**(S1-23\_AIMLCZG557)**

**(Artificial and Computational Intelligence)**

**Academic Year 2023-2024**

**Assignment 2 – Problem Statement-9**

Contents

[**Contribution** **Table** 2](#_Toc161946656)

[**Problem Statement 1 (3 player tic tac toe)** 2](#_Toc161946657)

[**PEAS** 3](#_Toc161946658)

[**Properties of Environment** 3](#_Toc161946659)

[**Justification of the design of logic** 4](#_Toc161946660)

[**Static Evaluation Function** 4](#_Toc161946661)

[**Changes to Minmax Algorithm/Code** 5](#_Toc161946662)

[**Stepwise program output:** 7](#_Toc161946663)

[Move1: X=(1,1) 7](#_Toc161946664)

[Move2: X=(2,2) 8](#_Toc161946665)

[Move3: X=(3,3) 8](#_Toc161946666)

[Move4: X=(4,4) (invalid cell) 9](#_Toc161946667)

[Move5: X=(5,5) 9](#_Toc161946668)

[Move6: X=(2,1) 10](#_Toc161946669)

[Move7: X=(3,1) (player O wins) 10](#_Toc161946670)

[**Problem Statement 2. (Prolog Logic)** 11](#_Toc161946671)

[Given Decision Tree. 11](#_Toc161946672)

[Note: To use DT as input 12](#_Toc161946673)

[Note: To use “happiness” as class variable: 12](#_Toc161946674)

[**Logic Design and Inferencing in Prolog** 13](#_Toc161946675)

[**Snapshot of output from prolog logic program:** 13](#_Toc161946676)

[RUN:0 14](#_Toc161946677)

[RUN:1 15](#_Toc161946678)

[RUN:2 15](#_Toc161946679)

[RUN:3 16](#_Toc161946680)

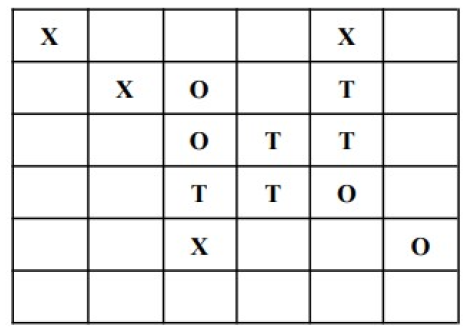
[RUN:4 (invalid response) 16](#_Toc161946681)

# **Contribution** **Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Sl. No.** | **Name (as appears in Canvas)** | **ID NO** | **Email ID** | **Contribution** |
| **1** | **Hemant Kumar Parakh** | **2023AA05741** | 2023aa05741@wilp.bits-pilani.ac.in | **100%** |
| **2** | **Sushil Kumar** | **2023aa05849** | 2023aa05849@wilp.bits-pilani.ac.in | **100%** |
| **3** | **NAGINENI SATISH BABU** | **2023aa05585** | 2023aa05585@wilp.bits-pilani.ac.in | **100%** |
| **4** |  |  |  |  |
| **5** |  |  |  |  |

# **Problem Statement 1 (3 player tic tac toe)**

Simulate the working of TicTacToe game extended to three players with coins ‘X’, ‘O’ and ‘T’ with the below sample larger game board of dimension 6\*6. The first player to match any four consecutive coins in the same row or same column or same diagonal wins. A sample state of the game board is given below for reference. Your program must start from empty configuration.



a. You are free to choose your own static evaluation function. Justify your choice of

static evaluation value design and explain with a sample game state. Do not use any machine learning model for the evaluation function.

b. Similar to the virtual lab example, one of the players must be a human ie., it must get dynamic inputs from us. The other two players must be simulated using the program.

c. Implement Python code for the design under part a, using Minimax Algorithm.

# **PEAS**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Agent | Performance Measure | Environment | Sensors | Actuators |
| 3 Player Tic Tac Tow Gamer | Winning, losing, or drawing the game based on the player's actions | 6x6 grid representing the tic-tac-toe board, state changes based on player actions | The sensors receive information about the current state of the game board, including the positions of X and O coins. It also informs about next play’s turn, and weather the game has ended with a draw/win or lose, in case of win/loss, it also reports which player win the game | Placing X, O or T coins on the grid for human and 2 other AI players using minmax and alpha beta puring techniques. Draws board after each placement. |

# **Properties of Environment**

- **Fully Observable**: The game board is visible to all agents, allowing them to make informed decisions based on the current state.

- **Multi** **Agents**: Players, including humans and AI, interact with the environment by selecting actions to place their respective symbols on the board.

- **Deterministic**: Player moves and game outcomes are entirely determined by the current state of the board and the actions taken, without randomness.

- **Episodic**: The game progresses in discrete episodes, with each episode consisting of sequential moves by the players until a win, loss, or draw occurs.

- **Static**: The static evaluation function assesses the strategic value of each player's position on the board without considering future moves.

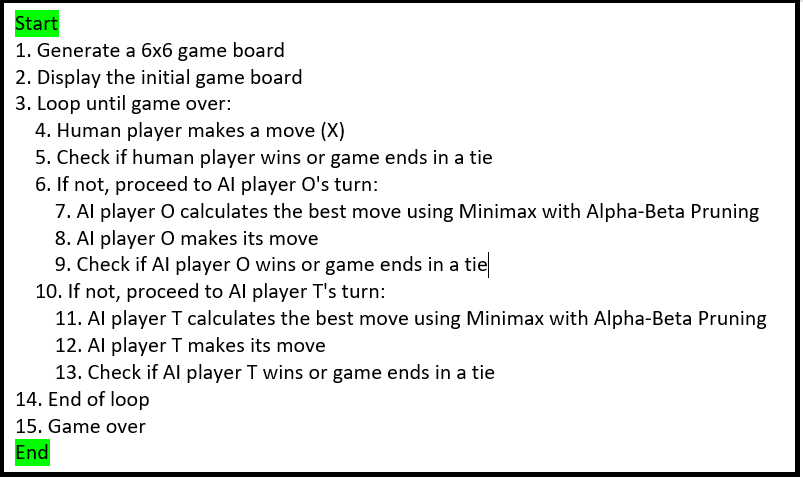
- **Discrete**: Player actions are discrete, involving selecting a cell on the grid to place their symbol, rather than continuous actions or movements.

# **Justification of the design of logic**

This code presented through the shared python notebook depicts a Three-Player Tic Tac Toe game employing the minimax algorithm enhanced with alpha-beta pruning to enhance efficiency. Below, we present the theoretical justification behind its design.

**Logic Design:**

* The game board is represented as a 6x6 grid initialized with empty cells.
* Functions are provided to generate the board, check for game over conditions, detect empty positions, display the board, check for consecutive pieces, detect winners, and evaluate the board for a given player.
* The game is played by human (X) against two AI players (O and T). Each player makes moves sequentially until a winner is determined or the game ends in a tie.
* The game uses Minimax and Alpha-Beta Pruning to help AI players make smart moves. Minimax looks at all possible moves to find the best one, while Alpha-Beta Pruning helps Minimax to ignore paths that won't lead to a good outcome, making the decision process faster and more efficient.
* An abstract flow of this game sequence has been attached below.



# **Static Evaluation Function**

The static evaluation function in this game evaluates the current state of the game board based on the presence of consecutive positions for each player and assigns scores, accordingly. These scores guide the AI players in making strategic moves to maximize their chances of winning the game, it also helps to minimize the chances of opponent’s winning. Here's the stepwise logic of the static evaluation function:

1. **Initialization**: Initialize variables score\_X, score\_O, and score\_T to keep track of the scores for each player ('X', 'O', and 'T' respectively). These scores will be updated based on certain conditions.

2. **Iterate Over the Game Board**: Loop through each cell of the game board.

3. **Check for Empty Cells**: If the current cell is empty (denoted by '-'), skip it and continue to the next cell.

4. **Evaluate Each Player's Positions**:

* For each non-empty cell, check if it belongs to any of the players ('X', 'O', or 'T').
* For each player:
  + Check for consecutive positions belonging to that player, both horizontally, vertically, and diagonally.
  + Assign scores based on the presence of consecutive positions:
    - Three consecutive positions (three in a row/column/diagonal): Assign a higher score (100 for 'X', 'O', and 'T') as these positions are strategically powerful and closer to winning.
    - Two consecutive positions: Assign a moderate score (50) to prioritize these positions, as they are also strategically valuable.
    - Opponent's consecutive positions: Deduct points if the opponent has consecutive positions, indicating a potential threat. Assign a score of 250 to block spaces next to the opponent's winning positions.
    - Opponent's two consecutive positions: Assign a score of 20 to block opponent's potential third position, which can lead to a win.
      * Detecting a winning move: Assign the highest score (1000) if the player has a winning move.

5. **Return Scores**: Return the accumulated scores for each player ('X', 'O', and 'T').

# **Changes to Minmax Algorithm/Code**

The Minimax algorithm used in the provided code is adapted for a Three-Player Tic Tac Toe game, which introduces additional complexity compared to the standard two-player Minimax algorithm used in games like regular Tic Tac Toe or Chess.

The Minimax function recursively explores possible moves up to a certain depth using the minimax algorithm with alpha-beta pruning. The algorithm considers the best move for the current player while minimizing the potential gains of opponents. To accommodate three players, the function dynamically determines the next player in each recursive call using the determine\_next\_virtual\_player()function.

Alpha-beta pruning is applied to improve performance by discarding branches that cannot lead to a better outcome than previously explored moves. The function returns the maximum evaluation value for the current player's perspective. Separate functions are provided to calculate the optimum move for each AI player calculate\_optimum\_move() based on the minimax algorithm.

Here's how the Minimax algorithm in the provided code differs from the standard two-player Minimax algorithm, along with the code changes made.

1. **Support for Three Players:**

In a standard Minimax algorithm for two-player games, the algorithm alternates between maximizing and minimizing players, with one player seeking to maximize the utility (score) of the board state and the other seeking to minimize it.

In this adapted Minimax algorithm for Three-Player Tic Tac Toe, there are three players: X, O, and T. Each player aims to maximize their own utility, while considering the moves of the other two players.

1. **Evaluation Function Modification:**

The static evaluation function in this game code is modified to accommodate the scoring criteria for a three-player game, where each player's moves are evaluated separately to determine their strategic value.

Additional scoring conditions are introduced to account for the presence of three consecutive positions and the need to block opponents' winning positions.

1. **Alpha-Beta Pruning:**

Alpha-Beta Pruning is a technique used to optimize the Minimax algorithm by pruning branches of the game tree that are deemed irrelevant to the final decision, thereby reducing the number of nodes that need to be evaluated.

This game code incorporates Alpha-Beta Pruning to improve the efficiency of the Minimax algorithm, allowing for quicker decision-making by the AI players.

1. **Next Virtual Player Determination:**

Since there are three players in the game, determining the next virtual player after each move becomes more complex compared to a two-player game. Additional functions are introduced to determine the next virtual player based on the current player's move.

1. **Code Structure:**

The overall structure of the Minimax algorithm is similar to the standard version, with recursive calls to evaluate potential moves and select the optimal one.

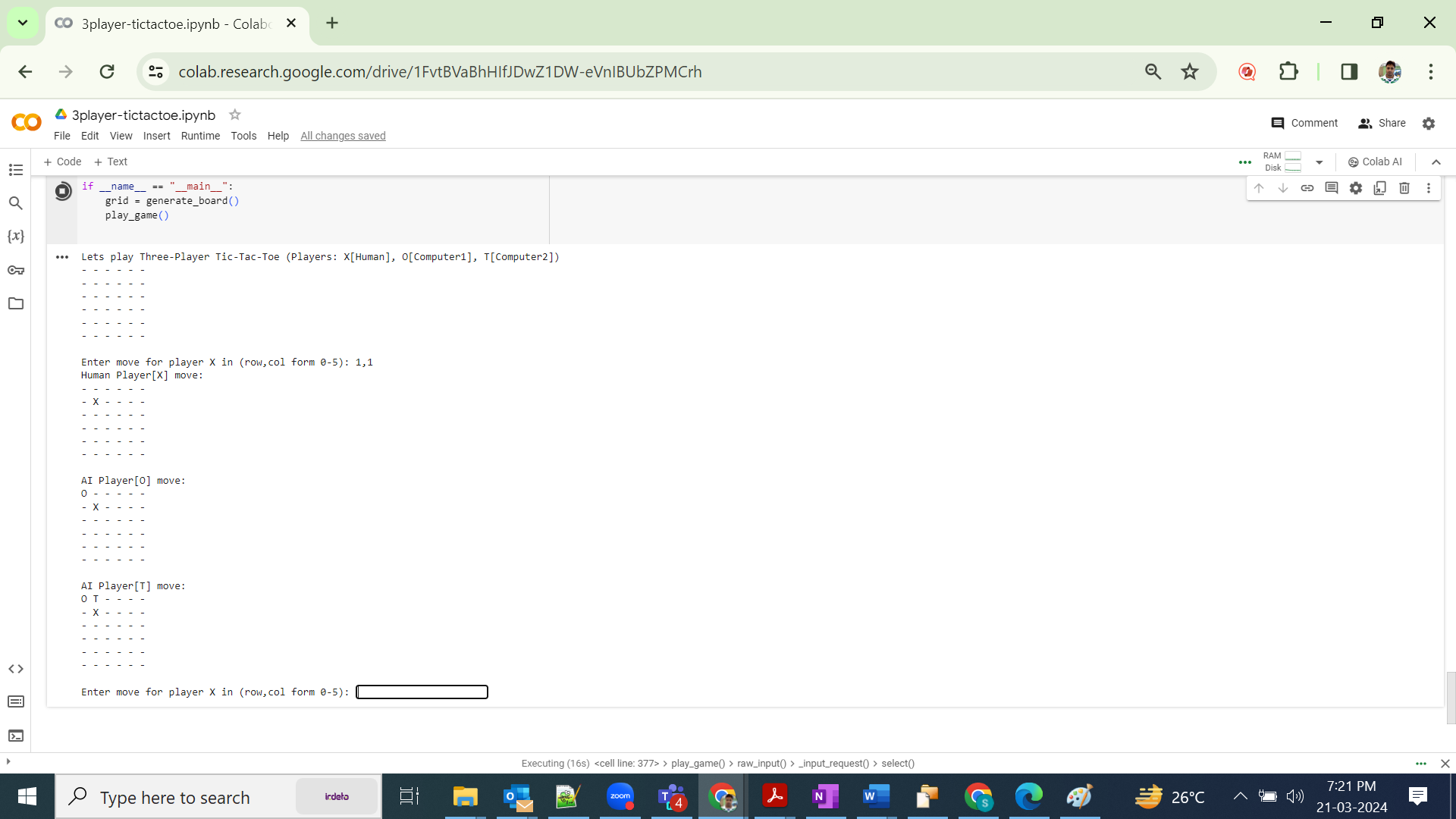
However, the implementation is adjusted to handle the three-player dynamics, including evaluating the game state for each player separately and considering the moves of all players in determining the best move. Game winning condition is changed to 4 consecutive coins by any player in a 6\*6 grid.

# **Stepwise program output:**

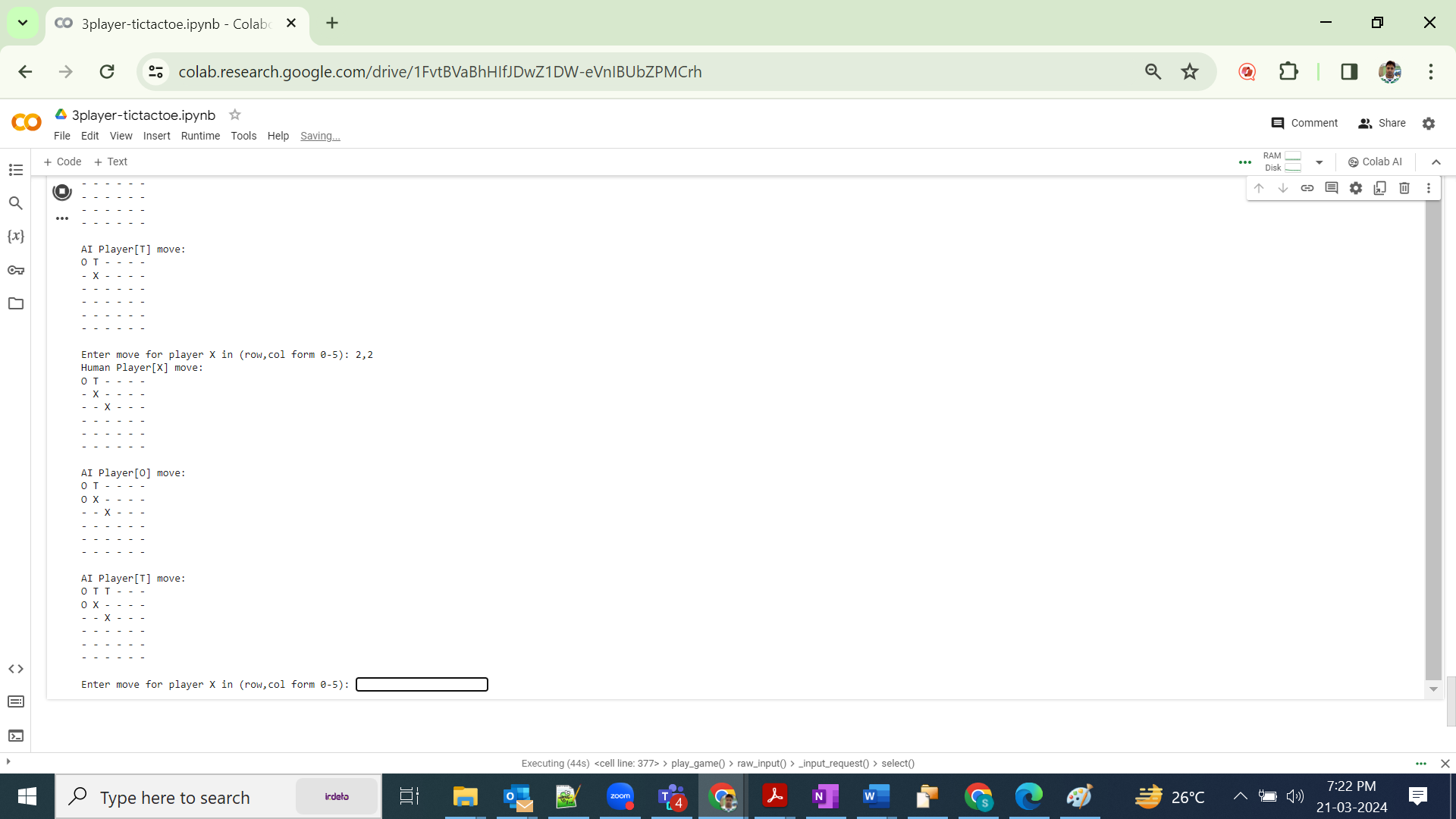
Attached below are the full screen snapshot of output of our game: 3 Player Tic Tac Toe:

**X: Human, O: Computer1, T: Computer2**

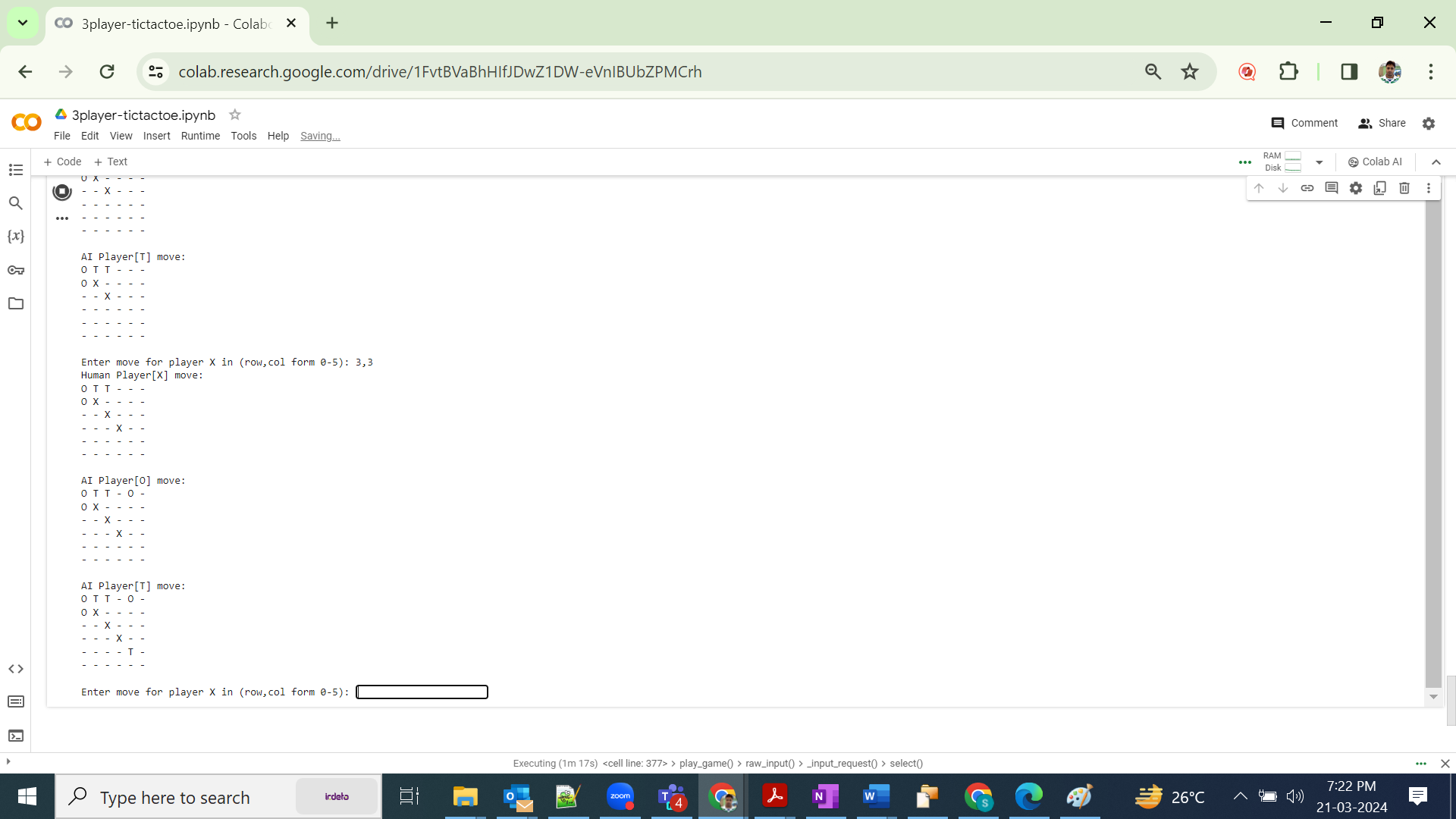
## Move1: X=(1,1)



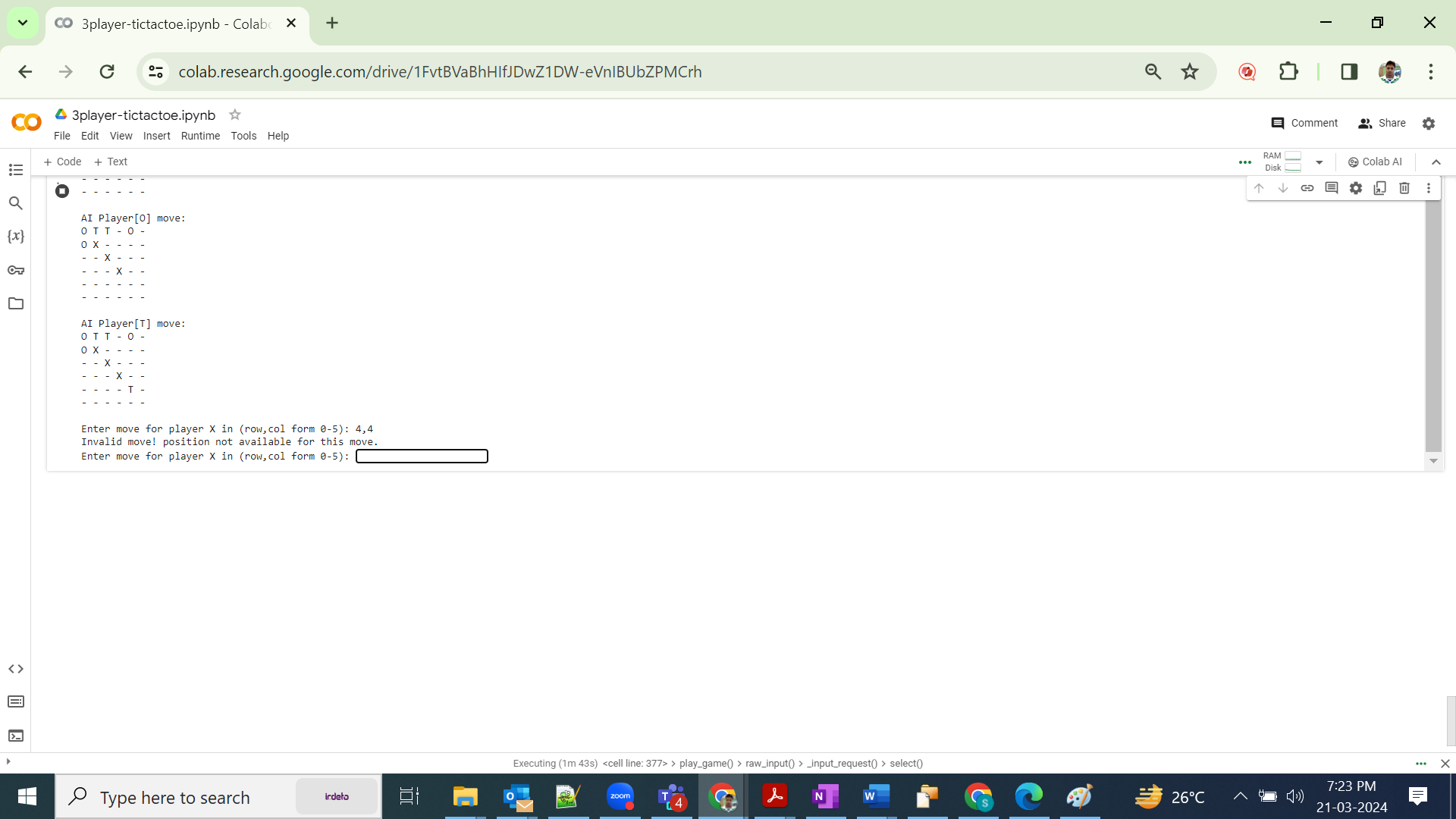
## Move2: X=(2,2)



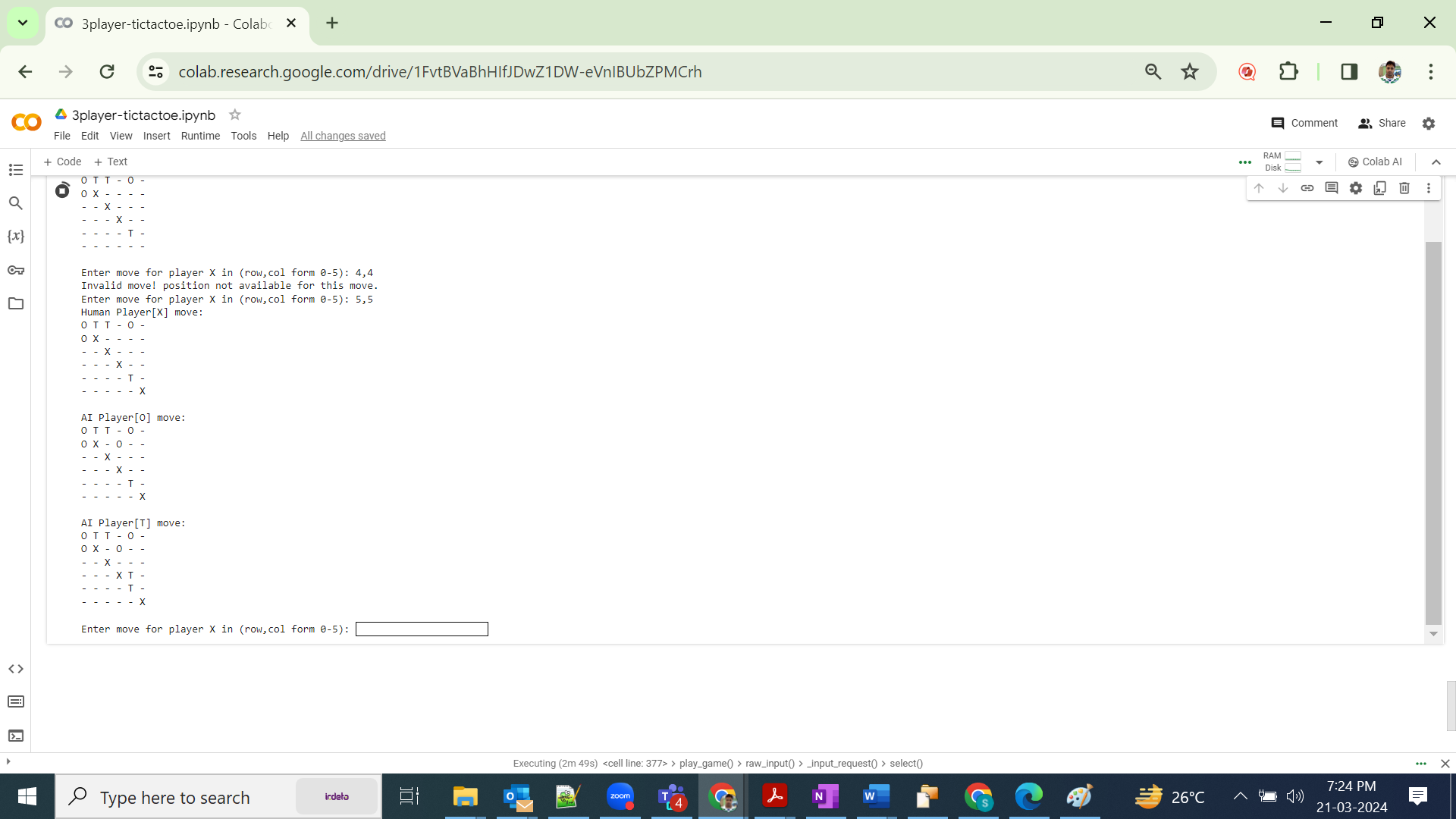
## Move3: X=(3,3)



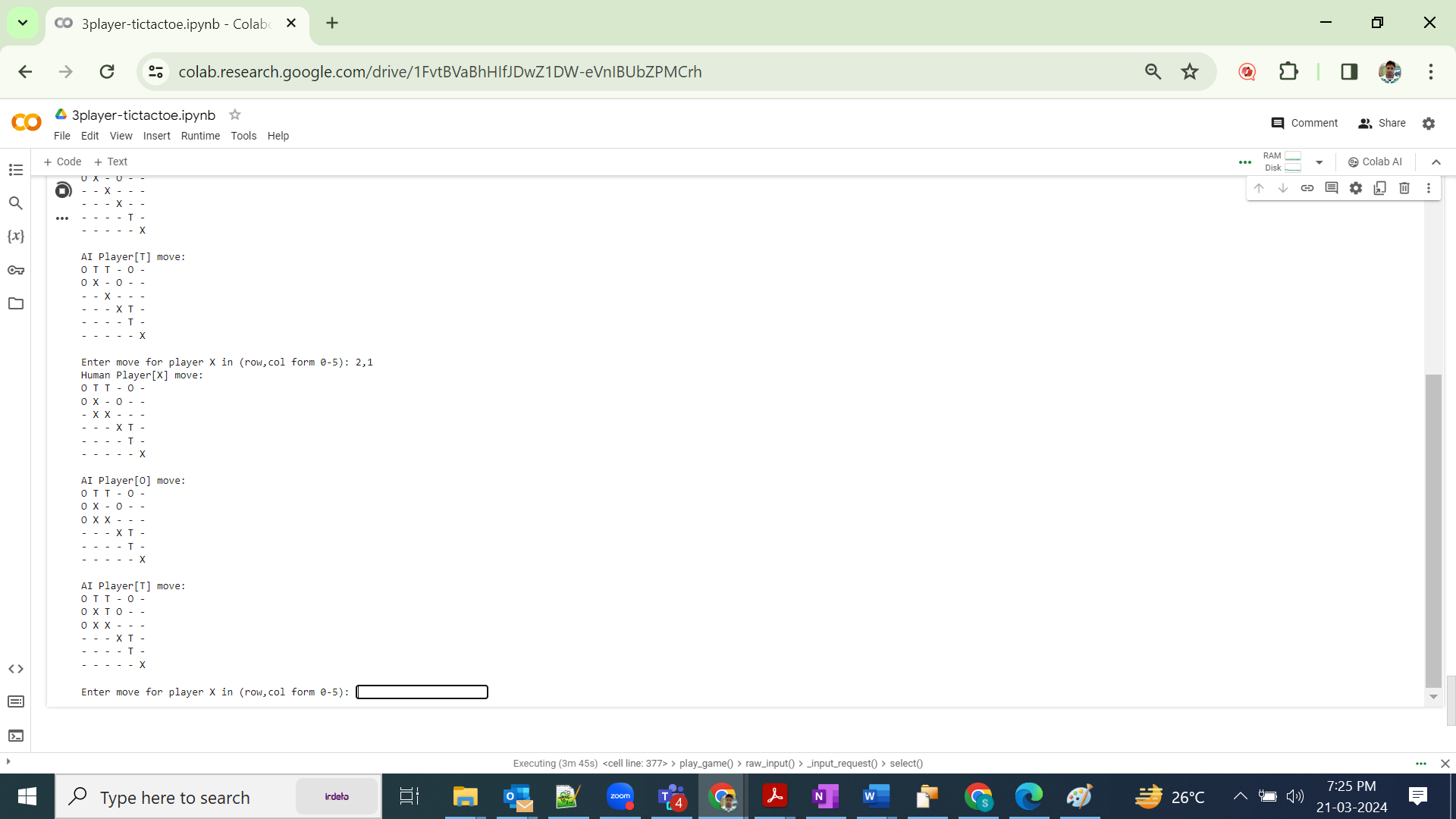
## Move4: X=(4,4) (invalid cell)



## Move5: X=(5,5)

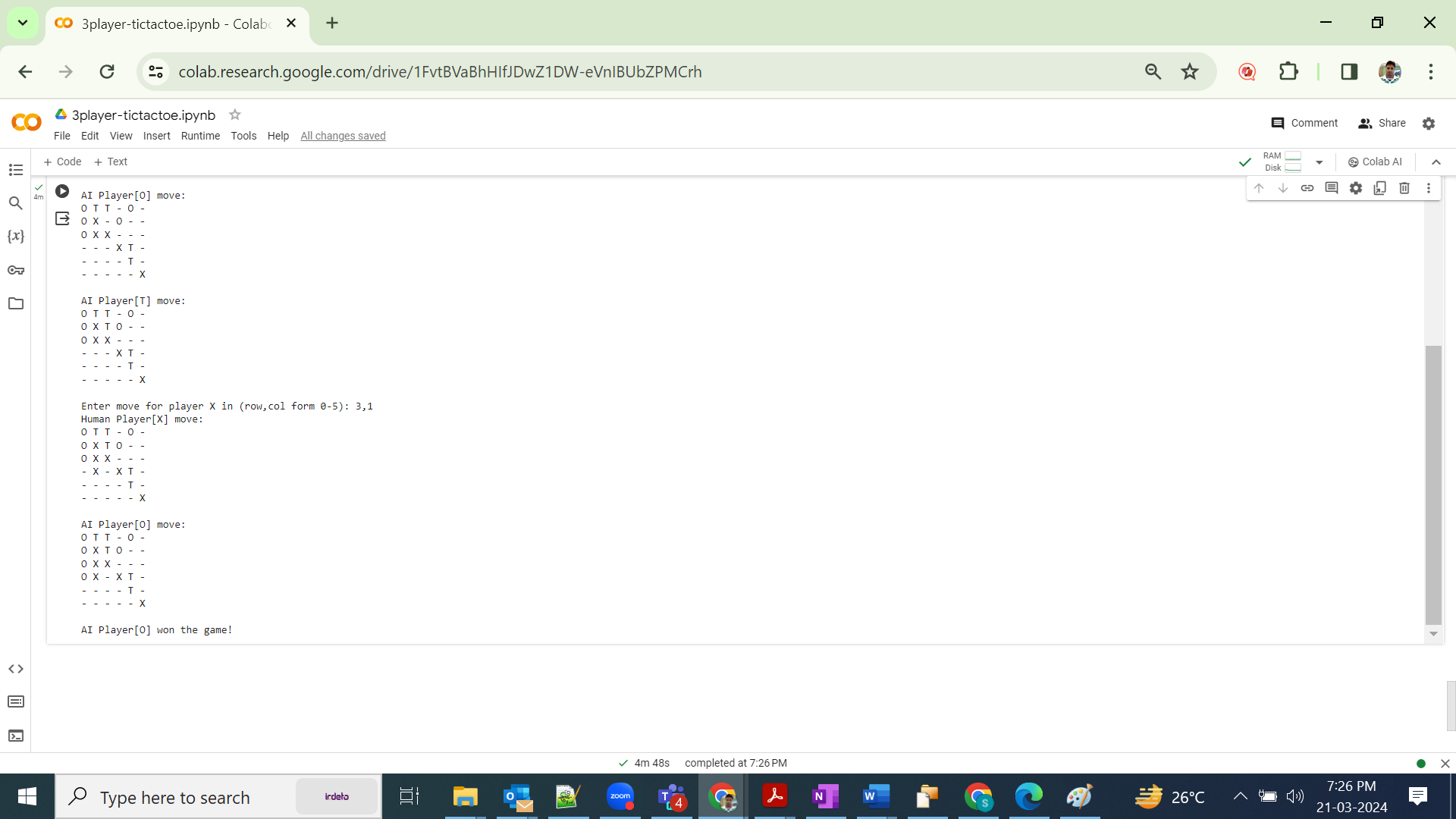


## Move6: X=(2,1)



## Move7: X=(3,1) (player O wins)

AI Player O won the game!



# **Problem Statement 2. (Prolog Logic)**

We have been gathering vital data from engineering students as part of a thorough investigation of the effect of 40-100 on people's success. We ask the following extremely objective questions in a totally voluntary survey that all students are obliged to complete:

● Do you attend parties? [party: yes/no]

● Are you smart? [smart: yes/no]

● Are you innovative? [innovative: yes/no]

● Will you be up-to-date to finish your assignments? [assignment: yes/no]

● Do you have a laptop? [laptop: yes/no]

● Does your project succeed between 40-100? [project: yes/no]

● Have you derived your project goal successfully? [goal: yes/no]

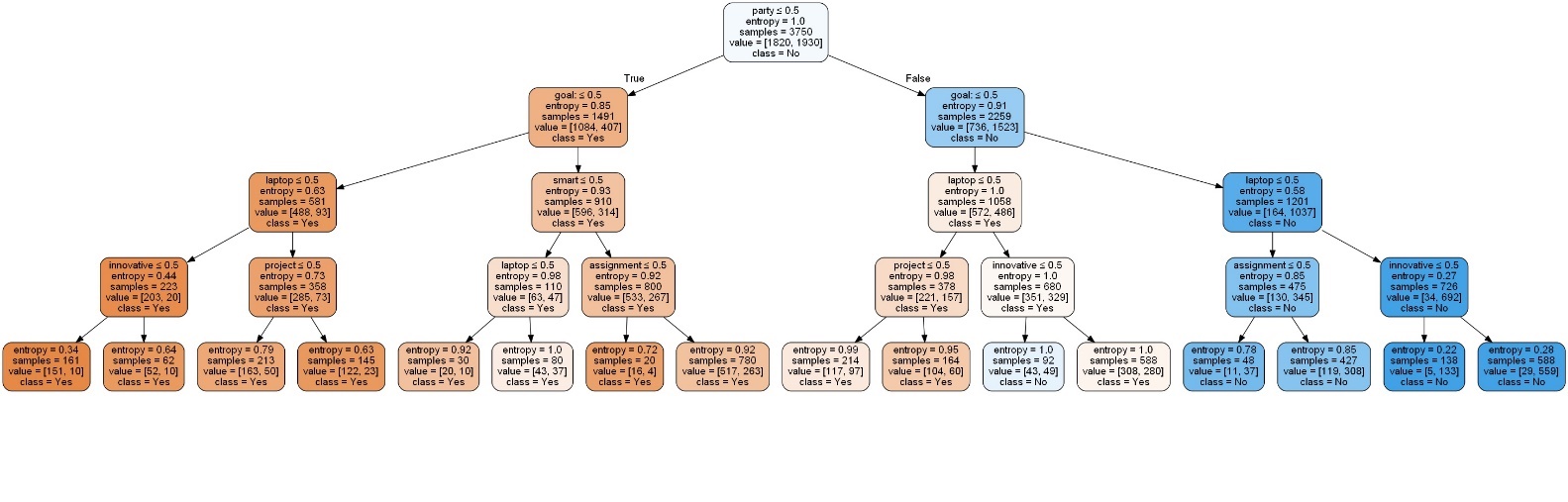
● Do you feel happy now? [happy: yes/no]

You can get the comma-separated survey results from following link.

<https://drive.google.com/file/d/1qHqCFLfBt1EZ7BQO7h904WqHp3iCskkS/view?usp=sharing>

## Given Decision Tree.

HappyOrNotDT.jpg



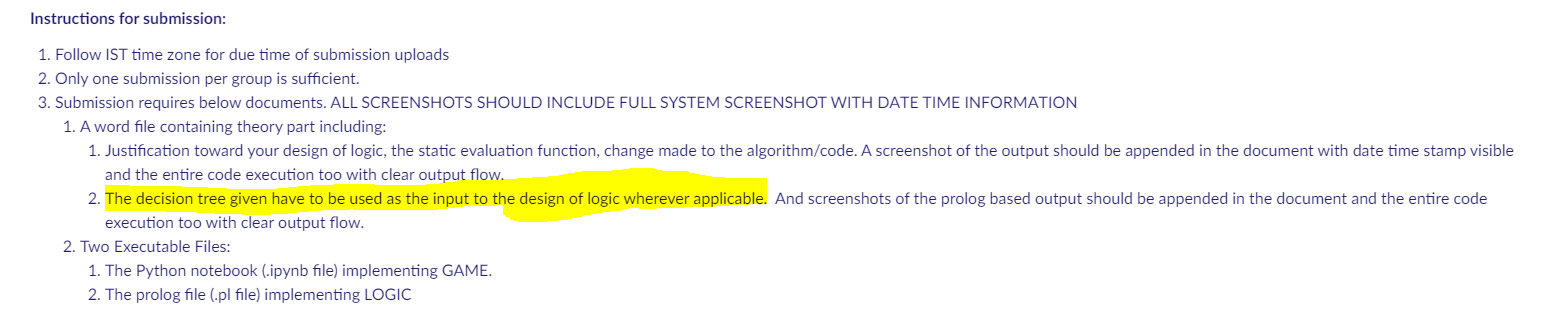
## Note: To use DT as input

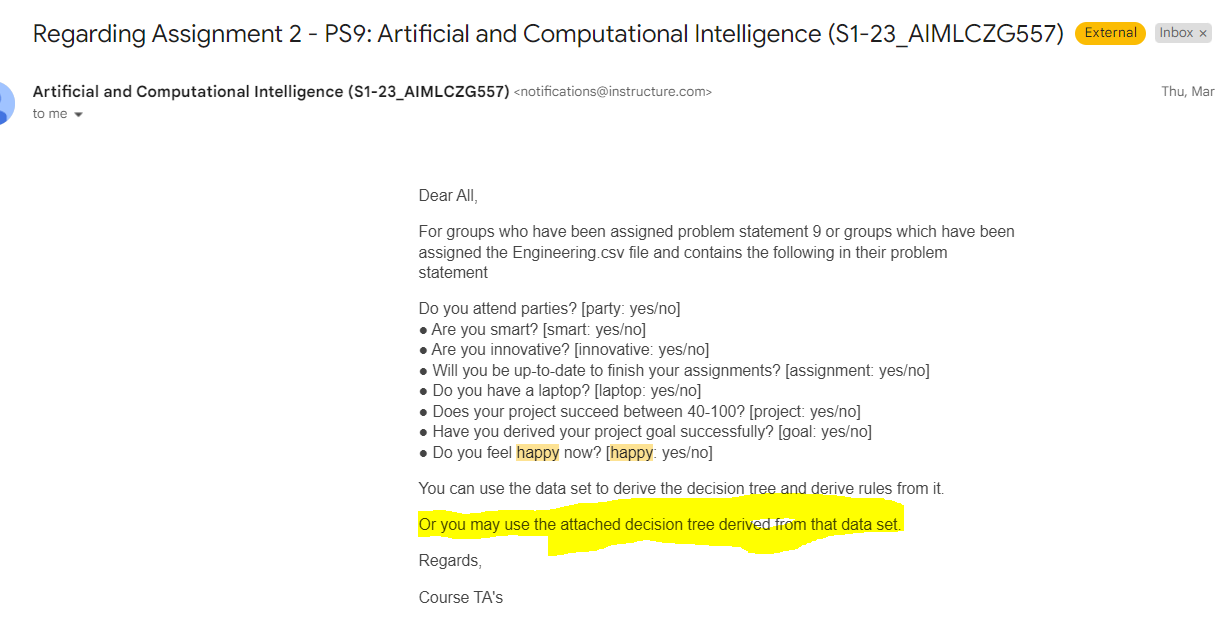
As this has been explicitly asked in email as well as mentioned in assignment instruction page, that we can either make use of the given datasheet to generate decision tree and then derive the prolog rules from it

OR we can use the shared decision tree and derive prolog logic rules out of it.

We are using 2nd approach to derive prolog logic rules from the given decision tree. We will verify our results later by cross checking data in given datasheet.

**Supporting documents:**



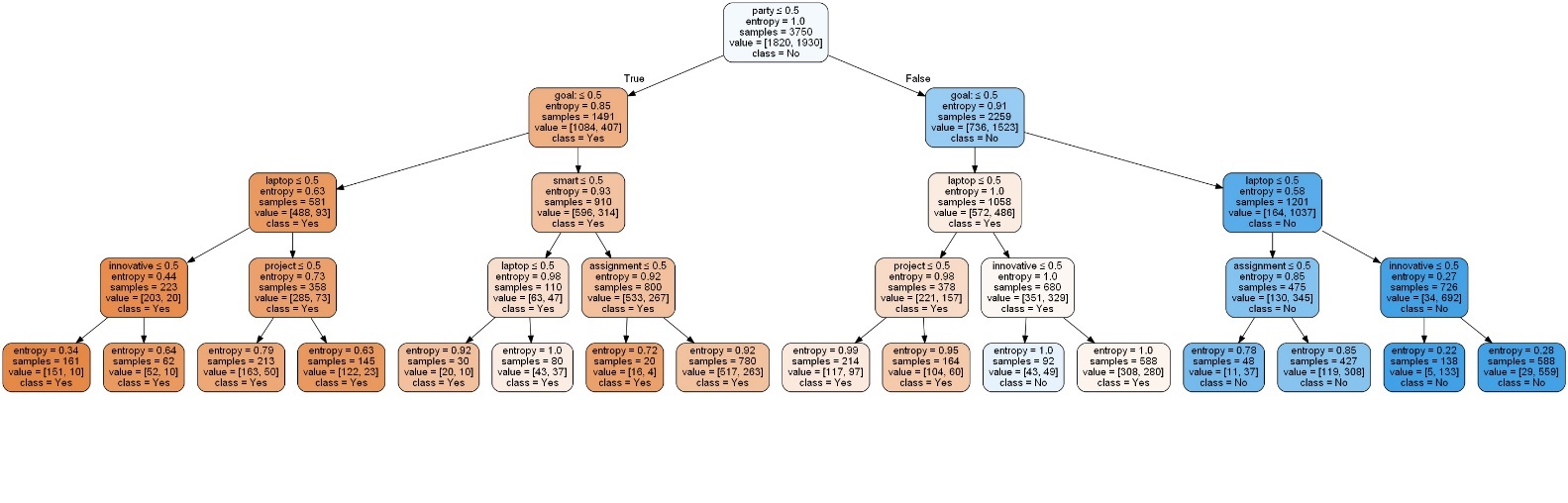


## Note: To use “happiness” as class variable:

In assignment instructions, its mentioned that we are predicting “the effect of 40-100 on people's success” however in the given decision tree, happiness is taken as class variable. We assume happiness as class variable too.

# **Logic Design and Inferencing in Prolog**

We derive below manual rules from the given Decision Tree.

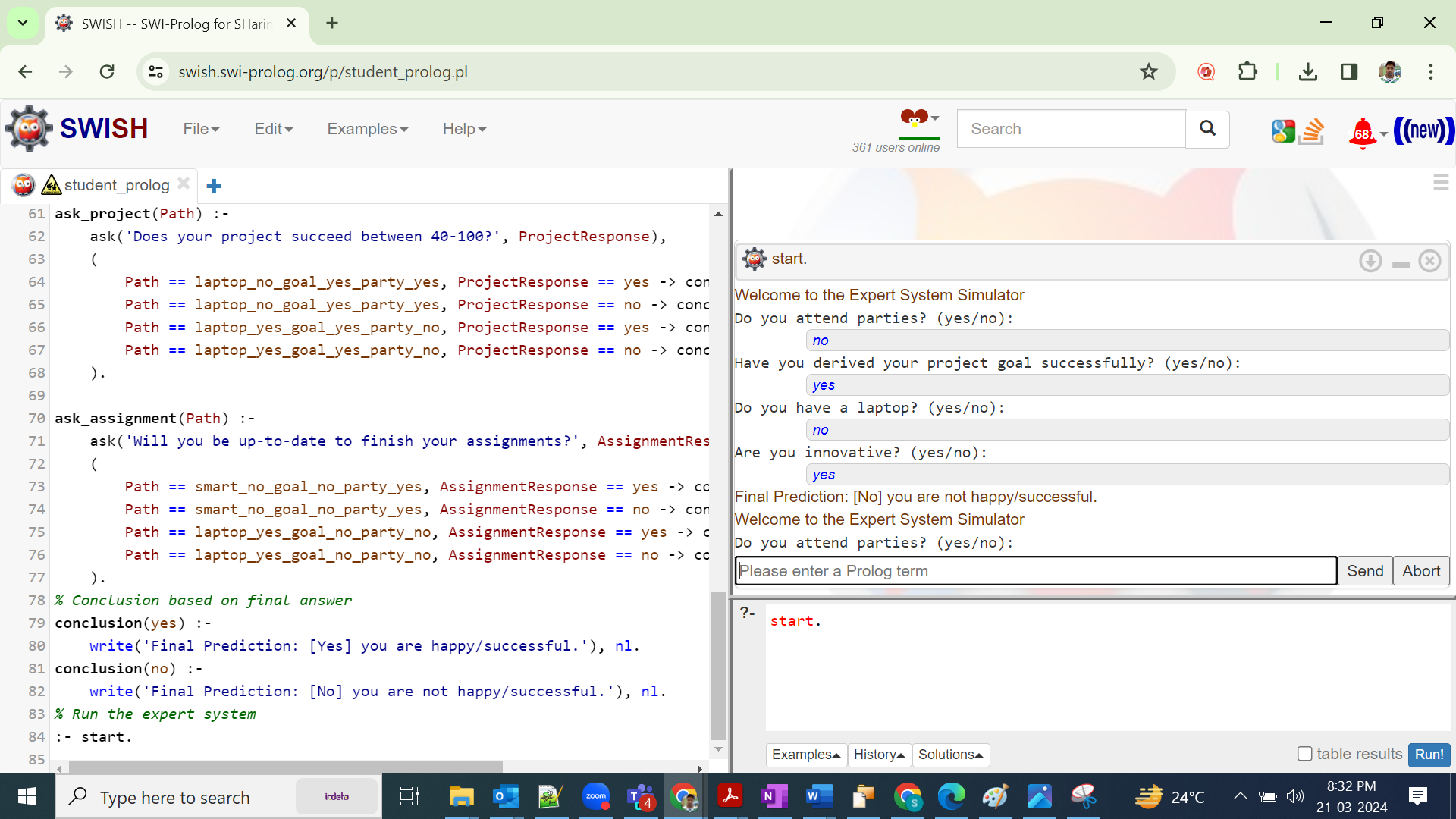


**Manually coded rules:**

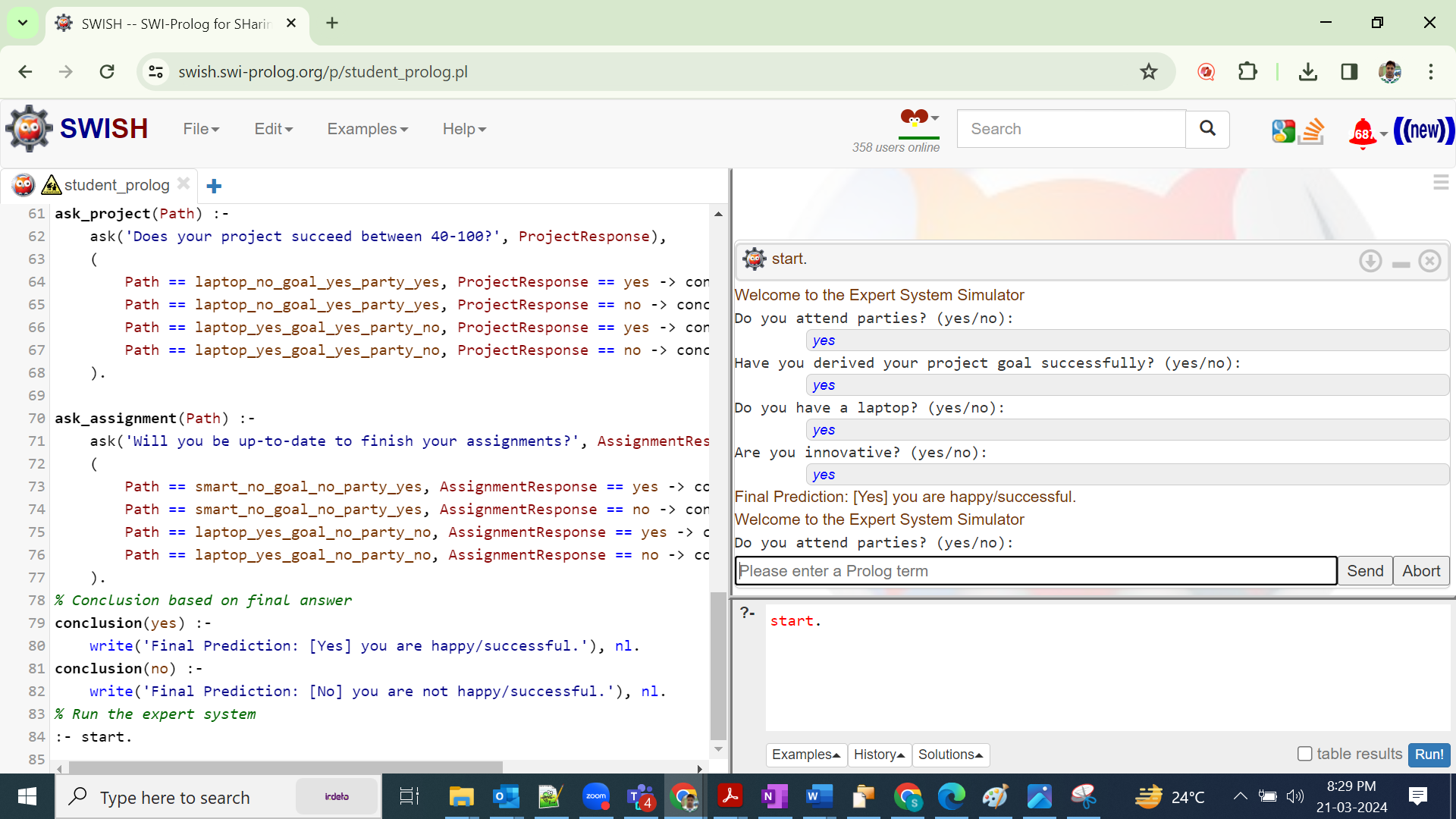
|  |
| --- |
| IF party=yes AND goal =yes AND laptop =yes AND innovative =yes THEN happy |
| IF party=yes AND goal =yes AND laptop =yes AND innovative =no THEN happy |
| IF party=yes AND goal =yes AND laptop =no AND project =yes THEN happy |
| IF party=yes AND goal =yes AND laptop =no AND project =no THEN happy |
| IF party=yes AND goal =no AND smart =yes AND laptop =yes THEN happy |
| IF party=yes AND goal =no AND smart =yes AND laptop =no THEN happy |
| IF party=yes AND goal =no AND smart =no AND assignment =yes THEN happy |
| IF party=yes AND goal =no AND smart =no AND assignment =no THEN happy |
| IF party=no AND goal =yes AND laptop =yes AND project =yes THEN happy |
| IF party=no AND goal =yes AND laptop =yes AND project =no THEN happy |
| IF party=no AND goal =yes AND laptop =no AND innovative =yes THEN not-happy |
| IF party=no AND goal =yes AND laptop =no AND innovative =no THEN happy |
| IF party=no AND goal =no AND laptop =yes AND assignment =yes THEN not-happy |
| IF party=no AND goal =no AND laptop =yes AND assignment =no THEN not-happy |
| IF party=no AND goal =no AND laptop =no AND innovative =yes THEN not-happy |
| IF party=no AND goal =no AND laptop =no AND innovative =no THEN not-happy |

# **Snapshot of output from prolog logic program:**

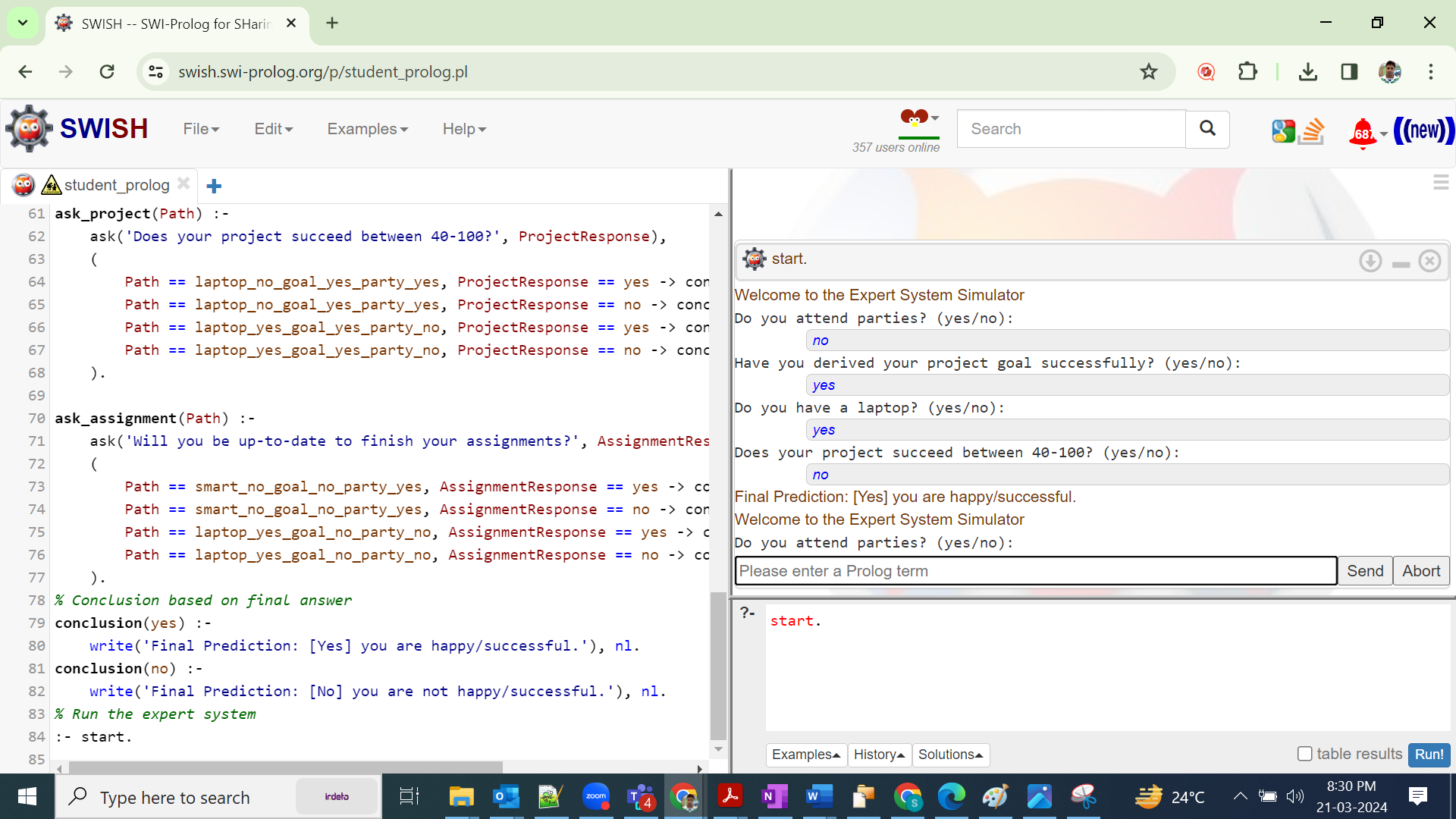
## RUN:0



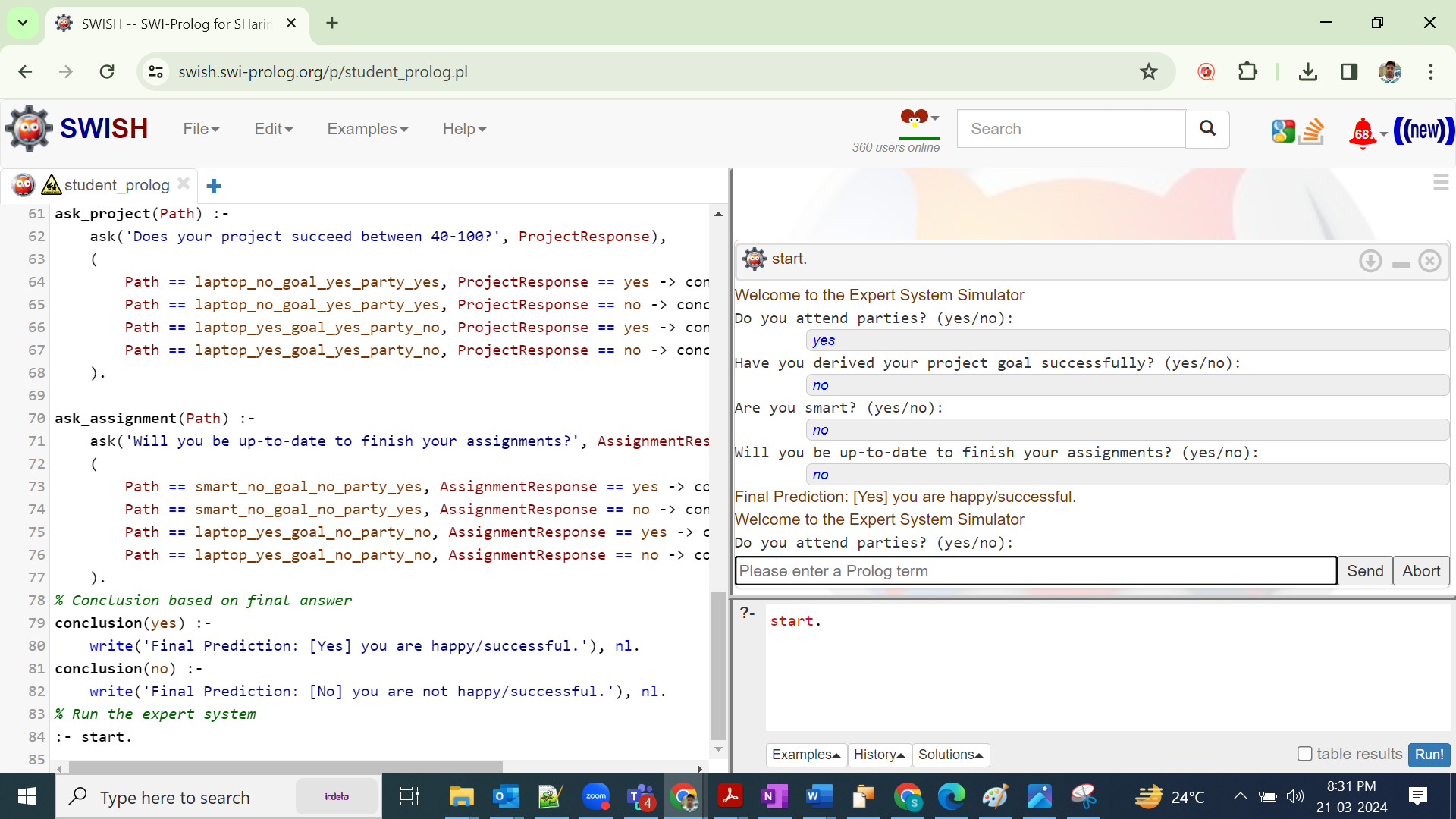
## RUN:1



## RUN:2



## RUN:3



## RUN:4 (invalid response)

