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.

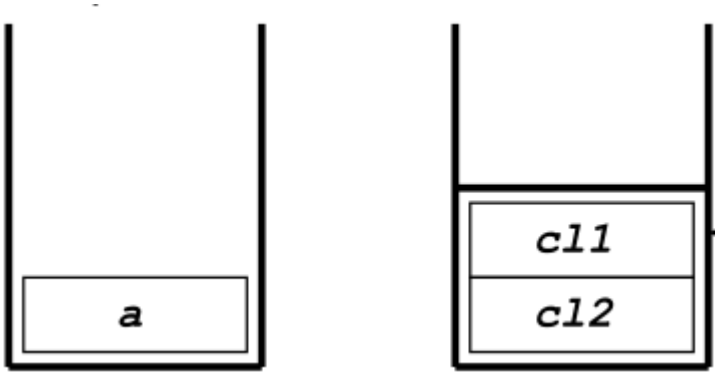
Any execution process is built upon two stacks:

- **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).

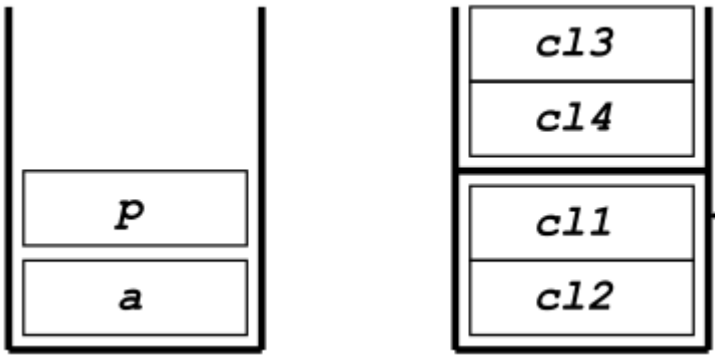
When a goal succeeds, Prolog might still have open choice points. When a goal fails, Prolog backtracks to the most recent choice point to try an alternative branch.

Example

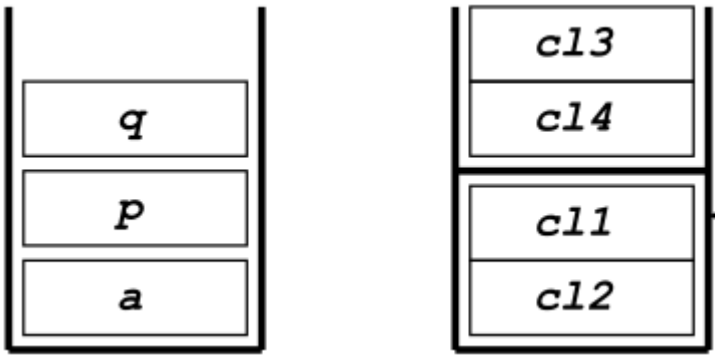
```
(cl1) a :- p, b.
(cl2) a :- r.
(cl3) p :- q.
(cl4) p :- r.
(cl5) r.
```



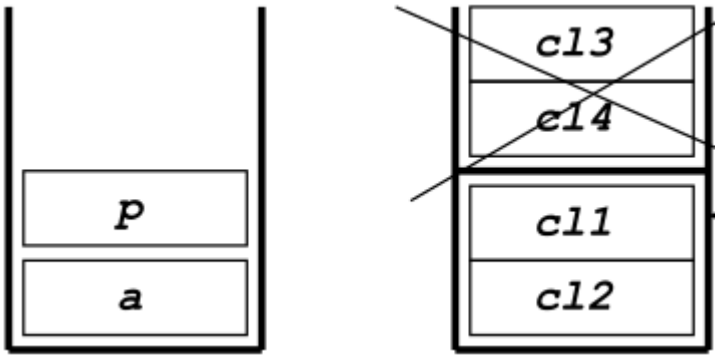
In the execution stack, we place the current goal A. In the backtracking stack, we place the references to all the clauses that match our current goal A.



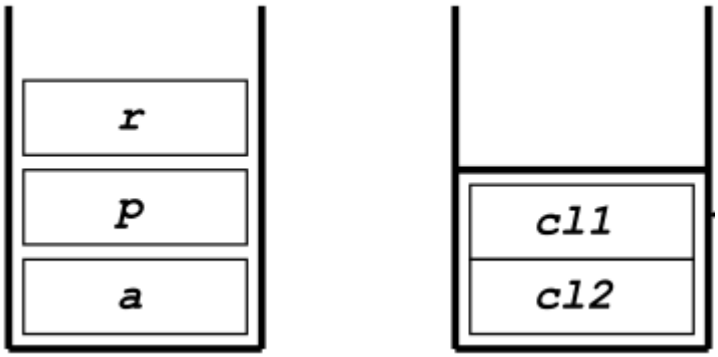
In the next step, our current goal will be P, as we can observe from the rule `a :- p, q`, and again inside the backtracking stack, we will put all the clauses that match the current goal `p`. We always choose the rule following the syntactical order, so `p :- q` comes before `p :- r`.



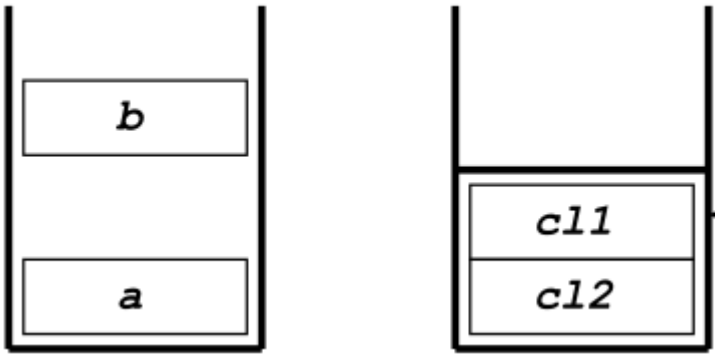
Following this behavior, our next goal will be Q. Again, we put it in the execution stack, but unfortunately, it will fail because it's not defined inside our program. So, the interpreter will select the next open choice point.



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:

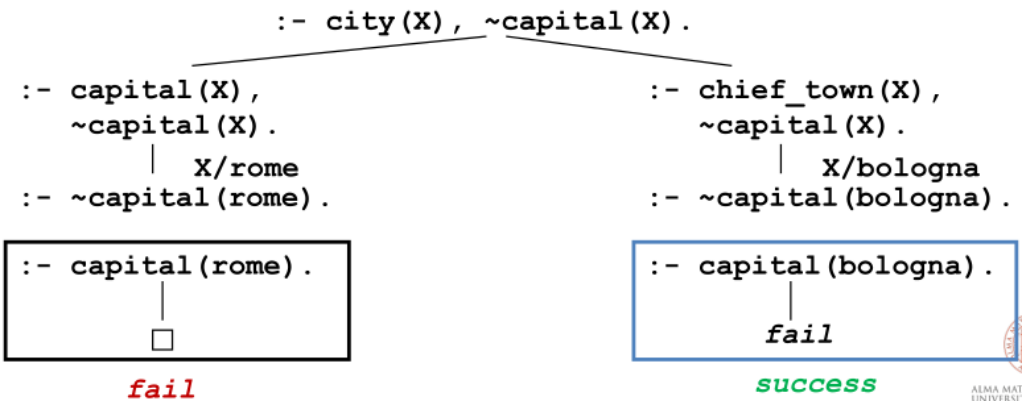
1. Do not select any negative literal L_i if it is **not ground** (meaning completely instantiated, with no variables).
2. If the selected literal L_i is positive, then apply a normal SLD resolution step.
3. If is a negative literal, `~A` with A ground, and A fails in finite time, then L_i succeeds, 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.