Rummikub Lite

NAME: MAJOR:

Gerda is a big fan of the Rummikub board game, en would like to predict the outcome of a game. She asks you to develop a tool, which can predict the winning player of the game. The goal of the game is to place all your blocks (each consisting of both a number and a color) on the table (the game board). The game board consists of rows of blocks (each at least 3 blocks long), where either

- the numbers are equal and the colors are all different.
- the colors are equal and the numbers form a subsequent series.

On his / her turn, a player can create new rows with his / her blocks, add blocks to existing rows or (if he can't perform any other action) draw a new block from the bag. For this assignment we make a few small simplifications to the game:

- A player can only extend existing rows at its endpoints, and can't destruct existing rows.
- A player performs exactly 1 action per turn : either extend an existing row with 1 block, or place 1 new row (of length 3) on the board, or draw 1 new block.

Assignment 1 : Cheaters!

For this part of the assignment, you will write a predicate which can verify whether a given game board satisfies the rules. We want to make sure neither of the players are cheating.

We represent blocks with `block(Number,Color)` terms. For `Number` we just use the Prolog build-in integers. For `Color` we use one of four possible terms : `red`, `blue`, `yellow` or `black`. You can always assume that **each block is unique**. For example, there are multiple red blocks, and multiple blocks with the number 5, but there can only ever be one red 5 in the game.

Next up, we represent rows with either the `crow(Blocks)` term for a color row, or the `nrow(Blocks)` term for a number row, where `Blocks` represents the list of blocks in the row. Finally, the table (the game board) is represented as a list of all rows on the board.

Write a predicate `valid_table(Table)` which checks whether each row on the board satisfies the game rules. Namely, a row needs to be at least 3 blocks long. And secondly

- in a `crow`, each of the numbers has to be identical, and all colors need to be unique.
- in a `nrow`, all colors need to be identical, and the numbers need to form a subsequent, **sorted** series.

```
?- Table1 = [ crow( [block(5,red) , block(5,yellow) , block(5,black)] )
            , nrow( [block(5,blue) , block(6,blue) , block(7,blue) , block(8,blue)] ) ],
   valid_table(Table1).
True
```

```
?- Table2 = [ crow( [block(5,red) , block(5,yellow)] ) ],
   valid_table(Table2).
False

?- Table3 = [ crow( [block(5,red) , block(5,yellow) , block(4,blue)] ) ],
   valid_table(Table3).
False

?- Table4 = [ nrow( [block(2,blue) , block(3,blue) , block(4,blue) , block(5,red)] ) ],
   valid_table(Table4).
False

?- Table5 = [ nrow( [block(5,blue) , block(6,blue) , block(8,blue)] ) ],
   valid_table(Table5).
False

?- Table6 = [ nrow( [block(5,blue) , block(7,blue) , block(6,blue)] ) ],
   valid_table(Table6).
False
```

Assignment 2 : Play the Game

For this assignment you will write a predicate `play_game(P1,P2,Table,Bag)` to simulate a game. The variables `P1` and `P2` represent the two players, where it is currently the left player's (`P1`) turn. The players are represented by a `player(Blocks,Actions)` term, where `Blocks` is this player's current list of blocks, and `Actions` is the list of **all** subsequent actions this player will perform in the game. The argument `Table` is the current game board and `Bag` is the list of the blocks which can be drawn. For the tests, we consider the 2 `Blocks` lists, the `Table` and the `Bag` as given, and the 2 `Actions` lists as the 'output' of your predicate.

As previously mentioned, we make a small simplification to the game, by limiting players to performing exactly 1 action per turn. The active player thus performs 1 of these actions, adding this to his / her `Actions` list :

- `playrow(crow(Blocks))` : Create a new row `crow(Blocks)` consisting of exactly 3 blocks (from the player's supply), and place this on the game board. All blocks in `Blocks` have to contain the same number and have different colors. Sort this row using `sort`, to make the tests succeed!
- `playrow(nrow(Blocks))` : Create a new row `nrow(Blocks)` consisting of exactly 3 blocks (from the player's supply), and place this on the game board. All blocks in `Blocks` have to be the same color and have to contain subsequent (sorted) numbers.
- `playblock(Block,Row)` : Extend an existing row on the game board, forming a new row `Row`, by placing 1 block `Block` from the player's supply

  - on the start or end of an existing `nrow`.
  - in an existing `crow` and resorting the row.

- `draw(Block)` : Only in the case where the above actions are not possible, should the player draw 1 new block `Block`, namely the **first** block from `Bag`.

You can find an example of all possible actions in Table 1.

The game ends when one of the players has no blocks left, where this player wins the game. Add the final action `win` and `lose` for the winning and losing player respectively. In the case where both players are out of blocks, or in case a player has to draw a new block while the `Bag` is empty, the game ends on a draw and the action `draw` is added for both players.

Note that the actions are not assigned priorities. Your predicate should thus return **all possible solutions**.

Also note that you should **not** return the same action multiple times. You can use `sort/2` or `list_to_set/2` for this. You may assume that every block in the game is unique, and consequently that no two rows on the table can be identical.

Finally, to be clear : The order of your actions **is** important. First placing block A on the table and then placing block B in your next turn, is in fact a different solution from first placing block B and then playing block A afterwards.

The examples below have been indented for readability.

```
?- P1Blocks = [block(1,red),block(2,red),block(3,red),block(2,blue)],
   P2Blocks = [block(5,red),block(5,yellow),block(5,black),block(4,red)],
   play_game(player(P1Blocks,P1Act),player(P2Blocks,P2Act),[],[block(5,blue)]).

P1Act = [ playrow(nrow([block(1, red), block(2, red), block(3, red)]))
        , draw(block(5, blue))
        , draw],
P2Act = [ playblock(block(4, red),
              nrow([block(1, red), block(2, red), block(3, red), block(4, red)]))
        , playblock(block(5, red),
              nrow([block(1, red), block(2, red), block(3, red), block(4, red), block(5, red)]))
        , draw]

P1Act = [ playrow(nrow([block(1, red), block(2, red), block(3, red)]))
        , draw(block(5, blue))
        , lose],
P2Act = [ playblock(block(4, red),
              nrow([block(1, red), block(2, red), block(3, red), block(4, red)]))
        , playrow(crow([block(5, black), block(5, red), block(5, yellow)]))
        , win]

P1Act = [ playrow(nrow([block(1, red), block(2, red), block(3, red)]))
```

```
        , draw(block(5, blue))
        , lose],
P2Act = [ playrow(crow([block(5, black), block(5, red), block(5, yellow)]))
        , playblock(block(4, red),
             nrow([block(1, red), block(2, red), block(3, red), block(4, red)]))
        , win]

?- P1Blocks = [block(2,red),block(3,red),block(4,red),block(2,blue),block(2,black)],
   P2Blocks = [block(5,red),block(6,red),block(8,blue),block(8,black)],
   play_game(player(P1Blocks,P1Act),player(P2Blocks,P2Act),[],
             [block(7,red),block(8,red),block(9,red)]).

P1Act = [ playrow(crow([block(2, black), block(2, blue), block(2, red)]))
        , draw(block(8, red))
        , playblock(block(8, red),
             nrow([block(5, red), block(6, red), block(7, red), block(8, red)]))
        , playblock(block(4, red),
             nrow([block(4, red), block(5, red), block(6, red), block(7, red), block(8, red)]))
        , playblock(block(3, red),
             nrow([block(3, red), block(4, red), block(5, red), block(6, red), block(7, red),
                   block(8, red), block(9, red)]))
        , win],
P2Act = [ draw(block(7, red))
        , playrow(nrow([block(5, red), block(6, red), block(7, red)]))
        , draw(block(9, red))
        , playblock(block(9, red),
             nrow([block(4, red), block(5, red), block(6, red), block(7, red), block(8, red),
                   block(9, red)]))
        , lose]

P1Act = [ playrow(crow([block(2, black), block(2, blue), block(2, red)]))
        , draw(block(8, red))
        , playblock(block(4, red),
             nrow([block(4, red), block(5, red), block(6, red), block(7, red)]))
        , playblock(block(8, red),
             nrow([block(4, red), block(5, red), block(6, red), block(7, red), block(8, red)]))
        , playblock(block(3, red),
             nrow([block(3, red), block(4, red), block(5, red), block(6, red), block(7, red),
                   block(8, red), block(9, red)]))
        , win],
P2Act = [ draw(block(7, red))
```

```
        , playrow(nrow([block(5, red), block(6, red), block(7, red)]))
        , draw(block(9, red))
        , playblock(block(9, red),
              nrow([block(4, red), block(5, red), block(6, red), block(7, red), block(8, red),
                    block(9, red)]))
        , lose]

P1Act = [ playrow(crow([block(2, black), block(2, blue), block(2, red)]))
        , draw(block(8, red))
        , playblock(block(4, red),
              nrow([block(4, red), block(5, red), block(6, red), block(7, red)]))
        , playblock(block(3, red),
              nrow([block(3, red), block(4, red), block(5, red), block(6, red), block(7, red)]))
        , draw],
P2Act = [ draw(block(7, red))
        , playrow(nrow([block(5, red), block(6, red), block(7, red)]))
        , draw(block(9, red))
        , draw]

P1Act = [ playrow(nrow([block(2, red), block(3, red), block(4, red)]))
        , draw(block(7, red))
        , playblock(block(7, red),
              nrow([block(2, red), block(3, red), block(4, red), block(5, red), block(6, red),
                    block(7, red)]))
        , draw(block(9, red))
        , playblock(block(9, red),
              nrow([block(2, red), block(3, red), block(4, red), block(5, red), block(6, red),
                    block(7, red), block(8, red), block(9, red)]))
        , draw],
P2Act = [ playblock(block(5, red),
              nrow([block(2, red), block(3, red), block(4, red), block(5, red)]))
        , playblock(block(6, red),
              nrow([block(2, red), block(3, red), block(4, red), block(5, red), block(6, red)]))
        , draw(block(8, red))
        , playblock(block(8, red),
              nrow([block(2, red), block(3, red), block(4, red), block(5, red), block(6, red),
                    block(7, red), block(8, red)]))
        , draw]

P1Act = [ playrow(nrow([block(2, red), block(3, red), block(4, red)]))
        , draw(block(7, red))
```

Table 1: Possible Actions

Action	Player Blocks	Table	New Table
`playrow(crow(Row))`	4 red, 4 blue, 4 black		4 black, 4 blue, 4 red
`playrow(nrow(Row))`	4 red, 5 red, 6 red		4 red, 5 red, 6 red
`playblock(Block,NRow)`	7 red	4 red, 5 red, 6 red	4 red, 5 red, 6 red, 7 red
`playblock(Block,NRow)`	3 red	4 red, 5 red, 6 red	3 red, 4 red, 5 red, 6 red
`playblock(Block,NRow)`	4 red	4 black, 4 blue, 4 yellow	4 black, 4 blue, 4 red, 4 yellow

```
        , playblock(block(7, red),
              nrow([block(2, red), block(3, red), block(4, red), block(5, red), block(6, red),
                    block(7, red)]))
        , draw(block(9, red))
        , lose],
P2Act = [ playblock(block(5, red),
              nrow([block(2, red), block(3, red), block(4, red), block(5, red)]))
        , playblock(block(6, red),
              nrow([block(2, red), block(3, red), block(4, red), block(5, red), block(6, red)]))
        , draw(block(8, red))
        , playrow(crow([block(8, black), block(8, blue), block(8, red)]))
        , win]
```

Assignment 3 : Predictions

In the previous assignment you wrote a predicate to simulate a possible game outcome. Now write a new predicate `count_wins(P1Blocks,P2Blocks,Bag,WinPercentage)` which simulates all possible outcomes for a given game, and returns the percentage of outcomes where the first player wins. The game always starts with an empty game board, the initial blocks of player 1 and player 2 are `P1Blocks` and `P2Blocks` respectively, and `Bag` is the list of blocks that can be drawn. You can assume that a given game always has at least 1

solution (`win`, `lose` or `draw`), so you don't have to take divisions by 0 into account.

```
?- P1Blocks = [block(1,red),block(2,red),block(3,red),block(2,blue)],
   P2Blocks = [block(5,red),block(5,yellow),block(5,black),block(4,red)],
   count_wins(P1Blocks,P2Blocks,[block(5,blue)],Res).

Res = 0

?- P1Blocks = [block(5,red),block(5,yellow),block(5,black),block(4,red)],
   P2Blocks = [block(1,red),block(2,red),block(3,red),block(2,blue)],
   count_wins(P1Blocks,P2Blocks,[block(5,blue)],Res).

Res = 1

?- P1Blocks = [block(2,red),block(3,red),block(4,red),block(2,blue),block(2,black)],
   P2Blocks = [block(5,red),block(6,red),block(8,blue),block(8,black)],
   count_wins(P1Blocks,P2Blocks,[block(7,red),block(8,red),block(9,red)],Res).

Res = 0.4
```

Good luck!