Programming Assignment 2
(Inductive Logic Programming)
Due 10/24/2017 on e-learning

- This homework is courtesy of Dr. David Page and is taken from his ILP homework at following link: http://pages.cs.wisc.edu/~dpage/ilplab.html

Exercise 1

Make sure you can start Progol. The executable is:

```
~cs731-1/public/progol/progol
```

If you are successful, you should see a prompt that looks something like:

```
CProgol Version 4.2
|-
```

Exercise 2

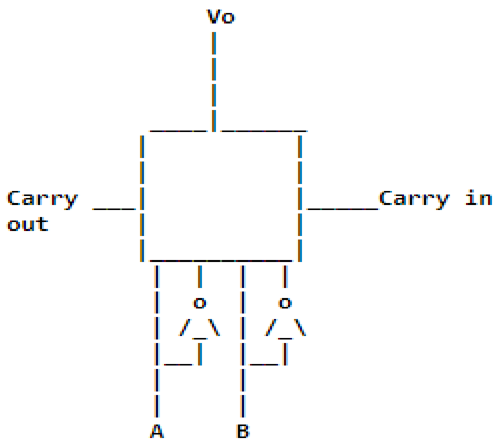
Try the following command:

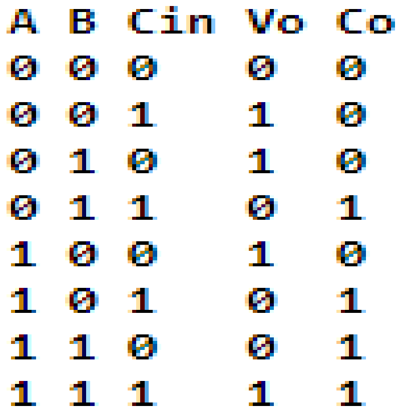
```
help?
```

Progol is terminated by depressing the Control and D keys together (^D).

Propositional Logic - A 1-bit Adder

A 1-bit adder circuit looks like this:





A	B	Cin	Vo	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

truth-table specification for this adder

The 1-bit adder can be represented as definite clauses in propositional logic. For example, the following propositions can be used to denote the input:
1) a, b, c_in: true if A or B or Cin are true, respectively; and
2) not_a, not_b, not_c_in: true if A or B or Cin are false, respectively.

Propositions used to denote output are:

1) v_o, c_o: true if Vo or Co are true, respectively; and
2) not_v_o, not_c_o: true if Vo or Co are false, respectively; and

The following definite-clauses encode the truth table for the adder:

```
v_o:-
    not_a, not_b, c_in.
v_o:-
    not_a, b, not_c_in.
v_o:-
    a, not_b, not_c_in.
v_o:-
    a, b, c_in.

not_v_o:-
    not_a, not_b, not_c_in.
not_v_o:-
    not_a, b, c_in.
not_v_o:-
    a, not_b, c_in.
not_v_o:-
    a, b, not_c_in.

c_o:-
    not_a, b, c_in.
c_o:-
    a, not_b, c_in.
c_o:-
    a, b, not_c_in.
c_o:-
    a, b, c_in.

not_c_o:-
    not_a, not_b, not_c_in.
not_c_o:-
    not_a, not_b, c_in.
not_c_o:-
    not_a, b, not_c_in.
not_c_o:-
    a, not_b, not_c_in.
```

Exercise 3
(a) Type these clauses into a file, say adder.pl, using your favorite editor.
(b) Also include definitions for the current state of the inputs A, B and Cin when switch A is "on", and B, Cin are "off".
(c) Start Progol. At the prompt, consult your definitions in with the command:

```
consult(adder)?
```

To check if Progol has the correct definitions, first try the command:

```
listing?
```

Progol's response should be something like:

```
The following user predicates are defined:
 a      c_o     not_b   not_c_in   not_c_o   not_v_o   v_o
[Total number of clauses = 19]
yes
[:- listing? - Time taken 0.00s]
```

You can check individual clause definitions by the "listing" command. For example:

```
listing(not_c_o)?
```

will list the definitions for the proposition not_c_o.

You are now able to investigate the consequences of the switch settings. This is done by examining whether which of the "output" propositions v_o, not_v_o, c_o, not_c_o are logical consequences of the set of definite-clauses in adder.pl and inputs.pl.

Progol can be asked if v_o is a logical consequence by typing the following at the prompt:

```
v_o?
```

to which should reply

```
yes
```

Exercise 4

(a) Verify (using the truth-table) that Progol gives correct answers for the logical status of each the four output propositions.

(b) Can you simplify any of the definitions further. That is, do you think the adder circuit could be represented with either fewer, or simpler clauses?

Negation

Let us return to the 1-bit adder. Recall that separate propositions were needed for both an output and its negation (for example, v_o and not_v_o). Why did we have to be so clumsy? If v_o is a logical consequence of the clauses, then should we not be able to state confidently that not_v_o cannot possibly be a consequence as well? The answer is yes, provided one assumption holds. This is that the definitions we use are not just correct, but complete. In other words, we have not left out any condition(s) which are true. This assumption is called the closed world assumption or CWA.

Once you have complete and correct definitions for v_o and c_o, you can dispense with the definitions for not_v_o and not_c_o. Instead, you can use a predicate called not.
The complete set of definitions for the adder is now as follows:

```
v_o:-
    not(a), not(b), c_in.
v_o:-
    not(a), b, not(c_in).
v_o:-
    a, not(b), not(c_in).
v_o:-
    a, b, c_in.

c_o:-
    not(a), b, c_in.
c_o:-
    a, not(b), c_in.
c_o:-
    a, b, not(c_in).
c_o:-
    a, b, c_in.
```

The new symbol not is a special metalogical predicate built-in to Progol, that implements negation under the closed world assumption. Any of the propositions a, b, or c_in that are known to be true must be in this file. If any proposition, say c_in is not in the file, then, by CWA, not_c_in is true.

Exercise 5

Consult your new (smaller) set of definitions. You can now examine if not_v_o or not_c_o are logical consequences of the definitions of your definitions by asking:

not(v_o)? or not(c_o)?

Work through the truth table again, verifying the consequences of your adder definitions and truth value assignments to a, b, and c_in.

Sets
One way to represent sets is with monadic predicates. The set S = {a, b, c,...} is represented by the predicates s(a), s(b), etc. Assume for the moment, that there are only 2 sets of interest in the world, whose elements are represented by the monadic predicates p/1 and q/1.

Note: You will keep coming up against the notation Name/Number (like p/1). By this, it will be understood that we are talking about a predicate with a name, and with some number of arguments.

Set operations with this monadic representation are easy. Consider the union operator.

Union of P and Q is a set p_union_q = {e| e in P or e in Q}

Or, as definite-clauses:
```
p_union_q(X):-
    p(X).
p_union_q(X):-
    q(X).
```

Instead of having a new unary predicate for each set, it is possible to represent sets with predicates that have the name of the set as first argument. For example, the sets P= {a, b} and Q = {1,2} can both be represented with a binary predicate elem/2, as elem(a, p), elem(b, p), elem(1,q), and elem(2,q).

The statement:
x in S1 U S2 iff x in S1 or x in S2

can now be represented by the following definite clauses:

```
in_union(X,S1,S2):-
    elem(X,S1).
in_union(X,S1,S2):-
    elem(X,S2).
```

Exercise 6
Write definite clauses for the following statements:

x in S1 & S2 iff x in S1 and x in S2
<x,y> in S1 x S2 iff x in S1 and y in S2
x in S1 \ S2 iff x in S1 and x not in S2

Numbers
Let N5 = {0,1,2,3,4}, the set of the first 5 natural numbers. The successor function that associates a natural number with another one, namely the next one can be represented as a set of ordered pairs. For the set N5: successor5 = {<x,y> | x in N5, y in N5, and y = x + 1}

Later we will look at representing functions like addition (+). For the moment, the ordered pairs in successor5 can be represented by the following "extensional" definition:

```
successor5(0,1).
successor5(1,2).
```

Exercise 7

(a) Write a similar extensional definition for the less_than5 relation, that describes all ordered pairs (x,y) such that the first element in the pair is less than (lt) the second.

(b) Given sets A, B with elements from N5 write a definition for the binary relation:
lt_AxB = {<x,y> | <x,y> in AxB, and <x,y> in less_than5}

Enter the following definitions:

```
bachelor(X):- not(married(X)), male(X).
married(john).
male(john).
male(bill).
```

Exercise 8

(a) Does the following query succeed correctly:
bachelor(X)

(b) What about:
bachelor(bill)
and
bachelor(john)

(c) What happens if the definition of bachelor was written as:
bachelor(X):- male(X), not(married(X)).

Simple recursion

Suppose the edges in a graph can be represented by the predicate g/2. In particular, consider this graph:

```
g(1,2).
g(1,3).
g(2,4).
g(2,5).
```

```
g(3,4).
g(4,7).
g(5,6).
g(6,7).
```

The transitive closure of a directed graph has an edge wherever the graph has a path. These are all definitions of transitive closure:

Definition 1:

```
        tc(X,Y):- g(X,Y).
        tc(X,Y):- g(X,Z), tc(Z,Y).
```

Definition 2:

```
        tc(X,Y):- g(X,Y).
        tc(X,Y):- tc(X,Z), g(Z,Y).
```

Definition 3:

```
        tc(X,Y):- tc(X,Z), g(Z,Y).
        tc(X,Y):- g(X,Y).
```

Exercise 9

(a) Which of the following are true for each of the preceding definitions:

tc(3,6)

tc(3,7)

(b) For which values of X (and Y) are the following true:

tc(X,5)

tc(5,X)

tc(X,Y)

(c) For the following definition of a graph:

```
g(1,2).
g(1,3).
g(2,3).
g(3,4).
g(4,1).
```

What are the first 10 answers returned by Progol for

tc(1,X)

using Definition 1 for transitive closure?

Note: You can force Progol to enumerate all X's for which tc(1,X) is true by typing a semi-colon (";") after each answer returned.

Elementary arithmetic

In this exercise, you will see how you can execute the definitions of simple arithmetic functions. It is possible to treat numbers as any other constant, and functions and predicates on them just like any other predicates. But the naming scheme of Zero = the constant symbol 0, One = s(0), Two = s(s(0)) is cumbersome. It would be much better to name them using the standard Indo-Arabic conventions. Progol allows this.

Next, what do we do with predicates for comparing numbers. Do we declare them explicitly each time, for the numbers that we want, or do we "pre-load" these into Progol. If so, how many entries do we consider pre-loading (comparisons of all integers upto 2^32 perhaps?)

For all practical purposes, you may assume that Progol has access to a (nearly) infinite table of such facts. How it does it is not really our concern here. Sufficient to say that, given a pair of numbers, Progol "looks-up" its internal table, and returns the truth-value of the comparison.
Progol also has the built-in functions +,-,* and /.

There is one more predicate that you may not have spotted. is/2 will be used often when dealing with numbers in Progol. This predicate is used to evaluate arithmetic expressions.

Exercise 10

(a) To understand is/2, try entering the following at the Progol prompt:

```
X = 2 + 3?
```

(b) Now try:

```
X is 2 + 3?
```

This should immediately give you an idea of what is/2 does (and for that matter, what =/2 doesn't do when dealing with arithmetic expressions).
So, the moral is as follows: if you want an arithmetic expression to be evaluated, then you MUST use is/2.

ILP with Progol

To be able to construct clausal definitions for some target predicate (like mem/2) most ILP systems require the following:

1. Language constraints:
   control the sort of definitions that are constructed for the target predicate.
2. Background knowledge:
   definitions of other relations that may be useful in constructing the definition of the target predicate.
3. Positive and negative examples of the target predicate:
   act as a partial specification of the relation specified by the target predicate.

For Progol these three bits of information are usually provided in a single file. As always, the file's name should have a ".pl" suffix.

Exercise 11

Consider a simple problem of learning to classify an animal based on the following observable attributes:

What covering does the animal have?
Does it have milk?
Is it warm blooded?
Where does it live?
Does it lay eggs?
Does it have gills?

Progol statements that specify this are in the file:
~cs731/public/Examples/animals.pl
(a) Link or copy this file to your directory.
(b) Examine this file and note that:
1. The language constraints look like:
:- some_name(.....)?
2. The background knowledge is a set of clauses
3. Positive examples look like:
target_predicate_name(....).
4. Negative examples look like:
:- target_predicate_name(....).

Introduction to Modes

Consider Progol's task when acting as an ILP system. The job is to construct a set of clauses to act as a definition for the target predicate. All positive examples provided must be derivable from this set of clauses, and the set must be consistent with the negative examples. Later, we will look at the logical constraints that such a clause set must satisfy. For now, consider what each clause constructed by Progol would look like. It would typically be a definite clause like:

```
target_predicate(.....):-
    body_literal_1(....),
    body_literal_2(....),.... .
```

Which predicate symbols would we like to appear in the head or body of the clause? Do we know anything about the argument types of such literals (for example, are they always natural numbers)? Do we require some arguments for a literal to be bound before it can appear in the clause (for example, recall the constraint on not/1)?
In effect, by stating such preferences, we are really defining a language for clauses to be constructed by Progol. The primary means of doing this is via mode declarations.

There are two sorts of mode declarations. Those that look like:

```
:- modeh(RecallNumber,Template)?
```

and those that look like:

```
:- modeb(RecallNumber,Template)?
```

Ignore the RecallNumber for the moment. Using a template which we will look at shortly), the first of these (the modeh one) describes legal forms of literals that can appear in the head of a Progol clause. The second type of mode declaration (the modeb one) describes legal forms of literals that can appear in the body of a Progol clause.

Now look at the modeh declaration in animals.pl.

```
:- modeh(1,class(+animal,#class))?
```

In this the template provided for a head literal is:

```
class(+animal,#class)
```

What this says is that any clause that Progol constructs should be of the form:

```
class( , ):- Body.
```

In fact, it does more than just this. It also says that the first argument is variable of type animal and the second is constant of type class (more on this in "Argument Annotations" below).

So, clauses constructed will look like:

```
class(A, mammal):- ....
class(A, fish):- ....
  (etc.)
```

Exercise 12

Which predicate symbols can appear in the body of clauses for class/2?

Types

Progol checks the types of terms by using unary predicate definitions which are provided as background knowledge. That is, if Progol wanted to check if some constant, say mammal was of type class it checks to see if the call:

```
class(mammal)?
```

Succeeds

Exercise 13

Is the following a legal type definition. Give reasons for your answer.

```
class(X):- member(X,[mammal,fish,reptile,bird]).
```

Argument Annotations
Notice that in the modes arguments are annotated by the symbols + and # . In fact, there are 3 such syntactic constraints on the arguments: +, -, and #. To understand these, you will need to realize that Progol constructs its clauses literal by literal, starting from the head literal. Clauses are constructed using the following rules:

1. A clause is of the form H:- B1, B2, ..., Bc.

2. Variables will appear in any arguments of H, Bi with + or - annotations. Constants will appear in any arguments annotated by #. By convention all + variables will be called input variables and all - variables will be called output variables.

3. Any input variable of type T in a body literal Bi (1 =< i =< c) must appear as an output variable of type T in a body literal Bj (1 =< j < i) or as an input variable of type T in H.

4. Any output variable of type T in H must appear as an output variable of type T in a Bi (1 =< i).

Exercise 14

Consider the mode declarations:

```
:- modeh(1,class(+animal,#class))?
:- modeb(1,has_milk(+animal))?
```

Are the following clauses allowed? Give reasons for your answer.
(a) class(A,B):- has_milk(B).
(b) class(A,B):- has_milk(A).
(c) class(A, mammal):- has_milk(platypus).
(d) class(platypus, mammal):- has_milk(platypus).
(e) class(A, mammal):- has_milk(A).

Recall Number
Finally, a note on the RecallNumber. This concerns the "determinacy" of the predicate with the template provided. In other words, it specifies the maximum number of times a call to the predicate would succeed, with a given set of input variables. This need not worry you too much. Usually, you will only be concerned with predicates with a recall number of either:

1: for a given set of input variables the output variables and constants are determinate. That is, a call to the predicate, with any set of bindings for these input variables, succeeds at most once
*: for a given set of input variables the output variables and constants are not determinate. That is, a call to the predicate, with any set of bindings for these input variables, succeeds some finite number of times.

Exercise 15

Why is the recall number for habitat/2 a * ?

Exercise 16

Given the set of natural numbers N, let

dec(X,Y): X in N and Y = X - 1
plus(X,Y,Z): X,Y in N and Z = X + Y
mult(X,Y,Z): X,Y in N and Z = X*Y

You may (recall this definition of mult/3):

```
mult(0,A,0).
mult(A,B,C):-
    dec(A,D),
    mult(D,B,E).
    plus(E,B,C).
```

Assume that you had some predicate nat/1 that could be used to check if a variable was in N. Also assume that you were provided definitions for dec/2 and plus/3.

(a) Specify language constraints in the form of mode-declarations that allow the construction of this definition of mult/3.

Suppose Progol had constructed the following definition for choosing "m" elements from "n" (m,n are natural numbers >=0):

```
n_choose_m(A,B,C):-
    dec(B,D),
    dec(A,E),
    n_choose_m(E,D,F),
    multiply(F,A,G),
    divide(G,B,C).
```

(b) What mode declarations could have allowed the construction of such a definition?

Instructions on how to use Progol?

- Download Progol using following link: https://www.doc.ic.ac.uk/~shm/progol.html
- You will need to use Linux/Mac to run Progol on your machine. If you already have it, that's good, if not, you can use UTD's public Linux server. For that you will have to connect to **csgrads1.utdallas.edu** using putty (for windows) with your NetID and password. You can use WinSCP to transfer your files to your account on Linux server.
- Once you are connected to csgrads1.utdallas.edu, follow readme file from above link. You will have to change your .login file to include path name for source directory of Progol so that you can use "progol" as a command in Linux.
- When you open .login file it shows you where and how to add your source directory path.
- Once you have edited your .login file, log out from putty and log in again to reflect the changes you made to the file and now you are ready to use progol.
- To use any .pl file, type the command "progol <File_Name>" at the terminal. After that enter into progol prompt and now you can consult your file.