PROLOG

<https://cs.union.edu/~striegnk/courses/esslli04prolog/index.php>

**Lectie 1**

Exercitiul 2:

|  |
| --- |
| male(paul).  male(albert).  male(vernon).  male(james).  male(dudley).  male(harry).  female(helen).  female(ruth).  female(petunia).  female(lili).  parent\_of(paul, petunia).  parent\_of(paul, lili).  parent\_of(helen, petunia).  parent\_of(helen, lili).  parent\_of(albert, james).  parent\_of(ruth, james).  parent\_of(veron, dudley).  parent\_of(petunia, dudley).  parent\_of(lili, harry).  parent\_of(james, harry).  father\_of(F, C) :- male(F), parent\_of(F,C).  mother\_of(M,C) :- female(M), parent\_of(M,C).  grandfather\_of(GF, C) :- father\_of(GF, P) parent\_of(P,C).  grandmother\_of(GM, C) :- mother\_of(GM, P), parent\_of(P,C).  sister\_of(X,Y) :- female(X), parent\_of(P,X), parent\_of(P,Y), X\==Y.  aunt\_of(A,P) :- sister\_of(A,X), parent\_of(X,P).  brother\_of(X,Y) :- male(X), parent\_of(P,X), parent\_of(P,Y), X\==Y.  uncle\_of(X,Y) :- brother\_of(X,P), parent\_of(P,Y). |

Exercitiul 3:

|  |
| --- |
| starter(greenSalad).  starter(melon).  starter(tomatoSalad).  starter(pate).  main(salmon).  main(beef).  main(pasta).  desert(cheese).  desert(yoghurt).  desert(pastry).  menu(hungry,X,Y,Z) :- starter(X), main(Y), desert(Z).  menu(not\_so\_hungry, X,Y, nothing) :- starter(X), main(Y).  menu(not\_so\_hungry, nothing, Y, Z) :- main(Y), desert(Z).  menu(on\_diet, X, nothing, nothing) :- starter(X). |

Exercitiu extra 1: recursivitate

|  |
| --- |
| ancestor\_of(X,Y) :- parent\_of(X,Y).  ancestor\_of(X,Y) :- parent\_of(X,P), ancestor\_of(P,Y). |

**LECTIE 2 – recursivitate**

Ex 1

|  |
| --- |
| connected(1,2).  connected(3,4).  connected(5,6).  connected(7,8).  connected(9,10).  connected(12,13).  connected(13,14).  connected(15,16).  connected(17,18).  connected(19,20).  connected(4,1).  connected(6,3).  connected(4,7).  connected(6,11).  connected(14,9).  connected(11,15).  connected(16,12).  connected(14,17).  connected(16,19).  path(X,Y):-connected(X,Y).  path(X,Y):- connected(X,Z), path(Z,Y). |

Ex 2

We are given the following knowledge base of travel information:

Write a predicate travel/2 which determines whether it is possible to travel from one place to another by 'chaining together' car, train, and plane journeys. For example, your program should answer yes to the query travel(valmont,raglan).

So, by using travel/2 to query the above database, you can find out that it is possible to go from Valmont to Raglan. In case you are planning a travel, that's already very good information, but what you would then really want to know is how exactly to get from Valmont to Raglan. Write a predicate travel/3 which tells you how to travel from one place to another. The program should, e.g., answer yes to the query travel(valmont,paris,go(valmont,metz,go(metz,paris))) and X = go(valmont,metz,go(metz,paris,go(paris,losAngeles))) to the query travel(valmont,losAngeles,X).

Extend the predicate travel/3 so that it not only tells you via which other cities you have to go to get from one place to another, but also how (i.e. by car, train, or plane) you get from one city to the next.

|  |
| --- |
| byCar(auckland,hamilton).  byCar(hamilton,raglan).  byCar(valmont,saarbruecken).  byCar(valmont,metz).    byTrain(metz,frankfurt).  byTrain(saarbruecken,frankfurt).  byTrain(metz,paris).  byTrain(saarbruecken,paris).    byPlane(frankfurt,bangkok).  byPlane(frankfurt,singapore).  byPlane(paris,losAngeles).  byPlane(bangkok,auckland).  byPlane(losAngeles,auckland).  step(X,Y) :- byCar(X,Y).  step(X,Y) :- byPlane(X,Y).  step(X,Y) :- byTrain(X,Y).  /\*avel(X,Y):- step(X,Y).  travel(X,Y) :- step(X,Z), travel(Z,Y).  \*/  travel1(X,Y,go(X,Y)):-step(X,Y).  travel1(X,Y, go(X,Z,Path)):-step(X,Z), travel1(Z,Y,Path).  travel(X,Y,go(X,Y,Transport)) :- onestep(X,Y,Transport).  travel(X,Y,go(X,Z,Transport,Path)) :- onestep(X,Z,Transport),  travel(Z,Y,Path).  onestep(X,Y,byCar) :- byCar(X,Y).  onestep(X,Y,byTrain) :- byTrain(X,Y).  onestep(X,Y,byPlane) :- byPlane(X,Y). |

Ex 3:

|  |
| --- |
| word(abalone,a,b,a,l,o,n,e).  word(abandon,a,b,a,n,d,o,n).  word(enhance,e,n,h,a,n,c,e).  word(anagram,a,n,a,g,r,a,m).  word(connect,c,o,n,n,e,c,t).  word(elegant,e,l,e,g,a,n,t).  crosswd(V1,V2,V3,H1,H2,H3) :-  word(V1,\_,A,\_,B,\_,C,\_),  word(V2,\_,D,\_,E,\_,F,\_),  word(V3,\_,G,\_,H,\_,I,\_),  word(H1,\_,A,\_,D,\_,G,\_),  word(H2,\_,B,\_,E,\_,H,\_),  word(H3,\_,C,\_,F,\_,I,\_). |
|  |

**LECTIA 3 – LISTE**

**Am o lista full de a-uri si vreau sa o transform intr-o lista full de b-uri.**

|  |
| --- |
| trans([],[]).  trans([a|InTail], [b|OutTail]):- trans(InTail, OutTail). |

**Verific daca un element apare intr-o lista**

|  |
| --- |
| element\_of(X, [X|Tail]).  element\_of(X, [\_|Tail]) :- element\_of(X, Tail). |

Sau exista functia member/2

Member(a,[a,b,c]) – true

Member(X, [a,b,c]) – returneaza pe rand a,b,c

**Concatenez 2 liste**

|  |
| --- |
| concatenate([], L, L). % concateneaza prima chestie la a doua si intoarce rez in a 3a  concatenate([H|T], L, [H|T1]) :- concatenate(T,L,T1). |

**Ex 1: inmultesc fiecare elem al unei liste cu un scalar**

|  |
| --- |
| scalar(X, [],[]).  scalar(X, [H|T], [H1|T1]) :- H1 is X\*H, scalar(X, T, T1). |

**Ex 1.2 : am doua liste, inmultesc v[i] cu w[i] si adun rezultatul.**

|  |
| --- |
| dot([],[],0).  dot([H1|T1], [H2|T2], R) :- dot(T1,T2,R1), R is (H1\*H2)+R1. |

**Ex 2 : elementul maxim dintr o lista**

|  |
| --- |
| max([H], H).  max([H|T], Max) :- max(T, Max1), H>Max1, Max=H.  max([H|T], Max) :- max(T,Max1), H =< Max1, Max=Max1. |

**Ex 3: prefixul unei liste ( [a,b] e prefix pt [a,b,c,d])**

|  |
| --- |
| Var 1  prefix([],L).  prefix([H|T], [H|T1]) :- prefix(T,T1). |
| Var 2  prefix(P,L) :- append(P,\_,L). |

**Ex 3.1 : verifica daca o lista e sufixul altei liste ( [c,d] pt [a,b,c,d])**

|  |
| --- |
| suffix(S,L) :- append(\_,S,L). |

**Ex 3.2 : verifica daca o lista e sublista**

|  |
| --- |
| sublist(Sub,L) :- suffix(S,L), prefix(Sub,S). |

**Ex 4: sa returnez al n-lea element dintr o lista**

|  |
| --- |
| nth\_elem(N,L,R) :- elem(N, 1 ,L, R).  elem(N,N,[H|T],H).  elem(N, I, [H|T], R) :- I<N, I1 is I+1, elem(N,I1,T,R). |

**Ex 5 : [a,b,[c,d],[[1,2]],foo] we get the list [a,b,c,d,1,2,foo] and when we flatten the list [a,b,[[[[[[[c,d]]]]]]],[[1,2]],foo,[]] we also get [a,b,c,d,1,2,foo].**

|  |
| --- |
| my\_flatten([],[]).  my\_flatten([H|T], L) :- my\_flatten(H,H1), my\_flatten(T,T1), append(H1,T1,L).  my\_flatten([H|T], [H|T1]) :- H\=[], H\=[\_|\_], my\_flatten(T,T1). |

Ex 6 : There is a street with three neighboring houses that all have a different color. They are red, blue, and green. People of different nationalities live in the different houses and they all have a different pet. Here are some more facts about them:

The Englishman lives in the red house.

The jaguar is the pet of the Spanish family.

The Japanese lives to the right of the snail keeper.

The snail keeper lives to the left of the blue house.

Define a predicate zebra(?N) that helps you find out the nationality N of the owner of the zebra.

To solve this task you first have to come up with a suitable way of representing streets and houses.

|  |
| --- |
| %% zebra(N) is true if the zebra is the pet of the person with  %% nationality N.  zebra(N) :-  %% the street is represented as list of 3 houses  Street = [\_House1,\_House2,\_House3],  %% a house is represented as a 3-place (color, nationality, pet) complex term  %% there is a red house in the street  member(house(red,\_,\_), Street),  %% there is a blue house in the street  member(house(blue,\_,\_), Street),  %% there is a green house in the street  member(house(green,\_,\_), Street),  %% the Englishman lives in the red house  member(house(red,english,\_), Street),  %% the jaguar is the pet of the Spanish family  member(house(\_,spanish,jaguar), Street),  %% the Japanese lives to the right of the snail keeper  sublist([house(\_,\_,snail),house(\_,japanese,\_)], Street),  %% the snail keeper lives to the left of the blue house  sublist([house(blue,\_,\_),house(\_,\_,snail)], Street),  %% the zebra belongs to the person with nationality N  member(house(\_,N,zebra),Street).  %% member and append are built-in predicates, but you have to define  %% sublist and for that you need prefix and suffix.  prefix(P,L) :- append(P,\_,L).  suffix(S,L) :- append(\_,S,L). |

**Ex 7 : acumulatori**

**Lungime lista**

|  |
| --- |
| list\_length2(List,Length) :- list\_length2(List,0,Length).  list\_length2([],Length,Length).  list\_length2([\_|Tail],Accumulator,Length) :-  NewAcc is Accumulator + 1,  list\_length2(Tail,NewAcc,Length). |

**Max dintr o lista**

|  |
| --- |
| %% The 2-place wrapper predicate. It calls the 3-place predicate  %% max. The accumulator is instantiated to the head of the input list.  **max([Head|Tail],Max) :- max(Tail,Head,Max).**  %% The 3-place predicate max uses an accumulator to store the greatest  %% element that has been found so far. The first argument is the part  %% of the list that has not been processed so far, the second argument  %% is the accumulator, and the third argument is for the final result.  %% If the list is empty, the value that is in the accumulator (i.e.,  %% the current maximal element) is the maximum of the whole list.  **max([],Max,Max).**  %% If the head is greater than the value of the accumulator, then the  %% head is greatest element that we have found so far. It becomes our  %% new accumulator value.  **max([Head|Tail],Acc,Max) :- Head > Acc,**  **max(Tail,Head,Max).**  %% If the head is smaller or equal to the accumulator value, the  %% accumulator is still the greatest element that we have found so  %% far. So, the accumulator stays as it is.  **max([Head|Tail],Acc,Max) :- Head =< Acc,**  **max(Tail,Acc,Max).** |

**Reverse list**

|  |
| --- |
| mirror(L1, L2) :- mirror(L1, [], L2).  mirror([],L,L).  mirror([H|T], L, Out) :- mirror(T, [H|L], Out). |

**Ala ciudat cu flatten list**

|  |
| --- |
| %% Wrapper: calls the 3-place predicate with the accumulator argument  %% being instantiated to the empty list.  my\_flatten(In,Out) :-  my\_flatten(In,[],Out).  %% When the list is empty, the accumulator contains the final result.  **my\_flatten([],Acc,Acc).**  %% If the list is non-empty, flatten the tail of the list. The output  %% is TOut. Then use TOut to instantiate the accumulator when  %% flattening the head of the list.  **my\_flatten([H|T],Acc,Out) :- my\_flatten(T,Acc,TOut), my\_flatten(H,TOut,Out).**  %% A special clause to deal with cases where my\_flatten is called with  %% a non-list as first argument. This can happen because the previous  %% rule calls my\_flatten for the head of its input list and this head  %% is not necessarily a list. In this case, the term is simply added  %% to the front of the accumulator.  **my\_flatten(X,Acc,[X|Acc]) :- X \= [], X \= [\_|\_].** |

**Ex 8: elimina prima aparitie a unui elem.**

|  |
| --- |
| remove(\_,[],[]).  remove(X,[X|T], T) :- ! . % practic e un fel de break, daca a ajuns aici iese din program  remove(X,[H|T],[H|L]) :- remove(X,T,L). |

**LABORATOARE**

**Lab 1:**

**Ex 2 : arborele genealogic**

|  |
| --- |
| father\_of(F,C):- male(F), parent(F,C).  mother\_of(M,C):- female(M),parent(M,C).  grandmother\_of(G,C):- mother\_of(G,P), parent(P,C).  grandfather\_of(G,C):- father\_of(G,P), parent(P,C).  sister\_of(S,P) :- parent(Q,S), parent(Q,P), female(S), S \= P.  brother\_of(B,P):- parent(Q,B), parent(Q,P), male(B), B \= P.  aunt\_of(A,P) :- sister\_of(A,B), parent(B,P).  uncle\_of(A,B) :- brother\_of(A,C), parent(C,B). |

**Ex 3 : not parent**

|  |
| --- |
| % not\_parent(X,Y) :- not(parent(X,Y)). – asa nu  person(X) :- male(X).  person(X) :- female(X).  not\_parent(X,Y) :- person(X), person(Y), X\=Y, not(parent(X,Y)). |

**LAB 2**

**Ex 1: distanta intre 2 puncte**

|  |
| --- |
| distance((A,B),(C,D),X) :- X is sqrt((C-A)\*\*2+(D-B)\*\*2). |

**Ex 2 : al n-lea numar fibonacci**

|  |
| --- |
| fib(0,1).  fib(1,1).  fib(N,X) :- 2 =< N, M is N - 1, fib(M, Y), P is N - 2, fib(P, Z), X is Y + Z.  fibo(0,0,1).  fibo(1,1,1).  fibo(N,Z,X) :- 2 =< N, M is N-1, fibo(M,Y,Z), X is Y + Z.  fibg(N,X) :- fibo(N,\_,X). |

**Ex 3 : sa desenez un patrat de n x n caractere**

|  |
| --- |
| line(0,\_).  line(X,C):- X>0, Y is X-1, write(C), line(Y,C).  rectangle(0,\_,\_):-nl.  rectangle(X,Z,C):- X>0, Y is X-1, line(Z,C), nl, rectangle(Y,Z,C).  square(X,C) :- rectangle(X,X,C). |

**Ex 4.1 : verific daca o lista e formata doar din a-uri**

|  |
| --- |
| all\_a([]).  all\_a([a|T]) :- all\_a(T). |

**Ex 4.2 : transforma o lista de a-uri intr-o lista de b-uri**

|  |
| --- |
| trans\_ab([],[]).  trans\_ab([a|T], [b|L]) :- trans\_ab(T, L). |

**Ex 5.1 : inmultesc fiecare element cu un scalar**

|  |
| --- |
| scalar(\_, [], []).  scalar(X, [H|T], [H1|T1]) :- H1 is X\*H, scalar(X,T,T1). |

**Ex 5.2 : produsul scalar al 2 vect**

|  |
| --- |
| dot([],[],0).  dot([H|T], [H1|T1], R) :- dot(T,T1,R1), R is R1+H\*H1. |

**Ex 5.3 : maximul dintr o lista**

|  |
| --- |
| max([H|T], Max) :- max(T,H,Max).  max([],Max,Max).  max([H|T], I, Max) :- H>I, max(T,H,Max).  max([H|T],I,Max) :- H=<I, max(T,I,Max) |

**Lab 3**

**Ex 1 : palindrom. Verific daca o lista este palindrom**

|  |
| --- |
| rev([],[]).  rev([H|T], L) :- rev(T,N), append(N,[H],L).  palindrom(L) :- rev(L,L). |

**Ex 2: sterge toate duplicatele dint-o lista**

|  |
| --- |
| remove\_d([],[]).  remove\_d([H|T],L) :- remove\_d(T, L), member(H,L).  remove\_d([H|T],[H|L]) :- remove\_d(T, L), not(member(H,L)). |

**Extra : daca vreau sa pastrez prima aparitie(pastrez ordinea)**

|  |
| --- |
| remove\_d\_orig(L, R) :- remove\_d\_orig(L, [], R).  remove\_d\_orig([], \_, []).  remove\_d\_orig([H|T], Seen, R) :- member(H, Seen), !, remove\_d\_orig(T, Seen, R).  remove\_d\_orig([H|T], Seen, [H|R]) :- remove\_d\_orig(T, [H|Seen], R). |

**Ex 3 : elementul din primul arg aparent in lista de k ori**

|  |
| --- |
| atimes(\_,[],0).  atimes(N,[N|T],X) :- atimes(N,T,Y), X is Y + 1.  atimes(N,[H|T],X) :- atimes(N,T,X), H \= N. |

SORTARI

**Insert sort**

|  |
| --- |
| insertsort([],[]).  insertsort([H|T],L) :- insertsort(T,L1), insert(H,L1,L).  insert(X,[],[X]).  insert(X,[H|T],[X|[H|T]]) :- X < H.  insert(X,[H|T],[H|L]) :- X >= H, insert(X,T,L). |

**Quick Sort**

|  |
| --- |
| quicksort([],[]).  quicksort([H|T],L) :- split(H,T,A,B), quicksort(A,M), quicksort(B,N), append(M,[H|N],L).  split(\_,[],[],[]).  split(X,[H|T],[H|A],B) :- H < X, split(X,T,A,B).  split(X,[H|T],A,[H|B]) :- H >= X, split(X,T,A,B). |

**LABORATOR 4**

**Ex 1 : listaNelem(L,N,M) este adev cand M este o lista cu N elem care sunt toate ale lui L.**

|  |
| --- |
| lista(\_,0,[]).  lista(L,N,[H|T]) :- N>0, P is N-1, member(H,L), lista(L,P,T). |

**Ex 2 : puzzle ul ala enervant**

|  |
| --- |
| word(abalone,a,b,a,l,o,n,e).  word(abandon,a,b,a,n,d,o,n).  word(enhance,e,n,h,a,n,c,e).  word(anagram,a,n,a,g,r,a,m).  word(connect,c,o,n,n,e,c,t).  word(elegant,e,l,e,g,a,n,t).  crosswd(V1,V2,V3,H1,H2,H3) :-  word(V1,\_,A,\_,B,\_,C,\_),  word(V2,\_,D,\_,E,\_,F,\_),  word(V3,\_,G,\_,H,\_,I,\_),  word(H1,\_,A,\_,D,\_,G,\_),  word(H2,\_,B,\_,E,\_,H,\_),  word(H3,\_,C,\_,F,\_,I,\_). |

**Ex 3 : cu drumurile**

|  |
| --- |
| path(X,X,[X]).  path(X,Y,[X|L]) :- connected(X,Z), path(Z,Y,L).  pathc(X,Y) :- path(X,Y,\_). |

Ex 4 :

**Part 1 : descompune un atom intr o lista de caractere**

|  |
| --- |
| word\_letters(X,Y) :- atom\_chars(X,Y). |

**Part 2 : verifica daca prima lista e inclusa in totalitate in a doua (daca o litera apare de 2 ori in prima, trebuie sa apara de minim 2 ori si in a doua)**

|  |
| --- |
| liminus([C|L],C,L).  liminus([D|L],C,[D|M]) :- D\==C, liminus(L,C,M).  cover([],\_).  cover([H|T],L) :- liminus(L,H,M), cover(T,M). |

**Part 3 : primeste o lista de litere, si un scor si returneaza cuvantul din lista cu n litere**

|  |
| --- |
| solution(Letters, Word, Len) :- word(Word), word\_letters(Word,WordLetters), length(WordLetters,Len), cover(WordLetters, Letters). |

**Cod final pt exerictiu**

|  |
| --- |
| word\_letters(X,Y) :- atom\_chars(X,Y).  liminus([C|L],C,L).  liminus([D|L],C,[D|M]) :- D\==C, liminus(L,C,M).  cover([],\_).  cover([H|T],L) :- liminus(L,H,M), cover(T,M).  solution(Letters, Word, Len) :- word(Word), word\_letters(Word,WordLetters), length(WordLetters,Len), cover(WordLetters, Letters).  search\_solution(\_,'no solution',0).  search\_solution(ListLetters,Word,X) :- X > 0, solution(ListLetters,Word,X).  search\_solution(ListLetters,Word,X) :- X > 0, not(solution(ListLetters,Word,X)), Y is X-1, search\_solution(ListLetters,Word,Y).  topsolution(ListLetters,Word) :- length(ListLetters, MaxScore), search\_solution(ListLetters,Word,MaxScore). |

**LAB 5**

**Ex 1: un predicat care are ca prim argument o formula si al doilea lista cu multimea variabilelor care apar in formula**

|  |
| --- |
| vars(V,[V]) :- atom(V).  vars(non(X),S) :- vars(X,S).  vars(si(X,Y),S) :- vars(X,T), vars(Y,U), union(T,U,S).  vars(sau(X,Y),S) :- vars(X,T), vars(Y,U), union(T,U,S).  vars(imp(X,Y),S) :- vars(X,T), vars(Y,U), union(T,U,S). |

Ex 2:

|  |
| --- |
| val(V,[(V,A)|\_],A).  val(V,[\_|T],A) :- val(V,T,A).  %Solutie alternativa:  val(V,E,A) :- member((V,A),E). |

Ex 3:

|  |
| --- |
| bnon(0,1). bnon(1,0).  bsi(0,0,0). bsi(0,1,0). bsi(1,0,0). bsi(1,1,1).  bsau(0,0,0). bsau(0,1,1). bsau(1,0,1). bsau(1,1,1).  % X -> Y = (non X) sau Y  bimp(X,Y,Z) :- bnon(X,NX), bsau(NX,Y,Z). |

Ex 4:

|  |
| --- |
| eval(V,E,A) :- atom(V), val(V,E,A).  eval(non(X),E,A) :- eval(X,E,B), bnon(B,A).  eval(si(X,Y),E,A) :- eval(X,E,B), eval(Y,E,C), bsi(B,C,A).  eval(sau(X,Y),E,A) :- eval(X,E,B), eval(Y,E,C), bsau(B,C,A).  eval(imp(X,Y),E,A) :- eval(X,E,B), eval(Y,E,C), bimp(B,C,A). |

Ex 5:

|  |
| --- |
| evals(\_,[],[]).  evals(X,[E|Es],[A|As]) :- eval(X,E,A), evals(X,Es,As). |

Ex 6:

|  |
| --- |
| evs([],[[]]).  evs([V|T],Es) :- evs(T,Esp), adauga(V,Esp,Es).  adauga(\_,[],[]).  adauga(V,[E|T], [[(V,0)|E],[(V,1)|E]|Es]) :- adauga(V,T,Es). |

Ex 7 :

|  |
| --- |
| all\_evals(X,As) :- vars(X,S), evs(S,Es), evals(X,Es,As). |

Ex 8:

|  |
| --- |
| all\_ones([]).  all\_ones([1|T]) :- all\_ones(T).  taut(X) :- all\_evals(X,As), all\_ones(As). |

**LAB SUPLIMENTAR**

**Chestii useful**

|  |
| --- |
| last\_but\_one(X, [X, \_]).  last\_but\_one(X, [\_ | T]) :- last\_but\_one(X, T).  % fibonacci eficient  fiba(0, 1, \_).  fiba(1, 1, 1).  %trb pus ultimul Rtt la 1 pt calc lui fiba(2)  fiba(X, R, Rt):- Xt is X - 1,  fiba(Xt, Rt, Rtt),  R is Rt + Rtt.  fib(X, R):- fiba(X, R, \_).  len([], 0).  len([\_|T], K):- len(T, K1), K is K1 + 1.  element\_of([X|\_], X).  element\_of([\_|T], X):- element\_of(T, X).  max([X], X).  max([H|T], R) :- max(T, R1), H > R1, R is H.  max([H|T], R) :- max(T, R1), H =< R1, R is R1.  reverse([], []).  reverse([H|T], R):- reverse(T, R1), append(R1, [H], R).  palindrom(L):- reverse(L, L).  concat\_lists([], L2, L2).  concat\_lists([H|L1], L2, [H|L3]):- concat\_lists(L1, L2, L3).  remove\_duplicates([], []).  remove\_duplicates([H|T], L) :- remove\_duplicates(T, L),  member(H, L).  remove\_duplicates([H|T], [H|L]) :- remove\_duplicates(T, L),  \+ member(H, L).  % atimes(E, L, N) - E apare de N ori in L  atimes(\_, [], 0).  % in caz ca interogarea mea il va avea pe E ca necunoscuta, trb sa adaug si acest pas de oprire.  atimes(X, [X], 1).  atimes(E, [H|T], K) :- atimes(E, T, K1), H == E, K is K1 + 1.  atimes(E, [H|T], K) :- atimes(E, T, K), H \== E.  % bonus: dropN(L, R, N) - elimina fiecare al N-lea element din lista L  dropN(L, [], N) :- length(L, N).  dropN([H|T], [H|R], N) :- length(T, L1), L1 >= N, dropN(T, R, N).  dropN(L, R, N) :- append(R, L1, L), length(L1, N).  % insertion sort  insertsort([],[]).  insertsort([H|T],L) :- insertsort(T,L1), insert(H,L1,L).  insert(X,[],[X]).  insert(X,[H|T],[X|[H|T]]) :- X < H.  insert(X,[H|T],[H|L]) :- X >= H, insert(X,T,L).  % bonus: quicksort  quicksort([],[]).  quicksort([H|T],L) :- split(H,T,A,B), quicksort(A,M), quicksort(B,N),  append(M,[H|N],L).  split(\_,[],[],[]).  split(X,[H|T],[H|A],B) :- H < X, split(X,T,A,B).  split(X,[H|T],A,[H|B]) :- H >= X, split(X,T,A,B). |

|  |
| --- |
| %Apartenenta unui element la o lista  apartine(X, [X|\_]).  apartine(X, [\_|T]) :- apartine(X, T).  % lungimea unei liste  len([],0).  len([\_|T], CNT):- len(T,CN), CNT is CN + 1.  % Verifica daca toate elementele unei liste sunt la fel  all([],\_).  all([C|T], C) :- all(T,C).  % Factorial  fac(0,1).  fac(Element, Produs) :- N is Element - 1, fac(N, Aux),  Produs is Element\*Aux.  % Numerele pare dintr-o lista  even(List, X) :- findall(Y, (member(Y, List), 0 is Y mod 2), X).  par([],[]).  par([H|T], [H|B]) :- par(T,B), 0 is H mod 2.  par([\_|T], B) :- par(T,B).  % Numerele impare pana la N(valoare data)  impare\_pana\_la(List, N) :- findall(Y, (between(1, N, Y), 1 is Y mod 2), List).  % Inversarea unei liste  rev([],[]).  rev([H|T],L) :- rev(T, N), append(N, [H], L).  % MODEL EXAMEN LABORATOR  %EX1  lista\_puncte([],\_,[]).  lista\_puncte([punct(X,Y)|T1], Val, [punct(X,Y)|Lrez]):- Y>Val, lista\_puncte(T1, Val, Lrez).  lista\_puncte([punct(\_,Y)|T1], Val, Lrez):- Y=<Val, lista\_puncte(T1, Val, Lrez).  %EX2  dropN(L, R, N) :- append(R, L1, L), length(L1, N).  %EX3  rmdn(Phi, Phi) :- atom(Phi).  rmdn(non(Phi), non(Phi)) :- atom(Phi).  rmdn(non(non(Phi)), Psi) :- rmdn(Phi, Psi).  rmdn(non(si(Phi,Psi)),non(A)) :- rmdn(si(Phi,Psi),A).  rmdn(non(sau(Phi,Psi)),non(A)) :- rmdn(sau(Phi,Psi),A).  rmdn(non(imp(Phi,Psi)),non(A)) :- rmdn(imp(Phi,Psi),A).  rmdn(si(Phi, Psi), si(Phi1, Psi1)) :- rmdn(Phi, Phi1), rmdn(Psi, Psi1).  rmdn(sau(Phi, Psi), sau(Phi1, Psi1)) :- rmdn(Phi, Phi1), rmdn(Psi, Psi1).  rmdn(imp(Phi, Psi), imp(Phi1, Psi1)) :- rmdn(Phi, Phi1), rmdn(Psi, Psi1). |

**Model 2024**

|  |
| --- |
| ex1:  lista\_puncte([],\_,[]).  lista\_puncte([punct(X,Y)|T1], Val, [punct(X,Y)|Lrez]):- Y>Val, lista\_puncte(T1, Val, Lrez).  lista\_puncte([punct(\_,Y)|T1], Val, Lrez):- Y=<Val, lista\_puncte(T1, Val, Lrez).  ex2:  dropN(L, R, N) :- append(R, L1, L), length(L1, N).  ex3:  rmdn(Phi, Phi) :- atom(Phi).  rmdn(non(Phi), non(Phi)) :- atom(Phi).  rmdn(non(non(Phi)), Psi) :- rmdn(Phi, Psi).  rmdn(non(si(Phi,Psi)),non(A)) :- rmdn(si(Phi,Psi),A).  rmdn(non(sau(Phi,Psi)),non(A)) :- rmdn(sau(Phi,Psi),A).  rmdn(non(imp(Phi,Psi)),non(A)) :- rmdn(imp(Phi,Psi),A).  rmdn(si(Phi, Psi), si(Phi1, Psi1)) :- rmdn(Phi, Phi1), rmdn(Psi, Psi1).  rmdn(sau(Phi, Psi), sau(Phi1, Psi1)) :- rmdn(Phi, Phi1), rmdn(Psi, Psi1).  rmdn(imp(Phi, Psi), imp(Phi1, Psi1)) :- rmdn(Phi, Phi1), rmdn(Psi, Psi1). |

**Model 2023**

|  |
| --- |
| /\*  Problema 1  Teste:  1) ?- consec([5, 4, 3, 2, 1]).  2) ?- consec([1, 2, 3, 5, 4]).  3) ?- consec([3, 4, 5, 6, 7]).  Rezultate:  1) false  2) false  3) true  Explicatie:  consec([]) si consec[[X]] sunt implicit adevarate.  consec([X, Y|T]) verifica daca elementele X si Y sunt consecutive cu intructiunea Y =:= X + 1, care evalueaza mai intai X + 1 si apoi il compara cu Y.  Daca intructiunea este adevarata, se trece recursiv in lista fara X, si anume prin consec([Y|T]).  \*/  consec([]).  consec([\_]).  consec([X, Y|T]):- Y =:= X + 1, consec([Y|T]).  /\*  Problema 2  Teste:  1) ?- lista\_angajati([angajat(ion, 10), angajat(mirela, 11), angajat(marcel, 12), angajat(ioana, 13), angajat(andrei, 14)], 12, R).  2) ?- lista\_angajati([angajat(mircea, 100), angajat(ioana, 200), angajat(mihai, 300)], 301, R).  3) ?- lista\_angajati([angajat(a, 9), angajat(b, 8), angajat(c, 7), angajat(d, 6), angajat(e, 5), angajat(f, 4), angajat(g, 3)], 2, R).  Rezultate:  1) R = [ioana, andrei]  2) R = []  3) R = [a, b, c, d, e, f, g]  Explicatie:  lista\_angajati([], \_, []) reprezinta cazul de baza, care returneaza o lista vida cand lista de angajati este vida, oricat ar fi pragul de salariu  lista\_angajati([angajat(X, Y)|T], S, [X|R]) reprezinta cazul in care Y, adica salariul angajatului X, este strict mai mare decat pragul de salariu, caz in care X se adauga la R prin [X|R] si se apleaza recursiv lista\_angajati(T, S, R).  lista\_angajati([angajat(\_, Y)|T], S, R) reprezinta cazul in care Y este mai mic sau egal fata de pragul de salariu, caz in care nu se adauga nimic la R si se apleaza recursiv pana la lista vida.  \*/  lista\_angajati([], \_, []).  lista\_angajati([angajat(X, Y)|T], S, [X|R]):-  Y > S, lista\_angajati(T, S, R).  lista\_angajati([angajat(\_, Y)|T], S, R):-  Y =< S, lista\_angajati(T, S, R). |