Part I

Logic programming: PROLOG

1 Introduction

What is Prolog?

Prolog is a programming language

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

The KB declared in Prolog is based on Horn’s Clauses. To answer the question, Prolog uses Backward Chaining.

2 Syntax and Examples

Constants and Variables

Definition 1. 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 2. A Fact is a predicate. p(...). (i.e. p(...)). A fact can be seen as the Head of a Horn’s clause.

Definition 3. A Rule is a complete Horn clause: p(...) :- q(...), ..., r(...). (i.e. q(...) ∧ ... ∧ r(...) ⇒ p(...))

Definition 4. A Query is a set of predicates: s(...), ..., t(...). A query can be seen as the Body of a Horn’s clause.
My first program

Here is the KB to program:

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

father(charlie, david).
father(henri, charlie).
grandfather(X, Y) :- father(X, Z), father(Z, Y).

My first program

```
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
```

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
Order of the answers

In Prolog, the order of answers is from the top to the bottom and from the left to the right.

Example:

```
?- grandfather(X,Y).
No
?- grandfather(X,Y).
X = henri
Y = david ;
No
?- grandfather(henri,X).
X = david ;
No
```

Functions

In Prolog, we can also declare a function of FOL. A function has no result, it is just a functional relation.

**Example 5. John’s wife:** 

```
wife(john)
```

Such a term is always included in a predicate in Prolog: `name(wife(john), marie).`

Be careful about the confusion between the function `wife(john)` which represents the wife of John and the predicate `wife(john)` which says that John is a wife!

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 from a start state X: dfs(X) :- goal(X).
    dfs(X) :- successor(X,S) dfs(S).

Factorial:

Redundant inference and infinite loops

link(a,b). link(b,c). path(X,Z) :- link(X,Z).
    path(X,Z) :- path(X,Y), link(Y,Z).

link(a,b). link(b,c). path(X,Z) :- path(X,Y), link(Y,Z). path(X,Z) :- link(X,Z).

What is the difference between version 1 and version 2?

Proof tree: version 1

Example 6. path(a,c)

Proof tree: version 1

Example 7. fail

Proof tree: version 1

Example 8. fail

4
Example 9.

Proof tree: version 1

Example 10.

Proof tree: version 1

Example 11.

Proof tree: version 2

Example 12.

Proof tree: version 2
Example 13.

Proof tree: version 2

Example 14.

Term comparison and unification

- $T_1 \equiv T_2$ succeeds if $T_1$ and $T_2$ are identical (equality of FOL)
- $T_1 \neq T_2$ succeeds if $T_1$ and $T_2$ are not identical

- $T_1 = T_2$ is the Unification of $T_1$ and $T_2$ (i.e. UNIFY($T_1$, $T_2$) is called)
- $T_1 \nless T_2$ succeeds if (i.e. UNIFY($T_1$, $T_2$) has no solution)
Lists

The empty list is represented by: [ ]

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

*Example 15.* The list $a, b, c$ is denoted in Prolog: [ a || b || c || [ ] ]

Lists: examples

*Example 16.*
1. [ X | L ] = [ a, b, c ] →

Lists: examples

*Example 17.*
1. [ X | L ] = [ a, b, c ] → $X = a, L = [b,c]$
2. [ X | L ] = [a] →

Lists: examples

*Example 18.*
1. [ X | L ] = [ a, b, c ] → $X = a, L = [b,c]$
2. [ X | L ] = [a] → $X = a, L = [ ]$
3. [ X | L ] = [ ] →

Lists: examples

*Example 19.*
1. [ X | L ] = [ a, b, c ] → $X = a, L = [b,c]$
2. [ X | L ] = [a] → $X = a, L = [ ]$
3. [ X | L ] = [ ] → fail
4. [ X , Y ] = [ a, b, c ] →

Lists: examples

*Example 20.*
1. [ X | L ] = [ a, b, c ] → $X = a, L = [b,c]$
2. [ X | L ] = [a] → $X = a, L = [ ]$
3. [ X | L ] = [ ] → fail
4. [ X , Y ] = [ a, b, c ] → fail
5. [ X, Y | L ] = [ a, b, c ] →
Lists: examples

Example 21. 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 | L ] = [ a, b, c ] \rightarrow \text{fail}

5. \[ X, Y | L ] = [ a, b, c ] \rightarrow X = a, Y = b, L = [ c ]

6. \[ X | L ] = [ X, Y | L2 ] \rightarrow \text{fail}

Lists: examples

Example 22. 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 | L ] = [ a, b, c ] \rightarrow \text{fail}

5. \[ X, Y | L ] = [ a, b, c ] \rightarrow X = a, Y = b, L = [ c ]

6. \[ X | L ] = [ X, Y | L2 ] \rightarrow L = [ Y | L2 ]

Sum of elements

Example 23. sumElements([ ], 0). sumElements([ A | B ], C) :- \text{sumEl is D} + A. Query:

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

Wildcard pattern: ith

Example 24. 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

Predicate append

append is a predefined predicate to append lists

Example 25. ?- 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(L1, L2, [ b, c, a, d, e ]), append(L1, [ a ], L2). L2 = [ b, c, a] L3 = [ d, e] L1 = [ b, c ]

Sort

Example 26. 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).
Negation as failure

Prolog allows a “kind of” negation called *negation as failure*. If Prolog is not able to prove $P$ then $\neg P$ is proved!

*Example 27.* $\text{alive}(X) :- \text{not} \text{ dead}(X)$.

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 $\neg \text{dead}(X)$

The cut

Imagine the following rules:

$R1: \text{belong}(X, [X \mid .])$. $R2: \text{belong}(X, [, L ]) :- \text{belong}(X,L)$.

and the query $\text{belong}(X, [a,b,c])$. Solution: $X = \text{a}$, $X = \text{b}$, $X = \text{c}$  
*Proof tree:* at each node of the tree, we choose $R1$ and THEN $R2$.

$R1: \text{belong}(X, [X \mid .])$: !. $R2: \text{belong}(X, [, L ]) :- \text{belong}(X,L)$.

and the query $\text{belong}(X, [a,b,c])$. Solution: $X = \text{a}$  
*Proof tree:* We *cut* the complete proof tree. At each node of the tree, we choose only the rule that are before “!” (i.e. $R1$)

Last example :-)

$\text{person(\text{yannick}).}$

$\text{study(\text{people, anu}).}$

$\text{have(\text{people, m1}).}$

$\text{goodlectureslogic(m1).}$

$\text{students(X) :- study(X, anu)}.$

$\text{gives(\text{yannick, X, people}) :- goodlectureslogic(X) , have(people, X).}$

$\text{goodteacher(X) :- person(X), gives(X,Y,Z), goodlecturesfol(Y) , students(Z).}$

$\text{goodlecturesfol(X) :- goodlectureslogic(X).}$

Query:

?- goodteacher(Yannick).

Yes

?- goodteacher(Z).

$Z = \text{Yannick}$