Very Short Introduction to Prolog

1. Introduction

Let me start by explaining how to write logical statements in Prolog. Take the sentence "Kathmandu is the capital of Nepal". The way we write this fact in Prolog is very similar to how we write statements in first order predicate logic (FOPL). What we have in this sentence is a relationship between two objects, capital. The prolog statement that represents this fact is:

```
capital(kathmandu,nepal).
```

There are some conventions to follow in Prolog.

- Objects name starts with a lowercase letter. Here, kathmandu and nepal are the objects. Alternatively, we can write them inside single quotes if we want to use uppercase.
  `capital('Kathmandu','Nepal').`
- Predicate name must start with a lowercase letter. We can use underscore but cannot include space to name predicates. *capital_of and capitalOf* are correct whereas *capital of* and *Capital_of* are incorrect.
- Numbers are not of much interest in Prolog. Prolog is not meant for numerical calculations. We will see integers being used in lists (similar to arrays).
- A variable name must start with an uppercase letter or underscore. *Subject* and *_subject* are correct variable names whereas *subject* is incorrect.
- All prolog statement end will a full-stop.

Not let us get back to our first program, World Capitals.

Open Notepad or your favourite text editor and type the lines given below. Save the file with 'pl' extension e.g. 'Capitals.pl'. Create a directory/folder in one of your drives where you will save the codes for this course. I will save the file inside 'G:\prolog'.

```
capital('Kathmandu','Nepal').
capital('New Delhi','India').
capital('Paris','France').
capital('Madrid','Spain').
```

Now open the Prolog application. You will see the Prolog prompt `?-`

In order to run your code, first you need to load the file and compile it.

```
?- consult('G:/prolog/capitals.pl').
```

If you see the output "true.", that means you have an error-free code.

If you have errors, you will have to edit it. You have two choices. You can edit it in your favourite editor and then consult again. You can also edit the code inside Prolog.

```
?- edit('G:/prolog/capitals.pl').
```

Once you are done with the editing, save your file by going to File>Save Buffer. Then close the editor. You will have to call consult again to compile your now edited code.

```
?- consult('G:/prolog/capitals.pl').
```

As you may see, it gets tiresome to type in the whole path all the time. We can get around it by changing the present working directory. You can see what directory it is using by typing

```
?- pwd.
```

To change the pwd we use the predicate working_directory/2

```
?- working_directory(_,'G:/prolog').
```

The first argument is an underscore character. The second is the path where you store your Prolog codes. Now you can simple use the filename in the consult and edit predicates, `consult('capitals.pl').` and `edit('capitals.pl').`

Now you can ask "queries" over the knowledge base.

1. What is the capital of Nepal?
   ?- `capital(X,'Nepal').` [In Prolog we call this a query/question]
2. What country is Paris the capital of?
   ?- `capital('Paris',Y).`
3. What are all the capital cities in my KB?
   `capital(X,_).` [Press enter and then keep on pressing semicolon to get all the answers]
4. What are all the countries in my KB?
   `capital(_,Y).` [Press enter and then keep on pressing semicolon to get all the answers]

So far there are only facts in our code. Prolog allows us to create rules (if-then statements) to do useful things with the knowledge base. Suppose we want to add a new knowledge that "All capital cities are crowded". We can add a rule on prolog that encodes this knowledge. Add the following line at the end of your code Capitals.pl.

```
crowded(X):- capital(X,_).
```

Here the :- (colon followed by a dash) is read as "if". This rule states "if X is a capital of some country then it is crowded"

Now, we can ask the following questions.

5. Is Kathmandu crowded?
6. `?- crowded('Kathmandu').`
   The answer should be True because Kathmandu is a capital city.
7. Is France crowded?
   `?- crowded('France').`
   The answer should be False because France is not a capital city.
8. What are all the crowded cities?
   `?- crowded(Z).` [Press enter and then keep on pressing semicolon to get all the answers]

2. Simple Practice Queries

Task A:

1. Add these new facts to your code.
    - Nepal is in Asia.
    - India is in Asia.
    - France is in Europe.
    - Spain is in Europe.
2. Write a query that returns the continent Nepal belongs.
3. Write a query that returns all the countries in Asia.
4. Add a rule (predicate) that takes two countries as arguments and return True if both are located in the same continent and False otherwise. [Think about how you would approach it. I have yet not shown an example of how to write such a rule.]

Conjunction and Disjunction

```
wealthy(dhiraj).
healthy (dhiraj).

happy(Person):- wealthy(Person), healthy(Person).
```

The rule for happy states that "if a person is wealthy AND healthy then s/he is happy". Comma is the conjunction connective in Prolog.

If we were to add another condition for being happy which states "a person is happy if that person is wise", we have two ways to write that in Prolog.

Option 1:

```
happy(Person):- wealthy(Person), healthy(Person) ; wise(Person).
```

Here, semi-colon is the OR connective. Since AND has precedence over OR, it connects wealthy and healthy predicates.

Option 2:

We can instead write separate rules for the two alternative conditions of happiness.

```
happy(Person):- wealthy(Person), healthy(Person).
happy(Person):-wise(Person).
```

I prefer the second style of writing OR definitions/conditions. It is easier to read and less prone to mistakes.

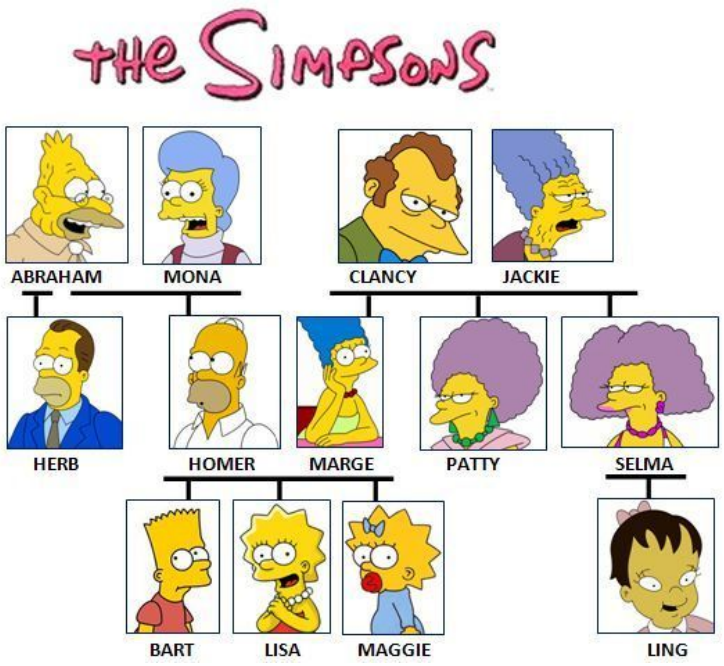
Task B:

1. Write rule for the predicate happy which is defined as following. A person is happy if the person is healthy and either rich or famous.
2. Translate the sentence to Prolog rule "If Santosh likes tennis Santosh likes football."
3. Suppose we have the facts:
```
likes(sunita, momo).
likes(sunita, pizza).
likes(prakriti, momo).
likes(prakriti, sunita).
```

What does the following queries return: (a) `?- likes(prakriti, X).` (b) `?- likes(_,momo).` (c) `?- likes(sunita, X), likes(prakriti, X).`

2. Simpson's Family Tree

We will now look at a more complete application to better understand how Prolog works and what we can do in it. We will look at Simpson's family tree.



Tasks

1. Add facts that represents the parent/child relationship shown in the family tree above. For example, Homer and Marge are the parents of Lisa, Bart and Marge. To do that, define a predicate `parent` that takes two argument (arity 2). The first argument is the parent and the second the child.
2. Define rules to encode the following relationships. Use the `parent` predicate you created in 1 to define these rules. you will soon see that you need more facts than parent to represent these rules. Add the facts that you find necessary to define these rules. [*Hint: you will require to add facts about gender.*]
   (a) `father`, (b) `mother`, (c) `brother`, (d) `sister`, (e) `son`, (f) `daughter`, (g) `aunt`, (h) `uncle`, (i) `grandfather`, (j) `grandmother`, (k) `grandparent`
3. Define a rule for `sibling` that returns true if two people have the same parents.
4. Define a rule for `ancestor` relationship using the predicate `parent` that returns true if one of them is the ancestor of another. [*Hint: You have to think of the relationship as a recursive definition.*]

Exercises in Prolog Programming

1. Recursion in Prolog

Let us start with the familiar recursive definition for calculating factorial that we all should be familiar with. The recursive formulation is:

factorial(0) = 1
factorial(n) = n x factorial(n-1) [for n>0]

C implementation would look something like:

```
/* Factorial in C */

int factorial(int n) {

    if(n==0)
      return(1);

    return(n*factorial(n-1)); /*This is the recursion part */

}
```

A Prolog implementation of the same idea looks like this:

```
%Factorial in Prolog

factorial(0,1).
factorial(X,Y) :- X1 is X-1, factorial(X1,Y1), Y is X * Y1.
```

Can you see the similarities? In every recursive formulation there is a base condition (i.e. solution for the simplest case). In case of factorial it is when n=0. Once we know how to calculate factorial of zero we can calculate factorial of 1 which is simply 1 x factorial(0). Once we know this we can calculate factorial of 2 which is 2 x factorial(1) x factorial(0). And so on.

Notice that in Prolog we use the keyword "is" instead of "=" in arithmetic expressions. So, X1 is X-1 is just X1 = X-1.

Be comfortable with writing basic predicates and recursive definitions to answer the questions below.

2. Questions

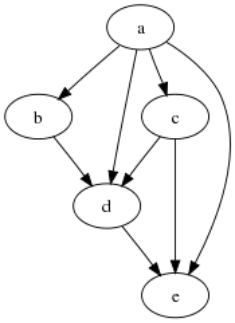
a. The predicate student is defined as `person(firstname,lastname,gender,programme).` The gender term is either male or female. The programme term is either BEIT or BEComputer. Fill in the information for 10 students.
   - What query will you give to find all students whose name is Suraj? (Note: the statement you write at the `?-` prompt is called the query /goal)
   - Write the query to find all the first name and last name of students in BEIT.
   - Find all the first names that are common to BEIT and BEComputer.

b. Write a rule for the predicate `greaterthan` which prints (write predicate) the greater of the two numbers.

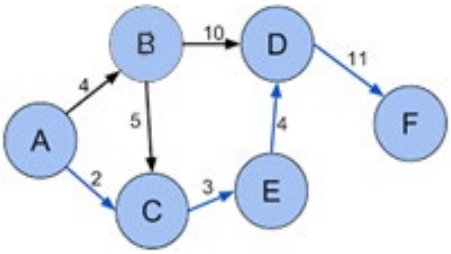
c. Two people are relatives if (a) one is the ancestor of the other, or (b) they have a common ancestor, or (c) they have a common successor. Define the rule `relatives(X,Y)` which is True if X and Y are related.

d. Modify the predicate `ancestor` you defined in the Simpson's family program to count the number of generations. For example, the count of generations for Homer and Bart is 1, because there are in a direct parent relationship. The count for Abraham and Bart is 2, because Abraham is Bart's parent's parent. [*Note: Use same idea that we used for factorial calculation.*]

e. Write a program to represent the directed graph shown below in prolog. Define the predicate `edge(X,Y)` to store the facts.
   - Use edge predicate to define a rule `connected(X,Y)` which is True if there is an edge from X to Y or Y to X.
   - Use edge predicate to define a rule `path(X,Y)` which is True if there is path from node X to node Y. For example, `path(a,c)` is True, `path(a,e)` is True, `path(d,a)` is False and so on.



f. Write a program to represent the weighted directed graph given below. The edge predicate will have an additional argument weight. For example, the edge from A to B has weight 4. This can be represented by the predicate `edge(X,Y,W)`. For this example, the predicate will be `edge(a,b,4).`



Modify the predicate `path` which you defined for the earlier question so that it is True if there is path from source node to the target node. This predicate should also calculate the cost of the path. For example, the path from A to F is {A,C,E,D,F} and has the cost 2+3+4+11 = 20. [*Note: Use same idea that we used for factorial calculation.*]