INTRODUCTION TO PROLOG

Ho Chi Minh City University of Technology
Faculty of Computer Science and Engineering

Huỳnh Tấn Đạt
Email: htdat@cse.hcmut.edu.vn
Home Page: http://www.cse.hcmut.edu.vn/~htdat/
References

[1]. Ivan Bratko (1990), Prolog Programming For Artificial Intelligence, Addition-Westley.


Prolog

- PROgramming in LOGic
- Alain Colmerauer & Philippe Roussel, 1972
- A declarative programming language to describe problems

https://fb.com/tailieudientucntt
Prolog

Irrefutable reasoning processes (Aristotle):

- Socrates is a man
- All men are mortal

\[ \Rightarrow \text{Socrates is mortal} \]

- Jerry is a robber
- Tom is a child of John
- John is rich
- X is rich if X’s father is rich or X is a robber

\[
\begin{align*}
\text{man(socrates).} \\
\text{mortal(X) :} & \text{ man(X).} \\
\text{robber(jerry).} \\
\text{childof(tom, john).} \\
\text{rich(john).} \\
\text{rich(X) :} & \text{ childof(X, Y), rich(Y).} \\
\text{rich(X) :} & \text{ robber(X).}
\end{align*}
\]
A predicate always returns true or false
Clause

- Two types of clause:
  - Fact
  - Rule

- Clauses are separated by ‘.’ character.

robber(jerry).
childof(tom, john).
rich(john).
rich(X) :- childof(X, Y), rich(Y).
rich(X) :- rober(X).
Rule

- A rule returns true if all sub-clauses return true.
- The clauses of a predicate should be consecutive.

rich(X) :- childof(X, Y), rich(Y).
rich(X) :- rober(X).
Clause

- Literals, or atoms, start with lower-case letters, or enclose in quotes ‘ ‘

- Whereas variables start with Upper-case letters.

  man(socrates).
  mortal(X) :- man(X).

- The arguments in sub-clauses must be variables, not expressions.

- Example: factorial(N – 1, R1). % Wrong!!!

  factorial(0, 1).
  factorial(N, R1) :- N1 is N – 1, factorial(N1, R1), R is R1 * N.
How Prolog answers questions?

- A question to Prolog is always a sequence of one or more goals.
- Prolog tries to satisfy all the goals by matching clauses.
- Create the associations (unifications) between arguments in the question and parameters declared in the head of clauses.
- Execute the sub-clauses of the current clauses.
- If all sub-clauses are satisfied, the variables in question will be bound to values => We have got a solution.
How Prolog answers questions?

Example:

man(socrates).
man(xeda).
king(xeda).
happy(X) :- man(X), king(X).

?- happy(xeda)
?- happy(Y)
How Prolog answers questions?

Example:

\[
\text{man(socrates).} \\
\text{man(xeda).} \\
\text{king(xeda).} \\
\text{happy}(X) :- \text{man}(X), \text{king}(X). \\
\]

Unify( \text{happy(xeda)}, \text{happy}(X) ) \Rightarrow X = \text{xeda}

?- \text{happy(xeda)}
How Prolog answers questions?

- Example:

\[
\text{man(socrates).} \\
\text{man(xeda).} \\
\text{king(xeda).} \\
\text{happy(X) :- man(X), king(X).}
\]

Unify( \text{man(scocrates), man(X) } ) \quad X = \text{xeda}

\Rightarrow \text{fail!}

?- happy(xeda)
How Prolog answers questions?

Example:

\[
\begin{align*}
\text{man(socrates).} \\
\text{man(xeda).} \\
\text{king(xeda).} \\
\text{happy}(X) &: \text{man}(X), \text{king}(X).
\end{align*}
\]

Unify( \text{man(xeda)}, \text{man}(X) ) \quad X = \text{xeda}

\Rightarrow \text{successful}

?- \text{happy}(\text{xeda})
How Prolog answers questions?

- Example:

  man(socrates).
  man(xeda).
  king(xeda).
  happy(X) :- man(X), king(X).

  Unify( king(xeda), king(X) )
  => successful

  X = xeda

  ?- happy(xeda)
How Prolog answers questions?

Example:

man(socrates).
man(xeda).
king(xeda).
happy(X) :- man(X), king(X).

?- happy(xeda)
yes
How Prolog answers questions?

- Example:

```prolog
man(socrates).
man(xeda).
king(xeda).
happy(X) :- man(X), king(X).
```

?- happy(Y)
How Prolog answers questions?

Example:

\[
\begin{align*}
\text{man(socrates).} \\
\text{man(xeda).} \\
\text{king(xeda).} \\
\text{happy(X) :- man(X), king(X).} \\
\text{?- happy(Y)}
\end{align*}
\]

Unify( happy(Y), happy(X) ) => X = Y
How Prolog answers questions?

Example:

\[\text{man(socrates)}.\]
\[\text{man(xeda)}.\]
\[\text{king(xeda)}.\]
\[\text{happy(X) :- man(X), king(X)}.\]

Unify( \text{man(scocrates)}, \text{man(X)} ) \quad X = Y = \text{socrates}

\[\text{?- happy(Y)}\]
How Prolog answers questions?

Example:

\[
\begin{align*}
\text{man(socrates).} \\
\text{man(xeda).} \\
\text{king(xeda).} \\
\text{happy}(X) \leftarrow \text{man}(X), \text{king}(X).
\end{align*}
\]

Unify( \text{king(xeda)}, \text{man}(X) \) \quad X = Y = \text{socrates}

=> Fail

=> Backtrack!

?- happy(Y)
How Prolog answers questions?

Example:

\[
\text{man(socrates).}
\]

\[
\text{man(xeda).}
\]

\[
\text{king(xeda).}
\]

\[
\text{happy(X) :- man(X), king(X).}
\]

Backtrack!  \[ X = Y = \text{socrates} \]

Release the bound value of \( X \)  \[ => X = Y \]

?- happy(Y)
How Prolog answers questions?

- Example:

\[
\text{man(socrates).}
\]

\[
\text{man(xeda).}
\]

\[
\text{king(xeda).}
\]

\[
\text{happy(X) :- man(X), king(X).}
\]

Unify( man(xeda), man(X) )  \[ X = Y = xeda \]

?- happy(Y)
How Prolog answers questions?

Example:

\[
\begin{align*}
\text{man}(\text{socrates}). \\
\text{man}(\text{xeda}). \\
\text{king}(\text{xeda}). \\
\text{happy}(X) & : \text{- man}(X), \text{king}(X).
\end{align*}
\]

Unify( king(xeda), king(X) ) \quad X = Y = xeda

=> successful

?- happy(Y)
How Prolog answers questions?

Example:

man(socrates).
man(xeda).
king(xeda).

happy(X) :- man(X), king(X).

?- happy(Y)
Y = xeda

X = Y = xeda
Controlling backtracking

- Prolog automatically backtrack if this is necessary for satisfying a goal.
- The results may be different if we change the order of clauses of a predicate.
- Example:

```prolog
p(X) :- q(X).
p(3).
q(1).
q(2).
?- p(X)
```

Cuu Duong Than Cong. com

https://fb.com/tailieudientucntt
Controlling backtracking

- Uncontrolled backtracking may cause inefficiency in a program.
- Cut mechanism is used to prevent the backtracking of prolog at the point indicated by ‘CUT’

\[ \text{p(X) :- q(X), !.} \]
\[ \text{p(3).} \]
\[ \text{q(1).} \]
\[ \text{q(2).} \]

?- p(X)
Controlling backtracking

- If $X < 3$ then $Y = 0$
- If $3 \leq X < 6$ then $Y = 2$
- If $6 \leq X$ then $Y = 4$

?- $f(1, Y), 2 < Y$

$=>$ $Y = 0$ and the sub-goal $2 < 0$ fails

$=>$ Prolog uses backtracking mechanism to try two useless alternatives (rule no. 2 and 3)
Controlling backtracking

- The 3 rules about f relation are mutually exclusive, as soon as one rule succeeds, we do not need to try the other rules.

f(X, 0) :- X < 3, !.
f(X, 2) :- 3 =< X, X < 6, !.
f(X, 4) :- 6 =< X.

- The third version of the program:

f(X, 0) :- X < 3, !.
f(X, 2) :- X < 6, !.
f(X, 4).

?- f(1, Y), 2 < Y.

Rules no. 2 and 3 are not matched.
Problems with CUT

Advantages
- We can improve the efficiency of the program
- We can specify ‘mutually exclusive rules
  
  p:- c, !, s1.
  p:- s2.

Disadvantages
- We can lose the valuable correspondence between the declarative and procedural meaning of programs
- A change in the order of clauses may affect the declarative meaning. => We can get different results.
Problems with CUT

- Example

\[
\begin{align*}
p & : - a, b. \ % p & : - a, b ; c \\
p & : - c. & \quad p \leftarrow (a \text{ and } b) \text{ or } c \\
\end{align*}
\]

Insert a CUT:

\[
\begin{align*}
p & : - a, !, b. & \quad p \leftarrow (a \text{ and } b) \text{ or } c \\
p & : - c. & \quad p \leftarrow (a \text{ and } b) \text{ or } (\text{not } a \text{ and } c) \\
\end{align*}
\]

If we swap the clauses:

\[
\begin{align*}
p & : - c. & \quad p \leftarrow c \text{ or } (a \text{ and } b) \\
p & : - a, !, b. & \quad p \leftarrow (a \text{ and } b) \text{ or } (\text{not } a \text{ and } c) \\
\end{align*}
\]

- When we use the CUT facility we have to pay more attention to the procedural aspects.
fail is a special sug-goal, which always fails.

“Mary likes all animals but snakes”:
like(mary, X) :- snake(X), !, fail.
like(mary, X) :- animal(X).

different(X, Y):
different(X, X) :- !, fail.
different(X, Y).

CuuDuongThanCong.com
https://fb.com/tailieudientucntt
NOT Predicate

- Many Prolog implementations support this predicate. If not, we can define NOT ourselves.
  
not(P) :- P, !, fail.
  not(_).

- The two examples can be rewritten with NOT as:
  likes(mary, X) :- animal(X), not(snake(X)).
  different(X, Y) :- not(X == Y).  (%(X =\!\!= Y)
Closed World Assumption

- When processing a not(P) goal, Prolog does not try to prove this goal directly. Instead, it tries to prove the opposite. And if the opposite cannot be proved then Prolog assumes that the not(P) goal succeeds.

- Such reasoning is based on the so-called closed world assumption.

- The world is closed in the sense that everything that exists is in the program or can be derived from a program.

- If something is not in the program (or cannot be derived from it) then it is not true and consequently its negation is true.
Closed World Assumption

Example

- If we not explicitly enter the following clause into our program

  \texttt{human(mary)}.

then Prolog will answer ‘no’ for this question

\begin{verbatim}
?- human(mary)
no
\end{verbatim}

There is not enough information in the program to prove that Mary is human.
A list is sequence of any number of items.

- Example: [ann, tennis, tom, skiing]

An empty list: [ ]

An non-empty list:

[ann, tennis, tom, skiing]

[The first item (head) | The remaining part of the list (tail)]

Prolog uses ‘|’ to separate the head and tail of a list.

Tail itself is a list.

The elements of a list can be objects of any kind. They can be lists.
List

- Prolog provides 3 notations for lists:
  - [Item1, Item2, ...]
  - [Head | Tail]
  - [Item1, Item2, ... | Others]
Example of Operations on List

- **Length of a list:**
  - \text{length}(L, R): R is the length of the list L.
  - \text{length}([], 0).
  - \text{length}([_ | T], R) \leftarrow \text{length}(T, R1), R is R1 + 1.
  - **Anonymous variable:** `_`.

- **Membership:**
  - \text{member}(X, L): return true if X occurs in L
  - **Example:** \text{member}(b, [a, b, c]) \Rightarrow true
    - \text{member}(d, [a, b, c]) \Rightarrow false
    - \text{member}(H, [H | _]) \Leftarrow !.
    - \text{member}(H, [_ | Tail]) \Leftarrow \text{member}(H, Tail).

Faculty of Computer Science and Engineering – HCMUT
Exercises

- Get the first element of a list L.
- Insert an element into the first position of a list L.
- Get the $n^{th}$ element of a list L.
- Append two lists L1 and L2 into L3.
- Write count1(L, R) predicate where L is a list of integers and the output R is the number of positive numbers in L.
  - Example: count1([1, 2, -1, 3, 2], R) => R = 4
- Write count2(LL, R) predicate where LL is 2-dimentional list of integers and the output R is the number of positive numbers in LL.
  - Example: count2([1, 2, -1, 3, 2], [ ], [2], R) => R = 5