Esercizi prolog

Si consiglia di utilizzare inoltre i libri consigliati e gli esercizi contenuti nel zip file fornito dal docente

Esercizio Prolog 1

Definire il predicato Prolog: `no_common_elements(-List1, -List2)`
che sia vero se sia List1 che List2 non hanno elementi in comune.

Soluzione:

```
%una possibile soluzione è rappresentata dal seguente listato prolog:

no_common_elements([], _).
no_common_elements([H1|L1], L2) :-
      \+ member(H1,L2),
      no_common_elements(L1, L2).

%si scorre la lista di sinistra e si controlla che nessun elemento sia
contenuto nella lista di destra


%o anche, più elegantemente:

no_common_elements(List1, List2) :-
      member(Element, List1),
      memberchk(Element, List2),
      !, fail.

no_common_elements (_,_).


%o ancora:


no_common_elements (List1, List2) :-
      \+(member(X, List1), member(X, List2)).
```

Esercizio Prolog 2

Scrivere un programma in Prolog che, data una lista di interi, stampi solo quelli pari.

- Sugg: usare solo i predicati standard per definire il predicato di "parità" di un numero. Se il compito risulta troppo difficile, avvalersi dell'operatore `mod`, che restituisce il resto di una divisione intera. Tale soluzione avrà una valutazione minore in quanto "meno elegante".
- Es:

```
?- A is 17 mod 5.
A = 2.
```

Soluzione:

```
%soluzione generale che riportiamo in due modi:

stampa_pari([]).
stampa_pari([Num|RestList]) :- dispari(Num),stampa_pari(RestList).
stampa_pari([Num|RestList]) :- writeln(Num),stampa_pari(RestList).

%come è possibile osservare, si scorre la lista banalmente con
stampa_pari, poi si controlla che il numero esaminato (in testa) sia
dispari con il predicato dispari/1. Se il test fallisce (il numero è
quindi pari), lo si stampa.

%Versione più compatta, usando l'operatore or (;), le due sono ovviamente
identiche:

stampa_pari([]).
stampa_pari([Num|RestList]) :-
      (
            dispari(Num)
            ;
            writeln(Num)
      ),
      stampa_pari(RestList).

%per il predicato dispari/1, forniamo diverse soluzioni possibili:

%predicato dispari/1 soluzione 1
%la più semplice, mod fornisce il resto della divisione intera
dispari(Num) :-
      1 is Num mod 2.

%predicato dispari/2 soluzione 2
%Utilizzo l'operatore di divisione intera
dispari2(Num) :-
      Div is Num // 2,
      1 is Num - (Div*2).

%predicato dispari/1 soluzione 3
%usa solamente l'operatore di sottrazione!.
dispari3(Num) :-
      dispari_aux(Num, disp),!.

dispari_aux(0,disp) :- fail.
dispari_aux(0,par).
dispari_aux(Num,disp) :- NewNum is Num - 1, dispari_aux(NewNum,par).
dispari_aux(Num,par) :- NewNum is Num - 1, dispari_aux(NewNum,disp).
```

Esercizio Prolog 3

Definire il predicato Prolog: `doppia(-List1, -List2)`
Che sia vero se List1 ha lunghezza doppia di List2.

Soluzione:

```
/*definizione del predicato lunghezza
definiamo le due clausole: la condizione di uscita (un fatto) stabilisce
che la lista vuota ha lunghezza 0. Il caso ricorsivo (una regola) dice
che
la lunghezza puo' essere calcolata sommando 1 alla coda della lista:*/
lunghezza([],0).
lunghezza([A|B],N):- lunghezza(B,N1), N is N1+1.

%definiamo il predicato listadoppia
doppia(X,Y):- lunghezza(X,A), lunghezza(Y,B), A is 2*B.

%Seconda soluzione (più elegante e semplice):
doppia([],[]).
doppia([_H11,_H12|RL1], [_H2|RL2]) :- doppia(RL1, RL2).
```

Esercizio Prolog 4

Definire il predicato Prolog `palindroma(-List)`
che sia vero se List è palindroma.

Suggerimento: se non si riesce a definirlo, considerare come scontato il predicato `reverse/2` che fornita una lista restituisce la sua inversa.

Soluzione:

```
%predicato pensato specificatamente per verificare se una lista è
palindroma
palindroma([]). %una lista vuota è palindroma
palindroma([_]).  %una lista con un solo elemento è palindroma.

palindroma([Head|List]) :-
      append(CenterList,[Head],List),
      palindroma(CenterList).
/*in quest'ultima clausola chiamo append con il secondo elemento ed il
terzo instanziati; sto effettivamente chiedendo se il resto della lista
(List) può unificare con una lista sconosciuta (CenterList) attaccata ad
un singolo elemento (l'ultimo) che sia identico alla testa della lista
originale.
Controllo successivamente se la parte centrale della lista è asua volta
palindroma*/



%soluzione dell'esercizio, utilizzando la reverse (meno efficientemente):
%banalmente…
palindroma2(List) :- reverse(List,List).

%definizione di reverse/2
reverse([],[]).
reverse([Head|Tail],RevList) :-
      reverse(Tail, PartReverse),
      append(PartReverse,[Head],RevList).
```

Nota1: è possibile calcolare in modo più efficiente la reverse, senza l'uso di append:

```
reverse(List,ReverseList) :- accRev(List,[],ReverseList).
accRev([H|T],A,R) :- accRev(T,[H|A],R).
accRev([],A,A).
```

Tale soluzione non è comunque di facile intuizione, e non è pertanto "attesa" nell'esercizio.

Esercizio Prolog 5

D. Avendo a disposizione il predicato atomic/1 che mi informa se il suo unico argomento è un atomo, definire il predicato prolog flatten/2, es: flatten (L,F)
Che sia vero se F è una lista di tutti gli elementi semplici, arbitrariamente annidati nella lista (di liste) L.

e.g. `flatten( [[[3,c],5,[4,[]]],[1,b],a] , [3,c,5,4,1,b,a] ).`

R_1. Una possibile soluzione è:

```
flatten( A, [A] ):- atomic( A ), A \= [].
flatten( [], [] ).
flatten( [H|T], P ):-
     flatten( H, M ),
     flatten( T, N ),
     append( M, N, P ) .

append([],L,L).
append([X|L1],L2,[X|L3]) :-
        append(L1,L2,L3).
```

R_2. Un'altra, più efficiente, è:

```
flatten(Xs,Ys):-flatten(Xs,[],Ys).
flatten([X|Xs],As,Ys):- flatten(Xs,As,As1), flatten(X,As1,Ys).
flatten(X,As,[X|As]) :- atomic(X), X\=[].
flatten([],Ys,Ys).
```

Esercizio Prolog 6

D. Definire il predicato Prolog:

```
fib(N,F).
```

che sia vero se F rappresenta l'N-esimo numero della sequenza di fibonacci.

Ricordiamo che la sequenza di Fibonacci è definita dalle seguenti:

$f(0) = 1$

$f(1) = 1$

$f(\ ) = f(N - 1) + f(N - 2)$

$\mathbf{R_1}$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fibonacci TopDown (come si è usi in prolog)
%                              ( è esponenziale )
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fib(0, 0).
fib(1, 1).
fib(N, F) :-
        N > 1,
        N1 is N - 1,
        N2 is N - 2,
        fib(N1, F1),
        fib(N2, F2),
        F is F1 + F2.
```

$\mathbf{R_2}$.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Fibonacci BottomUp (più efficiente: è lineare)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fibo_bu(N,F):-fibo_bu1(0,0,1,N,F).

fibo_bu1(N,F,_,N,F).
fibo_bu1(N1,F1,F2,N,F):-
  N1<N, N2 is N1+1, F3 is F1+F2,
  fibo_bu1(N2,F2,F3,N,F).
```

ovviamente la prima risposta viene comunque riconosciuta come valida e da diritto a punteggi pieno.