

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.

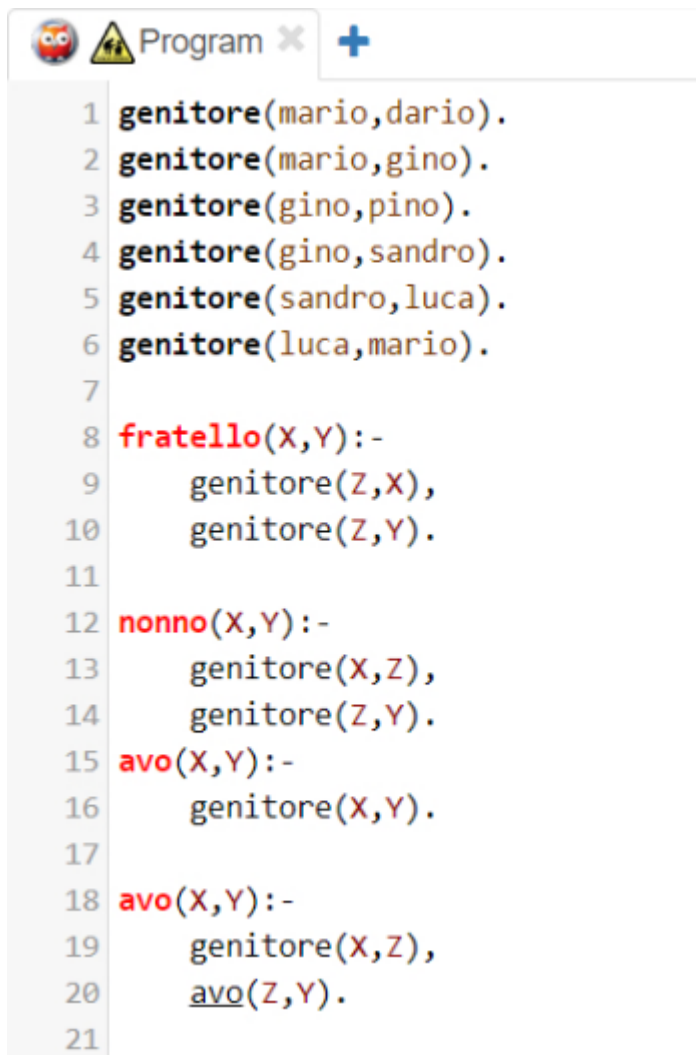


```
1 genitore(mario,dario).
2 genitore(mario,gino).
3 genitore(gino,pino).
4 genitore(gino,sandro).
5 genitore(sandro,luca).
6 genitore(luca,mario).
```

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)

```
fratello(X,gino).  
X = dario  
X = gino  
false
```

query) ?- avo (X,Y):

```
avo(X,Y).  
X = mario,  
Y = dario  
X = mario,  
Y = gino  
X = gino,  
Y = pino  
X = gino,  
Y = sandro  
X = sandro,  
Y = luca  
X = luca,  
Y = mario  
X = mario,  
Y = pino  
X = mario,  
Y = sandro  
X = mario,  
Y = luca  
X = Y, Y = mario  
X = mario,  
Y = dario  
X = mario,  
Y = gino  
X = mario,  
Y = pino  
Next 10 100 1,000 Stop  
?- avo(X,Y).
```

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

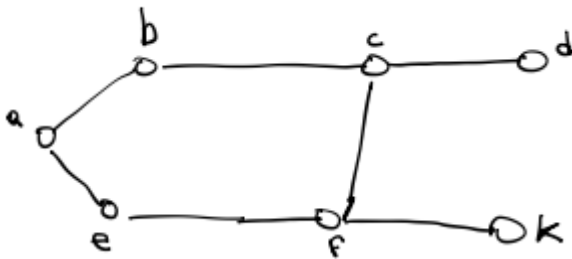
```
avo(X,Y):-  
    genitore(X,Z),  
    avo(Z,Y).
```

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.

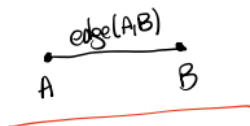
Prolog Swish per rappresentare il grafo utilizziamo questa forma:

```
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:

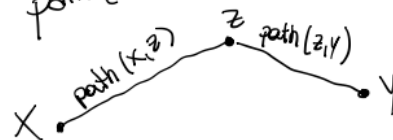
PASSO BASE:

\exists path(A,B) se esiste

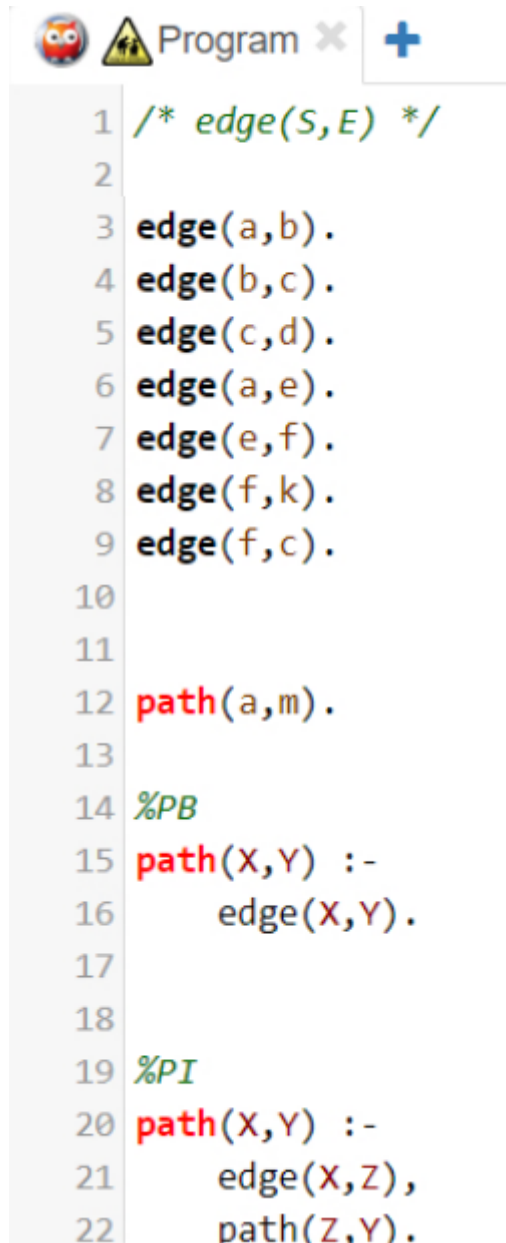


PASSO INDUTTIVO:

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

```
path(X, Y) :- edge(X, Z), 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([],[])`.

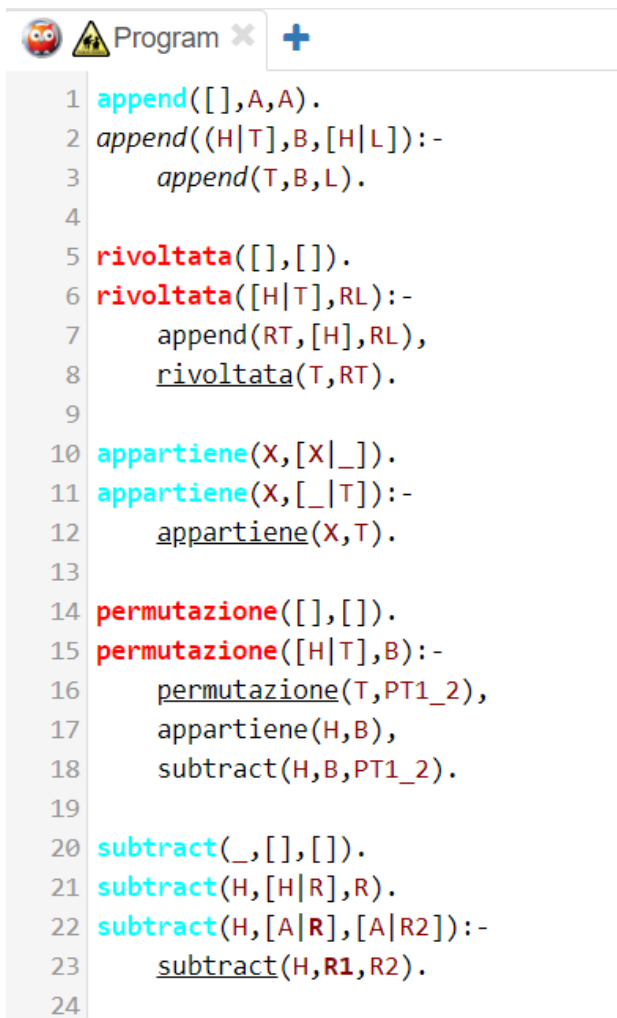
Questa clausola base specifica che se entrambe le liste sono vuote (`[]`), allora sono sicuramente permutazioni tra loro.

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.

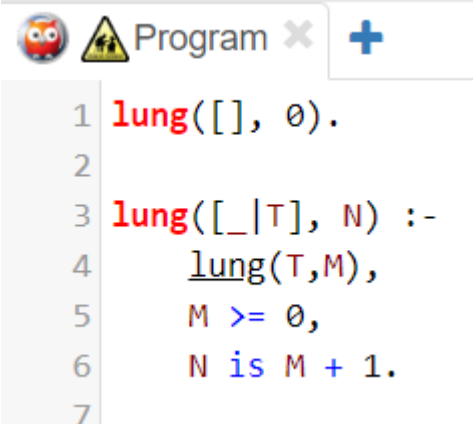
Predicato `lung([H|T], N):`

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.

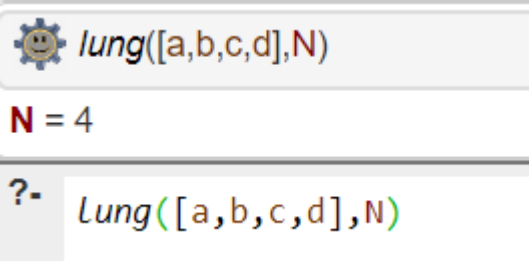
Il predicato utilizza la ricorsione per calcolare la lunghezza della lista:

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)
```


`lung(L,5)`

`L = [_, _, _, _, _]`

`?- lung(L,5)`

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)