Lab 14

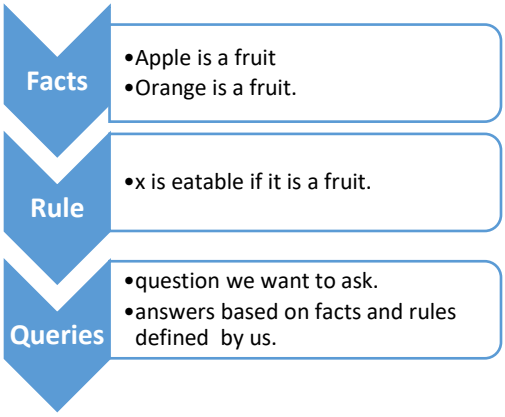
Prolog

Prolog stands for Programming in logic. It is used in artificial intelligence programming. Prolog is a declarative programming language.

For example: While implementing the solution for a given problem, instead of specifying the ways to achieve a certain goal in a specific situation, user needs to specify about the situation (rules and facts) and the goal (query). After these stages, Prolog interpreter derives the solution.

Prolog is useful in AI, NLP, databases but useless in other areas such as graphics or numerical algorithms.

Knowledge Base:



Domain:

A domain refers to the set of possible values that a variable can take. It defines the range or type of values that can be assigned to a variable. Specifying the domain helps constrain the possible solutions during the execution of Prolog programs. This is often done through the use of facts and rules that define relationships and constraints within the specified domain.

For example, if you are working on a Prolog program that deals with family relationships, you might have a domain for individuals that includes different people like parents, children, grandparents, etc. Each variable representing a person would have a domain that restricts it to possible values within the set of individuals.

Predicates:

Predicates are fundamental components that express relationships, conditions, or actions. Predicates consist of a predicate name and a certain number of arguments. Here are some examples to illustrate the concept of predicates in Prolog:

Basic Predicate

```
% Predicate definition
likes(john, pizza).
% John likes pizza
likes(jane, chocolate).
% Jane likes chocolate
```

Arithmetic Predicates

```
% Arithmetic predicates
sum(X, Y, Result) :- Result is X + Y.
```

The sum/3 predicate takes two arguments and calculates their sum.

Predicate with Rules

```
% Rule defining a predicate
 father(X, Y) :- male(X), parent(X, Y).
 % Facts
 male(john).
 parent(john, mary).
```

Clause

A clause is a fundamental building block of logic rules and facts. Clauses are used to define relationships, conditions, and actions in a Prolog program. There are two main types of clauses in Prolog: facts and rules.

Symbols

Prolog expressions are comprised of the following truth-functional symbols, which have the same interpretation as in the predicate calculus.

English	PROLOG
and	,
or	;
if	:-
not	not

Variables and Names

- Variables begin with an uppercase letter.
- Predicate names, function names, and the names for objects must begin with a lowercase letter. Constants: begin with lower-case letter or enclosed in single quotes.
- Rules for forming names are the same as for the predicate calculus.

```
mother_of
male
female
greater_than
Socrates
```

Facts

A fact is a simple clause that states a piece of information or a relationship without any conditions or variables.

- Facts are often used to declare base knowledge in a Prolog program.
- They are written in the form of predicate(term1, term2, ..., termN).

Example:

```
likes(john, susie).                    /* John likes Susie */
likes(X, susie).                       /* Everyone likes Susie */
likes(john, Y).                        /* John likes everybody */
likes(john, Y), likes(Y, john).        /* John likes everybody and everybody
likes John */
likes(john, susie); likes(john,mary). /* John likes Susie or John likes Mary
*/
not(likes(john,pizza)).                /* John does not like pizza */
likes(john,susie) :- likes(john,mary)./* John likes Susie if John likes
Mary.
```

Rules

A **rule** is a predicate expression that uses logical implication (:-) to describe a relationship among facts. Thus a Prolog rule takes the form

left_hand_side :- right_hand_side .

This sentence is interpreted as: *left_hand_side* ***if*** *right_hand_side*.

The **left_hand_side** is restricted to a **single, positive, literal**, which means it must consist of a positive atomic expression. It cannot be negated and it cannot contain logical connectives.

This notation is known as a **Horn clause**. In Horn clause logic, the left hand side of the clause is the conclusion, and must be a single positive literal. The right hand side contains the premises. The Horn clause calculus is equivalent to the first-order predicate calculus.

Examples of valid rules:

```
friends(X,Y) :- likes(X,Y),likes(Y,X).            /* X and Y are friends if
they like each other */
hates(X,Y) :- not(likes(X,Y)).                    /* X hates Y if X does not
like Y. */
enemies(X,Y) :- not(likes(X,Y)),not(likes(Y,X)).  /* X and Y are enemies if
they don't like each other */
```

Examples of invalid rules:

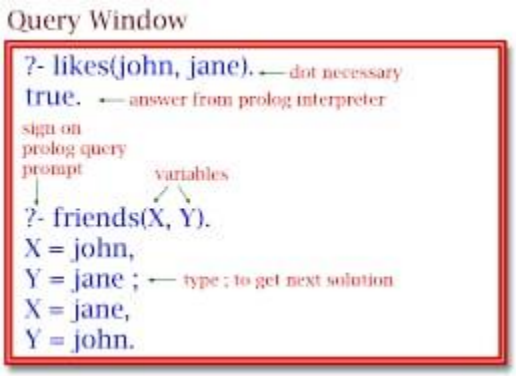
```
left_of(X,Y) :- right_of(Y,X)                     /* Missing a period */
likes(X,Y),likes(Y,X) :- friends(X,Y).            /* LHS is not a single
literal */

not(likes(X,Y)) :- hates(X,Y).                    /* LHS cannot be negated */
```

Queries

The Prolog interpreter responds to **queries** about the facts and rules represented in its database. The database is assumed to represent what is true about a particular problem domain. In making a query you are asking Prolog whether it can prove that your query is true. If so, it answers "yes" and displays any **variable bindings** that it made in coming up with the answer. If it fails to prove the query true, it answers "No".

Whenever you run the Prolog interpreter, it will **prompt** you with **?-**. For example, suppose our database consists of the following facts about a fictitious family.




```
father_of(joe,paul).
father_of(joe,mary).
mother_of(jane,paul).
mother_of(jane,mary).
male(paul).
male(joe).
female(mary).
female(jane).
```

We get the following results when we make queries about this database. (I've added comments, enclosed in /*..*/, to interpret each query.)

```
mother_of(jane, paul).
mother_of(jane, mary).

male(paul).
male(joe).

female(mary).
female(jane).

father_of(joe, paul).
father_of(joe, mary).

(10 ms) yes
| ?- father_of(joe,paul).

true ?

yes
| ?- father_of(paul,mary).

no
| ?- father_of(X,mary).

X = joe

yes
```

Prolog's Proof Procedure

In responding to queries, the Prolog interpreter uses a **backtracking** search. To see how this works, let's add the following rules to our database:

```
 parent_of(X,Y) :- father_of(X,Y).        /* Rule #1 */
 parent_of(X,Y) :- mother_of(X,Y).        /* Rule #2 */
```

And let's trace how PROLOG would process the query. Suppose the facts and rules of this database are arranged in the order in which they were input. This trace assumes you know how unification works.

```
   ?- parent_of(jane,mary).
  parent_of(jane,mary)      /* Prolog starts here and searches
                               for a matching fact or rule. */
    parent_of(X,Y)          /* Prolog unifies the query with the rule #1
                               using {jane/X, mary/Y}, giving
                               parent_of(jane,mary) :- father_of(jane,mary)
*/
    father_of(jane,mary)    /* Prolog replaces LHS with RHS and searches. */
                            /* This fails to match father_of(joe,paul) and
                               and father_of(joe,mary), so this FAILS. */
                            /* Prolog BACKTRACKS to the other rule #2 and
                               unifies with {jane/X, mary/Y}, so it matches
                               parent_of(jane,mary) :- mother_of(jane,mary)*/

    mother_of(jane,mary)    /* Prolog replaces LHS with RHS and searches. */

    YES.                    /* Prolog finds a match with a literal and so
succeeds.
```

Here's a trace of this query using Prolog's **trace** predicate:

```
| ?- trace,parent_of(jane,mary).
{The debugger will first creep -- showing everything (trace)}
   1  1  Call: parent_of(jane,mary) ?
   2  2  Call: father_of(jane,mary) ?
   2  2  Fail: father_of(jane,mary) ?
   2  2  Call: mother_of(jane,mary) ?
   2  2  Exit: mother_of(jane,mary) ?
   1  1  Exit: parent_of(jane,mary) ?

yes
{trace}
| ?-
```

Exercises

```
father_of(joe,paul).
father_of(joe,mary).
father_of(joe,hope).
mother_of(jane,paul).
mother_of(jane,mary).
mother_of(jane,hope).

male(paul).
male(joe).
male(ralph).
male(X) :- father_of(X,Y).

female(mary).
female(jane).
female(hope).

son_of(X,Y) :- father_of(Y,X),male(X).
son_of(X,Y) :- mother_of(Y,X),male(X).

daughter_of(X,Y) :- father_of(Y,X),female(X).
daughter_of(X,Y) :- mother_of(Y,X),female(X).

sibling_of(X,Y) :- !,father_of(Z,X),father_of(Z,Y),X\=Y.
sibling_of(X,Y) :- !,mother_of(Z,X),mother_of(Z,Y),X\=Y.
```

1. Add a `male()` rule that includes all fathers as males.
2. Add a `female()` rule that includes all mothers as females.
3. Add the following rules to the family database:

```
son_of(X,Y)
daughter_of(X,Y)
sibling_of(X,Y)
brother_of(X,Y)
sister_of(X,Y)
```

4. Given the addition of the **`sibling_of`** rule, and assuming the above order for the facts and rules, show the PROLOG trace for the query **`sibling_of(paul,mary).`**