UNIVERSIDADE ESTADUAL DO SUDOESTE DA BAHIA
CIÊNCIA DA COMPUTAÇÃO
PROGRAMAÇÃO DECLARATIVA
STÊNIO LONGO ARAÚJO

ATIVIDADE DE PROLOG 03

COMPONENTES DO GRUPO:
- ALAN BOMFIM
- BRENO CARVALHO
- DENISE NOGUEIRA
- ÉRIC VINÍCIUS
- VITOR ROSENBERGRE

LINK VÍDEO:
https://drive.google.com/file/d/15xR_FWxN1GUxDBzjcLfN8r9lGA6nN2jJ/view

```
% 10- /* L é uma lista */
% Se ela estiver vazia ou possuir 1 ou mais elementos, ela é uma lista.
list([]).
list([_|_]).

test_List(x) :-
        trace,write("[0,1,2,3,4] = "),list([0,1,2,3,4]),nl;
        trace,write("[] = "),list([]),nl;
        trace,write("[0] = "),list([0]),nl;
        trace,write(" 0 = "),list(0),nl.

% 11- /* X é um elemento da lista L */
% Se o elemento procurado pro o primeiro da lista, já aceita.
% Se não, continua buscando no restante da lista.
% Member 1 e 2 são formas de fazer diferentes mas possuem o mesmo algoritmo.
%%-------------------------------------------------------%%
member(X, [X|_]).
member(X, [_|L]) :- member(X, L).

member2(X,[H|_]) :- X==H,!.
member2(X,[_|L]) :- member2(X,L).

%test com a primeira regra (member)
test_Member(x) :-
   trace,write("2 M [0,1,2,3,4] = "),member(2,[0,1,2,3,4]),nl;
   trace,write("2 M [2,0] = "),member(2,[2,0]),nl;
   trace,write("2 M [] = "),member(2,[]),nl.

%teste com a segunda regra (member2)
test_Member2(x) :-
        trace,write("2 M [0,1,2,3,4] = "),member2(2,[0,1,2,3,4]),nl;
```

```
        trace,write("2 M [2,0] = "),member2(2,[2,0]),nl;
        trace,write("2 M [] = "),member2(2,[]),nl.

% 12- /* X não é um elemento da lista L */
% Se esvaziou a lista e não encontrou o elemento, então a regra é verdadeira.
% Se faz uma analise em todos os elementos da lista para verificação.
%  A segunda regra (not_member2) utiliza da regra (member) e verifica, se
%  não for member, ela é verdadeira, caso contrário, falsa.
%%-------------------------------------------------------%%
not_member(_, []).
not_member(X, [Y|L]) :-
   dif(X,Y),
   not_member(X, L).

not_member2(X,L) :- not(member(X,L)).

%test com a primeira regra (not_member)
test_notMember(x) :-
        trace,write("2 NM [] = "),not_member(2,[]),nl;
        trace,write("2 NM [0,1,2,3,4] = "),not_member(2,[0,1,2,3,4]),nl.

%teste com a segunda regra (not_member2)
test_notMember2(x) :-
        trace,write("2 NM [] = "),not_member2(2,[]),nl;
        trace,write("2 NM [0,1,2,3,4] = "),not_member2(2,[0,1,2,3,4]),nl.


% 13- prefix(Prefix,List) /* Prefix é um prefixo de List */
% Verifica o se o Prefix vai ser um prefixo de List.
%%-------------------------------------------------------%%

prefix([],_).
prefix([X|P], [X|L]) :- prefix(P, L).

test_Prefix(x) :-
        trace,write("[1,2,3] P [1,2,3,4] = "),prefix([1,2,3],[1,2,3,4]),nl;
        trace,write("[] P [] = "),prefix([],[]),nl;
        trace,write("[1,3] P [1,2,3] = "),prefix([1,3],[1,2,3]),nl.

% 14- sufix(Sufix,List)/* Sufix é um sufixo de List */
% Verifica o se o Sufix vai ser um sufixo de List.
% Utiliza a regra auxiliar equal, que vai ver se os
% elementos das listas passadas são iguais.
%%-------------------------------------------------------%%

equal([], []).
equal([X|L1], [X|L2]) :- equal(L1, L2).
```

```
sufix([], []).
sufix([X|S], [X|L]) :- equal(S, L).
sufix(S, [_|L]) :- sufix(S, L).

test_Sufix(x) :-
        trace,write("[1,2,3] S [0,4,2,1,2,3] = "),sufix([1,2,3],[0,4,2,1,2,3]),nl;
        trace,write("[] S [] = "),sufix([],[]),nl;
        trace,write("[1,2,3] S [1,2,3,4,5] = "),sufix([1,2,3],[1,2,3,4,5]),nl.

% 15 - sublist(Sub,List)/* Sub é uma sublista de List */
% Vai verificar se a lista Sub é encontrada na List.
% Se os elementos iniciais de cada lista forem iguais, ele faz o prefix.
% Se não, ele continua buscando na List.
%%-------------------------------------------------------%%

sublist([], []).
sublist([X|Sub], [X|List]) :-
   prefix([X|Sub], [X|List]).
sublist([X|L1], [Y|L2]) :- dif(X,Y),sublist([X|L1], L2).

test_Sublist(x) :-
        trace,write("[1,2,3] Sublist [0,4,2,1,2,3] = "),sublist([1,2,3],[0,4,2,1,2,3]),nl;
        trace,write("[] Sublist [] = "),sublist([],[]),nl;
        trace,write("[1,2,3] Sublist [1,3,4,5,2] = "),sublist([1,2,3],[1,3,4,5,2]),nl.

% 16 - append(L1,L2,List)/* List é o resultado da concatenação de L1 e L2 */
% Passa por toda a lista L1 e L2 e concatena seus elementos em List,
% até as listas L1 e L2 estiverem vazias.
% As duas regras são basicamente o mesmo algoritmo
%%-------------------------------------------------------%%
append2([], [], []).
append2([], [X|L2], List) :-
   append2([], L2, List2),
   insert(X, List2, List).
append2([X|L1], L2, List) :-
   append2(L1, L2, List2),
   insert(X, List2, List).

append([],[],[]).
append([],[H|T],[H|Y]) :- append([],T,Y).
append([H|T],L2,[H|Y]) :- append(T,L2,Y).

%test com a primeira regra (append)
test_Append(x) :-
        trace,write("[1,2,3] AP [0,4,2,1,2,3] = "),append([1,2,3],[0,4,2,1,2,3],A),write(A),nl;
        trace,write("[] AP [] = "),append([],[],A),write(A),nl;
        trace,write("[1,2,3] AP [1,3,4,5,2] = "),append([1,2,3],[1,3,4,5,2],A),write(A),nl.
```

```
%test com a segunda regra (append2)
test_Append2(x) :-
        trace,write("[1,2,3] AP [0,4,2,1,2,3] = "),append2([1,2,3],[0,4,2,1,2,3],A),write(A),nl;
        trace,write("[] AP [] = "),append2([],[],A),write(A),nl;
        trace,write("[1,2,3] AP [1,3,4,5,2] = "),append2([1,2,3],[1,3,4,5,2],A),write(A),nl.

% 17 - reverse(List,Rev) /* Rev é o resultado de reverter List */
% Vai pegar todos os elementos da lista e concetenar numa nova lista de tras
% para frente, utilizando a regra append2 como auxilio.
%%-------------------------------------------------------%%
reverse([], []).
reverse([X|List], Rev) :-
   reverse(List, Rev2),
   append2(Rev2, [X], Rev).

test_Reverse(x) :-
        trace,write("[1,2,3] = "),reverse([1,2,3],R),write(R),nl;
        trace,write("[] = "),reverse([],R),write(R),nl;
        trace,write("[1,3,4,5,2] = "),reverse([1,3,4,5,2],R),write(R),nl.

% 18 - adjacent(X,Y,List)/* X e Y são elementos adjacentes em List */
% Verifica se X e Y são elementos adjacentes em List, verificando os elementos adjacentes
da lista
% e verificando se X e Y se encaixam na adjacencia. Se não encontrar nos primeiros
adjacentes,
% continua buscando na lista List.
% %%-------------------------------------------------------%%
adjacent(X, Y, [X1|[Y1|_]]) :-
   ( X =:= X1,
     Y =:= Y1 )
   ;
   ( X =:= Y1,
     Y =:= X1).
adjacent(X, Y, [_|[Y1|List]]) :-
   insert(Y1, List, List2),
   adjacent(X, Y, List2).

adjacent2(X,Y,[H|[H2|_]]) :- X==H, Y==H2,!.
adjacent2(X,Y,[H|[H2|_]]) :- X==H2, Y==H,!.
adjacent2(X,Y,[_|T]) :- adjacent2(X,Y,T).

%test com a primeira regra (adjacent)
test_Adjacent(x) :-
   trace,write("1,2  ADJ  [0,1,2] = "),adjacent(1,2,[0,1,2]),nl;
   trace,write("1,2  ADJ  [0,2,1,0] = "),adjacent(1,2,[0,2,1,0]),nl;
   trace,write("1,2  ADJ  [] = "),adjacent(1,2,[]),nl.

%test com a segunda regra (adjacent2)
```

```
test_Adjacent2(x) :-
   trace,write("1,2  ADJ  [0,1,2] = "),adjacent2(1,2,[0,1,2]),nl;
   trace,write("1,2  ADJ  [0,2,1,0] = "),adjacent2(1,2,[0,2,1,0]),nl;
   trace,write("1,2  ADJ  [] = "),adjacent2(1,2,[]),nl.

% 19 - length(List,N)/* N é o número de elementos de List */
% Enquanto a lista não estiver vazia, Vai somando +1 no valor de N.
% Quando estiver vazia, soma N  = N +0 e imprime N.
%%----------------------------------------------------%%
length2([], 0).
length2([_|List], N) :-
   length2(List, N1),
   N is N1+1.

test_Length(x) :-
   trace,write("[0,1,2] = "),length2([0,1,2],N),write(N),nl;
   trace,write("[0,2,1,0] = "),length2([0,2,1,0],N),write(N),nl;
   trace,write("[] = "),length2([],N),write(N),nl.

% 20 - first(X,List) /* X é o primeiro elemento de List */
% Verifica o primeiro elemento de List.
%%----------------------------------------------------%%
first(First, [First|_]).

test_First(x) :-
   trace,write("First [0,1,2] = "),first(N,[0,1,2]),write(N),nl;
   trace,write("First [2,1,0] = "),first(N,[2,1,0]),write(N),nl;
   trace,write("First [] = "),first(N,[]),write(N),nl.


% 21 - last(X,List) /* X é o último elemento de List */
% % Verifica o ultimo elemento de List.
%%----------------------------------------------------%%
last(Last, [Last]).
last(Last, [_|List]) :- last(Last, List).


test_Last(x) :-
   trace,write("Last [0,1,2] = "),last(N,[0,1,2]),write(N),nl;
   trace,write("Last [2,1,0] = "),last(N,[2,1,0]),write(N),nl;
   trace,write("Last [] = "),last(N,[]),write(N),nl.

% 22 - nth(X,N,List) /* X é o N-ésimo elemento de List */
% Vai até uma posicao na lista e mostra o elemento.
%%----------------------------------------------------%%
nth(X, 0, [X|_]).
nth(X, N, [_|List]) :-
   N1 is N-1,
```

```
    nth(X, N1, List).

test_Nth(x) :-
    trace,write("NTH 2 [0,1,2] = "),nth(N,2,[0,1,2]),write(N),nl;
    trace,write("NTH 3 [0,2,1,0,3,7,5] = "),nth(N,3,[0,2,1,0,3,7,5]),write(N),nl;
    trace,write("NTH 0 [] = "),nth(N,0,[]),write(N),nl.

% 23 - double(L, LL) /* Cada elemento de L aparece em LL em duplicado */
% Duplica cada elemento e insere em LL. Utiliza a regra insert para fazer a dupla
% inserção do elemento da lista.
%%-------------------------------------------------------%%
double([], []).
double([X|L], LL) :- double(L, LL2),
    insert(X, LL2, LL3),
    insert(X, LL3, LL).

test_Double(x) :-
    trace,write("[0,1,2] = "),double([0,1,2],N),write(N),nl;
    trace,write("[0,2,1,0,3,7,5] = "),double([0,2,1,0,3,7,5],N),write(N),nl;
    trace,write("[] = "),double([],N),write(N),nl.

% 24 - sum(Xs,Sum) /* Sum é a soma dos elementos de Xs */
% Passa pela lista armazena o valor da soma dos elementos em Num.
% Que como disse, vai representar a soma dos elementos da lista.
%%-------------------------------------------------------%%
sum([], 0).
sum([X|Xs], Num) :-
   sum(Xs, Y),
   Num is X+Y.

test_Sum(x) :-
    trace,write("[0,1,2] = "),sum([0,1,2],N),write(N),nl;
    trace,write("[0,2,1,0,3,7,5] = "),sum([0,2,1,0,3,7,5],N),write(N),nl;
    trace,write("[] = "),sum([],N),write(N),nl.

% 25- delete(L1,X,L2) /* L2 resulta da eliminação de todos os X de L1 */
% Remove todos os X de L1 e insere os elementos não X em L2.
% Utiliza da regra insert para inserção e dif para analisar a diferença dos elementos da
% lista com o X. Vai remover todos os elementos de L1 e fazer a analise. Todos
% os diferentes de X vão ser inseridos em L2.
%%-------------------------------------------------------%%
delete([], _, []).
delete([E|L1], E, L2) :- delete(L1, E, L2).
delete([E|L1], F, L2) :-
    dif(E,F),
    delete(L1, F, L3),
          insert(E, L3, L2).
```

```
delete2([],_,[]).
delete2([H|T],X,L2) :- X==H, delete2(T,X,L2),!.
delete2([H|T],X,[H|L2]) :- delete2(T,X,L2),!.

%test com a primeira regra (delete)
test_Delete(x) :-
   trace,write("[0,1,1,1,2] D 1 = "),delete([0,1,1,1,2],1,N),write(N),nl;
   trace,write("[0,2,1,0,3,7,0,5] D 0 = "),delete([0,2,1,0,3,7,0,5],0,N),write(N),nl;
   trace,write("[] D 2 = "),delete([],2,N),write(N),nl.

%test com a segunda regra (delete2)
test_Delete2(x) :-
   trace,write("[0,1,1,1,2] D 1 = "),delete2([0,1,1,1,2],1,N),write(N),nl;
   trace,write("[0,2,1,0,3,7,0,5] D 0 = "),delete2([0,2,1,0,3,7,0,5],0,N),write(N),nl;
   trace,write("[] D 2 = "),delete2([],2,N),write(N),nl.

% 26 - select(X,L1,L2) /* L2 resulta da eliminação de um X de L1 */
% Vai receber uma lista e eliminar o elemento X dela. Quando fizer a eliminação (ou não),
% vai retornar L2. Numa regra usa o dif para fazer a diferenciação dos elementos e o X e
noutra
% verifica se são iguais. Na que verifica a igualdade, faz o inverso da primeira regra
(select).
% Utiliza da regra insert no (select) e o selectAux no (select2).
%%-------------------------------------------------------%%
select(_,[],[]).
select(X, [X|L1], L1).
select(X, [E|L1], L2) :-
   dif(X, E),
   select(X, L1, L3),
         insert(E, L3, L2).

select2(X,[H|T],L2) :- X==H, selectAux(X,T,L2),!.
select2(X,[H|T],[H|L2]) :- select2(X,T,L2).

selectAux(_,[],[]).
selectAux(X,[H|T],[H|L2]) :- selectAux(X,T,L2).

%test com a primeira regra (select)
test_Select(x) :-
   trace,write("[0,1,1,1,2] S 1 = "),select(1,[0,1,1,1,2],N),write(N),nl;
   trace,write("[0,2,1,0,3,7,0,5] S 0 = "),select(0,[0,2,1,0,3,7,0,5],N),write(N),nl;
   trace,write("[] S 2 = "),select(2,[],N),write(N),nl.

%test com a segunda regra (select2)
test_Select2(x) :-
   trace,write("[0,1,1,1,2] S 1 = "),select2(1,[0,1,1,1,2],N),write(N),nl;
   trace,write("[0,2,1,0,3,7,0,5] S 0 = "),select2(0,[0,2,1,0,3,7,0,5],N),write(N),nl;
   trace,write("[] S 2 = "),select2(2,[],N),write(N),nl.
```

```
% 27 - insert(X,L1,L2) /* L2 resulta da inserção de X em L1 */
% insere o elemento X no inicio de L1. L2 vai ser L1 com a inserção de X.
%%-------------------------------------------------------%%
insert(X, L, [X|L]).

test_Insert(x) :-
   trace,write("1 I [0,1,1,1,2] = "),insert(1,[0,1,1,1,2],N),write(N),nl;
   trace,write("0 I [0,2,1,0,3,7,0,5] = "),insert(0,[0,2,1,0,3,7,0,5],N),write(N),nl;
   trace,write("2 I [] = "),insert(2,[],N),write(N),nl.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%29/33
questões%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%-------------------------------------------------------%%

%% regras auxiliares
%%-------------------------------------------------------%%

% concatenação de Listas
% Concatena L3 vai se a concatenação de L1 e L2.
% concatenar(L1,L2,L3).

concatenar([],L,L).
concatenar([H|T],L,[H|L2]) :- concatenar(T,L,L2).

%remover elemento
%Remove T vai ser a remoção de X na Lista passada,

remover(X,[X|T],T).
remover(X,[Y|C],[Y|D]) :- remover(X,C,D).

%ordenar uma lista
% L2 vai ser a ordenação de L1. ordenar(L1,L2).
% Utiliza inserir para fazer a inserção do maior ou menor elemento.
ordenar([],[]).
ordenar([H|T],LO) :- ordenar(T,TO), inserir(H,TO,LO).

inserir(E,[],[E]).
inserir(E,[H|T],[H|L]) :- E>=H, inserir(E,T,L).
inserir(E,[H|T],[E,H|T]) :- E<H.

% fim das regras auxiliares
%%-------------------------------------------------------%%

%union(L1,L2,L3) /* L3 resulta da união de L1 com L2 */
%ordenar L1 e L2, remove as repeticoes, concatena em uma só lista, e depois
%usa a regra unionAux para organizar a uniao(ordenar e tirar repeticoes).
union(L1,L2,L3) :- ordenar(L1,LO1), removerR(LO1,LP1),
```

```
    ordenar(L2,LO2), removerR(LO2,LP2), concatenar(LP1,LP2,LPR),
    unionAux(LPR,L3),!.

unionAux(LPR,L3):- ordenar(LPR,LPO), removerR(LPO,L3).

removerR([],[]).
removerR([H|[H2|T]],LR) :- H==H2, removerR([H2|T],LR).
removerR([Y|C],[Y|D]) :- removerR(C,D).

% membro da lista

membro(E,[E|_]).
membro(E,[_|T]) :- membro(E,T).

%intersection(L1,L2,L3) /* L3 resulta da intersecção de L1 com L2 */
% Ver se um elemento é membro de outra lista (utilizando membro), se for true, então
armazena em L3.
% Fez a ordenação e tirou as repeticoes. Apos isso, chamou a regra intersectionAux para
verificar os elementos.
% intersectionAux(L1,L2,L3), L1 e L2 são as listas ordenadas sem repeticoes, e L3 é a
intesecção entre eles.

intersection(L1,L2,L3) :- removerR(L1,L1R), removerR(L2,L2R),
    intersectionAux(L1R,L2R,LI), ordenar(LI,L3),!.

intersectionAux([],_,[]).
intersectionAux([H|T],L,[H|D]) :- membro(H,L), intersectionAux(T,L,D).
intersectionAux([_|T],L,LR) :- intersectionAux(T,L,LR).

%diference(L1,L2,L3) /* L3 resulta da diferença de L1 com L2 */
% Faz a ordenação de L1 e L2, e chama a regra deferenceAux para ver a diferenca entre os
conjuntos(listas).
% diferenceAux(L1,L2,L3), L1 e L2 são os conjuntos, e L3 é a diferenca entre eles.
diference(L1,L2,L3) :- removerR(L1,L1R), removerR(L2,L2R),
    diferenceAux(L1R,L2R,LD), ordenar(LD,L3).

diferenceAux([],_,[]).
diferenceAux([H|T],L,LD) :- membro(H,L), diferenceAux(T,L,LD),!.
diferenceAux([H|T],L,[H|D]) :- diferenceAux(T,L,D).

%equivalence(L1,L2) /* L1 é um conjunto equivalente a L2 */
% Faz a ordenação e retirada das repetições das listaspara simular conjuntos.
% Apos isso chama a regra equivalenceAux(L1,L2), onde vai verificar se eles são
equivalente sou não.

equivalence(L1,L2) :- ordenar(L1,L1O), removerR(L1O,L1R),
    ordenar(L2,L2O) ,removerR(L2O,L2R), equivalenceAux(L1R,L2R),!.
```

```
equivalenceAux([],[]).
equivalenceAux([H|T],L) :- membro(H,L), remover(H,L,LR), equivalence(T,LR).

%subset(L1,L2) /* L1 é um subconjunto de L2 */
% Faz a ordenação e retirada das repetições das listaspara simular conjuntos.
%  a regra subsetAux(L1,L2), onde vai verificar se L1 é um subconjunto de L2.
subset(L1,L2) :- ordenar(L1,L1O), removerR(L1O,L1R),
   ordenar(L2,L2O) ,removerR(L2O,L2R), subsetAux(L1R,L2R),!.

subsetAux([],_).
subsetAux([H|T],L) :-membro(H,L), remover(H,L,LS), subset(T,LS).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%TESTES-CONJUNTOS%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

test_Union(x) :-
trace,write("[7,4,3,5,3,1,7] U [0,1,9,9,2,4,0] =
"),union([7,4,3,5,3,1,7],[0,1,9,9,2,4,0],U),write(U),nl;
trace,write("[] U [5,4,2,1] = "),union([],[5,4,2,1],U),write(U),nl;
trace,write("[8,3,2,0] U [] = "),union([8,3,2,0],[],U),write(U),nl;
trace,write("[] U [] = "),union([],[],U),write(U),nl.

test_Intersection(x) :-
trace,write("[9,8,7,6,5,4] I [4,5,6,7,8,9] = "),intersection([9,8,7,6,5,4],[4,5,6,7,8,9],I),write(I),nl;
trace,write("[] I [2,4,8,10] = "), intersection([],[2,4,8,10],I), write(I),nl;
trace,write("[1,3,5,9] I [] = "), intersection([1,3,5,9],[],I), write(I),nl;
trace,write("[] I [] = "), intersection([],[],I), write(I),nl.

test_Diference(x) :-
trace,write("[9,7,5,3,1,0] D [0,3,7,9,2,8] = "),diference([9,7,5,3,1,0],[0,3,7,9,2,8],D),write(D),nl;
trace,write("[] D [8,5,7,3,0] = "),diference([],[8,5,7,3,0],D),write(D),nl;
trace,write("[1,6,3,9] D [] = "),diference([1,6,3,9],[],D),write(D),nl;
trace,write("[] D [] = "),diference([],[],D),write(D),nl.

test_Equivalence(x) :-
  trace,write("[1,2] == [2,2,1]:"),nl,equivalence([1,2],[2,2,1]);
  trace,write("[1,2,4,1] == [2,4,1]:"),nl,equivalence([1,2,4,1],[2,4,1]);
  trace,write("[4,5,6] == [4,5,6]:"),nl,equivalence([4,5,6],[4,5,6]);
  trace,write("[] == []"),nl,equivalence([],[]);
  trace,write("[5,1,3] == [5,2,3]:"),nl,equivalence([5,1,3],[5,2,3]).

test_Subset(x) :-
  trace,write("[0,2,1,2,1] S [0,1,3,2,0,3,2] = "), subset([0,2,1,2,1],[0,1,3,2,0,3,2]),nl;
  trace,write("[0,1,2] S [1,2,0,3] = "), subset([0,1,2],[1,2,0,3]),nl;
  trace,write("[] S [2,3,7,1] = "), subset([],[2,3,7,1]),nl;
  trace,write("[] S [] = "), subset([],[]),nl;
  trace,write("[9,6,3,10] S [] = "), subset([9,6,3,10],[]),nl.
```