PROGRAM = DATABASE = KB = SET OF CLAUSES

3 KIND OF CLAUSES.

- FACTS

  ALWAYS TRUE. EXAMPLE:

  father(GUIDO, ANDRE).

  THEY CAN CONTAIN VARIABLES

- RULES

  Depend on something: HAVE IMPLICATION.

  parent(P_1, P_2) :- father(P_1, P_2).

  head — body

  THEY CAN CONTAIN VARIABLES

- QUERIES/GOAL

  Are the ones that we ask TO THE PROGRAM USING THE INTERPRETER.

  EXAMPLE: :- father(P_1, P_2)

FACTS + RULES = DATABASE/PROGRAM /KB

(ALL THESE ARE PREDICATES, INDEED A FORMULA CAN'T BE A FUNCTION ALONE)

SYNTAX:

- COMMAS ARE LIKE $\wedge$!!
- MIND THE "." AT the end of each CLAUSE
- THE ORDER IS IMPORTANT AS WE'LL SEE. FACTS ARE BETTER TO BE POSITIONED BEFORE RULES, USUALLY.
- RULES MUST HAVE THIS FORM: $A :- B_1, B_2, B_3 \ldots$.
- FACTS MUST HAVE THIS FORM: A.
- QUERIES/GOALS MUST HAVE THIS FORM: $:- B_1, B_2, B_3 \ldots$.

WHERE A, B_i are ATOMIC FORMULAS: $P(t_1, t_2 \ldots t_n)$ where P IS A PREDICATIVE SYMBOL AND t are terms.

WE'LL DISCUSS WHAT ARE TERMS LATER, NOW LOOK HOW DOES PROLOG WORK →

- HORN CLAUSE : THE HEAD MUST BE AN ATOM.

HOW DOES PROLOG REPLIES TO QUERIES?

WITH UNIFICATION:

AN ATOMIC GOAL/QUERY IS PROVED BY CONFRONTATION WITH THE KB. IT'S TRIED THE UNIFICATION WITH FACTS OR WITH HEADS OF THE RULES (IF IT UNIFILES WITH A HEAD, THEN THE BODY (SUBSTITUTED) OF THAT RULE IS TREATED AS A GOAL → MUST BE UNIFIED... IT'S A RECURSIVE REASONING), UNTIL THE UNIFICATION IS DONE WITH A FACT, OR IT'S NOT POSSIBLE TO UNIFY.

IF THE QUERY/GOAL HAS MORE ATOMIC GOALS, THEY ARE PROVED FROM LEFT TO RIGHT.

EXAMPLE:

append IS NOT INTERPRETED. BUT THANKS TO UNIFICATION

$$KB = \begin{cases} \text{append}([\,], X, X). & \text{FACT: IT'S SAYING: } [\,] + X = X \text{ (we give it this MEANING)} \\ \text{append}([X|Z], Y, [X|T]) :- \text{append}(Z, Y, T) & \text{RULE: WE GIVE IT THIS MEANING:} \\ & \text{IF } [Z]+[Y]=T \Rightarrow [X|Z]+Y=[X|T] \end{cases}$$

QUERY / GOAL:

:- append([a,b],[c,d],[a,b,c,d]) WE ARE ASKING IF $[a,b]+[c,d]=[a,b,c,d]$

PROLOG TO ANSWER DOES THIS:

1) TRIES TO UNIFY append([a,b],[c,d],[a,b,c,d]) and append([ ],X,X)
BUT IT FAILS

2) TRIES TO UNIFY append([a,b],[c,d],[a,b,c,d]) WITH append([X|Z],Y,[X|T])

IT'S POSSIBLE: {X/a, Z/[b], Y/[c,d], T/[a,b,c,d]}

note that [X|Z]
THIS MAKES Z BE UNIFIED WITH [b] INSTEAD OF b

THEN THE BODY IS TREATED AS A GOAL/QUERY, SO WE MUST FIND UNIFICATION FOR THE BODY SUBSTITUTED: append([b],[c,d],[a,b,c,d])

→ CONTINUE

IT DOESN'T UNIFY WITH THE FACT. SO WE TRY TO UNIFY IT WITH THE HEAD OF THE RULE:

append [b], [c,d], [b,c,d])
append ([X|Z], Y, [X|T])

IT UNIFIES WITH THIS UNIFIER:

$\{X/b, Z/[], Y/[c,d], T/[c,d]\}$

SO WE MUST UNIFY THE BODY NOW (SUBTITUTED): append(Z, Y, T) ≡ append([], [c,d], [c,d])

IT FINALLY CAN BE UNIFIED WITH THE FACT: append([], X, X])
append([], [c,d], [c,d])
$\{X/[c,d]\}$

SO THE QUERY IS PROVED!

SO PROLOG IS SAYING US THAT IT'S TRUE ~~THAT~~ THE INITIAL GOAL/QUERY:
append([a,b], [c,d], [a,b,c,d])

NOTE: THIS IS WHAT PROLOG DOES BUT YOU NEED TO KNOW IT AND UNDERSTAND IT BECAUSE YOU MUST WRITE FACTS AND RULES TO MAKE IT WORK.

TERMS:

- CONSTANTS: NUMBERS OR STRINGS STARTING WITH lower case character.
  example: 2, pippo, a92, 135, aB
- FUNCTIONS: LOOK LIKE PREDICATE BUT YOU DISTINGUISH THEM BECAUSE FUNCTIONS MUST BE WITHIN A PREDICATE (AND PREDICATES CAN'T CONTAIN PREDICATES.).
  YOU DISTINGUISH THEM FROM CONSTANTS BECAUSE FUNCTIONS HAVE INPUTS. examples: $f(t_1)$, $f(g(1))$, $b(a)$
- VARIABLES: THEY START WITH UPPER CASE CHARACTER.
  ALSO "_" IS A SPECIAL VARIABLE CALLED "ANONIMOUS"

  examples: X, Pippo, X1, _x, _X

IT'S ALWAYS IMPLICIT THE UNIVERSAL QUANTIFIER:

$p(t_1, t_2, \dots)$ actually means: $\forall x_1, \forall x_2 .. \; p(t_1, t_2 ..)$
IF x_1, x_2 appears in t_1, t_2 ..

SO FOR RULES IF WE CALL $x_1 .. x_m$ THE VARIABLES OF THE HEAD
and " $y_1 .. y_n$ " " " " BODY
THE MEANING OF A RULE IS:

$$\forall x_1 \forall x_2 .. \; \forall y_1 \forall y_2 .. \; ((B_1, B_2, B_3) \rightarrow A)$$

WHICH IS THE SAME OF

$$\forall x_1 \forall x_2 .. \; ((\exists y_1 . \exists y_2 .. (B_1, B_2, B_3)) \rightarrow A)$$

EXAMPLE:
father(x,y) means "X is the FATHER OF Y"

nonno(X,Y) :- father(X,z), father(z,y) means "~~for~~ ∀ Y, X is GRANDPA OF Y IF ∃ z s.t. X is FATHER OF z and z IS FATHER OF Y."

FORMALLY WHAT WE'VE SEEN WITH THE EXAMPLE OF THE ANSWER TO THE QUERY "APPEND(...)" IS DESCRIBED IN THIS WAY:

PROLOG USES THE "SLD RESOLUTION" TECHNIQUE (WHICH CORRESPONDS TO "BACKWARD CHAINING" FOR HORN CLAUSES). WHICH IN GENERAL IS NOT COMPLETE, BUT IT IS IN CASE OF HORN CLAUSES.

INFORMALLY IT MEANS THAT WHEN WE'RE DOING THE PROOF OF A GOAL WE'RE LOOKING FOR A FORMULA WHICH UNIFY WITH IT, BELONGING TO THE KB. BUT WE HAVE 2 CHOICE POINT:

1) THE GOAL CAN HAVE MANY ATOMS., which one DO WE UNIFY FIRST? "COMPUTATION RULE"
2) THE KB HAS MANY FORMULAS, which one DO WE UNIFY FIRST? "SEARCH STRATEGY"

PROLOG HAS THIS STRATEGY:
1) START WITH THE LEFT MOST ATOM OF THE GOAL
2) START WITH THE TOP MOST FORMULA OF THE KB

INDEED CAN BE THOUGHT AS A TREE SEARCH

TO BE MORE PRECISE WHEN WE DO UNIFICATION WITH AN ATOM AND A FORMULA OF THE KB, THE FORMULA IS RENAMED (OTHERWISE I WOULD SUBSTITUTE VARIABLES IN THE WHOLE KB).

MORE DETAILS:
- THE UNIFICATION IS DONE WITH THE MOST GENERAL UNIFIER = MGU
- THE GOAL IS ALSO CALLED "RESOLVENT".
- QUANDO DURANTE L'UNIFICAZIONE MUST BE UNIFIED A VARIABLE WITH A COMPOSED TERM (A FUNCTION), IT MUST BE CHECKED THAT THE VARIABLE DOESN'T OCCUR WITHIN THE FUNCTION, OTHERWISE WE FAIL.

SO WE HAVE TWO CHOICES → "OCCUR CHECK": WE CHECK IT TO BE SURE, BUT THE COMP. COST IS HIGH

↓ WE USE ALTERNATIVE METHODS WHICH DO NOT CHECK IT EXHAUSTIVELY ED SOMETIMES THEY OUTPUT WRONG ANSWERS ⇒ THE PROOF IS NOT CORRECT/SOUND

THE RESOLUTION SLD HAS 3 POSSIBLE OUTPUTS:

1) SUCCESS
2) FINITE FAIL (NOT POSSIBLE UNIFICATION)
3) INFINITE FAIL (INFINITELY SUBSTITUTION)

(ACTUALLY LOOP AVOIDED THANKS TO OCCUR CHECKING, BUT THAT'S THE OUTPUT (?))

LET'S DO AN EXAMPLE OF THE 3° CASE!

$$KB = \begin{cases} SUM(0, x, x). \\ SUM(S(x), y, S(z)) :- SUM(x, y, z). \end{cases}$$

GOAL/QUERY: $:- SUM(A, B, C)$ (NOTE THAT IN THIS CASE THE GOAL HAS VARIABLES, SO THE SUBSTITUTION COULD BE OF THEM)

1° ITERATION:

- NO UNIFICATION WITH THE FACT. → ERRORE: IT DOES UNIFY WITH THE FACT ACTUALLY. BUT LOOK THE EXAMPLE ANY WAY!
- UNIFICATION WITH THE RULE HEAD:
  - BEFORE THE RULE IS RENAMED: $SUM(S(x_1), y_1, S(z_1)) :- SUM(x_1, y_1, z_1)$
  - UNIFY WITH $SUM(A, B, C)$

$\Rightarrow \theta = \{A/S(x_1), B/y_1, C/S(z_1)\}$ SUBSTITUTION

$\Rightarrow$ NEW GOAL $= (BODY)\theta = :- SUM(x_1, y_1, z_1)$

2° ITERATION, NEW GOAL: $SUM(x_1, y_1, z_1)$

- NO UNIF. WITH THE FACT.
- UNIFICATION WITH THE RULE HEAD:
  - BEFORE THE RULE IS RENAMED: $SUM(S(x_2), y_2, S(z_2)) :- SUM(x_2, y_2, z_2)$
  - UNIFY WITH $SUM(x_1, y_1, z_1)$

$\Rightarrow \theta = \{x_1/S(x_2), y_1/y_2, z_1/S(z_2)\}$

$\Rightarrow$ NEW GOAL $= (BODY)\theta = SUM(x_2, y_2, z_2)$

↳ NOTE THAT WITHOUT RENAMING YOU WOULD OBTAIN A WRONG NEW GOAL

3° ITERATION

4° "

⋮

∞ INFINITE FAIL

- THERE COULD BE MORE POSSIBLE SUBSTITUTIONS. TO ASK THEM USE THE ";" AT THE END OF THE GOAL INSTEAD OF "."
- THIS WAS "PURE PROLOG". TO BE A REAL "PROGRAMMING LANGUAGE" SOME BUILT-IN PREDICATES ARE IMPLEMENTED.
- THE ANSWER GIVEN BY PROLOG IS YES OR NO + THE SUBSTITUTION OF THE GOAL VARIABLES.

PROLOG DOESN'T HAVE A CONSTRUCT FOR ITERATIONS (LIKE "WHILE", "FOR").

SO TO OBTAIN IT WE MUST DO "RECURSIVE DEFINITIONS".

LET'S TALK ABOUT LIST TO EXPLAIN IT:

BUT FIRST UNDERSTAND THIS:

- A LIST IS USUALLY THOUGHT AS [HEAD | TAIL] (NOT NECESSARY. IT'S USEFUL FOR RECURSION)
- WHEN GIVEN [a, b, c] → HEAD = a, TAIL = [b, c]
  - HEAD = "FIRST ELEMENT"
  - TAIL = "ALL THE REST, AS A LIST."

- IF YOU WRITE [H1, H2|T] THE UNIFICATION WITH [a, b, c] WILL GIVE YOU H1/a, H2/b, T/[c]

- IF YOU WRITE [_, H2|_] THE UNIFICATION WITH [a, b, c] ⇒ H2/b

- [H|T] UNIFY [] ⇒ FAIL
- [a,[]] UNIFY [A,B|T] ⇒ A=a, B=[], T=[]
- [H|T] UNIFY [a] ⇒ H=a T=[]
- [a,b,c] UNIFY [a|T] ⇒ T=[b,c]
- [] UNIFY WITH []
- [a,b,c] UNIFY [b|T] ⇒ FAIL
- [One] UNIFY [two|[]] ⇒ One=two.

EXAMPLES:

KB = p([H|T], H, T).

QUERY:

:- p([a,b,c], X, Y)

ANSWER:

{H/a, T/[b,c], X/a, Y/[b,c]}

↳ THE ANSWER IS ONLY THE SUBSTITUTION OF YOUR QUERY: {X/a Y/[b,c]}

:- p([a], X, Y)

{H/a, T/[], X/a, Y/[]}

#

{X/a; Y/[]}

:- p([], X, Y)

FAIL!

LIST SEARCH:

YOU USE THE NOTATION [HIT] TO CHECK IF THE ELEMENT YOU'RE LOOKING FOR IS IN THE HEAD. IF IT IS NOT YOU DISCARD THE FIRST ELEMENT AND TRY AGAIN. YOU DO IT UNTIL THE LIST IS EMPTY.

IMPLEMENTATION:

KB = { on_list(ITEM, [ITEM|REST]).
on_list(ITEM, [DISCARD_HEAD|TAIL]) :- on_list(ITEM, TAIL).

QUERY:

?- on_list(pear, [apple, pear, peach])

PROCESS DONE BY PROLOG:

1) TRIES TO UNIFY on_list(pear, [apple, pear, peach])
WITH THE FACT on_list(ITEM, [ITEM|REST]) ⇒ θ = {ITEM/pear, ITEM/apple} ↺ FAIL

2) TRIES TO UNIFY on_list(pear, [apple, pear, peach])
WITH THE RULE on_list(ITEM, [DISCARD_HEAD|TAIL]) ⇒ θ = {ITEM/pear, DISCARD_HEAD/apple, TAIL/[pear, peach]}

OK

SO NEW_GOAL = (on_list(ITEM, TAIL)) θ = on_list(pear, [pear, peach])

3) TRIES TO UNIFY on_list(pear, [pear, peach])
WITH THE FACT on_list(ITEM, [ITEM|REST]) ⇒ θ = {ITEM/pear, REST/[peach]}

ANSWER: YES.

LIST CONSTRUCTION

TRY TO WRITE THE PROGRAM/KB WITH CH GIVEN THE QUERY
:- append([a,b],[c,d],RESULT) GIVES YOU RESULT = [a,b,c,d]. (OR ANY OTHER CASES OF LIST1, LIST2)

KB = { append([], L2, L2)
append([H|T], L2, [H|RESULT]) :- append(T, L2, RESULT)

("THE FINAL LIST IS [H|RESULT] IF RESULT = appnd T, L2.")

IT WORKS BUT WHAT HAPPENS IF THE QUERY IS append([], [c,d], RESULT)?
IT FAILS BECAUSE IT CAN'T UNIFY [H|T] WITH [].
SO YOU MUST ADD THIS: append([], L2, L2)

GIVEN THE QUERY :- new_list([1,12,3,14,5,8], RESULT). THE PROGRAM MUST OUTPUT RESULT = [12,14,8] SO ONLY THE VALUES >6.

KB = { new_list([], []). /* BASE CASE */
new_list([H|T], [H|RESULT]) :- H>6, new_list(T, RESULT) /* TEST PASSED */
new_list([THROW|T], RESULT) :- new_list(T, RESULT) /* TEST NOT PASSED

LET'S TRY IT WITH QUERY :- new_list([8,5,7], RESULT)

1° ITER:
UNIFIES new_list([8,5,7], RESULT) => 6 = { H/8, RESULT/[8,RES], T/[5,7] }
new_list([H|T], [H|RES])

SO NEW GOAL: 8>6 = TRUE ∧ new_list([5,7], [8,RES])

COMPUTE LENGTH OF A LIST

KB = {
length([], 0).
length([H,T], s(N)) :- length(T, N).

TIP:
1) WRITE ALWAYS THE SPECIAL CASE
2) THINK ABOUT THE SEMANTICS: "THE LENGTH OF [H|T] IS S(N) IF THE length OF T IS N"

CHECK THAT THE INPUT IS A LIST:

KB = {
list([]).
list([_|L]) :- list(L). → THIS IS LIKE DISCARD ELEMENT OF THE LIST SEARCH.

DELETE AN ELEMENT X FROM L, AND RETURN L WITHOUT IT.

KB = {
delete(X, [], []). /* BASE CASE */
delete(X, [X|T], T). /* SIMPLE CASE */
delete(X, [H|T], [H|T1]) :- delete(X, T, T1).

[5, [1,5,3], L] X/5, H/1, T/[5,3], L/[1,T2]

=> 5, [5,3], T2

X/5, T/[3], T/T2 => T2/[3]

REVERSE OF A LIST

KB =
```
reverse([], []).
reverse([H|T], LR) :- reverse(T, T2), append(T2, [H], LR).
```

PROLOG PROCEDURE TO THE QUERY ?- reverse([1,2,3], LR]

1) reverse([H|T], LR') / reverse([1,2,3], LR) → θ = {H/1, T/[2,3], LR'/LR}

BECAUSE ACTUALLY YOU SHOULD HAVE COMPARE THE QUERY WITH A FRESH VARIABLE RENAMING.

NEW GOALS: reverse([2,3], L1), append(L1, [1], LR)

2) reverse([H|T], LR'') / reverse([2,3], L1) → θ = {H/2, T/[3], LR''/L1}

NEW GOALS: reverse([3], L2), append(L2, [2], L1), append(L1, [1], LR)

3) . . . H/3, T/[]

NEW GOALS: reverse([], L3), append(L3, [3], L2), append(L2, [2], L1), append(L1, [1], LR)

4) . . . L3/[]

NEW GOALS: append([], [3], L2), append(L2, [2], L1), append(L1, [1], LR)

5) . . . L2/[3]

NEW GOALS: append([3], [2], L1), append(L1, [1], LR)

6) . . . L1/[3,2]

NEW GOALS: append([3,2], [1], LR)

7)
LR = [3,2,1]

TYPICAL PROGRAMS:

```
length([], 0).
length([_|T], s(N)) :- length(T, N).
```

```
is_member(X, [X|_]).
is_member(X, [_,T]) :- is_member(X,T).        → THIS DISCARDS THE HEAD
```

```
append([], L, L).
append([H|T], L2, [H|RES]) :- append(T, L2, RES).
```

```
is_list([]).
is_list([_,L]) :- list(L)                     → LIKE DISCARD IN is_member
```

```
delete_one(X,[],[]).
delete_one(X,[X|T],T).
delete_one(X,[H|T],[H|T1]) :- delete_one(X,T,T1).
```

```
delete_all(X,[],[]).
delete_all(X,[X|T],T1) :- delete_all(X,T,T1)
delete_all(X,[H|T],[H|T1]) :- delete_all(X,T,T1)
```

```
reverse([],[]).
reverse([H|T], LR) :- reverse(T,TLR), append(TLR,[H],LR).
```

```
pal([]).
pal(L) :- reverse(L,L)
```

```
first(X,L) :- append(X,_,L).
prefix(X,L) :- append([X],_,L).
last(X,L) :- append(_,[X],L).
suffix(X,L) :- append(_,X,L).
```

```
same_list(L,L).
...
```

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

```
split([],[],[]).
split([X|L],[X|L1],L2) :- split(L,L2,L1)
```

```
flatten([X|T],F) :- flatten(X,F1), flatten(T,F2), append(F2,F2,F).
flatten(X,[X]) :- constant(X), X =!= [].
flatten([],[]).
```

PREDICATO CUT "!"

- ESERCITA BLOCCHI DALLO STACK DI BACK TRACKING, L'EFFETTO
- È QUELLO DI FORCE PRUNING SULL'ALBERO SLD
- PUÒ ANCHE RENDERE CORRETTI ALCUNI PROGRAMMI.
- CAMBIA SEMANTICA

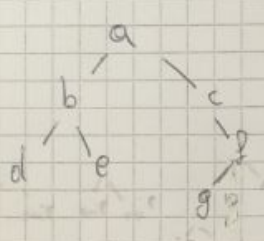
ABOUT TREES IN PROLOG: THE CONVENTION IS TO REPRESENT BINARY TREES LIKE THIS:

tree (ROOT, LEFT, RIGHT).

IN SHORT LET'S WE WRITE: t(RO, LE, RI)

THIS WILL PERMIT TO DO RECURSIVE RULES ON TREE.

EXAMPLE: TO REPRESENT THIS TREE:



~~t(a, t(b, d, e), t(c, -, t(g, g, -)))~~

t(a, t(b, t(d, nil, nil), t(e, nil, nil)), t(c, nil, t(g, t(g, nil, nil), nil)))

YOU MUST USE nil TO SPECIFY EACH LEAVE WITH NO SON!!

EXERCISES:

CHECK IF IT IS A TREE:

IMAGINE THIS QUERY: is_tree(t(a, t(b, nil, nil), nil)).

```
is_tree(nil).
is_tree(_, L, R) :- is_tree(L), is_tree(R).
```

COUNT LEAVES OF A BINARY TREE:

```
count_leaves(nil, 0).
count_leaves((_, nil, nil), 1).
count_leaves((_, L, R), SUM) :- count_leaves(L, SUML), count_leaves(R, SUMR),
                                SUM is SUML+SUMR.
```

SUM OF NODES VALUES:

```
sum_nodes(nil, 0).
sum_nodes(tree(Root, Left, Right), S) :- sum_nodes(Left, SL),
                                         sum_nodes(Right, SR),
                                         S is SL+SR+Root.
```