Faculdade de Engenharia da Universidade do Porto

# VLKNO

(*Volkano*)

***'Programação Funcional e Lógica'***

**Class 12 Group 3**

Gabriel Moura (up202200038) Participation: 50%

Gonçalo Sousa (up202207320) Participation: 50%

## About the Development

The game we chose to implement was the VLKNO (Volkano). Below on the report is described the game functioning with all the rules and conditions.

The working method we used to develop this project was pretty straightforward - divided the basic tasks, scheduled sporadic discord meetings when relevant work was developed and merge/discuss what or what not we wanted to keep. In the following list are the tasks each of us performed. Note that we focused on the task and not the predicates each of us implemented, since most of these had slight touches of each of us, this just refers to the global overview.

Gabriel:
- Initial game cycle logic and board display
- Initial pawn movements
- Difficult Bots and Bots vs Bots game modes

Gonçalo:
- Initial stone logic
- Plays with values
- Easy Bots

Both:
- Final board display
- Values attributed adjustments
- Game over situation
- Surrender option

## Installation and Execution

To download SICStus Prolog release 4.9, please visit the following URL:
- http://sicstus.sics.se/sicstus/products4/

Your license information for download 'linux' is as follows:

| | |
|---|---|
| Site name: | student.fe.up.pt |
| License code: | 4fa3-asxr-nobo-c5bd-t6k2 |
| Expiration date: | permanent |

This information is entered either during installation or by modifying your license information after installation. This is done by typing the following at a command prompt:

```
> splm -i student.fe.up.pt
> splm -a sicstus4.9_linux permanent 4fa3-asxr-nobo-c5bd-t6k2
```

Your license information for download 'win32' is as follows:

|  |  |
|---|---|
| Site name: | student.fe.up.pt |
| License code: | a5dg-asxr-nobo-b33p-ue2q |
| Expiration date: | permanent |

This information is entered either during installation or by modifying your license information after installation. This is done from the 'Settings' menu item in the SICStus application (spwin.exe).

After the installation of the Prolog in your environment, you will need to download and execute our game code, by using the following commands:

- git clone https://github.com/GabrielHenriqueSantosdeMoura/PFL_TP2_T06.git

Now, to execute the game you should open your SICStus application and consult the game.pl inside the src directory of the file that you cloned.

After it to start the game you should just write “play.” in the SICSTUS terminal.

## Description of the game

The VLKNO (Volkano) is a two-player abstract strategy game played on a 5x5 grid with 25 stackable stones.

Each player controls two pawns of their color, starting at diagonally opposite corners of the grid.

The goal is to strategically move your pawns and manipulate the stacks of stones on the board to trap your opponent, rendering them unable to make a legal move.

Gameplay Summary:

- Pawn Movement: On your turn, move one of your pawns to an adjacent stack of stones. You can only move to a stack that is the same height as the stack your pawn is currently on, one stone taller, or one stone shorter, and you can’t move your pawn to an unoccupied space.
- Stone Pickup: After moving, pick up a stone from the shortest unoccupied stack on the board. If multiple stacks qualify as the shortest, you choose which one. You cannot pick up from the stack your pawn just moved from.
- Stone Placement: Place the stone you picked up onto any pawn unoccupied stack on the board and where it has at least one stone.

Objective:

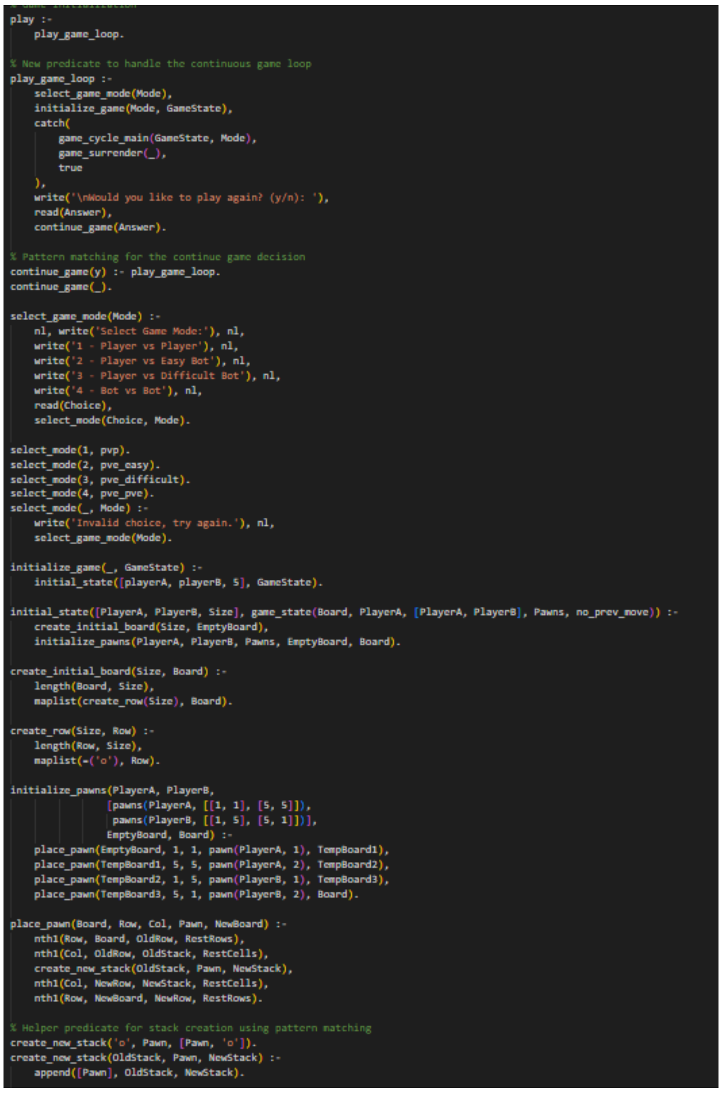
You win the game when your opponent cannot complete a valid move of at least one of his pawns.

By moving your pawns strategically and redistributing stones across the board, you create scenarios where your opponent has no valid moves, forcing them into a loss.

# Game Logic

## Game Configuration

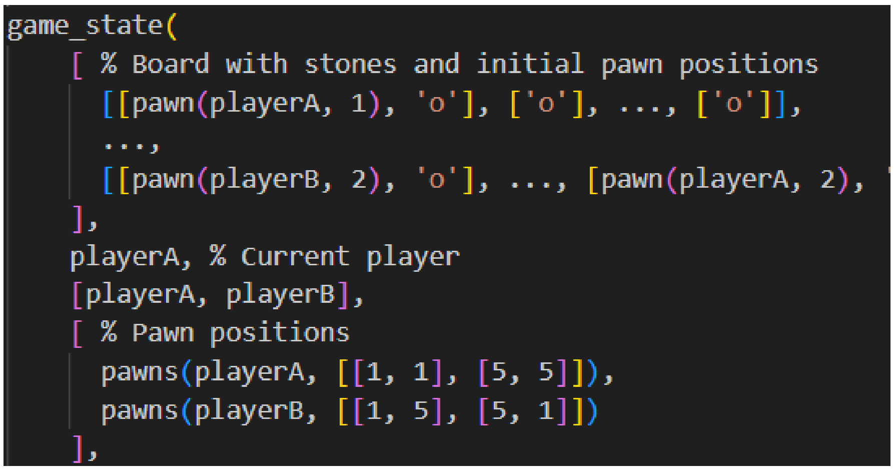
- Information Required: The configuration includes the 5x5 grid of stacks, the initial positions of the pawns, the current player, and a record of the last move.
- Internal Representation: The board is represented as a list of rows, where each row is a list of cells. Each cell contains a stack represented as a list of stones (['o', 'o', ...]) and possibly pawns (pawn(Player, Index)). The pawns are stored in a list indicating their positions on the grid. Players are stored in a list, and the current player is tracked separately.
- initial_state/2 Predicate: Initializes the game by creating an empty board, placing two pawns for each player in their starting positions, and populating the stacks with stones.



```
play :-
    play_game_loop.

% New predicate to handle the continuous game loop
play_game_loop :-
    select_game_mode(Mode),
    initialize_game(Mode, GameState),
    catch(
        game_cycle_main(GameState, Mode),
        game_surrender(_),
        true
    ),
    write('\nWould you like to play again? (y/n): '),
    read(Answer),
    continue_game(Answer).

% Pattern matching for the continue game decision
continue_game(y) :- play_game_loop.
continue_game(_).

select_game_mode(Mode) :-
    nl, write('Select Game Mode:'), nl,
    write('1 - Player vs Player'), nl,
    write('2 - Player vs Easy Bot'), nl,
    write('3 - Player vs Difficult Bot'), nl,
    write('4 - Bot vs Bot'), nl,
    read(Choice),
    select_mode(Choice, Mode).

select_mode(1, pvp).
select_mode(2, pve_easy).
select_mode(3, pve_difficult).
select_mode(4, pve_pve).
select_mode(_, Mode) :-
    write('Invalid choice, try again.'), nl,
    select_game_mode(Mode).

initialize_game(_, GameState) :-
    initial_state([playerA, playerB, 5], GameState).

initial_state([PlayerA, PlayerB, Size], game_state(Board, PlayerA, [PlayerA, PlayerB], Pawns, no_prev_move)) :-
    create_initial_board(Size, EmptyBoard),
    initialize_pawns(PlayerA, PlayerB, Pawns, EmptyBoard, Board).

create_initial_board(Size, Board) :-
    length(Board, Size),
    maplist(create_row(Size), Board).

create_row(Size, Row) :-
    length(Row, Size),
    maplist(=('o'), Row).

initialize_pawns(PlayerA, PlayerB,
                [pawns(PlayerA, [[1, 1], [5, 5]]),
                 pawns(PlayerB, [[1, 5], [5, 1]])],
                EmptyBoard, Board) :-
    place_pawn(EmptyBoard, 1, 1, pawn(PlayerA, 1), TempBoard1),
    place_pawn(TempBoard1, 5, 5, pawn(PlayerA, 2), TempBoard2),
    place_pawn(TempBoard2, 1, 5, pawn(PlayerB, 1), TempBoard3),
    place_pawn(TempBoard3, 5, 1, pawn(PlayerB, 2), Board).

place_pawn(Board, Row, Col, Pawn, NewBoard) :-
    nth1(Row, Board, OldRow, RestRows),
    nth1(Col, OldRow, OldStack, RestCells),
    create_new_stack(OldStack, Pawn, NewStack),
    nth1(Col, NewRow, NewStack, RestCells),
    nth1(Row, NewBoard, NewRow, RestRows).

% Helper predicate for stack creation using pattern matching
create_new_stack('o', Pawn, [Pawn, 'o']).
create_new_stack(OldStack, Pawn, NewStack) :-
    append([Pawn], OldStack, NewStack).
```

## Internal Game State

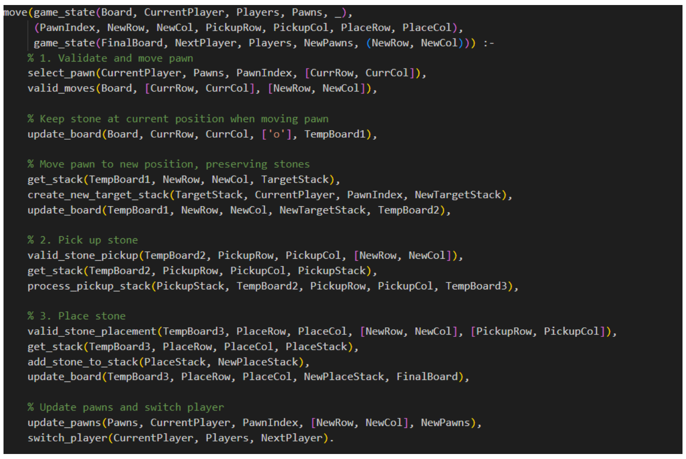
- Information Required: Board configuration, current player, list of players, pawn positions, and the previous move.
- Representation: Initial State: Pawns are at [1, 1], [5, 5] for Player A, and [1, 5], [5, 1] for Player B. All other cells have one stone.
- Intermediate State: Stones and pawn positions update dynamically based on moves.
- Final State: A state where one player cannot move.



```
game_state(
    [ % Board with stones and initial pawn positions
      [[pawn(playerA, 1), 'o'], ['o'], ..., ['o']],
      ...,
      [[pawn(playerB, 2), 'o'], ..., [pawn(playerA, 2),
    ],
    playerA, % Current player
    [playerA, playerB],
    [ % Pawn positions
      pawns(playerA, [[1, 1], [5, 5]]),
      pawns(playerB, [[1, 5], [5, 1]])
    ],
```

## Move Representation

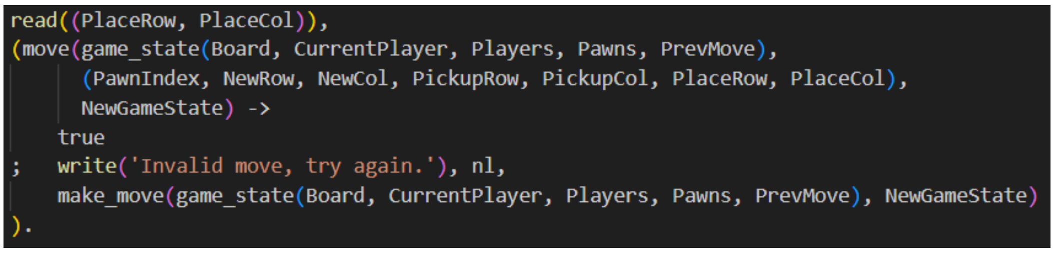
- Information Required: the pawn to be moved, its new position, the position to pick up a stone, the position to place the stone.
- Internal Representation: Moves are represented as a tuple: (PawnIndex, NewRow, NewCol, PickupRow, PickupCol, PlaceRow, PlaceCol).
- Usage in move/3: to validate the move we check if the pawn can move to the new position based on stack heights, ensure the picked-up stack is the shortest unoccupied stack and ensure the placed stone is valid. To apply the move we update the board to reflect the new pawn position, remove a stone from the picked-up stack and add it to the placed stack.



```
move(game_state(Board, CurrentPlayer, Players, Pawns, _),
     (PawnIndex, NewRow, NewCol, PickupRow, PickupCol, PlaceRow, PlaceCol),
     game_state(FinalBoard, NextPlayer, Players, NewPawns, (NewRow, NewCol))) :-
    % 1. Validate and move pawn
    select_pawn(CurrentPlayer, Pawns, PawnIndex, [CurrRow, CurrCol]),
    valid_moves(Board, [CurrRow, CurrCol], [NewRow, NewCol]),

    % Keep stone at current position when moving pawn
    update_board(Board, CurrRow, CurrCol, ['o'], TempBoard1),

    % Move pawn to new position, preserving stones
    get_stack(TempBoard1, NewRow, NewCol, TargetStack),
    create_new_target_stack(TargetStack, CurrentPlayer, PawnIndex, NewTargetStack),
    update_board(TempBoard1, NewRow, NewCol, NewTargetStack, TempBoard2),

    % 2. Pick up stone
    valid_stone_pickup(TempBoard2, PickupRow, PickupCol, [NewRow, NewCol]),
    get_stack(TempBoard2, PickupRow, PickupCol, PickupStack),
    process_pickup_stack(PickupStack, TempBoard2, PickupRow, PickupCol, TempBoard3),

    % 3. Place stone
    valid_stone_placement(TempBoard3, PlaceRow, PlaceCol, [NewRow, NewCol], [PickupRow, PickupCol]),
    get_stack(TempBoard3, PlaceRow, PlaceCol, PlaceStack),
    add_stone_to_stack(PlaceStack, NewPlaceStack),
    update_board(TempBoard3, PlaceRow, PlaceCol, NewPlaceStack, FinalBoard),

    % Update pawns and switch player
    update_pawns(Pawns, CurrentPlayer, PawnIndex, [NewRow, NewCol], NewPawns),
    switch_player(CurrentPlayer, Players, NextPlayer).
```

## User Interaction

- Game Menu System: Provides options for starting a new game with three modes of use: Human/Human, Human/Computer and Computer/Computer, in the Human/Computer mode the user has the option to choose an easy or difficult bot.
- Interaction with User: input prompts ask for the pawn to move, the new position, the stack to pick from, and the stack to place the stone.
- Input validation checks: ensure the input corresponds to a valid pawn index, verify the chosen positions follow the game rules and prompt the user to re-enter if invalid.



```prolog
read((PlaceRow, PlaceCol)),
(move(game_state(Board, CurrentPlayer, Players, Pawns, PrevMove),
      (PawnIndex, NewRow, NewCol, PickupRow, PickupCol, PlaceRow, PlaceCol),
      NewGameState) ->
    true
;   write('Invalid move, try again.'), nl,
    make_move(game_state(Board, CurrentPlayer, Players, Pawns, PrevMove), NewGameState)
).
```

## Conclusions

The implementation of the Prolog-based game has successfully demonstrated the use of logical programming to manage complex game mechanics, including the representation of game states, moves, and user interactions. This project highlights the flexibility and expressiveness of Prolog for creating board games, particularly in scenarios requiring non-linear, logic-driven decisions.

We had some limitations while doing this project, like the command-line interface that limits the game's appeal and accessibility, it lacks the visual and interactive appeal of graphical interfaces. Also, while comprehensive, the move validation process can become computationally expensive for larger boards or more complex rule sets.

Another barrier we found while making this project was the learning curve of this programming language - mastering the stack handling situations was really hard for us, especially when the stone logic was implemented.

In the end, this project turned out to be a challenge that we were successfully able to overcome. This taught us the basic/medium usage of Prolog language and helped us understand what declarative programming is.

## Bibliography

VLKNO - https://boardgamegeek.com/boardgame/427638/vlkno (Used to see the VLKNO game rules)
ChatGPT - https://chatgpt.com/ (Used to help us with some debugging stages to find errors)