

Struttura base del predicato del “BAGOF”

```
Program x +
1 :- dynamic appoggio/1.
2
3 n(11).
4 n(2).
5 n(4).
6 n(5).
7 n(8).
8
9 appoggio([]).
10
11 numeri(L):-
12     n(Num),
13
14     appoggio(L),
15     write(L),nl,
16     append(L,[Num],LN),
17     retract(appoggio(L)),
18     assert(appoggio(LN)),
19
20     write(Num),nl,
21     fail.
22
23
24 numeri(L):-
25     appoggio(L),
26     retract(appoggio(L)),
27     assert(appoggio([])),
28     write(fine),nl.
```

Analisi del codice:

1. Fatti:

- **`n(11)`**.: Asserta il fatto che il numero 11 appartiene all'insieme.
- **`n(2)`**.: Asserta il fatto che il numero 2 appartiene all'insieme.
- **`n(4)`**.: Asserta il fatto che il numero 4 appartiene all'insieme.
- **`n(5)`**.: Asserta il fatto che il numero 5 appartiene all'insieme.
- **`n(8)`**.: Asserta il fatto che il numero 8 appartiene all'insieme.
- **`appoggio([])`**.: Definisce un caso base per il predicato **`appoggio`**. Un elenco vuoto indica che l'insieme è vuoto.

2. Predicati:

- **`numeri(L)`**: Questo predicato genera tutti gli elementi dell'insieme definito dai fatti **`n(Num)`**.
- **`appoggio(L)`**: Questo predicato verifica se un elemento appartiene all'insieme definito dai fatti **`n(Num)`**.

Spiegazione del predicato `numeri`:

Il predicato **`numeri`** utilizza la ricorsione e la backtracking per generare tutti gli elementi dell'insieme. Il predicato funziona come segue:

1. Estrazione elemento: Il predicato **`n(Num)`** estrae un elemento **`Num`** dall'insieme definito dai fatti **`n(Num)`**.
2. Verifica appartenenza: Si verifica se l'elemento **`Num`** appartiene all'insieme utilizzando il predicato **`appoggio(L)`**.
3. Aggiunta elemento: Se **`appoggio(L)`** conferma che **`Num`** appartiene all'insieme, l'elemento viene aggiunto all'elenco **`L`** utilizzando **`append(L, [Num], LN)`**.
4. Aggiornamento supporto: Si rimuove l'elenco **`L`** dal predicato **`appoggio`** utilizzando **`retract(appoggio(L))`**.
5. Asserzione nuovo supporto: Si asserisce un nuovo elenco **`LN`** che include l'elemento **`Num`** utilizzando **`assert(appoggio(LN))`**.
6. Stampa elemento: Si stampa l'elemento **`Num`** sulla console utilizzando **`write(Num),nl`**.

7. Backtracking: Il predicato **fail** viene utilizzato per forzare il backtracking, permettendo al predicato di esplorare altre soluzioni.
8. Caso base: Se il predicato **appoggio(L)** verifica che l'insieme è vuoto (cioè **L** è vuoto), il predicato **numeri** stampa "fine" e termina.

Comportamento del predicato:

Quando si esegue il predicato **numeri(L)**, esso genera tutti gli elementi dell'insieme uno alla volta, stampandoli sulla console. La stampa di "fine" indica che la generazione è terminata.

Esempio di esecuzione:


Se si esegue il codice in Prolog, l'output sarà:


```
[]  
11  
[11]  
2  
[11, 2]  
4  
[11, 2, 4]  
5  
[11, 2, 4, 5]  
8  
fine  
L = [11, 2, 4, 5, 8]
```

Note:

- Questo codice implementa una versione semplificata del predicato Bagof. Non gestisce le variabili e la molteplicità.
- Il predicato **appoggio/1** è un predicato dinamico, il che significa che la sua definizione può cambiare durante l'esecuzione del programma.
- Il predicato **fail** viene utilizzato per forzare il backtracking, permettendo al predicato di esplorare diverse soluzioni.

query SWISH_


```
 numeri(L).  
[]  
11  
[11]  
2  
[11, 2]  
4  
[11, 2, 4]  
5  
[11, 2, 4, 5]  
8  
fine  
L = [11, 2, 4, 5, 8]  
?- numeri(L).
```

 `numeri(L), numeri(L1).`


```
[]  
11  
[11]  
2  
[11, 2]  
4  
[11, 2, 4]  
5  
[11, 2, 4, 5]  
8  
fine  
[]  
11  
[11]  
2  
[11, 2]  
4  
[11, 2, 4]  
5  
[11, 2, 4, 5]  
8  
fine  
L = L1, L1 = [11, 2, 4, 5, 8]
```

?- `numeri(L), numeri(L1).`

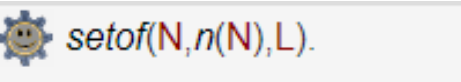
PREDICATO BAGOF E SETOF



```
1  
2 n(11).  
3 n(2).  
4 n(4).  
5 n(5).  
6 n(8).  
7 n(4).  
8 n(8).  
9
```



```
bagof(N,n(N),L).  
L = [11, 2, 4, 5, 8, 4, 8]  
?- bagof(N,n(N),L).
```



```
setof(N,n(N),L).  
L = [2, 4, 5, 8, 11]  
?- setof(N,n(N),L).
```

1. bagof:

- **bagof** esegue un'iterazione su tutti i fatti che corrispondono a **n(N)**. Per ogni fatto corrispondente, il valore di **N** viene aggiunto all'elenco **L**.
- Poiché ci sono fatti duplicati per 4 e 8, **bagof** includerà quei duplicati nell'elenco risultante **L**.

2. setof:

- **setof** si comporta in modo simile a **bagof** ma con due differenze fondamentali:
 - **Unicità:** **setof** garantisce che solo elementi univoci vengano aggiunti all'elenco **L**. In questo caso, anche se ci sono fatti duplicati per 4 e 8, **setof** li aggiungerà solo una volta all'elenco.
 - **Ordinamento:** **setof** ordina l'elenco risultante **L** in ordine ascendente.

Esecuzione:

L'esecuzione di questo codice produrrà probabilmente due output diversi a seconda dell'implementazione Prolog:

Output bagof:

`L = [11, 2, 4, 5, 8, 4, 8]` // L'ordine può variare, ma i duplicati saranno presenti

•

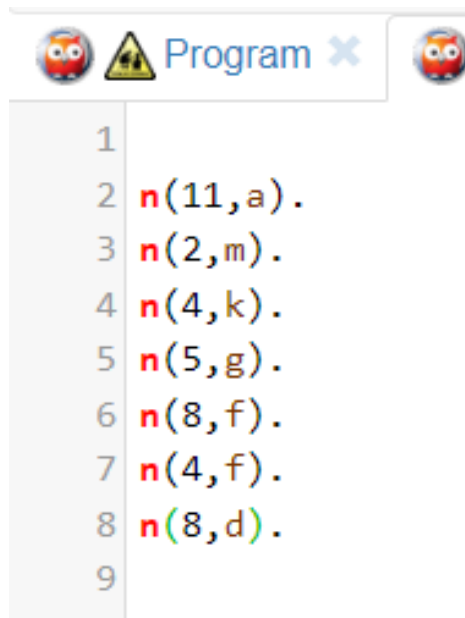
Output setof:

`L = [2, 4, 5, 8, 11]` // Gli elementi sono univoci e ordinati in ordine ascendente


•

In sintesi:

- **bagof** raccoglie tutti gli elementi che soddisfano l'obiettivo, inclusi i duplicati.
- **setof** raccoglie solo elementi univoci che soddisfano l'obiettivo e li ordina.



```
1  
2 n(11,a).  
3 n(2,m).  
4 n(4,k).  
5 n(5,g).  
6 n(8,f).  
7 n(4,f).  
8 n(8,d).  
9
```

 `bagof(L,n(N,L),Lista).`

Lista = [m],

N = 2

Lista = [k, f],

N = 4

Lista = [g],

N = 5

Lista = [f, d],

N = 8


Lista = [a],

N = 11

?- `bagof(L,n(N,L),Lista).`


Restituisce per ogni lettera il corrispondente numerico

```
1
2 n(11,a).
3 n(2,m).
4 n(4,k).
5 n(5,g).
6 n(8,f).
7 n(4,f).
8 n(8,d).
9
10
11 l(L):-
12     n(_,L).
13 num(L):-
14     n(L,_).
```

 bagof(N,L^n(N,L),Lista).

Lista = [11, 2, 4, 5, 8, 4, 8]

?- bagof(N,L^n(N,L),**Lista**).

 L=k, bagof(N,L^n(N,L),Lista).

L = k,

Lista = [4]

?- L=k, bagof(N,L^n(N,L),**Lista**).

Spiegazione del codice Prolog

Questo codice Prolog definisce due predicati: `l/1` e `num/1`. I predicati utilizzano i fatti definiti con `n/2` per generare liste di elementi.

Fatti:

- `n(Elemento, Carattere)`: Questi fatti definiscono un insieme di coppie elemento-carattere. Ad esempio, `n(11, a)` indica che l'elemento 11 è associato al carattere "a".

Predicati:

- `l(L)`:
 - Questo predicato genera una lista `L` contenente tutti i **caratteri** presenti nei fatti `n(Elemento, Carattere)`.
 - `n(_, L)`: In questa espressione, la prima variabile (`Elemento`) viene ignorata utilizzando l'underscore (`_`). Il predicato controlla solo il secondo elemento (il carattere) che viene unificato con la variabile `L`.
- `num(L)`:
 - Questo predicato genera una lista `L` contenente tutti gli **elementi numerici** presenti nei fatti `n(Elemento, Carattere)`.
 - `n(L, _)`: In questa espressione, la seconda variabile (il carattere) viene ignorata utilizzando l'underscore (`_`). Il predicato controlla solo il primo elemento (l'elemento numerico) che viene unificato con la variabile `L`.

Esempio di esecuzione:

Se si esegue questo codice, i risultati potrebbero essere:

?- l(L).

L = [a, m, k, g, f, d]

?- num(L).

L = [11, 2, 4, 5, 8, 4, 8]

```

1
2 n(11,a,la).
3 n(2,m,kl).
4 n(4,k,jd).
5 n(5,g,lv).
6 n(8,f,ma).
7 n(4,f,gh).
8 n(8,d,lk).
9
10
11 l(L):-
12     n(_,L).
13 num(L):-
14     n(L,_).

```

 `bagof(N,(L,B)^n(N,L,B),Lista).`

Lista = [11, 2, 4, 5, 8, 4, 8]

?- `bagof(N,(L,B)^n(N,L,B),Lista).`

```
1
2 n(11,a,la).
3 n(2,m,kl).
4 n(4,k,jd).
5 n(5,g,lv).
6 n(8,f,ma).
7 n(4,f,gh).
8 n(8,d,lk).
9
10
11
12 numero_lettere_diverse(Num):-
13     setof(L,(N,B)^n(N,L,B),Lista),
14     length(Lista,Num).
```



numero_lettere_diverse(N).

N = 6

?-

numero_lettere_diverse(N).

PROBLEMA DELLE 8 REGINE



Program x



Program x



```
1 controlloriga([A,B],[A,C]).
2
3 controllocolonna([A,B],[C,A]).
4
5 controllodiagonale([A,B],[C,D]):-
6     X is A-C,
7     Y is B-D,
8     X =\= Y,
9     X =\= -Y.
10
11 controllocoppia([A]).
12
13 controllocoppia([A,B|T]):-
14     controlloriga(A,B),
15     controllocolonna(A,B),
16     controllodiagonale(A,B),
17     controllocoppia([A|T]).
18
19 controllosoluzione([A]).
20
21 controllosoluzione([H|T]):-
22     \+controllocoppia([H|T]),
23     controllosoluzione([T]).
```

true

Next

10

100

1,000

Stop

?-

```
controllosoluzione([[1,3],[3,4],[4,2],[2,1]])
```

Spiegazione del codice per il problema delle 8 regine

Il codice fornito implementa un algoritmo di backtracking per risolvere il problema delle 8 regine. L'obiettivo è posizionare 8 regine su una scacchiera 8x8 in modo che nessuna di esse possa minacciarne un'altra. Il codice utilizza una rappresentazione per righe e colonne, dove ogni elemento della lista rappresenta la posizione di una regina nella riga corrispondente.

Analisi del codice:

1. Predicati di controllo:

- `controlloriga(R1, R2)`: Verifica se due regine in righe diverse (R1 e R2) si attaccano.
- `controllocolonna(C1, C2)`: Verifica se due regine in colonne diverse (C1 e C2) si attaccano.
- `controllodiagonale(P1, P2)`: Verifica se due regine su diagonali diverse (P1 e P2) si attaccano.
- `controllocoppia([Regina])`: Controlla se una singola regina è minacciata da altre regine.
- `controllocoppia([Regina1, Regina2|T])`: Controlla se due regine (Regina1 e Regina2) e la lista successiva di regine sono posizionate in modo valido (nessun attacco).

2. Predicato di soluzione:

- `controllosoluzione([Regina])`: Controlla se la lista di regine rappresenta una soluzione valida (nessun attacco) per il problema delle 8 regine.

Funzionamento dell'algoritmo:

1. Il predicato `controllosoluzione` avvia la ricorsione, verificando se la lista di regine rappresenta una soluzione valida.
2. Se la lista è vuota (una regina), il predicato controlla se la regina è minacciata da altre regine già posizionate. Se non lo è, la soluzione è parziale.

3. Se la lista contiene ancora regine, il predicato **controllocoppia** viene utilizzato per verificare se l'ultima regina (testa della lista) e la soluzione parziale (coda della lista) sono valide.
 - Se la soluzione parziale è valida, il predicato si ripete ricorsivamente con la regina successiva aggiunta alla coda della lista.
 - Se la soluzione parziale non è valida, si esplorano alternative posizionando la regina in una colonna diversa nella stessa riga.
4. La ricorsione continua fino a quando non vengono trovate tutte le soluzioni o non ci sono più configurazioni valide da esplorare.

Esempio di esecuzione:

Supponiamo di voler posizionare la prima regina nella riga 1. Il predicato **controllosoluzione** verifica se la regina è minacciata da altre regine (nessuna in questo caso).

Proseguiamo posizionando la seconda regina. Il predicato **controllocoppia** controlla se la seconda regina nella riga 2 e la prima regina nella riga 1 si attaccano (non si attaccano).

L'algoritmo continua in questo modo, posizionando una regina alla volta e verificando la validità della soluzione parziale fino a trovare tutte le 92 soluzioni possibili per il problema delle 8 regine.

VISITA DFS

```
3
4  /* DFS */
5  edge(a,b).
6  edge(b,c).
7  edge(c,d).
8  edge(a,e).
9  edge(e,f).
10 edge(f,k).
11 edge(f,c).
12
13 path(a,m).
14
15 path(X,Y,[X,Y]) :-
16     edge(X,Y).
17
18 path(X,Y,[X|P_Z_Y]) :-
19     edge(X,Z),
20     path(Z,Y,P_Z_Y).
21
```

Struttura del codice:

1. Fatti:

- Definiscono le connessioni tra i nodi del grafo usando predicati `arco(nodo1, nodo2)`.

2. Regole:

- `percorso(nodo_iniziale, nodo_finale)`: Clausola obiettivo che specifica la ricerca del percorso.
- `percorso(nodo_corrente, nodo_finale, [nodo_corrente])`: Regola base che indica che il percorso da un nodo a se stesso è semplicemente il nodo stesso.

- `percorso(nodo_corrente, nodo_finale, [nodo_corrente|percorso_restante])`: Regola ricorsiva che esplora i nodi vicini.

Algoritmo DFS:

1. Inizializzazione:

- Impostare il nodo corrente `X` su `nodo_iniziale`.
- Impostare il nodo finale `Y` su `nodo_finale`.
- Creare un insieme vuoto `visitato` per tenere traccia dei nodi già visitati.

2. Ricerca ricorsiva:

- Se `X` è uguale a `Y`, il percorso è stato trovato: restituire la lista `[X]`.
- Se `X` è già presente in `visitato`, significa che è stato visitato in precedenza: tornare indietro (esplorare un altro ramo).
- Aggiungere `X` a `visitato` per evitare di rivisitarlo.
- Ottenere i vicini di `X`.
- Per ogni vicino `Z`:
 - Chiamare ricorsivamente `percorso(Z, Y, percorso_restante)` per trovare il percorso da `Z` a `Y`.
 - Se la chiamata ricorsiva ha successo (restituisce un percorso), prependere `X` al percorso e restituirlo.
- Se nessun vicino ha un percorso verso `Y`, il percorso non esiste: restituire `None`.

VISITA BFS

1° parte (controllo delle frontiere)

```
23 /* BFS */
24
25 /* Pf(F,PF). */
26
27 prossimafrontiera([], []).
28
29 prossimafrontiera([X|R], F):-
30     setof(Z,edge(X,Z),RZ), %Lista di nodi raggiungibili
31     prossimafrontiera(R,FF),
32     append(RX,RF,F).|
```

Obiettivo:

Implementare l'algoritmo di ricerca in ampiezza (BFS) per esplorare un grafo rappresentato da spigoli e trovare il percorso più breve dal nodo iniziale al nodo finale.
Spiegazione:

1. Predicati:

- **prossimafrontiera(Frontiera,ProssimaFrontiera)**: Questo predicato rappresenta la parte principale dell'algoritmo BFS. La prima variabile, **Frontiera**, rappresenta una lista di nodi da esplorare. La seconda variabile, **ProssimaFrontiera**, rappresenta la prossima frontiera da considerare, ovvero la lista dei nodi raggiungibili dai nodi della frontiera attuale.

2. Regole:

- **Regola base:**
 - **prossimafrontiera([], [])**. Questa regola indica che se la frontiera è vuota (nessun nodo da esplorare), anche la prossima frontiera è vuota.

- Regola ricorsiva:
 - `prossimafrontiera([X|R], F):- setof(Z,edge(X,Z),RZ),
prossimafrontiera(R,FF), append(RX,RF,F)`. Questa regola gestisce la parte principale dell'esplorazione del grafo. Essa scompone in passi:
 1. `setof(Z,edge(X,Z),RZ)`: Questa clausola utilizza il predicato `setof` per generare un insieme `RZ` contenente tutti i nodi raggiungibili dal nodo corrente `X`. I nodi raggiungibili sono quelli collegati da uno spigolo a `X`.
 2. `prossimafrontiera(R,FF)`: Si chiama ricorsivamente `prossimafrontiera` sulla restante parte della frontiera `R` per ottenere la prossima frontiera da esplorare.
 3. `append(RX,RF,F)`: Si combina la lista di nodi raggiungibili `RZ` (rinominata `RX`) con la prossima frontiera ottenuta dalla chiamata ricorsiva `FF` (rinominata `RF`) per formare la prossima frontiera complessiva `F`.

Schema dell'algoritmo BFS:

1. Inizializzazione:

- Impostare la frontiera iniziale con il nodo di partenza.
- Impostare la prossima frontiera vuota.

2. Ciclo di esplorazione:

- Finché la frontiera non è vuota:
 - Estrarre il primo nodo `X` dalla frontiera.
 - Trovare tutti i nodi raggiungibili da `X` e inserirli nella lista `RZ`.
 - Combinare `RZ` con la prossima frontiera ottenuta dalla chiamata ricorsiva per formare la nuova prossima frontiera `F`.
 - Aggiornare la frontiera con la restante parte della frontiera iniziale (esclusa `X`).

3. Risultato:

- Se il nodo finale è stato raggiunto durante l'esplorazione, il percorso più breve è contenuto nella frontiera o nelle frontiere precedenti.
- Se il nodo finale non è stato raggiunto, significa che non esiste un percorso dal nodo di partenza al nodo finale.

2° parte (Verifica del path attraverso le frontiere)

```
23 /* BFS */
24
25 /* Pf(F,PF). */
26
27 prossimafrontiera([], []).
28
29 prossimafrontiera([[X,PX]|R], F):-
30     setof([Z|[X|PX]],edge(X,Z),RZ), %Lista di nodi raggiungibili
31     prossimafrontiera(R,FF),
32     append(RX,RF,F).
33
34 opf(F,FR,Y):-
35     prossimafrontiera(F,FR),
36     member(Y,FR).
37 opf(F,FR,Y):-
38     prossimafrontiera(F,FRZ),
39     opf(FRZ,FR,Y).
40
41 path(X,Y,P):-
42     prossimafrontiera([X|R],F),
43     member(Y,F),
44     reverse(F,P).
45     opf([X],FF,Y).
46
```

Obiettivo:

Implementare un algoritmo di ricerca in ampiezza (BFS) modificato per memorizzare i percorsi completi dai nodi di partenza a tutti i nodi raggiungibili nel grafo.

Spiegazione:

1. Predicati:

- **prossimafrontiera(Frontiera, ProssimaFrontiera)**: Come nella versione precedente, questo predicato gestisce l'esplorazione del grafo utilizzando l'algoritmo BFS, ma con una modifica importante. In questa versione, ogni nodo nella frontiera è rappresentato come una lista **[Nodo, Percorso]**, dove **Nodo** è il nodo corrente e **Percorso** è la lista che rappresenta il percorso finora raggiunto da **X** (il nodo iniziale).
- **opf(Frontiera, FrontiereFinali, NodoFinale)**: Questo nuovo predicato, "OPF" (Open Path Finder), serve a trovare tutti i percorsi completi dai nodi di partenza a tutti i nodi raggiungibili nel grafo.
- **path(NodoIniziale, NodoFinale, Percorso)**: Questo predicato rimane lo stesso della versione precedente, trovando il percorso più breve dal nodo iniziale al nodo finale.

2. Regole:

- **Regola di prossima frontiera modificata:**
 - **prossimafrontiera([[X,PX] | R], F)**: Questa regola è simile alla versione precedente, ma invece di memorizzare solo i nodi raggiungibili, memorizza anche i percorsi finora raggiunti. Il percorso per un nodo **Z** raggiungibile da **X** è ottenuto prependendo **Z** al percorso **PX** di **X**.
- **Regole OPF:**
 - **opf(F, FR, Y)**: Se il nodo finale **Y** è presente nella frontiera **F** ottenuta dall'esplorazione BFS, significa che è stato raggiunto un percorso completo da **X** a **Y**. La lista **F** contiene tutti i percorsi completi, quindi **FR** viene impostata su **F**.
 - **opf(F, FRZ), opf(FRZ, FR, Y)**: Se il nodo finale **Y** non è presente nella frontiera **F**, si chiama ricorsivamente **opf** sulla frontiera successiva **FRZ** ottenuta dall'esplorazione BFS. Se il nodo finale viene trovato in una frontiera successiva, il percorso viene propagato indietro fino alla lista **FR**.
- **Regola path**: Rimane invariata rispetto alla versione precedente.

Schema dell'algoritmo BFS con memorizzazione dei percorsi:

1. Inizializzazione:

- Impostare la frontiera iniziale con il nodo di partenza e un percorso vuoto **[]**.
- Impostare le frontiere finali vuote.

2. Ciclo di esplorazione:

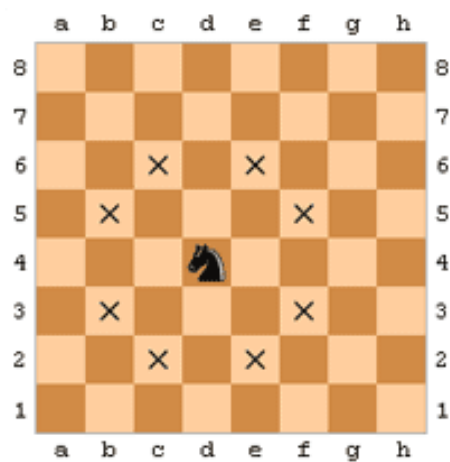
- Finché la frontiera non è vuota:

- Estrarre il primo nodo $[X, PX]$ dalla frontiera.
- Trovare tutti i nodi raggiungibili da X e costruire i percorsi completi per essi prependendo Z a PX .
- Combinare i nuovi percorsi con la prossima frontiera ottenuta dalla chiamata ricorsiva per formare la nuova prossima frontiera F .
- Aggiornare le frontiere finali con i percorsi completi trovati in F .
- Aggiornare la frontiera con la restante parte della frontiera iniziale (esclusa $[X, PX]$).

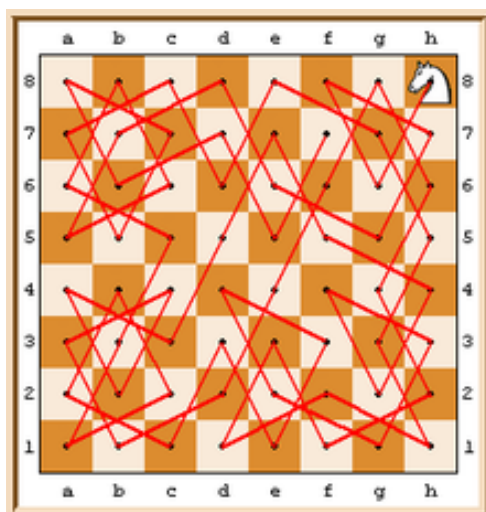
3. Risultato:

- Le frontiere finali FR contengono tutti i percorsi completi dai nodi di partenza a tutti i nodi raggiungibili nel grafo. Ogni percorso è rappresentato come una lista $[Nodo, Percorso]$, dove $Nodo$ è il nodo finale e $Percorso$ è la lista di nodi attraversati per raggiungerlo.
- Se il nodo finale Y non è raggiungibile da nessun nodo di partenza, la lista FR rimane vuota.

IL PROBLEMA DEL PERCORSO DEL CAVALLO



Voglio far occupare dal cavallo tutte le caselle possibili passandoci solo una volta.



1	42	13	28	3	44	15	30
24	27	2	43	14	29	4	45
41	12	25	60	53	64	31	16
26	23	52	63	56	59	46	5
11	40	61	58	51	54	17	32
22	37	50	55	62	57	6	47
39	10	35	20	49	8	33	18
36	21	38	9	34	19	48	7


```

1 controllo_unicita(_, []).
2
3 controllo_unicita(A/B, [A/C|T]):-
4     B \= C,
5     controllo_unicita(A/B,T).
6
7 controllo_unicita(A/B, [_/B|T]):-
8     A \= C,
9     controllo_unicita(A/B,T).
10
11 controllo_unicita(A/B, [_/_|T]):-
12     A \= C,
13     B \= D,
14     controllo_unicita(A/B,T).
15
16 controllo_unicita_totale([_]).
17 controllo_unicita_totale([M|T]):-
18     controllo_unicita(M,T),
19     controllo_unicita_totale(T).
20
21 mossa_valida(A/B,C/D):-
22     abs(A-C, 1),
23     abs(B-D, 2).
24
25 mossa_valida(A/B,C/D):-
26     abs(A-C, 2),
27     abs(B-D, 1).
28
29 controllo_mosse([_]).
30
31 controllo_mosse([M1, M2|T]):-
32     mossa_valida(M1,M2),
33     controllo_mosse([M2|T]).
34
35 controllo_soluzione(L):-
36     controllo_unicita_totale(L),
37     controllo_mosse(L).
38
39

```

1. Controllo unicità

Il primo algoritmo, `controllo_unicità`, verifica se una lista contiene elementi distinti. Funziona in modo ricorsivo:

- `controllo_unicità(A/B, [])` restituisce `true` se la lista è vuota.
- `controllo_unicità(A/B, [A/C|T])` verifica se `B` è diverso da `C` e richiama ricorsivamente `controllo_unicità` sulla coda della lista `T`.
- `controllo_unicità(A/B, [_/B|T])` verifica se `A` è diverso da `C` e richiama ricorsivamente `controllo_unicità` sulla coda della lista `T`.
- `controllo_unicità(A/B, [_/_|T])` verifica se `A` è diverso da `C` e `B` è diverso da `D` e richiama ricorsivamente `controllo_unicità` sulla coda della lista `T`.

2. Controllo soluzione

Il secondo algoritmo, `controllo_soluzione`, verifica se una lista rappresenta un percorso valido del Cavallo:

- `controllo_unicità_totale([_])` restituisce `true` se la lista contiene un solo elemento.
- `controllo_unicità_totale([M|T])` verifica se `M` è un elemento distinto dagli altri nella lista usando `controllo_unicità` e richiama ricorsivamente `controllo_unicità_totale` sulla coda della lista `T`.
- `mossa_valida(A/B, C/D)` verifica se la mossa da `(A,B)` a `(C,D)` è valida secondo le regole del Cavallo (spostamento di una casella in orizzontale e due in verticale o viceversa).
- `controllo_mosse([_])` restituisce `true` se la lista contiene un solo elemento.
- `controllo_mosse([M1, M2|T])` verifica se `M2` è una mossa valida rispetto a `M1` usando `mossa_valida` e richiama ricorsivamente `controllo_mosse` sulla coda della lista `T`.
- `controllo_soluzione(L)` verifica se la lista `L` rappresenta un percorso valido del Cavallo usando `controllo_unicità_totale` e `controllo_mosse`.

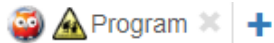
Definizione in italiano

In sintesi, il codice definisce due funzioni:

1. `controllo_unicità(lista)`: verifica se la lista contiene elementi distinti.
2. `controllo_soluzione(lista)`: verifica se la lista rappresenta un percorso valido del Cavallo, controllando l'unicità delle caselle visitate e la validità delle mosse tra caselle consecutive.

Queste funzioni possono essere utilizzate per risolvere il problema del Cavallo, trovando un percorso che visiti tutte le caselle della scacchiera esattamente una volta.

GRAMMATICA IN PROLOG



```
1 % data una grammatica dire a quali stringhe appartengono al linguaggio
2
3 % data la stringa baa dire se appartiene al linguaggio
4
5 'A'(L,R1):-
6     'B'(L,R),
7     'C'(R,R1).
8 'A'(L,R2):-
9     'C'(L,R),
10    'B'(R,R1),
11    'B'(R1,R2).
12
13 /*Scrivi la seguenti query (?- listing.) e sostituisci le righe 15 e 16
14 'B'-->'a'.
15 'A'-->'b'.
16 */
17
18 %Sostituisci con :
19
20 'B'([H|T],T):-|
21     H = 'a'.
22 'C'([H|T],T):-
23     H = 'b'.
```

Grammatica:

La grammatica definisce le regole per la generazione di stringhe valide nel linguaggio. In questo caso, la grammatica è composta da due regole:

1. **A(L, R1)**: questa regola dice che una stringa **A** può essere generata da una stringa **B** seguita da una stringa **C**. I parametri **L** e **R1** rappresentano le liste di simboli che compongono le stringhe rispettivamente **B** e **C**.
2. **A(L, R2)**: questa regola è simile alla prima, ma invertita. Dice che una stringa **A** può essere generata da una stringa **C** seguita da due stringhe **B**. I parametri **L**, **R** e **R1** hanno lo stesso significato di prima.

Sostituzione di righe:

Le righe 14 e 15 del codice originale definiscono le regole per i simboli **B** e **C**. Queste regole vengono sostituite con le seguenti:

1. $B([H|T], T)$: questa regola dice che un simbolo **B** può essere generato da un singolo simbolo **a** seguito da una lista **T**. In altre parole, **B** è composto solo dal simbolo **a** e da una coda di simboli **T**.
2. $C([H|T], T)$: questa regola dice che un simbolo **C** può essere generato da un singolo simbolo **b** seguito da una lista **T**. In altre parole, **C** è composto solo dal simbolo **b** e da una coda di simboli **T**.

Spiegazione passo passo:

1. Definizione dei predicati:

- $A(L, R)$: questo predicato rappresenta una stringa **A** composta da una lista di simboli **L** seguita da una lista di simboli **R**.
- $B(L, R)$: questo predicato rappresenta una stringa **B** composta da una lista di simboli **L** seguita da una lista di simboli **R**.
- $C(L, R)$: questo predicato rappresenta una stringa **C** composta da una lista di simboli **L** seguita da una lista di simboli **R**.

2. Regole di generazione:


- La prima regola $A(L, R1)$ dice che una stringa **A** può essere generata da una stringa **B** seguita da una stringa **C**. In altre parole, la stringa **A** è composta dalla concatenazione di una stringa **B** e una stringa **C**.
- La seconda regola $A(L, R2)$ dice che una stringa **A** può essere generata da una stringa **C** seguita da due stringhe **B**. In altre parole, la stringa **A** è composta dalla concatenazione di una stringa **C**, una stringa **B** e un'altra stringa **B**.

3. Regole di composizione:

- La regola $B([H|T], T)$ dice che un simbolo **B** può essere generato da un singolo simbolo **a** seguito da una lista di simboli **T**. In altre parole, il simbolo **B** è composto dal simbolo **a** e da una coda di simboli **T**.
- La regola $C([H|T], T)$ dice che un simbolo **C** può essere generato da un singolo simbolo **b** seguito da una lista di simboli **T**. In altre parole, il simbolo **C** è composto dal simbolo **b** e da una coda di simboli **T**.

Query:

- La query **?- listing.** serve a stampare tutte le stringhe generate dalla grammatica. In altre parole, elenca tutte le possibili combinazioni di simboli che soddisfano le regole della grammatica.

 listing.

```
'C'([H|T], T) :-
    H=b.

'A'(L, R1) :-
    'B'(L, R),
    'C'(R, R1).
'A'(L, R2) :-
    'C'(L, R),
    'B'(R, R1),
    'B'(R1, R2).

writeq(A) :-
    pengine_writeq(A).

read(A) :-
    pengine_read(A).

write_canonical(A) :-
    pengine_write_canonical(A).

read_line_to_string(A, B) :-
    pengine_read_line_to_string(A, B).

format(A, B) :-
    pengine_format(A, B).

read_line_to_codes(A, B) :-
    pengine_read_line_to_codes(A, B).

tab(A) :-
    pengine_tab(A).

listing :-
    pengine_listing.

listing(A) :-
    pengine_listing(A).

'B'([H|T], T) :-
    H=a.

:- dynamic screen_property/1.

screen_property(height(568.391)).
screen_property(width(927)).
screen_property(rows(37)).
screen_property(cols(130)).

writeln(A) :-
    pengine_writeln(A).

portray_clause(A) :-
    pengine_portray_clause(A).

display(A) :-
    pengine_display(A).

write_term(A, B) :-
    pengine_write_term(A, B).

write(A) :-
    pengine_write(A).

print(A) :-
    pengine_print(A).

nl :-
    pengine_nl.

flush_output :-
    pengine_flush_output.

format(A) :-
    pengine_format(A).
```

true1

Appartenenza di una stringa:

 'A'([b,a,a],[,]).	  
true	1
 'A'([b,a,b],[,]).	  
false	
 'A'([b,a,a],[,]).	  
true	1

SPLIT

```
Program x +
1 split([],[],[]).
2
3 split(L,LS,LD):-
4     append(LS,LD,L),
5     LS=[_|_],
6     LD=[_|_].
7
8 operazione(X,Y,X+Y).
9 operazione(X,Y,X-Y).
10 operazione(X,Y,X*Y).
11 operazione(X,Y,X/Y):-
12     Y \= 0.
13
14 termine([X],X).
15
16 termine(L, T):-
17     split(L,LS,LD),
18     termine(LS,TS),
19     termine(LD,TD),
20     operazione(TS,TD,T).
21
22 equazione(L):-
23     split(L,LS,LD),
24     termine(LS,TS),
25     termine(LD,TD),
26     TS := TD.
```

1. Funzioni di base:

- **append(Lista1, Lista2, ListaConcatenata)**: Concatena due liste **List**1 e **List**2 in **List**Concatenata.
- **operazione(X, Y, Risultato)**: Esegue un'operazione aritmetica su due numeri **X** e **Y** e memorizza il risultato in **Risultato**. Supporta addizione (+), sottrazione (-), moltiplicazione (*) e divisione (/) controllando che il divisore (**Y**) sia diverso da zero.

2. Funzioni per termini:

- **termine([Elemento], Elemento)**: Se la lista contiene un singolo elemento, considera quell'elemento come il termine (risultato finale).
- **termine(Lista, Termine)**: Questa funzione è ricorsiva e divide la lista in due sottoliste con **split**. Poi, richiama ricorsivamente **termine** su entrambe le sottoliste ottenendo i termini **TS** e **TD**. Infine, applica l'operazione (**operazione**) su **TS** e **TD** e memorizza il risultato in **Termine**.

3. Funzione equazione:



- **equazione(Lista)**: Controlla se la lista rappresenta un'equazione valida.
 - **split(Lista, SottoListaSinistra, SottoListaDestra)**: Divide la lista in due sottoliste che rappresentano i lati sinistro e destro dell'equazione.
 - **termine(SottoListaSinistra, TermineSinistro)**: Calcola il termine della sottolista sinistra.
 - **termine(SottoListaDestra, TermineDestra)**: Calcola il termine della sottolista destra.
 - **TermineSinistro ::= TermineDestra**: Confronta i termini ottenuti per verificare se i lati sinistro e destro dell'equazione sono uguali.

Schema step-by-step:

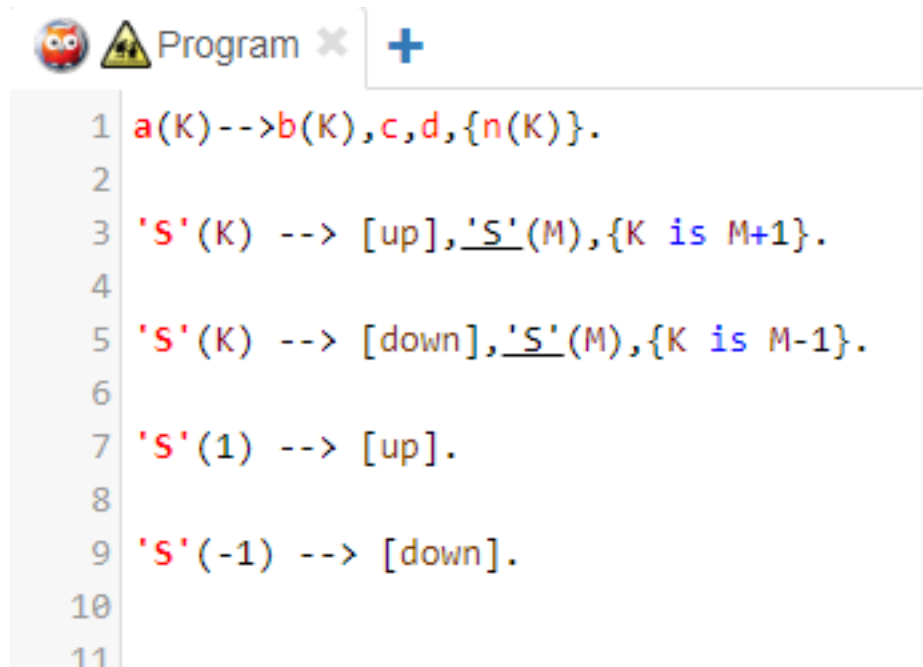
1. L'utente inserisce una lista che rappresenta l'espressione o l'equazione.
2. **equazione(Lista)** viene chiamata.
3. **split** divide la lista in due sottoliste per i lati sinistro e destro (se è un'equazione).
4. **termine** viene chiamato ricorsivamente su entrambe le sottoliste.
 - Se la sottolista ha un singolo elemento, quello è il termine.
 - Altrimenti, **split** divide ulteriormente la sottolista e **termine** viene chiamato su entrambe le parti.
 - L'operazione aritmetica viene applicata sui termini ottenuti dalle sottoliste.
5. Se è un'equazione, i termini sinistro e destro vengono confrontati con **::=** per verificare l'uguaglianza.
6. Il risultato finale dipende da cosa rappresenta la lista:

- Se è un'espressione, **termine** restituisce il risultato finale dell'operazione.
- Se è un'equazione, **equazione** restituisce **true** se i lati sinistro e destro sono uguali, altrimenti **false**.

Questo codice permette di valutare espressioni aritmetiche e risolvere equazioni semplici rappresentate come liste in Prolog.

```
 equazione([2,3,5,7,11]).  
true  
true  
true  
true  
true  
true  
true  
true  
true  
true  
true  
  
?- equazione([2,3,5,7,11]).
```


OPERAZIONI SU GRAMMATICHE CF



```
1 a(K) --> b(K), c, d, {n(K)}.  
2  
3 'S'(K) --> [up], 'S'(M), {K is M+1}.  
4  
5 'S'(K) --> [down], 'S'(M), {K is M-1}.  
6  
7 'S'(1) --> [up].  
8  
9 'S'(-1) --> [down].  
10  
11
```

Regole di $a(K)$

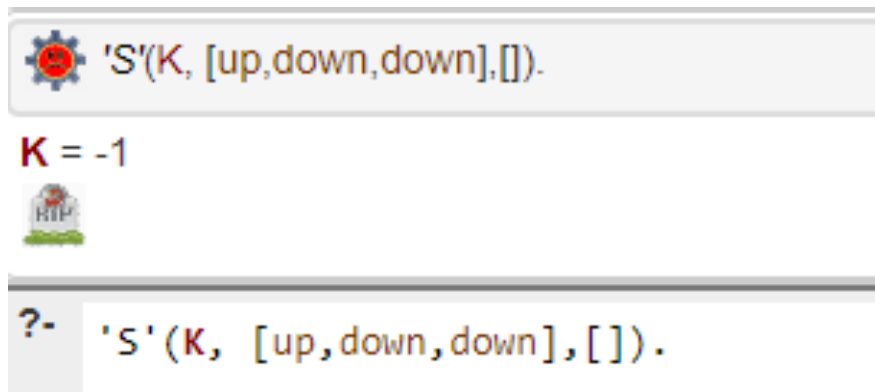
1. $a(K) \rightarrow b(K), c, d, \{n(K)\}$.
 - Questa regola afferma che per ogni K , il termine $a(K)$ può essere riscritto come una sequenza composta da $b(K)$, seguito da c , seguito da d , e infine da una condizione $\{n(K)\}$.
 - $\{n(K)\}$ è una condizione che deve essere soddisfatta affinché la regola possa essere applicata.

```
a(K, A, B) :-  
    b(K, A, C),  
    c(C, D),  
    d(D, E),  
    n(K),  
    B=E.
```

Queste regole possono essere viste come un modo per generare sequenze binarie (sequenze di 0 e 1) basate sul valore di K . La generazione delle sequenze è condizionata dalle regole di incremento e decremento di K e dalle condizioni base ('S' (1) e 'S' (-1)).

- Le regole con 'Up' e 'Down' determinano il contenuto delle liste e come K cambia in relazione a M .
- Le condizioni $\{K \text{ is } M+1\}$ e $\{K \text{ is } M-1\}$ assicurano che K venga aggiornato correttamente durante la generazione della sequenza.
- Le regole base ('S' (1) \rightarrow [1] e 'S' (-1) \rightarrow [0]) definiscono i casi terminali della generazione.

Query :-



esercizio per casa:

