# Programming languages

## Programming in PROLOG

***Program Execution and Recursive Programming***

# How does PROLOG work?

- In Prolog there are NO control statements.
- Its execution is based on two concepts:
  - unification and
  - backtracking.
- Thanks to unification, each goal determines a subset of clauses that can be executed.
- Each of these is called *a choice point*.

# Backtracking

- Prolog selects the first choice point and continues executing the program until it determines whether the goal is true or false.
- In case a choice point is false, **backtracking comes into play** .
- Backtracking consists of undoing everything executed, placing the program in the same state it was in just before reaching the choice point.
- Then the next pending choice point is taken and the process is repeated again.

# How does PROLOG work?

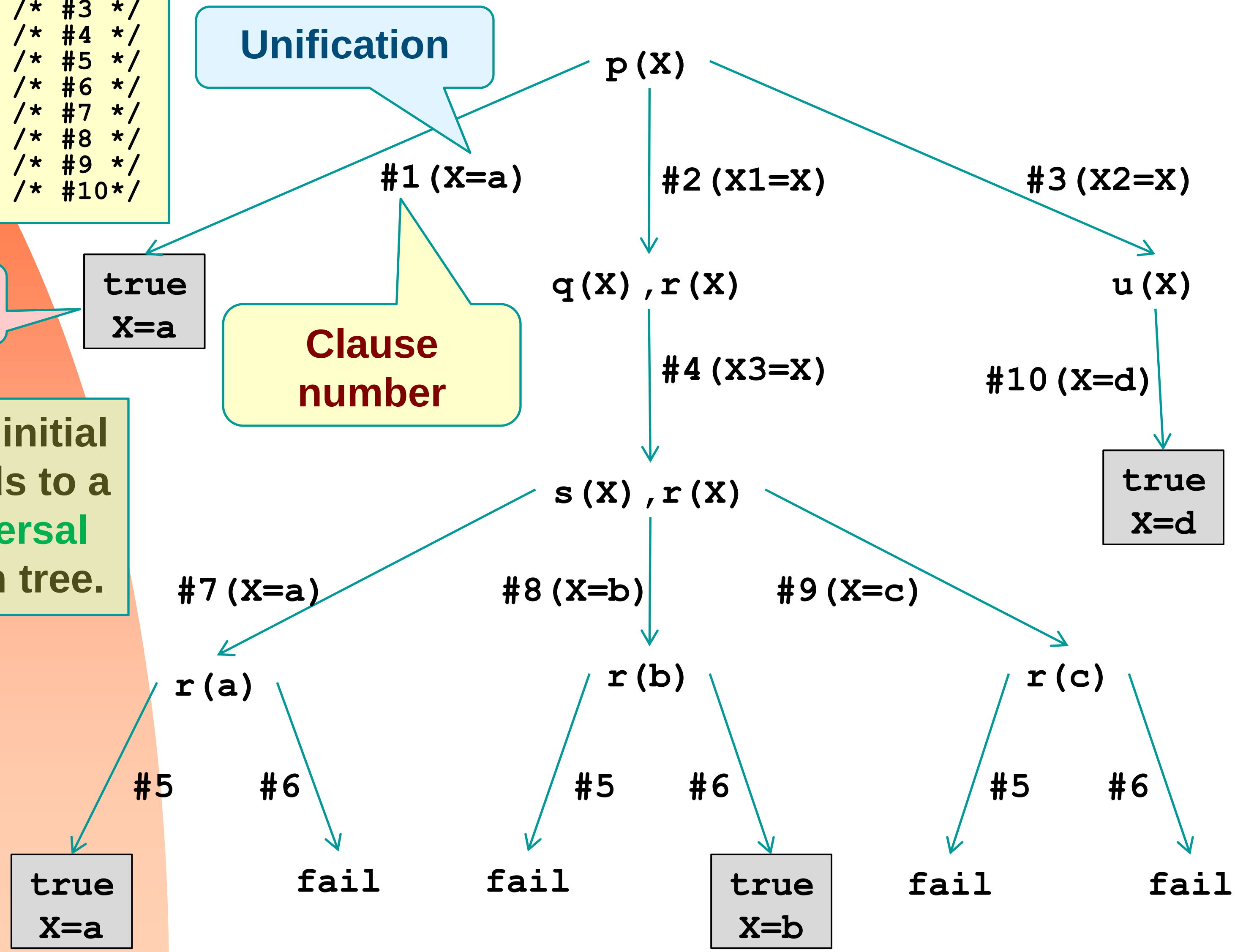
**To illustrate how Prolog obtains the responses for programs and goals, consider the following program.**

```
/* program P */
p(a).                  /* #1 */
p(X1):-q(X1),r(X1).   /* #2 */
p(X2):-u(X2).          /* #3 */
q(X3):-s(X3).          /* #4 */
r(a).                  /* #5 */
r(b).                  /* #6 */
s(a).                  /* #7 */
s(b).                  /* #8 */
s(c).                  /* #9 */
u(d).                  /* #10*/
```

- **Exercise1** :
  1. Load program P into Prolog
  2. Notice what happens for the goal: ?- p(X).
  3. Wear ; to display all answers.
- **Exercise2** :
  1. Load program P into Prolog
  2. Activate the trace with trace.
  3. Notice what happens for the goal: ?- p(X).
  4. Wear ; to display all answers.

# Execution for p(X).

```
/* program P */
p(a).                   /* #1 */
p(X1):-q(X1),r(X1).     /* #2 */
p(X2):-u(X2).           /* #3 */
q(X3):-s(X3).           /* #4 */
r(a).                   /* #5 */
r(b).                   /* #6 */
s(a).                   /* #7 */
s(b).                   /* #8 */
s(c).                   /* #9 */
u(d).                   /* #10*/
```




The trace of the initial goal corresponds to a **depth-first traversal** of the derivation tree.

# Programming in Prolog

- RELATIONAL operators:
  - same as **=:=**
  - different **=\=**
  - less than **<**
  - greater than **>**
  - less than **=<**
  - greater than equal to **>=**
- They form clauses in infix format and by themselves they return a truth value.

# Example

```
governed(epn, 2012, 2018).
governed(fch, 2006, 2012).
governed(vfq, 2000, 2006).
governed(ezp, 1994, 2000).
governed(csg, 1988, 1994).
governed(mmh, 1982, 1988).
governed(jlp, 1976, 1982).
governed(lea, 1970, 1976).

was_president(Person, Year):-
  governed(Person, Start, End),
  Year >= Start, Year =< End.
```

# Programming in Prolog

- ARITHMETIC operators:
  - sum **+**
  - subtraction **-**
  - multiplication *****
  - division **/**
  - remainder **mod**
- They conform clauses in infix format whose result must be instantiated to a variable through the **is** operator.

# Operator **is**

- It is a requirement to use it when an arithmetic evaluation is required.
- Format:

  *Variable* ***is*** *arithmetic_expression*
- Instantiate the variable with the result of the expression, and the clause is **TRUE** by default.

# Example

- Suppose you have FACTS defined for:

```
population(Country, Amount).
area(Country, Space).
```

```
density(Country, D) :-
     population(Country, P),
     area(Country, A),
     D is P/A.
```

# Rules as modules

- Under a modular abstraction approach, the rules are MODULES, and the rule variables are input and/or output parameters, depending on the case.

*Input* *Output*

```
density(Country, D) :-
  population(Country, P),
  area(Country, A), D is P/A.
```

# Control Mechanisms

- **There is no iteration!**
- ...although something similar can be simulated like this:

```
test :-
  governed(P,_,_),
  write(P),nl,
  fail.
test.

?- test.
   epn
   fch
   ...
```

Solution builder

loop body

test (in this case, it always causes backtracking!)

# Alternative: recursive rules

- Rules whose body, have terms that correspond to the head of the rule itself.

- Example:

```
grandparent(X,Y) :- father(X, Z),
   father(Z,Y).
great-grandfather(X,Y) :- father(X, Z),
   grandfather(Z,Y).
great-great-grandfather(X,Y):-
   father(X, Z),
   great-grandfather(Z,Y).
…
ancestor(X,Y) :- father(X,Y).
ancestor(X,Y) :- father(X,Z),
                ancestor(Z, Y).
```

# Recursive thinking

- It is applied in the same way.
- The implementation involves having at least one rule for the base case, and at least one recursive rule (which calls itself).
- The decision to evaluate one case or the other is implicit in the way the interpreter works.

# Example

- Factorial of a number.
- Relation between a number and its factorial.
- BASE CASE : **0! = 1**

```
factorial(0, 1).
```

- GENERAL CASE : **n! = n * (n-1)!**

```
factorial(N, R) :-
          X is N-1,
          factorial(X, W),
          R is N*W.
```

# Common mistakes

$$x * y = \begin{cases} 0 & \text{if } x = 0 \\ (x-1) * y + y & \text{if } x > 0 \end{cases}$$

```
product(0,X,0).
product(X,Y,W+Y) :- X>0, product(X-1,Y,W).
```

nested operations

```
product(0,X,0).
product(X,Y,Z) :- X>0, product(X-1,Y,W), Z is W+Y.
```

nested operations

```
product(0,X,0).
product(X,Y,Z) :- X>0, X is X-1, product(X,Y,W), Z is W+Y.
```

destructive assignment

**Correct solution:**

```
product(0,X,0).
product(X,Y,Z) :- X>0, N is X-1, product(N,Y,W), Z is W+Y.
```

# Formatted writing

- **`write()`** predicate that prints its argument (only one):

```
write('Hello world').
myhello(X) :- write('Hello '),
  write(X), write('!').
```

- Line breaks are with “ **`nl`** ”:

```
write('one line'),nl,
write('and another line'),nl.
```

# Data reading

- Prolog can read data from files or from the terminal using the **`read(X)`** predicate.
- **Example** : Display the average of a given student.

```
avgread :-
   write('Student id? '), nl,
   read(stdId),
   promalum(stdId,Avg),
   calif(stdId,Name,_),
   write('The average of student '),
   write(Name), write(' is '),
   write(Avg), nl.
```

# Use of the fail clause

- It makes a goal fail.
- It is useful to force the system to deliver all results.
- **Example** : List the averages of all the students.

```
stdAvg :-
    grade(_,Name,part(One,Two,Three)),
    Avg is (One+Two+Three)/3,
    write('The average of student '),
    write(Name), write(' is '),
    write(Avg), nl,
    fail.
```