of my time. If I hadn't chosen to mess around with this graphics idea, I would have ended up with a much more polished and interesting story with more and better things for the player to do.

The screens weren't a bad idea, necessarily, and if they had been planned from the start and set off as something extra to do if time permitted, the end result would have been much better.

This leads me to the discussion of two very important aspects of software development that I basically ignored entirely: Planning and prioritizing.

Planning (or lack thereof):

As I mentioned at the start of this paper, I only sketched out a rough idea of the features I wanted before I got to work. I didn't know much about what makes up a text-based game or what I wanted the game, story, presentation, or feeling of the game to be. Overall, there were a huge number of aspects of the game that I hadn’t accounted for at all. Even though I knew how important and intentionally long the requirements gathering stage of software development usually is, I was excited to get started and got ahead of myself. The result was a lot of supplementary code tacked onto established functionality, large amounts of time spent refactoring previously written code functions that needed to account for new things, and in general, a very spaghetti-like program structure. I’m not very happy with this outcome, but at a certain point I had to stop writing code and start writing this report.

Another part of the planning issue boils down to a lack of research. I thought I had done a decent amount of research when I started my project, but as I was working, I found myself needing to go back and study aspects of text-based games or game loop mechanics, among other things.

Looking back, I can see how doing more research before starting to plan, and then planning until I had most of the game laid out, would have led to a much more interesting player experience and better-quality code/program structure.

Priorities:

Essentially, I believe I had my priorities backwards. Apart from implementing the basic commands in the beginning, I focused on making things more interesting or aesthetically engaging instead of just making things work. Now, this approach might work for people that are more experienced with making text-based games or making larger programs with Haskell, but for myself (a beginner in both of these things), just going with the flow was not the correct approach. If I had focused on building things up from the bottom instead of flitting around all over the place, implementing this or that on a whim, I would have had a much better time and a far more cohesive program to show for it.

Ultimately, I believe my inability to focus on or maintain priorities in this project follow directly from the lack of planning. I hadn’t laid out a set of goalposts, so I had to prioritize things as they came up, which is *not* a winning strategy. The lessons that I learned from this project are very valuable ones indeed, and I’m glad I was exposed to them now rather than later.

Haskell:

I want to talk about Haskell briefly. I thought I knew quite a bit about Haskell by the time I started this project, but I really didn’t. There were many things that I had been exposed to in class that I really didn’t comprehend as well as I thought I had, and this project really helped me identify those areas and forced me to deal with (most of) them. As far as actually being able to use Haskell effectively, this was *extremely* helpful to me! I’m somewhat embarrassed to admit that I didn’t actually understand how function composition and the ‘.’ operator worked until two weeks into my project. I could never get it to work and instead clung tightly to the ‘$’ operator and obnoxious amounts of parentheses, but I *finally* figured it out through experimentation while working on this project.

That doesn’t even come close to the number of Haskell concepts, Prelude functions, libraries, and conventions that I was exposed to during this project. And I even understood some of it! I didn’t really touch monads though; too scary!

Overall, this project was a great learning experience for me. I had *fun* and I definitely got a lot out of it.

References

Libraries used:

Libraries covered in class - Data.Char, Data.List, Data.Maybe, and System.IO.

System.Console.ANSI library - http://hackage.haskell.org/package/ansi-terminal-0.9/docs/System-Console-ANSI.html

System.Console.Readline library - http://hackage.haskell.org/package/readline-1.0.3.0/docs/System-Console-Readline.html

Resources that directly influenced code:

[*Note: Apart from the functions defined in the libraries mentioned above and the resources listed below, the source code written for this project was entirely of my own creation, for better or worse.*]

The number guessing game that was written in class: The **play** function and general game control structure that I created were inspired by this code - https://github.com/zipwith/funlangs/blob/master/game.lhs

Although they didn't influence code directly, these text-based games influenced design decisions and functionality in various ways:

Zork I by Dave Lebling and Marc Blank - http://www.infocom-if.org/games/zork1/zork1.html

Colossal Cave Adventure by William Crowther and Don Woods - https://en.wikipedia.org/wiki/Colossal_Cave_Adventure

The Hitchhiker's Guide to the Galaxy by Douglas Adams and Steve Meretzky (*30th anniversary edition* by BBC) - https://www.bbc.co.uk/programmes/articles/1g84m0sXpnNCv84GpN2PLZG/the-game-30th-anniversary-edition

ASCII Art computer by Roland Hangg - https://www.asciiart.eu/computers/computers

Other resources:

Haskell 98 function definitions and examples by Miloslav Nic - http://zvon.org/other/haskell/Outputglobal/index.html

Haskell tutorial by Derek Banas - http://www.newthinktank.com/2015/08/learn-haskell-one-video/

Programming in Haskell, 2nd Edition by Graham Hutton - http://www.cs.nott.ac.uk/~pszgmh/pih.html

Learn You a Haskell for Great Good! by Miran Lipovaca - http://learnyouahaskell.com/

Hoogle by Neil Mitchell - https://hoogle.haskell.org/

Code samples for Functional Languages by Mark P Jones - https://github.com/zipwith/funlangs

Hackage - https://hackage.haskell.org/

[*Note: The remainder of these resources are things that seem like they would have influenced my code, but due to my inability to comprehend the code or the difficulty of translating from one programming language to another, they did not. Having said that, it still seems appropriate to include them here.*]

Haskell Text-Adventure Game by StackExchange user Addison - https://codereview.stackexchange.com/questions/159069/haskell-text-adventure-game

"Writing a game in Haskell" by YouTube user Elise Huard - https://www.youtube.com/watch?v=1MNTerD8IuI

Writing a Text-Based Adventure Game in Python by YouTube user Doug McNally - https://www.youtube.com/watch?v=miuHrP2O7Jw

Choose Your Own Adventure Game in Python (Beginners) by YouTube user Tech With Tim - https://www.youtube.com/watch?v=DEcFCn2ubSg

ASMR How to Program a Text Adventure [Tutorial] [Python] by YouTube user _asmr4u_ - https://www.youtube.com/watch?v=XeIgM7Hu41o

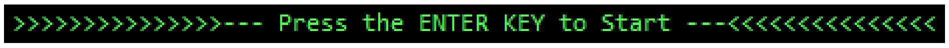
Appendix A - Testing

Testing was mainly done in stages, and much like the rest of this project, I essentially tested things as they came up. In retrospect, it would have been much better to have documented all of the testing that I did, but unfortunately, I did not do this and am now struggling to find a way to do meaningful tests. The program is relatively big and the functions are (almost unnervingly) interleaved.

To test this game, I'm going to use a rather vague form of testing methodology that is common with video games: playtesting. For brevity's sake, I'm going to omit those parts of the program's output that are not relevant whenever possible.

The first question to ask is 'Does the game start?'. We can test this by loading all of the program modules into GHCI and then calling the function that starts the game, called **start**:

Input to GHCI: start

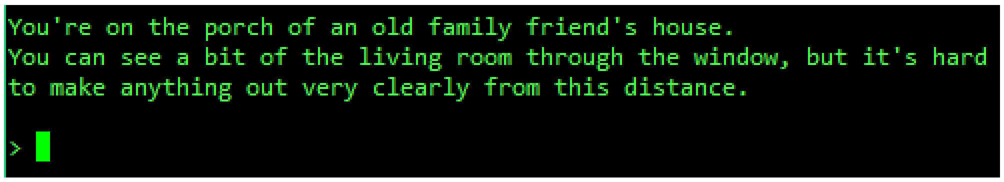
Output: 

```
>>>>>>>>>>>>>>>>--- Press the ENTER KEY to Start ---<<<<<<<<<<<<<<<
```

We are now at the start screen, which is what we expected. We are now out of GHCI and in the game environment. Next, can we start a new game:

Input to game: Pressed the Enter key on the keyboard.

Output:

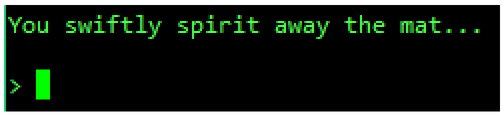


```
You're on the porch of an old family friend's house.
You can see a bit of the living room through the window, but it's hard
to make anything out very clearly from this distance.

> 
```

This is the first area in the game, and the expected start point for a new game. The description of the room is the correct output as well. Looking good so far. Let’s try to take an item that is supposed to be in the room:

Input to game: take the welcome mat

Output:

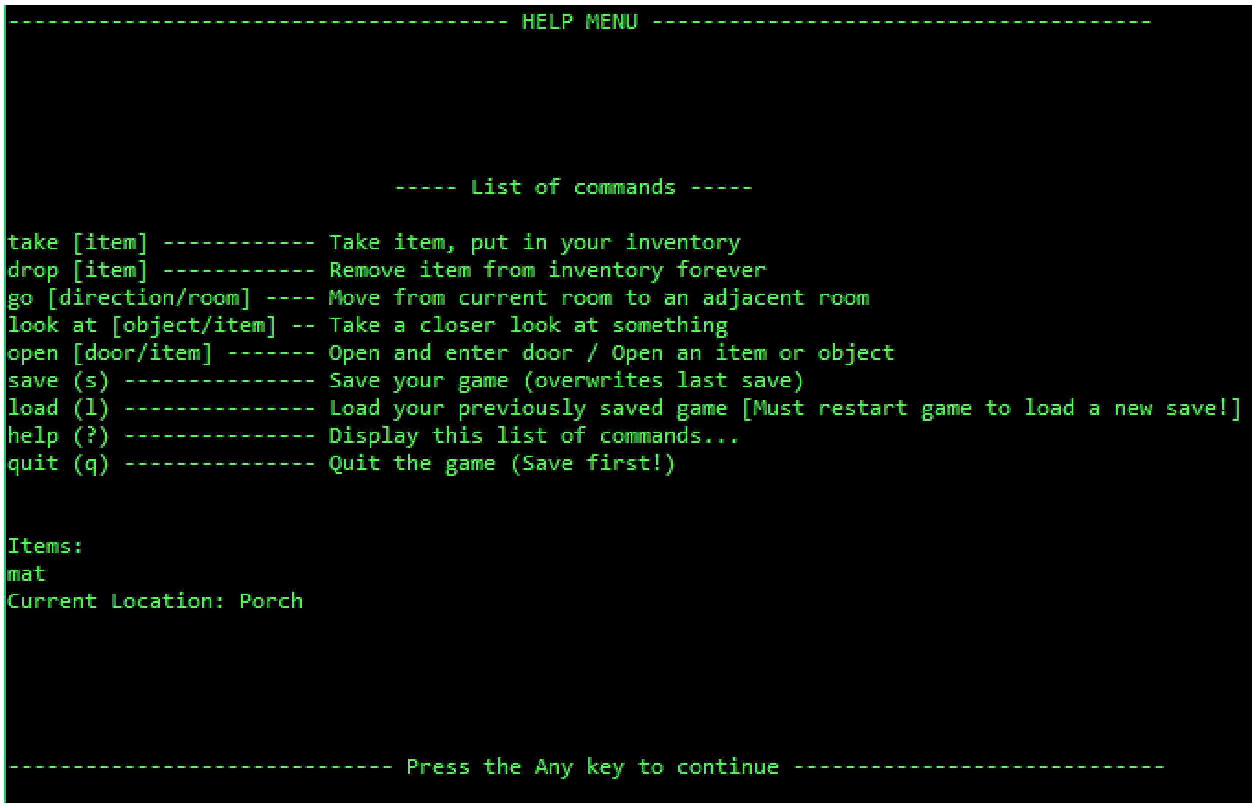


```
You swiftly spirit away the mat...

>
```

This appears to have worked. The help menu should tell us what’s in the inventory so that we can make sure the take command worked, so let’s take this opportunity to try out the help menu:

Input to game: help

Output:

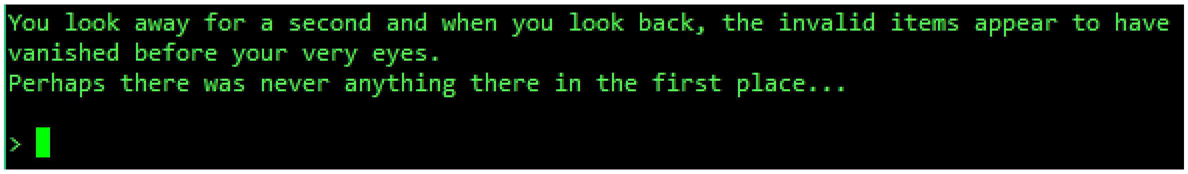


```
--------------------------------------- HELP MENU ---------------------------------------




                              ----- List of commands -----

take [item] ------------ Take item, put in your inventory
drop [item] ------------ Remove item from inventory forever
go [direction/room] ---- Move from current room to an adjacent room
look at [object/item] -- Take a closer look at something
open [door/item] ------- Open and enter door / Open an item or object
save (s) --------------- Save your game (overwrites last save)
load (l) --------------- Load your previously saved game [Must restart game to load a new save!]
help (?) --------------- Display this list of commands...
quit (q) --------------- Quit the game (Save first!)


Items:
mat
Current Location: Porch





------------------------------ Press the Any key to continue ------------------------------
```

The help menu appears to be working, and the mat that we picked up is even listed under the items in our inventory, so the take command seems to be working with valid input as well. But what about invalid input? Let's try to pick up an item that isn't in the room:

Input to game: take invalid items

Output:

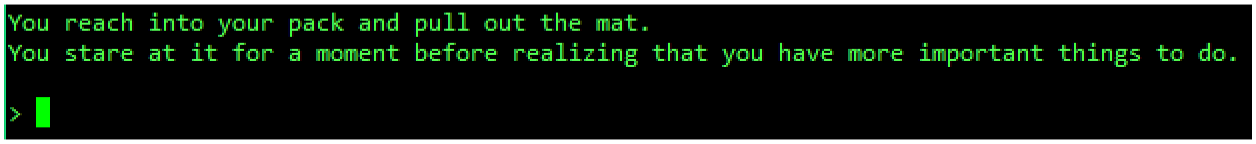


```
You look away for a second and when you look back, the invalid items appear to have
vanished before your very eyes.
Perhaps there was never anything there in the first place...

>
```

As convoluted as it may be, this is the expected output, which means we can't pick up invalid items. Good! Now, what happens when we try to take an item that we already have in our inventory? Let's try to take the welcome mat again:

Input to game: take the welcome mat

Output:

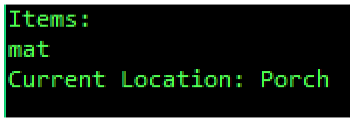


```
You reach into your pack and pull out the mat.
You stare at it for a moment before realizing that you have more important things to do.

>
```

Okay, that seems reasonable. Let's use the help command to check our inventory again after all that (not) taking:

Input to game: help me please!

Output: (omitting the irrelevant parts of the help menu)

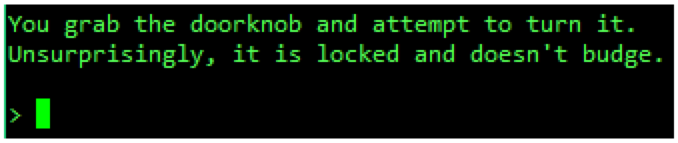


```
Items:
mat
Current Location: Porch
```

Good, we didn't pick anything else up, so an unchanged inventory makes sense. Next, let's try to open the door. It should be locked, since we haven't picked up the key yet:

Input to game: open door

Output:



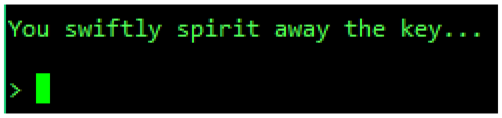
```
You grab the doorknob and attempt to turn it.
Unsurprisingly, it is locked and doesn't budge.

>
```

Good news. We are still on the porch and the door won't open. Let's grab the key and try again:

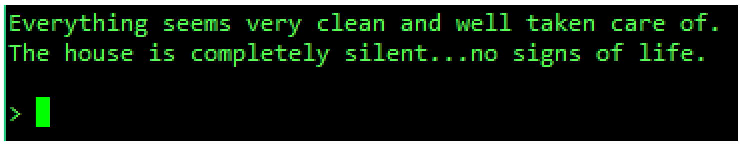
Input to game: take key

open door

Output:



```
You swiftly spirit away the key...

>
```



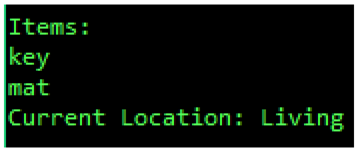
```
Everything seems very clean and well taken care of.
The house is completely silent...no signs of life.

>
```

We took the key, used it to open the door, and are presumably now inside the house. Let's verify this with a call to the help menu:

Input to game: h

[*Note: I used the abbreviated version of the help command here.*]

Output:

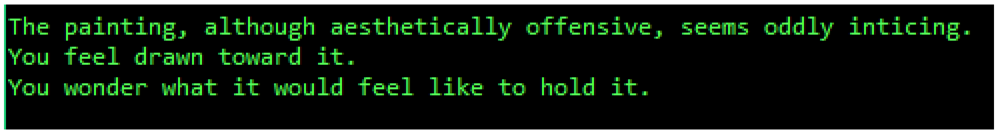


```
Items:
key
mat
Current Location: Living
```

This is expected, because the living room area is actually referred to as "Living" in the code, and we now have the key. Moving on, let's try the examine command on something:

Input to game: examine painting

Output:



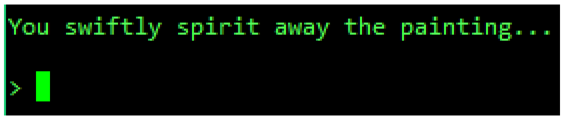
```
The painting, although aesthetically offensive, seems oddly inticing.
You feel drawn toward it.
You wonder what it would feel like to hold it.
```

This looks good. However, if we take the object and then examine it again, we should get a different description:

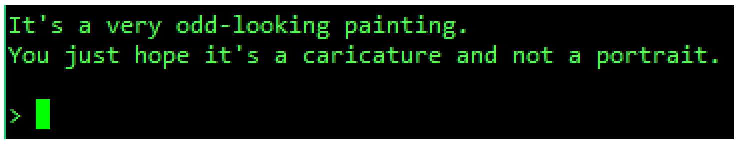
Input to game: take painting

examine painting

Output:



```
You swiftly spirit away the painting...

>
```

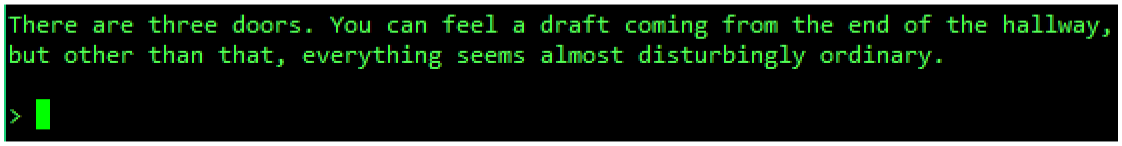


```
It's a very odd-looking painting.
You just hope it's a caricature and not a portrait.

>
```

Well, that feature appears to be working. Let's try moving around:

Input to game: go forward

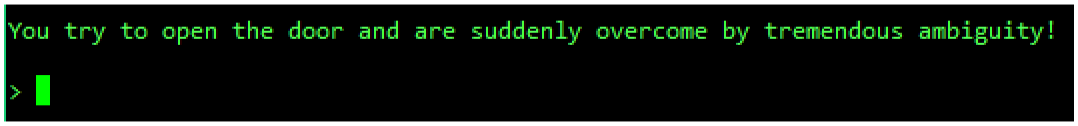
Output:


```
There are three doors. You can feel a draft coming from the end of the hallway,
but other than that, everything seems almost disturbingly ordinary.

>
```

It appears that we're in the hallway now, which is the expected outcome. There should be a special result if we try to open *the* door here, so let's try that:

Input to game: open door

Output:

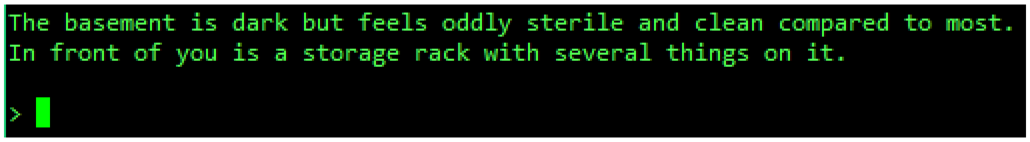

```
You try to open the door and are suddenly overcome by tremendous ambiguity!

>
```

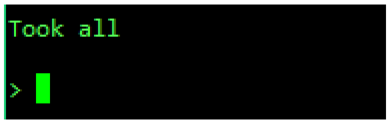
This makes some sense because there are three doors, so this is a nonsensical command in this specific case. Moving on, let's go to the basement to get the endgame item:

Input to game: go right

take all

Output:


```
The basement is dark but feels oddly sterile and clean compared to most.
In front of you is a storage rack with several things on it.

>
```

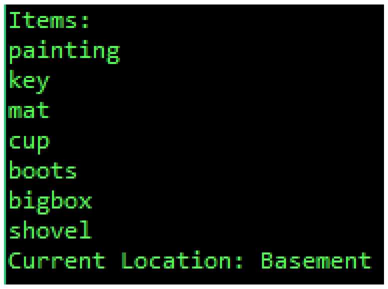

```
Took all

>
```

In going right, we ended up in the basement, which is exactly what was expected. I then used the special take all command, which should have taken every ownable item in the room, including the shovel, which will allow us to finish the game.

Let’s check the help screen’s inventory readout again to figure out if it looks how it should:

Input to game: ? (another abbreviation of the help command)

Output: (Again omitting irrelevant info)

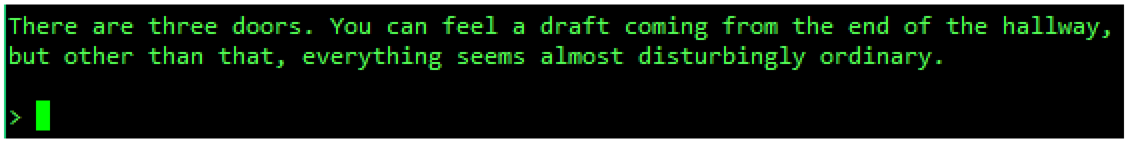


```
Items:
painting
key
mat
cup
boots
bigbox
shovel
Current Location: Basement
```

Our location looks good, and we have the shovel. Every other item is either something we already had or something from the basement that we got with the take all command just now. Let’s try moving to a room using the room’s name:

Input to game: go to the hallway

Output:

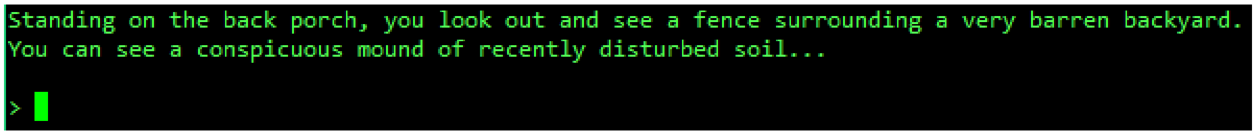


```
There are three doors. You can feel a draft coming from the end of the hallway,
but other than that, everything seems almost disturbingly ordinary.

>
```

We’re back in the hallway. Great! Let’s go to the backyard to see if we can trigger the end of the game:

Input to game: go to the backyard

Output:

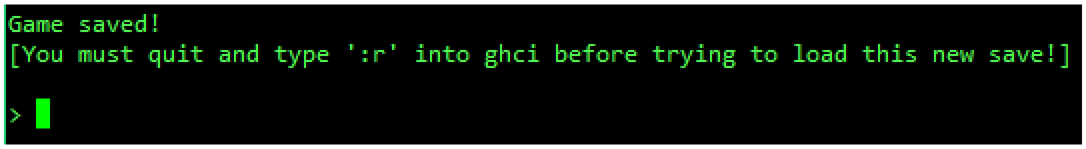


```
Standing on the back porch, you look out and see a fence surrounding a very barren backyard.
You can see a conspicuous mound of recently disturbed soil...

>
```

Perfect, we have arrived at the backyard. With the shovel in our inventory, we should be able to examine the mound to end the game. But before we do that, let's try to save the game:

Input to game: save

Output:

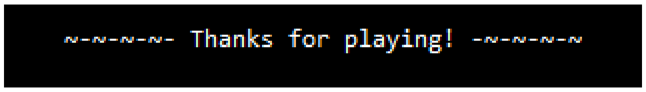


```
Game saved!
[You must quit and type ':r' into ghci before trying to load this new save!]

> 
```

It looks like it worked, but to be sure, let's follow the directions by first quitting the game with the quit command:

Input to game: quit

Output:

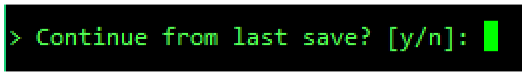


```
    ~-~-~-~- Thanks for playing! -~-~-~-~
```

This text is accompanied by the return of the GHCI prompt, and it's worth noting that the color scheme has changed back (it was changed after starting the game with the start function). Having now reloaded the modules, let's try to start the game again. We should be greeted by a prompt to load the game this time:

Input to GHCI: start

Output:

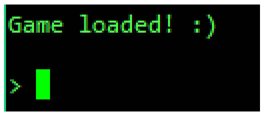


```
> Continue from last save? [y/n]: 
```

Perfect! The game recognized our save! Let's enter 'y' to load the file and see if it actually loads it correctly:

Input to game: y (and then pressed the Enter key on keyboard)

Output:

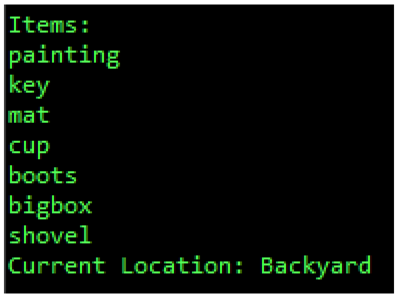


```
Game loaded! :)

>
```

It says the game was loaded, but to be sure, let's consult the help menu:

Input to game: help

Output: (Again omitting irrelevant info)

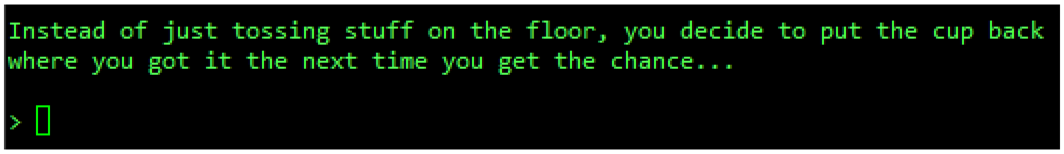


```
Items:
painting
key
mat
cup
boots
bigbox
shovel
Current Location: Backyard
```

Referring to the last time we looked at the inventory before saving, we have the same items, but were in the basement. Since we since moved to the backyard, it makes sense that our Current Location reflects that. So, the save and load functionality seems to be working pretty well. While we're here, let's try to drop an item and see how that goes:

Input to game: drop the cup

Output:

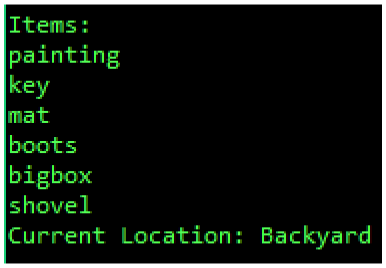


```
Instead of just tossing stuff on the floor, you decide to put the cup back
where you got it the next time you get the chance...

>
```

Although the wording perhaps doesn't make it very clear, the cup should not be in our inventory anymore. Let's just verify that quickly:

Input to game: help

Output: (Again omitting irrelevant info)

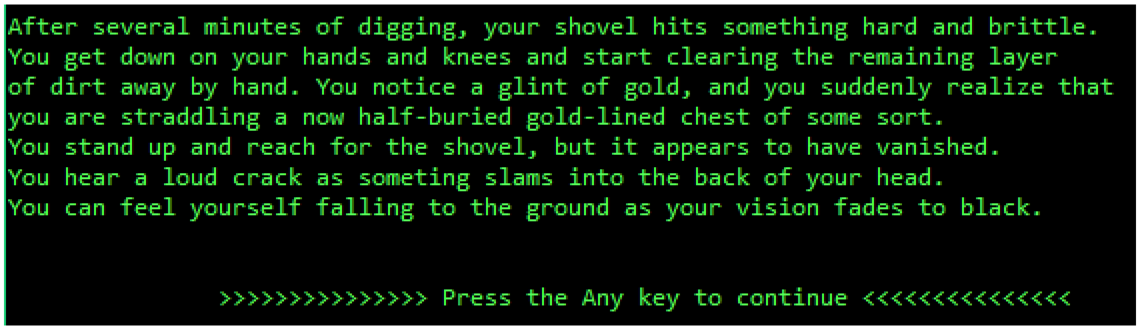


```
Items:
painting
key
mat
boots
bigbox
shovel
Current Location: Backyard
```

Comparing this to the previous inventory readout shows us that the cup has indeed been removed from our inventory. Great! Now, let’s attempt to end the game by examining the mound in the backyard:

Input to game: look at mound

Output:



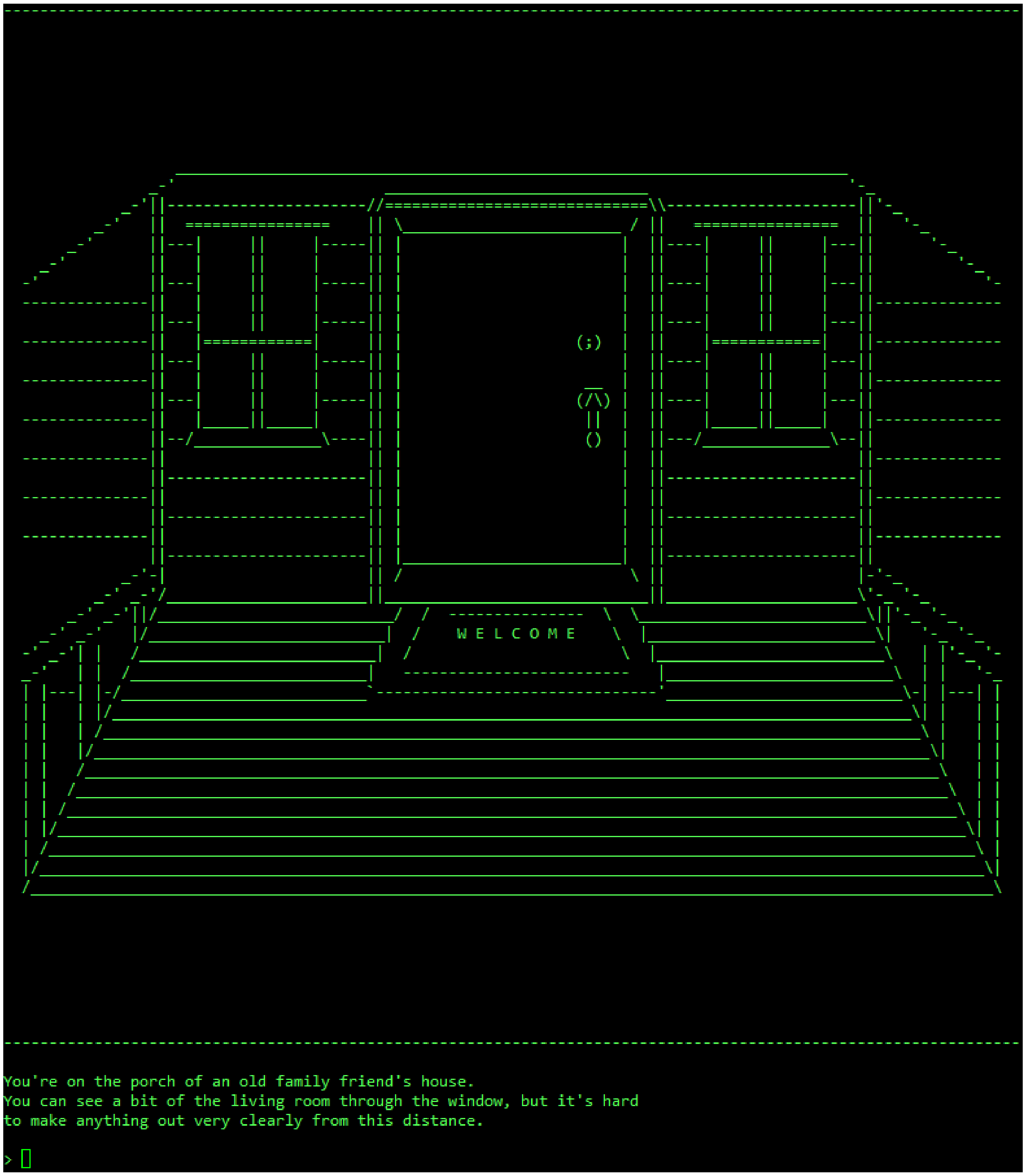
```
After several minutes of digging, your shovel hits something hard and brittle.
You get down on your hands and knees and start clearing the remaining layer
of dirt away by hand. You notice a glint of gold, and you suddenly realize that
you are straddling a now half-buried gold-lined chest of some sort.
You stand up and reach for the shovel, but it appears to have vanished.
You hear a loud crack as someting slams into the back of your head.
You can feel yourself falling to the ground as your vision fades to black.


               >>>>>>>>>>>>>>> Press the Any key to continue <<<<<<<<<<<<<<<
```

This is precisely what was expected, and the game appears to now be over. Pressing the Enter key on the keyboard gets us the same outcome as when we used the quit command earlier, which seems like a reasonable response to the game ending naturally.

In conclusion, this was clearly not thorough testing, but I mainly just wanted to demonstrate my testing procedure. Of course, I have been doing far more in-depth and thorough playtesting throughout development.

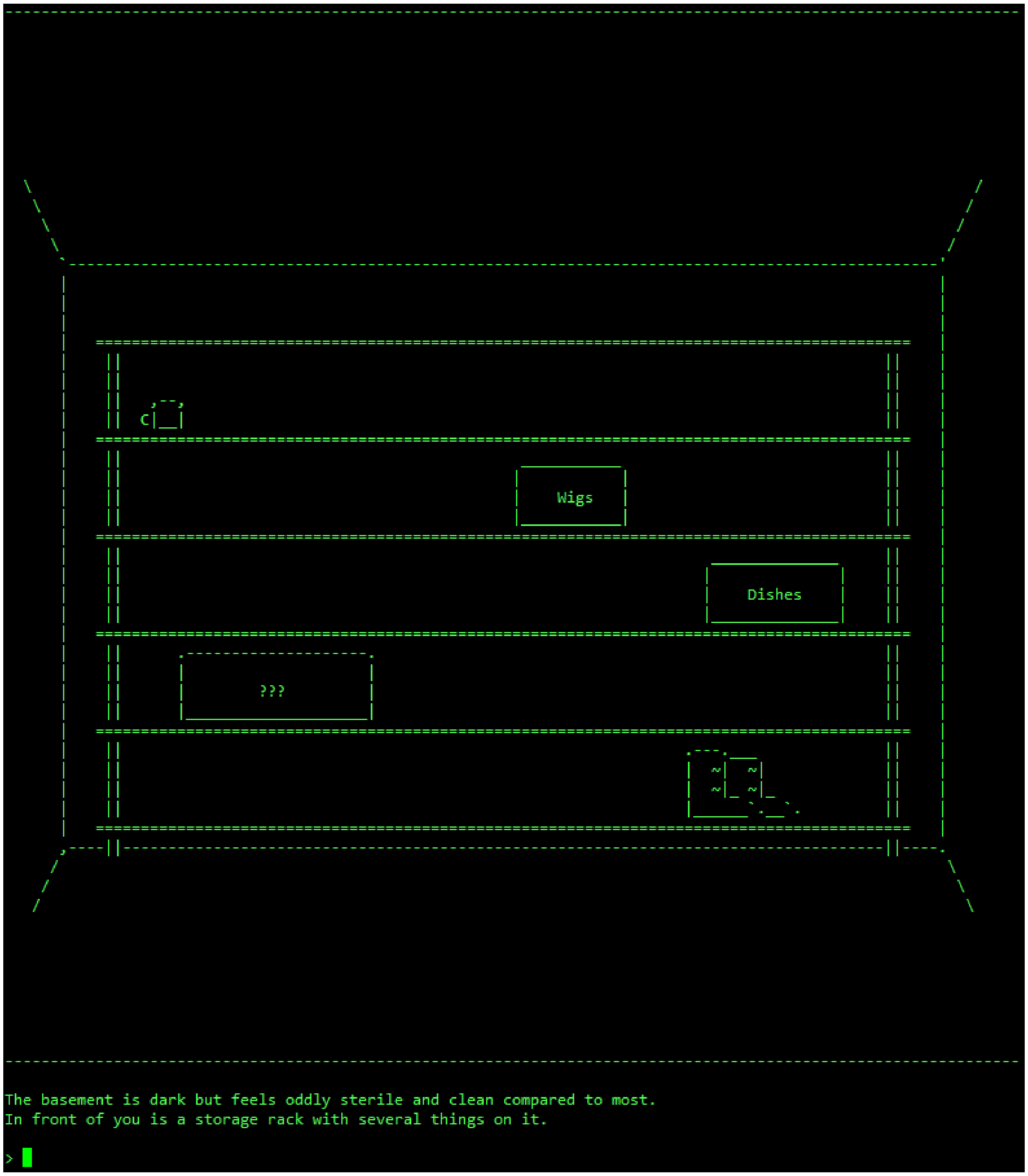
Appendix B - Screenshots

The opening scene of the game:



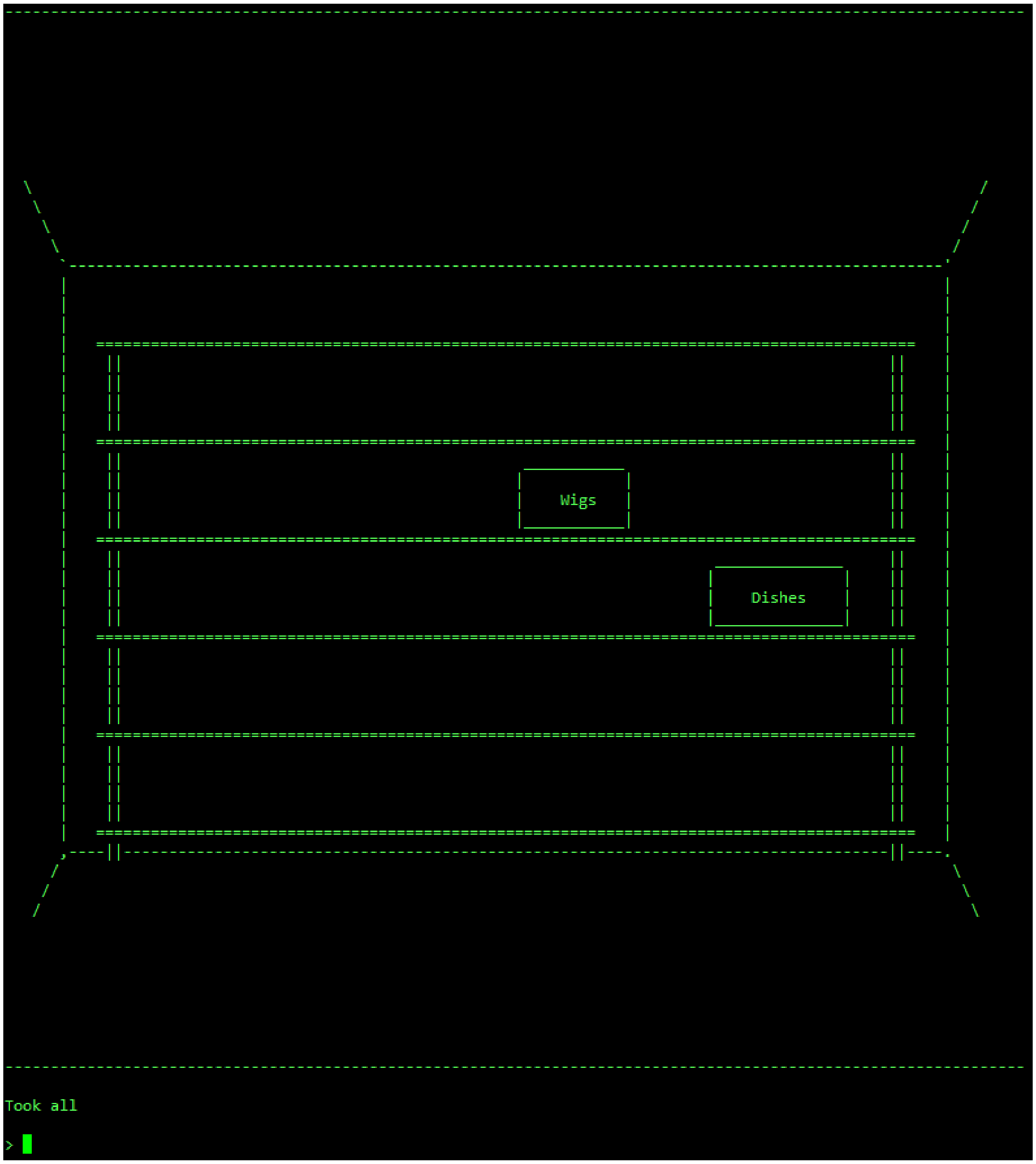
```
You're on the porch of an old family friend's house.
You can see a bit of the living room through the window, but it's hard
to make anything out very clearly from this distance.

>
```

The basement of the house, as it first appears to the player:



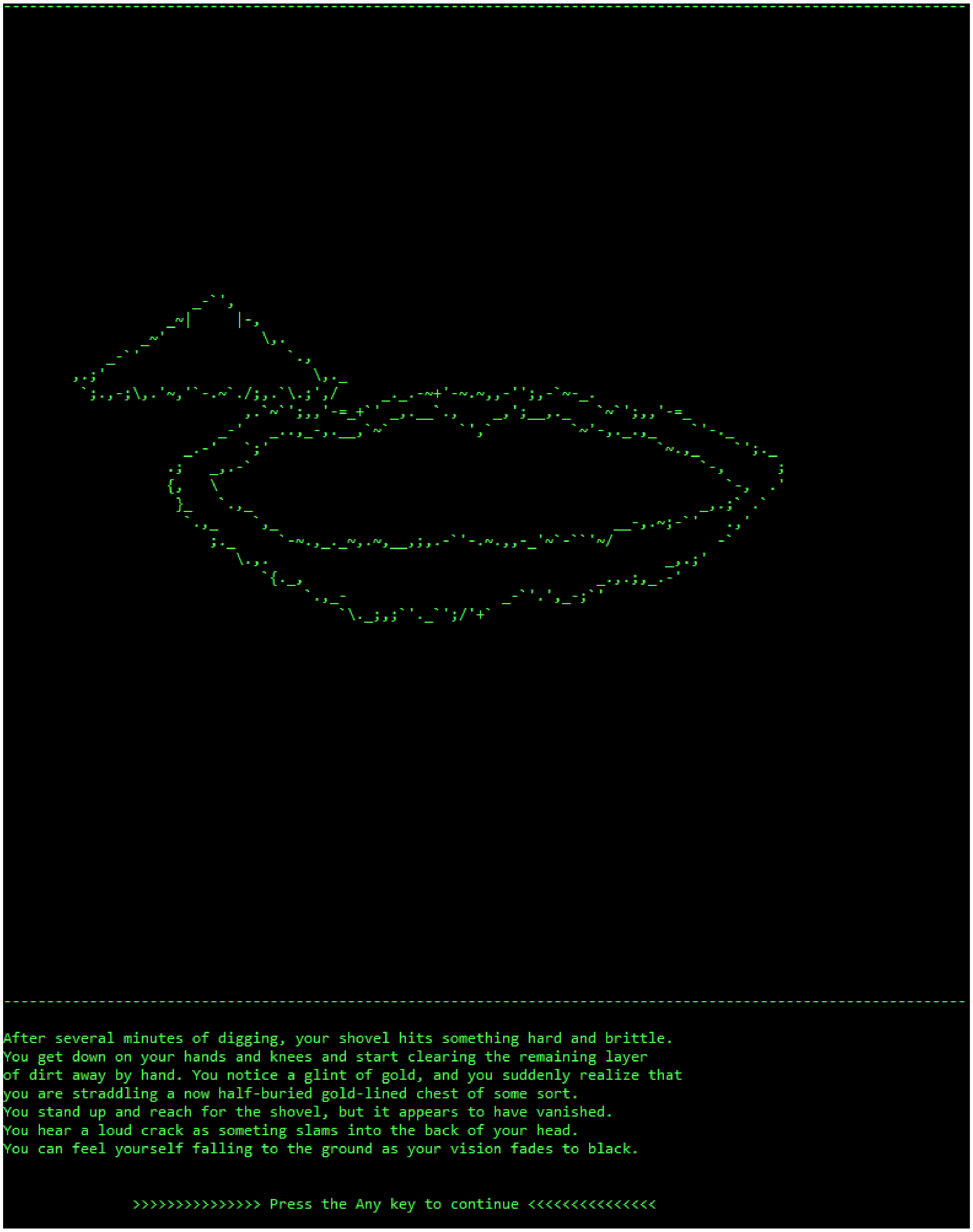
```
The basement is dark but feels oddly sterile and clean compared to most.
In front of you is a storage rack with several things on it.

>
```

Basement after the *take all* command has been given by the user:



```
Took all
>
```

Final scene of the game, which takes place in the backyard:



```
After several minutes of digging, your shovel hits something hard and brittle.
You get down on your hands and knees and start clearing the remaining layer
of dirt away by hand. You notice a glint of gold, and you suddenly realize that
you are straddling a now half-buried gold-lined chest of some sort.
You stand up and reach for the shovel, but it appears to have vanished.
You hear a loud crack as someting slams into the back of your head.
You can feel yourself falling to the ground as your vision fades to black.


               >>>>>>>>>>>>>>>> Press the Any key to continue <<<<<<<<<<<<<<<<
```

Appendix C - Code

[*Note: In an effort to make this section more useful, I've listed out an index of the core functions in alphabetical order, along with the page that they can be found on. I've also done the same with everything else that shows up in the source code. The hope is that this will enable the reader to look up referenced code for context relatively quickly.*] **FORMAT:** fncName[# w/same name] (Page # found on)

Functions

Miscellaneous

Source

A text-based adventure game made in Haskell by Alec Deschene

```haskell
> module Main where

> import Commands  -- Commands.lhs
> import Save      -- Save.lhs
> import Screens   -- Screens.lhs
> import Output    -- Output.lhs
> import Helpers   -- Helpers.lhs
> import Data.Char
> import System.IO
> import System.Console.Readline -- Rich terminal IO (binding of GNU library)
> import System.Console.ANSI -- Useful terminal manipulation functionality
```

In order to make the game easier and more intuitive to start, we have the player start the game by typing 'start', which calls start, sets up game colors, loads start screen, and calls the main function to deal with saves.

```haskell
> start :: IO ()
> start = do setSGR [ SetColor Foreground Vivid Green, SetColor Background Dull Black ]
>            loadScreen "Menu" []
>            main
```

Introductory function, serves as a kind of menu, looks for a save file, and asks the player if they want to load the game if it finds one.

```haskell
> main :: IO ()
> main  = case loadGame save of
>           Just loadState ->
>             do answer <- readline "\n> Continue from last save? [y/n]: "
>                case answer of
>                  Nothing -> freshStart -- Start game at porch with empty inventory
>                  Just "" -> freshStart
>                  Just s  ->
>                    case (toLower . head) s of
>                      'y' -> play save "Game loaded! :)\n"
>                      _   -> freshStart
>           Nothing       -> do hideCursor; readline startText; showCursor; freshStart
>   where freshStart = do moveTo ([],"Porch") ("porch")
```

```haskell
> module GTypes where
```

GState contains the game state, which consists of a list of the items that the player currently has in their inventory and the room that the player is currently in.

```haskell
> type GState   = ([Item], Room)
> type Item     = String -- Physical item that can be owned by the player
> type Room     = String -- Physical location that the player can be in
> type Object   = String -- Same as an item, but can't be held or taken by player
> type Prompt   = String -- The text displayed to the user to prompt for input
> type Command  = String -- The player's desired command
> type Argument = String -- What the player wants the command to act upon
```

```haskell
> module Save where
> save = (["key","lighter","mat"],"Porch")
```

```
> module Helpers where

> import Rooms     -- Rooms.lhs
> import Screens   -- Screens.lhs
> import GTypes    -- GTypes.lhs
> import Data.Maybe
> import Data.Char
> import Data.List
> import System.Console.ANSI -- Useful terminal manipulation functionality
```

In order to avoid runtime errors, I decided to implement a safer version of the !! operator as the function getElem. Returns Just a if n is in bounds or Nothing if n is OOB.

```
> getElem                                :: Int -> [a] -> Maybe a
> getElem n xs | n >= 0 && n < length xs = Just (xs !! n)
>              | otherwise               = Nothing
```

We need to take the text entered by the user after the initial command, split it up into individual words, search through for the FIRST relevant keyword (an item name, object name, etc.) and then return that keyword, or Nothing if none of the arguments were meaningful.

```
> parseArgs  :: String -> String
> parseArgs s = let x = find (`elem` keywords) tailS in
>                 case x of
>                      Just x  -> x
>                      Nothing -> unwords tailS
>   where tailS = (tail . words) s
```

To maintain consistency when working with item names, we need to convert every item name referenced to an all lowercase version of that name before doing anything with it.

```
> lowerStr  :: Item -> Item
> lowerStr i = [toLower x | x <- i]
```

Defining a function to check whether an item is present in the current room allows us to more strictly define individual rooms and their boundaries.

```
> isInRoom :: Item -> Room -> Bool
> isInRoom i r | i `elem` currRoom = True
>              | otherwise         = False
>                where currRoom = case r of
>                                   "Porch"    -> pItems++pObjs
>                                   "Living"   -> lrItems++lrObjs
>                                   "Bathroom" -> brItems++brObjs
>                                   "Hallway"  -> hwItems++hwObjs
>                                   "Closet"   -> clItems++clObjs
>                                   "Backyard" -> byItems++byObjs
>                                   "Basement" -> bmItems++bmObjs
>                                   _          -> []
```

We need to be able to determine which rooms are accessible from a given room, so that we can control the player's movement in a realistic way.

```
> isAdjTo     :: Room -> Room -> Maybe Int
> isAdjTo r r' = case find (==r') validRooms of
>                     Just e  -> findIndex (==e) roomKeywords
>                     Nothing -> Nothing
>   where validRooms = case r of
>                           "Porch"    -> pAdjRooms
>                           "Living"   -> lrAdjRooms
>                           "Bathroom" -> brAdjRooms
>                           "Hallway"  -> hwAdjRooms
>                           "Closet"   -> clAdjRooms
>                           "Backyard" -> byAdjRooms
>                           "Basement" -> bmAdjRooms
>                           _          -> []
```

Helpers for finding correct room when player attempts to move based on direction instead of by room name.

```
> findFwd    :: Room -> Argument -> Room
> findFwd r d = case r of
>                 "Porch"   -> "living"
>                 "Living"  -> "hallway"
>                 "Hallway" -> "backyard"
>                 _         -> "BAD_ROOM"

> findBwd    :: Room -> Argument -> Room
> findBwd r d = case r of
>                 "Living"   -> "porch"
>                 "Bathroom" -> "hallway"
>                 "Hallway"  -> "living"
>                 "Closet"   -> "living"
>                 "Backyard" -> "hallway"
>                 "Basement" -> "hallway"
>                 _          -> "BAD_ROOM"

> findRgt    :: Room -> Argument -> Room
> findRgt r d = case r of
>                 "Living"  -> "closet"
>                 "Hallway" -> "basement"
>                 _         -> "BAD_ROOM"

> findLft    :: Room -> Argument -> Room
> findLft r d = case r of
>                 "Hallway" -> "bathroom"
>                 _         -> "BAD_ROOM"
```

Find the list of items in a given room and return it for when the player wants to "take all" in a room.

```
> findAllItems  :: Room -> [Item]
> findAllItems r = case r of
>               "Porch"    -> pItems
>               "Living"   -> lrItems
>               "Bathroom" -> brItems
>               "Hallway"  -> hwItems
>               "Closet"   -> clItems
>               "Backyard" -> byItems
>               "Basement" -> bmItems
```

Clears the screen and draws a new screen for whatever room or scene version the player is in. Some rooms have different screen versions to correspond to what the player has done in the room.

```
> loadScreen     :: Room -> [Item] -> IO ()
> loadScreen r is = do clearScreen
>                      case r of
>                        "Menu"     -> putStr mainMenuS
>                        "Porch"    -> if "mat" `elem` is
>                                        then if "key" `elem` is
>                                          then putStr porchS3
>                                        else putStr porchS2
>                                      else putStr porchS1
>                        "Living"   -> if "painting" `elem` is
>                                        then if "vase" `elem` is
>                                               then putStr livingRmS4
>                                             else putStr livingRmS2
>                                        else if "vase" `elem` is
>                                            then putStr livingRmS3
>                                      else putStr livingRmS1
>                        "Hallway"  -> putStr hallwayS
>                        "Bathroom" -> if "paper" `elem` is
>                                        then putStr bathroomS2
>                                      else putStr bathroomS1
>                        "Closet"   -> if "shoebox" `elem` is
>                                        then putStr closetS2
>                                      else putStr closetS1
>                        "Basement" -> if "cup" `elem` is
>                                        then if "bigbox" `elem` is
>                                               then if "shovel" `elem` is
>                                                      then if "boots" `elem` is
>                                                             then putStr basementS10
>                                                           else putStr basementS4
>                                                    else if "boots" `elem` is
>                                                           then putStr basementS9
>                                                         else putStr basementS3
>                                             else if "boots" `elem` is
>                                                    then putStr basementS8
>                                                  else putStr basementS2
>                                      else if "boots" `elem` is
>                                             then if "bigbox" `elem` is
>                                                    then if "shovel" `elem` is
>                                                           then putStr basementS7
>                                                         else putStr basementS6
>                                                  else putStr basementS5
>                                           else if "bigbox" `elem` is
>                                                  then if "shovel" `elem` is
>                                                         then putStr basementS12
>                                                       else putStr basementS11
>                                                else putStr basementS1
>                        "Backyard" -> putStr backyardS
>                        "Quit"     -> putStr quitS
>                        _          -> putStr drawLine
```

```
> module Commands where

> import Screens   -- Screens.lhs
> import Rooms     -- Rooms.lhs
> import Save      -- Save.lhs
> import Helpers   -- Helpers.lhs
> import Screens   -- Screens.lhs
> import GTypes    -- GTypes.lhs
> import Output    -- Output.lhs
> import Data.Char (toLower,isSpace)
> import Data.Maybe
> import Data.List (nub)
> import System.IO
> import System.Console.Readline -- Rich terminal IO (binding of GNU library)
> import System.Console.ANSI -- Useful terminal manipulation functionality
```

The main game loop. Prompts the user for input and passes it to the exeCmd function. Also maintains history of user input that the player can recall with the arrow keys.

```
> play         :: GState -> Prompt -> IO ()
> play (is,r) p = do
>      loadScreen r is
>      l <- readline (p++"\n> ")
>      case l of
>        Nothing    -> play (is,r) noCommand
>        Just ""    -> play (is,r) noCommand
>        Just input -> do
>         addHistory input -- Enables player to use up/down arrows to look at past input lines
>          exeCmd (is,r) input
```

The readCmd function isn't named very intuitively, because all it really does is take in a command that was entered by the player, see if it's valid, and return it in all lowercase if it is. If it isn't a valid command, we just return "BAD_CMD" to inform the calling function.

```
> readCmd  :: Maybe String -> String
> readCmd s | (isJust s) && lowCmd `elem` cmds = lowCmd
>           | otherwise                        = "BAD_CMD"
>               where lowCmd = map toLower (fromJust s)
```

Takes user input, grabs and passes the command to readCmd, then calls the function that corresponds to the determined command, if appropriate.

```
> exeCmd :: GState -> Argument -> IO ()
> exeCmd (is,r) s =  case readCmd (getElem 0 (words s)) of
>                      c|c `elem` takeSyns     -> addItem (is,r) (parseArgs s)
>                      c|c `elem` dropSyns     -> remItem (is,r) (parseArgs s)
>                      c|c `elem` moveSyns     -> moveTo  (is,r) (parseArgs s)
>                      c|c `elem` examSyns     -> exam (is,r) (parseArgs s)
>                      c|c `elem` openSyns     -> openSsm (is,r) (parseArgs s)
>                      c|c `elem` saveSyns     -> do saveGame (is,r)
>                                                    play (is,r) gameSaved
>                      c|c `elem` loadSyns     ->
>                        case loadGame save of
>                             Just saveState -> play saveState "Game loaded! :)\n"
>                             Nothing        -> play (is,r) loadFail
>                      c|c `elem` whoaSyns     -> play (is,r) noViolence
>                      c|c `elem` helpSyns     -> helpDisp (is,r)
>                      c|c `elem` quitSyns     -> do setSGR []; loadScreen "Quit" []
>                      _ -> play (is,r) (fromMaybe "Nothing" (getElem 0 (words s))
>                                     ++" is not a valid command!\n")
```

In order to separate out the command validation process from the argument parsing, and to declutter those functions a little, we can just create a big function that takes a command and its validated arguments and figures out what to do with them:

```
> doCmd :: Command -> Argument -> GState -> IO ()
> doCmd c a (is,r) = case c of
>                      "add"   -> if a `elem` staticObjs
>                                   then play (is,r) addFail3
>                                 else play (a:is,r) (addSucc a)
>                      "drop"  -> case a of
>                                   "mat"            -> play (is,r) dropMat
>                                   "key"            -> play (is,r) dropKey
>                                   z|z `elem` boxes -> play (is,r) dropBox
>                                   _                -> play (filter (\x -> x /= a) is,r)
(dropSucc a)
>                      "examObj"  -> case a of
>                                   "mat"      -> play (is,r) eMat
>                                   "key"      -> if isForbidden (is,r) "exam" "key"
>                                                   then play (is,r) examFail
>                                                 else play (is,r) eKey
>                                   "window"   -> if "painting" `elem` is
>                                                   then if "vase" `elem` is
>                                                     then displayWait (is,r) windowS4 eWindow
>                                                    else displayWait (is,r) windowS2 eWindow
>                                                 else if "vase" `elem` is
>                                                    then displayWait (is,r) windowS3 eWindow
>                                                     else displayWait (is,r) windowS1 eWindow
>                                   "lighter"  -> play (is,r) eLighter
>                                   "painting" -> displayWait (is,r) paintingS ePainting1
>                                   "vase"     -> play (is,r) eVase1
>                                   "door"     -> play (is,r) eDoor
>                                   "doors"    -> play (is,r) eDoors
>                                   "toilet"   -> play (is,r) eToilet
>                                   "sink"     -> play (is,r) eSink
>                                   "mirror"   -> play (is,r) eMirror
>                                   "shower"   -> play (is,r) eShower
>                                   "paper"    -> play (is,r) ePaper
>                                   "box"      -> if r == "Closet"
>                                                   then play (is,r) eShoeBox
>                                                 else play (is,r) eQBox
>                                   "clothes"  -> play (is,r) eClothes
>                                   "fence"    -> play (is,r) eFence
>                                   z|z `elem` dirtSyns -> if "shovel" `elem` is
>                                                            then digUp (is,r)
>                                                         else displayWait (is,r) dirtS1 eDirt
>                                   "cup"      -> play (is,r) eCup
>                                   "boots"    -> play (is,r) eBoots
>                                   "shovel"      -> if isForbidden (is,r) "exam" "shovel"
>                                                   then play (is,r) examFail
>                                                 else play (is,r) eShovel
>                                   _          -> play (is,r) (notExamable a)
>                      "examItem"  -> case a of
>                                    "mat"      -> play (is,r) eMat
>                                    "key"      -> play (is,r) eKey
>                                    "lighter"  -> play (is,r) eLighter
>                                    "painting" -> play (is,r) ePainting2
>                                    "vase"     -> play (is,r) eVase2
>                                    "paper"    -> play (is,r) ePaper
>                                    "shoebox"  -> play (is,r) eShoeBox
>                                    "bigbox"   -> play (is,r) eQBox
>                                    "cup"      -> play (is,r) eCup
>                                    "boots"    -> play (is,r) eBoots
>                                    "shovel"   -> play (is,r) eShovel
>                                    _          -> play (is,r) (notExamable a)
>                       "open"  -> case a of
>                                   "door" -> case r of
>                                               "Porch" -> if isForbidden (is,r) "open" "door"
>                                                              then play (is,r) doorLocked
>                                                            else moveTo (is,r) ("living")
>                                               "Living" -> moveTo (is,r) ("closet")
>                                               "Bathroom"    -> moveTo (is,r) ("hallway")
>                                               "Hallway"     -> play (is,r) hwAmbiguity
>                                               "Closet"      -> moveTo (is,r) ("hallway")
>                                               "Basement"    -> moveTo (is,r) ("hallway")
>                                               "Backyard"    -> moveTo (is,r) ("hallway")
>                                   _      -> play (is,r) (notOpenable a)
>   where staticObjs =
["door","doors","window","toilet","sink","mirror","shower","clothes","fence","dirt"]
>         boxes = ["box","bigbox","shoebox"]
>         dirtSyns = ["dirt","soil","mound"]
>         digUp (is,r) = do displayWait (is,r) dirtS2 digText
```

In order to add an item to the list of player's items in the game state, we must check if the player doesn't have it already, if the item is even in the room with the player, and if the player is trying to "take all", and then react accordingly.

```
> addItem         :: GState -> Item -> IO ()
> addItem (is,r) i = if lowi `elem` is
>                      then play (is,r) (addFail1 lowi) -- Already in inventory: Fail
>                    else if lowi `isInRoom` r == False
>                           then if (dropWhile isSpace lowi) == ""
>                                  then play (is,r) addFail3 -- No text after cmd: Fail
>                                else if lowi == "all"
>                                       then takeAll (is,r)
>                                     else if lowi == "box"
>                                            then case r of
>                                                   "Closet"   -> doCmd "add" "shoebox" (is,r)
>                                                   "Basement" -> doCmd "add" "bigbox" (is,r)
>                                                   _          -> play (is,r) (addFail2 lowi)
>                                          else play (is,r) (addFail2 lowi)
>                         else case isForbidden (is,r) "add" lowi of
>                           True  -> play (is,r) addFail3 -- Contextually forbidden item: Fail
>                           False -> doCmd "add" lowi (is,r) -- Item is valid: Success
>   where lowi = lowerStr i
>         takeAll (is,r) = play ((nub $ is++(findAllItems r)),r) "Took all\n"
```

Checks to make sure the player isn't trying take items or do things that are not allowed in the current context. Applies to multiple commands.

```
> isForbidden           :: GState -> Command -> Argument -> Bool
> isForbidden (is,r) c a = case c of
>                           x|x `elem` ["add","exam"] -> case a of
>                             "key"    -> "mat" `notElem` is
>                             "shovel" -> "bigbox" `notElem` is
>                             _     -> False
>                           "open" -> case a of
>                                       "door" -> "key" `notElem` is
>                                       _      -> False
>                           "move" -> case a of
>                              y|y `elem` ["living","forward"] -> "key" `notElem` is
>                              _                               -> False
>                           _      -> False
```

If we can add an item, we need a way to remove an item, but some items are too crucial to allow the player to drop because it would slightly break the game, so we check for that in doCmd. Only fails directly if the item isn't in the player inventory.

```
> remItem         :: GState -> Item -> IO ()
> remItem (is,r) i = if lowi `elem` is || lowi == "box"
>                       then doCmd "drop" lowi (is,r)
>                    else play (is,r) dropFail -- Item not in inventory: Fail
>   where lowi = lowerStr i
```

Command to move from one room to another. Allows the player to move to adjacent rooms by mentioning them by name or to move around via directional keywords.

```
> moveTo          :: GState -> Room -> IO ()
> moveTo (is,r) r' = case r `isAdjTo` r' of
>                        Just x  -> if isForbidden (is,r) "move" r'
>                                     then play (is,r) doorLocked -- Door is locked; No key
>                                   else play (is,(rooms !! x)) (roomPrompt !! x)
>                        Nothing -> if r' `elem` dirKeywords
>                                     then moveDir (is,r) r'
>                                   else play (is,r) moveFail -- In wrong room: Fail
>   where moveDir (is,r) d = case d of
>                              "forward" -> moveTo (is,r) (findFwd r d)
>                              "back"    -> moveTo (is,r) (findBwd r d)
>                              "right"   -> moveTo (is,r) (findRgt r d)
>                              "left"    -> moveTo (is,r) (findLft r d)
```

Examining something is either looking at an item, looking at an object, or look at the player's general surroundings. Respond differently for items in vs out of inventory.

```
> exam         :: GState -> Object -> IO ()
> exam (is,r) i = if lowi `elem` is
>                   then doCmd "examItem" lowi (is,r)
>                 else if lowi `isInRoom` r || lowi == "box"
>                        then doCmd "examObj" lowi (is,r) -- Success
>                      else moveTo (is,r) (lowerStr r) -- Examine current room by default
>   where lowi = lowerStr i
```

Opening is supposed to be applied to items and objects in general, but is currently only applicable to doors, which essentially just results in moving to the room on the other side.

```
> openSsm         :: GState -> Object -> IO ()
> openSsm (is,r) i = doCmd "open" lowi (is,r)
>   where lowi = lowerStr i
```

This is a workaround to not have to deal with IO from readFile directly. Writes to and loads from a module SavedGame.lhs, meaning the game must be recompiled before a new save will be recognized and usable. Not exactly ideal, but it works for now...

```
> saveGame       :: GState -> IO ()
> saveGame (is,r) = writeFile "./Save.lhs"
>                        ("> module Save where\n\n> save = ("++(show is)++","++(show r)++")")
```

Before loading a game we determine whether or not there even is a saved game in SavedGame.lhs. We compare to the default game state in order to see if loading would result in a change from default.

```
> loadGame :: GState -> Maybe GState
> loadGame (is,r) = if is /= [] || r /= "Porch"
>                     then Just save
>                    else Nothing
```

Display a temporary informational screen and then wait for user input to continue.

```
> displayWait               :: GState -> String -> String -> IO ()
> displayWait gs screen text =
>             do clearScreen; putStr screen; putStr text; hideCursor;
>                readline genericPrompt; showCursor
>                if screen == dirtS2
>                  then exeCmd gs "quit" -- GAME OVER
>                else play gs "Well, that was exciting, eh?\n"
```

Displays the help menu. Displays command info and current inventory and location for player.

```
> helpDisp       :: GState -> IO ()
> helpDisp (is,r) =
>      do clearScreen; putStr helpText; displayCmds; displayState (is,r);
>         hideCursor; readline helpPrompt; showCursor; play (is,r) "Hope that helped ;)\n\n"
```

Dislaying the information contained within the game state may be called with a user command at some point. Currently shows the list of items and then prints the room below that.

```
> displayState       :: GState -> IO ()
> displayState ([],r) = putStrLn ("\nItems: None\n\n"++"Current Location: "++r)
> displayState (is,r) = putStrLn ("\nItems:\n"++(unlines is)++"Current Location: "++r)
```

Using displayCmds with the play function, we can display all the possible commands in a nice format to the player when they invoke the "help" (or "?") command. We do this using the cmds and dCmds lists.

```
> displayCmds :: IO ()
> displayCmds  = putStrLn (helpCmdTitle++(unlines hCmds))
```

Basic commands and their synonyms, if any:

```
> takeSyns = ["take","t","grab","pick","steal"]
> dropSyns = ["drop","d"]
> moveSyns = ["move","m","walk","go","enter","cd"]
> examSyns = ["examine","e","look","view","check","read","inspect"]
> openSyns = ["open","o"]
> saveSyns = ["save","s"]
> loadSyns = ["load","l"]
> whoaSyns = ["punch","kick","bite","kill","stab","beat","die","murder"]
> helpSyns = ["help","h","?","how","what"]
> quitSyns = ["quit","q","exit","shutdown"]
```

cmds is a list of strings that represents the possible valid commands that a player can enter. hCmds is a helper list for zipping with cmds when displaying command usage to player.

```
> cmds  = takeSyns++dropSyns++moveSyns++examSyns++openSyns++saveSyns
>         ++loadSyns++whoaSyns++helpSyns++quitSyns

> hCmds = ["take [item] ------------ Take item, put in your inventory"
>         ,"drop [item] ------------ Remove item from inventory forever"
>         ,"go [direction/room] ---- Move from current room to an adjacent room"
>         ,"look at [object/item] -- Take a closer look at something"
>         ,"open [door/item] ------- Open and enter door / Open an item or object"
>         ,"save (s) --------------- Save your game (overwrites last save)"
>         ,"load (l) --------------- Load saved game [Must restart game to load a new save!]"
>         ,"help (?) --------------- Display this list of commands..."
>         ,"quit (q) --------------- Quit the game (Save first!)"]
```

```
> module Rooms where

> import GTypes -- GTypes.lhs
> import Data.Char (toLower)
> import Data.List (find,findIndex)
```

Essentially, there are two versions of the rooms used throughout the program. These lists represent those two versions. I'm not sure how or why this is the case, but it should be changed.

```
> rooms = ["Porch","Living","Bathroom","Hallway","Closet","Backyard","Basement"]
> roomKeywords = ["porch","living","bathroom","hallway","closet","backyard","basement"]
```

Directional keywords for moving intuitively based on contextual position.

```
> dirKeywords = ["forward","back","left","right"]
```

For each room, we want to keep a list of all the obtainable items:

```
> univItems = ["all","box"]
> pItems    = ["key","lighter","mat"]
> lrItems   = ["painting","vase"]
> brItems   = ["paper"]
> hwItems   = []
> clItems   = ["shoebox"]
> byItems   = []
> bmItems   = ["cup","boots","bigbox","shovel"]
```

It's useful to keep a list of the interactable objects in each room:

```
> pObjs  = ["door","key","lighter","mat","window"]
> lrObjs = ["door","painting","vase"]
> brObjs = ["toilet","sink","mirror","shower","paper"]
> hwObjs = ["doors"]
> clObjs = ["clothes","shoebox"]
> byObjs = ["fence","dirt","soil","mound"] -- Includes dirt object synonyms
> bmObjs = ["cup","boots","bigbox","shovel"]
```

Another useful set of lists to have available is that of the lists of
rooms that can be accessed from each room:

```
> pAdjRooms  = ["porch","living"]
> lrAdjRooms = ["living","porch","hallway","closet"]
> brAdjRooms = ["bathroom","hallway"]
> hwAdjRooms = ["hallway","living","bathroom","basement","backyard"]
> clAdjRooms = ["closet","living"]
> byAdjRooms = ["backyard","hallway"]
> bmAdjRooms = ["basement","hallway"]
```

Although having the individual lists is great, it's also useful to have a list of every (possibly) valid item, object, and room regardless of context:

```
> keywords = pItems++lrItems++brItems++hwItems++clItems++byItems++bmItems
>            ++pObjs++lrObjs++brObjs++hwObjs++clObjs++byObjs++bmObjs
>            ++roomKeywords++dirKeywords++univItems
```

Text that is displayed when player first enters a room or uses the look command w/o arguments.

```
> roomPrompt = [
>   "You're on the porch of an old family friend's house.\n"
>   ++"You can see a bit of the living room through the window, but it's hard\n"
>   ++"to make anything out very clearly from this distance.\n",
>   "Everything seems very clean and well taken care of.\nThe house is completely "
>   ++"silent...no signs of life.\n",
>   "The bathroom, like everything else, is immaculate, and smells vaguely of lemony "
>   ++"disinfectant.\n",
>   "There are three doors. You can feel a draft coming from the end of the hallway,\n"
>   ++"but other than that, everything seems almost disturbingly ordinary.\n",
>   "The closet is filled with clothes that are placed in identical dress-holders, which "
>   ++"seems odd.\n",
>   "Standing on the back porch, you look out and see a fence surrounding a very barren "
>   ++"backyard.\nYou can see a conspicuous mound of recently disturbed soil...\n",
>   "The basement is dark but feels oddly sterile and clean compared to most.\n"
>   ++"In front of you is a storage rack with several things on it.\n"
>              ]
```

```
> module Output where
```

Strings for use with the play function:

```
> addSucc i   = "You swiftly spirit away the "++i++"...\n"
> addFail1 i = "You reach into your pack and pull out the "++i++".\nYou stare at it for a
moment "
>               ++"before realizing that you have more important things to do.\n"
> addFail2 i = "You look away for a second and when you look back, the "++i++" appear to
have\n"
>               ++"vanished before your very eyes.\nPerhaps there was never anything there in
the first place...\n"
> addFail3   = "You reach out and snatch at the empty space in front of you.\nYou're not sure
why, though..."
> dropSucc i  = "Instead of just tossing stuff on the floor, you decide to put the "++i++"
back\nwhere you got it "
>               ++"the next time you get the chance...\n"
> dropFail    = "Now now, dropping something that you haven't picked up is likely to break
something!\n"
> dropMat     = "As crazy as it sounds, you've grown very fond of the Mat and just find it too
hard to let go.\n"
```

```
> dropKey     = "On second thought, you realize that throwing away a key seems a bit
irresponsible.\n"
> dropBox     = "You contemplate tossing it, but boxes are just so useful! What if you need to
store something?\n"
> noCommand   = "You stare into space briefly and then raise your arm to check your
watch,\nonly "
>               ++"to remember that you don't wear a watch\nas you stare at your bare wrist
awkwardly...\n"
> gameSaved   = "Game saved!\n[You must quit and type ':r' into ghci before trying to load
this new save!]\n"
> loadFail    = "Load failed: There appears to be no saved game!\n"
> noViolence  = "Hey, come on now, there's no need to get violent :)\n"
> moveFail    = "You try to start walking but your legs won't budge!\nAs it turns out"
>               ++",you were actually trying to move left and right at the same time.\n"
> startText   =
>   "                     >>>>>>>>>>>>>>>--- Press the ENTER KEY to Start ---
<<<<<<<<<<<<<<<\n\n"
> helpText    = "-------------------------------------- HELP MENU -----------------"
>               ++"----------------------\n\n\n\n\n"
> helpCmdTitle= "\n                              ----- List of commands "
>               ++"-----                              \n\n"
> helpPrompt  = "\n\n\n\n\n------------------------------ Press the Any key to continue "
>               ++"----------------------------\n\n"
> examFail    = "Strangely, instead of doing what you had just decided to do, you stand
violently\n"
>               ++"still and do absolutely nothing with a blinding speed.\n"
> hwAmbiguity = "You try to open the door and are suddenly overcome by tremendous
ambiguity!\n"
> notOpenable i = "You can't open a "++i++"! :O\n"
> notExamable i = "You try to look at the "++i++" but something flies into your eye "
>                 ++"and distracts you for several seconds.\n"
> doorLocked  = "You grab the doorknob and attempt to turn it.\n"
>               ++"Unsurprisingly, it is locked and doesn't budge.\n"
> genericPrompt = "\n\n              >>>>>>>>>>>>>>> Press the Any key to continue "
>                 ++"<<<<<<<<<<<<<<<             \n"
> digText = "After several minutes of digging, your shovel hits something hard and brittle.\n"
>           ++"You get down on your hands and knees and start clearing the remaining layer\n"
>           ++"of dirt away by hand. You notice a glint of gold, and you suddenly realize
that\n"
>           ++"you are straddling a now half-buried gold-lined chest of some sort.\n"
>           ++"You stand up and reach for the shovel, but it appears to have vanished.\n"
>           ++"You hear a loud crack as someting slams into the back of your head.\n"
>           ++"You can feel yourself falling to the ground as your vision fades to black.\n"
```

Examination info for things:

---Items---

```
> eMat = "It looks to be an almost brand-new welcome mat that reads \"WELCOME\"...\nGo
figure...\n"
> eKey = "A standard looking house-key that was hidden in a very clever and not-at-all-cliche
way...\n"
> eLighter = "It's a green lighter that looks well used.\nIt has a pungent smell to it.\n"
> ePainting2 = "It's a very odd-looking painting.\nYou just hope it's a caricature and not a
portrait.\n"
> eVase2 = "It's kind of awkward to avoid spilling the contents on the floor, but the flowers
sure\n"
```

```
>          ++"do smell nice ^_^\n"

---Objects---

> eWindow = "You press your face against the glass and stare very intrusively through the
window.\n"
>           ++"Your absolute brazenness is rewarded by an impeccable view of a drab and
uninteresting interior.\n"
> ePainting1 = "The painting, although aesthetically offensive, seems oddly inticing.\n"
>              ++"You feel drawn toward it.\nYou wonder what it would feel like to hold it.\n"
> eDoor = "An extremely basic wooden door.\nThere's nothing of note or interest about it.\n"
> eDoors = "There are three doors: one on the right, one on the left, and one dead ahead.\n"
> eVase1 = "A vase filled with vibrant-red flowers that you can smell from where you're
standing.\n"
>         ++"You don't know what kind of flowers they are, but then again, you aren't a
Florist...\n"
> eToilet = "It's a toilet...\n"
> eSink = "A nicely-detailed and decorated bathroom sink and countertop.\n"
> eMirror = "For some reason, the mirror doesn't seem to be very reflective...\n"
> eShower = "Residual water on the shower curtain tells you that it has recently been
used...\n"
> ePaper = "A small piece of paper with the number \"4 2 4 2\" scrawled on it...\n"
> eShoeBox = "A box that once held a pair of boots, it would seem.\nThere's nothing
inside...\n"
> eClothes = "They almost look like exact copies of the same outfit...\n"
> eFence = "A very uniform and well-maintained fence that marks off the entire perimeter of
the house.\n"
> eDirt = "The mound of dirt seems like an interesting thing to examine, but the thought of
digging around\n"
>         ++"with your hands doesn't seem very inviting...\n"
> eQBox = "A strange box that, oddly enough, doesn't seem to have anything interesting in
it.\n"
> eCup = "A standard coffee cup, with the words \"WORLD'S BEST BOSS\" written on it.\n"
> eBoots = "The boots are caked in semi-dried mud, but otherwise look brand new.\n"
> eShovel = "A shovel with some residual mud half-dried on it. Looks mighty useful!\n"

-----------------------------------------------------------------------------------------------
```

[NOTE: In the interest of saving trees, I have omitted the actual ASCII art for each screen but left the short descriptions. They weren't going to look right anyways. Take a look at Appendix B for some examples of what the screens actually look like!]

```
> module Screens where
```

Basic line made of hyphens that each screen uses as top and bottom borders.

```
> drawLine = "---------------------------------------------"
>          ++"-----------------------------------------------------------------\n"
>
```

Start screen; meant to be a placeholder for a main menu screen. Credit to Roland Hangg for art.

```
> mainMenuS = ...
```

Porch where player start off the game with no items having yet been taken.

```
> porchS1 = ...
```

Porch after welcome mat has been taken; key now visible.

```
> porchS2 = ...
```

Porch after mat and key have both been taken.

```
> porchS3 = ...
```

View from front porch window with nothing inside having been taken.

```
> windowS1 = ...
```

View from front porch window with just the picture taken.

```
> windowS2 = ...
```

View from front porch window with just the vase taken.

```
> windowS3 = ...
```

View from front porch window with both vase and painting having been taken.

```
> windowS4 = ...
```

Living room with nothing taken.

```
> livingRmS1 = ...
```

Living room with just painting taken.

```
> livingRmS2 = ...
```

Living room with just vase taken.

```
> livingRmS3 = ...
```

Living room with both vase and painting taken.

```
> livingRmS4 = ...
```

View from hallway.

```
> hallwayS = ...
```

Bathroom with paper on counter.

```
> bathroomS1 = ...
```

Bathroom after paper has been taken.

```
> bathroomS2 = ...
```

Basement with nothing taken.

```
> basementS1 = ...
```

Basement with cup taken.

```
> basementS2 = ...
```

Basement with cup and box taken.

```
> basementS3 = ...
```

Basement with cup, box, and shovel taken.

```
> basementS4 = ...
```

Basement with boots taken.

```
> basementS5 = ...
```

Basement with boots and box taken.

```
> basementS6 = ...
```

Basement with boots, box, and shovel taken.

```
> basementS7 = ...
```

Basement with boots and cup taken.

```
> basementS8 = ...
```

Basement with boots, cup, and box taken.

```
> basementS9 = ...
```

Basement with boots, cup, box and shovel (all items) taken.

```
> basementS10 = ...
```

Basement with box taken.

```
> basementS11 = ...
```

Basement with box and shovel taken.

```
> basementS12 = ...
```

Closet with box.

```
> closetS1 = ...
```

Closet after box has been taken.

```
> closetS2 = ...
```

Backyard with dirt mound and fence.

```
> backyardS = ...
```

Close-up of the painting item.

```
> paintingS = ...
```

The dirt mound before being dug up.

```
> dirtS1 = ...
```

The dirt mound after being dug up; final screen of the game at this point.

```
> dirtS2 = ...
```

Screen that the player is left with when the game ends or the player quits the game.

```
> quitS = unlines [
>   drawLine,
>   "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n",
>   "                                     ~-~-~-~- Thanks for playing! -~-~-~-~",
>   "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n",
>   drawLine++"\n\n"]
```
