**Yahtzee**

Project in Prolog

Samman Bhetwal

Ramapo College of New Jersey

**1. Bug Reports**

1. **Known Bugs:**

* No known bugs

1. **Bugs during the demonstration**

* Same as Known bugs

**2. Feature Report**

1. **Missing Features**

* The computer player doesn’t announce prioritized categories based on the die values it receives.
* For the first roll, and if the computer receives dice values which can fill the following categories Yahtzee, Five Straight, Four Straight, and Full house it lists the range of points and gives context specific explanation to its decision and moves but does not for other dice values.
* For the second roll, it does not explain specific informational reason why it wants to pursue certain categories but gives generic information on why it is doing what it is doing.
* In the first roll, if the human player were to receive dice values which can fill the following categories Yahtzee, Five Straight, Four Straight, and Full house it lists the range of points and gives context specific explanation. However, for other dice values, it just gives generic help whether to keep duplicates or keep sequences based on the categories available in the scorecard.
* In subsequent rolls, it does recommend the category or categories, the dice values human should keep, the range of points that can be earned for each category.
* Does not allow the human to input the categories they may want to fill based on their dice roll. The program does it for them.

1. **Extra Features:**

* No extra Feature

**3. Data Structures**

In the Yahtzee project, various data structures have been utilized to manage the game state, player actions, and game logic. Below is an overview of the key data structures used in the project:

**1. Lists**

Lists are a core data structure in Prolog and have been extensively used in this program for various purposes, such as storing dice rolls, player names, and categories.

**2. List of Lists**

Lists of lists are used to store structured data with multiple attributes. Examples include:

**i. Scorecard**

The scorecard is represented as a list of key-value pairs where each key represents a scoring category, and the value represents the score for that category. This structure helps in easily retrieving and updating scores for each category.

Example representation:

[ones-0, twos-0, threes-0, fours-0, fives-0, sixes-0, three\_of\_a\_kind-0, four\_of\_a\_kind-0, full\_house-0, small\_straight-0, large\_straight-0, yahtzee-0, chance-0]

**ii. Player Scores**

Player scores are stored as a list of pairs, where each pair consists of a player and their score. This structure helps in easily retrieving and updating player scores.

Example representation:

[human-50, computer-45]

**iii. Game State**

The game state is represented as a structured term that includes the scorecard, the current round, and the list of players. This helps in managing the overall state of the game efficiently.

Example representation:

game\_state(ScoreCard, CurrentRound, Players)

**iv. Dice Rolls**

Dice rolls are stored as lists of integers, representing the values rolled. This structure is used for both manual and automatic dice rolls.

Example representation:

[1, 3, 5, 6, 2]

**3. Compound Terms**

Compound terms are used to represent complex data structures that involve multiple attributes. Examples include:

**i. Game State**

The game state is represented using a compound term game\_state/3, which encapsulates the current state of the game, including the scorecard, the current round, and the list of players.

Example representation:

game\_state([ones-0, twos-0, threes-0], 1, [player(name('Alice'), human), player(name('Bob'), computer)])

**ii. Player**

Players are represented as a list of player terms, where each player term contains information about the player, such as their name and whether they are a human or computer player.

Example representation:

[player(name('Alice'), human), player(name('Bob'), computer)]

**4. Key-Value Pairs**

Key-value pairs are used to store data that can be easily accessed and updated based on a key. This structure is particularly useful for the scorecard and player scores.

Example representation:

[ones-0, twos-0, threes-0]

**Summary**

These data structures help in organizing and managing the various aspects of the Yahtzee game, making the code more readable and maintainable. By using lists, lists of lists, compound terms, and key-value pairs, we can efficiently handle the game state, player actions, and game logic, ensuring a smooth and enjoyable gameplay experience.

**4. Log**

Yahtzee Project Development Log

November 24 - December 10

Total Hours Spent: 72 hours

November 24 (6 hours)

* Task: Initial project setup and planning.
* Details:
  + Created the project repository and set up the initial file structure.
  + Planned the main modules and their responsibilities.
  + Set up the main Prolog file (main.pl) and basic module imports.

November 25 (6 hours)

* Task: Implementing basic game initialization.
* Details:
  + Implemented play\_yahtzee/0 to start the game.
  + Added initialize\_scorecard/1 and initialize\_game/3 to set up the game state.
  + Created basic game loop structure in game\_loop/1.

November 26 (6 hours)

* Task: Implementing user input functions.
* Details:
  + Created iofunctions.pl to handle user input and output.
  + Implemented user\_wants\_to\_load\_game/1 to ask the user if they want to load a game.
  + Added get\_serial/1 and save\_game\_procedure/1 for game state serialization.

November 27 (6 hours)

* Task: Implementing dice roll functions.
* Details:
  + Created get\_manual\_dice\_roll/2 to prompt the user to enter dice rolls manually.
  + Ensured input validation to handle non-numeric and out-of-range values.
  + Added get\_auto\_die\_roll/1 for automatic dice rolls.

November 28 (6 hours)

* Task: Implementing game logic for player turns.
* Details:
  + Created turn.pl to handle player turns.
  + Implemented play\_turn/3 to simulate a player's turn.
  + Added logic to roll dice, keep dice, and check for sequences or duplicates.

November 29 (6 hours)

* Task: Implementing sequence detection.
* Details:
  + Created find\_sequences/2 and find\_sequence/2 to identify sequences in dice rolls.
  + Implemented has\_sequence/1 to check for sequences of length 3 or more.
  + Added keep\_sequence\_dice/3 to keep dice that form a sequence.

November 30 (6 hours)

* Task: Implementing duplicate detection.
* Details:
  + Created keep\_highest\_duplicates/3 to keep the highest duplicates in dice rolls.
  + Integrated duplicate detection with the main game logic.
  + Ensured that the game correctly handles both sequences and duplicates.

December 1 (6 hours)

* Task: Implementing computer player logic.
* Details:
  + Created computer.pl to handle computer player decisions.
  + Implemented computer:get\_dice\_roll/3 for automatic dice rolls.
  + Added logic for the computer to decide whether to keep sequences or duplicates.

December 2 (6 hours)

* Task: Implementing game state transitions.
* Details:
  + Ensured that the game state transitions correctly between rounds.
  + Implemented game:play\_round/2 to play a round and update the game state.
  + Added logic to check if the game is over and determine the winner.

December 3 (6 hours)

* Task: Implementing scorecard management.
* Details:
  + Created scorecard.pl to manage the scorecard.
  + Implemented scorecard:get\_possible\_categories/3 to get possible scoring categories.
  + Added scorecard:is\_category\_filled/2 to check if a category is filled.

December 4 (6 hours)

* Task: Implementing game serialization.
* Details:
  + Ensured that the game state can be saved and loaded correctly.
  + Implemented iofunctions:save\_game\_procedure/1 to save the game state.
  + Added iofunctions:get\_serial/1 to load the game state.

December 5 (6 hours)

* Task: Implementing user interface improvements.
* Details:
  + Improved user prompts and messages for better clarity.
  + Added more detailed instructions and feedback for the user.
  + Ensured that all user inputs are validated and handled correctly.

December 6 (6 hours)

* Task: Testing and debugging.
* Details:
  + Conducted extensive testing of all game functionalities.
  + Fixed bugs related to sequence detection and game state transitions.
  + Ensured that the game runs smoothly and correctly handles all edge cases.

December 7 (6 hours)

* Task: Finalizing game logic.
* Details:
  + Ensured that all game rules are correctly implemented.
  + Added final touches to the game logic and user interface.
  + Conducted final testing to ensure the game is ready for release.

December 8 (6 hours)

* Task: Documentation and code cleanup.
* Details:
  + Added comments and documentation to all code files.
  + Cleaned up the code to ensure readability and maintainability.
  + Prepared the project for release.

December 9 (6 hours)

* Task: Final review.
* Details:
  + Conducted a final review of the project.
  + Made any necessary last-minute adjustments.

**5. AI Assistance**

Since this is my first time writing code in Prolog, I used AI assistance in the following areas:

• **Learning The Coding Language:** I used ChatGPT learn Prolog including but not limited to its syntax, recursive structures, and list-manipulation methodologies. It helped me differentiate Prolog from C++ which helped me design my project in a logic-based programming paradigm.

• **Designing the Project Structure:** I explained to AI what my project structure in C++ was like and asked how I could accomplish a similar project structure for easier management of the various logics in the project.

• **Debugging Code:** Whenever I encountered errors, warnings, or unexpected behavior in my code, I used AI to identify and fix the issues. It helped me to track bugs, put print statements all over the code to understand how my project is running and where I am encountering the issues.

• **Brainstorming Strategy:** Since my C++ strategy required heavy reliance on data structures which are not available in Prolog I had to change my strategy altogether. Therefore, I would come up with strategies on my own and ask AI for the validity, robustness, and refinement of the strategy

**6. Screenshots**

**A screenshot of a computer

Description automatically generated**

Fig: Rolling to see who goes first

A screenshot of a computer

Description automatically generated

Fig: Computer assisting human to select a high point earning category

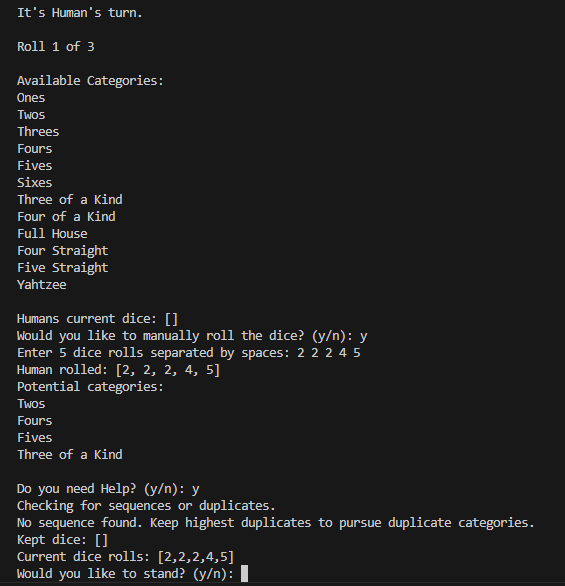


Fig: Computer assisting human to pursue keep dice values and pursue a category

A screenshot of a computer program

Description automatically generated

Fig: Computer making decision to keep dice values when high scoring category is available

A screenshot of a computer

Description automatically generated

Fig: Computer standing on a category, filling the scorecard, then asking if user wants to save game.

A screenshot of a computer screen

Description automatically generated

Fig: Winner of the tournament being announced

**Some Other images of the game:**

**A screenshot of a computer program

Description automatically generated**

Fig: Loading the game the

**7. How to Start**

- Make sure you have swipl prolog installed on your computer.

- Install Prolog runner extension in your visual studio code.

- main.pl contains the main function of the game. Click on Run Prolog code and the project will run on the terminal.