UNIT 2 A. I. LANGUAGES-2: PROLOG

Structure **Page Nos.**

2.0 INTRODUCTION

We mentioned in the previous unit that there are different *styles* of problem solving with the help of a computer. For a given type of application, some style is more appropriate than others. Further, for each style, some programming languages have been developed to support it. In this context, we have already discussed two styles of solving problems viz *imperative* style and *functional* style. Imperative style is supported by a number of languages including C and FORTRAN. Functional style is supported by, among others, the language LISP. The language LISP is more appropriate for A. I. applications.

There is another style viz *declarative* style of problem solving. A declarative style is *non-procedural* in the sense that a program written according to this style does not state exactly *how* the computational process is to be carried out. *Rather,* a *program consists of mainly a number of declarations representing relevant facts and rules concerning the problem domain*. *The solution to be discovered is also expressed as a question to be answered or, to be more precise, a goal to be achieved. This question/goal also forms a part of the PROLOG program that is intended to solve the problem under consideration*. The main technique, in this style, based on resolution method suggested by Robinson (1965), is that of **matching** goals (*to be discussed*) with facts and rules. The matching process generates new facts and goals. The matching process is repeated for the whole set of goals, facts and rules, including the newly generated ones. The process, terminates when either all the initial goals, alongwith new goals generated later, are satisfied or when it may be judged or proved that the goals in the original question are not satisfiable.

Logic programming is a special type of *declarative* style of programming, in which the various program elements and constructs are expressed *in the notations similar to that of predicate logic*. Two interesting features of logic programs are *non-determinism* and *backtracking*. A *non-deterministic* program may find a number of solutions, rather than just one, to a given problem. *Backtracking* mechanism allows exploration of potential alternative directions for solutions, when some direction, currently being investigated, fails to find an appropriate solution. The language

PROLOG, which sports Logic Programming style, has been found useful in A.I. applications. In this unit, we have introduce the programming language PROLOG,

2.1 OBJECTIVES

After going through this unit, you should be able to:

- explain the *declarative style* and *logic programming style* of problem solving, which form the basis of PROLOG programming;
- explain the difference between *declarative style* and *imperative style*, where the later forms the basis of conventional programming languages like, FORTRAN, Pascal, C etc.
- tell us the rules of syntax of PROLOG;
- to write PROLOG programs, using the syntax of the language;
- explain how PROLOG system solves a problem;
- explain the concept of *backtracking* and its use in solving problems;
- enumerate the various data types and data structures available in PROLOG, and further be able to use the available types and structures in defining the data for a program;
- enumerate and apply various PROLOG operations on lists;
- enumerate and use predicates in expressions, and
- explain the significance of the operator *Cut* and operator *Fail* and further should be able to use the operators 'cut' and 'fail' in programs to solve problems.

2.2 FOUNDATIONS OF PROLOG

The programming language PROLOG derives its name from ***PRO****gramming in* ***LOG****ic*

.

The fundamentals of PROLOG were developed in the early 1970's by Alain Colmerauer and Phillipe Roussel of the Artificial Intelligence group at the University of Marseille together with Robert Kowalski of Department of Artificial Intelligence at University of Edinbury, U.K. There are a number of dialects of PROLOG. *Two most important characteristics of PROLOG are as given below:*

(i) The facts and rules are represented using a syntax similar to that of predicate logic. These facts and rules constitute what is called PROLOG database/knowledge base.

(ii) The process of problem solving through PROLOG is carried out mainly using an in-built inferencing mechanism based on Robinson's resolutions method. Through the inferencing mechanism, in the process of meeting the initially given goal(s), new facts and goals are generated which also become part of the PROLOG database.

The syntax of PROLOG is based on *Horn Clause*, a special type of clause in predicate logic. We know that a ***Clause*** is a predicate logic expression of the form $p_1 \vee p_2 \ldots\ldots \vee p_i \vee \ldots\ldots p_r$, where each p_i is a literal, and '$\vee$' denotes disjunction. Further a ***literal*** q is of the form p or ~ p (negation of p) where p is an *atomic* symbol. A ***Horn Clause*** is a clause that has at most one positive literal, i.e., an atom.

There is some minor difference between the predicate logic notation and notation used in PROLOG. The formula of predicate logic $P \wedge Q \wedge R \rightarrow S$ is written as

S: - P, Q, R.

(note the full stop following the last symbol R)

Where the symbol obtained from writing ':' (colon) followed by '-' (hyphen) is read as 'if '. Further the conjunction symbol is replaced by ',' (Comma).
Repeating, the symbol ':-' is read as 'if' and comma on R.H.S stands for conjunction.
Summarizing, the predicate logic clause $P \wedge Q \wedge R \rightarrow S$ is equivalently represented in PROLOG as S: - P, Q, R. *(Note the full stop at the end)*

Problem solving style using PROLOG requires statements of the relevant *facts* that are true in the problem domain and *rules* that are valid, again, in the domain of the problem under consideration.

A **fact** is a proposition, i.e., it is a sentence to which a truth value TRUE is assigned. (*the only other truth value is FALSE*). Facts are about various properties of objects of the problem domain and about expected-to-be useful relations between objects of the problem domain.

Examples of the statements that may be facts are:

1. Mohan is tall.
   In this case is_tall is a property and if Mohan is actually tall (by some criteria), then the fact may be stated as
   is_tall (mohan).

(the reason for starting the name Mohan with lower-case 'm' and not with upper-case letter 'M' is that any sequence of letters starting with an upper-case letter, is treated, in PROLOG, as a variable, where the names like Mohan etc. denote a particular person and, hence, denote a constant. Details are given later on.

2. Anuj is father of Gopal,
   if true, may be stated as:
   father (anuj, gopal).

3. Ram and Sita are parents of Kush,
   if true, may be stated as
   parents (kush, ram, sita).

4. Gold is a precious metal
   may be stated as:
   is_precious (gold).

5. The integer 5 is greater than the integer 4
   may be stated as:
   is_greater (5, 4).

6. Aslam is richer than John,
   may be stated as
   is_richer (aslam, john).

7. Mohan is tall and Aslam is richer than John
   may be stated , using conjunct notations of predicate logic, as
   is_tall (mohan) $\wedge$ is_richer (aslam, john).

The above statements show that facts are constituted of the following types of entities viz

(i) ***Objects*** *(or rather object names) like, Mohan, Ram, Gold etc.* **In PROLOG names of the objects are called atoms.**

(ii) **Numerals** like, 14, 36.3 etc. But numerals are also names of course, that of numbers. ***However, generally, numerals are called numbers, though, slightly incorrectly.***
Also, **a constant** is either an **atom** or a **number, and**

(iii) ***Predicates or relation names*** *like, father, parent, is_precious, is_ richer and is_greater etc.* In a PROLOG statement.

Predicates are also called Functors in PROLOG

Further, the following type of entities will be introduced soon:

(iv) Variables

8. **On the other hand, there are facts** such as:

If x is father of y and y is father of z then x is grand_father of z,

which are better stated in general terms, in stead of being stated though innumerable number of facts in which x's are given *specific* names of persons, y's specific names of their respective fathers and z's specific names of respective grand-fathers.

Similarly, in stead of stating innumerable number of facts about the relation of sister as

Anita is sister of Anuj.
Sabina is sister of Aslam.
Jane is sister of Johan.,
.
.
. ,

we may use the *general statement involving variables, viz, X, Y, M and F*, in the form of the following rule:

X is sister of Y if X is a female and X and Y have the same parents M and F.

The above rule may be stated in PROLOG as:
is_sister-of (X, Y):- female(X), parents (X, F, M), parents (Y, F, M).

From the above discussion, it is now clear that knowledge of a problem domain can be stated in terms of facts and rules. Further, facts and rules can be stated in terms of

(i) **Atoms (which represent object names)** like, Mohan, Ram, Gold etc.
(ii) **numbers**

(iii) **Variables** like, X, Y and Z. A variable may be thought of as something that stands for some object from a set but, it is not known for which particular object.

(iv) **Predicates or relation names** like, father, parent, is-precious, is_richer and is_greater etc. and

(v) **Comments:** A string of characters strings generally enclosed between the pair of signs, viz, '/*' and '*/', denotes a comment in PROLOG. The comments are ignored by PROLOG system.

(vi) **Atomic formula or structure (*atomic formula is different from atom)***

Out of the eight examples of statements which were considered a while ago, the first six *do not involve* any of the logical operators, viz,
~ (negation), $\wedge$ (conjunction), $\vee$ (disjunction) $\rightarrow$ (implication) and $\leftrightarrow$ (bi-implication).

Such statements which do not contain any logical operators, are called atomic formulae. **In PROLOG, atomic formulae are called structures. In general, a structure is of the form:**

functor (parameter list)

where functor is a predicate and parameter list is list of atoms, numbers variables and even other structures. We will discuss structure again under data structures of PROLOG.

Terms: functors, structures and constants including numbers are called terms.

In a logical language, the atoms, numbers, predicates, variables and atomic formulas are basic building blocks for expression of facts and rules constituting knowledge of the domain under consideration. **And, as mentioned earlier, in logic programming style of problem solving, this knowledge plays very important role in solving problems.**

2.3 NOTATIONS IN PROLOG FOR BUILDING BLOCKS

Alphabet Set of a Language:

In any written language, whether natural or formal, the various linguistic constructs like words expressions, statements etc. are formed from the elements of a set of characters. This set is called **alphabet set of the language**.

The alphabet set of PROLOG is:

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z	*(upper-case letters)*
a b c d e f g h i j k l m n o p q r s t u v w x y z	*(lower-case letters)*
0 1 2 3 4 5 6 7 8 9	*(numbers)*
+ – * / \ ^ < > = ‘ ~ : . ? @ # $ &	
! ” % ( ) ’ | { } [ ] - ; ,	

For each type of terms, viz, numbers, atoms, variables and structures, there are different rules to build the type of terms. Next we discuss these rules.

Constants in PROLOG represent specific objects and specific relationships. Constants are of two types, viz., numbers and atoms.

Numbers: How numbers are represented in PROLOG is illustrated through the following examples of representation of numbers:

8 7 – 3.58 0 87.6e2 35.03e –12

In the above, the "e" notation is used to denote a power of 10. For example, 87.6 e 2 denotes the number 87.6 x 10^2 or 8760. The term 35.03e–12 denotes 35.03 x 10^{-12}

Atom: an atom is represented by

(i) either a *string* in which the first symbol is a *lower-case letter* and other characters in the string are letters, digits and underscores (but no sign character)

(ii) a string or a sequence of characters from the alphabet set *(including sign characters)* enclosed between apostrophes.

(iii) all special symbols like "?-" and ":-" are also atoms in PROLOG

Examples of Atoms

(i) circle (ii) b (iii) *=(equal to sign)* (iv) *_(underscore)*

(v) '→' (vi) _beta (vii) mohan (viii) 3

(ix) abdul_kalam(*uses underscore*)

(x)'abdul-kalam'

(uses hyphen. Hyphen is not allowed to be a part of an atom, but within single quotes all character of the alphabet are allowed)

(xi) 'Anand Prakash'

(the blank symbol is not allowed within a single atom.) But the whole sequence of characters in 'Anand Prakash' including single quotes, represents a single atom.

The following are not atoms

(i) 3mohan (*starts with a number*)

(ii) abdul-kalam (*has hyphen in-between*)

(iii) Abdul_kalam (*starts with a capital letter*)

Predicate: Predicate have the same notations as atoms

Variables: The following notations are used for variables in PROLOG:

(i) A string of letters, digits and underscores that begins with an *uppercase letter*.

(ii) any sequence of letters, digits and underscores which begins with an underscore, e.g,

_result

(iii) The symbol "_" (*underscore*) alone.

Underscore denotes a special type of variable, called **anonimous variable**. For example, if we want to know if there is anyone who likes Mohan, without being interested in who likes Mohan, then we may use the following statement:

?-likes (_ , mohan).

The difference between the behaviours of other variables like 'X' and '_' is explained by the following two queries.

In the query

?-likes (X, X).

the two occurrences of X denote the same person, hence, if first occurrence of X is associated with Ankit then second occurrence of X is automatically associated with Ankit. Thus, the above statement in this case says 'Ankit like himself'. However, in the following query, different occurrences of "-" may be associated with different constants:

?- likes (-, -).

The query above asks to find someone who likes someone, the second someone may be different from the first someone. Thus, for

?- likes (-,-).

The PROLOG system, may respond as

'*Ankit likes Suresh*', where first occurrence of '- ' is associated with Ankit and the second occurrence of '-' is associated with 'Suresh'.

Structure: As mentioned earlier, a structure is of the form:

Predicate (parameter list)

where parameter list consists of atoms and variables separated by commas .

Term: We have already mentioned that a term is either a constant or a structure. And, we have also already discussed representations of constants and structures.

Fact: A simple statement *(i.e., statement not involving logical operators:* $\sim \ \wedge \ \vee \ \rightarrow$ *and* $\leftrightarrow$*)* is represented by a structure *followed by a full-stop.*

Example

is_sister(anita, ankur).

(states the fact that Anita is a sister of Ankur).

owns (mohan, book).

(states the fact that mohan owns a book)

Fact with Compound Statement: *For expressing facts and relations, the syntax of PROLOG does not allow arbitrary clauses but only* ***Horn Clauses****.* However, in PROLOG goals may be conjuncted. And, we know a Horn Clause can have at most one positive literal. Therefore, the following single (non-atomic) statement of predicate calculus

$is\_tall\ (mohan) \wedge is\_richer\ (aslam, john)$

has to be expressed as two Prolog statements as follows:

is_tall (mohan).
is_richer (aslam, john).

(note the dot at the end of each statement)

Head & Body of a Rule

In the rule

a_4*:-* a_1*,* a_2*,* a_3*,* a_5*.*

a_4 is the **Head** of the rule and the R.H.S of ':-' , i.e., 'a_1, a_2, a_3, a_5' is the **body** of the rule.

Repeating what has already been said.

The ***Head*** *of a rule represents a goal to be achieved and the* ***Body*** *represents one or more of the subgoals (each represented by single (atomic) structure)* ***each of which*** *must be achieved if the goal represented by the Head can be said to have been achieved.*

Rule Statements: In general, a rule in PROLOG is of the form:

Head:- Body.

where *Head* is a (single) structure and *Body* is a finite sequence of structures separated by commas.

Query/Question Statements:

We mentioned earlier, computer programming in PROLOG consists of:

- specifying facts,
- defining rules, and
- asking questions,

about objects and their relationships

We have already discussed PROLOG notation for specifying facts and defining rules. ***Next, we discuss PROLOG representation of questions*.** The PROLOG representation of questions is almost the same as that for facts. The only difference in the representations is that representation of a fact when preceded by the symbol '?-' (*question mark followed by hyphen*) becomes the representation of a question. For example, the statement

is_tall (mohan).

represents **the fact** which when expressed in English becomes '*Mohan is tall*'. However, **the following PROLOG notation**

?- is_tal (mohan).

denotes **the question** which when expressed in English becomes: '*Is Mohan tall*?'

We mentioned earlier that for representing facts and rules in PROLOG, we are restricted to using only Horn Clauses. However, the solving of a problem using a PROLOG system, *requires asking questions, any one of which may be an atomic goal or may be a conjunct of more than one atomic goals*. In the later case, **the conjunct of atomic goals** representing the (composite) question is represented in Prolog by writing the atomic goals separated by commas.
For example, if we want to know whether '*Mohan is tall* ***and*** *Aslam is richer than John*?' then the question may be stated in PROLOG as

?- is_tall (mohan), richer (aslam, john).

Some interpretations of question forms in PROLOG
Let us consider the rule:
X is sister of Y if X is a female and X and Y have the same parents M and F.
The above rule may be stated in PROLOG as:

is_sister of (X, Y):- female(X), parents (X, F, M), parents (Y, F, M).

Case (i) Then the question in PROLOG

?- is_sister (jane, john).

represents the question (in English): *Is Jane sister of John?*

Case (ii) Then the question in PROLOG.

?-is_sister (X, john).

represents the question:
Who (represent by upper-case letter X) are John's sisters, if any?
Rather, the above PROLOG statement is a sort of the following command:
Find the names *(represents by upper-case letter X)* of (all) sisters of John.

Case (iii) Further, the PROLOG question:

?-is_sister (jane, X).

represents the command:
Find the names (represented by the upper-case letter X) of all siblings (brothers and sisters) of jane.

Case (iv) Still further, the PROLOG question:

?-is_sister (X, Y).

represents the command:
Find all pairs (represented by pair (X, Y) of, persons in which the first person is a sister of the second person in the pair, where second person may be a male or a female.

A **PROLOG program** consists of a finite sequence of facts, rules and a query or goal statement.

In order to discuss solutions of problems with a PROLOG system, in addition to what we have discussed so far we need to discuss in some detail representation of **arithmetic facts, rules and goals**. Also, we need to discuss in some details **data structures** in PROLOG. These topics will be taken up later on. Next, we discuss how a PROLOG system solves a problem.

2.4 HOW PROLOG SYSTEM SOLVES PROBLEMS?

We have mentioned earlier also that solution of a problem through PROLOG system depends on

(i) (contents of) PROLOG database or knowledge base. **PROLOG database consists of facts and rules.**

(ii) PROLOG inferencing system, which mainly consists of three mechanisms viz
   (i) Backtracking,
   (ii) Unification,
   (iii) Resolution.

If a PROLOG database does not contain sufficient relevant facts and rules in respect of a particular query, then the PROLOG system will say 'fail' or 'no', even if the facts in the query, be true in everyday life.

For example: Suppose that 'Sita is a sister of Mohan' is a fact in the real world. However, if in the database, we are **not given any of the following:**
   (i) is_sister (sita, mohan).
   (ii) parents (sita, m, f).
   (iii) parents (mohan, m, f).
   (iv) mother (sita, m) and mother (mohan, m).
   (v) father (sita, f) and father (mohan, f).

More generally, **if we are not given** any set of statements or relations from which we can conclude that *Sita is sister of Mohan*, then PROLOG system would answer the query:

?- is_sister (sita, moha).

as something like: **"Sita is a sister of Mohan" is not true.**

Further, even if all the statements given under (i) to (v) above are in the database, but the following rule

(vi) is_sister (X, Y):- female (X),
parents (X, M, F), parents (Y, M, F).

is not given in the PROLOG base then also, the system would answer as something like: **"Sita is a sister of Mohan" is not true.**

Further, even if all the statements given under (ii) and (iii) and even rule (vi) are in the database, but some statement equivalent to the fact *female (sita).*

is not given, then also the system would answer as something like:

"Sita is a sister of Mohan" is not true.

*Let us Assume the PROLOG database is **complete** in the sense that the database contains all the required facts and rules about the real world that are sufficient to answer any relevant query.* Then, we discuss how the PROLOG system solves a problem under consideration. In order to explain the in-built mechanism of PROLOG system, we start with the particular database given below and some queries and then generalize the results of our discussions.

Given Database

(i) female (sita).
(ii) female (zarina).
(iii) female (sabina)
(iv) female (jane).
(v) is_sister (sita, sarita).
(vi) is_sister (anita, anil).
(vii) parents (sita, luxmi, raj).
(viii) parents (sarita, luxmi, raj).
(ix) parents (sabina, roshnara, Kasim).
(x) parents (isaac, mary, albert).
(xi) parents (aslam, roshnara, Kasim).
(xii) parents (zarina, roshnara, Kasim).
(xiii) father (jane, albert).
(xiv) father (john,albert).
(xv) father (Phillips, albert)
(xvi) mother (jane, mary).
(xvii) mother (john, mary).
(xviii) mother (phillps, ann).

(jane is half sister of Phillips and according to the definition of the PROLOG database, jane is not a sister of Phillips.)

(xix) is_sister (X, Y):- female (X),
parents (X, M, F), parents (Y, M, F).

(xx) is_sister (X, Y):- female (X), mother (X, M),
mother (Y, M), father (X, F),
father (Y, F).

The last rule could have been replaced by the following rule:

(xxi) parents (X, Y, Z):- mother (X, Y), father (X, Z)

But to explain some important points we consider rule (xx) and do not consider the last rule.

Now, through a number of query examples, we illustrate the way in which a PROLOG system solves problems.

Query No. 1 For the database given above, consider the query:

?-is_sister (anita, anil).

The PROLOG system starts from the top of the database and matches the functor *is_sister* of the query with the functor *female* of the statement which do not match (*both being constants must be identical for matching*). Then PROLOG system passes to the next statement and attempts to match *is_sister* with the functor also *female* of the second statement in the database. In this case also, the corresponding functors do

not match. Similarly, functor in the query does not match the functor in each of the next two statements.

The PROLOG system passes to next (i.e., fifth) statement. The functors in the query and the third statement are identical. Hence, the query system attempts to match, one by one, the rest of the parts of the query with the corresponding parts of the fifth statement. The first argument viz *anita* of the query does not match (*being constants, are required to be identical for matching*) to the first argument viz *sita* of the fifth statement.
Hence, the PROLOG system passes to sixth statement in the database. The various components of the query match (in this case are actually identical) to the corresponding components of the sixth statement. *Hence, the query is answered as 'yes'.*

Query No. 2 With the same database, consider the query

?- is_sister (Sabina, aslam).

The PROLOG system attempts to match the functor *is_sister* of the query with funtors, one by one, of the facts and rules of the database. The first possibility of match is in Fact (v) which also has functor *is_sister*. However, the first argument of the functor in Fact (v) is *sita* which does not match *sabina*, the corresponding argument. Hence, the PROLOG system attempts to match with Fact (vi) which also does not match the query. The PROLOG system is not able to find any appropriate matching upto fact (xviii).

However, the functor *is_sister* in the query matches the functor *is_sister* of the L. H. S. of the Rule (xix). In other words, PROLOG system attempts to match the constant *sabina* given in the query with the variable X given in the rule (xix). **Here matching takes a different meaning called unification.**

Unification of two terms, out of which at least one is a variable (*i.e., the first letter of the term is an upper case letter*), **is defined as follows:**

(i) If the term other than the variable term is a constant, then the constant value is temporarily associated with the variable. And further throughout the statement or rule, the variable is temporarily replaced by the constant.

(ii) If both the terms are variables, then both are temporarily made synonym in the sense any value associated with one variable will be assumed to be associated temporarily with the other variable.

In this case, the variable temporarily associated with the constant *sabina* for all occurrence of X in rule (xix).

Next, PROLOG system attempts to match second arguments of the functor *is_sister* of the query and of L. H. S. of rule (xix). Again this matching is a case of unification of Y being temporarily associated with the constant *aslam* through out the rule (xix).

Thus, in this case, rule (xix) takes the following form:

is_sister (sabina, aslam):- female (sabina), parents (sabina, M, F), parents (aslam, M, F).

And, we know the symbol ':-' stands for 'if '. Thus, in order to satisfy the fact (or to answer whether) sabina is sister of aslam, the PROLOG system need to check three subgoals viz, *female (sabina), parents (sabina, M, F) and parents (aslam, M, F).*

Before starting to work on these three subgoals, the PROLOG system marks the rule (xix) for future reference or for backtracking (*to be explained*) to some earlier fact or rule.

The first subgoal*: female (sabina)* is trivially matched with fact (iii).
In view of the fact that except sabina the other two arguments in the next (sub)goal viz *parents (sabina, M, F)* are variables, the subgoal *parents (sabina, M, F)* is actually a sort of question of the form:

Who are the parents of sabina?

For the satisfaction of this subgoal, the PROLOG system again starts the exercise of matching/unification from the top of the database, i.e, from the first statement in the database. The possibilities are with facts (vii), (viii),….., (xii), in which the functor *parents* **of** the subgoal occurs as the functor of the facts (vii), (viii),….., (xii). However, the first arguments of the functors in (vii) and (viii) do not match the first argument of parents in the subgoal. Hence the PROLOG system proceeds further to Fact (ix) for matching. At this stage the subgoal is

parents(sabina, M, F).

and the fact (ix) is

parents (sabina, roshnara, Kasim).

As explained earlier, M and F, being upper case letters, represent variables. Hence, the exercise of *matching* becomes exercise of *unification*. From the way we have explained earlier, the constant *roshnara* gets temporarily (*for the discussion of rule (xix) only*) gets associated with the variable M and the constant *kasim* gets associated with the variable F. As a consequence, the next goal *parents (aslam, M, F)* becomes the (sub) goal *parents (aslam, roshnara, kasim)*. To satisfy this subgoal, the PROLOG system again starts the process of matching from the first statement in the PROLOG database. Ultimately, after failure of matching with the first eight facts in the database, subgoal is satisfied, because of matching with the fact (ix) in the database.

The PROLOG system answers the query with 'yes'.

Remarks: In respect of the above database, the relation of *is_sister* is not reflexive (*i.e, is_sister (X, X) is not true for all numbers of the database*), i.e., no male is his own sister. However, for the set of all females, the relation is reflexive.

Remarks: In respect of the above database, the relation of *is_sister* is also not symmetric (i.e., if *is_sister (X, Y)* is true then it is not necessary that *is_sister (Y, X)* must be true). Any female X may be a sister of a male Y, but the reverse is not true, because the condition *female (Y)* will not be satisfied. But the relation is symmetric within the set of all females.

Remarks: However, the relation of *is_sister is* (fully) transitive (i.e., if *is_sister (X, Y)* is true and if *is_sister (Y, Z)* is true, then *is_sister (X, Z)* must be true)

Ex 1: Query No. 3 With the same database, discuss how the following query is answered by the Prolog system:

?- is_sister (aslam, sabina).

Ex 2: Query No 4. With the same database, discuss how the following query is answered by the Prolog system:

?- is_sister (jane, john)

Ex 3: Query No. 5 With the same database, discuss how the following query is answered by the Prolog system:

?- is_sister (jane, Phillips)

2.5 BACKTRACKING

The concept of *backtracking* is quite significant in PROLOG, specially in view of the fact that it provides a powerful tool in searching alternative directions, when one of the directions being pursued for finding a solution, leads to a failure. Bactracking is also useful when to a query, there are more than one possible solutions. First, we illustrate the concept through an example and then explain the general idea of backtracking.

Example 1: For the database given earlier consider the query

?- is_sister (sabina, Y), is_sister (Y, zarina).

The query when translated in English becomes: *'Find the names (denoted by Y) of all those persons for whom sabina is a sister and further who (denoted by Y) is a sister of zarina.'*

In order to answer the above query, the first solution which the PROLOG system comes with after searching the database is : **'Associate sabina with Y',** i.e., *sabina is one of the possible answers to the query (which is a conjunct of two propositions).* In other words, associate Y with *sabina*, which, according to the facts and rules given in the database, satisfies *is_sister (sabina, sabina)* and *is_sister (sabina, zarina)*.

If we are interested in more than one answers, which are possible in this case, then, after the answer 'sabina' the user should type the symbol ';' (i.e., type semi-colon). Typing a semi-colon followed by return' serves as a direction to PROLOG system to search the database from the beginning once again for an alternative solution.

In order to prevent the PROLOG system from attempting to find the same answer: *sabina* again, the system puts *markers* – one on Rule (xix) and after that on Fact (ix) (*in that order*).

Once the instruction from user through semi-colon is received to find another solution, the system proceeds to satisfy Rule (xix) from Fact (x) (*including*) onwards. Then through Fact (xi), for the first occurrence of Y, *aslam* is associated. The variable X is already associated with constant *sabina*. Then *aslam* replaces Y in the rule (xix). Next, PROLOG system attempts to satisfy the second subgoal which, at present, is of the form:

is_sister (aslam, zarina).

For satisfying this goal, the PROLOG system searches the database from the top again. Again rule (xix) is to be used. For satisfying L.H.S. of (xix) the subgoal on R.H.S. of (xix) need to be satisfied. The first subgoal on R. H. S. of (xix) is to satisfy the fact: *female (aslam)*, which is not satisfied.

At this stage, the PROLOG system goes back to the association i.e. *aslam to Y*, and removes this association of Y with *aslam*. Next, the PROLOG system attempts to associate some other value to Y, further from the point where Y was associated with aslam. **This is what is meant by Backtracking.**

Next, through Fact (xii), zarina is associated with Y and sabina is already associated with with X. Thus, the subgoal to be searched becomes *is_sister (zarina, zarina)*. This goal can be satisfied, because, three subgoals for this goal, viz, *female (zarina), parents (zarina, M, F)* (where zarina is associated with X) and *parents (zarina, M, F)* (where zarina is associated with Y) can be easily seen to be satisfiable from the database. Hence, the second answer which the system gives is *zarina*.

Again, the user may seek for another answer to the query by typing ';'. The system has already marked the fact (xii) in the database while finding the answer: *zarina*. Therefore, for another answer, the PROLOG system starts from the next statement, i.e, Fact (xiii) to search. It can be easily seen that the PROLOG system will not find any more answers. Hence, it returns 'fail' or 'NO'.

2.6 DATA TYPES AND STRUCTURES IN PROLOG

We have already discussed the concepts of **atom** and **number**. These are the only two elementary data *types* in PROLOG. We also mentioned how atoms and numbers are represented in PROLOG.

We will discuss briefly: *relations between numbers, operations on numbers* and *statements about numbers* later. The discussion is being postponed in view of the fact that main goal of PROLOG is *symbolic processing* rather than *numeric processing*.

We next discuss how *symbolic data* is structured out of the *two elementary data types* viz. *atoms* and *numbers*. **PROLOG has mainly two data structures:**
(i) List
(ii) Structure.

To begin with, we consider the data structure: *Structure*

Structure

We have already discussed the concept of *structure*, particularly, in context of representing facts, rules and queries. A **structures** in PROLOG is of the form:
(predicate (list-of-terms),
where *predicate* is an atom and *list-of-terms* is a list of terms.

A **fact** is represented in PROLOG by simply putting a full stop at the end of appropriate structure. Similarly, a rule or a query can be obtained from appropriate structures.

Here we discuss briefly *structure* from another perspective, viz, that of representing complex data. *Structure* in this respect is like *record structure* of other programming languages.
Structure is a *recursive* concept, i.e., in the representation *predicate (list-of-terms)*, a term itself may be a structure again. Of course, a term may also be a constant or a variable. For example, an employee may be defined as
employee (name, designation, age, gross-pay, address)
but then name may be further structured as

Name (first-name, middle-name, last-name)

further first-name may be written, for example, as

(first-name mohan)
Thus, the information about an employee Mohan may be written in PROLOG as a fact using a (complex) structure:

employee *(name (first_name mohan), (middle-name lal),*
(last_name sharma)),
(designation assistant_registrar),
(age 43), (gross_pay 35000
(address delhi).

Similarly address may be further structured, for example, as

address ((house-number 139),
(street khari-baoli)
(area chandni-chowk)
(city delhi))

A query in order to know the gross-pay of Mohan may be given as:
?_ employee (name (mohan), _ , _), _ , _) (gross_pay X)).
It will return 3500. In the above query, the underscores may represent many different variables.

Ex 4: Give the information about the book: *Computer Networks, Fourth Edition by Andrew S. Tanenbaum published by Prentice Hall PTR in the year 2003* as a structure in PROLOG. Further, write a query in PROLOG to know the name of the author, assuming the author's name is not given

---

Example 2: The following simple sentence:
Mohan eats banana
may be represented in PROLOG as

sentence ((noun mohan), (verb_phrase (verb eats), (noun banana))).

In general, the structure of a sentence of the form given above may be expressed in PROLOG as

Sentence ((noun N1), (verbphrase (verb v), (noun N2)).

Further, A query of the form

?_sentence ((noun mohan), (verbphrase (verb eats), (noun X)))

tells us about what Mohan eats as follows:
(*noun banana)*

Next we consider the data structure: List

List: The *list* is a common data structure which is built-in in almost all programming languages, specially programming languages meant for *non-numeric* processing.

Informally, **list** is an ordered sequence of elements and can have any length *(this is how a list differs from the data structure array; array has a fixed length)***.** The elements of a list may be any terms (i.e., constants, variables and structures) and even other lists. Thus, *list* is a recursive concept.

***Formally*, a *list* in PROLOG may be defined recursively as follows:**

(i) [ ], representing the empty list, is a list,
(ii) [e_1 , e_2 ,…… e_n] is a list if each of e_i is a term, a list or an expression *(to be defined).*

Examples of Lists: [ ], [1,2,3], [1, [2, 3]]

(*note the last two are different lists and also note that the last list has two elements viz 1 and [2,3]*

Also, [3, X, [a, [b, X + Y]]] is a list

provided X + Y is defined. For example, X and Y are numbers, then as we shall see later that X + Y is a valid expression.

Expression is a sequence of terms and operators formed according to syntactic rules of the language. For example, X + Y * Z may be an expression, if X, Y and Z are numbers and if *infix* notation for operations is used.

However, if *prefix* notation is used, then X + Y * Z is *not* an expression, but +X∗YZ be may an expression.

Also [_ , _, Y] is a list of three elements, out of which, first two are '*don't care*' or *anonymous* variables.

2.7 OPERATIONS ON LISTS IN PROLOG

I. The operation denoted by the vertical bar, i.e. '|' is used to associate Head and Tail of a list with two variables as described below:

(i) [X |Y] = [a, b, c]

then X is associated with the element **a**, the first element of the list and Y is associated with the list [b, c], obtained from the given list by removing its first element, if any.

(ii) [X |Y] = [[a, b] , c]]

then X is associated with the list [a, b], the first element of the list on R.H.S. and Y is associated with [c].

(iii) [X |Y] = [[a, b, c]] then

X = [a, b, c] and Y = [ ]

(iv) [X |Y] = [a, [b, c]]

then X is associated with **a** and Y is associated with [[b, c]]

(v) [X|Y] = [a, b, [c]] then

then X is associated with **a** and Y is associated with [b, [c]]

(vi) The operator '|' is not defined for the list [ ], having no elements

(vii) [a + b , c + d] = [X, Y]
then X is associated with a + b and Y is associated with c + d
(Note the difference in the response because vertical bar is replaced by comma)

II. Member Function the function is used to determine whether a given argument X is a member of a given list L. Though the member function is a *built-in function* in almost every implementation of PROLOG, yet the following simple (recursive) program in PROLOG for *member* achieves the required effect:

```
member(X, [X| _ ].                          (i)
member (X, [_ |Y]):- member (X, [Y]).       (ii)
```

Definition of member under (i) above says that if first argument of member is Head of the second argument, then predicate *member* is true. If *not so, then, go to definition under (ii)*. The definition of member under (ii) above says that, in order to find out whether first argument X is a member of second argument then find out whether X is a member of the list obtained from the second argument by deleting the first element of the second argument. **From the above definition, it is clear that the case member (X, [ ]) is not a part of the definition. Hence, the system returns FALSE.** *Next, we discuss example to explain how the PROLOG system responds to the queries involving the member function.*

Example 3:

?-member (pascal, [prolog, fortran, pascal, cobol])

The PROLOG system first attempts to verify(i) in the definition of 'member', i.e., system matches *pascal* with the Head of the given list [*prolog, fortran, pascal, cobol*]. i.e., with prolog. The two constants are not identical, hence, (i) fails. Therefore, the PROLOG system uses the rule (ii) of the definition to solve the problem. According to rule (ii) the system attempts to check whether *pascal* is a member of the tail [*fortran, pascal, cobol*] of the given list.

Again fact (i) of the definition is applied to the new list, i.e., [*fortran, pascal, cobol*] to check whether *pascal* belongs to it. As *pascal* does not match the Head, i.e, *fortran* of the new list. Hence, rule (ii) is applied to the new list. By rule (ii), *pascal* is a member of the list [*fortran, pascal, cobol*], if *pascal* is a member of the tail [*pascal, cobol*] of the current list.
Again fact (i) is applied to check whether *pascal* is a member of the current list [*pascal, cobol*]. According to fact (i) for *pascal* to be a member of the current list pascal should be Head of the current list, which is actually true. **Hence, the PROLOG system returns 'yes'.**

The above procedure may be concisely expressed as follows:

Goal	**X**	**[– | Y]**	**Comment**
1.	pascal	[prolog|fortran, pascal, cobol]	*pascal does not match prolog hence apply rule (ii)*
2.	pascal	[fortran, pascal, cobol]	*by rule (ii)*
3.	pascal	[fortran|pascal, cobol]	*using fact (i)*

			pascal is not identical to head of the list Hence apply rule (ii)
4.	pascal	[pascal, cobol]	
5.	pascal	[pascal|cobol]	*(again apply fact (i))*

Head of the current list is identical with pascal, hence, the system responds with a 'yes'.

Ex 5: Explain how the PROLOG system responds to the following query:
?-member (pascal, [prolog, fortran, cobol]).

III. APPEND function is also generally, a built-in function in PROLOG. However, we can see how a simple program in PROLOG can be written for the function.

The append function takes two lists and returns a single list such that the elements of the first list followed by elements of the second list constitute the elements of the returned list. For example

(i) append ([a, b] [c , X, Y]) *returns the list*
[a, b, c, X, Y].
(ii) append ([a, [b, c]], [[c, X], Y]) *returns the list*
[a, [b, c], [c, X], Y]
(iii) append ([ ], [a, b, c]) *returns the list*
[a, b, c].
(iv) append ([a, b, c], [ ]) also *returns the list*
[a, b, c].

The append function in PROLOG may be written as:

```
append ([ ], X, X).                                          (i)
append ([Head|Tail], Y, [Head|Z]:- append ([Tail], Y, Z).    (ii)
```

Explanation: The above PROLOG program states that *append* takes three arguments, each of which should be a list. The result of appending the elements of the lists in the first and second arguments, is stored in the list represented by third argument.

Further, the result of application of *(i) of the definition of append* to an empty list in first argument with a list X in second argument is the same list X as given in second argument, but written in third argument.

However, if the first argument is not [ ] then apply rule (ii) above.
According to rule (ii) above, Head of the list in first argument becomes Head of the resultant list (i.e., of the third argument) and the tail of the resultant list is obtained by appending tail i.e., [tail] of the first argument to the list given as second argument of append. Executing *append* according to the above definition is a recursive process. *In each successive step, the size of the list in the first argument is reduced by one and finally the list in the first argument becomes [ ].* In the last case, fact (i) is applicable and the process terminates.

Next, we explain how PROLOG system responds to a query involving append.

Example 4: Let us consider the query

?-append ([prolog, lisp], [C, fortran], Z).

As first list is not [ ], therefore the system associates *prolog* with Head and [*lisp*] with Tail. Then by rule (ii) prolog becomes the Head of the resultant list and after that PROLOG system attempts to append the list [*lisp*] with [*C, fortran*] and the result will form the tail of the originally required resultant list. Next, list is the Head of new first argument and [ ] is the tail of the new first argument. However, second argument remains unchanged. The result of the second append is a list whose *first* element is *prolog*, second element is *lisp* and the rest of the elements of the resultant list will be obtained by appending [ ] to [C, fortran]. In this case however rule (i) is applicable which returns [C, fortran] as the result of append [ ] to [C, fortran], and which will form the tail of the resultant list. Finally, the resultant list is [prolog, lisp, C, fortran]

Next, we consider another list function.

IV. prefix (X, Z) function which *returns true if* X is a sublist of Z, such that X is either [ ] or X contains any number of consecutive elements of the list Z starting from the first element of Z onwards. Otherwise, prefix (X, Z) returns 'No' or 'False'. Further explain, justify and trace your program with an example.
The PROLOG program prefix is just one statement program:

prefix (X, Z): - append (X, Y, Z).

where, we have already defined append as
append ([ ], X, X]). (i)
append ([Head | Tail], Y, [Head | Z]):- append ([Tail], Y, Z). (ii)

Explanation: The one-line program, returns true, if by appending any list Y to X we get Z. Further, next two-statement program *append* says that append, a function of three arguments X, Y, Z returns Z as a list in which first, all the elements of X occur preserving their original order in X. The last element of X when placed in Z is followed by elements of Y, again preserving their order in Y.

Further explanation may be given with the following example query:

?-prefix ([a, b], [a, b, c]).

returns yes. The processing for the response, by the PROLOG system may be described as follows:

For satisfying the query, the rule becomes
prefix ([a, b], [a, b, c]):- append ([a, b], Y, [a, b, c])
Next, the system need to satisfy the goal:
append ([a, b], Y, [a, b, c].
First fact of append is not applicable as [a, b] is not [ ]. The rule (ii) takes the form
append ([a | b], Y , [a | b, c]):-append ([a], Y, [b, c]).
Which on another application of rule (ii) takes the form
append ([a |], Y, [c]):- append ([ ], Y, [c]).
Y is associated with [c]. The PROLOG system responds with
Y = [c]

Let us consider another query:

?-prefix (X, [a, b, c]).

In response to the query the PROLOG system, first returns X as [ ], then if the user gives (;) to find another answer, then PROLOG system returns X as the list [a]. If, further another answer is required then PROLOG system returns X as the list [a, b]. If still another answer is required, the answer [a, b, c] is returned. Finally, if still another response is required by the user, then the system responds with a 'No'.

Ex 6: Another Example query
?-prefix ([a, c], [a, b, c]

Explain the sequence of steps of processing by PROLOG system.

Ex 7: Write a PROLOG program **suffix (Y, Z)** which returns true if 'Y is a sublist of Z such that either Y is [ ] or Y contains any number of consecutive elements of Z starting anywhere in the list but the last element of Y must be the last element of Z. Further, explain, justify and trace your program with an example.

2.8 THE EQUALITY PREDICATE '='

Let *Term1* and *Term2* be two terms of PROLOG, where a term may be a constant, a variable or a structure. Then the success or failure of the goal

?-Term1 = Term2.

is discussed below:

Case I: Both Term1 and Term2 are constants and, further, if both are identical then the goal succeeds, and, if not identical, then the goal fails.

Examples: The query

?-mohan = mohan *succeeds*
?-1287 = 1287 *succeeds*
?-program = programme *fails*
?- 1287 =1289 *fails.*

Case II: One of the terms is a variable and if the variable is not instantiated then the goal always succeeds

For example

?-brother (mohan, sita) = X.

Then the goal succeeds and the variable X is instantiated to the term
brother (mohan, sita)

Case III: One of the terms is a variable and the variable is instantiated:

Then apply '=' recursively to the case which is obtained by replacing the variable by what is its instantiation. For example, consider the query

?-X = tree.

And X is already instantiated to *tree,* then, replace X by tree in the given query, which takes the new form:

?tree = tree

On encountering this new query, PROLOG system returns *success.* However, if X is already instantiated to any other constant say *flower,* then the new query becomes:

?-flower = tree.

Applying Case I above, we get *Fail.*

Case IV: If both terms are variables, say, the query is of the form:

?-X = Y

then if

(a) both are *uninstantiated* the query succeeds, but, if during further processing one of X or Y gets instantiated to some term then other variable also gets instantiated to the same term.

(b) If one or both are instantiated, then replace X and/or Y by their respective instantiations to generate a new query and apply '=' to the new query.

Case V: Both terms are structures: Two structures satisfy '=' if they have the same functor and the same number of components and, further, if the definition of '='is applied recursively to the corresponding components, then each pair of corresponding components satisfies '='.

For example:

?-brother (mohan, X) = brother (Y, sita).

Then the above goal succeeds and X is instantiated to *sita* and Y is instantiated to *mohan*

Similarly the following goal also succeeds

?-p(Q, r, S, T, u, v) = p(q, r, s, W, U, V).

because, first of all, the functors, i.e., p on the two sides of '=' are identical. Next, success of '=' for the terms at corresponding positions is discussed below:
variable Q is instantiated to q, the variable S to s, U to u and the variable V to v. Further, the variables T and W are, though, not instantiated, yet they become co-referred variables in the sense that if one of these is instantiated to some constant then the other also gets instantiated to the same constant.

Exercise 8: Discuss success/failure of each of the following queries:

(i) ?-g(X, a(b, c)) = g (Z, a (Z, c))
(ii) ?-letter(H) = word (letter)
(iii) ?-g(X, X) = g(a, b).
(iv) ?- noun (alpha) = Noun.
(v) ?-noun (alpha) = noun.

2.9 ARITHMETICS IN PROLOG

The language PROLOG has been designed mainly for symbolic processing for A.I applications. However, some facilities for numeric processing are also included in PROLOG. For this purpose, the **arithmetic operations** viz

*+ (addition), – (substraction), * (multiplication), / (division) and ∧ (exponentiation)*

are built-in and are used in **infix notation**.

Further **relational operators** viz
< (less than), > (greater than), = (equal to), = < (greater than or equal to), > = (less than or equal to) have their usual meanings and infix notation is used in expressions using these relational operators. *The symbol for 'not equal to' is /=*
Please note the notation for 'less than or equal to' and 'greater than or equal to'

The 'is' operator

It may be noted that in PROLOG, *the expression '3 + 7' is not the same as the number 10*. The operation of '+' does not execute automatically. For the purpose of execution of an operation, PROLOG provides the operator 'is'.

Thus, in response to the query

?- X is 3 + 7.

the variable X gets instantiated to 10
and the prolog system responds as

X = 10
Yes

In order to explain numerical programming in PROLOG, let us consider the PROLOG program for factorial:

factorial (0, 1).
factorial (Number, Result):- Number > 0, New is Number – 1, factorial (New, Partial),
*Result is Number * Partial.*

Note: In the above definition, each of *Number, New, Partial and Result*, denotes a variable.

Explanation of PROLOG program for factorial: If the given number is 0, then its factorial is 1. Further, result of computation for factorial of any number *Number* will be associated with the variable *Result* and the goal can be achieved through the following four subgoals:

(i) Number > 0 should be true,
(ii) The number *Number – 1* is calculated through the operator **'is'** and is associated with variable *New*,
(iii) Factorial of the number *(Number – 1)* through the operator **'is'**, i.e., of the number *New,* is recursively calculated and the result of the calculation is associated with the variable *Partial*
(iv) Finally, through the operator **'is'**, the product of the Number with *Partial*, (the result of the factorial of New (i.e. of Number – 1)) is calculated and associated with Result.

2.10 THE OPERATOR CUT

In some situations, as discussed by the following example, if the *goal is met once*, the problem is taken as *solved* without need for iterations any more.

Example: It is required to check whether an element is a member of a list X, where, say X = {c, d, a, b, a, c}, then, if we test the elements of X from left to right for equality with **a**, then the third element in the list matches **a** and once that happens, we are not interested in any more occurrences of **a** in the list X. Thus in such a situation, it is required that the PROLOG system to stop any further computation.

However, we know that PROLOG system attempts to re-satisfy a goal, even after satisfaction of the goal once. For example, for the following one statement PROLOG program

prefix (X, Z):- append (X, Y, Z).

if the query is

?-prefix (X, [a, b]).,

then first X is associates with [ ] as X = [ ] as an answer to the query.

In the next iteration, the PROLOG system associates the list [a] to X, i.e., .X = [a].
In the still next iteration, the PROLOG system responds with

X = [a, b].

And finally, the system responds with a 'no'.

The PROLOG system, provides a special mechanism or function called 'cut' by which, if required, the subgoals on the right-hand side of a rule may be prevented from being tried for more than once, if all subgoals succeed once. The operator cut is denoted as ! (the exclamation mark) and inserted at a place where the interruption for not retrying is to be inserted.

Example 5: Through the following general example we explain how PROLOG system behaves in the presence of *cut denoted by !* when inserted on the R. H. S of a rule.

Consider the rule
trial: - a, b, c, !, d, e, f, g.

The above rule says predicate *trial* succeeds if the subgoals on R.H.S. succeed. The PROLOG system may backtrack between subgoals a, b and c as long as it is required by the system answer the query, for the predicates a, b and c. *The change in the behaviour of PROLOG system due to the presence of '!' occurs only after the subgoal c succeeds and PROLOG system encounters the* ***cut symbol '!'.*** **PROLOG system always succeeds on the cut operator represented by '!'**. And hence PROLOG system attempts to satisfy the subgoal d. Further, if d succeeds then backtracking may occur between d, e, f and g.

In the process, it is possible that several backtrackings may occur between a, b and c and several (independent) backtrackings may occur between d, e, f and g. **However, once d fails and crosses on the left to the operator !**, then no more attempts will be made to re-satisfy c on the left of the operator !.

Example 6: We define below the membership for a set (*in a set each element occurs only once)*. Hence, once a particular element is found to occur then there is no need for further testing, for the occurrence of the element in a set. The predicate *member,* in this sense, may be defined as

member (X, [X | –]):- !. (i)
member (X, [_| Y]: - member (X, Y). (ii)

The statement (i) above says that if the element X is the Head of the list (i.e., X is the first element of the list) then the R.H.S. must succeed. However, R.H.S. consists of only the cut symbol '!' which is always true. Further, in the case when the operator is the cut symbol then PROLOG system is not allowed to go to its Left. Hence, the PROLOG system after succeeding on '!' exits program execution because of the restriction that the system is not allowed to backtrack from '!'.

Example 7: One of the possible PROLOG programs for finding *maximum* of two given numbers is the following two-statement program:

max (X, Y, X):- X > = Y. (i)
/* *third argument is for the variable supposed to be associated with the result* */
max (X, Y, Y): - X < Y. (ii)

But we know that if X > = Y then X is the maximum of the two, otherwise Y is the maximum. In other words, the comparison X < Y is not required if the first rule fails. This wastage of effort may be saved by using the following program for *maximum*, in stead of the one given through (i) and (ii) above.

The more efficient program for the required solution is

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

Example 8: To write a PROLOG program that adds an element X to a given set L *(i.e., L does not contain any duplicate elements)*. Therefore, if X ∈ L, then nothing needs to be done, else, a new list L_1 may be returned in which X is the Head of L_1 and additionally contains all the elements of L and no other elements.

Thus the program may be written as

```
add (X, L, L): - member (X, L), !.
add (X, L, [X | L]).
```

Next, we see how the procedure *add* behaves on various arguments.

```
?-add (c, [d, f], L)
/* the system responds with */
L = [c, d, f]

?-add (X, [d, f], L)
/* the system after executing only the first statement returns */
L = [d, f]
X = d
```

Example: In order to explain the use of *cut*, we write a program to find the *factorial(N)* using cut as follows:

```
fact (N, 1) :- N < = 1, !
fact (N, F):- M is N – 1, !
              fact (M, F1),
              F is F1 * N.
```

2.11 CUT AND FAIL

Before discussion of the *cut and fail* combination, let us first discuss the predicate *fail*. The predicate *fail* wherever occurs always fails *and initiates backtracking*. For example, Let *number* denote a predicate such that *number (X)* is true whenever X is a number, else it fails. Further, suppose *sum (X, Y, Z)* associates with Z the sum of the numbers X and Y. We define a function add (*X, Y, Z*) which is similar to sum (*X, Y, Z*), except that the function *add* first verifies that each of X and Y is a number.
Then add can be written as

```
add (X, Y, Z):- number (X), number (Y), sum (X, Y, Z).
add (X, Y, Z):- fail.
```

Cut & fail

Example: Suppose, we want to write a program to calculate *tax payable by a person* for a given income. However, according to the law of a country, a foreigner is not required to pay tax. *Then one of the possible (incorrect) programs may be written as:*

```
tax (Name, Income, Tax):- foreigner(Name),fail.
tax (Name, Income, Tax):-…….
tax (Name, Income, Tax):-…….
.
.
.
```

Except for the first one, all others rules are left unspecified. However, these other rules do not involve the predicate *foreigner* but may depend upon income, concession/rebate etc.

Now suppose *Alberts* is a foreigner. Then According to the first rule *foreigner (alberts)* succeeds and then the predicate *fail* makes the goal on L.H.S not to succeed. Hence PROLOG system backtracks and attempts the second and later rules which do not involve foreigner.

Hence, some of the later goals may succeed, despite the fact that the tax for *albert* should not be calculated. However, the error can be rectified by using *cut*. *The following programme outline gives the correct result in the case of foreigners:*

```
tax (Name, Income, Tax):- foreigner (Name), !, fail.
tax (Name, Income, Tax):-…….
tax (Name, Income, Tax):-…….
.
.
.
```

Because of the presence of *cut* in the first rule before *fail*, if *foreigner (name)* succeeds, no backtracking takes place after returning to cut from the execution of fail. Hence, If *foreigner (name)* succeeds, the program execution stops after returning back to '!'.
However, if *foreigner (Name)* fails, then processing by PROLOG system continues from the second rule onwards, according to normal rules of PROLOG execution.

2.12 SUMMARY

The programming language PROLOG is based on *Declarative paradigm* and *Logic Programming* paradigm of solving problems. These paradigms are quite different from the normally used paradigm viz *imperative paradigm* of solving problems. The issues about these paradigms are discuss in the *Introduction* to the unit, i.e., in Section 2.0.

In Section 2.2 the basic concepts of PROLOG, including those of *atom, predicate, variable, atomic formula, goal and fact etc*. are discussed and defined in English. The syntax for various elements and constructs of PROLOG including for the above-mentioned concepts and also for the concepts of *structure, term, rule, query, goal, subgoal and program etc* are (formally) defined in Section 2.3.

A problem in PROLOG is defined by
(i) defining a database and
(ii) a query representing the goal to be achieved.

The concept of *database* is defined in Section 2.4. However, Section 2.4 is mainly devoted to explaining how a PROLOG system solves a problem where the problem is defined to the system in the way discussed above. In the process of explanation, important concepts like **matching** and **unification** are introduced and defined.

Alongwith matching and unification, another mechanism viz *backtrakcing* play a significant role in problem solving by PROLOG system. The concept of **backtracking** is explained in Section 2.5.

We know built-in data types and data structures play an important role in any *programming environment* which is generally based on some problem solving style/paradigm including the *imperative style* which is the dominant style of solving problems using computer. Data types and data structures play a significant role in problem solving in PROLOG environment also and are discussed in Section 2.6. Prolog has basically two (elementary) data types viz **atom** and **number**. Also, PROLOG has mainly two data structures viz ***List*** and ***Structure***.

The operations available in PROLOG, for data structure **List** are discussed in Section 2.7. The predicate *equal (denoted as '=')* which plays an important role in PROLOG, specially in matching and unification, is discussed in Section 2.8.

PROLOG is mainly a *symbol processing language*. But, in view of the fact that even for the problems that require for their solutions dominantly symbol processing, some numeric processing may also be required. For this purpose, the arithmetic facilities built in PROLOG are discussed and defined in Section 2.9. ***Recursion*** and ***iteration*** are among major mechanism of PROLOG problem solving process. However, in order to check undesirable repetitions (through recursion or iteration) PROLOG system provides for two very important predicates viz **Cut** and **Fail**. These predicate are discussed in Sections 2.10 and 2.11.

2.13 SOLUTIONS/ANSWERS

Ex 1: The goal fails, because, after matching with either rule (xix) or rule (xx) of the query, one of the subgoals generated is *female (aslam)*. But this sub-goal can not be satisfied by the facts and other rules in the database.

Ex 2: As in the previous query, the PROLOG system while attempting to match and then rejecting the previous facts and rules, reaches rule (xix). And, as in the previous query, requires to satisfy two subgoals, which after appropriate unification, become

parents(jane, M, F).
parents (john, M, F).

But, the first of these fails and hence the whole of rule (xix) itself is taken as not matching and is abandoned for the next rule/fact in the database. The next rule (xx)

```
is_sister (X, Y):- female (X), mother (X, M), father (X, F)
                   mother (Y, M), father (Y, F).
```

has functor *is_sister* in the goal and hence matches the functor of the goal. As explained earlier, matching becomes the exercise of unification of the goal, i.e., of

is_sister (jane, john) *with*
is_sister (X, Y), which is the goal of the rule (xx).

Through unification, variable X is temporarily associated *throughout rule (xx)* with constant *jane*, and variable Y with constant *john* throughout rule (xx).
In the process, R.H.S. of rule (xx) generates the following five (new) subgoals viz

```
female (jane).
mother (jane, M).
mother (john, M).
father (jane, F).
father (john, F).
```

First of these subgoals is easily satisfied by the Fact (iv) of the database. **Second** of these subgoals viz *mother (jane, M)* is a sort of question of the form: *Who is jane's mother?*.

Fact (xvi) in the database tells us that *mary* is mother of *jane*, and hence, the variable M temporarily (*for the whole of rule (xx))* gets associated with the constant *mary*. Thus, the **next subgoal** '*mother (john, M).*' becomes the goal '*mother (john, mary).*' which is given as Fact (xvii) and hence is satisfied.

Fourth subgoal *father (jane, F)* is equivalent to the question: '*Who is janes Father?*' The Fact (xiii) in the database associates to variable F the constant *albert* for the whole of rule (xx). In the light of this association, the last subgoal becomes *father (john, albert)*. But this goal is given as Fact (xiv) in the database and hence is satisfied. **The PROLOG system answers the query as 'yes'.**

Ex 3: The PROLOG system proceeds as in previous query and generates five subgoals, viz,

female (jane).
mother (jane, M).
mother (phillips, M)
father (jane, F).
father (phillips, F).

As, in the case of previous query, while satisfying subgoal *mother (jane, M)*, the variable M is associated with *mary* for all the subgoals. Hence the subgoal *mother (phillips, M)* becomes *mother (phillips, mary)*. But there is no fact that matches the last subgoal. Hence the query fails.
The PROLOG system respeonds with 'No'.

Ex 4: *book ((title computer_networks), (edition fourth),*
(author (first_name Andrew),
(middle_initial S), (last_name tanenbaum)
)
(year 2003), (publisher prentice_hall_ptr)).

For the following query:

?_book ((tittle computer_networks), (edition fourth), X, (year 2003), (publisher prentice_hall_ptr)).
The system returns the name of the author as *(author ((first_name Andrew), (middle_initial, (last_name tanenbaum)).*

Ex 5: As discussed in the preceding Example 3, the PROLOG system takes similar steps upto Step 4 above, except the constant *prolog* does not occur in the list under [– | Y]. From step 5 onwards the procedure adopted by PROLOG system is as follows

Goal	**X**	**[– | Y]**	**Comment**
5.	pascal	[cobol |]	*fact (i) not satisfied. Hence apply rule (ii)*
6.	pascal	[ ]	*cannot be satisfied. The system says the goal cannot be satisfied.*

Ex 6: *The PROLOG system processes the query as follows:*
Using the only rule for prefix, the system instantiates variables to get

prefix ([a, c], [a, b, c]):- append ([a, c], Y, [a, b, c]).

,i.e., in order to satisfy the query, the system has a new goal viz

append ([a, c], Y, [a, b, c]).

for which the value of Y, if it exists, is to be found. If no such value exists, then the system returns: No

Next the rule in the recursive definition of append gives

Append ([a|c], Y, [a|b, c]):- append ([c], Y, [b, c])

which when written as ([c|], Y, [b|c]) and when matched with append ([Head|Tail], Y, and [Head|Z] is not able to associate any value to Head. Hence PROLOG system returns: No

Ex 7: The suffix PROLOG program is just one statement program:-

suffix (Y, Z):- append (X, Y, Z).
where we have already defined append as
append ([ ], X , X]. (i)
append ([Head|Tail], Y, [Head|Z]):- append (Tail, Y, Z) (ii)

Explanation The only statement in the program, states that the statement: '*The list Y is a suffix of list Z'* is true if there is a list X to which if Y is appended then the list Z is obtained.

The process generated by the program is explained through the following examples:

Example query

?-suffix (Y, [a, b, c]).

Then through the rule in the definition, we get

suffix (Y, [a, b, c]):-append (X, Y, [a, b, c]).

Applying rule (ii) *of append,* we get the subgoal given on R.H.S. above is satisfied by associating X to [ ], Y to [a, b, c].

Therefore the system returns yes with

Y = [a, b, c]

If the user desires another answer by a typing ';' then the system applies rule (ii) *of append* to

Append [(Head|Tail), Y, [a|b, c]):-append (Tail, Y, [b, c]).

and 'a' is associated to Head.

Again to satisfy the new goal given on R.H.S above, we get through the application of rule (ii) *of append,* the system associates [b, c] to Y and returns

Y = [b, c]

Similarly, through another iteration, Y = [c] is returned.

Through still another iteration, Y = [ ] is returned, And, finally, if another iteration is desired by the user, then 'No' is returned.

Ex 8: (i) succeeds, because, first of all the two variables X and Z become co-referred variables. Then Z gets instantiated to c and hence X also gets instantiated to c

(ii) fails, because, the two predicates/functors do not equal.

(iii) also fails, because the variable X gets instantiated to the constant **a** but then system can not instantiate the second occurrence of X to some other constant, in this particular case, to the constant **b**.

(iv) succeeds because the R.H.S. is a variable. The variable Noun is associated to the constant noun (alpha)

(v) fails, both sides of '=' are constants but not identical:

2.14 FURTHER READINGS

1. Clocksin, W.F. & Mellish, C.S. : *Programming in Prolog (Fifth Edition)*, Springer (2003).
2. Clocksin, W.F. & Mellish, C.S. : *Programming in Prolog (Third Edition),* Narosa Publishing House (1981).
3. Tucker, A. & Noonan, R. : *Programming Languages: Principles and Paradigms*, Tata McGraw-Hill Publishing Company Limited (2002).
4. Sebesta, R. W. : *Concepts of Programming Languages*, Pearson Education Asia (2002).