Project Report

Systems Programming – Spring 2025

Üstün YILMAZ, 2023400108
Ulaş Sertan KEMEÇ, 2022400063
Department of Computer Engineering
Boğaziçi University

April 14, 2025

Abstract

This document presents the report for the CMPE230 Project Assignment ”Witcher Tracker”. It briefly talks about a broad overview of the project, and describes the problems to handle, in detail. It moves on to explain the methodology behind the project and how it's implemented in the program. Finally, the performance aspects of the project are discussed.

1 Introduction

The project wants us to implement an interpreter that takes input from stdin and prints the correct results of the interpreted query to stdout. This is all done in C programming language, standard C99, in our implementation. The interpreter will interpret two different types of commands, where one corresponds to the Witcher, Geralt's actions and the other corresponds to the queries about the consequences of these actions for Geralt. There is also a reserved exit command to exit the program.

2 Problem Description

The first problem here is to make this interpreter abide strictly by the Backus-Naur form CFG given in the description and print out ”INVALID” for any given invalid inputs. After this has handled, a second and more mild problem arises, which is to process any valid inputs and print out their corresponding results properly. Also the use of data structures is needed in the background while handling these valid inputs, since we are required to record the consequences of Geralt's actions briefly mentioned in the introduction.

The given constraints for the I/O and grammar are given as:

- Inputs are read as lines from `stdin`, after the prompt `>>`, and can be at most 1024 characters long.
- Any invalid input must result in printing `INVALID` (without further processing).
- An input of the form `Exit` or `exit` should terminate the program.

- All keywords and entity names are matched *case-sensitively.*
- Valid *action* commands must be exactly one of:
  - `Geralt loots <ingredient_list>`
  - `Geralt trades <trophy_list> for <ingredient_list>`
  - `Geralt brews <potion>`
  - `Geralt learns <sign> sign is effective against <monster>`
  - `Geralt learns <potion> potion is effective against <monster>`
  - `Geralt learns <potion> potion consists of <ingredient_list>`
  - `Geralt encounters a <monster>`
- Valid *query* commands must be exactly one of:
  - `Total ingredient <name> <0_or_more_spaces>?`
  - `Total potion <name> <0_or_more_spaces>?`
  - `Total trophy <name> <0_or_more_spaces>?`
  - `Total ingredient <0_or_more_spaces>?`
  - `Total potion <0_or_more_spaces>?`
  - `Total trophy <0_or_more_spaces>?`
  - `What is effective against <monster> <0_or_more_spaces>?`
  - `What is in <potion> <0_or_more_spaces>?`
- Any input not matching one of the above patterns must be rejected as `INVALID`.
- Ingredients, monster names, trophies and signs must be a single word that are separated by one or more spaces in the respective commands they are utilized in, while potions must always have exactly one space between their words. No numeric characters are allowed in any of these.
- Quantities must be positive unsigned integers.

3 Methodology

The solution for the first problem, which is to make the interpreter abide by the BNF is done by implementing a `validity_checker()` function with proper tokenization done at `typedefs.h` and `bools.c` to check that tokenization. The `validity_checker()` function accepts a string called `command`, in both mutable and immutable forms, and a parsed version of it called `parsed_command`. Then the boolean functions in `bools.c` are utilized to check each token of that command. If any inconsistency is found, the program skips that input and prints `INVALID` to stdout. The algorithm that does this is an $O(n)$ complexity algorithm where n is the number of characters in any command. Everything is handled via non-nested for loops in which the ones in `bools.c` check the tokens' characters one-by-one and the ones in `validity_checker()` check the tokens one-by-one. The second problem is to process the inputs and give an output. It is handled by defining the corresponding structures of the tracker in `typedefs.h` and uses a linked-list and a dynamic array structure to add new stuff to the inventory. The same inventory is then used by the `parse_query()` function to handle any valid query commands about the inventory. All examples the codes for our algorithms are given at section 4.2, along with implementation details.

4 Implementation

The project is implemented using a variety of C programming fundamentals, like structs and dynamic memory allocation. Structs help us easily differentiate between the types of tokens and how to process them, and dynamic memory allocation ensures that the program efficiently handles every operation. More details about the structure of the code are mentioned below.

4.1 Code Structure

The program consists of 11 files, in which 1 is the `main.c` file, the driver code for the program, 1 is the `typedefs.h` file, which defines all structs and function prototypes, and 10 other `.c` files that contain the operations related to its name. (e.g. `potion.c` contains all operations regarding the potions) All these files are compiled together to produce the executable `witchertracker`, which can read inputs from `stdin` and print them to `stdout`.

4.2 Sample Code

The sample code is in the order of:

1. An example from `bools.c` to check a token
2. An example from `validity_checker()` to check a command
3. An example linked list and the dynamic array structure from `typedefs.h`
4. An example from `handle_loot()`
5. An example from `parse_query()`

```

int is_word_token_with_comma(const char *token) {
   if (*token == '\0') return 0;
   const char *current_character = token;
   while (*current_character) current_character++; // Go to the end of the token
   if (*(current_character - 1) != ',') return 0;
   current_character = token;
   while (*(current_character + 1)) {
      if ((*current_character < 'A' || *current_character > 'Z') && (
             *current_character < 'a' || *current_character > 'z'))
         return 0;
      current_character++;
   }
   return 1;
}
```

```c

... if (is_brew_sentence(parsed_command)) {
        // If there is more than one space between potion names, the input is invalid
        int word_count = 0;
        int space_count = 0;
        int is_in_word = 0;
        for (int i = 0; i < (int)strlen(const_command); i++) {
            if (const_command[i] == ' ') {
                if (space_count == 0) {
                    if (is_in_word){
                        word_count++;
                        is_in_word = 0;
                    }
                    space_count++;
                } else {
                    if (word_count > 2) {
                        printf("INVALID\n");
                        return 0;
                    }
                    space_count++;
                }
            } else {
                is_in_word = 1;
                space_count = 0;
            }
        }
        // Commence similar operations like the loot sentence case
        if (parsed_command->size <= 2) {
            printf("INVALID\n");
            return 0;
        }
        if (parsed_command->size == 3) {
            if (!is_word_token(parsed_command->line_array[2])) {
                printf("INVALID\n");
                return 0;
            }
        }
        for (int i = 2; i < (int) parsed_command->size - 1; i++) {
            char *token = parsed_command->line_array[i];
            char *token2 = parsed_command->line_array[i + 1];


            if (!(is_word_token(token) && is_word_token(token2))) {
                printf("INVALID\n");
                return 0;
            }
        }
        return 1;
    }

typedef struct bestiary{
    char* monster_name;
    dynamic_array* effective_signs;
    dynamic_array* effective_potions;
    int monster_count;
    struct bestiary* next_bestiary;
}bestiary;
```

```c
typedef struct dynamic_array {
    void** ptr_arr;
    int* int_arr;
    size_t size;
    size_t capacity;
}dynamic_array;

void handle_loot(inventory *inv, const parsed_line *line) {
    int index = 2;  // skip "Geralt loots"
    while (index < (int)line->size) {
        // Skip any commas
        if (strcmp(line->line_array[index], ",") == 0) {
            index++;
            continue;
        }

        // Parse the quantity
        int quantity = string_to_int(line->line_array[index]);
        if (quantity <= 0) {
            return;
        }
        index++;

        // Get the ingredient name
        if (index >= (int)line->size) {
            return;
        }
        char *ing_name = line->line_array[index++];
        ing_name = remove_coma(ing_name);  // The commas are removed in parse(), but this also handles any edge cases

        // Add the ingredient to inventory
        ingredient *found = ingredient_in_inventory(inv, ing_name);
        if (found) {
            found->quantity += quantity;
        } else {
            ingredient *new_ing = create_ingredient(ing_name, quantity);
            new_ing->next = inv->ingredient_inventory;
            inv->ingredient_inventory = new_ing;
        }

        // Skip any commas
        if (index < (int)line->size && strcmp(line->line_array[index], ",") == 0) {
            index++;
        }
    }

    printf("Alchemy␣ingredients␣obtained\n");
}

if (strcmp(parsed_command0->line_array[0], "Total") == 0 && (strcmp(parsed_command0->line_array[1], "potion") == 0 || strcmp(parsed_command0->line_array[1], "potion?") == 0)) {
        // Parse the line here
        int array_length = (int) (parsed_command0->size);
        int last_word_index = array_length - 1;
        // Remove any question marks that are parsed as separate tokens
        if (strcmp(parsed_command0->line_array[last_word_index], "?") == 0)
            last_word_index--;
        int potion_name_length = 0;
        // Calculate potion name length
        for (int i = 2; i <= last_word_index; i++) {
            potion_name_length += (int) strlen(parsed_command0->line_array[i]); // Account for words
            if (i < last_word_index)
                potion_name_length++; // Account for whitespaces
        }
        potion_name_length++; // Account for "\0"

        char *potion_name = check(malloc(potion_name_length));
        char *write_pointer = potion_name; // For more efficiency compared to strcat

        // Build the potion name
        for (int i = 2; i <= last_word_index; i++) {
            char *token = parsed_command0->line_array[i];
            int length = (int) strlen(token);

            if (i == last_word_index && length > 0 && token[length - 1] == '?')
                length--; // Remove any trailing question marks

            memcpy(write_pointer, token, length); // Copy the word
            write_pointer += length;

            if (i < last_word_index) {
                *write_pointer = ' '; // Account for whitespaces
                write_pointer++;
            }
        }

        *write_pointer = '\0'; // Account for terminator


        // If the query is asking for a specific potion
        if (strcmp(potion_name, "") != 0) {
            potion *query = NULL;
            potion *current_potion = current_inventory->potion_inventory;
            // Get the potion in Geralt's inventory
            while (current_potion != NULL) {
                if (strcmp(current_potion->potion_name, potion_name) == 0) {
                    query = current_potion;
                    break;
                }
                current_potion = current_potion->next_potion;
            }
            // Print the quantity of the potion
            if (query != NULL) {
                printf("%d\n", query->potion_quantity);
            } else {
                // Always exit to avoid null pointer dereference for
```

```
                    query
                printf("0\n");
                free(potion_name);
                return;
            }
        }
        // If the query is asking for all potions in Geralt's inventory
        else {
            // Count the potions in the inventory
            int potion_count = 0;
            potion *current_potion = current_inventory->potion_inventory;
            while (current_potion != NULL) {
                if (current_potion->potion_quantity > 0) {
                    potion_count++;
                }
                current_potion = current_potion->next_potion;
            }


            if (potion_count == 0) {
                printf("None\n");
                free(potion_name);
                return;
            }

            // Allocate the potions array
            potion **potions = check(malloc(potion_count * sizeof(potion *)));

            // Populate it with potions
            int index = 0;
            current_potion = current_inventory->potion_inventory;
            while (current_potion != NULL) {
                if (current_potion->potion_quantity > 0) {
                    potions[index++] = current_potion;
                }
                current_potion = current_potion->next_potion;
            }

            // Sort and print the potions
            qsort(potions, potion_count, sizeof(potion *), compare_potions);
            for (int i = 0; i < potion_count; i++) {
                if (i != potion_count - 1)
                    printf("%d␣%s,␣", potions[i]->potion_quantity, potions[i]->potion_name);
                else
                    printf("%d␣%s\n", potions[i]->potion_quantity, potions[i]->potion_name);
            }

            free(potions);
            free(potion_name);
        }
    }
```

5 Results

There are two main test cases we tested our program with: The ones for executing valid commands and the ones that test the validity checking part. The combined inputs and their respective outputs are:

Inputs:

```
Geralt loots 8932 Ustun, 10 Ulas, 20 Cay, 30 Kek, 88 karpuz
Geralt encounters a alper
Geralt learns Igni sign is effective against      alper
Geralt encounters a alper
Geralt encounters a alper
Geralt encounters a alper
Geralt learns black blood potion consists of 100      Ustun, 10 Ulas , 5 Cay
Geralt brews black blood
Geralt learns black blood potion is effective       against vural
Geralt encounters a vural
Total potion black blood ?
Geralt encounters a vural
Total potion black blood ?
Geralt trades 1 vural, 1      alper trophy for 12 kahve     , 1 terlik, 3 tavuk
Total ingredient ?
Total ingredient Ustun ?
       Geralt        encounters      a      alper
Geralt learns black blood potion is effective against alper
Geralt brews black blood
Total potion black blood ?
Geralt encounters a alper
Total potion black blood ?
Exit
```

Outputs:

```
Alchemy ingredients obtained
Geralt is unprepared and barely escapes with his life
New bestiary entry added: alper
Geralt defeats alper
Geralt defeats alper
Geralt defeats alper
New alchemy formula obtained: black blood
Alchemy item created: black blood
New bestiary entry added: vural
Geralt defeats vural
0
Geralt is unprepared and barely escapes with his life
0
Trade successful
15 Cay, 30 Kek, 8832 Ustun, 12 kahve, 88 karpuz, 3 tavuk, 1 terlik
8832
```

```
Geralt defeats alper
Bestiary entry updated: alper
Not enough ingredients
0
Geralt defeats alper
0
```

6 Discussion

The solution runs in O(n) complexity where n is either the character number or the word number for the command. In some intermediate parts of the code, the performance may degrade to O(n^2) but these parts do not take in large parts of the input by nature, therefore they are omitted. A limitation is that since we have implemented the CFG by for/while loops rather than recursion, we have put many edge case handling methods, which reduces our flexibility. A possible improvement of the implemented solution would be to implement it using recursion for the given Backus-Naur form. However, debugging is easier in this for/while loop form.

7 Conclusion

The project helped teach us a lot about string operations, standard input/output, pointer arithmetic, and dynamic memory allocation in C. It also shows us how role-playing games like The Witcher may have worked in their early stages, giving an idea about how game-programming and state-storing works. Since we have also handled a lot of edge cases, this also improved our error-handling capabilities. A roadmap for such future enhancements would surely start with cleaning up the edge case handlers, implementing recursive grammar handling functions and closures for better abstraction.

References

[1] Gökçe Uludoğan Can Özturan. C programming language slides and ps. *Boğaziçi University*, 1(1):All pages, 2024.

AI Assistants

The only AI assistant used in this project was ChatGPT 4o. It was used in debugging and handling some string operations and parsing functions. No code was copied directly but ChatGPT also helped us understand the workflow behind some procedures involved. It was also used to correct for grammatical errors in this exact PDF document in LaTeX.