Learning notes of Prolog Note

Prolog 学习笔记 笔记

复旦大学软件学院逻辑程序设计课程笔记

整理：Jason Guo

整理时间：May 11, 2014

Email: zqguo@zqguo.com

Version: 1.00

目　　录

第 1 章

Prolog: A Beginner's Guide

The syntax and semantics of Prolog

1.1 General

A Prolog program usually contains of:

1. Some facts about objects.
2. Some rules about objects.
3. Asking questions about objects and how they relate.

Note: *Objects above means things. It's not the object in OOP.*

1.2 Fact

We should use lowercase letters and end the fact with a "." character, e.g.:

```
1 male(john).
2 female(mary).
3 likes(john, mary).
```

Note: *likes(john, mary). means whether john likes mary, but likes(mary, john) means whether mary likes john.*

In the example, "male", "female" and "likes" are called predicates. "mary" and "john" are called argument. We can use instruction "[filename]." to import the Prolog file in the current directory. Now we import a Prolog file name myfile.py:

```
1 likes(joe,fish).
2 likes(joe,mary).
3 likes(mary,book).
```

```
likes(john,book).
likes(john,france).
```

We can ask questions now:

```
?– likes(joe,fish).
true.

?– likes(joe,animals).
false.
```

1.3 Variable

Variables allow us to ask more complex questions. In Prolog, variables should begin with uppercase letter. With variables, we can ask questions like "who likes fish?" or "what does mary like" using instructions:

```
likes(WhatPerson, fish).
likes(mary, Something).
```

Prolog finds facts that match from top to bottom. If there are more than one answer, you can input ";" to find all facts that match.

Note: *";" means "OR". And "," means "AND".*

Sometimes in Prolog we want to ask a question but not want to know what a variable actually unified with. In this case we use *anonymous variable* "_".

1.4 Rules

A rule allow us to deduce some facts from current facts that we have known.

```
son_of(X, Y) :– parent(Y, X), male(X).
```

Note: *The ":-" means "if". The part before ":-" is the head. The part after ":-" is body. And "," means "AND".*

1.5 Recursion in Prolog

Some facts of fimaly:

```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
female(pam).
female(liz).
female(ann).
female(pat).
male(bob).
male(tom).
male(jim).
```

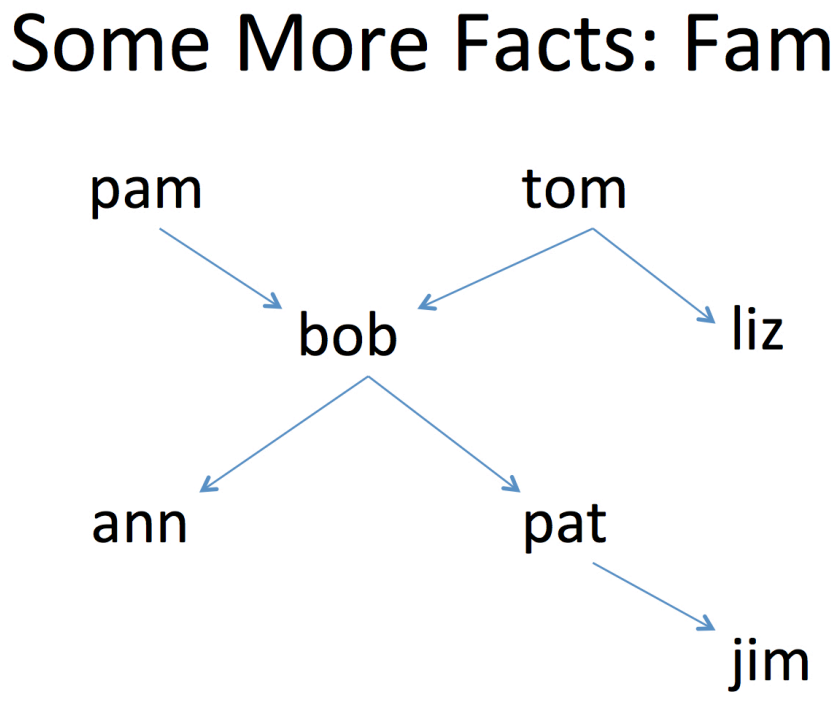

图 1.1: The structure of fimily

There are four predecessor relationship rules:

```
pred1(X,Z) :-
    parent(X,Z).
pred1(X,Z) :-
    parent(X,Y),
```

```
    pred1(Y,Z).
?- pred1(X,jim).
X = pat;
X = pam;
X = tom;
X = bob;
false.

pred2(X,Z) :-
    parent(X,Y),
    pred2(Y,Z).
pred2(X,Z) :-
    parent(X,Z).

pred3(X,Z) :-
    parent(X,Z).
pred3(X,Z) :-
    pred3(X,Y),
    parent(Y,Z).

pred4(X,Z) :-
    pred4(X,Y),
    parent(Y,Z).
pred4(X,Z) :-
    parent(X,Z).
```

第 2 章

Objects States Arithmetic

Data objects contain simple objects and structures. Simple objects contain constants and variables. And atoms and numbers are contained in constants.

2.1 Atoms & Numbers

Simple constant, non-numeric values. Start with *lowercase letter*. Use letters, numbers or "_". You can also have strings, which can start with an uppercase leJer and can contain spaces but are then enclosed in single quotes ''.
Prolog allows integers and real numbers. They are wriJen without any quotes. And Real numbers are rarely used.

2.2 Structures

The general form of structure in Prolog is functor(arg1, arg2, . . .).
There is a nice example. We can denifinition vertical and query it like this:

```
1 vertical(seg(point(X, _), point(X, _))).
2
3 ?- vertical(seg(point(1,1),point(1,2))).
4 true.
5 ?- vertical(seg(point(1,1),point(2,Y))).
6 false.
```

2.3 Arithmetic in Prolog & Relational Operators

Prolog provides us with the standard arithme=c operators: +, -, *, /, ** (exponent), div (integer division) and mod. Please use "is" to present "=". So my_plus(X,Y,Z) :- Z is X + Y. can get correct answer, but my_pus(X,Y,Z) :- Z = X + Y. will get wrong things.
Prolog provides the following relational operators: >, <, >=, <=, =:=, =\=

第 3 章

List

3.1 Definition of Prolog

Prolog provides us with a data structure called list. A list is a sequence of any number of items: it is enclosed in square brackets, like [1, 2, 3], [fred, mary, peter, john] and [].

3.2 Basic Operators of Prolog

Prolog does not provide us with a while or for loop. But it let us write the list as [Head | Tail]. "Head" will match the first item in the list, and "Tail" will match the remainder of the list. We can use the vertical bar more generally, like this: [a | [b,c]], [a,b | [c]], [a,b,c | []]. They all refer to the same list. Before the pipe is one or more objects, and after the pipe is a list.
Common operations: member(Element, List), append(List1, List2, Answer), delete(List, Element, Answer).

```
1  my_member(X, [X | Tail]).
2  my_member(X, [Head | Tail]) :-
3      my_member(X, Tail).
4
5  my_conc([],L,L).
6  my_conc([X|L1], L2, [X|L3]) :-
7      my_conc(L1,L2,L3).
8
9  del_all1(X, [], []).
10 del_all1(X,[X | Rest], L1) :-
11     del_all1(X,Rest,L1).
12 del_all1(X,[Y|Rest],[Y|L1]) :-
13     X \= Y,
14     del_all1(X,Rest,L1).
```

第 4 章

Cut

4.1 Basic Concept

Prolog can automatically backtracks. But sometimes we might want to control the backtracking ourself. So Prolog provides the "cut" operation, written "!".
When the conditions are mutually exclusive, only one can succeed. So we can add "!" to prevent Prolog from ever backtracking to find an alternative.

4.2 More Cut

There are some facts and rules:

```
1  data(one).
2  data(two).
3  data(three).
4
5  cut_test_a(X) :- data(X).
6  cut_test_a('last clause').
7
8  cut_test_b(X) :- data(X), !.
9  cut_test_b('last clause').
10
11 cut_test_c(X,Y) :- data(X), !, data(Y).
12 cut_test_c('last clause').
13
14 % Query results
15 ?- cut_test_a(X), write(X), nl, fail.
16 one
17 two
```

```
three
last clause
false.

?– cut_test_b(X), write(X), nl, fail.
one
false.

?– cut_test_c(X,Y), write(X–Y), nl, fail.
one–one
one–two
one–three
false.
```

In *cut_test_a*, there is no *cut*, so query will backtrack every fact of *data*, and print them. In *cut_test_b*, *cut* prevent backtracking, so swipl only prints "one". In *cut_test_c*, *cut's* left succeeds and prevents backtracking neither the *data* on *cut's* left nor cut_test_c, but the *data* on *cut's* right can backtrack, so swipl prints "one-one", "one-two" and "one-three".

Another example:

```
max1(X, Y, X) :– X >= Y, !.
max1(X, Y, Y).

max2(X, Y, Max) :–
    X >= Y, !, Max = X;
    Max = Y.

max3(X, Y, X) :– X >= Y, !.
max3(X, Y, Y) :– X < Y.

% Query with (3, 1, 1).
?– max1(3, 1, 1).
true.

?– max2(3, 1, 1).
false.

?– max3(3, 1, 1).
```

```
19 false.
```

The *max1* returns true because (3, 1, 1) matches the *max1(X, Y, Y)*.. *X >= Y* succeeds, so *cut* prevents *max2* from backtracking and *cut* stop the swipl querying instruction (*Max = Y*) after semicolon. (Does the swipl query "OR" by using backtracking?). I think *max3* is easy to understand. Let's skip it.

When a cut has no effect on the declara=ve meaning of the clause, it is considered a green cut. When a cut has an effect on the declara=ve meaning of the clause, it is considered a red cut. If you remove red cut, the program works differently.

4.3 Build in Predicates

not is problematic. Suppose we ask Prolog: *?- not(human(mary))*. This does not mean "mary is not human". It means "mary cannot be proven to be human". Prolog will try to see if it can prove that mary is human. But if there is no way to achieve this goal then it will conclude that it is false.This is called the *Closed World assumption*.

```
1  good_quality(coat).
2  good_quality(jacket).
3  expensive(coat).
4  bargain(X) :- not(expensive(X)).
5
6  ?- good_quality(X), bargain(X).
7  X = jacket.
8
9  ?- bargain(X), good_quality(X).
10 false.
```

not(expensive(X)) for all X: not(expensive(X)), not exists X such that not(expensive(X)).

Other build in predicates:

1. var(X) is true is X is an uninstantiated variable.
2. nonvar(X) is true if X is instantiated.
3. atom(X) is true if X currently stands for an atom.
4. number(X) means X currently stands for a number.
5. atomic(X) means X is a number or an atom.
6. compound(X) means X is a structure.

第 5 章

Sort & Logic Puzzles

5.1 Sort

Insert Sort

```
insertsort([], []).
insertsort([X | Tail], Sorted) :-
    % sort the tail
    insertsort(Tail, SortedTail),
    % insert x in correct place
    insert(X, SortedTail, Sorted).

insert(X, [Y | Sorted], [Y | Sorted1]) :-
    X > Y, !,
    insert(X, Sorted, Sorted1).
insert(X, Sorted, [X | Sorted]).
```

Bubble Sort

```
bubblesort(List, Sorted) :-
    swap(List, List1), !,
    bubblesort(List1, Sorted).
bubblesort(Sorted, Sorted).

swap([X, Y | T], [Y, X | T]) :-
    X > Y.
swap([Z | T], [Z | T1]) :-
    swap(T, T1).
```

Merge Sort

```
mergesort([], []).
mergesort([A], [A]).
```

```
3  mergesort(L, Sorted) :-
4      my_divide(L, L1, L2),
5      mergesort(L1, S1),
6      mergesort(L2, S2),
7      my_merge(S1, S2, Sorted), !.
8
9  my_divide([], [], []).
10 my_divide([A], [A], []).
11 my_divide([A, B | R], [A | Ra], [B | Rb]) :-
12     my_divide(R, Ra, Rb).
13
14 my_merge(L, [], L).
15 my_merge([], L, L).
16 my_merge([X | T1], [Y | T2], [X | Merged]) :-
17     X < Y, !,
18     my_merge(T1, [Y | T2], Merged).
19 my_merge([X | T1], [Y | T2], [Y | Merged]) :-
20     my_merge([X | T1], T2, Merged).
```

Quick Sort

```
1  quicksort([], []).
2  quicksort([X | Tail], Sorted) :-
3      split(X, Tail, Small, Big),
4      quicksort(Small, SortedSmall),
5      quicksort(Big, SortedBig),
6      append(SortedSmall, [X | SortedBig], Sorted).
7
8  split(_, [], [], [] ).
9  split(X, [Y | Tail], [Y | Small], Big) :-
10     X > Y, !,
11     split(X, Tail, Small, Big).
12 split(X, [Y | Tail], Small, [Y | Big]) :-
13     split(X, Tail, Small, Big).
```

5.2 Logic Puzzle

I think it is very insteresting, and I will give three examples provided by my teacher.

Puzzle 1:

Four ladies meet weekly on Thursdays to play bridge. On each meeting they decide what everybody had to bring for the next meeting. For next week: Mrs. Andrews will bring chocolate cake; Neither Mrs. Brown nor Vivien nor Ann Clark will bring cookies; Mary will not bring wine; Rachel, who is not from the Davidson family, will bring coffee.

Find the whole name of each lady and what she is supposed to bring next week.

How can we model this?

We want Prolog to find details about four people. For each person we want 3 pieces of information: family name, given name and what they will bring. For this we can use a list with three elements in it to represent each person. The final result will be a list containing four of these lists.

```
1  sol(Final):-
2  Final=[[_, _, wine], [_, _, cookie],
3      [_, _, coffee], [_, _, cake]],
4  member([andrews, _, cake], Final),
5  member([brown, _, Bb], Final), Bb \== cookie,
6  member([_, vivien, Bv], Final), Bv \== cookie,
7  member([clark, ann, Ba], Final),  Ba \== cookie,
8  member([_, mary, Bm], Final),  Bm \== wine,
9  member([davidson, _, _], Final),
10 member([X, rachel, coffee], Final), X \== davidson.
```

Puzzle 2:

1) There are 5 coloured houses in a row, each having an owner, which has an animal, a favourite food, a favourite drink.

2) The English lives in the red house.

3) The Spanish has a dog.

4) They drink coffee in the green house.

5) The Ukrainian drinks tea.

6) The green house is to the right of the white house.

7) The potato eater has a serpent.

8) In the yellow house they eat rice.

9) In the middle house they drink milk.

10) The Norwegian lives in the first house from the lep.

11) The noodle eater lives beside the man with the fox.

12) In the house beside the house with the horse they eat rice. 13) The tofu eater drinks juice.

14) The Japanese eat sushi.

15) The Norwegian lives beside the blue house.

"The English lives in the red house": *member([english,_,_,_,red], Sol).* "They drink coffee in the green house": *member([_,_,_,coffee,green], Sol).* "In the middle house they drink milk": *Sol=[_,_,[_,_,_,milk,_],_,_].*

To describe relational position like "The green house is to the right of the white house" and "The noodle eater lives beside the man with the fox", we need to define predicates for these situations:

```
1  right(X, Y, L):-
2      append(_, [Y, X | _], L).
3
4  beside(X, Y, L):-
5      right(X, Y, L); right(Y, X, L).
```

The whole solution:

```
1  start(Sol):- length(Sol, 5),  %1
2  member([english, _, _, _, red], Sol),  %2
3  member([spanish, dog, _, _, _], Sol),  %3
4  member([_, _, _, coffee, green], Sol),  %4
5  member([ukrainian, _, _, tea, _], Sol),  %5
6  right([_, _, _, _, green], [_, _, _, _, white],  Sol),  %6
7  member([_, snake, potato, _, _], Sol),  %7
8  member([_, _, rice, _, yellow], Sol),  %8
9  Sol= [_, _, [_, _, _, milk, _], _, _],  %9
10 Sol= [[norwegian, _, _, _, _], _, _, _, _],  %10
11 beside([_, _, noodle, _, _], [_, fox, _, _, _], Sol),  %11
12 beside([_, _, rice, _, _], [_, horse, _, _, _],  Sol),  %12
13 member([_, _, tofu, juice, _], Sol),  %13
14 member([japanese, _, sushi, _, _], Sol),  %14
15 beside([norwegian, _, _, _, _], [_, _, _, _, blue], Sol),  %15
16 member([_, _, _, water, _], Sol),  % someone drinks water
17 member([_, zebra, _, _, _], Sol). % someone has a zebra
```

Puzzle 3:

A bomb has 7 flip switches. To disarm the bomb, you must put the switches in their correct position. You know there are several situations that are not safe, so the bomb will go off.

BOOM.

If switch 3 is on plus 2 and 4 are off, boom!

If 1 and 4 off plus 7 is on, boom!

If 1, 3 and 4 are off, boom!

If 6 is off plus 2 and 3 are on, boom!

If 4 and 3 are on, boom!

If 6 is on and if, as long as 7 is on, 1 is on too, boom!

If 6 is on and, provided 7 is on, 1 is on too, boom!

If 1 and 5 are on plus 7 is off, boom!

If 3 is off plus 4 and 5 are on, boom!

If 1 and 7 are on, boom!

If 5 off and if, as long as 2 and 6 are on, 3 is on too, boom!

If 7 off plus 3 or 4 on, boom!

If switch 6 and 7 are in different posi=ons, boom!

If switch 2, 3 and 5 are off, boom!

If switch 1 and 2 are on and switch 5 and 7 are off, boom!

If 6 and 7 are off, boom!

Solution:

```
1  boom([_, 0, 1, 0, _, _, _]).
2  boom([0, _, _, 0, _, _, 1]).
3  boom([0, _, 0, 0, _, _, _]).
4  boom([_, 1, 1, _, _, 0, _]).
5  boom([_, _, 1, 1, _, _, _]).
6  boom([1, _, _, _, _, 1, 1]).
7  boom([1, _, _, _, 1, _, 0]).
8  boom([_, _, 0, 1, 1, _, _]).
9  boom([1, _, _, _, _, _, 1]).
10 boom([_, 1, 1, _, 0, 1, _]).
11 boom([_, _, 1, 1, _, _, 0]).
12 boom([_, _, _, _, _, 0, 1]).
13 boom([_, _, _, _, _, 1, 0]).
14 boom([_, 0, 0, _, 0, _, _]).
15 boom([1, 1, _, _, 0, _, 0]).
16 boom([_,_,_,_,_,0,0]).
17
18 defuse([A, B, C, D, E, F, G]) :-
19     binlist([A, B, C, D, E, F, G]),
```

```
20      not(boom([A, B, C, D, E, F, G])).
21
22  binlist([]).
23  binlist([X|T]) :-
24      member(X, [0 ,1]),
25      binlist(T).
```