TUTORIAL 5

CSCI3230 (2019-2020 First Term)

By Ran WANG (rwang@cse.cuhk.edu.hk)

Outline

- Built-in Predicates
- Control the flow of satisfaction
- Examples

BUILT-IN PREDICATES

Built-in Predicates

- Most Prolog systems provide many built-in predicates and functions:
  - Arithmetic functions (+, -, mod, is, sin, cos, floor, exp, …)
  - Bit-wise operations (/\, \/, \, <<, >>, xor)
  - Term comparison (==, \==, @<, @>, …)
  - Input/Output (read, write, nl, …)
  - Control
  - Meta-logical
  - …

For more: http://www.swi-prolog.org/pldoc/man?section=builtin

Built-in Predicates

Example 1

```
?- X is sin(2*pi). %Built-in constant and function sin
X = -2.4492127076447545e-16.
?- X is 4 >> 1.     %Bitwise right shift
X = 2
?- Y @< b.
true.
?- read(X),r                                               ).
|: 2.                                                      .
|: f(pi).
Complete
X = 2,
Y = pi,
Z = 1.0.
```

Standard order:
Variables < Numbers < Atoms < Compound Terms

- **Variables** are sorted by address.
- ***Numbers*** are compared by value.
- ***Atoms*** are compared alphabetically.
- ***Compound*** **terms**
  - ❖ first checked on arity
  - ❖ then on functor name (alphabetically)
  - ❖ finally recursively on their arguments, left to right.

Equivalence

Operator	Meaning	Description
TermA == TermB	Testing for equivalence	A variable is only identical to a sharing variable.
TermA =@= TermB	Testing for a variant (or structurally equivalence)	True iff there exists a renaming of the variables in A that makes A equivalent (==) to B and vice versa.
TermA = TermB	Testing for unification	True if the unification succeeds, and the terms in A and B will be unified.
TermA is TermB	Testing for numerical value	True if both terms has the same numerical value after evaluation of TermB.

For more: http://www.swi-prolog.org/pldoc/man?section=compare

Equivalence

Example 2-1 is

```
?- X is 6+3, S is 9.
X = 9,
S = 9.
?- X is 9, S is 10, X is S.
false.
?- 1+3 is 1+3.
false. %Evaluation only on the last term of is
```

is	numerical value
==	equivalence
=@=	variant
=	unification

Example 2-2 == & =@=

```
?- f(A,B) == f(A,B).
true.
?- f(A,B) == f(X,Y).
false.
?- X=A, Y=A, X==Y.
true.
?- f(A,B) =@= f(X,Y).
true.%Renaming A to X, B to Y
```

Example 2-3 =@= & =

```
?- f(A,B) =@= f(A,b).
false. %B is var, b is atom
?- f(A,B) = f(a,b).
A = a,
B = b. %Can be unified
```

Test yourself

TermA =@= TermB:

- Checking for variant
- True iff there exists a renaming of the variables in A that makes A equivalent (==) to B and vice versa.

1. a =@= A
2. A =@= B
3. x(A,A) =@= x(B,C)
4. x(A,A) =@= x(B,B)
5. x(A,A) =@= x(A,B)
6. x(A,B) =@= x(C,D)
7. x(A,B) =@= x(B,A)
8. x(A,B) =@= x(C,A)

Assert and Retract

- Modify a (running) program *during execution*
  - NOT encouraged unless you have some good reasons, e.g. memorization.
- **ASSERT** to insert a fact or rule
- **RETRACT** to remove a fact or rule
  - Abolish is evil.

Example 3

```
?- assert(color(apple,red)).
true.
?- color(apple,red).
true.
```

For more http://www.swi-prolog.org/pldoc/man?section=dynpreds

findall(Object,Goal,List).

- Produces a list *List* of all the objects *Object* that satisfy the goal *Goal*.

Example 6

```
dessert(froyo).
dessert(lava_cake).
dessert(marble_cake).
likes(mary,froyo).
likes(mary,lava_cake).
likes(mary,banana).
likes(kate,froyo).
likes(kate,marble_cake).
likes_dessert(P,F):-dessert(F),likes(P,F).
```

```
?- findall(F,likes_dessert(mary,F),L).
L = [froyo, lava_cake].
?- findall(F,likes_dessert(P,F),L).
L = [froyo, froyo, lava_cake, marble_cake].
```

CONTROL THE FLOW OF SATISFACTION

Cut !

Fail

Control the Flow of Satisfaction

- The semantics of Prolog programs does not care about order
- Conjunction is commutative
  - E.g. `P :- Q,R,S.` should mean the same as `P :- S,R,Q.` logically
- In practice
  - the order matters
  - most Prolog systems use left to right DFS, top to bottom order
- To control the order of matching for query
  - Place the facts and rules in a suitable sequence
  - Use **!** and **fail** operator

Recap: Backtracking

- When asked `P`$_1$`(..),P`$_2$`(..),…,P`$_n$`(..).`
  - If anyone fails (due to instantiation), say `P`$_i$, Prolog backtracks, and try an alternative of `P`$_{i-1}$
- After a successful query,
  - If user presses '`;`', backtrack and try alternatives.

Tutorial 4: Example 6

```
likes(mary,donut). %Fact 1
likes(mary,froyo). %Fact 2
likes(kate,froyo). %Fact 3
```

false

```
?- likes(mary,F),likes(kate,F). %Sth both Mary and Kate like
F = froyo.
```

backtrack

The prolog program will record the choice points for the goals to backtrack.

Cut !

- **!** is used for search control
  - When it is first encountered as a goal, it succeeds
  - Discard all choice points created since entering the predicate in which the cut appears.

Example 4-1

Discard choice points

```
award(Winner,Prize,P1Got,_)   :- Winner is 1, !, P1Got = Prize.
award(_,Prize,_,P2Got)        :- P2Got = Prize.
```

```
?- award(1,apple,P1Got,P2Got).
P1Got = apple.
?- award(2,apple,P1Got,P2Got).
P2Got = apple.
```

Stop

Without Cut

Example 4-2 (Remove !)

```
award(Winner,Prize,P1Got,_)   :- Winner is 1, P1Got = Prize.

award(_,Prize,_,P2Got)        :- P2Got = Prize.
```

```
?- award(1,apple,P1Got,P2Got).
P1Got = apple ;
P2Got = apple.
```

Illustrating Cut: Code View

X=1 Y=1 Z=1

p(X,Y,Z):-P_{01}(X),P_{02}(X,Y),P_{03}(X), P_{04}(X,Y,Z).

X=1
X=2 Y=1 stop Z=1

p(X,Y,Z):-P_{11}(X),P_{12}(Y,X),P_{13}(X),!,P_{14}(X,Z).

Ignore these predicates

p(X,Y,Z):-P_{21}(X),P_{22}(Z,X),P_{23}(Y), P_{24}(X).

Ignore these predicates

…

Cut !

Example 4-3

```
max(X,Y,X) :- X > Y.
max(X,Y,Y) :- X < Y.
```

max(X, Y, Max)

```
?- max(6,3,Max).
Max = 6.
```

Example 4-4

```
max(X,Y,X) :- X > Y, !.
max(_,Y,Y).
```

```
?- max(6,3,Max).
Max = 6.
```

- Reduce memory usage as less backtracking points are stored
- Checking less rules or facts in the database

Fail

- **<u>FAIL</u>** is a predicate which is always false.

<u>Example 5</u>

```
illegal(X,Y) :- X = Y, !, fail.
illegal(X,Y). %Is illegal iff X and Y cannot be unified
```

```
?- illegal(a,b).
true.
?- illegal(a,B).
false. %NOT illegal, Atom a can be unified with variable B
```

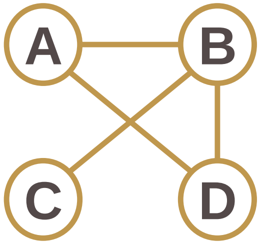
What if we just remove the “fail”?

```
illegal(X,Y) :- X = Y, !.
illegal(X,Y).
```

EXAMPLES

- Undirected graph
- Two predicates of list: membership, append

Example: Links in Graph




```
1.  link(a,b).
2.  link(b,c).
3.  link(a,d).
4.  link(b,d).
5.  link(X,Y):-
    link(X,Z),link(Z,Y).
```

```
?- link(a,K).
K = b ;
K = d ;
K = c ;
K = d ;
ERROR: Out of local stack
```

In our usage of *link*, it always matches to the fifth rule, which means the base case does not work because the problem will be decomposed into smaller sub-problems.

Explanation

link(a,K).

1. Matches 1, Return K=b, Press ;
2. Matches 3, Return K=d, Press ;
3. Match 5

   New sub-goal: link(a,Z), link(Z,K).

   1. link(a,Z) matches 1, unified Z to b.
   2. link(b,K) matches 2. Return K=c., Press ;
   3. link(b,K) matches 4. Return K=d., Press ;

   New sub-goal: link(b,Z), link(Z,K).

   1. link(b,Z) matches 2, unified Z to c.
   2. link(c,K) matches 5,

   New sub-goal: link(c,Z),link(Z,K).

   1. link(c,Z) matches 5. (let Z be Z_{old})

   New sub-goal: link(c,Z),link(Z,Z_{old}).

   1. link(c,Z) matches 5.

   …

   (loop forever)

Renaming facts and rules

```
link(a,b).
link(b,c).
link(a,d).
link(b,d).
path(X,Y):-link(X,Y);link(Y,X). %Single hop
path(X,Y):-link(X,Z),link(Z,Y). %More than one hop
path(X,Y):-link(Z,X),link(Z,Y).
path(X,Y):-link(X,Z),link(Y,Z).
path(X,Y):-link(Z,X),link(Y,Z).
```

```
link(a,b).
link(b,c).
link(a,d).
link(b,d).
edge(X,Y):-link(X,Y);link(Y,X).
path(X,Y):-edge(X,Y).                %Single hop
path(X,Y):-edge(X,Z),edge(Y,Z).      %More than one hop
```

Example: Membership

- Define `member(X,Y)` to be true iff `X` (a term) is a member of the list `Y`

Example 6

```
member(X,[X|_]). %Recall [a] is equivalent to [a|[]]
member(X,[_|T]) :- member(X,T).
```

```
?- member(s,[f,i,s,h]).
true ;
?- member(X,[f,i,s,h]).
X = f ;
X = i ;
X = s ;
X = h ;
false.
```

Example: Membership

Example 6

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

```
?- member(X,[f,i,s,h]).
```

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

```
member(X,[f|[i,s,h]])     X=f
member(X,[f|[i,s,h]]) T=[i,s,h]
```

New sub-goal: `member(X,[i,s,h])`

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

```
member(X,[i|[s,h]])     X=i
member(X,[i|[s,h]])     T=[s,h]
```

New sub-goal: `member(X,[s,h])`

`X=s`

`T=[h]`

New sub-goal: `member(X,[h])`

```
member(X,[X|_]).
member(X,[_|T]) :- member(X,T).
```

```
member(X,[h|[]])     X=h
member(X,[h|[]])     T=[]
```

New sub-goal: `member(X,[])`

`false`

Example: Append

Given two lists, append one list to another.
Example: [a,b] + [c,d] → [a,b,c,d]

Example 7

```
append([],Y,Y).
append([H|T],Y,[H|Z]):-append(T,Y,Z).
```

```
?- append([],[a,b,c],Z).
Z = [a,b,c].
?- append([a,b],[c,d],Z).
Z = [a,b,c,d].
?- append(M,N,[a,b,c,d]).
M = [],
N = [a, b, c, d] ;
M = [a],
N = [b, c, d] ;
M = [a, b],
N = [c, d] ;
M = [a, b, c],
N = [d] ;
M = [a, b, c, d],
N = [] ;
false.
```

N is deduced after fixing the elements in M.

Different combinations of M and N to produce [a,b,c,d]

Example: Append

Given two lists, append them and return the product. Example: [a,b] + [c,d] → [a,b,c,d]

Example 7

```
append([],Y,Y).
append([H|T],Y,[H|Z]):-append(T,Y,Z).
```

```
?- append(M,N,[a,b,c,d]).
```

```
append([],Y,Y).

append([H|T],Y,[H|Z]):-
append(T,Y,Z).
```

```
append(M,N,[a,b,c,d])
                                   M=[],N=[a,b,c,d]
 ;
append([a|T],Y,[a|[b,c,d]])
                   M=[a|T],N=Y,H=[a],Z=[b,c,d]
```

New sub-goal: `append(T,Y,[b,c,d])`

```
append([],Y,Y).

append([H|T],Y,[H|Z]):-
append(T,Y,Z).
```

```
append(T,Y,[b,c,d])
;          T=[],M=[a|T]=[a],N=Y=[b,c,d]
append([b|T1],Y,[b|[c,d]])
                      H=[b],T=[b|T1],Z=[c,d]
```

New sub-goal: `append(T1,Y,[c,d])`

```
(M=[a|[b|T1]])
```

Example: Append

Given two lists, append them and return the product. Example: [a,b] + [c,d] → [a,b,c,d]

New sub-goal: `append(T1,Y,[c,d])`

```
append([],Y,Y).

append([H|T],Y,[H|Z]):-
append(T,Y,Z).
```

```
append([],Y,[c,d])
; T1=[],M=[a|[b|T1]]=[a,b],N=Y=[c,d]
append([c|T2],Y,[c|d])
                    H=[c],T1=[c|T2],Z=[d]
                    (M=[a|[b|[c|T2]]])
```

New sub-goal: `append(T2,Y,[d])`

```
append([],Y,Y).

append([H|T],Y,[H|Z]):-
append(T,Y,Z).
```

```
append([],Y,[d])
       T2=[],M=[a|[b|[c|T2]]]=[a,b,c],
       N=Y=[d]
;
append([d|T3],Y,[d|[]])
                    H=[d],T2=[d|T3],Z=[]
                    (M=[a|[b|[c|[d|T3]]]])
```

New sub-goal: `append(T3,Y,[])`

Example: Append

Given two lists, append them and return the product. Example: [a,b] + [c,d] → [a,b,c,d]

New sub-goal: `append(T3,Y,[])`

```
append([],Y,Y).



append([H|T],Y,[H|Z]):-
append(T,Y,Z).
```

```
append([],Y,[])
        T3=[], N=Y=[]
;       M=[a|[b|[c|[d|T3]]]])=[a,b,c,d]
append(T3,Y,[])     false
```

Summary

- Build-in Predicates
  - ==, =@=, …
- Flow of Satisfaction
  - Order
  - Cut !
  - Fail
- Examples
  - Undirected graph
  - Membership
  - Append

Try it yourself

- Given a list L of integer, write *findmax(L,Ans)* to find the largest one and stored it in Ans.
- Tower of Hanoi: Move N disks from the left peg to the right peg using the center peg as an auxiliary holding peg. At no time can a larger disk be placed upon a smaller disk. Write *hanoi(N)*, where N is the number of disks on the left peg, to produce a series of instructions, e.g. “move a disk from left to middle”.
- Fill in a 3x3 grid with number from 1-9 with each number appearing once only. Write a *puzzle3x3(Ans)* to do this. The answer in Ans is a list, e.g. [1 2 3 4 5 6 7 8 9].

Hands on Lab

Reference

- Reference manual of SWI-Prolog
  - http://www.swi-prolog.org/pldoc/refman/
- More advanced Prolog
  - The Craft of Prolog by Richard A. O'Keefe
- A debug technique
  - http://stackoverflow.com/questions/13111591/prolog-check-if-two-lists-have-the-same-elements