NOVA IMS Information Management School

AI

Artificial Intelligence

Knowledge Representation and Reasoning
PROLOG

Marco Antonio Silva
madasi@microsoft.com

Instituto Superior de Estatística e Gestão de Informação
Universidade Nova de Lisboa

Welcome to Prolog world !!!☺

Logic Programming

For each language, the way of assigning a value or even comparing data changes. This change is simple for those who know at least one identical language, ie: for those who often program with Imperative or Object Oriented languages.

For instance:

C	**for (i=1; i=10; i++) {printf("%d ", i);}**
C#	**for (int i = 0; i < 11; i++) { Console.WriteLine(i); }**
Java	**for (int i = 0; i < 11; i++) {System.out.println("i: " + i); }**

However, when we feel the need to represent certain relations that are not imperative but declarative, we may have a problem.

How to represent, in a language like C, C # or Java, the relation X is a child of Y?

What if we want to know who are the children of Y?

Logic Programming

Everything is complicated. Switching from an imperative language to a logical and declarative language facilitates the process of implementing and handling data.

However, by having these benefits, we can no longer enjoy some of the advantages of imperative languages.

We will see throughout this introduction the advantages and disadvantages of using a declarative and logical language through various examples and implementations in Prolog.

Logic Programming

Fundamental Idea:

Declarative phrase of Form P if Q and R

(can be interpreted procedurally as: To solve P, solve Q and R)

Program: Set of Axioms

Computation: Constructive Proof of a Sentence from the Program

Basis: Unification Algorithm + Principle of Resolution (Robinson, 1965)

J. Alan Robinson (1965), A machine-oriented logic based on the resolution principle. *Journal of the ACM*, Volume 12, Issue 1, pp. 23–41.

An introduction to Prolog

- The PROLOG Language was created in the 1970s by Alain Colmareur at the University of Marseille.
- The name of the language comes from PROgramming in LOGic, (follows the logic programming paradigm)
- It is in conjunction with the LISP language, created in the 1950s, one of the specific languages for the development of Artificial Intelligence Systems.
- While LISP has had an impact on the US, PROLOG has achieved notoriety in Europe and Japan.
- The main PROLOG standard was proposed in Edinburgh by Clocksin & Mellish.

Source: www.dei.isep.ipp.pt/~jtavares

Phrases

- Facts
- Rules
- Questions

A program is composed by Facts and Rules.

An unique data Structure: **Terms**

Terms

- Constants
- Variables
- Compound Terms

An introduction to Prolog

- Constants

Constants are also known as atoms and are the simplest ways to represent data. They usual start with a character which must be lower case. Some examples:

simple constant: **marco**

connection between words with underscores is allowed. eg: **marco_silva**

The quotation marks can delimit the constants. eg: ‘marco silva’ and also, allows start a constant with capital letters and have spaces. eg:**'Marco Silva'**.

An introduction to Prolog

- **Logical Variables**

- Variables names usually start with an uppercase letter, but they also may start with an underscore. Some examples:

- simplest variable: X
- with underscore: _X
- with underscore, but the second one is lowercase: _x = 9,
- underscore works as a special name variable: _ = 9.

Logic Variable → Unspecified Entity

Conventional Variable → Single Memory Location

An introduction to Prolog

Compound Terms

A compound term -> Functor + Sequence of arguments (terms)

$f(t_1, t_2, \ldots t_n)$

Examples of terms:

wine(green)

name (john, morse)

etc (X)

list (a, (list (b, nil))

a (a (nil (2, nil), X, a (Y, 1, nil))

Functor: Name and Arity

Ex: wine/1 name/2 a/3

A term where there are no variables is called a **closed term.**

An introduction to Prolog

- **Facts**

The simplest expression we can find in a logical program is the **fact**.

One **fact** translates a relationship between objects.

An example is: **father(abraham, isaac)** <- abraham and isaac are two constants

This fact tells us that **abraham** is the father of **issac**. It consists of two arguments and ends with a period. It should be noted that the fact presented does not reflect the inverse relationship and should be read only as stated above.

Other facts can be defined:

son(isaac, abraham).	- Son relationship.
mother(joaquina, isaac).	- Mother relationship
macho(isaac).	- Isaac is a macho.
macho(abraham).	- Abraham is a macho.
female(joaquina),	- Joaquina is a female.

A finite set of facts is a program. This is the simplest form of a logic program.

An introduction to Prolog

- **Question**

The second basic expression that exists in a logic program is the **question** which, in English terminology, is known as query.

Answering a question is equivalent to determining if the question is a logical consequence of the program. Asking a question reflects the need for an answer to the previously acquired knowledge.

An example:

- female(joaquina).

The question asked the system is whether joaquina is female, and the answer in this particular case will be:

yes.

An introduction to Prolog

A more elaborate question, which reflects the need to know the males in the knowledge base, might be:

- male (X). - X is a variable.

X = abraham ->

A question always ends with two possible answers: If the theorem (question) is demonstrable in a given context, then the answer is **yes**. Otherwise, the answer will be **no**.

Along the way are some of the intermediate steps, where the answers are none of the above. This situation fits the previous question:

- macho(X).

The first answer results from the unification of X - variable - with the constant that belongs to the first fact to be found in the database..

An introduction to Prolog

Two situations can happen: or the answer is considered satisfactory and the final answer will be **yes**.

X = abraham

yes

Or, if the intended answer is the visualization of all males, then one can proceed (using the (;) key in some tools).

X = abraham→;

X = isaac →;

no

The answer ends with **no**, because there are no more males at the base.

A negative answer means the impossibility of proving. Not to be confused with proof of falsehood.

An introduction to Prolog

Substitution: finite set of pairs $x_i = t_i$ where x_i is variable and t_i a term, $x_i \neq x_j$ for all $i \neq j$ and x_i does not occur in t_i for any I and j.

Applying Substitutions to Terms:

Application from θ to T: $T\theta$

$T\theta$ - term obtained by substituting each occurrence of X in T for t, for all pair x = t in θ.

Eg: father (abraham, X) {X = isaac}

= father father(abraham, isaac)

Instance: A is an instance of B if there is a substitution θ such that A = Bθ

Existential Questions

father(abraham, X)? is equivalent to $\exists$ X: father(abraham, x)

Computation for existential questions: find a fact that is the instance of the question.

An introduction to Prolog

Universal facts

nul(0, X, 0).

Ad(0, X, X).

Q: ad(0, 2, 2) ?

A: yes

C is a common instance of A and B if C is an instance of A and C is an instance of B.

$C = A\theta_1 = B\theta_2$

Q: nul(A, 3, B) ?

A: {A=0 , B=0} ----- > nul(0, 3, 0)

- **Rules**

Rule = Logical Axiom

A rule may or may not use facts. Normally, rules are predicates that allow us to relate the various logical relationships that exist in the database / knowledge.

Logic Program = Finite Set of Rules

An introduction to Prolog

A Simple Abstract Interpreter

A "Goal" G is a logical consequence of a Program P if there is in P a clause with a closed instance A ← B_1,…., B_n ($n \geq 0$) such that B_1,…., B_n are logical consequences of P and A is an instance of G.

Inputs:	A **P** Program
	A closed Goal **G**
Outputs:	**Yes,** if G is proved from P
	No, otherwise
State:	The solver (current value of G)

Example: To know who is descended from whom, let's use a database we have been developing, and try to build the rule that will allow us to answer this question.

An introduction to Prolog

- **Rules**

The process of implementing such a response mechanism relies on writing the descendant relationships of objects belonging to the knowledge base.

It becomes obvious this solution is not at all preferred as it is very complex. The set of rules that can answer the question could be something like:

- a descendant is someone who is a descendant of his father,

- a descendant is someone who is a descendant of his father's father,

- a descendant is someone who is a descendant of his father's father's father,

- a descendant is someone who is a descendant of the father of the father of his father's father,

- a descendant is someone who is a descendant of his father .... of his father.

An introduction to Prolog

The recursion relation that exists in these sentences is evident. Most of the rules are built using recursion.

Making this interminable relationship better, the direct descendant is the one who is a father child, and in the form of a rule:

- **X is** descendant of **Y,** if **X is** son of **Y.**

In Prolog:

```
descendant (X, Y) :-  son (X, Y).
```

So what is a son? Is the son predicate set? The answer is negative. So what is a son? Someone is someone else's child if the that person is the father of the first.

- **X is son of Y, if Y is the father of X.**

An introduction to Prolog

In Prolog would be:

son(X, Y) :- father(Y, X).

Defining the child relationship from the parent relationship, we can now ask the question: who is a direct descendant of abraham?

- descendant(abraham, X).

X = issac

If we want to know who all Abraham's descendants are, we must complete the descending rule by constructing another rule:

descendant(X, Y) :- father (Z, X), descendant(Z, Y).

This rule tells us that X is only descended from Y if X is the parent of someone who, in turn, is descended from Y.

An introduction to Prolog

descendant(X, Y) :- father (Z, X), descendant(Z, Y).

The recursion problem of this rule, which will allow the search for all descendants of X, is: "how will the rule stop"?

Or in other words: what is the stopping condition of the rule? The answer is quite simple: the rule stops when finds a direct descent relationship, that is: when find or not the direct descendant of Y.

The first rule is known as the stopping condition and the second is the one that allows us to know everything about the offspring relationship. The complete theorem is:

descendant (X, Y):- son(X, Y).

descendant(X, Y):- son(X, Z), descendant(Z, Y).

The stopping condition must be the first to be written, as it is the first to be sought by the interpreter.

An introduction to Prolog

To test these rules, let's make a little debug by adding some new facts:.

son (teresa, grandfather panda).
son (joana, grandfather panda).
son (alice, joana).

The question to demonstrate is: from whom descended alice? descendant(alice, X).

The debug will be built on a proof tree of the theorem to be demonstrated.
- descendant(alice, X).

Path I:
son(alice, X)
son(alice, joana)
descendant(alice, joana).

An introduction to Prolog

Path 2

son(alice, X)
descendant(joana, X)
son(joana, X)
son(joana, grandfather panda)
descendant(joana, grandfather panda)
descendant(joana, grandfather panda).

Answers :

X = joana →; .
X = grandfather panda →
yes.

An introduction to Prolog

Path 2

son(alice, X)
descendant(joana, X)
son(joana, X)
son(joana, grandfather panda)
descendant(joana, grandfather panda)
descendant(joana, grandfather panda).

Answers:

X = joana →; .
X = grandfather panda →
yes.

An introduction to Prolog

Example;

son(X,Y) ← father(Y,X), malec(X).
daughter(X,Y) ← father(Y,X), fem(X).

avô(X,Y) ← father(X,Z), father(Z,Y).
avô(X,Y) ← father(X,Z), mother(Z,Y).

<u>Declarative Reading (Rule = Logical Axiom)</u>

$\forall x \forall y(father(Y,X) \wedge male(X) \rightarrow son(X,Y))$

"X is Y's son if Y is X's father and X is male)

<u>Procedural Reading</u>

"To determine if X is son of Y, determine if Y is father of X and determine if X is male"

An introduction to Prolog

- **Some fundamental symbols and predicates**

How to represent the relationships of the logical operators “and” and “or”

- **“And” operator**

- X if a and b and c. This means that X will only happen if a and b and c are true. In Prolog, such situation is implemented with one rule:

x:-

a,

b, b is performed only if a is true.

c. c is performed only if b is true.

An introduction to Prolog

- **"Or" operator**

x if a or b or c. It means that x only happens if a or b or c is true. Pure Prolog writing of this notation is accomplished with three predicates:

```
x:-     a.
x:-     b.
x:-     c.
```

Another way to represent this operation is (only supported on some compilers) ´use the symbol ";"

```
x:-
a;
b;
c.
```

The first form should be preferred over the second because of its portability.

An introduction to Prolog

- ***Back-Tracking* and *Cut operator (!)***

Back-tracking is a mechanism of logical languages. When executing several lines of code in a predicate, it may happen that one of them is not true.

Let's imagine the previous case and the operation "**and**". After successfully executing **a**, **b** fails. Automatically, **a** runs again. There was a back-tracking phenomenon. Another attempt will be made trying demonstrate **a** (but not by the same criteria as above). If not possible, the predicate assumes failure (no); otherwise, **b** will be tried again.

The demonstration process will be as the previous.

But this effect is not always desired and to avoid this we can use the operator ! (cut) whose function is to prevent back-tracking.

In the "family example" (son relation, descendant), we can observe several phenomena followed by back-tracking.

An introduction to Prolog

An example of using "Cut" is the procedure to calculate the maximum between 2 numbers.

```
max(X,Y,Max).

max(X,Y,X):- X >= Y
max(X,Y,Y):- X <  Y
```

These rules are mutually exclusive (<u>if one succeeds the other fails</u>)

It would be better to write If X> = Y then Max = X but Max = Y

So, using Cut we got →

```
max(X,Y,X):- X >= Y, !
max(X,Y,Y).
```

An introduction to Prolog

- ***Comments***

Comments in Prolog are achieved in two ways: the first is using the % symbol and the second is identical to the C language.

```
1, % this is a comment
2. /* and this too  :P */
```

- ***Assignments***

The assignments are slightly different. Thus, to assign a value to a variable the reserved word **is** must be used. Consider the following example:

```
A is 3+4, the result is A = 7
A = 3+4, the result A = 3+4
```

The **=** operator assigns the same value to the variable. The variables have the property of not obeying types, thus dispensing with their declarations.

Some arithmetic operations & functions

Operator	Meaning	Example
+	addition	2 is 1 + 1.
–	subtraction	1 is 2 – 1.
–	unary minus	Try the query X is 1, Y is - X.
*	multiplication	4 is 2 * 2.
/	division	2 is 6 / 3.
//	integer division	1 is 7 // 4.
mod	integer remainder	3 is 7 mod 4.
**	exponentiation	1.21 is 1.1 ** 2.

Function	
abs(Exp)	absolute value of Exp : i.e. Exp if Exp ≥ 0, –Exp if Exp < 0
atan(Exp)	arctangent (inverse tangent) of Exp : result is in radians
cos(Exp)	cosine of the Exp : Exp is in radians
exp(Exp)	eExp : e is 2.71828182845…
log(Exp)	natural logarithm of Exp : i.e. logarithm to the base* e
sin(Exp)	sine of the Exp : Exp is in radians
sqrt(Exp)	square root of the Exp
tan(Exp)	tangent of the Exp: Exp is in radians
sign(Exp)	sign (+1 or –1) of the Exp: sign(–3) = –1 = sign(–3.7)
float(Exp)	float of the Exp: float(22) = 22.0 - see also float the predicate
floor(Exp)	largest integer ≤ Exp: floor(1.66) = 1
truncate(Exp)	remove fractional part of Exp: truncate(–1.5) = –1, truncate(1.5) = 1
round(Exp)	round Exp to nearest integer: round(1.6) = 2, round(1.3) = 1
ceiling(Exp)	smallest integer ≥ Exp: ceiling(1.3) = 2

Numerical Relational Predicates in Prolog

Comparison	Definition
X = Y	succeeds if X and Y unify (match) in the Prolog sense
X \= Y	succeeds if X and Y do not unify; i.e. if not (X = Y)
T1 == T2	succeeds if terms T1 and T2 are identical; e.g. names of variables have to be the same
T1 \== T2	succeeds if terms T1 and T2 are not identical
E1 =:= E2	succeeds if values of expressions E1 and E2 are equal
E1 =\= E2	succeeds if values of expressions E1 and E2 are not equal
E1 < E2	succeeds if numeric value of expression E1 is < numeric value of E2
E1 =< E2	succeeds if numeric value of expression E1 is ≤ numeric value of E2
E1 > E2	succeeds if numeric value of expression E1 is > numeric value of E2
E1 >= E2	succeeds if numeric value of expression E1 is ≥ numeric value of E2
T1 @< T2	succeeds if T1 is alphabetically < T2
T1 @=< T2	succeeds if T1 is alphabetically ≤ T2
T1 @> T2	succeeds if T1 is alphabetically > T2
T1 @>= T2	succeeds if T1 is alphabetically ≥ T2

An introduction to Prolog

Predicates Input/Output and manipulation of the database

- Data writing and reading (screen and keyboard) predicates:
  - **read(Term)** - read a stdin term/constant (requires . after term)
  - **write(Term)** - write a term / constant to stdout (does not require. after term)

- Predicate to change lines : **nl**

- Character Code Manipulation Predicates:
  - **get(X)** - read character whose ASCII code is X
  - **put(X)** - write character whose ASCII code is X
  - **name(C,L)** - associa a constante C à lista L dos códigos dos seus caracters

Ex: name(g,X) ? X = [103]

name(X, [80,114,11,108,108,11,103] ? X = ´Prolog´

An introduction to Prolog

A database in is a special set of relationships between data.

In Prolog a database is a set of Fact and Rules (clauses).

During the execution of a program it is possible to add new clauses or remove existing clauses.

The database is handled as simply as possible. There are four basic predicates:

- **retrat (clause)** - which removes the first occurrence of a "clause" in the database
- **assert(clause)** - that inserts a "clause" in the database

  Ex: assert((faster(X,Y):- fast(X),slow(Y)).
- **asserta(clause)** - that inserts a "clause" in beginning of the database
- **assertz(clause)** - that inserts a "clause" in end of the database

An introduction to Prolog

- ***Other constants and predicates***
  - **not(X) -** fails if X is satisfied and succeeds in the opposite case
  - **fail -** always fails
- ***if_then_else***

The programmer who is used to imperative languages may feel a bit unprotected due to the lack of some control structures in a logical program that he often uses in the implementation of his programs. This is the case of the for loop and the if .. then ... else structure.

The if .. then ... else structure refers to the complete case.

A simpler structure: if .... then.

An introduction to Prolog

The simplest structure:

if_then(A, B) If A, then execute B (B can be several instructions)

A,

B. B only runs if successfully passing A

if_then(_,_) The predicate must return true even if A is false.

Hypothetical questions could be:
if_then (true, write (true)) - Writes true
if_then (false, write (false)) - Do not write anything (enter second rule)
if_then (true, (nl, write (true), nl)) - Change line, type true change line.

An introduction to Prolog

The second control structure is more complete:

if_then_else(A, B, C):- A, B. If A is true then execute B

if_then_else(A, B, C):- C. If not, execute C.

Another representation allowed by some compilers is:

```
if_then_else(A, B, C):-
A,
B;                          "Or" Operator
C.
```

Example:

```
if_then_else(true, write(true), write(false)).      R: true
if_then_else(false, write(true), write(false)).     R: false
```

For cycle

The ***for cycle*** can replace all known cycles (while, do .. while, Repeat,…).

Its implementation is possible in a logical language, although it slightly violates its principles..

```
for(N, N, N):- !.
for(N, A, N).
for(N, A, B):- N1  is N + 1,  for(N1, A, B).
```

Stop condition

An introduction to Prolog

Debug of this predicate for the question

for (1 3, I), write (I) - Write from 1 to 3 for(1, 3, 1)

write(1) *Clicking on; another iteration is made*

for(1, 3, 1)

2 is 1 + 1

for(2, 3, 2)

for(1, 3, 2)

write(2) *Clicking on; another iteration is made*

for(1, 3, 2)

for(2, 3, 2)

3 is 2 + 1

for(3, 3,_)

! *Cut, stop backtracking*

for(3, 3, 3)

for(2, 3, 3)

for(1 3, 3)

write(3) *End of the cycle*

Bibliography

- **I Bratko, "Prolog Programming for Artificial Intelligence (4th ed.)", Addison-Wesley, 2014**
- Leon Sterling and Ehud Shapiro, The Art of Prolog
- Mathematical Logic for Computer Science , M. Ben-Ari, 2001, Springer-Verlag
- Logic in Computer Science: Modelling and Reasoning about Systems , M. Huth e M. Ryan, 2004, Cambridge University Press
- Visual Prolog Reference Guide and resources:

  http://wiki.visual-prolog.com/

  http://www.visual-prolog.com/