

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.

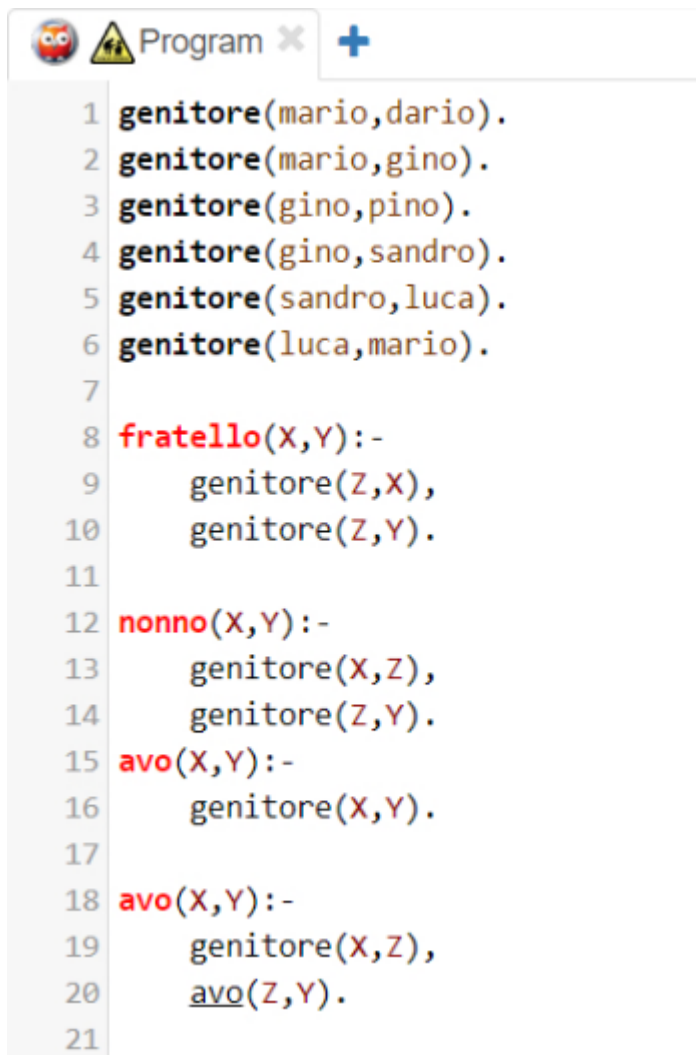


```
1 genitore(mario,dario).
2 genitore(mario,gino).
3 genitore(gino,pino).
4 genitore(gino,sandro).
5 genitore(sandro,luca).
6 genitore(luca,mario).
```

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.

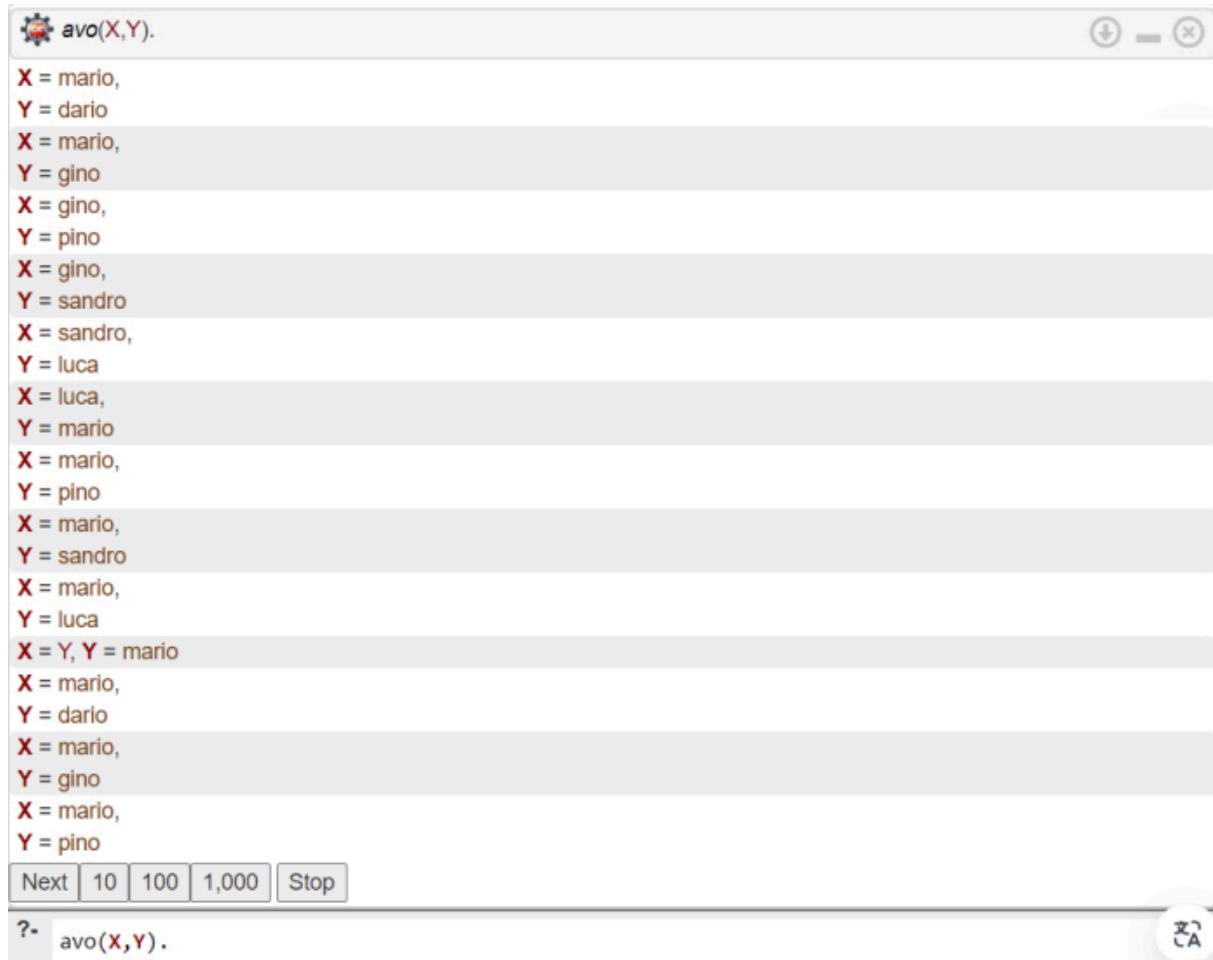
5. Caso dell'avo:

- 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).
```

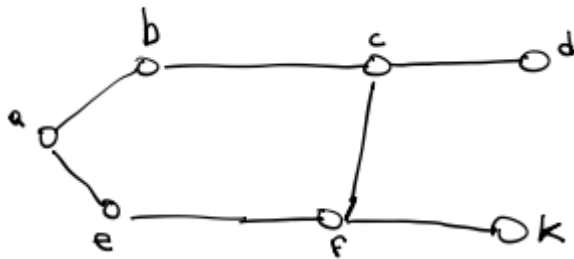
```
avo(X,Y):-  
    genitore(X,Z),  
    avo(Z,Y).
```

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.

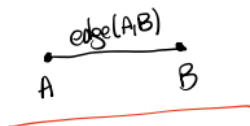
Prolog Swish per rappresentare il grafo utilizziamo questa forma:

```
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:

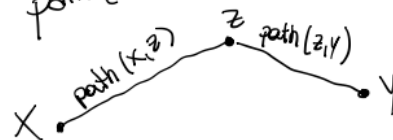
PASSO BASE:

\exists path(A,B) se esiste

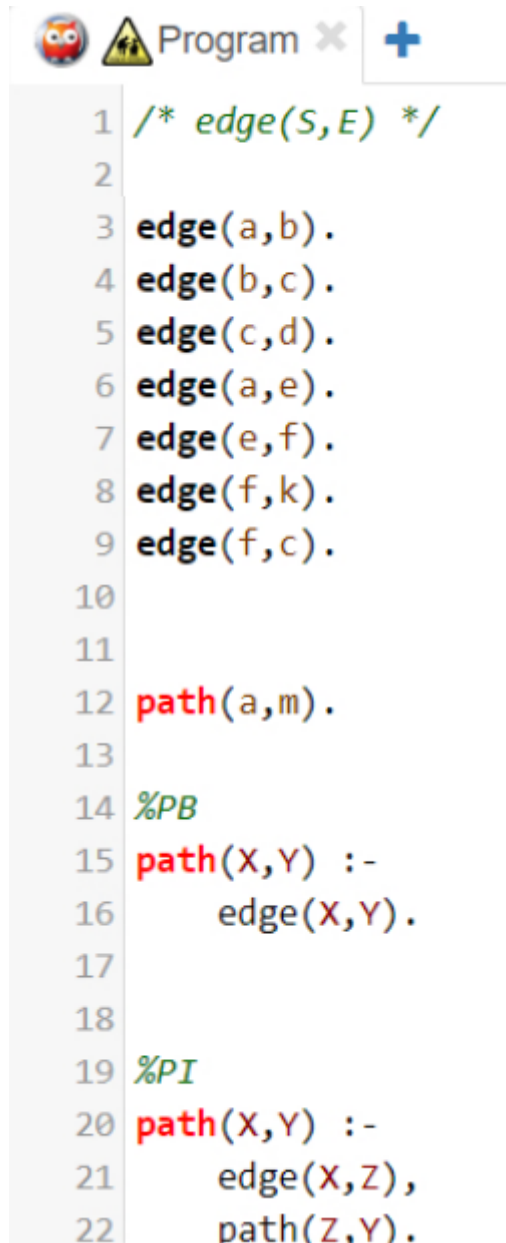


PASSO INDUTTIVO:

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([],[])`.

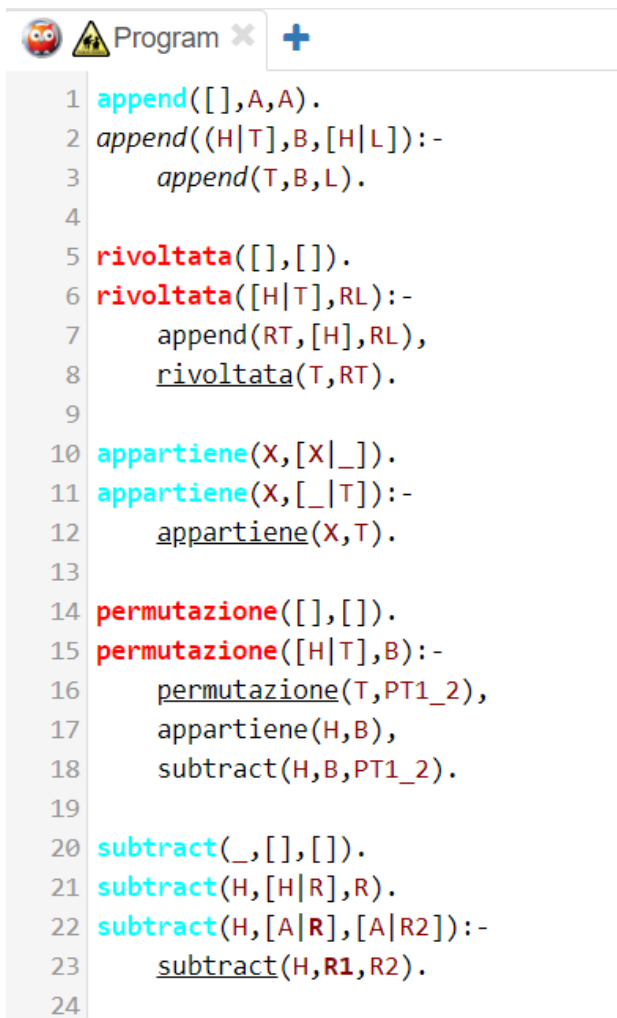
Questa clausola base specifica che se entrambe le liste sono vuote (`[]`), allora sono sicuramente permutazioni tra loro.

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.

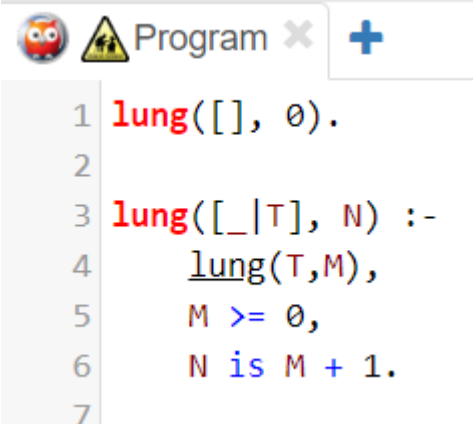
Predicato `lung([H|T], N):`

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.

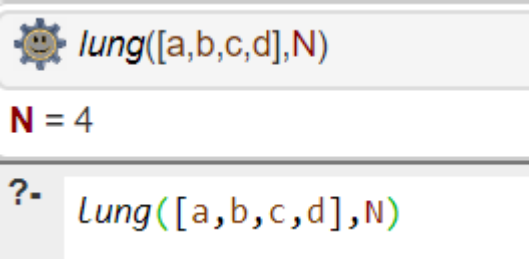
Il predicato utilizza la ricorsione per calcolare la lunghezza della lista:

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`).

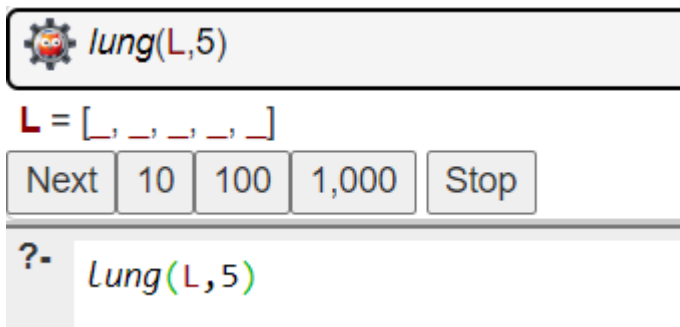


```
1 lung([], 0).
2
3 lung([_|T], N) :-
4     lung(T, M),
5     M >= 0,
6     N is M + 1.
7
```

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 $f(A)$.

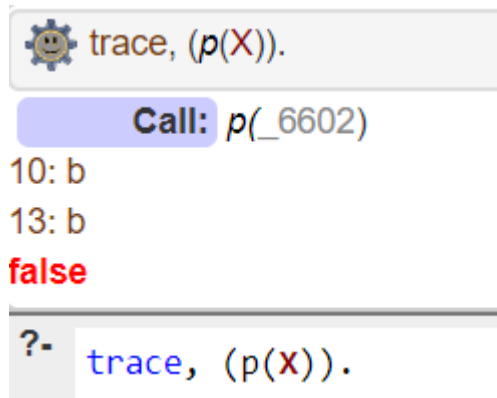
```
11 f(b).
12 g(a).
13 g(b).
14 g(j).
15 k(a).
16
17
18
19
20
21 p(A):-
22     f(A),
23     write('10: '),write(A),nl,
24     !,
25     g(A),
26     write('13: '),write(A),nl,
27     k(A).
28
```

```
⚙️ 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:

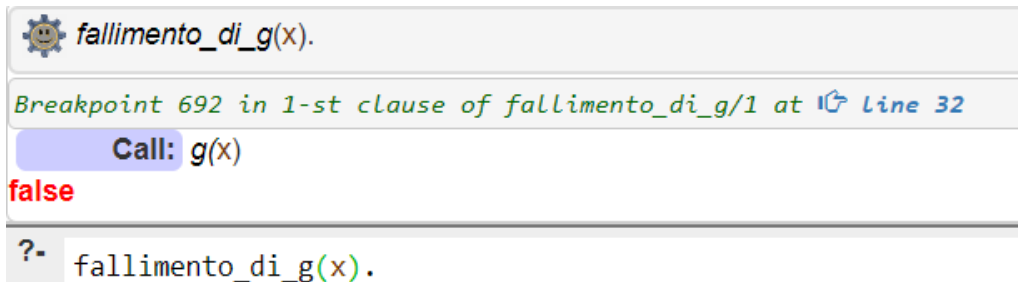
Comportamento:

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 debugger window for the predicate `fallimento_di_g(x)`. A breakpoint is set at line 32, which is the first clause of the predicate. The call being executed is `g(x)`, and the result is `false`. The prompt `?-` is followed by `fallimento_di_g(x).`

```
fallimento_di_g(x).
Breakpoint 692 in 1-st clause of fallimento_di_g/1 at Line 32
Call: g(x)
false
?- 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)`.

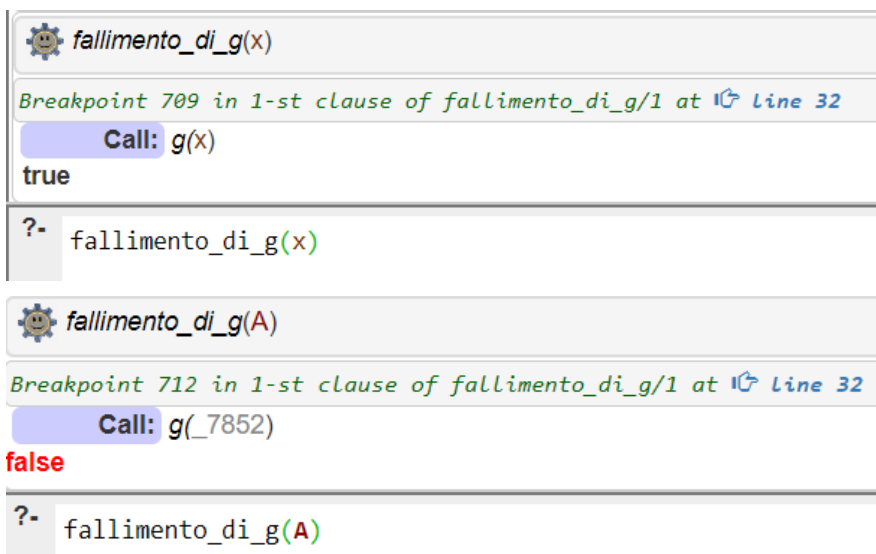
Analisi:

- **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.