Favor for the Ringmaster

About

Project 1 for CMPE230 - Spring 2024 Course.
A REPL program (language shell) that takes either questions or sentences in a particular syntax.

Based on the sentences, it alters the program state, where a program state is "subject with an inventory of items in a particular location".

The program also responds to questions regarding inventory and location of subjects.

Usage

`make` to compile the program.
`./ringmaster` to run the program.

`make run` to compile and run the program

`make clean` to delete the executable file

`make grade` to test the program with 4 test cases under `/test-cases`

Example Input - Outputs

```
>> Ali go to Ankara
OK
>> Ali where ?
Ankara
>> Ali buy 2 pencil and Veli buy 3 pencil and 3 book
OK
>> Ali total ?
2 pencil
>> Ali and Veli total pencil ?
5
>> Ali go to Istanbul if Veli has more than 2 book
OK
>> Ali where ?
Istanbul
>> Veli where ?
NOWHERE
>> who at Bursa ?
NOBODY
>> exit
```

```
>> Ali and Ali go to Ankara
INVALID
>> Ali and ali go to Ankara
OK
>> who at Ankara ?
Ali and ali
>> Ali and Veli buy 2 pencil and Ahmet sell 1 pencil to Ali
OK
>> Ahmet buy 2 pencil and Ahmet sell 1 pencil to Ali and Veli
INVALID
>> Ali sell 1 pencil to Ali
INVALID
>> Ali
INVALID
>> exit
```

Program Structure

```
├── src
│   ├── ringmaster.c
│   ├── question.c
│   ├── question.h
│   ├── sentence.c
│   ├── sentence.h
│   ├── utils.c
│   ├── utils.h
│   ├── struct.h
│   └── state.h
│   ...
│   ...
│   ...
└── Makefile
```

The main program is in `ringmaster.c`. The program has utility functions for tokenization, parsing and logging in `utils.c`. Program structs are defined in `struct.h`.

Program Design and Rationale

Tokenization

The program reads the input file line by line and tokenizes the input by splitting the line by spaces and newline characters. Therefore, each token is merely a single word, delineated by spaces or newline characters.

The parser checks tokens against expected formats to identify and categorize questions. The program uses an Ad-hoc approach to parse the question and extract the necessary information from the question.
For instance,

Parsing Questions

Upon tokenization, the program checks whether last token is a question mark, in which case the program treats the input as a question. The program uses Ad-hoc approach to parse the question and extract the necessary information from the question. This is done via checking and matching relevant tokens in the question to the expected format of the questions.

Each input is evaluated to identify the type of question posed:

- Total Item Questions: Involves checking for a numeric calculation related to items owned by subjects.
- Location Questions: Determines where a subject is located.
- Identity Questions: Identifies who is at a specific location.
- Inventory Questions: Queries about the total number of items a subject possesses.

The parsing functions, `question_total`, `question_where`, `question_who_at`, and `question_inventory`, then validate whether the tokens match these expected patterns and return a flag indicating the question type.

Executing Questions

Once a question type is identified, `execute_question` orchestrates the response by invoking the relevant actions based on the question's nature:

Each case involves specific checks and responses, such as existence verification (`does_exist`), subject creation (`create_subject`), and data retrieval (`get_subject_location`, `get_index_of_subject`, and `get_subjects_from_location`). The responses are formatted and displayed directly to the user within this function.

Parsing Sentences

The `parse_sentences` function is designed to parse a series of tokens into structured sentences, each representing a combination of actions or conditions. This function is designed to take a tokens that is known not to be a question and parse it into a structured sentence. The program posseses structs for sentences, a sentence is further divided into other structs: Action and Condition.

Memory for sentences is pre-allocated based on the assumption that the minimum tokens per sentence is four. This heuristic simplifies memory management but may allocate more memory than necessary for some inputs.

A separate memory block (noSentences) is allocated for returning in cases of invalid parsing scenarios. This allows the function to handle errors gracefully by returning a consistent type without causing memory leaks. Upon encountering an invalid token or token sequence, the function prints "INVALID" and returns the noSentences array, indicating a failure to parse the input correctly. This ensures that the caller can continue running the program without crashing.

Based on the tokens, the parser identifies subjects and then determines the type of action. Depending on the action type, the state of the action is updated, and the relevant locations or items are finalized. Additionally, if the subsequent token is "to" or "from," the parser constructs the "from" or "to" attributes of the action, each of which includes one subject.

Once an action is completed, the parser checks for an "if" following the action. Absence of "if" implies that the sentence is not complete, and the parser begins to construct condition structures based on the number of conditions specified.

The construction of a condition structure follows a similar process to that of an action.

Perhaps the most subtle and critical part of this function is to decide whether there is an action after the last condition struct. The program takes the subjects as condition subjects because program doesn't know the state of the sentence. If there is a disrepancy in condition verb, program ends the sentence and starts a new sentence.

Parsing State

The parsing state consists of but not limited to the variables `SentenceState`, `ActionState`, and `ConditionState`. Based on a state and a given next token, the program decides the next state to transition to. Sometimes though, the next state is indecisive just from the current state and the next token. In such cases, the program uses look-ahead. This is because simplicity and readability. Defining a formal grammar and creating explicit states for each possibility would have been an overkill for this project. Therefore the states are defined in a purely semantic fashion.

State enums are defined in `/src/state.h`.

The start and end states are used to process common procedures. This helps to keep the code clean and readable. For instance, the program decides whether to parse another action or switch to another sentence when `ActionState = E_END`.

Although the program has start and end states, some of the procedures are done within other states for the sake of not introducing more complexity to the index and state transitions.

Executing Sentences

The functionality to execute parsed sentences is critical in altering the program state based on user commands or querying it for information. This process is managed through a series of functions that execute conditions, actions, and handle each sentence sequentially.

Overview of Functions

- `execute_sentences()`:
  - This function orchestrates the overall execution of sentences parsed from the input tokens. It processes each sentence sequentially if the sentence is valid.
- `execute_sentence()`:
  - Executes individual sentences by first checking if all conditions within the sentence are met. If they are, it proceeds to execute the actions defined in the sentence.
- `execute_actions()`:
  - Executes all actions within a sentence as long as they are valid and the required conditions are met.
- `check_conditions()`:
  - Evaluates all conditions specified in a sentence to determine if the subsequent actions can be executed.
- `check_condition()`:
  - Evaluates the given 1 condition and returns its truth value according to program's current state.
- `execute_action()`:
  - Performs the specified action by modifying the state of subjects based on the action type (e.g., moving locations, buying or selling items).
  - Based on the action type (`GO_TO`, `BUY`, `SELL`, etc.), updates the subject's location or inventory. Actions that involve transactions (`BUY_FROM`, `SELL_TO`) check if the inventory alterations are feasible before committing them.

This subsystem of executing sentences ensures that the program responds dynamically to user inputs, updating the state accurately and providing feedback when necessary. By structuring the execution process into distinct but interconnected functions, the program maintains clarity and modularity, facilitating easier maintenance and scalability.

Logging and Debugging

- The expected output of a sentence is either "OK" or "INVALID", hence the program does not wrap these with functions, they are repeated in various places.

- The answers to questions are printed within the function `execute_question`. And there is one helper function called `print_inventory`.

- The program has DEBUG flags in some of the source files to make use of verbose debug logs via functions such as `log_current_state`, `print_sentence`, `print_action`, `print_condition`. This helps us have an easier debugging process.

Some Challenges

- Although parsing questions was really straightforward, deciding the right level of formalism when parsing sentences was a challenge. We wanted to keep the code simple and readable. Therefore we decided to use some an ad-hoc approach to parsing sentences. However, it comes with a cost. The code is more prone to errors and harder to debug. Questions were in a very strict format, so we could use token indexes to extract the necessary information. But sentences were more flexible and required more complex parsing. We first thought we needed a rigorous LR1 parser to parse the sentences. Upon doing some research and experiments with LR1 approach, it was made clear that it had shortcomings, for example not being able make multiple steps of look-ahead.

- Test cases were also a challenge. We did not have abundant test cases. We manually developed test cases, which was time-consuming. We might as well have noted the test cases we have found and check whether they break as we alter the program so that we would not have to test each edge case in each iteration of the program.

- The differences on the machine and the local environment were also a challenge. Tokenization was done differently (incorrectly) on WSL. In order to overcome these issue we tested how Github Servers tokenize the input to make sure the program works expected. `.github/workflows/test-tokenizer.yml` was created for this reason.

- Thinking too much but not having our hands dirty and the resulting anxiety were possibly the biggest challenges along the way. We had less time to cover all cases than we thought we'd have. But in return; our structure has been very robust, easy to understand and maintain. Hence we were able to add new features and fix bugs easily.

- Use of 3 start and end states (both for sentence, action, and condition) required a thorough consideration of the interplay between them and very careful index handling.

Contributors

- Eray YÜKLÜ - 2021400273
- Ümit Can Evleksiz - 2020400114