GIST A2

BY ANDREW SIKOWITZ

OVERVIEW FOR A2

- Create a text adventure game engine!
  - Read specifically formatted JSON files that specify a static adventure (right now, just a map)
  - Allow players to jump into an adventure, like a person in a room in the map
    - Dynamic game state (potentially) changes with each command
    - Players interact via a REPL, inputting text commands (right now, just `go` and `quit`)
- You are now in teams!
  - This team will last for at least 4 assignments. It can last the whole semester if you want.
  - Set up communication channel and **private** git repo
- You will extend your project in A3. Build your program with this in mind!

A2 DELIVERABLES

- Team Expectations Agreement
- Coping with Hitchhikers Response
- Zip file, created by `make zip`, which requires work in:
  - [adventure.ml]: A representation of an adventure, which is static and specified by a JSON file.
  - [command.ml]: Module for parsing player commands.
  - [state.ml]: A representation of a game state, which is dynamic, changing as the user plays the game.
  - [main.ml]: Entry point for executing the game.
  - [test.ml]: Test suite for Adventure, Command, and State tests.
  - [authors.ml] and [authors.mli]: Assignment metadata.

TEAM ROLES

- Things you should make sure to do for this assignment:
  - Work as a team – 3-4 people are expected to accomplish more than a single person
  - Implement the assignment as specified – this should be obvious
  - Upload a correctly formatted submission – don't lose easy points for dumb mistakes
- Accordingly, the Policies for Teamwork page specifies the following roles:
  - Coordinator
  - Monitor
  - Checker
- https://www.cs.cornell.edu/courses/cs3110/2018fa/teams/policies.html for details

Before getting started…

- More Makefile additions!
  - `make play`: Play the game by executing [main.byte]
  - `make zip`: Generate the zip file for your CMS submission
  - `make docs-public`: Generate doc files for .mli files, in directory [doc.public]
  - `make docs-private`: Generate doc files for .ml **and** .mli files, in directory [doc.private]
  - `make docs`: Run the two above
- [.ocamlinit] hidden file
  - Specifies commands to run before you enter utop
  - A2 provided file #require-s external packages and #load-s your compiled bytecode

SET-LIKE LISTS

- The specification for several functions mentions **set-like** lists
- Set-like lists are values of type ‘a list, but with set properties
  - There can be no duplicates in a set-like list
  - Set-like lists with the same set of elements are equivalent – order does not matter
- Whenever a function states that it returns a set-like list, make sure you return one
  - Look at the documentation in [adventure.mli] **and** [state.mli]
  - Think about the best way to ensure a list is a set-like list

RAISING EXCEPTIONS

- Some functions are specified to raise specific exceptions in specific circumstances
  - You must do this: consider it as one of the function's postconditions
- These are specified in the documentation for functions in the .mli files
  - Check your .mli files! They are your best friends
- Raise an exception as so:
  - If the exception takes no argument: `raise ExceptionName`
  - If the exception takes an argument: `raise (ExceptionName arg)`
    - Those parentheses are necessary!

CATCHING EXCEPTIONS

- try … with

```
try e0 with
| Exception1 -> e1          (* assumes Exception1 takes no argument *)
| Exception2 arg -> e2      (* assumes Exception2 takes an argument *)
```

- Evaluates to e0 if no exception is raised, e1 if Exception1 is raised, and so on
  - e0, e1, e2 **must** be the same type!

```
let x = (try e0 with _ -> e1)  (* get value from try block *)
```

CATCHING EXCEPTIONS

- match … with

```
match e0 with
| p1 -> e1
| p2 -> e2
| exception Exception1 -> e3
| exception Exception2 arg -> e4
```

- Evaluates as matching normally does, unless an exception is thrown when evaluating e0
  - If the exception matches one of the `exception ExceptionName -> e` cases, evaluates to e

ADVENTURE MODULE

- Come up with a type [t] to represent an adventure
  - You should know how you plan to implement [from_json], which converts JSON to your type [t]
  - Given a value of type [t], you should be able to implement the other functions in [adventure.mli]
- Implement [from_json], which involves parsing a [Yojson.Basic.json] value
  - Go through the JSON tutorial
  - Yojson.Basic.Util: member, to_string, to_list, to_assoc
  - List: map, assoc, mem_assoc, sort_uniq
- Implement the functions that take an adventure of type [t]
  - Make sure to raise the correct exceptions in the specified scenarios!

[schema.json]

- This is **not** an adventure file!
  - Instead, it is a specially formatted JSON file that specifies how **other** JSON files should look
  - These other JSON files are the adventure files (like [lonely_room.json] and [ho_plaza.json])
- The type of each JSON value is specified by the "type" field. Additionally…
  - Each "object" has:
    - its (key : value) pairs specified by the "properties" field
    - its required keys specified by the "required" field
  - Each "array" has its values specified by the "items" field

COMMAND MODULE

- Parse player commands of the form <verb> <object>
  - <object> can have multiple words separated by spaces
  - Multiple spaces should be treated as a single space
  - See [command.mli] for details
- Things to keep in mind:
  - String.split_on_char
  - Deep pattern matching – try to avoid nested matching!
- Really this time, check the **exact** circumstances in which you are to raise exceptions

STATE MODULE

- Come up with a type [t] to represent game state
  - Given a value of type [t], you should be able to implement [current_room_id] and [visited]
  - Does **not** need all the information in [Adventure.t] – [go] takes an [Adventure.t] as argument
- Implement [go] which facilitates progression of the game state
- Make sure to test

MAIN MODULE

- Implement a REPL (Read Eval Print Loop) that allows a user to play the game
  - Players must first enter the adventure file they would like to play
  - Players then enter the specified commands to play the game
  - Pay close attention to the writeup on what you are supposed to print
  - [read_line] to get a line of input from the user
  - [String.concat] may also come in handy…
- All I/O (e.g. reading input, printing) should be in this module, **not [State]**
- Note that [main.mli] is empty
  - All that matters is executing [main.byte] runs the game, done by: `let () = main ()`

ACCESSING MODULES

- `open M`
  - Should be at the top of the file / module; puts `M` in scope for that file / module
- `let open M in ...`
  - `M` in scope for everything under "in"
- `M.(...)`
  - `M` in scope inside parentheses
- `M.something`
  - You've seen this before; no scoping

Explaining [.ocamlinit]

- #use “<filename.ml>”;;
  - As if you copied all the code in the filename into the utop REPL, typed ;; and then hit enter
  - Will **not** work if the code relies on a module that has not been loaded into utop
- #load “<filename.cmo>”;;
  - Load compiled bytecode from a file
  - #load_rec “<filename.cmo>”;; for recursive dependencies
- #directory “<path>”;;
  - Add the directory <path> to the list of directories in which to search for files
- #require “<package_name>”;;
  - Load an **external** package (e.g. oUnit, yojson) that you likely installed with opam (run `opam list`)

FINAL TIPS

- Look at .mli files! Every function is very well-documented
- Try to implement all the functionality, including the excellent scope
  - You will have to do it for A3, anyway
- Think about what sort of extensions you would like to add, now!
  - These changes will likely affect all parts of the assignment
    - Adventure: type [Adventure.t], JSON parsing, and new exposed (in .mli) functions
    - Command: More commands
    - State: type [State.t], conversion from [Adventure.t] to [State.t], and new ways to change game state
    - Main: More information to print
  - Don't implement anything that would violate the A2 spec or cause `make check` to fail though!