Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Giuroiu Tudor/Bălaș Andrei
Group: 30434
Email: tudorgiu@gmail.com/balas.andrei32@gmail.com

Teaching Assistant: Alexandru Lecu
lecu.alex@yahoo.com

Contents

1 Introduction

1.1 Objective

This project is centered around the proficient utilization of Prover9 or Mace4 to illustrate the practical applications of first-order logic in problem-solving. It simulates two stages of the "Cruce" romanian-hungarian game, which have the logic calculated via Mace4.

1.2 Context

This project employs the Mace4 application to handle the back-end logic of the "Cruce" game simulation. Mace4, a crucial component of the project's toolkit, serves as a specialized tool for processing the intricacies of the game's logic.

Mace4, part of the TPTP (Thousands of Problems for Theorem Provers) library, is adept at generating models that satisfy given sets of conditions in first-order logic.

The project simulates just two parts of the "Cruce" card game. The first part is the card placement section of the game, where the player is allowed to put down a card in relation to the already placed cards. In "Cruce", the first player will play the first card face up, and then, in clockwise order, each player must respond with a card of the same suit. A player who does not have any cards of the same suit must play a trump card. If the player has no trump cards, only then can they play any card of their choice. In order to not complicate things and focus on the logic behind the game, the first 3 players' cards will be already put down. This means that the user will be able to play as the 4th player, placing down the last card, depending on the previous 3 cards.

The second implemented part is the round winning decision-making. In "Cruce", after each player has played a card, the cards will be taken by the holder of the highest trump card (or the only trump card) played, or, if no trump card has been played, the cards will be taken by the player who played the highest card of the required suit (the suit of the first card) or the first card if no card was higher. Cards of other suits will not be higher, regardless of their value. Once the 4 cards are placed down, the application will determine which card is leading, and, together with it, the winner of the round.

Because this project is aimed to show how can Mace4 use first-order logic to determine the output of the game using some rules, the following aspects of the game "Cruce" will be omitted: announcement, bidding, 4 player gameplay, scoring, and match ending.

You can read further rules, in Romanian, here.

2 Implementation

2.1 Mace4

Mace4 needs an input file, with constants and predicates, in order to be able to generate an output. We have 2 files, one for the card placement part of the game (**cruce.in**), and the other one for the round winning part (**winning_card.in**).

The Mace4 input file will be interpreted using the command:

```
Shell
~$ mace4 -c -f <input_file> | interpformat portable > <output_file>
```

-c option asks Mace4 to ignore Prover9 options it does not understand. Here it is used as a safety net, in case any Prover9 syntax gets into the input file (although it should not happen). **-f** option specifies the input file for Mace4, so it is necessary at all times. **interpformat** is a command used to format the output in different ways. The **portable** parameter formats the output file in json format, making it very easy to read it with python.

2.1.1 cruce.in

To ensure that Mace4 accurately identifies each card within the "Cruce" game, encompassing a total of 24 cards, it is necessary to specify that the cards are distinct and discernible from one another. This can be obtained by using the **different_from(card1, card2).** predicate, in combination with the predicate **right_neighbour(card1, card2).**. These two predicates will determine Mace4 to distinct between different cards and to achieve a domain size of 24 (the same as the number of cards in the game).

different_from(card1, card2). predicate is used in the input file between every 2 cards. There is also the general predicate **different_from(x,y) ->different_from(y,x).**, which helps achieve commutativity.

right_neighbour(card1, card2). predicate is used between 2 consecutive cards, so there will be 23 such predicates written in the input file. It is also necessary to negate all the other possibilities. This way, the cards are ordered in a certain manner, helping Mace4 distinguish between them.

Next, the game state is defined. There is the predicate **tromf(card).** which marks the trump card. Then there are the suit predicates, **ghinda(card).**, **rosu(card).**, **bata(card).**, and **frunza(card).**. These predicates tell Mace4 what suit a certain card has, in order to be able later to formulate the rules which determine the placeable cards from the user's hand. There is also need for the **your_hand(card).** predicate, which informs Mace4 of the user's hand.

Based on the properties of the cards presented above the rules of the game will be implemented. The card that dictates which card can be put down or not is the first card, so we defined the predicate **is_first(card_name).**. Based on this card we will need to check our hand and to see if we have any card of the suit of the first placed card. We went through all suit cases, if first card is ghinda, bata, rosu, frunza(verde), and we marked any card of the first's

card type as **must_place(card).** . Then if the card is a must place and is in our hand we marked it as **unlocked(card).** meaning we have to place it to be a legal move.

```
is_first(x) & ghinda(x) & ghinda(y) -> must_place(y).
is_first(x) & bata(x) & bata(y) -> must_place(y).
is_first(x) & rosu(x) & rosu(y) -> must_place(y).
is_first(x) & frunza(x) & frunza(y) -> must_place(y).
must_place(x) & your_hand(x) -> unlocked(x).
```

The next case begins if we don't have any type of the first card in the hand what we do. We create the predicate **color_in_hand.** that is true if we have any card of the first card's suit in hand and false if not. If it is true the next sentences will not be executed. If the predicate is false we will have to place a trump card if we have it in hand. In the check to see if we need to place a trump card we negate the unlocked card to mark it as true if it is false and then we check if it is a trump is in our hand and if the color_in_hand negated is true only then we unlock the trump card as it will need to be put down.

```
must_place(x) & your_hand(x) -> color_in_hand.
-unlocked(x) & tromf(x) & your_hand(x) & -color_in_hand -> unlocked(x).
```

But what happens if we don't have any trump card? Easy answer, we can put any card we want from the deck. In the same manner as we did above we check if we had any trump in hand. If not we need to mark any card that is in our hand as unlocked.

```
your_hand(x) & tromf(x) -> tromf_in_hand.
-tromf_in_hand & -color_in_hand & -unlocked(x) & your_hand(x) -> unlocked(x).
```

2.1.2 winning_card.in

The beginning of the **winning_card.in** input file is similar to the beginning of the **cruce.in** file. Both of them contain the same **different_from(card1, card2).** and **right_neighbour(card1, card2).** predicates, again, to inform Mace4 that there are 24 different cards (constants).

The first change appears when talking about the **is_bigger(card1,card2).** predicate. Is serves as a comparison between 2 cards of the same suit. This predicate is transitive, and odd. That is why these lines need to be added:

```
is_bigger(x,y) & is_bigger(y,z) -> is_bigger(x,z).
is_bigger(x,y) -> -is_bigger(y,x).
```

The **same_color(card1, card2).** predicate is an essential one. In this stage of the game, there is no need to know what actual suits certain cards are. It is enough to know whether they are or not of the same suit. Between two cards that don't have the same suit, all the negations must be written in the input file.

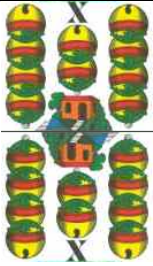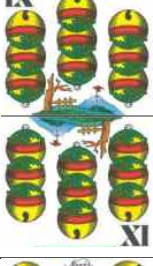Next comes the rules of which card is the leading one. These are:

```
played_first(x) -> played(x).
played(x) & played(y) & tromf(x) & tromf(y) & is_bigger(x,y) -> nu_duce(y).
played(x) & played(y) & tromf(x) & -tromf(y) -> nu_duce(y).
played_first(x) & played(y) & -same_color(x,y) & -tromf(y) -> nu_duce(y).
played(x) & played(y) & same_color(x,y) & is_bigger(x,y) -> nu_duce(y).
```
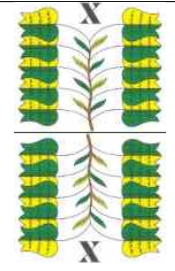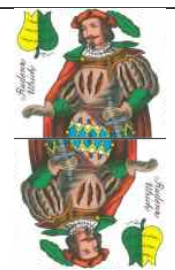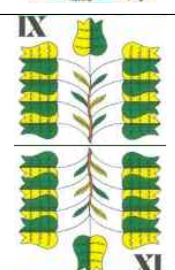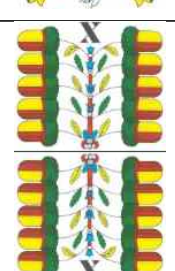
nu_duce(card). predicate marks the argument card as not leading. This will be used by the project application to determine which card is the leading one, excluding, from all placed down cards, the ones that have **nu_duce(card).** true.
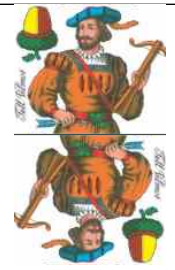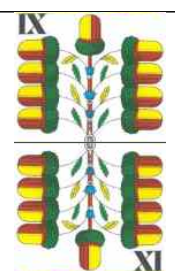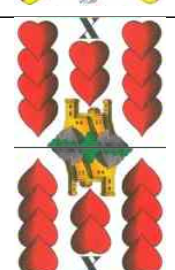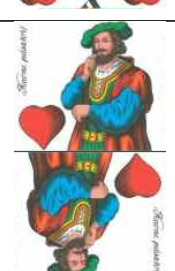
There is need for **tromf(card).** predicate, to mark the trump cards, and **-tromf(card).** negated predicate, to mark all the non-trump cards. There are also the **played(card).** and the **played_first(card).** predicates, which signal to Mace4 which are placed down cards.
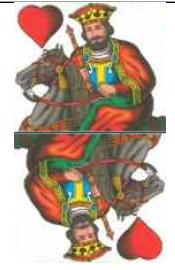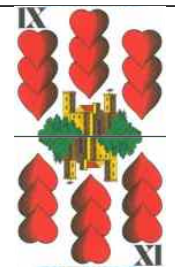
2.2 Python

In the context of this project, cards are implemented as strings.

String	Card
bc10	
bc2	
bc3	
bc4	
bc9	
bca	

fc10	
fc2	
fc3	
fc4	
fc9	
fca	
gc10	

gc2	
gc3	
gc4	
gc9	
gca	
rc10	
rc2	

rc3	
rc4	
rc9	
rca	

The main role of the python in this project is to create an user friendly environment in which the game the we implemented in mace4 can be played. To create this interface we had interpret the output of the mace4 and to change the input files as the game is being played.

Also python is the technology that parses the Mace4 output files, which enables us to use Mace4 in our project.

The user interface is built using pygame and some images. Here is the general look of the project's UI:
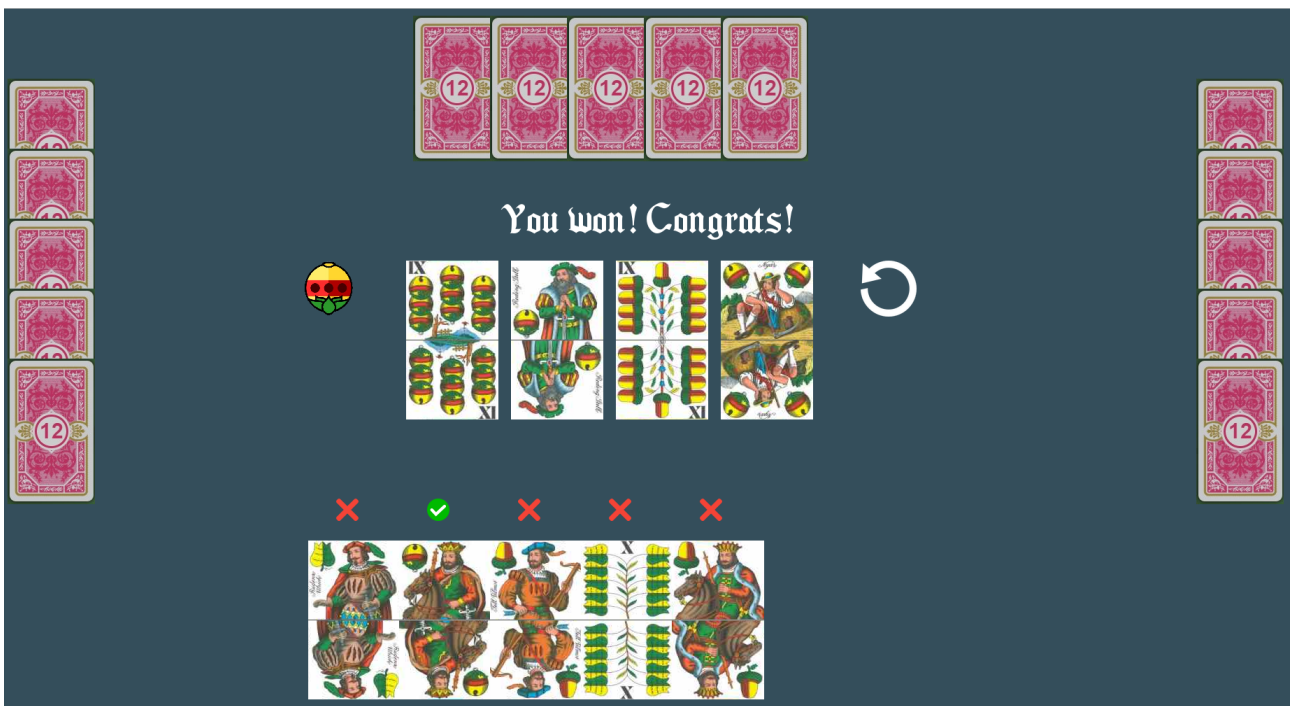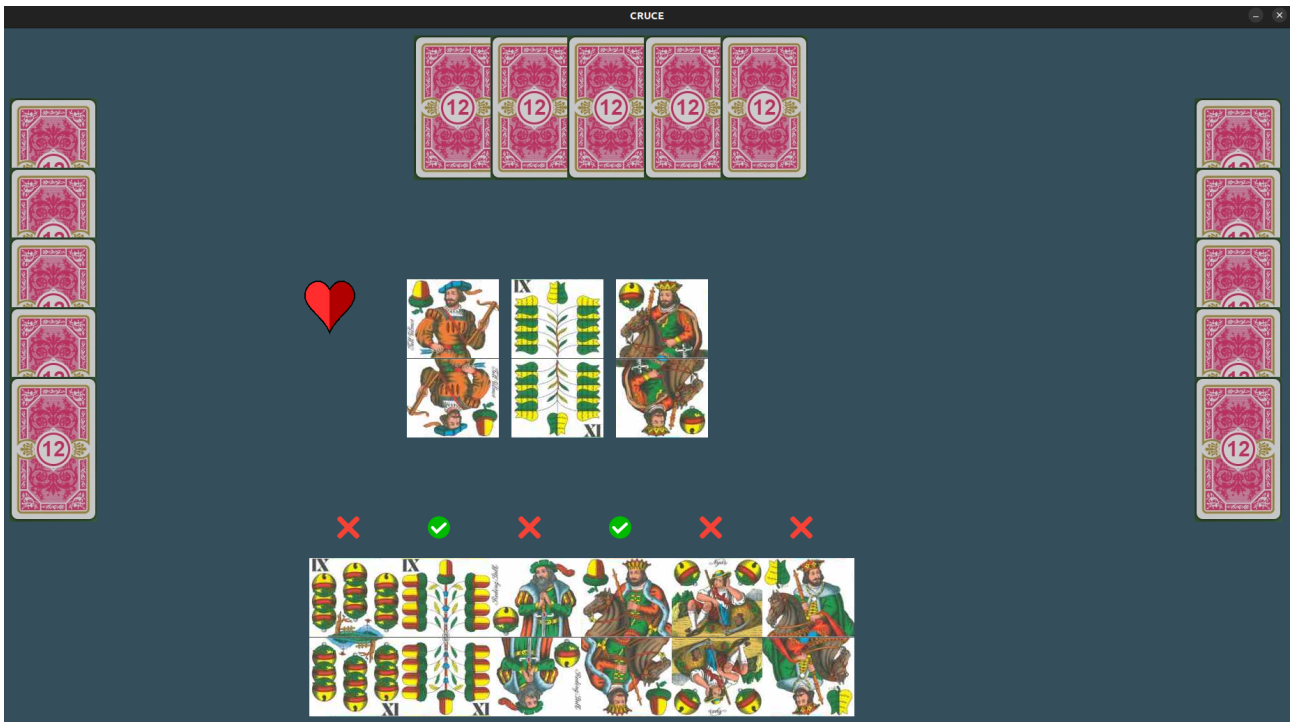
2.2.1   cruce_file_operations.py

The output file that is generated by the mace4 command it's based on arrays of 0 and 1 which means either true or false. The first thing that matters to us in the file is the list of the cards and the index of each card in the arrays.

```
function(bc10, [0]),
function(bc2, [2]),
function(bc3, [3]),
function(bc4, [1]),
function(bc9, [4]),
function(bca, [5]),
```

```
function(fc10, [6]),
function(fc2, [8]),
function(fc3, [9]),
function(fc4, [7]),
function(fc9, [10]),
function(fca, [11]),
function(gc10, [12]),
function(gc2, [14]),
function(gc3, [15]),
function(gc4, [13]),
function(gc9, [16]),
function(gca, [17]),
function(rc10, [18]),
function(rc2, [20]),
function(rc3, [21]),
function(rc4, [19]),
function(rc9, [22]),
function(rca, [23]),
```

Based on this we created a list in python that has the same structure.

```python
CARDSFULL = ["bc10", "bc4", "bc2", "bc3", "bc9", "bca", "fc10", "fc4",  "fc2",
    "fc3", "fc9", "fca", "gc10", "gc4", "gc2",
        "gc3", "gc9", "gca", "rc10", "rc4", "rc2", "rc3", "rc9", "rca"]
```

Next thing that we are interested in from the output file is our hand of cards and what card we can put down from our hand.

```
relation(unlocked(_), [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0]),
relation(your_hand(_), [1,0,0,0,1,0,1,1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,0,0]),
```

For example in this output in our hand we have bc10, bc9, fc10, fc4, fc3, rc2 and we can put down only rc2.

```python
def interpret_Output():
to_return = []

commands = ["./Prover9Mace4/bin/mace4 -c -f cruce.in |
    ./Prover9Mace4/bin/interpformat > cruce.out"]
for arg in commands:
    if os.system(arg) != 0:
        print("Failed to execute command " + arg)
        exit(-1)

# Specify the path to your cruce.out file
file_path = 'cruce.out'

# Read the content of the file
with open(file_path, 'r') as file:
    file_content = file.read()
```

```python
    # Define a regular expression pattern to match the desired line
    your_hand = re.compile(r'relation\(your_hand\(_\), \[(.*?)\]\)')
    CARDSFULL_to_place = re.compile(r'relation\(unlocked\(_\), \[(.*?)\]\)')

    # Find all matches in the file content
    matches = your_hand.findall(file_content)
    match_to_place = CARDSFULL_to_place.findall(file_content)


    # Print the result
    if matches:
        content_inside_brackets = matches[0]
        content_inside_brackets_to_place = match_to_place[0]
        # Safely evaluate the content inside brackets as a Python literal
        your_hand_array = ast.literal_eval(content_inside_brackets)
        to_place = ast.literal_eval(content_inside_brackets_to_place)

        for i in range(0, 24):

            if your_hand_array[i] != 0:

                if to_place[i] == 0:

                    to_return.append((CARDSFULL[i], False))
                else:

                    to_return.append((CARDSFULL[i], True))

    return to_return
```

In the python function we run the mace4 command we direct it to the output file and then we read the file. With pattern matching we find the cards that are in our hand and the cards needed to place from our hand if we have any matches we transform our content inside the brackets into a list of 0 and 1. Next we create the list of tuples that contains the cards from our hand and for each card a bool value if we can put that card down this round or not.

To change input file we used this function:

```python
def change_file(CARDSFULL_in_hand, first_card, my_turn, trump):
    filename = 'cruce.in'

    # Read the content of the file
    with open(filename, 'r') as file:
        lines = file.readlines()

    if trump == "bata":
        index = 0
        for i in range(len(lines)):
```

```python
        if i >= 867:
            if 'tromf' in lines[i]:
                lines[i] = f'tromf({CARDSFULL[index]}).\n'
                index += 1
            else:
                break
if trump == "verde":
    index = 6
    for i in range(len(lines)):

        if i >= 867:
            if 'tromf' in lines[i]:
                lines[i] = f'tromf({CARDSFULL[index]}).\n'
                index += 1
            else:
                break
if trump == "ghinda":
    index = 12
    for i in range(len(lines)):

        if i >= 867:
            if 'tromf' in lines[i]:
                lines[i] = f'tromf({CARDSFULL[index]}).\n'
                index += 1
            else:
                break
if trump == "rosu":
    index = 18
    for i in range(len(lines)):

        if i >= 867:
            if 'tromf' in lines[i]:
                lines[i] = f'tromf({CARDSFULL[index]}).\n'
                index += 1
            else:
                break
for i in range(len(lines)):
    if 'is_first' in lines[i] and not my_turn:
        lines[i] = f'is_first({first_card}).\n'
        break
    elif 'is_first' in lines[i] and my_turn:
        lines[i] = f'%is_first({first_card}).\n'
        break

    for i in range(len(lines)):
        if '-i_place' in lines[i] and my_turn:
            lines[i] = 'i_place.\n'
            break
        elif 'i_place' in lines[i] and not my_turn:
            lines[i] = '-i_place.\n'
            break
```

```
    index = 0

    for i in range(len(lines)):
        if i >= 902:
            if index != len(CARDSFULL_in_hand):
                lines[i] = f'your_hand({CARDSFULL_in_hand[index]}).\n'
                index += 1
            elif i > (902 + index - 1) and i < 908:
                if 'your_hand' in lines[i]:
                    lines[i] = ''
    # Write the updated content back to the file
    with open(filename, 'w') as file:
        file.writelines(lines)
```

This function changes the file, marking a new round as well, based on the cards in hand, first card that is put down, and the trump of that round (my_turn variable will be used for the functionalities that could be implemented in the future).

2.2.2 winning_card_file_operations.py

In order to change the input file **winning_card.in**, the project employs regex using to file the lines that need to be changed.

```
def setup_winning_card_input_file(tromf, first_card, second_card, third_card,
    fourth_card):

    # read the input file
    filename = WINNING_CARD_INPUT_FILENAME
    with open(filename, 'r') as file:
        lines = file.readlines()

    cards = CARDS.copy()
    trumps = trump_to_trump_cards(tromf, cards)
    not_trumps = [card for card in cards if card not in trumps]
    played_cards = [second_card, third_card, fourth_card]

    # setup the regexes
    trump_pattern = re.compile(r"^.*tromf\([^)]*\)\..*$")
    first_card_pattern = re.compile("^.*played_first\([^)]*\)\..*$")
    played_card_pattern = re.compile("^played\([^)]*\)\..*$")

    modified_lines = []

    index_trumps = 0
    index_not_trumps = 0
    index_played_cards = 0

    # modify each matched line
    for l in lines:
```

```python
            if trump_pattern.match(l):
                if l[0] == '-':
                    l = "-tromf(" + not_trumps[index_not_trumps] + ").\n"
                    index_not_trumps += 1
                else:
                    l = "tromf(" + trumps[index_trumps] + ").\n"
                    index_trumps += 1
            elif first_card_pattern.match(l):
                l = "played_first(" + first_card + ").\n"
            elif played_card_pattern.match(l):
                l = "played(" + played_cards[index_played_cards] + ").\n"
                index_played_cards += 1
            modified_lines.append(l)

    # write back to the input file the modified content
    with open(filename, 'w') as file:
        file.writelines(modified_lines)
```

For the parsing of the **winning_card.out** file, the project uses regex again. This function returns the winning card in the format presented breviously in the table.

```python
def return_winning_card_from_winning_card_output_file():

    # opent the output file (which is in json format)
    filename = WINNING_CARD_OUTPUT_FILENAME

    with open(filename, 'r') as file:
        output = json.load(file)

    domain_index_of_cards = {}
    duce_list = []
    played_list = []

    # parse easily the json which is saved in variable output
    for elem in output[0][2]:
        if elem[0] == 'function':
            domain_index_of_cards[elem[1]] = elem[3]
        if elem[0] == 'relation':
            if elem[1] == 'nu_duce':
                duce_list = elem[3]
            elif elem[1] == 'played':
                played_list = elem[3]

    winning_card = None
    played_cards = [card for card in CARDS if
        played_list[domain_index_of_cards[card]] == 1]

    # check which of the played cards are leading (which have 0 in the duce_list)
    for card in played_cards:
        if duce_list[domain_index_of_cards[card]] == 0:
```

```
        winning_card = card

    return winning_card
```

3   Usage

There are 2 options for running the project. The first one is from the PycharmIDE, where the IDE will detect the **requirements.txt** file and ask the user to install the python packages. After the installation, the user can run the **game.py** file from the IDE and see the game begin.

Another way of running the project is from the command line. The user needs to be sure he is in the project's home directory. After this step, the user needs to install the necessary requirements using:

Shell
```
~$ pip install -r requirements.txt
```

Next, the user needs to give **game.py** executable permissions. That can be done by using the command:

Shell
```
~$ chmod +x game.py
```

Following, simply run the following command and the game will begin:

Shell
```
~$ ./game.py
```

4   Conclusion

The "Cruce" game simulation project serves as a great demonstration of the robust capabilities present in Mace4 and First-Order logic. The formulated rules and predicates exemplify the flexible nature of this logical language, showcasing its capacity to succinctly represent and solve complex real-world logic problems.

Furthermore, the project highlights the simple processes of output parsing and input configuration within the Mace4 environment, emphasising its ease of use and showing its ability to perform diverse problem-solving scenarios.

5   Bibliography

www.tromf.ro
https://www.cs.unm.edu/ mccune/prover9/manual/2009-11A/
https://www.pygame.org/docs/