CISC449 Project README

Synopsis: In this project, we designed a player versus computer tic tac toe game using C. The computer accepts a user input specifying a mark on a 3x3 square board. After this, the computer marks a spot on the board. The board will be marked by X's and O's. The computer states who wins once each square on the board is marked. If there is no winner, the computer states that there is a tie. A certain amount of functions were used to implement this game. The following list contains a description of each function, specifications, and what was able to be proven.

Functions:

initialize: This function is used to initialize the game. It takes in a two dimensional 3x3 board (matrix) and sets it up with empty chars. The board is proven valid along with empty spaces. The loop invariants of this function say that the conditions are true immediately before and after each iteration of the loops. The assigns condition assigns the board matrix which says that those expressions are the only variables in which the pre and post state may differ. The loop assigns condition assigns i, j, and the board matrix. This has the same effect, but with the variables in the loop. Loop variant N-i and loop variant N-j both prove that the procedure terminates and that the post conditions hold.

print_board: This function simply prints a 3x3 two dimensional board on the screen. We were able to prove that the values in the board may differ in the pre and post states. In addition, for every integer i, it was proven that its pre and post states may change and that the function will terminate with the postconditions held.

end_game: The end_game function is used to show the final result which is reported by the computer. For example, if the computer wins, the outcome shows "Winner is: Computer." This function assigns no variable. If the computer were to win the game, it is proven that the result will show the string "Winner is: Computer." If it were to be a tie game, then it is proven that the result will show the string "Tie game." Now, if the computer did not win, and it wasn't a tie game, the program is verified to compile.

comp_turn: The comp_turn function ensures that the computer's move is valid. It uses the minimax algorithm (specified later) which assures that the computer can never lose. It returns the end_game function that searches the board for matching marks.

player_turn: This function allows the user to pick a spot on the board. It scans the user's mark and prints the new board with the chosen mark. If the user chooses an invalid move, the computer will respond with "Wrong selection, try again" and will reprint the board's previous state. The function requires a valid read of the board and doesn't assign any variables.

gridTurn: The gridTurn function calculates the corresponding row and column coordinates from the grid number. It requires a valid read of the board and assigns it as well. We were able to prove the correct column transpositions regarding the grid numbers for the iterations. In addition, each integer value associated their contract was proven correct.

coordTurn: The coordTurn function checks the grid to make sure the coordinates are valid. We were able to prove this function using behavior clauses. This explains how each case will end up. For example, we proved that a valid coordinate contains x and y values associated with the values represented on the grid. We were able to specify what content makes an empty space on the board and what happens when the values are false. In addition, we specified each complete behavior in the function.

win_check: This function checks each and every direction of the board in order to declare the outcome of the game. The function checks the rows and columns to find matching marks in which would return the final state of the game. This required a valid read of the board. For each direction of the board, we proved each outcome that it would result in. For example, if a mark were to have three of a kind in a row, the game state would return and show the winner. However, if there are no three marks of a kind, then the game state would return and show that there is a tie. Since there's a loop in the function, it is proven that the postconditions hold and that the procedure terminates.

diag_check: This function helps the win_check function. Its purpose is to look for matching marks of a kind in the left and right diagonal parts of the grid. Similar to the win_check method, we proved that for each diagonal intersection that contains a certain mark of a kind, the game state would return and show the winner. This was proven with behaviors.

tie_check: This function's purpose is to check the grid for marks indicating a tie using nested loops. It is proven that the truth of the formula is preserved by the loop for every invariant listed. In addition, we proved that for n-i and n-j, the postconditions hold and that the procedure terminates. This means that integers i and j strictly decrease with each iteration until the procedure terminates. Also, we proved that if the board is incomplete, that means that there is an open spot yet to be marked.

minNum: The minNum function is used for artificial intelligence. This function also aids the maxNum and minimax functions described later, and is a big factor in the minimax algorithm. This function takes in a player and a valid read of the board. The minNum of a player is the smallest value that the other players can force the player to receive without knowing the actions. In other words, it is the largest value the player can be sure to be given when they know the actions of other players. The order of the minNum and maxNum operators is inversed. Using behaviors, we proved that the game result equals a certain value for each test.

Each part of the function was able to be proved. If player i (computer) tries to maximize their value before knowing what the others will do, then player i is in a better position. This means that it maximizes their value knowing what the other player did.

maxNum: The maxNum function is similar to minNum but it makes sure that max (computer) is always at the top of the minimax tree. The maxNum of a player is the highest value that the player can be certain to get without knowing the action of the other player. In other words, it is the lowest value the other player can force the player to obtain when they know the player's operation. Calculating this value of a player is done in the worst-case approach. Similarly to minNum, we proved that the game result equals a certain value for each test. It requires a valid read of the board and assigns the game result and board. This means that the variables listed may have values in the pre and post state that differ. Also, the loop invariants were able to be proven for each loop. In addition, it is proven that the post conditions will hold after the procedure is terminated by calling the N-i and N-j loop variants. Each part of the function was able to be proved.

new_board_check: This function aids the artificial intelligence functions minNum, maxNum, and minimax. Its task is to check the new board for variables that will help the computer to complete its next move. In other words, this function responds to the user's move. It requires a valid board and assigns a new board. This means that the boards' pre and post conditions are the only ones that may differ. We were able to prove that the formula's truth is preserved by the loop for integers x and y. In addition, it is proved that when the procedure terminates, the postconditions hold. The N-x and N-y loop variants indicate that the integer strictly decreases until procedure termination.

minimax: The minimax function is in charge of the computer's movements. It also completes the ultimate goal: the computer can never lose. This artificial intelligence function was implemented in order to make the game unbeatable for the player. The key to this algorithm is a back and forth between two players (computer and user), where the player whose turn it is picks the move with the max score. The scores for each of the obtainable moves are determined by the opposing player currently taking their turn and deciding which obtainable move has the minimum score. The scores for opposing moves are once again determined by the player currently taking their turn trying to maximize their score and so on. This procedure's end state is when the players move all of the way down the tree. This algorithm is recursive and it goes back and forth between the computer and user until the final score is determined. If there were to be an error in this algorithm, then the string "Minimax error" would appear. This function requires a valid read of the board. We proved that for each loop, the formula's truth is preserved by the loop. It is also proven that the postconditions hold when the procedure terminates. With each iteration of the loop, the variables listed will strictly decrease until the

procedure ends. For example, for the loop variants N-i and N-j, the integers will decrease until termination. Each part of this function was able to be proved. The following photos are visual representations of the minimax algorithm used in this game.