PCS 2428 / PCS 2059
Inteligência Artificial

Prof. Dr. Jaime Simão Sichman
Profa. Dra. Anna Helena Reali Costa

Linguagem Prolog

(Luis Gustavo Nardin)

Agenda

1. Introduction
2. Prolog Syntax
3. Clauses
4. How Prolog Answer Queries
5. Backtracking
6. List
7. Operator
8. Arithmetic
9. How to Run Prolog
10. Bibliography

Introduction

- The first, official version of Prolog was developed
  - at the University of Marseille, France
  - by Alain Colmerauer
  - in the early 1970s
  - as a tool for PROgramming in LOGic
- Preferred for AI programming and mainly used in such areas as:
  - Theorem proving, expert systems, NLP, …
- Logical programming is the use of mathematical logic for computer programming.

Introduction

- Used for symbolic and non-numerical computation
- Has a built in intelligent search mechanism
- Can handle complex problems in compact programs
- Writing a program in Prolog means writing facts and rules which together comprise knowledge base
- Facts and rules use predicates which represent relationships among data objects.

Introduction

- For symbolic, non-numeric computation
  - e.g. : **parent (tom, bob).**

  Parent is a relation between its parameters: **tom** and **bob**
- The whole thing is called a clause
- Each clause declares one fact about a relation
- Prolog is a **Declarative Language**

Introduction

- **Declarative Language** means that
  - The programmer
    - declares **facts**
    - defines **rules** for reasoning with the facts
  - Prolog uses deductive reasoning to
    - determine new facts from old
    - decide whether a proposed fact (**goal**) can be logically derived from known facts

      (such a decision is called a **conclusion**)

Prolog Syntax

- **Terms in Prolog**
  - Comments
  - Simple
    - Constants:
      - Atoms
      - Numbers
        - » Integer
        - » Real
    - Variables
  - Complex Structures

Prolog Syntax

- **Comments**
  - **Multi-line**

    /* This is a comment
    This is another comment ***/**

  - **Short**

    % This is also a comment

Prolog Syntax

English	Predicate calculus	Prolog
And	∧	,
Or	∨	;
Onlyif	⇐	:-
Not	¬	not

Prolog Syntax

- **Conjunction and Disjunction**
  - Conjunction → **,**
  - Disjunction → **;**
    - **P :- Q ; R.**
    - **P :- Q**
    - **P :- R**
  - '**,**' has more priority
    - **P :- Q , R ; S , T , U .**
    - **P :- (Q , R) ; (S , T , U) .**

Prolog Syntax

- **Atoms**
  - Strings of letters, digits, underscore character starting with lower case letter:

    sarah_jones, x25, x__y, x_yAB
  - String of special characters:

    <-->, ===>, ….
  - Strings of characters enclosed in single quotes:

    'India', 'Tom'

Prolog Syntax

- **Numbers**
  - Include integers and real numbers

    1, 3131, -0.0035, 3.14
- **Variables**
  - String of letters, digits and underscore characters that starts *either* with an upper-case letter *or* with an underscore:

    Y, Child, _a23, Student_List

Prolog Syntax

- **Structures**
  - Objects that have many components
  - Components can themselves be structures
  - *Functor* is used to combine components into single structure

    date(1, jan, 2007), date(Date, Month, 2007)

    date(31, cat, -4.3), segment(point(1,1),point(3,3))
  - *Functors* are recognized by:
    - Name
    - Number of arguments (Arity)

Prolog Syntax

- **Predicate**
  - A predicate consists of a head and a number of arguments
  - Is a function which returns true/false
  - For example:

    father(sam, pat). %sam is father of pat

Clauses

- There are three categories of clauses in Prolog:
  - **Facts:** Those are true statements that form the basis for the knowledge base.
  - **Rules:** Similar to functions in procedural programming (C++, Java…) and has the form of if/then.
  - **Queries:** Questions that are passed to the interpreter to access the knowledge base and start the program.

15

Clauses

- **Facts**
  - A fact is a one-line statement that ends with a full-stop.
    - **male(terach).**
    - **male(abraham).**
    - **female(sarah).**
    - **female(milcah).**
    - **father(haran, milcah).**
    - **father(haran, yiscah).**
    - **mother(sarah, issac).**

16

Clauses

- **Rules**
  - A Rule consists of
    - a condition part (right-hand side) → **body of clause**
    - a conclusion part (left-hand side) → **head of clause**
    - They are separated by ':-' which means 'if'
  - Parent relation
    - parent(X, Y) : X is a parent of Y
    - ∀ X,Y (parent(X, Y) ← father(Y, X) ^ male(Y))
    - parent(X, Y) :- father(Y, X), male(Y).

      head (parent(X, Y)) body (father(Y, X), male(Y))

17

Clauses

- **Rules**
  - Variables in head of rules are universally quantified
  - Variables appearing only in the body are existentially quantified
  - **Rules** vs. **Facts**
    - A Fact is something unconditionally true
    - Rules specify things that are true if some condition is satisfied

18

Clauses

- **Recursive Rules**
  - The recursion in any language is a function that can call itself until the goal has been succeed.
  - In Prolog, recursion appears when a predicate contain a goal that refers to itself.
  - In Prolog, a recursive definition always has at least two parts. A first fact that act like a stopping condition and a rule that call itself simplified. At each level the first fact is checked. If the fact is true then the recursion ends. If not the recursion continue.
  - A recursive rule must never call itself with the same arguments.

Clauses

- **Recursive Rules**

  Example: Ancestor
  - Define ancestor relation based on parent relation.

```
ancestor (X, Z) :-
        parent (X, Z).
ancestor (X, Z) :-
        parent (X,Y) , parent (Y, Z).
ancestor (X, Z) :-
  parent (X, I) , parent (I, Y) , parent (Y, Z).
```

  - Solution is Recursion

20

Clauses

- **Recursive Rules**

  Example: Ancestor
  - Rules in Prolog are like functions in procedural programming languages
  - For recursion we should define the ancestor relation in terms of itself
  - Base Case :
    ancestor(X, Z) :- parent (X, Z).
  - Recursion Step :
    ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z).

21

Clauses

- **Recursive Rules**

  Example: Factorial
  - The best way in Prolog to calculate a factorial is to do it recursively.

```
factorial(0,1).
factorial(X,Y) :-
        X1 is X - 1,
        factorial(X1,Z),
        Y is Z*X,!.
```

  - Now if you enter :

```
?- factorial(5,X).
X = 120
```

Clauses

- **Queries**
  - Queries are questions
  - The engine tries to entail the query (goal) using the Facts and Rules in KB
  - There are two kinds of answer
    - Yes/No: **parent(terach, abraham).**
    - Unified Answer/No: **parent(terach, Y).** →
      - Other possible answer(s) can be found using semicolon (return for stopping)

X=terach
Y=abraham;

X=terach
Y=nachor;

X=terach
Y=haran;

no

23

Clauses

- **Queries**
  - Q: Who is a grandparent of **issac**? (using parent relationship)
    - Who is a parent of issac? Assuming "Y"
    - Who is a parent of "Y"? Assuming "X"
    - ?- parent(Y, issac) , parent (X, Y).
    - If we change the order of them the logical meaning remains the same
  - Q: Who are **nachor**'s grandchildren?

24

How Prolog Answers Queries

- Instead of starting with simple facts given in the program, prolog starts with the goals. In fact, Prolog does goal driven search.
- Using rules, Prolog substitutes the current goals (which matches a rule head) with new sub-goals (the rule body), until the new sub-goals happen to be simple facts.
- Prolog returns the first answer matching the query. When prolog discovers that a branch fails or if you type ';' to get other answers, it backtracks to the previous node and tries to apply an alternative rule at that node.

25

How Prolog Answers Queries

- **Facts**
  - **parent (pam, bob).**
  - **parent (tom, bob).**
  - **parent (tom, liz).**
  - **parent (bob, ann).**
  - **parent (bob, pat).**
  - **parent (pat, jim).**
- **Rules**
  - **ancestor (X, Z) :- parent (X, Z).**
  - **ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z).**
- **Query**
  - ?- **ancestor (tom, pat).**
- The rule that appears first, is applied first
- Unifying: {tom/X} , {pat/Z}
  - The goal is replaced by : **parent (tom, pat).**

26

How Prolog Answers Queries

- Applying the next rule
  - **ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z)**
- Unifying: {tom/X} , {pat/Z}
  - New Goal: parent (tom, Y) , ancestor (Y, pat)
  - Prolog tries to satisfy them in order in which they are written
  - The first one matches one of the facts {bob/Y}
  - Second sub-goal: ancestor (bob, pat)
  - The same steps should be done for this sub-goal

27

How Prolog Answers Queries

- **Orders of Clauses and Goals**

1. ancestor (X, Z) :- parent (X, Z).
   ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z).
2. ancestor (X, Z) :- parent (X, Y) , ancestor (Y, Z).
   ancestor (X, Z) :- parent (X, Z).
3. ancestor (X, Z) :- parent (X, Z).
   ancestor (X, Z) :- ancestor (Y, Z) , parent (X, Y).
4. ancestor (X, Z) :- ancestor (Y, Z) , parent (X, Y).
   ancestor (X, Z) :- parent (X, Z).

28

How Prolog Answers Queries

- **Orders of Clauses and Goals**
  - It turns out that :
    - The first and second variations are able to reach and answer for ancestor.
    - The third sometimes can and sometimes can't
    - And the forth can never reach and answer (infinite recursion)
  - "Try simple things first".

29

Backtracking

- **Cut**
  - Up to this point, we have worked with Prolog's backtracking.
  - Sometimes it is desirable to selectively turn off backtracking.
  - Prolog provides the predicate called **cut** for this, represented by an exclamation point (!).
  - The cut effectively tells Prolog to freeze all the decisions made so far in this predicate. That is, if required to backtrack, it will automatically fail without trying other alternatives.

Backtracking

- **Cut**
  - **Example**

```
beautiful(claudia).
beautiful(sharon).
beautiful(denise).

intelligent(margaret).
intelligent(sharon).
intelligent(denise).

smart(claudia).
smart(sharon).
```

```
bride1(A) :-
    beautiful(A),
    intelligent(A).

bride2(A) :-
    beautiful(A), !,
    intelligent(A).

bride3(A) :-
    beautiful(A),
    smart(A).

bride4(A) :-
    beautiful(A), !,
    smart(A).
```

List

- List is a sequence of any number of items
- **Example:**
  - [mech, elec, civil, aero, cse, john, [1,2], [], [X]]
- List consists of two parts i.e L = [Head|Tail] :
  - Head – First item of the list
  - Tail – Remaining part of the list (tail itself is a list)
- Empty list is also a list ([ ])
- Lists are handled in Prolog as trees
- List1=[a, b, c]

  [a, b, c] = [a | [b,c]] = [a, b | [c]] = [a, b, c| [ ]]

List

- **Membership**
  - member(X,L) : X is an object that occurs in L
  - X is a member of list L if either:
    - X is the head of L. or
    - X is a member of the tail of L.

```
member( X, [X| Tail] ).
member( X, [Head| Tail] ) :-
                member( X, Tail ).
```

Example:

```
?-  member( a, [a, b, c]).
Yes.
?- member( [b, c], [a, [b, c]]).
Yes.
?- member(b, [a, [b, c]]).
No.
```

List

- **Concatenation**
  - conc(L1,L2,L3) : L3 is the concatenation of L1 & L2
  - If the first argument is empty list then second and third arguments must be the same list.

    conc([ ], L, L).
  - If the first argument is non-empty list then it can be represented as [X|L1]. Then the result of concatenation will be [X|L3] where L3 is the concatenation of L1 and L2.

    conc([X|L1], L2, [X|L3]) :- conc(L1, L2, L3).

List

- **Concatenation**

  Examples:
  - ?- conc([a,[b,c],d], [a, [ ],b], L).

    L = [a, [b,c], d, a, [ ], b]
  - ?- conc(Before, [feb| After], [jan, feb, mar]).

    Before = [jan]

    After = [mar]

  Note: Refer to rule -

  conc([X|L1], L2, [X|L3]) :- conc(L1, L2, L3).

List

- **Deleting Item**
  - del(X,L,L1) : L1 is equal to L with the first occurrence of item X removed.
  - If X is the head of the list then the result after deletion is the tail of the list.

    del(X, [X| Tail], Tail).
  - If X is in the tail then it is deleted from there.

    del(X, [Y| Tail], [Y| Tail1]) :- del(X, Tail, Tail1)

  Example

```
?- del(a, [a, b, a, c], L).
L = [b, a, c];
L = [a, b, c];
no
```

List

- **Inserting Item**
  - insert(X,L,L1) : L1 is any list such that deleting X from L1 gives L.
  - insert(X, List, BiggerList) :-

        del(X, BiggerList, List).

  Example

```
?- insert( a, [ b, c, d], L).
L = [a, b, c, d];
L = [b, a, c, d];
L = [b, c, a, d];
L = [b, c, d, a];
no
```

List

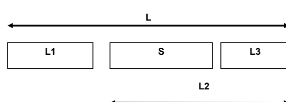
- **Adding Item**
  - add(X,L,L1) : L1 is the list obtained by adding X to L. X is added in front of the list L.
  - The resulting list L1 is simply [X|L].
  - add(X,L,[X|L]).

  Example:

```
? add(4.2, [1, a, f, [2] ], L1).
L1 = [4.2, 1, a, f, [2] ]
```

List

- **Sublist**
  - S is a sublist of L if:
    - L can be decomposed into two lists, L1 and L2, &
    - L2 can be decomposed into two lists, S and some L3



List

- **Sublist**
  - sublist(S,L):-

        conc(L1,L2,L), conc(S,L3,L2)

  Example

```
? sublist([b,c],[a,b,c,d,e]).
Yes
? sublist (S, [a,b,c]).
S = [ ];
S = [a];
S = [a,b];
S = [a,b,c];
S = [ ];
S= [b];
...
```

Operator

- Some of the operators are in-built in Prolog.
- Some predefined arithmetic operators:
  - \+ addition
  - \- subtraction
  - * multiplication
  - / division
  - mod modulo, the remainder of integer division
- Precedence of operator decides the correct interpretation of expressions.
- Operator with highest precedence is the principle functor of the term.
- Operators are functors. 3+4 = 3+4 (3+4 ≠ 7)

Operator

- New operators can be defined by inserting into special kind of clauses called directives. Example:

  :-op(600,xfx,<==>)
- Three group of operators:
  - infix: xfx, xfy, yfx
  - prefix: fx, fy
  - postfix: xf, yf
- These definition helps in unambiguous interpretation of expressions which have sequence of binary operators.

Operator

- x represents an argument whose precedence must be strictly lower than that of operator
- y represents an argument whose precedence is lower or equal to that of operator
- Precedence of an argument in parentheses or unstructured object is zero
- Precedence of a structured argument is equal to the precedence of its principal functor
- Example:

  :-op(500,yfx,-)

  a – b – c

  Interpreted as (a-b)-c and not as a-(b-c)

Arithmetic

- '=' is matching operator (it does not evaluate on its own)
- Matches right hand side with left hand side
- **Examples**

1. ? X=1+2

   X=1+2
2. ? 1+3=2+2

   No
3. ?X+3=4+Y

   X=4

   Y=3

Arithmetic

- '**is**' operator forces evaluation of expression on RHS forcing instantiation of values on LHS to the evaluated value
- **Examples**

1. X is 1+2

   X=3
2. Y is 5 mod 2

   Y=1

- At the time of evaluation all arguments must be instantiated to numbers

Arithmetic

- Values of arithmetic expressions can be compared by arithmetic operators:
  - X<Y :- X is less than Y
  - X>Y :- X is greater than Y
  - X >=Y :- X is greater than or equal to Y
  - X =<Y :- X is less than or equal to Y
  - X =:= Y :- the values of X and Y are equal
  - X =\= Y :- the values of X and Y are not equal

Example

- Calculating number of elements in a list
- length(List,N) % N is the number of elements in List

```
length([ ], 0).                          %R1
length( [Head|Tail], N):-
     length(Tail, N1),
     N is 1+N1.                          %R2
```

- Query:

  ? length ([a,b,c,d],N).

  N = 4

How to Run

- If you want to use Prolog you need a compiler.
- There are many compilers downloadable on internet.
- You can use SWI-Prolog. SWI-Prolog is free Prolog compiler licensed under the GPL.

  http://www.swi-prolog.org/

  Available for Windows, Linux, Unix and MacOS

How to Run

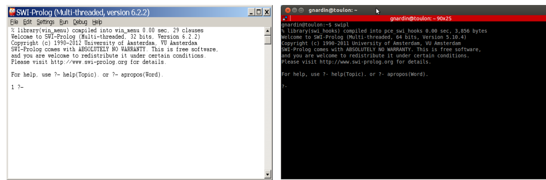
- Prolog has an interactive interpreter
- After starting SWI-Prolog, the interpreter can start reading your Prolog files and accept your queries.
  – Linux and MacOS, type the command on terminal: **swipl**
  – Windows: **Start** -> **Programs** -> **SWI-Prolog** -> **Prolog**
- To exit Prolog simply type the command '**halt.**'
- Prolog program files usually have the extension .pl

How to Run

- Prolog has an interactive interpreter
- After starting SWI-Prolog, the interpreter can start reading your Prolog files and accept your queries.

How to Run

- To start Prolog
  – Linux and MacOS
    • Type the command on terminal: **swipl**
  – Windows
    • **Start** -> **Programs** -> **SWI-Prolog** -> **Prolog**



Windows **Linux**

How to Run

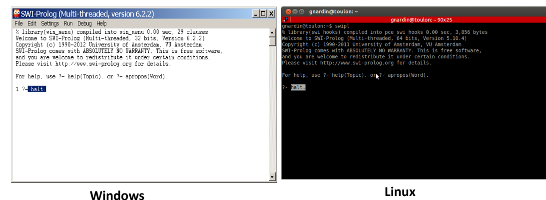
- **Useful commands**
  – **consult('<*filename*>').** *OR* **['<*filename*>']**
    • Load a source file, where **<*filename*>** is the full path to the prolog file name. E.g. **consult(likes).** OR **['/home/user/prolog/likes'].**
  – **reconsult ('<*filename*>').**
    • Reconsult a changed source files. E.g. **reconsult('/home/user/prolog/sample').**
  – **listing.**
    • List all the predicates
  – **make.**
    • Reload all files that have been changed since they were last loaded. Normally used after editing one or more files.
  – **help(<*Spec*>).**
    • Give help on **<*Spec*>**, which is normally the name of a predicate or C interface function.

How to Run

- **Useful commands**
  – **trace.**
    • Switch on trace
  – **notrace.**
    • Switch off trace

How to Run

- To exit Prolog simply type the command '**halt.**'



Windows **Linux**

References

- Slides based from
  - www.me.iitb.ac.in/~aks/Introduction%20to%20PROLOG.ppt
  - http://users.encs.concordia.ca/~m_razma/Comp472-W08/Slides/Tut1.ppt

Bibliography

- Bratko, I. (2001) *Prolog Programming for Artificial Intelligence.* International Computer Science Series, Addison Wesley.
- Clocksin, W. F. and C. S. Mellish (1981) *Programming in Prolog*. Springer-Verlag, Berlin.
- O'Keefe, R. (1990) *The Craft of Prolog*. MIT Press, Cambridge, Massachusetts.
- Ross, P. (1989) *Advanced Prolog*. Addison Weseley.
- Sterling, L. and E. Shapiro (1986) *The Art of Prolog*. MIT Press, Cambridge, Massachusetts.