Declarative Programming Formative Assignment ’22-’23
Playing Tic-Tac-Toe in Prolog

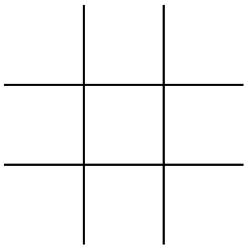
1 Introduction

This practical is the first formally assessed exercise for MSc students on the Prolog module. The intent is to implement a simple game – Tic-Tac-Toe – from first principles. Certain parts of the program (such as input/output) are supplied in library files.

This practical counts for 15% of the marks for this course. In itself, it is marked out of 100, and the percentage points available are shown at each section.

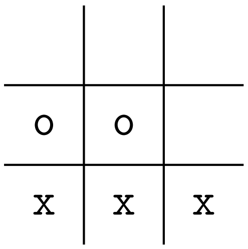
2 How to play the game

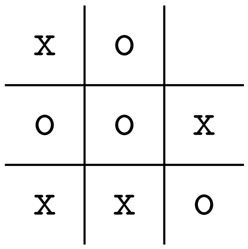
The rules of Tic-Tac-Toe are simple. We have a board which is 3×3 squares large, like this:



We will number the squares from left to right and top to bottom, so the top left is $(1, 1)$, the bottom left is $(1, 3)$ and the bottom right is $(3, 3)$.

There are two players, `o` and `x`, and they take turns in occupying successive squares on the board until someone wins or the board is full or stalemate is reached. One one player may occupy a square, and once taken a square stays taken. The winner of the game, if there is one, is the player who gets a complete line of pieces, across, down, or diagonally. Here are two examples, one of a win for `x` and one of a stalemate, where no-one can win:





There are some simple strategies which will help a computer win at Tic-Tac-Toe. They do not involve any real planning, but can often lead to a win, or at least hold off a loss. Applied in this order, they are:

1. If there is a winning line for self, then take it;
2. If there is a winning line for opponent, then block it;
3. If the middle space is free, then take it;
4. If there is a corner space free, then take it;
5. Otherwise, dumbly choose the next available space.

We will use these simple *heuristics* at the end of the practical.

3 The Implementation

3.1 Program structure

This practical is quite strictly structured, so as to give a feel for how good program design is done. Even if you are an experienced programmer, please follow the style here. In particular, you *must* follow the instructions for data-representation, or else the supplied code will not work.

3.2 Library software

In this practical, you will be provided the libraries `io` and `fill`, and you will likely use the library `lists`. You access the libraries by using the `use_module/1` predicate – just put the following at the top of your program file. Note that the `io.pl` and `fill.pl` files must be in the same directory as your source code.

```
:-  use_module(  [library(lists),
                  library( 'io' ),
                  library( 'fill' )] ).
```

The three library modules contain useful predicates, which saves you repeating other people's work. The `lists` library is documented in the SWIPL manual:

`https://www.swi-prolog.org/pldoc/man?section=lists`

The other two libraries are built specifically for this practical. The `io` library exports seven predicates, which work as follows:

`display_board/1` prints out a representation of the board, depending on what symbols you have used to represent noughts, crosses and blank spaces. Its argument is your representation of the board. If the representation is correct, it always succeeds. Argument: `Board`.

`get_legal_move/4` requests the coordinates of a board square, checks that it is empty, and returns the coordinates to the main program. It keeps asking until legal coordinates are given (*ie* a square on the board which is not already taken). If the representation is correct, it always succeeds. Arguments: `Player, Xcoordinate, Ycoordinate, Board`.

`report_move/3` prints out a move just selected. If the representation of the board is correct, it always succeeds. Arguments: `Player, Xcoordinate, Ycoordinate`.

`report_stalemate/0` prints out a warning that there is a stalemate and the game is drawn. It always succeeds.

`report_winner/1` prints out a warning that there is a winner – the player named in the one argument. It always succeeds. Argument: `Player`.

`welcome/0` prints out a welcome to the game. It always succeeds.

The `fill` library contains one predicate:

`fill_square/5` which takes a coordinate pair, a player, and a board representation, and replaces the square indicated by the coordinates with the piece for the player, to give a new board representation. Arguments: `Xcoordinate, Ycoordinate, Player, OldBoard, NewBoard`. It succeeds if the input representations are correct, and does not check for illegal moves.

You can look at the definitions of these predicates in the files `fill.pl` and `io.pl`, but you may not change them in any way.

3.3 The Practical

The following sections lead you through the practical step by step. You should be able to test your code at all times, and you will not need anything beyond what has been covered in the lectures on Basic Logic Programming (i.e., everything up difference lists). You do not need to understand how the library code works to complete the practical. It is *imperative* that you follow the instructions closely; otherwise, some of the library code, which uses your code, may not work.

3.4 Board representation (15%)

Design a representation for the board, using some kind of term representation. You will need a one-character symbol for each player and one for a blank space on the board. It needs to be one-character to fit in with the library software.

Implement the following predicates. Wherever possible, implement each predicate in terms of predicates you have already defined. Note that you may not need to use all these predicates in your final program, but some of them are used in the libraries. All of these predicates may be called in any mode – that is, you should not assume that any argument will be instantiated.

`is_cross/1` succeeds when its argument is the cross character in the representation.

`is_nought/1` succeeds when its argument is the nought character in the representation.

`is_empty/1` succeeds when its argument is the empty square character in the representation.

`is_piece/1` succeeds when its argument is either the cross character or the nought character.

`other_player/2` succeeds when both its arguments are player representation characters, but they are different.

`row/3` succeeds when its first argument is a row number (between 1 and 3) and its second is a representation of a board state. The third argument will then be a term like this: `row( N, A, B, C )`, where N is the row number, and `A, B, C` are the values of the squares in that row.

`column/3` succeeds when its first argument is a column number (between 1 and 3) and its second is a representation of a board state. The third argument will then be a term like this: `col( N, A, B, C )`, where N is the column number, and `A, B, C` are the values of the squares in that column.

`diagonal/3` succeeds when its first argument is either `top_to_bottom` or `bottom_to_top` and its second is a representation of a board state. The third argument will then be a term like this: `dia( D, A, B, C )`, where `D` is the direction of the line (as above), and `A, B, C` are the values of the squares in that diagonal. The diagonal direction (*eg* top-to-bottom) is moving from left to right.

`square/4` succeeds when its first two arguments are numbers between 1 and 3, and its third is a representation of a board state. The fourth argument will then be a term like this: `squ( X, Y, Piece )`, where (X,Y) are the coordinates of the square given in the first two arguments, and Piece is one of the three square representation characters, indicating what if anything occupies the relevant square.

`empty_square/3` succeeds when its first two arguments are coordinates on the board, and the square they name is empty.

`initial_board/1` succeeds when its argument represents the initial state of the board.

`empty_board/1` succeeds when its argument represents an uninstantiated board (*ie* with Prolog variables in all the spaces).

3.5 Spotting a winner (15%)

We need a predicate to tell us when someone has won.

`and_the_winner_is/2` succeeds when its first argument represents a board, and the second is a player who has won on that board. (Hint: use the predicates above here; you need 3 clauses.)

Test your predicate on some hand-made data.

3.6 Running a game for 2 human players (20%)

To start off with, we will build a program which acts as a board for two human players, displaying each move, and checking for a win or draw.

We will assume that `x` is always going to start. We will use a predicate called `playHH/0` to begin a game, defined as follows:

```
playHH  :-  welcome,
            initial_board( Board ),
            display_board( Board ),
            is_cross( Cross ),
            playHH( Cross, Board ).
```

You will need to define two predicates to make this work:

`no_more_free_squares/1` succeeds if the board represented in its argument has no empty squares in it.

`playHH/2` is recursive. It has two arguments: a player, the first, and a board state, the second. For this section of the practical, it has three possibilities:

1. The board represents a winning state, and we have to report the winner. Then we are finished.
2. There are no more free squares on the board, and we have to report a stalemate. Again, we are finished.
3. We can get a (legal) move from the player named in argument 1, fill the square he or she gives, switch players, display the board and then play again, with the updated board and the new player.

When you get to this point, test out your program thoroughly, playing several games, trying out the various possibilities for winning, drawing *etc.*

4 Running a game for 1 human and the computer (20%)

Having checked out the part of the program which runs the game and displays it, we now need to extend the program to play itself. We will assume that it will always play `o`.

Define another predicate called `playHC/0` to begin a game, defined similarly to `playHH/0`, except calling `playHC/2` instead of `playHH/2`.

`playHC/2` is the same as `playHH/2` for steps 1 and 2 above, but replaces step 3 with two possibilities:

- The current player is `x`, we can get a (legal) move, fill the square, display the board, and play again, with the new board and with nought as player.
- The current player is `o`, we can choose a move (see below), we tell the user what move we've made (see `io` library), we can fill the square, display the board, and play again, with the new board and with cross as player.

5 Implementing the heuristics (20%)

In order to make the computer play, we need to implement one more predicate:

`choose_move/4` which succeeds when it can find a move for the player named in its first argument, at the (X,Y)-coordinates given in its second and third arguments, respectively. It has five alternatives, corresponding with the heuristics given in section 2; the last clause looks like this:

```
% dumbly choose the next space
choose_move( _Player, X, Y, Board )  :-  empty_square( X, Y, Board ).
```

5.1 Spotting a stalemate (10%)

Finally, it would be nice to spot a stalemate as soon as it happens, so that the players don't have to go on until the whole board is full. To do this, you will need to replace step 2 of `playHH/2` or `playHC/2` as follows.

Define predicates `playSS/0` and `playSS/2` similarly to `playHC/0` and `playHC/2`, except for step 2.

`possible_win/2` is recursive. It succeeds if the addition of one square to the board (represented in the second argument) yields a win for a player (represented in the first argument). Alternatively, it succeeds if the result of adding one square to the board leads to a possible win, swapping players as it goes. In total, it succeeds if any player can win from the current position, allowing for whose move it is now.

`playSS/2` We replace the second step of `playHH/2` or `playHC/2` above as follows:

2. If no possible win is available, report a stalemate. Then we have finished.

6 Assessment

At the end, you should have three versions of the game corresponding to section 3 (`playHH/0, playHH/2`), section 4 (`playHC/0, playHC/2`), and section 5 (`playSS/0, playSS/2`). Though there is significant overlap between these different versions, they must all be able to run separately according to the specification in each section.

7 Documentation, Style, and Testing

Your code should have concise documentation in the comments for most predicates. This means you should describe the argument modes, like input (`+`), output (`-`), both (`?`), etc., (see https://www.swi-prolog.org/pldoc/man?section=preddesc), and a one or two line description of the predicate. There is no need for a paragraph description for each predicate. For example:

```
% append(?List1, ?List2, ?List1List2)
%   List1List2 is the concatenation of List1 and List2.
append(List1, List2, List1List2) :- ...
```

If a predicate has many different clauses, or the clauses are especially complex, it is also good to describe what each clause does separately, but this is not necessary in general. It is also helpful to indicate larger sections of the code that go together with a header.

Not only should your code conform to standard best practices of software engineering, like keeping predicates small, atomic, and non-redundant, but **your code should be written in declarative, idiomatic Prolog** as much as possible. This means taking advantage of unification and backtracking to leverage the built-in search in Prolog. If you find yourself frequently using `findall/3`, then rethink how you are approaching the problem to try to use backtracking and/or negation instead.

In terms of code formatting, please keep line length short. The recommended way to do this is to keep the head of the predicate and each goal in body in on its own line, with the body indented. See the provided library files for an example of this formatting.

Avoid using cuts (`!`) and if-thens (`->`). Using these actually means your program is logically incorrect. So, unless you provide a very good reason, you will be penalized for their usage. If you think you need to use a cut to stop unwanted backtracking, you can usually solve this by making your clauses mutually exclusive. If you think you need to use an if-then, you can usually just split it into separate clauses instead.

Your program should have also good test coverage in order to be confident that it behaves as desired. This means there should be decent amount of unit tests for the most important predicates (you don't need to unit test every single predicate) and integration tests that test larger and larger portions together. The best way to do this is to write test predicates that call your other predicates to ensure they are working. Please put these test predicates , or other evidence of testing, at the bottom of the source code; do not interlace them throughout.

8 Submission

You should combine together whatever source code you develop for the different parts of the exercise into a **single file** and submit it **via Canvas by 14/11/2022**.

Please ensure that you place any text in your code inside comment markers, so that the file will load without further editing. You will be penalized if you fail to do this. Also ensure that the program loads without errors and warnings. The submission must be made by the advertised deadline.

9 Referencing and Cheating

When programmers are stumped on a bug or issue, we often go straight to Google and adapt the resulting hits to our needs, often directly from StackOverflow. Being able to specify and solve your issues in this way is often useful, especially when working in the industry. However, in the academic setting, this makes it difficult to not only evaluate your competency as a Prolog programmer, but also to identify which code is actually yours! Therefore, if you need to adapt code from the web, simply provide a link to that resource (i.e. URL) along with a one-line description of where it's from and what it is. For example,

```
% StackOverflow: All combination of the elements of a list
% https://stackoverflow.com/questions/41662963
combs([], [])
... and so on.
```

Doing this whenever you take code from online removes any suspicion of cheating. **Any failure to do so will be regarded as cheating!** In addition, **you must not share code**; therefore, you must not reference another student who has taken or is currently taking the course. There are a few places where refering in this manner is not required:

- Lecture slides (Encouraged!)
- SWIPL built-ins, libraries, and documentation (Encouraged!)