

CMPE230, Assignment 1

This is the documentation paper for the CMPE230 class, 1st assignment. The purpose of the program which we have implemented is essentially to provide an interface for several operations. We have implemented an interpreter for Tolkien to enable him to keep track of his characters, items, and locations in C Language. We make interpretations from a language somewhat similar to the natural language English. We store information and states about entities and perform some actions when the user prompts so.

First of all, our program asks for user input and runs until the user prompts “exit”. We have a total of 5 actions which are “Buy”, “Buy ... from”, “Sell”, “Sell ... to”, and “go to”. We perform these actions on items or locations after checking for the validity of the user prompt. We inform the user if the prompt is not valid. One feature of our interpreter is it allows action and entity chaining by using the “and” keyword. Users can use it many times as he/she likes to give several commands, subjects, or items at once. However, one should be careful while using it since there are some constraints such as the “Sell ... to” command allows for only a single buyer, etc.

As it comes to our implementation, we first read a line from the user input and tokenize it, meaning that split it into its keywords. After that, we check whether the user input is a question or not. We answer when the user asks valid questions by querying. We prompt the user “INVALID” if the question is not in a format we require. If it is a question, we give the results by simply going over the arrays which we initialize and update during the run time. Note that since it will work like a storage system, we don’t clear these arrays. Our design decision is like this:

We have structs for subjects, items, and locations. Our subject struct holds both items and location parameters while our location struct also holds subjects that are in that location. Note that for a subject to be at a location, the user must prompt “*Subject go to Location*”. Initially, we don’t have any subjects, locations, or items and as the user gives prompts, we tokenize the prompts and create/update the entities in their arrays.

Another challenge for us was correctly tokenizing the user input. Since the “and” keyword may be used in several places, we had to be careful not to cause any ambiguity. We first thought if we could find a way to identify which “**and**”s are used for separating the sentences and which ones are used to separate entities, our job would then be just performing the sentences one by one. However, since the language we required was a primitive one compared to natural English, it was a little harder than we first thought. Then we started to look for another solution and we came up with a more straightforward one.

Our final algorithm works as follows. We go from left to right along the line and perform the actions when we can. However, since there can be conditional sentences, we don’t perform the actions immediately but wait for the sentence to finish (bold **and**’s in the description mean the sentence has finished and a new sentence is on the way). While we are traversing the input line, we manually check the possible next keywords and entities and act accordingly. If we came across a word that we didn’t expect (Not counting the special entity names) such as an “and” after “and”, we can say that the user input for that line is INVALID. If the line is valid, we store the words in different arrays according to their types (Subject, Item, Location, Action and Condition). We update each action’s subject, item, location or condition window to determine which entities and conditions belong to which actions. A window is basically created by using two pointers and the entities between these two pointers construct a window. Since we know the relationship of each word and it’s sentence, we can then easily perform the actions.

Finally, note that we say OK if the input format is correct although the action can't be performed due to conditional restrictions. Also since we make dynamic memory allocation while reading a new line from the user input, we free the space after each line to make sure that we don't cause any memory leak.

To use the program from the command line, you must go to the root directory in which the Makefile file exists. Then by using "\$make" and "\$./ringmaster" commands, you can run the program. Once you are finished with using it, typing "exit" will get you back to the root directory. Example inputs/outputs of the program is given next.

```
berkaybgk@berkaybgk-2 Desktop % make
gcc -w -o ringmaster mainFile.c
berkaybgk@berkaybgk-2 Desktop % ./ringmaster
>> Frodo go to Rivendell
OK
>> who at Shire ?
NOBODY
>> Frodo      where ?
Rivendell
>> Frodo sell 2 bread to Sam
OK
>> Frodo and Sam buy 3 bread and 2 ring
OK
>> Sam sell 1 ring to Sauron
OK
>> Sauron where ?
NOWHERE
>> Sam total ?
3 bread and 1 ring
>> Frodo buy 2 palantir and Frodo go to NOWHERE
INVALID
>> Frodo total palantir ?
0
>> Frodo total Bread ?
0
>> Frodo go to mount_doom and Gandalf buy 3 arrow if Sauron has 1 ring and Frodo at Rivendell
OK
>> who at mount_doom ?
Frodo
>> Legolas buy 100 hairclip from Arwen if Galadriel at Lothlorien
OK
>> Legolas total hairclip ?
0
>> Legolas buy 100 hairclip from Arwen if Gandalf has more than 2 arrow
OK
>> Legolas total hairclip ?
0
>> exit
berkaybgk@berkaybgk-2 Desktop %
```

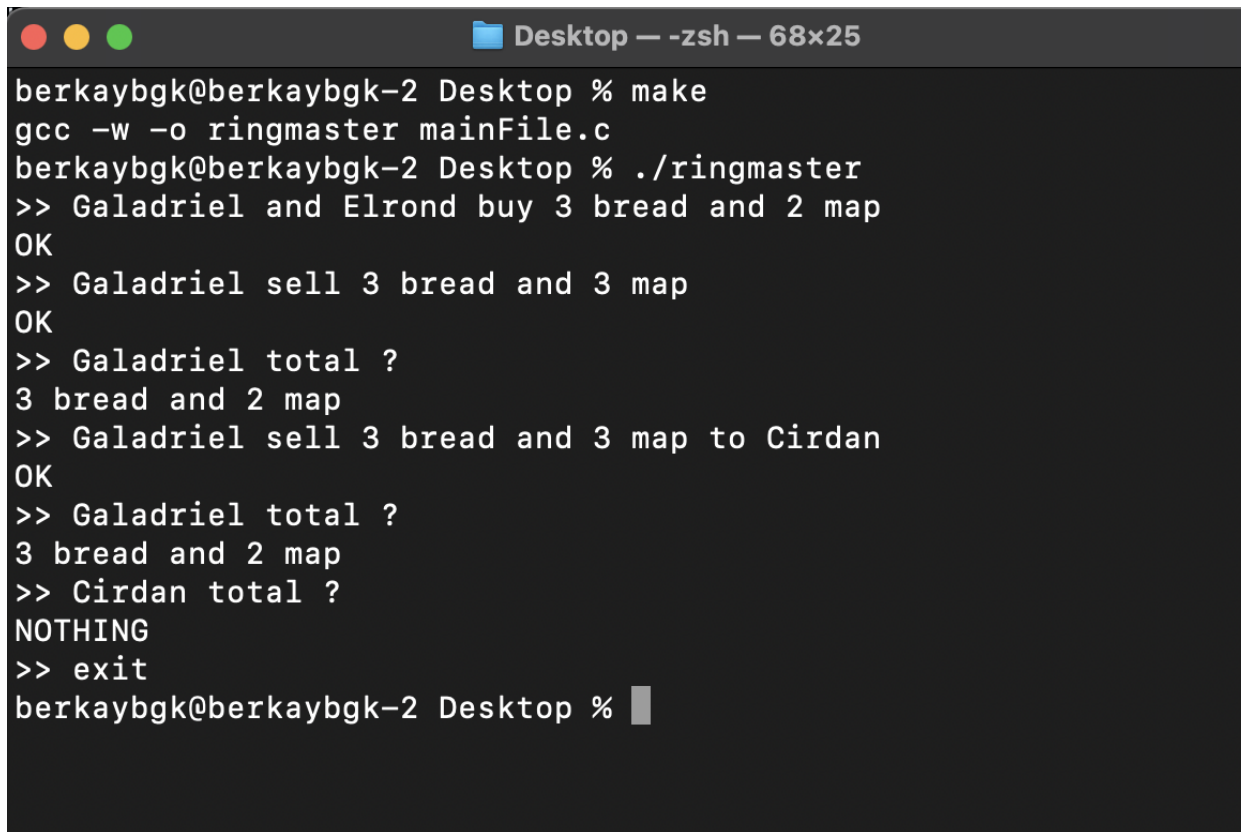
Another test case:

```
berkaybgk@berkaybgk-2 Desktop % make
gcc -w -o ringmaster mainFile.c
berkaybgk@berkaybgk-2 Desktop % ./ringmaster
>> Galadriel and Elrond and Cirdan buy 100 Narya and 100 Narya and 100 Vilya
OK
>> Balrog and Saruman buy 10 Vilya and 10 Narya from Cirdan
OK
>> Cirdan total ?
100 Narya and 80 Narya and 80 Vilya
>> Balrog total ?
10 Vilya and 10 Narya
>> Balrog and Cirdan sell 10 Narya to Legolas
OK
>> Legolas total Narya ?
0
>> Balrog and Cirdan sell 10 Vilya to Legolas
OK
>> Legolas total Vilya ?
20
>> Balrog and Cirdan total Narya ?
90
>> exit
berkaybgk@berkaybgk-2 Desktop %
```

Another test case:

```
berkaybgk@berkaybgk-2 Desktop % make
gcc -w -o ringmaster mainFile.c
berkaybgk@berkaybgk-2 Desktop % ./ringmaster
>> someone go to somewhere and somebody buy 1 somethings and 2 SoMeTHiNG if someone has less than 2
somethings and 1 nothinG and someone go to home_of_someone if someone has 10 something
OK
>> someone where ?
somewhere
>> somebody where ?
NOWHERE
>> who at home_of_someone ?
NOBODY
>> everyone and somebody and someone go to everywhere and this is invalid if everyone at everywhere
INVALID
>> everyone where ?
NOWHERE
>> exit
berkaybgk@berkaybgk-2 Desktop %
```

Another test case:

A terminal window titled "Desktop — -zsh — 68x25" with a dark background and light gray text. The window shows the execution of a program named "ringmaster". The user enters "make" to compile the program, then runs it with "./ringmaster". The program prompts for commands. The user enters "Galadriel and Elrond buy 3 bread and 2 map", which is confirmed with "OK". Then the user enters "Galadriel sell 3 bread and 3 map", also confirmed with "OK". The user then asks for the total for Galadriel, and the program outputs "3 bread and 2 map". The user then enters "Galadriel sell 3 bread and 3 map to Cirdan", confirmed with "OK". The user asks for the total for Galadriel again, and the program outputs "3 bread and 2 map". The user then asks for the total for Cirdan, and the program outputs "NOTHING". Finally, the user enters "exit", and the terminal returns to the shell prompt "berkaybgk@berkaybgk-2 Desktop %".

```
berkaybgk@berkaybgk-2 Desktop % make
gcc -w -o ringmaster mainFile.c
berkaybgk@berkaybgk-2 Desktop % ./ringmaster
>> Galadriel and Elrond buy 3 bread and 2 map
OK
>> Galadriel sell 3 bread and 3 map
OK
>> Galadriel total ?
3 bread and 2 map
>> Galadriel sell 3 bread and 3 map to Cirdan
OK
>> Galadriel total ?
3 bread and 2 map
>> Cirdan total ?
NOTHING
>> exit
berkaybgk@berkaybgk-2 Desktop %
```

This homework is done by Cengiz Bilal Sarı and Berkay Buğra Gök. (2021400201 and 2021400258)