Lecture 2 - Introduction in Prolog

1. Prolog language

The Prolog language (PROgrammation en LOGique) was developed at the University of Marseille around 1970, as a tool for programming and solving problems involving symbolic representations of objects and their relationships.

Prolog has a very wide range of applications: relational databases, artificial intelligence, mathematical logic, proof of theorems, expert systems, solving abstract problems or symbolic equations, etc.

The ISO-Prolog standard.

No standard for object-oriented programming in Prolog, there are only extensions: TrincProlog, SWI-Prolog.

Prolog implementations: Turbo Prolog, Visual Prolog, GNU-Prolog, Sicstus Prolog, Parlog, etc.

We will study the implementation of SWI-Prolog - the syntax is very close to that of the ISO-Prolog standard.

SWI-Prolog - 1986

- provides a two-way interface with C and Java languages
- uses XPCE - an object oriented GUI system
- *multithreading* - based on the multithreading support offered by the standard C language.

Prolog Program

Descriptive character: a Prolog program is a collection of definitions that describe relationships or functions to be calculated - symbolic representations of objects and relationships between objects. The solution to the problem is no longer seen as a step-by-step execution of a sequence of instructions.

- program - collection of logical statements, each being a Horn clause of the form p, $p \rightarrow q$, $p_1 \wedge p_2 \ldots \wedge p_n \rightarrow q$
- conclusion to be demonstrated - in form $p_1 \wedge p_2 \ldots \wedge p_n$

Control structure used by the Prolog interpreter

Is based on logical statements called clauses

- **fact** - what is known to be true
- **rule** - what can be deduced from given facts (indicates a conclusion that is known to be true when other conclusions or facts are true)
- **goal** – conclusion to be proven

- Prolog uses (linear) resolution to prove whether the conclusion (theorem) is true or not, starting from the hypothesis established by the defined facts and rules (axioms).
- Reasoning is applied backwards to demonstrate the conclusion.
- The program is read from top to bottom, from left to right, the search is depth-first and is done using backtracking.

  $\neg p \vee q$ also written as $p \rightarrow q$ is written in Prolog using the clause q : - p. (q if p.)

  $\wedge$ is written in Prolog using ","

$p_1 \wedge p_2 \ldots \wedge p_n \rightarrow q$ is written in Prolog as q:-$p_1, p_2, \ldots, p_n$.

$\vee$ is written in Prolog using ";" or a separate clause.

$p_1 \vee p_2 \rightarrow q$ is written in Prolog

- using the clause q :- $p_1; p_2$.
- using 2 separate clauses

  q:- p1.

  q:- p2.

<u>Examples</u>

Logic	**(SWI-) Prolog**
$\forall x p(x) \wedge q(x) \rightarrow r(x)$	*r(X) :- p(X), q(X).*
$\forall x w(x) \vee s(x) \rightarrow p(x)$	*p(X) :- w(X).* *p(X) :- s(X).*

w$\vee$s$\rightarrow$p ≡ ¬(w$\vee$s)$\vee$p ≡ (¬w$\wedge$¬s)$\vee$p ≡ (¬w$\vee$p)$\wedge$(¬s$\vee$p) ≡ (w$\rightarrow$p)$\wedge$(s$\rightarrow$p)

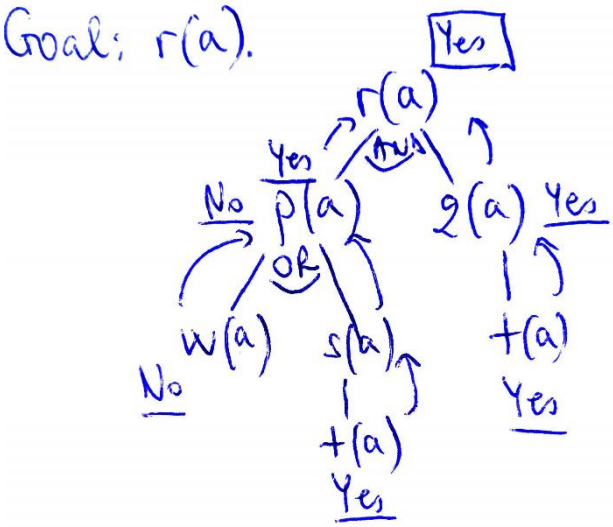
$\forall x t(x) \rightarrow s(x) \wedge q(x)$	*s(X) :- t(X).* *q(X) :- t(X).*
$t(a)$	$t(a)$.
$w(b)$	$w(b)$.

Conclusion	**Goal**
$r(a)$	? $r(a)$. true.
$p(a)$	? *p(a).* true.

r (b)

? r (b).

false



Logic

$\forall x s(x) \rightarrow p(x) \vee q(x)$

Prolog

????

1.1 Basic elements of the SWI-Prolog language

1. **Term**

- **SIMPLE**
  a. constant
    - symbol
      - sequence of letters, numbers, _
      - starts with a lowercase letter
    - number = integer, real (number)
    - string: 'text' (character: 'c', '\ t', ...)

ATOM = SYMBOL + NUMBER + STRING + SPECIAL-CHARACTERS + [ ] (empty list)

- special characters + * / <> =:. & _ ~

  b. variable
    - sequence of letters, numbers, _
    - starts with a capital letter
    - the anonymous variable is represented by the underline character (_).

- **COMPOUND**
  - lists are a special class of compound terms

2. **Comment**

% This is a comment
/ * This is a comment * /

3. **Predicate**

a) standard (ex: fail, number, ...)
b) user

name [(object [, object ....)]
the symbolic name of the relationship

Types

number (integer, real)
atom (symbol, string, special-string)
list (sequence of items) specified as list = base_type *

ex. (homogeneous) list of integers [1,2,3]

% define type: el =integer list = el *

!!! the empty list [] is the only list that is considered in Prolog as an atom.

<u>Convention</u>.

In SWI-Prolog there are no predicate signature declarations, nor domain / type statements (eg as in Turbo-Prolog).

specifying a predicate through comments

% define types, if applicable
% name [(param1: type1 [, param2: type2 ...)]
% predicate flow model (i, o, ...) - see Section 4
% param1 - the meaning of parameter 1
% param2 - the meaning of parameter 2
….

4. Clause

- fact
  - relationship between objects
  - *predicate_name*[(object [, object ....)]
- rule
  - allows deduction of facts from other facts

Example:
consider the predicates

father(X, Y) representing the relation "Y is the father of X"
mother(X, Y) representing the relation "Y is the mother of X"

and the following facts corresponding to the two predicates:

```
mother(a, b).
mother(e, b).
father(c, d).
father(a, d).
```

Goal: using the above definitions define the predicates

parent(X, Y) representing the relation "Y is the parent of X"
sibling(X, Y) representing the relation "Y is the sibling of X"

Clauses in SWI-Prolog

```
parent (X, Y) :- father (X, Y).
parent (X, Y) :- mother (X, Y).
```

% "\=" Represents the "different" operator - see Section 1.5

```
sibling (X, Y) :-    parent (X, Z),
                     parent (Y, Z),
                     X \= Y.
```

5. <u>Question</u> (goal)

it is of the form

predicate1 [(object [, object ....)], predicate2 [(object [, object ....)] .... .

CWA - Closed World Assumption

- lack of knowledge implies falsity
- a statement that is true is known to be true
- a statement not known to be true is considered false
- **false** = negation as a failure

Using the above definitions, we ask the following questions:

```
?- parent (a, b).
true.

?- parent (a, X).
X = d;
X = b.

?- parent (a, f).
false.

?- sibling (a, X).
X = c;
X = e.

?- sibling (a, _).
true.
```

1.2 Matching. How do variables receive their values?

Prolog has no assignment instructions. Variables in Prolog receive their values by matching constants from facts or rules.

Until a variable receives a value, it is free; when the variable receives a value, it is bound. But it is bound as long as needed to find a solution to the problem. Then, Prolog unbinds it, backtracks and looks for alternative solutions.

Remark. It is important to note that information cannot be stored by assigning values to variables. Variables are used as part of a matching process, not as a type of information storage.

What is a match?

Here are some rules that will explain the term 'match':

1. Identical structures match together
   - p (a, b) matches p (a, b)
2. Usually a match involves free variables. If X is free,
   - p (a, X) matches p (a, b)
   - X is bound to b.
3. If X is bound, it behaves like a constant. With X bound to b,
   - p (a, X) matches p (a, b)
   - p (a, X) does NOT match p (a, c)
4. Two free variables match each other.
   - p (a, X) matches p (a, Y)

Remark. The mechanism by which Prolog tries to 'match' the part of the question it wants to solve with a certain predicate is called unification.

Flow models

In Prolog, variable bindings are done in two ways: when entering the clause or when exiting the clause. The direction in which a value binds is called a 'flow model. When a variable is given at the input of a clause, this is an input parameter (i), and when a variable is given at the output of a clause, it is an output parameter (o). A certain clause can have several flow models. For example the clause

factorial (N, F)

can have the following flow models:

- (i, i) - check if N! = F;
- (i, o) - assign F: = N !;
- (o, i) - find that N for which N! = F.

Remark. The property of a predicate to work with multiple flow models depends on the programmer's ability to program the predicate properly.

Remark. The predicate **var(X)** returns **true** if X is a free variable and **false** otherwise.

1.3 Syntax of rules

The rules are used in Prolog when a fact depends on the success (validity) of other facts or successions of facts. A Prolog rule has three parts: the head, the body, and the if (:-) symbol that separates the two.

Here is the generic syntax of a Prolog rule:

```
head of rule: -  subgoal,
                 subgoal,
                 ...,
                 subgoal.
```

Each subgoal is a call to another Prolog predicate. When the program makes this call, Prolog tests the called predicate to see if it can be true. Once the current subgoal has been satisfied (it has been found to be true), it returns and the process continues with the next subgoal. If the process was successful, the rule succeeded. To successfully use a rule, Prolog must satisfy all of its sub-goals, creating a consistent set of variable bindings. If a subgoal fails (found false), the process returns to the previous subgoal and looks for other variable bindings, and then continues. This mechanism is called backtracking.

1.4 Equality operators

X = Y checks if X and Y can be unified

If X is a free variable and Y is bound, or Y is a free variable and X is bound, the sentence is satisfied by unifying X with Y.
If X and Y are bound variables, then the sentence is satisfied if the equality relation takes place.

```
?- [a, b] = [a, b].
true.

?- [X, Y] = [a, b].
X = a,
Y = b.

?- [a, b] = [X, Y].
X = a,
Y = b.
```

X \= Y checks if X and Y cannot be unified
\ + X = Y

?- [X, Y, Z] \= [a, b].
true.

?- [X, Y] \= [a, b].
false.

?- [a, b] \= [X, Y].
false.

?- \+ a = a.
false.

?- \+ [X,Y]=[a,b].
false.

?- \+ [a,b]=[X,Y,Z].
true.

X == Y checks if X and Y are bound to the same value.

?- [2,3] == [2,3].
true.

?- a == a.
true.

?- R == 1.
false.

X \== Y checks if X and Y have not been bound to the same value.

?- [2,3] \== [3,2].
true.

?- a \== a.
false.

?- R \== 1.
true.

1.5 Arithmetic operators

!!! Important

- 2 + 4 is just a structure, its use does not perform the assembly
- Using 2 + 4 is not the same as using 6.

Arithmetic operators

=, \=, ==, \== see section above

?- 2 + 4 = 6.
false.

?- 2 + 4 \= 6.
true.

?- 6 == 6.
true.

?- 6 \= 7.
true.

?- 6 == 2 + 4.
false.

?- 2 + 4 = 2 + 4.
true.

?- 2 + 4 = 4 + 2.
false.

?- X = 2 + 4 - 1.
X = 2 + 4 - 1.

=:= • tests arithmetic equality
• forces the arithmetic evaluation of both sides
• operands must be numeric
• variables are BOUND

=\= tested "different" arithmetic operator

?- 2 + 4 =:= 6. ?- 2 + 4 =\= 7. ?- 6 =:= 6.
true. **true.** **true.**

is • the right side is BOUND and numeric
• the left side must be a variable
• if variable bound, check numerical equality (like =:=)
• if variable not bound, evaluate the right side and then the variable is related to the result of the evaluation

?- X is 2 + 4 - 1. ?- X is 5.
X = 5 X = 5

Inequalities

< smaller than
=< less than or equal to
> greater than
>= greater than or equal to

• evaluates both sides
• BOUND variables

?- 2 + 4 =< 5 + 2. ?- 2 + 4 =\= 7. ?- 6 =:= 6.
true. **true.** **true.**

Some predefined arithmetic functions

mod (X, Y)	
X mod Y	returns the rest of dividing X by Y
div (X, Y)	
X div Y	
X//Y	returns the quotient of dividing X by Y
abs (X)	returns the absolute value of X
sqrt (X)	returns the square root of X
round (X)	returns the value of X rounded to the nearest integer
truncate(X)	convert float to integer, delete the fractional part
floor(X) `	round down
ceiling(X)	round up
random (L,H,X)	return a random value between L and H
between (L,H.X)	return all values between L and H

Standard order of terms

When comparing and unifying arbitrary terms, terms are ordered in the so-called “standard order". This order is defined as follows:

1. *Variables* < *Numbers* < *Strings* < *Atoms* < *Compound terms*
2. Variables are sorted by address (i.e. by age, *older* are ordered before *newer*).
3. *Numbers* are compared by value.
4. *Strings* are compared alphabetically.
5. *Atoms* are compared alphabetically.

6. *Compound terms* are first checked on their arity, then on their functor name (alphabetically) and finally recursively on their arguments, leftmost argument first.

@< The left term is before the right term in the standard order of terms

@=< Both terms are == or the left term is before the right term in the standard order of terms

@> The left term is after the right term in the standard order of terms

@>= Both terms are == or the left term is afterthe right term in the standard order of terms

Examples:

```
?- 1 @< 2.          ?- 1 < 2.
true                true

?- 9 @< a.          ?- Z @< a.  % free variables sort before all
true                true        % variables sort before

?- 1+2 < 5.         ?- 1+2 @< 5.
true                false       % 1+2 is compound

?- 9 @< '0'.        ?- "Z" @< 'A'.
true.               true.

?- Z @< "A".        ?- 1 + 2 @< 1 - 2.
true.               true.
```

2. Example - the "factorial" predicate

Given a natural number n, calculate the factorial of the number.

$$fact(n) = \begin{cases} 1 & if\ n = 0 \\ n \cdot fact(n-1) & otherwise \end{cases}$$

In Version 2, ! represents the predicate "cut" (red cut, in this context), used to prevent backtracking to the next clause.

Version 1

```
% fact1(N: integer, F: integer)
% (i, i) (i, o)

fact1(0, 1).
fact1(N, F) :-
        N > 0,
        N1 is N-1,
        fact1(N1, F1),
        F is N * F1.

go1 : - fact1(3, 6).
```

Version 2

```
% fact2(N: integer, F: integer)
% (i, i) (i, o)

fact2(0, 1) :- !.
fact2(N, F) :-
        N1 is N-1,
        fact2(N1, F1),
        F is N * F1.
```

go2 : - fact2 (3, 6).

Version 3

```
% fact3(N: integer, F: integer)
% (i, i), (i, o)

fact3(N, F) :-
        N > 0,
        N1 is N-1,
        fact3(N1, F1),
        F is N * F1.
fact3(0, 1).
```

?- fact3 (3, N).
N = 6.