CSC384: Intro to Artificial Intelligence
Prolog Tutorials 1&2

Prolog Terms

- TERMS
  - **Constants**: e.g. mary, foo, george, 1, 2.
  - **Variables**: anything that starts with an UPPER CASE letter or _: e.g. X, Mary, Foo, XXX, X2, _A, _
  - **Function application**: <constant_symbol>($Term_1$, …, $Term_k$).
    - A constant symbol (the name of the function), followed by k arguments each of which is a term (perhaps function applications themselves).
    - Examples: foo(a,b), f(g(a),-(2,3)), +(1,2), +(/(6,3),*(2,2))
    - Prolog also allow you to specify that a function can be expressed *infix* rather than *prefix*. All of the arithmetic functions are so defined. So the last two examples could also be expressed as:
      - 1+2, 6/3 + 2*2

Prolog Terms …

- Many things in prolog are actually terms built up by function application, e.g., **lists** are complex terms built up by function application.
  - [a, b, c] ➔ cons(a, cons(b, cons(c, [])))
    Where [] is a predefined constant symbol.
  - However prolog does not recognize the name "cons". Instead it uses the infix function '|' and square brackets '[', ']'
    - [c] ➔ [c | []]
    - [b, c] ➔ [b | [c | []]]
    - [a,b,c] ➔ [a | [b | [c | []]]]

Prolog Predicates

- PREDICATES: A constant symbol (the name of the predicate), followed by k arguments each of which is a term:

  <constant_symbol>(Term1, …, Termk).
  - Same syntax as function application. But the meaning is different. A predicate can either be **true** of **false**.
  - Usually referred to as *name/n* where n is the arity.
  - SWI has a large number of useful built in predicates, e.g. consult/1, halt/0, </2, number/1 .
  - User defined predicates like `male(albert), parent(albert, bob), book(AIMA,green,2003).`
    - `Again numeric predicates like >, <, can be written in infix notation, so instead of >(3*3, 4*4) we can write 3*3 > 4*4`

Prolog Rules

- RULES
  - predicate0 :- predicate1, predicate2, …, predicatek.
  - This is essentially a horn clause that logically means
    predicate0 ⇐ predicate1 ∧ predicate2 ∧ … ∧ predicatek.
  - Predicate0 is called the HEAD of the rule, and the other predicates that follow the ':-' are the BODY of the rule. (A period has to follow the rule).
  - If k = 0, we have a rule of the form
    - predicate.
  - Such rules are called FACTS.

Prolog Programa and Queries

- A Prolog program is specified by writing a set of rules. The order in which the rules appear in the file is important.
- Given a prolog program we invoke a computation by entering a **QUERY**.
  - A **QUERY** is predicate0, predicate1, …, predicateN
    - A non-empty sequence of predicates.
  - Generally we enter a query by typing it into the prolog interpreter's command line. (But there are ways of putting queries into the program file so that the program automatically starts some computation when loaded.

Prolog program example

family.pl

```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(albert,edward).                %albert is parent of edward
Parent(victoria,edward).
father(X,Y):- parent(X,Y), male(X).   %X is father of Y
mother(X,Y):- parent(X,Y), female(X).
```

- We can comment a line with %
- For multiple-line commenting we use /* */
- Let's load this file in SWI…

Running SWI on CDF

```
skywolf:~% pl
Welcome to SWI-Prolog (Multi-threaded, Version 5.2.11)
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).

?-    ← this is the prompt. You can load you program and ask queries.
?- consult(family).    %loading file. We can use full-name with quotation 'family.pl'
                       % alternatively, we can use ['family.pl'] in front of the prompt.
?- halt.               %exiting SWI!
```

- For working from home download SWI for windows, it has a nice **GUI** and a graphical debugger! But make sure your program works on CDF before submitting your assignments!

Asking Some Queries

```
?-male(albert).
Yes.
?-male(victoria).
No.
?-male(z).
No.
?-male(X).
X=albert;   %we use ; to ask for more answers.
X=edward;
No
?-father(F,P).
F=….
```

```
male(albert).
male(edward).
female(alice).
female(victoria).
parent(albert,edward).
parent(victoria,edward).
father(X,Y):- parent(X,Y), male(X).
mother(X,Y):- parent(X,Y), female(X).
```


How Prolog Works: Unification

- To understand how prolog's computation works we have to understand unification.
- Two predicates q(t1, t2, …, tk) and p(s1, s2, …, sn) can be unified (MATCHED) if and only if
  1. q is the same symbol as p.
  2. k = n, i.e. they both have the same number of arguments.
  3. For each i (1,…,n) terms ti and si can be unified.
- The built in predicate '=' tests if two terms are unifiable.

Unification of two terms

- Unifying two terms ti and si
  - If both are constants, then they can be unified only if they are identical.
  - If one is an unbound variable X, then we can unify the two terms by binding X to the other term (i.e., we set X's value to be the other term).
    - E.g., X = f(a) ➔ yes X is bound to the value f(a).
  - If one or both are bound variables then we have to try to unify the values the variables are bound to
    - E.g. X = f(a), Y = X ➔ first X is bound to the value f(a), then when we try to unify Y with X, X is bound so we must Y with X's value, so Y also becomes bound to f(a).
  - If ti = f(x1,x2, …, xk) and si = g(y1, y2, …, ym) then ti and si can be unified if and only if
    - f is identical to g.
    - k = m (both functions take the same number of arguments).
    - xi and yi can be recursively unified for all i = 1…m

Unification Examples

- Which of the followings are unifiable:

		Bindings
X	f(a,b)	X=f(a,b)
f(X,a)	g(X,a)	
3	2+1	Use is to evaluate
book(X,1)	book(Z)	
[1,2,3]	[X|Y]	X=1, Y=[2,3]
[a,b,X]	[Y|[3,4]]	
[a|X]	[X|Y]	X=a Y=a improper list
X(a,b)	f(Z,Y)	
[X|Y|Z]	[a,b,c,d]	X=a. Y=b, Z=[c,d]

Solving Queries

- How Prolog works:
  - Unification
  - Goal-Directed Reasoning
  - Rule-Ordering
  - DFS and backtracking
- When given a query Q = q1, q2, …, qn Prolog performs a search in an attempt to solve this query. The search can be specified as follows

Details of Solving Queries by Prolog

```
//Variable bindings are global to the procedure.

bool evaluate(Query Q)
  if Q is empty
    SUCCEED: print bindings of all variables in original query
       if user wants more solutions (i.e. enters ';') return FALSE
       else return TRUE
  else
    remove first predicate from Q, let q1 be this predicate
    for each rule R = h :- r1, r2, ..., rj in the program in
                the order they appear in the file
      if(h unifies with q1 (i.e., the head of R unifies with q1))
        put each ri into the query Q so that if the Q was originally
        (q1, q2, ..., qk) it now becomes (r1, r2, ..., rj, q2, ... qk)
              NOTE: rule's body is put in front of previous query predicates.
              NOTE: also some of the variables in the ri's and q2…qk might
                now be bound because of unifying h with q1
         if (evaluate(Q) )   //recursive call on updated Q
            return.  //succeeded and printed bindings in recusive call.
```


Computing with Queries

```
for each rule R = h :- r1, r2, ..., rj in the program in
            the order the rules appear in the prolog file
  if(h unifies with q1 (i.e., the head of R unifies with q1))
    put each ri into the query Q so that if the Q was originally
    (q1, q2, ..., qk) it now becomes (r1, r2, ..., rj, q2, ... qk)
          NOTE: rule's body is put in front of previous query predicates.
          NOTE: also some of the variables in the ri's and q2…qk might
            now be bound because of unifying h with q1
     if(evaluate(Q) )
        return.  //succeeded and printed bindings in recusive call.
     else
        UNDO all changes to the variable bindings that arose from
        unifying h and q1

end for
     //NOTE. If R's head fails to unify with q1 we move on to try the
           next rule in the program. If R's head did unify but unable
           to solve the new query recursively, we also move on to
           try the next rule after first undoing the variable bindings.
```


Computing with Queries

```
end for
     //NOTE. If R's head fails to unify with q1 we move on to try the
           next rule in the program. If R's head did unify but unable
           to solve the new query recursively, we also move on to
           try the next rule after first undoing the variable bindings.
return FALSE
     //at this point we cannot solve q1, so we fail. This failure will
       unroll the recursion and a higher level recursion with then try
       different rule for the predicate it is working on.
```

Query Answering is Depth First Search

- This procedure operates like a depth-first search, where the order of the search depends on **the order of the rules** and predicates in the rule bodies.
- When Prolog backtracks, it undoes the bindings it did before.

Example: Let's see how queries on family.pl are answered:

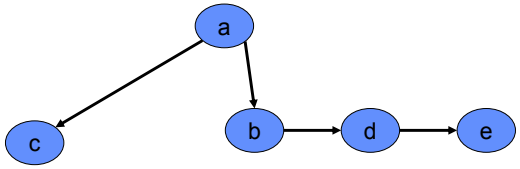
```
parent(A,B).
father(F).
```

[interactive SWI with graphical debugger]

Another Example

- Route finding in in a directed acyclic graph:

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




```
path(X,Y) :- path(X,Z), edge(Z,Y).
path(X,Y) :- edge(X,Y).
```

- The above is problematic. Why?
- Here is the correct solution:

```
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```


Search Tree for Path Predicate

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

path(X,Y) :- edge(X,Y).
path(X,Y) :-  edge(X,Z), path(Z,Y).
```

Queries:

```
path(a,e).    [Let's run SWI interactively.]
path(c,R).
path(S,c).
```

What is the graph is undirected?

```
undirEdge(X,Y):- edge(X,Y).
undirEdge(X,Y):- edge(Y,X).
```

Then replace edge predicate by undirEdge in the definition of path.

Notes on Prolog Variables

- Prolog variables do not operate like other programming languages. You cannot change the value of variables, once they are bound to a value they remain bound. However:
  - If a variable binding contains another variable that other variable can be bound thus altering the original variable's value, e.g.
    X = f(a, g(Y)), Y = b ➔ X is bound to f(a, g(b)); Y is bound to b
  - Final answers can be computed by passing the variable's value on to a new variable. E.g.,
    X = a, Y = b, Z = [X,Y] ➔ X is bound to a, Y is bound to b, and Z is bound to [a,b].

List Processing in Prolog

- Much of prolog's computation is organized around lists. Two key things we do with a list is iterate over them and build new ones.
- E.g. checking membership:
  member(X,Y) X is a member of list Y.

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

What if we define member like this:

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

what is the result of member(X,[a,b,c,d])?

- E.g. building a list of integers in range [i, j].
  build(from, to, NewList)

```
build (I,J,[ ]) :- I > J.
build (I,J,[I | Rest]) :- I =< J, N is I + 1,
                          build(N,J,Rest).
```