

o Hello World

Creando un file "hello.pl" possiamo caricarlo dalla GUI di SWI con "[hello]."
Il punto (.) fa copiare a Prolog che c'è un endpoint.

hello.pl

```
main :- write ('this is a simple program'),  
        write ('hello world!').
```

[hello].
main.
↓
output

o Prolog Basics

- knowledge base → quello che Prolog sa di base
- facts
- rules } blochi di costruzione per la programmazione logica
- queries

Fatti: possiamo definire i fatti come relazioni esplicite fra oggetti e le proprietà che questi possono avere. I fatti sono veri di natura:

- Tom è un gatto
- A kumal piace la posta
- I peli sono neri
- A Nowaz piacciono i giochi
- Tom è lazy

Questi sono fatti veri. Per precisione, => queste sono affermazioni che consideriamo essere sempre vere e prescindere.

Per scrivere fatti usiamo delle regole:

- nomi di proprietà/relazioni sono lower case letters
- le relazioni appaiono come primo termine
- gli oggetti sono separati da virgolette e fra parentesi
- un punto ":" serve per completare un fatto
- gli oggetti sono lowercase, lettere o numeri o stringhe fra apostrofi:
color(penning, 'red'). phoneNo(Luca, 3429730010).

Sintassi: relazione (obj1, obj2 ...)

- Tom è un gatto → cat(tom)
- A kumal piace la posta → loves-poste(kumal)
- I peli sono neri → of-color(hair, black)
- A Nowaz piacciono i giochi → loves-games(Nowaz)
- Tom è lazy → lazy(Tom)

NB: usare sempre LOWER CASE! Le maiuscole indicano una variabile!

Regole: possiamo definire le regole come relazioni implicite fra oggetti.
Quindi le regole sono vere condizionalmente. Quando una condizione è vera allora anche il predicato è vero.

Lily è felice se bolla

Tom ha fame se sta cucinando

Jock e Billy sono amici se entrambi giocano a calcio \Rightarrow andare a giocare se la scuola è chiusa ed è libero

Quando è vera la porta destra, allora la porta sinistra è vera. Queste regole sono vere condizionali.

Il simbolo ":-" si legge "if" o "implica che". Viene chiamato anche neck (collo). Il simbolo sx viene chiamato Head (testa) mentre a destra abbiamo il body (corpo). Usiamo la virgola "," come congiunzione e la virgola ";" come discongiunzione.

Sintassi:

nome-regola (obj₁, obj₂, ...) :- fact/rule (obj₁, obj₂, ...).

Se una clausola è tipo: P:- Q,R possiamo scrivere anche P:-Q, P:-R.
 $P:-Q, R, S, T, U \Rightarrow P:- (Q, R); (S, T, U)$. o $P:-Q, R$. e $P:- S, T, U$.

hungry(lily) :- dances(lily).

hungry(tom) :- is-cooking(tom).

friends(jock, Billy) :- playsoccer(jock), playsoccer(billy).

go-play(ryan) :- isClosed(school), isFree(ryan).

Queries: sono domande sulle relazioni fra oggetti e le loro proprietà.
Una query può essere una qualsiasi domanda.

Tom è un gatto?
Keanu sa la pasta?
Lily è felice?
Ryan giocherà?

Stando alle query, Prolog (o un L di programmazione logica) può trovare le risposte alle domande e ritornarle.

Knowledge Base in Logic Programming:

Fatti e regole insieme compongono la KB. Di conseguenza possiamo definire la KB come collezione di fatti e regole. La KB viene usata per trovare una risposta alle query poste.

esempio

↓

KB:

sing-a-song (ononye).
listens-to-music (rohit).

] fatti

listens-to-music (ononye) :- sing-a-song (ononye).
happy (ononye) :- sing-a-song (ononye).
happy (rohit) :- listens-to-music (rohit).
plays-guitar (rohit) :- listens-to-music (rohit).

] regole

Quando sappiamo che rohit è felice se ascolta la musica e suona se ascolta, quindi è felice quando suona? Teme.

• Relationships in Prolog

Nei programmi Prolog, specificano relazioni fra oggetti e proprietà. Espresso con fatti e regole. Se avessimo una affermazione tipo "Anit possiede una bici" e poi poniamo la query stiamo cercando una risposta alla relazione di appartenenza.

Alcuni tipi di relazione vengono espressi con regole e non fatti, tipo "sono fratelli se hanno la stessa madre e sono maschi".

↓ es:

* { parent (sudip, roje).
parent (sudip, boje).
male (roje).
male (boje).
brother (X, Y) :- parent (Z, X), parent (Z, Y), male (X), male (Y). } esprimono la base (KB) regole in prolog

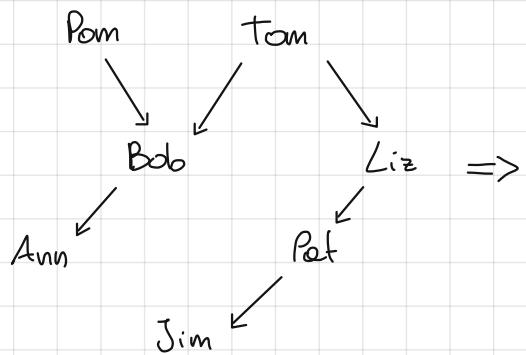
?:- brother (roje, boje).] poniamo la query

yes ottieniamo la risposta

* Dobbiamo anche dire che x e y devono essere persone diverse, altrimenti roje o boje possono essere fratelli di se stessi.

Indichiamo differenze con " \neq ". X è diverso da $Y \rightarrow X \neq Y$

Relazione familiare complicata



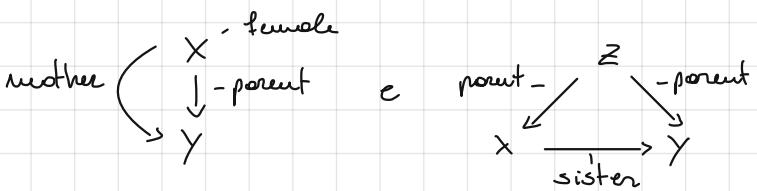
parent (pom, bob).
 parent (tom, bob).
 parent (tom, liz).
 parent (bob, ann).
 parent (liz, pet).
 parent (pet, jim).

Punti chiave .
 - Abbiamo definito la rel.
 genitore con n tuple di obj
 - Si può fare query su chi è
 genitore di chi in Prolog
 - Gli argomenti possono essere
 costanti (pet, jim) ma anche
 obj generici come x, y.
 pet → atomo X → variabile

Alcuni fatti possono essere scritti in più modi:
 per dire che Jane è femmina possiamo scrivere

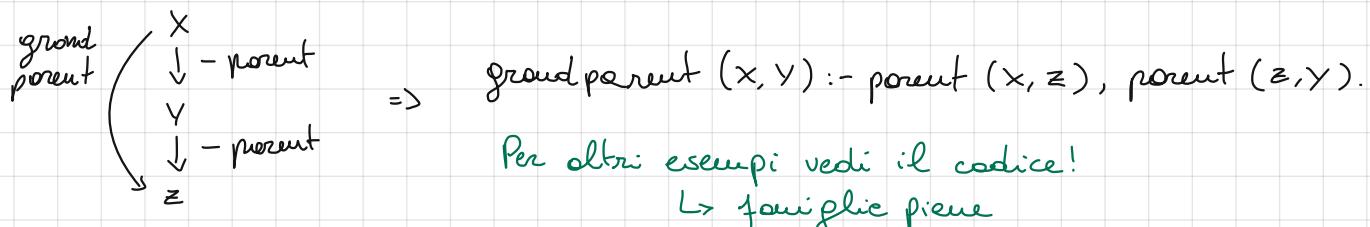
female (jane). oppure sex (jane, feminine).
 male (tom). oppure sex (tom, masculine).

Per una relazione "madre" e "sorella" possiamo dire:



mother (x, y) :- parent (x, y), female (x).
 sister (x, y) :- parent (z, x), parent (z, y), female (x), x \= y .

Relazioni più profonde come nonno possono essere espresse tipo:

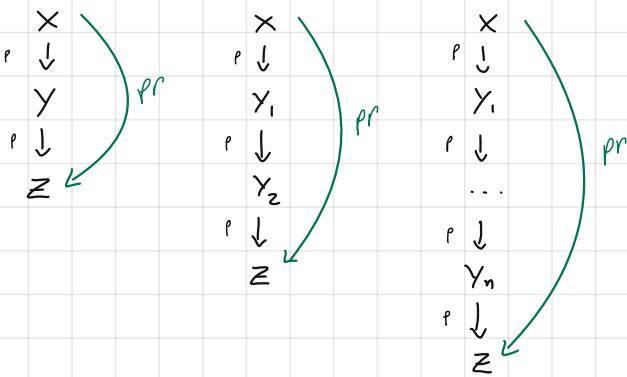


Tracciare l'output.

In Prolog è possibile tracciare l'output → entrore in trace mode "trace".
 Per uscire basta usare "notrace." La trace mode ci mostra le query,
 le variabili usate e gli assegnamenti fra variabili e atomi.

Ricorsione in relazioni familiari:

Abbiamo definito sopra relazioni statiche di natura. Possono però creare anche relazioni ricorsive in Prolog, espresse come:



L'arco $\rho \rightarrow$ è parent
L'arco $\rho \leftarrow \rho$ è predecessor

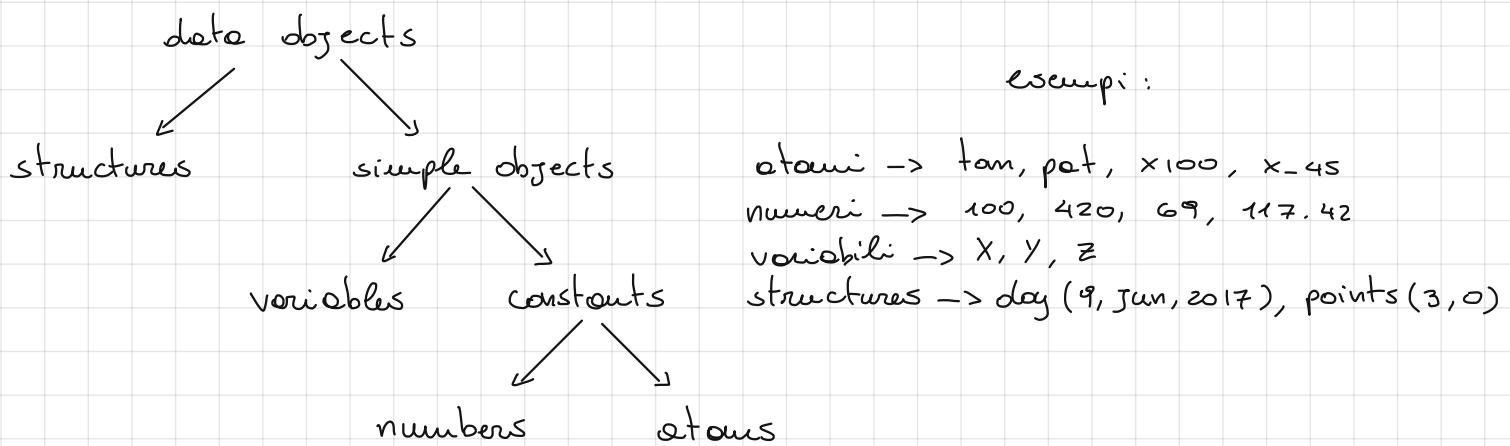
Vediamo quindi come la relazione predecessor sia ricorsiva e la esprimiamo

$\text{predecessor}(X, Z) :- \text{parent}(X, Z).$

$\text{predecessor}(X, Z) :- \text{parent}(X, Z), \text{predecessor}(Y, Z).$

• Data objects in Prolog

Gli oggetti dati possono essere divisi in categorie:



Atomi e variabili:

Gli atomi sono una variazione delle costanti \rightarrow nomi o oggetti.

Buona pratica è descrivere gli atomi con stringhe di lettere, numeri e underscore, che iniziano sempre con lettere lower case.

\hookrightarrow ezhor, b-45, n3t-pawn, morcolino420

Caratteri speciali (stringhe):

NB: in Prolog certe stringhe hanno funzioni predefinite:

<--->
=====>
...
:::
:=

Possiamo anche racchiudere fra quotes delle stringhe, per forza che se davessero avere caratteristiche speciali queste non vengono calcolate.

Le variabili che hanno sempre uno maiuscolo tipo "Mark" se racchiuso, diventa un semplice atom -> 'Mark'.

Numeri:

Anche i numeri sono una variazione di costanti. Il range in Prolog va da -16383 a 16383 per gli integer normali. Possono essere sia negativi che positivi. C'è il supporto per i reali, ma vengono usati poco dato la natura del Prolog.

Oggetti semplici:

Le variabili sono obg semplici. Possono essere usate nei casi misti in precedenze. Possono definirle come letterali o stringhe di letterali con underscore. Per dichiarare una variabile come tale, deve iniziare con una maiuscola.

Variabili anonime:

Sono variabili che non hanno un nome. Vengono dichiarate usando un underscore '_'. De notare che ogni variabile anonima viene trattata differentemente e non vengono calcolate insieme. Sono diverse.

L'utilizzo è semplice -> esempio: Jim odia tom, Pat odia Bob. Se Tom volesse controllare chi lo odia, può usare una variabile. Questo funziona solo esplicitando chi, se Tom volesse controllare se qualcuno (senza sapere chi) lo odia, userebbe una variabile anonima.

es: hates(jim, tom).
hates(pat, bob).

?- hates(x, tom).

x = jim ?;

x = peter

yes

?- hates(_, tom).

true ?;

yes

◦ Operatori in Prolog:

Comparison operators:

Usati per comporre due espressioni o stati:

$X > Y$	ug>
$X < Y$	min.
$X \geq Y$	ug. ug.
$X \leq Y$	min.ug.
$X =:= Y$	egualità
$X =\backslash= Y$	disegualità

NB: $1+2 =:= 2+1$ ritorna true ma $1+2 = 2+1$ ritorna false. Questo perché nel primo caso controlla se $1+2$ uguale a $1+2$ mentre il secondo controlla che l'intera seq "1+2" e "2+1" lo sono.

Arithmetic operators:

Sono usati per performare operazioni aritmetiche:

+	addizione
-	sottrazione
*	moltiplicazione
/	divisione
**	potenza
//	integer division
mod	modulo

◦ Loops and decision making

Loops:

I loop sono usati per ripetere blocchi di codice più volte. In generale non sono cicli for/while come li conosciamo in Java, C++ ecc.

I blocchi eseguiti più volte usano la logica dei predicati ricorsiva. Non ci sono loop diretti in Prolog ma possono comunque simularli.

es:

```
count-to-10(10) :- write(10), nl.  
count-to-10(X) :-  
    write(X), nl,  
    Y is X+1,  
    count-to-10(Y).
```

Decision making:

Portiamo degli statement if-else. Quando vogliamo fare match su certe condizioni, usiamo questi statement.

If <condition> is true, then <do this>, Else <do that>

In Prolog, le differenze di tutti gli altri linguaggi, avviamente non abbiamo questi costrutti. Per creare degli if-else usiamo i predicati:

gt(X, Y) :- X >= Y, write('X is greater or equal').] if then else
gt(X, Y) :- X < Y, write('X is smaller').] else

gte(X, Y) :- X > Y, write('X is greater').] if
gte(X, Y) :- X == Y, write('X is equal').] else if
gte(X, Y) :- X < Y, write('X is smaller').] else

o Congiunzioni e disgiunzioni

Prolog crea congiunzioni e disgiunzioni mediante le logiche AND e OR, come negli altri linguaggi di programmazione.

Congiunzione:

La congiunzione (AND) può essere implementata con le virgolette (,).
Quindi, due predicati separati da una virgola vengono "joinati" con un AND.
Supponendo di avere un predicato parent(john, bob). e uno male(john), possiamo creare il predicato father(john, bob) :- male(john), parent(john, bob)

John è padre di bob if John è maschio AND John è genitore di bob

Disgiunzione:

io le chiama così
↓

La disgiunzione (OR) può essere implementata con la semicolon(;) .
Quindi, due predicati separati da una semic. sono "joinati" con un OR.
Supponiamo di avere un predicato father(john, bob). e uno mother(lily, bob).
Se creiamo il predicato child_of() questo sarà vero (per bob) se father() o mother() sono veri.

es: father(john, bob). male(john).
 mother(lily, bob). + female(lily).

congiunzione → father(X, Y) :- parent(X, Y), male(X, Y).

disgiunzione → child_of(X, Y) :- father(X, Y); mother(X, Y).

• Liste in Prolog

Le liste sono strutture dati usate in differenti casi per programmazione non numerica. Sono usate in Prolog per contenere atomi come collezione.

Representazione delle liste:

Sono consistenti di un numero n di oggetti. Sono rappresentate, per esempio, come `[red, blue, green]` e sempre racchiuse da parentesi quadre.
Una lista può essere vuota (ed esiste come atomo `[]`), altrimenti viene distinta la testa (`head`) della lista che è il primo elemento e il resto che viene detto coda (`tail`).

Sempre usando `[r, g, b]` r è `head` e g, b è `tail`. La coda di una lista e la testa vengono trattate "quasi" come due liste diverse.

Consideriamo la lista $L = [a, b, c]$. Se scriviamo $Tail = [b, c]$, allora possiamo anche scrivere la lista come $L = [a | Tail]$ dove la barra verticale " $|$ " separa testa e coda.

La seguente rappresentazione è valida in diversi modi:

- $[a, b, c] = [x | [b, c]]$
- $[a, b, c] = [a, b | [c]]$
- $[a, b, c] = [a, b, c | []]$

Quindi per dare una definizione:

Una lista è una strutt. dati vuota o consistente di due parti - una testa e una coda. La coda stessa è una lista.

Operazioni di base sulle liste:

1. Membership checking → controlla se un elem. è membro
2. Length calculation → calcola lunghezza lista
3. Concatenation → usata per join/odd due liste
4. Delete items → cancella un elem. specifico
5. Append items → aggiunge una lista in un'altra (come oggetto)
6. Insert items → inserisce un elem. specifico

1. Membership

Possiamo controllare se un elem. X fa parte di una lista $L \rightarrow$ per farlo definiamo un predicato: `list_member(X, L)`. Ricordiamo che X può essere sia la testa che la coda, che sono due casi diversi.

`list_member(X, [X | _]).`

`list_member(X, [_ | TAIL]) :- list_member(X, TAIL).`

2. Length

list_length (L, N) con L lista ed N verrà istanziato al numero di elem. in lista.
La lista è o vuota (0) oppure 1+ elementi della coda.

list_length ([], 0).

list_length ([_ | TAIL], N) :- list_length (TAIL, N1), N is N1 + 1.

3. Concatenazione :

Aggiunge gli elementi della seconda lista nella prima.

Se una lista è vuota e la seconda è L , il risultato sarà L . Se la prima non è vuota concateniamo Tail di L_1 con L_2 ricorsivamente fino ad ottenere [Head | NewList].

list_concat ([], L, L).

list_concat ([X | L1], L2, [X, | L3]) :- list_concat (L1, L2, L3).

3. Delete :

Per cancellare x da L abbiamo 3 casi:

- x è nello head → la lista risultante sarà la tail
- x è nello tail → cancelliamo ricorsivamente da lì
- x è l'unico elemento → dopo la canc. la lista ritorna vuota

list_delete (x, [x], []).

list_delete (x, [x | L1], L1).

list_delete (x, [Y | L2], [Y | L1]) :- list_delete (x, L2, L1).

4. Append

Dobbiamo considerare due casi: aggiungere due liste insieme o aggiungere una come opposto dell'altra. Se l'elem. è già presente in lista append non va.

Se A è elem. e L_1 lista, l'output è sempre L_1 se lì già A .

Altrimenti la nuova lista sarà $L_2 = [A | L_1]$.

list_member (x, [x | _]).

list_member (A, T, T) :- list_member (A, T).

list_append (A, T, T) :- list_member (A, T), !.

list_append (A, T, [A | T]).



Qui inseriscono un cut (!).
Se quelle lines ha successo,
tagliano così da non dover
eseguire le prossime istruzioni

5. Insert

Inseriamo X in L . La risultante sarà R . Se usiamo $\text{list_insert}(X, L, R)$ possiamo anche usare $\text{list_delete}(X, L, R)$. Possiamo inserire X in una qualsiasi posizione di L .

$\text{list_delete}(X, [X], []).$

$\text{list_delete}(X, [X_1, L_1], L_1).$

$\text{list_delete}(X, [Y | L_2], [Y | L_1]) :- \text{list_delete}(X, L_2, L_1).$

Repositioning operations

- | | |
|----------------|---|
| 1. Permutation | → cambia posizioni degli elementi e genera le possibilità |
| 2 Reverse | → riarrangi al contrario |
| 3. Shift | → sposta un elemento a sx e rotazione |
| 4. Orders | → verifica che la lista sia ordinata |

1. Se la prima lista è vuota anche la seconda lo sarà. Altrimenti se non lo è e ha forma $[X | L]$, permutiamo L ottenendo L_1 e inserendo X in una qualsiasi posizione di L_1 .

$\text{list_delete}(X, [X | L_1], L_1).$

$\text{list_delete}(X, [Y | L_2], [Y, L_1]) :- \text{list_delete}(X, L_2, L_1).$

$\text{list_perm}([], [])$.

$\text{list_perm}(L, [X | P]) :- \text{list_delete}(X, L, L_1), \text{list_perm}(L_1, P).$

2. Da una lista $L = [a, b, c, d, e]$ ottieniamo $L = [c, d, a, b, e]$. Se $L = \emptyset$ allora $L_R = \emptyset$. Altrimenti $[Head | Tail]$ invertiamo $Tail$ e concateniamo $Head$.

$\text{list_concat}([], L, L).$

$\text{list_concat}([X_1, L_1], L_2, [X_1, L_3]) :- \text{list_concat}(L_1, L_2, L_3).$

$\text{list_rev}([], [])$.

$\text{list_rev}([Head | Tail], Reversed) :-$

$\text{list_rev}(Tail, Reversed), \text{list_concat}(RevTail, [Head], Reversed).$

3. Spostiamo un elemento a sx rotationally. Da $L = [a, b, c, d]$ ottieniamo $L = [b, c, d, a]$. Esprimiamo $[Head | Tail]$, e concateniamo ricorsivamente $Head$ dopo $Tail$, così da avere uno shift alla fine. Si può anche usare per controllare se due liste sono shiftate in una posizione o no.

$\text{list_concat}([], L, L).$

```
list-concat([X,  
           | L1], L2, [X1 | L3]) :- list-concat(L1, L2, L3).  
list-shift([Head | Tail], shifted) :- list-concat(Tail, [Head], Shifted).
```

4. Definiamo un predicato `list-order(L)` che controlla se L è ordinata o meno. Quindi se $L = [1, 2, 3, 4, 5, 6]$ il risultato è true. Se c'è solo un elemento, è già ordinata. Altrimenti prendiamo i primi due elem x, y come Head e il resto come Tail. Se $x \leq y$ richiamiamo con parametro $[y | tail]$ di modo da controllare ricorsivamente il prossimo elemento.

```
list-order([X, Y | tail]) :- X <= Y, list-order([Y | tail]).  
list-order([X]).
```

Set operation on lists:

Genera tutti i possibili subset di un dato set. Se il set è $[a, b]$ risulterà $[]$, $[a]$, $[b]$, $[a, b]$. Per farlo il predicato prenderà L e ritornerà ogni subset in X . Se la lista è vuota, il subset è vuoto.

Trova il subset ricorsivamente tenendo la testa e fa un'altra chiamata ricorsiva dove rimuoviamo la testa.

```
list-subset([], []).  
list-subset([H | T], [H | subset]) :- list-subset(T, subset).  
list-subset([H | T], subset) :- list-subset(T, subset).
```

con H e T
Head, Tails

Union Operation on lists:

Unisce due liste L_1 e L_2 in L_3 . Se le due liste contengono lo stesso elemento, questo sarà presente una sola volta.

```
list-member(X, [X1-]).  
list-member(X, [- | TAIL]) :- list-member(X, TAIL).
```

```
list-union([X | Y], Z, W) :- list-member(X, Z), list-union(Y, Z, W).  
list-union([X | Y], Z, [X1 | W]) :- \+ list-member(X, Z), list-union(Y, Z, W).  
list-union([], Z, Z).
```

operatore NOT

Intersection operation:

Prendiamo L_1 e L_2 , intersechiamo e ritorniamo L_3 . L'intersezione ritorna gli elementi presenti sia in L_1 che L_2 . Se $L_1 = [a, b, c, d, e]$ e $L_2 = [a, e, i, o, u]$ allora $L_3 = [a, e]$.

```
list-member(X, [X1-]).  
list-member(X, [- | TAIL]) :- list-member(X, TAIL).
```

list-intersect ([X|Y], Z, [X|W]) :-

list-member(X, Z), list-intersect(Y, Z, W).

list-intersect ([X|Y], Z, W) :-

\+ list-member(X, Z), list-intersect(Y, Z, W).

list-intersect ([], Z, []).

o Misc operations on lists:

- Even and odd length finding \rightarrow verifica se la lista ha # elemenatori disponibili.
- Divide \rightarrow divide la lista in due, circa della stessa lunghezza.
- Max \rightarrow ritorna l'elemento più grande della lista.
- Sum \rightarrow ritorna la somma degli elementi in una lista.
- Merge Sort \rightarrow opera il merge sort su una lista

↓

! Per approfondire, vedi il codice sulle Misc CP

Ordina le liste usando 3 passaggi:

- prendi la lista e splitta in due sottoliste (ricorsivamente).
- ordini le split e unisci
- la lista sarà ordinata

o Ricorsione e strutture in Prolog

Ricorsione:

È una tecnica dove un predicato (o insieme ad altri) usa se stesso più volte per arrivare al goal.

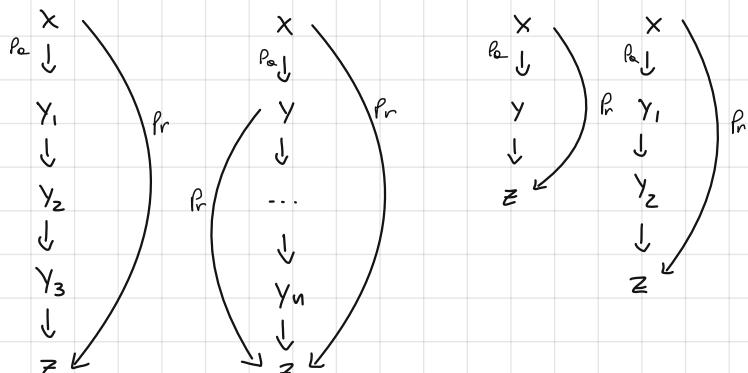
Esempio:

is-digesting(X, Y) :- just-eat(X, Y).

is-digesting(X, Y) :- just-eat(X, Z), is-digesting(Z, Y).

Il predicato è ricorsivo di natura: se diciamo just-eat(deer, grass) allora is-digesting(deer, grass) è vero. Se diciamo is-digesting(tiger, grass) sarà vero se is-digesting(tiger-grass) :- just-eat(tiger, deer), is-digesting(deer, grass).

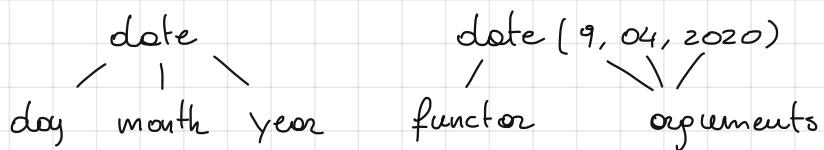
Possiamo vedere un esempio anche con la famiglia e predecessore:



parent(X, Z) :- parent(X, Z).
parent(X, Z) :- parent(X, Z), parent(Z, Y), parent(X, Y).
parent(X, Z) :- parent(X, Y), parent(Y, Z).

Strutture:

Le strutture sono oggetti dati che contengono multiple componenti. Una data, per esempio, può essere vista come albero:



NB: le strutture possono contenere altre strutture. Le strutture possono essere viste come alberi. Prolog può essere visto come linguaggio per processare alberi.

Matching in Prolog:

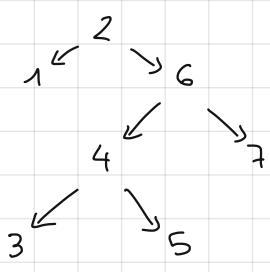
Viene usato per controllare se due termini dati sono identici o se le variabili in entrambi i termini possono avere gli stessi oggetti dopo essere state istanziate.

$\text{date}(D, M, 2020) = \text{date}(D_1, \text{apr}, Y_1)$ → indice che $D=D_1$, $M=\text{apr}$ e $Y_1=2020$

Per vedere se due termini S e T matchano, seguiamo delle regole:

- se S e T sono costanti, $S=T$ se sono lo stesso oggetto
- se S è una variabile e T è qualsiasi cosa, $T=S$.
- se T è una variabile e S è qualsiasi cosa, $S=T$.
- se S e T sono strutture, $S=T$ se:
 - S e T hanno lo stesso functor
 - tutti i loro argomenti matchano

Albero binario



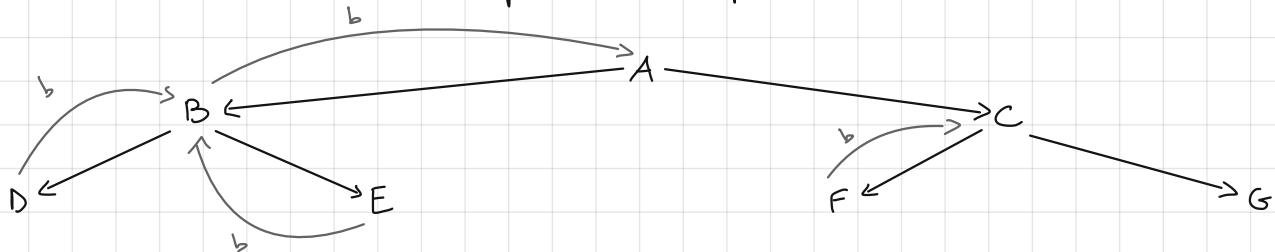
Esempio di albero binario usando una strutt. ricorsiva:

$\text{node}(2, \text{node}(1, \text{nil}, \text{nil}), \text{node}(6, \text{node}(4, \text{node}(3, \text{nil}, \text{nil}), \text{node}(5, \text{nil}, \text{nil})), \text{node}(7, \text{nil}, \text{nil}))$

Ogni nodo ha 3 campi: date e due nodi.

o Backtracking in Prolog

Il backtracking è una procedura in cui Prolog cerca valori di verità di diversi predicati controllando se sono veri o no. Finché non raggiunge destinazione continua e poi va in stop.



Se da $A \rightarrow G$ abbiamo regole e fatti, partiamo da A fino a raggiungere G . Il percorso corretto sarebbe $A \rightarrow C \rightarrow G$ ma prima andiamo da A a B e B a D . Non essendo D la destinazione, torniamo a B e andiamo a E . Visto che la situazione è la stessa torniamo a B poi A . Ripetiamo fino alla destinazione.

Come funziona il backtracking:

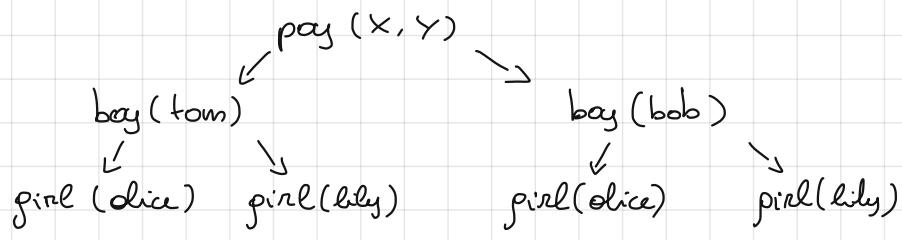
Durante il backtracking possono esserci diverse risposte, possiamo premere $(;)$ per proseguire uno per uno, altrimenti si ferma al primo risultato.

Esempio: due persone X e Y possono stare a vicenda, ma la condizione è che solo i maschi possono stare le femmine. Quindi X è male e Y female.

KB:

```

boy(tom)
boy(bob)
girl(alice)
girl(lily)      =>
pay(X,Y) :- boy(X), girl(Y).
    
```



Preventing backtracking:

Vediamo qualche svantaggio del backtracking: a volte scriviamo predicati più volte, quando serve qualcosa di ricorsivo per esempio. In certi casi, il backtracking non controllato può essere dispendioso per il programma. Per questi casi, usiamo la funzione CUT.

Esempio:

```

rule1 &minus; if X < 3 then Y=0
rule2 &minus; if 3 <= X and X < 6 then Y=2
rule3 &minus; if 6 <= X then Y=4
    
```

11

$f(x, 0) :- X < 3 \% \text{ Rule 1}$
 $f(x, 2) :- 3 = < X, X < 6 \% \text{ Rule 2}$
 $f(x, 4) :- 6 = < X. \% \text{ Rule 3}$

→ Ora, se chiediamo $f(1, Y)$, $Z < Y$ il 1° goal $f(1, Y)$ istanzia $Y = 0$. Il 2° diventa $Z < 0$ e fallisce. Prolog esce con backtracking le regole 2 e 3 che non vanno bene.

Quindi se le regole sono mutuamente esclusive, e solo una può essere vera, quando una è vera non ha più senso continuare a testare per le altre.

↓ introduciamo il cut

$f(x, 0) :- X < 3, !. \% \text{ Rule 1}$
 $f(x, 2) :- 3 = < X, X < 6, !. \% \text{ Rule 2}$
 $f(x, 4) :- 6 = < X. \% \text{ Rule 3}$

Negation as failure

Vogliamo permettere un fail se una condizione non è soddisfatta.

Se "A myy piacciono tutti gli animali ma non i serpenti":

→ if x is a snake, Mary likes x not true

→ altrimenti se x è un animale, Mary likes x

↓

$\text{likes}(\text{mary}, x) :- \text{snake}(x), !, \text{fail}.$

$\text{likes}(\text{mary}, x) :- \text{animal}(x)$

• Different and not

Definiamo due predicati: different e $\text{not} \rightarrow \text{different}$ controlla se due argomenti (del predicato) sono uguali o meno. Sono se sono uguali ritorna falso, altrimenti vero. Il predicato not viene usato per negare un statement, quindi se lo statement è true allora not statement sarà falso e viceversa.

$\text{different} \rightarrow x \neq y$ non sono letteralmente gli stessi

$x \neq y$ non matchano

i valori aritmetici delle espressioni x e y non sono uguali

$\text{different}(x, x) :- !, \text{fail}$

$\text{different}(x, y).$

$\text{not} \rightarrow$ quando un statement è vero not(statement) sarà falso e viceversa.

$\text{not}(P) :- P, !, \text{fail}; \text{true}.$

se P è vero cut and fail altrimenti è vero

o Input and outputs

Write(), read() e tab():

Possiamo usare il predicato write() per scrivere a console l'output. È un predicato built-in semplice ed equivale allo print che conosciamo.

Il predicato read() legge un valore da console. L'input può essere preso come valore e poi processato. Può anche essere usato per leggere da files.

Il predicato tab() può essere usato per inserire blank-spaces mentre scriviamo qualcosa. Prende un int come argomento e inserisce tanti spazi quanto è l'int.

Lettura e scrittura da file:

tell and told:

Per leggere e scrivere da file possiamo usare predici built-in:
per la scrittura possiamo usare tell() → prende un filename come argomento.
Se il file non esiste, lo crea. Il file viene aperto e possiamo scrivere finché non usiamo told. Con tell possiamo aprire più files, poi told li chiude.
→ dopo tell dobbiamo usare write("") per scrivere nel file.

See and seen:

Per leggere da un file e non da tastiera, combiniamo l'input stream: usiamo il predicato see() che prende filename come argomento. Quando l'operazione è completata usiamo seen per interrompere e trasferire il controllo di nuovo alla console.

File processing:

Per leggere / processare tutto il contenuto di un file usiamo una clausola.

```
process-file :-  
    read(Line),  
    Line \= end-of-file,  
    process(Line).  
process-file :- !.
```

```
process-file :-  
    write(Line), nl,  
    process-file.
```

Manipolazione dei caratteri:

Possiamo usare put(C) per scrivere un char alla volta nello stream di output. Possiamo usare get_char(C) per leggere un singolo char dallo stream di input. Se vogliamo il codice Ascii usiamo get_code(C).

Costruire atomi:

Possiamo costruire un atomo da una lista di caratteri con atom_chars() e atom_codes(). In entrambi i casi il primo argomento è una variabile ed il secondo una lista.

Decomporre atomi:

In modo simile possiamo usare gli stessi predici per ottenere una sequenza di caratteri da un atomo. La differenza è che in ogni caso il primo argomento sarà un atomo ed il secondo una variabile.

Consulting in Prolog:

Il consulting è una tecnica usata per unire predici da file diversi. Possiamo usare consult() e passare come argomento il filename per ottenere il predicato del file. Da notare che se due file hanno clausole diverse, tutto funziona, ma se ci sono predici uguali le copie verranno sovrascritte ogni volta che vengono lette da un nuovo file.

• Coroutines

dif(X,Y) → vero se due elem sono diversi; è un constraint.

• Built in predicates:

Nella maggior parte dei casi in Prolog dobbiamo definire i nostri predicati. Ci sono però 3 tipi di predicati built-in, usati per:

- identificare termini
- decomporre strutture
- collezionare soluzioni

<code>var(X)</code>	ha successo se X è una var non istanziata
<code>novar(X)</code>	ha successo se X non è una var o è istanziata
<code>atom(X)</code>	vero se X è un atomo
<code>number(X)</code>	vero se X è un numero
<code>integer(X)</code>	vero se X è un integer
<code>float(X)</code>	vero se X è un numero reale
<code>atomic(X)</code>	vero se X è un numero o un atomo
<code>compound(X)</code>	vero se X è una struttura
<code>ground(X)</code>	ha successo se X non contiene alcuna var non istanziata

arity = arità \rightarrow il # di arg. che un operando prende da una funzione

Decomposing structures:

Vediamo un altro gruppo di predicati built-in: quando usiamo compound structures non possiamo usare una variabile per controllore o fare un functor, ritornerebbe errore. Il nome di un functor non può essere rappresentato da una variabile. Vediamo i predicati:

- `functor(T, F, N)` predicate: ritorna vero se F è il functor principale di T , ed N è arità di F .
- `arg(N, Term, A)` predicate: ritorna vero se A è l' n -esimo arg in $Term$.
- `../2` predicate: $Term = ..L$ vero se L è una lista contenente il functor di $Term$,
 $\downarrow \dots$ due punti seguite dai suoi argomenti.

Collecting all solutions:

Vediamo una terza categoria di predicati built-in: abbiamo visto che per trovare tutte le soluzioni di un pool usiamo la semicolon.

1. `findall` tutti e 3 prendono 3 argomenti di input. Sono anche consueti:
2. `setoff` \Rightarrow come meta-predicates. Sono usati per manipolare le Prolog's Proof strategy.
3. `bagof`

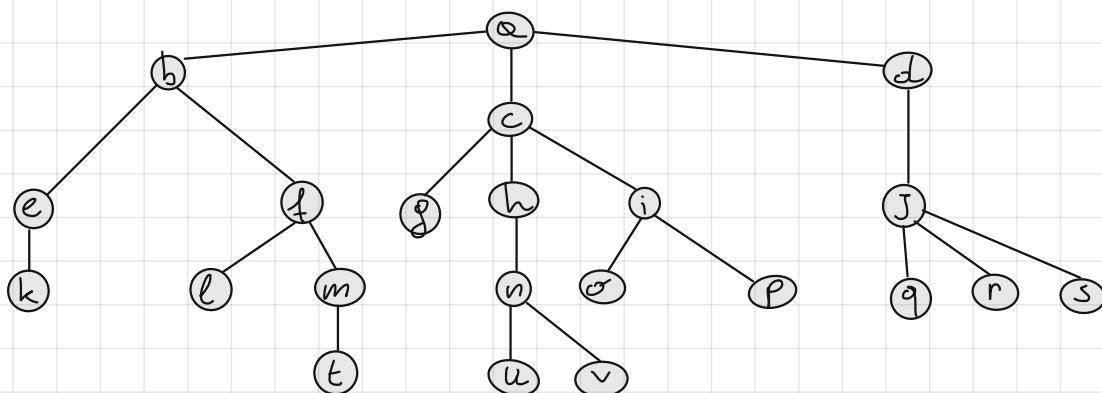
1. `findall/3` è usato per formare una lista L di tutte le soluzioni X del predicato P . Vengono solvate nell'ordine in cui sono trovate. I doppiioni non vengono eliminati, il processo non termina mai per infinite soluzioni.
 $\text{findall}(X, (\text{pred. on } X, \text{ other goal}), L).$
2. `setof/3` è simile a `findall` ma rimuove i doppiioni e le risposte sono ordinate. Se vengono usate variabili nel goal, non appaiono nel primo argomento. Ritorna risultato diverso per ogni possibile istanziazione delle var.
3. `bogof/3` è simile a `setof/3` ma non rimuove duplicati e ha un ordine.

Predicati matematici:

- <code>random(L, H, X)</code>	\rightarrow random val fra L e H
- <code>between(L, H, X)</code>	\rightarrow tutti i valori fra L e H
- <code>succ(X, Y)</code>	\rightarrow app. 1 è assegnato a X
- <code>abs(X)</code>	\rightarrow absolute val di X
- <code>max(X, Y)</code>	\rightarrow val maggiore fra X e Y
- <code>min(X, Y)</code>	\rightarrow val minore fra X e Y
- <code>round(X)</code>	\rightarrow arrotonda vicino a X
- <code>truncate(X)</code>	\rightarrow converti float a int, cancella post virgola
- <code>floor(X)</code>	\rightarrow arrotonda per eccesso
- <code>ceiling(X)</code>	\rightarrow arrotonda per difetto
- <code>sqrt(X)</code>	\rightarrow radice quadrata

o Case study : tree data structure

Vediamo come implementare un albero e creare i nostri predici:



abbiamo operazioni:

- `op(500, xf, 'is-parent').`
- `op(500, xf, 'is-sibling').`
- `op(500, xf, 'same-level').`
- `leaf(Node).`

500 e' la priorita dell'operatore
`xf` indica che l'operatore e' binario

Vedi il codice di `case-study.pl` per approfondire

Sempre considerando lo stesso albero definiamo anche: `path(Node)` e `locate(Node)`.

`path(Node)` ci mostrera' il comune delle radice al nodo indicato.
`locate(Node)` ci mostrera' il nodo cercato dalla radice

Vedi `case-study2.pl` per approfondire.

introduciamo anche:

- `ht(Node, H) -> trova l'altezza del nodo usando setof/3. Ci mostra anche se un nodo e' foglia -> se lo e' $h=0$`
- `max ([X|R], M, A) -> calcola l'elenco (M) max delle liste`
- `height(N, H) -> usando setof/3. Trova il set di risultati usando ht(N, Z) per il template Z. Trova poi il max di set, val 0, e salva in H.`

Vedi `case-study3.pl` per approfondire

- Code directory tree

- Hello World

- |

- |

- Bonus: more Prolog stuff (unindexed)

Esercizi, esempi, spiegazioni

- Liste: operazioni, tutorial, esempi

Una lista può essere formata da qualsiasi cosa:

$$L_1 = [\text{mice}, \text{john}, \text{bunny}, \text{dog}]$$

$$L_2 = [\text{mice}, \text{mole}(\text{john}), X, 2]$$

$$L_3 = []$$

$$L_4 = [[], \text{dead}(z), [2, [b, c]], X]$$

lista con atomi

lista con var, atomi e fatti

lista vuota

lista con appelli e altre liste

→ E' sempre una lista, fatto il primo elemento

$$L = [H | T] \Rightarrow \text{sempre composta da Head e Tail}$$



Il primo elemento di una lista L

Soltanto le liste non vuote sono composte da 2 elementi. Se la lista contiene solo un elemento ($L = [x]$) allora $H = x$ e $T = []$

L'operatore " | " viene usato per decomporre le liste in head e tail:

$$?- [H | T] = [\text{mice}, \text{john}, \text{vincent}, \text{morde}]. \quad \text{mentre}$$

$$H = \text{mice}$$

$$\Rightarrow ?- [H | T] = []$$

$$T = [\text{john}, \text{vincent}, \text{morde}]$$

no

yes

Head e tail non hanno nulla di speciale; sono semplicemente due variabili. Infatti $[H | T]$ è uguale a $[x | y]$

$$?- [x | y] = [[], \text{dead}(z), [2, [b, c]], [], z].$$

$$x = []$$

$$y = [\text{dead}(z), [2, [b, c]], [], -7800]$$

$$z = -7800$$

→ Prolog ha associato z alle var interne -7800

" | " è uno strumento flessibile → supponiamo di voler sapere i primi due elementi di una lista e il resto della lista come terzo elemento:

$$?- [X, Y | W] = [[], \text{dead}(z), [2, [b, c]], [], z].$$

$$X = []$$

$$Y = \text{dead}(z)$$

$$W = [[2, [b, c]], [], z]$$

$$z = -7800$$

Quindi " | " può essere usato per dividere una lista in un qualsiasi punto, non soltanto head e tail.

NB: $[sx \downarrow | dx]$

↳ riceviamo quello che rimane, tolto quello a sx
quanti elementi vogliamo togliere dal fronte delle liste

- Variabili oramme:

Se volessimo trovare il 2° e 4° elem. di una lista, potremmo dire:

?- $[], \text{dead}(z), [z, [b, c]], [], z.$

?- $[X_1, X_2, X_3, X_4 | \text{Toil}] =$
 $[[], \text{dead}(z), [z, [b, c]], [], z].$

$X_1 = []$

$X_2 = \text{dead}(z)$

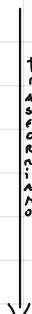
$X_3 = [z, [b, c]]$

$X_4 = []$

$\text{Toil} = [-3910]$

$z = -8910$

yes



Questo query è corretto, ma non ha senso dare esplicitamente 4 variabili quando ci interessa solamente X_2 e X_4 . Traianino anche Toil e z che probabilmente non ci interessano.

?- $[-, X, -, Y | -] =$
 $[[], \text{dead}(z), [z, [b, c]], z].$

$X = \text{dead}(z)$

$Y = []$

$Z = -10087$

yes



Il simbolo $-$ è una variabile oramme.
Più lo usiamo quando abbiamo bisogno di essere una variabile, ma non ci interessa a cosa viene istanziata. Sarà omessa nell'output e ogni variabile oramme è indipendente dall'altra.

Se per esempio, nelle nostre liste $[], \text{dead}(z), [z, [b, c]], z$. volessimo trovare $[b, c]$ avendo estratto la coda del terzo elemento, potremmo dire:

?- $[-, -, [- / X] | -] =$
 $[[], \text{dead}(z), [z, [b, c]], z].$

$X = [b, c]$

$Z = -10087$

yes

- Member e ricorsione:

Le liste sono ricorsive di struttura. Se volessimo scrivere un predicato che controlla se una X appartiene alla lista L , dovremmo usare la ricorsione:

1. $\text{member}(X, [X | T]).$ diciamo member di nuovo, ma senza head // fact
2. $\text{member}(X, [H | T]) :- \text{member}(X, T).$ // rule

↳ le regole è ricorsive!

1. X è membro della lista se è uguale alla testa di quella lista
 2. X è membro della lista se è membro della coda della lista.

Vediamo proceduralmente come funziona:

?- member(yolanda,[yolanda, trudy, kumor]).

yes

↳ immediato da (1) perché yolanda è testa della lista e Prolog può unificare le occorrenze di X nella clausola (1) di member/z.

?- member(kumor,[yolanda, trudy, kumor]).

La prima regola fallisce, quindi passiamo a quelle ricorsive: il nuovo goal diventa member(kumor, [trudy, kumor]). che fallisce di nuovo per via delle clausole (1). Quando viene chiamata la (2) di nuovo, il nuovo goal sarà member(kumor, [kumor]). e ovviamente finisce true.

Se ponessimo una query che non può essere soddisfatta tipo:

?- member(zed, [yolanda, trudy, kumor]). la ricorsione arriverebbe fino a member(zed, []). ma siccome [] non può essere splitto in due le regole non può essere applicate ed il predicato fallisce.

V2.0 member(X, [X|T]).
 member(X, [-|T]) :- member(X, T).

$T = [+|-]$

- Scorrere ricorsivamente le liste verso il basso

Lavorando con le liste, spesso vogliamo concatenarle o copiare pezzi di una nell'altra → usiamo lo scorrimento ricorsivo.

es: ?- qzb([a,a,a], [b,b,b]). → vogliamo yes

?- qzb([a,c,e], [b,d,f]) → vogliamo no

↓

Il caso più semplice è sempre quello della lista vuota → ?- qzb([], []).

Poi, quando la H di L₁ è "a" e la H di L₂ è "b" e le due code sono liste di a e b dello stesso lunghezza.

↳ qzb([a|Ta], [b|Tb]) :- qzb(Ta, Tb).

↙

Dice: il pred qzb/z ha successo se il 1° arg è una lista con H=a e se il 2° arg è una lista con H=b e ha successo chiamandolo sulle code.

Il caso delle liste vuote è anche quello che ci permette di soddisfare il goal

- Esempi di ricorsione:

digesting(X, Y) :- ate(X, Y) // escape clause: if X ate Y then X is digesting Y
digesting(X, Y) :-
 ate(X, Z),
 digesting(Z, Y).
] -> if X ate Z, and Z is digesting Y, then
X is digesting Y too.



ate(mosquito, blood(john)).

ate(frog, mosquito).

ate(stork, frog).

query: digesting(stork, mosquito).

il caso base fallisce, quindi provo con
il secondo -> digesting(stork, mosquito) :-

 ate(stork, Z),

 digesting(Z, mosquito).

NB: Quando scriviamo un predicato ricorsivo, servono due clausole: la clausola base, che ferma la ricorsione, e la clausola che contiene la ricorsione.



Ese:

KB:

NON-ricorsive:

child(anne, britt).
child(britt, corol).
child(corol, donna).
child(donna, emily).

descend :- child(X, Y).

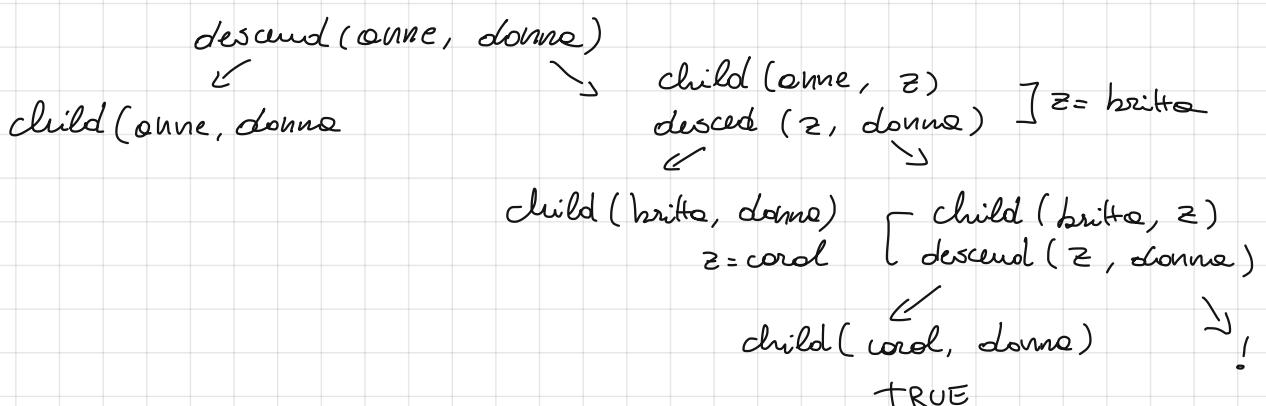
descend :- child(X, Z),
 child(Z, Y).

! funzione solamente x z
pernizioni e non oltre

ricorsivamente:

descend(X, Y) :- child(X, Y). -> se Y è figlia di X, allora discende
descend(X, Y) :- child(X, Z),] -> se Z è figlia di X e Y discende da Z,
 descend(Z, Y).] allora Y è discendente di X.

Quindi ?- descend(anne, donna) -> child(anne, donna) fallisce
e chiama il caso 2 -> child(anne, -633), -> trova da KB britt
 <- descend(-633, anne). e istanzia Z = britt
quindi chiama descend(britt, anne) -> child fallisce e chiama
child(britt, Z) che trova Z = corol quindi chiama
descend(corol, donna)
quindi il caso base è child(corol, donna) che è vero perché in KB
quindi true quindi descend(anne, donna) -> yes!



- Arithmetica in Prolog:

Per le quattro operazioni di base:

$$\begin{array}{lll} 6+2 = 8 & 8 \text{ is } 6+2 & x \text{ is } 6+2 \\ 6-2 = 4 & \Rightarrow 4 \text{ is } 6-2 & \Rightarrow x = 8 \\ 6 \times 2 = 12 & 12 \text{ is } 6 \times 2 & y \text{ is } 6 \times 2 \\ 6 \div 2 = 3 & 3 \text{ is } 6 \div 2 & y = 12 \end{array}$$

Prolog non svolge nessuna operazione aritmetica se non usiamo "is".
 $x = 3+2$ assegna x al termine complesso $3+2$ ma non altro.
È $x \text{ is } 3+2$ che fa svolgere l'operazione a Prolog.

Per il resto si usa mod $\rightarrow R \text{ is mod}(7,2) \quad R = 1$

Possiamo usare l'aritmetica per scrivere predici: `odd3double/2` prende un int, aggiunge 3 poi lo duplica:

`odd3double(X, Y) :- Y is (X+3)*2.`

Per sapere quanto sia lunga una lista, usiamo l'aritmetica e diamo una definizione ricorsiva: lista vuota è lunga 0, senno è $1 + \text{len}(T)$ dove $\text{len}(T)$ è la lunghezza della coda della lista.

↳ `len([], 0)`
`len([_|T], N) :- len(T, X), N is X+1`

Possiamo anche usare accumulatori, un po' come faremmo in C++/Java:

`accLen(List, Acc, Length)` inizialmente vale 0, poi +1 per ogni ricorsione

`accLen([_|T], A, L) :- Anew is A+1, accLen(T, Anew, L)`
`accLen([], A, A).`

+
`length(List, Length) :- accLen(List, 0, Length).` \Rightarrow ci permette di porre query più complicate tipo `length([a,b,c,[a,b]], nL).`

Per comporre:

= è un operatore di assegnamento, non composizione!

$$\begin{array}{ll} x = y & x =:= y \\ x \neq y & x \neq y \\ x > y & x > y \\ x \geq y & x \geq y \\ x < y & x < y \\ x \leq y & x \leq y \end{array}$$

Accumulatore ritorna max di una lista:

`accMax([H|T], A, Max) :- H > A, Max is A, accMax(T, H, Max).` } testiamo se la testa è il n
`accMax([H|T], A, Max) :- H <= A, accMax(T, A, Max).` } più grande trovato finora e
`accMax([], A, A).` } scorre fino alla fine
} continuano a scorrere.
 \rightarrow alla fine ottieni la clausola di stop.

- Liste, Liste, Liste :

Predicato append/3 prende due liste e le concatenano in una terza lista.

append([], L, L). $L = \emptyset + L = L$

append([H|T], L2, [H|L3]) :- append(T, L2, L3). quando concateniamo
[H|T] + L2 abbiamo una lista [H|T+L2]

?- append([a,b,c], [1,2,3], -xxx)

|

[a| -xxx]

append([b,c], [1,2,3], -xxx)

|

[b| -xxx] [a, b | -xxx]

append([c], [1,2,3], -xxx)

|

[c| -xxx] [b, c | -xxx] [a, b, c | -xxx]

append([], [1,2,3], -xxx)

|

[1,2,3] [c, 1, 2, 3]

□ [b, c, 1, 2, 3] [a, b, c, 1, 2, 3]

- Reversing a list:

Naive approach:

1. reverse empty \rightarrow get empty
2. reversing $[H|T] \rightarrow$ get reverse T and concatenate H.

naive_rev([], []).

naive_rev([H|T], R) :- naive_rev(T, RevT), append(RevT, [H], R).



Usare un accumulatore è molto meglio!

L: [a,b,c,d]

A: []

L: [b,c,d]

A: [a]

L: [c,d]

A: [b,a]

L: [d]

A: [c,b,a]

L: []

A: [d,c,b,a]

occrev([H|T], A, R) :- occrev(T, [H|A], R).
 \Rightarrow occrev([], A, A).

Le clausole ricorsive togli la testa della lista e la mette nell'accumulatore

Esercizi prolog • PDF → files / vor-exercises

• rel - genitore - figlio

```
fratello (X, Y) :-  
    genitore (Z, X),  
    genitore (Z, Y),  
    X \= Y.
```

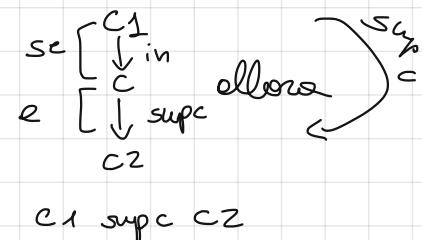
$\frac{\begin{matrix} w & z \\ \downarrow g & \downarrow \text{genitore} \\ x & y \end{matrix}}{\text{fratelli}}$ con $x \neq y$ rende cugini.

```
cugino (X, Y) :-  
    genitore (W, X),  
    genitore (Z, Y),  
    fratello (W, Z),  
    X \= Y.
```

• Superclasse

```
superclass (C1, C2) :-  
    inherit (C2, C1).  
OK
```

```
superclass (C1, C2) :-  
    inherits (C, C1),  
    superclass (C, C2).
```



chiudere transitività: posso trovare tutte le superclassi di una data classe, anche quelle sopra di più livelli gerarchici.

• genitore

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

$\rightarrow X$ è antenato di Y se esiste una persona Z tale che X è genitore di Z e Z è antenato di Y .

Abbiamo bisogno di $\text{antenato}(X, X)$ perché l'altro predicato non copre il fatto che una X è anche antenato di se stesso.

• Automi

$\text{gen}(S, []) :- \text{final}(S).$ base case \rightarrow se la lista è vuota allora lo stato corrente è lo stato finale.

$\text{gen}(S, [X|Y]) :- \text{trans}(S, X, S2), \text{gen}(S2, Y).$ \rightarrow caso in cui non è stato finale, perché la lista non è vuota \rightarrow allora trova una transizione che può portare lo stato corrente S ad uno nuovo $S2$ (usando $\text{trans}/3$) e poi chiama se stesso ricorsivamente per generare il resto delle transizioni.

`generates(L) :- initial(S0), gen(S0, L).` la lista L di transizioni può essere
generata partendo dallo stato iniziale (S0) e usando gen per il resto delle seq.