N7PD Declarative Programming

Logic Programming – Prolog

Christophe Garion
ISAE-SUPAERO/DISC

License CC BY-NC-SA 3.0

What is logic programming?

There are several programming paradigms: imperative, functional, procedural, object-oriented etc.

What is logic programming?

There are several programming paradigms: imperative, functional, procedural, object-oriented etc.

Logic programming is a programming paradigm with the following properties:

1. based on **mathematical logic**
2. programs contain **facts and rules**
3. computation is seen as **automated reasoning** over the facts and rules

What is logic programming?

There are several programming paradigms: imperative, functional, procedural, object-oriented etc.

Logic programming is a programming paradigm with the following properties:

1. based on **mathematical logic**
2. programs contain **facts and rules**
3. computation is seen as **automated reasoning** over the facts and rules

Points 1 and 2 implies that we will manipulate **predicates**, i.e. expressions that have a truth value.

$Integer(one)$	*one* is an integer
$Plays(john, soccer, isae)$	John plays soccer for the ISAE team

What is logic programming?

There are several programming paradigms: imperative, functional, procedural, object-oriented etc.

Logic programming is a programming paradigm with the following properties:

1. based on **mathematical logic**
2. programs contain **facts and rules**
3. computation is seen as **automated reasoning** over the facts and rules

Points 1 and 2 implies that we will manipulate **predicates**, i.e. expressions that have a truth value.

$Integer(one)$	*one* is an integer
$Plays(john, soccer, isae)$	John plays soccer for the ISAE team

Point 3 implies that we will **ask questions to programs**, e.g. "who plays for the ISAE soccer team?"

Predicates, terms and functions

Predicates represent **properties** verified by **objects**.

For instance, *Plays*$/3$ is a predicate of arity 3 representing the fact that somebody (the first argument of the predicate) plays a sport (the second argument of the predicate) in a team (the third argument).

Predicates, terms and functions

Predicates represent **properties** verified by **objects**.

For instance, *Plays*$/3$ is a predicate of arity 3 representing the fact that somebody (the first argument of the predicate) plays a sport (the second argument of the predicate) in a team (the third argument).

In first-order logic, **objects** are represented by **terms**.

Terms can be:

- **constants**, e.g. $\mathit{Plays}(\mathit{john}, \mathit{soccer}, \mathit{isae})$
- **functions applied to terms**, e.g. $\mathit{Plays}(\mathit{father}(\mathit{john}), \mathit{soccer}, \mathit{isae})$
- **variables**, e.g. $\forall x \; \mathit{Plays}(x, \mathit{soccer}, \mathit{isae})$

Going back to natural numbers (again)

For instance, let us consider natural numbers. We will define them by:

- a **constant term** z
- an **unary function** s (successor)
- a **predicate** $=$ (in infix notation)

$s(s(z))$ represents the successor of the successor of z (2 in real life) and $s(z) = z$ should be false.

Going back to natural numbers (again)

For instance, let us consider natural numbers. We will define them by:

- a **constant term** z
- an **unary function** s (successor)
- a **predicate** $=$ (in infix notation)

$s(s(z))$ represents the successor of the successor of z (2 in real life) and $s(z) = z$ should be false.

Exercise

We want to define addition. What will you choose, a function, a constant, a predicate?

Going back to natural numbers (again)

For instance, let us consider natural numbers. We will define them by:

- a **constant term** z
- an **unary function** s (successor)
- a **predicate** $=$ (in infix notation)

$s(s(z))$ represents the successor of the successor of z (2 in real life) and $s(z) = z$ should be false.

Exercise

We want to define addition. What will you choose, a function, a constant, a predicate?

Exercise

We want now to define a **predicate** *add* representing addition. What is its arity?

Addition of two natural numbers

We can define addition with a predicate *add/3* and the following fact and rule:

1. $\forall x \; add(z, x, x)$
2. $\forall x \forall y \forall r \; add(x, y, r) \rightarrow add(s(x), y, s(r))$

Using this theory, we can exhibit a **proof** of $add(s(s(z)), s(s(z)), s(s(s(s(z)))))$ in a particular **formal system**.

$$\dfrac{\dfrac{add(z, s(s(z)), s(s(z))) \qquad add(x, y, r) \rightarrow add(s(x), y, s(r))}{add(s(z), s(s(z)), s(s(s(z))))} \qquad add(x, y, r) \rightarrow add(s(x), y, s(r))}{add(s(s(z)), s(s(z)), s(s(s(s(z)))))}$$

Yes, $2 + 2 = 4$ ☺

What is Prolog?

Prolog (*Programmation en logique*) is a logic programming language initially developed in the 70s for natural language processing.

In Prolog, rules can only have the following form (they are called Horn clauses)

$$B_1 \wedge \ldots \wedge B_n \rightarrow A$$

i.e. "B_1 and …and B_n implies A".

What is Prolog?

Prolog (*Programmation en logique*) is a logic programming language initially developed in the 70s for natural language processing.

In Prolog, rules can only have the following form (they are called Horn clauses)

$$B_1 \wedge \ldots \wedge B_n \rightarrow A$$

i.e. "B_1 and …and B_n implies A".

Prolog is both

- **declarative**: you describe the problem to solve
- **procedural**: there is a procedure that explains how Prolog answers questions

Prolog basic syntax and principles

Instructions in a Prolog program can be viewed as the **premises** of an argument.

A request can be viewed as the **conclusion** of an argument from the previous premises.

The fact that the conclusion can be deduced from the premises is proved by using the **Resolution** formal system for First-Order Logic (FOL).
Variables are used and instanciated.

Prolog basic syntax and principles

Instructions in a Prolog program can be viewed as the **premises** of an argument.

A request can be viewed as the **conclusion** of an argument from the previous premises.

The fact that the conclusion can be deduced from the premises is proved by using the **Resolution** formal system for First-Order Logic (FOL). **Variables** are used and instanciated.

Syntax (Prolog)

- representing data: using **terms**
- identifier beginning by a **lowercase** letter: function or predicate symbol
- identifier beginning by a **uppercase** letter: variable symbol

Prolog: back to addition

The Prolog program corresponding to our “definition” of addition:

add.pl

```
add(z, X, X).
add(s(X), Y, s(R)) :- add(X, Y, R).
```

Prolog: back to addition

The Prolog program corresponding to our "definition" of addition:

add.pl

```
add(z, X, X).
add(s(X), Y, s(R)) :- add(X, Y, R).
```

We can "ask" Prolog questions: `add(s(s(z)),s(s(s(z))),W)`?

- ➥ meaning: is there a `W` such that `add(s(s(z)),s(s(s(z))),W)` holds, i.e. `W = 2 + 3`?

In this case, Prolog answers `W = s(s(s(s(s(z)))))`.

Cool. Prolog knows how to add two natural numbers.

Prolog applications

Is Prolog really used in real life?

- NASA Clarissa
  - voice-operated procedure browser in International Space Station
  - help astronauts with complex procedures
- IBM Watson
  - a question-answering computer system
  - won the quiz show Jeopardy in 2011

Lally, Adam and Paul Fodor (Mar. 2011).
Natural Language Processing With Prolog in the IBM Watson System.
https://www.cs.nmsu.edu/ALP/2011/03/natural-language-processing-with-prolog-in-the-ibm-watson-system/.

Outline

Program clause

Definition (Prolog program)

A Prolog program is a **sequence** of clauses.

Definition (Prolog program)

A Prolog program is a **sequence** of clauses.

Syntax (Program clause)

$$A \ \texttt{:-} \ B_1, \ldots, B_n \ \texttt{.}$$

Program clause

Definition (Prolog program)

A Prolog program is a **sequence** of clauses.

Syntax (Program clause)

$$\underbrace{\boxed{A}}_{\text{head}} \;\texttt{:-}\; \underbrace{\boxed{B_1, \ldots, B_n}}_{\text{body}}\;.$$

Program clause

Definition (Prolog program)

A Prolog program is a **sequence** of clauses.

Syntax (Program clause)

$$\underbrace{A}_{\text{head}} \;\texttt{:-}\; \underbrace{B_1, \ldots, B_n}_{\text{body}}\;\texttt{.}$$

Intuition: for all possible values for variables, if $B_1, \ldots, B_n$ are all true, then A is true.

Program clause

Definition (Prolog program)

A Prolog program is a **sequence** of clauses.

Syntax (Program clause)

$$\underbrace{\boxed{A}}_{\text{head}} \;\texttt{:-}\; \underbrace{\boxed{B_1, \ldots, B_n}}_{\text{body}}\;\textbf{.}$$

Intuition: for all possible values for variables, if $B_1, \ldots, B_n$ are all true, then A is true.

If $n = 0$, then the clause is a **fact** and is simply denoted by A**.**

Definition (Prolog program)

A Prolog program is a **sequence** of clauses.

Syntax (Program clause)

$$\underbrace{A}_{\text{head}} \;\texttt{:-}\; \underbrace{B_1, \ldots, B_n}_{\text{body}}.$$

beware of us!

Intuition: for all possible values for variables, if $B_1, \ldots, B_n$ are all true, then A is true.

If $n = 0$, then the clause is a **fact** and is simply denoted by A.

Query clause

Syntax (Query clause)

$$:\text{-}\, B_1, \ldots, B_n\,.$$

Query clause

Syntax (Query clause)

$$\texttt{:-}B_1, \ldots, B_n\texttt{.}$$

Intuition: are there values for variables such that $B_1, \ldots, B_n$ are **all** true?

Query clause

Syntax (Query clause)

$$:\!-B_1, \ldots, B_n.$$

Intuition: are there values for variables such that $B_1, \ldots, B_n$ are **all** true?

Beware

For the program clause $B(X)$:- $C(Y, X)$.

- X is **universally** quantified
- Y is **existentially** quantified

Terminology: Prolog resolvent

Definition (Most general unifier)

If $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$ are two atoms such thath the subsitution σ is a most general unifier of those atoms, then $p(t_1, \ldots, t_n)$ and $p(s_1, \ldots, s_n)$ are unifiable by **mgu** σ.

Definition (Prolog resolvent)

Let $R = \texttt{:-}\ A_1, \ldots, A_m$ be a query clause and $C = A'_1\ \texttt{:-}\ B_1, \ldots, B_p$ be a program clause with $m > 0$ and $p \geq 0$. If A_1 and A'_1 are unifiable by σ, then the new query clause $R' = \texttt{:-}\ \sigma(B_1, \ldots, B_p, A_2, \ldots, A_m)$ is called **Prolog resolvent** of R and C.

Idea behind Prolog resolvent

The idea behind Prolog resolvent is the same as behind Resolution in First-Order Logic (cf. last slides in this part).

More intuitively, consider a program clause $A' \texttt{:-} B_1, \ldots, B_n$. It can be read as “if $B_1, \ldots, B_n$ hold, then A' holds”...

Idea behind Prolog resolvent

The idea behind Prolog resolvent is the same as behind Resolution in First-Order Logic (cf. last slides in this part).

More intuitively, consider a program clause $A' \texttt{:-} B_1, \ldots, B_n$. It can be read as “if $B_1, \ldots, B_n$ hold, then A' holds”...
...but you can also understand the clause as “to prove A', it is sufficient to prove $B_1, \ldots, B_n$”.

Idea behind Prolog resolvent

The idea behind Prolog resolvent is the same as behind Resolution in First-Order Logic (cf. last slides in this part).

More intuitively, consider a program clause A' `:-` $B_1, \ldots, B_n$. It can be read as "if $B_1, \ldots, B_n$ hold, then A' holds"...
...but you can also understand the clause as "to prove A', it is sufficient to prove $B_1, \ldots, B_n$".

Thus, when you have a request involving A such that A and A' are unifiable, you can replace the A part of the request by $B_1, \ldots, B_n$ (given the substitution).

Idea behind Prolog resolvent

The idea behind Prolog resolvent is the same as behind Resolution in First-Order Logic (cf. last slides in this part).

More intuitively, consider a program clause $A' \texttt{:-} B_1, \ldots, B_n$. It can be read as "if $B_1, \ldots, B_n$ hold, then A' holds"...
...but you can also understand the clause as "to prove A', it is sufficient to prove $B_1, \ldots, B_n$".

Thus, when you have a request involving A such that A and A' are unifiable, you can replace the A part of the request by $B_1, \ldots, B_n$ (given the substitution).

Does it end? It depends (cf. next slides), but facts can be used, e.g.:

clause	`add(X, z, X).`
request	`add(s(s(z)), z, W).`
resolvent	empty clause with substitution {`X/s(s(z))`, `W/X`}

Evaluation algorithm

There is a **non-determinist** evaluation algorithm for Prolog.

Inputs: a Prolog program P and a query clause R

Output: two possibilities

- a substitution σ for the variables appearing in R (if R does not contain variables, the output is `YES`);
- `NO`

Evaluation algorithm

There is a **non-determinist** evaluation algorithm for Prolog.

Inputs: a Prolog program P and a query clause R

Output: two possibilities

- a substitution σ for the variables appearing in R (if R does not contain variables, the output is `YES`);
- `NO`

Given a Prolog program P and a query R, an evaluation of R can have three issues:

- ending with success
- ending with `NO`
- **no ending!**

Algorithm

Algorithm 1.1: Prolog evaluator

Input: a Prolog program P and a query R
Output: a substitution σ for variables appearing in R, else NO

$R_c \leftarrow R$;
$mgu_c \leftarrow \emptyset$;
while $R_c =$:- $G_1, \ldots, G_k \neq \emptyset$ **do**
    choose $C = G'_1$:- $D_1, \ldots D_t \in P$ st G_1 et G'_1 unifiable by σ ;
    if C does not exist **then**
        break ;
    end
    $R_c \leftarrow$ Prolog resolvent of G_1 and C (replace G_1) ;
    $mgu_c \leftarrow mgu_c \circ \sigma$;
end
if $R_c = \emptyset$ **then**
    compute restriction σ' of mgu_c to R variables ;
    if $\sigma' = \emptyset$ **then**
        return YES ;
    else
        return σ' ;
    end
else
    return No ;
end

Outline

Example

Program

```
1 parent(jack,mary).
2 parent(louise,jack).
3 parent(franck,john).
4 ancestor(X,Y) :- parent(X,Y).
5 ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
```

Example

Program

```
1 parent(jack,mary).
2 parent(louise,jack).
3 parent(franck,john).
4 ancestor(X,Y) :- parent(X,Y).
5 ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).
```

Query

```
:- ancestor(W,mary)
```

Example

Program

1. `parent(jack,mary).`
2. `parent(louise,jack).`
3. `parent(franck,john).`
4. `ancestor(X,Y) :- parent(X,Y).`
5. `ancestor(X,Y) :- ancestor(X,Z), parent(Z,Y).`

Query

```
:- ancestor(W,mary)
```

Some evaluation **examples** are presented in the next slides.

Example 1: evaluation with success

$\sigma = \emptyset$

```
ancestor(W,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 1: evaluation with success

$\sigma = \{W/X1, Y1/mary\}$

```
ancestor(W,mary)
       |
parent(X1,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 1: evaluation with success

$$\sigma = \{W/X1, Y1/mary, X1/jack\}$$

```
ancestor(W,mary)
       |
parent(X1,mary)
       |
       ∅
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 1: evaluation with success

$\sigma = \{W/X1, Y1/mary, X1/jack\ \}$

```
ancestor(W,mary)
       |
parent(X1,mary)
       |
       ∅
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Answer: $\{W/jack\}$

Example 2: evaluation with success

$\sigma = \emptyset$

```
ancestor(W,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 2: evaluation with success

$$\sigma = \{W/X1, Y1/mary\}$$

```
      ancestor(W,mary)
              |
ancestor(X1,Z1),parent(Z1,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 2: evaluation with success

$$\sigma = \{W/X1, Y1/mary, X1/X2, Z1/Y2\}$$

```
      ancestor(W,mary)
              |
ancestor(X1,Z1),parent(Z1,mary)
              |
 parent(X2,Y2),parent(Y2,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
```
`ancestor(X,Y) :- parent(X,Y).`
```
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 2: evaluation with success

$$\sigma = \{W/X1, Y1/mary, X1/X2, Z1/Y2, X2/louise, Y2/jack\}$$

```
ancestor(W,mary)
        |
ancestor(X1,Z1),parent(Z1,mary)
        |
parent(X2,Y2),parent(Y2,mary)
        |
parent(jack,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 2: evaluation with success

$$\sigma = \{W/X1, Y1/mary, X1/X2, Z1/Y2, X2/louise, Y2/jack\}$$

```
ancestor(W,mary)
        |
ancestor(X1,Z1),parent(Z1,mary)
        |
parent(X2,Y2),parent(Y2,mary)
        |
parent(jack,mary)
        |
```
$\emptyset$

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 2: evaluation with success

$$\sigma = \{W/X1, Y1/mary, X1/X2, Z1/Y2, X2/louise, Y2/jack\}$$

```
ancestor(W,mary)
        |
ancestor(X1,Z1),parent(Z1,mary)
        |
parent(X2,Y2),parent(Y2,mary)
        |
parent(jack,mary)
        |
        ∅
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Answer: $\{W/louise\}$

Example 3: evaluation with failure

$\sigma = \emptyset$

```
ancestor(W,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 3: evaluation with failure

$$\sigma = \{W/X1, Y1/mary\}$$

```
ancestor(W,mary)
        |
ancestor(X1,Z1),parent(Z1,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 3: evaluation with failure

$$\sigma = \{W/X1, Y1/mary, X1/X2, Z1/Y2\}$$

```
      ancestor(W,mary)
              |
ancestor(X1,Z1),parent(Z1,mary)
              |
 parent(X2,Y2),parent(Y2,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 3: evaluation with failure

$$\sigma = \{W/X1, Y1/\mathit{mary}, X1/X2, Z1/Y2, X2/\mathit{franck}, Y2/\mathit{john}\}$$

```
ancestor(W,mary)
        |
ancestor(X1,Z1),parent(Z1,mary)
        |
parent(X2,Y2),parent(Y2,mary)
        |
parent(john,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Example 3: evaluation with failure

$$\sigma = \{W/X1, Y1/mary, X1/X2, Z1/Y2, X2/franck, Y2/john\}$$

```
ancestor(W,mary)
        |
ancestor(X1,Z1),parent(Z1,mary)
        |
parent(X2,Y2),parent(Y2,mary)
        |
parent(john,mary)
```

```
parent(jack,mary).
parent(louise,jack).
parent(franck,john).
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- ancestor(X,Z),
                 parent(Z,Y).
```

Answer: `NO`

Example: evaluation with no ending

Using only clause `ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).`

Example: evaluation with no ending

Using only clause `ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).`

```
ancestor(W,mary)
```

Example: evaluation with no ending

Using only clause `ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).`

```
            ancestor(W,mary)
                   |
ancestor(X1,Z1),parent(Z1,mary)
```

Example: evaluation with no ending

Using only clause `ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).`

```
                 ancestor(W,mary)
                        |
        ancestor(X1,Z1),parent(Z1,mary)
                        |
ancestor(X2,Z2),parent(Z2,Y2),parent(Y2,mary)
```

Example: evaluation with no ending

Using only clause `ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).`

```
              ancestor(W,mary)
                     |
      ancestor(X1,Z1),parent(Z1,mary)
                     |
ancestor(X2,Z2),parent(Z2,Y2),parent(Y2,mary)
                     |
                     ⋮
```

Example: evaluation with no ending

Using only clause `ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).`

```
ancestor(W,mary)

ancestor(X1,Z1),parent(Z1,mary)

ancestor(X2,Z2),parent(Z2,Y2),parent(Y2,mary)
⋮
⋮
```

Example: evaluation with no ending

Using only clause `ancestor(X,Y) :- ancestor(X,Z),parent(Z,Y).`

```
                 ancestor(W,mary)
                        |
        ancestor(X1,Z1),parent(Z1,mary)
                        |
ancestor(X2,Z2),parent(Z2,Y2),parent(Y2,mary)
                        |
                        ⋮
                        |
                        ⋮
```

No ending!

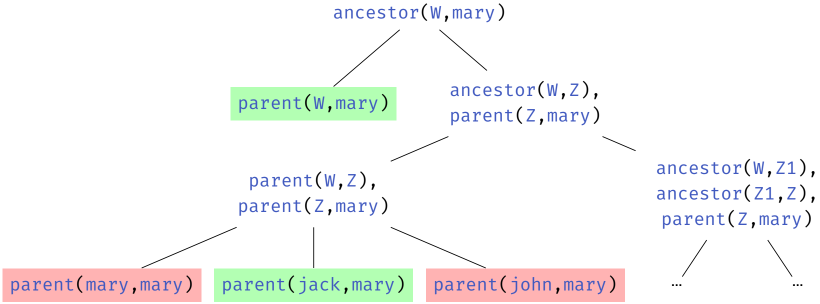
Outline

Definition (Prolog search tree)

A Prolog search tree for a program P and a query R is a tree whose nodes are query clauses and such that:

- its **root** is R;
- if a node is a non-empty query `:-` $A_1, \ldots, A_n$ (where $n > 0$) and $C_1, \ldots, C_k$ (where $k > 0$) are the program clauses (**appearing in this order** in P) whose heads are unifiable with A_1, then this node has k children $Res_1, \ldots, Res_k$ where for $i \in \{1, \ldots, k\}$, Res_i **is the Prolog resolvent** of A_1 with the clause C_i;
- if a node is a non-empty query `:-` $A_1, \ldots, A_n$ (where $n > 0$) and if there is no program clause whose head is unifiable with A_1, then this node is a **failure leaf**;
- if a node is the empty query, then this node is a **success leaf**.

Prolog search tree for our example (simplified)



The Prolog tree on our example is **infinite on the right**.

Depth-first exploration

- if the current branch ends with success, the evaluation stops and give the corresponding answer;
- if the current branch ends with failure, the next branch is considered. The next clause usable for the query represented by the parent node of the current node is chosen (**backtracking**);
- if after having given the result of a success branch the user send a “continue” instruction, the next branch is considered as if the current branch had failed (**backtracking**);
- if the branch does not end, the evaluator does not stop.

Thus...

- the answers to the query R are given into an order that depends of the writing order of clauses and atoms in P;
- infinite branch $\Rightarrow$ the interpreter does not stop;
- the answer `NO` corresponds to the case where the tree is **finite** and where every branch is a failure branch.

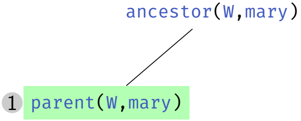
Prolog search tree building on our example (simplified)

`ancestor(W,mary)`

- - - - - - - - - → user backtracking after success

- - - - - - - - - → Prolog backtracking after failure

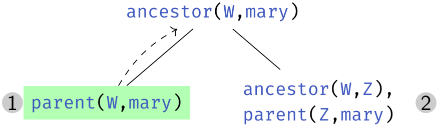
Prolog search tree building on our example (simplified)



- - - - - - - - -> user backtracking after success

- - - - - - - - -> Prolog backtracking after failure

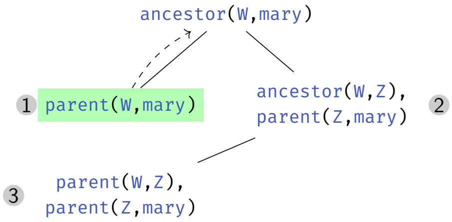
Prolog search tree building on our example (simplified)



user backtracking after success

Prolog backtracking after failure

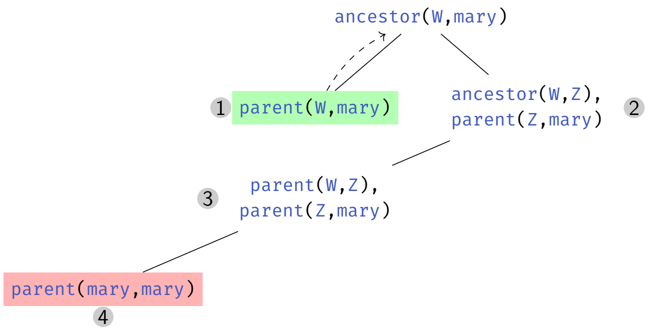
Prolog search tree building on our example (simplified)

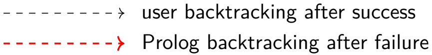


- - - - - - - - -> user backtracking after success

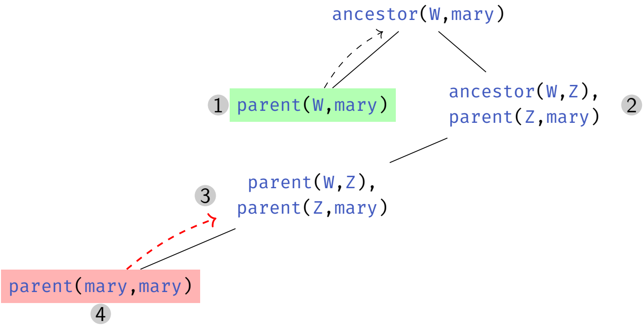
- - - - - - - - -> Prolog backtracking after failure

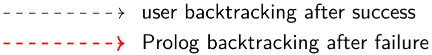
Prolog search tree building on our example (simplified)



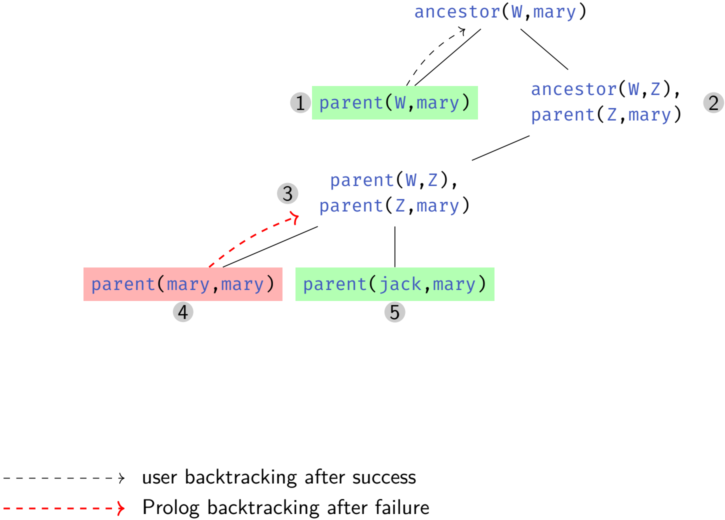


Prolog search tree building on our example (simplified)

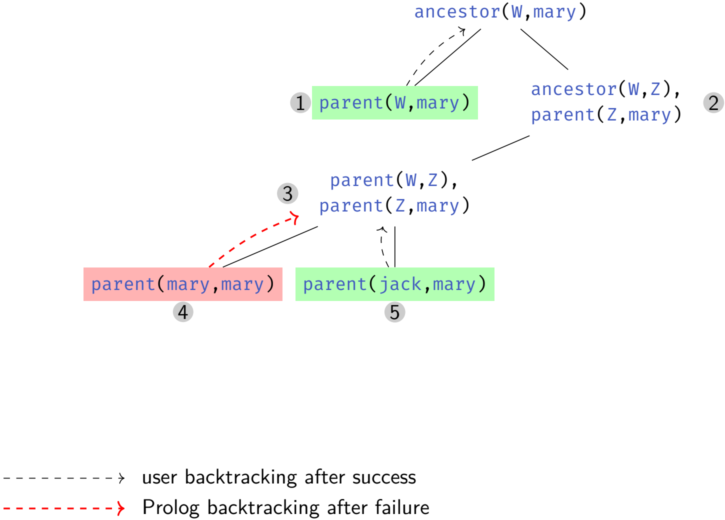




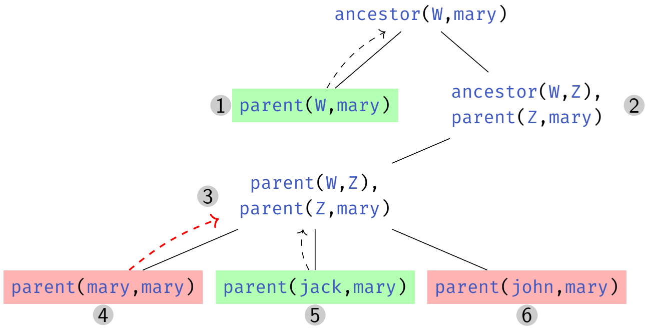
Prolog search tree building on our example (simplified)



Prolog search tree building on our example (simplified)



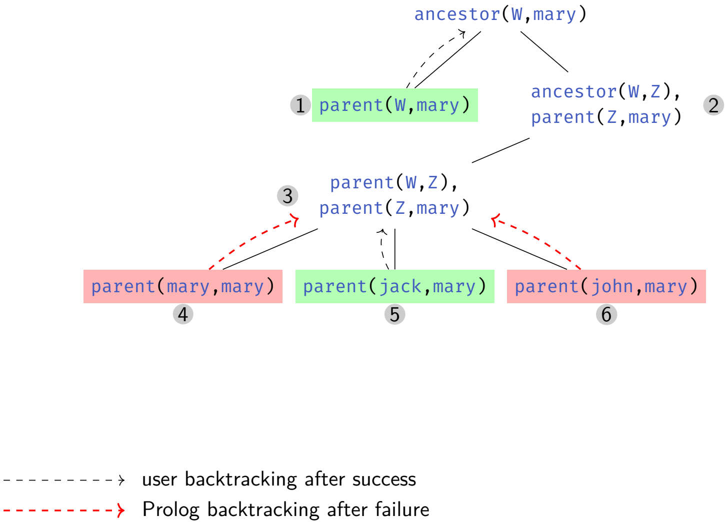
Prolog search tree building on our example (simplified)



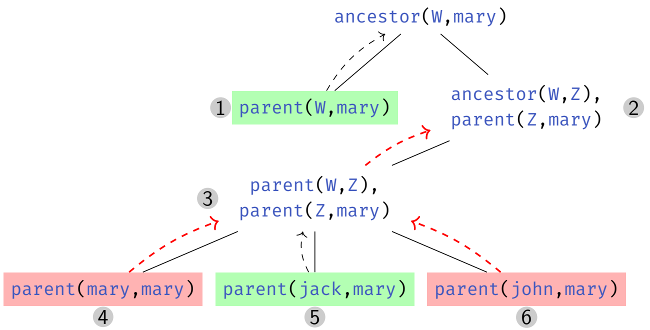
- - - - - - - - -› user backtracking after success

- - - - - - - - -› Prolog backtracking after failure

Prolog search tree building on our example (simplified)



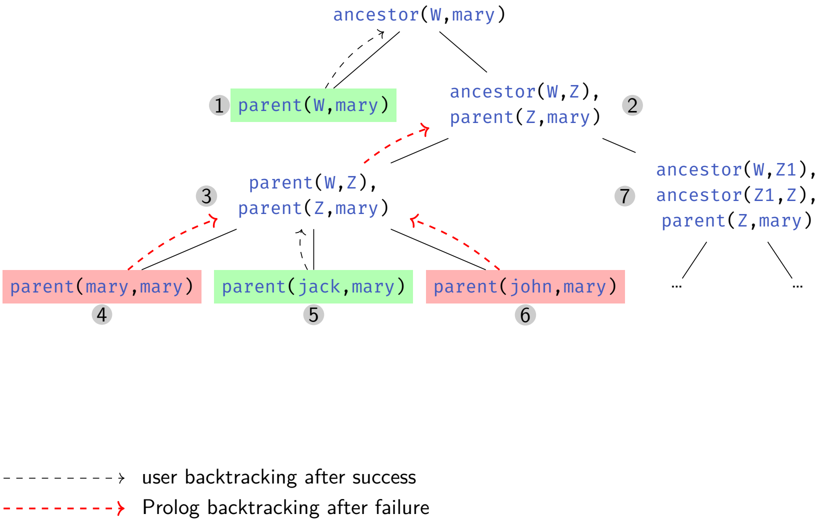
Prolog search tree building on our example (simplified)



user backtracking after success

Prolog backtracking after failure

Prolog search tree building on our example (simplified)



Atoms order is important!

Theorem

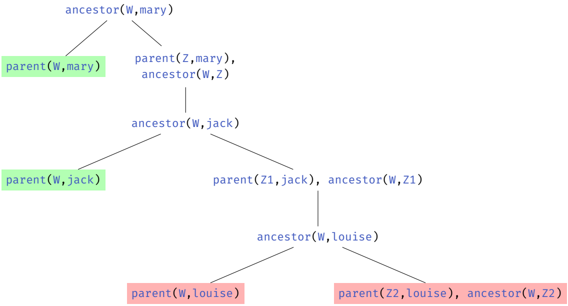
*The order of **atoms** in the clauses bodies determine the structure of the search tree.*

Atoms order is important!

Theorem

*The order of **atoms** in the clauses bodies determine the structure of the search tree.*

For instance, replace in the previous program the clause 5 by
`ancestor(X,Y) :- parent(Z,Y),ancestor(X,Z).`



Clauses order is important!

Theorem

*The order of the **clauses** in the program is important:*

- *the answers order can change*
- *the Prolog interpretor can loop*

Clauses order is important!

Theorem

*The order of the **clauses** in the program is important:*

- *the answers order can change*
- *the Prolog interpreter can loop*

For instance, swap clauses 4 and 5...

```
ancestor(W,mary)
        |
ancestor(W,Z1), parent(Z1,mary)
        |
ancestor(W,Z2), parent(Z2,Z1), parent(Z1,mary)
        |
       ...
```

Outline

Using negation

Let us suppose that we want to express the following fact: "Christophe likes all programming languages except ugly languages like MATLAB or Visual Basic".

We can write

```
likes(christophe, X) :- prog_language(X).
```

but we must exclude ugly languages...

Negation as failure

The previous situation is so common that a predicate **`not`** has been defined.
The **`not`** operator is also written `\+`. It is called **negation as failure**.

Negation as failure

The previous situation is so common that a predicate **not** has been defined.
The **not** operator is also written \+. It is called **negation as failure**.

We could have rewritten the previous program as

```
likes(christophe, X) :- \+ ugly_language(X),
                        prog_language(X).
```

The problem with negation as failure

Consider the following program

```
smart(albert).
smart(edsger).
```

and ask the following question: `\+ smart(christophe).`

Does it mean that Christophe is not smart (maybe)?

The problem with negation as failure

Consider the following program

```
smart(albert).
smart(edsger).
```

and ask the following question: `\+ smart(christophe).`

Does it mean that Christophe is not smart (maybe)?

The meaning of Prolog answer to the question is the following: given the program, there is not enough information to prove that Christophe is smart.

This is called the **closed world assumption** (CWA): everything that is true can be derived from the program.

Negation as failure: example

Let us consider the following program to choose the university you want to attend a MSc in:

```
good_standard(mit).
good_standard(berlin).

expensive(mit).

reasonable(X) :- \+ expensive(X).
```

Ask the following questions. What happens?

- `good_standard(X), reasonable(X).`
- `reasonable(X), good_standard(X).`

Negation as failure: example

Let us consider the following program to choose the university you want to attend a MSc in:

```
good_standard(mit).
good_standard(berlin).

expensive(mit).

reasonable(X) :- \+ expensive(X).
```

Ask the following questions. What happens?

- `good_standard(X), reasonable(X).`
- `reasonable(X), good_standard(X).`

Important

`\+ expensive(X)` in the request means "**for all X**, not `expensive(X)`".

Outline

Prolog and Resolution?

The formula associated to $A_1 :- B_1, \ldots, B_m$ with variables $X_1, \ldots, X_n$ is $\forall x_1 \ldots \forall x_n \; ((B_1 \land \ldots \land B_m) \rightarrow A_1)$, thus $\neg B_1 \lor \ldots \lor \neg B_m \lor A_1$.

NB: there is only one positive literal in the clause.

Prolog and Resolution?

The formula associated to $A_1 :- B_1, \ldots, B_m$ with variables $X_1, \ldots, X_n$ is $\forall x_1 \ldots \forall x_n \; ((B_1 \wedge \ldots \wedge B_m) \rightarrow A_1)$, thus $\neg B_1 \vee \ldots \vee \neg B_m \vee A_1$.

NB: there is only one positive literal in the clause.

The formula associated to the query $:- B_1, \ldots, B_m$ is $\forall x_1 \ldots \forall x_n \; (B_1 \wedge \ldots \wedge B_m) \rightarrow \bot$, thus $\neg B_1 \vee \ldots \vee \neg B_m$.

Intuition: find a refutation with Resolution!

Prolog and Resolution?

The formula associated to $A_1 :- B_1, \dots, B_m$ with variables $X_1, \dots, X_n$ is $\forall x_1 \dots \forall x_n\ ((B_1 \wedge \dots \wedge B_m) \rightarrow A_1)$, thus $\neg B_1 \vee \dots \vee \neg B_m \vee A_1$.

NB: there is only one positive literal in the clause.

The formula associated to the query $:- B_1, \dots, B_m$ is $\forall x_1 \dots \forall x_n\ (B_1 \wedge \dots \wedge B_m) \rightarrow \bot$, thus $\neg B_1 \vee \dots \vee \neg B_m$.

Intuition: find a refutation with Resolution!

Definition (Horn clause)

A clause is **defined** if it contains one and only one positive literal. A clause is **negative** if it does not contain positive literal.
A **Horn clause** is either a defined clause, either a negative clause.

Link between Prolog and Resolution

For instance: the first successful branch.

Link between Prolog and Resolution

For instance: the first successful branch.

A successful branch **is just a refutation using Resolution** from the set of Horn clauses.

$\exists x_1 \ldots \exists x_n \ (B_1 \wedge \ldots \wedge B_m)$ is a logical consequence of the program.

Link between Prolog and Resolution

For instance: the first successful branch.

A successful branch **is just a refutation using Resolution** from the set of Horn clauses.

$\exists x_1 \ldots \exists x_n \ (B_1 \wedge \ldots \wedge B_m)$ is a logical consequence of the program.

A **constructive** proof is found (the variables are assigned).

Link between Prolog and Resolution

For instance: the first successful branch.

A successful branch **is just a refutation using Resolution** from the set of Horn clauses.

$\exists x_1 \dots \exists x_n \ (B_1 \wedge \dots \wedge B_m)$ is a logical consequence of the program.

A **constructive** proof is found (the variables are assigned).

Prolog uses the Selective Linear Definite clause Resolution: only one literal can be **selected** for Resolution application, namely the most recent one in the clause.