Lab 3 – Instructions

This lab is about working with trees representing games of a simple strategy game, Tic-tac-toe. Two players, X and O, take turns filling empty squares on a 3x3 grid. The objective is to put three of one's own symbol in a horizontal, vertical, or diagonal line, before the opponent manages to do the same. If all nine squares are filled without either X or O succeeding to do this, the game is a draw.

`https://en.wikipedia.org/wiki/Tic-tac-toe`

1. Background

In our **game trees**, each node is a (Clojure) map, of the following form:

`{:board` *board* `:player` *player* `:win` *winner* `:children` *<list of game trees>*`}`

A *board* is a vector of nine values, each either X, O or _, where _ represents an empty square, with the three squares of the first row followed by those of the second followed by those of the third. So, for example, the board position in the picture above would be the vector [O X X X O O X O O].

A *player* is either X or O, the *winner* is either X, O or _, where _ means that the game is a draw. The *list of game trees* are the trees resulting from the *player* making a move on the *board*. For example, if the player is X and the board is [_ _ _ _ _ _ _ _ _], i.e. completely empty, the children would be game trees whose boards would be [X _ _ _ _ _ _ _ _], [_ X _ _ _ _ _ _ _], and so forth.

The root of the game tree is a node containing the initial board, which is normally empty. However, you can also create a tree with a partially occupied board, to explore different positions, such as [_ O _ _ X _ _ _ _] and [O _ _ _ X _ _ _ _], which result when X makes a move into the center square, and O responds either in the middle of the top row, or the top left corner, respectively.

X or O win if they can place three Xs or Os in a row, horizontally, vertically, or diagonally. In that case, they become the player in the `:win` field of the corresponding game tree node. The game tree ends at that point, i.e. there will be no children of a node that is a winning position. Also, if the board is full, i.e. there is no square that is _, it will have no children. If neither player won in that position, it is a draw, i.e. the `:win` field is _.

Based on this, it is recursively determined who wins in a given position as follows. If in a position it's player p's move, and there is at least one child node that is a win for p, then that position is also a win for p. Otherwise, if there is at least one child position that is a draw, then this position is a draw. Otherwise, all children are wins for p's opponent, and then so is this position. Once you have constructed a game tree t, you can therefore easily determine who will win by simply doing

```
(t :win)
```

In this lab, we will construct three different kinds of game trees. The first, built by the function `gametree`, contains all possible moves starting from a position. From this, we build a reduced

version, by filtering out the nodes that do not have the same winner as the root. This is then used to compute what we will call the “optimal” game tree, which is one where the winning player only makes moves that lead to a win as soon as possible, while losing players only make moves that push the end of the game out as far as possible (draws, of course, are always the same number of moves away, since they occur when the board is full). These are determined based on computing and comparing tree heights.

2. What to do

The logistics of this lab are similar to the previous ones:

1. Download the skeleton project and unpack it.

2. cd into its top-level directory, `edaa40lab3`. Start the Leiningen REPL there.

3. Have a look at the file

```
edaa40lab3/src/edaa40/lab3.clj
```

This is the file you are supposed to edit. As in previous labs, it contains commented-out incomplete code skeletons.

Your task is to implement all the definitions that have been commented out. Do not change the name or the parameter list, just add the function body or definition. Every commented-out function is preceded by a **declare** statement. Its purpose is to keep the compiler calm and allow you to load the package in spite of the fact that some functions initially remain undefined. If you try to call one of the declared-but-not-defined functions, you will get an error.

Right after some of the commented-out function skeletons are one or more **test?** statements. Their purpose is to help you check whether you are on the right track. Once you have an implementation, uncomment these tests, reload the package from the REPL (see below), and if the test passes (you will see something printed out on the REPL console) you might just have done it right.

As in previous labs, in order to load/reload the package, type in

```
(use 'edaa40.lab3 :reload)
```

in the REPL.

Do this lab step by step, from top to bottom in the source.
Make sure all the tests pass before you move on.

Once you have completed the lab, you have a few game trees, reduced game trees, and optimal game trees, the function `rand-moves` that generates a game from them (in the form of a list of board positions), and the function `print-boards` that prints a list of board positions in “readable” form. Try them out.

```
(print-boards (rand-moves B0-OGT))

(print-boards (rand-moves B1-OGT))

(print-boards (rand-moves B2-OGT))
```