Lecture 8

Prolog

Prolog is a language for *logic programming*. Its concept is based on the First Order Logic.

8.1 Prolog Syntax

Constant Symbols – a word starting with a *lowercase* letter, or numbers e.g. `john`, `richard`, `mickey`, `123`.

Variable Symbols – a word starting with a *uppercase* letter, or an underscore, e.g. `X`, `Person`, `_`.

Predicate Symbols – a predicate name starting with a *lowercase* letter e.g. `male(john)`, `likes(john,apple)`, `larger(mickey,X)`.

Connective Symbols – a comma (`,`) = and, a semicolon (`;`) = or, `\+` = not, `->` = implies.

Equality Symbol – `=`

Evaluation predicate – `is`

8.2 Hello, World!

```
?- write('Hello, World!').
Hello, World!
true.
```

8.3 Prolog Source File

A statement in Prolog can be categorized into two types: *fact* and *rule*.

8.3.1 Facts

Facts are represented by predicates. They show relations among object. Each predicate must end with a period (.).

Example 8.1 Facts showing the family relationships from the Bible.

```
father(terach,abraham).          father(terach,nachor).
father(terach,haran).            father(abraham,isaac).
father(haran,lot).               father(haran,milcah).
father(haran,yiscah).

male(terach).   male(abraham).   male(nachor).   male(haran).
male(isaac).    male(lot).

mother(sarah,isaac).

female(sarah).  female(milcah).  female(yiscah).
```

8.3.2 Rules

Rules are statements written in form of

$$A \leftarrow B_1, B_2, \ldots, B_n.$$

where $n \geq 0$. Here, A is the *head* of the rule, and $B_1, \ldots, B_n$ is the *body* of the rule. All $B_i (1 \leq i \leq n)$ are connected by commas. When all $B_i (1 \leq i \leq n)$ are true, we can conclude that A is true.

Example 8.2 Rules for the Biblical family.

```
/* X is a son of Y, if Y is a father of X and X is a male */
son(X,Y) :- father(Y,X), male(X).

/* X is a grandfather of Y, */
/* if X is a father of Z and Z is a father of Y */
grandfather(X,Y) :- father(X,Z), father(Z,Y).
```

8.4 Queries

Queries are for retrieving information from the loaded program. A query can be either a single predicate or a conjunction of predicates.

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.6.0)
Copyright (c) 1990-2013 University of Amsterdam, VU 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).

?- [bible].
% bible compiled 0.00 sec, 20 clauses
true.

?- father(abraham,isaac).
true.

?- father(abraham,haran).
false.

?- father(X,haran).
X = terach.

?- son(abraham,terach).
true.

?- son(lot,X).
X = haran.

?- son(X,haran).
X = lot ;
false.

?- father(haran,X).
X = lot ;
X = milcah ;
X = yiscah.
```

```
?- father(X,isaac),mother(Y,isaac).
X = abraham,
```

```
Y = sarah.

?- son(isaac,X),father(terach,X).
X = abraham.
```

Exercise 8.1 Translate the following English sentences into Prolog queries. Use the facts from the Biblical family.

1. Is **`terach`** is a father of **`nachor`**?

2. Is **`haran`** is a brother of **`abraham`**?

3. Who is a grandfather of **`isaac`**?

4. Who is a sister of **`lot`**?

Exercise 8.2 Write the output for the following queries using the given program.

```
isa(mickey,mouse). isa(minnie,mouse). isa(kitty,cat).
isa(doraemon,cat). isa(doraemon,robot). isa(nobita,boy).
isa(shizuka,girl).

isa(X,human) :- isa(X,boy).
isa(X,human) :- isa(X,girl).

eat(cat,meat). eat(mouse,cheese). eat(human,meat).
eat(human,vegetable). eat(doraemon,dorayaki).
eat(human,chicken).

carnivore(X)  :- isa(X,Y), (eat(Y,meat) ; eat(Y,chicken)).
```

1. `?- isa(doraemon,cat).`

2. `?- isa(nobita,X).`

3. `?- carnivore(mickey).`

4. `?- isa(X,human).`

5. `?- isa(X,robot), isa(X,cat).`

8.5 Query Evaluation

Prolog evaluates a query based on *matching between two terms* and *depth-first search.*

Example 8.3 Write the output of the following queries using the given program.

```
input(0).    input(1).

and(0,0,0).  and(0,1,0).  and(1,0,0).  and(1,1,1).
or(0,0,0).   or(0,1,1).   or(1,0,1).   or(1,1,1).
xor(0,0,0).  xor(0,1,1).  xor(1,0,1).  xor(1,1,0).

circuit(In1,In2,Out1,Out2) :- input(In1), input(In2),
    xor(In1,In2,Out1), and(In1,In2,Out2).
```

1. `?- circuit(1,1,X,Y).`

```
Resolvent = [circuit(1,1,X,Y)]
Choose circuit(1,1,X,Y)
    Match with circuit(In1,In2,Out1,Out2) :-
                    input(In1), input(In2),
                    xor(In1,In2,Out1), and(In1,In2,Out2).

Resolvent = [input(1), input(1), xor(1,1,X), and(1,1,Y)]
Choose input(1)
    Match with input(1).

Resolvent = [input(1), xor(1,1,X), and(1,1,Y)]
Choose input(1)
    Match with input(1).

Resolvent = [xor(1,1,X), and(1,1,Y)]
Choose xor(1,1,X).
    Match with xor(1,1,0). (X=0)

Resolvent = [and(1,1,Y)]
Choose and(1,1,Y).
    Match with and(1,1,1). (Y=1)

Resolvent = []
Return X=0, Y=1
```

2. ?- circuit(1,0,0,1).

```
Resolvent = [circuit(1,0,0,1)]
Choose circuit(1,0,0,1)
    Match with circuit(In1,In2,Out1,Out2) :-
                    input(In1), input(In2),
                    xor(In1,In2,Out1), and(In1,In2,Out2).

Resolvent = [input(1), input(0), xor(1,0,0), and(1,0,1)]
Choose input(1)
    Match with input(1)

Resolvent = [input(0), xor(1,0,0), and(1,0,1)]
Choose input(0)
    Match with input(0)

Resolvent = [xor(1,0,0), and(1,0,1)]
Choose xor(1,0,0)
    Unable to match

Return false
```

3. ?- circuit(X,Y,1,0).

```
Resolvent = [circuit(X,Y,1,0)]
Choose circuit(X,Y,1,0)
    Match with circuit(In1,In2,Out1,Out2) :-
                    input(In1), input(In2),
                    xor(In1,In2,Out1), and(In1,In2,Out2).

Resolvent = [input(X), input(Y), xor(X,Y,1), and(X,Y,0)]
Choose input(X)
    Match with input(0) (X=0)

Resolvent = [input(Y), xor(0,Y,1), and(0,Y,0)]
Choose input(Y)
    Match with input(0) (Y=0)

Resolvent = [xor(0,0,1), and(0,0,0)]
Choose xor(0,0,1)
    Unable to match

Backtrack
Resolvent = [input(Y), xor(0,Y,1), and(0,Y,0)]
Choose input(Y)
    Match with input(1) (Y=1)

Resolvent = [xor(0,1,1), and(0,1,0)]
Choose xor(0,1,1)
    Match with xor(0,1,1)

Resolvent = [and(0,1,0)]
Choose and(0,1,0)
    Match with and(0,1,0)

Resolvent = []
Return X=0, Y=1
```

```

----User inputs ';'
Backtrack
Resolvent = [input(Y), xor(0,Y,1), and(0,Y,0)]
Choose input(Y)
    No more option

Backtrack
Resolvent = [input(X), input(Y), xor(X,Y,1), and(X,Y,0)]
Choose input(X)
    Match with input(1) (X=1)

Resolvent = [input(Y), xor(1,Y,1), and(1,Y,0)]
Choose input(Y)
    Match with input(0) (Y=0)

Resolvent = [xor(1,0,1), and(1,0,0)]
Choose xor(1,0,1)
    Match with xor(1,0,1)

Resolvent = [and(1,0,0)]
Choose and(1,0,0)
    Match with and(1,0,0)

Resolvent = []
Return X=1, Y=0

----User inputs ';'
...
```

Exercise 8.3 Answer the questions from the following Prolog program.

1. Write the output of `?- circuit(1,0,X,Y).`

2. Write the output of `?- circuit(1,1,1,Y).`

3. Write the output of `?- circuit(X,Y,0,1).`

4. Write a rule **dosth(X,Y,Z)** where $Z = (X \wedge Y) \vee X$.

8.6 Recursive Rules

Some rules may need to use themselves recursively to describe a more general concept. From the Biblical family data, we can write a set of rules for representing the ancestors as:

```
   parent(X,Y) :- father(X,Y).
   parent(X,Y) :- mother(X,Y).
   grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
   greatgrandparent(X,Y) :- parent(X,Z), grandparent(Z,Y).
   greatgreatgrandparent(X,Y) :- parent(X,Z),greatgrandparent(Z,Y).
```

We can define rules of the general concept of ancestors by using recursion.

```
ancester(X,Y) :- parent(X,Y).                /* base case */
ancester(X,Y) :- parent(X,Z), ancester(Z,Y). /* recursive case */
```

Example 8.4 Define rules for **factorial(N,F)** which is a predicate "$N!$ is F".

```
factorial(0,1) :- !.
factorial(N,F) :- M is N-1, factorial(M,G), F is N*G.
```

Note: ! is the cut operator. It forces Prolog to stop backtracking when the operator is encountered.

Exercise 8.4 Write rules for **pow(X,Y,Z)** which is a predicate "X^Y is Z".

Example 8.5 A predicate **`edge(X,Y)`** is a fact representing a directed edge from node X to node Y. For example, **`edge(a,b)`** shows an edge from a to b, but not from b to a. The following program shows some example facts.

```
edge(a,b). edge(a,c). edge(b,c). edge(b,d).
edge(c,e). edge(c,f). edge(c,g).
```

Write rules for **`path(P,Q)`** which is a predicate "There is a directed path from node P to node Q". For example, **`path(a,b)`** is true, **`path(a,e)`** is true, but **`path(d,f)`** is false.

Example 8.6 From the previous path, write rules for **path(P,Q,R)** which is an extension of **path(P,Q)** with **R** showing how to get to **Q** from **P**. Here are some expected query results:

```
?- path(a,b,go(a,b)).
true .

?- path(a,c,X).
X = go(a, c) ;
X = go(a, b, go(b, c)) ;
false.
```

References

Russell, Stuart and Norvig, Peter *Artificial Intelligence A Modern Approach*, 3rd Edition, Prentice Hall 2010. ISBN-10 0-13-604259-7
Kari, Lila *Predicate Calculus (first order logic)*, Lecture notes.
Mooney, Raymond J. *First-Order Logic (First-Order Predicate Calculus)*, Lecture notes.