CS 3410: MUD

Introduction

In this assignment, you will write a simple Multi-User Dungeon, or MUD, which is a kind of multi-player text role-playing game that eventually evolved into the MMORPGs of today. The goals of this assignment include:

- the basics of Go
- to work with a simple SQL database
- to work with basic network connections
- basic multi-threaded programming
- synchronization using an event loop

The assignment is in three parts.

Part 1

In the first part you will write a simple loop to accept commands from the keyboard and process them. A big part of this step is to get a working Go programming environment and learn to use the associated tools.

Getting started

Start by installing Go on the system where you intend to work. Start at the Go documentation page:

- [https://golang.org/doc/](https://golang.org/doc/)

The “Getting Started” page talks about how to install and test a Go build system. I also recommend reading/watching “How to write Go code”, which talks more about structuring projects and using the build tools.

Go is currently in a transition. Make sure you have Go 1.10 or later, which will give you two choices for how to structure your code:

1. Create an environment variable on your system called `GOPATH` that points to the root of where your Go projects will live. For example, I use my home directory. All source code lives under the `src/` subdirectory of `GOPATH`, and compiled binaries are automatically placed in the `bin/` subdirectory.

   Within `src/`, projects are normally housed in a directory that reflects where the source code is tracked using git. For example, if my code will live on Github, I would use a path to reflect that. For a project called “mud” I would put my source in:

   ```
   /home/russ/src/github.com/russross/mud
   ```

   My home directory is `/home/russ`, and since that is also my `GOPATH`, the `src` directory is housed there. My github username is `russross`, so I put my project in `github.com/russross` and then use the name of the project as the final part of the pathname.

   When I compile the project using `go install`, it creates an executable binary called

   ```
   /home/russ/bin/mud
   ```

   Again, `/home/russ` is my `GOPATH`, so it puts the `bin` directory within that directory, and `mud` is the name of the project (taken from the name of the project directory) so `go install` uses that as the binary name.

2. If you do not wish to set your `GOPATH` variable, it will default to the directory `go` within your home directory. In the example above, the paths would be:

   ```
   /home/russ/go/src/github.com/russross/mud
   /home/russ/go/bin/mud
   ```

   The Go tools are also adding support for a new approach without `GOPATH`, but it is new enough that I suggest sticking with `GOPATH` for now.

Next, work through “A Tour of Go”, which walks you through the code language features. It can be completed in the browser, but I recommend running at least some of the exercises in your own build
environment to make sure you are comfortable with it.

**The project**

The purpose of this part is to write a little bit of Go code in the environment you have set up, and to explore the standard library a bit.

Write a command processing loop. It should:

- Print a prompt
- Let the user type a line of text
- Parse that line of text and process the command the user typed
- Repeat

Start by writing code to print the prompt (which can be anything) and accept a line of input. You will need help from the standard library, which is documented here:

- [https://golang.org/pkg/](https://golang.org/pkg/)

The `fmt` package contains code for printing formatted text. To use it, you will need to import it near the top of your source file:

```go
import "fmt"
```

or if it is one of multiple imports you can use:

```go
import {
    "fmt"
    "log"
}
```

Then to print something you would reference the package and the function:

```go
fmt.Println("Hello, world")
```

To read lines of input, I suggest starting with the `bufio` package and using the `Scanner` API. Note that the `bufio` package documentation has a complete example that does exactly what you need to do: it reads lines of input and processes each one.

Once you can read lines of input, you need to parse them to recognize commands. A command in a MUD always starts with the command itself, and then has additional text to support it if appropriate. Example commands:

- look
- look north
- sigh
- laugh
- north
- n
- say I’ll be right back
- tell alice I am at the shop

In each case the command is the first part of the line (separated by one or more space characters from the rest of the line). Look at the `strings` package to find helpful functions to parse and process strings. There are more complex parsing functions in the standard library, but `strings` will let you do simple tasks like splitting a string on whitespace into words (the `Fields` function), etc.

Next, you should implement a way to dispatch different commands in a way that lets you easily add to the list of supported commands. You can do this in many different ways, but the most straightforward way is to create a `map` from a command name to a function that processes the command. For example, if the function that processes the look command looks like:

```go
func doLook(line string) {
    // process the look command
}
```

You might have a map of this type:
When the user types a command, you would look it up in the `map`. If it is present, it is a valid command and you can call the associated function. If not, tell the user you did not understand their request.

There are other ways to implement a command dispatcher, and you are welcome to use whatever makes sense to you. However:

- Make sure each command is implemented in its own function. Do not use a giant `switch` statement with all of the commands implemented in one giant function.

- Make sure it is easy to add new commands. I suggest having a function that adds commands to the list. Something like:

```go
func addCommand(command string, action func(string)) {...}
```

Then when your code is starting up, you would register all of the commands, each with a call to `addCommand`.

In addition, MUD users end up typing a lot and expect to be able to use shortcuts. You should be able to recognize a prefix of a command as being the same as the command. So these should all be recognized as the “north” command:

- north
- n
- no
- nor
- nort

If you are using a `map`, you could add duplicate entries for all prefixes of the command. There may be multiple commands with some of the same prefixes, in which case you should be able to prioritize which one wins. More important/command commands should be able to use the shortest abbreviates. So “e” and “ea” are short for “east”, not “eat”, because moving around is more common than eating.

The examples above all show the command processing function as taking a single string parameter. This is just for demonstration purposes: you should figure out what your command processor actually needs, and you may find that the requirements will change as the project progresses.

**Part 2**

In this part you will add a simple SQL database to your MUD to store the rooms and zones that make up the world.

**Using sqlite**

Go has a standard SQL database driver ([database/sql](https://golang.org/doc/database/sql)), but to use it you must include a driver for the specific database you are using. We will use sqlite3 in this project, which is an embedded database. To install the driver, use:

```go
go get github.com/mattn/go-sqlite3
```

In one of your source files (technically it does not matter which), add the driver to your imports using:

```go
package main

import "github.com/mattn/go-sqlite3"

```

The underscore means that you will not be referring to the package directly in your code (you will work with the database/sql package of the standard library), but you need it to load so it can register itself.

To see the basics of how to use the driver, see the simple example from the driver package:


Sqlite keeps all of its data in a single file; when you open the database, you supply the name of that file.

**The world database**

The world is organized into *zones*, and each zone has some number of *rooms*. I have extracted the basic
world information from the classic MERC mud distribution so that you do not have to start from scratch. That data is supplied here:

- **world.sql**

This is a sqlite database dump. It is a text file containing a series of commands that, when executed in order, reproduce an entire database. Search through it and look at the `CREATE` statements to see the schema. The `INSERT` statements provide the data. Note that there are three tables: zones, rooms, and exits (the connections between rooms).

You can create the database from this file using the following command:

```sql
sqlite3 world.db < world.sql
```

This assumes that you have sqlite3 installed on your system (you should install it if you do not already have it).

The starting point of this world is room 3001, The Temple of Midgaard. The original data included items, mobs (short for mobile objects, another name for monsters), shops, etc., but we will only be using the basic room descriptions and connections.

**An example**

A Linux binary demonstrating the functionality is available here:

- **mud-part2-example.tar.gz**

Download it, untar it, and run it in the same directory as the `world.db` file. The supported commands include:

- north, east, west, south, up, down
- look, look north, look east, etc.
- recall (to return to the starting location)
- smile

To quit, type `ctrl-d` to end the stream of input.

**Queries**

To retrieve a kind of object from the database, you will need to run a query. You can run these by hand within the sqlite3 tool:

```sql
sqlite3 world.db
```

and then type a query, such as:

```sql
select id, zone_id, name, description from rooms where id = 3001;
```

You can change the way the data is presented, see the database schema, and various other operations as well. Type `.help` for details.

To run a query within your Go program, you need to use the database driver. See the example cited above for the basics of how to run commands. As an example, you could retrieve all of the zones using:

```sql
SELECT id, name FROM zones ORDER BY id
```

An example in the standard library documentation shows how to issue a query and read in the results using a `Rows` object:

- [https://golang.org/pkg/database/sql/#example_Rows](https://golang.org/pkg/database/sql/#example_Rows)

I suggest the following datatypes in Go:

```go
type Zone struct {
    ID int
    Name string
    Rooms []*Room
}
```
To get started, I suggest the following:

- Write a Go function that opens the database and reads a single room (use the query given earlier to load room 3001). Store the [ID], [Name], and [Description] fields in a [Room] object and prints it out.

- Add a new function to read all of the zones (see the [Rows] examples from the standard library documentation and the example from the sqlite driver library). Collect all of the zones into a map where the keys are zone IDs and the values are [Zone] pointers. Print them all out.

- Refactor your code so that `main` opens the database and starts a transaction, calls the function to read in all the zones with that transaction, then commits the transaction when it returns (or rolls it back if there is an error).

- Write a new function to read in all of the rooms. Have it accept an open transaction as a parameter and return a slice of [Room] pointers. In addition, have it accept the map of zones as a parameter. When you get a zone ID from the database, use it to find the corresponding [Zone] pointer and store it in the [Room] object. Make sure that `main` opens a transaction before calling this function, then either commits or rolls back the transaction depending on whether or not there were errors.

- Note: if you detect an error, you should return an error. Have `main` actually report the error to the console and decide what to do next (usually quitting). The functions that it calls should just return errors. If you want to add additional information, make your own error. For example:

```go
if err != nil {
    return fmt.Errorf("reading a room from the database: %v", err)
}
```

`fmt.Errorf` is a version of `Printf` that constructs an error, so this example uses it to construct an error message with useful information in it that also embeds the error message it detected.

- Write another similar function to read in all of the exits. For each one, find the room it leaves from and fill in the corresponding exit field of the room. Note that this is a rare occasion where using an array (not a slice) of objects makes sense. Every room has exactly six exits (though some may not be used), and declaring them this way embeds the storage for the exits directly in the room record, which simplifies memory a bit. Because the type has exactly six exits in the array, you never have to check the length (as you would normally do with a slice). Note that the index of the exit tells its direction. You should use:

  - 0 → north
  - 1 → east
  - 2 → west
  - 3 → south
  - 4 → up
  - 5 → down

I suggest making lookup tables that make it easy to translate from a direction number to a name and vice versa.

- Now go and integrate this into your command handler. Implement at least:

  - The movement commands “north”, “east”, “west”, “south”, “up”, and “down”.
  - A “look” command that displays the current room name, description, and exits (see my example implementation for details).
  - The ability to “look” in a direction and get the description of the exit in that direction.
A “recall” command that takes you back to the starting room, which is room 3001.

You should create a `Player` object that holds information about the player. For now, it will mainly contain the player’s location. Update your command processor to accept a pointer to the `Player` object as a parameter to each command.