Documentation

1- Introduction

This program is designed to track the inventories and current locations of given subjects within a given input by parsing it, which is a complex sentence that contains subjects, objects, actions and conditions in order to answer question related to subjects, locations and inventories.

2- Concept and Usage

The program accepts 3 types of inputs:

A- Sentence

A sentence is built with **3 types of constructs**; namely *Entities*, *Action* and optionally *Conditions*.

Sentences can be constructed in 3 different ways hence there are **3 types of sentences**; namely *Basic Sentence*, *Conditional Sentence* and *Sequential Sentence*.

Sentence constructs:

1. **Entities:**
    i. **Subject(s):** Subjects describe the entity that takes the action, a valid subject contains only uppercase or lowercase letters and underscores.
    ii. **Items:** Specifies the objects on which the actions are acted upon. Each valid object should contain a non-negative number specifying a quantity and an object name containing only uppercase and lowercase letters and underscores.
    iii. **Location:** Represents a location in which subjects can be found.
2. **Actions:**
    i. **buy:** Adds certain number of items to the subject's inventory. If there is a "from" flag, objects are removed from "from" subject's inventory and added to the buyer's inventory. Items and subjects are separated with and's if there are more than one. If there are not enough items in the from subjects inventory, task is not executed but OK is printed.

    Format:

    ```
    <SUBJECTs> buy <ITEMs> (OPTIONAL: <FROM> <SUBJECT>)
    where <ITEM> = <QUANTITY> <ITEM_NAME>
    ```

    Example Usage:

    ```
    >> Ali and Veli buy 3 bread and 5 apple
    >> OK
    >> Ali buy 5 banana from Veli
    >> OK
    ```

    ii. **sell:** Removes certain number of items from the subject's inventory. If there is a "to" flag, objects are added to the "to" subject's inventory and removed form the buyer's inventory. Items and subjects are separated with and's if there are more than one. If there are not enough items in the subjects inventory, task is not executed but OK is printed.

    Format:

    ```
    <SUBJECTs> sell <ITEMs> (OPTIONAL: <TO> <SUBJECT>)
    where <ITEM> = <QUANTITY> <ITEM_NAME>
    ```

Example usage:

```
Example usage:
>> Ali and Veli sell 3 bread and 5 apple
>> OK
>> Ali sell 5 banana to Veli
>> OK
```

iii. **go to:** Changes the location of the subject to the given location. Subjects are separated with and's if there are more than one.

Format:

```
<SUBJECTs> go to <LOCATION>
```

Example usage:

```
>> Ali go to Istanbul
>> OK
>> Ali and Veli go to Istanbul
>> OK
```

3- **Conditions:**

i. **at:** True if the subjects is at the given location. Subjects are separated with and's if there are more than one.

Format:

```
<SUBJECTs> at <LOCATION>
```

Example usage:

```
>> Ali and Veli at Istanbul
```

ii. **has:** True if the subjects has exact number of given items. Subjects and items are separated with and's if there are more than one.

Format:

```
<SUBJECTs> has <ITEMS>
```

Example usage:

```
>> Ali and Veli has 3 bread and 2 banana
```

iii. **has more than:** True if the subjects has more than certain number of given items. Subjects and items are separated with and's if there are more than one.

Format:

```
<SUBJECTs> has more than <ITEMS>
```

Example usage:

```
>> Ali and Veli has more than 3 bread and 2 banana
```

iv. **has less than:** True if the subjects has less than certain number of given items. Subjects and items are separated with and's if there are more than one.

   Format:

   ```
   <SUBJECTs> has less than <ITEMS>
   ```

   Example usage:

   ```
   >> Ali and Veli has less than 3 bread and 2 banana
   ```

Note that conditions cannot exist independently.

Constructing a sentence

Sentences can be made in three ways.

i. **Basic sentences**

   Contains a single action

   Example:

   ```
   Ali and Veli sell 3 apple to Mehmet.
   ```

ii. **Conditional sentences**

   In this type of sentences, action will be executed if the conditions are met.

   Example:

   ```
   Ali and Veli sell 3 apple to Mehmet if Ayse at Bursa
   ```

iii. **Sequential sentences**

   Combination of basic sentences and conditional sentences where there are groups of sentences in which the single or multiple basic sentences are connected to zero/single/multiple conditions

   Example:

   ```
   Frodo and Sam go to Bree and Frodo buy 3 map from Aragorn and
   Sam sell 2 dagger to Legolas if Frodo has more than 2 ring and
   Legolas and Gimli at Bree and Frodo and Sam go to Rivendell if
   Aragorn has 5 map and Frodo has less than 5 potion and Sam has
   3 dagger and Frodo sell 1 potion to Arwen and Legolas and Gimli
   go to Rivendell
   ```

B- Question

There are 4 types of questions available.

i. Subjects total item question: Prints the total number of the given item for all subjects (There may be only one subject or more than one subjects that are separated by and's).

Format:

```
<SUBJECTs> total <ITEM> ?
```

Example usage:

```
>> Ali and Veli total bread ?
>> 5
```

```
>> Ali total bread ?
>> 3
```

ii. Where Question: Prints the location of the given subject. If the subject's location is not initialized, prints "NOWHERE".

Format:

```
<SUBJECT> where ?
```

Example usage:

```
>> Ali where ?
>> Bursa
```

iii. Who at Question: Prints all the subjects located at the given location. If there is no subject in that location, prints "NOBODY".

Format:

```
who at <LOCATION> ?
```

Example usage:

```
>> who at Istanbul ?
>> Ali
```

iv. Total Inventory Question: Prints out all items and their quantities in the given subject's inventory. If there are no items with the quantity greater than 0, it prints out "NOTHING".

Format:

```
<SUBJECT> total ?
```

Example usage:

```
>> Ali total ?
>> 3 bread and 2 banana
```

C- Exit

Exits the program

Example Usage:

```
>> exit
```

3- Design and Implementation

The program is designed to have 5 main parts, which are parser, executor, inventory handler, question handler and ringmaster. The parser function takes the user input as plain text and carefully parses it. Then, it creates a package in a certain format and sends it to the executor. The executor function takes this input and using the inventory handling functions, it executes the task. Questions are separately parsed and executed by the question handler.

A. Parser

```
struct sentences_conditions_pair **parser(char *statement, char ***res,
char ****statement_parts, char ****final_sentences);
```

takes user input and returns a reference to sentences_conditions_pair struct array which is defined as follows:

```
struct sentences_conditions_pair
{
  struct sentence **sentences;
  struct condition **conditions;
  int is_valid;
};
```

the return struct contains a reference to an array of sentence struct and a reference to an array of condition struct.

Sentence struct represents a *basic sentence* (a sentence that contains a single action) and condition struct represents the condition of a *conditional sentence* (basic sentence + condition). By having an array of sentences_conditions_pair we can represent *sequential sentences* which is constructed of groups of single/multiple basic sentences connected to single/multiple or zero conditions.

The algorithm for parsing a complex sentence starts with finding so called *"big ands"* in a sentence. These big ands are the and words which connect basic or conditional sentences to build a sequential sentences. To find the indices of big ands in a given input the function *statement_parser* is called which is defined as follows:

```
int *statement_parser(char **all_words, int word_length, int *no_of_big_ands)
```

Explanation of algorithm: Basically algorithm looks for an action, and as soon as we encounter an action, it is obvious that the next and is either connecting 2 objects or connecting a new sentence to the current one if there is no following condition. We can distinguish if the and connects objects or sentences in absence of condition by observing valid objects must contain numbers for specifying quantities. In case there is a conditional statement following the basic sentence, we again look for if actions (which are the actions we have in conditions) and expect to find a big and afterwards. We can again distinguish the ands that connects items buy their numbers. Also asserting that in a valid sentence the first action we would observe after a big and lets us distinguish all the big and from others that connect subjects, items or conditions.

After obtaining the index array of big ands for a given input, we may start allocating memory for our words in a more structured way.

```
*statement_parts = (char ***)malloc((no_of_big_ands + 2) * sizeof(char **));
```

Then we fulfill the temporary storage *statement_parts* where for each sentence exactly the number of words it contains is allocated for using memory efficiently by the information we gain from big and indices as so:

```
for (int i = 0; i < no_of_big_ands; i++)
  {
    int end_index = big_and_indices[i];
    (*statement_parts)[i] = (char **)malloc((end_index - start_index + 1)
        * sizeof(char *));
    (*statement_parts)[i][end_index - start_index] = NULL;
    for (int j = 0; j < end_index - start_index; j++)
    {
      (*statement_parts)[i][j] = (*res)[start_index + j];
    }
    start_index = end_index + 1;
  }
```

Note that in our languages sentences A and B and C if D is interpreted as D implies both A, B and C; missing D implies A would be erroneous. In our current structure of word storage conditions are not structurally connected with the basic sentences they cover. Hence we will first find the basic sentences which contain conditions and than connect this sentence with all the basic sentences that the conditional statement also cover. For this more structured storage we allocate *final_sentences* as follows:

```
*final_sentences = (char ***)malloc((if_counter +
sentences_after_last_if + 1) * sizeof(char **));
```

where *if_counter* is the number of conditional statements where between each conditional statement there exists a sequential sentence or conditional sentence whose condition is the latter statement with if and *sentences_after_last_if* represents all the basic sentences after the last conditional statement (which indeed create a sequential sentence which has zero conditions). After fulfilling the final_sentences by the information we have from big_and_indices and if_indices we can start to parse our sequential sentences which is stored in final_sentences.

We iterate over each sequential sentence:

```
char **current_sentence = (*final_sentences)[i];
```

and pass this sequential sentence to the function *sentence_parse* which returns a reference to a single sentences_conditions_pair which is basically a sequential sentence representation (multiple or single basic sentence with zero/single/multiple conditions).

```
struct sentences_conditions_pair *pair =
sentence_parser(current_sentence, current_sentence_length);
```

For a given sequential sentence, sentence_parser splits it to conditional statement(s) and basic sentences. Each basic sentence is parsed within the function *little_sentence_parser,* which yields a reference to a sentence struct:

```
struct sentence *little_sentence =
little_sentence_parser(sentence, start_idx, end_idx);
```

For each condition, *if_sentence_parser* is called for parsing conditions, returning a reference to a condition struct

```
struct condition *condition =
if_sentence_parser(sentence, condition_start_idx, condition_end_idx);
```

By parsing all the basic sentences and conditions for all groups of a sequential sentence, we are able to successfully detect every action, subject and items for each condition or basic sentence.

The burden of validating sentences are distributed among parsers. Each parser checks for specific rule about the constructs and organization of sentence which they take as arguments. If a parser detects a syntax error, it immediately returns null. If a parser gets a null result from an inner parser it also understands that there is a syntax error and it also returns null immediately as well. For example:

```
>> Ali adn Ve4li buy 3 apple and 4 bread if Hakan at Denizli
```

This sentence is invalid since the subject name "Ve4li" contains a number. And parser detects it and returns null. All parsers in the hierarchy also returns value null to signal that parser found a syntax error as follows.

```
parser --> sentence_parser --> little_sentence_parser --> and_parser (detects error)
```

```
and_parser (returns null) -> little_sentence_parser (returns null) -->
sentence_parser (returns null) --> parser (returns null)
```

After parser returns the array of sentences_conditions_pair struct, we cannot immediately free our structs as other functions of our program will use them. To overcome this issue, we have declared our variables that will point to the dynamically allocated arrays out of parser, and passed their references to parser functions, that is the reason we used final_sentences and statement_parts with dereferencing them all along the parser function. Only after all other functions are done with current input we shall cal the free_memory function with our structs' references.

```
void free_memory(struct sentences_conditions_pair **pairs, char ***res,
char ****statement_parts, char ****final_sentences)
```

Another challenge we have encountered was that some dynamically allocated memory was pointed by multiple structs and our initial input structure. To overcome this issue we have used *can_free_objects* flag as an inner field of structs such that if it equals to 0 than trying to freeing the objects in this struct will definitely cause double free error. For example in our free_memory function we have first checked this field before freeing in order to prevent errors.

```
if (sentence->can_free_objects == 1)
      {
        while (objects != NULL && objects[k] != NULL)
        {
          free(objects[k]);
          objects[k] = NULL;
          k++;
        }
      }
```

B. Executor

The executor function consists of two main parts, checking the conditions and executing the sentences. For each sentence-condition pair, if all of the conditions are satisfied, all sentences are executed.

```
for each SENTENCE_CONDITION_PAIR:
    if CONDITIONS_SATISFIED:
        EXECUTE_SENTENCES
```

i. Checking the conditions

In order to check a list of conditions, all of the them are traversed using a for loop. For each condition, singleConditionChecker function is called. This function checks the truth value of a single condition using functions provided by the inventory handler.

```
for each CONDITON in CONDITIONS:
    if singleConditionChecker( CONDITION ) is False:
        return False
return True
```

ii. Executing the sentences

In order to execute a list of sentences, all of them are traversed using a for loop, and executed one by one by the singleSentenceExecutor function. This function executes a single sentence using functions provided by the inventory handler. If it is not possible to execute the sentence, the function does nothing. Thus, only possible sentences in the for loop are executed.

```
for each SENTENCE in SENTENCES:
    singleSentenceExecutor( SENTENCE )
```

C. Inventory Handler

This part of the program is responsible for keeping the track of the subjects, their inventories and their current locations. This part can be separated to two parts:

i. Storage of subjects and their items

Each subject is stored as a Person struct. In order to store all subjects effectively in the memory, they are stored in a linked list instead of a preallocated array. Thus, size of the list increases as new subjects are introduced. The linked list structure is achieved with the help of pointers. Each item and person has a data field to point to the next one.

```
// Person struct definition
struct Person
{
  char name[MAX_NAME_LENGTH];
  char location[MAX_LOCATION_LENGTH];
  // First item in the inventory
  struct Item dummyItem;
  // Pointer to the next person
  struct Person *nextPersonPtr;
};
```

Similarly, to be memory efficient, items of each subject are stored in a linked list instead of a preallocated array.

```
struct Item
{
  char itemName[MAX_ITEM_NAME_LENGTH];
  int quantity;
  // Pointer to the next item
  struct Item *nextItemPtr;
};
```

First elements of both linked lists are set to be dummy and last item in the list points to NULL. The structure can be represented in this way:

```
DummySubject --> Subject1 --> Subject2 --> ... --> SubjectN --> NULL

DummyItem --> Item1 --> Item2 --> ... --> ItemN --> NULL
```

ii. Operations

This part of the inventory handler provides the functions for getting and setting the items and subjects stored in the linked lists. Also, some specific functions to check conditions directly such as hasExactItems() are provided. These functions simply reach the corresponding data field of the items using for loops and then return them or create new items or subjects.

```
void addItem(struct Person *person, char *itemName, int quantity)
void removeItem(struct Person *person, char *itemName, int quantity)
int getQuantity(struct Person *person, char *itemName)
int hasEnoughItem(struct Person *person, char *itemName, int quantity)
int hasExactItem(struct Person *person, char *itemName, int quantity)
int hasMoreThanItem(struct Person *person, char *itemName, int quantity)
int hasLessThanItem(struct Person *person, char *itemName, int quantity)
void changeLocation(struct Person *person, char *location)
char *getLocation(struct Person *person)
struct Person *getPersonByName(struct Person *firstPerson, char *name)
```

D. Question Handler

This part of the program handles question type inputs. If a text ending with a question mark is inputted, this part of the program is called. Since only certain formats of questions are valid, question_handler searches for certain keywords to decide the type of the question. After deciding the type of the question, the input text is parsed. While parsing the input text, if it is not in the correct format, -1 is returned to the ringmaster. If the format is correct, then using functions provided by the inventory handler, the output is printed out.

```
if QUESTION_TYPE is Q1:
    if CORRECT_FORMAT:
        PRINT(RESPONSE)
    else:
     return -1
else if QUESTION_TYPE is Q2:
    if CORRECT_FORMAT:
        PRINT(RESPONSE)
    else:
     return -1
.
.
.
else :
    return -1
```

E. ringmaster

This is the main part of the program. It takes input until the user inputs “exit”. In each while loop, if the input text is not a question, it is sent to parser. If parser doesn’t return NULL, which means invalid, output of the parser is sent to executor. If the parser returns NULL, “INVALID” is printed out. If the input is a question, it is directly sent to question_handler. Similarly, if the question_handler returns -1 as an exit code, “INVALID” is printed out.

```
WHILE(INPUT != "exit"):
    TAKE INPUT
    if INPUT is QUESTION:
        if QUESTION_HANDLER(INPUT) == -1:
            PRINT("INVALID")
    else:
        if PARSER(INPUT) != NULL:
            EXECUTOR(PARSER(INPUT))
        else:
            PRINT("INVALID")
```