
UNIT 1 EXPERT SYSTEMS

Structure	Page Nos.
1.0 Introduction	5
1.1 Objectives	5
1.2 An Introduction to Expert Systems	6
1.3 Concept of Planning, Representing and using Domain Knowledge	7
1.4 Knowledge Representation Schemes	7
1.4.1 Semantic Networks	
1.4.2 Frames	
1.4.3 Proposition and Predicate Logic	
1.4.4 Rule Based Systems	
1.4.4.1 Forward Chaining Systems	
1.4.4.2 Backward Chaining Systems	
1.4.4.3 Probability Certainty Factors in Rule Based Systems	
1.5 Examples of Expert Systems: MYCIN, COMPASS	24
1.6 Expert System Building Tools	25
1.6.1 Expert System Shells	
1.6.1.1 Knowledge Base	
1.6.1.2 Knowledge Acquisition Subsystem (Example: COMPASS)	
1.6.1.3 Inference Engine	
1.6.1.4 Explanation Sub-system (Example: MYCIN)	
1.6.1.5 User Interface	
1.6.1.6 An Expert System shell: EMYCIN	
1.7 Some Application Areas of Expert Systems	32
1.8 Summary	32
1.9 Solutions/Answers	33
1.10 Further Readings	35

1.0 INTRODUCTION

Computer Science is the study of how to create models that can be represented in and executed by some computing equipment. A number of new types of problems are being solved almost everyday by developing relevant models of the domains of the problems under consideration. One of the major problems which humanity encounters is in respect of scarcity of human experts to handle problems from various domains of human experience relating to domains including domains those of health, education, economic welfare, natural resources and the environment. In this respect, the task for a computer scientist is to create, in addition to a model of the problem domain, a model of an expert of the domain as problem solver who is highly skilled in solving problems from the domain under consideration. The field of Expert Systems is concerned with creating such models. The task includes activities related to eliciting information from the experts in the domain in respect of how they solve the problems, and activities related to codifying that information generally in the form of rules. This unit discusses such issues about the design and development of an expert system.

1.1 OBJECTIVES

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

- discuss the various knowledge representation scheme;
- tell about some of the well-known expert systems;
- enumerate and use of various tools for building expert systems;
- explain the various expert system shells, and
- discuss various application areas for expert systems.

First of all we must understand that an expert system is nothing but a computer program or a set of computer programs which contains the knowledge and some inference capability of an expert, most generally a human expert, in a particular domain. As expert system is supposed to contain the capability to lead to some conclusion based on the inputs provided, information it already contains and its processing capability, an expert system belongs to the branch of Computer Science called Artificial Intelligence.

Mere possessing an algorithm for solving a problem is not sufficient for a program to be termed an expert system, **it must also possess knowledge** i.e., if there is an expert system for a particular domain or area and if it is fed with a number of questions regarding that domain then sooner or later we can expect that these questions will be answered. So we can say that the knowledge contained by an expert system must contribute towards solving the problems for which it has been designed.

Also knowledge in a expert system must be regarding a **specific domain**. As a human being cannot be an expert in every area of life, similarly, an expert system which tries to simulate the capabilities of an expert also works in a particular domain. Otherwise it may be require to possess potentially infinite amount of knowledge and processing that knowledge in finite amount of time is an impossible task.

Taking into consideration all the points which have been discussed above, let us try to give one of the many possible definitions of an Expert System:

An Expert System is a computer program that possesses or represents knowledge in a particular domain, has the capability of processing/ manipulating or reasoning with this knowledge with a view to solving a problem, giving some achieving or to achieve some specific goal.

An expert system may or may not provide the complete expertise or functionality of a human expert but it must be able to assist a human expert in fast decision making. The program might interact with a human expert or with a customer directly.

Let us discuss some of the basic properties of an expert system:

- It tries to simulate human reasoning capability about a specific domain rather than the domain itself. This feature separates expert systems from some other familiar programs that use mathematical modeling or computer animation. In an expert system the focus is to emulate an expert's knowledge and problem solving capabilities and if possible, at a faster rate than a human expert.
- It perform reasoning over the acquired knowledge, rather than merely performing some calculations or performing data retrieval.
- It can solve problems by using heuristic or approximate models which, unlike other algorithmic solutions are not guaranteed to succeed.

AI programs that achieve expert-level competence in solving problems in different domains are more called **knowledge based systems**. A **knowledge-based system** is any system which performs a job or task by applying rules of thumb to a symbolic representation of knowledge, instead of employing mostly algorithmic or statistical methods. Often the term *expert systems* is reserved for programs whose knowledge base contains the knowledge used by human experts, in contrast to knowledge gathered from textbooks or non-experts. **But more often than not, the two terms, expert systems and knowledge-based systems are taken us synonyms.** Together

they represent the most widespread type of *AI* application. The area of human intellectual endeavour to be captured in an expert system is sometimes called the task domain. **Task** refers to some goal-oriented, problem-solving activity. **Domain** refers to the area within which the task is being performed. Some of the typical tasks are diagnosis, planning, scheduling, configuration and design. For example, a program capable of conversing about the weather would be a knowledge-based system, even if that program did not have any expertise in meteorology, but an expert system must be able to perform weather forecasting.

Building a expert system is known as **knowledge engineering** and its practitioners are called **knowledge engineers**. It is the job of the knowledge engineer to ensure to make sure that the computer has all the knowledge needed to solve a problem. **The knowledge engineer must choose** one or more forms in which to represent the required knowledge i.e., s/he must choose **one or more knowledge representation schemes** (A number of knowledge representing schemes like semantic nets, frames, predicate logic and rule based systems have been discussed above. We would be sticking to rule based representation scheme for our discussion on expert systems). S/he must also ensure that the computer can use the knowledge efficiently by selecting from a handful of reasoning methods.

1.3 CONCEPT OF PLANNING, REPRESENTING AND USING DOMAIN KNOWLEDGE

From our everyday experience, we know that in order to solve difficult problems, we need to do some sort of planning. Informally, we can say that **Planning** is the process that exploits the structure of the problem under consideration for designing a sequence of actions in order to solve the problem under consideration.

The knowledge of nature and structure of the problem domain is essential for planning a solution of the problem under consideration. For the purpose of planning, the problem environments are divided into two categories, viz., classical planning environments and non-classical planning environments. The **classical** planning environments/domains are fully observable, deterministic, finite, static and discrete. On the other hand, **non-classical** planning environments may be only partially observable and/or stochastic. In this unit, we discuss planning only for classical environments.

1.4 KNOWLEDGE REPRESENTATION SCHEMES

One of the underlying assumptions in Artificial Intelligence is that **intelligent behaviour can be achieved through the manipulation of symbol structures** (representing bits of knowledge). One of the main issues in *AI* is to find appropriate representation of problem elements and available actions as symbol structures so that the representation can be used to intelligently solve problems. In *AI*, **an important criteria about knowledge representation schemes or languages is that they should support inference**. For intelligent action, the inferencing capability is essential in view of the fact that we can't represent explicitly everything that the system might ever need to know—**some things have to be left implicit, to be inferred/deduced by the system** as and when needed in problem solving.

In general, a good knowledge representation scheme should have the following features:

- It should allow us to express the knowledge we wish to represent in the language. For example, the mathematical statement: *Every symmetric and transitive relation on a domain, need not be reflexive* is not expressible in First Order Logic.
- It should allow new knowledge to be inferred from a basic set of facts, as discussed above.
- It should have well-defined syntax and semantics.

Some popular knowledge representation schemes are:

- **Semantic networks,**
- **Frames,**
- **First order logic, and,**
- **Rule-based systems.**

As semantic networks, frames and predicate logic have been discussed in previous blocks so we will discuss these briefly here. We will discuss the rule-based systems in detail.

1.4.1 Semantic Networks

Semantic Network representations provide a **structured knowledge representation**. In such a network, parts of knowledge are clustered into semantic groups. In semantic networks, the concepts and entities/objects of the problem domain are represented by nodes and relationships between these entities are shown by arrows, generally, by directed arrows. In view of the fact that semantic network **representation is a pictorial depiction** of objects, their attributes and the relationships that exist between these objects and other entities. A semantic net is just a graph, where the nodes in the graph represent concepts, and the arcs are labeled and represent binary relationships between concepts. These networks provide a more natural way, as compared to other representation schemes, for mapping to and from a natural language.

For example, the fact (a piece of knowledge): ***Mohan struck Nita in the garden with a sharp knife last week***, is represented by the semantic network shown in Figure 4.1.

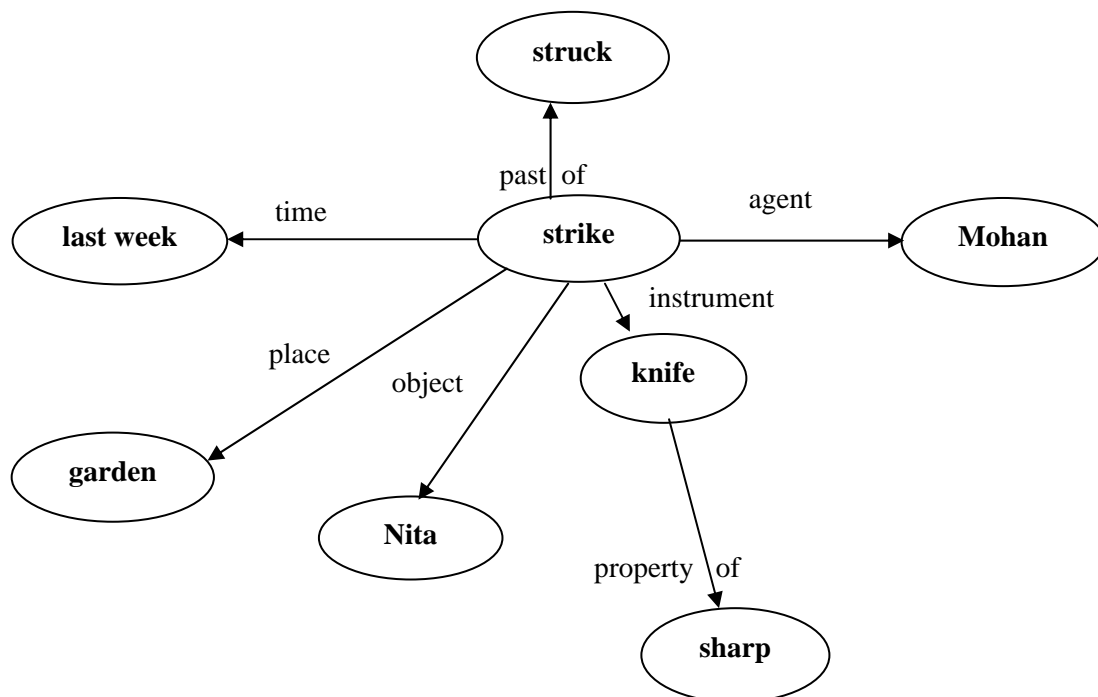


Figure 1.1 Semantic Network of the Fact in Bold

The two most important relations between concepts are (i) *subclass* relation between a class and its superclass, and (ii) *instance* relation between an object and its class. Other relations may be *has-part*, *color* etc. As mentioned earlier, relations are indicated by labeled arcs.

As information in semantic networks is clustered together through relational links, the knowledge required for the performance of some task is generally available within short spatial span of the semantic network. This type of knowledge organisation in some way, resembles the way knowledge is stored and retrieved by human beings.

Subclass and *instance* relations allow us to use **inheritance** to infer new facts/relations from the explicitly represented ones. We have already mentioned that **the graphical portrayal of knowledge in semantic networks**, being visual, is easier than other representation schemes for the human beings to comprehend. This fact helps the human beings to guide the expert system, whenever required. This is perhaps the reason for the popularity of semantic networks.

Exercise 1: Draw a semantic network for the following English statement:
Mohan struck Nita and Nita's mother struck Mohan.

1.4.2 Frames

Frames are a variant of semantic networks that are one of the popular ways of representing non-procedural knowledge in an expert system. In a frame, all the information relevant to a particular concept is stored in a single complex entity, called a frame. Frames look like the data structure, record. Frames support inheritance. They are often used to capture knowledge about *typical* objects or events, such as a car, or even a mathematical object like rectangle. As mentioned earlier, a frame is a structured object and different names like *Schema*, *Script*, *Prototype*, and even *Object* are used in stead of frame, in computer science literature.

We may represent some knowledge about a lion in frames as follows:

Mammal :

Subclass : Animal
warm_blooded : yes

Lion :

subclass : Mammal
eating-habbit : carnivorous
size : medium

Raja :

instance : Lion
colour : dull-Yellow
owner : Amar Circus

Sheru :

instance : Lion
size : small

A particular frame (such as Lion) has a number of *attributes* or *slots* such as *eating-habit* and *size*. Each of these slots may be filled with particular values, such as the *eating-habit* for lion may be filled up as *carnivorous*.

Sometimes a slot contains additional information such as how to apply or use the slot values. Typically, a slot contains information such as (*attribute*, *value*) pairs, default values, conditions for filling a **slot**, pointers to other related frames, and also procedures that are activated when needed for different purposes.

In the case of frame representation of knowledge, **inheritance is simple** if an object has a single parent class, and if each slot takes a single value. For example, if a mammal is warm blooded then automatically a lion being a mammal will also be warm blooded.

But **in case of multiple inheritance** i.e., in case of an object having more than one parent class, we have to decide which parent to inherit from. For example, a lion may inherit from “wild animals” or “circus animals”. In general, both the slots and slot values may themselves be frames and so on.

Frame systems are pretty complex and sophisticated knowledge representation tools. This representation has become so popular that special high level frame based representation languages have been developed. Most of these languages use LISP as the host language. It is also possible to represent frame-like structures using object oriented programming languages, extensions to the programming language LISP.

Exercise 2: Define a frame for the entity *date* which consists of *day*, *month* and *year*. each of which is a number with restrictions which are well-known. Also a procedure named *compute-day-of-week* is already defined.

1.4.3 Proposition and Predicate Logic

Symbolic logic may be thought of as a formal language, which incorporates a precise system for reasoning and deduction. Propositional logic and predicate logic are two well-known forms of symbolic logic. Propositional logic deals with propositions or statements.

A proposition or a statement is a sentence, which can be assigned a truth-value *true* or *false*. For example, the statement:

The sun rises in the west, has a truth value *false*.

On the other hand, none of the following sentences can be assigned a truth-value, and hence none of these, is a statement or proposition:

- (i) *Ram must exercise regularly.* (Imperative sentence)
- (ii) *Who was the first Prime Minister of India?* (Interrogative sentence)
- (iii) *Please, give me that book.* (Imperative sentence)
- (iv) *Hurry! We have won the trophy.* (Exclamatory sentence).

Generally, the following **steps are followed for solving problems using propositional logic**, where the problems are expressed in English and are such that these can be solved using propositional logic (**PL**):

- a) First problem statements in English are translated to formulae of Propositional logic

- b) Next, some of the rules of inference in PL including the ones mentioned below, are used to solve the problem, if, at all, the problem under consideration is solvable by PL.

Some of the rules of Inference of Propositional Logic:

$$\begin{array}{l} \text{(i) Modus Ponens} \quad P \\ \quad \quad \quad \quad \quad P \rightarrow Q \\ \hline \quad \quad \quad \quad \quad Q \end{array}$$

(The above notation is interpreted as: if we are given the two formulae P and $P \rightarrow Q$ of the propositional logic, then conclude the formula Q . In other words, if both the formulae P and $P \rightarrow Q$ are assumed to be true, then by modus ponens, we can assume the statement Q also as true.)

$$\begin{array}{l} \text{(ii) Chain Rule} \\ \quad \quad \quad \quad \quad P \rightarrow Q \\ \quad \quad \quad \quad \quad Q \rightarrow R \\ \hline \quad \quad \quad \quad \quad P \rightarrow R \end{array}$$

$$\begin{array}{l} \text{(iii) Transposition} \\ \quad \quad \quad \quad \quad P \rightarrow Q \\ \hline \quad \quad \quad \quad \quad \sim Q \rightarrow \sim P \end{array}$$

Example

Given the following three statements:

- (i) Matter always existed
- (ii) If there is God, then God created the universe
- (iii) If God created the universe, then matter did not always exist.

To show the truth of the statement: *There is no God.*

Solution:

Let us denote the atomic statements in the argument given above as follows:

M: Matter always existed

TG: There is God

GU: God created the universe.

Then **the given** statements in English, become respectively the formulae of PL:

- (i) M
- (ii) $TG \rightarrow GU$
- (iii) $GU \rightarrow \sim M$
- (iv) **To show** $\sim TG$

Applying transposition to (iii) we get

- (v) $M \rightarrow \sim GU$

using (i) and (v) and applying Modus Ponens, we get

- (vi) $\sim GU$

Again, applying transposition to (ii) we get

- (vii) $\sim GU \rightarrow \sim TG$

Applying Modus Ponens to (vi) and (vii) we get

- (viii) $\sim TG$

The formula (viii) is the same as formula (iv) which was required to be proved.

Exercise 3: Using prepositional logic, show that, if the following statements are assumed to be true:

- (i) *There is a moral law.*
- (ii) *If there is a moral law, then someone gave it.*

- (iii) *If someone gave the moral law, then there is God.*
 then the following statement is also true:
 (iv) *There is God.*

The trouble with propositional logic is that it is unable to describe properties of objects and also it lacks the structure to express relations that exist among two or more entities. Further, propositional logic does not allow us to make generalised statements about classes of similar objects. However, in many situations, the explicit knowledge of relations between objects and generalised statements about classes of similar objects, are essentially required to solve problems. Thus, propositional logic has serious limitations while reasoning for solving real-world problems. For example, let us look at the following statements:

- (i) *All children more than 12 years old must exercise regularly.*
 (ii) *Ram is more than 12 years old.*

Now 'these statements should be sufficient enough to allow us to conclude: *Ram must exercise regularly.* However, in propositional logic, each of the above statement is indecomposable and may be respectively denoted by P and Q. Further, whatever is said inside each statement is presumed to be not visible. Therefore, if we use the language of propositional logic, we are just given two symbols, viz., P and Q, representing respectively the two given statements. However, from just two propositional formulae P and Q, it is not possible to conclude the above mentioned statement viz., *Ram must exercise regularly.* To draw the desired conclusion with a valid inference rule, it would be necessary to use some other language, including some extension of propositional logic.

Predicate Logic, and more specifically, **First Order Predicate Logic (FOPL)** is an extension of propositional logic, which was developed to extend the expressiveness of propositional logic. In addition to just propositions of propositional logic, the predicate logic uses predicates, functions, and variables together with variable quantifiers (*Universal and Existential quantifiers*) to express knowledge.

We already have defined the structure of formulae of FOPL and also have explained the procedure for finding the meaning of formulae in FOPL. Though, we have already explained how to solve problems using FOPL, yet just for recalling the procedure for solving problems using FOPL, we will consider below one example.

In this context, we may recall the **inference rules of FOPL**. The inference rules of PL including *Modus Ponens*, *Chain Rule* and *Rule of Transposition* are valid in FOPL also after suitable modifications by which formulae of PL are replaced by formulae of FOPL.

In addition to these inference rules, the **following four inference rules of FOPL**, that will be called Q₁, Q₂, Q₃ and Q₄, have no corresponding rules in PL. *In the following F denotes a predicate and x a variable/parameter:*

$$Q_1: \frac{\sim (\exists x)F(x)}{(\forall x) \sim F(x)} \quad \text{and} \quad \frac{(\forall x) \sim F(x)}{\sim (\exists x)F(x)}$$

The first of the above rules under Q₁, says:

From *negation of there exists x F (x)*, we can infer *for all x not of F (x)*

$$Q_2: \frac{\sim (\forall x)F(x)}{(\exists x) \sim F(x)} \quad \text{and} \quad \frac{(\exists x) \sim F(x)}{\sim (\forall x)F(x)}$$

The first of the above rules under Q_2 , says:

From *negation of for all x $F(x)$* , we can infer *there exists x such that not of $F(x)$*

$$Q_3: \frac{(\forall x)F(x)}{F(a)}, \text{ where } a \text{ is (any) arbitrary element of the domain of } F$$

The rule Q_3 is called **universal instantiation**

$$Q_3^|: \frac{F(a), \text{ for arbitrary } a}{(\forall x)F(x)}$$

The rule is called **universal generalisation**

$$Q_4: \frac{(\exists x)F(x)}{F(a)}, \text{ where } a \text{ is a particular (not arbitrary) constant.}$$

This rule is also called **existential instantiation**:

$$Q_4^|: \frac{F(a) \text{ for some } a}{(\exists x)F(x)}$$

The rule is called **existential generalisation**

Steps for using Predicate Calculus as a Language for Representing Knowledge

Step 1: Conceptualization: First of all, all the relevant entities and the relations that exist between these entities are explicitly enumerated. Some of the implicit facts like ‘a person dead once is dead forever’ have to be explicated.

Step 2: Nomenclature and Translation: Giving appropriate names to the objects and relations. And then translating the given sentences given in English to formulae in FOPL.

Appropriate names are essential in order to guide a reasoning system based on FOPL. It is well-established that no reasoning system is complete. In other words, a reasoning system may need help in arriving at desired conclusion.

Step 3: Finding appropriate sequence of reasoning steps, involving selection of appropriate rule and appropriate FOPL formulae to which the selected rule is to be applied, to reach the conclusion.

While solving problems with FOPL, generally, the proof technique is proof by contradiction. Under this technique, the negation of what is to be proved is also taken as one of the assumptions. Then from the given assumptions alongwith the new assumption, we derive a contradiction, i.e., using inference rules, we derive a statement which is negation of either an assumption or is negation of some earlier derived formula.

Next, we give an example to illustrate how FOPL can be used to solve problems expressed in English and which are solvable by using FOPL. However, the proposed solution does not use the above-mentioned method of contradiction.

We are given the statements:

- (i) *No feeling of pain is publically observable*
 - (ii) *All chemical processes are publically observable*
- We are to prove that
- (iii) *No feeling of pain is a chemical process*

Solution:

For translating the given statements (i), (ii) & (iii), let us use the notation:

$F(x)$: x is an instance of feeling of pain
 $O(x)$: x is an entity that is publically observable
 $C(x)$: x is a chemical process.

Then (i), (ii) and (iii) in FOPL can be equivalently expressed as

- (i) $(\forall x) (F(x) \rightarrow \sim O(x))$
- (ii) $(\forall x) (C(x) \rightarrow O(x))$

To prove

- (iii) $(\forall x) (F(x) \rightarrow \sim C(x))$

From (i) using generalized instantiation, we get

- (iv) $F(a) \rightarrow \sim O(a)$ for any arbitrary a .

Similarly, from (ii), using generalized instantiation, we get

- (v) $C(b) \rightarrow O(b)$ for any arbitrary b .

From (iv) using transposition rule, we get

- (vi) $O(a) \rightarrow \sim F(a)$ for any arbitrary a

As b is arbitrary in (v), therefore we can rewrite (v) as

- (vii) $C(a) \rightarrow O(a)$ for any arbitrary a

From (vii) and (vi) and using chain rule, we get

- (viii) $C(a) \rightarrow \sim F(a)$ for any arbitrary a

But as a is arbitrary in (viii), by generalized quantification, we get

- (ix) $(\forall x) (C(x) \rightarrow \sim F(x))$

But (ix) is the same as (iii), which was required to be proved.

Problems with FOPL as a system of knowledge representation and reasoning:

FOPL is not capable of easily representing some kinds of information including information pieces involving

- (i) properties of relations. For example, the mathematical statement:
Any relation which is symmetric & transitive may not be reflexive
is not expressible in FOPL.
- (ii) linguistic variables like hot, tall, sweat.
For example: *It is very cold today*,
can not be appropriately expressed in FOPL.
- (iii) different belief systems.
For example, *s/he know that he thinks India will win the match, but I think India will lose*,
also, can not be appropriately expressed in FOPL.

Some shortcomings in predicate calculus

Now, after having studied predicate logic as a knowledge representation scheme, we ask ourselves the inevitable question, **whether it can be used to solve real world problems and to what extent.**

Before we try to answer the above question, let us review some of the properties of logic reasoning systems including predicate calculus. We must remember that three important properties of any logical reasoning system are *soundness*, *completeness* and *tractability*. To be confident that an inferred conclusion is true we require soundness. To be confident that inference will eventually produce any true conclusion, we require completeness. To be confident that inference is feasible, we require tractability.

Now, as we have discussed above also, in predicate calculus resolution refutation as an inference procedure is **sound** and **complete**. Because in case that the well formed formula which is to be proved is not logically followed or entailed by the set of well formed formulas which are to be used as premises, **the resolution refutation procedure might never terminate**. Thus, resolution refutation is not a full decision procedure. Also there is not other procedure that has this property or that is fully decidable. **So predicate calculus is semi-decidable** (semi-decidable means that if the set of input (well formed formulas) do not lead to the conclusion then the procedure will never stop or terminate).

But the situation is worse than this, as even on problems for which resolution refutation terminates, the procedure is **NP-hard** – as is any sound and complete inference procedure for the first-order predicate calculus i.e., it may take exponentially large time to reach a conclusion.

How to tackle these problems:

People who have done research in Artificial Intelligence have shown various ways: **First**, they say that we should not insist on the property of soundness of inference rules. Now what does it mean – basically it means that sometimes or occasionally our rules might prove an “untrue formula”.

Second, they say that we should not insist on the property of completeness i.e., to allow use of procedures that are not guaranteed to find proofs of true formulas.

Third, they also suggest that we could use a language that is less expressive than the predicate calculus. For example, a language in which everything is expressed using only Horn Clauses (Horn clauses are those which have at most one positive literal).

1.4.4 Rule Based Systems

Rather than representing knowledge in a declarative and somewhat static way (as a set of statements, each of which is true), rule-based systems represent knowledge in terms of a set of rules each of which specifies the conclusion that could be reached or derived under given conditions or in different situations. A rule-based system consists of

- (i) Rule base, which is a set of IF-THEN *rules*,
- (ii) A bunch of *facts*, and
- (iii) Some interpreter of the facts and rules which is a mechanism which decides which rule to apply based on the set of available facts. The interpreter also initiates the action suggested by the rule selected for application.

A Rule-base may be of the form:

R₁: If A is an animal and A barks, then A is a dog

F1: Rocky is an animal

F2: Rocky Barks

The rule-interpreter, after scanning the above rule-base may conclude: Rocky is a dog. After this interpretation, the rule-base becomes

R₁: If A is an animal and A barks, then A is a dog

F1: Rocky is an animal

F2: Rocky Barks

F3: Rocky is a dog.

There are two broad kinds of rule-based systems:

forward chaining systems,
and *backward chaining systems.*

In a **forward** chaining system we start with the initial facts, and keep using the rules to draw new intermediate conclusions (or take certain actions) given those facts. The process terminates when the final conclusion is established. In a **backward** chaining system, we start with some goal statements, which are intended to be established and keep looking for rules that would allow us to conclude, setting new sub-goals in the process of reaching the ultimate goal. In the next round, the subgoals become the new goals to be established. The process terminates when in this process all the subgoals are given fact. Forward chaining systems are primarily **data-driven**, while backward chaining systems are **goal-driven**. We will discuss each in detail.

Next, we discuss in detail some of the issues involved in a rule-based system.

Advantages of Rule-base

A basic principle of rule-based system is that each rule is an independent piece of knowledge. In an IF-THEN rule, the IF-part contains all the conditions for the application of the rule under consideration. THEN-part tells the action to be taken by the interpreter. The interpreter need not search anywhere else except within the rule itself for the conditions required for application of the rule.

Another important consequence of the above-mentioned characteristic of a rule-based system is that no rule can call upon any other and hence rules are ignorant and hence independent, of each other. This gives a highly modular structure to the rule-based systems. Because of the highly modular structure of the rule-base, the rule-based system addition, deletion and modification of a rule can be done without any danger side effects.

Disadvantages

The main problem with the rule-based systems is that when the rule-base grows and becomes very large, then checking (i) whether a new rule intended to be added is redundant, i.e., it is already covered by some of the earlier rules. Still worse, as the rule-base grows, checking the consistency of the rule-base also becomes quite difficult. By consistency, we mean there may be two rules having similar conditions, the actions by the two rules conflict with each other.

Let us first define working memory, before we study forward and backward chaining systems.

Working Memory: A working is a representation, in which

- lexically, there are application –specific symbols.
- structurally, assertions are list of application-specific symbols.
- semantically, assertions denote facts.
- assertions can be added or removed from working memory.

1.4.4.1 Forward Chaining Systems

In a forward chaining system the facts in the system are represented in a **working memory** which is continually updated, so on the basis of a rule which is currently being applied, the number of facts may either increase or decrease. Rules in the system represent possible actions to be taken when specified conditions hold on items in the working memory—they are sometimes called **condition-action or antecedent-consequent rules**. The conditions are usually **patterns** that must match items in the working memory, while the actions usually involve **adding or deleting** items from the working memory. So we can say that in forward chaining proceeds forward, beginning with facts, chaining through rules, and finally establishing the goal. Forward chaining systems usually represent rules in standard implicational form, with an antecedent or condition part consisting of positive literals, and a consequent or conclusion part consisting of a positive literal.

The interpreter controls the application of the rules, given the working memory, thus controlling the system's activity. It is based on a cycle of activity sometimes known as a **recognize-act** cycle. The system first checks to find all the rules whose condition parts are satisfied i.e., the those rules which are applicable, given the current state of working memory (**A rule is applicable if** each of the literals in its antecedent i.e., the condition part can be unified with a corresponding fact using consistent substitutions. This restricted form of unification is called pattern matching). It then selects one and performs the actions in the action part of the rule which may involve addition or deleting of facts. The actions will result in a new i.e., updated working memory, and the cycle starts again (**When more than one rule is applicable**, then some sort of external conflict resolution scheme is used to decide which rule will be applied. But when there are a large numbers of rules and facts then the number of unifications that must be tried becomes prohibitive or difficult). This cycle will be repeated until either there is no rule which fires, or the required goal is reached.

Rule-based systems vary greatly in their details and syntax, let us take the following example in which we use forward chaining :

Example

Let us assume that the working memory initially contains the following facts :

(day monday)

(at-home ram)

(does-not-like ram)

Let, the existing set of rules are:

R1 : IF (day monday)
THEN ADD to working memory the fact : (working-with ram)

```

R2      :                               IF      (day      monday)
      THEN ADD to working memory the fact : (talking-to ram)

```

- R3 : IF (talking-to X) AND (working-with X)
THEN ADD to working memory the fact : (busy-at-work X)
- R4 : IF (busy-at-work X) OR (at-office X)
THEN ADD to working memory the fact : (not-at-home X)
- R5 : IF (not-at-home X)
THEN DELETE from working memory the fact : (happy X)
- R6 : IF (working-with X)
THEN DELETE from working memory the fact : (does-not-like X)

Now **to start the process of inference through forward chaining**, the rule based system will first search for all the rule/s whose antecedent part/s are satisfied by the current set of facts in the working memory. For example, in this example, we can see that the rules R1 and R2 are satisfied, so the system will chose one of them using its conflict resolution strategies. Let the rule R1 is chosen. So (working-with ram) is added to the working memory (after substituting “ram” in place of X). So working memory now looks like:

(working-with ram)
(day monday)
(at-home ram)
(does-not-like ram)

Now this cycle begins again, the system looks for rules that are satisfied, it finds rule R2 and R6. Let the system chooses rule R2. So now (taking-to ram) is added to working memory. So now working memory contains following:

(talking-to ram)
(working-with ram)
(day monday)
(at-home ram)
(does-not-like ram)

Now in the next cycle, rule R3 fires, so now (busy-at-work ram) is added to working memory, which now looks like:

(busy-at-work ram)
(talking-to ram)
(working-with ram)
(day monday)
(at-home ram)
(does-not-like ram)

Now antecedent parts of rules R4 and R6 are satisfied. Let rule R4 fires, so (not-at-home, ram) is added to working memory which now looks like :

(not-at-home ram)

(busy-at-work ram)
(talking-to ram)
(working-with ram)
(day monday)
(at-home ram)
(does-not-like ram)

In the next cycle, rule R5 fires so (at-home ram) is removed from the working memory :

(not-at-home ram)
(busy-at-work ram)
(talking-to ram)
(working-with ram)
(day monday)
(does-not-like ram)

The forward chaining will continue like this. But we have to be sure of one thing, that the ordering of the rules firing is important. A change in the ordering sequence of rules firing may result in a different working memory.

Some of the conflict resolution strategies which are used to decide which rule to fire are given below:

- Don't fire a rule twice on the same data.
- Fire rules on more recent working memory elements before older ones. This allows the system to follow through a single chain of reasoning, rather than keeping on drawing new conclusions from old data.
- Fire rules with more specific preconditions before ones with more general preconditions. This allows us to deal with non-standard cases.

These strategies may help in getting reasonable behavior from a forward chaining system, but **the most important thing is how should we write the rules**. They should be carefully constructed, with the preconditions specifying as precisely as possible when different rules should fire. Otherwise we will have little idea or control of what will happen.

1.4.4.2 Backward Chaining Systems

In forward chaining systems we have seen how rule-based systems are used to draw new conclusions from existing data and then add these conclusions to a working memory. **The forward chaining approach is most useful when** we know all the initial facts, but we don't have much idea what the conclusion might be.

If we know what the conclusion would be, or have some specific hypothesis to test, forward chaining systems may be inefficient. In forward chaining we keep on moving ahead until no more rules apply or we have added our hypothesis to the working memory. But in the process the system is likely to do a lot of additional and irrelevant work, adding uninteresting or irrelevant conclusions to working memory. Let us say that in the example discussed before, suppose we want to find out whether "ram is at home". We could repeatedly fire rules, updating the working memory, checking each time whether (**at-home ram**) is found in the new working memory. But maybe we

had a whole batch of rules for drawing conclusions about what happens when I'm working, or what happens on Monday—we really don't care about this, so would rather only have to draw the conclusions that are relevant to the goal.

This can be done by **backward chaining** from the goal state or on some hypothesized state that we are interested in. This is essentially how Prolog works. Given a goal state to try and prove, for example **(at-home ram)**, the system will first check to see if the goal matches the initial facts given. If it does, then that goal succeeds. If it doesn't the system will look for rules whose conclusions i.e., actions match the goal. One such rule will be chosen, and the system will then try to prove any facts in the preconditions of the rule using the same procedure, setting these as new goals to prove. **We should note that a backward chaining system does not need to update a working memory.** Instead it needs to keep track of what goals it needs to prove its main hypothesis. So we can say that **in a backward chaining system, the reasoning proceeds “backward”, beginning with the goal to be established, chaining through rules, and finally anchoring in facts.**

Although, in principle same set of rules can be used for both forward and backward chaining. However, **in backward chaining, in practice we may choose to write the rules slightly differently.** In backward chaining we are concerned with matching the conclusion of a rule against some goal that we are trying to prove. So the 'then or consequent' part of the rule is usually not expressed as an action to take (e.g., add/delete), but as a state which will be true if the premises are true.

To learn more, let us take a different example in which we use backward chaining (The system is used to identify an animal based on its properties stored in the working memory):

Example

1. Let us assume that the working memory initially contains the following facts:

(has-hair raja)	representing	the fact “raja has hair”
(big-mouth raja)	representing	the fact “raja has a big mouth”
(long-pointed-teeth raja)	representing	the fact “raja has long pointed teeth”
(claws raja)	representing	the fact “raja has claws”

Let, the existing set of rules are:

1. IF (gives-milk X)
THEN (mammal X)
2. IF (has-hair X)
THEN (mammal X)
3. IF (mammal X) AND (eats-meat X)
THEN (carnivorous X)
4. IF (mammal X) AND (long-pointed-teeth X) AND (claws X)
THEN (carnivorous X)
5. IF (mammal X) AND (does-not-eat-meat X)
THEN (herbivorous X)
6. IF (carnivorous X) AND (dark-spots X)

THEN (cheetah, X)

7. IF (herbivorous X) AND (long-legs X) AND (long-neck X) AND (dark-spots X)
THEN (giraffe, X)
8. IF (carnivorous X) AND (big-mouth X)
THEN (lion, X)
9. IF (herbivorous X) AND (long-trunk X) AND (big-size X)
THEN (elephant, X)
10. IF (herbivorous, X) AND (white-color X) AND ((black-strips X)
THEN (zebra, X)

Now to start the process of inference through backward chaining, the rule based system will first form a hypothesis and then it will use the antecedent – consequent rules (previously called condition – action rules) to work backward toward hypothesis supporting assertions or facts.

Let us take the initial hypothesis that “raja is a lion” and then reason about whether this hypothesis is viable using backward chaining approach explained below :

- The system searches a rule, which has the initial hypothesis in the consequent part that someone i.e., raja is a lion, which it finds in rule 8.
- The system moves from consequent to antecedent part of rule 8 and it finds the first condition i.e., the first part of antecedent which says that “raja must be a carnivorous”.
- Next the system searches for a rule whose consequent part declares that someone i.e., “raja is a carnivorous”, two rules are found i.e., rule 3 and rule 4. We assume that the system tries rule 3 first.
- To satisfy the consequent part of rule 3 which now has become the system’s new hypothesis, the system moves to the first part of antecedent which says that X i.e., raja has to be mammal.
- So a new sub-goal is created in which the system has to check that “raja is a mammal”. It does so by hypothesizing it and tries to find a rule having a consequent that someone or X is a mammal. Again the system finds two rules, rule 1 and rule 2. Let us assume that the system tries rule 1 first.
- In rule 1, the system now moves to the first antecedent part which says that X i.e., raja must give milk for it to be a mammal. The system cannot tell this because this hypothesis is neither supported by any of the rules and also it is not found among the existing facts in the working memory. So the system abandons rule 1 and try to use rule 2 to establish that “raja is a mammal”.
- In rule 2, it moves to the antecedent which says that X i.e., raja must have hair for it to be a mammal. The system already knows this as it is one of the facts in working memory. So the antecedent part of rule 2 is satisfied and so the consequent that “raja is a mammal” is established.
- Now the system backtracks to the rule 3 whose first antecedent part is satisfied. In second condition of antecedent it finds its new sub-goal and in turn forms a new hypothesis that X i.e., raja eats meat.

- The system tries to find a supporting rule or an assertion in the working memory which says that “raja eats meat” but it finds none. So the system abandons the rule 3 and try to use rule 4 to establish that “raja is carnivorous”.
- In rule 4, the first part of antecedent says that raja must be a mammal for it to be carnivorous. The system already knows that “raja is a mammal” because it was already established when trying to satisfy the antecedents in rule 3.
- The system now moves to second part of antecedent in rule 4 and finds a new sub-goal in which the system must check that X i.e., raja has long-pointed-teeth which now becomes the new hypothesis. This is already established as “raja has long-pointed-teeth” is one of the assertions of the working memory.
- In third part of antecedent in rule 4 the system’s new hypothesis is that “raja has claws”. This also is already established because it is also one the assertions in the working memory.
- Now as all the parts of the antecedent in rule 4 are established so its consequent i.e., “raja is carnivorous” is established.
- The system now backtracks to rule 8 where in the second part of the antecedent says that X i.e., raja must have a big-mouth which now becomes the new hypothesis. This is already established because the system has an assertion that “raja has a big mouth”.
- Now as the whole antecedent of rule 8 is satisfied **so the system concludes that “raja is a lion”**.

We have seen that the system was able to work backward through the antecedent – consequent rules, using desired conclusions to decide that what assertions it should look for and ultimately establishing the initial hypothesis.

How to choose the type of chaining among forward or backward chaining for a given problem ?

Many of the rule based deduction systems can chain either forward or backward, but which of these approaches is better for a given problem is the point of discussion.

First, let us learn some basic things about rules i.e., **how a rule relates its input/s (i.e., facts) to output/s (i.e., conclusion)**. Whenever in a rule, a particular set of facts can lead to many conclusions, the rule is said to have a high degree of **fan out**, and a strong candidate of backward chaining for its processing. On the other hand, whenever the rules are such that a particular hypothesis can lead to many questions for the hypothesis to be established, the rule is said to have a high degree of fan in, and a high degree of **fan in** is a strong candidate of forward chaining.

To summarize, the following points should help in choosing the type of chaining for reasoning purpose :

- If the set of facts, either we already have or we may establish, can lead to a large number of conclusions or outputs , but the number of ways or input paths to reach that particular conclusion in which we are interested is small, then **the degree of fan out is more than degree of fan in. In such case, backward chaining is the preferred choice.**

- But, if the number of ways or input paths to reach the particular conclusion in which we are interested is large, but the number of conclusions that we can reach using the facts through that rule is small, then **the degree of fan in is more than the degree of fan out. In such case, forward chaining is the preferred choice.**
- For case where **the degree of fan out and fan in are approximately same**, then in case if not many facts are available and the problem is check if one of the many possible conclusions is true, **backward chaining is the preferred choice.**

1.4.4.3 Probability Certainty Factors in Rule Based System

Rule based systems usually work in domains where conclusions are rarely certain, even when we are careful enough to try and include everything we can think of in the antecedent or condition parts of rules.

Sources of Uncertainty

Two important sources of uncertainty in rule based systems are:

- ✓ The theory of the domain may be vague or incomplete so the methods to generate exact or accurate knowledge are not known.
- ✓ Case data may be imprecise or unreliable and evidence may be missing or in conflict.

So even though methods to generate exact knowledge are known but they are impractical due to lack of data, imprecision or data or problems related to data collection.

So rule based deduction system developers often build some sort of certainty or probability computing procedure on and above the normal condition-action format of rules. Certainty computing procedures attach a **probability between 0 and 1** with each assertion or fact. Each probability reflects how certain an assertion is, whereas certainty factor of 0 indicates that the assertion is definitely false and certainty factor of 1 indicates that the assertion is definitely true.

Example 1: In the example discussed above the assertion (ram at-home) may have a certainty factor, say 0.7 attached to it.

Example 2: In MYCIN a rule based expert system (which we will discuss later), a rule in which statements which link evidence to hypotheses are expressed as decision criteria, may look like :

IF patient has symptoms s1,s2,s3 and s4
AND certain background conditions t1,t2 and t3 hold
THEN the patient has disease d6 with certainty 0.75

For detailed discussion on certainty factors, the reader may refer to probability theory, fuzzy sets, possibility theory, Dempster-Shafter Theory etc.

Exercise 4

In the “Animal Identifier System” discussed above use forward chaining to try to identify the animal called “raja”.

1.5 EXAMPLES OF EXPERT SYSTEMS: MYCIN, COMPASS

The first expert system we choose as an example is MYCIN, which is of the earliest developed expert systems. As another example of an expert system we briefly discuss COMPASS.

MYCIN (An expert system)

Like every one else, we are also tempted to discuss MYCIN, one of the earliest designed expert systems in Stanford University in 1970s.

MYCIN's job was to diagnose and recommend treatment for certain blood infections. To do the proper diagnosis, it is required to grow cultures of the infecting organism which is a very time consuming process and sometime patient is in a critical state. So, doctors have to come up with quick guesses about likely problems from the available data, and use these guesses to provide a treatment where drugs are given which should deal with any type of problem.

So MYCIN was developed in order to explore how human experts make these rough (but important) guesses based on partial information. Sometimes the problem takes another shape, that an expert doctor may not be available every-time every-where, in that situation also an expert system like MYCIN would be handy.

MYCIN represented its knowledge as a set of IF-THEN rules with certainty factors. One of the MYCIN rules could be like :

IF infection is primary-bacteremia AND the site of the culture is one of the sterile sites
AND the suspected portal of entry is the gastrointestinal tract
THEN there is suggestive evidence (0.8) that bacteroid infection occurred.

The 0.8 is the certainty that the conclusion will be true given the evidence. If the evidence is uncertain the certainties of the pieces of evidence will be combined with the certainty of the rule to give the certainty of the conclusion.

MYCIN has been written in Lisp, and its rules are formally represented as Lisp expressions. The action part of the rule could just be a conclusion about the problem being solved, or it could be another Lisp expression.

MYCIN is mainly a **goal-directed system**, using the **backward chaining** reasoning approach. However, to increase its reasoning power and efficiency MYCIN also uses various heuristics to control the search for a solution.

One of the strategies used by MYCIN is **to first ask the user a number of predefined questions that are most common** and which allow the system to rule out totally unlikely diagnoses. Once these questions have been asked, the system can then focus on particular and more specific possible blood disorders. It then uses backward chaining approach to try and prove each one. This strategy avoids a lot of unnecessary search, and is similar to the way a doctor tries to diagnose a patient.

The other strategies are related to the sequence in which rules are invoked. One of the strategies is simple i.e., given a possible rule to use, MYCIN first checks all the antecedents of the rule to see if any are known to be false. If yes, then there is no point

using the rule. The other strategies are mainly related to the **certainty factors**. MYCIN first find the rules that have greater degree of certainty of conclusions, and abandons its search once the certainties involved get below a minimum threshold, say, 0.2.

There are three main stages to the interaction with MYCIN. In the first stage, initial data about the case is gathered so the system can come up with a broad diagnosis. In the second more directed questions are asked to test specific hypotheses. At the end of this section it proposes a diagnosis. In the third stage it asks questions to determine an appropriate treatment, on the basis of the diagnosis and facts related to the patient. After that it recommends some treatment. At any stage the user can ask why a question was asked or how a conclusion was reached, and if a particular treatment is recommended the user can ask if alternative treatments are possible.

MYCIN has been popular in expert system's research, but it also had a number of problems or shortcomings because of which a number of its derivatives like NEOMYCIN developed.

COMPASS (An expert system)

Before we discuss this let us understand the functionality of telephone company's switch. A telephone company's switch is a complex device whose circuitry may encompass a large part of a building. The goals of switch maintenance are to minimize the number of calls to be rerouted due to bad connections and to ensure that faults are repaired quickly. Bad connections caused due to failure of connection between two telephone lines.

COMPASS is an expert system which checks error messages derived from the switch's self test routines, look for open circuits, reduces the time of operation of components etc. To find the cause of a switch problem, it looks at a series of such messages and then uses its expertise. The system can suggest the running of additional tests, or the replacement of a particular component, for example, a relay or printed circuit board.

As expertise in this area was scarce, so it was a fit case for taking help of an expert system like COMPASS (We will discuss later, how knowledge acquisition is done in COMPASS).

1.6 EXPERT SYSTEM BUILDING TOOLS

Expert system tools are designed to provide an environment for development of expert systems mainly through the approach of prototyping.

The expert systems development process is normally a mixture of prototyping and other incremental development models of software engineering rather than the conventional "waterfall model". Although using incremental development has a problem of integrating new functionality with the earlier version of expert system but the expert system development environments try to solve this problem by using modular representations of knowledge (some of the representation schemes like frames, rule based systems etc. have already been discussed above).

Software tools for development of expert systems mainly fall into the following Categories:

- **Expert System Shells:** These are basically a set of program and to be more specific - abstractions over one or more applications programs. One of the major examples is EMYCIN which is the rule interpreter of the famous expert system called MYCIN (a medical diagnostic system). EMYCIN also constitutes related data structures like knowledge tables and indexing mechanism over these tables. Some recent versions of EMYCIN like M.4 is a very sophisticated shell which combine the backward chaining of EMYCIN with frame – like data structures.
- **High Level Programming Languages :** These languages are basically used to provide a higher level of abstraction by hiding the implementation details because of which the programmer need not worry about efficiency of storing, accessing and manipulating data. One of the good examples is OPS5 rule language. But many of such languages are research tools and may not available commercially.
- **Multiple programming environments :** These provide a set of software modules to allow the user to use and mix a number of different styles of artificial intelligence programming. One of the examples is called LOOPS which combines rule based and object – oriented approaches of knowledge representation.

We will now keep our focus on “Expert System Shells”.

1.6.1 Expert System Shells

An expert system tool, or shell, is a software development environment containing the basic components of expert systems. Associated with a shell is a prescribed method for building applications by configuring and instantiating these components.

Expert system shells are basically used for the purpose of allowing non-programmers to take advantage of the already developed templates or shells and which have evolved because of the efforts of some pioneers in programming who have solved similar problems before. The core components of an expert systems are the knowledge base and the reasoning engine.

A generic expert system shell is shown below :

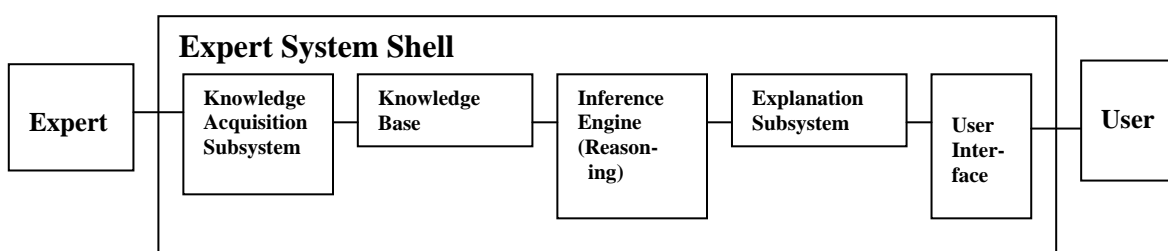


Figure 1.2: Components of Expert System Tool (Shell)

As we can see in the figure above, **the shell includes** the *inference engine*, a *knowledge acquisition subsystem*, an *explanation subsystem* and a *user interface*. When faced with a new problem in any given domain, we can find a shell which can provide the right support for that problem, so all we need is the knowledge of an expert. There are many commercial expert system shells available now, each one adequate for a different range of problems. Taking help of expert system shells to develop expert systems greatly reduces the cost and the time of development as compared to developing the expert system from the scratch.

Let us now discuss the components of a generic expert system shell. We will discuss about:

- **Knowledge Base**
- **Knowledge Acquisition Subsystem**
- **Inference Engine**
- **Explanation Subsystem**
- **User Interface**

1.6.1.1 Knowledge Base

It contains facts and heuristic knowledge. Developers try to use a uniform representation of knowledge as far as possible. There are many knowledge representation schemes for expressing knowledge about the application domain and some advanced expert system shells use both frames (objects) and IF-THEN rules. In PROLOG the knowledge is represented as logical statements.

1.6.1.2 Knowledge Acquisition Subsystem

The process of capturing and transformation of potentially useful information for a given problem from any knowledge source (which may be a human expert) to a program in the format required by that program is the job of a knowledge acquisition subsystem. So we can say that this subsystem helps experts build knowledge bases.

As an expert may not be a computer literate, so capturing information includes interviewing, preparing questionnaires etc. which is a very slow and time-consuming process. So **collecting knowledge needed to solve problems** and building the knowledge base has always been **the biggest bottleneck** in developing expert systems.

Some of the reasons behind the difficulty in collecting information are given below :

- *The facts and rules or principles of different domains cannot easily be converted into a mathematical or a deterministic model, the properties of which are known.* For example a teacher knows how to motivate the students but putting down on the paper, the exact reasons, causes and other factors affecting students is not easy as they vary with individual students.
- *Different domains have their own terminology* and it is very difficult for experts to communicate exactly their knowledge in a normal language.
- *Capturing the facts and principles alone is not sufficient to solve problems.* For example, experts in particular domains which information is important for specific judgments, which information sources are reliable and how problems can be simplified, which is based on personal experience. Capturing such knowledge is very difficult.
- *Commonsense knowledge found in humans continues to be very difficult to capture*, although systems are being made to capture it.

The idea of automated knowledge capturing has also been gaining momentum. In fact **“machine learning”** is one of the important research areas for some time now. The goal is that, a computing system or machine could be enabled to learn in order to solve problems like the way humans do it.

Example (Knowledge acquisition in COMPASS).

As we already know that COMPASS is an expert system which was built for proper maintenance and troubleshooting of telephone company's switches.

Now, for knowledge acquisition, knowledge from a human expert is elicited. An expert explains the problem solving technique and a knowledge engineer then converts it into an if-then-rule. The human expert then checks if the rule has the correct logic and if a change is needed then the knowledge engineer reformulates the rule.

Sometimes, it is easier to troubleshoot the rules with pencil and paper (i.e., hand simulation), at least the first time than directly implementing the rule and changing them again and again.

Let us summarize the knowledge acquisition cycle of COMPASS :

- Extract knowledge from a human expert.
- Document the extracted knowledge.
- Test the new knowledge using following steps:
 - Let the expert analyze the new data.
 - Analyze the same data using pencil and paper using the documented knowledge.
 - Compare the results of the expert's opinion with the conclusion of the analysis using pencil and paper.
 - If there is a difference in the results, then find the rules which are the cause of discrepancy and then go back to rule 1 to gather more knowledge from the expert to solve the problem. Otherwise, exit the loop.

1.6.1.3 Inference Engine

An inference engine is used to perform reasoning with both the expert knowledge which is extracted from an expert and most commonly a human expert) and data which is specific to the problem being solved. Expert knowledge is mostly in the form of a set of IF-THEN rules. The case specific data includes the data provided by the user and also partial conclusions (along with their certainty factors) based on this data. In a normal forward chaining rule-based system, the case specific data is the elements in the working memory.

Developing expert systems involve knowing how knowledge is accessed and used during the search for a solution. Knowledge about what is known and, when and how to use it is commonly called **meta-knowledge**. In solving problems, a certain level of planning, scheduling and controlling is required regarding what questions to be asked and when, what is to be checked and so on.

Different strategies for using domain-specific knowledge have great effects on the performance characteristics of programs, and also on the way in which a program finds or searches a solution among possible alternatives. Most knowledge representations schemes are used under a variety of reasoning methods and research is going on in this area.

1.6.1.4 Explanation Subsystem (Example MYCIN)

An explanation subsystem allows the program to explain its reasoning to the user. The explanation can range from how the final or intermediate solutions were arrived at to justifying the need for additional data.

Explanation subsystems are important from the following points of view :

- (i) **Proper use of knowledge:** There must be some for the satisfaction of knowledge engineers that the knowledge is applied properly even at the time of development of a prototype.
- (ii) **Correctness of conclusions:** User's need to satisfy themselves that the conclusions produced by the system are correct.
- (iii) **Execution trace:** In order to judge that the knowledge elicitation is proceeding smoothly and successfully, a complete trace of program execution is required.
- (iv) **Knowledge of program behavior:** For proper maintenance and debugging, the knowledge of program behavior is necessary for the programmers.
- (v) **Suitability of reasoning approach:** Explanation subsystems are necessary to ensure that reasoning technique applied is suitable to the particular domain.

Explanation in expert systems deals with the issue of control because the reasoning steps used by the programs will depend on how it searches for a solution.

Explanation subsystems are also related to evaluation as by checking the outputs produced by a system and after examining the trace of its actions performed while reasoning, it can be decided that whether or not a system is producing the right answer for right reasons. Without a good explanation subsystem, an expert will be unable to judge the system's performance or will be unable to find ways to improve it.

To give an idea of how an explanation subsystem works, let us have a simple example:

Example:

Suppose there is a simple rule based system **to diagnose the problems in a car.**

Let the system has following rules :

R1: IF engine-gets-petrol AND engine-turns-over
THEN spark-plug-fault

R2: IF NOT engine-turns-over AND NOT lights-come-on
THEN battery-problem

R3: IF NOT engine-turns-over AND lights-come-on
THEN starter-not-working

R4: IF fuel-tank-has-petrol
THEN engine-gets-petrol

We also assume that the system has no initial facts.

Explanation subsystems allow the user to ask ***why** it asked a particular question*, and ***how** it reached some conclusion*. These questions are answered by referring to the system goals, the rules being used, and any existing problem data.

Now let us focus on the example given above to see the explanation facilities provided, which involve a dialogue involving *why* and *how* questions and their explanations.

System : Does the fuel tank has gas?
User : Yes.

System : Does the engine turns over?

User : No

System : Do lights come on?

User : **Why?**

System : **I am trying to prove that there is a battery problem.** It has been established

that the engine does not turns over. Therefore if it is not the case that the lights

come on, then there is a battery problem.

Is it true that the lights come on?

User : No.

System : I conclude that there is a battery problem.

User : **How?**

System : This follows from Rule 2:

IF NOT engine-turns-over AND NOT lights-come-on was given by the user.

Providing such an explanation facility involves stating what rules are used in reaching conclusions, and using these records to compose explanations like the ones above. Giving simple explanations like those above is not very difficult, and is quite useful.

Explanation facilities in expert systems are sometimes not used, and sometimes they are not easily accepted by their users. There are a whole lot of reasons for this. One reason is that the explanations just reference the “surface” knowledge encoded in the rules, rather than providing the “deep” knowledge about the domain which originally motivated the rules but which is usually not represented. So, the system will say that it concluded something because of rule 5, but not explain what rule 5 intends to say. In the example given below, maybe the user needs to understand that both the lights and the starter use the battery, which is the underlying purpose of the second rule in this example. Another reason for the frequent failure of explanation facilities is the fact that, if the user fails to understand or accept the explanation, the system can’t re-explain in another way (as people can). Explanation generation is a fairly large area of research, concerned with effective communication i.e., how to present things so that people are really satisfied with the explanation, and what implications does this have for how we represent the underlying knowledge.

Explanation Subsystem in MYCIN (An overview)

MYCIN is one of the first popular expert systems made for the purpose of medical diagnosis. Let us have a look at how the explanation subsystem in MYCIN works :

To explain the reasoning for deciding on a particular medical parameter’s or symptom’s value, it retrieves a set of rules and their conclusions. It allows the user to ask questions or queries during a consultation.

To answer the questions the system relies on its ability to display a rule invoked at any point during the consultation and also recording the order of rule invocations and associating them with particular events (like particular questions).

As the system using backward chaining so most of the questions belong to “Why” or “How” category. To answer “Why” questions, the system looks up the hierarchy (i.e., tree) of rules to see which goals the system is trying to achieve and to answer “Why”

questions, the system must look down the hierarchy (i.e., tree) to find out that which sub-goals were satisfied to achieve the goal.

We can see that explanation process is nothing but a search problem requiring tree traversal.

As MYCIN keeps track of the goal to sub-goal sequence of the computation, so it can answer questions like:

“Why did you ask that if the stain of the organism is gram negative ?”

In response to this, the system would quote the rules which states “gram negative staining” may be in conjunction with other conditions and would state that what it was trying to achieve.

Similarly, if there is “How” question like:

“How do say that Organism-2 might be proteus ?”

In its reply, MYCIN would state the rules that were applied in reaching this conclusions and their degree of certainty and what was the last question asked etc.

Thus we can see that because of the backward chaining approach, the system is able to answer “Why” and “How” questions satisfactorily. But the rule application would not be easy if the chains of reasoning are long.

1.6.1.5 User interface

It is used to communicate with the user. The user interface is generally not a part of the expert system technology, and was not given much attention in the past. However, it is now widely accepted that the user interface can make a critical difference in the utility of a system regardless of the system’s performance.

Now as an example, let us discuss an expert system shell called EMYCIN.

1.6.1.6 EMYCIN (An expert system shell)

EMYCIN provides a domain-independent framework or template for constructing and running any consultation programs. EMYCIN stands for “Empty MYCIN” or “Essential MYCIN” because it basically constitutes a MYCIN system minus its domain-specific medical knowledge. However, EMYCIN is something more than this, as it offers a number of software tools for helping expert system designers in building and debugging performance programs.

Some characteristics of EMYCIN are:

- It constitutes an abbreviated rule language, which uses ALGOL-like notation and which is easier than LISP and is more concise than the English subset used by MYCIN.
- It uses backward chaining which is similar to MYCIN.
- It indexes rules, in-turn organising them into groups, based on the parameters which are being referenced.
- It has an interface for system designer which provides tools for displaying, editing and partitioning rules, editing knowledge held in tables, and also running rule sets on sets of problems. As part of system designer’s interface, EMYCIN

also included a knowledge editor (a program) called TEIRESIAS whose job was to provide help for the development and maintenance of large knowledge bases.

- It also has a user interface which allows the user to communicate with the system smoothly.

1.7 SOME APPLICATION AREAS OF EXPERT SYSTEMS

The scope of applications of expert systems technology to practical problems is so wide that it is very difficult to characterize them. The applications find their way into most of the areas of knowledge work. Some of the main categories of applications of an expert system are given below.

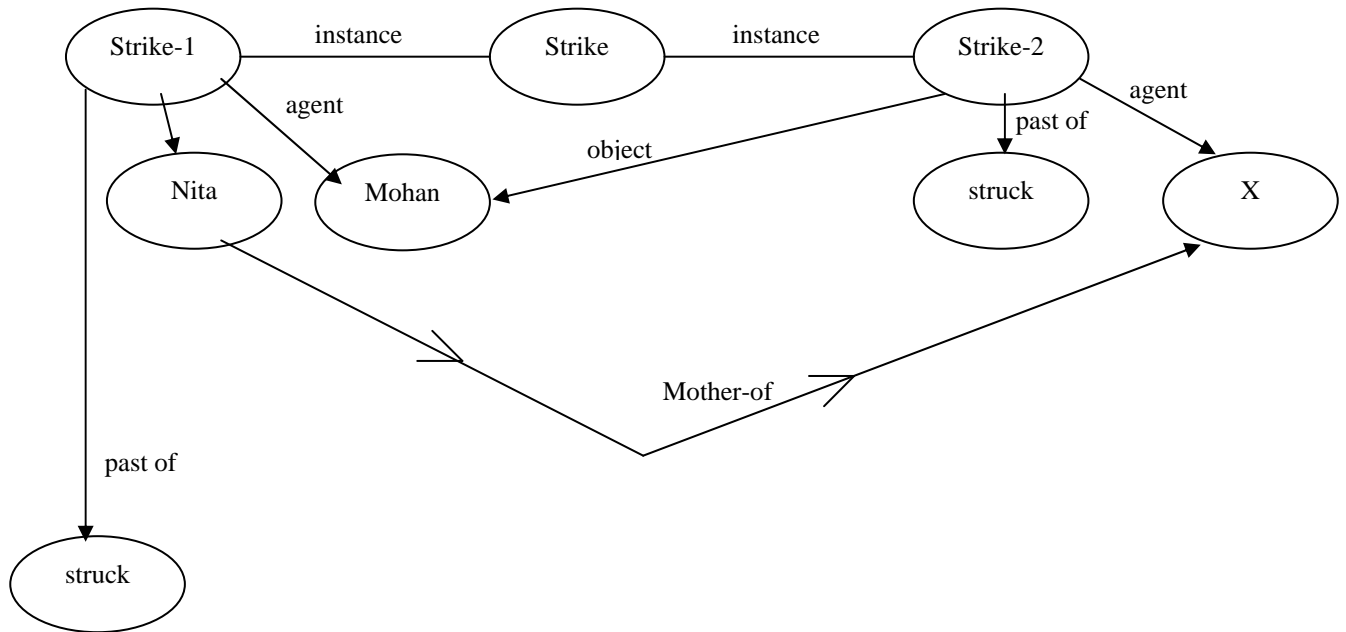
- **Diagnosis and Troubleshooting:** This class comprises systems that deduce faults and suggest corrective actions for a malfunctioning device or process. Medical diagnosis was one of the first knowledge areas to which ES technology was applied, but use of expert systems for solving and diagnosis of engineered systems has become common.
- **Planning and Scheduling:** Systems that fall into this class analyze a set of one or more potentially complex and interacting goals in order to determine a set of actions to achieve those goals. This class of expert systems has great commercial potential. Examples include scheduling of flights, personnel, manufacturing process planning etc.
- **Process Monitoring and Control:** Systems falling in this class analyze real-time data from physical devices with the goal of noticing errors, predicting trends, and controlling for both optimality and failure correction. Examples of real-time systems that actively monitor processes are found in the steel making and oil refining industries.
- **Financial Decision Making:** The financial services industry has also been using expert system techniques. Expert systems belonging to this category act as advisors, risk analyzers etc.
- **Knowledge Publishing:** This is a relatively new, but also potentially explosive area. The primary function of the expert system is to deliver knowledge that is relevant to the user's problem, in the context of the user's problem.
- **Design and Manufacturing:** These systems assist in the design of physical devices and processes, ranging from high-level conceptual design of abstract entities all the way to factory floor configuration of manufacturing processes.

1.8 SUMMARY

In this unit, we have discussed various issues in respect of expert systems. To begin with, in section 1.2 we define the concept of 'expert system'. In view of the fact that an expert system contains knowledge of a specific domain, in section 1.4, we discuss various schemes for representing the knowledge of a domain. Section 1.5 contains examples of some well-known expert systems. Tools for building expert systems are discussed in section 1.6. Finally, applications of expert systems are explained in section 1.7.

1.9 SOLUTIONS/ANSWERS

Ex. 1) In this case, it is not the same 'strike' action but two strike actions which are involved in the sentence. Therefore, we use 'strike' to denote a generic action of striking whereas 'strike-1' and 'strike-2' are its instances or members. Thus, we get the semantic network.



Ex. 2) (date
 (day (integer (1....31)))
 (month (integer (1.....12)))
 (year (integer (1.....10000)))
 (day-of-the-week (set (Mon Tue Wed Thu Fri Sat Sun)))
 (procedure (compute day-of-the-week (day month year))),

Where the procedure day-of-the-week takes three arguments.

Some problems in Semantic Networks and Frames:

There are problems in expressing certain kinds of knowledge when either semantic networks or frames are used for knowledge representation. For example, **it is difficult** although not impossible **to express disjunctions and hence implications, negations, and general non-taxonomic knowledge** (i.e., non-hierarchical knowledge) in these representations.

Ex. 3) In order to translate in PL, let us use the symbols:

ML: There is a Moral Law

SG: Someone Gave it (the word 'it' stands for moral law)

TG: There is God.

Using these symbols, the **statements** (i) to (iv) become the **formulae** (i) to (iv) of PL as given below:

(i) ML

(ii) ML→SG

(iii) SG→TG and

(iv) TG

Applying Modus Ponens to formulae (i) and (ii) we get the formula

(v) SG

Applying Modus Ponens to (v) and (iii), we get

(vi) TG

But formula (vi) is the same as (iv), which is required to be established. Hence the proof.

Ex. 4) Initially, the working memory contains the following assertions :

(has-hair raja)	representing the fact “raja has hair”
(big-mouth raja)	representing the fact “raja has a big mouth”
(long-pointed-teeth raja)	representing the fact “raja has long pointed teeth”
(claws raja)	representing the fact “raja has claws”

We start with one of the assertions about “raja” from the working memory.

Let us choose the first assertion : (raja has-hair)

Now we take this assertion and try to match the antecedent part of a rule. In the rule 2, the antecedent part is satisfied by substituting “raja” in place of X. So the consequent (mammal X) is established by replacing “raja” in place of X.

The working memory is now updated by adding the assertion (mammal raja).

So the working memory now looks like:

(mammal raja)
 (has-hair raja)
 (big-mouth raja)
 (long-pointed-teeth raja)
 (claws raja)

Now we try to match assertion (mammal raja) to the antecedent part of a rule. The first rule whose antecedent part supports the assertion is rule 3. So the control moves to the second part of rule 3, which says that (eats-meat X), but this is not found in any of the assertions present in working memory, so rule 3 fails.

Now, the system tries to find another rule which matches assertion (mammal raja), it find rule 4 whose first part of antecedent supports this. So the control moves to the second part of the antecedent in rule 4, which says that something i.e., X must have pointed teeth. This fact is present in the working memory in the form of assertion (long-pointed-teeth raja) so the control now moves to the third antecedent part of rule 4 i.e., something i.e., X must have claws. We can see that this is supported by the assertion (claws raja) in the working memory. Now, as the whole antecedent of rule 4 is satisfied so the consequent of rule 4 is established and the working memory is updated by the addition of the assertion (carnivorous raja), after substituting “raja” in place of X.

So the working now looks like:

(carnivorous raja)

(mammal raja)
(has-hair raja)
(big-mouth raja)
(long-pointed-teeth raja)
(claws raja)

Now in the next step, the system tries to match the assertion (carnivorous raja) with one of the rules in working memory. The first rule whose antecedent part matches this assertion is rule 6. Now as the first part of the antecedent in rule 6 matches with the assertion, the control moves to the second part of the antecedent i.e., X has dark spots. There is no assertion in working memory which supports this, so the rule 6 is aborted.

The system now tries to match with the next rule which matches the assertion (carnivorous raja). It finds rule 8 whose first part of antecedent matches with the assertion. So the control moves to the second part of the antecedent of rule 8 which says that something i.e., X must have big mouth. Now this is already present in the working memory in the form of assertion (big-mouth raja) so the second part and ultimately the whole antecedent of rule 8 is satisfied.

And, so the consequent part of rule 8 is established and the working memory is updated by the addition of the assertion (lion raja), after substituting “raja” in place of X.

The working memory now looks like:

(lion raja)
(carnivorous raja)
(mammal raja)
(has-hair raja)
(big-mouth raja)
(long-pointed-teeth raja)
(claws raja)

Hence, as the goal to be achieved i.e., “raja is a lion” is now part of the working memory in the form of assertion (lion raja), **so the goal is established** and processing stops.

1.10 FURTHER READINGS

1. Russell S. & Norvig P., *Artificial Intelligence A Modern Approach* (Second Edition) (Pearson Education, 2003).
2. Patterson D. W: *Introduction to Artificial Intelligence and Expert Systems* (Prentice Hall of India, 2001).

UNIT 2 INTELLIGENT AGENTS

Structure	Page Nos.
2.0 Introduction	36
2.1 Objectives	37
2.2 Definitions	37
2.3 Agents and Rationality	39
2.3.1 Rationality vs. Omniscience	
2.3.2 Autonomy and learning capability of the agent	
2.2.3 Example: A boundary following robot	
2.4 Task Environment of Agents	42
2.4.1 PEAS (Performance, Environment, Actuators, Sensors)	
2.4.2 Example An Automated Public Road Transport Driver	
2.4.3 Different Types of Task Environments	
2.4.3.1 Fully Observable vs. Partially Observable Environment	
2.4.3.2 Static vs. Dynamic Environment	
2.4.3.3 Deterministic vs. Stochastic Environment	
2.4.3.4 Episodic vs. Sequential Environment	
2.4.3.5 Single agent vs. Multi-agent Environment	
2.4.3.6 Discrete Vs. Continuous Environment	
2.4.4 Some Examples of Task Environments	
2.4.4.1 Crossword Puzzle	
2.4.4.2 Medical Diagnosis	
2.4.4.3 Playing Tic-tac-toe	
2.4.4.4 Playing Chess	
2.4.4.5 Automobile Driver Agent	
2.5 The Structure of Agents	49
2.5.1 SR (Simple Reflex) Agents	
2.5.2 Model Based reflex Agents	
2.5.3 Goal-based Agents	
2.5.4 Utility-based Agents	
2.5.5 Learning Agents	
2.6 Different Forms of Learning in Agents	56
2.7 Summary	58
2.8 Solutions/Answers	58
2.9 Further Readings	58

2.0 INTRODUCTION

Since the time immemorial, we, the human beings, have always toyed with the idea of having some sort of slaves or agents, which would act as per our command, irrespective of the shape they take, as long as they do the job for which they have been designed or acquired. With the passage of time, human beings have developed different kinds of machines, where each machine has been intended to perform specific operation or a set of operations. However, ***with the development of the computer, human aspirations have increased manifolds as it has allowed us to think of and actually implement non-human agents, which would show some level of independence and intelligence. Robot, one of the most popular non-human agents***, is a machine capable of perceiving the environment it is in, and further capable of taking some action or of performing some job either on its own or after taking some command.

Despite their perceived benefits, the dominant public image of the artificially embodied intelligent machines is more as potentially dangerous than potentially beneficial machines to the human race. The mankind is worried about the potentially dangerous capabilities of the robots to be designed and developed in the future.

Actually, any technology is a double-edged sword. For example, the Internet along with World Wide Web, on one hand, allows us to acquire information at the click of a button, but, at the same time, the Internet provides an environment in which a number of children (and, of course, the adults also) become addicts to downloading pornographic material. Similarly, **the development of non-human agents might not be without its tradeoffs. For example**, the more intelligent the robots are designed and developed, the more are the chances of a robot pursuing its own agenda than its master's, and more are the chances of a robot even attempting to destroy others to become more successful. Some intellectuals even think that, not in very distant future, there might be robots capable of enslaving the human beings, though designed and developed by the human beings themselves. Such concerns are not baseless. However, the (software) agents developed till today and the ones to be developed in the near future are expected to have very limited capabilities to match the kind of intelligence required for such behaviour.

In respect of the design and development of intelligent agents, with the passage of time, the momentum seems to have shifted from hardware to software, the latter being thought of as a major source of intelligence. But, obviously, some sort of hardware is essentially needed as a home to the intelligent agent.

2.1 OBJECTIVES

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

- define the concept of an agent;
- explain the concepts of a 'rational agent' and 'rationality';
- tell about the various task environments, for which the use of agents is appropriate to solve problems from;
- explain the structure of an agent, and
- discuss various forms of used for agents to learn.

2.2 DEFINITIONS

An **agent** may be thought of as an entity that acts, generally on behalf of someone else. More precisely, an **agent** is an entity that *perceives* its environment through *sensors* and *acts* on the environment through *actuators*. Some experts in the field require an agent to be additionally autonomous and goal directed also.

A **percept** may be thought of as an input to the agent through its sensors, over a unit of time, sufficient enough to make some sense from the input.

Percept sequence is a sequence of percepts, generally long enough to allow the agent to initiate some action.

In order to further have an idea about what a *computer agent is*, let us consider one of the first definitions of agent, which was coined by John McCarthy and his friends at MIT.

A software agent is a system which, when given a goal to be achieved, could carry out the details of the appropriate (computer) operations and further, in case it gets stuck, it can ask for advice and can receive it from humans, may even evaluate the appropriateness of the advice and then act suitably.

Essentially, a computer agent is a computer software that additionally has the following attributes:

- (i) it has autonomous control i.e., it operates under its own control
- (ii) it is perceptive, i.e., it is capable of perceiving its own environment
- (iii) it persists over a long period of time
- (iv) it is adaptive to changes in the environment and
- (v) it is capable of taking over others' goals.

As the concept of (software) agent is of relatively recent origin, different pioneers and other experts have been conceiving and using the term in different ways. There are two distinct but related approaches for defining an agent. The **first approach** treats an agent as an **ascription** i.e., the perception of a person (which includes expectations and points of view) whereas the **other approach** defines an agent on the basis of the **description** of the properties that the agent to be designed is expected to possess.

Let us first discuss the definition of agent according to first approach. Among the people who consider *an agent as an ascription*, a popular slogan is “**Agent is that agent does**”. In everyday context, **an agent is expected to act on behalf of someone** to carry out a particular task, which has been delegated to it. But to perform its task successfully, the agent must have knowledge about the domain in which it is operating and also about the properties of its current user in question. In the course of normal life, we hire different agents for different jobs based on the required expertise for each job. Similarly, a **non-human intelligent agent also is imbedded with required expertise of the domain as per requirements of the job under consideration**. For example, *a football-playing agent would be different from an email-managing agent*, although both will have the common attribute of modeling their user.

According to the second approach, an agent is defined as an entity, which functions continuously and autonomously, in a particular environment, which may have other agents also. **By continuity and autonomy of an agent, it is meant** that the agent must be able to carry out its job in a flexible and intelligent fashion and further is expected to adapt to the changes in its environment without requiring *constant* human guidance or intervention. Ideally, an agent that functions continuously in an environment over a long period of time would also **learn** from its experience. In addition, we expect an agent, which lives in a multi-agent environment, to be able to **communicate and cooperate** with them, and perhaps move from place to place in doing so.

According to the **second approach** to defining agent, **an agent is supposed to possess some or all of the following properties:**

- **Reactivity:** The ability of sensing the environment and then acting accordingly.
- **Autonomy:** The ability of moving towards its goal, changing its moves or strategy, if required, without much human intervention.
- **Communicating ability:** The ability to communicate with other agents and humans.
- **Ability to coexist by cooperating:** The ability to work in a multi-agent environment to achieve a common goal.
- **Ability to adapt to a new situation:** Ability to learn, change and adapt to the situations in the world around it.
- **Ability to draw inferences:** The ability to infer or conclude facts, which may be useful, but are not available directly.
- **Temporal continuity:** The ability to work over long periods of time.
- **Personality:** Ability to impersonate or simulate someone, on whose behalf the agent is acting.
- **Mobility:** Ability to move from one environment to another.

2.3 AGENTS AND RATIONALITY

Further, a **rational agent** is an agent that acts in a manner that achieves best outcome in an environment with certain outcomes. In an uncertain environment, a rational agent through its actions attempts the best-expected outcome.

It may be noted that *correct inferencing* is one of the several possible mechanisms for achieving *rationality*. However, sometimes a rational action is also possible without inferencing. For example, removing hand when a very hot utensil is touched *unintentionally* is an example of rationality based on *reflex action* instead of based on *inferencing*.

We discuss the concepts of **rationality** and **rational agent** in some more detail.

Attempting to take always the correct action, possibly but not necessarily involving logical reasoning, is only one part of being rational. Further, if a perfectly correct action or inference is not possible then taking an approximately correct, but, optimal action under the circumstances is a part of rationality.

Like other attributes, we need some *performance measure* (i.e., the criteria to judge the performance or success in respect of the task to be performed) to judge rationality. A good **performance measure for rationality** must be:

- Objective in nature
- It must be measurable or observable and
- It must be decided by the designer of the agent keeping in mind the problem or the set of problems for handling which the agent is designed.

In summary, rationality depends on:

- The performance measure chosen to judge the agent's activities.
- The agent's knowledge of the current environment or of the world in which it exists. The better the knowledge of the environment, the more will be probability of the agent taking an appropriate action.
- The length of the percept sequence of the agent. In the case of a longer percept sequence, the agent can take advantage of its earlier decisions or actions for similar kind of situations.
- The set of actions available to the agent.

From the previous discussion, we know that a **rational agent** should take an action, which **would correct its performance measure** on the basis of its knowledge of the world around it and the percept sequence.

By the *rationality of an agent*, we do not mean it to be always successful or it to be **omniscient**. *Rationality is concerned with* the agent's capabilities for **information gathering, exploration and learning** from its environment and experience and is also concerned with the **autonomy** of the agent.

2.3.1 Rationality vs. Omniscience

The basic difference between being rational and being omniscient is that **rationality** deals with trying to maximize the output on the basis of current input, environmental conditions, available actions and past experience whereas being **omniscient** means having knowledge of *everything*, including knowledge of the future i.e., what will be the output or outcome of its action. Obviously being omniscient is next to impossible.

In this context, let us consider the following scenario: Sohan is going to the nearby grocery shop, but unfortunately when Sohan is passing through the crossing suddenly a police party comes at that place chasing a terrorist and attempts to shoot the terrorist but unfortunately the bullet hits Sohan and he is injured.

Now the question is: *Is Sohan irrational in moving through that place.* The answer is no, because the human agent *Sohan* has no idea, nor is expected to have an idea, of what is going to happen at that place in the near future. Obviously, Sohan is *not omniscient* but, from this incident can *not be said to be irrational*.

2.3.2 Autonomy and Learning Capability of the Agent

Autonomy means the dependence of the agent on its own perceptions (what it perceives or receives from the environment through senses) rather than the prior knowledge of its designer. In other words, an agent is **autonomous** if it is capable of learning from its experience and has not to depend upon its prior knowledge which may either be not complete or be not correct or both. Greater the autonomy more flexible and more rational the agent is expected to be. So we can say that a rational agent should be autonomous because it should be able to learn to compensate for the incomplete or incorrect prior knowledge provided by the designer. In the initial stage of its operations, the agent may not, rather should not, have *complete autonomy*. This is desirable, in view of the fact that in the initial stage, the agent is yet to acquire knowledge of the environment should use the experience and knowledge of the designer. But, as the agent gains experience of its environment, its behavior should become more and more independent of its prior knowledge provided to it by its designer.

Learning means the agent can update its knowledge based on its experience and changes in the environment, for the purpose of taking better actions in future. Although the designer might feed some prior knowledge of the environment in the agent but, as mentioned earlier, as the environment might change with the passage of time, therefore, feeding complete knowledge in the agent, at the beginning of the operations is neither possible nor desirable. Obviously, there could be some extreme cases in which environment is static, i.e., does not change with time, but such cases are rare. In such rare cases, however, giving complete information about the environment may be desirable. So, in general, in order to update itself, the agent should have the learning capability to update its knowledge with the passage of time.

2.3.3 Example (A boundary following robot)

Let us consider the first **example of an agent**, which is *a robot that follows a boundary wall*.

The definition of the problem, to solve, which an agent is to be designed, consists in giving details of

- (i) The perception capabilities that are to be available to the agent
- (ii) The actions that the agent would be able to perform
- (iii) The environment in which the agent is expected to perform the task or solve the problem.

Environment and tasks

The robot is assumed to be enclosed in a room, which is perceived as a two-dimensional grid-space. The room is enclosed on all sides by walls high enough not to allow the robot to escape. The room may have some immovable objects. For the purpose of movement of the robot, the room is supposed to be divided into cells, in

the sense that at one time, the robot occupies only one cell and does not have its parts spread over two or more cells. To be capable of boundary-following behavior, the robot must be *able to sense* whether or not certain cells are free for it to possibly occupy in the next move. The *only action* it is able to take is *to move from the currently occupied cell to* any one of the unoccupied surrounding cells, e.g., to c_1, c_2, \dots, c_8 as shown in the following diagram.

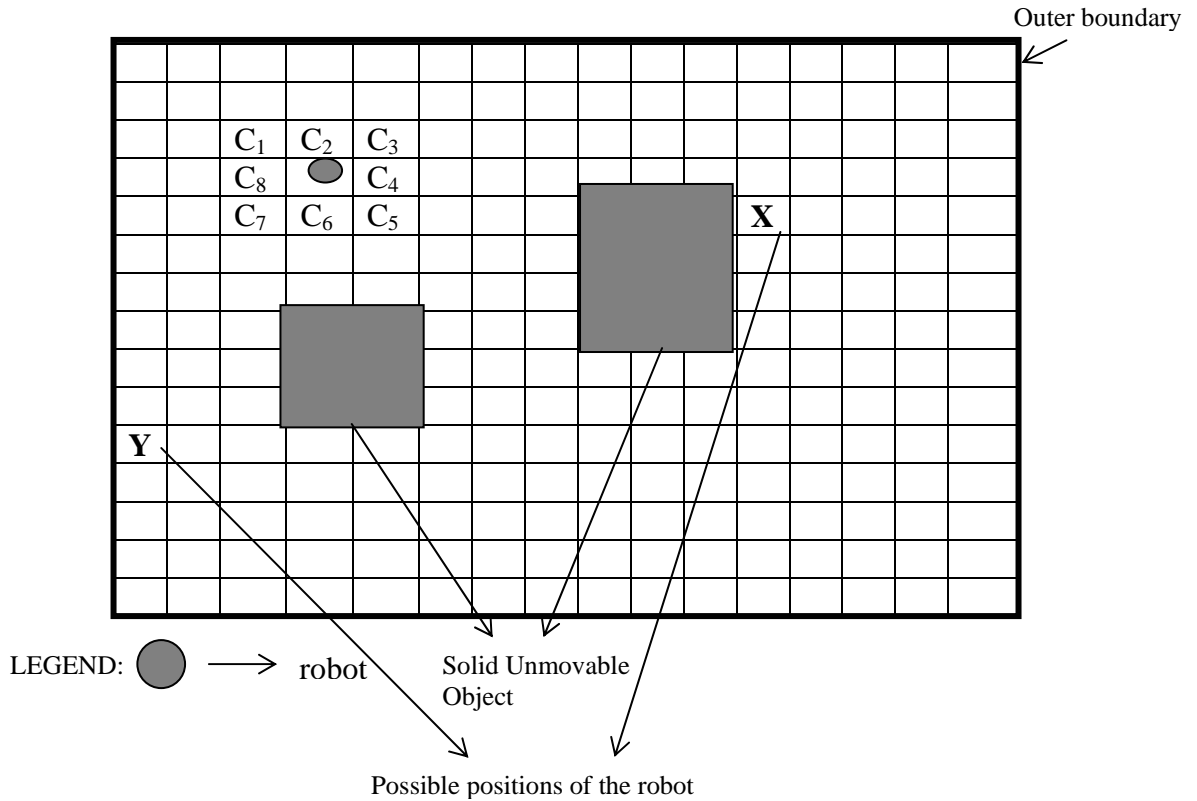


Figure 2.1: A Boundary Following Robot

The sensory input about the surrounding cells can be represented in the form of an 8-tuple vector say $\langle c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8 \rangle$ where each c_i is a binary number i.e., c_i is either a 0 for representing a **free cell** or a 1 for representing the fact that the cell is **occupied** by some object. If, at some stage, the robot is in the cell represented by **X**, as shown in Figure 2.1, then the sense vector would be $\langle 0, 0, 0, 0, 0, 0, 0, 1 \rangle$. It may be noted that the corner and the boundary cells may not have exact eight surrounding cells. However, for such cells, the missing neighbouring cell may be treated as occupied. For example, for the cell **y** of Figure 2.1, the neighbouring cells to the left of the cell **y** do not exist. However, we assume these cells exist but are occupied. Thus, if the robot is in position **Y** of Figure 2.1 then the sense vector for the robot is $\langle 1, 0, 0, 0, 0, 0, 1, 1 \rangle$.

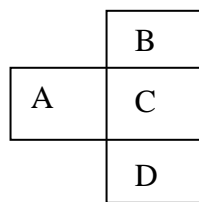
Also, the robot performs only one type of action i.e., it can move in one of the free cells adjacent to it. In case the robot attempts to move to a cell, which is not free, the action will have no effect i.e., there will be no action.

On the basis of the properties of the environment in which the robot is operating, **it is the job of the designer to develop an algorithm for the process that converts robot's precept sequences** (which in this case is the sensory input c_1 to c_8) **into corresponding actions**. Let us divide the task of determining the action the agent should take on the basis of the sensory inputs into two phases, viz., **perception** phase and **action** phase.

Perception Phase:

There is an enclosed space, which we call a room. The room is thought of as divided into cells. A cell has just enough space for the robot to be accommodated. Further, at any time, robot, except during transition, can not lie across two cells.

- Placement of the agent in some cell of the room is essentially random.
- Agent is to perform boundary following.
- The sensory input for the robot consists of the binary values c_1, c_2, \dots, c_8 in the eight cells adjacent to the position occupied by the robot. Because there are 8 adjacent cells, so the possible combinations of these values is 2^8 i.e., 256 combinations. The movement of the robot to any one of the unoccupied/free surrounding cells depends upon the robot's environmental conditions. For example, if the (traffic) signal to the right is green and some cells on the right of the cell currently occupied by the robots, are free then move to the uppermost cell of free cells to the right. For example,



if A is the cell currently occupied by the robot and the cell B is occupied by some object, but cells C and D are free. And, further, signal on the right is green, then the robot moves to cell C. However, if both B and C are occupied, then the robot moves to cell D. These environmental conditions are determined in terms of what we call *features*. For the current problem, let us assume that there are 4 binary valued features of the agent, namely f_1, f_2, f_3 and f_4 and these features are used to calculate the action to be taken by the agent.

The rules for assigning binary values to features may be assumed to be as given below:

IF $c_2 = 1$ OR $c_3 = 0$ THEN $f_1 = 1$ ELSE $f_1 = 0$.

IF $c_4 = 0$ OR $c_5 = 1$ THEN $f_2 = 1$ ELSE $f_2 = 0$.

IF $c_6 = 1$ OR $c_7 = 1$ THEN $f_3 = 1$ ELSE $f_3 = 0$.

IF $c_8 = 1$ OR $c_1 = 1$ THEN $f_4 = 1$ ELSE $f_4 = 0$.

Action Phase:

After calculating the values of the feature set of the robot, a *pre-defined function* is used to determine boundary following action of the robot (i.e., the movement to one of the surrounding cell). In addition to specified actions determined by the features and the pre-defined function, there may be a default action of movement to a surrounding free cell. The default action may be executed when no action is possible.

In order to illustrate the type of possible actions, let us consider an example of specifying the rules for different actions to be performed by the robot:

Action 1: IF $f_1 = 0$ AND $f_2 = 0$ AND $f_3 = 0$ AND $f_4 = 0$
 THEN move up to the right-most free surrounding cell. If no such cell is free then Attempt Action 3.

Action 2: IF $f_1 = 1$ AND $f_2 = 0$

THEN move topmost free surrounding cell on the right by one cell. If no such cell is free, then stop.

Action 3: IF $f_2 = 1$ AND $f_3 = 0$

THEN move down to the left-most free cell. If no such cell is free then attempt Action 2.

Action 4: IF $f_3 = 1$ AND $f_4 = 0$

THEN move to the bottom-most free cell on the left. If no such surrounding cell is free then move to action 1.

We can see that the combinations of the features values serve as conditions under which the robot would perform its actions.

Next, we enumerate below some of the possible parameters for performance measures in case of such a simple agent.

- The number of times agent has to reverse its earlier courses of actions (more times indicates lower performance).
- The maximum of the average of distances from the walls (the less the maximum, higher the performance).
- The boundary distance traversed in particular time period (the more the distance covered, better the performance).

In the next section, we discuss some of the commonly used terms in context of the Agents.

Exercise 1

On the basis of the figure given below, find the sense vector of the robot if:

- (a) It starts from the location L_1
- (b) It starts from the location

The robot can sense whether the 8 cells adjacent to it are free for it to move into one of them. If the location of the robot is such that some of the surrounding cells do not exist, then these cells are, assumed to exist and further assumed to be occupied

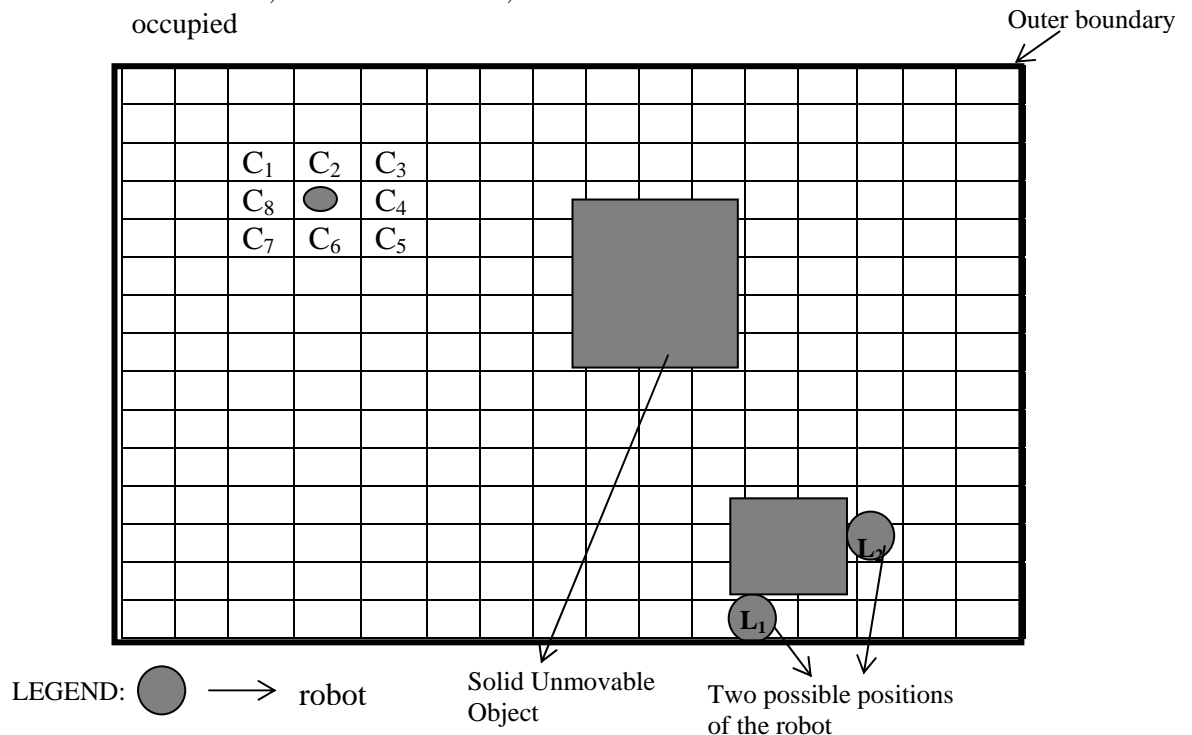


Figure 2.2 Environment of the robot

2.4 TASK ENVIRONMENT OF AGENTS

Task environments or problem environments are the environments, which include all the elements involved in the problems for which agents are thought of as solutions. Task environments will vary with every new task or problem for which an agent is being designed. Specifying the task environment is a long process which involves looking at different measures or parameters. Next we discuss a standard set of measures or parameters for *specifying* a task environment under the heading **PEAS**.

2.4.1 PEAS (Performance, Environment, Actuators, Sensors)

For designing an agent, the first requirement is to specify the task environment to the maximum extent possible. The task environment for an agent to solve one type of problems, may be described by the four major parameters namely, **performance** (which is actually the expected performance), **environment** (i.e., the world around the agent), **actuators** (which include entities through which the agent may perform actions) and **sensors** (which describes the different entities through which the agent will gather information about the environment).

The four parameters may be collectively called as PEAS.

We explain these parameters further, **through an example of an automated agent**, which we will preferably call **automated public road transport driver**. This is a much more complex agent than the simple **boundary following robot** which we have already discussed.

2.4.2 Example (An Automated Public Road Transport Driver Agent)

We describe the task environment of the agent on the basis of PEAS.
Performance Measures:

Some of the performance measures which can easily be perceived of an

automated public road transport driver would be :

- Maximizing safety of passengers
- Maximizing comfort of passengers
- Ability to reach correct destination
- Ability to minimize the time to reach the destination
- Obeying traffic rules
- Causing minimum discomfort or disturbance to other agents
- Minimizing costs, etc.

Environment (or the world around the agent)

We must remember that the environment or the world around the agent is extremely uncertain or open ended. There are unlimited combinations of possibilities of the environment situations, which such an agent could face. Let us enumerate some of the possibilities or circumstances which an agent might face:

- Variety of roads e.g., from 12-lane express-ways, freeways to dusty rural bumpy roads; different road rules including the ones requiring left-hand drive in some parts of the world and right-hand drive in other parts.
- The degree of knowledge of various places through which and to which driving is to be done.

- Various kinds of passengers, including high cultured to almost ruffians etc.
- All kind of other traffic possibly including heavy vehicles, ultra modern cars, three-wheelers and even bullock carts.

Actuators:

These include the following:

- Handling steering wheel, brakes, gears and accelerator
- Understanding the display screen
- A device or devices for all communication required

Sensors:

The agent acting as automated public road transport driver must have some way of sensing the world around it i.e., the traffic around it, the distance between the automobile and the automobiles ahead of it and its speed, the speeds of neighbouring vehicles, the condition of the road, any turn ahead etc. *It may use sensors like odometer, speedometer, sensors telling the different parameters of the engine, Global Positioning System (GPS)* to understand its current location and the path ahead. Also, there should be some sort of sensors to calculate its distance from other vehicles etc.

We must remember that the agent example *the automated public road transport driver*, which we have considered above, is quite difficult to implement. However, there are many other agents, which operate in comparatively simpler and less dynamic environments, e.g., a game playing robot, an assembly line robot control, and an image processing agent etc.

2.4.3 Different Types of Task Environments

Next, we discuss some of the important characteristics of the environments of the problems which are generally considered for solution through agent technology.

- Fully observable vs. partially observable
- Static vs. dynamic
- Deterministic vs. stochastic
- Episodic vs. sequential
- Single agent vs. multi-agent

2.4.3.1 Fully Observable vs. Partially Observable Environment

If the agent knows *everything* about its environment or the world in which it exists, through its sensors, then we say that the environment is **fully observable**. It would be quite convenient for the agent if the environment is a fully observable one because the agent will have a wholesome idea of what all is happening around before taking any action. The agent in a fully observable environment will have a complete idea of the world around it at all times and hence it need not maintain an internal state to remember and store what is going around it. **Fully observable environments are found rarely in reality.**

A **partially observable environment** is that in which the agent can sense **some** of the aspects or features of the world around it, but not all because of any number of reasons including the limitations of its sensors. Normally, there are greater chances of finding a partially observable environment in reality. *In the first example*, viz., **a boundary following robot**, which we have studied, the agent or robot has a limited

view of the environment around it, i.e., the agent has information only about the eight cells immediately adjacent to it in respect of whether each of these is occupied or free. It has no idea of how many non-moving objects are there within the room. Nor, it has an idea of its distance from the boundary wall. Also, *in the second example*, viz., **an automated public road transport driver**, *the driver or the agent has a very restricted idea of the environment around it*. It operates in a very dynamic environment, i.e., an environment that changes very frequently. Further, it has no idea of the number of vehicles and their positions on the road, the action the driver of the vehicle ahead is expected to take, the location of various potholes it is going to face in the next one kilometer and so on. So we can see that in both of these examples, the environment is partially observable.

2.4.3.2 Static vs. Dynamic Environment

Static environment means an environment which does not change while the agent is operating. Implementing an agent in a static environment is relatively very simple, as the agent does not have to keep track of the changes in the environment around it. So the time taken by an agent to perform its operations has no effect on the environment because the environment is static. The **task environment of the “boundary following robot” is static** because no changes are possible in the environment around it, irrespective of how long the robot might operate. Similarly, the environment of an agent solving a crossword puzzle is static.

A dynamic environment on the other hand may change with time on its own or because of the agent's actions. Implementing an agent in a dynamic environment is quite complex as the agent has to keep track of the changing environment around it. The **task environment of the “Automated public road transport driver” is an example of a dynamic environment** because the whole environment around the agent keeps on changing continuously.

In case an environment does not change by itself but an agent's performance changes the environment, then the environment is said to be **semi-dynamic**.

2.4.3.3 Deterministic vs. Stochastic Environment

A deterministic environment means that the current state and the current set of actions performed by the agent will completely determine the next state of the agent's environment *otherwise* the environment is said to be **stochastic**. The decision in respect of an environment being stochastic or deterministic, is taken from the point of view of the agent. If the environment is *simple and fully observable* then there are greater chances of its being *deterministic*. However, if the environment is *partially observable and complex* then it may be *stochastic* to the agent. For example, the *boundary following robot* exists in a deterministic environment whereas *an automated road transport driver agent* exists in a stochastic environment. In the later case, the agent has no prior knowledge and also it cannot predict the behavior of the traffic around it.

2.4.3.4 Episodic vs. Sequential Environment

In an episodic environment, the agent's experience is divided into *episodes*. An agent's actions during an episode depend only on that episode because subsequent episodes do not depend on previous episodes. For example, an agent checking the tickets of persons who want to enter the cinema hall, works in an episodic environment, as the process of checking the tickets of every new person is an episode which is independent of the previous episode i.e., checking the ticket of the previous person, and also the current action taken by the ticket checking agent does not effect the next action to be taken in this regard. Also a robot while checking the seals on the

bottles on an assembly line, also works in an episodic environment as checking the seal of every new bottle is a new and independent episode.

In sequential environments there are no episodes and **the previous actions could affect the current decision or action and further, the current action could effect the future actions or decisions**. The task environment of “An automated public road transport driver” is an example of a sequential environment as any decision taken or action performed by the driver agent may have long consequences.

Comparatively, working in an episodic environment is much simpler for an agent than in a sequential environment, because, in the former case, previous actions do not effect the current actions of the agent. Also, while taking current action, the agent need not think of its future effects.

2.4.3.5 Single-agent vs. Multi-agent Environment

A single-agent environment is the simplest possible environment as the agent is the only active entity in the whole environment and it does not have to synchronize with the actions or activities of any other agent. Some of the examples of a single-agent environment are *boundary following robot, a crossword puzzle solving agent* etc.

If there is a possibility of other agents also existing in addition to the agent being described then the task environment is said to **be multi-agent environment**. In a multi-agent environment, the scenario becomes quite complex as the agent has to keep track of the behavior and the actions of the other agents also. There can be two general scenarios, one in which all the agents are working together to achieve some common goal, i.e., to perform a collective action. Such a type of environment is called **cooperative multi-agent environment**.

Also, it may happen that all or some of the agents existing in the environment are competing with each other to achieve something (for example, in a tic-tac-toe game or in a game of chess both of which are two agent games, each player tries to win by trying to predict the possible moves of the other agent), such an environment is called **competitive multi-agent environment**.

An **important issue in a multi-agent environment** is to determine whether other entities existing in the environment have to be treated as *passive objects or agents*. Normally, if an entity tries to optimize its performance which have an effect on the agent in question then that entity is treated as an agent, but there is no fixed rule to decide this fact. Also, **another very important issue** in a multi-agent environment is to decide how the **communication between different agents** will occur.

2.4.3.6 Discrete vs. Continuous Environment

The word discrete and continuous are related to the way time is treated, i.e., whether time is divided into slots (i.e., discrete quantities) or it is treated as a continuous quantity (continuous).

For example, the task environment of a “boundary following robot” or a “chess playing agent” is a discrete environment because there are finite number of discrete states in which an agent may find itself. On the other hand, *an automated public transport driver* agent’s environment is a continuous one, because, the actions of the agent itself as well as the environment around it are changing continuously. In the case of a driver agent, the values received through the sensor may be at discrete intervals but generally the values are received so frequently that practically the received values are treated as a stream of continuous data.

2.4.4 Some Examples of Task Environments

- (i) Planning crossword puzzle
- (ii) Medical diagnosis
- (iii) Playing tic-tac-toe
- (iv) Playing chess
- (v) Driving automobile

2.4.4.1 Crossword Puzzle

As we have already discussed the environment is *single agent* and *static* which makes it simple as there are no other players, i.e., agents and the environment does not change. But the environment is *sequential* in nature, as the current decision will affect all the future decisions also. On the simpler side, the environment of a crossword puzzle is *fully observable* as it does not require remembering of some facts or decisions taken earlier and also the environment is *deterministic* as the next state of the environment fully depends on the current state and the current action taken by the agent. Time can be divided into *discrete quanta* between moves and so we can say that the environment is discrete in nature.

2.4.2.2 Medical Diagnosis

The first problem in the task environment of an agent performing a medical diagnosis is to decide whether the environment is to be treated as a single-agent or multi-agent one. If the other entities like the patients or staff members also have to be viewed as agents (obviously based on whether they are trying to maximize some performance measure e.g., Profitability) then it will be a *multi-agent* environment *otherwise* it will be a *single* agent environment. To keep the discussion simple, in this case, let us choose the environment as *a single agent* one.

We can very well perceive that the task environment is *partially observable* as all the facts and information needed may not be available readily and hence, need to be remembered or retrieved. The environment is *stochastic* in nature, as the next state of the environment may not be fully determined only by the current state and current action. Diagnosing a disease is stochastic rather than deterministic.

Also the task environment is partially *episodic* in nature and partially *sequential*. The environment is episodic because, each patient is diagnosed, irrespective of the diagnoses of earlier patients. The environment is sequential, in view of the fact that for a particular patient, earlier treatment decisions are also taken into consideration in deciding further treatment. Furthermore, the environment may be treated as *continuous* as there seems to be no benefit of dividing the time in discrete slots.

2.4.4.3 Playing Tic-tac-toe

As it is a two-player game so obviously the environment is *multi-agent*. Moreover, both the players try to maximize their chances of winning, so it is a *competitive* one. Further, the environment is a *fully observable* environment. The environment here is *neither fully deterministic nor stochastic* as the environment is deterministic except for one fact that action of the other agent is unpredictable, so the environment is *strategic*. Obviously the environment is *semi-dynamic* as the environment itself does not change but the agents performance score does change with time. As time may be divided into discrete slots between moves so the environment may be viewed as *discrete*. Further, the current decisions while making a move affect all the future decisions also, therefore, the environment is *sequential* also.

2.4.4.4 Playing Chess

The environment of a chess-playing agent is also the same as is that of an agent-playing tic-tac-toe. It is also a two-player i.e., **multi-agent** game. **Some people treat the environment as fully observable** but actually for some of the rules require remembering the game history so the full environment is not observable from the current state of the chess board. Thus, **strictly speaking the environment is partially-observable**. Further, using the same arguments as given in case of tic-tac-toe, the environment is **strategic, semi-dynamic, discrete and sequential** in nature.

2.4.4.5 Automobile Driver Agent

We have already discussed one subclass of a general driver agent in detail, viz., *an automated public road transport driver* which may include a bus driver, taxi driver or auto driver etc. A driver agent might also be christened as a *cruise control agent*, which may include other types of transport like water transport or air transport also with some variations in their environments.

Coming back to *an automated public road transport driver*, we can see that this is one of the most complex environments discussed so far. So the environment is **multi-agent and partially observable** as it is not possible to see and assume what all other agents i.e., other drivers are doing or thinking. Also the environment is fully **dynamic**, as it is changing all the time as the location of the agent changes and also the locations of the other agents change. The environment is **stochastic** in nature as anything unpredictable can happen, which may not be perceived exactly as a result of the current state and the actions of various agents. The environment is **sequential** as any decision taken by the driver might affect or change the whole future course of actions. Also the time is **continuous** in nature although the sensors of the driver agent might work in discrete mode.

2.5 THE STRUCTURE OF AGENTS

There are two parts of an agent or its structure:

- A (hardware) device with sensors and actuators in which that agent will reside, called the *architecture* of the agent.
- An agent program that will convert or map the percepts in to actions.

Also, the agent program and its architecture are related in the sense that for a different agent architecture a different type of agent program is required and vice-versa. For example, in case of a *boundary following robot*, if the robot does not have the capability of sensing adjacent cells to the right, then the agent program for the robot has to be changed.

Next, we discuss different categories of agents, which are differentiated from each other on the basis of their agent programs. Capability to write efficient agent programs is the key to the success for developing efficient rational agents. Although the table driven approach (in which an agent acts on the basis of the set of all possible percepts by storing these percepts in tables) to design agents is possible yet the approach of developing equivalent agent programs is found much more efficient.

Next we discuss some of the general categories of agents based on their agents programs:

- *SR (Simple Reflex) agents*
- *Model Based reflex agents*

- *Goal-based agents*
- *Utility based agents*

2.5.1 Simple Reflex (SR) Agents

These are the agents or machines that have no internal state (i.e., they don't remember anything) and simply react to the current percepts in their environments. An interesting set of agents can be built, the behaviour of the agents in which can be captured in the form of a simple set of functions of their sensory inputs. One of the earliest implemented agent of this category was called *Machina Speculatrix*. This was a device with wheels, motor, photo cells and vacuum tubes and was designed to move in the direction of light of less intensity and was designed to avoid the direction of the bright light. **A boundary following robot is also an SR agent.** For an automobile-driving agent also, some aspects of its behavior like applying brakes immediately on observing either the vehicle immediately ahead applying brakes or a human being coming just in front of the automobile suddenly, show the simple reflex capability of the agent. Such a simple reflex action in the agent program of the agent can be implemented with the help of simple condition-action rules.

For example : **IF** a human being comes in front of the automobile suddenly
THEN apply breaks immediately.

Although implementation of SR agents is simple **yet on the negative side** this type of agents have very limited intelligence because **they do not store or remember anything**. As a consequence they cannot make use of any previous experience. In summary, they do not learn. Also **they are capable of operating correctly only if the environment is fully observable**.

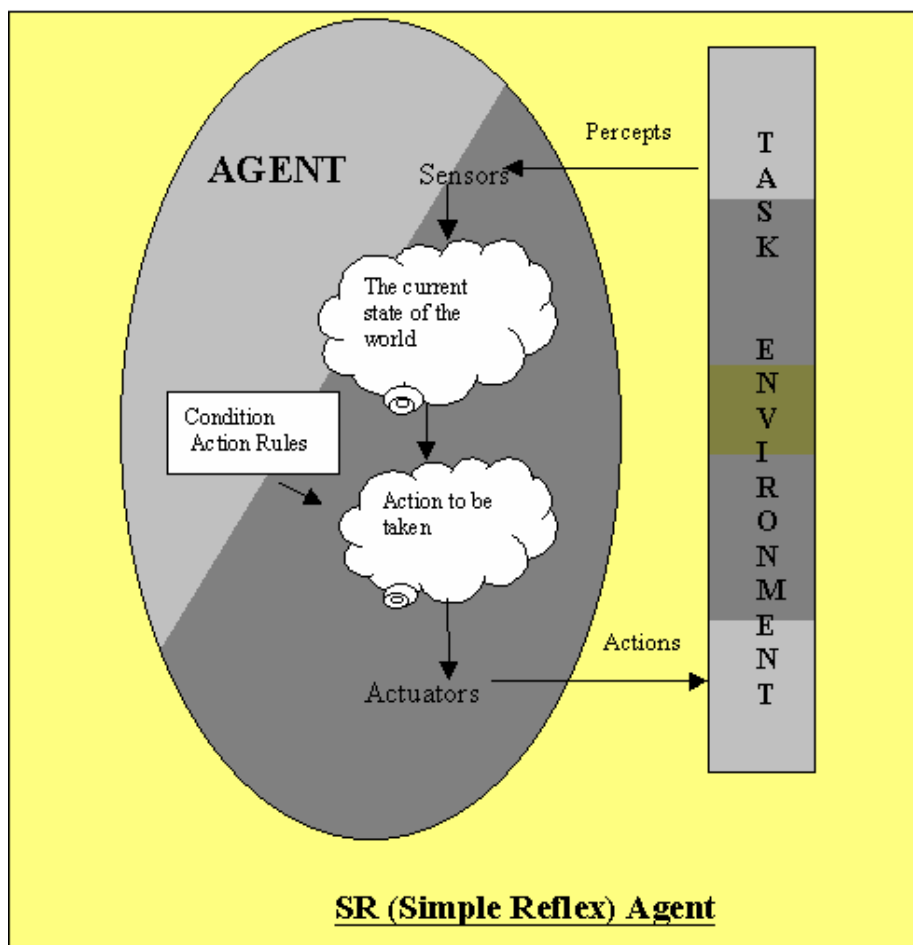


Figure 2.3: SR (Simple Reflex) Agent

2.5.2 Model Based Reflex agents

Simple Reflex agents are not capable of handling task environments that are not fully observable. In order to handle such environments properly, in addition to reflex capabilities, the agent should, maintain some sort of internal state in the form of a function of the sequence of percepts recovered up to the time of action by the agent. Using the percept sequence, the internal state is determined in such a manner that it reflects some of the aspects of the unobservable environment. Further, in order to reflect properly the unobserved environment, the agent is expected to have a model of the task environment encoded in the agent's program, where the model has the knowledge about–

- (i) the process by which the task environment evolves independent of the agent and
- (ii) effects of the actions of the agent have on the environment.

Thus, in order to handle properly the partial observability of the environment, the agent should have a model of the task environment in addition to reflex capabilities. Such agents are called **Model-based Reflex Agents**.

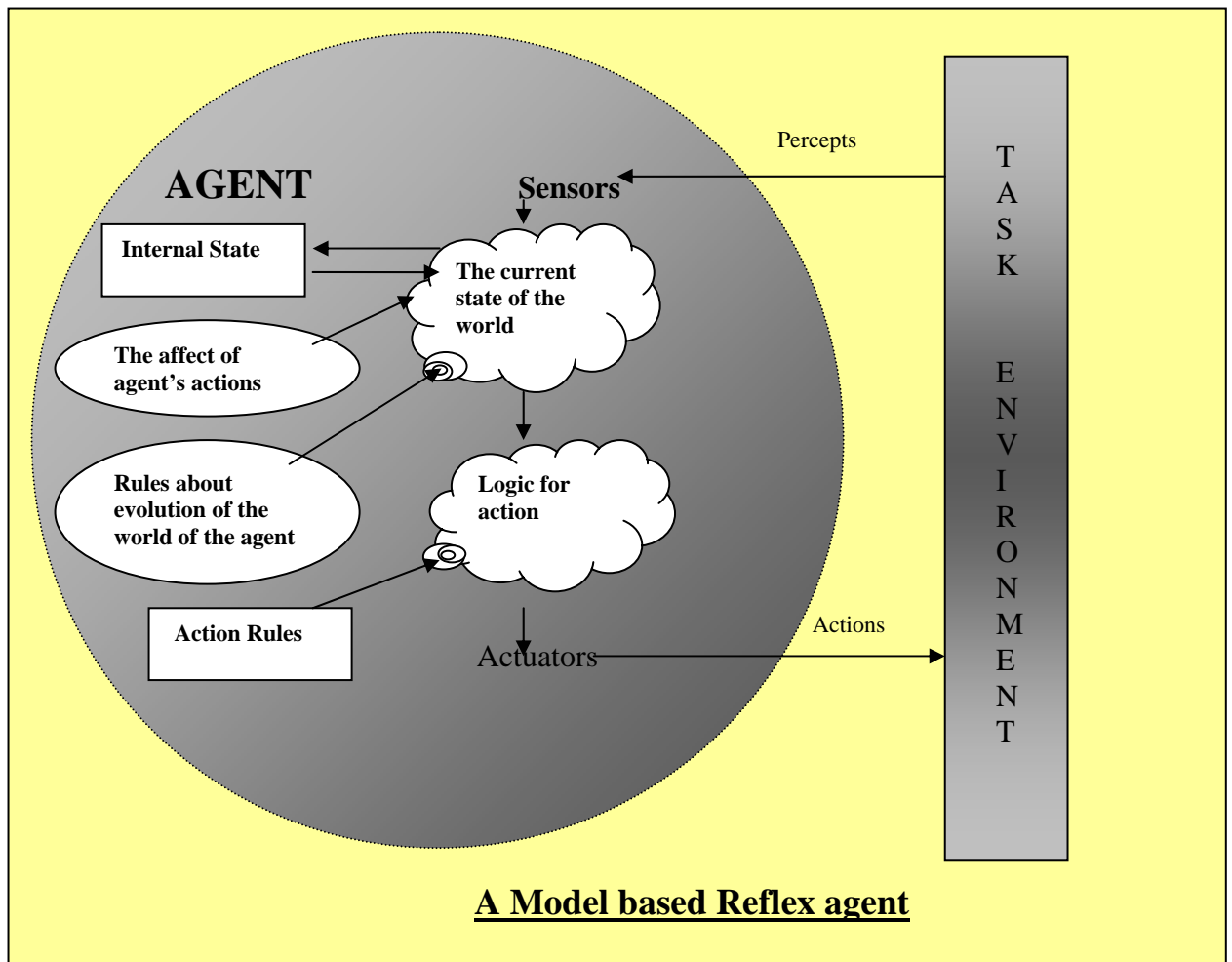


Figure 2.4: A Model based Reflex agent

2.5.3 Goal Based Agents

In order to design appropriate agent for a particular type of task, we know the nature of the task environment plays an important role. Also, it is desirable that the complexity of the agent should be minimum and just sufficient to handle the task in a particular environment. In this regard, first we discussed the simplest type of agents,

viz., Simple Reflex Agents. The action of this type of agent is decided by the current precept only. Next, we discussed the Model-Based Reflex Agents, for which an action is decided by taking into consideration not only the latest precept, but the whole precept history summarized in the form of internal state. Also, action for this type of agent is also decided by taking into consideration the knowledge of the task environment, represented by a model of the environment and encoded into the agent's program. However, in respect of a number of tasks, even this much knowledge may not be sufficient for appropriate action. For example, when we are going from city A to city B, in order to take appropriate action, it is not enough to know the summary of actions and path which has taken us to some city C between A and B. We also have to remember the goal of reaching to city B.

Goal based agents are **driven by the goal** they want to achieve, i.e., **their actions are based on the information regarding their goal, in addition to, of course, other information in the current state**. This goal information is also a part of the current state description and it describes everything that is desirable to achieve the goal. As mentioned earlier, an example of a goal-based agent is an agent that is required to find the path to reach a city. In such a case, if the agent is *an automobile driver agent*, and if the road is splitting ahead into two roads then the agent has to decide which way to go to achieve its goal of reaching its destination. Further, if there is a crossing ahead then the agent has to decide, whether to go straight, to go to the left or to go to the right. In order to achieve its goal, the agent needs some information regarding the goal which describes the desirable events and situations to reach the goal. The agent program would then use this goal information to decide the set of actions to take in order to reach its goal.

Another desirable capability which a good goal based agent should have is that if an agent finds that a part of the sequence of the previous steps has taken the agent away from its goal then it should be able to retract and start its actions from a point which may take the agent toward the goal.

In order to take appropriate action, decision-making process in goal-based agents may be simple or quite complex depending on the problem. Also, **the decision-making required by the agents of this kind needs some sort of looking into the future**. For example, it may analyze the possible outcome of a particular action before it actually performs that action. In other words, we can say that ***the agent would perform some sort of reasoning of if-then-else type***, e.g., an automobile driver agent having one of its goals as not to hit any vehicle in front of it, when finds the vehicle immediately ahead of it slowing down may not apply brakes with full force and in stead may apply brakes slowly so that the vehicles following it may not hit it.

As the goal-based agents may have to reason before they take an action, these agents might be slower than other types of agents but will be more flexible in taking actions as their decisions are based on the acquired knowledge which can be modified also. Hence, **as compared to SR agents** which may require rewriting of all the condition-action rules in case of change in the environment, the goal-based agents can adapt easily when there is any change in its goal.

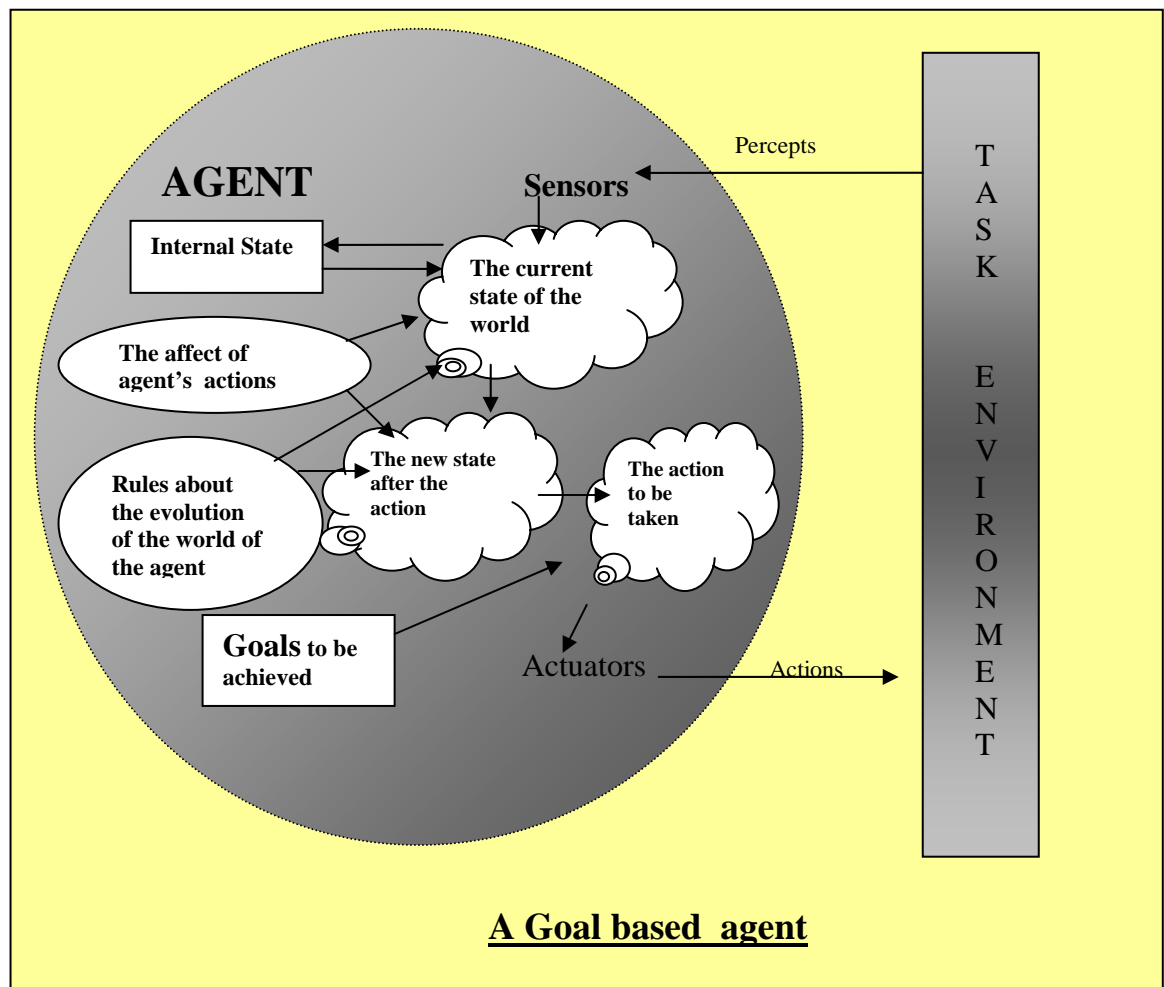


Figure 2.5: A Goal based agent

2.5.4 Utility Based Agents

Goal based agent's success or failure is judged in terms of its capability for achieving or not achieving its goal. A goal-based agent, for a given pair of environment state and possible input, only knows whether the pair will lead to the goal state or not. Such an agent will not be able to decide in which direction to proceed when there are more than one conflicting goals. Also, in a goal-based agent, there is no concept of partial success or somewhat satisfactory success. Further, if there are more than one methods of achieving a goal, then no mechanism is incorporated in a Goal-based agent of choosing or finding the method which is faster and more efficient one, out of the available ones, to reach its goal.

A more general way to judge the success or happiness of an agent may be, through assigning to each state a number as an approximate measure of its success in reaching the goal from the state. In case, the agent is embedded with such a capability of assigning such numbers to states, then it can choose, out of the reachable states in the next move, the state with the highest assigned number, out of the numbers assigned to various reachable states, indicating possibly the best chance of reaching the goal.

It will allow the goal to be achieved more efficiently. Such an agent will be more useful, i.e. will have more utility. A utility-based agent uses a **utility function**, which maps each of the world states of the agent to some degree of success. If it is possible

to define the utility function accurately, then the agent will be able to reach the goal quite efficiently. Also, a utility-based agent is *able to make decisions in case of conflicting goals*, generally choosing the goal with higher success rating or value. Further, in environments with multiple goals, the utility-based agent quite likely chooses the goal with least cost or higher utility goal out of multiple goals.

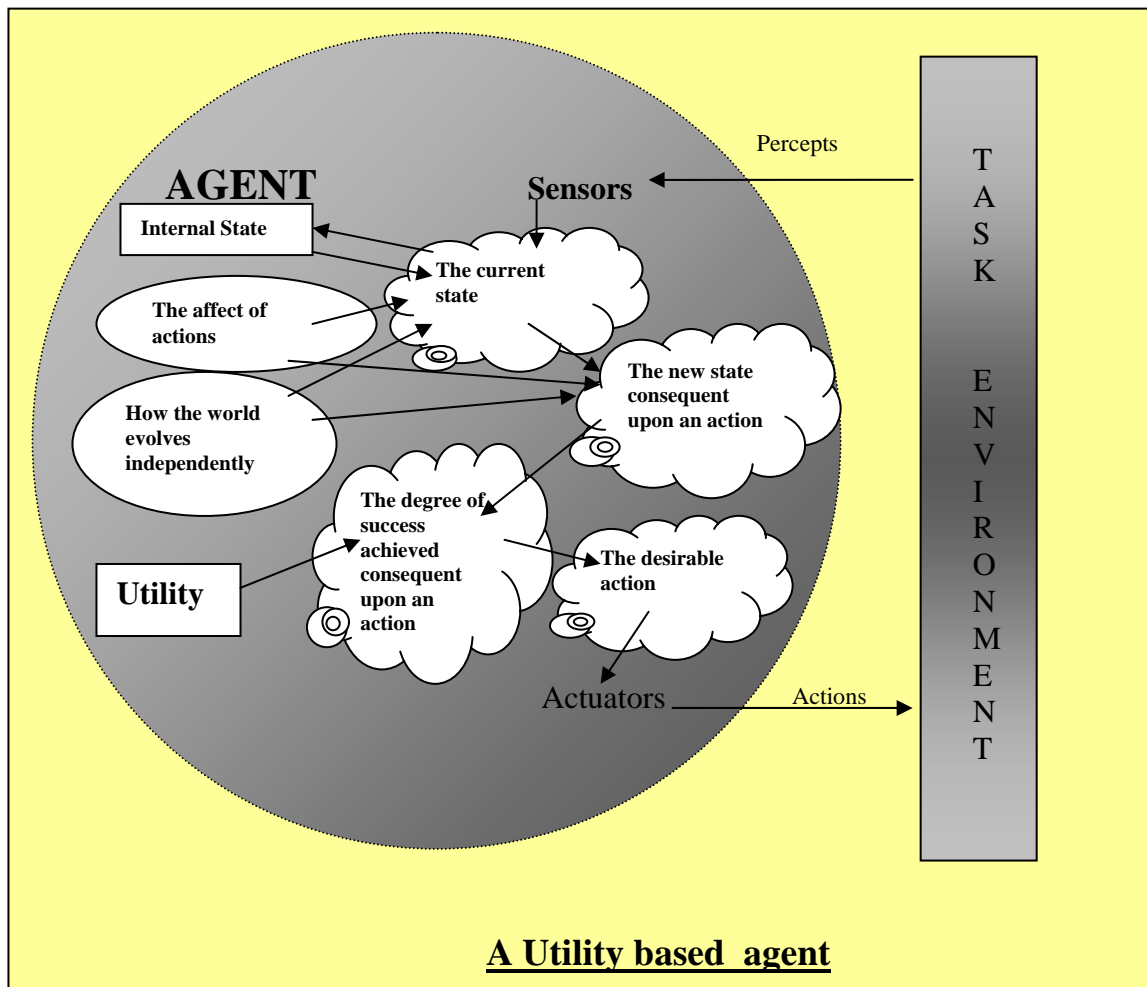


Figure 2.6: A Utility based agent

2.5.5 Learning Agents

It is not possible to encode all the knowledge in advance, required by a rational agent for optimal performance during its lifetime. This is specially true of the real life, and not just theoretical, environments. These environments are **dynamic** in the sense that the environmental conditions change, not only due to the actions of the agents under considerations, but due to other environmental factors also. For example, all of a sudden a pedestrian comes just in front of the moving vehicle, even when there is green signal for the vehicle. In a multi-agent environment, all the possible decisions and actions an agent is required to take, are generally unpredictable in view of the decisions taken and actions performed simultaneously by other agents. Hence, **the ability of an agent to succeed in an uncertain and unknown environment depends on its learning capability** i.e., its capability to change approximately its knowledge of the environment. For an agent with learning capability, some initial knowledge is coded in the agent program and after the agent starts operating, it learns from its actions the evolving environment, the actions of its competitors or adversaries etc. so as to improve its performance in ever-changing environment. If approximate learning component is incorporated in the agent, then the knowledge of the agent gradually

increases after each action starting from its initial knowledge which was manually coded into it at the start.

Conceptually the learning agent consists of four components:

- (i) **Learning Component:** It is the component of the agent, which on the basis of the percepts and the feedback from the environment, gradually improves the performance of the agent.
- (ii) **Performance Component:** It is the component from which all actions originate on the basis of external percepts and the knowledge provided by the learning component.

The design of learning component and the design of performance element are very much related to each other because a learning component is of no use unless the performance component can be designed to convert the newly acquired knowledge into better useful actions.

- (iii) **Critic Component:** This component finds out how well the agent is doing with respect to a certain fixed performance standard and it is also responsible for any future modifications in the performance component. ***The critic is necessary to judge the agent's success with respect to the chosen performance standard, specially in a dynamic environment. For example,*** in order to check whether a certain job is accomplished, the critic will not depend on external percepts only but it will also compare the current state to the state, which indicates the completion of that task.

- (iv) **Problem Generator Component:** This component is responsible for suggesting actions (some of which may not be optimal) in order to gain some fresh and innovative experiences. Thus, ***this component allows the agent to experiment a little*** by traversing sometimes uncharted territories by choosing some new and suboptimal actions. This may be useful, because the actions which may seem suboptimal in a short run, may turn out to be much better in the long run.

In the case of *an automobile driver agent*, this agent would be of little use if it does not have learning capability, as the environment in which it has to operate is totally dynamic and unpredictable in nature. **Once the automobile driver agent starts operating, it keeps on learning from its experiences, both positive and negative.** If faced with a totally new and previously unknown situation, e.g., encountering a vehicle coming from the opposite direction on a one-way road, the problem generator component of the driver agent might suggest some innovative action to tackle this new situation. Moreover, the learning becomes more difficult in the case of an automobile driver agent, because the environment is only partially observable.

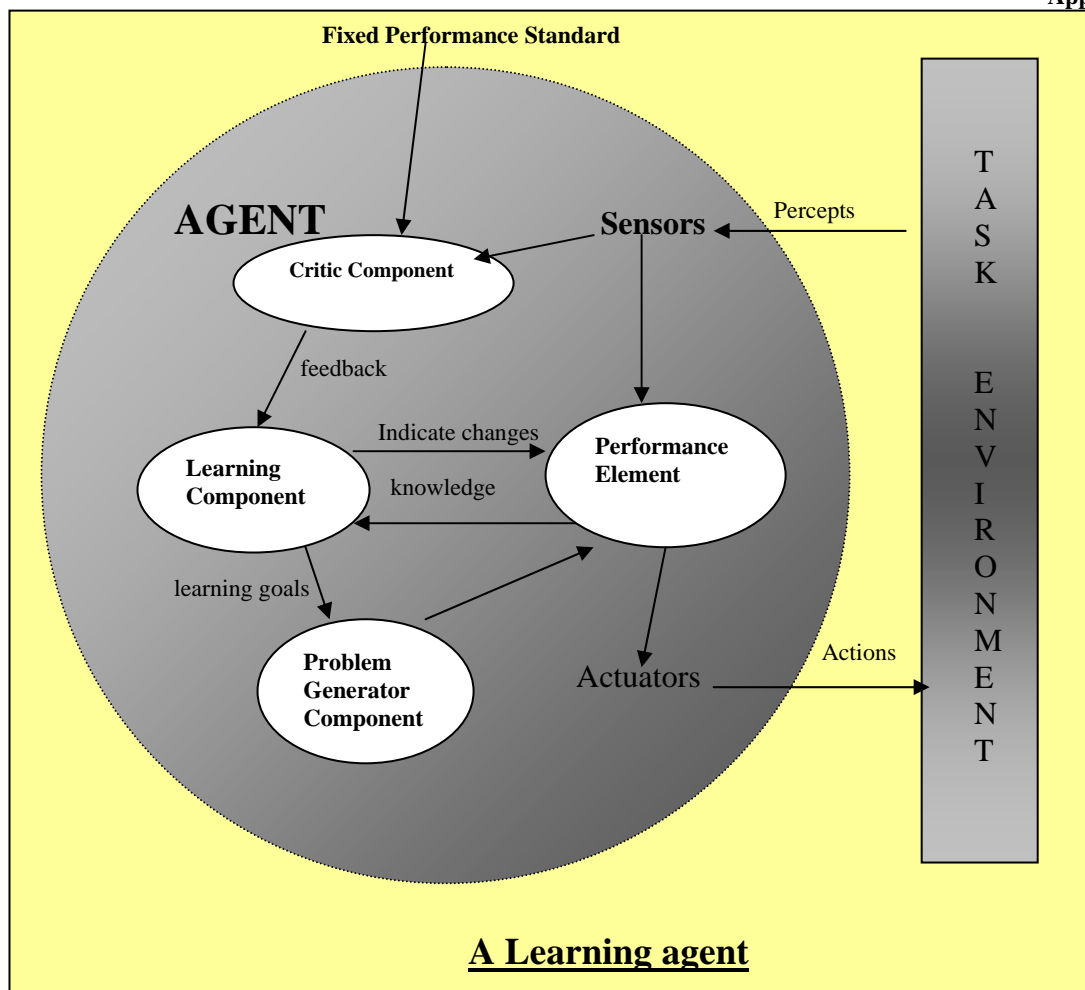


Figure 2.7: A Learning agent

2.6 Different Forms of Learning in Agents

The purpose of embedding learning capability in an agent is that it should not depend totally on the knowledge initially encoded in it and on the external percepts for its actions. The agent learns by evaluating its own decisions and/or making observations of new situations it encounters in the ever-changing environment.

There may be various criteria for developing learning taxonomies. The criteria may be based on –

- The type of knowledge learnt, e.g., concepts, problem-solving or game playing,
- The type of representation used, e.g., predicate calculus, rules or frames,
- The area of application, e.g., medical diagnosis, scheduling or prediction.

We mention below another classification, which is independent of the representation method and knowledge representation used. Under this classification scheme, there are five learning methods:

- (i) Rote learning or memorization
- (ii) Learning through instruction
- (iii) Learning by analogy
- (iv) Learning by induction and
- (v) Learning by deduction.

Rote learning is the simplest form of learning, which involves least amount of inferencing. In this form of learning, the knowledge is simply copied in the knowledge base. This is the learning, which is involved in memorizing multiplication tables.

Next type of learning is *learning through instruction*. This type of learning involves more inferencing because the knowledge in order to be operational should be integrated in the existing knowledge base. This is the type of learning that is involved in the learning of a pupil from a teacher.

Learning by analogy involves development of new concepts through already known similar concepts. This type of learning is commonly adopted in textbooks, where some example problems are solved in the text and then exercises based on these solved examples are given to students to solve. Also, this type of learning is involved when, on the basis of experience of driving light vehicles, one attempts to drive heavy vehicles.

Learning by induction is the most frequently used form of learning employed by human being. This is a form of learning which involves inductive reasoning — a form of reasoning under which a conclusion is drawn on the basis of a large number of positive examples. For example, after seeing a large number of cows, we conclude a cow has four legs, white colour, two horns symmetrically placed on the head etc. Inductive reasoning, though usually leads to correct conclusions, yet the conclusions may not be *irrefutable*.

For instance, in the case of the concept of cow as discussed above, we may find a black cow, or we may find a three-legged cow who has lost one leg in an accident or a single-horn cow.

Finally, we discuss *deductive learning*, which is based on deductive inference — an *irrefutable* form of reasoning. By *irrefutable* form of reasoning we mean that the conclusion arrived at through deductive (i.e., any *irrefutable*) reasoning process is always correct, if the hypotheses (or given facts) are correct. This is the form of reasoning which is dominantly applied in Mathematics.

Inductive learning occupies an important place in designing the learning component of an agent. The learning in an agent is based on—

- The subject matter of learning, e.g., concepts, problem-solving or game playing etc.
- The representation used, predicate calculus, frame or script etc.
- The critic, which gives the feedback about the overall health of the agent.

Learning based on feedback is normally categorized as:

- Supervised learning
- Unsupervised learning
- Reinforcement Learning.

Supervised Learning: *It involves a learning function from the given examples of its inputs and outputs.* Some of the examples of this type of learning are:

- Generating useful properties of the world around from the percepts
- Mapping from conditions regarding the current state to actions
- Gathering information about the way world evolves.

Unsupervised Learning: In this type of learning, the pair of inputs and corresponding expected outputs are not available. Hence, the learning system, on its own, has to find

relevant properties from the otherwise unknown objects. For example, finding shortest path, without any prior knowledge, between two cities in a totally unknown country.

Reinforcement (Rewards) Learning: *In many problems, the task or the problem is only realized, but cannot be stated.* Further, the task may be an ongoing one. The user expresses his or her satisfaction or dissatisfaction regarding the performance of the agent by occasionally giving the agent positive or negative rewards (i.e., reinforcements). ***The job of the agent is to maximize the amount of reward (i.e., reinforcement) it receives.*** In case of a simple goal achieving problem, the agent can be rewarded positively when it achieves the goal and punished every time it fails to do so.

In this kind of task environment, an action policy is needed to maximize reward. But *sometimes in case of ongoing and non-terminating tasks the future reward might be infinite, so it is difficult to decide how to maximize it.* In such a scenario, a method of progressing ahead is to discount future rewards beyond a certain factor. i.e., the agent may prefer rewards in the immediate future to those in the distant future.

Learning action policies in environments in which rewards depend on a sequence of earlier actions is called **delayed-reinforcement learning**.

2.7 SUMMARY

In this unit, the concept of 'Intelligent Agent' is introduced and further various issues about intelligent agents are discussed. Some definitions in this context are given in section 2.2. The next section discusses the concept of rationality in context of agents. Section 2.4 discusses various types of task environments for agents. Next section discusses structure of an agent. Finally, section 2.6 discusses various forms of learning in an agent.

2.8 SOLUTIONS/ANSWERS

Ex. 1) For L_1

The left upper cell is unoccupied, therefore, $C_1=0$, $C_2=0=C_3=C_4$ and $C_8=0$
However, as C_5 , C_6 and C_7 do not exist hence are assumed to be occupied.
Thus, the sense-vector is (0, 0, 0, 0, 1, 1, 1, 0)

For L_2

$C_1=1=C_8$ and $C_2=C_3=C_4=C_5=C_6=C_7=0$
Thus, the sense-vector is (1, 0, 0, 0, 0, 0, 0, 1)

2.9 FURTHER READING

1. Russell S. & Norvig P, *Artificial Intelligence: A Modern Approach* (Second Edition), (Pearson Education, 2003).

UNIT 1 A.I. LANGUAGES-1: LISP

Structure	Page Nos.
1.0 Introduction	5
1.1 Objectives	6
1.2 Basics of LISP	6
1.3 Data Structures and Data Values	9
1.4 The EVAL Function and Some Evaluations	10
1.5 Evaluation of Primitive Functions	13
1.6 Primitive List Manipulation Functions	14
1.7 Built-in Predicates	16
1.8 Logical Operators: AND, OR and NOT	18
1.9 Evaluation of Special Forms involving DEFUN and COND	18
1.10 The special forms DO and LET	20
1.11 Input/Output Primitives	23
1.12 Recursion in LISP	24
1.13 Association List and Property List	25
1.14 Lambda Expression, APPLY, FUNCALL and MAPCAR	29
1.15 Symbol, Object, Variable, Representation and Dotted Pair	31
1.16 Destructive Updates, RPLACE, RPLACD and SETF	34
1.17 Arrays, Strings and Structures	35
1.18 Summary	38
1.19 Solutions/Answers	39
1.20 Further Readings	41

1.0 INTRODUCTION

The task of solving problems using computer as a tool, in general, is a quite a comprehensive task. Ever since the use of computers in solving problems, it has been found that the solving of problems can be facilitated by using appropriate style/paradigm for a given type of problem and using a language designed and developed according to the basic principles of the style.

Some of the well-known programming languages like C support **imperative style** of programming for solving problems with the help of a computer. The major feature of imperative style is that the proposed solution is expressed in terms of *variables, declarations, expressions and commands*. *Declarations* assign names to locations in the memory and associate types with the values. *Commands* may be thought of as names for actions, that are required to be executed by a computing system, mainly to change values stored in the memory locations. Commands are generally executed in the order, from top to bottom, as these appear in the program, though through conditional and unconditional jumps, flow of execution can be changed. One of very important concept in imperative style of programming is that of *the state (of memory)*, i.e., the set of values assigned to various locations in the memory at a particular point of time.

In this style of programming, the programmer is required to think and express proposed solution (to a problem under consideration) in terms of the basic actions that a machine is to carry out. For complex problems, the task of the programmer becomes more and more difficult with increase in the complexities of the problems to be solved. In view of this difficulty, *alternative paradigms or styles* for solving problems have evolved since almost the beginning of problem solving with computers. The language LISP, we are going to discuss in this unit, supports an alternative paradigm, namely, **functional paradigm**, for solving problems with the help of a computer. In

LISP has jokingly been called '*the most intelligent way to misuse a computer*'. I think that description is a great compliment, because it transmits the full flavour of liberation; it has assisted a number of most gifted fellow humans in thinking previously impossible thoughts'

Edsger W. Dijkstra

functional paradigm, the solution is primarily considered as an exercise in *defining functions* and *applying functions* either recursively or through composition. Common LISP is not a pure functional style programming language. *It has some features of imperative style also.* For example, the symbol SETQ allows assigning names to memory locations and storing values in the memory locations. But, Lisp incorporates features dominantly of functional programming style.

1.1 OBJECTIVES

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

- explain what is *functional paradigm* of problem solving using a computing system;
- explain how functional paradigm is different from the *normally used imperative paradigm*;
- discuss LISP as a language having features of functional paradigm;
- enumerate and discuss characteristic features of LISP;
- enumerate and discuss data types and data structures of LISP;
- use these types and structures in defining data required to solve a problem under consideration ;
- explain the role of the inbuilt function EVAL;
- explain and use various built-in general functions, list manipulation functions, predicates, and logical operators;
- explain the role of the special forms viz. Defun, COND, DO, LET and use these in writing complex programs in LISP;
- explain and use the various input/output primitives available in LISP;
- explain the significance of the concept of *recursion* in solving problem, and how recursion is achieved in LISP;
- explain the role of *association lists* and *property lists* in developing database and use these in defining objects and their attributes/properties;
- explain the role of a *symbol* as a *variable* and further should be able to explain the concepts of *bound variable* and *free variable*;
- explain the concept of dotted pairs and be able to use the concept in representing lists in LISP, and
- Write complex LISP programs.

1.2 BASICS OF LISP

The **programming language LISP** takes its name from **List Processing**. LISP* was developed by **John McCarthy**, during **1956-58** and was implemented during 1959-62. LISP has a number of dialects. However, with the development of COMMON LISP in the 1980s and its acceptance by a large number of system implementers and manufactures, it has become almost standard for LISP users. We also shall be using and discussing COMMON LISP only.

LISP has been one of the most popular languages for AI applications. **LISP was specifically designed for A.I. applications**, and we have already mentioned that AI applications involve *symbolic processing*, instead of mere *numeric processing*. Also, AI systems are generally large and complex. Their development requires that the implementation language and support environment provide *flexibility*, *rapid prototyping* and *good debugging tools*. On all these counts, LISP wins over all other

*LISP is based on a formal system called λ -calculus (Lambda-calculus) originally proposed by Alonzo church and who developed it later alongwith Stephen Kleene as a foundation for Mathematics.

programming languages. Hence, the language was used in its earliest applications for writing programs which performed symbolic differentiation, integration and mathematical theorem proving. Later applications mainly written in LISP, include expert systems and programs for *common sense reasoning, natural language interfaces, education and intelligent support systems, learning, speech and vision*. LISP has been found quite useful for the purpose of systems programming to the extent that LISP machines have been developed in which whole of the programming from top to bottom is in LISP. In **LISP machines**, which are personal computers, the operating system, the user utility programs, the compilers, and the interpreters are all written in LISP.

We summarise below the main characteristics of LISP.

- (i) It is an **applicative/functional language**. *In a functional language the primary effect is achieved by applying functions either recursively or through composition.* For example, in order to evaluate the arithmetic expression $(x*y) + (z - u)$ where the variable x, y, z and u have values respectively 9, 2, 8 and 6, the following LISP expression

$$(+ (* x y) (- z u))$$
evaluates to 20. For evaluating the expression, first the function '*' is applied to the values of x and y, next the function '-' is applied to the values of z and u and then recursively the function '+' is applied to the results of the applications.
- (ii) The above example also shows that **prefix notation is used in LISP**, i.e., the operator '*' comes before operands x and y. The prefix notation has the advantage that the operators (like +, -, * etc.) can be easily located in an expression.

In contrast, the programming languages introduced to us earlier like, FORTRAN and C are all **imperative languages**. A language is said to be *imperative which achieves its primary effect by changing the state of variables by assignment*. For example, to compute the arithmetic expression $(x * y) + (z - u)$, the programme code in *Pascal* would include a sequence of instruction like,

```
Product    = x * y;
Difference = z - u
Result = product + sum
```

It is easily seen that **the emphasis is on assigning values** to the variables, viz, product, difference and result.

- (iii) *The language LISP allows programs to be used as data and vice-versa.*

LISP has mainly one data structures viz list, in addition to the elementary data types: number and symbol. All expressions, whether data or programs, mainly are expressed in terms of lists.

The main advantage of this property of LISP is that the *declarative knowledge, i.e., information about properties of an object can be easily integrated with procedural knowledge, i.e., information about what actions to be performed*. The facility of uniform representation **in LISP is also useful in the sense that it allows us to write**

- LISP programs which can modify other programs (written in any language) including themselves.
- LISP program that can write entirely new LISP programs.
- AI programs that learn new tasks.

- (iv) LISP is a highly *modular* language and hence suitable for development of large software.
- (v) The LISP environment provides a facility called *Trace* by using which programs written in LISP can easily keep track of the various instructions that have been executed, the number of times each has been executed and the order in which the instructions have been executed.
- (vi) LISP, being based on the mathematical discipline of λ -calculus, is **the most well-defined of all the programming languages**. Hence, programs in LISP are more reliable. The well-definedness of a language is a very important issue as can be seen from the fact that due to a small error in FORTRAN program of the type quoted below and **FORTRAN environment's incapability to detect it, lead to the loss a spaceship**.

In FORTRAN, the statement

DO 12 I = 1,5

denotes the beginning of a loop, whereas the statement

DO 12 I = 1.5

is an assignment statement. Through the execution of the second statement, the value 1.5 is assigned to the variable DO12I, because blanks are ignored at all places in FORTRAN.

- (vii) **Comments in LISP** are given by using the character for semicolon i.e. ';' as the first symbol on the line which is to be treated as a comment. Sequence of characters on a given line after semi-colon, is treated as a comment.

We included this feature here because while explaining various features of LISP, we would be required to provide comments in LISP environment.

- (viii) **The types of the variables are not required to be declared** in the beginning as is done in imperative languages like, FORTRAN or C. Also a variable name say x, we can assume any type for the values of the variable within the same program or procedure.

- (ix) LISP is format-free. Any valid LISP expression, say

(x (yz) u)

can be written in any one of the following (or even other) formats:

$\left. \begin{array}{l} (x (\\ yz) \\ u \\) \end{array} \right\}$	or	$\left\{ \begin{array}{l} (x (y \\ z \\) \\ u) \end{array} \right.$
--	----	--

- (x) We should note that the **two parentheses** viz the left parenthesis denoted by '(' and the right parenthesis denoted by ')' **are two most important characters in LISP and must be used very carefully**. They are used to denote lists and for each left parenthesis, there is a right parenthesis for any valid LISP expression. In the light of the above fact, the expressions
xyx) or (fg

are illegal symbols or expressions.

Next most important character in LISP is quote, the role of which is explained after some time under (ii) of evaluation of S-expr.

- (xi) **Separator.** Blanks are used to separate S-exprs, i.e., any valid Lisp entity or object, specially numbers and symbols. Lists are always separated automatically from other S-exprs. **Comma is used for special purposes and not as a separator.**
- (xii) **List is defined recursively.** A list is a sequence of atoms and/or other lists enclosed within parentheses. Each of the following three expressions is an example of a valid list.

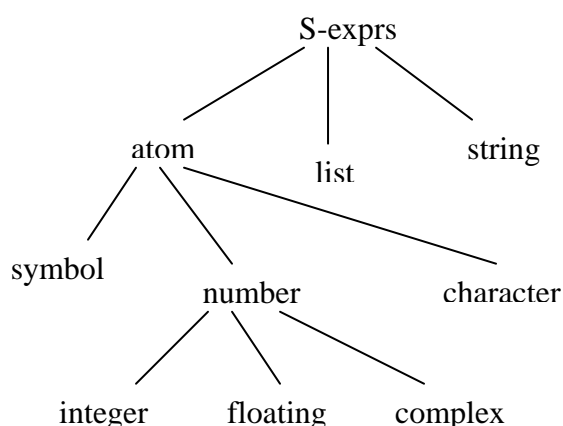
```
( 3 a ( c d ) )
( this is a list of only symbols)
( )
```

But the following expressions, one on each of the next three lines is not a valid list.

```
( b a 3
this – is – not – a – list
) a b c (
```

1.3 DATA STRUCTURES AND DATA VALUES

Notation for a *valid* object in LISP is called **S-expressions** or just *S-exprs* (**shorthand for Symbolic Expression**). Even *objects themselves* are also referred to sometime as S-exprs. **The main data types of LISP objects** and their interrelationships are shown in the following diagram:



As shown above, **an atom** is a number, a symbol or a character. **A number** is a sequence of digits, possibly, involving a dot and the letter E appropriately so that the value is either an integer or a real (decimal) number. The following are **examples of numbers**:

11.5E-3 -14.E3

A **symbol** is any string of characters which does not represent a number and does not include parentheses and quotes. **However, each of the expressions, ‘()’ and ‘nil’, represents an empty list, and, is both an atom as well as a list.**

Each of the following is a symbol

3 + 11 *Name* BLOCK#8

Note that 3 + 11 is not a number but a symbol. The reason for this is that any arithmetic expression involving at least one operator, when represented in LISP, has to be a list with first and the last characters as ‘(’ and ‘)’ respectively, and first element within the list being necessarily an operator. If we intend to represent the arithmetic expression 3 + 11 which is equivalent to 14, then it is represented as (+ 3 11).

String: A sequence of characters enclosed within double quotes is a string, e.g.,
“abc 2 @ string ”
is a string

However, the following is *not* a string

“square of two is four

because right double quote is missing.

1.4 THE EVAL FUNCTION

Initially LISP was designed as an *interpreted* language, though later compiler based versions also became available. In the interpreted mode, the prompt visible on the screen is the symbol ‘→’, which, of course, may be changed.

LISP environment provides an inbuilt function called *eval* or commonly known as read-eval-print loop. Any legal LISP object, i.e., **an S-expr** typed after the prompt is considered both as an S-expr and as an input to the read-eval-print loop. The S-expr is *evaluated* according to the rules to be explained and then printed. The value, so obtained, is also an S-expr, provided that the input is an S-expr. Else, error will be printed.

Next, we explain, in some detail, how LISP expressions are evaluated.

- (i) A **quoted expression** is evaluated to the expression obtained by removing the quote. For example,

The following expression

→ '(+ x (* y z))
evaluates to the expression

→ (+ x (* y z))

→ 'Colour
; where colour is a symbol we get respectively the
; following situations after printing
→ colour

- (ii) **For evaluating a symbol** say *colour*, first of all, some value or binding must have been associated at some stage before the current evaluation. And then evaluation of the symbol returns the associated value or binding. Suppose, earlier at some stage, the symbol *colour* is given the value RED, then, if give the input colour, i.e., if we have

→ colour
; then RED is returned, i.e., we get after evaluation

→ RED

- (iii) **Evaluating a number:** A number evaluates to itself
For example if the input is given as

→ 41
; then after evaluation the number 41 is returned.

- (iv) **Evaluating a function application**

A list of the form

(<function-name> <param1> <param2> <paramk>)

with <function-name> being an atom and the name of a function, and <parami> being an S-expr for each i, is called a **function application**. For evaluating a function application, first <param1> is evaluated to get arg1. Similarly <parami> is evaluated to get argi, first for i=2, then for i=3, and so on. Once all the arguments are evaluated then the function is applied to the arguments. Finally the value so obtained is printed. For example, if the following expression is given as input

→ (* 4 11)

as 4 evaluates to 4 and 11 evaluates to 11, therefore 44 is returned and printed. Next suppose, the symbol *x* is bound to 2 and *y* to 5 then for the input

→ (+ (* x 3) (- y 1))

first (* x 3) is evaluated, which in turn requires evaluation of symbol *x*. The symbol *x* being bound to 2 is evaluated as 2. Thus, the expression (* x 3) evaluates to 6. Similarly, the expression (- y 1) evaluates to 4 and finally the whole of the above expression evaluates to 10.

Next, assume *sum-sq* is a function defined in a program which returns the sum of the squares of its arguments and again suppose *x* is bound to 2 and *y* is bound to 5, then if we have the following expression is given as input

→ (sum-sq (* x 3) (- y 1))

the expression is evaluated and printed as

→ 52.

on the monitor

(v) Evaluating a special form

A list of the form

(<special-word> <param 1> <param 2> <param k>)

with <special-word> being a special word in LISP (*to be discussed*) and <parami> an S-expr, is called a special form. The evaluation of special form depends upon the special word. The parameters <parami> may or may not be evaluated depending upon the <special-word>.

Some of the well-known special words are: **defun**, **cond**, **do**, **quote**. We shall discuss evaluation of these special forms at appropriate place. The special form (*quote x*) is just equivalent to 'x and hence evaluates to x.

→ (quote (* 3 7));

evaluates to

→ (* 3 7)

Note that '(* 3 7) or (quote (* 3 7)) evaluates to (* 3 7) and not to 21

(vi) Evaluating a list of the form

(<non-atom> <param1> <paramk>),

where <non-atom> is an S-expr is evaluated as follows :

Each of the parameter <parami> is evaluated yielding arguments and then the S-expression <non-atom> is applied to the evaluated arguments. We shall discuss examples of such evaluations later.

- (vii) Certain atoms have preassigned meaning or evaluations as follows in LISP, therefore, should not be used for other purposes.

Atom	Associated meaning
t	logical value <i>true</i>
nil	denotes either logical value <i>false</i> or the empty list () depending on the context.

The special symbol Setq

Before coming to evaluation of functions in LISP we consider evaluation of special-form for the special word SETQ. It binds symbols to values and will be useful in explaining evaluation of other functional expressions. The special word *setq* takes two parameters, the first a *symbol* and the second an *S-expr*. The first parameter is not evaluated. *Actually SETQ may be taken as shorthand for SET QUOTE. And, as we have explained earlier, QUOTE EXPRESSION evaluates to EXPRESSION and not to the value of EXPRESSION. e.g. (QUOTE 3 + 5) evaluates to 3 + 5 and not to 8.* The second parameter is evaluated and the value so obtained is bound to the symbol represented by first parameter, e.g., the S-expr

(*setq x 27*) binds 27 to x and also the value 27 is returned
 (*setq x (* 3 5)*) binds 15 to x and returns the value 15,
 (*setq x'(* 3 5)*) binds the list (* 3 5) to x and also returns the list (*3 5) and not 15.

We have already mentioned that the action of *return* includes printing of the returned value. Also the action of *return* includes the fact that returned value can be utilised in further processing, e.g., the S-expr

(+ (*setq x 7*) (*setq y 3*))

not only binds x to 7 and y to 3 but also the values 7 and 3 respectively are used in further evaluation and

(+ (*setq x 7*) (*setq y 3*))

returns 10, in addition to binding x to 7 and y to 3.

Sometimes the pairs of arguments to several occurrences of SETQ are run together and given to a single SETQ. In such a situation, odd-numbered arguments are not evaluated and even-numbered arguments are evaluated. Further each even-numbered value is associated or bound to the immediately preceding (or bound to the immediately preceding) odd numbered argument, e.g., the S-expr

(*setq x' (1 2) y 7 z 11*)

binds the list (1 2) to x, 7 to y and 11 to z.
 associates or binds (12) to x, 7 to y and 11 to z. It may be noted that the list (12) and the number 12 is associated with x.

Ex 1: Explain the effect of execution of the following statements:

- (i) (* (+ (*setq x 5*) x) (+ (*setq y 7*) y))
 (ii) ' (- (+ *setq p 9*) (*setq s 3*)) (* p s))
-

1.5 EVALUATION OF PRIMITIVE FUNCTIONS

LISP has a number of basic functions. In this section, we discuss how S-exprs involving these primitive functions are evaluated.

We have already mentioned that LISP uses prefix notation for representing functional expressions.

We explain of evaluation of LISP objects through examples, preceded by some explanatory remarks, if required.

(i) Some Primitive Numeric Functions: As numeric evaluation is self-evident, therefore, the following table is sufficient for the purpose

Function call	Value	Comments
(+ 3 4 8 11 -2)	24	+ or plus can take any finite number of appropriate arguments
(plus 3 4 8 11 -2)	24	
(- 13 55)	-42	differences or - takes exactly two arguments
(difference 13 53)	-42	
(* 2 3 4)	24	times or * may take any finite number of appropriate arguments
(times 2 3 4)	24	
(/ 9 3)	3	quotient or / 'takes exactly two arguments
(quotient 9 3)	3	
(abs -5.7)	5.7	only one argument
(abs 5.7)	5.7	
(expt 3 2)	9	exponential function
(sqrt 4.0)	2.0	positive square-root
(max 8 11 9 7)	11	any finite number of appropriate arguments
(min 8 11 9 7)	7	
(truncate 14 4)	3	truncate returns the quotient in integer division neglecting the remainder;
(rem 14 4)	2	rem returns the remainder on division
(round 14.3)	14	round returns the integer nearest to its argument.
(round 14.6)	15	
(float 14)	14.0	integer argument; returns real.

Ex 2: Evaluate the following:

- (i) (+ (* 2 3 4) (- 8 9) (truncate (15 7)))
 (ii) (* (rem 17 6) (truncate 8 9) (max 5 9 11))

Ex 3: Write a function *division* which divides a number X by Y such that if Y = 0 then the function returns the symbol 'infinity' else it returns the quotient X/Y.

1.6 PRIMITIVE LIST MANIPULATION FUNCTIONS

We have already mentioned that the programming language LISP is mainly designed for symbolic processing though it may be used for numeric purposes also. Symbolic processing in LISP is mainly about manipulating lists. **Here we consider main list processing operations.**

- (i) **car:** It takes a list as an argument and returns the first element of the list. (`car '(d b c)`) returns the element `d`, (`car '((d b) c)`) returns `(d b)`. We should note that argument has to be a quoted list. This is in consonance with our earlier statement that for all functions denoted by an atom, the parameters are evaluated to return arguments for the function.

Thus for evaluating (car ' ((d b) c)), first of all its parameter viz. ' ((d b) c) is evaluated. Also we mentioned earlier any quoted expression is evaluated to its unquoted part i.e., ' ((d b) c) evaluates to the list ((d b) c). And the first element ((d b) c) is (d b) . Hence (d b) is returned.

Further

(car (car '((d b) c))) is evaluated to the atom d.
(car '(LISP IS AN AI LANGUAGE))

returns the symbol LISP.

If the statement (setq x '(a b c)) is followed by the statement (car x) then the symbol a is returned.

- (ii) **cdr** (pronounced as 'KUDDR') also takes a list as its argument and returns a list obtained from the given list by deleting its first element e.g.

```
(cdr '(d b c))      returns the list (b c)
(cdr '((d b) c))    returns the list (c)
(cdr '(LISP IS AN AI LANGUAGE)) returns the list
```

(IS AN AI LANGUAGE).

Also if the statement (setq x '(a b c)) is followed by the statement (cdr x) then the value returned is (b c).

But note (cdr 'x) returns error, because, 'x evaluates to x and not to the list (a b c) and the cdr of a symbol, in this case x, is not defined.

Sequence of CARs and CDRs

```
( car ( cdr ( cdr ( cdr ' ( person ( Name Raj )
                                ( Residence K-76 HauzKhas )
                                ( City New Delhi )
                                ) ) ) ) )
returns K - 76
```

LISP provides facilities to simplify notation for a sequence of cars and cdrs e.g. the S-expr

$$(car(car(cdr(car(cdr(cdr\ given-list)))))).$$

where given-list is bound to some valid list, can be simplified to

```
( caadaddr given-list)
```

or to any other S-expr like

```
( caar ( cdadr ( cdr given-list ) ) ).
```

Remark: The functions *car* and *cdr* take things apart. Next we describe three functions *cons*, *list* and *append* which put things together.

- (iii) **Cons:** takes two arguments, the first may be any (valid) S-expr but the second must be a list. Then *cons* returns a new list in which the first argument is the first element in the returned list followed by the elements of the list given as second argument, preserving the earlier order of occurrence in the second argument.

```
( Cons ' * ' ( 4 7 ) ) returns the list ( * 4 7 )
```

```
( Cons ' ( a b ) ' ( b c ) ) returns the list ( ( a b ) b c )
```

```
( Cons ' a nil ) returns the list ( a )
```

Also, if S-exprs (*setq* x ' (a b)) and (*setq* y ' (c d)) are followed by the S-expr (*Cons* y x) then the list ((c d) a b) is returned but (*Cons* ' y x) returns the list (y a b).

- (iv) **list:** may take any number of parameters, each of which is an S-expr, it evaluates the parameters and then the arguments so obtained are grouped into a list in the same order as the corresponding parameters are given initially, e.g., if the inputs (*setq* x ' (a b)) and (*setq* y ' intelligence) are followed by the input s-expr (*list* ' x y x ' knowledge ' y) then on evaluation of the last expr, we get the list (x intelligence (a b) knowledge y)

- (v) **Append:** the parameters of the function *append* can not be arbitrary S-exprs but must evaluate to lists. It removes the parenthesis from the arguments obtained by evaluating the parameters and puts all the lisp objects so obtained into a list. For example, if the S-exprs (*setq* x ' (a b)) and (*setq* y ' (c d)) are followed by (*append* x y ' ((a b))) then the last S-expr on evaluation returns (a b c d (a b)). If we try to evaluate (*append* ' x y) then an error is returned, because value of 'x is a symbol x and a symbol can not be an argument of *append*.

Next, we define some more built-in list processing functions, which are quite useful in writing LISP programs for solving problems requiring symbolic processing.

- (vi) **Reverse:** takes a list as its argument and reverses the top level elements of the argument e.g.

```
( reverse ' ( a b ( a b ) ( c d ) ) )
```

returns

```
(( c d ) ( a b ) b a )
```

- (vii) **Length:** again takes a list as its argument and returns the number of the top level elements, e.g.,
(*length* ' (a (a b) (c d) e)) returns the number 4.

- (viii) **Last:** again takes a list as argument and returns the last top-level element of the list, e.g.,

```
( last ' ( a b ( c d ) ) ) returns the list ( c d ).
```

- (ix) **Subst** (*stands for substitution*): takes three arguments, such that each occurrence of second argument in the third argument are replaced by the first argument. Second argument must be an atom, e.g.,

```
( subst ' A 'B ' ( D B A ) )
returns
( D A A )
Also
( subst ' ( A B ) 'C ' ( D B A C ) )
returns
( D B A ( A B ) )
```

- (x) **eval**: In some situations, we may need another evaluation, in addition to the evaluation provided by *read-eval-print loop*. The function eval is explained with following examples:

```
→ (setq x 'y)
→ (setq y z) ; then x evaluates to y but ( eval x ) evaluates to z.
```

Ex 4: Explain the sequence of steps of evaluation of the following LISP expression:
(length (append (setq x '(a b)) '(c d) (reverse (sublist 'x '(s t) '(u v x)))))

1.7 BUILT-IN PREDICATES

Predicates are functions which return nil or t depending upon the values of their arguments. The evaluation by some important predicates is explained in the following table:

Function call	Value Returned	Comments
(> 7 (+ 2 3))	t	normal 'greater than' relation
(< 7 3)	nil	normal 'less than' relation
(= (* 3 7) (+ 16 5))	t	' = ' tests equality of only numbers
(= equal '(one two) '(one two))	t	'equal' tests
(equal '(two one) '(one two))	nil	equality of any two S-exprs,
(equal ' (a b) (car ((a b) c)))	t	
(evenp (+ 4 7))	nil	returns t if arguments is an integer
(evenp (* 2 5))	t	and is even, else returns nil,
(numberp 2.1416)	t	returns t if parameter evaluates to a
(numberp 'x)	nil	number else evaluates to nil.

Further if we have (setq fifty 50) before the next S-expr then

```
( numberp fifty )          t
```

But
 (number 'fifty) nil
 (zerop 0) t

if we have (setq x 0) then

(zerop x) t

But
 (zerop 'x) nil
 (zerop .00001) nil

The predicate 'null': null returns t if its argument is nil else returns nil, e.g.,

(null ()) returns t
 (null 'man) returns nil
 (null ' (a b)) returns nil

The predicate 'member': *The predicate 'member' has a little different behaviour. It may not return t and/or nil. The predicate member tests an atom for the membership of a list. If an atom is not a member of the list then it returns nil, else it returns the portion of the list starting with the atom in the list up to the last element of the list. For example, if we define*

```
(setq last-alphabet '( u v w x y z ) )
; then
( member      'a      last-alphabet ) returns nil
( member      'w      last-alphabet ) returns ( w x y z )
```

However, the predicate *member* tests the atom only for *top membership* of the argument list. Hence,

(member 'w ' (u v (w x y) z) returns nil,
 because w is not a member of the list given by the second argument, but w is a member of a member, viz of (w x y) of the list given by the second argument.

The predicate eql: We considered two forms of predicates for equality viz. '*equal*' and '='. We consider another predicate *eql* for equality. The predicate *eql* checks the equality of the internal structure of its arguments. If the *structures* of arguments are identical, it returns t else nil. In order to explain the behaviour of eql, we need the following additional information:

Each time we use the *function list* even with the same elements, it takes new memory cells and creates the list. Thus the two s-exprs,

```
(setq list1      ( list 'x 'y 'z ) )
and
(setq list2      ( list 'x 'y 'z ) )
```

creates two lists viz list1 and list2 which *are internally different* though each of them consists of the same three elements x, y and z. However, further we have

```
( setq  list3  list1)
```

then list3 and list1 point to the same memory locations and hence

```
( eql list1 list2 ) returns nil
( eql list1 list3 ) returns t
( eql list2 list3 ) returns nil
```

1.8 LOGICAL OPERATORS: AND, OR and NOT

The main differences in the behaviour of the logical operators in LISP from their behaviour in Boolean Algebra or in some other/programming languages are:

- i) The operators AND and OR may take one, two or more than two arguments
- ii) The values operated by logical operators in LISP are not exactly *true* or *false*

but the values are *nil* and *non-nil*, i.e., in LISP any S-expr which is not *nil* has the same logical status as that of *true* in Boolean Algebra. Hence, modified definitions of the three logical operators are:

NOT: Not of *nil* is *t*, and, Not of *non-nil* is *nil* or *()*

For example, (not ' (a b)) is *nil*
(not ()) is *t*

AND and OR are treated as special forms as described below:

AND: The arguments of AND are evaluated from left to right until some S-expr evaluates to *nil* then other arguments are not evaluated. If, at any stage, an argument evaluates to *nil*, then *nil* is returned. However, if none of the arguments evaluates to *nil* then the value of the last argument is returned.

OR: The arguments of OR are evaluated from left to right, until either some value returned is *non-nil*, then the value is returned as the value of application of OR and the rest of the arguments are not evaluated. However, every argument evaluates to *nil*, then *nil* is returned.

Examples:

(not nil)	returns t
(not t)	returns nil
(not 'dog)	returns nil
(and t 'dog)	returns dog
(and t nil 'dog)	returns nil
(or t nil 'dog)	returns t ; first argument that is non-nil, if any
(or 'dog nil t)	returns dog;
(or 'dog)	returns dog
(and 'dog)	returns dog
(or nil ())	returns nil.

1.9 EVALUATION OF SPECIAL FORMS INVOLVING DEFUN and COND

So far we have discussed only built-in functions including predicates and relations. The special word DEFUN allows us to write our own functions and build our own programs. **To build up highly complex programs, we need**

- (i) **iteration**, i.e., repeated execution of a sequence of statements, and

(ii) **Selection.**

In LISP, capability for iteration is provided by the *Do construct*. On the other hand, the selection capability in LISP is provided by COND.

(i) **Functions are defined using the syntax:**

```
(defun <function-name> <parameter-list>
  <function-body> )
```

where <function-name> is a symbol which names the function being defined, <parameter-list> is a list of distinct symbols, which forms a list of parameters to the function and <function-body> is the sequence of S-exprs which describes (or denotes) the desired computation.

Simple examples: The LISP function, corresponding to the mathematical function

$f(x, y) = x^3 + y^3$ for all x, y , is given by the function definition

```
(defun sumcube (x y)
  (+ (* x x x) (* y y y)))
```

Note 1: We have already mentioned that in interpreter mode, every S-expr, which appears after the LISP prompt '→' is read, evaluated and printed. The execution of an S-expr that defines a function, **returns the name of the function**. The name, acting as a symbol, has a value obtained through execution, associated with it and the associated value can be used in further processing. For example,

```
→ ( defun sumcube ( x y ) ( + ( * x x x ) ( * y y y ) ) ); returns
   ; sumcube
Next S-expr
```

```
→ ( length ( list ( defun sumcube ( x y ) ( + (* x x x) (* y y y) ) )
; returns
;the integer 1 because ( defun ....) returns symbol sumcube then (list ...) returns the
;list (sumcube) and finally (length ..... ) returns the integer 1.
```

Note 2: Applying a function to its arguments is termed as making a **function call**. For the above definition of the function sumcube, the following sequence

```
→ (setq x 3 ) ; returns 3
→ ( sumcube 2 x ) ; returns 2 * 2 * 2 + 3 * 3 * 3, i.e. 35
```

The capability of conditional or selective evaluation is provided by the special symbol COND.

The syntax (or legal form) for COND is:

```
(cond
  ( < test-1 > < S-expr > < S-expr > ... < S-expr > )
  ( < test-2 > < S-expr > < S-expr > ... < S-expr > )
  ( < test-n > < S-expr > < S-expr > ... < S-expr > )
)
```

Each list of the form (< test-i > < S-expr > ... < S-expr >) in the above is called a **clause**.

The COND form is evaluated according to the following rule :

Evaluate the first test viz. $\langle \text{test-1} \rangle$ in clause 1. If it evaluates to non-nil then the remaining $\langle \text{S-expr} \rangle$'s in the clause are evaluated in the order from left to right and the value of the whole COND is the same as the **value of the last S-expr in clause1**. If $\langle \text{test-1} \rangle$ evaluates to nil, the same sequence of steps is repeated for second clause, and so on. Until either some $\langle \text{test-i} \rangle$ evaluates to non-nil, for which earlier described sequence of steps for the non-nil case, is followed. However, if all $\langle \text{test-i} \rangle$ evaluate to nil then COND form evaluates to nil.

A special case of COND form is the one in which $\langle \text{test-i} \rangle$ is the first test which invariably evaluates to non-nil and the corresponding ith clause has no other S-expr, i.e., ith clause is of the form $(\langle \text{test-i} \rangle)$.

In this case, the value of $\langle \text{test-i} \rangle$ which is assumed to be non-nil is returned.

Examples of COND special form

```
( defun    our-max-3      ( x y z )
  (  cond
    ( ( > x y )
      ( cond
        ( ( > x z ) x )
        ( t    z )
      )
    )
    ( ( > y z ) y )
    ( t    z )
  )
)
```

Comment (a): The example given above graphically demonstrates a style of placing corresponding parentheses in the code. *This is allowed as LISP code is format free.*

Comment (b): We also know that t evaluates to itself and hence is non-nil. Therefore, the two occurrences of the clause $(t \ z)$ state that in case $(> x \ y)$ is true but $(> x \ z)$ is false then z is returned. Similarly, if $(> x \ y)$ is false then the next clause as given below is executed

```
(
  ( ( > y z ) y )
  ( t    z )
)
```

In this clause, first of all, condition $(> y \ z)$ is evaluated which, if the value happens to be nil then the condition t in the next sub-clause $(t \ z)$ is tested. As t always evaluates to non-nil, hence z is returned.

Comment (c): As z is a number and a number always evaluates to non-nil, therefore, each of the two occurrences of $(t \ z)$ in the definition of our-max-3 can be replaced by just (z) .

1.10 THE SPECIAL FORMS *DO* AND *LET*

The special form Do provides the power of iteration to LISP. We may recall from our earlier studies that iterative constructs are very useful, specially for denoting long sequences of actions by shorter code.

Also, LET is special form useful in LISP because, it facilitates the creation of local variables and often yields code which is both compact and efficient. *Let us first discuss the special form Do in detail.*

The do construct has the following form:

```
(
  ( do
    ( ( var1  init1  step1 )
      ( var2  init2  step2 )
      ( vari   initi  stepi )
      ( varm  initm  stepi )
    ) ; this part initiates various variables
      ; also specifies the possible modifications to the value of var-i through step-i
  ( end-test          ; test to see if Do loop

  end-form1          ; is to be terminated

  end-form-n return-value
)
body1                ; body of Do-loop
body2

body-n ; where each body i is an S-exp.
))
```

The above Do form is evaluated through the following sequence of steps:

Step 1: Variables var-i are initially bound to init-i for all i in parallel

Step 2: If the S-expr viz *end-test* is present, it is examined. If end-test evaluates to nil then the following sub-steps are followed:

- (i) Each of body-j is evaluated, and if in body-j any S-expr of the form (*return value*) is encountered, *do* is exited and its value is the *value* in return value.
- (ii) We should note that only utility of the body is for exiting or for side-effects.
- (iii) Next iteration starts (only if end-test evaluated as nil) with binding of each of the var-i to the value of step-i. If step-i is omitted, the var-i is left unchanged.
- (iv) Repeat whole of step2 again if in step2, end-test evaluates to nil else go to step3

The next two steps are executed when end-test evaluates to non-nil.

Step 3: Each of end-form-i is evaluated, the utility of which is for exiting or side-effects.

Step 4: Return-value is evaluated and is returned as value of DO loop.

To illustrate the applications of Do construct, we solve two problems using the construct.

Example 1: to print the first n natural numbers, where n is a parameter to the function, The required function in LISP is:

→ (defun print-beginning-integers (n)

```
( do
  (
    ( count 1 ( + 1 count ) )
  )
  ( ( equal count n ) 'done )
  ( print count )
)
```

→ (print-beginnning-integers 12)

123456789101112 done

; here *done* is used to indicate the successful

; completion of the loop. The I/O function

; print shall be explained later.

Next we explain the LET construct:

As we have already mentioned that LET is used for creating local variables. The purpose and use of LET may be explained through the simple example:

```
( defun explain-let ( x y )
  ( let
    (
      ( x 1 )
      ( y 2 )
    )
    ( print 'x = x )
    ( terpri )
    ( print ' y = y )
    ( terpi )
  )
  ( print 'x = x )
  ( print 'y = y )
)
```

The printing command (terpri) asks the printer to leave the line and start printing on the next line.

Let us call *explain-let* with x and y respectively as 8 and 9

→ (explain-let 8 9) ; we get

```
x = 1
y = 2
```

(∴ through let, we define a local loop in which x is 1 and y is 2. But, once execution exits the loop, then x and y assume the assigned values.

```
x = 8
y = 9
```

Remarks: As in Do, the values of the variables within LET structure are bound in parallel. If we wish to bind values in sequential order then we use LET *

Ex 5: Write a LISP program *expo* to compute i raise to power j where i and j are natural numbers.

- (i) The *read* statement for input is explained through the following:
→ (+ 7 (read));
→ 8 ; the value given by the user;
→ 15 ; the value returned by the system.
- (ii) The primitive TERPRI directs the printer to start on a new line.
- (iii) The primitive PRINT takes one argument. It prints its argument in the same form in which it is received and also returns the argument as value of the print statement.

→ (print '(x y z u)); returns two lines of (X Y Z U) as shown below.

```
( X Y Z U)
( X Y Z U)
```

→ (print 'good morning'); the statement an execution returns the following two lines

```
'good morning',  
'good morning',
```

```
(print 'left') (print 'right')
      left
      right
→ ( print 'left' ) ( prin1 'right' )
      left right
→
```

→ (princ “good morning”)
good morning “good morning”
→

(*format* < *destination* > < *string* > *arg1 arg2*)

Here *< destination >* specifies where the output is to be directed, e.g., to the printer or to the monitor or some other external file. Default value is generally the monitor. The word *< string >* in the format clause indicates the desired output string which is mixed up with format directives. The format directives specify how each argument is to be represented. The order of occurrence of the directives is the same as the order in which the arguments are to be printed. The character ~ is used before each directive to identify the directives. **Most common directives are:**

- ~ % : indicates that new line is to be printed.
- ~ A : is a placeholder for a value which will be printed as if print were used,
- ~ S : is a placeholder for a value which will be printed as if prin1 is used,
- ~ D : is a place holder for a value which must be an integer and which will be printed as a decimal number,
- ~ F : is a place holder for a value which must be a floating point number and will be printed as a decimal floating point number,
- ~ C : is a place holder for a value which will be printed as character output,

Next, field widths for appropriate argument values are specified by an integer between tilde (i.e., ~) and the directive, e.g., ~ 3D indicates the integer field width is 3.

1.12 RECURSION IN LISP

Recursion: LISP expresses recursive computation in a very natural way. Let us first recall below what is a recursive function:

Recursive function: is a function which calls itself repeatedly, but each call with simpler arguments than the arguments used by the preceding call.

Definition of a recursive function requires

- (i) a recursive step.
- (ii) stopping condition for stopping the processing.

*One of the most well-known example of recursive definition in Mathematics is that of factorial, which in the following definition is named as **fact** and is defined as follows:*

fact (1) = 1	(the stopping/termination condition)
fact (n) = n*fact (n-1)	(the recursive step)

In LISP, the above function is expressed as

```
→ (defun fact (n)
  (cond
    (
      (( = n 1) 1)
      (t (* n fact (- n 1)
    )
  )
)
```

FACT

→ (fact 5)

Another simple example is given below to explain recursion in LISP:

We define our own function LEN that returns the number of top-most elements in a given list say L:

```
( defun LEN ( L )
  ( Cond
    (
      ( ( null L ) 0 )
      ( t ( + 1 ( LEN ( Cdr L ) ) ) )
    )
  )
)
```

Ex 6: Write a function *deep-length* that counts the number of atoms (not necessarily distinct) in a given list. The atoms may be in a list which is a member of another list which at some level occurs as an element of the given list. For example, for the list

L = (1 (2 (3 4)) (5))

length L is three. However, deep-length of L is five.

1.13 ASSOCIATION LIST AND PROPERTY LIST

Association lists are useful tools to associate attributes and their values with objects. For example, to describe a particular book viz. *LISP by Winston & Horn published by Addison-Wesley Publishing Company in 1984*, we may use the representation:

```
( setq book ' ( ( title ( LISP second edition ) )
  ( author ( Winston & Horn ) )
  ( year 1984 )
  ( publisher addison-wesley )
)
```

The value of the attribute *title* is a list viz. (*LISP second edition*) whereas the value of the attribute *year* is *1984*. The values of the attributes may be any S-expr, e.g., number, symbol or list. *Formally we define.*

Association List is a list of embedded sublists, in which first element of each sublist is a key. In the example of book given above, the symbols *title*, *author*, *year* and *publisher* are keys.

The procedure ASSOC: to retrieve values of keys or attributes, the procedure ASSOC takes two arguments viz *the key* and *the object-name* and returns the list of two element, the first element is the key and the second element is the associated value of the key, e.g.

```
→ (ASSOC 'year book ); returns
( year 1984 )
```

For a given object, ASSOC looks down the sublist (*each sublist representing key s associated value of the key*) starting from the first sublist in the list, and matches the car of each sublist with the key given as first argument of ASSOC. If the two do not

match, ASSOC goes further down to next sublist. However, if the key and the car of the sublist match then whole of the sublist is returned.

Property List, and the primitives GET, PUTPROP and SETF

Another way of associating properties and their associated values to objects in LISP, is through property lists. Considering again the earlier example of book with title as *LISP*, author as *Winston & Horn*, Publisher as *Addison-Wesley*, year as *1984*, we can put this information in the database using the above-mentioned primitives as follows:

→ (putprop 'book 'LISP 'Title) ; putprop returns the attribute values LISP

→ (putprop 'book ' (Winston & Horn) 'Author)

' (Winston & Horn)

→ (putprop 'book 'AddisonWesley 'Publisher)

Addison-wesley

→ (putprop 'book 1984 'year)
1984

The general form of putprop statement in LISP is

(putprop < object – name – symbol > < attribute – value > < attribute – name >)

where < object – name – symbol > and < attribute – value > must be symbols and < attribute –value > may be any S – expr.

The newer version of COMMON LISP avoid PUTPROP and instead use SETF.

The primitive SETF : SETF is like SETQ. However, SETF is more general than SETQ. The primitive SETF also takes two arguments, but the first argument is allowed to be an **access function** in addition to being an atom. An **access function** includes car, cdr and get. Second argument to SETF is the value, as is in the case of SETQ. The above-mentioned LISP statements using putprop can equivalently be replaced by the following statements using SETF and GET :

→ (SETF (GET 'book 'Title) 'LISP)
LISP

→ (SETF (GET, book ' Author) ' (Winston & Horn))
(Winston & Horn)

→ (SETF (GET 'book 'Publisher) 'Addison-Wesley)
Addison-Wesley

→ (SETF (GET 'book 'year) 1984)
1984

The general format for associating values to attributes of an object using SETF and GET is

(SETF (GET < object – name –symbol > < attribute – name >)
< attribute – value >)

SETF can also be used to replace values of car or cdr of a list as follows:

→ (SETQ L ' (x y z))

```

      ( x y z )
→ ( SETF (Car L) 'a )
      ( a y z )
→ ( SETF (Cdr L) ' ( u v w ) )
      ( a u v w )

```

In general (SETF (car < list >) < expr >) replaces the car of < list > by < expr >

and (SETF (cdr < list > < expr >))

replaces the cdr of < list > by < expr >

This usage of SETF allows us to change the values of attributes, whenever required.

In order to retrieve values of attributes or properties, the primitive GET is used.

Assuming, we have already put the information about the book: LISP by Winston & Horn in the database. Then the relevant information pieces may be retrieved as follows:

```

→ ( GET 'book 'year )
    1984
→ ( GET 'book 'Title )
    LISP
→ ( GET 'book 'Author )
    ( Winston & Horn )

```

etc.

The general format of GET, in order to find the value of the attribute having name as < attribute – name > of the object having name as < object – name >, is :

```
( GET < object – name > < attribute > )
```

If there is no value in the data-base for < attribute –name > and < object – name >, the value nil is returned.

Note : When more than one SETF or PUTPROP are used to give *different values to the same attribute* of a given object then the effect of *only the latest* remains. Earlier values are overwritten. In order to change values of attributes, we write another statement using SETF or PUTPROP. Thus, in continuation of our example about the book entitled *LISP by Winston & Horn*, if in addition to the earlier statement, we give the following statement:

```
→ ( SETF ( GET 'book 'year ) 1987 ) ; returns
```

the year of publication as 1987, replacing the year 1984.

Optional parameters: So far we have restricted to the definition of those functions which have *fixed number of arguments* that are always evaluated. However, there are situations, in which it may be desirable to define functions with *variable number* of parameters. For example, we want to define a function *exponentiate* such that either the value of *base b* is given or else it is assumed as 10. For this purpose, we use the inbuilt function *expt* which returns m^n for (expt m, n). Thus, *exponentiate* may require one argument or two arguments. For such situations LISP provides a key word &OPTIONAL, which when used in parameter-list indicates that the parameters listed

after *&OPTIONAL* may or may not have arguments corresponding to them. If the arguments corresponding to some parameters after *&optional* are present, the function uses these in determining the output result, else there would be no error because of their absence. For example, the function *exponentiate* as defined above can be described in LISP as follows:

```
→ ( defun exponentiate ( n & optional m )
      ( Cond
        ( ( null m ) ( expt 10 n ) )
        ( t ( expt m n ) )
      )
    )
    exponentiate
```

;exponentiate, in the previous line, is the value returned by the definition

It may be noted that if *&optional* keyword had not been available and/or had we defined exponentiate by replacing the parameter-list (n & optional m) by (n m), then there would have been an error if m is not supplied assuming it to be 10.

The key word &rest

In order to explain the use of *&rest*, we consider the following example:

```
→ ( defun our-sum-3 ( n1 n2 & optional n3)
      (Cond
        ( ( null n3 ) ( + n1 n2 ) )
        ( t      ( + n1 n2 n3 ) )
      )
    )
```

Our-sum-3 ; returned by the definition.

Let us make the following calls:

```
→ ( our-sum-3 5 6 )
→ ( our-sum-3 ) 5 6 7 )
    18
→ ( our-sum-3 5 6 8 9 )
    ERROR
```

The above ERROR occurred, because for correct response by our-sum-3, minimum number of arguments in this case must be 2 (i.e., number of parameters before *&optional* and *maximum number* in this case, must be 3 (i.e. number of all parameter before or after *&optional*), but in the last call to our-sum-3, we supplied four arguments viz. 5, 6, 7, and 8.

Now, it is not always possible to remember the exact number of optional parameters and hence not always possible to check erroneous function calls. To remedy this situation, LISP provides for the keyword *&REST* which is followed by exactly one argument say 'remaining'. Then if m denotes number of parameters before *&optional* and n the number of parameters after *&optional* but before *&rest* and whenever k arguments are supplied and $k > m + n$, then all the *remaining* ($k - (m + n)$) arguments are grouped into a list and bound to *&rest*. In order to explain the ideas explained above, let us define a function say specialsum-3 as follows.

```
→ (defun specialsum-3 ( n1 n2 &optional n3 &rest n4 )
      ( Cond ( ( and ( null n3 ) ( null n4 ) ) ( + n1 n2 ) )
        ( ( null n4 ) ( + n1 n2 n3 ) )
      )
    )
```

```

      ( t      ( ( + n1 n2 n3 ) ( print n4 ) ) )
    )
  )
; the definition returns.
Special-sum-3

```

```

→ (special-sum-3      5 7)
    12
→ ( special-sum-3 5 7 9 )
    21
→ ( special-sum-3 5 7 9 11)
    21 ( 11 )

→ ( special-sum-3 5 7 9 11 12 13 )
    21 ( 11 12 13 )

```

1.14 LAMBDA EXPRESSION, APPLY, FUNCALL AND MAPCAR

When a function is to be called *only once* in a program then we may not like to give a name to the function in the definition of the function. In such a situation, instead of the keyword DEFUN we use the keyword LAMBDA. Rest of the definition of the function remains the same as it would have been under DEFUN. Suppose we need to compute $(x^2 - y^2)^2$, the following LAMBDA expression will accomplish the task:

```

→ (LAMBDA ( X Y )
    ( * ( - ( * X X ) ( * Y Y ) )
      ( - ( * X X ) ( * Y Y ) )
    )
  )

```

Application of a LAMBDA expression is similar to that of application of a function under normal definition through DEFUN, e.g.,

```

→ ( ( LAMBDA      ( X Y )
    ( * ( - ( * X X ) ( * Y Y ) )
      ( * X X ) ( * Y Y )
    )
  )
  ) (3 4); returns
49

```

The functions APPLY and FUNCALL

APPLY takes two arguments, each of which is evaluated. The first argument, which is either a function-name or LAMBDA expression, is applied to second argument which is a list. **FUNCALL** is similar to the function APPLY *with the difference* that arguments are supplied without boundary parentheses of a list. Function-names are preferably quoted with #' in stead of just quote.

Examples:

```

( APPLY #'* ( 2 3 ) )
6
→ (FUNCALL #'* 2 3 )
6

```

For the earlier defined function *our-sum-3* which returns sum of 2 or 3 arguments, what ever number of arguments out of 2 or 3, are supplied. Let us consider

```
→ ( APPLY #' our-sum-3 ( 4 5 ) )
      9
→ ( FUNCALL #' our-sum-3 4 5 6 )
      15
→ ( APPLY #' ( LAMBDA ( X Y ) ( * ( + x x ) ( + y y ) ) ) ( 3 4 ) )
      48
→ ( funcall #' ( LAMBDA ( X Y ) ( * ( + X X ) ( + Y Y ) ) ) 3 4 )
      48
```

The Backquote facility : The backquote is just like quoted expression and evaluates to itself except the following difference : Those subexpressions of the expression that are preceded by a comma or by the comma followed by the symbol @ are evaluated and substituted appropriately before returning the result. Let A be bound to '(3 x 4) then

```
→ ' ( A B C ) ; evaluates to
      ( A B C )

→ ' ( ,A B C ) ; evalutes to
      ( ( 3 x 4 ) B C )

→ ' ( A ,A B ,@ A C ) ; evaluates to
      ( A ( 3 x 4 ) B 3 x 4 c )
```

Explanation of evaluation of backquoted expressions: Any subexpression which is preceded by a comma is evaluated and substituted by the value so obtained. Further, if a subexpression is preceded by , @ then it is evaluated and splice substituted. i.e., substitution after removing bounding parentheses, where a the result of evaluation of the argument must be a list.

The function MAPCAR: MAPCAR is a useful function which is to be repeatedly applied to a set of lists where each list constitutes one argument list for the function. Let f be a function of arity k, i.e., the function f requires k arguments in one application.

The application of mapcar is explained through the following examples.

```
→ ( mapcar #' + ' ( 1 2 3 4 ) ' ( 3 4 ) )
      10 7
(i.e., elements of each of the two lists are added separately)
→ (mapcar #' (lambda (x) ( + x 7 ) ) ' ( 2 3 4 ) )
; returns
9 10 11
→ (mapcar #' list' (x y z ) (a b c ) )
;returns
(x a) (y b) ( z c )
```

1.15 SYMBOL, OBJECT, VARIABLE AND REPRESENTATION

In the contexts in which a symbol has or is expected to have some object or S – expr associated with it, it is called a **variable**. The symbol book1 becomes a variable, when associated with the object which represents a book entitled *LISP*, authored by

Winston & Horn in the year 1984. The association between the symbol and the object may be achieved through the LISP statement:

```
→ ( setq book1 ' ( ( title LISP ) ( author ( Winoston & Horn ) )
    ( year 1984 ) ( printing first 1984 )
  ) ; returns
book1
```

The associated object may be referred to as the value of the symbol. The variable may be considered as the ordered pair: (symbol, value).

Also, a symbol used in the parameter list of a function definition, though does not have any associated value or object at the time of definition, yet is a variable because it is expected to be associated with some object at the time of application of the function.

Bound & Free Variables: A symbol that appears in the parameter list of a procedure, is called a *bound* variable w.r.t the procedure. A symbol, that does not appear in the parameter list of a procedure, is called a *free* variable w.r.t the procedure.

Representation of Symbols: In LISP environment, the link between a symbol and the associated object is unique and is achieved through the following mechanism.

LISP system maintains a Symbol Table (in some part of the memory) in which each symbol, when encountered for the first time, is entered along with some starting address, say 3000, of some location in the memory where the associated object is stored. We may note that some of the components of the object may be changed over time, e.g., if the copies of book1 are again printed in the in the year 1988. In such a case, the component ' (printing first 1984)' of the object is changed by ' (printing second 1988)'. However, the entry (book1 3000) remains unchanged. Next time when book1 occurs in a program, the LISP system searches through its possible occurrences in the symbol table and on finding it there, does not attempt to associate with it another address or location in memory. Further, the statements like,

```
( setq book2 ' book1 ) and
( setq book3 ' book1 )
```

will associate address 3000 (i.e. the address associated with book1) with symbols book2 and book3 as their address parts of (symbol, address) pairs. Thus, we may also say that a variable is an ordered pair (symbol, pointer).

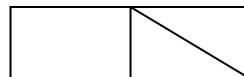
The Predicate EQ : EQ returns t if and only if *the internal structures* of its arguments are identical. Hence, continuing with the earlier discussion, the value t is returned in all the following three cases:

```
( EQ ' book1 ' book1 )
( EQ ' book1 ' book2 )
( EQ ' book2 ' book3 )
```

Internal Representation of Lists, cons-cells in memory and the data structure

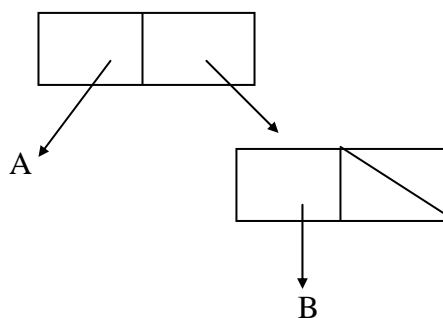
Dotted Pair: We pointed out that **in order to store symbols**, LISP system marks a section of memory, called symbol table and which is considered as composed of cells to store (symbol, pointer) pairs. The first component of each cell is interpreted as a symbol and the second component as an address. Similarly, **in order to store lists**, LISP system marks out a segment of memory constituting of what is known as **cons-cell**. Each cons-cell is a pair (*pointer, pointer*), i.e., each cons-cell contents are interpreted as pair of pointers or addresses. The first pointer, in the cons cell to a list

say L, points to the location in memory where information about CAR of L can be found and second component of a cons cell points to a location where information about CDR of L can be found. The cons-cell may be diagrammatically represented as in the diagrams below, where the left arrow points to CAR and right arrow points to CDR of the list to be represented by the cons-cell. The CDR of a single element list is nil and is represented by the cons-box, where we may call the boxes of the form as cons boxes.

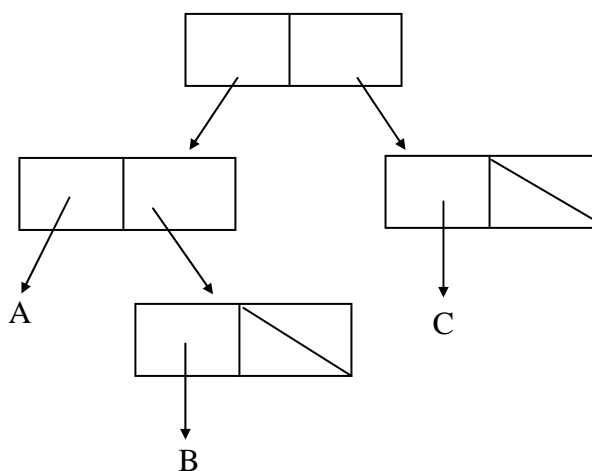


Example of representation of lists by cons cells

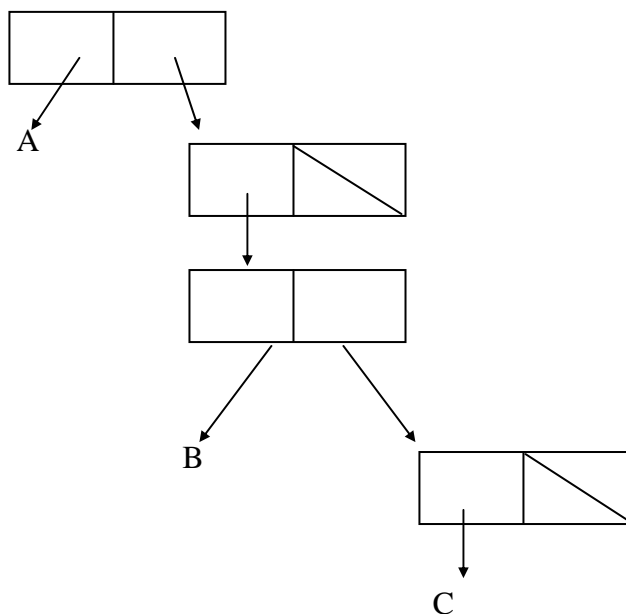
1. The list (A B) is represented by the cons cell structure:



2. The list ((A B) C) is represented by the cons cell structure:

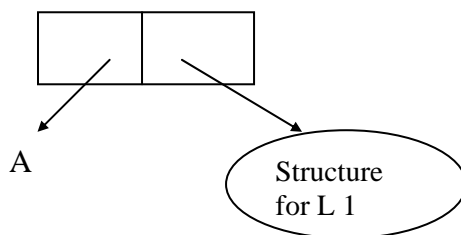


3. The list (A (B C)) is represented by the cons cell structure:

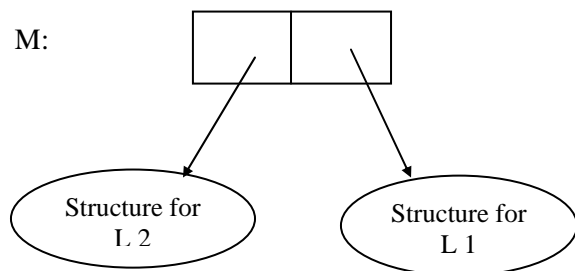


Remarks: The name cons in the cons-cell structures, is justified on the following grounds:

Let L1 be a list and A be an atom and we have LISP statement (*setq* L (*Cons* ' A L1)) then L is represented by adding one cons cell in the memory as shown below:



Also if L2 is another list then, on the command, (*setq* M (*Cons* L2 L1)) resultant list M is obtained by adding one cons cell in memory as shown below:



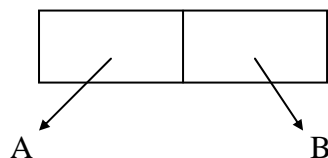
It can be easily seen that using the Cons cells representation for lists, operation like CAR, CDR, ATOM etc can be efficiently implemented.

Ex 7: Draw cons-cell structure for the list ((A B) (C D)).

The Data Structure: Dotted Pair

The cons-cell structure suggests that a cons-cell in memory may represent a LISP object in which the CDR need not be a list but may be an atom or a symbol.

Dotted Pair: A LISP data structure pair is a structure like list with the difference that CDR of a dotted pair may be an atom also. **A dotted pair** with CAR as symbol A and **CDR as symbol B** is denoted by (A . B) with spaces around dot on both sides. Thus, cons-box representation for dotted pair (A . B) is



and (CAR ' (A . B)) is the symbol A
and (CDR ' (A . B)) is the symbol B.

1.16 DESTRUCTIVE UPDATES, RPLACA, RPLACD & SETF

We have earlier discussed updates of attribute values in property lists through **PUTPROP** and **SETF**. But, so far, we have not discussed primitives, in context of general lists, which achieve *destructive updates* in parts of given lists. SETF may be used for the purpose. However, first we introduce two mnemonics for **replace CAR**

and for **replace CDR**. The mnemonic *RPLACA* is used for *replace CAR* and the mnemonic *RPLACD* is used for *replace CDR*.

Now we explain the two primitives. Both RPLACA and RPLACD take two arguments. For RPLACA, first argument is a non-empty *list* say bound to a symbol, say, X and second is an arbitrary LISP *object* say bound to a symbol, say, Y. Then (RPLACA X Y) replaces (Car X) by Y in the given list and the resulting list is still bound to X. For RPLACD, the first argument is again a non-empty list bound to the symbol, say, X and the second argument *also must be a list*, say, bound to Y then (RPLACD X Y) replaces (CDR X) by Y and the resulting list is still bound to X.
Examples:

→ (SETQ X) '((a b) c d); returns
((a b) c d)

→ (SETQ Y 3); returns
3

→ (REPLACA X Y); returns
(3 c d); Further if we give

→ (SETQ Z ' (3 7 9)); returns
(3 7 9)

→ (SETQ U ' (c e f) g)); returns
((c e f))

→ (REPLACA Z U); returns;
(((c e f) g) 7 9); this list is bound to Z

→ (SETQ V' ((a b) c)); returns
((a b) c)

→ (REPLACD Z V); returns
(((e e f) g) ((a b) c))

The above two primitives viz RPLACA and RPLACD can be obtained from SETF as follows:

(RPLACA L S) is same as (SETF (CAR L) S)
and (RPLACD L S) is same as (SETF (CDR L) S)

In general we can make changes to lists in arbitrary positions instead of just to the (CAR L) and (CDR L), as follows:

(RPLACA) (Cxxxxr L) S) or (SETF (Caxxxxr L) S)
and (RPLACD (Cxxxxr L) S) or (SETF (Cdxxxxr L) S)
where xxxx is a sequence made out of A's and D's.

Example :

→ (SETQ X ' ((a b) (c (d e) f))); returns
((a b) (c (d e) f))

→ (RPLACA (Caddr X) 'p)
((a b) (c p f)); still bound to X

→ (SETQ Z ' ((a b) (c d e)))
((a b) (c d e))

→ (REPLACD (Cdadr Z) ' (F G)); returns
((a b) (c d F G))

Ex 8: Find

- (i) (RPLACA (Cdadr '((u v w) (s (t u) m))) '(a b c))
(ii) (RPLACD (Cdadr '((u v w (s (t u) m))) '(a b c))

1.17 ARRAYS, STRINGS AND STRUCTURES

I. ARRAYS: LISP provides the primitive MAKE-ARRAY to create array structures.

To create an array structure of *dimensionality* n and *dimensions* $\langle \text{dim-1} \rangle, \dots, \langle \text{dim-n} \rangle$ the following syntax using MAKE-ARRAY is used:

(MAKE-ARRAY '($\langle \text{dim-1} \rangle$ $\langle \text{dim-2} \rangle \dots \langle \text{dim-n} \rangle$))

For example to create an array structure named *Matrix-3* with dimensionality 3 and $\langle \text{dim-1} \rangle$ as 2, $\langle \text{dim-2} \rangle$ as 2 and $\langle \text{dim-3} \rangle$ as 3 of integers, the following LISP statement is used:

→ (SETQ Matrix-3 (MAKE-ARRAY '(2 2 3) & KEY : integer))

; returns the name Matrix-3

The above statement creates an array Matrix-3 of 12 elements. The slots in Matrix-3 are empty and we will discuss how to fill values in the slots. The 12 slots in Matrix-3 are referred to as

Matrix-3 (0, 0, 0), Matrix-3 (0, 0, 1) Matrix-3 (0, 0, 2)
Matrix-3 (0, 1, 0), Matrix-3 (0, 1, 1), Matrix-3 (0, 1, 2)
Matrix-3 (1, 0, 0), Matrix-3 (1, 0, 1), Matrix-3 (1, 0, 2)
Matrix-3 (1, 1, 0), Matrix-3 (1, 1, 1), Matrix-3 (1, 1, 2)

The primitive AREF is used to refer to a particular slot in the array, e.g., .

(AREF Matrix-3 1 0 2) refers to the slot Matrix-3 (1, 0, 2)

In order to assign a value to Matrix-3 (i, j, k), the following statement is used:

→ (SETF (AREF Matrix-3 1 0 2) 10); assigns value 10 to the slot (1 , 0 , 2) of Matrix-3

The above value-assigning statement can be easily used to give value say integer g to (i, j, k)th element of Matrix-3 as

→ (SETF (AREF Matrix-3 i j k) g)

Further, it can be generalised for any array in stead of Matrix-3.

In order to retrieve values from any slot, say (i, j, k) of Matrix-3 we use the following LISP statement:

→ (AREF Matrix-3 i j k)
; the value g is returned
; if g is the value stored at
; Matrix-3 (i, j, k) then g is returned

Next, we consider defining of strings in LISP:

II. A string is a one-dimensional array, whose elements are *characters*. MAKE-ARRAY or MAKE-STRING, a new primitive may be used for the purpose, e.g.,

```
→ ( SETQ A (MAKE-ARRAY '(4) &KEY :character) )  
A ; creates a string-structure A capable of storing  
   ; characters  
or equivalently we can write the statement  
→ (SETQ A (MAKE-STRING 4))
```

In order to store say 'WORD' in the string structure A

we can use either

```
( SETF ( AREF A 0 ) #\W )  
( SETF ( AREF A 1 ) #\0 )  
( SETF ( AREF A 2 ) #\R )  
( SETF ( AREF A 3 ) #\D )
```

or

```
( SETF (AREF A "WORD")
```

Note that the two-character sequence viz. #\ is used preceding a character to indicate that the following is to be interpreted as character.

III. Structure: Next, we describe commands in LISP for

- defining Pascal's record-like structures in LISP environment,
- assigning values to components of such a structure and
- retrieving values from the components of such a structure.

The primitive DEFSTRUCT is used to define structures. The syntax to **create a structure** (without values assigned to components) is

```
( DEFSTRUCT < structure-name > < slot-1 > ... < slot-k > ).
```

The structure created by the above type of LISP statement will be named < structure-name > and will have <slot-i>'s as slots. We may recall that '<entity>' enclosed between angular brackets indicate place-holder for *entity* to be suitable *replaced*. The above type of LISP statement automatically generates the keyword constructor called MAKE – <structure-name> and also automatically creates the selector functions as <structure-name> – <slot-i> for each i.

We explain the somewhat terse description above through an example of defining a binary-tree structure. For this purpose, let us recall the definition of a **binary tree**: A binary tree, is either empty or it consists of a node called the root together with two binary trees called the *left subtree* and the *right subtree* of the root.

For this purpose, a node of binary tree will have three slots: left-tree, value, right-tree. The left-tree and right-tree are pointers.

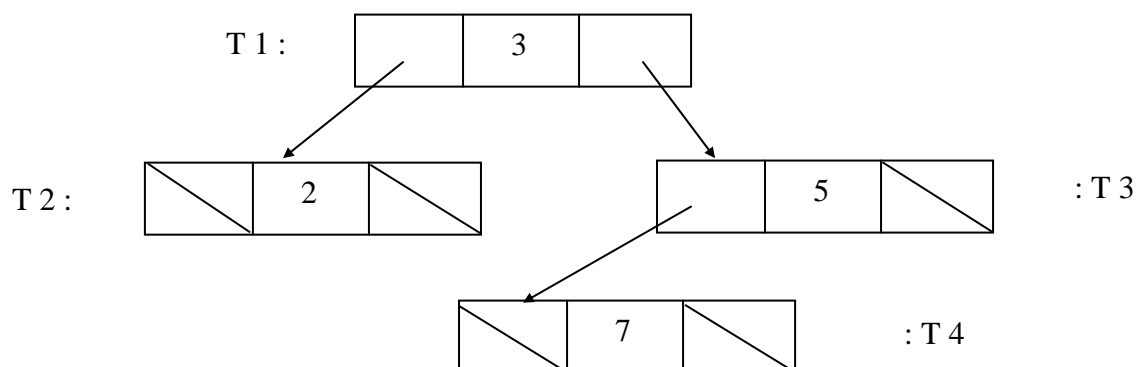
Using DEFSTRUCT, we create the structure, which we name as *bin-tree* through LISP statement:

```
(DEFSTRUCT bin-tree left-tree value right-tree)
```

The in-built mechanisms automatically create the following:

- i) Constructor function *MAKE-bin-tree*
- ii) Selector function *bin-tree-left-tree*
- iii) Selector function *bin-tree-value* and
- iv) Selector function *bin-tree-right-tree*.

Using the above description let us create the following binary tree to be called T1. The children and grand-children nodes are named as T2, T3 and T4. And a diagonally crossed cell indicates nil pointer.



The following sequence of LISP statements create the tree shown above:

→ (SETQ T4 (MAKE-bin-tree: left-tree nil : value 7 :right-tree nil))
 ;# S (nil 7 nil) is the value returned
 ; Note we can state : left-tree, : value, : right-tree in any order, e.g.

→ (SETQ T3 (MAKE-bin tree : value 5 : left-tree T4 : right-tree nil))
 ;#S (#S (nil 7 nil) 5 nil) is the value returned.

→ (SETQ T2 (MAKE-bin-tree : right-tree nil : left-tree nil: value 2))
 ;#S (nil 2 nil) is the value returned

→ (SETQ T1 (MAKE-bin-tree: left-tree T2 : right-tree T3 : value 3))
 ; the value returned is the following
 # S (# S (nil 2 nil) 3 # S (# S (nil 7 nil) 5 nil))

In the above the symbol #S indicates the fact that the part following # is a structure. In order to access values of: left-tree, : right-tree or : value the selectors *bin-tree-left-tree*, *bin-tree-right-tree* and *bin-tree-value* respectively are used, for example

→ (bin-tree-value T1) ; returns
 3

→ (bin-tree-left-tree T1) ; returns
 #S (nil 2 nil)

→ (bin-tree-left-tree T1); returns
 #S (nil 2 nil)

→ (bin-tree-right-tree T1); returns
 #S(#S (nil 7 nil) 5 nil)

Also if we may give the name of a structure then the whole of the structure would be available, e.g.

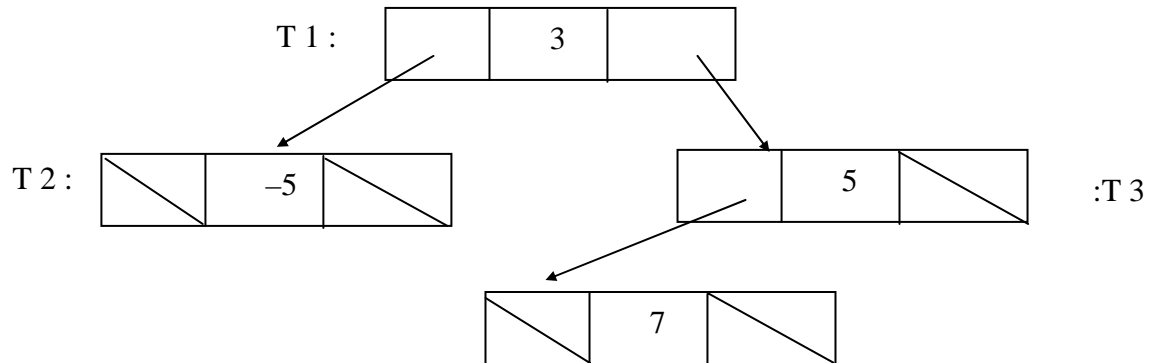
→ T3; the following value is returned
 #S (#S (nil 7 nil) 5 nil)

→ T1; the following value is returned
 # S (#S (nil 2 nil) 3 # S(# S(nil 7 nil) 5 nil))

Further, in order to change or even create value of any component, we use the primitive SETF. For example, if we wish to change : *value* component of T2 to – 5 we can use

→ (SETF (bin-tree-value T2) – 5); returns

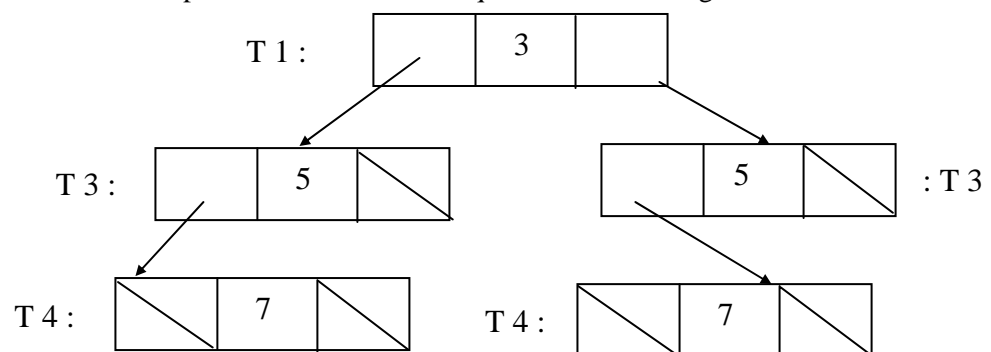
;the following tree as



If further, we want T1 to be more symmetrical having T3 as both : *left-tree* as well as : *right-tree*, then we can use

→ (SETF (bin-tree-left-tree T1) T3); returns T3 as value
#S (#S (nil 7 nil) 5 nil)

The new shape of tree T1 after the sequence of two changes mentioned above is like:



Further if give the command

→ T1; the new structure returned is

#S (#S (#S (nil 7 nil) 5 nil) 3 #S (#S (nil 7 nil) 5 nil))

1.18 SUMMARY

The programming language LISP is based on the *functional paradigm* of solving problems using a computer system. The concepts of '*paradigm of solving problem*', '*functional paradigm*' and '*imperative paradigm*' are briefly discussed in Section 1.0, i.e., in the *Introduction* section. Elements of the syntax of the language LISP are introduced in Section 1.2. The issues relating to structuring of data and representing data in LISP are discussed in Section 1.3. How a LISP system evaluates a valid LISP object is explained in the next section. Section 1.5 explains how primitive LISP functions are evaluated. Primitive List manipulation functions are discussed in Section 1.6. The next section discusses built-in predicates of LISP. The various logical operators are discussed in Section 1.8. Some facilities to write complex programs in LISP are provided in the form of special forms which use the special words DEFUN,

COND, DO and LET etc. These special forms are explained in Sections 1.9 and 1.10. The input/output functions and facilities are discussed in Section 1.11. The concept/mechanism of *recursion* plays a very important role in *programming* in general, and *symbolic programming* in particular. Recursion in LISP is discussed in Section 1.12.

In order to define objects in terms of their attributes and attribute values, *association lists* and *property lists* are used in LISP. These concepts are discussed in Section 1.13. Some more general and robust facilities in LISP defining and applying functions in the form of **Lambda Expression, Apply, Funcall and Mapcar** are discussed in Section 1.14. The representation of *symbols* and associated/represented objects in LISP environment and representation of operations on such representations are discussed in the next three sections.

1.19 SOLUTIONS/ANSWERS

Ex 1: (i) The variable x is bound to 5 and the variable y is bound to 7. Further the value $(5 + 5) * (7 + 7)$ is evaluated to 140

(ii) The expression having a quote in the beginning itself is evaluated to

$(- (+ (\text{setq } p \ 9) (\text{setq } s \ 3)) (p * s)).$

No binding of p to 9 and s to 3 takes place.

Ex 2: (i) The expression in the first round reduces to $(+ \ 24 \ (- \ 1) \ 2)$ which reduces to 25

(ii) The expression in the first round reduces to $(* \ 5 \ 0 \ 11)$ which reduces to zero (0).

Ex 3: $(\text{Defun } (X \ Y)$
 $(\text{cond } (= Y \ 0) \ 'infinity)$
 $(t \ (/ \ X \ Y)))$

Ex 4:

Stepwise explanation

1. x is associated with (a b) and (a b) is returned.
2. '(c d) evaluates to (c d)
3. through subst x is replaced by list (s t) and we get a new list
 $(u \ v \ (s \ t))$
4. through reverse, from the last list, we get $((s \ t) \ v \ u)$
5. the append of three lists viz (a b), (c d) and $((s \ t) \ (v \ u))$ returns the list
 $(a \ b \ c \ d \ (s \ t) \ v \ u)$
6. Finally 7, the length as number of topmost elements, is returned.

Ex 5:

```
(defun expo (i j)
  (do
    (answer i (* i answer))
      ; initially answer is i and is
      ; multiplied in each iteration by i
    (power j (- power 1))
    (counter (- j 1) (- counter 1))
      ; initially power is j and in each iteration power is reduced by 1.
      ; counter is an auxiliary variable
  )
)
```

```

    )
  )
  → ( expo 2 3 )
  8

```

Remarks: The clause (power j (– power 1)) is actually not required. However, it is introduced to explain. It can be deleted without affecting the overall (final) result. But it has been introduced to explain an important point about do-loop. We *may be tempted to write the above function expo* by replacing the clause (counter (– j 1) (– counter 1)) by (counter (– j 1) (– power 1)) i.e. replacing last occurrence of counter by power; because it is also being computed in the earlier clause. But this *replacement will be wrong*, leading to incorrect result because of the fact that in Do loop all the variables, viz *answer*, *power*, and *counter* in the above example are computed *in parallel*, using values from the previous iteration/ initialization. *Current* values are available only in the same clause. Therefore, if ‘power’ replaces ‘counter’ then *previous* value of power would be available for processing whereas we require the current value.

*In many situations, we need **sequential** computation of the variables in the loop. For this purpose LISP Provides do*. Now, the function of Example 2 above may be rewritten using the earlier computed values of power as is given below:*

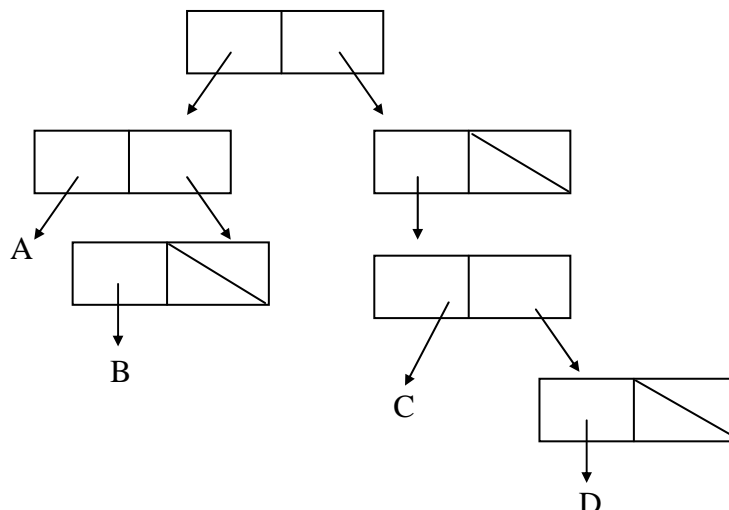
```

( defun expo ( i j )
  ( do* (
    ( answer i      ( * i answer ) )
    ( power j      ( – power 1 ) )
    ( counter      ( – power 1 ) )
    (              ( – power 1 ) )
  )
  )
  ( ( zerop counter ) answer )
)

```

Ex 6: (defun deep-length (L)
 (cond
 ((null L) 0)
 ((list p (car L)) (+ deep-length (car L)) (deep-length (cdr L))
 (t (+ 1 (cdr L)))
)
)

Ex 7: The list ((A B) (C D)) is represented by the cons cell structure



Ex 8:

- (i) ((u v w) (s (a b c) m))
- (ii) ((u v w) (s (t u) a b c))

1.20 FURTHER READINGS

1. Graham P. : *ANSI Common Lisp* Prentice Hall (1996).
2. Sangal R. : *Prgramming Paradigms in LISP* McGraw-Hill, Inc. (1990).
3. Patterson D.W.: **Chapter 3 of** *Introduction to Artificial Intelligence and Expert Systems*; Prentice-Hall of India (2001).

UNIT 2 A. I. LANGUAGES-2: PROLOG

Structure	Page Nos.
2.0 Introduction	42
2.1 Objectives	43
2.2 Foundations of Prolog	43
2.3 Notations in Prolog for Building Blocks	46
2.4 How Prolog System Solves Problems	50
2.5 Back Tracking	54
2.6 Data Types and Structures in Prolog	55
2.7 Operations on Lists in Prolog	57
2.8 The Equality Predicate '='	61
2.9 Arithmetic in Prolog	62
2.10 The Operator Cut	63
2.11 Cut and Fail	65
2.12 Summary	66
2.13 Solutions/Answers	67
2.14 Further Readings	70

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

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 latter 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 **LOGic**.

The fundamentals of PROLOG were developed in the early 1970's by Alain Colmerauer and Philippe Roussel of the Artificial Intelligence group at the University of Marseille together with Robert Kowalski of Department of Artificial Intelligence at University of Edinburgh, 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 $p_1 \vee p_2 \dots \vee p_i \vee \dots \vee p_r$, where each p_i is a literal, and ' \vee ' denotes disjunction. Further a **literal** q is of the form p or $\sim p$ (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
 $\text{is_tall}(\text{mohan}) \wedge \text{is_richer}(\text{aslam}, \text{john})$.

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, \sim (*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 87.6×10^2 or 8760. The term 35.03e–12 denotes 35.03×10^{-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 **anonymous 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

a₄:- a₁, a₂, a₃, a₅.

a₄ is the **Head** of the rule and the R.H.S of ':-' , i.e., 'a₁, a₂, a₃, a₅' 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 (represented 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.

- (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).

(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).
```

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

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

?-is sister (anita, anil).

51

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 functors, 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. Backtracking 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 *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) $[e_1, e_2, \dots, e_n]$ 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 $+X*YZ$ may be 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:

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

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.

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

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

$$(iii) \quad [X | Y] = [[a, b, c]] \text{ then}$$

$$X = [a, b, c] \text{ and } Y = []$$

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

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

$$(v) \quad [X|Y] = [a, b, [c]] \text{ then}$$

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

$$(vi) \quad \text{The operator ‘|’ is not defined for the list } [], \text{ having no elements}$$

(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]	<i>pascal does not match prolog hence apply rule (ii)</i>
2.	pascal	[fortran, pascal, cobol]	<i>by rule (ii)</i>
3.	pascal	[fortran pascal, cobol]	<i>using fact (i)</i>

5. pascal [pascal|cobol]

(again apply
fact (i))

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

- (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].`

append ([], X, X). (i)

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

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), - (subtraction), * (multiplication), / (division) and \wedge (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 $X \geq Y$ 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 $X \in L$, 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.

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.

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]	<i>fact (i) not satisfied. Hence apply rule (ii)</i>
6.	pascal	[]	<i>cannot be satisfied. The system says the goal cannot be satisfied.</i>

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).

UNIT 1 THE FIRST-ORDER PREDICATE LOGIC (FOPL)

Structure	Page Nos.
1.0 Introduction	5
1.1 Objectives	7
1.2 Syntax of Predicate Logic	7
1.3 Prenex Normal Form (PNF)	12
1.4 (Skolem) Standard Form	15
1.5 Applications of FOPL	17
1.6 Summary	18
1.7 Solutions/Answers	19
1.8 Further/Readings	24

1.0 INTRODUCTION

In the previous unit, we discussed how propositional logic helps us in solving problems. However, one of the major problems with propositional logic is that, sometimes, it is unable to capture even elementary type of reasoning or argument as represented by the following statements:

Every man is mortal.
Raman is a man.
Hence, he is mortal.

The above reasoning is intuitively correct. However, if we attempt to simulate the reasoning through Propositional Logic and further, for this purpose, we use symbols P, Q and R to denote the statements given above as:

P: Every man is mortal,
Q: Raman is a man,
R: Raman is mortal.

Once, the statements in the argument in English are symbolised to apply tools of propositional logic, we just have three symbols P, Q and R available with us and apparently no link or connection to the original statements or to each other. The connections, which would have helped in solving the problem become invisible. In Propositional Logic, there is no way, to conclude the *symbol* R from the *symbols* P and Q. However, as we mentioned earlier, even in a natural language, the conclusion of the *statement* denoted by R from the *statements* denoted by P and Q is obvious. Therefore, we search for some **symbolic** system of reasoning that helps us in discussing *argument forms* of the above-mentioned type, in addition to those forms which can be discussed within the framework of propositional logic. **First Order Predicate Logic (FOPL)** is the most well-known symbolic system for the purpose.

The symbolic system of FOPL treats an atomic statement *not as an indivisible unit*. Rather, FOPL not only treats an atomic statement divisible into subject and predicate but even further deeper structures of an atomic statement are considered in order to handle larger class of arguments. How and to what extent FOPL symbolizes and establishes *validity/invalidity* and *consistency/inconsistency* of *arguments* is the subject matter of this unit.

In addition to the baggage of concepts of propositional logic, FOPL has the following additional concepts: terms, predicates and quantifiers. These concepts will be introduced at appropriate places.

In order to have a glimpse at how FOPL extends propositional logic, let us again discuss the earlier argument.

Every man is mortal. Raman is a man.
Hence, he is mortal.

In order to derive the validity of above simple argument, instead of looking at an atomic statement as indivisible, to begin with, we divide each statement into *subject* and *predicate*. The two predicates which occur in the above argument are:
'is mortal' and *'is man'*.

Let us use the notation
IL: *is_mortal* and
IN: *is_man*.

In view of the notation, the argument on para-phrasing becomes:
For all x, if IN (x) then IL (x).
IN (Raman).
Hence, IL (RAMAN)

More generally, relations of the form *greater-than* (x, y) denoting the phrase ' x is greater than y ', *is_brother_of* (x, y) denoting ' x is brother of y ', *Between* (x, y, z) denoting the phrase that ' x lies between y and z ', and *is_tall* (x) denoting ' x is tall' are some **examples of predicates**. The variables x, y, z etc which appear in a predicate are called **parameters** of the predicate.

The parameters may be given some appropriate values such that after substitution of appropriate value from all possible values of each of the variables, the predicates become *statements*, for each of which we can say whether it is 'True' or it is 'False'.

For example, for the predicate *greater-than* (x, y), if x is given value 3 then we obtain *greater-than* ($3, y$), for which still it is not possible to tell whether it is True or False. Hence, '*greater-than* ($3, y$)' is also a predicate. Further, if the variable y is given value 5 then we get *greater* ($3, 5$) which, as we known, is False. Hence, it is possible to give its Truth-value, which is *False* in this case. Thus, from the *predicate greater-than* (x, y), we get the *statement greater-than* ($3, 5$) by assigning values 3 to the variable x and 5 to the variable y . These values 3 and 5 are called parametric values or *arguments* of the predicate *greater-than*.

(Please note 'argument of a function/predicate' is a mathematical concept, different from logical argument)

Similarly, we can represent the phrase x likes y by the *predicate LIKE* (x, y). Then *Ram likes Mohan* can be represented by the statement *LIKE* (*RAM, MOHAN*).

Also *function symbols* can be used in the first-order logic. For example, we can use *product* (x, y) to denote $x * y$ and *father* (x) to mean the '*father of x*'. The statement: *Mohan's father loves Mohan* can be symbolised as *LOVE* (*father* (*Mohan*), *Mohan*). Thus, we need not know name of father of Mohan and still we can talk about him. A function serves such a role.

We may note that *LIKE* (*Ram, Mohan*) and *LOVE* (*father* (*Mohan*), *Mohan*) are atoms or atomic statements of PL, in the sense that, one can associate a truth-value *True* or

False with each of these, and each of these does not involve a logical operator like \sim , \wedge , \vee , \rightarrow or \leftrightarrow .

Summarizing in the above discussion, *LIKE* (*Ram*, *Mohan*) and *LOVE* (*father* (*Mohan*) *Mohan*) are **atoms**; where as GREATER, LOVE and LIKE are **predicate symbols**; x and y are **variables** and 3, *Ram* and *Mohan* are **constants**; and *father* and *product* are **function symbols**.

1.1 OBJECTIVES

After studying this unit, you should be able to:

- explain why FOPL is required over and above PL;
- define, and give appropriate examples for, each of the new concepts required for FOPL including those of quantifier, variable, constant, term, free and bound occurrences of variables, closed and open wff;
- check consistency/validity, if any, of closed formulas;
- reduce a given formula of FOPL to normal forms: Prenex Normal Form (PNF) and (Skolem) Standard Form, and
- use the tools and techniques of FOPL, developed in the unit, to solve problems requiring logical reasoning.

1.2 SYNTAX OF FOPL

In the introduction of the unit, we had a bird's eye view of:

- (i) How analysis of an atomic statement of PL can and should be carried out.
- (ii) What are the new concepts and terms that are required to discuss the subject matter of FOPL.
- (iii) How (i) and (ii) above will prove useful in solving problems using FOPL over and above the set of problems solvable using only PL.

Also, in the introduction to the previous unit, we mentioned that a symbolic logic is a formal language and hence, all the rules for building constructs of the language must be specified clearly and unambiguously.

Next, we discuss how various constructs are built up from the alphabet.

For this purpose, from the discussion in the Introduction, we need at least the following concepts.

- i) **Individual symbols or constant symbols:** These are usually names of objects, such as *Ram*, *Mohan*, numbers like 3, 5 etc.
- ii) **Variable symbols:** These are usually lowercase unsubscripted or subscripted letters, like x , y , z , x_3 .
- iii) **Function symbols:** These are usually lowercase letters like f , g , h , or strings of lowercase letters such as *father* and *product*.
- iv) **Predicate symbols:** These are usually uppercase letters like P , Q , R , or strings of lowercase letters such as *greater-than*, *is_tall* etc.

A function symbol or predicate symbol takes a fixed number of arguments. If a *function symbol* f takes n arguments, f is called an *n-place function symbol*. Similarly, if a predicate symbol Q takes m arguments, P is called an *m-place predicate symbol*. For example, *father* is a one-place *function symbol*, and GREATER and LIKE are

two-place *predicate* symbols. However, *father-of* in $\text{father_of}(x, y)$ is a, *two place predicate* symbol.

The symbolic representation of an argument of a function or a predicate is called a *term* where a **term** is defined recursively as follows:

- i) A variable is a term.
- ii) A constant is a term.
- iii) If f is an n -place function symbol, and t_1, \dots, t_n are terms, then $f(t_1, \dots, t_n)$ is a term.
- iv) Any term can be generated only by the application of the rules given above.

For example: Since, y and 3 are both terms and *plus* is a two-place function symbol, *plus* ($y, 3$) is a term according to the above definition.

Furthermore, we can see that *plus* (*plus* ($y, 3$), y) and *father* (*father* (*Mohan*)) are also terms; the former denotes $(y + 3) + y$ and the later denotes *grandfather of Mohan*.

A predicate can be thought of as a function that maps a list of constant arguments to T or F. For example, GREATER is a predicate with GREATER ($5, 2$) as T, but GREATER ($1, 3$) as F.

We already know that in PL, an atom or atomic statement is an indivisible unit for representing and validating arguments. Atoms in PL are denoted generally by symbols like P, Q, and R etc. But in FOPL,

Definition: An Atom is

- (i) either an atom of Propositional Logic, or
- (ii) is obtained from an n -place predicate symbol P , and terms t_1, \dots, t_n so that $P(t_1, \dots, t_n)$ is an atom.

Once, the atoms are defined, by using the logical connectives defined in Propositional Logic, and assuming having similar meaning in FOPL, we can build complex formulas of FOPL. Two special symbol \forall and \exists are used to denote qualifications in FOPL. The symbols \forall and \exists are called, respectively, the *universal* quantifier and *existential* quantifier. For a variable x , $(\forall x)$ is read as *for all x* , and $(\exists x)$ is read as *there exists an x* . Next, we consider some examples to illustrate the concepts discussed above.

In order to symbolize the following statements:

- i) There exists a number that is rational.
- ii) Every rational number is a real number
- iii) For every number x , there exists a number y , which is greater than x .

let us denote x is a rational number by $Q(x)$, x is a real number by $R(x)$, and x is less than y by $LESS(x, y)$. Then the above statements may be symbolized respectively, as

- (i) $(\exists x) Q(x)$
- (ii) $(\forall x) (Q(x) \rightarrow R(x))$
- (iii) $(\forall x) (\exists y) LESS(x, y)$.

Each of the expressions (i), (ii), and (iii) is called a **formula** or a well-formed formula or **wff**.

Next, we discuss three new concepts, viz **Scope** of occurrence of a quantified variable, Bound occurrence of a quantifier variable or quantifier and *Free occurrence* of a variable.

Before discussion of these concepts, we should know the *difference between a variable and occurrence of a variable in a quantifier expression*.

The variable x has THREE occurrences in the formula
 $(\exists x) Q(x) \rightarrow P(x, y)$.

Also, the variable y has only one occurrence and the variable z has zero occurrence in the above formula. Next, we define the three concepts mentioned above.

Scope of an occurrence of a quantifiers is the smallest but complete formula following the quantifier sometimes delimited by pair *f* parentheses. For example, $Q(x)$ is the scope of $(\exists x)$ in the formula

$(\exists x) Q(x) \rightarrow P(x, y)$.

But the scope of $(\exists x)$ in the formula: $(\exists x) (Q(x) \rightarrow P(x, y))$ is $(Q(x) \rightarrow P(x, y))$.

Further in the formula:

$(\exists x) (P(x) \rightarrow Q(x, y)) \wedge (\exists x) (P(x) \rightarrow R(x, 3))$,

the scope of **first** occurrence of $(\exists x)$ is the formula $(P(x) \rightarrow Q(x, y))$ and the scope of **second** occurrence of $(\exists x)$ is the formula

$(P(x) \rightarrow R(x, 3))$.

As another example, the scope of the only occurrence of the quantifier $(\forall y)$ in $(\exists x) ((P(x) \rightarrow Q(x)) \leftrightarrow (\forall y) (Q(x) \rightarrow R(y)))$ is $(Q(x) \rightarrow R(y))$. But the scope of the only occurrence of the existential variable $(\exists x)$ in the same formula is the formula:

$(P(x) \rightarrow Q(x)) P \leftrightarrow (\forall y) (Q(x) \rightarrow R(y))$

An **occurrence** of a variable in a formula is **bound** if and only if the occurrence is within the scope of a quantifier employing the variable, or is the occurrence in that quantifier. An occurrence of a variable in a formula is **free** if and only if this occurrence of the variable is not bound.

Thus, in the formula $(\exists x) P(x, y) \rightarrow Q(x)$, there are three occurrences of x , out of which first two occurrences of x are *bound*, where, the last occurrence of x is *free*, because scope of $(\exists x)$ in the above formula is $P(x, y)$. The only occurrence of y in the formula is free. Thus, x is both a bound and a free variable in the above formula and y is only a free variable in the formula so far, we talked of an *occurrence of a variable* as free or bound. Now, we talk of (only) a *variable* as free or bound. A variable is **free** in a formula if at least one occurrence of it is free in the formula. A variable is **bound** in a formula if at least one occurrence of it is bound.

It may be noted that a variable can be **both free and bound** in a formula. In order to further elucidate the concepts of *scope*, *free* and *bound occurrences of a variable*, we consider a similar but different formula for the purpose:

$(\exists x) (P(x, y) \rightarrow Q(x))$.

In this formula, *scope* of the only occurrence of the quantifier $(\exists x)$ is the whole of the rest of the formula, viz. scope of $(\exists x)$ in the given formula is $(P(x, y) \rightarrow Q(x))$

Also, all three occurrence of variable x are bound. The only occurrence of y is free.

Remarks: It may be noted that a bound variable x is just a **place holder** or a **dummy variable** in the sense that all occurrences of a bound variable x may be replaced by another free variable say y , which does not occur in the formula. However, once, x is replaced by y then y becomes bound. For example, $(\forall x) (f(x))$ is the same as $(\forall y) f(y)$. It is something like

$$\int_1^2 x^2 dx = \int_1^2 y^2 dy = \frac{2^3}{3} - \frac{1^3}{3} = \frac{7}{3}$$

Replacing a bound variable x by another variable y under the restrictions mentioned above is called **Renaming of a variable x**

Having defined an atomic formula of FOPL, next, we consider the definition of a general formula formally in terms of atoms, logical connectives, and quantifiers.

Definition A well-formed formula, **wff** a just or formula in FOPL is defined recursively as follows:

- i) An atom or atomic formula is a *wff*.
- ii) If E and G are *wff*, then each of $\sim (E)$, $(E \vee G)$, $(E \wedge G)$, $(E \rightarrow G)$, $(E \leftrightarrow G)$ is a **wff**.
- iii) If E is a *wff* and x is a free variable in E , then $(\forall x)E$ is a *wff*.
- iv) A *wff* can be obtained only by applications of (i), (ii), and (iii) given above.

We may drop pairs of parentheses by agreeing that quantifiers have the least scope. For example, $(\exists x) P(x, y) \rightarrow Q(x)$ stands for $((\exists x) P(x, y)) \rightarrow Q(x)$

We may note the following two cases of translation:

- (i) for all x , $P(x)$ is $Q(x)$ is translated as $(\forall x) (P(x) \rightarrow Q(x))$
(the other possibility $(\forall x) P(x) \wedge Q(x)$ is not valid.)
- (ii) for some x , $P(x)$ is $Q(x)$ is translated as $(\exists x) P(x) \wedge Q(x)$
(the other possibility $(\exists x) P(x) \rightarrow Q(x)$ is not valid)

Example

Translate the statement: *Every man is mortal. Raman is a man. Therefore, Raman is mortal.*

As discussed earlier, let us denote “ x is a man” by $MAN(x)$, and “ x is mortal” by $MORTAL(x)$. Then “every man is mortal” can be represented by

$$(\forall x) (MAN(x) \rightarrow MORTAL(x)),$$

“Raman is a man” by

$$MORTAL(Raman).$$

The whole argument can now be represented by

$$(\forall x) (MAN(x) \rightarrow MORTAL(x)) \wedge MAN(Raman) \rightarrow MORTAL(Raman).$$

as a single statement.

In order to further explain symbolisation let us recall the axioms of natural numbers:

- (1) For every number, there is one and only one immediate successor,
- (2) There is no number for which 0 is the immediate successor.
- (3) For every number other than 0, there is one and only one immediate predecessor.

Let the *immediate successor* and *predecessor* of x , respectively be denoted by $f(x)$ and $g(x)$.

Let $E(x, y)$ denote x is equal to y . Then the axioms of natural numbers are represented respectively by the formulas:

- (i) $(\forall x) (\exists y) (E(y, f(x)) \wedge (\forall z) (E(z, f(x)) \rightarrow E(y, z)))$
- (ii) $\sim ((\exists x) E(0, f(x)))$ and
- (iii) $(\forall x) (\sim E(x, 0) \rightarrow ((\exists y) E(x, y) \wedge (\forall z) (E(z, g(x)) \rightarrow E(y, z))))$.

From the semantics (for meaning or interpretation) point of view, the **wff of FOPL** may be divided into two categories, each consisting of

- (i) wffs, in each of which, **all** occurrences of variables are **bound**.
- (ii) wffs, in each of which, **at least one** occurrence of a variable is **free**.

The wffs of FOPL in which there is no occurrence of a free variable, are like *wffs* of PL in the sense that we can call each of the wffs as **True, False, consistent, inconsistent, valid, invalid etc.** Each such a formula is called **closed formula**. However, when a wff involves a free occurrence, then it is not possible to call such a wff as True, False etc. **Each of such a formula is called an open formula.**

For example: Each of the formulas: $\text{greater}(x, y)$, $\text{greater}(x, 3)$, $(\forall y) \text{greater}(x, y)$ has one free occurrence of variable x . Hence, each is an **open** formula. Each of the formulas: $(\forall x) (\exists y) \text{greater}(x, y)$, $(\forall y) \text{greater}(y, 1)$, $\text{greater}(9, 2)$, does not have free occurrence of any variable. Therefore each of these formulas is a closed formula.

Next we discuss some equivalences, and inequalities

The following equivalences hold for any two formulas $P(x)$ and $Q(x)$:

- (i) $(\forall x) P(x) \wedge (\forall x) Q(x) = (\forall x) (P(x) \wedge Q(x))$
- (ii) $(\exists x) P(x) \vee (\exists x) Q(x) = (\exists x) (P(x) \vee Q(x))$

But the following inequalities hold, in general:

- (iii) $(\forall x) (P(x) \vee Q(x)) \neq (\forall x) P(x) \vee (\forall x) Q(x)$
- (iv) $(\exists x) (P(x) \wedge Q(x)) \neq (\exists x) P(x) \wedge (\exists x) Q(x)$

We justify (iii) & (iv) below:

Let $P(x)$: x is odd natural number,

$Q(x)$: x is even natural number.

Then L.H.S of (iii) above states *for every natural number it is either odd or even, which is correct*. But R.H.S of (iii) states that *every natural number is odd or every natural number is even, which is not correct*.

Next, L.H.S. of (iv) states that: there is a natural number which is both even and odd, **which is not correct**. However, R.H.S. of (iv) says *there is an integer which is odd and there is an integer which is even, correct*.

Equivalences involving Negation of Quantifiers

- (v) $\sim (\forall x) P(x) = (\exists x) \sim P(x)$
- (iv) $\sim (\exists x) P(x) = (\forall x) \sim P(x)$

Examples: For each of the following closed formula, Prove

- (i) $(\forall x) P(x) \wedge (\exists y) \sim P(y)$ is inconsistent.
- (ii) $(\forall x) P(x) \rightarrow (\exists y) P(y)$ is valid

Solution: (i) Consider

$$\begin{aligned} & (\forall x) P(x) \wedge (\exists y) \sim P(y) \\ &= (\forall x) P(x) \wedge \sim (\forall y) P(y) \text{ (taking negation out)} \end{aligned}$$

But we know for each bound occurrence, a variable is dummy, and can be replaced in the whole scope of the variable uniformly by another free variable. Hence,

$$R = (\forall x) P(x) \wedge \sim (\forall x) P(x)$$

Each conjunct of the formula is either

True or False and, hence, can be thought of as a formula of PL, in stead of formula of FOPL, Let us replace $(\forall x) (P(x))$ by Q , a formula of PL.

$$R = Q \wedge \sim Q = \text{False}$$

Hence, the proof.

(ii) Consider

$$(\forall x) P(x) \rightarrow (\exists y) P(y)$$

Replacing ' \rightarrow ' we get

$$= \sim (\forall x) P(x) \vee (\exists y) P(y)$$

$$= (\exists x) \sim P(x) \vee (\exists y) P(y)$$

$$= (\exists x) \sim P(x) \vee (\exists x) P(x) \text{ (renaming } x \text{ as } y \text{ in the second disjunct)}$$

In other words,

$$= (\exists x) (\sim P(x) \vee P(x)) \text{ (using equivalence)}$$

The last formula states: *there is at least one element say b, for $\sim P(b) \vee P(b)$ holds i.e., for b, either P(b) is False or P(b) is True.*

But, as P is a predicate symbol and b is a constant $\sim P(b) \vee P(b)$ must be True. Hence, the proof.

Ex. 1 Let P(x) and Q(x) represent “x is a rational number” and “x is a real number,” respectively. Symbolize the following sentences:

- (i) Every rational number is a real number.
- (ii) Some real numbers are rational numbers.
- (iii) Not every real number is a rational number.

Ex. 2 Let C(x) mean “x is a used-car dealer,” and H(x) mean “x is honest.” Translate each of the following into English:

- (i) $(\exists x)C(x)$
- (ii) $(\exists x) H(x)$
- (iii) $(\forall x)C(x) \rightarrow \sim H(x)$
- (iv) $(\exists x) (C(x) \wedge H(x))$
- (v) $(\exists x) (H(x) \rightarrow C(x)).$

Ex. 3 Prove the following:

- (i) $P(a) \rightarrow \sim ((\exists x) P(x))$ is consistent.
- (ii) $(\forall x) P(x) \vee ((\exists y) \sim P(y))$ is valid.

1.3 PRENEX NORMAL FORM

In order to facilitate problem solving through PL, we discussed two normal forms, viz, the conjunctive normal form **CNF** and the disjunctive normal form **DNF**. In **FOPL**, there is a normal form called the **prenex normal form**. The use of a prenex normal form of a formula simplifies the proof procedures, to be discussed.

Definition A formula G in FOPL is said to be in a **prenex normal form** if and only if the formula G is in the form

$$(Q_1x_1) \dots (Q_n x_n) P$$

where each $(Q_i x_i)$, for $i = 1, \dots, n$, is either $(\forall x_i)$ or $(\exists x_i)$, and P is a quantifier free formula. The expression $(Q_1x_1) \dots (Q_n x_n)$ is called the **prefix** and P is called the **matrix of the formula G**.

Examples of some formulas in prenex normal form:

- (i) $(\exists x) (\forall y) (R(x, y) \vee Q(y)), (\forall x) (\forall y) (\sim P(x, y) \rightarrow S(y)),$
- (ii) $(\forall x) (\forall y) (\exists z) (P(x, y) \rightarrow R(z)).$

Next, we consider a method of transforming a given formula into a prenex normal form. For this, first we discuss equivalence of formulas in FOPL. Let us recall that two formulas E and G are **equivalent**, denoted by $E = G$, *if and only if the truth values of F and G are identical under every interpretation*. The pairs of equivalent formulas given in Table of equivalent Formulas of previous unit are still valid as these are quantifier-free formulas of FOPL. However, there are pairs of equivalent formulas of FOPL that contain quantifiers. Next, we discuss these additional pairs of equivalent formulas. We introduce some notation specific to FOPL: *the symbol G denote a formula that does not contain any free variable x* . Then we have the following pairs of equivalent formulas, where Q denotes a quantifier which is either \forall or \exists . Next, we introduce four laws for **pairs of equivalent formulas**.

In the rest of the discussion of FOPL, $P[x]$ is used to denote the fact that x is a free variable in the formula P , for example, $P[x] = (\forall y) P(x, y)$. Similarly, $R[x, y]$ denotes that variables x and y occur as free variables **in the formula R** Some of these equivalences, we have discussed earlier.

Then, the following laws involving quantifiers hold good in FOPL

- (i) $(Qx) P[x] \vee G = (Qx) (P[x] \vee G).$
- (ii) $(Qx) P[x] \wedge G = (Qx) (P[x] \wedge G).$

In the above two formulas, Q may be either \forall or \exists .

- (iii) $\sim ((\forall x) P[x]) = (\exists x) (\sim P[x]).$
- (iv) $\sim ((\exists x) P[x]) = (\forall x) (\sim P[x]).$
- (v) $(\forall x) P[x] \wedge (\forall x) H[x] = (\forall x) (P[x] \wedge H[x]).$
- (vi) $(\exists x) P[x] \vee (\exists x) H[x] = (\exists x) (P[x] \vee H[x]).$

That is, the universal quantifier \forall and the existential quantifier \exists can be distributed respectively over \wedge and \vee .

But we must be careful about *(we have already mentioned these inequalities)*

- (vii) $(\forall x) E[x] \vee (\forall x) H[x] \neq (\forall x) (P[x] \vee H[x])$ and
- (viii) $(\exists x) P[x] \wedge (\exists x) H[x] \neq (\exists x) (P[x] \wedge H[x])$

Steps for Transforming an FOPL Formula into Prenex Normal Form

Step 1 Remove the connectives ' \leftrightarrow ' and ' \rightarrow ' using the equivalences

$$P \leftrightarrow G = (P \rightarrow G) \wedge (G \rightarrow P)$$

$$P \rightarrow G = \sim P \vee G$$

Step 2 Use the equivalence to remove even number of \sim 's

$$\sim (\sim P) = P$$

Step 3 Apply De Morgan's laws in order to bring the negation signs immediately before atoms.

$$\begin{aligned}\sim (P \vee G) &= \sim P \wedge \sim G \\ \sim (P \wedge G) &= \sim P \vee \sim G\end{aligned}$$

and the quantification laws

$$\begin{aligned}\sim ((\forall x) P[x]) &= (\exists x) (\sim P[x]) \\ \sim ((\exists x) P[x]) &= (\forall x) (\sim P[x])\end{aligned}$$

Step 4 rename bound variables **if necessary**

Step 5 Bring quantifiers to the left before any predicate symbol appears in the formula. This is achieved by using (i) to (vi) discussed above.

We have already discussed that, if all occurrences of a bound variable are replaced uniformly throughout by another variable not occurring in the formula, then the equivalence is preserved. Also, we mentioned under (vii) that \forall does not distribute over \wedge and under (viii) that \exists does not distribute over \vee . In such cases, in order to bring quantifiers to the left of the rest of the formula, we may have to first rename one of bound variables, say x , may be renamed as z , which does not occur either as free or bound in the other component formulas. And then we may use the following equivalences.

$$\begin{aligned}(\forall x) P[x] \vee (\forall z) H[z] &= (\forall x) (\forall z) (P[x] \vee H[z]) \\ (\exists x) P[x] \wedge (\exists z) H[z] &= (\exists x) (\exists z) (P[x] \wedge H[z])\end{aligned}$$

Example: Transform the following formulas into prenex normal forms:

- (i) $(\forall x) (Q(x) \rightarrow (\exists x) R(x, y))$
- (ii) $(\exists x) (\sim (\exists y) Q(x, y) \rightarrow ((\exists z) R(z) \rightarrow S(x)))$
- (iii) $(\forall x) (\forall y) ((\exists z) Q(z, y, z) \wedge ((\exists u) R(x, u) \rightarrow (\exists v) R(y, v)))$.

Part (i)

Step 1: By removing ' \rightarrow ', we get

$$(\forall x) (\sim Q(x) \vee (\exists x) R(x, y))$$

Step 2: By renaming x as z in $(\exists x) R(x, y)$ the formula becomes

$$(\forall x) (\sim Q(x) \vee (\exists z) R(z, y))$$

Step 3: As $\sim Q(x)$ does not involve z , we get

$$(\forall x) (\exists z) (\sim Q(x) \vee R(z, y))$$

Part (ii)

$$(\exists x) (\sim (\exists y) Q(x, y) \rightarrow ((\exists z) R(z) \rightarrow S(x)))$$

Step 1: Removing outer ' \rightarrow ' we get

$$(\exists x) (\sim (\sim ((\exists y) Q(x, y))) \vee ((\exists z) R(z) \rightarrow S(x)))$$

Step 2: Removing inner ' \rightarrow ', and simplifying $\sim (\sim ())$ we get

$$(\exists x) ((\exists y) Q(x, y) \vee (\sim ((\exists z) R(z)) \vee S(x)))$$

Step 3: Taking ' \sim ' inner most, we get

$$(\exists x) (\exists y) Q(x, y) \vee ((\forall z) \sim R(z) \vee S(x))$$

As first component formula $Q(x, y)$ does not involve z and $S(x)$ does not involve both y and z and $\sim R(z)$ does not involve y . Therefore, we may take out $(\exists y)$ and $(\forall z)$ so that, we get

$(\exists x) (\exists y) (\forall z) (Q(x, y) \vee (\sim R(z) \vee S(x)))$, which is the required formula in prenex normal form.

Part (iii)

$$(\forall x) (\forall y) ((\exists z) Q(x, y, z) \wedge ((\exists u) R(x, u) \rightarrow (\exists v) R(y, v)))$$

Step 1: Removing ' \rightarrow ', we get

$$(\forall x) (\forall y) ((\exists z) Q(x, y, z) \wedge (\sim ((\exists u) R(x, u)) \vee (\exists v) R(y, v)))$$

Step 2: Taking ' \sim ' inner most, we get

$$(\forall x) (\forall y) ((\exists z) Q(x, y, z) \wedge ((\forall u) \sim R(x, u) \vee (\exists v) R(y, v)))$$

Step 3: As variables z, u & v do not occur in the rest of the formula except the formula which is in its scope, therefore, we can take all quantifiers outside, preserving the order of their occurrences, Thus we get

$$(\forall x) (\forall y) (\exists z) (\forall u) (\exists v) (Q(x, y, z) \wedge (\sim R(x, u) \vee R(y, v)))$$

Ex: 4 (i) Transform the formula $(\forall x) P(x) \rightarrow (\exists x) Q(x)$ into prenex normal form.

(ii) Obtain a prenex normal form for the formula

$$(\forall x) (\forall y) ((\exists z) (P(x, y) \wedge P(y, z)) \rightarrow (\exists u) Q(x, y, u))$$

1.4 (SKOLEM) STANDARD FORM

A further refinement of Prenex Normal Form (PNF) called (Skolem) Standard Form, is the basis of problem solving through Resolution Method. The Resolution Method will be discussed in the next unit of the block.

The **Standard Form of a formula of FOPL** is obtained through the following three steps:

- (1) The given formula should be converted to Prenex Normal Form (PNF), and then
- (2) Convert the Matrix of the PNF, i.e, quantifier-free part of the PNF into conjunctive normal form
- (3) Skolemization: Eliminate the existential quantifiers using skolem constants and functions

Before illustrating the process of conversion of a formula of FOPL to Standard Normal Form, through examples, we discuss briefly skolem functions.

Skolem Function

We in general, mentioned earlier that $(\exists x) (\forall y) P(x, y) \neq (\forall y) (\exists x) P(x, y) \dots \dots (1)$

For example, if $P(x, y)$ stands for the relation ' $x > y$ ' in the set of integers, then the L.H.S. of the inequality (i) above states: *some (fixed) integer (x) is greater than all integers (y)*. This statement is False.

On the other hand, R.H.S. of the inequality (1) states: *for each integer y, there is an integer x so that $x > y$* . This statement is True.

The difference in meaning of the two sides of the inequality arises because of the fact that on L.H.S. x in $(\exists x)$ is independent of y in $(\forall y)$ **whereas** on R.H.S x of dependent on y . In other words, x on L.H.S. of the inequality can be replaced by some constant say ' c ' whereas on the right hand side x is some function, say, $f(y)$ of y .

Therefore, the two parts of the inequality (i) above may be written as

$$\text{L.H.S. of (1)} = (\exists x) (\forall y) P(x, y) = (\forall y) P(c, y),$$

Dropping x because there is no x appearing in $(\forall y) P(c,y)$

$$\text{R.H.S. of (1)} = (\forall y) (\exists x) P(f(y),y) = (\forall y) P(f(y), y)$$

The above argument, in essence, explains what is meant by each of the terms viz. *skolem constant, skolem function and skolemisation*.

The constants and functions which replace existential quantifiers are respectively called **skolem constants and skolem functions**. The process of replacing all existential variables by skolem constants and variables is called **skolemisation**.

A form of a formula which is obtained after applying the steps for

- (i) reduction to PNF and then to
- (ii) CNF and then
- (iii) applying skolemization is called **Skolem Standard Form** or just **Standard Form**.

We explain through examples, the skolemisation process after PNF and CNF have already been obtained.

Example: Skolemize the following:

$$(i) (\exists x_1) (\exists x_2) (\forall y_1) (\forall y_2) (\exists x_3) (\forall y_3) P(x_1, x_2, x_3, y_1, y_2, y_3)$$

$$(ii) (\exists x_1) (\forall y_1) (\exists x_2) (\forall y_2) (\exists x_3) P(x_1, x_2, x_3, y_1, y_2) \wedge (\exists x_1) (\forall y_3) (\exists x_2) (\forall y_4) Q(x_1, x_2, y_3, y_4)$$

Solution (i) As existential quantifiers x_1 and x_2 precede all universal quantifiers, therefore, x_1 and x_2 are to be replaced by *constants*, but by distinct constants, say by 'c' and 'd' respectively. As existential variable x_3 is preceded by universal quantifiers y_1 and y_2 , therefore, x_3 is replaced by some function $f(y_1, y_2)$ of the variables y_1 and y_2 . After making these substitutions and dropping universal and existential variables, we get the skolemized form of the given formula as $(\forall y_1) (\forall y_2) (\forall y_3) (c, d, f(y_1, y_2), y_1, y_2, y_3)$.

Solution (ii) As a first step we must bring all the quantifications in the beginning of the formula through Prenex Normal Form reduction. Also,

$$\begin{aligned} & (\exists x) \dots P(x, \dots) \wedge (\exists x) \dots Q(x, \dots) \neq (\exists x) (\dots P(x) \wedge \dots Q(x, \dots)), \\ & \text{therefore, we rename the second occurrences of quantifiers } (\forall x_1) \text{ and } (\forall x_2) \text{ by} \\ & \text{renaming these as } x_5 \text{ and } x_6. \text{ Hence, after renaming and pulling out all the} \\ & \text{quantifications to the left, we get} \\ & (\exists x_1) (\forall y_1) (\exists x_2) (\forall y_2) (\exists x_3) (\exists x_5) (\forall y_3) (\exists x_6) (\forall y_4) \\ & (P(x_1, x_2, x_3, y_1, y_2) \wedge Q(x_5, x_6, y_3, y_4)) \end{aligned}$$

Then the existential variable x_1 is independent of all the universal quantifiers. Hence, x_1 may be replaced by a constant say, 'c'. Next x_2 is preceded by the universal quantifier y_1 hence, x_2 may be replaced by $f(y_1)$. The existential quantifier x_3 is preceded by the universal quantifiers y_1 and y_2 . Hence x_3 may be replaced by $g(y_1, y_2)$. The existential quantifier x_5 is preceded by again universal quantifier y_1 and y_2 . In other words, x_5 is also a function of y_1 and y_2 . But, we have to use a different function symbol say h and replace x_5 by $h(y_1, y_2)$. Similarly x_6 may be replaced by $j(y_1, y_2, y_3)$.

Thus, (Skolem) Standard Form becomes

$$(\forall y_1) (\forall y_2) (\forall y_3) (P(c, f(y_1), g(y_1, y_2), y_1, y_2) \wedge Q(h(y_1, y_2), j(y_1, y_2, y_3))).$$

Ex 5. Obtain a (skolem) standard form for each of the following formula:

- (i) $(\exists x) (\forall y) (\forall v) (\exists z) (\forall w) (\exists u) P(x, y, z, u, v, w)$
- (ii) $(\forall x) (\exists y) (\exists z) ((P(x, y) \vee \sim Q(x, z)) \rightarrow R(x, y, z))$

1.5 APPLICATIONS OF FOPL

We have developed tools of FOPL for solving problems requiring logical reasoning.

Now, we attempt solve the problem mentioned in the introduction the unit to show insufficiency of Propositional Logic.

Example: Every man is mortal. Raman is a man. Show that Raman is mortal. The problem can be symbolized as:

- (i) $(\forall x) (MAN(x) \rightarrow MORTAL(x))$.
- (ii) $MAN(Roman)$.

To show

- (iii) $Mortal(Raman)$

Solution:

By Universal Instantiation of (i) with constant Raman, we get

- (iv) $Man(Raman) \rightarrow Mortal(Raman)$

Using Modus Ponens with (iii) & (iv) we get $Mortal(Raman)$

Ex: 6 No used-car dealer buys a used car for his family. Some people who buy used cars for their families are absolutely dishonest. Conclude that some absolutely dishonest people are not used-car dealers.

Ex: 7 Some patients like all doctors. No patient likes any quack. Therefore, no doctor is a quack.

When it is convenient, we shall regard a set of literals as synonymous with a clause. For example, $P \vee Q \vee \sim R = \{P, Q, \sim R\}$. A clause consisting of r literals is called an r -literal clause. A one-literal clause is called a unit clause. When a clause contains no literal, we call it the empty clause. Since, the empty clause is always false. We customarily denote the empty clause by **False**.

The disjunctions $(\sim P(x, f(x)) \vee R(x, f(x), g(x)))$ and $Q(x, g(x)) \vee R(x, f(x), g(x))$ of the standard form

$$\sim P(x, f(x)) \vee R(x, f(x), g(x)) \wedge Q(x, g(x)) \vee R(x, f(x), g(x))$$

are clauses. A set S of clauses is regarded as a conjunction of all clauses in S , (1) with the condition that every variable that occurs in S is considered governed by a universal quantifier. By this convention, a standard form can be simply represented by a set of clauses.

For example, the standard form the above mentioned formula of (1) can be represented by the set.

$$\{P(x, f(x)) \vee R(x, f(x), g(x)), Q(x, g(x)) \vee R(x, f(x), g(x))\}.$$

Example:

Find a standard form for the following formula:

- a) $\sim ((\forall x) P(x) \rightarrow (\exists y) (\forall z) Q(y, z))$

Solution: Removing the logical symbol ' \rightarrow ', we get

$$\sim (\sim (\forall x) P(x)) \vee (\exists y) (\forall z) Q(y, z)$$

Taking ' \sim ' inside, we get

$$= (\forall x) P(x) \wedge \sim (\exists y) (\forall z) Q(y, z)$$

Again taking \sim inside, we get
 $= (\forall x) P(x) \wedge (\forall y) (\exists z) \sim Q(y, z)$

As variables x , y and z do not occur anywhere else, except within their respective scopes, therefore, the quantifiers may be taken in the beginning of the formula without any changes. Hence, we get
 $(\forall x) (\forall y) (\exists z) (P(x) \wedge \sim Q(y, z))$
 which is the required standard form.

Notation: Once, the standard form is obtained, there are no existential quantifications left in the formula. Also universal quantifications are dropped, because whatever variables appear in the rest of the formula can have only universal quantification and hence universal quantifications are implied. Next, if the standard form (without any quantifiers appearing) which is by definition also in CNF, is of the form:

$C_1 \wedge C_2 \wedge \dots \wedge C_n$ then, we may denote the standard form as a set $\{c_1, c_2, \dots, c_n\}$.

For example, in the previous example, the standard normal form was
 $(\forall x) (\forall y) (\forall z) (P(x) \wedge \sim Q(y, z))$, after dropping quantifiers becomes
 $P(x) \wedge \sim Q(y, z)$.

The last expression can be written as $\{P(x), \sim Q(y, z)\}$.

Ex: 8 Obtain a standard form of the formula

$$(\exists x) (\forall y) (\forall z) (\exists u) (\forall v) (\exists w) P(x, y, z, u, v, w).$$

Ex: 9 Obtain a standard form of the formula

$$(\forall x) (\exists y) (\exists z) ((\sim P(x, y) \wedge Q(x, z)) \vee R(x, y, z)).$$

Ex: 10 Using Show that G is logical conclusion of H_1 and H_2 , where

$$H_1 : (\forall x) (C(x) \rightarrow (W(x) \wedge R(x)))$$

$$H_2 : (\exists x) (C(x) \wedge O(x))$$

$$G : (\exists x) (O(x) \wedge R(x))$$

Ex: 11 Using resolution method, solve the following logic problem.

- (i) Some patients like all doctors.
- (ii) No patient likes any quack.
- (iii) Therefore, no doctor is a quack.

Ex: 12 Conclude that some of the officials were drug pushers where we know the following

- (i) The custom officials searched everyone who entered this country who was not a VIP.
- (ii) Some of the drug pushers entered this country and they were only searched by drug pushers.
- (iii) No drug pusher was a VIP.
- (iv) Some of the officials were drug pushers.

Ex: 13 From the given statement: *Everyone who saves money earns interest*, conclude that if there is no interest, then nobody saves money.

1.6 SUMMARY

In this unit, initially, we discuss how inadequacy of PL to solve even simple problems, requires some extension of PL or some other formal inferencing system so as to compensate for the inadequacy. First Order Predicate Logic (FOPL), is such an extension of PL that is discussed in the unit.

Next, syntax of proper structure of a formula of FOPL is discussed. In this respect, a number of new concepts including those of quantifier, variable, constant, term, free and bound occurrences of variables; closed and open wff, consistency/validity of wffs etc. are introduced.

Next, two normal forms viz. Prenex Normal Form (PNF) and Skolem Standard Normal Form are introduced. Finally, tools and techniques developed in the unit, are used to solve problems involving logical reasoning.

1.7 SOLUTIONS/ANSWERS

Ex. 1 (i) $(\forall x) (P(x) \rightarrow Q(x))$
(ii) $(\exists x) (P(x) \wedge Q(x))$
(iii) $\sim (\forall x) (Q(x) \rightarrow P(x))$

Ex. 2

- (i) There is (at least) one (person) who is a used-car dealer.
- (ii) There is (at least) one (person) who is honest.
- (iii) All used-car dealers are dishonest.
- (iv) (At least) one used-car dealer is honest.
- (v) There is at least one thing in the universe, (for which it can be said that) if that something is Honest then that something is a used-car dealer

Note: the above translation is not the same as: Some no gap one honest, is a used-car dealer.

Ex 3: (i) After removal of ' \rightarrow ' we get the given formula

$$= \sim P(a) \vee \sim ((\exists x) P(x))$$

$$= \sim P(a) \vee (\forall x) (\sim P(x))$$

Now $P(a)$ is an atom in PL which may assume any value T or F. On taking $P(a)$ as F the given formula becomes T, hence, consistent.

(ii) The formula can be written

$(\forall x) P(x) \vee \sim (\forall x) (P(x))$, by taking negation outside the second disjunct and then renaming.

The $(\forall x) P(x)$ being closed is either T or F and hence can be treated as formula of PL. Let $\forall x P(x)$ be denoted by Q. Then the given formula may be denoted by $Q \vee \sim Q = \text{True (always)}$ Therefore, formula is valid.

Ex: 4 (i) $(\forall x) P(x) \rightarrow (\exists x) Q(x) = \sim ((\forall x) P(x)) \vee (\exists x) Q(x)$ (by removing the connective \rightarrow)

$$= (\exists x) (\sim P(x)) \vee (\exists x) Q(x) \text{ (by taking '\sim' inside)}$$

$$= (\exists x) (\sim P(x) \vee Q(x)) \text{ (By taking distributivity of } \exists x \text{ over } \vee)$$

Therefore, a prenex normal form of $(\forall x) P(x) \rightarrow (\exists x) Q(x)$ is $(\exists x) (\sim P(x) \vee Q(x))$.

(ii) $(\forall x) (\forall y) ((\exists z) (P(x, y) \wedge P(y, z)) \rightarrow (\exists u) Q(x, y, u))$ (removing the connective \rightarrow)

$$= (\forall x) (\forall y) (\sim ((\exists z) (P(x, z) \wedge P(y, z))) \vee (\exists u) Q(x, y, u)) \quad \text{(using De Morgan's Laws)}$$

$$= (\forall x) (\forall y) ((\forall z) (\sim P(x, z) \vee \sim P(y, z)) \vee (\exists u) Q(x, y, u))$$

$$= (\forall x) (\forall y) (\forall z) (\sim P(x, z) \vee \sim P(y, z) \vee (\exists u) Q(x, y, u))$$

$$\vee \sim P(y, z) \vee Q(x, y, u) \quad (\text{as } z \text{ and } u \text{ do not occur in the rest of the formula except their respective scopes})$$

Therefore, we obtain the last formula as a prenex normal form of the first formula.

Ex 5 (i) In the given formula $(\exists x)$ is not preceded by any universal quantification. Therefore, we replace the variable x by a (skolem) constant c in the formula and drop $(\exists x)$.

Next, the existential quantifier $(\exists z)$ is preceded by two universal quantifiers viz., v and y . we replace the variable z in the formula, by some function, say, $f(v, y)$ and drop $(\exists z)$. Finally, existential variable $(\exists u)$ is preceded by three universal quantifiers, viz., $(\forall y)$, $(\forall v)$ and $(\forall w)$. Thus, we replace in the formula the variable u by, some function $g(y, v, w)$ and drop the quantifier $(\exists u)$. Finally, we obtain the standard form for the given formula as

$$(\forall y) (\forall v) (\forall w) P(x, y, z, u, v, w)$$

(ii) First of all, we reduce the matrix to CNF.

$$\begin{aligned} &= (P(x, y) \vee \sim Q(x, z)) \rightarrow R(x, y, z) \\ &= (\sim P(x, y) \wedge Q(x, z)) \vee R(x, y, z) \\ &= (\sim P(x, y) \vee R(x, y, z)) \wedge (Q(x, z) \vee R(x, y, z)) \end{aligned}$$

Next, in the formula, there are two existential quantifiers, viz., $(\exists y)$ and $(\exists z)$. Each of these is preceded by the only universal quantifier, viz. $(\forall x)$.

Thus, each variable y and z is replaced by a function of x . But the two functions of x for y and z must be different functions. Let us assume, variable, y is replaced in the formula by $f(x)$ and the variable z is replaced by $g(x)$. Thus the initially given formula, after dropping of existential quantifiers is in the standard form:

$$(\forall x) ((\sim P(x, y) \vee R(x, y, z)) \wedge (Q(x, z) \vee R(x, y, z)))$$

Ex: 6 Let

- (i) $U(x)$, denote x is a used-car dealer,
- (ii) $B(x)$ denote x buys a used car for his family, and
- (iii) $D(x)$ denote x is absolutely dishonest,

The given problem can be symbolized as

- (i) $(\forall x) (U(x) \rightarrow \sim B(x))$
- (ii) $(\exists x) (B(x) \wedge D(x))$.
- (iii) $(\exists x) (D(x) \wedge \sim U(x))$ (to be shown)

By Existential Instantiation of (iii) we get that for some fixed c

$$(iv) B(c) \wedge D(c)$$

Using Universal Instantiation of (i), with c we get

$$(v) U(c) \rightarrow \sim B(c)$$

Using simplification of (iv) we get

$$(vi) B(c) \quad (vii) D(c)$$

Using Modus Tollens with (v) and (vi) taking $B(c) = \sim(\sim(B(c)))$, we get

$$(vii) \sim U(c)$$

Using conjunction of (vii) & (viii), we get

$$(ix) D(c) \wedge \sim U(c)$$

Using Existential Generalization of (ix) we get

$$(\exists x) (D(x) \wedge \sim U(x)),$$

which is required to be established.

Ex: 7

Let us use the notation for the predicates of the problem as follows:

P(x) : x is a patient,
D(x): x is a doctor,
Q(x): x is a quack,
L(x, y): x likes y.

The problem can be symbolized as follows:

- (i) $(\exists x) ((P(x) \wedge (\forall y) (D(y) \rightarrow L(x, y)))$
- (ii) $(\forall x) (P(x) \rightarrow (\forall y) (Q(y) \rightarrow \sim L(x, y)))$
- (iii) $(\forall x) (D(x) \rightarrow \sim Q(x))$. (to be shown)

Taking Existential Instantiation of (i), we get a specific c such that

- (iv) $P(c) \wedge (\forall y) (D(y) \rightarrow L(c, y))$
By simplification of (iv), we get
- (v) $P(c)$
- (vi) $(\forall y) (D(y) \rightarrow L(c, y))$

By Universal Instantiation of (ii) with x as c (because the fact in (ii) is true for all values of x and for the already considered value c also. This type of association of an already used value c may not be allowed in Existential Instantiation)

we get

- (vii) $P(c) \rightarrow \forall(y) (Q(y) \rightarrow \sim L(c, y))$

Using Modus Ponens with (v) and (vii) we get

- (viii) $\forall(y) (Q(y) \rightarrow \sim L(c, y))$

As $(\forall y)$ is the quantifier appearing in both (vi) and (viii),

we can say that for an arbitrary a, we have

- (ix) $D(a) \rightarrow L(c, a)$ for every a (from (vi)) and
- (x) $Q(a) \rightarrow \sim L(c, a)$ for every a (from (viii))

(Using the equivalent $P \rightarrow Q = \sim Q \rightarrow \sim P$, we get from (x):

- (xi) $L(c, a) \rightarrow \sim Q(a)$

Ex: 8 In the formula, $(\exists x)$ is preceded by no universal quantifiers, $(\exists u)$ is preceded by $(\forall y)$ and $(\forall z)$, and $(\exists w)$ by $(\forall y)$, $(\forall z)$ and $(\forall v)$. Therefore, we replace the existential variable x by a constant a, u by a two-place function f(y, z), and w by a three-place function g(y, z, v). Thus, we obtain the following standard form of the formula:

$$(\forall y) (\forall z) (\forall v) P(a, y, z, f(y, z), v, g(y, z, v)).$$

Ex: 9 As a first step, the matrix is transformed into the following conjunctive normal form:

$$(\forall x) (\exists y) (\exists z) ((\sim P(x, y) \vee R(x, y, z)) \vee R(x, y, z)) \wedge (Q(x, z) \vee R(x, y, z)).$$

As the existential variables $(\exists y)$ and $(\exists z)$ are both preceded by $(\forall x)$, the variables y and z are replaced, by one-place function f(x) and g(x) respectively.

In this way, we obtain the standard form of the formula as:

$$(\forall x) ((\sim P(x, f(x)) \vee R(x, f(x), g(x))) \wedge (Q(x, g(x)) \vee R(x, f(x), g(x)))).$$

Ex: 10 As we are going to use resolution method, we consider $\sim G: \sim (\exists x) (O(x) \wedge R(x))$ as an axiom.

We use the resolution method to show these clauses as unsatisfiable.

As H_1 unsolves only quantifier at extreme left, its standard form is:

$$H_1: \sim (x) \vee (W(x) \wedge R(x)) = (\sim (x) \vee W(x)) \wedge (\sim C(x) \vee R(x))$$

$$H_2: C(a) \wedge O(a) \quad (a \text{ for } \exists x)$$

For standard form of $\sim G$

$$\sim G = \sim (\exists x) (O(x) \wedge R(x)) = (\forall x) (\sim O(x) \vee \sim R(x))$$

$$\text{Standard form of } \sim G = \sim O(x) \vee \sim R(x)$$

Thus, the clauses of the wffs of the problem are:

- (i) $\sim C(x) \vee W(x)$
- (ii) $\sim C(x) \vee R(x)$ from H_1
- (iii) $C(a)$
- (iv) $O(a)$ from H_2
- (v) $\sim O(x) \vee \sim R(x)$ from $\sim G$.
- Resolving (ii) and (iii), we get
- (vi) $R(a)$ Resolving (iv) and (v), we get
- (vii) $\sim R(a)$ Resolvent (v) and (iv)
- Resolving (vi) and (vii) we get
- (viii) False

Hence, G is a logical consequence of F_1 and F_2 .

Ex : 11 Let us use the symbols:

$P(x)$: x is a patient

$D(x)$: x is a doctor

$Q(x)$: x is a quake

$L(x, y)$: x likes y .

Therefore, the given statements in the problem are symbolized as:

- (i) $(\exists x) (P(x) \wedge (\forall y) (D(y) \rightarrow L(x, y)))$
- (ii) $(\forall x) (P(x) \rightarrow (\forall y) (Q(y) \rightarrow \sim L(x, y)))$
- (iii) $(\forall x) (D(x) \rightarrow \sim Q(x))$.

The clauses which are obtained after reducing to standard form are:

- (iv) $P(a)$ from (i) to standard
- (v) $\sim D(y) \vee L(a, y)$ from (i) form are:
- (vi) $\sim P(x) \vee \sim Q(y) \vee \sim L(x, y)$ from (ii)
- (vii) $D(b)$ from \sim (iii)
- (viii) $Q(b)$ from $\sim G$ from \sim (iii)

Resolving (v) and (vii), we get

$$(ix) L(a, b)$$

Resolving (iv) and (vi) we get

$$(x) \sim Q(y) \vee \sim L(a, y)$$

Resolving (viii) and (x), we get

$$(xi) \sim L(a, b)$$

Resolving (ix) and (xi), we get

$$(xii) \text{ False}$$

Ex: 12 $E(x)$: x entered this country

$V(x)$: x was VIP,

$S(x, y)$: y searched x ,

$C(x)$: x was a custom official, and

$P(x)$: x was a drug pusher.

When symbolized, the known facts become:

- (i) $(\forall x) (E(x) \wedge \sim V(x) \rightarrow (\exists y) (S(x, y) \wedge C(y)))$
- (ii) $(\exists x) (P(x) \wedge E(x) \wedge (\forall y) (S(x, y) \rightarrow P(y)))$
- (iii) $(\forall x) (P(x) \rightarrow \sim V(x))$

and on symbolization, the conclusion becomes

- (iv) $(\exists x) (P(x) \wedge C(x))$.

As resolution method is to be used, we assume \sim (iv)

After converting to standard form, we get the clauses:

- (v) $\sim E(x) \vee V(x) \vee S(x, f(x))$ (from (i))
- (vi) $\sim E(x) \vee V(x) C(f(x))$ (from (i))
- (vii) $P(a)$ (from (ii))
- (viii) $E(a)$ (from (ii))
- (ix) $\sim S(a, y) \vee P(y)$ (from (ii))
- (x) $\sim P(x) \vee \sim V(x)$ (from (iii))
- (xi) $\sim P(x) \vee \sim C(x)$ (from (iv))

Resolving (vii) and (x) we get

- (xii) $\sim V(a)$

Resolving (vi) and (viii), we get

- (xiii) $V(a) \vee C(f(a))$

Resolving (xii) and (xiii), we get

- (xiv) $Cf(a)$

Resolving (v) and (viii), we get

- (xv) $V(a) \vee S(a, f(a))$

Resolving (xii) and (xv) we get

- (xvi) $S(a, f(a))$

Resolving (xvi) and (ix)

- (xvii) $P(f(a))$

Resolving (xvii) and (xi), we get

- (xviii) $\sim C(f(a))$

Resolving (xiv) and (xviii)

- (xix) False

Hence, the proof by resolution method.

Ex: 13 Let us use the symbols:

$S(x, y)$: x saves y,

$M(x)$: x is money,

$I(x)$: x is interest,

$E(x, y)$: x Earns y.

Then the given statement on symbolization becomes:

- (i) $(\forall x) ((\exists y) (S(x, y) \wedge M(y)) \rightarrow (\exists y) (I(y) \wedge E(x, y)))$
the conclusion on
- (ii) $\sim (\exists x) I(x) \rightarrow (\forall x) (\forall y) (S(x, y) \rightarrow \sim M(y))$
- (iii) $\sim (\sim (\exists x) I(x) \rightarrow (\forall x) (\forall y) (S(x, y) \rightarrow \sim M(y)))$

The negation of (ii) becomes (iii) $\sim ((\exists x) I(x) \rightarrow (\forall x) (\forall y) (S(x, y) \rightarrow \sim M(y)))$

After converting to standard form, we get the clauses:

- (iv) $\sim S(x, y) \vee \sim M(y) \vee I(f(x))$ (from (i))
- (v) $\sim S(x, y) \vee \sim M(y) \vee E(x, f(x))$. (from (ii))

- | | |
|------------------------------------|--------------|
| (vi) $\sim I(z)$ | (from (iii)) |
| (vii) $S(a, b)$ | (from (iii)) |
| (viii) $M(b)$ | (from (iii)) |
| Resolving (iv) and (vi), we get | |
| (ix) $\sim S(x, y) \vee \sim M(y)$ | |
| Resolving (vii) and (ix) we get | |
| (x) $\sim M(b)$ | |
| Resolving (viii) and (x), we get | |
| (xi) False | |

1.8 FURTHER READINGS

1. McKay, Thomas J., *Modern Formal Logic* (Macmillan Publishing Company, 1989).
2. Gensler, Harry J. *Symbolic Logic: Classical and Advanced Systems* (Prentice Hall of India, 1990).
3. Klenk, Virginia *Understanding Symbolic Logic* (Prentice Hall of India 1983)
4. Copi Irving M. & Cohen Carl, *Introduction Logic IX edition*, (Prentice Hall of India, 2001).
5. Carroll, Lewis, *Symbolic Logic & Game of Logic* (Dover Publication, 1955).
6. Wells, D.G., *Recreations in Logic* (Dover Publications, 1979).
7. Suppes Patrick, *Introduction to Logic* (Affiliated East-West Press, 1957).
8. Getmanova, Alexandra, *Logic* (Progressive Publishers, Moscow, 1989).
9. Crossely, J.N. et al *What is Mathematical Logic?* (Dover Publications, 1972).
10. Mendelson, Elliott: *Introduction to Mathematical Logic (Second Edition)* (D.Van Nostrand Company, 1979).

UNIT 2 DEDUCTIVE INFERENCE RULES AND METHODS

Structure	Page Nos.
2.0 Introduction	25
2.1 Objectives	26
2.2 Basic Inference Rules and Application in PL	26
2.3 Basic Inference Rules and Application in FOPL	31
2.4 Resolution Method in PL	37
2.5 Resolution Method in FOPL	40
2.6 Summary	44
2.7 Solutions/Answers	45
2.8 Further Readings	49

2.0 INTRODUCTION

In order to establish validity/invalidity of a conclusion C , in an argument, from a given set of facts/axioms A_1, A_2, \dots, A_n ; so far, we only know that *either* a truth table should be constructed for the formula $P: A_1 \wedge A_2 \wedge \dots \wedge A_n \rightarrow C$, *or* this formula should be converted to CNF or DNF through substitutions of equivalent formulas and simplifications. There are other alternative methods also. However, the problem with these methods is that as the number n of axioms becomes larger, the formula becomes complex (imagine $n = 50$) and the number of involved variables, say k , also, generally, increases. With number of variables k involved in the argument, the size of Truth-table becomes 2^k . For large k , the number of rows, i.e. 2^k becomes, almost unmanageable. Therefore, we need to search for alternative methods which instead of processing the whole of the argument as a single formula, process each of the *individual* formulas A_1, A_2, \dots , and C of the argument and their derivatives by applying some rules which preserve validity.

In **Section 3.2**, we introduce eight inference rules for drawing valid conclusions in PL. Next, in **Section 3.3**, we introduce four quantification rules, so that all the twelve inference rules are used to validate conclusions in FOPL. The methods of drawing valid conclusions, discussed so far, are cases of an *approach* of drawing valid conclusions, called **natural deduction approach** of making inferences in which the reasoning system initiates reasoning process from the axioms, uses inferencing rules and, if the conclusion can be validly drawn, then ultimately reaches the intended conclusion.

On the other hand, there is another *approach* called **Refutation approach** of drawing valid conclusions. According to this approach, negation of the intended conclusion is taken as an additional axiom. If the conclusion can be validly drawn from the axioms, then through application of inference rules, a contradiction is encountered, i.e., two formulas which are mutual negations, are encountered during the process of making inference.

Resolution method is a single rule refutation method. Resolution method and its applications for PL are discussed in **Section 3.4**. Resolution Method and its applications for FOPL are discussed in **Section 3.5**.

2.1 OBJECTIVES

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

- enumerate basic inference rules of PL and also be able to apply these in solving problems requiring PL reasoning;
- enumerate four basic quantification rules and be able to apply these rules along with basic rules of PL to solve problems involving FOPL reasoning;
- explain Resolution method for PL and apply it in solving problems requiring PL reasoning, and
- explain Resolution method for FOPL to solve problems involving FOPL reasoning.

2.2 BASIC INFERENCE RULES AND APPLICATIONS IN PL

In this section, we study a method which uses a number of *rules of inference* for drawing valid conclusions, and later we study *Resolution Method* for establishing validity of arguments.

We introduce eight rules of inference. Each of these rules has a specific name. In order to familiarize ourselves with

- what a rule of inference is ,
- how a rule is represented, and
- how a rule of inference helps us in solving problems.

We discuss in some detail, one of the rules known as Modus Ponens.

Rule 1 Modus Ponens (M. P.)

$$\text{Notations for M. P.: } \frac{P \rightarrow Q, P}{Q}$$

(The comma is read as 'and'. The rule may also be written as

$$\frac{P, P \rightarrow Q}{Q}, \text{ i.e., we may assume commutativity of comma})$$

The rule states that if formulas P and $P \rightarrow Q$ (of either propositional logic or predicate logic) are True then we can assume the Truth of Q .

The assumption is based on the fact that through truth-table method or otherwise we can show that if P and $P \rightarrow Q$, each is assigned truth value T then Q must have truth value T.

Consider the Table

P	Q	$P \rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

From the above table, we can see that P and $P \rightarrow Q$ both are True only in the first row and in the first row Q , the formula which is inferred, is also True.

The same is the reason for allowing use of other rules of inference in deducing new facts.

Rule 2 Modus Tollens (M. T.)

$$\frac{P \rightarrow Q, \sim Q}{\sim P}$$

The rule states if $P \rightarrow Q$ is True, but Q , the consequent of $P \rightarrow Q$ is False then the antecedent P of $P \rightarrow Q$ is also False.

The validity of the rule may again be established through truth-table as follows:

P	Q	$P \rightarrow Q$
T	T	T
T	F	F
F	T	T
F	F	T

In the above table $P \rightarrow Q$ is T and Q is False simultaneously only in the last row and in this row P , the formula which is inferred, is False.

Note: The validity of the rest of the rules will not be established. However, it is desirable that the students verify the validity of the other inference rules also through Truth-Table or otherwise.

Rule 3 Hypothetical Syllogism (H.P.)

$$\frac{P \rightarrow Q, Q \rightarrow R}{P \rightarrow R}$$

The rule states that if we assume that both the formulas $P \rightarrow Q$ and $Q \rightarrow R$ are True then we may assume $P \rightarrow R$ is also True.

Rule 4 Simplification (Simp.)

$$(i) \frac{P \wedge Q}{P} \text{ and } (ii) \frac{P \wedge Q}{Q}$$

The rule says that if $P \wedge Q$ is True then P can be assumed to be True (and similarly Q may be assumed to be True.)

Some of us may be surprised at the mention of the rule, thinking that if $P \wedge Q$ is True then P must be True. The symbol \wedge is generally read as 'and'. But the significance of the rule is that ' \wedge ' is merely a symbol and its meaning in the sense of 'and' comes only through this rule of inference.

Rule 5 Conjunction (Conj.)

$$\frac{P, Q}{P \wedge Q}$$

The rule states if formulas P and Q are simultaneously True then the formula $P \wedge Q$ can be assumed to be True.

Rule 6 Disjunctive Syllogism (D.S.)

$$(i) \frac{P \vee Q, \sim P}{Q} \text{ and } (ii) \frac{P \vee Q, \sim Q}{P}$$

The two rules above state that if it is given that (a) $P \vee Q$ is true and (b) one of P or Q is False, then other must be True

Rule 7 Addition (Add.)

$$(i) \frac{P}{P \vee Q} \text{ and } (ii) \frac{Q}{P \vee Q}$$

The rules state that if one of P and Q is assumed to be True, then we can assume $P \vee Q$ to be True.

Rule 8 Dilemma (Dil.)

$$\frac{P \rightarrow Q, R \rightarrow S, P \vee R}{Q \vee S}$$

The rule states that if both the formulas $P \rightarrow Q$ and $R \rightarrow S$ are assumed to be True and if $P \vee R$, i.e. disjunction of the antecedents is assumed to be 'True', then assume Truth of $Q \vee S$, which is disjunction of consequents.

We demonstrate how the above-mentioned rules of inference can be used in solving problems.

Example: Symbolize and construct a proof for the following valid argument using rules of inference:

(i) If you smoke or drink too much then you do not sleep well, and if you do not sleep well or do not eat well then you feel rotten, (ii) If you feel rotten, you do not exercise well and do not study enough, (iii) You do smoke too much, *therefore*, (iv) You do not study enough.

Solution: Let us symbolize the statements in the argument as follows:

S: You smoke too much

D: You drink too much

W: You sleep well

E: You eat well

R: You feel rotten

X : You exercise well

T: You study enough

Then the three given statements marked as (i), (ii) and (iii) are symbolized as follows:

(i) $((S \vee D) \rightarrow \sim W) \wedge ((\sim W \vee \sim E) \rightarrow R)$

(ii) $R \rightarrow (\sim X \wedge \sim T)$

(iii) S

(iv) $\sim T$.(To show)

Through simplification of (i), i.e., by using $\frac{P \wedge Q}{P}$, we get

(v) $S \vee D \rightarrow \sim W$

Using Add on (iii) $\left(\text{i.e. by using } \frac{S}{S \vee D}, \text{ we get} \right)$

(vi) $S \vee D$

and using syllogism on (v) and (vi) we get

(vii) $\sim W$

Again through simplification of (i), we get

(viii) $(\sim W \vee \sim E) \rightarrow R$

and by addition of (vii), we get

(ix) $\sim W \vee \sim E$

From (viii) and (ix) using M.P., we get

(x) R

Again using M.P. with (ii) & (x), we get

(xi) $\sim X \wedge \sim T$

Again using simplification of (xi) we get the required formula

(xi) $\sim T$

Example: Symbolize and construct a proof for the following valid argument: (i) If the Bible is literally true then the Earth was created in six days, (ii) If the Earth was created in six days then carbon dating techniques are useless and scientists are frauds, (iii) Scientists are not frauds, (iv) The Bible is literally true, *therefore*, (v) God does not exist.

Solution: Let us symbolize as follows:

B: Bible is literally true

E: The Earth was created in six days

C: Carbon dating techniques are useless

S: Scientists are frauds

G: God exists

Therefore the statements in the given arguments are symbolically represented as :

- (i) $B \rightarrow E$
- (ii) $E \rightarrow C \wedge S$
- (iii) $\sim S$
- (iv) B
- (v) $\sim G$ (to show)

Using M.P. on (i) and (iv), we get

(vi) E

Using M.P. on (ii) & (vi) we get

(vii) $C \wedge S$

Using Simp on (vii), we get

(viii) S

Using Addition on (viii), we get

(ix) $S \vee \sim G$

Using (D.S.) on (iii) & (ix) we get

(x) $\sim G$

The last statement is what is to be proved.

Remarks: In the above deduction, (iii) and (viii) are contradicting each other. In general, if in the process of derivation, we encounter two statement (like S and $\sim S$) which contradict each other, then we can **deduce any statement** even if the statement can never be True in any sense. Thus, if both S and $\sim S$ have already occurred in the process of derivation, then we can assume the truth of any statement. For example, we can assume the truth of the statement: 'Moon is made of green cheese'

The **technique, to prove any non-sense** statement say NON-SENSE in a situation where already two mutually contradicting statements say S and $\sim S$ have already been encountered, is the same as is used in deriving (ix) by applying Addition Rule to (viii).

Thus, once we encounter S , where $\sim S$ has already occurred, use Addition rule to get $S \vee \text{NON-SENSE}$ from S . Then use D.S. on $S \vee \text{NON-SENSE}$ and $\sim S$, we get NON-SENSE.

Ex.1 Given the following three statements:

- (i) *Matter always existed*
- (ii) *If there is God, then God created the universe.*
- (iii) *If God created the universe, then matter did not always exist.*

Show the truth of the statement: (iv) *There is no God.*

Ex.2 Using propositional logic, show that, if the following statements are assumed to be true:

- (i) *There is a moral law.*
- (ii) *If there is a moral law, then someone gave it.*
- (iii) *If someone gave the moral law, then there is God.*

then the following statement is also true:

- (iv) *There is GOD*
-

2.3 BASIC INFERENCING RULES AND APPLICATIONS IN FOPL

In the previous unit, we discussed eight inferencing rules of Propositional Logic (PL) and further discussed applications of these rules in exhibiting validity/invalidity of arguments in **PL**. In this section, the earlier eight rules are extended to include four more rules involving quantifiers for inferencing. Each of the new rules, is called a **Quantifier Rule**. The extended set of 12 rules is then used for validating arguments in First Order Predicate Logic (**FOPL**).

Before introducing and discussing the Quantifier rules, we briefly discuss why, at all, these rules are required. For this purpose, let us recall the argument discussed earlier, which Propositional Logic could not handle:

- (i) Every man is mortal.
- (ii) Raman is a man.
- (iii) Raman is mortal.

The equivalent symbolic form of the argument is given by:

- (i') $(\forall x) (\text{Man}(x) \rightarrow \text{Mortal}(x))$
- (ii') $\text{Man}(\text{Raman})$
- (iii') $\text{Mortal}(\text{Raman})$

If, instead of (i') we were given

- (iv) $\text{Man}(\text{Raman}) \rightarrow \text{Mortal}(\text{Raman})$,

(which is a formula of Propositional Logic also)

then using Modus Ponens on (ii') & (iv) in *Propositional Logic*, we would have obtained (iii') *Mortal (Raman)*.

However, from (i') & (ii') we cannot derive in Propositional Logic (iii'). This suggests that there should be mechanisms for dropping and introducing quantifier appropriately, i.e., in such a manner that *validity* of arguments is not violated. Without discussing the validity-preserving characteristics, we introduce the four Quantifier rules.

(i) Universal Instantiation Rule (U.I.):

$$\frac{(\forall x)p(x)}{p(a)}$$

Where is an a arbitrary constant.

The rule states if $(\forall x)p(x)$ is True, then we can assume $P(a)$ as True for any constant a (where a constant a is like Raman). It can be easily seen that the rule associates a formula $P(a)$ of Propositional Logic to a formula $(\forall x)p(x)$ of FOPL. The significance of the rule lies in the fact that once we obtain a formula like $P(a)$, then the reasoning process of Propositional Logic may be used. The rule may be used, whenever, its application seems to be appropriate.

(ii) Universal Generalisation Rule (U.G.)

$$\frac{P(a), \text{ for all } a}{(\forall x)p(x)}$$

The rule says that if it is known that for all constants a , the statement $P(a)$ is True, then we can, instead, use the formula $(\forall x)p(x)$.

The rule associates with a set of formulas $P(a)$ for all a of Propositional Logic, a formula $(\forall x)p(x)$ of FOPL.

Before using the rule, we must ensure that $P(a)$ is True for all a , Otherwise it may lead to wrong conclusions.

(iii) Existential Instantiation Rule (E. I.)

$$\frac{(\exists x) P(x)}{P(a)} \quad (E.I.)$$

The rule says if the Truth of $(\exists x) P(x)$ is known then we can assume the Truth of $P(a)$ for **some fixed** a . The rule, again, associates a formula $P(a)$ of Propositional Logic to a formula $(\forall x)p(x)$ of FOPL.

An inappropriate application of this rule may lead to *wrong* conclusions. The source of possible errors lies in the fact that the choice 'a' in the rule is *not arbitrary* and can not be known at the time of deducing $P(a)$ from $(\exists x) P(x)$.

If during the process of deduction some other $(\exists y) Q(y)$ or $(\exists x) (R(x))$ or even another $(\exists x)P(x)$ is encountered, then each time a new constant say b, c etc. should be chosen to infer $Q(b)$ from $(\exists y) Q(y)$ or $R(c)$ from $(\exists x) (R(x))$ or $P(d)$ from $(\exists x) P(x)$.

(iv) Existential Generalization Rule (E.G)

$$\frac{P(a)}{(\exists x)P(x)} \quad (E.G)$$

The rule states that if $P(a)$, a formula of Propositional Logic is True, then the Truth of $(\exists x) P(x)$, a formula of FOPL, may be assumed to be True.

The Universal Generalisation (U.G) and Existential Instantiation rules should be applied with utmost care, however, other two rules may be applied, whenever, it appears to be appropriate.

Next, The purpose of the two rules, viz.,

(i) Universal Instantiation Rule (U. I.)

(iii) Existential Rule (E. I.)

is to associate formulas of Propositional Logic (PL) to formulas of FOPL in a manner, the validity of arguments due to these associations, is not disturbed. Once, we get formulas of PL, then any of the eight rules of inference of PL may be used to validate conclusions and solve problems requiring logical reasoning for their solutions.

The purpose of the other Quantification rules viz. for generalisation, i.e.,

(ii) $\frac{P(a), \text{ for all } a}{(\forall x) P(x)}$

(iv) $\frac{P(a)}{(\exists x) P(x)}$

is that the conclusion to be drawn in FOPL is not generally a formula of PL but a formula of FOPL. However, while making inference, we may be first associating formulas of PL with formulas of FOPL and then use inference rules of PL to conclude formulas in PL. But the conclusion to be made in the problem may correspond to a formula of FOPL. These two generalisation rules help us in associating formulas of FOPL with formulas of PL.

Example: Tell, supported with reasons, which one of the following is a correct inference and which one is not a correct inference.

- (i) To conclude $F(a) \wedge G(a) \rightarrow H(a) \wedge I(a)$
from $(\forall x)(F(x) \wedge G(x)) \rightarrow H(x) \wedge I(x)$
using Universal Instantiation (U.I.)

The above inference or conclusion is *incorrect* in view of the fact that the scope of universal quantification is only the formula: $F(x) \wedge G(x)$ and not the whole of the formula.

The occurrences of x in $H(x) \wedge I(x)$ are free occurrences. Thus, one of the correct inferences would have been:

$$F(a) \wedge G(a) \rightarrow H(x) \wedge I(x)$$

- (ii) To conclude $F(a) \wedge G(a) \rightarrow H(a) \wedge I(a)$ from
 $(\forall x)(F(x) \wedge G(x) \rightarrow H(x) \wedge I(x))$ using U.I.
The conclusion is correct in view of the argument given in (i) above.

- (iii) To conclude $\sim F(a)$ for an arbitrary a , from $\sim(\forall x)F(x)$ using U.I.

The conclusion is incorrect, because actually
 $\sim(\forall x)F(x) = (\exists x)\sim F(x)$

Thus, the inference is not a case of U.I., but of Existential Instantiation (E.I.)

Further, as per restrictions, we can not say for which a , $\sim F(x)$ is True. Of course, $\sim F(x)$ is true for some constant, but not necessarily for a pre-assigned constant a .

- (iv) to conclude $(F(b) \wedge G(b) \rightarrow H(c))$
from $(\exists x)(F(b) \wedge G(x) \rightarrow H(c))$

Using E.I. is *not* correct

The reason being that the constant to be substituted for x cannot be assumed to be the same constant b , being given in advance, as an argument of F . However,

to conclude $(F(b) \wedge G(a) \rightarrow H(c))$
from $(\exists x)(F(b) \wedge G(x) \rightarrow H(c))$ is correct.

Ex. 3: Tell for each of the following along with appropriate reasoning, whether it is a case of correct/incorrect reasoning.

- (i) To conclude

$F(a) \wedge G(a)$ by applying E.I. to

$$(\exists x) F(x) \wedge \exists (x) G(x)$$

(ii) To conclude $F(a) \vee (G(a) \wedge H(a))$ from

$$(\exists x) F(x) \vee (G(x) \wedge H(x))$$

(iii) To conclude $(\exists x) (\sim F(x) \rightarrow \sim G(x))$ from

$$\sim F(a) \rightarrow \sim G(a)$$

(iv) To conclude $\sim ((\exists x)(F(x) \wedge G(x)))$ from $\sim (F(a) \wedge G(a))$

Step for using Predicate Calculus as a Language for Representing Knowledge & for Reasoning:

Step 1: Conceptualisation: First of all, all the relevant entities and the relations that exist between these entities are explicitly enumerated. Some of the implicit facts like, 'a person dead once is dead for ever' have to be explicated.

Step 2: Nomenclature & Translation: Giving appropriate names to objects and relations. And then translating the given sentences given in English to formulas in FOPL. Appropriate names are essential in order to guide a reasoning system based on FOPL. It is well-established that no reasoning system is complete. In other words, a reasoning system may need help in arriving at desired conclusion.

Step 3: Finding appropriate sequence of reasoning steps, involving selection of appropriate rule and appropriate FOPL formulas to which the selected rule is to be applied, to reach the conclusion.

Applications of the 12 inferencing rules (8 of Propositional Logic and 4 involving Quantifiers.)

Example: Symbolize the following and then construct a proof for the argument:

- (i) Anyone who repairs his own car is highly skilled and saves a lot of money on repairs
- (ii) Some people who repair their own cars have menial jobs. Therefore,
- (iii) Some people with menial jobs are highly skilled.

Solution: Let us use the notation:

$P(x)$:	x is a person
$S(x)$:	x saves money on repairs
$M(x)$:	x has a menial job
$R(x)$:	x repairs his own car
$H(x)$:	x is highly skilled.

Therefore, (i), (ii) and (iii) can be symbolized as:

- (i) $(\forall x) (R(x) \rightarrow (H(x) \wedge S(x)))$
- (ii) $\exists (x) (R(x) \wedge M(x))$
- (iii) $(\exists x) (M(x) \wedge H(x))$ (to be concluded)

From (ii) using Existential Instantiation (E.I), we get, for some fixed a

$$(iv) \quad R(a) \wedge M(a)$$

Then by simplification rule of Propositional Logic, we get

$$(v) \quad R(a)$$

From (i), using Universal Instantiation (U.I.), we get

$$(vi) \quad R(a) \rightarrow H(a) \wedge S(a)$$

Using modus ponens w.r.t. (v) and (vi) we get

$$(vii) \quad H(a) \wedge S(a)$$

By specialisation of (vii) we get

$$(viii) \quad H(a)$$

By specialisation of (iv) we get

$$(ix) \quad M(a)$$

By conjunctions of (viii) & (ix) we get

$$M(a) \wedge H(a)$$

By Existential Generalisation, we get

$$(\exists x) (M(x) \wedge H(x))$$

Hence, (iii) is concluded.

Example:

- (i) Some juveniles who commit minor offences are thrown into prison, and any juvenile thrown into prison is exposed to all sorts of hardened criminals.
- (ii) A juvenile who is exposed to all sorts of hardened criminals will become bitter and learn more techniques for committing crimes.
- (iii) Any individual who learns more techniques for committing crimes is a menace to society, if he is bitter.
- (iv) Therefore, some juveniles who commit minor offences will be menaces to the society.

Example: Let us symbolize the statement in the given argument as follows:

- (i) $J(x)$: x is juvenile.
- (ii) $C(x)$: x commits minor offences.
- (iii) $P(x)$: x is thrown into prison.
- (iv) $E(x)$: x is exposed to hardened criminals.
- (v) $B(x)$: x becomes bitter.
- (vi) $T(x)$: x learns more techniques for committing crimes.
- (vii) $M(x)$: x is a menace to society.

The statements of the argument may be translated as:

- (i) $(\exists x) (J(x) \wedge C(x) \wedge P(x)) \wedge ((\forall y) (J(y) \rightarrow E(y)))$
 - (ii) $(\forall x) (J(x) \wedge E(x) \rightarrow B(x) \wedge T(x))$
 - (iii) $(\forall x) (T(x) \wedge B(x) \rightarrow M(x))$
- Therefore,
- (iv) $(\exists x) (J(x) \wedge C(x) \wedge M(x))$

By simplification (i) becomes

- (v) $(\exists x) (J(x) \wedge C(x) \wedge P(x))$ and
- (vi) $(\forall y) (J(y) \rightarrow E(y))$

From (v) through Existential Instantiation, for some fixed b , we get

$$(vii) \quad J(b) \wedge C(b) \wedge P(b)$$

Through simplification (vii) becomes

$$(viii) \quad J(b)$$

- (ix) $C(b)$ and
(x) $P(b)$

Using Universal Instantiation, on (vi), we get

- (xi) $J(b) \rightarrow E(b)$

Using Modus Ponens in (vii) and (xi) we get

- (xii) $E(b)$

Using conjunction for (viii) & (xii) we get

- (xiii) $J(b) \wedge E(b)$

Using Universal Instantiation on (ii) we get

- (xiv) $J(b) \wedge E(b) \rightarrow B(b) \wedge T(b)$

Using Modus Ponens for (xiii) & (xiv), we get

- (xv) $T(b) \wedge B(b)$

Using Universal Instantiation for (iii) we get

- (xvi) $T(b) \wedge B(b) \rightarrow M(b)$

Using Modus Ponens with (xv) and (xvi) we get

- (xvii) $M(b)$

Using conjunction for (viii), (ix) and (xvii) we get

- (xviii) $J(b) \wedge C(b) \wedge M(b)$

From (xviii), through Existential Generalization we get the required (iv), i.e.

$$(\exists x) (J(x) \wedge C(x) \wedge M(x))$$

Remark: It may be noted the occurrence of quantifiers is not, in general, commutative i.e.,

$$(Q_1x) (Q_2x) \neq (Q_2x) (Q_1x)$$

For example

$$(\forall x) (\exists y) F(x,y) \neq (\exists y) (\forall x) F(x,y) \quad (A)$$

The occurrence of $(\exists y)$ on L.H.S depends on x i.e., occurrence of y on L.H.S is a function of x . However, the occurrence of $(\exists y)$ on R.H.S is independent of x , hence, occurrence of y on R.H.S is not a function of x .

For example, if we take $F(x,y)$ to mean:

y and x are integers such that $y > x$,

then, L.H.S of (A) above states: *For each x there is a y such that $y > x$.*

The statement is true in the domain of real numbers.

On the other hand, R.H.S of (A) above states that: There is an integer y which is greater than x , for all x .

This statement is not true in the domain of real numbers.

Ex. 4 We are given the statements:

- (i) *No feeling of pain is publically observable*
(ii) *All chemical processes are publically observable*

We are to prove that

-
- (iii) *No feeling of pain is a chemical process.*
-

2.4 RESOLUTION METHOD IN PL

Basically, there are two different **approaches** for proving a theorem or for making a valid deduction from a given set of axioms:

- i) natural deduction
- ii) refutation method

In the **natural deduction** approach, one starts with a the set of axioms, uses some rules of inference and arrives at a conclusion. This approach closely resembles of the intuitive reasoning of human beings.

On the other hand, in a **refutation method**, one *starts with the negation of the conclusion to be drawn* and derives a contradiction or FALSE. Because of having assumed the conclusion as false, we derive a contradiction; **therefore, the assumption that the conclusion is wrong, itself is wrong**. Hence, the argument of resolution method leads to the validity of the conclusion.

So far, we have discussed methods, of solving problems requiring reasoning of propositional logic, that were based on

- i) Truth-table construction
- ii) Use of inference rules,
and follow, directly or indirectly, **natural deduction approach**.

In this section, we discuss another method, viz., **Resolution Method** suggested by **Robinson** in 1965 which is based on **refutation approach**. The method is important in view of the fact that the Robinson's method has been a basis for some automated theorem provers. Even, the logic programming language PROLOG (*subject matter of Unit 2, Block 3*) is based on Resolution Method. The resolution method, as mentioned above, is a **refutation method**.

In this section, we discuss how the resolution method is applied in solving problems using only Propositional Logic (PL). The general resolution method for FOPL is discussed in the next section.

The resolution method in PL is applied only after converting the given statements or wffs into clausal forms. A **clausal form** of a wff is obtained by *first* converting the wff into its equivalent Conjunctive Normal Form (CNF). We already know that a *clause* is a formula (only) of the form:

$$A_1 \vee A_2 \vee \dots \vee A_n,$$

where A_i is either an atomic formula or negation of an atomic formula.

The resolution method is a generalization of the Modus Ponens, i.e., of

$$\frac{P, P \rightarrow Q}{Q} \text{ when written in the equivalent form } \frac{P, \sim P \vee Q}{Q}$$

(replacing $P \rightarrow Q$ by $\sim P \vee Q$).

This simple special case, of **general resolution principle** to be discussed soon, states that if the two formulas P and $\sim P \vee Q$ are given to be True, then we can assume Q to be True.

The validity of (general) resolution method can be established by constructing truth-table.

In order to discuss the resolution method, first we discuss some of its applications.

Example: Let $C_1 : Q \vee R$ and $C_2 : \sim Q \vee S$ be two given clauses, so that, one of the literals i.e., Q occurs in one of the clauses (in this case C_1) and its negation ($\sim Q$) occurs in the other clause C_2 . Then application of resolution method in this case tells us to take disjunction of the remaining parts of the given clause C_1 and C_2 , i.e., to take $C_3 : R \vee S$ as **deduction** from C_1 and C_2 . Then C_3 is called a **resolvent** of C_1 and C_2 . The two literals Q and ($\sim Q$) which occur in two different clauses are called **complementary literals**.

In order to illustrate resolution method, we consider another example.

Example: Let us be given the clauses $C_1 : \sim S \vee \sim Q \vee R$ and $C_2 : \sim P \vee Q$.

In this case, complementary pair of literals viz. Q and $\sim Q$ occur in the two clause C_1 and C_2 .

Hence, the resolution method states:

Conclude $C_3 : \sim S \vee R \vee (\sim P)$

Example: Let us be given the clauses $C_1 : \sim Q \vee R$ and $C_2 : \sim Q \vee S$. Then, in this case, the clauses do not have any complementary pair of literals and hence, resolution method cannot be applied.

Example: Consider a set of three clauses

$C_1 : R$

$C_2 : \sim R \vee S$

$C_3 : \sim S$

Then, from C_1 and C_2 we conclude, through resolution:

$C_4 : S$

From C_3 and C_4 , we conclude,

$C_5 : \text{FALSE}$

However, a resolvent FALSE can be deduced only from an **unsatisfiable set of clauses**. Hence, the set of clauses C_1 , C_2 and C_3 is an unsatisfiable set of clauses.

Example: Consider the set of clauses

$C_1 : R \vee S$

$C_2 : \sim R \vee S$

$C_3 : R \vee \sim S$

$C_4 : \sim R \vee \sim S$

Then, from clauses C_1 and C_2 we get the resolvent

$C_5 : S \vee S = S$

From C_3 and C_4 we get the resolvent

$C_6 : \sim S$

From C_5 and C_6 we get the resolvent

$C_7 : \text{FALSE}$

Thus, again the set of clauses C_1 , C_2 , C_3 and C_4 is unsatisfiable.

Note: We could have obtained the resolvent FALSE from only two clauses, viz., C_2 and C_3 . Thus, out of the given four clauses, even set of only two clauses viz, C_2 and C_3 is unsatisfiable. Also, a superset of any unsatisfiable set is unsatisfiable.

Example: Show that the set of clauses:

$C_1: R \vee S$

$C_2: \sim S \vee W$

$C_3: \sim R \vee S$

$C_4: \sim W$ is unsatisfiable.

From clauses C_1 and C_3 we get the resolvent

$C_7: S$

From the clauses C_7 and C_2 we get the resolvent

$C_8: W$

From the clauses C_8 and C_4 we get

$C_9: \text{FALSE}$

Hence, the given set of clauses is unsatisfiable.

Problem Solving using resolution method. We have mentioned earlier, the resolution method is a refutation method. Therefore, proof technique in solving problems will be as follows:

After symbolizing the problem under consideration, add the negation of the wff which represents conclusion, as an additional premise. From this enhanced set of premises/axioms, derive FALSE or contradiction. If we are able to conclude FALSE, then the conclusion, that was required to be drawn, is valid and problem is solved.

However, through all efforts, if we are **not able to derive FALSE**, then we **cannot say** whether the conclusion is valid or invalid. Hence, the **problem** with given axioms and the conclusion **is not solvable**.

Let us now apply Resolution Method for the problems considered earlier.

Example: Suppose the stock prices go down if the interest rate goes up. Suppose also that the most people are unhappy when stock prices go down. Assume that the interest rate goes up. Show that we can conclude that most people are unhappy.

To show the above conclusion, let us denote the statements as follows:

A : Interest rate goes up,

S : Stock prices go down

U : Most people are unhappy

The problem has the following four statements

- 1) If the interest rate goes up, stock prices go down.
- 2) If stock prices go down, most people are unhappy.
- 3) The interest rate goes up.
- 4) Most people are unhappy. (to conclude)

These statements are first symbolized as wffs of PL as follows:

(1') $A \rightarrow S$

(2') $S \rightarrow U$

(3') A

(4') U. (to conclude)

Converting to clausal form, we get

(i) $\sim A \vee S$

(ii) $\sim S \vee U$

(iii) A

(iv) U (to be concluded)

As per resolution method, assume (iv) as false, i.e., *assume $\sim U$ as initially given statement, i.e., an axiom.*

Thus, the set of axioms in clausal form is:

- (i) $\sim A \vee S$
- (ii) $\sim S \vee U$
- (iii) A
- (iv) $\sim U$

Then from (i) and (iii), through resolution, we get the clause
(v) S .

From (ii) and (iv), through resolution, we get the clause

(vi) $\sim S$

From (vi) and (v), through resolution we get,

(viii) FALSE

Hence, the conclusion, i.e.,

(iv) U : *Most people are unhappy*

is valid.

We might have observed from the above solution using resolution method, that clausal conversion is a major time-consuming step after translation to wffs. Generally, once the clausal form is obtained, proof, at least, by a human being can be easily visualised.

Ex. 5: Given that if the Parliament refuses to enact new laws, then the strike will not be over unless it lasts more than one year and the president of the firm resigns, will the strike not be over if the Parliament refuses to act and the strike just starts?

2.5 RESOLUTION METHOD IN FOPL

In the beginning of the previous section, we mentioned that resolution method for FOPL requires discussion of a number of complex new concepts. Also, in Block 2, we discussed (Skolem) Standard Form and also discussed how to obtain Standard Form for a given formula of FOPL. In this section, we introduce two new, and again complex, concepts, viz., *substitution and unification*.

The complexity of the resolution method for FOPL mainly results from the fact that a clause in FOPL is generally of the form : $P(x) \vee Q(f(x), x, y) \vee \dots$, in which the variables x, y, z , may assume any one of the values of their domain.

Thus, the atomic formula $(\forall x) P(x)$, *which after dropping of universal quantifier, is written as just $P(x)$* stands for $P(a_1) \wedge P(a_2) \dots \wedge P(a_n)$ where the set $\{a_1, a_2, \dots, a_n\}$ is assumed here to be domain (x) .

Similarly, $(\exists x) P(x)$ stands for $(P(a_1) \vee P(a_2) \vee \dots \vee P(a_n))$

However, in order to resolve two clauses – one containing say $P(x)$ and the other containing $\sim P(y)$ where x and y are universal quantifiers, possibly having some restrictions, we have to know which values of x and y satisfy both the clauses. For this purpose we need the concepts of **substitution** and **unification** as defined and discussed in the rest of the section.

Instead of giving formal definitions of substitution, unification, unifier, most general unifier and resolvent, resolution of clauses in FOPL, we illustrate the concepts through examples and minimal definitions, if required

Example: Let us consider our old problem:

To conclude

(i) Raman is mortal

From the following two statements:

(ii) Every man is mortal and

(iii) Raman is a man

Using the notations

MAN (x) : x is a man

MORTAL (x) : x is mortal,

the problem can be formulated in symbolic logic as: Conclude

MORTAL (Raman)

from

(ii) $((\forall x) (MAN(x) \rightarrow MORTAL (x)))$

(iii) MAN (Raman).

As resolution is a refutation method, assume

(i) $\sim MORTAL (Raman)$

After Skolemization and dropping $(\forall x)$, (ii) in standard form becomes

(i) $\sim MAN (x) \vee MORTAL (x)$

(ii) MAN (Raman)

In the above x varies over the set of human beings including Raman. Hence, one special instance of (iv) becomes

(vi) $\sim MAN (Raman) \vee MORTAL (Raman)$

At the stage, we may observe that

(a) $MAN(Raman)$ and $MORTAL(Raman)$ do not contain any variables, and, hence, their truth or falsity can be determined directly. Hence, each of like a formula of PL. In term of formula which does not contain any variable is called **ground term** or **ground formula**.

(b) Treating $MAN (Raman)$ as formula of PL and using resolution method on (v) and (vi), we conclude

(vii) MORTAL (Raman),

Resolving (i) and (vii), we get **False**. Hence, the solution.

Unification: In the process of solution of the problem discussed above, we tried to make the two expression $MAN(x)$ and $MAN(Raman)$ identical. Attempt to make identical two or more expressions is called *unification*.

In order to unify $MAN(x)$ and $MAN(Raman)$ identical, we found that because one of the possible values of x is $Raman$ also. And, hence, we replaced x by one of its possible values : $Raman$.

This replacement of a variable like x , by a term (*which may be another variable also*) which is one of the possible values of x , is called **substitution**. The substitution, in this case is denoted formally as $\{Raman/x\}$

Substitution, in general, **notationally** is of the form $\{t_1 / x_1, t_2 / x_2 \dots t_m / x_m\}$ where $x_1, x_2 \dots, x_m$ are variables and $t_1, t_2 \dots t_m$ are terms and t_i replaces the variable x_i in some expression.

Example: (i) Assume Lord Krishna is loved by everyone who loves someone (ii) Also assume that no one loves nobody. Deduce Lord Krishna is loved by everyone.

Solution: Let us use the symbols

Love (x, y): x loves y (or y is loved by x)

LK : Lord Krishna

Then the given problem is formalized as :

(i) $(\forall x) ((\exists y) \text{Love}(x, y) \rightarrow \text{Love}(x, \text{LK}))$

(ii) $\sim (\exists x) ((\forall y) \sim \text{Love}(x, y))$

To show : $(\forall x) (\text{Love}(x, \text{LK}))$

As resolution is a refutation method, assume negation of the last statement as an axiom.

(iii) $\sim (\forall x) \text{Love}(x, \text{LK})$

The formula in (i) above is reduced in standard form as follows:

$(\forall x) (\sim (\exists y) \text{Love}(x, y) \vee \text{Love}(x, \text{LK}))$

$= (\forall x) ((\forall y) \sim \text{Love}(x, y) \vee \text{Love}(x, \text{LK}))$

$= (\forall x) (\forall y) (\sim \text{Love}(x, y) \vee \text{Love}(x, \text{LK}))$

($\because (\forall y)$ does not occurs in $\text{Love}(x, \text{LK})$)

After dropping universal quantifications, we get

(iv) $\sim \text{Love}(x, y) \vee \text{Love}(x, \text{LK})$

Formula (ii) can be reduced to standard form as follows:

(ii) $= (\forall x) (\exists y) \text{Love}(x, y)$

y is replaced through skolemization by $f(x)$

so that we get

$(\forall x) \text{Love}(x, f(x))$

(v) $\text{Love}(x, f(x))$

The formula in (iii) can be brought in standard form as follows:

(iii) $= (\exists x) (\sim \text{Love}(x, LK))$

As existential quantifier x is not preceded by any universal quantification, therefore, x may be substituted by a constant a , i.e., we use the substitution $\{a/x\}$ in (iii) to get the standard form:

(vi) $\sim \text{Love}(a, LK)$.

Thus, to solve the problem, we have the following standard form formulas for resolution:

(iv) $\sim \text{Love}(x, y) \vee \text{Love}(x, LK)$

(v) $\text{Love}(x, f(x))$

(vi) $\sim \text{Love}(a, LK)$.

Two possibilities of resolution exist for two pairs of formulas viz.

one possibility: resolving (v) and (vi).

second possibility : resolving (iv) and (vi).

The possibilities exist because for each possibility pair, the predicate *Love* occurs in complemented form in the respective pair.

Next we attempt to resolve (v) and (vi)

For this purpose we attempt to make the two formulas $\text{Love}(x, f(x))$ and $\text{Love}(a, LK)$ identical, through unification involving substitutions. We start from the left, matching the two formulas, term by term. First place where matching may fail is when 'x' occurs in one formula and 'a' occurs in the other formula. **As, one of these happens to be a variable**, hence, the substitution $\{a/x\}$ can be used to unify the portions so far.

Next, possible disagreement through term-by-term matching is obtained when we get the two disagreeing terms from two formulas as $f(x)$ and LK . **As none of $f(x)$ and LK is a variable** (note $f(x)$ involves a variable but is itself not a variable), hence, no unification and, hence, no resolution of (v) and (vi) is possible.

Next, we attempt unification of **(vi) $\text{Love}(a, LK)$ with $\text{Love}(x, LK)$** of (iv).

Then first term-by-term possible disagreement occurs when the corresponding terms are 'a' and 'x' respectively. As one of these is a variable, hence, the substitution $\{a/x\}$ unifies the parts of the formulas so far. Next, the two occurrences of LK , one each in the two formulas, match. Hence, the whole of each of the two formulas can be unified through the substitution $\{a/x\}$. Though the unification has been *attempted* in corresponding smaller parts, substitution has to be carried **in the whole of the formula**, in this case in whole of (iv). Thus, after substitution, (iv) becomes

(viii) $\sim \text{Love}(a, y) \vee \text{Love}(a, LK)$

resolving (viii) with (vi) we get

(ix) $\sim \text{Love}(a, y)$

In order to resolve (v) and (ix), we attempt to unify **Love (x, f(x))** of (v) with **Love (a, y)** of (ix). The term-by-term matching leads to possible disagreement of *a* of (ix) with *x* of (v). As, one of these is a variable, hence, the substitution $\{a/x\}$ will unify the portions considered so far. Next, possible disagreement may occur with *f(x)* of (v) and *y* of (ix). As one of these are a variable viz. *y*, therefore, we can unify the two terms through the substitution $\{f(x)/y\}$. Thus, the complete substitution $\{a/x, f(x)/y\}$ is required to match the formulas.

Making the substitutions, we get

(v) becomes Love (a, f(x))

and (ix) becomes \sim Love (a, f(x))

Resolving these formulas we get **False**. Hence, the proof.

Ex. 6: Unify, if possible, the following three formulas:

- (i) $Q(u, f(y, z))$,
- (ii) $Q(u, a)$
- (iii) $Q(u, g(h(k(u))))$

Ex. 7: Determine whether the following formulas are unifiable or not:

- (i) $Q(f(a), g(x))$
- (ii) $Q(x, y)$

Example: Find resolvents, if possible for the following pairs of clauses:

- (i) $\sim Q(x, z, x) \vee Q(w, z, w)$ and
- (ii) $Q(w, h(v, v), w)$

Solution: As two literals with predicate *Q* occur and are mutually negated in (i) and (ii), therefore, there is possibility of resolution of $\sim Q(x, z, x)$ from (i) with $Q(w, h(v, v), w)$ of (ii). We attempt to unify $Q(x, z, x)$ and $Q(w, h(v, v), w)$, if possible, by finding an appropriate substitution. First terms *x* and *w* of the two are variables, hence, unifiable with either of the substitutions $\{x/w\}$ or $\{w/x\}$. Let us take $\{w/x\}$. Next pair of terms from the two formulas, viz. *z* and *h(v, v)* are also unifiable, because, one of the terms is a variable, and the required substitution for unification is $\{h(v, v)/z\}$.

Next pair of terms at corresponding positions is again $\{w, x\}$ for which, we have determined the substitution $\{w/x\}$. Thus, the substitution $\{w/x, h(v, v)/z\}$ unifies the two formulas. Using the substitutions, (i) and (ii) become resp. as

- (iii) $\sim Q(w, h(v, v), w) \vee Q(w, h(v, v), w)$
- (iv) $Q(w, h(v, v), w)$

Resolving, we get

$Q(w, h(v, v), w)$,

which is the required resolvent.

2.6 SUMMARY

In this unit, eight basic rules of inference for PL and four rules involving quantifiers for inferencing in FOPL, are introduced respectively in Section 3.2 and Section 3.3, and then these rules are used in solving problems. Further, a new method of drawing inference called Resolution method based on refutation approach, is discussed in the next two Sections. In Section 3.4, Resolution method for PL is introduced and applied in solving problems involving PL reasoning. In Section 3.5, Resolution method for FOPL is introduced and used for solving problems involving FOPL reasoning.

FOPL is not capable of easily representing some kinds of information including information pieces involving.

(i) Properties of relations. For example, the mathematical statement:

Any relation which is symmetric and transitive may not be reflexive is not expressible in FOPL. A relation in FOPL can only be constant, and not a variable. Only in second and higher order logics, the relations may be variable. This type of logics are not within the scope of the course.

(ii) linguistic variable like hot, tall, sweat.

For example: *It is very cold today*,
can not be appropriately expressed in FOPL.

(iii) different belief systems.

For example, *I know that he thinks India will win the match, but I think India will lose*, also, cannot be appropriately expressed in FOPL.

2.7 SOLUTIONS/ANSWERS

Ex.1: Assuming the statements (i), (ii) and (iii) given above as True we are required to Show the truth of (iv)

The **first** step is to mark the logical operators, if any, in the statements of the argument/problem under consideration.

In the above-mentioned problem, statement (i) does not contain any logical operator. Each of the statements (ii) and (iii) contains the logical operator 'If....then....'

The next step is to use symbols, P, Q, R, for atomic formulas occurring in the problem. The symbols are generally mnemonic, i.e., names used to help memory.

Let us denote the atomic statements in the argument given above as follows:

M: Matter always existed,

TG: There is God,

GU: God created the universe.

Then **the given** statements in English, become respectively the following *formulas* of PL:

- (i) M
- (ii) $TG \rightarrow GU$
- (iii) $GU \rightarrow \sim M$
- (iv) $\sim TG$ (*To show*)

Applying transposition to (iii) we get

- (v) $M \rightarrow \sim GU$

using (i) and (v) and applying Modus Ponens, we get

- (vi) $\sim GU$

Again, applying transposition to (ii) we get

(vii) $\sim GU \rightarrow \sim TG$

Applying Modus Ponens to (vi) and (vii) we get

(vii) $\sim TG$

The formula (viii) is the same as formula (iv) which was required to be proved.

Ex.2 In order to translate in PL, let us use the symbols:

ML: there is a moral law,

SG: someone gave it, (the word 'it' stand for moral law)

TG: There is God.

Using these symbols, the **Statement** (i) to (iv) become the **formula** (i) to (iv) of PL as given below:

(i) ML

(ii) $ML \rightarrow SG$

(iii) $SG \rightarrow TG$ and

(iv) TG

Applying Modus Ponens to formulae (i) and (ii) we get the formula

(v) SG

Applying Modus Ponens to (v) and (iii), we get

(vi) TG

But formula (vi) is the same as (iv), which is required to be established. Hence the proof.

Ex. 3: (i) Concluding $F(a) \wedge G(a)$ from $(\exists x)F(x) \wedge (\exists x)G(x)$ is *incorrect*, because, as mentioned earlier also, the given Quantified Formula may be equivalently written as $(\exists x)F(x) \wedge (\exists y)G(y)$. And in the case of each existential quantification, we can not assign an already-used constant. Therefore, a correct conclusion may be of the form

$F(a) \wedge G(b)$

(ii) The conclusion of $F(a) \vee (G(a) \wedge H(a))$

from $(\exists x)F(x) \vee (G(x) \wedge H(x))$

is again *incorrect*, in view of the fact that scope of existential variable in the formula, is only $F(x)$ and not the whole formula. Hence, the last two occurrences of x are free. Therefore, a correct conclusion can be $F(a) \vee (G(x) \wedge H(x))$

(iii) The conclusion is correct

(iv) The conclusion is *incorrect*, because, from the given fact $\sim (F(a) \wedge G(a))$,

we may conclude $((\exists x) (\sim (F(x) \wedge G(x)))$

which is equivalent to $\sim (\forall x) (F(x) \wedge G(x))$

and not to $\sim (\exists x) (F(x) \wedge G(x))$

Ex. 4: For translating the given statements (i), (ii) & (iii), let us use the notation:

$F(x)$: x is an instance of feeling of pain

$O(x)$: x is an entity that is publically observable

$C(x)$: x is a chemical process.

Then, the statement (i), (ii) and (iii) can be equivalently expressed as formulas of FOPL

(i) $(\forall x) (F(x) \rightarrow \sim O(x))$

(ii) $(\forall x) (C(x) \rightarrow O(x))$

To prove

(iii) $(\forall x) (F(x) \rightarrow \sim C(x))$

From (i) using generalized instantiation, we get

(iv) $F(a) \rightarrow \sim O(a)$, for any arbitrary a

Similarly, from (ii), using generalized instantiation, we get

(v) $C(b) \rightarrow O(b)$, for arbitrary b

From (iv) using transposition rule, we get

(vi) $O(a) \rightarrow \sim F(a)$, for arbitrary a

As b is arbitrary in (v), therefore we can rewrite (v) as

(vii) $C(a) \rightarrow O(a)$, for arbitrary a

From (vii) and (vi) and using chain rule, we get

(viii) $C(a) \rightarrow \sim F(a)$, for any arbitrary a

But as a is arbitrary in (viii), by generalized quantification, we get

(ix) $(\forall x) (C(x) \rightarrow \sim F(x))$

But (ix) is the same as (iii), which was required to be proved.

Ex. 5: Let us symbolize the statements in the problem given above as follows:

A: The Parliament refuses to act.

B: The strike is over.

R: The president of the firm resigns.

S: The strike lasts more than one year.

Then the facts and the question to be answered can be symbolized as:

E1: $(A \rightarrow (\sim B \vee (R \wedge S)))$ represents the statement: If the congress refuses to enact new laws, then the strike will not be over unless it lasts more than one year and the president of the firm resigns.

(Note: Punless $Q = P \vee Q$)

E2 : A represents the statement: The congress refuses to act, and

E3: $\sim S$ represent the statement: The strike just starts.

E4: $\sim B$ (to be concluded)

As we are going to use resolution method, we use **E₅ the negation of E₄** as an axiom in addition to E₁, E₂ and E₃.

E₅: B

As a first step, we convert E₁, E₂, E₃ and E₅ into clausal forms as follows:

E₁: $\sim A \vee (\sim B \vee (R \wedge S))$

Using associativity of \vee , we get

$= (\sim A \vee \sim B) \vee (R \wedge S)$

Using distributivity of \vee over \wedge , we get

$= (\sim A \vee \sim B \vee R) \wedge (\sim A \vee \sim B \vee S)$

replacing E₁ by two clauses

E₁₁: $(\sim A \vee \sim B \vee R)$ and

E₁₂: $(\sim A \vee \sim B \vee S)$

E₂, E₃ and E₅ are already in clausal form

We get the axioms, including the negation of the conclusion, in the clausal form as

E₁₁: $(\sim A \vee \sim B \vee R)$

E₁₂: $(\sim A \vee \sim B \vee S)$

E₂: A

E₃: $\sim S$

E₅: $\sim (\sim B) = B$

By resolving E₂ with E₁₂, we get the resolvent

E₆: $\sim B \vee S$

By resolving E₅ with E₆, we get the resolvent as

E₇: S

By resolving E₇ with E₃, we get the resolvent as

E₈: FALSE

Hence, the conclusion $\sim B$: The strike will not be over, is valid.

Ex. 6: First, we attempt to unify (i) and (ii)

As the predicate is Q in each of the given terms, therefore, we should attempt matching terms. The first terms match, as each is u. Next second terms are 'a' and f(y, z), none of which is a variable. Hence, (i) and (ii) are not unifiable.

In the similar manner (i) and (iii) are not unifiable as the second terms f(y, z) and g(h(k(u))) are such that none is a variable.

Ex. 7: The predicate symbols (*each being Q*) match. Hence, we may proceed. Next, the first two terms viz. f(a) and x, are not identical. However, as one of these terms is a variable viz. 'x', hence, the corresponding terms are unifiable with substitution {f(a)/x}.

Next, the two terms g(x) and y, one from each of the formula at corresponding positions, are again unifiable by the substitution {g(x)/y}.

Hence, the required substitutions {f(a)/x, g(f(a))/y} using the substitution {f(a)/x} in g(x)/y to get the substitution {g(f(a))/y}.

Therefore the two formulas are unifiable and after unification the formulas become Q(f(a), g(f(a)))

2.8 FURTHER READINGS

1. McKay, Thomas J., *Modern Formal Logic* (Macmillan Publishing Company, 1989).
2. Gensler, Harry J. *Symbolic Logic: Classical and Advanced Systems* (Prentice Hall, 1990).
3. Klenk, Virginia *Understanding Symbolic Logic* (Prentice Hall 1983)
4. Mendelson, Elliott: *Introduction to Mathematical Logic (Second Edition)* (D. Van Nostrand Company, 1979).
5. Copi Irving M. & Cohen Carl, *Introduction Logic, IX edition*, (Prentice Hall of India, 2001).
6. Stuart Russell, Peter Norving *Artificial Intelligence (Second Edition)* (Pearson Education 2003).

UNIT 3 SYSTEMS FOR IMPRECISE/INCOMPLETE KNOWLEDGE

Structure	Page Nos.
3.0 Introduction	50
3.1 Objectives	51
3.2 Fuzzy Systems	51
3.3 Relations on Fuzzy Sets	55
3.4 Operations on Fuzzy Sets	57
3.5 Operations Unique to Fuzzy Sets	59
3.6 Non-Monotonic Reasoning Systems	62
3.7 Default Reasoning Systems	64
3.8 Closed World Assumption Systems	65
3.9 Other Non-Deductive Systems	66
3.10 Summary	67
3.11 Solutions/ Answers	67
3.12 Further Readings	68

3.0 INTRODUCTION

In the earlier three units of the block, we discussed PL and FOPL systems for making inferences and solving problems requiring logical reasoning. However, these systems assume that the domain of the problems under consideration is complete, precise and consistent. But, in the real world, the knowledge of the problem domains is generally neither precise nor consistent and is hardly complete.

In this unit, we discuss a number of techniques and formal systems that attempt to handle some of these blemishes. To begin with, in Sections 4.2 to 4.5, we discuss **fuzzy systems** that attempt to handle **imprecision** in knowledge bases, specially, due to use of natural language words like hot, good, tall etc.

Then, we discuss **non-monotonic systems** which deal with **indefiniteness** of knowledge in the knowledge bases. The significance of these systems lies in the fact that most of the statements in the knowledge bases are actually based on **beliefs** of the concerned persons or actors. These beliefs get revised as better evidence for some other beliefs become available, where the later beliefs may be in conflict with the earlier beliefs. In such cases, the earlier beliefs may have to be temporarily suspended or permanently excluded from further considerations.

In Sections 4.7 and 4.8, we discuss two formal systems that attempt to handle **incompleteness** of the available information. These systems are called **Default Reasoning Systems** and **Closed World Assumption Systems**. Finally, we discuss some inference rules, viz, **abductive** inference rule and **inductive** inference rule that are, though not deductive, yet are quite useful in solving problems arising out of everyday experience.

3.1 OBJECTIVES

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

- enumerate various formal methods, which deal with different types of blemishes like incompleteness, imprecision and inconsistency in a knowledge base;
- discuss, why fuzzy systems are required;
- discuss, develop and use fuzzy arithmetic tools in solving problems, the descriptions of which involve imprecision;
- discuss default reasoning as a tool for handling incompleteness of knowledge;
- discuss Closed World Assumption System, as another tool for handling incompleteness of knowledge, and
- discuss and use non-deductive inference rules like abduction and induction, as tools for solving problems from everyday experience.

3.2 FUZZY SYSTEMS

In the symbolic Logic systems like, PL and FOPL, that we have studied so far, any (closed) formula has a truth-value which must be binary, viz., *True or False*. However, in our everyday experience, we encounter problems, the descriptions of which involve some words, because of which, to statements of situations, it is not possible to assign a truth value: *True or False*. For example, consider the statement: *If the water is too hot, add normal water to make it comfortable for taking a bath.*

In the above statement, for a number of words/phrases including ‘too hot’ ‘add’, ‘comfortable’ etc., it is not possible to tell when exactly water is too hot, when water is (at) normal (temperature), when exactly water is comfortable for taking a bath.

For example, we cannot tell the temperature T such that for water at temperature T or less, truth value False can be associated with the statement ‘*Water is too hot*’ and at the same time truth-value True can also be associated to the same statement ‘*Water is too hot*’ when the temperature of the water is, say, at degree $T + 1$, $T + 2$etc.

Some other cases of Fuzziness in a Natural Language

Healthy Person: we cannot even enumerate all the parameters that **determine** health. Further, it is even more difficult to tell for what value of a particular parameter, one is healthy or otherwise.

Old/young person: It is not possible to tell exactly upto exactly what age, one is young and, by just addition of one day to the age, one becomes old. We age gradually. Aging is a **continuous** process.

Sweet Milk: Add small sugar cube one at a time to glass of milk, and go on adding upto, say, 100 small cubes.

Initially, without sugar, we may take milk as not sweet. However, with addition of each one small sugar particle cube, the sweetness **gradually** increases. It is not possible to say that after addition of 100 small cubes of sugar, the milk becomes sweet, and, till addition of 99 small cubes, it was not sweet.

Pool, Pond, Lake,....., Sea, Ocean: for different sized water bodies, we can not say when exactly a pool becomes a pond, when exactly a pond becomes a lake and so on.

One of the reasons, for this type of problem of our inability to associate one of the two-truth values to statements describing everyday situations, is due to the use of natural language words like hot, good, beautiful etc. Each of these words does not denote something constant, but is a sort of linguistic variable. The context of a particular *usage* of such a word may delimit the scope of the word as a linguistic variable. The range of values, in some cases, for some phrases or words, may be very large as can be seen through the following three statements:

- Dinosaurs ruled the earth for a long period (*about millions of years*)
- It has not rained *for a long period* (*say about six months*).
- I had to wait for the doctor *for a long period* (*about six hours*).

Fuzzy theory provides means to handle such situations. A **Fuzzy theory** may be thought as a technique of providing ‘**continuization**’ to the otherwise binary disciplines like Set Theory, PL and FOPL.

Further, we explain how using fuzzy concepts and rules, in situation like the ones quoted below, we, the human beings solve problems, despite ambiguity in language.

Let us recall the case of crossing a road discussed in Unit 1 of Block 1. We mentioned that a step by step method of crossing a road may consist of

- (i) Knowing (exactly) the distances of various vehicles from the path to be followed to cross over.
- (ii) Knowing the velocities and accelerations of the various vehicles moving on the road within a distance of, say, one kilometer.
- (iii) Using Newton’s Laws of motion and their derivatives like $s = ut + \frac{1}{2}at^2$, and calculating the time that would be taken by each of the various vehicles to reach the path intended to be followed to cross over.
- (iv) Adjusting dynamically our speeds on the path so that no collision takes place with any of the vehicle moving on the road.

But, we know the human beings not only do not follow the above precise method but cannot follow the above precise method. **We, the human beings rather feel comfortable with fuzziness than precision.** We feel comfortable, if the instruction for crossing a road is given as follows:

Look on both your *left hand* and *right hand* sides, particularly in the beginning, to your right hand side. If there is no vehicle within *reasonable* distance, then attempt to cross the road. You may have to retreat back while crossing, from *somewhere* on the road. Then, try again.

The above instruction has a number of words like *left*, *right* (it may 45° to the right or 90° to the right) *reasonable*, each of which does not have a definite meaning. But we feel more comfortable than the earlier instruction involving precise terms.

Let us consider another example of *our being comfortable with imprecision than precision*. The statement: ‘*The sky is densely clouded*’ is more comprehensible to human beings than the statement: ‘*The cloud cover of the sky is 93.5 %*’.

Thus is because of the fact that, we, the human beings are still better than computers in **qualitative** reasoning. Because of better qualitative reasoning capabilities

- just by looking at the eyes only and/or nose only, we may recognize a person.

- just by taking and feeling a small number of grains from cooking rice bowl, we can **tell** whether the rice is properly cooked or not.
- just by looking at few buildings, we can identify a locality or a city.

Achieving Human Capability

In order that computers achieve human capability in solving such problems, computers must be able to solve problems for which *only incomplete and/or imprecise* information/knowledge is available.

Modelling of Solutions and Data/Information/Knowledge

We know that for any problem, the plan of the proposed solution and the relevant information is fed in the computer in a form acceptable to the computer.

However, the problems to be solved with the help of computers are, in the first place, felt by the human beings. And then, the plan of the solution is also prepared by human beings.

It is conveyed to the computer mainly for execution, because computers have much better executional speed.

Summarizing the discussion, we conclude the following facts

- (i) We, the human beings, sense problems, desire the problems to be solved and express the problems and the plan of a solution using imprecise words of a natural language.
- (ii) We use computers to solve the problems, because of their executional power.
- (iii) Computers function better, when the information is given to the computer in terms of *mathematical entities* like numbers, sets, relations, functions, vectors, matrices graphs, arrays, trees, records, etc., and when the *steps of solution* are generally precise, involving no ambiguity.

In order to meet the mutually conflicting requirements:

- (i) *Imprecision* of natural language, with which the human beings are comfortable, where human beings feel a problem and plan its solution.
- (ii) *Precision* of a formal system, with which computers operate efficiently, where computers execute the solution, generally planned by human beings
a new formal system viz. Fuzzy system based on the concept of 'Fuzzy' was suggested for the first time in 1965 by L. Zadeh.

In order to initiate the study of Fuzzy systems, we quote two statements to recall the difference between a precise statement and an imprecise statement.

A precise Statement is of the form: 'If income is more than 2.5 lakhs then tax is 10% of the taxable income'.

An *imprecise* statement may be of the form: 'If the *forecast* about the rain being *slightly less* than previous year *is believed*, then there is around 30% **probability** that economy may suffer heavily'.

The **concept of 'Fuzzy'**, which when applied as a **prefix/adjective to mathematical entities like set, relation, functions, tree, etc., helps us in modelling the imprecise data, information or knowledge through mathematical tools.**

Crisp Set/Relation vs. Fuzzy Set/Relation: In order to differentiate the sets, normally used so far, from the *fuzzy sets* to be introduced soon, we may call the normally called sets as *crisp sets*.

Next, we explain, how the fuzzy sets are defined, using mathematical entities, **to capture imprecise concepts**, through an example of the concept : tall.

In Indian context, we may say, a *male adult*, is

- (i) **definitely tall** if his height > 6 feet
- (ii) **not at all tall** if height < 5 feet and
- (iii) if his height = 5' 2" a **little bit tall**
- (iv) if his height = 5' 6" **slightly tall**
- (v) if height = 5' 9" **reasonably tall** etc.

Next step is **to model 'definitely tall', 'not at all tall', 'little bit tall', 'slightly tall', 'reasonably Tall'** etc. in terms of mathematical entities, e.g., numbers; sets etc. In **modelling the vague concept like 'tall', through fuzzy sets**, the numbers in the **closed set $[0, 1]$ of reals** may be used on the following lines:

- (i) '**Definitely tall**' may be represented as '*tallness having value 1*'
- (ii) '**Not at all tall**' may be represented as '*Tallness having value 0*'

other adjectives/adverbs may have values between 0 and 1 as follows:

- (iii) '**A little bit tall**' may be represented as '*tallness having value say .2*'.
 - (iv) '**Slightly tall**' may be represented as '*tallness having value say .4*'.
 - (v) '**Reasonably tall**' may be represented as '*tallness having value say .7*'.
- and so on.

Similarly, the values of other concepts or, rather, other **linguistic variables like sweet, good, beautiful**, etc. may be considered **in terms of real numbers between 0 and 1**.

Coming back to the **imprecise concept of tall**, let us think of five male persons of an organisation, viz., Mohan, Sohan, John, Abdul, Abrahm, with heights 5' 2", 6' 4", 5' 9", 4' 8", 5' 6" respectively.

Then had we talked only of crisp set of tall persons, we would have denoted the

Set of tall persons in the organisation = {Sohan}

But, a fuzzy set, representing tall persons, include **all the persons alongwith respective degrees of tallness**. Thus, **in terms of fuzzy sets**, we write:

Tall = {Mohan/.2; Sohan/1; John/.7; Abdul/0; Abrahm/.4}.

The values .2, 1, .7, 0, .4 are called **membership values or degrees**:

Note: Those elements which have value 0 may be dropped e.g.

Tall may also be written as Tall = {Mohan/.2; Sohan/1; John/.7; Abrahm/.4}, neglecting Abdul, with associated degree zero.

If we **define short/Diminutive** as exactly **opposite of Tall** we may say
Short = {Mohan/.8; Sohan/0; John/.3; Abdul/1; Abrahm/.6}

3.3 RELATIONS ON FUZZY SETS

In the case of *Crisp sets*, we have the concepts of *Equality of sets*, *Subset of a set*, and *Member of a set*, as illustrated by the following examples:

(i) **Equality of two sets**

Let $A = \{1, 4, 3, 5\}$
 $B = \{4, 1, 3, 5\}$
 $C = \{1, 4, 2, 5\}$

be three given sets.

Then, Set A is equal to set B denoted by $A = B$. But A is not equal to C, denoted by $A \neq C$.

(ii) **Subset**

Consider sets $A = \{1, 2, 3, 4, 5, 6, 7\}$
 $B = \{4, 1, 3, 5\}$
 $C = \{4, 8\}$

Then B is a subset of A, denoted by $B \subset A$. Also C is not a subset of A, denoted by $C \not\subset A$.

(iii) **Belongs to/is a member of**

If $A = \{1, 4, 3, 5\}$

Then each of 1, 4, 3 and 5 is called an *element or member* of A and the fact that 1 is a *member of A* is denoted by $1 \in A$.

Corresponding Definitions/ concepts for Fuzzy Sets

In order to define for fuzzy sets, the concepts corresponding to the concepts of *Equality of Sets*, *Subset* and *Membership of a Set* considered so far only for crisp sets, first we illustrate the concepts through an example:

Let X be the set on which fuzzy sets are to be defined, e.g.,

$X = \{\text{Mohan, Sohan, John, Abdul, Abrahm}\}$.

Then X is called the **Universal Set**.

Note: In every fuzzy set, all the elements of X with their corresponding memberships values from 0 to 1, appear.

(i) Degree of Membership: In respect of fuzzy sets, we do not speak of just 'membership', but speak of 'degree of membership'.

In the set

$A = \{\text{Mohan}/.2; \text{Sohan}/1; \text{John}/.7; \text{Abrahm}/.4\}$,

Degree (Mohan) = .2, degree (John) = .4

For (ii) Equality of Fuzzy sets: Let A, B and C be fuzzy sets defined on X as follows:

Let $A = \{\text{Mohan}/.2; \text{Sohan}/1; \text{John}/.7; \text{Abrahm}/.4\}$

$B = \{\text{Abrahm}/.4, \text{Mohan}/.2; \text{Sohan}/1; \text{John}/.7\}$.

Then, as degrees of each element in the two sets, equal; we say fuzzy set A equals fuzzy set B, denoted as $A = B$

However, if $C = \{\text{Abrahm}/.2, \text{Mohan}/.4; \text{Sohan}/1; \text{John}/.7\}$, then

$A \neq C$.

(iii) Subset/Superset

Intuitively, we know

- (i) The **Set of ‘Very Tall’ people** should be a **subset** of the set of **Tall people**.
- (ii) If the **degree of ‘tallness’** of a person is **say .5** then degree of **‘Very Tallness’** for the person should be **lesser say .3**.

Combining the above two ideas we, may say that if

$A = \{\text{Mohan}/.2; \text{Sohan}/1; \text{John}/.7; \text{Abrahm}/.4\}$ and

$B = \{\text{Mohan}/.2, \text{Sohan}/.9, \text{John}/.6, \text{Abraham}/.4\}$ and further,

$C = \{\text{Mohan}/.3, \text{Sohan}/.9, \text{John}/.5, \text{Abraham}/.4\}$,

then, in view of the fact that for each element, degree in A is greater than or equal to degree in B, **B is a subset of A** denoted as $B \subset A$.

However, degree (Mohan) = .3 in C and degree (Mohan) = .2 in A,

,therefore, C is **not** a subset of A.

On the other hand degree (John) = .5 in C and degree (John) = .7 in A,

therefore, A is also not a subset of C.

We generalize the ideas illustrated through examples above

Let **A and B be fuzzy sets** on the universal set

$X = \{x_1, x_2, \dots, x_n\}$

(X is called the Universe or Universal set)

s.t.

$A = \{x_1/v_1, x_2/v_2, \dots, x_n/v_n\}$ and

$B = \{x_1/w_1, x_2/w_2, \dots, x_n/w_n\}$

with that $0 \leq v_i, w_i \leq 1$.

Then fuzzy set A equals fuzzy set B, denoted as $A = B$, *if and only if*

$v_i = w_i$ for all $i = 1, 2, \dots, n$.

Further if **and** $w \leq v_i$ **for all i**.

then B is a fuzzy subset of A.

Example: Let $X = \{\text{Mohan}, \text{Sohan}, \text{John}, \text{Abdul}, \text{Abrahm}\}$

$A = \{\text{Mohan}/.2; \text{Sohan}/1; \text{John}/.7; \text{Abrahm}/.4\}$

$B = \{\text{Mohan}/.2, \text{Sohan}/.9, \text{John}/.6, \text{Abraham}/.4\}$

Then B is a fuzzy subset of A.

In respect of fuzzy sets vis-à-vis (crisp) sets, we may note that:

- ◆ Corresponding to the concept of ‘**belongs to**’ of (**Crisp**) **set**, we use the concept of ‘**degree of membership**’ for fuzzy sets.
- ◆ It may be noted that every **crisp set** may be thought of as a **Fuzzy Set**, but **not conversely**. For example, if **Universal set is**
 $X = \{\text{Mohan}, \text{Sohan}, \text{John}, \text{Abdul}, \text{Abrahm}\}$ and
 $A =$ set of those members of X who are **at least graduates**, say,
 $= \{\text{Mohan}, \text{John}, \text{Abdul}\}$

then we **can rewrite A as a fuzzy set** as follows:

$A = \{\text{Mohan}/1; \text{Sohan}/0; \text{John}/1; \text{Abdul}/1; \text{Abrahm}/0\}$, in which degree of each member of the crisp set, is taken as one and degree of each element of the universal set which does not appear in the set A, is taken as zero.

However, conversely, a fuzzy set may not be written as a crisp set. Let C be a fuzzy set denoting **Educated People**, where **degree of education is defined as follows**:

degree of education (Ph.D. holders) = 1
 degree of education (Masters degree holders) = 0.85
 degree of education (Bachelors degree holders) = .6
 degree of education (10 + 2 level) = 0.4
 degree of education (8th Standard) = 0.1
 degree of education (less than 8th) = 0.

Let us $C = \{\text{Mohan}/.85; \text{Sohan}/.4; \text{John}/.6; \text{Abdul}/1; \text{Abrahm}/0\}$.

Then, we cannot think of C as a crisp set.

Next, we define some more concepts in respect of fuzzy sets.

Definition: Support set of a Fuzzy Set, say C, is a crisp set, say D, containing all the elements of the universe X for which **degree of membership in Fuzzy set is positive**.

Let us consider again

$C = \{\text{Mohan}/.85; \text{Sohan}/.4; \text{John}/.6; \text{Abdul}/1; \text{Abrahm}/0\}$.

Support of C = D = {Mohan, Sohan, John, Abdul}, where **the element Abrahm does not belong to D, because, it has degree 0 in C**.

Definition: Fuzzy Singleton is a fuzzy set in which there is exactly one element which has positive membership value.

Example:

Let us define a fuzzy set OLD on universal set X in which degree of OLD is zero if a person in X is below 20 years and Degree of Old is .2 if a person is between 20 and 25 years and further suppose that

Old = C = {Mohan/0; Sohan/0; John/.2; Abdul/0; Abrahm/0},
 then support of old = **{John}** and hence old is a fuzzy singleton.

Ex. 1: Discuss equality and subset relationship for the following fuzzy sets defined on the Universal set $X = \{a, b, c, d, e\}$

$A = \{a/.3, b/.6, c/.4, d/0, e/.7\}$

$B = \{a/.4, b/.8, c/.9, d/.4, e/.7\}$

$C = \{a/.3, b/.7, c/.3, d/.2, e/.6\}$

3.4 OPERATIONS ON FUZZY SETS

For Crisp sets, we have the operations of **Union, intersection and complementation**, as illustrated by the example:

Let $X = \{x_1, x_2, \dots, x_{10}\}$

$A = \{x_2, x_3, x_4, x_5\}$

$B = \{x_1, x_3, x_5, x_7, x_9\}$

Then $A \cup B = \{x_1, x_2, x_3, x_4, x_5, x_7, x_9\}$

$A \cap B = \{x_3, x_5\}$

$A' \text{ or } X \sim A = \{x_1, x_6, x_7, x_8, x_9, x_{10}\}$

The concepts of Union, intersection and complementation for **crisp sets may be extended to FUZZY** sets after observing that for crisp sets A and B, we have

- (i) $A \cup B$ is the **smallest** subset of X **containing** both A and B .
- (ii) $A \cap B$ is the **largest** subset of X **contained in** both A and B .
- (iii) **The complement A' is such that**
 - (a) A and A' **do not have any element in common** and
 - (b) Every element of the universal set **is in either A or A'** .

Fuzzy Union, Intersection, Complementation:

In order to motivate proper definitions of these operations, we may recall

(1) when a **crisp set** is treated as a **fuzzy set** then

- (i) membership in a crisp set is indicated by degree/value of membership as 1 (one) in the corresponding Fuzzy set,
- (ii) non-membership of a crisp set is indicated by degree/value of membership as **zero** in the corresponding Fuzzy Set.

Thus, **smaller the value** of degree of membership, a sort of **lesser it is a member** of the Fuzzy set.

(2) While taking union of Crisp sets, members of both sets are included, and none else. However, in each Fuzzy set, all members of the universal set occur but their degrees determine the level of membership in the fuzzy set.

The facts under (1) and (2) above, lead us to define:

The **Union of two fuzzy sets** A and B , is the set C with the same universe as that of A and B such that, the degree of an element of C is equal to the **MAXIMUM** of degrees of the element, in the two fuzzy sets.

(if Universe $A \neq$ Universe B , then take Universe C as the union of the universe A and universe B)

The **Intersection C of two fuzzy sets A and B is the fuzzy set in which**, the degree of an element of C is equal to the **MINIMUM** of degrees in the two fuzzy sets.

Example:

$A = \{\text{Mohan}/.85; \text{Sohan}/.4; \text{John}/.6; \text{Abdul}/1; \text{Abrahm}/0\}$
 $B = \{\text{Mohan}/.75; \text{Sohan}/.6; \text{John}/0; \text{Abdul}/.8; \text{Abrahm}/.3\}$

Then

$A \cup B = \{\text{Mohan}/.85; \text{Sohan}/.6; \text{John}/.6; \text{Abdul}/1; \text{Abrahm}/.3\}$
 $A \cap B = \{\text{Mohan}/.75; \text{Sohan}/.4; \text{John}/0; \text{Abdul}/.8; \text{Abrahm}/0\}$

and, the complement of A denoted by A' is given by

$C' = \{\text{Mohan}/.15; \text{Sohan}/.6; \text{John}/.4; \text{Abdul}/0; \text{Abrahm}/1\}$

Properties of Union, Intersection and Complement of Fuzzy Sets:

The following properties which hold for ordinary sets, also, hold for fuzzy sets

Commutativity

- (i) $A \cup B = B \cup A$
- (ii) $A \cap B = B \cap A$

We prove only (i) above just to explain, how the involved equalities, may be proved in general.

Let $U = \{x_1, x_2, \dots, x_n\}$, be universe for fuzzy sets A and B
 If $y \in A \cup B$, then y is of the form $\{x_i/d_i\}$ for some i
 $y \in A \cup B \Rightarrow y = \{x_i/e_i\}$ as member of A and
 $y = \{x_i/f_i\}$ as member of B and
 $d_i = \max \{e_i, f_i\} = \max \{f_i, e_i\}$
 $\Rightarrow y \in B \cup A$.

Rest of the properties are stated without proof.

Associativity

(i) $(A \cup B) \cup C = A \cup (B \cup C)$

(ii) $(A \cap B) \cap C = A \cap (B \cap C)$

Distributivity

(i) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

(ii) $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$

DeMorgan's Laws

$$(A \cup B)' = A' \cap B'$$

$$(A \cap B)' = A' \cup B'$$

Involution or Double Complement

$$(A')' = A$$

Idempotence

$$A \cap A = A$$

$$A \cup A = A$$

Identity

$$A \cup U = U \quad A \cap U = A$$

$$A \cap \phi = A \quad \phi \cap A = \phi$$

where

ϕ : empty fuzzy set = $\{x/0 \text{ with } x \in U\}$

and

U : universe = $\{x/1 \text{ with } x \in U\}$

Ex. 2: For the following fuzzy sets

$A = \{a/.5, b/.6, c/.3, d/0, e/.9\}$ and

$B = \{a/.3, b/.7, c/.6, d/.3, e/.6\}$,

find the fuzzy sets $A \cap B$, $A \cup B$ and $(A \cap B)'$

3.5 OPERATIONS UNIQUE TO FUZZY SETS

Next, we discuss three operations, viz., *concentration*, *dilation* and *normalization*, that are relevant only to fuzzy sets and can not be discussed for (crisp) sets.

(1) Concentration of a set A is defined as

$$\text{CON}(A) = \{x/m_A^2(x) | x \in U\}$$

Example:

If $A = \{\text{Mohan}/.5; \text{Sohan}/.9; \text{John}/.7; \text{Abdul}/0; \text{Abrahm}/.2\}$

then

$\text{CON}(A) = \{\text{Mohan}/.25; \text{Sohan}/.81; \text{John}/.49; \text{Abdul}/0; \text{Abrahm}/.04\}$.

In respect of concentration, it may be noted that the associated values being between 0 and 1, on squaring, become smaller. In other words, the values concentrate towards zero. This fact may be used for giving increased emphasis on a concept. If *Brightness* of articles is being discussed, then *Very bright* may be obtained in terms of *CON. (Bright)*.

(2) Dilation (Opposite of Concentration) of a fuzzy set A is defined as

$$\text{DIL}(A) = \{x/\sqrt{m_A(x)} | x \in U\}$$

Example:

If $A = \{\text{Mohan}/.5; \text{Sohan}/.9; \text{John}/.7; \text{Abdul}/0; \text{Abrahm}/.2\}$
 then
 $\text{DIL}(A) = \{\text{Mohan}/.7; \text{Sohan}/.95; \text{John}/.84; \text{Abdul}/0; \text{Abrahm}/.45\}$

The associated values, that are between 0 and 1, on taking square-root get increased, e.g., if the value associated with x was .01 before dilation, then the value associated with x after dilation becomes .1, i.e., ten times of the original value. This fact may be used for *decreased emphasis*. For example, if colour say 'yellow' has been considered already, then 'light yellow' may be considered in terms of already discussed 'yellow' through Dilation.

(3) Normalization of a fuzzy set, is defined as

$$\text{NORM}(A) = \left\{ x / \left(\frac{m_A(x)}{\text{Max}} \right) \mid x \in U \right\}.$$

$\text{NORM}(A)$ is a fuzzy set in which membership values are obtained by dividing values of the membership function of A by the maximum membership function.

The resulting fuzzy set, called the **normal**, (or **normalized**) **fuzzy set**, has the maximum of membership function value of 1.

Example:

If $A = \{\text{Mohan}/.5; \text{Sohan}/.9; \text{John}/.7; \text{Abdul}/0; \text{Abrahm}/.2\}$
 $\text{Norm}(A) = \{\text{Mohan}/(.5 \div .9 = .55.); \text{Sohan}/1; \text{John}/(.7 \div .9 = .77.); \text{Abdul}/0; \text{Abrahm}/(.2 \div .9 = .22.)\}$

Note: If one of the members has value 1, then $\text{Norm}(A) = A$,

Relation & Fuzzy Relation

We know from our earlier background in Mathematics that a relation from a set A to a set B is a subset of $A \times B$.

For example, The relation of father may be written as $\{\{\text{Dasrath, Ram}\}, \dots\}$, which is a subset of $A \times B$, where A and B are sets of persons living or dead.

The relation of Age may be written as

$$\{(\text{Mohan}, 43.7), (\text{Sohan}, 25.6), \dots\},$$

where A is set of living persons and B is set of numbers denoting years.

Fuzzy Relation

In fuzzy sets, every element of the universal set occurs with some degree of membership. **A fuzzy relation may be defined in different ways.** One way of defining fuzzy relation is to assume the underlying sets as crisp sets. We will discuss only this case.

Thus, a **relation from A to B, where we assume A and B as crisp sets, is a fuzzy set, in which with each** element of $A \times B$ is associated a degree of membership between zero and one.

For example:

We may define the relation of UNCLE as follows:

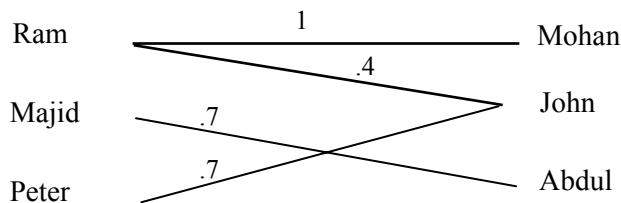
- (i) x is an UNCLE of y with degree **1** if x is brother of mother or father,
- (ii) x is an UNCLE of y with degree **.7** if x is a brother of an UNCLE of y , and x is not covered above,
- (iii) x is an UNCLE of y with degree **.6** if x is the son of an UNCLE of mother or father.

Now suppose

Ram is UNCLE of Mohan with degree **1**, Majid is UNCLE of Abdul with degree **.7** and Peter is UNCLE of John with degree **.7**. Ram is UNCLE of John with degree **.4**. Then **the relation of UNCLE can be written** as a set of ordered-triples as follows:

$\{(Ram, Mohan, 1), (Majid, Abdul, .7), (Peter, John, .7), (Ram, John, .4)\}$.

As in the case of ordinary relations, we can use matrices and graphs to represent FUZZY relations, e.g., the relation of UNCLE discussed above, may be graphically denoted as



Fuzzy Graph

Fuzzy Reasoning

In the rest of this section, we just have a fleeting glance on Fuzzy Reasoning. Let us recall the well-known Crisp Reasoning Operators

- (i) AND
- (ii) OR
- (iii) NOT
- (iv) IF P THEN Q
- (v) P IF AND ONLY IF Q

Corresponding to each of these operators, there is a fuzzy operator discussed and defined below. For this purpose, we assume that P and Q are fuzzy propositions with associated degrees, respectively, $\deg(P)$ and $\deg(Q)$ between 0 and 1.

The $\deg(P) = 0$ denotes P is False and $\deg(P) = 1$ denotes P is True.

Then the operators are defined as follows:

(i) **Fuzzy AND to be denoted by \wedge , is defined as follows:**

For given fuzzy propositions P and Q, the expression $P \wedge Q$ denotes a fuzzy proposition with $\text{Deg}(P \wedge Q) = \min(\text{deg}(P), \text{deg}(Q))$

Example: Let P: *Mohan is tall* with $\text{deg}(P) = .7$

Q: *Mohan is educated* with $\text{deg}(Q) = .4$

Then $P \wedge Q$ denotes: '*Mohan is tall and educated*' with degree $((\min)\{.7, .4\}) = .4$

(ii) **Fuzzy OR to be denoted by \vee , is defined as follows:**

For given fuzzy propositions P and Q, $P \vee Q$ is a fuzzy proposition with

$\text{Deg}(P \vee Q) = \max(\text{deg}(P), \text{deg}(Q))$

Example: Let P: *Mohan is tall* with $\text{deg}(P) = .7$

Q: *Mohan is educated* with $\text{deg}(Q) = .4$

Then $P \vee Q$ denotes: '*Mohan is tall or educated*' with degree $((\max)\{.7, .4\}) = .7$

3.6 NON-MONOTOMIC REASONING SYSTEMS

Monotonic Reasoning: The conclusion drawn in PL and FOPL are only through (valid) deductive methods. When some axiom is *added* to a PL or an FOPL system, then, through deduction, we can draw *more* conclusions. Hence, more additional facts become available in the knowledge base with the addition of each axiom. Adding of axioms to the knowledge base increases the amount of knowledge contained in the knowledge base. Therefore, the set of facts through inferences in such systems **can only grow larger** with addition of each axiomatic fact. Adding of new facts can not reduce the size of K.B. Thus, amount of knowledge **monotonically** increases with the number of independent premises due to new facts that become available.

However, in everyday life, many times in the light of new facts that become available, we may have to revise our earlier knowledge. For example, we consider a sort of deductive argument in FOPL:

- (i) Every bird can fly long distances
 - (ii) Every pigeon is a bird. (iii) Tweety is a pigeon.
- Therefore, Tweety can fly long distances.

However, later on, we come to know that Tweety is actually a hen and a hen cannot fly long distances. Therefore, we have to revise our belief that Tweety can fly over long distances.

This type of situation is not handled by any monotonic reasoning system including PL and FOPL. This is appropriately handled by Non-Monotomic Reasoning Systems, which are discussed next.

A **non-monotomic reasoning system** is one which allows *retracting of old knowledge due to discovery of new facts* which contradict or invalidate a part of the current knowledge base. Such systems also take care that retracting of a fact may necessitate a chain of retractions from the knowledge base or even reintroduction of earlier retracted ones from K.B. Thus, *chain-shrink* and *chain emphasis* of a K.B and reintroduction of earlier retracted ones are part of a non-monotomic reasoning system.

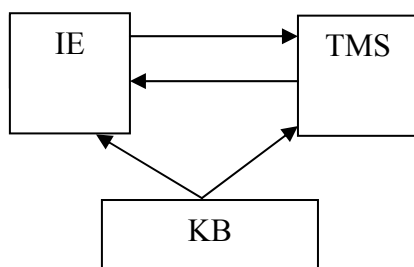
To meet the requirement for reasoning in the real-world, we need non-monotonic reasoning systems also, in addition to the monotonic ones. This is true specially, in view of the fact that it is not reasonable to expect that **all the knowledge needed** for a set of tasks could be acquired, validated, and loaded into the system just at the outset. In general, initial knowledge is an *incomplete* set of *partially true* facts. The set may also be redundant and may contain inconsistencies and other sources of uncertainty.

Major components of a Non-Monotonic reasoning system

Next, we discuss a typical non-monotonic reasoning system (NMRS) consists of the following three major components:

- (1) Knowledge base (KB),
- (2) Inference Engine (IE),
- (3) Truth-Maintenance System (TMS).

The **KB** contains information, facts, rules, procedures etc. relevant to the type of problems that are expected to be solved by the system. The component **IE** of NMRS gets facts from KB to draw new inferences and sends the new facts discovered by it (i.e., IE) to KB. The component **TMS**, after addition of new facts to KB, either from the environment or through the user or through IE, checks for validity of the KB. It may happen that the new fact from the environment or inferred by the IE may conflict/contradict some of the facts already in the KB. In other words, an *inconsistency* may arise. In case of inconsistencies, TMS retracts some facts from KB. Also, it may *lead to a chain of retractions* which may require interactions between KB and TMS. Also, some new fact either from the environment or from IE, may invalidate some earlier retractions *requiring reintroduction* of earlier retracted facts. This may lead to a chain of reintroductions. These retrievals and introductions are taken care of by TMS. The IE is completely relieved of this responsibility. Main job of IE is *to conclude* new facts when it is supplied a set of facts.



Next, We explain the ideas discussed above through an example:

Let us assume KB has two facts P and $\sim Q \rightarrow \sim P$ and a rule called *Modus Tollens*. When **IE** is supplied these knowledge items, it concludes Q and sends Q to KB. However, through interaction with the environment, **KB** is later supplied with the information that $\sim P$ is more appropriate than P . Then **TMS**, on the addition of $\sim P$ to KB, finds that KB is no more consistent, at least, with P . The knowledge that $\sim P$ is more appropriate, suggests that P be retracted. Further Q was concluded *assuming P as True*. But, in the new situation in which P is assumed to be not appropriate, Q also becomes inappropriate. **P and Q are not deleted from KB, but are just marked as dormant or ineffective.** This is done in view of the fact that later on, if again, it is found appropriate to include P or Q or both, then, instead of requiring some mechanism for adding P and Q , we just remove marks that made these dormant.

Non-monotonic Reasoning Systems deal with

- 1) Revisable belief systems
- 2) incomplete K.B.
 - Default Reasoning
 - \ Closed World assumption

3.7 DEFAULT REASONING SYSTEMS

In the previous section, we discussed *uncertainty due to beliefs* (which are not necessarily *facts*) where beliefs are changeable. Here, we discuss *another form of uncertainty* that occur as a result of *incompleteness* of the available knowledge at a particular point of time.

One method of handling uncertainty due to **incomplete** KB is through **default reasoning** which is also a form of non-monotonic reasoning and is based on the following mechanism:

Whenever, for any entity relevant to the application, information is not in the KB, then a **default value** for that type of entity, is assumed and is assigned to the entity. The default assignment is not arbitrary but is based on experiments, observations or some other rational grounds. However, the typical value for the entity is removed if some information contradictory to the assumed or default value becomes available.

The advantage of this type of a reasoning system is that we need not store all facts regarding a situation. **Reiter has given one theory of default reasoning, which is expressed as**

$$\frac{a(x) : Mb_1(x), \dots, Mb_k(x)}{C(x)} \quad (A)$$

where M is a *consistency operator*.

The inference rule (A) states that if $a(x)$ is true and none of the conditions $b_k(x)$ is in conflict or contradiction with the K.B, then you can deduce the statement $C(x)$

The idea of default reasoning is explained through the following example:

Suppose we have

$$(i) \quad \frac{\text{Bird}(x) : \text{Mfly}(x)}{\text{Fly}(x)}$$

(ii) Bird (twitty)

M fly (x) stands for a statement of the form ‘*KB does not have any statement of the form that says x does not have wings etc, because of which x may not be able to fly*’. In other words, *Bird (x) : M fly (x)* may be taken to stand for the statement ‘*if x is a normal bird and if the normality of x is not contradicted by other facts and rules in the KB*.’ then we can assume that *x can fly*. Combining with *Bird (Twitty)*, we conclude that if KB does not have any facts and rules from which, it can be inferred that *Twitty can not fly*, then, we can conclude that *twitty can fly*.

Further, suppose, KB also contains

(i) Ostrich (twitty)

(ii) $\text{Ostrich}(x) \rightarrow \sim \text{FLY}(x)$.

From these two facts in the K.B., it is concluded that Twitty being an ostrich, can not fly. In the light of this knowledge the fact that Twitty can fly has to be withdrawn. Thus, Fly (twitty) would be locked. Because, default Mfly (Twitty) is now inconsistent.

Let us consider another example:

$$\frac{\text{Adult}(x) : \text{Mdrive}(x)}{\text{Drive}(x)}$$

The above can be interpreted in the default theory as:

If a person x is an adult and in the knowledge base there is no fact (*e.g., x is blind, or x has both of his/her hands cut in an accident etc*) which tells us something making x incapable of driving, **then x can drive**, is assumed.

3.8 CLOSED WORLD ASSUMPTION SYSTEMS

Another mechanism of handling incompleteness of a KB is called ‘Closed World Assumption’ (CWA).

This mechanism is useful in applications where *most of the facts are known* and therefore it is reasonable to assume that if a proposition cannot be proved, then it is FALSE. This is called CWA with failure as negation.

This means if a ground atom $P(a)$ is not provable, then assume $\sim P(a)$. A predicate like LESS (x, y) becomes a **ground atom** when the variables x and y are replaced by constants say x by 2 and y by 3, so that we get the ground atom LESS (2, 3).

Example of an application where CWA is reasonable is that of *Airline reservation* where city-to-city flight not explicitly entered in the flight schedule or time table, are assumed not to exist.

AKB is **complete** if for each ground atom $P(a)$; either $P(a)$ or $\sim P(a)$ can be proved.

By the use of CWA any incomplete KB becomes complete **by the addition of the meta rule**:

If $P(a)$ can not be proved then assume $\sim P(a)$.

Example of an incomplete K.B: Let our KB contain only

- (i) $P(a)$.
- (ii) $P(b)$.
- (iii) $P(a) \rightarrow Q(a)$.
- (iv) Rule of Modus Ponens: From P and $P \rightarrow Q$, conclude Q .

The above KB is *incomplete* as we can not say anything about $Q(b)$ (or $\sim Q(b)$) from the given KB.

Remarks: In general, KB argumented by CWA need *not be* consistent i.e., it may contain two mutually conflicting wffs. For example, if our KB contains only $P(a) \vee Q(b)$.

(Note: from $P(a) \vee Q(b)$, we can not conclude either of $P(a)$ and $Q(b)$ with definiteness)

As neither $P(a)$ nor $Q(b)$ is provable, therefore, we add $\sim P(a)$ and $\sim Q(b)$ by using CWA.

But, then, the set of $P(a) \vee Q(b)$, $\sim P(a)$ and $\sim Q(b)$ is inconsistent.

3.9 OTHER NON-DEDUCTIVE SYSTEMS

PL and FOPL are *deductive* inferencing systems: i.e., the conclusions drawn are *invariably true* whenever the premises are *true*. However, due to limitations of these systems for making inferences, as discussed earlier, we must have other systems inferences. In addition to *Default Reasoning systems* and *Closed World Assumption systems*, we have the following useful reasoning systems:

- 1) **Abductive inference** System, which is based on the use of causal knowledge to explain and justify a (*possibly invalid*) conclusion.

Abduction Rule

$$\frac{P \rightarrow Q \quad Q}{P}$$

Note that *abductive inference rule* is different from *Modus Ponens inference rule* in that in abductive inference rule, the *consequent* of $P \rightarrow Q$, i.e., Q is assumed to be given as True and the *antecedent* of $P \rightarrow Q$, i.e., P is inferred.

The abductive inference is useful in *diagnostic applications*. For example while diagnosing a disease (*say P*), the doctor asks for the symptoms (*say Q*). Also, the doctor knows that for given the disease, say, Malaria (P); the symptoms include high fever starting with feeling of cold etc. (Q)

i.e., doctor knows $P \rightarrow Q$

The doctor then attempts to diagnose the disease (i.e., P) from symptoms. However, it should be noted that the conclusion of the disease from the symptoms may not always be correct. In general, abductive reasoning leads to correct conclusions, but the conclusions may be *incorrect* also. In other words, Abductive reasoning is not a **valid form** of reasoning.

Inductive Reasoning is a method of generalisation from a finite number of instances.

The rule, generally, denoted as

$$\frac{P(a_1), P(a_2), \dots, P(a_n)}{(x) P(x)},$$

states that from n instances $P(a_i)$ of a predicate/property $P(x)$, we infer that $P(x)$ is *True for all x* .

Thus, from a finite number of observations about some property of objects, we generalize, i.e., make a *general* statement *about all* the elements of the domain in respect of the property.

For example, we may, conclude that: *all cows are white*, after observing a large number of white cows. However, this conclusion may have some exception in the sense that we may come across a black cow also. Inductive Reasoning like Abductive Reasoning, Closed World Assumption Reasoning and Default Reasoning is not *irrefutable*. In other words, these reasoning rules lead to conclusions, which may be True, but not necessarily always.

However, all the rules discussed under Propositional Logic (PL) and FOPL, including Modus Ponens etc are deductive i.e., lead to irrefutable conclusions.

3.10 SUMMARY

In this unit, we briefly discussed some formal systems which take care of at least one of the blemishes in the knowledge base, namely, of inconsistency, imprecision and incompleteness of the knowledge base. In Sections 4.2 to 4.5, we discuss Fuzzy systems, which attempt to handle imprecision due to use of words, having multiple meanings, of a natural language. The words appear in the description of the problems to be solved by man-machine systems.

In Section 4.6, we briefly discuss non-monotonic (formal) systems, which mainly deal with problems involving *beliefs*, in stead of *facts*. A belief may be revised by the believer, when strong evidence becomes available for the revision of the belief.

In Sections 4.7 and 4.8, we discuss two formal systems which attempt to deal with *incompleteness* of the available knowledge of the problem domain. *Default reasoning* systems discussed in Section 4.7, attempt to handle the problem of incompleteness of knowledge, through assumption of default values for the missing values. The default values may be withdrawn, in case some knowledge contrary to the default values, becomes available.

On the other hand, another formal system, viz., *closed world assumption system* discussed in Section 4.8, assume that, if the truth of a statement is not available in the knowledge base, then assume the statement false. Finally, in Section 4.9, we discuss some inference rules, namely, *abductive* and *inductive* rules, which though are not deductive, yet prove quite useful in everyday problems, particularly, in diagnostic problems.

3.11 SOLUTIONS/ANSWERS

Ex. 1: Both A and C are subsets of the fuzzy set B, because $\deg(x \text{ in } A) \leq \deg(x \text{ in } B)$ for all $x \in X$

Similarly $\deg(x \text{ in } C) \leq \deg(x \text{ in } B)$ for all $x \in X$

Further, A is not a subset of C, because,

$\deg(c \text{ in } A) = .4 > .3 = \deg(c \text{ in } C)$

Also, C is not a subset of A, because,

$\deg(b \text{ in } C) = .7 > .6 = \deg(b \text{ in } A)$

Ex. 2: $A \cap B = \{a/.3, b/.6, c/.3, d/0, e/.6\}$,

where $\deg(x \text{ in } A \cap B) = \min \{ \deg(x \text{ in } A), \deg(x \text{ in } B) \}$.

$A \cup B = \{a/.5, b/.7, c/.6, d/.3, e/.9\}$,

where $\deg(x \text{ in } A \cup B) = \max \{ \deg(x \text{ in } A), \deg(x \text{ in } B) \}$.

The fuzzy set $(A \cap B)'$ is obtained from $A \cap B$, by the rule:

$\text{degree}(x \text{ in } (A \cap B)') = 1 - \text{degree}(x \text{ in } A \cap B).$

Hence

$$(A \cap B)' = \{ a/.7, b/.4, c/.7, d/1, e/.4 \}$$

3.12 FURTHER READINGS

1. Munikata, Toshinori *Chapter 5 of Fundamentals of the New Artificial Intelligence: Beyond Traditional Paradigm* (springer, 1998).
2. Nguyen, H.T. Walker E.A. *A First Course in Fuzzy Logic* (CRC Press, 1997)
3. Patterson, D.W. *Introduction to Artificial Intelligence and Expert Systems*(Prentice-Hall of India, 2001)

UNIT 1 INTRODUCTION TO INTELLIGENCE AND ARTIFICIAL INTELLIGENCE

Structure	Page Nos.
1.0 Introduction	5
1.1 Objectives	6
1.2 Some Simple Definition of A.I.	6
1.3 Definition by Eliane Rich	6
1.4 Definition by Buchanin and Shortliffe	8
1.5 Another Definition by Elaine Rich	12
1.6 Definition by Barr and Feigenbaum	13
1.7 Definition by Shalkoff	18
1.8 Summary	19
1.9 Further Readings/References	20

1.0 INTRODUCTION

In this unit, we discuss intelligence, both machine and human. However, as our subject matter in the course is machine intelligence, or artificial intelligence, our discussion of the subject matter is mainly from the point of view of machine intelligence. Machine intelligence is popularly known as *Artificial Intelligence* and is generally referred to by its abbreviation viz. **AI**. We also shall use the name AI for the discipline throughout. The style of discussion in this unit is to start with a definition of AI by some pioneer in the field, and then elaborate the ideas involved in the definition. Further, while elaborating the ideas involved in the definition, we introduce a number of relevant new ideas, concepts and definitions to be used later. In this process, we have introduced and/or explained the following:

- i) Artificial Intelligence & Human Intelligence
- ii) When a problem necessarily requires parallel processing for its solution
- iii) Symbol vs. number issue
- iv) Numeric vs. symbolic processing
- v) Algorithm vs. non-algorithmic method and limitation of algorithmic approach
- vi) Limitations of computational abilities of logical devices
- vii) Heuristics – an important A.I technique
- viii) Time/space complexities of programs and problems, exponential time vs. polynomial time, hard problems
- ix) Role of search and knowledge in solving hard problems; search as an important AI technique
- x) Enumeration of issues about knowledge
- xi) Information: one of the four fundamental properties of nature
- xii) Organisation; relations between information and organisation and between information and intelligence
- xiii) A principle of intelligence
- AI as a science and as an engineering discipline
- xiv) Controversial issue about the possibility of machine intelligence at least equating or surpassing human intelligence.
- xv) Brief history of AI ... the name and as a subject

1.1 OBJECTIVES

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

- discuss the concepts of ‘intelligence’ and artificial intelligence’ as visualised by a number of leading experts in the field;
 - enumerate the fields in which human beings are still better than computers;
 - tell the difference between the concepts of:
 - (i) Symbol and number
 - (ii) Algorithmic and non-algorithmic methods
 - (iii) Information and knowledge
 - (iv) Polynomial time and exponential time complexities
 - tell the relation of *information* to *organisation* and to *intelligence*.
-

1.2 SOME SIMPLE DEFINITIONS OF A.I.

Before looking at what A.I. is in the expert’s opinions that involve technical terms needing some explanation, we state below three simple definitions from completely non-specialists’ point of view:

1. A.I. is the study of making computers smart.
2. A.I. is the study of making computer models of human intelligence; and finally
3. A.I. is the study concerned with building machines that simulate human behaviour.

The **first one** of the above definitions is based on **behaviour-oriented** approach to A.I. According to this approach, AI is concerned with programming computers *to behave* intelligently. The **next definition** is more from a **psychologists** point of **view**, where the purpose is to use computer as a tool to understand better the *mechanisms of the human mind*, and the **final definition**, which we may call **robotic approach to A.I.**, includes under the domain of A.I., not only writing of computer programs but *building also* the whole of an intelligent system or *machine including its mechanical, electronic, optical components and other components*.

In order to have still better and concrete opinion about what is AI and its subject-matter, we consider definitions suggested by leading writers and pioneer contributors to the development of A.I. We supplement these definitions with comments to facilitate the understanding of the underlying ideas and of the technical terms involved in the definitions.

1.3 DEFINITION BY ELIANE RICH

Definition 1: The first definition we consider is **by Elaine Rich**, the author of the book entitled ‘*Artificial Intelligence*’[1]. **It states: Artificial Intelligence is the study of how to make computers do things, at which, at the moment, people are better.**

Comment 1, Definition 1: Implicit in the Rich’s definition is the idea that there are mental tasks that computers can do better than human beings and *vice-versa*, there are tasks which *at the moment* human beings can do better than computers. It is well-known that **computers are better than human beings** in the matter of

- *numerical computation*,

- *information storage, and*
- *repetitive tasks.*

On the other hand, at the moment, **human beings are much better than machine** in the matter of

- *understanding* including the capability of explaining,
- predicting the behaviour and structure of a system,
- in the matter of *common-sense reasoning*,
- in drawing conclusions when available information is either incomplete, inconsistent or even both, and
- also, in visual understanding and speech understanding, which require simultaneous availability (availability in parallel) of large amount of information.

In essence, it is found that **computers are better than human beings in tasks requiring sequential but fast computations**, where **human beings are better than computers in tasks, requiring essentially parallel processing**. In order to clarify what it is for a problem to essentially require parallel processing for its solution, we consider the following problem:

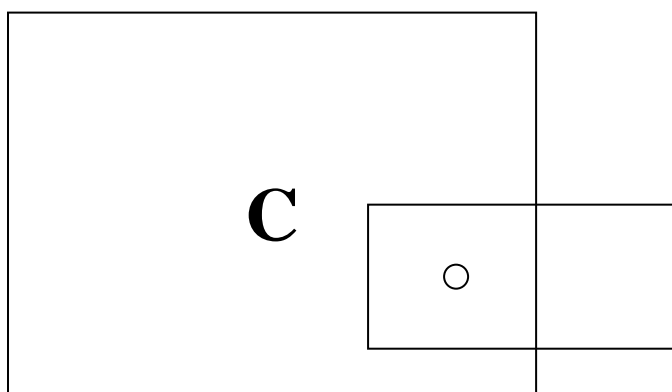
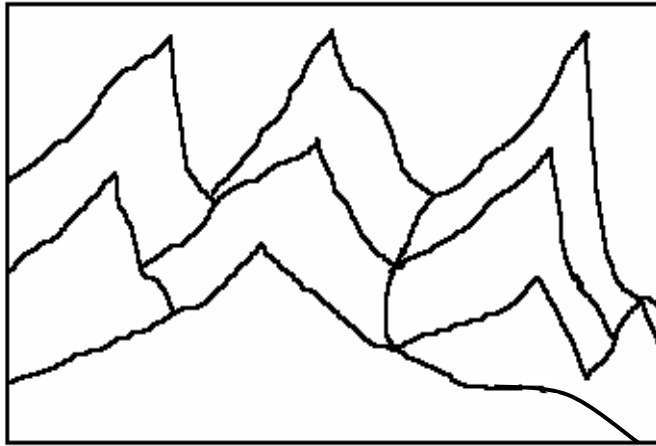


Figure 1.1

We are given a paper with some letter, say, C written on it and a card-board with a pin-hole in it. The card board is placed on the paper in such a manner that the letter is fully covered by the card board as shown in *Figure 1.1*. We are allowed to look at the paper only through the pin-hole in the card-board. The problem is to tell correctly the letter written on the paper by just looking through the pin-hole. As the information about the black and white pixels is not available simultaneously, it is not possible to figure out the letter written on the paper. The figuring out the letter on the paper requires, simultaneous availability of the whole of the grey-level information of all the points constituting the letter and its surrounding on the paper. The gray-level information of the surrounding of the letter provides the context in which to interpret the letter.

We consider another example that shows the significance of contextual information or knowledge and its simultaneous availability for visual understanding. From the following picture, we can conclude that one of the curved lines represents a river and other curved lines represent sides of the hills only on the basis of the simultaneous availability of information of the pixels.



Contextual information plays a very important role not only in the visual understanding but also in the language and speech understanding. In case of speech understanding, consider the following example, in which the word ‘*with*’ has a number of meanings (or connotations) each being determined by the context.

- Mohan saw the boy in the park *with* a telescope.
- Mohan saw the boy in the park *with* a dog.
- Mohan saw the boy in the park *with* a statue.

Further, the phrase ‘*for a long time*’ may stand for a few hours to millions of years, but again determined by the context, as explained below.

For a long time.....

- He waited in the doctor’s room *for a long time*.
- It has not rained *for a long time*.
- Dinosaurs ruled the earth *for a long time*.

Comment 2, Definition 1: In addition to the advantage that human beings have in the matter of parallel processing as explained above, **Boden [12]** says: *humans have two psychological strengths which are yet to be approached by computer systems: a teeming richness of conceptual sources and the ability to evaluate new ideas in many different ways. The first of these is difficult enough for AI to emulate, the second is even more problematic.*

Comment 3, Definition 1: The definition is rather weak in the sense that it fails to include some areas of potentially large importance viz, problems that can be solved at present neither by human beings nor by computers. Also, it may be noted that, by and by, if computer systems become so powerful that there is no problem left, which human beings can solve better than computers, then nothing is left of AI according to this definition.

1.4 DEFINITION BY BUCHANIN AND SHORTLIFFE

Next, we consider a definition obtained by rephrasing and combining the two definitions, viz., the first by **Bruce G. Buchanin** as given in ‘*Encycolopedia Britanica*’ and the second by **BUCHANIN & SHORTLIFFE** as given in *Rule-Based Expert Systems [2]*. **It states:**

Definition 2 AI is the branch of computer science that deals with *symbolic* rather than numeric processing and *non-algorithmic* methods including the *rules of thumb* or *heuristics* instead of *algorithms* as *techniques* for solving problems.

Comments/Explanations 1, Definition 2: Symbolic processing vs numeric

processing: We generally think and use 128 as a *number* which has a definite relation with the number say 105 (*that of greater than*), also with 64 (*that of being double of*) and again with 2 (*that of being a multiple of*). Also, 128 can be multiplied, through *built-in mechanisms*, with any number say 3 to get 384. However, *if* the numbers mentioned above including 128 denote the *route numbers of buses or house numbers a residential colony* then none of the relations or operations mentioned above, may hold. Rather, in this context, these relations of 128 w.r.t. 105, etc. and the operations like multiplication even *do not make any sense*. We cannot tell what is meant by saying ‘House Number 128 is greater than House Number 105’ in a normally acceptable way.

On the other hand, *even a non-digital character sequence* say ‘ABC’ may represent a *number*, for example, in hexadecimal number system. Also, words of English (or any other) language *when considered lexicographically ordered, acquire some numeric attributes*.

The conclusion we draw from the above discussion, is that a word as a sequence of characters (including digits) may denote a number or a symbol (**henceforth, a symbol stands for non-numeric symbol**) depending upon the *context* in which it is used. *And the context is determined by the nature of the problem under consideration*. If the problem can be solved using only numerical aspects of the objects in the domain and environment of the problem, then we have the advantage of having *built-in relations* (like less than, equal to etc.) and the *built-in operations* (like +, -, * etc.) that can be readily used without having to define these relations and operations explicitly.

But, unfortunately, most of the problems, we encounter for our day to day survival or even for our intellectual pursuits, involve *not only quantitative, but qualitative aspects also of the objects* of the problem domain. In order to solve these problems, we use *common sense reasoning, exploit our capability for visual and linguistic understanding, try to get meaning out of incomplete and even inconsistent information that is available*, in addition to a number of other known and unknown mechanism. **Qualitative aspects**, their ideal representations, defining relations and operations involving these aspects, are generally different for different types of problems. Hence, it is impossible to capture in general relevant relations and operations for all types of problems, and then defining these as built-in operations of the machine, because there are potentially infinite types of problems that we encounter and try to solve.

This discussion explains the basic difference between numeric processing and (non-numeric) symbolic processing. Summarizing, **numeric processing involves only** a small number of well-defined relations and operations having universally accepted meanings, and hence, these relations and operations can be incorporated as a part of a computer system. On the other hand, **in symbolic processing** the relations and operations required to solve a problem *depend upon* the problem under consideration, and hence, have to be defined explicitly along-with or as a part of programs constituting the solutions of the problems.

The weakness of numeric processing, however, is that it can be used in solving a small fraction of the set of problems we want to or need to solve. The numeric processing can be used in solving only those problems, the solutions of which involve only *numeric aspects* of the objects involved in the domain and environment of the problem under consideration. For the solution of other solvable problems, we need to

use *symbolic processing*. It is not out of place to mention **that not all problems which even can be stated precisely or formally, are amenable to computer solutions using even symbolic processing**. More discussion in this respect follows next, under Comments/Explanations 2 for Definition 2.

Comments/Explanation 2, Definition 2 : Algorithmic method vs non-algorithmic method, heuristics : We recall that an **Algorithm** is a step-by-step procedure with well-defined starting and ending points, which is guaranteed to reach a solution to a specific problem. A solution to a problem which can be expressed as an algorithm is called an *algorithmic solution*. An algorithmic solution may involve only numeric processing or may involve symbolic processing with/without numeric processing. *For the purpose of further discussion, ‘symbolic processing’ includes/subsumes numeric processing.* **Algorithmic approach even when using symbolic processing has limitations. During 1930’s, a number of logicians and mathematicians including Gödel, Church, Post, Turing and Kleene suggested a number of mathematical models of a computer, and through these models tried to explain the nature of computation, established a number of useful results about computation and also found the limits of computational power.**

They proved that even through a problem may be expressed precisely *or formally* (i.e., *in terms of mathematical entities like sets, relations functions etc.*), yet it need not yield to an *algorithmic* solution. A problem which has at least one algorithmic solution is called a **solvable problem**. They further proved that out of even solvable problems, only a small fraction can be solved if only *feasible amount* of resources like, time and space are used. Informally, **feasible amount** of resources means that the requirement for resources does not increase too rapidly with the increase in *size* of the problem. The notion of the **size of a problem** will be defined formally later on (*under comment 1 on Definition 3*). However, an *intuitive idea about the concept of the size of a problem* and its role in estimating the resource requirement for solving the problem can be had through the simple problem of calculating income tax for each of the tax-payers. The requirement of resources like, time and computing equipment for *1000 tax-payers* would be much less, as compared to the requirement of resources for computing income-tax for *one million tax payers*. In this problem, *n, the number of tax-payers for whom the income-tax is to be calculated, may be taken as size of the problem.*

This limitation and other difficulties with algorithmic solutions has given impetus to efforts for finding non-algorithmic solutions of problems. **Neural Network approach** to solving many difficult problems, is a well-known **alternative to algorithmic methods** of solving problems. In AI, there are mainly two approaches to solve problems, which generally difficult to solve with algorithmic methods. One approach is Neural approach, mentioned just above. The other approach is called symbolic approach. The symbolic approach cannot be said to be non-algorithmic. The main difference between symbolic approach of AI and algorithmic approach is that symbolic approach of AI emphasizes exploitation of the knowledge of the domain and the environment of the problem under consideration. Some of this knowledge is in the form rules of thumb, generally, called heuristics in AI.

In order to realise **the limitations of algorithmic approach to solving problems**, we need not refer to highly theoretical work by the earlier mentioned logicians/mathematicians. *The limitation of the approach may be appreciated through the following simple example.*

Consider the problem of crossing from one side over to the other side of a busy road on which a number of vehicles are moving at different velocities. A step-by-step (i.e., algorithmic) method of solving this problem may consist of:

- (i) Knowing (exactly) the distances of various vehicles from the path to be followed to cross over.
- (ii) Knowing the velocities and accelerations of the various vehicles moving on the road within a distance of, say, one kilometer.
- (iii) Using Newton's Laws of motion and their derivatives like $s = ut + \frac{1}{2}at^2$, and calculating the times that would be taken by each of the various vehicles to reach the path intended to be followed to cross over.
- (iv) Adjusting dynamically our speeds on the path so that no collision takes place with any of the vehicle moving on the road.

The above is a systematic step-by-step method, i.e., *an algorithm*, of crossing the road that may ensure no collision with any vehicle. But, how many of us can follow it? Hardly anybody! *First of all*, it is practically impossible to measure distances, velocities and accelerations of various vehicles on the road, even within a radius of one kilometer. *Secondly*, even if we assume theoretically that it is possible to measure distances, velocities and accelerations of various vehicles and to calculate safe timings to cross the road, we would not like or care to follow the above-mentioned algorithm, because *our past experience*, our sense of survival and other built-in mechanisms have allowed us, in the past, to cross over safely without following any *systematic method*. All of us just *guess* the distances of the vehicles, safe enough to cross over, and then actually cross over at an appropriate time. Not even one in 1000, on an average gets hurt when crossing a road using only guesses, in a crowded city like, Delhi, where movement of vehicles is one of the most chaotic and unruly in the whole world. However, *this is not to deny that once in a while, the guess is incorrect and someone or other gets hurt or even is killed almost every day*.

Each one of us every day, comes across hundreds of problems similar to the one of crossing of a road. And, for each such problem one uses a good guess and one generally is able to solve the problem satisfactorily each time, though the solutions may not be the best possible ones. And, or once in a while, we even fail to get any solution using the guess. However, if we insist on only following a systematic step-by-stop method that guarantees best possible solution for solving each problem, then we would hardly be able to make any progress in our day to day business of even mere survival.

The essence of the above discussion is that while attempting solutions of many of the problems, it is not only desirable but almost essential that for each of such problems we follow some **good guess instead of following a step-by-step systematic method** that guarantees the best solution. In **A.I.**, these guesses are called **heuristics**. In later chapters, we discuss heuristics in detail. However, for the time being, we state that **heuristics** are good guesses, possibly based on past experience, judgement, intuition or hunches, which lead us most of the time to reasonably good solutions, though these guesses do not guarantee the best solutions or even any solution for every instance of the problem under consideration.

The advantage of using heuristics is that we do not have to rethink completely everytime we are faced with a problem of the type of which another problem has already been solved satisfactorily. If we have a *handy rule of thumb* that may apply to the current problem, it may suggest to us how to proceed.

1.5 ANOTHER DEFINITION BY ELAINE RICH

The next definition, again by Elaine Rich [1] is more technical and involves some concepts from Theory of Computation. It states:

Definition 3: Artificial Intelligence is the study of techniques for solving exponentially hard problems in polynomial time exploiting knowledge about the problem domain.

Comments/Explanations 1, Definition 3: For deeper understanding of the concepts like *hard, solvable and unsolvable problems*, any one of the books by Brady [3], by Lewis and Papadimitriou [4] or by Hopcroft and Ullman [5] may be consulted. However, for our purpose of appreciating Definition 3 of A.I., we briefly discuss only the required essentials from Theory of Computation (TOC). In the comments on Definition 2, we have already talked about the mathematical models of computation and also about the limitations of algorithmic solutions.

As computer study is partly engineering in nature, in the sense that we design and implement or produce computer solutions for different types of problems and hence these products, i.e., solutions, need to be evaluated *vis-a-vis* problem specifications and other measures like, efficiency in respect of time and space requirements of the solutions. In order to measure the efficiency of a suggested computer solution of a problem, the earlier mentioned logicians/mathematicians suggested the concepts of **time complexity** and **space complexity** for the solutions and even for the problems. The basic idea behind these complexity measures is that all the operations that a computer (present or future generations) can execute, may be thought of as composed of a *small number of* basic operations. These basic operations can be easily compared for their relative requirements for time and space. For the basic operation say O_1 , which is expected to take minimum time (or space) among all the basic operations, the time (or space) complexity is assigned the number one. For any other basic operation, complexity is a positive number depending upon the expected relative requirement for time (or space) for the operation as compared to that for the operation O_1 . For other computer operations, time/space complexity may be computed from those for the basic operations. Also from these complexities, we can compute the complexities of the programs using the size of the input data as an additional parameter. For example, to multiply two $n \times n$ matrices we require n^3 multiplications and $(n^3 - n^2)$ additions.

Thus, complexity of the straight-forward method of multiplication of two $n \times n$ matrices is $n^3 \cdot \beta + (n^3 - n^2) \alpha$, where α and β are complexities of, respectively, the operations of addition and that of multiplication of two numbers. The time/space complexity of a **problem** may be defined as the time/space complexity of the program which has the least complexity among all the known programs that solve the problem. Further, a problem is said to be **polynomial time problem**, if the time complexity of the problem is some polynomial $a_0 n^k + a_1 n^{k-1} + \dots + a_i n^{k-i} + \dots + a_k$, where n is the size of the data. Similarly, **exponentially hard problem** is one for which time complexity is of the form a^n , with $a > 1$. For large n , the value of an exponential function increases at a much faster rate than the increase in the value of any given polynomial functions in n . For a given polynomial function $f(n)$ and an exponential function $g(n)$, it is always possible to find a positive integer k such that $g(n) > f(n)$ for all integers $n \geq k$. Thus, the problems requiring exponential time are considered harder than the problems requiring polynomial time. **‘Polynomial time’ is considered as reasonable amount of time**, and on the other hand, **‘exponential time’ is considered as impractical or infeasible amount of time from computational point of view**. This is why, the problems requiring exponential time are considered as **hard problems**. Also, using the fact that the complexity of a problem is the least of the complexities of its known algorithms, we can not solve an exponential time problem in polynomial time.

Comment 2, Definition 3: Role of knowledge in solving hard problems:

In view of the previous comments, no polynomial time algorithmic solution can exist for any (exponentially) hard problem. However, there are mechanisms/techniques which when used in a solution of a hard problem, though divest the solution of its step-by-step or algorithmic characteristic, yet may make it a polynomial time solution.

Use of appropriate knowledge of the problem domain has been found useful in techniques that when used, solve hard problems in polynomial time. Definition 3

declares the scope of (or the subject-matter) **AI as the study of techniques that**

exploit appropriate knowledge to solve hard problems in polynomial time. The

role of appropriate knowledge in reducing time complexity of a solution cannot be

overemphasized. **The following simple example supports this claim abundantly:**

Ms X is to meet Ms Y at her residence. Initially, let us assume that Ms X knows only that Ms Y lives in Delhi and knows nothing else about Ms Y's residence. A step-by-step or algorithmic solution to the problem may be to search the residential places, one by one, in some order, in Delhi and to stop when Ms. Y's place is located. The complexity of the algorithm, on the average, is undoubtedly very large. However, if X further knows that Y lives in some particular colony say Hauz Khas in Delhi, then search is substantially reduced by searching residential places only within Hauz Khas.

Further, if Ms X also knows the house number in Hauz Khas, then there is hardly any search required and X can *directly* reach Y's residence. Next, consider just opposite situation so far as availability of knowledge is concerned. Let us X even do not know that Y lives in Delhi. We can easily guess the plight of X when she, if follows a step-by-step method, is required to search, possibly all over the world, for the residence of Y.

The importance of (relevant) knowledge in solving difficult problems was recognised by the pioneers in the very early stages in the development of A.I. As we shall find subsequently, **major portion of A.I. is constituted of discussion of various issues about knowledge:** methods for *acquisition* of knowledge, for *representation* of knowledge, for *organisation* of knowledge, for *manipulation* of knowledge, for *maintenance* of knowledge and for *restricting search* of the problem domain by exploiting the knowledge of the domain.

1.6 DEFINITION BY BARR AND FEIGENBAUM

Next, we come to another definition of A.I. which involves *human intelligence* – a phenomenon only partially understood yet. Rather, computers and some A.I. techniques are being used in helping the psychologists in establishing their theories about intelligence and other mental processes. But this definition provides another angle to look at A.I. as the study of attempts at incorporating *intelligence*, whatever we understand of it yet, in machine. This definition, in a way, would also justify the inclusion of the word '*intelligence*' in the name '*Artificial Intelligence*' for the subject-matter of our study. **The definition, by Barr and Feigenbaum in 'The Handbook of Artificial Intelligence' [6], is as given below.**

Definition 4: Artificial Intelligence is the part of computer science concerned with designing intelligent computer systems, i.e., systems that exhibit the characteristics we associate with intelligence in human behaviour.

Discussion/Comments 2. Definition 4: *What is intelligence or intelligent behaviour in humans?* In order to have good grasp on the intent of this definition of A.I., we attempt to enumerate some known characteristics of *Intelligence*. There must be some basic mechanisms behind intelligent behaviour and some important attributes/characterises of intelligence which have defined human recognition or

understanding, because of which we are not able to describe the phenomenon of intelligence in its totality. Capturing the total essence of the phenomenon of intelligence in humans through a definition is almost impossible, as is noted by one of the leaders of A.I viz. **Patrick Winston [7]** of Massachusetts Institute of Technology (MIT), when he states **“defining intelligence usually takes a semester-long struggle, and even after that I am not sure we ever get a definition really nailed down”**. However, there are some characteristics of intelligence which are readily acceptable, some others acceptable after some thinking and still others that may be controversial. We enumerate the characteristics as considered by some A.I. writers and contributors and others. Enumeration of these characteristics here is essential because as A.I. technologists, we would study various techniques that help us in incorporating these characteristics, through computer programs, into machines, which we attempt to make intelligent according to Definition 4 of Artificial Intelligence. We give below the attributes verbatim from the respective sources.

Douglass R. Holstadter in his book: ‘Gödel Escher, Bach: An Eternal Golden Braid’ [8], which won him Pulitzer Prize and was a best-seller mentions on Page 26 of the book, the following as essential abilities for intelligence:

- to respond to situations very flexibly;
- to take advantage of fortuitous circumstances;
- to make sense out of ambiguous or contradictory messages; to recognize the relative importance of different elements of situation;
- to find similarities between situations despite differences which may separate them;
- to draw distinctions between situations despite similarities which may link them;
- to synthesize new concepts by taking old concepts and putting them together in new ways;
- to come up with ideas which are novel.

Fisher and Firschein in their book ‘Intelligence: The Eye, the Brain and the Computer’ [9] on Page 4 state that they expect an intelligent agent to be able to:

- Have mental attitudes (beliefs, desires and intentions)
- Learn (ability to acquire new knowledge)
- Solve problems, including the ability to break complex problems into simpler parts.
- Understand, including the ability to make sense out of ambiguous or contradictory information.
- Plan and predict the consequence of contemplated actions, including the ability to compare and evaluate alternatives.
- Know the limits of its (own) knowledge and abilities.
- Draw distinctions between situations despite similarities.
- Be original, synthesize new concepts and ideas, and acquire and employ analogies.
- Generalize (find a common underlying pattern in superficially distinct situations)
- Perceive and model the external world
- Understand and use language and related symbolic tools.

They further state that there are a number of human attributes that are related to the concept of intelligence, but are normally considered distinct from it:

- Awareness (consciousness)
- Aesthetic appreciation (art, music)
- Emotion (anger, sorrow, pain, pleasure, love, hate)

- Sensory acuteness
- Muscular coordination (motor skills)

Next, we discuss ‘intelligence’ from more fundamental level. The ideas explained below are based on the *Information Transfer Model* of scientific phenomena due to **Norbert Wiener** (1894-1964). **Norbert Wiener**, an intellectual prodigy and author of the famous book entitled *Cybernetics* [14], suggested the *Transfer of Information* model to be a better model than the prevailing model based on *Transfer of Energy* for explanation of a number of scientific phenomena. Through the Wiener’s theory, a new discipline was born, also, called *Cybernetics*

However, our discussion is mainly based on ideas explained in the book ‘Beyond Information’ by Tom Stonier [10]: According to the ideas explained in Stonier, there are four **fundamental properties of the universe viz. energy, matter, information and evolution (or change)**. The cardinality of information in the universal scheme of things can be judged from the following argument: All the entities from down to nucleons to the whole of the universe, each is known to us as *an organised system* of simpler objects, e.g., fundamental particles organise into nucleolus, nucleolus organise to form atomic nuclei, which alongwith electrons and protons organise into atoms and so on. Molecules, polymers, membranes, organs, living beings, societies, planets, planetary systems, galaxies ... and finally the whole universe, each is known as an organised system of some simpler objects. An organisation builds upon pre-existing organisations. **Thus an organised system is recursively obtained (or defined) as an interdependent assembly of elements and/or organised systems. And it is ‘information’** what is exchanged between components of an organised system to effect their interdependence and *to maintain the integrity of the system as long as the system survives against the fourth fundamental property of the universe, i.e., evolution or change*. Gravitational pull, now an established entity, is just an information processing activity. **Thus ‘information’ is no more or no less an abstract concept than ‘energy’ or ‘matter’.** **What mass is to matter and the heat is to energy, so is organisation to information.** Each of the former is a visible and measurable form of the corresponding latter. More the mass, more the matter in a system; more the heat, more the capacity to do work, i.e., energy in the system; similarly higher *the degree* (or more the complexity) *of the organization* (in terms of underlying organizations of the components and their components and so on, and in terms of the number and levels of interactions and relations between components at a particular level) higher is the *information content* of the system.

The relation between information and organisation and the characteristic difference between the two is exactly what **is the relation and characteristic difference between a number and a numeral**. A number is an *abstract* concept, whereas a numeral is its *physical manifestation or representation*. A number may have many representations and even may use many mediums for representations or manifestations. In the form **of, writing on the paper**, as patterns of ink dots on a piece of paper, the same number may be represented as 7 in decimal, 111 in binary, and even $4 + 2 + 1$ again in decimal. In computer’s memory, the same number is represented with the help of electronic components, a different medium, and not as shapes composed of ink-dots. In human brain the same number is represented, possibly, as some neural net.

Summarising, a number is a concept which needs a medium for its manifestation or physical representation for the purpose of conveying, or transformation. This representation is called a *numeral*. But it should be clear that when we say that ‘*I need two books*’, the word ‘two’ is not just the sequence of three letters viz ‘t’, ‘w’ and ‘o’ i.e., the representation, which is intended to be conveyed but just it is the *abstract* number which is intended to be conveyed. Because of the tangibility or

perceptual ‘visibility’ of the representation, we always use the representation for various purposes like ‘applying some operations’ or for conveying, but it is not the representations, but the instances of the idea or concept (of number) which are intended to be transformed or conveyed.

Similarly, information is a concept and an organisation is its representation, i.e., physical manifestation. For the purpose of applying operations (like refining information, adding information etc) or for conveying information we use organisation (as patterns of ink dots on paper or as neural net in brain etc). Then, we manipulate the organisation or representation for applying operations on information (operations again are abstract, whereas manipulations are their physical realizations). Also, we communicate the organization for conveying information (communication is physical realisation of conveying). As in the case of number, information’s representation may be through various organisations on various type of media such as patterns of ink dots on paper, neural nets in brain, or on flip-flops in electronic memory. For example, the information content of the organisation in the form of pattern of inkdots in the sentence ‘Heat is a form of energy’ is stored in the brain as an organization in the form of a Neural Network etc.

Remark 1: We have already mentioned that an atom is an organised system and so are organ’s in the human body and so are the galaxies in the universe. Also every organized system contains information. Hence, as we say ‘God gave the numbers’ so we can say ‘God created information’ **and information is not just a product of human mental activities.**

Remark 2: Information organises not only matter and energy but itself as well. Evolution leads to discontinuities, i.e., to something which is *qualitatively different* from the earlier existing entities. And intelligence is the phenomenon which has evolved out of information but which is qualitatively different from information.

Remark 3: Intelligence, being an outcome of evolution from information over a period spanning back almost upto the Big Bang, **must be a spectrum of phenomena and can not be an all or nothing affair.** Further, intelligence can not be a single-dimensional phenomenon. The veracity of this claim can be judged by analogy with evolution of matter and energy into myriad forms differing from each other in almost innumerable ways.

However, in order to draw a sort of fuzzy boundary between intelligent organisations or systems and the other systems, let us consider the case of living matter. Matter evolved from subatomic particles to atoms, molecules and so in potentially infinite number of different material objects. Out of these materials, there is a large number of types of objects (*for example, human beings*) for which we can say with surety that these are living matter and again for a large number of types of material objects (for example soil), we can say with surety that these are not living matters. Of course, there may still be large number of objects which may not be distinctly characterised as either. How we decide living or non-living is based on a finite collection of attributes of matter and degree of each such attribute.

In the similar manner, we consider a finite set of attributes and degrees for each attribute for organizations, i.e., information processing systems, which allow us to categorise systems as intelligent or otherwise in such a way that the systems which are generally considered as intelligent are categorised as intelligent and further whatever systems are generally considered as non-intelligent are categorized as non-intelligent. As evolution has taken over billions of years, hence divergence among information processing systems intelligence-wise must be potentially infinite. Thus any categorization based on only finite number of attributes would always be incomplete

and leave large number of cases ‘uncategorisable’. **To begin with, we start with a working definition of intelligence and then later expand on it:**

Intelligence is a property of advanced information processing systems, which not only engage in information processing, but are able to analyse their dynamically changing environment and to respond to it in such a way that:

- i) survivability of the system is enhanced**
- ii) its reproducibility is enhanced (reproducibility is sort of self propagation through another system)**
- iii) if the system is goal-oriented, then achievability of goal is enhanced.**

In stead of attempting to categorize most of the information processing systems as either intelligent or non-intelligent, if we are interested in their relative merit as intelligent systems, then the following principle of intelligence may be useful. **The principle quoted in Stonier [10] states: The intelligence exhibited by a system may, at least in theory, be measured as a ratio, or quotient, of the ability of a system to control its environment, versus the tendency of the system to be controlled by the environment.**

The above principle fits best, at least, in the limiting cases: At one extreme is a cube of sugar dissolving in a cup of tea. Although highly organised, the cube is totally controlled by environmental elements and hence, according to the above principle, it has zero intelligence. This is exactly what we also feel. On the other extreme is technologically advanced human society which can divert the waters of rivers to irrigate plains to provide an assured supply of food to its population. Thus intelligence measure of a technologically advanced society as a whole is, according to the above principle, quite high. This conclusion of the above principle is in consonance with what we also feel.

Below we include some more attributes and/or definitions of intelligence by leading computer scientists and A.I. researchers. The purpose is to be aware of as many facts as possible of yet-to-be completely understood phenomena of intelligence. Only then, it may be possible to design and develop really intelligent programs to solve those hard problems which are so far not amenable to computer solutions. Also, in this process, we are providing a list of attributes, against which an A.I. engineer can test their products for the quality of their product as intelligent one.

Hofstadter [8] on Page 37 says: It is an inherent property of intelligence that it can jump out of the task which it is performing and survey what it has done. **Self-evaluation and self-criticism are part of intelligent behaviour.**

Fishler and Firschein [9] on Page 4 state: Intelligence involves learning capability and goal-oriented behaviour. Additional attributes of intelligence include reasoning, common-sense, planning, perception, creativity, memory retention & recall.

Shanks [11] on Page 49 observes: The simplest and perhaps safest definition of intelligence is the ability to react to something new in a non-programmed way. The ability to be surprised or to think for oneself is really what we mean by intelligence.

In order to explain the concept of A.I. through ‘Definition 4’, we discussed the concept of intelligence itself as a phenomenon. Next, we quote another definition of A.I. again based on the concept of intelligence and given but from engineering point of view by another pioneer in the field, viz Shalkoff, a Professor of Electrical Engineering.

1.7 DEFINITION BY SHALKOFF

Definition 5: Shalkoff [13] says: ‘Perhaps broadest definition is that AI is a field of study that seeks to explain and emulate intelligent behaviour in terms of computational processes’.

Comments 1, Definition 5: According to the above definition, AI is partly *scientific* in nature because it seeks to ‘*explain*’ the phenomena of intelligence, and partly **engineering** because it seeks to ‘*emulate*’ intelligent behaviour through computational processes, i.e., by generating representations (of knowledge) and development of programs that automatically (autonomously) solve problems, so far solved by only intelligent humans beings.

In view of the fact that A.I. is partly an engineering discipline according to the above definition, let us recall what is meant by the concept *engineering*.

Engineering may be thought of as the application of science and mathematics by which properties of matter and sources of energy in nature are made useful (*meeting some requirements and according to some specifications*) to man in structures, machines, products, systems and processes etc.

Again, in the light of the definition of **Engineering** given above, a part of the definition by Shalkoff may be paraphrased as ‘...*through application of A.I., products are obtained that exhibit intelligent behaviour....*’ This paraphrased part of the definition by Shalkoff raises another issue: *How to judge/evaluate whether a product obtained through an application of A.I., is actually intelligent.*

The issue of testing an A.I. product as intelligent product was considered by the pioneers themselves including Alan Turing, the most well known name among the pioneers. In honour of Turing, the most prestigious award for contributions to the field of computer science, has been instituted and is given annually.

Turing suggested a test, which is well known as **Turing Test**, for testing whether a product has intelligence. An outline of the Turing test is given below.

For the purpose of the test, there are three rooms. In one of the rooms is a computer system claimed to have imbedded intelligence. In the other two rooms, two persons are sitting, one in each room. The role of one of the persons, let us call **A**, is to put questions to the computer and to the other person to be called **B**, without knowing to whom a particular question is being directed, and, of course, with the specific purpose of identifying the computer. On the other hand, the computer would answer in such a way that its identity is not revealed to **A**.

The communication among the three is only through computer terminals so that identity of the computer or the person **B** can be known only on the basis of quality of responses as intelligent or otherwise, and not just on the basis of other human or machine characteristics. If **A** is not able to know the identity of the computer, then computer is intelligent. More appropriately, if the computer is able to conceal its identity from **A**, then the computer is intelligent.

We may note here that, in order to be called intelligent, the computer should be clever enough not to give answer too quickly, at least not within a fraction of a second, even if it can, say, to a question involving finding of the product of two numbers each of more than 20 digits.

Objections to Turing Test: There have been a number of objections to the Turing test as a test of intelligence of a machine. One of the most well known objections is called **Chinese Room Test** proposed by John Searle. The essence of the Chinese Room Test, that we are going to explain below, is that convincing successfully by a system, say **A**, of possessing qualities of another system, say **B**, does not imply that the system **A** actually possesses the qualities of **B**. For example, the capability of convincing others by a male human of being a woman, does not give the male the quality of bearing a child like a woman.

The scenario for the **Chinese Room Test** consists of a single room with two windows. In the room a scholar on Shakespeare, knowing English, but not knowing Chinese, is sitting with a sort of encyclopedia on Shakespeare. The encyclopedia is printed in such a way that for each pair of facing pages, one page is written in Chinese characters and the other page is translation in English of the contents of the facing page in Chinese. Through one of the windows questions on Shakespeare's literature in Chinese characters are sent to the person sitting inside. The person looks through the encyclopedia and on finding in the encyclopedia the exact copy of the sequence of characters sent in, reads its translation in English, thinks of its answer and writes the answer in English for his/her own understanding, finds the corresponding sequence of Chinese characters in the encyclopedia, and sends the sequence of Chinese characters through the other window. Now, Searle says that, though the scholar successfully behaves as if s/he knows Chinese, but, as per assumption it is not so. Just from the fact that a system is able to simulate a quality, it can not be inferred that the system possesses the quality.

1.8 SUMMARY

This is an introductory unit to the course. The unit gives a bird's eye view of the whole of the course of *Artificial Intelligence*. The approach, in the unit, is to start with a definition by some pioneer in A.I. In the process of discussion of the definition, a number of relevant new concepts are gradually built up and discussed.

In **Section 0.3**, we discuss definition of A.I., as given by Eliane Rich. **It states: Artificial Intelligence is the study of how to make computers do things, at which, at the moment, people are better.**

In this context, it was discussed that human beings are still better than computers in the problem areas, which require parallel processing and simultaneous availability of information.

According to the next definition of A.I., as given by Buchamin & Shortliffe:
AI is the branch of computer Science that deals with *symbolic* rather than numeric processing and *non-algorithmic* methods including the *rules of thumb* or *heuristics* in stead of *algorithms* as *techniques* for solving problems.

In **Section 0.4**, we discuss the differences (i) between number and symbol, (ii) between algorithmic and non-algorithmic methods of solving problems.

In the **Section 0.5**, another definition by Eliane Rich, as given below, is discussed:
Artificial Intelligence is the study of techniques for solving exponentially hard problems in polynomial time exploiting knowledge about the problem domain.

In context of this definition, we discuss the difference between 'exponentially hard problems' versus 'polynomial time' problem.

In **section 0.6**, we discuss the following definition of A.I. by Barr & Feigenbaum:
Artificial Intelligence is the part of computer science concerned with designing

intelligent computer systems, i.e., systems that exhibit the characteristics we associate with intelligence in human behaviour.

In context of this definition, we discuss various characteristics of human intelligence. In the process, we discuss, relation between information and organisation and relation between information and intelligence.

Finally, in Section 0.7, we discuss a definition of A.I by Shalkoff, an engineer. According to this definition, A.I. is partly an engineering and partly a scientific discipline. As an engineering discipline A.I. is the study of designing and developing intelligent machines. In context of testing whether a machine is intelligent, we discuss Turing test and its criticism.

1.9 FURTHER READINGS/REFERENCES

1. Rich E. & Knight K. (1991). *Artificial Intelligence*. Tata McGraw-Hill Publishing Company Limited
2. Buchanan B.G. & Shortliffe E.H. eds. (1984), *Rule-Based Expert Systems*. Addison-Wesley
3. Brady J.M. (1977). *The Theory of Computer Science*. Chapman and Hall.
4. Lewis H.R. & Papadimitriou C.H. (1981). *Elements of Theory of Computation* Prentice-Hall International Editions.
5. Hopcroft J.E. & Ullman J.D. (1987). *Introduction to Automata Theory, Languages and Computation*. Narosa Publishing House
6. Barr A. & Feigenbaum E.A. (1981-82): *The Handbook of Artificial Intelligence Vol.1*, Kaufman
7. Winston P.H. & Prendergast K.A (1984): "Perspective in the AI Business", eds. MIT Press
8. Hofstadter D.R. (1979): *Gödel, Escher, Bach: An Eternal Golden Braid* Penguin Books.
9. Fischler M.A. & Firschein O. (1987). *Intelligence: The Eye, the Brain, and the Computer*. Addison-Wesley Publishing Company
10. Stonier T. (1992). *Beyond Information*. Springer-Verlag.
11. Schank R.C. with Childers P.G. (1984). *The Cognitive Computer on Language Learning and Artificial Intelligence*. Addison-Wesley Publishing Company
12. Boden M. (1998). *The Computer can act as a Brain stormer*, The Times of India, New Delhi dated March 06, 1998.
13. Shalkoff R.J. (1990). *Artificial Intelligence: An Engineering Approach* McGraw-Hill International.
14. Wiener N. (1948). *Cybernetics*. Wiley, New York.
15. Boden M. (1990). *The Philosophy of Artificial Intelligence*. Oxford University Press.

UNIT 2 THE PROPOSITIONAL LOGIC

Structure	Page Nos.
2.0 Introduction	21
2.1 Objectives	23
2.2 Logical Study of Valid and Sound Arguments	23
2.3 Non-Logical Operators	25
2.4 Syntax of Propositional Logic	26
2.5 Semantics/Meaning in Propositional Logic	27
2.6 Interpretations of Formulas	29
2.7 Validity and Inconsistency of Propositions	30
2.8 Equivalent forms in the Propositional Logic (PL)	32
2.9 Normal Forms	33
2.10 Logical Deduction	35
2.11 Applications	37
2.12 Summary	38
2.13 Solutions/Answers	38
2.14 Further/Readings	43

2.0 INTRODUCTION

Symbolic logic may be thought of as a *formal language* for representing facts about objects and relationships between objects of a problem domain **alongwith** a precise *inferencing mechanism* for reasoning and deduction. An **inferencing mechanism** derives the knowledge, which is not explicitly/directly available in the knowledge base, but can be logically inferred from what is given in the knowledge base.

The reason why the subject-matter of the study is called **Symbolic Logic** is that **symbols** are used to denote facts about objects of the domain and relationships between these objects. Then the **symbolic representations** *and not the original facts and relationships* are manipulated in order to make conclusions or to solve problems.

Also, we mentioned that a Symbolic Logic, apart from having other characteristics, is a formal language. As a formal language, there must be clearly stated unambiguous rules for defining various constituents or constructs, viz. alphabet set, words, phrases, sentences etc. of the language and also for associating meaning to each of these constituents.

The study of **Symbolic Logic** is significant, specially, for academic pursuits, in view of the fact that it is not only **descriptive** (*i.e., it tells how the human beings reason*) but it is also **normative** (*i.e., it tells how the human beings should reason*).

In this unit, we shall first study the simplest form of symbolic logic, viz, the *Propositional Logic* (PL). In the next unit, we consider a more general form of logic called the *First-Order Predicate Logic* (FOPL). Subsequently, we shall consider other symbolic systems including Fuzzy systems and some Non-monotonic systems.

In the *propositional logic*, we are interested in *declarative* sentences, i.e., sentences that can be either true or false, but not both. Any such declarative sentence is called a **proposition or a statement**. For example

- (i) The proposition: “*The sun rises in the west*,” is False,
- (ii) The proposition: “*Sugar is sweet*,” is True, and

- (iii) The truth of the proposition: “*Ram has a Ph. D degree.*” depends upon whether Ram is actually a Ph. D or not.
Though, at present, it may not be known whether the statement is True or False, yet it is sure that the sentence is either True or False and not both True and False simultaneously.

For a given declarative sentence, its being ‘True’ or ‘False’ is called its *Truth-value*. Thus, truth-value of (i) above is False and that of (ii) is True.

On the other hand, none of the following sentences can be assigned a truth-value, and hence none of these, is a statement or a proposition:

- (i) Who was the first Prime Minister of India? (*Interrogative sentence*)
- (ii) Please, give me that book. (*Imperative sentence*)
- (iii) Ram must exercise regularly. (*Imperative, rather Deontic*)
- (iv) Hurrah! We have won the trophy. (*Exclamatory sentence*)

In propositional logic, as mentioned earlier also, symbols are used to denote propositions. For instance, we may denote the propositions discussed above as follows:

P : The sun rises in the west,
Q : Sugar is sweet,
R : Ram has a Ph.D. degree.

The symbols, such as P, Q, and R, that are used to denote propositions, are called **atomic formulas, or atoms**. As discussed earlier, in this case, the truth-value of P is False, the truth-value of Q is True and the truth-value of R, though not known yet, is exactly one of ‘True’ or ‘False’, depending on whether Ram is actually a Ph. D or not.

At this stage, it may be noted that once symbols are used in place of given statements in, say, English, then the propositional system, and, in general, a symbolic system is aware **only** of symbolic representations, and the associated truth values. The system operate only on these representations. And, except for possible final translation, is **not aware** of the original statements, generally given in some natural language, say, English.

We can build, from atoms, *more complex propositions*, sometimes called **compound propositions**, by using logical connectives.

Examples of such propositions are:

- (i) *Sun rises in the east **and** the sky is clear*, and
- (ii) *If it is hot **then** it shall rain*.

The logical connectives in the above two propositions are “*and*” and “*if...then*”. In the propositional logic, **five logical operators or connectives**, viz., \sim (not), \wedge (and), \vee (or), \rightarrow (*if... then*), and \leftrightarrow (*if and only if*), are used. These five logical connectives can be used to build compound propositions from given atomic formulas. More generally, they can be used to construct more complicated compound propositions from compound propositions by applying the connectives repeatedly. For example, **if** each of the letters P, Q, C is used as a symbol for the corresponding statement, as follows:

P: The wind speed is high.
Q: Temperature is low.
C: One feels comfortable.

then the sentence:

“If the wind speed is high and the temperature is low, then one does not feel comfortable”

may be represented by the formula $((P \wedge Q) \rightarrow (\sim C))$. Thus, a compound proposition can express a complex idea. In the propositional logic, an expression that represents a proposition, such as P , or a compound proposition, such as $((P \wedge Q) \rightarrow (\sim C))$, is called a *well-formed formula*.

2.1 OBJECTIVES

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

- tell about what is Logic, Symbolic Logic, and Propositional Logic (PL); further, about why we study each of these; and about some detailed subject matter of each of these;
- tell the difference between a Proposition/Statement, which forms the basis of PL, and a sentence in a natural language;
- explain the difference between a logical operator and a non-logical operator; any symbolic logic uses only logical operators;
- explain the concept of arguments in a logical system and further should be able to explain mutual differences between a (i) valid argument (ii) sound argument (iii) invalid argument, and (iv) unsound argument;
- differentiate between an expression that is a well-formed formula **wff** of PL and an expression which is not a wff.;
- find the truth-value, or meaning, of a wff of PL and should be able to explain how the truth value of a wff is obtained from the truth values of atomic wffs.
- explain the difference between various types of wffs, viz, valid wff; consistent wff, invalid wff and inconsistent wff;
- explain about the various tools, like truth table, logical deduction and reduction to normal forms that are used to establish validity/invalidity of arguments, and further should be able to use these tools for the purpose, and
- use the tools and techniques of PL in solving problems that can be solved within a PL system.

2.2 LOGICAL STUDY OF VALID AND SOUND ARGUMENTS

Logic is the analysis and appraisal of arguments.

An **argument** is a set of statements consisting of a finite number of premises, i.e., assumed statements and a conclusion.

Valid Argument: A valid argument is one in which it would be contradictory for the premises to be true but the conclusion false.

In logical studies we are interested in valid arguments.

Example of Valid Argument

- (i) If you overslept, you will be late
 - (ii) You are not late.
- \therefore you did not oversleep.

Example of Invalid argument

- (i) If you overslept, you will be late
- (ii) You did not oversleep
 \therefore you are not late

(This argument is invalid, because despite not having overslept, one may be late because of some other engagements or laziness.)

Another Invalid Argument

- (i) If we are close to the top of Mt. Everest then we have magnificent view.
- (ii) We are having a magnificent view.
Therefore,
- (iii) We are the near the top of Mt. Everest.

(This argument is invalid, because, we may have a magnificent view even if we are not close to the top of Mt. Everest. The two given statements do not falsify this claim)

How to establish logical validity/invalidity of an argument

We have already discussed invalidity of some arguments, but invalidity above was based on our *intuition*. However, intuition may also lead us to incorrect conclusion. To be sure about the validity of our argument, we need some formal method. In Section 1.5, we discuss how a Truth table (*a formal tool*) can be used to establish the validity/invalidity of an argument.

Sound Argument

We may note that, in the case of a *valid* argument, it is **not required** that the premises/axioms or assumed statements must be True. The *assumptions may not be True*, and still the *argument may be valid*. For example, **the following argument is valid, but its premises and conclusion both are false**:

Premise 1: If moon is made of green cheese
Then $2 + 2 = 5$

Premise 2: Moon is made of green cheese
(*False premise*)

From Premise 1 and Premise 2, by applying Modus Ponens, we conclude through **valid** argument that $2 + 2 = 5$ (*which is False*).

However, in order to solve problems of everyday life, we need generally to restrict to *only true premises and valid arguments*. Then such an argument is called **sound argument**.

Sound Argument: is an argument that is valid and has true premises.

- (i) If you are reading this, then
you are not illiterate
- (ii) You are reading this (*true premise*)
You are not illiterate (*sound conclusion*)

Example of valid but not sound argument with correct conclusion.

- (i) If moon is made of green cheese then $2 + 2 = 4$
- (ii) Moon is made of green cheese (*False premise*)
To conclude $2 + 2 = 4$ (*correct*) makes the argument a Valid Argument

Example of Invalid Argument

I (i) If you overslept, you are late.

(ii) you are late.

Therefore, you overslept.

II (i) If you are in Delhi, you are in India.

You are in India.

Therefore, you are in Delhi (*invalid argument, though conclusion may be True*)

2.3 NON-LOGICAL OPERATORS

One of reason why special **symbols**:

\wedge \vee \sim \leftrightarrow \rightarrow

are used in symbolic logic in stead of the corresponding natural languages **words**:

and, or, not, if... Then, if and only if, is that the **words** may have different meaning in different contexts. For example, the use of the word **and** in one sentences has different connotation or meaning from the use in others in the following:

(i) Ram **and** Mohan are good hockey players.

(*the statement can be equivalently broken into two statements:*

(i) Ram is a good hockey player (ii) Mohan is a good hockey player)

(ii) Ram **and** Mohan are good friends.

(*though the word **and** joins two words Ram & Mohan, but can not be equivalently broken into two statements viz. (i) Ram is a friend (ii) Mohan is a friend*)

(iii) Mohan drove a car to reach home, met an accident **and** got slightly injured.

(*Here, the use of the word 'and' is not in a logical sense, but, it is in temporal sense of 'and then' because statement (iii) has different sense from the statement given in (iv) below*)

(iv) Mohan met an accident, got slightly injured **and** drove a car to reach home.

Thus from the above statements, it can be seen that the natural language word **and** may have many senses, both logical and non-logical. Similarly, the words **since**, **hence** and **because** are frequently used in arguments to establish some facts. But as shown from the following two arguments, their use in logical arguments is **risky** in the sense that some of the arguments involving any of these words may lead to **incorrect** conclusions:

Argument (1): Using the word *because*, we get correct conclusion from True statements.

Let

P: Dr. Man Mohan Singh was Prime Minister of
India in the year 2006 (*True statement*)

Q: Congress party and its allies commanded majority in Indian Parliament in the year
2006 (*True statement*)

Then the following statement:

P because Q (*True statement/conclusion*)

Thus, by using the connective *because* we get a correct/True conclusion from two True statements viz. P and Q.

Argument (2)

In the following using the word, because, we get incorrect/false conclusion from True statements

Let

P: Dr. Man Mohan Singh was Prime Minister of India in the year 2006 (True statement)

R: Chirapoonji, a town in north-east India, received maximum average rainfall in the world during 1901-2000. (True statement)

However to say

P because R, i.e., to say

Dr. Man Mohan Singe was Prime Minster of India in 2006, because Chirapoonji, a town in north-east India, received maximum average rainfall in the world during 1901-2000.

is at least **incorrect**, if not ludicrous.

Thus from two True statements, P and R and by using connective 'because', in this case, the conclusion is *incorrect*.

Thus, by using connective **because**, in one argument we get a correct conclusion from two True statements and, on the other hand, we get an incorrect conclusion from True statements.

2.4 SYNTAX OF PROPOSITIONAL LOGIC

A Well-formed formula, or *wff* or *formula* in short, in the propositional logic is defined recursively as follows:

1. An atom is a wff.
2. If A is a wff, then $(\sim A)$ is a wff.
3. If A and B are wffs, then each of $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, and $(A \leftrightarrow B)$ is a wff.
4. Any wff is obtained only by applying the above rules.

From the above recursive definition of a wff it is not difficult to see that expression: $((P \rightarrow (Q \wedge (\sim R))) \leftrightarrow S)$ is a wff; because, to begin with, each of P, Q, $(\sim R)$ and S, by definitions is a wff. Then, by recursive application, the expression: $(Q \wedge (\sim R))$ is a wff. Again, by another recursive application, the expression: $(P \rightarrow (Q \wedge (\sim R)))$ is a wff. And, finally the expression given initially is a wff.

Further, it is easy to see that according to the recursive definition of a wff, each of the expressions: $(P \rightarrow (Q \wedge))$ and $(P (Q \wedge R))$ is **not** a wff.

Some pairs of parentheses may be dropped, for simplification. For example, $A \vee B$ and $A \rightarrow B$ respectively may be used in stead of the given wffs $(A \vee B)$ and $(A \rightarrow B)$, respectively. We can omit the use of parentheses by assigning *priorities in increasing order* to the connectives as follows:

$\leftrightarrow, \rightarrow, \vee, \wedge, \sim$.

Thus, ' \leftrightarrow ' has least priority and ' \sim ' has highest priority. Further, if in an expression, there are no parentheses and two connectives between three atomic formulas are used, then the operator with higher priority will be applied first and the other operator will be applied later.

For example: Let us be given the wff $P \rightarrow Q \wedge \sim R$ without parenthesis. Then among the operators appearing in wff, the operator ' \sim ' has highest priority. Therefore, $\sim R$ is replaced by $(\sim R)$. The equivalent expression becomes $P \rightarrow Q \wedge (\sim R)$. Next, out of the two operators viz ' \rightarrow ' and ' \wedge ', the operators ' \wedge ' has higher priority. Therefore, by applying parentheses appropriately, the new expression becomes $P \rightarrow (Q \wedge (\sim R))$. Finally, only one operator is left. Hence the *fully parenthesized expression* becomes $(P \rightarrow (Q \wedge (\sim R)))$

2.5 SEMANTICS/MEANING IN PROPOSITIONAL LOGIC

Next, we define the rules of finding the truth value or meaning of a wff, when truth values of the atoms appearing in the wff are known or given.

1. The wff $\sim A$ is *True* when A is *False*, and $\sim A$ is *False* when A is *true*. The wff $\sim A$ is called the **negation** of A .
2. The wff $(A \wedge B)$ is True if A and B are both True; otherwise, the wff $A \wedge B$ is False. The wff $(A \wedge B)$ is called the **conjunction** of A and B .
3. The wff $(A \vee B)$ is true if at least one of A and B is True; otherwise, $(A \vee B)$ is False. $(A \vee B)$ is called the **disjunction** of A and B .
4. The wff $(A \rightarrow B)$ is False if A is True and B is False; otherwise, $(A \rightarrow B)$ is True. The wff $(A \rightarrow B)$ is read as "If A , then B ," or " **A implies B** ." The symbol ' \rightarrow ' is called **implication**.
5. The wff $(A \leftrightarrow B)$ is True whenever A and B have the same truth values; otherwise $(A \leftrightarrow B)$ is False. The wff $(A \leftrightarrow B)$ is read as " **A if and only if B** ."

Table 1.5

	A	B	$\sim A$	$(A \wedge B)$	$(A \vee B)$	$(A \rightarrow B)$	$(A \leftrightarrow B)$
(i)	T	T	F	T	T	T	T
(ii)	T	F	F	F	T	F	F
(iii)	F	T	T	F	T	T	F
(iv)	F	F	T	F	F	T	T

The above relations can be summarized by Table 1.5 given below.

The table may be read as follows:

Let the symbol T stand for True and the symbol F stand for False. Then, *Row (i)* is interpreted as: **if** we assign T (i.e. True) to A and T to B **then** the truth values of $(\sim A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ and $(A \leftrightarrow B)$ are respectively F, T, T, T, T.

Further row (iii), for example, is interpreted as: if we assign truth-value F (False) to A and T (True) to B then truth values of $(\sim A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$ and $(A \leftrightarrow B)$ are respectively T, F, T, T, F.

This table, shall be used to evaluate the truth values of a wff in terms of the truth values of the atoms occurring in the formula.

Now, we discuss the issue, raised in Section 1.2, of how to check validity/invalidity of an argument through formal means.

Validity through Truth-Table.

(i) If I overslept, then I am late, i.e., symbolically

$$S \rightarrow L$$

(ii) I am not late, i.e., symbolically

$$\sim L$$

To conclude

(iii) I did not oversleep, i.e., symbolically

$$\sim S$$

To establish the validity/Invalidity of the argument, consider the Truth-Table

S	L	$S \rightarrow L$	$\sim L$	$\sim S$
F	F	T	T	T
F	T	T	F	T
T	F	F	T	F
T	T	T	F	F

There is only one row, viz., first row, in which both the premises viz. $S \rightarrow L$ and $\sim L$ are True. But in this case the conclusion represented by $\sim S$ is also True. Hence, the conclusion is valid.

Invalidity through Truth-Table

(i) If I overslept, then I am late

$$S \rightarrow L$$

(ii) I did not oversleep, i.e.,

$$\sim S$$

To conclude

(iii) I would not be late, i.e.,

$$\sim L \text{ (invalid conclusion)}$$

S	L	$(S \rightarrow L)$	$\sim S$	$\sim L$
F	F	T	T	T
F	T	T	T	F
T	F	F	F	T
T	T	T	F	F

The invalidity of the argument is established, because, for validity last column must contain True in those rows for which all axioms/premises are True. But in the second row both $S \rightarrow L$ and $\sim S$ are True but $\sim L$ is False

Ex. 1 Express the following statements in Propositional Logic.

- If he campaigns hard, he will be elected.
- If the humidity is high, it will rain either today or tomorrow.
- Cancer will not be cured unless its cause is determined and a new drug for cancer is found.
- It requires courage and skills to climb a mountain.

Ex. 2: Let

P : He needs a doctor,

R : He has an accident,

U : He is injured.

Q : He needs a lawyer,

S : He is sick,

State the following formulas in English.

- | | | | |
|----|--|----|---|
| a) | $(S \rightarrow P) \wedge (R \rightarrow Q)$ | b) | $P \rightarrow (S \vee U)$ |
| c) | $(P \wedge Q) \rightarrow R$ | d) | $(P \wedge Q) \leftrightarrow (S \wedge U)$ |
-

2.6 INTERPRETATIONS OF FORMULAS

In order to find the truth value of a given formula G , the truth values for the *atoms* of the formula are either given or assumed. The set of initially given/assumed values of all the atomic formulas occurring in a formula say G , is called an **interpretation** of the formula G . Suppose that A and B are two atoms and that the truth values of A and B are T and F respectively. Then, according to third row of Table 1.5, when A is F and B is T we find that the truth values of $(\sim A)$, $(A \wedge B)$, $(A \vee B)$, $(A \rightarrow B)$, and $(A \leftrightarrow B)$ are T , F , T , T and F , respectively. By developing a Truth-table of a(ny) formula, its truth value can be evaluated in terms of its interpretation, i.e., in terms of the truth values associated with the constituent atoms.

Example

Consider the formula

$$G : ((A \wedge B) \rightarrow (R \leftrightarrow (\sim S))).$$

(Please note that the string, in this case G , before the symbol ':', is the name of the formula which is the name of the string of symbols after ':'. Thus, G is the name of the formula $((A \wedge B) \rightarrow (R \leftrightarrow (\sim S)))$).

The atoms in this formula are A , B , R and S . Suppose the truth values of A , B , R , and S are given as T , F , T and T , respectively. **Then (in the following and elsewhere also, if there is no possibility of confusion, we use T for 'True' and F for 'False'.)**

- $(A \wedge B)$ is F since B is F ;
- $(\sim S)$ is F since S is T ;
- $(R \leftrightarrow (\sim S))$ is F since R is T and $(\sim S)$ is F ; and hence,
- $(A \wedge B) \rightarrow (R \leftrightarrow (\sim S))$ is T since $(A \wedge B)$ is F (and $(R \leftrightarrow (\sim S))$ is F , which does not matter).

Note: In view of the fact that when $(A \wedge B)$ is F , the truth-value of

$$(A \wedge B) \rightarrow \text{Any Formula}$$

must be T and, hence, we need not compute the value of $(R \leftrightarrow (\sim S))$.

Therefore, the formula G is T if A , B , R , and S are assigned truth values T , F , T and T , respectively.

The assignment of the truth values T , F , T , T to A , B , R , S , respectively, is called an **interpretation of the formula G** . Since, each one of A , B , R , and S can be assigned one of the two values, viz., either T or F , there are $2^4 = 16$ possible interpretations of the formula G . In Table 1.6, we give the truth values of the formula G under all these 16 interpretations.

The above procedure may be repeated to find truth value of any formula from any interpretation, i.e., from any assignment to the atomic formulas occurring in the given formula.

Table 1.6 Truth Table of $(A \wedge B \rightarrow (R \leftrightarrow (\sim S)))$

A	B	R	S	$\sim S$	$(A \wedge B)$	$(R \leftrightarrow (\sim S))$	$(A \wedge B) \rightarrow (R \leftrightarrow (\sim S))$
T	T	T	T	F	T	F	F
T	T	T	F	T	T	T	T
T	T	F	T	F	T	T	T
T	T	F	F	T	T	F	F
T	F	T	T	F	F	F	T
T	F	T	F	T	F	T	T
T	F	F	T	F	F	T	T
T	F	F	F	T	F	F	T
F	T	T	T	F	F	F	T
F	T	T	F	T	F	T	T
F	T	F	T	F	F	T	T
F	T	F	F	T	F	F	T
F	F	T	T	F	F	F	T
F	F	T	F	T	F	T	T
F	F	F	T	F	F	T	T
F	F	F	F	T	F	F	T

A table, such as given above, that displays the truth values of a formula G for *all possible assignments of truth values to atoms occurring in G* is called a **Truth table of G** .

NOTATION: If A_1, \dots, A_n are all the atoms in a formula, it may be more convenient to represent an *interpretation* by a set (m_1, \dots, m_n) , where m_i is either A_i or $\sim A_i$. m_i is written as A_i if T is assigned to A_i . But m_i is written as $\sim A_i$ if F is assigned to A_i .

For example, the set $\{A, \sim B, \sim R, S\}$ represents an interpretation of a formula in which A, B, R, and S are the only atoms and which are, respectively, assigned T, F, F, and T. We will use the notation throughout.

Ex. 3: Construct a truth table for the formula.

P: $(\sim A \vee B) \wedge (\sim (A \wedge \sim B))$

2.7 VALIDITY AND INCONSISTENCY OF PROPOSITIONS

It may be noted that in Section 1.2, we discussed the concept of valid **Argument**. Here, we study **formulas** or propositions. Next, we shall consider wff that are **true under all** possible interpretations and wff that are **false under all** possible interpretations.

Example

Let us consider the wff

$$G : (((A \rightarrow B) \wedge A) \rightarrow B).$$

The formula G has $2^2 = 4$ possible interpretations in view of the fact it has two atoms viz A and B. It can be easily seen from the following table that the wff G is True under all its interpretations. **Such as a wff which is True under all interpretation is called a valid formula (or a tautology).**

Truth Table of $((A \rightarrow B) \wedge A) \rightarrow B$

A	B	$(A \rightarrow B)$	$(A \rightarrow B) \wedge A$	$((A \rightarrow B) \wedge A) \rightarrow B$
T	T	T	T	T
T	F	F	F	T
F	T	T	F	T
F	F	T	F	T

Consider another formula

$$G : ((A \rightarrow B) \wedge (A \wedge \sim B))$$

The truth table of the formula G given below shows that G is False under all its interpretations. Such a formula which is False under all interpretations is called an **inconsistent formula (or a contradiction)**.

Truth Table of $(A \rightarrow B) \wedge (A \wedge \sim B)$

A	B	$\sim B$	$(A \rightarrow B)$	$(A \wedge \sim B)$	$((A \rightarrow B) \wedge (A \wedge \sim B))$
T	T	F	T	F	F
T	F	T	F	T	F
F	T	F	T	F	F
F	F	T	T	F	F

Next, we formally define the concepts discussed above.

Definition: A formula is said to be **valid** if and only if it is **true under all** its interpretations. A formula is said to be **invalid** if and only if it is *not true under at least one* interpretation. A valid formula is also called a **Tautology**. A formula is **invalid** if there is **at least one** interpretation for which the formula has a truth value False.

Definition: A formula is said to be **inconsistent (or unsatisfiable)** if and only if it is **False under all** its interpretations. A formula is said to be **consistent or satisfiable** if and only if it is not inconsistent. In other words, a formula is *consistent if there is at least one interpretation* for which the formula has a truth value true.

From the definitions given above, it is easily seen that

- (i) A formula is *valid* if and only if its negation is *inconsistent*.
- (ii) A formula is *invalid* if and only if there is *at least one* interpretation under which the formula is *false*.
- (iii) A formula is *consistent* if and only if there is *at least one* interpretation under which the formula is *true*.
- (iv) If a formula is *valid*, then it is *consistent*, but not *vice versa*. (example given below)
- (v) If a formula is *inconsistent*, then it is *invalid*, but not *vice versa*. (example given below)

Definition: If a formula P is *True under an interpretation I*, then we say that **I satisfied P**, or P is satisfied by I. If a formula P is *False under an interpretation I*, then we say that **I falsifies P** or P is **falsified by I**.

As for an example, the formula $(A \wedge (\sim B))$ is satisfied by the interpretation $\{A, \sim B\}$ i.e., by taking A as T and B as F, but is **falsified** by the interpretation $\{A, B\}$ i.e., when A is taken as T and B is taken as T. An interpretation I that satisfies a formula P, is called a **model** of the formula F.

Examples:**(i) A Valid Formula:**

- (a) Even *True* is a wff which is always True and, hence, True is a valid formula.
- (b) $G_1: A \vee (\sim A)$ is True for all its interpretations. As G_1 has only one atom viz. A, therefore, it has only two interpretations. Let *one interpretation of G_1 be : A is True*. But then G_1 assumes the value $(\text{True} \vee (\sim \text{True})) = \text{True}$. The *other interpretation of G_1 is : A is False*. Then G_1 assumes the value $(\text{False} \vee \sim \text{False}) = \text{True}$.

(ii) Consistent (True for at least one interpretation) but not valid Formula (i.e. is invalid, i.e., False for at least one interpretation):

- (a) The simplest example of such a formula is the formula $G_2: A$. Then, for the assignment A as True, G_2 is True. Therefore G_2 is consistent. On the other hand, the interpretation of G_2 with A as False, makes G_2 false. Therefore, $G_2: A$ is not valid.
- (b) Both $G_3: A \vee B$ and $G_4: A \wedge B$ are consistent but not valid. Both G_3 and G_4 are True under the assignment A as True and B as True. On the other hand, both are False under the interpretation A as False and B as False.

(iii) Invalid (False for at least one interpretation) but not inconsistent (not False for all interpretations): Any one of the examples in (ii) above**(iv) Inconsistent formula (i.e., which is false for all interpretations)**

- (a) Even 'False' is a wff; which is always False, and hence is inconsistent.
- (b) $G_5: A \wedge (\sim A)$ is False, for all interpretations of G_5 . Actually, there are only two interpretations of G_5 . One is : A is True. The other is : A is False. In both cases G_5 is False.

It will be shown later that the proof of the validity or inconsistency of a formula is a very important problem. In the propositional logic, since the number of interpretations of a formula is finite, one can *always decide* whether or not a formula in the propositional logic is valid (inconsistent) by exhaustively examining all of its possible interpretations.

Ex. 4: For each of the following formulas, determine whether it is valid, inconsistent, consistent or some combination of these.

- (i) $E: \sim (\sim A) \rightarrow B$
 - (ii) $G: (A \rightarrow B) \rightarrow (\sim B \rightarrow \sim A)$
 - (iii) $H: (A \vee \sim A) \rightarrow (A \wedge B) \wedge (\sim A)$
 - (iv) $J: (A \wedge B) \wedge (\sim A) \rightarrow (B \vee \sim B)$
-

2.8 EQUIVALENT FORMS IN THE PROPOSITIONAL LOGIC (PL)

Definition: Logically Equivalent Formulas: Two formulas G_1 and G_2 are said to be (logically) *equivalent* if for each interpretation i.e., truth assignment to all the atoms that occur in either G_1 or G_2 ; the truth values of G_1 and G_2 are identical. In other words, for each interpretation, G_1 is True if and only if G_2 is True. And, for each interpretation, G_1 is False if and only if G_2 is False.

As will be clear later, it is often necessary to transform a formula from one form to another, especially to a *normal form*. This is accomplished by replacing a formula in the given formula by a formula *equivalent* to it and repeating this process until the desired form is obtained.

Example

We can verify that the formula **E**: $\sim (A \rightarrow B)$ is equivalent the formula **G**: $A \wedge \sim B$ by examining the following truth table. The corresponding values in the last two columns are identical.

Table Joint Truth table of $\sim (A \rightarrow B)$ and $(A \wedge \sim B)$

A	B	$\sim B$	$(A \rightarrow B)$	$\sim(A \rightarrow B)$	$A \wedge \sim B$
T	T	F	T	F	F
T	F	T	F	T	T
F	T	F	T	F	F
F	F	T	T	F	F

Solutions of problems using symbolic logic can be simplified, if we can simplify involved formulas by some equivalent simpler formulas given in table below. These equivalences can be verified by using truth tables.

Table of Equivalences of PL

(1.1)	$E \leftrightarrow G = (E \rightarrow G) \wedge (G \rightarrow E)$	
(1.2)	$E \rightarrow G = \sim E \vee G$	
(1.3)(a)	$E \vee G = G \vee E;$	(b) $E \wedge G = G \wedge E$
(1.4)(a)	$(E \vee G) \vee H = E \vee (G \vee H);$	(b) $(E \wedge G) \wedge H = E \wedge (G \wedge H)$
(1.5)(a)	$E \vee (G \wedge H) = (E \vee G) \wedge (E \vee H);$	(b) $E \wedge (G \vee H) = (E \wedge G) \vee (E \wedge H)$
(1.6)(a)	$E \vee \text{False} = E;$	(b) $E \wedge \text{True} = E$
(1.7)(a)	$E \vee \text{True} = \text{True}$	(b) $E \wedge \text{False} = \text{False}$
(1.8)(a)	$E \vee \sim E = \text{True};$	(b) $E \wedge E = E$
(1.9)	$\sim(\sim E) = E$	
(1.10)(a)	$\sim(E \vee G) = \sim E \wedge \sim G;$	(b) $\sim(E \wedge G) = \sim E \vee \sim G$

In the table given above, True denotes the fact that the wff is True under all interpretations and False denotes the wff that is False under all interpretations.

Laws (1.3a), (1.3b) are often, called **commutative laws**; (1.4a), (1.4b) **associative laws**; (1.5a), (1.5b), **distributive laws**; and (1.10a), (1.10b), **De Morgan's laws**.

2.9 NORMAL FORMS

Some Definitions: A **clause** is a disjunction of literals. For example, $(E \vee \sim F \vee \sim G)$ is a clause. But $(E \vee \sim F \wedge \sim G)$ is not a clause. A **literal** is either an atom, say A, or its negation, say $\sim A$.

Definition: A formula E is said to be in a **Conjunctive Normal Form (CNF)** if and only if E has the form $E : E_1 \wedge \dots \wedge E_n, n \geq 1$, where each of E_1, \dots, E_n is a **disjunction** of literals.

Definition: A formula E is said to be in **Disjunctive Normal Form (DNF)** if and only if E has the form $E : E_1 \vee E_2 \vee \dots \vee E_n$, where each E_i is a **conjunction** of literals.

Examples: Let A, B and C be atoms. Then $F : (\sim A \wedge B) \vee (A \wedge \sim B \wedge \sim C)$ is a formula in a disjunctive normal form.

Example: Again $G: (\sim A \vee B) \wedge (A \vee \sim B \vee \sim C)$ is a formula in Conjunctive Normal Form, because it is a conjunction of the two disjunctions of literals viz of $(\sim A \vee B)$ and $(A \vee \sim B \vee \sim C)$

Example: Each of the following is neither in CNF nor in DNF

- (i) $(\sim A \vee B) \vee (A \wedge \sim B \vee C)$
- (ii) $(A \rightarrow B) \wedge (\sim B \wedge \sim A)$

Using table of equivalent formulas given above, any valid Propositional Logic formula can be transformed into CNF as well as DNF.

The steps for conversion to DNF are as follows

Step 1: Use the equivalences to remove the logical operators ' \leftrightarrow ' and ' \rightarrow ':

- (i) $E \leftrightarrow G = (E \rightarrow G) \wedge (G \rightarrow E)$
- (ii) $E \rightarrow G = \sim E \vee G$

Step 2 Remove \sim 's, if occur consecutively more than once, using

$$(iii) \sim(\sim E) = E$$

(iv) Use De Morgan's laws to take ' \sim ' nearest to atoms

- (v) $\sim(E \vee G) = \sim E \wedge \sim G$
- (vi) $\sim(E \wedge G) = \sim E \vee \sim G$

Step 3 Use the distributive laws repeatedly

- (vii) $E \vee (G \wedge H) = (E \vee G) \wedge (E \vee H)$
- (viii) $E \wedge (G \vee H) = (E \wedge G) \vee (E \wedge H)$

Example

Obtain a disjunctive normal form for the formula $\sim(A \rightarrow (\sim B \wedge C))$.

$$\begin{aligned} \text{Consider } A \rightarrow (\sim B \wedge C) &= \sim A \vee (\sim B \wedge C) && (\text{Using } (E \rightarrow F) = (\sim E \vee F)) \\ \text{Hence, } \sim(A \rightarrow (\sim B \wedge C)) &= \sim(\sim A \vee (\sim B \wedge C)) \\ &= \sim(\sim A) \wedge \sim(\sim B \wedge C) && (\text{Using } \sim(E \vee F) = \sim E \wedge \sim F) \\ &= A \wedge (B \vee (\sim C)) && (\text{Using } \sim(\sim E) = E \text{ and } \sim(E \wedge F) = \sim E \vee \sim F) \\ &= (A \wedge B) \vee (A \wedge (\sim C)) && (\text{Using } E \wedge (F \vee G) = (E \wedge F) \vee (E \wedge G)) \end{aligned}$$

However, if we are to obtain CNF of $\sim A \rightarrow (\sim B \wedge C)$, in the last but one step, we obtain

$\sim(A \rightarrow (\sim B \wedge C)) = A \wedge (B \vee \sim C)$, which is in CNF, because, each of A and $(B \vee \sim C)$ is a disjunct.

Example: Obtain conjunctive Normal Form (CNF) for the formula: $D \rightarrow (A \rightarrow (B \wedge C))$

Consider

$$\begin{aligned}
 & D \rightarrow (A \rightarrow (B \wedge C)) \quad (\text{using } E \rightarrow F = \sim E \vee F \text{ for the inner implication}) \\
 & = D \rightarrow (\sim A \vee (B \wedge C)) \quad (\text{using } E \rightarrow F = \sim E \vee F \text{ for the outer implication}) \\
 & = \sim D \vee (\sim A \vee (B \wedge C)) \\
 & = (\sim D \vee \sim A) \vee (B \wedge C) \quad (\text{using Associative law for disjunction}) \\
 & = ((\sim D \vee \sim A \vee B) \wedge (\sim D \vee \sim A \vee C))
 \end{aligned}$$

The last line denotes the conjunctive Normal Form of $D \rightarrow (A \rightarrow (B \wedge C))$
(using distributivity of \vee over \wedge)

Note: If we stop at the last but one stop, then we obtain $(\sim D \vee \sim A) \vee (B \wedge C) = \sim D \vee \sim A \vee (B \wedge C)$ is a **Disjunctive Normal Form** for the given formula: $D \rightarrow (A \rightarrow (B \wedge C))$

Ex. 5: Transform the following into disjunctive normal forms.

$$(i) \sim (A \vee \sim B) \wedge (S \rightarrow T) \quad (ii) (A \rightarrow B) \rightarrow R$$

Ex. 6: Transform the following into conjunctive normal forms.

$$(i) (A \rightarrow B) \rightarrow R$$

$$(ii) (\sim A \wedge B) \vee (A \wedge \sim B)$$

Ex. 7: Verify each of the following pairs of equivalent formulas by transforming formulas on both sides of the sign $=$ into the same normal form.

$$(i) (A \rightarrow B) \rightarrow (A \wedge B) = (\sim A \rightarrow B) \wedge (B \rightarrow A)$$

$$(ii) A \wedge B \wedge (\sim A \vee \sim B) = \sim A \wedge \sim B \wedge (A \vee B)$$

2.10 LOGICAL DEDUCTION

Definition: A formula G is said to be a logical **consequence** of given formulas E_1, \dots, E_n (or **G is logical derivation of E_1, \dots, E_n**) if and only for any interpretation I in which $E_1 \wedge E_2 \wedge \dots \wedge E_n$ is true, for the interpretation I , G is also true. The proposition E_1, E_2, \dots, E_n are called *axioms/premises* of G .

Next, we state without proof two very useful theorems for establishing logical derivations:

Theorem 1: Given formulas E_1, \dots, E_n and a formula G , G is a logical derivation of E_1, \dots, E_n if and only if the formula $((E_1 \wedge \dots \wedge E_n) \rightarrow G)$ is valid, i.e., True for all interpretations of the formula.

Theorem 2: Given formulas E_1, \dots, E_n and a formula G , G is a logical consequence or derivation of E_1, \dots, E_n if only if the formula $(E_1 \wedge \dots \wedge E_n \wedge \sim G)$ is inconsistent, i.e., False for all interpretations of the formula.

The above two theorems are very useful. They show that proving a particular formula as a logical consequence of a finite set of formulas is equivalent to proving that a *certain single but related formula* is valid or inconsistent.

Note: Significance of the above two theorems lies in the fact that logical consequence relates **two** formulas, where as validity/inconsistency is only about **one** formula. Also, there are a number of well-known methods, including truth-table method, for

establishing inconsistency/validity of a formula. Thus, formula G logically follows from a given set of formulas, we check validity of single formula. And, for checking validity of a single formula, we already have some methods including Truth-table method.

Definition: If the formula G is a logical consequence of the formula E_1, \dots, E_n , then the single formula $((E_1 \wedge \dots \wedge E_n) \rightarrow G)$ is called a **theorem**, and G is also called the **conclusion** of the theorem.

There are at least three alternative methods of establishing formula G as a logical consequence of given formulas E_1, E_2, \dots, E_n .

According to **one of these methods**, through truth table or otherwise, it should be established that for any interpretation for which each of E_1, \dots, E_n , is true then for that interpretation G must be true.

According to **second method**, using Theorem 1, we should show that the formula:

$$(E_1 \wedge E_2 \wedge \dots \wedge E_n) \rightarrow G$$

is valid, i.e., True for each of its interpretations. Again validity can be shown either through a truth table or otherwise.

The **last of the three methods** uses Theorem 2. According to this method, in order to show, G as a logical consequence of E_1, E_2, \dots, E_n , it should be established that the formula $(E_1 \wedge E_2 \wedge \dots \wedge E_n \wedge \sim G)$ is inconsistent, i.e., is False under all its interpretations. Next, we apply these methods through an example.

Example: We are given the formulas

$$E_1 : (A \rightarrow B), E_2 : \sim B, G : \sim A$$

We are required to show that G is a logical consequence of E_1 and E_2 .

Method 1: From the following Table, it is clear that whenever $E_1: A \rightarrow B$ and $E_2: \sim B$ both are simultaneously True, (which is true only in the last row of the table) then $G: \sim A$ is also True. Hence, the proof.

A	B	$A \rightarrow B$	$\sim B$	$\sim A$
T	T	T	F	F
T	F	F	T	F
F	T	T	F	T
F	F	T	T	T

Method 2: We prove the result by showing the validity of $E_1 \wedge E_2 \rightarrow G$, i.e., of $((A \rightarrow B) \wedge \sim B) \rightarrow \sim A$ by transforming it into a conjunctive normal form.

$$\begin{aligned} (A \rightarrow B) \wedge \sim B \rightarrow \sim A &= \sim ((A \rightarrow B) \wedge \sim B) \vee \sim A \quad (\text{using } E \rightarrow F = (\sim E \vee F)) \\ &= \sim ((\sim A \vee B) \wedge \sim B) \vee \sim A \end{aligned}$$

$$\begin{aligned} &= \sim ((\sim A \wedge \sim B) \vee (B \wedge \sim B)) \vee \sim A \\ &= \sim ((\sim A \wedge \sim B) \vee \text{False}) \vee \sim A \\ &= \sim ((\sim A \wedge \sim B)) \vee \sim A \quad (\text{using De Morgan's Laws}) \\ &= (A \vee B) \vee \sim A = \\ &= (B \vee A) \vee \sim A \\ &= B \vee (A \vee \sim A) \\ &= B \vee \text{True} \end{aligned}$$

$$= \text{True (always)}$$

Thus, $((A \rightarrow B) \wedge B) \rightarrow \sim A$ is valid.

Ex. 8: Using Truth Table show that G is a logical consequence of E_1 and E_2 where $E_1 : (A \rightarrow B)$, $E_2 : \sim B$, $G : \sim A$, by establishing validity of the formula $(E_1 \wedge E_2 \rightarrow G)$.

Ex. 9: Use (i) the truth table technique (ii) reduction to DNF/CNF to show that $(A \rightarrow B) \wedge \sim B \wedge A$ is inconsistent which, in turn proves that $\sim A$ is a logical consequence of $(A \rightarrow B)$ and $\sim B$.

2.11 APPLICATIONS

Next, we discuss some of the applications of Propositional Logic.

Example

Suppose the stock prices go down if the interest rate goes up. Suppose also that most people are unhappy when stock prices go down. Assume that the interest rate goes up. Show that we can conclude that most people are unhappy.

To show the above conclusion, let us denote the statements are as follows:

A : Interest rate goes up,
S : Stock prices go down
U : Most people are unhappy

The problem has the following four statements:

- 1) If the interest rate goes up, stock prices go down.
- 2) If stock prices go down, most people are unhappy.
- 3) The interest rate goes up.
- 4) Most people are unhappy. *(to conclude)*

The above-mentioned statements are symbolised as,

- (1') $A \rightarrow S$
- (2') $S \rightarrow U$
- (3') A
- (4') U . *(to conclude)*

In order to establish the conclusion, we should show that (4') is logical consequence of (1'), (2') and (3'). For this purpose, we show that (4') is true whenever $(1') \wedge (2') \wedge (3')$ is true.

We transform $((A \rightarrow S) \wedge (S \rightarrow U) \wedge A)$ (representing $(1') \wedge (2') \wedge (3')$) into a normal form:

$$\begin{aligned} ((A \rightarrow S) \wedge (S \rightarrow U) \wedge A) &= ((\sim A \vee S) \wedge (\sim S \vee U) \wedge A) && \text{(by using } E \rightarrow F = \sim E \vee F) \\ &= (A \wedge (\sim A \vee S) \wedge (\sim S \vee U)) && \text{(by using } E \wedge F = F \wedge E, \text{ (to bring the last clause } A \text{ in the beginning)} \end{aligned}$$

$$\begin{aligned}
&= (((A \wedge \sim A) \vee (A \wedge S)) \wedge (\sim S \vee U)) \text{ (by using associative laws and then using distributivity of 'A } \wedge \text{' over the next disjunct } (\sim A \vee S)) \\
&= ((\text{False} \vee (A \wedge S)) \wedge (\sim S \vee U)) \text{ (using False } \vee E = E) \\
&= (A \wedge S) \wedge (\sim S \vee U) \\
&= (A \wedge S \wedge \sim S) \vee (A \wedge S \wedge U) \\
&= (A \wedge \text{False}) \vee (A \wedge S \wedge U) \text{ (using } A \wedge \text{False} = \text{False}) \\
&= \text{False} \vee (A \wedge S \wedge U) \\
&= A \wedge S \wedge U
\end{aligned}$$

Therefore, if $((A \rightarrow S) \wedge (S \rightarrow U) \wedge A)$ is true, then $(A \wedge S \wedge U)$ is true. Since $(A \wedge S \wedge U)$ is true then each of A, S, and U is true, we conclude that U is true. Hence, U is a logical consequence of 1), 2) and 3) given above.

Ex. 10: Given that if the Parliament refuses to enact new laws, then the strike will not be over unless it lasts more than one year and the president of the firm resigns, will the strike not be over if the Parliament refuses to act and the strike just starts?

2.12 SUMMARY

In this unit, to begin with, we discuss what is Symbolic Logic **and** why it is it is important to study it. The subject matter of symbolic logic consists of **arguments**, where an argument consists of a number of statements — one of which is called the **conclusion** and is supposed to be logically drawn from the others. Each one of the other is called a **premise**. To be more specific, the subject of Symbolic Logic is the study of how to develop tools and technique to draw correct conclusions from a given set of premisses or to verify whether a conclusion is correct or not. A conclusion is correct in the sense: Whenever all the premisses are True then conclusion is necessarily True. An argument with correct conclusion is called a **valid** argument. Next, a **sound** argument is defined as a valid argument in which premises also have to be True.
(in some world).

In this unit, we study only a specific branch of symbolic logic, viz. Propositional Logic (PL).

Next, we discuss how a **statement**, also called a well-formed formula (**wff**) and also a **Proposition**, which is the basic unit of an argument in PL, is appropriately **denoted** and how it is **interpreted**, i.e., how a wff is given meaning. The meaning of a wff in PL is only in terms of True or False. The wffs are classified as *valid, invalid, consistent and inconsistent*.

Then tools and techniques in the form of Truth-table, logical deduction, normal forms etc are discussed to test these properties of wffs and also to test validity of arguments. Finally a number of applications of these concepts, tools and techniques of PL are used to solve problems that involve logical reasoning of PL systems.

2.13 SOLUTIONS/ANSWERS

Ex. 1

- (a) Let H: He campaigns hard ; E: He will be elected
Then the statement becomes the formula:

$$H \rightarrow E$$

- (b) Let H: The Humidity is high, RTY: It will rain today
RTW: It will rain tomorrow.
Then

$$H \rightarrow RTY \vee RTW$$

- (c) Let C: Cancer will be cured
D: Cancer's cause will be determined
F: A new drug for cancer will be found
Then the statement becomes the formula:

$(\sim C) \vee (D \wedge F)$. This formula may also be written as:

$$C \rightarrow D \wedge F$$

- (d) Let C: One has courage
S: One has skill
M: One climbs mountain

Then the statement becomes the formula:

$$M \rightarrow C \wedge S$$

- Ex 2:** (a) If he is sick then he needs a doctor, but, if he has an accident then he needs a lawyer
(b) If One requires a doctor then one must be either sick or injured.
(c) If he needs both a doctor and a lawyer then he has an accident.
(d) One requires a doctor and also a lawyer if and only if one is sick and also injured.

Ex. 3:

- (i) Truth table of the formula: $P: (\sim A \vee B) \wedge (\sim (A \wedge \sim B))$ is as given below.

A	B	$\sim A$	$\sim B$	$\sim A \vee B$	$A \wedge \sim B$	$\sim (A \wedge \sim B)$	P
T	T	F	F	T	F	T	T
T	F	F	T	F	T	F	F
F	T	T	F	T	F	T	T
F	F	T	T	T	F	T	T

Ex. 4:

- (i) Consistent but not valid, because, for For B as T and A as F, the formula is T. But, for A as T and B as F the formula is F.
(ii) It can be easily that $\sim B \rightarrow \sim A$ has same truth-value as $(A \rightarrow B)$ for any interpretation. Therefore, in stead of the given formula, we may consider the formula
 $(A \rightarrow B) \rightarrow (A \rightarrow B)$
which can be further written as $P \rightarrow P$, writing $(A \rightarrow B)$ as P. Even $P \rightarrow$ can be written as $P \vee P \equiv P \equiv (A \rightarrow B)$, The last formula is F when F and A is T. The formula is T when A is F and B is T. Hence, the formula is neither valid nor inconsistent.
Therefore, the formula is consistent but not valid
(iii) For all truth assignments to A and B, L. H.S. of the formula is always T and R. H.S. is always F. Hence the formula is inconsistent, i.e., always F
(iv) The L. H. S. of the given formula is F under all interpretations. Hence, the formula is T under all interpretation. Therefore, the formula is valid.

Ex. 5: (i) Removing ' \rightarrow ', we get

$$\sim (A \vee \sim B) \wedge (\sim S \vee T)$$

Taking ' \sim ' inside we get

$$(\sim A \wedge B) \wedge (\sim S \vee T) \quad (\text{using De Morgan's Law})$$

Using distributivity of \wedge over \vee we get

$$(\sim A \wedge B \wedge \sim S) \vee (\sim A \wedge B \wedge T)$$

which is the required form

(ii) Removing outer \rightarrow we get
 $\sim (A \rightarrow B) \vee R$
 Removing the other ' \rightarrow ' we get
 $\sim (\sim A \vee B) \vee R$
 Taking \sim inside, we get
 $(A \wedge \sim B) \vee R,$
 which is the required form

Ex. 6:

(i) Using distributive law in the last formula of 5 (ii) above, we get
 $(A \vee R) \wedge (\sim B \vee R)$
 which is the required CNF

(ii) Using Left distributivity of \vee over \wedge we get
 $((\sim A \wedge B) \vee A) \wedge (\sim A \wedge B) \vee \sim B)$
 Using Right distributivity inside each pair of parentheses of \vee over \wedge we get
 $((\sim A \vee A) \wedge (B \vee A) \wedge ((\sim A \vee \sim B) \wedge (B \vee \sim B)))$
 Using $\sim A \vee A = T = B \vee \sim B$, we get
 $(T \wedge (B \vee A)) \wedge ((\sim A \vee \sim B) \wedge T)$
 which is equivalent to
 $(B \vee A) \wedge ((\sim A \vee \sim B) \wedge T) = (A \vee B) \wedge (\sim A \vee \sim B)$
 is the required CNF.

Ex. 7: (i) Consider L.H.S

Removing inner \rightarrow on L. H.S., we get
 $(\sim A \vee B) \rightarrow (A \wedge B)$
 removing the other ' \rightarrow '
 $= \sim (\sim A \vee B) \vee (A \wedge B)$
 Using De Morgan's Laws, we get
 $= (\sim (\sim A) \wedge (\sim B)) \vee (A \wedge B)$
 $= (A \wedge \sim B) \vee (A \wedge B)$ (i)
 which is in DNF

For R.H.S, removing the two implications, we get

$(\sim (\sim A) \vee B) \wedge (\sim B \vee A)$
 $= (A \vee B) \wedge (\sim B \vee A)$
(which is in CNF, but we require DNF)
 Using Left distributivity of \wedge over \vee , we get
 $= ((A \vee B) \wedge (\sim B)) \vee ((A \vee B) \wedge A)$
 Using Right distributivity of \wedge over \vee , we get
 $= ((A \wedge \sim B) \vee (B \wedge \sim B)) \vee ((A \wedge A) \vee (B \wedge A))$
 Using $B \wedge \sim B = F$ $A \wedge A = A$
 And $P \vee F = P$ we get
 $= (A \wedge \sim B) \vee (A \vee (B \wedge A))$
 $= (A \wedge \sim B) \vee (A) = (A \wedge \sim B) \vee (A \wedge T)$ (ii)
 $= (A \wedge \sim B) \vee (A \wedge (B \vee \sim B))$
 $= (A \wedge \sim B) \vee (A \wedge \sim B) \vee (A \wedge B)$
 Using $P \vee P = P$, we get
 $= (A \wedge \sim B) \vee (A \wedge B)$

(ii) **R.H.S** Applying associative laws, we get

$(\sim A \wedge \sim B) \wedge (A \vee B)$
 Using left distributivity of \wedge over \vee we get
 $= ((\sim A \wedge \sim B) \wedge A) \vee ((\sim A \wedge \sim B) \wedge B)$
 Again using associativity of \wedge and using $\sim A \wedge A = F = \sim B \wedge B$ we get

R.H. S. = F

Consider L.H.S, applying associativity of \wedge , we get

$$= ((A \wedge B) \wedge (\sim A \vee \sim B)),$$

using left distributivity and commutativity of \wedge we get

$$= ((A \wedge B) \wedge \sim A) \vee ((A \wedge B) \wedge \sim B)$$

Using associativity of \wedge and using $A \wedge \sim A = F = B \wedge \sim B$

$$= (B \wedge F) \vee (A \wedge F)$$

Using $A \wedge F = F = B \wedge F$

$$= F$$

Ex. 8: The following table shows that $((A \rightarrow B) \wedge \sim B) \rightarrow \sim A$ is true in every interpretation. Therefore $((A \rightarrow B) \wedge \sim B) \rightarrow \sim A$ is valid and according to the First theorem, $\sim A$ is a logical consequence of $(A \rightarrow B)$ and $\sim B$.

Truth Table of $((A \rightarrow B) \wedge \sim B) \rightarrow \sim A$

A	B	$A \rightarrow B$	$\sim B$	$(A \rightarrow B) \wedge \sim B$	$\sim A$	$(A \rightarrow B) \wedge \sim B \rightarrow \sim A$
T	T	T	F	F	F	T
T	F	F	T	F	F	T
F	T	T	F	F	T	T
F	F	T	T	T	T	T

Ex. 9: (i) From the following table, $((A \rightarrow B) \wedge \sim B \wedge A)$ being False for all interpretations, is inconsistent.

Truth Table of $(A \rightarrow B) \wedge \sim B \wedge A$

A	B	$A \rightarrow B$	$\sim B$	$(A \rightarrow B) \wedge \sim B \wedge A$
T	T	T	F	F
T	F	F	T	F
F	T	T	F	F
F	F	T	T	F

(ii) Prove the inconsistency of $E_1 \wedge E_2 \wedge \sim G$, i.e., of $(A \rightarrow B) \wedge \sim B \wedge A$ by transforming, into a disjunctive normal form:

$$\begin{aligned}
 (A \rightarrow B) \wedge \sim B \wedge A &= (\sim A \vee B) \wedge (\sim B \wedge A) \\
 &= (\sim A \wedge \sim B \wedge A) \vee (B \wedge \sim B \wedge A) \text{ (Distributive Law)} \\
 &= (\sim A \wedge A \wedge \sim B) \vee (F \wedge A) \\
 &= \text{False} \vee \text{False} = \text{False}
 \end{aligned}$$

Thus $(A \rightarrow B) \wedge \sim B \wedge A$ is inconsistent.

Ex. 10:

Let us symbolize the statements in the problem state of above as follows:

A: The Parliament refuses to act.

B: The strike is over.

R: The president of the firm resigns.

S: The strike lasts more than one year.

Then the facts and the question to be answered in the problem can be symbolized as:

E1: $(A \rightarrow (\sim B \vee (R \wedge S)))$ represents the statement 'If the congress refuses to enact new laws, then the strike will not be over unless it lasts more than one year and the president of the firm resigns.'

E2 : A, represents the statement 'The congress refuses to act, by and'

E3: $\sim S$ represent the statement 'The strike just starts.'

E4: $\sim B$ (to be concluded)

Ex. 10: We solve the problem by showing that the formula $P: ((A \rightarrow (\sim B \vee (R \wedge S))) \wedge A \wedge \sim S) \rightarrow \sim B$ is valid by two methods: (i) by reducing to CNF/DNF (ii) by constructing truth-table of the formula.

Methods (i) Removing the two occurrences of ' \rightarrow ', we get

$$P = \sim ((\sim A \vee (\sim B \vee (R \wedge S))) \wedge A \wedge \sim S) \vee \sim B$$

Using De Morgan's Laws, we get

$$= \sim ((\sim A) \vee (\sim B \vee (R \wedge S))) \vee \sim A \vee \sim \sim S \vee \sim B$$

$$= (A \wedge (\sim \sim B \wedge \sim (R \wedge S))) \vee \sim A \vee S \vee \sim B$$

$$= (A \wedge (B \wedge \sim (R \wedge S))) \vee \sim A \vee S \vee \sim B$$

$$P = (A \wedge (B \wedge (\sim R \vee \sim S))) \vee \sim A \vee S \vee \sim B \dots (i)$$

Consider the case R is assigned value F

Then the formula P becomes

$$(A \wedge (B \wedge (\sim F \vee \sim S))) \vee (\sim A \vee \sim B \vee S)$$

$$= ((A \wedge B) \wedge T) \vee (\sim (A \wedge B) \vee S)$$

$$= (A \wedge B) \vee (\sim (A \wedge B) \vee S)$$

By denoting $A \wedge B$ by H we get $P = H \vee (\sim H \vee S) = T$ whether $(A \wedge B)$ is T or F

Consider the case when R is assigned T

Then the formula P given by (i) becomes

$$(A \wedge (B \wedge (\sim T \vee \sim S)) \vee (\sim A \vee \sim B \vee S) \text{ (using De Morgan Laws)}$$

$$= ((A \wedge B) \wedge \sim S) \vee (\sim (A \wedge B) \vee S)$$

$$= ((A \wedge B) \wedge \sim S) \vee (\sim (A \wedge B \wedge \sim S))$$

Denoting $(A \wedge B \wedge \sim S)$ by K we get

$$P = K \vee \sim K = T$$

Hence P is valid. Hence, the proof.

Method (ii)

The solution of the problem lies in showing that $\sim B$ logical follows from E_1 , E_2 , and E_3 . This is equivalent to showing that $P: ((A \rightarrow (\sim B \vee (R \wedge S))) \wedge A \wedge \sim S) \rightarrow \sim B$ is a valid formula. The truth values of the above formula under all the interpretations are shown in given table

A	B	R	S	$\sim B$	$\sim B \vee (R \wedge S)$
T	T	T	T	F	T
T	T	T	F	F	F
T	T	F	T	F	F
T	T	F	F	F	F
T	F	T	T	T	T
T	F	T	F	T	T
T	F	F	T	T	T
T	F	F	F	T	T
F	T	T	T	F	T
F	T	T	F	F	F
F	T	F	T	F	F
F	T	F	F	F	F
F	F	T	T	T	T
F	F	T	F	T	T
F	F	F	T	T	T
F	F	F	F	T	T

A	B	R	S	E ₁	E ₂	E ₃	$\sim B$	$\sim B \vee (R \wedge S)$	E ₁	$(E_1 \wedge E_2 \wedge E_3) \rightarrow \sim B$
T	T	T	T	T	T	F	F	T	T	T
T	T	T	F	F	T	T	F	F	F	T
T	T	F	T	F	T	F	F	F	F	T
T	T	F	F	F	T	T	F	F	F	T
T	F	T	T	T	T	F	T	T	T	T
T	F	T	F	T	T	T	T	T	T	T
T	F	F	T	T	T	F	T	T	T	T
T	F	F	F	T	T	T	T	T	T	T
F	T	T	T	T	F	F	F	T	T	T
F	T	T	F	T	F	T	F	F	T	T
F	T	F	T	T	F	F	F	F	T	T
F	T	F	F	T	F	T	F	F	T	T
F	F	T	T	T	F	F	T	T	T	T
F	F	T	F	T	F	T	T	T	T	T
F	F	F	T	T	F	F	T	T	T	T
F	F	F	F	T	F	T	T	T	T	T

Under all interpretations formula is True. Hence, the formula P a valid formula. $\sim B$ is a logical consequence of E1, E2 and E3. Hence, the “The strike will not be over” is a valid conclusion.

2.14 FURTHER READINGS

(In the order from *elementary* to *advanced*)

1. McKay, Thomas J., *Modern Formal Logic* (Macmillan Publishing Company, 1989).
2. Gensler, Harry J. *Symbolic Logic: Classical and Advanced Systems* (Prentice Hall, 1990).
3. Klenk, Virginia *Understanding Symbolic Logic* (Prentice Hall 1983)
4. Copi Irving M. & Cohen Carl, *Introduction to Logic, IX edition*, (Prentice Hall of India, 2001).
5. Carroll, Lewis, *Symbolic Logic & Game of Logic* (Dover Publication, 1955).
6. Wells, D.G., *Recreations in Logic* (Dover Publications, 1979).
7. Suppes Patrick, *Introduction to Logic* (Affiliated East-West Press, 1957).
8. Getmanova, Alexandra, *Logic* (Progressive Publishers, Moscow, 1989).
9. Crossely, J.N. et al *What is Mathematical Logic?* (Dover Publications, 1972).
10. Mendelson, Elliott: *Introduction to Mathematical Logic (Second Edition)* (D.Van Nostrand Company, 1979).