CHAPTER - 4

EXPERT SYSTEMS AND TURBO PROLOG

In this dissertation,the program which diagnoses the cardiac diseases is written in Turbo Prolog. The program creates a dynamic knowledge base and uses it to diagnose the different diseases. Since the concept of knowledge base is very much related to expert systems, a short discussion on expert system is presented in this chapter. An overview of Turbo Prolog language has also been introduced here.

4.1 Artificial intelligence

Artificial intelligence(AI) is simply a way of making a computer think intelligently. This is accomplished by studying how people think when they are trying to make decesions and solve problems, breaking those thought processes down into basic steps, and designing a computer program that solves problems using those same steps. The definition of AI may be given in the following way [2]:

" Artificial intelligence is the part of computer science concerned with designing intelligent computer systems,that is,systems that exhibit the characteristics we associate with intelligence in human behaviour - understanding language,learning reasoning, solving problems and so on."

In other words, AI is concerned with programming computer to perform tasks that are presently done better by

humans,because they involve such higher mental processes as perceptual learning, memory organisation and judgemental reasoning.

Thus, AI is about the simulation of human behaviour : the discovery of techniques that will allow us to design and program machines which emulate and extend our mental capabilities. The general introduction to AI may be found in [34,42,120,133].

4.2 Expert systems

An artificial intellignece system created to solve problems in a particular domain is called an **expert system.** All expert systems are based on up-to-date expert information for a given domain. Expert systems are capable of representing and reasoning about some knowledge-rich domain with a view to solving problems and giving advice. The process of constructing an expert system is often called **knowledge engineering** and is considered to be **applied artificial intelligence** [35]. Recent reviews of expert system research can be found in [2,11,13,32,42,47,132]. For tutorial on the structure of expert system [30,116] may be consulted. Discussion of expert systems in popular domains of science and medicine may be found in [19.36,122].

Expert systems use knowledge and inference techniques to solve problems that are usually solved with human expertise. An expert system stores a large body of facts,along with rules about how these facts can be used to solve problems. This collection of knowledge is called **knowledge base.** Unlike a traditional computer program, an expert system can be used to solve problems that are unstructured and where no formal procedure for finding a solution

exists. The expert system can use its internal knowledge and rules to formulate its own procedure based on the problem definition. The expert system is ideal for problems involving formal reasoning.

In most cases expert systems are used as tools by human experts to sift through large database of knowledge when decisions must be made objectively and quickly. There is a growing tendency to use expert systems in applications where human expertise is expensive or not available at all.

A computer always stores a very limited amount of knowledge in comparison with the human brain. In any expert system, knowledge generally is stored for only a single subject area, called a **domain**. The computer can analyze and solve only problems that are relevent to this domain. Because current microcomputers have very limited amount of memory, it is generally best for Prolog applications to confine this domain to as small a realm as possible.

4.3 The knowledge engineer

The biggest problem in the design of an expert system is abstracting the knowledge of the expert and putting it into an objective form for the database. A human expert normally approaches a problem at least partially subjectively. The computer in contrast, can only work with objective representations [17].

Expert systems are designed by **knowledge engineers**. It is the knowledge engineer who must observe, talk to, and work with the human expert to determine how to express the reasoning process of the human expert in an objective form. There is no general

theory or rules that can be used in expert system design. The problem that the human expert is trying to solve is generally unstructured and resists attempts at formal solutions.

4.4 Knowledge representation

In the field of expert systems, knowledge representation implies some systematic way of codifying what an expert knows about some domain. In knowledge representation theory the real world entities are objects. The design of expert systems involves paying close attention to the details of how knowledge is accessed and applied during the search for a solution [138]. Knowing what one knows, and knowing when and how to use it, seems to be an important part of expertise, this is usually termed **meta knowledge** i.e, knowldege about knowledge.

4.5 The production system

A **production system** is a type of expert system in which the knowledge is stored as rules and facts. Each rule is said to be a **production rule** [12]. The rules express relationship between facts. As Prolog uses rules and facts, it is very easy to implement a production system in Turbo Prolog.

A production system consists of a rule set(sometimes called **production memory**), a rule interpreter that decides how and when to apply which rules, and a **working memory** that can hold data, goals or intermediate results.

The block diagram of a production system is shown in Fig. 4.1. It has two primary components: the knowledge base and the inference engine [17].

The knowledge base

The **knowledge base** [17] is the data or knowledge used to make decisions. The knowledge base contains the rules and facts to approach a problem in a way similar to human expert.

The knowledge base consists of two parts: the working memory and the rule base. The **rule base** consists of facts and rules that are compiled as part of the problem. These do not change during a particular consultation. The rule base corresponds to Prolog's static database.

The second part of the knowledge base is the **working memory** . It consists of the facts relative to a particular consultation. The working memory corresponds to Prolog's dynamic database. At the begining of a consultation the working memory (dynamic database) is empty. As the consultation progress the inference engine(discussed next) uses facts and rules in the rule base, in conjunction with user input, to add facts to the working memory.

The inference engine

The **inference engine** [17] has two functions : inference and control. **Inference** is the basic formal reasoning process. It uses facts that are known to derive new facts by means of rules. Such inference operates by **modus ponens**. This is the basis for all formal logic. Simply stated, according to modus ponens,if a rule states IF A,THEN B, if A is true, then B will also be true.

The control function determines the order in which the rules are tested and what happens when a rule succeeds or fails. The inference engine takes the facts that it knows are true from

the rule base (static database) and working memory (dynamic database) and uses these to test the rules in the database. When a rule succeedes, the rule is said to fire, and the conclusion(head) of the rule is added to the working memory.

Turbo Prolog is really its own inference engine. It determines the order in which to scan the rules. This feature makes designing an expert system relatively easy because one has to write only the knowledge base. This feature means, however,that the user has less control. For example, Prolog scans the rules using **backwarding chaining** , that is, it starts at the goals and works backward. Prolog is also limited to **depth first scanning**. All rules relative to a particular goal are scanned as deeply as possible for a solution before Prolog backtracks and tries an alternative goal.

More complex expert systems may have additional components such as a natural language interface or an explanatory system.

A natural language interface improves user communication with the system. With most consultation systems, the user can enter a yes or no answer, select from a multichoice menu, or enter the value for an attribute (such as patient's age). A natural language interface permits the user to communicate with the expert system during the consultation process using a human language.

An **explanatory system** permits the user to answer a question with **why**. The system then responds with information about the goal it is trying to prove and so helps the user understand the system's reasoning process.

4.6 Certainity factor

Since heuristics are based on expert rules that are learned from experience,it is not always completely certain that an IF-THEN rule is correct. The user of an expert system cannot be certain that the value provided when instantiating a variable is a hundred percent correct. Therefore a certainity factor (CF) may be assigned to a rule. The CF has a value that approximates the degree to which one thinks the rule is correct. Values of CF range between -1 and +1. A CF of a negative number indicates a predominance of opposing evedience for the rule being correct. A positive CF indicates a predominance of confirming evidence for the rule being correct. Therefore, a CF of +1 indicates abosolute certainity that the rule is correct, while a CF of -1 indicates absolute certainity that the rule is incorrect. Of course, the rule with a CF that equated -1 would not be considered at all.

The method of calculating CF of rules and certainity level may be given as follows [136].

1. From a rule take the minimum CF of clauses connected by AND.
2. If there are OR connections in the rule, take the maximum CF value of all the AND clauses that are connected by ORs.
3. Multiply the final CF of the clauses by the CF of the rule.
4. If there is more than one rule leading to the same conclusion take as the final CF the maximum CF values of all those rules.

Example 4.1

IF A (CF = 0.3) AND B(CF = 0.6)

THEN C(CF = 0.5).

IF D(CF = 0.4) AND E(CF = 0.7)

THEN C(CF = 0.9).

The CF of C is the higher of the CFs of the two rules, as follows.

maximum ((minimum (0.3, 0.6) * 0.5), (minimum (0.4,

0.7 * 0.9))

= maximum ((0.3 * 0.5), (0.4 * 0.9))

= maximum (0.15, 0.36)

= 0.36

Example 4.2

IF A (CF = 0.3) AND

B (CF = 0.6) OR

D (CF = 0.5)

THEN C (CF = 0.4).

The CF for C is

maximum (minimum (0.3, 0.6), 0.5) * 0.4

= maximum (0.3, 0.5) * 0.4

= 0.5 * 0.4

= 0.2

In many cases certainity levels are given. What this means is that an inference is valid only if the CF exceeds this level. If the CF is below this level, a search through the knowledge base continues until an inference exceeds the level. The way the certainity level is calculated is as follows.

Assume that the CF for a particular inference is 0.6. Then that inference 0.4 is placed into a CF accumulator. The CF accumulator is compared with the certainity level (assume it to be 0.8). The accumulated value is less than the level and therefore the knowledge base search proceeds. The next time the same

inference is encountered in the knowledge base search, the CF for this new inference is multiplied by 1 minus the value in the CF accumulator and the result is addedto the CF accumulator. The 1 in the equation comes from the absolute certainity value = 1. Now the CF accumulator is compared with the certainity level. If greater, the answer will be obtained otherwise the search will be continued.

The equation is :

Accumulated CF = old accumulated CF + (1 - old accumulated Cf) * CF of the rule)

Example 4.3

Certainity level = 0.8

Rule = if A then B (assume CF = 0.6)

Accumulated CF = 0.6

New rule = if C then B (assume CF = 0.7)

Accumulated CF = 0.6 + (1 - 0.6) * 0.7

= 0.88 (exceeds certainity level; stop).

4.7 *Using Prolog to design rule based systems*

Prolog is short for Progimming in Logic.The language was originally developed in 1972 by Alain Colmerauer and P. Roussel at the Universlity of Marsilles in France. It is a computer language that was created especially for answering questions about a knowledge base that consists of rules and facts. Prolog has 'backward chaining' built right in and also utilizes another technique known as **backtracking**. Backward chaining is a technique in which a conclusion or consequence is assumed to be true, and

then knowledge base of rules and facts is examined to see if it supports the assumption. If the assumption turns out 'not' to be correct, backtracking is used to get rid of the original assumption and replace it with a new one.

Like procedural languages, Prolog does not use algorithms . Algorlithms are objecltive procedures that, when followed, guarantee a solution. Object-oriented languages, such as Prolog, use heuristics. Heuristics are rules of thumb that are useful in reaching a goal. They donot guarantee a solution, and are useful when no algorithm exists. Prolog uses only data about objets and their relationshps. Prolog also emphasizes symbolic processing.

With Prolog, the user defines a goal (a problem or objective), and the computer must find both the procedure and solution. A Prolog program is a collection of data or facts and the relationships among these facts. In other wards, the program is a database.

There is no standard for the Prolog language,but the closest thing to an accepted standard is the Prolog described in Clocksin and Mellish's book which is generally known as C&M Prolog [171]. Most commercial Prologs have evolved as supersets of this core language.

Turbo Prolog is both a superset and a subset of C&M Prolog. It has many features that are not a part of C&M Prolog,yet it does not support many features that are supported in the core language. In addition, Turbo Prolog supports some features (such as input and output) by structures more similar to those of the C Language than those of C&M Prolog. Program written in a C&M Prolog

will not run under Turbo Prolog without modification - generally,extensive modification. Similarly, Turbo Prolog programs must be extensively modified to run under C&M Prolog.

In the next section an introduction to Turbo Prolog Language has been presented. The material is taken from [17,99].

4.8 Turbo Prolog

We can express knowledge in Turbo Prolog in terms of either **facts** or **rules**. The basic units for building facts or rules are **predicates**, i.e., expressions that say simple things about the individuals in our universe. For instance, the piece of information "The right speaker of the stereo system is not emitting sound (is dead)" is represented in Turbo Prolog as

is(right-speaker,dead).

The factual expression in Prolog is called a **clause**. In this example :

right-speaker and **dead** are **objects**. An object is the name of an element of a certain type. It represents an entity or a property of an entity in the real world.Turbo Prolog permits the user to use six different object types viz, char, integer, real, string, symbol and file.

The word **is** is the relation in the example. A relation is a name that defines the way in which a collection of objects (or objects and variables referring to objects) belong together.

The entire expression before the period is called a **predicate**. A predicate is a function with a value of true or false. Predicates express a property or a relationship. The word before the parentheses is the name of the relation. The elements

within the parentheses are the arguments of the predicate, which may be objects or variables .

Here are a few examples of predicates :

```
employee(bill)

student(tom)

married_to(bob,mary)
```

If a period is added to a predicate, we may get a complete clause.

In the Turbo Prolog language, an object is something complely different. In Turbo Prolog, an object is the name of any constant. The Turbo Prolog object could be an attribute,a property, or a value.

Rules

Rules permit Prolog to infer new facts from existing facts.

A rule is an expression that indicates that the truth of a particular fact depends upon one or more other facts. Let us consider the following example.

IF there is a body stiffness or pain in the joints

AND there is a sensitivity to infections,

THEN there is probably a vitamin C deficiency.

This rule could be expressed as the following Turbo Prolog clause.

```
hypothesis(vitc_deficiency) if

        symptom(arthritis) and

        symptom(infection_sensitivity).
```

The conclusion of the rule is stated first and is following by the word if. The conditions upon which conclusion

depends are stated next, each connected by the word and. The conclusion can be viewed as Prolog goal. The goal is true if all the conditions specified for the goal are true.

Every rule has a conclusion (or head) and an antecedent (or body). The antecedent consists of one or more premises. The premises in the antecedent form a conjuction of goals that must be satisfied for the conclusion to be true. If all the premises are true, the conclusion is true, if any premise fails, the conclusion fails.

Since rules are such an inportant part of any Prolog program, a shorthand has been developed for expressing rules. In this abbreviated form the previous rule becomes

```
hypothesis(vitc_deficiency) :-
        symptom(arthritis),
        symptom(infection_sensitivity).
```

The :- operator is called a break. A comma expresses and relationship, and semicolon expresses an or relationship. However, if we wish to express an or relationship, it is generally clearer to use two rules :

```
hypothesis(vitc_deficiency) :-
        symptom(arthritis),
        symptom(infection_sensitivity).
hypothesis(vitc_deficiency) :-
        symptom(colitis),
        symptom(infection_sensitivity).
```

In English , these expressions state there is evidence of a vitamin C deficiency if there is either arthritis and infection sensitivity or colitis and infection sensitivity. If one

or other pair of premises is true, the conclusion is true. The process of using rules and facts to solve a problem is called **formal reasoning** .

The Turbo Prolog program

Turbo Prolog program consists of two or more sections. The main body of the program the **clauses** section, contains the clauses and consists of facts and rules. The relations used in the clauses section are defined in the predicates section. Each relation in each clause must have a corresponding predicate definition in the **predicates** section. The only exceptions are built-in predecates that are an integral part of Turbo Prolog.

The **domains** section also is a part of most Turbo Prolog programs. It defines the type of each object.

The **goal** section of the program defines the internal goal. If Turbo Prolog can match the goal with a fact or rule in the data base, it **succeeds** otherwise it **fails**. **Variables** are used in a Turbo Prolog clause or goal to specify an unknown quantity. A variable name must begin with a capital letter and may be from 1 to 250 characters long. Except for the first character in the name, one may use uppercase or lowercase letters, digits, or the underline character. If a variable has a value at a particular time in a program, it is said to be **bound**. If a variable does not have a value at a particular time, it is said to be **free**. For a goal to succeed, all variables in the goal must become bound.

The Turbo Prolog program can consist of upto eight sections. Most program only use a few of these sections, but the complete list gives the extensive flexibility : global predicates

for modular programming, dynamic databases,internal goals, and compiling directions.

Although not all sections will be used in all programs, those that are used must be in the following order for the compile operation.

1. Compiler directives.
2. Domains.
3. Global domains.
4. Database.
5. Predicates.
6. Global predicates.
7. Goal.
8. Clauses.

Unification

The process by which Prolog tries to match a term against the facts or the heads of the other rules in an effort to prove a goal is called **unification**. Unification is a pattern matching process.

Predicates unify with each other if

1. They have the same relational name.
2. They have the same number of arguments
3. All argument pairs unify with each other.

Unification is similar to parameter passing in procedural programming. Values for one term are passed to another term, binding any variables in that term.

Prolog execution rules

The rules for Prolog execution are given below :

1. Prolog executes using a matching process.
2. When the original goal is specified, Prolog tries to find a fact or the head (conclusion) of a rule that matches the goal
3. If a fact is found, the goal succeeds immediately.
4. If a rule is found, Prolog then tries to prove the head of the rule by using the antecedent (the body of premises) as a new goal and proving each premise of the antecedent. If any premise of the antecedent fails, Prolog backtracks and tries to solve the preceding premises with other bindings. If Prolog is not successful, it tries to find another fact or rule that matches the original goal. If all premises succeed, the original goal succeeds.
5. Turbo Prolog continues to execute until all possible solutions for the goal are tested.

There are no global variables ; that is, variables that maintain a value throughout the entire program's execution. All variables are local to the clause of which they are a part. Even if the same variable name is used in another clause, it is not the same varible. If a particular clause fails, Prolog backtracks and tries to solve the same goal another way using clauses.

Built-in predicates

Most versions of Prolog contain a variety of built-in, or standard, predicates that can support a variety of functions, such as control and data input and output. Turbo Prolog offers dozens of built-in predicate to support not only input and output

operations, but graphics, file operations, string handling and type conversion. These predicates can be used in rules or facts, just like any other predicate. They do not need to be defined in the predicate section, as they are integral to Turbo Prolog.

The Turbo Prolog predicates are classified into nine groups :

1. Control predicates - For controlling program execution, forcing or preventing backtracking.
2. Reading predicates - For reading data from the keyboard or file to a variable.
3. Writing predicates - For writing data to the screen, printer, or a file.
4. File system predicates - For managing disk files from a Turbo Prolog program.
5. Screen-handling predicates - For graphic control of the display and sound control.
6. String-handling predicates - For various operations on string data.
7. Type-conversion predicates - For converting data types from one form to another.
8. System-level predicates - For access to DOS functions from within a Turbo Prolog program.

Some of the built-in predicates will be discussed now.

The write predicate

The write predicate displays the value of the variables or objects. The variable must be bound before the write predicate is invoked. For example,the clause

```
test :-
    write("This is an exmple "),
    write("of multiple write statements").
```

displays

This is an example of multiple write statemets

The output may be written in two lines using the control predicate nl .

The predicate nl indicates a new line.

Any output to the display can be directed instead to a printer or a file. To redirect output, the writedevice built-in predicate is used.

The following code directs the output of the write predicate to the printer and then redirects further output back to the screen.

```
writedevice(printer),
write("This will print on the printer"),
writedevice(screen).
```

Likewise Turbo Prolog provides the readdevice predicate for redirecting input.

The read predicate

Turbo Prolog provides several built-in input predicates. One of these is a read predicate. There are four types of read predicate : readln, readchar, readint and readreal.

The readln predicate permits a user to read any string or symbol into a variable. Let us consider the following example :

```
symptom(Patient,fever) :-
        write("Does the ",Patient,"have a fever(yes/no)?"),
        readln(Reply),
        Reply = "yes"
```

When this is invoked, it displays the question and then pauses for an answer. The answer must be terminated with a carriage return. The rule will succeed if the user enters *yes* and fail if the user enters anything else.

In the same way, **readchar** predicate permits to read any character, **readint** to read any integer and **readreal** to read any real value into a variable.

The fail predicate

In Prolog, forcing a rule to fail under certain conditions is a type of control and is essential to good programming.

Failure can be forced in any rule by using the built-in **fail** predicate. The fail forces backtracking in an attempt to unify with another clause. Whenever this predicate is invoked, the goal being proved immediately fails, and backtracking is initiated. The predicate has no arguments, so failing at the fail predicate is not dependent on variable binding; the predicate always fails.

Example 4.4

```
go :-
        test,
        write("you will never get here").
```

```
test :-
     fail.
```

If the goal is specified as go, Prolog will unify with the head of the first rule and then try to prove it premises. The test premise will unify with the head of the second rule, whose premise is fail, and the goal will fail. Prolog will backtrack to the first rule, and the go goal will fail again. The write predicate will never be executed.

The same program may be tried again, reversing the two premises and changing the test string slightly :

```
go :-
     write("you will get here."),
     test.
test :-
     fail.
```

Again the goal will fail, but the text string will be displayed this time.

The not predicate

If we want to express explicitly in the database that a particular fact is not true, the built-in not predicate has to be used. The not predicate cannot be used to express a fact or appear in the head of a rule. It can only be used in a premise, as in

```
replace(right-speaker) :-
     not(is(right-speaker,functional)).
```

In this case, if

```
is(right-speaker,functional).
```

is in the database, the rule will fail.

The cut predicate

The cut is one of the most important, and also one of the most complex,features of Prolog. The primary purpose of the cut is to prevent or block backracking based on a specified condition. The cut predicate is specified as an exclamation point(!). It has no arguments. the cut predicate always succeeds, and once it succeeds,it acts as a fence,effectively blocking any backtracking beyond the cut. If any premise beyond the cut fails,Prolog can only backtrack as far as the cut to try another path. If the rule itself fails and the cut is the last premise, no other rules with the same head can be tried. Prolog must accept failure or success of the predicate based on that particular clause.

There are two types of cuts. In Prolog,these are called the green and the red cuts. The green type of cut is used to force binding to be retained,once the right clause is reached. Green cuts are used to express determinism. A program is nondeterministic if it is capable of generating multiple solutions on backtracking. Thr red type of cut is used to omit explicit conditions.

The use of any type of cut in a Prolog program is controversial. It implies a type of procedural control,which is in sharp contrast to the declarative style of Prolog programming.If used with caution,however,cuts improve the clarity and efficiency of most programs. Of the two types of cut,the green cut is the more acceptable type. One can often use the not predicate instead of the red cut.

Example 4.5

Let us assume a rule of the form :

```
go:-
  premise1,
  premise2, ! ,
  premise3,
  premise4.
```

Prolog will try to prove the go by backtracking between premise1 and premise2 as necessary until both are true. Once this occurs,the cut is reached. The cut predicate always evaluates as true,so testing begins on premise3. Once the cut is crossed , Prolog cannot backtrack across the cut. If primise3 fails,the rule fails. If premise4 fails,Prolog backtracks to premise3 to try another path for this premise, but Prolog goes no further. If either premise3 or premise4 does not evaluate as true,the rule fails without going back to premises before the cut and attempting to prove them in a different way,for example,with different bindings. All variables in premise1 and premise2 are bound when the cut is crosed, and Prolog is committed to all choices before the cut.

The makewindow predicate

A window may be created at any time in a program by using the makewindow predicate. The general form of the makewindow predicate is

```
makewindow(WindowNo, ScrAttr, FrameAttr, Header,
                        Row, Col, Height, Width)
```

All except the header argument are integers.The header is a

string or symbol.The arguments are explained as follows:

WindowNo	Assigns a number to the window.
ScrAttr	Defines the video attributes of the window display.
FrameAttr	Defines the video attributes of the border.
Header	Defines a header for the window.
Row,Col	Defines the starting point (upper left corner) for the window.
Height,Width	Defines the size of the window.

Example 4.6

```
makewindow(1,7,7,"ToastWindow",0,0,25,80)
```

Several windows can be created with seperate number. The shift between windows can be performed using **shiftwindow** predicate: **shiftwindow(WindowNo)**

When a window is shifted, the window will be clear, with the cursor at the upper left.The **Cursor(Row,Col)** predicate positions the cursor at (Row,Col) in the currently active window.

Prolog databases

A Prolog program is a collection of facts and rules about a particular knowledge domain. The program really is a database, and Prolog is very powerful query language for this database, permitting one to select facts from the database through unification .

The program, however, is a static database; that is,the database does not change over time.

To solve the medical diagnostic problem, Prolog permits us to add a dynamic database to the program. Built-in predicates

permit to add facts to or remove facts from this dynamic database during program execution.

To store information in a dynamic database, one or more database predicates have to be created. Facts can then be stored in these predicates during program execution using the built-in **asserta** or **assertz** predicates.

To creat database predicates, a database section must to be added to the program. The database predicates are defined in this section. The section must follow the domains section and precede the predicates section.

Example 4.7

```
Domains
        disease, symptom = symbol
        query = string
        reply = char
database
        xpositive(symptom)
        xnegative(symptom)
predicates
        hypothesis(disease)
        symptom(symptom)
        ...
        ...
        ...
```

Here two database predicates are used in the database section : **xpositive** and **xnegative**. The database predicates are not defined in the predicates section. With these database predicates one can store the facts, learned from the questions, in the

database (facts proven true in xpositive and facts proven false in xnegative) and query the database before asking the same question again.

The asserta and assertz predicates

The asserta(fact) predicate stores a fact at the begining of the database. The assertz(fact) predicate stores a fact at the end of the database. Which of these will be used in the program depends upon where one wishes to put the fact in the database.

The retract predicate

Once a fact is stored in the database, it can be removed only using the built-in retract(fact) predicate.

Example 4.8

retract(xpositive(fever)) removes the fact fever from the database xpositive(fact).

To save the current dynamic database, the built-in predicate save(filename) predicate may be used.

File opening predicates

There are four file-opening predicates in Turbo Prolog. These are :

1.	openread (SymbolicFilename,Filename)	Opens file for reading. If the file is not there, the predicate fails.

2.	openwrite (SymbolicFilename, Filename)	Opens the file only for writing. Any previous file with the same name is deleted.
3.	openappend (SymbolicFilename, Filename)	Appends any new output to the end of Filename. If Filename does not exist, the program terminates execution.
4.	openmodify (SymbolicFilename, Filename)	Opens file for writing and reading using random access.

All the predicates discussed here have the same number and type of arguments. Any of them can be used to open a file. The first argument is the symbolic file name that will be used for the file in the program. The symbolic file name can be any name so long as it meets the Turbo Prolog syntax rules; that is it must starts with a lowercase letter and consists of alphabetical characters, digits, or the underscore. The name must also by typed in the **domains** section as a file type.

The second argument, **Filename**, is the name of the file on the disk. The name is a string value and contains from one to eight characters plus an extension. A file may be accessed randomly using the **filepos** built-in predicate. The general form of the filepos predicate is

filepos(SymbolicFilename, Position, Mode)

where **Mode** determines how **Position** is measured as follows

0 = from begining of the file

1 = from current position

2 = from end of file.

The end of the file during a read operation may be checked using

eof(SymbolicFilename)

This test will be succeed if current position is at the end of the file; otherwise, it will fail.

To prevent loss of data, **closefile** predicate is used to close a file after using it. The general form of the closefile predicate is **closefile (SymbolicFilename)**.

4.0 Concluding remarks

A brief discussion on expert systems and Turbo Prolog Language is presented.

Expert systems are computing systems which embody organised knowledge of a specialized domain such as medical diagnosis. All the knowledge in the expert system is provided by people who are experts in that domain. These expert (or knowledge based) systems are now recognised as key element in the computing systems of the future; and the most significant applications of artificial intelligence . Expert systems can advise, diagnose, analyze and categorize using a previously defined knowledge base. The knowledge base is a collection of rules and facts, often written in Prolog. Turbo Prolog like other implementations of Prolog, is an object oriented language and uses no procedures and essentially no program (However, it is convenient to refer to the systems written in Prolog as programs).

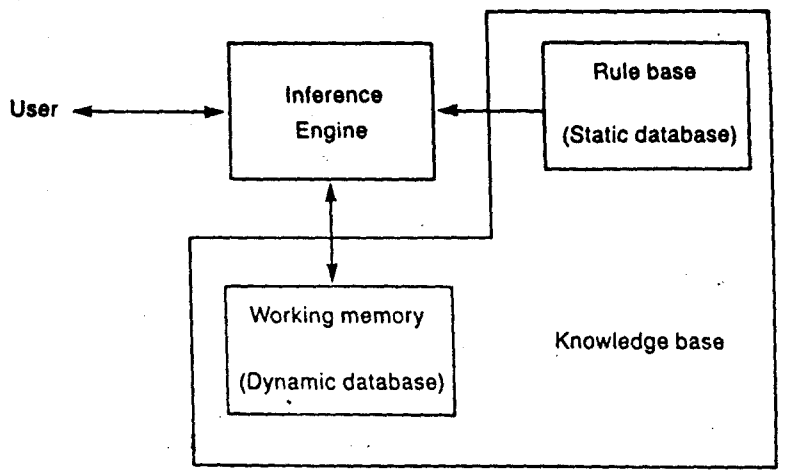

Fig.4.1 The production system.