

## Appunti di Linguaggi di Programmazione

### Linguaggio - Prolog

#### Domande e risposte

#### 1) Definire i seguenti termini: Definizione, Assioma, Ipotesi e Teorema

→ Una **definizione** caratterizza e descrive le proprietà che distinguono un oggetto di interesse dagli altri oggetti.

**Esempio:** Sia  $n$  un naturale, tale  $n$  può essere

- **pari**: numeri interi divisibili per 2;
- **dispari**: numeri interi non pari;
- **primi**: numeri divisibili soltanto per 1 e sè stessi.

→ Un **assioma** è un principio che è considerato vero senza la necessità di dimostrarlo. Costituisce il punto di partenza per lo sviluppo e lo studio di una disciplina formale.

**Esempio:**

- 0 non è successore di nessun numero naturale;
- ogni numero naturale ha al massimo 1 predecessore;
- la **geometria euclidea** si basa su cinque assiomi.

→ Un'**ipotesi** è una proposizione considerata temporalmente vera durante il processo di dimostrazione. L'**ipotesi** è fondamentale per l'**induzione**, ma anche utile in dimostrazioni dove ci sono diversi casi da analizzare.

→ Un **teorema** è una conseguenza logica degli **assiomi**. Per appurare che una determinata proposizione sia un teorema, è necessario dimostrarla (**dimostrazione**).

I teoremi sono anche chiamati **lemma**, **corollario** e **proposizione**.

#### 2) Cosa si intendono per proposizione, formula e logica proposizionale?

→ Una **proposizione** è un'affermazione che può essere vera o falsa.

→ Una **formula** è un'espressione booleana che combina diverse proposizioni e può anch'essa assumere valori di vero e falso in base ai valori di verità delle proposizioni che la compongono.

→ La **logica proposizionale** è una branca della matematica che si occupa di studiare formule e i valori di verità.

#### 3) Definizione generale di formula ben formata, assegnazione e valutazione

Siano:

- **A** un insieme **non vuoto** di **proposizioni atomiche**;
- **Op1** un insieme di operatori (connettivi) unari ( $\neg$ );

• **Op2** un insieme di operatori binari ( $\wedge, \vee$ ).

L'insieme  $F$  di formule ben formate su  $(A, Op_1, Op_2)$  è definito **ricorsivamente**:

- $A \subseteq \mathcal{F}$  (ogni proposizione atomica è una fbf)
- se  $F \in \mathcal{F}$  e  $*$   $\in Op_1$  allora  $(*F) \in \mathcal{F}$
- se  $F, G \in \mathcal{F}$  e  $\circ \in Op_2$  allora  $(F \circ G) \in \mathcal{F}$

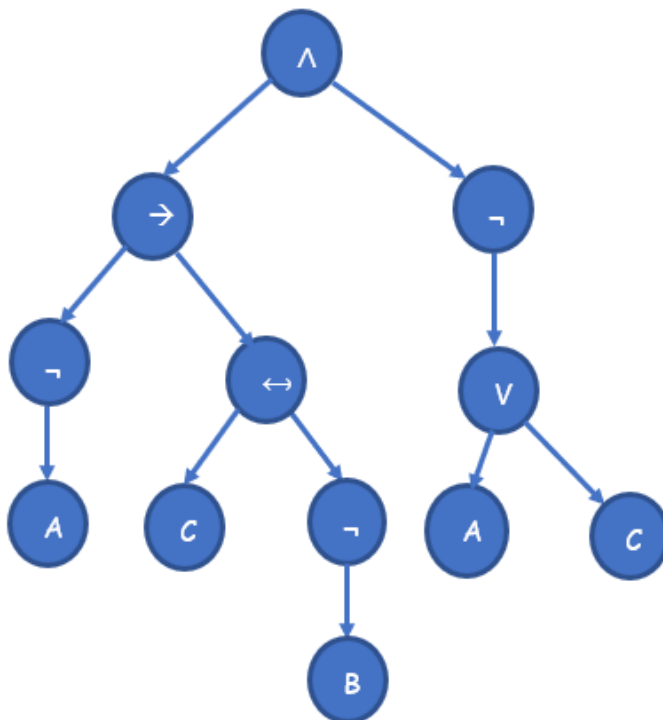
Si chiamano:

- 1 - **atomi** (o **variabili**): gli elementi di  $A$ ;
- 2 - **literal** ( $A$  e  $\neg A$ ) dove  $A$  è un elemento di  $A$ 
  - $A$  è un **literale positivo**;
  - $\neg A$  è un **literale negativo**
- 3 - **formule** gli elementi di  $F$ .

Per ogni **formula ben formata non atomica**  $F$  esiste **esattamente un connettivo principale**  $\odot$ .  $F$  è formato dall'applicazione di  $\odot$  a una o due formule ben formate.

Un **albero sintattico** rappresenta una **formula ben formata**.

$(\neg A \rightarrow (C \leftrightarrow \neg B)) \wedge \neg(A \vee C)$



Un' **assegnazione booleana** è una funzione totale  $V: A \rightarrow \{0,1\}$ .

Una **valutazione** stabilisce quali proposizioni atomiche sono vere e quali false.

$A$	$\neg A$
0	1
1	0

A	B	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \leftrightarrow B$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

Due formule sono **equivalenti** solo se non sono distinguibili tramite assegnazioni.

**F** e **G** sono **equivalenti** se entrambe definiscono la stessa tavola di verità.

Una **formula** viene detta:

- **tautologia**: se non ha contro modelli (**sempre vera** sotto ogni valutazione);
- **contraddizione**: se non ha modelli (**sempre falsa** sotto ogni valutazione);
- **soddisfacibile non tautologica**: se non né **tautologica** né una **contraddizione**.

#### 4) Regole di Inferenza: definizione

Sono un insieme di regole che permettono di passare da un numero finito di proposizioni assunte come premesse a una proposizione che funge da conclusione.

Forma

$$\frac{F_1, \dots, F_n}{F}$$

$F_1, \dots, F_n \rightarrow$  premesse della regola;

$F \rightarrow$  conclusione

**Esempio:**

$$\frac{F \wedge G}{F}$$

$$\frac{F \rightarrow G, F}{G}$$

$$\overline{F \vee \neg F}$$

#### 5) Regola di Inferenza: modus ponens

$$\frac{p \Rightarrow q, p}{\therefore q}$$

**Esempio:** Se piove, allora la strada è bagnata

a)  $p \rightarrow q$ : se piove, allora la strada è bagnata;

b)  $p$ : piove;

c)  $q$ : allora, la strada è bagnata.

La regola sintattica del **modus ponens** ci permette di aggiungere le conclusioni di un'implicazione all'insieme delle formule ben formate "vere".

### 6) Regola di Inferenza: modus tollens

$$\frac{p \Rightarrow q, \neg q}{\therefore \neg p}$$

**Esempio:** Se piove, allora la strada è bagnata

a)  $p \rightarrow q$ : se piove, allora la strada è bagnata;

b)  $\neg q$ : la strada non è bagnata;

c)  $\neg p$ : (allora) la strada è bagnata.

La regola sintattica del **modus tollens** ci permette di aggiungere la premessa negata di una regola all'insieme delle formule ben formate "vere".

### 7) Regola di Inferenza: eliminazione ed introduzione di "e" (congiunzione)

$$\begin{array}{c} \phi \\ \psi \\ \hline \phi \&\psi \end{array} \quad \begin{array}{c} \phi \&\psi \\ \phi \\ \hline \end{array} \quad \begin{array}{c} \phi \&\psi \\ \hline \psi \end{array}$$

**Esempio:**

a) Piove e la strada è bagnata

b) segue che piove

La regola sintattica **dell'eliminazione della congiunzione** ci permette di aggiungere all'insieme **FBF** i singoli componenti di una congiunzione.

### 8) Regola di Inferenza: introduzione di "o" (disgiunzione)

$$\frac{a_i}{a_1 \vee a_2 \vee \dots \vee a_i \vee \dots \vee a_n}$$

**Esempio:**

a) Piove

b) Piove o c'è vita su Marte.

La regola sintattica **dell'introduzione della disgiunzione** ci permette di aggiungere i singoli componenti di una formula complessa.

### Altre regole di inferenza

$$\frac{p \vee \neg p}{\text{vero}} \quad [\text{terzo escluso}]$$

$$\frac{p \wedge \text{vero}}{p} \quad [\text{eliminazione } \wedge]$$

$$\frac{p \wedge \neg p}{q} \text{ [contraddizione]}$$

$$\frac{\neg \neg p}{p} \text{ [eliminazione } \neg \text{]}$$

Tutte queste **regole di inferenza** fanno parte del calcolo naturale o **calcolo di Gentzen**.

### 9) Calcolo di Gentzen

Il **calcolo di Gentzen** formalizza i modi di derivare delle conclusioni a partire da un insieme di premesse (permette di derivare "direttamente" una formula ben formata mediante una sequenza di passi ben codificati).

### 10) Principio di Risoluzione: definizione e caratteristiche

Il **principio di risoluzione** è una **regola di inferenza** che ha le seguenti caratteristiche:

- opera su formule ben formate trasformate in forma normale congiunta;
- ognuno dei congiunti di queste formule viene detta **clausola**.

L'osservazione fondamentale alla base del principio di risoluzione è l'estensione della nozione di rimozione dell'implicazione sulla base del principio di contraddizione. Solitamente è usata per fare **dimostrazioni per assurdo**.

$$\frac{p \vee \neg r, \quad s \vee r}{p \vee s} \quad \text{Clausola risolvente}$$

$$\frac{\neg r, \quad r}{\perp} \quad \text{Clausola vuota}$$

La generazione della clausola vuota, corrisponde all'aver dimostrato che l'insieme di **formule ben formate** contiene una contraddizione.

$$\frac{\neg r, \quad r}{\perp} \quad \text{Clausola vuota}$$

### 11) Principio di Risoluzione: risoluzione unitaria

$$\frac{\neg p, \quad q_1 \vee q_2 \vee \dots \vee q_k \vee p}{q_1 \vee q_2 \vee \dots \vee q_k} \text{ [unit resolution]}$$

$$\frac{p, \quad q_1 \vee q_2 \vee \dots \vee q_k \vee \neg p}{q_1 \vee q_2 \vee \dots \vee q_k} \text{ [unit resolution]}$$

La **risoluzione unitaria** viene utilizzata quando una delle due clausole da risolvere è un **letterale** (preposizione o predicato) anche **negato** (come nel caso di p).

## 12) Dimostrazione per assurdo

Supponiamo di avere un insieme di **formule ben formate** (FBF) e supponiamo di voler dimostrare che una certa **preposizione p** (o **formula atomica**) è vera.

Utilizzando il metodo della **dimostrazione per assurdo**:

→ si assume che  **$\neg p$**  sia vera;

→ combinandola con le proposizioni in **FBF** ottengo una **contraddizione**, allora concludo che **p** deve essere vera.

**Esempio**: proviamo che p è vera

**FBF** = {p}

$\neg p \Rightarrow \{p\} \cup \{\neg p\} \rightarrow$  contraddizione

Quindi p deve essere vera.

## 13) Assiomi: conoscenze pregresse

Vi sono 5 **proposizioni** sempre vere (**tautologie**):

**A1**:  $A \rightarrow (B \rightarrow A)$

**A2**:  $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$

**A3**:  $(\neg B \rightarrow \neg A) \rightarrow ((\neg B \rightarrow A) \rightarrow B)$

**A4**:  $\neg(A \ \& \ \neg A)$  (principio di non contraddizione)

**A5**:  $A \vee \neg A$  (principio del terzo escluso)

## 14) Logica predicativa o logica di primo ordine

→ **Definizione**

La **logica dei predicati** è un ramo della logica matematica che si occupa di introdurre nomi per individui e per predicati e le variabili sono solo quelle individuali. Gli elementi della logica predicativa sono costanti, simboli relazionali, simboli funzionali e variabili.

→ **Semantica**

La **semantica** è basata da interpretazione che è un "mondo possibile" che esprime un potenziale per tutti i simboli.

→ **Interpretazione**

Una **interpretazione** ha un **dominio  $\Delta$**  (non vuoto) e una **funzione di interpretazione  $I$**  che legge ogni simbolo:

- una costante in un elemento di  $\Delta$ ;
- un simbolo **relazionale** n-ario in una relazione in  $\Delta^n$ ;
- un simbolo **funzionale** n-ario in una funzione  $\Delta^n \rightarrow \Delta$ ;
- variabili non sono interpretate: dipendono dal quantificatore.

### → Valori di verità

La **formula** è un predicato (una frase) che può essere vero o falso. La verità di un predicato dipende dall'interpretazione che viene considerata per la logica. Come nella logica proposizionale, esistono tautologie e contraddizioni. La logica non legge il linguaggio naturale (italiano).

### → Rappresentazione della conoscenza

La **logica predicativa** viene utilizzata per descrivere il mondo di interesse e le formule vengono utilizzate come assiomi che devono essere veri. Nel mondo della logica predicativa, le interpretazioni che le falsificano sono irrilevanti.

### → Dominio e Realtà

Le formule descrivono un dominio che può essere collegato o meno con la realtà.

### → Compromessi

Se si aggiungono conoscenze per rappresentare al meglio un dominio si rischia di complicare la base di conoscenza. Per questo motivo che è necessario trovare il giusto punto intermedio.

### → Obiettivo: trasformare le formule in linguaggio naturale.

Trasformare la conoscenza umana in un insieme di formule può presentare un problema: il linguaggio naturale è ambiguo, mentre la logica non lo è.

### → Simboli

I **simboli** hanno una funzionalità specifica:

- **costanti** parlano di entità;
- **relazioni** parlano di tuple con una proprietà;
- **funzioni** ci ritornano una nuova entità;
- **variabili** "unificano" con entità in base al bisogno.

È fondamentale comprendere la funzione di ogni elemento per costruire formule adatte.

### Esempio

- ha\_barba(grande\_puffo)
- $\exists x.(\text{Puffo}(x) \wedge \text{ha\_barba}(x))$
- $\exists x.(\text{cappello}(x) \wedge \text{indossa}(\text{grande\_puffo}, x) \wedge \text{colore}(x, \text{rosso}))$
- $\text{colore}(\text{cappello\_di}(\text{grande\_puffo}), \text{rosso})$
- $\forall x.(\text{Puffo}(x) \rightarrow \exists y.(\text{cappello}(y) \wedge \text{indossa}(x, y)))$
- $\forall x.(\text{colore}(\text{cappello\_di}(x), \text{rosso}) \rightarrow x = \text{grande\_puffo})$

Una possibile interpretazione sarebbe

- Grande Puffo ha la barba.
- Esiste un puffo che ha la barba.
- Grande Puffo indossa un cappello rosso.
- Il cappello di Grande Puffo è rosso.
- Tutti i puffi indossano un cappello.
- Tutti i cappelli di Grande Puffo sono rossi.

## PROLOG - Programmazione Logica

IDE: SWI - Prolog



15) Cos'è Prolog? Quali sono le sue caratteristiche?

**Prolog** (Programmazione Logica) è un linguaggio di programmazione che adotta il paradigma di programmazione logica, che utilizza la logica del primo ordine (FOL) sia per elaborare che rappresentare l'informazione. La programmazione logica differisce dalla programmazione tradizionale (imperativa) in quanto richiede e nello stesso tempo consente al programmatore di descrivere la struttura logica del problema piuttosto che risolverlo.

Le caratteristiche di **Prolog** sono:

- linguaggio di **general - purpose**;
- utilizzato in molti programmi di **Intelligenza Artificiale**, data la semplicità e la chiarezza della semantica e della sintassi;
- si basa sul calcolo dei predicati e la sintassi è limitata a **formule** dette **clausole di Horn** che sono disgiunzioni di letterali in cui al massimo un letterale è positivo;
- l'esecuzione del **programma Prolog** è **comparabile** alla dimostrazione di un teorema mediante la **risoluzione unitaria**;
- grande potere espressivo;
- computazione come costruzione di una dimostrazione di una affermazione.

16) Descrivere la forma normale a clausole

Ogni **formula ben formata (FBF)** di un linguaggio logico del **primo ordine** può essere riscritta in **forma normale a clausole**.

Esistono due **forme normali a clausole**:



a) **Forma normale congiunta (CNF)**: la **formula** è una congiunzione di disgiunzioni di predicati o di negazioni di predicati.

$$\bigwedge_i \left( \bigvee_j L_{ij} \right)$$

clausole

**Esempi**

$$\underbrace{(p(x) \vee q(x, y) \vee \neg t(z))}_{\text{clausola 1}} \wedge \underbrace{(p(w) \vee \neg s(u) \vee \neg r(v))}_{\text{clausola 2}}$$

$$\underbrace{(\neg t(z))}_{\text{clausola 1}} \wedge \underbrace{(p(w) \vee \neg s(u))}_{\text{clausola 2}} \wedge \underbrace{(p(x) \vee s(x) \vee q(y))}_{\text{clausola 3}}$$

Se si scarta il simbolo di **congiunzione**, si rimane solo con le clausole disgiuntive

$$p(x) \vee q(x, y) \vee \neg t(z)$$

$$p(w) \vee \neg s(u) \vee \neg r(v)$$

Le **clausole** relative al primo esempio sono riscrivibili come

$$t(z) \Rightarrow p(x) \vee q(x, y)$$

$$s(u) \wedge r(v) \Rightarrow p(w)$$

b) **Forma normale disgiunta (DNF)**: una **formula** è una disgiunzione di congiunzioni di predicati o negazioni di predicati (letterali positivi e letterali negativi)

$$\bigvee_j \left( \bigwedge_i L_{ij} \right)$$

dove

$$L_{ij} \equiv P_{ij}(x, y, \dots, z) \text{ o } L_{ij} \equiv \neg Q_{ij}(x, y, \dots, z)$$

## 17) Sintassi di Prolog

Un **programma Prolog** è costituito da un insieme di clausole della forma

a.

c :- b1, b2, ..., bn

:- q1, q2, q3, ..., qn

?- q1, q2, q3, ..., qn

**Termini**: a, bi, c e qi

**Goal**: congiunzione di termini

Ricorda che:

→ ogni fatto o regola o funzione DEVE terminare con un punto '.

→ ogni variabile DEVE iniziare con una maiuscola;

→ I commenti si inseriscono dopo un «%» (commento di linea) o tra «/\*» e «\*/» (come in **C/C++** e **Java**)

```

%%% Questa è una linea di commento
f(a,b). % Solo questa parte di riga è un commento.

%%% Ecco un commento
%%% su più righe.
%%% Anche questo è un commento.

/* Questo commento può essere
 * scritto su più righe
 */

```

### 18) Termini di Prolog: definizione e caratteristiche

I **termini** sono espressioni di Prolog che possono essere:

- **atomi**: sequenza di caratteri alfanumerici che inizia con un carattere minuscolo (può contenere anche l'underscore "\_");
- **variabili**: sequenza di caratteri alfanumerici che inizia con un carattere maiuscolo o con il carattere "\_" (variabili composte solo da "\_" vengono chiamate **indifferenza**);
- **termini composti**: costituiti dal **funtore** (simbolo di **funzione** o predicato definito come **atomo**) e da una sequenza di termini racchiusi tra parentesi tonde e separati da virgole (chiamati argomenti del **funtore**).

Validi	Non validi
foo	Hello Sam
hello	hello gian
hello_gian	1a
X	f(a
f(X)	f (a,b)
f(hello,gian)	X(a,b)
f(g(X))	1(a,b)

### 19) Fatti o Predicati di Prolog: definizione e caratteristiche

Un **fatto** o **predicato** consiste in:

- nome di predicato (inizia con la lettera minuscola);
- zero o più argomenti.

I **fatti** devono essere terminati da un punto ".".

#### Esempi

```

libro(pippol,prolog).
donna(yoshimoto).
gatto(tom).
topo(jerry).
risposta('42').

```

## 20) Quando vengono utilizzate le regole in Prolog?

Le **regole** vengono utilizzate quando si vuole esprimere che un determinato fatto dipende da un insieme di altri fatti.

**Esempio:**

1 - Se esci ora, cuoci.

2 - Uso la macchina, se il viaggio è lungo.

Le regole sono usate per esprimere definizioni

**X è fratello di Y**

→ se X è maschio;

→ X e Y hanno gli stessi genitori.

In **Prolog** una **regola** consiste in una **testa** e di un **corpo** e le caratteristiche sono:

→ **testa** e **corpo** sono collegati dall'operatore ":-";

→ la **testa** di una regola corrisponde al conseguente di un'implicazione;

→ il **corpo** di una regola corrisponde all'antecedente di un'implicazione logica.

Le **regole** di **Prolog** corrispondono alle **clausole di Horn**.

Gli operatori:

→ ":-" esprime il se;

→ ",", esprime e (congiunzione)

**Esempi**

"Il cane è un animale a quattro zampe che abbaia"

```
cane(X) :- animale(X), haQuattroZampe(X), abbaia(X).
```

"Gianluigi ama chiunque piace la moto"

```
ama(Gianluigi,X) :- piace(X,moto).
```

"Gianluigi ama le donne a cui piace il vino"

```
ama(Gianluigi,X) :- donna(X), ama(X,vino).
```

## 21) Come viene espressa una relazione in Prolog?

→ Una **relazione** (ad esempio **genitore**) può essere definita da più **regole** (ovvero da più **clausole**) aventi lo stesso predicato come conclusione.

**Esempio:**

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

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

→ Una **relazione** può anche essere definita **ricorsivamente**. In questo caso la definizione richiede almeno due proposizioni: una è quella ricorsiva che

corrisponde al caso generale, l'altra esprime il caso particolare più semplice.

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

## 22) Come vengono indentificati gli operatori logici in Prolog?

Gli **operatori logici** in **Prolog** vengono identificati da:

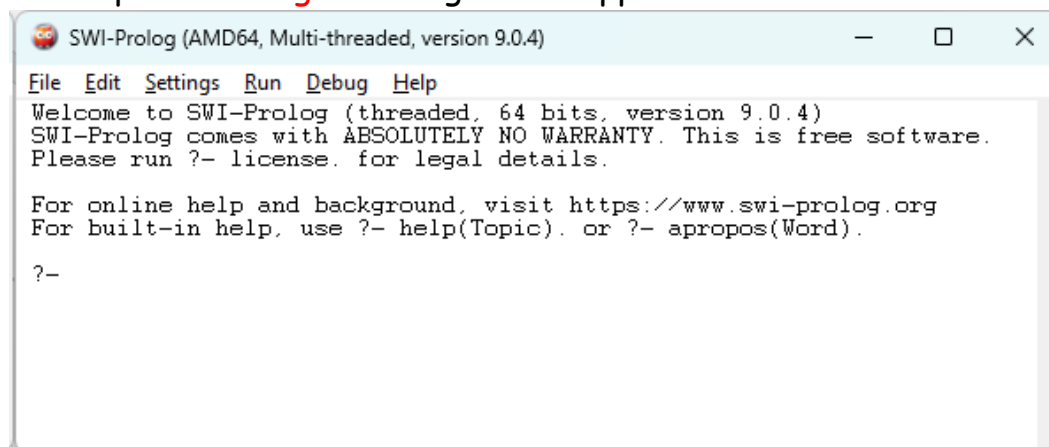
→ "," : che corrisponde all'operatore **AND logico**;

→ ";" : che corrisponde all'operatore **OR logico**.

## 23) Come viene eseguito un programma scritto in Prolog?

Eseguire un programma in **Prolog** significa **interrogare** il suo **interprete**.

L'interprete **Prolog** ha la seguente rappresentazione



## Esempio: Query all'interprete Prolog

Dato il seguente predicato:

```
%%% Mode: Prolog
%%% inizio programma: Esempio
male(gianluca).
%%% fine programma: Esempio
```

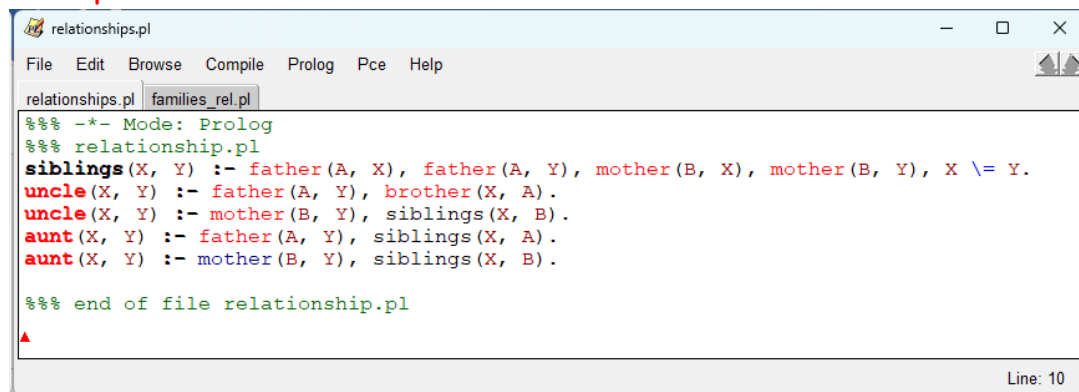
Una volta caricato il fatto e interrogato l'interprete, il sistema **Prolog** risponde "**true**", come mostrato di seguito:

```
?- male(gianluca).
true.
```

Interrogare il programma significa richiedere la dimostrazione di un teorema.

## 24) E' possibile introdurre variabili nelle query Prolog?

Sì, è possibile. Tali variabili vengono denominate **variabili esistenziali**, le quali vengono istanziate quando il sistema **Prolog** prova a rispondere alla domanda. Le variabili istanziate verranno poi mostrate nella risposta.

**Esempio:**


```

relationships.pl
File Edit Browse Compile Prolog Pce Help
relationships.pl families_rel.pl
%%% -*- Mode: Prolog
%%% relationship.pl
siblings(X, Y) :- father(A, X), father(A, Y), mother(B, X), mother(B, Y), X \= Y.
uncle(X, Y) :- father(A, Y), brother(X, A).
uncle(X, Y) :- mother(B, Y), siblings(X, B).
aunt(X, Y) :- father(A, Y), siblings(X, A).
aunt(X, Y) :- mother(B, Y), siblings(X, B).

%%% end of file relationship.pl

```

```

%%% male/1
:
male(terach).
male(abraham).
male(nachor).
male(haran).
male(isaac).
male(lot).
male(esau).
male(jacob).
male(bethuel).
male(laban).
male(benjamin).
male(joseph).
male(shantanu).
male(bhishma).
male(parashara).
male(chitrangada).
male(vichitravirya).
male(vyasa).
male(pandu).
male(dhritarashtra).
male(vidura).
male(yudhishtira).
male(bhima).
male(arjuna).
male(nakula).
male(sahadeva).
male(vyasa).
male(karna).
male(dharma).
male(vayu).
male(indra).
male(surya).
male(duryodhana).
male(dushasana).
male(vikarna).

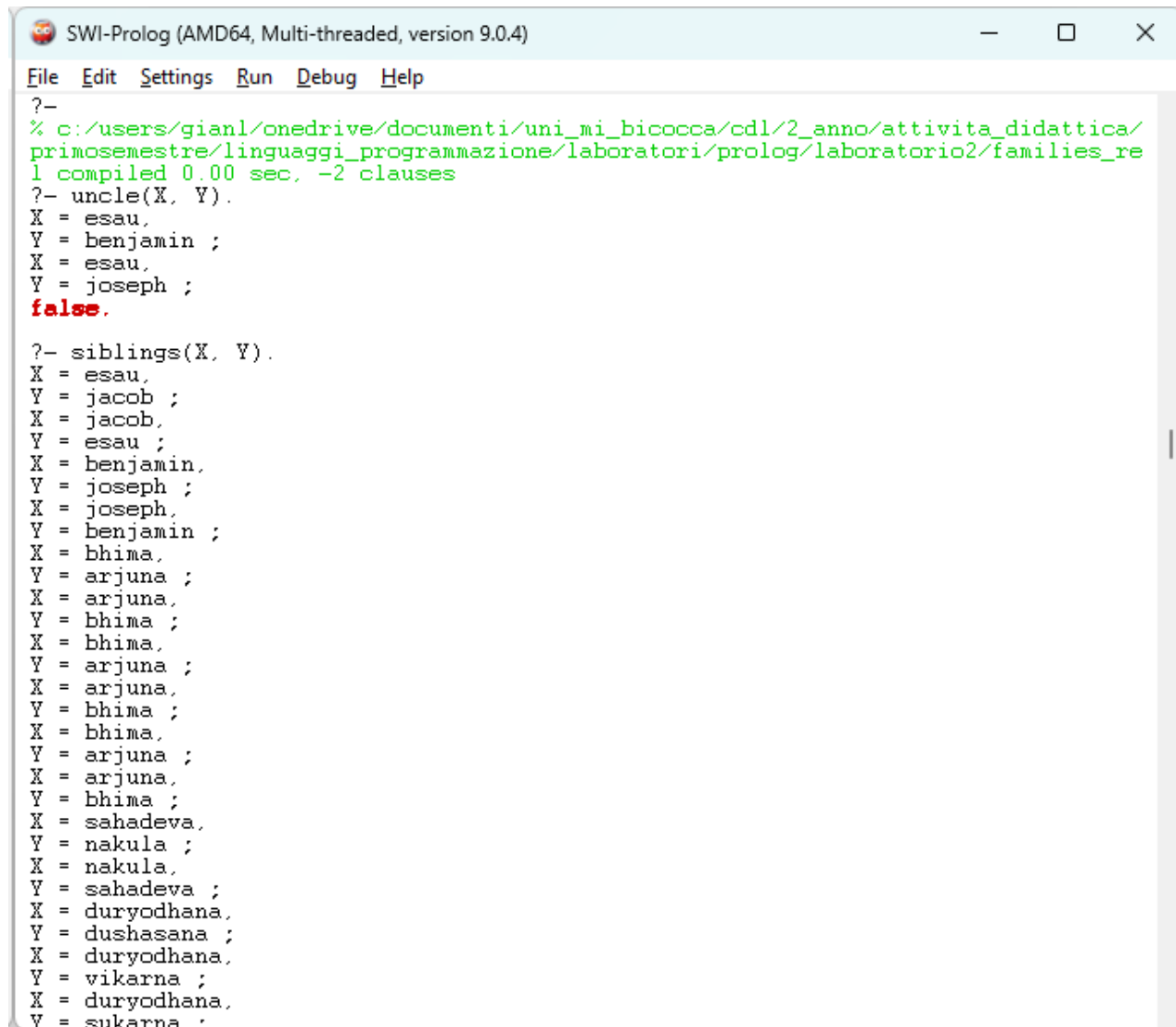
```

```
↑  
%%% female/1  
:  
female(sarah).  
female(milcah).  
female(yiscah).  
female(rebecca).  
female(rachel).  
female(ganga).  
female(satyavati).  
female(amba).  
female(ambika).  
female(ambalika).  
female('ambalika and ambika\'s maid').  
female(gandhari).  
female(kunti).  
female(madri).  
female(death).  
female(despair).  
female(desire).  
female(delirium).  
:  
:  
%%% father/2 --  
%%% father(X, Y) means 'X is father of Y'.  
:  
father(terach, abraham).  
father(terach, nachor).  
father(terach, haran).  
father(abraham, isaac).  
father(haran, lot).  
father(haran, milcah).  
father(haran, yiscah).  
father(nachor, bethuel).  
father(isaac, esau).  
father(isaac, jacob).  
father(jacob, benjamin).  
father(jacob, joseph).  
father(shantanu, bhishma).  
father(shantanu, chitrangada).  
father(shantanu, vichitravirya).  
father(parashara, vyasa).
```

```

%%% mother/2
%%% mother(X, Y) means 'X is mother of Y'.
.
mother(sarah, isaac).
mother(rebecca, esau).
mother(rebecca, jacob).
mother(rachel, benjamin).
mother(rachel, joseph).
mother(milcah, bethuel).
mother(satyavati, vyasa).
mother(ambika, dhritarashtra).
mother(ambalika, pandu).
mother('ambalika and ambika\'s maid', vidura).
mother(kunti, yudhishtira).
mother(kunti, bhima).
mother(kunti, arjuna).
mother(kunti, karna).
mother(madri, nakula).
mother(madri, sahadeva).
mother(gandhari, duryodhana).
mother(gandhari, dushasana).
mother(gandhari, vikarna).
mother(gandhari, sukarna).

```



SWI-Prolog (AMD64, Multi-threaded, version 9.0.4)

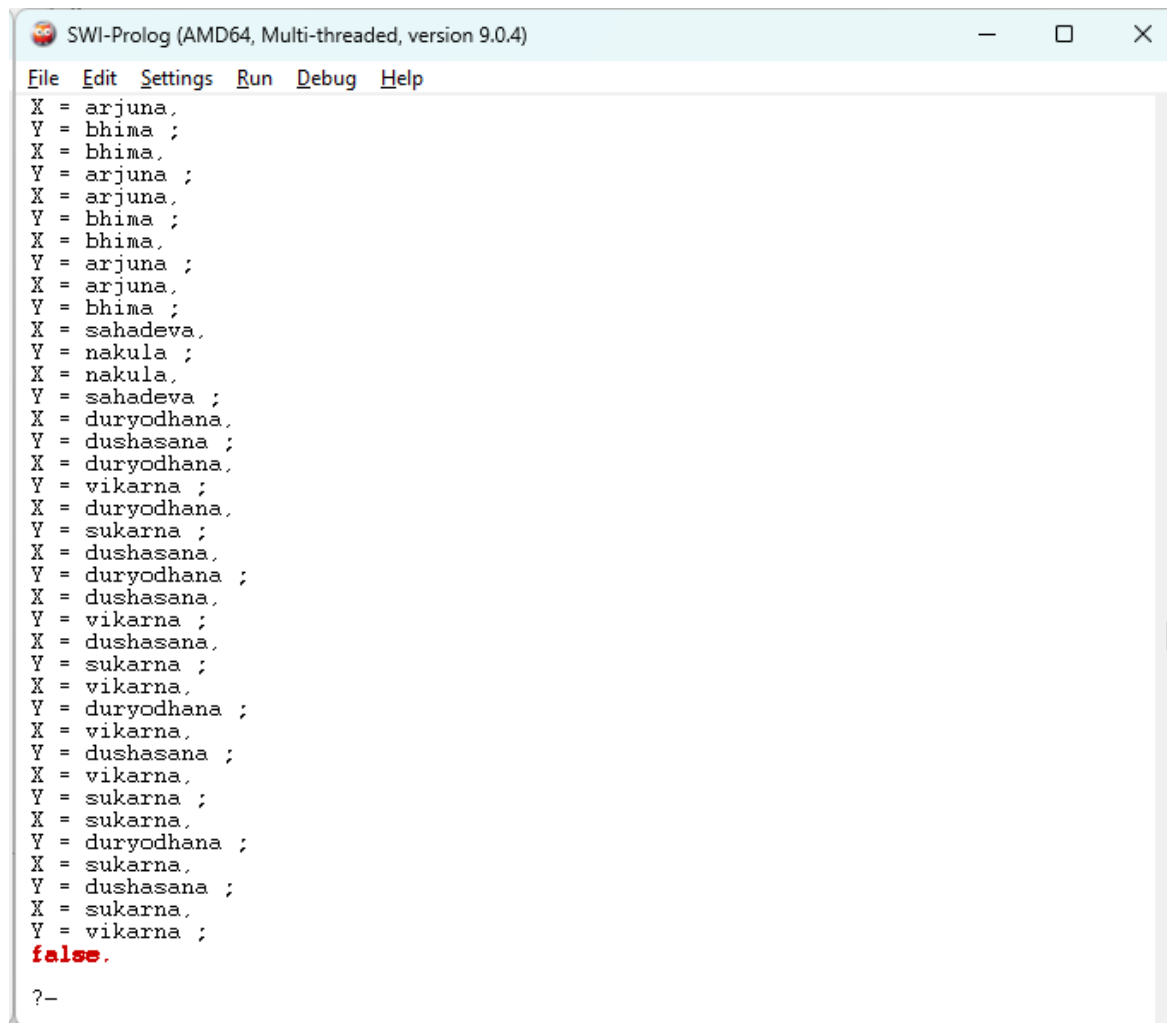
File Edit Settings Run Debug Help

```

?-
% c:/users/gianl/onedrive/documenti/uni_mi_bicocca/cdl/2_anno/attivita_didattica/
primosemestre/linguaggi_programmazione/laboratori/prolog/laboratorio2/families_re
l compiled 0.00 sec, -2 clauses
?- uncle(X, Y).
X = esau,
Y = benjamin ;
X = esau,
Y = joseph ;
false.

?- siblings(X, Y).
X = esau,
Y = jacob ;
X = jacob,
Y = esau ;
X = benjamin,
Y = joseph ;
X = joseph,
Y = benjamin ;
X = bhima,
Y = arjuna ;
X = arjuna,
Y = bhima ;
X = bhima,
Y = arjuna ;
X = arjuna,
Y = bhima ;
X = bhima,
Y = arjuna ;
X = arjuna,
Y = bhima ;
X = sahadeva,
Y = nakula ;
X = nakula,
Y = sahadeva ;
X = duryodhana,
Y = dushasana ;
X = duryodhana,
Y = vikarna ;
X = duryodhana,
Y = sukarna ;

```



```

SWI-Prolog (AMD64, Multi-threaded, version 9.0.4)
File Edit Settings Run Debug Help
X = arjuna,
Y = bhima ;
X = bhima,
Y = arjuna ;
X = arjuna,
Y = bhima ;
X = bhima,
Y = arjuna ;
X = arjuna,
Y = bhima ;
X = sahadeva,
Y = nakula ;
X = nakula,
Y = sahadeva ;
X = duryodhana,
Y = dushasana ;
X = duryodhana,
Y = vikarna ;
X = duryodhana,
Y = sukarna ;
X = dushasana,
Y = duryodhana ;
X = dushasana,
Y = vikarna ;
X = dushasana,
Y = sukarna ;
X = vikarna,
Y = duryodhana ;
X = vikarna,
Y = dushasana ;
X = vikarna,
Y = sukarna ;
X = sukarna,
Y = duryodhana ;
X = sukarna,
Y = dushasana ;
X = sukarna,
Y = vikarna ;
false.
?-

```

## 25) Operazione di unificazione e Most General Unifier

L'**unificazione** è l'operazione che permette di creare un insieme di **sostituzioni** delle variabili per rendere uguali due termini. Il **Most General Unifier (Mgu)** è la procedura di unificazione che costruisce un insieme di **sostituzioni**. La **sostituzione** è indicata come una sequenza ordinata di coppie variabile/valore.

**Esempi:**

```

Mgu(42, 42) -> {}
Mgu(42, X) -> {X/42}
Mgu(X, 42) -> {X/42}
Mgu(foo(bar, 42), foo(bar, X)) -> {X/42}
Mgu(foo(Y, 42), foo(bar, X)) -> {Y/bar, X/42}
Mgu(foo(bar(42), baz), foo(X, Y)) -> {X/bar(42), Y/baz}
Mgu(foo(X), foo(bar(Y))) -> {X/bar(Y), Y/_G001}

```

La **Most General Unifier (Mgu)** rappresenta il risultato finale della procedura di valutazione. Per rendere esplicita l'operazione di unificazione, è possibile utilizzare il comando **Prolog "="**. Di seguito vengono riportati degli esempi:



```

?- 42 = 42.
true.

?- 42 = X.
X = 42.

?- foo(bar, 42) = foo(bar, X).
X = 42.

?- foo(Y, 42) = foo(bar, X).
Y = bar,
X = 42.

?- foo(bar(42), baz) = foo(X, Y).
X = bar(42),
Y = baz.

?- foo(X) = foo(bar(Y)).
X = bar(Y).

?- foo(42, bar(X), trillion) = foo(Y, bar(Y), X).
false.

```

## 26) Diverse rappresentazioni di dati e interrogazioni

Considerando il seguente esempio: vogliamo descrivere un insieme di fatti riguardanti i corsi offerti dal dipartimento.

**1 possibilità:** tutte le informazioni sono concentrate in una relazione con 6 campi

```

corso(linguaggi, lunedì, '9:30', 'U4', 3, antoniotti).
corso(biologia_computazionale, lunedì, '14:30', 'U14', t023, antoniotti).

```

A partire da questa definizione è possibile poi costruire altri predicati

```

aula(Corso, Edificio, Aula) :-
    corso(Corso, _, _, Edificio, Aula, _).
docente(Corso, Docente) :-
    corso(Corso, _, _, _, _, Docente).

```

**2 possibilità:** Tutte le informazioni sono concentrate in una relazione con 4 campi; le informazioni sono concentrate in termini funzionali che rappresentano informazioni raggruppate logicamente.

```

corso(linguaggi, orario(lunedì, '9:30'), aula('U4', 3), antoniotti).
corso(biologia_computazionale,
      orario(lunedì, '14:30'),
      aula('U4', 3),
      antoniotti).

```

A partire da questa definizione possiamo poi costruire altri predicati

```

aula(Corso, Edificio, Aula) :- corso(Corso, _, aula(Edificio, Aula), _).
docente(Corso, Docente) :- corso(Corso, _, _, Docente).

```

%%% oppure...

```

aula(Corso, Luogo) :- corso(Corso, _, Luogo, _).

```

**3 possibilità:** I predicati definiti dalle relazioni con 6 o 4 campi possono essere ricodificate con predicati **binari** (cfr., le triple **XML** usate negli strumenti e nella ricerca sulla «**semantic web**»).

```
giorno(linguaggi, martedì).
orario(linguaggi, '9:30').
edificio(linguaggi, 'U4').
aula(linguaggi, 3).
docente(linguaggi, antoniotti).
```

La costruzione di schemi **RDF/XML** (e, a volte, **SQL**) corrisponde a questa operazione di ri-rappresentazione con triple.

## 27) Liste in Prolog

→ La **lista** in **Prolog** viene definita racchiudendo gli elementi tra parentesi quadre '[' e ']' e separandoli da virgole.

→ Gli elementi di una **lista** in **Prolog** possono essere termini qualsiasi o liste.

→ [] indica la **lista** vuota.

→ Ogni **lista** non vuota può essere divisa in due parti:

a) la **testa** è il primo elemento della lista;

b) la **coda** rappresenta il resto ed è sempre una **lista**.

Esempio:

[1, 2, 3] → 1 è la **testa** e [2, 3] è la **coda**.

[1, 2] → 1 è la **testa** e [2] è la **coda**

[1] → 1 è la **testa** e [] è la **coda**

[[1]] → [1] è la **testa** e [] è la **coda**

[[1, 2], 3] → [1, 2] è la **testa** e [3] è la **coda**

[[1, 2], [3], 4] → [1, 2] è la **testa** e [[3], 4] è la **coda**

## 28) Operatore |

**Prolog** possiede uno speciale operatore usato per distinguere la testa dalla coda di una lista: l'operatore **|**.

Esempio:

1)

```
?- [X | Ys] = [1, 2, 3, 4].
```

```
X = 1,
```

```
Ys = [2, 3, 4].
```

2)

```
?- [X, Y | Zs] = [the, answer, is, 42].
```

```
X = the,
```

```
Y = answer,
```

```
Zs = [is, 42].
```

3)

```
?- [X, 42 | _] = [41, 42, 43, foo(bar)].
X = 41.
```

### 29) La lista vuota []

La **lista vuota []** in **Prolog** viene gestita come una **lista speciale**.

```
?- [X | Ys] = [].
false.
```

### 30) L'interprete Prolog: consult

Il comando **consult** inizializza o carica un insieme di fatti e regole nell'ambiente **Prolog**. Appare come un predicato da valutare (un goal) e prende almeno un termine che denota un file come argomento.

```
?- consult('guida-astrostoppista.pl').
Yes
```

```
?- consult('Projects/Lang/Prolog/Code/esempi-liste.pl').
Yes
```

Può essere usato anche per inserire fatti e regole direttamente alla console usando il termine speciale **user**.

```
2 ?- consult(user).
|: f(42).
|: friends(zaphod, trillian).
|: ^D
```

### 31) L'interprete Prolog: reconsult

Il predicato **reconsult** deve invece essere usato quando si vuole ricaricare un file (ovvero un data o knowledge base) nell'ambiente **Prolog**.

**Esempio:**

```
?- reconsult('guida-astrostoppista.pl').
Yes
```

### 32) Clausole di Horn

Un programma **Prolog** è un insieme di **clausole di Horn** che rappresentano:

→ **fatti** riguardanti gli oggetti in esame e le relazioni che intercorrono tra di loro;

→ **regole** sugli oggetti e sulle relazioni;

→ **interrogazioni (goal o query)** sulla base della conoscenza definita.

Le **clausole di Horn** che sono **disgiunzioni di letterali in cui al massimo un letterale è positivo**, come ad esempio

$$A \leftarrow B_1 \wedge B_2 \wedge \dots \wedge B_m$$

Tali **clausole** vengono classificate in:

- **Fatti**:  $A \leftarrow$
  - **Regole**:  $A \leftarrow B_1 \& B_2 \& \dots \& B_m$
  - **Goals**:  $\leftarrow B_1 \& B_2 \& \dots \& B_m$
  - **Contraddizione**:  $\leftarrow$
- In **Prolog** diventa:
- **Fatti**:  $A.$
  - **Regole**:  $A :- B_1, B_2, \dots, B_m$
  - **Goals**:  $:- B_1, B_2, \dots, B_m$
  - **Contraddizione**: `fail`

### 33) Sostituzione: definizione formale

La **sostituzione** è una funzione applicabile a  $T$ , insieme dei termini, che specifica quali valori possono essere sostituiti alle variabili.

$$\sigma: T \rightarrow T$$

$$\sigma = \{X_1/v_1, X_2/v_2, \dots, X_k/v_k\}$$

**Esempio:**

$$\sigma(\text{bar}(X, Y)) = \text{bar}(42, \text{foo}(s(0)))$$

### 34) Esecuzione Programma Prolog

Considerando almeno un **Goal**  $GO$  da provare, si deve dimostrare che da  $P \cup \{GO\}$  è possibile derivare la **clausola vuota**, ovvero effettuare una dimostrazione per assurdo mediante applicazione del **Principio di Risoluzione**.

**Esempio:**

Dato un programma  $P$  e la query

$$:- p(t_1, t_2, \dots, t_m).$$

se  $x_1, x_2, \dots, x_n$  sono le variabili che compaiono in  $t_1, t_2, \dots, t_m$ , il significato della query è

$$\exists x_1, x_2, \dots, x_n. p(t_1, t_2, \dots, t_m)$$

e l'obiettivo è quello di trovare una sostituzione

$$s = \{x_1/s_1, x_2/s_2, \dots, x_n/s_n\}$$

dove gli  $s_i$  sono termini tali per cui  $P \vdash s[p(t_1, t_2, \dots, t_m)]$

### 35) Risoluzione ad Input Lineare: SLD

La **risoluzione ad Input Lineare** dimostra la veridicità o meno di un'interrogazione eseguendo una sequenza di passi di risoluzione, che avviene sempre fra l'ultimo **goal** derivato in ciascun passo e una **clausola** di programma. Il **risultato finale** può essere:

- **successo**: viene generata la **clausola vuota**;
- **insuccesso finito**: se per  $n$  finito,  $G_n$  non è uguale a ":-";
- **insuccesso infinito**: se è sempre possibile derivare nuovi risolvanti.

### 36) Strategia di selezione di un sotto - goal

Possono esserci più clausole di programma utilizzabili per applicare la risoluzione con il goal corrente. Si possono adottare due diverse strategie di ricerca per queste clausole:

- **in profondità (Depth First)**: si sceglie una clausola e si mantiene fissa questa scelta, finché si arriva alla clausola vuota o all'impossibilità di fare nuove risoluzioni (si riconsiderano le scelte fatte precedentemente);
- **in ampiezza (Breadth First)**: si considerano in parallelo tutte le alternative.

**Prolog** adotta la strategia di **risoluzione in profondità** per risparmiare memoria pur non essendo completa per le **clausole di Horn**.

### 37) Albero di Derivazione SLD

Dato un **programma logico P**, un **goal GO** e una regola di calcolo **R**, un **albero SLD** per  **$P \cup \{GO\}$**  via **R** è definito sulla base del processo di prova:

- la **radice** è il **goal**;
- ciascun **nodo** è un **goal**.

La **regola R** è variabile :

- **Left - most**: la scelta avviene nell'atomo più a sinistra nel goal e con tale regola, si ottiene un **albero SLD** che ha un ramo di **successo** con  $\{x/f(a,b)\}$ , e un ramo **insuccesso infinito**;
- **Right - most**: la scelta avviene nell'atomo più a destra nel goal e con tale regola si ottiene un albero SLD che ha un solo ramo di **successo** con  $\{x/f(a,b)\}$ ;
- scelta "**sotto - goal casuale**";
- scelta del "**sotto - goal migliore**".

Ad ogni ramo dell'albero corrisponde una **derivazione SLD** di **successo**. Il numero di cammini di successo è la stessa qualsiasi regola di calcolo si scelga, bensì influenza il numero di cammini di fallimento.

Il **Prolog** adotta una regola **left - most**. L'**albero SLD** generato dal sistema **Prolog** ordina i figli di un nodo secondo l'ordine dall'alto verso il basso delle regole e dei fatti del **programma P**.

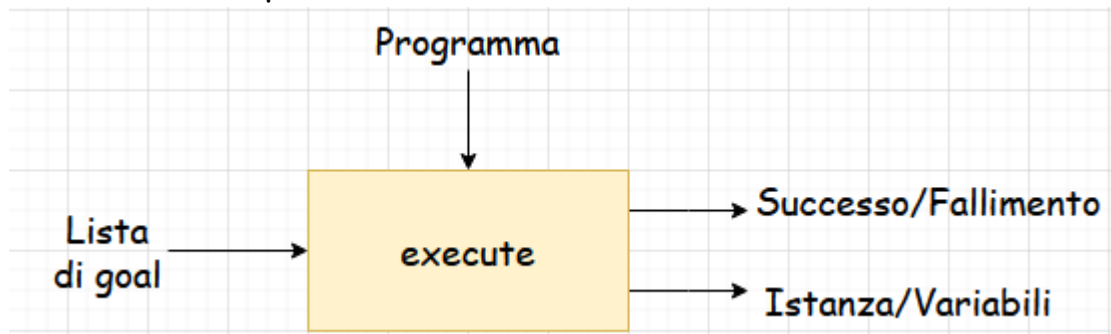
### 38) Descrivere il modello di esecuzione di un programma Prolog

Dato il seguente programma **Prolog**:

```
?- p :- q, r.
```

Come interpretazione dichiarativa il fatto che p è vero se e solo se lo sono q ed r. Come interpretazione procedurale, il problema p può essere scomposto nei problemi q ed r.

Modello di interpretazione



Il **goal** può essere visto come una **chiamata di procedura**.

Una **regola** può essere vista come la definizione di una **procedura** in cui la **testa** è l'intestazione mentre la parte destra è il **corpo**.

L'esecuzione del **programma Prolog** si basa su due **stack**:

→ **stack di esecuzione**: che contiene i **record di attivazione** delle procedure, ovvero le sostituzioni per l'unificazione delle varie regole;

→ **backtracking stack**: che contiene l'insieme dei "punti di scelta"; ad ogni fase della valutazione, contiene dei puntatori alle scelte aperte nelle fasi precedenti della dimostrazione.

### Modello di Esecuzione

Consideriamo il seguente programma **Prolog**

```
(c11) a :- p, b.
(c12) a :- p, c.
(c13) p.
goal := ?- a.
```

1 - Seguendo il programma, si inizia dalla query **?- a.**

a
---

Stack di Esecuzione

c11
c12

Stack di Backtracking

c11: scelta

2 - Si mette p in cima allo stack

p
a

c13
c11
c12

Stack di Esecuzione

Stack di Backtracking

(c11, c13): scelta corrente

3 - p ha successo, allora si mette b in cima allo stack.

b
a

c13
c11
c12

Stack di Esecuzione

Stack di Backtracking

(c11, c13): scelta corrente

4 - b fallisce e viene attivato il meccanismo di **backtracking**.

b
a

c11
c12

Stack di Esecuzione

Stack di Backtracking

c11: scelta corrente

5 - si mette p in cima allo stack e si va a considerare la seconda clausola.

p
a

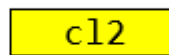
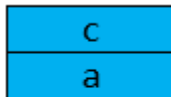
c13
c12

Stack di Esecuzione

Stack di Backtracking

(c12, c13): scelta corrente

6 - la valutazione ha avuto successo, si inserisce c:

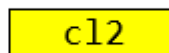
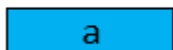


Stack di Esecuzione

Stack di Backtracking

c12: scelta corrente

7 - La valutazione di c fallisce; quindi viene attivato il meccanismo di **backtracking**; ma visto che non ci sono più clausole anche a fallisce e quindi lo stack di esecuzione si svuota.



Stack di Esecuzione

Stack di Backtracking

### 39) Cosa si intende per cut?

Il **cut** è un predicato di controllo che permette di interrompere il meccanismo di backtracking, eliminando tutti i punti di scelta creati quando si è entrati nel predicato nel quale appare il **cut**. Modifica soltanto l'attraversamento nell'albero di derivazione ma non la sua struttura.

```
a :- p, !, b.
```

### 40) Quali sono le due tipologie di cut?

Il **cut** può essere di due tipologie:

- **green cuts**: utilizzati per esprimere determinismo e rendere il programma più efficiente;
- **red cuts**: utilizzati per scopi di efficienza: omettono alcune condizioni esplicite e modificano la semantica del programma (tendenzialmente indesiderabili).

### 41) Predicati metalogici: descrizione e caratteristiche

I **predicati metalogici** servono per stabilire l'input e l'output in predicati non invertibili (e.g. convertitore Celsius / Fahrenheit e viceversa), che non hanno quindi la tipica invertibilità di varie queries. Ciò si deve all'utilizzo di vari predicati aritmetici (>, <, =, is, ...) che sacrificano la semantica.

Essi trattano variabili come oggetti:

- **var(X)**: restituisce **true** se l'argomento è una variabile;
- **nonvar(X)**: restituisce **true** se l'argomento non è una variabile;



**Esempi:**

```
?- var(X).
true.

?- nonvar(ciao).
true.

?-
```

**42) Come possono essere i termini?**

I **termini** possono essere di due tipologie:

→ **atomici**: corrispondenti alle costanti in un linguaggio logico del primo ordine (**Prolog**: `atomic(X)`);

→ **composti**: corrispondenti alle funzioni e ai predicati (**Prolog**: `compound(X)`);

**43) Quali sono i predicati che permettono di ispezionare un termine?**

In **Prolog**, un termine **Term** può essere ispezionato mediante tre predicati:

→ `functor(Term, F, Arity)`: vero se **Term** è un termine, con **Arity** argomenti, il cui funtore (simbolo di funzione o di predicato) è **F**;

→ `arg(N, Term, Arg)`: vero se l'**N**-esimo argomento di **Term** è **Arg**;

→ `Term =.. L`: questo predicato, `=..`, viene chiamato (per motivi storici) **univ**; è vero quando **L** è una **lista** il cui primo elemento è il funtore di **Term** ed i rimanenti elementi sono i suoi argomenti.

**Esempi:**

```
?- functor(foo(24), foo, 1).
true.

?- arg(3, node(x, _, [], []), X).
X = [].

?- father(haran, lot) =.. Ts.
Ts = [father, haran, lot].

?-
```

**44) Programmazione di ordine superiore: descrizione e caratteristiche**

A volte, vi è la necessità di effettuare delle richieste che non direttamente formidabili al primo ordine. **Prolog** mette a disposizione dell'utente una serie di predicati su insiemi che estendono il modello computazionale:

→ `findall(Template, Goal, Set)`: vero se `set` contiene tutte le istanze di `template` che soddisfano `goal`;

→ `bagof(Template, Goal, Set)`: vero se `bag` contiene tutte le alternative di `template` che soddisfano `goal` e con  $\text{Var}^G$  si definiscono le variabili che non vanno considerate libere;

→ `setof(Template, Goal, Set)`: si comporta come `bagof`, ma `Set` non contiene soluzioni duplicate;

→ call(G) :- G. : variabili interpretabili come goal.

### Esempi:

```
?- findall(C, father(X, C), Kids).
Kids = [abraham, nachor, haran, isaac, lot, milcah, yiscah, bethuel, esau|...].

?- bagof(C, father(X, C), Kids).
X = abraham,
Kids = [isaac] ;
X = ashwini,
Kids = [sahadeva, nakula] ;
X = dharmā,
Kids = [yudhistira, bhima, arjuna] ;
X = dhritarashtra,
Kids = [duryodhana, dushasana, vikarna, sukarna] ;
X = dream,
Kids = [orpheus] ;
X = haran,
Kids = [lot, milcah, yiscah] ;
X = indra,
Kids = [yudhistira, bhima, arjuna] ;
X = isaac,
Kids = [esau, jacob] ;
X = jacob,
Kids = [benjamin, joseph] ;
X = nachor,
Kids = [bethuel] ;
X = parashara,
Kids = [vyasa] ;
X = shantanu,
Kids = [bhishma, chitrangada, vichitravirya] ;
X = surya,
Kids = [karna] ;
X = terach,
Kids = [abraham, nachor, haran] ;
X = vayu,
Kids = [yudhistira, bhima, arjuna] ;
X = vyasa,
Kids = [dhritarashtra, pandu, vidura].

?- setof(C, father(X, C), Kids).
X = abraham,
Kids = [isaac] ;
X = ashwini,
Kids = [nakula, sahadeva] ;
X = dharmā,
Kids = [arjuna, bhima, yudhistira] ;
X = dhritarashtra,
Kids = [duryodhana, dushasana, sukarna, vikarna] ;
X = dream,
Kids = [orpheus] ;
X = haran,
Kids = [lot, milcah, yiscah] ;
X = indra,
Kids = [arjuna, bhima, yudhistira] ;
X = isaac,
Kids = [esau, jacob] ;
X = jacob,
Kids = [benjamin, joseph] ;
X = nachor,
Kids = [bethuel] ;
X = parashara,
Kids = [vyasa] ;
X = shantanu,
Kids = [bhishma, chitrangada, vichitravirya] ;
X = surya,
Kids = [karna] ;
X = terach,
Kids = [abraham, haran, nachor] ;
X = vayu,
Kids = [arjuna, bhima, yudhistira] ;
X = vyasa,
Kids = [dhritarashtra, pandu, vidura].
```

**45) Manipolazione Basi di Dati: descrizione e caratteristiche**

Esistono dei predicati che manipolano direttamente la base di dati:

→ `listing`: stampa la base di dati;

→ `assert(X)`: asserisce X

a) `assertz(X)`: accoda X

b) `asserta(X)`: appende

→ `retract(X)` : inverso di `assert(X)` (rimuove solo una asserzione, usando una variabile si possono eliminare più asserzioni).

**46) Gestione I/O in Prolog: descrizione e caratteristiche**

I predicati primitivi principali per la gestione dell'I/O in **Prolog** sono essenzialmente due, `read` e `write`, a cui si aggiungono i vari predicati per la gestione dei files e degli streams: `open`, `close`, `seek`, etc.

I predicati di **gestione I/O** in **Prolog** sono:

→ `read(X)`: invoca il parser **Prolog**;

→ `write(X)`: è equivalente all'invocazione di un metodo `toString` **Java** su un oggetto di classe "termine" (il predicato `write_term` dà più controllo su come il termine può essere «scritto»).

**Esempi:**

```
?- write(42).
42
true.

?- foo(bar) = X, write(X).
foo(bar)
X = foo(bar).

?- read(What).
|: foo(42, Bar).

What = foo(42, _).
```

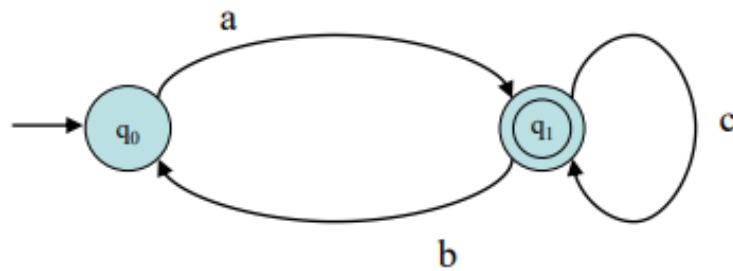
**47) Interpreti in Prolog: automi**

**Prolog** è un linguaggio che si presta per la costruzione di interpreti per la manipolazione di linguaggi specializzati (esempio interpreti per automi).

**Costruzione di automi**

```
accept([I | Is], S) :- delta(S, I, N), accept(Is, N).
accept([], Q) :- final(Q).
```

Considerando il seguente **automa**:



La costruzione di tale **automa** è il seguente:

```

accept([I | Is], S) :- delta(S, I, N), accept(Is, N).
accept([], Q) :- final(Q).
  
```

```

initial(q0).
final(q1).
  
```

```

delta(q0, a, q1).
delta(q1, b, q0).
delta(q1, c, q1).
  
```

```

recognize(Input) :- initial(S), accept(Input, S).
  
```

**Query:**

```

?- recognize([a, b, a, c, c, b, a]).
true.

?- recognize([a]).
true.

?- recognize([a, b]).
false.
  
```

**Costruzione di automi a pila**

```

accept([I | Is], Q, S) :- delta(Q, I, S, Q1, S1), accept(Is, Q1, S1).
accept([], Q, []) :- final(Q).
  
```

```

initial(q0).
final(q1).
  
```

```

delta(q0, a, P, q0, [a | P]).
delta(q0, b, P, q0, [b | P]).
delta(q0, c, P, q0, [c | P]).
delta(q0, r, P, q1, P).
delta(q1, c, [c | P], q1, P).
delta(q1, b, [b | P], q1, P).
delta(q1, a, [a | P], q1, P).
  
```

**Query:**

```

% c:/users/gianl/onedrive/documenti/prolog/stackautoma compiled 0.00 sec, 0 clauses
?- recognize([a, b, a, c, r, c, a, b, a]).
true ;
false.

?- recognize([a, b, a, c, r, c, a, b, a]).
true ;
false.

?- recognize([a, b, a, c, r, b]).
false.
  
```