

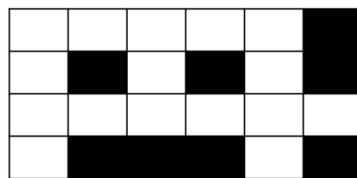
LINGUAGGI E METODOLOGIE DI PROGRAMMAZIONE

- **Modulo 2**

Esempio 1: Costruire un Cruciverba

Vogliamo costruire gli incastri di parole per un cruciverba. Per fare ciò, disegniamo uno schema e indichiamo le parole che possiamo usare. Il programma ci dirà quali parole usare e come riempire lo schema.

- Schema →



- Parole usabili → dog, four, baker, prolog, run, lost, forum, vanish, top, mess, green, wonder, five, unit, super, yellow.
- Soluzione possibile →

F	O	R	U	M	
I		U		E	
V	A	N	I	S	H
E				S	

- Possibile scrittura in Prolog:

```
word(d,o,g).  
word(r,u,n).  
word(t,o,p).  
word(f,o,u,r).  
word(l,o,s,t).  
word(m,e,s,s).  
word(u,n,i,t).  
word(f,i,v,e).  
word(b,a,k,e,r).  
word(g,r,e,e,n).  
word(f,o,r,u,m).  
word(s,u,p,e,r).  
word(p,r,o,l,o,g).  
word(v,a,n,i,s,h).  
word(w,o,n,d,e,r).  
word(y,e,l,l,o,w).
```

```

cruciverba(A1,A2,A3,A4,A5,B1,B3,B5,C1,C2,C3,C4,C5,C6,D1,D5) :-  

    word(A1,A2,A3,A4,A5),  

    word(A1,B1,C1,D1),  

    word(C1, C2, C3, C4, C5, C6),  

    word(A3,B3,C3),  

    word(A5,B5,C5,D5).  
  

?- cruciverba (A1, A2, A3, A4, A5, B1, B3, B5, C1, C2, C3, C4, C5, C6, D1,  

D5).

```

Avendo le seguenti scritture

- 1) $x = 2$
 $x = x + 1$
- 2) $\begin{cases} x + y = 0 \\ x + 3y = 2 \end{cases}$

Possiamo notare che la 1 è un esempio di scrittura di programmazione procedurale (assegnare variabili a valore), dove la variabile x è uno spazio di memoria con un nome (in questo caso x) dove operiamo. Qua non abbiamo soluzioni e cambia nel tempo.

Nel 2 possiamo notare che è un esempio di scrittura di programmazione dichiarativa (unificare variabili a valore), dove si presuppone che il valore delle variabili x e y va cercato e può essere “assegnato”. Qua si ha una soluzione che non cambia nel tempo.

Noi per studiare Prolog ci concentreremo sul linguaggio dichiarativo. Prolog è, anche, un linguaggio di programmazione logica. Infatti si basa su una restrizione della logica del primo ordine (Horn Clauses).

Il mondo di Prolog è fatto di Predicati. Infatti scrivere un programma Prolog significa scrivere un *insieme di predicati*. Questi predicati possono essere definiti da *fatti* e *regole*.

Un fatto è un predicato seguito da . e può essere composto da più termini

In Prolog abbiamo i seguenti simboli:

- la *e* la indichiamo con la *virgola* (,)
- la *o* la indichiamo con il *punto e virgola* (;)
- il *se* lo indichiamo con :-
- ogni regola termina con il *punto* (.)
- la *domanda* la indichiamo con ? – *condizione* (X).
- - (*underscore*) serve per non unificare due parti di una regola.
- *a* rappresenta una costante.
- *1, 2, 3, ...* rappresentano numeri.
- ‘ _ ‘ racchiude le costanti alfanumeriche.

ESERCIZIO 1)

X è fratello di Y se Z è genitore di X e di Y

I fatti sono:

genitore(Z, X). → Z è genitore di X (relazione di base)
genitore(Z, Y). → Z è genitore di Y (relazione di base)

Le Regole sono:

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

Dove: fratello(X, Y) è la **testa della regola**
:- separa la testa dal corpo della regola
genitore(Z, X) e genitore(Z, Y) è il **corpo della regola**
la **regola** dice che X è fratello di Y se hanno lo stesso
genitore Z

Domanda (o Query) :

?- fratello(x, y).

Possiamo unificare, cioè associo il nome alla variabile

? - fratello(X, Y).]
↓ ↓
DARIO GINO] DOMANDA

Fratello(X, Y):-
↓ ↓
DARIO GINO
genitore (Z, X),
↑ DARIO
genitore (Z, Y).
↓
GINO
HARLO } {
genitore (Z, dario)
genitore (Z, gino)
genitore (pino, gino)

N.B: 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".

Tutto ciò possiamo inserirlo su swish. Avremo:



```
Program +  
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
```

In Prolog i dati inseriti in input sono inseriti come fatti e regole:

- I fatti sono relazioni semplici tra due entità.
- Le regole definiscono relazioni più complesse, come la nozione di "fratello"

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

Possiamo inserire le seguenti Query:

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

Otteniamo:

- Query fratello:



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

- Query avo

The screenshot shows a Prolog query window with the following content:

```
?- 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
```

Below the results are navigation buttons: Next, 10, 100, 1,000, Stop. At the bottom left is the query `?- avo(X,Y).` and at the bottom right is a small icon.

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

The screenshot shows a Prolog editor with the following code:

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

8 fratello(X,Y):- 
9     genitore(Z,X),
10    genitore(Z,Y).

12 nonno(X,Y):- 
13     genitore(X,Z),
14     genitore(Z,Y).

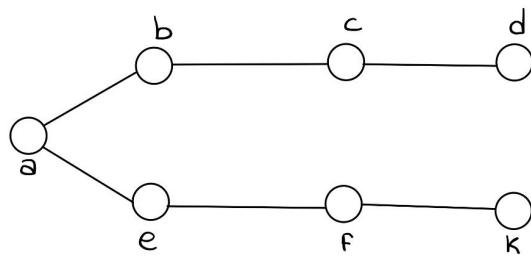
15 avo(X,Y):- 
16     genitore(X,Y). → PASSO BASE

18 avo(X,Y):- 
19     genitore(X,Z),
20     avo(Z,Y). → PASSO INDUTTIVO
```

The code defines predicates for parentage, siblinghood, and grandfatherhood. The `avo` predicate is defined with two rules: a base case (rule 15) and an inductive step (rule 18). Annotations with arrows point from the text to the respective rule numbers: "PASSO BASE" points to rule 15, and "PASSO INDUTTIVO" points to rule 18.

- **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.

ESEMPIO 2) Rappresentiamo un grafo in Prolog



Il grafo può essere rappresentato 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

Per definirlo in Prolog usiamo l'induzione. Questa può essere utilizzata per definire predicati più complessi, come la raggiungibilità di un nodo in un grafo.

```

    Program + 
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). → RAPPRESENTAZIONE PERCORSO
13
14 %PB → PASSO BASE
15 path(X,Y) :- edge(X,Y).
16
17
18
19 %PI → PASSO INDUTTIVO
20 path(X,Y) :- edge(X,Z),
21         path(Z,Y).
22

```

Definiamo meglio Passo Base e Passo Induttivo:

- **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:

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

Spiegazione:

- La regola dice che se esiste un arco edge (X, Y) nel grafo, allora c'è un percorso 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 edge (a, b). In questo caso, la regola 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:

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

Spiegazione:

- La regola dice che se esiste un arco edge (X, Z) che collega X a un nodo intermedio Z, e se esiste un percorso path (Z, Y) da Z a Y, allora esiste un percorso 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 edge(a, b) e edge(b, c). In questo caso, la regola path(a, c) è vera perché:

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

Quindi, la regola conclude che c'è un percorso path(a, c) da a a c.

Adesso osserviamo l'output di alcune query:

```
?- path(a, d)
```

```
path(a,d)
true
true
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
```

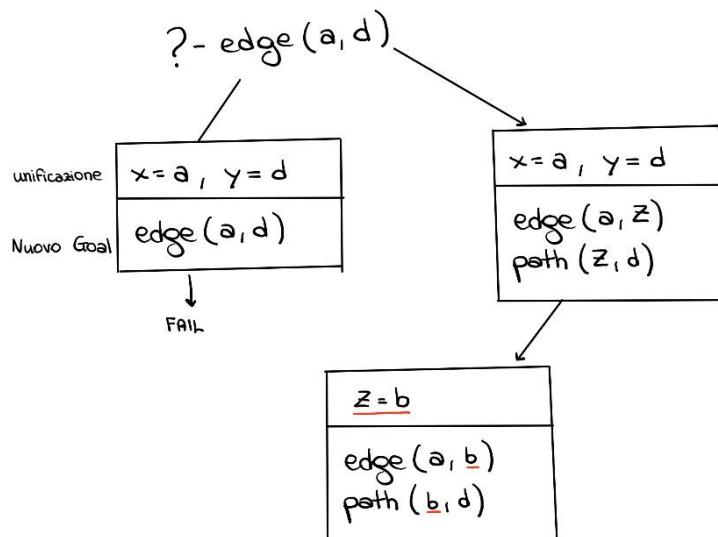
```
?- listing(path)
```

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

true

In Prolog si svolge tutto dall'alto verso il basso, da sinistra verso destra
 L'algoritmo di risoluzione in Prolog si chiama “Albero della dimostrazione”.

ESEMPIO) ecco l'algoritmo di risoluzione dell'esercizio precedente



Le **liste** in Prolog sono molto utilizzate. Useremo l'induzione strutturale, cioè un metodo per dimostrare proprietà di strutture ricorsive, come le liste. Una lista è una sequenza di vari elementi (anche ripetuti), che possono essere anche a loro volta delle liste.

ESEMPIO 1) [a, b, c, d, e]

ESEMPIO 2) [a, b, [c, d]]

La lista può essere vuota []

Ogni lista è composta da due parti [H | T]:

- H: è il primo elemento
- T: è il resto della lista (quindi a sua volta è una lista)

ESEMPIO 1) [H | T] = [a, b, c, d, e]

$$\begin{aligned} \text{dove: } H &= a \\ T &= [b, c, d, e] \end{aligned}$$

ESEMPIO 2) [H | T] = [a]

$$\begin{aligned} \text{dove: } H &= a \\ T &= [] \end{aligned}$$

ESEMPIO 3) [H | T] = [] → False

Operazioni sulle liste:

- `appartiene(X, L)` → significa “X appartiene a L”

ESERCIZIO 1)

Data una lista L, l'elemento x appartiene a L se è primo elemento, X appartiene a L se appartiene alla coda.

Possiamo inserirlo su swish. Avremo:

```
Program +  
1 appartiene(X,[X|_]).  
2  
3 appartiene(X,[_|T]) :-  
4     appartiene(X,T).
```

Alcune Query che possiamo inserire:

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

```
appartiene(X,[1,2,3])  
X = 1  
X = 2  
X = 3  
false
```

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

```
appartiene(3,[1,2,3])  
true
```

- concatena(A, B, C) → ciò significa che C sarà la lista formata dalla concatenazione della lista A e dalla lista B. Quindi $C = [H | L]$ dove H è il primo elemento della lista A = [H | T] e L è la concatenazione di T (coda di A) e B.

Per risolverlo con Prolog faremo:

- Passo Base: Se A è una lista vuota allora C è semplicemente B
- Passo di induzione strutturata: Se A ha un elemento testa H e una coda T allora C ha come primo elemento H e la coda di C la otteniamo concatenando T con la lista B.

```
Program +  
1 concatena([],A,A).  
2  
3 concatena([H|T],B,[H|L]) :-  
4     concatena(T,B,L).
```

Alcune Query che possiamo inserire:

```
?- concatena([1,2],[3,4],C)
```

concatena([1,2],[3,4],C)

C = [1, 2, 3, 4]

```
?- concatena([1,2],M,C)
```

concatena([1,2],M,C)

C = [1, 2|M]

```
?- concatena([],B,C)
```

concatena([],B,C)

B = C

```
?- concatena(X,B,C)
```

concatena(X,B,C)

B = C,
X = []

Next 10 100 1,000 Stop

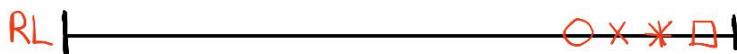
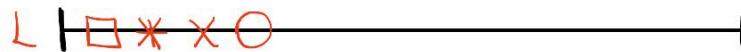
▪ Estrazione di più elementi

Una lista può contenere più elementi all'inizio, rappresentati usando la notazione [H₁, H₂ | 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]

- `rivoltata(L, RL)` → prende due argomenti L una lista generica e RL una lista vuota o che conterrà la lista L ribaltata



```

Program + 
1 %PB: Passo Base
2 rivoltata([],[]).
3
4 %PI: Passo Induttivo
5 rivoltata([H|T], RL):- 
6     append(RT, [H], RL),
7     rivoltata(T, RT).

```

Definiamo meglio Passo Base e Passo Induttivo:

- **Passo base (PB):**

Nel passo base, definiamo due liste inizialmente vuote, dove con una indichiamo L (la lista) e con l'altra indichiamo RL (la lista ribaltata)

- **Passo Induttivo (PI):**

Nel passo induttivo, definiamo che la lista di input viene divisa in testa H e coda T, se la funzione rivoltata viene poi chiamata ricorsivamente sulla coda T, e il risultato viene assegnato a RT. Infine, la testa H viene aggiunta alla fine di RT per formare la lista invertita RL.

Alcune Query possibili:

?- `rivoltata([a,b,c], [c,a,b]) .`

The window shows the query `rivoltata([a,b,c], [c,a,b]).` and the response `false`.

?- `rivoltata([a,b,c], X) .`

The window shows the query `rivoltata([a,b,c], X).` and the response `X = [c, b, a]`. There is also a small icon of a grave with 'RIP' written on it.

?- rivoltata([a,b,c], [b,c,a]).

```
rivoltata([a,b,c], [b,c,a]).
```

false

?- rivoltata([a,b,c], [c,b,a]).

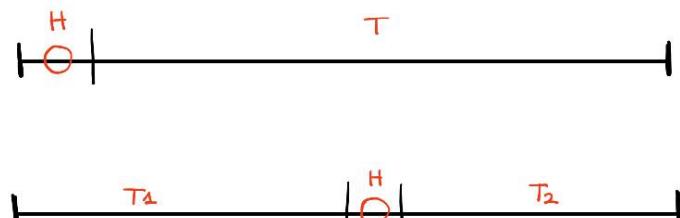
```
rivoltata([a,b,c],[c,b,a]).
```

true

Next 10 100 1,000 Stop

1

- permutazione (A, B) → data una lista la permutazione di questa lista è una lista che contiene tutti e soli gli elementi della lista originale. In altre parole, controlla se B contiene tutti gli stessi elementi di A, senza alcun duplicato e in qualsiasi ordine.



```

Program × +
```

```

1 append([],A,A).
2 append((H|T),B,[H|L]):- append(T,B,L).
3
4
5 rivoltata([],[]).
6 rivoltata([H|T],RL):- append(RT,[H],RL),
7     rivoltata(T,RT).
8
9
10 appartiene(X,[X|_]).
11 appartiene(X,[_|T]):- appartiene(X,T).
12
13
14 permutazione([],[]).
15 permutazione([H|T],B):- permutazione(T,PT1_2),
16     appartiene(H,B),
17     subtract(H,B,PT1_2).
18
19
20 subtract(_,[],[]).
21 subtract(H,[H|R],R).
22 subtract(H,[A|R],[A|R2]):- subtract(H,R1,R2).
23
24

```

Definiamo meglio Passo Base e Passo Induttivo:

- **Passo base (PB):**

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

- **Passo Induttivo (PI):**

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.

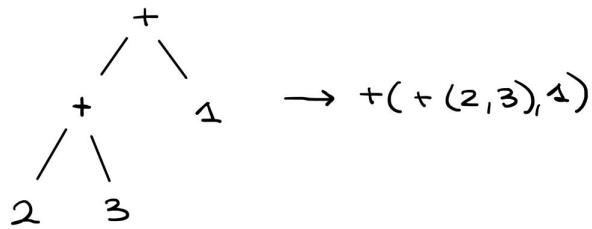
In Prolog possiamo fare dei calcoli aritmetici, ma non nella forma che conosciamo, cioè $4 = 3 + 1$ Prolog non lo comprende

The screenshot shows a Prolog interface with a query window and a results window. The query window contains the term `4 = 3+1`. The results window below it displays the word `false` in red text, indicating that the query did not succeed.

Infatti, per fare la seguente operazione bisogna usare l'operatore **is**

The screenshot shows a Prolog interface with a query window and a results window. The query window contains the term `4 is 3+1`. The results window below it displays the word `true` in green text, indicating that the query succeeded.

Alcuni operatori sono già definiti in Prolog, un esempio è il `+`, ma si scrivono nel seguente modo



Per calcolare la lunghezza di una lista posso fare la seguente operazione:

- `lung(L, Lung)` → dove L è la lista e Lung è il valore della lunghezza della lista

Il codice sarà scritto nel seguente modo:

```

1 %PB:Passo Base
2 lung([], 0).
3
4 %PI:Passo Induttivo
5 lung([_|T], N):-
6     lung(T, M),
7     N is M +1.

```

Definiamo meglio Passo Base e Passo Induttivo:

- **Passo base (PB):**

Questo passo base specifica che abbiamo una funzione chiamata lunghezza che è formata da una lista vuota la cui lunghezza è 0.

- **Passo Induttivo (PI):**

Nel passo induttivo, invece, stiamo definendo che la lunghezza di una lista viene divisa in testa H e coda T, se la lunghezza della coda T è incrementata di 1.

Alcune Query possibili:

?- `lung([a, b, c, 3, 4, 5], N).`

lung([a,b,c,3,4,5],N)

N = 6

```
?- lung([a,b,c],4).
```

The screenshot shows a Prolog interface with a toolbar at the top featuring icons for download, minimize, and close. The main area displays the query `lung([a,b,c],4)` and the response `false`.

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

The screenshot shows a Prolog interface with a toolbar at the top featuring icons for download, minimize, and close. The main area displays the query `lung([a,b,c,d],4)` and the response `true`. A green number '1' is visible in the bottom right corner.

- `numerodiEl(L, El, N)` → dove L è la lista, El è il numero della lista ed N è il numero di volte che El si trova nella lista.

The screenshot shows a Prolog editor window titled "Program". It contains the following code:

```
1 %PB: Passo Base
2 numerodiEl([], _, 0).
3
4 %PI: Passo Induttivo
5 numerodiEl([El|T], El, N):-_
6     numerodiEl(T, El, M),
7     N is M +1.
8 numerodiEl([X|T], El, M):-_
9     X \= El,
10    numerodiEl(T, El, M).
```

Definiamo meglio Passo Base e Passo Induttivo:

- **Passo base (PB):**

Questo passo base specifica che avendo una funzione chiamata `numerodiEl` che è formata da una lista vuota, da un'occorrenza non specificata (infatti c'è `_`), e 0 che indica la lunghezza della lista.

- **Passo Induttivo (PI):**

Nel passo induttivo, invece, è definito da 2 regole:

Nella prima regola abbiamo l'elemento attuale della lista (indicato con `El`) corrisponde all'elemento cercato (il secondo `El`), se incrementando di 1 cerca pure nella coda della lista (indicata con `T`) l'elemento `El`.

Nella seconda regola, indichiamo il caso in cui l'elemento corrente della lista (X) non è uguale all'elemento che stiamo cercando.

Alcune possibili Query:

? - numerodiEl([1,1,1],1,N).

numerodiEl([1,1,1],1,N)

N = 3

Next 10 100 1,000 Stop

? - numerodiEl([1,1,1],2,N).

numerodiEl([1,1,1],2,N)

N = 0

? - numerodiEl([1,2,2],2,N).

numerodiEl([1,2,2],2,N)

N = 2

Next 10 100 1,000 Stop

Ecco alcuni predicati: not, fail, cut

Il predicato **NOT** (`=/=_`) viene utilizzato per verificare se qualcosa non è vero. Quindi se io in Prolog ho un fatto *P*, `not (P)` mi verifica se il goal P fallisce.

ES: se io ho il seguente fatto

Program +

1 amici(mario,maria).

Posso avere le seguenti Query:

?- amici(mario, maria). → in questa query verifico se c'è un'amicizia tra una persona chiamata Mario e un'altra chiamata Maria, in base al fatto che ho Prolog mi restituirà True



amici(mario,maria).



true

1

?-amici(mario,lucia). → stessa cosa accade nella seguente query



amici(mario,lucia).



false

Se voglio esprimere che un fatto è falso avrò:

?- not(amici(mario,maria)).



not(amici(mario,maria)).



false

Invece, se P fallisce, il not (P) mi indicherà un successo. Infatti avrò:

?- not(amici(mario,lucia)).



not(amici(mario,lucia)).



true

1

In una query, posso esprimere il not pure nel seguente modo:



mario \= lucia.



true

1

Il **CUT** (rappresentato dal seguente simbolo !) è un operatore che taglia un ramo della ricerca (*backtracking*) nell'esecuzione di un programma Prolog. Una volta che viene eseguito, non si può tornare indietro su quel ramo. Quindi quando abbiamo cose da verificare non vere ci serve il cut.

ES1)

```
1 %fatti
2 q(a).
3 q(b).
4 q(c).
5 r(a).
6 r(b).
7
8 %regola
9 p(X):-
```

10 **not**(q(**X**)),
11 !,
12 r(**X**).

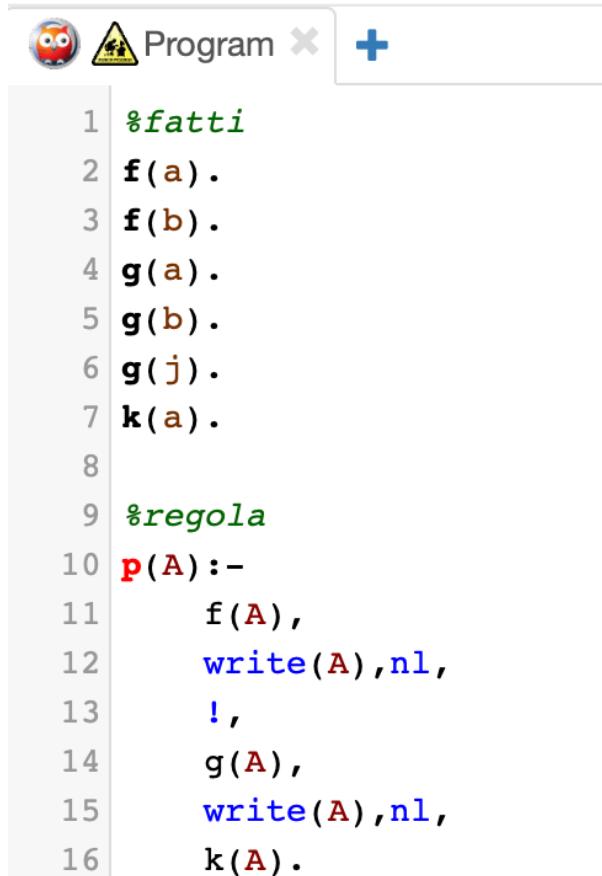
?- p(X) .

```
p(X).
```

false

In questo caso la regola ci dice che `p(X)` è vero per un certo `X` se il valore `X` non soddisfa `q(X)`. Se `q(X)` fallisce, il cut impedisce a prolog di cercare ulteriori soluzioni, e la regola `r(X)` viene soddisfatta.

ES2)



```
1 %fatti
2 f(a).
3 f(b).
4 g(a).
5 g(b).
6 g(j).
7 k(a).
8
9 %regola
10 p(A):-
```

11 **f**(**A**) ,
12 **write**(**A**) , **nl** ,
13 !,
14 **g**(**A**) ,
15 **write**(**A**) , **nl** ,
16 **k**(**A**) .

```
?- p(X).
```

The screenshot shows a window titled "p(X)". The output area contains three lines of text: "a", "a", and "X = a".

Il predicato **FAIL** viene utilizzato per forzare Prolog a fallire e a esplorare un altro ramo della ricerca.

ES)

The screenshot shows a Prolog program in a code editor. The code is as follows:

```
1 f(a).
2 f(b).
3 g(a).
4 g(b).
5 g(j).
6 k(a).
7
8 p(A) :-
9     f(A),
10    write('10: '), write(A), nl,
11    !,
12    g(A),
13    write('13: '), write(A), nl,
14    k(A).
15
16 fallimento_di_g(A) :-
17     g(A), fail.
```

```
?- fallimento_di_g(j).
```

The screenshot shows a window titled "fallimento_di_g(j)". The output area contains the word "false" in red.

```
?- fallimento_di_g(x) .
```

The screenshot shows the SWI-Prolog debugger interface. At the top, it says "fallimento_di_g(X)." with a gear icon. Below that, a message says "Breakpoint 704 in 1-st clause of fallimento_di_g/1 at line 1". A number "7" is also present. The stack trace is displayed in a list:

- Call: g(_7390)
- Exit: g(a)
- Call: fail
- Fail: fail
- Redo: g(_688)
- Exit: g(b)
- Call: fail
- Fail: fail
- Redo: g(_688)
- Exit: g(j)
- Call: fail
- Fail: fail
- Fail: fallimento_di_g(_464)

The word "false" is highlighted in red at the bottom.

ES2)

The screenshot shows the SWI-Prolog code editor. At the top, it has tabs for "Program" and a "+" button. The code is as follows:

```
1  f(a).  
2  f(b).  
3  g(a).  
4  g(b).  
5  g(j).  
6  k(a).  
7  
8  p(A):-  
9    f(A),  
10   write('10: '), write(A), nl,  
11   !,  
12   g(A),  
13   write('13: '), write(A), nl,  
14   k(A).  
15  
16 fallimento_di_g(A):-  
17   g(A), fail.  
18  
19 fallimento_di_g(_).
```

```
?- fallimento_di_g(X) .
```

The screenshot shows the SWI-Prolog debugger interface again. At the top, it says "fallimento_di_g(X)." with a gear icon. Below that, the word "true" is displayed. In the top right corner, there is a small green number "1".

```
?- fallimento_di_g(X) .
```

```
fallimento_di_g(A).
```

true

Mynot, si comporta come il not, cioè se il predicato è vero allora mynot(predicato) è falso

```
Program +  
1 f(a).  
2 f(b).  
3 g(a).  
4 g(b).  
5 g(j).  
6 k(a).  
7  
8 p(A):-  
9     f(A),  
10    write('10: '), write(A), nl,  
11    !,  
12    g(A),  
13    write('13: '), write(A), nl,  
14    k(A).  
15  
16 fallimento_di_g(A):-  
17     g(A), fail.  
18  
19 fallimento_di_g(_).  
20  
21 mynot(Predicato):-  
22     Predicato, !, fail.  
23 mynot(_).
```

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

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

false

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

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

true

- `numerodiEl(X, L, N)` → X è l'elemento da contare, L è la lista in cui cercare l'elemento e N è il numero di occorrenze trovate.

Il predicato `numdiEl` o `num_elementi` in Prolog serve a contare il numero di occorrenze di un elemento specifico all'interno di una lista. Questo predicato è utile per determinare quante volte un determinato elemento compare all'interno di una lista.

```

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) .
```



The screenshot shows a Prolog interface with the following output:

```

num_elementi(a,[a,b,a,k],N).
N = 2
false
```

ESERCITAZIONE - 28 Marzo 2024

Testo esercizio: 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 in 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.

Svolgimento:

C'è una separazione netta tra programma e dati, un programma può essere un dato di un altro programma

È possibile in prolog cambiare il programma nel tempo. Il nostro predicato in prolog è determinato da 2 componenti: fatti e regole. Se possiamo modificare questi (cioè fatti e regole), possiamo modificare il programma in tempo reale, cioè possiamo cambiare il significato di un predicato.

Per modificare dinamicamente il comportamento dei predicati durante l'esecuzione del programma possiamo usare: assert () e retract ()

assert() → aggiunge un nuovo fatto alla base di conoscenza

assertZ(Fatto) → aggiunge il fatto alla fine

assertA(Fatto) → aggiunge il fatto all'inizio

retract() → rimuove un fatto

retractAll() → rimuove tutti i fatti unificati tra loro

retractA() → rimuovere un fatto dall'inizio

retractZ() → rimuovere un fatto dalla fine

Per dichiarare che un predicato può essere modificato dinamicamente durante l'esecuzione del programma useremo dynamic.

`:– dynamic(predicato/n) →` dichiara che per un predicato posso asserire o ritrattare

In Prolog, l'operatore *univ* viene indicato con il seguente simbolo = .. e lo usiamo per decomporre o costruire termini.

Per decomporre un termine in Prolog, si utilizza la seguente sintassi

Termine = .. *Lista*

In questo modo, il termine viene decomposto in una lista di elementi, dove il primo elemento è il funtore (nome della funzione) e gli elementi successivi sono gli argomenti del termine.

%Esempio: Decomposizione di un termine

`lun(0, 1) =.. Lista.`

% Output atteso: `Lista = [lun, 0, 1]`

Per costruire un termine a partire da una lista di elementi, si utilizza la seguente sintassi

Termine = . . [Funtore | Argomenti]

In questo modo, il funtore e gli argomenti specificati nella lista vengono combinati per formare un nuovo termine.

% Costruzione di un termine

Termine = . . [funtore, arg1, arg2].

% Output atteso: Termine = funtore(arg1, arg2)

L'operatore *univ* consente di manipolare dinamicamente termini in Prolog, sia decomponendoli che costruendoli a partire da liste di elementi.

Il comando *listing* in Prolog, viene mostrato l'elenco dei fatti e delle regole dei predicati definiti durante la sessione corrente. Questo comando fornisce una panoramica dei predicati presenti nel programma, consentendo di visualizzare i dettagli delle definizioni dei predicati, inclusi i fatti e le regole associati a ciascun predicato.

ESEMPIO) *Non fatto da lui a lezione*



```
1 %fatti
2 mammifero(elefante).
3 mammifero(leone).
4 mammifero(cane).

6 %regola
7 predatore(X) :- mammifero(X), X \= cane.
```

? - listing (mammifero).

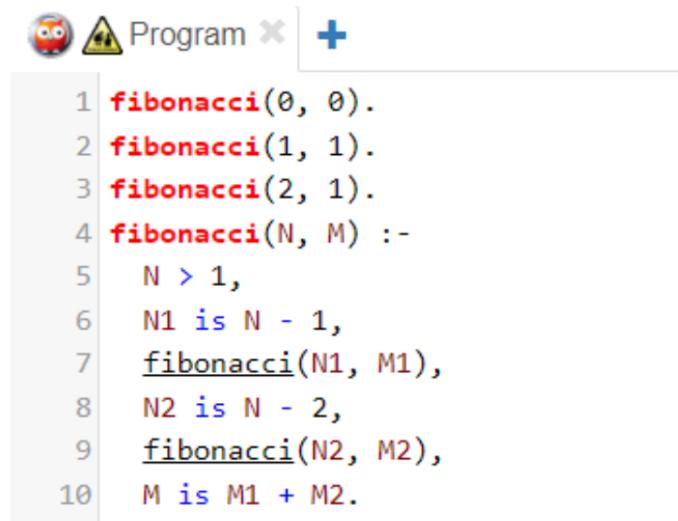


```
listing(mammifero).
mammifero(elefante).
mammifero(leone).
mammifero(cane).
```

Fibonacci: Programmazione Dinamica

Implementiamo il calcolo della sequenza di Fibonacci utilizzando la programmazione dinamica in Prolog.

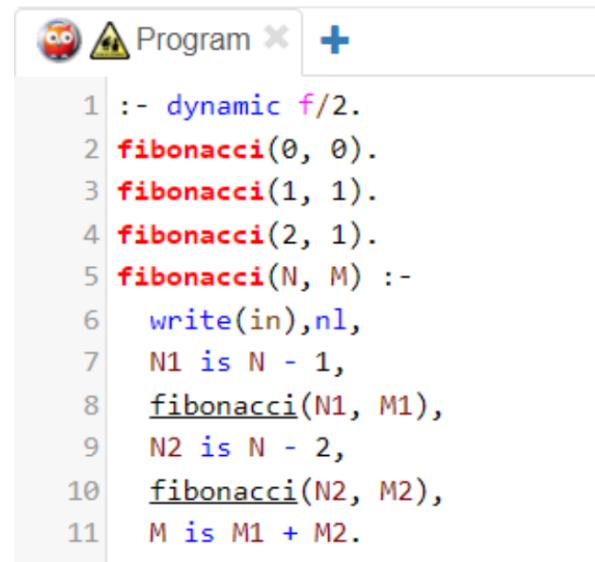
- 1° codice: no dynamic



A screenshot of a Prolog IDE window titled "Program". The code editor contains the following Prolog code:

```
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.
```

- 2° codice: dynamic



A screenshot of a Prolog IDE window titled "Program". The code editor contains the following Prolog code:

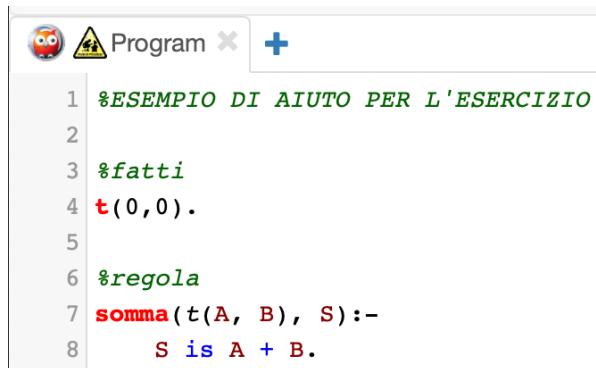
```
1 :- dynamic f/2.
2 fibonacci(0, 0).
3 fibonacci(1, 1).
4 fibonacci(2, 1).
5 fibonacci(N, M) :- 
6     write(in), nl,
7     N1 is N - 1,
8     fibonacci(N1, M1),
9     N2 is N - 2,
10    fibonacci(N2, M2),
11    M is M1 + M2.
```

ESERCIZIO) Vogliamo costruire una struttura arborea ed un predicato foglia

`leaf(T, F)` → è vero se T è un albero e F è una foglia (*leaf*)
`node(T, N)` → è vero se N è un nodo dell'albero T

`t(R, Children)` → Descrive la struttura di un albero, dove R è la radice e Children è una lista di elementi dell'albero.

`R(Children) = . . []` → Spiega come utilizzare l'operatore *univ* per decomporre la struttura dell'albero in Prolog.



```
1 %ESEMPIO DI AIUTO PER L'ESERCIZIO
2
3 %fatti
4 t(0,0).
5
6 %regola
7 somma(t(A, B), S):-
```

S is A + B.

? - `somma(t(3,4), S).`

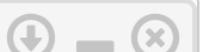
 `somma(t(3,4), S).`



S = 7

? - `somma(t(3,4), 9).`

 `somma(t(3,4), 9).`



false

? - `somma(t(3,4), 7).`

 `somma(t(3,4), 7).`



true

1

Avendo un albero t con nodi A, B → `t(A, B)`; posso dire che la regola somma i nodi dell'albero e ti restituisce tale somma in output → `somma(t(A, B), S) :- S is A+B`



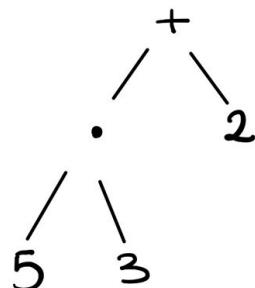
Program

```
1 %PASSO BASE_1: dove R, cioè la radice, è la mia unica foglia
2 leaf(t(R, []), R).
3
4 %PASSO INDUTTIVO_1: devo verificare se la foglia è un figlio dell'albero
5 leaf(t(_, Children), L) :- 
6     member(C, Children),
7     leaf(C, L).
8 %member dice che C appartiene al figlio
9 %e poi si unifica C con L
10
11 %PASSO BASE_2
12 node(t(R, _), R).
13
14 %PASSO INDUTTIVO_2
15 node(t(_, Children), L) :- 
16     member(C, Children),
17     node(C, L).
```

Operatori in Prolog

Noi tutti sappiamo che $2 + 3 \times 5 = 17$, poiché sappiamo che la precedenza c'è l'ha la moltiplicazione e quindi andremo a fare prima 3×5 e poi al risultato sommiamo 2.

In prolog scriviamo quest'operazione nel seguente modo +(2,*(3,5))



La precedenza dei simboli in Prolog la andremo a definire noi con delle regole.

x	▷	:	▷	+	▷	-
100	200	300		400		

Per definirla andremo ad usare il seguente predicato:

op (PRIORITÀ, TIPO, NOME)

Dove:

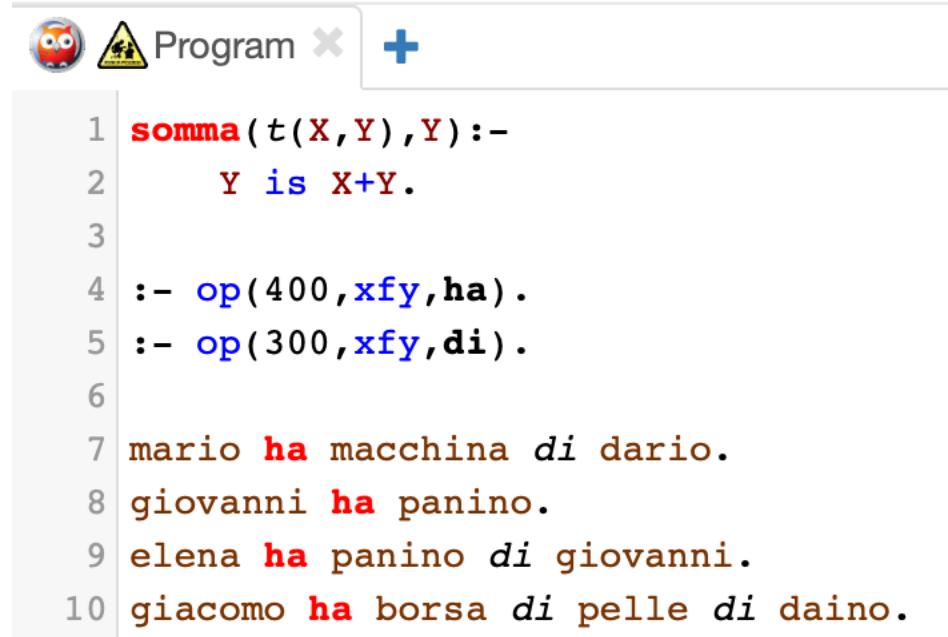
- Priorità è un numero intero che rappresenta la priorità dell'operatore, i numeri più piccoli indicano priorità più alta.
- Tipo: esistono vari tipi possibili:
 - fx, fy = si usa per indicare operatori unari prefissi
 - xf, yf = si usa per indicare operatori unari postfissi
 - xfx, yfx, xfy = si usa per operatori binari

N.B: f è la posizione dell'operatore e x e y sono l'argomento

Quindi ogni operatore in Prolog è caratterizzato da:

- Un nome
- Un numero di argomenti (1 o 2)
- Una posizione: prefissa, infissa, postfissa
- Precedenza
- Associatività

Vediamo il seguente codice:



```
1 somma(t(X,Y),Y):-  
2     Y is X+Y.  
3  
4 :- op(400,xfy,ha).  
5 :- op(300,xfy,di).  
6  
7 mario ha macchina di dario.  
8 giovanni ha panino.  
9 elena ha panino di giovanni.  
10 giacomo ha borsa di pelle di daino.
```

- Possibile Query: ? - Chi ha Cosa.



Chi ha Cosa.

Chi = mario,
Cosa = macchina didario

Chi = giovanni,
Cosa = panino

Chi = elena,
Cosa = panino digiovanni

Chi = giacomo,
Cosa = borsa dipelle didaino

- var(x)

Il predicato `var(x)` è usato per testare se l'argomento x è stanziato con una variabile

- nonvar(X)

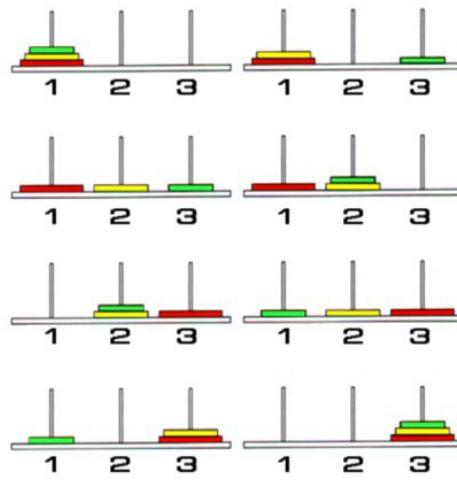
Il predicato `nonvar(X)` è usato per verificare se l'argomento X è istanziato con un termine che non è una variabile. Può essere considerato il complementare del predicato `var`.

Torre di Hanoi: Ordinare una lista

Il problema delle torri di Hanoi è un problema matematico/logico che deriva da un’antica leggenda. Questo “rompicapo” è composto da un insieme di “torri” sulle quali sono sovrapposte una serie di dischi. Il “rompicapo” consiste nel trasferire tutti i dischi da una “torre” all’altra seguendo alcune regole, come:

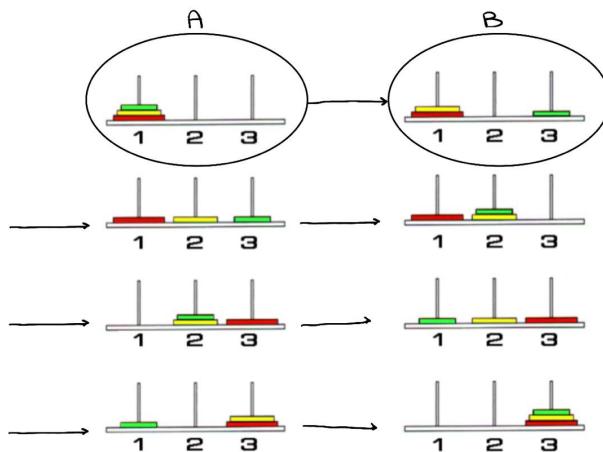
- Si può spostare solo un disco alla volta
- Si può spostare solo il disco in cima alla “torre”
- Non si può sovrapporre un disco grande sopra ad un disco più piccolo

ESEMPIO) Ecco un esempio di torre di Hanoi considerando 3 “torri”

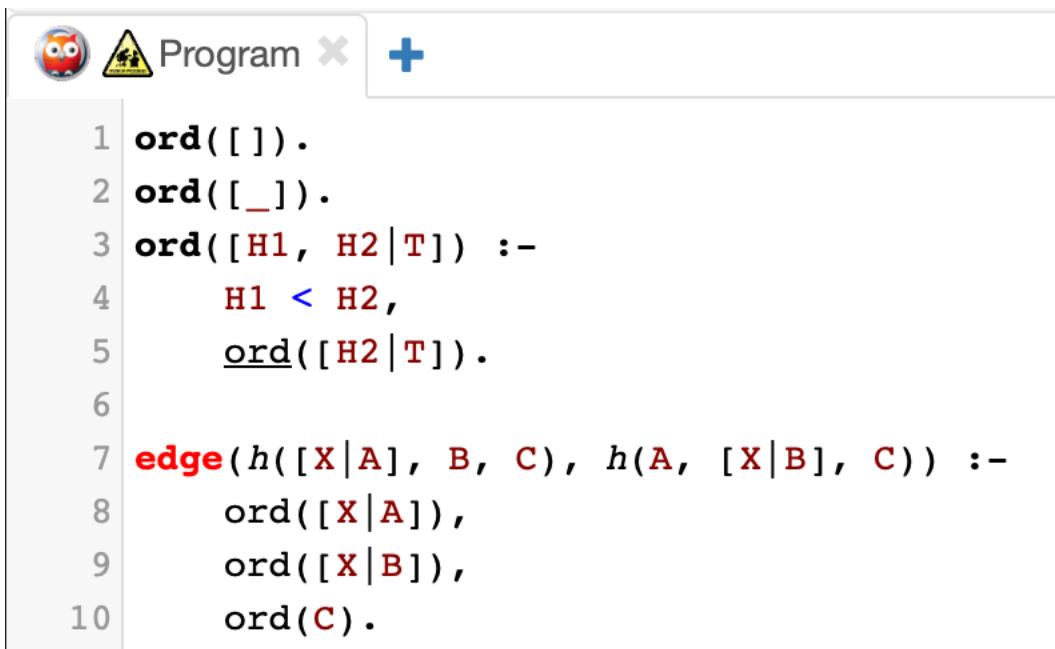


Posso implementare in Prolog il seguente “rompicapo”

Considero le Torri come nodi di un grafo:



Implementando avremo il seguente codice:



```
1 ord([ ]) .  
2 ord([_]) .  
3 ord([H1, H2|T]) :-  
4     H1 < H2,  
5     ord([H2|T]) .  
6  
7 edge(h([X|A], B, C), h(A, [X|B], C)) :-  
8     ord([X|A]),  
9     ord([X|B]),  
10    ord(C) .
```

Avendo un nodo (edge) formato da 3 torri (A,B,C) dove x è il “disco” da spostare da un’altra ad un’altra. Il nodo è definito così se x è ordinato in modo crescente.

La regola sopra ci fa capire quando una lista è ordinata.

- ? - `edge(h([1,2,3],[],[]), h([2,3],[1],[])) .`

 `edge(h([1,2,3],[],[]), h([2,3],[1],[])).`



true

[Next](#) [10](#) [100](#) [1,000](#) [Stop](#)