# Course Name: Principles of Programming Languages

# Course Code: Cmpe260

# Student: İrem Uğuz

# Student ID: 2015400165

# Project Title: Cmpe260- Project1

# Project Type: Programming Project

# Submission Date: 22.04.2018

# Introduction:

The aim of this project is to create Prolog predicates to do the given tasks in the project definition file. The written predicates are put in predicates.pl file. The program aims to reach the goal user gives using the predicates on the predicates.pl and cl\_base.pl (or any other data base file the user gives). There are four given task on the project definition file. Those tasks are:

1. Implement the predicate allTeams(L,N). L is the list containing all the teams in the database where N is the number of elements in the list.
2. Implement the following predicates:

• wins(T,W,L,N) implies that L involves the teams defeated by team T when we are in week W and N is the number of elements in L.

• losses(T,W,L,N) implies that L involves the teams that defeated team T when we are in week W and N is the number of elements in L.

• draws(T,W,L,N) is very similar but now L involves the teams that team T could not defeat also did not lose to.

1. Implement scored(T,W,S) where S is the total number of goals scored by team T up to (and including) week W. Implement conceded(T,W,C) where C is the total number of goals conceded by team T up to week W. Assume T and W are given as constants.

Implement average(T,W,A) where A is the average (goals scored – goals conceded) of a team T gathered up to (and including) week W. Assume T and W are given as constants.

1. Implement order(L,W) where W (week) is given as constant and league order in that week will be retrieved in L. Additionally, implement topThree([T1,T2, T3],W) where T1 T2 and T3 are the top teams when we are in the given week W. Assume that the order is decided according to average i.e the one with the highest average will be at the top.

# Program Interface:

The program is written on swi-prolog program as instructed on the ps session. User can run the program on swi-prolog as well. To run the program user must go to the directory in which the .pl files are in. Then in terminal, user should invoke swi-prolog by writing “swipl” command on terminal. Then user should install the database on swi-prolog on terminal. To do this user must write the command “[predicates.pl].” on terminal. After the database is successfully installed on swi-prolog, the “true” message appears on screen. After that user can try the predicates by writing them on terminal with necessary and true parameters. This instructions are for Ubuntu’s terminal, it can be different on any other operating system so it is recommended that the program is tried on Ubuntu.

# Program Execution:

After opening the swi-prolog on terminal in the way described above in the section Program Interface, user can execute the predicates and see if they succeed or not. There are nine predicates that is implemented by me for the tasks given in the project description. Those are:

* allTeams(L,N).

User can try this predicate if this succeeds by giving a list L and N as the parameters. And after one result is shown, if user clicks ; on terminal, there should appear the permutations of list L as long as user keeps clicking on ;. If the given parameters to the predicate does not succeed, “false” appears on screen.

* wins(T,W,L,N).

User can try this predicate by giving up to 4 parameters. T is the team name and W is the current week. L is the list of the teams that team T won a match with till week W (is included). The list L is empty if T has never won a game.

* losses(T,W,L,N).

User can try this predicate by giving up to 4 parameters. T is the team name and W is the current week. L is the list of the teams that team T lost a match with till week W (is included). The list L is empty if T has never lost a game.

* draws(T,W,L,N).

User can try this predicate by giving up to 4 parameters. T is the team name and W is the current week. L is the list of the teams that team T scored the same in a match with till week W (is included). The list L is empty if T has never scored the same score with a team before.

* scored(T,W,S).

T is team, W is week and S is the scored goal number by team T till week W. User can try this predicate by giving T(team) and W(week) ad constants. If user doesn’t give S, program tries to find S that succeeds and writes this S on the screen. If user gives an S, then program checks if it can succeed with that S. If it cannot then writes false.

* conceded(T,W,C).

T is team, W is week and C is the conceded goal number by team T till week W. User can try this predicate by giving T(team) and W(week) ad constants. If user doesn’t give C, program tries to find C that succeeds and writes this C on the screen. If user gives a C, then program checks if it can succeed with that C. If it cannot then writes false.

* average(T,W,A).

T is team, W is week and A is the average number that T gained till week W where average is scored goals-conceded goals till week W. If user doesn’t give A, program tries to find A that succeeds and writes this A on the screen. If user gives an A, then program checks if it can succeed with that A. If it cannot then writes false.

* order(L,W).

User can try this predicate by giving W as constant. If user gives L, then program checks if given L succeeds. If user doesn’t give L, then program tries to find an L that succeeds.

* topThree([T1,T2, T3],W).

User can try this predicate by giving W as constant. If [T1,T2, T3] is given by user, program checks if those values succeeds, if they are not given program finds values that succeeds.

# Input and Output:

The program takes its inputs from terminal and gives its outputs on terminal as well. As explained above in Program Execution section, user can give predicates as inputs. The outputs of the program will be according to these inputs. If user gave some input to find the values of variables, output will be these values if they exist, and false if they don’t exist. If user gave inputs to check if the predicate succeeds then outputs will be true or false.

# Program Structure:

**allTeams(L,N) :- findall(X,team(X,\_),Teamlist), permutation(L,Teamlist),length(Teamlist,N).**

Here in this predicate findall built-in predicate is used to find all the predicates that succeeds team predicate and listed teams in Teamlist variable. Then with the permutation built-in predicate, finds the various versions of L. N is found with the built-in predicate of length.

**winMatch(T,Y,W) :- match(W, T, W1,Y,W2), W1>W2.**

**winMatch(T,Y,W) :- match(W, Y, W1, T,W2), W1<W2.**

winMatch predicate implies that team T had a match with team Y on week W and in this match, T’s score was greater than Y’s score.

**wins(\_,W,\_,\_) :- W<1 , break.**

**wins(Team,1,List,Number) :- findall(Y, winMatch(Team,Y,1), List) , length(List, Number).**

**wins(T,W,L,N) :- W>1 , findall(Y, winMatch(T,Y,W), List1) , W1 is W-1, wins(T,W1, List2, \_) , append(List1, List2, L), length(L,N).**

Those three predicates of wins are recursive predicates. First line is control predicate. When the week is less than one, which is not possible in normal circumstances, the predicate fails. Second line is the stopping predicate for recursion. When W is 1, it stops calling itself. The wins in week 1 is the all wins possible up to this week since weeks start with 1. On the third wins predicate it finds the matches that team T won in week W and calls the wins predicate again for previous week. Then appends the resulting two lists N(the size of list) is found with length predicate.

**lossMatch(T,Y,W) :- match(W, T, L1,Y,L2), L1<L2.**

**lossMatch(T,Y,W) :- match(W, Y, L1, T,L2), L1>L2.**

lossMatch predicates imply that team T had a match with team Y on week W and in this match, T’s score was less than Y’s score.

**losses(\_,W,\_,\_) :- W<1 , break.**

**losses(Team,1,List,Number) :- findall(Y, lossMatch(Team,Y,1), List) , length(List, Number).**

**losses(T,W,L,N) :- W>1 , findall(Y, lossMatch(T,Y,W), List1) , W1 is W-1, losses(T,W1, List2, \_) , append(List1, List2, L), length(L,N).**

Those three predicates of losses are recursive predicates. First line is control predicate. When the week is less than one, which is not possible in normal circumstances, the predicate fails. Second line is the stopping predicate for recursion. When W is 1, it stops calling itself. The losses in week 1 is the all losses possible up to this week since weeks start with 1. On the third losses predicate it finds the matches that team T lost in week W and calls the losses predicate again for previous week. Then appends the resulting two lists N(the size of list) is found with length predicate.

**drawMatch(T,Y,W) :- match(W, T, D1,Y,D2), D1=D2.**

**drawMatch(T,Y,W) :- match(W, Y, D1, T,D2),D1=D2.**

drawMatch predicates imply that team T had a match with team Y on week W and in this match, T’s score equal to Y’s score.

**draws(\_,W,\_,\_) :- W<1 , break.**

**draws(Team,1,List,Number) :- findall(Y, drawMatch(Team,Y,1), List) , length(List, Number).**

**draws(T,W,L,N) :- W>1 , findall(Y, drawMatch(T,Y,W), List1) , W1 is W-1, draws(T,W1, List2, \_) , append(List1, List2, L), length(L,N).**

Those three predicates of draws are recursive predicates. First line is control predicate. When the week is less than one, which is not possible in normal circumstances, the predicate fails. Second line is the stopping predicate for recursion. When W is 1, it stops calling itself. The draws in week 1 is the all draws possible up to this week since weeks start with 1. On the third draws predicate it finds the matches that team T drew in week W and calls the draws predicate again for previous week. Then appends the resulting two lists. N(the size of list) is found with length predicate.

**scored(\_,W,\_) :- W<1 ,break.**

**scored(T,W,S) :- W=1 , findall(X, match(W,T,X,\_,\_), HomeScore), findall(Y,match(W,\_,\_,T,Y),GuestScore), append(HomeScore, GuestScore, Score), sum\_list(Score, S).**

**scored(T,W,S) :- W>1, findall(X, match(W,T,X,\_,\_), HomeScore), findall(Y,match(W,\_,\_,T,Y),GuestScore), append(HomeScore, GuestScore, CurrentScore),sum\_list(CurrentScore, Score), W1 is W-1, scored(T,W1, PrevScore), S is Score+PrevScore .**

scored predicates are working recursively as well. When W is less than 1, predicate returns false with break predicate. And when W is one, this means there is only one week of scores to calculate so the scores of the matched that team T did are found with findall predicate and the resulting list’s sum is found with sum\_list. I used sum\_list because it is not written on project description that one team can only do one match in one week. So I assumed that a team can do a few matches in one week. When week W is more than one, then it calls the previous week’s score predicate and sums it with current week’s score.

**conceded(\_,W,\_) :- W<1, break.**

**conceded(T,W,C) :- W=1 , findall(X, match(W,T,\_,\_,X), HomeConceded), findall(Y,match(W,\_,Y,T,\_),GuestConceded), append(HomeConceded, GuestConceded, Conceded), sum\_list(Conceded, C).**

**conceded(T,W,C) :- W>1, findall(X, match(W,T,\_,\_,X), HomeConceded),findall(Y,match(W,\_,Y,T,\_),GuestConceded), append(HomeConceded, GuestConceded, CurrentConceded),sum\_list(CurrentConceded, Conceded), W1 is W-1, conceded(T,W1, PrevConceded), C is Conceded+PrevConceded .**

conceded predicates work in the same way as scored predicates. They are recursive as well.

**average(\_,W,\_) :- W<1, break.**

**average(T,W,A) :- scored(T,W,S) , conceded(T,W,C), A is S-C.**

average predicste fails when W is less than 1 like previous predicates. When W is not 1, then A is scored predicate’s result minus conceded predicate’s result.

**insert\_sort(List,Sorted,W):-i\_sort(List,[],Sorted,W). i\_sort([],Acc,Acc,\_). i\_sort([H|T],Acc,Sorted,W):-insert(H,Acc,NAcc,W),i\_sort(T,NAcc,Sorted,W). insert(X,[Y|T],[Y|NT],W):- average(X,W,AvX),average(Y,W,AvY),AvX=<AvY, insert(X,T,NT,W). insert(X,[Y|T],[X,Y|T],W):- average(X,W,AvX),average(Y,W,AvY), AvX>AvY. insert(X,[Y|T],[Y|NT],W):- average(X,W,AvX),average(Y,W,AvY),AvX=AvY, insert(X,T,NT,W). insert(X,[Y|T],[X,Y|T],W):- average(X,W,AvX),average(Y,W,AvY), AvX=AvY.**

**insert(X,[],[X],\_).**

Above predicates are classical insertion sort algorithm, but it compares team’s scores and inserts according to them. When two team’s scores are the same, it can either instert or not insert. It doesn’t matter as written in the project description.

**order(L,W) :- W>=1, allTeams(Teams,\_) , insert\_sort(Teams, L, W).**

Order predicate call allTeams predicate then sorts all teams’ list according to their averages with insert\_sort predicate.

**topThree([T1,T2, T3],W) :- order(L,W), L=[T1,T2,T3|\_].**

topThree predicate takes the list resulting from order predicate and puts T1,T2,T3 the list’s first three elements.

# Examples:

There are several examples below:

?- allTeams(L,N). L = [galatasaray, realmadrid, juventus, kobenhavn, manutd, realsociedad, shaktard, bleverkusen, omarseille, arsenal, fcnapoli, bdortmund]

N = 12;

L = [galatasaray, realmadrid, juventus, kobenhavn, manutd, realsociedad, shaktard, bleverkusen, omarseille, arsenal, bdortmund, fcnapoli]

N = 12;

L = [galatasaray, realmadrid, juventus, kobenhavn, manutd, realsociedad, shaktard, bleverkusen, omarseille, bdortmund, arsenal, fcnapoli]

N = 12

True

?- allteams([galatasaray, realmadrid, juventus, kobenhavn, manutd, realsociedad, shaktard, bleverkusen, omarseille, arsenal, fcnapoli, bdortmund], 12).

True

?- wins(galatasaray,4,L,N).

L = [kobenhavn] N = 1 ;

False

?- scored(juventus,5,S).

S = 9

?- conceded(juventus,5,C).

C = 8

?- average(kobenhavn, 3, A).

A = -6

?- order(L, 6).

L = [realmadrid, manutd, bdortmund, arsenal, fcnapoli, shaktard, juventus, bleverkusen, realsociedad, galatasaray, kobenhavn, omarseille]

?-topThree(L, 6).

L = [realmadrid, manutd, bdortmund]

# Improvements and Extensions:

The program works quite well for the test cases that are given in the project description. Right now, it is fast. But the sort algorithm can be improved. Insertion sort is a classical sorting algorithm and there are more efficient sorting algorithms available. So in the future that kind of algorithms can be used. But right now, the time complexity is not a major concern in project so this version can do well. Also instead of using break predicate to stop, I would like to use !. I think that is more beautiful way of terminating a predicate but we are not allowed to use it.

# Difficulties Encountered:

This is a project that is written with Prolog language, using logic programming styles. Because it is the first program I ever write using Prolog, it was a little bit hard for me to get used to it. But after a while with meddling with predicates and watching practice videos I could write the project without a problem. Stopping the recursion was a hardship for me but after looking for online I found break predicate, that was really important that I found it. Also, if I haven’t heard the findall function from my friends, it would have been really hard for me to find that function from Swi-Prolog web site. Also, coding the insertion sort was hard as well but with the help of Internet, I could do that without problems.

# Conclusion:

This project was a really important opportunity for me to practice logic programming with Prolog. I wouldn’t understand using logic programming if it was not for this project. I believe I improved a lot in understanding new kinds of languages with this project.

# Appendices:

The source code is given below:

/\*\* allTeams(L,N) is the predicate in which L is the list that contains all teams in random order and N is the number of elements in this list L. To find all the teams, findall(Object, Goal,List) predicate is used. That is a built in predicate of Prolog. To find N, the built-in predicate of Prolog lenght(List, Length) is used. And to list all permutations of the resulting list from findall predicate, permutation predicate is used.\*/

allTeams(L,N) :- findall(X,team(X,\_),Teamlist), permutation(L,Teamlist),length(Teamlist,N).

/\* winMatch implies that in the match between teams Y and T, T won the match on week \* W. \*/ winMatch(T,Y,W) :- match(W, T, W1,Y,W2), W1>W2.

winMatch(T,Y,W) :- match(W, Y, W1, T,W2), W1<W2.

/\*If the given week is a number less than 1 in wins predicate, it halts because week can not be less than one. Other \*parameters of wins predicate is \_(anonymous variable) because they don't affect the result when W is wrong number.\*/

wins(\_,W,\_,\_) :- W<1 , break.

/\*In wins predicate if the team is on week, then there is only the matches played in first week. This condition is also the ending step \*of recursion. \*/

wins(Team,1,List,Number) :- findall(Y, winMatch(Team,Y,1), List) , length(List, Number).

/\*In this predicate of wins, week is more than 1, meaning there is at least one previous week to calculate. So wins predicate \*of previous week is called in recursive fashion then resulting list is appended with the current weeks list. \*/

wins(T,W,L,N) :- W>1 , findall(Y, winMatch(T,Y,W), List1) , W1 is W-1, wins(T,W1, List2, \_) , append(List1, List2, List), sort(List,L), length(L,N).

/\* lossMatch implies that in the match between teams Y and T, T lost the match on week \* W. \*/ lossMatch(T,Y,W) :- match(W, T, L1,Y,L2), L1<L2.

lossMatch(T,Y,W) :- match(W, Y, L1, T,L2), L1>L2.

/\*If the given week is a number less than 1 in losses predicate, it halts because week can not be less than one. Other \*parameters of lossed predicate is \_(anonymous variable) because they don't affect the result when W is wrong number.\*/

losses(\_,W,\_,\_) :- W<1 , break.

/\*In losses predicate if the team is on week, then there is only the matches played in first week. This condition is also the ending step \*of recursion. \*/

losses(Team,1,List,Number) :- findall(Y, lossMatch(Team,Y,1), List) , length(List, Number).

/\*In this predicate of losses, week is more than 1, meaning there is at least one previous week to calculate. So losses predicate \*of previous week is called in recursive fashion then resulting list is appended with the current weeks list. \*/

losses(T,W,L,N) :- W>1 , findall(Y, lossMatch(T,Y,W), List1) , W1 is W-1, losses(T,W1, List2, \_) , append(List1, List2, List), sort(List,L), length(L,N).

/\*\* drawMatch implies that in the match between teams Y and T, their scores were the same on

week \* W.\*/

drawMatch(T,Y,W) :- match(W, T, D1,Y,D2), D1=D2.

drawMatch(T,Y,W) :- match(W, Y, D1, T,D2),D1=D2.

/\*If the given week is a number less than 1 in draws predicate, it halts because week can not be less than one. Other \*parameters of draws predicate is \_(anonymous variable) because they don't affect the result when W is wrong number.\*/

draws(\_,W,\_,\_) :- W<1 , break.

/\*In draws predicate if the team is on week, then there is only the matches played in first week. This condition is also the ending step \*of recursion. \*/

draws(Team,1,List,Number) :- findall(Y, drawMatch(Team,Y,1), List) , length(List, Number).

/\*In this predicate of draws, week is more than 1, meaning there is at least one previous week to calculate. So draws predicate \*of previous week is called in recursive fashion then resulting list is appended with the current weeks list. \*/

draws(T,W,L,N) :- W>1 , findall(Y, drawMatch(T,Y,W), List1) , W1 is W-1, draws(T,W1, List2, \_) , append(List1, List2, List), sort(List,L), length(L,N).

/\*If the given week is less than 1, that means error so the predicate is stopped.\*/

scored(\_,W,\_) :- W<1 ,break.

/\*If the given week is first week, then this implies that the scored goal number on this week is the S we are looking for. S is the total number of scored goals in matches in home and in guest team. To sum those the built-in function sum\_list is used. This is also the end of recursion. \*/

scored(T,W,S) :- W=1 , findall(X, match(W,T,X,\_,\_), HomeScore), findall(Y,match(W,\_,\_,T,Y),GuestScore), append(HomeScore, GuestScore, Score), sum\_list(Score, S).

/\*If the given week is more than one, that means there is at least one previous week to calculate. So in this predicate at first the current week's scored goal number is calculated like in the previous scored predicate and this Score is added to the previous weeks score.\*/

scored(T,W,S) :- W>1, findall(X, match(W,T,X,\_,\_), HomeScore), findall(Y,match(W,\_,\_,T,Y),GuestScore), append(HomeScore, GuestScore, CurrentScore),sum\_list(CurrentScore, Score), W1 is W-1, scored(T,W1, PrevScore), S is Score+PrevScore .

/\*Like in the previous predicates when W is less than one predicate terminates.\*/

conceded(\_,W,\_) :- W<1, break.

/\*If the given week is first week, then this implies that the conceded goal number on this week is the C we are looking for. C is the total number of conceded goals in matches in home and in guest team. To sum those the built-in function sum\_list is used. This is also the end of recursion. \*/ conceded(T,W,C) :- W=1 , findall(X, match(W,T,\_,\_,X), HomeConceded), findall(Y,match(W,\_,Y,T,\_),GuestConceded), append(HomeConceded, GuestConceded, Conceded), sum\_list(Conceded, C).

/\*If the given week is more than one, that means there is at least one previous week to calculate. So in this predicate at first the current week's conceded goal number is calculated like in the previous conceded predicate and this Conceded is added to the previous week's conceded.\*/ conceded(T,W,C) :- W>1, findall(X, match(W,T,\_,\_,X), HomeConceded),

findall(Y,match(W,\_,Y,T,\_),GuestConceded), append(HomeConceded, GuestConceded, CurrentConceded),sum\_list(CurrentConceded, Conceded), W1 is W-1, conceded(T,W1, PrevConceded), C is Conceded+PrevConceded .

/\*Week can not be less than one.\*/

average(\_,W,\_) :- W<1, break.

/\*S is the scored goal number by team T up to week W and C is the conceded goal number up to week W. A is S-C.\*/

average(T,W,A) :- scored(T,W,S) , conceded(T,W,C), A is S-C.

/\*Below predicates are the classical insertion sort algorithm that sorts a list of teams according to their averages on given week W in an descending order.\*/

insert\_sort(List,Sorted,W):-i\_sort(List,[],Sorted,W). i\_sort([],Acc,Acc,\_).

i\_sort([H|T],Acc,Sorted,W):-insert(H,Acc,NAcc,W),i\_sort(T,NAcc,Sorted,W).

insert(X,[Y|T],[Y|NT],W):- average(X,W,AvX),average(Y,W,AvY),AvX<AvY,insert(X,T,NT,W). insert(X,[Y|T],[X,Y|T],W):- average(X,W,AvX),average(Y,W,AvY), AvX>AvY.

insert(X,[Y|T],[Y|NT],W):- average(X,W,AvX),average(Y,W,AvY),AvX=AvY,insert(X,T,NT,W). insert(X,[Y|T],[X,Y|T],W):- average(X,W,AvX),average(Y,W,AvY), AvX=AvY.

insert(X,[],[X],\_).

/\*W must be larger or equal to one in order predicate because week con not be less than one. In this predicate Teams list is allTeams then L is the sorted version of Teams list.\*/

order(L,W) :- W>=1, allTeams(Teams,\_) , insert\_sort(Teams, L, W).

/\*T1,T2 and T3 in topThree predicate are the first three elements of the list L in order(L,W) predicate. \*/

topThree([T1,T2, T3],W) :- order(L,W), L=[T1,T2,T3|\_].