KRR5: Logic programming: PROLOG

Yannick Pencolé
Yannick.Pencole@anu.edu.au

17 Mar 2005
What is Prolog?

Prolog is a programming language

Declarative programming
The programmer declares a knowledge base (KB) and asks a question. Prolog does the rest.

How does it work
The KB declared in Prolog is based on Horn’s Clauses. To answer the question, Prolog uses Backward Chaining.
Constants and Variables

Definition

A **Constant** is

1. **Number:** 12, 3.5
2. **Atoms:**
   - any string that begins with a small letter
   - any string between " "
   - empty lists symbol []
3. **Variables:**
   - any string that begins with a capital letter
   - any string that begins with _
   - wildcard pattern _
### Three kinds of knowledge

| Definition | A **Fact** is a predicate.  
|            | \( p(\ldots) \) (i.e. \( p(\ldots) \)).  
|            | A fact can be seen as the **Head** of a Horn’s clause. |

| Definition | A **Rule** is a complete Horn clause:  
|            | \( p(\ldots) :- q(\ldots), \ldots, r(\ldots) \).  
|            | (i.e. \( q(\ldots) \land \ldots \land r(\ldots) \Rightarrow p(\ldots) \)) |

| Definition | A **Query** is a set of predicates:  
|            | \( s(\ldots), \ldots, t(\ldots) \).  
|            | A query can be seen as the **Body** of a Horn’s clause. |
My first program

KB

Here is the KB to program:

\( \text{father(charlie, david)} \)
\( \text{father(henri, charlie)} \)
\( \text{father}(X, Z) \land \text{father}(Z, Y) \Rightarrow \text{grandfather}(X, Y) \)

In Prolog

\text{father(charlie,david).}
\text{father(henri,charlie).}
\text{grandfather}(X,Y) :- \text{father}(X,Z) , \text{father}(Z,Y).
pencole@chef$ swiprolog
The binary name `swiprolog' is deprecated in favour of `swipl'.
Please use the new name instead.

Welcome to SWI-Prolog (Multi-threaded, Version 5.2.13)
Copyright (c) 1990-2003 University of Amsterdam.
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software,
and you are welcome to redistribute it under certain conditions.
Please visit http://www.swi-prolog.org for details.

For help, use ?- help(Topic). or ?- apropos(Word).

?- [father].
% father compiled 0.00 sec, 1,148 bytes

Yes
My first program

```
Yes
?- father(charlie,david).

Yes
?- father(charlie,henri).

No
?- father(X,Y).

X = charlie
Y = david ;

X = henri
Y = charlie ;

No
?- 
```
My first program

?- grandfather(X, Y).
No
?- grandfather(X, Y).
X = henri
Y = david ;
No
?- grandfather(henri, X).
X = david ;
No
Order of the answers

?- listing.

mother(sophie, charlie).
mother(anne, david).

parents(A, B, C) :-
    father(B, A),
mother(C, A).

% Foreign: rl_read_init_file/1

father(charlie, david).
father(henri, charlie).
father(david, luc).

?- parents(X, Y, Z).
X = david
Y = charlie
Z = anne ;

X = charlie
Y = henri
Z = sophie ;

No

Prolog “reads” clauses from the top to the bottom and “explores” from the left to the right.
In Prolog, we can also declare a function of FOL. A function has no result, it is just a functional relation.

**Example**

John’s wife: \( \text{wife}(\text{john}) \)

Such a term is always included in a predicate in Prolog:
\( \text{name}(\text{wife}(\text{john}), \text{marie}). \)

Be careful about the confusion between the function \( \text{wife}(\text{john}) \) which represents the wife of John and the predicate \( \text{wife}(\text{john}) \) which says that John is a wife!
Arithmetic

How to play with arithmetic

- Comparisons: >, <, >=, <=, =:=, =\==
- Assignation: is
  - ?- X is 3+2.
  - X=5

- Predefined functions: −, +, *, /, ^, mod, abs, min, max, sign, random, sqrt, sin, cos, tan, log, exp...
Recursive programming

Depth-first search

Depth-first search from a start state X:
\[
dfs(X) :- \text{goal}(X).
dfs(X) :- \text{successor}(X, S) \ dfs(S).
\]

Factorial

Factorial:
\[
fact(A,B) :- \text{fact}(A,1,B).
fact(A,B,C) :- A > 1, D \ is \ B \times A, \ E \ is \ A-1, \ fact(E,D,C).
fact(1,A,A).
\]
Redundant inference and infinite loops

Find a path: version 1

\[
\begin{align*}
\text{link}(a, b). \\
\text{link}(b, c).
\end{align*}
\]

\[
\begin{align*}
\text{path}(X, Z) & :\text{= } \text{link}(X, Z). \\
\text{path}(X, Z) & :\text{= } \text{path}(X, Y), \text{link}(Y, Z).
\end{align*}
\]

Find a path: version 2

\[
\begin{align*}
\text{link}(a, b). \\
\text{link}(b, c).
\end{align*}
\]

\[
\begin{align*}
\text{path}(X, Z) & :\text{= } \text{path}(X, Y), \text{link}(Y, Z). \\
\text{path}(X, Z) & :\text{= } \text{link}(X, Z).
\end{align*}
\]

What is the difference between version 1 and version 2?
Example

path(a,c)
Example

- path(a,c)
- link(a,c)
- fail
Example

Proof tree: version 1

path(a,c)

link(a,c)

fail

path(a,Y)
Example

Proof tree: version 1

path(a,c)

link(a,c)
fail

path(a,Y)

link(a,Y)

\{Y/b\}
Proof tree: version 1

Example

- **path(a,c)**
  - **link(a,c)**
    - fail
  - **path(a,Y)**
    - **link(a,Y)**
      - {Y/b}
  - **link(b,c)**
    - {}


Example

path(a,c)
Example

path(a,Y)
path(a,c)
link(Y,c)
Example

- path(a, Y)
- link(Y', Y)
- path(a, c)
- link(Y', c)
Example

path(a,Y)

path(a,Y') link(Y',Y)

path(a,Y'') link(Y'',Y')
## Term comparison and unification

### Comparison
- $T_1 == T_2$ succeeds if $T_1$ and $T_2$ are identical (equality of FOL)
- $T_1 \not== T_2$ succeeds if $T_1$ and $T_2$ are not identical

### Unification
- $T_1 = T_2$ is the Unification of $T_1$ and $T_2$ (i.e. $\text{UNIFY}(T_1,T_2)$ is called)
- $T_1 \not= T_2$ succeeds if (i.e. $\text{UNIFY}(T_1,T_2)$ has no solution)
Lists

Empty list

The empty list is represented by: [ ]

General case

A list has a Head and a Tail: [ Head | Tail ]

Example

The list $a, b, c$ is denoted in Prolog: [ a | [ b | [ c | [ ] ] ] ]
Lists: examples

Example

1 \[ X \mid L \] = [ a, b, c ] \rightarrow
Lists: examples

Example

1. \([ X | L ] = [ a, b, c ] \rightarrow X = a, L = [b,c] \)
2. \([ X | L ] = [a] \rightarrow \)
Lists: examples

Example

1. \([ X \mid L ] = [ a, b, c ] \rightarrow X = a, L = [b,c]\\
2. \([ X \mid L ] = [a] \rightarrow X = a, L = [ ]\\
3. \([ X \mid L ] = [ ] \rightarrow [ ]\)
Lists: examples

Example

1. \[ [ \text{X} \mid \text{L} ] = [ \text{a, b, c} ] \rightarrow \text{X = a, L = [b,c]} \]
2. \[ [ \text{X} \mid \text{L} ] = [\text{a}] \rightarrow \text{X = a, L = [ ]} \]
3. \[ [ \text{X} \mid \text{L} ] = [\ ] \rightarrow \text{fail} \]
4. \[ [ \text{X, Y} ] = [ \text{a, b, c} ] \rightarrow \]
Lists: examples

Example

1. $[X | L] = [a, b, c] \rightarrow X = a, L = [b, c]$
2. $[X | L] = [a] \rightarrow X = a, L = []$
3. $[X | L] = [] \rightarrow \text{fail}$
4. $[X, Y] = [a, b, c] \rightarrow \text{fail}$
5. $[X, Y | L] = [a, b, c] \rightarrow$
### Lists: examples

**Example**

1. \[ X \mid L \] = \[ a, b, c \] → \( X = a, L = \{ b, c \} \)
2. \[ X \mid L \] = \[ a \] → \( X = a, L = \emptyset \)
3. \[ X \mid L \] = \[ \] → fail
4. \[ X, Y \] = \[ a, b, c \] → fail
5. \[ X, Y \mid L \] = \[ a, b, c \] → \( X = a, Y = b, L = \{ c \} \)
6. \[ X \mid L \] = \[ X, Y \mid L2 \] →
Lists: examples

Example

1. \[ [ X \mid L ] = [ a, b, c ] \rightarrow X = a, L = [b,c] \]
2. \[ [ X \mid L ] = [a] \rightarrow X = a, L = [ ] \]
3. \[ [ X \mid L ] = [ ] \rightarrow \text{fail} \]
4. \[ [ X, Y ] = [ a, b, c ] \rightarrow \text{fail} \]
5. \[ [ X, Y \mid L ] = [ a, b, c ] \rightarrow X = a, Y = b, L = [c] \]
6. \[ [ X \mid L ] = [ X, Y \mid L_2 ] \rightarrow L = [Y|L_2] \]
Sum of elements

Example

```prolog
sumElements([], 0).
sumElements([ A | B ], C) :-
    sumElements(B, D),
    C is D + A.
```

Query:

?- sumElements([1, 2, 3, 5], N).
N = 11 ;
No
Example

ith([ X | _],1,X).
ith([ _ | L ], R, Y) :-
    Rm1 is R-1, ith(L,Rm1,Y).

Query:
?- ith([a,b,c,d],2,N).
N = b ;
No
Append

append is a predefined predicate to append lists

Example

?- append([a,b,c],[d,e],L)

L = [ a,b,c,d ]

How to find the last element of a list?
?- append(_, [X],[a,b,c,d])

X= d

How to create sub-lists from lists?
?- append(L2,L3,[ b,c,a,d,e ]), append(L1, [ a ], L2).
L2 = [b,c,a]
L3 = [d,e]
L1 = [ b , c ]
Example

Given two sorted lists L1, L2 the predicate `merge` merges the lists to build a new sorted list:

```
merge([], L, L).
merge( L, [], L).
merge( [X|L1], [Y | L2 ], [ X | L ] ) :- X=<Y, merge(L1, [ Y | L2 ], L).
merge( [X|L1], [Y | L2 ], [ Y | L ] ) :- X>Y, merge([X | L1],L2, L).
```
Prolog allows a “kind of” negation called negation as failure. If Prolog is not able to prove \( P \) then \( \text{not}P \) is proved!

Example

\[
\text{alive}(X) :- \text{not} \text{ dead}(X).
\]

means: “Everyone is alive if not provably dead”.

Be careful the \text{not} is NOT the \( \neg \) of FOL. If we are not able to prove \( \text{dead}(X) \), we cannot say anything about \( \neg \text{dead}(X) \).
Imagine the following rules:
R1: \( \text{belong}(X, [X \mid \_\_]) \).
R2: \( \text{belong}(X, [\_ \mid L \_]) \) :- \( \text{belong}(X,L) \).
and the query
\( \text{belong}(X, [a, b, c]) \).
Solution: \( X = a, X = b, X = c \)

**Proof tree**: at each node of the tree, we choose R1 and THEN R2.

\[ R1: \text{belong}(X, [X \mid \_\_]) :- ! \]
\[ R2: \text{belong}(X, [\_ \mid L \_]) :- \text{belong}(X,L) \]
and the query
\( \text{belong}(X, [a, b, c]) \).
Solution: \( X = a \)

**Proof tree**: We cut the complete proof tree. At each node of the tree, we choose only the rule that are before “!” (i.e. R1)
Teaching at the ANU

person(yannick).
study(people,anu).
have(people,m1).
goodlectureslogic(m1).
students(X) :- study(X,anu).
gives(yannick,X,people) :- goodlectureslogic(X) , have(people,X).
goodteacher(X) :- person(X), gives(X,Y,Z),
goodlecturesfol(Y) , students(Z).
goodlecturesfol(X) :- goodlectureslogic(X).

Query:
?- goodteacher(Yannick).
Yes
?- goodteacher(Z).
Z = Yannick