Project Report

Systems Programming - Spring 2025

Musa Kaan Güney, 2022400300
Hasan Yiğit Akıncı, 2022400138

April 14, 2025

1 Introduction

In the Witcher world, Geralt has lost all his memories and belongings. He must relearn alchemy, potion brewing, and monster-hunting techniques to survive. The main objective of the project is to design and implement a command-line interpreter in C that simulates Geralt's journey by tracking inventory, knowledge, and encounters based on structured user inputs.

Objectives

- Develop a console-based interpreter that:
  - Processes structured natural language commands.
  - Tracks and updates Geralt's inventory for ingredients, potions, and trophies.
  - Manages the knowledge about monsters and effective signs and potions for them.
  - Handles alchemy recipes for brewing potions.
  - Simulates monster encounters, validating Geralt's preparedness based on signs he knows and potions he owns.
  - Responds accurately to queries about Geralt's inventory, knowledge about recipes, and effective potions and signs against beasts.
- Ensure all input follows a strict grammar format and reject invalid entries.
- Comply with project constraints.

Motivation

This project requires systems programming skills allowing to:

- Gain experience with parsing and interpreting structured inputs.
- Handle state management and simulation in a user-driven environment.
- Appreciate the importance of syntax rules and command validation.

Moreover, it serves as a foundational exercise in building more complex interpreters or command-line tools, and reflects applications of language parsing, stateful systems, and modular programming in C.

2 Problem Description

Detailed Problem Description

The Witcher Tracker project is developing a command-line interpreter in C that simulates Geralt's skills and resources. After losing his memory, Geralt must relearn how to brew potions, encounter monsters, and loot resources. The program acts as a text-based simulation that processes user commands and maintains Geralt's inventory, knowledge, and progress.

System Behavior Overview

The program must support the following functions:

1. **Input Types**
   The program must handle three types of user inputs, which follow specific grammar rules:
   - **Sentence** – Actions or knowledge updates such as:
     - Inventory changes (Geralt loots 5 Rebis)
     - Alchemy (Geralt brews Swallow)
     - Learning (Geralt learns Black Blood potion consists of ...)
     - Monster encounters (Geralt encounters a Bruxa)
   - **Question** – Queries to retrieve current game state:
     - Total potion Swallow?
     - What is effective against Harpy?
     - What is in Black Blood?
   - **Exit command** – Ends the program:
     - Exit

Key Constraints

- **Strict grammar:** Inputs must exactly follow the defined Backus-Naur Form (BNF) grammar. Any deviation results in the response:

```
<< INVALID
```

- **Whitespace rules:** Inputs may include extra spaces between tokens (except where disallowed, like between words in potion names).
- **Case sensitivity:** All input tokens are case-sensitive.
- **Line length:** Each input line can be maximum 1024 characters long.
- **Execution time:** Total execution time for all queries must be under 30 seconds.

Expected Outputs

For every valid input, the program provides proper output. A few examples:

- **Action Sentences**

```
>> Geralt loots 5 Rebis
<< Alchemy ingredients obtained

>> Geralt brews Black Blood
<< Alchemy item created: Black Blood
```

- **Encounter Example**

```
>> Geralt encounters a Harpy
<< Geralt defeats Harpy

If Geralt has no effective potion or sign:
<< Geralt is unprepared and barely escapes with his life
```

- **Trade Example**

```
>> Geralt trades 2 Wyvern trophy for 8 Vitriol, 3 Rebis
<< Trade successful

If inventory conditions aren't met:
<< Not enough trophies
```

- **Query Example**

```
>> What is in Black Blood?
<< 3 Vitriol, 2 Rebis, 1 Quebrith

>> Total trophy?
<< 2 Bruxa , 1 Harpy
```

- **Invalid Input Example**

```
>> Geralt loots Rebis
<< INVALID
```

3 Methodology

Solution Approach

The project is implemented as a command-line interpreter in C, which processes user inputs to simulate Geralt's actions. The solution uses modular design, breaking the functionality into manageable components: input parsing, command dispatching, inventory management, bestiary handling, and alchemy operations.

Flowchart of Command Execution

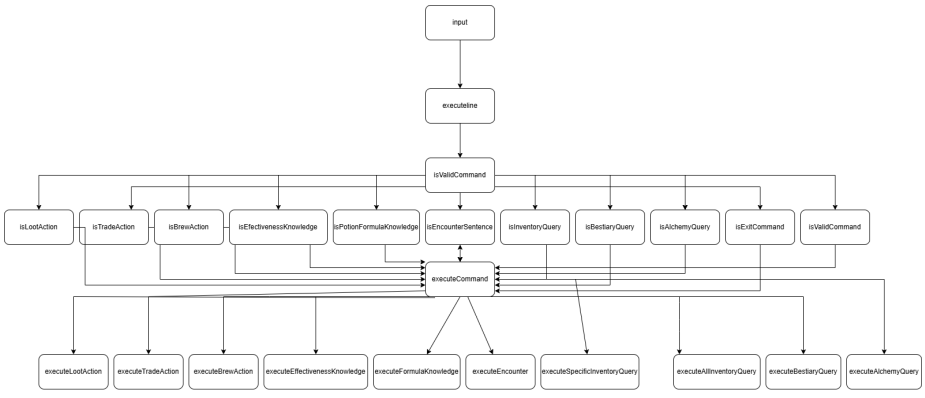

Figure 1: Flowchart of taking input and printing its output

Tokenizer Function Pseudocode

```
function tokenizeInput(input):
    Initialize tokens array, count = 0
    Skip leading whitespace

    // Handle "What" queries
    if input starts with "What":
        Add "What" to tokens
        count = 1
        Advance past "What"
        Skip whitespace

        // Check for "is in" (alchemy query) or "is effective against" (bestiary query)
        if next word is "is":
            Add "is" to tokens
            Advance past "is"
            Skip whitespace

            // Check for alchemy query
            if next word is "in":
                Add "in" to tokens
                Advance past "in"
                Skip whitespace

                // Read potion name until '?'
                Extract potion name from current position to '?' or end
                Add potion name to tokens

                // Add question mark if present
                if '?' found:
                    Add '?' to tokens
                    Parse any trailing tokens after '?'

                return tokens

            // Check for bestiary query
            else if next word is "effective":
                Add "effective" to tokens
                Advance past "effective"
                Skip whitespace

                if next word is "against":
                    Add "against" to tokens
                    Advance past "against"
                    Skip whitespace
```

```
                // Read monster name until '?'
                Extract monster name from current position to '?' or end
                Add monster name to tokens

                // Add question mark if present
                if '?' found:
                    Add '?' to tokens
                    Parse any trailing tokens after '?'

                return tokens

    // Handle legacy "What is in" query pattern if above didn't match
    Try to match expected sequence: "is", then "in"
    Read potion name
    Add '?' if present
    return tokens

// Handle "Total" queries
if input starts with "Total":
    Add "Total" to tokens
    count = 1
    Advance past "Total"
    Skip whitespace

    // Read category (potion/ingredient/trophy)
    Extract category word
    Add category to tokens
    Skip whitespace

    // Check for question mark right after category
    if next character is '?':
        Add '?' to tokens
        Parse any trailing tokens after '?'
        return tokens

    // Return if end of input
    if end of input reached:
        return tokens

    Skip whitespace

    // Check query type
    if next character is '?':
        Add '?' to tokens
        Parse any trailing tokens after '?'
```

```
            return tokens

        // Read item name until '?'
        Extract item name

        // Handle cases where '?' is attached to word
        if item name ends with '?':
            Split into name and '?' tokens
        else:
            Add item name to tokens
            Look for '?' and add it if found

        Parse any trailing tokens after '?'
        return tokens

    // Handle "Geralt" commands
    if input starts with "Geralt":
        Add "Geralt" to tokens
        count = 1
        Advance past "Geralt"
        Skip whitespace

        // Check for "brews" command
        if next word is "brews":
            Add "brews" to tokens
            Advance past "brews"
            Skip whitespace

            // Capture potion name
            Extract potion name
            Add potion name to tokens
            return tokens

        // Check for "learns" command
        else if next word is "learns":
            Add "learns" to tokens
            Advance past "learns"
            Skip whitespace
            Store position after "learns"

            // Look ahead for "sign" or "potion"
            Read words until "sign" or "potion" is found
            if "sign" or "potion" found:
                Extract name before "sign"/"potion"
                Add name to tokens
                Add "sign" or "potion" to tokens
```

```
        // Check for "is effective against"
        if next words match "is effective against":
            Add "is", "effective", "against" to tokens
            Extract monster name
            Add monster name to tokens
            return tokens

        // Check for "consists of"
        if next words match "consists of":
            Add "consists", "of" to tokens

            // Parse ingredient list
            while not end of input:
                Skip whitespace

                if digit found:
                    Extract quantity
                    Add quantity to tokens
                    Skip whitespace
                    Extract ingredient name
                    Add ingredient name to tokens
                else:
                    Extract word
                    Add word to tokens

                Handle comma if present

            return tokens

    return empty/error

// Check for "trades" command
else if next word is "trades":
    Add "trades" to tokens
    Advance past "trades"

    // Process remaining tokens
    while not end of input:
        Skip whitespace
        if comma found:
            Add comma as token
            continue

        Extract word until comma or space
        Add word to tokens
```

```
        return tokens

// Generic fallback tokenizer
Reset position to beginning
count = 0

while not end of input:
    Skip whitespace
    if end of input:
        break

    if comma found:
        Add comma as token
        continue

    Extract word until comma or space
    Add word to tokens

return tokens
```

High-Level Overview

This tokenizer handles different input patterns:

- **Query patterns**:
  - What is in [potion]? - Alchemy query
  - What is effective against [monster]? - Bestiary query
  - Total [category]? - List query
  - Total [category] [item]? - Specific item query
- **Command patterns**:
  - Geralt brews [potion]
  - Geralt learns [name] sign/potion is effective against [monster]
  - Geralt learns [name] sign/potion consists of [ingredients]
  - Geralt trades [items]
- **Generic fallback** for any input that doesn't match specific patterns

The function stores tokens in an array and returns the count of tokens found.

4 Implementation Details

The code is organized into several functions, each responsible for handling a specific type of action or query. The following subsections describe the main components of the program and the underlying data structures that support its functionality.

4.1 Code Structure and Components

Action Execution Functions: These functions handle specific actions that Geralt can perform. Each function parses the input command, updates the game state, and provides feedback to the user.

- `executeLootAction`: Handles looting actions where Geralt collects ingredients.
- `executeTradeAction`: Manages trading actions where trophies are exchanged for ingredients.
- `executeBrewAction`: Deals with brewing potions using known formulas and available ingredients.
- `executeEffectivenessKnowledge`: Updates the bestiary with information about effective signs or potions against specific monsters.
- `executeFormulaKnowledge`: Adds new potion formulas to Geralt's knowledge.
- `executeEncounter`: Simulates an encounter with a monster, checking if Geralt is prepared.
- `executeSpecificInventoryQuery`: Queries the inventory for a specific item.
- `executeAllInventoryQuery`: Lists all items in a specific category.
- `executeBestiaryQuery`: Retrieves information about a specific monster from the bestiary.
- `executeAlchemyQuery`: Provides details about the ingredients required for a specific potion.

Utility Functions: Functions such as `tokenizeInput` are used to parse input strings into tokens for easier processing.

4.2 Data Structures Overview

The program is designed to manage various aspects of Geralt's adventures, including his inventory, knowledge of potions, and encounters with monsters. The data is organized into several key structures, each serving a specific purpose.

Ingredient Structure

Purpose: Represents an alchemy ingredient that Geralt can collect and use in potion brewing.

Attributes:

- **Name:** A string representing the name of the ingredient.
- **Quantity:** An integer indicating how many units of the ingredient Geralt currently possesses.
- **Storage:** Ingredients are stored in a fixed-size array, with a counter to track the number of different ingredients.

Trophy Structure

Purpose: Represents a trophy obtained from defeating a monster, which can be used in trades.

Attributes:

- **Name:** A string representing the name of the trophy.
- **Quantity:** An integer indicating how many of this trophy Geralt has collected.
- **Storage:** Trophies are stored in a fixed-size array.

Potion Structure

Purpose: Represents a potion that Geralt can brew, including its recipe and availability.

Attributes:

- **Name:** A string representing the name of the potion.
- **Ingredients:** A list of indices pointing to the required ingredients, along with the quantities needed for each.
- **Ingredients Count:** An integer indicating the number of different ingredients required.
- **Quantity:** An integer indicating how many of this potion Geralt currently has.
- **Storage:** Potions are stored in a fixed-size array, with a counter to track the number of known potions.

Sign Structure

Purpose: Represents a magical sign that can be used against monsters.
Attributes:

- **Name:** A string representing the name of the sign.
- **Storage:** Signs are stored in a fixed-size array.

Beast Structure

Purpose: Represents a monster that Geralt may encounter, including known effective counters.
Attributes:

- **Name:** A string representing the name of the monster.
- **Effective Signs:** A list of indices pointing to signs known to be effective against the monster.
- **Effective Signs Count:** An integer indicating the number of effective signs.
- **Effective Potions:** A list of indices pointing to potions known to be effective against the monster.
- **Effective Potions Count:** An integer indicating the number of effective potions.
- **Storage:** Beasts are stored in a fixed-size array.

Storage and Initialization

Static Arrays: Each type of item or entity is stored in a static array, meaning the size is determined at compile time. This approach is chosen for simplicity and performance, avoiding the overhead of dynamic memory allocation.

Initialization: Arrays are initialized to zero, ensuring that all quantities and counts start at zero, and all strings are empty. This provides a clean slate for managing the game state.

Interaction with Functions

The program's functions interact with these data structures to perform various tasks, such as adding new items, checking quantities, updating knowledge, and handling encounters.

Indices are used within the potion and beast structures to efficiently reference ingredients and effective counters, facilitating quick lookups and updates.

This structured approach allows the program to efficiently manage Geralt's inventory, knowledge, and encounters, providing a robust framework for the game's mechanics.

5 Results

Test Case 1: Loot Action

Input: Geralt loots 3 mandrake, 2 wolfsbane
Output: Alchemy ingredients obtained
Explanation: Geralt collects 3 units of mandrake and 2 units of wolfsbane, which are added to his inventory.

Test Case 2: Trade Action

Input: Geralt trades 1 griffin trophy for 5 mandrake
Output: Trade successful
Explanation: Geralt trades 1 griffin trophy for 5 units of mandrake. The inventory is updated accordingly.

Test Case 3: Brew Action

Input: Geralt brews Swallow
Output: Alchemy item created: Swallow
Explanation: Geralt successfully brews a potion named Swallow, assuming he has the necessary ingredients.

Test Case 4: Effectiveness Knowledge

Input: Geralt learns Igni sign is effective against Drowner
Output: New bestiary entry added: Drowner
Explanation: The program updates the bestiary to indicate that the Igni sign is effective against Drowners.

Test Case 5: Potion Formula Knowledge

Input: Geralt learns Swallow potion consists of 1 celandine, 1 drowner brain
Output: New alchemy formula obtained: Swallow
Explanation: Geralt learns the formula for the Swallow potion, which is added to his known recipes.

Test Case 6: Encounter

Input: Geralt encounters a Drowner
Output: Geralt defeats Drowner
Explanation: Geralt encounters a Drowner and successfully defeats it, assuming he has the necessary effective counters.

Test Case 7: Specific Inventory Query

Input: Total potion Swallow?
Output: 1
Explanation: The program queries the inventory for the number of Swallow potions, returning the quantity.

Test Case 8: All Inventory Query

Input: Total ingredient?
Output: 2 celandine, 3 mandrake, 2 wolfsbane
Explanation: The program lists all ingredients in the inventory with their quantities.

Test Case 9: Bestiary Query

Input: What is effective against Drowner?
Output: Igni
Explanation: The program lists all known effective counters against Drowners.

Test Case 10: Alchemy Query

Input: What is in Swallow?
Output: 1 celandine, 1 drowner brain
Explanation: The program lists the ingredients required to brew the Swallow potion.

6 Discussion

Performance Analysis

Efficiency

- The program uses static arrays for storing data, providing fast access and manipulation of elements. This results in efficient time complexity for operations like lookups and updates, typically O(1) due to direct indexing.
- The use of fixed-size arrays avoids the overhead of dynamic memory allocation, beneficial in systems programming where performance is critical.

Scalability

- The program is limited by the fixed sizes of its arrays (e.g., `MAX_INGREDIENTS`, `MAX_POTIONS`). This restricts the number of items it can handle, which may not suffice for larger datasets or more complex scenarios.

Responsiveness

- The command parsing and execution are designed to be responsive, providing immediate feedback for valid and invalid commands, which is important for maintaining a smooth user experience.

Limitations

Fixed Capacity

- The use of static arrays with fixed capacities limits the number of items (ingredients, potions, etc.) that can be managed, posing a significant limitation if the game world expands or more complex scenarios are introduced.

Error Handling

- While the program checks for invalid commands and provides feedback, the error handling could be more robust. More detailed error messages could help users understand why a command failed.

Flexibility

- The program's design is somewhat rigid due to its reliance on fixed-size arrays and predefined command structures, making it less adaptable to changes in requirements or the addition of new features.

User Interface

- The command-line interface, while functional, may not be the most user-friendly option for all users. A graphical user interface (GUI) could enhance usability.

Possible Improvements

Dynamic Data Structures

- Implementing dynamic data structures, such as linked lists or dynamic arrays, could allow the program to handle a larger and more variable number of items, improving scalability and flexibility.

Enhanced Error Handling

- Providing more informative error messages and handling edge cases more gracefully could improve the user experience and make the program more robust.

Performance Optimization

- While the current implementation is efficient for small datasets, profiling the program to identify bottlenecks and optimizing critical sections could further enhance performance, especially if the program is scaled up.

7 Conclusion

Project Summary

The Witcher Tracker program is a command-line application designed to simulate inventory and knowledge management for Geralt, a character from "The Witcher" series. The program allows users to execute various commands related to alchemy ingredients, potions, monster encounters, and knowledge acquisition. It efficiently manages Geralt's inventory and bestiary using static data structures, providing immediate feedback on actions and queries.

Key Features

- **Inventory Management**: Track and update quantities of ingredients, potions, and trophies.
- **Alchemy**: Learn and brew potions using known formulas and available ingredients.
- **Bestiary**: Record and query effective counters against monsters.
- **Command Parsing**: Interpret and execute a variety of user commands with immediate feedback.

Future Enhancements

Dynamic Data Structures

- Transition from static arrays to dynamic data structures like linked lists or hash tables to handle a larger and more flexible set of items. This would improve scalability and allow the program to adapt to more complex scenarios.

Enhanced Error Handling

- Implement more detailed error messages and robust error handling mechanisms. This would help users understand command failures and guide them in correcting input errors.

Modular Code Design

- Refactor the codebase to improve modularity, separating concerns such as command parsing, execution, and data management. This would facilitate maintenance and future development.

Performance Optimization

- Profile the program to identify performance bottlenecks and optimize critical sections. This would ensure the program remains efficient as it scales up or handles more complex operations.

Multiplayer or Networked Features

- Explore the possibility of adding multiplayer capabilities or networked features, allowing users to interact or trade with each other in a shared game world.

By implementing these enhancements, the Witcher Tracker program could become a more robust, scalable, and user-friendly application, capable of supporting a wider range of features and user interactions.