Comparison

Python Prolog C++

The languages I chose were Python, Prolog, and C++, following the procedural, logic, and object oriented paradigm accordingly. The procedural and OO paradigm carry many similarities but my solutions make use of the differences and specific paradigmatic features. SWI Prolog is unlike the other two however. Logic programming is an entirely different approach and my solution showcases a professional level implementation.

Comparison A

Python

```python
# take_turn(+player)
# Each player takes a turn and their building is updated
def take_turn(player):
    print("It's player " + player + "'s turn\n")
    print("Press 1 to spread water in your building or press 0 to spread fire on your opponent\n")
    waterOrFire = check_integer("Enter 1 or 0: \n")
    roomNumber = check_integer("Which room do you chose (1-20): \n") # Get the room number index position
    row = 0
    col = 0
    row, col = update_building(roomNumber, row, col)    # Get the row and column to be updated in the building array,
    if player == "1":
        if waterOrFire == 1:
            spread_water(playerOneBuilding, row, col)   # If P1 chose water then spread water in P1's building
        elif waterOrFire == 0:
            spread_fire(playerTwoBuilding, row, col)    # If P1 chose fire then spread fire in P2's building
    elif player == "2":
        if waterOrFire == 1:
            spread_water(playerTwoBuilding, row, col)   # If P2 chose water then spread water in P2's building
        elif waterOrFire == 0:
            spread_fire(playerOneBuilding, row, col)    # If P2 chose fire then spread fire in P1's building
```

C++

```
192     // take_turn()
193     void take_turn(string player) {
194         cout << "It's Player " << player << "'s turn\n";
195         choose_water_or_fire();
196         get_row_col(roomNumber, row, col);
197 
198         if (player == "1") {
199             if (waterOrFire == 1) {
200                 playerOneBuilding.spread_water(row, col);
201             } else if (waterOrFire == 0) {
202                 playerTwoBuilding.spread_fire(row, col);
203             }
204         } else if (player == "2") {
205             if (waterOrFire == 1) {
206                 playerTwoBuilding.spread_water(row, col);
207             } else if (waterOrFire == 0) {
208                 playerOneBuilding.spread_fire(row, col);
209             }
210         }
211     }
```

Prolog

```
204 % take_turn(+Player, +BuildingOne, +BuildingTwo)
205 take_turn(Player, BuildingOne, BuildingTwo) :-
206     format('~nIts player ~ws turn~n', [Player]),
207     % format('BuildingOne before: ~w~n', [BuildingOne]),
208     % format('BuildingTwo before: ~w~n', [BuildingTwo]),
209     format('Input 0 to spread fire and 1 to spead water: ~n'),
210     read(FireOrWater),
211     format('~nChoose a room to spread to: ~n'),
212     read(RoomNumber),
213     update_building(RoomNumber, Row, Col),
214     player_turn(Player, FireOrWater, BuildingOne, BuildingTwo, Row, Col).
215 
```

```
% player_turn(+Player, +FireOrWater, +BuildingOne, +Build
% Player 1 chose to spread fire
player_turn(1, 0, _, BuildingTwo, Row, Col) :-
    spread_fire(BuildingTwo, Row, Col, NewBuildingTwo),
    retract(buildingTwo(BuildingTwo)),
    assert(buildingTwo(NewBuildingTwo)).

% Player 1 chose to spread water
player_turn(1, 1, BuildingOne, _, Row, Col) :-
    spread_water(BuildingOne, Row, Col, NewBuildingOne),
    retract(buildingOne(BuildingOne)),
    assert(buildingOne(NewBuildingOne)).

% Player 2 chose to spread fire
player_turn(2, 0, BuildingOne, _, Row, Col) :-
    spread_fire(BuildingOne, Row, Col, NewBuildingOne),
    retract(buildingOne(BuildingOne)),
    assert(buildingOne(NewBuildingOne)).

% Player 2 chose to spread water
player_turn(2, 1, _, BuildingTwo, Row, Col) :-
    spread_water(BuildingTwo, Row, Col, NewBuildingTwo),
    retract(buildingTwo(BuildingTwo)),
    assert(buildingTwo(NewBuildingTwo)).
```

Similarities

Python and C++ hold the biggest similarities. The use of if else statements in their own way of formatting. All three solutions call their respective ‘spread’ methods. The different languages IO systems are also present with ‘print’, ‘cout’, and ‘format’.

Differences

As noted in the similarities, all three call ‘spread’ methods but they all operate slightly differently. Python simply calls the method with. All three pass in variables but Python’s variables are untyped, C++’s are earlier in the code, and Prolog even returns an untyped variable back. Going back to the if else statements, my Prolog solution operates completely differently. Here I have four predicates of the same name, each with the options for which player, ability, and the building to update. Instead of checking who and what like in Python

and C++, Prolog simply has predicates for different rules. This is also known as pattern matching, a concept the other languages cannot implement. The C++ version is written within a class, an OO only methodology.

Comparison B

Python

```python
# check_player_one_win(-playerOneWins)
# If player one's building has no fire (all 1's) or player two's building is all fire (all 0's) then player one wins
def check_player_one_win(playerOneWins):
    if playerOneBuilding == [[ 1, 1, 1, 1, 1],
                             [ 1, 1, 1, 1, 1],
                             [ 1, 1, 1, 1, 1],
                             [ 1, 1, 1, 1, 1]] or playerTwoBuilding == [[ 0, 0, 0, 0, 0],
                                                                        [ 0, 0, 0, 0, 0],
                                                                        [ 0, 0, 0, 0, 0],
                                                                        [ 0, 0, 0, 0, 0]]:
        playerOneWins = True
        return playerOneWins

# check_player_two_win(-playerTwoWins)
# If player two's building has no fire (all 1's) or player one's building is all fire (all 0's) then player two wins
def check_player_two_win(playerTwoWins):
    if playerTwoBuilding == [[ 1, 1, 1, 1, 1],
                             [ 1, 1, 1, 1, 1],
                             [ 1, 1, 1, 1, 1],
                             [ 1, 1, 1, 1, 1]] or playerOneBuilding == [[ 0, 0, 0, 0, 0],
                                                                        [ 0, 0, 0, 0, 0],
                                                                        [ 0, 0, 0, 0, 0],
                                                                        [ 0, 0, 0, 0, 0]]:
        playerTwoWins = True
        return playerTwoWins
```

C++

```cpp
    // check_player_win(+building)
    // Pass in a player's building and see if their building is all 1s or all 0s
    bool check_player_win(Building playingPlayerBuidling, Building opponentPlayerBuidling) {
        bool allOnes = true;
        bool allZeros = true;
        for(int i=0; i<4; i++) {
            for(int j=0; j<5; j++) {
                if(playingPlayerBuidling.building[i][j] != 1) {
                    allOnes = false;
                }
                if(opponentPlayerBuidling.building[i][j] != 0) {
                    allZeros = false;
                }
            }
        }
        return allOnes || allZeros;
    }
```

Prolog

```
% check_player_one_win(+PlayerOneBuilding, +PlayerTwoBuilding)
% Check if player one has won by extinguishing their own buildin
check_player_one_win(PlayerOneBuilding, _PlayerTwoBuilding) :-
    building_extinguished(PlayerOneBuilding).

% Check if player one has won by burning down their opponents bu
check_player_one_win(_PlayerOneBuilding, PlayerTwoBuilding) :-
    building_in_flames(PlayerTwoBuilding).

% check_player_two_win(+PlayerOneBuilding, +PlayerTwoBuilding)
% Check if player two has won by extinguishing their own buildin
check_player_two_win(_PlayerOneBuilding, PlayerTwoBuilding) :-
    building_extinguished(PlayerTwoBuilding).

% Check if player two has won by burning down their opponents bu
check_player_two_win(PlayerOneBuilding, _PlayerTwoBuilding) :-
    building_in_flames(PlayerOneBuilding).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%% Building

% is_room_water(+Room)
% Check if building is all 1's (water)
is_room_water([]).                          % Base case
is_room_water([1 | Rest]) :-                % If the current roo
    is_room_water(Rest).                    % Loop back to check

% building_extinguished(+Building)
% Check if a matrix contains all ones
building_extinguished([]).                  % Base case
building_extinguished([Row | Rest]) :-      % Separate the matri
    is_room_water(Row),                     % Check if the curre
    building_extinguished(Rest).            % Loop back to check

% is_room_fire(+Room)
% Check if building is all 0's (fire)
is_room_fire([]).                           % Same as above but
is_room_fire([0 | Rest]) :-
    is_room_fire(Rest).

% building_in_flames(+Building)
% Check if a matrix contains all ones
building_in_flames([]).
building_in_flames([Row | Rest]) :-
    is_room_fire(Row),
    building_in_flames(Rest).
```

Similarities

While the code for Prolog and C++ looks drastically different, they actually function the same. Loop through a given array and check whether all values are ones or zeros. They both focus on one element at a time via two loops. My Python and Prolog code both have two different predicate/procedures for each player and some repeated code however the solutions are still clean.

Differences

Complexity. While the code I created for Prolog is larger in sheer size, it's complexity is quite simple and easy to read, although I would say the other solutions are more clean. Python's solution is probably the most easy to understand and with Python's coding style, I could simply check if an array consists of all ones or zeros. A method I used for C++ and Prolog could be implemented here however, I saw no point when this solution works just as well and is simple to create. However due to this, the C++ and Prolog version will be quicker to execute than Python. My Prolog implementation is the only one that uses recursion -this is a key concept and benefit of Prolog. Where in C++ I use a for loop, Prolog I recurse.

Comparison C

Python

```
81  # display_building(+buildingArray)
82  # Print the building array with a given
83  def display_building(buildingArray):
84      for row in buildingArray:
85          print(' '.join(map(str, row))) #
86      print("\n")
87
```

C++

```
30      // display_building()
31      void display_building() {
32          for (int i = 0; i < 4; i++) {
33              for (int j = 0; j < 5; j++) {
34                  cout << building[i][j] << " ";
35              }
36              cout << endl;
37          }
38      }
```

Prolog

```
75  display_building([]).
76
77  display_building([BuildingFloor | RemainingFloors]) :-
78      print_row(BuildingFloor),                   % Print th
79      nl,                                         % New line
80      display_building(RemainingFloors).          % Loop bac
81
82  % print_row(+Row)
83  % Print a row of the building recursively by looping throu
84  print_row([]).
85
86  print_row([RoomNumber | RemainingRooms]) :-
87      write(RoomNumber),                          % Print th
88      write(' '),                                 % Print a
89      print_row(RemainingRooms).                  % Loop bac
90
```

Similarities

Three approaches to displaying the room numbers of a building. All accomplish this through the use of loops, row by row.

Differences

My Python solution looks through each row and separates each item in the array with a blank space via the use of the “.join” method. This is the cleanest solution. The C++ version does exactly the same, just this time I had to loop each row, column by column, requiring two

loops. Finally the Prolog- the recursion solution. So three solutions all in slightly different ways.

Comparison D

No screenshots this time, just a short explanation related to variables. In Python, variables are mostly untyped. Meaning we can define a variable of any type and the compiler will infer what we desired. The variable naming scheme only requires the start to be a letter. Prolog also doesn't make use of explicitly typed variables -it uses dynamic typing. Prolog variables require a capital letter as their first character. C++ variables are explicitly typed. The compiler won't infer, it needs to be told e.g. int, string. C++ only asks that the first character is a letter, no other formatting needed.

Conclusion

Through my explanations, you can understand how the three languages I chose differ and compare. While more obvious similarities between Python and C++ are evident, much of Prolog shares common approaches to problems as well. Prolog is clearly quite different and my solution highlights good use of the logic paradigm, contrasted to the procedural implementation of Python and OO with C++. The benefits and differences of OO programming is used throughout my C++ solution but these often share similar traits to Python's procedural methodologies.