ASSIGMENT 2 OVERVIEW

KYRYLO CHERNYSHOV- KC875

1. Summary

1.1. **Challenges.** The hardest part, for me, was parsing the JSON file. Normally, JSON parsers convert JSON directly to an associative array, but Yojson only made me a tree, which I had to parse the information out of manually. The good thing about that, though, is that after writing the code to parse out the information, I got the schema validation for free - if anything that should have been in the JSON was not, I would get a list parsing error somewhere in my code. I decided to catch it and rethrow as a Bad_json exception. This was OK, since a bad JSON file violates the preconditions of init_state, which means behaviour is undefined, so it can do anything, including throwing an exception.

Note: my custom adventure is in the file becker.json.

1.2. **Design, Implementation and Testing.** Something that I found very useful was creating two helper functions called map_reduce and filter_map_reduce, both documented but fairly self-explanatory. These were a God-send, and I used them many times over in various places in my code. I think I only performed one or two list iterations manually, one of which was my REPL - the rest was a combination of filter, map and reduce. Reduce isn't provided by default in OCaml as far as I'm aware, so I implemented its functionality using List.fold_left. These functions, although not new to me in this course, were very useful in terms of implementing functionality, since a lot of this assignment was about iterating over data structures.

I made my own string_to_lowercase function, since I wasn't 100% sure if we could use the one in the String module. It is a long piece of code, but I wrote it in about two minutes by generating 26 lines of the pattern matching code using JavaScript, which has its own built-in toLowerCase(). It was kind of annoying though.

1.2.1. *The state and command types.* Making the state type was fairly straightforward. I don't think there was a lot of variation possible: to me it made sense to make a record and basically copy the format used in the JSON. I defined a few sub-types like item, description, location, et cetera, and I used all of them in my main state type, in pretty much the same way as the JSON schema does it. I added a few fields that weren't there, such as a field to track whether a room was visited or not, a field to track how many turns the user has taken, and a field to track the current room the user is in.

The command type was even more straightforward. There were eight distinct types of command, three of which (take, drop, go) required an argument. I therefore made the command type a variant (since it is "one-of" rather than "each-of"), with the constructors for take, drop and go being of string. This allowed me to easily pattern-match in both main.ml and state.ml when processing commands.

1.2.2. *Testing.* By far the vast majority of my testing was done in utop. Utop allowed me to have a very short development cycle, by continuously testing each function as I implemented it. I did not spend any significant amount of time debugging, like I usually do in my personal projects, likely because of this short cycle that allowed me to avoid writing bugs in the first place. I also have about 70-80 test cases in my OUnit suite. I think that, between the test suite and the utop testing, I have done enough testing to uncover any major implementation bugs. I feel that one thing this project helped me considerably with was developing a better attitude to testing. I was genuinely surprised at how little time I spent debugging in the traditional sense, because whenever I discovered a "bug", it would be in the code that I had just written, so I could just use utop instead of spending half an hour looking at stack traces and error messages.

2. Known problems

I didn't have that much time to work on code that validates the game file, so that would be a problem in my submission. My code will detect a file whose JSON doesn't fit the schema, is not legal JSON, or a file that does not exist, and will assign blame accordingly, but it's not so good at detecting more subtle errors in a legal JSON file, for example references to non-existent items or rooms.

Also, I'm not sure if I had enough test cases to satisfy the grading criteria. I am confident that my functions work, though, due to testing done in utop.

3. Comments and feedback

One thing that was rather annoying and counter-intuitive to me was the specification for do'. I feel that it would have made much more sense to have do' take a string rather than a command, call Command.parse on it, and then if the command was invalid, throw an exception. The way I had to do it was call Command.parse in main, and also detect the validity of the command in main. I think that this is a bad model, since the first way means that there is zero interaction between Command and main, and also that there is a lot less unnecessary interaction between main and State, in checking if a command was valid or not before passing it into do'. Because of that, I had to expose a few functions in state.mli that really didn't need to be exposed, since this check could have just been done in state.ml, in do'.

Another thing I found very useful that I feel the course staff should include in the writeup is the fact that you can supply an optional argument (printer = string_of_int) to assert_equal. This makes it print out the expected value and the actual value if the assertion fails, which is typically automatically done by most testing libraries. I was told this by a TA in office hours but it would have been really cool to have known this at the start!