CHAPTER 3

PROPOSED MODEL FOR STATISTICAL EXPERT SYSTEM FOR INFORMATION MANAGEMENT

3.1 Introduction

Adequate conceptualization and background building have gone on, in the last two chapters, that it is easier now to deal with what needs to be done and what resources need to be gathered before a Statistical Expert System (SESYS) can be designed. The review and appraisal of literature in the first chapter has provided for a gestalt of ideas that make up the expert system, in all its structure, design and relevance in today’s world, as a decision support system for various people with multifarious requirements. The first chapter has also focused on the problem under study and the objectives, besides important definitions and the scope of the study. In the second chapter, the notion of a SESYS has been elaborated in terms of its architecture, inference machine, program structure, functions and modules, such that the SESYS can be further modeled, designed, integrated and built so that its components and how it works can be understood.

The purpose of the current chapter is to explicitly indicate what goes into the design and building of the SESYS, of its merits and demerits, and of its workability. In the development and implementation of an Expert System application for solving problems, two most significant problems appear to be (Ansari and Modarress, 1990):

(i) the lack of qualified knowledge engineers and Expert System designers; and
(ii) the lack of commitment on the part of top management and availability of experts from whom the necessary knowledge must be derived.
The present chapter is about the statistical expert system, under design and development for the purpose of information management. This chapter therefore speaks of the architecture, the statistical inference engine, and the Turbo Prolog for designing statistical expert systems are elaborated. The structure of the system and the structure of modules of the system are further elaborated with a view to beginning the coding of the program. Steps in modular programming, statistical facts, goals and rules for programming are being set down. Segments of the program have all been written and their functions have then been discussed. The codes generated in Turbo Prolog for the segments are also given and an appreciation of how they work are shown. The strength of the chapter lies in sequencing the program and having it written in a manner that makes the use of it simple and straightforward.

3.2 Prolog to Design SESYS

Object-oriented programming is based on the description of objects. These objects send messages to (other) objects and apply methods when messages are received. These methods can send additional messages and/or cause changes in the status of the object.

PROLOG is a declarative programming paradigm. In this paradigm, a problem is described using facts and rules. In the strictest sense of the paradigm, no procedural sequence may be formulated.

As PROLOG is a declarative language, the knowledge can easily be represented in the form of production rules (Bratko, 1990; Leigh, 1987; Robinson, 1987; and Schildt, 1987). PROLOG machines are primarily developed within the framework of the "Fifth-Generation Program" in Japan and England. They are not yet available to the general public. Dedicated machines are intended as workstations. In the absence of such machines, the
alternative of DOS system / IBM compatible / Windows based may be used, but programs need to be written for executions.

Using Turbo Prolog to Design Statistical Expert System. Turbo Prolog was created especially for answering about a knowledge base that consists of rules and facts. Turbo Prolog has backward chaining built right in and also utilises another technique known as “backtracking”. Backward chaining is a technique in which a conclusion or consequence is assumed to be true, and then a knowledge base of rules and facts is examined to see if it supports the assumption. If the assumption turns out to be incorrect, backtracing is used to get rid of the original assumption and replace it with a new one (Levine, 1990).

Kowalski (1979) wrote an article on this subject entitled “Algorithm = Logic + Control”. In this article, he shows that the logic component in algorithms can be separated from the control mechanisms. The logic is represented in the form of predicates, which appear in three forms: as facts, rules and questions. The logic formulated as facts and rules is designated the knowledge base. The questions are then directed at this knowledge base.

The queries are processed by the so-called theorem prover, which checks whether the predicate to be proved can be deduced from the knowledge base. The theorem prover is part of the Turbo Prolog interpreter and represents the control mechanism, which consists of unification and backtracking.

The expert system designer needs to do little more than design the logic of a program, and no longer has to program the individual computing steps. This type of programming is called declarative, as opposed to the conventional procedure-oriented programming (Nebendahl, 1988).

SESYS Syntax and Semantics. The syntax of Turbo Prolog is based on logic expressions, and its semantics is defined using the concept of resolution and unification (Tucker, 1997). The description of Turbo Prolog
syntax is restricted to those parts necessary for the sample program, and therefore, does not represent the full range of Turbo Prolog syntax and semantics. The sample program in the next section contains six predicates. The predicates consist of one or more clauses \((\text{statistical method}(\text{technique}), \text{consists of five clauses})\). A clause (also called a \text{Horn} clause) always extends from the \text{predicate name} (\text{statistical method}) up to the terminating period. A clause can be a \text{statistical fact}:

\[
\begin{align*}
\text{statistical method('laspeyres index')} \\
\text{city('chennai')}. \\
\text{known(prices\_in\_current\_year)}. \\
\text{pincode('600005')}. 
\end{align*}
\]

or a \text{statistical rule}:

\[
\begin{align*}
\text{move\_method(From,To) :-} \\
\quad \text{decide\_method(From,To,Reasons)} \\
\quad \text{print\_decision(From,To,Reasons)}. 
\end{align*}
\]

The rule consists of the head \((\text{move\_method(From,To)})\) and the body \((\text{decide\_method(From,To,Reasons)}, \text{print\_decision(From,To,Reasons)})\), separated by \(\text{ :-}\). A statistical fact has only one head.

The symbol \(\text{ :-}\) stands for a logical implication, but one which works in the opposite direction: if the body of a statistical rule is true, then head is also true. A statistical fact is therefore always true, since it has no condition. A body is true if each of its parts is true.

A predicate name, together with the list of its parameters in parentheses, is called a \text{structure}.

\[
\text{decide\_method(From,To,Reasons)}. 
\]

In addition to the predicate name, every predicate has a fixed number of parameters.

Symbols, which begin with a lower-case letter or that which are enclosed in single quotes, are constants:

\[
\text{statistical\_method, 'laspeyres index'}. 
\]
Symbols which begin with an upper-case letter are variables. Variables used in a clause are local, that is they have different meanings in other clauses:

**Method, City, Pincode, Technique, From, To, Any.**

The data structure used most often in Turbo Prolog is the list. A list consists of the opening square bracket " [ " , the list elements, which are separated by commas " , " and the closing square bracket " ] ":

[ ' is indexnumber', 'is more appropriate'].

A list element can itself be another list.

**Important Mechanisms of Turbo Prolog.** Important mechanisms used in Turbo Prolog include: recursion, instantiation, verification, unification, backtracking, and interchangeability of unknowns. **Recursion** represents the most important program structure. Branches and **FOR** loops are not provided in the Turbo Prolog syntax, and **WHILE** loops are very difficult to implement, since variables can be linked only once. Recursion is more suitable than other program structure for processing recursive data structures such as lists, and is characterised in these cases by simpler representation and better clarity (Nebendahl, 1988). **Instantiation** is the linking of a variable to a constant or structure. The linked variable then behaves like a constant. **Verification** denotes the attempt to derive from statistical knowledge base (statistical facts and rules) the structure of a query, which is to be proved. If this is possible, the structure is true; otherwise it is false. **Unification** is the main component in the verification of structures. A structure is proved if it can be unified (matched: see below) with a fact, or if it can be unified with the head of a statistical rule, and the structures in the body of this rule can be verified.

**Example** of Unification (see also sample program)
The structure "known(Method)" can be unified with the fact known('lasperyestindex') and thus proved. The structure move_method(Method1, Method2), on the other head, can be matched only with the clauses head move_method(From, To), and is only satisfied if the clause body has also been verified.

An attempt is now made to prove decide_method(From, To, Reasons) and print_decision(From, To, Reasons).

The unification of two structures work as follows: the set of statistical facts and rules is searched from top down in the sequence in which they are written, until a statistical fact or a clause head is found which belongs to the same predicate as the statistical fact which is to be proved. The individual parameters are then checked in pairs to determine whether they match: two constants must be identical, two variables are linked to one another, one variable assumes the value of the constant.

After the unification of known_method(Method) with known_method('lasperyestindex'), Method has the value 'lasperyestindex'. In other words, Method is instantiated to 'lasperyestindex'.

In the verification of a structure, it is possible that this structure cannot be unified with a found structure of the same predicate. In such a case, the statistical knowledge base is searched further for a suitable statistical fact or rule head, which can be unified.

If none can be found, the structure is not verifiable. If the structure to be proved is the goal of a query, the query is answered in the negative. If, on the other hand, it is only part of a clause body, then the links (instantiations) established in the preceding structure of the clause body are cancelled and another attempt is made at unification. This is followed by an attempt to verify the second structure. This process of cancelling instantiations and backing up in the computation when verification fails is known as
backtracking. Chaudhuri (2001) has discussed the architecture for approximate query processing.

Interchangeability of unknowns is illustrated in the user dialog section. In contrast to procedures written in other programming languages, such as PASCAL, predicates in Turbo Prolog do not define which parameters are input parameters, output parameters, or transient parameters. A predicate behaves in different ways, depending on whether a parameter is a variable or a constant:

move_method('laspeyresindex', 'pascheindex') determines the reasons for a move_method. If 'pascheindex' is not a suitable statistical technique, the answer is not verifiable.

move_method('laspeyresindex', New_method) supplies the set of suitable statistical techniques, together with the respective reasons.

move_method(Old_method, 'pascheindex') finds all statistical techniques for which a move_method to statistical technique 'pascheindex' would be an improvement.

This example also illustrates the difference when compared to procedure-oriented programming. Procedure-oriented programming would require three different procedures for these actions.

Backward chaining is implemented in the Turbo Prolog control mechanism. A rule IF A and B and C then D correspond to the Turbo Prolog clause D:-A,B,C. However, this backward chaining is normally too weak to process rules for an expert system, since there are no control options for rule interpretation, nor trace facilities (Levine, 1990).

In spite of its declarative character, Turbo Prolog does possess control structures. In some cases it is desirable or even necessary to prevent backtracking within a part of a predicate. This is made possible through the use of the pseudo-predicate ‘!’ (pronounced “cut”). During backtracking,
instantiations before the `!' are not cancelled; instead, the predicate is abandoned immediately.

**Sample Program.** An example of a program is given below for determining a suitable statistical technique to suggest. A statistical technique is considered to be suitable if there is no better statistical technique available. Because not all statistical techniques can be compared to one another, there may be several suitable statistical techniques.

The Turbo Prolog program contains six predicates:

```
move_method, decide_method, print_decision,
statistical_method, known and better_method.
```

The statistical facts given are the statistical techniques (available methods) and a comparison of methods (predicate **better_method**)

```
statistical_method('laspeyres index').
statistical_method('paasche index').
statistical_method('weighted index').
known('-------').
known('-------').
known('-------').

Better_method('laspeyres index', 'paasche index', [' is better', 'current year quantities known']).
Better_method('laspeyres index', 'paasche index', [' is better', 'current year quantities known']).
Better_method('laspeyres index', 'paasche index', [' is better', 'current year quantities known']).
```

The last predicate states: statistical technique 'weighted average of relatives quantity index' is better than 'un-weighted average of relative quantity index' because the quantities and prices that determine values we use for weights are known. In all other respects, the two methods are same.

The most important predicate is **move_method**; it determines the reasons for a move with the help of the predicate **decide_method**, and prints out the result with **print_decision**.

```
move_method(From, To):-
  decide_method(From,To,Reasons),
  print_decision(From,To,Reasons).
```

**decide_method** tests whether a statistical technique is suitable for a move, and records the reasons in favour of this statistical technique.
decide_method(Old_method, New_method, ReasonsForNew_method):-
  statistical_technique(AnyTechnique),
  better_method(AnyTechnique, Old_method, Reasons1),
  decide_method(AnyTechnique, New_method, Reasons2),
  append(Reasons1, Reasons2, ReasonsForNew_method).

A search is made for a statistical technique (AnyTechnique), a test
determines whether this technique is better than the previous technique
(Old_method), and the reasons in favour of this statistical technique are
recorded in Reasons1. The predicate decide_method is then applied recursively
to this statistical technique, and a search is made for a technique that is more
suitable than the statistical technique appropriate/available. The reasons for
the new method, more suitable technique, are represented by the union of the
reasons for the statistical technique appropriate/available and the reasons
from the recursion. The auxiliary predicate append is needed to concatenate
these two lists.

decide_method(Technique, Technique, [ ]):-
  statistical_technique(Any_method),
  not(better_method(Any_method, Technique, Reasons)).

The second case ends the recursion: for all available statistical
techniques (Any_method), it is true that they are not better than Technique.
If that is the case, then the suitable statistical technique is Technique (the
second parameter of decide_method). There are no additional reasons in
favour of this statistical technique.

print_decision(From, To, Reasons):-
  nl, nl, nl,
  write("A move from statistical technique"),
  write(From),
  write(" to statistical technique"),
  write(To),
  write(" is supported by the following reasons: ")
  nl, nl,
  write(Reasons).

For information, several sections which were originally written to be
included in this chapter, from SESYS Input and Output to Saving the SESYS
Dynamic Database, have now been moved to Chapter IV, because it is in
there that the implementation of the SESYS is demonstrated. This section
contains some of the program segments, which should have been here, logically. But they are moved to a later chapter because they fit in very well with the later chapter.

**SESYS User Interface.** This component determines how the SESYS interacts with the user.

- How should the statistical questions be answered by the user?
- How will SESYS' responses to these questions be formulated?
- What information is to be represented graphically?

The following requirements must be for the user interface:

- Operation must be easy to learn.
- Erroneous input must be prevented to the extent possible.
- Results must be supplied in a form appropriate for the user.
- Questions and explanations must be understandable (Nebendahl, 1988).

In the long run, only an interface that is acceptable to users will guarantee the success of SESYS.

Since the SESYS2001-Version of SESYS was developed for a line-oriented screen, its operation is not nearly as convenient as that of the other shells (S.l, KEE), which provides the user with windows and mouse-activated menus. SESYS also lacks graphics-based trace options, which can be used to continuously follow the application of rules during the course of consultation.

Nevertheless, SESYS does have an explanation component and several other functions that can be called when it asks a question.

Before the user answers the question, he can

- have the rules and facts and (answers to questions), which led to the current question displayed (this serves as an explanation component);
- change the answer to a previous question, if necessary by requesting the output of all questions asked thus far;
have information about the current status of the SESYS displayed (order of priority of the diagnoses in the agenda).

• Terminate the session with STOP.

SESYS Functional Design. SESYS was designed with the goal of providing a statistical expert system for information management, which is simple, and yet interesting to use. The system is particularly useful for demonstrating some of the basic paradigms, which are important for the implementation of SESYS.

SESYS consists of the following components:

1. The Statistical Knowledge Base (SKB), in a file with several access and modification functions;
2. A Blackboard, used as a dynamic knowledge base;
3. An Explanation component;
4. A Main task for controlling the dialog conducted via the user interface.

Statistical Knowledge Base. In principle, the statistical knowledge base of SESYS should contain production rule structure. Special support is provided in the form of several incarnations, access and update functions. These functions provide general support for the selection of statistical techniques.

Blackboard. The blackboard is used by the statistical inference mechanism to store intermediate or final results of inference, or to record speculations, which are to be considered later in the consultation. Entries are made in the blackboard when specific statistical rules “fire” (Figure 3.1). These rules are called – Demons.

1. The “last answer” field provides quick access to the last entry made by the user and the last decision made by the SESYS. In principle, the statistical inference mechanism can access all
previous decisions and information through the statistical knowledge base.

2. "Conclusions" are definitive results deduced by the statistical inference mechanism, which can be used in subsequent decision-making. "Conclusions" can only be added; they cannot be modified during a consultation.

3. "Vague results" are deduced results which restrict the set of possible answers. The cardinality of this set is greater than 1.

<table>
<thead>
<tr>
<th>Last Answer</th>
<th>Conclusion</th>
<th>Vague Results</th>
<th>Assumptions</th>
<th>Contributions</th>
</tr>
</thead>
</table>

Figure 3.1: Blackboard Structure

4. If the statistical inference mechanism supplies only vague results, "assumptions" can be formulated, which must then be verified or proved wrong.

5. If an optimal step cannot be deduced from the previous conclusions and assumptions, it is advantageous to record all possible alternatives, evaluate them, and select the "best" alternative. These tasks are handled by the heuristic rules (Nebendahl, 1988). The development on this may be carried out as further research work.

**Statistical Inference Mechanism.** The statistical inference mechanism first activates the Demons. The firing demon then writes:

- conclusions,
- vague results and
- assumptions into the blackboard.
**Explanation Component.** The explanation component uses only statistical information from the blackboard. This component supplies the following statistical information:

- Stage in the selection of the statistical technique.
- Blackboard (intermediate information or result)

(Conclusions
Vague Results
Assumptions
Contributions)

- Justification of decisions regarding moves made in the statistical method selection.

**Main Task.** The main task:

- Controls the user-dialog with the SESYS via the user interface during consultation, and

- Terminates the session after a successful consultation or as a result of other termination criteria.

The main task of SESYS is implemented in the form of a loop, which includes the following:

- Supplying user information and requesting user input;

- Partial updating of the static and dynamic statistical knowledge base;

- SESYS's decisions (conflict resolution) regarding the next step, and renewed updating of the statistical knowledge bases (calling the statistical inference mechanism); and

- Terminating or re-entering the loop.

Modular programming of SESYS is the process of breaking it into components and developing each component separately. In Turbo Prolog parlance, the total product of SESYS that is designed and built is a *project.* Each component is Turbo Prolog program file containing information used
by SESYS. Each component of SESYS is developed and tested individually. After all, the modules have been constructed and checked, they are linked into a single executable project file that is SESYS.

3.3 Proposed Algorithm for the SESYS Design

The basic steps in the algorithm proposed for SESYS Design through Modular Programming are outlined below.

**Step 1: Deciding what is global data in SESYS**

Programs that are linked into SESYS must share information, call each other into execution, and interact with the same statistical data files. Data variables, constants, and data files that need to be accessible to more than one module is called *global data*. Information that can be confined to a specific module is called *local data* to that module. Early in the design of SESYS, we must decide what data will be global and what will be local. Data used in modules, from the main module to quit module, with the exception of System and ‘C’ Interface module (Figure 3.2) will normally be global. Those used in the System and ‘C’ Interface module will be local, because the software evoked is of ‘C’ language and not Turbo Prolog, which is the language used in all other modules.

**Step 2: Designing SESYS Modules**

The next step in SESYS modular programming is the design of the individual modules. This step has two phases. First, we determine how to break the total SESYS into modules. Then we design how each individual module will work.

**Step 3: Writing and Testing the SESYS Modules**

Turbo Prolog codes that make up SESYS must be written and debugged. Essentially, the processes in this step are the same as those we use for a stand-alone Turbo Prolog program.
Step 4: Preparing SESYS Modules for Implementation

During the process of testing and debugging individual modules in isolation from the rest of the SESYS, we have to make allowances for the absence of other modules. We have to create test data that is local to our program at one stage. But in an interconnected system, the database becomes global. Because of this, some modules require modification when they are linked into the final design of the SESYS.

Step 5: Compiling and Linking SESYS Modules

Turbo Prolog includes an integrated linker-compiler that makes the compiling and linking process easy. In Turbo Prolog, which has its own peculiarities and syntax, it (compiling and linking) is a single command. But understanding how this step works, what can go wrong, and how to make corrections are still important for us in designing the SESYS. There are defect-reduction strategies that should be kept in mind in the design and development of SESYS [25].

Step 6: Using SESYS to Determine Statistical Method and Procedures

In this step, the SESYS designed is used in a problem context. This context is that of a given problem in information management, for which SESYS provides the solution by querying the user and determining, based on his/her answers, the statistical procedure that should be used to compute the problem. Figure 2.1 shows the step-by-step methodology through which the statistical procedure could be found.

Developing systems with well-defined component interfaces offers opportunities for effective reuse and maintenance (Nuseibeh, 2001). But understanding how this step works, what can go wrong, and how to make corrections are still important for us in designing SESYS.
The method of algorithmic problem solving is not usually discussed in algorithm textbooks. In particular, the implementation part of the method/process is almost always ignored, probably under the assumption that it has been done in preceding programming courses (Levitin, 2000).

One very common goal is to prepare the SESYS software for change. This is important to minimise efforts required for maintenance and especially when an architecture for a family of products is designed (Bachmann, 2001).

3.4 SESYS and Modules

The SESYS is made up of six modules. Figure 3.2 depicts SESYS and its components.

i) The first element of SESYS is the “SESYS.PRJ”. This file contains the names of all modules that make up the SESYS. Since SESYS program is to be made up of six modules, Turbo Prolog requires a project definition that specifies names of the modules involved. The SESYS is associated with a unique project definition.

To setup SESYS called SESYS.PRJ containing reference to these six modules, the following steps are adopted:

- Choose Edit PRJ file from the Options menu.
- Type the name of the file SESYS in the input box, and then press Enter.
- Turbo Prolog will invoke the editor and create the file SESYS.PRJ.
- SESYS modules are sequenced with the use of plus sign, as in main+skbm+techalt+interface+reason+quit+
Figure 3.2: SESYS and its Components

Each module is specified only by its first name followed by a plus (+) sign or a new line or both. The file name of the project definition file (SESYS.PRJ) becomes the name of the statistical expert system project for information management.

ii) When we are through editing, we press F10 to save the file and return to the main menu.

The project definition file SESYS.PRJ is placed on the current directory, which is also where the .PRO files are located. SESYS.PRJ has two purposes:

1. The contents of SESYS.PRJ files are used at link time; the system inserts the names of the modules from the project definition file (SESYS.PRJ) into the link command.
2. The computer uses the project name during compilation to identify a symbol table shared by all modules in SESYS.PRJ. The symbol table is stored in a file in the OBJ directory; the name of the symbol table is same as the project name, with a .SYM extension (that is SESYS.SYM). This file is automatically generated and updated during compilation.

For Turbo Prolog to identify the SESYS symbol table, we must give the name of SESYS in each module. To do this, we insert the project compiler directive at the very top of the module; this directive takes the form project "SESYS".

Global Definitions of SESYS. To force Turbo Prolog program modules within SESYS to transfer information through predicate calls, we must declare the predicates as global. And because domains must be defined in Turbo Prolog before corresponding predicates will work, we must also declare appropriate domains as global.

Global Declarations of SESYS. By default, all names in SESYS module are local. Turbo Prolog programs communicate across module boundaries using the predicates defined in the global predicates and global database sections. The domains used in these global sections must be defined as global domains, or else they must be standard domains.

All the modules of SESYS need to know exactly the same global database and global domains. The easiest way to achieve it is by writing all global declarations in one single file, which we may then include in every relevant module with an include directive. We have all our global declarations in a file called GLOBDEF.PRO, and we may include this file in every relevant module by adding the directive

\texttt{include "GLOBDEF.PRO"}

at the top of each module. All global declarations of SESYS must appear before any local declarations.
Global Domains of SESYS. We make a domain global by writing it in a global domains section. In all other respects, global domains are the same as ordinary (local) domains. Note if any global domain definition of SESYS is changed, all modules in SESYS must be recompiled.

First, we should identify all the predicates, which need to pass information back and forth through predicates; that is, the predicates that the modules have in common. Making a list of predicates that appear in all segment of a Turbo Prolog program is an easy and efficient way of determining which predicates must be declared as global.

Open a new file in the Turbo Prolog Editor window so we can set up our global definitions. First, we define the global domain technique.

    global domains technique=symbol.

As we can see, declaring a global domain is identical to declaring a local domain except we add the key word global to the definition. Before proceeding, we save this file under the name GLOBDEF.PRO and leave it in the Editor window.

Global Database of SESYS. We make a database section global to SESYS by preceding the database keyword with the keyword global. We can only give initialising facts for global database in the main module, which is the one containing the goal section. The goal section must appear before the global database clauses in the main module. Note if any global database definition of SESYS is changed, all modules in SESYS must be recompiled.

Global Predicates of SESYS. In Turbo Prolog both predicate sharing and variable sharing are possible. This facility is offered by global predicates. Global predicate declarations differ from ordinary (local) predicate declarations because they must contain a description of the flow pattern(s) by which each given predicate can be called. Each global predicates declaration must follow the scheme.

    gi1_pred(d1, d2, ..., dn) - (f, f, ..., f) (f, f, ..., f)
Where \texttt{gll\_pred} is predicate name, there must be in next line \textit{d1, d2, ..., dn} that are global domains and each \textit{(f; f; ..., f)} group denotes a flow pattern where each \textit{f} is either \textit{i}(input) or \textit{o}(output).

For example, in the following global predicate declaration,

\texttt{index} and \texttt{laspery} are of type domain \texttt{string}, and \texttt{value} is of type \texttt{real}; the arguments to \texttt{index} can either be all bound \texttt{(i,i,i)} or all free \texttt{(o,o,o)}:

\texttt{index\_pred(index,laspery,\textit{value}) \texttt{- (i,i,i)} \texttt{(o,o,o)}}

Note if any global predicate definition is changed, only the modules that refer to that predicate need to be recompiled.

Now all program segments in the Turbo Prolog project SESYS recognise the term \textit{technique} as being a symbol. But there is need to ensure that \textit{statistical\_method} predicate is also known to all program segments.

To accomplish this, we must define the predicate as global. In the same file in which we defined the global domain \textit{technique}, add the following two lines:

\texttt{global\_predicates}
\texttt{statistical\_method(technique,technique) \texttt{- (i,o),(o,i),(i,i),(o,o).}}

As we can see, declaring a global predicate is similar to declaring a predicate in a single Turbo Prolog program, with two exceptions. First, the word \textit{global} appears in the line where the segment begins. Second, the definition of the predicate is followed by some combinations of the letters \textit{i} and \textit{o}. These combinations of \textit{i} and \textit{o} represent Turbo Prolog flow \textit{patterns}.

Note, before Turbo Prolog can compile and link the modules of SESYS, the following conditions must be fulfilled:

1. Each module must be headed with the \textit{project} compiler directive and the \textit{include} directive for the global declarations.

\texttt{Project \textquotedblleft \texttt{sesys}\textquotedblright}
\texttt{include \textquotedblleft \texttt{globdef.pro}\textquotedblright}
2. One (and only one) module must contain a goal section. This module is called the main module.

The modules of SESYS can be compiled separately into .OBJ files. When we choose Compile/EXE file for a module, the SESYS will automatically be linked together (provided the other modules have all been compiled into .OBJ files). When we choose Compile/Project, all the modules named in the SESYS.PRJ file will be compiled and linked together.

3.5 Program Segments of SESYS Combining Modules

In this section, the steps involved in combining six modules of SESYS into a single program are given. The six modules are:

MAIN.PRO, SKBM.PRO, TECHSLT.PRO, INTRFACE.PRO, REASON.PRO, QUIT.PRO, the project is called SESYS, and the necessary global declarations are saved in the file GLOBDEF.PRO.

Step 1: Create a Project File by using Options/Edit PRJ File, then edit the content to appear as follows:

/* SESYS.PRJ */
main+skbm+techslt+intrface+reason+quit+

Press F2 to save the project file (SESYS.PRJ).

Step 2: Create, Edit and Save the global declarations file, GLOBDEF.PRO, so that it looks like:

/* GLOBDEF.PRO */
global domains
  name=string
global predicates
  statistical_method(name) - (i)

Step 3: Create, Edit and Save the main module file, MAIN.PRO, so that it looks like this:

/* MAIN.PRO */
project "sesys"
include "globdef.pro"
predicates
classify
goal
classify
classes
   classify :-
      clearwindow,
      write("welcome to sesysver 1.0"),
      nl,nl,nl,
      write("do you know the statistical method ? ( y/n)"),
      readin (answer1),
      get_1(answer1).
get_1(answer1):-
   answer1 = 'y',
   invoke_spacc.
/* system interface to invoke and spss....*/
get_1(answer1):-
   answer1 = 'n',
   invoke-sesys.
identify.
identify :-
   clearwindow,
   write("welcome to sesysver 1.0"),
   nl,nl,nl,
   write("are you interested in getting statistical advice ? y/n"),
   readin (answer),
   welcome(answer).
identify :-
   clearwindow,
   write("thanks for using sesys software"),
   nl,nl,nl,
   write("you are always welcome")

Step 4: Create, Edit and Save the module SKBM.PRO, as follows:

   /* SKBM.PRO */
   projects "SESYS"
   include "globdef.pro"
   clauses
      welcome(answer) :-
      answer = "y",
      write("have a nice day"),
      sound(100,200),
      query.

Step 5: Create, Edit and Save the module query as follows:

   /* TECHSLT.PRO */
   project "SESYS"
   include "globdef.pro"
   clauses
      query:-
      makewindow(1,7,7,"",0,0,25,80),
      go,
      c_interface.

Step 6: Create, Edit and Save the module C_INTRFACE.PRO, as follows:

   /* INTRFACE.PRO */
   projects "SESYS"
Step 7: Create, Edit and Save the module REASON.PRO, as follows:

```prolog
/* REASON.PRO */
projects "SESYS"
include "globdef.pro"
clauses
  reasoning :-
    makewindow(...),
    explanation_component,
    check_exit.
```

Step 8: Create, Edit and Save the QUIT.PRO, as follows:

Before calling routines and functions written in 'C' language, we need to declare them as external predicates in Turbo Prolog. We also need to understand the correct calling conventions and parameter pushing sequences, and we need to know how to name the different flow variants of our external predicates.

Step 9: Select the project from the Compile menu, give SESYS, and press Enter.

The Turbo Prolog system will automatically compile and link the files \texttt{INIT.OBJ}, \texttt{QUIT.OBJ}, \texttt{REASON.OBJ}, \texttt{INTERFACE.OBJ}, \texttt{TECHSLT.OBJ}, \texttt{SKBM.OBJ}, \texttt{MAIN.OBJ}, \texttt{SESYS.SYM} and \texttt{PROLOG.LIB} to give the executable program file \texttt{SESYS.EXE}. The Turbo Prolog modules (the .PRO files) and the SESYS.PROJ file should reside in the current directory.

We can use TLINK, the PC_DOS linker (version 3.20 or later), or the PLINK86 overlay linker (version 1.48 or later) from Phoenix Software Associates, Ltd., as external linkers. When we use PLINK86, each module in SESYS can be placed in separate overlays. TLINK is included as part of our Turbo Prolog 2.0 distribution package.
**The Driver Routine.** We create the bulk of SESYS programs as we write and test each module. But to get things started and to gain an appreciation of the overall structure of the SESYS, we write the main driver routine, which governs the SESYS execution. This driver routine contains the *goal* statement. The goal is then defined as being satisfied by interaction with the user, disk files, or other devices external to the program. The main driver routine is as follows:

```pro
project "sesys"
include "globdef.pro"
predicates
  start
goal
  start.
clauses
  start :-
    method_known_unknown,
    get_response,
    check_method(technique),
    time_to_stop.
```

The first line of this module refers to the SESYS.PRJ file we have just created using the Files Module list option. This same line must be included in every module in the SESYS Project. It is used during compilation to ensure the correct production of the symbol table. We have already discussed global predicates and their inclusion in SESYS project. Any module that uses globally defined predicates must contain the second line in its listing.

The goal called *start* is made up of four clauses whose names do not appear in the predicates section of the driver routine. Each of these appears in a different module of SESYS, as we may identify from their names. Because these predicates appears in this main driver routine and must also appear in individual modules, they are clearly global and must be defined in a file called GLOBDEF.PRO. This file can be built as we design the SESYS system.

Note that the inclusion of these clauses implies that each of the sub-modules of SESYS project has a main clause that is the first one involved
when the predicates contained in that module are executed. In other words, each module must be designed so that its processing is initiated by calling the clause whose name appears in the main driver routine MAIN.PRO.

**Quit Module.** The quit module is as follows:

```prolog
projects "sesys"
include "globdef.pro"
clauses
    time_to_stop :-
        clearwindow,
        write("Thanks for using sesys ")
        write("now it is time to close the consultation").
```

We type this program into Turbo Prolog and save it as QUIT.PRO. Add a predicate declaration to the GLOBDEF file for the new `time_to_stop` predicate.

**Turbo Prolog and SESYS Modular Programming.** Turbo Prolog’s superb development environment includes two features designed primarily to facilitate modular programming (Shafer, 1987).

**SESYS Module Lists.** Turbo Prolog includes a definition of a specific type of file called a *module list*. In Turbo Prolog, the *Files menu* option offers a choice called *Module list*. When we select this option, Turbo Prolog displays a list of all files on the current directory with a .PRJ (for project) extension.

A module list contains the names of all modules of SESYS. The SESYS takes its name from the name of the module list. A file called SESYS.PRJ, for example, denotes a project called SESYS.

Each file in the SESYS (project) is listed without its file extension (which must be .PRO), and all are connected by a plus sign. The plus sign at the end of the module list is required.

**Menu-Driven Compilation and Linkage.** After we define SESYS’s module list file and complete the modules for the construction of SESYS design, the *Options menu* alternatives come into play.
Before we compile the SESYS, select the Project (all modules) option from this menu. When we do, nothing happens until we use the “C” option to order Turbo Prolog to compile SESYS. At that point, the SESYS’s module list is called into memory and used as a map to tell Turbo Prolog what files to compile. If the compilation proceeds without errors, an executable file with the .EXE extension is created. Linkage of the modules along with any external files explicitly included in the programs is automatic.

Note that to work correctly, the module list for SESYS must have a .PRJ extension and must reside in the directory defined as the .OBJ directory. The other modules must be located in the .PRO directory. Use the Setup menu option to change these directories if necessary before compiling.

Statistical Facts and Goals. Turbo Prolog is an Objected-Oriented Computer Language. It is concerned with objects and relationships of these objects with other objects. When grouped together as logical expressions, they form a database from which information can be extracted (Nebendahl, 1988). For example, the following logical expressions constitute a small database that classifies statistical methods:

1. Kruskal-Wallis test is a non-parametric test.
2. The pivotal method is an interval estimator.
3. Secular trend is time series.
4. Laspeyres method is an index number.
5. Paasche technique is a weighted aggregate index.
6. Christie’s technique is decision-tree analysis.
7. Studying the effect of two or more independent variables on a dependent variable is multiple regression.
8. Multinomial experiment is Chi-square test.
9. Test statistic for a goodness-of-fit test is chi-square test.
10. Test statistic for a test of independence is chi-square test.
11. Test statistic for a test of hypothesis about $\sigma^2$ is chi-square test.

If we give this information to Turbo Prolog, it would see Kruskal-Wallis, Secular-trend, Paasche and Christies as objects. It would see the relationship as is-a-nonparametric-test, is-time-series, is-weighted-aggregate-index and is-decision-tree-analysis, respectively.

A statistical database (SDB) is one of the main ingredients of a Turbo Prolog program. For the statistical database to be useful, it would have to include more about the known statistical techniques and their classifications. It would be a large statistical database indeed. In addition to objects, the statistical knowledge base provides statistical rules; these are represented in the form:

**If premise then conclusion and/or action**

In the premise part, questions are asked about the logical links between the characteristics of the objects. In the conclusion part, new facts and characteristics are added to the statistical knowledge base and/or actions are executed. This is frequently referred to as rule based programming. The following questions arise with regard to creating a statistical knowledge base:

- Which objects will be defined?
- What are the relationships between the objects?
- How are the rules formulated and processed?
- Is the statistical knowledge base complete with regard to solving the statistical problem?
- Is the statistical knowledge base consistent?

These are questions that must be answered by the knowledge engineer, who works with the statistician to the extent necessary.

3.6 **Statistical Interface Mechanism (SIM)**

The statistical interface mechanism represents the logical unit by means of which conclusions are drawn from the statistical knowledge base.
according to a defined statistical problem-solving method, which simulates the problem-solving process of statisticians. A conclusion is reached by applying a statistical rule to the existing statistical facts.

Example:

A rule states : IF f1 and f2 and f3, THEN R
The facts are : f1, f2 and f3

Where

f1, f2, f3 and R are statistical facts and
f1 = More than one independent variable.
f2 = Independent variables are used to estimate the dependent variable.
f3 = Increase of the accuracy of the estimate of the dependent variable.
R = Multiple regression methods.

f1, f2 and f3 are precisely those facts specified in the IF part of the statistical rule, that the conditions for the applicability of the statistical rule are given. Applying the statistical rule means: from the statistical facts f1, f2 and f3, conclude the statistical fact R. A statistical fact exists in SESYS if it is contained in the statistical knowledge base (Figure 3.3).

The statistical facts specified in the IF part of the rule are called premises, and the statistical facts contained in the THEN part is called conclusion when a statistical rule is applied to any statistical fact(s), we say it fires. The firing of a statistical rule results in the entry of the new (inferred) statistical fact into the statistical knowledge base. The functions of the statistical interface mechanism include the following:

- To determine which actions are to be executed between the individual part of the SESYS, how they are to be executed, and in which sequence.
- To determine how and when the statistical rules will be processed and, if applicable, to select which statistical rules will be processed.
- To control the user’s dialogue.
The statistical rule-processing mechanism chosen, that is, the search strategies implemented are of primary importance in determining the performance of the entire system. Different statistical problems or different types of statistical problems naturally require different types of statistical inference mechanisms. The statistical inference mechanism links sequences of the said production rules together by means of backward or forward chaining to extrapolate conclusions from the statistical knowledge base (McCabe, 1993). The statistical inference mechanism must be ‘adapted’ to the statistical problem to be solved.

**SESYS Explanation Component.** The statistical solutions determined by SESYS must be reproducible, both by the knowledge engineer during the test phase and by the user. The statistician, of course, can only verify the statistical solutions. There are advantages in knowing, at any point during the SESYS’s session, how far along is the system into processing the statistical problem.

- Which questions are being asked, and why?
• How has the SESYS arrived at intermediate solutions?
• What characteristics do the individual objects have? and so on.

Despite the repeated emphasis on the importance of the explanation component, it is very difficult to meet all the requirements, it is very difficult to meet all the requirements of a good explanation component, and most attempts so far have been marginally successful.

On demand, SESYS can explain end results and reasoning direction (Butler et al., 1988). Many explanation components represent the consultation steps in graphic form. Furthermore, the explanation components attempt to deal with the statistical problem by backtracking through the statistical solution path. The existing explanation components may well be adequate for the knowledge engineer, who is very familiar with the specified EDP environment, and in some cases they may suffice for the experts; but for the user, who often has little or no EDP experience, the existing explanation components are still unsatisfactory.

**Statistical Knowledge Acquisition Component.** The most difficult phase of SESYS development is statistical knowledge acquisition, in which one or several domain specialist(s) interact in order to encode problem solving statistical knowledge (Lutz, 1993). The work of the knowledge engineer gets considerable support from a good knowledge acquisition component. The knowledge engineer can then concentrate mainly on structuring the statistical knowledge, and does not have to devote as much attention to programming. The statistical knowledge acquisition component should have the following characteristics:

• Statistical knowledge, that is, statistical rules, statistical facts, and relationships between statistical facts, must be easy to enter.
• Easy-to-understand methods of representing all statistical information contained in the statistical knowledge base.
• Automatic syntax checks.
Continuous access to the underlying programming language.

How the individual requirements are implemented depends on the programming language Turbo Prolog and on the hardware of the computer machine. The most important step in building the SESYS is statistical knowledge acquisition followed by its representation for machine use and interpretation. After building statistical knowledge base, statistical inference engine and user interface are created so that statistical inferences are drawn and made available to the outside user (Mishra, 2001). The expert should be somewhat familiar with the statistical knowledge acquisition component, so that he/she can make simple changes himself/herself.

**Structuring Statistical Knowledge Representation.** Processing and administration of statistical knowledge in SESYS requires that the statistical knowledge be formalised and statistical structured. Generally, the statistical knowledge is made available in the form of case descriptions obtained from interviews with statisticians, or by observing representative activities or through text materials. Formal methods of statistical knowledge representation are forms of logic; for example, predicate logic.

Statistical knowledge representation procedures have been developed which can efficiently support the structuring and processing of statistical knowledge. These include the following:

*Production Rules:* Based on predicate logic, these describe statistical knowledge in the form of "IF......, THEN......" rules.

This procedure, whether in the representation or the processing of statistical knowledge, is based on the form of

*Predicate Calculus:* Logical deduction based on propositions: a logical conclusion can be drawn if certain conditions are met. The solution can assume the value "true" or "false".

The following section discusses statistical knowledge representation method based on the production rules for SESYS.
3.7 SESYS Production Rules

In SESYS, the most comprehensive form of statistical knowledge representation is based on production rules, which are descriptions of condition-dependent actions. A single production rule is understood to be a single item (chunk) of statistical information. These chunks of information are the smallest units of statistical information in the entire system of SESYS.

It is apparent that expert can best formulate statistical knowledge in the form of "IF.... THEN...." rules. This is probably the reason why most expert systems today, at least the most successful ones, are based on production rules (Patterson, 2001).

In the implementation of SESYS based on production rules, it soon becomes necessary to process uncertain statistical knowledge. In these cases, results can be weighted with the help of certainty factors (confidence factors). These factors are not to be interpreted in the same sense as confidence factors used in probability and statistics; rather, they are arbitrary weighting factors generally in the range of \(-1\) to \(+1\). In this context, \(-1\) could mean "definitely not," \(-0.5\) "probably not", \(0\) "unknown", \(+0.5\) "probable", and \(+1\) "certain". The value range is continuous in the selected statistical solution space.

Example for SESYS Production Rules. From SESYS, a statistical expert system for information management used in the determination of multinomial experiment is:

If

1. It consists of \(N\) independent trials (repetitions), and
2. Each trail results in one of \(k\) possible outcomes (or categories), where \(k>2\), and
3. The trials are independent, and
4. The probabilities of the various outcomes remain constant for each trial.

Then

There is a suggestive evidence (0.9) that the identity of the statistical method is multinomial experiment.
SESYS is a production system, since, systems that are set up on the basis of production rules are called production systems. The architecture of the SESYS, with production rules, is shown in Figure 3.4.

The most important component of SESYS is the statistical inference mechanism. This mechanism controls the processing and the selection of statistical rules. A good statistical inference mechanism is characterised by efficient methods and conflict-resolving strategies when selecting a statistical rule from a number of possible statistical rules. The SESYS keeps a statistical rule base with rules of the form \( x_1 \ldots x_n \rightarrow y_1 \ldots y_m \) where the \( x_i \) represent statistical facts and the \( y_i \) represent actions to be performed (Goebel, 1998).

![Figure 3.4: Architecture of SESYS with Production Rules](image)

**Statistical Facts and Rules and Logical Operators.** Turbo Prolog can also solve goals with more than one predicate. A compound goal consists of two or more goals, called subgoals, joined by the logical operators AND or OR.

**The AND Operator.** The form of a compound goal with the AND logical operator is:
**AND Logical Operator**

If A is true AND If B is true

Subgoal-1 Subgoal-2

The compound goal is True if both Subgoal-1 and Subgoal-2 are True.

To successfully satisfy a compound goal with the logical operator **AND**, each of the subgoals must be satisfied (Cafolla, 1989).

i. **Multinomial Experiment IF**

   *It consists of n independent trials (repetitions) AND*
   *Each trial results in one of k possible outcomes (or categories), where k>2 AND The trials are independent AND*
   *The probabilities of the various outcomes remain constant for each trial.*

ii. **Multiple Regression Techniques IF**

   *More than one independent variable AND*
   *Independent variables are used to estimate variable AND*
   *Increase the accuracy of the estimate of the dependent variable.*

**The OR Operator.** We can also form a compound goal with the connective word **OR**. The form of a compound goal using the OR operator is:

**OR Logical Operator**

If A is true **OR** If B is true

Subgoal-1 Subgoal-2

This compound goal is true if either Subgoal-1 or Subgoal-2 is true.

To satisfy a goal with the logical operator **OR**, only one of the subgoals need be satisfied (Cafolla, 1989).

i. **Time Series IF**

   *To see what pattern of changes take place over time in an event we are observing* *(OR)*
To compute the seasonal indices and apply to data to find the de-seasonalised value.

To develop the secular trend line by applying the least squares method or the parabolic curve method using the de-seasonalised data or to estimate the cyclical variation around the trend line using the percent of trend or the relative cyclical residual method or an attempt to identify and isolate the cause of any irregular variation are the problems that could be dealt with using time series.

**Statistical Rules.** When we add rules to a Statistical Database (SDB), the database becomes a Statistical Knowledge Base (SKB) ready to answer questions about the information it contains. Similarly, when we add statistical rules to a statistical database, the statistical database becomes a statistical knowledge base ready to answer statistical questions.

In logic, the **AND** and **OR** operators are important. When either of these operators is used with two or more logical expressions, the result is always a logical expression. In Turbo Prolog logical expressions are represented by clauses in the statistical database. To solve complex statistical problems, Turbo Prolog must be able to **infer** information not specified in the statistical database.

To do this, Turbo Prolog uses the rules of logic. In Turbo Prolog, statistical rules are logical expressions that express relationships between statistical facts.

New definitions can be expressed through the use of rules. For example, assume we know the following statistical facts (Table 3.1): Using these statistical facts, we can derive a statistical rule to express the conditions under which the SESYS will suggest the Laspeyres Index.
Table 3.1: Statistical Facts and SESYS Clauses

<table>
<thead>
<tr>
<th>Statistical Fact</th>
<th>SESYS Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prices in the current year is known</td>
<td>Known(prices_in_current_year).</td>
</tr>
<tr>
<td>Prices in the base year is known</td>
<td>Known(prices_in_base_year).</td>
</tr>
<tr>
<td>Quantities sold in the base year is known</td>
<td>Known(quantities_sold_in_base_year).</td>
</tr>
</tbody>
</table>

An example for Horn clause and statistical rule is shown below in Table 3.2:

Table 3.2: Horn Clauses and Statistical Rules

<table>
<thead>
<tr>
<th>Horn Clause</th>
<th>Statistical Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. aspeyes index if</td>
<td>Laspeyresindex :-</td>
</tr>
<tr>
<td>The prices in the current year is</td>
<td>Known(prices_in_current_year).</td>
</tr>
<tr>
<td>known and the prices in the base</td>
<td>Known(prices_in_base_year).</td>
</tr>
<tr>
<td>year is known and the quantities</td>
<td>Known(quantities_sold_in_base_year).</td>
</tr>
<tr>
<td>sold in the base year is known.</td>
<td></td>
</tr>
</tbody>
</table>

Notice that the statistical rule looks like any other clause. In fact, statistical rules are part of the clauses segment in SESYS program and treated the same way as any other statistical fact. Assuring the above statistical rule is part of a Turbo Prolog statistical knowledge base, Turbo Prolog can infer whether or not we should consider the Laspeyres index by matching the \( \text{known}(\text{prices} \_\text{in} \_\text{current} \_\text{year}), \text{known}(\text{prices} \_\text{in} \_\text{base} \_\text{year}) \) and \( \text{known}(\text{quantities} \_\text{sold} \_\text{in} \_\text{base} \_\text{year}) \) parts of the statistical rule with similar statistical facts in the statistical knowledge base.

Given the goal: \text{Goal} : laspeyresindex

SESYS responds: Yes

1 Solution.

The answer 'Yes' is inferred by Turbo Prolog, because each of the conditions in the statistical rule can be matched with a statistical fact in the statistical knowledge base.
Parts of a Statistical Rule. A statistical rule has two parts (1) a conclusion, and (2) the conditions that must be met to satisfy the conclusion (Figure 3.5). The conclusion is called the head of the statistical rule, and the conditions that must be met are called the body of the statistical rule. In this example, the head of the statistical rule is \( \text{laspeyreindex} \), and the body of the statistical rule is

\[
\begin{align*}
\text{known(} & \text{prices\_in\_current\_year)}, \\
\text{known(} & \text{prices\_in\_base\_year)} \quad \text{and} \\
\text{known(} & \text{quantities\_sold\_in\_base\_year).}
\end{align*}
\]

The statistical rule is true (succeeds) because the conditions to satisfy the conclusion succeed.

![Figure 3.5: Parts of a Statistical Rule](image)

Turbo Prolog Clauses. When we write a Turbo Prolog program, the first segment we write is the statistical knowledge base. We write the statistical knowledge base in the form of Turbo Prolog clauses. These clauses come in two forms: as statistical facts and as statistical rules.

A Turbo Prolog statistical fact declares a relationship between one or more objects. A Turbo Prolog statistical fact is always true. A Turbo Prolog statistical rule is a conditional statistical fact that depends on the truth or
falsity of one or more statistical facts. A statistical rule can be either true or false.

The remainder of this chapter is concerned with how to write statistical facts as Turbo Prolog Clauses, how to ask questions in Turbo Prolog, and how the designer makes Turbo Prolog answer these questions.

**Logic.** Logic, a method of reasoning formalised by the ancient Greeks, characterises ways of thinking that arrive at conclusions by following an orderly set of statistical rules. As a formal way of manipulating statistical facts, the statistical rules of logic can be described in mathematical terms. Turbo Prolog uses logic to answer questions and to solve statistical problems.

In logic, a **conclusion** is reached by using statistical facts. Consider this classic example.

\[
\text{All nonparametric methods are used to reach conclusions about population when the shapes of their distribution are unknown.}
\]

\[
\text{And}
\]

\[
\text{Kruskal-Wallis test is a nonparametric method}
\]

Therefore,

\[
\text{Kruskal-Wallis test is used to reach conclusions about populations when the shapes of their distribution are unknown}
\]

The conclusion is the result of *valid reasoning*. Reasoning is considered valid if the statistical facts support the conclusion. If possible, the statistical knowledge must be reorganised according to the required reasoning style (Riesbeck, 1984).

### 3.8 A Statistical Logic Machine (SLM)

Our Statistical Logic Machine (SLM) contains only statistical facts that we provide with and the statistical rules of logic. This statistical
knowledge base, consisting of statistical facts and rules, represents the total sum of what is known by the SLM.

We imagine that the SLM speaks English and that it can answer questions about statistical facts from its statistical knowledge base. When asked a question, the SLM attempts at answering the question using the statistical facts and statistical rules in the statistical knowledge base. If no statistical facts exist in the statistical knowledge base to answer the question, it simply assumes the answer is 'No'.

Notice that each statistical fact is patterned according to a common form. The form is:

X is a member of the class Y. In this form, fact - 1 is interpreted as: Multinomial experiment is a member of the class of Chi-square test. Each succeeding statistical fact can be expressed in this form. Now that the SLM has some statistical facts stored in it, let us ask some questions. Recall that the SLM bases its answers only on the statistical facts stored in it.

Here is a picture of the SLM with a listing of the statistical facts in its statistical knowledge base (Table 3.3). Here are some examples:

<table>
<thead>
<tr>
<th>Question:</th>
<th>is multinomial experiment chi-square test?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLM Answer:</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question:</th>
<th>is test of homogeneity chi-square test?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLM Answer:</td>
<td>Yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question:</th>
<th>is Kruskal_Wallis test chi-square test?</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLM Answer:</td>
<td>No</td>
</tr>
</tbody>
</table>

For the third question, the SLM says "No" because it does not contain this statistical fact in its statistical knowledge base. The SLM assumes that all statistical facts are true. If we add some false statements to the statistical knowledge base, or even some opinions, the SLM still consider them as statistical facts.

The point of all this is that the quality of the answers derived from a statistical knowledge base is directly dependent upon the quality of the statistical facts and statistical rules it contains. The SLM cannot differentiate
a true statistical fact from an erroneous statistical fact. Neither can Turbo Prolog.

**Table 3.3: Statistical Logic Machine**

- **Fact - 1.** A multinomial experiment is Chi-square test
- **Fact - 2.** Observed and expected frequencies determine Chi-square test
- **Fact - 3.** Degree of freedom for a goodness-of-fit test is given by Chi-square test
- **Fact - 4.** Test statistic for a goodness-of-fit test is chi-square test
- **Fact - 5.** Degrees of freedom for a test of independence are given by Chi-square test
- **Fact - 6.** Test statistic for a test of independence are given by Chi-square test
- **Fact - 7.** Expected frequencies for a test of independence are given by Chi-square test
- **Fact - 8.** A test of homogeneity offered is Chi-square test
- **Fact - 9.** Sampling distribution of \((n-1)s^2/\sigma^2\) is Chi-square test

**Turbo Prolog and Logic.** Turbo Prolog works almost the same way as our imaginary logic machine. It has a built-in processing device called inference engine. The inference engine, the heart of Turbo Prolog, solves statistical problems by making statistical inferences from the statistical knowledge base. It attempts to match the question with the statistical facts to arrive at conclusions. Turbo Prolog's inference engine guides all of its computing actions.

Now let us take a look at how Turbo Prolog works. First, it must have a statistical knowledge base to which questions may be posed. Second, it must allow us a manner of asking these questions. Turbo Prolog calls questions as goals, Let us look at the statistical knowledge base (Table 3.4). We can use these facts to answer questions. If the question is, 'Is sign test non-parametric'? then you could answer "yes". Think about how we will arrive at this answer. The steps we would use could be:

- Look at the question (is paasche method an index number?)
- Start at the top of the statistical knowledge base (fact-1)
- Can fact -1 answer the question?
- If it can, give the answer as 'Yes'.
- If not, go to the next fact on the statistical knowledge base.
Continue until either a match is found or the end of the statistical knowledge base is reached.

Statistical facts must be expressed in a way that Turbo Prolog can understand. They are expressed as a relationship between one or more objects. This special way of expressing statistical facts is called the **predicate** form. Expressing statistical facts in the predicate form is easy. It is like this:

**Table 3.4: Example of Statistical Facts for Index number**

| Fact - 1: Kruskal-Wallis test is a non-parametric test |
| Fact - 2: Sign test is a non-parametric test |
| Fact - 3: Laspeyres method is an index number (fact from index number) |
| Fact - 4: Paasche method is an index number |
| Fact - 5: Fixed weight aggregates method is an index number |

\[
\text{relationship}(\text{object1, object2, .... objectN}).
\]

Predicate \hspace{2cm} Objects

The first part of the clause (relationship) is called the predicate. It represents the relationship among the objects. The objects, sometimes called arguments, are the objects of the relationship (Cafolla, 1989).

### 3.9 Example for Index Number Construction

The statistical fact that is Laspeyres index depends upon prices in the current year; prices in the base year; and quantities sold in the base year can be written as:

\[
\text{depends upon(laspeyres_index, prices_in_the_current_year, prices_in_the_base_year, quantities_sold_in_the_base_year).}
\]

This time the statistical facts are expressed as Turbo Prolog clauses (Table 3.5).
Table 3.5: Logic Facts and Turbo Prolog Clauses

<table>
<thead>
<tr>
<th>Logic Facts</th>
<th>Turbo Prolog Clauses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kruskal-Wallis tests is a non-parametric</td>
<td>Nonparametric(kruskal_wallis).</td>
</tr>
<tr>
<td>Sign test is a non-parametric</td>
<td>Nonparametric(sign_test).</td>
</tr>
<tr>
<td>Laspeyres method is an Index number</td>
<td>indexnumber(laspayres_method).</td>
</tr>
<tr>
<td>Paasche method is an Index number</td>
<td>Indexnumber(paasche_method).</td>
</tr>
<tr>
<td>Fixed weight aggregates is index number</td>
<td>Indexnumber(fixed_weight_aggregates_method).</td>
</tr>
</tbody>
</table>

Notice that the translation from logical facts to Turbo Prolog clauses follows regular rules.

**Rules for Turbo Prolog Clauses.**

1. Names of objects and predicates begin with a lowercase letter.

2. The lowercase letter can be followed by any number of characters, including letters, numbers or the underscore character[_]. For example, we could rewrite the clause *nonparametric(kruskal_wallis)* as *nonparametric_is(kruskal_wallis)*. We have connected non-parametric to 'is' by an underscore character. Turbo Prolog treats it as one word.

3. Clauses must end with a period. Other terms for the ending period are full stop symbol or termination symbol.

In many cases, predicates are verbs and the objects are nouns. This is not a hard and fast rule, but it may help us to formulate statistical facts that have some meaning to us. Not that Turbo Prolog cares about the meaning; it is only interested in pattern matching.
How to Ask Questions in Turbo Prolog? Now that we know how to express statistical facts as Turbo Prolog clauses, we will next see how to ask questions in Turbo Prolog. This part is easy because we already know how to ask questions. Turbo Prolog uses the same form for question as it uses for statistical facts. The Turbo Prolog term for question is goal. There is a difference between statistical facts and goals. The difference is not in how they look, but in how they work. Shown below are goals to be used in the statistical knowledge base (Table 3.6). Note that Turbo prolog clauses in Table 3.5 and Turbo Prolog forms here are identical.

Table 3.6: Statistical Knowledge Base (SKB)

<table>
<thead>
<tr>
<th>Logical Expressions</th>
<th>Turbo Prolog Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kruskal-Wallis test is nonparametic.</td>
<td>Nonparametric(kruskal_wallis).</td>
</tr>
<tr>
<td>Sign test is nonparametric</td>
<td>nonparametric(sign_test).</td>
</tr>
<tr>
<td>Laspeyres method is an Index number</td>
<td>indexnumber(laspeyres_method).</td>
</tr>
<tr>
<td>Paasche method is an index number</td>
<td>indexnumber(paasche_method).</td>
</tr>
<tr>
<td>Fixed weight aggregates is index number</td>
<td>indexnumber(fixed_weight_aggregates_method).</td>
</tr>
</tbody>
</table>

To ask questions directed to this database, we form the questions in the English language. Then, we write them in the predicate form as Turbo Prolog goals. If these goals were to be presented to the database, Turbo Prolog would respond as shown below.

The question, 'Is Kruskal-Wallis test nonparametric?', is translated into the Turbo Prolog clause nonparametric(kruskal_wallis). When we type the goal in the predicate form, Turbo Prolog's inference engine, starting at the top of the statistical knowledge base, searches through the statistical facts until a match is found between the predicate of the goal and the predicate of
the statistical fact. It then compares object of the goal with the object of the statistical fact. If they match, Turbo Prolog then concludes that the goal is true and answers 'Yes'.

If Turbo Prolog's inference engine cannot find a match with the first clause, it continues the search with the next clause. This continues until a match is found or until the end of the statistical knowledge base is reached. If it reaches the end without finding a match, then the Turbo Prolog replies 'No' and the program stops.

**Predicates with More Than One Object.** A predicate clause can have more than one object. For example, we may want to express the statistical fact that degrees of freedom for a goodness-of-fit test depend on the number of possible outcomes (or categories) for the experiment. This statistical fact implies the existence of a relationship between 'degrees of freedom for a goodness-of-fit test' and 'number possible outcomes' \textit{depends on} becomes the predicate of the clause. The objects are:

\begin{itemize}
  \item 'degrees of freedom of a goodness-of-fit test' and
  \item 'number of possible outcomes'
\end{itemize}

There are two ways to express this in Turbo prolog:

1. \textit{depends on}(degrees of freedom of a goodness-of-fit test, number of possible outcomes).
2. \textit{depends on}(number of possible outcomes, degrees of freedom of a goodness-of-fit test).

Both clauses can mean the same thing. The interpretation depends upon us. We might interpret the first clause as \textit{degrees of freedom of a goodness-of-fit test} depends on \textit{number of possible outcomes} and the second clause as \textit{number of possible outcomes} depends on \textit{degrees of freedom of a goodness-of-fit test} (Table 3.7).
The order of the object is not important to Turbo Prolog as long as we are consistent. Turbo Prolog does not attach any meaning to the clauses. That is left to us, the SESYS designers.

Table 3.7: Logical Expression and Turbo-Prolog Clause

<table>
<thead>
<tr>
<th>Logical Expression</th>
<th>Turbo Prolog Clause</th>
</tr>
</thead>
<tbody>
<tr>
<td>H0 is null hypothesis</td>
<td>is(No,null_hypothesis).</td>
</tr>
<tr>
<td>H1 is alternate hypothesis</td>
<td>is(Hi,alternate_hypothesis).</td>
</tr>
<tr>
<td>n is sample sign</td>
<td>is(n,sample_sign).</td>
</tr>
<tr>
<td>Test statistic for a goodness-of-fit test is chi-square</td>
<td>is(test_statistic_for_a_goodness_of_fit_test,chi_square_test).</td>
</tr>
<tr>
<td>Chi-square test depends on observed and expected frequencies</td>
<td>depends_on(chi_square,observed,expected_frequency).</td>
</tr>
<tr>
<td>Variance within samples depends on within-samples sum of squares, number of different samples and number of values in all samples</td>
<td>depends_on(variance_within_samples,within_samples_sum_of_squares,no_of_different_samples,no_of_values_in_all_samples).</td>
</tr>
<tr>
<td>Total sum of squares depends on between-samples sum of squares and within-samples sum of squares</td>
<td>depends_on(Total_sum_of_squares,between_samples_sum_of_squares,within_samples_sum_of_squares).</td>
</tr>
</tbody>
</table>

3.10 Segments of SESYS Program

The SESYS program consists of four segments. These segments are called:

domains
predicates
goals
clauses

In writing the program, we will use three of these segments: The clauses segment, the predicates segment, and the domains segment. Now, we shall not use the goals segment here. We will key in the goals as the program executes.

Goals Segment. In Turbo Prolog, questions must be stated in a form that Turbo Prolog can understand. Turbo Prolog supports both external and internal goals. When the SESYS uses an external goal, Turbo Prolog
executes the program until all possible statistical solutions are found and displayed. The same program with the internal goal to the program will find one statistical solution only and then stop. The internal goals are located within the SESYS program and they are entered into a special segment of the program called the goal segment. This is done just as we would enter the clauses, predicates, and domains segments. The structure of SESYS program that includes an internal goal is shown in Figure 3.6.

The Clauses Segment. The Turbo Prolog facts available in the Statistical Knowledge Base (SKB) can be entered as follows in the clauses segment:

```prolog
clauses
  indexnumber(laspayres_method).
  indexnumber(paasche_method).
  indexnumber(fixed_weights_aggregates_method).
  nonparametric(kruskal_wallis).
  nonparametric(sign_test).
```

Notice that 'period' is required in the clauses segment.

```
domains
  ------------
  ------------
predicates
  ------------
  ------------
goal
  ------------ Internal goal is written here.
clauses
  ------------
  ------------
```

Figure 3.6: Structure of SESYS Program with Internal Goal

Turbo Prolog is a compiled version of Prolog. This means that Turbo Prolog converts the text into binary code before it actually runs. To accommodate the compiler, Turbo Prolog requires two more segments to be added to the program before we could run SESYS. These segments are the predicates segment and the domains segment.
The Predicates Segment. The predicates segment is where we furnish to the Turbo Prolog compiler a picture of each predicate used in the statistical knowledge base. It precedes the clauses segment in the program.

The statistical knowledge base uses two predicates, indexnumber and nonparametric. Thus, in the predicates segment, the actual names of the predicates are included along with a general name for the objects of the predicates. The predicate names in the predicates segment must match the predicates in the clauses segment. However, the object or argument names do not have to match. It is good practice to name the objects in the predicates segment with a word that describes the object’s general classification. In the statistical knowledge base, the objects are names of statistical techniques (Cafolla, 1989).

Two choices would be good: method and technique. We have chosen technique. Other choices would be just as good. A general form of these predicates might look the following:

\[
\begin{align*}
\text{indexnumber} & \text{(technique)} \\
\text{nonparametric} & \text{(technique)}
\end{align*}
\]

Again, these statements define the predicates indexnumber and nonparametric that are used in the clauses segment. Technique represents the objects in the clauses segment.

We enter the clauses segment first. This is because after we have written the clauses segment, the other segments are quite easy to write/key in. Follow the steps below to enter the predicates segment.

\[
\begin{align*}
\text{predicates} \\
\text{indexnumber(technique)} \\
\text{nonparametric(technique)}
\end{align*}
\]

Notice that a ‘period’ is not required in the predicates segment. Refer to Turbo Prolog’s user manual for standard predicates.

The Domains Segments. The next segment is the domains segment. This segment informs Turbo Prolog about data types of the objects used in
the SESYS program. These data types are called domains. We refer to Turbo Prolog's user manual for standard domain types.

The statistical knowledge base uses one class of object, which we have designated as technique. The data type of technique is symbol. Symbols are a grouping of letters that stand for something. Symbols must begin with a lower case letter and must contain no blanks. We enter the domains segment as shown below:

```prolog
domains
  technique=symbol

predicates
  idenumber(technique)
  nonparametric(technique)

clauses
  idenumber(laspayres_method).
  idenumber(paasche_method).
  idenumber(fixed_weights_aggregates_method).
  nonparametric(kruskal_wallis).
  nonparametric(sign_test).
```

**Turbo Prolog Rules.** In logic, the Horn clause uses the following form:

```
conclusion if
  condition 1 and
  condition 2 and
  condition...
```

The conclusion is a true statement if the conditions are met.

**File Operations.** Turbo Prolog allows two kinds of file operations and they are file reading (input), and file writing (output). In fact, Turbo Prolog treats all program input or output as files (Cafolla, 1989). It provides a full complement of file handling predicates. These predicates are used in SESYS design for the following operations:

- Create statistical knowledge base file
- Open file
- Retrieve statistical information from file
- Add statistical information to file.

Turbo Prolog files are classified into two types and they are (i) predefined file; and (ii) user-defined file. These are disk files that we can create to store statistical information (Table 3.8).

**Turbo Prolog Predefined Files.** The predicate used in the ‘opening a statistical file for input’ is enabled to change the input device and the ‘symbolic_filename’ is to name the new input device such as keyboard or com1. Similarly, the predicate used in the ‘opening a statistical file for output’ is enabled to change the new output device and the ‘symbolic_filename’ is to name new output device such as screen or printer or com1. The writedevice predicate works the same way as the readdevice predicate.

**User-defined Output Files.** As far as the user-defined output files are concerned, these files are used to store statistical information on the disk. The predicate used to create a file has two arguments. They are ‘symbolic_filename’ and ‘dos_filename’. The symbolic_filename is the name used by Turbo Prolog and dos_filename is the file used by DOS operating system, which must be put within quotes. The predicate used in “writing to a file” directs the output to the file identified by the ‘symbolic_filename’ argument. The predicate used in “using a file for writing” writes (outputs) to a file or device. Once a file is opened with openwrite, all further output goes to that file. The write predicate uses an argument called symbol, which may be any one of the following:

1. A string like “Chi-square test”
2. Value that is Name or Method, which writes the value bound to the variable Name or Method like kruskal-wallis method, paasche method, and
Table 3.8: File Types and Purposes in Turbo Prolog

<table>
<thead>
<tr>
<th>Predicate name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Predefined files:</strong></td>
<td></td>
</tr>
<tr>
<td>readdevice(symbolic_filename)</td>
<td>Opening a statistical file for input</td>
</tr>
<tr>
<td>wriedevic0(symbolic_filename)</td>
<td>Opening a statistical file for output</td>
</tr>
<tr>
<td><strong>User-defined output files:</strong></td>
<td></td>
</tr>
<tr>
<td>openwrite(symbolic_filename,&quot;dos_name&quot;)</td>
<td>Creating a file</td>
</tr>
<tr>
<td>wriedevic0(symbolic_filename)</td>
<td>Writing to a file</td>
</tr>
<tr>
<td>write(symbol)</td>
<td>To use a file for writing</td>
</tr>
<tr>
<td>closefile(symbolic_filename)</td>
<td>To close a file</td>
</tr>
</tbody>
</table>

3. symbol that is a symbol like technique.

The predicate used in 'closing a file' enables the user to close the symbolic name of the file to be closed.

In this segment we explain how to:

1. create a statistical knowledge file
2. open the file
3. write statistical facts and rules or information to the file
4. close the file

Let us assume the DOS file name is SMETHOD.TXT:

The symbolic file name in Turbo Prolog is FILE1.

Given below is a rule called use_file that accomplishes the four steps to create a file in a Turbo Prolog program. For statistical knowledge base maintenance, the rule has four subgoals:

- kbm_subgoal_1 to create a statistical knowledge base file
- kbm_subgoal_2 opens the file
- kbm_subgoal_3 writes some data to the file
- kbm_subgoal_4 closes the file
The `kbm_subgoal-1` creates a DOS file called `SMETHOD.TXT`. Turbo Prolog refers to this file by its symbolic name `FILE1`.

The `kbm_subgoal-2` predicate opens the file with the symbolic name `FILE1` so that information may be written to it. This sets the file `FILE1` as the output file. All further `write` predicates will send their information to this file until it is closed.

The `kbm_subgoal-3` is used to write some statistical information to the file. We will create a new predicate. This predicate will be defined later.

```prolog
write_some_info
```

This predicate will write symbols to the statistical knowledge base file. The `kbm_subgoal-4` is written to close the file named `FILE1`.

```prolog
close_file(file1)
```

Add `use_file` predicate to the clauses segment as shown below:

```prolog
The use_file predicate is

```prolog
clauses
use_file :-
    openwrite(file1,"SMETHOD.TXT") and
    writedevice(file1) and
    write_some_info and
    close_file(file1).
```

The definition of `write_some_info` looks like this:

```prolog
write_some_info :-
    statistical_method( Name ) and
    write( Name ) and
    flush(file1) and
    nl and fail.
```

The `fail` predicate is used for a new line and backtrack. We add some objects to the statistical knowledge base using the predicate `statistical_method` like

```prolog
statistical_method(kruskal-wallis).
statistical_method(paascheindex).
statistical_method(chi_square).
```
Figure 3.7 is a program segment of the clauses segment for statistical information management.

/* Begin of Program */

clauses

use_file :-
    openwrite(file1, "SMETHOD.TXT") and
    writedevice(file1) and
    write_some_info and
    closefile(file1).

write_some_info :-
    statistical_method(Name) and write (Name) and
    flush(file1) and
    nl and fail.

statistical_method(kruskal-wallis).
statistical_method(paascheindex).
statistical_method(chi_square).
compares_level_of_prices(pricesindex).
over_all_prices_changes(pricesindex).
measures_quality_of_a_variable_changes_overtime(quantity_index).
measures_changes_in_total_monetarywork(valueindex).
known(current_year_quantity).
known(base_year_quantity).
known(current_year_prices).
known(base_year_prices).
known(quantities_from_a_representative_year).
known(number_of_elements_used_in_index).
known(quantities_and_prices_that_determine_values_we_use_for_weights).

Figure 3.7: The Clauses Segment for Statistical Problems for Information Management

Of course, before we can run the program, we must declare the predicates statistical_method, use_file, write_some_info and known, in the predicate segment. The predicate segment is shown in Figure 3.8. We add this to our program.

predicates

use_file
write_some_info
statistical_method(name)

Figure 3.8: Predicates Segment for Statistical Problem

Neither use_file nor write_some_info requires any arguments. However, we must declare the data type of name in the domain segment. In addition to declaring name as a symbol, we also need to tell Turbo Prolog
that the object name File1 refers to a file. Figure 3.9 shows the domains segment. We add this to our program as well.

```prolog
domains
  file=File1
  name=symbol

Figure 3.9: Domains Segment of Statistical Problem
```

Turbo Prolog allows only one file declaration in the domains segment of the SESYS. However, more than one symbolic file name may be declared by separating them with a semicolon. If we have more than one file to declare, they would be written as shown below, where

```
file1; file2; file3…….would be our filenames (Cafolla, 1989).
```

The complete program, including the domains, predicates and clauses segments, is shown in Figure 3.10.

```
/* Begin of Program */

domains
  file=File1
  name=symbol

predicates
  use_file
  write_some_info
  statistical_method(name)

clauses
  use_file :-
    openwrite(file1,"SMETHOD.TXT") and
    writedevice(file1) and
    write_some_info and
    closefile(file1).

write_some_info :-
  statistical_method(Name) and write(Name) and
  flush(file1) and
  nl and fail.

statistical_method(kurskal-wallis).
statistical_method(pascheindex).
statistical_method(chi_square).
compares_level_of_prices(pricesindex).
over_all_prices_changes(pricesindex).
measures_quality_of_a_variable_changes_overtime(quantity_index).
measures_changes_in_total_monetarywork(valueindex).
known(current_year_quantity).
known(base_year_quantity).
known(current_year_prices).
known(base_year_prices).
known(quantities_from_a_representative_year).
known(number_of_elements_used_in_index).
known(quantities_and_prices_that_determine_values_we_use_for_weights).
```

Figure 3.10: Statistical Knowledge Base
We run the program. If there are no errors, type the following goal.

\[ \text{goal : use\_file.<return>} \]

Save the File currently in the Editor Window

Return to the Main Menu \hspace{1cm} [ESC]
Select the File Menu \hspace{1cm} [F]
Select **WRITE** to command \hspace{1cm} [W]

At this point, we will be asked to supply a filename to our program.

Type **INDEXNO.PRO <ENTER>**

The program currently in the Editor Window will be saved on the current directory C with the name **INDEXNO.PRO**. Similarly the other .PRO programs are written for other modules of statistical problems.

**Rule to Create Statistical Method File.** How to create a statistical method file is shown in Figure 3.11.

```
classes
  create_file :-
      openwrite(methods,"meth.txt") and
      get\_names and
      closefile(methods).
```

Figure 3.11: Rule to Create Statistical Method File

The rule get\_names is used to obtain input from the keyboard and writes it into a file. We declare the predicates create\_file and get\_names in the predicate segment as shown in Figure 3.12.

```
Predicates
  create\_file
  get\_names

clauses
  create\_file :-
      openwrite(methods,"meth.txt") and
      get\_names and
      closefile(methods).
```

Figure 3.12: Predicates Declaration for create\_file Rule.

The methods file must be declared in the domain segment. The file is declared with the symbolic filename **methods**. Add the domains declarations as shown in Figure 3.13.
domains
   file=methods
predicates
   create_file
   get_names
clauses
   create_file :-
      opnwrite(methods,"meth.txt") and
      get_names and
      closefile(methods).

Figure 3.13: Domains Declaration for Methods File

The get_names sets readdevice to the keyboard and reads in the user.s response. It then calls another rule and actually writes the input to the file. The get_names rule is as shown in Figure 3.14.

get_names :-
   wrihtdevicr(screen) and
   write("Enter Name or <RETURN> to Exit :") and
   readdevicr(keyboard) and
   readln(Name) and
   write_to_file(Name).

Figure 3.14: Predicates to get_names and Write_to_File

The first subgoal of this rule sets the write device to the screen. Although the screen is the default write device, the get_names rule calls another rule write_to_file that changes the write device. Therefore, it is necessary to reset the proper input device. The new rule write_to_file has two jobs. The first is to check the input for the blank character. The user generates this character by pressing the [RETURN] key. This is the way the user signals that all the details regarding a statistical method have been entered. This is the exit condition. The rule that accomplishes this is

write_to_file("") :- !.

If we enter a blank carriage return, the cut (!) is encountered. Thus, both write_to_file and its calling routine get_names complete their tasks.

The next rule tells Turbo Prolog what to do if the user enters a method (some details other than a carriage return). Now, we enter these new predicates and the domains and predicates segment as shown in Figure 3.15.
/* Begin of Program */

domains
  file=methods
  name=symbol

predicates
  create_file
  get_names
  write_to_file(name)

clauses
  create_file :-
      openwrite(methods,"meth.txt") and
      close_file(methods).

  get_names :-
      wrietedevice(screen) and
      write("enter name or <RETURN> to Exit :") and
      readdevice(keyboard) and
      readln(Name) and
      write_to_file(Name).

  write-to-file(""") :- !

  write_to_file(Name) :-
      wrietedevice(methods) and
      write(Name,"","" and
      get_names.

/* End of Program */

Figure 3.15: write to file Predicate

Statistical Knowledge Base File: Read and Print Rules. Reading the statistical knowledge base file requires two rule predicates. The first rule is read_file. This rule opens the "meth.txt" file and directs Turbo Prolog to obtain input from (read) the meth.txt file. Later, we will define a predicate called print_file to write the file on the screen. The read_file predicates is shown below.

read_file :-
    openread(methods,"meth.txt") and
    readdevice(methods) and
    clearwindow and
    print_file and
    close_file(methods).

Once we open a file for reading, we want the details in it to be displayed or printed on the screen. Two rules of print_file handle this. The first print_file rule tells Turbo Prolog what to do if the end of file has not been encountered. The second print_file rule tells Turbo Prolog what to do
when it finds itself at the end of the file. The first print_file rule is as follows:

```prolog
print_file :-
    not(eof(methods)) and
    readchar(Letter) and
    write(Letter) and
    print_file.
```

The first clause of this rule fails when the next character is the end of file marker. This clause of Turbo Prolog helps to move down the statistical knowledge base to the second print_file rule. The second print_file rule tells Turbo Prolog what to do when the end of the file is encountered. This refers to the boundary condition for the print_file rules. This predicate resets the read device to the keyboard and the write device to the screen. It tells us that it is finished, waits for a key to be pressed and then closes the file.

```prolog
print_file :-
    nl and
    readdevice(keyboard) and
    wriitdevice(screen) and
    write("press space bar") and
    readchar(_).
```

To use these predicates, we add them to the clauses segment as shown below. We also add the new predicates to the predicate segment. The new segments are shown in Figure 3.16.

```prolog
/* Begin of Program */

domains
    file=methods
    name=symbol
predicates
    create_file
    get_names
    write_to_file(Name)
    read_file
    print_file
clauses
    create_file :-
        openwrite (methods,"meth.txt") and
        get_names and
        closefile(methods).

    get_names :-
        writedevice(screen) and
        write("Enter Name or <RETURN> to Exit :") and
```
readdevice(keyboard) and
return(Name) and
write_to_file(Name).

write-to-file("") :- !

write_to_file(Name) :-
readdevice(methods) and
write(Name","","") and
get_names.

read_file :-
openread(methods,"meth.txt") and
readdevice(methods) and
clearwindow and
print_file and
closefile(methods).

print_file :-
not(eof(methods)) and
readchar(Letter) and
write(Letter) and
print_file.

print_file :-
nl and
readdevice(keyboard) and
write(device(screen)) and
write("press space bar") and
readchar(_).
/*End of Program*/

Figure 3.16: The read_file and print_file Predicates of SESYS.

Our program can now accept the names of statistical methods and their details. Also it can save those methods and details to a disk file and display the contents on the screen, if required.

Adding Statistical Data to an Existing Statistical Knowledge Base File. It is necessary to create a new rule, which we call append_file. This rule will open the statistical knowledge base file for appending and preserving the original contents of the file. The append_file rule is the same as create_file except that it opens the file for appending. It uses the standard predicates openappend to open the statistical knowledge base file. We will still use the get_names rule for processing the keyboard input.

append_file :-
openappend(methods."meth.txt") and
get_names and
closefile(methods).
When we open a statistical knowledge base file for writing with `openappend`, Turbo Prolog assumes it is an existing file. This preserves the content of the statistical knowledge base file. In order for these predicates to succeed, the statistical knowledge base file must exist on the disk prior to opening it with `openappend`.

`append_file` predicate must be included in the predicates segment.

### 3.11 Adding a Menu to SESYS Program

Now we will add a user-friendly menu to SESYS program. This will make it easy for the non-expert user to use the SESYS with ease. The only knowledge the non-expert requires is how to codify the statistical knowledge into the form that Turbo Prolog can understand.

The first new rule is called `menu`. This predicate will display a menu that tells the user what the program does. The routine asks the user to input a choice. When the user enters a choice, the `goAhead` predicate processes it. The `repeat` predicate (line 1) is explained below:

```prolog
/* Begin of Program */

menu :-
    repeat and
    clearwindow and
    nl and nl and nl and
    write("1. Create SKB file \n") and
    write("2. Append SKB file \n") and
    write("3. Read SKB file \n") and
    write("4. Exit") and
    nl and nl and nl and
    write("Enter your choice :") and
    readint(Choice) and
    goAhead(Choice).
```

We want the menu to reappear after each task. To do this, we use a loop structure. Unfortunately, Turbo Prolog does not have such a structure. Therefore, we must use a special technique to create a loop using recursion. The rule `repeat` is as follows:

```prolog
repeat.
repeat :-
    repeat.
```
These two rules of repeat use recursion to force the entire menu rule to be repeated until the user chooses menu item 4 to exit. Add the menu and repeat rules to the clauses segment and declare them in the predicates segment as shown in Figure 3.17.

/* Begin of Program */

domains
    file=methods
    name=symbol
predicates
    create_file
    get_names
    write_to_file(Name)
    read_file
    print_file
    append_file
    menu
    repeat
clauses
    
    menu :-
        repeat and
        clearwindow and
        nl and nl and nl and
        write(" 1. Create SKB file \n") and
        write(" 2. Append SKB file \n") and
        write(" 3. Read SKB file \n") and
        write(" 4. Exit") and
        nl and nl and nl and
        write(" Enter your choice :") and
        readint(Choice) and
        go-ahead(Choice).

repeat.
repeat :- repeat.

/* End of Program */

Figure 3.17: The menu and repeat Rules in SESYS

The go-ahead predicate processes the user's choice. It calls up the proper rules requested by the user. The rules are shown below:

    go-ahead(1) :-
        create_file.
    go-ahead(2) :-
        append_file.
    go-ahead(3) :-
        read_file.
    go-ahead(4) :-
        exit.
    go-ahead(X) :-
\( X \geq 1 \) \text{ and } \text{error}.

\text{go}_\text{ahead}(X) :-
\quad X \geq 1 \text{ and } \text{error}.

Notice the \textit{exit} predicate in choice 4. This is a new standard predicate that succeeds by ending the program. The user must be informed of the error and error rule may be as follows:

\begin{verbatim}
\text{error} :-
    \text{write(\text{"Choice not valid."}) and write(\text{"Try again.\n"}) and write(\text{"Press space bar"}) and readchar(_) and menu.}
\end{verbatim}

We add the \textit{go}_\text{ahead} and \textit{error} rules to the clauses and predicates segments of the SESYS program. We must also declare the argument \textit{choice} in the domain segment. This argument is of the domain type \texttt{integer}. The predicates \textit{create}_\text{file}, \textit{read}_\text{file} and \textit{append}_\text{file} must be modified so that when they completed their tasks, the user is returned to the \textit{menu}. These modifications are done just by adding the menu predicate at the end of each segment, with \texttt{and} operator. Before writing the statistical method module program, we will enter it with a goal. We will enter menu as the internal goal by adding the following predicates to the goal segment.

\begin{verbatim}
\text{goal}
    \text{clearwindow and menu}
\end{verbatim}

The clearwindow command clears the Dialog Window. The statistical knowledge base file creation program is shown in Figure 3.18. Indeed such statistical knowledge base files have been created and tested in respect of two earlier applications of the statistical expert system have been made by the researcher, namely, the Knowledge Based Statistical Software Program (KBSSP) for psychologists and the Statistical Expert System for Administrators (SESA).

/* Begin of Program */

\begin{verbatim}
domains
    file=file1
    name=symbol
\end{verbatim}
choice=integer

predicates
create_file
get_names
write_to_file(Name)
read_file
print_file
append_file
menu
repeat
goAhead(Choice)
error

goal

clearwindow and
menu.

clauses
menu :-
    repeat and
    clearwindow and
    nl and nl and
    write(" 1. Create SKB file \n") and
    write(" 2. Append SKB file \n") and
    write(" 3. Read SKB file \n") and
    write(" 4. Exit") and
    nl and nl and nl and
    write(" Enter your choice :") and
    readInt(Choice) and
    goAhead(Choice).

goAhead(1) :-
    create_file.

goAhead(2) :-
    append_file.

goAhead(3) :-
    read_file.

goAhead(4) :-
    exit.

goAhead(X) :-
    X>4 and error.

goAhead(X) :-
    X>1 and error.

error :-
    write("Choice not valid.") and
    write("Try again\n") and
    write("Press space bar") and
    readchar(\_) and
    menu.

create_file :-
    openwrite (methods,"c:meth.txt") and
    get_names and
    closefile(methods) and
    menu.

get_names :-
    wriitedevice(screen) and
    write("Enter name or <RETURN> to Exit:") and
    readdevice(keyboard) and
    readln(Name) and
    write_to_file(Name).
Figure 3.18: A Sample Statistical Knowledge Base File of SESYS.

The two systems will be discussed in their implementation in Chapter IV, which concerns itself with the implementation of the SESYS. This means that the KBSSP and SESA could be, procedurally, integrated with the newly developed SESYS, which could in turn be used for information management.

3.12 Compiling and Linking the SESYS Modules

With all of the modifications to the modules in the SESYS; we are ready to convert the SESYS into a standalone program that is executable from the DOS prompt level.
First, select the SESYS (all modules) option from the Options Menu. When asked for the name of the SESYS file, type SESYS.

Next, type ‘C’ to instruct Turbo Prolog to compile the SESYS. There is a considerable amount of disk activity, much of it reflected in displays in the Message Window. When the program has finished compiling and linking, we will see the question *Execute (y/n) ?* in the Message Window. If we want to run the program now, type y. If we want to save the program for later execution, type n.

To run this program later, just get to the DOS prompt with the proper path specified and type SESYS.

3.13 Summary

The present chapter is a critical chapter in the entire thesis in that it is here that the proposed model for the Statistical Expert System for Information Management is spelt out, threadbare, in relation to developing the SESYS. There has been an effective attempt at using the Turbo Prolog Programming for developing the SESYS. As such, the components of the SESYS, the user and system interfaces, functional design, steps in modular programming of the expert system and so on have been discussed.

As Turbo Prolog is a declarative language, it facilitates the representation of knowledge in the form of production rules. It is eminently suitable for writing the SESYS syntax and semantics, even while it allows for mechanisms such as recursion, instantiation, verification, unification, backtracking and interchangeability.

Along the way, however, the SESYS consultation paradigm has been introduced and elaborated, and the use of WINDOWS, dynamic databases, and dynamic database predicates have been discussed and codes written for the segments. Then comes the SESYS User Interface that has been developed for a line-oriented screen, but its operation is not as convenient as that of other shells. The SESYS functional design, however, has been
designed with the goal of providing a simple SESYS but quite interesting in conception and use. Its functional design components have then been elaborated, leading to the steps in SESYS modular programming. The six steps have been illustratively programmed and by segments. Compiling and linking SESYS modules have been discussed and an example combining the six modules of SESYS into a single program has been written.

Statistical interface mechanism, its explanation component, the SESYS user interface, statistical logic machine, including Turbo Prolog and logic, rules for its clauses and various segments of SESYS program have been shown. There are several advantages of using the SESYS: it makes the users’ job easier; it makes possible applications through statistical packages in use as well as through SESYS; it updates statistical knowledge from the users’ expertise; and it allows C routines to be used in interfacing / computing and updating. The SESYS has advantages for the untrained users, as well: it advises the best way, help the user improve and update statistical knowledge, besides permitting all other advantages mentioned above for the trained users.

In the next chapter, the discussion turns to the implementation of the SESYS and some examples of programs are written and shown how they may be run and used.