Project Report

Systems Programming – Spring 2025

Barış Çakmak, 2022400000
Tuğçe Yücel, 2022400105

April 14, 2025

1 Introduction

This project implements an interpreter and an inventory tracking system for Geralt.The system allows Geralt to manage ingredients, potions, trophies, and information about monsters. Main goal is executing given input line by line and calling necessary functions after validating every line.

2 Problem Description

The task is to create an interpreter that interpretes the given input line by line and completes the following task correctly

- For each line, parse the input and detect any grammar mistakes that violate grammer rules
- Then, call corresponding methods for a given specific input
- Update the corresponding data structures if the method that is called requires to do so
- Lastly, for each line, give an output representing the respond to the given input line

3 Methodology

Our project includes various data structures such as hashmap, linked list and further specially designed data structures. Even if hashmap and linked list is not necessary considering the size of possible inputs, we thought It was better to use hashmaps for good coding practice. Hashmaps and linked lists were used in following areas.

- Storing ingredients, trophies, potions, monsters in a hashmap and each entry in the hasmap is a linked list to resolve collisions

To acquire a solid and dynamic structure, we used generic hashmaps that are created inside heap. With this improvement, our code will also work in very large inputs in seconds and can be reutilized for different data types since it is generic.

4 Implementation

To begin with, given the input line, we will tokenize each line by white spaces. Doing so, our job will be easier when handling inputs with so many white spaces. After each word is tokenized, execute_line() method is called for checking whether the input is grammerly correct or not.
After the correctness of the input is guaranteed, we can now safely send the information to the method we want to call.

4.1 Code Structure

4.1.1 Modules

- `main.c`: Handles input reading, command tokenization, grammer checking.
- `actions.c`: Implements core logic for loot, trade, brew, learn, and encounter actions.
- `queries.c`: Handles queries related to ingredients, potions, trophies, and bestiary.
- `hashmap.c`: Implements a hashmap using separate chaining with dynamic resizing.
- `helper_methods.c`: Includes helper functions for parsing, validation, and sorting.
- `structures.c`: Defines custom data structures like `Pair`, `Potion`, `Bestiary` and `PairArray`.

4.1.2 Functions

- `execute_line`: Parses and validates user input, then sends necessary data to corresponding function.
- `checkPairs`: Validates input syntax for ingredients and trophy pairs, ensuring grammar correctness.
- `constructPairArray`: Converts valid token sequences into arrays of `Pair` objects.
- `isNameValid`: Ensures a string contains only alphabetic characters and single spaces between words.
- `extractPotionName`: Extracts potion names by removing trailing characters from command strings.
- `createArrayOfKeys`: Extracts all keys from a hashmap into an array of strings.
- `loot`: Updates Geralt's ingredient inventory with newly looted items.
- `trade`: Trades trophies for ingredients only if all required trophies are available.
- `brew`: Creates a potion if Geralt knows the formula and has the necessary ingredients.
- `learnPotionRecipe`: Adds a new potion formula to the potion hashmap.

- `learnSign`: Records effective signs for known or new monsters in the bestiary
- `learnPotion`: Records effective potions for known or new monsters in the bestiary.
- `encounter`: Geralt fights with the monster and the result is determined based on known effectiveness.
- `specificIngredients`, `specificPotion`, `specificTrophies`: Look up to corresponding hashmap and retrieve the amount of the item.
- `allIngredients`, `allPotions`, `allTrophies`: Print sorted summaries of current inventory contents.
- `potionFormula`: Outputs the recipe of a known potion, sorted by amount, then by name.
- `potionSignEffectiveness`: Prints all known effective signs and potions against a given monster.

4.2 Sample Code

Below, a descriptive function is provided for every module of the project.

`actions.c`:

```c
void learnPotionRecipe(HashMap *potions, char *potion, PairArray *
    ingredients){

    if (contains(potions, potion)){ // If formula is already known
        printf("Already known formula\n");
        return;
    }

    Potion *potionWithRecipe = malloc(sizeof(Potion)); // Allocate
        memory for the potion
    potionWithRecipe->recipe = ingredients; // Set the recipe
    potionWithRecipe->potionCount = 0;

    insert(potions, potion, potionWithRecipe, sizeof(Potion)); //
        Insert the potion to the potions Hashmap
    printf("New alchemy formula obtained: %s\n" , potion);
}
```

`hashmap.c`:

```c
int contains(HashMap *map, const char* key){

    int index = hash(map, key); // Find the possible index of the key

    HashNode *currentNode = map->table[index]; // Get the head of the
        linked list

    // Traverse the linked list
    while (currentNode){
        if (strcmp(currentNode->key, key) == 0)
            return 1;
        currentNode = currentNode->next;
```

```
    }

    return 0;
}
```

queries.c:

```
void specificPotion(HashMap *potions, char *potion){
    if(contains(potions, potion)){  // Checks if potion is available
        Potion *p = (Potion *)get(potions, potion);
        printf("%d\n", p->potionCount);
        return;
    }
    else{  // If it is not in the potions list
        printf("0\n");
    }
}
```

helper_methods.c

```
void extractPotionName(char *potionStart, char *firstWord){
    char *temp = potionStart; // Potion start address
    char *firstWhiteSpace; // First white space after potion name,
       initially uninitialized

    // Loop until we reach first word after potion name
    while (temp != firstWord){
        if (*temp == ' '){ // If we reach a white space, save it and
           skip adjacent white spaces
            firstWhiteSpace = temp;
            temp++;
            while (*temp == ' '){
                temp++;
            }
        }else{
            temp++;
        }
    }

    // Here firstWhiteSpace is pointing to the first white space after
       potion name
    *firstWhiteSpace = '\0';
}
```

structures.c

```
void resizeArray(PairArray *pairArray){
    int newSize = pairArray->capacity*2;
    Pair **newArray = realloc(pairArray->array, newSize*sizeof(Pair*));

    if (newArray == NULL){
        printf("Memory reallocation failed!");
        exit(1);
    }
    pairArray->capacity = newSize;
    pairArray->array = newArray;
}
```

5 Results

5.1 Invalid

```
>> Geralt loot 3 Rebis, 2 Aether
INVALID
Explanation: "loot" instead of "loots"

>> Geralt learns baris  tugce potion consists of 3 Rebis, 1 Aether
INVALID
Explanation: More than one spaces in a potion name

>> Geralt brews baris  tugce potion
INVALID
Explanation: After potion name nothing should come and potion name is again wrong

>> Geralt loots -3 Rebis, 0 Coconuts
INVALID
Explanation: Non-positive integers are not allowed for quantity

>> what is in baris  tugce ?
INVALID
Explanation: w should be capital and potion name is wrong

>> Geralt brews a baris  tugce
INVALID
Explanation: After brews just the name of the potion must come
```

5.2 Valid

```
>> Geralt loots 3 Rebis   ,  2 Aether
Alchemy ingredients obtained.

>> Geralt learns  baris tugce    potion   consists of 3 Rebis, 1 Aether
New alchemy formula obtained: baris tugce

>> Geralt brews baris tugce
Alchemy item created: baris tugce

>> Total ingredient ?
2 Aether

>> What is in baris tugce ?
3 Rebis, 1 Aether

>> Geralt brews baris tugce
Not enough ingredients
```

6 Discussion

Our hashmap is very well designed and functions properly, but linked list chaining in hashmaps may sometimes be a problem when inputs are specifically given to hash to same location. In the worst case, if every element is hashed to the same index, the hashmap will not perform operations in O(1) but instead O(n), where n is the number of elements in the hashmap.

Other than that, since we assumed there will not be vast inputs, malloc and realloc checks were not made strictly. Hence, if a massive input is given to program and hashmap becomes so large, when it tries to rehash it may not succeed. If this project is going to be improved for a larger scale, this point should be considered.

We are all humans and we can always make mistakes. Therefore, even if a programmer thinks that there are no memory leaks in his program, he should consider adding a memory leak detection to the program to ensure validity and facilitate debugging.

This project is just made for the given pdf description and may not be appropriate for other usages of interpretation and grammer checks even if it seems like it would work. Adjustments adviced are listed below

- Design a more robust hashmap implementation with different collision resolution methods
- Ensure memory safety with a memory leak detector
- Consider other cases from the ones in the provided code as well to obtain a generalized version of the code

7 Conclusion

This project demonstrated modular C design, dynamic memory management, and interactive input handling. By accurately reflecting Geralt's adventures through commands, we created a robust and extensible command interpreter.

References

- ChatGPT — While designing the hashmap and which methods to use in hashmap, GPT helped us a lot at understanding how a double pointer works and how to free it without any errors. Besides this, we had minor errors in our code at some places and GPT assisted us to resolve those. Lastly, to check if our code works, We asked for GPT to generate us an input file and we took it as a grader for out project.
- Gökçe Uludoğan — Examined the GitHub repo provided in pdf and it helped us to create the structure of this documentation.

  Link: `https://github.com/bouncmpe230/project-report-template`