Prolog

`A Prolog program is a set of definite Horn clauses.`

1. Syntax

But, what is a clause? A clause is set of different logic terms, which typically are:

- **Variables**: strings starting with a **uppercase** letter.
- **Constants**: numbers or strings starting with a **lowercase** letter.
- **Atomic formulas**: defined as $p(t_1, t_2, \ldots, t_n)$ where p is a predicate.
- **Compound terms**: known also as **structures**, they are defined similarly to a traditional function $f(t_1, t_2, \ldots, t_n)$, where f is a **function symbol** and $t_1, t_2, \ldots, t_n$ are **terms**.

These definitions may seem difficult to understand. Let's consider a more intuitive approach with examples:

```
X, X1, Goofey, _goofey, _x, _ % variables, the underscore symbol "_" is
usually used for variables.

a, goofey, aB, 9, 135, a92 % constants.

p, p(a, f(x)), p(y), q(1) % atomic formulas.

f(a), f(g(1)), f(g(1), b(a), 27) % compound terms.
```

In addition to the key elements of **Prolog**, we have different types of clauses:

- **Fact**: `A.` represents a statement that is always true.
- **Rule**: `A :- B1, B2, ..., Bn.` meaning that A is true if and only if $B_1, B_2, \ldots, B_n$ are true.
- **Goal**: `:- B1, B2, ..., Bn.` is a question asked to the system.

```
q. % fact

p :- q, r % rule

r(z). % fact

p(x) :- q(X, g(a)) % rule
```

The comma symbol `,` represents the logical **conjuction** $\wedge$. The neck symbol `:-` defines the implication $\leftarrow$, read from right to left.

2. Declarative and Procedural Interpretations

Every Prolog program has two main interpretations:

- **Declarative interpretation**: the declarative interpretation explains **what** the program means. Variables within a clause are universally quantified. For each fact:

```
p(t1, t2, ..., tn).
```

If $X_1, X_2, \ldots, X_n$ are the variables appearing in $t_1, t_2, \ldots, t_n$ the intended meaning is: `∀X1, ∀X2, ..., ∀Xn` the fact `p(t1, t2, ..., tn)` is verified.

The meaning changes when we discuss **rules**. For each rule:

```
A :- B1, B2, ..., Bn.
```

If $Y_1, Y_2, \ldots, Y_m$ are the variables appearing **only** in the body of the rule the intended meaning is: `∀X1, ∀X2, ..., ∀Xn ((∃Y1, ∃Y2, ..., ∃Ym (B1, B2, ..., Bn))) → A))`, in other words, for each variable X_i, if there exists variable Y_j the head of the clause A is verified. Let's see an example for a better understanding.

```
happyperson(X) :- has(X, Y), car(Y)
```

For each person X, if exists a car Y (anyone) that X holds, X is a happy person.

If $X_1, X_2, \ldots, X_n$ are the variables appearing in **both** the body and the head of the rule, the intended meaning is: `∀X1, ∀X2, ..., ∀Xn ∀Y1, ∀Y2, ..., ∀Ym ((B1, B2, ..., Bn) → A)`, in other words, for each variable X_i and variable Y_j, that make the body true, the head of clause A is also verified.

```
father(X, Y). % defining the facts of the universe described.
mother(X, Y).

grandfather(X, Y) :- father(X, Z), father(Z, Y) % rules heavily
dependent on facts.
grandmother(X, Y) :- mother(X, Z), mother(Z, Y)
```

- **Procedural interpretation**: the procedural interpretation of a Prolog program explains **how** the system executes a goal, in contrast to the declarative interpretation, which only explains what the program means. A **procedure** is a set of clauses with the same predicate name in the head and the same number of parameters, also called **arity**. Prolog adopts the **SLD resolution process** and has two main charateristics:
  - It selects the **left-most** literal in any query.

```
? :- G1, G2, ..., Gn. % starting from G1 and then move on.
```

- It performs **Depth-First search (DFS)** strategy. Based on the selected search strategy, the order of the clauses in the program may greatly affect termination and, consequently, the **completeness**. DFS is a search strategy that does not guarantee completeness if the search tree contains **loopy path**.

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

?- p.
   loopy path
```

In this toy example the fact `p.` comes after the rule `p :- q, r.`. If we ask the Prolog interpreter the query `?- p.`, the program will enter a loopy path, failing to solve it. Defining the correct order, the query will be immediately solved.

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

?- p.
   yes
```

There may exist multiple answer for a query. The way to retrieve them is easy: after getting an answer, we can force the interpreter to search for the **next solution**. Pratically, this means asking the procedure (a set of clauses with the same head and arity) to explore the remaining part of the **search tree**. In Prolog, the standard way involves using the operator `;`.

```
:- sister(maria, W).
   yes W = giovanni;
   yes W = annalisa;
   no
```

The knowledge `giovanni, annalisa` comes from the previously defined facts.

Royal Family Exercise

```
female(mum).
female(anne).
female(kydd).
female(zara).
female(sarah).
female(diana).
```

```
female(sophie).
female(louise).
female(eugenie).
female(margaret).
female(beatrice).
female(elizabeth).

male(mark).
male(harry).
male(peter).
male(james).
male(george).
male(philip).
male(andrew).
male(edward).
male(spencer).
male(charles).
male(william).

married(mark, anne).
married(george, mum).
married(andrew, sarah).
married(charles, diana).
married(edward, sophie).
married(philip,elizabeth).

parent(george, margaret).
parent(george, elizabeth).

parent(mum, margaret).
parent(mum, elizabeth).

parent(elizabeth, anne).
parent(elizabeth, andrew).
parent(elizabeth, edward).
parent(elizabeth, charles).

parent(philip, anne).
parent(philip, andrew).
parent(philip, edward).
parent(philip, charles).

parent(kydd, diana).
parent(spencer, diana).

parent(diana, harry).
parent(diana, william).

parent(charles, harry).
parent(charles, william).

parent(anne, zara).
parent(anne, peter).
```

```
parent(mark, zara).
parent(mark, peter).

parent(andrew, eugenie).
parent(andrew, beatrice).

parent(sarah, eugenie).
parent(sarah, beatrice).

parent(edward, james).
parent(edward, louise).

parent(sophie, james).
parent(sophie, louise).

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

sibling(X, Y) :- parent(G, X), parent(G, Y), X \= Y.
full_sibling :- father(G1, X), father(G1, Y), mother(G2, X), mother(G2, Y),
X \= Y.

brother(X, Y) :- male(X), sibling(X, Y), X \= Y.
sister(X, Y) :- female(X), sibling(X, Y), X \= Y.

child(X, Y) :- parent(Y, X).
son(X, Y) :- parent(Y, X), male(X).
daughter(X, Y) :- parent(Y, X), female(X).

wife(X, Y) :- female(X), married(Y, X).
husband(X, Y) :- male(X), married(X, Y).

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

cousin(X, Y) :- parent(G1, X), parent(G2, Y), sibling(G1, G2), X \= Y, \+
parent(X, Y), \+ parent(Y, X).

aunt(X, Y) :- parent(G, Y), sibling(G, X), female(X).
uncle(X, Y) :- parent(G, Y), sibling(G, X), male(X).

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

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

brotherinlaw(X, Y) :- male(X), sibling(Z, Y), married(X, Z).
brotherinlaw(X, Y) :- male(X), male(Y), married(X, Z1), married(Y, Z2),
sibling(Z1, Z2).
brotherinlaw(X, Y) :- male(X), female(Y), married(X, Z1), married(Z2, Y),
sibling(Z1, Z2).

sisterinlaw(X, Y) :- female(X), sibling(Z, Y), married(Z, X).
sisterinlaw(X, Y) :- female(X), male(Y), married(Z1, X), married(Y, Z2),
```

```
sibling(Z1, Z2).
sisterinlaw(X, Y) :- female(X), female(Y), married(Z1, X), married(Z2, Y),
sibling(Z1, Z2).
```

---

3. Arithmetic and Math in Prolog

Arithmetic in Prolog is not a standard logical feature; it relies on special **built-in predicates** to force its evaluation. In Prolog, an expression like `2 + 3` is just a **term**, not the numeric value `5`. For instance, the interpreter will associate the structure `+ (2, 3)` with the fact `p(2 + 3).`.

However, the special and predefined predicate `is` forces the evaluation of any mathemical expression. Its syntax is simple: define the variable, then the predicate `is` and finally the expression to evaluate.

```
T is Expr
```

We previously mentioned that Prolog is based on the SLD resolution process, which always evaluates the left-most literal. But this is not the case with `is`. The predicate `is` forces the interpreter to evaluate the right-most literal (the mathematical expression), and the final result will be associated to the variable in the next step.

```
?- X is 2 + 3.
   yes X = 5

?- X1 is 2+3, X2 is exp(X1), X is X1 * X2.
   yes X1 = 5, X2 = 148.143, X = 742.065

?- X is Y - 1.
   no
   (or Instantion Fault, depending on the prolog system)
```

```
?- X is 2 + 5, X is 4 + 1.
   yes X = 5
```

In this example, the second goal becomes:

```
:- 5 is 4 + 1.
```

X has been instantiated by the evaluation of the first goal. As before, the **order** of the goal is very important:

```
(a) :- X is 2 + 3, Y is X + 1.
(b) :- Y is X + 1, X is 2 + 3.
```

Goal `(a)` succeeds and returns `yes X = 5, Y = 6`; goal `(b)` fails due to the incorrect order defined (the variable Y is processed before the evaluation of the mathematical expression for X).

A term representing an expression is evaluated **only** if it is the argument of the predicate `is`. For instance:

```
p(a, 2 + 3 * 5).
p(b, 2 + 3 + 5).
q(X, Y) :- p(a, Y), X is Y.

(q(X, Y) :- p(_, Y), X is Y.) % this clause will use both procedures,
achieved by the anonymus symbol.

?- q(X, Y)
   yes X = 17 Y = 2 + 3 * 5 (Y=+(2, *(3, 5)))
```

Initially, the predicate `p(a, Y)` is unified with the fact `p(a, 2 + 3 * 5).`. The association defines the atomic structure `+(2 *(3, 5))`. The second step involves the evaluation of the mathematical expression `X is 2 + 3 * 5`. (why do we define the constant `a` inside the fact **p**? As we already know, a procedure is a set of clauses with same head and arity; the constant `a` allow us to distinguish which predicate we are dealing with!)

Additionally, it's also possible to compare expressions results using the standard **relational operators**, which are: `>, <, >=, =<, ==, =/=`. The last two operators are named respectvely **arithmetically equal to** (`==`) and **arithmetically not equal to** (`=/=`). The syntax is pretty similar to the predicate `is`:

```
Expr1 REL Expr2
```

`REL` is the relational operator, `Expr1` and `Expr2` are the evaluated expressions. It's **crucial** that both expressions are **completely instantiated** before the comparison: otherwise the Prolog program will fail.

```
p(a, 2 + 3 * 5).
p(b, 2 + 3 + 5).

comparison_values(V1, V2, equal) :- V1 == V2.
comparison_values(V1, V2, first_value_greater) :- V1 > V2.
comparison_values(V1, V2, second_value_greater) :- V1 < V2.

comparison_expressions(Type1, Type2, Result):-
    p(Type1, Expr1),
    p(Type2, Expr2),
    Value1 is Expr1,
    Value2 is Expr2,
```

```
    comparison_values(Value1, Value2, Result).

?- comparison_expressions(a, b, R).
```

Now we have all the necessary ingredients to build **math functions** in Prolog. Given any function f with a certain arity n, we can implement it through a $(n+1)$ predicate. Given the function $f : x_1, x_2, \ldots, x_n \rightarrow y$, it is represented by a predicate as follows: `f(X1, X2, ..., Xn, Y)`. We must always indicate the result variable `Y` within the predicate's scope so the interpreter knows exactly what the output will be.

```
fatt(0, 1).
fatt(N, Y) :-
    N > 0,
    N1 is N - 1,
    fatt(N1, Y1),
    Y is N * Y1.
```

Before moving on, it's crucial to understand its behavior and how it truly works. The example above uses the **recursion** to solve the `factorial problem`: it begins by constructing the search tree until it reaches the leaf nodes `fact(0, 1)` and then moves towards the root node, computing the mathematical expression at each step.

4. Iteration and Recursion

In Prolog, **iteration** as in `while`, `foreach` or `repeat` **does not exist**. However, we can simulate iterative behavior through **recursion**, as already done in the `factorial example`. Prolog models iteration by defining a predicate (remember: a predicate is a set of clauses, not just one!) with two essential parts:

- **Base case**: a non-recursive clause, generally a **fact**, that defines the **termination condition** of the process.
- **Recursive case**: a rule that performs a single step of the operation and then calls itself with modified arguments, moving the process closer to the base case.

```
print(1) :- write(1), nl.
print(N) :-
    N > 1,
    write(N - 1), nl,
    N1 is N - 1,
    print(N1).
```

The `print(N)` predicate is different from the `factorial(N, Y)` predicate; where the interpreter completed the search tree **before** computing the expressions, here it immediately shows the results through the `write` predicate.

Even though any well-structured recusion works fine, a spefic type is more desirable for efficiency: **tail recursion**. A function is **tail-recursive** if the recursive call is the **most external call** in its definition. There are many cases where a non-tail recursion can be re-written as a tail recursion.

```
fatt1(N, Y):- fatt1(N, 1, 1, Y).
fatt1(N, M, ACC, ACC) :- M > N.
fatt1(N, M, ACCin, ACCout) :-
    ACCtemp is ACCin * M,
    M1 is M + 1,
    fatt1(N, M1, ACCtemp, Accout).
```

The factorial is computed using an **accumulator**. An accumulator is an extra argument passed to the predicate, which holds the running or partial result of the computation at each step. The main advantage is that the evaluation of the mathematical expression is done **before** the recursive call, avoiding **backtracking** and mantaining a constant **space complexity**.

Iteration Exercises

```
% Define a Prolog program that receives in input a number N,
% and print all the numbers between 1 and N.
printA(1) :- write(1), nl.
printA(N) :-
    N > 1,
    write(N), nl,
    N1 is N - 1,
    printA(N1).

% Define a Prolog program that receives in input a number N,
% and print all the numbers between 1 and N, from the smallest
% to the greatest.
printB(1) :- write(1), nl.
printB(N) :-
    N > 1,
    N1 is N - 1,
    printB(N1),
    write(N), nl.
% Define a Prolog program that computes the Fibonacci number.
fibonacci(0, 0).
fibonacci(1, 1).
fibonacci(N, R) :-
    N > 1,
    N1 is N - 1,
    fibonacci(N1, R1),
    N2 is N - 2,
    fibonacci(N2, R2),
    R is R1 + R2.

% Write a predicate about a number N, that is true if N is prime.
check_divisor(N, D) :-
    D >= N.
check_divisor(N, D) :-
    D < N,
    Rest is N mod D,
    Rest \= 0,
```

```
    Div is D + 1,
    check_divisor(N, Div).

is_prime(N) :-
    N > 1,
    check_divisor(N, 2).

% Write a predicate that, given a number N, prints out all the prime
numbers between 2 and N.
print_is_prime(N) :-
    is_prime(N),
    write(N), nl.
print_is_prime(N) :-
    \+is_prime(N).

print_all_primes(N, J) :-
    J > N.
print_all_primes(N, J) :-
    J =< N,
    print_is_prime(J),
    J1 is J + 1,
    print_all_primes(N, J1).

all_primes(N) :-
    N >= 2,
    print_all_primes(N, 2).
```

---

5. Lists

Lists are one of the most fundamental and widely used data structures in any programming languages. In Prolog, lists are terms built upon the special atom `[ ]`, called **empty list**, and the **constructor operator** `.`. A list is recursively defined as:

- The **empty list**, `[ ]`.
- A non-empty list consisting of a **head** and a **tail**, where the tail is itself a list, `.(T, List)`.

Standard notation	Head-Tail notation
[a]	.(a, [])
[a, b]	.(a, .(b, []))
[a, b, c]	.(a, .(b, .(c, [])))

Since the **head-tail notation** might be quite difficult to use, the term `.(T, List)` can be also represented as `[T | List]`. Once again, the **head** is `T` and the **tail** is `List`.

Standard notation	Head-Tail notation
[a]	[a | [ ]]

[a, b]	[a | [b | [ ]]]
[a, b, c]	[a | [b | [c | [ ]]]]

Even in this case, the recursive notation `[T | List]` is rather **verbose**. Therefore, we can use a more simplified syntax, such as `[a, b, c]` for the term `[a | [b | [c | [ ]]]]`.

The greatest power about lists in Prolog comes from the easy way to manipulate them using an **unification algorithm**. This provides a complete method for accessing and deconstructing list content.

```
p([1, 2, 3, 4, 5, 6, 7, 8, 9]).

:- p(X).
   yes X = [1, 2, 3, 4, 5, 6, 7, 8, 9]

:- p([X|Y]).
   yes X = 1 Y = [2, 3, 4, 5, 6, 7, 8, 9]

:- p([X,Y|Z]).
   yes X = 1 Y = 2 Z = [3, 4, 5, 6, 7, 8, 9]

:- p([_|X]).
   yes X = [2, 3, 4, 5, 6, 7, 8, 9]
```

This code snippet represents some examples about list unification processes. In particular, it's important to focus on the last predicate shown: we used the **anonymus symbol** `_` to create a new list containing all the previous values **except** the first one. The anonymus symbol allows us to "ignore" the first value of the current data structure.

List operations are inherently recursive, using the `[Head|Tail]` data structure to process one element at a time until the base case `[]` (empty list) is reached. Many list-related predicates can be written using this rule as the main principle.

1. **isList**

   `isList` checks if an argument is a list. The base case is the empty list `[]`, and the recursive part checks if the tail is also a list.

   ```
   isList([]).
   isList([X|Tail]) :- isList(Tail).

   :- isList([1, 2, 3]).
      yes

   :- isList([a|b]).
      no
   ```

2. **member**

`member` checks if an element X is in a given list. The base case is when the element coincides with the head of the list, and the recursive case checks if the element is in the tail of the list.

```
member(X, [X| ]).
member(X, [_|Tail]) :- member(X, Tail).

:- member(1, [1, 2, 3]).
   yes

:- member(4, [1, 2, 3]).
   no

:- member(X, [1, 2, 3]).
   yes X = 1;
       X = 2:
       X = 3;
   no
```

3. **length**

`length` defines the size of the list. It takes a list as first argument and the number of elements contained in the list as the second argument.

```
:- length(Tail, NT),
   N is NT + 1.

:- length([1, 2, 3], 3).
   yes

:- length([1, 2, 3], Result).
   yes Result = 3
```

4. **append**

`append` takes three lists as arguments: the first two are the actual, instantiated lists, and the third argument is the list obtained by concatenating the first two. This is a highly reversible and powerful tool.

   1. If the first list is empty, the result is the second list.
   2. If the first list has a **head**, we keep it and recursively append the rest of the first list (the **tail**) to the second list.

```
append([], L1, L1).
append([H | Rest1], L2, [H | NewTail]) :-
    append(Rest1, L2, NewTail).

:- append([1, 2], [3, 4, 5], L).
   yes
```

```
L = [1, 2, 3, 4, 5]
:- append([1, 2], L2, [1, 2, 4, 5]).
   yes

L2 = [4, 5]
:- append([1, 3], [2, 4], [1, 2, 3, 4]).
   no
```

5. **deleteFirstOccurrence**

`deleteFirstOccurrence` takes an element and a list as its first and second arguments, respectvely, and the third argument is the list without the first occurrence of the element.

```
deleteFirstOccurence(El, [], []).
deleteFirstOccurence(El, [El|T], T).
deleteFirstOccurence(El, [H|T], [H|T1]) :- deleteFirstOccurence(El, T,
T1).
```

6. **deleteAllOccurrences**

`deleteAllOccurrences` is a predicate that takes an element and a list as its first and second arguments; the third parameter is the list without all the terms that unify with the element.

```
deleteAllOccurences(El, [], []).
deleteAllOccurences(El, [El|T], Result):- deleteAllOccurrences(El, T,
Result).
deleteAllOccurences(El, [H|T], [H|T1]) :- deleteAllOccurences(El, T,
T1).
```

7. **reverse**

`reverse` takes two lists as arguments, returning a list that is the reverse of the second list given (the typical usage is `reverse(List, ReverseList)`).

```
reverse([],[]).
reverse([H|T], Result) :-
    reverse(T, Partial),
    append(Partial, [H], Result).

:- reverse([], []).
   yes

:- reverse([1, 2], Lr).
   yes Lr = [2, 1]
```

```
 :- reverse(L, [2, 1]).
    yes L = [1, 2]
```

List Exercises

```
% Write a predicate that given a list, it returns the last element.
searching_last_element([X], X).
searching_last_element([_|Tail], X) :-
    searching_last_element(Tail, X).

% Write a predicate that given two lists L1 and L2, returns true
% if and only if L1 is a sub-list of L2.
sub_list([X], L2) :-
    member(X, L2).
sub_list([Head|Tail], L2) :-
    member(Head, L2),
    sub_list(Tail, L2).

% Write a predicate that returns true if and only if a list is
% a palindrome.
check_list_palindrome([X1], [X2]) :-
    X1 is X2.
check_list_palindrome([H1|T1], [H2|T2]) :-
    H1 is H2,
    check_list_palindrome(T1, T2).

palindrome(L1) :-
    reverse(L1, L2),
    check_list_palindrome(L1, L2).

% Write a predicate that, given a list returns a new list with
% repeated elements.
repeated_elements([_], []).
repeated_elements([Head|Tail], Result) :-
    \+ member(Head, Tail),
    repeated_elements(Tail, Result).
repeated_elements([Head|Tail], Result) :-
    member(Head, Tail),
    repeated_elements(Tail, Acc),
    Result = [Head | Acc].

% Write a predicate that given a list L and a term T, counts the
% number of occurrences of T in L.
counting_occurrences(_, [], 0).
counting_occurrences(T, [X], 1) :-
    T = X.
counting_occurrences(T, [X], 0) :-
    T \= X.
counting_occurrences(T, [Head|Tail], Result) :-
    counting_occurrences(T, Tail, TailCount),
    T = Head,
```

```
    Result is TailCount + 1.
counting_occurrences(T, [Head|Tail], Result) :-
    counting_occurrences(T, Tail, TailCount),
    T \= Head,
    Result is TailCount.

% Write a predicate that, given a list, returns a new list
% obtained by flattening the first list.
flattening_list([], []).
flattening_list([Head|Tail], L) :-
    is_list(Head),
    flattening_list(Head, RestHead),
    flattening_list(Tail, RestTail),
    append(RestHead, RestTail, L).
flattening_list([Head|Tail], L) :-
    \+ is_list(Head),
    flattening_list(Tail, RestTail),
    L = [Head|RestTail].

% Write a predicate that given a list, returns a new list that
% is the first one, but ordered.
find_min([X], X).
find_min([Head|Tail], Min) :-
    find_min(Tail, TailMin),
    (Head =< TailMin -> Min = Head; Min = TailMin).

ordering_list([], []).
ordering_list(L, Result) :-
    find_min(L, Min),
    select(Min, L, Rest),
    ordering_list(Rest, RestResult),
    Result = [Min|RestResult].
```

---

6. The CUT

The **cut operator** `!` is a predefined predicate that allows interference with and control over the execution process of a Prolog program. It has no logic meaning or declarative semantic, but it heavily affects the execution process.

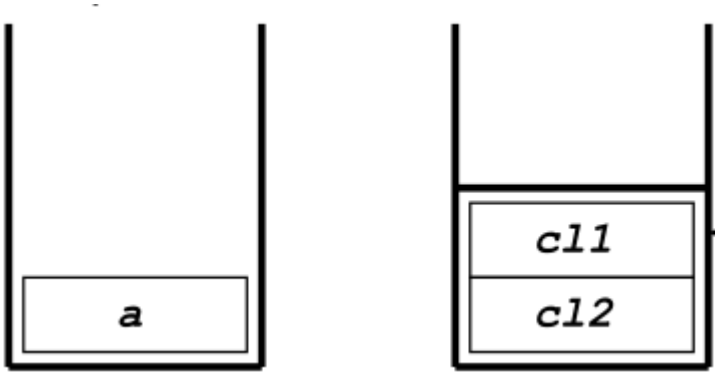
Any execution process is built upon two stacks:

- **Execution stack**. Contains the activation records of the predicates.
- **Backtracking stack**. Contains the set of open **choice points**. A choice point marks an alternate clause that can be explored if the current path fails (similar to any search strategy seen so far, like how in A*, we keep the set of node to explore in the fringe).

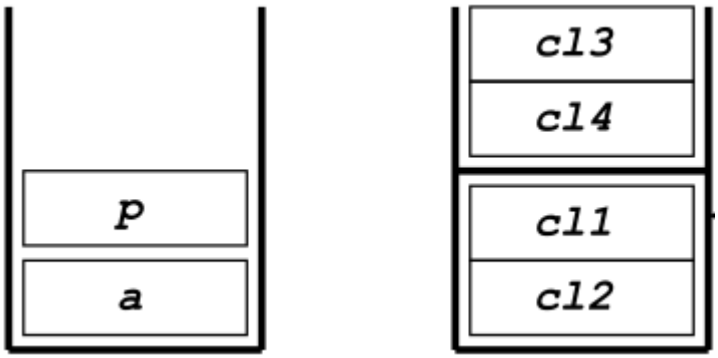
When a goal succeeds, Prolog might still have open choice points. When a goal fails, Prolog backtracks to the most recent choice point to try an alternative branch.

Example

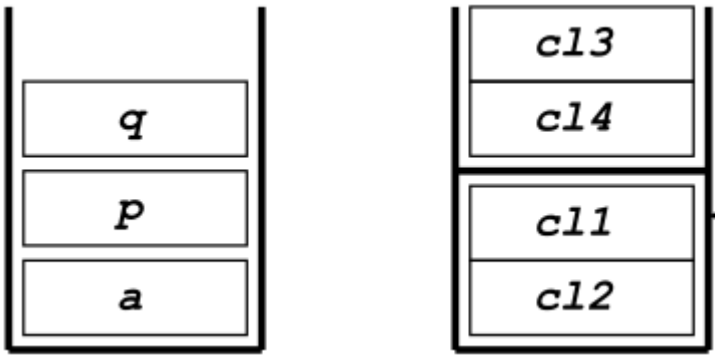
```
(cl1) a :- p, b.
(cl2) a :- r.
(cl3) p :- q.
(cl4) p :- r.
(cl5) r.
```



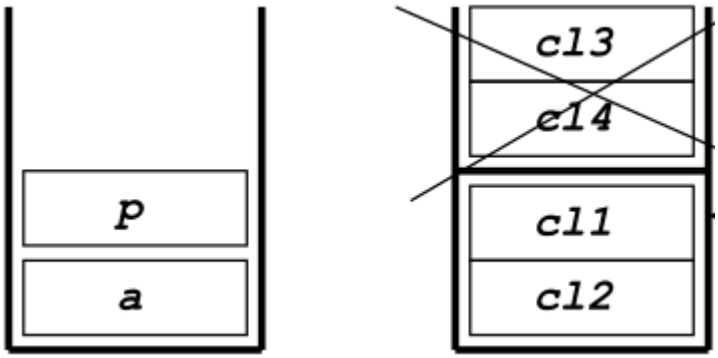
In the execution stack, we place the current goal A. In the backtracking stack, we place the references to all the clauses that match our current goal A.



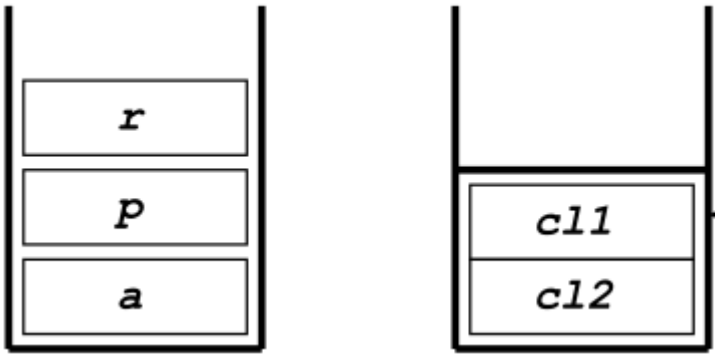
In the next step, our current goal will be P, as we can observe from the rule `a :- p, q`, and again inside the backtracking stack, we will put all the clauses that match the current goal `p`. We always choose the rule following the syntactical order, so `p :- q` comes before `p :- r`.



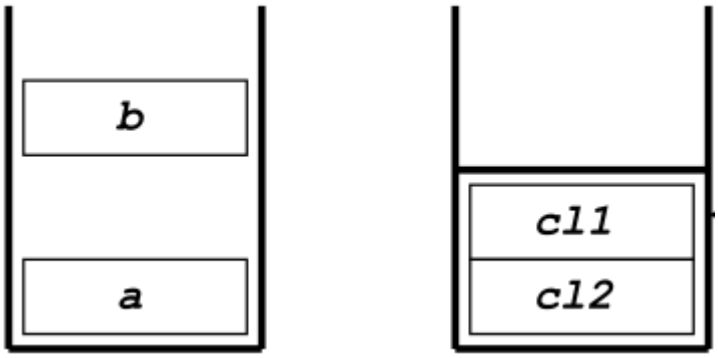
Following this behavior, our next goal will be Q. Again, we put it in the execution stack, but unfortunately, it will fail because it's not defined inside our program. So, the interpreter will select the next open choice point.



By performing backtracking, the interpreter knows there is another clause corresponding to P, so it will place the fact `r.` in the execution stack. P is true if R is true. Looking at the program, we see that R is a fact. P succeeds, so we move to the second part of the A clause, `a :- p, b.`. Finally, R and P are removed from the execution stack.



Now we must focus on B. However, B is not defined inside our Prolog program, it will certainly fail.



Since there is no unification with the B part of the clause, the resolution of the overall clause will fail.

The main idea behind the **CUT** operator is to interfere with the backtracking stack. Whenever we execute the cut, we **throw away** some alternatives that are stored in the backtracking stack. Doing this makes the execution deterministic because we choose only one branch and we stay within that single branch of the search tree.

There are advantages and disadvanteges about using the **CUT** operator:

- **Advantages**: greater **efficiency** and lower **memory** usage.

- **Disadvantages**: we lose **completeness**, as more alternatives define more possible solutions (it's not the same as Alpha-Beta pruning).

Effect of the CUT operator

Given the clause:

```
p :- q1, q2, ..., qi, !, qi+1, qi+2, qn.
```

The real meaning is that all choices made about proving the predicate `p` and literals `q1` up to `qi` are made **final**; they cannot be backtracked anymore. All alternative choices for their predicates are discarded.

If the remaining part of the clause fails, the literals after the cut operator $q_{i+1}, q_{i+2}, \ldots, q_n$, the entire rule will fail.

The real effect is that CUT operator **removes branches of the search tree**.

```
a(X,Y) :- b(X), !, c(Y).
a(0,0).
b(1).
b(2).
c(1).
c(2).

:- a(X,Y).
   yes X=1 Y=1;
       X=1 Y=2;
   no
```

As we can see from the example, the interpreter will always take the first fact of `b`, which is `b(1).`. The alternatives `b(2)` and `a(0, 0).` are simply ignored (remember: the CUT operator affects the predicates defined **before** the cut, but also **alternative clauses** for the predicate in which it appears!).

Why do we need the CUT operator? It may seem like a counterintuitive tool since it doesn't consider all possibilities and does not guarantee completeness. The real answer is that sometimes using the CUT operator allows us to **simplify** our Prolog program.

Example

```
p(X) :- q(X), !, r(X).
q(1).
q(2).
r(2).

:- p(X).
   no
```

This example fails because the only match for `q(X)` that Prolog finds first is 1. The variable X becomes 1. Since the fact is `r(2).`, the predicate `r` does not have any match for 1, so the query with $X = 1$ fails. Without the cut operator, or by defining `q(2).` before `q(1).`, the query $X = 2$ would succeed after backtracking.

The first use of the CUT operator is to achieve **mutually exclusive conditions** between two clauses. In any programming languages, we would typically write the mutual exclusive condition as:

```
if a then b else c
```

Doing that in prolog is simply:

```
p(x) :- a(X), !, b.
p(x) :- c.
```

Filtering Example

Problem: `Write a predicate that receives a list of integers, and returns a new list containing only positive numbers.`

```
filter([], []).
filter([H|T], [H|Rest]) :- H > 0, filter(T, Rest).
filter([H|T], Rest) :- H =< 0, filter(T, Rest).
```

1. If the list is empty, an empty list is returned.
2. If the first element of the list, H, is positive, a new list is returned, consisting of H and the $Rest$, where $Rest$ is obtained by the recursive call `filter` over the T part.
3. We throw away H if its value is less than or equal to 0.

However, the third clause contains a redudant condition ($H =< 0$). We already know from the second clause that if the condition $H > 0$ is not met, the remaining condition must be $H \leq 0$.

In a declarative way, we have to `pay this price` if we want mutual exclusivity without the CUT.

```
filter([], []).
filter([H|T], [H|Rest]) :- H>0, !, filter(T, Rest).
filter([_|T], Rest) :- filter(T, Rest).
```

Here, we are saying that if the condition in the second clause succeeds, the right most part of the clause will **never** be reached because of the CUT. We have to imagine that the branch of the search tree is already set up with the two predicates `filter([H|T], [H|Rest])` and `filter([_|T], Rest)`. If the first one succeeds, the second predicate is removed by the CUT operator.

If the condition $H > 0$ in the second clause fails, the CUT operator is never executed. The interpreter will start to backtrack and choose the next choice point, which is `filter([_|T], Rest)`. For doing this, we must define the recursive call in third part of the first rule, otherwise the interpreter will never choose the next choice point.

7. Negation

There is a key point in Prolog that we haven't discussed yet: Prolog is based on Horn clauses, which are a subset of First Order Logic. We don't assume the whole scope of First Order Logic, we focus on a smaller fragment of it. However, this fragment allow us to have the same expressive power, even though some things cannot be written in Prolog.

As the definition states, Prolog allows only definite clauses, which **cannot contain negative literals**. The standard SLD resolution process is designed to derive **positive** informations and is incapable to derive **negative** information. If we provide a simple knowledge base such as:

```
person(mary).
person(caterina).
person(annalisa).

dog(meave).
```

Intuitively, we can derive that `meave` is not a person. But what about Prolog? Is it capable of defining if `meave` is **NOT** a person? From the previous knowledge base, Prolog cannot derive that `meave` is not a person since it is not explicitly written inside the program

Closed World Assumption

The **Closed World Assumption** (our knowledge is closed with the respect to the World) is one of the main concept about **Database Theory** that formalizes the notion that `if it's not recorded, it's false`.

Back to the previous example:

```
person(mary).
person(caterina).
person(annalisa).

dog(meave).
```

Can we prove that `mary` is a dog? The real answer is **NO**, but using CWA, we could infer

$$\neg dog(mary).$$

In practical terms, the Closed World Assumption might be either a good or a bad way to model our world; it relies expecially about the concepts we are expressing. In Prolog the Closed World Assumption has been chosen as the primary way to treat the **knowledge base**.

CWA Example

```
capital(rome).
city(X) :- capital(X).
city(bologna).
```

From this instance, we know for sure that if X is a `capital`, it is also a `city`.

Using the Closed World Assumption, we can infer

$$\neg capital(bologna)$$

since we don't have this kind of information in our knowledge base. The interpreter will try to solve the query `:- capital(bologna)`, it will fail, allowing us to derive that `bologna` is not a capital.

By the way, the Closed World Assumption is **non-monotonic**: adding new axioms to the program might change the answer `capital(bologna)`.

Additionally, First Order Logic is **undecidable**: if a formula *A* is not a logical consequence of the program *P*, the proof procedure is **not guaranteed to terminate in a finite time**.

Example

```
city(rome) :- city(rome).
city(bologna).
```

The SLD resolution process cannot prove in a finite time that `city(rome)` predicate is not a logical consequence of our program. Since SLD uses the Depth-First strategy to explore the state space, the interpreter will end up in a **loopy path** because it tries to solve the goal using the recursive rule.

Since **CWA** sometimes fails to prove that a predicate is not a logical consequence of our program, we restrict the Closed World Assumption to those atoms whose proof terminates in a finite time.

This restriction is called **Negation as Failure**.

Negation as Failure

Prolog adopts a safer rule called **Negation as Failure (NAF)** to address the termination problem of CWA.

- **Principle**: NAF derives the negation only of atoms whose proof fails in a **finite time**.
- **Formalism**: given *FF(P)*, the set of atoms which proofs fail in a finite time, the NAF rule is: $NF(P) = \neg A | A \in FF(P)$.

Therefore, we are not supporting general negation and the Closed World Assumption; instead, we are restricting them to a further smaller set.

To handle goals containing negative literals, such as `~A`, SLD resolution is extended to **SLDNF**, the same resolution process with Negation as Failure. Let $:\text{-}L_1, \ldots, L_m$ be the goal, where $L_1, \ldots, L_m$ are literals

(atoms or negation of atoms). Since it is only an extension of the previous resolution process, an SLDNF step is defined as follows:

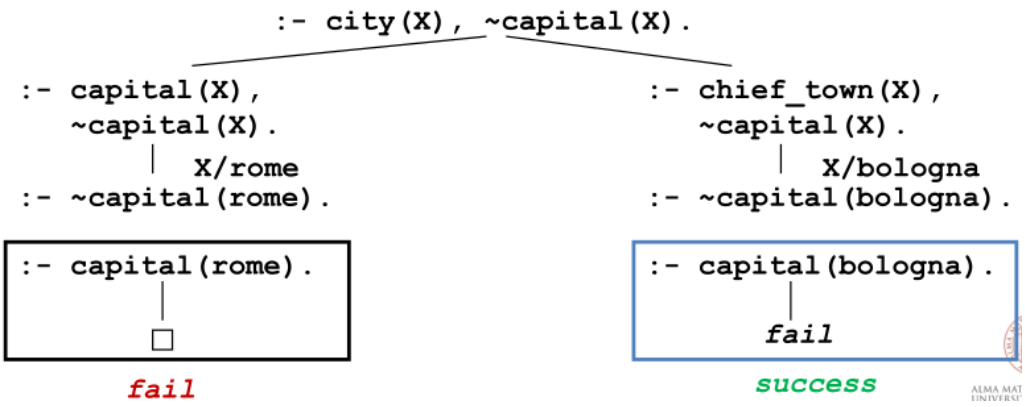
1. Do not select any negative literal L_i if it is **not ground** (meaning completely instantiated, with no variables).
2. If the selected literal L_i is positive, then apply a normal SLD resolution step.
3. If is a negative literal, `~A` with A ground, and A fails in finite time, then L_i succeeds, and the new resolvent is: $L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_m$

The selection of ground negative literals is necessary to ensure **completeness** and **correctness** of SLDNF rule.

SLDNF Example

```
capital(rome).
chief_town(bologna)
city(X):- capital(X).
city(X):- chief_town(X).

:- city(X), ~capital(X).
```



Here, we are asking if exists X such that is a `city` but not a `capital`. The SLD search tree is composed as follows:

The interpreter tries to solve the query by the first branch. The positive literal `city(X)` unifies with `city(rome)` via `capital(rome)`. X is instantiated to `rome`. The negative literal becomes $\neg capital(rome)$, which fails since `capital(rome)` is true. It keeps going by backtracking, and then it selects the second branch, the right-most one. The positive literal `city(X)` unifies with `city(bologna)` via `chief_town(bologna)`. X is instatiated to `bologna`. It tries to solve the negative literal $\neg capital(bologna)$. Since `capital(bologna)` fails in finite time, the negative literal $\neg capital(bologna)$ succeeds. We have just found an atom `bologna` that is a city but not a capital.

Prolog does not use exactly this extension to resolve negative literals; it selects always the left-most literal, **without checking if it is ground**, so if the literal is completely instatiated.

Example

```
capital(rome).
chief_town(bologna)
city(X):- capital(X).
city(X):- chief_town(X).

:- ~capital(X), city(X).
```

In the example, we have just switched the subgoals of the query. However, this change has a heavily impact on the execution. Given this knowledge base, the interpreter will always return **false**.

The main problem lies in the meaning of the quantifiers of variables appearing in negative literals. The intended meaning of $\neg capital(bologna)$ is: "does exist an X such that X is not a capital?" Given the knowledge base, it does exist an entity that is not a capital.

$$F = \exists X \neg capital(X).$$

Instead, in Prolog SLDNF, we are looking for a proof for $capital(X)$

$$:\text{-}capital(X)$$

with explicit quantifiers, it becomes:

$$F = \exists X capital(X).$$

Then, the result is negated:

$$F = \neg(\exists X capital(X))$$

and according to syntactic rules, the previous definition is equal to:

$$F = \forall X(\neg capital(X)).$$

In summary, if there is a X that is a capital, the proof F will always fail. The failure is due to the unsafe order; if the subgoals were switched, the query would succeed. Another final peculiarity is that Prolog tries to solve the query even though the negative literal is **not ground**, violating the **safe rule** of SLDNF resolution process.

Meta-predicates in Prolog

`Equivalance of programs and data`.

1. Introduction

There is one thing that make Prolog a really powerful programming language: **predicates** and **terms** (also viewed as programs and data) share the same **syntactic structure**. As a consequence of this equivalence, programs and data can be easily interchanged. But what does it really mean?

Prolog provides several predefined predicates, known as **meta-predicates**, to deal with these structures and work with them.

2. Call Predicate

The `call` predicate is a fundamental meta-predicate that allows the execution of a term as if it were a goal. The key thing to observe is that with the `call` predicate, we can dynamically test a program inside the same program!

Functionality

1. If a term T is meant to be a predicate, we can prepare and create that term in a proper way and then **execute it**.
2. **call(T)**: the term T is treated as a goal, and the Prolog interpreter is requested to evaluate it.
3. The term T must be a **non-numeric term** at the moment of the evaluation, since a number cannot be evaluated in a logical sense.

Why Call is a Meta-Predicate

The predicate `call` is considered to be a meta-predicate because:

1. Its evaluation interfere with the Prolog interpreter, as it stops the evaluation of the current **goal** in order to evaluate the subject of the `call` predicate.

2. It directly alterates the program's execution flow.

   Example

   ```
   p(a).
   q(X) :- p(X).

   :- call(q(Y)).
      yes Y = a.
   ```

   Asking for `q(y)`, it is unified with the clause `q(x)`. Our goal becomes `p(y)`, and `p(y)` is true if Y is unified with constant `a`. The final answer is: `q(y)` is verified if $Y = a$.

This code snippet demonstrates the previous definition: we can pass to the `call` meta-predicate pieces of a program instead of just data.

The predicate `call` can also be used to call other predicates.

```
p(X):- call(X).
q(a).

:- p(q(Y)).
   yes Y = a.
```

Example `if_then_else` Construct

`call` is crucial for implementing control flow structures, such as an `if_then_else` construct. The goal is to define a `if_then_else(Cond, Goal1, Goal2)` clause such that: if `Cond` is true, execute `Goal1`; otherwise, execute `Goal2`.

```
if_then_else(Cond,Goal1,Goal2):-
    call(Cond), !,
    call(Goal1).
if_then_else(Cond,Goal1,Goal2):-
    call(Goal2).
```

The program `if_then_else` takes in input **three different programs**: `Cond`, `Goal1` and `Goal2`. (Note: in this case, the **CUT operator** is essential to prevent the interpreter from backtracking the second clause if `Cond` succeeds).

3. Fail predicate

The `fail` predicate is a simple, arity-zero predicate used primarly to **explicitly control backtracking**.

Functionality

1. `fail` takes **no argument**.
2. Its evaluation **always fails**.
3. This failure forces the interpreter to **explore other alternatives** (in other words, it activates the backtracking).

Applications

Forcing a proof to fail might seem counter-intuitive, but it serves three main purposes:

1. To obtain some form of **iteration** over data.
2. To implement the **Negation as Failure** mechanism.
3. To implement a form of **logical implication**.

Iteration Example

Let us consider a **knowledge base** with facts `p(X).` and suppose we want to apply a predicate `q(X)` on all X that satisfy a fact `p(X)`.

```
iterate :-
    call(p(X)),
    verify(q(X)),
    fail.

iterate.

verify(q(X)) :- call(q(X)), !.
```

- The first clause of `iterate` finds a solution for `p(X)`, executes `q(X)`, and then fails, triggering the search for the next solution for `p(X)`.
- This process continues until `call(p(X))` eventually fails, at which point Prolog moves to the second clause, `iterate.`, which succeeds, stopping the overall goal.

Negation as Failure Example

Defining a predicate `not(P)` which is true if `P` is not a **logical consequence** of the program.

```
not(P) :-
    call(P),
    !,
    fail.

not(P).
```

- If `call(p)` succeeds, the cut `!` prevents backtracking, and `fail` ensures `not(P)` fails.
- If `call(p)` fails, the first clause is skipped, and the second clause `not(P).` succeeds.

(Note: in this examplem, there are two items that don't have any **declarative meaning**: the predicate `call` and the CUT operator `!`).

Combining fail and CUT Example

The sequence `!, fail` is often used to force a **global failure** of a predicate, stopping not only backtracking within the predicate but also preventing all the other possible alternatives for it.

Define the **fly property**, that is true for all the birds except penguins and ostriches.

```
fly(X) :-
    penguin(X),
    !,
    fail.

fly(X) :-
```

```
    ostrich(X),
    !,
    fail.

fly(X) :-
    bird(X).
```

If X is a penguin, the first clause succeeds up to the `fail.`. The `!` prevents Prolog from trying the next `fly(X)` clauses, forcing the global failure for that specific X. Although this program is a good starting point for handling exceptions, it grows linearly according to the number of exceptions handled.

4. Setof and Bagof Predicates

These meta-predicates address **second-order queries**, which ask for the collection of elements that satisfy a goal, rather than just a single solution.

Existentially Quantification

In Prolog, the usual query `:- p(X).` returns a **possible substitution** for variables of `p` that satisfies the query, implying that X is existentially quantified (is there an X such that `p(x)` is true?).

Sometimes, it can be helpful to set up a query that asks: *which is the set S of element X such that `p(X)` is true?* This type of query is named **second-order query**.

`setof(X, P, S)`

- **Functionality**. S is the **set** of instances X that satisfy the goal P.
- **Property**. It generally returns a set **without repetitions** and the elements are typically sorted.
- **Failure**. If no X satisfied P, the predicate **fails**.

`bagof(X, P, L)`

- **Functionality**. L is the **set** of instances X that satisfy the goal P.
- **Property**. It returns a list that may contain **repetitions**.
- **Failure**. If no X satisfies P, the predicate **fails**.

Examples

Given the knowledge base:

```
p(1).
p(2).
p(0).
p(1).
q(2).
r(7).
```

The results of the following second-order queries are:

```
:- setof(X, p(X), S).
   yes S = [0, 1, 2]
   X = X

:- bagof(X, p(X), L).
   yes L = [1, 2, 0, 1]
   X = X
```

As we can see, the set S from `setof(X, P, S)` does not include repetitions, while `bagof(X, P, L)` returns a list that does.

Furthermore, these meta-predicates allow for the **conjuction** of goals within their own scope.

```
:- setof(X, (p(X), q(X)), S).
   yes S = [2]
   X = X

:- bagof(X, (p(X), q(X)), L).
   yes L = [2]
   X = X

:- setof(X, (p(X), r(X)), S).
   no

:- bagof(X, (p(X), r(X)), L).
   no
```

The last two queries tell us that the knowledge base does not have any X that satisfies the conjuction of goals `p(x), r(x)`.

```
:- setof(p(X), p(X), S).
   yes S = [p(0), p(1), p(2)]
   X = X

:- bagof(p(X), p(X), S).
   yes S = [p(1), p(2), p(0), p(1)]
   X = X
```

These meta-predicates can also retrieve the **terms** that make our goal true. For instance, the query `:- setof(p(x), p(x), S)` returns the set of terms `p(X)` that makes the goal `p(X)` verified.

Example

Given the knowledge base:

```
father(mario, aldo).
father(mario, paola).
father(giovanni, mario).
father(giuseppe, maria).
father(giovanni, giuseppe).
```

We want to derive which individuals are fathers.

```
:- setof(X, Y^father(X, Y), S).
   yes [giovanni, mario, giuseppe]
   X = X
   Y = Y
```

The goal part uses a new **syntactic rule**, the **existential quantifier** `Y^`. This allows us to retrieve the set of X values such that there **exists** a Y that satisfies the goal `father(X, Y)`. If the existential quantifier is not used, the final result will be diplayed as multiple solutions, one for each unique (X, Y) pair that makes the goal `father(X, Y)` true.

```
:- setof((X, Y), father(X, Y), S).
   yes S = [(giovanni, mario), (giovanni, giuseppe),
          (mario, paola), (mario, aldo),
          (giuseppe, maria)]
   X = X
   Y = Y
```

The final code snippet describes all the **tuples** retrieved by the knowledge base that make the goal `father(X, Y)` true.

5. Findall predicate

The `findall` meta-predicate returns the list S of instances X for which predicate P is true. If there is no X satisfying P, the meta-predicate returns an empty list.

Its behavior is essentially equal to the **existential quantifier**, it searches for the list S of instances X such that there exists Y that satisfies the predicate P.

Example

Given the knowledge base:

```
father(mario,aldo).
father(mario, paola).
father(giovanni,mario).
father(giuseppe,maria).
father(giovanni,giuseppe).
```

We want to define which individuals are father.

```
:- findall(X, father(X, Y), S)
   yes S = [mario, giovanni, giuseppe]
   X = X
   Y = Y
```

This code snippet is the same as:

```
:- setof(X, Y^father(X, Y), S)
   yes S = [mario, giovanni, giuseppe]
   X = X
   Y = Y
```

The meta-predicates `setof`, `bagof` and `findall` works also when the property to be verified is not a simple fact, but it is defined by rules.

Example

Given the knowledge base.

```
p(X, Y) :- q(X), r(X).

q(0).
q(1).
r(0).
r(2).
```

The `findall` predicate returns the set of instances X that satisfy the rule `p(X, Y) :- q(X), r(X)`.

```
:- findall(X, p(X, Y), S)
   yes S = [0]
   X = X
   Y = Y
```

6. Implication through setof

Let's suppose we have a knowledge base containing facts about `father(X, Y)` and `employee(Y)`. We want to verify if it is true that for every Y for which `father(X, Y)` holds, then Y is an *employee* (so, basically we are asking if exists Y such that X is the father of Y and it is an *employee*).

In logical terms, this query can be seen as a simple implication.

$$father(X, Y) \rightarrow employee(Y)$$

```
imply(Y) :- findall(Y, father(X, Y), S), verify(S).

verify([]).
verify([H|T]) :- employee(H), verify(T).
```

First of all, `findall(Y, father(X, Y), S)` returns a list S containing all the sons already present in the knowledge base (remember: the left-most part of the clause is always evaluated first by the Prolog interpreter). The same list S is used to verify if all the instances Y are *employee*. If only one of them is not an *employee*, the whole clause `imply(Y)` fails.

7. Iteration through setof

Given a Prolog program, we can execute a procedure `q` on each element for which `p` is true.

```
iterate :- setof(X, p(X), L), filter(L).

filter([]).
filter([H|T]):- call(q(H)), filter(T).
```

8. Clause predicate

As we already know, in Prolog terms and predicates share the same structure, therefore we can interchange them without any problem.

Given the knowledge base.

```
h.
h :- b1, b2, ..., bn.
```

They correspond to the terms.

```
(h, true)
(h, ','(b1, ','(b2, ','( ...','(bn - 1, bn) ...))))
```

Functionality

1. It takes two arguments: `Head`, representing the head of the clause, and `Body`, representing the body of the clause.
2. `Head` must be **bound** to a non-numeric term when the predicate is evaluated.
3. `Body` can be a variable or a term describing the body of the clause. If the body of the clause is a **fact**, the `Body` term is unified with **true**.

4. Its evaluation return **true** if `(Head, Body)` is unified with a clause stored within the database program.
5. Its evaluation open more **choice points**, if more clauses with the same head are available.

Given the knowledge base.

```
p(1).
q(X, a) :- p(X), r(a).
q(2, Y) :- d(Y).
```

```
?- clause(p(1), BODY).
   yes BODY = true

?- clause(p(X), true).
   yes X = 1

?- clause(q(X, Y), BODY).
   yes X = 1 Y = a BODY = p(_1), r(a);
   yes X = 2 Y = _2 BODY = d(_2);
   no

?- clause(HEAD, true).
   Error - invalid key to data-base
```

Let's take a look at the third query, `?- clause(q(X, Y), BODY)`. The first result shows that there exists a clause `q(X, Y)`, in which X is unified with the value 1 and Y with the constant a. After the **disjunction** `;` (used for checking if there are other solutions), we get `X = 2` and `Y = _2`; this last unification expresses that Y remains a **free variable**.

Meta-interpreters in Prolog

`A program fully developed in Prolog`.

1. Introduction

The main difference between **meta-predicates** and **meta-interpreters** is: **meta-predicates** are built-in Prolog tools that influence the standard execution of the program. In contrast, **meta-interpreters** are programs, designed and tested by a developer, that take another program in input and define their own interpretation rules.

Simply, we can develop a new interpreter that follows our **interpretation rules**.

The meta-interpreters use two main meta-predicates:

1. **clause(Head, Body)**: used to detect if it exists a given clause inside the database program.

```
p(1).
q(x, a) :- p(x), r(a).

?- clause(q(x, y) BODY)
   yes X = 1 Y = a BODY = p(1), r(a);
   no
```

2. **call(T)**: evaluation of the subject *T* of the **call** meta-predicate.

2. Meta-interpreter

In `Prolog`, as we already know, there is no **difference** between programs and data. From this first assumption and by the set of **built-in predicates**, we could design any kind of **meta-interpreter**.

A meta-interpreter is a specific type of meta-program, that takes in input a program and returns another program. In other words, we can define any given meta-interpreter as: `a program written in a specific language L that is able to execute other programs written in that language L`.

In `Prolog`, we shall say that a meta-interpreter for a language **L** is defined as an interpreter for **L**, but fully written in **Prolog**.

Given these premises, would it be possible to make a **Prolog machine** written in Prolog for interpreting another Prolog program? This query is answered by the **vanilla meta-interpreter**.

3. Vanilla meta-interpreter

The **vanilla meta-interpreter** is a Prolog machine designed in Prolog for executing Prolog programs.

Assuming a predicate `solve(Goal)` (outside of the set of built-in predicates) that answers true if the variable `Goal` can be proved using the clauses of the current program taken in account, a possible vanilla

meta-interpreter would be:

```
solve(true) :- !.
solve((A, B)) :- !, solve(A), solve(B).
solve(A) :- clause(A, B), solve(B).
```

This snippet of code has three main passages:

- 1st step. If we pass a **true fact** to the procedure, then all the other possibilities are thrown away from the search tree and the computation stops.
- 2nd step. `(A, B)` is the **conjuction** of different literals. The goal of the predicate is to prove the truth of the conjuction. This clause answers to the question: *which is the resolution order of the conjuction?*. By this example, we have choosen the `left-most` rule.
- 3rd step. Given a rule `A`, the predicate will check if, inside the database program, `A` is already defined as a rule with a body `B`, and then we solve `B` in order to find the truth of the rule.

The last instance provides us the real meaning of the vanilla meta-interpreter: given a `goal`, it will use our Prolog program to prove that `goal`.

But, what is the main advantage of using a `home-made` interpreter? We have already a Prolog interpreter that does the same and also better than us. Vanilla interpreter mimics the same behavior of the Prolog interpreter, with a main difference: vanilla is developed in Prolog, indeed the standard one is written in C, so its compilation time is very fast.

However, we can use the vanilla meta-interpreter **to change** the general behavior of Prolog program language. Let's see some snippets for better understanding.

Right-most rule example

Define a Prolog interpreter that adopts the rule `right-most`.

From the previous snippet, we can achieve the `right-most` rule by changing the order of the resolution process.

```
solve(true) :- !.
solve((A, B)) :- !, solve(B), solve(A).
solve(A) :- clause(A, B), solve(B).
```

Note: the predicate `solve(...)` does not belong to the set of built-in ones.

Number of steps example

Define a prolog interpreter `solve(Goal, Step)` that:

- It is true if Goal can be proved.
- In case Goal is proved, Step is the number of resolution steps used to prove the Goal.

```
solve(true, 0) :- !.
solve((A, B), S) :- !, solve(A, SA), solve(B, SB), S is SA + SB.
solve(A, S) :- clause(A, B), solve(B, SB), S is 1 + SB.
```

As before, we focus on the main passages:

- 1st step. Solving a **true fact** requires 0 steps.
- 2nd step. The total number of steps to solve the conjuction `(A, B)` is equal to the sum of the steps for solving `A` and `B`.
- 3rd step. The amount of steps necessary to solve the rule `A` is equal to the steps to solve the **body** of the rule.

```
a :- b, c.
b :- d.
c.
d.

?- solve(a, Step)
   yes Step = 4
```

For solving the rule `a` are necessary 4 steps. The number is simply computed by all the unifications done: `a :- b, c.` → `a :- d, c.` → `a :- true, c.` → `a :- true, true.`.

4. Expert System

An **expert system** is a construction over the knowledge base that allows us to make queries about the same knowledge base.

Let's suppose we have a knowledge base, and we want to query it looking to prove something.

```
good_pet(X) :- bird(X), small(X).
good_pet(X) :- cuddly(X), yellow(X).
bird(X) :- has_feathers(X), tweets(X).
yellow(tweety).
```

We want to know if `tweety` is a good pet:

```
?- good_pet(tweety).
ERROR. Undefined procedure: has_feathers/1
```

`Does it mean that we do not know if tweety has feathers?`
No, since Prolog assumes the **Close World Assumption** (every unstaged information is considered false), `tweety` does not have feathers. However, when we look to the knowledge base we are more to the side of

the **Open World Assumption**: the informations not stated are not false but simply unknown. We have to define a way to retrieve them.

The basic idea is to extend our **knowledge base**. If we try to prove a `Goal` using the given knowldge base and we fail, then we could ask more informations to the user. By the way, this approach has two main problems:

- Following this method, our Prolog program will end up asking everything to the user.
- At every new start we lose all the informations already asked to the user.

We need a method that allows us to store the retrieved informations inside a permanent storage.

5. Modifying dynamically the program

Any time a Prolog program is consulted, its representation in terms of data structure is loaded into a database. This database is the **program database** loaded in memory, handled by the common DBMS (DataBase Management System) techniques.

Note: a Prolog program is a set of **Horn clauses** and they are splitted by the domains of the database, in order to store them as a table definition.

We can use this representation to update the knowledge base. The main methods to update the knowledge base are:

- 1st `assert(T)`
  The clause `T` is added to the database program in a non-specified position of the data structure.
- 2nd `asserta(T)`
  The clause `T` is added at the beginning of the database.
- 3rd `assertz(T)`
  The clause `T` is added at the end of the database.
- 4th `retract(T)`
  Removes the first clause that matches `T`.
- 5th `abolish(T)`/`retract_all(T)`
  Removes all the occurencies of that clause from the database.

All the clauses `T` must be already instantiated before the executions of the described predicates. We have to pay attention when we use these methods, they can change the **declarative semantic** of the Prolog program.

Upper Ontologies

`A limited number of very high-level concepts that appear constantly`.

1. Introduction

There are many different types of knowledge. However, in almost of the cases as humans being we tend to characterize any given knowledge by two **mental constructions**:

- Categories.
- Objects.

Therefore, we can assume that an object belongs to a category. We will use them to express any kind of information.

2. Upper Ontologies

`Definition`
An ontology is a formal, explicit description of a domain of interest.

A simpler definition expresses an ontology as: an attempt to write down our mental structure when we deal within a specific knowledge. It has four main peculiarities:

- 1st **formal**. An ontology should be written using a language with a clear and non-ambiguous semantic. A correct ontology should not leave space to ambiguity; in this case, we cannot use **Natural Language** since that it's the primary cause of ambiguity.
- 2nd **explicit**. The information should be available or, in the worst case, derivable in a finite time.
- 3rd **description**. It should provide us interesting informations.
- 4th **domain**. Such description, it should be related to some topic of interest.

As humans being, we always make use of ontologies to describe our mental structure about a specific domain or topic of interest; at the same time, we tend to organize our own mental structures through two notions:

- **Categories**.
- **Objects**.

Example

Let's us suppose the following problem:
`the new university director wants to organize all the employees, with the only purpose of the salary management`

From the description of the problem we can already determine which is the domain of the ontology: **the management of the salaries**.

In addition, we could imagine that the university employees belong to:

- Administratives group.

- Technicians group.
- Researches group.

Given our goal (salary management), we have to define which are the properties that have a meaning in the ontology representation. For example, since we have the notion of *employee* a more general category of it should be `Person`. However, this new possible category is out of the scope, we should not care about the `Person` entity.

Categories can be more or less general (person vs. researcher) and, often, these categories can be organized following a hierchical order, like the subclass and superclass interpretation in Object-Oriented programming. Generally speaking, this distinction between categories describes also the **depth** of the ontology used. If we are handling within the **more general categories** usually we say that we are dealing with **upper ontologies**.

`Definition`
An **upper ontology** is an ontology that focuses on the more general domain.

At least an upper ontology has two characteristics:

- It should be applicable to any specific domain; since that it's the more general one, if we cannot derive this behavior the ontology used is not an upper ontology.
- We should be able to combine different general concepts without incurring in inconsistencies.

3. Categories

It is commonly accepted that humans represent the knowledge in terms of categories and objects belonging to. Part of the mental structure relies on categories and the same time the goals of the knowledge can be either **instance-specific** or **category-driven**.

But, why do we need categories? One possible answer would be: categories allow us to predict how the future will evolve or what will happen in the future. Therefore, given the categories, we may recognize new objects classifying them by the known categories. The act of classify a new object is a **prediction**.

In First Order Logic, we have two possible alternatives to make a prediction:

- 1st representing **categories as predicates**. Following this approach we end up in the Second Order Logic, not practicable.
- 2nd through **reification**, representing **categories as objects**. This is possible by the notion of `membership` and `subclass`.

By the way, there's one main problem: what is it the right language to represent categories, instances and their relationships? This query defines one of the most difficult task because, in the computational field, we have adopted the most simple approach: determine categories and their entities by the mathematical notion of set.

Therefore, the notion of `membership` is defined by the $\in$ symbol used through sets and, indeed, the notion of `subclass` is described by the $\subset$ symbol.

Example

Given the object `aa123bb`, we know that `aa123bb` belongs to the category `Car`:

$$aa123bb \in Car.$$

Any member of the category `Car` is also a member of the category `Vehicle`:

$$Car \subset Vehicle.$$

Usually, each member of a category enjoys some **properties** (called **necessity properties**):

$$(x \in Car) \rightarrow hasWheels(x).$$

Members of a category can be recognized by some properties (named **sufficiency properties**):

$$hasPlate(x) \land hasWheels(x) \rightarrow x \in Car.$$

Additionally, categories can be seen as properties of other categories:

$$Car \in VehicleType.$$

Categories relate each other by the subclass relation, but in some cases given a set of categories S, the categories theirselves can be disjointed. Formally, the disjoint relation is described as follows:

$$Disjoint(S) \Leftrightarrow (\forall c_1, c_2 \in S \land c_1 \neq c_2 \rightarrow c_1 \cap c_2 = \emptyset).$$

So, basically, we are saying that the two categories c_1 and c_2 do not share the same objects.

Subcategories might also cover all the possible categories of the parent category. Formally, given a category c and a set S of categories, S is an **exhaustive decomposition** (meaning that we can cover the general category c by all its subcategories in S) of c if:

$$ExhaustiveDecomposition(S, c) \rightarrow (\forall i \in c \Leftrightarrow \exists c_2, c_2 \in S \land i \in c_2).$$

If a category can be decomposed in more categories and each of them is disjointed from the others, then we have a **partition**. Given a category c, a partition of the category c occurs when:

- We have **disjointness** between subcategories of c.
- We can describe an **exhaustive decomposition** of c.

Formally, given a category c and a set S of subcategories, S is a **partition** of c if:

$$Partition(S, c) \Leftrightarrow Disjoint(S) \land ExhaustiveDecompotion(S, c).$$

4. Physical Composition

Physical composition is another discipline based around the study of when something is a part of a more large entity. This discipline is called **meriology**, that studies the relation between object and its parts (note: we are assuming categories like object representations).

Example

- A shopping bag contains three pieces of bread, ...

- A car is composed of an engine, four wheels, five seats, ...
- A laptop is made of a motherboard, a cpu, a keyboard, a screen, ...

Sometimes between the parts of an object could be a **structural relation**. This feature may help us to distinguish objects in the physical composition discipline:

- If it exists, the relation is named **PartOf**. The PartOf relation enjoys some properties, as:
  - 1st **Transitivity**. $PartOf(x, y) \land PartOf(y, z) \rightarrow PartOf(x, z)$.
  - 2nd **Reflexivity**. $PartOf(x, x)$.
- If it does not exist, the relation is named **BunchOf**. The BunchOf relation aims to define objects in terms of composition of other **countable** objects.

Frames and Semantic Network

`One way to represent and reason about categories and objects`.

1. Introduction

Done the **upper ontologies**, during the 60s and 70s, reasearchers start to reason about the way to represent all this stuff. As we already knew, **categories** are a powerful mechanism for describing the knowledge representation.

The basic idea is to define categories in a **computable form**. First Order Logic is a great tool for describing knowledge, however, sometimes, it is not the easiest way for this goal. Historically, the approach that has been decided in knowledge representation field is the **semantic network**.

2. Semantic Network

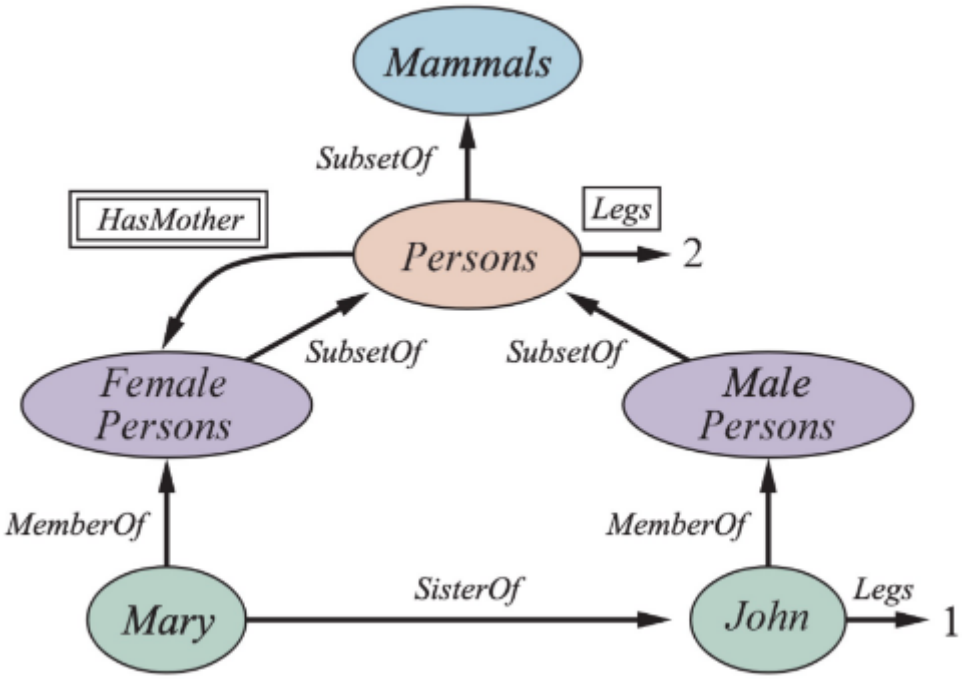
Semantic networks provide a graphical way for visualizing a knowledge base. Generally, any network, even the easiest one, has two main representations:

- **Boxes**, used to determine either objects and categories.
- **Links**, used to describe the relationships between boxes.

In particular, at the beginning, there was some misunderstandings about the **nature** of the links, in other words, about their real meaning.

Example

Given the following example, we can observe some peculiarities.



Boxes are categories and objects; in this case, we distinguish:

- `Mammals`, `Persons`, `FemalePersons` and `MalePersons` as **categories**.
- `Mary` and `John` as **objects**.

Before moving on, the figure shows us several meanings of the relationships, as:

- 1st *MemberOf* is a membership link, `Mary` is an entity of `FemalePersons` category.
- 2nd every female person is an entity of the `Persons` category; we derive that `FemalePersons` is a subclass of `Persons`.
- 3rd the property *SisterOf* between `Mary` and `John` is a shortcut to describe a more complex logical axiom: `Mary` and `John` are siblings if and only if they share at least one parent.
- 4th `Legs = 1` is a property related only to `John`. Deafault properties can be assigned directly to the objects, like an overriding method.
- 5th every entity of `Persons` category has a mother that is a female person. Formally, the relationship would be:

$$\forall x, x \in Persons \rightarrow (\forall y \, \text{HasMother}(x, y) \rightarrow y \in FemalePersons)$$

Semantic networks are a very easy way to describe our knowledge; however, we are dealing within a syntax that any time has a different meaning: from the previous example, we have four different types of links and each of them has the same representation. This ambiguity could cause several problems, we must define a more formal way.

What it's nice about these network representations is that we can easily reason about **inheritance** and its properties.

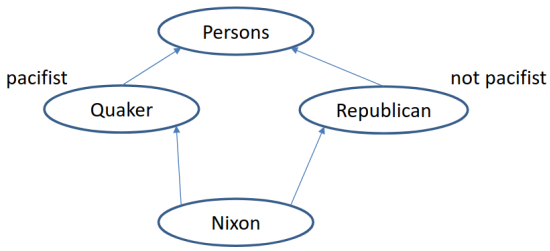
Example

For instance, given this query:

```
how many legs does Mary have?
```

we can derive this property simply go backword through the network, until we arrive to `Persons` category, where we learn that each instance of the category has 2 `Legs`. Therefore, we can derive some notions even though their are not directly attached to the box object.

By the way, things seem to be more difficult when we have to deal with **multiple inheritance**.



This chart is called the *diamond problem*: `Nixon` is an instance of two categories at the same time. Belonging to two subclasses at the same time could cause inconsistencies between properties of the instance described.

How do we have to interpret the `Nixon` object? Is it a pacifist or a not-pacifist? Can it be both? We have a sort of incoherence between properties, and from an inconsistent set of logical axioms we can retrieve any possible truth.

Many semantic networks systems allow to attach a **call** to a special procedure; any category or object could have a *piece of program* to execute.

3. Frames

The faculty to attach a call to a special procedure, by a category or an object, bring us to the notion of **frame**. A frame is another way to present a knowledge base; we can divided a network by a set of single frames.

`Definition`
A frame is a piece of knowledge that describes an object in terms of its properties.

The frame approach can be viewed as an attempt to integrate together declarative and procedural knowledge.

Every frame has:

- Unique name.
- Special representation of properties by couples of *slot/filler*.

Example

```
(tripLeg123
    <:Instance-of TripLeg>
    <:Destination toronto
    ...
)

(toronto
    <:Instance-of City>
    <:Province ontario>
    <:Population 4.5M>
    ...
)
```

`toronto` is an entity of the `City` category: `Province` and `Population` are the **slots** of the properties representation and `ontario`, `4.5M` are the **fillers**.

Slots can contain additional information about the fillers, named **facets**, such as:

- Default value.
- Allowed range.
- Type.

More interestingly, fillers can take the form of procedural attachments. Therefore, fillers can be **calls** of pieces of code. They are divided into:

- `if-added call`, used for adding value to a slot.
- `if-removed call`, used for removing value from a slot.
- `if-needed call`, used for looking for the value of a slot.

Example

```
(toronto
    <:Instance-of City>
    <:Province ontario>
    <:Population [IF-NEEDED QueryPopulationDB]>
    ...
)
```

Description Logic

`Another way to represent categories and objects`.

1. Introduction

Summing up, we represent, as humans being, a lot of informations by categories and objects. From the logic point of view, there exist several tools that allow us to deal with this kind of representations.

Semantic networks and **frames** were attempts to handle these mental constructions. However, we end up saying that we need a more **formal language** to describe properly the relationships among categories and objects.

We would like to:

- 1^{st} Represent complex concepts as the result of some **composition** of simpler concepts.
- 2^{nd} Know if an individual/object **belongs** to a category or not.

This is possible by **Description Logics**, designed to make easy to describe definitions and properties of categories, following a formalization of the networks mean.

2. A simple logic: *DL*

The most simple Description Logic is *DL* (Description Logics is a set of logics). *DL* is based on two different sets of symbols:

- Logical symbols, they have a fixed meaning.

  `Definition`
  Logical symbols is a set of:

  1. Punctuation (,), [,].
  2. Positive integers.
  3. Concept-forming operators **ALL**, **EXISTS**, **FILLS**, **AND**.
  4. Connectives $\sqsubseteq$, $\doteq$, $\rightarrow$.

  Each *concept-forming operator* has its own meaning and their semantic is influenced by the properties of the instances of a given category.

- Non-logical symbols, their meanings are domain-dependent.

  `Definition`
  Non-logical symbols is a set of:

  1. Atomic concepts.
  2. Constants.
  3. Roles.

> **Complex concepts** can be created by combining atomic concepts together, using the **concept-forming operators**.

In many research area there is a distinction between:

- **A-Box**, used to describe facts about an entity.
- **T-Box**, used to describe properties of a whole category.

Example

Given a category `Person` and an instance of it `Matteo`, we distinguish A-Box from T-Box in this way:

- `Student(Matteo)` is an Assertion-Box, related only to the individual `Matteo`.
- $\forall x \in \text{Person}, \text{Mortal}(x)$ is a Terminological-Box, related to all the instances of the category `Person`.

As we already said, our goals are: first of all, represent complex concepts and recognize if an object belongs to a category. This is reachable by the usage of *concept-forming operators*.

By these tools, we can formalize more complex concepts using several instances coming from different categories and, at the same time, we could describe the properties of instances already in the category to recognize if an unknown object belongs to it or not (instead of classify it retrieving its properties and compare them within the properties of the possible category).

- 1st **[ALL r d]**.

  `It stands for those individuals that are r-related only to inviduals of class d`. In other words, *[ALL r d]* is a new category composed by those individuals that respect the *role r* and they hold the *domain d*.

  #### Example

  1. **[ALL :HasChild Male]** → Defining a new category made of individuals that have a zero or more children, but all males.
  2. **[ALL :HaveStudents Male]** → Defining a new category composed by invididuals that have only male students.

- 2nd **[EXISTS n r]**.

  `It stands for the class of individuals in the domain that are related by relation r to at least n other individuals`. In this case, we do not have to respect the *domain d* but we accept only new categories within *at least n* individuals.

  #### Example

  1. **[EXISTS 1 :Child]** → Defining a new category that contains all the individuals that have at least one child.
  2. **[EXISTS 2 :HasCar]** → Defining a new category composed by invididuals that have at least two cars.

- 3rd **[FILLS r c]**.

  `It stands for those inviduals that are r-related to the individual identified by c`. It is the most specific case; we are defining a complex concept related to a specific instance of a category.

  Example

  1. **[FILLS :HasCar aa123bb]** → Defining a new category made of all the individuals that have the car with plate `aa123bb`.
  2. **[FILLS :AreAttendingTheCourse thisCourse]** → Defining a new category that contains all the individuals that are attending this course.

- 4th **[AND $d_1 \dots d_n$]**.

  `It stands for anything that holds all the domains d at the same time`. If we refer to the notion of sets, this concept-forming operator is like the **intersection** between sets.

  Example

```
[ AND
  Company
      [ EXISTS 7 :Director]
      [ ALL :Manager [ AND
              Woman
              [ FILLS :Degree phD ]
          ]
      ]
      [ FILLS :MinSalary $24.00/hour]
]
```

  Defining a set of companies that holds all the properties described below.

First Order Logic focuses on sentences. A sentence is an expression that can be either true or false depending from the knowledge base.

We can try to determine the truth of a sentence also by Description Logics, using three different syntax rules:

- $d_1 \subseteq d_2$. Concept d_1 is **subsumed** by concept d_2, if and only if every individual that satisfies d_1 satisfies also d_2.

  Example

  PhDStudent $\subseteq$ *Student* → Every phd student is also a student.

- $d_1 = d_2$. Concept d_1 is **equivalent** to concept d_2, if and only if the inviduals that satisfy d_1 are the same that satisfy d_2.

  Example

  PhDStudent = [AND Student Graduated HasFunding] → A phd student is a gratuated student that has some funding for his reasearch.

- $c \rightarrow d$. The individual denoted by c satisfies the description expressed by concept d.

  Example

  federico → *Professor* → Federico is an instance of category `Professor`.

Above the basic idea, we have to provide a formal semantic of *DL*, which is the real meaning of this logic. The formal description of *DL* is given by the notion of **interpretation**.

`Definition`
An Interpretation I is a pair *(D, I)* where:

- *D* is the set of objects, called **domain**.
- *I* is a mapping function, called **interpretation mapping**, that goes from the non-logical symbols (symbols dependent from the domain taking in account) to elements of the domain *D*:
  1. For every constant c, $I[c] \in D$.
  2. For every atomic concept a, $I[a] \subseteq D$.
  3. For every role r, $I[r] \subseteq D \times D$.

Given the notion of *interpretation*, we can use it to determine if a sentence is true or false following the definition below.

`Definition`
Given an interpretation $I = (D, I)$, a sentence α is true ($I \vDash \alpha$), according to the following rules:

1. $I \vDash (c \rightarrow d)$ iff $I[c] \in I[d]$.
2. $I \vDash (d \subseteq d')$ iff $I[d] \subseteq I[d']$.
3. $I \vDash (d = d')$ iff $I[d] = I[d']$.

`Definition`
Given a *set of sentences* S, if all the sentences in S are true in I, we say that I is a **model** of S, $I \vDash S$.

While in First Order Logic we could ask: the predicate is true or false given the knowledge base, in Description Logic there are four different questions, summed up in:

- **Satisfiability**.
- **Subsumption**.

Definition
A concept d is **subsumed** by a concept d' w.r.t. S if $I[d] \subseteq I[d']$ for every model I of S.

- **Equivalance**.
- **Disjointness**.

Why do we introduce only the definition of Subsumption? Satisfiability, Equivalance and Disjointness can be computed by reducing them to subsumption. The following proposition hold:

Definition
For concepts d and d':

- d is **unsatisfiable** $\Leftrightarrow$ d is **subsumed** by $\bot$.
- d and d' are **equivalent** $\Leftrightarrow$ d is **subsumed** by d' and d' is subsumed by d.
- d and d' are **disjoint** $\Leftrightarrow$ $d \cap d'$ is **subsumed** by $\bot$.

3. Closed World Assumption vs. Open World Assumption

The set of Description Logics are based on an **Open World Assumption**: if a sentence cannot be inferred, then its truthness value is **unknown**.

Instead in First Order Logic, we assume a **Closed World Assumption**: the informations not stated inside the database program are simply **false**.

With *OWA* we can formalize different contexts in which the unknown informations can take all the possible values.

Example

Given the following problem:
Oedipus killed his father, married his mother Iokaste, and had children with her, among them Polyneikes. Finally, also Polyneikes had children, among them Thersandros

we can derive the following knowledge base in *FOL*:

```
hasChild(iokaste, oedipus).
hasChild(oedipus, polyneikes).
hasChild(iokaste, polyneikes). obtained
hasChild(polyneikes, thersandros).

patricide(oedipus).
¬patricide(thersandros).
```

and the same concepts by *DL*:

```
iokaste → [FILLS :HasChild oedipus].
oedipus → [FILLS :HasChild polyneikes].
```

```
iokaste → [FILLS :HasChild polyneikes].
polyneikes → [FILLS :HasChild thersandros].

oedipus → Patricide.
thersandros → ¬Patricide
```

We want to know `if Iokaste has a child that is a patricide and itself has a child that is not patricide`.

`1st Hypothesis`
Indeed, we know `Oedipus` is a patricide, and we know also he has a son, `Polyneikes`, but we don't have enough informations to state if `Polyneikes` is a patricide or not.

`2nd Hypothesis`
Indeed, we know `Polyneikes` has a son, `Thersandors`, and we know `Thersandros` is not a patricide, but we do not know if `Polyneikes` is a patricide or not.

Under the *OWA*, the answer is: the knowledge base does not entail the sentence; but this reasoning is not correct! All the possible **models** ($I \vDash S$) can be split up in two classes: one in which `Polyneikes` is a patricide, one where he is not a patricide.

The final answer should be always **YES**: in the first class the answer is <`Iokaste`, `Polyneikes`, `Thersandros`>, instead, in the second class the answer is <`Iokaste`, `Oedipus`, `Polyneikes`>. In all the models Iokaste has always a child that is a patricide and itself has a child that is not a patricide.

In this way, both cases answer to the original question. However, the huge problem of this approach is given by the total amount of unknown informations, that in *OWA* might be infinite. *OWA* requires a lot of computational effort than *CWA*.

---

4. Extending DL

It's possible to extend the Description Logic in several directions, like adding new *concept-forming operators*.

We have already the **EXISTS** operator. We could similarly add the operator **AT-MOST**, where **[AT-MOST n r]** describes individuals related by *role r* to *at most n* individuals.

Example

[AT-MOST 1 :Child] → Denotes all the parents that have only one child.

This extension could be dangerous if it is not used carefully.

Example

```
[ AND
    [EXISTS 4 r]
```

```
    [AT-MOST 3 r]
]
```

How do we treat such situation? We have a logical incoherence: at the same time we are asking for individuals r-related that have at least 4 items and at most 3 items.

We can add to the concept-forming operator **EXISTS** a given domain *d*, it becomes: **[EXISTS n r d]**, meaning all the individuals that are *r-related* to *at least n* individuals that are instances of concept *d*.

Example

[EXISTS 2 :Child Male] → defines the individuals that have at least 2 male children.

---

5. Conclusion

Description Logics is a family of logics. Each logic is different from each other, depending on which **operators** are admitted in that logic. Of course, more operators means:

- Higher expressivity.
- Higher computational costs.
- The logic itself may be **undecidable**.

Semantic Web and Knowledge Graphs

`Integrating in one place heterogeneous information resources`.

1. Introduction

Tim Berners-Lee came out in 2007 saying: the whole internet structure is a **huge knowledge base**, we need to exploit this knowledge base in order to perform **reasoning**.

A web page is a set of informations that tell us how the data should be represented, according to the needs of humans being. Therefore, the content is published with the principal aim of being **human-readable**.

Usually, it's used the *HTML* standard to show informations. HTML is focused on **how** to represent content instead of **what** is represented. This means: looking to a news paper web page we recognize instantly where is located the title, even though there isn't any notion about what is a title.

In addition to the content, web pages contains also **links** to other pages. Links are problematic, they create connections between pages but do not tell us anything about these connections or any clue about the subsequent resource selected.

The problem is: it is not possible to automatically reason about the data. We need a different representation of this huge amount of informations.

2. Semantic Web

The **semantic web** is the integration and combination of data from diverse resources, where the original Web mainly concentrated on the interchange of documents.

It's based on the idea of extend the current web within the knowledge given by the content.

Semantic web should preserve:

- Globality.
- Information **distribution**. Since the web is a complex structure, it's great having different resources around the world.
- Information **inconsistency**. Different incoherent and inconsistent opinions about the same information source.
- Information **incompleteness**.

Adding information is not enough. First of all, we should describe a structure of the informations and, after that, we need an inference mechanism to be able to infer new knowledge.

For these main reasons, Tim Berners-Lee introduced some tools:

- **URI**, Uniform Resource Identifier.
  Every resource available is identified by an unique tag. Each URI corresponds to one and only one concept, but more URI can refer to the same concept.

- **XML**, eXtensible Markup Language.
  HTML is an extension of XML, created for supporting data exchange between heterogeneous systems.

- **RDF**, Resource Description Framework.
  Any concept can be described by the **triple**: $< \text{subject}, \text{predicate}, \text{object} >$. If we look deeper, this representation is the same seen so far in First Order Logic.

  ### Example

  Given the triple:

  $$< \text{John}, \text{has}, \text{" 30 "} >$$

  is equal to:

  ```
  has(John, "30").
  ```

---

3. Knowledge Graphs

The notion of **knowledge graph** is born above two main reasons, which are:

- Overcoming search approaches based on *stastical data retrieval* with something able to *represent and reason* upon the knowledge.
- *Semantic web* seems to be too complex; based on description logics, they returned out to be too expensive from the time complexity point of view.

Google answers the previous problems by:

- Creating a common and simple **vocabulary**.
- Creating a simple and **robust** corpus of types.
- Pushing the Web to **adopt** these standards.

Example

From *description logics*, the **T-Box** approach is very expensive (T-Box describes generic properties about a category).

Given the concept:

$$\forall x \in \text{Human also } x \in \text{Mortal}$$

where `Human` and `Mortal` are categories, they have been represented by Google as simple records stored inside a database.

As we already know, the enterprise is very strong in database management tasks; instead of storing the axiom `Every human is mortal`, for each entity of `Human` category it has been decided to store in their databases the notion of **mortality** per individual.

This approach guarantees fast data retrieval.

Since reasoning is expensive by the computational point of view, we can just store the informations in terms of **nodes** and **arcs**. In this way, billions of factual knowledge are represented in form of **triples**, mixing together data coming from heterogeneous sources.

The construction of a knowledge graph is similar to semantic networks: boxes used to reprensent concepts and links used to describe relationships between boxes.

The quality of a knowledge graph is given by:

- **Coverage**. *Does the graph have all the required informations?*
- **Correctness**. *Is the information correct?*
- **Freshness**. *Is the content up-to-date?*

(The trustability of the informations is not done by knowledge graph, but instead it's on people judge their correctness).

Reasoning over Time

`Reasoning about the past predicting the future`.

1. Introduction

We live in a temporal dimension, our environment changes as times goes by. Up to now, we deal only with categories and objects, providing us a static perception; we would like to represent a system that **evolves** along the temporal dimension.

We might need to take in account also a **dynamic dimension**, related to the passing time.

There exist two main approaches to describe the dynamical evolution:

- **Event Calculus**.
- **Allen's Temporal Logic**.

2. Propositional logic

The main idea is to use **propositional logic** to represent an agent's beliefs about the world. Introducing this kind of logic, it's possible to describe the current state of the world as set of **propositions**. Inside the set should appear only true propositions, thus representing what the agent believes to be true.

Different sets of propositions, put in some order, would represent the world evolution perceived by an agent.

Example

A child has a bow and an arrow. He can shoot the arrow, throw away the bow, but he can also run, hide and crouch. Our agent observes the child, and it believes that:

$$\mathrm{KB}^0 = \mathrm{hasBow}^0, \mathrm{hasArrow}^0$$

The child shoots the arrow, our agent observes the child and it belivies:

$$\mathrm{KB}^1 = \mathrm{hasBow}^0, \mathrm{hasArrow}^0, \mathrm{hasBow}^1, \neg\,\mathrm{hasArrow}^1$$

Some observations are already coming out:

- 1st. The knowledge base at the time instant 0 still available for the knowledge base at time instant 1.
- 2nd. If we take all the knowledge of the previous steps, our agent can easily enter in an infinite search space.
- 3rd. We take this last assumption for granted, even though it is not a feasible observation in a real world application (we don't have the capability to store so much informations).

What's really interesting is given by the dynamic evolution that bring us from the KB^0 to the KB^1. *What happened in between*?

There is a general concept about **action**, whose execution changes the world. We would like to describe which are the **effects** of the executed actions.

Additionally, there is a notion of **state** that captures the world at a certain instant. The evolution of the world is given by a sequence of states.

Given the clue about **action**, we need to define a formalism about its effects that could change the world, also called **effect axioms**.

Example

What does it mean *shooting an arrow*?

$$\text{shoot}^t \rightarrow (\text{hasArrow}^t \Leftrightarrow \neg\, \text{hasArrow}^{(t+1)})$$

If the action shoot is executed at time instant t, this implies that at time instant t our agent has the arrow and at next time instant $(t+1)$ it will not have anymore the arrow.

$$\text{KB}^0 = \text{hasArrow}^0$$
$$\text{KB}^1 = \text{hasArrow}^0, \neg\, \text{hasArrow}^1$$

However, we are missing something. *Does the child still have the bow?* We don't know nothing about what happened to the bow at time instant 0.

From the previous observation, we formalize the **frame problem**: everything is untouched from the execution of an action should be moved to the next state. The effect axioms fail to state what remains unchanged after the execution of some actions and this approach is not possible; we have to describe an axiom for every unchanged state.

A solution would be the **frame axioms**, for each proposition not affected by the execution of an action, we will state that it is unchanged.

By the way, neither with frame axioms we can reach optimal solutions. Assuming m actions and n propositions, the set of frame axioms will be $O(m \times n)$.

3. Situation Calculus

Up to now, we focus on actions: a general concept used to express changes of the current environment. However, changing the viewpoint, focusing on the propositions that describe the world, we obtain a totally different solution.

In this case, we don't talk anymore about propositions, but instead about **fluents** (a special construct that can be true or false along the temporal dimension).

Successor State Axioms

Each state is described by a set of fluents F. Then, we define the following axiom:

$$F^{t+1} \Leftrightarrow \text{ActionCausesF}^t \vee (F^t \wedge \neg\, \text{ActionCausesNotF}^t)$$

The set of fluents F holds at time instant $(t+1)$ if and only if:

- The action ActionCausesF^t that causes the set of fluents F was performed.
- The set of fluents F is already true at time instant t and no action performed change it.

Example

$$\text{hasBow}^{(t+1)} \Leftrightarrow \text{pickUpBow}^t \vee(\text{hasBow}^t \wedge\neg\, \text{throwBow}^t)$$

Our agent has the bow if and only it performed the pickUpBow action or it already had the bow and did not throw away it.

Situation calculus has three different key features:

- 1st. The initial state is called **situation**.
- 2nd. Given an action a and a situation s, then $\text{Result}(s, a)$ is a situation.
- 3rd. A function that changes from a situation to the next one is called **fluent**.

It introduces the notion of **preconditions** of an action: the preconditions are the properties which must hold before the execution of an action. Action's preconditions are defined by the **possibility axioms**:

$$\phi(s) \rightarrow Poss(a, s)$$

Situation calculus adopts the **Successor State Axioms** following the possibility notion:

$$\text{Poss}(a, s) \rightarrow (F(\text{Result}(a, s)) \Leftrightarrow a = \text{ActionCausesF} \vee(F(s) \wedge a \neq \text{ActionCausesNotF}))$$

Example

$$\text{Poss}(a, s) \rightarrow (F(Result(a, s)) \Leftrightarrow a = \text{shoot} \vee(\neg\, \text{hasArrow}(s) \wedge a \neq \text{reload}))$$

Despite the fact that through situation calculus we are able to determine what is true before and after the execution of an action, we cannot say anything about what is true during the action.

4. Event Calculus

Sergot and Kowalsky, in 1986, came out with **event calculus**, based on logics used to describe the evolution of a system. Without any distinction, all the fluents and events are treated like **terms**; the predicate

$$\text{HoldsAt(fluent, situation/action)}$$

gives us the truth value of the fluent in any situation or action.

The formulation of event calculus is based on a fixed ontology and two different set of axioms:

1. Event calculus ontology.
2. Domain-independent axioms.
3. Domain-dependent axioms.

The **event calculus ontology** has six fixed predicates, which are:

- 1st $\text{Holds}(F, T)$. Fluents F holds at time instant T.
- 2nd $\text{Happens}(E, T)$. Event E happened at time instant T.
- 3rd $\text{Initiates}(E, F, T)$. Event E causes fluent F to hold at time instant T.
- 4th $\text{Terminates}(E, F, T)$. Event E causes fluent F to cease to hold at time instant T.
- 5th $\text{Clipped}(T_1, F, T)$. Fluent F has been made false between T_1 and T.
- 6th Initially(F). Fluent F holds at time instant 0.

Instead, the domain-independent axioms are:

- 1st Used to detect when a fluent is true.

  $$\text{HoldsAt}(F, T) \leftarrow \text{Happens}(E, T_1) \wedge \text{Initiates}(E, F, T_1) \wedge (T_1 < T) \wedge \neg\, \text{Clipped}(T_1, F, T)$$

  The fluent F is true at time instant T if event E occured at time instant T_1, event E causes fluent F at time instant T_1, time instant T_1 comes before time instant T and no one has changed the fluent F between T and T_1.
- 2nd Used to detect when a fluent is true.

  $$\text{HoldsAt}(F, T) \leftarrow \text{Initially}(F) \wedge \neg\, \text{Clipped}(0, F, T)$$

  The fluent F is true at time instant T if it is verified at the initial state and no one has changed the fluent F between time instant O and T.
- 3 rd Used to define the clipping of a fluent.

  $$\text{Clipped}(T_1, F, T_2) \leftarrow \mathit{Happens}(E, T) \wedge (T_1 < T < T_2) \wedge \text{Terminates}(E, F, T)$$

  The fluent F has been made false between T_1 and T_2 if happened the event E at time instant T, time instant T comes after T_1 and before T_2 and event E causes fluent F to cease to hold at time instant T.

All the reasoning about **event calculus** is made by these three rules. Finally, the domain-dependent axioms are divided into:

- 1st $\text{Initially}(F)$. The fluent F holds at the beginning.
- 2nd $\text{Initiates(E, F, T)}$. The execution of the event E at time instant T makes the fluent F true.
- 3rd $\text{Terminates(E, F, T)}$. The execution of the event E at time instant T makes the fluent F false.

While situation calculus relies on deductive planning, event calculus is based on reactive rules, allowing us to represent the state of a system in logical terms, in particular in First Order Logic.

Despite we can easily implement event calcus by **First Order Logic**, it is not safe if a term contains variables and, also, another big problem is given by the representation of predicate $\neg\, \text{Clipped}(\dots)$ by **negation as failure**.

5. Allen's Logic for reasoing over temporal aspects

Event calculus is based on the notion of event or, in other in words, on the happening of things. *Are these events instantaneous or do they have a duration*?

Several observations could answer this query, but what is really true is the fact that do not exist any instantaneous action in a real world application, each of them has a duration.

Instead of thinking about **point in times**, we have to reason about **intervals** where properties are true or false. This is the basic idea about **Allen's logic**.

In Allen's view, intervals are more natural categories for humans, simplier to define when some properties or fluents hold (intervals as first entities).

Given an interval i, we know it has a beginning and an end; we can retrieve such informations by two different actions:

- $\text{Begin}(i)$, that returns the initial time point.
- $\text{Ends}(i)$, that returns the final time point.

Allen's logic is a powerful tool that allows us to describe the evolution of a system in terms of intervals. However, logic becomes easily undecidable when we have to deal with more complex entities, such as conjuction and disjuction of intervals.

Knowledge and Beliefs

`Sharing a common knowledge along the evolution of the time`.

1. Introduction

In this chapter we introduce the third mechanism used to reason about the dynamical evolution of a system, and it came out from the **modal logics**.

The set of modal logics was an attempt for representing agents inside a world, without **complete knowledge**. Each agent might have a different knowledge and, until now, it does not exist any notion of common knowledge.

However, we would like to describe contact points between agents: they saw same events inside the same environment, leading us to the formalization of shared knowledge.

First Order Logic is a great tool for describing knowledge, even though it misses useful shortcuts for dealing with concepts such as knowledge, beliefs, wishes.

2. Knowledge and Beliefs

So far, we discussed about representing knowledge and we adopted this point of view: our agent do some actions based of the knowledge available in the environment.

However, this assumption has some limits: our agent has a knowledge base but it doesn't know it; it does not have any consciussness about its knowledge. It would be great if agents had the capability to retrieve informations from each other.

We do not have the logical stuff to represent consciussness, but we can try using **modal logics**. Usually modal logics are used to define the **propositional attitudes** of an agent, such as: **Believes**, **Knows**, **Wants**, **Informs**. For each of them we have a **Modal Operator**.

The basic idea was to introduce new, special operators, called Modal Operatos, that behaves differently from logical operators seen so far.

We can summarize modal logics in few steps:

- 1st It has the same syntax as FOL.
- 2nd It uses modal operators.
- 3rd Modal operators take input sentences.
- 4th Each modal operator takes inside its scope: the name of an agent and a sentence.
- 5th Semantics is extended with the notion of **possible worlds**.
- 6th Possible worlds are related through an **accessibility relation**.

Example

$$\mathbf{K}_a P$$

The agent a knows the sentence P. $\mathbf{K}$ is the propositional attitude **Knows**.

---

3. Modal Logics

Modal logics is a family of logics, as description logics, where different types of modal operators has been introduced in order to represent attitudes.

However, we are not really interested about these operators; we introduced them in order to give a clue about accessibility between possible worlds: an agent *knows p* if in any accessible world, accessible from the current one, **p is true**.

Given a notion of world, we don't know how it will evolve, but right now we have the tools to access to the informations of the future worlds.

Like any logic, we have to discuss about the axioms of the modal logics. There are different axioms for each modal operator, we focus on the Knows modal operator ones, and, also, it's up to us to add new ones if we think they are strictly necessary.

- 1st. Axiom A0. All instances of propositional tautologies are valid.
- 2nd. Modus Ponens. If ϕ is valid and $\phi \rightarrow \psi$ is valida, then ψ is valid.
- 3rd. Distribution Axiom A1.

  $$K_i\phi \wedge K_i(\phi \rightarrow \psi) \rightarrow K_i\psi$$

  In other words: if the agent i knows ϕ and the implication $\phi \rightarrow \psi$ is valid, then the same agent knows also ψ. This axiom shows how could be dangerous the modal operator **Knows**: we can deduce anything from the implication of sentences **considered** true.
- 4th. Knowledge Generalization Rule. For all the worlds M, if $M \vDash \phi$, then $M \vDash K_i\phi$. It says: if ϕ is true in all the possible worlds, then for sure the agent i **knows** ϕ.

  This axiom is another example about the dangerousness of the **Knows** modal operator. We need a weaker operator in order to avoid logical incompatibilities.
- 5th. Knowledge Axiom A2. If an agent knows ϕ, then ϕ is true

  $$K_i\phi \rightarrow \phi$$

  But, *is it always true? Shall we say always that ϕ is true if the agent i knows ϕ?* The real answer is: it depends.
- 6th. Positive Introspection Axiom A3.

  $$K_i\phi \rightarrow K_iK_i\phi$$
- 7th. Negative Introspection Axiom A4.

  $$\neg K_i\phi \rightarrow K_i\neg K_i\phi$$

The agent might have a partial knowledge of the world, but with these axioms we get sure that at least it has a perfect knowledge about its own knowledge.

4. Linear-time Temporal Logic

Given the amount of the previous modal operators, only few of them are really interesting: they allow us to handle what will be true in the temporal dimension given a dynamic evolution.

LTL maps the accessibility relation between worlds into the temporal dimension. Each of them is labelled with an integer, usually a pedix, meaning a time instant.

Any world has two possible ways to evolve:

- **Linear-time**. From each world, there is only one other accessible world.
- **Branching-time**. From each world, many worlds can be accessed.

Temporal logics introduce the following operators:**Allen's Logic**

- $\circ\phi$. Something is true at the next moment in time.
- $\Box\phi$. There is a ϕ that is always true in the future. Now ϕ might be true o false, but at the next moment it will be surely true.
- $\diamond\phi$. There is a ϕ that is true sometimes in the future. Sooner or later it will become true.
- $\phi U \psi$. There exists a moment when ψ hold and ϕ will hold from now until this moment s (also called **strong until**).
- $\phi W \psi$. ϕ will hold from now on unless ψ happens, in which case ϕ will cease. Here, we are not sure that ψ will occur in the future (also called **weak until**).

LTL is the third mechanism for reasoning over time; previously we saw **Event Calculus** and **Allen's Logic**. It's quite effective for modelling distributed systems, where different agents can exchange messages and information, and, also, for modelling checking systems (checking if a system, given a certain world, contains or not some stuff).

Probabilistic Logic Programs

`Dealing with uncertainty by Probabilistic Reasoning`.

1. Introduction

Up to now, we discussed only about logic formula, that can be either true or false. However, sometimes, we do not have the ability to define their truth values; we have to take in account also the **probabilities** that an event occurs.

Prolog is a powerful programming language that allows us to represent and reason upon some given knowledge. *Why don't we expand Prolog to include probabilities as well*?

This is the basic idea about Probabilistic Logic Programming: since the notion of meta-interpreter, we can design a totally new Prolog machine that allows us to include also probabilities inside the reasoning schema.

There are already two systems that follow this approach:

- **ProbLog**.
- **LPAD** (Logic Programs with Annotated Disjunctions).

2. Probabilistic Logic Programming

A probabilistic logic program defines a **probability distribution** over normal logic programs, and each of them is a part of the distribution, named **possible world** or simply **world**.

In LPAD the clauses have a different structure. The **head** is extended with disjunctions and each **disjunct** is annotated with a probability (the head of the clause is the probability distribution of random variable).

Example

Assuming these sample points:

- *people with flu* also *sneeze in 70% of the cases*.
- *people with hay fever* also *sneeze in 80% of the cases*.

The respective snippet of code is:

```
sneezing(X): 0.7 ; null: 0.3 :- flu(X).
sneezing(X): 0.8 ; null: 0.2 :- hay_fever(X).
```

For each case we have an associated probability. But, *what happens if we get both*? *Which is the probability*? We add to the knowledge base other informations, such as:

- *people with flu* also *sneeze in 70% of the cases*.
- *people with hay fever* also *sneeze in 80% of the cases*.
- Bob has flu.
- Bob has hay fever.

Again, the snippet becomes:

```
sneezing(X): 0.7 ; null: 0.3 :- flu(X).
sneezing(X): 0.8 ; null: 0.2 :- hay_fever(X).
flu(bob).
hay_fever(bob).
```

Let's assume a Closed World Assumption, everything we know is already stated in the database program. We can obtain the *possible worlds* (normal logic programs that determine the probability distribution) by selecting an atom from the head of the clause (the clause must be already matched). The steps to follow are:

1. Ground the free variables with constants.
2. Select an atom from the head of each clause.
3. The total number of possible worlds is equal to 2^N (N is the amount of clauses stated).

Matching all the free variables with constants:

```
sneezing(bob) :- flu(bob).
sneezing(bob) :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

Defining all the possible worlds:

1st

```
null :- flu(bob).
sneezing(bob) :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

2nd

```
sneezing(bob) :- flu(bob).
null :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

3rd

```
null :- flu(bob).
sneezing(bob) :- hay_fever(bob).
```

```
flu(bob).
hay_fever(bob).
```

4th

```
null :- flu(bob).
null :- hay_fever(bob).
flu(bob).
hay_fever(bob).
```

Finally, we got more simple rules moving out the probabilities. However, we can infer the probability of each possible world, combining together the original ones:

- 1st $P(w_1) = 0.7 \times 0.8$.
- 2nd $P(w_2) = 0.7 \times 0.2$.
- 3rd $P(w_3) = 0.3 \times 0.8$.
- 4th $P(w_4) = 0.3 \times 0.2$.

These results are obtained by more formal steps, divided into:

- **Atomic Choice**. Selecting an atom from the head of the clause:

  $$(C, \theta, i)$$

  where:

  - C is the clause.
  - θ is the substitution of the free variables.
  - i is the index of the atom selected.

- **Compositive Choice**. The **set k** of atomic choices done, its probability is computed by:

  $$P(k) = \prod_{(C,\theta,i)} P(C, i).$$

- **Selection** σ. A **total** composite choice, we have chosen an atomic choice for each inner clause. This selection generates a possible world (a normal logic program part of the probability distribution). Its probability is given by the following formula:

  $$P(w_\sigma) = P(\sigma) = \prod_{(C,\theta,i)\in\sigma} P(C, i).$$

In general, given a query Q and a world w:

$$P(Q|w) = \begin{cases} 1 & \text{if } Q \text{ is true in } w \\ 0 & \text{altrimenti} \end{cases}$$

Given the query `sneezing(bob)`, we deduce that it's true in the first, second and third possible world. The probability of `sneezing(bob)` is equal to the sum of the possible worlds probabilities in which `sneezing(bob)` is true.

$$P(\text{sneezing(bob)}) = 0.56 + 0.21 + 0.14 = 0.93$$

Forward Reasoning

`Automating processes`.

1. Introduction

According to what we are looking for, we have a different types of knowledge and ways to represent them.

So far we have seen many examples of **backward chaining**: starting from the consequences of a rule, we look for the premises in order to prove it.

Example

Given the knowledge base:

```
p(X) :- q(X), r(X).
q(1).
r(1).

?- p(X).
   Yes X = 1.
```

The resolution process of the rule `p(X)` starts from the head of the clause and then it moves to the body of the same clause. The final result is true if and only if it's possible to prove the body of the rule.

`Starting from the consequence we look for the premises`.

Backward chaining is fine for a **static knowledge**, expecially when we need to reason about goal-oriented problems. However, it's not a feasible solution for real system applications, since the knowledge base changes continuosly.

A possible work-around could be: add new informations inside the storage and restart the reasoning. By the way, this is not really possible when we are dealing with a huge code base.

We would like to have a sort of knowledge base in which we can add **dynamically** new facts. This is achieved **rules** by the notion of **rule**.

2. Rules

There exists another mental constructor that we as humans being make so much use, named **rules**. Rules are a very common way for explaining knowledge and sometimes parts of it are easier to represent by them.

In addition, rules allow us to add dynamically new facts inside the knowledge base.

In contrast to backward chaining, rules adopt **forward chaining**: from the premises we try to prove the consequences.

Example

The previous query can be written following forward chaining:

```
p(X) :- q(X), r(X).

↓

p(X) ← q(X), r(X).

↓

q(X), r(X) → p(X).
```

The simplest form of a rule is the logical implication:

$$p_1, \ldots, p_i \rightarrow q_1, \ldots, q_j$$

- $p_1, \ldots, p_i$ are called the **premises** or the **Left Hand Side** (LHS).
- $q_1, \ldots, q_j$ are called the **consequences** or the **Right Hand Side** (RHS).

3. Production rules systems

Our goal is to define a rules system, based on forward chaning, that guarantees the dynamical addition of new facts within the knowledge base. This is possible by the **production rules approach**.

In the production rules approach, new facts can be added dynamically to the knowledge base, also called **working memory** (WM). If new facts match with the left hand side of any other rule, the whole mechanism is **triggered**, until it reaches **quiescence** again.

The right hand side are the effects of the rules that could influence the external world and the knowledge base itself. A side effect might be:

- 1st. Insertion of a fact.
- 2nd. Delition of a fact.
- 3rd. Retriggering a rule.

Several strategies have been introduced in order to get a consistent working memory and they follow the same behavior:

- 1st **Matching**. Searching for the rules whose LHS matches with the new fact.
- 2nd **Resolution**. Trigerring rules and conflicts between them are solved.
- 3rd **Execution**. RHS are executed and their effects are performed, changing again the working memory.
- 4th **Repeat**. Repeating the same steps until the quiescence is reached.

The real bottleneck is the first step: choosing which are the rules to activate. The performance of any production rules system can be heavily influenced by the efficiency of the working memory.

4. The RETE algorithm

The **RETE algorithm** focuses on the first step, since it's the most critical one. The **match step** consists on computing which are the rules whose LHS is matched within the new fact.

The LHS is usually expressed as a conjuction of patterns, and deciding whether if the premises are already met is the same to check if each pattern has a corresponding fact in the working memory.

Following this idea, the RETE algorithm tries always to avoid iteration over facts and rules, any time a new piece of knowledge is added.

In order to avoid iteration, it defines the **conflict set**: a set of all the instantiations of product rules completely matched (their LHSs are already matched with the working memory), ready for the execution. Each instantion is a couple described as follows:

$$< \text{Rule}, \text{List of facts that matches the LHS} >$$

The conflict set is obtained by particular construction:

- 1st **Matching**. It's little a bit different than before. When a new fact is added to the working memory, it is propagated through two different networks, which are:
  - 1α^{st} **Alpha Network**. Composed by patterns describing properties of the single entity.
  - 1β^{st} **Beta Network**. Composed by patterns describing shared properties between entities.

  At the end of each network, their outcomes are stored inside the alpha-memory and beta-memory. Memories are used to store partial result related to the patterns matched with the new facts.
- 2nd **Resolution**. The beta-memory contains already the conflict set. Right now, we have to decide in which order the rules must be executed. We can decide different orders, such as: rule priority, rule natural order, rule complexity, ...
- 3rd **Execution**. Each record of the conflict set, ready for the activation, is executed according to the choosen order.