Prolog

Prolog
- Introduzione
- Basic commands
- Unification
- Regole
- Definizioni
- Ricorsione delle regole
- Data objects
  - Atoms
  - Numeri
  - Variabili
- Operators
- Input and Output
- Built-in predicates
- Predicati ricorsivi
  - Cicli
  - Liste
    - Ordinamento
    - Lunghezza
    - Somma degli elementi
    - Concatenazione di due liste

Introduzione

- book: Prolog Programming for Artificial Intelligence
- SWI-Prolog
  - **listener**
    - richieste (query-goal) - interrogazioni al Knowledge Base
    - risposte (results) - di una richiesta
  - **database**
    - Knowledge Base
      - facts(arguments)
        - verifica se il fatto è vero o meno
        - risultato: true / false
        - i fatti con stesso predicato devono essere raggruppati insieme
      - rules(arguments)
        - verifica se la regola è vera o falsa
    - file con estensione `.pl`

Basic commands

- commenti `% commento`
- get current working directory: `working_directory(D,D).`
- set new working directory: `working_directory(D,"path").`
- print files in CWD: `ls.`

- switch db: `consult("DB.pl").`, `[DB].`
- clear the console: `tty_clear.`
- print to console: `write('something').`, `write(X),nl.` (nl = new line)

Unification

- le query in prolog includono il **pattern matching**
- query = richiesta = goal
- prolog pattern matching si chiama unification
- quando il DB contiene solo fatti, l'unificazione ha successo se:
  - il predicato usato come goal (query) e quello nel db corrispodono
  - i predicati hanno lo stesso numero di argomenti
  - tutti gli argomenti sono uguali

```
# DB family.pl

?- parent(tom,bob)       # Tom è padre di bob
parent/2                 #  Altra forma

# Query
?- parent(bob,pat). # True -> è un fatto verificato

# Trovare tutti i genitori di bob
# Argomento che inizia con lettera maiuscola -> variabile
?- parent(X,bob).        # X = pam; X = tom.

# Mostrare tutti i figli di bob
?- parent(bob,X).        # X = ann; X = pat.

# Mostrare tutte le combinazioni genitore-figlio
?- parent(X,Y);      # X -> genitore, Y -> figlio

# Chi è il nonno di Jim?
# congiunzione (AND) -> ','
?- parent(X,jim),parent(Y,X).    # parent(Y,jim) AND parent(X,Y)

# Chi sono i nipoti di Tom?
?- parent(tom,Y),parent(Y,X).    # Da verificare

# Verificare se Ann e Pat hanno lo stesso genitore
?- parent(X,ann),parent(X,pat).
```

Regole

- possiamo definire delle relazioni tramite regole

```
# Fatti su cui basare la regola
female(pam).
male(tom).
male(bob).

# Regola per definire la madre
# For all X and Y,
#   X is the mother of Y if
```

```
#       X is a parent of Y and X is female

mother(X,Y) :- parent(X,Y),female(X).

# For all X and Y,
#   X is a parent of Y then
#       X has a child
hasChild(X) :- parent(X,Y).
```

Definizioni

- clause (clausole)
  - **fatti** - dichiarano cose che sono sempre vere
    - sono clause che hanno soltanto l'head o il body
  - **rules** - dichiarano cose che sono vere in base ad una condizione
    - formato da head e (non-empty) body
  - **questions** - per mezzo di esse l'utente può chiedere quali cose sono vere

```
# Everybody who has a child is happy
happy(X) :- hasChild(X).    # if X hasChild then X is happy

# For all X, if X has a child who has a sister then X has two children
hasTwoChildren(X) :- parent(X,Y),sister(Y,_).

# Define the grandchild relation using the parent one
grandchild(X,Y) :- parent(Y,Z),parent(Z,X).

# Define the aunt relation using the parent and sister ones
aunt(X,Y) :- parent(Z,Y),sister(X,Z).
```

Ricorsione delle regole

- richiamo ricorsivo di una relazione logica fino a giungere ad un punto di arrivo (altrimenti ricadiamo in un caso di ciclo infinito)

```
# Es: mostrare gli antenati di X

# Definizione di antenato
# For all X and Z,
#   X is an ancestor of Z if
#       there is a Y such that
#           X is a parent of Y AND
#           Y is an ancestor of Z

# Antenati a 2 livelli di gerarchia (nonno)
ancestor(X,Z) :-
    parent(X,Y1),
    parent(Y1,Z).

# Antenati a 3 livelli di gerarchia (bisnonno)
ancestor(X,Z) :-
    parent(X,Y1),
    parent(Y1,Y2),
    parent(Y2,Z).
```

Data objects

- data objects
  - simple objects
    - constants
      - atoms (blocco che si sta usando)
      - numbers (integers, double)
    - variables
  - structures

Atoms

- si possono costruire in tre modi:
  - stringhe con lettere, numeri, underscore (non all'inizio)
  - stringhe con caratteri speciali
    - `:-` è predefinita
  - stringhe che iniziano per maiuscolo tra virgolette

Numeri

- Numeri reali e interi
- Notazione esponenziale: `7.15E-9`

Variabili

- iniziano sempre con un carattere maiuscolo o un `_`
- Ex: `Anna`, `_anna`

```
# Stampa il nome del figlio in caso esista
has_a_child(X) :- parent(X,Y).

# Restituisce True se il figlio esiste, senza specificarne il nome
has_a_child(X) :- parent(X,_).
```

- Ex:

```
# Creare dei predicati con un argomento che identifica il genere delle persone
di una famiglia (DB family.pl)
male(bob).
male(tom).
male(jim).
female(ann).
female(pam).
female(liz).
female(pat)

# Usare delle query che:
# 1. Confermano che una delle persone della famiglia sia maschio o femmina
?- male(bob).
# 2. Mostrano tutti i maschi della famiglia
?- male(X).
# 3. Mostrano tutte le femmine della famiglia
```

```
?- female(Y).
```

Operators

- functions to work arithmetic, logic, comparison and other operations
- **Arithmetic operators**
  - Addition (+)
  - Subtraction (-)
  - Multiplication (*)
  - Division (%)
  - Power (**)
  - Integer division (//)
  - Modules (mod)
  - Square root (sqrt)
  - maximum (max)

```
# Examples (necessary to use 'is')
?- ADD is 45 + 12
?- SUB is 45 - 12
?- MUL is 45 * 12
?- POW is 12 ^ 2
```

- **Comparison operators**

Operator	Meaning	Example
X < Y	greater than	X < Y
X > Y	less than	X > Y
X >= Y	greater than or equal to	X >= Y
X =< Y	less than or equal to	
X =:= Y	equal values	1+2 =:= 2+1 -> TRUE
X =\= Y	not equals	check X \= Y
X = Y	equal patterns	1+2 = 2+1 -> FALSE

```
# Examples
?- 13 * 2 > 12 + 3.             # yes
?- 13 * 2 < 12 + 3.             # no

# Equal operator
?- 12 + 13 =:= 13 + 12.         # yes
?- ?- 12 + 13 =:= 13 + 14.      # no
```

Input and Output

```
cube :-
    write('Write a number: '),
    read(Number),
    process(Number).
process(stop) :- !.
process(Number) :-
    C is Number * Number * Number,
    write('Cube of '),write(Number),write(': '),write(C),nl,cube.
```

- tab() predicate -> blank space between variables, etc.

```
?- write('hello'),tab(15),write('world').
# hello               world
?- write('We'),tab(5),write('use'),tab(5),write('tabs').
# We     use     tabs
```

Built-in predicates

- `var(X)`
  - when X is not initialized it will show true, otherwise false
  - it needs to be declared before

  ```
  ?- var(X).
  # yes
  ```

- `novar(X)`
  - when X is not initialized it will show false, otherwise true
  - it doesn't be to be declared
- `atom(X)`
  - it will return true when a non-variable term with 0 argument and a not numeric term is passed as X, otherwise false
- `number(X)`
  - it will return true if X is a number or a number declared variable, otherwise false

  ```
  X=5, number(X).
  # true
  ```

- `integer(X)`
  - it will return true when X is a positive or negative integer value, false otherwise
- `float(X)`
  - it will return true when X is a floating point value, false otherwisw
- `compound(X)`
  - compound structures: lists

Predicati ricorsivi

Cicli

- vengono implementanti attraverso la ricorsione

```
# Es: stampare i numeri da X fino a 10
count_to_10(10) :- write(10),nl.    # Final decision
count_to_10(X) :- write(X),nl, Y is X+1, count_to_10(Y).
```

```
# Es: determinare se un numero è intero o meno (DB is_int.pl)
# verifichiamo che sia ottenuto sommando ricorsivamente 1 a partire da 0
is_int(0).      # Caso base (0 è un numero intero)
is_int(X) :- is_int(Y), X is Y+1.
```

```
// Con la programmazione strutturata avremmo ottenuto:
bool is_int(int x) {
    if(x < 0)
        return false;
    if (x == 0)
        return true;    // caso base
return is_int(x-1);
}
```

```
# Es: creare un loop che prende il valore più piccolo e più grande
count_down(L,H) :-      # L -> valore più piccolo
    between(L,H,Y),     # if Lowest between H and Y
    Z is H - Y,
    write(Z),nl.

count_up(L,H) :-
    between(L,H,Y),
    Z is H + Y,
    write(Z),nl.
```

Liste

- struttura scalare (simile ai vettori) ma possono contenere elementi di tipi diversi
- sintassi: `[X,Y,Z,...]`
- possibilità di avere anche liste di liste: `[X,[Y,Z]]`

```
[X,3]=[51,Y]    # Dalla posizione si ottiene che X = 51 e Y = 3

[X,1]=[1,Y]     # In questo caso avremo X = Y e Y = 1
```

- vengono sempre viste come `[H|T]`:
    - **testa** (H) - primo elemento
    - **coda** (T) - elementi rimanenti

Ordinamento

- Fatti:
  - una lista vuota è ordinata
  - una lista con solo la testa è ordinata
- Regole:
  - una lista con testa e coda è ordinata se la testa è minore della coda AND la coda è a sua volta è ordinata

```
# DB ordina.pl

ordinata([]).
ordinata([H]).
ordinata([H|T]) :- H < T, ordinata([T]).

## WARNING: confronto tra un elemento (H) e una lista (T)
# I problemi sorgono con una lista composta da tre o più elementi
## FIX: definire la coda come lista a sua volta (T|L)

ordinata([]).
ordinata([H]).
ordinata([H|[T|L]]) :- H < T, ordinata([T|L]). # T = head della coda

# N.B: in questo caso l'output è solo True/False
```

Lunghezza

- Fatti:
  - la lunghezza di una lista vuota è zero
- Regole:
  - la lunghezza di una lista che ha almeno un elemento (`[_|T]`) è pari ad N se la lunghezza della coda T è pari ad N1, dove `N = N1+1`

```
# DB listlenght.pl

listlen([],0).
listlen([_|T],N) :- listlen(T,N1), N is N1+1.

# N.B: se scrivessimo [H|T] Prolog direbbe che c'è una variabile non utilizzata
(H)
```

Somma degli elementi

- Fatti:
  - la somma degli elementi di una lista vuota è pari a zero
  - la somma degli elementi di una lista con solo la testa è pari ad H
- Regole:
  - la somma degli elementi di una lista con più di due elementi è pari ad S se la somma degli elementi della coda è pari ad S1, con `S = S1+H`

```
sum([],0).
sum([H],H).
sum([H|T],S) :- sum(T,S1), S is S1 + H.
```

Concatenazione di due liste

- Fatti:
  - la concatenazione di una lista con una lista vuota restituisce la lista di partenza
- Regole:
  - la concatenazione di una lista `[H|T]` con una lista `L2` è pari a `[H|L3]` se l'unione della coda T con la lista L2 restituisce L3 (la coda della lista finale)

```
concat([],L2,L2).
concat([H|T],L2,[H|L3]) :- concat(T,L2,L3).

# Procedura (L1 = [1,2], L2 = [3,4]):
#
#          L1    L2    L3
# step 1: [1,2] [3,4] [1,]      -> passo H1 dentro L3
#         H1,T1
#
# step 2: [2,X] [3,4] [1,2,]    -> passo H2 dentro L3
#         H2,T2
#
# step 3: [X,X] [3,4] [1,2,]    -> L1 passato tutto, procedo con L2
#         H3,T3
```