Overview

1. Syntax: data
2. Manipulating data: Unification
3. Syntax: code
4. Semantics: meaning of programs
5. Executing logic programs
Syntax: Terms (Variables, Constants, and Structures)

- **Variables:** start with an uppercase character (or `_`), may include `_` and digits.
  
  Examples: X, Im4u, A_little_garden, _, _x, _22

- **Constructor:** lowercase first character, may include `_` and digits. Also, some special characters. Quoted, any character:
  
  Examples: a, dog, a_big_cat, x22, 'Hungry man', [], *, >

- **Structures:** a constructor (the structure name) followed by a fixed number of arguments between parentheses:
  
  Example: date(monday, Month, 1994)

  Arguments can in turn be variables, constants, and structures.

- **Constants:** structures without arguments (only name) and also numbers (with the usual decimal, float, and sign notations).
  
  Example: dad/0, 0, 999, 5, 3.2, 0.23e-5.

A term is ground if it does not contain free variables. A variable is free if it has not been assigned a value yet.

<table>
<thead>
<tr>
<th>Term</th>
<th>Constructor</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>dad</td>
<td>constant</td>
<td>dad/0</td>
</tr>
<tr>
<td>time(min, sec)</td>
<td>structure</td>
<td>time/2</td>
</tr>
<tr>
<td>pair(Calvin, Hobbes)</td>
<td>structure</td>
<td>pair/2</td>
</tr>
<tr>
<td>illegal</td>
<td>structure</td>
<td>time/2</td>
</tr>
<tr>
<td>legal</td>
<td>structure</td>
<td>dad/0</td>
</tr>
<tr>
<td>A good time</td>
<td>constant</td>
<td>legal</td>
</tr>
</tbody>
</table>

Examples of a first-order language: the data structures of a logic program. Variables, constants, and structures as a whole are called *terms* (they are the terms of a first-order language): the data structures of a logic program.

- A constant can be seen as a structure with arity zero.

  A *arity* is the number of arguments of a structure. Constructors are represented as name/arity (e.g., date/3).

- A variable is free if it has not been assigned a value yet.

- A term is ground if it does not contain free variables.

Syntact: Terms
Manipulating Data Structures (Unification)

• Unification is the only mechanism available in logic programs for manipulating data structures. It is used to:
  - Pass parameters.
  - Return values.
  - Access parts of structures.
  - Give values to variables.

• Unification is a procedure to solve equations on data structures.
  - As usual, it returns a minimal solution to the equation (or the equation system).
  - As many equation solving procedures it is based on isolating variables and then substituting them by their values.

Unifying two terms $A$ and $B$: is asking if they can be made syntactically identical by giving (minimal) values to their variables.

I.e., find a solution $\theta$ to equation $A = B$ (or, if impossible, fail).

Only variables can be given values!

Two structures can be made identical only by making their arguments identical.

(1) Structures with different name and/or arity cannot be unified because it would create an infinite term. This is known as the occurs check.

(2) A variable cannot be given as value a term which contains that variable.

E.g.

<table>
<thead>
<tr>
<th>$A$</th>
<th>$B$</th>
<th>$\theta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(X, g(t))$</td>
<td>$f(m(h), g(M))$</td>
<td>${X=m(h), M=t}$</td>
</tr>
<tr>
<td>$f(X, g(t))$</td>
<td>$f(m(h), g(t))$</td>
<td>$\theta$</td>
</tr>
<tr>
<td>$X$</td>
<td>$X$</td>
<td>$X$</td>
</tr>
<tr>
<td>$Y$</td>
<td>$Y$</td>
<td>$Y$</td>
</tr>
<tr>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>

- Unification: is a procedure to solve equations on data structures.
  - As usual, it returns a minimal solution to the equation (or the equation system).
  - As many equation solving procedures it is based on isolating variables and then substituting them by their values.

- Pass parameters.
- Return values.
- Access parts of structures.
- Give values to variables.
Unification Algorithm

Let $A$ and $B$ be two terms:

1. $\theta = \emptyset$, $E = \{ A = B \}$

2. while not $E = \emptyset$:
   2.1. delete an equation $T = S$ from $E$
   2.2. case $T$ or $S$ (or both) are (distinct) variables. Assuming $T$ variable:
      • (occur check) if $T$ occurs in the term $S$ → halt with failure
      • substitute variable $T$ by term $S$ in all terms in $\theta$
      • substitute variable $T$ by term $S$ in all terms in $E$
      • add $T = S$ to $\theta$
   2.3. case $T$ and $S$ are non-variable terms:
      • if their names or arities are different → halt with failure
      • obtain the arguments $\{ T_1, \ldots, T_n \}$ of $T$ and $\{ S_1, \ldots, S_n \}$ of $S$
      • add $\{ T_1 = S_1, \ldots, T_n = S_n \}$ to $E$

3. halt with $\theta$ being the m.g.u of $A$ and $B$

Unification Algorithm Examples (I)

1. Unify: $A = p(X, X)$ and $B = p(f(Z), f(W))$
   $\theta$ $E$ $T$ $S$
   \[
   \begin{array}{c|c|c|c}
   \emptyset & \{ (\lambda) \iff = Z, (\lambda) \iff = X \} & \{ (\lambda) \iff = Z \} & \{ Z = \lambda \} \\
   \{ (\lambda) \iff = Z \} & \{ (\lambda) \iff = X \} & \{ Z = X \} & \{ \} \\
   \{ (\lambda) \iff = X \} & \{ (\lambda) \iff = Z \} & \{ X = (\lambda) \iff \} & \{ Z = X \} \\
   \{ (\lambda) \iff = Z \} & \{ (\lambda) \iff = X \} & \{ Z = X \} & \{ \} \\
   (X'Z)^d = ((\lambda) \iff X)^d = (X'Z)^d(\lambda) \iff X & \{ (X'Z)^d((\lambda) \iff X)^d \} & \{ (X'Z)^d((\lambda) \iff X)^d \} & \{ \} \\
   S & L & F & \emptyset
   \end{array}
   \]

2. Unify: $A = p(X, f(Y))$ and $B = p(Z, X)$
   $\theta$ $E$ $T$ $S$
   \[
   \begin{array}{c|c|c|c}
   \emptyset & \{ (\lambda) \iff = Z, (\lambda) \iff = X \} & \{ (\lambda) \iff = Z \} & \{ Z = \lambda \} \\
   \{ (\lambda) \iff = Z \} & \{ (\lambda) \iff = X \} & \{ (\lambda) \iff = X \} & \{ (\lambda) \iff = Z \} \\
   \{ (\lambda) \iff = X \} & \{ (\lambda) \iff = Z \} & \{ (\lambda) \iff = X \} & \{ (\lambda) \iff = Z \} \\
   \{ (\lambda) \iff = Z \} & \{ (\lambda) \iff = X \} & \{ (\lambda) \iff = X \} & \{ (\lambda) \iff = Z \} \\
   (M) \iff \{ (Z) \iff = X \} & \{ (M) \iff \{ (Z) \iff = X \} \} & \{ (Z) \iff = X \} & \{ (M) \iff \{ (Z) \iff = X \} \} \\
   \{ (M) \iff \{ (Z) \iff = X \} \} & \{ (M) \iff \{ (Z) \iff = X \} \} & \{ (M) \iff \{ (Z) \iff = X \} \} & \{ (M) \iff \{ (Z) \iff = X \} \} \\
   ((M) \iff \{ (Z) \iff = X \})^d \{ (M) \iff \{ (Z) \iff = X \} \} & \{ ((M) \iff \{ (Z) \iff = X \})^d \{ (M) \iff \{ (Z) \iff = X \} \} \} & \{ ((M) \iff \{ (Z) \iff = X \})^d \{ (M) \iff \{ (Z) \iff = X \} \} \} & \{ \} \\
   S & L & F & \emptyset
   \end{array}
   \]
### Syntax: Literals and Predicates (Procedures)

<table>
<thead>
<tr>
<th>Literal</th>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{arrives(john, date(monday, Month, 1994))}</td>
<td>\texttt{name(barry), color(black)}</td>
</tr>
</tbody>
</table>

- Full predicate names are denoted as \texttt{name/arity(e.g., arrives/2)}.
- The number of arguments is the \texttt{arity} of the predicate.
- The arguments are \texttt{terms}.
- Literals are used to define procedures and procedure calls.
- Terms are data.

### Unification Algorithm Examples (II)

1. **Unify:**
   \[ A = p(X,f(Y)) \text{ and } B = p(a,g(b)) \]
   
   1. \[ \theta \in T \{ \}
   \]
   2. \[ p(X,f(Y)) = p(a,g(b)) \]
   3. \[ \{ X = a, f(Y) = g(b) \} \]
   4. \[ X \quad a \]
   5. \[ \{ f(Y) = g(b) \} \]
   6. \[ f(Y) \quad g(b) \]
   7. \[ \text{fail} \]

2. **Unify:**
   \[ A = p(X,f(X)) \text{ and } B = p(Z,Z) \]

   1. \[ \theta \in T \{ \}
   \]
   2. \[ p(X,f(X)) = p(Z,Z) \]
   3. \[ \{ X = Z, f(X) = Z \} \]
   4. \[ X \quad Z \]
   5. \[ \{ f(Z) = Z \} \]
   6. \[ f(Z) \quad Z \]
   7. \[ \text{fail} \]

- Literals are used to define procedures and procedure calls.
- Terms are data structures, so the arguments of literals.
- Terms are data, and literals are terms.
- Literals and terms are syntactically identical.
- But, they are distinguished by context:
  - They are distinguished by context.
  - Syntax: Literals and Predicates (Procedures)

- Example: 
  - \texttt{arrives(john, date(monday, Month, 1994))} 
  - \texttt{name(barry), color(black)} 
  - Full predicate names are denoted as \texttt{name/arity(e.g., arrives/2)}.
  - The number of arguments is the \texttt{arity} of the predicate.
  - The arguments are \texttt{terms}.
  - Literals are used to define procedures and procedure calls.
  - Terms are data structures, so the arguments of literals.

- Syntax: Literals and Predicates (Procedures)

- Example: 
  - \texttt{arrives(john, date(monday, Month, 1994))} 
  - \texttt{name(barry), color(black)} 
  - Full predicate names are denoted as \texttt{name/arity(e.g., arrives/2)}.
  - The number of arguments is the \texttt{arity} of the predicate.
  - The arguments are \texttt{terms}.
  - Literals are used to define procedures and procedure calls.
  - Terms are data structures, so the arguments of literals.
Syntax: Operators

- Functors and predicate names can be defined as prefix, postfix, or infix operators.
  
  **Examples:**
  - $a + b$ is the term $+(a, b)$ if $+/2$ declared infix
  - $-b$ is the term $-(b)$ if $-/1$ declared prefix
  - $a < b$ is the term $<(a, b)$ if $</2$ declared infix

- We assume some such operator definitions are always preloaded, so that they can be always used.

Syntax: Clauses (Rules and Facts)

- **Rule:** an expression of the form:
  
  $$p_0(t_1, t_2, \ldots, t_n) :- p_1(t_1, t_2, \ldots, t_n), \ldots, p_m(t_m, t_{m1}, t_{m2}, \ldots, t_{mn}).$$

  - $p_0(\ldots)$ to $p_m(\ldots)$ are **literals**.
  - $p_0(\ldots)$ is called the **head** of the rule.
  - The $p_i$ to the right of $:-$ are called **goals** and form the **body** of the rule.
  - They are also called **procedure calls**.
  - The $d$ to the right of $-$ are called the **head of the rule**.

- **Fact:** an expression of the form:
  
  $$p(t_1, t_2, \ldots, t_n).$$

  (i.e., a rule with empty body –no neck–).

**Examples:**

- Operators (just syntax).

  - Functions and predicate names can be defined as prefix, postfix, or infix.
Syntax: Clauses

Rules and facts are both called clauses (since they are clauses in first-order logic) and form the code of a logic program.

• Example:

meal(soup, beef, coffee).

meal(First, Second, Third) :-
  appetizer(First),
  main_dish(Second),
  dessert(Third).

• :- stands for ←, i.e., logical implication (but written “backwards”).

Comma is conjunction.

And thus, is a Horn clause of the form:

meal(First, Second, Third) ← appetizer(First) ∧ main_dish(Second) ∧ dessert(Third)

Therefore, the above rule stands for:

dessert(Third),
  main_dish(Second),
  appetizer(First) → meal(First, Second, Third).

Rules and facts are both called clauses (since they are clauses in first-order logic).

Syntax: Predicates and Programs

• Predicate (or procedure definition): a set of clauses whose heads have the same name and arity (the predicate name).

Examples:

pet(barry). animal(tim).

pet(X) :- animal(X), barks(X). animal(spot).

pet(X) :- animal(X), meows(X). animal(hobbes).

Predicate pet/1 has three clauses. Of those, one is a fact and two are rules.

Predicate pet/1 has three clauses. Of those, one is a fact and two are rules.

• Note (variable scope): the X vars. in the two clauses above are different, despite the same name. Vars. are local to clauses (and are renamed any time a clause is used), hence clauses all have the same name and any (the predicate name).

Predicate (or procedure definition): a set of clauses whose heads have the same name and arity (the predicate name).
Declarative Meaning of Facts and Rules

The declarative meaning is the corresponding one in first-order logic, according to certain conventions:

• **Facts**: state things that are true.

  (Note that a fact "p." can be seen as the rule "p ← true")

  Example: the fact animal(spot).
  can be read as "spot is an animal".

• **Rules**: state implications that are true.

  Example: the rules pet(X) :- animal(X), barks(X).
  pet(X) :- animal(X), meows(X).

  Express two ways for X to be a pet.

  rules: provide different alternatives (for p).

  Example: clauses: express the same predicate.

  Programs: are sets of logic formulae, i.e., a first-order theory: a set of statements assumed to be true. In fact, a set of Horn clauses.

- Thus, a rule represents "if p_1 and ... and p_m are true, then p is true".
- Thus a rule means "p ← p_1 ∧ ... ∧ p_m".

  Example: the rule "p ← q_1 ∧ ... ∧ q_m".
  Facts: state things that are true.

  Declarative Meaning of Predicates and Programs

  • **Predicates**: clauses in the same predicate

    p :- p_1 ∧ ... ∧ p_n.

    Example: the rules
    pet(X) :- animal(X), barks(X).
    pet(X) :- animal(X), meows(X).

    express two ways for X to be a pet.

    Examples: clauses in the same predicate.

    Programs: are sets of logic formulae, i.e., a first-order theory: a set of statements assumed to be true. In fact, a set of Horn clauses.

- Thus, a rule represents "if p_1 and ... and p_m are true, then p is true".
- Thus a rule means "p ← p_1 ∧ ... ∧ p_m".

  Example: the rule "p ← q_1 ∧ ... ∧ q_m".
  Facts: state things that are true.

  Declarative Meaning of Facts and Rules

  The declarative meaning is the corresponding one in first-order logic, according to certain conventions:

  • **Facts**: state things that are true.

    (Note that a fact "p." can be seen as the rule "p ← true")

    Example: the fact animal(spot).
    can be read as "spot is an animal".

  Example: the rule "p ← q_1 ∧ ... ∧ q_m".
  Facts: state things that are true.

  Declarative Meaning of Facts and Rules
Queries

• Query: an expression of the form:

\[-p_1(t_1, \ldots, t_{n_1}), \ldots, p_n(t_n_1, \ldots, t_{n_m})\] (i.e., a clause without a head)

⋄ The \(p_i\) to the right of \(-\) are called goals (procedure calls).

⋄ Sometimes, also the whole query is called a (complex) goal.

Example given the program above and the query:

\[-p(X)\] attempting to find an answer to the query

Execution: given a program and a query, executing the logic program is

\[\text{Example of a logic program:}\]

\[
\text{animal(tim).}
\]
\[
\text{animal(spot).}
\]
\[
\text{animal(hobbes).}
\]
\[
\text{pet}(X) \leftarrow \text{animal}(X), \text{barks}(X).
\]
\[
\text{pet}(X) \leftarrow \text{animal}(X), \text{meows}(X).
\]
\[
\text{meows(tim).}
\]
\[
\text{barks(spot).}
\]
\[
\text{roars(hobbes).}
\]
\[
\text{pet}(X) \leftarrow \text{animal}(X), \text{roars}(X).
\]

Example: \(-p(X)\)

A query is a clause to be deduced:

A query represents a question to the program.

A query is called a (complex) goal.

The \(\rightarrow\) to the right of \(-\) are called goals (procedure calls).

For \(\rightarrow\) to be true, \(-\) stands also for (→)

A clause without a head

Query: an expression of the form:

\[\left(\begin{array}{c}
\ldots
\ldots
\end{array}\right)\]
The Search Tree

A query + a logic program together specify a search tree.

Example:

? pet(X)

With the previous program generates this search tree (the boxes represent the "and" parts [except leaves]):

animal(tim) animal(spot) animal(hobbes)

pet(X)

animal(X), barks(X)

animal(hobbes) animal(spot) animal(tim)

animal(X), meows(X)

meows(tim) barks(spot)

Different query → different tree.

A particular execution strategy defines how the search tree will be explored during execution.

Node: execution always finishes in the leaves (the facts).

Explore branches all at the same time

Explore goals in boxes all at the same time

Explore branches left-to-right or right-to-left

Explore goals in boxes left-to-right or right-to-left

Explore the tree top-down → "call"

Explore the tree bottom-up → "deduce"

Explore goals in boxes left-to-right or right-to-left

Explore branches left-to-right or right-to-left

Explore goals in boxes all at the same time

Explore branches all at the same time
Running Programs: Interaction with the System

- Practical systems implement a particular strategy (all Prolog systems implement the same one).
- The strategy is meant to explore the whole tree, but returns solutions one by one:
  - Example: (?- is the system prompt)
    - ?- pet(X).
    - X = spot
    - yes
    - ?- pet(X).

- Prolog systems also allow to create executables that start with a given predefined query (which is usually main/0 and/or main/n).

Operational Meaning of Programs

- A logic program is operationally a set of procedure definitions (the predicates).
- A query - p is an initial procedure call.
- A procedure definition with one clause p :- p1, ..., pn means:
  - "to execute a call to p, call p1 and ... and pn".
- If several clauses (definitions) p :- p1, ..., pn, q1, ..., qm... means:
  - "to execute a call to p, call p1 and ... and pn, or, alternatively, q1 and ... and qm...

- In principle, the order in which p1, ..., pn are called does not matter, but, in practical systems, it is fixed.
- Unique to logic programming – it is like having several alternative procedure definitions.
- System usually stops when the first solution found, user can ask for more.
- Languages are designed to find several possible paths to a solution and they should be explored.
- Languages of logic programming – 1 is like having several alternative procedure definitions.

Starting the executable.

- Some systems allow to introduce clauses in the text of the program, starting with :- (resemble a rule without head).
- These are executed upon loading the file (or "evaluating" [X, Y, Z] which is usually main/0 and/or main/n).
- Prolog systems also allow to create executables that start with a given predefined query.

Example: (?- is the system prompt)

- The strategy is meant to explore the whole tree, but returns solutions one by one:
  - (all Prolog systems implement the same one).

Running Programs: Interaction with the System
A (Schematic) Interpreter for Logic Programs (Prolog)

Let a logic program \( P \) and a query \( Q \),

1. Make a copy \( Q' \) of \( Q \)
2. Initialize the resolvent \( R \) to be \( \{ Q' \} \)
3. While \( R \) is nonempty do:
   1. Make a copy \( \theta \) of \( R \)
   2. Initialize the resolvent \( R \) to be \( \theta \)
   3. Take the last pendling branch
      4. Obtain solution \( \{ \} \) to \( \theta = R \)
5. Explore the last pending branch for more solutions upon request
6. Output solution \( \mu \) to \( Q = Q' \)

Running Programs: Alternative Execution Paths

\[
\begin{array}{|c||c|c|}
\hline
\text{clause} & \text{resolvent} & \text{unified} \\
\hline
\text{pet(X)} & \text{parkes(tim)} & \text{X=tim} \\
\{ \text{X=tim} \} & \text{animal(tim)}, \text{parkes(tim)} & \text{X=tim} \\
\{ \} & \text{animal(X)}, \text{parkes(X)} & \text{X=tim} \\
\{ \} & \text{animal(X)}, \text{parkes(X)} & \text{X=tim} \\
\hline
\end{array}
\]

But solutions exist in other paths!

\[ \neg \text{pet(X)} \]
- top-down, left-to-right

\[
\begin{align*}
\text{animal(tim):} & \quad \text{parkes(tim)} \\
\text{animal(spot):} & \quad \text{parkes(spot)} \\
\text{animal(hobbes):} & \quad \text{roars(hobbes)} \\
\text{animal(spot):} & \quad \text{meows(spot)} \\
\text{animal(hobbes):} & \quad \text{meows(hobbes)} \\
\text{barks(spot):} & \quad \text{meows(spot)} \\
\text{meows(tim):} & \quad \text{pet(tim)} \\
\text{roars(hobbes):} & \quad \text{pet(tim)} \\
\end{align*}
\]
Running Programs: Different Branches

Deep backtracking: the application of the above procedure (undo the execution of the previous goal(s)).

Shallow backtracking: the clause selection performed in 3.2.2 (forwards execution again)

1. Take the last literal successfully executed
2. Take the clause against which it was executed
3. Take the unifier of the literal and the clause head
4. Undo the unifications
5. Go to 3.2.2 (forwards execution again)

Explore the last pending branch means:

(Schematic) Algorithm:

- it is a kind of "backwards execution"
- branches of the search tree

Backtracking is the way in which Prolog execution strategy explores different

---

<table>
<thead>
<tr>
<th>Clause</th>
<th>query</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta$</td>
<td>$\top$</td>
</tr>
</tbody>
</table>
Running Programs: Complete Execution (All Solutions)

1: pet(X) :- animal(X), barks(X).
2: pet(X) :- animal(X), meows(X).
3: animal(tim).
4: animal(spot).
5: animal(hobbes).
6: barks(spot).
7: meows(tim).
8: roars(hobbes).

?- pet(X).

Clause | θ | Choice-points
--- | --- | ---
pet(X) | pet(X) | C1*

pet(X) | pet(X) | C2*
pot(X) | pot(X) | C3*

pet(X) | pet(X) | C4*

pet(X) | pet(X) | C5*

pet(X) | pet(X) | C6*

pet(X) | pet(X) | C7*

pet(X) | pet(X) | C8*

Clause | θ | Choice-points
--- | --- | ---
pot(X) | pot(X) | C1*

pot(X) | pot(X) | C2*
pot(X) | pot(X) | C3*

pot(X) | pot(X) | C4*

pot(X) | pot(X) | C5*

pot(X) | pot(X) | C6*

pot(X) | pot(X) | C7*

pot(X) | pot(X) | C8*

Clause | θ | Choice-points
--- | --- | ---
pot(X) | pot(X) | C1*

pot(X) | pot(X) | C2*
pot(X) | pot(X) | C3*

pot(X) | pot(X) | C4*

pot(X) | pot(X) | C5*

pot(X) | pot(X) | C6*

pot(X) | pot(X) | C7*

pot(X) | pot(X) | C8*
Running Programs: Complete Execution (All Solutions)

C1: pet(X) :- animal(X), barks(X).
C2: pet(X) :- animal(X), meows(X).
C3: animal(tim).
C4: animal(spot).
C5: animal(hobbes).
C6: barks(spot).
C7: meows(tim).
C8: roars(hobbes).

?- pet(X).

The Search Tree Revisited

- Different execution strategies explore the tree in a different way.
- A strategy is complete if it guarantees that it will find all existing solutions.
- Prolog does it top-down, left-to-right (i.e., depth-first).

<table>
<thead>
<tr>
<th>Clause</th>
<th>Choice-points</th>
<th>Failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>pet(X)</td>
<td>0</td>
<td>false</td>
</tr>
<tr>
<td>animal(X), barks(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=hobbes}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=hobbes}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=hobbes}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=hobbes}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=hobbes}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=hobbes}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=hobbes}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=hobbes}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=hobbes}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=tim}</td>
<td>true</td>
</tr>
<tr>
<td>animal(X), meows(X)</td>
<td>{X=spot}</td>
<td>true</td>
</tr>
</tbody>
</table>
• All solutions are at finite depth in the tree.

• Failures can be at finite depth or in some cases, be an infinite branch.

• Incomplete: may fall through an infinite branch before finding all solutions.

• But very efficient: it can be implemented with a call stack, very similar to a traditional programming language.
Breadth-First Search

- Will find all solutions before falling through an infinite branch.
- But costly in terms of time and memory.
- Used in some of our examples (via Ciao's bfs package).

The Execution Mechanism of Prolog

- Always execute literals in the body of clauses left-to-right.
- At a choice point, take first unifying clause (i.e., the leftmost unexplored branch).
- On failure, backtrack to the next unexplored clause of last choice point.

```
grandparent(C,G):- parent(C,P), parent(P,G).
parent(C,P):- father(C,P).
parent(C,P):- mother(C,P).
father(charles,philip).
father(ana,george).
mother(charles,ana).
father(charles,philipp).
```

Check how Prolog explores this tree by running the debugger!
Comparison with Conventional Languages

• Conventional languages and Prolog both implement (forward) continuations: the place to go after a procedure call succeeds. I.e., in:

```
p(X,Y):- q(X,Z), r(Z,Y).
q(X,Z) :- ...
```
when the call to `q/2` finishes with `success` (the forward continuation). Thus:

• In Prolog, when there are procedures with multiple definitions, there is also a backward continuation: the place to go if there is a failure. I.e., in:

```
p(X,Y):- q(X,Z), r(Z,Y).
q(X,Z) :- ...
```
if the call to `q/2` fails at any point, execution continues at the second clause of `q/2` (the backward continuation). Since the execution strategy of Prolog is fixed, the ordering in which the programmer writes clauses and goals is important.

Ordering of Clauses and Goals

• Since the execution strategy of Prolog is fixed, the ordering in which the programmer writes clauses and goals is important. Thus:

- The order in which solutions are found.
- The order in which failure occurs (and backtracking triggered).
- The order in which infinite failure occurs (and the program founders).

Ordering of goals determines the order in which unification is performed. Thus:

• The order in which alternate paths are explored.

Again, the debugger (see later) can be useful to observe execution.

• The order in which alternate paths are explored.

Convention of languages and Prolog both implement (forward) continuations.
Ordering of Clauses

(1) Solutions (finite failure) plus all

- Tail solution
- Singleton

... ...

• An infinite computation which
  yields all solutions

- A finite failure plus all

... ...

• An infinite computation with no
  solutions (infinite failure)
Execution Strategies

- Search rule(s): how are clauses/branches selected in the search tree (step 3.2 of the resolution algorithm).

- Computation rule(s): how are goals selected in the boxes of the search tree (step 3.1 of the resolution algorithm).

- Prolog execution strategy:
  - Computation rule: left-to-right (as written)
  - Search rule: top-down (as written)

Summary

- A logic program declares known information in the form of rules (implications) and facts.
- Executing a logic program is deducing new information.
- A logic program can be executed in any way which is equivalent to deducing the query from the program.
- Different execution strategies have different consequences on the computation of programs.

Prolog is a logic programming language which uses a particular strategy (and goes beyond logic because of its predefined predicates).

Execution Strategies

- Step 3.1 of the resolution algorithm: how are rules selected in the boxes of the search tree.
- Computation rule(s): how are clauses/branches selected in the search tree (step 3.2 of the resolution algorithm).
- Search rule(s): how are goals selected in the search tree.
- Prolog execution strategy: left-to-right (as written).
Exercise

Write a predicate jefe/2 which lists who is boss of whom.

\[ \text{jefe}(X,Y) \text{ iff } X \text{ is above } Y \text{ in the chain of who is boss of whom.} \]

Write a predicate jefazo/2 (no facts) which reads:

\[ \text{jefazo}(X,Y) \text{ iff } X \text{ is above } Y \text{ in the chain of who is boss of whom.} \]

Write a predicate curritos/2 which lists pairs of people who have the same direct boss.

\[ \text{curritos}(X,Y) \text{ iff } X \text{ and } Y \text{ have a common direct boss.} \]

Write a predicate curritos/2 which lists pairs of people who have the same direct boss.

\[ \text{curritos}(X,Y) \text{ iff } X \text{ and } Y \text{ have a common direct boss.} \]

Write a predicate jefe/2 which lists who is boss of whom (a list of facts). It reads:

\[ \text{jefe}(X,Y) \text{ iff } X \text{ is direct boss of } Y. \]

Write a predicate jefe/2 which lists who is boss of whom (a list of facts). It reads:

\[ \text{jefe}(X,Y) \text{ iff } X \text{ is direct boss of } Y. \]