

1. Relazioni di base:

- `genitore(X, Y)`: X è genitore di Y.
- `fratello(X, Y)`: X è fratello di Y.

2. Definizione di "fratello":

Prolog

```
fratello(X, Y) :- genitore(Z, X), genitore(Z, Y).
```

3. Spiegazione della regola:

- `fratello(X, Y)` è la testa della regola.
- `:- (Se)`: separa la testa dal corpo della regola.
- `genitore(Z, X), genitore(Z, Y)` è il corpo della regola.
- La regola dice che X è fratello di Y se hanno lo stesso genitore (Z).

4. Variabili:

- X e Y sono variabili che possono essere istanziate con qualsiasi nome.
- Z è una variabile che non compare nella testa della regola, quindi è una variabile "anonima".

5. Simboli speciali:

- `_` (underscore) serve per non unificare due parti di una regola.
- `a` rappresenta una costante.
- `1, 2, 3, ...` rappresentano numeri.
- `' - '` racchiude le costanti alfanumeriche.

6. Esempio di query:

Prolog

```
?- fratello(dario, gino).
```

7. Risultato della query:

La query verifica se Dario e Gino sono fratelli. Se entrambi hanno lo stesso genitore, la query restituirà `true`. Altrimenti, restituirà `false`.

1. Introduzione:

Il testo descrive come un programma Prolog può essere utilizzato per rappresentare e interrogare relazioni familiari.

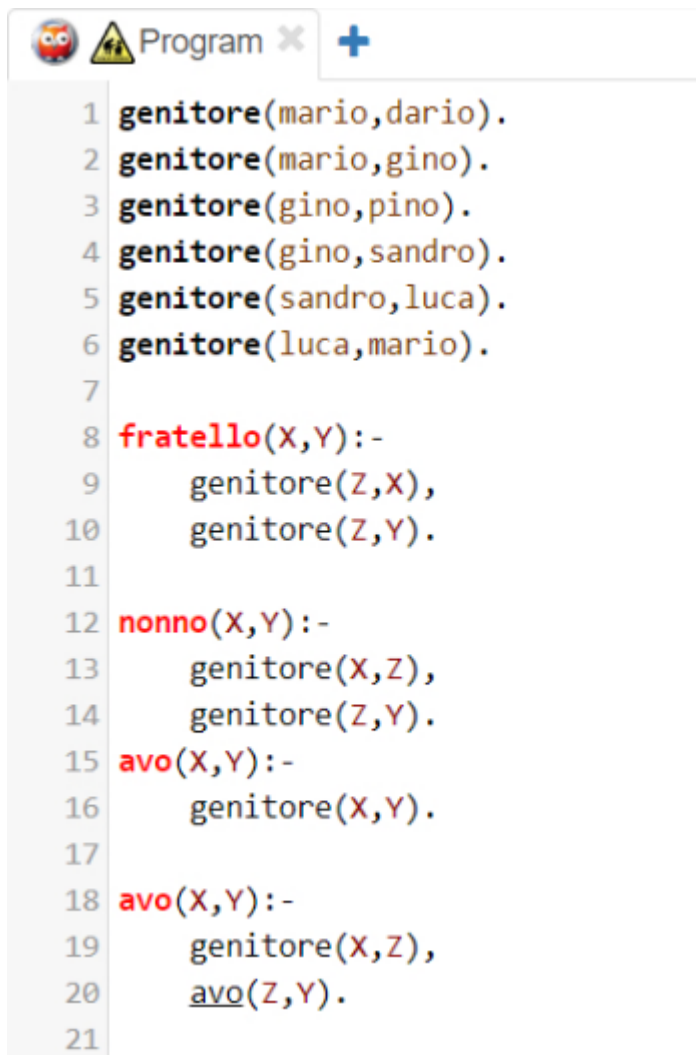


```
1 genitore(mario,dario).  
2 genitore(mario,gino).  
3 genitore(gino,pino).  
4 genitore(gino,sandro).  
5 genitore(sandro,luca).  
6 genitore(luca,mario).
```

2. Inserimento dati:

- I dati di input sono inseriti come fatti e regole in Prolog.
- I fatti sono relazioni semplici tra due entità.

- Le regole definiscono relazioni più complesse, come la nozione di "fratello".



```
1 genitore(mario,dario).
2 genitore(mario,gino).
3 genitore(gino,pino).
4 genitore(gino,sandro).
5 genitore(sandro,luca).
6 genitore(luca,mario).
7
8 fratello(X,Y):-
9     genitore(Z,X),
10    genitore(Z,Y).
11
12 nonno(X,Y):-
13     genitore(X,Z),
14     genitore(Z,Y).
15 avo(X,Y):-
16     genitore(X,Y).
17
18 avo(X,Y):-
19     genitore(X,Z),
20     avo(Z,Y).
21
```

3. Query:

- Il programma è progettato per rispondere a query che interrogano i legami familiari.
- Le query possono essere utilizzate per trovare fratelli, nonni e avi.

4. Esempio di query:

- La query `?- fratello(X, gino)` cerca tutti i fratelli di Gino.
- La query `?- avo(X, Y)` cerca tutti gli avi di Y.

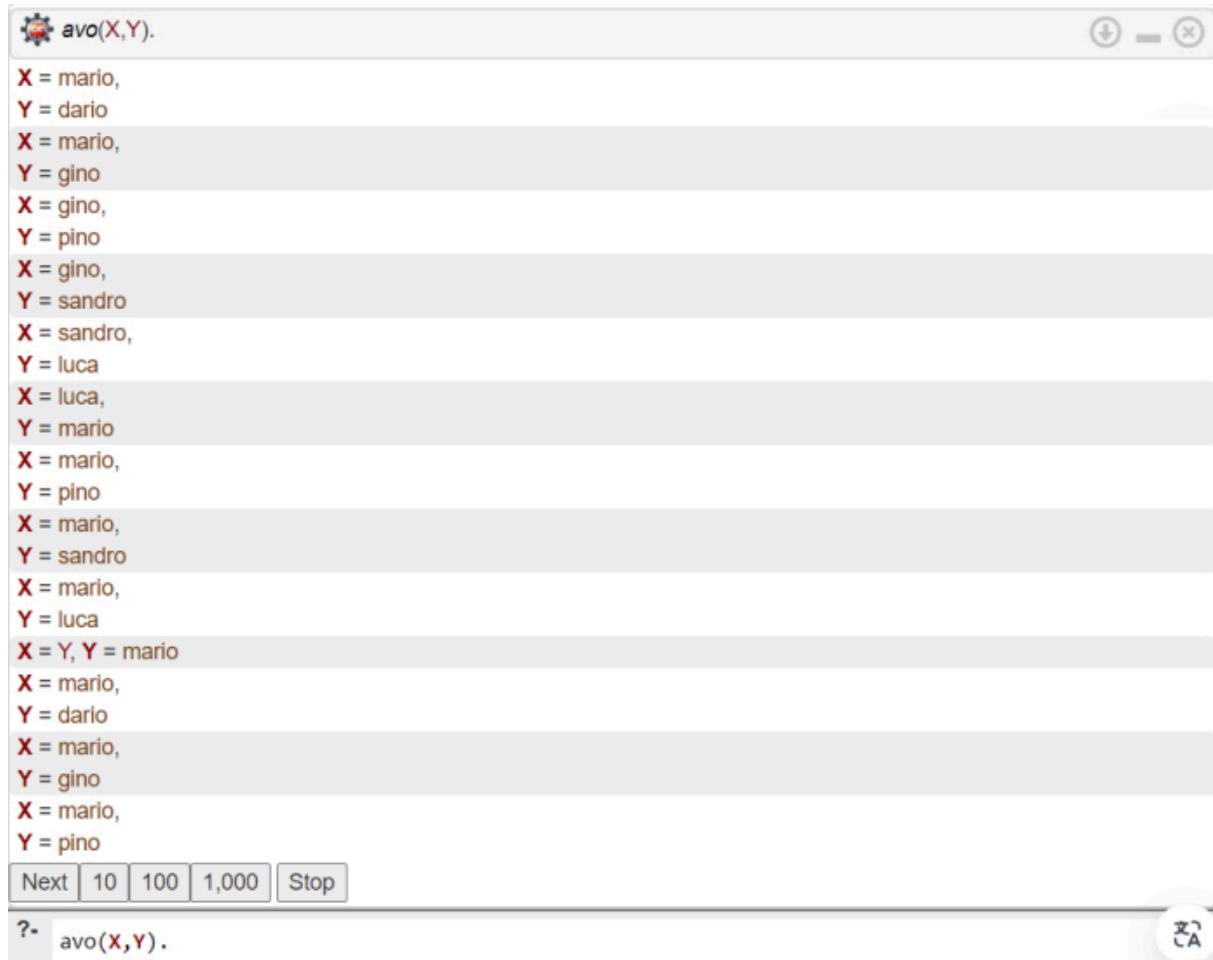
5. Caso dell'avo:

- Trovare l'avo è più complesso che trovare il nonno.
- Richiede un'induzione che collega più punti nel grafo di parentela.
- L'induzione ha un caso base e un passo induttivo.

query) ?- fratello(X,gino)



query) ?- avo (X,Y):



```
avo(X,Y):-  
    genitore(X,Y).
```

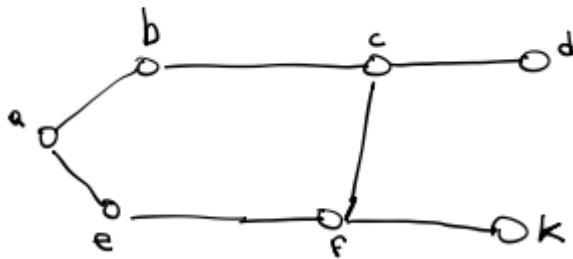
```
avo(X,Y):-  
    genitore(X,Z),  
    avo(Z,Y).
```

Puoi osservare sopra il passo base seguito dal passo induttivo per la soluzione al caso dell'avo.

LEZIONE 3:

Termini chiave:

- **Predicato:** una proposizione che può essere vera o falsa. In Prolog, i predicati sono rappresentati da nomi seguiti da parentesi.
- **Predicare:** affermare che un predicato è vero. In Prolog, si predica scrivendo il nome del predicato seguito da parentesi e dai suoi argomenti.
- **Induzione:** un metodo di ragionamento che permette di generalizzare da casi specifici a una regola generale.
- **Unificare:** trovare un valore comune per due variabili che le rende uguali.



Rappresentazione del grafo in Prolog:

- Il grafo può essere rappresentato in Prolog usando due predicati:
 - `path(X, Y)`: indica che esiste un percorso da X a Y nel grafo.
 - `edge(X, Y)`: indica che c'è un arco tra X e Y nel grafo.

Esempio:

Prolog

```
path(a, f). % c'è un percorso da a a f nel grafo
edge(a, b). % c'è un arco tra a e b nel grafo

edge(b, c). % c'è un arco tra b e c nel grafo
```

Induzione:

L'induzione può essere utilizzata per definire predicati più complessi, come la raggiungibilità di un nodo in un grafo.

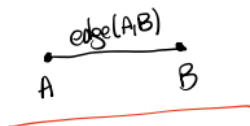
Prolog Swish per rappresentare il grafo utilizziamo questa forma:

```
Program x +
1  /* edge(S,E) */
2
3  edge(a,b).
4  edge(b,c).
5  edge(c,d).
6  edge(a,e).
7  edge(e,f).
8  edge(f,k).
9  edge(f,c).
10
11
```

Successivamente andremo a sviluppare il passo base e passo induttivo:

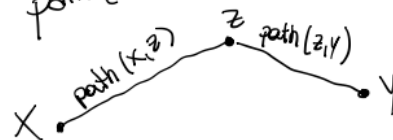
PASSO BASE:

\exists path(A,B) se esiste

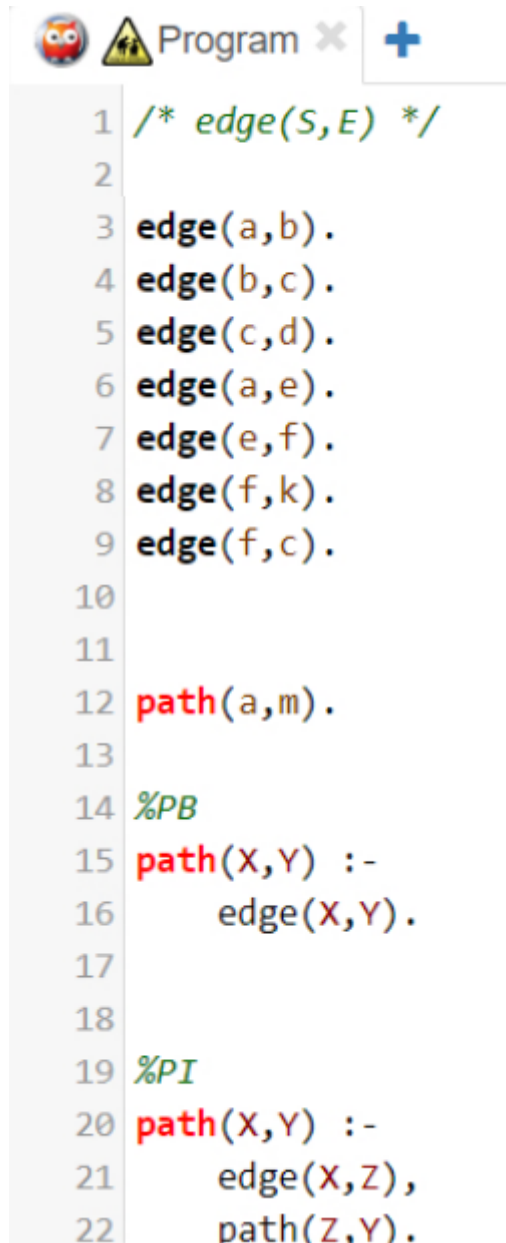


PASSO INDUTTIVO:

path(x,y)



Pertanto, una volta analizzato otterremo questa dicitura:



```
1  /* edge(S,E) */
2
3  edge(a,b).
4  edge(b,c).
5  edge(c,d).
6  edge(a,e).
7  edge(e,f).
8  edge(f,k).
9  edge(f,c).
10
11
12 path(a,m).
13
14 %PB
15 path(X,Y) :-
16     edge(X,Y).
17
18
19 %PI
20 path(X,Y) :-
21     edge(X,Z),
22     path(Z,Y).
```

Passo base (PB):

Nel passo base, definiamo la condizione più semplice per cui un percorso `path(X, Y)` esiste.

Condizione:

Esiste un arco `edge(X, Y)` che collega direttamente i nodi X e Y.

Regola Prolog:

Prolog

```
path(X, Y) :- edge(X, Y).
```

Spiegazione:

- La regola dice che se esiste un arco $\text{edge}(X, Y)$ nel grafo, allora c'è un percorso $\text{path}(X, Y)$ tra i nodi X e Y .
- Questa regola rappresenta il caso base dell'induzione, in cui il percorso è composto da un solo arco.

Esempio:

Supponiamo che il grafo contenga l'arco $\text{edge}(a, b)$. In questo caso, la regola $\text{path}(a, b)$ è vera perché esiste un arco diretto che collega i nodi a e b .

Passo induttivo (PI) :

Il passo induttivo definisce una regola per trovare percorsi più complessi che non sono solo archi diretti.

Regola Prolog:

Prolog

$\text{path}(X, Y) :- \text{edge}(X, Z), \text{path}(Z, Y).$

Spiegazione:

- La regola dice che se esiste un arco $\text{edge}(X, Z)$ che collega X a un nodo intermedio Z , e se esiste un percorso $\text{path}(Z, Y)$ da Z a Y , allora esiste un percorso $\text{path}(X, Y)$ da X a Y .
- In altre parole, questa regola permette di costruire percorsi composti da più archi concatenati.

Esempio:

Supponiamo che il grafo contenga gli archi $\text{edge}(a, b)$ e $\text{edge}(b, c)$. In questo caso, la regola $\text{path}(a, c)$ è vera perché:

1. Esiste un arco $\text{edge}(a, b)$ che collega a a b .
2. Esiste un percorso $\text{path}(b, c)$ da b a c (soddisfatto dal passo base).

Quindi, la regola conclude che c'è un percorso $\text{path}(a, c)$ da a a c .

Adesso osserviamo l'output di alcune query:

?- path(a,d)

```
path(a,d)
true 1
true 2
false
```

?- edge(K,M)

```
edge(K,M).
K = a,
M = b
K = b,
M = c
K = c,
M = d
K = a,
M = e
K = e,
M = f
K = f,
M = k
K = f,
M = c
```

?- path(K,d)

```
path(K,d).
K = c
K = a
K = b
K = a
K = e
K = f
```

?- listing(path)

```
listing(path)

path(a, m).
path(X, Y) :-
    edge(X, Y).
path(X, Y) :-
    edge(X, Z),
    path(Z, Y).

true 1
```

Schema: Induzione strutturale per la verifica di appartenenza in una lista

Introduzione

L'induzione strutturale è un metodo per dimostrare proprietà di strutture ricorsive, come le liste. In questo caso, la proprietà da dimostrare è se un elemento x appartiene a una lista L .

Definizioni

- **Lista:** Una struttura data rappresentata da parentesi quadrate che racchiudono elementi separati da virgole, ad esempio: $[a, b, c, d, e]$.
- **Modo Testa-Coda:** Una lista è rappresentata come $[H|T]$, dove:
 - H è l'elemento **testa** della lista.
 - T è la **coda** della lista, ovvero tutti gli elementi tranne H .
 - **Nota:** $[H|T] = []$ è sempre falso.

Predicato di appartenenza: $\text{appartiene}(X, L)$

Il predicato $\text{appartiene}(X, L)$ verifica se un elemento x appartiene a una lista L .

Dimostrazione con induzione strutturale

Per dimostrare che $\text{appartiene}(X, L)$ vale per tutte le liste L , usiamo l'induzione strutturale:

- Dimostriamo che la proprietà vale per la lista vuota $[]$.
- Se x appartiene a $[]$, allora x deve essere un elemento vuoto, il che non è possibile.
- Quindi, la proprietà è falsa per la lista vuota.
- Assumiamo che la proprietà valga per una qualsiasi lista S con elementi x e L .
- Dimostriamo che la proprietà vale anche per la lista $[x|S]$.
- Ci sono due possibilità:
 - **Caso 1:** x unifica con H , ovvero x è l'elemento testa della lista $[x|S]$.
 - In questo caso, x appartiene sicuramente alla lista $[x|S]$.
 - **Caso 2:** x non unifica con H , ovvero x non è l'elemento testa della lista $[x|S]$.
 - Per l'ipotesi induttiva, sappiamo che $\text{appartiene}(x, S)$ è vera.
 - Quindi, x appartiene alla coda S della lista $[x|S]$.
- In entrambi i casi, la proprietà è vera per la lista $[x|S]$.


- Analisi SWISH_

```

1 /* appartiene(X,L) */
2 /* caso1 X appartiene ad H --> Sostituiamo H con X --> L corrisponderà a [X/T]
3 --> Sostituiamo T con _ --> L = [X/_] */
4 appartiene(X,[X|_]).
5
6 appartiene(X,[_|T]):-
7     appartiene(X,T).


```

Progettiamo delle query:

 appartiene(2,[1,2,2,3,3,44])

true

Next 10 100 1,000 Stop

 appartiene(1,[0,2,3,4,5,6])

false

 appartiene(10,[0,2,10,4,5,6,1010])


true

Next 10 100 1,000 Stop


?- appartiene(10,[0,2,10,4,5,6,1010])

Ipotizziamo di domandare se l'elemento x appartiene alla lista [1,2,3]:

N.B x → MAIUSCOLO.

 appartiene(x,[1,2,3])

false

 appartiene(X,[1,2,3])

X = 1

X = 2

X = 3

false

?- appartiene(X,[1,2,3])

X assume i singoli valori della lista.

Schema: Operazioni sulle liste

Estrazione di più elementi

Una lista può contenere più elementi all'inizio, rappresentati usando la notazione

$[H1, H2 \mid T]$:

- $H1$ e $H2$ sono i primi **due** elementi della lista.
- T è la **coda** della lista, ovvero tutti gli elementi tranne i primi due.

Esempio:

Lista = [1, 2, 3, 4, 5]

$H1 = 1$

$H2 = 2$

$T = [3, 4, 5]$

Concatenazione

La funzione `concatena(A, B, C)` concatena due liste A e B in una nuova lista C :

- C è la lista **concatenata**, formata da:
 - A (la prima lista)
 - B (la seconda lista)

Sintassi:

`concatena(A, B, C)`

$C=[H|L] \rightarrow \{C \text{ è concatenazione di } A \text{ e } B \text{ sè } A=[H|T], H \text{ è il primo elemento di } C(1^\circ \text{ elementi di } A) \text{ ed } L \text{ è la concatenazione di } T(\text{ coda } A) \text{ e } B.$

Spiegazione:

- Se A è vuota ($A = []$), allora la lista concatenata C è semplicemente B (passo base).
- Altrimenti, se A ha un elemento testa H e una coda T , allora:
 - La lista concatenata C ha H come primo elemento.
 - La coda di C è ottenuta concatenando la coda di A (T) con la lista B .

Induzione strutturata:

- **Passo base:** `concat([], A, A)`
- `concatena(T, B, L)`

- ---> `concatena([H|T],B,C)` (la proprietà che vogliamo raccontare) ---> `A=[H|T] C=[H|L], concatena(T,B,L)` (dove `L= T+B`).
- ---> `concatena([H|T],B,[H|L])`.


Osserviamo come scrivere questa regola su SWISH:


```

10 /* PB: */
11 concatena([],A,A).
12
13 /* INDUZ STRUTT */
14 concatena([H|T],B,[H|L]):-
15     concatena(T,B,L).


```


Ipotizziamo possibili query:






B = C,
X = []





C = [1, 2|B]



Definizione del predicato `rivoltata`

Il predicato `rivoltata` prende due argomenti:

- `L`: una lista generica
- `RL`: una lista vuota o una lista che conterrà la lista `L` ribaltata

Il predicato restituisce `true` se la lista `L` viene ribaltata e memorizzata nella lista `RL`.

Spiegazione passo dopo passo

Il predicato è definito da due clausole:

Clausola 1:

Prolog

```
rivoltata([],RL) .
```

Questa clausola base specifica che se la lista L è vuota ($[]$), allora la lista ribaltata è anch'essa vuota (RL). In altre parole, la lista vuota è il suo stesso "ribaltamento".

Clausola 2:

Prolog

```
rivoltata([H|T], RL):-  
    rivoltata(T,RT) ,  
    append(RT,[H],RL) .
```

Questa clausola ricorsiva si occupa di ribaltare una lista non vuota. Ecco come funziona:

1. **Ricorsione:** Viene effettuata una chiamata ricorsiva al predicato `rivoltata` sulla coda della lista L , memorizzando il risultato nella lista RT . In altre parole, la coda della lista viene ribaltata e memorizzata in RT .
2. **Composizione:** La testa della lista H viene aggiunta alla fine della lista ribaltata RT , ottenendo la lista ribaltata completa RL .
3. **Unificazione:** La lista ribaltata RL viene unificata con l'argomento RL della chiamata originale.

In sostanza, la clausola 2 scompone la lista L in testa e coda, ribalta ricorsivamente la coda, e aggiunge la testa alla coda ribaltata per ottenere la lista ribaltata completa.

```
1 /* INDUZIONE STRUTTURALE rivoltata( L, RL ) */  
2 |  
3 rivoltata([],[]).  
4  
5  
6 rivoltata([H|T], RL):-  
7     rivoltata(T,RT),  
8     append(RT,[H],RL).  
9
```

Testiamo la query:

```
⚙️ rivoltata([1,2,3],X)
```

X = [3, 2, 1]

```
?- rivoltata([1,2,3],X)
```

```
⚙️ rivoltata([a,b,c,d,e,f],X)
```

X = [f, e, d, c, b, a]

```
?- rivoltata([a,b,c,d,e,f],X)
```

Definizione del predicato `permutazione(A,B)`

Il predicato `permutazione(A,B)` verifica se due liste, **A** e **B**, sono permutazioni l'una dell'altra. In altre parole, controlla se **B** contiene tutti gli stessi elementi di **A**, senza alcun duplicato e in qualsiasi ordine.

Implementazione usando l'induzione strutturale:

Il codice fornito utilizza l'induzione strutturale per definire il predicato `permutazione(A,B)`. L'induzione strutturale è una tecnica per definire predicati su strutture dati ricorsive come le liste.

Spiegazione passo dopo passo:

1. Passo base:

- `permutazione([],[])`.

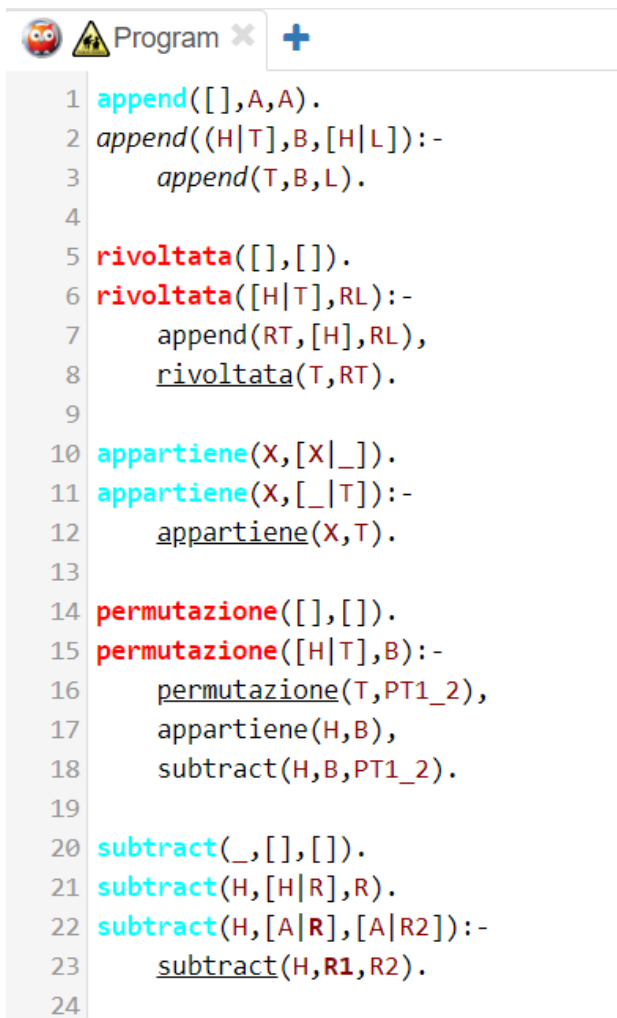
Questa clausola base specifica che se entrambe le liste sono vuote (`[]`), allora sono sicuramente permutazioni tra loro.

2. Passo induttivo:

- `permutazione([H|T],B) :-`
- `permutazione(T,Pt1_2),`
- `appartiene(H,B),`
- `subtract(H,B,PT1_2).`

Questa clausola ricorsiva si occupa di verificare se una lista non vuota ($[H|T]$) è una permutazione di un'altra lista B . Ecco come funziona:

1. **Induzione:** Viene effettuata una chiamata induttiva al predicato `permutazione(T, Pt1_2)` per verificare se la coda T della lista è una permutazione di B . Il risultato viene memorizzato nella lista `Pt1_2`.
2. **Verifica appartenenza:** Si controlla se la testa H della lista è presente nella lista B utilizzando il predicato `appartiene(H, B)`.
3. **Rimozione elemento:** Se H è presente in B , viene rimosso da B utilizzando il predicato `subtract(H, B, Pt1_2)`. La lista `Pt1_2` rappresenta la lista B con l'elemento H rimosso.
4. **Unificazione:** Se tutte le verifiche sono soddisfatte, la lista `Pt1_2` viene unificata con l'argomento B della chiamata originale, indicando che A e B sono permutazioni.



```
1 append([],A,A).
2 append([H|T],B,[H|L]):-
3     append(T,B,L).
4
5 rivoltata([],[]).
6 rivoltata([H|T],RL):-
7     append(RT,[H],RL),
8     rivoltata(T,RT).
9
10 appartiene(X,[X|_]).
11 appartiene(X,[_|T]):-
12     appartiene(X,T).
13
14 permutazione([],[]).
15 permutazione([H|T],B):-
16     permutazione(T,Pt1_2),
17     appartiene(H,B),
18     subtract(H,B,Pt1_2).
19
20 subtract(_,[],[]).
21 subtract(H,[H|R],R).
22 subtract(H,[A|R],[A|R2]):-
23     subtract(H,R1,R2).
24
```


Spiegazione del codice Prolog: subtract

Questo codice Prolog definisce un predicato chiamato `subtract/3` che serve per sottrarre elementi da liste. Prende tre argomenti:

- **Testa (Head):** L'elemento da rimuovere potenzialmente dalla prima lista.
- **Lista1:** La prima lista dalla quale potrebbero essere sottratti elementi.
- **Lista2:** La lista risultante dopo aver sottratto `Testa` da `Lista1`.

Il codice funziona in modo ricorsivo, ovvero richiama se stesso all'interno delle sue clausole. Vediamo la spiegazione di ciascuna clausola:

1. Caso Base (Lista Vuota):

- `subtract(_, [], [])`
 - Questa clausola afferma che se `Lista1` (il secondo argomento) è vuota (`[]`), allora anche il risultato `Lista2` (il terzo argomento) dovrebbe essere vuota (`[]`).
 - Intuitivamente, non c'è nulla da sottrarre da una lista vuota, quindi la lista risultante rimane vuota.

2. Corrispondenza Testa:

- `subtract (Testa, [Testa | Coda], Coda)`
 - Questa clausola gestisce il caso in cui l'elemento `Testa` corrisponde al primo elemento di `Lista1`.
 - Se `Testa` è uguale al primo elemento in `Lista1` (rappresentato come `[Testa | Coda]`), allora la `Lista2` risultante è semplicemente la `Coda` (elementi rimanenti) di `Lista1`.
 - In sostanza, questa clausola rimuove l'elemento `Testa` corrispondente dall'inizio di `Lista1`.

3. Caso induttivo (Nessuna Corrispondenza):

- `subtract (Testa, [A | Coda], [A | Resto]) :- subtract (Testa, Resto, Coda).`
 - Questo è il caso induttivo che si occupa delle situazioni in cui `Testa` non corrisponde al primo elemento di `Lista1`.
 - Chiama `subtract (Testa, Resto, Coda)` ricorsivamente.
 - `Resto` è una variabile temporanea che conterrà il risultato della sottrazione di `Testa` dagli elementi rimanenti di `Lista1` (escluso il primo elemento).
 - `Coda` rappresenta l'intera `Lista1` tranne il primo elemento.

- Dopo la chiamata induttiva, il primo elemento di `Lista1 (A)` viene aggiunto di nuovo a `Resto` utilizzando `[A | Resto]` per formare la `Lista2` finale.

Operatore 'is' permette di unificare due componenti emettendo prima una formula di richiesta

```

4 is 3 + 1
true
A is 3 + 1
A = 4
?- A is 3 + 1

```

Ipotizziamo di studiare il caso `A is 1, A is A + 1`. Questo caso è False poichè non è possibile effettuare un assegnamento in linguaggio prolog (A non è unificabile a se stesso):

```

A is 1, A is A + 1
false
?- A is 1, A is A + 1

```

Definizione del predicato `lung(A, N)`

Analisi e spiegazione del predicato `lung([], 0)` e

`lung([H|T], N) :`

`lung([], 0) :`

Questo predicato rappresenta la **base** della definizione ricorsiva della lunghezza di una lista.

- `lung([], 0) :`
 - `[]`: Rappresenta una lista vuota.
 - `0`: Indica che la lunghezza di una lista vuota è 0.

In parole semplici, questo predicato stabilisce che la lunghezza di una lista vuota è sempre 0.

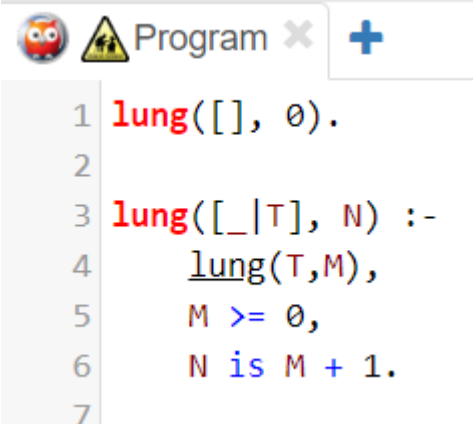
Predicato `lung([H|T], N):`

Questo predicato rappresenta il **caso Induttivo** della definizione della lunghezza di una lista.

- `lung([H|T], N):`
 - `[H|T]`: Rappresenta una lista non vuota, dove `H` è la testa (primo elemento) e `T` è la coda (lista rimanente).
 - `N`: Una variabile che rappresenta la lunghezza totale della lista.

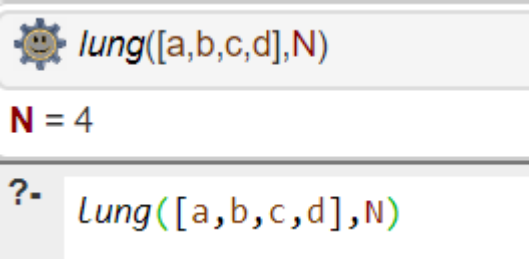
Il predicato utilizza la ricorsione per calcolare la lunghezza della lista:

1. `lung(T, M)`: Esegue una chiamata ricorsiva per calcolare la lunghezza della coda (`T`) della lista e la assegna alla variabile `M`.
2. `M >= 0`: Verifica che la lunghezza della coda (`M`) sia un valore non negativo, come ci si aspetta per la lunghezza di una lista.
3. `N is M + 1`: Calcola la lunghezza totale (`N`) della lista sommando 1 (per la testa) alla lunghezza della coda (`M`).

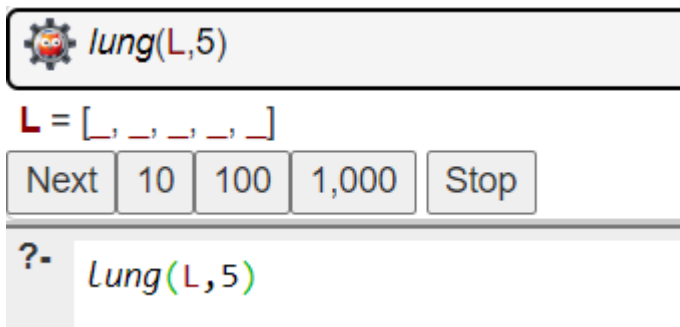


```
1 lung([], 0).
2
3 lung([_|T], N) :-
4     lung(T, M),
5     M >= 0,
6     N is M + 1.
7
```

Query:



```
lung([a,b,c,d],N)
N = 4
?- lung([a,b,c,d],N)
```



Definizione del predicato `numero_di_el(Lista, Elemento, Numero)`.

Il predicato **`numero_di_el(Lista, Elemento, Numero)`** calcola il numero di occorrenze di un elemento specifico all'interno di una lista.

Analisi e spiegazione del predicato `numero_di_el/3`:

Il predicato `numero_di_el/3` calcola il numero di occorrenze di un elemento specifico all'interno di una lista.

Sintassi:

Prolog

```
numero_di_el(Lista, Elemento, Numero)
```

- **Lista**: La lista da analizzare.
- **Elemento**: L'elemento da cercare nella lista.
- **Numero**: Una variabile che conterrà il numero di occorrenze di **Elemento** in **Lista**.

Comportamento:

Il predicato si basa su due clausole:

1. Clausola per lista vuota:

Prolog

```
numero_di_el([], _, 0).
```

- Se la lista `Lista` è vuota (`[]`), il numero di occorrenze di `Elemento` è 0 (0).

2. Clausola per lista non vuota:

Prolog

```
numero_di_el([El|T], El, N) :-
    numero_di_el(T, El, M),
    N is M + 1.
```

- Se la lista `Lista` non è vuota:
 - Se la testa (`El`) della lista coincide con `Elemento`:
 - Viene effettuata una chiamata ricorsiva a `numero_di_el/3` per calcolare il numero di occorrenze di `Elemento` nella coda (`T`) della lista, memorizzando il risultato in `M`.
 - Il numero totale di occorrenze (`N`) è calcolato sommando 1 (per l'occorrenza nella testa) al numero di occorrenze nella coda (`M`).

3. Mancanza di una clausola per lista non vuota con testa diversa:

Nella versione fornita del codice, manca una clausola per gestire il caso in cui la testa della lista non coincide con l'elemento da cercare. Di conseguenza, se la testa non corrisponde, il predicato fallisce senza fornire alcun risultato.

Comportamento corretto:

Per un comportamento completo, è necessario aggiungere una clausola per gestire il caso in cui la testa non coincide con l'elemento:

```
numero_di_el([_|T], El, M) :-
    numero_di_el(T, El, M).
```


- Se la testa (`_`) della lista non coincide con `Elemento`:
 - Viene effettuata una chiamata ricorsiva a `numero_di_el/3` per calcolare il numero di occorrenze di `Elemento` nella coda (`T`) della lista, memorizzando il risultato in `M`.

```

11 /* numero_di_elementi(Lista, Elemento, Numero) */
12
13 numero_di_el([], _, 0).
14 numero_di_el([El|T], El, N) :-
15     numero_di_el(T, El, M),
16     N is M + 1.
17 numero_di_el([X|T], El, M):-
18     X \= El,
19     numero_di_el(T,El,M).
20

```

query:

 numero_di_el([1,1,1,2,2,3,4,5],1,N)

N = 3

false

?- numero_di_el([1,1,1,2,2,3,4,5],1,N)

Funzioni NOT, CUT e FAIL in Prolog:

NOT: `!=`

- La funzione `not` in Prolog non è un operatore logico come in altri linguaggi.
- In Prolog, `not(P)` verifica se il goal `P` **fallisce**. Se `P` fallisce, `not(P)` ha successo. Se `P` ha successo, `not(P)` fallisce.
- La funzione `not` viene spesso utilizzata per implementare la negazione in Prolog.

Es:

- Mario e Maria sono amici

```
1 amici(mario, maria).
```

-oss:

- Se scrivo qualche fatto diventa ****vero****, tutto il resto è ****falso****.
- Però per verificare che sia falso devo verificare che non è stato scritto quindi.
- Se ho una lista di amici

```
2 amici(mario, maria).  
3 amici(mario, dario).  
4 amici(mario, pino).  
5
```

- ?- not amici(mario, rino)

CUT:

- Il `cut` in Prolog è un operatore che **blocca il backtracking**.
- Se un `cut` viene eseguito con successo, il backtracking non è più possibile per le clausole precedenti al `cut`.
- Il `cut` può essere utilizzato per migliorare l'efficienza del programma Prolog evitando il backtracking inutile.

Esempio:

Prolog

```
p(X) :- not(q(X)), !, r(X).
```

$q(a)$.

$r(b)$.

In questo esempio, la regola $p(X)$ verifica se X non è un valore che soddisfa $q(X)$. Se $q(X)$ fallisce, il `cut` blocca il backtracking e la regola $r(X)$ viene verificata.

Analisi del predicato $p(A)$ in Prolog:

Il predicato $p(A)$ in Prolog è composto da diverse clausole che definiscono il suo comportamento:

1. $f(A)$:

- La prima clausola richiede che il goal $f(A)$ abbia successo.
- $f(A)$ può essere qualsiasi predicato definito nel programma Prolog.
- Il successo di $f(A)$ non ha effetto sul valore di A .

2. $write(A), nl$:

- Dopo il successo di $f(A)$, il valore di A viene stampato sulla console.
- Il carattere `nl` aggiunge un ritorno a capo alla stampa.

3. `!`:

- Il simbolo `!` indica un `cut`.
- Il `cut` blocca il backtracking su questa clausola.
- In altre parole, una volta che il `cut` viene eseguito, non è possibile tornare a questa clausola per provare alternative.

4. $g(A)$:

- La quarta clausola richiede che il goal $g(A)$ abbia successo.
- $g(A)$ può essere qualsiasi predicato definito nel programma Prolog.
- Il successo di $g(A)$ non ha effetto sul valore di A .

5. $write(A), nl$:

- Il valore di A viene nuovamente stampato sulla console.

6. $k(A)$:

- L'ultima clausola richiede che il goal $k(A)$ abbia successo.

- $k(A)$ può essere qualsiasi predicato definito nel programma Prolog.
- Il successo di $k(A)$ determina il successo del predicato $p(A)$.

Comportamento del predicato:

- Il predicato $p(A)$ esegue le seguenti azioni:
 - Esegue il predicato $f(A)$.
 - Stampa il valore di A .
 - Blocca il backtracking sulla clausola.
 - Esegue il predicato $g(A)$.
 - Stampa nuovamente il valore di A .
 - Esegue il predicato $k(A)$.
- Il valore di A può essere modificato dai predicati $f(A)$, $g(A)$ e $k(A)$.

```

10 f(a).
11 f(b).
12 g(a).
13 g(b).
14 g(j).
15 k(a).

```

```

21 p(A):-
22     f(A),
23     write(A),nl,
24     !,
25     g(A),
26     write(A),nl,
27     k(A).
28

```

 $p(X)$.

a

a

X = a

?- $p(x)$.

Spiegazione del codice Prolog: $p(A)$

Questo codice Prolog definisce un predicato chiamato $p(A)$. Vediamo passo passo cosa succede quando viene chiamato $p(A)$:

1. $f(A)$:

- Innanzitutto, viene eseguito il predicato $f(A)$. Non sappiamo esattamente cosa fa $f(A)$ perché non è definito nel codice mostrato.
- Presumiamo che $f(A)$ svolga qualche operazione su A e che il suo successo o fallimento possa influenzare il valore di A .

2. Stampa con valore A :

- Se $f(A)$ ha successo, allora viene eseguita la clausola successiva:
 - `write('10: '), write(A), nl:`
 - Questa riga stampa prima la stringa "10: " sulla console.
 - Poi, stampa il valore corrente di A .
 - Infine, aggiunge un carattere di "a capo" (`nl`) per andare su una nuova linea.

3. Cut (!):

- Il simbolo `!` rappresenta un `cut`.
- Il `cut` è un'istruzione potente ma delicata in Prolog.
- In questo caso, il `cut` **blocca il backtracking** su questa clausola di $p(A)$.
- Significa che una volta eseguito il `cut`, il programma non tornerà più a provare alternative per $f(A)$, anche se $g(A)$ o $k(A)$ falliscono.

4. $g(A)$:

- Dopo il `cut`, viene eseguito il predicato $g(A)$.
- Ancora una volta, non sappiamo cosa fa $g(A)$, ma presumiamo che possa svolgere qualche operazione su A .
- Il successo o fallimento di $g(A)$ è fondamentale per il successo finale di $p(A)$.

5. Seconda stampa con valore A :

- Se $g(A)$ ha successo, viene eseguita la clausola successiva:
 - `write('13: ', write(A), nl):`
 - Questa riga è simile alla precedente, ma stampa "13: " prima del valore di A .

6. $k(A)$:

- Infine, viene eseguito il predicato $k(A)$.
- Similmente a $f(A)$ e $g(A)$, non sappiamo cosa fa $k(A)$, ma presumiamo che possa svolgere qualche operazione su A .
- Il successo o fallimento di $k(A)$ determina il successo finale dell'intero predicato $p(A)$.

Riassunto:

- $p(A)$ esegue $f(A)$, stampa il valore di A , esegue $g(A)$ (senza backtracking su $f(A)$), stampa nuovamente il valore di A , ed esegue $k(A)$.
- Il valore di A può essere modificato da $f(A)$, $g(A)$, o $k(A)$.
- Il `cut` assicura che il programma non torni indietro e provi alternative per $f(A)$.

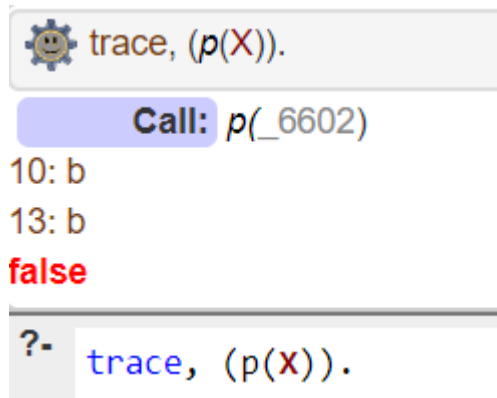
```
11 f(b).
12 g(a).
13 g(b).
14 g(j).
15 k(a).
16
17
18
19
20
21 p(A):-
22     f(A),
23     write('10: '),write(A),nl,
24     !,
25     g(A),
26     write('13: '),write(A),nl,
27     k(A).
28
```

```
⚙️ p(X).
10: b
13: b
false

?- p(X).
```

Debug in Prolog:

Il debug in Prolog è il processo di individuazione e correzione degli errori nel codice Prolog.



```
⚙ trace, (p(X)).  
Call: p(_6602)  
10: b  
13: b  
false  
?- trace, (p(X)).
```

Strumenti di debug:

Esistono diversi strumenti per il debug del codice Prolog:

- **Tracciamento:** Il tracciamento consente di seguire passo dopo passo l'esecuzione del codice Prolog. Vengono visualizzate informazioni su quali predicati vengono chiamati e quali valori vengono utilizzati.
- **Spie:** Le spie sono punti di arresto che consentono di interrompere l'esecuzione del codice Prolog e di esaminare lo stato del programma.
- **Debugger:** I debugger Prolog offrono un'interfaccia grafica per il debug del codice. Consentono di impostare spie, esaminare le variabili e il loro valore e di eseguire il codice passo dopo passo.

FAIL:

Il `fail` in Prolog è un predicato che **fa fallire il goal corrente**.

Il `fail` viene spesso utilizzato in combinazione con il `cut` per implementare la negazione in Prolog.

Descrizione del predicato `fallimento_di_g(A)`:

Il predicato `fallimento_di_g(A)` in Prolog è definito come segue:

Prolog

```
fallimento_di_g(A) :-  
    g(A),  
    fail.
```

Analisi:

- **Nome:** `fallimento_di_g(A)`
- **Argomenti:** Un argomento `A`
- **Comportamento:**
 1. **Chiamata a `g(A)`:** Il predicato `g(A)` viene chiamato con l'argomento `A`.
 2. **Falsificazione:** Se `g(A)` ha successo, il predicato `fail` viene chiamato per **forzare il fallimento** di `fallimento_di_g(A)`.

Significato:

Il predicato `fallimento_di_g(A)` ha successo se e solo se il predicato `g(A)` **fallisce**. In altre parole, `fallimento_di_g(A)` serve a testare se `g(A)` fallisce per un determinato valore di `A`.

```
30 %fail  
31 fallimento_di_g(A):-  
32     g(A),  
33     fail.
```

`fallimento_di_g(X).`

Breakpoint 654 in 1-st clause of `fallimento_di_g/1` at [Line 33](#)

Call: `fail`

Call: `fail`

Call: `fail`

false

`fallimento_di_g(X).`

Breakpoint 659 in 1-st clause of `fallimento_di_g/1` at [Line 32](#)

Call: `g(_7578)`

false

?- `fallimento_di_g(X).`

Analisi del predicato `fallimento_di_g(A)` in Prolog:

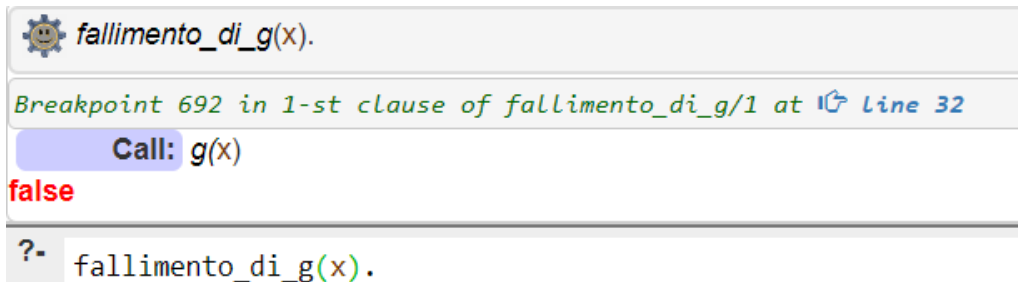
Comportamento:

Il predicato `fallimento_di_g(A)` in Prolog ha lo scopo di **far fallire il predicato** `g(A)`.

Analisi passo-a-passo:

1. `g(A)`: Il predicato `g(A)` viene chiamato. Non sappiamo cosa fa `g(A)` perché non è definito nel codice mostrato. Presumiamo che `g(A)` possa svolgere operazioni su `A` e che il suo successo o fallimento dipenda da `A`.
2. `fail`: Se `g(A)` ha successo, viene eseguito il predicato `fail`.
 - `fail` è un predicato speciale in Prolog che **fa fallire il goal corrente**.
 - In questo caso, il goal corrente è `fallimento_di_g(A)`.

```
30 %fail
31 fallimento_di_g(A):-
32     g(A),fail.
```



The screenshot shows a Prolog debugger window. At the top, there is a gear icon and the text `fallimento_di_g(x).`. Below this, a message states: `Breakpoint 692 in 1-st clause of fallimento_di_g/1 at Line 32`. Underneath, a blue box contains the text `Call: g(x)`. Below the blue box, the word `false` is displayed in red. At the bottom, a prompt `?-` is followed by `fallimento_di_g(x).`

Descrizione del predicato `fallimento_di_g(A)` :

Il predicato `fallimento_di_g(A)` in Prolog definisce un caso particolare per il fallimento del predicato `g(A)`.

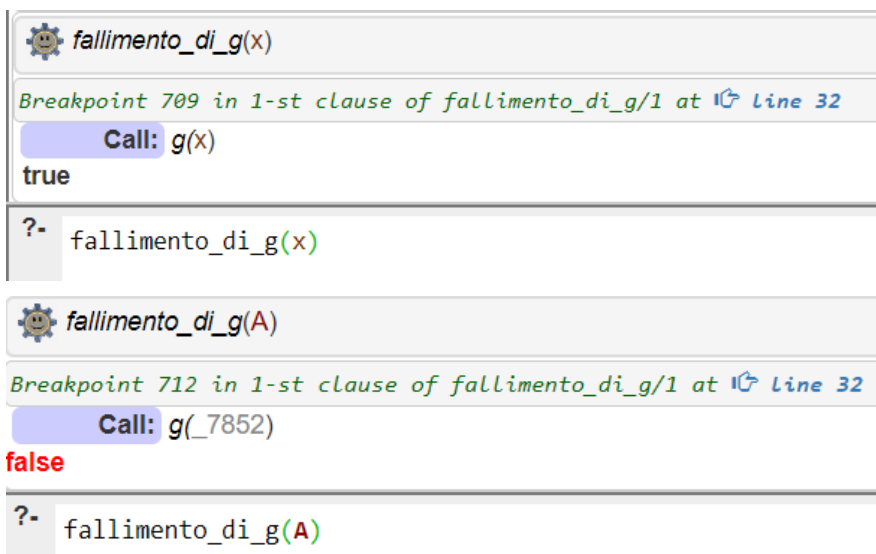
Analisi:

- **Nome:** `fallimento_di_g(A)`
- **Argomenti:** Un argomento `A`
- **Comportamento:**
 1. **Prima clausola:**
 - Se `g(A)` ha successo, viene eseguito il `cut (!)`.
 - Il `cut` blocca il backtracking su questa clausola.
 - Successivamente, viene eseguito `fail`, che fa fallire il predicato `fallimento_di_g(A)`.
 2. **Seconda clausola:**
 - Se `g(A)` fallisce o non viene unificato con `A`, questa clausola viene attivata.
 - In questo caso, `fallimento_di_g(A)` ha successo.


In parole semplici:

- Il predicato `fallimento_di_g(A)` ha successo se e solo se `g(A)` fallisce.
- Se `g(A)` ha successo, `fallimento_di_g(A)` fallisce.

```
30 %fail
31 fallimento_di_g(A):-
32     g(A),!,fail.
33 fallimento_di_g(_).
```

 `fallimento_di_g(x)`
Breakpoint 709 in 1-st clause of `fallimento_di_g/1` at [Line 32](#)
Call: `g(x)`
true

?- `fallimento_di_g(x)`

 `fallimento_di_g(A)`
Breakpoint 712 in 1-st clause of `fallimento_di_g/1` at [Line 32](#)
Call: `g(_7852)`
false

?- `fallimento_di_g(A)`

MYNOT:

ciò che è dentro è vero, tutto il resto è falso.

```
36 %negation as failure
37 mynot(Predicato):-
38     Predicato,!fail.
39 mynot(_).
```

```
mynot(g(b)).
```

false

```
mynot(g(x)).
```

true

```
?- mynot(g(x)).
```

Num_elementi:

Se verifico i primi due nEl non entrerà nel 3, però se ci sono più elementi si ferma sulla prima occorrenza.

- Cioè non arriveremo a controllare altri valori di occorrenze ma con il "!" ci fermeremo alla prima variabile
- Quindi il cut bisogna saperlo usare perché qui mi serviva vedere tutti gli elementi se facevo num_elementi([1,2, 2, 3], A, X)

```
43 %num_elementi(X,L,N).
44
45 num_elementi(_,[],0).
46 num_elementi(X,[X|T],N):-
47     !,
48     num_elementi(X,T,N1),
49     N is N1 + 1.
50 num_elementi(X,[_|T],N):-
51     num_elementi(X,T,N).
```

```
num_elementi(a,[a,b,a,k],N).
```

N = 2

false

```
?- num_elementi(a,[a,b,a,k],N).
```


Esercitazione 28/Marzo/2024

Parte dichiarativa

Abbiamo individuato un business molto interessante: vendere sogni alle persone. Si vuol far credere che il futuro delle persone dipenda dall'uso delle vocali all'interno dell'oroscopo per il loro segno zodiacale. La giornata è positiva se nell'oroscopo la frequenza media delle vocali è esattamente uguale alla frequenza media delle consonanti.

Si vuole dunque definire un predicato prolog che consenta di calcolare la frequenza media delle vocali e quella delle consonanti e di un altro che poi permetta di dire se una giornata è fortunata.

vocale('a').

vocale('e').

vocale('i').

vocale('o').

vocale('u').

lung([], 0).

lung([_|T], A):-

lung(T,B),

A is B+1.

nV([],0).

nV([E|T],M):-

vocale(E),!,

nV(T,N),

M is N+1.

nV([_|T],M):-

nV(T,M).

nC([],0).

nC([E|T],M):-

\+vocale(E),!,

nC(T,N),

M is N+1.

nC([_|T],M):-

nC(T,M).

calcolo(A,B):-

A = B,!,

write('Giornata fortunata').

calcolo(A,B):-

\+A=B,

write('Giornata sfortunata').

giornata(Segno):-

nV(Segno,A),

nC(Segno,B),

V is A/5,

C is B/16,

calcolo(V,C).

Predicati in Prolog: assert() e retract()

Definizione di un predicato:

In Prolog, un predicato è una regola che definisce una relazione tra un nome (il predicato stesso) e un certo numero di argomenti. Il comportamento del predicato è determinato da due componenti:

- **Fatti:** Sono affermazioni atomiche che descrivono lo stato.
- **Regole:** Sono istruzioni che definiscono come il predicato può essere utilizzato per derivare nuove informazioni. Le regole sono composte da una testa e un corpo. La testa è il predicato stesso, mentre il corpo è una serie di altri predicati che devono essere soddisfatti affinché la regola sia vera.

Modificando fatti e regole:

Modificando i fatti o le regole che definiscono un predicato, si può modificare il suo comportamento. Ad esempio, se aggiungiamo il fatto `lun([a],1)` alla nostra base di conoscenza, il predicato `lun` sarà ora in grado di riconoscere anche le liste con un solo elemento come liste lunari.

Predicati `assert()` e `retract()`:

- `assert()`: aggiunge un nuovo fatto alla base di conoscenza.
- `retract()`: rimuove un fatto dalla base di conoscenza.

Questi due predicati sono particolarmente utili per modificare dinamicamente il comportamento dei predicati durante l'esecuzione del programma.

`retractall()`, `assertZ` e `assertA` in Prolog

In Prolog, manipoliamo la base di conoscenza con predicati specifici. Vediamo cosa fanno `retractall()`, `assertZ()` e `assertA()` e come si differenziano.

`retractall()`

- Questo predicato rimuove tutte le clausole che corrispondono a uno schema specifico dal database di Prolog.

`assertz()` vs `asserta()`

Questi predicati vengono utilizzati per aggiungere clausole al database Prolog, ma differiscono in base alla posizione in cui inseriscono la nuova clausola:

- `assert(Fatto)` : Aggiunge il fatto in una posizione arbitraria nel database.
- `assertZ(Fatto)` : Aggiunge il fatto alla fine.
- `assertA(Fatto)` : Aggiunge il fatto all' inizio.

L'ordine delle clausole può talvolta influenzare il modo in cui Prolog esegue il programma. Quindi, `assertZ` e `assertA` forniscono un maggiore controllo sul posizionamento delle clausole rispetto al più semplice `assert`.

Dichiarazione di predicati dinamici in Prolog

Sintassi:

Prolog

```
:- dynamic(predicato/n)
```

Funzione:

- La direttiva `dynamic(predicato/n)` dichiara che il predicato `predicato/n` è dinamico.
- Un predicato dinamico può avere un numero variabile di clausole aggiunte o rimosse durante l'esecuzione del programma.

es: `:-dynamic(lungh/2)`

L'operatore `Univ` non è un predicato standard integrato in Prolog. Tuttavia, esistono alcuni modi per ottenere funzionalità simili a seconda di ciò che si desidera fare.

1. Utilizzare `..` (`Univ`) per la costruzione di termini:

L'operatore `..` in Prolog consente di costruire un termine a partire da una lista. Gli elementi della lista diventano il funtore (nome della funzione) e gli argomenti del termine.

`lungh(0,1)= .. [lungh,0,1] → [argomenti] → Generiamo un nuovo predicato!`

esempio con variabile :

$A = .. [lungh, 0, 1] \rightarrow$ Generiamo un nuovo predicato unificato ad una variabile.

SWISH:

```
?- assert(primo(a)).
```

```
true.
```

```
?- assert(primo(b)).
```

```
true.
```

```
?- listing(primo).
```

```
:- dynamic primo/1.
```

```
primo(a).
```

```
primo(b).
```

```
true.
```

```
?- assert(secondo(X):- primo(X)).
```

```
true.
```

```
?- secondo(a).
```

```
true.
```

```
?- |
```

Analisi e spiegazione dei predicati in sequenza:

1. `assert(primo(a))` e `assert(primo(b))`:

- Entrambi i predicati utilizzano `assert` per aggiungere nuovi fatti al database di Prolog.
- Il primo fatto afferma che "a" è un numero primo.
- Il secondo fatto afferma che "b" è un numero primo.

- Entrambi i predicati restituiscono `true` ad indicare che l'asserzione è stata eseguita correttamente.

2. `listing(primo):`

- Il predicato `listing` elenca tutti i fatti e le regole associate a un predicato specifico.
- In questo caso, elenca tutti i fatti relativi al predicato `primo`.
- L'output mostra che il database contiene due fatti: `primo(a)` e `primo(b)`.
- La prima riga `:- dynamic primo/1.` indica che `primo/1` è un predicato dinamico.

3. `assert(secondo(X):- primo(X)):`

- Questo predicato aggiunge una regola al database di Prolog.
- La regola definisce il predicato `secondo` come segue:
 - `secondo(X)` è vero se `X` è un numero primo.
- In altre parole, `secondo(X)` è vero se `primo(X)` è vero.
- Il predicato `assert` restituisce `true` ad indicare che l'asserzione è stata eseguita correttamente.

4. `secondo(a):`

- Questo predicato verifica se `"a"` è un numero secondo.
- Poiché `"a"` è un numero primo, la regola `secondo(X)` viene soddisfatta.
- Il predicato `secondo(a)` restituisce `true`.

Conclusione:

In questa sequenza di predicati, abbiamo:

- Aggiunto due fatti al database usando `assert`.
- Elencato tutti i fatti relativi a `primo` usando `listing`.
- Aggiunto una regola che definisce `secondo` in base a `primo`.
- Verificato se `"a"` è un numero secondo usando `secondo`.

?- primo(b)=..L.

L = [primo, b].

?- primo(b)=..[primo,b].

true.

?- primo(b)=..[primo,B].

B = b.

?- primo(b)=..[A,B].

A = primo,

B = b.

?- X=..[primo,B].

X = primo(B).

?- X=..[primo,B],Y=..[terzo,B],assert(Y:-X).

X = primo(B),

Y = terzo(B).

?- terzo(a).

true.

?- listing(terzo).

:- dynamic terzo/1.

terzo(A) :-

primo(A).

true.

?-

Analisi e spiegazione dei predicati in sequenza:

1. primo(b)=..L:

- Questo predicato unifica il termine primo(b) con la lista L.

- La lista `L` viene istanziata con `[primo, b]`, che rappresenta la scomposizione del termine `primo(b)` nel suo funtore (`primo`) e argomento (`b`).

2. `primo(b)=..[primo,b]:`

- Questo predicato verifica se il termine `primo(b)` è uguale alla lista `[primo, b]`.
- Poiché entrambi sono uguali, il predicato restituisce `true`.

3. `primo(b)=..[primo,B]:`

- Questo predicato unifica il termine `primo(b)` con la lista `[primo, B]`.
- La variabile `B` viene istanziata con `b`, che rappresenta l'argomento del termine `primo(b)`.

4. `primo(b)=..[A,B]:`

- Questo predicato unifica il termine `primo(b)` con la lista `[A, B]`.
- La variabile `A` viene istanziata con `primo` e la variabile `B` viene istanziata con `b`.

5. `X=..[primo,B]:`

- Questo predicato assegna alla variabile `x` il termine `primo(B)`.
- La variabile `B` può essere istanziata con un valore specifico o rimanere libera.

6. `X=..[primo,B],Y=..[terzo,B],assert(Y:-X):`

- Questa sequenza di predicati:
 - Assegna alla variabile `x` il termine `primo(B)`.
 - Assegna alla variabile `y` il termine `terzo(B)`.
 - Aggiunge una regola al database di Prolog: `terzo(B) :- primo(B)`.
- La regola implica che se `B` è un numero primo, allora `B` è anche un "terzo".

7. `terzo(a):`

- Questo predicato verifica se "a" è un "terzo".
- Poiché "a" è un numero primo (come definito dalla regola `terzo(B) :- primo(B)`), il predicato `terzo(a)` restituisce `true`.

8. `listing(terzo):`

- Questo predicato elenca tutti i fatti e le regole associate al predicato terzo.
- L'output mostra la regola `terzo(A) :- primo(A).`

Conclusione:

In questa sequenza di predicati, abbiamo:

- Scomposto un termine in una lista usando `=..`
- Verificato l'uguaglianza tra un termine e una lista.
- Estrapolato le variabili da un termine.
- Definito una regola che collega due predicati.
- Verificato se un elemento soddisfa la regola definita.
- Elencato tutti i fatti e le regole relative a un predicato.

Query: `funtore(A,Funtore).`

?- `assert(funtore(A,Funtore):-A=..[Funtore|_]).`
true.



?- `listing(funtore).`

`:- dynamic funtore/2.`


`funtore(A, B) :-`
`A=..[B|_].`

true.

Predicato Prolog per la funzione Fibonacci (NO DYNAMIC)

  Program × +

```
1 fibonacci(0, 0).  
2 fibonacci(1, 1).  
3 fibonacci(2, 1).  
4 fibonacci(N, M) :-  
5     N > 1,  
6     N1 is N - 1,  
7     fibonacci(N1, M1),  
8     N2 is N - 2,  
9     fibonacci(N2, M2),  
10    M is M1 + M2.
```




 fibonacci(15,X).

X = 610




Predicato Prolog per la funzione Fibonacci (DYNAMIC)

Composizione del predicato senza assert.

 Program  

```
1 :- dynamic f/2.  
2 fibonacci(0, 0).  
3 fibonacci(1, 1).  
4 fibonacci(2, 1).  
5 fibonacci(N, M) :-  
6   write(in),nl,  
7   N1 is N - 1,  
8   fibonacci(N1, M1),  
9   N2 is N - 2,  
10  fibonacci(N2, M2),  
11  M is M1 + M2.
```

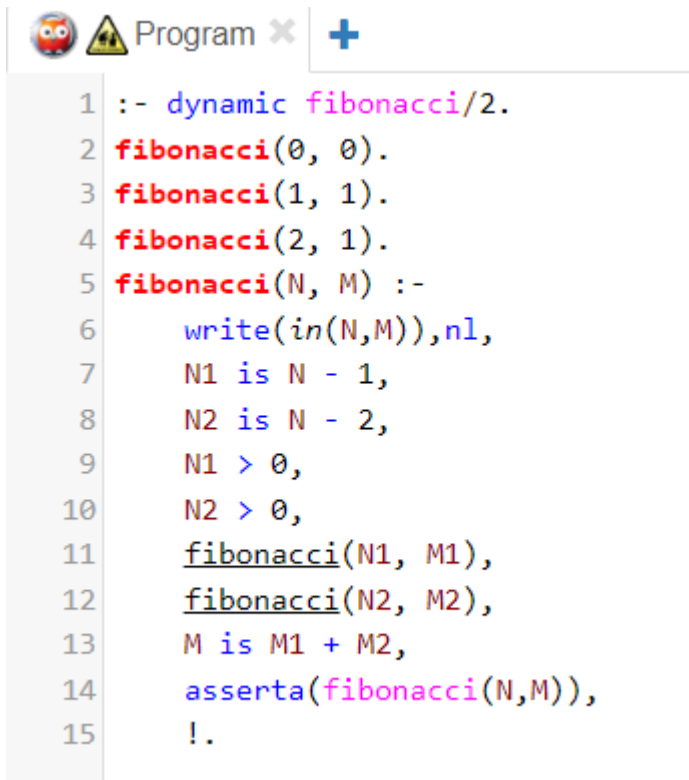
 fibonacci(6,M).

in
in
in
in
in
in
in
in
M = 8

Next 10 100 1,000 Stop

?- fibonacci(6,M).

Fibonacci (dynamic [assert])



```
1 :- dynamic fibonacci/2.
2 fibonacci(0, 0).
3 fibonacci(1, 1).
4 fibonacci(2, 1).
5 fibonacci(N, M) :-
6     write(in(N,M)),nl,
7     N1 is N - 1,
8     N2 is N - 2,
9     N1 > 0,
10    N2 > 0,
11    fibonacci(N1, M1),
12    fibonacci(N2, M2),
13    M is M1 + M2,
14    asserta(fibonacci(N,M)),
15    !.
```

Spiegazione del codice Prolog per il calcolo di Fibonacci in Italiano

Questo codice Prolog definisce un programma per calcolare i numeri di Fibonacci utilizzando la memorizzazione. Ecco una spiegazione dettagliata riga per riga:

1. `:- dynamic fibonacci/2.`

Questa riga dichiara il predicato `fibonacci/2` (un predicato a due argomenti che accetta due interi) come dinamico. Ciò significa che Prolog può aggiungere o rimuovere dinamicamente fatti (clausole) per calcolare i numeri di Fibonacci durante l'esecuzione del programma.

2. `fibonacci(0, 0).`

- **Caso base:** Questa clausola definisce il primo caso base per la sequenza di Fibonacci. Stabilisce che `fibonacci(0, 0)` è vera, ovvero il numero di Fibonacci di 0 è 0.

3. `fibonacci(1, 1).`

- **Caso base:** Questa clausola definisce il secondo caso base. Stabilisce che `fibonacci(1, 1)` è vera, indicando che il numero di Fibonacci di 1 è 1.

4. `fibonacci(2, 1).`

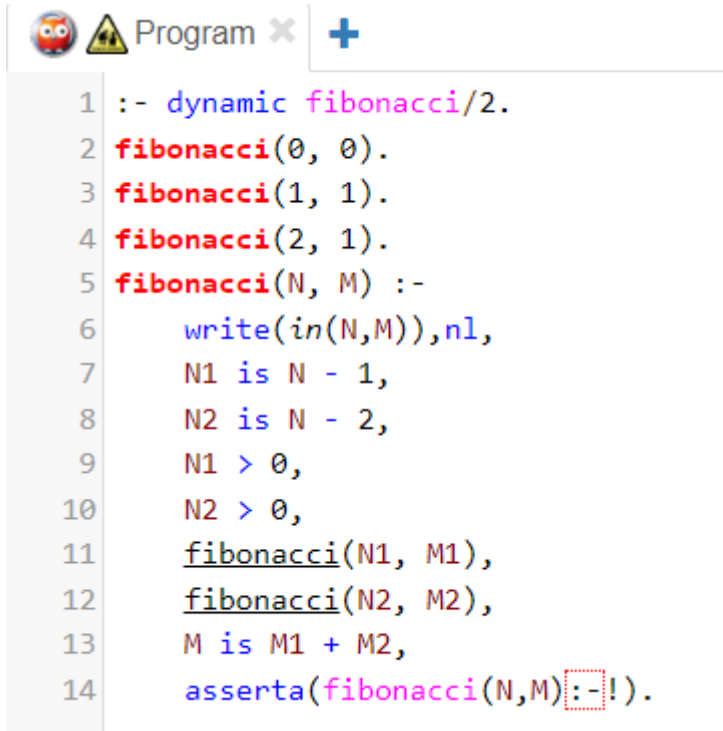
- **Predefinendo il numero di Fibonacci di 2 come 1**, si evita potenzialmente una chiamata ricorsiva aggiuntiva.

5. `fibonacci(N, M) :-`

- **Regola ricorsiva:** Questa clausola definisce la regola ricorsiva per calcolare il numero di Fibonacci. Stabilisce che `fibonacci(N, M)` è vera se valgono le seguenti condizioni:
 - `write(in(N, M)), nl` : Questa riga, pur non essenziale per la funzionalità principale, potrebbe essere utilizzata per scopi di debug. Stampa "in" seguito ai valori di input (N e M) e un carattere di a capo, indicando che la funzione viene chiamata con argomenti specifici.
 - `N1 is N - 1, N2 is N - 2` : Queste righe calcolano i due valori successivi più piccoli (N-1 e N-2) necessari per il calcolo di Fibonacci.
 - `N1 > 0, N2 > 0` (opzionale): Questi controlli garantiscono che N1 e N2 siano valori positivi. Questo può essere aggiunto per la gestione degli errori o per evitare chiamate ricorsive non necessarie per input negativi.
 - `fibonacci(N1, M1), fibonacci(N2, M2)` : Queste sono chiamate ricorsive allo stesso predicato `fibonacci`. Calcolano i numeri di Fibonacci per N-1 e N-2, memorizzando i risultati rispettivamente in M1 e M2.
 - `M is M1 + M2` : Questa riga calcola il numero di Fibonacci effettivo M sommando i risultati (M1 e M2) ottenuti dalle chiamate ricorsive. Questo è il calcolo principale per la sequenza di Fibonacci.
 - `asserta(fibonacci(N, M))` : Questa riga, utilizzando la dichiarazione dinamica precedente, inserisce un nuovo fatto `fibonacci(N, M)` nel database Prolog. Questa è l'essenza della memorizzazione, dove i risultati precedentemente calcolati vengono memorizzati per evitare calcoli ridondanti.
 - `!.` (Cut): L'operatore cut (!) viene utilizzato qui per impedire che vengano provate clausole alternative per lo stesso obiettivo

`fibonacci(N, M)`. Questo migliora l'efficienza terminando il processo di backtracking una volta trovata una clausola riuscita.

2) CUT predicato ASSERT:



```
1 :- dynamic fibonacci/2.
2 fibonacci(0, 0).
3 fibonacci(1, 1).
4 fibonacci(2, 1).
5 fibonacci(N, M) :-
6     write(in(N,M)),nl,
7     N1 is N - 1,
8     N2 is N - 2,
9     N1 > 0,
10    N2 > 0,
11    fibonacci(N1, M1),
12    fibonacci(N2, M2),
13    M is M1 + M2,
14    asserta(fibonacci(N,M):-!).
```

Spiegazione

Il codice implementa un approccio di programmazione dinamica per calcolare la sequenza di Fibonacci. Funziona in questo modo:

1. Casi Base:

- `fibonacci(0, 0)`: Il termine 0° della sequenza è definito come 0.
- `fibonacci(1, 1)`: Il 1° termine è definito come 1.
- `fibonacci(2, 1)`: Questo è un caso base aggiuntivo (non strettamente necessario per la sequenza di Fibonacci) che definisce esplicitamente il 2° termine come 1.

2. Induzioni strutturate:

- `fibonacci(N, M)` (per $N > 2$):
 - `write(in(N,M)),nl` (per scopi di debug): Questa riga stampa un messaggio che indica una chiamata a `fibonacci(N, M)`, insieme a un carattere di a capo. Puoi commentarla se il debug non è necessario.
 - `N1 is N - 1, N2 is N - 2`: Queste righe calcolano le posizioni dei due termini precedenti nella sequenza ($N1$ e $N2$).

- $N1 > 0, N2 > 0$: Questi controlli assicurano che $N1$ e $N2$ siano posizioni non negative, poiché gli indici negativi non avrebbero senso nella sequenza di Fibonacci.
- `fibonacci(N1, M1), fibonacci(N2, M2)`: Queste sono chiamate ricorsive a `fibonacci` per calcolare i numeri di Fibonacci nelle posizioni $N1$ e $N2$, memorizzando i risultati rispettivamente in $M1$ e $M2$.
- `M is M1 + M2`: Questa riga calcola il numero di Fibonacci nella posizione N sommando i risultati ($M1$ e $M2$) dalle chiamate ricorsive. Questa è la logica centrale della sequenza di Fibonacci.

3. Memoizzazione (Asserzione Dinamica):

- `asserta(fibonacci(N,M):-!)`: Questa riga utilizza l'asserzione dinamica per memorizzare il risultato (M) di `fibonacci(N, M)`. La memoizzazione consiste nell'archiviare i risultati calcolati in precedenza per evitare calcoli ridondanti. La direttiva `asserta` aggiunge un nuovo fatto `fibonacci(N, M)` al database Prolog, con l'operatore `:-!` che assicura che solo il primo fatto corrispondente venga asserito per motivi di efficienza. Questa ottimizzazione contribuisce a migliorare le prestazioni del codice, soprattutto per il calcolo di numeri di Fibonacci più alti.

% c:/Users/mrchp/Downloads/Nuovo Documento di testo.pl compiled 0.00 sec, 0 clauses

[1] ?- listing(fibonacci).

:- dynamic fibonacci/2.

fibonacci(0, 0).

fibonacci(1, 1).

fibonacci(2, 1).

fibonacci(N, M) :-

 write(in(N, M)),

 nl,

 N1 is N+ -1,

 N2 is N+ -2,

 N1>0,

 N2>0,

 fibonacci(N1, M1),

 fibonacci(N2, M2),

 M is M1+M2,

 asserta((fibonacci(N, M):-!)).

true.

QUERY SWI-PROLOG:

```
[1] ?- fibonacci(3,M).  
in(3,_3696)  
M = 2 ,
```

```
[1] ?- fibonacci(8,M).  
in(8,_6148)  
in(7,_6476)  
in(6,_6484)  
in(5,_6492)  
in(4,_6500)  
M = 21 |
```

Spiegazione degli output di Prolog:

1. fibonacci(3, M):

- La query `fibonacci(3, M)` chiede a Prolog di calcolare il 3° numero di Fibonacci e di assegnarlo alla variabile `M`.
- L'output mostra due righe:
 - `in(3,_3696)`: Indica che è stata fatta una chiamata a `fibonacci(3, _3696)`. Il trattino basso (`_`) indica che il valore di `M` non è ancora noto in questa fase.
 - `M = 2`: Mostra il risultato finale, ossia il 3° numero di Fibonacci è 2.

2. fibonacci(8, M):

- La query `fibonacci(8, M)` calcola l'8° numero di Fibonacci.
- L'output è più lungo e mostra le chiamate ricorsive effettuate durante il calcolo:
 - `in(8,_6148)`: Prima chiamata a `fibonacci(8, _6148)`.
 - `in(7,_6476)`: Chiamata ricorsiva a `fibonacci(7, _6476)`.
 - `in(6,_6484)`: Chiamata ricorsiva a `fibonacci(6, _6484)`.

- `in(5, _6492)`: Chiamata ricorsiva a `fibonacci(5, _6492)`.
- `in(4, _6500)`: Chiamata ricorsiva a `fibonacci(4, _6500)`.
- `M = 21`: Il risultato finale, l'8° numero di Fibonacci è 21.

Perché l'output per `fibonacci(8, M)` è più lungo?

- La sequenza di Fibonacci è definita ricorsivamente: ogni numero è la somma dei due numeri precedenti.
- Per calcolare l'8° numero, Prolog deve quindi calcolare i numeri 7, 6, 5 e 4.
- L'output mostra le chiamate ricorsive a `fibonacci` per questi numeri.

Cosa significa il trattino basso (`_`) nei nomi delle variabili?

- Il trattino basso (`_`) indica una variabile anonima, il cui valore non è importante in questo contesto.
- In questo caso, Prolog non ha bisogno di conoscere i valori intermedi di `M` durante le chiamate ricorsive.


```
[1] ?- listing(fibonacci).  
:- dynamic fibonacci/2.
```

```
fibonacci(8, 21) :-  
    !.  
fibonacci(7, 13) :-  
    !.  
fibonacci(6, 8) :-  
    !.  
fibonacci(5, 5) :-  
    !.  
fibonacci(4, 3) :-  
    !.  
fibonacci(3, 2) :-  
    !.  
fibonacci(0, 0).  
fibonacci(1, 1).  
fibonacci(2, 1).  
fibonacci(N, M) :-  
    !,  
    write(in(N, M)),  
    nl,  
    N1 is N+ -1,  
    N2 is N+ -2,  
    N1>0,  
    N2>0,  
    fibonacci(N1, M1),  
    fibonacci(N2, M2),  
    M is M1+M2,  
    asserta((fibonacci(N, M):-!)).
```

```
true.
```

STRUTTURA ALBOREA / PREDICATO leaf & node

Definire una struttura albero e un predicato leaf vero se: nella struttura c'è una foglia che gli diamo per nome.

Creiamo un predicato che crea un albero dove R è la root e Children sono i figli di essa.

Caso 1) : `t(R,Children):-`

Caso 2) : `R(Children):..[]`

Esempio per focalizzare l'iter induttivo:

`t(3,4)`

`somma(T,S)`

è vero se in T mettiamo `t(3,4)` e S abbiamo 7

`somma(t(X,Y),S):- S is X+Y.`

N.B leaf se è una foglia ; node se è un qualsiasi nodo

Procedimento in Prolog:

```
Program ✕ +
1 leaf(t(R,[]), R). % ritorna vero se R sono uguali e se R ha [] figli (solo le foglie non hanno figli)
2
3 leaf(t(_,Child),L):-
4     member(C,Child),
5     leaf(C,L).
6
7
8 node(t(R,_),R).
9
10 node(t(_,Child),L):- % ritorna vero se R sono uguali e se R ha _(qualsiasi cosa) figli
11     member(C,Child),
12     node(C,L).
```

[predicato member = predicato appartenere]

PREDICATO LEAF

`leaf/2`: Questo predicato controlla se un nodo è una foglia, cioè se non ha figli.

Ha due clausole:

- La prima clausola corrisponde al caso in cui un nodo ha una lista vuota di figli (`[]`). In questo caso, il predicato restituisce vero se il valore del nodo è uguale al valore passato come primo argomento.
- La seconda clausola controlla se uno dei figli del nodo è una foglia. Questo viene fatto iterando attraverso la lista dei figli (`Child`) e richiamando ricorsivamente il predicato `leaf/2` per ogni figlio.

PREDICATO NODE

`node/2`: Questo predicato identifica i nodi non foglia, cioè quelli che hanno almeno un figlio. Ha due clausole:

- La prima clausola corrisponde al caso in cui un nodo ha almeno un figlio. Restituisce vero se il valore del nodo è uguale al valore passato come primo argomento.
- La seconda clausola controlla se uno dei figli del nodo non è una foglia. Questo viene fatto iterando attraverso la lista dei figli (`Child`) e richiamando ricorsivamente il predicato `node/2` per ogni figlio.

In generale, l'albero binario è rappresentato come una struttura `t/2`, dove il primo argomento rappresenta il valore del nodo e il secondo argomento rappresenta una lista di figli. Le foglie sono rappresentate come nodi con una lista vuota di figli, mentre i nodi non foglia hanno almeno un figlio.

Albero di Interpretazione simboli matematici

$$2 + 3 \times 5 = ? \longrightarrow +(2, *(3,5))$$



La precedenza dei simboli è una prerogativa standard di regole (della grammatica) che andiamo a definire stilando le regole di precedenza dei simboli i quali godono della proprietà transitiva.

x	⊃	:	⊃	+	⊃	-
100	200	300	400			

Prolog

`op(priorità, operatore, nome) .`

Questo codice definisce un predicato chiamato `op` che ha tre argomenti:

- **priorità:** Un numero intero che rappresenta la priorità dell'operatore. I numeri più piccoli indicano una priorità più alta.
- **operatore:** Un simbolo che rappresenta l'operatore stesso. Ad esempio, '+' per l'addizione, '-' per la sottrazione, '*' per la moltiplicazione e '/' per la divisione.
- **nome:** Una stringa che rappresenta il nome dell'operatore. Ad esempio, "somma" per l'addizione, "differenza" per la sottrazione, "prodotto" per la moltiplicazione e "quoziente" per la divisione.

Questo predicato viene utilizzato per definire le regole di precedenza degli operatori in un'espressione Prolog. Le regole di precedenza determinano l'ordine in cui vengono valutate le operazioni in un'espressione.

Ad esempio, la seguente regola definisce che l'operazione di moltiplicazione ha una priorità più alta rispetto all'operazione di addizione:

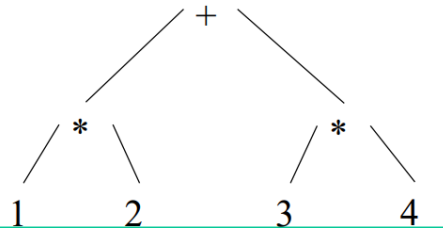
op(100, *, "prodotto").

op(200, +, "somma").

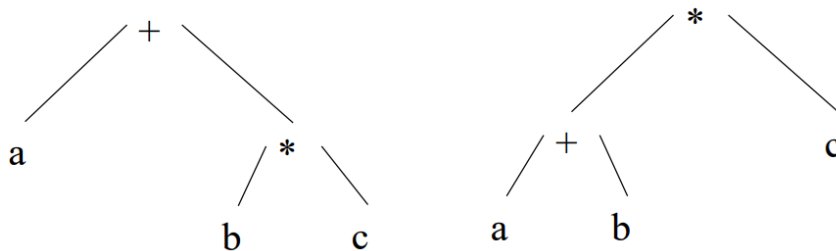
Con questa regola, l'espressione $2 * 3 + 1$ verrà valutata come $(2 * 3) + 1$, ovvero 7, e non come $2 + (3 * 1)$, ovvero 5.

Slide lezione 3 Drive ZNZ - (CHIARIMENTO)

- $1*2+3*4$ ha i due operatori + e *
- la scrittura in Prolog sarebbe:
 - $+(*(1,2), *(3,4))$



- Ogni operatore ha una sua priorità
- $a + b*c$ come deve essere letto?
 - $+(a, *(b,c))$?
 - $*(+(a,b), c)$?
- Nel senso comune trasmessoci, * lega di più di +,

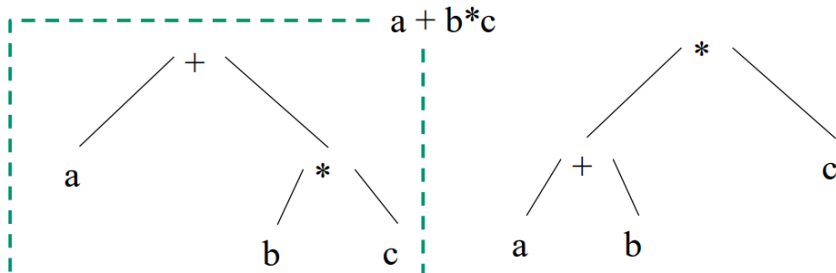


Codificare la priorità: l'albero delle interpretazioni ha priorità decrescenti

+ ha priorità 500

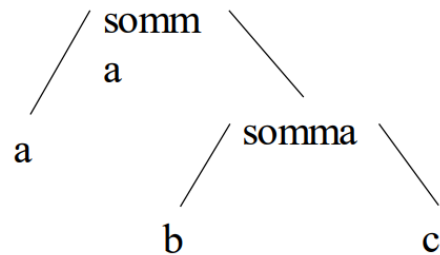
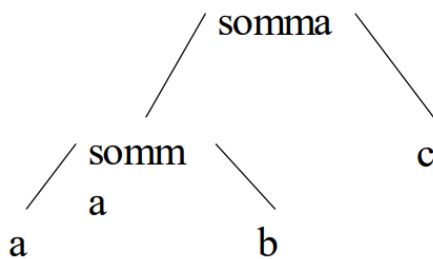
* ha priorità 400

(e quindi + ha priorità più alta di *)



- :- op(Priorità, Tipo, Operatore).
- Priorità è un numero tra 0 e 1200
- Tipo:
 - infisso : xfx, xfy, yfx
 - prefisso: fx, fy
 - postfisso: xf, fy
- Operatore: il nome/simbolo dell'operatore

- Il tipo serve ad indicare anche la precedenza degli operatori:
 - x : la sua priorità deve essere minore di quella dell'operatore
 - y: la sua priorità deve essere minore o uguale a quella dell'operatore
- :- op(700, yfx, somma).
- Qual è l'albero risultante di
 - 9 somma 5 somma 7 ?
- :- op(700, yfx, somma).
- 9 somma 5 somma 7



FINE SLIDE-

OPERATORE SOMMA (SWISH)

```

1
2 :- op(300,yfx,somma).
3
4 somma(t(X,Y,Z),S):-
5     S is X+Y+Z.
  
```

Spiegazione del codice Prolog

Il codice fornito definisce un predicato chiamato `somma` che calcola la somma di tre numeri. Ecco una spiegazione dettagliata del codice:

1. Direttiva `op`:

La prima riga del codice è una direttiva `op`. Questa direttiva indica al compilatore Prolog come interpretare un operatore specifico. In questo caso, la direttiva definisce che l'operatore `YFX` con precedenza 300 rappresenta l'operazione di somma.

2. Predicato `somma`:

La seconda riga del codice definisce un predicato chiamato `somma`. Questo predicato ha tre argomenti:

- `t(X, Y, Z)`: Il primo argomento è un termine composto che rappresenta una tupla contenente tre numeri. La lettera `t` indica un termine composto, e `X`, `Y` e `Z` rappresentano le variabili che contengono i valori numerici.
- `S`: Il secondo argomento è una variabile che conterrà la somma dei tre numeri.

3. Regola del predicato `somma`:

La terza riga del codice definisce una regola per il predicato `somma`. Questa regola specifica che la somma di tre numeri `X`, `Y` e `Z` può essere calcolata utilizzando l'operatore `YFX` (somma) e assegnando il risultato alla variabile `S`.

In altre parole, la regola dice: "Per calcolare la somma di tre numeri rappresentati dalla tupla `t(X, Y, Z)`, esegui l'operazione `X YFX Y YFX Z` e assegna il risultato alla variabile `S`."

OPERATORE “HA” (possesso) E “COSA”

```
8 :- op(100,yfx,di).
9 :- op(300,yfx,ha).
10
11 mario ha macchina di dario.
12 giovanni ha panino.
13 elena ha panino di giovanni.
14 giacomo ha borse di pelle di daino.
15
16 /* ? Chi ha cosa */
```



Chi ha Cosa

Chi = mario,

Cosa = macchina di dario

Chi = giovanni,

Cosa = panino

Chi = elena,

Cosa = panino di giovanni

Chi = giacomo,

Cosa = borse di pelle di daino

?-

Chi ha Cosa

Spiegazione del codice Prolog

Il codice fornito definisce due operatori personalizzati e quattro fatti che descrivono relazioni di possesso tra persone e oggetti. Ecco una spiegazione dettagliata del codice:

1. Direttive `op`:

Le prime due righe del codice sono direttive `op`. Queste direttive definiscono due operatori personalizzati:

- `op(100, yfx, di)`: Definisce l'operatore `di` con precedenza 100. Questo operatore rappresenta la relazione di possesso tra due entità.
- `op(300, yfx, ha)`: Definisce l'operatore `ha` con precedenza 300. Questo operatore rappresenta la relazione generica di possesso.

2. Fatti:

Le righe successive del codice definiscono quattro fatti che descrivono relazioni di possesso:

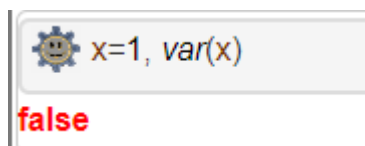
- `mario ha macchina di dario`: Questo fatto afferma che Mario possiede una macchina che appartiene a Dario. La parola `di` indica che la macchina è posseduta da Dario.
- `giovanni ha panino`: Questo fatto afferma che Giovanni possiede un panino.
- `elena ha panino di giovanni`: Questo fatto afferma che Elena possiede un panino che appartiene a Giovanni. La parola `di` indica che il panino è posseduto da Giovanni.
- `giacomo ha borse di pelle di daino`: Questo fatto afferma che Giacomo possiede borse fatte di pelle di daino. La parola `di` indica che la pelle di daino è il materiale di cui sono fatte le borse.

`var(X)` e `nonvar(X)` in Prolog

In Prolog, `var(X)` e `nonvar(X)` sono due predicati utilizzati per determinare lo stato di una variabile.

`var(X)`

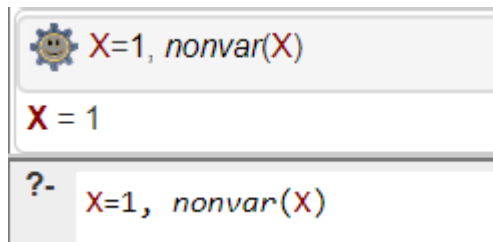
Il predicato `var(X)` verifica se la variabile `X` è non istanziata. In altre parole, controlla se `X` non è legata ad alcun valore concreto. Se `X` è non istanziata, `var(X)` avrà successo. Se `X` è legata ad un valore concreto, `var(X)` fallirà.



`nonvar(X)`

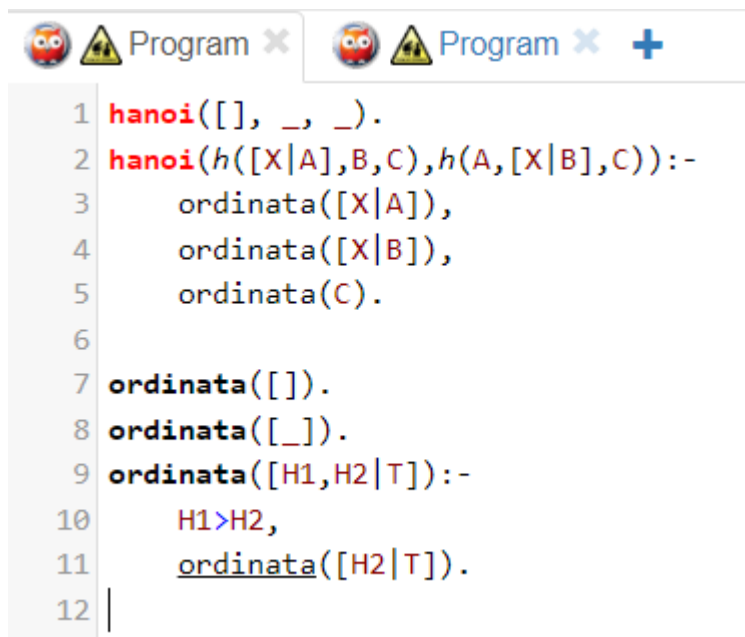
Il predicato `nonvar(X)` verifica se la variabile `X` è istanziata. In altre parole, controlla se `X` è legata ad un valore concreto, indipendentemente dal fatto che il valore sia

atomico o composto. Se X è istanziata, `nonvar(X)` avrà successo. Se X è non istanziata, `nonvar(X)` fallirà.



LA TORRE DI HANOI

Immaginiamo di ricostruire la rappresentazione dei nodi in maniera strutturata per il passaggio da uno stato all'altro per ricostruire le transizioni da un nodo ad un altro.



Spiegazione del codice Prolog per il problema di Hanoi

Il codice fornito implementa la soluzione ricorsiva del problema di Hanoi utilizzando il linguaggio di programmazione Prolog. Ecco una spiegazione dettagliata del codice:

Regole di Hanoi:

- `hanoi([], _, _)` .: Questa regola rappresenta il caso base della ricorsione. Indica che se la lista di dischi è vuota (`[]`), il problema è risolto.
- `hanoi(h([X|A], B, C), h(A, [X|B], C)) :-` Questa regola definisce la ricorsione per risolvere il problema di Hanoi. Essa indica che per spostare la pila di dischi `h([X|A])` da A a C, è necessario:
 - Verificare che le liste `[X|A]`, `[X|B]` e C siano ordinate in modo decrescente (cioè il disco più grande si trova in cima). Questo viene fatto utilizzando le regole `ordinata`.
 - Spostare il disco più grande (x) dalla pila A alla pila C. Questo viene fatto utilizzando la funzione `append` per combinare la lista `[X]` con la lista B, creando la nuova lista `[X|B]`.
 - Ricomporre la ricorsione chiamando `hanoi` sulla pila rimanente `h([X|A])` e sulle pile aggiornate `h(A, [], C)`.

Regole di ordinamento:


- `ordinata([])` .: Questa regola indica che una lista vuota (`[]`) è considerata ordinata.
- `ordinata([_])` .: Questa regola indica che una lista con un solo elemento (`[_]`) è considerata ordinata.
- `ordinata([H1, H2|T]) :- H1 > H2, ordinata([H2|T])` .: Questa regola verifica se una lista con due o più elementi (`[H1, H2|T]`) è ordinata. Se il primo elemento (H1) è maggiore del secondo elemento (H2), e la parte rimanente della lista (`[H2|T]`) è ordinata, allora la lista originale è considerata ordinata.

Funzionamento del codice:

Il codice inizia con la definizione della regola `hanoi` per il caso base (lista vuota). Quindi, la regola ricorsiva `hanoi` viene definita per spostare i dischi da una pila all'altra. La regola verifica l'ordinamento delle pile e sposta il disco più grande dalla pila di partenza alla pila di destinazione. La parte rimanente della pila viene ricomposta e la ricorsione continua fino a quando la lista di dischi è vuota (caso base).

Le regole `ordinata` vengono utilizzate per verificare se le liste di dischi sono ordinate in modo decrescente. Questo è necessario per garantire che i dischi siano posizionati correttamente secondo le regole del problema di Hanoi.

Struttura base del predicato del “BAGOF”



```
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:

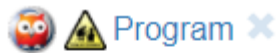
- Questo codice implementa una versione semplificata del predicato Bagof. Non gestisce le variabili e la molteplicità.
- Il predicato **appoggio/1** è un predicato dinamico, il che significa che la sua definizione può cambiare durante l'esecuzione del programma.
- Il predicato **fail** viene utilizzato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

•

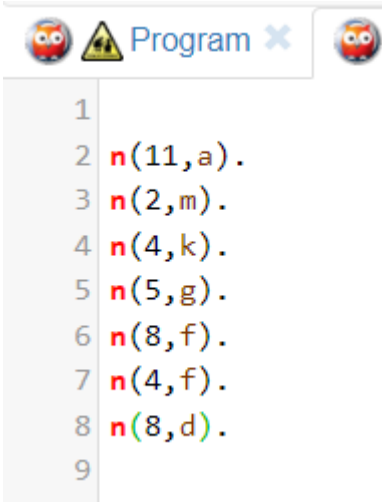
Output setof:

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

•

In sintesi:

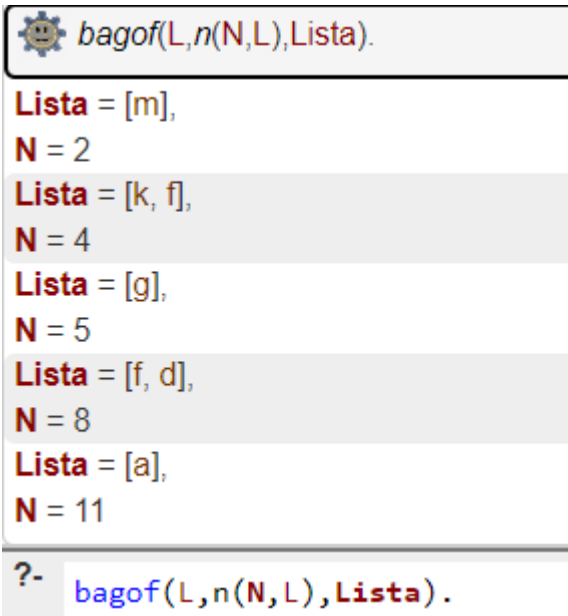
- **bagof** raccoglie tutti gli elementi che soddisfano l'obiettivo, inclusi i duplicati.
- **setof** raccoglie solo elementi univoci che soddisfano l'obiettivo e li ordina.



```

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 ×   Program × +

```
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.
- `controllo diagonale(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

```
1
2
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,RZ,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.

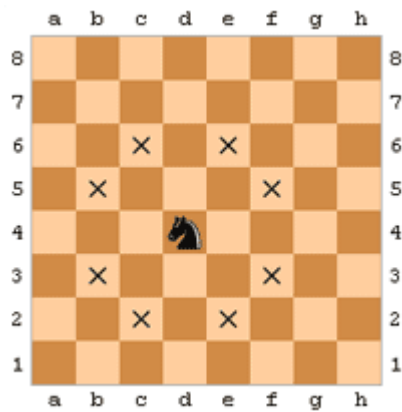
2. Ciclo di esplorazione:

- 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 $[X, PX]$).

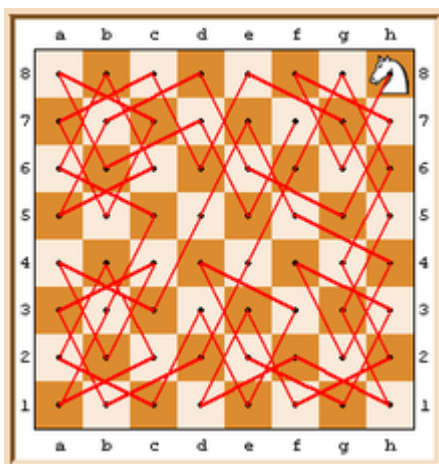
3. Risultato:

- 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_unicita`, verifica se una lista contiene elementi distinti. Funziona in modo ricorsivo:

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

2. Controllo soluzione

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

- `controllo_unicita_totale([_])` restituisce `true` se la lista contiene un solo elemento.
- `controllo_unicita_totale([M|T])` verifica se `M` è un elemento distinto dagli altri nella lista usando `controllo_unicita` e richiama ricorsivamente `controllo_unicita_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_unicita_totale` e `controllo_mosse`.

Definizione in italiano

In sintesi, il codice definisce due funzioni:

1. `controllo_unicita(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

```
Program ✕ +
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