Theme Song: Gotta Go Fast

Welcome to CSCI 1270! Throughout the semester, you'll implement many components of a database in Go. This assignment is meant to get you up to speed with the language and with the codebase you'll spend the semester on. Creating a strong foundation with this language is crucial to succeeding in the rest of the class; if you ever have any questions about the Go language, don't hesitate to ask a TA for support!

---

1 Background Knowledge

1.1 Golang

To get started with Go, work through A Tour of Go first, then reading through our Guide to Go to fill in any gaps. The guide has more linked to useful resources to aid and abet your understanding of the language; feel free to check them out. Even if you're already familiar with or proficient in the language, everybody benefits from a brief refresher.

1.2 The Codebase

After you clone the stencil, you should see a few folders and files. As the semester grows, you will add more packages and grow this codebase. Worry not; **each assignment is designed such that you only have to touch the files we explicitly tell you to touch**. You never have to create any new files beyond what we provide in each assignment's stencil code, and the overall codebase design is done for you. Your job throughout the semester is to finish the rest.

The stencil has a few files in the root directory, and then a few subdirectories. We'll explain every relevant part of the codebase to you now.

- `./`
  - `README.md` is an empty file for you to details your designs.
  - `.gitignore` contains a list of files for Git to ignore.
  - `Makefile` contains a list of build targets that will be useful to you. Poke through to see what commands are available - they are rather basic commands. Please don't change existing targets.
- `cmd/` contains the project's `main` packages. For now, there should only be one folder: `bumble`, which will build into an executable that you can run.
- `pkg/` contains the project packages, each one roughly corresponding to a project. In future assignments, you will be given stencil code to put in this directory. For now, you should see three subfolder: `config`, `list`, and `repl`. These are all focuses of this assignment.

You'll notice that the `cmd/bumble/main.go` file is already filled out for you, with a lot of commented out code. Essentially, when you get to the corresponding assignment, all you need to do is comment out the portions underneath and the system should play nice. In the case that our planning did not work as we hoped, we'll release a new `main.go` with each assignment. This file is the primary entrypoint into the project; peruse through it to get a sense of how your REPL code will be called.

1.3 Doubly Linked Lists

You may recall from your introductory sequence that a Linked List is a recursive data structure composed of nodes, in which each node stores a value and a pointer to the next node. A Doubly Linked List (sometimes called a double-ended queue, or deque) is the same, **except that it also stores a pointer to the previous node**. We want our deque to be efficient, so by storing pointers to both the head and the tail (first and last elements of the list), we can support constant time insertion and removal from the ends of the list.

For this assignment, don't worry enforcing acyclicity; if a loop arises, it means that the link was used poorly.

1.4 Read-Evaluate-Print-Loop (REPL)

A Read-Evaluate-Print-Loop, also known as a REPL, is the primary user interface for command line applications. When the application is run, the REPL is first provisioned with the resources it needs to fulfill user requests (*e.g.* setting up a database, reading config files, etc.). Then, it enters a loop, where it repeats the following actions until terminated using an EOF, SIGINT, SIGKILL, or SIGTERM (if you don't know what these mean, search up EOF and POSIX Signals; in this course, signal handlers are implemented for you):

- **Read** in user input from the command line,
- **Evaluate** the user's request,
- **Print** the results of the requested action, and finally,
- **Loop**.

As a concrete example, consider a database CLI. It might read input like `SELECT * FROM table`. Then it would evaluate the meaning of the command, and execute some code to get data from the requested table. Lastly, it would print out that information back to the user, and ready itself to read input again.

---

2 Assignment Spec

2.1 Overview

In this assignment, you will implement a doubly linked list to be used later on in the semester and a reusable REPL to use your linked list. The following are the files you will need to alter:

- `pkg/list/list.go`
- `pkg/repl/repl.go`

2.2 Part 1: Doubly Linked List

Relevant files: - `pkg/list/list.go`

In your stencil should be two defined structs: a `List` struct and a `Link` struct. A `List` contains pointers to the head and tail of the list, and a `Link` contains a pointer to the list it's a part of, a value, the previous link, and the next link. Some of these data fields are not necessary for a Doubly Linked List, but will prove useful when we apply this data structure later. Implement the following functions (in no particular order):

```
// Create a new list.
func NewList() *List

// Get a pointer to the head of the list.
func (list *List) PeekHead() *Link

// Get a pointer to the tail of the list.
func (list *List) PeekTail() *Link

// Add an element to the start of the list. Returns the added link.
func (list *List) PushHead(key interface{}) *Link

// Add an element to the end of the list. Returns the added link.
func (list *List) PushTail(key interface{}) *Link

// Find an element in a list given a boolean function, f, that evaluates to true on the desired
    element.
func (list *List) Find(f func(*Link) bool) *Link

// Apply a function to every element in a list in-place. f should alter Link in place.
func (list *List) Map(f func(*Link))


// Get the list this link belongs to.
func (link *Link) GetList() *List

// Get the link's value.
func (link *Link) GetKey() interface{}

// Set the link's value.
func (link *Link) SetKey(value interface{})

// Get the link's prev.
func (link *Link) GetPrev() *Link

// Get the link's next.
func (link *Link) GetNext() *Link

// Remove this link from the list.
func (link *Link) PopSelf()
```

Some notes:

- You may notice that there is no `Remove` method - the intended usage is to `Find` the link, then call `PopSelf` on it to remove it from a list.
- You may notice that some of these methods take in functions as parameters; you can use the following construct to define an inline function in Go:

```
f := func(l *Link) { /* do something */ }
```

- This is a great opportunity to get yourself acquainted with Golang testing - try it out on a `sample_test.go` file!

2.3 Part 2: REPL

Relevant files: - `pkg/repl/repl.go`

In your stencil should be a `REPL` struct containing two fields: a map of commands and a map of help strings. Our codebase will rely heavily on this REPL construct, so it's important that we build a reusable and extendible REPL. With this design, each package can return its own REPL instance, and we can combine them into one that we'll actually run. To that end, implement the following functions:

```
// Create a new REPL.
func NewREPL() *REPL

// Combine a slice of REPLs. If no REPLs are passed in,
// return a NewREPL(). If REPLs have overlapping triggers,
// return an error. Otherwise, return a REPL with the union
// of the triggers.
func CombineRepls(repls []*REPL) (*REPL, error)

// Add a command to the REPL.
func (r *REPL) AddCommand(trigger string, action func(string, *REPLConfig) error, help string)

// Print out all of the usage information (in no order).
func (r *REPL) HelpString() string

// Start the REPL loop with prompt. Hint: use bufio.NewScanner(reader)
func (r *REPL) Run(c net.Conn, clientId uuid.UUID, prompt string)
```

You'll notice another function named `RunChan` - you do **not** need to implement this function; this function will be provided to you in a later assignment.

Moreover, your REPL should have some meta commands that are available no matter which other commands you use. These commands should not be allowed to be overwritten; whether you throw an error or refuse to take commands that begin with a period is up to you. For now, implement the following meta command:

- `.help`: Prints out every command and meta command available to you (hint: use the information we injected through `AddCommand` in Part 2).

Some notes:

- We **highly** recommend defining a function to clean user input. Extra whitespace around and between tokens should be ignored.
- The `strings` library may be useful to you.
- The REPL should run print the `prompt` string on every iteration of the loop.
- The scanner should be used in a `for` loop; use `for scanner.Scan(){ /* do stuff */ }` to loop until the user inputs EOF or SIGINT, and use `scanner.Text()` to get the inputted text.
- You should pass the entire line to the first parameter in an `action` when a command is run. Don't remove the equivalent of `argv[0]` - pass the whole string!
- You will notice that, in the stencil code, we provide a `reader` and a `writer` - instead of using `fmt` to handle input and output, please use `io.WriteString(writer, string)` to write to output.
- You'll notice that the `action` function takes in a `*REPLConfig` - this is used so that any actions that are run know which user is making the request and where it should write output to. When needed, pass it the defined `replConfig`.
- You'll notice that the `Run` function takes in a `net.Conn` and a `uuid.UUID` - if you need to, pass in `nil` and `uuid.New()` respectively for those two fields for now.

2.4 Part 3: Executable

Relevant files: - `pkg/list/list.go`

Now, you'll put your REPL to good use and define a way to interact with a linked list! Your ListREPL should contain a List instance within it, which the user will provide commands to interact with. Define the following function in `list.go`:

```
// Returns a ListREPL.
func ListRepl(list *List) REPL
```

That supports the following commands:

- `list_print`: Prints out all of the elements in the list in order, separated by commas (e.g. "0, 1, 2")
- `list_push_head <elt>`: Inserts the given element to the List as a string.
- `list_push_tail <elt>`: Inserts the given element to the end of the List as a string.
- `list_remove <elt>`: Removes the given element from the list.
- `list_contains <elt>`: Prints "found!" if the element is in the list, prints "not found" otherwise.

Each of these commands should do something very similar: - Check that the number of fields is correct and throw an error otherwise (you may find `strings.Fields` useful) - Run the corresponding function on the included list. - Return any errors that may have been returned; otherwise, return `nil`.

To get a sense of how your `ListRepl` is being called, take a look at the `main.go` file we've included.

2.5 Error Handling

One pattern that you will see very often in Go is the error handling pattern. We want to emphasize it since forgetting to handle errors is often a common site for bugs. Whenever we have a function that returns an error, we always check it immediately:

```
value, err := f()
if err != nil {
    // Do something else with the error
    return err
}
```

2.6 Building

Once you've implemented the above functionality, you should be able to build and run. Running `make build` then `./bumble` should run the binary. If you are having any troubles with building or running, please let us know on Campuswire. We recommend building often to verify that your code is syntactically correct; the compiler should guide you as you learn the language.

2.7 Testing

When you submit, Gradescope will automatically test your code and report your grade back. This is the grade you will receive on the assignment, no strings attached. We don't believe in hiding test cases - if you're able to pass all of our tests, you deserve to know, and you deserve to be finished.

However, we highly suggest you test your own code anyways! Your stencil should contain a sample test that should show you how to write tests in Go. Testing is not graded, but is useful in verifying the completeness of your code.

Lastly, do not rely on the autograder to verify that everything in your codebase works. Certain things we don't have the infrastructure to test; for example, whether the REPL prints out reasonable help strings or terminates as expected all the time. Please actually build and run your code rather than just trying to pass our autograder tests.

2.8 Getting started

First, make sure you have a local development environment set up! See the local development guide for help, and let us know via Campuswire or TA Hours if you need help getting your dev environment set up.

To get started, get your stencil repository here. When cloning, **do NOT clone using ssh**! You must clone using HTTP for the build to work. A good first step is to ensure that your build pipeline is working by writing a quick "Hello, world!" program.

Please note: **you may NOT change any of the function headers defined in the stencil**. Doing so will break the autograder; if you don't understand a function header, please ask us what it means and we'll be happy to clarify.

---

3 FAQ

- **How is this graded?**: See the course guide.
- **How do I hand this in?**: See the course guide.
- **What packages can I import?**: You may import any standard library package that doesn't trivialize the assignment. You may not import any packages outside of the standard library (*e.g.* from GitHub). If you are ever unsure whether or not a package is fair game, please post on Campuswire.
- **Why is everything panicking?**: Remember to remove all of the panics once you've implemented a function!

Feedback

As this is a new course, we appreciate any feedback you have for us! If you enjoyed this assignment, hated this assignment, or have other thoughts to share, please do so here!