Logic and Knowledge Representation

*Programming as problem-solving, First steps in Prolog*

20 April 2018

Giovanni Sileno gsileno@enst.fr
Télécom ParisTech, Paris-Dauphine University
This course is an introduction to symbolic techniques in Artificial Intelligence (AI), following an homonym course held by Jean-Louis Dessalles.

The course website:
http://aicourse.r2.enst.fr:4242/SCIA

Frontal teacher: Giovanni Sileno, gsileno@enst.fr
Do not hesitate to write me or ask questions in class!
Logic and Knowledge Representation

Being a new course, we will plausibly tweak on the fly the content, the exercises, and the workload.

In principle, your grade will be a weighted average of:

- weekly exercises proposed on the course webpages
- a final exam
- a programming project
“Mechanical” computing
Pascal: Pascaline ~ 1650

Helping his father (tax accountant of Normandy, appointed by Richelieu), Pascal invented a machine for *mechanic calculation*, performing *addition* and *subtraction*.
Before him, Schickard had already invented an “arithmetical instrument”, but unfortunately he was not able to publicly present a full working copy.
Leibniz: Stepped Reckoner ~1680

Influenced by the Pascaline, Leibniz proposed a mechanic calculator performing all four operations: **addition, subtraction, multiplication and division.**
Furthermore, Leibniz believed that calculation would be the key to settle all human conflicts and disagreements...

→ *characteristics numbers* instead of concepts.

Gottfried Wilhelm von Leibniz
Leibniz: *Calculemus!* ~1686

Furthermore, Leibniz believed that calculation would be the key to settle all human conflicts and disagreements...

→ *characteristics numbers* instead of concepts.

animal = 2  
rational = 3  
man = rational animal = 2*3 = 6

Gottfried Wilhelm von Leibniz
Furthermore, Leibniz believed that calculation would be the key to settle all human conflicts and disagreements...

→ characteristics numbers instead of concepts.

animal = 2
rational = 3
man = rational animal = 2*3 = 6

Is every man rational?
Yes, because 6 is divisible by 3.
Machines as symbol handlers

Starting from the Pascaline, computing machines respond to the need to displace tedious, repetitive (symbolic) work.
A physical symbol system has the necessary and sufficient means for general intelligent action

Allen Newell and Herbert A. Simon
Computer Science as Empirical Inquiry: Symbols and Search (1976)
A physical symbol system has the necessary and sufficient means for general intelligent action

Allen Newell and Herbert A. Simon
Computer Science as Empirical Inquiry: Symbols and Search (1976)

...basis for Good Old Fashioned AI (GOFAI)
Machines as symbol handlers

Starting from the Pascaline, computing machines respond to the need to displace tedious, repetitive (symbolic) work.

but how to say to the machine what to do?
Machine code/instructions
Related to the physical structure of the computer.

+ powerful and fast
- long programs
- difficult to be written
- difficult to be revised
Natural (human) language

It is the language we use in all our communications, learned since our childhood.

+ Expressively rich.
- Ambiguous, redundant.
Programming languages

A programming language is a language which is intermediary between machine code and natural language.
Programming languages

VanRoy 2009,
Weinberg 1977
Programming languages

A programming language is a language which is *intermediary* between machine language and natural language.

- But **what** we have to tell to the machine?
Programming as problem-solving
A well-defined problem is usually defined in terms of

- an initial state or situation
- a goal state, i.e. a desired outcome,
- certain resources (which put contraints on the possible paths towards the goal).
An ancient puzzle ~ 9th century

- Once upon a time a farmer went to market and purchased a **fox**, a **goose**, and a **bag of beans**. On his way home, the farmer came to the bank of a river. In crossing the river by boat, the farmer could carry only himself and a single one of his purchases - the fox, the goose, or the bag of the beans.
Once upon a time a farmer went to market and purchased a fox, a goose, and a bag of beans. On his way home, the farmer came to the bank of a river. In crossing the river by boat, the farmer could carry only himself and a single one of his purchases - the fox, the goose, or the bag of the beans.
An ancient puzzle ~ 9th century

- Once upon a time a farmer went to market and purchased a fox, a goose, and a bag of beans. On his way home, the farmer came to the bank of a river. In crossing the river by boat, the farmer could carry only himself and a single one of his purchases - the fox, the goose, or the bag of the beans.
An ancient puzzle ~ 9th century

- If left together, the fox would eat the goose, or the goose would eat the beans.
- The farmer's challenge was to carry himself and his purchases to the far bank of the river, leaving each purchase intact. How did he do it?
An ancient puzzle ~ 9th century

- What is the goal?
- What is the initial situation?
- Which are the resources/constraints?
From *problems* to *solvers*

Problem → Analysis → Problem solving method → Planning → Problem solving task → Execution → Solution

Problem → Analysis → Algorithm → Programming → Program → Execution → Outcome
From *problems* to *solvers*

The "real" problem is not programming but finding the path toward the solution.
Imperative vs Declarative
Imperative vs Declarative

• Imperative:
  - programming focusing on the **sequence of operations** necessary to solve the problem (which in turn usually stays implicit)

• Declarative
  - programming focusing on describing the **problem** (while the sequence of operations to be performed is left implicit)
Imperative programming

Focus: *how to compute*

Based on instructions, correspondent to actions **commanded** to the machine.

- It assumes that the computer can maintain the changes (the *side-effects*) caused by the computation process.
Imperative programming

Focus: *how to compute*

- Most popular programming languages implement the imperative paradigm:
  - it most closely resembles the actual machine itself, so the programmer thinks in a much closer way to the machine;
  - because of such closeness, it was until recently the only one efficient enough for widespread use.
Imperative programming

- Advantages
  - efficient as close to the machine
  - popular
  - familiar

- Disadvantages
  - a program can be complex to understand, because the *referential transparency* does not hold (due to *side effects*)
  - *abstraction* is more limited
  - *order* is crucial, which is not suited in certain problems
Declarative programming

Focus: *what to compute (as desired outcome)*

- It is not concerned about how to do things, but what should be obtained.
  - Languages: domain specific (e.g. HTML), query (SQL), logic (Prolog).
Declarative/Logic programming

Focus: *what to compute* (as desired outcome)

- Various logical assertions about a situation are made, describing all known **facts** and **rules** about the modeled world. Then **queries** are made.

- The role of the computer is to *maintain data* and to perform *inferences*.
Algorithm = Logic + Control

“An algorithm can be regarded as consisting of

- a logic component, which specifies the knowledge to be used in solving problems, and

- a control component, which determines the problem-solving strategies by means of which that knowledge is used.

The logic component determines the meaning of the algorithm whereas the control component only affects its efficiency.”

Robert Kowalsky, Algorithm = Logic + Control (1979)
Imperative vs Declarative

• Imperative:
  – inside-to-outside approach: all execution alternatives are explicitly specified and new alternatives must be explicitly added

• Declarative
  – outside-to-inside approach: constraints implicitly specify execution alternatives as all alternatives that satisfy the constraints; adding new constraints usually means discarding some execution alternatives
Imperative:
you command
the directions
Imperative:
you command
the directions
Imperative: you command the directions

- What if the labyrinth changes?
Declarative: you give just the labyrinth. The computer finds the way.
Declarative: you give just the labyrinth. The computer finds the way.

- For instance, via *trial*, *error* and backtracking
Declarative: you give just the labyrinth. The computer finds the way.

- For instance, via *trial, error* and backtracking
Declarative: you give just the labyrinth. The computer finds the way.

- For instance, via *trial, error* and backtracking
Declarative: you give just the labyrinth. The computer finds the way.

- For instance, via *trial, error* and backtracking
First steps in Prolog
History of Prolog

1965 Resolution algorithm by J. A. Robinson followed by SLD resolution by R. Kowalski

1972 PROgrammation en LOGique created by A. Colmerauer and P. Roussel in Luminy, Marseille

1980 Prolog acknowledged as a major A.I. language

... now various versions, used in Constraint Programming, or basis/reference for alternative techniques, as DataLog, Answer Set Programming (ASP), etc.
First program

parent(marge, lisa).
parent(marge, bart).
parent(marge, maggie).
parent(homer, lisa).
parent(homer, bart).
parent(homer, maggie).
parent(abraham, homer).
parent(abraham, herb).
parent(mona, homer).
parent(jackie, marge).
parent(clancy, marge).
parent(jackie, patty).
parent(clancy, patty).
parent(jackie, selma).
parent(clancy, selma).
parent(selma, ling).
First program

parent(marge, lisa).
parent(marge, bart).
parent(marge, maggie).
parent(homer, lisa).
parent(homer, bart).
parent(homer, maggie).
parent(abraham, homer).
parent(abraham, herb).
parent(mona, homer).
parent(jackie, marge).
parent(clancy, marge).
parent(jackie, patty).
parent(clancy, patty).
parent(jackie, selma).
parent(clancy, selma).
parent(selma, ling).

child(X,Y) :-
   parent(Y,X).
Prolog clauses

- Fact
  
  female(marge).
Prolog clauses

• Fact
  female(marge).

• Rule
  child(X,Y) :- parent(Y,X).
Prolog clauses

• Fact
  female(marge).

• Rule
  child(X,Y) :- parent(Y,X).

Exercise: write

  mother(X,Y)  grandparent(X,Y)
  ancestor(X,Y) cousin(X,Y)
Prolog syntax and bonuses
Prolog syntax: Terms

- Constants

  **Identifiers** strings of letters, digits, or underscore “_” that start with lower case letters.

  marge lisa x25 x_25 alpha_beta
Prolog syntax: Terms

- **Constants**
  
  **Identifiers** strings of letters, digits, or underscore “_” that start with lowercase letters.

  marge lisa x25 x_25 alpha_beta

  **Numbers**

  1.001 2 3.03
Prolog syntax: Terms

• Constants

  Identifiers strings of letters, digits, or underscore “_” that start with lower case letters.

    marge lisa x25 x_25 alpha_beta

Numbers

    1.001 2 3.03

Strings enclosed in single quotes

    'Mary' '.01' 'string'

Note that, because of the enclosing, strings can start with upper case letter, or can be a number now treated as a string.
Prolog syntax: Terms

• Constants

  Identifiers strings of letters, digits, or underscore “_” that start with lower case letters.

        marge lisa x25 x_25 alpha_beta

Numbers

        1.001 2 3.03

• Variables strings of letters, digits or underscore that start with an upper case letter or the underscore

        _x, Anna, Successor_State,

Note: underscore by itself (“_”) is a special anonymous variable.
Prolog syntax: Terms

- Structures

  \(<\text{identifier}>\)(\text{Term1}, ..., \text{Termk})

  recursive definition: each term can itself be a structure

  \begin{align*}
  \text{date}(1, \text{may}, 1983) \\
  \text{point}(X, Y, Z) \\
  \text{date}(+(0,1), \text{may}, +(1900,-(183,100)))
  \end{align*}
Prolog syntax: Terms

Structures can be thought as trees!

date(+0,1), may, +(1900,-183,100)
Prolog syntax: Predicates

- **Predicates** are syntaxically the same as structures.

\[
<\text{identifier}> (\text{Term}_1, \ldots, \text{Term}_k)
\]

\[
\text{parent}(\text{marge}, \text{bart})
\]

\[
\text{female}(\text{marge})
\]
Prolog syntax: Predicates

- **Predicates** are syntaxically the same as structures.

\[ \text{<identifier>}(\text{Term1}, \ldots, \text{Termk}) \]

- parent(marge, bart)

- female(marge)

\[ \text{.. but they are not terms!} \]

\[ \rightarrow \text{predicate identifiers} \neq \text{structure identifiers} \]
Prolog syntax: Predicates

things said about objects

→ predicate identifiers ≠ structure identifiers
Prolog Syntax

• A prolog program is a sequence of facts and rules.

• Facts are predicates terminated by a period ‘.’
  \(<\text{identifier}>)(\text{Term1, ..., Termk}).\)
Prolog Syntax

- A prolog program is a sequence of facts and rules.

- Facts are predicates terminated by a period “.”

  `<identifier>(Term1, ..., Termk).

Facts encode assertions, things that are declared certain!

  female(marge).
  parent(homer, bart).
Prolog Syntax

• A prolog program is a sequence of facts and rules.

• Rules are in the form:

  predicateH :- predicate1, .., predicatek.

  the left of " :- " is named head, the right body.
Prolog Syntax

• A prolog **program** is a sequence of **facts** and **rules**.

• **Rules** are in the form:

  
  \[ \text{predicate}\text{H} :\text{-} \text{predicate1}, \ldots, \text{predicatek}. \]

• Rules encode ways of deriving or computing a new fact.

  \[ \text{animal}(X) :\text{-} \text{elephant}(X). \]
  
  We can show that \( X \) is an animal if we can show that it is an elephant.
Prolog Syntax

• A prolog **program** is a sequence of **facts** and **rules**.

• **Rules** are in the form:

  \[ \text{predicate}_H :\neg \text{predicate}_1, \ldots, \neg \text{predicate}_k. \]

• Rules encode ways of deriving or computing a new fact.

  \[ \text{animal}(X) :\neg \text{elephant}(X). \]
  We can show that X is an animal if we can show that it is an elephant.

  \[ \text{father}(X,Y) :\neg \text{parent}(X,Y), \text{male}(X). \]
  We can show that X is a father of Y if we can show that X is a parent of Y and that X is male.
Prolog's main operation

- A **query** is a sequence of predicates
  
  \[\text{predicate1, predicate2, ..., predicatek}\]

- using the facts and rules in the Prolog program, the solver tries to *prove* that this sequence of predicates is true.

- in proving the sequence it (should) perform the computation you want.
Prolog's main operation

• A **query** is a sequence of predicates

  \[ \text{predicate}_1, \text{predicate}_2, \ldots, \text{predicate}_k \]

  – using the facts and rules in the Prolog program, the solver tries to *prove* that this sequence of predicates is true.
  – in proving the sequence it (should) perform the computation you want.

**delicate point:** *procedural computation* might be mixed with *declarative computation*
Prolog's resolution strategy
Horn clauses

- Facts and rules falls under the definition of **Horn clauses**, the structures on which SLD resolution was proven:

  General form: $F : - F_1, F_2, \ldots, F_n$.

- **To prove** $F$, one must successively **prove** $F_1, \ldots, F_n$. 
Horn clauses

- Facts and rules falls under the definition of **Horn clauses**, the structures on which SLD resolution was proven:

  General form: \( F :­ F_1, F_2, \ldots, F_n. \)

- **To prove** \( F \), one must successively **prove** \( F_1, \ldots, F_n \).

- \( F \) is the clause’s **head**.

- \( F_1, F_2, \ldots, F_n \) constitute the clause’s **tail** or **body**.

- A fact is a clause with an empty tail.
Prolog's strategy

- To answer the question, Prolog builds a search tree:
  - set of possibilities (clauses matching the question) represented as a tree
  - each choice is a node of the search tree
  - trial and error sequential search, following the order of declaration
Prolog's strategy, example

- Query: `?- p(1).`
Prolog's strategy, example

\[\begin{align*}
p(1) & :\neg a(1).
p(1) & :\neg b(1).
a(1) & :\neg c(1).
c(1) & :\neg d(1).
c(1) & :\neg d(2).
b(1) & :\neg e(1).
e(1).
d(3).
\end{align*}\]

- Query: \( ?- \ p(1). \)
Prolog's strategy, example

\begin{itemize}
  \item \textit{Query:} \texttt{?- p(1)}. \\
\end{itemize}

p(1) :- a(1).
p(1) :- b(1).
a(1) :- c(1).
c(1) :- d(1).
c(1) :- d(2).
b(1) :- e(1).
e(1).
d(3).

\begin{tikzpicture}[level distance=1.5cm, sibling distance=1.5cm,]
  \node {p(1)}
    child {node {a(1)}}
    child {node {b(1)}};
\end{tikzpicture}
Prolog's strategy, example

\[
\text{p(1) :- a(1).} \\
\text{p(1) :- b(1).} \\
\text{a(1) :- c(1).} \\
\text{c(1) :- d(1).} \\
\text{c(1) :- d(2).} \\
\text{b(1) :- e(1).} \\
\text{e(1).} \\
\text{d(3).}
\]

- Query: ?– p(1).
Prolog's strategy, example

\begin{verbatim}
\texttt{p(1) :- a(1).}
\texttt{p(1) :- b(1).}
\texttt{a(1) :- c(1).}
\texttt{c(1) :- d(1).}
\texttt{c(1) :- d(2).}
\texttt{b(1) :- e(1).}
\texttt{e(1).}
\texttt{d(3).}
\end{verbatim}

- Query: \texttt{?– p(1).}
Prolog's strategy, example

\[ p(1) :\neg a(1). \]
\[ p(1) :\neg b(1). \]
\[ a(1) :\neg c(1). \]
\[ c(1) :\neg d(1). \]
\[ c(1) :\neg d(2). \]
\[ b(1) :\neg e(1). \]
\[ e(1). \]
\[ d(3). \]

- Query: \(?- p(1).\)
Prolog's strategy, example

\[ p(1) :- a(1). \]
\[ p(1) :- b(1). \]
\[ a(1) :- c(1). \]
\[ c(1) :- d(1). \]
\[ c(1) :- d(2). \]
\[ b(1) :- e(1). \]
\[ e(1). \]
\[ d(3). \]

- Query: \(?- p(1).\)
Prolog's strategy, example

\begin{align*}
p(1) & :\neg a(1). \\
p(1) & :\neg b(1). \\
a(1) & :\neg c(1). \\
c(1) & :\neg d(1). \\
c(1) & :\neg d(2). \\
b(1) & :\neg e(1). \\
e(1). \\
d(3).
\end{align*}

• Query: \(?-\ p(1).\)
Prolog's strategy, example

\[
\begin{align*}
p(1) & :\quad a(1). \\
p(1) & :\quad b(1). \\
a(1) & :\quad c(1). \\
c(1) & :\quad d(1). \\
c(1) & :\quad d(2). \\
b(1) & :\quad e(1). \\
e(1). \\
d(3). \\
\end{align*}
\]

- Query: \( ?- p(1). \)
Prolog's strategy, example

\[ p(1) :\neg a(1). \]
\[ p(1) :\neg b(1). \]
\[ a(1) :\neg c(1). \]
\[ c(1) :\neg d(1). \]
\[ c(1) :\neg d(2). \]
\[ b(1) :\neg e(1). \]
\[ b(1) :\neg d(3). \]
\[ e(1). \]
\[ d(3). \]

- Query: \(?- p(1).\)

with backtracking we can get more answers by using ";"
Prolog's strategy

• The solver follows a *depth-first strategy*
  
  – **success node**: solution found! the solver displays it and stops.
  
  – **failure node**: the solver backtracks up in the tree, until it finds a choice point with unexplored branches.
Prolog's strategy

- If backtracking encounters no choice point left, Prolog stops. No further solution.
- NOTE: Some branches are infinite! So search may not stop...

Take care of:
- The order of goals in clause tails
- The order of clauses
Resolution with variables
Using variables...

- Variables allow us to:
  - compute more than yes/no answers
    ```prolog
    ?- parent(marge, X).
    ```
Using variables...

- Variables allow us to:
  - compute more than yes/no answers
    
    ```prolog
    ?- parent(marge, X).
    ```
  - compress the program.
    
    ```prolog
    parent(marge, lisa) :- child(lisa, marge).
    parent(marge, bart) :- child(bart, marge).
    ...
    parent(X, Y) :- child(Y, X).
    ```
Using variables...

• Variables allow us to:
  
  - compute more than yes/no answers
    
    ?- parent(marge, X).
  
  - compress the program.
    
    parent(marge, lisa) :- child(lisa, marge).
    parent(marge, bart) :- child(bart, marge).
    ...
    parent(X, Y) :- child(Y, X).
  
  - **reversibility** (when declarativity is maintained)
    
    ?- parent(Y, lisa).
Variables matching via Unification

?- brother(bart, Who).

brother(X,Y) :-
    male(X),
    parent(X,Z),
    parent(Y,Z),
    X \== Y.

Unification with
brother(X,Y)
X=bart, Y=Who

male(bart)
parent(bart,Z)
parent(bart,Z)
bart \== Who
Unification examples

- Unification predicate: “=”

  ?- a(B,C) = a(2,3).
  YES {B=2, C=3}

  ?- a(X,Y,L) = a(Y,2,carole).
  YES {X=2, Y=2, L=carole}

  ?- a(X,X,Y) = a(Y,u,v).
  NO
Unification examples

• Unification predicate: "="

?– a(B,C) = a(2,3).
YES  {B=2,  C=3}

?– a(X,Y,L) = a(Y,2,carole).
YES  {X=Y,  Y=2,  L=carole}

?– a(X,X,Y) = a(Y,u,v).
NO

Exercise
Unify \( p(X,b(Z,a),X) \) with \( p(Y,Y,b(V,a)) \)
Unification algorithm

A **free variable** can be seen as a pointer to NIL. When not free, it is said **bound variable**. **Dereferencing** a variable means reaching the value to which it is bound.

```plaintext
procedure unify(t1,t2)

  t3 := dereference(t1); t4 := dereference(t2)
  if t3 is a variable then
      t3 points to t4; return success
  else if t4 is a variable then
      t4 points to t3; return success
  else if t3 is an atom and t4 is an atom then
      if t3 = t4 then return success
      else return fail
  else let t3 = f(t31, .., t3n) and t4 = g(t41, .., t4m)
      if f = g and n = m then
          for i := 1 to n do
              if unify(t3i, t4i) fails then return fail
          return success
      else
          return fail
```

- X and **marge** where X is bound to the value marge will match.
- X and Y where X is bound to marge and Y is bound to marge will match,
- X and **marge** where X is bound to lisa will not match.
Lists
Representing lists

- **Lists** are a crucial data structure in Prolog. They are usually written as:
  
  \[a, b, c, d]\n
  This corresponds to the structured term:
  
  \[a | [b | [c | [d | []]]]]\n
  where \[\] is a special constant the empty list.
Representing lists

- Each list is of the form \([\langle \text{head} \rangle \mid \langle \text{rest\_of\_list} \rangle]\)
  
  \(\langle \text{head} \rangle\) an element of the list (not necessarily a list).
  
  \(\langle \text{rest\_of\_list} \rangle\) is a list (a sub-list).

\[
[a, b, c, d] = [a \mid [b, c, d]]
\]

\[a: \text{head}; [b, c, d]: \text{tail}\]

\[
[a, b, c, d] = [a, b \mid [c, d]]
\]

\_[\_\_] has at least one element.
Cut
Backtracking control using CUT

- Sometimes it is useful to control the backtracking, and this can be done using the ‘!’ operator, the cut operator.
  - once it is executed, it **disallows** backtracking
Backtracking control using CUT

- Sometimes it is useful to control the backtracking, and this can be done using the “!”, the cut operator.

  once it is executed, it **disallows** backtracking

\[
p :\neg \ b_1, \ b_2, \ !, \ a_1, \ a_2, \ a_3.
\]
\[
p :\neg \ r_1, \ r_2.
\]
\[
p :\neg \ r_3.
\]

Before reaching cut, there might be backtracking on \( b_1 \) and \( b_2 \) or trying other rules for \( p \) if one of \( b_1 \) or \( b_2 \) cannot be satisfied. After reaching \( ! \), no more backtracking. The second and third rule will not be searched.
Backtracking control using CUT

- Sometimes it is useful to control the backtracking, and this can be done using the "!" symbol, the cut operator.

- Once it is executed, it disallows backtracking.

```prolog
\[\begin{align*}
p(X, Y) & : - q(X), \\
& \quad !, \\
& \quad r(X, Y). \\
p(X, Y) & : - s(X). \\
q(X) & : - !, r(a, Y).
\end{align*}\]
```
...undermining declarativity

- All times in which we play with the control we are *undermining* the declarative properties of the language.
- Therefore, uses of cut (and as we will see *negation as failure*) or any other properties having side effects remove properties as *reversibility*. 
Prolog and logic
Prolog and Logic

- A Prolog clause is a *generalised disjunction*
  \[ a : : b, c. \]
  
  In logic:
  \[ b \land c\Rightarrow a \]
  
  As material implication \( p \Rightarrow q \) equivalent to \( \neg p \lor q \)
  \[ \neg b \lor \neg c \lor a \]

- Prolog inherits the correctness and completeness proof methods from First Order Logic, and, for the restriction to Horn Clauses, enjoys *decidable algorithms*... More about this the next weeks.
A Prolog clause

\[ a \leftarrow b. \]

In logic would be:

\[ b \Rightarrow a \]

which is equivalent to

\[ \neg a \Rightarrow \neg b \]
Prolog and Logic - 2

- A Prolog clause
  \[ a :\neg b. \]

  In logic *would* be:
  \[ b \Rightarrow a \]

  which is equivalent to
  \[ \neg a \Rightarrow \neg b \]

  but the clause does not specify that!!

...*difference due to the different meaning of negation!*
NAF – Negation as Failure

• In propositional logic, propositions can be **true** and **false**.
NAF – Negation as Failure

- In propositional logic, propositions can be true and false. But propositions could be also unknown.
  
  Example: obviously, it may rain, or not rain. But if I'm indoor, and there is no window, I do not know!
NAF – *Negation as Failure*

- In propositional logic, propositions can be *true* and *false*. But propositions could be also *unknown*.
  
  Example: obviously, it may rain, or not rain. But if I'm indoor, and there is no window, I do not know!

- When the inference engine is not able to assess if the proposition is true, the result will be false. This is called *Negation as Failure* (NAF).
NAF – *Negation as Failure*

• In propositional logic, propositions can be **true** and **false**. But propositions could be also **unknown**.

  Example: obviously, it may rain, or not rain. But if I'm indoor, and there is no window, I do not know!

• When the inference engine is not able to assess if the proposition is true, the result will be false. This is called *Negation as Failure* (NAF).

• In general, there may be two possible negations:
  - strong negation (↔ “classical” negation)
  - NAF negation (↔ “*undecidable*”)
**NAF – Negation as Failure**

- When *NAF implies a strong negation* we are under the **closed-world assumption**.

  If I do not know something (i.e. I cannot infer that something), than that something will be false.
NAF – Negation as Failure

• When *NAF implies a strong negation* we are under the 
closed–world assumption.
    
    If I do not know something (i.e. I cannot infer that something), than 
that something will be false.

• Ex. UFOs do not exist!
NAF – Negation as Failure

• When *NAF implies a strong negation* we are under the 
closed-world assumption.
  If I do not know something (i.e. I cannot infer that something), than 
that something will be false.

• Ex. UFOs do not exist!

  If *it is not the case* 
  that ufos exist, 
  then *it is the case* 
  that ufos do *not* exist.
NAF – *Negation as Failure*

• When *NAF implies a strong negation* we are under the **closed-world assumption**.

  If I do not know something (i.e. I cannot infer that something), than that something will be false.

• Ex. UFOs do not exist!

  If *it is not* the case that ufos exist, then *it is the case* that ufos do *not* exist. **That's the case of Prolog!**
Negation as failure

- Negation as failure can be used to implement *defaults*.

```prolog
fancy(belle_d_argent).
fancy(maximus).

expensive(belle_d_argent).
affordable(Restaurant) :-
    not(expensive(Restaurant)).

?- fancy(X), affordable(X).
   X=maximus
```
Negation as failure

• Negation as failure can be used to implement defaults.

fancy(belle_d_argent).
fancy(maximus).

expensive(belle_d_argent).
affordable(Restaurant) :-
    not(expensive(Restaurant)).

?- fancy(X), affordable(X).
   X=maximus

?- affordable(X), fancy(X).
   FALSE
Negation as failure with CUT

• This predicate behaves just as \texttt{not p(X)}:

\begin{verbatim}
q(X) :- p(X), !, fail.
q(X).
\end{verbatim}
Conclusions
Guidelines

• Go on http://aicourse.r2.enst.fr:4242/SCIA
  – Read and work with the material.
  – Responses to questions will be recorded up to a limit date. (generally 2 weeks from the lecture) and graded.
  – After sending a response you receive a possible correction.

• Try to be in scheduling with the course!

• Collaborative learning is welcome, copying is prohibited!

• Question and comments are welcome, just write me: gsileno@enst.fr